OPERATING SYSTEM

Process Management

Chapter 2

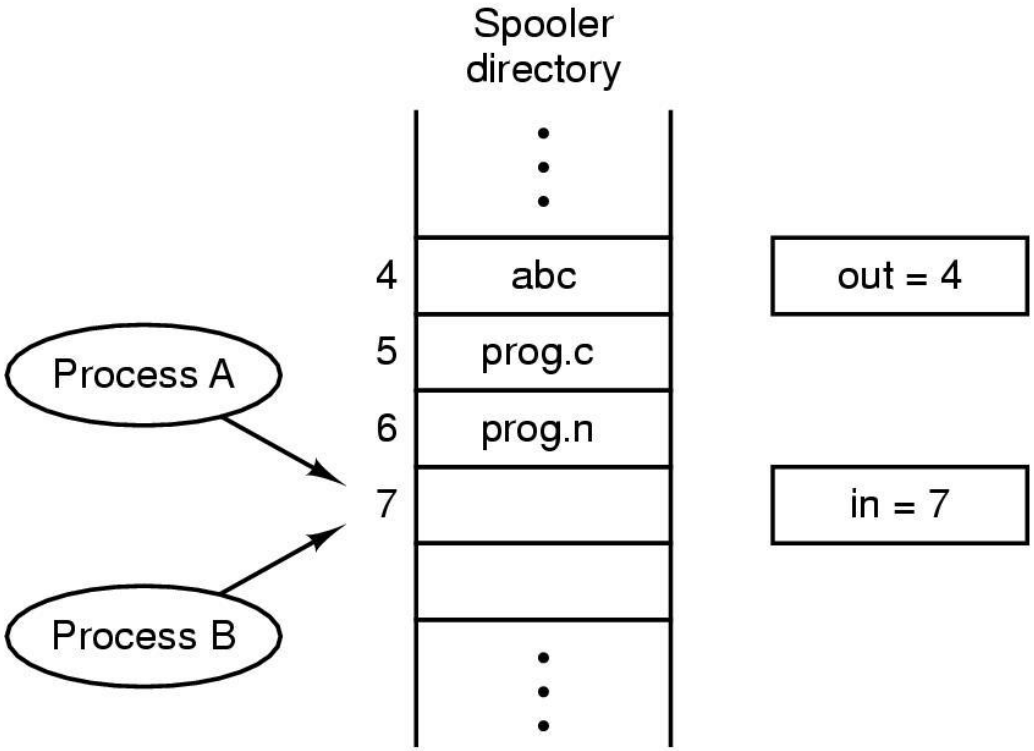
Processes and Threads

2.4 Interprocess communication

Interprocess Communication

- Processes often need to communicate with other processes
- Example: shell pipeline
- Now we'll look at some issues related to IPC inshaAllah.

Race Conditions



Two processes want to access shared memory at same time

How to avoid race condition?

