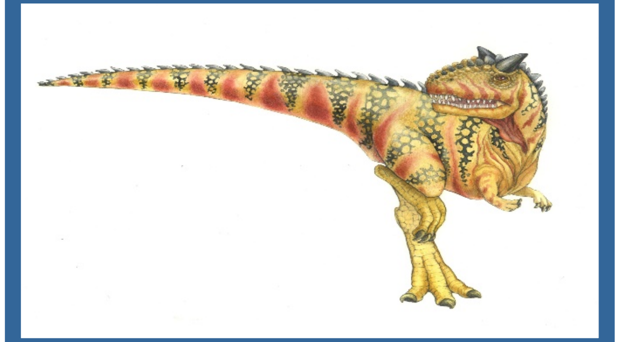
Chapter 5: Process Synchronization

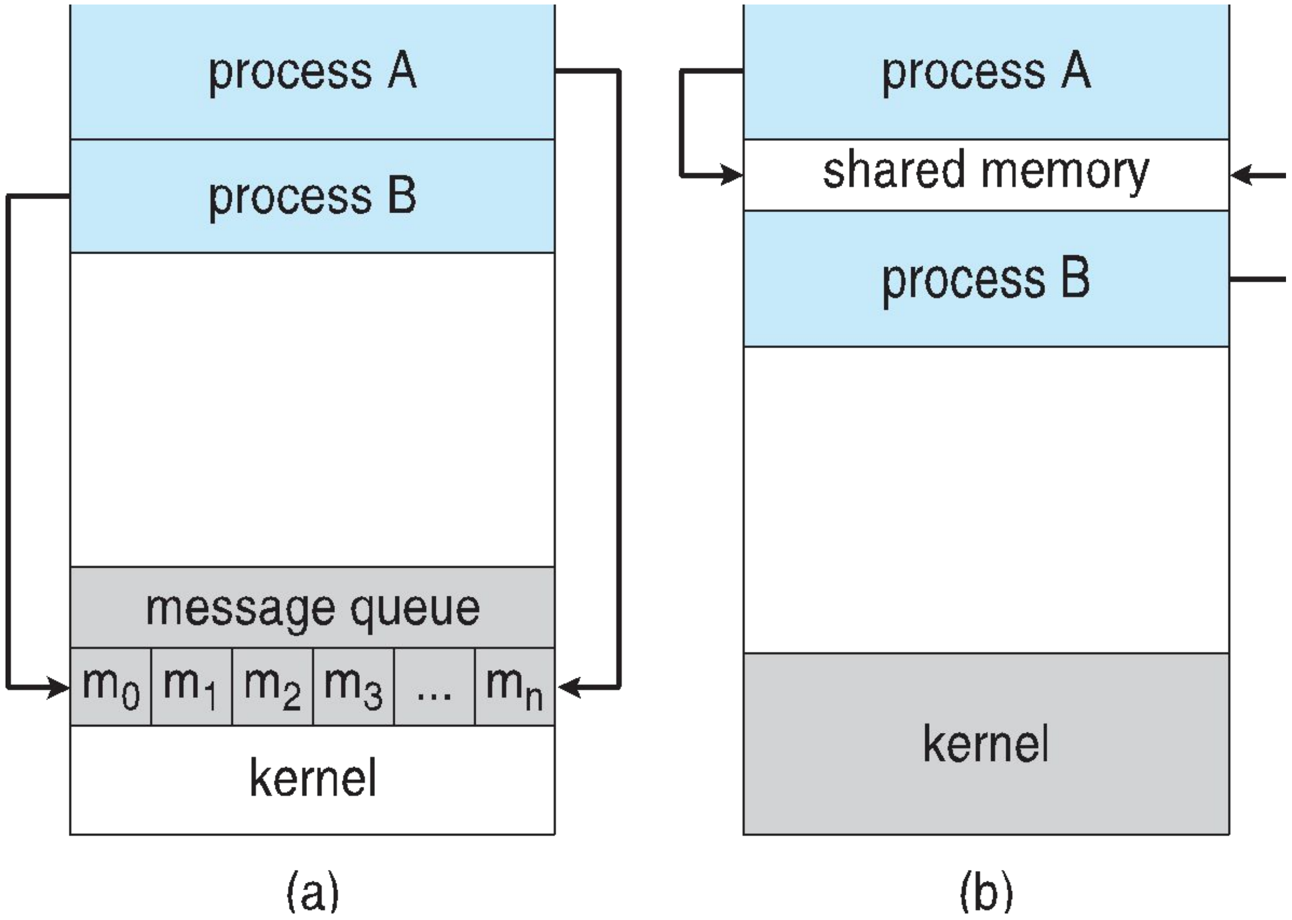


Interprocess Communication

- 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 **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

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.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – 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

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

```
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 */
}
```

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
```

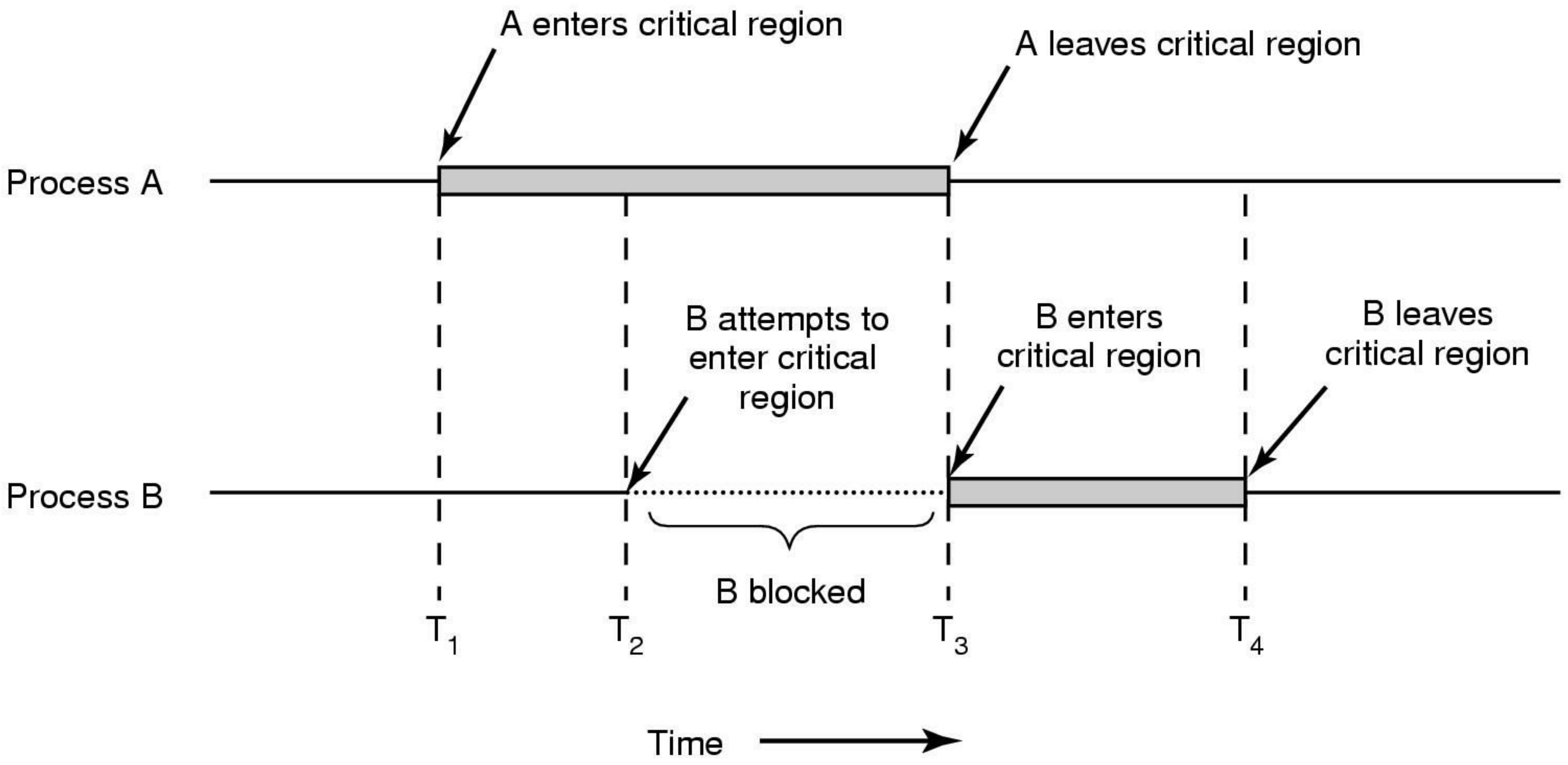
- Consider this execution interleaving with "count = 5" initially:

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}

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section



Critical Section

- General structure of process P_i

```
do {
     entry section
          critical section
     exit section
          remainder section
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the n processes

Synchronization Hardware

Possible solutions:

- Disabling Interrupts
- Lock Variables
- Strict Alternation
- Test and Set Lock
- Peterson's solution

Disabling Interrupts

- How does it work?
  - Disable all interrupts just after entering a critical section
  - Re-enable them just before leaving it.
- Why does it work?
  - With interrupts disabled, no clock interrupts can occur
  - No switching can occur
- Problems:
  - What if the process forgets to enable the interrupts?
  - Multiprocessor? (disabling interrupts only affects one CPU)

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

Lock Variable Method

```
int lock = 0;
while (lock !=0);
lock = 1;
//EnterCriticalSection;
   access shared variable;
//LeaveCriticalSection;
lock = 0;
```

Strict Alternation

initially turn=0

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Problems

- Busy waiting: Continuously testing a variable until some value appear
  - Wastes CPU time
- Violates progress
  - When one process is much slower than the other

Peterson's solution

- Consists of 2 procedures
- Each process has to call
  - `enter_region` with its own process # before entering its C.R.
  - And `Leave_region` after leaving C.R.

```
do {

    entry section

        critical section

    exitsection

        remainder section

} while  (TRUE);
```

Peterson's solution (for 2 processes)

```
#define FALSE  0
#define TRUE   1
#define N      2                         /* number of processes */

int turn;                                /* whose turn is it? */
int interested[N];                       /* all values initially 0 (FALSE) */

void enter_region(int process);          /* process is 0 or 1 */
{
      int other;                         /* number of the other process */

      other = 1 − process;               /* the opposite of process */
      interested[process] = TRUE;        /* show that you are interested */
      turn = process;                    /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)           /* process: who is leaving */
{
      interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Peterson's Solution: Analysis(1)

- Let Process 1 is not interested and Process 0 calls enter_region with 0
- So, turn = 0 and interested[0] = true and Process 0 is in CR
- Now if Process 1 calls enter_region, it will hang there until interested[0] is false. Which only happens when Process 0 calls leave_region i.e. leaves the C.R.

Peterson's Solution: Analysis(2)

- Let both processes call enter_region simultaneously
- Say turn = 1. (i.e. Process 1 stores last)
- Process 0 enters critical region: while (turn = = 0 && …) returns false since turn = 1.
- Process 1 loops until process 0 exits: while (turn = = 1 && interested[0] = = true) returns true.

Sleep & wakeup

- When a process has to wait, change its state to **BLOCKED/WAITING**
- Switched to **READY** state, when it is OK to retry entering the critical section
- Sleep is a system call that causes the caller to block
  - be suspended until another process wakes it up
- Wakeup system call has one parameter, the process to be awakened.

Producer Consumer Problem

- Also called bounded-buffer problem
- Two (*m*+*n*) processes share a common buffer
- One (*m*) of them is (are) **producer**(s): put(s) information in the buffer
- One (*n*) of them is (are) **consumer**(s): take(s) information out of the buffer
- Trouble and solution
  - Producer wants to put but buffer full- Go to **sleep** and **wake up** when consumer takes one or more
  - Consumer wants to take but buffer empty- go to sleep and wake up when producer puts one or more

Sleep and Wakeup

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Sleep and Wakeup: Race condition

- **Race condition**
- Unconstrained access to *count*
  - CPU is given to P just after C has read count to be 0 but not yet gone to sleep.
  - P calls wakeup
  - Result is lost wake-up signal
  - Both will sleep forever

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation (Down operation)`**

```
void wait(sem_t *S)
{
S->Value = S->Value - 1;
if (S->value < 0)
block on semaphore and put the process in suspended list
Sleep()
}
```

Semaphore

- Definition of the **`signal()` operation (Up operation)**

```
void signal(sem_t *S)
{
S->value = S->value + 1;
if (S->value <= 0)
unblock one process or thread that is blocked on semaphore
wakeup()
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**`synch`**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore S as a binary semaphore

Semaphore

- Operation “down”:
  - if value > 0; value-- and then continue.
  - if value = 0; process is put to sleep without completing the down for the moment
    - Checking the value, changing it, and possibly going to sleep, is all done as an ***atomic*** *action*.
- Operation “up”:
  - increments the value of the semaphore addressed.
  - If one or more process were sleeping on that semaphore, one of them is chosen by the system (e.g. at random) and is allowed to complete its *down*
    - The operation of incrementing the semaphore and waking up one process is also indivisible
  - No process ever blocks doing an *up.*

Semaphore

- Operations must be atomic
  - Two *down*'s together can't decrement value below zero
  - Similarly, process going to sleep in *down* won't miss wakeup from *up* – even if they both happen at same time

Producer & consumer

```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */
```

```
void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                down(&empty);
                down(&mutex);
                insert_item(item);
                up(&mutex);
                up(&full);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                down(&full);
                down(&mutex);
                item = remove_item( );
                up(&mutex);
                up(&empty);
                consume_item(item);
        }
}
```

Semaphores in Producer Consumer Problem: Analysis

- 3 semaphores are used
  - *full* (initially 0) for counting occupied slots
  - *Empty* (initially *N*) for counting empty slots
  - *mutex* (initially 1) to make sure that Producer and Consumer do not access the buffer at the same time
- Here 2 uses of semaphores
  - Mutual exclusion (mutex)
  - Synchronization (full and empty)
    - To guarantee that certain event sequences do or do not occur

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed