K22 – Operating Systems: Design Principles and Internals

Fall 2026 @dit (α.μ. mod 2 = 1)

Vaggelis Atlidakis

Lecture 08

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

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

- 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

- **Threads**
  - Q1: What is a thread?
  - Q2: Threads vs. processes?
  - Q3: How does the OS implement threads?
  - Q4: From uniprogramming to multiprogramming?
  - Q5: "How to" concurrency?

The thread abstraction

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

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 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

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**

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

- Execution state

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.)

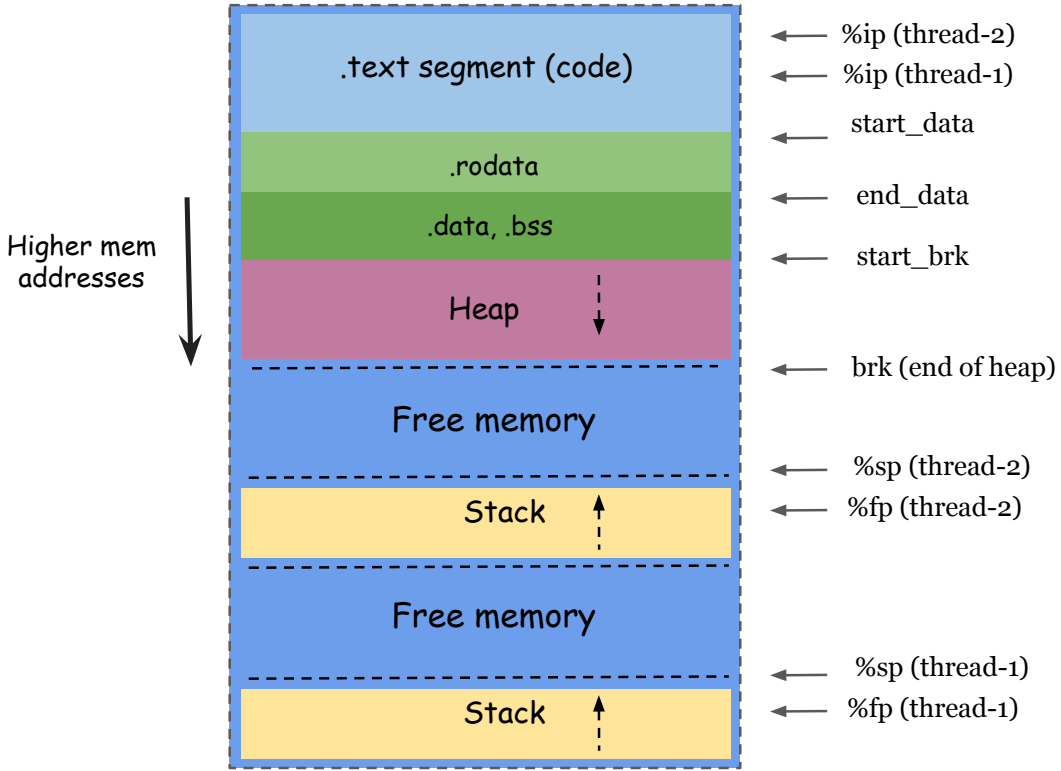
› All threads of a process share

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

› Each thread has its own

- Execution state
- 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.)

> **Unlike processes**

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

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)

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

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 / no context switch

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 / no context switch ⇒ TLB remains hot

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 / no context switch ⇒ TLB remains hot
- Communication b/w threads of the same process is inexpensive

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 / no context switch ⇒ TLB remains hot
- Communication b/w threads of the same process is inexpensive
  - Can be implemented with no OS intervention

The thread abstraction

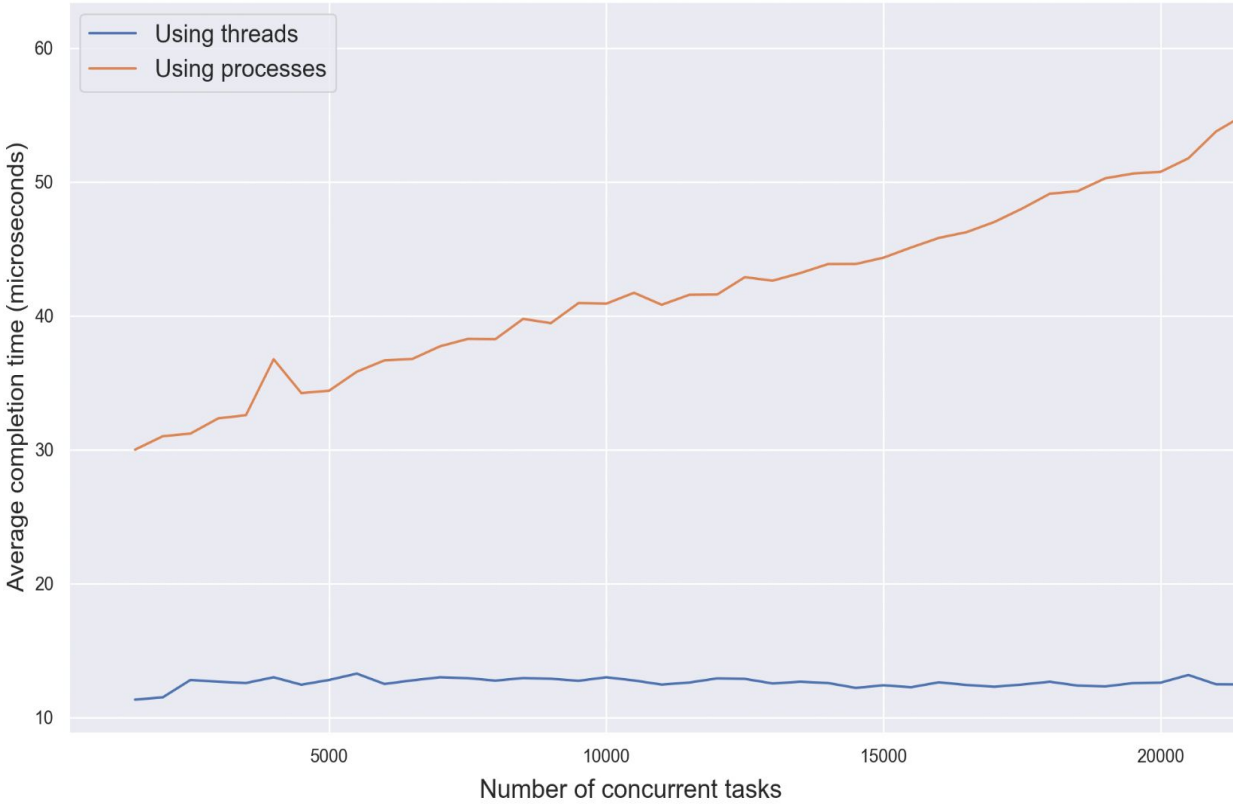
- What is a thread? "A single flow of control within a process." (Strict POSIX definition, 3/190.)

- 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 / no 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

How does the OS implement threads

- POSIX-compliant OSes need to manage both threads and processes
- Easy to support threads, if already supporting processes

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

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

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)

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 P from ready queue
   Change to the new address space
     Execute P until time Q expires
       Put  P back in ready queue
} while(1)
```

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

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

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)

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

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

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

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

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)

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)

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)

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

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

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

Historical Evolution: From uniprogramming to multitasking

- Uniprogramming: Load a program in memory and execute it to completion

Historical Evolution: From uniprogramming to multitasking

- Uniprogramming: Load a program in memory and execute it to completion
  - Human operator acts as the dispatcher

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
    }

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

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

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)

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

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)

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

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

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)

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

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

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

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

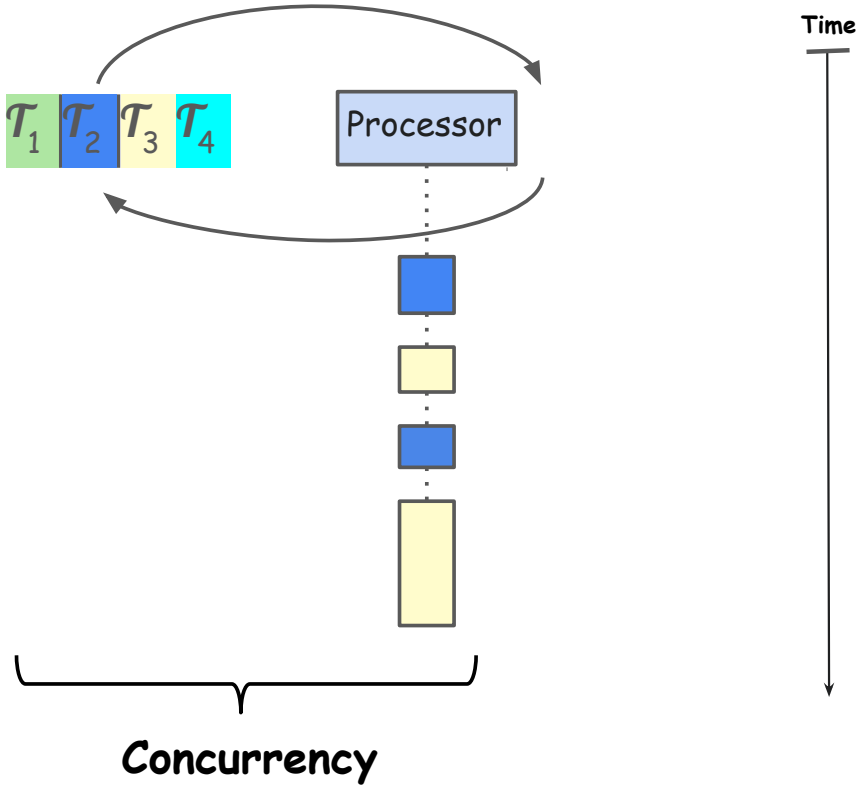
Historical Evolution: From uniprogramming to multitasking

- 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

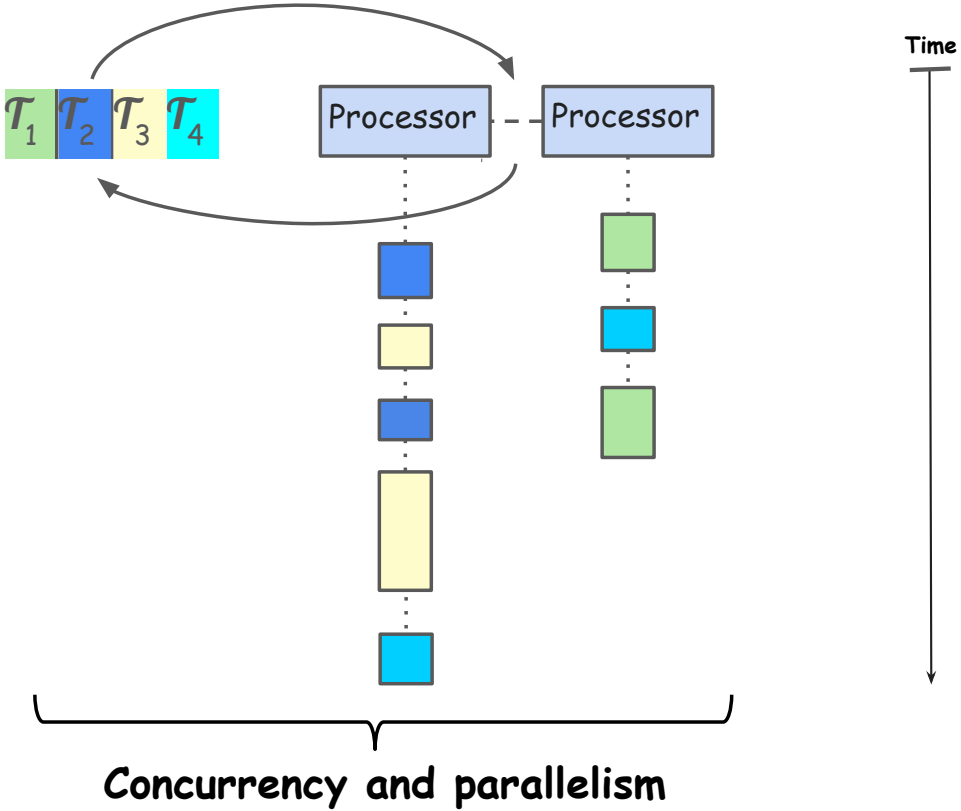
› Improves processor utilization ⇒ Better throughput

› Improves system responsiveness ⇒ Immediate feedback to users

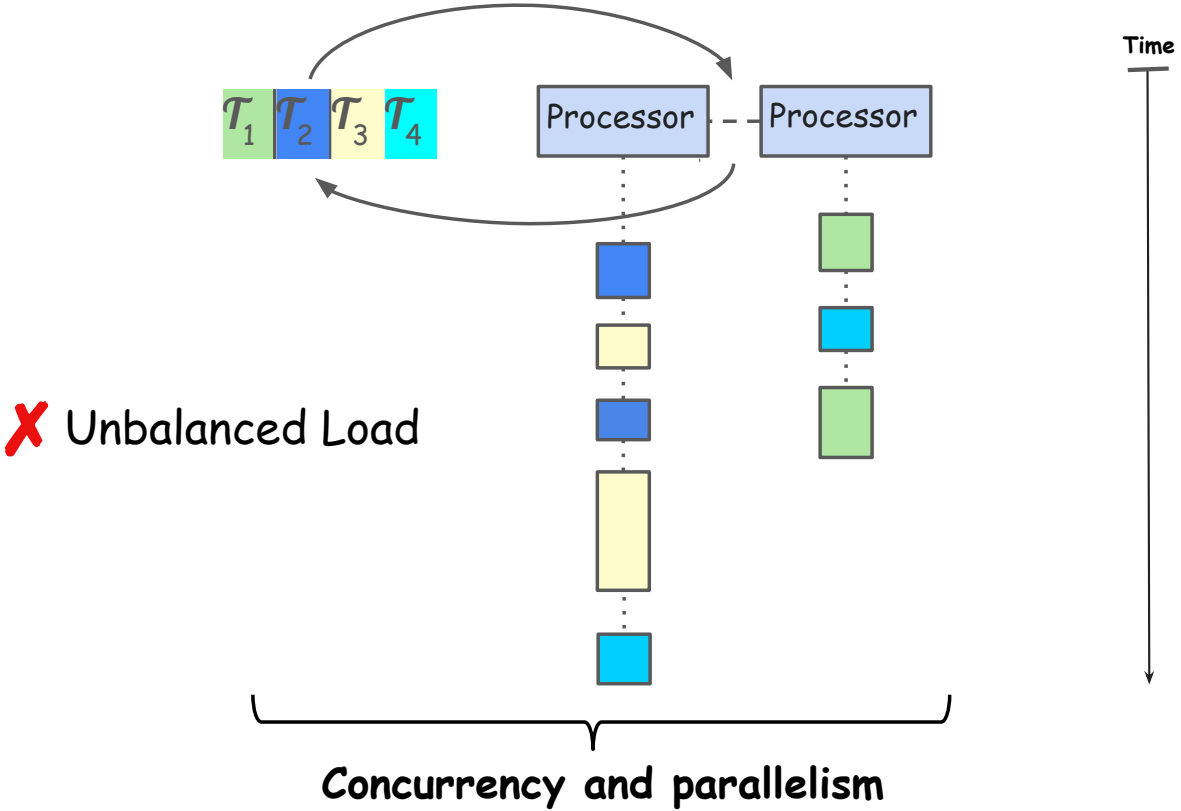
Multitasking on one processor



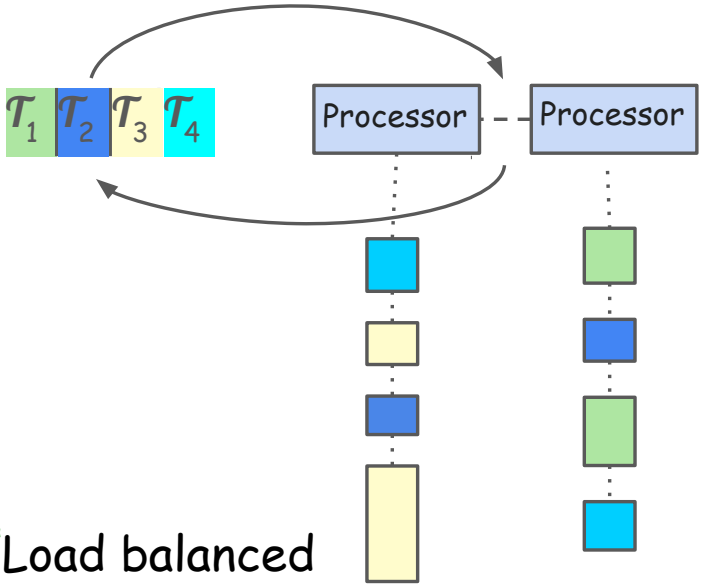
Multitasking on Symmetric Multiprocessor (SMP)



Multitasking on Symmetric Multiprocessor (SMP)