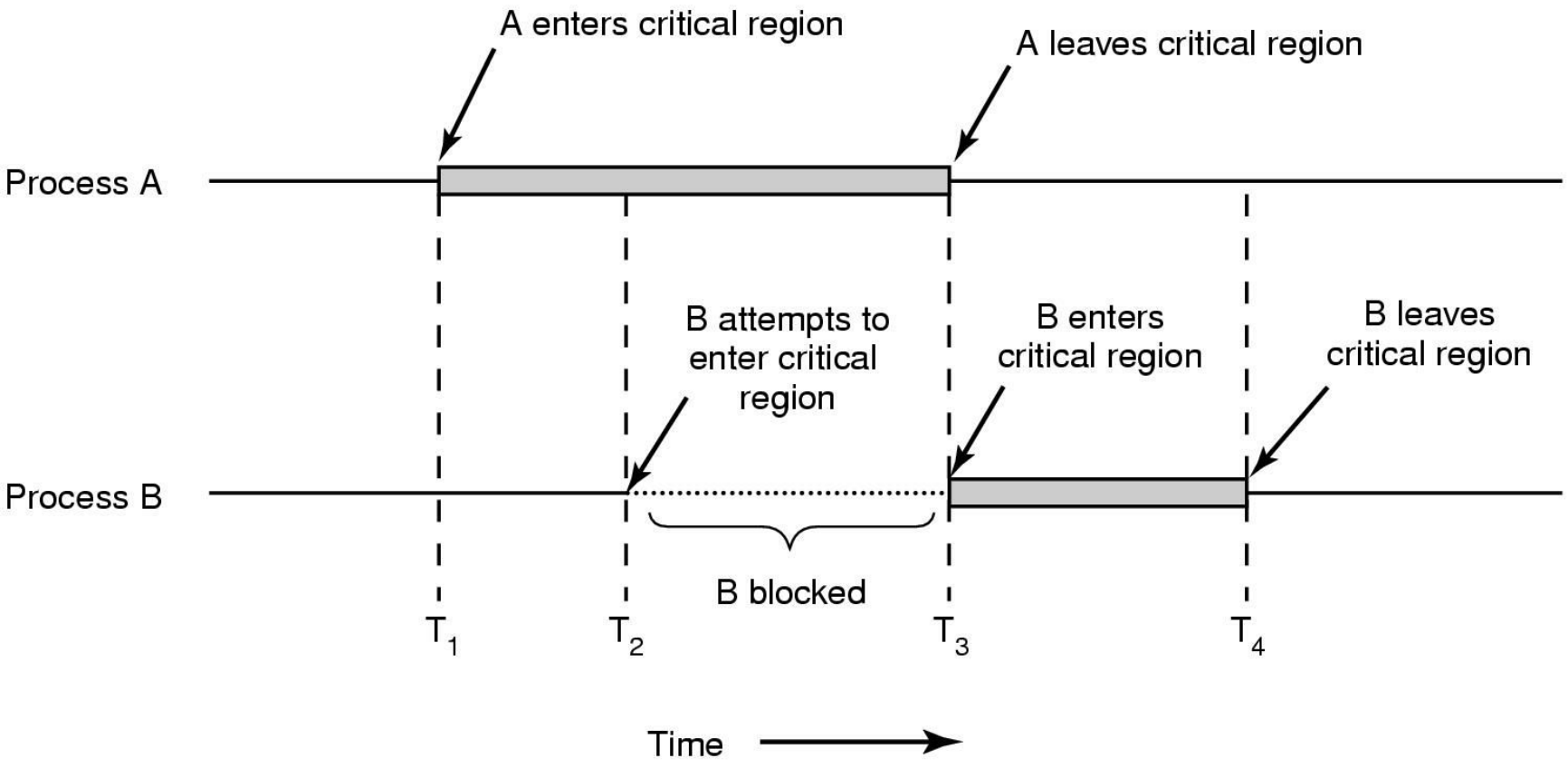
- Mutual Exclusion:
  - No 2 processes use a shared data at the same time
  - If one process is using a shared data , the other process will be excluded from doing the same thing
- Critical Region:
  - That part of the program that do critical things such as accessing shared memory
  - Can lead to race condition

