CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

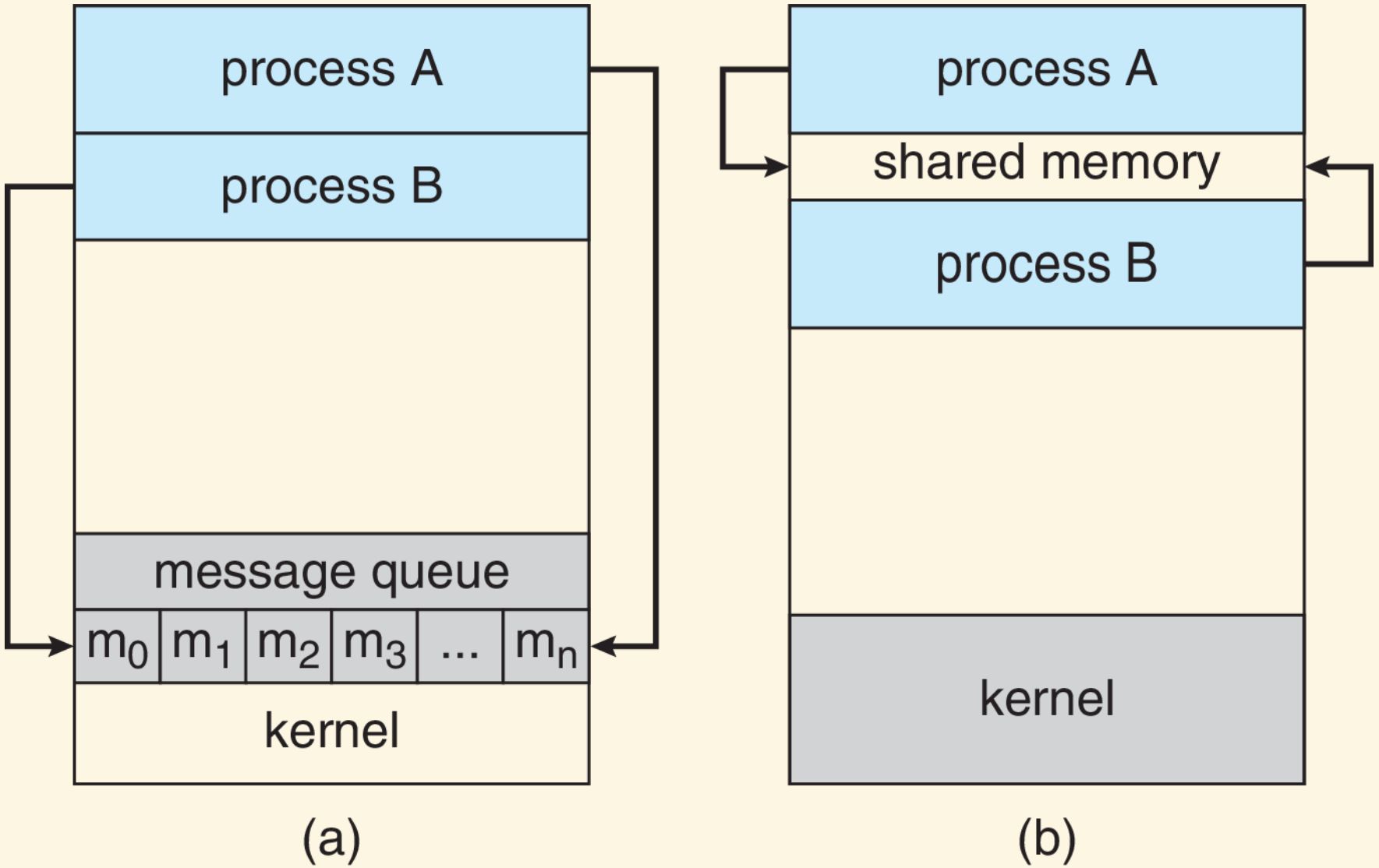
Message Passing

Shared memory

- two or more processes agree to share a region of memory
- OS must specifically allow this exception to the rules
- processes communicate by reading/writing shared memory

Message passing

- process requests that OS sends a message to another process
- OS handles many underlying details


process A
process B
message queue
m_0 m_1 m_2 m_3 ... m_n
kernel
(a)
process A
shared memory
process B
kernel
(b)

How does message passing work?

Fundamentally, it's just that simple -- OS is responsible for passing a message from one process to another

In particular, this ensures that processes do not overwrite one another's messages

Processes need two new functions: `send(msg)` and `receive(msg)`

Lots of different options

Message size can be fixed or variable

Variable-length messages are easy for communicating processes, but harder for OS

Fixed-length messages are the opposite

Direct vs indirect naming

In direct naming, receiving process is hard coded. This reduces flexibility of code

In indirect naming, messages are sent to a *mailbox* (also called a *port*)

Just like connecting to a port on a server -- you send an HTTP request to port 80, not to a particular process

Mailboxes are managed by OS, so system calls must be established to create mailboxes

Allows communication between more than two processes

Two processes can communicate through more than one mailbox for different purposes

Blocking vs non-blocking (*synchronous* vs *asynchronous*)

If a send is...

- blocking: sender waits until receiver has message before moving on
- non-blocking: sender can move on immediately after call to `send`

If a receive is...

- blocking: receiver waits until message is ready
- non-blocking: receiver can move on without receiving message (receives a `NULL` message instead)

Different combinations of blocking/non-blocking sends and receives are possible

Buffering

Related to blocking vs non-blocking -- if sender and receiver are not blocking, data must be stored *somewhere* while awaiting transfer

Zero capacity: OS does not set aside buffer space. Sender must block until recipient receives.

Limited capacity: sender can continue without waiting as long as there is room in the buffer. Once the buffer fills up, all sends are blocking

Unlimited capacity: assume OS has set aside unlimited storage space for buffering. Sender need never block.

Not realistic.

Producer-Consumer

With blocking send and receive, solution to producer-consumer problem becomes trivial.

```
message next_produced;

while (true) {
    /* produce an item in next_produced */

    send(next_produced);
}
```

```
message next_consumed;

while (true) {
    receive(next_consumed);

    /* consume the item in next_consumed */
}
```

This solution is not efficient, however. Why?

Shared Memory vs Message Passing Tradeoffs

Pros of shared memory:

- simple once memory is set up (with caveats)
- fast (with caveats)
- no system calls required after setup
- sending large messages is cheap

Cons of shared memory:

- User is responsible for synchronization
- May be slower on multicore systems due to cache coherency issues

Shared Memory vs Message Passing Tradeoffs

Pros of message passing:

- conceptually simple
- OS handles underlying details

Cons of message passing:

- every `send` or `receive` is a syscall
- may require separate buffers
- expensive to double copy large or frequent messages (copy to mailbox, then copy to receiver)