Introduction to OS

Synchronization

MOS 2.3

Mahmoud El-Gayyar

elgayyar@ci.suez.edu.eg

Challenge

- How can we help processes synchronize with each other?
  - E.g., how does one process “know” that another has completed a particular action?
  - E.g., how do separate processes “keep out of each others’ way” with respect to some shared resource
  - E.g., how do process share the load of particularly long computations

Definition – *Atomic Operation*

- An operation that either happens entirely or not at all
  - No partial result is visible or apparent
  - Appears to be non-interruptible
- If two atomic operations happen “at the same time”
  - Effect is as if one is first and the other is second
  - (Usually) don’t know which is which

Hardware Atomic Operations

- On (nearly) all computers, reading and writing operations of machine words can be considered as *atomic*
  - Non-interruptible
  - It either happens or it doesn't
  - Not in conflict with any other operation
- When two attempts to read or write the *same* data, one is first and the other is second
  - Don't know which is which!
- No other guarantees
  - (unless we take extraordinary measures)

Race Condition

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

The final value is 1 instead of the expected result of 2.

Critical Region

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Mutual Exclusion.

Can the same problem occur among processes?

Definitions

- Definition: *race condition*
  - When two or more concurrent activities are trying to do something with the same variable resulting in different values
  - Random outcome

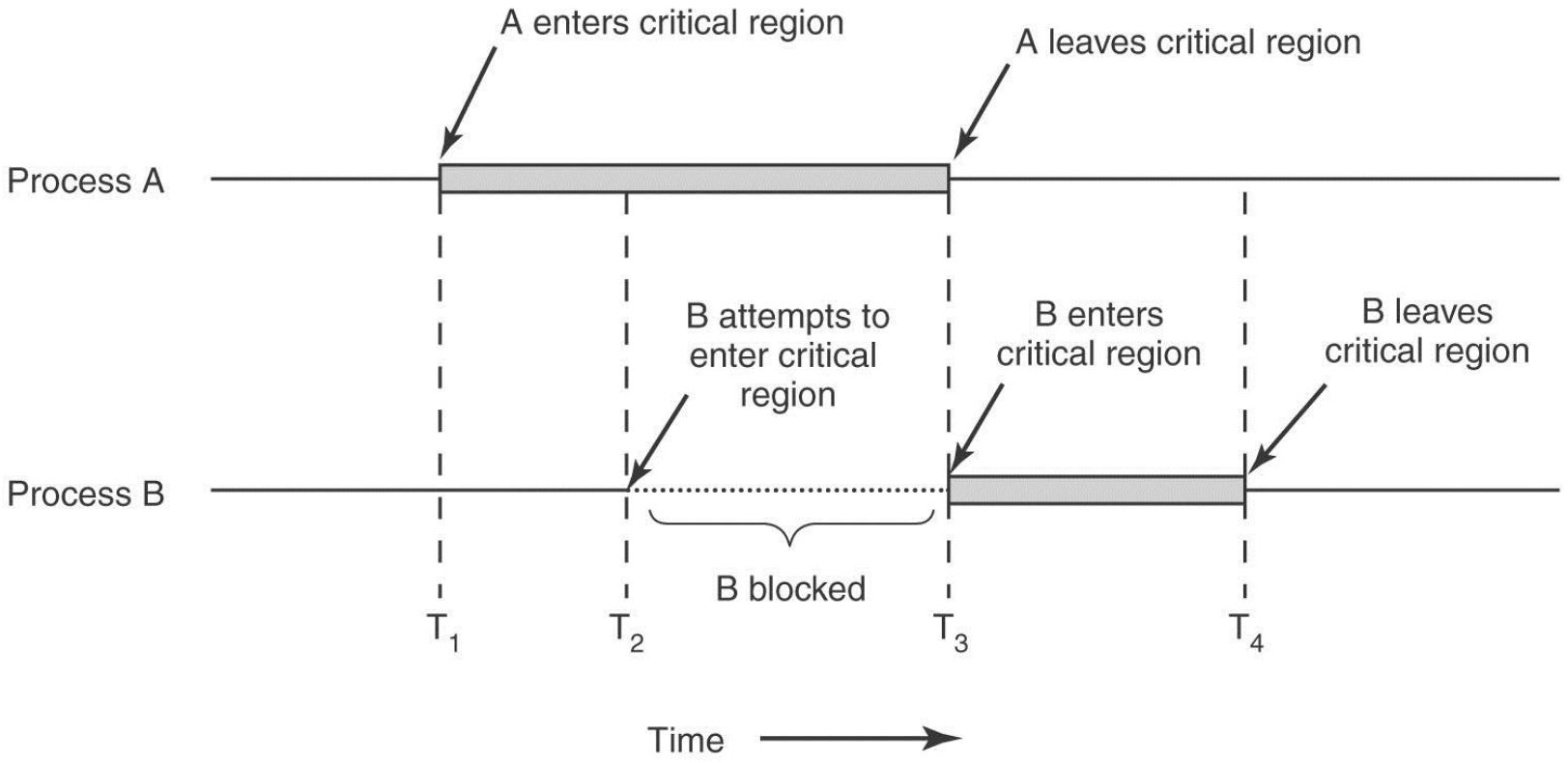
- *Critical Region* (aka *critical section*)
  - One or more fragments of code that operate on the same data, such that at most one activity at a time may be permitted to execute anywhere in that set of fragments.

