Interprocess Communication

Courtesy: Slides are adapted from the textbook “Operating System Concepts” by Silberschatz et al

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- A process is ***independent*** if it **cannot affect or be affected** by the other processes executing in the system.
  - Any process that does not share data with any other process is independent.
- A process is ***cooperating*** if it **can affect or be affected** by the other processes executing in the system.
  - Any process that shares data with other processes is a cooperating process.
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

Cooperating Processes

- Advantages of process cooperation
  - Information sharing
    - Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
  - Computation speed-up
    - If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores
  - Modularity
    - We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads
  - Convenience
    - Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel

Interprocess Communication

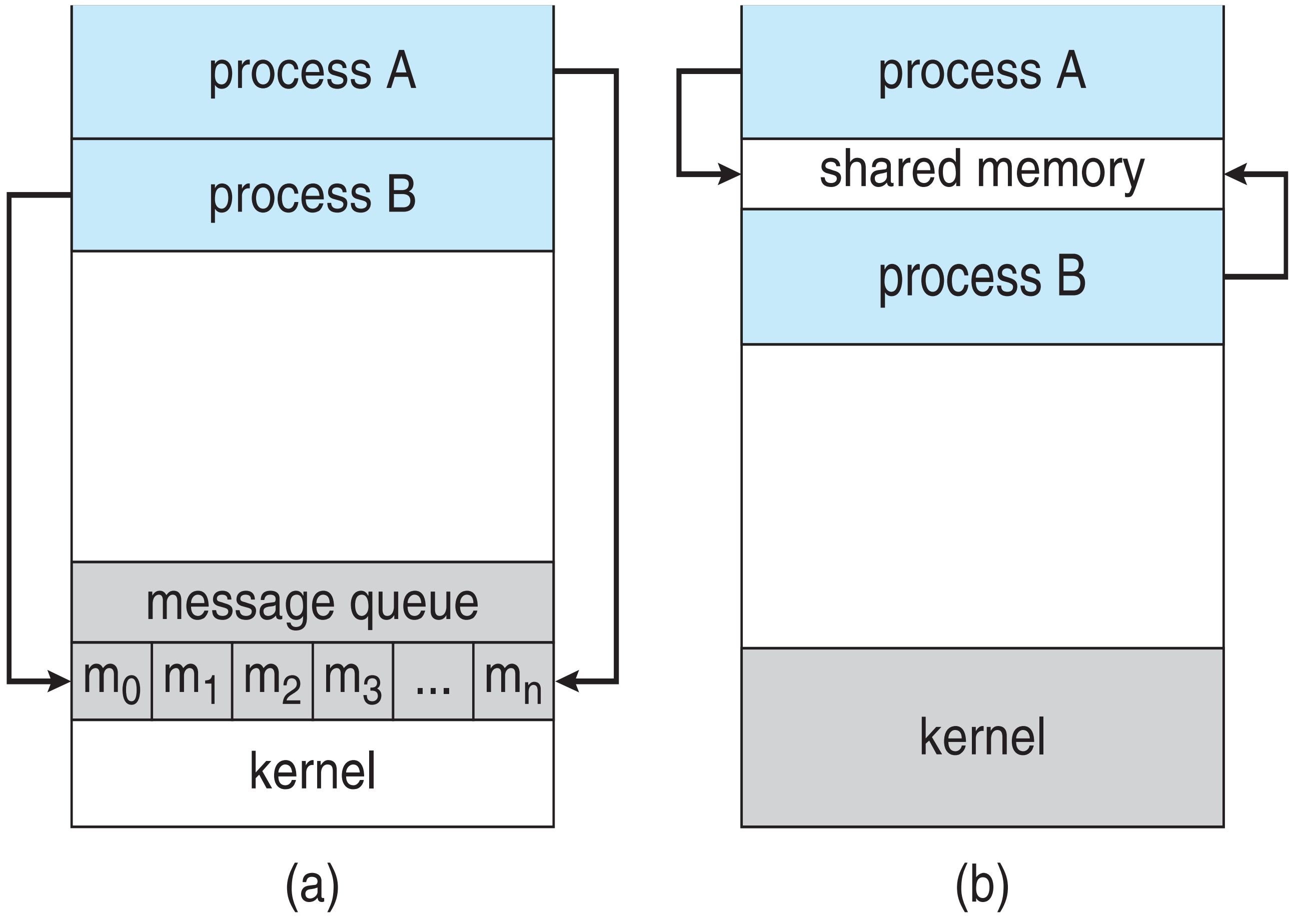
- Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them **to exchange data and information**
- Two models of IPC
  - **Shared memory**
    - a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region
  - **Message passing**
    - Communication takes place by means of messages exchanged between the cooperating processes
    - Message passing is useful for exchanging smaller amounts of data, because **no conflicts need be avoided**.
    - Message passing is also easier to implement in a distributed system than shared memory.

Interprocess Communication

- Shared memory can be **faster** than message passing, since message-passing systems are typically implemented using **system calls** thus require the **more time-consuming task of kernel intervention**.
- In shared-memory systems, system calls are required **only to establish shared memory regions**. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.
- Recent research on systems with **several processing cores** indicates that **message passing provides better performance** than shared memory on such systems. **Shared memory suffers from cache coherency issues**, which arise because shared data migrate among the several caches. As the number of processing cores on systems increases, it is possible that we will see message passing as the preferred mechanism for IPC.

Communications Models

(a) Message passing. (b) shared memory.



Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.
  - The producer–consumer problem also provide metaphor for the client-server paradigm. Ex. a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.
  - **Unbounded-buffer** places no practical limit on the size of the buffer
    - The consumer may have to wait for new items, but the producer can always produce new items
  - **Bounded-buffer** assumes that there is a fixed buffer size
    - the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

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

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
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;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
  - processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable
  - If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult
  - Conversely, variable-sized messages require a more complex system level implementation, but the programming task becomes simpler.

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network (Chapter 17)
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - 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
- This scheme exhibits ***symmetry*** in addressing; that is, both the sender process and the receiver process must name the other to communicate.
- In ***asymmetry*** addressing, only the sender names the recipient; the recipient is not required to name the sender
  - send(P, message)—Send a message to process P.
  - receive(id, message)—Receive a message from any process.

Indirect Communication

- I***ndirect communication:*** the messages are sent to and received from ***mailboxes***, or ***ports***
  - A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
    - Each mailbox has a unique identification.
    - A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  `send`(*A, message*) – send a message to mailbox A

  `receive`(*A, message*) – receive a message from mailbox A

Indirect Communication

- Properties of communication link
  - Link established only if processes share a common 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

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations of send() and receive() are possible
  - If both send and receive are blocking, we have a **rendezvous**

Synchronization (Cont.)

- Producer-consumer 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 */
}
```

Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue
  - Queue of messages attached to the link.
- implemented in one of three ways
  - **Zero capacity** – The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message (rendezvous)
  - **Bounded capacity** – finite length of *n* messages. the sender can continue execution without waiting. Sender must wait if link full
  - **Unbounded capacity** – infinite length. Sender never waits
- The zero-capacity case is sometimes referred to as a message system with no buffering.
- The other cases are referred to as systems with automatic buffering