

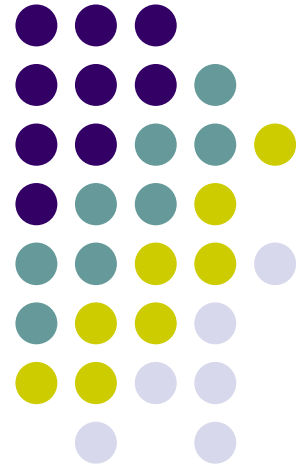
# CSC369H3

---

Operating Systems

Sina Meraji

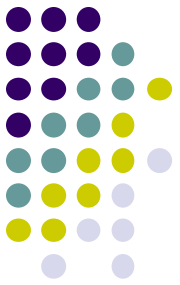
U of T



# Logistics



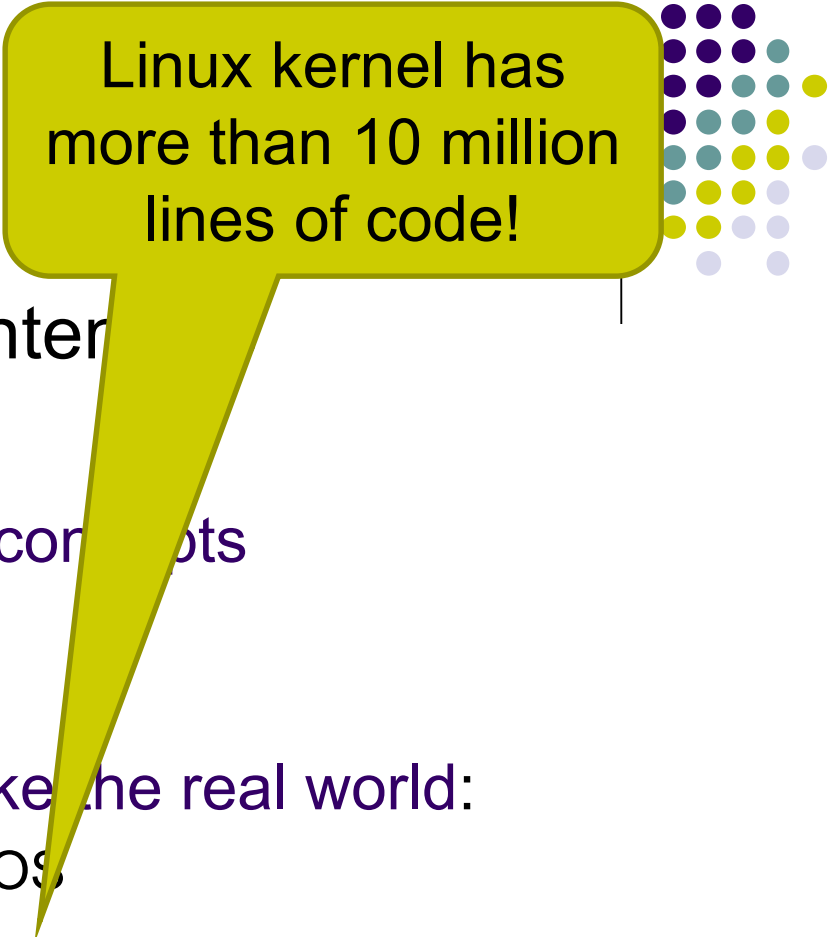
- Instructor: Sina Meraji
  - Email: [sina.mrj@gmail.com](mailto:sina.mrj@gmail.com)
  - Office hours: Tue 17-18 pm(by appointment)
- TAs:
  - Kelechukwu Udonsi
  - Arnamoy Bhattacharyya
  - hao wang
  - Sukwon Oh
- Webpage:
  - <http://www.cdf.toronto.edu/~csc369h/summer/index.shtml>



# Course Objective

- To understand
  - The role of the OS
  - Its major components
  - The design principles and implementations
  - Working on real Operating System assignments

# Workload



Linux kernel has  
more than 10 million  
lines of code!

- This course is very work-intensive
- Why?
  - Lectures cover a lot of new concepts
    - Many of them very abstract
    - Learning by doing!
  - This course is much more like the real world:
    - Assignments build on realistic OS
      - Lot of existing code given to you
      - Don't expect to understand all of it
    - You need to be able to work in a team
    - You need to be comfortable with the prereqs.

# Assignments (40%)



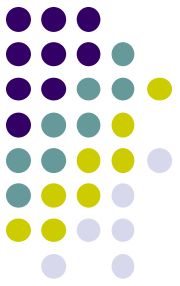
- Code Reading component:
  - Questions about the code we give you
  - Reinforce lecture concepts, familiarize you with the code **before** you start writing
- Programming component
  - Language for this course is C
  - Correctness and performance considerations
- Design document component
  - **Think** about what you need to build **first**
  - Tell us in plain English what you were thinking.
  - Tell us how to find your code.

# Exams



- Midterm (20%)
  - Tuesday, in class time
  - 20% each
  - Question about assignments and course material
- Final (40%),
  - Will be cumulative
  - Question about assignments
  - Lots of final exams from CSC369 are available for study
  - **You have to get more than 40(out of 100) to pass the course**

# There's tons of help ...



- Instructor
- TAs
- Piazza:
  - Check course webpage
- Who to ask in case of questions?

**Question regarding lecture material or logistics?**

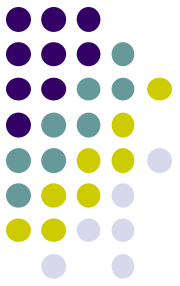
Me: [sina.mrj@gmail.com](mailto:sina.mrj@gmail.com)

Include "C69" in subject line

**Question regarding assignment & grading?**

- 1) Tutorial: Tue 8-9pm
- 2) Discussion board
- 3) TAs

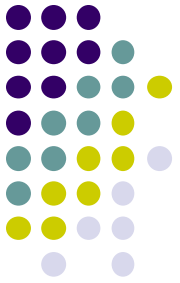
# Academic Dishonesty



- Plagiarism and cheating
  - Please don't do it!
  - Serious academic offenses
  - Can discuss tools and concepts with classmates
  - Can discuss solutions to assignments with your partner only!
  - All potential cases will be investigated fully
  - We run Moss

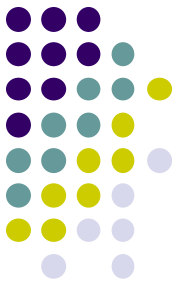


# Grading



Work	Notes	Weight	Due Date
Assignment 1		13%	June 13 <sup>th</sup>
Midterm		20%	July 4 <sup>th</sup>
Assignment 2		15%	July 11 <sup>th</sup>
Assignment 3		12%	August 14 <sup>th</sup>
Final		40%	you have to get more than 40%

# Late policy



- Assignments will be submitted electronically
- Due at 11:55pm on due date
- You can submit up to 2 days late
  - Except for last assignment
- 10% penalty for each day

