K22 – Operating Systems: Design Principles and Internals

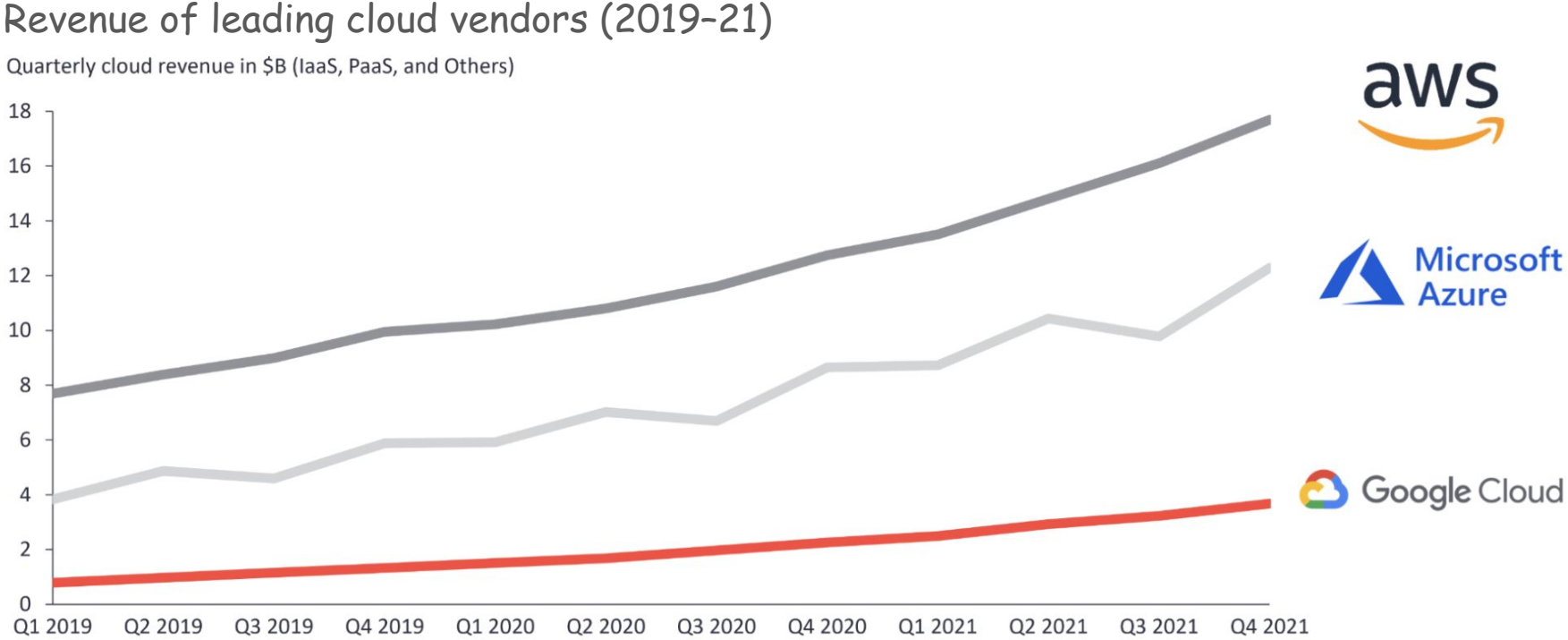
Fall 2025 @dit

Vaggelis Atlidakis

std05010

References: Similar OS courses @Columbia, @Stanford, @UC San Diego, @Brown, @di (previous years); and textbooks: Operating Systems: Three Easy Pieces, Operating Systems: Principles and Practice, Operating System Concepts, Linux Kernel Development, Understanding the Linux Kernel

Why study OSes?



[1] https://iot-analytics.com/cloud-market/

Why study OSes?

- **These days:** Code developed with superficial understanding
  - No free lunch: Bugs won't fix themselves in production

- **In the near future:** Requirement for software engineers with legit systems understanding will soon be at an all-time high
  - A line of code is not just a line of code
  - Myriads of things happen under the hood

- Learning to navigate a complex codebase, such as an OS kernel, will make you a **fearless** software engineer!

What you really need to do?

- Come to class, ask questions, be curious
- Choose a team you can collaborate well with
- Own your code
  - You know what you must not do…
  - This is not a class you pass by outsourcing your work
  - There is no magic: You do the work ⇒ You pass the class
  - GenAI usage: Use LLMs to help you understand

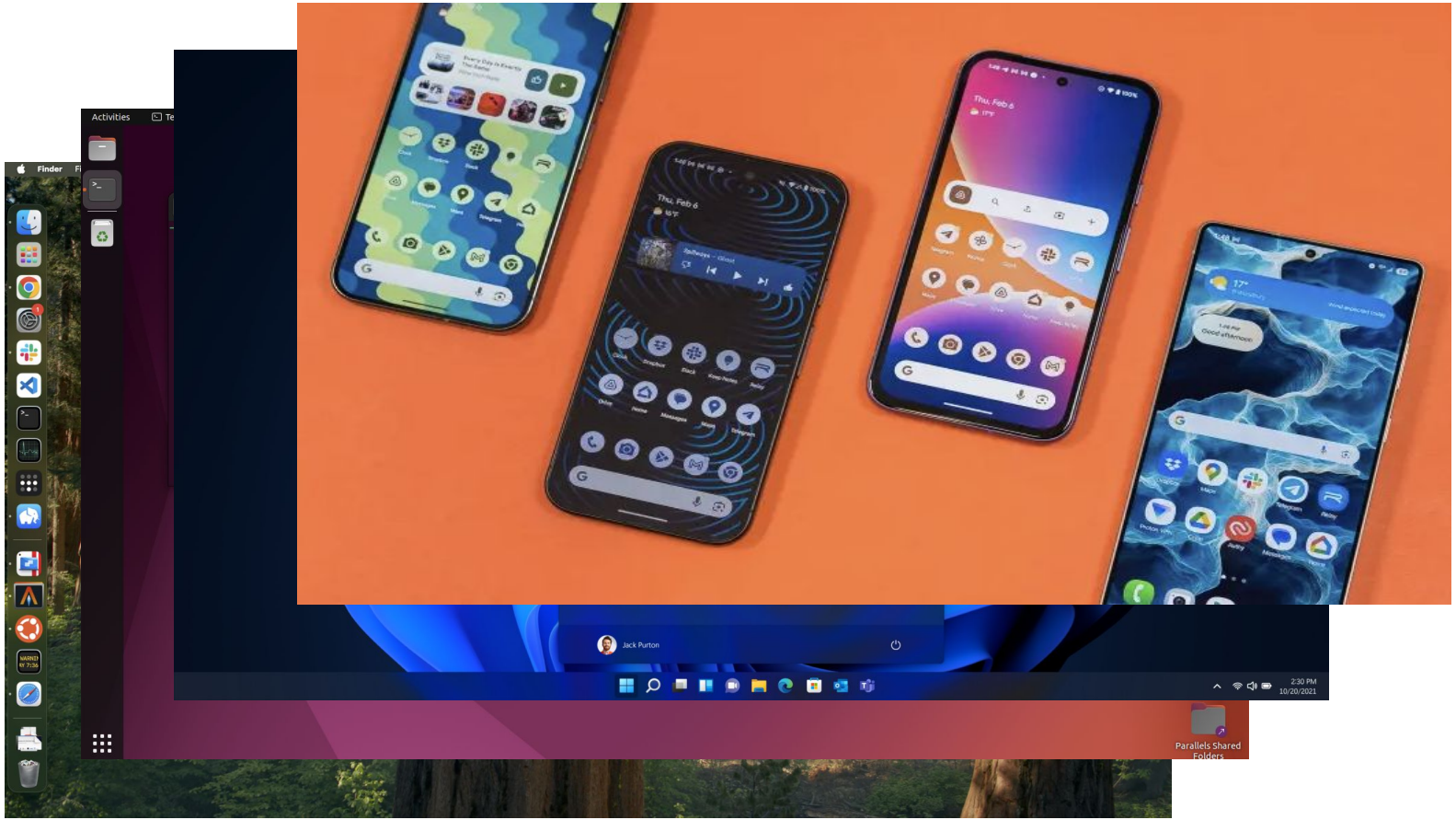
- It's a difficult course. Hang in there, we will help…

Overview

- We'll start from hardware and follow a question-oriented approach
  - Intro [Q: What is an OS?]
  - Events [Q: When does the OS run?]
  - Runtime [Q: How does a program look like in memory?]
  - Processes [Q: What is a process?]
  - IPC [Q: How do processes communicate?]
  - Threads [Q: What is a thread?]
  - Synchronization [Q: What goes wrong w/o synchronization?]
  - Time Management [Q: What is scheduling?]
  - Memory Management [Q: What is virtual memory?]
  - Files [Q: What is a file descriptor?]
  - Storage Management [Q: How do we allocate disk space to files?]

* Basic (H/W & S/W)
* **Abstractions**
* **Primitives**
* **Mechanisms**

What is an OS?



What is an OS?

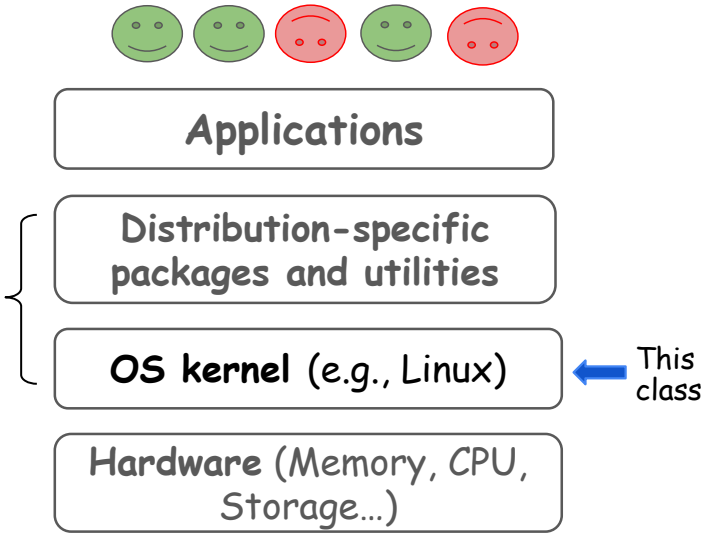
- A layer of abstraction that translates the hardware of a machine into (standardized) software concepts that applications use

Why need an OS?

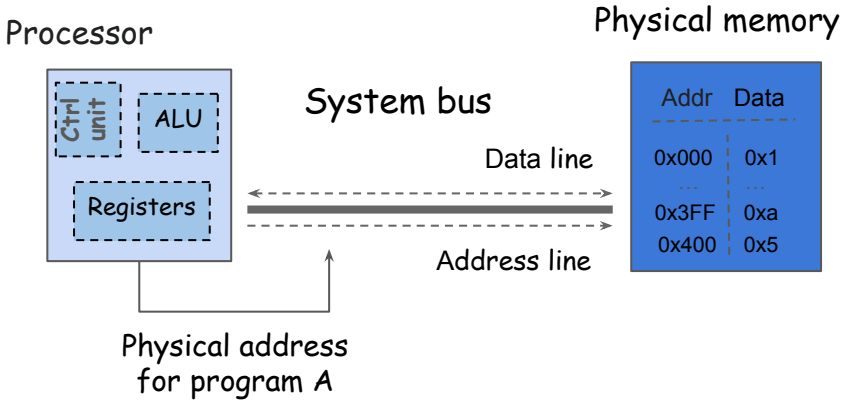
- Multiple users not necessarily sane, build and run different applications, at the same time on shared hardware (design assumptions)

Desirable properties?

- ***Security***: Does the system behave as expected in adversarial contexts?
- ***Reliability***: Does the system behave as expected given benign failures?
- ***Portability***: Is it easy for developers to build and maintain apps?
- ***Fairness***: Is the system "pleasant" to use?



What is the hardware we build OSes for?



- **Physical Memory (PM)**
  - Stores data addressable on a byte granularity
- **Processor**
  - Reads data from memory to its registers
  - Performs computations
  - Writes the data from its registers to memory
- **System bus**
  - Memory and processor communication channel
- Is this model satisfactory?
- Reminds you of something?

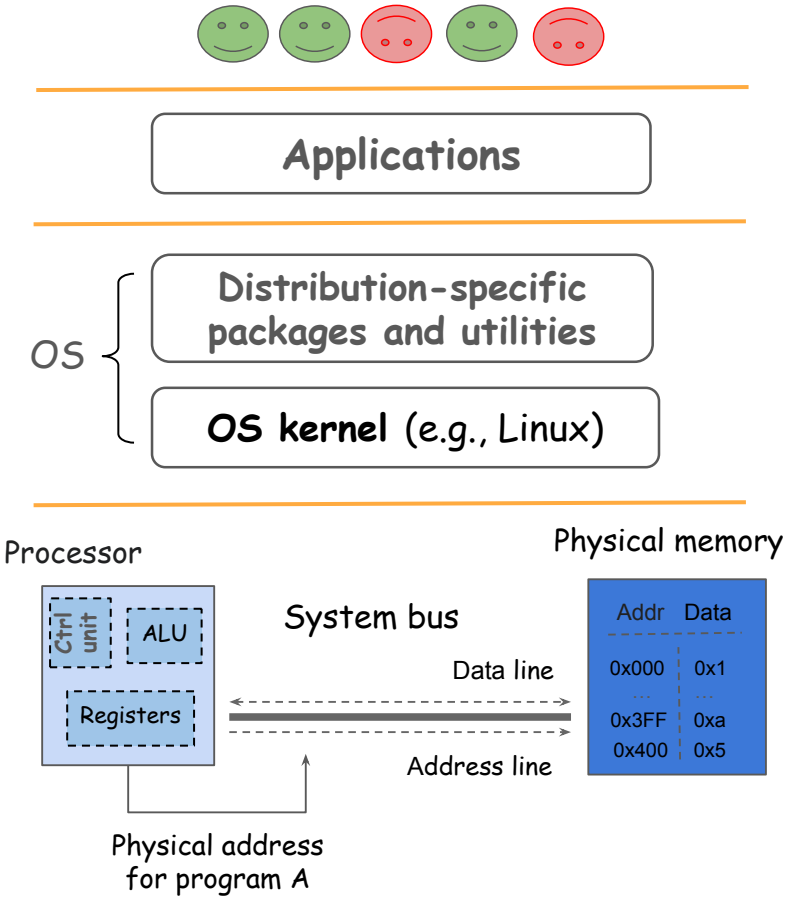
Designing an OS kernel

Desirable properties

1) Security, 2) Reliability, 3) Portability, 4) Fairness

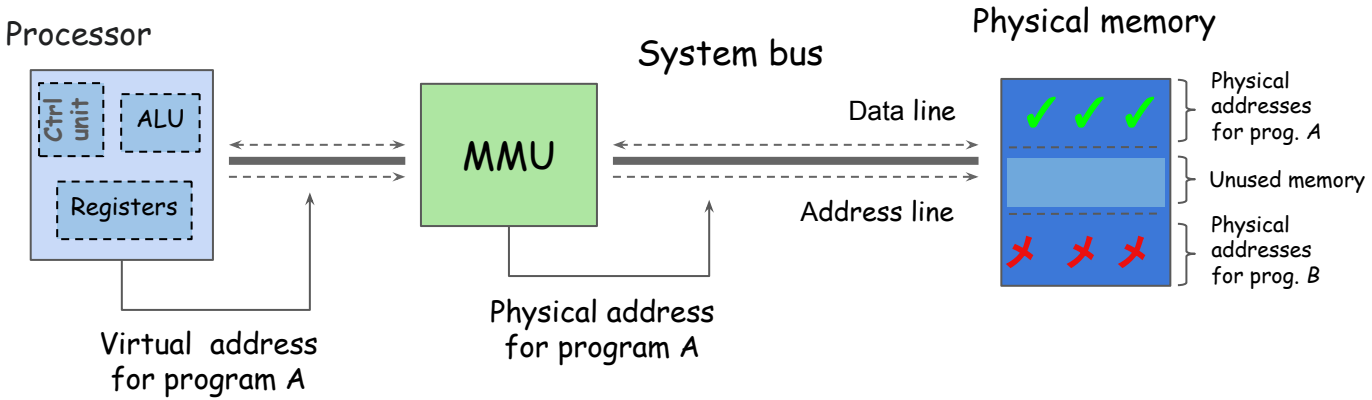
From desirable properties to design principles

- ***Fault Isolation***: Errors in one running program do not affect any other program
- ***Principle of Least Privilege (PoLP)***: Any program has the minimum privileges necessary to perform its function
- ***Preemption***: The OS is always able to take control of the processor, regardless of what programs are running



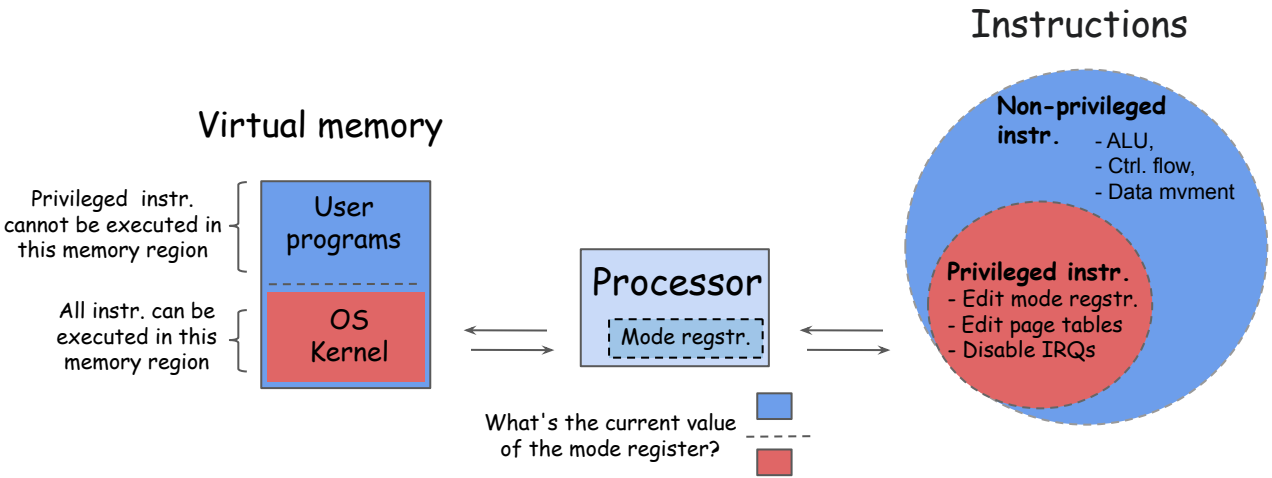
Implementing fault isolation

- In simple words: load/ store/ jmp instructions of a program cannot read, write, or jump to another program's memory



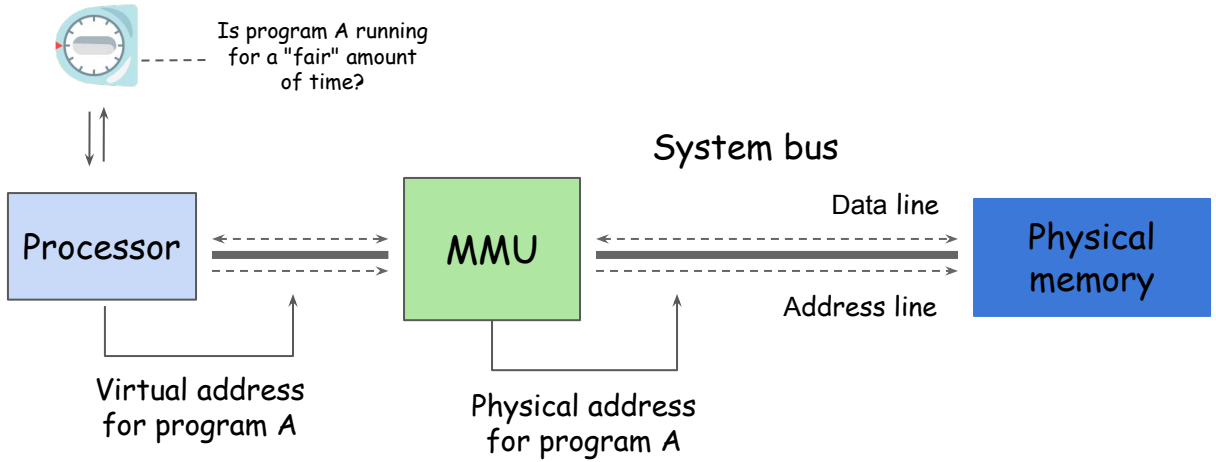
Implementing dual-mode execution

- In simple words: A trusted portion of the code (the OS) must have full control of the hardware

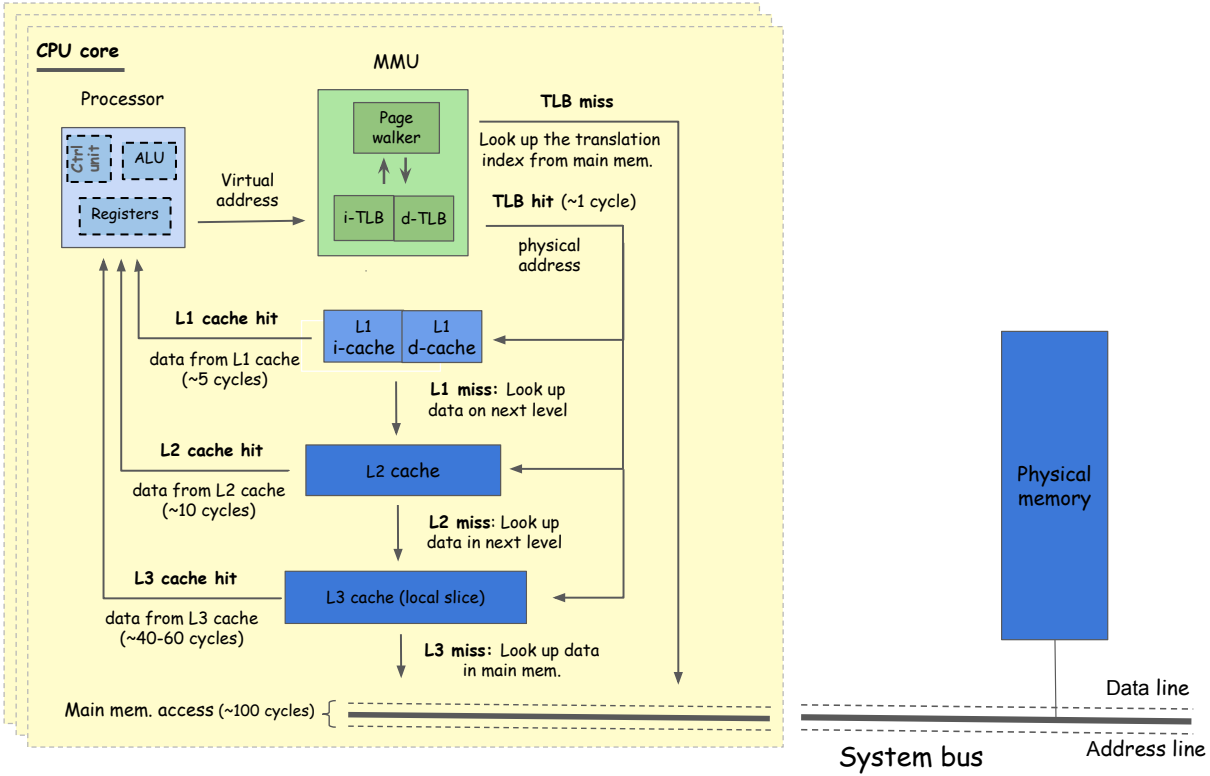


Implementing preemption

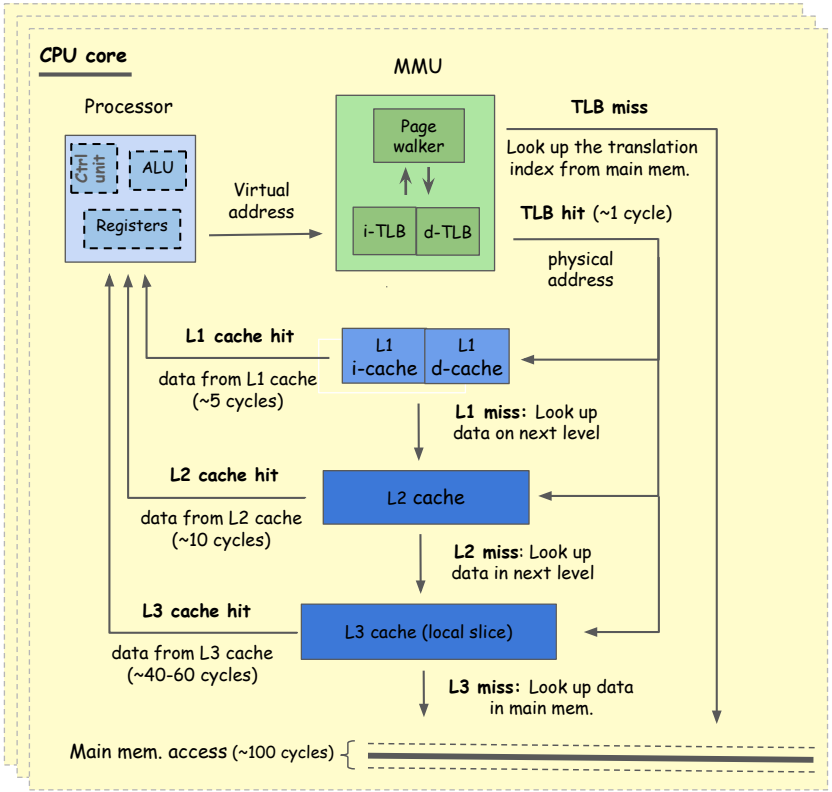
- In simple words: The OS should be able to periodically gain control of the processor regardless of what programs are executing

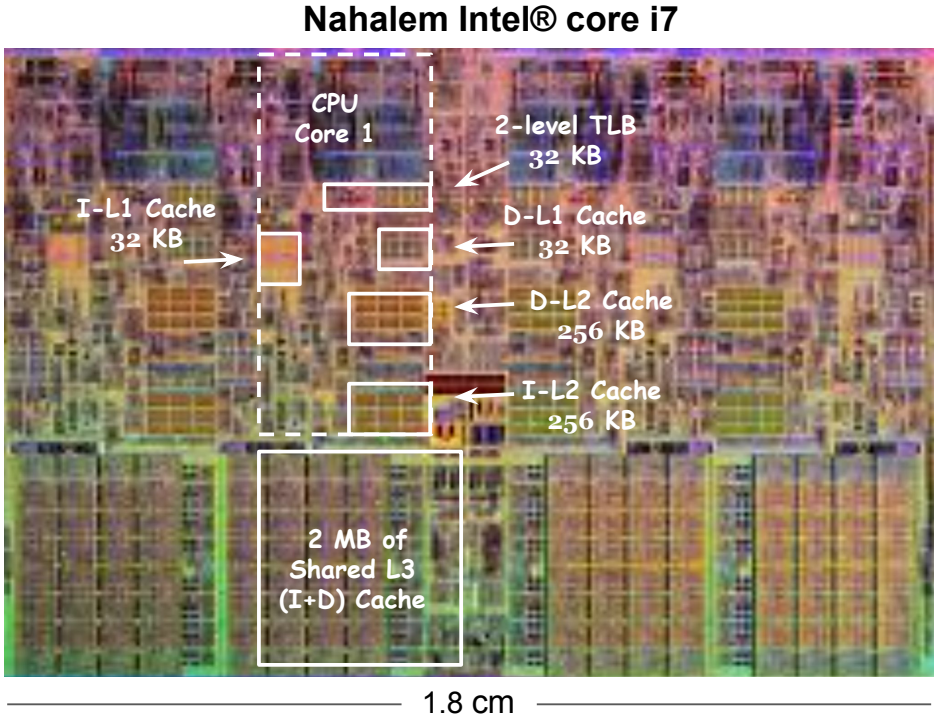


Hardware components leveraged by OS mechanisms

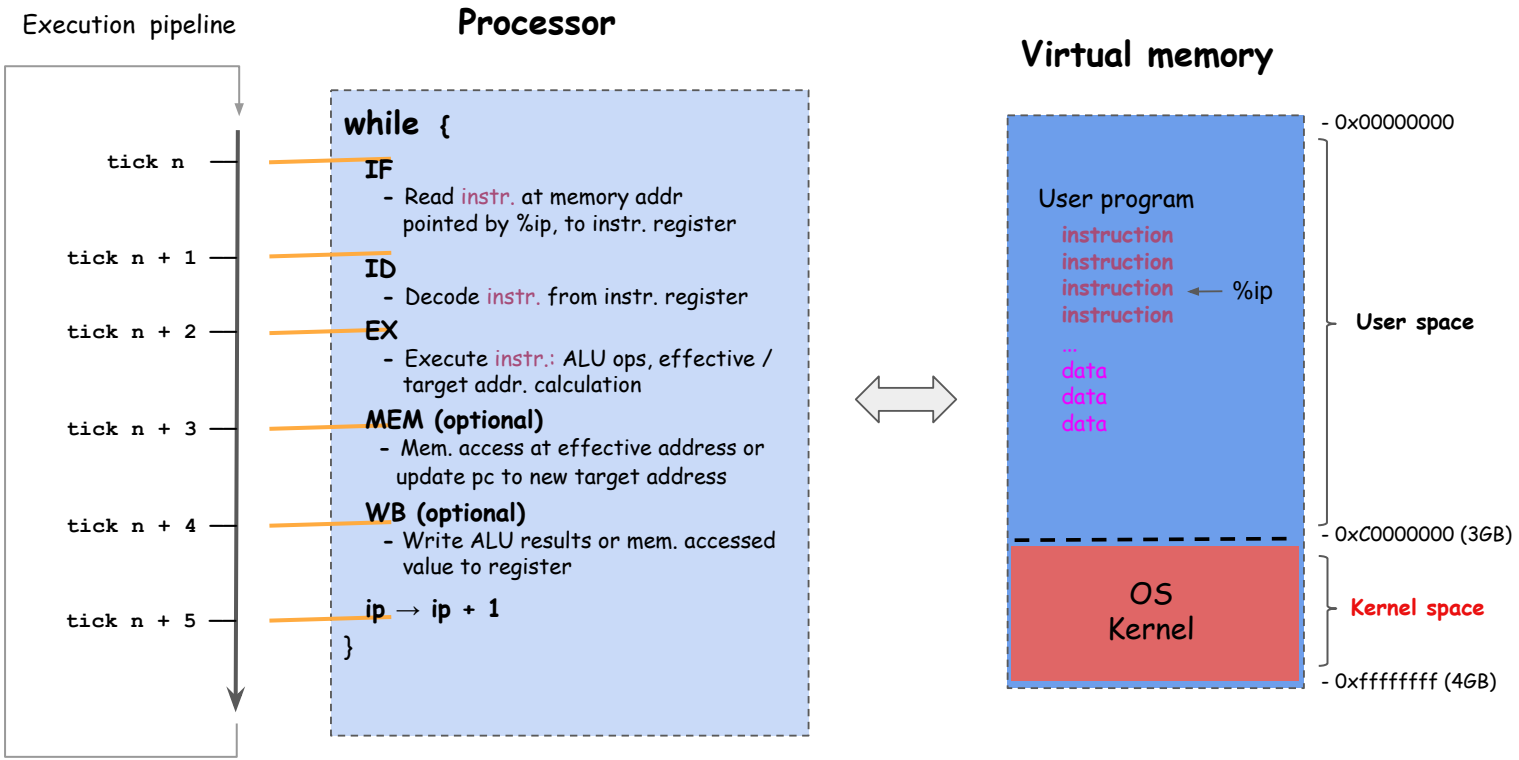


Reality check on hardware's scale and access times

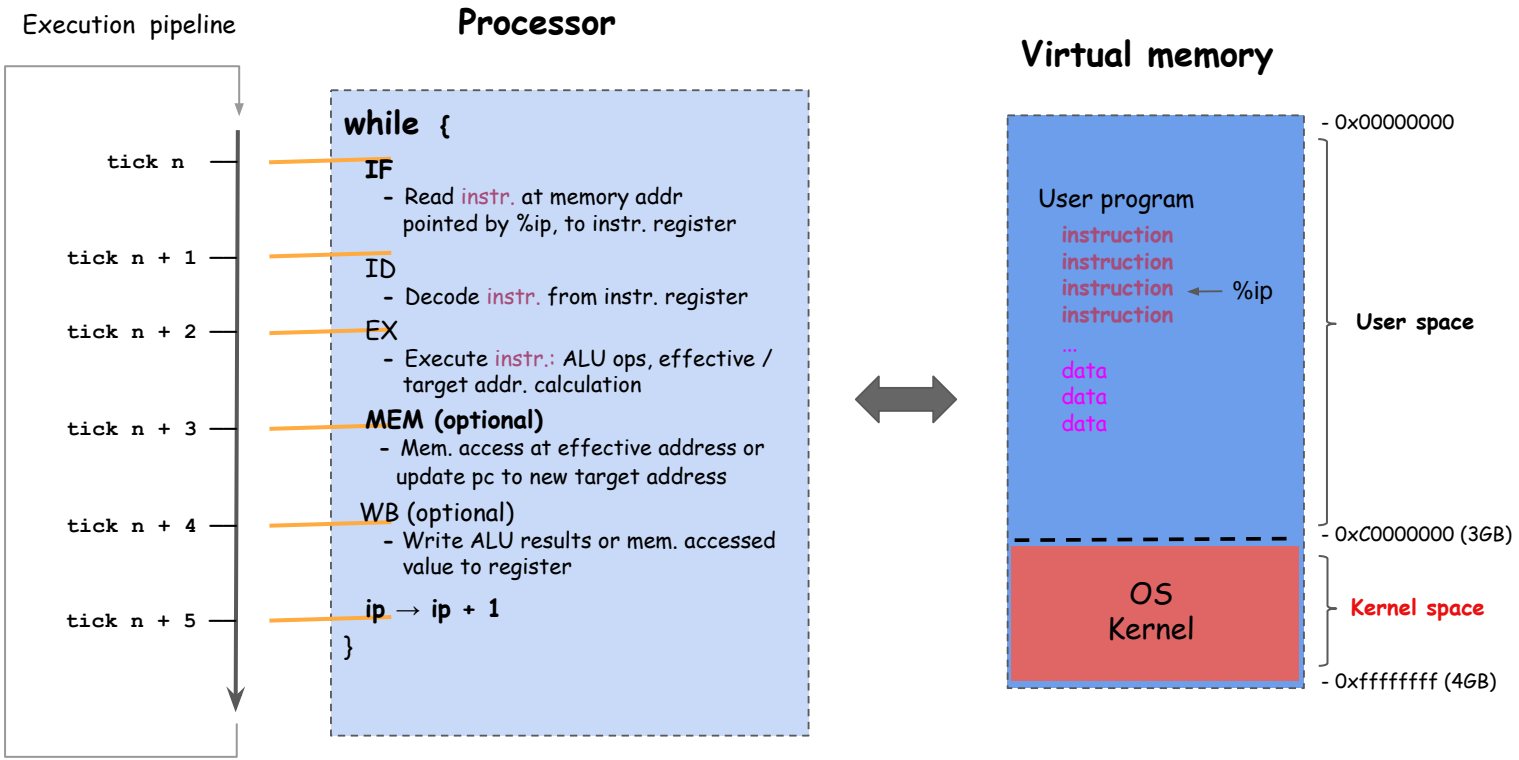




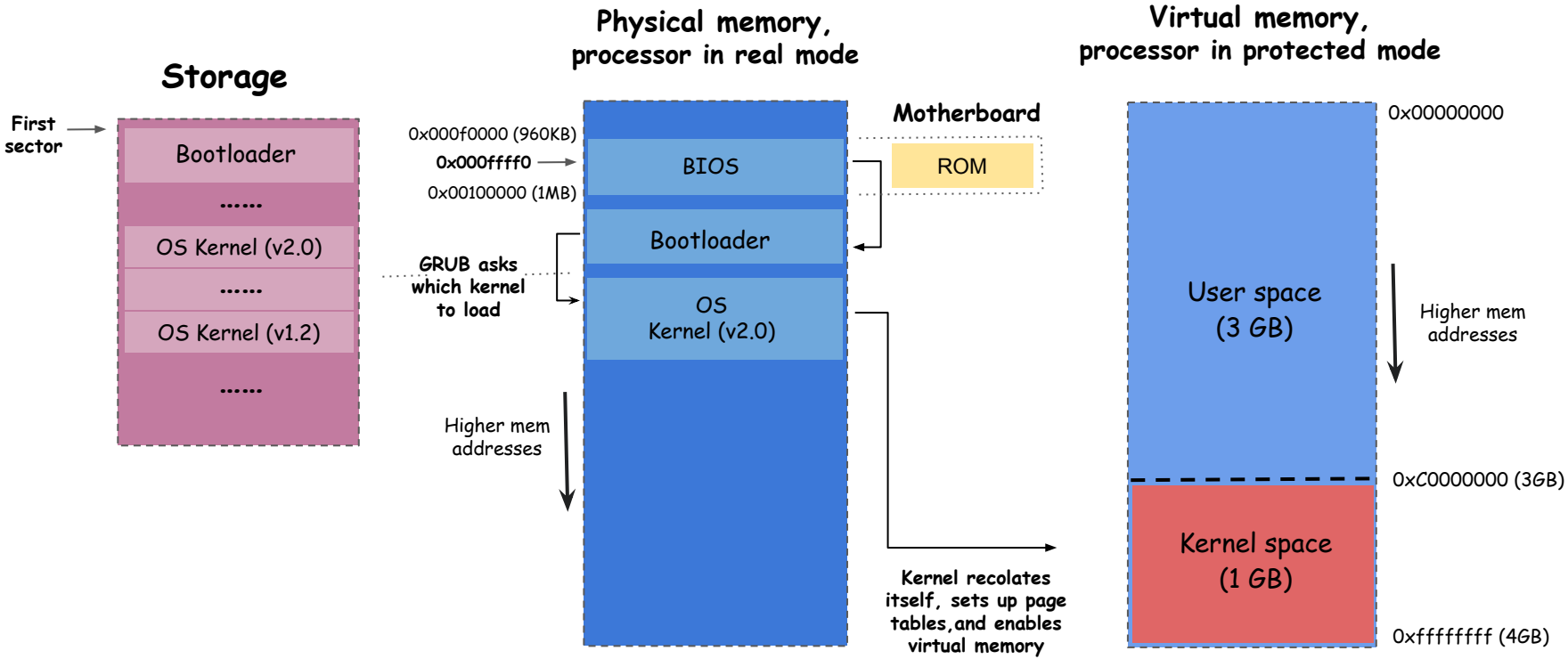
How does the processor execute programs?



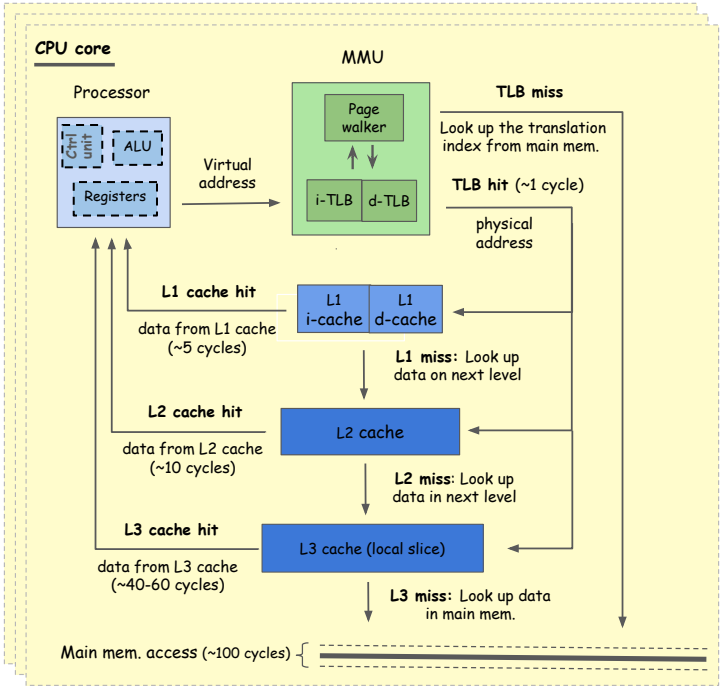
When may the processor access memory?



How does the system boot?

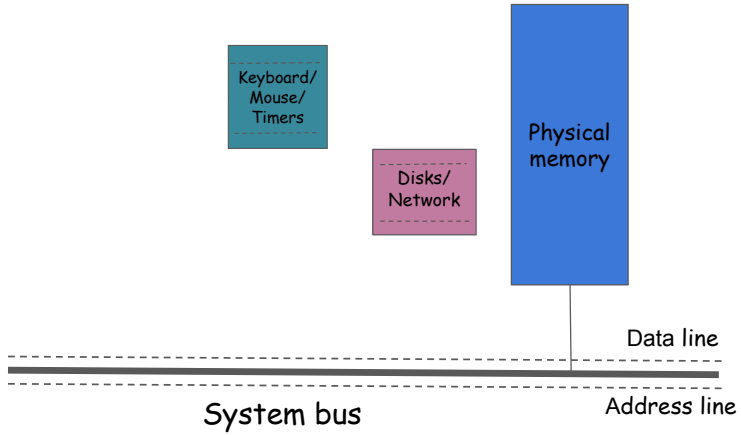


Handling I/O devices

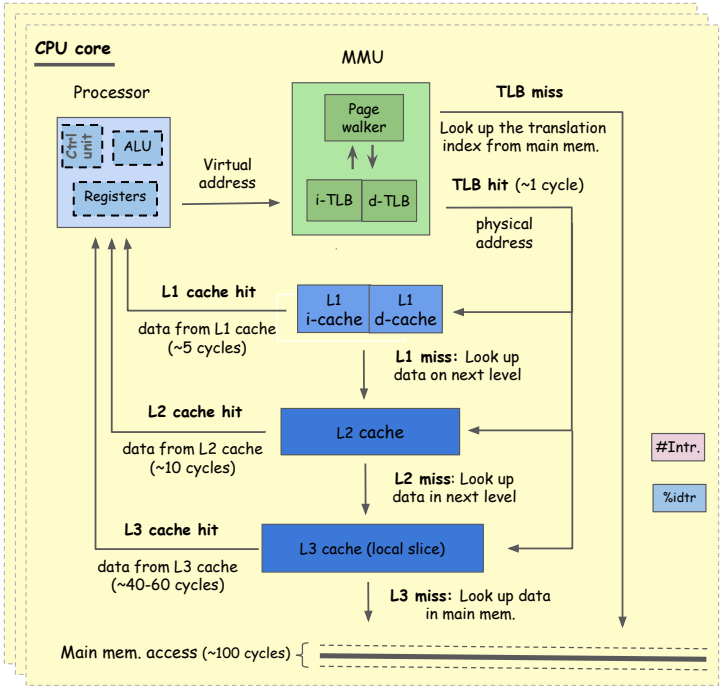


- What are we missing in terms of hardware?
  - Keyboard? Mouse?
  - Disk? Network?

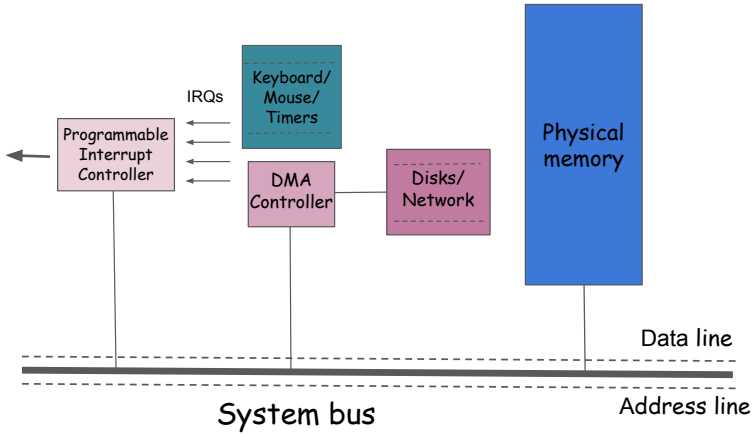
  } I/O devices



Handling I/O devices



- I/O devices
  - Need a way to get processor's attention
  - Interrupts, Interrupt Requests (IRQs), Direct Memory Access (DMA) controler...

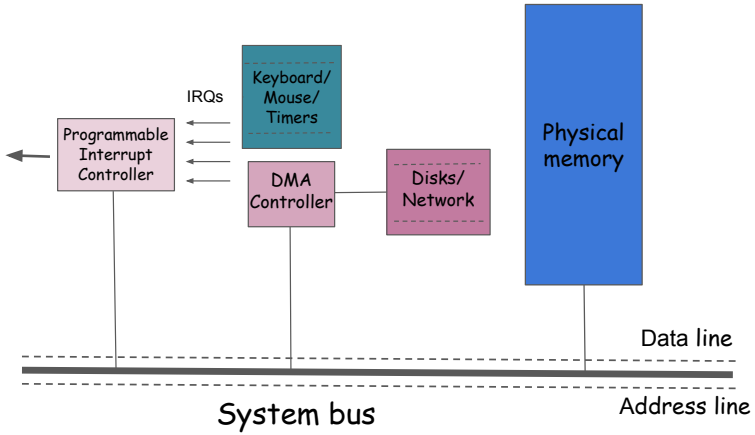


Handling I/O devices

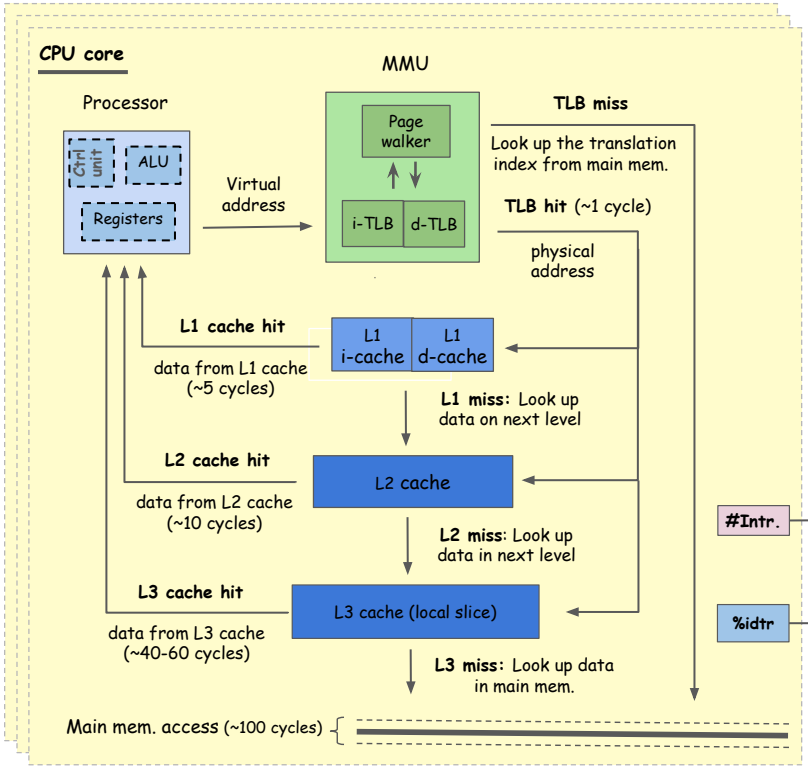
Execution pipeline

```
while {
    IF - Instruction Fetch
    ID - Instruction Decode
    EXE - Execute Instruction
    MEM - Memory Access
    WB - Write back
    ip → ip + 1                          Instruction
    if (Interrupt) {                     boundary
        What??
    }
}
```

- I/O devices
  - Need a way to get processor's attention
  - Interrupts, Interrupt Requests (IRQs), Direct Memory Access (DMA) controler…

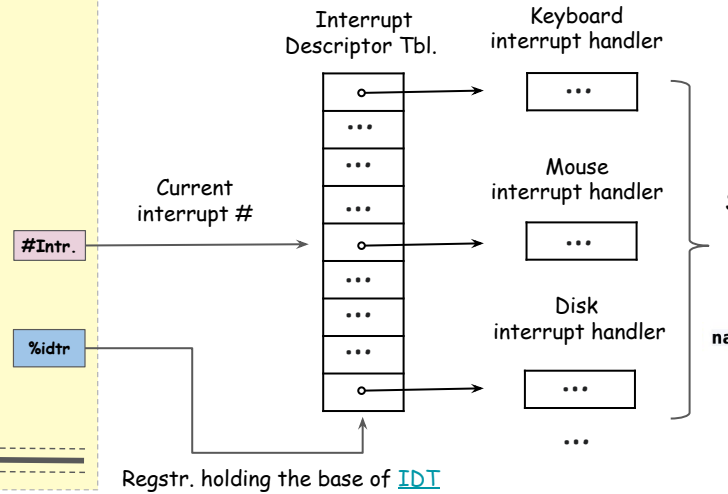


Interrupt handling



do_handle_interrupt()

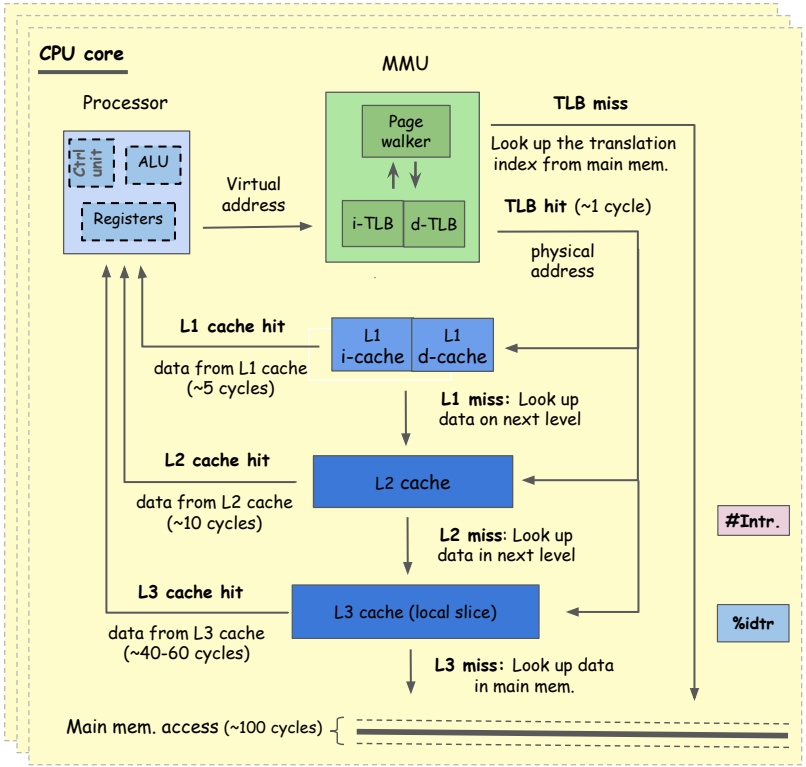
- **Switch to kernel stack, if in user mode**
- Save CPU registers and interrupt error code (if any)
- Use #intr. and %idtr to execute the appropriate handler
- Restore CPU context



Set up in kernel space during early boot see: init_IRQ()

```
native_load_idt( desc_ptr *dtr)
  asm volatile("lidt %0"::"m" (*dtr));
```

Interrupt handling (revised for clarity)



do_handle_interrupt()

- Switch stack
  - Kernel stack, if in user mode
  - Dedicated stack for critical exceptions
- Save CPU registers
- Save interrupt error code (if any)
- Use #intr. and %idtr to execute the appropriate interrupt handler
- Restore CPU context

When does the OS run?

The OS is a giant handler of events, which runs in response on two types of events

- ***Asynchronous events***: Events that occur due to reasons external to the program instructions that the processor was currently executing ⇒ Interrupts
- ***Synchronous events***: Events that occur synchronously as a result of the execution of a program's instructions

What synchronous events invoke the OS? (revised for clarity)

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
  - Does not break program continuity
  - Next instruction is the faulting instruction (restart)
  - The hope is that the OS will be able to "revert the mess" that caused it
  - Example: Page fault
- ***Aborts***: Exceptions used to report severe errors
  - Breaks program continuity
  - The processor may be unable report the precise instruction that caused it
  - Example: Double fault (invalid mem. access in a fault handler), machine check error
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction
  - Does not break program continuity
  - Next instruction is the one following the trapping instruction
  - The old way of making system calls!

X86 exceptions vector (addition)

Vector NR	Exception/Interrupt Name	Type
0	Divide Error	Fault
1	Debug Exception	Fault/Trap
2	Non-Maskable Interrupt (NMI)	Interrupt
3	Breakpoint Exception	Trap
4	Overflow Exception	Trap
5	Bound Range Exceeded	Fault
6	Invalid Opcode	Fault
7	Device Not Available (No Math Coprocessor)	Fault
8	Double Fault	Abort
9	Coprocessor Segment Overrun (Legacy)	Fault
10	Invalid TSS	Fault
11	Segment Not Present	Fault
12	Stack-Segment Fault	Fault
13	General Protection Fault (GPF)	Fault
14	Page Fault	Fault
15	Reserved	-
...		
20	Virtualization Exception	Fault
21-31	Reserved	-
32-255	User-Defined Interrupts	Interrupt

- From Intel's Software Developer's Manual, p. 3268

- Check also Tab.-7-4/5: Conditions for a Double Fault

- And see kernel/idt.c, for where the page fault exception handler is one of the first to be added

What synchronous events invoke the OS? (addition)

Exceptions invoking the OS

- ***Faults***: Exceptions that allow the program to be restarted without loss of continuity
- ***Aborts***: Exceptions used to report severe errors
- ***Traps***: Exceptions reported immediately after the execution of a trapping instruction

System calls: The interface between user programs and the OS

- The defacto mechanism for user programs to request services from the OS
- Linux x86_64 defines ~500 syscalls and aarch64 ~460:

x86/entry/syscalls/syscall_64.tbl

/ arch / x86 / entry / syscalls / syscall_64.tbl

```
60      common  exit            sys_exit
61      common  wait4           sys_wait4
62      common  kill            sys_kill
        ...
```

arm64/tools/syscall_64.tbl

/ arch / arm64 / tools

```
93  common  exit            sys_exit
94  common  exit_group      sys_exit_group
95  common  waitid          sys_waitid
        ...
```

Syscall argument passing (aarch64 example) (addition)

DESCRIPTION

_exit(int *status*): Terminates the calling process…

The value *status & 0xFF* is returned … as the exit status…

```
SYSCALL_DEFINE1(exit, int, error_code)
{
        // keep only the last byte, and shift it one byte left
        do_exit((error_code&0xff)<<8);
}

int main(void) {
   __asm__ __volatile__ ("mov x0, #123");   // Syscall arg0 to reg. %x0, and so on
   __asm__ __volatile__ ("mov x8, #93");    // Syscall NR to reg. %x8 (NR 93 is exit)
   __asm__ __volatile__ ("svc  #0");        // Execute the syscall
}
```

```
$ ./svc_test

$ echo $?

123
```

Syscall argument passing (x86_64 example) (addition)

DESCRIPTION

_exit(int *status*): Terminates the calling process…

The value *status & 0xFF* is returned … as the exit status…

```
SYSCALL_DEFINE1(exit, int, error_code)
{
        // keep only the last byte, and shift it one byte left
        do_exit((error_code&0xff)<<8);
}

int main(void) {
   __asm__ __volatile__ ("mov $123, %rdi");  // Syscall arg0 to reg. %rdi, and so on
   __asm__ __volatile__ ("mov $60, %rax");   // Syscall NR to reg. %rax (NR 60 is exit)
   __asm__ __volatile__ ("syscall");         // Execute the syscall
}
```

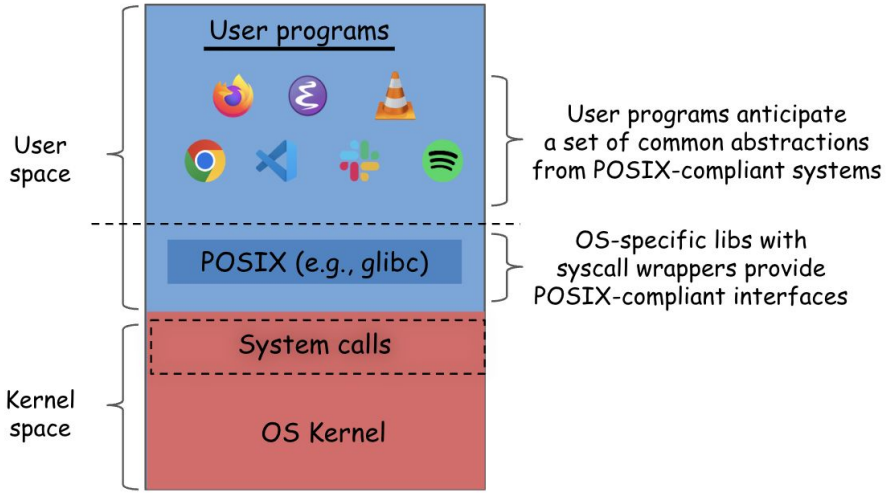
```
$ ./syscall_test

$ echo $?

123
```

How do user programs know what services an OS offers?

- We need a standard for that
  - Portable Operating System Interface for UNIX (**POSIX**) is the IEEE standard for portable UNIX-based OS syscall interfaces
  - ***Describes a set of fundamental abstractions needed for portable application development***
  - User programs can directly invoke system calls, but going through POSIX ensures portability across POSIX-compliant OSes.
    - User programs must not break on kernel updates
    - OK to recompile code, if I move to another OS
    - But…don't make me rewrite it

How do user programs know what services an OS offers?

- We need a standard for that
  - Portable Operating System Interface for UNIX (**POSIX**) is the IEEE standard for portable UNIX-based OS syscall interfaces

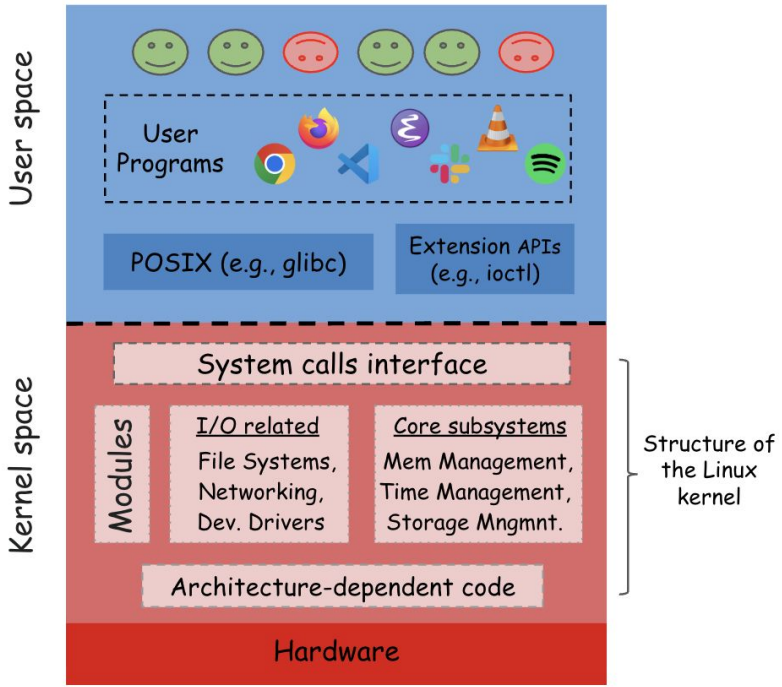


What does POSIX define?

POSIX defines ~1,200 interfaces, around the following abstractions

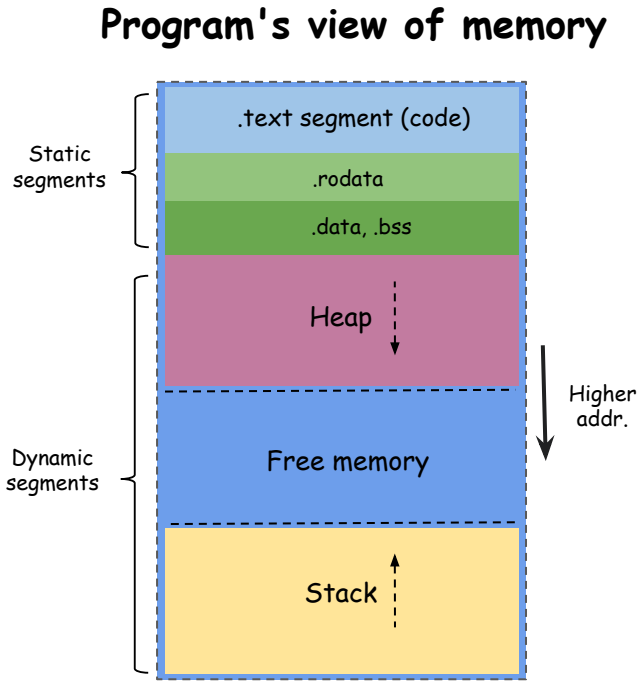
- ***Processes***: A process is an address space with one or more threads executing within that and the required system resources for those threads
- ***Threads***: A thread is a single flow of control within a process, with its own system resources required to support a flow of control
- ***Files***: A file is an object that can be written to, read from, or both
- The complete spec with all definitions is here

What is Linux?



