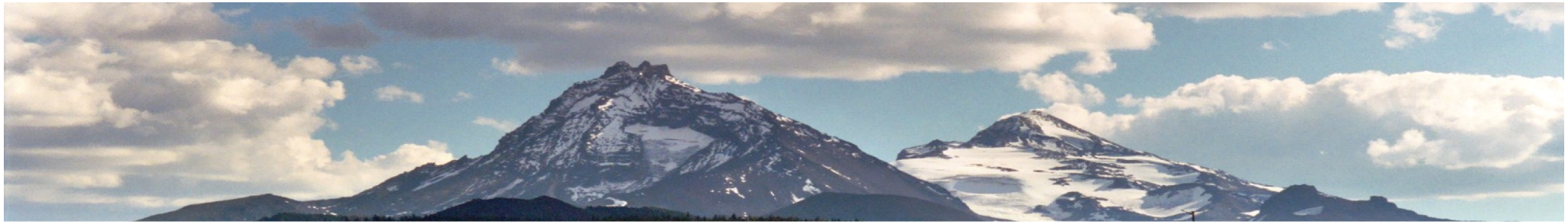


CS 415
Operating Systems
IPC

Prof. Jieyang Chen
Department of Computer and Information Science
Fall 2025

UNIVERSITY OF OREGON

Logistics

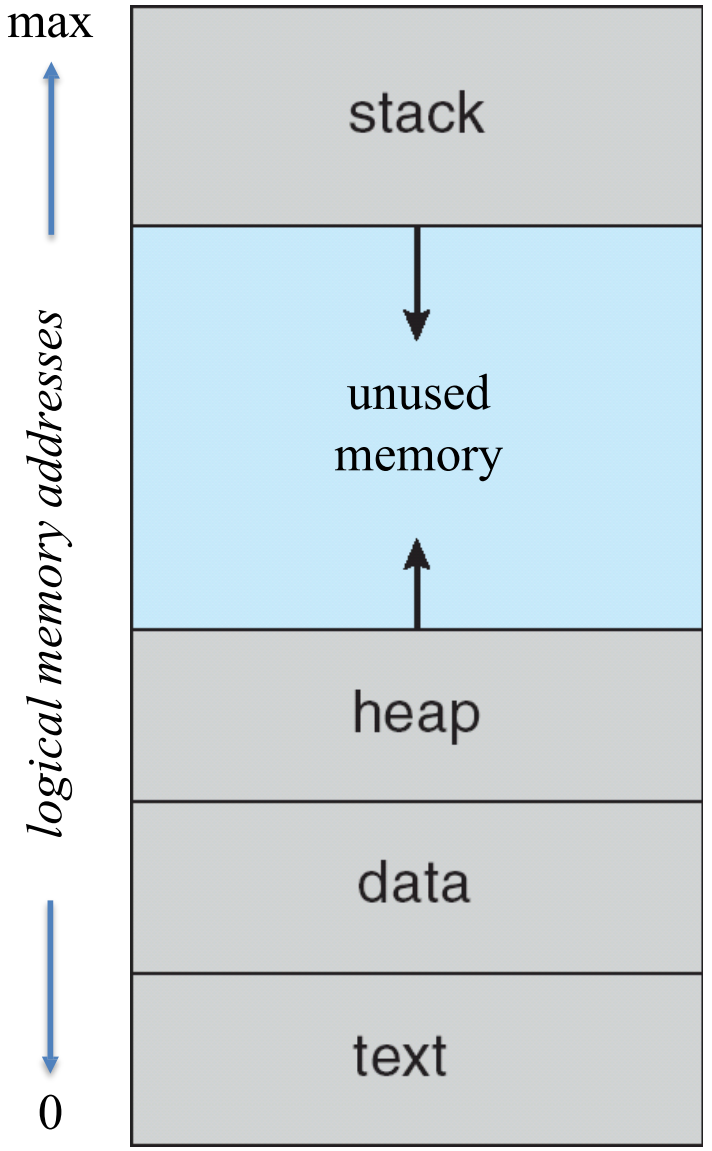
- Keeping working hard on Project 1
- Keep up on your book reading!
- Look at all lecture slides, even if we do not have time to cover in class
  - See Zoom recording posted on lecture page
- Programs posted in Lecture 3 and 4 on Canvas
  - See update in the Lecture 3 example

Outline

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

Process Address Space

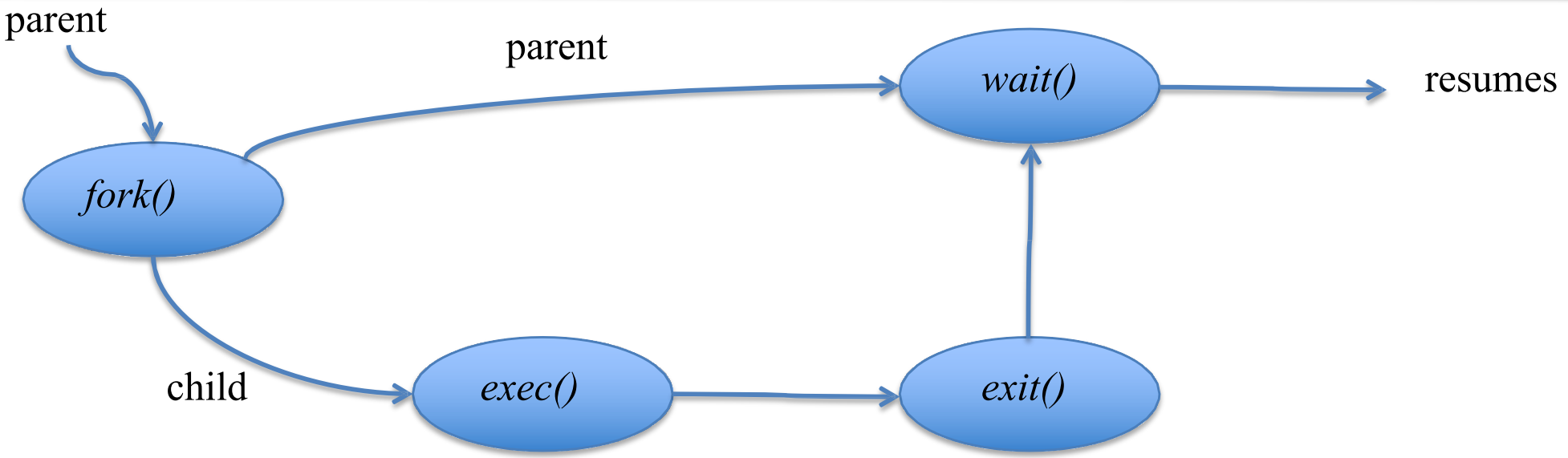
- Any address that could be generated by a process
  - Logical address from instruction execution of the program code
  - All possible logical addresses produced by a process during its execution
- Organized and structured
  - Code must know where things are
- Effectively, it is the executable code that determines the process address space



Logical versus Physical Memory

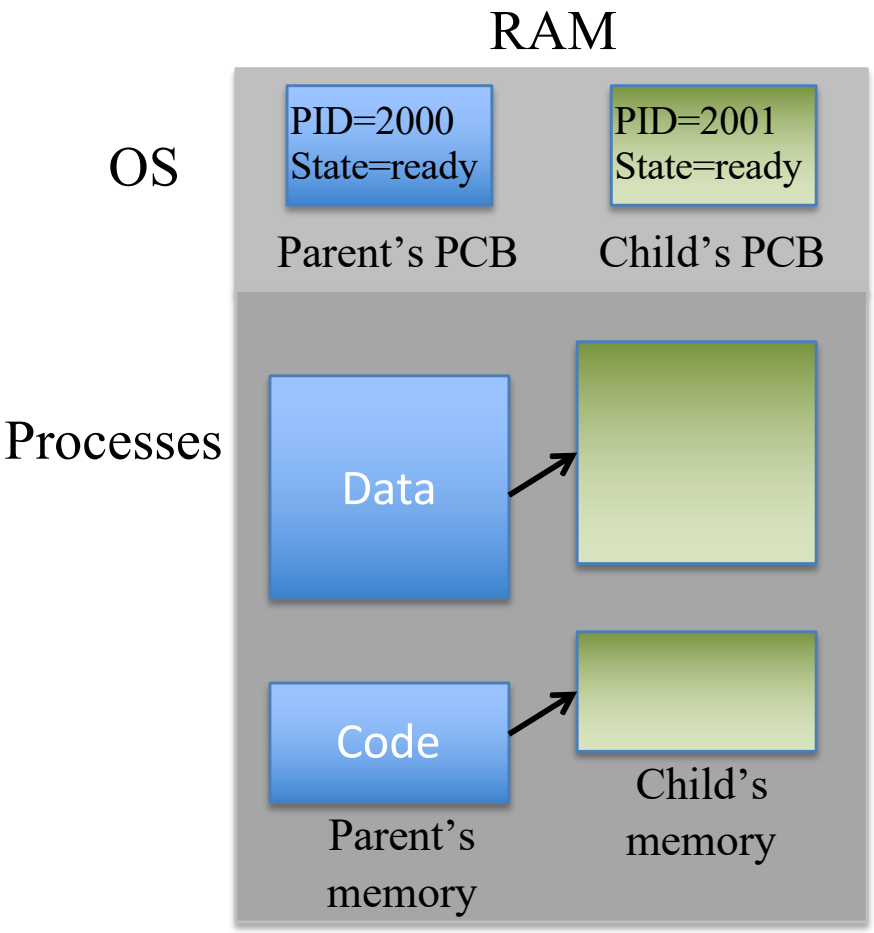
- Logical memory of a process represents all of the logical addresses used by a process
- A process must have physical memory to execute
  - When a process is created and executes, it will be allocated physical memory by the OS
- Logical address are translated to physical addresses
- Changes to process data made by instructions during execution must occur in physical memory
  - Data in physical memory is part of the process execution state and must be maintained by the OS
  - However, the process data is not stored in the PCB!
  - Where is it stored? More to come …

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

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!