# Introduction

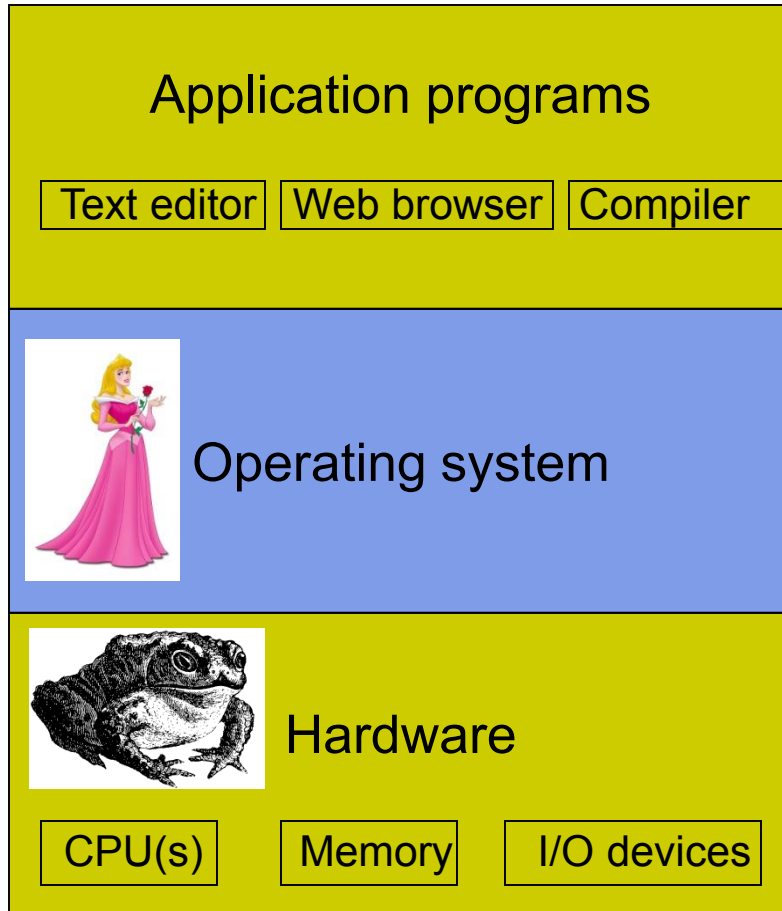


- Read Chapter 1
  - Some (much?) of this should be review
- What is an OS and why do I want one?
  - How does it relate to the other parts of a computer system?
  - Review some computer organization and C concepts
- What are the major parts of an OS?

# What is an Operating System?

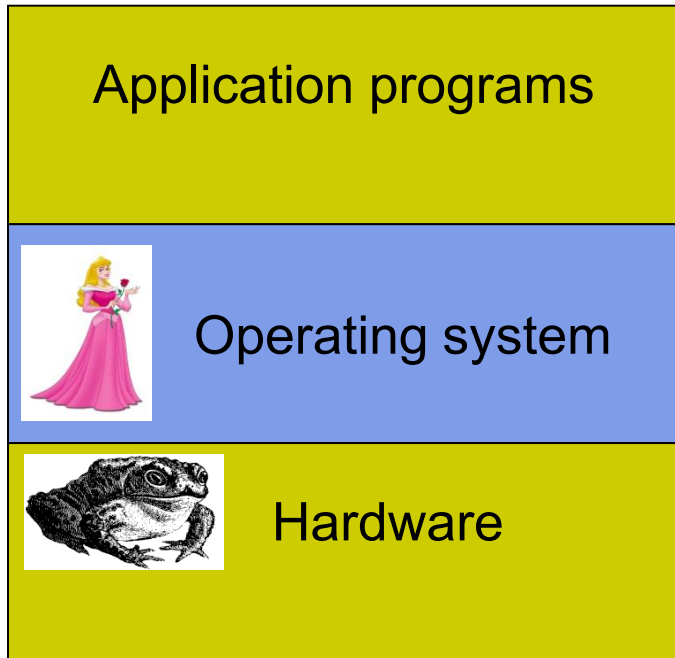
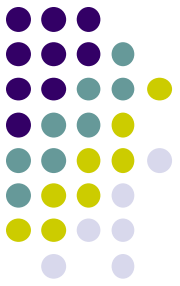


The software layer between user applications and hardware.



- Turns ugly hardware into beautiful abstractions (provides services)
- Serves as a resource manager
  - Allows proper use of resources (hardware, software, data)
- Serves as a control program (protection)
  - Controls execution of user programs to prevent errors and improper use of the computer

# What is an Operating System?

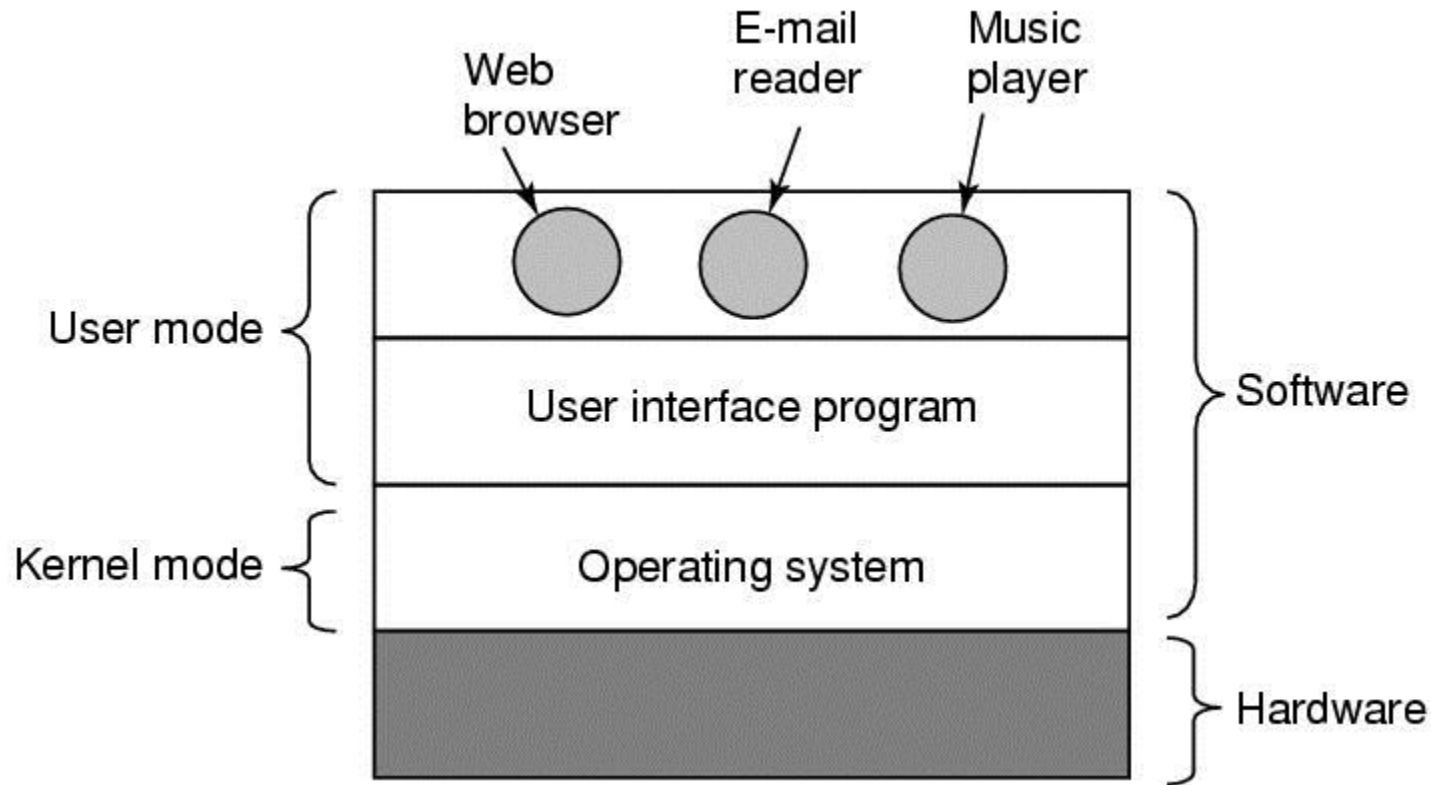


The software layer between user applications and hardware.

