

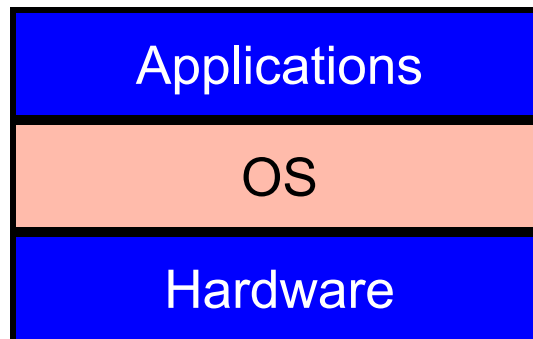
ECE 1175
Embedded Systems Design
Operating Systems - I

Wei Gao

What is an Operating System?

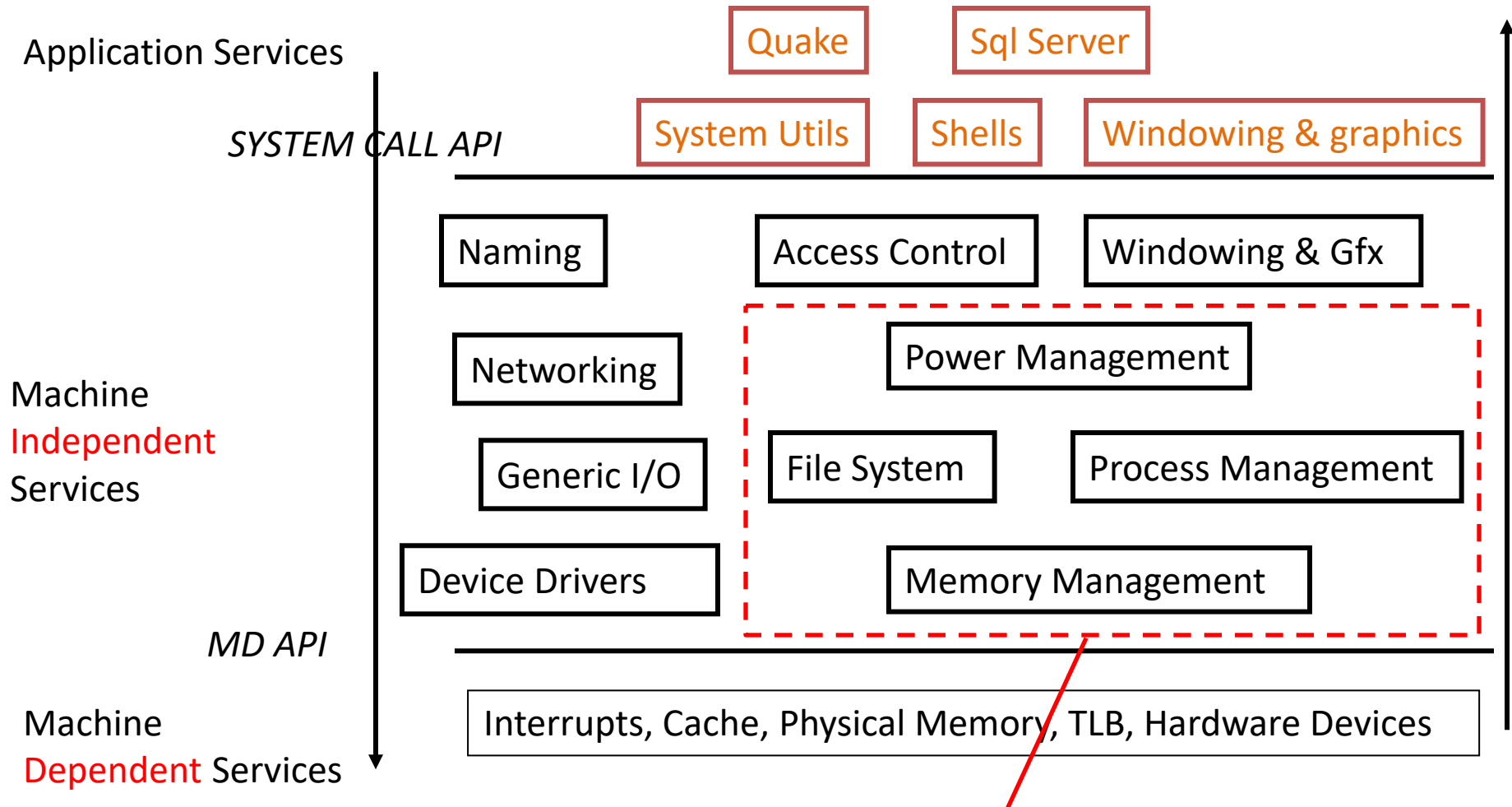
- Operating System

- A **software** layer to **abstract away** and **manage details** of hardware resources
- A set of utilities to **simplify application development**



