Process Synchronization (1)

Dr. Jun Zheng

CSE325 Principles of Operating Systems

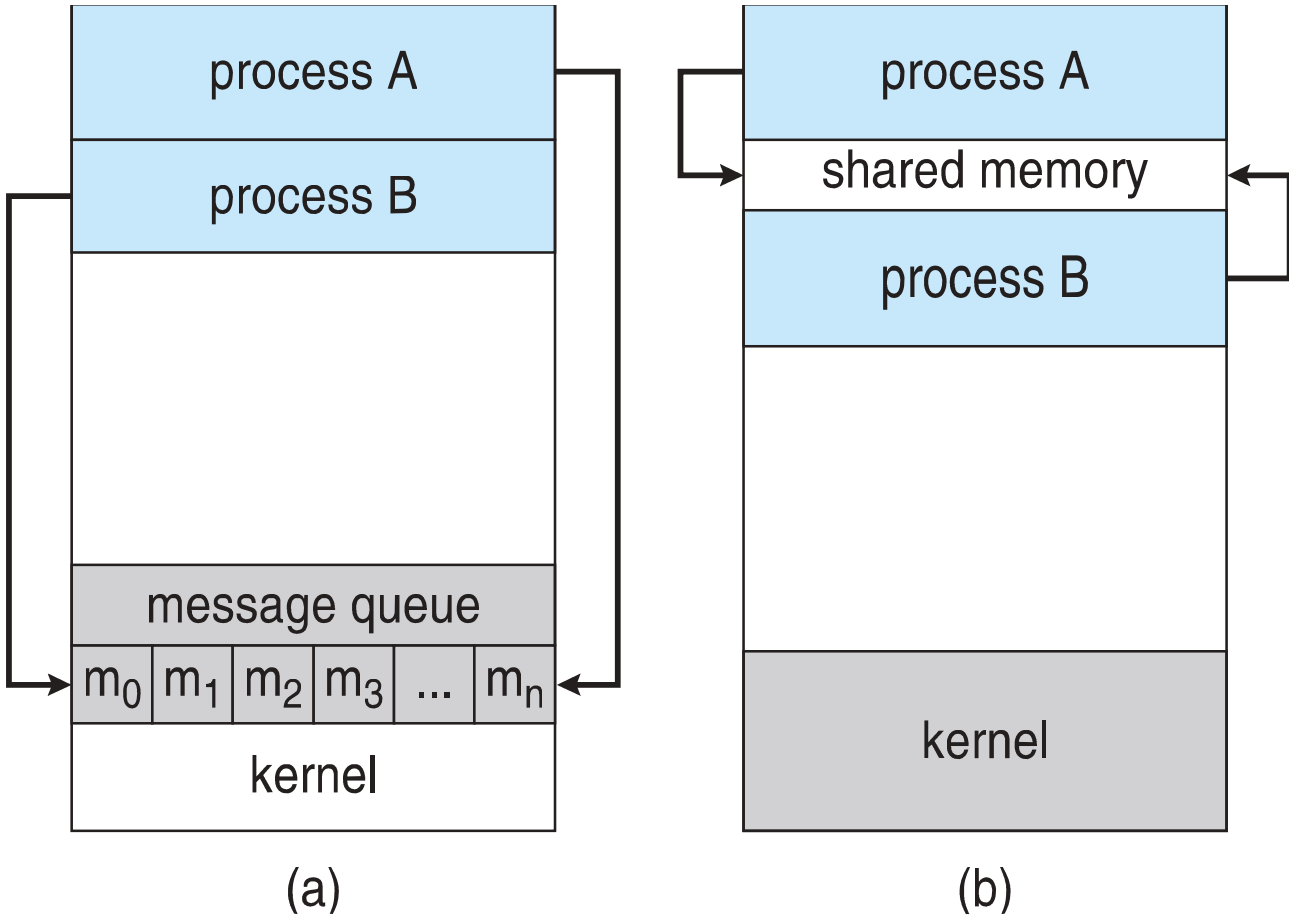
9/25/2019

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Cooperating processes need **interprocess communication** (**IPC**)
  - Message passing
  - Shared memory

Two Communications Models

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



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.
  - Concurrent access to shared data may result in data inconsistency

Producer-Consumer Problem

- A common paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **Shared memory (a buffer of items filled by the producer and emptied by the consumer)**
    - **unbounded-buffer** places no practical limit on the size of the buffer
    - **bounded-buffer** assumes that there is a fixed buffer size
  - Producer and consumer must be synchronized

Bounded-Buffer – Shared-Memory Solution

Shared data

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

item buffer[BUFFER_SIZE];
int in = 0;  // next free position
int out = 0; // first full position
int counter = 0;
```

Empty: in == out, Full: ((in+1)%BUFFLE_SIZE) == out

NEW MEXICO TECH
SCIENCE · ENGINEERING · RESEARCH · UNIVERSITY

Producer

```
while (true) {
   /* produce an item in next produced */

   while (counter == BUFFER_SIZE) ;
      /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Consumer

```
while (true) {
   while (counter == 0)
      ; /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
         counter--;
   /* consume the item in next consumed */
}
```

- Does not consider the producer process and the consumer process attempt to access to the shared buffer concurrently

Race Condition

`counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

`counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}