USAL
جامعة العلوم والاداب اللبنانية
UNIVERSITY OF SCIENCES & ARTS IN LEBANON

CSCI320 – Operating Systems

Ahmad Fadlallah

References

- These slides are based on the official slides of the following textbooks
  - Operating Systems: Internals and Design Principles (9th Edition), William Stallings, Pearson
  - Operating System Concepts with Java (10th Edition), Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Wiley

USAL
جامعة العلوم والاداب اللبنانية
UNIVERSITY OF SCIENCES & ARTS IN LEBANON

Concurrency: Mutual Exclusion and Synchronization

Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors

Background

The Critical-Section Problem
Peterson's Solution
Synchronization Hardware
Mutex Locks
Semaphores
Classic Problems of Synchronization
Monitors

Multiple Processes

- Operating System design is concerned with the management of processes and threads:
  - **Multiprogramming**: multi-processes, uniprocessor
  - **Multiprocessing**: multi-processes, multi-processors
  - **Distributed Processing**: Multiprocessing on distributed computer systems

Muti*  Concurrency

Concurrency related-issues

Communication among processes

Sharing of and **Competing for** resources

Synchronization of the activities of multiple processes

Allocation of processor time to processes.

Concurrency arises in three different contexts

Multiple Applications	Structured Applications	Operating System Structure
• invented to allow processing time to be shared among active applications	• extension of modular design and structured programming • some applications can be effectively programmed as a set of concurrent processes	• OS themselves implemented as a set of processes or threads

Producer-Consumer Problem

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Producer-Consumer Problem - Illustration

- Consider a solution to the consumer-producer problem that fills all the buffers.
  - Use an integer counter that keeps track of the number of full buffers.
  - Counter initialized to 0
  - Incremented by the producer after it produces a new buffer
  - Decremented by the consumer after it consumes a buffer.

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

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

Producer

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
  - $register_1 = counter$
  - $register_1 = register_1 + 1$
  - $counter = register_1$
- counter-- could be implemented as
  - $register_2 = counter$
  - $register_2 = register_2 - 1$
  - $counter = register_2$
- Consider this execution interleaving with "count = 5" initially:
  - S0: producer execute $register_1 = counter$ {$register_1 = 5$}
  - S1: producer execute $register_1 = register1 + 1$ {$register_1 = 6$}
  - S2: consumer execute $register_2 = counter$ {$register_2 = 5$}
  - S3: consumer execute $register_2 = register_2 - 1$ {$register_2 = 4$}
  - S4: producer execute $counter = register_1$ {counter = 6 }
  - S5: consumer execute $counter = register_2$ {counter = 4}

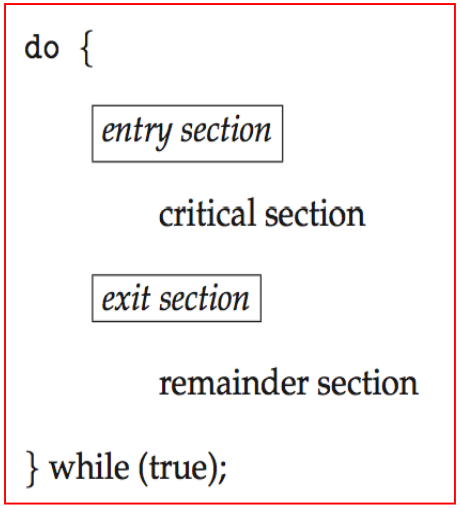
Background

The Critical-Section Problem

Peterson's Solution
Synchronization Hardware
Mutex Locks
Semaphores
Classic Problems of Synchronization
Monitors

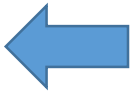
Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … 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 is in critical section**, no other may be in its critical section
- ***Critical section problem*** is to design a 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**

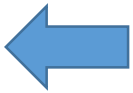


Algorithm for Process Pi

```
do {

    while (turn == j);

        critical section

    turn = j;

        remainder section

} while (true);
```