- “all the code you didn’t write” in order to implement your application

Logical OS Structure



Crucial to Embedded Systems!

Key OS Issues

■ Multi-Programming

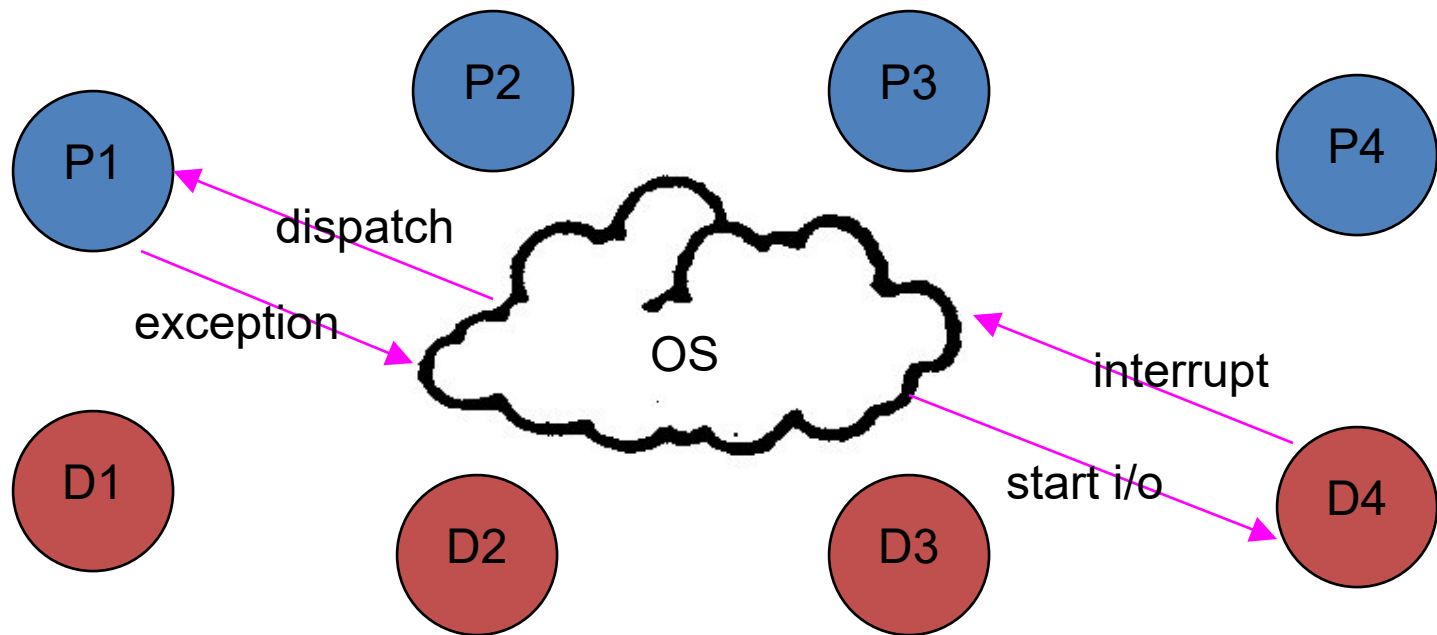
- keeps multiple runnable jobs loaded in memory at once
- overlaps I/O of a job with computing of another
 - while one job waits for I/O completion, OS runs instructions from another job
- goal: **optimize system throughput**
 - perhaps at the cost of response time...

■ Timesharing

- multiple terminals into one machine
- each user has illusion of entire machine to him/herself
- **optimize response time**, perhaps at the cost of throughput
- Timeslicing
 - divide CPU equally among the users

How OS Works?

