Lecture 6: Interprocess Communication

Fall 2019
Jason Tang

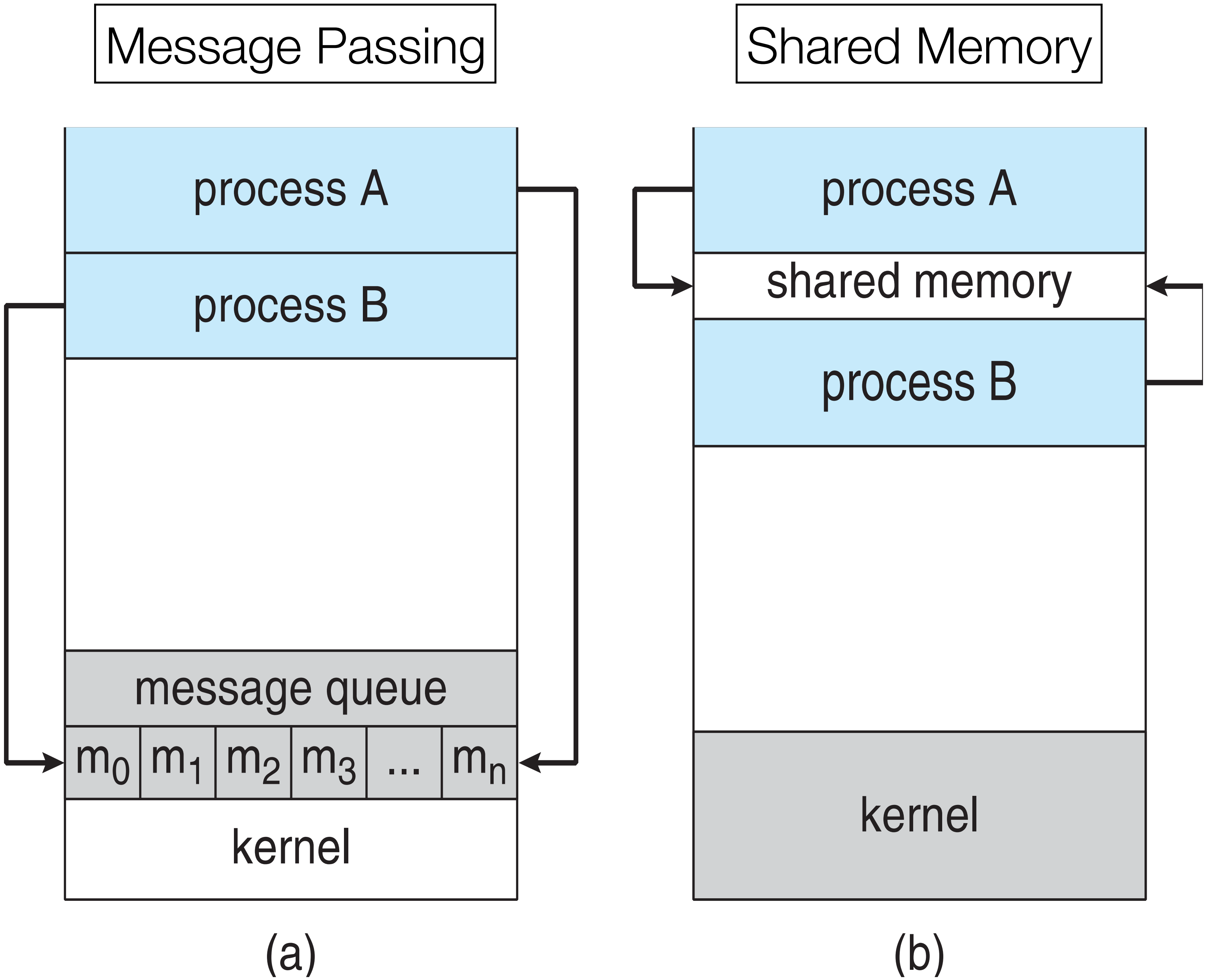
Topics

- Shared Memory
- Message Passing
- Examples of IPC

Interprocess Communication

- Processes may be independent of each other, or cooperating with each other
- Cooperating systems can be affected by other processes:
  - Information sharing
  - Modularity and speedup
  - Convenience
- Cooperating processes need IPC: shared memory and/or message passing

Communication Models



Producer-Consumer

- Classic paradigm for cooperating processes:
  - Producer (often only one process) pushes data to a buffer
  - Consumer (often multiple processes) retrieves data from buffer
- Unbounded buffer: no practical limit on size of buffer
- Bounded buffer: fixed buffer size

Example Bounded Buffer Code

```c
#define BUFFER_SIZE 10
typedef struct {
    …
} Item;
Item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- `in` holds the index to next free buffer element
- `out` holds the index to first used buffer element
- Buffer is empty when `in == out`, and is full when `((in + 1) % BUFFER_SIZE) == out`

Producer and Consumer Code

Producer

```
while (true) {
    Item next_produced = foo();
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* wait for consumer */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
while (true) {
    while (in == out)
        ; /* wait for producer*/
    Item next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    bar(next_consumed);
}
```

- What is the usable capacity of buffer?
- What issues are there with this code?

Shared Memory

- Memory buffer(s) shared between cooperating processes and/or operating system
- Can be used to share lots of data
- Synchronization is significant problem
  - What if two producers write to same buffer location?

Message Passing

- Send data from one entity (process or kernel) to another entity
  - **direct**: one-to-one relationship
  - **broadcast**: one-to-many (or maybe no listeners)
- Main operations are `send`(*message*) and `receive`(*message*)
- *message* is either fixed size or variable length
  - Usually, *message* is somewhat short

Message Passing Implementation

- To pass a message from process **P** to process **Q**, they need to:
  - First establish a communications link
  - Then exchange messages via `send()` / `receive()`
- Usually, links are unidirectional; bi-directional achieved using two links
- Usually, no built-in acknowledgement when message received
  - Software needs to implement a message passing scheme that defines message format, acknowledgement, error handling, etc

Communications Links

- Physical:
  - shared memory
  - hardware bus
  - network
- Logical:
  - direct or indirect
  - blocking or non-blocking
  - automatic or explicit buffering

Direct Communication

- Both sender and receiver must know each other's identity
  - `send`(**Q**, *message*): send *message* to process **Q**
  - `receive`(**P**, *message*): receive *message* from process **P**
- OS establishes link automatically when message sent
- Address symmetry: both **P** and **Q** must name each other to communicate
- Disadvantage is lack of discovery

Indirect Communication

- Messages are directed and received through a common intermediary, known as mailboxes or ports
  - Each mailbox has a unique ID
  - `send`(**A**, *message*): send *message* to mailbox **A**
  - `receive`(**A**, *message*): receive *message* from mailbox **A**
  - New operations: create and destroy mailbox
- Address asymmetry: processes do not have to know each other, just existence of mailbox **A**

Mailbox Sharing

- May have multiple senders and receivers
    - If **P** sends message, and **Q** and **R** receives, who gets message?
- Possible resolutions:
    - Disallow multiple receivers
    - Permit only one `receive()` operation at a time
    - OS chooses who gets message (typically via round-robin)
    - Allow peeking at message

Message Synchronization

- Blocking send: sender blocked until message is received
- Blocking receive: receiver blocked until message is available
- Non-blocking send: sender sends message and continues, does not wait for receiver
- Non-blocking receive: receiver gets an available message, or a special code (often `NULL`) to indicate no messages available
- Sender and receiver do not have to choose same blocking/non-blocking scheme

Buffering

- Queue of messages attached to link
  - Zero capacity: no queue at all; sender must wait for receiver (a so-called rendezvous)
  - Bounded capacity: maximum capacity of **n**
    - If less than **n** messages in queue, a blocking sender will add to queue and continue
    - If **n** messages already in queue, a blocking sender blocks
  - Unbounded capacity: infinite size; sender never blocks

POSIX Signals

- Unidirectional, direct, non-blocking, buffered (**n** = 1) message from one process to another
- Signal is an unsigned integer value
- Used for asynchronous notification
    - A process does not normally wait for a signal to arrive
    - When a process receives a signal, the OS forces a jump to a signal handler to process the signal; when that handler returns, control resumes at prior location (a so-called software interrupt)

Examples of Signals

- If no signal handler is explicitly set, then instead jump to a default handler

Signal Name	Signal Number	Meaning	Default Handler
SIGINT	2	Interrupt from keyboard (Ctrl-C)	Terminate process
SIGKILL	9	Kill signal	Terminate process, cannot be overridden
SIGSEGV	11	Invalid memory reference	Terminate, and generate core file
SIGCHLD	20,17,18	Child stopped or terminated	Ignored

Example of Signal Handling

```c
#define _POSIX_SOURCE
#include <signal.h>
#include <stdio.h>

static void my_fault_handler(int signum) {
    printf("Caught signal number %d\n", signum);
}

int main(void) {
    sigset_t mask;
    sigemptyset(&mask);
    struct sigaction sa = {
        .sa_handler = my_fault_handler,
        .sa_mask = mask,
        .sa_flags = 0
    };
    sigaction(SIGSEGV, &sa, NULL);
    raise(SIGSEGV);
    return 0;
}
```

Remote Procedure Calls

- Client-server design: one producer and multiple consumers
  - Example: HTTP daemon and multiple web browsers
- One use of client-server is to implement RPC
  - Client connects to server
  - Client sends name of procedure to invoke, and its parameters
  - Daemon does work
  - Daemon sends back results

RPC Implementation

- Stub: client-side proxy representing procedure
- When RPC invoked, stub locates server and marshals parameters
    - Data reformatted to a common format, such as External Data Representation (XDR): big-endian, 32-bit words, strings padded to 4 bytes
- RPC daemon unmarshalls data into its native format and performs work
- OS typically has mechanism to advertise RPC services (the matchmaker)

Execution of RPC