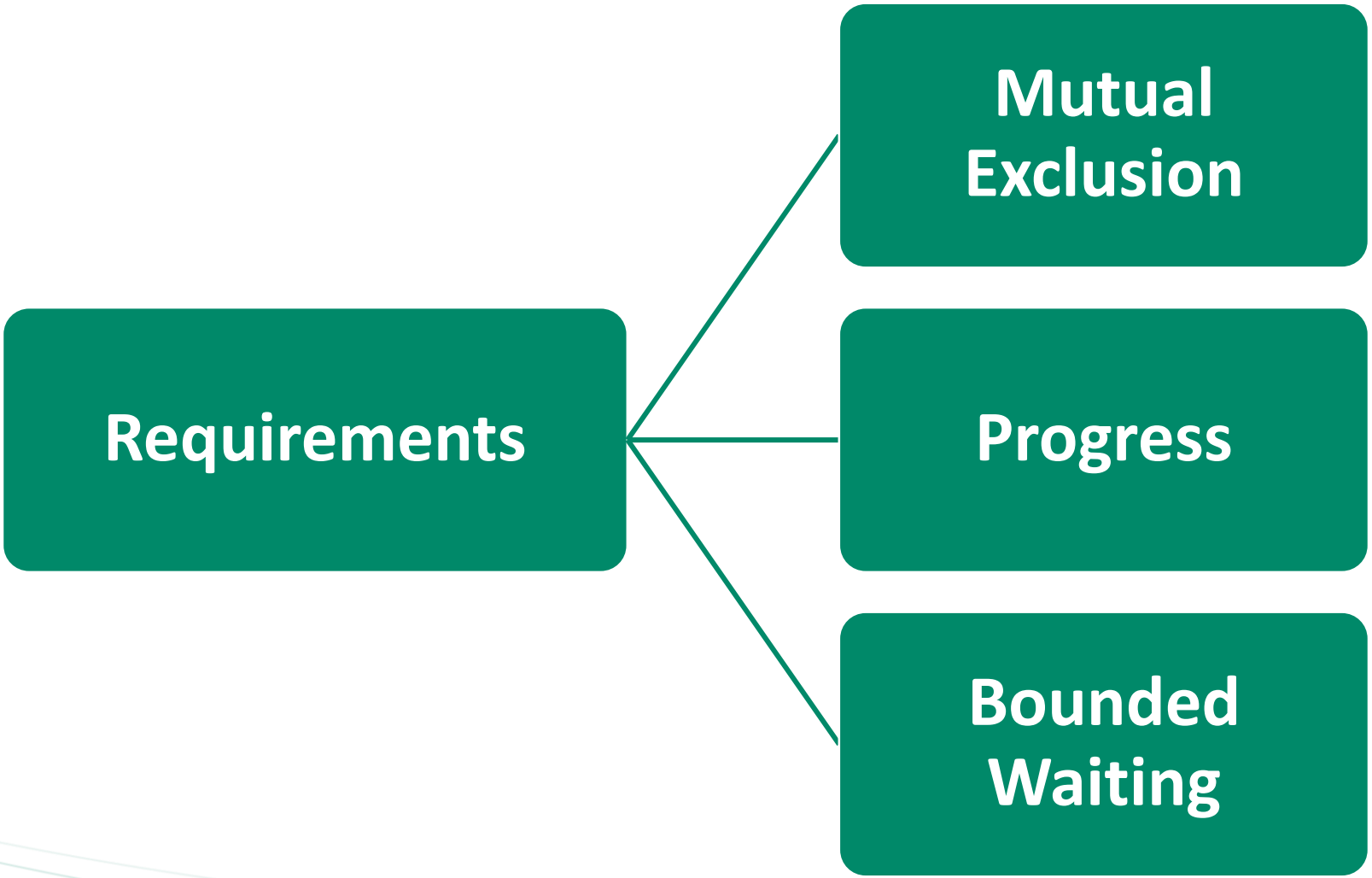


Entry section



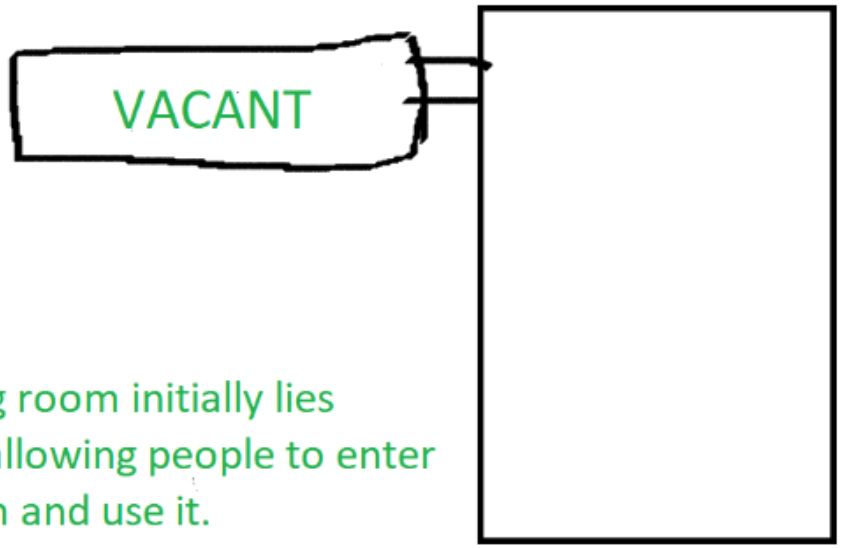
Exit section

Solution to Critical-Section Problem

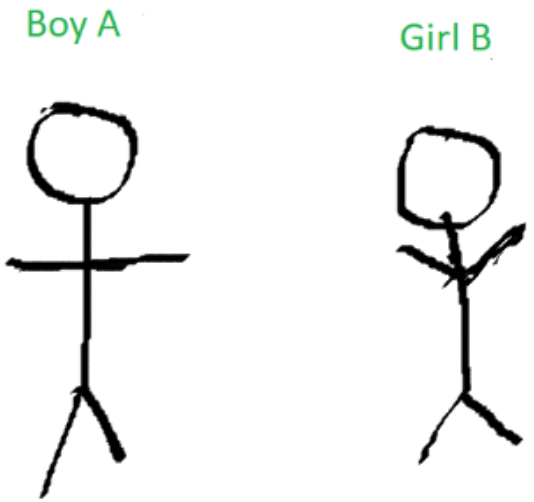


Solution to Critical-Section Problem - Requirements

- **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

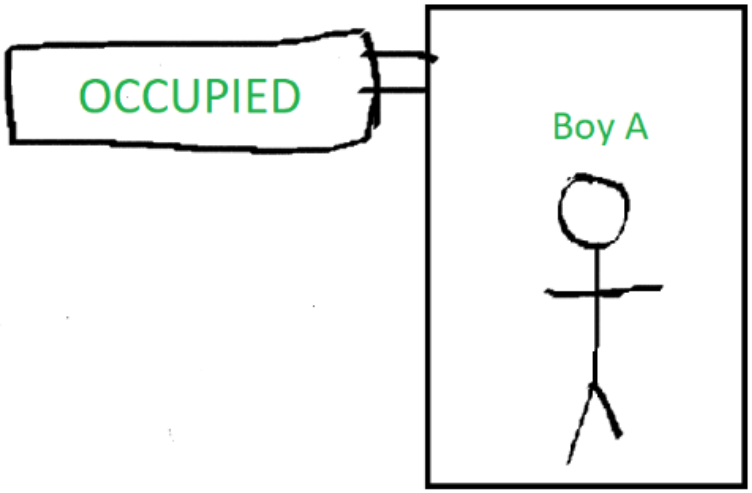


Changing room initially lies vacant, allowing people to enter the room and use it.



Solution to Critical-Section Problem - Requirements

- **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

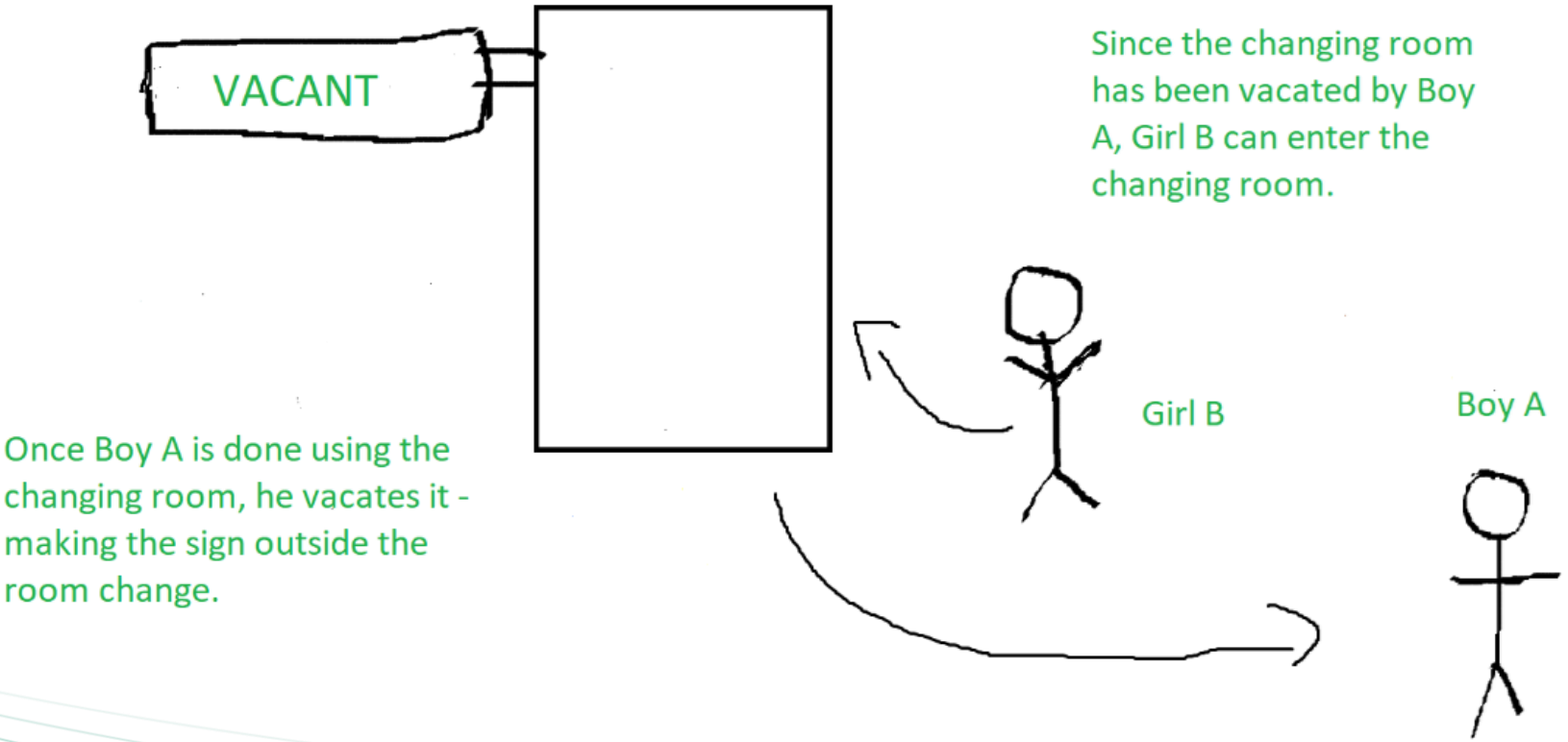


Since Boy A is inside the changing room, the sign on it prevents the others from entering the room.

Girl B has to wait outside the changing room till Boy A comes out.

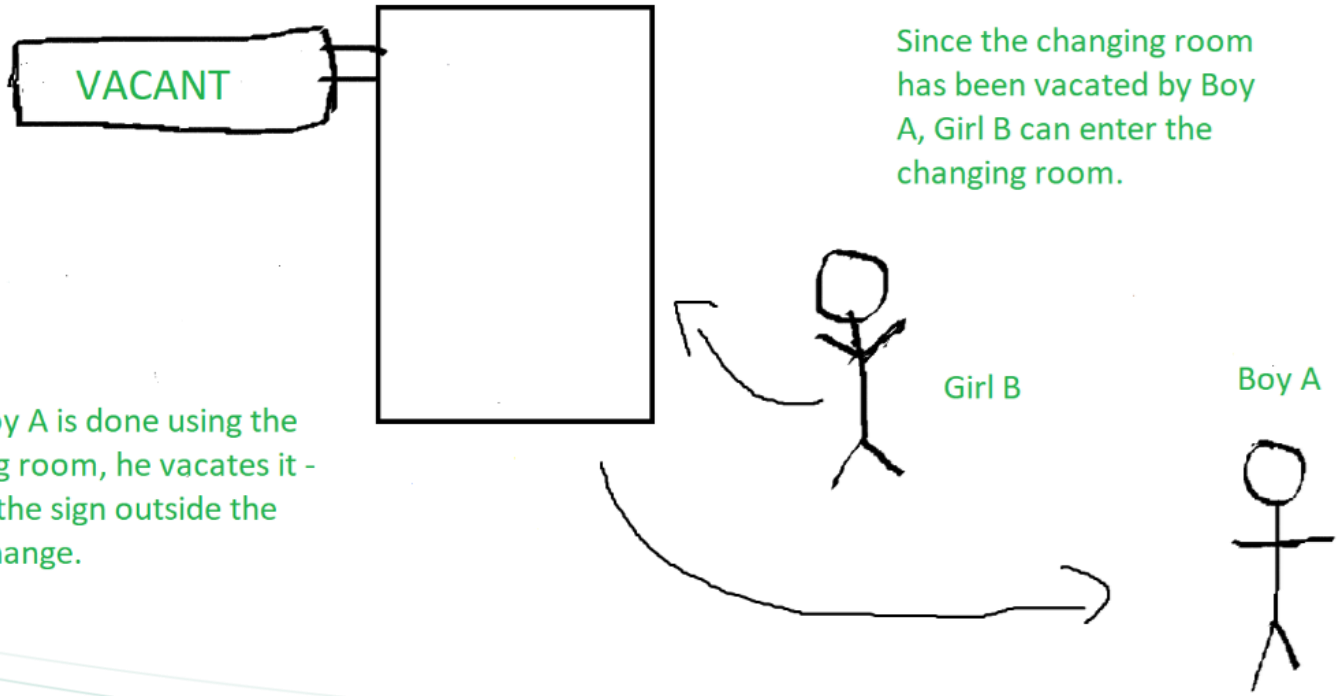
Solution to Critical-Section Problem - Requirements