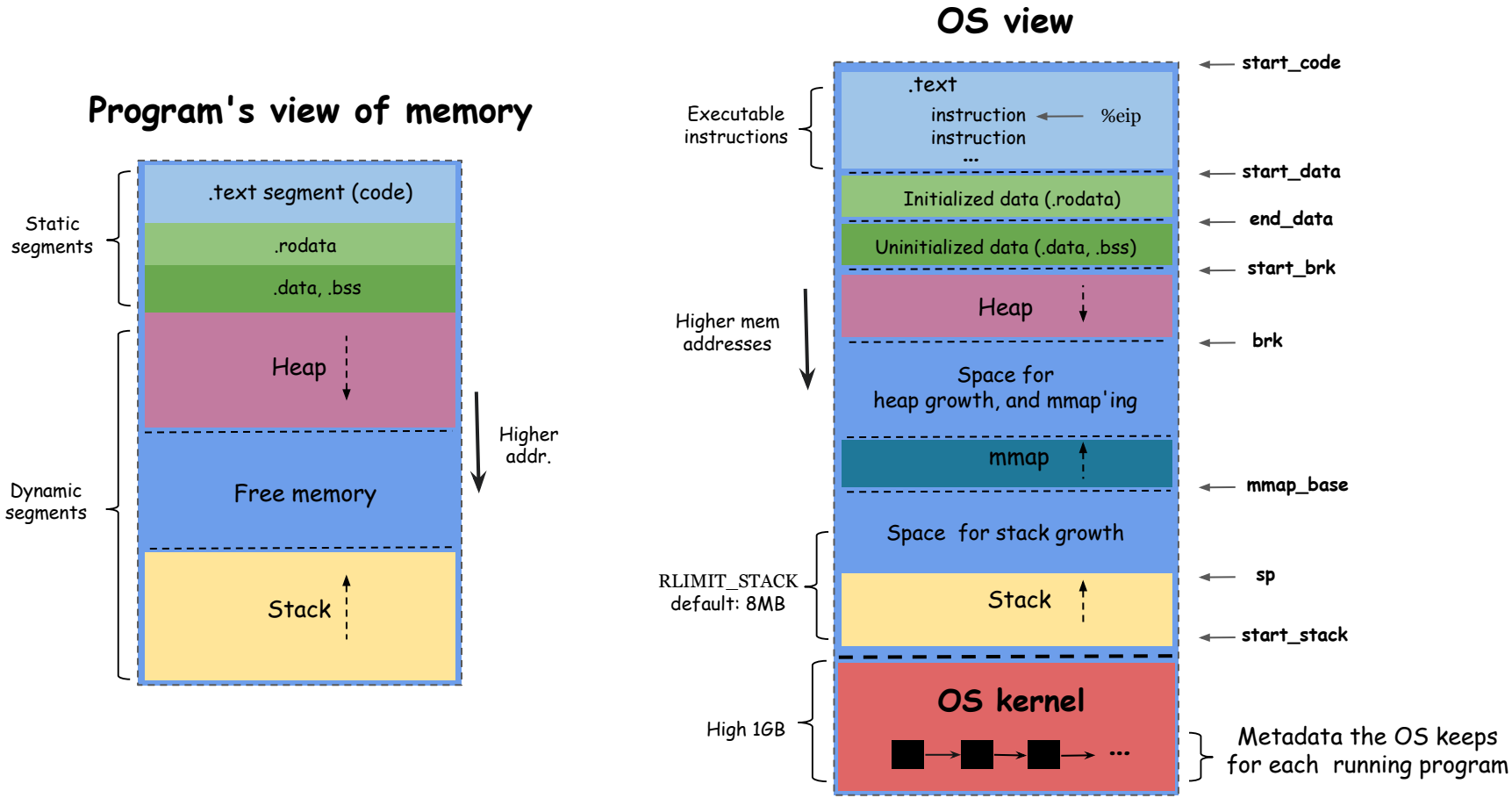
- A modern, open-source, POSIX-compliant OS kernel.
- **Written in '91 by Linus Torvalds from scratch ~100 KLoC (Jan '25: > 40 MLoC).**
- ***The most popular Unix-like OS today***
  - Servers: Over 96% of the world's top 1 million websites run on Linux servers.
  - Cloud: AWS, Google Cloud, and Azure rely heavily on Linux
  - Supercomputers: Almost 100% of the top 500 supercomputers run Linux
  - Android devices: >3 billion devices

How does a program look like in memory?



- ***Static Segments***: *Their size is static during runtime*
  - .text: Executable instr. of the program—read-execute perms
  - .rodata: Constant values—read-only perms
  - .data: Initialized global and static variables—read-write perms
  - .bss: Uninitialized global and static variables—read-write perms
- ***Dynamic segments***: *Their size can grow during runtime*
  - Heap: Grows (commonly) towards higher addresses and contains variables dynamically allocated—read-write perms
  - Stack: Grows (commonly) towards lower addresses and is used for bookkeeping during function calls—read-write perms

How does a program look like in memory?



How does a program look like in memory?

```
char uninitialized_global[100];
const char *message = "Hello, World!\n";

void foo() {
  unsigned long sp;
  __asm__ (
    "mov %0, sp"
    : "=r" (sp)
  );
  printf("@foo / Current stack pointer (sp): %lx\n", sp);
}

void main() {
  unsigned long sp;
  printf("@.rodata variable: %p\n", message);
  printf("@.bss variable: %p\n", &uninitialized_global);
  __asm__ (
    "mov %0, sp"
    : "=r" (sp)
  );
  char *heap = (char *)malloc(50 * sizeof(char));
  printf("@heap variable: %p\n", heap);
  printf("@main / Current sp: %lx\n", sp);

  foo();
}
```

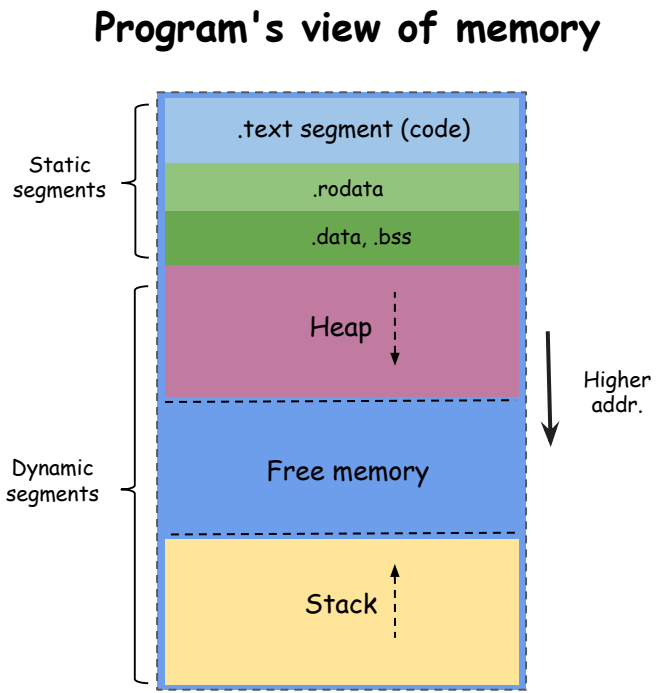
How does a program look like in memory?

```
char uninitialized_global[100];
const char *message = "Hello, World!\n";

void foo() {
   unsigned long sp;
   __asm__ (
      "mov %0, sp"
      : "=r" (sp)
   );
   printf("@foo / Current stack pointer (sp): %lx\n", sp);
}

void main() {
   unsigned long sp;
   printf("@.rodata variable: %p\n", message);
   printf("@.bss variable: %p\n", &uninitialized_global);
   __asm__ (
      "mov %0, sp"
      : "=r" (sp)
   );
   char *heap = (char *)malloc(50 * sizeof(char));
   printf("@heap variable: %p\n", heap);
   printf("@main / Current sp: %lx\n", sp);

   foo();
}
```

```
➜  cat /proc/1245382/maps
aaaadcda0000-aaaadcda1000 r-xp    …   /os/sample  [.text]
aaaadcdb0000-aaaadcdb1000 r--p    …  /os/sample  [.rodata]
aaaadcdb1000-aaaadcdb2000 rw-p    …  /os/sample  [.bss, .data]
aaaaed20d000-aaaaed22e000 rw-p    …  [heap]
ffffbe610000-ffffbe798000   r-xp      …  /usr/lib/libc.so.6
ffffbe798000-ffffbe7a7000   ---p      …  /usr/lib/libc.so.6
ffffbe7a7000-ffffbe7ab000   r--p      …  /usr/lib/libc.so.6
ffffbe7ab000-ffffbe7ad000   rw-p      … /usr/lib/libc.so.6
ffffbe7ad000-ffffbe7b9000   rw-p      …
ffffff6461000-ffffff6482000   rw-p    … [stack]
```

```
➜  ./sample
@ .rodata variable: 0xaaaadcdb0100
@ .bss  variable: 0xaaaadcdb1020
@ heap variable: 0xaaaaed20d6b0
@ main – Current sp: ffffff64810e0
@ foo   – Current sp: ffffff64810c0
```

How does a program look like in memory?




```
➜  cat /proc/1245382/maps
aaaadcda0000-aaaadcda1000 r-xp    …   /os/sample  [.text]
aaaadcdb0000-aaaadcdb1000 r--p    …  /os/sample  [.rodata]
aaaadcdb1000-aaaadcdb2000 rw-p    …  /os/sample  [.bss, .data]
aaaaed20d000-aaaaed22e000 rw-p    …  [heap]
ffffbe610000-ffffbe798000  r-xp      …  /usr/lib/libc.so.6
ffffbe798000-ffffbe7a7000  ---p      …  /usr/lib/libc.so.6
ffffbe7a7000-ffffbe7ab000  r--p      …  /usr/lib/libc.so.6
ffffbe7ab000-ffffbe7ad000  rw-p     … /usr/lib/libc.so.6
ffffbe7ad000-ffffbe7b9000  rw-p     …
fffff6461000-fffff6482000  rw-p     … [stack]

➜  ./sample
@ .rodata variable: 0xaaaadcdb0010
@ .bss  variable: 0xaaaadcdb1020 > 0xaaaadcdb0010
@ heap variable: 0xaaaaed20d6b0
@ main – Current sp: fffff64810e0
@ foo   – Current sp: fffff64810c0 < fffff64810e0 (main's sp)
```

Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)
  - Reads a program from storage (called an *executable*), "interprets" it, and sets up the appropriate segments it in memory
    - We need a specification to serve as the contract (interface) between executables and the loader
    - **ELF**: The Executable and Linkable Format (ELF)

Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)
  - Reads a program from storage (called an *executable*), "interprets" it, and sets up the appropriate segments it in memory
  - A ELF executable loaded in memory will be called a *process* soon
    - The static segments are initialized by copying from the ELF
    - The default dynamic segments are laid out by the loader, and the OS intervenes to manage them

Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)
  - Reads a program from storage (called an *executable*), "interprets" it, and sets up the appropriate segments it in memory
  - A ELF executable loaded in memory will be called a *process* soon
  - The loader transfers control to the ELF's entry point (e.g., _start)
    - If the executable is statically linked, the loader's job is complete

Static linking

hello.c

```
extern void foo(int);

int bar(int a){
   a += 1;
   return a;
}

void main(void){
   foo(1);
   bar(2);
}
```

➜ gcc -S hello.c ⇒ gcc -c hello.s ⇒ hello.o
➜ readelf -s hello.o

```
# Table of all global, exported symbols with offset and  section number
Num:    Value              Size Type    Bind   Vis      Ndx Name
  2: 0000000000000000     0    SECTION LOCAL  DEFAULT    1 .text
  3: 0000000000000000     0    SECTION LOCAL  DEFAULT    3 .data
  4: 0000000000000000     0    SECTION LOCAL  DEFAULT    4 .bss
  …
 10: 0000000000000000    32    FUNC    GLOBAL DEFAULT    1 bar
 11: 0000000000000020    36    FUNC    GLOBAL DEFAULT    1 main
 12: 0000000000000000     0    NOTYPE  GLOBAL DEFAULT  UND foo
```

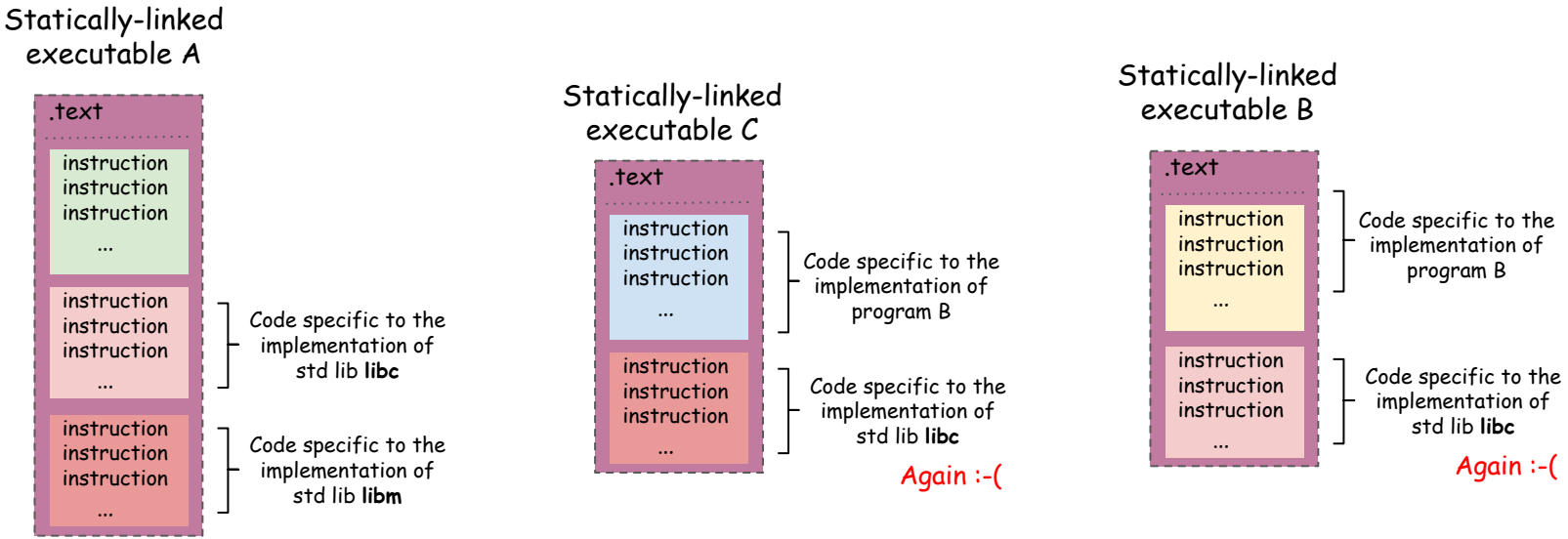
➜ nm hello.o

```
# If don't need full functionality of readelf, use nm
                 U foo
0000000000000000 T bar
0000000000000020 T main
```

➜ gcc -o exe foo.o hello.o
➜ nm exe

```
# All relocations from statically-linked objects are resolved
…
0000000000000758  T foo
0000000000000714  T bar
0000000000000734  T main
```

Static linking: The problem



Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)
  - Reads a program from storage (called an *executable*), "interprets" it, and sets up the appropriate segments it in memory
  - A ELF executable loaded in memory will be called a *process* soon
  - The loader transfers control to the ELFs entry point (e.g., _start)
    - If the executable is statically linked, the loader's job is complete
    - If the executable is dynamically linked, we need more…

Dynamic linking

- What if we know how to find this at runtime?

```
➜ gcc -S hello.c ⇒ gcc -c hello.s ⇒ hello.o
➜ readelf -s hello.o

# Table of all global, exported symbols with offset and section number
Num:    Value              Size Type    Bind   Vis      Ndx Name
  2: 0000000000000000     0    SECTION LOCAL  DEFAULT    1 .text
  3: 0000000000000000     0    SECTION LOCAL  DEFAULT    3 .data
  4: 0000000000000000     0    SECTION LOCAL  DEFAULT    4 .bss
  …
 10: 0000000000000000    32    FUNC    GLOBAL DEFAULT    1 bar
 11: 0000000000000020    36    FUNC    GLOBAL DEFAULT    1 main
 12: 0000000000000000     0    NOTYPE  GLOBAL DEFAULT  UND foo

➜ nm hello.o

# If don't need full functionality of readelf, use nm
                 U foo
0000000000000000 T bar
0000000000000020 T main

➜ gcc -o exe foo.o hello.o
➜ nm exe

# All relocations from statically-linked objects are resolved
…
0000000000000758 T foo
0000000000000714 T bar
0000000000000734 T main
```

Dynamic linking

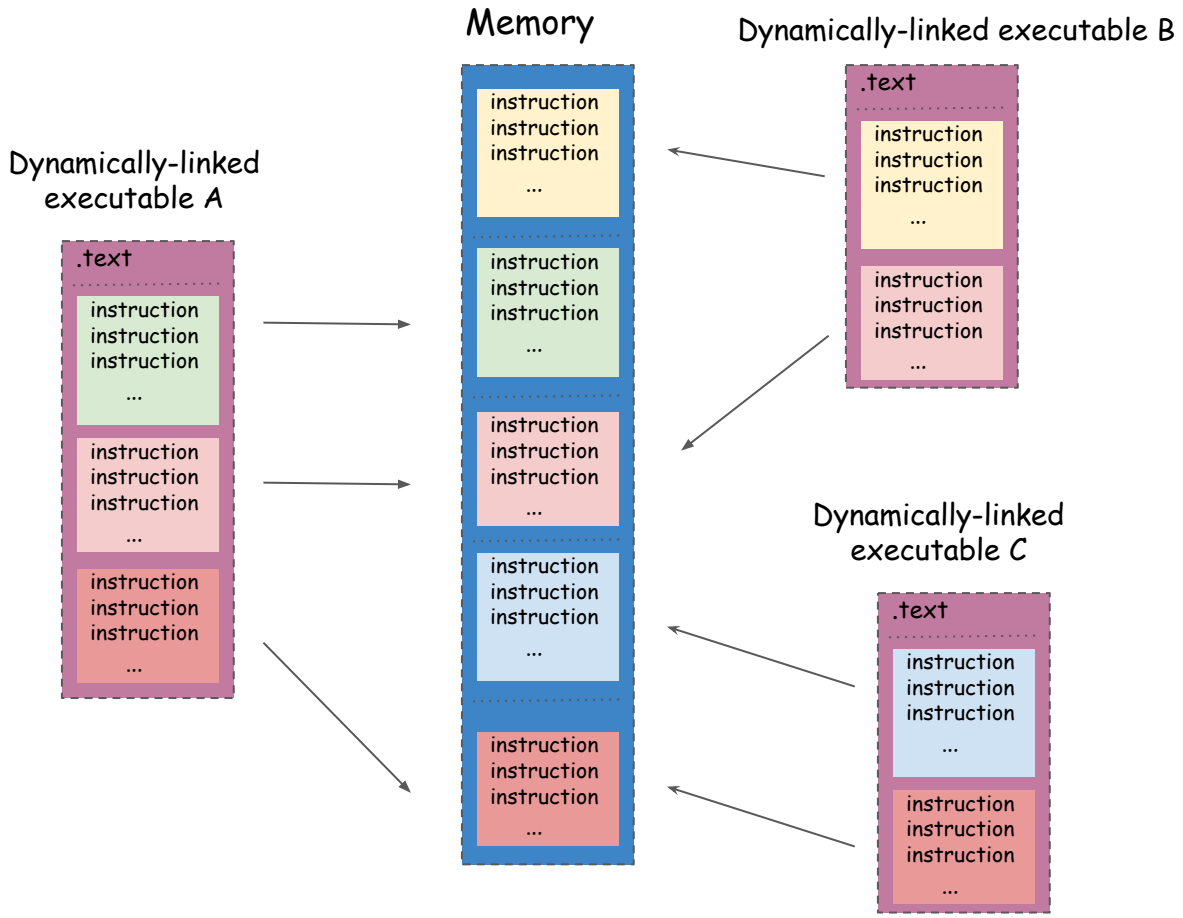
- What if we know how to find this at runtime?
- Well…another drama starts…

```
➜  gcc -S hello.c ⇒ gcc -c hello.s ⇒ hello.o
➜  readelf -s hello.o

# Table of all global, exported symbols with offset and  section number
Num:    Value          Size Type    Bind   Vis      Ndx Name
  2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
  3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3 .data
  4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4 .bss
  …
 10: 0000000000000000    32 FUNC    GLOBAL DEFAULT    1 bar
 11: 0000000000000020    36 FUNC    GLOBAL DEFAULT    1 main
 12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND foo

➜  nm  hello.o

# If don't need full functionality of readelf, use nm
                 U foo
0000000000000000 T bar
0000000000000020 T main

➜  gcc -o exe foo.o hello.o
➜  nm exe

# All relocations from statically-linked objects are resolved
…
0000000000000758  T foo
0000000000000714  T bar
0000000000000734  T main
```

Who sets up the program in memory?

- The loader is responsible for loading a program into memory (code here)
  - Reads a program from storage (called an *executable*), "interprets" it, and sets up the appropriate segments it in memory
  - A ELF executable loaded in memory will be called a *process* soon
  - The loader transfers control to the ELFs entry point (e.g., _start)
    - If the executable is statically linked, the loader's job is complete
    - If the executable is dynamically linked we need more…
  - **The dynamic linker resolves symbols at runtime**
    - Specified in the .interp section of the ELF (e.g., ld-linux.so)
    - Yet another time we deferred stuff for later

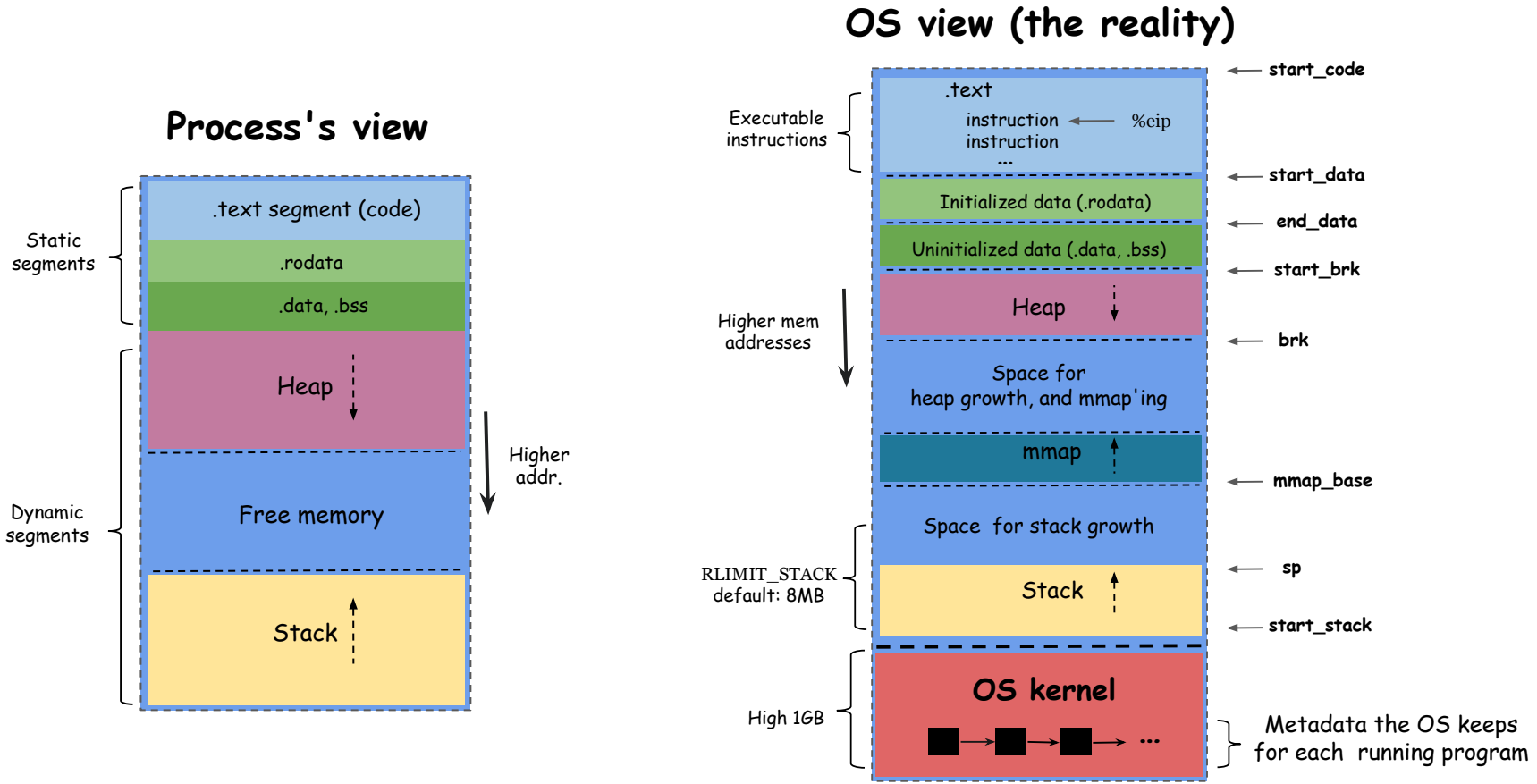
Dynamic Linking / Loading



The process abstraction

- What is a process? "An address space with one or more threads executing within it." (Strict POSIX definition, 3/189.)

Virtual Address Space (VAS)



The process abstraction

- What is a process? "An address space with one or more threads executing within it." (Strict POSIX definition, 3/189.)

- Popularized in 1974, in the context of the Bell Labs paper called "The UNIX Time-Sharing System."

- In simple words: A process is an instance of a program in execution
  - A program is a recipe ⇒ A process is the mess in your kitchen

- How we use it?
  - Want to run a program? Use a process
  - Want to run multiple programs? Use multiple processes

The process abstraction

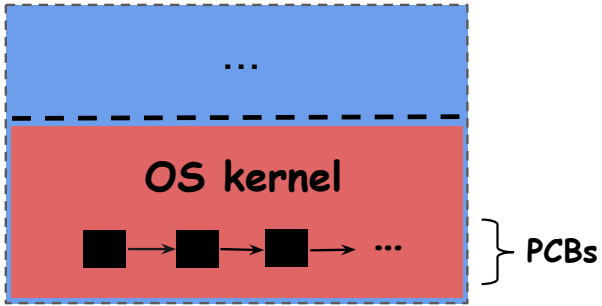
- Why we love it?

The process abstraction

- Why we love it? The definite illusion: User programs can be written as if they will get isolated access to all system resources
- Noone else will exist in memory
  - Remember printf("%x | %d\n", &x, x) from quiz01?
  - Two processes may see a different value at the same virt. address
- Noone else will be using the processor
  - Remember preemption and this timer ticking?
  - Every process has its own "virtual" processor time
- Noone else will be using using the storage
  - This is for later, but the ideas are similar…
  - Every process may read from a file as if no other is writing to it

Components of a process

- Process Control Block (PCB): A struct used by the OS to keep track of each running process (see task_struct)



Components of a process

- Process Control Block (PCB): A struct used by the OS to keep track of each running process (see task_struct)
  - Virtual Address Space (see struct mm): A linear array of bytes with all static and dynamic segments in virtual memory of a running process
  - Execution state (see thread_struct): An instruction pointer, a stack pointer, and a set of general-purpose registers with their values
  - Control metadata: Scheduling (see sched_entity and sched_statistics), identity (see struct cred), and more...
  - Shared system resources: Open files (see files_struct) and open network connections (see struct sock), and more...

Process creation: To POSIX or not to POSIX?

- Create a new process by cloning an existing process
  - Pause the current process and save its state
  - Duplicate its address space and its PCB
  - Add the new PCB to the in-kernel queue of PCBs
  - Use the return value to diverge
  - Requires distinguished init process…

Process creation: To POSIX

DESCRIPTION

pid_t fork(void): Creates a new process by duplicating the calling process

- Returns process ID (PID) of the new process in the "parent" process
- Returns 0 in the "child" process

```
SYSCALL_DEFINE0(fork)
{
    ...
    return kernel_clone(&args);
}
```

Process creation: To POSIX

int fork ()

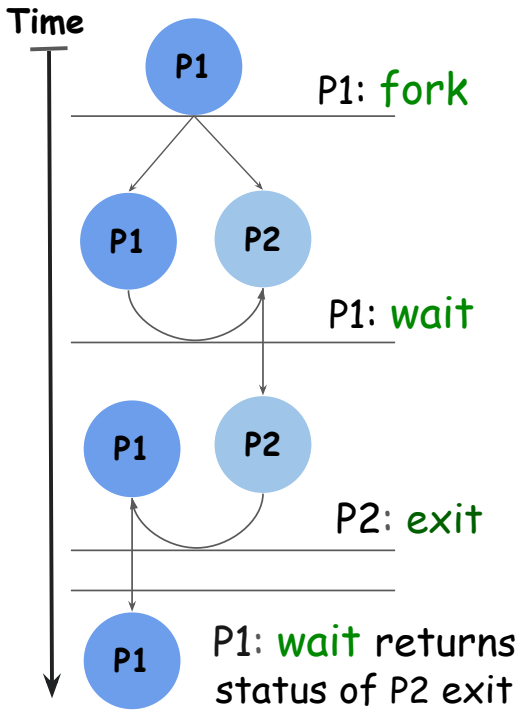
- **Creates a new process by duplicating the calling process**
- Returns: Pid of the new process in "parent" process
- Returns: 0 in the "child" process

int waitpid (int pid, int *stat, ...) ⇐ Parent calls this

- **pid**: process to wait for, or -1 for all
- **stat**: will contain exit value, or signal
- Returns process ID, or -1 on error

void exit (int status)

- **status**: shows up in waitpid (shifted)
- Current process ceases to exist
- Convention: pass 0 on success, non-zero on error



Process creation: To POSIX or not to POSIX?

- Create a new process by cloning an existing process
  - Pause the current process and save its state
  - Duplicate its address space and its PCB
  - Add the new PCB to the in-kernel queue of PCBs
  - Use the return value to diverge
  - Requires distinguished init process…
- Creating a new process from scratch
  - Create and initialize a new PCB
  - Add a new PCB to the in-kernel queue of PCBs
  - Does not require a distinguished process…but….

Process creation: Not to POSIX

DESCRIPTION

CreateProcessA: Creates a new process and its primary thread

```
BOOL CreateProcessA(
        [in, optional]          LPCSTR                  lpApplicationName,
        [in, out, optional]     LPSTR                   lpCommandLine,
        [in, optional]          LPSECURITY_ATTRIBUTES   lpProcessAttributes,
        [in, optional]          LPSECURITY_ATTRIBUTES   lpThreadAttributes,
        [in]                    BOOL                    bInheritHandles,
        [in]                    DWORD                   dwCreationFlags,
        [in, optional]          LPVOID                  lpEnvironment,
        [in, optional]          LPCSTR                  lpCurrentDirectory,
        [in]                    LPSTARTUPINFOA          lpStartupInfo,
        [out]                   LPPROCESS_INFORMATION   lpProcessInformation
);
```

The elegant simplicity of POSIX!

DESCRIPTION

pid_t fork(void): Creates a new process by duplicating the calling process

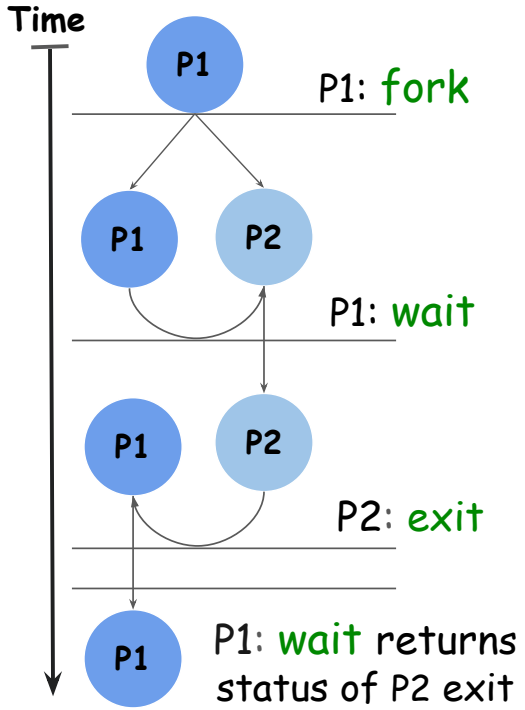
- Returns process ID of new process in the "parent" process
- Returns 0 in the "child" process

```
SYSCALL_DEFINE0(fork)
{
        ...
        return kernel_clone(&args);
}
```

How to execute a program?

int execve (char *prog, char **argv, ...)

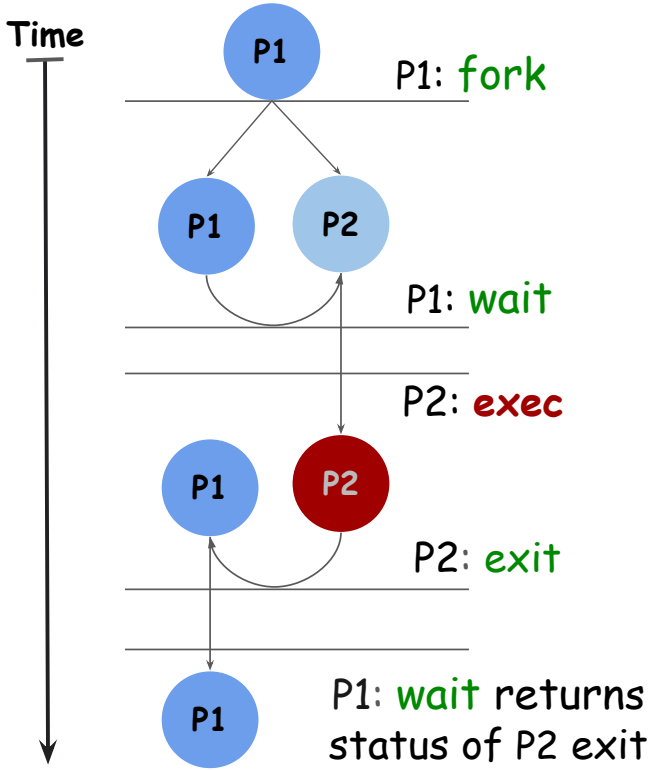
- **prog**: Full pathname of a program to run
- **argv**: Arguments that get passed to main
- **envp**: Environment variables, e.g., PATH, HOME
- Does not return on success



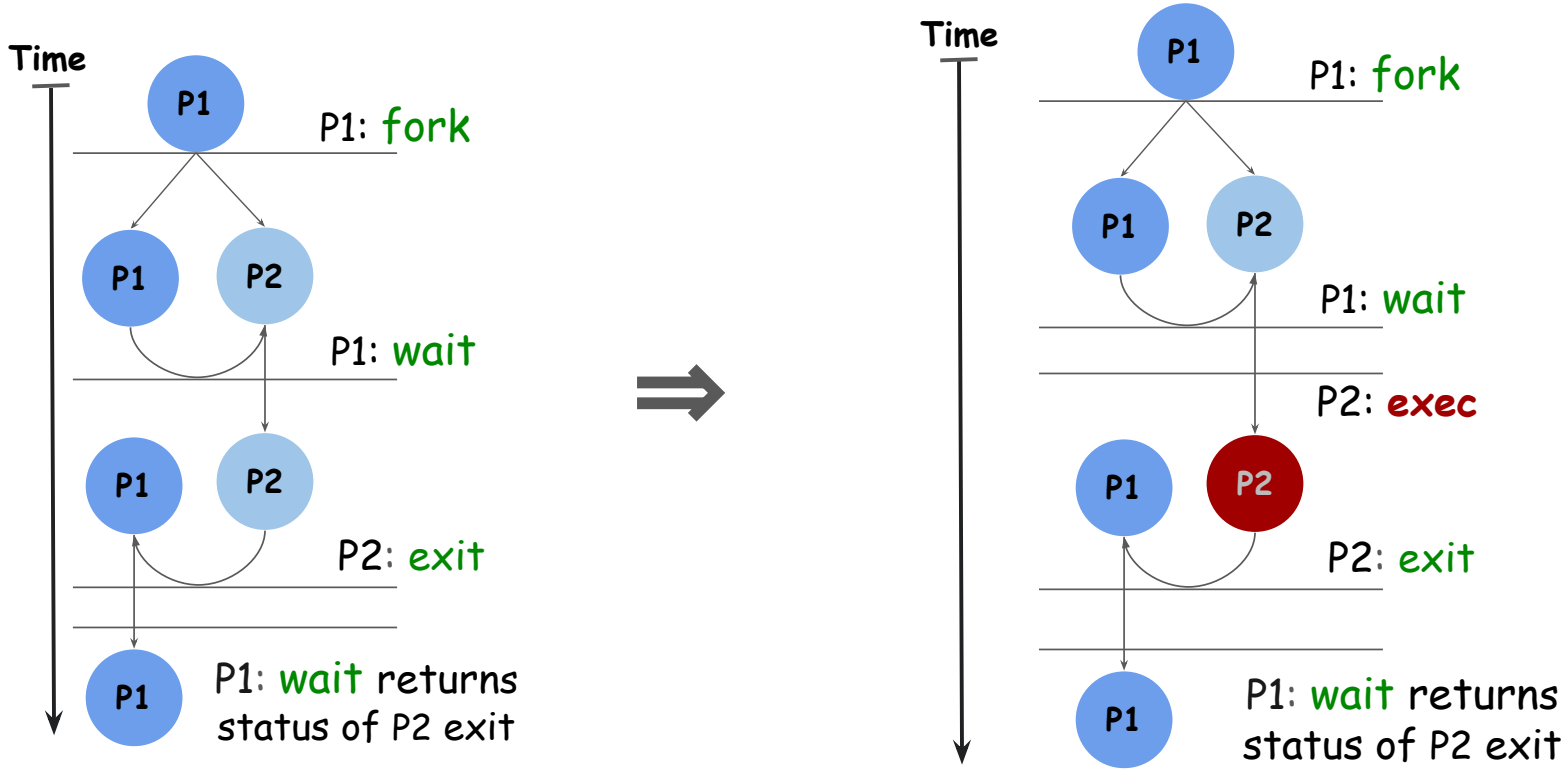
How to execute a program?

int execve (char *prog, char **argv, ...)

- **prog**: Full pathname of a program to run
- **argv**: Arguments that get passed to main
- **envp**: Environment variables, e.g., PATH, HOME
- Does not return on success



How to execute a program?



Process states

- A POSIX process has an execution state which indicates what the process is currently "doing"

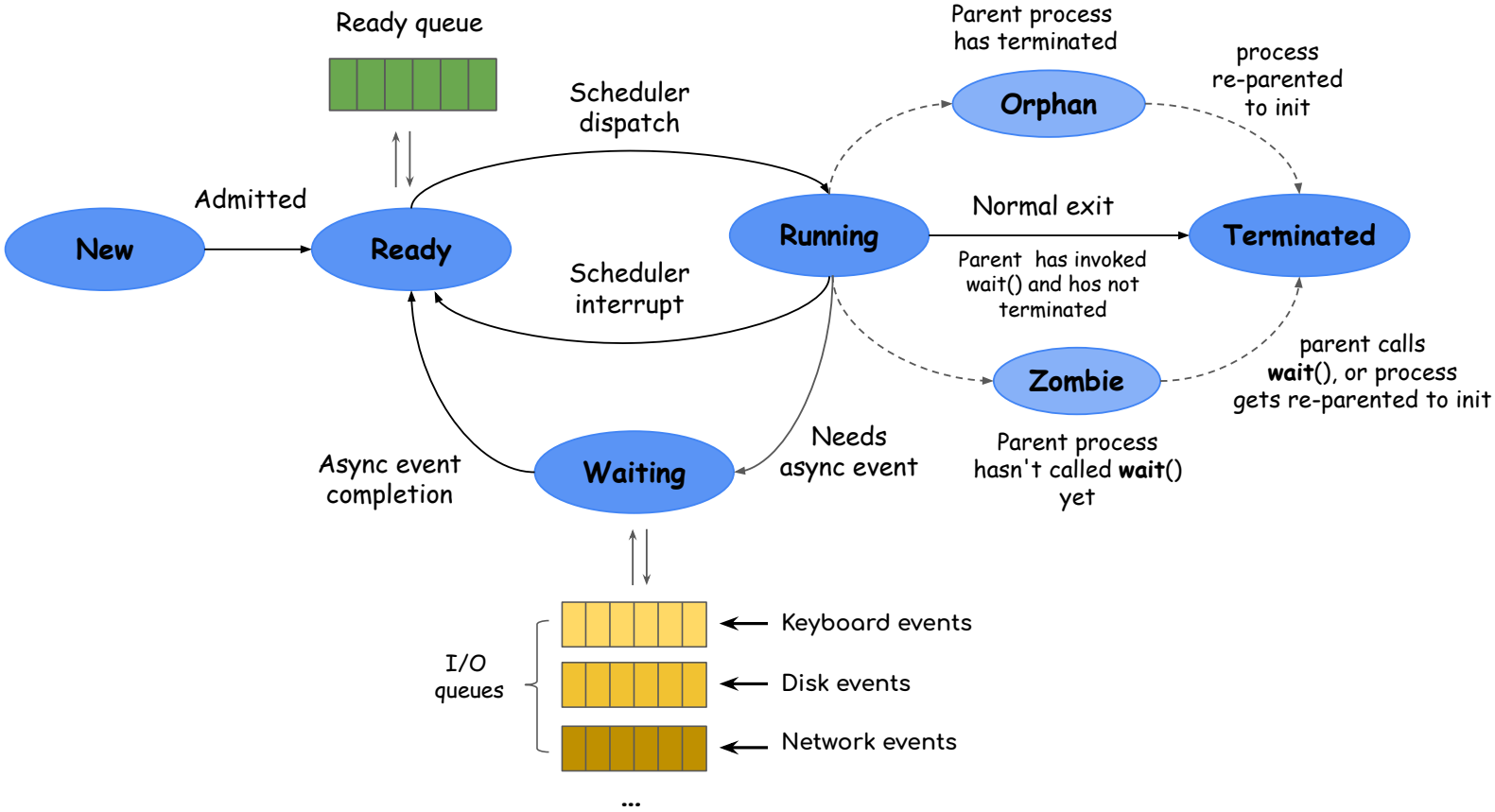
- Each process' PCB is queued on the respective queue

- As the process executes ⇒ It transitions from state to state

Process states

- Ready: The process is ready to be executed, but it's not executing yet because another process is using the processor
- Waiting (blocked): The process is waiting for an async event to complete (e.g., a disk I/O), and cannot progress until the event completes
- Running: The process is executing on the processor until either
  - (i) An async event is required ⇒the process transitions to the "waiting" queue
  - (ii) It exceeds its maximum quantum ⇒ a scheduler interrupt occurs
- **Terminated**: The process finished execution
  - Normally: By calling exit after its parent has called wait
  - As a "zombie:" The parent exists, but hasn't called wait() yet
  - Orphan: The parent has exited already

Process states



Process dispatching

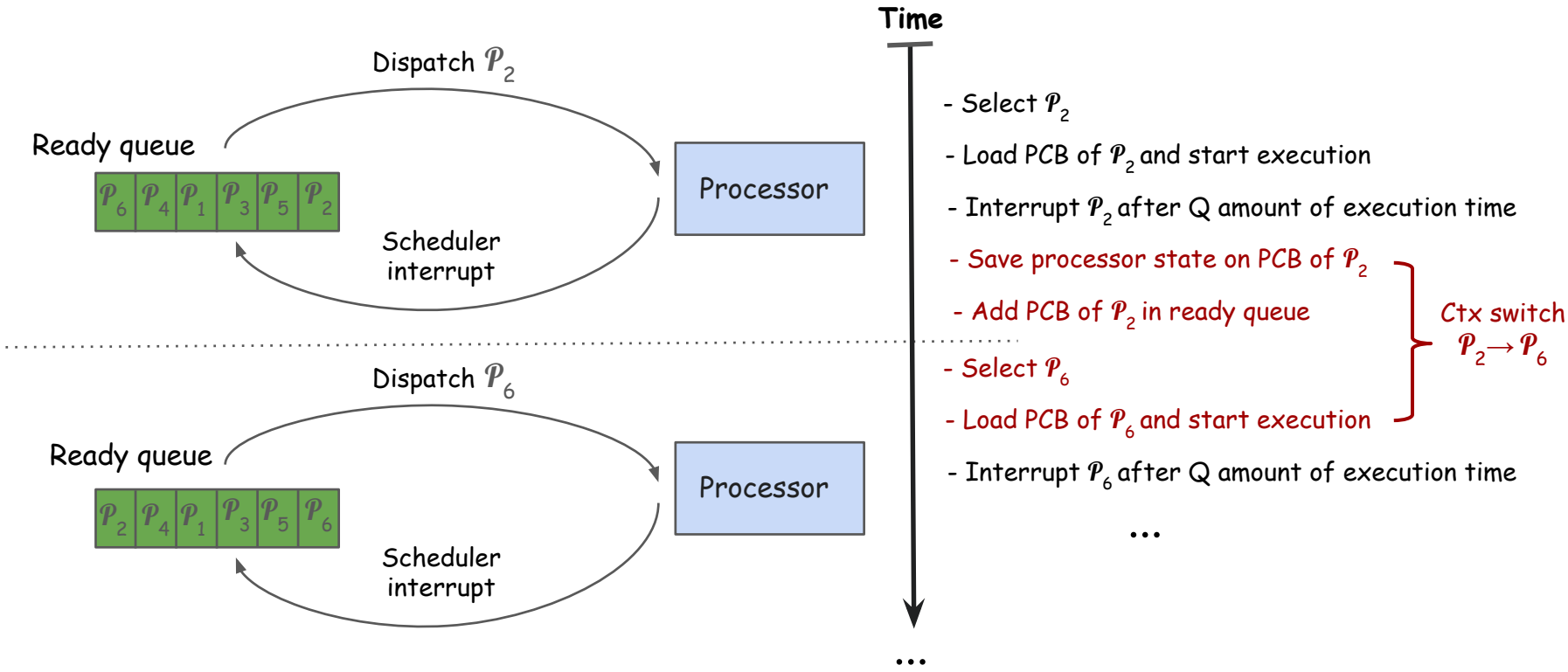
- Many processes in memory
- One allocated on the processor

} Multiprogramming

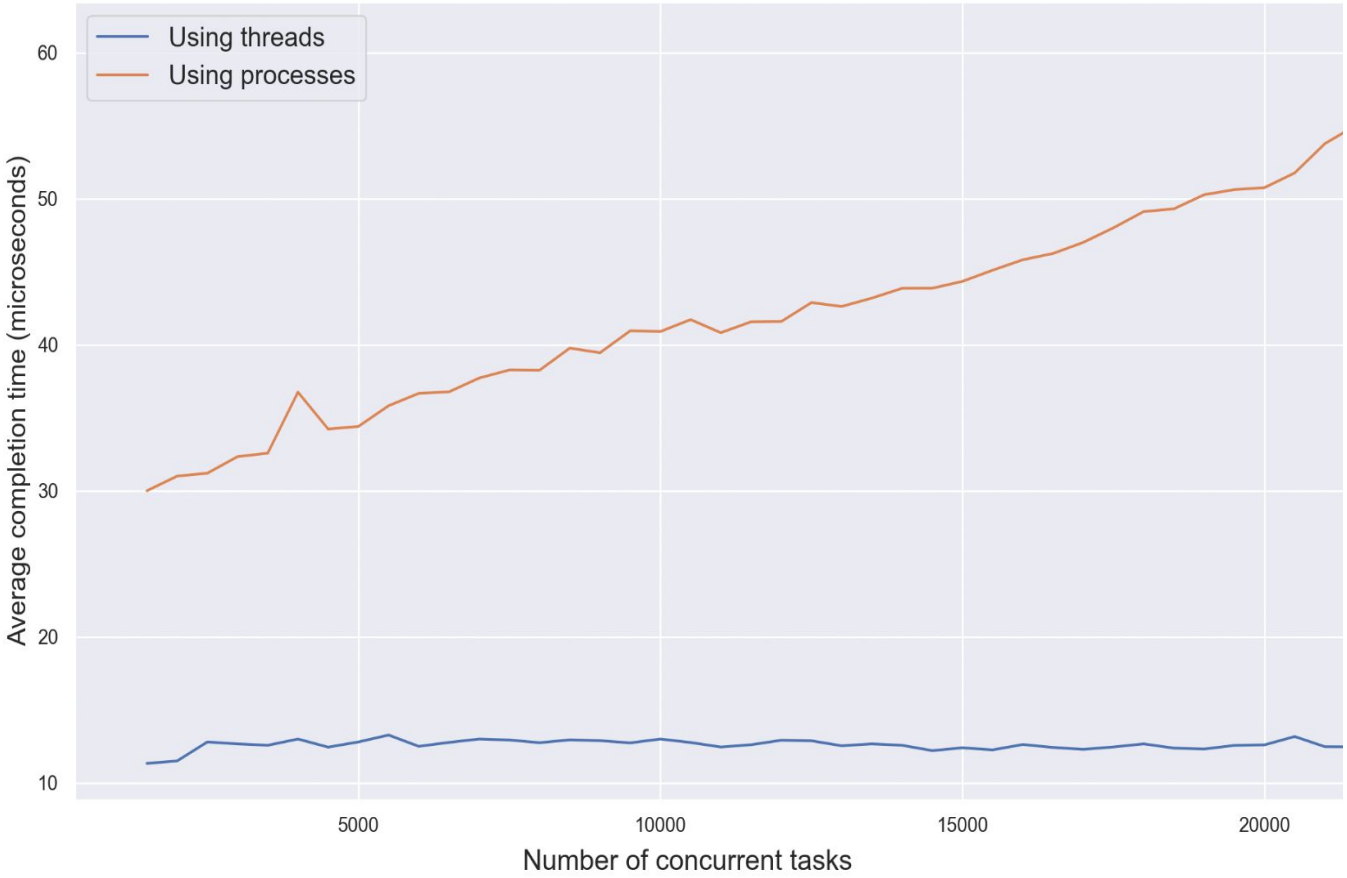
- **Our goal:** Give each process the illusion it has the full processor
- **In other words**: Run multiple processes simultaneously
- Timesharing dispatching loop: Preemption — periodic timer interrupt

```
do {
    Get a process P from ready queue
      Execute P until time Q expires
        Put  P back in ready queue
} while(1)
```

Process dispatching



The miserable cost of fork()...




```c
void process_task(int num){
  is_prime(num);
  _exit(0);
}

int main(int argc, char **argv) {
  ...
 for (int i = 0; i < num_tasks; i++)
    numbers[i] = rand() % 10000000;
 ...
 for (int i = 0; i < num_tasks; i++) {
    pid_t pid = fork();
    if (pid == 0)
        process_task(numbers[i]);
 }

 for (int i = 0; i < num_tasks; i++)
    wait(NULL);
}
```

Inter-Process Communication (IPC)

A process has, so far, been the de-facto isolation mechanism

- What if we wish a process to communicate with another?

*) Why need communication b/w processes? ⇒ Cooperation!

> The OS primitive for communication between processes is called Inter-Process Communication (IPC)

> Two core paradigms for IPC

- Synchronous: The recipient can be assumed "prepared"
- Asynchronous: The recipient cannot be assumed "prepared"

Asynchronous and synchronous IPC

> Asynchronous IPC: The recipient process is not necessarily waiting for a "message" to be delivered to it

- Example: POSIX signals

> Synchronous IPC: The recipient process is waiting for a message to be delivered to it

- Example: POSIX pipes, POSIX shared memory

Asynchronous IPC: Signals

- POSIX signals is the most common async IPC mechanism
  - The syscall int kill(pid_t pid, int signo) is used to ask the kernel to deliver a signal from one process to another
  - A "signal" is a short message, identified by a small integer
  - There are **32 predefined** POSIX signals
    - SIGINT(2): Terminal interrupt signal
    - SIGSEGV(11): Invalid memory reference
  - int sigaction(int **signum**, … sigaction ***act**, … sigaction ***oldact**)
    - Used by a process to define an signal "handler": Action to take upon delivery of a signal to it
    - Signals that cannot be "handled": SIGKILL(9), SIGSTOP(19) [see this]

Asynchronous IPC: Signals

> 3.336 Signal (POSIX spec)
  - A mechanism by which a process or thread may be notified of, or affected by, an event occurring in the system
  - Examples include hardware exceptions and specific actions by processes
  - The term signal is also used to refer to the event itself

3.337 Signal Stack (POSIX spec)
  - Memory established for a thread, in which signal handlers catching signals sent to that thread are executed

3.13 Alternate Signal Stack (POSIX spec) – Why do we need this?
  - Memory associated with a thread, established upon request by the implementation for a thread, separate from the thread signal stack, in which signal handlers responding to signals sent to that thread may be executed

> See int sigaltstack(stack_t ***ss**, stack_t ***oss**)

Asynchronous IPC: Signals

> 2.4.1 Signal Generation and Delivery (POSIX spec)

- At the time of generation, a determination shall be made whether the signal has been generated for a process or for a specific thread within a process
- Signals generated by some action attributable to a particular thread shall be delivered to the thread that caused the signal to be generated

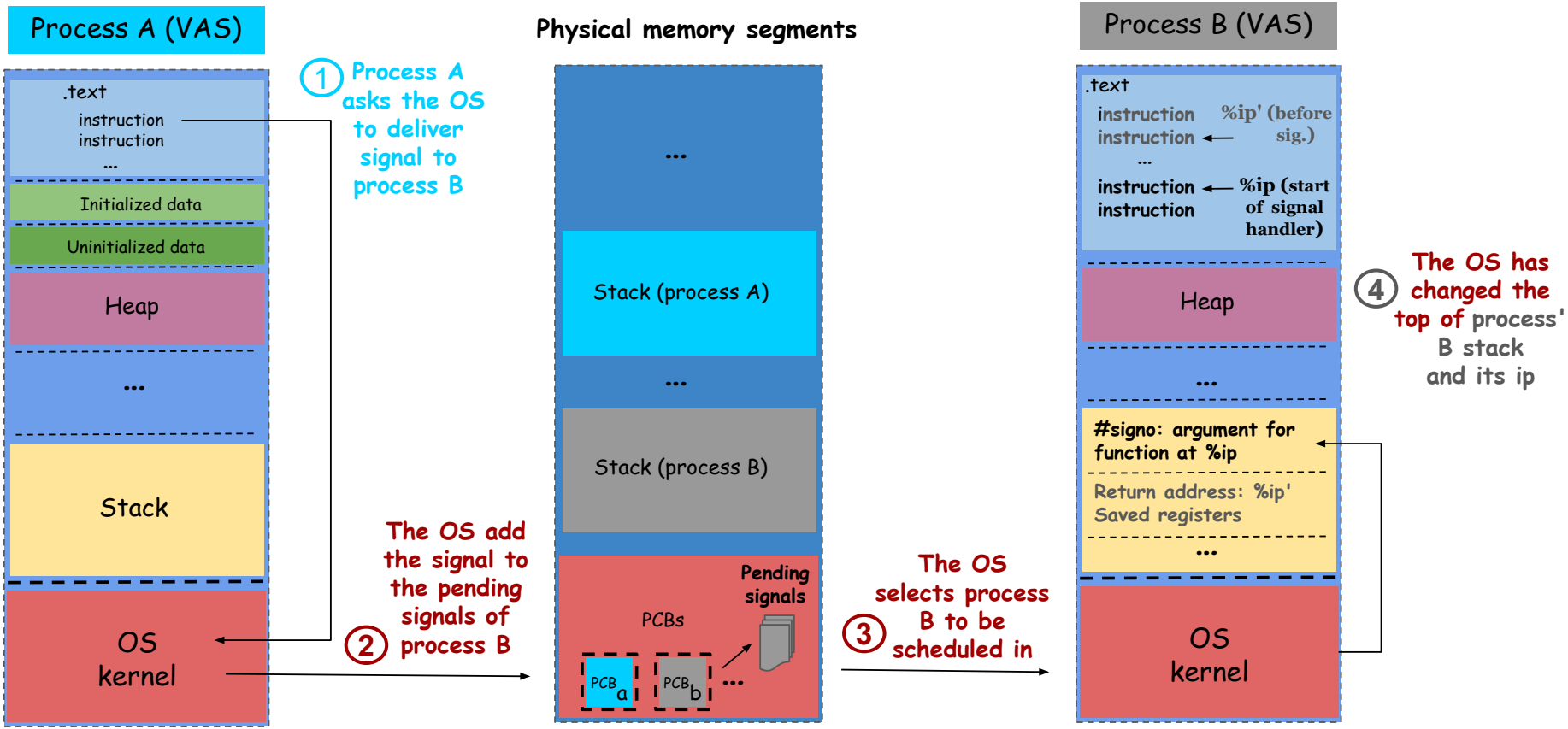
- Signals that are generated in association with a process ID or a process group ID or an asynchronous event shall be delivered to that process or process group

Asynchronous IPC: Signals

> 2.4.1 Signal Generation and Delivery (POSIX spec)

- During the time between the generation of a signal and its delivery or acceptance, the signal is said to be pending, and a signal can be blocked from delivery to a thread
- Signals generated for a process, shall be delivered to exactly one of the threads within the process which is in a call to sigwait() function selecting that signal, or has not blocked the delivery of the signal
- If the action associated with a blocked signal is anything other than to ignore the signal, and if that signal is generated for the thread, the signal shall remain pending until it is either (i) unblocked, or (ii) selected and returned by a call to sigwait() function, or (iii) the action associated with it is set to ignore the signal

Asynchronous IPC: Signals



Asynchronous IPC: Signals

> Sending a signal (Linux)

→ kill()
→ prepare_kill_siginfo()
→ kill_something_info()
→ kill_proc_info()
→ kill_pid_info()
→ kill_pid_info_type()
→ group_send_sig_info()
→ do_send_sig_info()
→ send_signal_locked()
→ __send_signal_locked()

> Handling a signal (Linux x86)

→ exit_to_user_mode_loop()
→ arch_do_signal_or_restart()
→ handle_signal()
→ setup_rt_frame()
→ x64_setup_rt_frame()

From an address translation error to a SIGSEGV

```
> do_translation_fault()
                |     do_page_fault()
                |       ...
                |       arm64_force_sig_fault(SIGSEGV, ...)
0.500 us        |         ...
                |         force_sig_fault()
                |           force_sig_info_to_task()
                |             send_signal_locked()
                |               __send_signal_locked()
0.666 us        |                 prepare_signal();
                |                   ...
                |                 complete_signal()
                |                   signal_wake_up()
                |                     signal_wake_up_state()
0.250 us        |                       wake_up_state()
                |                     try_to_wake_up_state
                |                       ...
0.333 us        |                     kick_process()
1.875 us        |                       ...
```

```c
int main(void) {
  char *p = 0x123;
  *p = 0;
}
```

This is only a small subset of all things the OS does when your program causes a SIGSEGV

 # How to enable kernel tracing

```
$ cd /sys/kernel/debug/tracing
$ echo function_graph > current_tracer
$ echo 114194 > set_ftrace_pid
$ echo 1 > tracing_on
$ cat trace > /tmp/kernel_trace.log
$ echo 0 > tracing_on
```

Synchronous IPC: Pipes (unnamed)

- POSIX pipes is the most common synchronous IPC mechanism
  - The syscall int pipe (int fds[2]) is used to ask the kernel to create a synchronous unidirectional communication channel b/w two processes
    - fds[0]: The read end of the pipe
    - fds[1]: The write end of the pipe

Operations on pipes: read/ write/ close (similar to files, later...)

- Read on fds[0] will block until data is written to fds[1]
- On success, the number of bytes read is returned
- A SIGPIPE will be delivered to a process trying to write on a pipe whose read end is closed

Synchronous IPC: Pipes (unnamed)

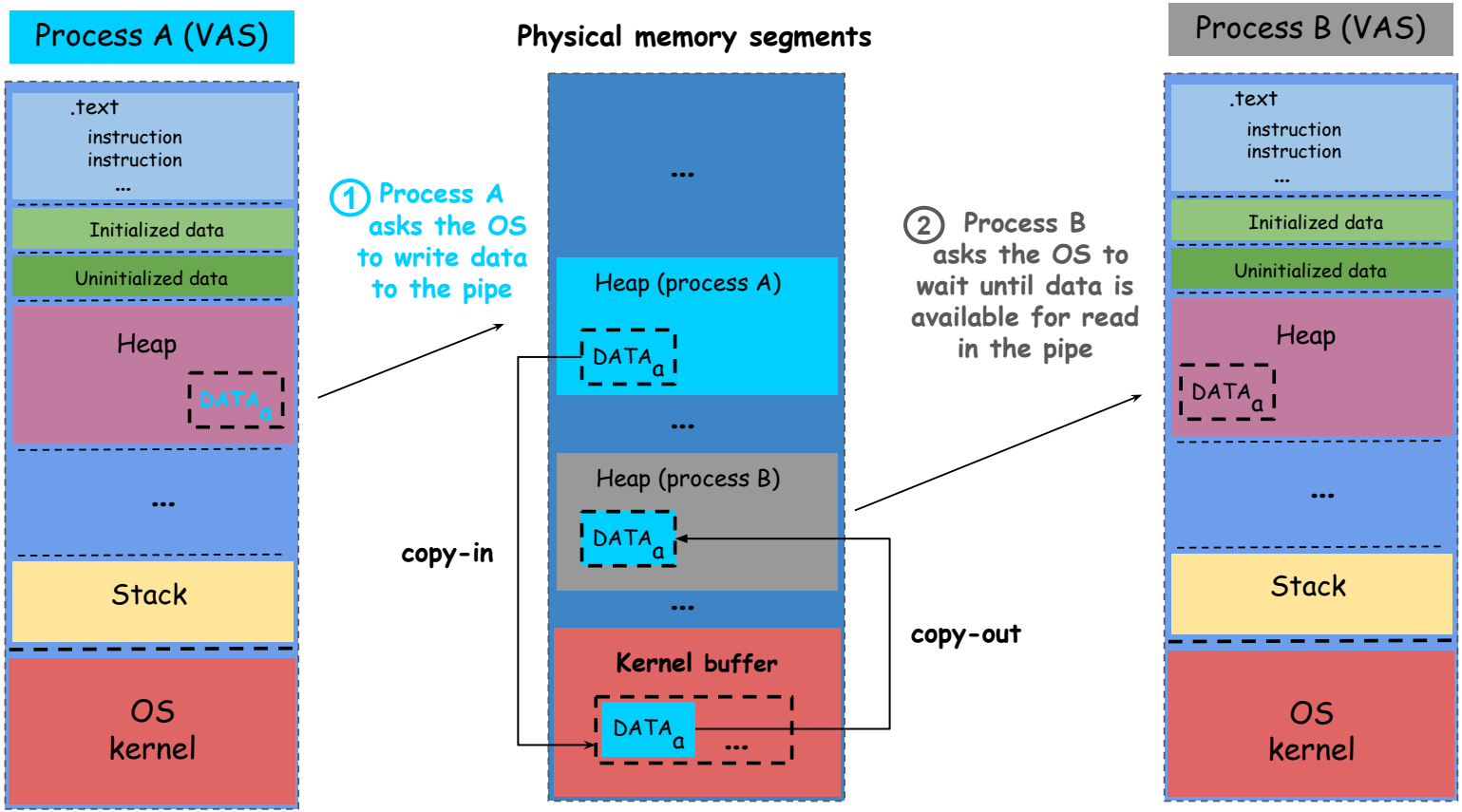
- POSIX pipes is the most common synchronous IPC mechanism
  - The syscall int pipe (int fds[2]) is used to ask the kernel to create a synchronous unidirectional communication channel b/w two processes
    - fds[0]: The read end of the pipe
    - fds[1]: The write end of the pipe

Limitation of unnamed pipes

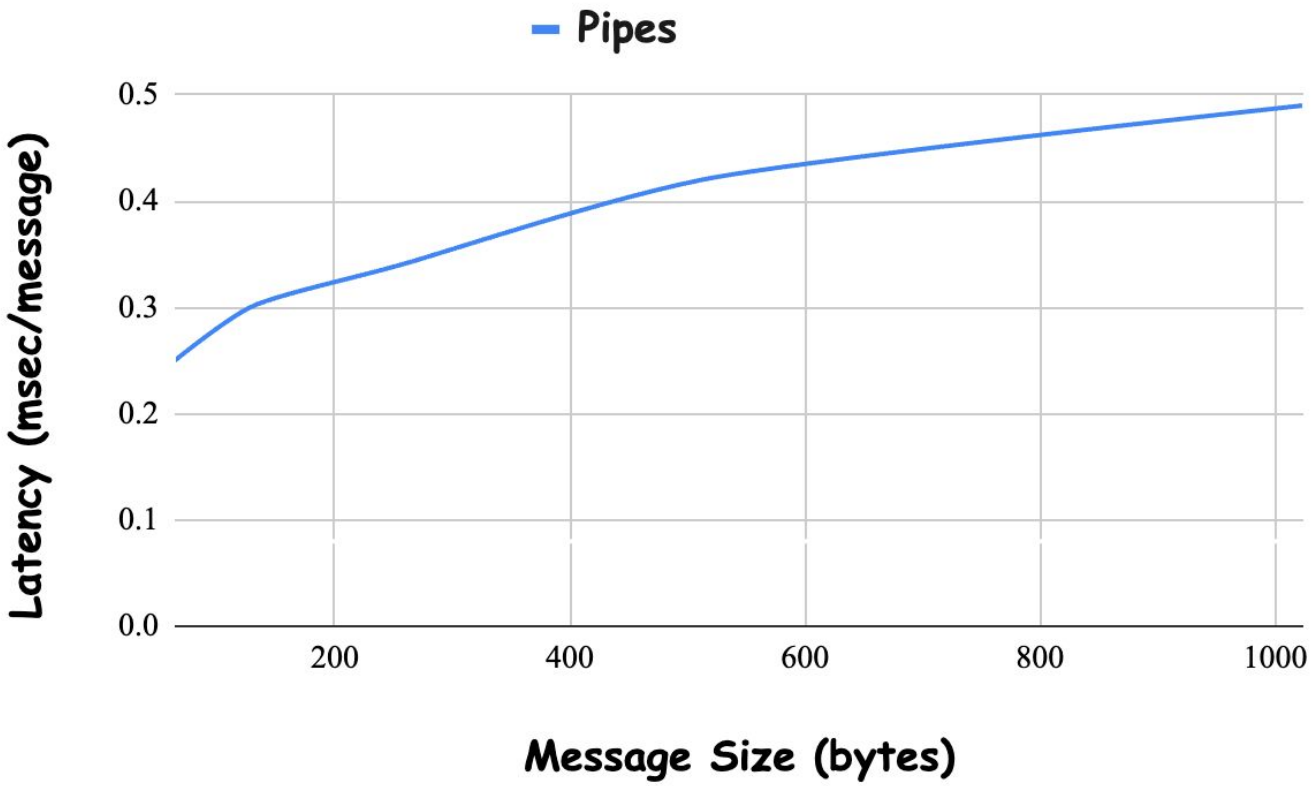
- The channel is unidirectional
- The channel can only be established b/w descendant processes

- Bidirectional channels? See int socket(int domain, int type, int protocol)
- System-wide visible pipes? See int mkfifo(char *pathname, mode_t mode)

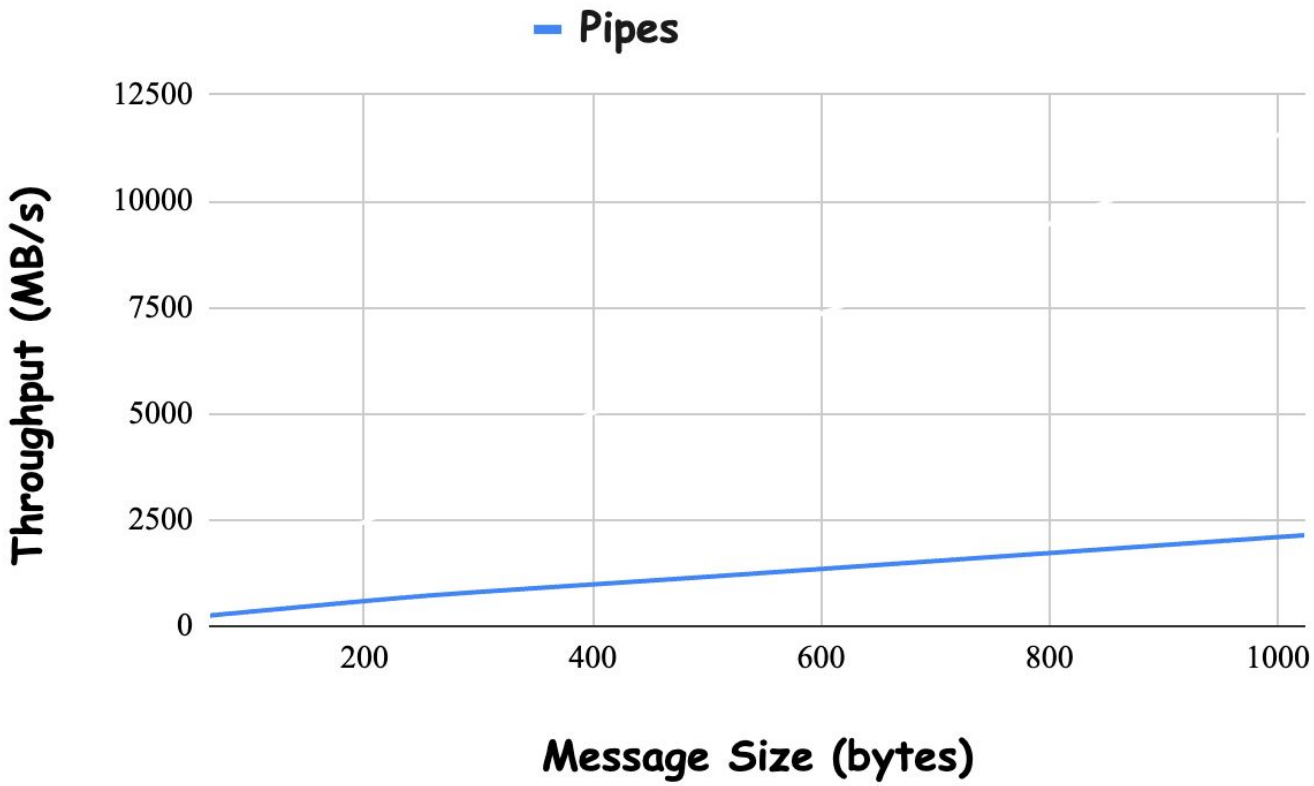
Synchronous IPC: Pipes (unnamed)



Synchronous IPC: Pipes (unnamed)



Synchronous IPC: Pipes (unnamed)



Synchronous IPC: Shared memory

› Pipe-based IPC (the tradeoff: what you get / what you lose…)

- Little responsibility ⇒ Wait for a message to be delivered
- **Slow**: Two copies (in/out) are required on every message exchanged

› How can processes avoid this overhead? **Ask the kernel to point directly to a memory region that is shared among processes (shared memory)**

- int shmget(key_t **key**, size_t size, ...)
  - Creates a shared memory (shm) segment of "size", associated with "key"
  - Returns the shm identifier
- int shmat(int **shmid**, const void *shmaddr, ...)
  - Attaches the shm segment identified by shmid to the VAS of the calling process
  - If shmaddr is NULL, the OS chooses an unused virt. address to attach the segment

Synchronous IPC: Shared memory

› Pipe-based IPC (the tradeoff: what you get / what you lose…)

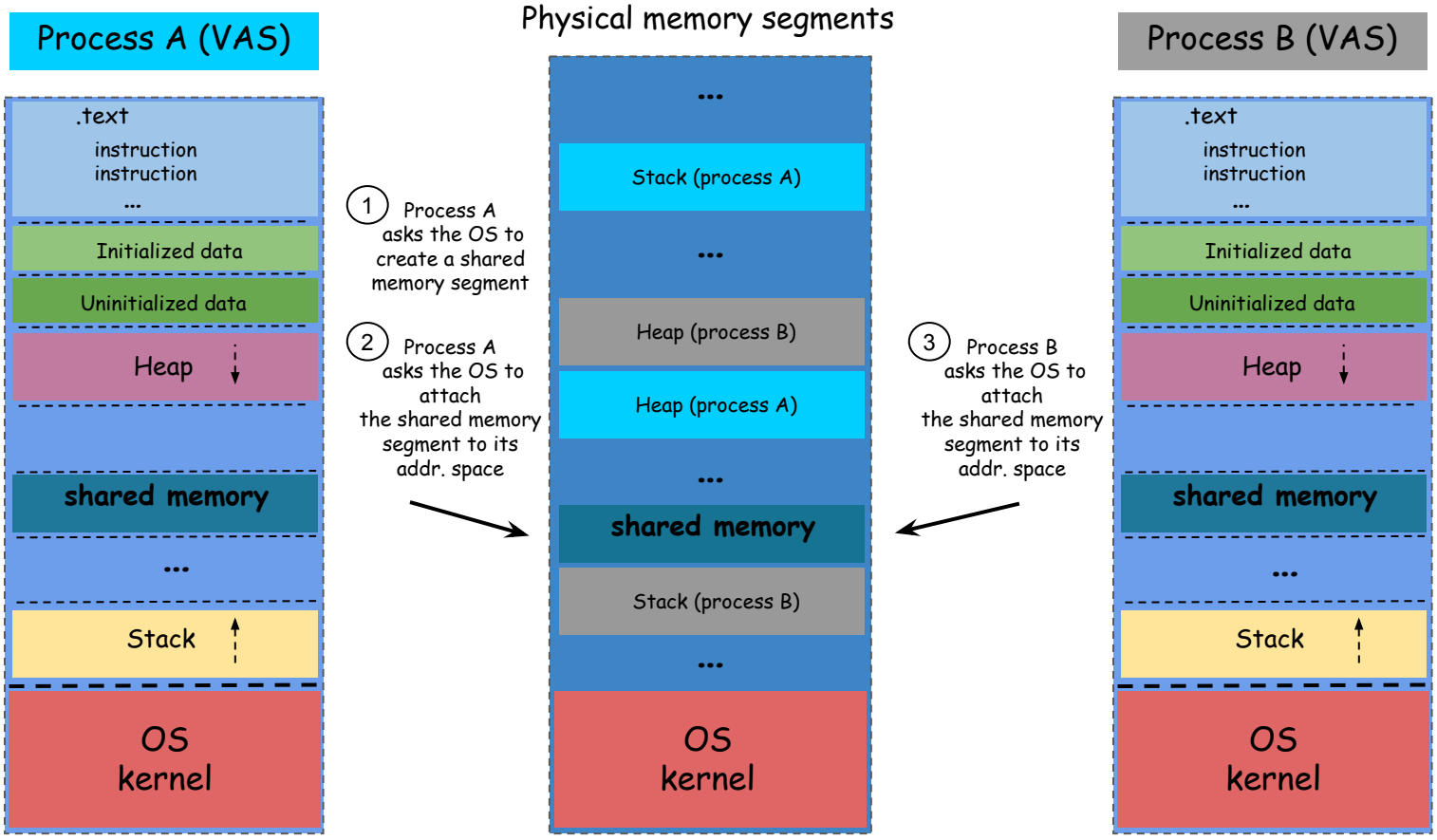
- Little responsibility ⇒ Wait for a message to be delivered
- **Slow**: Two copies (in/out) are required on every message exchanged

› How can processes avoid this overhead? **Ask the kernel to point directly to a memory region that is shared among processes (shared memory)**

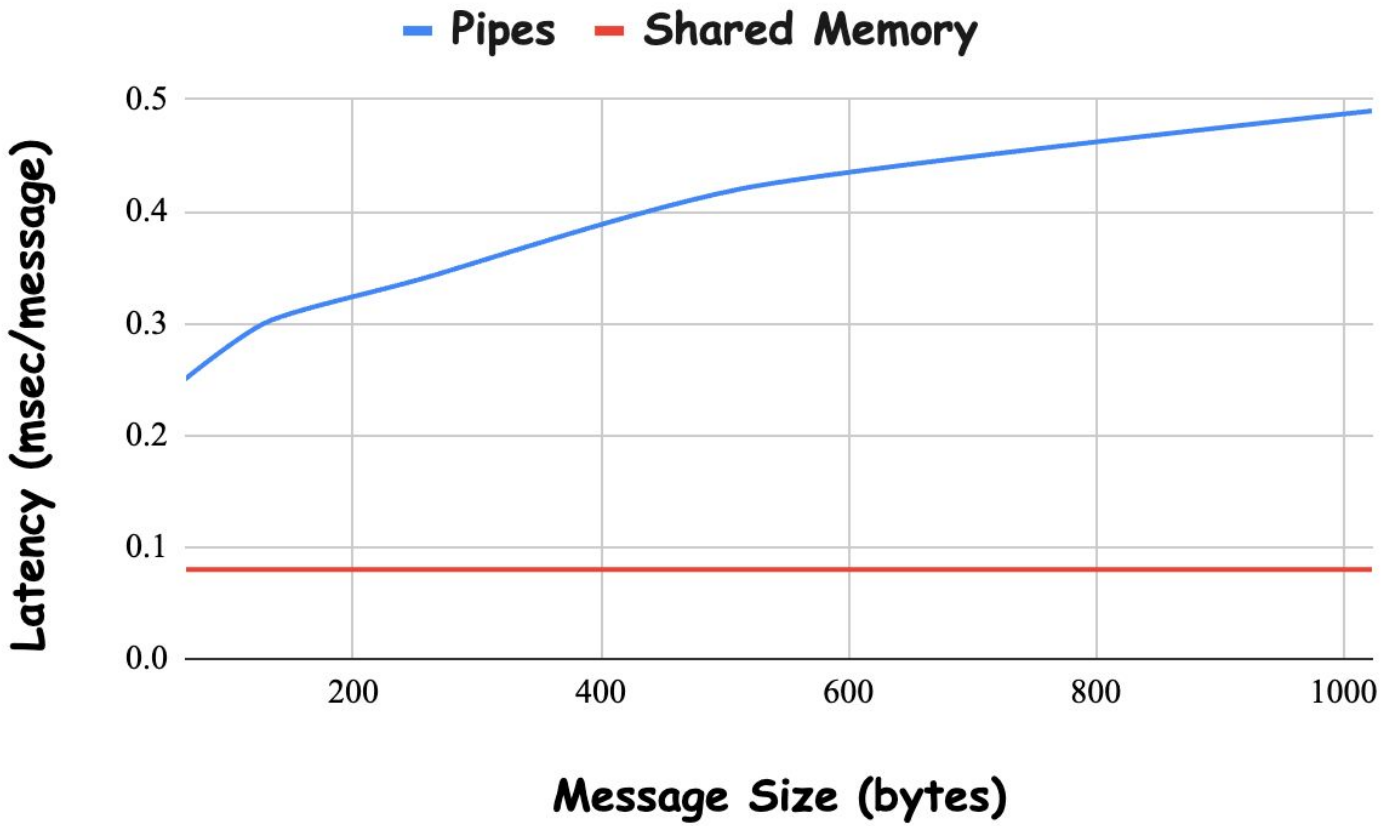
› Zero unnecessary copies

- New powers ⇒ New responsibilities
- Synchronization ⇒ More on this drama later

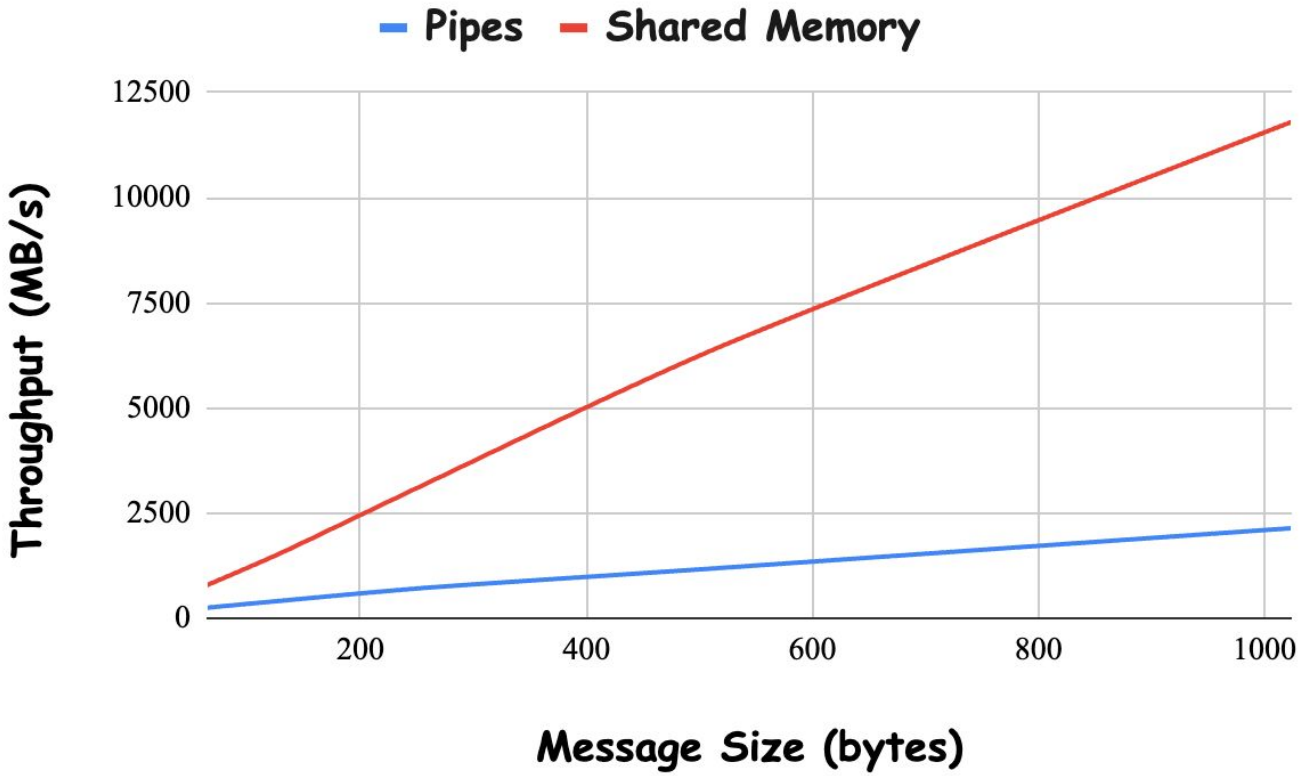
Synchronous IPC: Shared memory



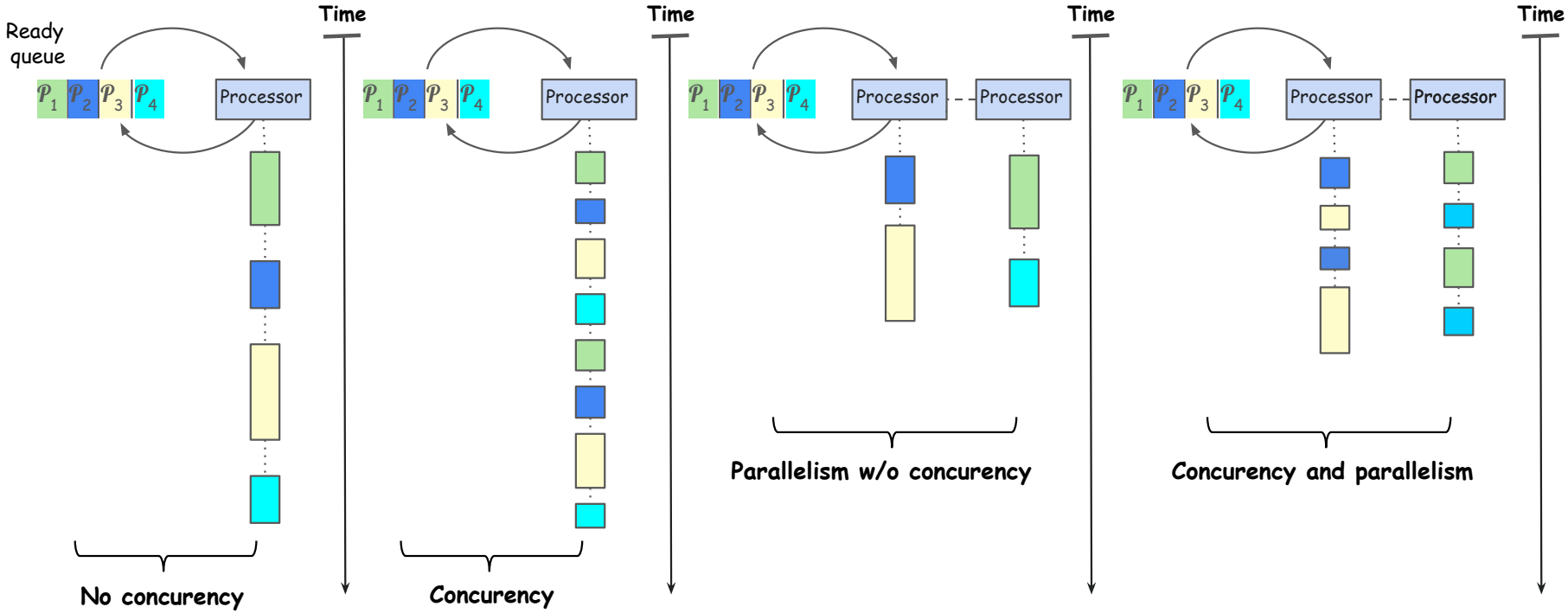
Synchronous IPC: Shared memory vs Pipes



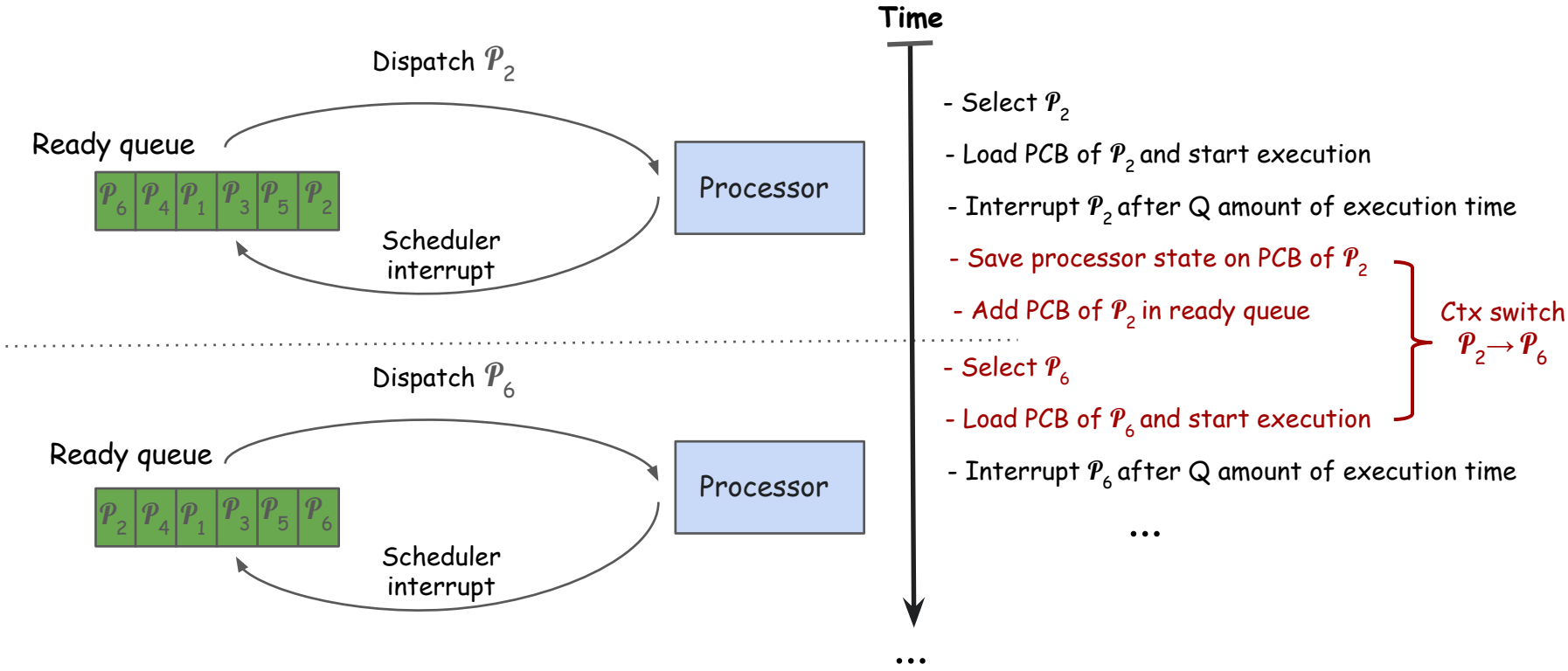
Synchronous IPC: Shared memory vs Pipes



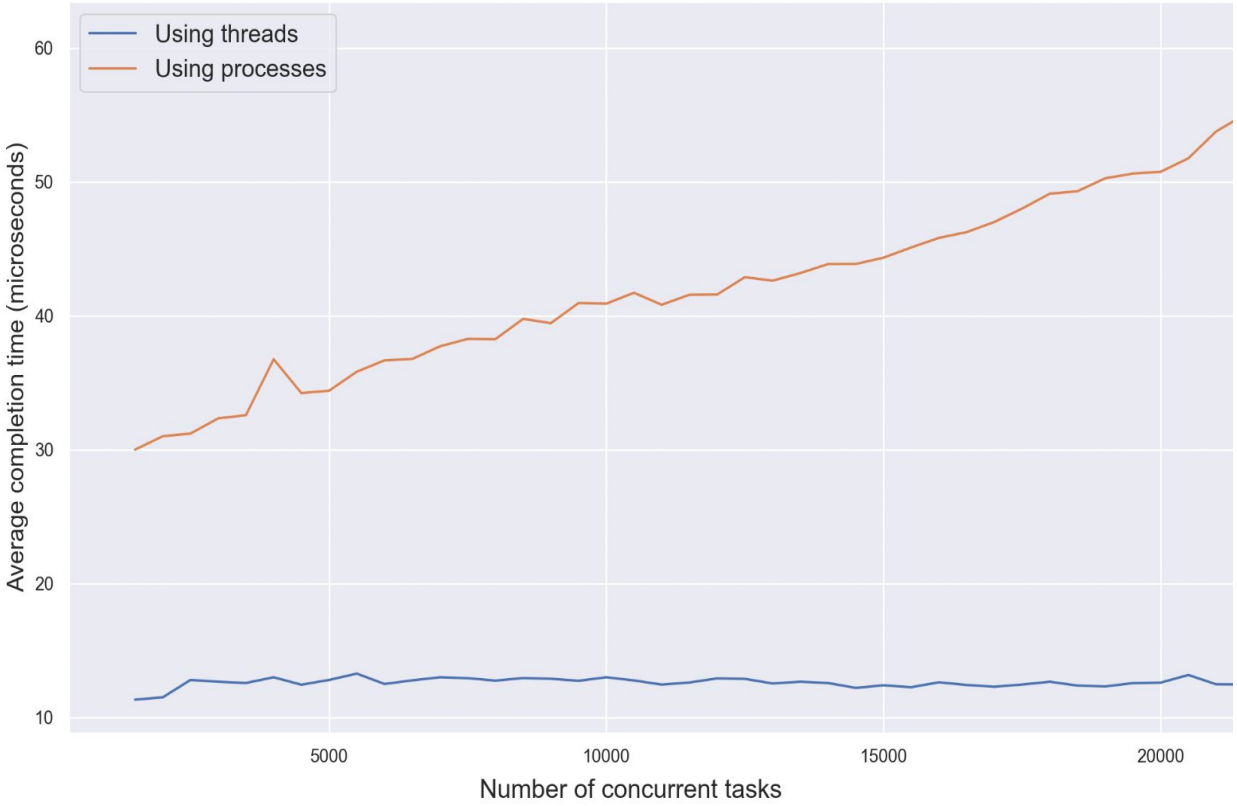
A primer on concurrency



Process dispatching



Threads vs. Processes




```
void process_task(int num){
  is_prime(num);
  _exit(0);
}

int main(int argc, char **argv) {
  ...
 for (int i = 0; i < num_tasks; i++)
    numbers[i] = rand() % 10000000;
 ...
 for (int i = 0; i < num_tasks; i++) {
    pid_t pid = fork();
    if (pid == 0)
        process_task(numbers[i]);
 }

 for (int i = 0; i < num_tasks; i++)
    wait(NULL);
}
```

The thread abstraction

- What is a thread? "A single flow of control within a process." (Strict POSIX definition, 3/190.)

- What is a process? "An address space with one or more threads executing within it." (Strict POSIX definition, 3/189.)

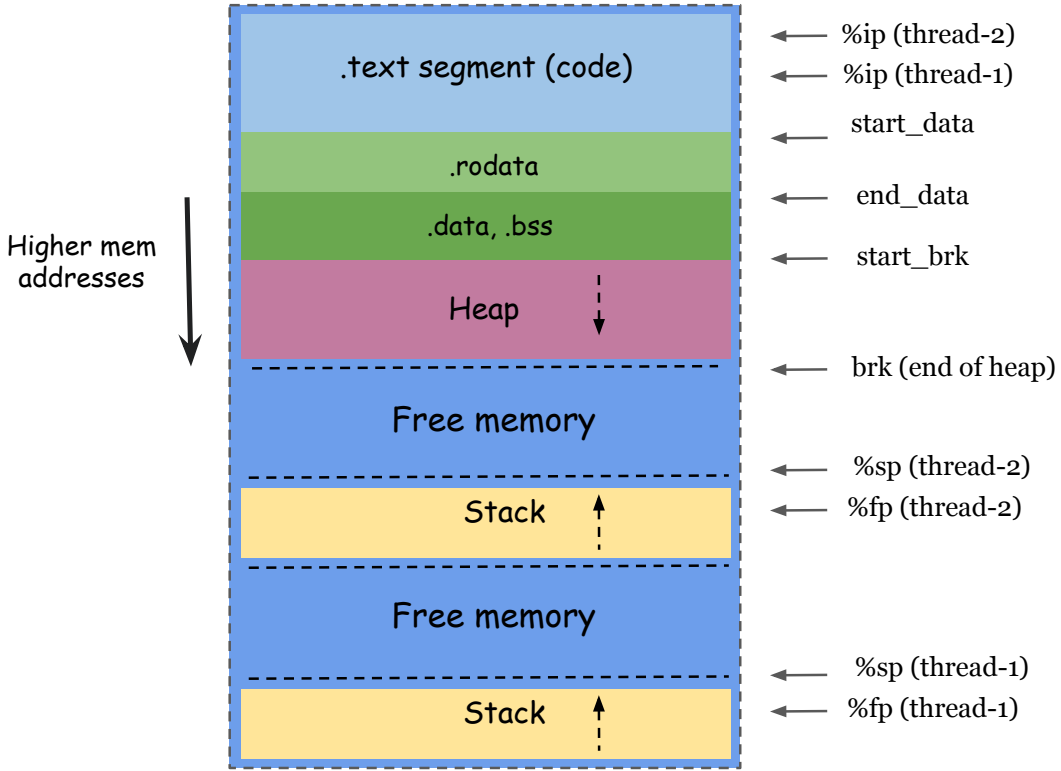
› All threads of a process share

- The code, data, and heap segments
- Shared system resources allocated to their process

› Each thread has its own

- Status (e.g., ready, running, or waiting)
- Execution state (i.e., processor registers)
- Thread-specific portion of the stack

Multithreaded process VAS



Multithreaded process VAS

```c
#define NUM_THREADS 3

void print_stack_pointer(int thread_id) {
  uint64_t  sp;
  asm volatile ("mov %0, sp" : "=r" (sp));
  printf("[tid: %d]; sp: 0x%lx, &sp: %p\n",
         thread_id, sp, &sp);
}

void* foo(void* arg) {
  int thread_id = *(int*) arg;
  print_stack_pointer(thread_id);
  return NULL;
}

void main() {

  int thread_ids[NUM_THREADS];
  pthread_t threads[NUM_THREADS];

  for (int i = 0; i < NUM_THREADS; i++) {
    thread_ids[i] = i + 1;
    pthread_create(&threads[i], NULL, foo,
                   &thread_ids[i]);
  }

 for (int = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
}
```

--- before threads creation ----

```
aaaab0510000-aaaab0512000 r-xp …      thread_vmas
aaaab0521000-aaaab0522000 r--p …      thread_vmas
aaaab0522000-aaaab0523000 rw-p …      thread_vmas
aaaad9392000-aaaad93b3000 rw-p …      [heap]
ffff8d080000-ffff8d208000 r-xp …      /usr/lib/libc.so.6
…
ffff8d27e000-ffff8d27f000 r-xp …      [vdso]
ffff8d27f000-ffff8d281000 r--p …      /usr/lib/ld-linux-aarch64.so.1
ffff8d281000-ffff8d283000 rw-p …      /usr/lib/ld-linux-aarch64.so.1
ffffd83b9000-ffffd83da000 rw-p …      [stack]
```

[tid: 0]; **sp: 0xffffd83b91a0, &sp: 0xffffd83b91b0**

--- after threads creation ----

```
aaaab0510000-aaaab0512000 r-xp …   thread_vmas
aaaab0521000-aaaab0522000 r--p …   thread_vmas
aaaab0522000-aaaab0523000 rw-p …   thread_vmas
aaaad9392000-aaaad93b3000 rw-p …   [heap]
ffff8c060000-ffff8c070000 ---p ……
ffff8c070000-ffff8c870000 rw-p …
ffff8c870000-ffff8c880000 ---p …
ffff8c880000-ffff8d080000 rw-p …
ffff8d080000-ffff8d208000 r-xp …      /usr/lib/libc.so.6
…
ffff8d27e000-ffff8d27f000 r-xp …      [vdso]
ffff8d27f000-ffff8d281000 r--p …      /usr/lib/ld-linux-aarch64.so.1
ffff8d281000-ffff8d283000 rw-p …      /usr/lib/ld-linux-aarch64.so.1
ffffd83b9000-ffffd83da000 rw-p …      [stack]
```

[tid: 2]; **sp: 0xffff8c86e810, &sp: 0xffff8c86e830**
[tid: 1]; **sp: 0xffff8d07e810, &sp: 0xffff8d07e830**

The thread abstraction

- What is a thread? "A single flow of control within a process." (Strict POSIX definition, 3/190.)

- What is a process? "An address space with one or more threads executing within it." (Strict POSIX definition, 3/189.)

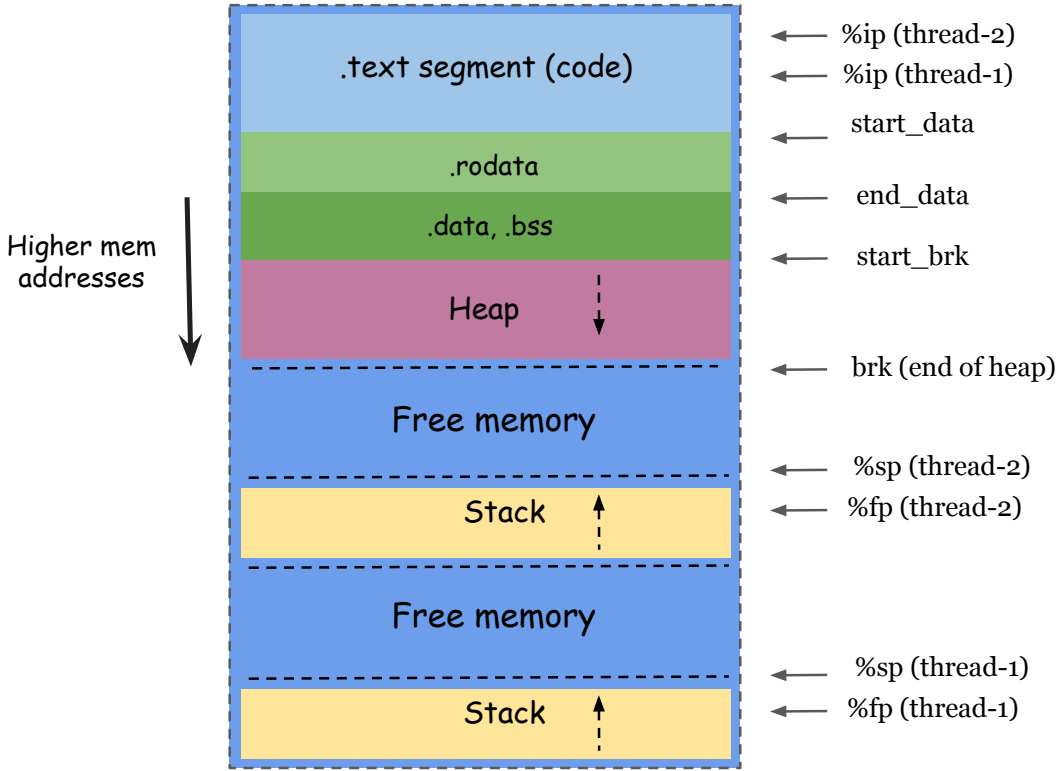
> Each thread has its own

- Stack

> All threads of a process share

- The code, data, and heap segments
- Shared system resources allocated to their process

Multithreaded process VAS



Multithreaded process VAS

```c
#define NUM_THREADS 3

void print_stack_pointer(int thread_id) {
  uint64_t  sp;
  asm volatile ("mov %0, sp" : "=r" (sp));
  printf("[tid: %d]; sp: 0x%lx, &sp: %p\n",
         thread_id, sp, &sp);
}

void* foo(void* arg) {
  int thread_id = *(int*) arg;
  print_stack_pointer(thread_id);
  return NULL;
}

void main() {

  int thread_ids[NUM_THREADS];
  pthread_t threads[NUM_THREADS];

  for (int i = 0; i < NUM_THREADS; i++) {
    thread_ids[i] = i + 1;
    pthread_create(&threads[i], NULL, foo,
                   &thread_ids[i]);
  }

  for (int = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
}
```

--- before threads creation ----

aaaab0510000-aaaab0512000 r-xp … thread_vmas
aaaab0521000-aaaab0522000 r--p … thread_vmas
aaaab0522000-aaaab0523000 rw-p … thread_vmas
aaaad9392000-aaaad93b3000 rw-p … [heap]
ffff8d080000-ffff8d208000 r-xp … /usr/lib/libc.so.6
…
ffff8d27e000-ffff8d27f000 r-xp … [vdso]
ffff8d27f000-ffff8d281000 r--p … /usr/lib/ld-linux-aarch64.so.1
ffff8d281000-ffff8d283000 rw-p … /usr/lib/ld-linux-aarch64.so.1
ffffd83b9000-ffffd83da000 rw-p … [stack]

[tid: 0]; **sp: 0xffffd83b91a0, &sp: 0xffffd83b91b0**

--- after threads creation ----

aaaab0510000-aaaab0512000 r-xp … thread_vmas
aaaab0521000-aaaab0522000 r--p … thread_vmas
aaaab0522000-aaaab0523000 rw-p … thread_vmas
aaaad9392000-aaaad93b3000 rw-p … [heap]
ffff8c060000-ffff8c070000 ---p ……
ffff8c070000-ffff8c870000 rw-p …
ffff8c870000-ffff8c880000 ---p …
ffff8c880000-ffff8d080000 rw-p …
ffff8d080000-ffff8d208000 r-xp … /usr/lib/libc.so.6
…
ffff8d27e000-ffff8d27f000 r-xp … [vdso]
ffff8d27f000-ffff8d281000 r--p … /usr/lib/ld-linux-aarch64.so.1
ffff8d281000-ffff8d283000 rw-p … /usr/lib/ld-linux-aarch64.so.1
ffffd83b9000-ffffd83da000 rw-p … [stack]

[tid: 2]; **sp: 0xffff8c86e810, &sp: 0xffff8c86e830**
[tid: 1]; **sp: 0xffff8d07e810, &sp: 0xffff8d07e830**

The thread abstraction

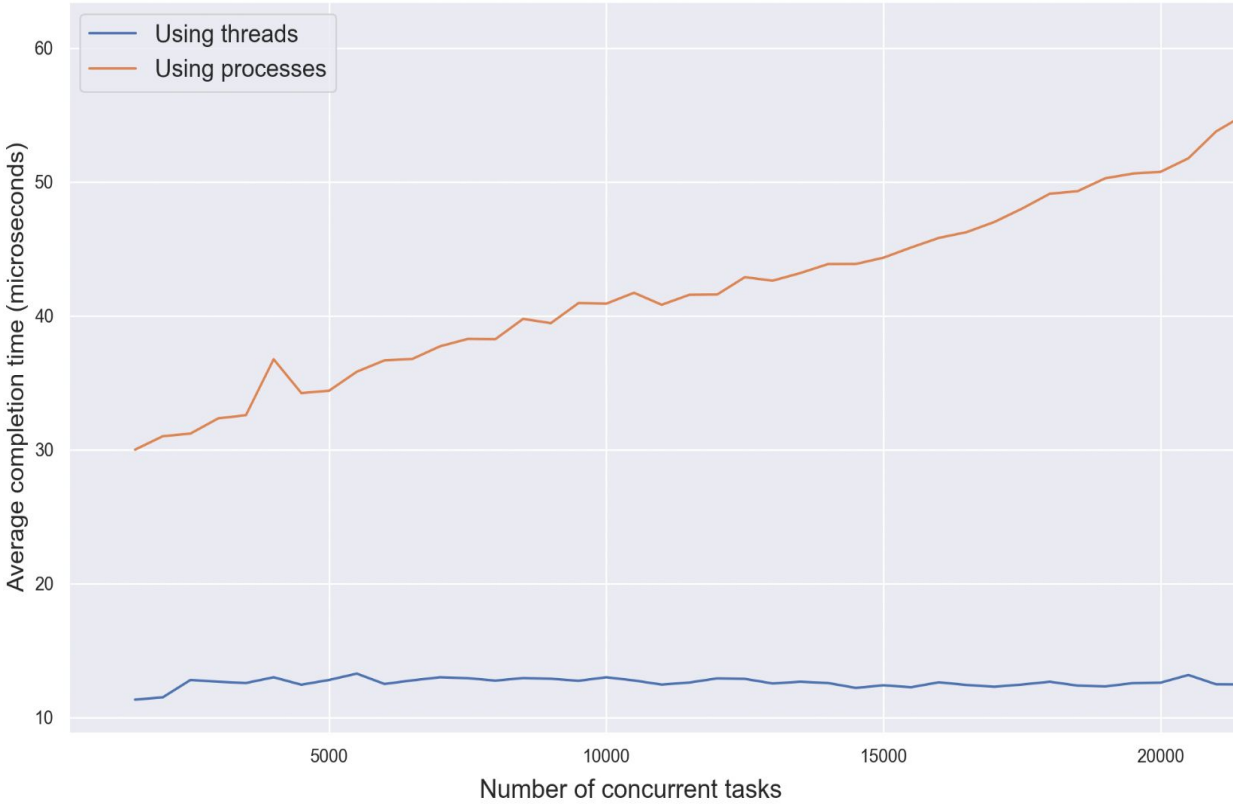
- What is a thread? "A single flow of control within a process." (Strict POSIX definition, 3/190.)

- What is a process? "An address space with one or more threads executing within it." (Strict POSIX definition, 3/189.)

> **Unlike processes**

- Thread creation is inexpensive (no need to duplicate the addr. space)
- Switching between threads of the same process is inexpensive
  - Same address space / context switch ⇒ TLB remains hot
- Communication b/w threads of the same process is inexpensive
  - Can be implemented with no OS intervention
- New powers ⇒ New responsibilities…

Threads vs. Processes




```
void process_task(int num){
  is_prime(num);
  _exit(0);
}

int main(int argc, char **argv) {
  ...
 for (int i = 0; i < num_tasks; i++)
    numbers[i] = rand() % 10000000;
 ...
 for (int i = 0; i < num_tasks; i++) {
    pid_t pid = fork();
    if (pid == 0)
        process_task(numbers[i]);
 }

 for (int i = 0; i < num_tasks; i++)
    wait(NULL);
}
```

How does the OS implement threads

- POSIX-compliant OSes need to manage both threads and processes
- Easy to support threads, if already supporting processes
  - Scheduling decisions ⇒ On threads
  - Address space decisions ⇒ On processes
  - Book-keeping decisions ⇒ On processes (modulo execution state)

---

do {
Get a process $\mathcal{P}$ from ready queue
Change to the new address space
Execute $\mathcal{P}$ until time $\mathcal{Q}$ expires
Put $\mathcal{P}$ back in ready queue
} while(1)

do {
Get a thread $\mathcal{T}$ from ready queue
Change address space, **if needed**
Execute $\mathcal{T}$ until time $\mathcal{Q}$ expires
Put $\mathcal{T}$ back in ready queue
} while(1)

How does the OS implement threads

- POSIX does not dictate whether threads should have their own schedulable context and run simultaneously on different processors
- Threads can be implemented as user-level or kernel-level threads

› User-level threads (Solaris "green" threads, java threads)

  - Multiple threads are mapped to one schedulable kernel context
  - No real concurrency: One thread blocks ⇒ All process' threads block
  - The process is the only kernel scheduled entity
  - Only one syscall per time ⇒ One kernel stack per process
  - Faster to create (syscalls: ~70 cycles >> procedure calls: ~5 cycles)

How does the OS implement threads

- POSIX does not dictate whether threads should have their own schedulable context and run simultaneously on different processors
- Threads can be implemented as user-level or kernel-level threads

› kernel-level threads (e.g., glibc pthreads in Linux)

  - Slower to create and interact with (glibc uses clone3 syscall)
  - Each thread is mapped to one schedulable kernel context (1:1)
  - Integrated with OS scheduling decisions
  - One thread blocks ⇒ the OS will schedule another

Watch out: Kernel-level thread ≠ Kernel-space thread (kthread)

How does the OS implement threads

- POSIX does not dictate whether threads should have their own schedulable context and run simultaneously on different processors
- Threads can be implemented as user-level or kernel-level threads

> kernel-level threads (e.g., glibc pthreads in Linux)

  - Slower to create and interact with (glibc uses clone3 syscall)
  - Each thread is mapped to one schedulable kernel context (1:1)
  - Integrated with OS scheduling decisions
  - One thread blocks ⇒ the OS will schedule another

Read the scheduler activations paper for more on a hybrid approach

Historical Evolution: From uniprogramming to multitasking

- Uniprogramming: Load a program in memory and execute it to completion
  - Human operator acts as the dispatcher
  - **while (jobs) {**
    load a program in memory
    execute to completion
    }
  - Simple idea: The OS is just a library of device drivers for primitive hardware resources
  - OK idea for the 70's mainframes
    - Resource underutilization: one job blocks, everyone waits
    - Leads to bad throughput: job completion per unit of time

Historical Evolution: From uniprogramming to multitasking

- Multiprogramming: Multiple jobs in memory, at the same time
  - The OS gives the processor to a new process every time the current process needs to block (e.g., while waiting for I/O)
  - No strict time allocation: A process may keep running until it blocks, or to completion, or indefinitely
  - Functionality required
    - Virtual memory (fault isolation to keep many procs in mem.)
    - Interrupts (for async events; e.g., to support disk DMAs)

› Improves processor utilization ⇒ Better throughput

› Violates one of the three OS desirable properties [Q: Which?]

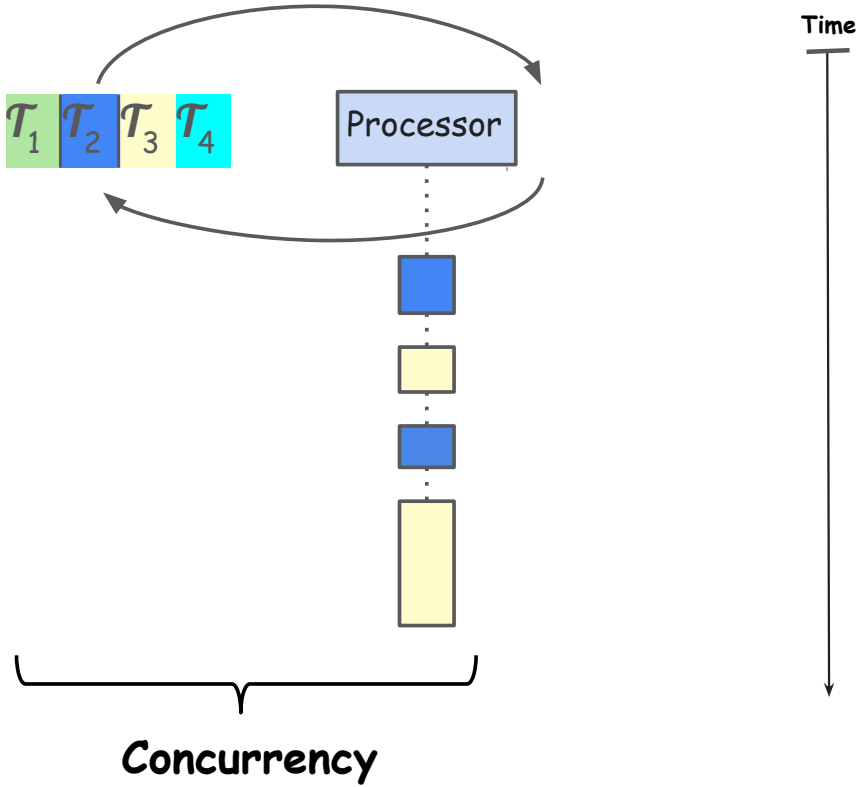
Historical Evolution: From uniprogramming to multitasking

- Multitasking: Multiple jobs in memory, at the same time, the OS allocates the processor to each of them with an upper bound
  - Implemented with preemptive scheduling (e.g., timesharing)
  - Each processor has a dedicated timer which expires periodically
    - The OS takes control ⇒ Inspects execution statistics
    - Decides where to allocate the processor next
  - Fast switching gives tasks the illusion of a dedicated processor

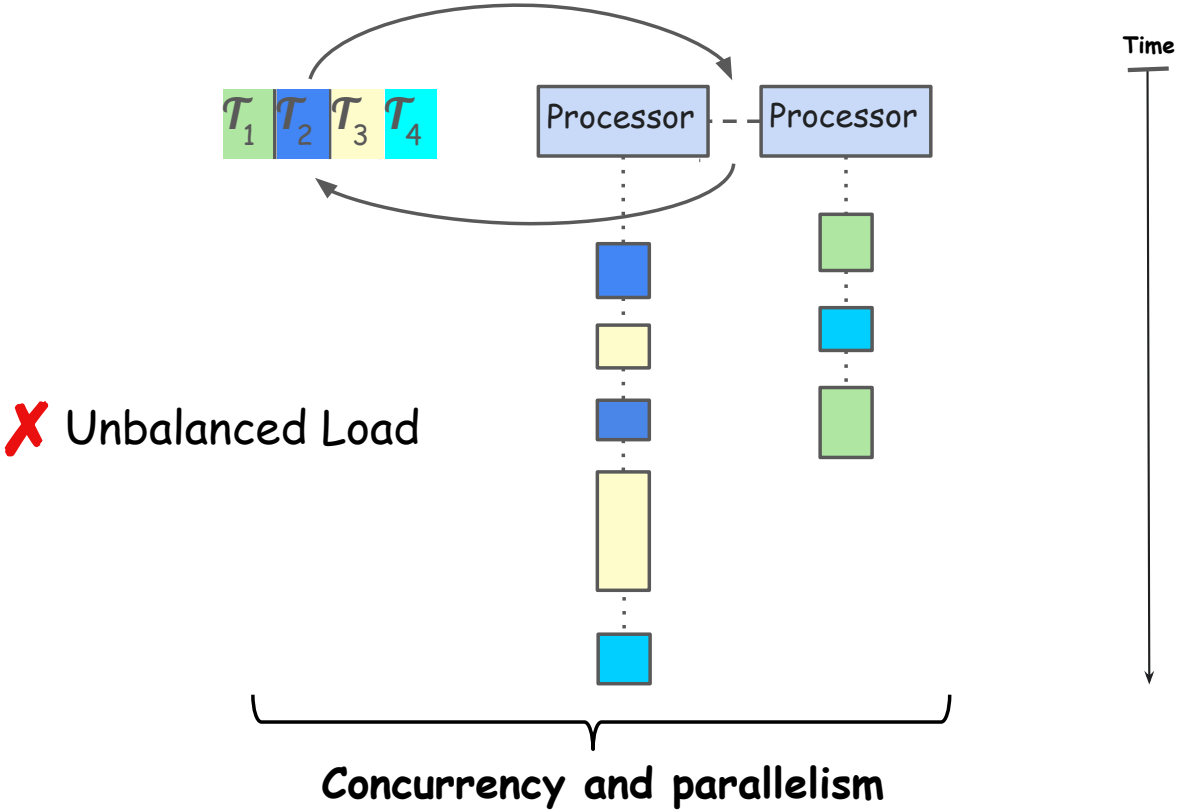
› Improves processor utilization ⇒ Better throughput

› Improves system responsiveness ⇒ Immediate feedback to users

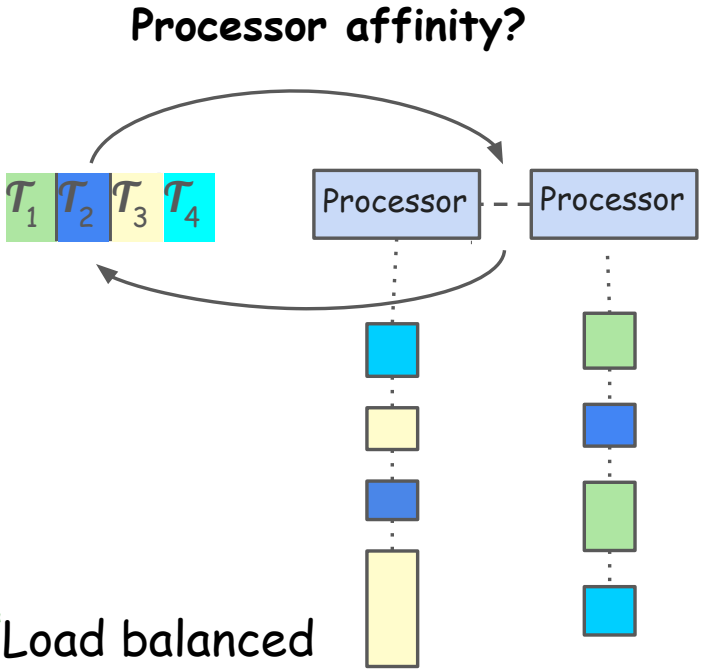
Multitasking on one processor



Multitasking on Symmetric Multiprocessor (SMP)

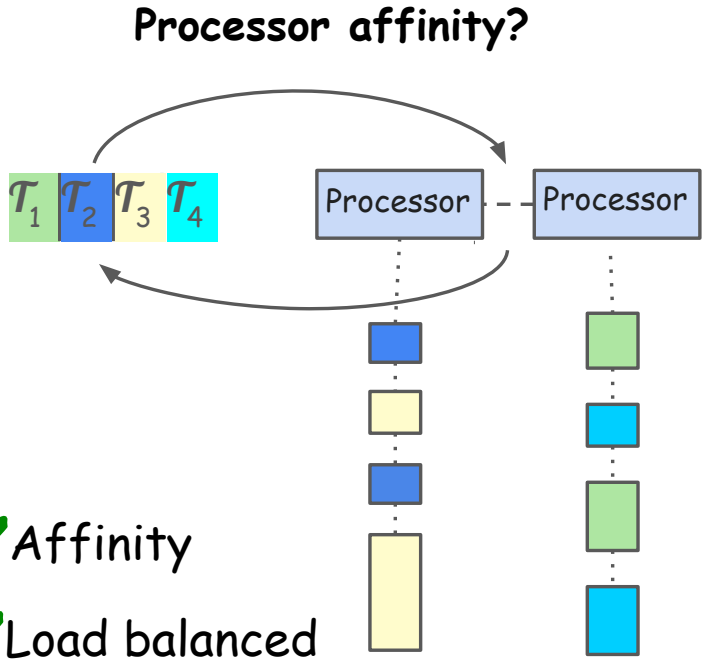


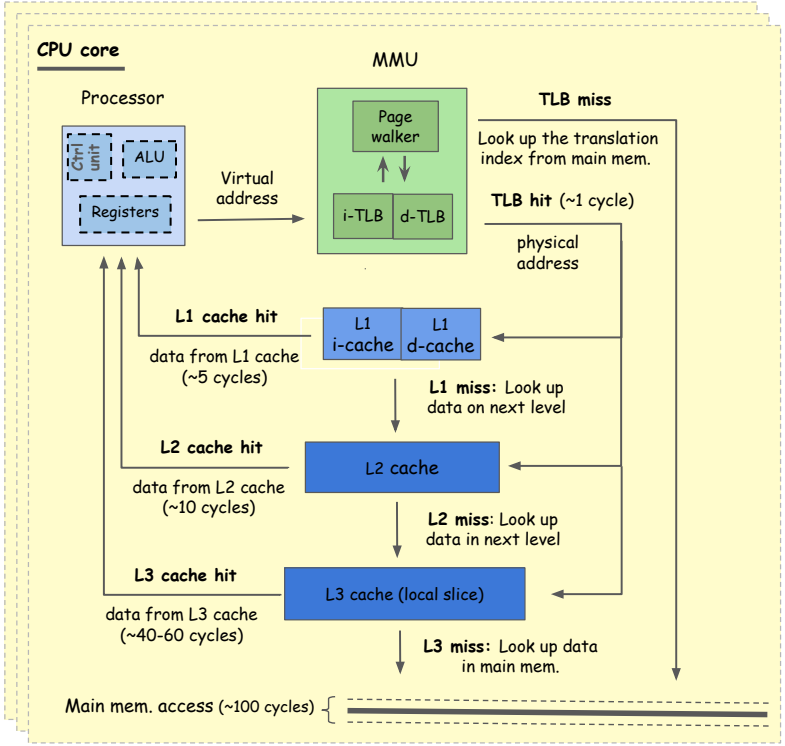
Multitasking on SMP and load balancing



✔Load balanced

Multitasking on SMP and load balancing w/ affinity





Implementing OS support for concurrency

- Concurrency *(distributed systems concept)*: Multiple operations run seemingly simultaneously on shared hardware resources
- Multitasking *(OS implementation of concurrency)*: Multiple tasks run for at most a time quantum (timesharing), and either yield the processor (voluntarily ctx switch) or get preempted by the OS (involuntarily ctx switch)
- Small quantum (10–100 ms) + Multiple processors
  - Different processors execute different tasks in parallel
  - Each processor concurrently executs many tasks per second

Concurrency + Parallelization

> Concurrency + Parallelization is the desirable execution paradigm ♡

- *Responsiveness*: No task stays blocked for perceptibly long
- *Throughput*: No single task can monopolize the processor ⇒ Allowing, overall, more tasks to complete in a unit of time
- *Scalability*: More hardware resources available, more processors available to handle tasks concurrently in a unit of time

> Programmer's responsibilities:

- Divide and conquer: Split code in small routines of independent tasks
- Avoid shared state ⇒ Avoid coordination / locks

Necessary glossary to talk about *synchronization*

- **Parallel operations:** Operations that are happening at the same time, on different processors
- **Concurrent operations:** Operations that are happening in overlapping time intervals, seemingly simultaneously
- **Interleaving of execution:** The order with which concurrent operations are scheduled in for and out of execution
- **Happens-before relationship?**

What is *synchronization*?

Given two concurrent operations p1, p2 with a "dependency" such that p1 must always "happen before" p2,
synchronization mandates that $t1(i) < t2(i)\ \forall\ i \in [1, n]$,
where

- t1(i) is the time when p1 ends its i-th execution
- t2(i) it the time when p2 starts its i-th execution

> We have a problem in the above definition…

Departing from temporal ordering

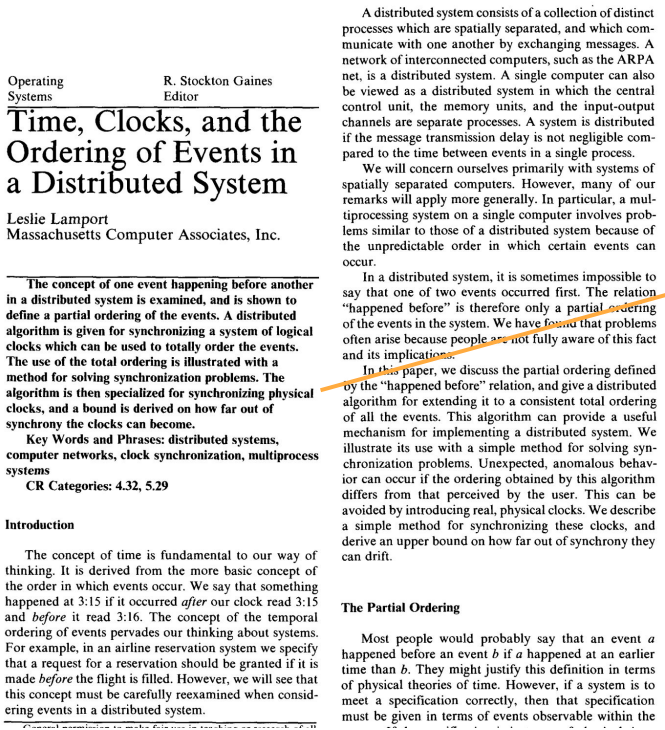
> Temporal ordering: Arrangement of events in a sequence according to physical time

- What most human understand when you talk about time!
- **Example:** An airline reservation request will be granted if (i) it is made before the flight is filled, and (ii) before the flight departs

> Ordering is not temporal on a multiprocessor (distributed) system

- Any conversation is in terms of physical time must be reexamined when considering concurrent events in a distributed system
- Real clocks are not perfectly accurate ⇒ can't keep precise phys. time

"happens before" on distributed systems



Operating Systems — R. Stockton Gaines, Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems
CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event *a* happened before an event *b* if *a* happened at an earlier time than *b*. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

> Time, Clocks, and the Ordering of Events in a Distributed System, 1978, by Leslie Lamport

Partial ordering of concurrent operations

> Logical clocks allows us to define a partial ordering of concurrent operations on a multiprocessor system

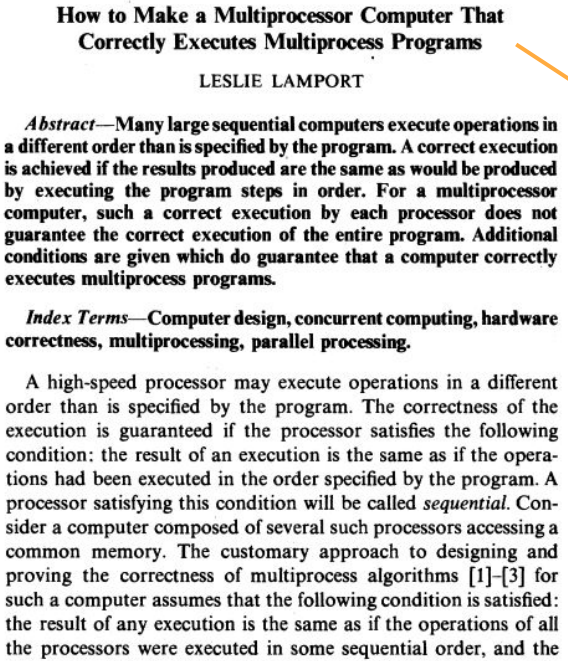
> Synchronization is used to enforce that a partial order of concurrent operations (i.e., some "happens-before" relationship) exists

Given two concurrent operations p1, p2
with a "dependency" such that p1 must always "happen before" p2,
synchronization mandates that $t1(i) < t2(i)\ \forall\ i \in [1, n]$, where
$t1(i)$ is the time when p1 ends its i-th execution
$t2(i)$ is the time when p2 starts its i-th execution

Necessary glossary to talk about *synchronization*

- **Parallel operations:** Operations that are happening at the same time, on different processors
- **Concurrent operations:** Operations that are happening in overlapping time intervals, seemingly simultaneously
- **Interleaving of execution:** The order with which concurrent operations are scheduled in for and out of execution
- **Happens-before relationship:** A *partial ordering* of concurrent operations of a program
- **Sequential consistency?**

Reasoning about sequential consistency



How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

Index Terms—**Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.**

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the execution is guaranteed if the processor satisfies the following condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called *sequential*. Consider a computer composed of several such processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms [1]–[3] for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

> How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, 1977, by Leslie Lamport

Reasoning about sequential consistency

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

Index Terms—**Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.**

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the execution is guaranteed if the processor satisfies the following condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called *sequential*. Consider a computer composed of several such processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms [1]–[3] for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the

Abstract—**Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.**

of each individual processor does not guarantee that the multiprocessor computer is sequentially consistent. In this brief note, we describe a method of interconnecting sequential processors with memory modules that insures the sequential consistency of the resulting multiprocessor.

> How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, 1977, by Leslie Lamport

Necessary glossary to talk about *synchronization*

- **Parallel operations:** Operations that are happening at the same time, on different processors
- **Concurrent operations:** Operations that are happening in overlapping time intervals, seemingly simultaneously
- **Interleaving of execution:** The order with which concurrent operations are scheduled in for and out of execution
- **Happens-before relationship:** A *partial ordering* of concurrent operations of a program
- **Sequential consistency:** The result of any execution of concurrent operations is the same as if all operations on all processors were executed in some sequential (global) order, and the operations of each individual processor appear in this sequence in the order specified by its program

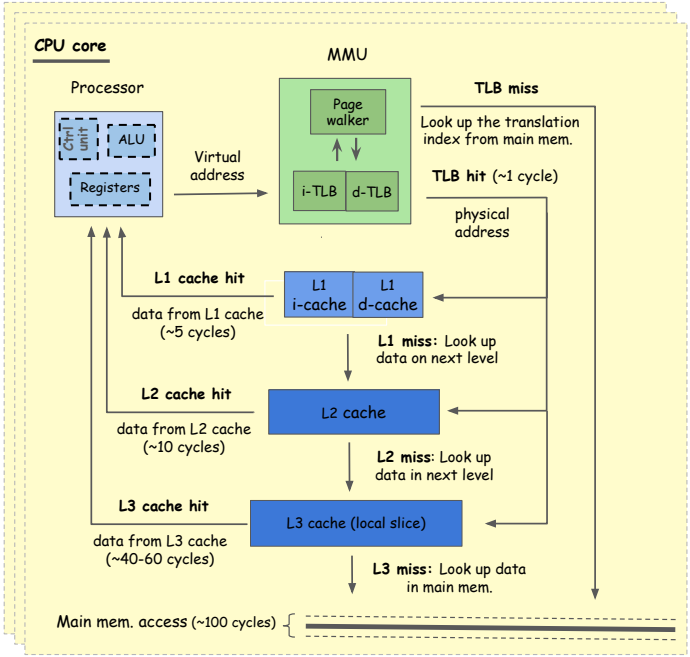
Sequential consistency

Pragmatic explanation

- Processors issue loads and stores in their local memory respecting their local program order
- Every load from a memory address gets its value from the last store before it, on the same memory address, in global memory

Sequential consistency: Requirements

- **Sequential consistency:** Every load from a memory address would get its value from the last store before it to the same address in **global memory**



> Easy to reason / Impractically slow

- The effects of each instruction must be visible on all cores before starting the next instruction

- The first level of "global" memory is the L3 cache with an overhead of at least 40 cycles for access time

- In practice: We relax the memory consistency model to hide store (write) latency and avoid processor stalls

"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized, concurrent operations, at least one of which mutating shared state
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operation is not what the programer expected

What is so hard about correct concurrent code?

> Concurrent progs have too many execution interleavings

- Too many ways something erroneous could happen
- Need to explore an enormous state space

> Correctness needs a definite and **complete** answer

- Inspected 100% of the state space ⇒ Can make an assessment
- Inspect 99.9% of the state space ⇒ Can't make any assessment

How many is "too many"?

> Concurrent progs have too many execution interleavings

- Too many ways something erroneous could happen
- Need to explore an enormous state space

> Correctness needs a definite and **complete** answer

- Inspected 100% of the state space ⇒ Can make an assessment
- Inspect 99.9% of the state space ⇒ Can't make any assessment

Permutations of the word "MISSISSIPPI"

> We are counting permutations
  - Let's do the exercise

> M-I-S-S-I-S-S-I-P-P-I
  - Length: 11, M: 1, I: 4, S: 4, P: 2
  - Distinct ways to permute a multiset of n elements, where ki is the multiplicity of the ith element?
  - Multinomial coefficient: (k1+k2+...+ kn)! / (k1!*k2!*...*kn!)
  - Permutations of MISSISSIPPI = (11!) / (1!4!4!2!) = **34,650**

How many is "too many"?

> Different schedules for four operations P1, P2, P3, and P4, which run in total 11 time quanta; and where

- P1 runs 1 time
- P2 runs 4 times
- P3 runs 4 times
- P4 runs 2 times

> How many different scheduler plans do we need to inspect, to cover the complete state space of possible interleavings? **34,650**

* Trivial example in terms of no of operations

- **We are not considering myriads of async events**
- **Yet's it's already too difficult**

A few dedicates of state space exploration

> We' ve been searching for decades ways to reduce the size of the state space of concurrent programs and test them

- Partial-Order Methods for the Verification of Concurrent Systems, in 1995, by Patrice Godefroid.
- Model checking to find serious file system errors, in 2006, by Junfeng Yang et al.
- RESTler: Stateful REST API Fuzzing, in 2019, by Atlidakis et al.

"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized concurrent accesses, at least one of which mutating a shared variable
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operations is not what the programer expected

Data races

> A program contains a data race iif two or more threads

(1) access the same memory location concurrently

AND (2) at least one of these accesses is a write

AND (3) at least one of the accesses is not atomic

AND (4) neither happens before the other

Such data races may result in undefined program behavior and may lead to unforeseen errors at runtime

- See the ISO/IEC 9899:2011(C11), Sec.-5.1.2.4/25, on multi-threaded executions and data races

Data races

```c
int total = 0;

void *add(void *arg) {
  for (int i = 0; i < 1e6; ++i)
    ++total;
  return NULL;
}

void main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Total-1: %d\n", total);

  total = 0;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t2, NULL);
  printf("Total-2: %d\n", total);
}
```

```
➜ obdjump -d ./counter

0000000000001159 <add>:

 1159: push %rbp                  # Save base pointer to stack
 115a: mov  %rsp, %rbp            # Set up new stack frame
 115d: mov  %rdi, -0x18(%rbp)     # *arg = %rdi
 1161: movl $0x0, -0x4(%rbp)      # i = 0
 1168: jmp  117d <add+0x24>       # for-loop start

                                  Data race!
 116a: mov  0x2ebc(%rip), %eax    #  %eax ← total
 1170: add  $0x1, %eax            #  %eax += 1
 1173: mov  %eax, 0x2eb3(%rip)    #  total ← %eax

 1179: addl $0x1, -0x4(%rbp)      # i += 1
 117d: cmpl $0xf423f, -0x4(%rbp)  # loop counter compare
 1184: jle  116a <add+0x11>       # for-loop jump
 1186: mov  $0x0, %eax            # rval = %eax
 118b: pop  rbp                   # Restore stack
 118c: ret                        # Return to caller
```

```
➜ git:(master) ✗ ./counter
Total-1: 1011367
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1011367
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1028085
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1011197
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1018502
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1013853
Total-2: 2000000
```

Data races

- Two domains of state for each thread
  - Processor regs (per-thread, local state) vs. Main memory (global state)
  - Processor on thread A
    - %reg <- value at main mem. [global vs. local state: consistent]
    - %reg <- %reg + 1 [global vs. local state: divergent]
    - value at main mem. <- %reg [global vs. local state: consistent]
  - Thread A gets preempted [shared value at main mem. +1]
  - Processor on thread B
    - %reg <- value at main mem. [global vs. local state: consistent]
    - %reg <- %reg + 1 [global vs. local state: divergent]
    - value at main mem. <- %reg [global vs. local state: consistent]
  - Thread B gets preempted [shared value at main mem. +1]

Data races

- Two domains of state for each thread

- Processor regs (per-thread, local state) vs. Main memory (global state)
- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]
- Thread A gets preempted [shared value at main mem. +1]
- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]
- Thread B gets preempted [shared value at main mem. +1]

OK Interleaving

Data races

> Another execution interleaving

- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]
- Processor on thread A
  - **OS job:** Load %reg with its value before ctx switch
  - Thread A "thinks" %reg ⇔ shared value at main mem.
  - **But:** shared value has been mutated by someone else

NOT OK Interleaving

Data races

> Yet, another execution interleaving

- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]
- Processor on thread B
  - **OS job:** Load %reg with its value before ctx switch
  - Thread B "thinks" %reg ⇔ shared value at main mem.
  - **But:** shared value has been mutated by someone else

NOT OK Interleaving

Data races

- **Bottom-line**: Concurrent writes on shared state?

Data races

- **Bottom-line**: Concurrent writes on shared state?
  - Each thread must finish its business before it gets preempted
  - Inseparable "instructions" ⇒ **Atomic operations**

- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]
- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]

"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized concurrent accesses, at least one of which mutating a shared variable
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operations is not what the programer expected

"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized concurrent accesses, at least one of which mutating a shared variable
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operations is not what the programer expected

Semantic ordering errors: Benign

```c
void* func1(void* arg) {
  printf("1\n");
  return NULL;
}

void* func2(void* arg) {
  printf("2\n");
  return NULL;
}

int main(void) {
  pthread_t t1, t2;

  pthread_create(&t1, NULL, func1, NULL);
  pthread_create(&t2, NULL, func2, NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return 0;
}
```

```
➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2
```

Semantic ordering errors: Detrimental

```c
int mode = 0;             // 1: Low-power; 2: High-power
int filter_engage = 0;    // 0: Filter not engaged; 1: Filter engaged
int input_finalized = 0;  // Has operator finished input?

void *filter_control(void *arg) {
  while (!input_finalized) {
    sched_yield();
  }
  usleep(100);
  if (mode == 2)
    filter_engage = 1;
  else
    filter_engage = 0;
  return NULL;
}

void beam_activate() {
  if (mode == 2 && filter_engage == 0)
    printf("🚨 Treatment with high-power beam and no filter in place\n");
  else
    printf("🟢 Safe setup\n");
}

int main(void) {
  pthread_t filter_control_t;
  pthread_create(&filter_control_t, NULL, filter_control, NULL);
  usleep(100);  // Time window 1:  operator does  initial setup
  mode = 1;
  input_finalized = 1;
  usleep(100); // Time window 2: operator does  final edits
  mode = 2;
  pthread_join(filter_control_t, NULL); // Control logic completed
  beam_activate();
}
```

```
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.

➜ concurrency git:(master) ✗ ./therac25
🚨 Treatment with high-power beam and no filter in place :-(
```

"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized concurrent accesses, at least one of which mutating a shared variable
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operations is not what the programer expected

"Problems" due to lack of synchronization

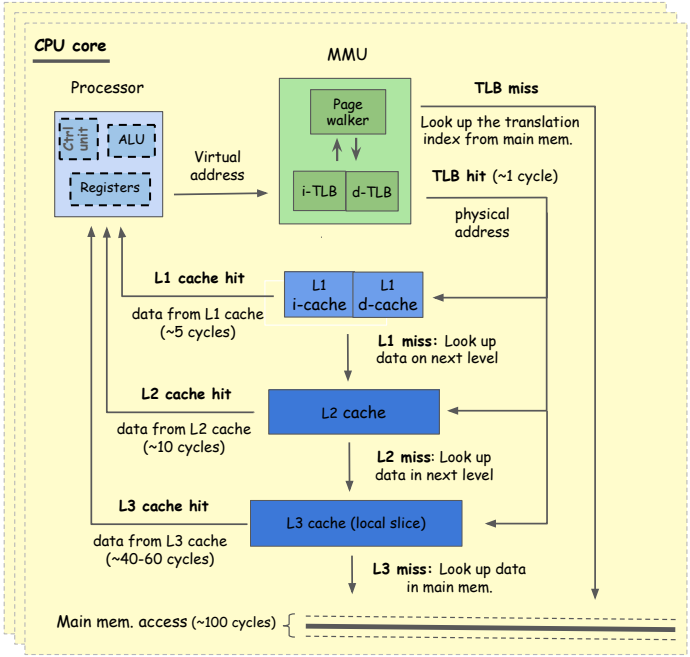
> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

- Data races: Non-atomic, unsynchronized concurrent accesses, at least one of which mutating a shared variable
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operations is not what the programer expected

Relaxing sequential consistency

- **Sequential consistency:** Every load from a memory address gets its value from the last store before it to the same address in **global memory**



> Easy to reason / Impractically slow

- The effects of each instruction must be visible on all cores before starting the next instruction

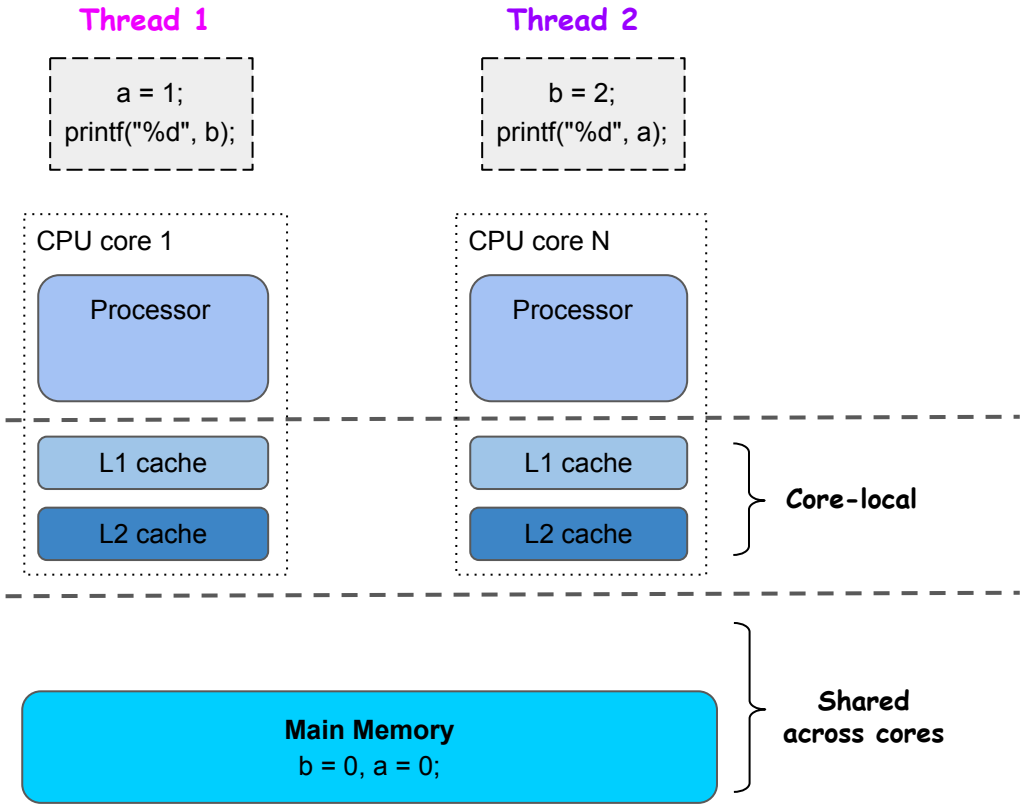
- The first level of "global" memory is the L3 cache with an overhead of at least 40 cycles for access time

- **In practice: We relax the memory consistency model to hide write latency and avoid processor stalls**

Relaxing sequential consistency

- **x86 Total Store Order (TSO)**
  - "Stores" are ordered between cores
  - "Store-Load" pairs can be reordered between cores
- **Why?** Store latency
  - Before committing data on a core-local cache line (L1 or L2 cache), a core must wait for the cache lines to be invalidated on all other cores
  - The hardware does this via cache coherency protocols
  - Latency of store instructions even on core-local mem. references
- Solution
  - Use of processor-internal write buffers to hide store latency
  - Memory model violates sequential consistency
  - Reorderings of operations ⇒ potential for unexpected program behaviour

Assume SC – Can this program print "00"?



"Problems" due to lack of synchronization

> Race conditions: A timing dependent error involving shared state which occurs when the interleaving of execution of concurrent operations leads to erroneous program behaviour

> **Reasons for race conditions:**

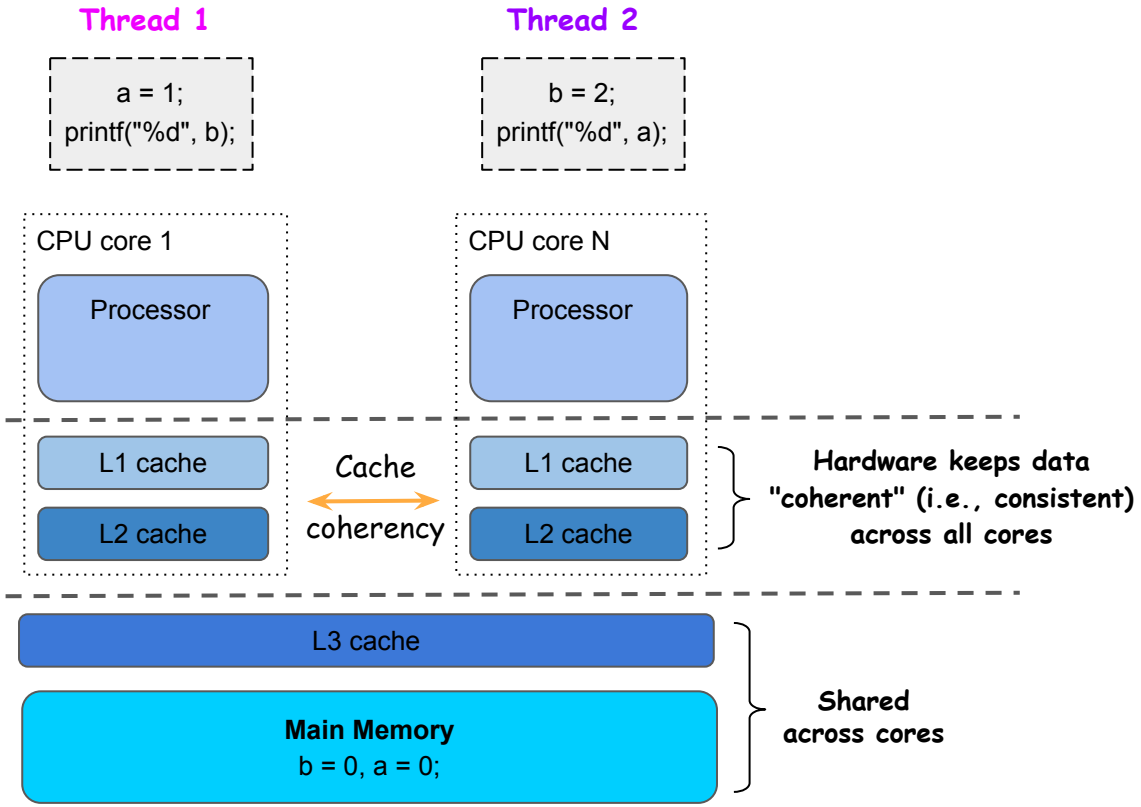
- Data races: Non-atomic, unsynchronized concurrent accesses, at least one of which mutating a shared variable
- Semantic ordering errors: Code that does not enforce the order programmers intended to for a group of memory accesses
- Weak memory consistency models: The set of allowed behaviours w.r.t. memory operations is not what the programer expected

Sequential consistency: Reminder

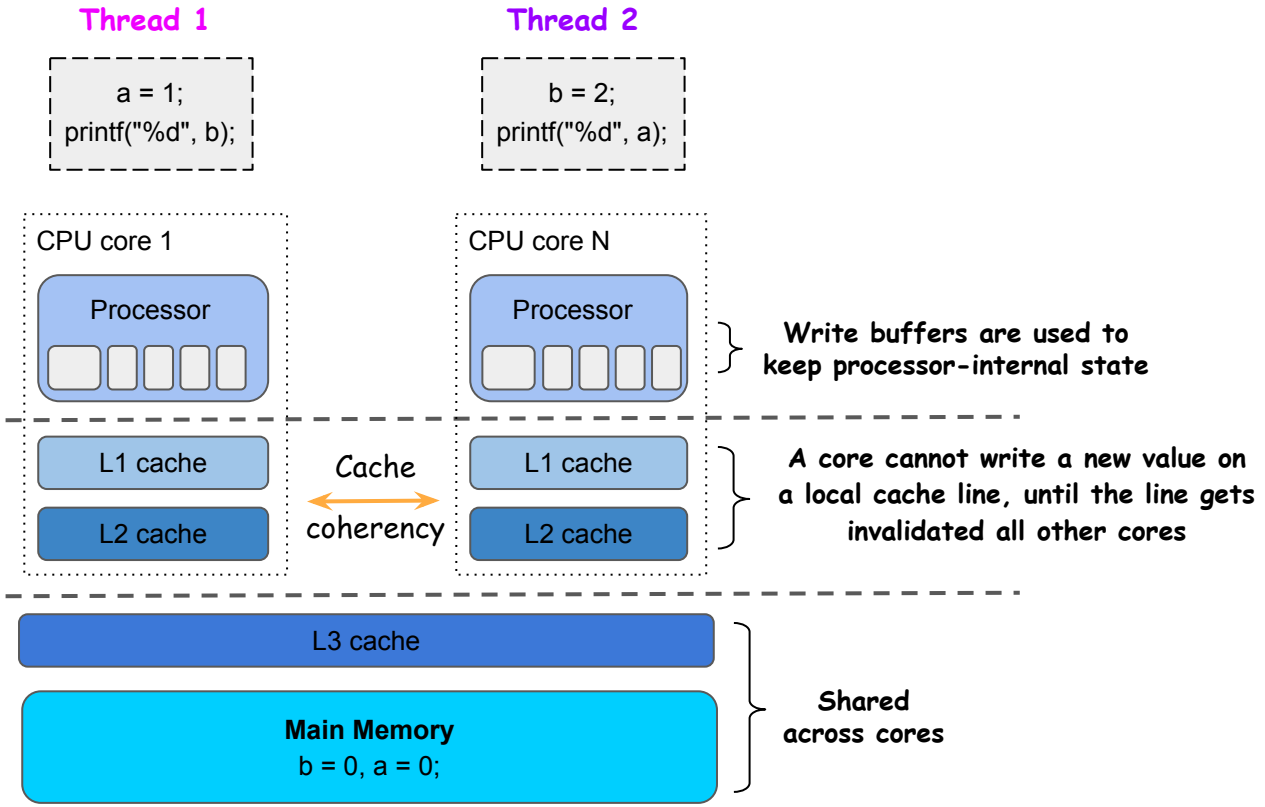
Pragmatic explanation

- Processors issue loads and stores in their local memory respecting their local program order
- Every load from a memory address gets its value from the last store before it, on the same memory address, in global memory

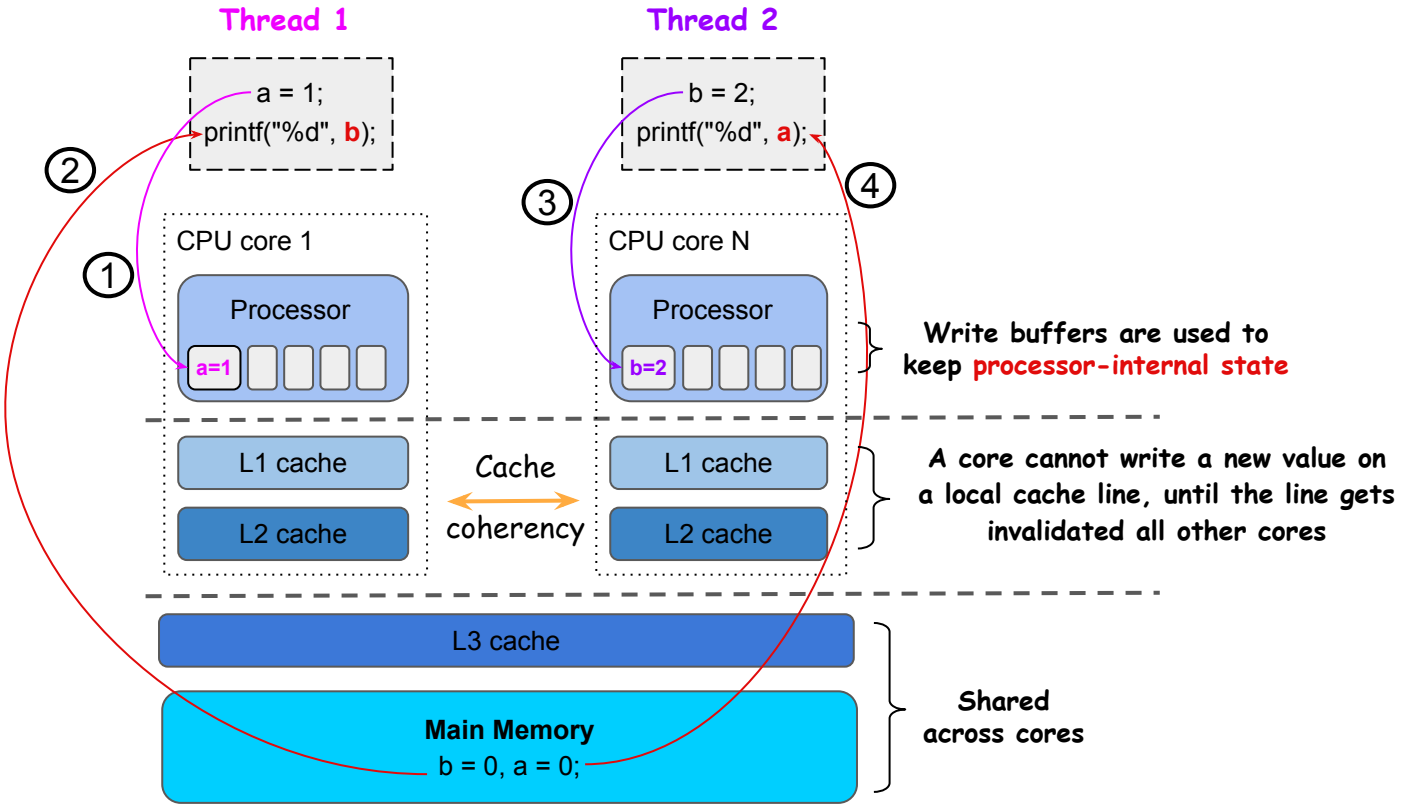
x86 Total Store Order (TSO)



x86 Total Store Order (TSO)



x86 Total Store Order (TSO)



How to prevent race conditions

> **Solutions**

- re:: Data races: Disable preemption (s/w)? Atomicity (h/w)?

>> Atomicity hits data races on their head **by definition**

>> "Relaxing" synchronization: "**a** happens before" ⇒ "**some** happens before"

- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]

Have h/w execute these "inseparable" instructions in one go?

- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]

Have h/w execute these "inseparable" instructions in one go?

How to prevent race conditions

- **Solutions**
  - re:: Data races: Disable preemption (s/w)? Atomicity (h/w)?
  - re:: Semantic ordering errors: h/w guardrails? Verification?
  - re:: Weak memory consistency models: Memory barriers

- Looks like we are getting somewhere, but…
  - What operations can "inseparable" instructions express?

Step back on the big picture

> **What are we trying to achieve?** Ensure the execution interleaving of concurrent operations does not lead to unforeseen, erroneous program behavior at runtime

- Looks like we are getting somewhere, but…
  - > What operations can "inseparable" instructions express?
    - %reg <- value at main mem.
    - %reg <- %reg + 1
    - value at main mem. <- %reg
  - > Too primitive ⇒ Well… use it as primitive…

Step back on the big picture

> **Where we are?** Solved most of the evils we've seen so far

>> **Memory barriers** => Prevent reorderings

>> **Atomicity** => Some "happens-before" => Enough for data races

- Atomic instructions only express too primitive operations
- Need atomicity of composite operations
- **Ideally:** atomicity of arbitrary many instructions

> Arbitrary many instructions ⇒ Critical Section

> Atomicity of arbitrary instructions ⇒ Mutual exclusion

How to prevent race conditions

› **Solutions**

- re:: Data races: Disable preemption (s/w)? Atomicity (h/w)?

>> Atomicity hits data races on their head **by definition**

>> "Relaxing" synchronization: "**a** happens before" ⇒ "**some** happens before"

- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]

Have h/w execute these "inseparable" instructions in one go?

- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]

Have h/w execute these "inseparable" instructions in one go?

How to prevent race conditions

> **Solutions**

✓ re:: Data races: Disable preemption (s/w)? Atomicity (h/w)?

✓ re:: Semantic ordering errors: h/w guardrails? Verification?

✓ re:: Weak memory consistency models: Memory barriers

> Looks like we are getting somewhere, but…

- What operations can "inseparable" instructions express?

Step back on the big picture

> **What are we trying to achieve?** Ensure the execution interleaving of concurrent operations does not lead to unforeseen, erroneous program behavior at runtime

- Looks like we are getting somewhere, but…
  - > What operations can "inseparable" instructions express?
    - %reg <- value at main mem.
    - %reg <- %reg + 1
    - value at main mem. <- %reg
  - > Too primitive ⇒ Well… use it as primitive…

Step back on the big picture

> **Where we are?** Solved most of the **evils** we've seen so far

>> **Memory barriers** ⇒ Prevent reorderings

>> **Atomicity** => Some "happens-before" => Enough for data races

- Atomic instructions only express too primitive operations
- Need atomicity of composite operations
- **Ideally:** atomicity of arbitrary many instructions

> Arbitrary many instructions ⇒ Critical Section

> Atomicity of arbitrary instructions ⇒ Mutual exclusion

The Critical Section (CS) problem

- ***Critical section*** (tangible set)
  - A sequence of instructions operating on shared state that ought to be executed by one thread at a time
- ***Mutual exclusion*** *(conceptual property)*
  - If I do, you wait; If you do, I wait...
  - **More generally:** At most one does at a time, all others wait
- ***The critical section problem***
  - How to allow only one thread at a time in a cs?
  - Rephrase: How to turn a sequence of instructions into an "atomic block"?
  - Rephrase: How to restrict the interleavings of thread executions re: entering the cs? (some happens-before relationship exists)

Designing a solution to the CS problem

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time
  - Progress: If multiple threads attempt to enter the CS, one must be allowed to proceed (a.k.a. liveness)
  - Starvation-free: If a thread is waiting to enter the CS, it must eventually enter (a.k.a. bounded-wait)
- Efficiency and Fairness properties (good to have)
  - Resource-efficiency: Don't waste processor cycles while waiting (busy-waiting) to enter the CS; voluntarily yield the processor
  - Fairness: All threads must wait approximately the same amount of time outside of the CS

Synchronized access to CS with locking

- We have reached our destination!
  - Divide code in two parts
    - **CS**: Executed with serialized access on some partial order
    - **Code outside** CS: Executed concurrently without any concerns
  - Synchronized access to critical sections
  - Coordinate among threads to restrict the possible interleavings
  - How? **Lock/Locking**

Implementing locks

> ***Lock:*** A token-like synchronization primitive used to coordinate concurrent thread accesses on CS

- While a thread cannot acquire the token, it waits until it acquires the token
- When a thread acquires the token, it holds it and no other thread can acquire it
- When a thread releases the token, any thread can acquire the token

Implementing locks

> ***Lock:*** A token-like synchronization primitive used to coordinate concurrent thread accesses on CS

- While a thread cannot acquire the **lock**, it waits until it acquires the **lock**
- When a thread acquires the **lock**, it holds it and no other thread can acquire it
- When a thread releases the **lock**, any thread can acquire the **lock**

Implementing locks

> ***Lock:*** A token-like synchronization primitive used to coordinate concurrent thread accesses on CS

- While a thread cannot acquire the **lock**, it waits outside of the CS until it acquires the **lock**
- When a thread acquires the **lock**, it execute instructions inside the cs, while all other threads wait outside the CS
- When a thread finishes executing instructions inside the CS, it releases the **lock** and any thread can acquire it

Draft API for locks

- Initialize an **shared** (noun: lock) variable
  >> Prototype: void **init_lock**(lock_type *lock_ptr);

- Acquire (verb: lock) the (noun: lock) variable
  >> Prototype: void **lock**(lock_type *lock_ptr);

- Release (verb: unlock) the (noun: lock) variable
  >> Prototype: void **unlock**(lock_type *lock_ptr);

Blast from the past

```c
int total = 0;

void *add(void *arg) {
  for (int i = 0; i < 1e6; ++i) {
    pthread_mutex_lock(&l);
    ++total;
    pthread_mutex_unlock(&l);
  }
  return NULL;
}

void main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Total-1: %d\n", total);

  total = 0;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t2, NULL);
  printf("Total-2: %d\n", total);
}
```

```
➜ obdjump -d ./counter

0000000000001159 <add>:
  1159: push %rbp                  # Save base pointer to stack
  115a: mov  %rsp, %rbp            # Set up new stack frame
  115d: mov  %rdi, -0x18(%rbp)     # *arg = %rdi
  1161: movl $0x0, -0x4(%rbp)      # i = 0
  1168: jmp  117d <add+0x24>       # for-loop start
  ...
  118e: lea  0x2eeb(%rip),%rax
  1195: mov  %rax,%rdi
  1198: call 1070 <pthread_mutex_lock@plt>   } lock the lock
  ...
  116a: mov  0x2ebc(%rip), %eax    #  %eax ← total
  1170: add  $0x1, %eax            #  %eax += 1
  1173: mov  %eax, 0x2eb3(%rip)    #  total ← %eax   } Critical section
  ...
  11ac: lea  0x2ecd(%rip),%rax
  11b3: mov  %rax,%rdi
  11b6: call 1040 <pthread_mutex_unlock@plt>   } Unlock the lock
  ...
  1179: addl $0x1, -0x4(%rbp)      # i += 1
  117d: cmpl $0xf423f, -0x4(%rbp)  # loop counter compare
  1184: jle  116a <add+0x11>       # for-loop jump
  1186: mov  $0x0, %eax            # rval = %eax
  118b: pop  rbp                   # Restore stack
  118c: ret                        # Return to caller
```

Blast from the past

> Yet, another execution interleaving

- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]
- Processor on thread B
  - **OS job:** Load %reg with its value before ctx switch
  - Thead B "thinks" %reg ⇔ shared value at main mem.
  - **But:** shared value has been mutated by someone else

Cannot happen

The Critical Section (CS) problem

- ***Critical section*** (tangible set)
  - A sequence of instructions operating on shared state that ought to be executed by one thread at a time
- ***Mutual exclusion*** *(conceptual property)*
  - If I do, you wait; If you do, I wait...
  - **More generally:** At most one does at a time, all others wait
- ***The critical section problem***
  - How to allow only one thread at a time in a cs?
  - Rephrase: How to turn a sequence of instructions into an "atomic block"?
  - Rephrase: How to restrict the interleavings of thread executions re: entering the cs? (some happens-before relationship exists)

Designing a solution to the CS problem

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time
  - Progress: If multiple threads attempt to enter the CS, one must be allowed to proceed (a.k.a. liveness)
  - Starvation-free: If a thread is waiting to enter the CS, it must eventually enter (a.k.a. bounded-wait)
- Efficiency and Fairness properties (good to have)
  - Resource-efficiency: Don't waste processor cycles while waiting (busy-waiting) to enter the CS; voluntarily yield the processor
  - Fairness: All threads must wait approximately the same amount of time outside of the CS

Synchronized access to CS with locking

- We have reached our destination!
  - Divide code in two parts
    - **CS**: Executed with serialized access on some partial order
    - **Code outside** CS: Executed concurrently without any concerns
  - Synchronized access to critical sections
  - Coordinate among threads to restrict the possible interleavings
  - **How? Locks/Locking**

Implementing locks

> ***Lock:*** A token-like synchronization primitive used to coordinate concurrent thread accesses on CS

- While a thread cannot acquire the **lock**, it waits outside of the CS until it acquires the **lock**
- When a thread acquires the **lock**, it execute instructions inside the cs, while all other threads wait outside the CS
- When a thread finishes executing instructions inside the CS, it releases the **lock** and any thread can acquire it

Draft API for locks

- Initialize an **shared** (noun: lock) variable
  >> Prototype: void **init**(lock_type *lock_ptr);

- Acquire (verb: lock) the (noun: lock) variable
  >> Prototype: void **lock**(lock_type *lock_ptr);

- Release (verb: unlock) the (noun: lock) variable
  >> Prototype: void **unlock**(lock_type *lock_ptr);

Blast from the past

```c
int total = 0;

void *add(void *arg) {
  for (int i = 0; i < 1e6; ++i) {
    pthread_mutex_lock(&l);
    ++total;
    pthread_mutex_unlock(&l);
  }
  return NULL;
}

void main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Total-1: %d\n", total);

  total = 0;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t2, NULL);
  printf("Total-2: %d\n", total);
}
```

```
➜ obdjump -d ./counter

0000000000001159 <add>:
  1159: push %rbp                  # Save base pointer to stack
  115a: mov  %rsp, %rbp            # Set up new stack frame
  115d: mov  %rdi, -0x18(%rbp)     # *arg = %rdi
  1161: movl $0x0, -0x4(%rbp)      # i = 0
  1168: jmp  117d <add+0x24>       # for-loop start
  ...
  118e: lea  0x2eeb(%rip),%rax
  1195: mov  %rax,%rdi
  1198: call 1070 <pthread_mutex_lock@plt>   } lock the lock
  ...
  116a: mov  0x2ebc(%rip), %eax    #  %eax ← total
  1170: add  $0x1, %eax            #  %eax += 1
  1173: mov  %eax, 0x2eb3(%rip)    #  total ← %eax   } Critical section
  ...
  11ac: lea  0x2ecd(%rip),%rax
  11b3: mov  %rax,%rdi
  11b6: call 1040 <pthread_mutex_unlock@plt>   } Unlock the lock
  ...
  1179: addl $0x1, -0x4(%rbp)      # i += 1
  117d: cmpl $0xf423f, -0x4(%rbp)  # loop counter compare
  1184: jle  116a <add+0x11>       # for-loop jump
  1186: mov  $0x0, %eax            # rval = %eax
  118b: pop  rbp                   # Restore stack
  118c: ret                        # Return to caller
```

Blast from the past

> Yet, another execution interleaving

- Processor on thread B
  - %reg <- value at main mem. [global vs. local state: consistent]
- Processor on thread A
  - %reg <- value at main mem. [global vs. local state: consistent]
  - %reg <- %reg + 1 [global vs. local state: divergent]
  - value at main mem. <- %reg [global vs. local state: consistent]
- Processor on thread B
  - **OS job:** Load %reg with its value before ctx switch
  - Thead B "thinks" %reg ⇔ shared value at main mem.
  - **But:** shared value has been mutated by someone else

Cannot happen

Uniprocessor lock (CONFIG_SMP=off, CONFIG_PREEMPT=off)

```
>> typedef uint8_t raw_spinlock_t;
>> void init(raw_spinlock_t *lock) {
        *lock = 0; // 0 = unlocked, 1 = locked
   }
>> void lock(raw_spinlock_t *lock) {
        return;
    }
>> void unlock(raw_spinlock_t *lock) {
        return;
    }
```

Uniprocessor lock (CONFIG_SMP=off, CONFIG_PREEMPT=on)

```
>> typedef uint8_t raw_spinlock_t;
>> void init(raw_spinlock_t *lock) {
        *lock = 0; // 0 = unlocked, 1 = locked
   }
>> void lock(raw_spinlock_t *lock) {
        disable_preemption();
    }
>> void unlock(raw_spinlock_t *lock) {
        enable_preemption();
    }
```

Another blast from the past

- What operations can "inseparable" instructions express?

  - %reg <- [mem.]
  - %reg <- %reg + 1
  - [mem.] <- %reg

  } Atomic operation

Another blast from the past

- Consider another atomic operation

  xchgb reg, [mem] (composite operation / "inseparable" instructions)

  - temp <- %reg1
  - %reg1 <- [mem.]
  - [mem.] <- temp

  } Atomic swap (exchange): [mem] <-> %reg

- Why the hassle?

```
uint8_t test_and_set(uint8_t *flag) {
    %reg <- 1              // "set" val of %reg to 1
    xchgb %reg, *flag      // val of reg <->  val of flag
    return %reg            // %reg has the old val of flag
 }
```

  >> **Return 1?** flag was 1 (**lock was locked**)

  >> **Return 0?** flag was 0, and is now 1 (**lock was unlocked and just got locked**)

Draft SMP spinlock implementation

```
>> void init(lock_type *lock) {
        *lock = 0; // 0 = unlocked, 1 = locked
   }
>> void lock(lock_type *lock) {
        preempt_disable();          // Don't lose processor, if getting the lock
        while (test_and_set(lock)) ; // try while lock not free
    }
>> void unlock(lock_type *lock) {
        __asm__ volatile("sfence" ::: "memory"); // mem. barrier
        // all write mem. ops from the cs must completed before this point
        *lock = 0;
        preempt_enable();
    }
```

Designing a "fair" spinlock

- Correctness properties (non-negotiable)
  - Mutual exclusion: At most one thread inside the CS at any time
  - Progress: If multiple threads attempt to enter the CS, one must be allowed to proceed (a.k.a. liveness)
  - Starvation-free: If a thread is waiting to enter the CS, it must eventually enter (a.k.a. bounded-wait)
- Efficiency and Fairness properties (good to have)
  - Resource-efficiency: Don't waste processor cycles while waiting (busy-waiting) to enter the CS; voluntarily yield the processor
  - Fairness: All threads must wait approximately the same amount of time outside of the CS ⇒ Lamport's Bakery algorithm ⇒ Ticket spinlock

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- Do use when: Cost of 2*ctx switch > cost of instruction of cs
- **Do NOT use when:** CS contains operations that may sleep
- What about "copy_from/to_user"?

>> Sleeping locks: Self-preempt and sleep if can't acquire the lock

- Do use when: Cost of 2*ctx switch << cost of instruction of cs
- Do use when: Relatively larger CS
- **Do NOT use when:** In interrupt handlers — Why?
- Known by the name "mutex"

Implementing a sleeping lock

```
typedef struct mutex_t {
  // the "lock" variable
  // initially "unlocked"
  int lock = 0;
  // keeps track of waiters
  queue_t queue = NULL;
} mt;
```

```
void mutex_lock(mutex_t *mt) {
  while (test_and_set(&mt->lock)) {

    // lock is being held
    // register self on waiters
    enqueue(&mt->queue, self);
    // yield the processor
    add_task_to_waitq(self);
  }
}
```

```
void mutex_unlock(mutex_t *mt) {
  // release the lock
  mt->lock = 0
  // let a waiters know
  if (!queue_empty(mt->queue)) {
    wake_up(dequeue(mt->queue))
  }
}
```

Implementing a sleeping lock: Lost wakeup

```
typedef struct mutex_t {
  // the "lock" variable
  // initially "unlocked"
  int lock = 0;
  // keeps track of waiters
  queue_t queue = NULL;
} mt;
```

```
void mutex_lock(mutex_t *mt) {
  while (test_and_set(&mt->lock)) {

    // lock is being held
    // register self on waiters
    enqueue(&mt->queue, self);
    // yield the processor
    add_task_to_waitq(self);
  }
}
```

```
void mutex_unlock(mutex_t *mt) {
  // release the lock
  mt->lock = 0
  // let a waiters know
  if (!queue_empty(mt->queue)) {
    wake_up(dequeue(mt->queue))
  }
}
```

Implementing a sleeping lock

```
typedef struct mutex_t {
  // the "lock" variable
  // initially "unlocked"
  int lock = 0;
  // lock is a shared variable
  raw_spinlock_t guard = 0;
  // keeps track of waiters
  queue_t queue = NULL;
} mt;
```

```
void mutex_lock(mutex_t *mt) {
  spinlock_lock(&mt->guard)
  // lock is being held
  while (mt->lock != 0) {
    // register self on waiters
    enqueue(&mt->queue, self);
    spinlock_unlock(&mt->guard);
    // yield the processor
    add_task_to_waitq(self);
    spinlock_lock(&mt->guard);
  }
  // acquire the lock
  mt->lock = 1;
  spinlock_unlock(&mt->guard);
}
```

```
void mutex_unlock(mutex_t *mt) {
  spinlock_lock(&mt->guard);
  // release the lock
  mt->lock = 0
  // let a waiters know
  if (!queue_empty(mt->queue)) {
    wake_up(dequeue(mt->queue))
  }
  spinlock_unlock(&mt->guard)
}
```

Types of locks

> Depending on what the thread trying to acquire an unavailable lock does, we have two categories of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

- Do use when: Cost of 2*ctx switch > cost of instruction of cs
- **Do NOT use when:** CS contains operations that may sleep
- What about "copy_from/to_user"?

>> Sleeping locks: Self-preempt and sleep if can't acquire the lock

- Do use when: Cost of 2*ctx switch << cost of instruction of cs
- Do use when: Relatively larger CS
- **Do NOT use when:** In interrupt handlers — Why?
- Known by the name "mutex"

Types of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

>> Sleeping locks: Self-preempt and sleep if can't acquire the lock

>> Fine-grained locks: Intention for getting the lock? Who are you competing with?

Understanding contention

>> Random hash-table insertions, 1024 buckets (machine: 12 vCPUs)

- Synchronization w/ global lock (concurrent, not parallel)
- Synchronization w/ fine-grained, per-bucket lock (concurrent + parallel)
- Lock-free implementation w/ atomics (concurrent + parallel; no lock)

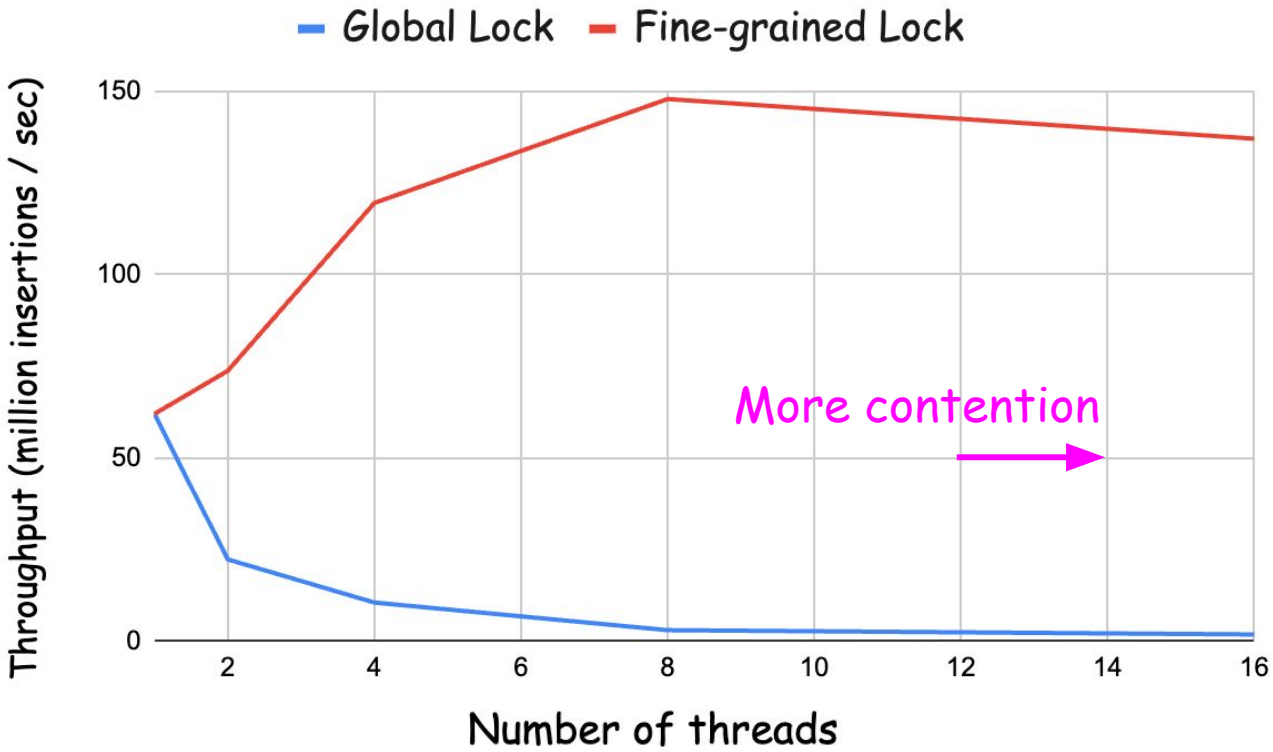
```
void global_insert(int key) {
    idx = hash(key);
    Node *n = malloc(sizeof(Node));
    n->key = key;
    n->next = table[idx].head;
    spinlock_lock(&global_lock);
    table[idx].head = n;
    spinlock_unlock(&global_lock);
}
```

```
void bucket_insert(int key) {
    idx = hash(key);
    Bucket *b = &table[idx];
    Node *n = malloc(sizeof(Node));
    n->key = key;
    n->next = b->head
    spinlock_lock(&b->lock);
    b->head = n;
    spinlock_unlock(&b->lock);
}
```

```
void lockfree_insert(int key) {
    idx = hash(key);
    Bucket *b = &table[idx];
    Node *old_head;
    Node *n = malloc(sizeof(Node));
    n->key = key;
    do {
      old_head = b->head;
      n->next = old_head;
    } while (!cas(&b->head, &old_head, n));
}
```

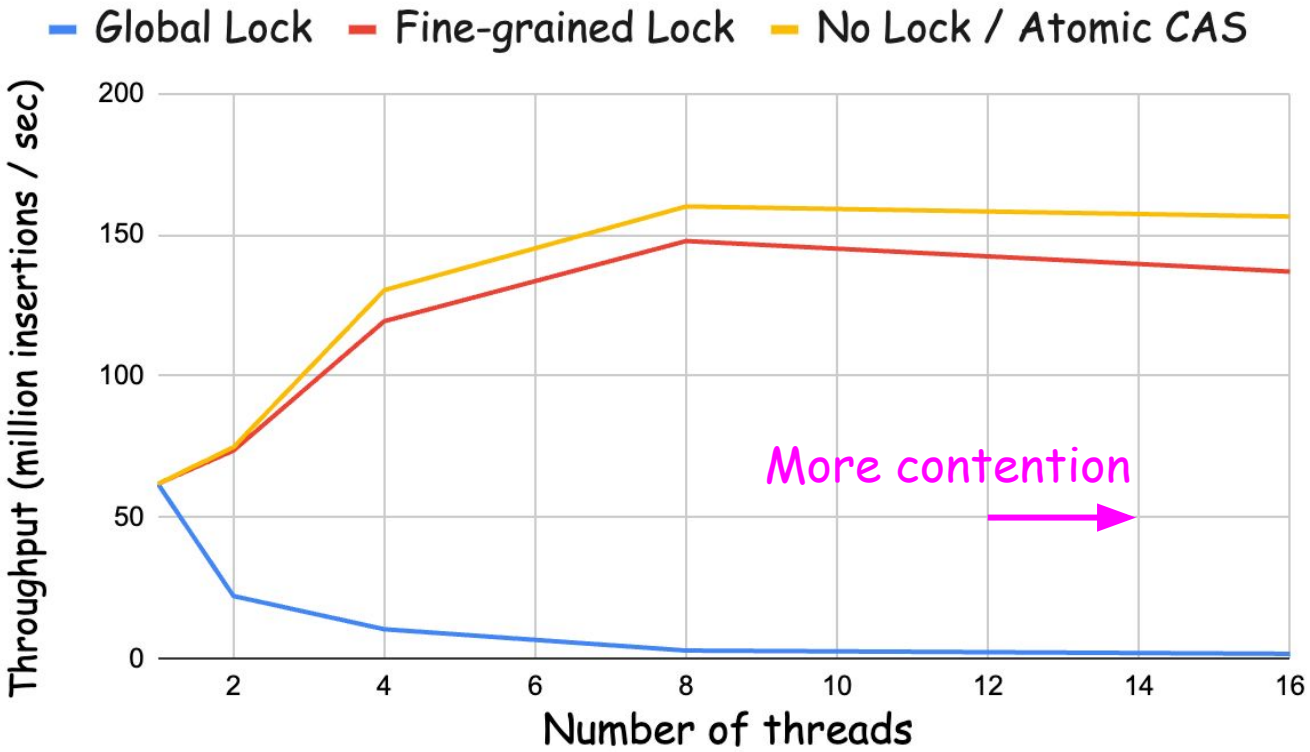
Understanding contention

>> Random hash-table insertions, 1024 buckets (machine: 12 vCPUs)



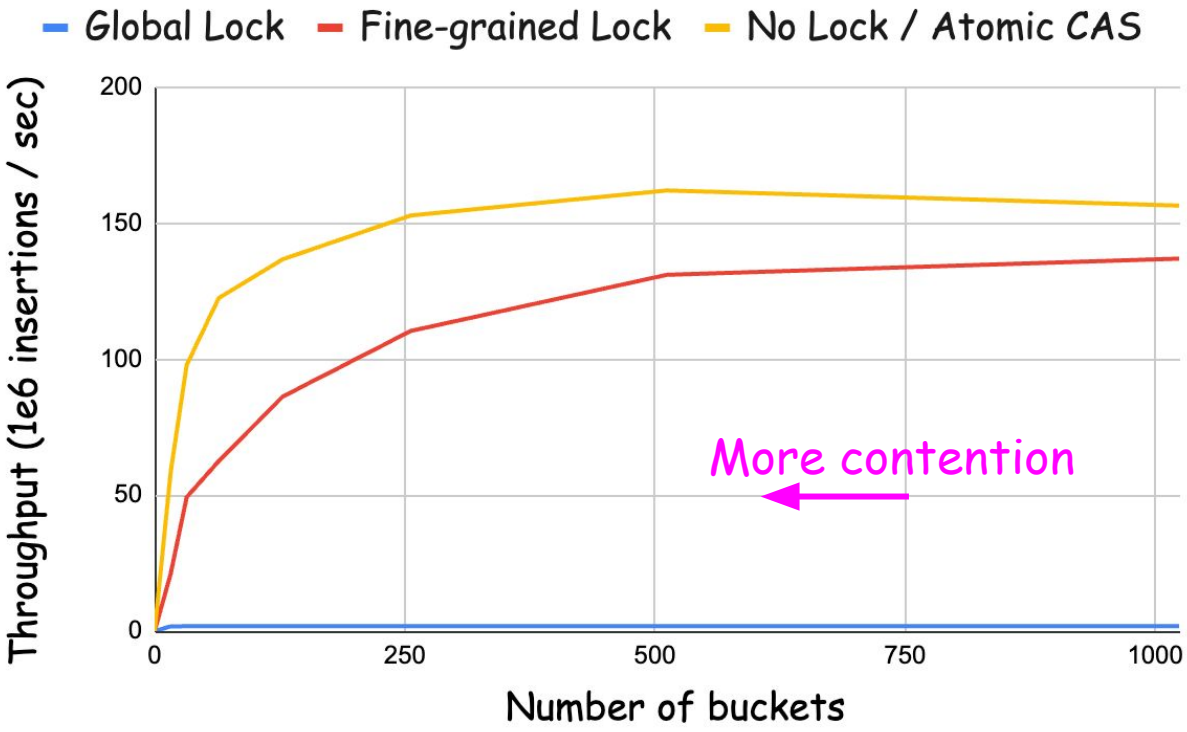
Understanding contention

>> Random hash-table insertions, 1024 buckets (machine: 12 vCPUs)



Understanding contention

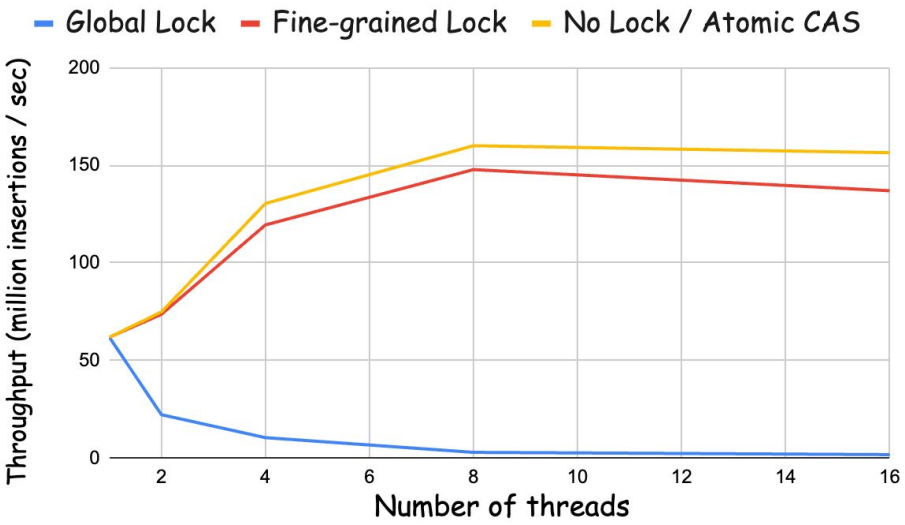
>> Random hash-table insertions, 16 (kernel) threads

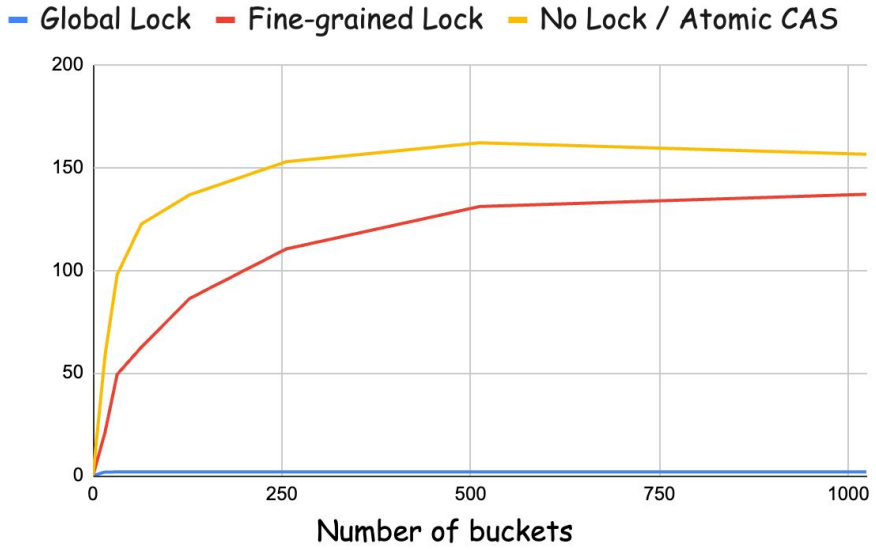


Understanding contention

>> Random hash-table insertions

- Which locking implementation would you choose?
- How many threads and how many buckets will you set?





Types of locks

>> Spinlocks: Spin continuously while trying to acquire the lock

>> Sleeping locks: Self-preempt and sleep if can't acquire the lock

>> Fine-grained locks: Intention for getting the lock? Readers don't block readers...

>> Hybrid kernel-/user-space locks: Can you avoid kernel intervention? when?

>> Lockless synchronization: Linux RCU lock; copy and modify local copy, then update

Read more here: https://www.kernel.org/doc/html/v4.18/kernel-hacking/locking.htm

Deadlocks: Using locks incorrectly

» What is a deadlock? **Unrecoverable error** where all members of a group of entities wait indefinitely on each other and cannot make progress, because each waits for another member of the group, including itself, to take action

Deadlocks: Using locks incorrectly

» **What causes a deadlock** (preconditions: "and")?

1) Mutual Exclusion: Exclusive access on a shared resource
2) No preemption: Once the shared resource is obtained by a thread, it cannot be taken away involuntarily
3) Hold and Wait: A thread holding a shared resource is also waiting for additional resources, held by other threads
4) Circular Wait: There exists a set of waiting threads, T = {T_1, T_2, …, T_n}, such that T_1 is waiting for a resource held by T_2, T_2 is waiting for a resource held by T_3, …, and T_n is waiting for a resource held by T_1

Deadlock prevention: Proactive

>> **How to prevent deadlocks** (proactively)? Remove one of the four **AND** preconditions

1) No Mutual Exclusion: No exclusive access on a shared resource
2) Enable preemption: No thread can hold the shared resource for more than a given time frame while others are trying to obtain it
3) Avoid hold and wait: No thread can hold the shared resource while requesting another, and must, instead, try to obtain all the necessary resources at once, at the beginning
4) Avoid circular waiting: Impose a hierarchical ordering on shared resource acquisition

Deadlock prevention: Proactive

» **How to prevent deadlocks** (proactively)? Remove one of the four AND preconditions

1) No Mutual Exclusion: No exclusive access on a shared resource...
2) Enable preemption: **Add timeout on the acquire operation, and raise an error if it expires while attempting to get the lock**
3) ~~Avoid hold and wait: No thread can hold the shared resource while~~ requesting another, and must, instead, try to obtain all the necessary resources at once, at the beginning
4) Avoid circular waiting: Impose a hierarchical ordering on shared resource acquisition

Deadlock prevention: Proactive

>> **How to prevent deadlocks** (proactively)? Remove one of the four AND preconditions

1) No Mutual Exclusion: No exclusive access to a shared resource…
2) Enable preemption: Add timeout on the acquire operation, and raise an error if it expires while attempting to get the lock
3) Avoid hold and wait: No thread can hold the shared resource while requesting another, and must, instead, try to obtain all the necessary resources at once, at the beginning
4) Avoid circular waiting: **Developers follow this in practice by acquiring locks in a well-documented hierarchical order**

The dining philosophers problem

- **Problem description:** N philosophers sitting around a circular table
  - Between each pair of philosophers there is a chopstick fork (N total)
  - Each philosopher alternates between two activities: Thinking and Eating
  - Rules to eat
    - Each philosopher needs two chopsticks to eat
    - Each philosopher can take either their left or their right chopstick
    - They cannot pick both chopsticks at once

> Typical CS problem — Everything we don't want, could happen

- Deadlock (i.e., no one can proceed): Could happen… How?
- Livelock (i.e., no real progress / no liveness): Could happen… How?
- Starvation (i.e., some no real progress): Could happen… How?

The dining philosophers problem

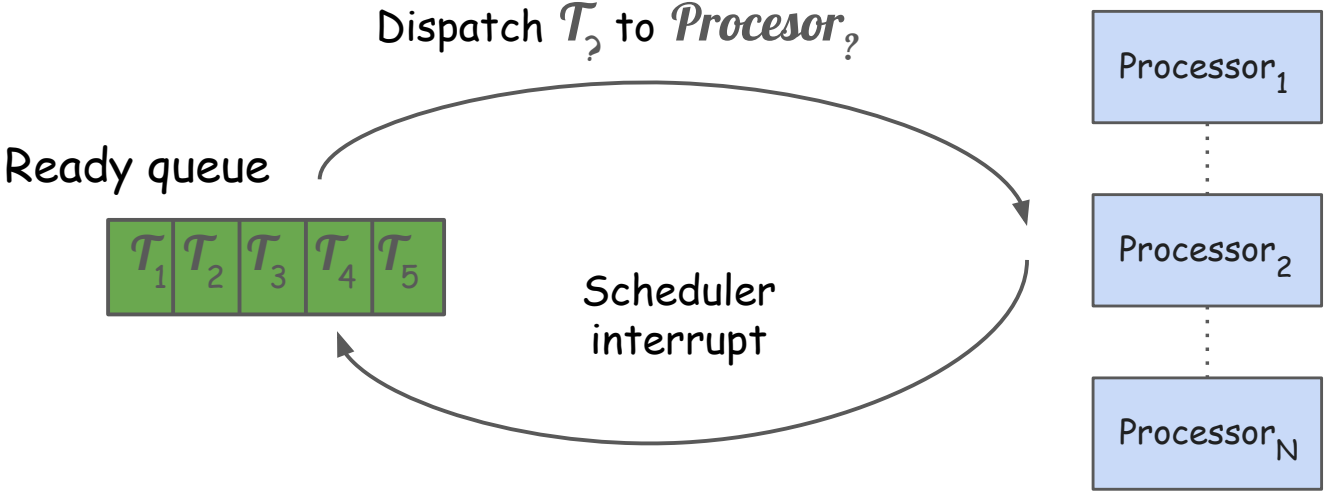
- **Problem description:** N philosophers sitting around a circular table
  - Between each pair of philosophers there is a chopstick fork (N total)
  - Each philosopher alternates between two activities: Thinking and Eating
  - Rules to eat
    - Each philosopher needs two chopsticks to eat
    - Each philosopher can take either their left or their right chopstick
    - They cannot pick both chopsticks at once

> Typical CS problem — **Designing a correct solution**

- Avoid deadlocks: **How?** Break circular wait [Pn will not get Cn, if C_1 is held]
- Avoid livelocks: **How?** Use random / exponential backoff
- Avoid starvation: **How?** Use a queue, or an aging / ticket mechanism

The scheduling problem

> Given k tasks ready to run in a system with N available processors, which task should be dispatched to which processor at any given point in time?



The scheduling problem

> Given k tasks ready to run in a system with N available processors, which task should be dispatched to which processor at any given point in time?

> **Quantitative goals**

- Minimize avg. completion time of all jobs
- Minimize the avg. response time of all jobs (latency)
- Maximize #jobs completed per unit of time (throughput)

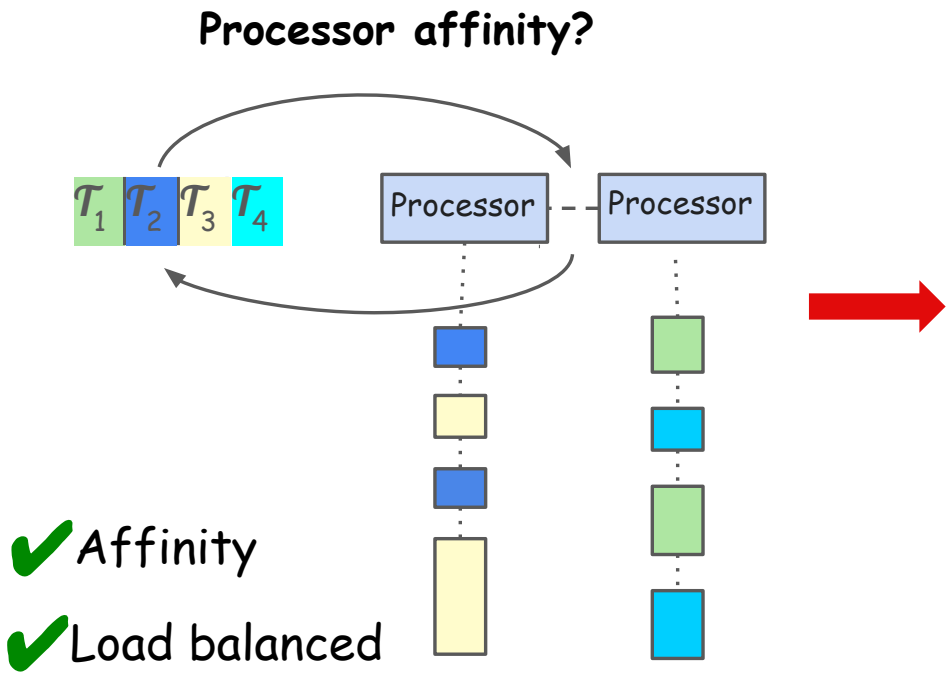
The scheduling problem

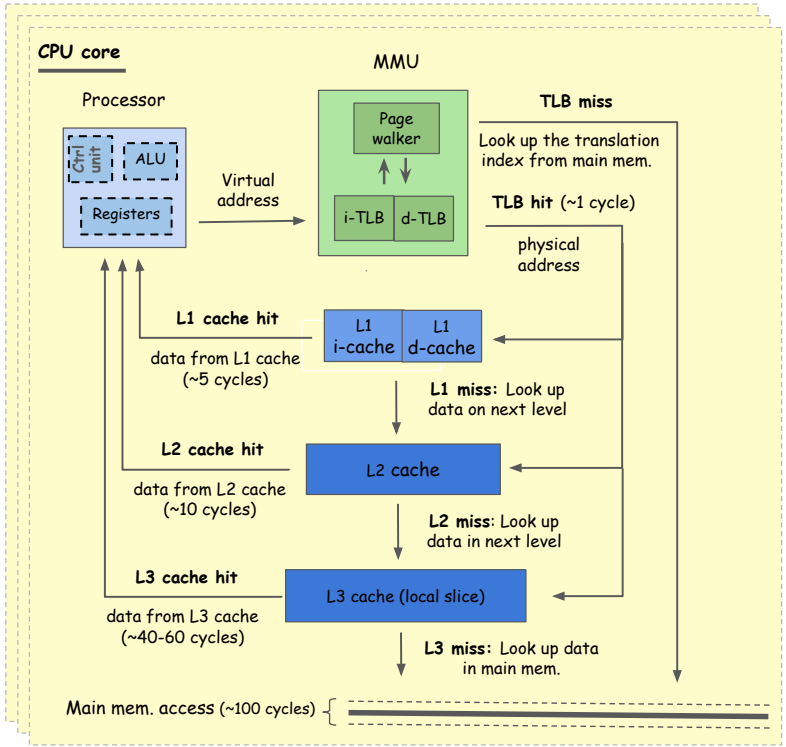
› Given k tasks ready to run in a system with N available processors, which task should be dispatched to which processor at any given point in time?

› **Qualitative goals**

- Jobs receive a similar share of available processors' time
- Upper bound on the maximum latency of jobs
- Uniform load across all available processors

SMP load balancing and processor affinity





Workloads and scheduling requirements

Real time workloads: Hard real time and soft real time

- Hard real time
  - Their tasks must finish within specific deadlines
  - Example: Pacemakers, Airbag deployment systems, Autopilots
  - Sched. goals: **Zero miss rate; Guarantees every time**
  - Sched. algorithms: Earliest Deadline First (EDF)
- Soft real time
  - Their tasks must receive priority over lower-priority tasks
  - Example: Video Streaming / Multimedia applications
  - Sched. goals: **Bounded latency**
  - Sched. algorithms: Priority-based scheduling

Workloads and scheduling requirements

CPU- vs I/O-bound workloads

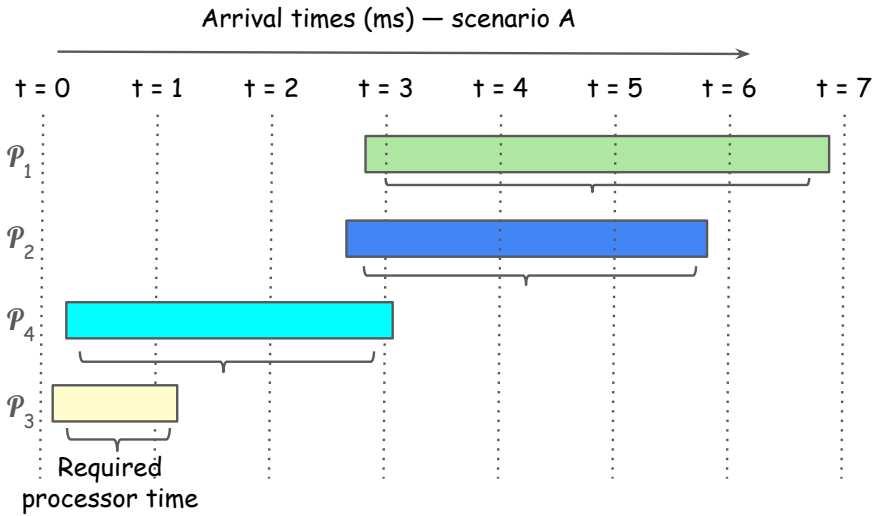
- CPU-bound
  - Their tasks spend most time doing intensive computation
  - Rarely yield voluntarily and rarely need to perform I/O
  - Example: Scientific simulations / computations
  - Sched. goals: **Balanced processor time / avoid starvation**
  - Sched. algorithms: RR with large quanta

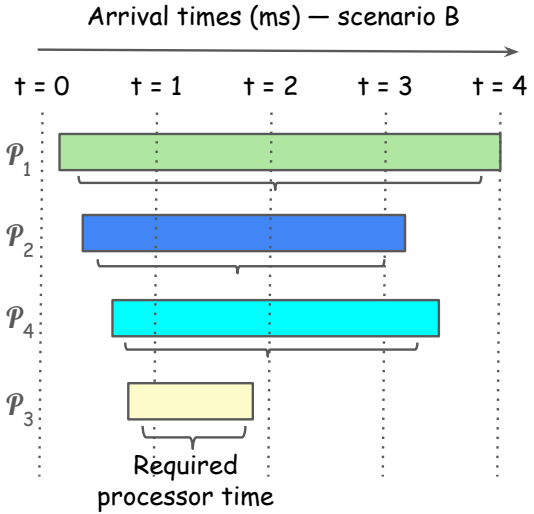
Workloads and scheduling requirements

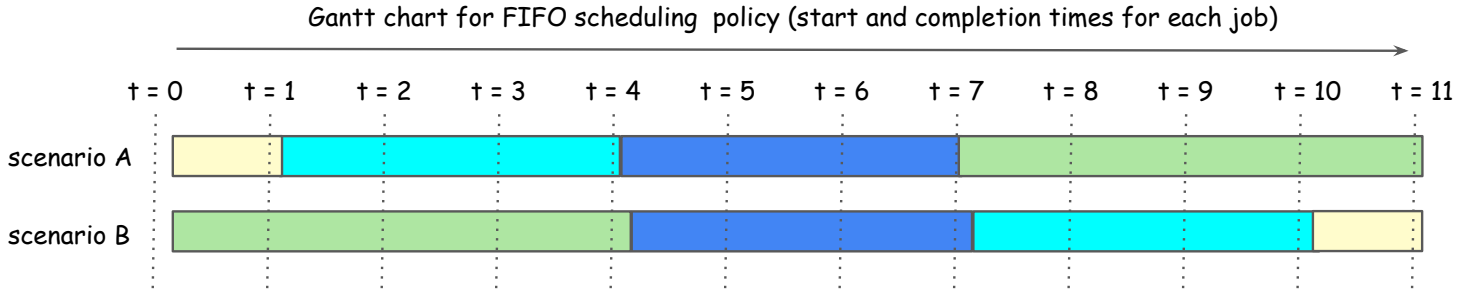
CPU- vs I/O-bound workloads

- I/O-bound workloads
  - Their tasks spend most of their time waiting for I/O
  - Short processor bursts and then block again
  - Example: Downloading a file / fetching data from disk
  - Sched. goals: Minimize I/O device idle periods by promptly allocating the processor for the brief time needed to initiate I/O requests (usually via DMA)
  - Sched. algorithms: Priority-based favoring I/O-bound tasks

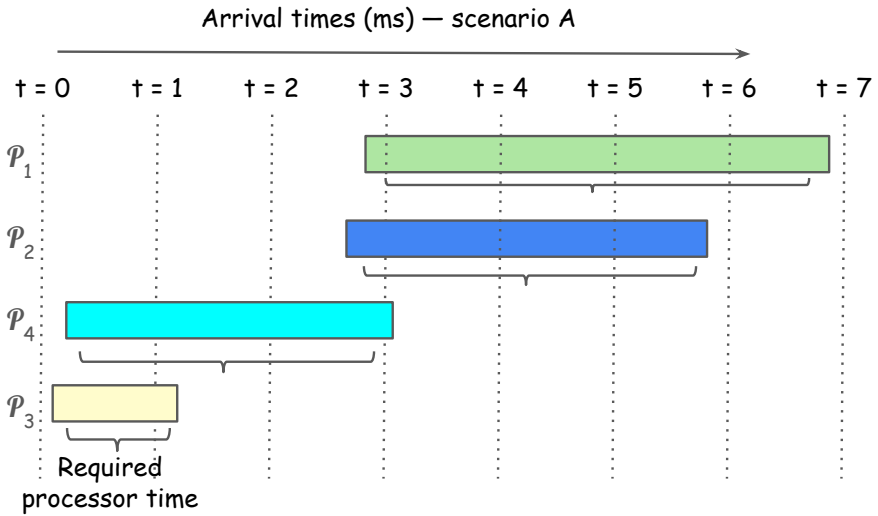
First in First Out scheduling policy (SCHED_FIFO)

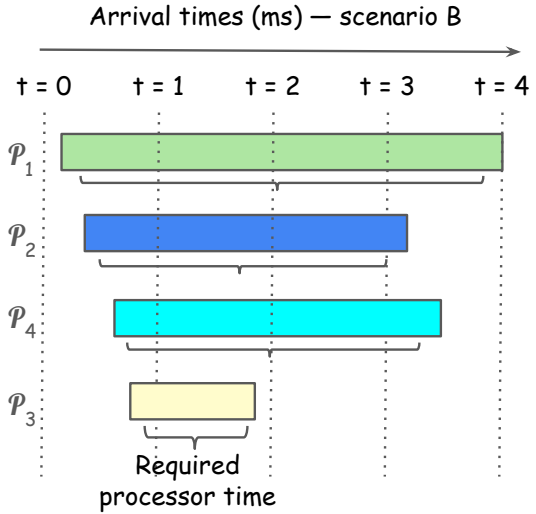


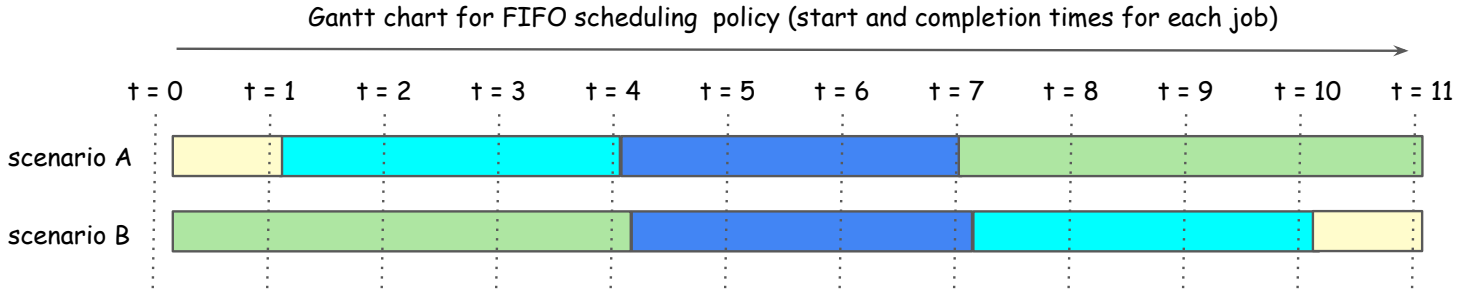




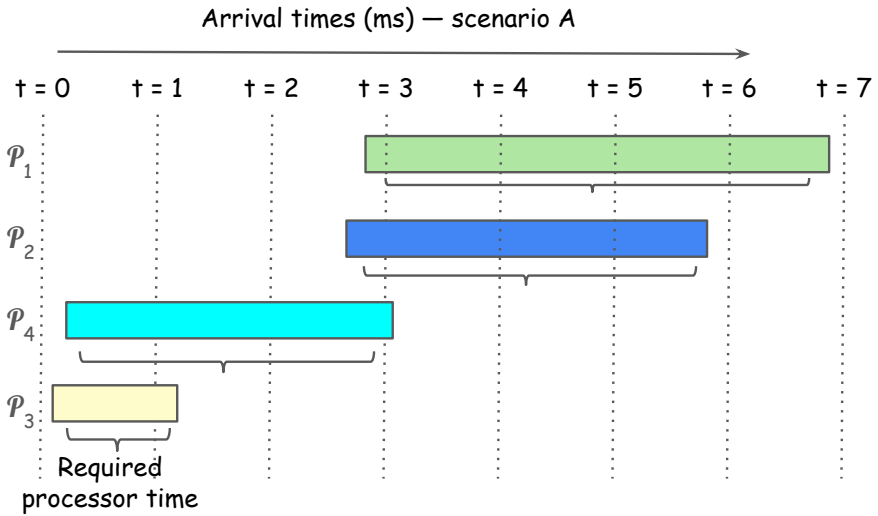
SCHED_FIFO: Avg.completion and response time?

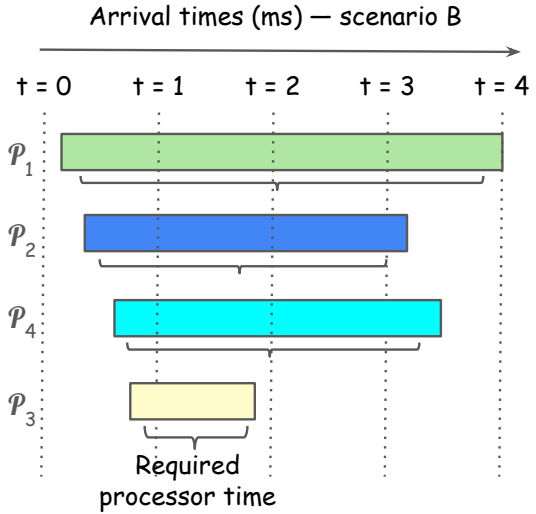


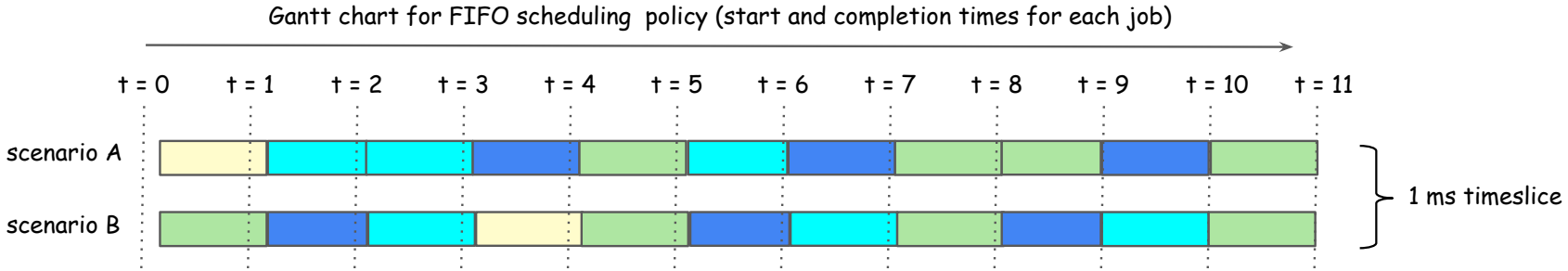




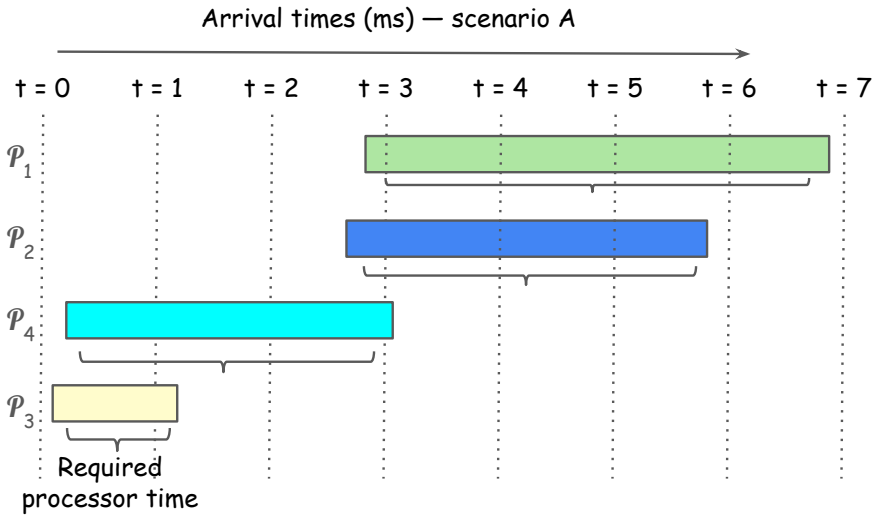
Round Robin scheduling policy (SCHED_RR)

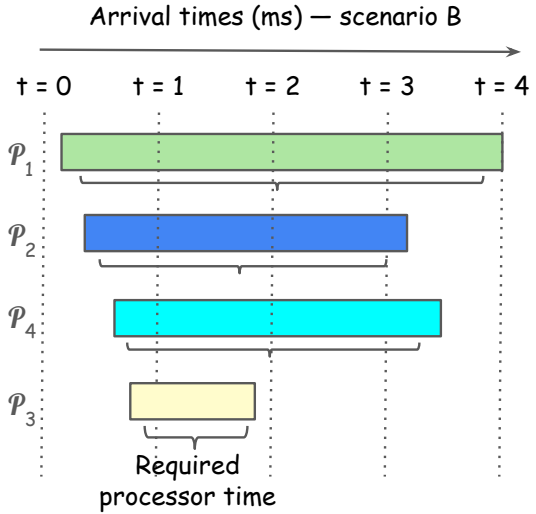


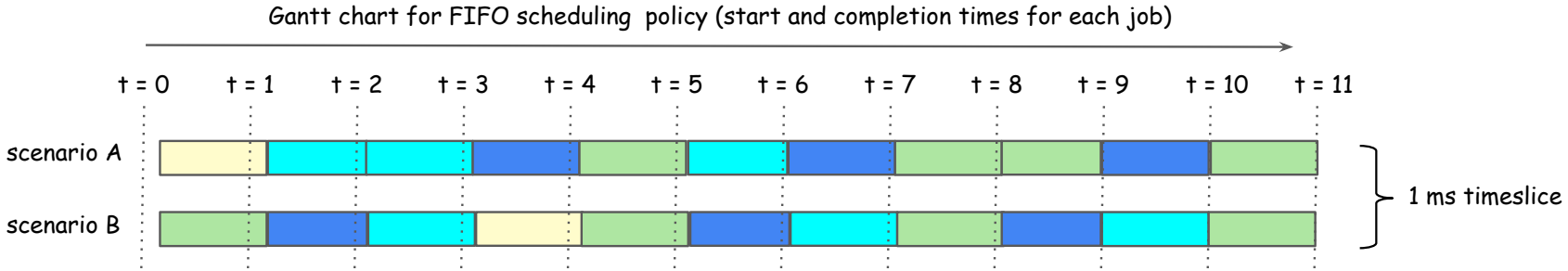




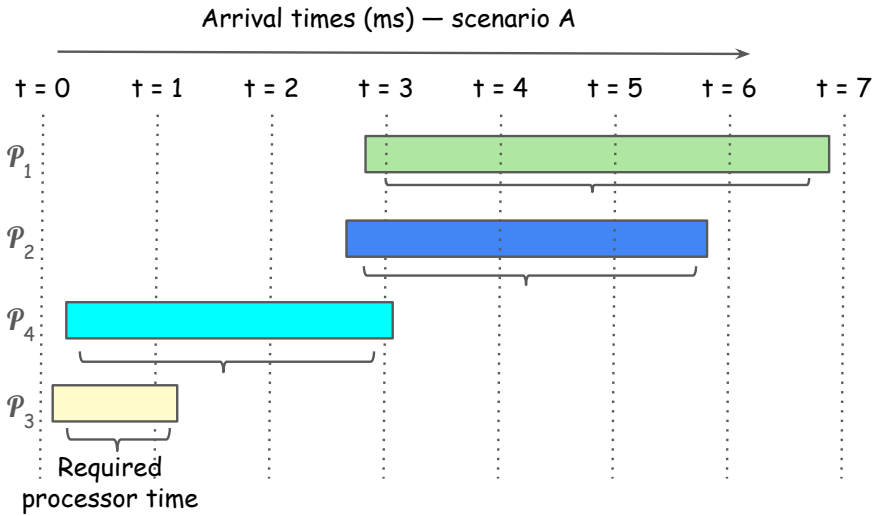
SCHED_RR: Avg.completion and response time?

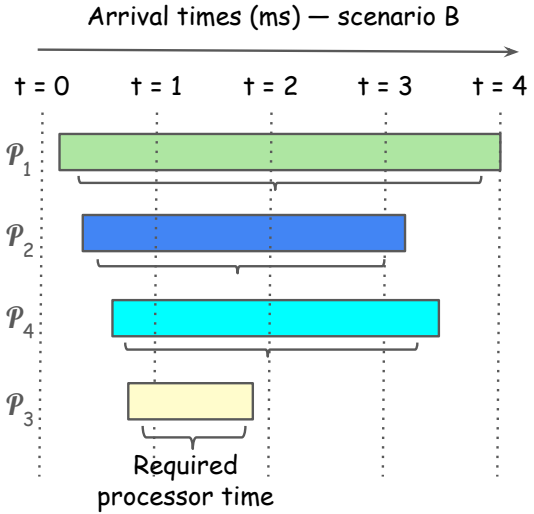


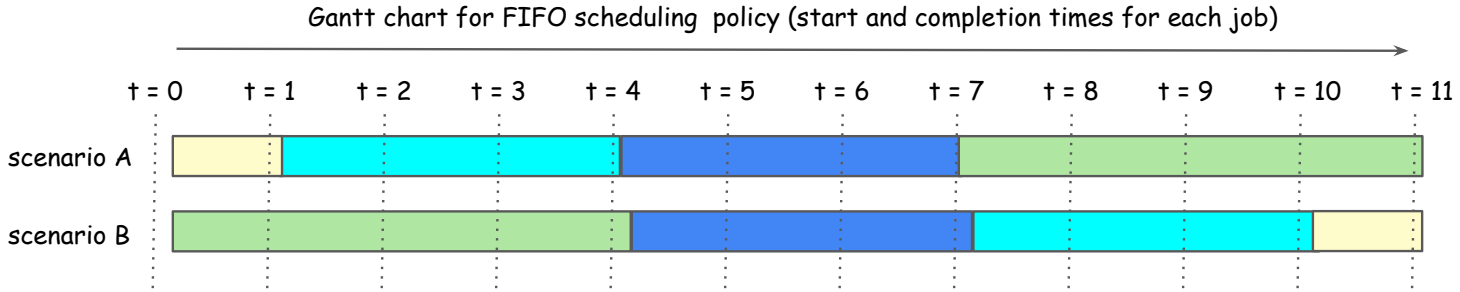




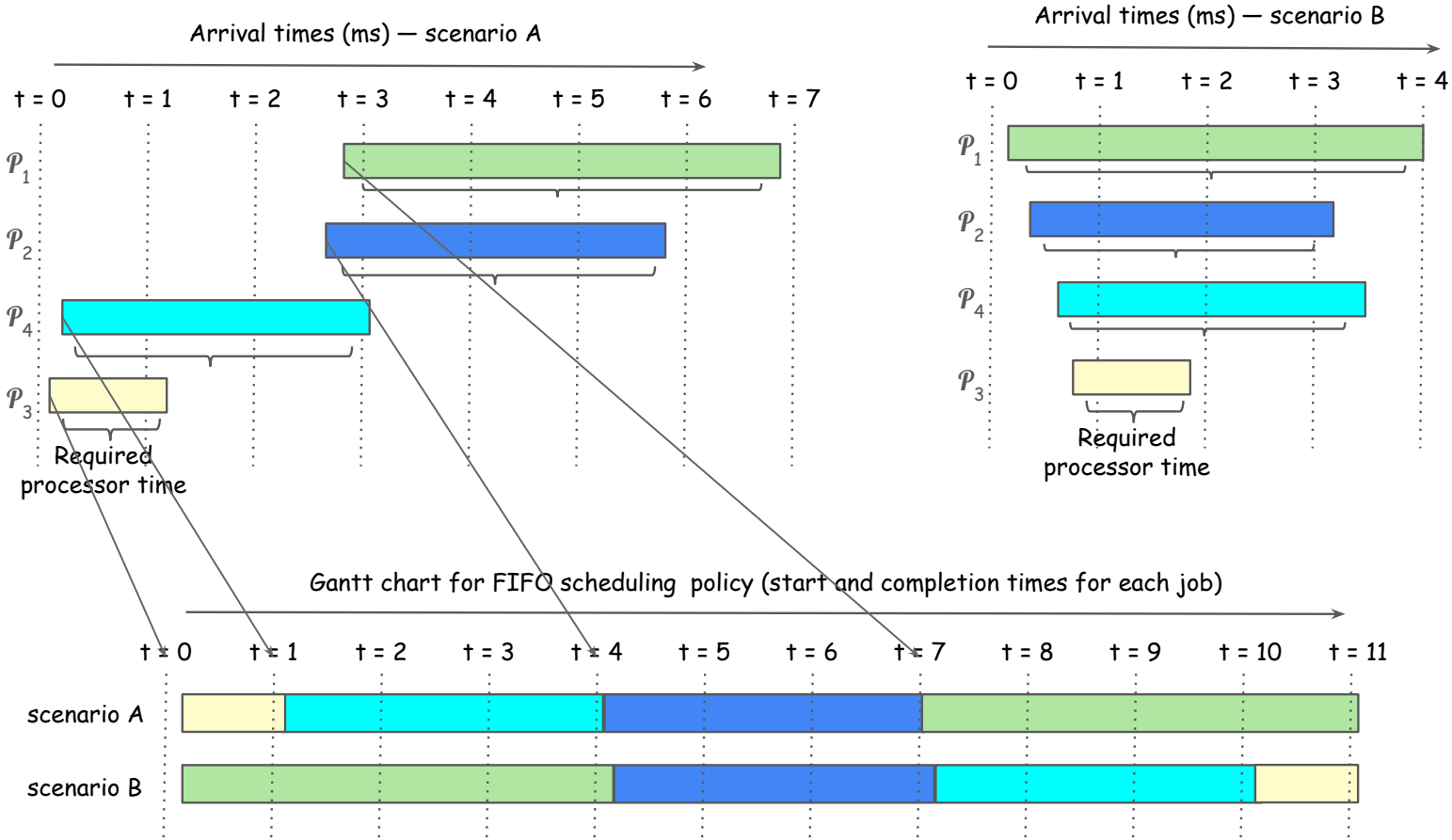
First in First Out scheduling policy (SCHED_FIFO)







Calculating response time (latency)



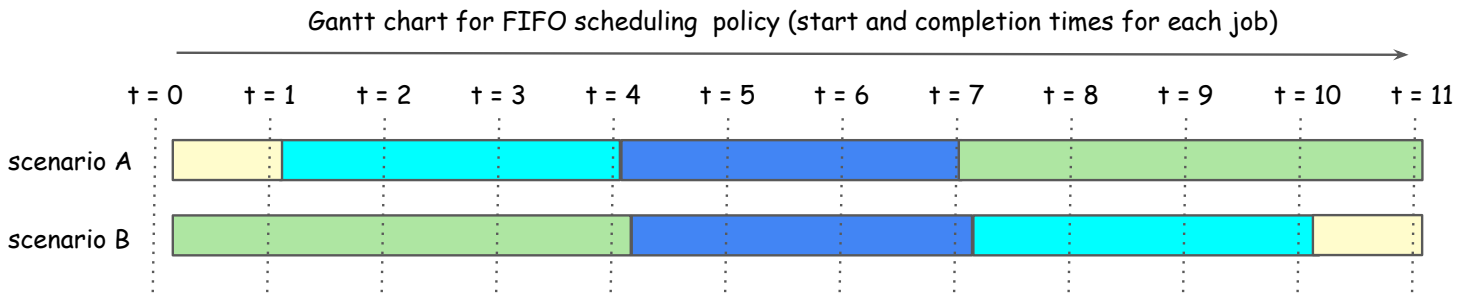
Calculating response time (latency)

> Average

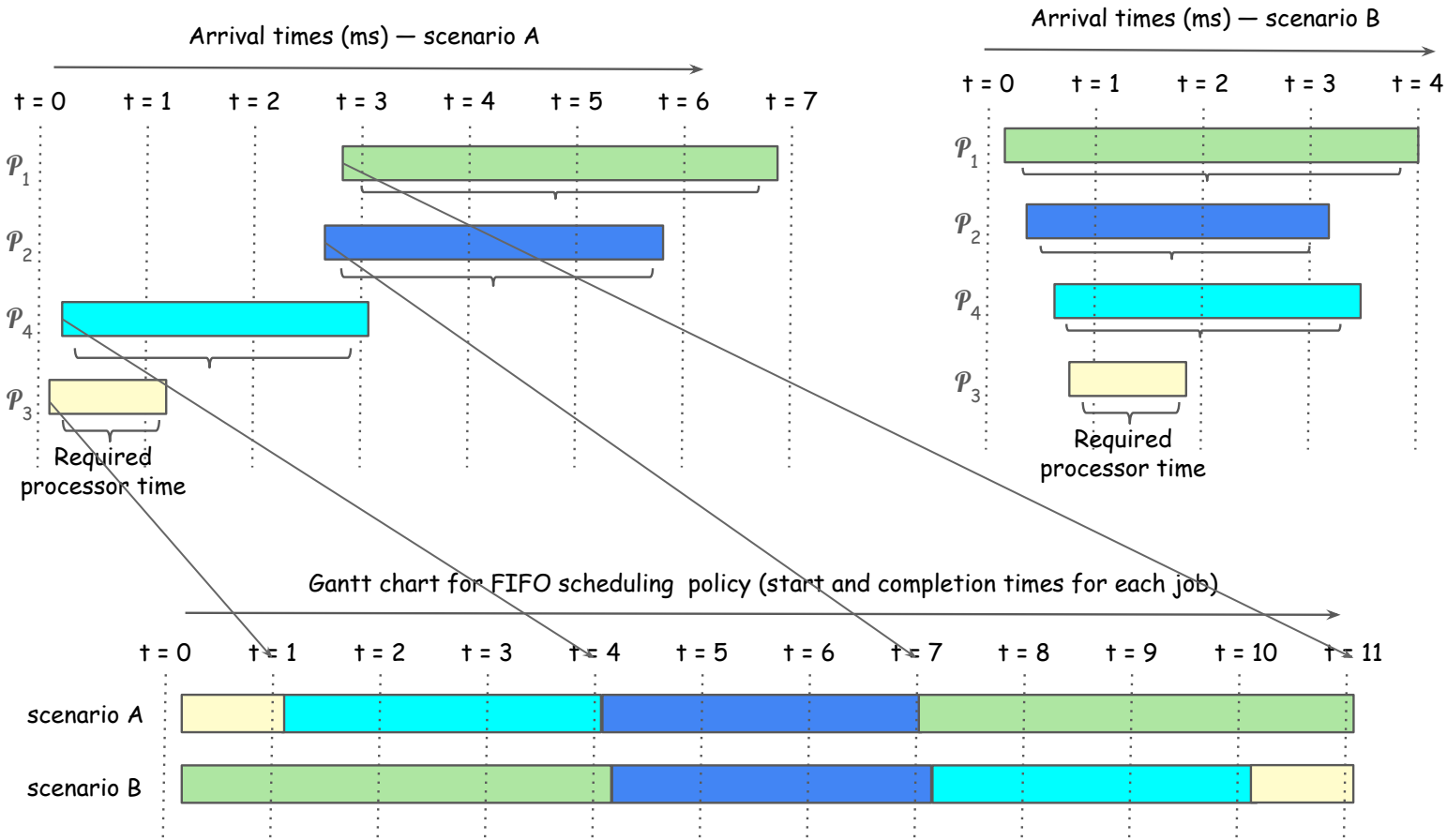
- **Scenario A**: (0ms + 1ms + 1ms + 4ms) / 4 = **1.5ms**
- **Scenario B**: (0ms + 4ms + 7ms + 10ms) / 4 = **5.25ms**

> Worst case

- **Scenario A**: (0ms + 1ms + 1ms + **4ms**) => **4ms**
- **Scenario B**: (0ms + 4ms + 7ms + **10ms**) => **10ms**



Calculating completion time



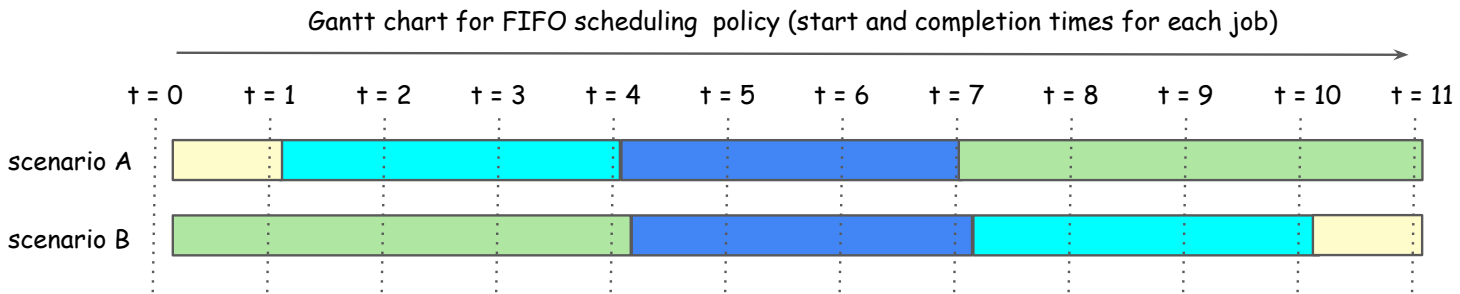
Calculating completion time

> Average

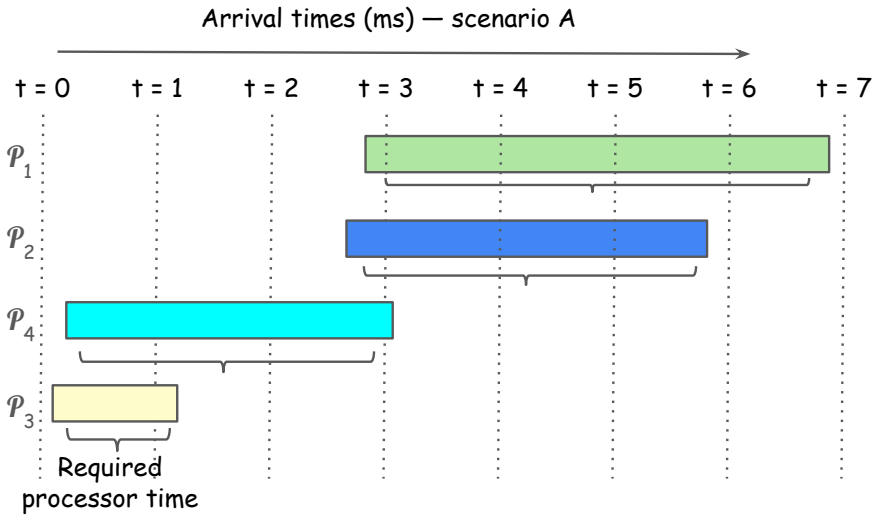
- **Scenario A**: (1ms + 4ms + 4ms + 8ms) / 4 = **4.25ms**
- **Scenario B**: (4ms + 7ms + 10ms + 11ms) / 4 = **8ms**

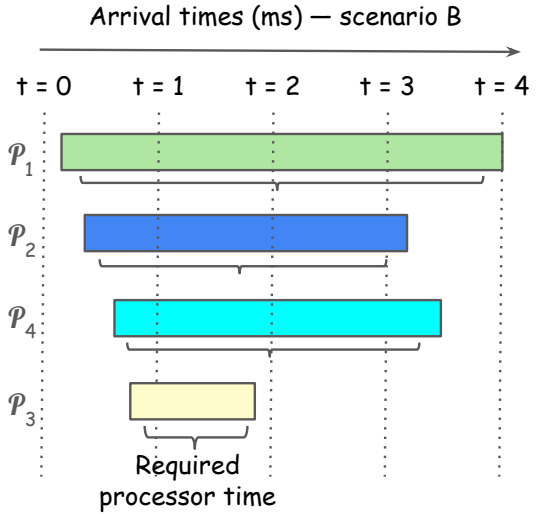
> Completion time (worst)

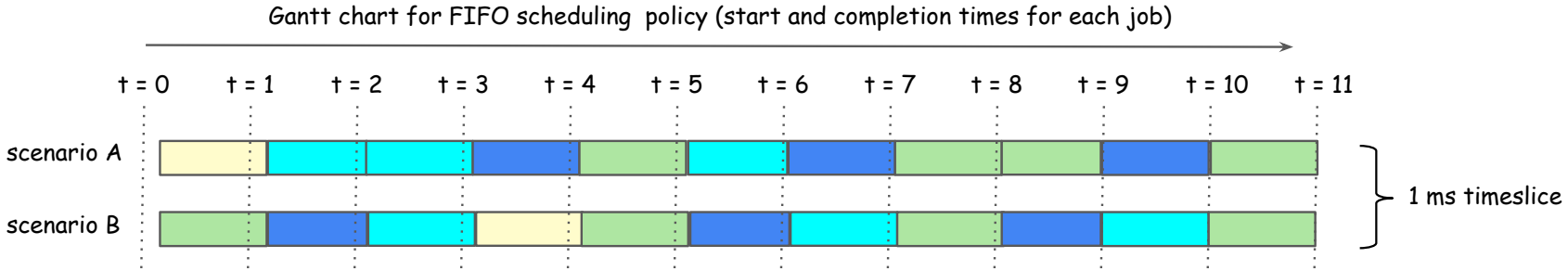
- **Scenario A**: (1ms + 4ms + 4ms + **8ms**) => **8ms**
- **Scenario B**: (4ms + 7ms + 10ms + **11ms**) => **11ms**



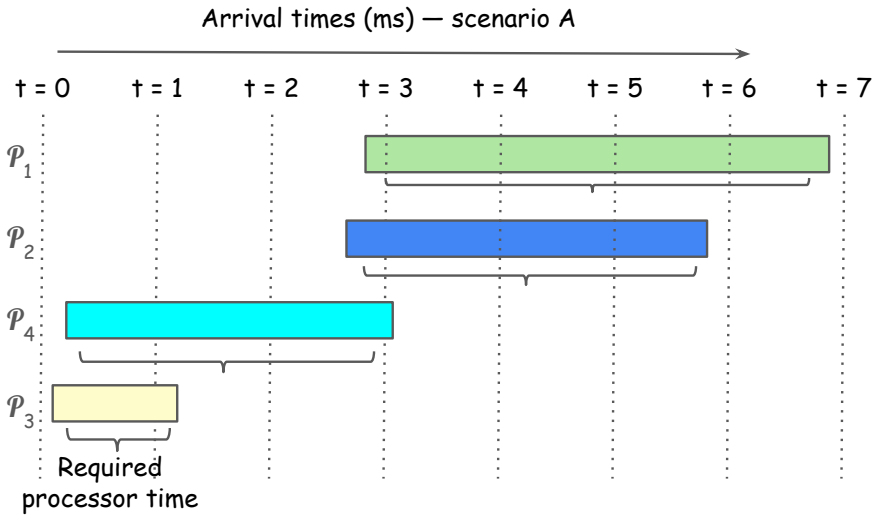
Round Robin scheduling policy (SCHED_RR)

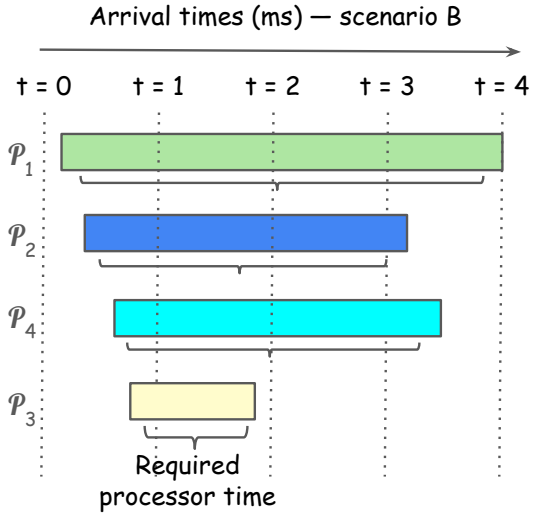


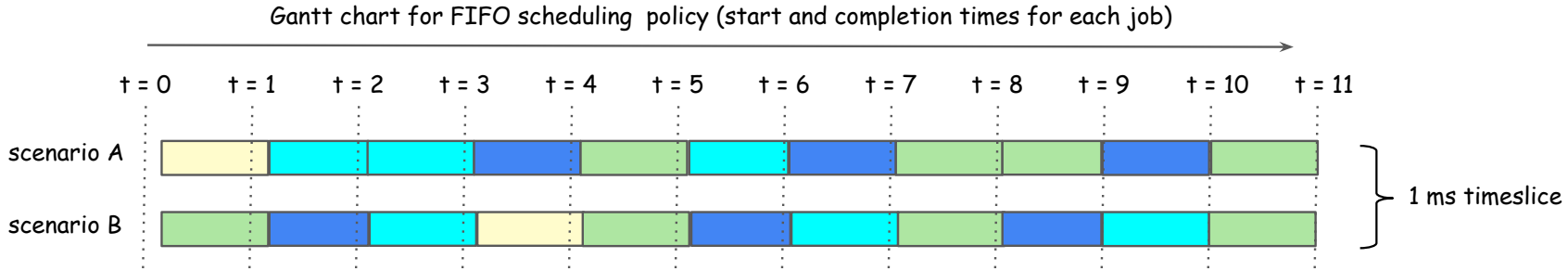




Calculating response time (latency)







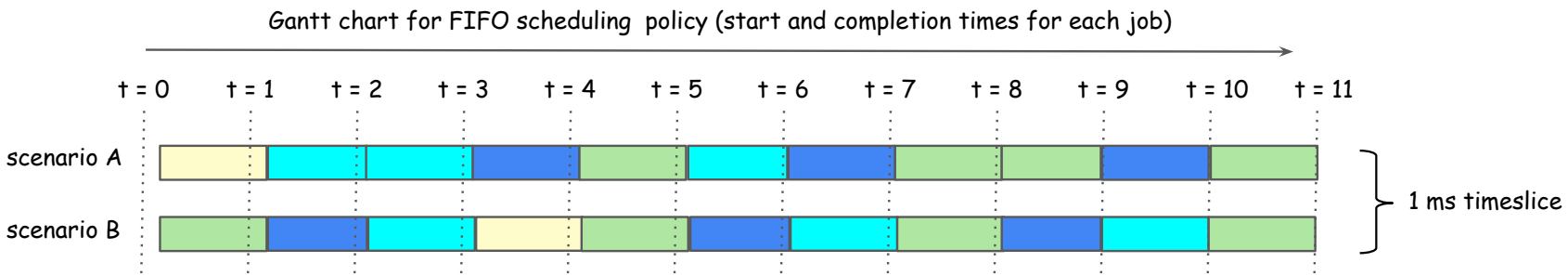
Calculating response time (latency)

› Average

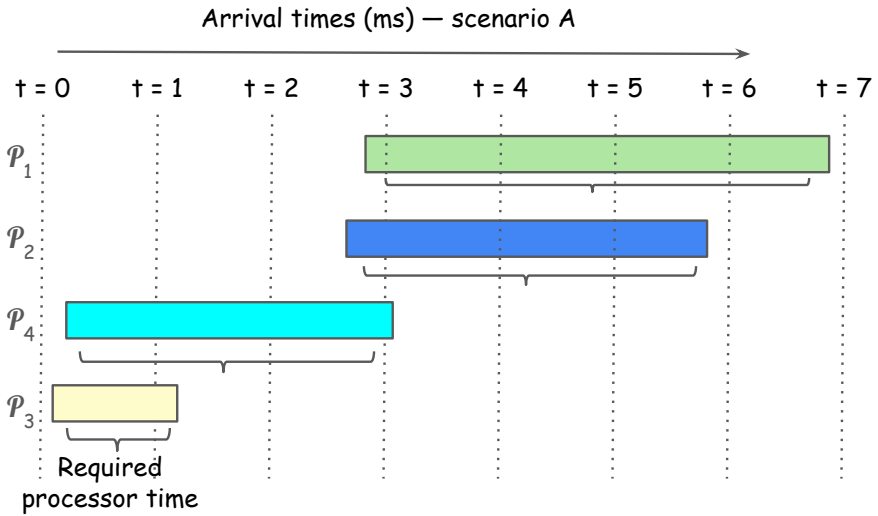
- **Scenario A**: (0ms + 1ms + 0ms + 1ms) / 4 = **0.5ms**
- **Scenario B**: (0ms + 1ms + 2ms + 3ms) / 4 = **1.5ms**

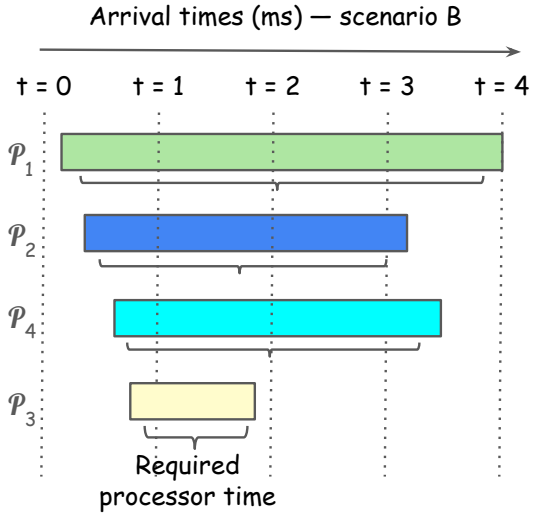
› Worst case

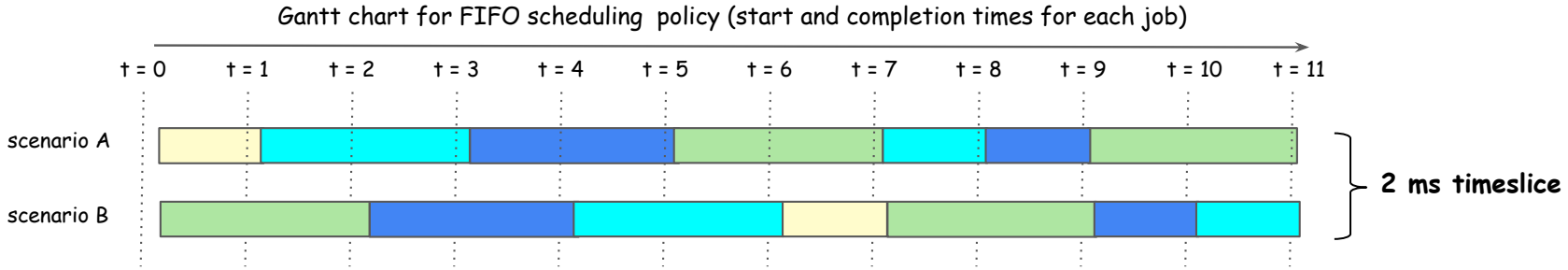
- **Scenario A**: (0ms + 1ms + 0ms + **1ms**) => **1ms**
- **Scenario B**: (0ms + 1ms + 2ms + **3ms**) => **3ms**



Calculating response time (latency)







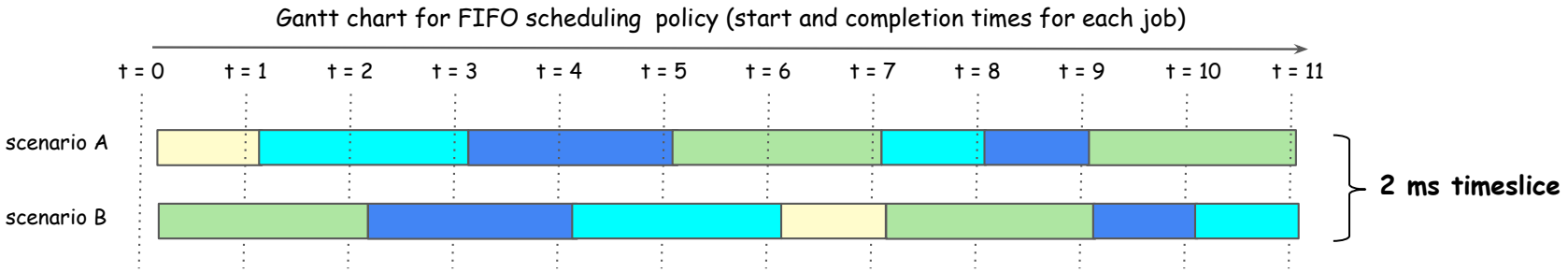
Calculating response time (latency)

› Average

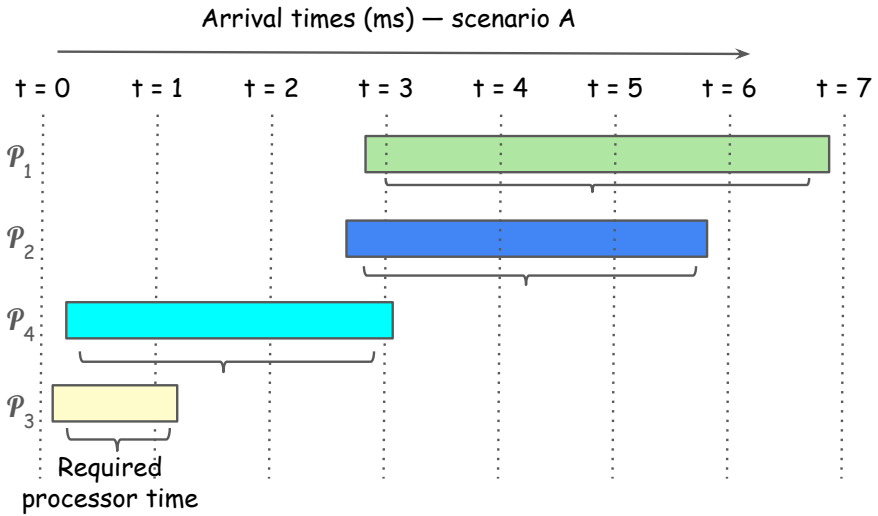
- **Scenario A**: (0ms + 1ms + 0ms + 3ms) / 4 = **1ms**
- **Scenario B**: (0ms + 2ms + 4ms + 6ms) / 4 = **3ms**

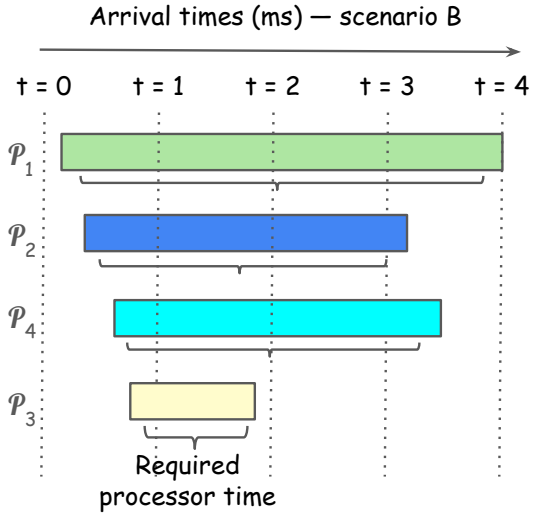
› Worst case

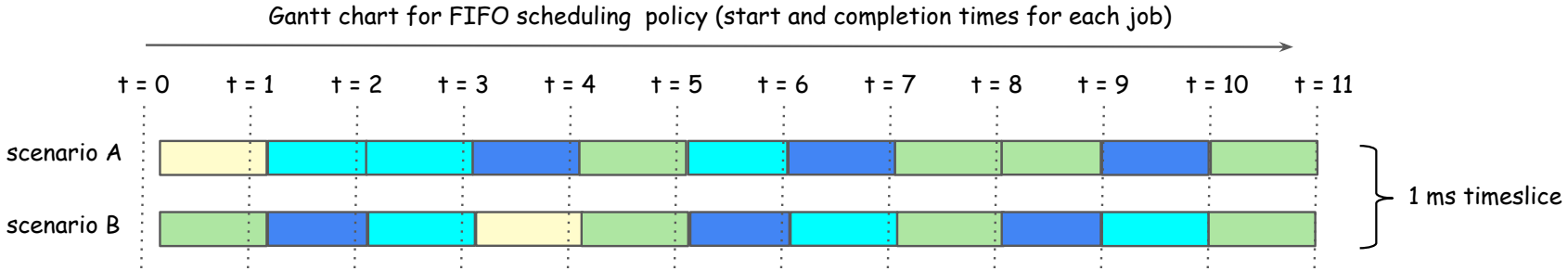
- **Scenario A**: (0ms + 1ms + 0ms + **3ms**) => **3ms**
- **Scenario B**: (0ms + 2ms + 4ms + **6ms**) => **6ms**



Calculating completion time







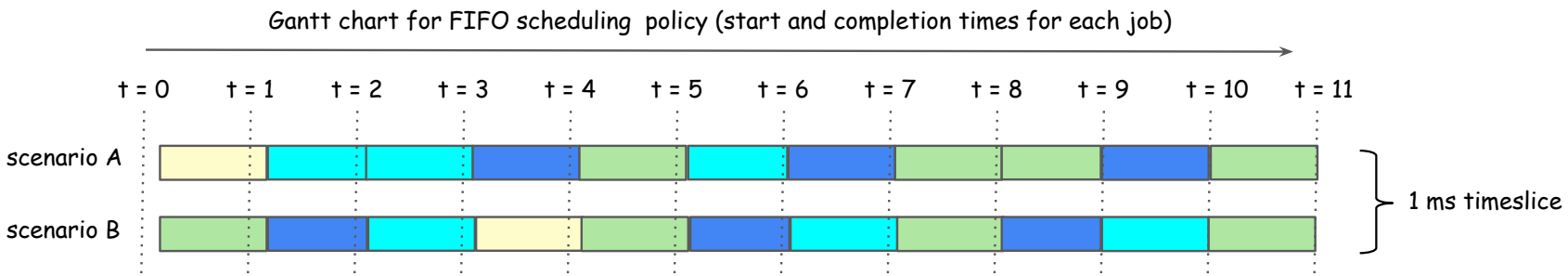
Calculating completion time

› Average

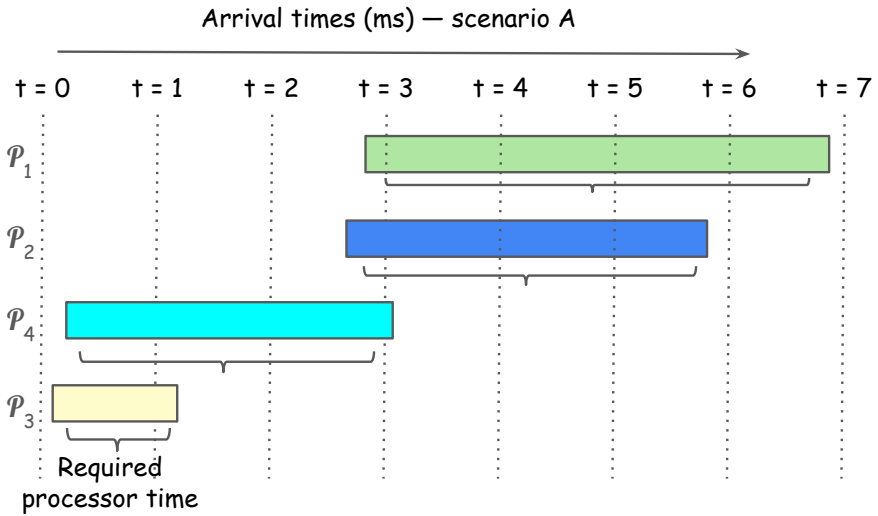
- **Scenario A**: (1ms + 6ms + 7ms + 8ms) / 4 = **5.5ms**
- **Scenario B**: (11ms + 10ms + 9ms + 4ms) / 4 = **8.5ms**

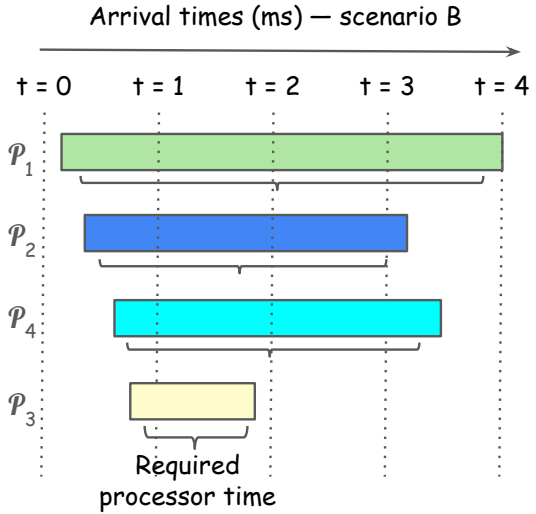
› Worst case

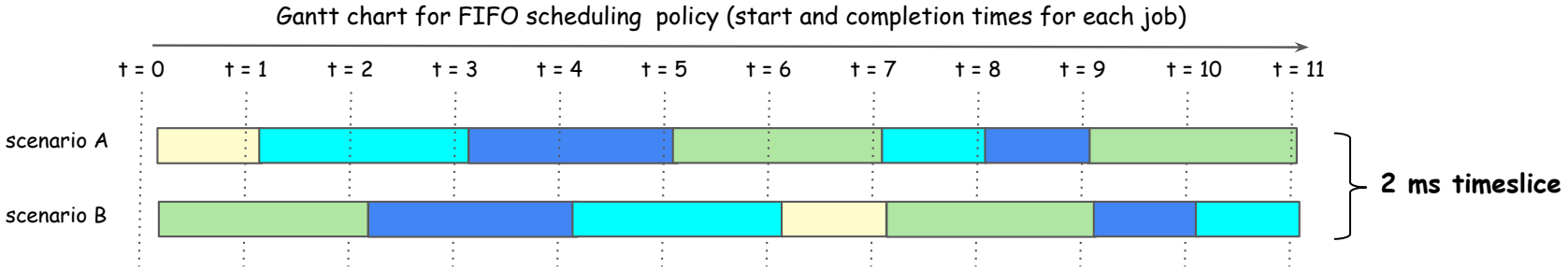
- **Scenario A**: (1ms + 6ms + 7ms + **8ms**) => **8ms**
- **Scenario B**: (**11ms** + 10ms + 9ms + 4ms) => **11ms**



Calculating completion time







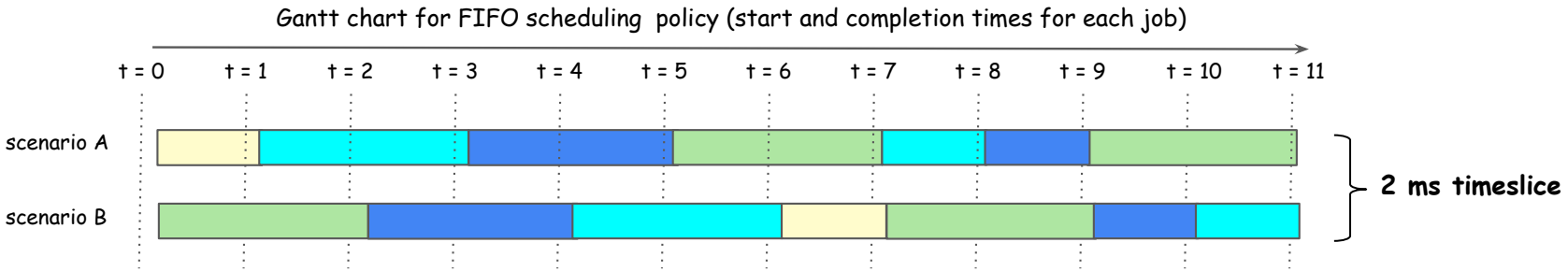
Calculating completion time

› Average

- **Scenario A**: (1ms + 3ms + 6ms + 8ms) / 4 = **4.5ms**
- **Scenario B**: (11ms + 10ms + 9ms + 7ms) / 4 = **9.25ms**

› Worst case

- **Scenario A**: (1ms + 3ms + 6ms + **8ms**) => **8ms**
- **Scenario B**: (**11ms** + 10ms + 9ms + 7ms) => **11ms**



Comparative view

- Response time
  - FIFO
    - Average: 1.5ms (sc.-A) vs. 5.25ms (sc.-B)
    - Worst: 4ms (sc.-A) vs. 10ms (sc.-B)
  - RR
    - Average: 0.5ms (sc.-A) vs. 1.5ms (sc.-B) / 1ms (sc.-A) vs. 3ms (sc.-B)
    - Worst: 1ms (sc.-A) vs. 3ms (sc.-B) / 3ms (sc.-A) vs. 6ms (sc.-B)

Comparative view

> Response time

>> FIFO

- Too much dependency on arrival times
- One long task may delay everyone (convoy effect)

>> RR

- Good pick when response time matters
- Dependency on the size of time slice => Need to balance number of ctx switches

Comparative view

> Response time

>> FIFO

- Too much dependency on arrival times
- One long task may delay everyone (convoy effect)

>> RR

- Good pick when response time matters => Need small timeslice
- Small time slice => Too many ctx switches / Large timeslice => Becomes FIFO

> Completion time

>> FIFO

- Average: 4.25ms (sc.-A) vs. 8ms (sc.-B)
- Worst: 8ms (sc.-A) vs. 11ms (sc.-B)

>> RR

- Average: 5.5ms (sc.-A) vs. 8.5ms (sc.-B) / 4.5ms (sc.-A) vs. 9.25ms (sc.-B)
- Worst: 8ms (sc.-A) vs. 11ms (sc.-B) / 8ms (sc.-A) vs. 11ms (sc.-B)

Comparative view

> Response time

>> FIFO

- Too much dependency on arrival times
- One long task may delay everyone (convoy effect)

>> RR

- Good pick when response time matters => Need small timeslice
- Small time slice => Too many ctx switches / Large timeslice => Becomes FIFO

> Completion time

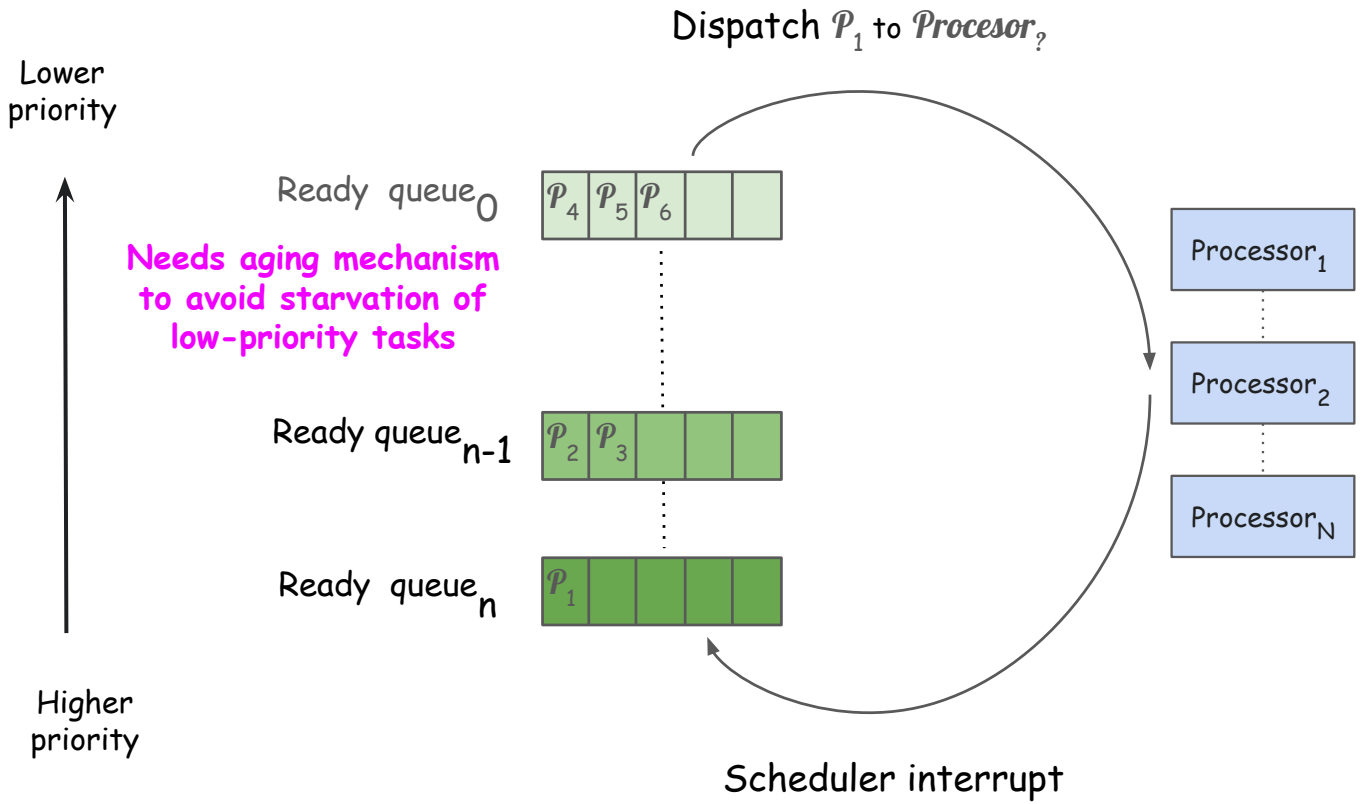
>> FIFO

- Good pick when completion time matters

>> RR

- Small timeslice => Bad pick for longer tasks
- Need to balance number of ctx switches

Hierarchical priority-based scheduling

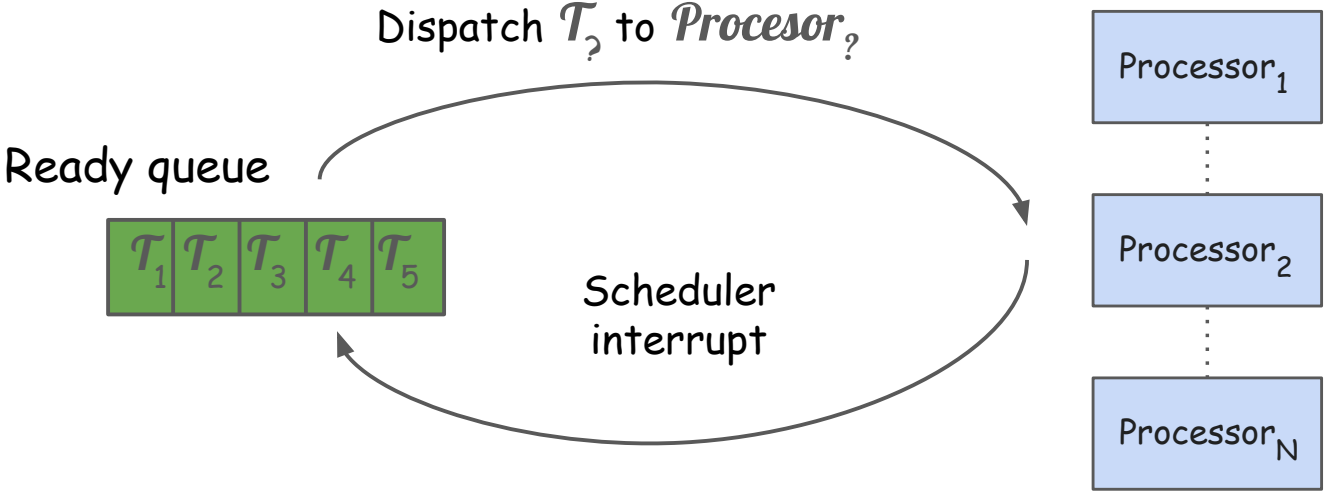


The Linux scheduler

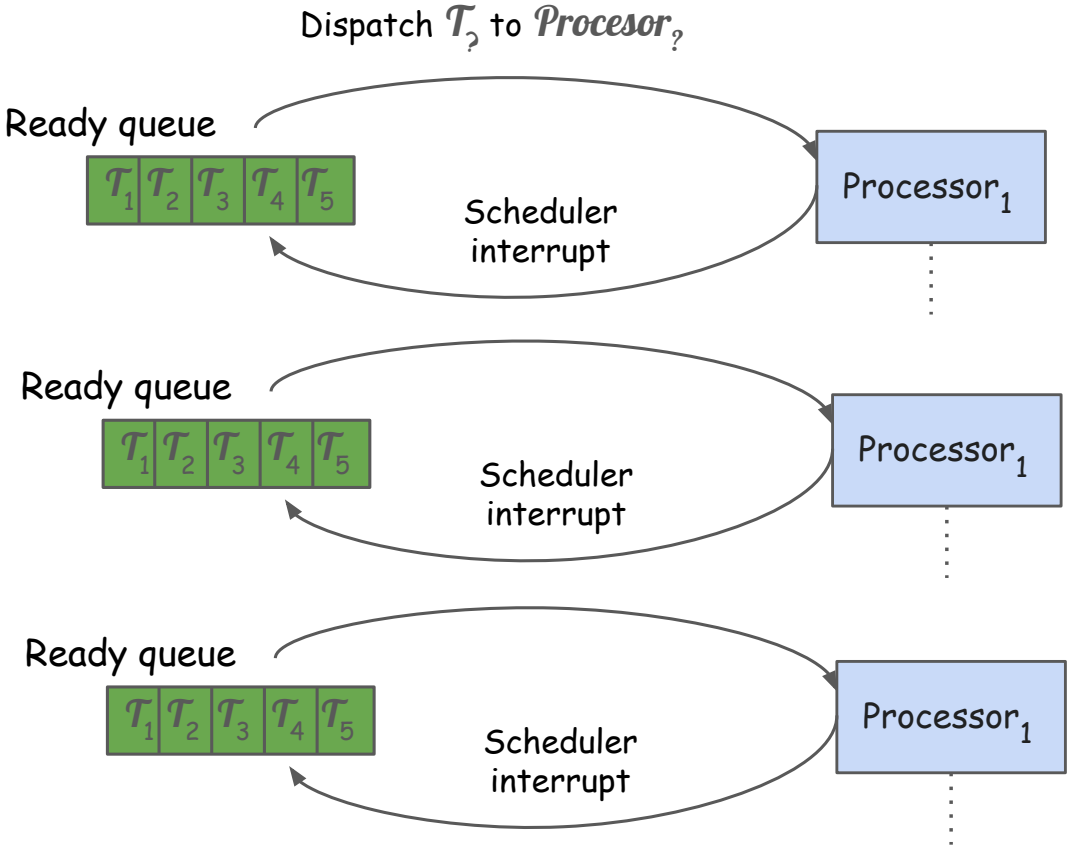
Precedence Order	Scheduler class	Implemented policies	Usecase	POSIX compliance
1	stop_sched_class	Run Linux kernel-internal tasks	Only used internally by the kernel; preempts anything running in the local processor	No
2	dl_sched_class	SCHED_DEADLINE	Hard real-time tasks whose execution deadlines must be met	No
3	rt_sched_class	SCHED_FIFO, SCHED_RR	Soft real-time tasks (e.g., audio daemon) with priorities [1–99]	Yes
4	cfs_sched_class, eevdf_sched_class	SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE	User tasks with "nice" values in the range [-20–19]	Partially Yes
5	idle_sched_class	Run the Linux kernel "idle" task	Runs when the local processor is idle, and has no other task to run	No

Unicore scheduling

> Given k tasks ready to run in a system with N available processors, which task should be dispatched to which processor at any given point in time?



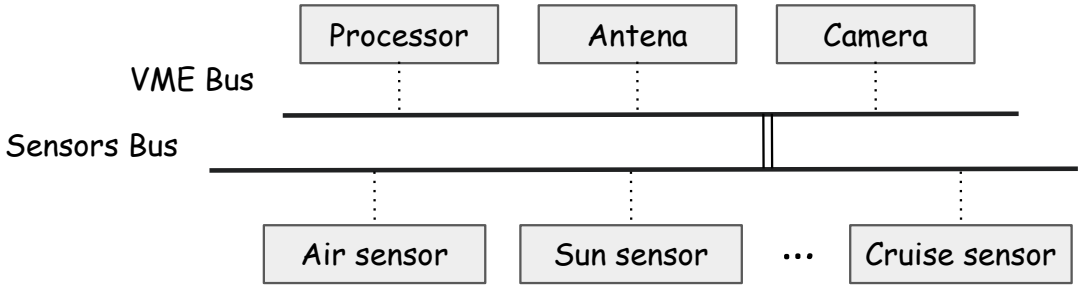
Multicore scheduling



Ready for your first bug in the outer universe

Mars Rover: software and hardware

- Works Real-Time Operating System (RTOS)
  - Tasks must meet strict timing constraints
  - Preemptive with priority-based scheduling
  - Scheduler ticks at 8 Hz (i.e., every 125ms)
- Hardware overview



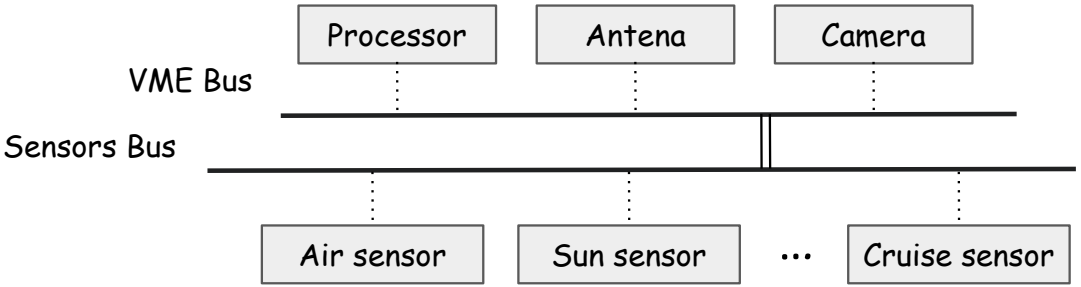
Mars Rover: software and hardware

- **Works Real-Time Operating System (RTOS)**
  - Tasks must meet strict timing constraints
  - Preemptive with priority-based scheduling
  - Scheduler ticks at 8 Hz (i.e., every 125ms)
- **Hardware overview**
  - Data from sensor bus to the VMA bus (to antenna)
  - Processor signal from VMA bus to sensor bus (cruise)

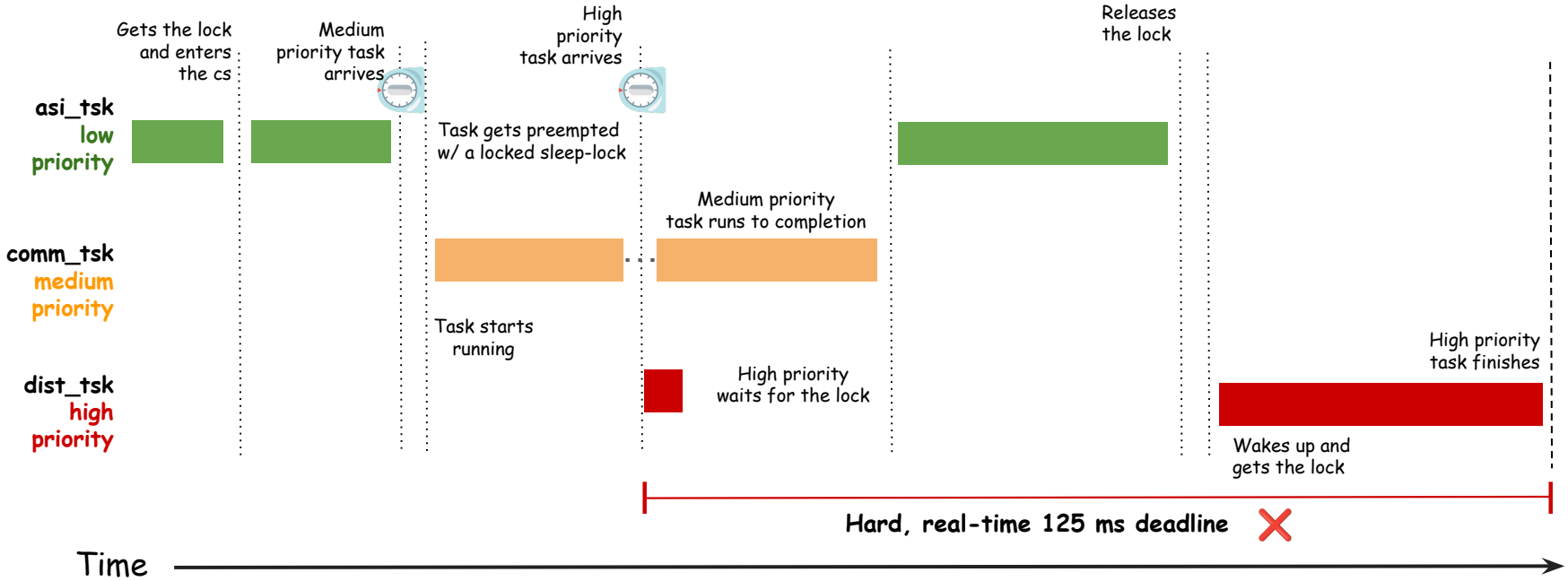
Mars Rover: software and hardware

- **Synchronization**
  - sched_tsk: Decides who transmits data next
  - dist_tsk: Decides who receives data next
  - comm_tsk: Uses the antenna to transmit data to Earth
  - asi_tsk: Uses the air sensor for scientific computations

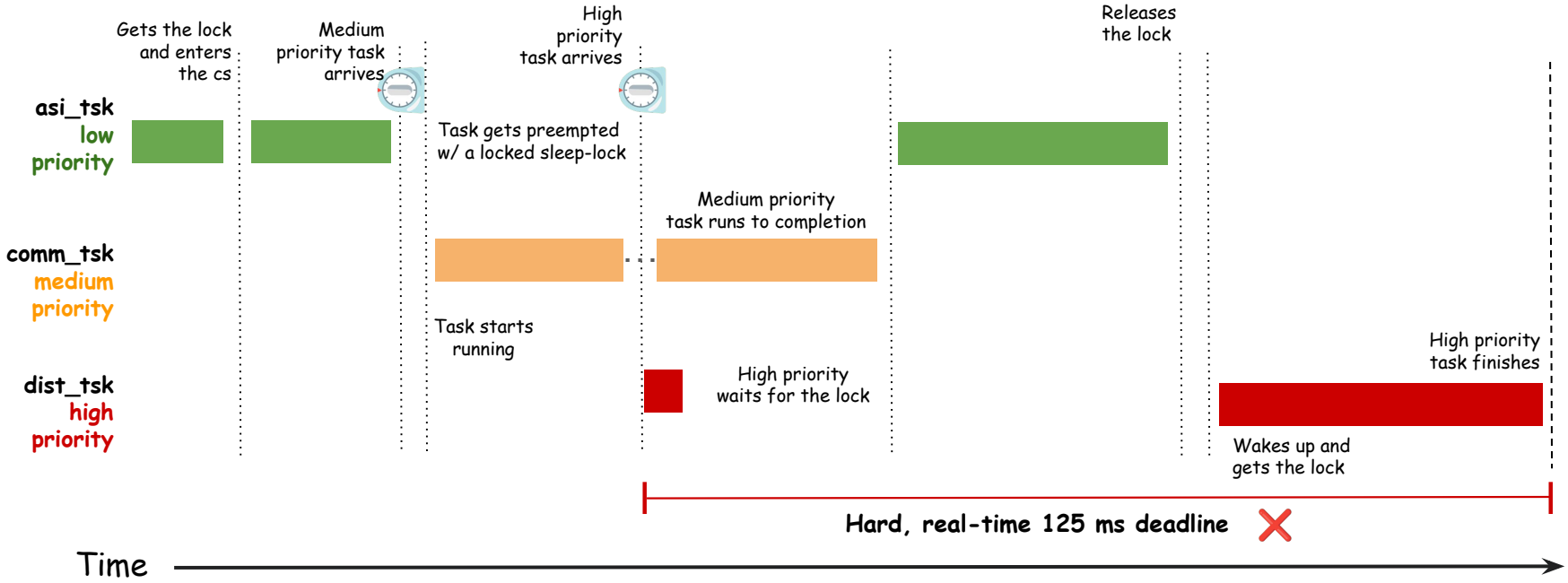
Priorities: sched_tsk > dist_tsk (high) > comm_tsk (medium) > asi_tsk (low)



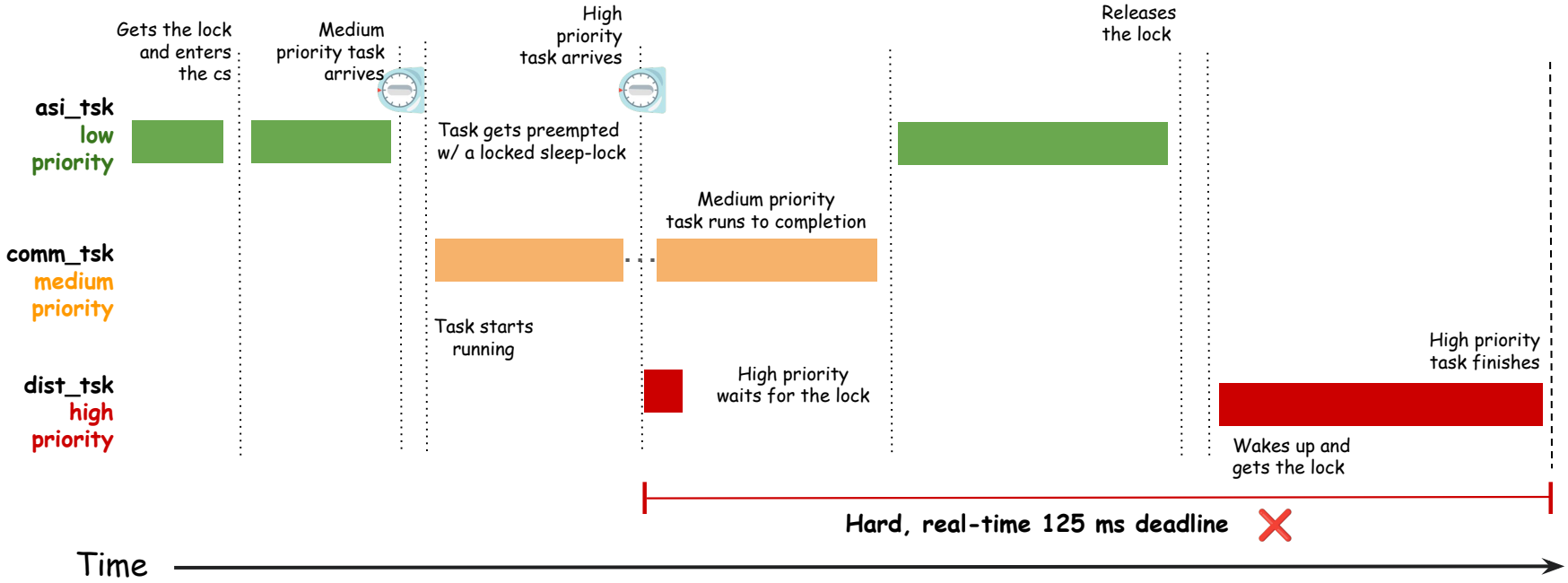
Ready for your first bug in the outer universe?



Classic example of priority inversion bug



Solution?


Gets the lock and enters the cs
Medium priority task arrives
High priority task arrives
Releases the lock
asi_tsk
low priority
Task gets preempted w/ a locked sleep-lock
Medium priority task runs to completion
comm_tsk
medium priority
Task starts running
High priority task finishes
dist_tsk
high priority
High priority waits for the lock
Wakes up and gets the lock
Hard, real-time 125 ms deadline
Time

How to manage stress?

> Personal advices on how to manage your stress

- Do not sleep long/well the night before an exam: Being well-rested on a situation that will hyperstimulate you is a bad idea
- Do not drink coffee for 7-8 hours before an exam: Same reasons, as above. You must minimize all source of hyperstimulation
- Decrease the levels of cortisol in your system: Cortisol is a confidence-killer hormon => Exercise few hours before to decrease it
- Disassociate yourself from the environment **during** the exam: Imaging that you are explaining the exam to someone who would easily get 100% ⇒ This someone is helping you silently in your head

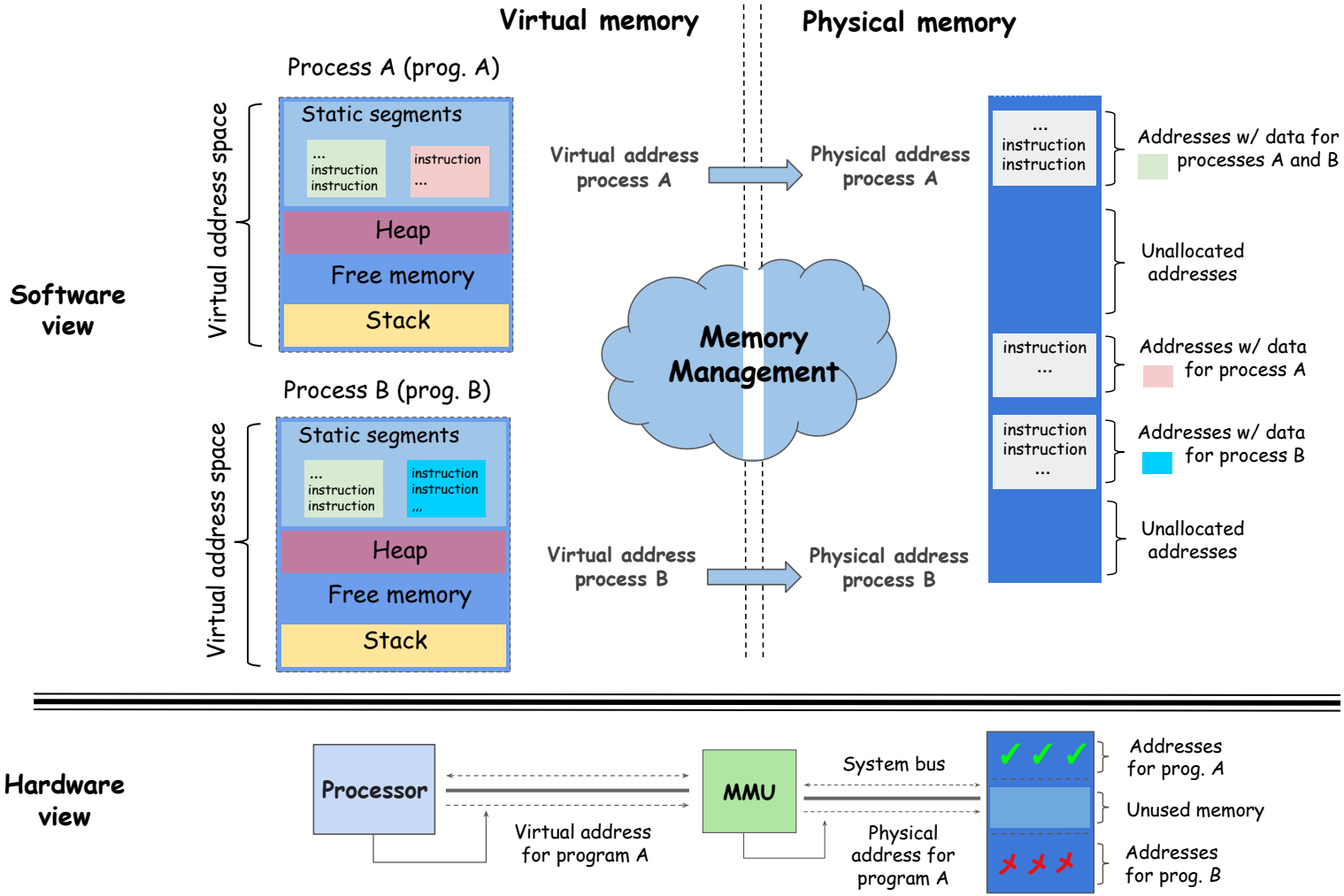
What is virtual memory?

>> A layer of abstraction

- Translates virtual addresses into physical addresses

  - **Virtual addr.**: the language processes talk to the processor about memory
  - **Physical addr.**: the language actual contents of memory are accessed

- The way programmers imagine programs interact with memory is an illusion on top of the virtual memory abstraction layer


Virtual memory
Physical memory
Software view
Process A (prog. A)
Virtual address space
Static segments
... instruction instruction
instruction ...
Heap
Free memory
Stack
Process B (prog. B)
Virtual address space
Static segments
... instruction instruction
instruction instruction ,,,
Heap
Free memory
Stack
Virtual address process A
Physical address process A
Virtual address process B
Physical address process B
Memory Management
... instruction instruction
Addresses w/ data for processes A and B
Unallocated addresses
instruction ...
Addresses w/ data for process A
instruction instruction ...
Addresses w/ data for process B
Unallocated addresses
Hardware view
Processor
Virtual address for program A
MMU
System bus
Physical address for program A
Addresses for prog. A
Unused memory
Addresses for prog. B

What is virtual memory? (demo)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

int main(int argc, char **argv) {
   int i;
   pid_t pid = getpid();

   i = atoi(argv[1]);

  printf("[PID: %d]; &i: %p; i: %d\n", pid, &i, i)
  printf("[PID: %d]; Sleep for %d sec\n", pid, i);
  sleep(i);
  printf("[PID: %d]; I am done now\n", pid, i);
  return 0;
}
```

```
➜ git:(master) ✗ echo 0 | sudo tee
/proc/sys/kernel/randomize_va_space
0

➜ vm git:(master) ✗ ./hello_vm 10 &
[1] 186394
[PID: 186394]; &i: 0xffffffffff210; i: 10
[PID: 186394]: Seep for 10 sec

➜ vm git:(master) ✗ ./hello_vm 3 &
[2] 186432
[PID: 186432]; &i: 0xffffffffff210; i: 3
[PID: 186432]: Sleep for 3 sec

➜ vm git:(master) ✗ [PID: 186432]; I am done now
[2] + 186432 done       ./hello_vm 3

➜ vm git:(master) ✗ [PID: 186394]; I am done now
[1] + 186394 done       ./hello_vm 10
```

Same address different value, both programs running…

Why need virtual memory?

- Fault isolation: Anything that goes wrong in the address space of one process does not affect any other process
- Illusion of continuous memory: Programs are written, compiled, assembled, linked, and loaded assuming access to **continuous** memory
- Frugal use of resources
  - >> Demand paging: Postpone allocating physical memory until necessary
  - >> Copy-On-Write: One physical replica of common readed-only code
- Performant use of resources
  - >> Programs have a memory footprint as much as the size of storage and run at speed close to the speed of CPU caches (how?...)
  - >> **CoW**: Laziness -> defering Vs. **Locality**: Proactivity -> prefetching

How is virtual memory implemented?

"We've rewritten the VM several times in the last ten years, and I expect it will be changed several more times in the next few years. Within five years, we'll almost certainly have to make the current three-level page tables be four levels, etc."

—Linus Torvald, 2001.

How is virtual memory implemented?

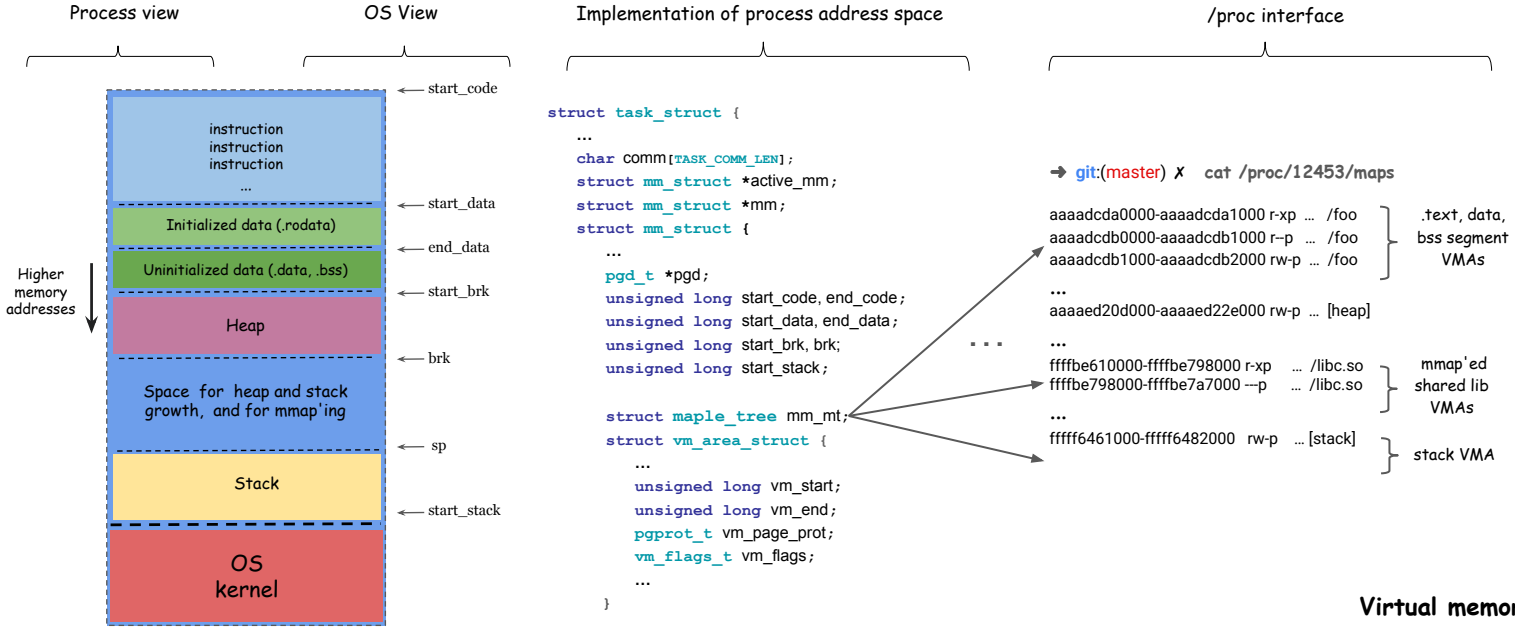
- Usually considered the most complex kernel subsystem
  - >> I strongly disagree => It's simple if you view it as follows
  - It is just an index…Ask the right questions

  Q1: What are the entries of the index?

  Q2: How are the entries of the index used?

  Q3: How are the entries of the index allocated?

  Q4: How are the entries of the index replaced?


Process view
OS View
Implementation of process address space
/proc interface
instruction
instruction
instruction
...
Initialized data (.rodata)
Uninitialized data (.data, .bss)
Heap
Space for heap and stack growth, and for mmap'ing
Stack
OS kernel
Higher memory addresses
start_code
start_data
end_data
start_brk
brk
sp
start_stack
struct task_struct {
...
char comm[TASK_COMM_LEN];
struct mm_struct *active_mm;
struct mm_struct *mm;
struct mm_struct {
...
pgd_t *pgd;
unsigned long start_code, end_code;
unsigned long start_data, end_data;
unsigned long start_brk, brk;
unsigned long start_stack;
struct maple_tree mm_mt;
struct vm_area_struct {
...
unsigned long vm_start;
unsigned long vm_end;
pgprot_t vm_page_prot;
vm_flags_t vm_flags;
...
}
➜ git:(master) ✗ cat /proc/12453/maps
aaaadcda0000-aaaadcda1000 r-xp ... /foo
aaaadcdb0000-aaaadcdb1000 r--p ... /foo
aaaadcdb1000-aaaadcdb2000 rw-p ... /foo
.text, data, bss segment VMAs
aaaaed20d000-aaaaed22e000 rw-p ... [heap]
ffffbe610000-ffffbe798000 r-xp ... /libc.so
ffffbe798000-ffffbe7a7000 ---p ... /libc.so
mmap'ed shared lib VMAs
ffffbe6461000-ffffbe6482000 rw-p ... [stack]
stack VMA
Virtual memory
Physical memory
Kernel space addresses - Page table entries for current process
Privileged register loaded with the base of current process's page table index
%cr3
PGD
PUD
PMD
PTE
PFN | perms
PTE base
PMD base
PUD base
Translation
User space addresses
Target page
Pages of current process

How is virtual memory implemented?

"We've rewritten the VM several times in the last ten years, and I expect it will be changed several more times in the next few years. Within five years, we'll almost certainly have to make the current three-level page tables be four levels, etc."

—Linus Torvald, 2001.

How is virtual memory implemented?

> Usually considered the most complex kernel subsystem

- It's simple if you view it as an index

Ask the right questions

Q1: What are the entries of the index?

Q2: How are the entries of the index used?

Q3: How are the entries of the index allocated?

Q4: How are the entries of the index replaced?

How is virtual memory implemented?

> Usually considered the most complex kernel subsystem

- It's simple if you view it as an index

Ask the right questions

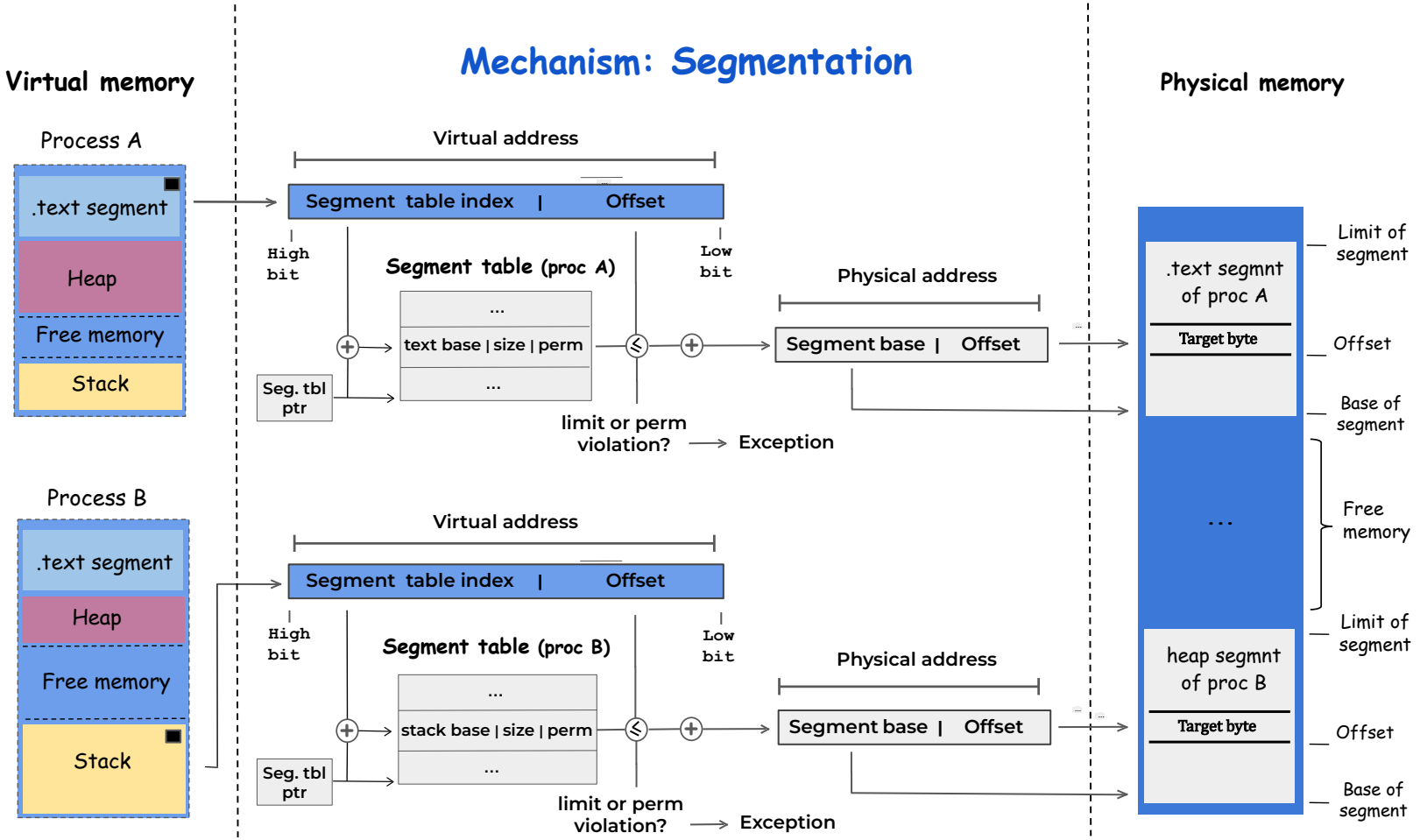
Q1: What are the entries of the index?

Q2: How are the entries of the index used?

Q3: How are the entries of the index allocated?

Q4: How are the entries of the index replaced?

Mechanism: Segmentation



Mechanism: Segmentation

› Assume 32-bit virtual and physical address space
w/ 8-bits for the segment table index (segment selector) and 24-bits for offset

Segment table (process A)

Seg No	Base	Size	Perm
0	0x4f00000	0xffffff	rw-
1	0x0100000	0x1fffff	r-x
2	0x3000000	0x9fffff	r--

Q1: How many segments can a process have?

- 2^8 = 256 segments

Q2: How large can a segment be in bytes?

- 2^24 bytes = 16*(2^20) = 16 MiB

Q3: What's the physical address for virtual address?

- 0x00001030 = [0000 0000] [0000 0000] [0001 0000] [0011 0000]
  = [0100 11111] [0000 0000] [0001 0000] [0011 0000]
  = 0x4f 00 10 30 => 00 10 30 < ff ff ff ? **OK**
  = 0x4f001030

Mechanism: Segmentation

› Assume 32-bit virtual and physical address space
w/ 8-bits for the segment table index (segment selector) and 24-bits for offset

Segment table (process A)

Seg No	Base	Size	Perm
0	0x4f00000	0xffffff	rw-
1	0x0100000	0x1fffff	r-x
2	0x3000000	0x9fffff	r--

Q1: How many segments can a process have?

- 2^8 = 256 segments

Q2: How large can a segment be in bytes?

- 2^24 bytes = 16*(2^20) = 16 MiB

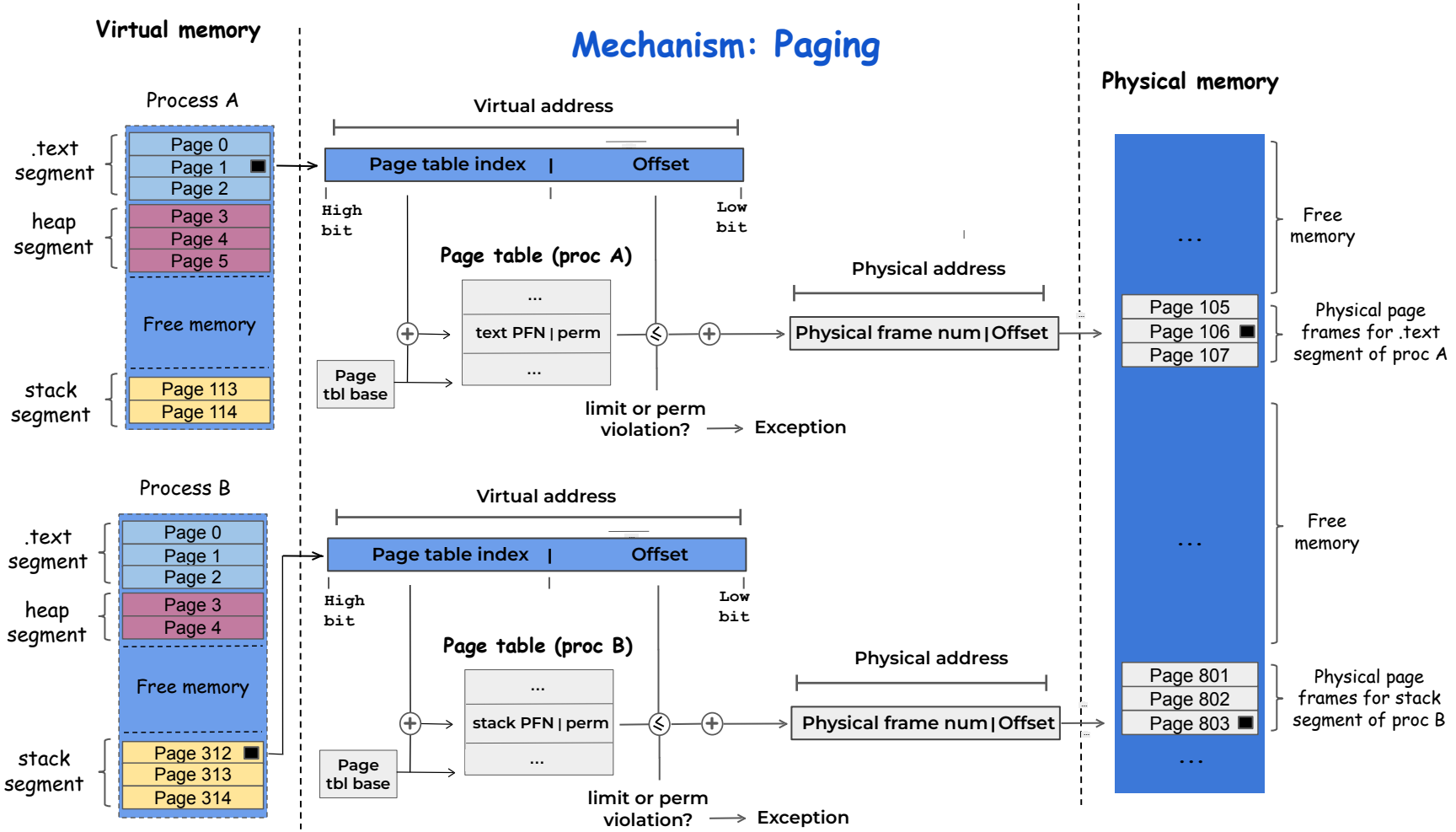
Q3: What's the physical address for virtual address?

- 0x02a20357 = [0000 0010] [1010 0010] [0000 0011] [0101 1111]
= [0011 0000] [1010 0010] [0000 0011] [0101 1111]
= 0x30 a2 03 5f => a2 03 5f < 9f ff ff ? NOT_OK

Mechanism: Segmentation

Limitations of Segmentation

- Fragmentation: Inability to use available memory
  - Internal fragmentation: Unused portions internally on each segment
  - External fragmentation: Free segments not usable due to unfit sizes


Mechanism: Paging
Virtual memory
Process A
.text segment
heap segment
stack segment
Page 0
Page 1
Page 2
Page 3
Page 4
Page 5
Free memory
Page 113
Page 114
Virtual address
Page table index | Offset
High bit
Low bit
Page table (proc A)
...
text PFN | perm
...
Page tbl base
Physical address
Physical frame num | Offset
limit or perm violation?
Exception
Physical memory
...
Free memory
Page 105
Page 106
Page 107
Physical page frames for .text segment of proc A
Free memory
Process B
.text segment
heap segment
Free memory
stack segment
Page 0
Page 1
Page 2
Page 3
Page 4
Page 312
Page 313
Page 314
Virtual address
Page table index | Offset
High bit
Low bit
Page table (proc B)
...
stack PFN | perm
...
Page tbl base
Physical address
Physical frame num | Offset
limit or perm violation?
Exception
Free memory
Page 801
Page 802
Page 803
Physical page frames for stack segment of proc B
...

Mechanism: Paging

› Assume 32-bit virtual and physical address space, 20-bits for page table index and 4KB pages (i.e., 12 bits for offset)

Page table (process A)

Index	PFN	Perm
0	302	rw-
1	**106**	**r-x**
2	19	r--

Q: What's the physical address for virtual address?

- 0x00000400 = [0000 0000] [0000 0000] [0000 0100] [0000 0000]
  = [0000 0000] [0001 0010] [1110 0100] [0000 0000]
  = 0x00 12 e4 00 = 0x0012e400

- 0x00001402 = [0000 0000] [0000 0000] [0001 0100] [0000 0010]
  = [0000 0000] [0000 0110] [1010 0100] [0000 0010]
  = 0x00 06 a4 02 = 0x0006a402

Mechanism: Paging

Limitations of segmentation

> Fragmentation: Inability to use available memory

>> Internal fragmentation: Unused portions internally on each segment

>> External fragmentation: Free segments not usable due to unfit sizes

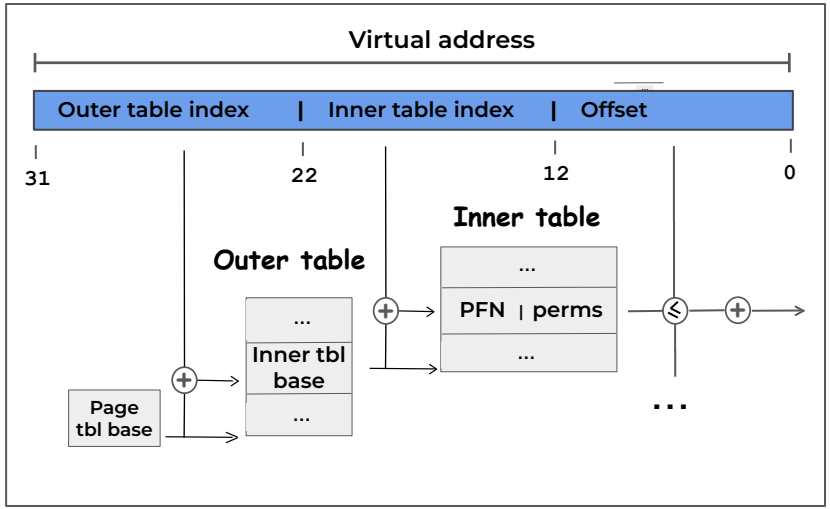
Limitations of single-level paging

> Size of page table: E.g., with 32-bit virtual addresses, need a 4MB flat array per process to map a sparse 4GB address space

> Internal fragmentation: Still an issue, but not as prominent

Mechanism: 2-level hierarchical paging

› Assume 32-bit virtual and physical address space, where the 12 lower bits are for the offset, the 10 middle bits are the index for the inner table, and the 10 higher bits are the index for the outer table



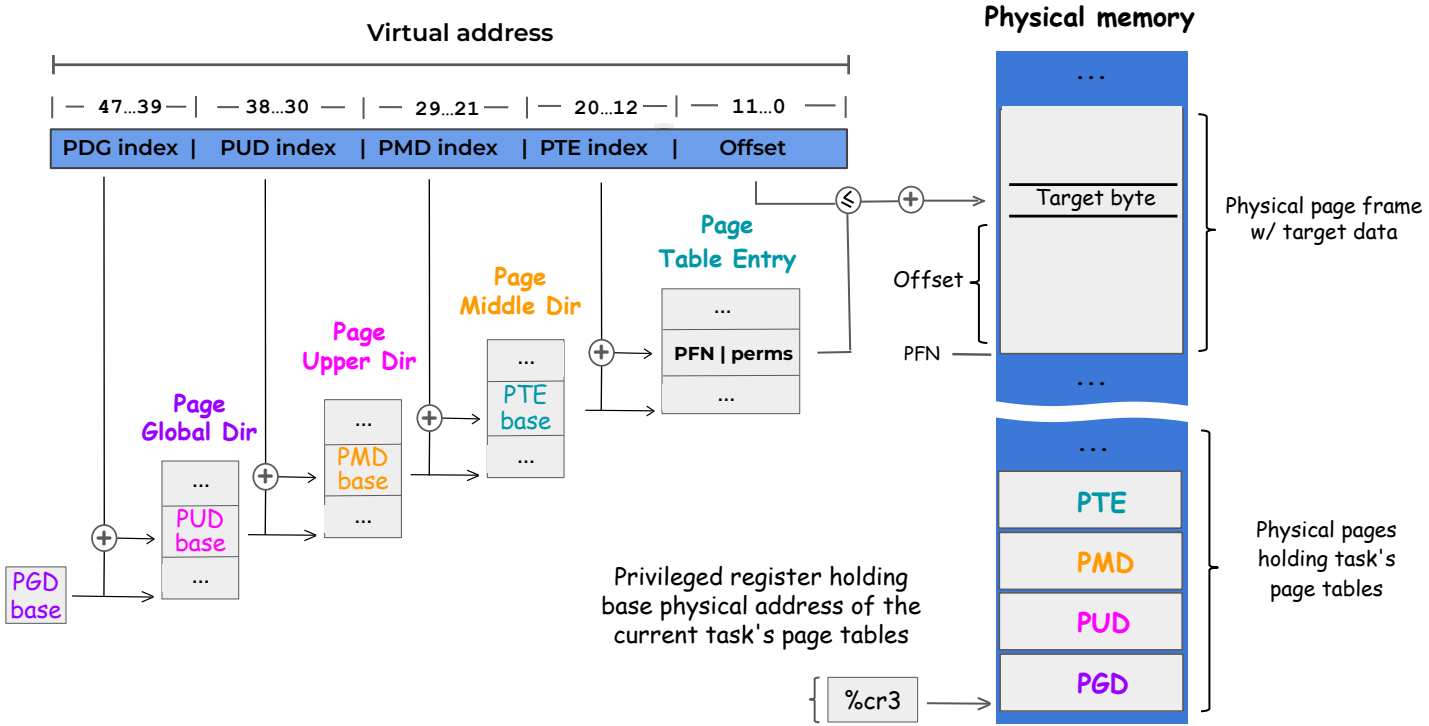
Q: Size of "map-able" address space?

› **With two pages** (one for each of the outer and inner table): 1 * 1024 PTEs * 4KB = 4MiB

› With each additional page in the inner table (i.e., 1024 PTEs), 4 additional MiB can be mapped

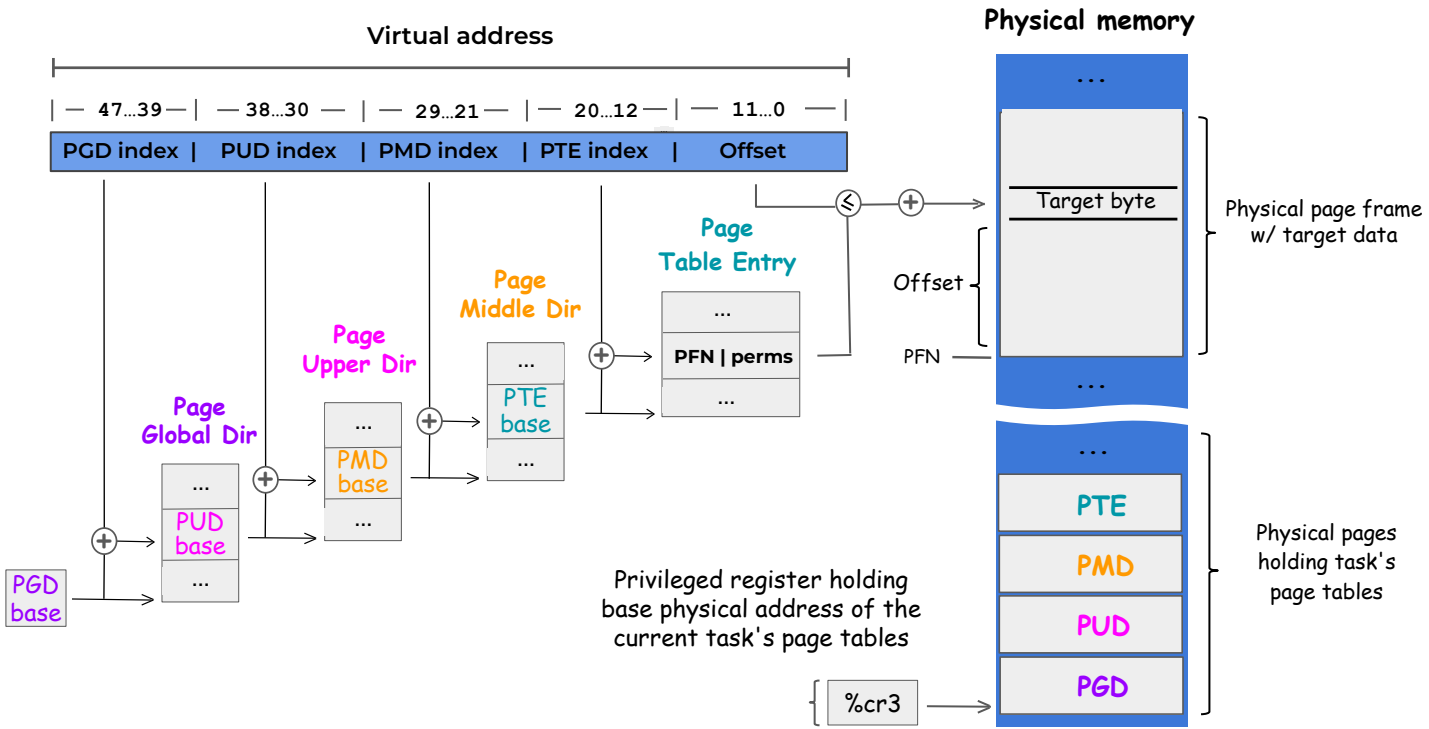
There is another positive side-effect…hang on…

Mechanism: Multi-level paging



› See relevant linux definitions: here

Mechanism: Multi-level paging



> What is the reach of one page of PTE/PMD/PUD/PGD entries? **Why care?**

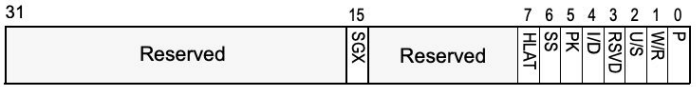
x86 PTE format that maps a 4KB page

63	62–61	60–59	58–52	51–M[1]	M-1–32	31–12	11–9	8	7	6	5	4	3	2	1	0	
XD	Prot. Key	Ignored	Rsvd.			Address of 4KB page frame	R	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																	0	PTE: not present

0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 5.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 5.8)

> See Intel Software Developer's Manual

Error codes on x86 Page fault (PGD @%cr3; faulting addr. @%cr2)



This will be placed to the stack by hardware

P	0	The fault was caused by a non-present page.
	1	The fault was caused by a page-level protection violation.
W/R	0	The access causing the fault was a read.
	1	The access causing the fault was a write.
U/S	0	A supervisor-mode access caused the fault.
	1	A user-mode access caused the fault.
RSVD	0	The fault was not caused by reserved bit violation.
	1	The fault was caused by a reserved bit set to 1 in some paging-structure entry.
I/D	0	The fault was not caused by an instruction fetch.
	1	The fault was caused by an instruction fetch.
PK	0	The fault was not caused by protection keys.
	1	There was a protection-key violation.
SS	0	The fault was not caused by a shadow-stack access.
	1	The fault was caused by a shadow-stack access.
HLAT	0	The fault occurred during ordinary paging or due to access rights.
	1	The fault occurred during HLAT paging.
SGX	0	The fault is not related to SGX.
	1	The fault resulted from violation of SGX-specific access-control requirements.

› See Intel Software Developer's Manual (p. 3213)

Checking the address space size of your machine

```
→  ~ cat /proc/$$/maps | head
aaaaab8d0000-aaaaab9a3000 r-xp 00000000 08:02 2375001
aaaaab9b3000-aaaaab9b5000 r--p 000d3000 08:02 2375001
aaaaab9b5000-aaaaab9bb000 rw-p 000d5000 08:02 2375001
aaaaab9bb000-aaaaab9cf000 rw-p 00000000 00:00 0
aaaaac024000-aaaaac40b000 rw-p 00000000 00:00 0
ffffb82b0000-ffffb8530000 r--s 00000000 08:02 2890570
ffffb8530000-ffffb853f000 r-xp 00000000 08:02 2495184
ffffb853f000-ffffb854e000 ---p 0000f000 08:02 2495184
ffffb854e000-ffffb854f000 r--p 0000e000 08:02 2495184
ffffb854f000-ffffb8550000 rw-p 0000f000 08:02 2495184
```

- Why am I seeing 12 hexadecimal digits?

Checking the address space size of your machine

```
→  ~ cat /proc/$$/maps | head
aaaaab8d0000-aaaaab9a3000 r-xp 00000000 08:02 2375001
aaaaab9b3000-aaaaab9b5000 r--p 000d3000 08:02 2375001
aaaaab9b5000-aaaaab9bb000 rw-p 000d5000 08:02 2375001
aaaaab9bb000-aaaaab9cf000 rw-p 00000000 00:00 0
aaaaac024000-aaaaac40b000 rw-p 00000000 00:00 0
ffffb82b0000-ffffb8530000 r--s 00000000 08:02 2890570
ffffb8530000-ffffb853f000 r-xp 00000000 08:02 2495184
ffffb853f000-ffffb854e000 ---p 0000f000 08:02 2495184
ffffb854e000-ffffb854f000 r--p 0000e000 08:02 2495184
ffffb854f000-ffffb8550000 rw-p 0000f000 08:02 2495184
→  ~ cat /boot/config-`uname -r` | grep "VA_BITS\|PA_BITS"
# CONFIG_ARM64_VA_BITS_39 is not set
CONFIG_ARM64_VA_BITS_48=y
CONFIG_ARM64_VA_BITS=48
CONFIG_ARM64_PA_BITS_48=y
CONFIG_ARM64_PA_BITS=48
```

How is virtual memory implemented?

> Usually considered the most complex kernel subsystem

- It's simple if you view it as an index

Ask the right questions

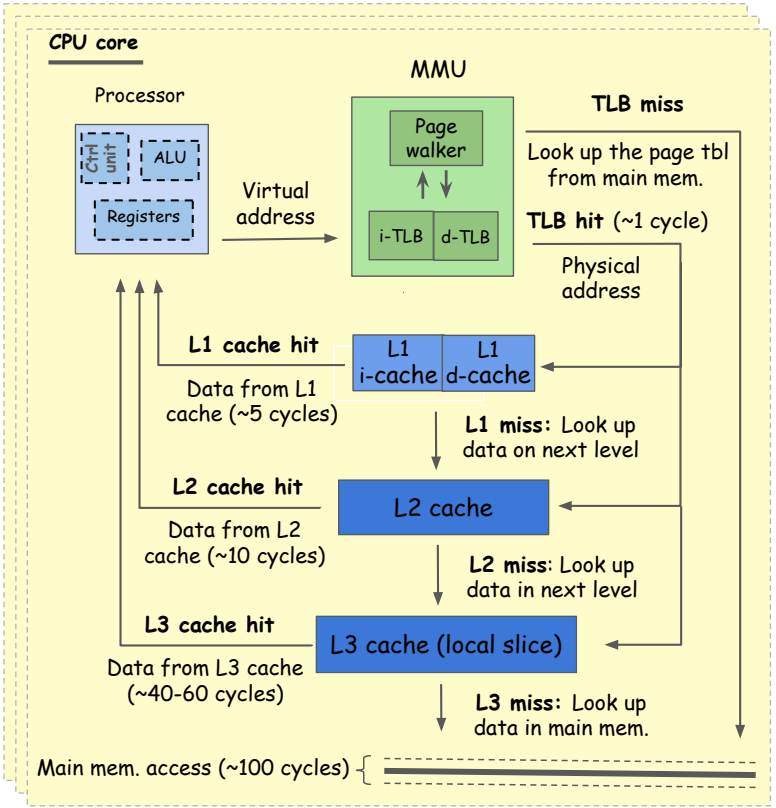
~~Q1: What are the entries of the index?~~

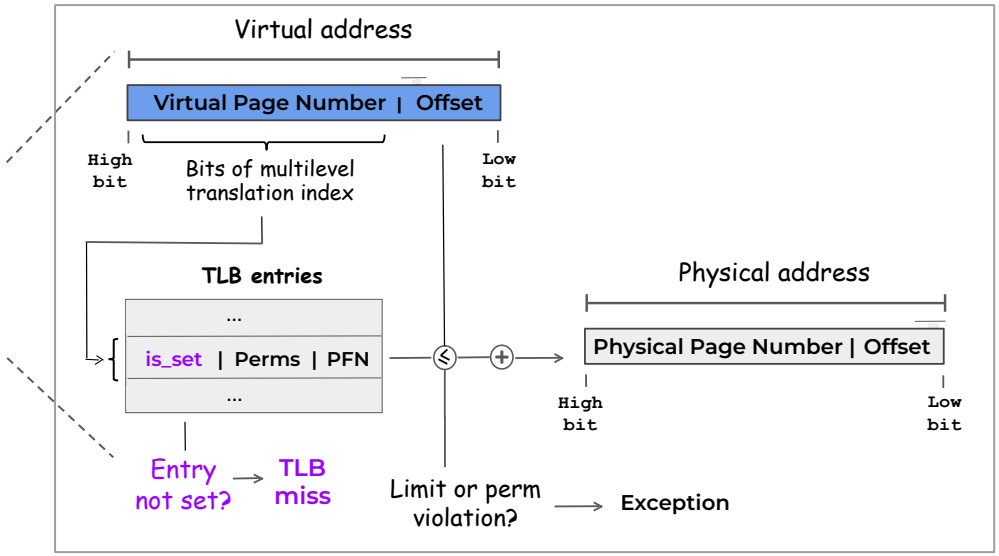
Q2: How are the entries of the index used?

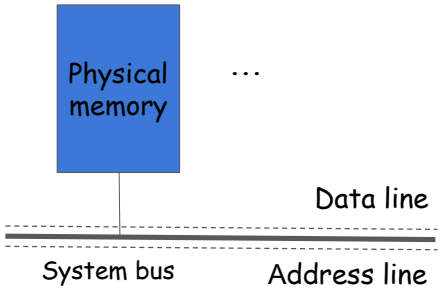
~~Q3: How are the entries of the index allocated?~~

Q4: How are the entries of the index replaced?

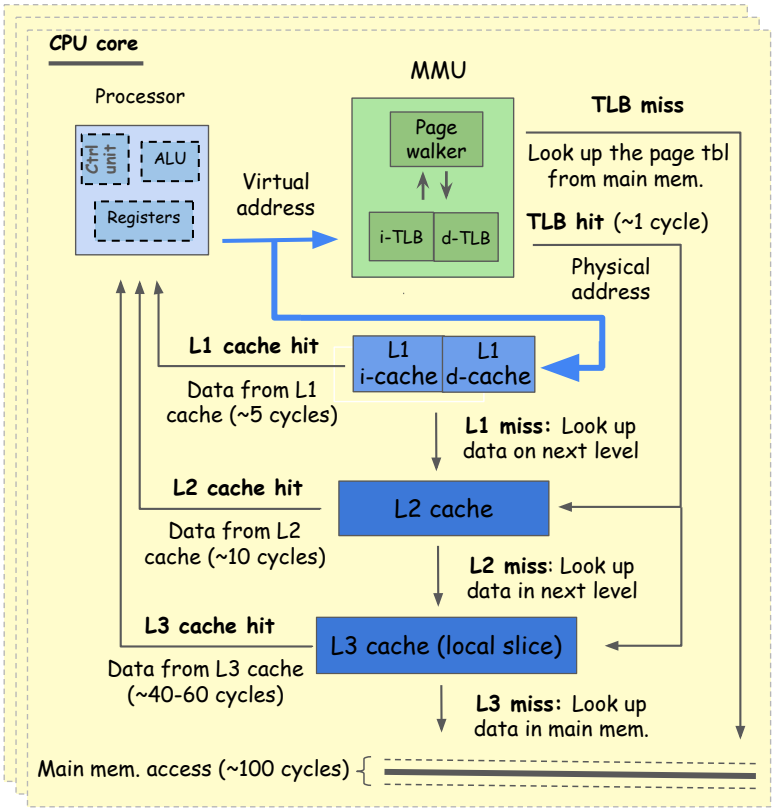
Translation Lookaside Buffer (TLB)

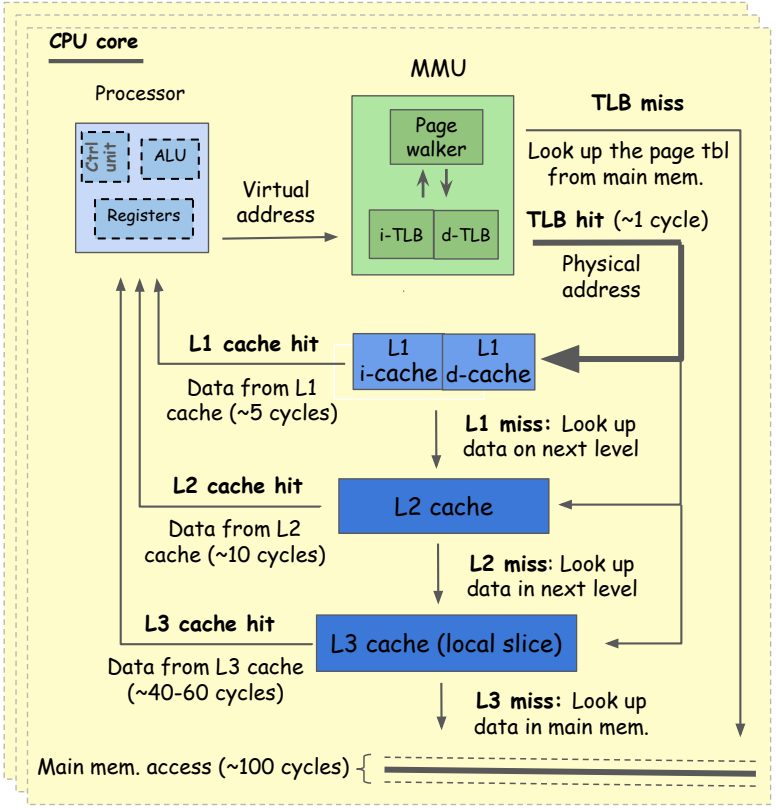




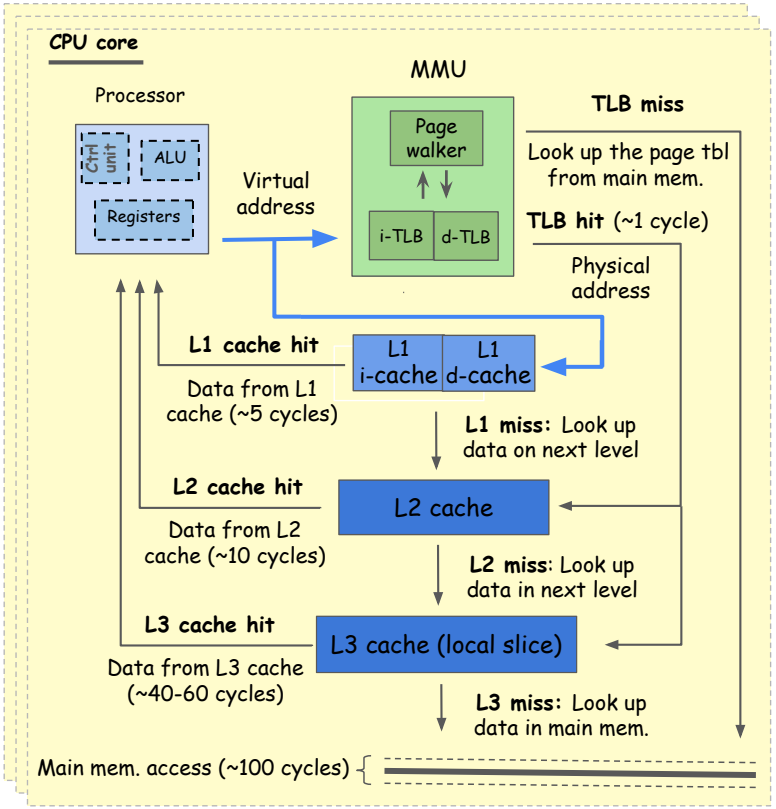


L1 cache: Why not virtually indexed and tagged?





L1 cache: Why not virtually indexed and tagged?



Homonym problem: After a context switch a proc can access leftover cache contents not mapped to its virtual address space (data leakage)

>> Flush caches on ctx switch? Slow

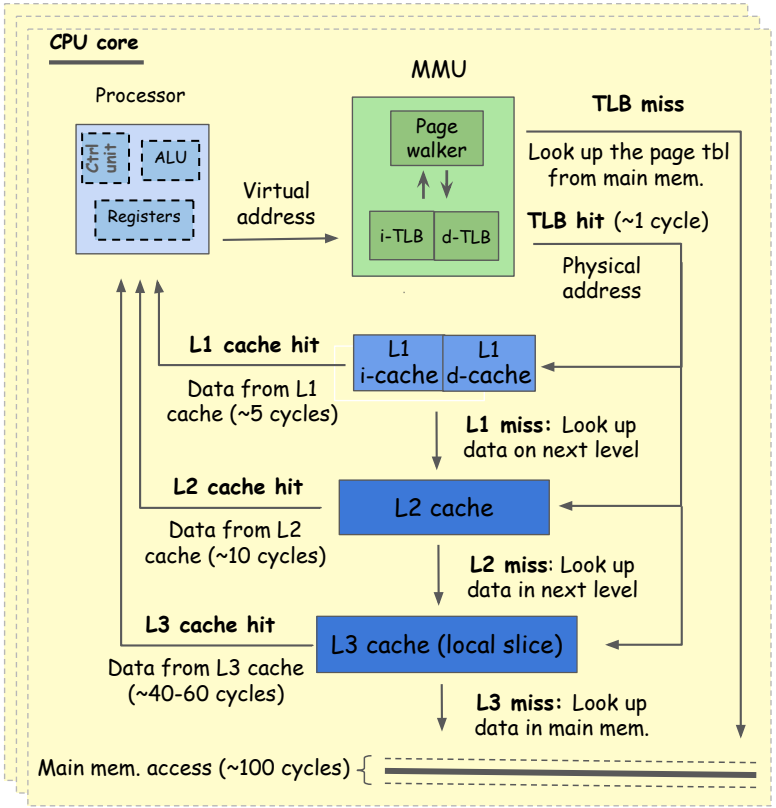
>> Use Addr. Space Ids in cache tags? Wasteful

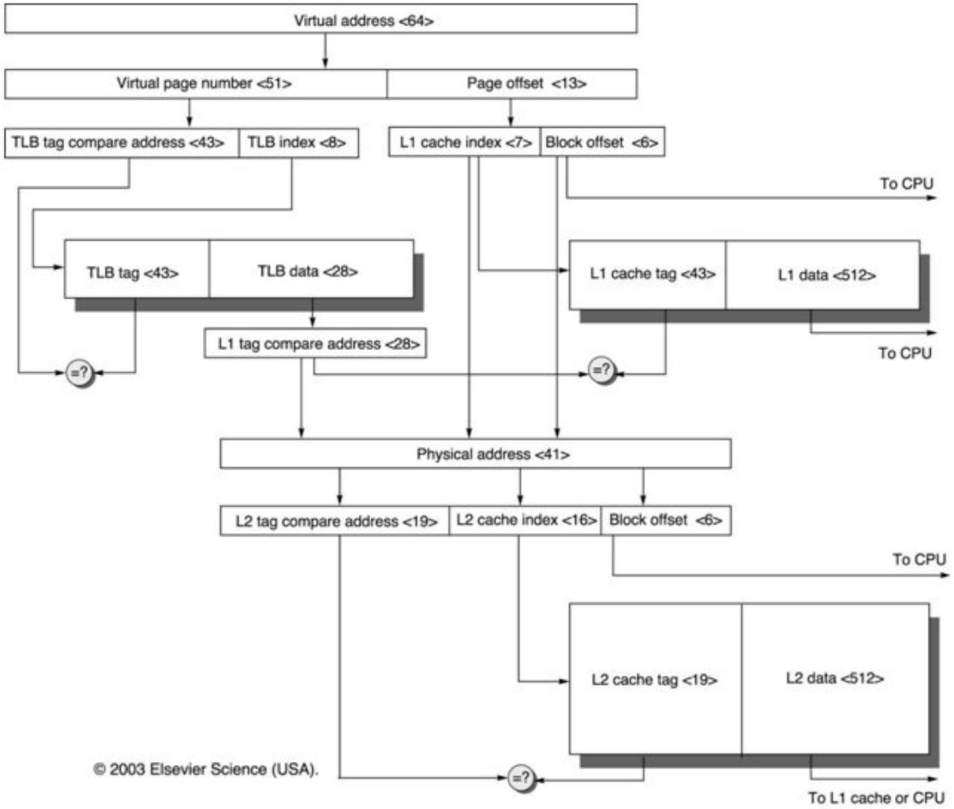
Synonym problem (a.k.a aliasing): Different virtual addresses reference the same data => OS decision => Invisible to hardware's cache coherency protocols...

>> Page coloring (beyond scope)

>> Use VIPT or PIPT caches

L1 cache: Virtually-indexed / Physically-tagged





How is virtual memory implemented?

- **Usually considered the most complex kernel subsystem**
- It's simple if you view it as an index

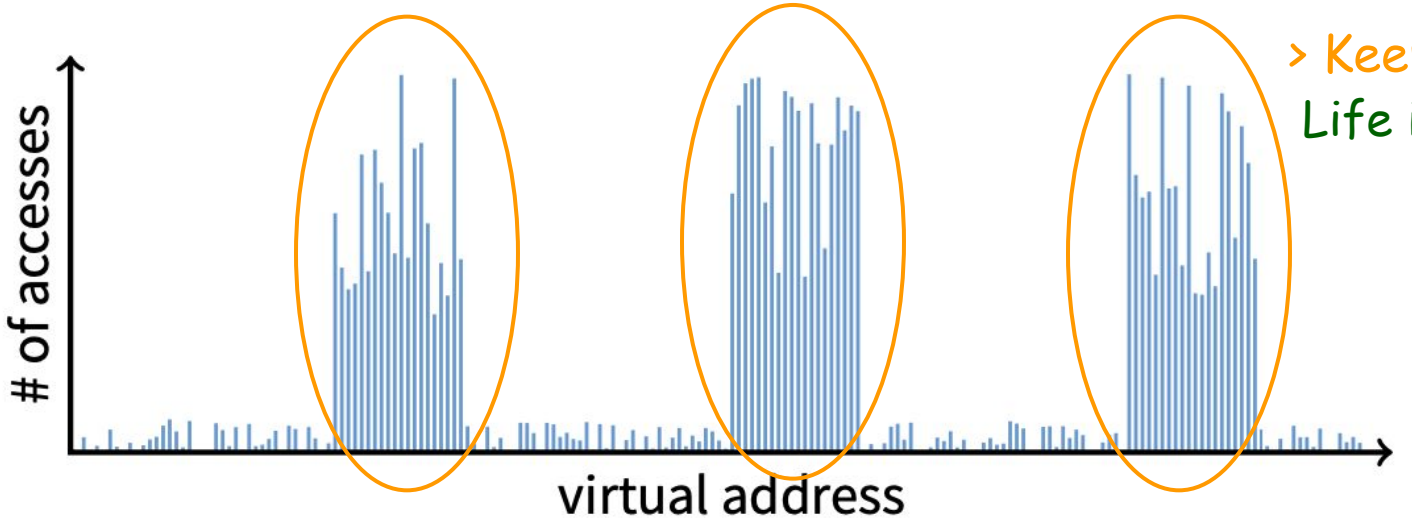
Ask the right questions

- ~~Q1: What are the entries of the index?~~
- ~~Q2: How are the entries of the index used?~~
- ~~**Q3: How are the entries of the index allocated?**~~
- ~~Q4: How are the entries of the index replaced?~~

The working set model for program behavior

> A process can be in main memory iff all the pages it is currently using can be in main memory, by P. Denning (1968)

Rule of thumb: 20% of memory gets 80% of accesses

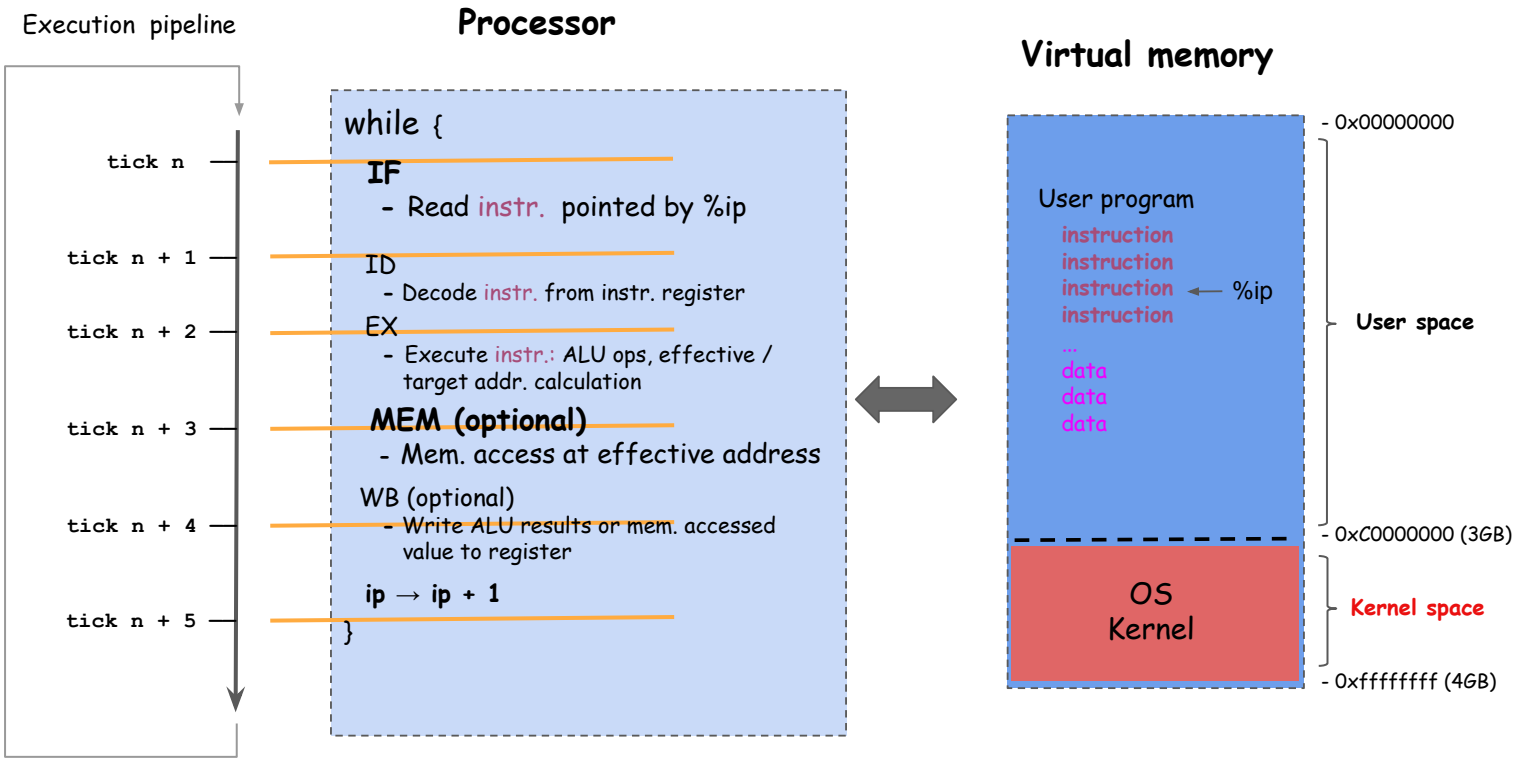


> Keep these "hot"?
Life is good...

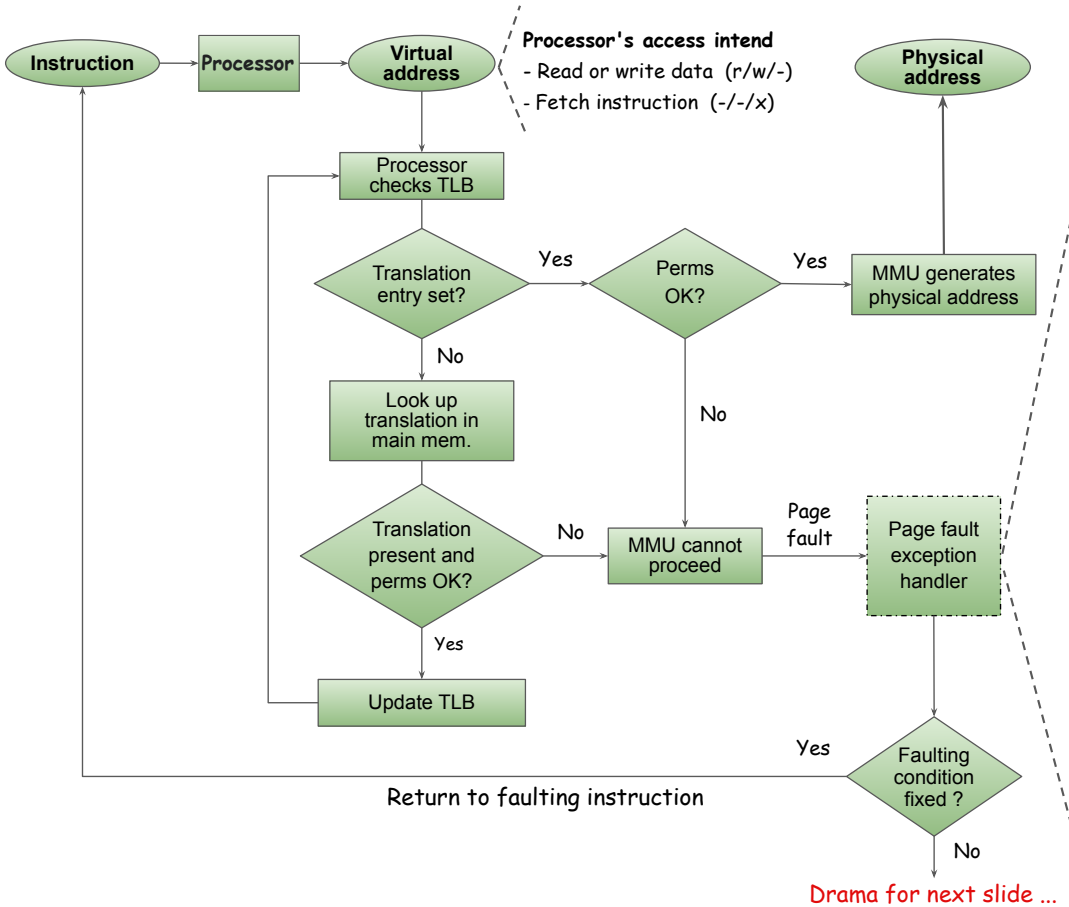
Demand paging via page fault handling

Demand paging via page fault handling

When may the processor access memory?



Demand paging via page fault handling



Page fault (p.f.) handling (see here and here)

> Access **not** permitted (see here) => SIGSEGV

>> Translation entry not set (see here)

>>> Not file-backed VMA (see here)

do_anonymous_page=> Mnr / Mjr p.f.

>>> File-backed VMA (see here)

r/? do_read_fault => Mnr / Mjr p.f.

w/shared? do_shared_fault => Mnr / Mjr p.f.

w/not-shared? do_cow_fault => Mnr / Mjr p.f.

>> Translation entry **set** (see here)

>>> "present" bit off (see here)

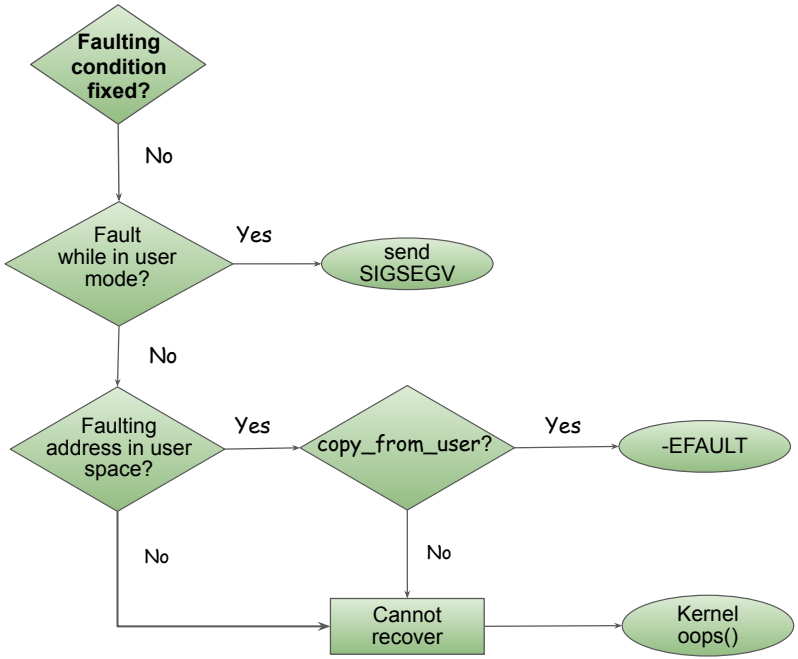
do_swap_page => Mjr p.f.

>>> "write" bit off (see here)

w/? do_wp_page => Mnr p.f.

* Major (Mjr) p.f.: Disk access required
* Minor (Mnr) p.f.: No Disk access required

Page fault handling (cont'ed)



Page fault demo: read->write

```c
int main(int argc, char **argv) {
  int vma_size = 2 * 4096; // size of 2 pages in Linux
  char a, *buffer = mmap(NULL, vma_size, PROT_READ | PROT_WRITE,
                         MAP_PRIVATE | MAP_ANONYMOUS,
                         -1, 0);
  for (int i = 0; i < vma_size; i += 4096) {
      start_time = clock_gettime_ns();
      a = buffer[i];
      end_time = clock_gettime_ns();
      printf("page-%d: Time elapsed: %lu ns (1st read)\n",
              i / 4096, end_time - start_time);
      start_time = clock_gettime_ns();
      a = buffer[i];
      end_time = clock_gettime_ns();
      printf("page-%d: Time elapsed: %lu ns (2nd read)\n",
              i / 4096, end_time - start_time);
  }
  for (int i = 0; i < vma_size; i += 4096) {
      start_time = clock_gettime_ns();
      buffer[i] = 'A';
      end_time = clock_gettime_ns();
      printf("page-%d: Time elapsed: %lu ns (1st write)\n",
              i / 4096, end_time - start_time);
      start_time =clock_gettime_ns();
      buffer[i] = 'A';
      end_time = clock_gettime_ns();
      printf("page-%d: Time elapsed: %lu ns (2nd write)\n",
              i / 4096, end_time - start_time);
  }
}
```

```
➜ git:(master) ✗ ./demo
(1) minor page faults: 87, major page faults: 0
page-0: Time elapsed: 2633 nanoseconds (1st read)
page-0: Time elapsed: 78 nanoseconds (2nd read)
page-1: Time elapsed: 1956 nanoseconds (1st read)
page-1: Time elapsed: 78 nanoseconds (2nd read)
(2) minor page faults: 89, major page faults: 0
page-0: Time elapsed: 4131 nanoseconds (1st write)
page-0: Time elapsed: 113 nanoseconds (2nd write)
page-1: Time elapsed: 3694 nanoseconds (1st write)
page-1: Time elapsed: 58 nanoseconds (2nd write)
(3) minor page faults: 91, major page faults: 0
```

Read pg faults

Write pg faults

Page fault demo: write->read

```c
int main(int argc, char **argv) {
  int vma_size = 2 * 4096; // size of 2 pages in Linux
  char a, *buffer = mmap(NULL, vma_size, PROT_READ | PROT_WRITE,
                         MAP_PRIVATE | MAP_ANONYMOUS,
                         -1, 0);
  for (int i = 0; i < vma_size; i += 4096) {
      start_time = clock_gettime_ns();
      buffer[i] = 'A';
      end_time = clock_gettime_ns();
      printf("page-%d: Time elapsed: %lu ns (1st write)\n",
             i / 4096, end_time - start_time);
      start_time = clock_gettime_ns();
      buffer[i] = 'A';
      end_time = clock_gettime_ns();
      printf("page-%d: Time elapsed: %lu ns (2nd write)\n",
             i / 4096, end_time - start_time);
  }
  for (int i = 0; i < vma_size; i += 4096) {
      start_time = clock_gettime_ns();
      a = buffer[i];
      end_time = clock_gettime_ns();
      printf("page-%d: Time elapsed: %lu ns (1st read)\n",
             i / 4096, end_time - start_time);
      start_time = clock_gettime_ns();
      a = buffer[i];
      end_time = clock_gettime_ns();
      printf("page-%d: Time elapsed: %lu ns (2nd read)\n",
             i / 4096, end_time - start_time);
  }
}
```

```
➜ git:(master) ✗ ./demo
(1) minor page faults: 87, major page faults: 0
page-0: Time elapsed: 2633 nanoseconds (1st read)
page-0: Time elapsed: 78 nanoseconds (2nd read)
page-1: Time elapsed: 1956 nanoseconds (1st read)
page-1: Time elapsed: 78 nanoseconds (2nd read)
(2) minor page faults: 89, major page faults: 0
page-0: Time elapsed: 4131 nanoseconds (1st write)
page-0: Time elapsed: 113 nanoseconds (2nd write)
page-1: Time elapsed: 3694 nanoseconds (1st write)
page-1: Time elapsed: 58 nanoseconds (2nd write)
(3) minor page faults: 91, major page faults: 0
➜ git:(master) ✗ ./write_read_page_faults
(1) minor page faults: 88, major page faults: 0
page-0: Time elapsed: 5868 nanoseconds (1st write)
page-0: Time elapsed: 115 nanoseconds (2nd write)
page-1: Time elapsed: 5487 nanoseconds (1st write)
page-1: Time elapsed: 48 nanoseconds (2nd write)
(2) minor page faults: 90, major page faults: 0
page-0: Time elapsed: 90 nanoseconds (1st read)
page-0: Time elapsed: 92 nanoseconds (2nd read)
page-1: Time elapsed: 59 nanoseconds (1st read)
page-1: Time elapsed: 46 nanoseconds (2nd read)
(3) minor page faults: 90, major page faults: 0
```

Read pg faults

Write pg faults

Write pg faults

Read pg faults

Page fault demo: strcpy

```c
int main(int argc, char **argv) {
  // mmap one page (4096 in Linux)
  char *buffer = mmap(NULL, 4096,
                      PROT_READ | PROT_WRITE,
                      MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

  start_time = clock_gettime_ns();
  strcpy(buffer, "Hello, mmap!");
  end_time = clock_gettime_ns();
  printf("1. Time elapsed: %lu ns\n", end_time - start_time);

  start_time = clock_gettime_ns();
  strcpy(buffer, "Hello, mmap!");
  end_time = clock_gettime_ns();
  printf("2. Time elapsed: %lu ns\n", end_time - start_time);

  if (mprotect(buffer, page_size, PROT_READ)) {
    perror("mproctect");
    return 1;
  }
  buffer[0] = 'C';
}
```

```
➜ git:(master) ✗ ./demo
1. Time elapsed: 3083 nanoseconds
2. Time elapsed: 167 nanoseconds
[1]    80193 segmentation fault  ./test
```

Calculating paze size by observing page fault latency

```c
#define NUM_PAGES 2
#define STEP_SIZE 512
void foo(char *ptr, int end_byte) {
   start_time = clock_gettime_ns();
   *ptr = 'A';
   end_time = clock_gettime_ns();
   printf("%d; Time elapsed: %lu ns\n", end_byte,
          (end_time - start_time));
}
int main(int argc, char **argv) {
   long page_size = sysconf(_SC_PAGE_SIZE);
   char *buf = mmap(NULL, NUM_PAGES*page_size,
                    PROT_READ | PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS,  -1, 0);
   int end_byte = STEP_SIZE;
   while (end_byte < NUM_PAGES * page_size) {
        foo(buf+end_byte, end_byte);
      end_byte += STEP_SIZE;
   }
   munmap(buf, NUM_PAGES*page_size);
   return 0;
}
```

```
➜ git:(master) ✗ ./demo
foo: 0; time elapsed: 2250 nanoseconds
foo: 512; time elapsed: 125 nanoseconds
foo: 1024; time elapsed: 125 nanoseconds
foo: 1536; time elapsed: 125 nanoseconds
foo: 2048; time elapsed: 125 nanoseconds
foo: 2560; time elapsed: 167 nanoseconds
foo: 3072; time elapsed: 167 nanoseconds
foo: 3584; time elapsed: 125 nanoseconds
foo: 4096; time elapsed: 3625 nanoseconds
foo: 4608; time elapsed: 125 nanoseconds
foo: 5120; time elapsed: 167 nanoseconds
foo: 5632; time elapsed: 83 nanoseconds
foo: 6144; time elapsed: 83 nanoseconds
foo: 6656; time elapsed: 125 nanoseconds
foo: 7168; time elapsed: 125 nanoseconds
foo: 7680; time elapsed: 334 nanoseconds
```

Calculating major vs minor page fault latency

```c
int main(int argc, char **argv) {
    // Assuming file already exists at path
    if ( (fd = open("/tmp/foo.txt", O_RDONLY, 0664)) < 0)
        return -1;
   char *buf = mmap(NULL, page_size, PROT_READ, MAP_PRIVATE, fd, 0);
   start_time = clock_gettime_ns();
   char a = buf[0];
   end_time = clock_gettime_ns();
   printf("time elapsed: %lu ns (1st read) \n", end_time -  start_time);
   start_time = clock_gettime_ns();
   char a = buf[0];
   end_time = clock_gettime_ns();
   printf("time elapsed: %lu ns (2nd read) \n", end_time -  start_time);
}
```

```
# 1 -> Drop in-kernel page caches
➜  git:(master) ✗  echo 1 | sudo tee /proc/sys/vm/drop_caches
1
➜  git:(master) ✗  ./demo
Time elapsed: 448834 ns (1-st read)   <-- Major page fault
Time elapsed: 41 ns (2-nd read)
➜  git:(master) ✗  ./demo
Time elapsed: 11041 ns (1-st read) <-- Minor page fault
Time elapsed: 42 ns (2-nd read)
```

How is virtual memory implemented?

> Usually considered the most complex kernel subsystem

- It's simple if you view it as an index

Ask the right questions

~~Q1: What are the entries of the index?~~

~~Q2: How are the entries of the index used?~~

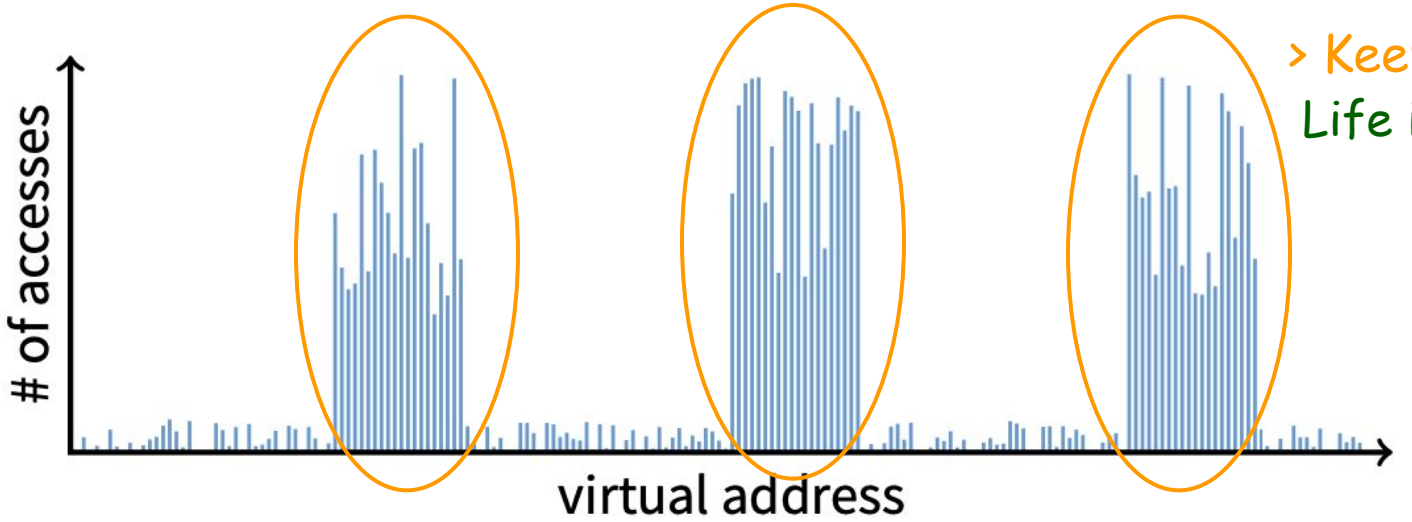
~~Q3: How are the entries of the index allocated?~~

Q4: How are the entries of the index replaced?

The working set model for program behavior

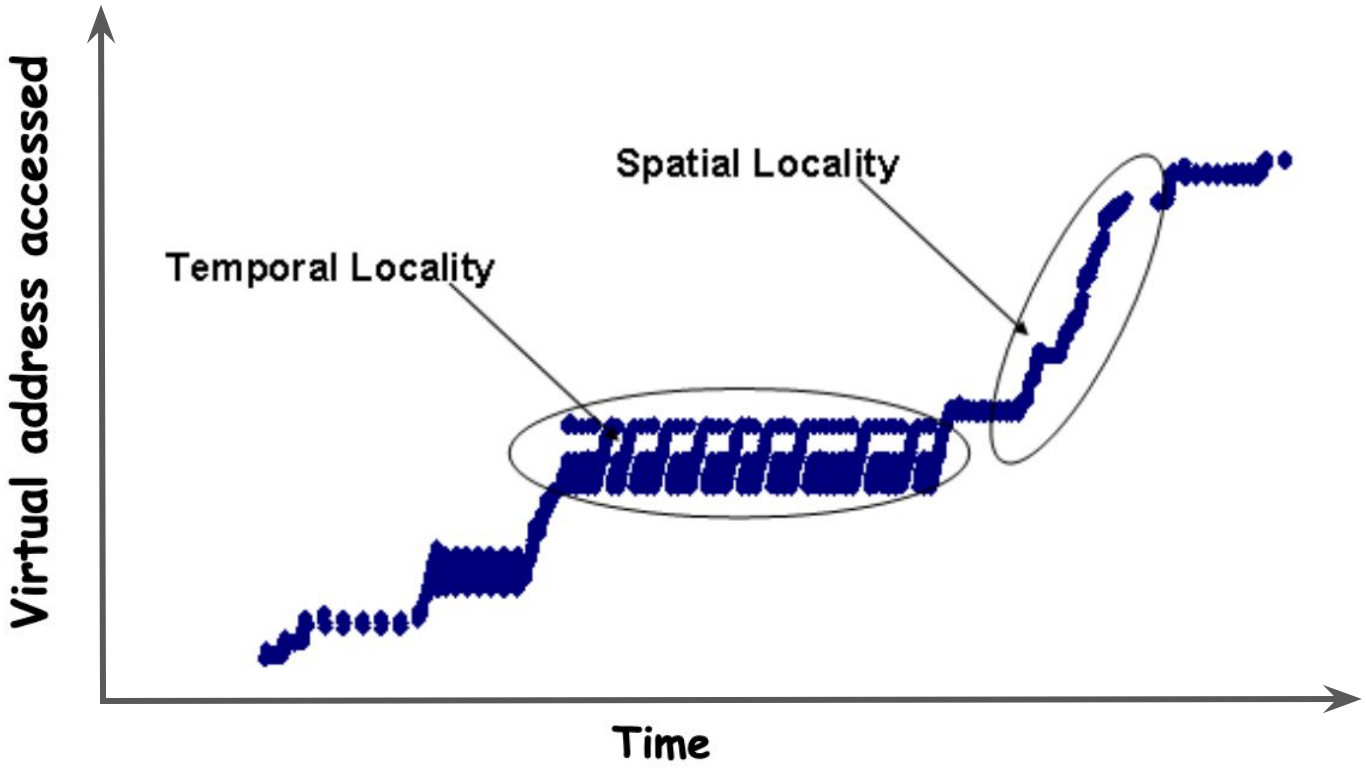
> A process can be in main memory iff all the pages that it is currently using can be in main memory, by P. Denning (1968)

Rule of thumb: 20% of memory gets 80% of accesses



> Keep these "hot"?
Life is good...

Temporal and spatial locality



Replacement policies

> When all physical memory is in use, the OS must reclaim some physical page frames in order to serve new translations

Which translations are "good candidates" to be evicted?

> Converse of locality property: The Last Recently Used (LRU) page is the least to be used soon ("best eviction candidate")

> Approximate LRU (a.k.a. CLOCK algorithm)

- LRU is expensive to implement in software => Hardware support?
- Remember PTE "referenced" bit? The OS periodically swipes through PTEs and resets "accessed" bits
- Translation entries w/ "accessed" bits off

Replacement policies

> When all physical memory is in use, the OS must reclaim some physical page frames in order to serve new translations

Which translations are "good candidates" to be evicted?

> Converse of locality property: The Last Recently Used (LRU) page is the least to be used soon ("best eviction candidate")

> Approximate LRU (a.k.a. CLOCK algorithm)

- LRU is expensive to implement in software => Hardware support?
- Remember PTE "referenced" bit? The OS periodically swipes through PTEs and resets "accessed" bits
- Translation entries w/ "accessed" bits off => Not recently accessed

Replacement policies

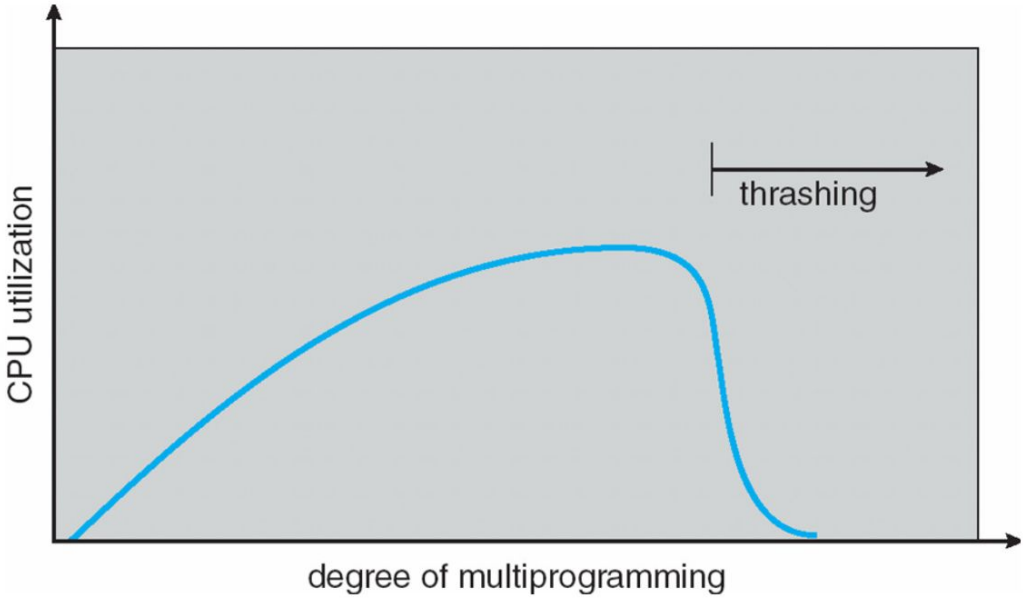
› Not all pages are equal: Reclaiming decision is made based on the the type of the occupied pages

- File-backed pages: Dropped from the kernel's page cache **first** since they can be reloaded later from the corresponding file
- Anonymous pages w/o file backing: Must be written to the swap space in order to be available for reloading; **dropped next**
- File-system-related kernel caches: **Dropped last** (next chapter)
- Last resort: Out-Of-Memory (OOM) killer terminates the process

Thrashing (or, "when nothing works")

> I/O devices at 100% utilization but CPU utilization drops to 0%

- Processes remain blocked waiting for pages to be fetched in
- The system does no useful work



Thrashing (or, "when nothing works")

> Reasons for thrashing

- Memory accesses w/o temporal locality => Past /= Future
- Processes' hot memory does not fit in maim memory

> What we ordered? Memory with the size of disk and the speed of CPU caches

> What we got? Memory with the access time of the disk :-(

POSIX Memory management syscalls (cont'ed)

void *mmap (void *addr, ...)

> Creates a new mapping in the virtual address space of the calling process

- addr: If NULL, the kernel chooses the address at which to create the mapping
- On success, mmap(...) returns a pointer to the mapped area

int munmap (void *addr, size_t length)

> Deletes the mappings for the specified address range

- addr: Start of the address range
- length: Size of the address range
- On success, munmap(...) returns 0

int mprotect (void *addr, size_t len, int prot)

> Updates the protections for page(s) in range [addr, addr+len) to "prot"

- addr: Must be aligned to a page boundary
- On success, mprotect(...) returns 0

POSIX Memory management syscalls (cont'ed)

int msync (void *addr, ...):

> Flushes to disk changes made to the in-kernel copy of a file that was mapped into memory using mmap

- Note: Without use of msync(...) there is no guarantee that changes made to the in-kernel copy of a file will be written back to disk before munmap(...) is called

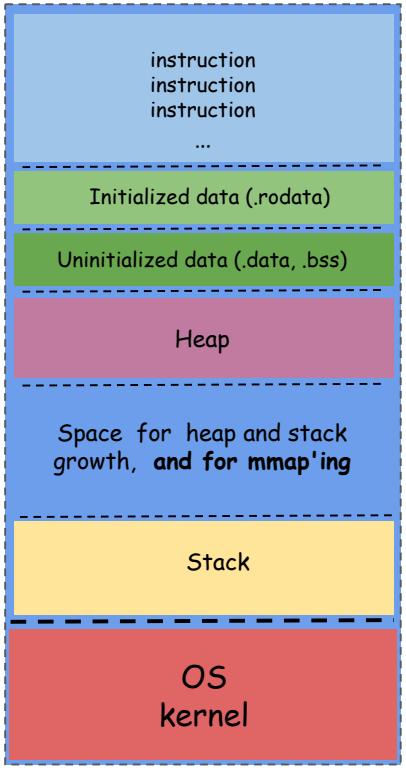
- On success, msync(...) returns 0

void *mlock/unlock (void *addr, ...)

> Locks/unlocks part or all of the calling process's virtual address space into main mem. preventing that mem. from being paged out to the swap area

- On success, mlock(...)/unlock(...) returns 0

Putting it all together: (1/3) Per-process index of VAS segments

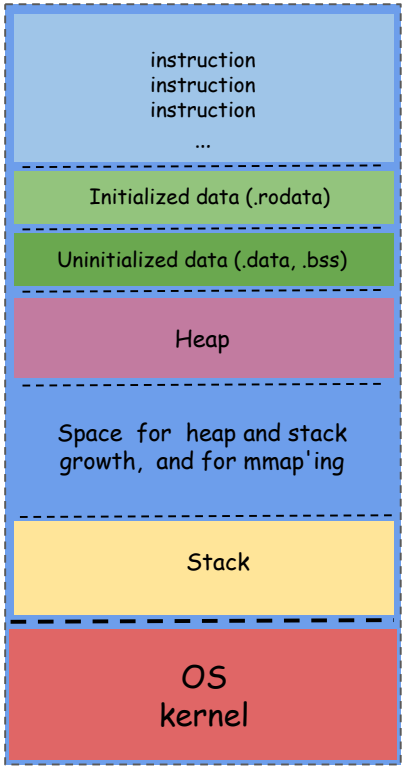



```
struct task_struct {
    ...
    char comm[TASK_COMM_LEN];
    struct mm_struct *active_mm;
    struct mm_struct *mm;
    struct mm_struct {
        ...
        pgd_t *pgd;
        unsigned long start_code, end_code;
        unsigned long start_data, end_data;
        unsigned long start_brk, brk;
        unsigned long start_stack;
        struct maple_tree mm_mt;
        struct vm_area_struct {
            unsigned long vm_start;
            unsigned long vm_end;
            pgprot_t vm_page_prot;
            vm_flags_t vm_flags;
            ...
        }
```

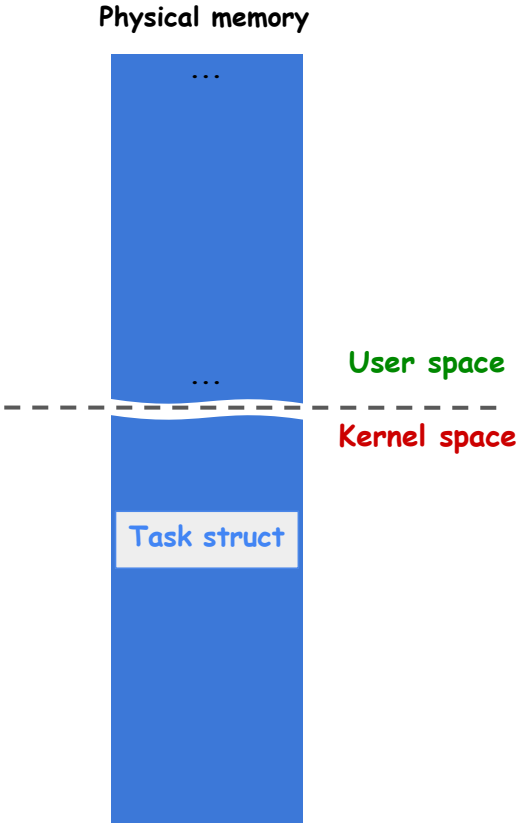
› Mapple tree: Read here

```
➜ git:(master) ✗  cat /proc/12453/maps
aaaadcda0000-aaaadcda1000 r-xp  ... /foo
aaaadcdb0000-aaaadcdb1000 r--p  ... /foo
aaaadcdb1000-aaaadcdb2000 rw-p ... /foo
...
aaaaed20d000-aaaaed22e000 rw-p ... [heap]
ffffbe610000-ffffbe798000 r-xp ... libc.so
ffffbe798000-ffffbe7a7000 ---p ... libc.so
...
fffff6461000-fffff6482000 rw-p ... [stack]
```

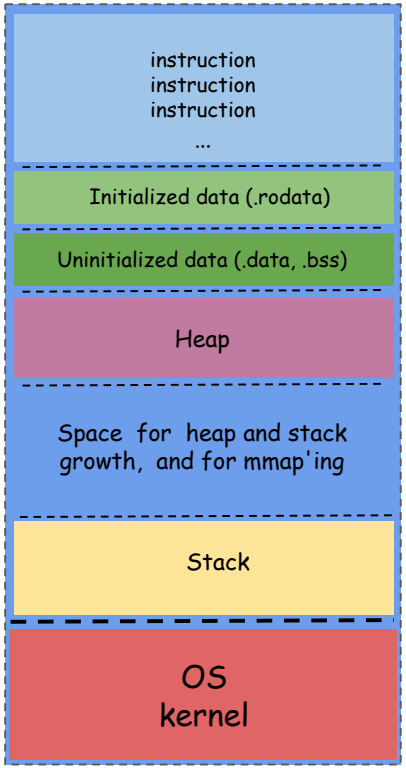
Putting it all together: (1/3) Per-process index of VAS segments

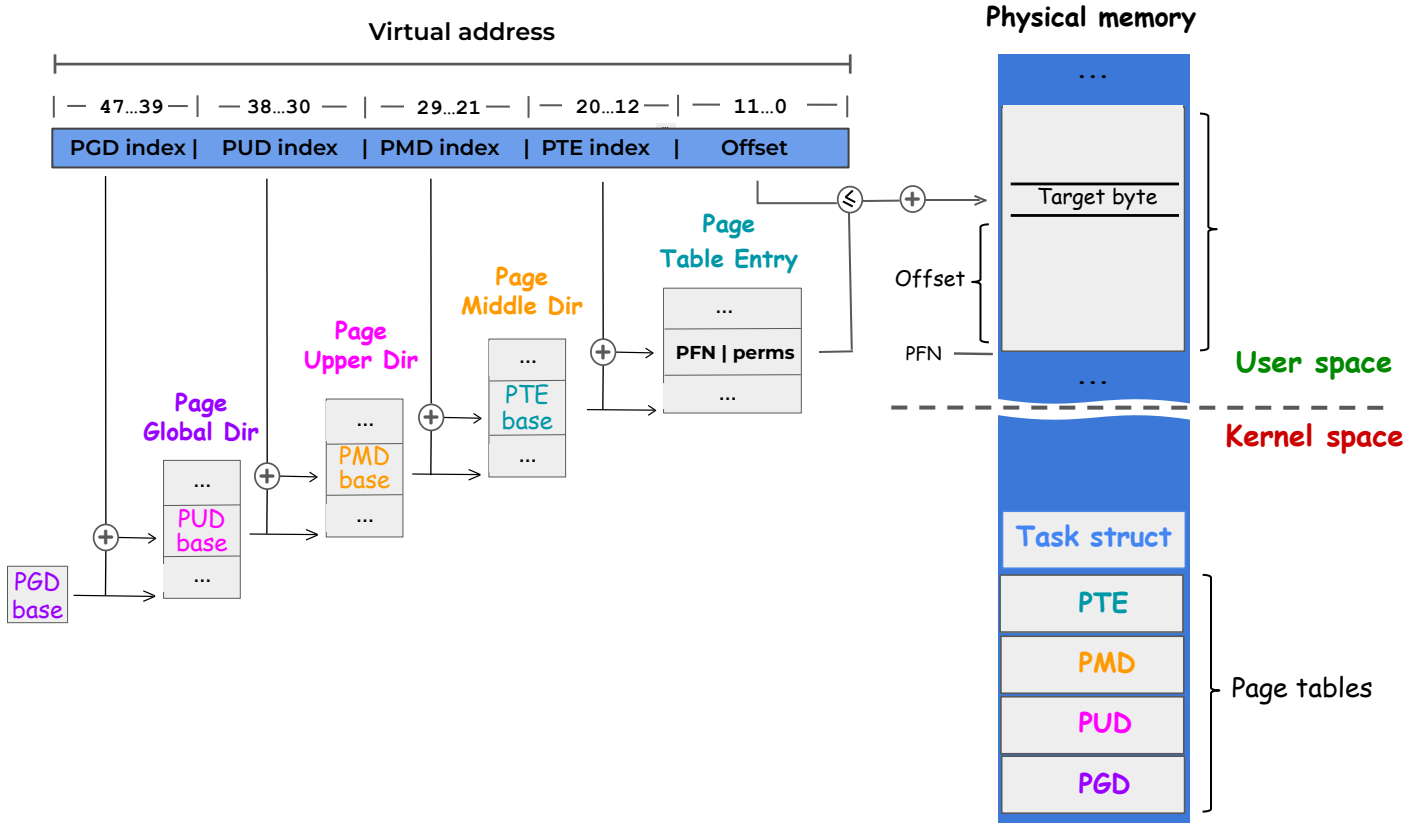



```
struct task_struct {
    ...
    char comm[TASK_COMM_LEN];
    struct mm_struct *active_mm;
    struct mm_struct *mm;
    struct mm_struct {
        ...
        pgd_t *pgd;
        unsigned long start_code, end_code;
        unsigned long start_data, end_data;
        unsigned long start_brk, brk;
        unsigned long start_stack;
        struct maple_tree mm_mt;
        struct vm_area_struct {
            unsigned long vm_start;
            unsigned long vm_end;
            pgprot_t vm_page_prot;
            vm_flags_t vm_flags;
            ...
    }
```

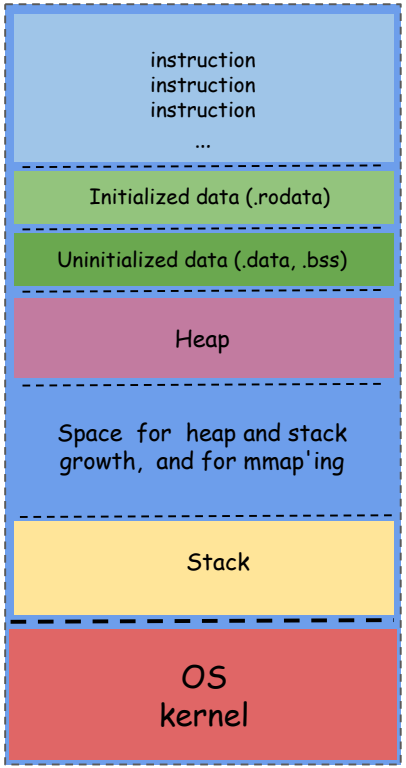


Putting it all together: (2/3) Per-process page table

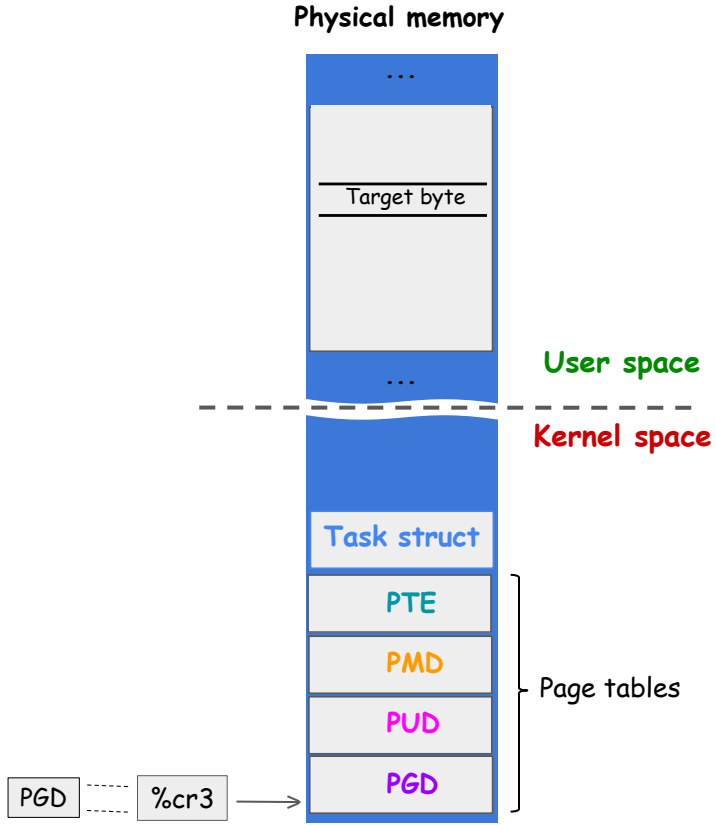




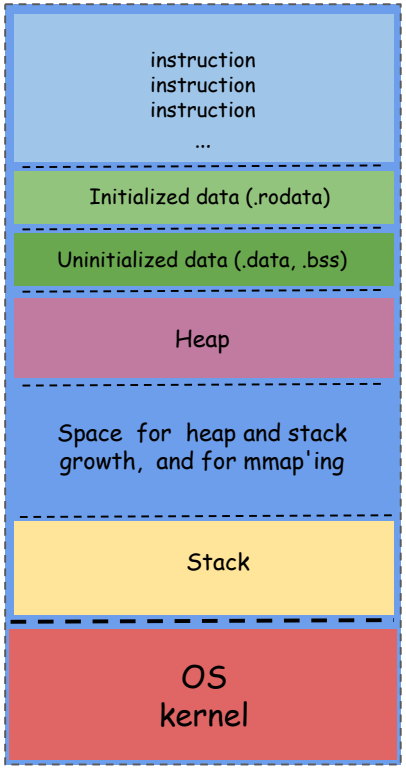
Putting it all together: (2/3) Per-process page table




```
struct task_struct {
    ...
    char comm[TASK_COMM_LEN];
    struct mm_struct *active_mm;
    struct mm_struct *mm;
    struct mm_struct {
        ...
        pgd_t *pgd;
        unsigned long start_code, end_code;
        unsigned long start_data, end_data;
        unsigned long start_brk, brk;
        unsigned long start_stack;
        struct maple_tree mm_mt;
        struct vm_area_struct {
            unsigned long vm_start;
            unsigned long vm_end;
            pgprot_t vm_page_prot;
            vm_flags_t vm_flags;
            ...
        }
```



Putting it all together: (3/3) System-wide page frame allocators



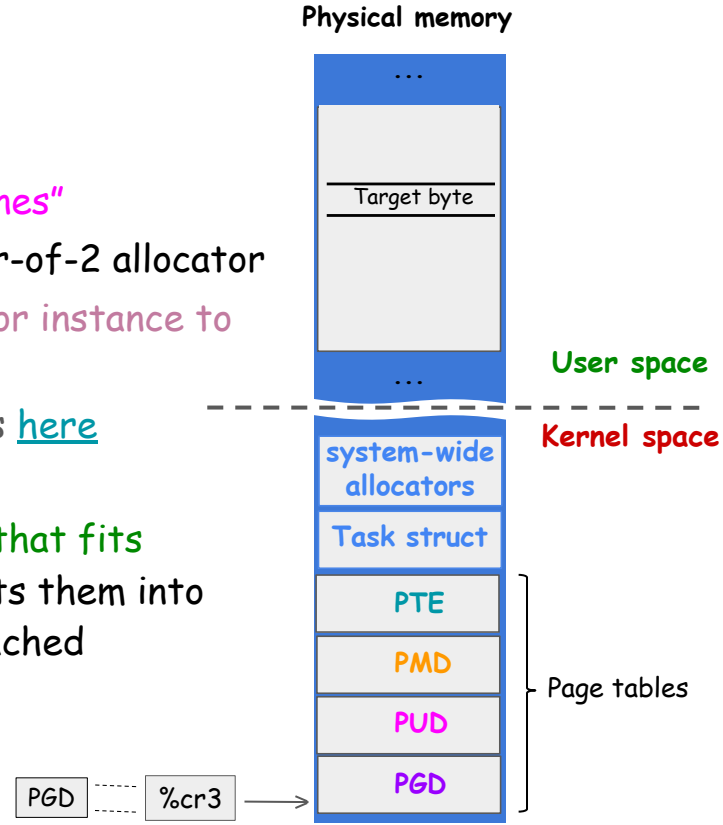
> Buddy memory allocator

>> Physical memory is divided into "zones"

- Buddy allocator: Range-based, power-of-2 allocator
- Each zone has its own buddy allocator instance to manage free physical pages
- Core of the zoned buddy allocator is here

>> Request comes in

- Allocator finds the smallest block that fits
- Only larger blocks available? Splits them into "buddies" until the right size is reached



Intercepting mem. accesses w/ mprotect and signals

```
static void handler(int sig, siginfo_t *si, void *unused) {
    printf("Got SIGSEGV\n");
    printf("Someone touched address: 0x%lx\n");
     exit(1); // Remove this line and see what happens...
}

int main(int argc, char **argv) {
   struct sigaction sa;
   sa.sa_flags = SA_SIGINFO;
   sa.sa_sigaction = handler;
   sigemptyset(&sa.sa_mask);
   sigaction(SIGSEGV, &sa, NULL);
   char *buf = mmap(NULL, 4096,  PROT_READ | PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS,  -1, 0);
   printf("1. Going to touch\n");
   buf[123] = 'a';
   if ( mprotect(buf, 4096, PROT_NONE)) {
      perror("mprotect");
      return 1;
   }
   show_self_proc_maps();
   printf("2. Going to touch\n");
   buf[123] = 'a';
}
```

```
➜  git:(master) ✗  ./test
1. Going to touch
aaaab6840000-aaaab6842000 r-xp
aaaab6851000-aaaab6852000 r--p
aaaab6852000-aaaab6853000 rw-p
aaaaeafda000-aaaaeaffb000   rw-p   [heap]
ffff88510000-ffff88698000 r-xp    libc.so.6
ffff88698000-ffff886a7000 ---p    libc.so.6
ffff886a7000-ffff886ab000 r--p    libc.so.6
ffff886ab000-ffff886ad000 rw-p    libc.so.6
ffff886ad000-ffff886b9000 rw-p
ffff886da000-ffff88705000 r-xp    ld-linux-aarch64.so.1
ffff8870e000-ffff8870f000 ---p
ffff8870f000-ffff88711000 rw-p
ffff88711000-ffff88713000 r--p    [vvar]
ffff88713000-ffff88714000 r-xp    [vdso]
ffff88714000-ffff88716000 r--p    ld-linux-aarch64.so.1
ffff88716000-ffff88718000 rw-p    ld-linux-aarch64.so.1
fffff5a11000-fffff5a32000  rw-p   [stack]
2. Going to touch
Got SIGSEGV
Someone touched address: 0xffff8870e07b
```

Tracking PTE invalidations with mprotect

```
void foo(char *buf) {
   start_time = clock_gettime_ns();
   *buf = 'a';
   end_time = clock_gettime_ns();
   printf("time elapsed: %lu nanoseconds \n", end_time -  start_time);
}
int main(int argc, char **argv) {
   char *buf = mmap(NULL, page_size, PROT_READ | PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
   for (int i = 0; i < page_size; i += page_size/4)
       foo(buf+i);

   char *buf = mmap(NULL, page_size, PROT_READ | PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
   for (int i = 0; i < page_size; i+=page_size/4) {
       foo(buf+i);
      mprotect(buf, page_size, PROT_WRITE | PROT_READ );
   }
   char *buf = mmap(NULL, page_size, PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
   for (int i = 0; i < page_size; i+=page_size/4) {
      foo(buf+i);
      mprotect(buf, page_size, PROT_WRITE | PROT_READ );
   }
   char *buf = mmap(NULL, page_size, PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
   for (int i = 0; i < page_size; i+=page_size/4) {
      foo(buf+i);
      mprotect(buf, page_size, PROT_READ );
      mprotect(buf, page_size, PROT_WRITE | PROT_READ );
   }}
```

```
➜ git:(master) ✗  ./test
time elapsed: 1417 nanoseconds
time elapsed: 125 nanoseconds
time elapsed: 334 nanoseconds
time elapsed: 125 nanoseconds
--------------------------------

time elapsed: 3084 nanoseconds
time elapsed: 42 nanoseconds
time elapsed: 125 nanoseconds
time elapsed: 84 nanoseconds
--------------------------------

time elapsed: 2334 nanoseconds
time elapsed: 3459 nanoseconds
time elapsed: 125 nanoseconds
time elapsed: 125 nanoseconds
--------------------------------

time elapsed: 1459 nanoseconds
time elapsed: 2000 nanoseconds
time elapsed: 2125 nanoseconds
time elapsed: 5167 nanoseconds
--------------------------------
```

Measuring row- vs column-major mem. access time

```
int main(int argc, char **argv) {
    int *buf = malloc(10000*sizeof(int *));
    for (int i = 0; i < 1000; i++)
        buf[i] = malloc(10000*sizeof(int));

    for (int i = 0; i < 1000; i++)
      for (int j = 0; i < 1000; j++)
          buf[i][j] = i + j;

    start = clock_gettime_ns();
    for (int i = 0; i < 1000; i++)
      for (int j = 0; i < 1000; j++)
          buf[i][j] = i + j;  // access row-major
    end = clock_gettime_ns();
    printf("%.1f ns per access\n", (end - start) / e8 );
    start_time = clock_gettime_ns();
    for (int i = 0; i < 1000; i++)
      for (int j = 0; i < 1000; j++)
          buf[j][i] = i + j;  // access column-major
    end_time = clock_gettime_ns();
    printf("%.1f ns per access\n", (end - start) / e8 );
```

```
➜ git:(master) ✗ ./test
0.7 nanoseconds per write access
4.5 nanoseconds per write access
```

What is a file?

"An object that can be written to, or read from, or both, with data and attributes, such as access perms and type." (POSIX def. 3/139.)

> A named byte-array persistent across reboots

- **Regular files:** Contain user data in text or binary format
- **Special files:** Devices for char-by-char (e.g., /dev/tty) or block-based (e.g., /dev/sda) data transfers
- **Named Pipes:** First-in-first-out IPC mechaninsm
- **Sockets:** Endpoint for network communication or IPC
- **Directories:** Contains a list of file names
- **Symbolic links:** A pointer or shortcut to another file or directory

Why need a file?

"An object that can be written to, or read from, or both, with data and attributes, such as access perms and type." (POSIX definition 3/139.)

> A named byte-array **persistent** **across reboots**

- Helps identify data by using natural language names
- Abstracts the details of the underlying storage devices
- First and only persistent abstraction
  - >> Persists across reboots
  - >> Persists across power failures
  - >> Storage devices healthy and filled w/ electricity? Life is good

Necessary file metadata?

```
➜  git:(master) ✗ /bin/ls -hlia ept.patch
32078924 -rw-r--r--  1 parallels parallels 18K Dec 30 23:28 ept.patch
```

- File Identifier: Identifies file within file system (inode)
- Access Control List (ACL): Controls users and allowed accesses
- Owner and Group: Used along with ACLs for permission checking
- Size: File size in bytes, KiB, and so on
- Timestamp: Time of last modification
- Filename: The name of the file, in human-readable format

What is a directory?

"A file that contains directory entries; that is, objects that associate filenames with a files" (POSIX definition 3/103.)

> Conceptually: A hierarchical organization technique, based on an acyclic-graph hierarchy: e.g., A/B, implies that file or directory B, lives under its parent directory A.

> Technically: each directory is a file whose data is a list of <*filename*, *index*> pairs.

> Root "/" directory: Special directory, root of the hierarchy

File-related POSIX syscalls

int open (const char *pathname, int flags, ...)

› Given a file pathname, open() returns a non-negative process-unique inheritable open file handle integer (called a **file descriptor**), for use in subsequent syscalls.

- pathname: The name identifying the target file
- flags: Must include one of the following access modes O_RD/WRONLY or O_RDWR.
- On success, open(...) returns a non-negative integer; or, -1 is returned, if an error occurred

What is a POSIX file descriptor?

"A per-process unique, non-negative integer used to identify an open file for the purpose of file access. The values 0, 1, and 2 are referred to as standard input, standard output, and standard error." (POSIX def 3/141.)

Expensive to resolve name to identifier on each access

› **Elegant POSIX solution:** Open file before access

Brief implementation details (more later..)

1. Search directories for file name, locate and check permission
2. Read file metadata into a system-wide in-memory open files table
3. In-process integer, called file descriptor (fd), indexes the open files table
4. Processes reuses fd by passing it to the OS for subsequent file access
5. Process needs to access a new file? Will add a new integer to its fd table

File-related POSIX syscalls

int open (const char *pathname, int flags, ...)

› Given a file pathname, open() returns a non-negative process-unique inheritable open file handle integer (called a **file descriptor**), for use in subsequent syscalls.

- pathname: The name identifying the target file
- flags: Must include one of the following access modes O_RD/WRONLY or O_RDWR.
- On success, open(...) returns a non-negative integer; or, -1 is returned, if an error occurred

int rename (const char *oldpath, const char *newpath)

› Renames a file, potentially moving it between directories if required. Any other hard links to the file as well as "oldpath"-related open fds are unaffected.

- oldpath: Origin path
- newpath: Destination path
- On success, rename(...) returns zero; or, -1 is returned, if an error occurred

File-related POSIX syscalls

int unlink (const char *pathname)

> Deletes a name from the filesystem and possibly the file it refers to. If "pathname" is the last link to a file and no process has the file open, the file is deleted and the space it was using is made available for reuse.

- pathname: The name identifying the target file
- On success, unlink(...) returns zero; or, -1 is returned, if an error occurred

int truncate (const char *path, off_t length)

> Cause the regular file named by path to be resized to precisely length bytes, such that if the file previously was larger, the extra data; or, it it was previously shorter, it is extended, and the extended part reads as null bytes

- pathname: The name identifying the target file
- length: The target, new length
- On success, truncate(...) returns zero; or, -1 is returned, if an error occurred

File-related POSIX syscalls

int read (int fd, int *buf, size_t count)

› Attempts to read up to *count* bytes from the file descriptor *fd* into *buf*.

- On success, read(...) returns the number of bytes read and the file position is advanced accordingly. Zero indicates end of file, while, it is not an error, if this number is smaller than count. On error, -1 is returned, and errno is set appropriately.

int write (int fd, int *buf, size_t count)

› Attempts to write up to *count* bytes from the buffer pointed *buf* to the file referred to by the file descriptor *fd*.

- On success, write(...) returns the number of bytes written, while, zero indicates that nothing was written. On error, -1 is returned, and errno is set appropriately.

File-related POSIX syscalls

off_t lseek (int fd, off_t offset, int whence)

> Given an open *fd*, lseek repositions the respective's file *offset* according to the directive *whence* to be at position (i) "offset" (SEEK_SET); (ii) "current position" + "offset" (SEEK_CUR); or, (iii) "size of file" + "offset" (SEEK_END).

- On success, lseek() returns the resulting offset location as measured in bytes from the beginning of the file; or, -1 is returned, if an error occurred

int fsync (int fd)

> Transfers, i.e., "flushes", all modified in-kernel data and metadata of the file associated with fd to the underlying storage devive. fsync(...) blocks until the device reports that the transfer has completed.

- On success, fsync(...) returns zero; or, -1 is returned, if an error occurred

Crash-tolerant file updates

> Typical goal when dealing with files: How to safely update a file, even given the potential for a crashes or power failure to occur?

Crash-tolerant file update pattern

1. write: data -> temp_file
2. fsync: temp_file
3. rename: temp_file -> target_file [rename is an atomic operation]
4. fsync: parent_dir
5. Assert temp_file does not exist

File accesses patterns

› Sequential Access
- Data read from or written to storage in order
- Good temporal locality => Can be efficiently proactive with prefetching
- Examples: User copying files, Compiler reading / writing files

› Random Access
- Randomly accessing any block
- Poor spatial Locality => Difficult to make fast / What to prefetch?
- Used to be a bigger problem in the past (seek time and rotational delay)
- Still problematic because it undoes prefetching benefitsproactivity
- Examples: Updating records in a database file

Allocating storage space to files

> Files are just an abstraction => Need actual physical storage, for the data akin to how virtual memory needs physical memory

- A longer conversation on Hard Disk Drives (HDDs) and rotational moves would be had, were it not for Solid State Drives (SSDs)...

> Storage devices w/o moving parts

> Faster and more reliable than HDD

> Still quite slow, compared to main memory

- Be proactive => Prefetch data (leverage spatial locality)
- Hide latency => Do slow storage operations asynchronously

Calculating major vs minor page fault latency

```c
int main(int argc, char **argv) {
    // Assuming file already exists at path
    if ( (fd = open("/tmp/foo.txt", O_RDONLY, 0664)) < 0)
        return -1;
   char *buf = mmap(NULL, page_size, PROT_READ, MAP_PRIVATE, fd, 0);
   start_time = clock_gettime_ns();
   char a = buf[0];
   end_time = clock_gettime_ns();
   printf("time elapsed: %lu ns (1st read) \n", end_time -  start_time);
   start_time = clock_gettime_ns();
   char a = buf[0];
   end_time = clock_gettime_ns();
   printf("time elapsed: %lu ns (2nd read) \n", end_time -  start_time);
}
```

```
# 1 -> Drop in-kernel page caches
➜  git:(master) ✗  echo 1 | sudo tee /proc/sys/vm/drop_caches
1
➜  git:(master) ✗  ./demo
Time elapsed: 448,834 ns (1-st read) <-- Major page fault / Going to strorage
Time elapsed: 41 ns (2-nd read)
➜  git:(master) ✗  ./demo
Time elapsed: 11,041 ns (1-st read) <-- Minor page fault / Staying in main mem.
Time elapsed: 42 ns (2-nd read)
```

Allocating storage space to files

How do we allocate persistent storage space to files?

> Systems people are very predictable…

> Split storage to fixed-size chucks, called *blocks*

> Use n blocks to serve a file, where n = filesize / blocksize

> Allocation strategies

- Contiguous allocation
- Linked allocation
- Indexed allocation
- Multi-level indexed allocation

Contiguous allocation

Allocate a contiguous set of blocks, of sufficient size to each file

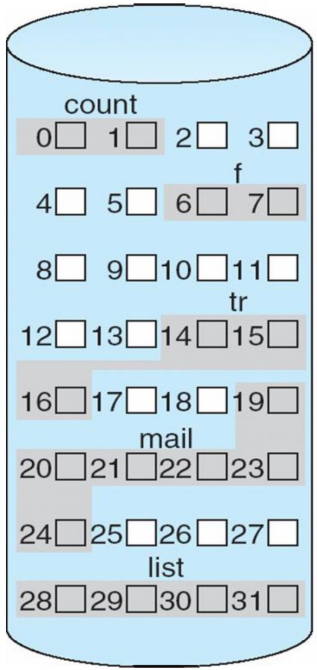
- File metadata: Starting block and no of blocks
- System-wide bitmap of free blocks

› Advantages

- Low storage overhead => Two vars per file
- Fast sequential access => Consecutive blocks
- Quick calculation of blocks for random accesses

› Disadvantages

- Difficult to grow a file
- External fragmentation



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Image from: OS Concepts, by A. Silberschatz et al.

Linked allocation

Allocate a linked list of blocks, with each block holding a pointer to the next block; essentially, per-file, an on-disk linked list

- File metadata: A pointer to the first block
- System-wide bitmap of free blocks

> Advantages

- No fragmentation
- Files can easily grow dynamically

> Disadvantages

- Slow on random accesses
- Storage overhead => One ptr per block
- "Unoptimizable:" Index cannot be cached
- **Unreliable:** Loose one block => loose everything

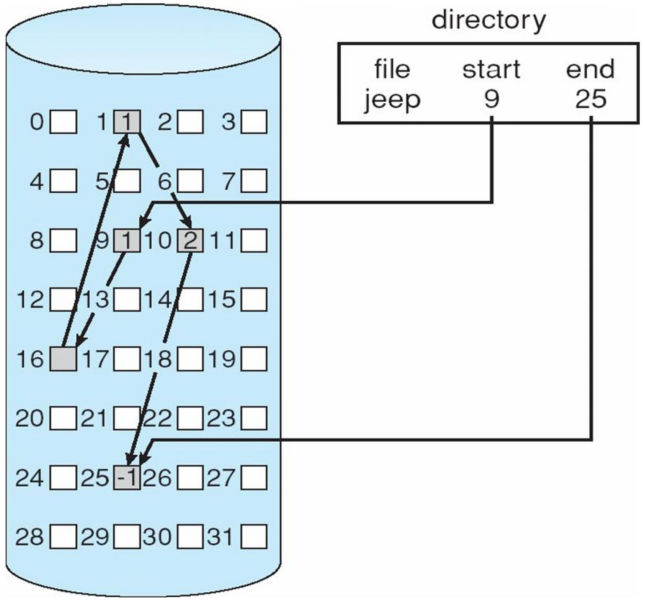


Image from: OS Concepts, by A. Silberschatz et al.

Indexed allocation

Use a special index block (*inode*) to store pointers to the data blocks

- File metadata: Location of the inode block on disk
- System-wide bitmap of free blocks

› Advantages
  - No fragmentation
  - Files can easily grow dynamically
  - Fast random access (**How?** Cache inodes)

› Disadvantages
  - Sequential bandwidth may not be good
  - What if one index block is not big enough?

    …We've seen this story before!

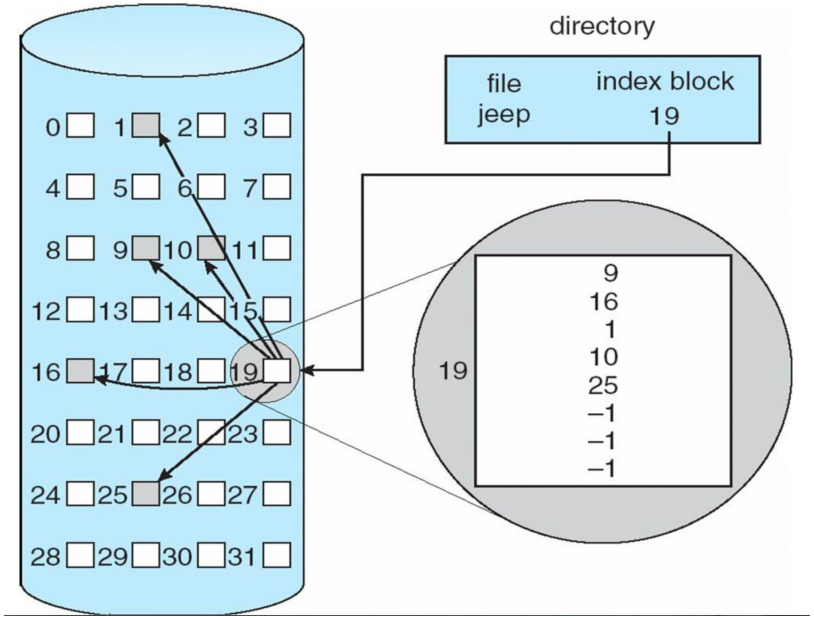


Image from: OS Concepts, by A. Silberschatz et al.

Multilevel Indexed allocation (Linux ext2/3)

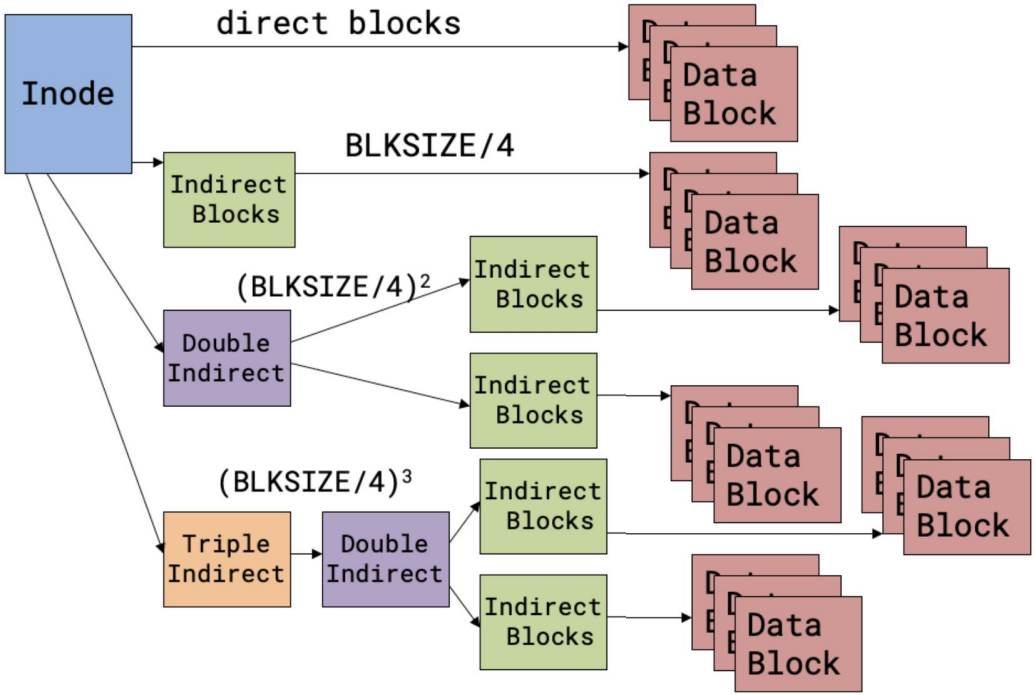


Image from: COMS 4118, Operating Systems I

Assume 4KB blocks and 4 bytes ptrs

> **A typical 256 bytes inode has**

- 12 direct block pointers
- 1 indirect block pointer
- 1 double indirect block pointer
- 1 triple indirect block pointer

What is the max supported file size?

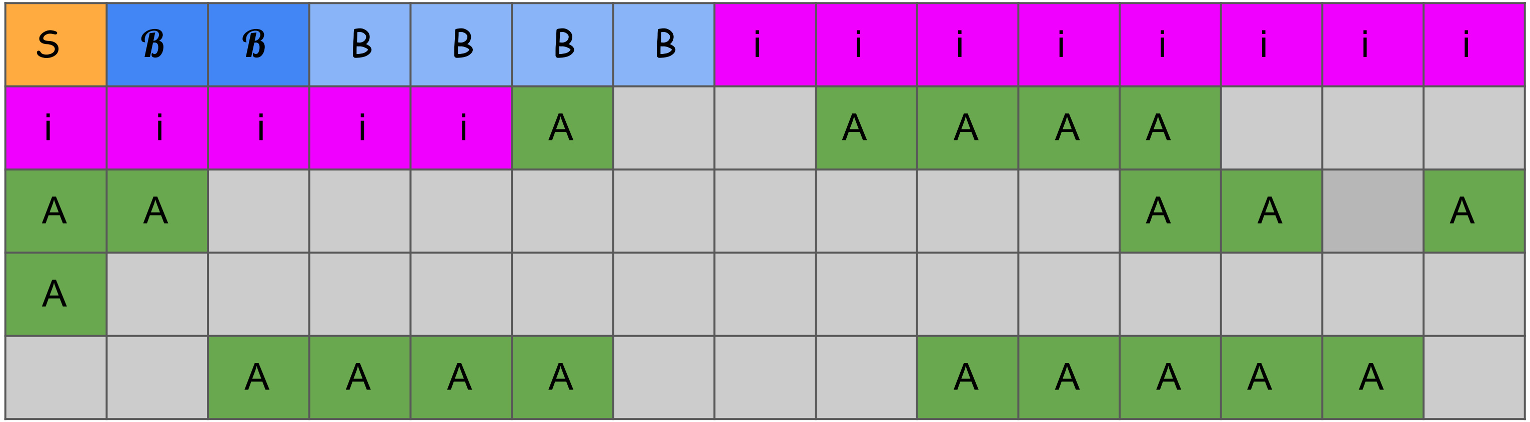
> (12 + 1024 + 1024^2 + 1024^3) *4KB > **4TB**

And for what index size?

> 12 + (1 + 1024 + 1024^2) * 4KB ~ **4GB**

Index grows dynamically, on demand...

Simplified Filesystem Layout (Linux ext2/3)



- S — Superblock (holds pointer to the inode of root dir "/")
- B — Bitmap blocks (inodes)
- B — Bitmap blocks (data)
- i — Inodes' blocks
- A — Data blocks (free)
- Data blocks (allocated)

File and Directory Names

> Humans do not refer to files and directories via inode numbers, but via file names and directory names (similarly to how programs refer to physical memory via virtual addresses)

- Files and dirs are organized on an acyclic-graph hierarchy
- Special "/" root directory: All names are paths starting it
  - "/foo/test.txt" => "/" -> "foo" -> "test.txt"
  - "/foo/bar/test.txt" => "/" -> "foo" -> "bar" -> "test.txt"

File and directory aliases

> Hard link: Associates a name with an inode (>=1 files, =1 dirs)

> Soft link: Associates a name with an inode of a file containing paths to files

Path Name Resolution

We need a fast translation from path names to inodes

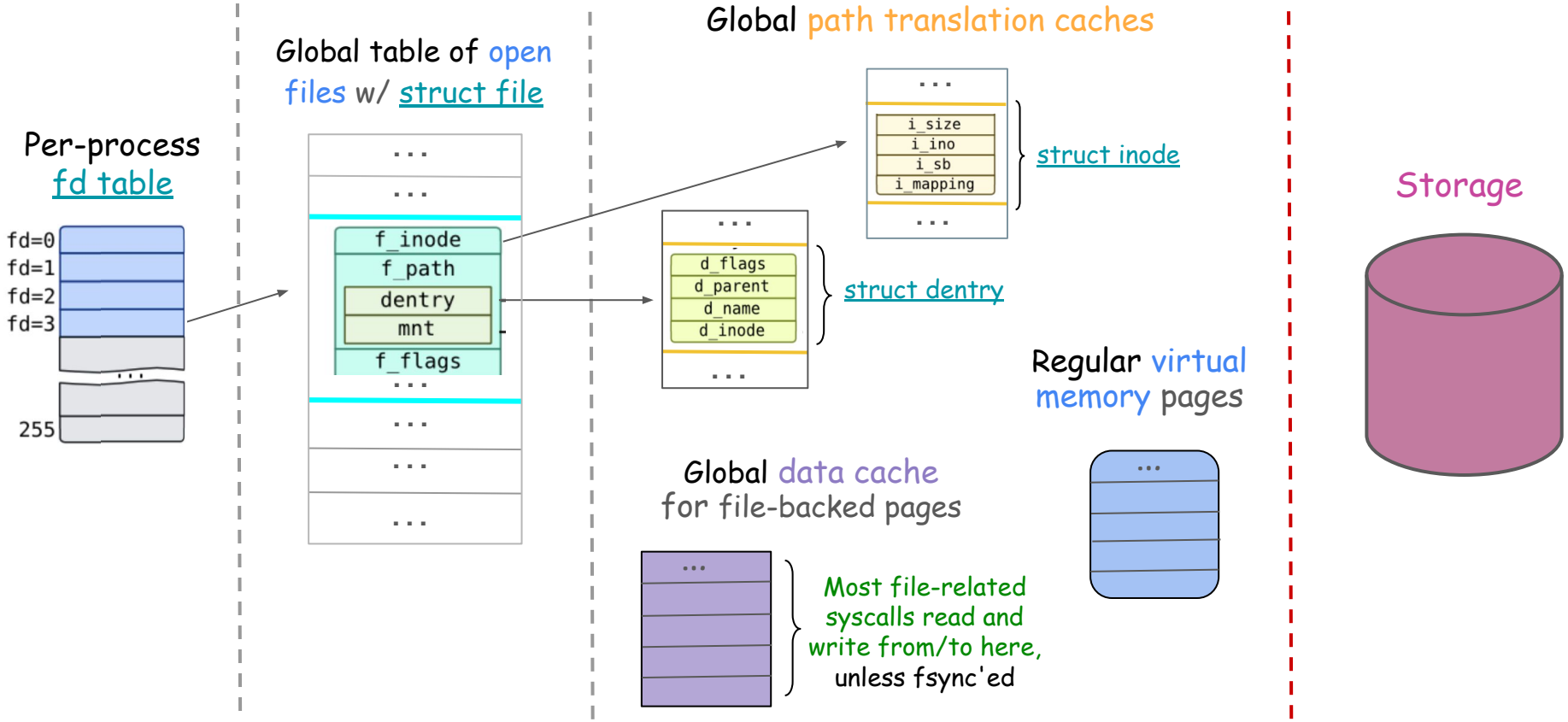
> File and dir names are **paths** starting from root (Let a TLB PTSD kick in!)

Example: "/foo/test.txt" => "/" -> "foo" -> "test.txt"

1. Read superblock to look up inode no of "/"
2. Read inode block of "/" to look up data blocks of "/"
3. Read data block of "/" to look up inode no of "/foo"
4. Read inode block of "/foo" to look up data blocks of "/foo"
5. Read data block of "/foo" to look up inode number of "/foo/test.txt"
6. Read inode block of "/foo/test.txt" to look up data blocks of "foo/test.txt"
7. Read data blocks of "foo/test.txt"

Need to speed this translation disaster up

Linux File System Data Structures



The consistent update problem

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

› Updating the file system from one consistent state to another requires atomically modifying several blocks (inode, bitmap, data)

› Storage devices allow atomic writes of one block at a time

› Crashes may happen at any time leaving the fs inconsistent

Update data	Update data bitmap	Update inode	Outcome
yes	no	no	Missed update
no	yes	no	Space leak
no	no	yes	fs inconsistent

} Could read garbage data—Example?

Ordered Writes

Assuming writing a single block is atomic, how to ensure that updates to the file system (fs) occur atomically?

> Ordered writes: Prevent fs inconsistencies by write blocks to disk in safe order: Write data blk -> Write data bitmap blk -> Write inode blk

- Ensures inodes never point to uninitialized data!
- Can leak resources (fixable: run *fsck* periodically)
- Cannot reorder writes and execute asynchronously <= Dealbreaker

Journaling (also called "write-ahead" logging)

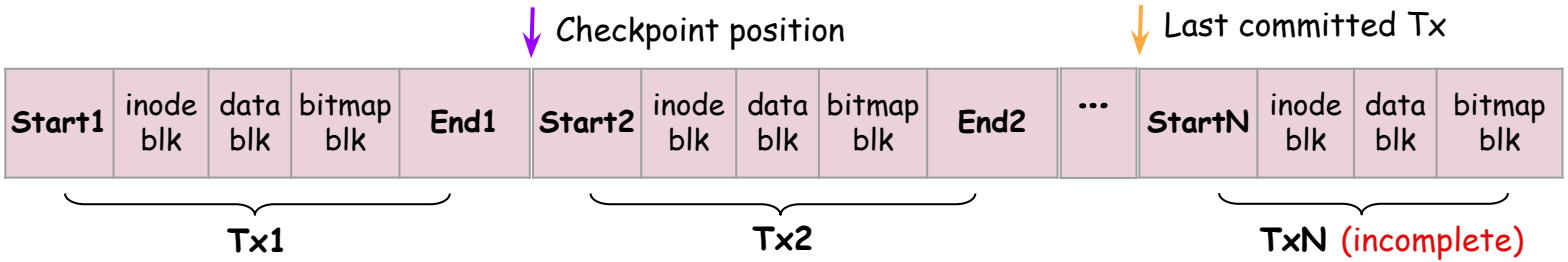
> Write down what you are going to do before doing it in a special append-only log file, called *"journal"* (see here for Linux ext3/4)

I) Journal Write: Write all blocks of TxN to the journal w/o *End* block

II) Journal Commit: Write TxN's *End* block to the journal

III) Journal Checkpoint: After data and metadata blks have been updated at their final storage destination, update ckpt postion

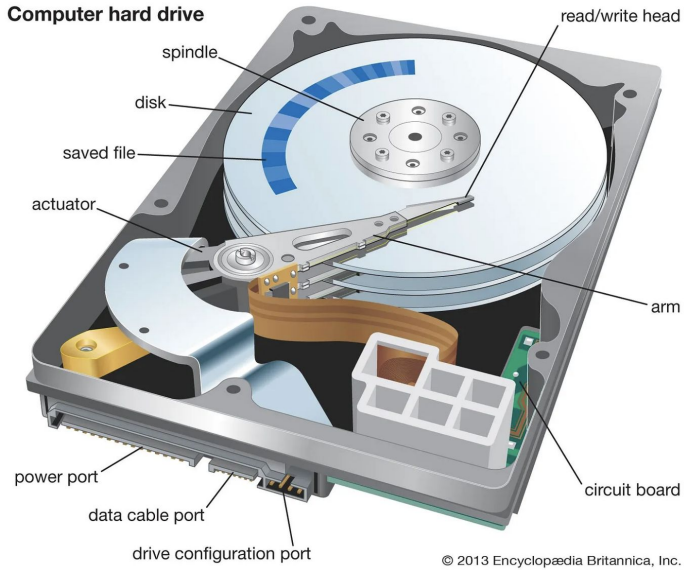
Crash occured? Replaying committed transactions after the last checkpoint will bring the fs to a consistent state (crash tolerance)



Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

- Storage failures are irreversible => No restart button



It is impressive that this piece of machinery carried us for so many decades...

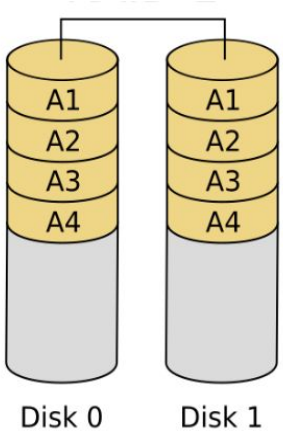
Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

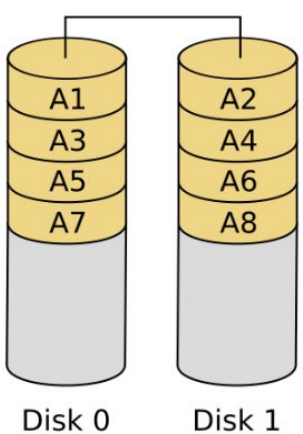
> How to build fault tolerant systems using unreliable hardware? Replication

Idea: Redundant Array of Inexpensive Disks (RAID), 1988, D. Patterson et al.

RAID-1 (disks >= 2)



RAID-0 (disks >= 2)



2x Throughput

No Fault tolerance

> Worse than single disk

Images from: Wikipedia

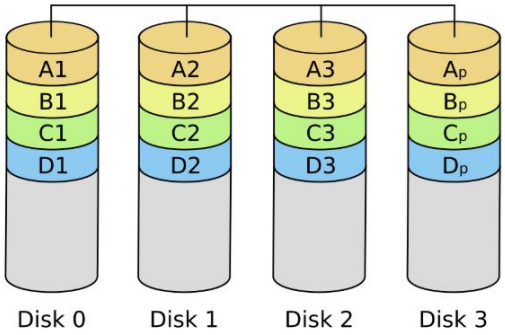
Fault Tolerance

A system's ability to continue operating correctly despite hardware failure (i.e., *faults*) is called fault tolerance

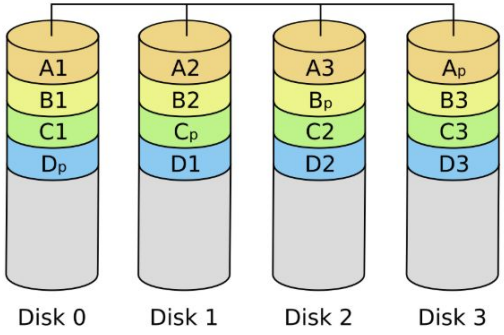
› How to build fault tolerant systems using unreliable hardware? Replication

Idea: Redundant Array of Inexpensive Disks (RAID), 1988, D. Patterson et al.

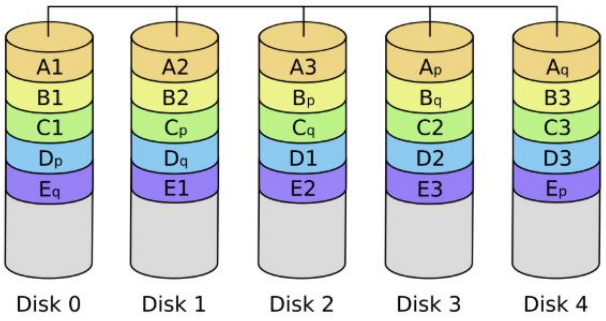
RAID-4 (disks >= 3)



RAID-5 (disks >= 3)



RAID-6 (disks >= 4)



>> **Fault Tolerance:** RAID-0 < RAID-4 ≈ RAID-5 < RAID-6 < RAID-1

Images from: Wikipedia

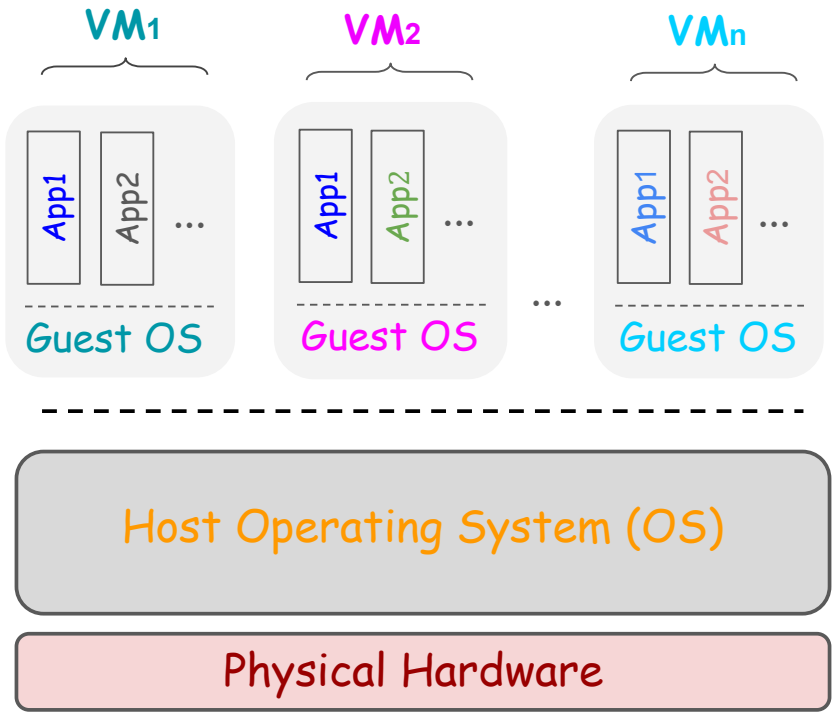
What is Virtualization?

The ultimate destination: Abstraction of physical host hardware into mu isolated guest machines, called Virtual Machines (VMs)

> Each VM can run **any** unmodified guest OS

> Guests OSes cannot distinguish whether they run on a VM, or on a physical machine

What is Virtualization?



Why virtualization?

Sustainable model for management of compute

- Want to maintain your servers?
  - Hardware, software, cooling, pay the bills?
- Electricity paradigm
  - Yeah...you can have your own generator, but would you?

Better hardware utilization of large host machines

- Guesses on avg. CPU utilization of major cloud providers?
  - Around 30%
- Resource Overcommitment: More virtual than physical CPUs (~10:1)
- Resource Elasticity
  - VMs on/off on demand (saves customer $$)
  - VMs live-migrated across hosts (saves provider $$)

Why virtualization?

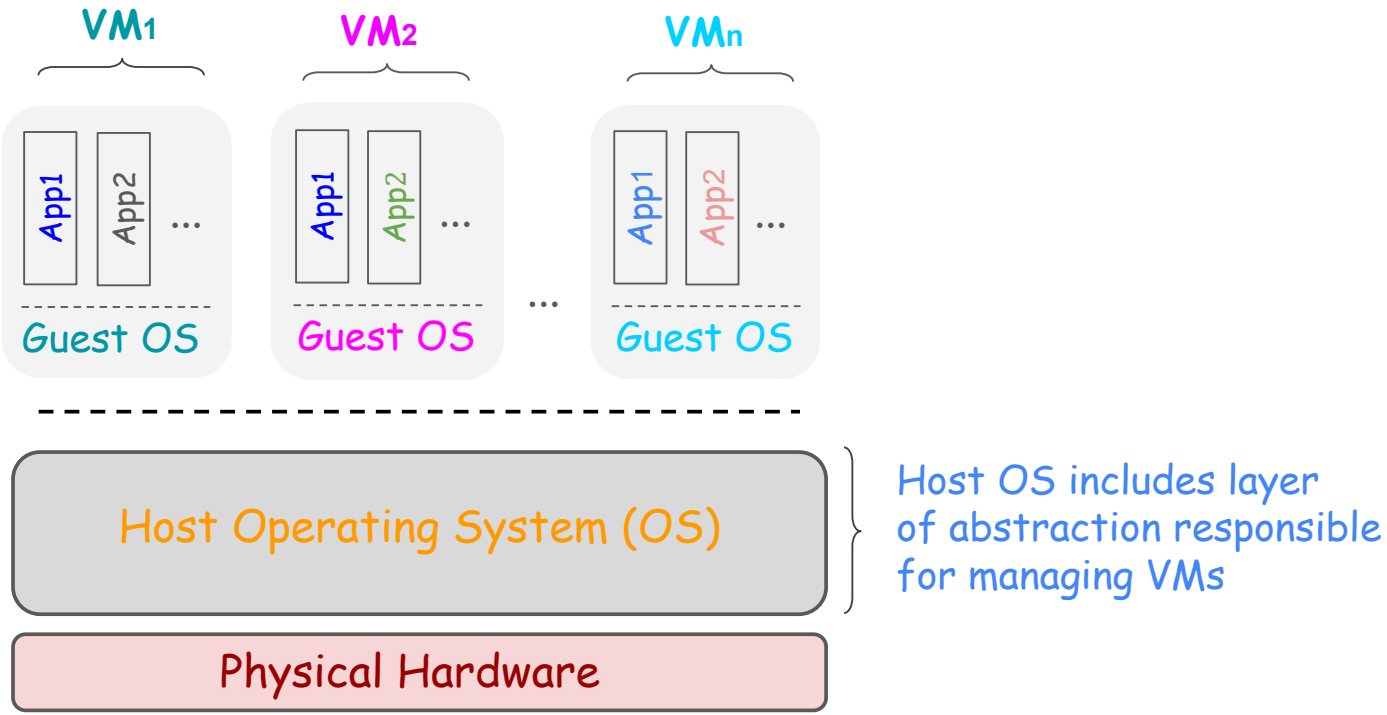
Fault isolation at machine level

- Bugs in one guest VM do not affect others
- Compromises in one guest VM do not affect others

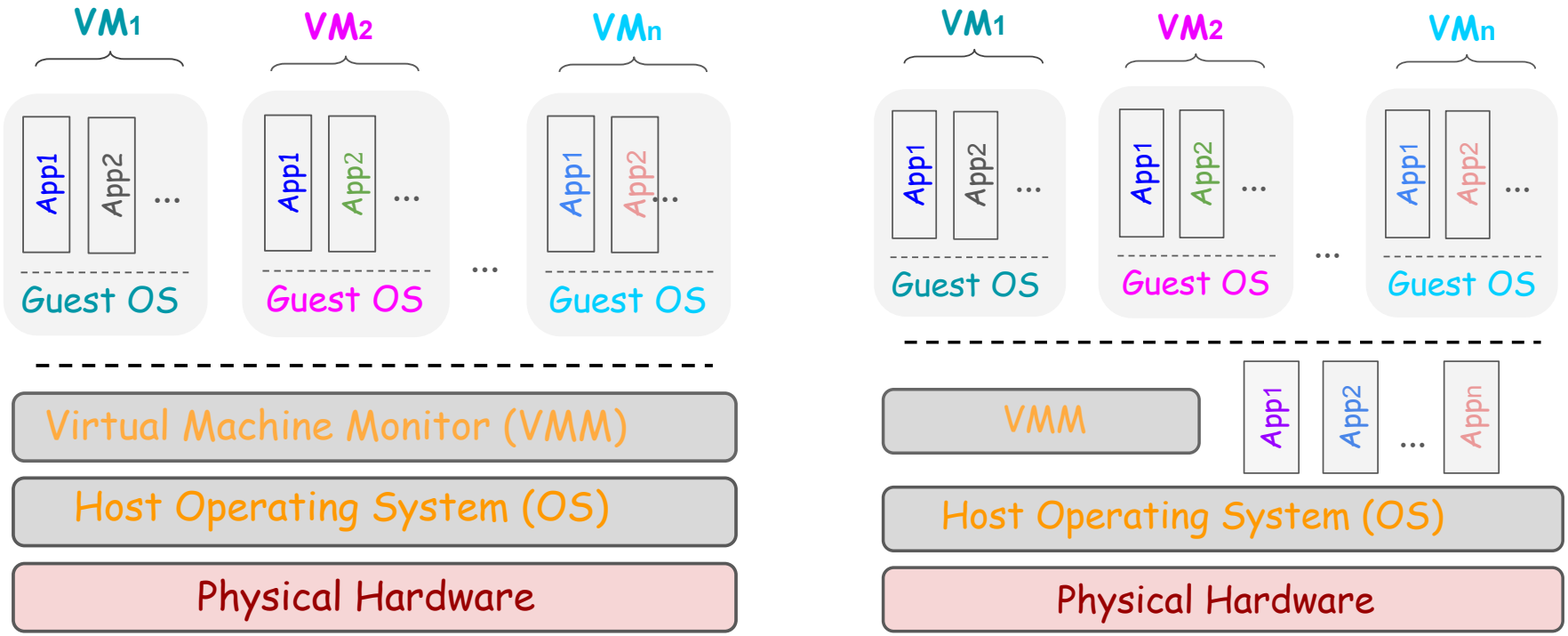
Speeds up development/testing of user-space code

- Develop and test apps for multiple OSes using a single machine

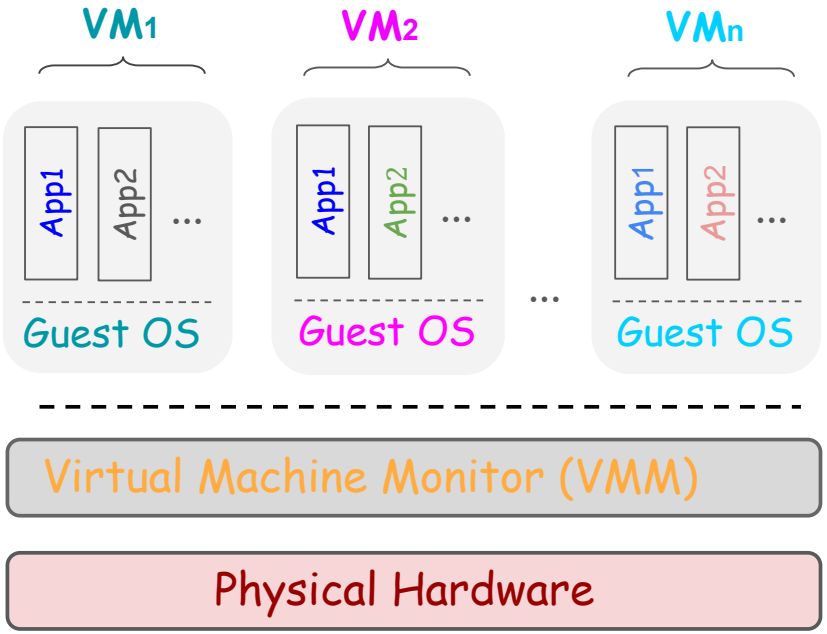
Virtual Machine Monitor (VMM) Approach



Virtual Machine Monitor (VMM) Approach



Type-1 Hypervisor (bare metal)

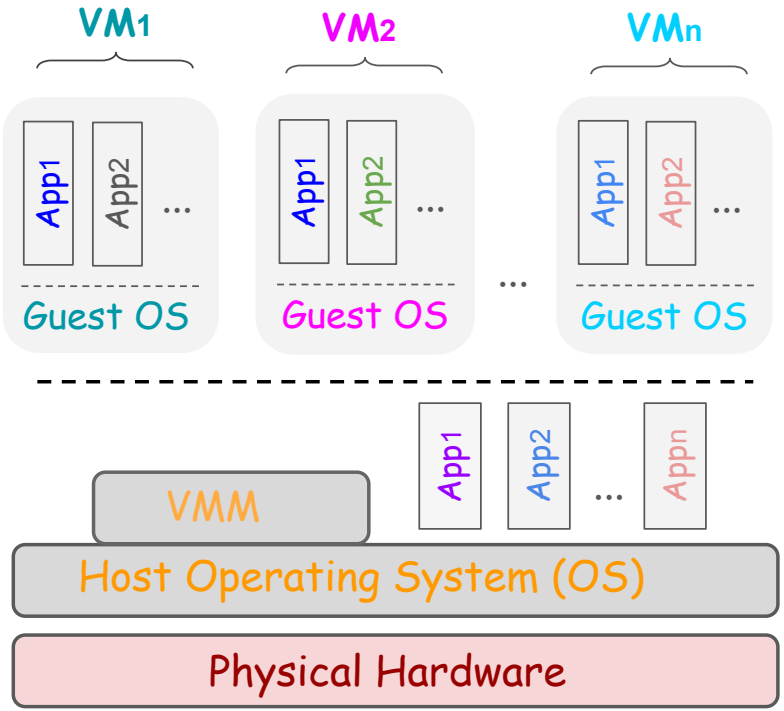


› Characteristics: Small codebase w/ scheduler, mem. management, and interrupt handling

› Popular representatives

- VMWare ESXi (Broadcom)
- Microsoft Hyper-V (Microsoft)
- Xen (University of Cambridge)

Type-2 Hypervisor (hosted)



> Popular representatives

- KVM (Linux)
- Parallels (Parallels)
- VirtualBox (Oracle)
- VMware Workstation (Broadcom)

Hypervisors used by Major Cloud Providers

Provider	Hypervisor	Hosts (est.)	VMs (est.)
AWS	KVM (nitro)	3 Millions	25 Millions
Azure	Hyper-V	2 Millions	20 Millions
GCP	KVM	1 Million	10 Millions
Alibaba	KVM	0.8 Million	7 Millions
Oracle Cloud	KVM	0.3 Million	2 Millions

What needs to be virtualized?

› Processor
- Timeshare physical host processor cores to guest VMs
- Handle privileged instructions

› Memory
- Spaceshare host physical mem. to guest VMs physical mem.
- Provision guest page tables; hardware walks them w/o asking…

› Events (Exceptions and Interrupts)
- Vector hardware events to the correct guest VM
- Multiplex external hardware devices

Virtualizing the Processor: Time

VM's CPU cores must run on host's physical CPU cores

- > The hypervisor time-slices the host's physical cores
- > Each VM cores runs for, at most, their dedicated timeslice
- > Type-1 hypervisors may use a simple RR scheduler
  - E.g., Xen: sched_init_vcpu
- > Type-2 hypervisors may use host's kernel sched subsystem
  - E.g., KVM: vmx_vcpu_create
  - Schedulable ctxts (user-space threads) on host's scheduler

Virtualizing the Processor: Privileged Instructions

Full instruction simulation

- > Interpret each guest instruction
- > Simulate its execution using guest's instructions
- > Maintain each VM's state purely in software
- > Example: Qemu

Problem: Too slow

Virtualizing the Processor: Privileged Instructions

Trap-and-emulate

> Execute non-sensitive instruction directly on host

> Privileged instructions? Trap => Hypervisor emulates them

> Need to emulate only a subset of commands

Problem: Not all sensitive instructions cause traps :-(

> E.g., Disabling interrupts on x86

Virtualizing the Processor: Privileged Instructions

Paravirtualization

> Change the guest OS to cooperate with the hypervisor

> Provide "hypervisor API" for guest sensitive ops

> Modify guest OSes to use hypervisor API

Tradeoff: Sacrifice transparency for better performance

Virtualizing the Processor: Privileged Instructions

Hardware-assisted virtualization

- Adding a new privilege level
- Unmodified guest OSes run there
- Sensitive operations cause a VM exit (akin to mode switch)
- E.g., Intel VT-x: VMX root vs. non-root mode
  - VMX non-root mode: Ring-0/3: Guest kernel-/user-space
  - VMX root mode: Hypervisor

Virtualizing Memory

Split host's physical mem. on guests' physical mem.

› Cannot interpose on a hardware-managed TLB miss
  - The walker will transparently walk the page tables
  - The hypervisor code has no chance of running
  - But…hypervisor must assign host pages to VMs

› Solution: Shadow page tables

Virtualizing Memory: Shadow Page Tables

> Hypervisor-managed replica of guest page table
  - Guest writes x86 %cr3? Privilege operation => Trap
  - Hypervisor marks guest page table read-only
  - Copies guest page table to shadow page table
  - Sets %cr3 point to shadow page table
  - Guest will use host's shadow page table w/o knowing

> Shadow page tables can be built on demand
- Guest reads its page table? Shadow page tbl. used
- Guest updates its page table? Shadow page tbl. updated

Virtualizing Events

Need to vector exceptions and interrupts to the correct VM

- Type-2 hypervisors (relatively easier to do)
  - VM exit occurs on interrupts
  - Hypervisor inspects exit reason (see KVM exit reasons)
  - Hypervisor delivers interrupt to vCPU
- Type-1 hypervisors
  - Full virtualization: Hypervisor emulates interrupt handlers' logic
  - Paravirtualization: Modify guest OSes w/ virtual interrupt event queue
  - Hardware support: Interrupt controller delivers events to guest OSes

AWS Case Study: Netflix

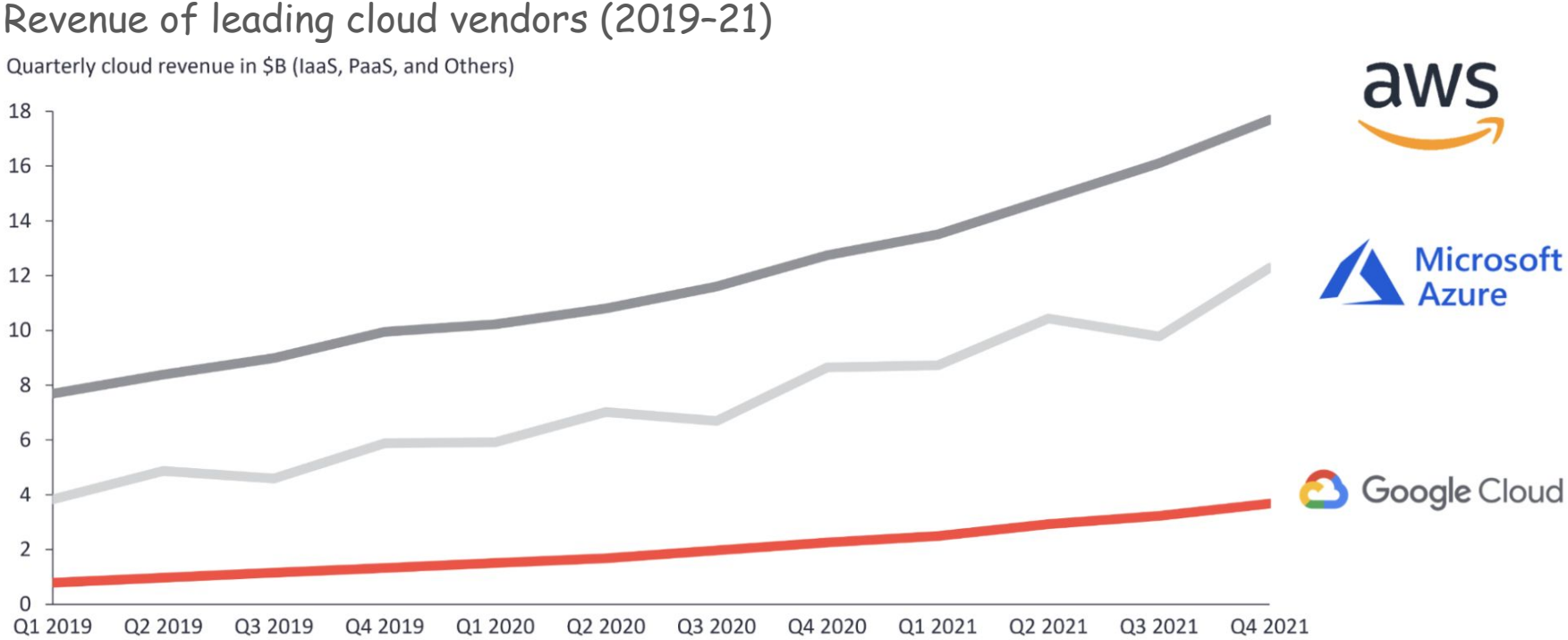
Seriously large company: 0.36% of S&P500

- Netflix market cap was $413B (as of yesterday)
- The GDP of Greece is ~$235B

Data is based on public resources [1,2,3,4]

- Phenomenal scale, even if data is a bit off
- Runs almost entirely on AWS using >100,000 VMs
- ~15% of global downstream internet traffic
- Serves >1,000 PB / day of video streaming

Blast from the past!



[1] https://iot-analytics.com/cloud-market/