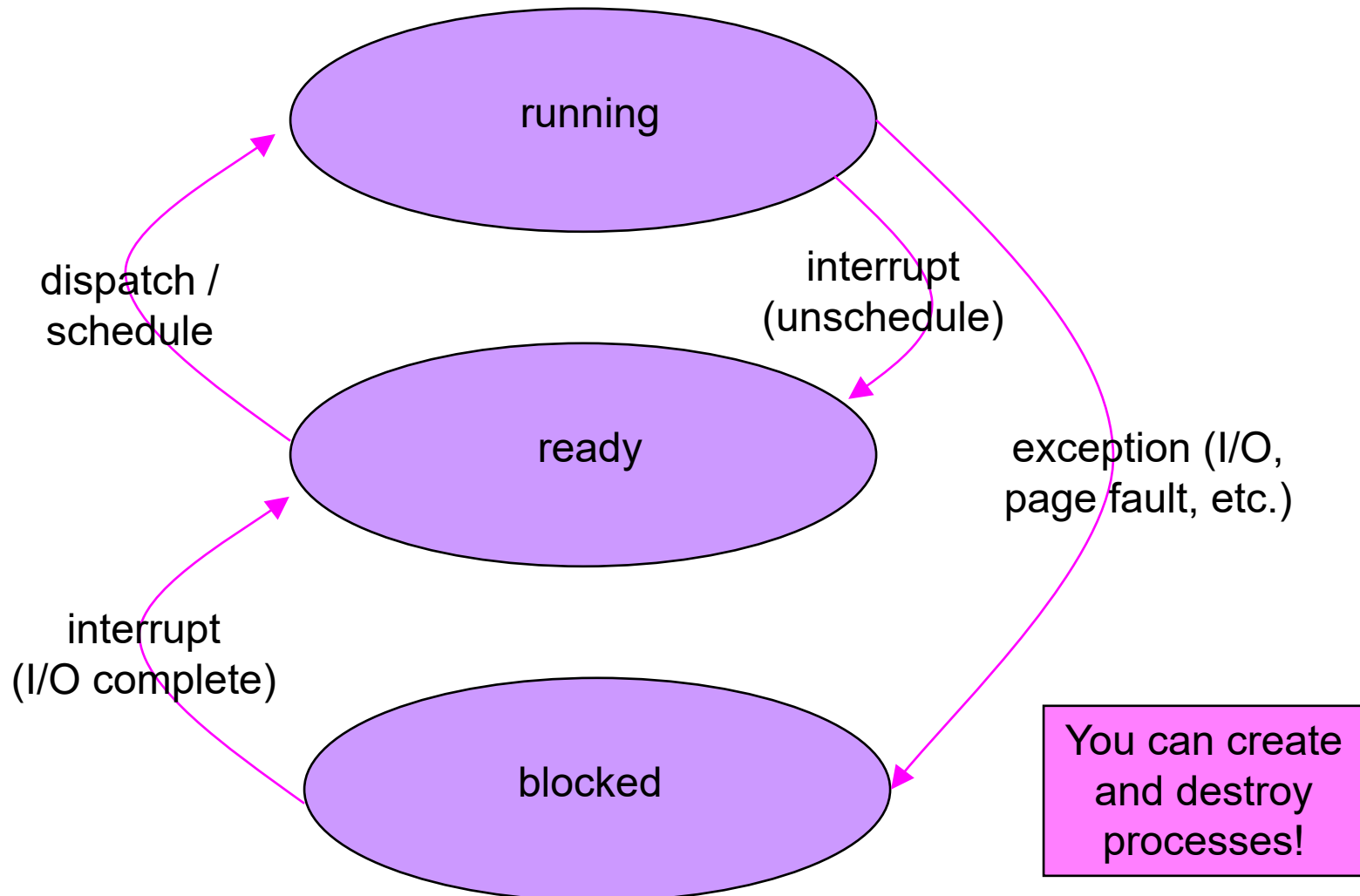
- The OS sits between application programs and the hardware
 - it mediates access and abstracts away ugliness
 - programs request services via exceptions (**traps** or **faults**)
 - devices request attention via **interrupts**



Basic Application Unit: Process

- A process is the application's
 - Unit of execution
 - Unit of scheduling
 - Unit of ownership
 - Dynamic (active) execution context
- Process Control Block (PCB)
 - Identified by an integer process ID (PID)
 - Keeps all of a process's hardware execution states
 - ready: waiting to be assigned to CPU
 - running: executing on the CPU
 - waiting: waiting for an event, e.g., I/O

A Process's Lifecycle



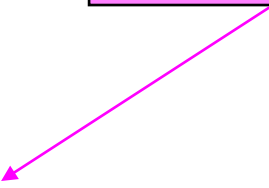
PCB Control

- Running state:
 - Hardware states are in the CPU
- Ready & blocked states:
 - PCB stores the hardware states from registers
- Context switch = PCB reloading

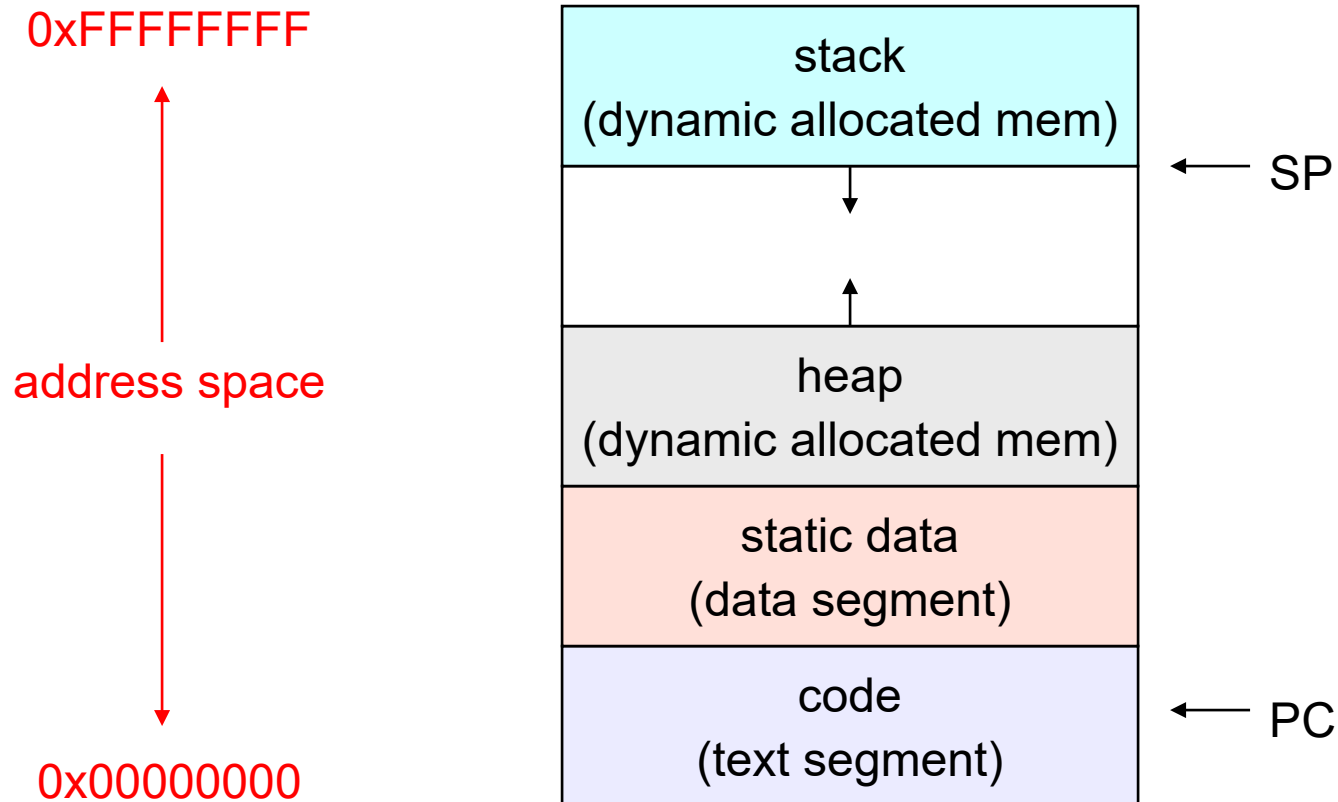
PCB Control

Process ID
Pointer to parent
List of children
Process state
Pointer to address space descriptor
Program count stack pointer (all) register values
uid (user id) gid (group id) euid (effective user id)
Open file list
Scheduling priority
Accounting info
Pointers for state queues
Exit ("return") code value

This is (a simplification of) what each of those PCBs looks like inside!



A Process's Address Space



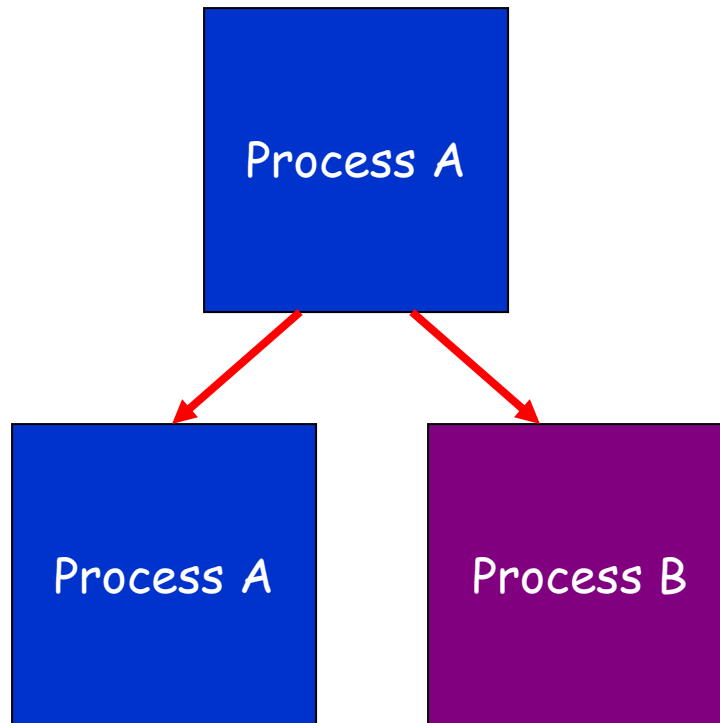
Process Creation

- New processes are created by existing processes
 - creator is called the **parent**
 - created process is called the **child**
 - what creates the first process, and when?
- In some systems, parent defines or donates resources and privileges for its children
 - UNIX: child inherits parent's uid, environment, open file list, etc.
- When child is created, parent may either wait for it to finish, or may continue in parallel, or both!

Process Creation

Create a process with fork:

- parent process keeps executing the old program;
- child process executes a new program.



Create a New Process using fork()

- A process can create a child process by making a **copy** of itself
- Parent process is returned with the child process ID
- Child process gets a return value of 0

```
child_id = fork();  
if (child_id == 0) {  
    /* child operations */  
} else {  
    /* parent operations */  
}
```

Overlay a Process using execv()

- Child process usually runs different code
- Use execv() to overlay the code of a process
- Parent uses wait() to wait for child process to finish and then release its memory

```
childid = fork();  
if (childid == 0) {  
    execv("mychild",childargs);  
    exit(0);  
} else {  
    parent_stuff();  
    wait(&csstatus);  
    exit(0);  
}
```

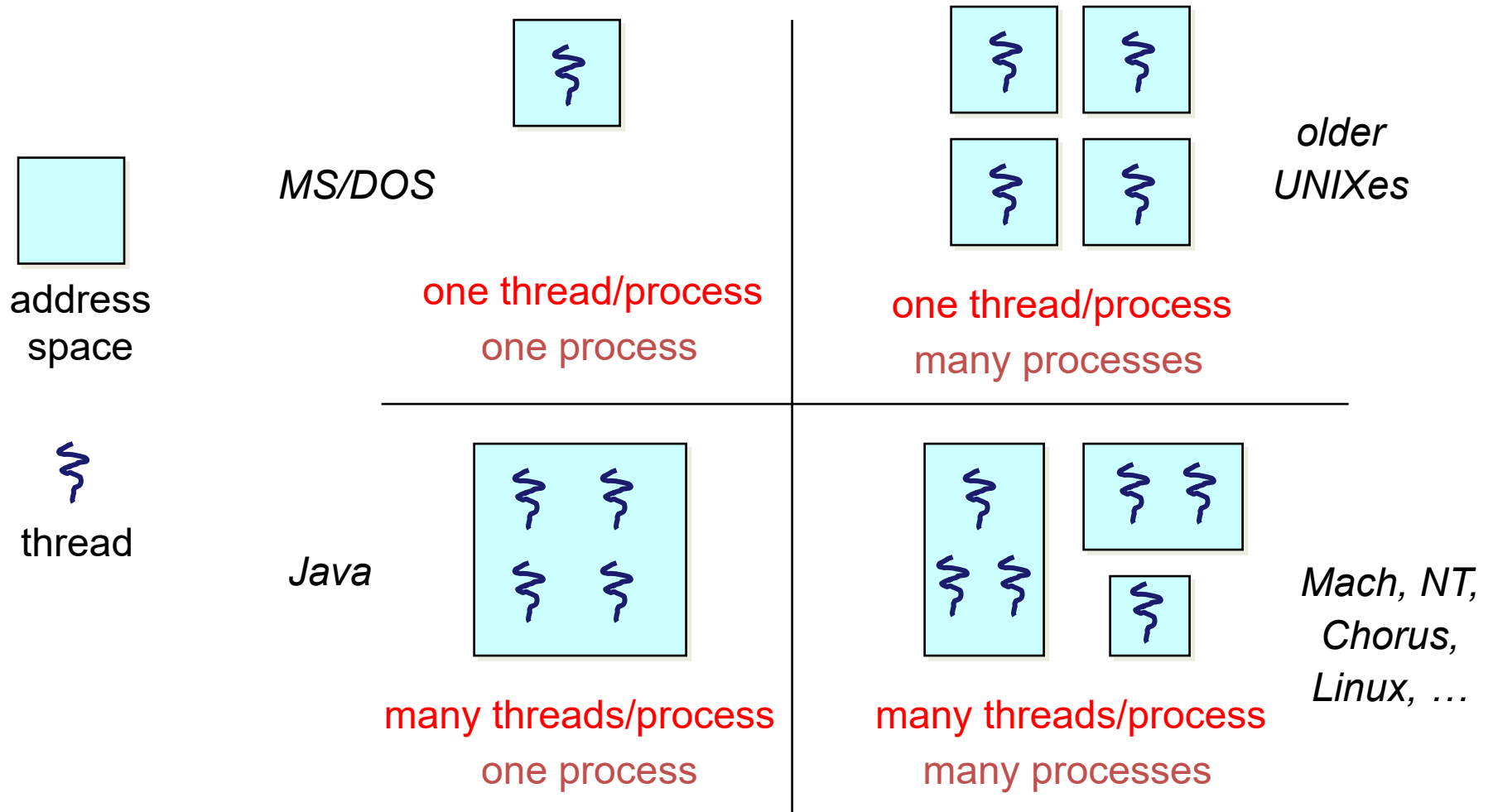
What's in a process?

- A process consists of (at least):
 - an address space
 - the code for the running program
 - the data for the running program
 - an execution stack and stack pointer (SP)
 - traces state of procedure calls made
 - the program counter (PC), indicating the next instruction
 - a set of general-purpose processor registers and their values
 - a set of OS resources
 - open files, network connections, sound channels, ...
- That's a lot!!
- Can we decompose a process?

