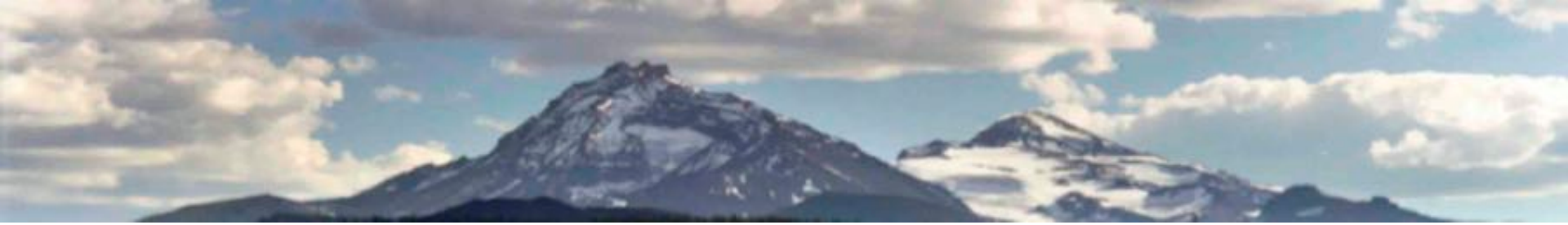


CIS 415
Operating Systems
Interprocess Communication

Prof. Allen D. Malony
Department of Computer and Information Science
Spring 2020

UNIVERSITY OF OREGON

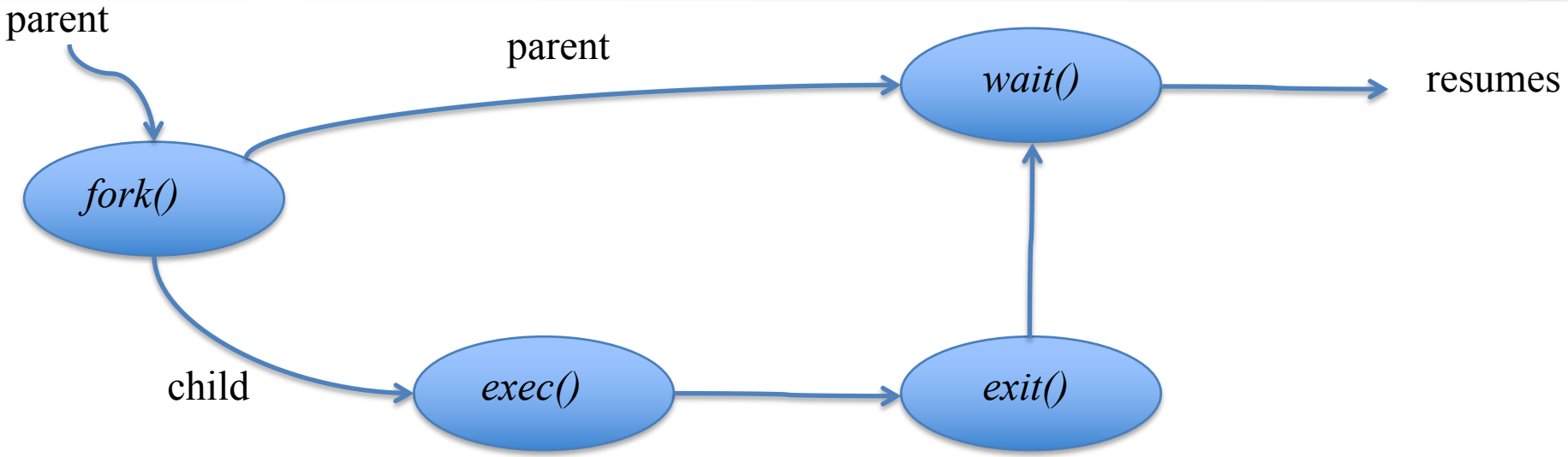
Logistics

- Get started on Project 1
  - Do not put it off
- Lab 2 tomorrow
- Read OSC Chapter 3
- Look at all lectures slides, even if we do not have time to cover in class

Outline

- Quick review of *fork()* and *exec()*
- Brief introduction to process scheduling
- Interprocess communication
- Remote procedure calls

Process Creation with New a Program



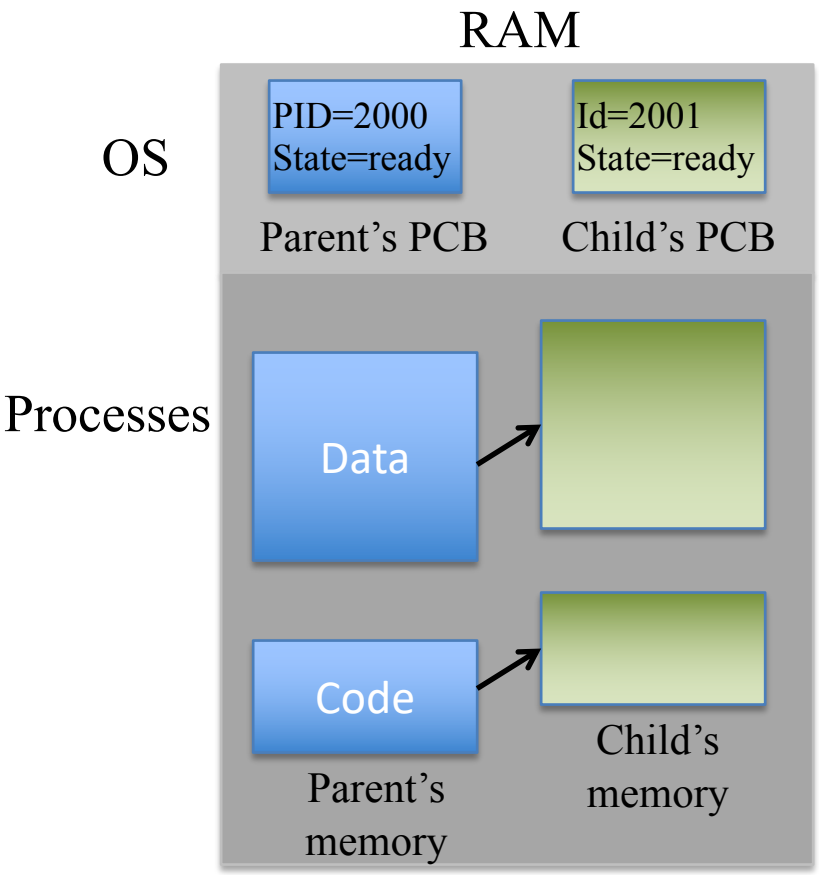
- Parent process calls *fork()* to spawn child process
  - Both parent and child return from fork()
  - Continue to execute the same program
- Child process calls *exec()* to load a new program

C Program Forking Separate Process

```
int main( )
{
pid_t  pid;
  /* fork another process */
  pid = fork( );
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }
  else if (pid == 0) { /* child process */
    execlp("/bin/ls", "ls", NULL); /* exec a file */
  }
  else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf ("Child Complete");
    exit(0);
  }
}
```

`execl`, `execlp`, `execle`, `execv`, `execvp`, `execvpe` all execute a file and are frontends to `execve`

Process Layout



Effects in memory after parent calls *fork()*

1. PCB with new PID created
2. Memory allocated for child initialized by copying over from the parent
3. If parent had called *wait()*, it is moved to a waiting queue
4. If child had called *exec()*, its memory is overwritten with new code and data
5. Child added to ready queue and is all set to go now!

Analogy

- Doing a *fork()* is analogous to cloning
  - Copy is exactly the same
  - Completely able to live independently
  - Each has its own ability to execute
  - Each has its own resources
    - gets a copy of the parent's process address space (unless you use another variant of *fork()*)
    - gets a copy of the parent's PCB (includes its memory management information)
  - Still executes the same code

- What if you want to change the clone's behavior
- Doing an *exec()* is analogous to replacing the brain
  - Executing new program code

Process Communication

- Process cooperation is a fundamental aspect of an OS and of the computing environment it is maintaining on behalf of executing applications
- Processes need to interact and share information
- Process model is a useful way to isolate running programs (separate resources, state, and so on)
  - It can simply programs (no need to worry about other processes)
  - But processes do not always work in isolation, nor do we want them to
  - Sometimes it is easier to design programs if there are multiple processes working together
- How to support process interoperation, communication, and sharing of information
- Discuss a variety of ways
  - Not talk about sharing of files or signals

Process Communication (Interoperation)

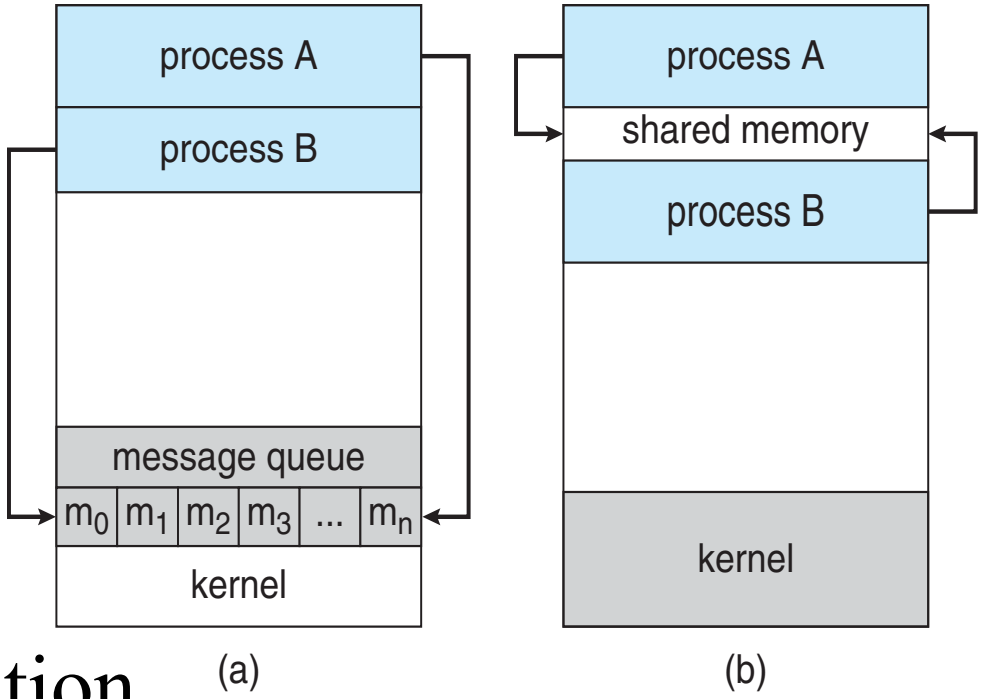
- When is communication necessary?
- Lots of examples in operating systems
  - Kernel/OS access to user process data
  - Processes sharing data via shared memory
  - Processes sharing data via system calls
  - Processes sharing data via file system
  - Threads with access to same data structures
- In general, there are numerous examples in computer science where interoperation is important
  - DB transactions, distributed computing, parallelism

Interprocess Communication (IPC)

- Mechanism for processes to communication and synchronize
- Logically, we want some sort of *messaging system*
- Logically, an IPC facility would provide 2 operations for processes to communicate:
  - Send(message) (message size fixed or variable)
  - Receive(message)
- However, first there needs to be a communication channel
  - If processes P and Q wish to communicate they open a channel
  - Then exchanging messages can proceed
- How is the communication channel (link) realized?
  - Physically, there are different alternatives
  - Logically, need abstract interfaces, protocols, and properties

IPC Mechanisms

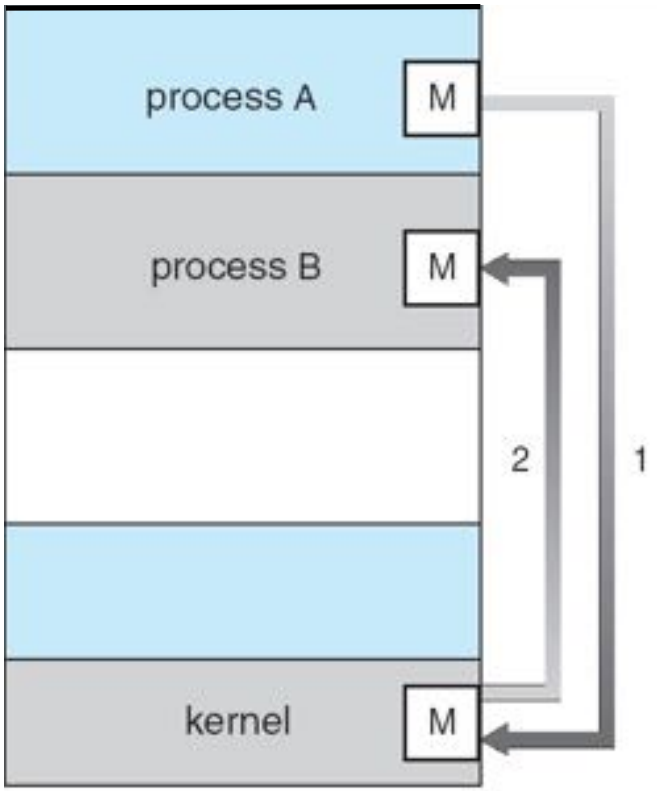
- Interprocess communication (IPC) supports the exchange of data between processes
- Two fundamental methods (OS supported):
  - *Shared memory*
    - pipes, shared buffer
  - *Message Passing*
    - mailboxes, sockets
- Which one would you use and why?
- Depends on the application



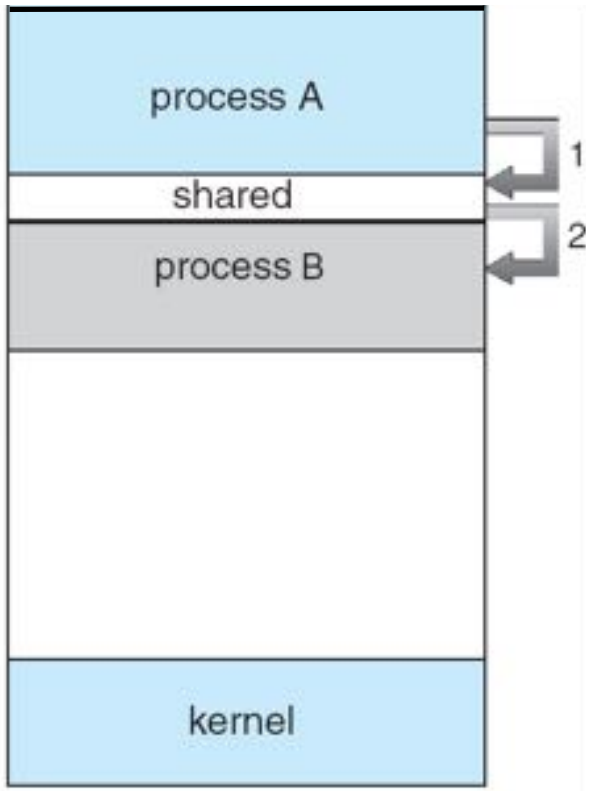
Implementation Questions

- How are links established?
- Can a link be associated with >2 processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of the link?
- How is the data flow regulated on the link?
- Is the message size fixed or variable?
- Is the link unidirectional or bi-directional?
- Does an actual network link have to be used or can the memory system be used somehow?

Communication Models



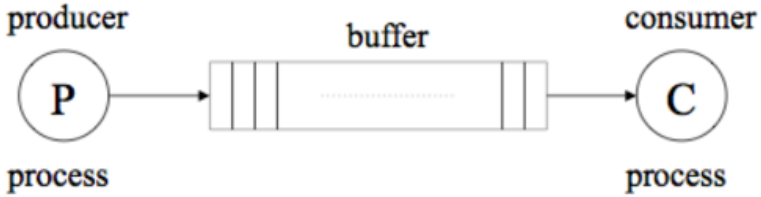
Message Passing



Shared Memory

Think Abstractly about the Problem

- Consider two processes sharing a memory region
  - *Producer* writes
  - *Consumer* reads
- Producer action
  - While the buffer not full …
  - … stuff can be written (added) to the buffer
- Consumer actions
  - When stuff is in the buffer …
  - … it can be read (removed)
- Must manage where new stuff is in the buffer
- Can think of the buffer as being
  - Bounded (Problems?)
  - Unbounded (Realistic?)



Shared Memory -- Producer

```
item nextProduced;

while (1) {
   while (((in + 1) % BUFFER_SIZE) == out)
      ; /* do nothing … buffer is full */
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Circular buffer

Assume that the consumer is modifying the `out` variable. Do you see any problems here?

Shared Memory -- Consumer

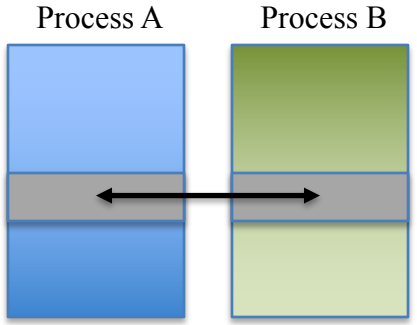
```
item nextConsumed;

while (1) {
  while (in == out)
    ; /* do nothing … buffer is empty */
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
}
```

Circular buffer

Assume that the producer is modifying the `in` variable. Do you see any problems here?

IPC with Shared Memory

- Communicate by reading/writing from a specific memory location (*in logical memory*)
  - Setup a shared memory region (*segment*) in your process
  - Permit others to attach to the shared memory region
- *shmget(2)* -- create shared memory segment
  - Permissions (read and write)
  - Size
  - Returns an identifier for the segment
- *shmat(2)* -- attach to existing shared memory segment
  - Specify identifier
  - Location in local address space
  - Permissions (read and write)
- Also, operations for detach and control



IPC with Pipes

- Producer-Consumer mechanism for data exchange
  - *prog1* | *prog2* (shell notation for pipe)
  - Output of prog1 becomes the input to prog2
  - More precisely, a connection is made so that the *standard output* of prog1 is connected to *standard input* of prog2
- OS sets up a fixed-size buffer (*OS manages it*)
  - System calls: *pipe(2)*, *dup(2)*, *popen(2)*
- Producer
  - Write to buffer, if space available
- Consumer
  - Read from buffer if data available

Management of Pipes

- Buffer management
  - A finite region of memory (array or linked-list)
  - Wait to produce if no room
  - Wait to consume if empty
  - Produce and consume complete items
- Access to buffer
  - Write adds to buffer (updates end of buffer)
  - Reader removes stuff from buffer (updates start of buffer)
  - Both are updating buffer state
- Issues
  - What happens when end is reached (e.g., in finite array)?
  - What happens if reading and writing are concurrent?
  - Who is managing the pipe?

IPC with Message Passing

- Mechanisms for processes to communicate and to synchronize actions
- Messaging system
  - Processes communicate without shared variables
  - Use messages instead
- Establish communication link
  - Producer sends on link
  - Consumer receives on link
- IPC Operations
  - *Send(P, message)*: send a message to process *P*
  - *Receive(Q, message)*: receive a message from process *Q*
- Issues
  - What if a process wants to receive from any proces?
  - What if communicating processes are not ready at same time?
  - What size message can a process receive?
  - Can other processes receive the same message from one process?

Synchronous Messaging

- Direct communication from one process to another
- Synchronous send
  - *Send(P, message)*
  - Producer must wait for the consumer to be ready to receive the message
- Synchronous receive
  - *Receive(ID, message)*
  - ID could be any process
  - Wait for someone to deliver a message
  - Allocate enough space to receive message
- Synchronous means that both have to be ready!
  - Otherwise process (sender or receiver) is blocked

Properties of Direct Communication Links

- Links are established automatically
- A link is associated with exactly one pair of communicating processes
- Between each pair there exists exactly one link
- The link may be unidirectional, but is usually bi-directional
- Messaging
  - Processes must name each other explicity

Asynchronous Messaging

- Indirect communication from one process to another
- Asynchronous send
  - *Send(M, message)*
  - Producer sends message to a buffer M (like a mailbox)
  - No waiting (modulo busy mailbox)
- Asynchronous receive
  - *Receive(M, message)*
  - Receive a message from a specific buffer (get your mail)
  - No waiting (modulo busy mailbox)
  - Allocate enough space to receive message
- Asynchronous means that you can send/receive when you're ready
  - What are some issues with the buffer?

Properties of Indirect Communication Link

- Link established only if processes share a mailbox
- A link may be associated with many processes
- Each pair of processes may share several communication links
- Link may be unidirectional or bi-directional
- Messages
  - Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique ID
  - Processes can communicate only if they share a mailbox

Synchronization

- Message passing may be either *blocking* or *non-blocking*
- Blocking is considered synchronous
  - Blocking send
    - sender is blocked until the message is received
  - Blocking receive
    - receiver is blocked until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send
    - sender sends the message and continue
  - Non-blocking receive
    - receiver receives: a valid message or a null message
- Different combinations possible
  - If both send and receive are blocking, we have a *rendezvous*

Client-Server Communication

- Sockets
- Remote procedure calls
- Remote method invocation (a la Java)

IPC with Sockets

- Defined as an end point for communication
  - Connect one socket to another (*TCP/IP*)
  - Send/receive message to/from another socket (*UDP/IP*)
- Sockets are named by
  - IP address (roughly, machine)
  - Port number (service: ssh, http, …)
  - Concatenate the two (**161.25.19.8:1625**)
- Communication consists of a pair of sockets
- Semantics
  - Bidirectional link between a pair of sockets
  - Messages: unstructured stream of bytes
- Connection between
  - Processes on same machine (UNIX domain sockets)
  - Processes on different machines (TCP or UDP sockets)
  - User process and kernel (netlink sockets)

Files and File Descriptors

- POSIX system calls for interacting with files
  - *open()*, *read()*, *write()*, *close()*
  - *open( )* returns a *file descriptor*
    - an integer that represents an open file
    - inside the OS, it's an index into a table that keeps track of any state associated with your interactions, such as the file position
    - you pass the file descriptor into *read*, *write*, and *close*
  - File descriptors are kept as part of the process information in the process control block

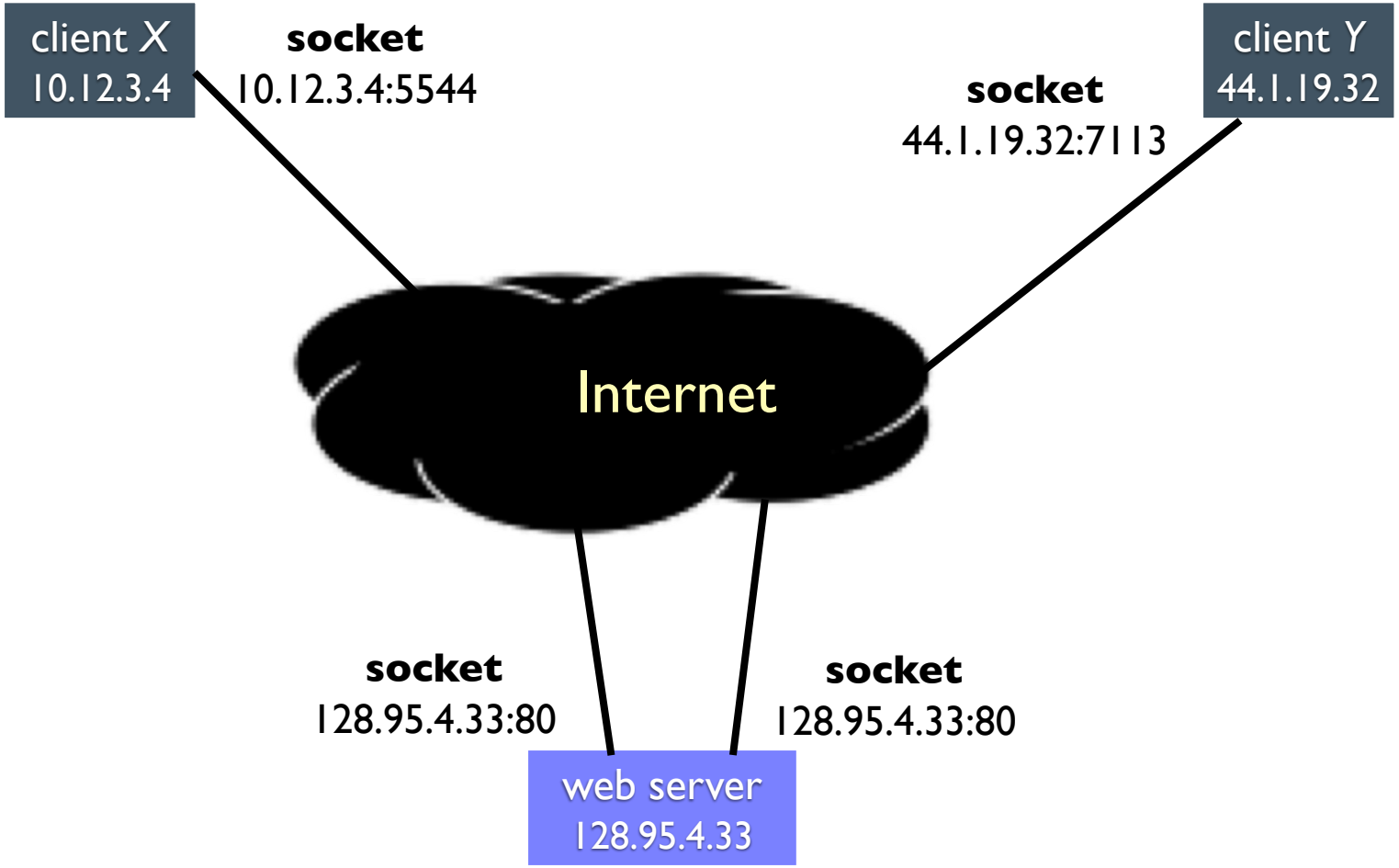
Networks and Sockets

- UNIX likes to make all I/O look like file I/O
  - The good news is that you can use *read( )* and *write( )* to interact with remote computers over a network!
- File descriptors are used for network communications
  - The socket is the file descriptor
- Just like with files....
  - Your program can have multiple network channels (sockets) open at once
  - You need to pass *read( )* and *write( )* the socket file descriptor to let the OS know which network channel you want to use

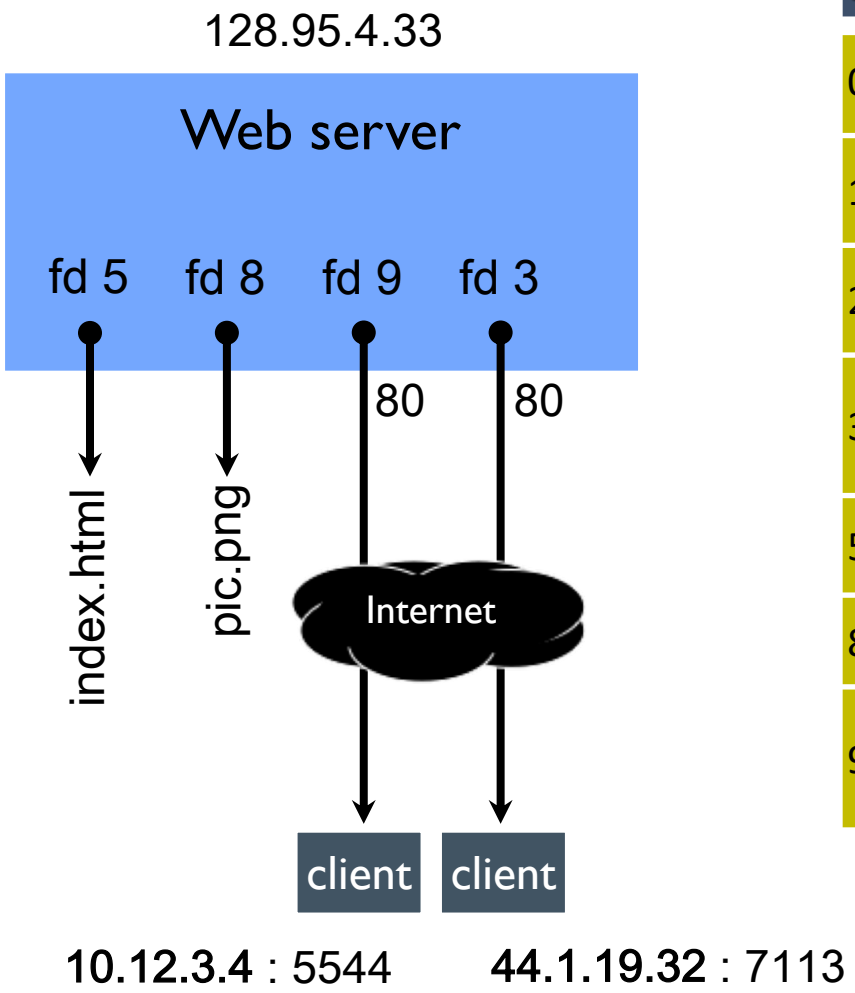
Examples of Sockets

- HTTP / SSL
- email (POP/IMAP)
- ssh
- telnet

IPC and Sockets



File Descriptors



file descriptor	type	connected to?
0	pipe	stdin (console)
1	pipe	stdout (console)
2	pipe	stderr (console)
3	TCP socket	local: 128.95.4.33:80 remote: 44.1.19.32:7113
5	file	index.html
8	file	pic.png
9	TCP socket	local: 128.95.4.33:80 remote: 102.12.3.4:5544

OS's file descriptor table

Types of Sockets

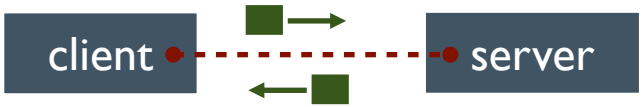
- Stream sockets
  - For connection-oriented, point-to-point, reliable bytestreams
    - uses TCP, SCTP, or other stream transports
- Datagram sockets
  - For connection-less, one-to-many, unreliable packets
    - uses UDP or other packet transports
- Raw sockets
  - For layer-3 communication
    - raw IP packet manipulation

Stream Sockets

- Typically used for client / server communications
  - But also for other architectures, like peer-to-peer

1. establish connection

- Client
  - An application that establishes a connection to a server



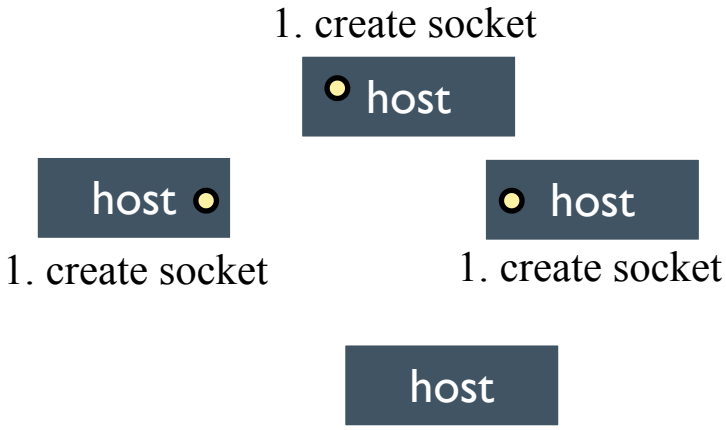
2. communicate

- Server
  - An application that receives connections from clients

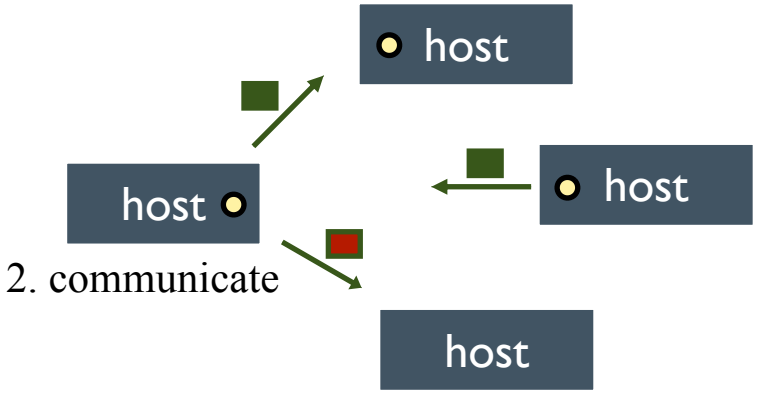
3. close connection

Datagram Sockets

- Used less frequently than stream sockets
  - They provide no flow control, ordering, or reliability



- Often used as a building block
  - Streaming media applications
  - Sometimes, DNS lookups



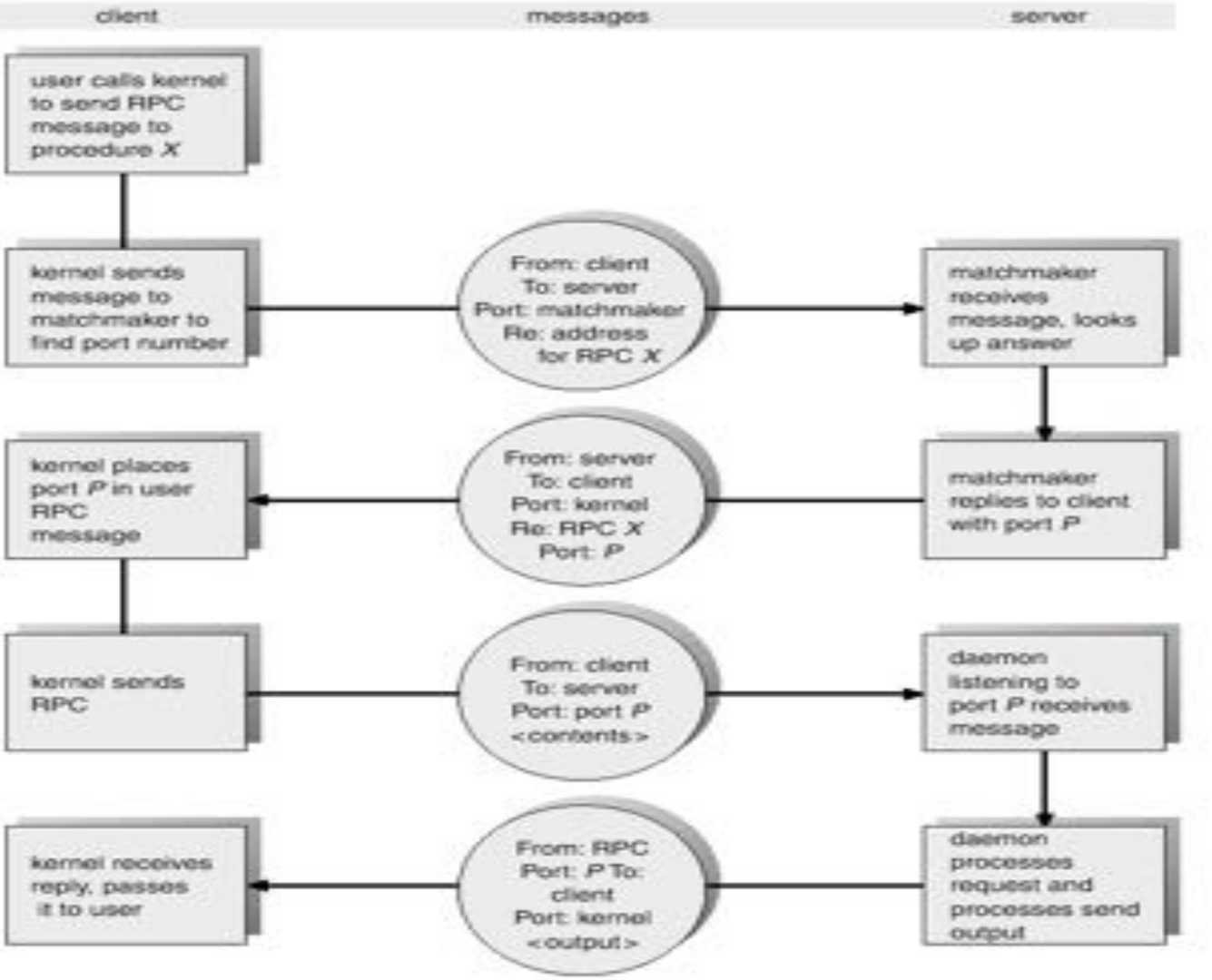
Issues using Sockets

- Communication semantics
  - Reliable or not
- Naming
  - How do we know a machine's IP address? DNS
  - How do we know a service's port number?
- Protection
  - Which ports can a process use?
  - Who should you receive a message from?
    - Services are often open -- listen for any connection
- Performance
  - How many copies are necessary?
  - Data must be converted between various data types

Remote Procedure Calls (RPC)

- Procedure calls between processes on network
  - Looks like a "normal" procedure call
  - However, the called procedure is run by another process
- RPC mechanism
  - Client stub (proxy for actual procedure call on server)
  - Client stub "marshalls" arguments
  - Client stub find destination (server) for RPC
  - Client stub send call and marshalled arguments to server
  - Server stub (performs the actual procedure call)
  - Server stub "unmarshalls" arguments
  - Server stub calls procedure with arguments
  - Server stub "marshall" result and returns to client

Remote Procedure Calls

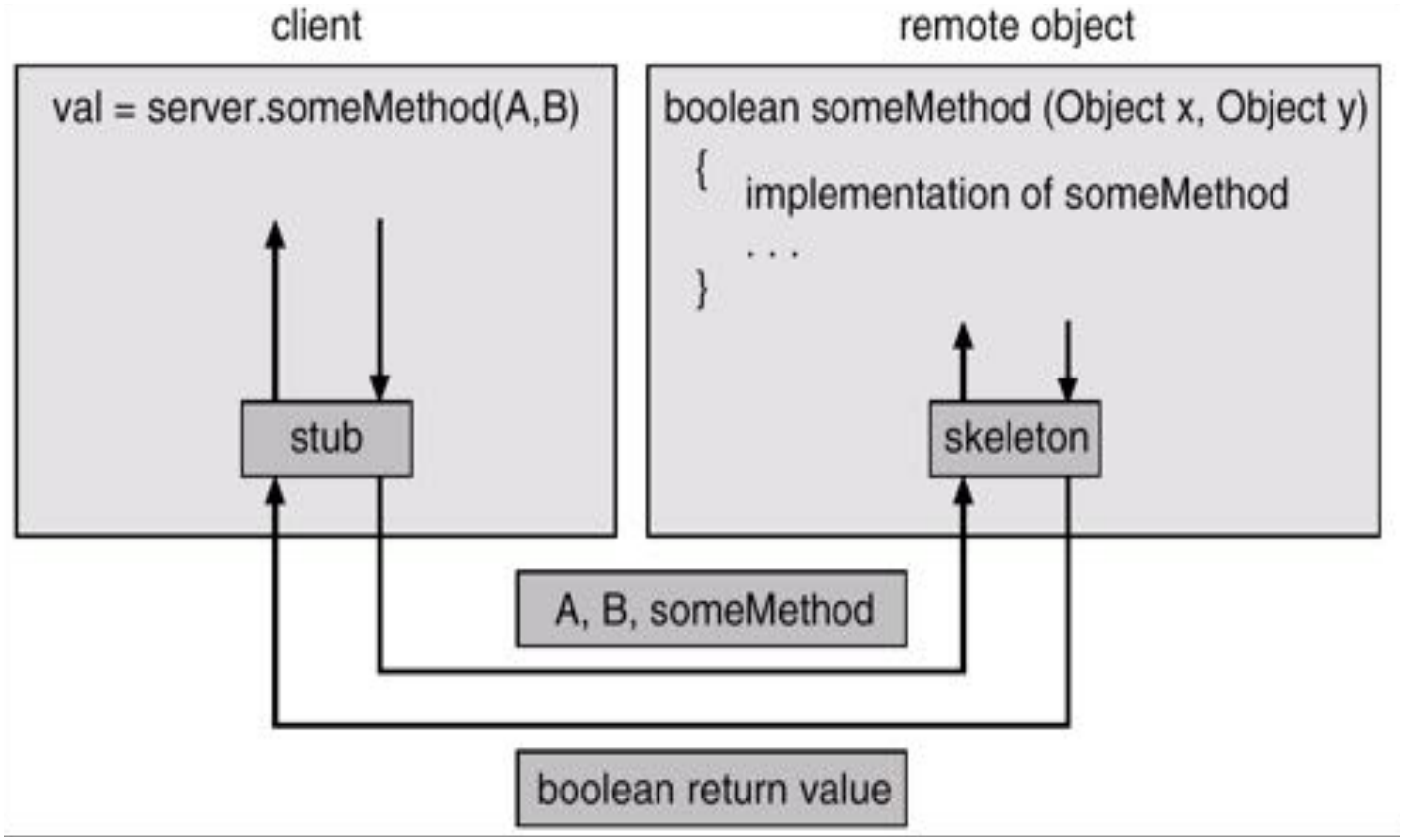


Remote Procedure Calls

- Supported by systems
  - CORBA
  - Java RMI
- Issues
  - Support to build client/server stubs
  - Marshalling arguments and code
  - Layer on existing mechanism (e.g., sockets)
  - Remote server crashes … then what?
- Performance versus abstractions
  - What if the two processes are on the same machine?

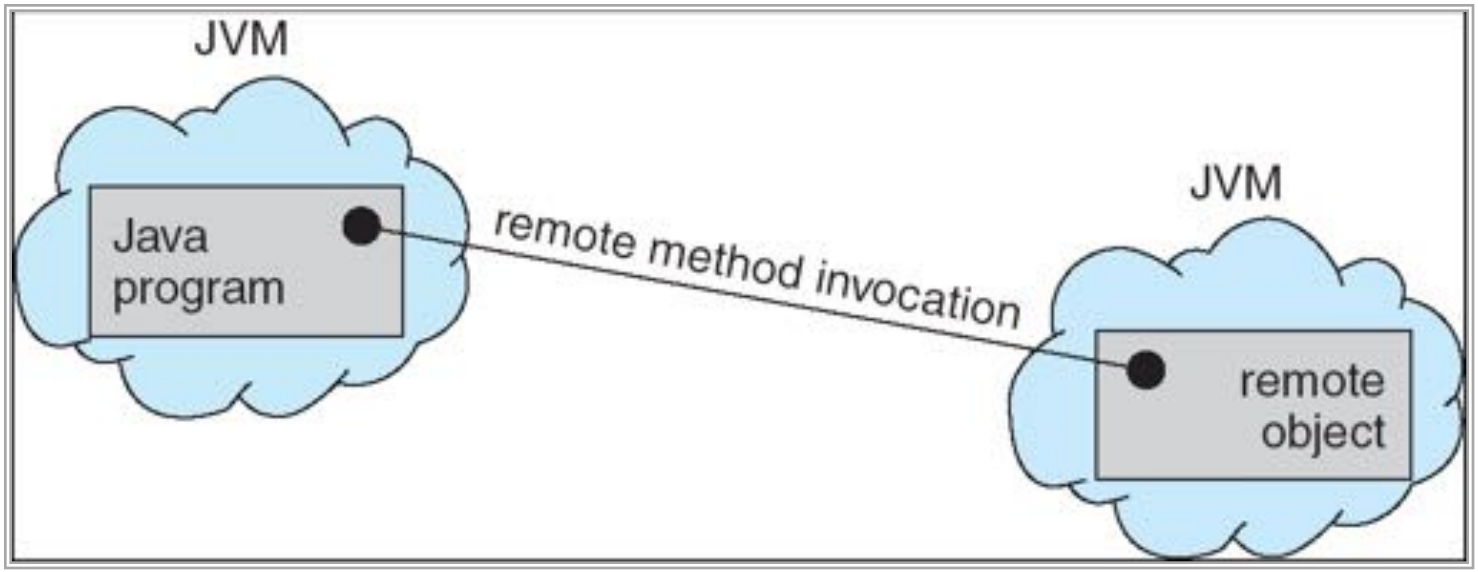
Remote Procedure Calls

- Marshalling



Remote Method Invocation (RMI)

- RMI is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object



IPC Summary

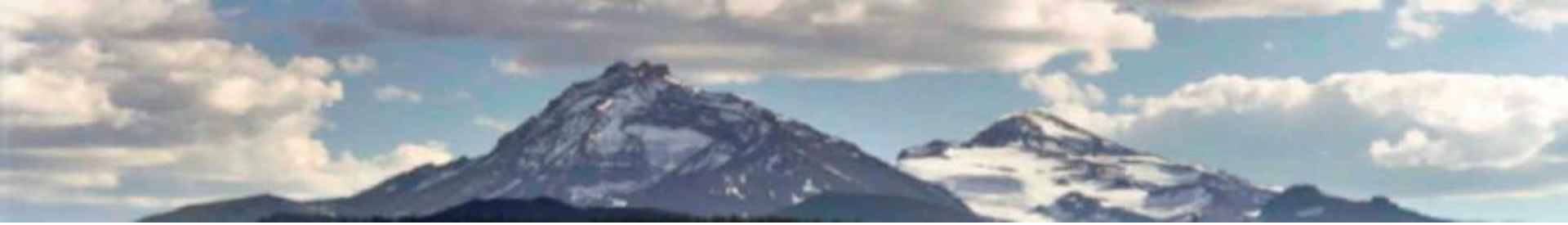
- Lots of mechanisms
  - Pipes
  - Shared memory
  - Sockets
  - RPC
- Trade-offs
  - Ease of use, functionality, flexibility, performance
- Implementation must maximize these
  - Minimize copies (performance)
  - Synchronous vs Asynchronous (ease of use, flexibility)
  - Local vs Remote (functionality)

Summary

- Process
  - Execution state of a program
- Process Creation
  - fork and exec
  - From binary representation
- Process Description
  - Necessary to manage resources and context switch
- Process Scheduling
  - Process states and transitions among them
- Interprocess Communication
  - Ways for processes to interact (other than normal files)

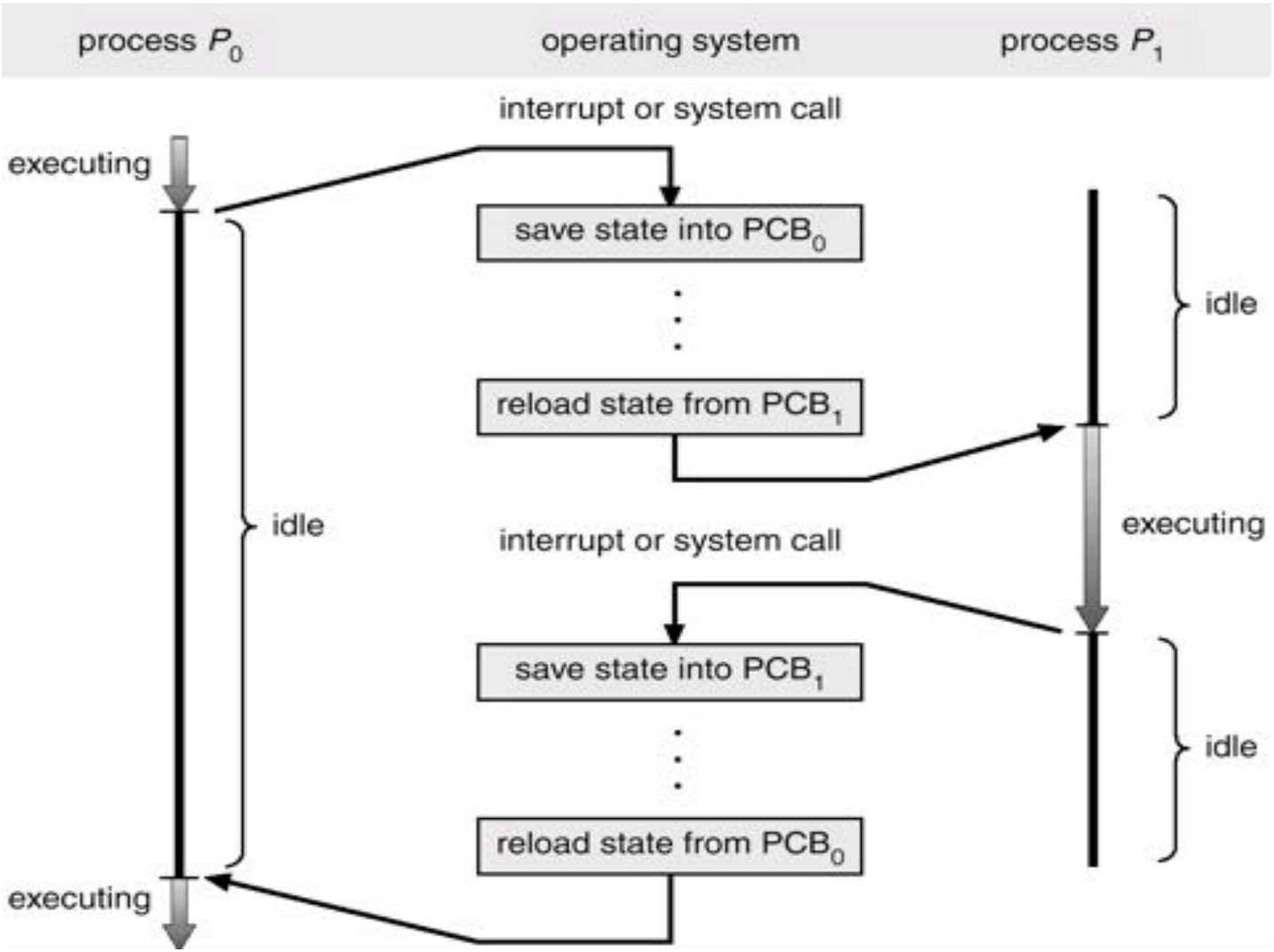
Next Class

- Threads



Additional Slides

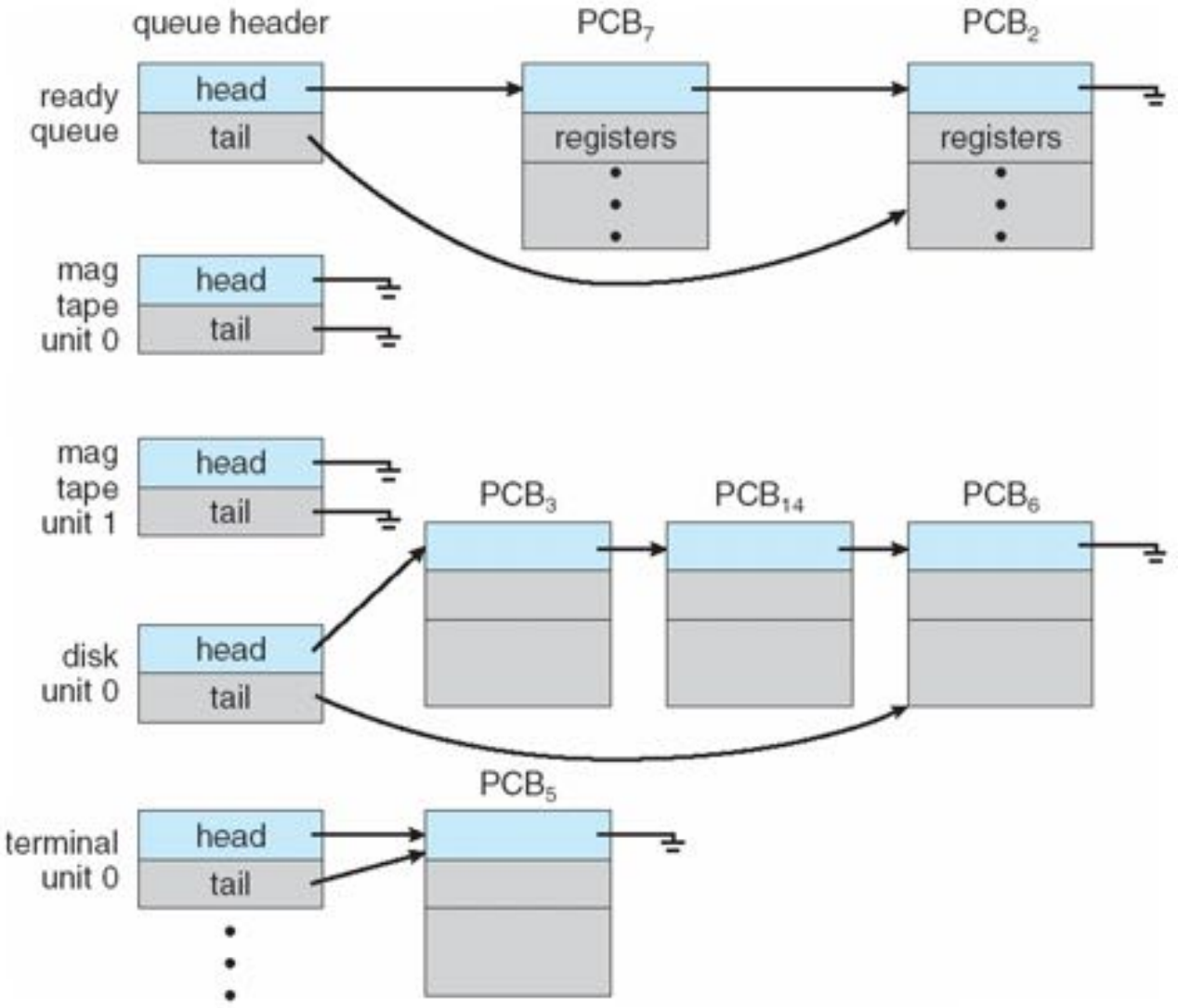
Context Switch



Process Scheduling

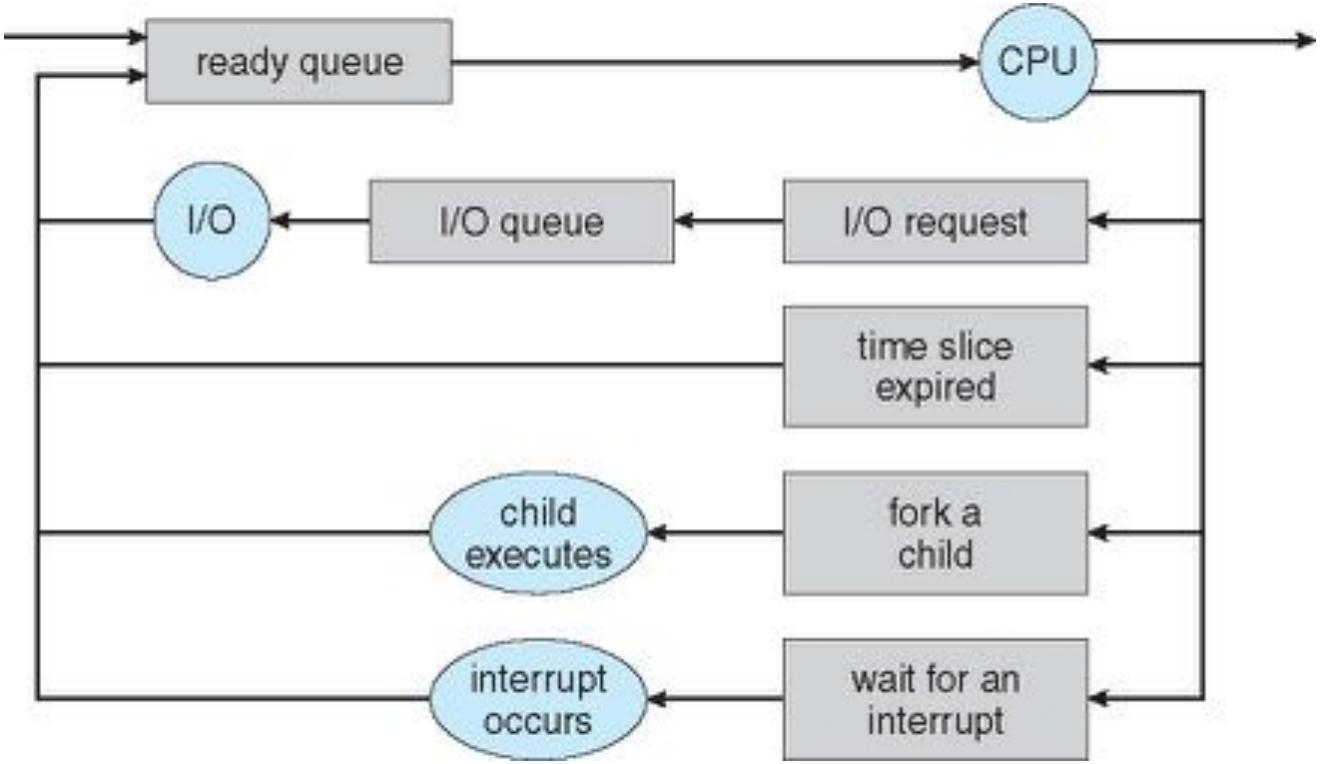
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes
  - *Job queue* – set of all processes in the system
  - *Ready queue* – set of all processes residing in main memory, ready and waiting to execute
  - *Device queues* – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- Process scheduling queueing diagram represents:
  - queues, resources, flows
- Processes move through the queues



Schedulers

- *Short-term scheduler* (or CPU scheduler)
  - Selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds)
    - must be fast because involves context switching
- *Long-term scheduler* (or job scheduler)
  - Selects which programs should be launched
    - program is not yet running as a process
    - once a program is launched, its process isbrought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes)
    - does not have to be fast
  - The long-term scheduler controls the degree of multiprogramming
  - It is possible that a process can be removed temporarily

Process Mix

- Processes can be described as either:
  - I/O-bound process
    - spends more time doing I/O than computations
    - many short CPU bursts
  - CPU-bound process
    - spends more time doing computations
    - few very long CPU bursts
- There can be sub-categories of these
- Long-term scheduler strives for good process mix

Addition of Medium Term Scheduling

- Medium-term scheduler can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution
  - Refer to this as job swapping