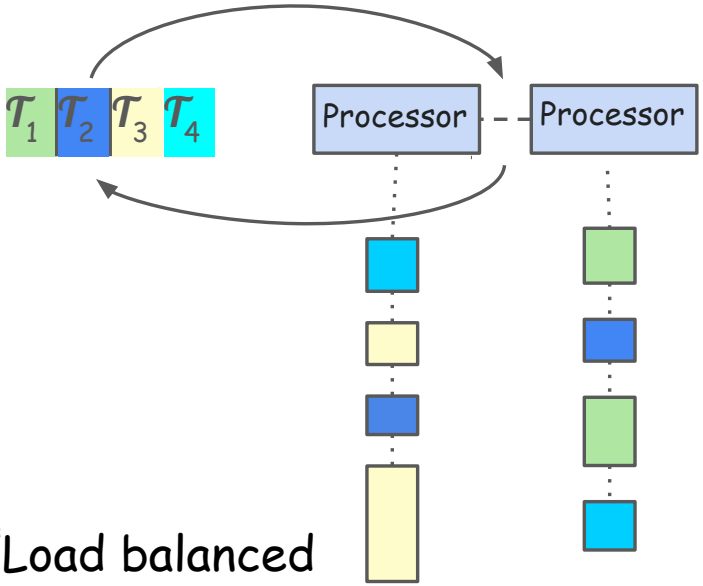
Multitasking on SMP and load balancing



✔Load balanced

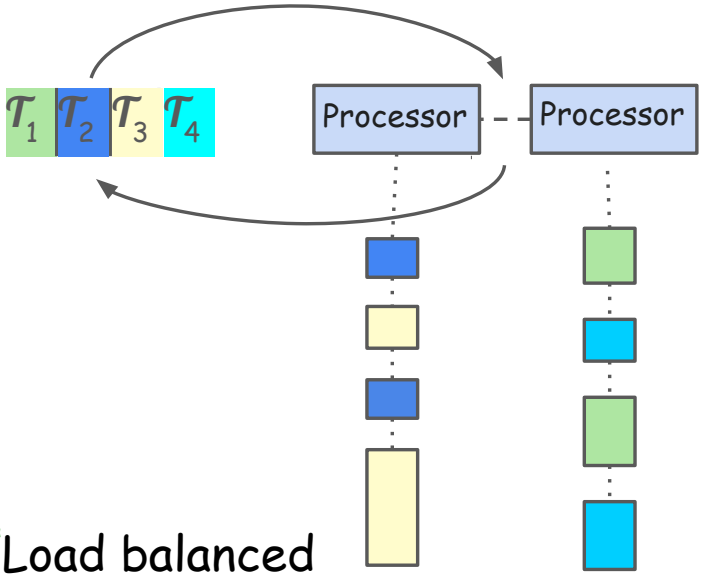
Multitasking on SMP and load balancing

Problem?



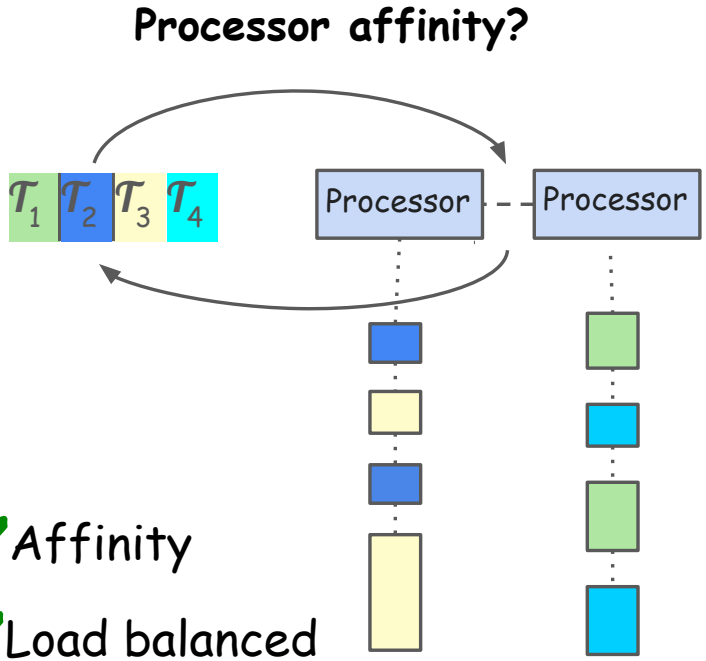
✔Load balanced

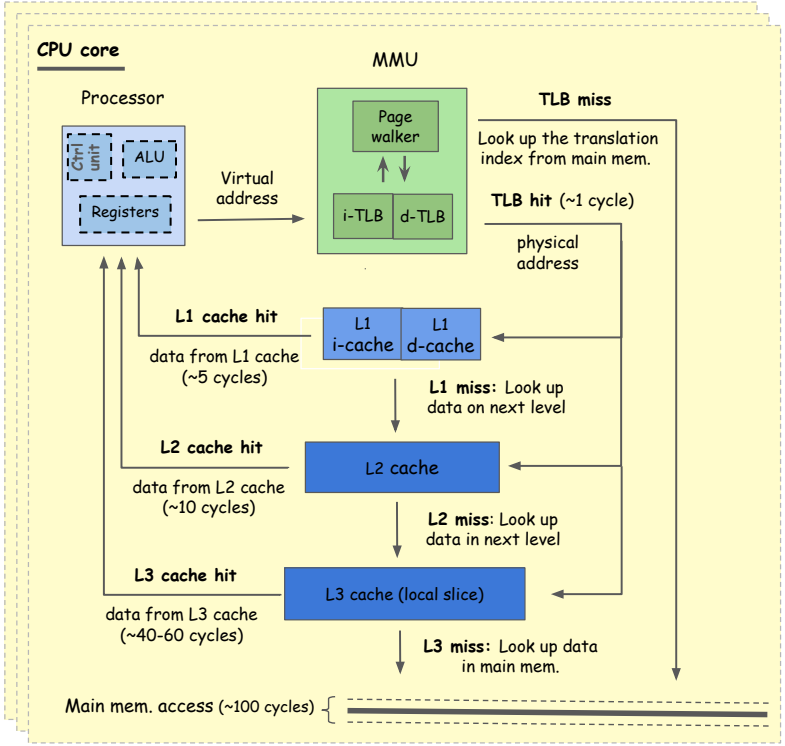
Multitasking on SMP and load balancing w/ affinity



✔Load balanced

Multitasking on SMP and load balancing w/ affinity





Implementing OS support for concurrency

Implementing OS support for concurrency

- Concurrency *(distributed systems concept)*: Multiple operations run seemingly simultaneously on shared hardware resources

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)

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 ♡

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

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