K22 - Operating Systems: Design Principles and Internals

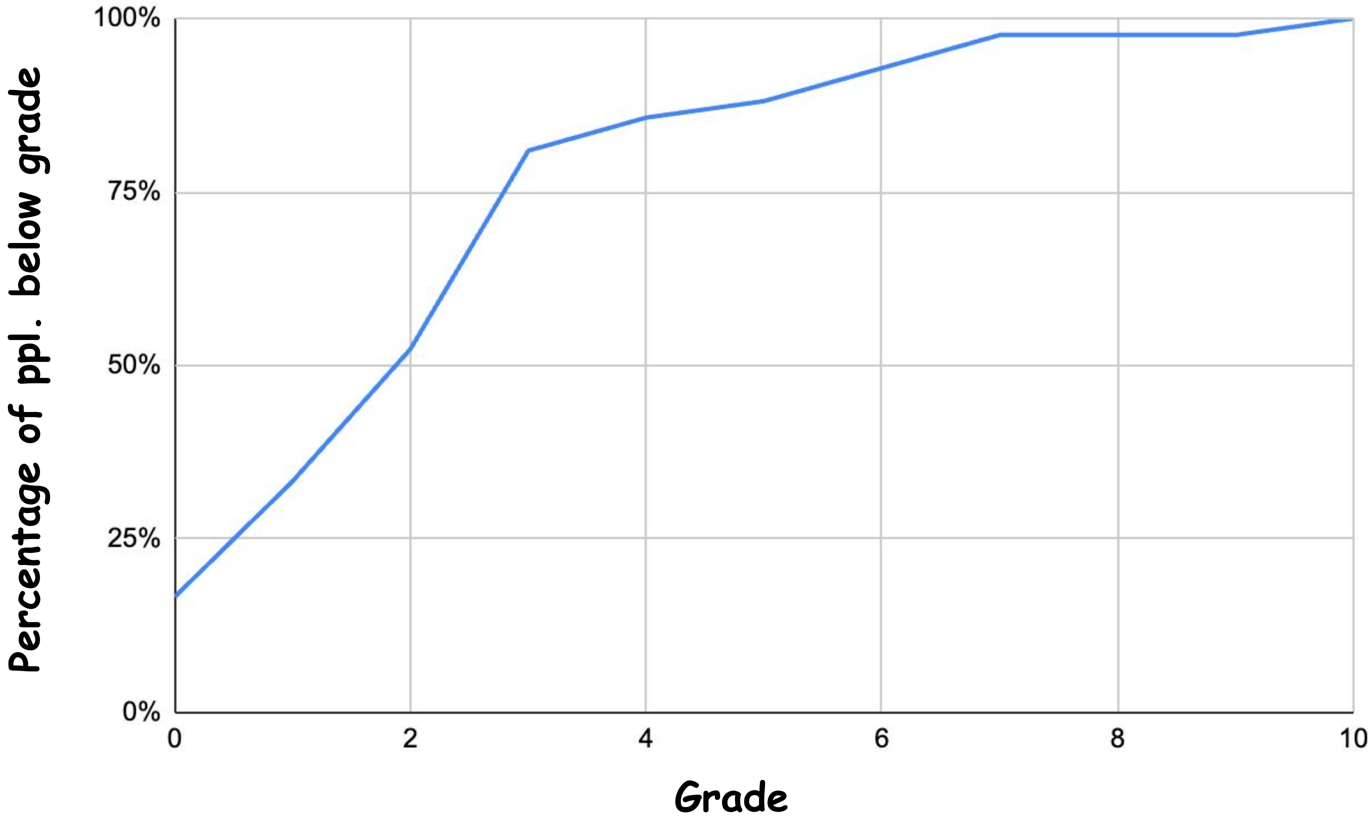
Fall 2025 @dit

Vaggelis Atlidakis

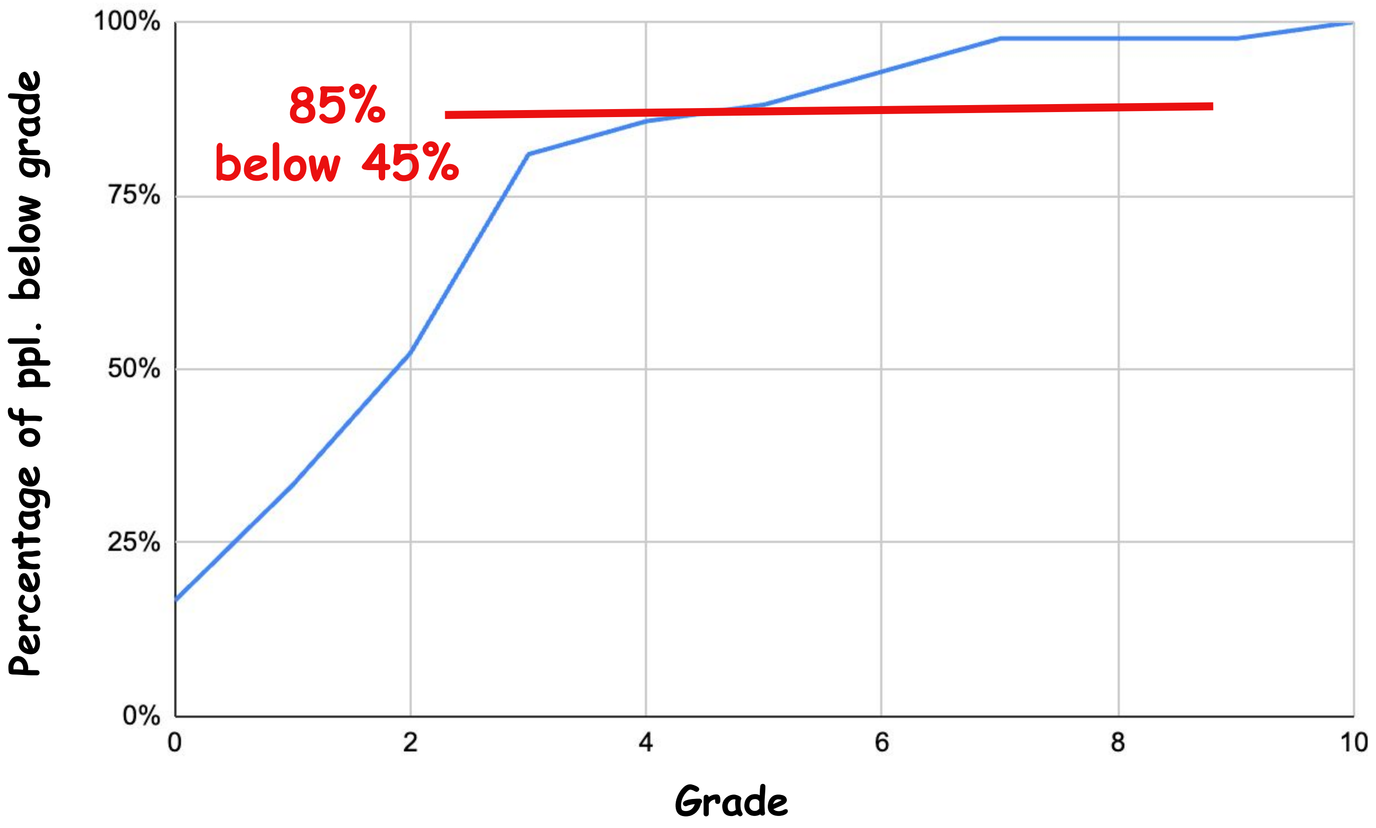
Lecture 06

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

Quiz-02



Quiz-02: You need to do something NOW.

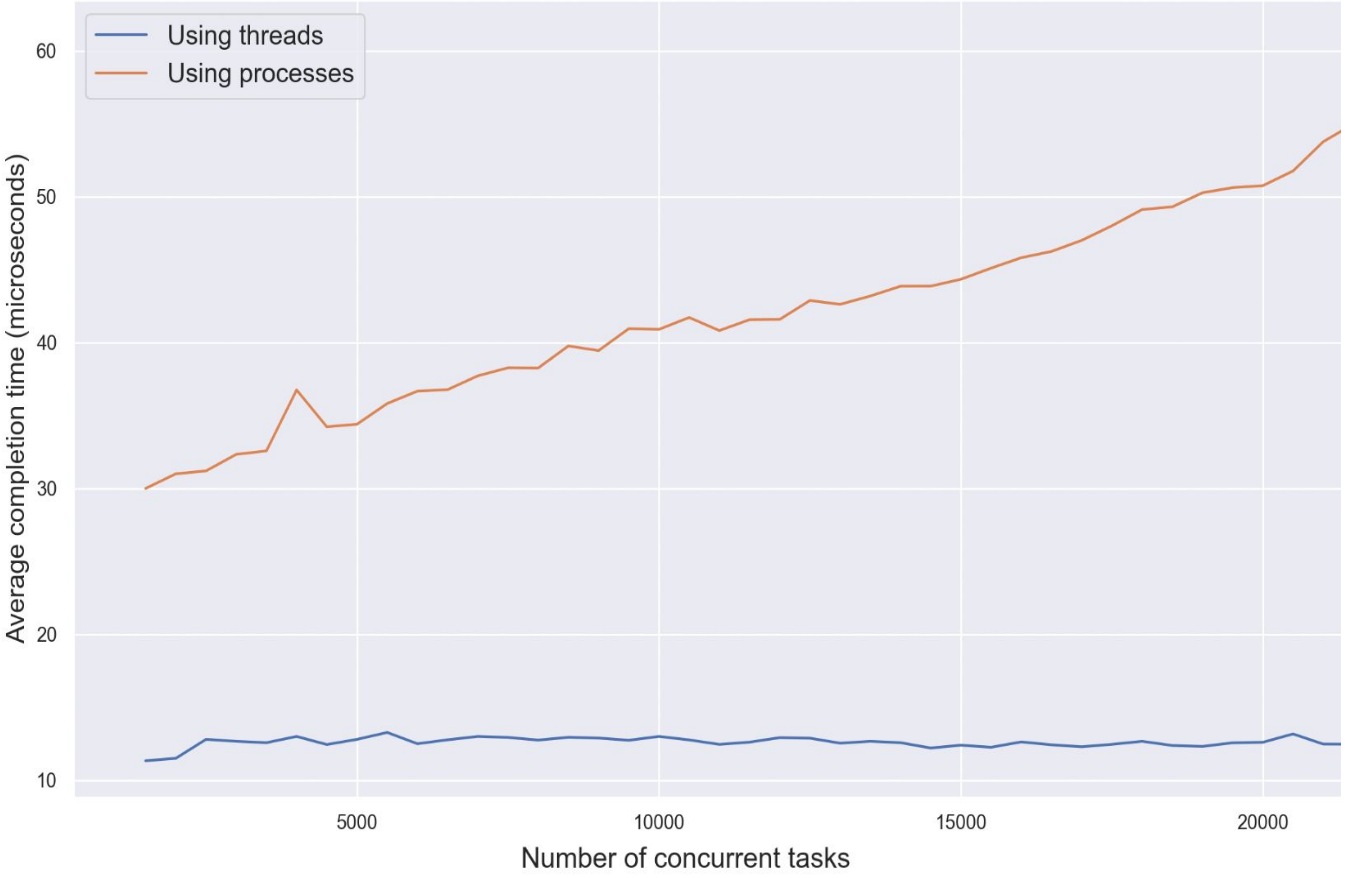


Overview

- 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

The miserable cost of fork()...




```
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);
}
```

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

Overview

- Inter-Process Communication (IPC)
  - Q1: How do processes communicate with each other?
  - Q2: Asynchronous IPC mechanisms?
  - Q3: Synchronous IPC mechanisms?

...

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?

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!

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)

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

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

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

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

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

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

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

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)

- 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

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

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)

  - 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

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 which are generated by some action attributable to a particular thread (such as a hardware fault) shall be delivered to the thread that caused the signal to be generated

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 which are generated by some action attributable to a particular thread (such as a hardware fault) 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 (such as terminal activity) 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