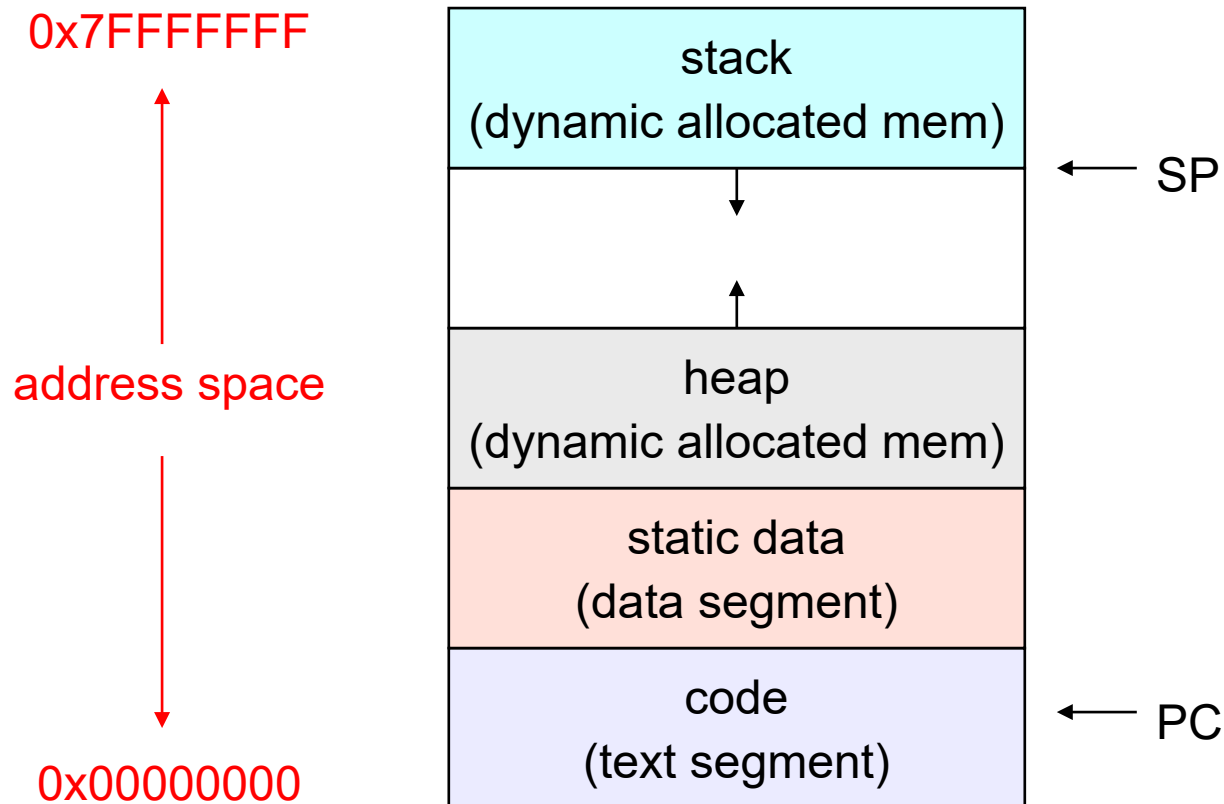
Thread

- A lightweight process
 - Separating the process's memory space
 - Better concurrency!
- Multithreading is useful even on a uniprocessor
 - even though only one thread can run at a time
 - creating concurrency does not require creating new processes
 - “faster / better / cheaper”

