CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Interprocess Communication

Often, processes on a computer need to *cooperate* to do their jobs

For example, in Linux, tools are often chained like

```
grep b names.txt | sort | uniq
```

As another example, multiple processes can be spawned to deal with a large amount of work in parallel assuming there is more than one processing core on the system

```
make -j 8
```

(Note that `make` may produce *threads* instead of processes, which we will talk about soon)

To do so, the processes need to *communicate*

However, the OS is specifically designed to keep processes separate!

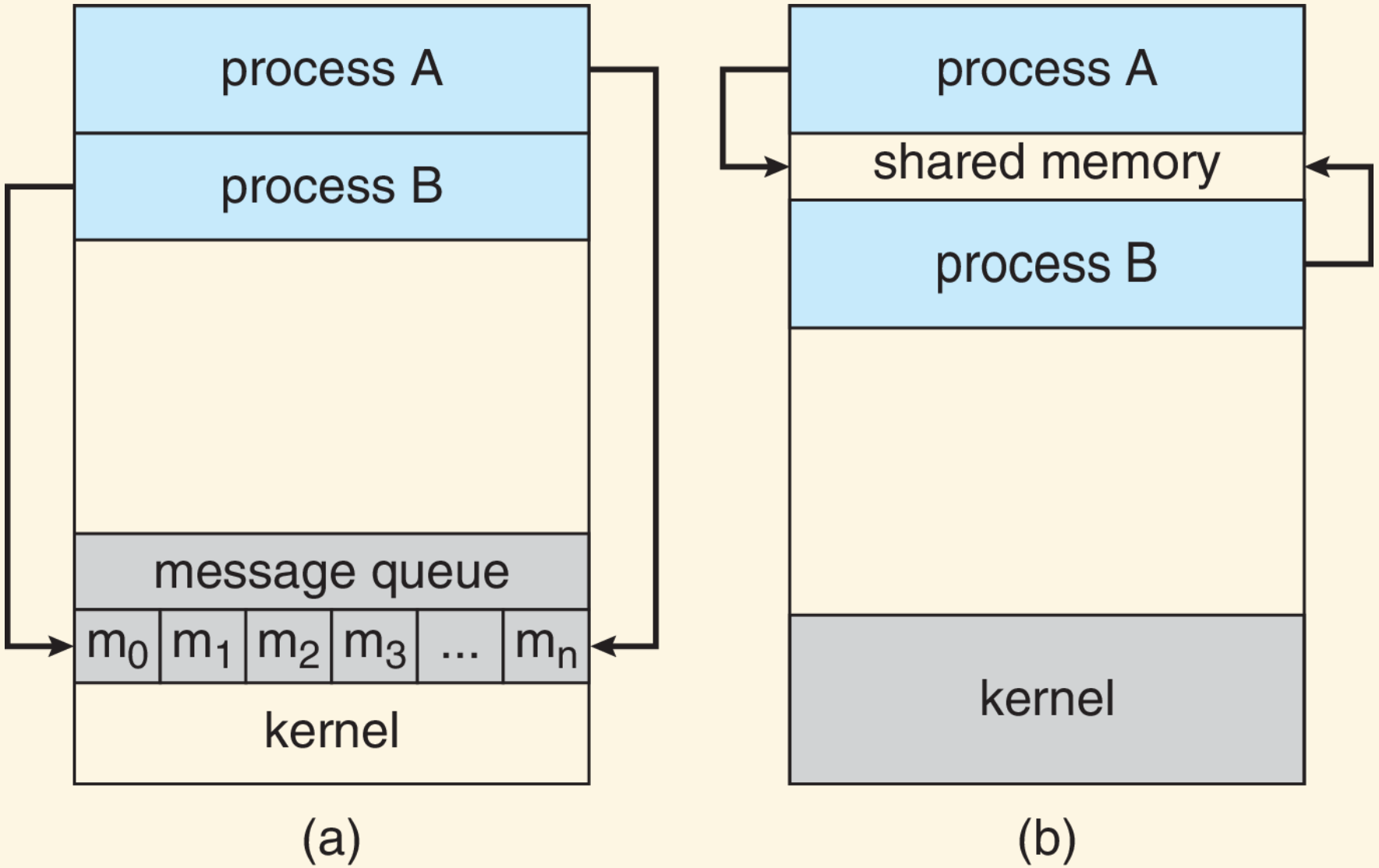
OS can optionally relax some of its rules as needed

Shared memory

- two or more processes agree to share a region of memory
- OS must specifically allow this execption 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 shared memory work?

Two or more processes must want to share memory

Make a system call to declare a section of memory as shared

After shared memory is established, processes are in complete control of what is written (OS does not know or care)

No guarantee from OS that processes won't write to same spot!

```
loc = find_safe_location()                  loc = find_safe_location(
data = compute_write_data()                 data = compute_write_data
write_to_shm(loc, data)                     write_to_shm(loc, data)
// Process A                                // Process B
```

Producer-Consumer Problem

Common way to think about cooperating processes

Two processes, one writing to shared buffer, the other reading

Imagine, for example, one process that fills up a print queue and another that removes jobs and prints them

Set up shared memory region with

```
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

`in` is next free position; `out` is first full position

If `in == out`, buffer is empty.

If `((in + 1) % BUFFER_SIZE) == out`, buffer is full.

Producer process

```
item next_produced;

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

    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumer process

```
item next_consumed;

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

    while (in == out)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```

Warning -- the solution that we use today **won't actually work**

Managing simultaneous processes is hard!

Much more on this soon