- Turns ugly hardware into beautiful abstractions (provides services)
  - Serves as a resource manager
  - Serves as a control program (protection)
- Will spend much of the rest of the semester on three core abstractions and resource management services:
    - Processes & threads
    - Memory management
    - File & I/O systems

OS is a software that manages hardware

# What is an Operating System



# Overview of Computer System



User App

User App

User App

Synchronization

Scheduling

Memory  
Mgmt

File System

Networking

OS

Machine Dependent Code

## Hardware

CPU

Memory Bus

Main  
Memory

I/O Bridge

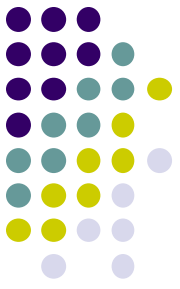
I/O Bus

Disk  
Controller

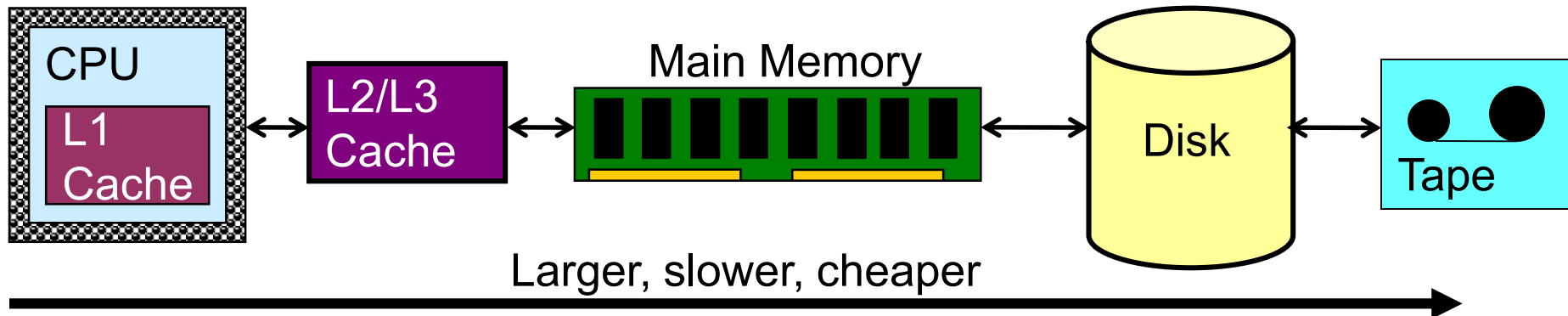
Display  
Controller

Ethernet  
Controller

# Storage Hierarchy

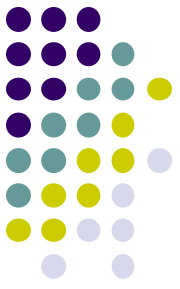


- Processor registers, main memory, and auxiliary memory form a rudimentary memory hierarchy
- The hierarchy can be classified according to memory speed, cost, and volatility
- Caches can be installed to hide performance differences when there is a large access-time gap between two levels





# Storage Structure



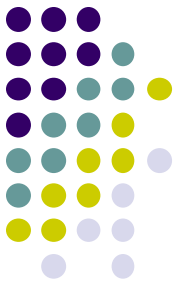
- Main memory (DRAM) stores programs and data during program execution
  - DRAM cannot store these permanently because it is too small & it is a volatile storage device
- Forms a large array of *bytes* ( $1 \text{ byte} = 8 \text{ bits}$ ) of memory, each with its own address
  - We say that main memory is *byte-addressable*

# Caching



- When the processor accesses info at some level of the storage hierarchy, that info may be copied to a cache memory closer to the processor, on a temporary basis
  - A cache is smaller and costlier
- Because caches have limited sizes, cache management is an important design problem
  - Coherency

# Concurrency



- Every modern computer is a multiprocessor
  - CPU and device controllers can execute concurrently, competing for memory cycles
  - A memory controller synchronizes access to shared memory
  - *Interrupts* allow device controllers to signal the CPU that some event has occurred (e.g. disk I/O complete, network packet arrived, etc.)
    - Generated by a hardware device
  - Interrupts are also used to signal errors (e.g. division by zero) or requests for OS service from a user program (a *system call*)
    - These types of interrupts are called *traps* or *exceptions*

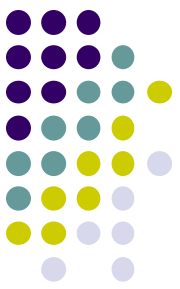


**An Operating System is an *event-driven program***

hardware issue interrupts

user program (syscall) issue traps/exceptions

# Aside: C Programming & Memory



- A variable in a C program is a symbolic name for a data item, stored in memory
  - The type of the variable indicates how much *storage* (how many bytes) it needs
  - Type also determines alignment requirements
  - The *address* of the variable is an index into the big array of memory words where the data item is stored
  - The *value* of the variable is the actual contents of memory at that location
  - A *pointer* type variable is just a data item whose contents are a memory location (usually the address of another var)

# C Example

```
int main() {
    char a = 'h';
    int b = 0xdeadbeef;
    char *c = &a;
    int *d = &b;

    printf("b=%d (0x%x)\n",
           b, b);
}
```

- **char** data type is 1 byte in size
- **int** data type is 1 word in size (32 bits for most current architectures)
  - occupies 4 bytes, should be word-aligned
- pointer types are all 1 word in size

little endian:

0xbffffeb08	
0xbffffeb07	'h' (0x68)
b06	Unused
b05	Unused
b04	Unused
b03	0xde
b02	0xad
b01	0xbe
0xbffffeb00	0xef
aff	0xbf
afe	0xff
afd	0xeb
0xbfffeaafc	0x07
afb	0xbf
afa	0xff
af9	0xeb
0xbfffeaf8	0x00

c=&a

d=&b

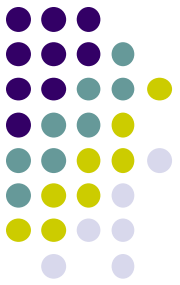
Increasing addresses

Memory



# Processes and Threads

Reading: 2.1-2.2

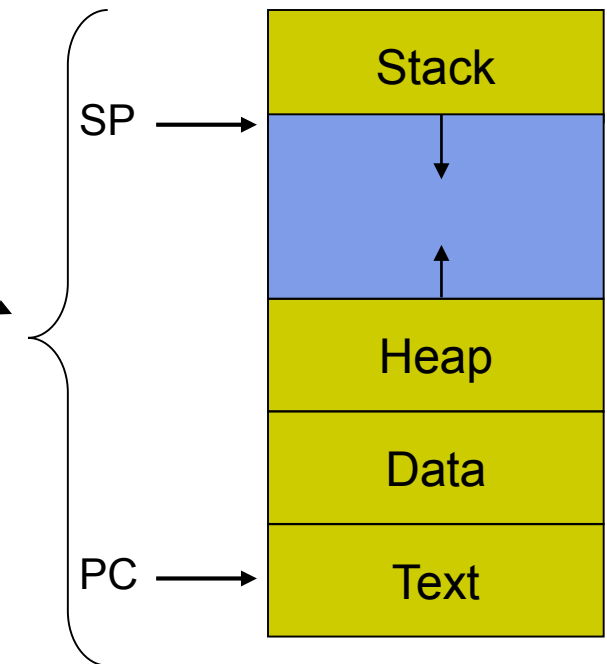


# Part 1: The Process Concept

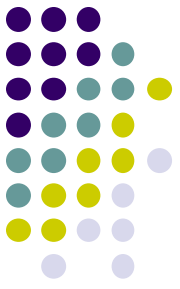


- Process = job / unit of work
- Process = a program in execution
- A process contains all state of program in execution

- An address space
- Set of OS resources
  - Open files network connections ...
- Set of general-purpose registers with current values



- A process is named by its process ID (PID)



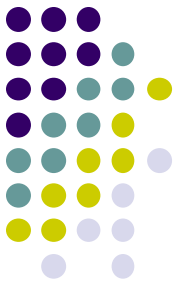
# Process Data Structures

How does the OS represent a process in the kernel?

