CMPT 300

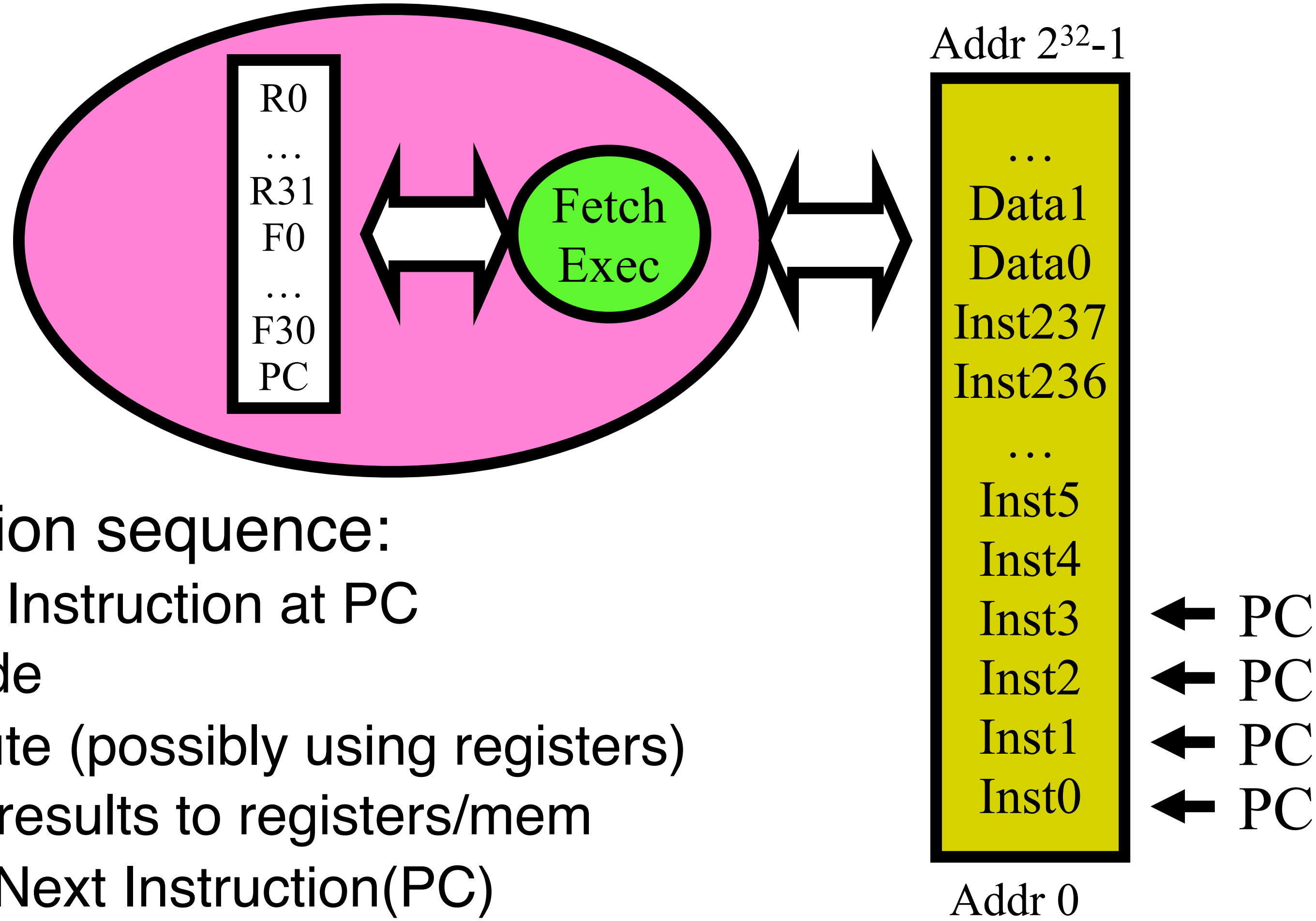
Introduction to Operating Systems

Operating Systems

Processes & Threads

Acknowledgement: Many thanks to Berkeley's CS162 and UT-Austin's CS 372

Review: Instruction Execution



Execution sequence:

- Fetch Instruction at PC
- Decode
- Execute (possibly using registers)
- Write results to registers/mem
- PC = Next Instruction(PC)
- Repeat

Concurrency

- A "thread" of execution is an independent Fetch/Decode/Execute loop
  - a sequential instruction stream
- Uni-programming: *one thread at a time*
  - MS/DOS, early Macintosh, Batch processing
  - Easier for operating system builder
  - Get rid concurrency by defining it away
- Multi-programming: *more than one thread*
  - Multics, UNIX/Linux, OS/2, Windows NT/2000/XP, Mac OS X

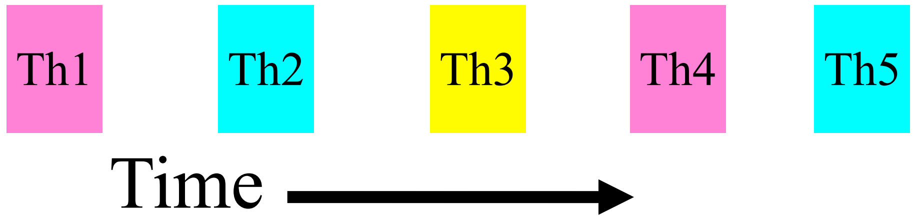
Concurrency vs. Parallelism

- Concurrency is from the application perspective
  - The application software consists of multiple threads of execution
- Parallelism is from the hardware perspective
  - The hardware platform consists of multiple CPUs
- A concurrent application can be executed on a single or multi-CPU hardware platform

The Basic Problem of Concurrency

- Must provide illusion to each application thread that it has exclusive access to the CPU
- Each thread is unaware of existence of other threads
- OS has to coordinate multiple threads

Multithreading



- How to provide the illusion of multiple CPUs with a single physical CPU?
  - Multiplex in time!
- Each thread has a data structure (TCB) to hold:
  - Program Counter (PC), Stack Pointer (SP), Register values (Integer, Floating point…)
- How switch from one thread to the next?
  - Save PC, SP, and registers in current TCB
  - Load PC, SP, and registers from new TCB
- What triggers switch?
  - Timer, voluntary yield, I/O…

Two Types of Resources

- CPU is an active resource that can be used by only one runtime entity
  - Can be multiplexed in time (scheduled)

- Memory is a passive resource that can be shared among multiple runtime entities
  - Can be multiplexed in space (allocated)

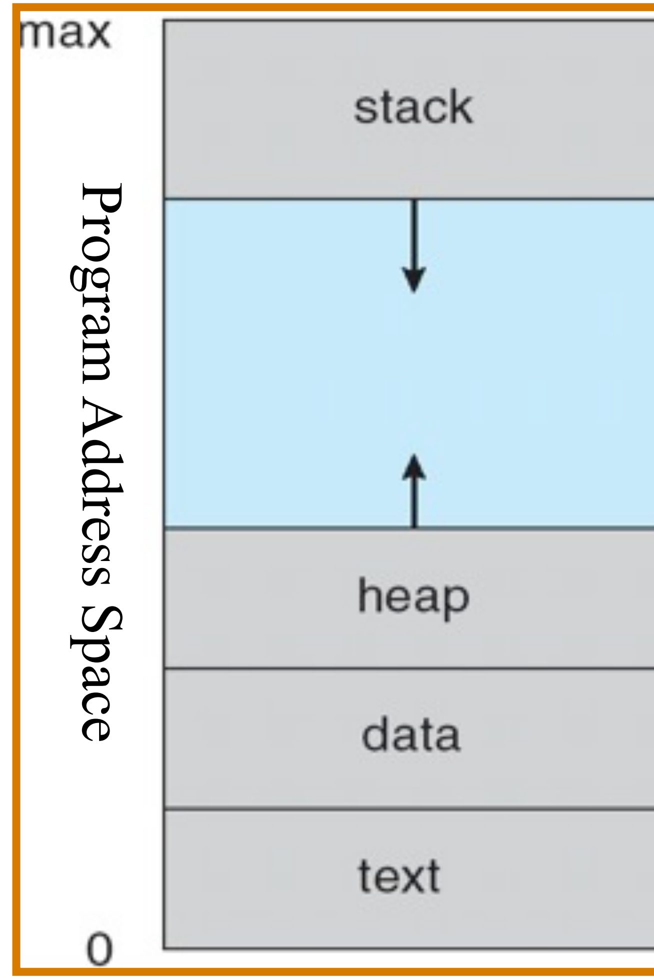
How to Protect Tasks, from each other?

- Protection of memory
  - Each task does not have access to all memory
- Protection of I/O devices
  - Each task does not have access to every device
- Protection of CPU
  - Timer interrupts to enforce periodic preemption
  - user code cannot disable timer

- **"Task" here refers to a runtime entity, can be either a thread or a process**

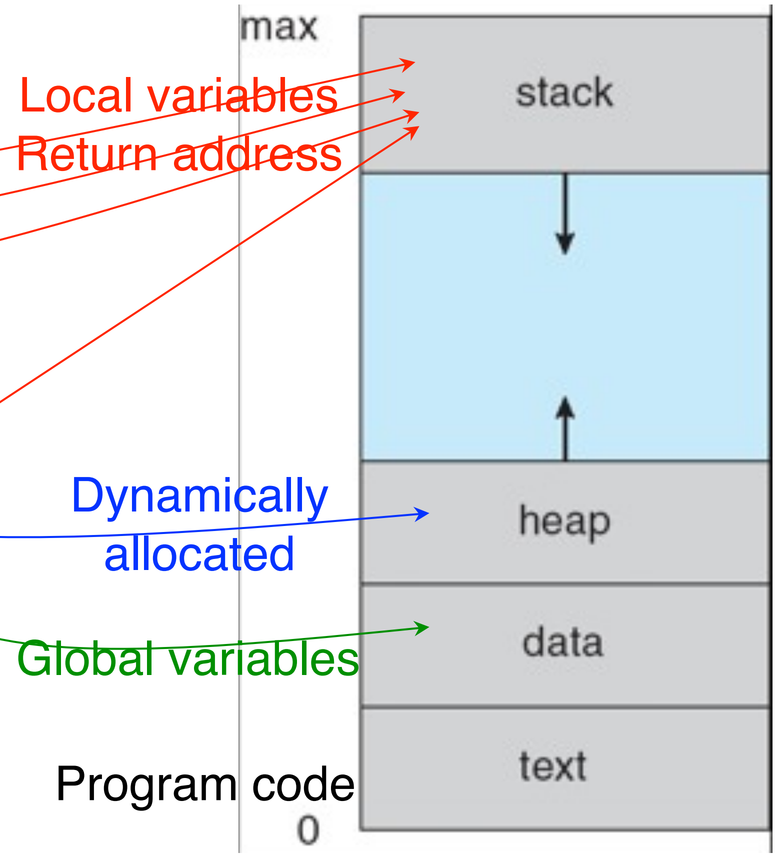
Review: Address Space

- Address space ⇒ set of accessible addresses + state associated with them (contents of the memory addresses):
  - For a 32-bit processor there are 2^{32} = 4 billion addresses



Review: a Process in Memory

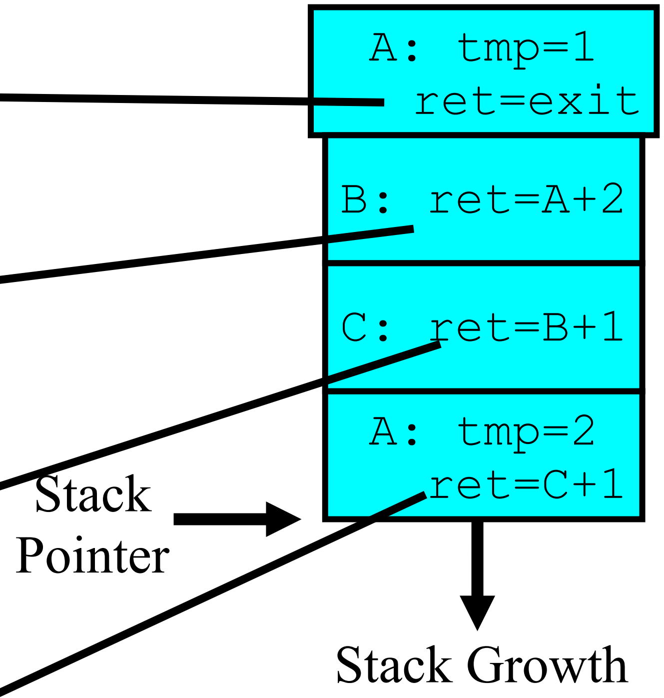
```
int global = 0;
int main (int arg)
{
        float  local;
        char *ptr;

        ptr = malloc(100);
        local = 0;
        local += 10*5;
        …..
        ….
        foo();
        …. /* return addr */
        ….
        return 0;
}
```



("text" is also called "code", referring to the binary executable code)

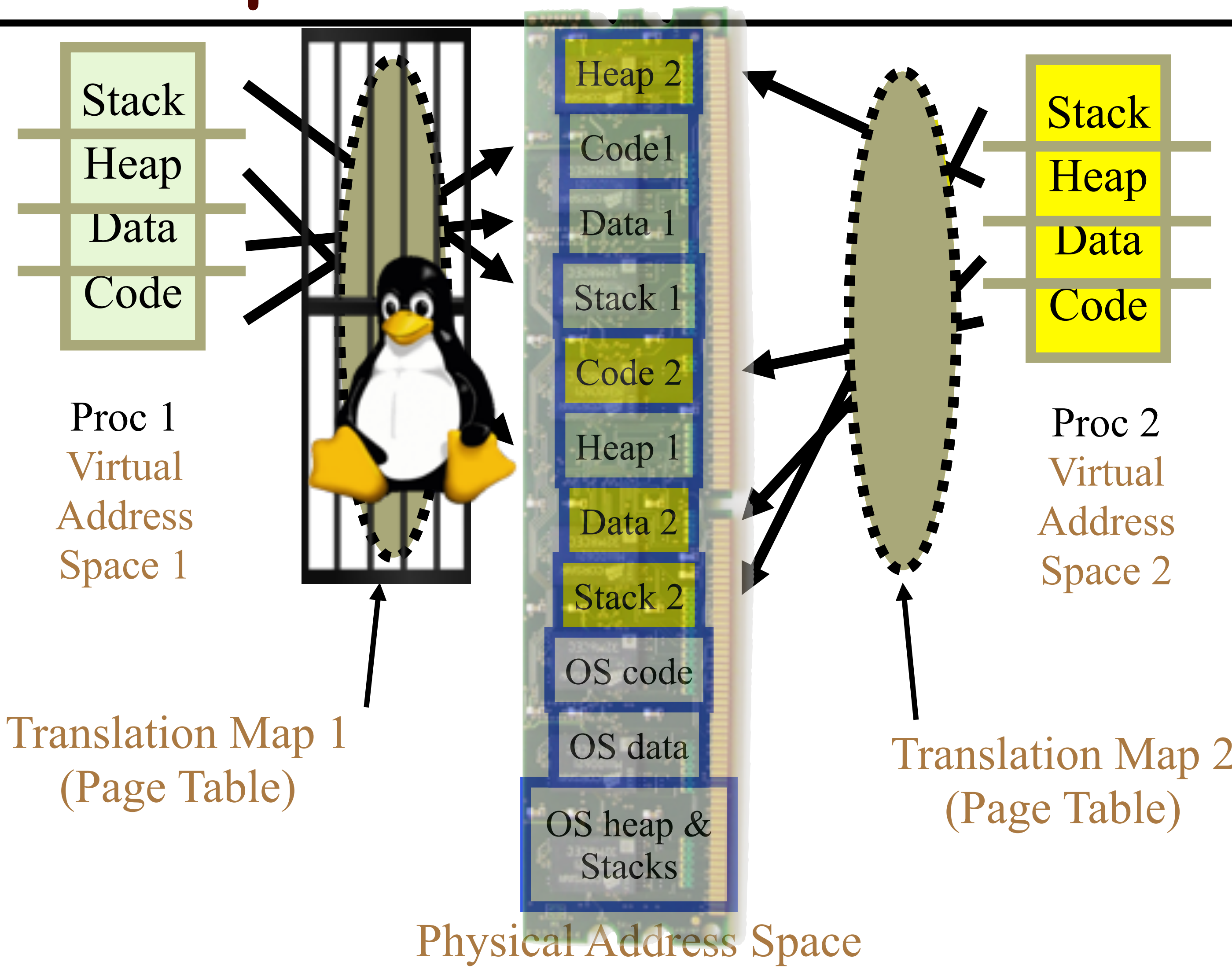
Review: Execution Stack

```
A(1);
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
```



- Stack holds temporary data and return address; Permits recursive execution
- A, B, C are procedure names, but also refer to memory address for procedure code when computing return address

Virtual Memory Provides Separate Address Space for Each Process



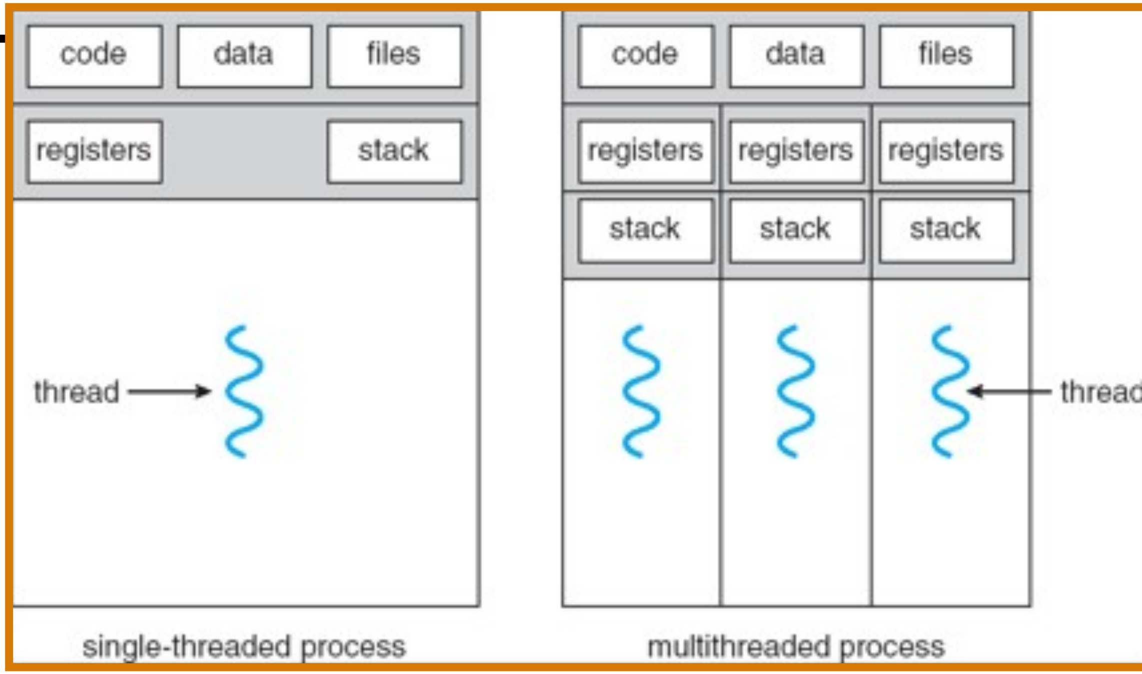
12

Processes vs. Threads

- Different procs. see separate addr. spaces
  - good for protection, bad for sharing

- All threads in the same process share
  - Address space: each thread can access the data of other thread (good for sharing, bad for protection)
  - I/O state (i.e. file descriptors)

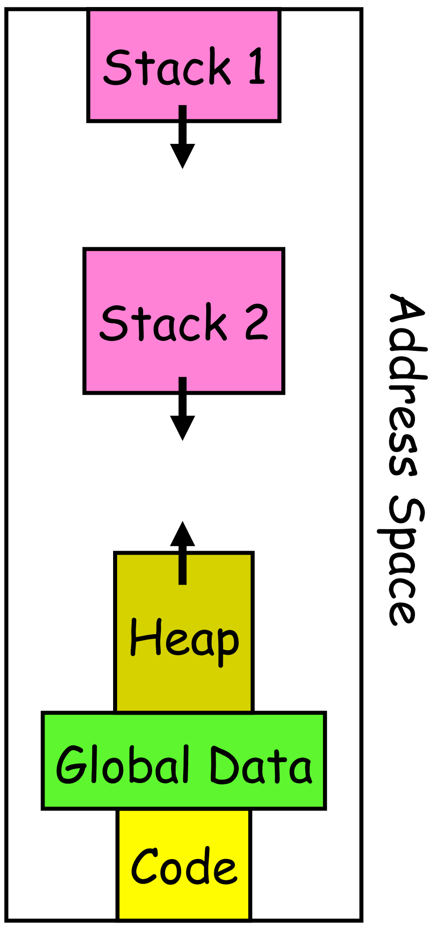
Single and Multithreaded Processes



- Threads encapsulate concurrency: “active” component
- Processes (address spaces) encapsulate memory protection:
- Each process should have at least one thread (at least one main() as the entry point of thread execution)

Address Space of a 2-Threaded Process

- It has two stacks
- Must make sure that the stacks and heap do not grow into each other, causing stack overflow



Classification

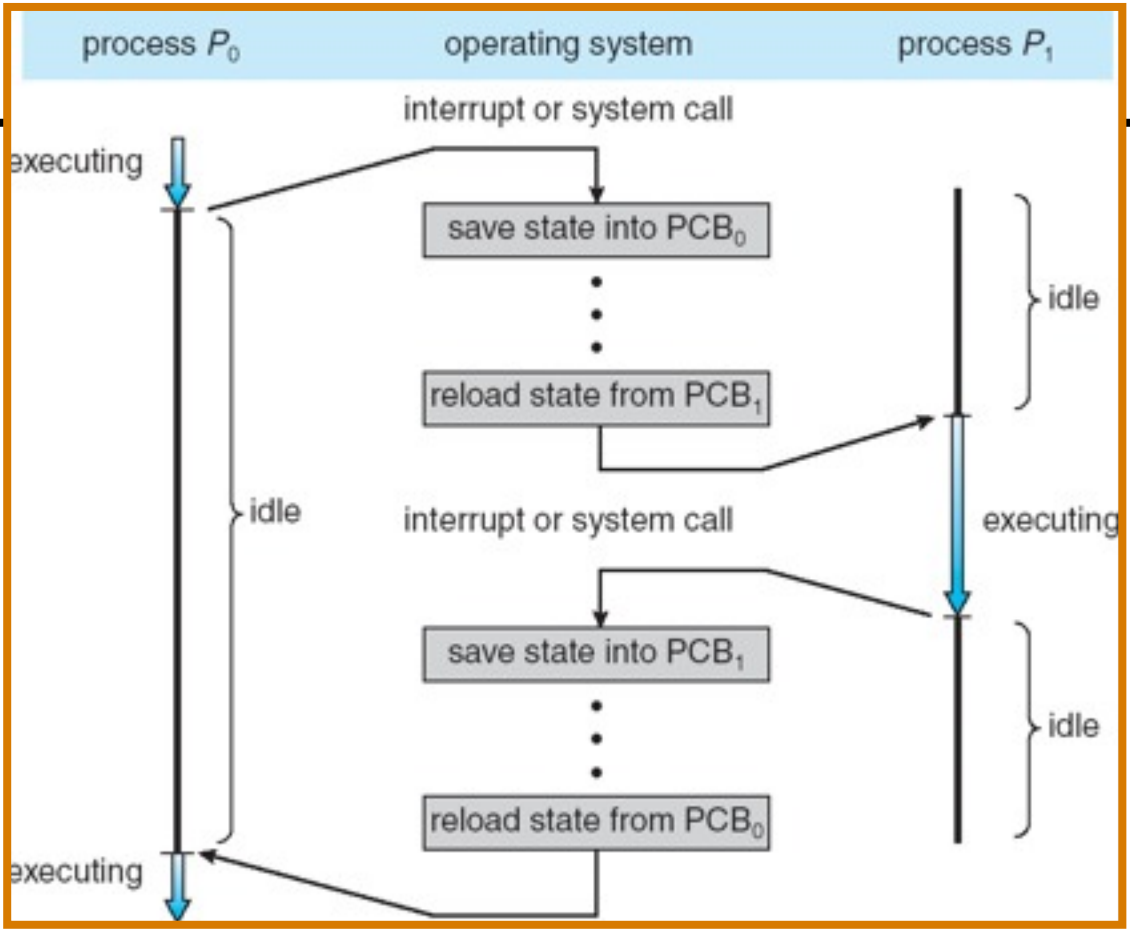
# threads Per process: / # of processes:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (QNX, VxWorks,etc)	Mach, OS/2, Linux Win NT,XP,7, Solaris, HP-UX, OS X

- Virtual memory mechanism requires HW support (Memory Management Unit) that may not be available in small embedded processors, hence embedded systems are often single-process

Traditional UNIX Process

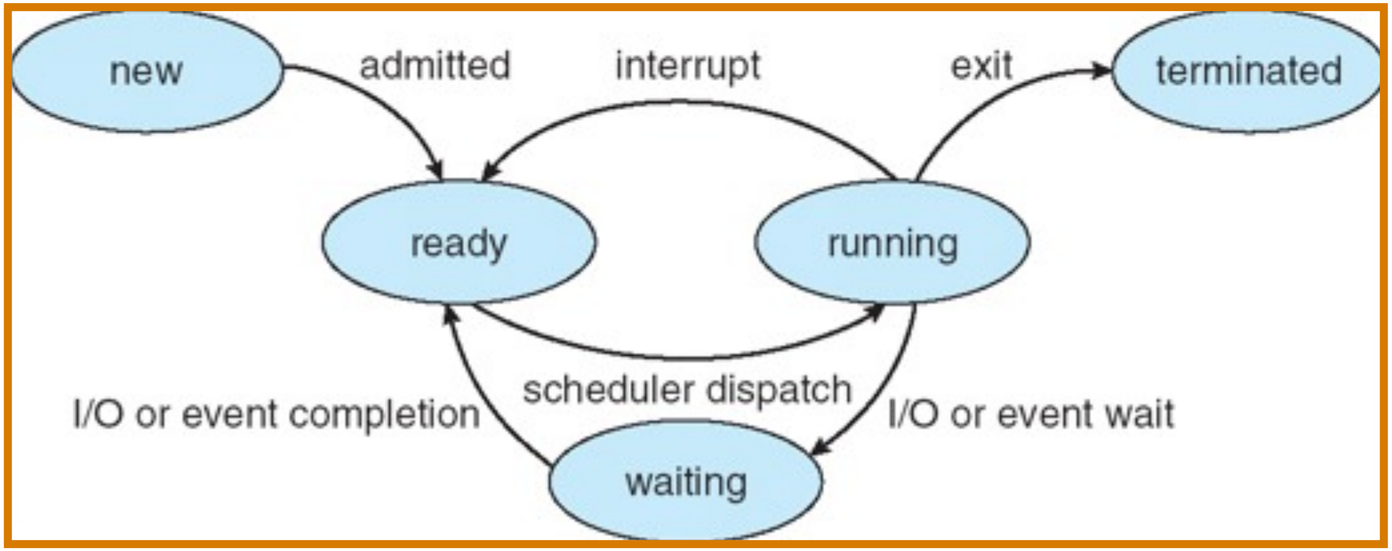
- Each process has a single thread
  - Called a “heavy-weight process”
- Similar to Thread Control Block, each process has a Process Control Block (PCB) that holds the process-related context.

CPU Switch



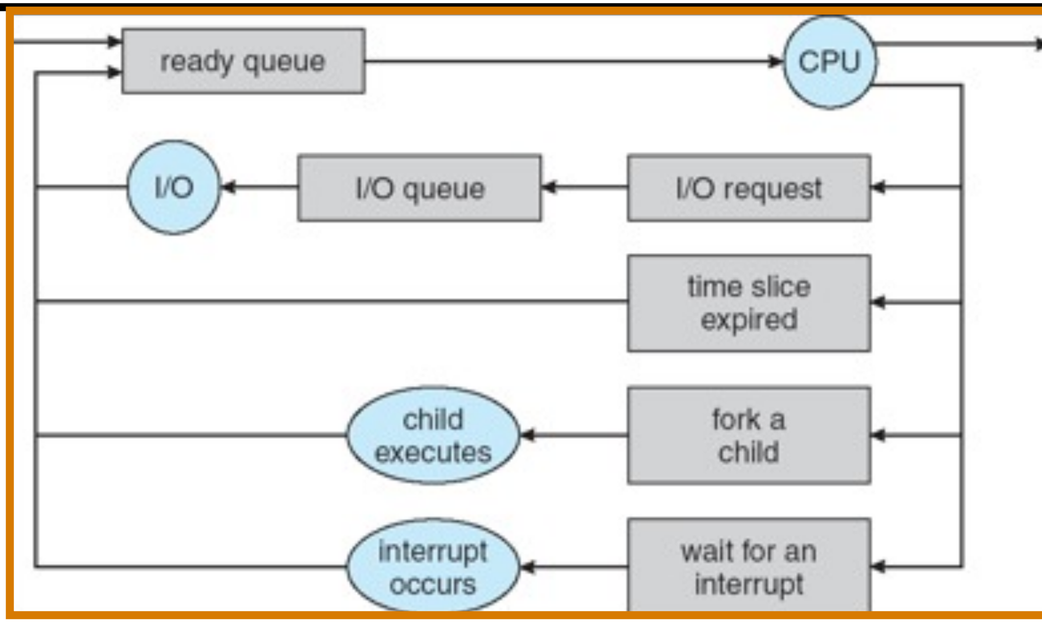
- ➡ Process context-switch has relatively large overhead
  - manipulating the page table ; copying memory
- ➡ Thread context-switch is similar

Process State Machine



- As a process executes, it changes *state*
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution
- See animation
- (This state machine also applies to threads)

Process Scheduling



- Processes (in actual implementation, their PCBs) move from queue to queue as they change state
  - Many scheduling algorithms possible
- (also applies to threads, with TCBs instead of PCBs)

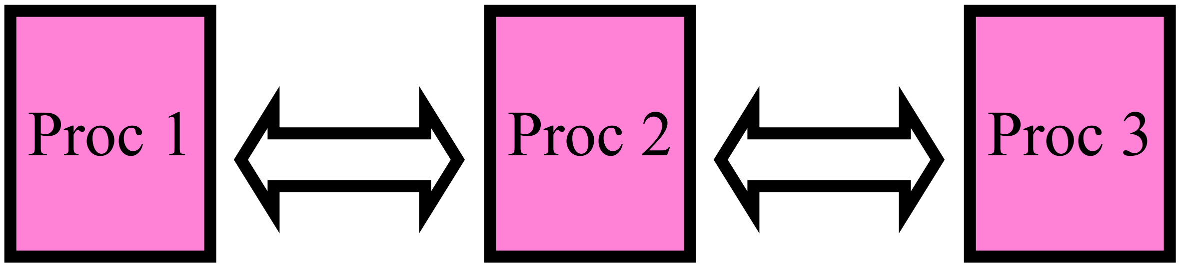
Motivation for Multi-Threading

- Why have multiple threads per process?
  - May need concurrency for a single application, and processes are very expensive – to start, switch between, and to communicate between
  - Communication between processes is not as convenient as between threads in the same process

What Does it Take to Create a Process?

- Must construct new PCB
  - Inexpensive
- Must set up new page tables for address space
  - More expensive
- Copy data from parent process (Unix `fork()`)
  - Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
  - Originally *very* expensive
  - Much less expensive with “copy-on-write” (initially shared; make a copy only when an address is written to)
- Copy I/O state (file handles, etc)

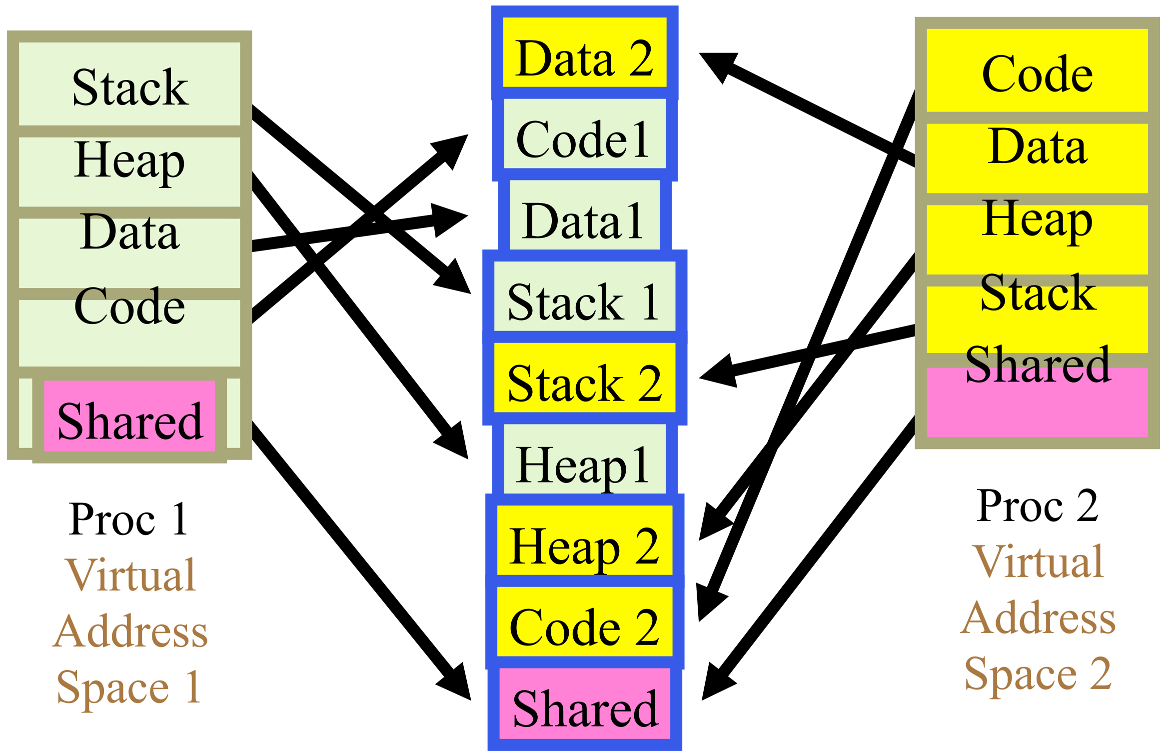
Multiple Processes Collaborate on a Task