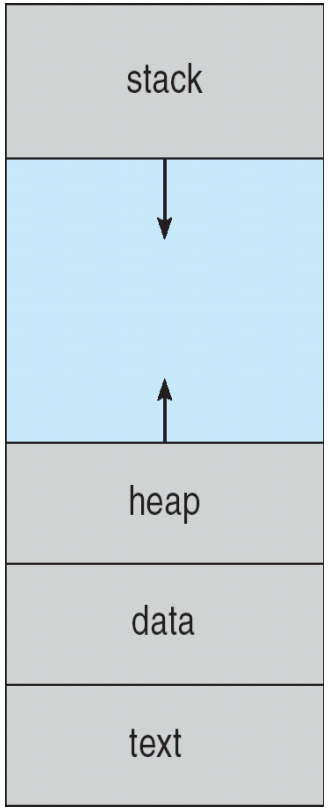
Copying the Parent's Memory

- While it is the case that the child process is initialized with the process address space of the parent, is the actual (logical) memory copied?
  - Logically, it is
  - In practice, it is not
- If the parent's memory (i.e., what it is actually using) is large, the copying cost would be huge
- In practice, what happens is that only the memory reference and (to be) modified is copied
  - This is called "copy on write"
  - Involves magic in the physical memory management
  - More to come …

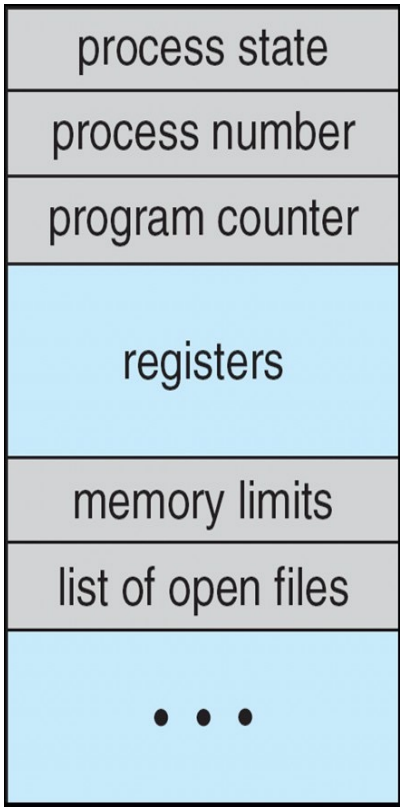
Process Address Space vs. PCB (1/2)

- Be careful not to confuse the two
- Process Address Space
  - This refers to the memory a process can utilize to execute.
  - *The house where a process lives (with rooms: code, stack, heap).*
- Process Control Block (PCB)
  - The PCB is a data structure inside the kernel that stores all information needed to manage a process.
  - *The city hall file about that house (who owns it, current status, utilities connected, etc.).*

Process Address Space



PCB



Process Address Space vs. PCB (2/2)

Aspect	Process Address Space	Process Control Block (PCB)
What it is	The memory “view” of a process (virtual memory layout).	Kernel data structure describing a process.
Where it lives	User space	Kernel space (protected from user access).
Purpose	Holds the content of the program (code, data, stack, heap).	Holds the metadata needed by OS to manage the process.
Accessed by	The process itself (through memory instructions).	The OS scheduler and kernel subsystems.
Changes when	Program loads new code, allocates memory, maps files.	Scheduler switches processes, updates state, tracks resources.

Process Communication

- Process model is a useful way to isolate running programs
  - Separates, isolates, and insulates processes
  - It can simplify programs (do not worry about other processes)
- Process concurrency and cooperation are fundamental to OS
  - Creating new processes increases concurrency
  - Concurrency allows multiple things at the same (logical) time
  - But processes do not always want to work in isolation
- How about creating a larger program that merges multiple processes? (not always a good idea…)
  - Sometimes it is easier to design/develop programs if there are multiple processes working together
- Processes still benefit from isolation and separation of concerns, but need to interact and share information

Multiple Processes

- Generally think of processes running in a system as being either *independent* or *cooperating*
  - Independent process
    - cannot affect or be affected by the execution of another process
  - Cooperating process
    - can affect or be affected by the execution of another process
- Cooperating processes interact with other processes through coordinate actions and sharing data
  - This is called *interprocess communication (IPC)*
- Independent processes do not need to use IPC because they do not interact with each other or other processes
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

Process Communication (Interoperation)

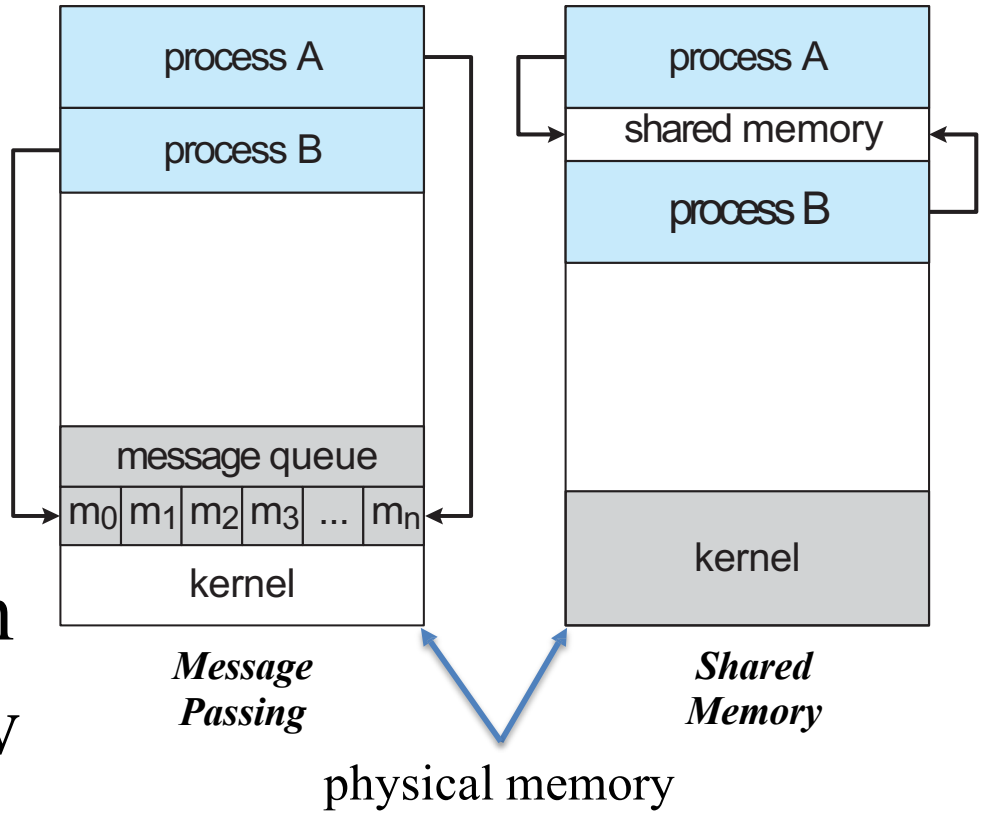
- When is communication necessary?
- Lots of examples in operating systems
  - Client–Server Model
  - Pipelines and Data Processing
  - Background Services
  - Parallelism / Multiprocessing
  - Synchronization Across Processes
  - System Event Notifications
  - Processes manage physical devices

Interprocess Communication (IPC)

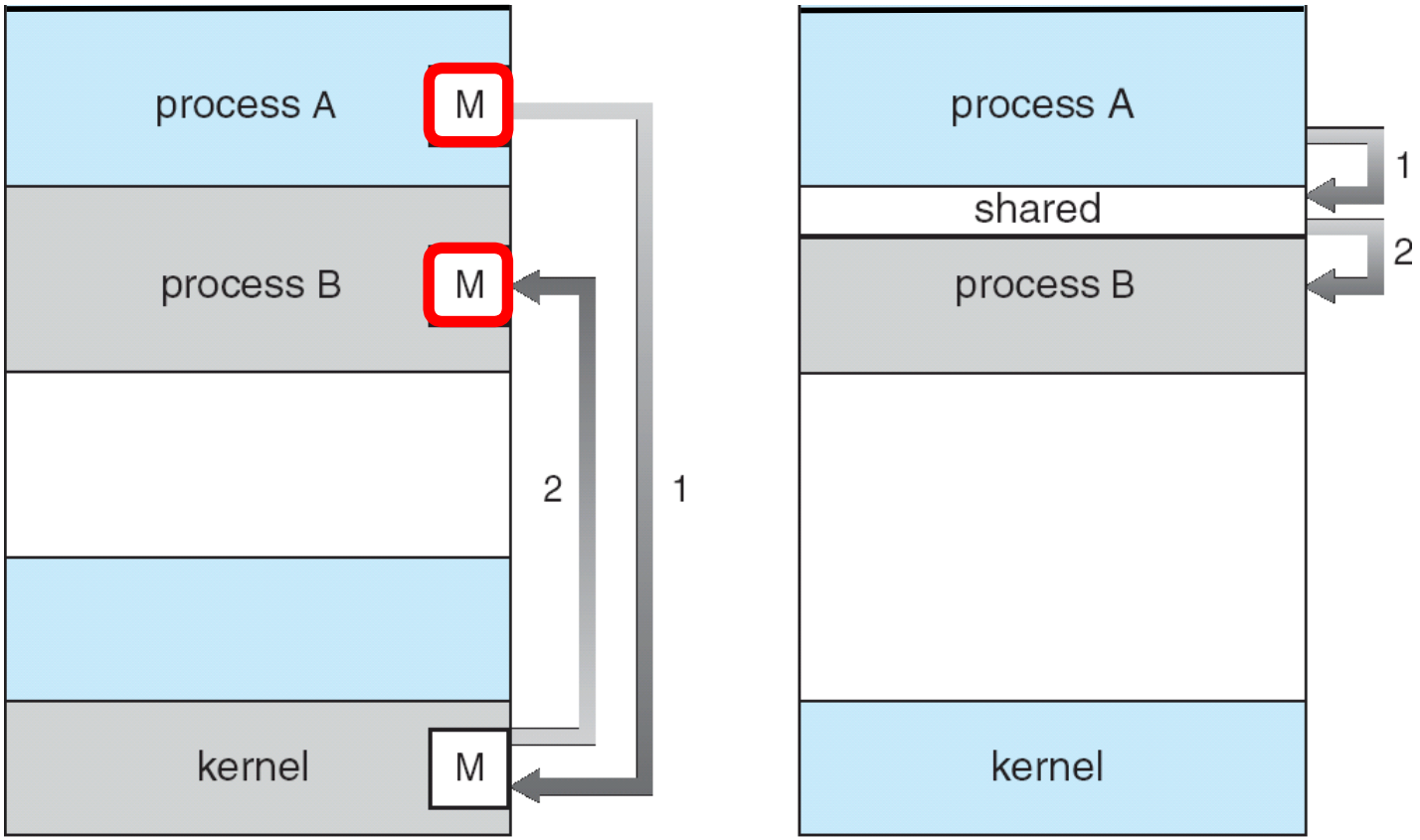
- Mechanism for processes to communicate and synchronize
- Logically, we want some sort of *messaging system*
- Logically, IPC would provide 2 operations to processes:
  - Send(message) (sender process to receiver process)
  - Receive(message) (receiver process from sender process)
- However, first, there needs to be a communication *channel*
  - If processes P and Q wish to communicate, they open a channel
  - Then the exchange of messages can proceed
- How is the communication channel (link) realized?
  - Physically, many forms (memory, storage, networks)
  - Logically, need to define:
    - abstract interfaces and properties
    - protocols for correct communication

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
- We are looking here at how the OS is supporting IPC



Communication Models



Message Passing

Shared Memory

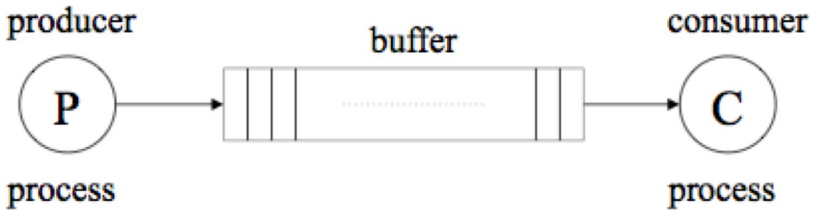
physical memory

Here the kernel maintains the message buffers in its memory. These buffers hold messages (M) sent between processes.

Here a memory area is created by the OS such that each process can reference the data using the same subset of addresses (i.e., shared)

Think Abstractly about the Problem

- Let's first look at shared memory
- Consider two processes sharing a memory region (*)
  - *Producer* writes
  - *Consumer* reads
- Producer action
  - While the buffer is 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?)



* Do not worry how exactly they do this

Properties of IPC

An IPC method can have a combination of the following properties:

- Direct or indirect communication
  - Do the processes directly talk to each other or is there a manager in between?
- Blocking or non-blocking communication
  - What happen when the buffer is full (when we want to send) or empty (when we want to read)?
- Automatic or explicit buffering
  - Who manages the buffer by the programmer explicitly or by the OS (kernel) automatically?

Keep in mind: we should think about these properties for every IPC technique we learnt.

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

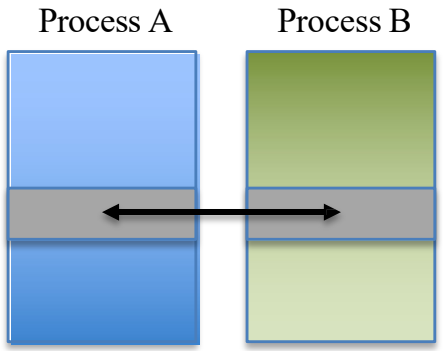
```
item nextConsumed;

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

- What does a shared memory region mean exactly?
- A shared memory segment is a range of logical addresses in the process address space
- If two or more processes "share" the segment, it means that it is the same range of addresses in each process



- In addition, the physical memory allocated to the segment is the SAME for each process
- Now we are going to look at how the OS supports this form of IPC

Creating Shared Memory with mmap()

- More modern and powerful way to map memory
- Used frequently to map files (file → memory)
  - Referred to as ***memory mapped files***
  - Files are mapped to logical memory address region
  - Enhanced performance of file operations
    - Access files as arrays in memory
    - Avoid traditional read() and write() system calls, which require switching between user and kernel modes
- Can also be used to map memory between processes (memory → memory)

This is what we need for IPC, but how do we do this?
(see next a few slides)

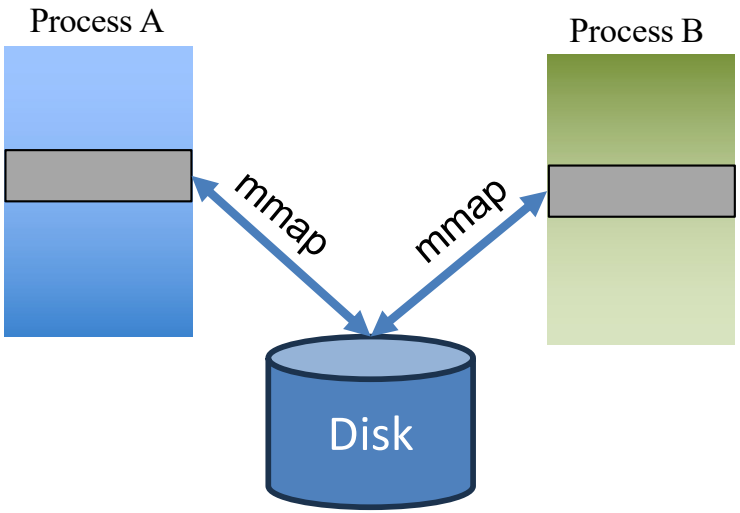
Creating Shared Memory with mmap()

```
MMAP(2)                    Linux Programmer's Manual                    MMAP(2)

NAME
       mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
       #include <sys/mman.h>

       void *mmap(void *addr, size_t length, int prot, int flags,
                  int fd, off_t offset);
       int munmap(void *addr, size_t length);

       See NOTES for information on feature test macro requirements.

DESCRIPTION
       mmap()  creates a new mapping in the virtual address space of the call-
       ing process.  The starting address for the new mapping is specified  in
       addr.   The  length argument specifies the length of the mapping (which
       must be greater than 0).

       If addr is NULL, then the kernel chooses the (page-aligned) address  at
       which to create the mapping; this is the most portable method of creat-
       ing a new mapping.  If addr is not NULL, then the kernel takes it as  a
       hint about where to place the mapping; on Linux, the kernel will pick a
       nearby page boundary (but always above or equal to the value  specified
       by /proc/sys/vm/mmap_min_addr) and attempt to create the mapping there.
       If another mapping already exists there, the kernel picks a new address
       that may or may not depend on the hint.  The address of the new mapping
       is returned as the result of the call.

       The contents of a file mapping (as opposed to an anonymous mapping; see
       MAP_ANONYMOUS  below),  are  initialized using length bytes starting at
       offset offset in the file (or other object) referred to by the file de-
       scriptor fd.  offset must be a multiple of the page size as returned by
       sysconf(_SC_PAGE_SIZE).

       After the mmap() call has returned, the file  descriptor,  fd,  can  be
       closed immediately without invalidating the mapping.

       The  prot  argument describes the desired memory protection of the map-
       ping (and must not conflict with the open mode of the file).  It is ei-
       ther PROT_NONE or the bitwise OR of one or more of the following flags:

       PROT_EXEC  Pages may be executed.

       PROT_READ  Pages may be read.

       PROT_WRITE Pages may be written.

       PROT_NONE  Pages may not be accessed.
```

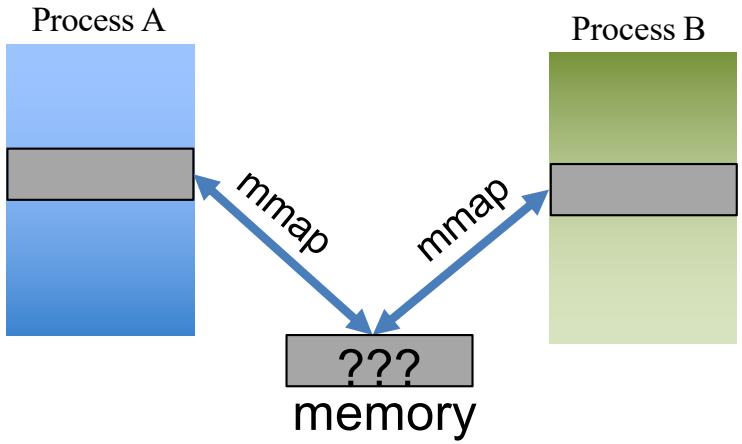
```
   The flags argument
       The  flags argument determines whether updates to the mapping are visi-
       ble to other processes mapping the same region, and whether updates are
       carried through to the underlying file.  This behavior is determined by
       including exactly one of the following values in flags:

       MAP_SHARED
              Share this mapping.  Updates to the mapping are visible to other
              processes  mapping  the  same  region, and (in the case of file-
              backed mappings) are carried through  to  the  underlying  file.
              (To  precisely  control  when updates are carried through to the
              underlying file requires the use of msync(2).)

       MAP_SHARED_VALIDATE (since Linux 4.15)
              This flag provides the same behavior as MAP_SHARED  except  that
              MAP_SHARED mappings ignore unknown flags in flags.  By contrast,
              when creating a mapping using  MAP_SHARED_VALIDATE,  the  kernel
              verifies  all  passed flags are known and fails the mapping with
              the error EOPNOTSUPP for unknown flags.  This  mapping  type  is
              also  required  to  be  able  to  use  some mapping flags (e.g.,
              MAP_SYNC).

       MAP_PRIVATE
              Create a private copy-on-write mapping.  Updates to the  mapping
              are  not  visible  to other processes mapping the same file, and
              are not carried through to the underlying file.  It is  unspeci-
              fied  whether changes made to the file after the mmap() call are
              visible in the mapped region.

       MAP_ANONYMOUS
              The mapping is not backed by any file; its contents are initial-
              ized to zero.  The fd argument is ignored; however, some imple-
              mentations require fd to be -1 if MAP_ANONYMOUS (or MAP_ANON) is
              specified, and portable applications should  ensure  this.   The
              offset  argument  should  be  zero.  The use of MAP_ANONYMOUS in
              conjunction with MAP_SHARED is supported  on  Linux  only  since
              kernel 2.4.

   munmap()
       The munmap() system call deletes the mappings for the specified address
       range, and causes further references to addresses within the  range  to
       generate  invalid  memory references.  The region is also automatically
       unmapped when the process is terminated.  On the  other  hand,  closing
       the file descriptor does not unmap the region.

RETURN VALUE
       On success, mmap() returns a pointer to the mapped area.  On error, the
       value MAP_FAILED (that is, (void *) -1) is returned, and errno  is  set
       to indicate the cause of the error.
```

Creating Shared Memory with mmap()

- mmap can be used to map any files to memory address
- But file I/O is slow, we actually need memory to memory share



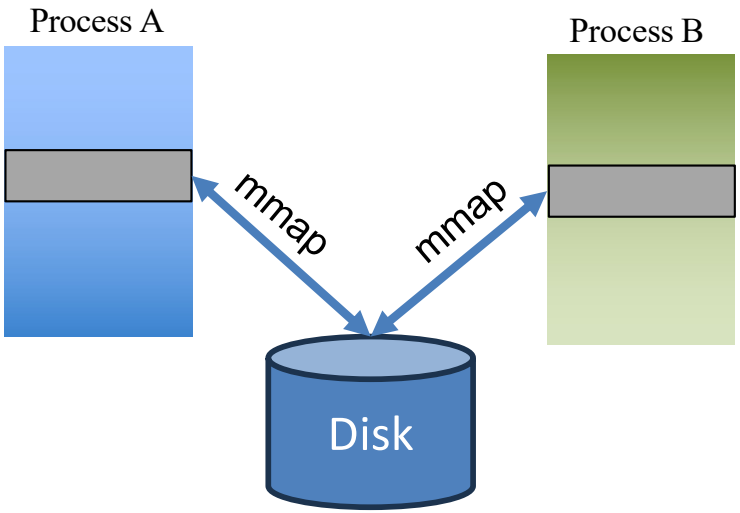
This is slow for IPC



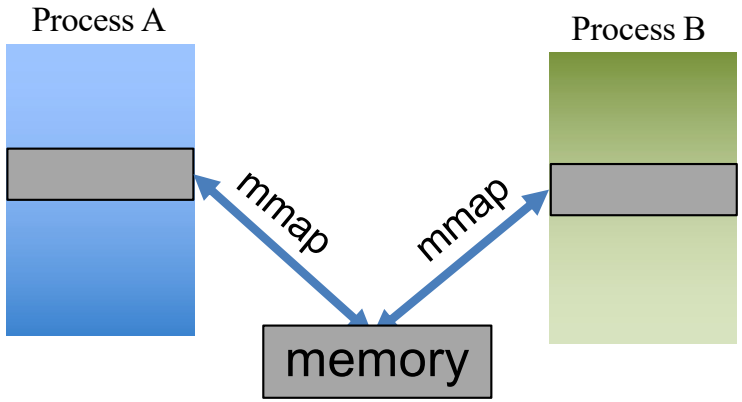
How to achieve this?

Creating Shared Memory with mmap()

- mmap can be used to map any files to memory address
- But file I/O is slow, we actually need memory to memory share



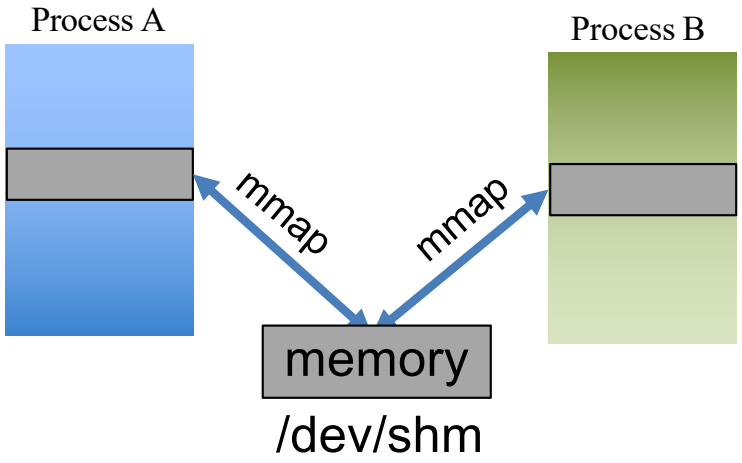
This is slow for IPC



How to achieve this?
Use a memory-backed filesystem
typically mounted at /dev/shm on Linux

Creating Shared Memory with mmap()

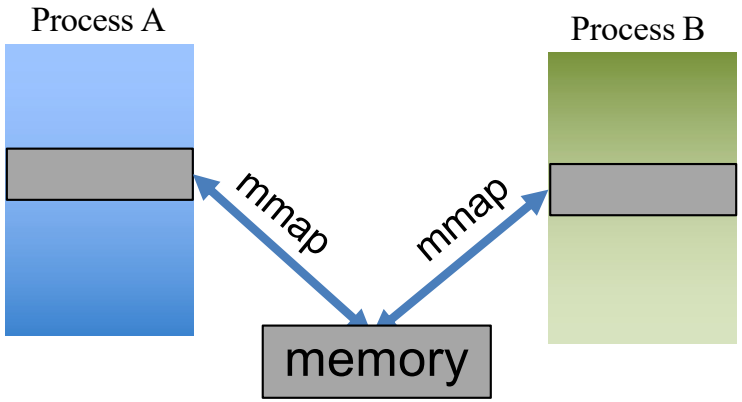
- Memory-backed filesystem
  - Created by Linux kernel.
  - Looks in the special filesystem mounted at /dev/shm.
  - Can creates a file-like object using shm_open()
  - Returns a file descriptor (fd) for it — just like open() would for a real file.
  - You can call ftruncate(), read(), write(), mmap(), close(), etc.
  - But it's not stored on disk; it's in RAM (tmpfs filesystem).



 See code example

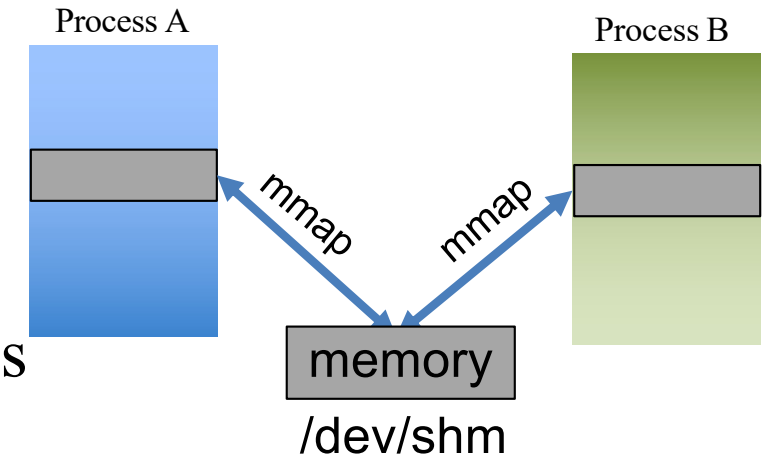
Creating Shared Memory with mmap()

- Many types of shared memory can be created with mmap()
- Two common types
  - Named shared memory
  - Anonymous shared memory



Creating Shared Memory with mmap()

- Named shared memory
  - Has a name in the POSIX shared memory namespace (usually /dev/shm/my_shm).
  - Backed by a file descriptor from the tmpfs filesystem.
  - Can be opened by unrelated processes using the same name.
  - Exists even if processes that created it exit — until explicitly shm_unlink()ed.

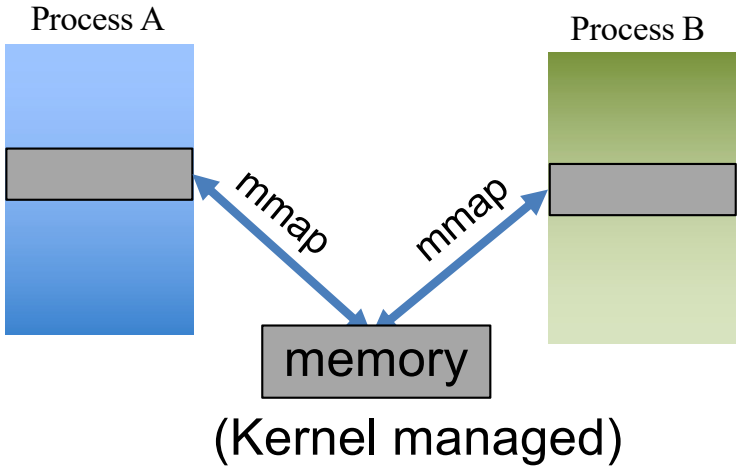



See code example

```
int fd = shm_open("/my_shm", O_CREAT | O_RDWR, 0666);
ftruncate(fd, size);
void *ptr = mmap(NULL, size, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
```

Creating Shared Memory with mmap()

- Anonymous shared memory
  - No name — it's not registered anywhere in /dev/shm.
  - Not backed by a file descriptor.
  - In memory directly managed by kernel.
  - Can only be shared between a parent and child after fork() — because the mapping is duplicated across them.
  - Exists only while at least one process has it mapped.



See code example: shmem-*.c

```
void *ptr = mmap(NULL, size,
                 PROT_READ | PROT_WRITE,
                 MAP_SHARED | MAP_ANONYMOUS,
                 -1, 0);
```

Creating Shared Memory with mmap()

Property	Named	Anonymous
Backing store	tmpfs (RAM-based file system)	Kernel memory
Creation speed	Slightly slower (requires shm_open and permissions)	Slightly faster
Persistence	Until unlinked	Until unmapped
Accessibility	Accessible by name (shm_open)	Only via inheritance (fork)
Security	File permissions control access	Inherited, not visible globally
Common use	Independent process IPC	Parent–child IPC

Properties of shared memory

An IPC method can have a combination of the following properties:

- Direct or indirect communication
  - Do the processes directly talk to each other or is there a manager in between?
- Blocking or non-blocking communication
  - What happen when the buffer is full (when we want to send) or empty (when we want to read)?
- Automatic or explicit buffering
  - Who manages the buffer by the programmer explicitly or by the OS (kernel) automatically?

IPC with Pipes (OS managed shared memory)

- Pipe is a kernel-managed circular buffer (shared memory)
- 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 between processes
  - It is a shared memory segment, but the OS manages the pipe
  - System calls: *pipe(2)*, *dup(2)*, *popen(2)*
- Producer
  - Writes to the pipe (i.e., buffer), if space is available
- Consumer
  - Reads from the pipe (i.e.,buffer) if data is available

IPC with Pipes (OS managed shared memory)

```
int fd[2];
pipe(fd);   // fd[0] = read end, fd[1] = write end
```

See code example: pipe.c

Writer process → [kernel pipe buffer] → Reader process

Uses file descriptors and read/write

```
int fd[2];
pipe(fd);
if (fork() == 0) {          // child
   close(fd[1]);
   char buf[100];
   read(fd[0], buf, sizeof(buf)); // blocks until parent writes
}
else {                      // parent
   close(fd[0]);
   sleep(2);
   write(fd[1], "hi", 2);  // unblocks child
}
```

How to find out the pipe buffer size:
cat /proc/sys/fs/pipe-max-size

← Blocks read if pipe is empty

Default to be blocking (can be disabled)

← Blocks write if pipe is full

IPC with Pipes (OS managed shared memory)

Writer process → [kernel pipe buffer] → Reader process

Feature	Pipe	POSIX Shared Memory (shm_open + mmap)
Purpose	Stream-based data transfer between processes: Sequential access (like a file)	Share a region of memory between processes: Random access (like an array in memory)
Communication Type	One-way byte stream (FIFO buffer)	Shared access to the same memory segment
Persistence	Exists only while processes are running	Can persist until explicitly shm_unlink()
Speed	Slower (data copied via kernel buffer)	Faster (direct access to shared memory)
Synchronization	Implicit — blocking reads/writes	Explicit — need semaphores/mutexes
Data Direction	Typically unidirectional (use two for bidirectional)	Fully bidirectional (both can read/write)
Best For	Streaming data, command pipelines, producer/consumer with simple messages	Large data sharing, frequent access, shared state, shared buffers
Backed By	Kernel-managed buffer	Anonymous file in /dev/shm (tmpfs memory)

Properties of pipe

An IPC method can have a combination of the following properties:

- Direct or indirect communication
  - Do the processes directly talk to each other or is there a manager in between?
- Blocking or non-blocking communication
  - What happen when the buffer is full (when we want to send) or empty (when we want to read)?
- Automatic or explicit buffering
  - Who manages the buffer by the programmer explicitly or by the OS (kernel) automatically?

Why do I care about shared memory?

- Why would you care to use shared memory IPC?
  - Think about addressing memory
- Using shared memory IPC is fast
  - Data is being accessed directly from memory
  - Either access using process addresses (random access)
  - … or through an abstract OS interface (such as pipe())
- When can I used shared memory IPC?
  - Think about where the processes are running
- Shared memory pipe mechanism is functionality provided by the OS
  - Efficient, performant, correct, supports pipe semantics, …
- However, a shared memory segment could be used directly by the processes
  - Do so at your own risk!

IPC with Message Passing (1)

- IPC provides 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*

IPC with Message Passing (2)

- Issues
  - What if a process wants to receive from any process?
  - What if communicating processes are not ready at same time?
  - What size message can a process receive?
- Advantages of IPC with message passing
  - Allows processes to be on different machines!

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

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

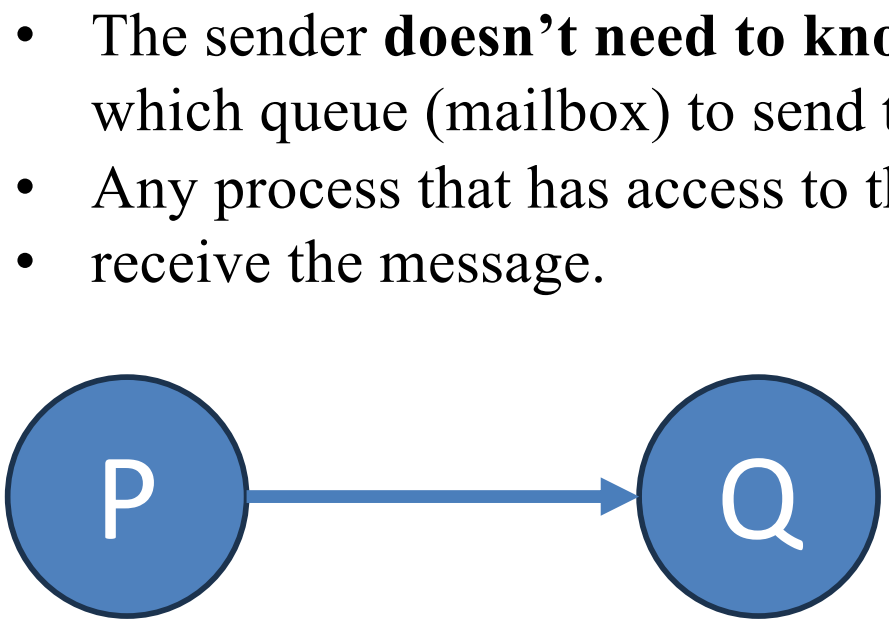
Direct vs. indirect communication

Direct communication

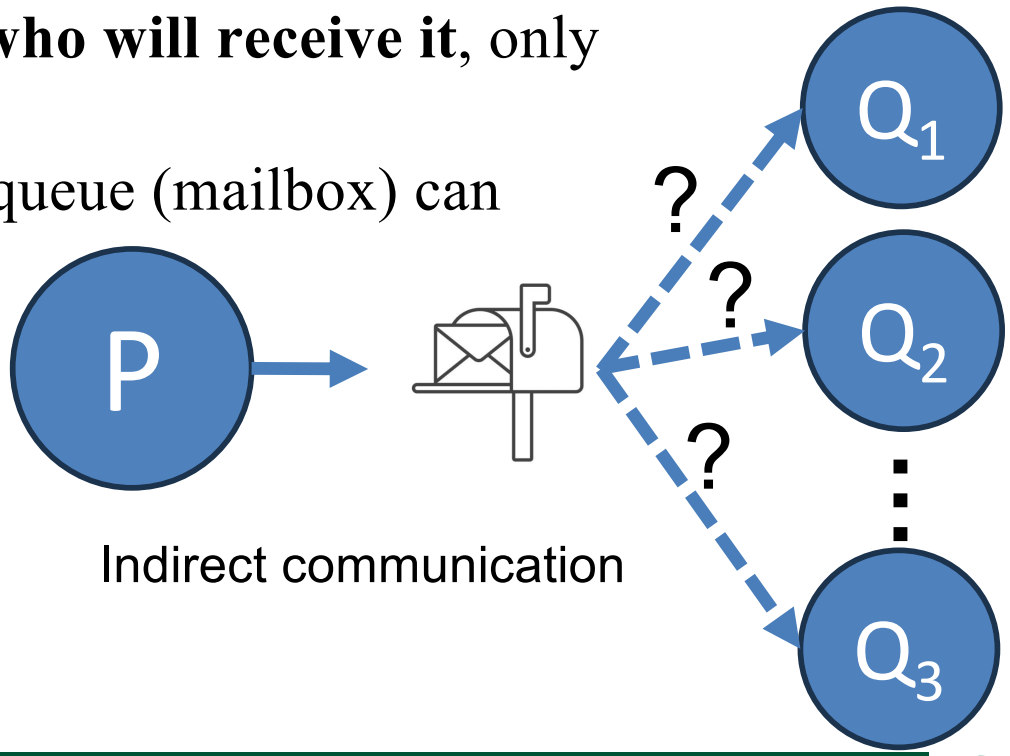
- The sending process specifies the **exact recipient process**.
- Example: send(p1, p2, message) → p1 knows p2 explicitly.

Indirect communication:

- The sending process sends the message to an **intermediary** (like a mailbox or queue).
- The sender **doesn't need to know who will receive it**, only which queue (mailbox) to send to.
- Any process that has access to that queue (mailbox) can
- receive the message.



Direct communication



Indirect communication

Blocking Communication

- Blocking send
  - *Send(P, message)*
  - The sending process is blocked until the message is received by the receiving process or by the mailbox
- Blocking receive
  - *Receive(Q, message)*
  - The receiver blocks until a message is available
- Blocking means that both have to be ready!
  - For direct communication, both have to be ready. Otherwise, the process (sender or receiver) is blocked.
  - For indirect communication, the mailbox must be ready (not full for sender and not empty for receiver)

Non-blocking Messaging

- Non-blocking send
  - *Send(M, message)*
  - Sender tries to send a message to a mailbox
  - No waiting (if mailbox is full, immediately return error)
  - Execution continues regardless of results
- Asynchronous receive
  - *Receive(M, message)*
  - Receive a message from a specific a mailbox
  - No waiting (if mailbox is empty, immediately return error)
  - Execution continues regardless of results
- Asynchronous means that you can send/receive when you're ready
  - The communication may not complete

Blocking vs Non-blocking

- 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

IPC Message Passing Mechanisms

- What mechanisms support IPC message passing?
  - Keep in mind that these mechanisms are being implemented by the OS
- POSIX Message Queues
- Sockets (ubiquitous, layed on network protocols)
- Remote procedure calls (RPC)
- Remote method invocation (Java)

IPC with POSIX Message Queues

- A POSIX message queue (MQ) is a kernel-supported, named queue for passing messages between processes on the same system.
- It's part of the POSIX real-time extensions (<mqueue.h>).
- Messages are discrete units, unlike pipes (which are streams of bytes).
- Supports blocking and non-blocking operations, message priorities, and optional asynchronous notification.
- Works between unrelated processes as long as they know the queue name.

IPC with POSIX Message Queues

- A POSIX message queue (MQ) is a kernel-supported, named queue for passing messages between processes on the same system.
- It's part of the POSIX real-time extensions (<mqueue.h>).
- Messages are discrete units, unlike pipes (which are streams of bytes).
- Supports blocking and non-blocking operations, message priorities, and optional asynchronous notification.
- Works between unrelated processes as long as they know the queue name.

IPC with POSIX Message Queues

- Create/Open a Queue