Solution

Four conditions for a good solution

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

Solution



Mutual exclusion using critical regions

Mutual Exclusion with Busy Waiting (solution 1)

```
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)

Proposed solution to critical region problem

(a) Process 0. (b) Process 1.

Problem

- Busy waiting: Continuously testing a variable until some value appear
  - Wastes CPU time
  - Strict Alternation of two process
- Violates condition 3
  - When one process is much slower than the other

Problem

Scenario of Violation 3 (Step-by-Step)

Violation: P0 is **blocked by P1**, which is **not in its critical region.**

Reason: P1 is much slower than P0

Step	Process 0	Process 1	Value of `turn`	Situation
1	P0 enters CR	P1 waiting (turn=0)	0	OK
2	P0 leaves CR, sets `turn=1`	-	1	P1 can enter now
3	P0 Waiting	P1 quickly enters CR, then exits and sets `turn=0`	0	OK
4	P0 leaves CR, sets `turn=1`	P1 still in non-CR	turn = 1	
5	P0 finish non-CR fast → wants to enter CR again	P1 still in non-CR	turn = 1	P0 blocked in `while(turn != 0)` even though P1 is not in CR

Mutual Exclusion with Busy Waiting (solution 2)

- Dekker's Algorithm: Dutch mathematician T. Dekker first developed a *software-based solution* to the mutual exclusion problem that did not require strict alternation, combining ideas of *turn-taking* and *lock/warning variables*.
- Peterson's Simplification: In 1981, G. L. Peterson introduced a simpler and more elegant algorithm for achieving mutual exclusion, effectively replacing Dekker's earlier solution.
- Mechanism Overview: In Peterson's algorithm, each process calls `enter_region()` before entering its critical section and `leave_region()` after finishing, ensuring safe and cooperative access to shared resources.

Mutual Exclusion with Busy Waiting (solution 2)

```
#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 for achieving mutual exclusion

Mutual Exclusion with Busy Waiting
Peterson's Solution: Analysis(1)

Step	Action / Situation	Explanation / Result
1	Both processes are outside their critical regions.	`interested[0] = FALSE`, `interested[1] = FALSE`. No one is using shared data yet.
2	Process 0 calls `enter_region(0)`.	Sets `interested[0] = TRUE` and `turn = 0`. Since P1 is not interested, P0 enters its critical region immediately.
3	Process 1 tries to enter later.	P1 sees `interested[0] = TRUE`, so it waits until P0 finishes.
4	Both processes call `enter_region()` almost at the same time.	Both set `interested[] = TRUE` and each sets `turn` to its own number. The process that writes to `turn` last decides the value.
5	Suppose turn = 1 (P1 wrote last).	P0's condition (`turn == 0`) is false, so P0 enters critical region. P1 keeps waiting because it's P0's turn.
6	P0 exits critical region.	P0 sets `interested[0] = FALSE`. Now P1's waiting condition becomes false.
7	P1 enters its critical region.	Safe alteration occurs; mutual exclusion is preserved.

Mutual Exclusion with Busy Waiting
Peterson's Solution: Analysis(2)

- 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 CR

Mutual Exclusion with Busy Waiting Peterson's Solution: Analysis(3)

- 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 ex: while (turn = = 1 && interested[0] = = true) returns true.
- Done!!

Mutual Exclusion with Busy Waiting: TSL

- Requires hardware support
- TSL instruction: test and set lock
  - Reads content of *lock* into a Register
  - Stores a nonzero value at *lock*.

Indivisible/Atomic

- CPU executing TSL locks the memory prohibiting other CPUs

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered


leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET | return to caller
```

Busy Waiting: Problems

- Waste CPU time since it sits on a tight loop
- May have unexpected effects (for priority scheduling algorithm):
  - Priority Inversion Problem
    - a high priority waits for a low priority to leave the critical section
    - the low priority can never execute since the high priority is not blocked.

Example:

Process H and L with **scheduling rule** that L is never scheduled while H is scheduled. Let L in CR and H is ready and wants to enter CR. Since H is ready it is given the CPU and it starts busy waiting. L will never get to run

Solution: Sleep and Wake-up

Sleep and Wake-up

Blocking vs. Busy Waiting: Some interprocess communication (IPC) methods use blocking (not busy waiting), meaning a process stops executing instead of wasting CPU time while waiting.

Sleep and Wakeup Primitives: sleep is a system call that makes a process block (suspend execution) until another process wakes it up.

wakeup is used to resume (wake up) a specific blocked process.

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 */
     }
}
```

Problems are discussed in the next slide

Producer-consumer problem

Sleep and Wakeup: 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.
  - Result is lost wake-up signal
  - Both will sleep forever

Lost Wakeup Signal

- Process A wants to wakeup B, who is not in sleep yet.
- Quick fix: wakeup waiting bit
  - When A calls wakeup(B) and B is not in sleep, set the bit
  - When B wants to sleep and the bit is set, B will not sleep
- Problem- When more than 2 processes? We need more bits. So the problem remains in principle.