- **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections



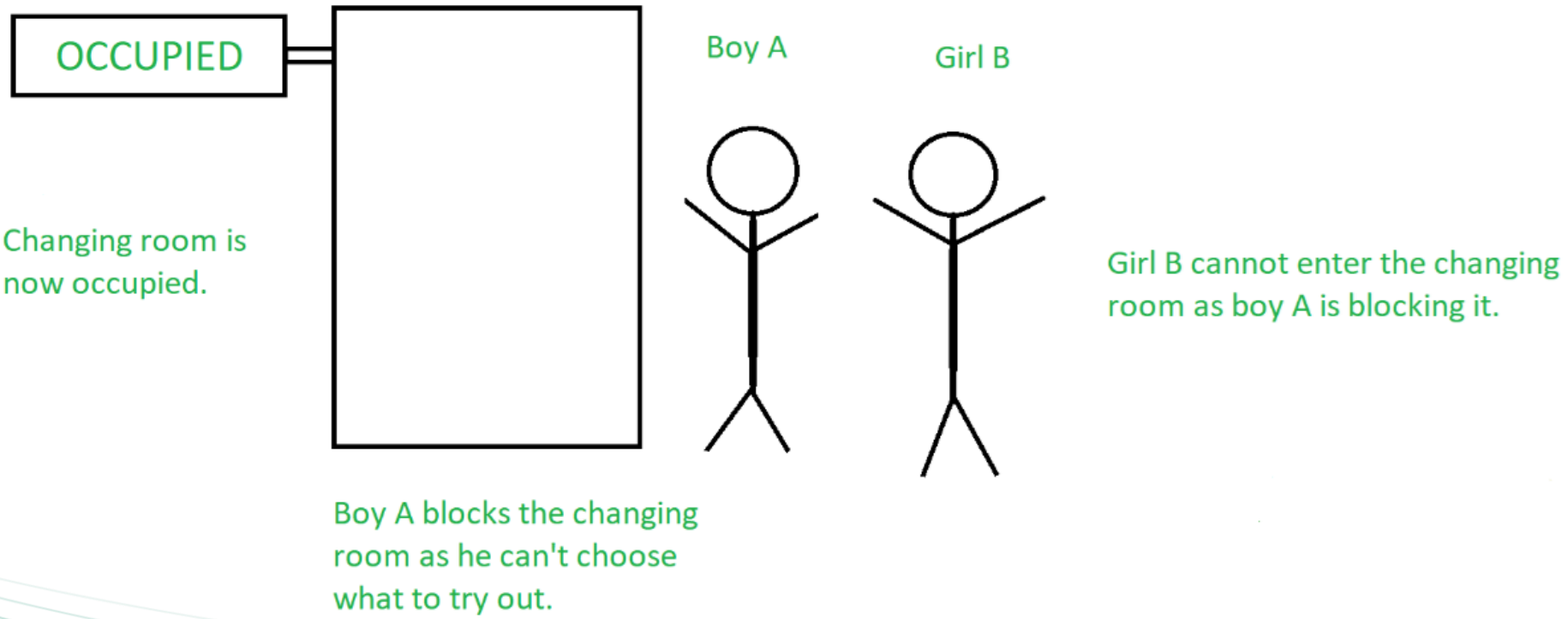
Solution to Critical-Section Problem - Requirements

- **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**



Solution to Critical-Section Problem - Requirements

- **Progress**: if a particular process do not want to enter into the critical section it must not delay other process to enter into the critical section



Solution to Critical-Section Problem - Requirements

- **Bounded Waiting:**
  - After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted.
  - So after the limit is reached, system must grant the process permission to get into its critical section.

Critical-Section Handling in OS

- Two approaches depending on if kernel is
  - **Preemptive**
    - Allows preemption of process when running in kernel mode
  - **Non-preemptive**
    - Runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of **race conditions** in kernel mode

Background
The Critical-Section Problem
Peterson's Solution
Synchronization Hardware
Mutex Locks
Semaphores
Classic Problems of Synchronization
Monitors

Peterson's Solution

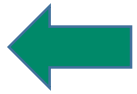
- Good algorithmic description of solving the problem
- **Two-process** solution
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted (?)
- The two processes share two variables:

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

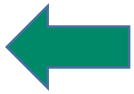
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section.
  - `flag[i]=true` implies that process $\mathbf{P_i}$ is ready!

Algorithm for Process P_i

```
do {
flag[i] = true;
turn = j;
while (flag[j] && turn == j);
   critical section
flag[i] = false;
   remainder section
 } while (true);
```



Entry section



Exit section

Process 0	Process 1
i=0, j=1	i = 1, j = 0

flag[0] := TRUE
turn := 1
check (flag[1] = TRUE and turn = 1)
- Condition is false because flag[1] = FALSE
- Since condition is false, no waiting in while loop
- Enter the critical section
- Process 0 happens to lose the processor
	do {
	flag[i] = true;
	turn = j;
	while (flag[j] && turn == j);
	critical section
	flag[i] = false;
	remainder section
	} while (true);

	flag[1] := TRUE **turn := 0** **check (flag[0] = TRUE and turn = 0)** Since condition is true, it keeps busy waiting until it loses the processor
- Process 0 resumes and continues until it finishes in the critical section - Leave critical section **flag[0] := FALSE** - Start executing the remainder (anything else a process does besides using the critical section) - Process 0 happens to lose the processor	
	check (flag[0] = TRUE and turn = 0) This condition fails because flag[0] = FALSE - No more busy waiting - Enter the critical section

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either `flag[j] = false` or `turn = i`
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Background
The Critical-Section Problem
Peterson's Solution
Synchronization Hardware
Mutex Locks
Semaphores
Classic Problems of Synchronization
Monitors

Synchronization Hardware

- Many systems provide **hardware support** for implementing the critical section code.
- Based on the idea of **locking**
  - Protecting critical regions via **locks**
- **Uniprocessors**: could disable interrupts
  - Currently running code would execute without *preemption*
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special **atomic hardware instructions**
  - **Atomic** = non-interruptible
  - Either *test memory word and set value*
  - Or *swap contents of two memory words*

Solution to Critical-section Problem Using Locks

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (TRUE);
```

`test_and_set` Instruction

Definition:

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = TRUE;
        return rv:
}
```

1. Executed **atomically**
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
          while (test_and_set(&lock)) //acquire lock
             ; /* do nothing */
                  /* critical section */
          lock = false; //release lock
                  /* remainder section */
       } while (true);
```

```
boolean test_and_set (boolean *target)
          {
               boolean rv = *target;
               *target = TRUE;
               return rv:
          }
```

compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {
        int temp = *value;
        if (*value == expected)
           *value = new_value;
     return temp;
    }
```

1. Executed atomically
2. Returns the original value of passed parameter "value"
3. Set "value" to the value of the passed parameter "new_value" but only if "value" =="expected" ➔ the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer "lock" initialized to 0
- Solution:

```
do {
        while (compare_and_swap(&lock, 0, 1) != 0) //acquire lock
           ; /* do nothing */
         /* critical section */
       lock = 0;  //release lock
         /* remainder section */
     } while (true);
```

```
int compare _and_swap(int *value, int expected, int new_value) {
        int temp = *value;
        if (*value == expected)
           *value = new_value;
     return temp;
    }
```

Background
The Critical-Section Problem
Peterson's Solution
Synchronization Hardware

Mutex Locks

Semaphores
Classic Problems of Synchronization
Monitors

Mutex Locks

- Previous solutions are complicated and generally **inaccessible to application programmers**
- OS designers build **software tools** to solve critical section problem
- Simplest is **mutex** lock
  - Mutex => Mutual Exclusion Object
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions

acquire() and release()

```
acquire() {
      while (!available)
         ; /* busy wait */
      available = false;;
   }
```

```
release() {
        available = true;
    }
```

```
do {
    acquire lock
       critical section
    release lock
      remainder section
 } while (true);
```

Issue: Solution requires **busy waiting**

This lock therefore called a **spinlock** (problem for uniprocessor systems)

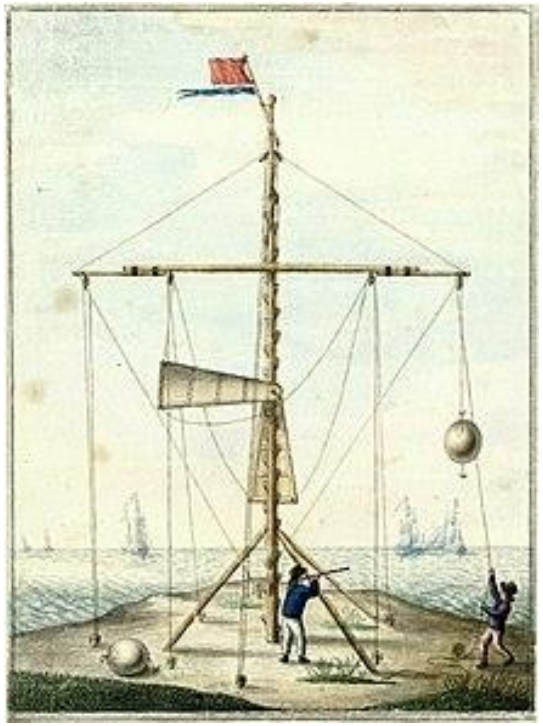
Background
The Critical-Section Problem
Peterson's Solution
Synchronization Hardware
Mutex Locks

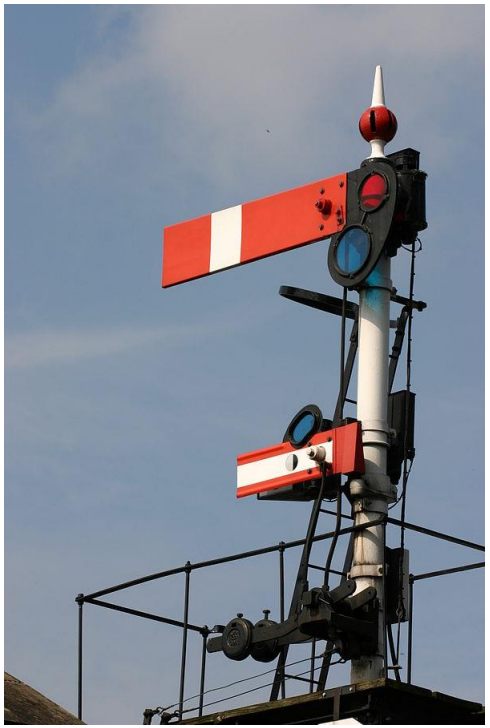
Semaphores

Classic Problems of Synchronization
Monitors

Semaphore

- **Synchronization tool** that provides more sophisticated ways (than **Mutex** locks) for process to synchronize their activities.





