Start of Lecture on January 20, 2014

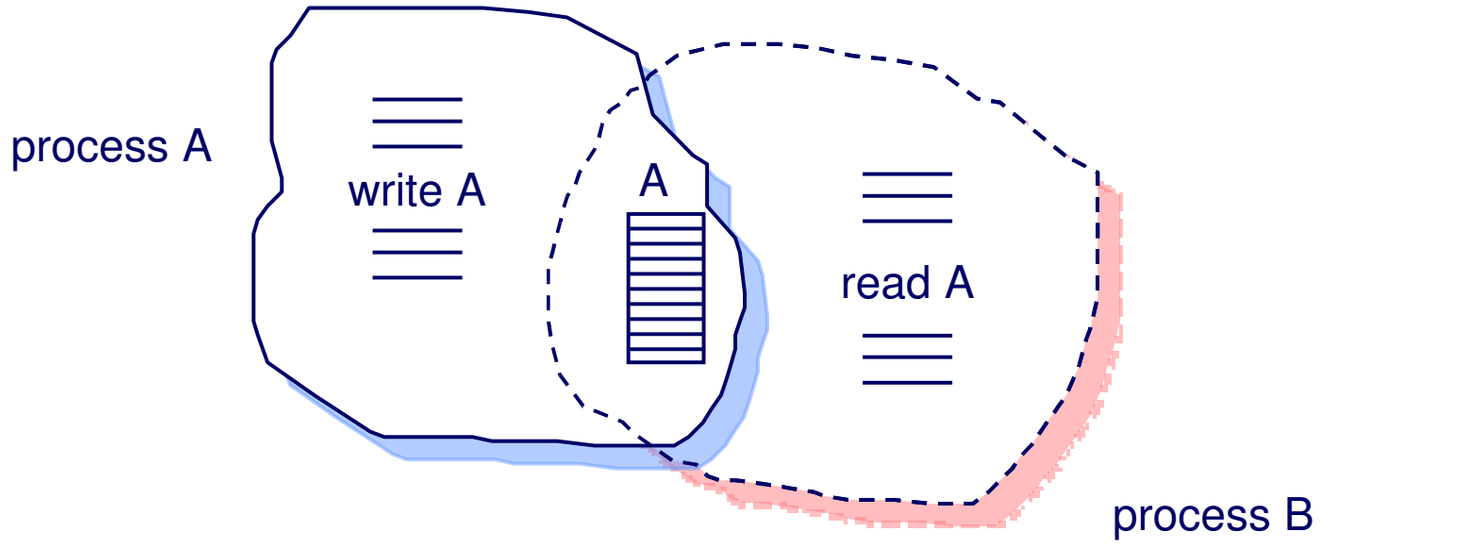
Reminders

- Exercise 1 is **due in the morning before class** on Wed
- Don't forget about Assignment 1
- What are your questions or comments?

