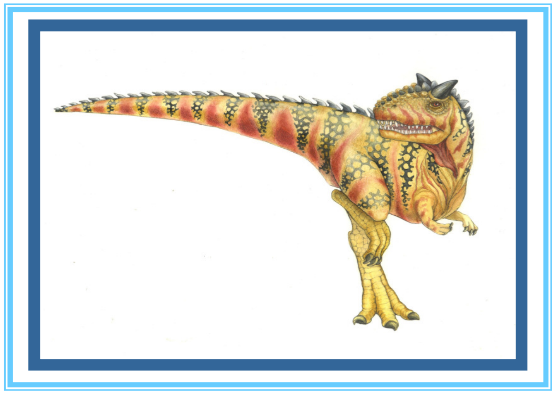
Lecture 11-13
Synchronization

Dr. Shamim Akhter



Silberschatz, Galvin and Gagne ©2009

Processes

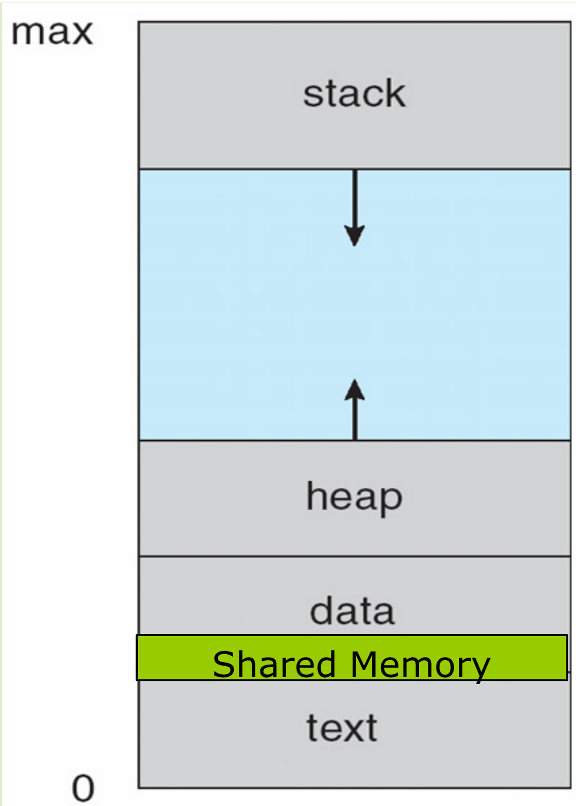
- A process contains everything needed for execution
  - Address space
  - PCB
- So far, processes have limited ability to pass data
  - Parents get one chance to pass everything at fork()
  - But what if the child wants to talk back?
  - And, what about processes with different ancestry?
- Yet sometimes processes may wish to cooperate
  - But how to communicate? Each process is an island
  - The OS needs to get involved to bridge the gap
  - OS provides system calls to support **Inter-Process Communication (IPC)**

Interprocess Communication

- Cooperating processes need **interprocess communication (IPC)**
- Two IPC models -OS provides mechanisms to communicate (System Call)
  - Shared memory
    - multiple processes can read/write same physical portion of memory;
    - System call to declare shared region
  - Message passing
    - communication channel provided through send()/receive() system calls

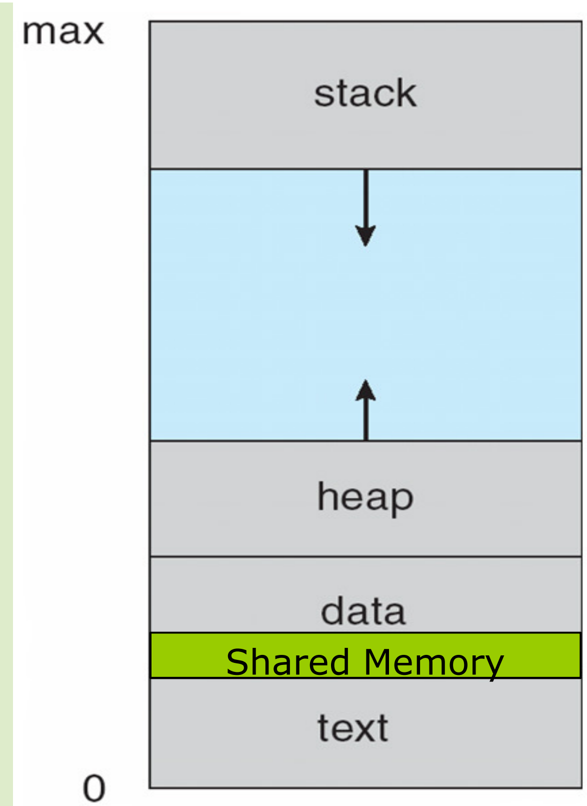
Shared Memory- 3.3.5(section)

Process P1		Process P2



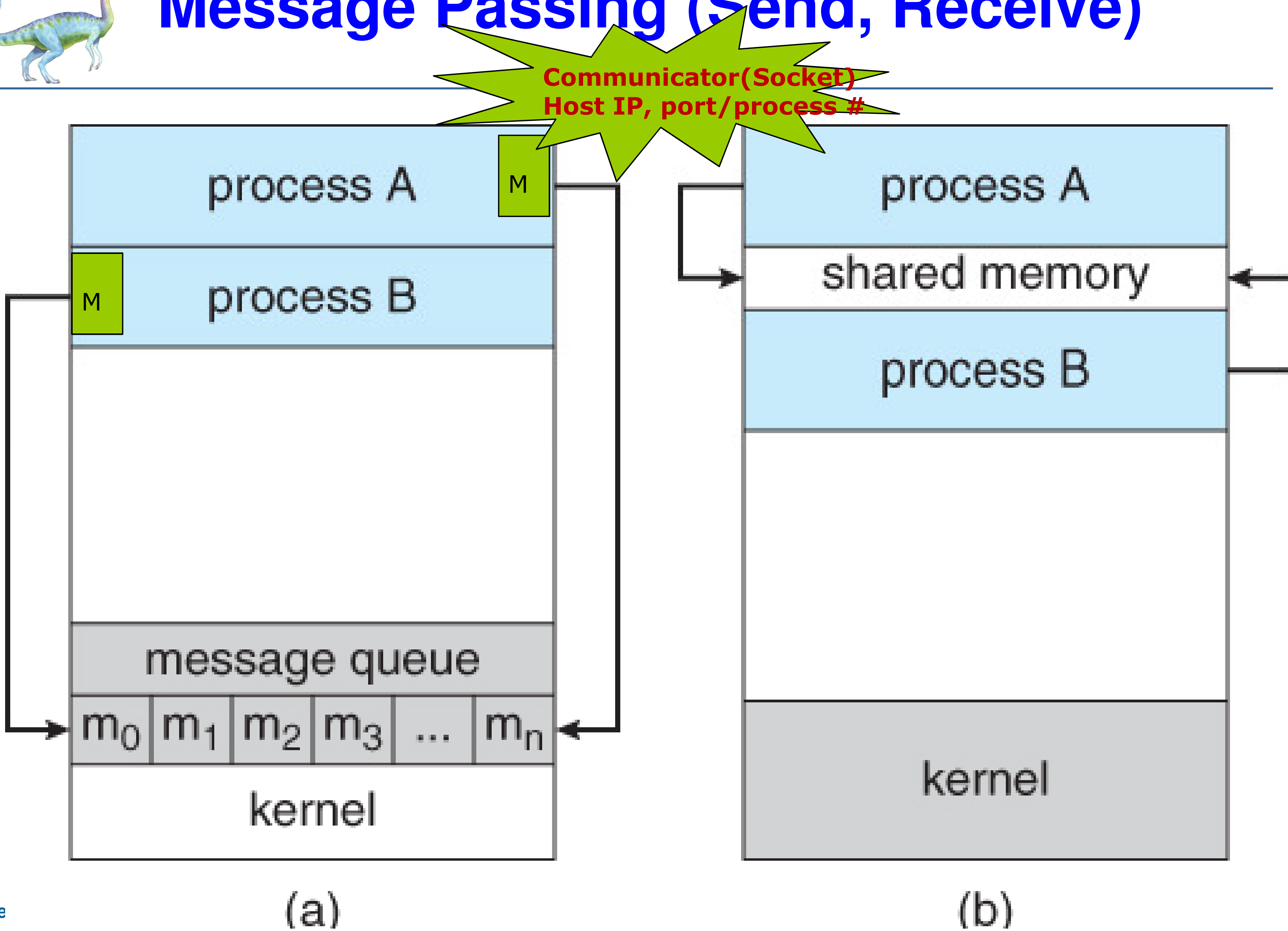
shmget()- system call
Creates shared
memory segment

Shared memory system
-Faster why?
- No need kernel call



shmctl()- system call
Assign ownership to
another user. Provides
permission to access
mapped address.

Message Passing (Send, Receive)



Shared-Memory Issues

How one process can pass information to another process

Emerging critical activities: two or more process do not get into each other's way

Process dependency needs proper sequencing

How about Threads??

Producer-Consumer Problem

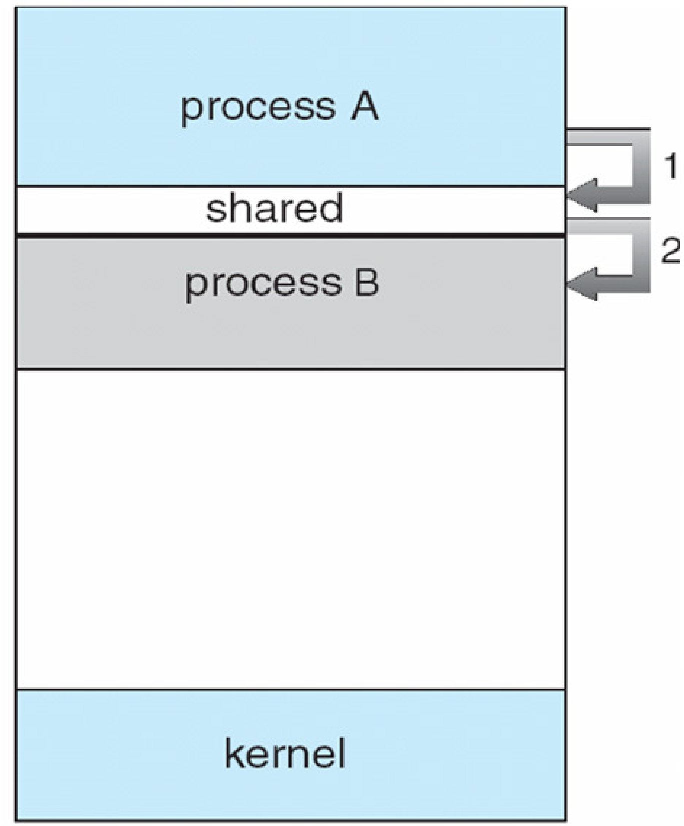
- Problem: There is a set of resource buffers shared by producer and consumer threads
- Producer inserts resources into the buffer set
  - Output, disk blocks, memory pages, processes, etc.
- Consumer removes resources from the buffer set
  - Whatever is generated by the producer
- Producer and consumer execute at different rates
  - No serialization of one behind the other
  - Tasks are independent (easier to think about)
  - The buffer set allows each to run without explicit handoff

- Paradigm for cooperating processes,
  - **unbounded-buffer :** places no practical limit on the size of the buffer
  - **bounded-buffer :** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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



bounded buffer

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  */
}
```

Count=5 @ shared memory

Producer	Consumer
Count++	Count--
4, 5, 6 !	
Reg1=Counter Reg1=Reg1+1 Counter=Reg1	Reg2=Counter Reg2=Reg2-1 Counter=Reg2
However, following situation may arise:	
Reg1=counter 5	
Reg1=Reg1+1 6	Reg2=counter 5
	Reg2=Reg2-1 4
Counter=Reg1	Counter=Reg2
Race Condition	

Nondeterministic
Behavior
4/6 !

Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.

How to avoid Race Condition?

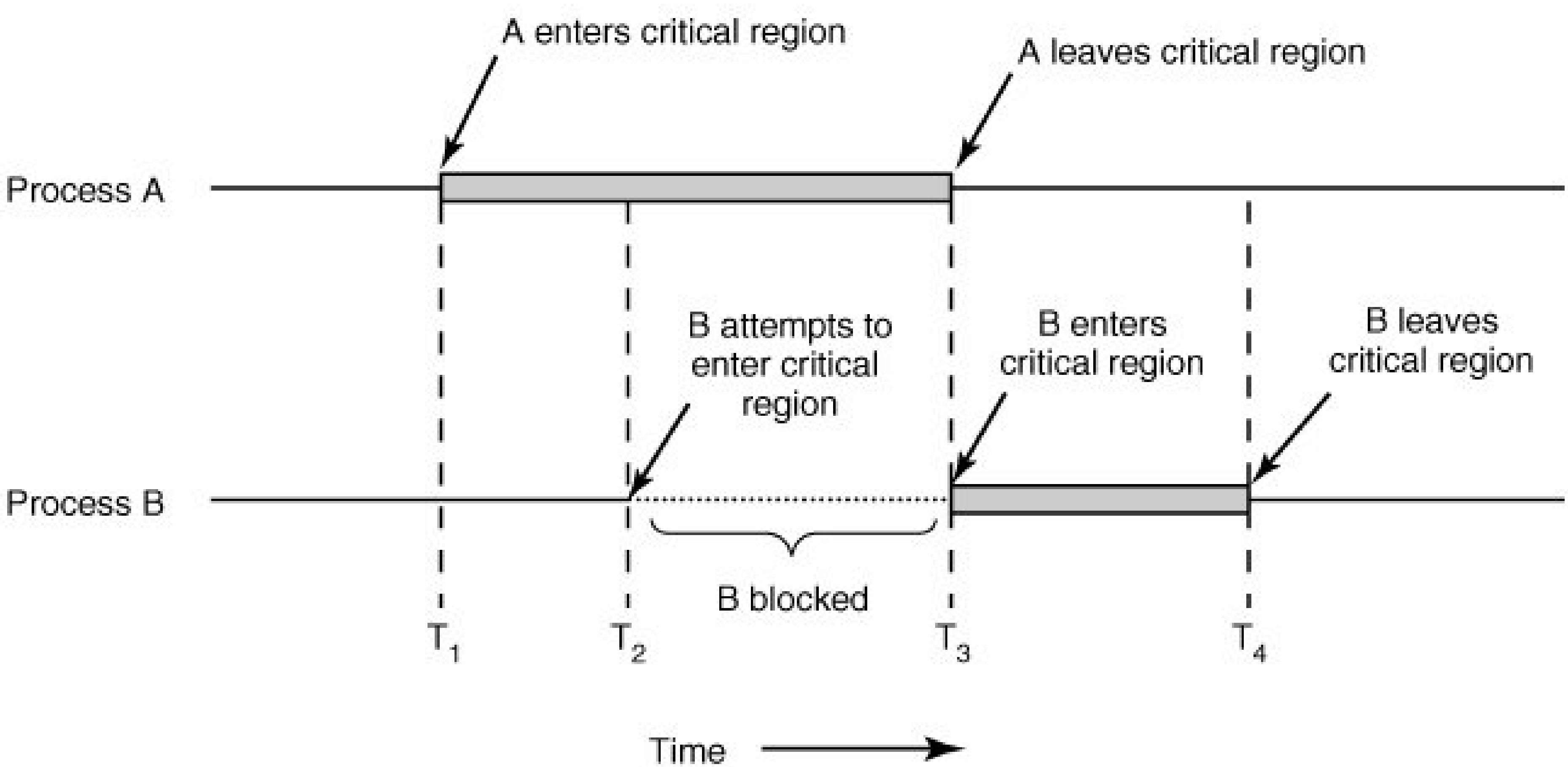
- Prohibit more than one process from reading and writing the shared data at the same time.

- **Mutual exclusion:**
  - making sure that if one process is using a shared variable/file, the other processes will be excluded from doing the same.

- **Critical section/Critical region:**
  - part of the program where the shared memory is accessed.

```
do{
        entry section
                Critical Section (updating variables, table, file etc)
        exit section
                remainder section
} while(1);
```

Mutual Exclusion using critical regions



Modern Operating System @Tanenbaum

Critical Section
Four Conditions to have a good solution

Mutual Exclusion

1. No two processes may be simultaneously inside their critical regions.

Relative Speed

2. No assumptions may be made about speeds or the number of CPUs.

Progress

3. No process running outside its critical region may block other processes.

Bounded Wait

4. No process should have to wait forever to enter its critical region.

Bounded # of time other process can enter the critical section, between a process request to a process granted to enter the critical section

Reason Of Race Condition

- Context Switch
  - allow concurrent (seems) access to the critical section.

- Sol^n:

  Why not stop context switching during critical section execution.

Disable Interrupt

— Disabling Interrupts

```
{
  ...
  disableINT();
  critical_code();
  enableINT();
  ...
}
```

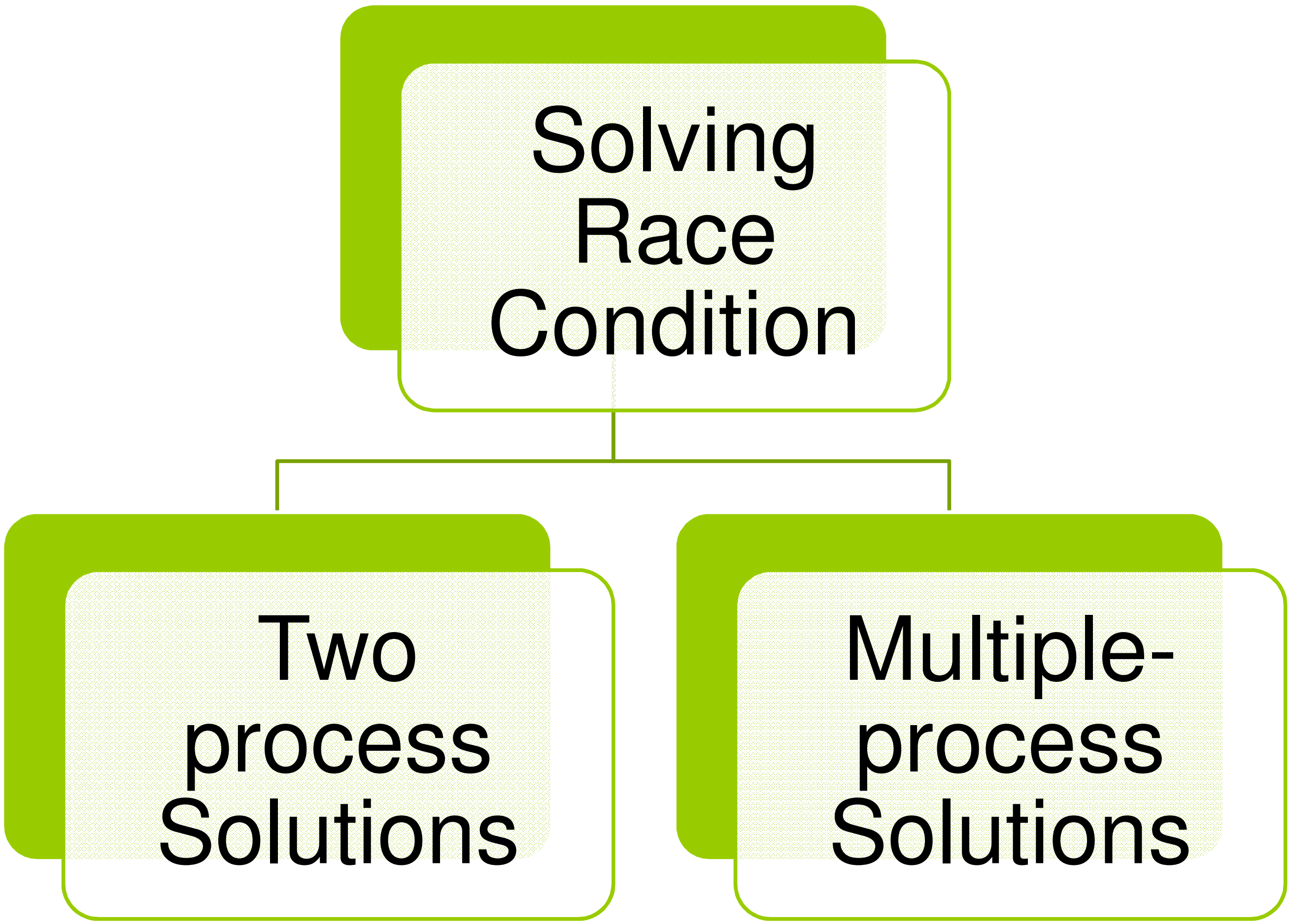
Problems:

- It's not wise to give user process the power of turning off INTs.
  - Suppose one did it, and never turned them on again
- useless for multiprocessor system

Disable/Enable message needs to pass other processors. Takes time and decrease performance

Disabling INTs is often a useful technique within the kernel itself but is not a general mutual exclusion mechanism for user processes.

Solving Race Condition



Mutual Exclusion with Busy Waiting- **Lock Variable Method**

— Lock Variables

```
1 int lock=0; //shared variable
2 {
3    ...
4    while(lock); //busy waiting
5    lock=1;
6    critical_code();
7    lock=0;
8    ...
9 }
```

$t0 = address(lock)
store $t0, 1

Violates Condition ME

Problem:

- What if an interrupt occurs right at line 5?
- Checking the lock again while backing from an interrupt?

Algorithm 1: Strict Alternation

— Strict Alternation

initially turn=0

Spin Lock

Process 0

turn==1

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

Process 1

turn==0

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

Problem: violates **Relative Speed/ Progress as well**

- One process can be blocked by another not in its critical region.
- Requires the two processes strictly alternate in entering their critical region.

Algorithm 2: Modified Strict Alternation

Problem Strict Alternation: Not sufficient information about process state.
Only remember-which process is allowed to enter Critical Section.
"turn =0/1"

Solution: Replace "turn" with a Boolean array flag[2];
flag[0]=flag[1]=0;

Process 0	Process 1
while(TRUE)	while(TRUE)
{	{
flag[0]=1;	flag[1]=1;
while (flag[1]);	while (flag[0]);
Critical Section	Critical Section
flag[0]=0;	flag[1]=0;
}	}

Context Switch

Mutual Exclusion-OK

Relative Speed- No problem

Progress - Not OK
- start same time
  context switch problem

Switching Order : while & flag

Results two (2) processes @ CS

flag[0]=flag[1]=0;

Process 0

```
while(TRUE)
{
    while (flag[1]);
      flag[0]=1;

          Critical Section

     flag[0]=0;
}
```

Process 1

```
while(TRUE)
{
    while (flag[0]);
     flag[1]=1;

          Critical Section

    flag[1]=0;

}
```

Algorithm 3: Peterson's Solution

— Peterson's Solution

Combined the idea of
-lock variable
-warning variable

```
int interest[0] = 0;
int interest[1] = 0;
int turn;
```

turn=0;

P0

```
interest[0] = 1;
turn = 1;
while(interest[1] == 1
             && turn == 1);
critical_section();
interest[0] = 0;
```

P1

```
interest[1] = 1;
turn = 0;
while(interest[0] == 1
             && turn == 0);
critical_section();
interest[1] = 0;
```

Wikipedia – Peterson's algorithm

http://en.wikipedia.org/wiki/Peterson%27s_algorithm

Multiple-Process Solution

Bakery Algorithm

Bakeries, Ice-cream Shops

Lowest Token is served next

Distributed Deterministic

Two Customers can receive same numbers

Need a **Tie Breaker**

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
            while (choosing[j]) ;
            while ((number[j] != 0) && (number[j,j] < number[i,i])) ;
    }
      critical section
    number[i] = 0;
      remainder section
} while (1);
```

number[j],j	number[i],i

Book's Mistake

- Notation <≡ lexicographical order (ticket #, process id #)
  - (a,b) < c,d) if $a < c$ or if $a = c$ and $b < d$
  - max (a_0,…, a_{n-1}) is a number, k, such that $k \geq a_i$ for i - 0, …, n – 1
- Shared data

    boolean choosing[n];

    int number[n];

  Data structures are initialized to **false** and **0** respectively

Silberschatz, Galvin and Gagne ©2009

Example: Bakery Algorithm

	Process 0	Process 1	Process 2
	do{	do{	do{
Case1	number[0]=1	number[1]=2	number[2]=3
Case2	number[0]=1	number[1]=2	number[2]=1
	} while(1);	}while(1);	}while(1);

Case1: all processes arrive in order.

(1,0<2,1), (2,1<3,2)

0 in CS   1 in CS

Case2: all processes arrive same time.

(1,0<1,2)   (2,1<1,2) [process 2 arrives before 1]

0 in CS   2 in CS

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
            while (choosing[j]) ;
            while ((number[j] != 0) && (number[j,j] < number[i,i])) ;
    }
      critical section
    number[i] = 0;
      remainder section
} while (1);
```

number[j],j   number[i],i

Always lowest token will be served next

Hardware Solution: TSL instruction

- **Atomic Operation**:

some operations those read and change data within a single, uninterruptible step.

Example: Test and Set Lock instruction

TSL returns the current value of a memory location and replaces it with a given new value- TRUE

Initially Lock =0

```
boolean TestAndSet ( boolean & target)
{
    boolean rv= target;
    target= true;
    return rv;
}
```

Two memory cycles
1) Read
2) Write

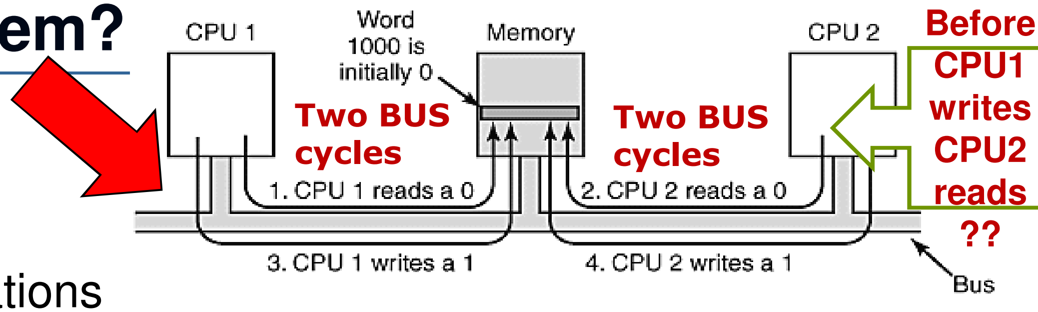
```
do
{
   while(TestAndSet(Lock));

          Critical Section

    lock=false;

}while(1);
```

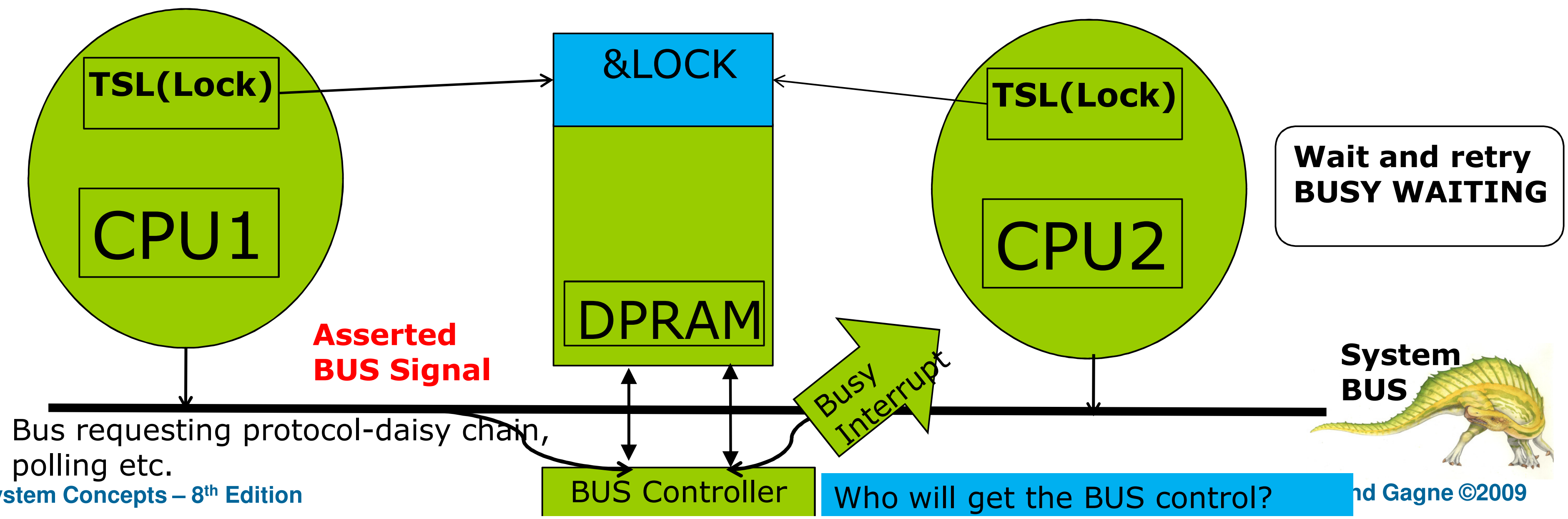
Why does it call- H/W Solution??

- **Why does it call- Hardware Solution?**
- **How about Multiprocessors system?**
- Solution:

Hardware Lock (Memory Barrier):

- Provide synchronization on memory operations
- A processor generates signal and that helps to prevent other processors from using the system bus.
- Example: Lock instruction 8086(intel)-a full barrier





- Both Peterson's solution and solution using TSL are correct
  - but both have BUSY WAITING.
  - Wasting CPU cycle.

Sleep and Wakeup System Calls

Sleep & Wakeup

Solves Busy Waiting

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

```c
void producer(){
  int item;
  while(TRUE){
    item = produce_item();
    if(count == N)
      sleep();
    insert_item(item);
    count++;
    if(count == 1)
      wakeup(consumer);
  }
}
```

```c
void consumer(){
  int item;
  while(TRUE){
    if(count == 0)
      sleep();
    item = rm_item();
    count--;
    if(count == N - 1)
      wakeup(producer);
    consume_item(item);
  }
}
```

Modern Operating System @Tanenbaum

Sleep and Wakeup System Calls

Sleep & Wakeup

Producer and consumer

Both can be in sleep() forever

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

```c
void producer(){
  int item;
  while(TRUE){
    item = produce_item();
    if(count == N)
      sleep();
    insert_item(item);
    count++;
    if(count == 1)
      wakeup(consumer);
  }
}
```

```c
void consumer(){
  int item;
  while(TRUE){
    if(count == 0)
      sleep();
    item = rm_item();
    count--;
    if(count == N - 1)
      wakeup(producer);
    consume_item(item);
  }
}
```

Deadlock!

Modern Operating System @Tanenbaum

Producer-Consumer Problem

— Race Condition

Problem

1. Consumer is going to sleep upon seeing an empty buffer, but INT occurs;
2. Producer inserts an item, increasing count to 1, then call wakeup(consumer);
3. But the consumer is not asleep, though count was 0. So *the wakeup() signal is lost*;
4. Consumer is back from INT remembering count is 0, and goes to sleep;
5. Producer sooner or later will fill up the buffer and also goes to sleep;
6. Both will sleep forever, and waiting to be waken up by the other process. Deadlock!

Producer-Consumer Problem

— Race Condition

Solution: Add a *wakeup waiting bit*

1. The bit is set, when a wakeup is sent to an awaken process;
2. Later, when the process wants to sleep, it checks the bit first. Turns it off if it's set, and stays awake.

What if many processes try going to sleep?

Higher-Level Synchronization

- We looked at using locks to provide mutual exclusion
- Locks work, but they have some drawbacks when critical sections are long
  - Spinlocks – inefficient
  - Disabling interrupts – can miss or delay important events
- Instead, we want synchronization mechanisms that
  - Block waiters
  - Leave interrupts enabled inside the critical section
- Look at two common high-level mechanisms
  - Semaphores: binary and counting semaphore
  - Monitors: mutexes and condition variables

Silberschatz, Galvin and Gagne ©2009

What is a Semaphore?

- A locking mechanism
- An integer or ADT

that can only be operated with:

Block waiters, interrupts enabled within CS

Hide implementation view.

During Semaphore updating: No other process can access the Semaphore

Atomic Operations	
P()	V()
Wait()	Signal()
Down()	Up()
Decrement()	Increment()
...	...

Automatically done all process by a single command

```
down(S){
  while(S<=0);
  S--;
}
```

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

Randomly Wakeup One process

More meaningful names:

- increment_and_wake_a_waiting_process_if_any()
- decrement_and_block_if_the_result_is_negative()

Semaphore

How to ensure atomic?

1. For single CPU, implement up() and down() as system calls, with the OS disabling all interrupts while accessing the semaphore;
2. For multiple CPUs, to make sure only one CPU at a time examines the semaphore, a lock variable should be used with the TSL instructions.

This section is based on (Tanenbaum, 2001)

Semaphore is a Special Integer

A semaphore is like an integer, with three differences:

1. You can initialize its value to any integer, but after that the only operations you are allowed to perform are *increment* (S++) and *decrement* (S- -).
2. When a thread decrements the semaphore, if the result is negative ($S \leq 0$), the thread blocks itself and cannot continue until another thread increments the semaphore.
3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

This section is based on (Tanenbaum, 2001)

Still busy waiting? Sol- process block

wait(s):

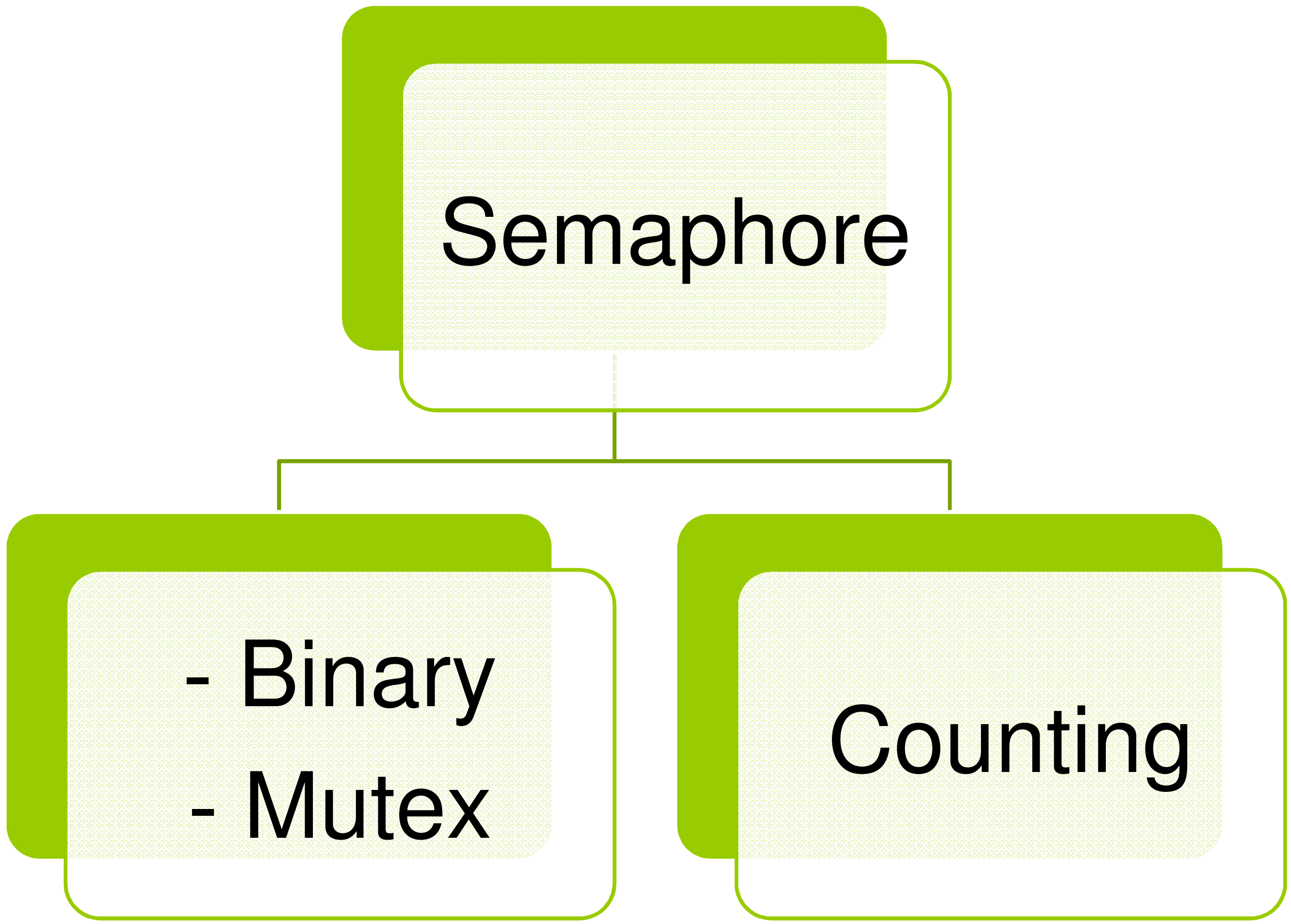
```
s.count=s.count-1;
if(s.count < 0) then
{
 place this process in s.queue;
  block this process
}
```

```
typedef  struct {

    int value;
    struct process *queue;

} semaphore;
```

signal(s):

```
s.count=s.count+1;

if(s.count <= 0) then
{
  remove a process P from s.queue;
  place process P on ready queue
}
```

Semaphore Types



Binary Semaphore

ADT- Hide details implementation from user	
wait( )	signal( )
{	{

if (BS. value=1) then
BS.value=0
else
Place process in S.Queue and
block the process
}
	if (BS. Queue is empty)
	BS.value=1
	else
	Remove a process from BS.Queue
	and place p in ready queue.
	}

WHY: do we need Binary semaphore code?
Can't we use COUNTING semaphore with value 1?

Answer: They are different, BS supports value 1/0.

Where as: signal() before wait() makes semaphore value to more than 1.

Example

i++ is not atomic in assembly language

```
LOAD    r0 , [i]     ;load the value of 'i' into
                     ;a register from memory
ADD   r0, r0, 1      ;increment the value
                     ;in the register
STORE   r0, [i]      ;write the updated
                     ;value back to memory
```

Interrupts might occur in between. So, i++ needs to be protected .

This section is based on (Tanenbaum, 2001), Pages 103 – 105
slides taken from youtube

Example: Binary and Counting Semaphore

	Binary		Counting
	BS=1 T1, T2		S=3 T1, T2, T3, T4, T5
	wait() i++ ; critical region (CR) Signal()		wait() i++ ; critical region (CR) Signal()
STEP 1	T1 enters CR , BS=0		T1 enters CR, S=2
STEP 2	T2 want to enters CR BS.Queue[0]=T2		T2 enters CR, S=1
STEP 3	T1 exits CR , BS=1, remove T2	STEP 3	T3 enters CR, S=0
STEP 4	T2 enters CR , BS=0	STEP 4	T4 enters CR, S=-1 S.Queue[0]=T4
		STEP 5	T5 enters CR, S=-2 S.Queue[1]=T5
		STEP 6	T1 exits CR S=-1 T5 enters
		STEP 7	T2 exits CR S =0 T4 enters

Mutex: A Special BS

```
mutex_lock:

  TSL REGISTER, MUTEX
  CMP REGISTER, #0
  JZE ok_go_CR

  JMP mutex_lock

ok_go_CR :
         critical section entered
```

```
mutex_unlock:

  MOVE  MUTEX, #0

  RET to caller
```

BS	MUTEX
P/T can be lock itself and unlock by others	P/T locks itself and unlocks itself
Work with signal	Work with Token
Traffic Light	ATM Booth
1 to enter CR	0 to enter CR

Problem with Semaphores

- Programmers must have knowledge to use semaphores.
- Otherwise: deadlock situation may arrive.

```
init(&sem1, 0);

init(&sem2, 0);
```

Process One

```
a1;
wait(&sem2);
signal(&sem1);
a2;
```

Process Two

```
b1;
wait(&sem1);
signal(&sem2);
b2;
```

Deadlock

Monitor

- High level synchronization construct
  - defined with collection of procedures, variables and data structures
- Only one process can active in a monitor at a time.
- Up to the compiler (trace monitor specially)
  - how to implement Mutual Exclusion (ME)
  - less likely to get something wrong
- Programmer just select critical region and put code inside monitor
  - compiler determines the rest

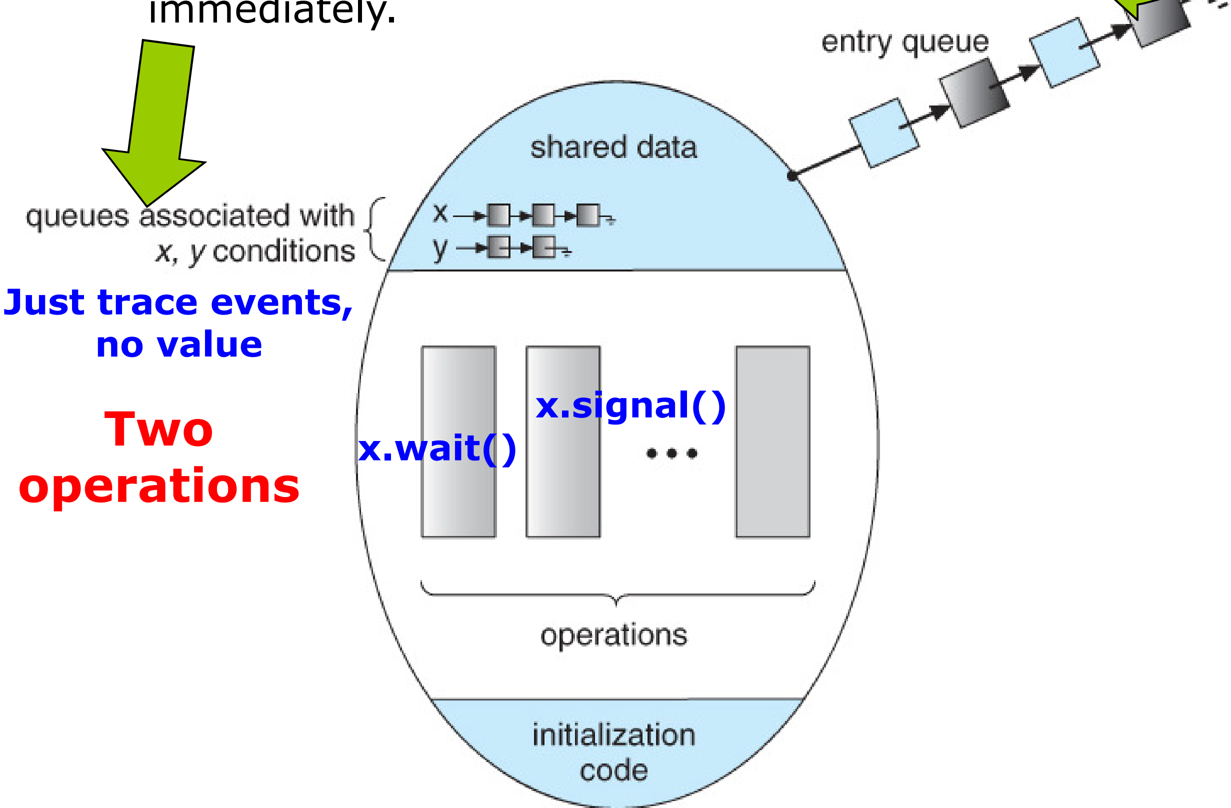
Monitor: Implementation

```
monitor monitor-name {
   shared variable declarations

   procedure body P1 ( . . . )
       . . .
   }

   procedure body P2 ( . . . )
       . . .
   }

   . . .
   {
       initialization code
   }
}
```

1. One process at a time can be active in monitor.
2. A thread/process @ monitor may have to block itself because of its request may not complete immediately.

Just trace events, no value

Two operations



Monitor Example

Synchronization mechanisms need more than just mutual exclusion; also need a way to wait for another thread/process to do something (e.g., wait for a character to be added to the buffer)

Process needs to block.
How they know buffer is full or empty?

Conditional variable with wait()& signal()

Step 1

C	P m

Entry Queue

full	empty
P	

Step 2

P	C m

Entry Queue

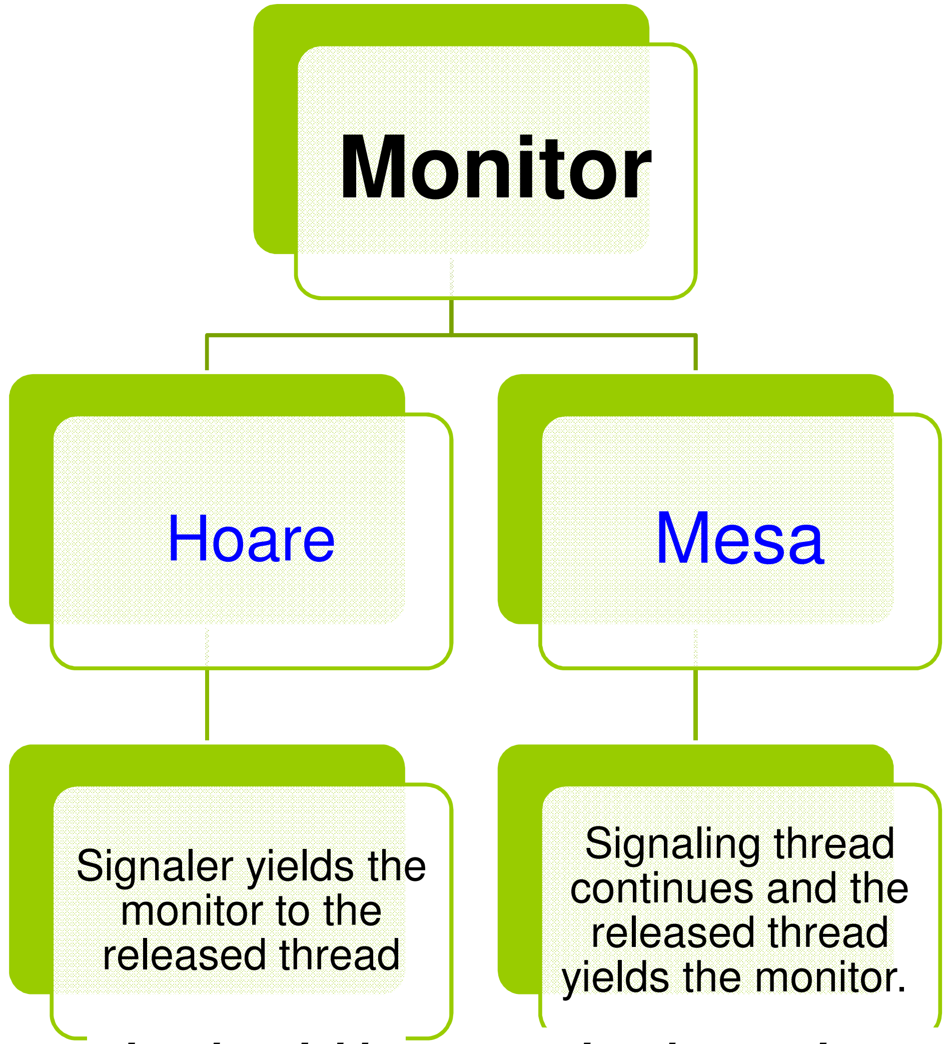
full	empty
	C

```
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;
```

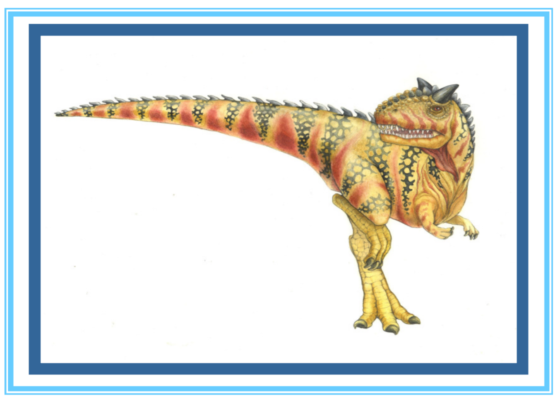
Who will hold the Monitor Lock!

Signaling process / Released process



signaler yields ***signaler continues***

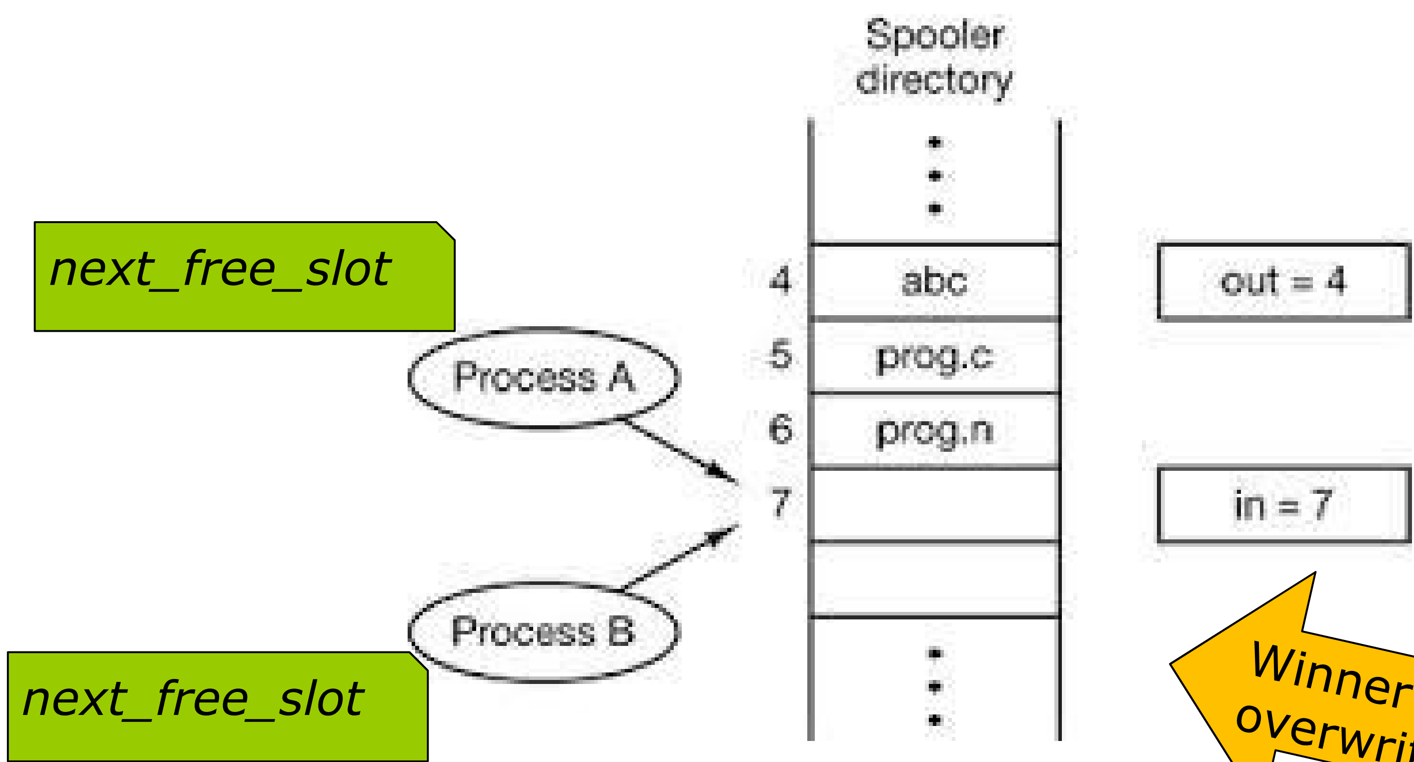
Reading 7.1-7.4, 7.6-7.7



Silberschatz, Galvin and Gagne ©2009

Race Conditions

- Multiple threads/processes attempt to access a shared resource
  - such as - a shared variable / a shared file,
- at least one of the accesses is an update, and the accesses can result in an error.



Winner's result will be overwritten by the loser.

Modern Operating System @Tanenbaum