- Creation/memory Overhead
- Context-Switch Overhead

- Need Communication mechanism:
  - Separate Address Spaces Isolates Processes
  - Shared-Memory Mapping
    - Mapping virtual addresses to common physical address
    - Read and Write through memory
  - Message Passing
    - `send()` and `receive()` messages
    - Works either locally or across a network

Shared Memory Communication



- Communication occurs by reading/writing to shared address page
- Establishing shared memory involves manipulating the translation map, hence can be expensive

Message-Based Inter-Process Communication (IPC)

- Mechanism for processes to communicate with each other without shared memory
- IPC facility provides two operations:
  - `send(message)` – message size fixed or variable
  - `receive(message)`
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive

Modern UNIX Process

- Multithreading: a single process consists of multiple concurrent threads
- A thread is sometimes called a "Lightweight process"
  - Thread creation and context-switch are much more efficient than process creation and context-switch
  - Inter-thread communication is via shared memory, since threads in the same process share the same address space

Why Use Processes?

Consider a Web server

get network message (URL) from client

create child process, send it URL

Child

fetch URL data from disk

compose response

send response

- If server has configuration file open for writing
  - Prevent child from overwriting configuration
- How does server know child serviced request?
  - Need return code from child process

The Genius of Separating Fork/Exec

- Life with `CreateProcess(filename);`
  - But I want to close a file in the child. `CreateProcess(filename, list of files);`
  - And I want to change the child's environment. `CreateProcess(filename, CLOSE_FD, new_envp);`
  - Etc. (and a very ugly etc.)
- **`fork()`** = split this process into 2 (new PID)
  - Returns 0 in child
  - Returns pid of child in parent
- **`exec()`** = overlay this process with new program (PID does not change)

The Genius of Separating Fork/Exec

- Decoupling fork and exec lets you do anything to the child's process environment without adding it to the CreateProcess API.

```
int ppid = getpid();        // Remember parent's pid
fork();                     // create a child
if(getpid() != ppid) {                       // child continues here
    // Do anything (unmap memory, close net connections…)
    exec("program", argc, argv0, argv1, …);
}
```

- fork() creates a child process that inherits:
  - identical copy of all parent's variables & memory
  - identical copy of all parent's CPU registers (except one)
- Parent and child execute at the same point after **fork()** returns:
  - by convention, for the child, fork() returns 0
  - by convention, for the parent, fork() returns the process identifier of the child
  - fork() return code a convenience, could always use getpid()

Program Loading: exec()

- The exec() call allows a process to “load” a different program and start execution at main (actually _start).
- It allows a process to specify the number of arguments (argc) and the string argument array (argv).
- If the call is successful
  - it is the same process …
  - but it runs a different program !!
- Code, stack & heap is overwritten
  - Sometimes memory mapped files are preserved.

What creates a process?

1. Fork
2. Exec
3. Both

General Purpose Process Creation

In the parent process:

```
main()
…
int ppid = getpid();                    // Remember parent's pid
fork();                                 // create a child
if(getpid() != ppid) {                  // child continues here
    exec_status = exec("calc", argc, argv0, argv1, …);
    printf("Why would I execute?");
}
else {                                  // parent continues here
    printf("Who's your daddy?");
    …
   child_status = wait(pid);
}
```

A shell forks and then execs a calculator

```
int pid = fork();
if(pid == 0) {
  close(".history");
  exec("/bin/calc");
} else {
  wait(pid);
```

```
int pid = fork();
if(pid == 0) {
  close(".history");
  exec("/bin/calc");
} else {
  wait(pid);
```

```
int calc_main(){
int q = 7;
  do_init();
  ln = get_input();
  exec_in(ln);
```

USER

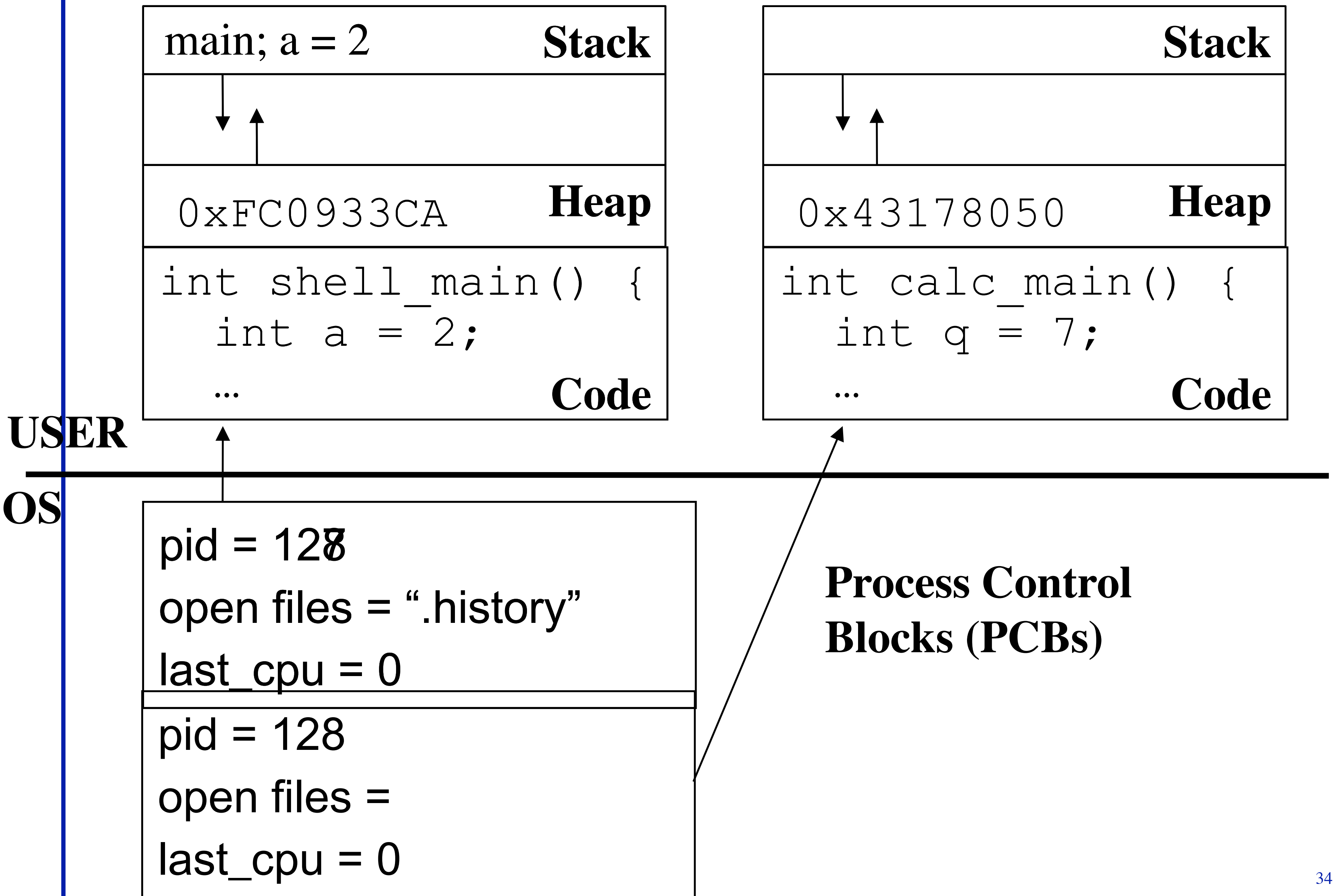
OS

pid = 127
open files = ".history"
last_cpu = 0

pid = 128
open files =
last_cpu = 0

Process Control Blocks (PCBs)

A shell forks and then execs a calculator



At what cost, fork()?

- Simple implementation of fork():
  - allocate memory for the child process
  - copy parent's memory and CPU registers to child's
  - *Expensive* !!
- In 99% of the time, we call exec() after calling fork()
  - the memory copying during fork() operation is useless
  - the child process will likely close the open files & connections
  - overhead is therefore high
- vfork()
  - a system call that creates a process "without" creating an identical memory image
  - child process should call exec() almost immediately
  - Unfortunate example of implementation influence on interface
    - Current Linux & BSD 4.4 have it for backwards compatibility
  - Copy-on-write to implement fork avoids need for vfork

Orderly Termination: exit()

- After the program finishes execution, it calls *exit*()
- This system call:
  - takes the "result" of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the zombie/defunct state
    - If not, it deallocates all data structures, the process is dead
  - cleans up all waiting zombies
- Process termination is the ultimate garbage collection (resource reclamation).

The wait() System Call

- A child program returns a value to the parent, so the parent must arrange to receive that value

- The wait() system call serves this purpose
  - it puts the parent to sleep waiting for a child's result
  - when a child calls exit(), the OS unblocks the parent and returns the value passed by exit() as a result of the wait call (along with the pid of the child)
  - if there are no children alive, wait() returns immediately
  - also, if there are zombies waiting for their parents, wait() returns one of the values immediately (and deallocates the zombie)

Tying it All Together: The Unix Shell

```
while(! EOF) {
read input
handle regular expressions
int pid = fork();                    // create a child
if(pid == 0) {                       // child continues here
    exec("program", argc, argv0, argv1, …);
}
else {                               // parent continues here
…
}
```

- Translates <CTRL-C> to the kill() system call with SIGKILL
- Translates <CTRL-Z> to the kill() system call with SIGSTOP
- Allows input-output redirections, pipes, and a lot of other stuff that we will see later

A Single-Threaded Program

- Consider the following C program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("clist.text");
}
```

- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

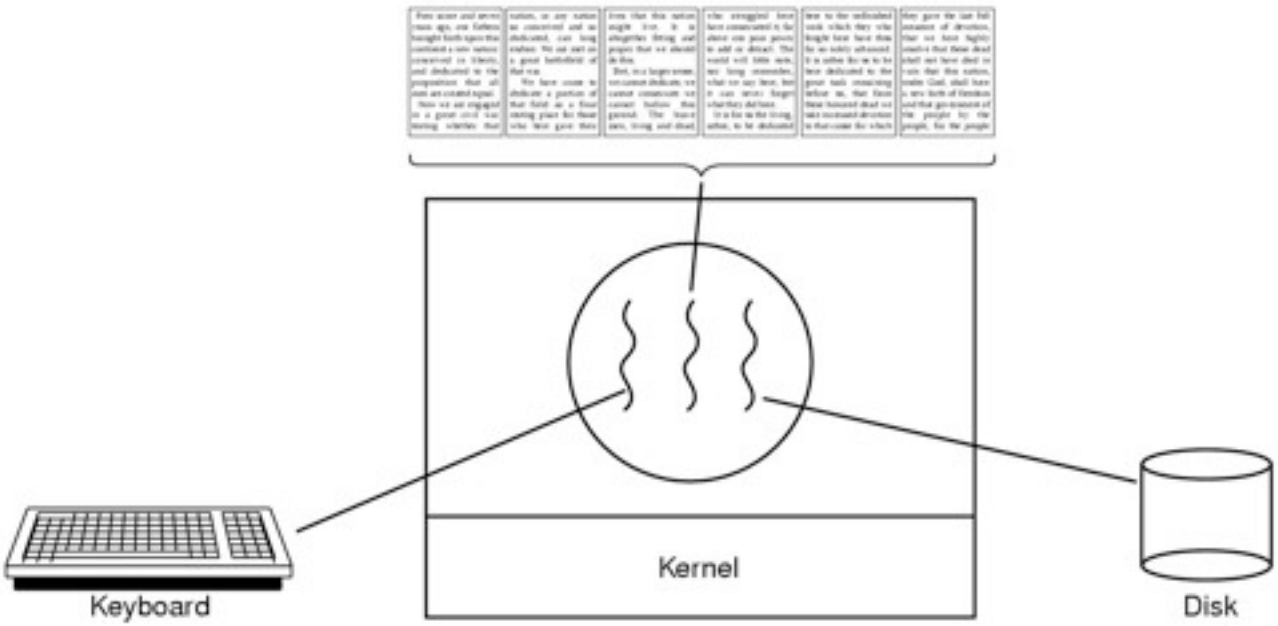
Use of Threads

- Version of program with Threads:

```
main() {
    CreateThread(ComputePI(“pi.txt”));
    CreateThread(PrintClassList(“clist.text”));
}
```

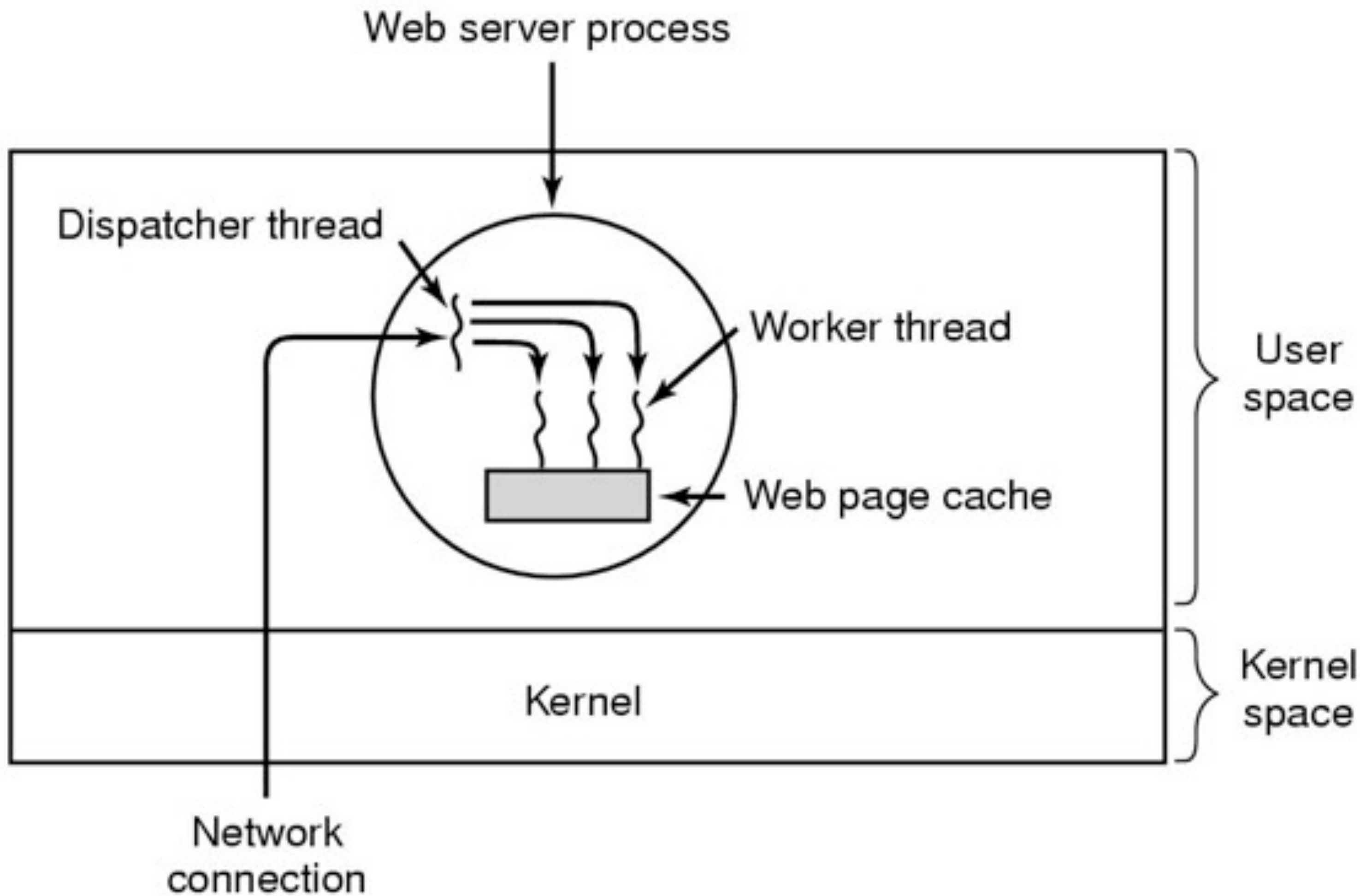
- “CreateThread” starts independent threads running given procedure name

Example: a Multi-Threaded Text Editor



- One thread for handling keyboard input; one for handling graphical user interface; one for handling disk IO
- 3 threads must collaborate closely and share data

Example: a Multi-Threaded Database Server



Database Server Implementation

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

- (a) Dispatcher thread. (b) Worker thread.
- A single dispatcher thread hands off work to a fixed-size pool of worker threads.
- The alternative of spawning a new thread for each request may result in an unbounded number of threads; it also incurs thread creation overhead for each request.
- By creating a fixed-size pool of threads at system initialization time, these

POSIX Thread API

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

POSIX (Portable Operating System Interface for Unix) is a family of related standards specified by the IEEE to define the API for software compatible with variants of the Unix operating system,

A Multithreaded POSIX Program

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS    10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d0, tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d0, i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d0, status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

- What is the output of this program?
  - Depends on the OS scheduling algorithm
  - Likely prints out thread IDs in sequence

Summary

- Processes have two aspects
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU:
  - Such context switching may be voluntary (`yield()`, I/O operations) or involuntary (timer, other interrupts)
  - Save and restore of either PCB or TCP
- Protection accomplished restricting access:
  - Virtual Memory isolates processes from each other

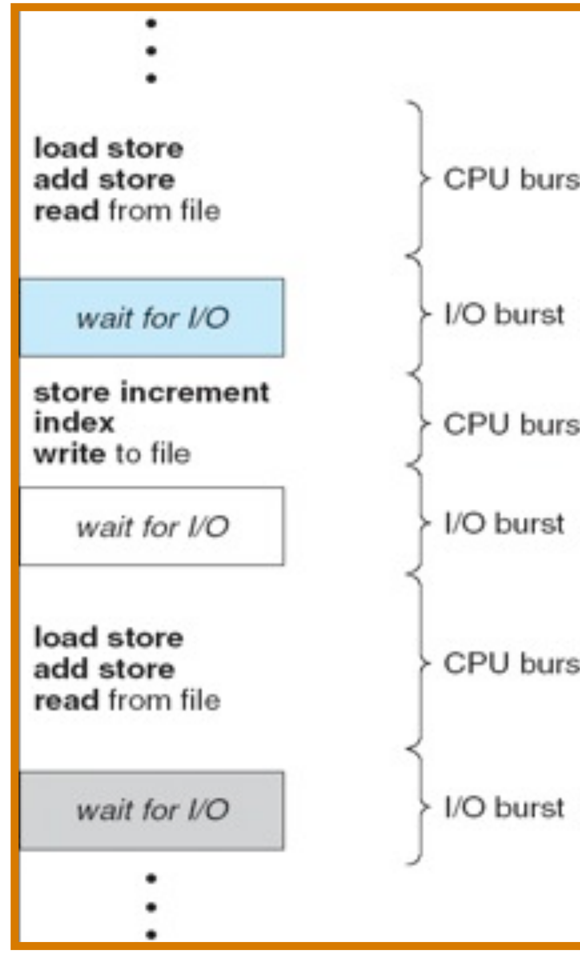
CMPT 300

Introduction to Operating Systems

Scheduling

CPU/IO Bursts

- A typical process alternates between bursts of CPU and I/O
  - It uses the CPU for some period of time, then does I/O, then uses CPU again



CPU-Bound vs. IO-Bound Processes

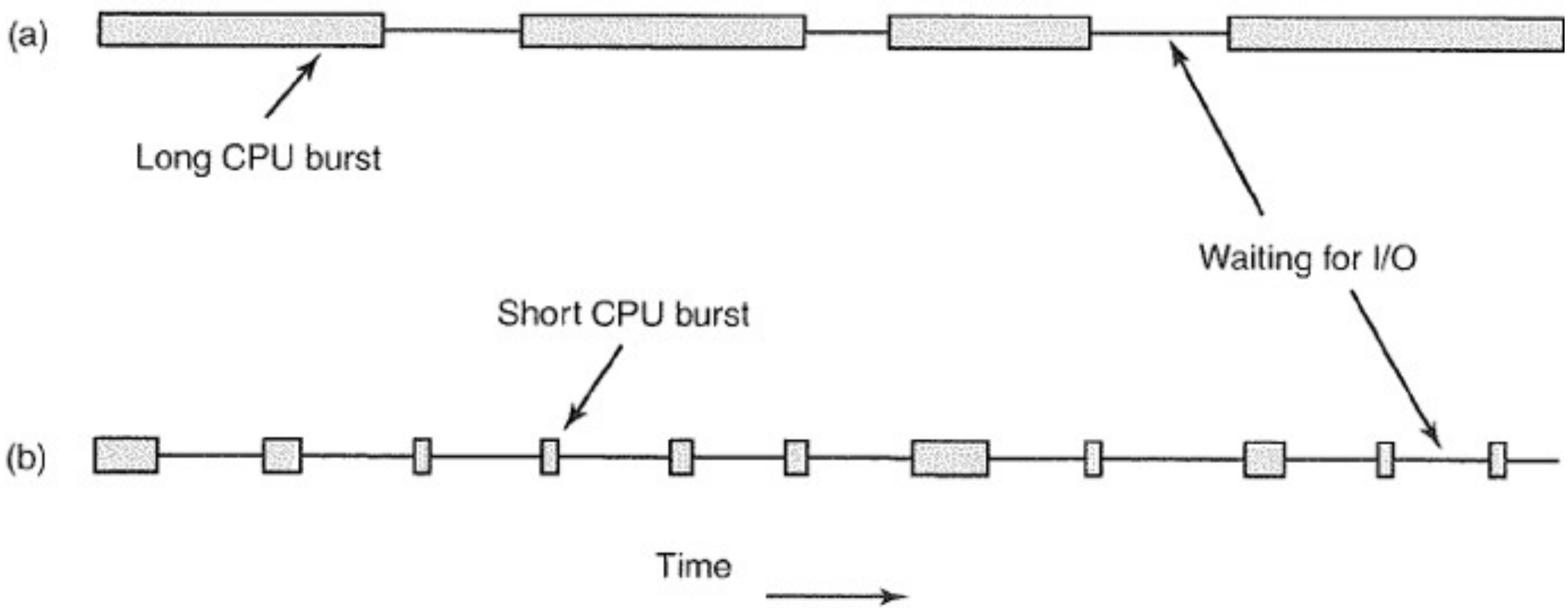
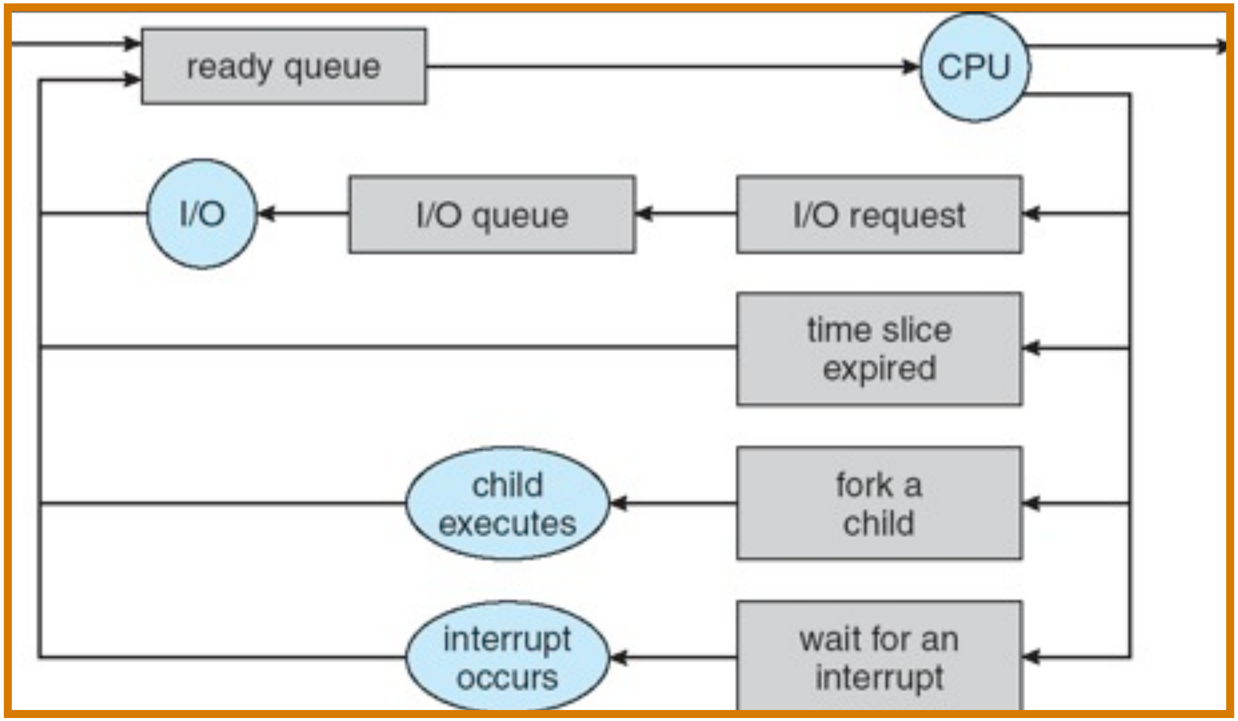


Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

Terminology

- By convention, we use the term “process” in this section, assuming that each process is single-threaded
  - The scheduling algorithms can be applied to threads as well
- The term “job” is often used to refer to a CPU burst, or a compute-only process

CPU Scheduling

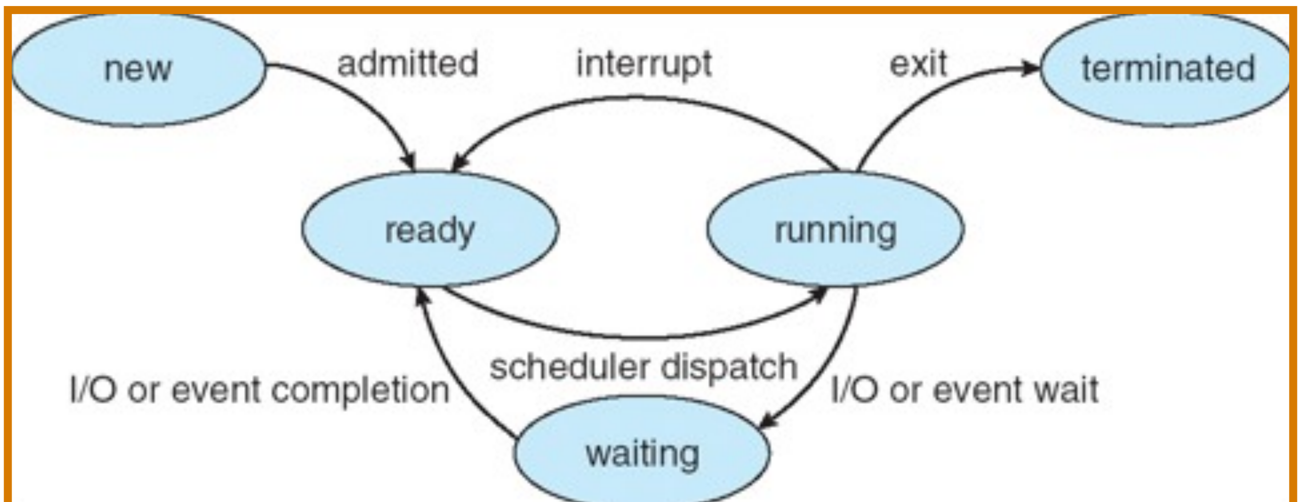


- When multiple processes are ready, the scheduling algorithm decides which one is given access to the CPU

Preemptive vs. Non-Preemptive Scheduling

- With non-preemptive scheduling, once the CPU has been allocated to a process, it keeps the CPU yield() or I/O.

- With preemptive scheduling, the OS can forcibly remove



Scheduling Criteria

CPU utilization – percent of time when CPU is busy

Throughput – # of processes that complete their execution per time unit

Response time – amount of time to finish a particular process

Waiting time – amount of time a process waits in the ready queue before it starts execution

Scheduling Goals

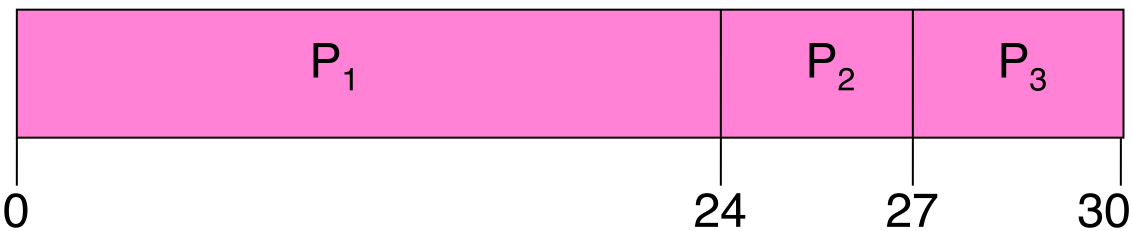
- Different systems may have different requirements
  - Maximize CPU utilization
  - Maximize Throughput
  - Minimize Average Response time
  - Minimize Average Waiting time
- Typically, these goals cannot be achieved simultaneously by a single scheduling algorithm

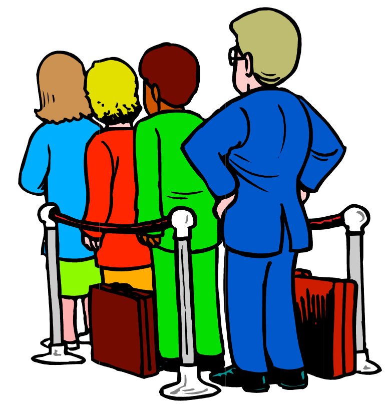
Scheduling Algorithms Considered

- First-Come-First-Served (FCFS) Scheduling
- Round-Robin (RR) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Priority-Based Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback-Queue Scheduling
- Lottery Scheduling

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also called “First In, First Out” (FIFO)
  - Run each job to completion in order of arrival
- Example: P1: 24 P2: 3 P3: 3
  The Gantt Chart for the schedule is:

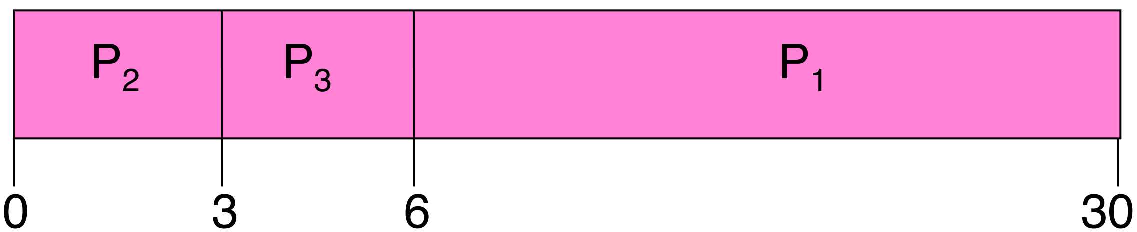




  - Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average response time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short jobs queue up behind long job

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that jobs arrive in the order: P_2 , P_3 , P_1:



  - Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
  - Avg. waiting time: (6 + 0 + 3)/3 = 3
  - Avg. response time: (3 + 6 + 30)/3 = 13
- In second case:
  - Average waiting time is much better (before it was 17)
  - Average response time is better (before it was 27)

- FCFS Pros and Cons:
  - Simple (+) Convoy effect (-); perf. depends on arrival order

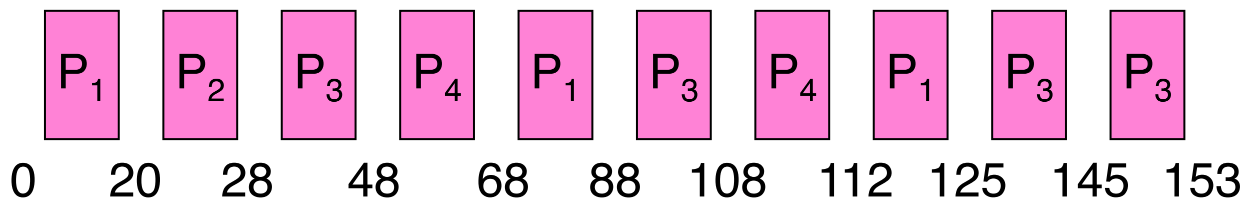
Round Robin (RR)

- Each process gets a quanta of CPU time 10ms
- When quantum expires,process is preempted

- If the current CPU burst finishes before quantum expires, the process blocks for IO
- n processes ; quantum is $q \Rightarrow$
  - Each process gets (roughly) $1/n$ of CPU time
  - In chunks of at most q time units
  - No process waits more than (n-1)q time units

RR with Time Quantum 20

P1: 53 P2: 8 P3: 68 P4: 24



- Waiting time for P_1= (68-20)+(112-88)=72 ; P_2 = 20
  P_3= (28-0)+(88-48)+(125-108)=85 ; P_4=(48-0)+(108-68)=88
- Avg. waiting time = (72+20+85+88)/4=66¼
- Avg. response time = (125+28+153+112)/4 = 104½

➡ RR Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switch time adds up for long jobs (-)

Choice of Time Slice

- How to choose time slice?
  - Too big?
    - Performance of short jobs suffers
  - Infinite (∞)?
    - Same as FCFS
  - Too small?
    - Performance of long jobs suffers due to excessive context-switch overhead
- Actual choices of time slice:
  - Early UNIX time slice is one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three users running? 3 seconds to echo each keystroke!
  - In practice:
    - Typical time slice today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms

FCFS vs. RR

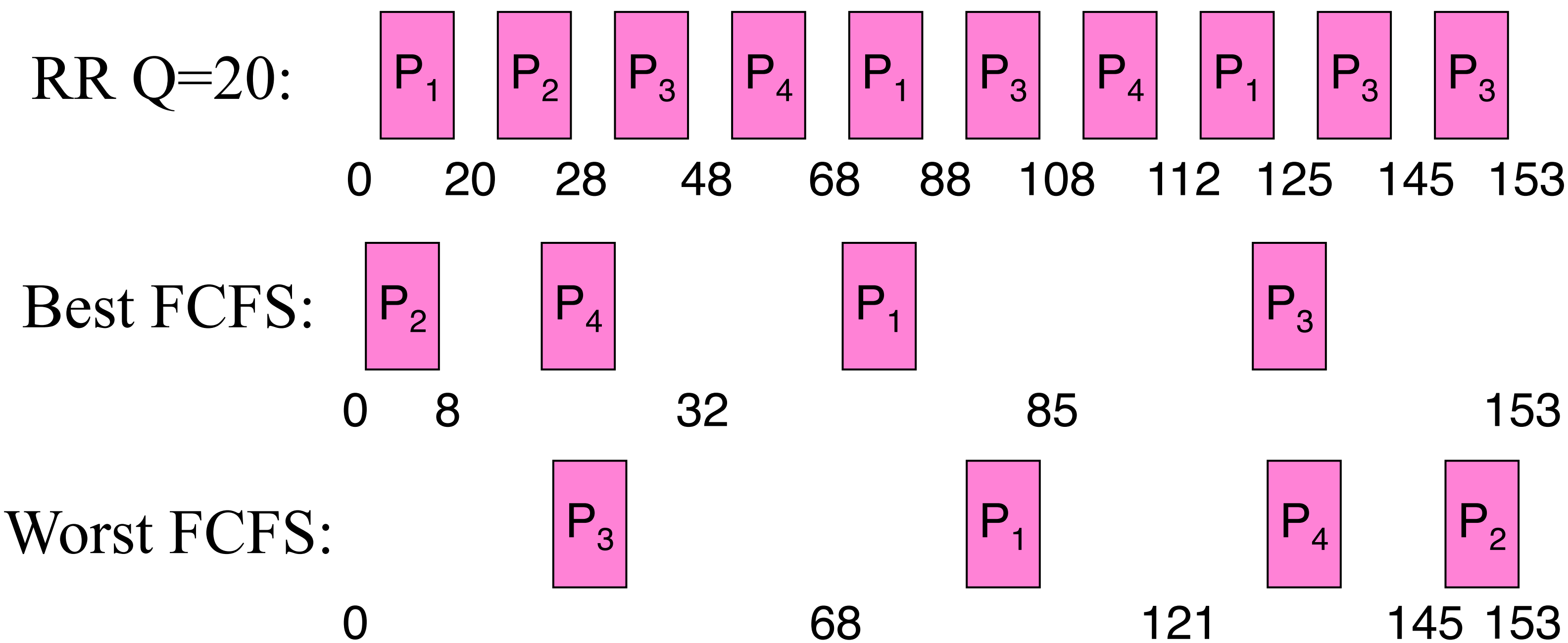
- Assuming zero-cost context-switching time, is RR always better than FCFS? No.
- Example: 10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
- Response times:

Job #	FCFS	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time
  - Average response time is much worse under RR!
    - Bad when all jobs same length

Uneven Jobs

P1: 53   P2: 8   P3: 68   P4: 24



- When jobs have uneven length, it seems to be a good idea to run short jobs first!

Eg. with Different Quanta

	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Response Time	Best FCFS	85	8	153	32	69½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Shortest-Job First (SJF) Scheduling

- This algorithm associates with each process the length of its next CPU burst
  - shortest next CPU burst is chosen
  - Big effect on short jobs, small effect on long;
  - Better avg. response time

- Problem: is length of a job known at its arrival time?
  - Generally no; possible to predict

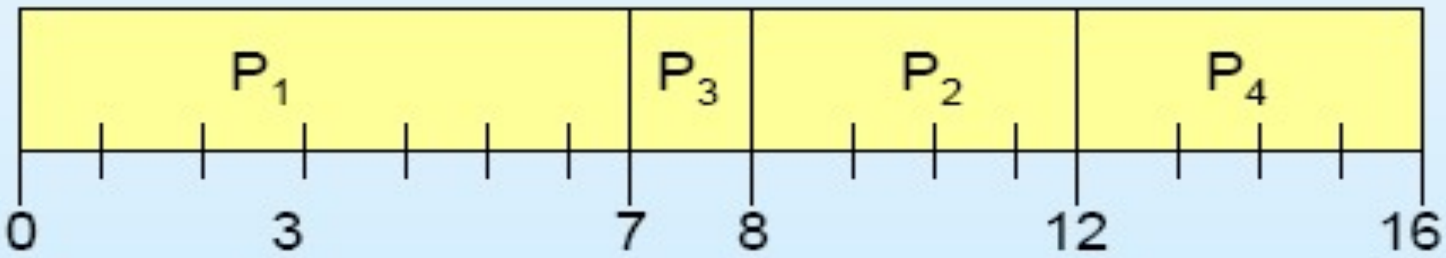
Two Versions

- Non-preemptive – once a job starts executing, it runs to completion
- Preemptive – if a new job arrives with remaining time less than remaining time of currently-executing job, preempt the current job.
  - Also called Shortest-Remaining-Time-First (SRTF)

Short job first scheduling-Non-preemptive

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (non-preemptive)

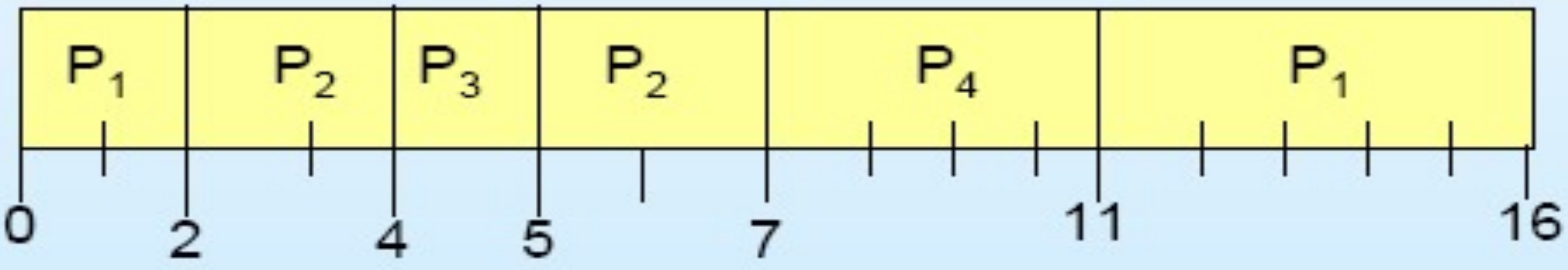


Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Short job first scheduling-Preemptive

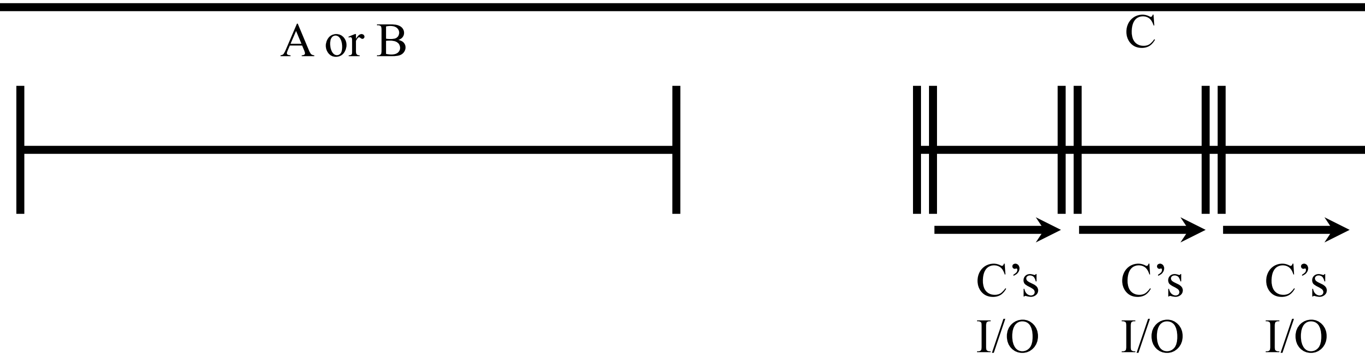
Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (preemptive)



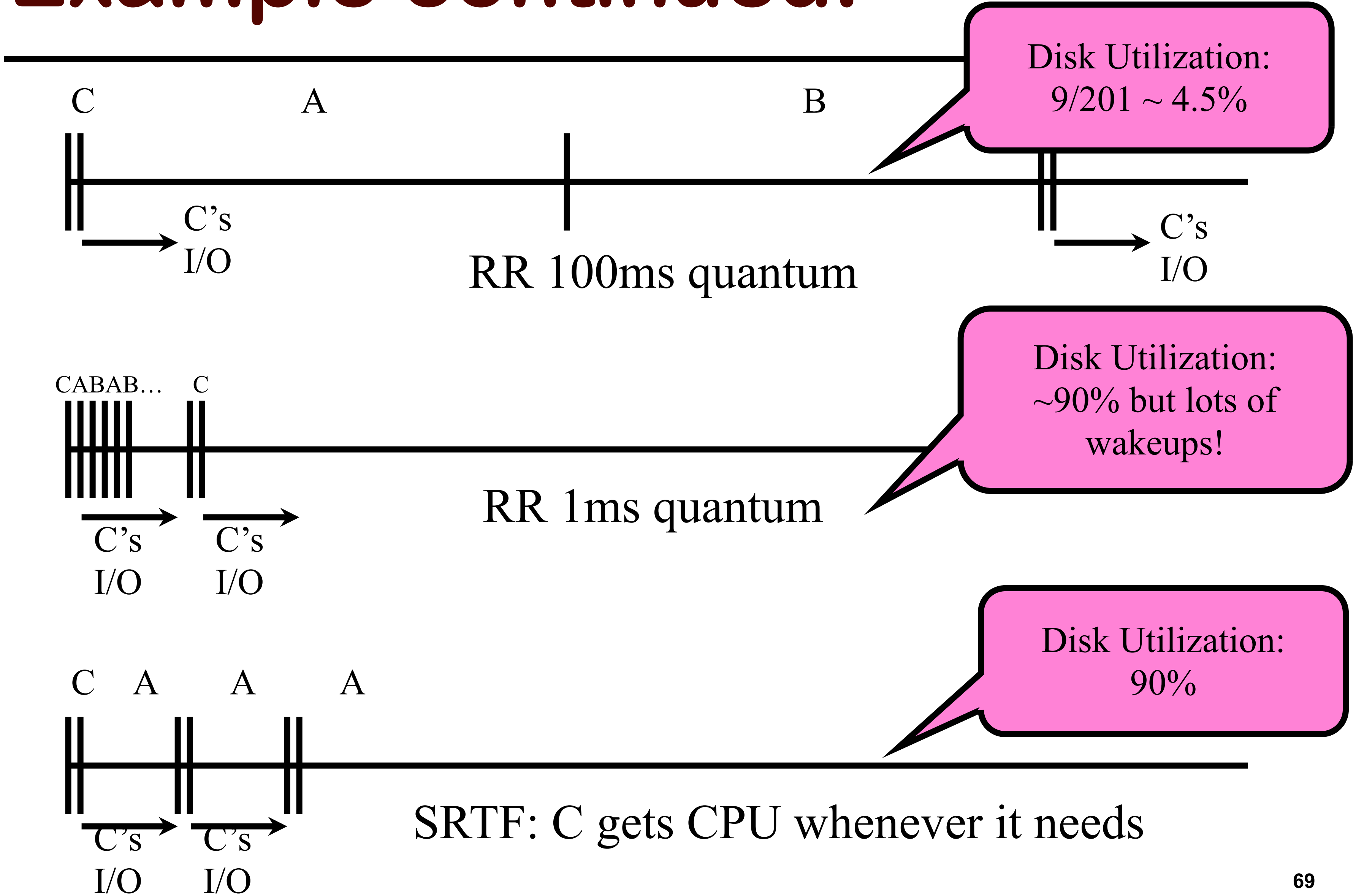
Average waiting time = (9 + 1 + 0 +2)/4 = 3

Example to Illustrate Benefits of SRTF



- Three processes:
  - A,B: both CPU bound, each runs for a week
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B use 100% of the CPU
- With FCFS:
  - Once A or B get in, keep CPU for two weeks

Example continued:



Discussions

- SJF/SRTF are provably-optimal algorithms
  - SRTF is always at least as good as SJF
- Comparison of SRTF with FCFS and RR
  - What if all jobs have the same length?
    - SRTF becomes the same as FCFS
  - What if CPU bursts have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones

SRTF Discussions Cont'

- Starvation
  - Long jobs never get to run if many short jobs
- Need to predict the future
  - Some systems ask the user to provide the info

- In reality, can't really know how long job will take
  - However, can use SRTF as a yardstick
    Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)

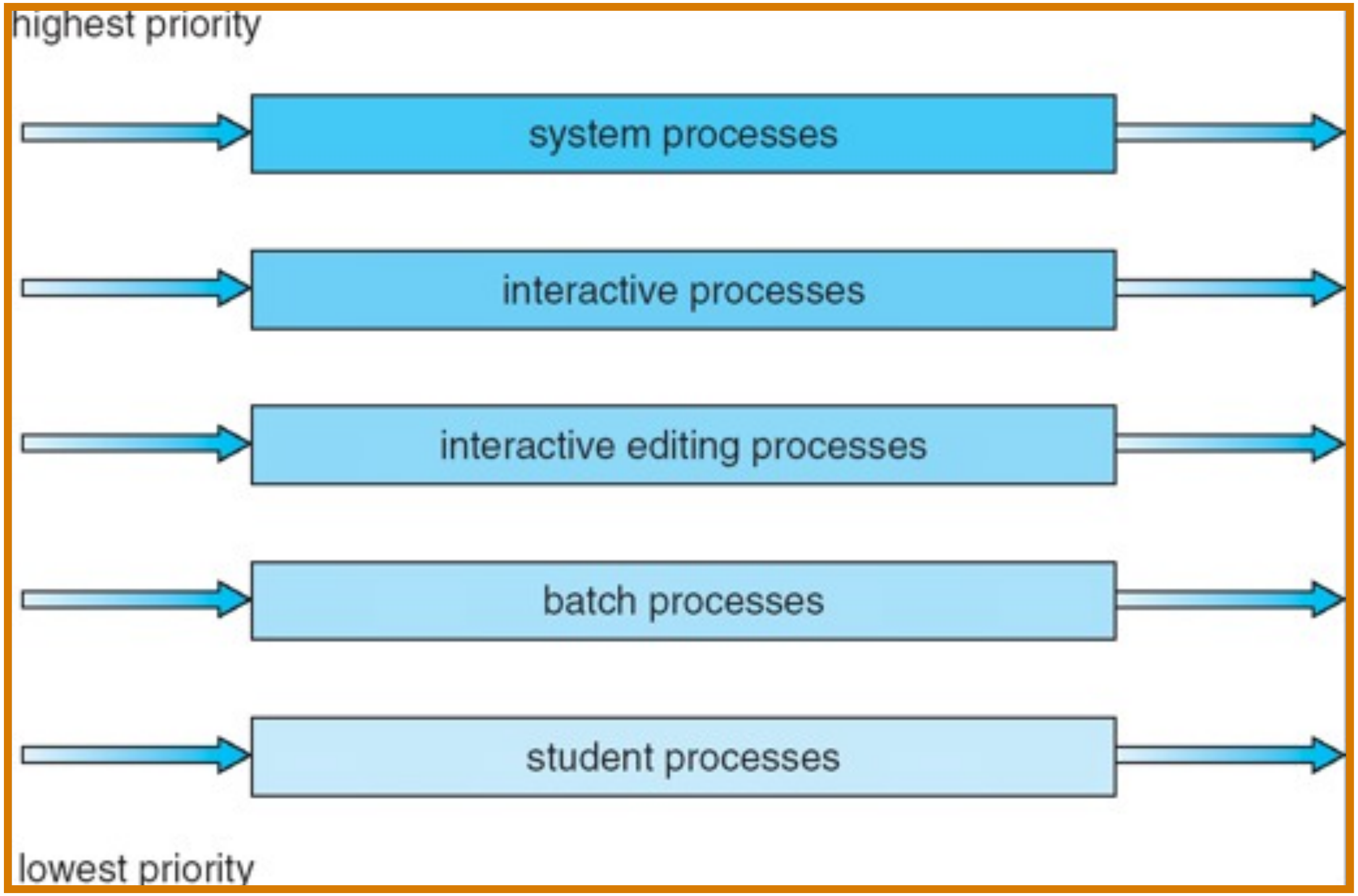
Priority-Based Scheduling

- A priority number (integer) is associated with each process;
  - (Convention: smallest integer ≡ highest priority)
- Can be preemptive or non-preemptive
- SJF/SRTF are special cases of priority-based scheduling
- Starvation – low priority processes may never execute
  - Sometimes this is the desired behavior!

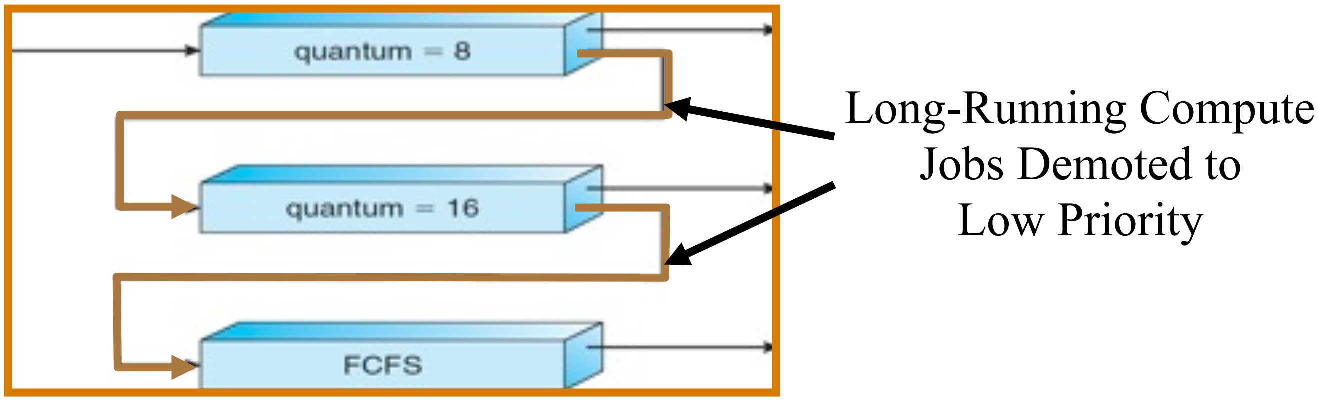
Multi-Level Queue Scheduling

- Ready queue is partitioned into multiple queues
- Each queue has its own scheduling algorithm
  - e.g., foreground queue (interactive processes) with RR scheduling, and background queue (batch processes) with FCFS scheduling
- Scheduling between the queues
  - Fixed priority, e.g., serve all from foreground queue, then from background queue

Multilevel Queue Scheduling



Multi-Level Feedback Queue Scheduling



- Dynamically adjust each process' priority
  - It starts in highest-priority queue
  - If quantum expires, drop one level
  - If it blocks for IO before quantum expires, push up one level

Scheduling Details

- Result approximates SRTF:
  - CPU-bound processes are punished
  - Short-running I/O-bound processes are rewarded
  - No need for prediction of job runtime; rely on past
- User action can foil intent of the OS designer
  - e.g., put in a bunch of meaningless I/O like printf()
  - If everyone did this, this trick wouldn't work!

Lottery Scheduling

- Unlike previous algorithms that are deterministic, this is a probabilistic
  - Give each process some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each process

- How to assign tickets?
  - To approximate SRTF, short running processes get more, long running jobs get fewer
  - To avoid starvation, every process gets at least a min number of tickets

Lottery Scheduling Example

- Assume each short process get 10 tickets; each long process get 1 ticket

# short procs/	% of CPU each	% of CPU each
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

Summary

- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- FCFS Scheduling:
  - Pros: Simple
  - Cons: Short jobs can get stuck behind long ones
- Round-Robin Scheduling:
  - Pros: Better for short jobs
  - Cons: Poor performance when jobs have same length

Summary Cont'

- Shortest Job First (SJF) and Shortest Remaining Time First (SRTF)
  - Run the job with least amount of computation
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- Priority-Based Scheduling
  - Each process is assigned a fixed priority
- Multi-Level Queue Scheduling
  - Multiple queues of different priorities
- Multi-Level Feedback Queue Scheduling:
  - Automatic promotion/demotion of process between queues
- Lottery Scheduling:
  - Give each process a number of tickets (short tasks $\Rightarrow$ more tickets)
  - Reserve a minimum number of tickets for every process to ensure forward progress