- At any time, there are many processes in the system, each in its own particular state
- The OS data structure representing each process is called the **Process Control Block (PCB)**
- The PCB contains all of the info about a process
- The PCB also is where the OS keeps all of a process' hardware execution state (PC, SP, regs, etc.) when the process is not running
  - This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware (FreeBSD, 81 fields, 408 bytes)

context scheduling...





# Process Control Block

- Generally includes:
  - Process state (ready, running, blocked ...)
  - Program counter: address of the next instruction
  - CPU registers: must be saved at an interrupt
  - CPU scheduling information: process priority
  - Memory management info: page tables
  - I/O status information: list of open files

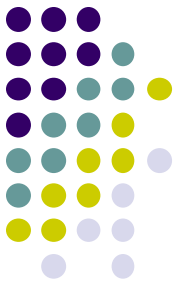
# Linux PCB



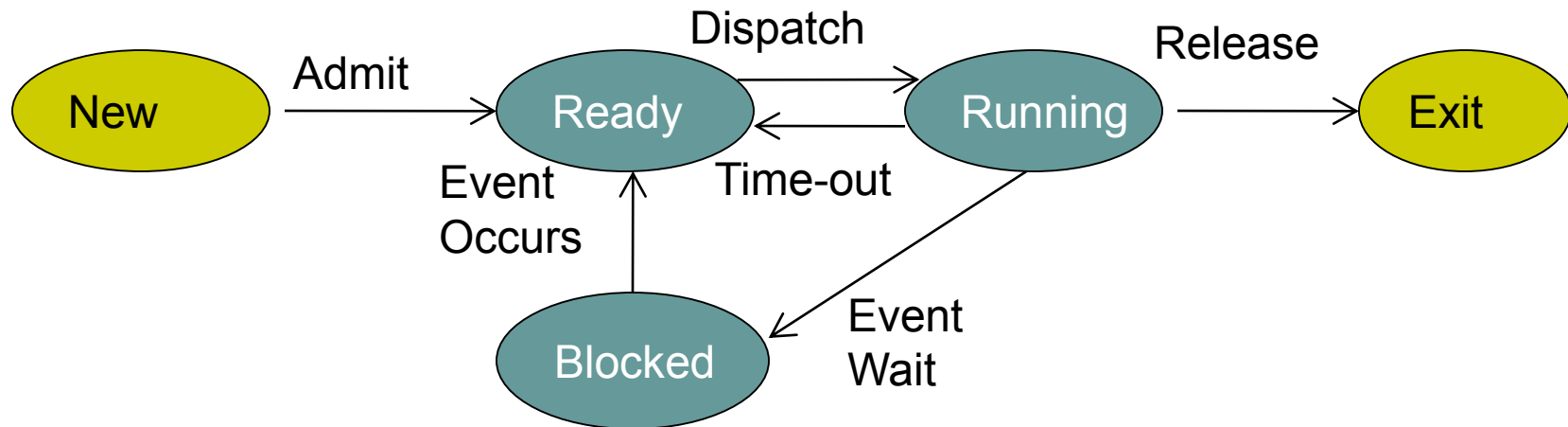
- Called the *task\_struct* in Linux
  - Defined in `/include/linux/sched.h`

```
struct task_struct {  
    /* these are hardcoded - don't touch */  
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */  
    long counter; long priority; unsigned long signal;  
    unsigned long blocked; /* bitmap of masked signals */  
    unsigned long flags; /* per process flags, defined below */  
    int errno; long debugreg[8]; /* Hardware debugging registers */  
    struct exec_domain *exec_domain;  
    /* various fields */  
    struct linux_binfmt *binfmt;  
    struct task_struct *next_task, *prev_task;  
    struct task_struct *next_run, *prev_run;  
    unsigned long saved_kernel_stack;  
    unsigned long kernel_stack_page;  
    int exit_code, exit_signal;  
    ...  
};
```

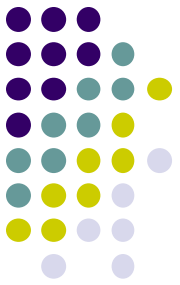
# Process states & state changes



- The OS manages processes by keeping track of their *state*
- Different *events* cause changes to a process state, which the OS must record/implement



PCB is in RAM; frequently used process has PCB in cache

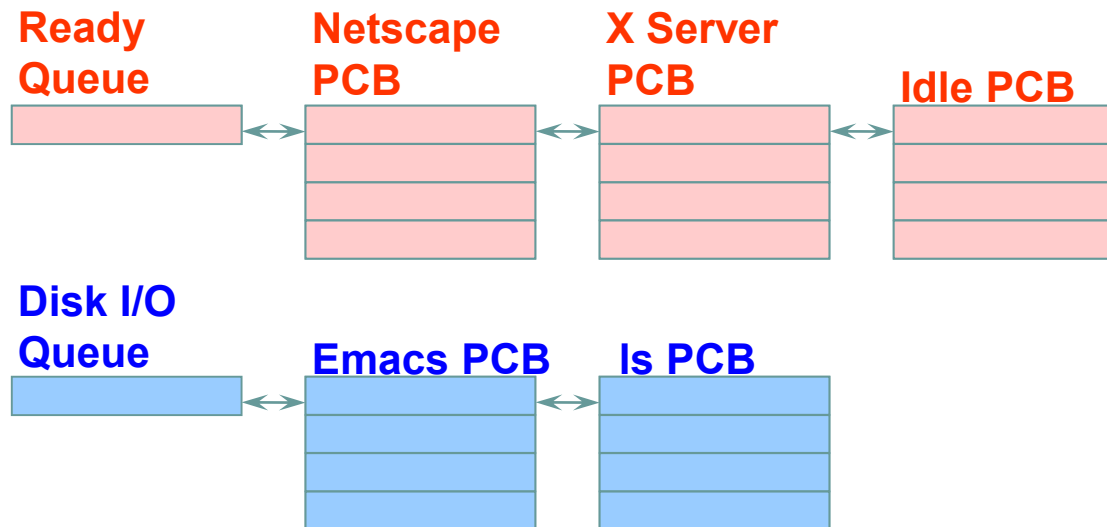


# State Queues

How does the OS keep track of processes?

- The OS maintains a collection of queues that represent the state of all processes in the system
- Typically, the OS has one queue for each state
  - Ready, waiting, etc.
- Each PCB is queued on a state queue according to its current state
- As a process changes state, its PCB is unlinked from one queue and linked into another

# State Queues



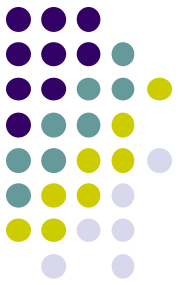
Console Queue

Sleep Queue

•  
•  
•

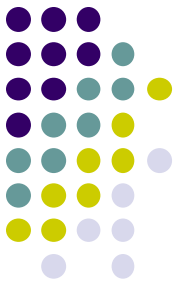
- There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)
- OS/161 maintains a single wait queue
  - "struct array \*sleepers" in kern/thread/thread.c
  - record address of thing thread is waiting for

every hardware is a resource, may need a queue



# PCBs and State Queues

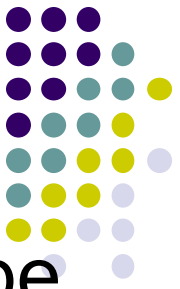
- PCBs are data structures dynamically allocated in OS memory
- When a process is created, the OS allocates a PCB for it, initializes it, and places it on the Ready queue
- As the process computes, does I/O, etc., its PCB moves from one queue to another
- When the process terminates, its PCB is de-allocated



# What is a Context Switch?

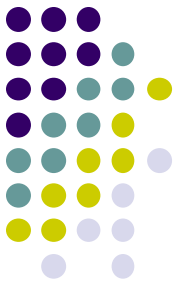
- **Context switch**: switch the CPU to another process, saving the state of the old process and loading the saved state for the new process
- Context switch time is pure overhead, so some systems offer specific hardware support
- A performance bottleneck, so new structures (threads) are being used to avoid it

# Operations on Processes



- Processes execute concurrently and must be created and deleted dynamically
- **Process creation:**
  - System Initialization
  - A running process
  - A user request
  - Initialization of a batch job
- **Process termination:**
  - When a process finishes executing last statement
  - When a parent causes the termination of a child
  - An Error occurred





# Process Creation

- A process is created by another process
  - Parent is creator, child is created
    - In Linux, the parent is the “PPID” field of “ps -f”
- In some systems, the parent defines (or donates) resources and privileges for its children
  - Unix: Process User ID is inherited – children of your shell execute with your privileges
- After creating a child, the parent may either wait for it to finish its task or continue in parallel (or both)

whether to wait or not

# Process Creation: Unix



- In Unix, processes are created using `fork()`  
`int fork()`
- `fork()`
  - Creates a new address space i.e. stack, heap, global, text,...
  - reason for overhead Initializes the address space with a **copy** of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
- Fork returns **twice**
  - Returns the child's PID to the parent, "0" to the child
  - Huh?

# fork()



```
int main(int argc, char *argv[])
{
    char *name = argv[0];
    int child_pid = fork();
    if (child_pid == 0) {
        printf("Child of %s is %d\n", name,
getpid());
        return 0;
    } else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```

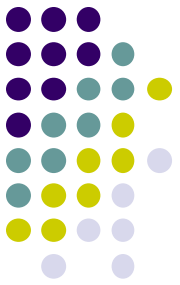


Child



Parent

What does this program print?



# Example Output

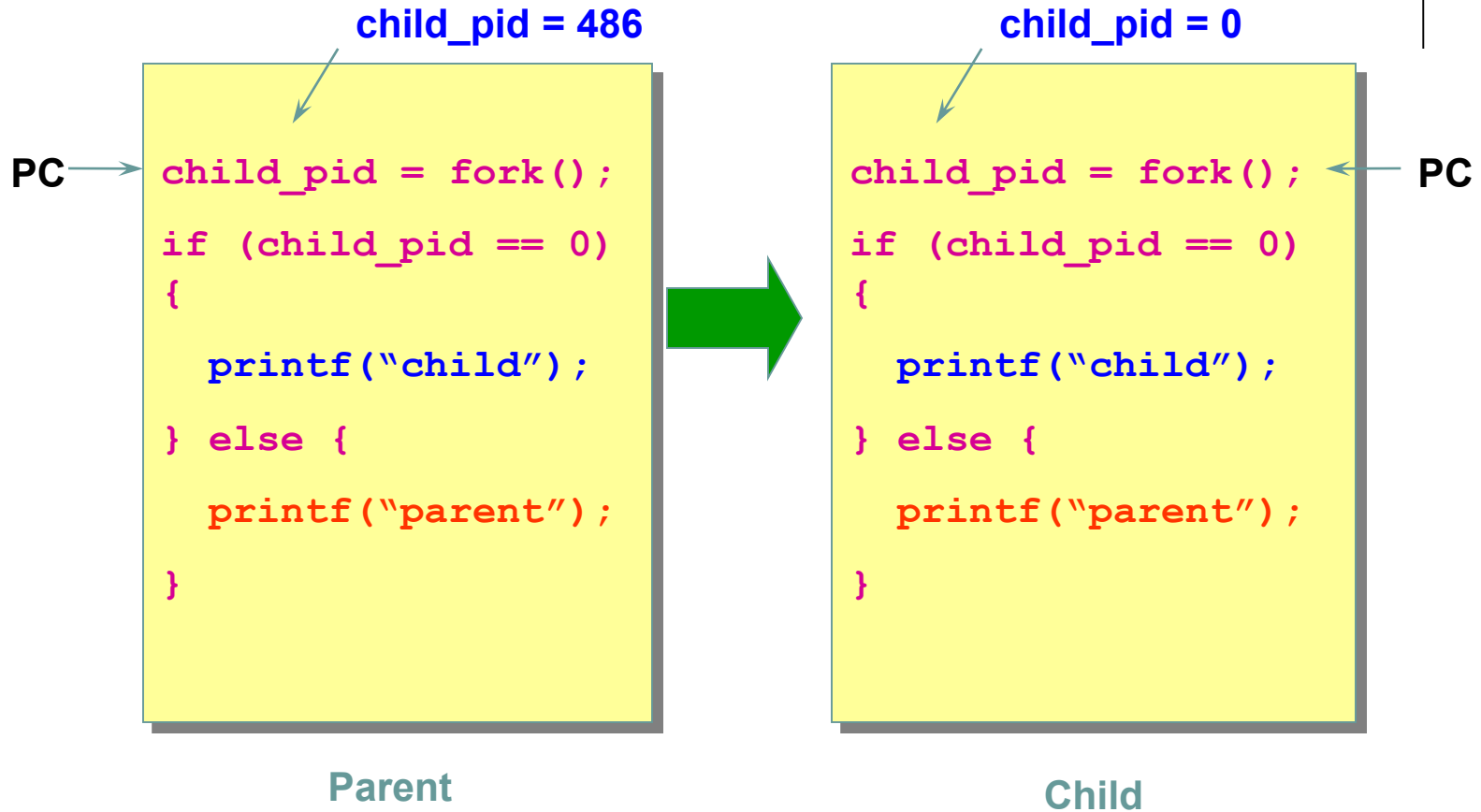
```
smeraji% cc t.c
```

```
smeraji% ./a.out
```

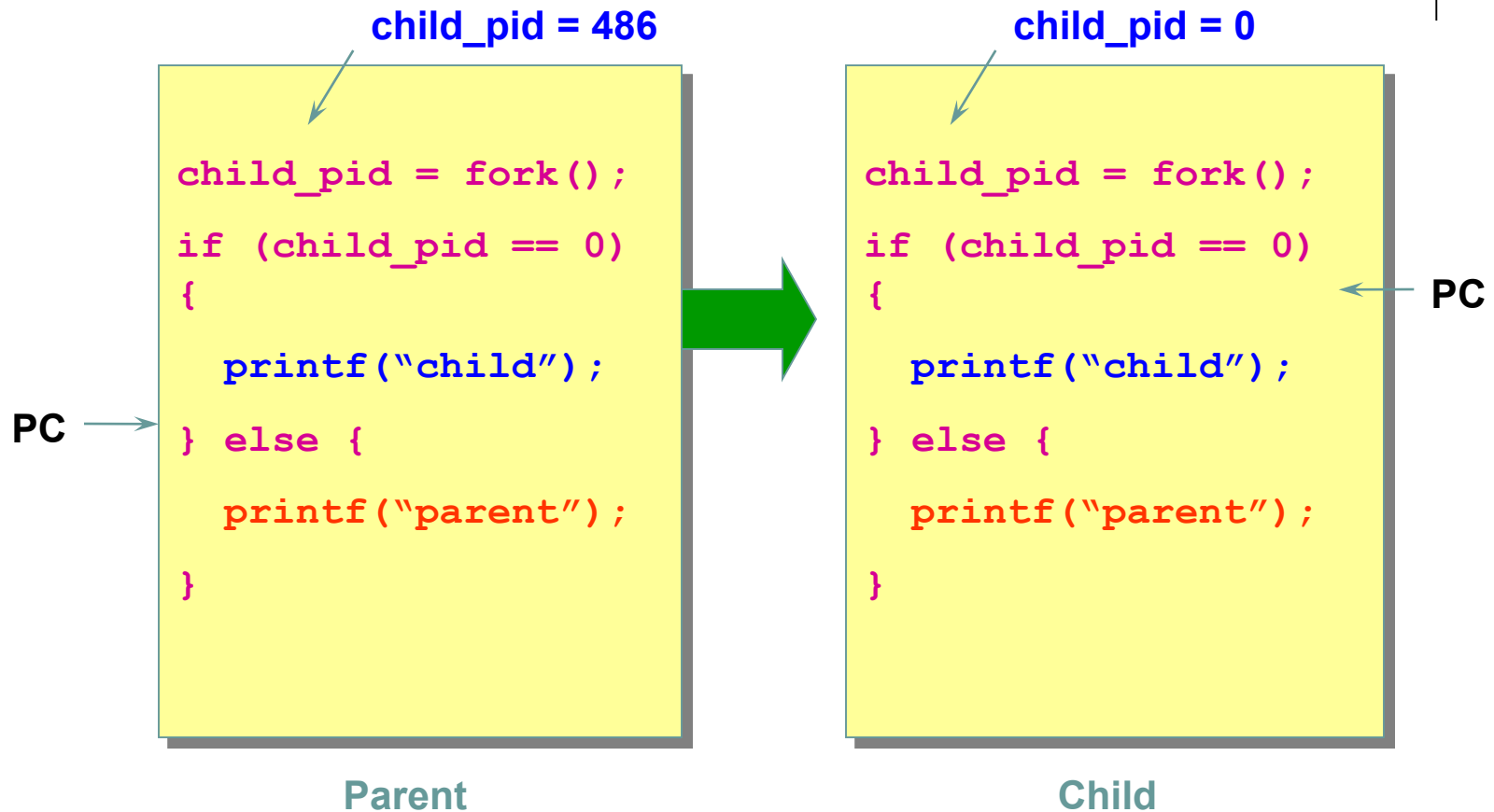
```
My child is 486
```

```
Child of a.out is 486
```

# Duplicating Address Spaces



# Divergence



# Why fork()?



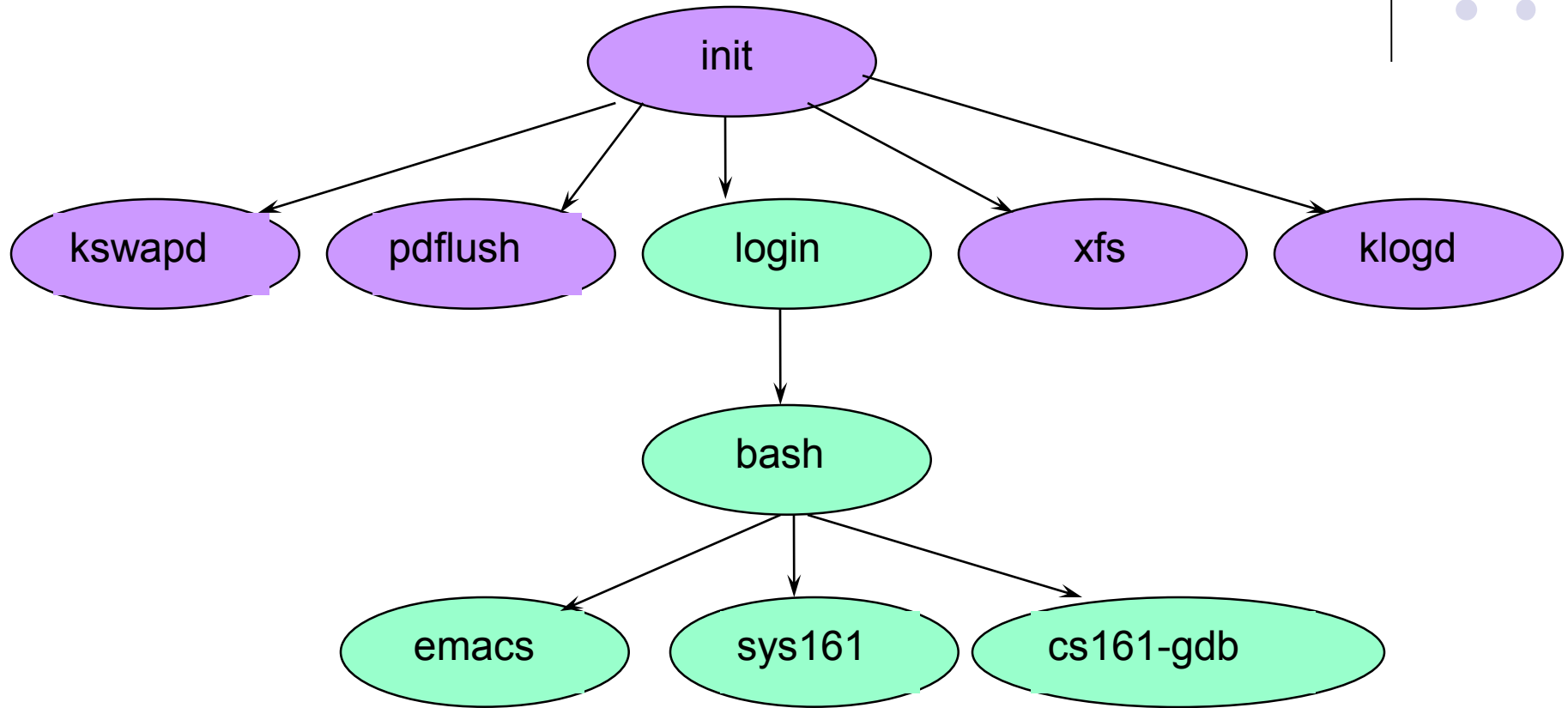
- Very useful when the child...
  - Is cooperating with the parent
  - Relies upon the parent's data to accomplish its task

- Example: Web server

```
while (1) {  
    int sock = accept();  
    if ((child_pid = fork()) == 0) {  
        Handle client request  
    } else {  
        ...  
    }  
}
```

parent process accepts request only  
child process process request

# Linux Process Tree





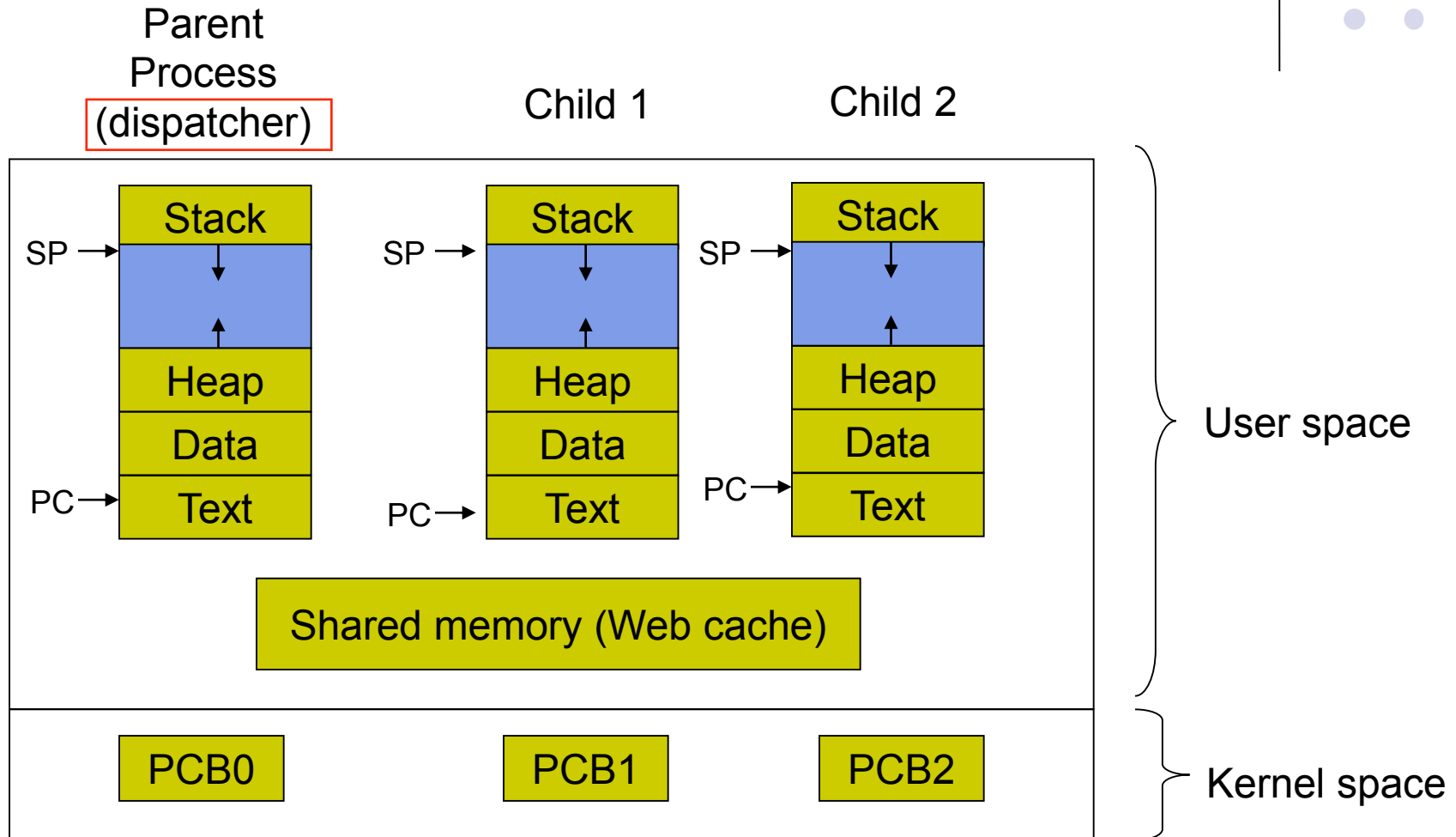
# Unix Shells

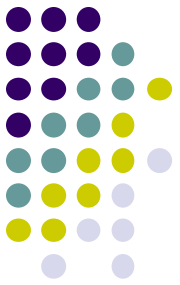


```
while (1) {  
    char *cmd = read_command();  
    int child_pid = fork();  
    if (child_pid == 0) {  
        exec(cmd);  
    } else {  
        wait(child_pid);  
    }  
}
```

parent waits for child to finish before printing the prompt

# Example for use of processes: Concurrent Web Server

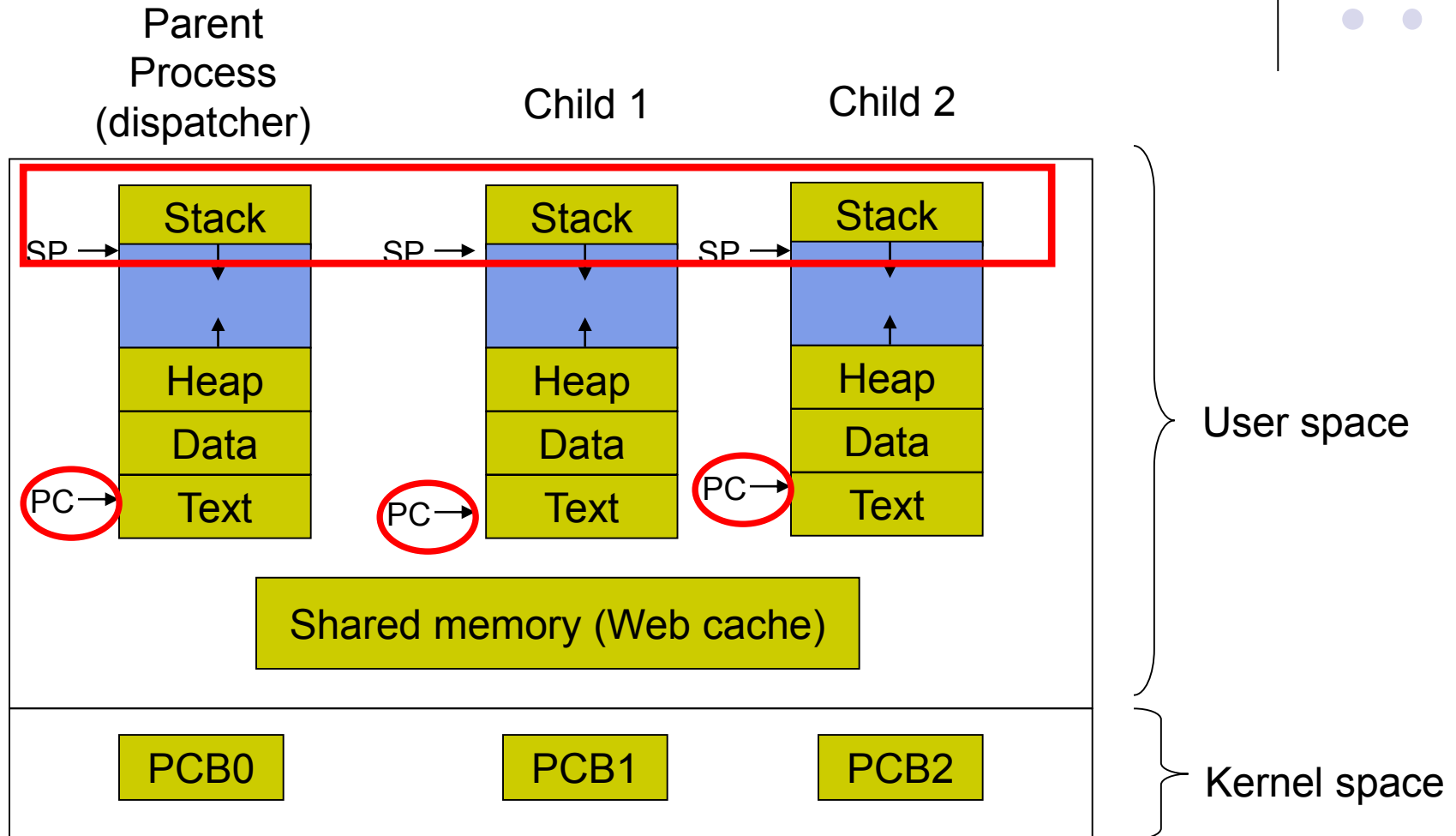
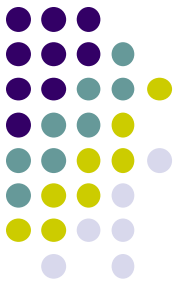


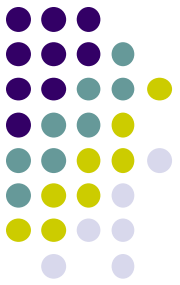


# Parallel Programs

- Recall our Web server example that forks off copies of itself to handle multiple simultaneous requests
  - Or any parallel program that executes on a multiprocessor
- To execute these programs we need to
  - Create several processes that execute in parallel
  - Create shared memory for processes to share data
  - Have the OS schedule these processes in parallel
- This situation is **very inefficient**
  - **Space:** PCB, page tables, etc.
  - **Time:** create data structures, fork and copy addr space, etc.

# Example for use of processes: Concurrent Web Server

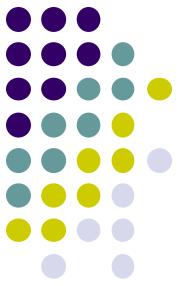




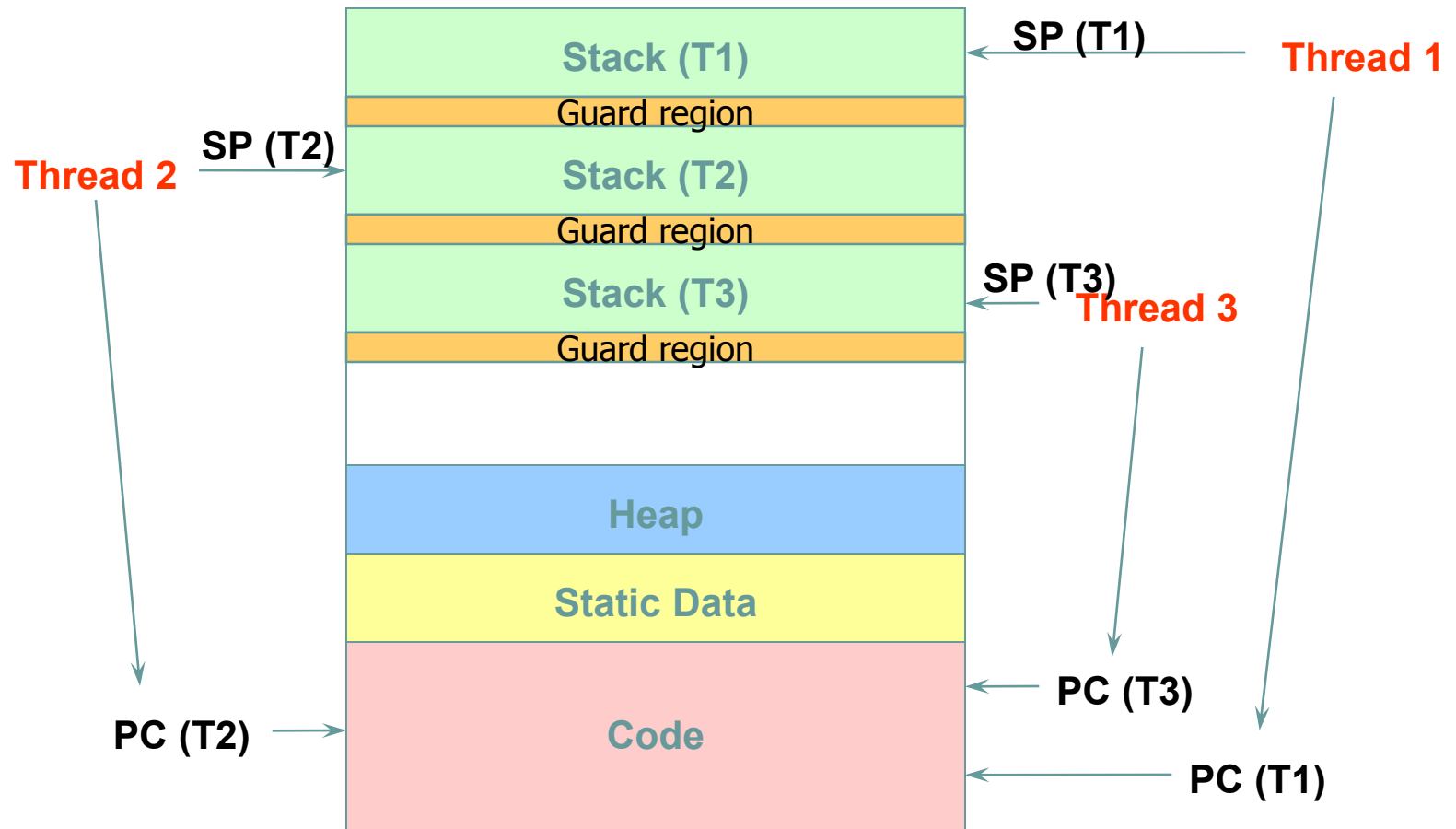
# Rethinking Processes

- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)
- What don't they share?
  - Each has its own execution state: PC, SP, and registers
- **Key idea:** Why don't we separate the concept of a process from its execution state?
  - **Process:** address space, privileges, resources, etc.
  - **Execution state:** PC, SP, registers
- Exec state also called **thread of control**, or **thread execution state**

# Idea: Have threads in a Process!

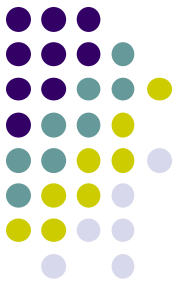


only PC and SP and hardware register are distinct

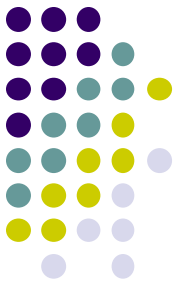


shared heap may introduced synchronization issues

# Threads



- Separate the concepts of *processes* and *threads*
  - The **thread** defines a sequential execution stream within a process (PC, SP, registers)
  - The **process** defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
  - Processes, however, can have multiple threads
  - Every process has at least one thread
- Processes are the containers in which threads execute
  - Processes become static, threads are the dynamic entities



# Threads: Concurrent Servers

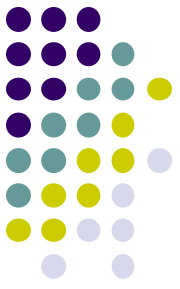
- Recall our forking Web server:

```
while (1) {  
    int sock = accept();  
    if ((child_pid = fork()) == 0) {  
        Handle client request  
    } else {  
        Close socket  
    }  
}
```

- Using `fork()` to create new processes to handle requests in parallel is overkill for such a simple task



# Threads: Concurrent Servers



- Instead, we can create a new thread for each request

```
web_server() {  
    while (1) {  
        int sock = accept();  
        thread_fork(handle_request, sock);  
    }  
}  
  
handle_request(int sock) {  
    Process request  
    close(sock);  
}
```

# Thread Interface (Pthread API)



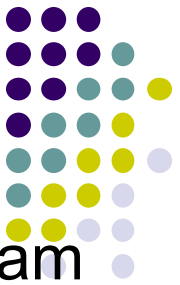
- **pthread\_create**(pthread\_t \*tid, pthread\_attr\_t attr, void \*(\*start\_routine)(void \*), void \*arg)
  - Create a new thread of control
  - New thread id returned in **tid**, new thread starts executing in **start\_routine** with argument **arg**
- **pthread\_join**(pthread\_t tid)
  - Wait for tid to exit
- **pthread\_cancel**(pthread\_t tid)
  - Destroy tid
- **pthread\_exit**()
  - Terminate the calling thread

# Thread Scheduling



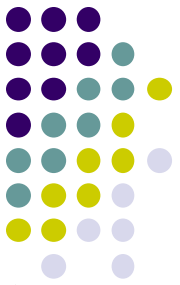
- The thread scheduler determines when a thread runs
- It uses queues to keep track of what threads are doing
  - Just like the OS and processes
  - But it is implemented at user-level in a library
- Run queue: Threads currently running (usually one)
- Ready queue: Threads ready to run
- Are there wait queues?
  - How would you implement `thread_sleep(time)`?

# Threads Summary

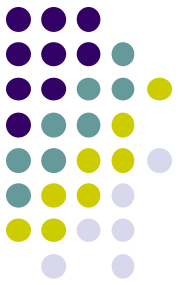


- The operating system is a large multithreaded program
  - Each process executes as a thread within the OS
- Multithreading is also very useful for applications
  - Efficient multithreading requires fast primitives
  - Processes are too heavyweight
- Solution is to separate threads from processes
- Now, how do we get our threads to correctly cooperate with each other?
  - Synchronization... coming right up!

# Cooperating Processes



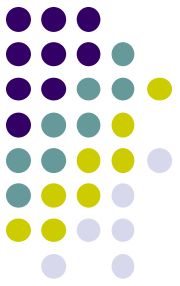
- A process is independent if it cannot affect or be affected by the other processes executing in the system
- No data sharing  $\Rightarrow$  process is independent
- A process is cooperating if it is not independent
- Cooperating processes must be able to communicate with each other and to synchronize their actions



# Interprocess Communication

- Cooperating processes need to exchange information, using either
  - Shared memory (e.g. `fork()`)
  - Message passing i.e. A saves to file on disk, before B reads file
- Message passing models
  - `send(P, msg)` – send msg to process P
  - `receive(Q, msg)` – receive msg from process Q

# Announcement



- Check course website regularly