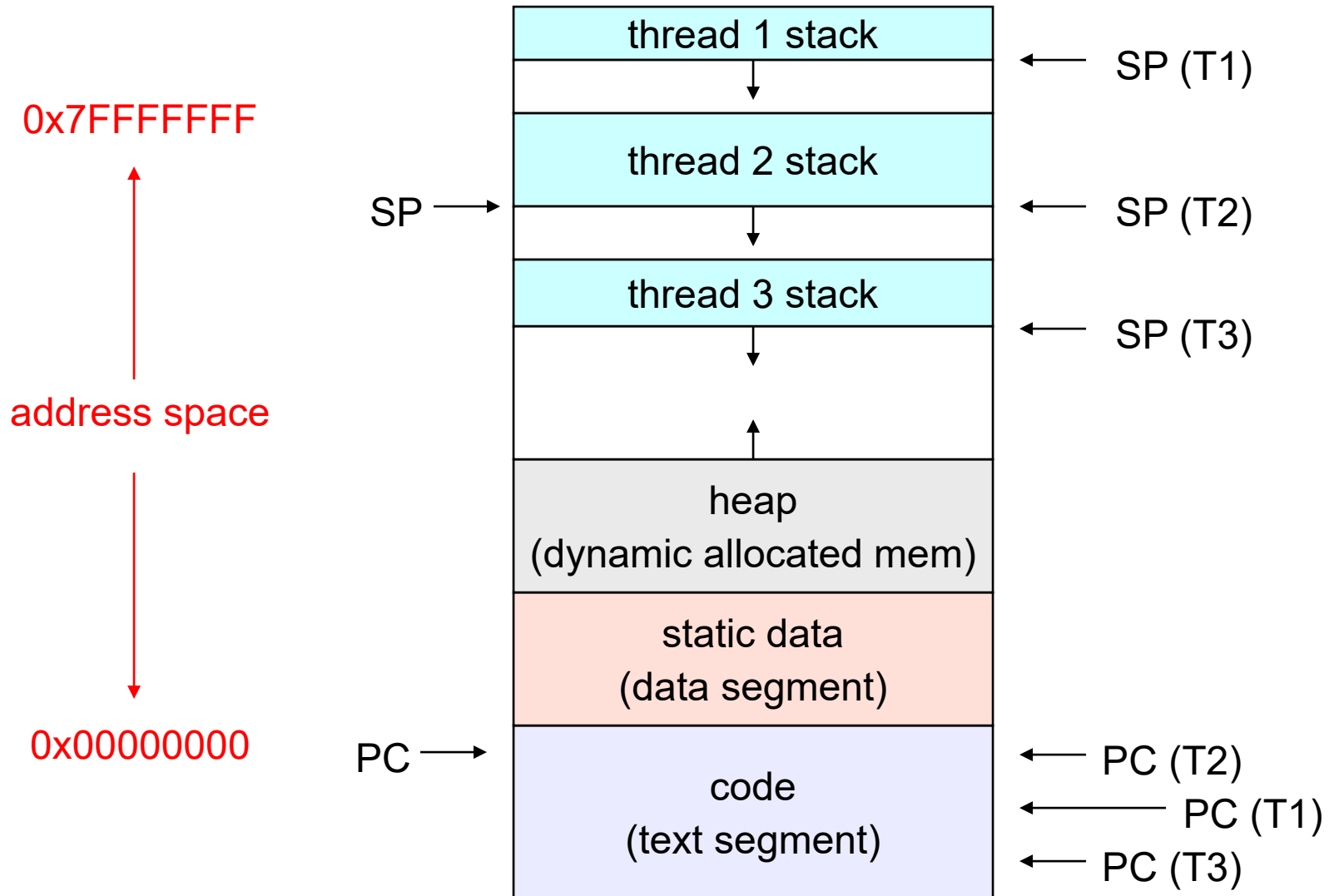
Thread



Process Memory Space



Memory Space with Threads



Where do threads come from?

- Natural answer: the kernel is responsible for creating/managing threads
 - for example, the kernel call to create a new thread would
 - allocate an execution stack within the process address space
 - create and initialize a **Thread Control Block (TCB)**
 - stack pointer, program counter, register values
 - stick it on the ready queue
 - we call these **kernel threads**

Where do threads come from?

- Threads can also be managed at the user level (that is, entirely from within the process)
 - a library linked into the program manages the threads
 - we call these **user-level threads**
- User-level threads are invisible to the OS
 - there is no integration with the OS
- As a result, the OS can make poor decisions
 - scheduling a process with only idle threads
 - blocking a process whose thread initiated I/O, even though the process has other threads that are ready to run
 - unscheduling a process with a thread holding a lock
 - inefficient on multi-processor systems

Thread Interface

- This is taken from the POSIX `pthread`s API:
 - `t = pthread_create(attributes, start_procedure)`
 - creates a new thread of control
 - new thread begins executing at `start_procedure`
 - `pthread_cond_wait(condition_variable)`
 - the calling thread blocks, sometimes called `thread_block()`
 - `pthread_signal(condition_variable)`
 - starts the thread waiting on the condition variable
 - `pthread_exit()`
 - terminates the calling thread
 - `pthread_wait(t)`
 - waits for the named thread to terminate

Performance Example

- On a 700MHz processor running Linux 2.2.16:
 - Processes
 - `fork/exit`: 251 μ s
 - Kernel threads
 - `pthread_create()/pthread_join()`: 94 μ s (2.5x faster)
 - User-level threads
 - `pthread_create()/pthread_join`: 4.5 μ s (another 20x faster)

Managing User-Level Threads

- Strategy 1: force everyone to cooperate
 - a thread willingly gives up the CPU by calling `yield()`
 - `yield()` calls into the scheduler, which context switches to another ready thread
 - what happens if a thread never calls `yield()`?
- Strategy 2: use preemption
 - scheduler requests that a timer interrupt be delivered by the OS periodically
 - usually delivered as a UNIX signal (man signal)
 - signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
 - at each timer interrupt, scheduler gains control and context switches as appropriate

Something to Remember

- Each thread shares everything with all the other threads in the process
 - They can read/write the exact same variables, so they need to synchronize themselves
 - They can access each other's runtime stack, so be very careful if you communicate using runtime stack variables
 - Each thread should be able to retrieve a unique thread id that it can use to access *thread local storage*
- Multi-threading is great, but use it wisely