Shared memory versus Message Passing

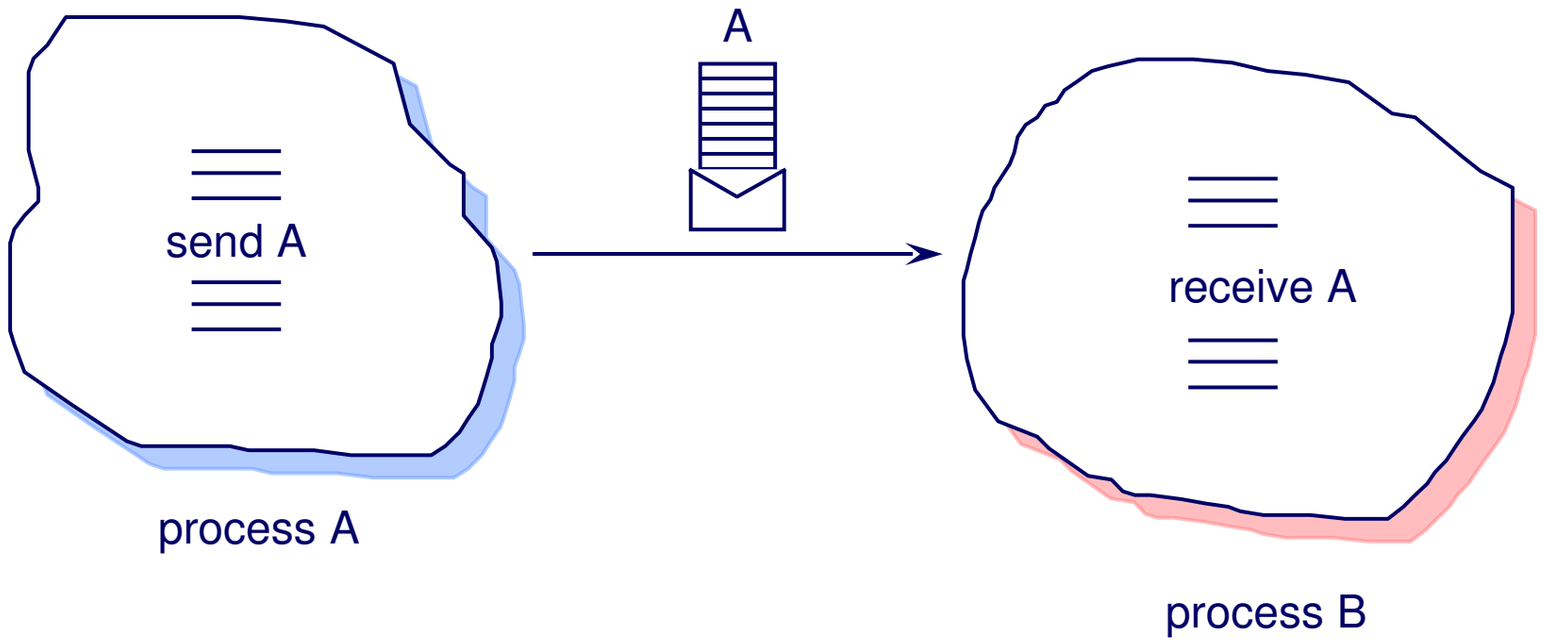
shared memory

(e.g., UNIX pipes)



message passing

(e.g., UNIX sockets)



IPC - Shared Memory

- A system call is usually issued to obtain a block of shared memory
- After initial system call, communication under control of user processes, not the operating system
- Major issue is to provide mechanism that enables processes to *synchronize* their actions when using shared memory, e.g. avoid write to same location
- We will discuss synchronization in detail later

Creating shared memory: POSIX API

Creating and writing to shared memory (in file producer.c):

```
const char *name = “OurSharedMemory”;
int shm_fd;
void *ptr;
/* create the shared memory object; system call */
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
/* configure the size of the shared memory object; system call */
ftruncate(shm_fd, SIZE);
/* memory map the shared memory object; system call */
ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
/* write to the shared memory object */
sprintf(ptr, “%s”,”Hello World”);
```

Creating and writing to shared memory (in file consumer.c) (same vars declared):

```
shm_fd = shm_open(name, O_RDONLY, 0666);
ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
/* read from the shared memory object */
printf(“%s”,(char *)ptr);
```

IPC - Message Passing

- A **message** can be a structured (language) object specified by type (e.g. string) or stream of information specified by its size/length
- Two basic operations on messages:
  - **send()** — transmission of a message
  - **receive()** — receipt of a message
- OS component which implements these operations is called a **message passing system**

Fundamental questions

- When a message is sent, does the sender wait until the message is received or can it continue executing?
- What happens if a process executes a receive(), but no message has been sent?
- Can a message be sent to *one* or *many* process?
- Does a receiver identify the sender of the message or can it accept messages from any sender?
- Where messages kept while in transit? How many stored?
- …

Design issues

When addressing these fundamental questions, the following are important design issues of a message passing system:

- **Form of communication** — message can be sent *directly* to recipient or *indirectly* through intermediate object
- **Buffering** — how and where messages are stored
- **Error handling** — how to deal with exception conditions

Direct communication

- Processes must explicitly name the receiver or sender of a message (**symmetric addressing**)
  - **send**(*P, message*) — Send *message* to process *P*
  - **receive**(*Q, message*) — Receive *message* from process *Q*
- In a **client-server system**, server does not know the name of a specific client in order to receive a message; use variant of receive that is **asymmetric addressing**
  - **send**(*P, message*) — Sender still needs to know recipient process
  - **listen**(*ID, message*) — Receive a **pending** (posted) *message* from any process; when *message* arrives, ID is set to the name of the sender