Semaphore

- Semaphore S : **Integer variable**
- Can only be accessed via two **indivisible** (atomic) operations
  - `wait()` and `signal()`
    - Originally called **`P()`** and **`V()`**
      - **Dutch words: test (proberen) and increment (verhogen)**

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

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

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
- Example: Consider P_1 and P_2 that require S_1 to happen before S_2

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

P1:	P2:
S_1; signal(synch);	wait(synch); S_2;

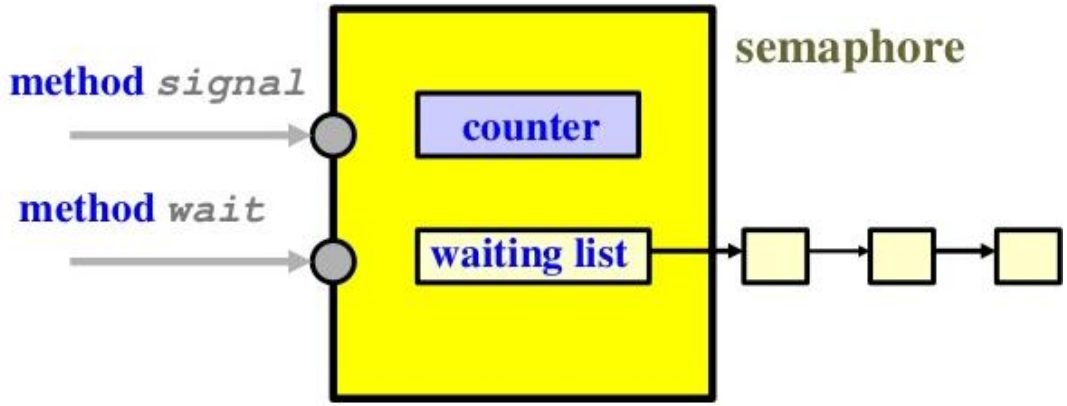
- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- The implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
- **Issue**: **Busy waiting** in critical section implementation
- **Solution**: rather than engaging in busy waiting, the process can block itself
  - The block operation places a process into a **waiting queue** associated with the semaphore, and the state of the process is switched to the **waiting state**

Semaphore Implementation with no Busy waiting

- **With each semaphore** there is an associated **waiting queue**
  - Any queuing mechanism can be used




```
typedef struct{
   int value;
   struct process *list;
   } semaphore;
```

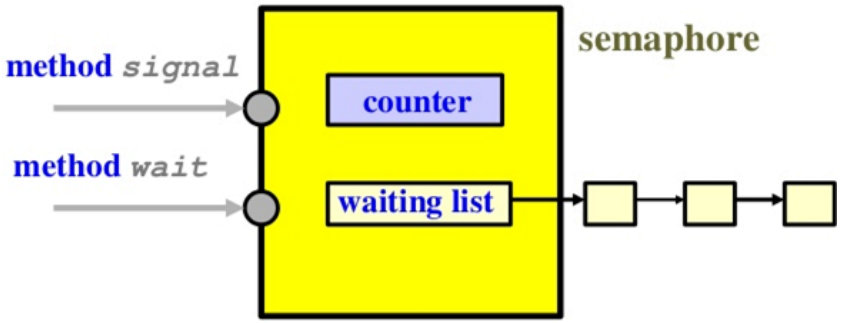
Two operations (provided by the OS as system calls)

- **block** : place the process invoking the operation on the appropriate waiting queue
- **wakeup** : remove one of processes in the waiting queue and place it in the ready queue

Implementation with no Busy waiting (Cont.)

Demo

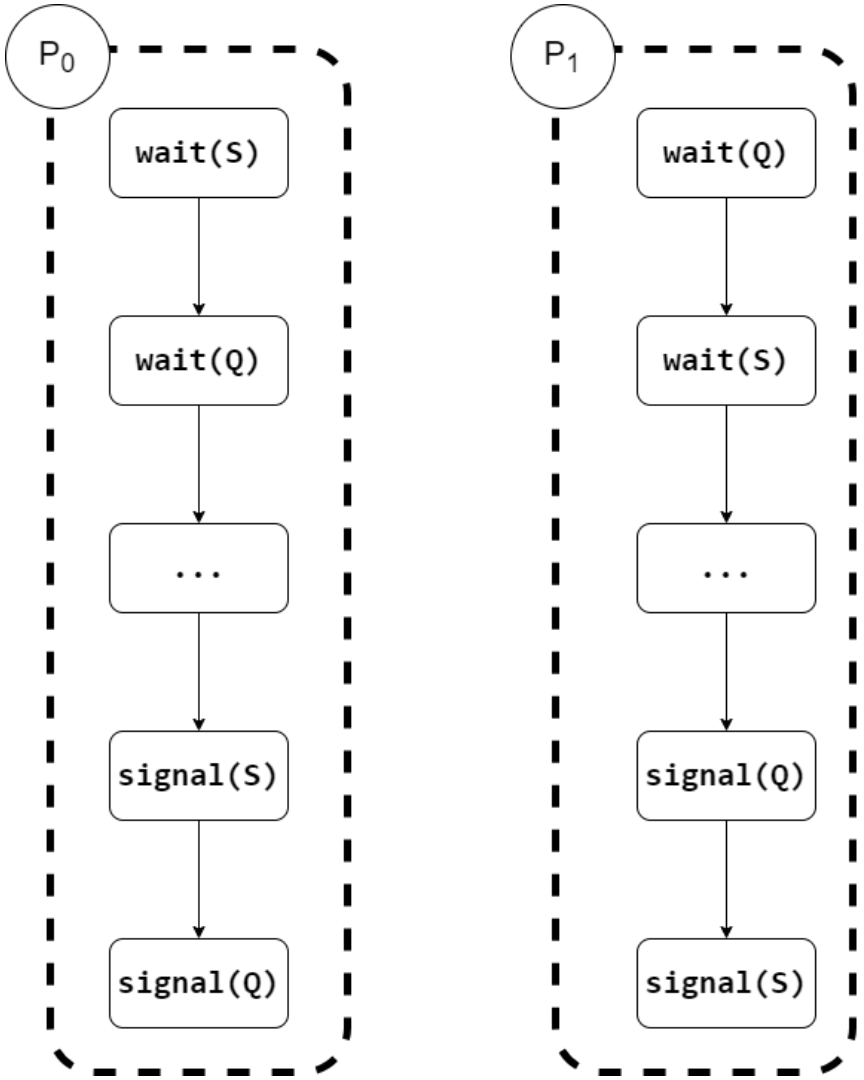
```
wait(semaphore *S) {
    S->counter--;
    if (S->counter < 0) {
     // add this process to S->list;
        block();
    }
}
```




```
signal(semaphore *S) {
    S->counter++;
    if (S->counter <= 0) {
        //remove a process P from S->list;
        wakeup(P);
    }
}
```

Deadlock

- Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Example: Let S and Q be two semaphores initialized to 1

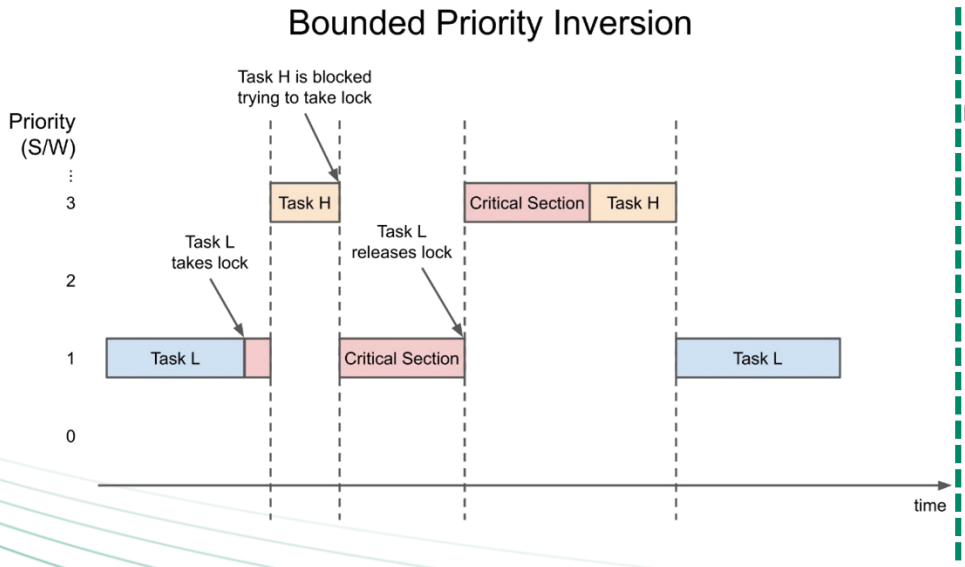


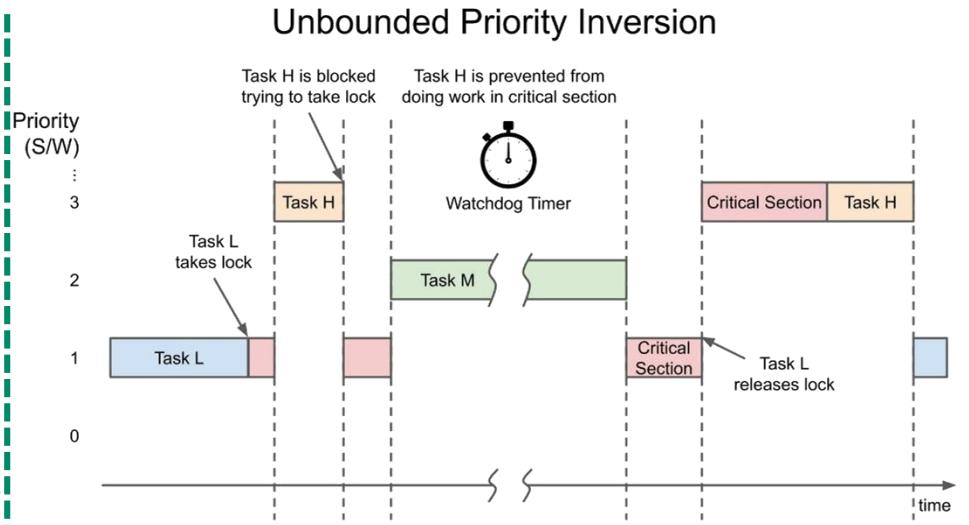
Starvation

- **Starvation: indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended (example LIFO queue used in Semaphore)
  - occurs when high priority processes keep executing and low priority processes get blocked for indefinite time.

Priority Inversion

- Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via priority-inheritance protocol
  - All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.





Background
The Critical-Section Problem
Peterson's Solution
Synchronization Hardware
Mutex Locks
Semaphores

Classic Problems of Synchronization

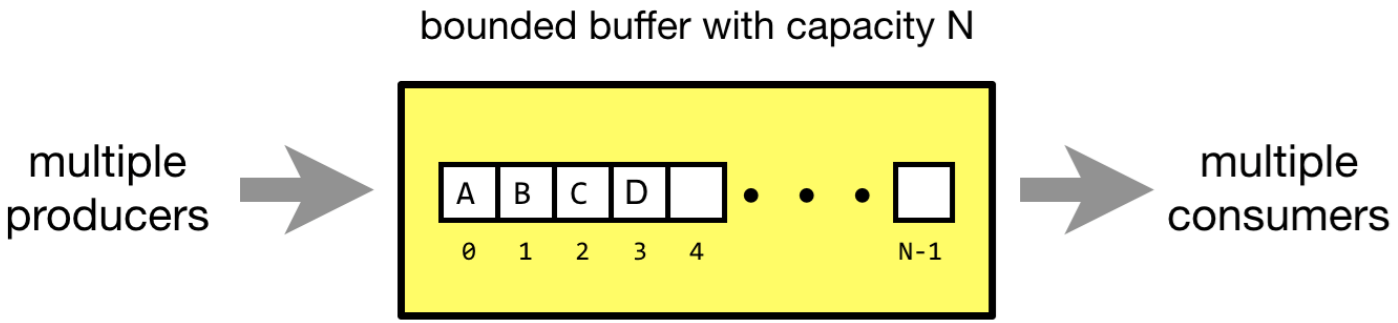
Monitors

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - **Bounded-Buffer Problem**
  - **Readers and Writers Problem**
  - **Dining-Philosophers Problem**

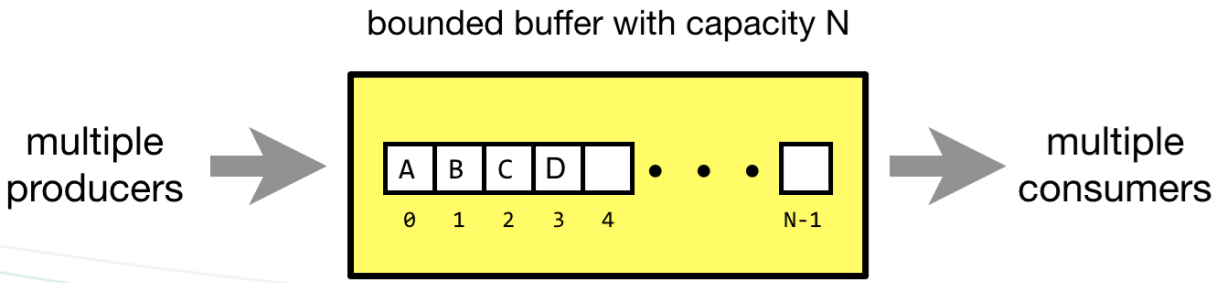
Bounded-Buffer Problem

- Also known as the **Producer-Consumer** problem
- A classic example of concurrent access to a shared resource
- A bounded buffer lets **multiple producers** and **multiple consumers** share a single buffer.
- **Producers write data** to the buffer and **consumers read data** from the buffer.



Bounded-Buffer Problem - Synchronization

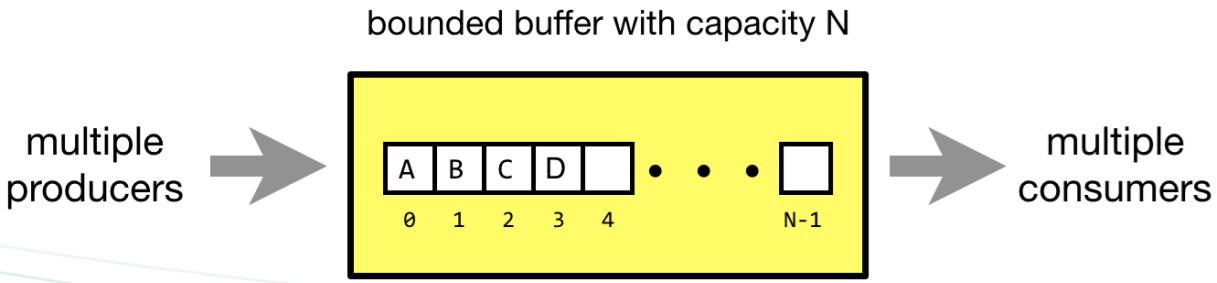
- Results of Improper synchronization
  - The producers do not block when the buffer is full
  - A Consumer consumes an empty slot in the buffer
  - A consumer attempts to consume a slot that is only half-filled by a producer.
  - Two producers write into the same slot.
  - Two consumers read the same slot.
  - And possibly more …



Bounded-Buffer Problem - Synchronization

- Results of Improper synchronization
  - The producers doesn't block when the buffer is full
  - And possibly more …

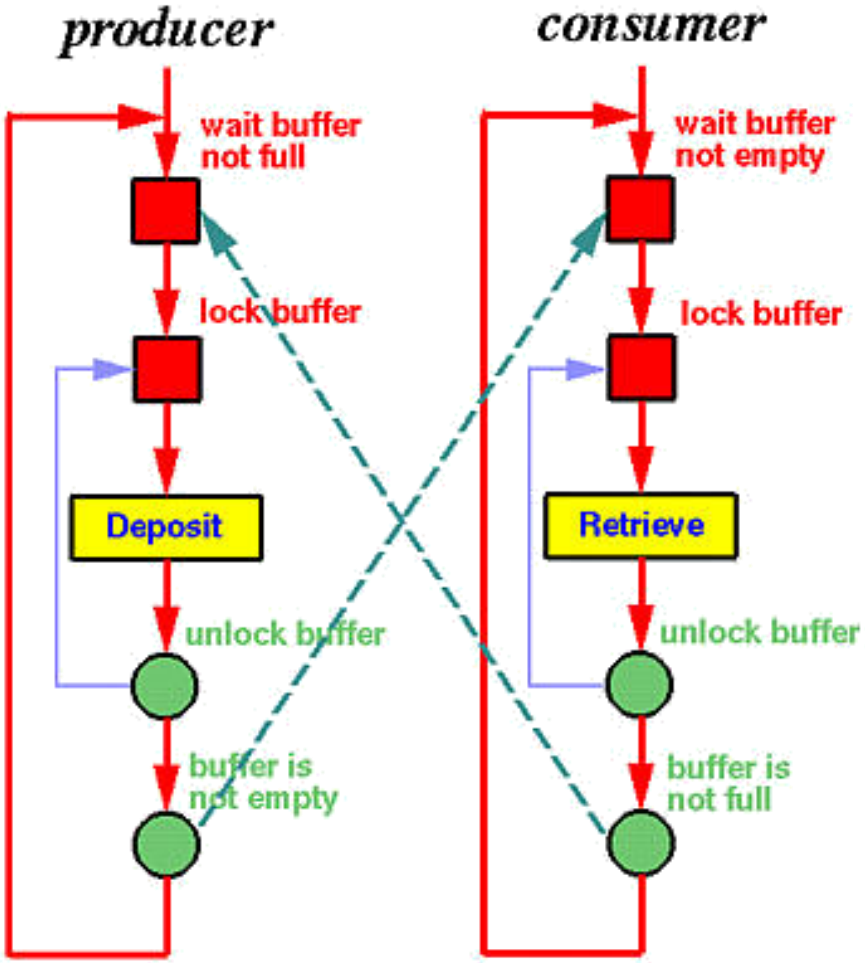
- Producers must block if the buffer is full
- Consumers must block if the buffer is empty



Bounded-Buffer Problem – Semaphore Solution

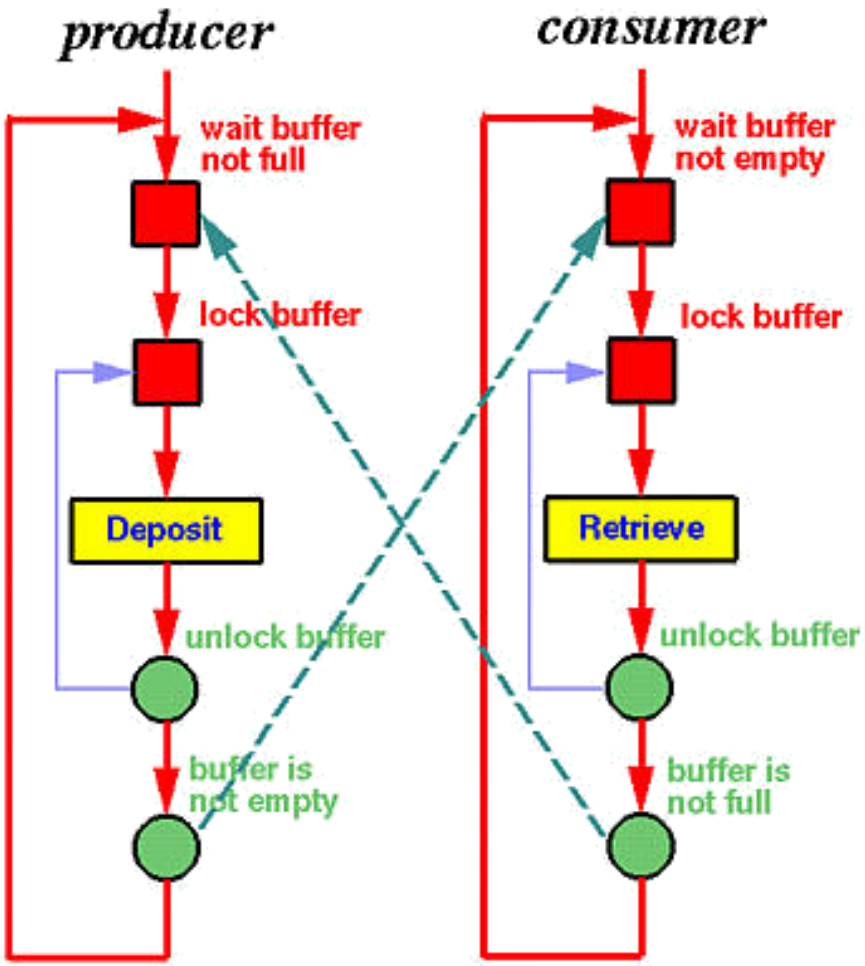
- **N** buffers, each can hold one item
- Semaphore `mutex`
  - Provides mutual exclusion for accesses to the buffer pool
  - Used to produce (add item) or consume (remove item)
  - Initialized to the value **1**
- Semaphore `full`
  - Counts the number of full buffers
  - Initialized to the value **0**
- Semaphore `empty`
  - Counts the number of empty buffers
  - initialized to the value **n**