```
mqd_t mq = mq_open("/myqueue", O_CREAT | O_RDWR, 0666, &attr);
```

- Send a Message

```
mq_send(mq, "Hello", 5, 0);  // 0 = priority
```

- Receive a Message

```
char buffer[128];
mq_receive(mq, buffer, sizeof(buffer), NULL);
```

- Close and Unlink

```
mq_close(mq);
mq_unlink("/myqueue");  // removes the queue
```

See code example: mq-*.c

Properties of Message Queue

An IPC method can have a combination of the following properties:

- Direct or indirect communication
  - Do the processes directly talk to each other or is there a manager in between?
- Blocking or non-blocking communication
  - What happen when the buffer is full (when we want to send) or empty (when we want to read)?
- Automatic or explicit buffering
  - Who manages the buffer by the programmer explicitly or by the OS (kernel) automatically?

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

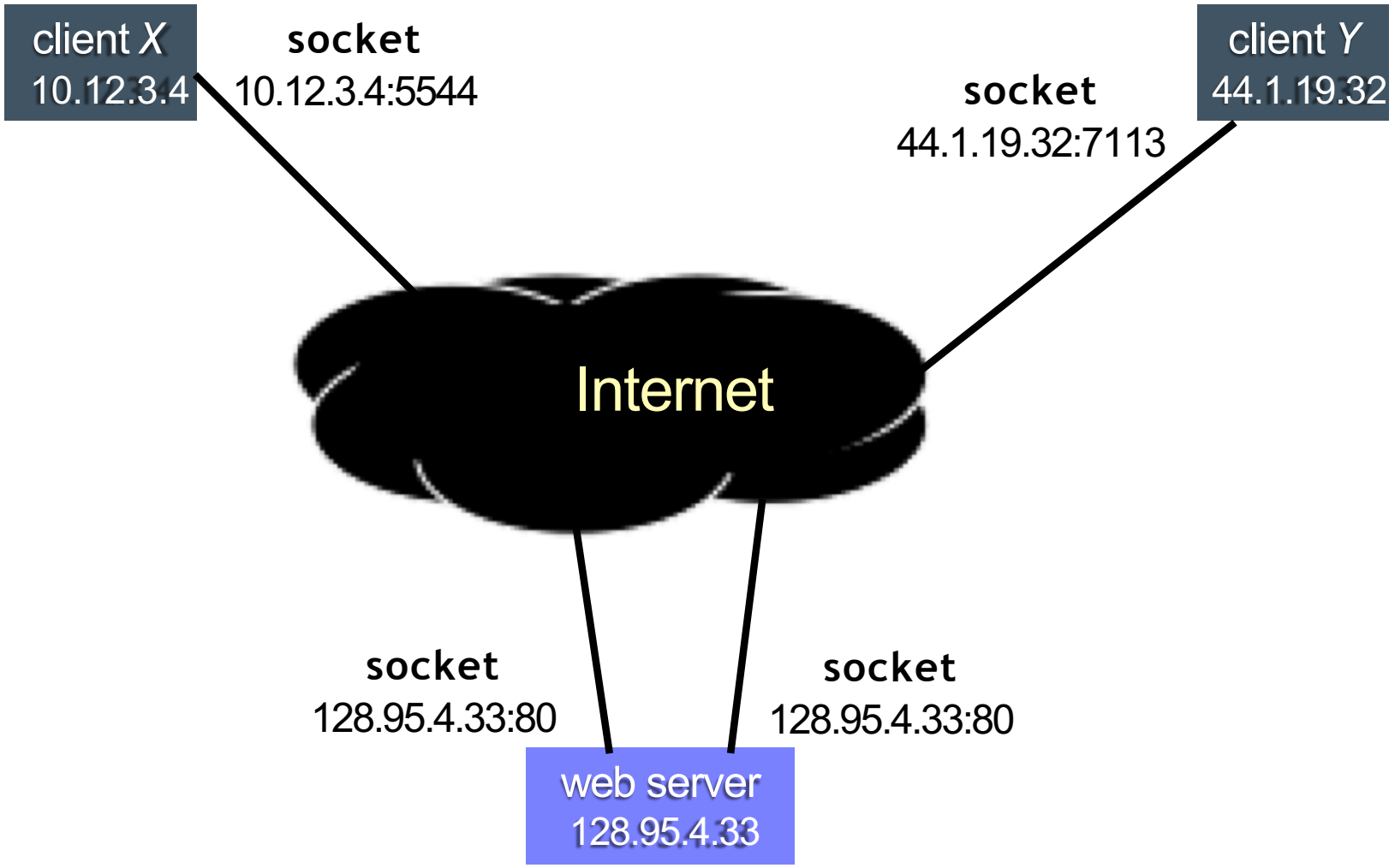
Networks and Sockets

- 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
  - A 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

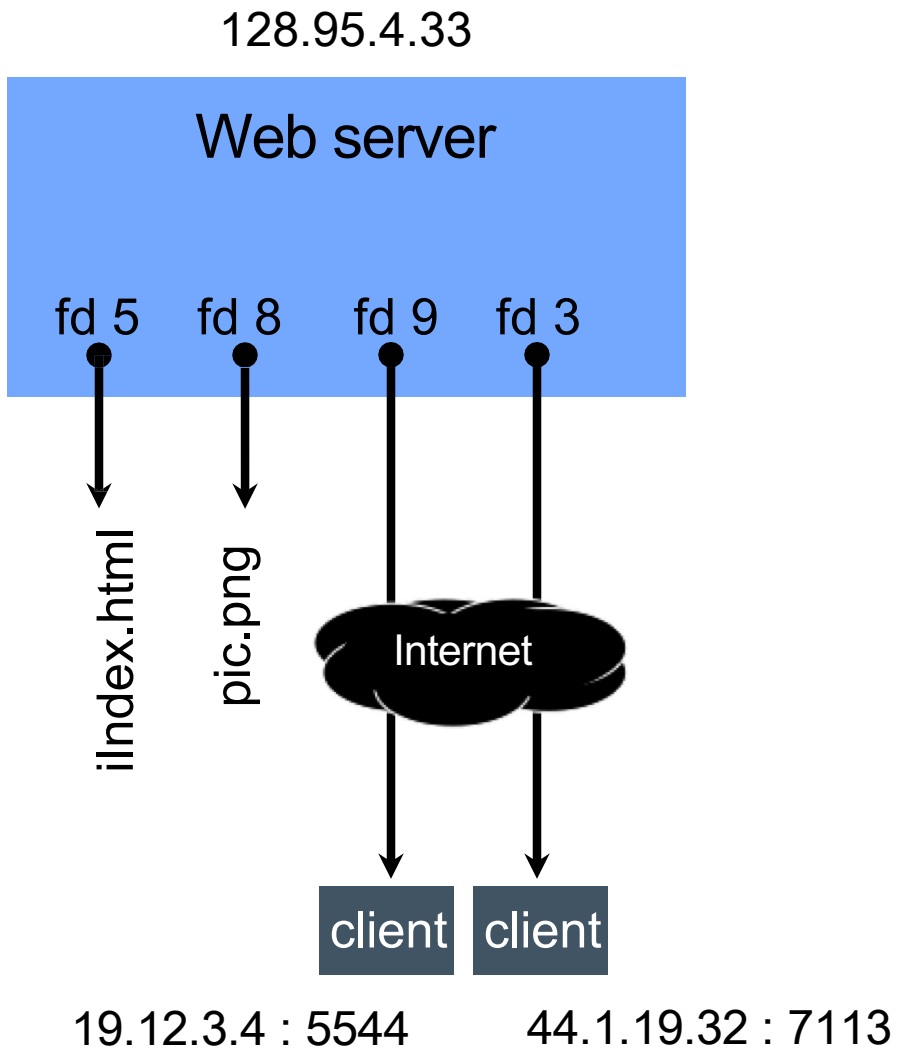
Examples of Sockets and Their Use

- 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

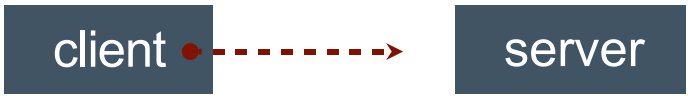
OS's file descriptor table

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

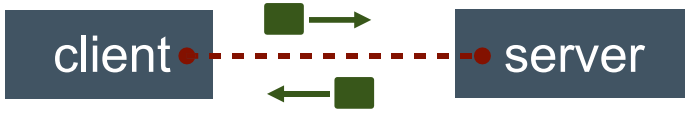
Stream Sockets

- 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



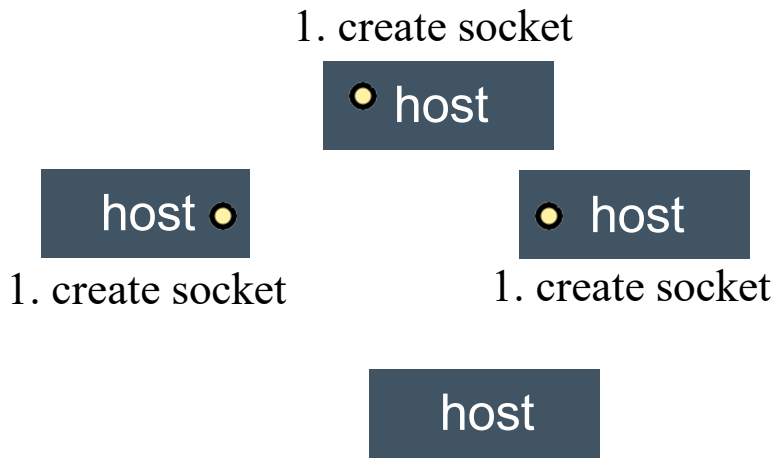
2. Communicate

- Server
  - An application that receives connections from clients

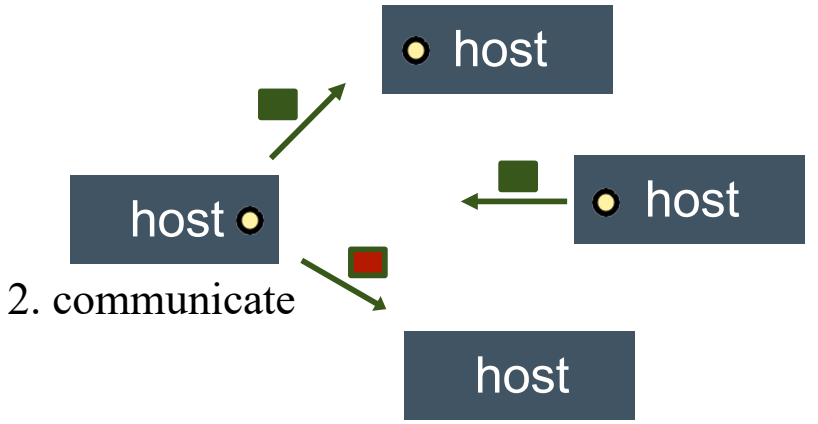
3. Close connection

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
- Big benefit is that processes can be on different machines

Properties of Socket

An IPC method can have a combination of the following properties:

- Direct or indirect communication
  - Do the processes directly talk to each other or is there a manager in between?
- Blocking or non-blocking communication
  - What happen when the buffer is full (when we want to send) or empty (when we want to read)?
- Automatic or explicit buffering
  - Who manages the buffer by the programmer explicitly or by the OS (kernel) automatically?

Feature	Shared Memory	Pipe	Message Queue	Socket
Communication Type	Direct (shared region in memory)	Indirect (via kernel buffer)	Indirect (via kernel-managed queue)	Indirect (via kernel network stack)
Data Model	Memory region	Byte stream (FIFO)	Discrete messages	Stream or datagram
Directionality	Bidirectional (shared read/write)	Unidirectional (unless using two pipes)	Bidirectional (depending on queue usage)	Bidirectional
Kernel Involvement	Only for setup & teardown	For every read/write	For every send/receive	For every send/receive
Persistence	Until shm_unlink()	Until all descriptors closed	Until removed or reboot	As long as socket connection exists
Synchronization	Explicit (need semaphores)	Implicit (blocking)	Implicit (message ordering)	Implicit (protocol-managed)
Speed	Fastest (no copy, direct access)	Moderate (kernel copies)	Moderate	Slowest (network protocol overhead)
Typical Scope	Local system	Local, parent–child	Local system	Local or network (can cross machines)
Data Size Suitability	Large or shared structures	Small/medium	Small/medium	Variable, including large streams
Ease of Use	Complex (requires sync)	Simple	Moderate	Moderate to complex
Reliability	Depends on programmer's sync	Reliable	Reliable (kernel-queued)	Reliable (TCP) or unreliable (UDP)

Remote Procedure Calls (RPC)

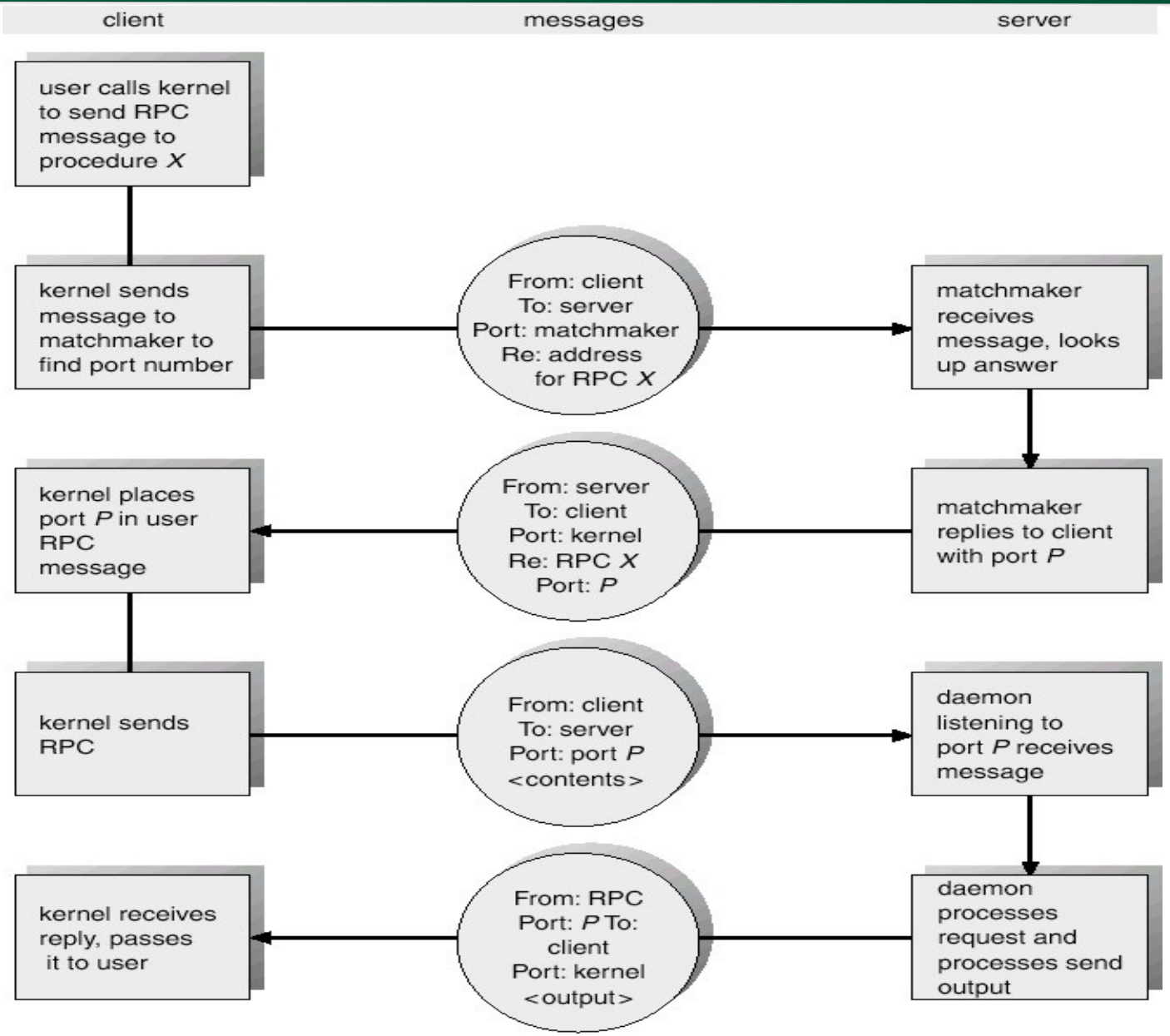
- Procedure calls between processes on network
  - Looks like a "normal" procedure call
  - However, "called" procedure is run by another process
- RPC dates back to early distributed systems

  https://en.wikipedia.org/wiki/Remote_procedure_call
- RPC mechanism was used in BSD (Berkeley Software Division) Unix to develop network files systems (NFS)
  - Ability to access files stored on another machine

RPC Mechanism

- Client – Server mechanism
- Client stub (proxy for procedure call on server)
  - Client stub "marshalls" arguments
  - Client stub find destination (server) for RPC
  - Client stub sends call + marshalled arguments to server
- Server stub (performs the actual procedure call)
  - Server stub "unmarshalls" arguments
  - Server stub calls actual procedure with arguments
  - Server stub "marshall" result and returns to client

Remote Procedure Calls

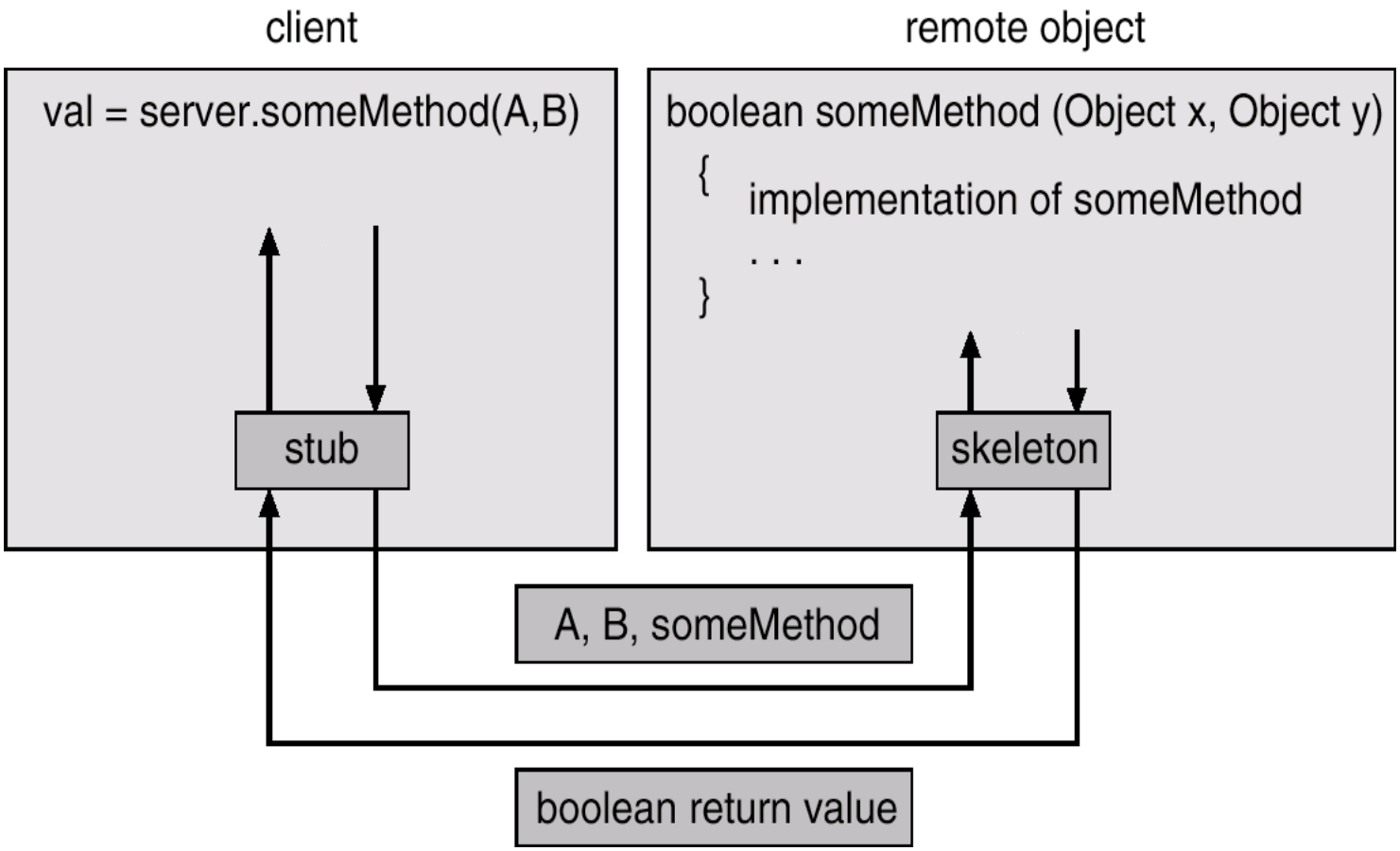


Remote Procedure Calls

- Supported by systems
  - Unix / Linux
  - Distributed system platforms (e.g., CORBA)
  - Java RMI (Remote Method Invocation)
- 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?

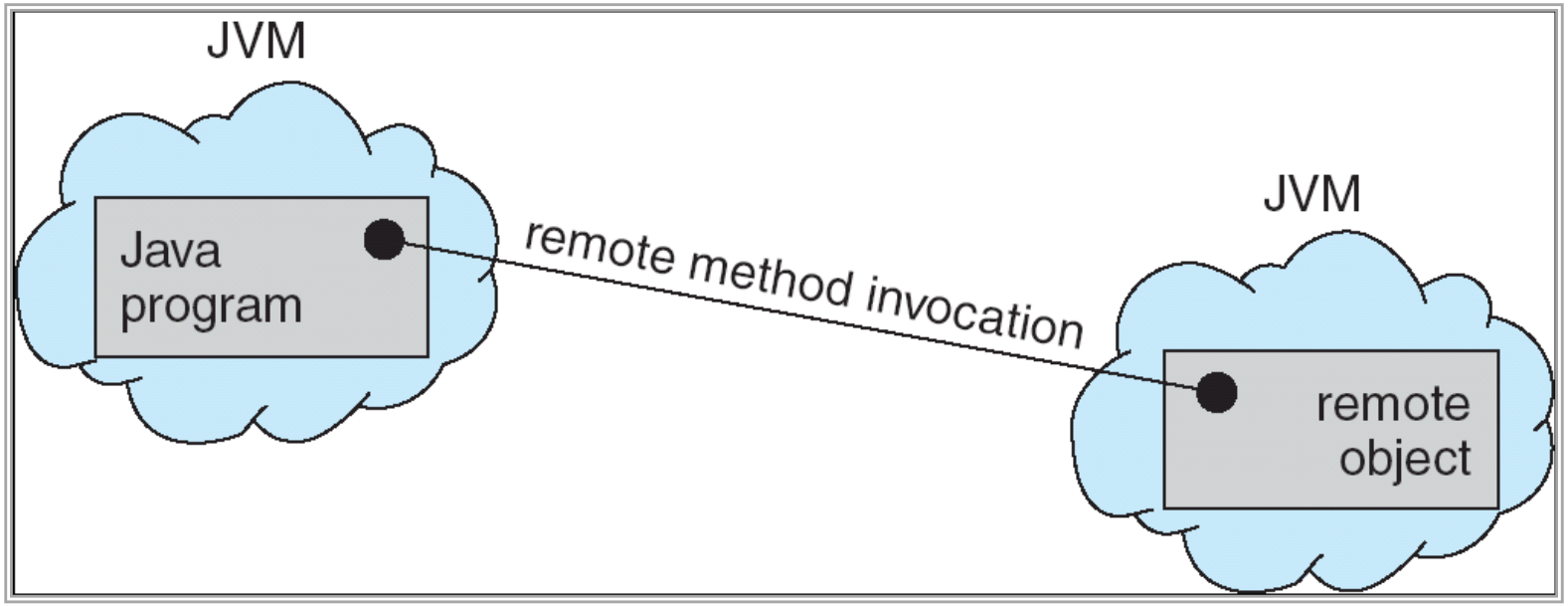
Remote Procedure Calls

- 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

  } Base on OS shared memory support

  - Message Queue
  - Sockets
  - RPC

  } Base on OS message communication support

- 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