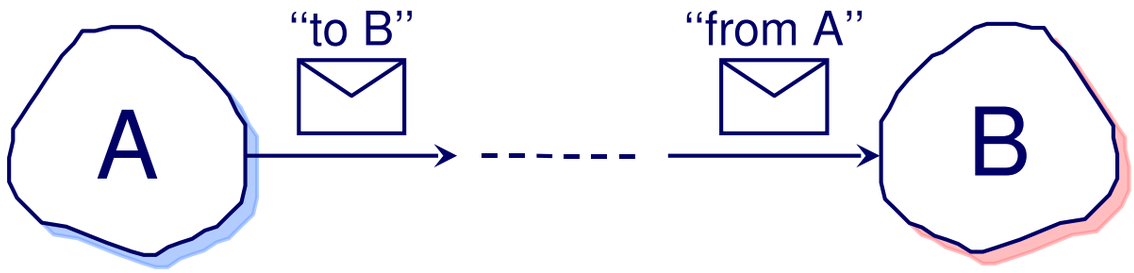
Properties of direct communication links

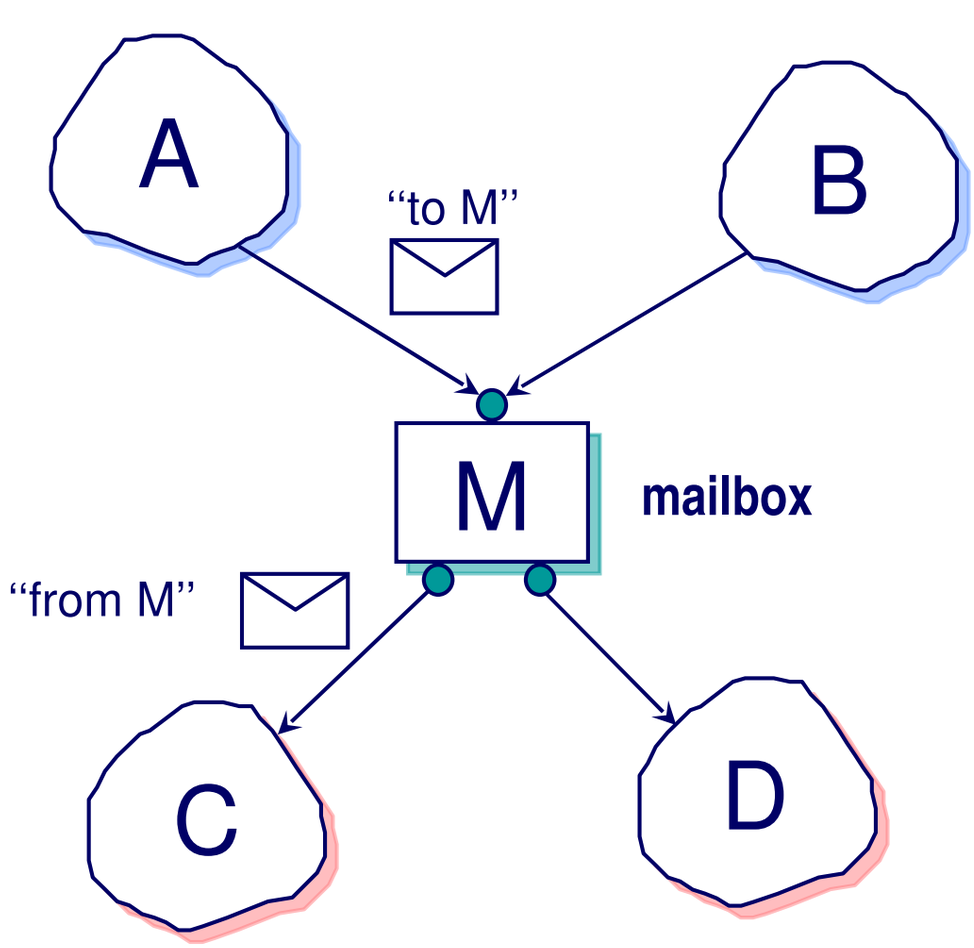
- Direct communication with symmetric addressing:
  - A link is automatically established between each pair of processes that know each other's identities and that try to communicate
  - Exactly one link is associated with exactly two processes
  - Link usually bi-directional, but can be uni-directional
  - Even for asymmetric addressing, sender knows receiver identity, so the link can be automatically established
- **Issue**: must explicitly state communicating process identifiers, hard-code solutions usually less desirable

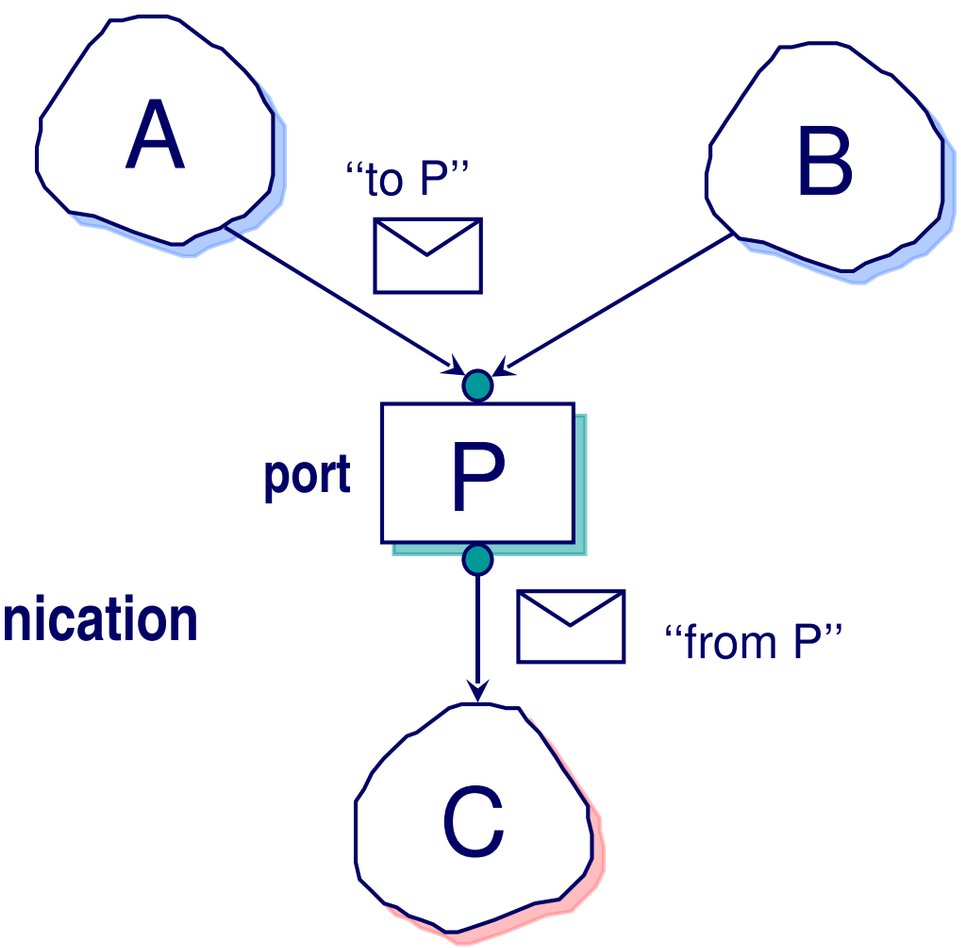
Indirect communication

- For indirect communication, messages sent to **mailboxes**; a message can be retrieved from this repository
  - **send**(*A*, *message*) — Send *message* to mailbox *A*
  - **receive**(*A*, *message*) — Receive *message* from mailbox *A*
- This form of communication decouples sender and receiver, which is more flexible
- Usually mailbox associated with many senders/receivers; if (statically) associate one receiver with a particular mailbox, then that mailbox is often called a **port**

Message Passing: Direct vs Indirect







Exercise: Review Questions

- Turn to your neighbour and
  - one of you explain to the other what happens when user asks kernel to make an "open" system call from c code
  - the other explain steps in OS running multiple programs on one CPU, focusing on how it switches between the programs
- Discuss the following thought questions:
  - What is the difference between having processes communicate through shared memory versus using threads that by default share memory?
  - As message-passing is typically implemented using system calls for each message, what is a disadvantage of message passing over shared memory?
  - In a multiprocessing environment, where each CPU has its own cache, what is a disadvantage of using shared memory?

Properties of indirect communication links

- A link is established between two processes if they have a shared mailbox
- Each pair of communicating processes may have more than one link, which each link corresponding to a different mailbox
- More than two processes could be linked, if they all opened the same shared mailbox

Design questions for indirect communication

- Suppose process P1, P2 and P3 share mailbox A. When P1 sends a message to A, and both P2 and P3 execute receive(), who gets the message?
- Possible design decisions:
  - Allow a link to be associated with at most two processes
  - Allow at most one process at a time to execute receive()
  - Allow system to decide who receives the message (e.g. round robin)

Mailbox Example: POSIX message queues

- Create/open mailbox: mq_open()
- Transfer message to queue: mq_send()
- Receive message from queue: mq_receive()
- Process currently done using mailbox: mq_close()
- Process permanently done using mailbox: mq_unlink()

Synchronization of communication

Message passing may be either **blocking** or **non-blocking**

- **Blocking** is considered **synchronous**
  - **Blocking send** — sender is blocked until message is received
  - **Blocking receive** — receiver is blocked until message available
- **Non-blocking** is considered a**synchronous**
  - **Non-blocking send** — sender sends message and continues
  - **Non-blocking receive** — receiver receives valid message or NULL message
- Different combinations possible
  - If both send and receive are blocking, its called a **rendezvous**

Buffering

- For both direct and indirect, messages reside in a temporary queue
- Queues can be:
  - **Zero capacity** — no buffering; sender must block (i.e. wait) until receiver receives the message (rendezvous)
  - **Bounded capacity** — queue of finite length n; sender has to block if queue full, until space is available
  - **Unbounded capacity** — queue of essentially infinite length; sender never blocks

Terminology we have learned

- direct vs indirect communication
- symmetric versus asymmetric addressing
- synchronous vs asynchronous (i.e. blocking vs non-blocking)
- The terminology is not specific to an implementation of IPC, but rather generic for describing IPC design

Clarifications on overlapping terms

- Port is used for mailboxes (indirect communication)
- Port is also used for sockets (direct communication); but because for sockets there is a port on either end, the receiver (server) knows exactly what port sent the information and creates a unique link to that specific port. Try not to get too confused by this.
- There is a distinction between the logical description of the communication (i.e. policy: how it should behave) versus actual implementation (i.e. mechanism)
  - e.g. OS could implement message passing with shared memory; the user does not see mechanics. Since they do not explicitly create shared memory and use message passing protocols, it is message passing IPC

Case Study: UNIX signals

- Signals are one of the oldest IPC methods in UNIX
- Recall, the operating system (and some processes) can send a process a signal (e.g. SIGINT, SIGKILL, SIGSEGV)
- Process can setup a signal handler to receive signals (a somewhat strange form of a receive function)

Case Study: UNIX signals — SIGSEGV

- Send operation — `kill(int pid, SIGSEGV);`
  - kernel updates descriptor of destination process
- Receive operation — `struct sigaction sa` has defined function, `sa.sa_handler`, that receives the message
  - replaced default handler with `sigaction(SIGSEGV, &sa, NULL)`
  - kernel updates by adding signal to `signal` field in `task_struct` of process; then continues work
  - Before process resumes execution in user mode (e.g. context switch back onto CPU or returned from system call in kernel mode), this signal field is checked by kernel. If signal fired, then `sa.sa_handler` (i.e. receive) called

Case Study: UNIX signals

- Is the link between kernel and the process uni-directional or bi-directional?
- Is the communication direct? If so, is the addressing symmetric or asymmetric?
- Is the send blocking (synchronous) or non-blocking (asynchronous)?
- Is the receive blocking (synchronous) or non-blocking (asynchronous)?

End of Chapter 3

- Any questions about processes?
- If you're too embarrassed to ask, remember that likely someone has the same question. There are no "dumb" questions; you are not dumb for not knowing an answer. Each question you ask makes you get more knowledgeable
- Still, if you are too embarrassed, ask on the anonymous forum