Semaphores(1)

- A new variable type: an integer variable to count the number of wakeups saved for future use.
- Value = 0 => No wakeups are saved
- Value > 0 => One or more wakeups are saved
- Two operations: Down and UP(similar to sleep and wakeup)
- Operation "down": if value > 0 value-- (uses up one stored wakeup) 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*. So once a semaphore operation has started, no other process can access it until the operation has completed or blocked.

Semaphores(2)

- 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*.

Semaphores(3)

```
#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) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

The producer-consumer problem using semaphores

Semaphores in Producer Consumer Problem: Analysis

`Initial values: empty = 3, full = 0, mutex = 1`

Step	Producer Action	Consumer Action	empty	full	mutex
1	P0 produces item	-	3	0	1
2	down(empty) → empty=2	-	2	0	1
3	down(mutex) → enters critical section	-	2	0	0
4	Inserts item, up(mutex)	-	2	0	1
5	up(full) → full=1	-	2	1	1
6	-	C0 calls down(full) → full=0	2	0	1
7	-	down(mutex) → enters critical section	2	0	0
8	-	Removes item, up(mutex)	2	0	1
9	-	up(empty) → empty=3	3	0	1

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 P and C donot access the buffer at the same time.
- Binary Semaphores: That are initialized to one and used by 2 or more processes to ensure that only one of them enter CR at the same time
- If each process does a down just before entering CR and an up just after leaving then mutual exclusion is guaranteed
- Here 2 uses of semaphores
  - Mutual exclusion (mutex)
  - Synchronization (full and empty): to guarantee that certain event sequences do or do not occur

Mutexes

- Simplified version of semaphore
- A *mutex* is a variable that can be in 1 of 2 states: locked and unlocked.
- Two procedures: mutex_lock and mutex_unlock
- Similar like enter_region with a crucial difference

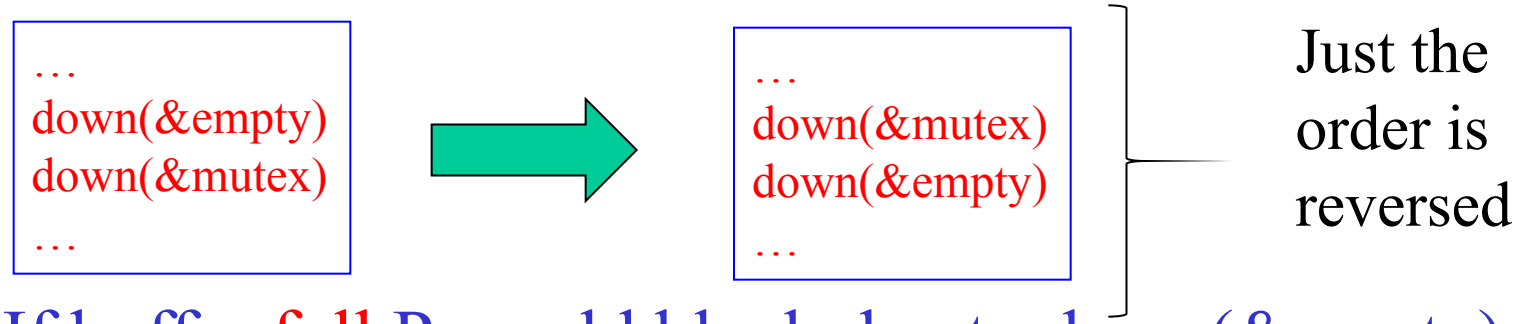
```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again later
ok:     RET | return to caller; critical region entered


mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET | return to caller
```

Implementation of *mutex_lock* and *mutex_unlock*

Semaphores: "Be Careful"

- Suppose the following is done in P's code



- If buffer full P would block due to down(&empty) with mutex = 0.
- So now if C tries to access the buffer, it would block too due to its down(&mutex).
- Both processes would stay blocked forever: DEADLOCK

Revisiting semaphores!

- Semaphores are very "low-level" primitives
  - Users could easily make small errors
  - Very **difficult to debug**
- Also, we seem to be using them in two ways
  - For mutual exclusion, the "real" abstraction is a **critical section**
  - But for Synchronization threads "**communicate**" using semaphores
- Simplification: Provide concurrency support in compiler
  - Monitors

Monitors (1)

- A higher level synchronization primitive
- A collection of procedures, variables and data structures grouped in a special kind of module or package.

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Example of a monitor

Monitors(2)

- Only one process can be active in a monitor at any instant
- Monitors are programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other calls.
- Because the compiler, not the programmer, is arranging the mutual exclusion it is **safer**
- We also need a way to block and wakeup: Wait and Signal (done on a condition variables)

Monitors(3)

- *wait* is called on some condition variables:
  - Calling process is blocked and another process that had been previously prohibited from entering the monitor is allowed to enter now.
- *signal* is called on some condition variable:
  - A process waiting on *that CV* is given the chance to get up.
  - Who should run? Caller or awakened one?

  Alternative#1: Let newly awakened process to run suspending the caller.

  Alternative#2: Process doing a signal must exit the monitor immediately i.e. signal statement may appear only as the final statement in a monitor procedure.

  Alternative#3: Let the caller run and when it exits the monitor then the waiting process is allowed to start.

  Note: If more than one processes are waiting on *full*, one of them is scheduled.

Monitors (4)

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count – 1;
                if count = N – 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

- Outline of producer-consumer problem with monitors
  - only one monitor procedure active at one time
  - buffer has *N* slots

- **Problems with monitors and semaphores**
  - **Semaphores are too low level**
  - **Monitors are not usable except in a few programming languages**
  - **Designed to work in an environment having access to a common memory**
  - **Doesn't allow information exchange among machines**
  - **None of them would work in a distributed systems (why?) consisted of multiple CPUs, each with its own private memory.**

Message Passing

- solution to the problem of semaphores and monitors w.r.t distributed systems
- A method of IPC that uses two primitives
  - send and receive: system calls.
  - send(destination, &message)
  - receive(source, &message)
  - If no message is available:
    - The receiver can block until one arrives
    - Return immediately with an error code

Message Passing

- Challenges:
  - Messages can be lost by the network.
  - Acknowledgement and retransmission issue.
  - Process naming.
  - Authentication.
  - Performance issue.

Producer Consumer with Message Passing

Assumptions:

- All messages are of same size
- Messages sent but not yet received are automatically buffered

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

The producer-consumer problem with N messages

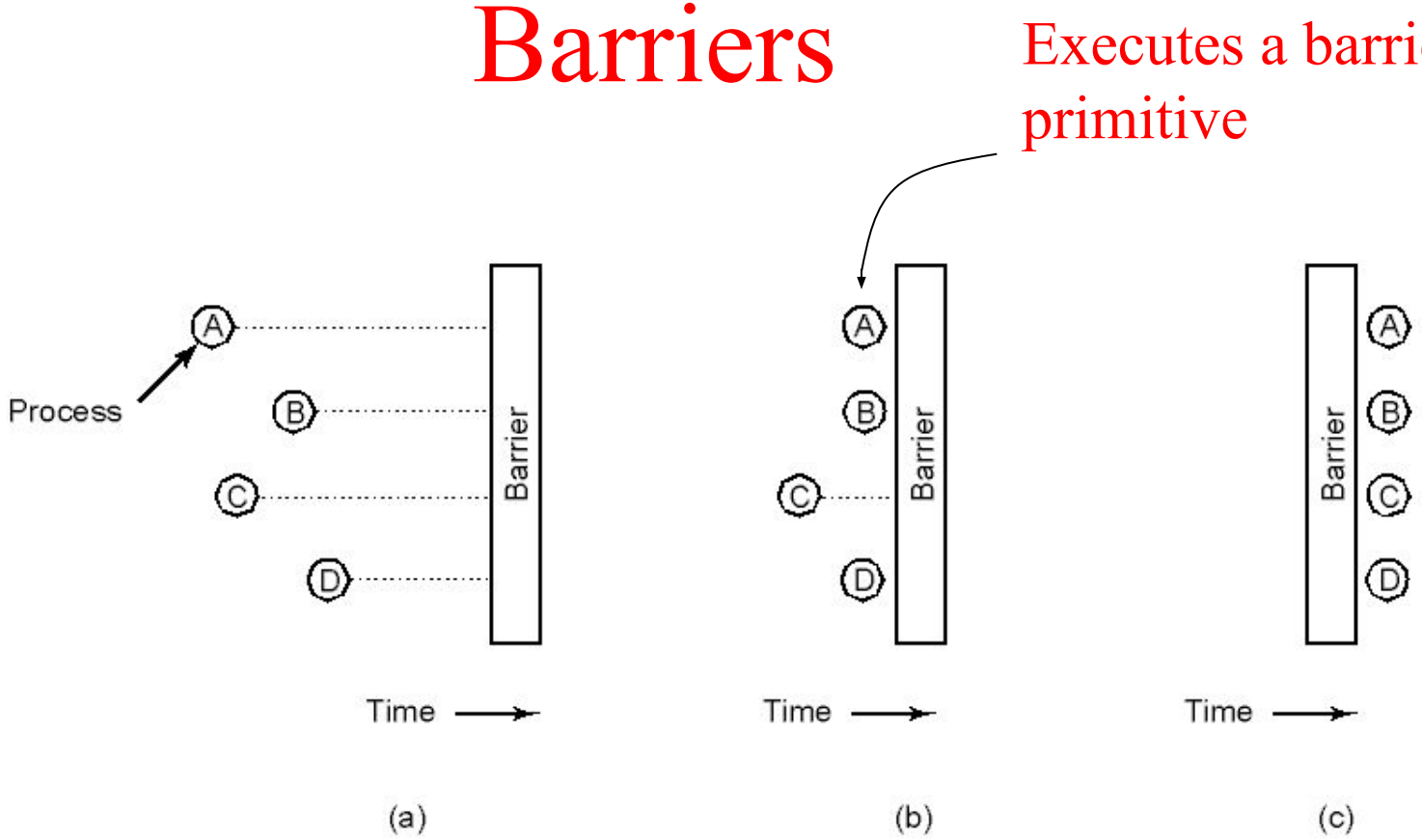
Many Variants of Message Passing

- How messages are addressed?
  - Assign each process a unique address and have messages be addressed to processes
  - Invent and use a new data structure, **mailbox**-a place to buffer a certain number of messages. So here mailboxes have addresses
- Buffering?
  - Mailbox is a kind of buffering
  - Other extreme is to eliminate buffering- if *send* is done first then sender is blocked until receive happens and vice versa (Easier to implement than buffering)

Barriers

- A synchronizing mechanism intended for groups of processes
- Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase.
- This can be done by placing a barrier at the end of each phase.
- When a process reaches the barrier, it is blocked until all processes have reached the barrier.

Barriers



- Use of a barrier
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through

Barriers

```
// Pseudocode Example: Shared variable
barrier(count = 3);

Thread_1() {
    compute_part1();
    wait_at_barrier();
    combine_results();
}

Thread_2() {
    compute_part2();
    wait_at_barrier();
    combine_results();
}

Thread_3() {
    compute_part3();
    wait_at_barrier();
    combine_results();
}
```

Semaphore Usage

- A lock permits only **1** process/thread to be in Critical Section.
- What if we want maximum **m** process/thread to be in the critical section simultaneously ?
- Semaphore from railway analogy
  - Here is a semaphore **initialized to 2** for resource control: