Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 2: Processes

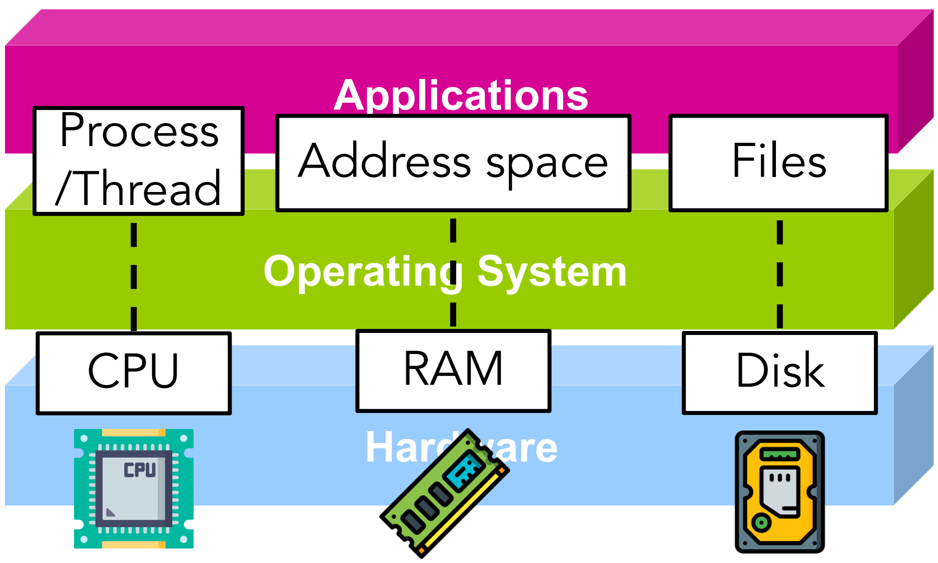
Prof. Ryan Huang

Administrivia

498-02* office hours not start until project 2

OS abstractions

Recap: OS provides abstractions to hide details of hardware from applications

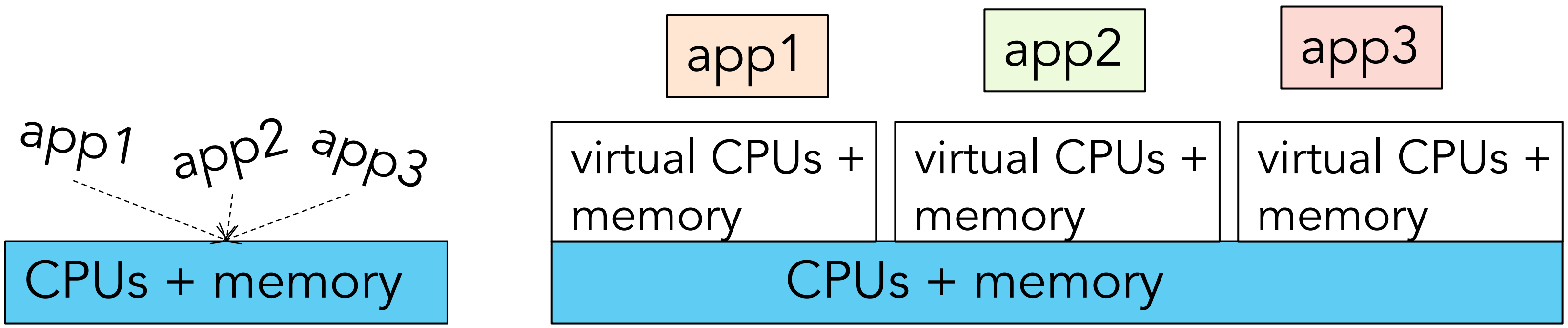


Process abstraction

Process is the OS abstraction for CPU (execution)

- Sometimes also called a job or a task

Decompose mix of activities running on a computer into several independent tasks



A process' view

Each process has its own view of the machine

- Its own virtual memory
  - 0xc000 means different thing in P1 & P2
- Its own virtual CPU
- Its own open files

Simplifies programming model

- gcc does not need to care that firefox is running

What is a process?

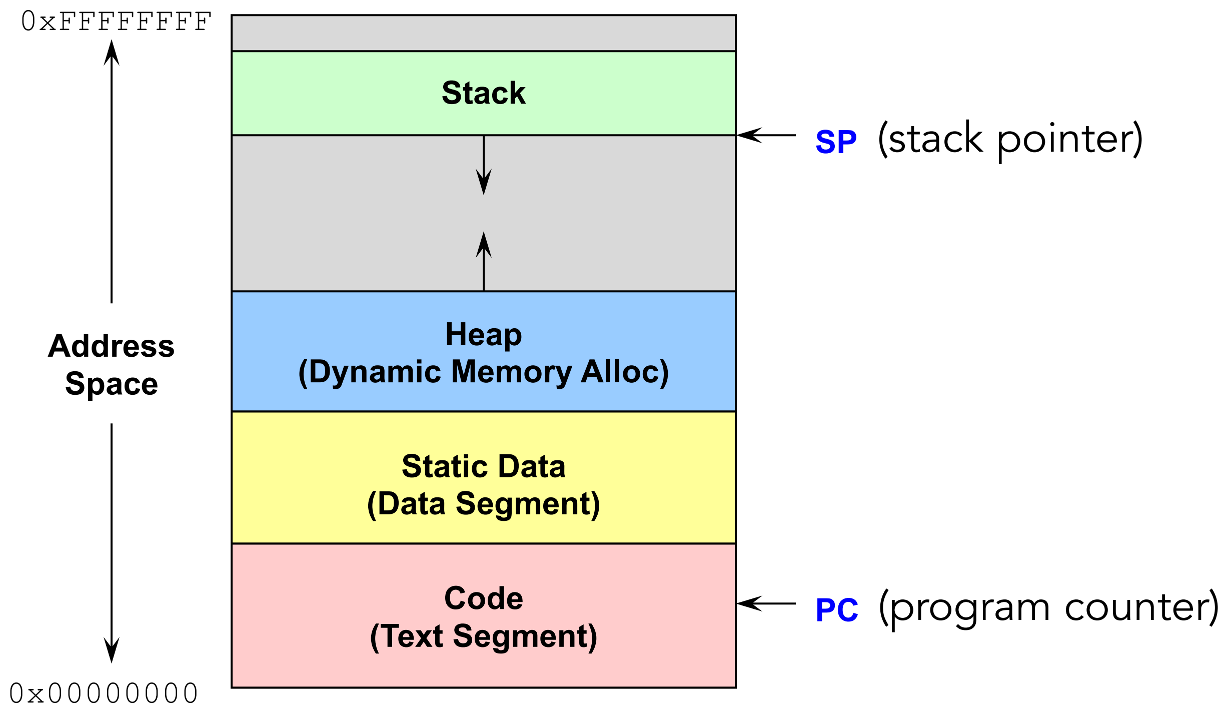
A process is a program in execution

- Programs are static entities with the potential for execution

It contains all state needed for program execution

- An address space (memory)
    - The code for the executing program
    - The data for the executing program
    - An execution stack encapsulating the state of procedure calls
- The program counter (PC) indicating the next instruction
- A set of general-purpose registers with current values
- A set of OS resources
    - Open files, network connections, etc.

Process address space



All addresses a process can *possibly* use

- Note: the addresses are virtual

Provides isolation

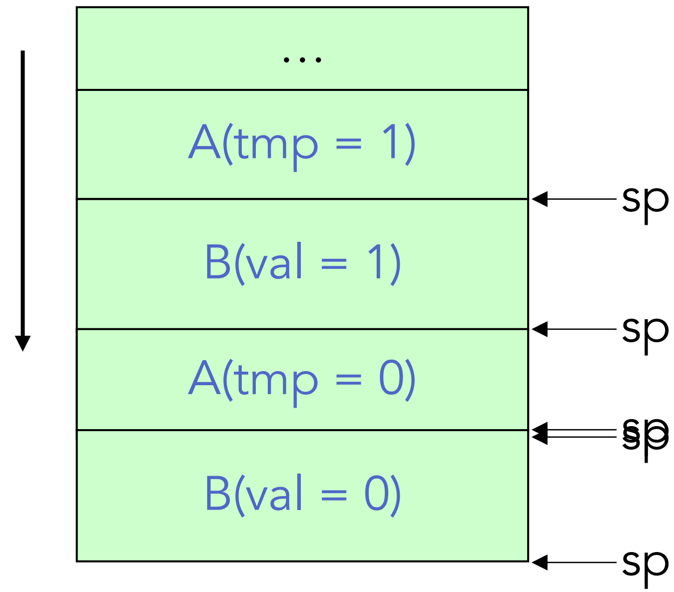
- Cannot be accessed by another process without permission

Review: the call stack

```
A(int tmp) {
    B(tmp);
}

B(int val) {
    C(val, val + 2);
    A(val - 1);
}

C(int foo, int bar) {
    int v = bar - foo;
}
```



A stack frame is created for each function invocation and is destroyed after the function call finishes

Thread

Modern OSes separate the concepts of processes and threads

Thread defines a sequence of execution stream

- PC, SP, registers

Process defines the address space and general process attributes (process ID, open files, etc.)

A thread is bound to a *single* process

But a process can have multiple threads

- Sometimes they interact
- Sometimes they work independently

Per-thread state vs. shared state

Type of state

- Registers (e.g., `eax`, `ebx`, ...)
- Code
  - + program counter (e.g., `eip`)
- Stack
  - + stack pointer (e.g., `esp`)
- Data segment (heap + static variables)

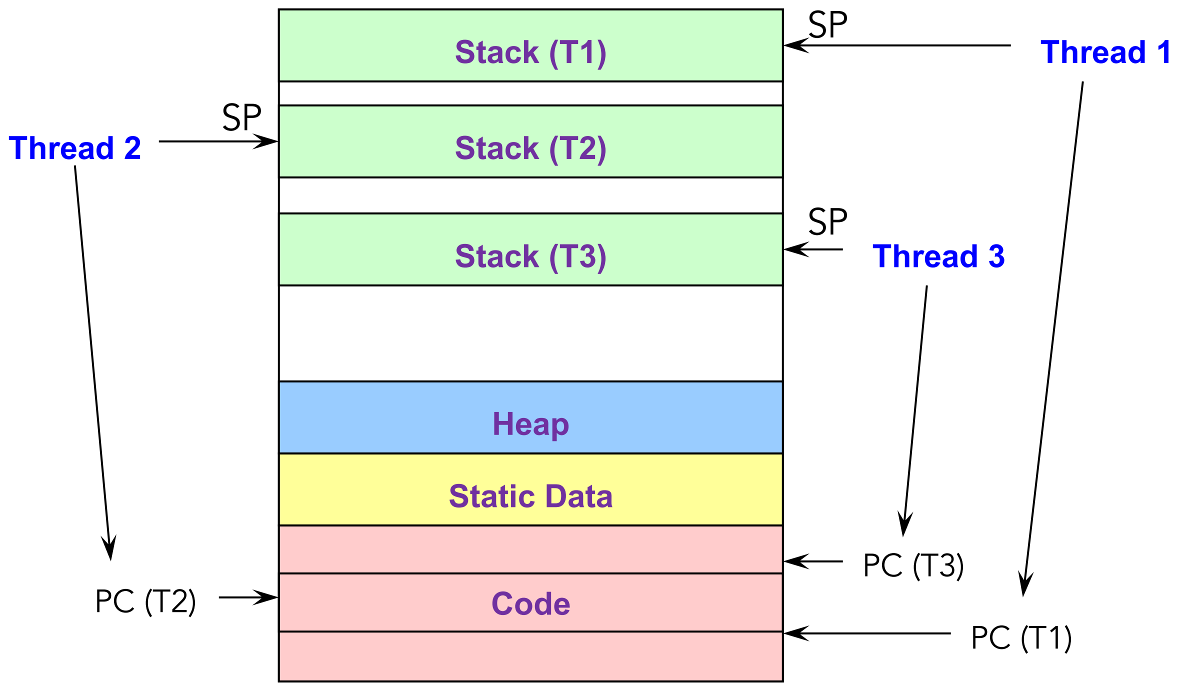
Per-thread state: which state is needed by (or unique to) each thread as it executes instructions?

- Stack, Registers, PC, SP

Other state is shared:

- Code and data segment

Threads in a process



If thread 1 does `p=malloc(..);` and passes `p` to thread 2, can thread 2 use `p`?

Upcoming topics

Thread: unit of concurrency

- How multiple threads cooperate to accomplish a task
- How multiple threads can share limited number of CPUs

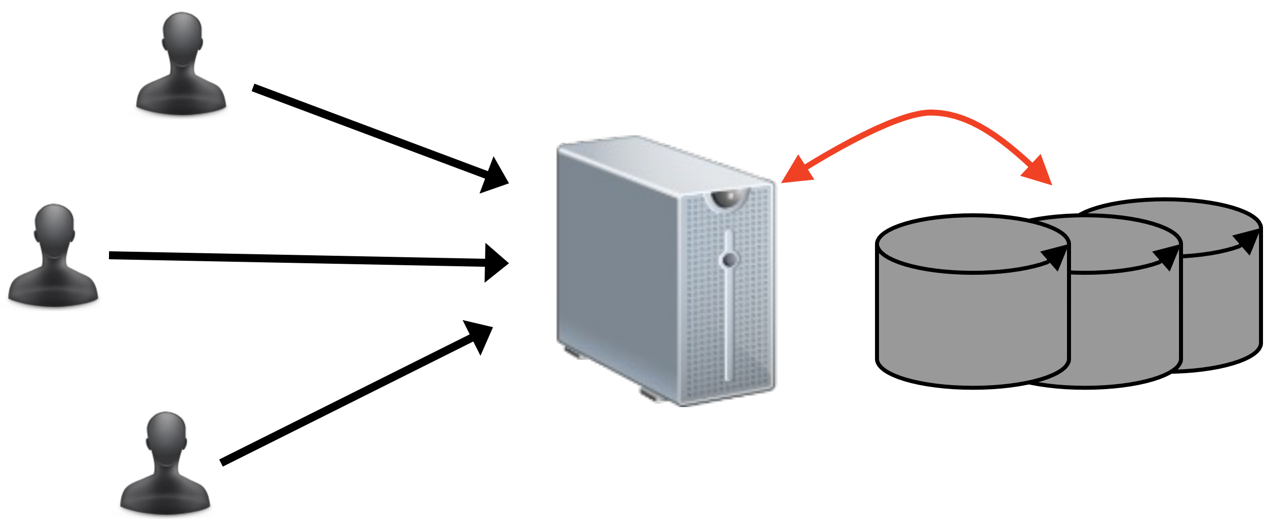
Address space: unit of state partitioning

- How do address spaces share single physical memory?
  - Efficiently
  - Flexibly
  - Safely

Why do we need threads?

Example: a web server application

- Receives multiple simultaneous requests
- Reads web pages from disk to satisfy each request



Option 1: Handle one request at a time

Request 1 arrives

Server receives request 1

Server starts disk I/O 1a and waits for it to complete..

Request 2 arrives

Server must finish request 1 before handling request 2

time

Easy to program, but slow

- Can't overlap disk requests with computation
- Can't overlap either with network sends and receives

Option 2: Asynchronous I/O (event-driven)

Issue I/Os, but don't wait for them to complete

Request 1 arrives
Server receives request 1
Server starts disk I/O 1a (but does not wait for it to complete)
Request 2 arrives
Server receives request 2
Server starts disk I/O 2a
Request 3 arrives
Disk I/O 1a finishes
Continue to process request 1

time

Fast, but difficult to program

- Lots of extra state to remember at each point
- What requests are being served, what stage they're in
- What disk I/Os are outstanding; which requests they belong to

Option 3: Multi-threaded web server

One thread per request

- Thread issues disk (or network) I/O, then waits for it to finish
- Though thread is blocked on I/O, other threads can run
- Where is the state of each request stored?

Thread 1	**Thread 2**	**Thread 3**
Request 1 arrives Receive request 1 Start disk I/O 1a		
	Request 2 arrives Receive request 2 Start disk I/O 2a	
		Request 3 arrives Receive request 3
Disk I/O 1a finishes Continue processing request 1		

Benefits of threads

Allow multiple things to happen in parallel

Lightweight to create compared to processes

- Mostly allocating a stack

Thread manager takes care of CPU sharing

- One thread can issue blocking I/O, while other threads can still progress
- Private state for each thread

Applications get a "simpler" programming model

- The illusion of a dedicated CPU per thread

Downsides of threads

Performance degradation under high concurrency

- Costs of context switching, lock contention, cache misses, scheduling, etc.
- Event-driven model can achieve higher performance
  - Used in high-concurrency software in practice, e.g., Nginx

Synchronization is hard!

- More on this later..

Long-standing debate

- *Why threads are a bad idea, Ousterhout, 1995*
- *Why events are a bad idea, von Behren et al., 2003*

When are threads useful?

Multiple things happening at once

Usually some slow resource

- Network, disk, user, …

Examples:

- Network server (e.g., web server)
- Controlling a physical system (e.g., airplane controller)
- Window system
- Parallel programming

Ideal scenario

Split computation into threads

Threads run independently of each other

- Divide and conquer works best if divided parts are independent

How practical is thread independence?

Completely independent threads?

Example 1: Microsoft Word

- One thread formats document
- Another thread spell checks document

Example 2: Desktop computer

- One thread compiles EECS 482 project
- Another thread runs Minecraft

Dependence among threads is often inevitable

- i.e., threads are often *cooperating*

Two types of sharing: application resource or hardware resource

Non-determinism

Problem:

- Speed of each processor is unpredictable
- Ordering of events across threads is non-deterministic

Thread A - >

Thread B -   -   -   -   -   -   -   -   -   -   - >

Thread C --->

Consequences:

- Many possible global ordering of events
  - Also known as *thread interleaving*
- Different orderings may produce different results

Non-deterministic ordering → Non-deterministic results

Printing example

Thread 1	Thread 2
`print "ABC"`	`print "123"`

What's being shared between these threads?

Possible outputs:

Impossible outputs:

Ordering within thread is sequential

Many ways to merge per-thread order into a global order

Non-deterministic ordering → Non-deterministic results

Assignment example (x is initially 0)

Thread A	Thread B
`x = 1`	`x = -1`

- What's being shared between these threads?
- Possible results:
- Impossible results:

Non-deterministic ordering → Non-deterministic results

Arithmetic example (y is initially 10)

Thread A	Thread B
`x = y + 1`	`y = y * 2`

- What's being shared between these threads?
- Possible results:

Non-deterministic ordering → Non-deterministic results

Assignment example (x is initially 0)

<u>Thread A</u>

```
x = 1
```

```
x = -100
x += 101
```

<u>Thread B</u>

```
x = -1
```

```
x = -100
x += 99
```

- Possible results:

- Impossible results:

Atomic operations

Before we can reason at all about cooperating threads, we must know that some operation is **atomic**

- Indivisible, i.e., happens in its entirety or not at all
- No events from other threads can occur in between start and end of an atomic operation

Thread 1: `print "ABC"` Thread 2: `print "123"`

- What if each print statement were atomic?
- What if printing a single character were **not** atomic?

Need an atomic operation to build a bigger atomic operation

Set of atomic instructions varies by ISA

- E.g., memory load is atomic; memory store is atomic; many other instructions are not atomic (e.g., double-precision floating point)

Example

Thread A

```
i=0
while (i < 10) {
      i++
}
print "A finished"
```

```
mov (i), %ebx
inc %ebx
mov %ebx,(i)
```

Thread B

```
i=0
while (i > -10) {
      i--
}
print "B finished"
```

```
mov (i), %ecx
dec %ecx
mov %ecx,(i)
```

Which thread will finish first?

Is the winner guaranteed to print first?

Is it guaranteed that someone will win?

Say both threads run at exactly the same speed, and start close together

- Is it guaranteed that both threads will loop forever?

Do you really need to worry about such far-fetched scenarios?

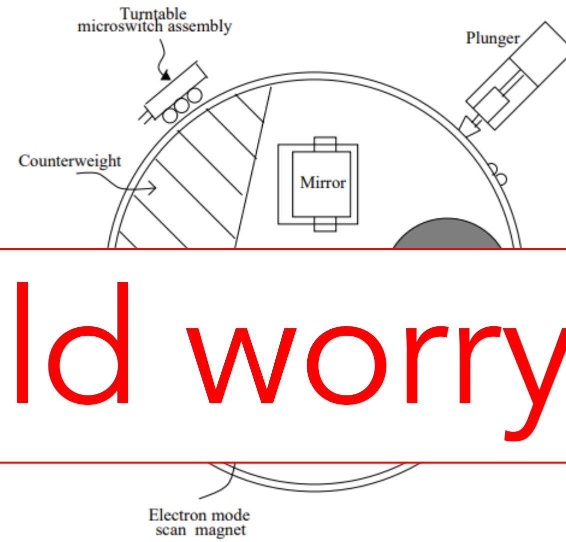


Figure 1: Upper turntable assembly.

Yes, you should worry!

Northeast blackout of 2003

Therac-25 radiation therapy [Leveson95]

Non-deterministic interleaving makes debugging challenging

- **Heisenbug: a bug that occurs non-deterministically**

Writing correct concurrent programs

Consider and control all possible interleavings of events from multiple threads

- All possible interleavings of atomic operations must result in correct output
- Non-atomic operations to a shared resource must not occur concurrently

Controlling how events from different threads can interleave is called synchronization

Synchronization

Goals

- Eliminate interleavings: all possible interleavings must produce a correct result
  - Trivial solution?
- Preserve interleavings: constrain thread interleavings as little as possible

Concurrency only matters for events that access some shared resource

A correct concurrent program should work no matter how fast the CPUs execute the threads