Operating Systems
CS F372

Synchronization

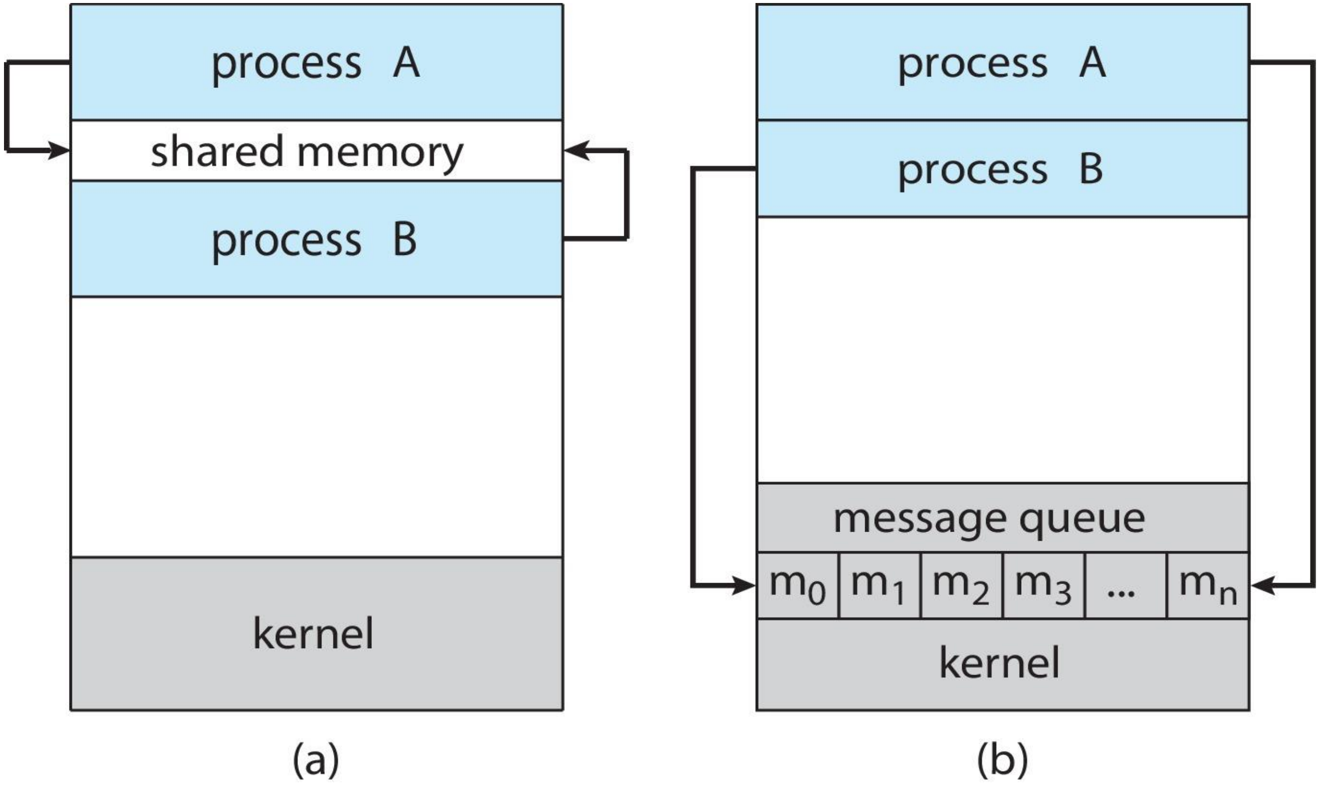
BIJU K RAVEENDRAN

Inter Process Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing, Computation speedup, Modularity, Convenience
- Cooperating processes need **inter process communication (IPC)**
  - Shared memory, Message passing

Communication Models

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



Message Passing

- Message system – processes communicate with each other without resorting to shared variables
  - Very useful in Distributed systems
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed/variable]
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive

Communication

- Direct
  - Processes must name each other explicitly:
  - **send** (*P, message*), **receive**(*Q, message*)
  - [exactly one] link [unidirectional or bi-directional] is associated with exactly one pair of processes
- Indirect
  - Mailbox [also referred to as port] with unique ID
    - Processes can communicate only if they share a mailbox
    - A link [uni /bi directional] may be associated with many processes
    - Each pair of processes may share several communication links, each link corresponds to a mail box
    - Operations: create new mailbox, send/receive messages, destroy mailbox

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send:** The sender blocks until the message is received
  - **Blocking receive:** The receiver blocks until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking:** The sender sends the message and continue
  - **Non-blocking:** The receiver receives a valid message or null

Synchronization

- If a message is lost or if process fails before sending , receive process is permanently blocked in case of blocking receive
- In case of non blocking receive, if the process executes receive before message is sent the message will be lost
- Allow process to test weather a message is waiting before issuing a receive primitive.
- Receive can also test for arrival before issuing receive.

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Producer – Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Producer – Consumer Problem

in ← 0, out ← 0

PRODUCER

```
while (1) {
    // Produce item;
    Buffer[in] = item;
    in = in + 1;
}
```

CONSUMER

```
while (1) {
    while (in == out);
    item = Buffer[out];
    out = out +1;
}
```

Bounded Buffer – Shared Memory Solution

- Shared data

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

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

Producer – Consumer Problem

```
/* Producer */
while (true) {
     /* Produce an item */
   while (((in+1)% BUFFER_SIZE)  == out);
                                        /* do nothing -- no free buffers */
     buffer[in] = item;
     in = (in + 1) % BUFFER_SIZE;
}
/*Consumer */
while (true) {
    while (in == out); // do nothing -- nothing to consume
     // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```

```
/* Producer */
while (true) {
   /*  produce an item and put in nextProduced  */
    while (count == BUFFER_SIZE); // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

```
/* Consumer  */
while (true)  {
         while (count == 0); // do nothing
         nextConsumed =  buffer[out];
         out = (out + 1) % BUFFER_SIZE;
         count--;
         /*  consume the item in nextConsumed */
}
```