Bounded-Buffer Problem – Semaphore Solution



Bounded-Buffer Problem – Semaphore Solution

```
do {
/* produce an item in
next_produced */
          ...
wait(empty);


wait(mutex);
     ...


/* add next produced
to the buffer */
       ...

signal(mutex);


signal(full);

} while (true);
```




```
Do {

wait(full);


wait(mutex);
            ...

/* remove an item
from buffer to
next_consumed */
 ...

signal(mutex);


signal(empty);
...
/* consume the item
in next consumed */
            ...
} while (true);
```

Readers-Writers Problem - Introduction

- A data set is shared among a number of concurrent processes
  - Readers: only read the data set; they do ***not*** perform any updates
  - Writers: can both read and write

Readers-Writers Problem - Introduction

- **Problem**
  - Allow **multiple readers to read at the same time**
  - Only **one single writer can access the shared data at the same time**

Readers-Writers Problem Variations

- ***First*** variation: no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation: once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks (in read or write mode)

Readers-Writers Problem – Shared Data

Data set

Semaphore **rw_mutex**

- For readers and writers
- Initialized to 1

Integer **read_count**

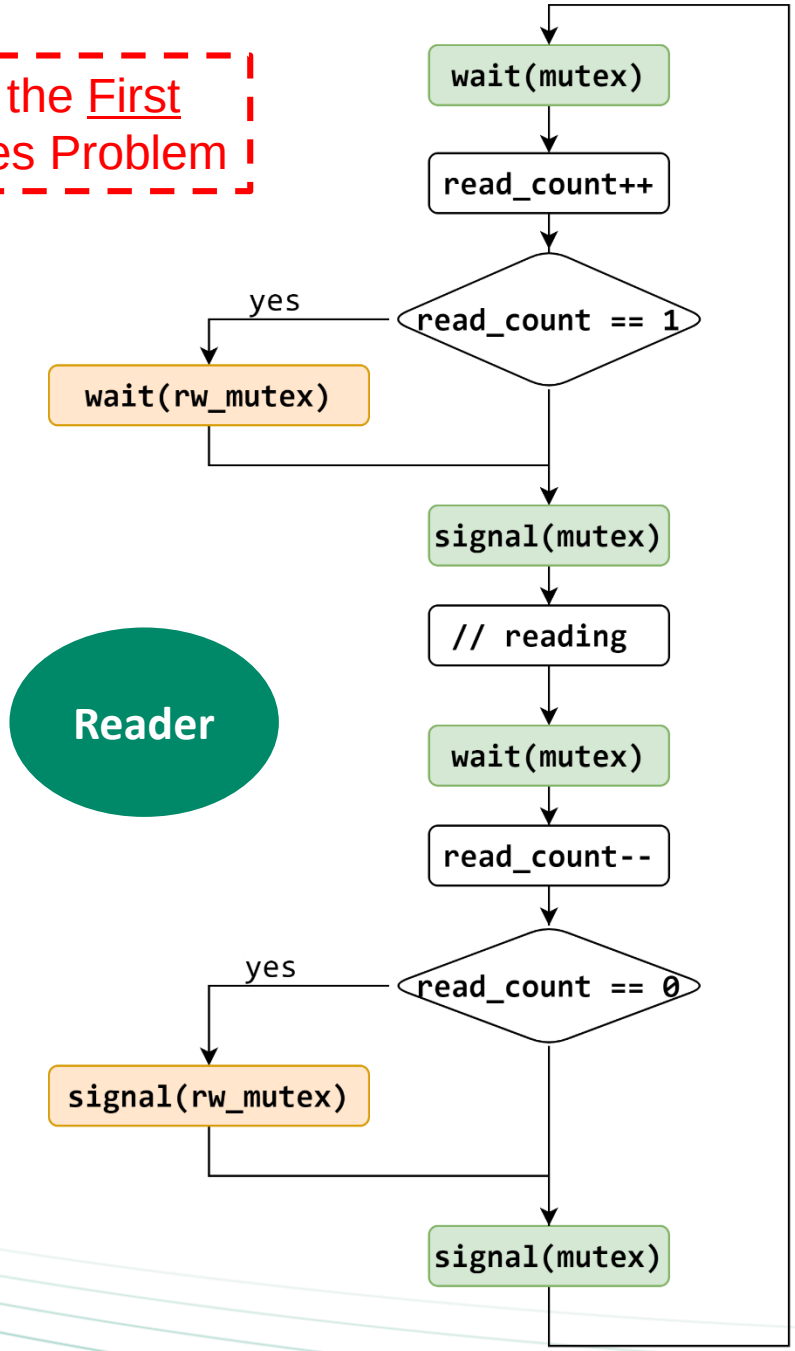
- Keeps track of how many processes are currently reading the object
- Initialized to 0

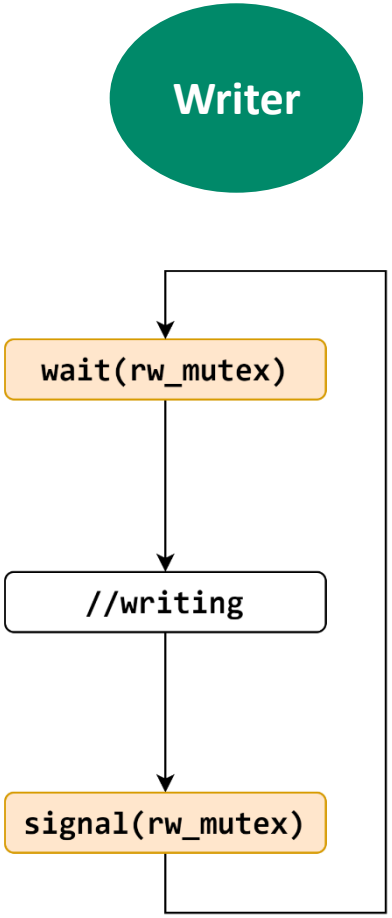
Semaphore **mutex**

- Used to ensure mutual exclusion when the variable **read_count** is updated
- Initialized to 1

Readers-Writers Problem

Solution for the First Readers-Writes Problem

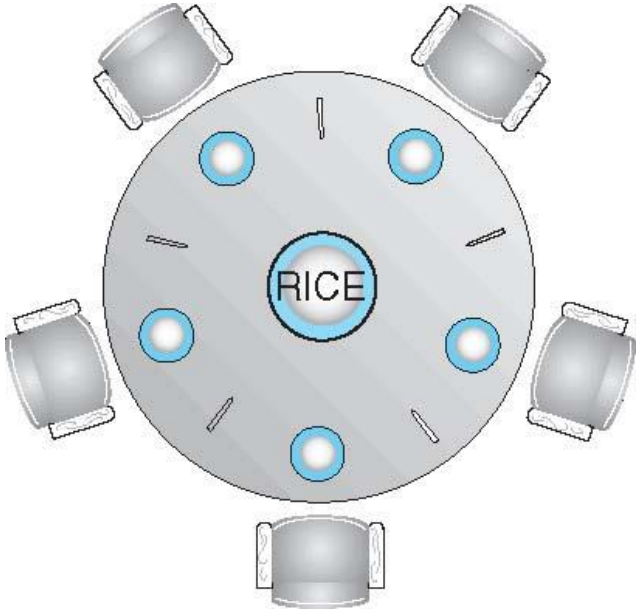




- **Semaphore rw_mutex**: For readers and writers, Initialized to 1
- **Integer read_count**: Keeps track of how many processes are currently reading the object, Initialized to 0
- **Semaphore mutex**: Used to ensure mutual exclusion when the variable *read_count* is updated, Initialized to 1

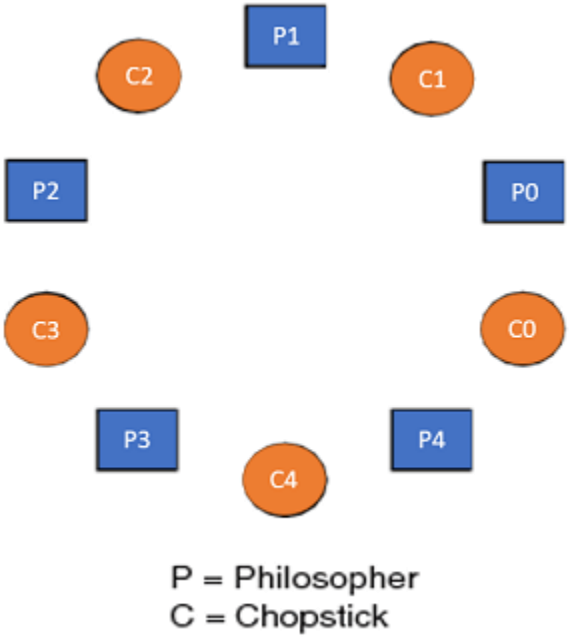
Dining-Philosophers Problem

- Philosophers spend their lives alternating **thinking** and **eating**
- Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
- Need both to eat, then release both when done



Dining-Philosophers Problem Algorithm

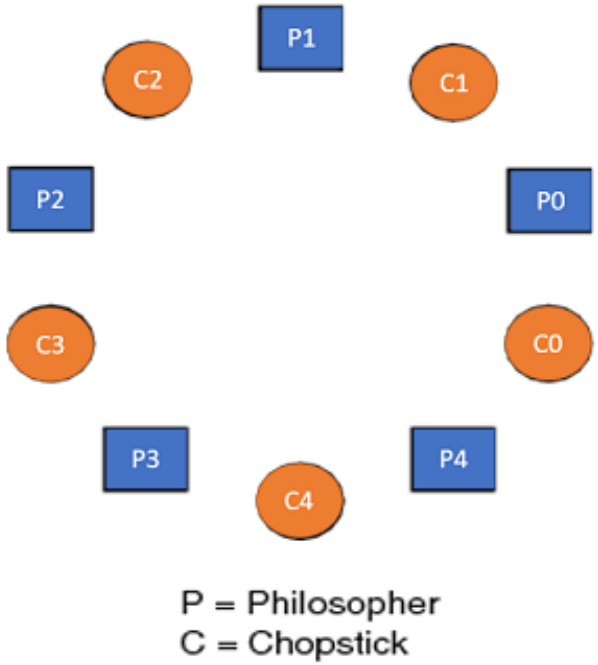
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1




```
do {
 wait (chopstick[i]);
 wait (chopStick[(i + 1) %5]);

       //  eat

signal (chopstick[i] );
signal (chopstick[ (i + 1) % 5]);

       //  think

} while (TRUE);
```

Dining-Philosophers Problem Algorithm

Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
All the elements of chopstick will now be equal to 0.
When each philosopher tries to grab her right chopstick, she will be delayed forever. (**Deadlock**)




```
do {
 wait (chopstick[i]);
 wait (chopStick[(i + 1) %5]);

       //  eat

signal (chopstick[i] );
signal (chopstick[ (i + 1) % 5]);

       //  think

} while (TRUE);
```

Dining-Philosophers Problem Algorithm (Cont.)

- **Deadlock handling**
  - Allow at most 4 philosophers to be sitting simultaneously at the table (with 5 Chopsticks).
  - Allow a philosopher to pick up the chopsticks only if both are available
  - Use an asymmetric solution:
    - An odd-numbered philosopher picks up first the left chopstick and then the right chopstick.
    - Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal (mutex)  …  wait (mutex)`
  - `wait (mutex)  …  wait (mutex)`
  - Omitting `wait (mutex)` or `signal (mutex)` (or both)
- Deadlock and starvation are possible.

Background
The Critical-Section Problem
Peterson's Solution
Synchronization Hardware
Mutex Locks
Semaphores
Classic Problems of Synchronization

Monitors

Why Monitors

- **Semaphores can be very useful for solving concurrency** problems, but only if programmers use them properly.
  - If even one process fails to abide by the proper use of semaphores, either accidentally or deliberately, then the whole system breaks down.
  - Concurrency problems are by definition rare events => the problem code may easily go unnoticed
- **A higher-level language construct** has been developed:

The Monitors

Monitors

- Thread-safe class, object, or module that uses wrapped mutual exclusion in order to safely allow access to a method or variable by more than one thread.
- The defining characteristic of a monitor is that its methods are executed with mutual exclusion
  - At each point in time, at most one thread may be executing any of its methods.
- By using one or more **condition variables** it can also provide the ability for threads to wait on a certain condition (thus using the above definition of a "monitor").

(Source:Wikipedia)

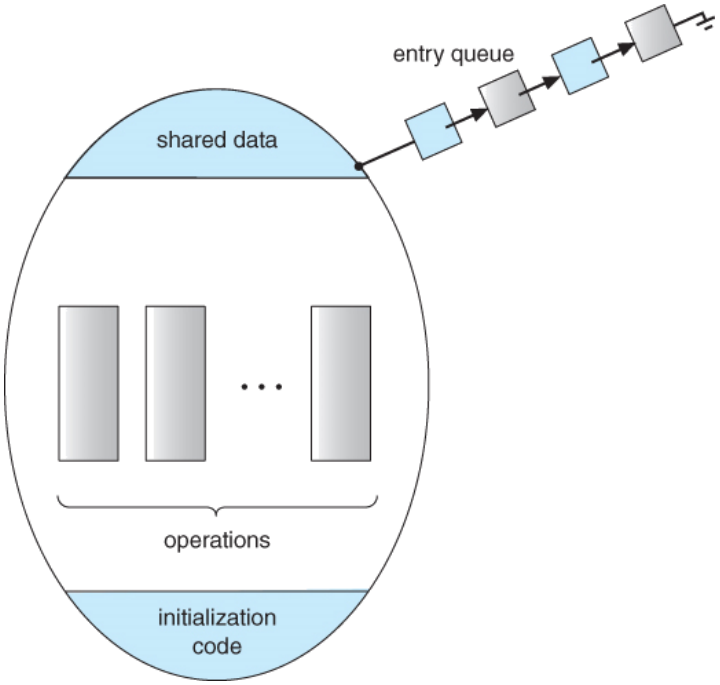
Monitors

- A **monitor** is a **software module** consisting of one or more procedures, an initialization sequence, and local data.

```
monitor monitor-name
{
      // shared variable declarations
      procedure P1 (…) { … }

      procedure Pn (…) {…}

     Initialization code (…) { … }
      }
}
```

Monitor Components

- The ***initialization*** component contains the code that is used exactly once when the monitor is created
- The ***private data*** section contains all private data, including private procedures, that can only be used *within* the monitor.
  - Not visible from outside of the monitor.
- The ***monitor procedures*** are procedures that can be called from outside of the monitor.
- The ***monitor entry queue*** contains all threads that called monitor procedures but have not been granted permissions.

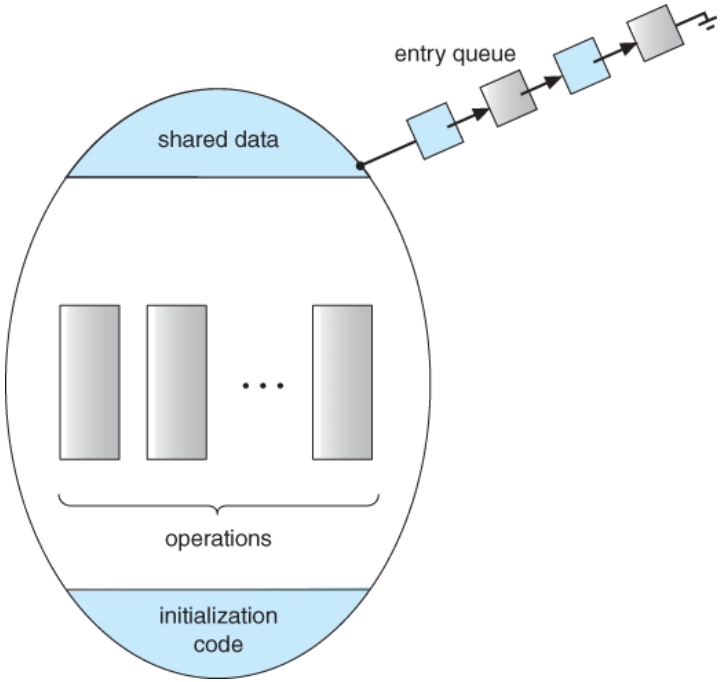


Monitors Characteristics

- The local data variables are accessible only by the monitor's procedures and not by any external procedure.
- A process enters the monitor **by invoking one of its procedures**.
- Only one process may be executing in the monitor at a time
  - Any other processes that have invoked the monitor are **blocked**, waiting for the monitor to become available.

Monitor Entry Queue

- **if a thread calls a monitor procedure**, this thread will be blocked if there is another thread executing in the monitor.
- Those **threads that were not granted** the entering permission will be **queued** to a **monitor entry queue** outside of the monitor.
- When the monitor becomes empty (i.e., no thread is executing in it), one of the threads in the entry queue will be released and granted the permission to execute the called monitor procedure.



Monitors

- To be useful for concurrent processing, the monitor must include synchronization tools.
  - Suppose a process invokes the monitor and, while in the monitor, must be blocked until some condition is satisfied.
    - A facility is needed by which the process is not only blocked but releases the monitor so that some other process may enter it.
    - When the condition is satisfied and the monitor is again available, the process needs to be resumed and allowed to reenter the monitor at the point of its suspension.
- A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor.

