Chapter 2: Processes & Threads

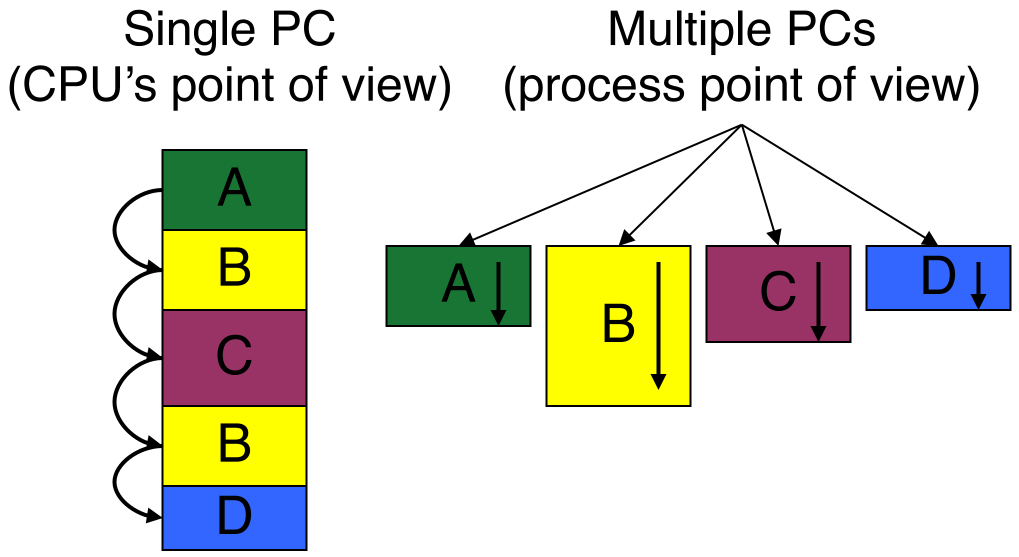
Processes and threads

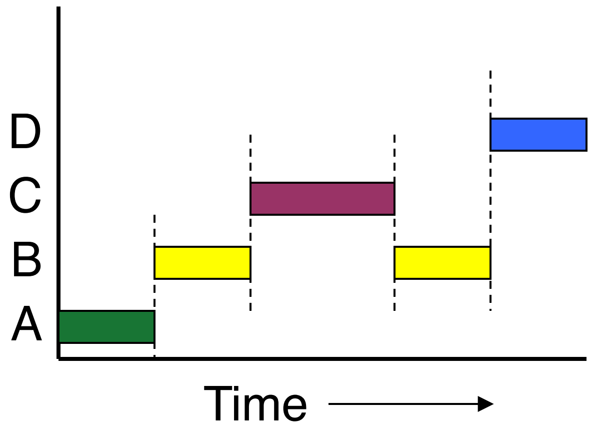
- Processes
- Threads
- Scheduling
- Interprocess communication
- Classical IPC problems

What is a process?

- Code, data, and stack
  - Usually (but not always) has its own address space
- Program state
  - CPU registers
  - Program counter (current location in the code)
  - Stack pointer
- Only one process can be running in the CPU at any given time!

The process model





- Multiprogramming of four programs
- Conceptual model
  - 4 independent processes
  - Processes run sequentially
- Only one program active at any instant!
  - That instant can be very short…

When is a process created?

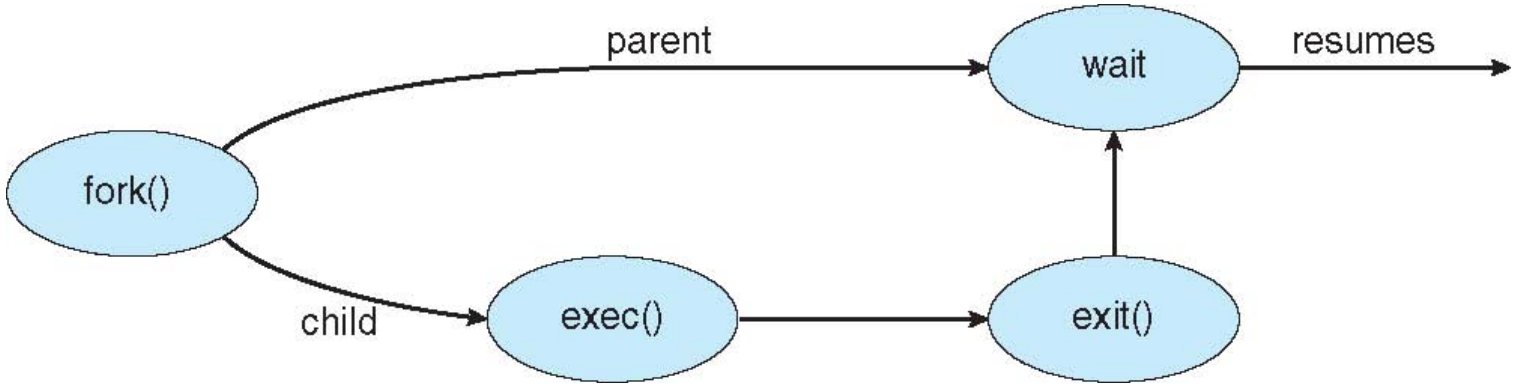
- Processes can be created in two ways
  - System initialization: one or more processes created when the OS starts up
  - Execution of a process creation system call: something explicitly asks for a new process
- System calls can come from
  - User request to create a new process (system call executed from user shell)
  - Already running processes
    - User programs
    - System daemons

Process Creation

- **Parent** process creates **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



When do processes end?

- Conditions that terminate processes can be
  - Voluntary
  - Involuntary
- Voluntary
  - Normal exit
  - Error exit
- Involuntary
  - Fatal error (only sort of involuntary)
  - Killed by another process

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow a child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

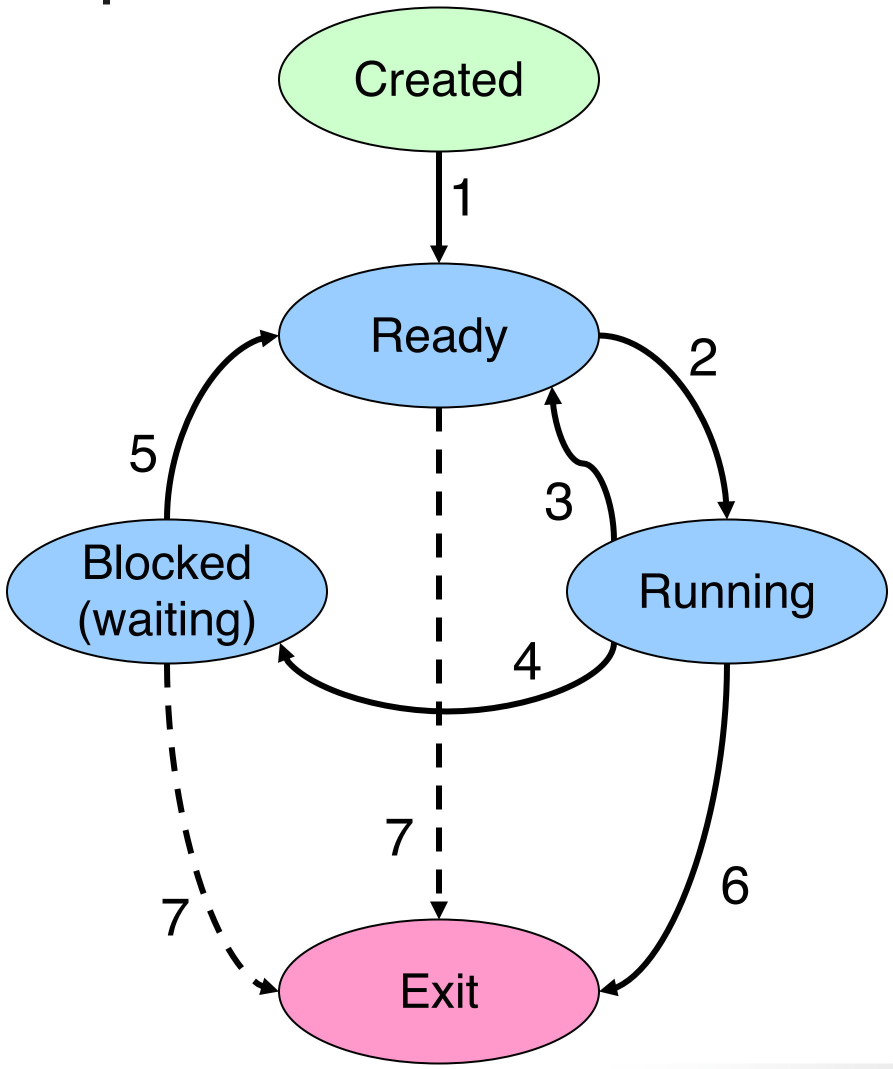
```
      pid = wait(&status);
```

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait`** , process is an **orphan**

Process hierarchies

- Parent creates a child process
  - Child processes can create their own children
- Forms a hierarchy
  - UNIX calls this a "process group"
  - If a process exits, its children are "inherited" by the exiting process's parent
- Windows has no concept of process hierarchy
  - All processes are created equal

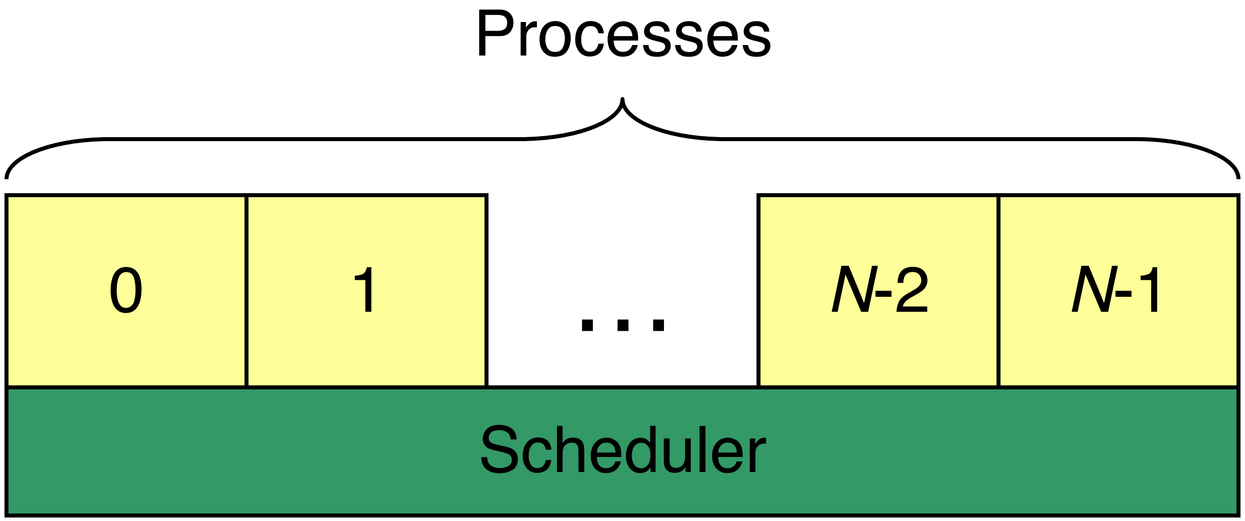
Process states



- Process in one of 5 states
  - Created
  - Ready
  - Running
  - Blocked
  - Exit
- Transitions between states
  1 - Process enters ready queue
  2 - Scheduler picks this process
  3 - Scheduler picks a different process
  4 - Process waits for event (such as I/O)
  5 - Event occurs
  6 - Process exits
  7 - Process ended by another process

Processes in the OS

- Two “layers” for processes
- Lowest layer of process-structured OS handles interrupts, scheduling
- Above that layer are sequential processes
  - Processes tracked in the *process table*
  - Each process has a *process table entry*



What’s in a process table entry?

May be stored on stack {

Process management

Registers
Program counter
CPU status word
Stack pointer
Process state
Priority / scheduling parameters
Process ID
Parent process ID
Signals
Process start time
Total CPU usage

File management

Root directory
Working (current) directory
File descriptors
User ID
Group ID

Memory management

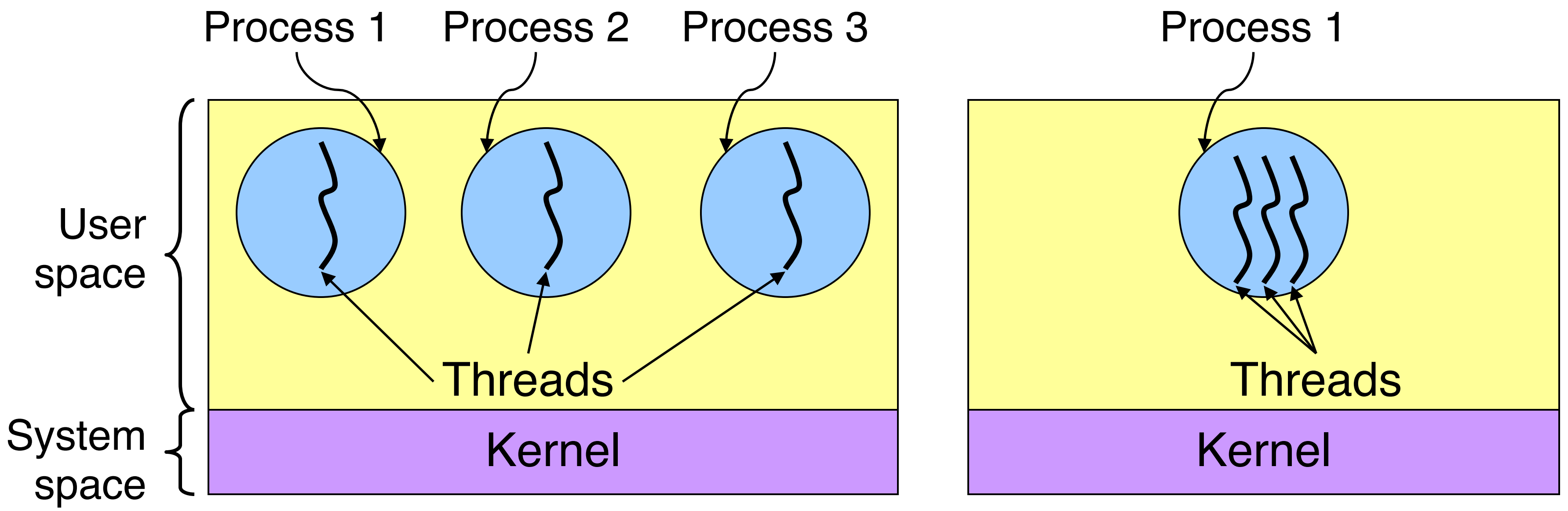
Pointers to text, data, stack
or
Pointer to page table

What happens on a trap/interrupt?

1. Hardware saves program counter (on stack or in a special register)
2. Hardware loads new PC, identifies interrupt
3. Assembly language routine saves registers
4. Assembly language routine sets up stack
5. Assembly language calls C to run service routine
6. Service routine calls scheduler
7. Scheduler selects a process to run next (might be the one interrupted…)
8. Assembly language routine loads PC & registers for the selected process

Threads: “processes” sharing memory

- Process == address space
- Thread == program counter / stream of instructions
- Two examples
  - Three processes, each with one thread
  - One process with three threads

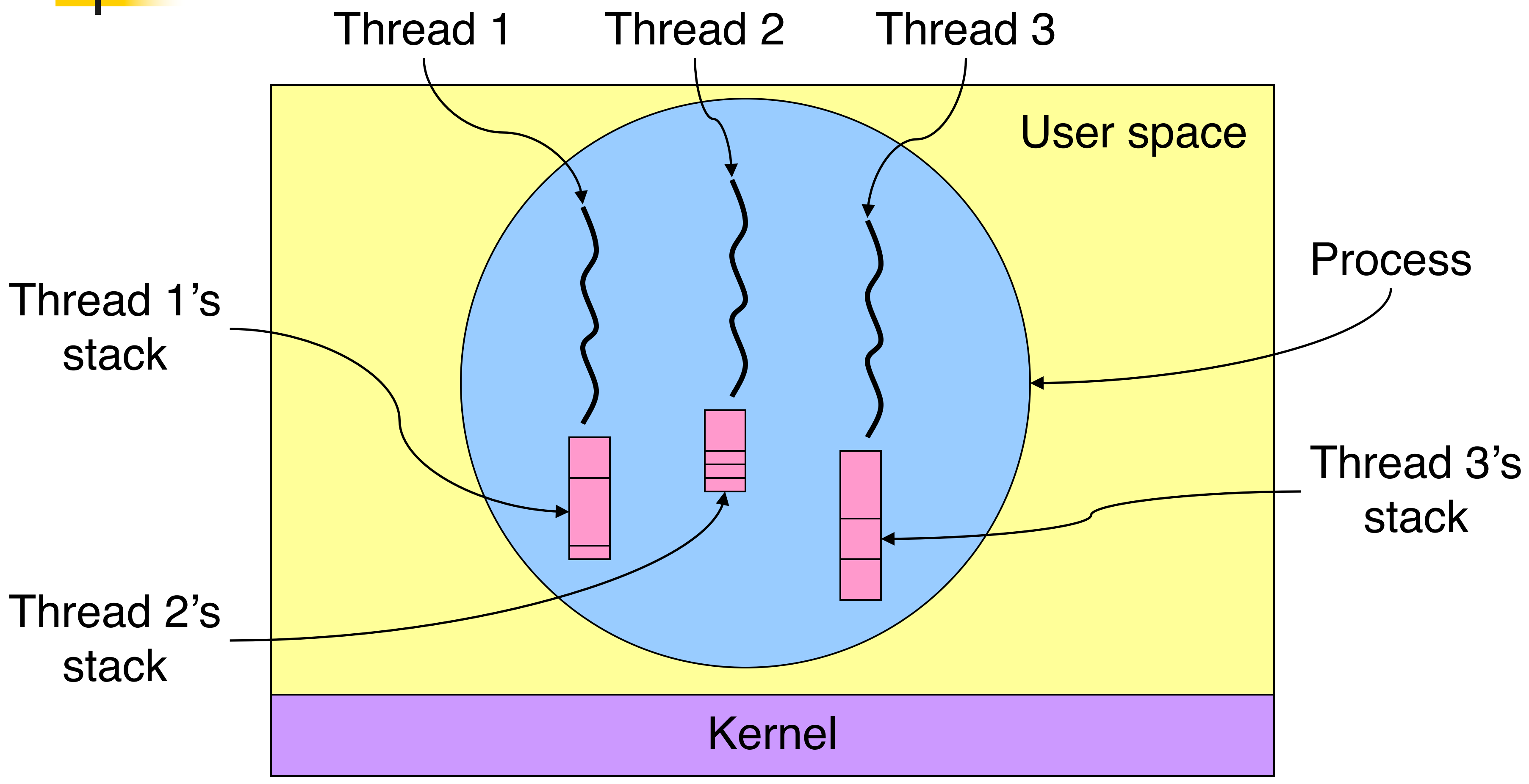


Process & thread information

Per process items
Address space
Open files
Child processes
Signals & handlers
Accounting info
Global variables

Per thread items	**Per thread items**	**Per thread items**
Program counter	Program counter	Program counter
Registers	Registers	Registers
Stack & stack pointer	Stack & stack pointer	Stack & stack pointer
State	State	State

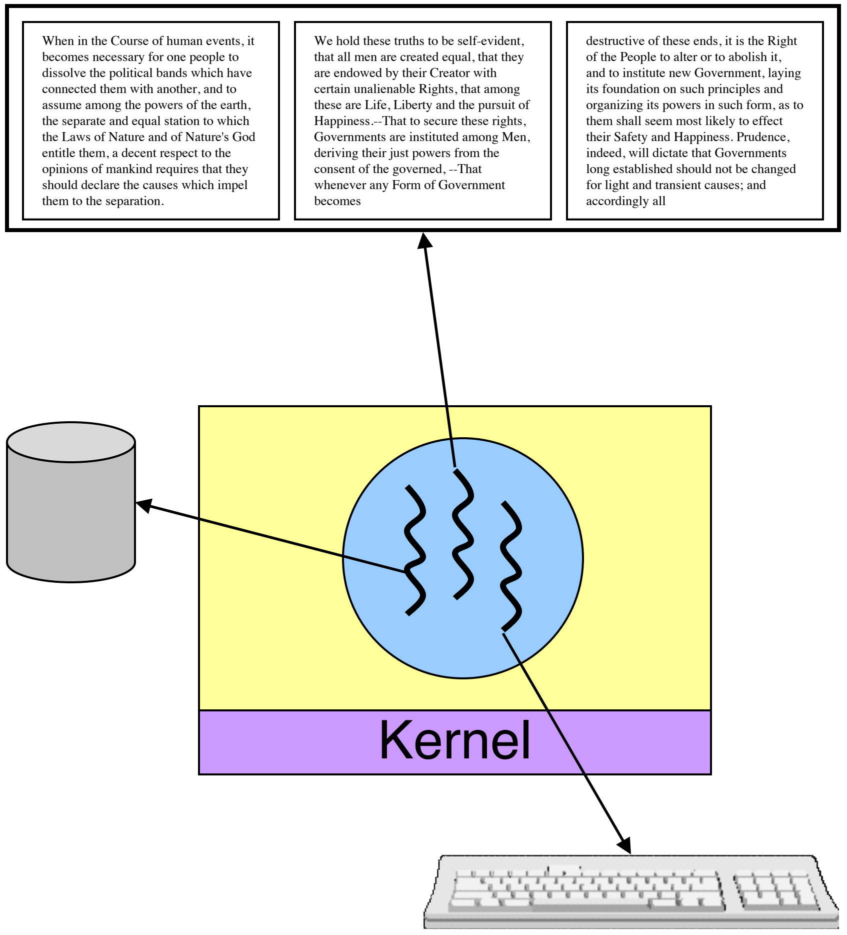
Threads & Stacks



=> Each thread has its own stack!

Why use threads?

- Allow a single application to do many things at once
  - Simpler programming model
  - Less waiting
- Threads are faster to create or destroy
  - No separate address space
- Overlap computation and I/O
  - Could be done without threads, but it's harder
- Example: word processor
  - Thread to read from keyboard
  - Thread to format document
  - Thread to write to disk

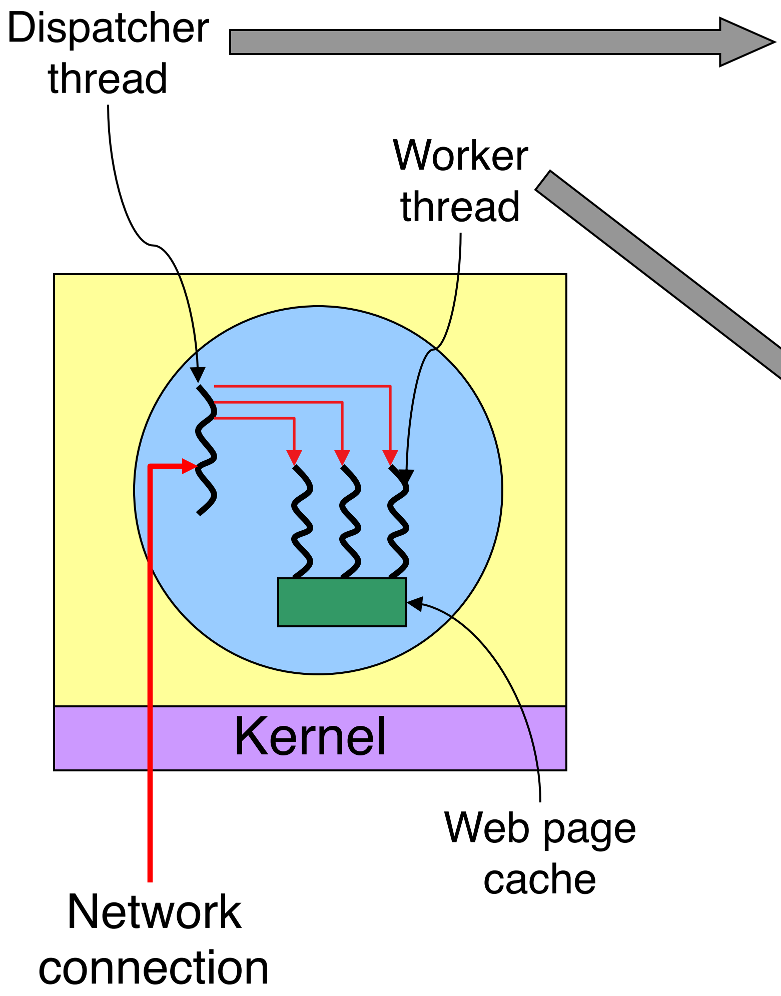


When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable Rights, that among these are Life, Liberty and the pursuit of Happiness.--That to secure these rights, Governments are instituted among Men, deriving their just powers from the consent of the governed, --That whenever any Form of Government becomes

destructive of these ends, it is the Right of the People to alter or to abolish it, and to institute new Government, laying its foundation on such principles and organizing its powers in such form, as to them shall seem most likely to effect their Safety and Happiness. Prudence, indeed, will dictate that Governments long established should not be changed for light and transient causes; and accordingly all

Multithreaded Web server



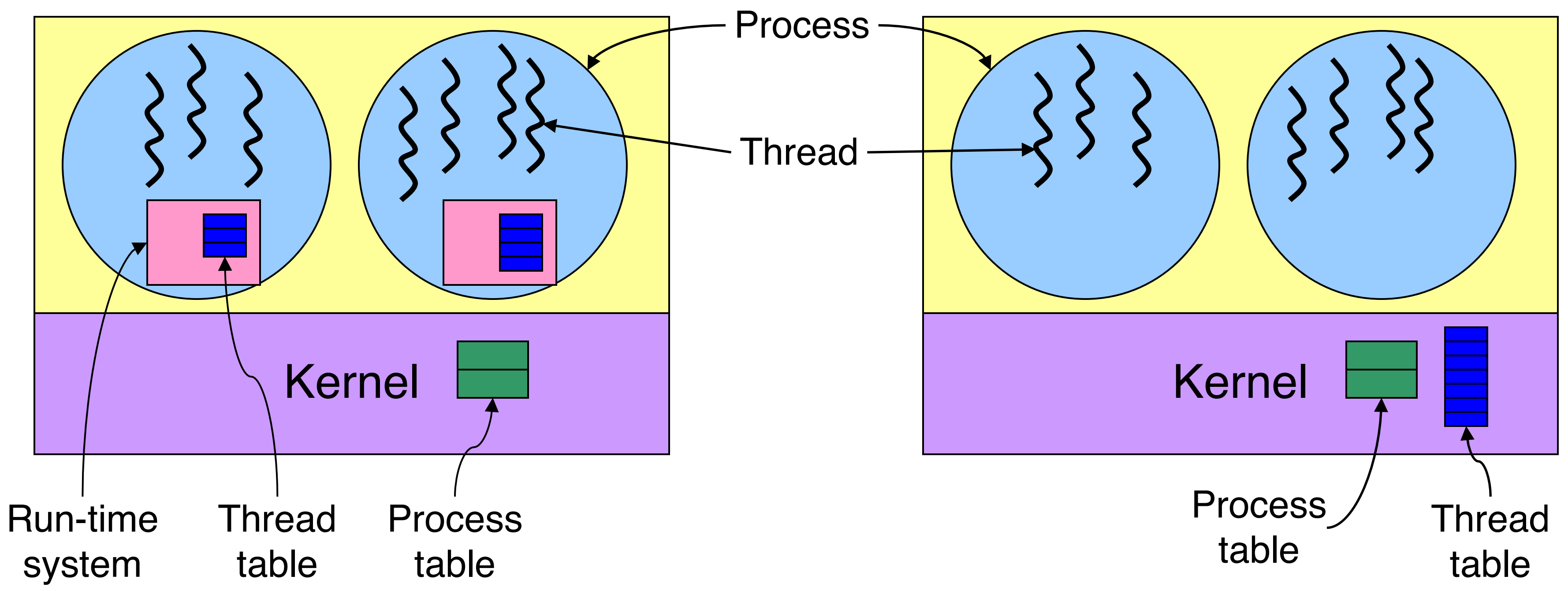
```
while(TRUE) {
  getNextRequest(&buf);
  handoffWork(&buf);
}
```

```
while(TRUE) {
  waitForWork(&buf);
  lookForPageInCache(&buf,&page);
  if(pageNotInCache(&page)) {
    readPageFromDisk(&buf,&page);
  }
  returnPage(&page);
}
```

Three ways to build a server

- Thread model
  - Parallelism
  - Blocking system calls
- Single-threaded process: slow, but easier to do
  - No parallelism
  - Blocking system calls
- Finite-state machine
  - Each activity has its own state
  - States change when system calls complete or interrupts occur
  - Parallelism
  - Nonblocking system calls
  - Interrupts

Implementing threads



User-level threads
+ No need for kernel support
- May be slower than kernel threads
- Harder to do non-blocking I/O

Kernel-level threads
+ More flexible scheduling
+ Non-blocking I/O
- Not portable

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
 . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) ==
out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

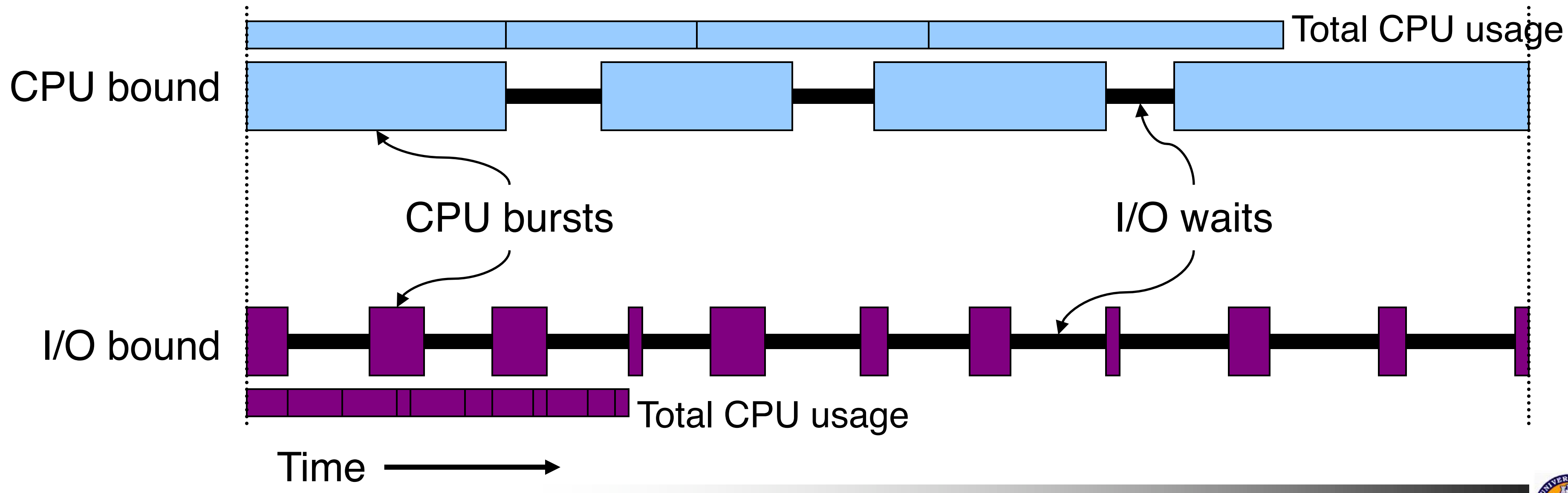
```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) %
BUFFER_SIZE;

    /* consume the item in next
consumed */
```

Scheduling

- What is scheduling?
  - Goals
  - Mechanisms
- Scheduling on batch systems
- Scheduling on interactive systems
- Other kinds of scheduling
  - Real-time scheduling

Why schedule processes?

- Bursts of CPU usage alternate with periods of I/O wait
- Some processes are *CPU-bound*: they don't make many I/O requests
- Other processes are *I/O-bound* and make many kernel requests



When are processes scheduled?

- At the time they enter the system
  - Common in batch systems
  - Two types of batch scheduling
    - Submission of a new job causes the scheduler to run
    - Scheduling only done when a job voluntarily gives up the CPU (*i.e.*, while waiting for an I/O request)
- At relatively fixed intervals (clock interrupts)
  - Necessary for interactive systems
  - May also be used for batch systems
  - Scheduling algorithms at each interrupt, and picks the next process from the pool of "ready" processes

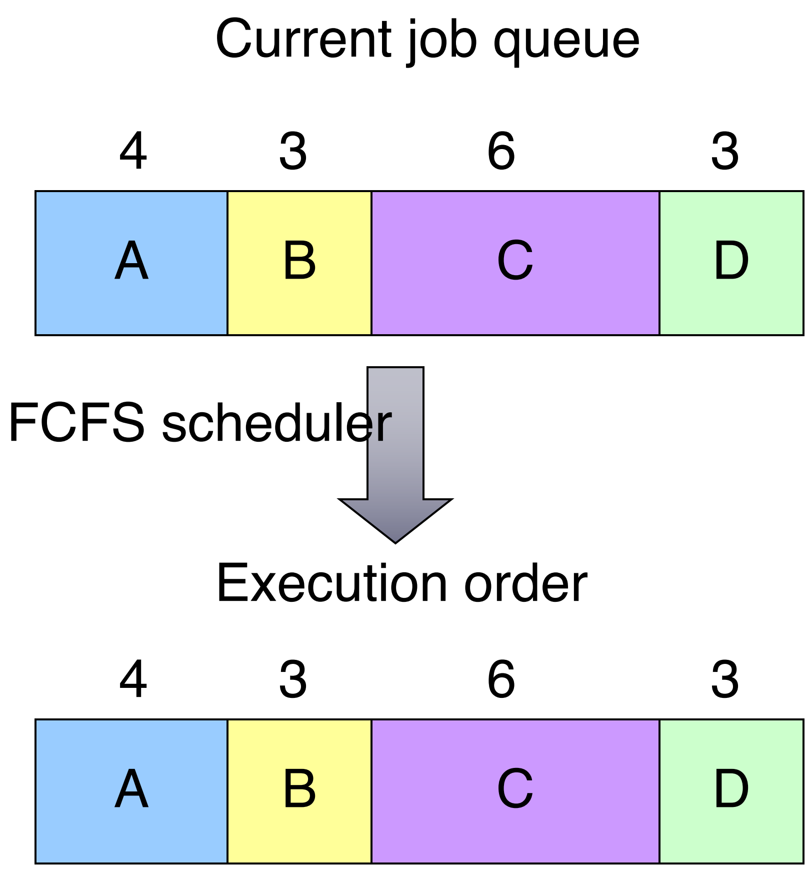
Scheduling goals

- All systems
  - Fairness: give each process a fair share of the CPU
  - Enforcement: ensure that the stated policy is carried out
  - Balance: keep all parts of the system busy
- Batch systems
  - Throughput: maximize jobs per unit time (hour)
  - Turnaround time: minimize time users wait for jobs
  - CPU utilization: keep the CPU as busy as possible
- Interactive systems
  - Response time: respond quickly to users' requests
  - Proportionality: meet users' expectations
- Real-time systems
  - Meet deadlines: missing deadlines is a system failure!
  - Predictability: same type of behavior for each time slice

Measuring scheduling performance

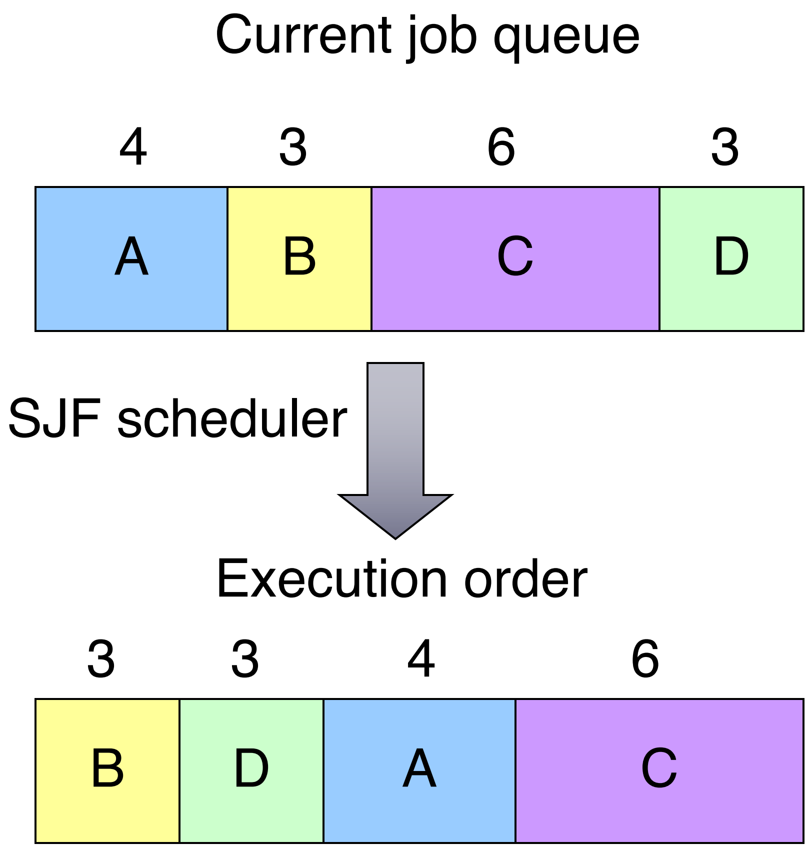
- Throughput
  - Amount of work completed per second (minute, hour)
  - Higher throughput usually means better utilized system
- Response time
  - Response time is time from when a command is submitted until results are returned
  - Can measure average, variance, minimum, maximum, …
  - May be more useful to measure time spent waiting
- Turnaround time
  - Like response time, but for batch jobs (response is the completion of the process)
- Usually not possible to optimize for *all* metrics with the same scheduling algorithm

First Come, First Served (FCFS)



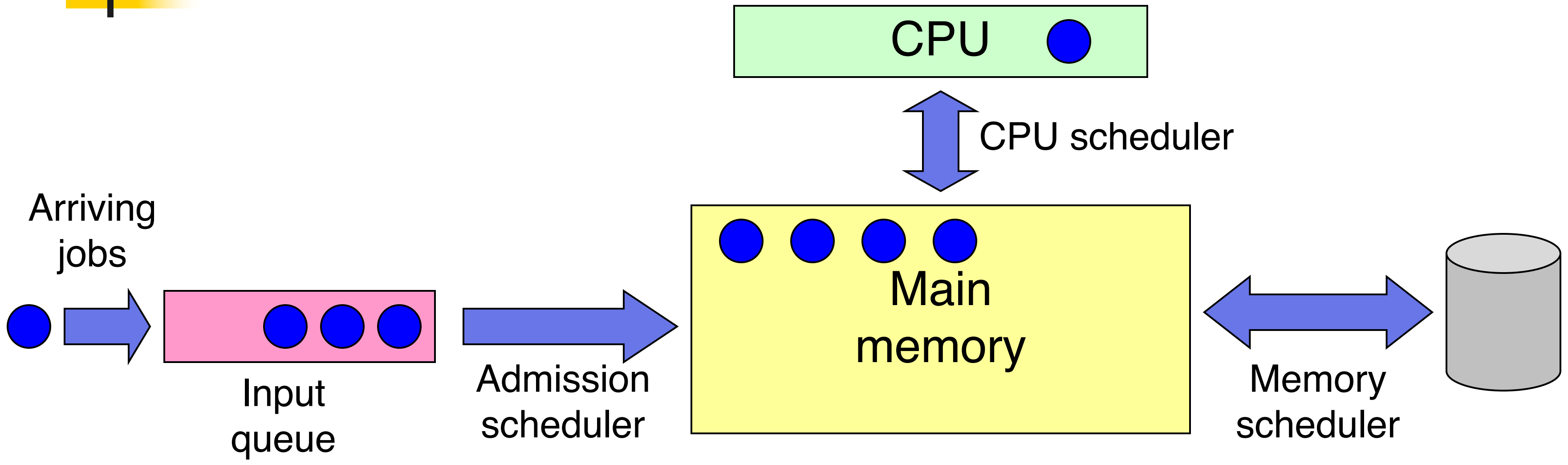
- Goal: do jobs in the order they arrive
  - Fair in the same way a bank teller line is fair
- Simple algorithm!
- Problem: long jobs delay every job after them
  - Many processes may wait for a single long job

Shortest Job First (SJF)



- Goal: do the shortest job first
  - Short jobs complete first
  - Long jobs delay every job after them
- Jobs sorted in increasing order of execution time
  - Ordering of ties doesn't matter
- Shortest Remaining Time First (SRTF): preemptive form of SJF
- Problem: how does the scheduler know how long a job will take?

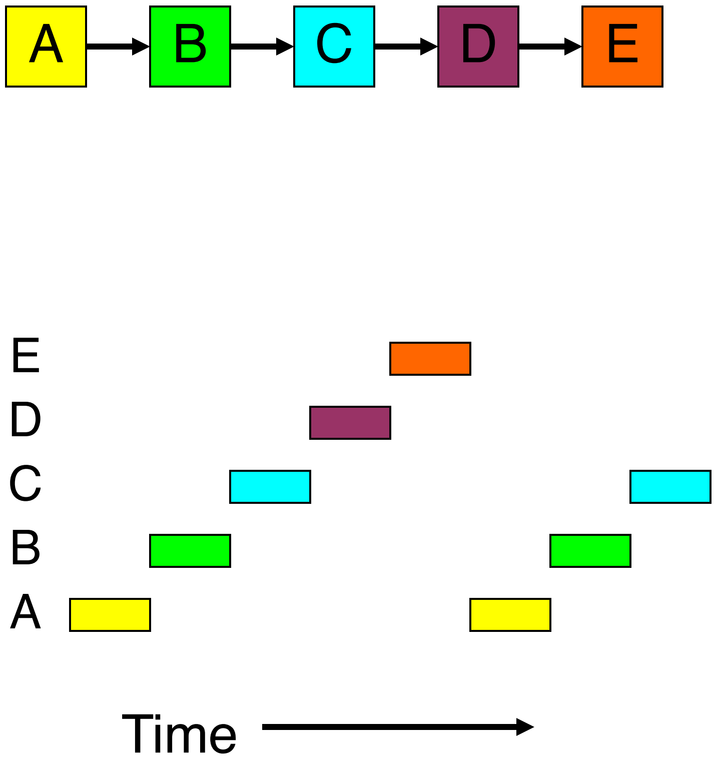
Three-level scheduling



- Jobs held in input queue until moved into memory
  - Pick "complementary jobs": small & large, CPU- & I/O-intensive
  - Jobs move into memory when admitted
- CPU scheduler picks next job to run
- Memory scheduler picks some jobs from main memory and moves them to disk if insufficient memory space

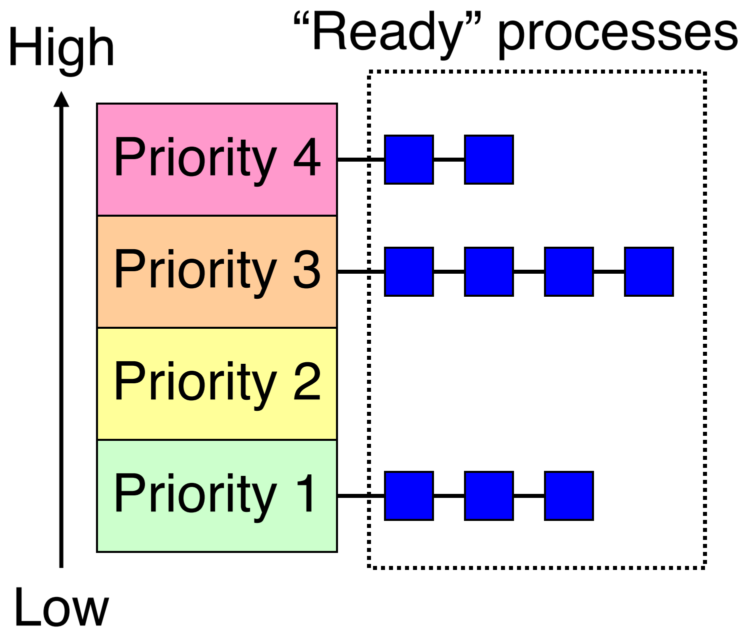
Round Robin (RR) scheduling

- Round Robin scheduling
  - Give each process a fixed time slot (*quantum*)
  - Rotate through "ready" processes
  - Each process makes some progress
- What's a good quantum?
  - Too short: many process switches hurt efficiency
  - Too long: poor response to interactive requests
  - Typical length: 10–50 ms



Priority scheduling

- Assign a priority to each process
  - "Ready" process with highest priority allowed to run
  - Running process may be interrupted after its quantum expires
- Priorities may be assigned dynamically
  - Reduced when a process uses CPU time
  - Increased when a process waits for I/O
- Often, processes grouped into multiple queues based on priority, and run round-robin per queue



Shortest process next

- Run the process that will finish the soonest
  - In interactive systems, job completion time is unknown!
- Guess at completion time based on previous runs
  - Update estimate each time the job is run
  - Estimate is a combination of previous estimate and most recent run time
- Not often used because round robin with priority works so well!

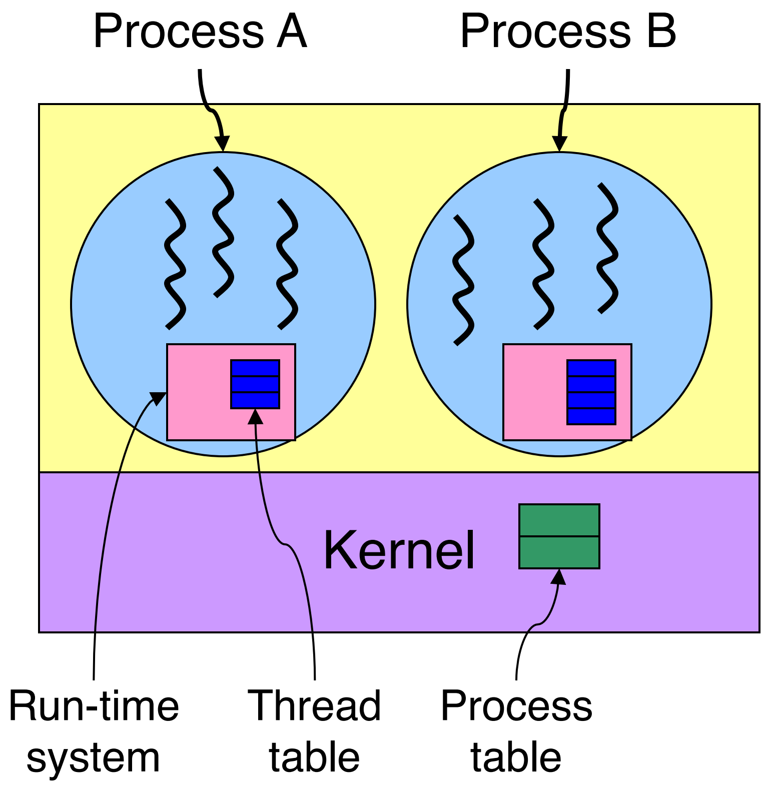
Lottery scheduling

- Give processes "tickets" for CPU time
  - More tickets => higher share of CPU
- Each quantum, pick a ticket at random
  - If there are *n* tickets, pick a number from 1 to *n*
  - Process holding the ticket gets to run for a quantum
- Over the long run, each process gets the CPU *m*/*n* of the time if the process has *m* of the *n* existing tickets
- Tickets can be transferred
  - Cooperating processes can exchange tickets
  - Clients can transfer tickets to server so it can have a higher priority

Policy versus mechanism

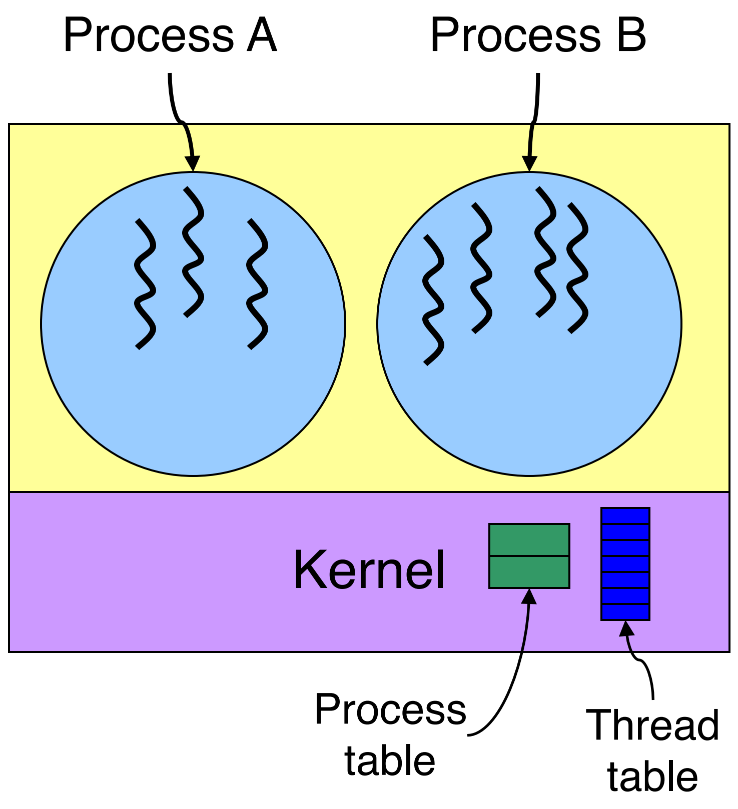
- Separate what ***may*** be done from ***how*** it is done
  - Mechanism allows
    - Priorities to be assigned to processes
    - CPU to select processes with high priorities
  - Policy set by what priorities are assigned to processes
- Scheduling algorithm parameterized
  - Mechanism in the kernel
  - Priorities assigned in the kernel or by users
- Parameters may be set by user processes
  - Don’t allow a user process to take over the system!
  - Allow a user process to voluntarily lower its own priority
  - Allow a user process to assign priority to its threads

Scheduling user-level threads



- Kernel picks a process to run next
- Run-time system (at user level) schedules threads
  - Run each thread for less than process quantum
  - Example: processes get 40ms each, threads get 10ms each
- Example schedule: A1,A2,A3,A1,B1,B3,B2,B3
- Not possible: A1,A2,B1,B2,A3,B3,A2,B1

Scheduling kernel-level threads



- Kernel schedules each thread
  - No restrictions on ordering
  - May be more difficult for each process to specify priorities
- Example schedule: A1,A2,A3,A1,B1,B3,B2,B3
- Also possible: A1,A2,B1,B2,A3,B3,A2,B1