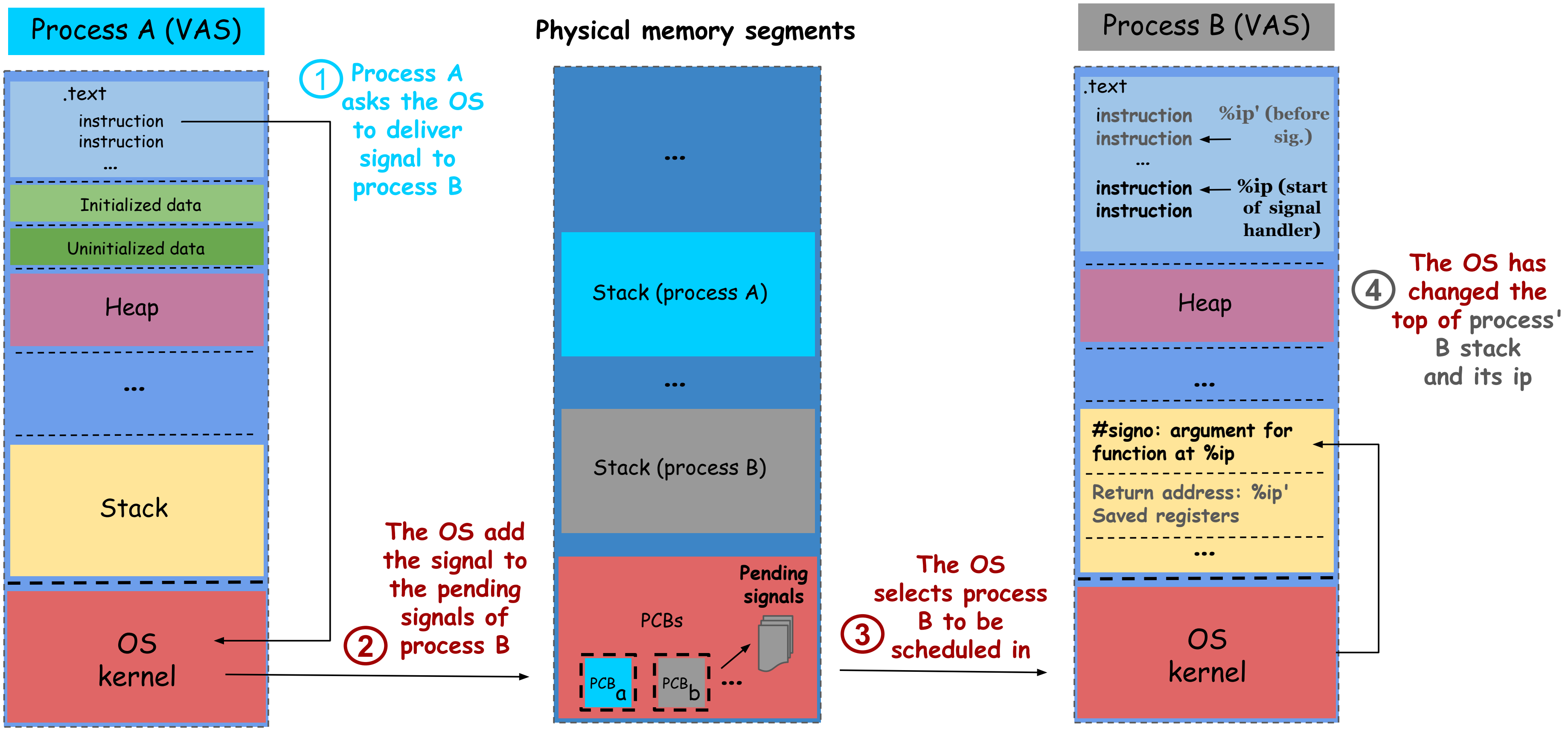
Asynchronous IPC: Signals

- 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

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

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

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

Synchronous IPC: Pipes (unnamed)

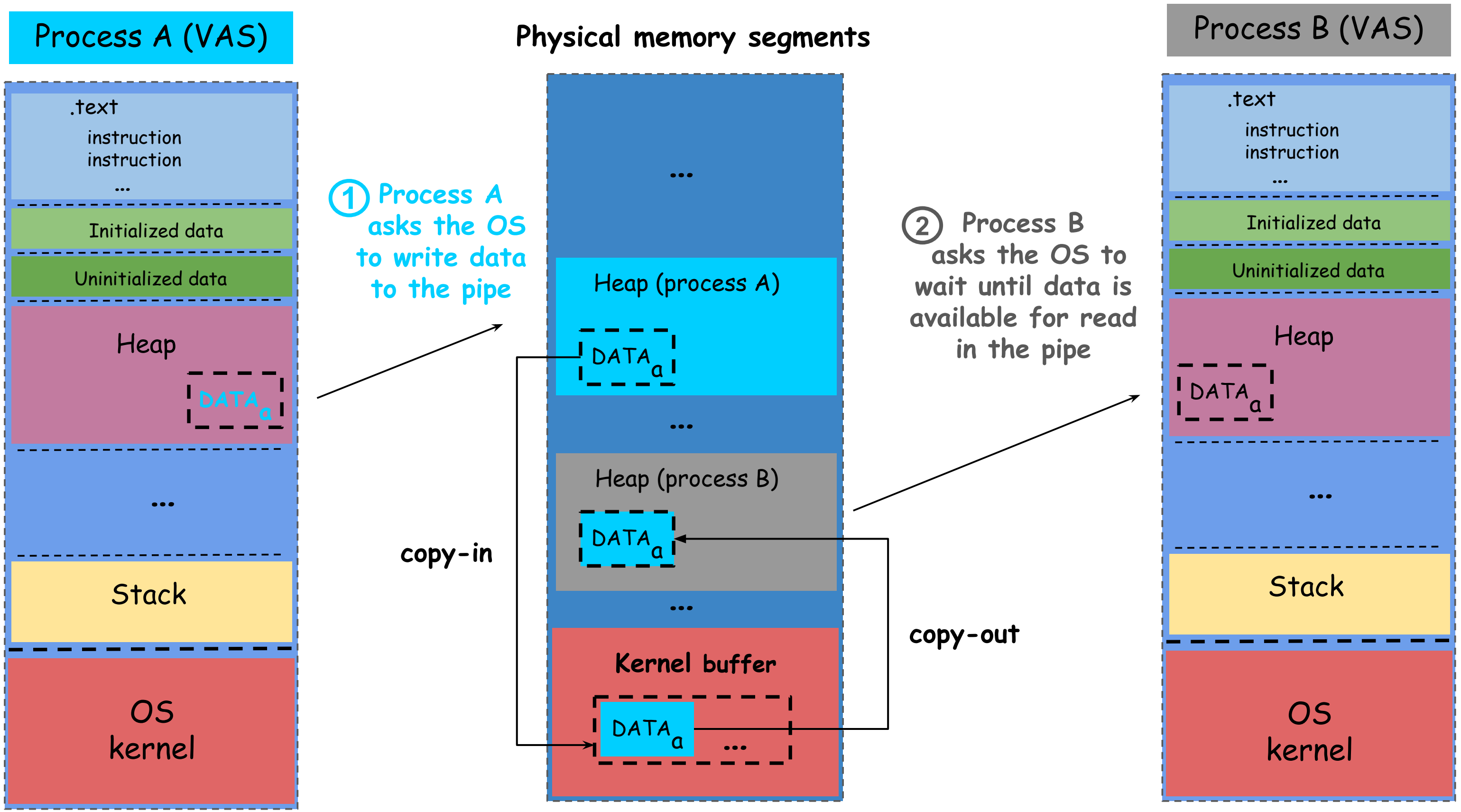
- POSIX pipes is the most common synchronous IPC mechanism
  - The syscall int pipe (int fds[2]) is used to ask the kernel to create a synchronous unidirectional communication channel b/w two processes
    - fds[0]: The read end of the pipe
    - fds[1]: The write end of the pipe