Critical Section

```
process ()
{
    entry_protocol()
        critical_section()
    exit_protocol()
}
```

- Entry protocol:
  - Process requests entry to critical section
  - Process has to communicate that it entered critical section
- Exit protocol:
  - process communicates to other processes that it leaves critical section

Synchronization – Critical Regions



Deadlock and Starvation

- Enforcing mutual exclusion creates two new problems
  - Deadlocks
    - Processes wait forever for each other to free resources
  - Starvation
    - A process waits forever to be allowed to enter its critical section

- *Implementing mutual exclusion has to account for these problems*

Class Discussion

- How do we keep multiple computations from being in a critical region at the same time?

Requirements – Controlling Access to a Critical Section

1. ***Mutual exclusion***: Only one computation in critical section at a time
2. No assumption about speeds or number of CPUs
3. No process running outside its critical region may block other processes.
4. No ***starvation*** — No process should have to wait forever to enter its critical region.

Possible ways to protect critical section

- Software
  - Use shared lock variables to control access to critical section
  - Busy waiting
- Hardware
  - Disable interrupts
  - Processor provides special instructions
- Higher operating system constructs
  - Semaphores, Monitor, message passing
  - Involvement of scheduler, processes are suspended

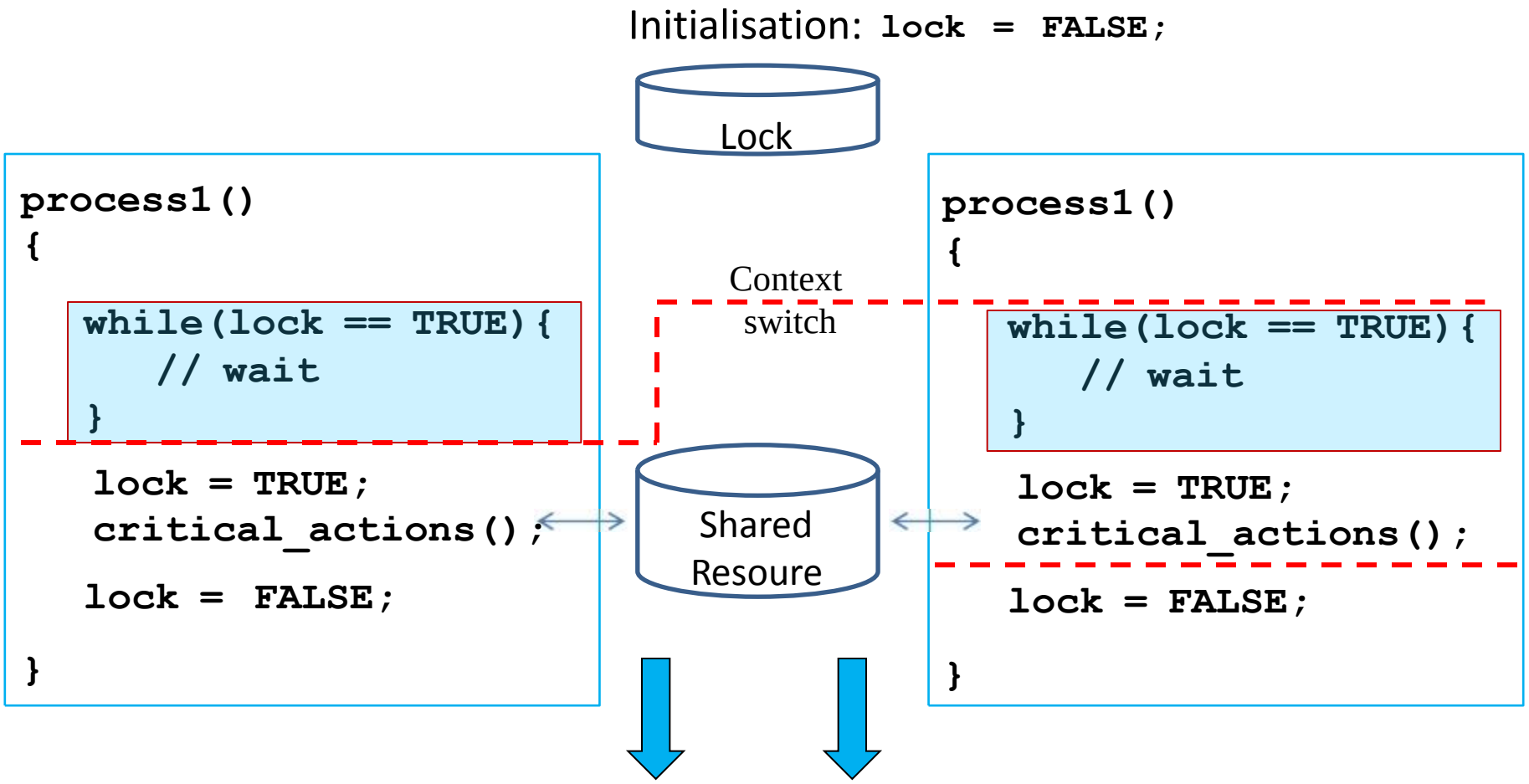
Software Solutions for Mutual Exclusion

Solving the Critical Section Problem

Lock Variables

- Use of shared memory for inter-process communication
- Shared variable **“lock”**
- Used to indicate whether one of the competing processes have entered critical section
  - If lock == 0 (FALSE), then lock is not set
  - If lock == 1 (TRUE), then lock is set
- All processes that compete for a shared resource also share this local variable
  - A process checks the lock
    - If lock not set, process sets the lock and enters critical section
    - Otherwise, waits
- Problem:
  - Lock variable is itself a shared resource, race condition can occur

Example: Shared Lock



Both processes are now in their critical sections !

Busy Waiting

- Also called ***"polling" or "spinning"***
- A process continuously evaluates and consumes CPU cycles without any progress.
- A lock that uses busy waiting is called a ***spin lock***.

Strict Alternation

- Busy-waiting Strategy
- Strict alternation between two processes
  - Use a “token” as a shared variable:
    - Value is ***process ID***
    - indicates which process is the next to enter critical section, set by previous process
- For two processes P0 and P1 (can be extended to n processes)
- Entry to critical section
  - Process Pi busy-waits until token == i (its own process ID)
- Exit from critical section
  - Process Pi sets token to next process ID

Example: Strict Alternation

```
int turn=0 ;
```

Process 0

```
while(TRUE){

   while(turn != 0){
      // wait
   }

     Critical_Section

   turn = 1;

     Non_Critical_Section
      ...
}
```

Process 1

```
while(TRUE){

   while(turn != 1){
      // wait
   }

     Critical_Section

   turn = 0;

     Non_Critical_Section
      ...
}
```

- Mutual exclusion guaranteed
- Lifeness / Progression problem:
  - ✓ Process not in critical region blocks another process (violates condition 3)
  - ✓ taking turns is not a good idea when one of the processes is much slower than the other

Use an Array of Flags

Global Variables

```
boolean flag[2];
```

```
flag[0]=FALSE
flag[1]=FALSE
```

Process 0

```
while(TRUE){
   flag[0] = TRUE;
   while(flag[1]==TRUE){
      // wait
   }
      Critical_Section
   flag[0] = FALSE;
      Non_Critical_Section
      ...
}
```

Context switch

Process 1

```
while(TRUE){
   flag[1] = TRUE;
   while(flag[0]==TRUE){
      // wait
   }
      Critical_Section
   flag[1] = FALSE;
      Non_Critical_Section
      ...
}
```

- Mutual exclusion guaranteed
- Problem: Deadlock may occur due to context switch

Peterson's Algorithm

- Combines strict alteration with flags for indicating interest in entering critical section
- Non-Atomic Locking: works even if there is a race condition
  - Is limited to two processes
  - Uses two shared data items for

```
process ( i )
{
    j = 1 -i ;
    flag[i] = TRUE ;
    turn = j ;
    while (flag[j] &&
           turn == j ) ;

    critical_section() ;

    flag[i] = FALSE ;

    remainder_section() ;
}
```

```
int turn ;
```

Indicates, which of the two processes is ***allowed*** to enter

```
boolean flag[2] ;
```

Indicates, which of the two processes is ***ready*** to enter (both can be ready at the same time)

Peterson's Algorithm

Global Variables

```
boolean flag[2];
int turn;
```

```
flag[0]=FALSE
flag[1]=FALSE
```

```
turn = 0; // or 1
```

Process 0

```
while(TRUE) {
    flag[0] = TRUE;
    turn = 1
    while(flag[1] == TRUE &&
          turn == 1){
      // wait
    }

    Critical_Section

    flag[0] = FALSE;

      Non_Critical_Section
      ...
}
```

Process 1

```
while(TRUE) {
    flag[1] = TRUE;
    turn = 0
    while(flag[0] == TRUE &&
          turn == 0){
      // wait
    }

    Critical_Section

    flag[1] = FALSE;

      Non_Critical_Section
      ...
}
```

Peterson's Algorithm – cont..

- Does it work if both processes enter almost simultaneously?
  - Both will set flag[processID] = TRUE
  - Both try to write the variable turn
  - This is a race condition: if process 1 is the last to write, it wins race
  - Example: Process 1 wins the race, turn = 1
  - Both processes arrive at the while loop
    - Process 0 immediately continues(as turn == 1)
    - Process 1 is waiting in the whileloop (as turn == 1 and flag[0] = TRUE)
- The race condition is not a problem

Peterson's Algorithm – cont..

- Peterson's Algorithm
  - Is a non-atomic locking algorithm
  - **Mutual Exclusion** is preserved
    - Even if flag[i] == flag[j] == TRUE (both processes are ready), the variable turn can only be either i or j (only one of them can enter critical section)
  - **Progress and Bounded Waiting**
    - Progress is guaranteed: If a process indicates interest to enter critical section, it will gain access after the other process is finished
- Problems
  - Still busy-waiting solution
  - Solution for only two processes, can be extended to n. processes, does not work for unknown number of processes

Possible ways to protect critical section

- Software
  - Use shared lock variables to control access to critical section
  - Busy waiting
- Hardware
  - Disable interrupts
  - Processor provides special instructions
- Higher operating system constructs
  - Semaphores, Monitor, message passing
  - Involvement of scheduler, processes are suspended

Hardware Solutions for Mutual Exclusion

Solving the Critical Section Problem

Mutual Exclusion: Disable Interrupts

- Interrupt Disabling
  - A process cannot be interrupted until it enables interrupts again
  - Therefore, guarantees mutual exclusion on a uniprocessor system

- *Disadvantages*
  - Does not work on a multiprocessor architecture
  - It is unwise to give user processes the power to turn off interrupts, what about disabling it and not turning it on again?!!!

The TSL (Test & Set Lock) Instruction

- The CPU executing the TSL instruction locks the ***memory bus*** to prohibit other CPUs from accessing memory until it is done.
- *Disadvantages*
  - Again is based on a ***busy-waiting*** solution.
  - ***Priority inversion problem:***
    - Consider a computer with two processes, H, with high priority, and L, with low priority.
    - The scheduling rules are such that H is run whenever it is in ready state.
    - At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O operation completes). H now begins busy waiting,
    - but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever.

Possible ways to protect critical section

- Software
  - Use shared lock variables to control access to critical section
  - Busy waiting
- Hardware
  - Disable interrupts
  - Processor provides special instructions
- Higher operating system constructs
  - Semaphores, Monitor, message passing
  - Involvement of scheduler, processes are suspended

OS Constructs Solutions for Mutual Exclusion

Solving the Critical Section Problem

Semaphores

Solving the Critical Section Problem

Semaphores

- Synchronization mechanism
- Two or more processes can communicate by means of simple signals
  - One process can be forced to wait for a signal from other processes
- Semaphore acts as a barrier, makes processes wait

```
process ()
{
    wait(S)

        critical_section() ;

    signal(S)

        remainder_section() ;
}
```

Semaphores cont…

```
process ()
{
    wait(S)

        critical_section() ;

    signal(S)

        remainder_section() ;
}
```

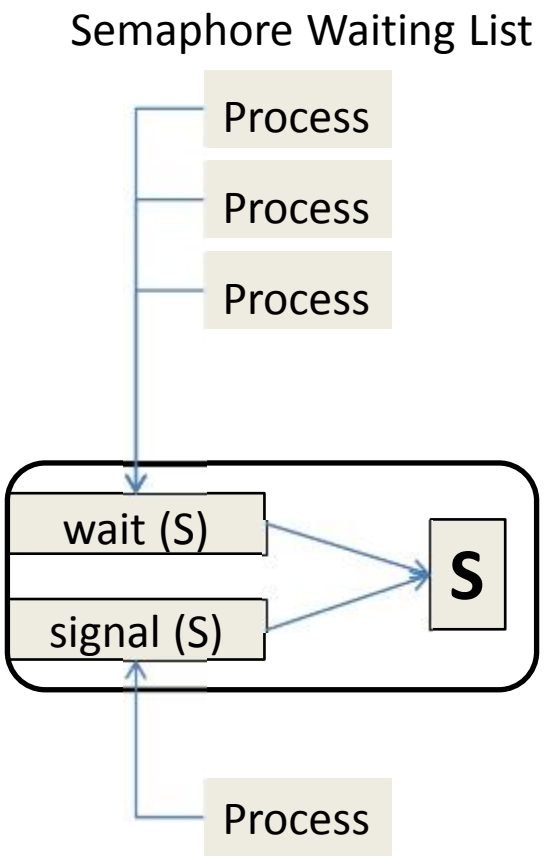
```
wait(S) {
    while(S <= 0) /* busy wait */;
    S -- ;
}
```

```
signal(S) {
    S ++ ;
}
```

- Semaphores are based on a decrement / increment mechanism:
  - The initial value of the semaphore determines how many processes may “pass” (do not have to wait) the semaphore at once
  - If semaphore S > 0, then process can proceed, S is decremented
  - If semaphore S<= 0, then process has to wait for signal

Semaphore Implementation with Blocking of Processes

- Semaphore is an object with the following elements
  - Public methods:
    - ***wait(S), signal(S)***
  - Private
    - Semaphore Counter
    - Waiting queue for processes
- Involves the scheduler:
  - Processes that have to wait will be de-scheduled
  - Waiting processes are held in the semaphore waiting queue



Semaphore Data Structure

- A semaphore is a data structure that contains
  - A counter
  - A waiting queue

```
typedef struct {
   int counter;
   Queue plist ;
} Semaphore ;
```

- Semaphore can only be accessed by two atomic functions
  - wait (S) : decrements semaphore counter
    - If a process calls wait(), the counter is decremented, if it is zero – semaphore blocks calling process
  - signal(S) : increments semaphore counter
    - processes calling signal(S) wake up other processes

Binary Semaphore

- Binary semaphore (also called a "***mutex***"):
  - Initialized with 1
  - Value of a binary semaphore toggles between 0 and 1
  - A process calling wait() may continue processing if semaphore value == 1, will set it to zero
  - Used for enforcing mutual exclusion

Binary Semaphore – Mutual Exclusion

Initialization:

```
init(mutex,1)
```

Process P_i:

```
wait(mutex);

Critical_Section

signal(mutex);

Non_Critical_Section
   ...
```

- Guarantees mutual exclusion
  - Semaphore initialised to 1: maximal one process may enter critical section
- Binary semaphores can be used to implement mutex locks
  - N processes share a semaphore “mutex”

Binary Semaphore – Mutual Exclusion

Semaphore:

```
   int value ;
   queue plist ;
} Semaphore ;
```

Initialization: `init(S,1)`

```
process ()
{
   wait(S)

      critical_section() ;

   signal(S)

      remainder_section() ;
}
```

```
wait(Semaphore S) {
   if(S.value > 0) S.value -- ;
   else {
      add this process to S.plist;
      block();
   }
}
```

```
signal(Semaphore S) {
   if(S.plist is empty) S.value ++;
   else {
      remove a process P from S.plist;
      wakeup(P);
   }
}
```

Monitors

Solving the Critical Section Problem

Monitor

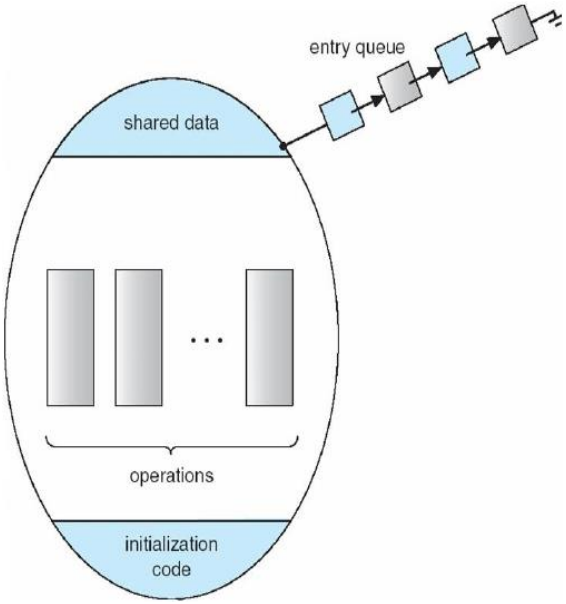
- A monitor is a software construct that serves two purposes:
  - enforces mutual exclusion of concurrent access to shared data objects
    - Processes have to acquire a lock to access such a shared resource
  - Support conditional synchronization between processes accessing shared data:
    - Multiple processes may use monitor-specific wait()/signal() mechanisms to wait for particular conditions to hold

Monitor cont…

- Monitors are typically supported by a programming language
  - Language-specific software construct
  - e.g. Java (synchronized keyword)
- Programs using monitors are supposed to allow easier implementation of mutual exclusion and synchronization

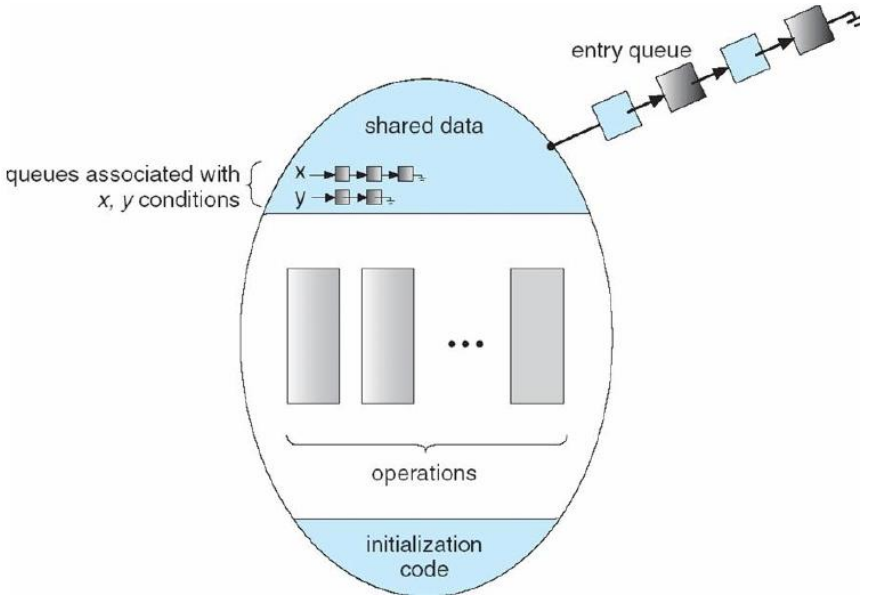
Monitor Characteristics

- A monitor is a software construct consisting of
  - One or more procedures
  - Some local data that can only be accessed via these procedures
- Object-oriented concepts
  - Local variables accessible only by the monitor's procedures (methods)
- Processes "enter" monitor when they invoke one of the monitor's procedures
- Mutual exclusion:
  - Only one process at a time may call one of these Procedures and "enter" the monitor
  - All other processes have to wait



Process Synchronization

- A monitor also supports process synchronization with condition variables
  - Only accessible within the monitor with the functions ***wait(condition)*** and ***signal(condition)***
- A monitor may maintain a set of these condition variables
- For each condition variable, the monitor maintains a waiting queue



Producer - Consumer

```
monitor boundedBuffer
{
        char b[N]; int count, in, out ;
        condition notfull, notempty;

        void append(char item) {
             if (count == N) wait(notfull) ;
             b[in] = item;
             in = (in+1) mod N;
             count++;
             signal(notempty);
        }
        char take() {
             if (count == 0) wait(notempty) ;
             item = b[out];
             out = (out+1) mod N;
             count--;
             signal(notfull);
        }
}
```

Acquire Lock

ReleaseLock

Acquire Lock

ReleaseLock

Mutual Exclusive Execution

Mutual Exclusion in Java

- Mutexes and condition variables are built in to every Java object.
  - no explicit classes for mutuxes and condition variables
- Every object is/has a “monitor” .
  - At most one thread may “own” any given object’s monitor.
  - A thread becomes the owner of an object’s monitor by
    - executing a method declared as ***synchronized***
- some methods may choose not to enforce mutual exclusion (unsynchronized)

Message Passing

Solving the Critical Section Problem

Message Passing

- So far, we used shared memory for synchronization
- Message passing can be used for mutual exclusion and synchronization
- Synchronization
  - Send(): blocking, non-blocking
  - Receive(): blocking, non-blocking, test for arrival
- Message management at receiver side
  - Queue, messages remain until consumed
  - Buffer: new messages overwrite old messages
- Addressing
  - 1:1, 1:N

Mutual Exclusion with Message Passing

Initialization:

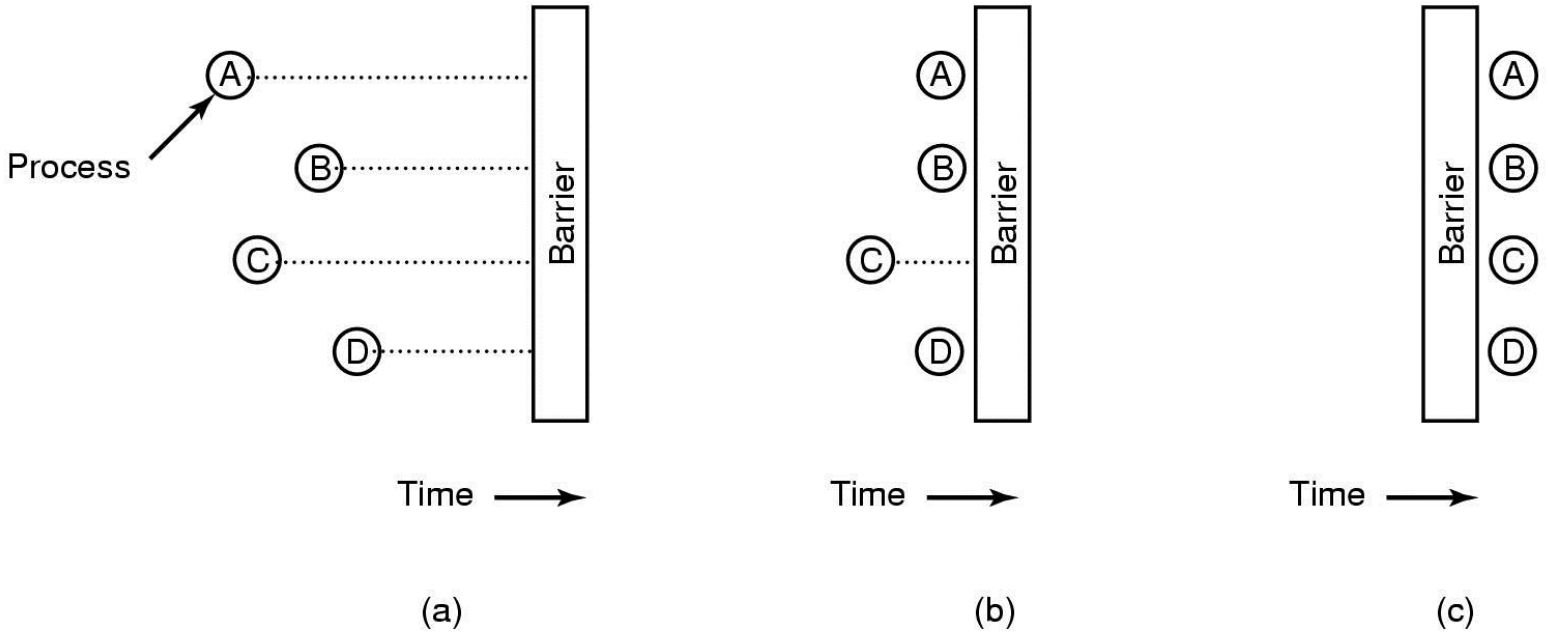
```
Mailbox mailbox;
send(mailbox, NULL); //put token into mailbox
```

Process P_i:

```
Message token;
receive(mailbox, token);
Critical_Section
send(mailbox, token);
Non_Critical_Section
   ...
```

- Empty message used as a “token” that is exchanged between processes
- Blocking receive()
- Non-blocking send()
- Mailbox initialized with a single empty message used as the token
- If process receives message, the receive() function is unblocked,
- it performs critical section and puts same message back into mailbox

Barriers



Barriers are intended for synchronizing groups of processes

Often used in scientific computations.

Review

- What is a race condition?
- Can the priority inversion problem happen with user-level?
- In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used? Explain

Review

- If a system has only two processes, does it make sense to use a barrier to synchronize them? Why or why not?
- Suppose that we have a message-passing system using mailboxes. When sending to a full mailbox or trying to receive from an empty one, a process does not block. Instead, it gets an error code back. The process responds to the error code by just trying again, over and over, until it succeeds. Does this scheme lead to race conditions?

Review

- A fast food restaurant has four kinds of employees: (1) order takers, who take customers‘ orders; (2) cooks, who prepare the food; (3) packaging specialists, who stuff the food into bags; and (4) cashiers, who give the bags to customers and take their money. Each employee can be regarded as a communicating sequential process. What form of inter-process communication do they use? Relate this model to processes in UNIX.