Limitation of unnamed pipes

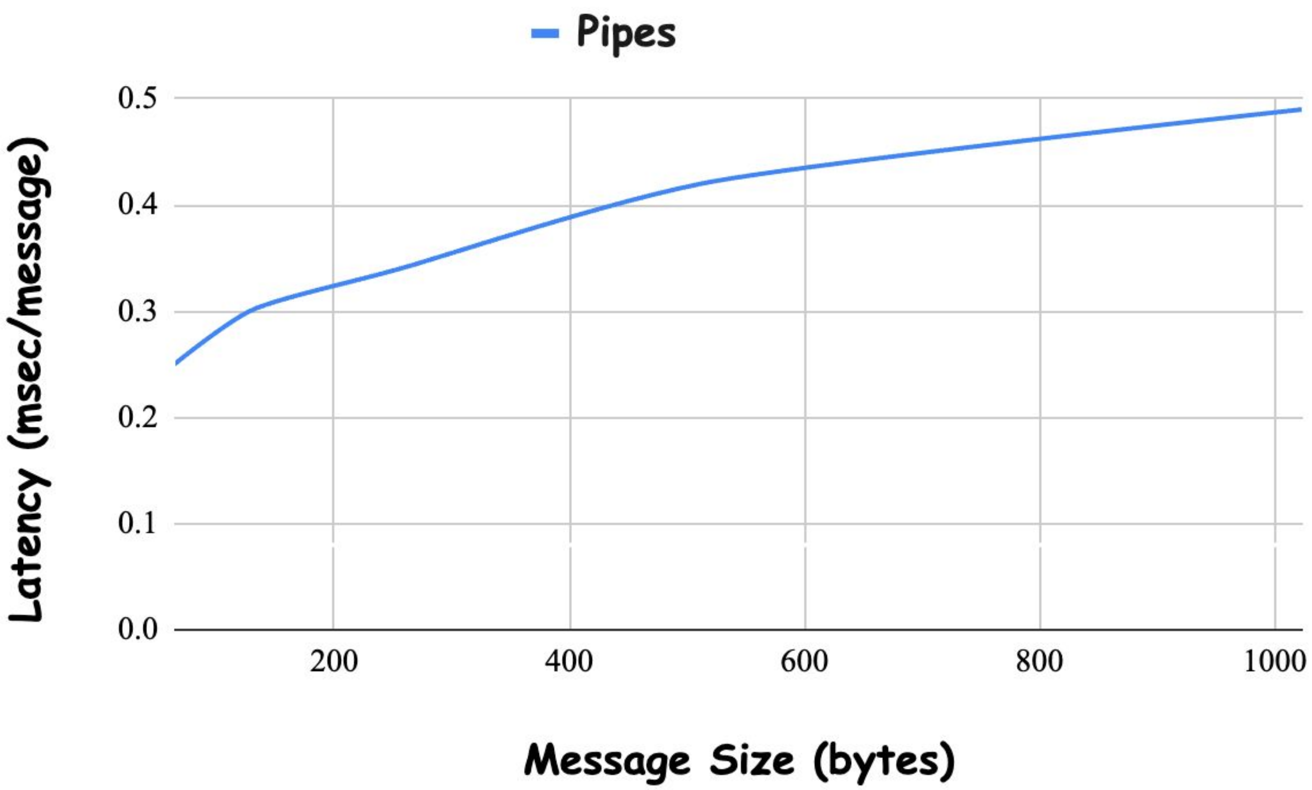
- The channel is unidirectional
- The channel can only be established b/w descendant processes

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

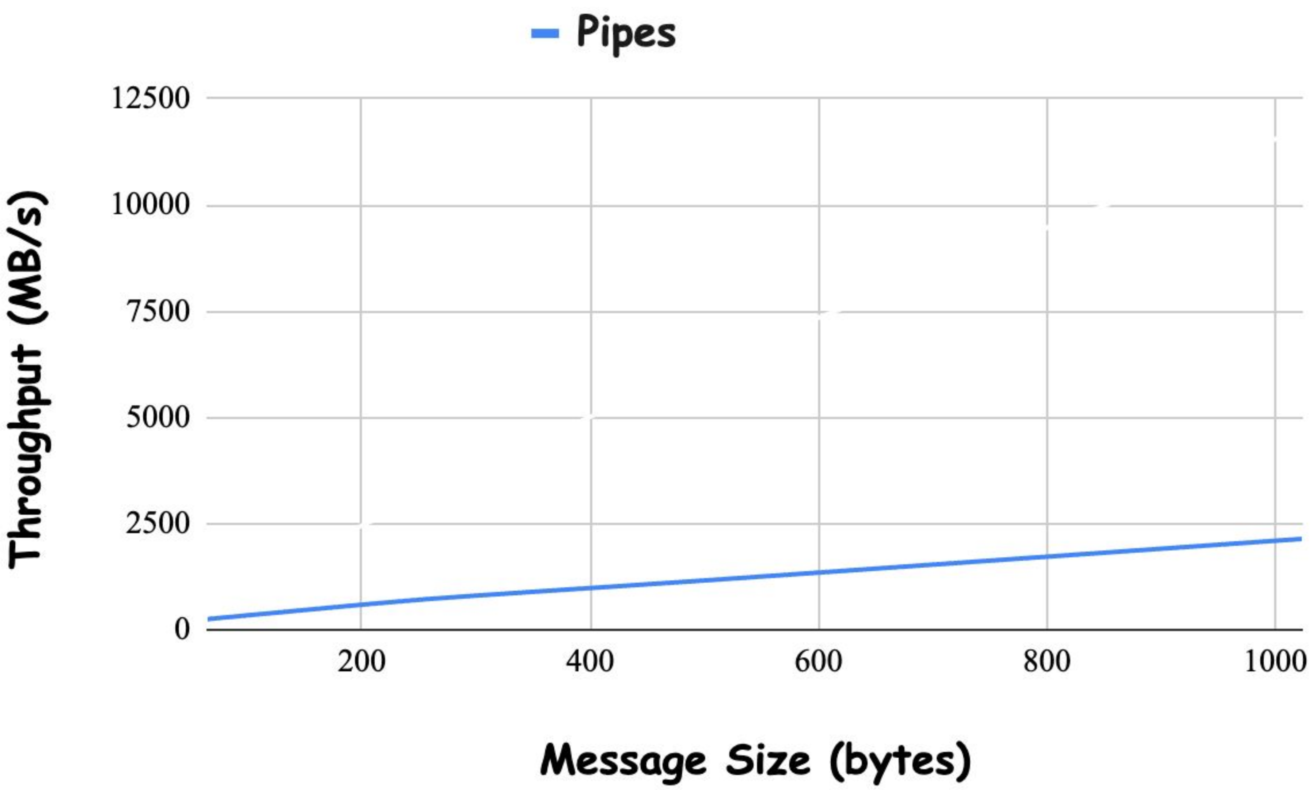
Synchronous IPC: Pipes (unnamed)



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

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)**

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

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

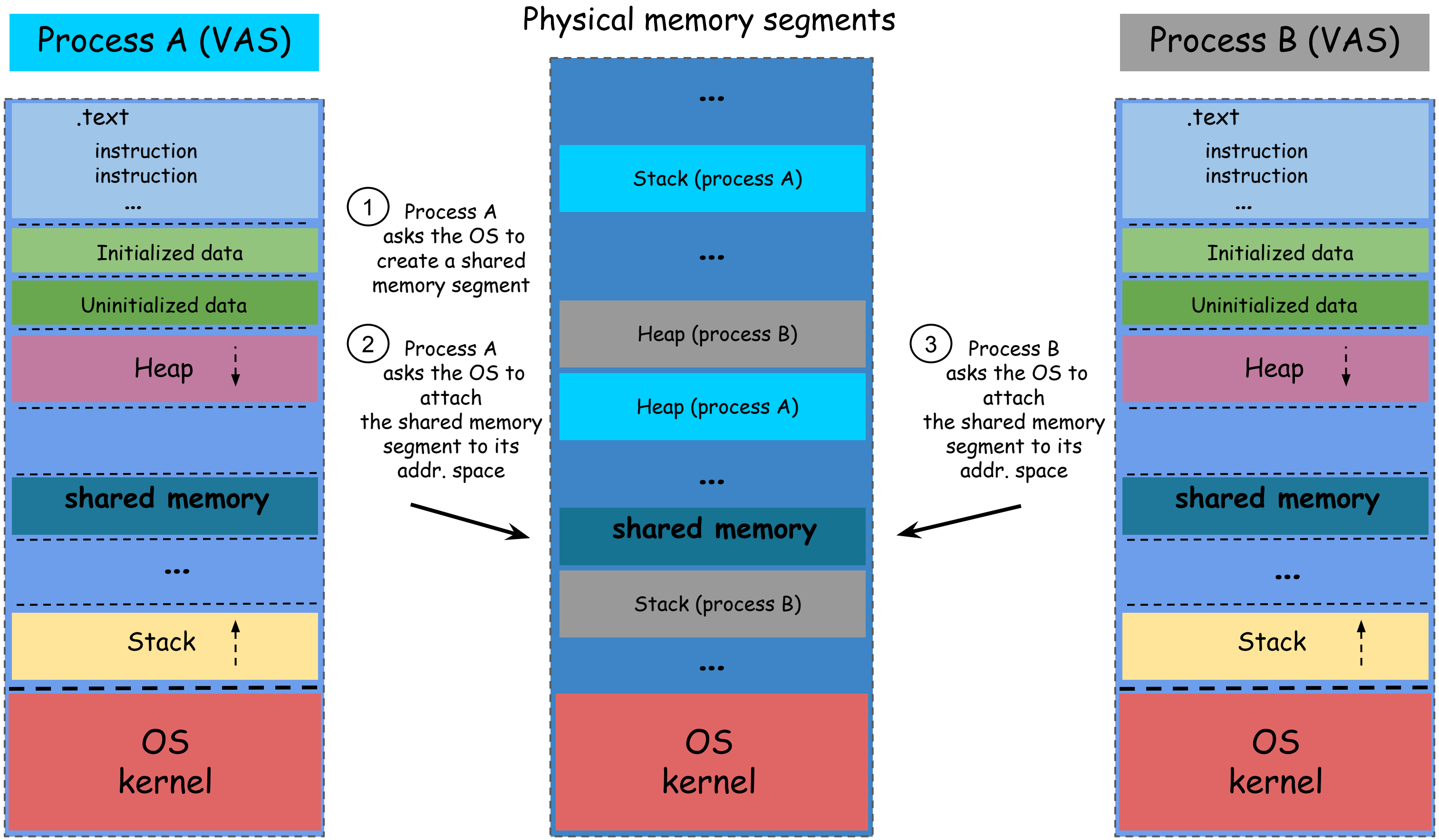
> How can processes avoid this overhead? **Ask the kernel to point directly to a memory region that is shared among processes (shared memory)**

- 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

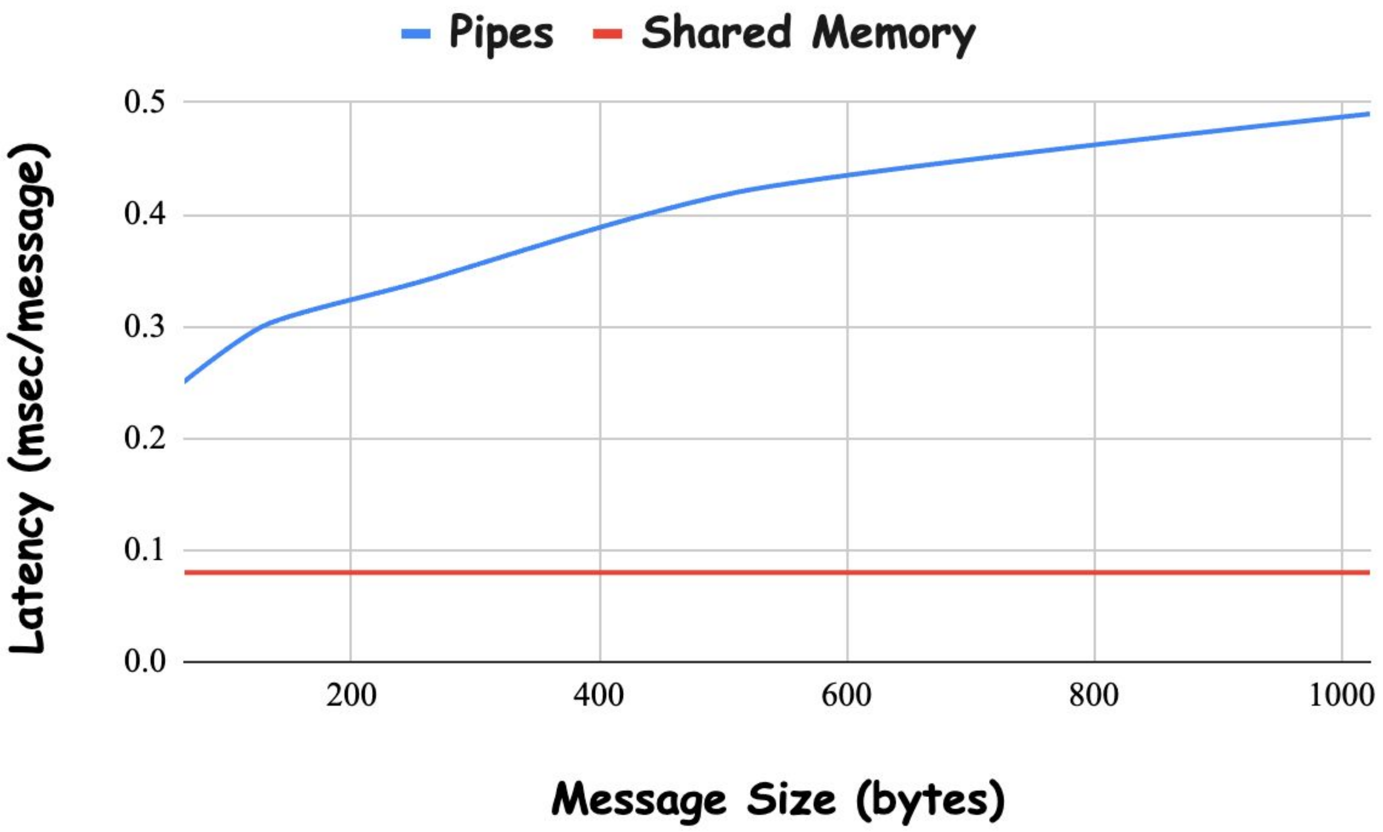
Synchronous IPC: Shared memory

- 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

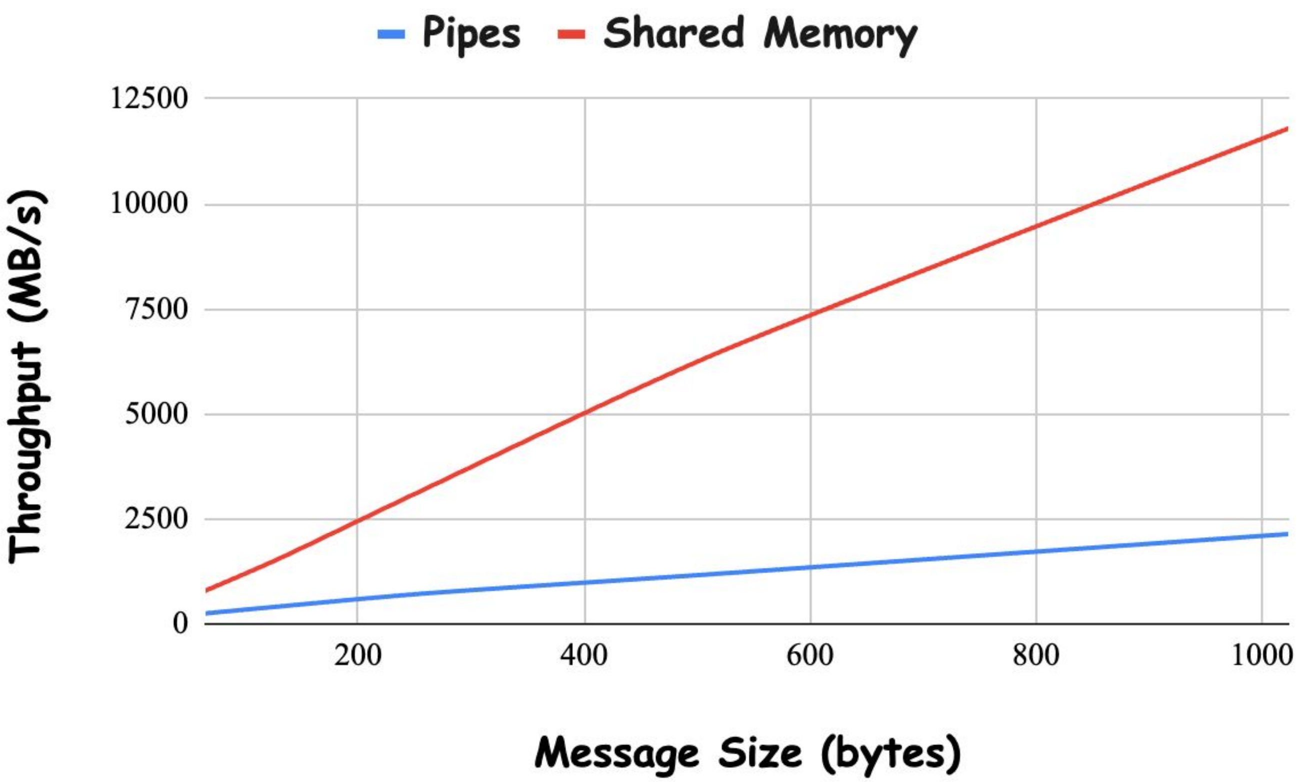
Synchronous IPC: Shared memory



Synchronous IPC: Shared memory vs Pipes



Synchronous IPC: Shared memory vs Pipes

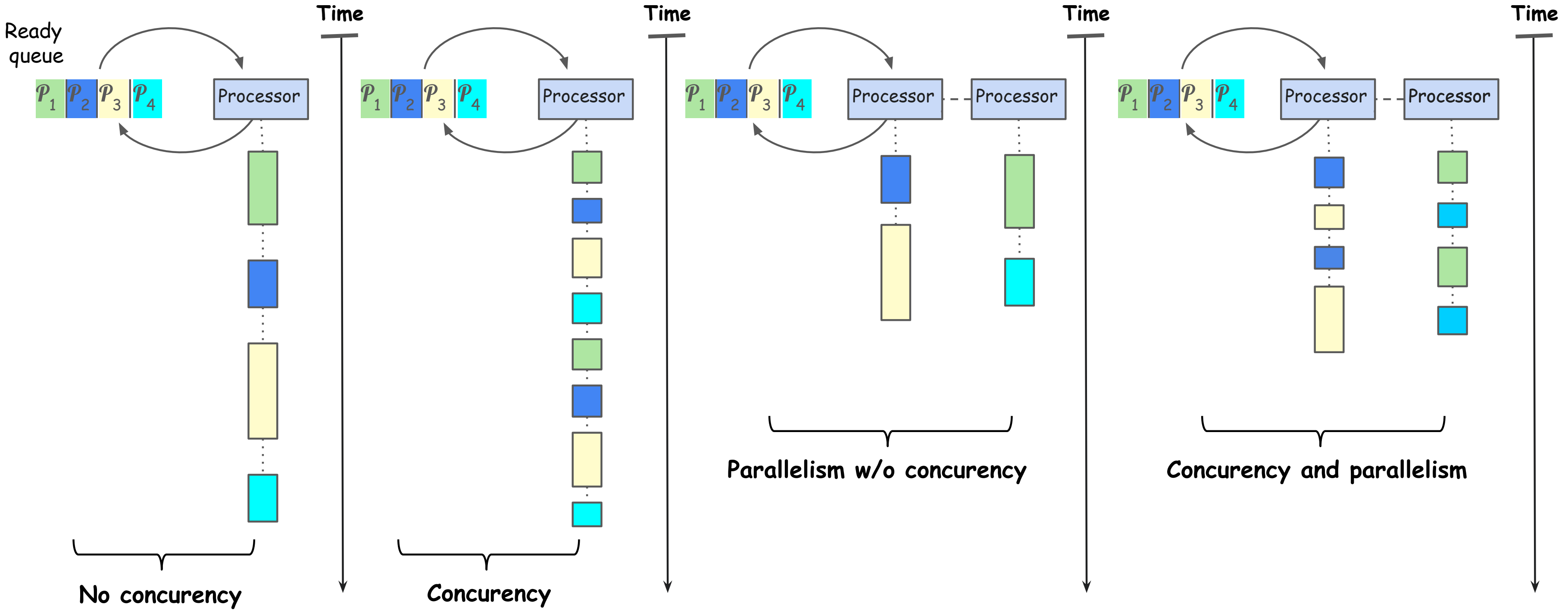


Overview

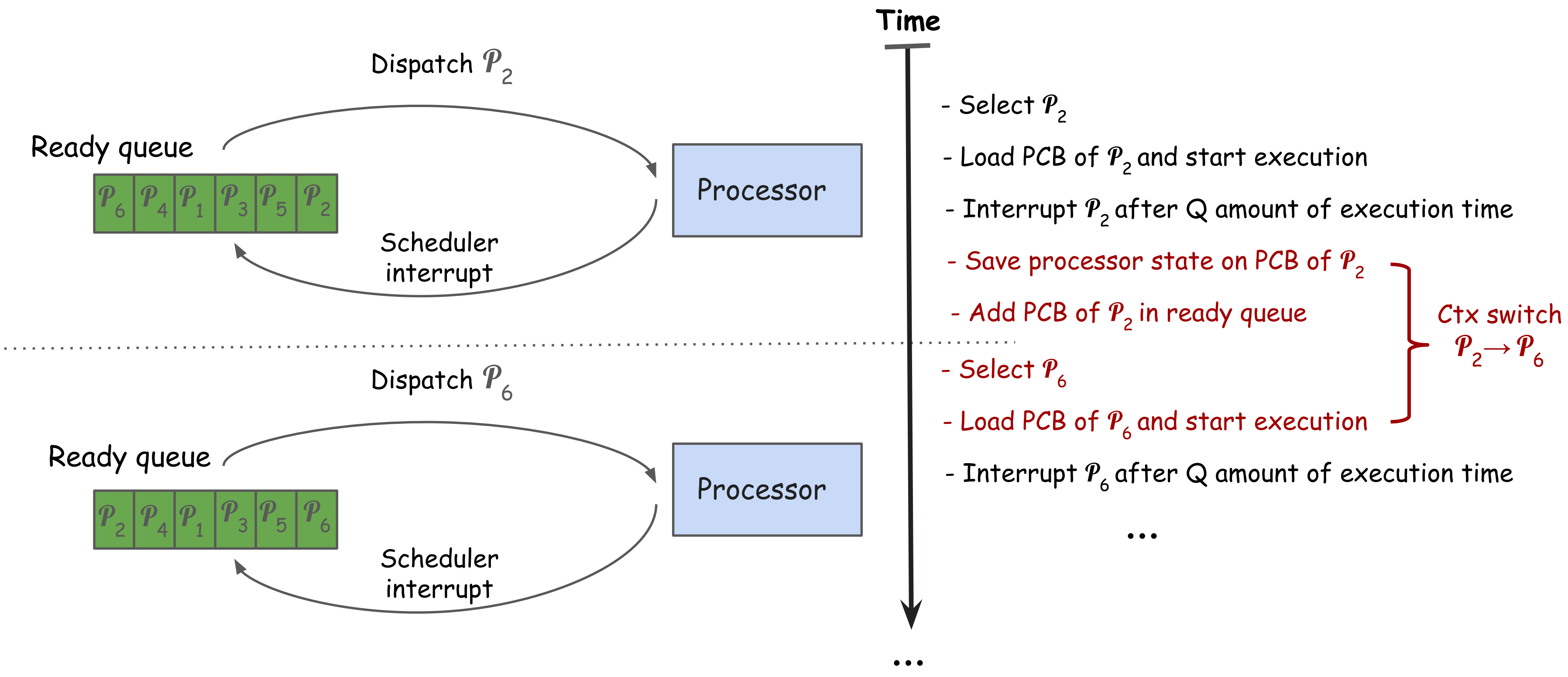
- We'll start from hardware and follow a question-oriented approach
  - ~~Intro [Q: What is an OS?]~~
  - ~~Events [Q: When does the OS run?]~~
  - ~~Runtime [Q: How does a program look like in memory?]~~
  - ~~Processes [Q: What is a process?]~~
  - ~~IPC [Q: How do processes communicate?]~~
  - **Threads [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

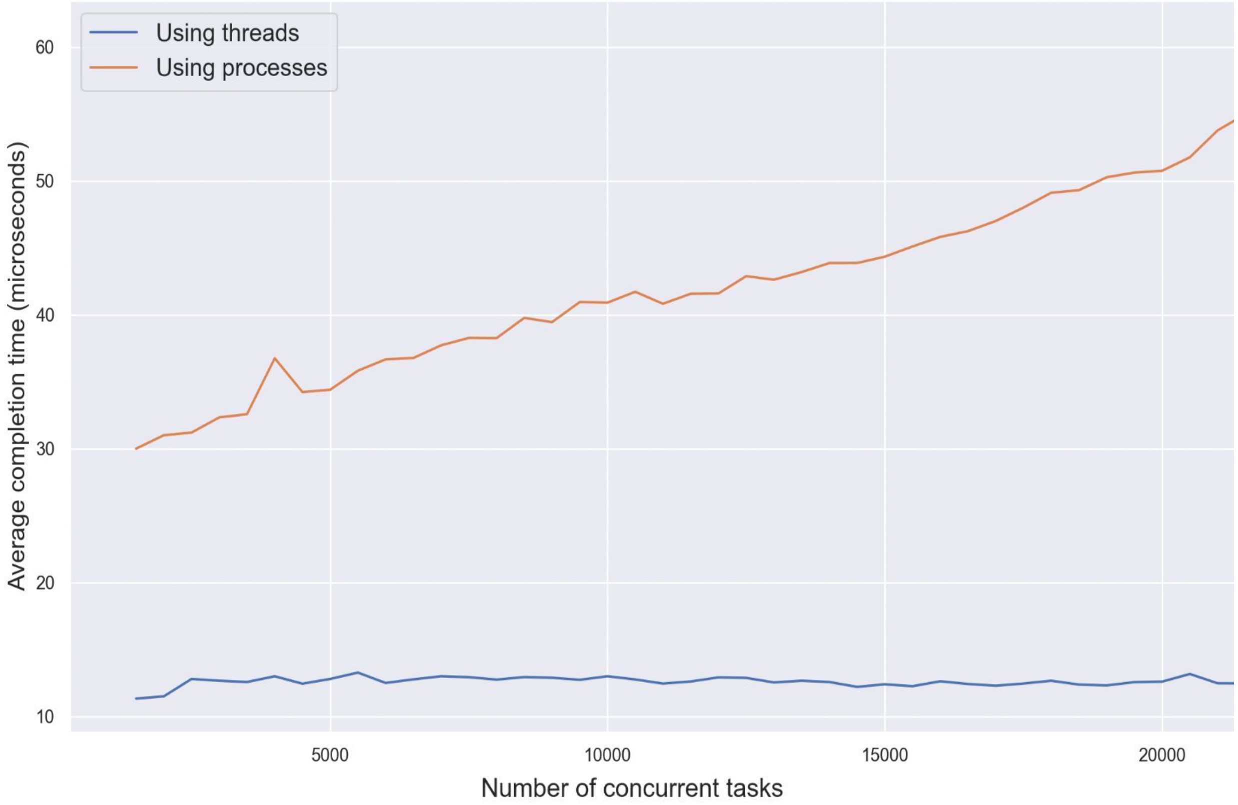
A primer on concurrency



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.)

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

The thread abstraction

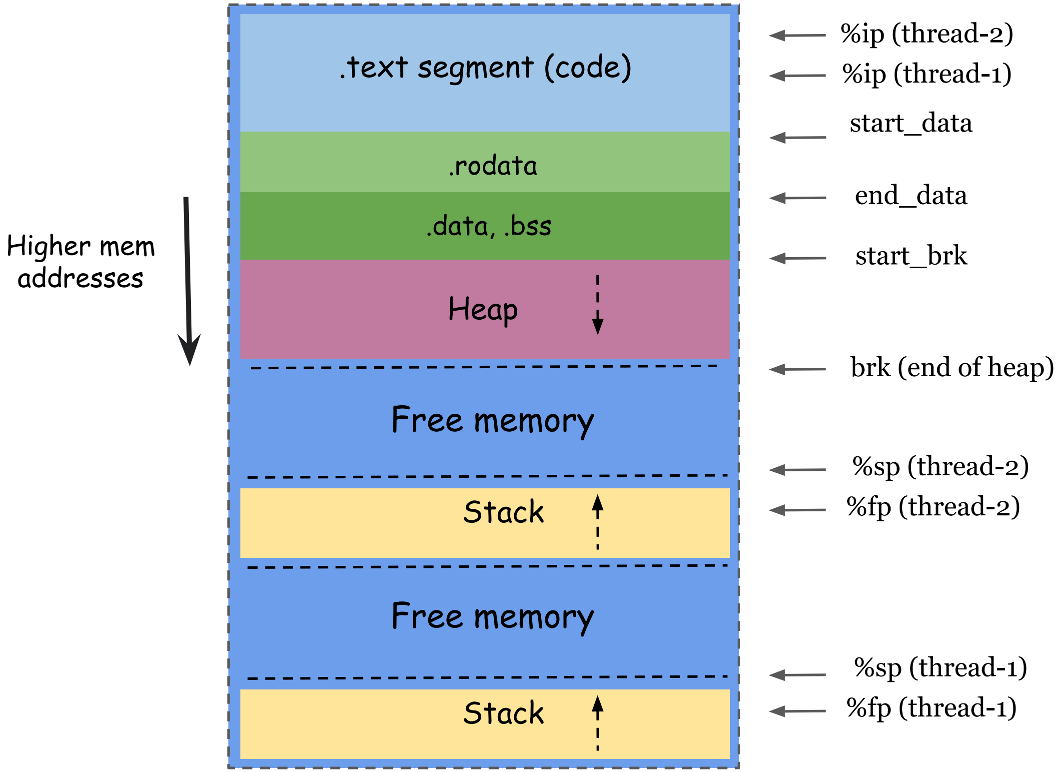
- What is a thread? "A single flow of control within a process." (Strict POSIX definition, 3/190.)

- 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
```