Condition Variables

- A **condition variable** essentially is a container of threads that are waiting for a certain condition.
- **Operations on condition variables**

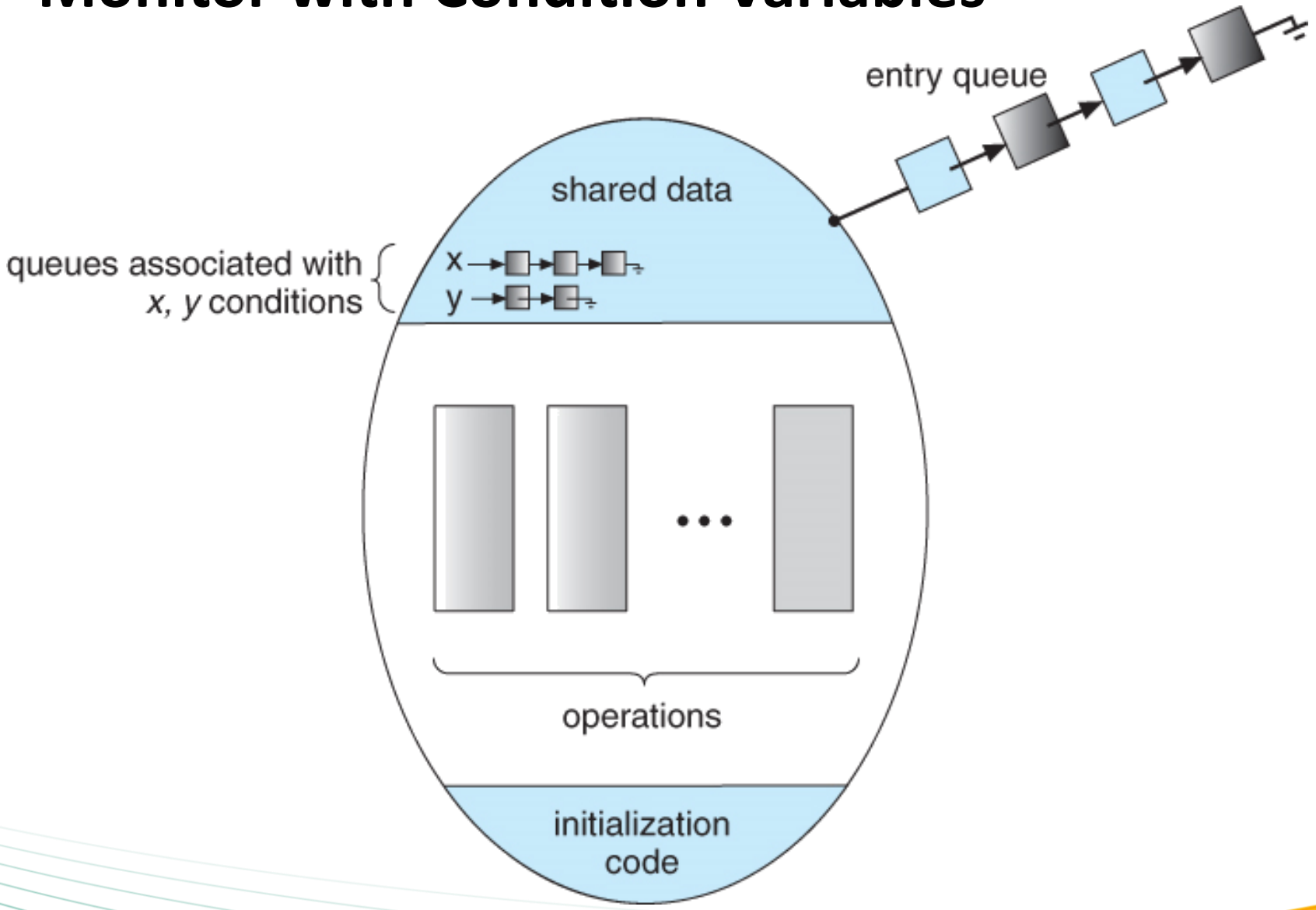
`x.wait()`

- a process that invokes the operation is suspended until **`x.signal()`**

`x.signal()`

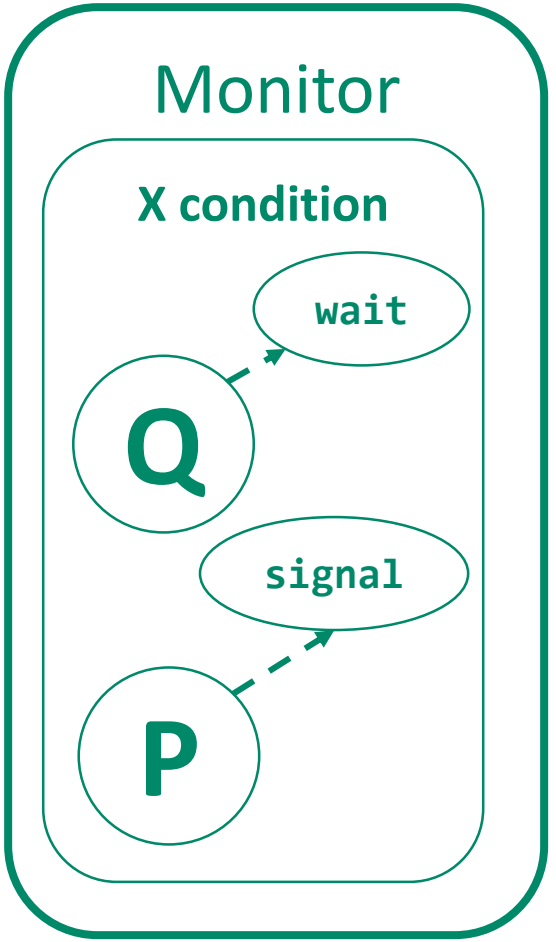
- Resumes one of processes (if any) that invoked **`x.wait()`**
- If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process **P within the monitor** issues a **`signal`** that would wake up **process Q also within the monitor**, then there would be two processes running simultaneously within the monitor, violating the exclusion requirement.
- Solutions:
  - **Signal and wait:** When process P issues the signal to wake up process Q, P then waits, either for Q to leave the monitor or on some other condition.
  - **Signal and continue:** When P issues the signal, Q waits, either for P to exit the monitor or for some other condition.
  - **Concurrent Pascal approach:** The signal call causes the signaling process to immediately exit the monitor, so that the waiting process can then wake up and proceed.



Monitor Solution to Dining Philosophers

- Restriction: a philosopher may only pick up chopsticks when both are available.
- Two key data structures :

```
enum { THINKING, HUNGRY,EATING} state[5]
```

A philosopher may only set their state to eating when hungry and neither of their adjacent neighbors is eating.

```
(state[ ( i + 1 ) % 5 ] != EATING &&
        state[i] == HUNGRY) &&
  state[ ( i + 4 ) % 5 ] != EATING )
```

```
condition self[ 5 ]
```

This condition is used to delay a hungry philosopher who is unable to acquire chopsticks.

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{ //start
enum { THINKING; HUNGRY, EATING) state [5] ;
condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING)
                 self[i].wait;
              }

void putdown (int i) {
     state[i] = THINKING;
      // test left and right neighbors
      test((i + 4) % 5);
      test((i + 1) % 5);
}
```

```
void test (int i) {
if ((state[(i + 4) % 5] != EATING) &&
          (state[i] == HUNGRY) &&
          (state[(i + 1) % 5] != EATING))
     {
          state[i] = EATING ;
          self[i].signal () ;
     }
   }

initialization_code() {
          for (int i = 0; i < 5; i++)
                 state[i] = THINKING;
           }
} // end
```

Solution to Dining Philosophers (Cont.)

- Each philosopher i invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup(i); // Acquire chopsticks but may
                              // block the process
EAT
DiningPhilosophers.putdown(i); // Release Chopsticks
```

- No deadlock, but starvation is possible

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- First Come First Served (FCFS)
- Assign (integer) **priorities**, and to **wake up the process with the smallest (best) priority**.
  - **conditional-wait** construct of the form `x.wait(c)`
    - Where c is **priority number**
    - Process with lowest number (highest priority) is scheduled next

Additional Exercises

Multiple Choice Questions

- In a concurrent program, it may happen that one thread is indefinitely delayed because other threads are always given preference. This undesirable scenario is called:
  a. Busy-waiting
  b. Deadlock
  c. Mutual exclusion
  d. Starvation

Multiple Choice Questions - Solution

- In a concurrent program, it may happen that one thread is indefinitely delayed because other threads are always given preference. This undesirable scenario is called:

a. Busy-waiting
b. Deadlock
c. Mutual exclusion
d. Starvation

Multiple Choice Questions

- Consider the program below where two concurrent threads A and B need to access a critical section. Instead of using a classical mutex lock, the authors choose to implement their own synchronization with two shared variables wa and wb (both initialized to false). Unfortunately, this approach does not work. But for what reason(s)?

a. This implementation does not guarantee mutual exclusion.

b. This implementation does not guarantee progress (i.e., bounded waiting).

c. This implementation requires that threads enter the critical section in strict alternation.

d. This implementation guarantees mutual exclusion but does not prevent deadlocks.

Thread A	Thread B

while(true)
{
wa = true;
while(wb == true) ; // busy wait
// Critical section
wa = false;
}
	while(true)
	{
	wb = true;
	while(wa == true) ; // busy wait
	// Critical section
	wb = false;
	}

Multiple Choice Questions - Solution

- Consider the program below where two concurrent threads A and B need to access a critical section. Instead of using a classical mutex lock, the authors choose to implement their own synchronization with two shared variables wa and wb (both initialized to false). Unfortunately, this approach does not work. But for what reason(s)?

a. This implementation does not guarantee mutual exclusion.

b. This implementation does not guarantee progress (i.e., bounded waiting).

c. This implementation requires that threads enter the critical section in strict alternation.

d. This implementation guarantees mutual exclusion but does not prevent deadlocks.

Thread A	Thread B

while(true)
{
wa = true;
while(wb == true) ; // busy wait
// Critical section
wa = false;
}
	while(true)
	{
	wb = true;
	while(wa == true) ; // busy wait
	// Critical section
	wb = false;
	}

Short Questions

- If the semaphore operations Wait and Signal are not executed atomically, then mutual exclusion may be violated. Assume that Wait and Signal are implemented as below:
- Describe a scenario of context switches where two threads, T1 and T2, can both enter a critical section guarded by a single mutex semaphore as a result of a lack of atomicity.

```
void Wait (Semaphore S) {
        while (S.count <= 0) {}
        S.count = S.count - 1;
}
```

```
void Signal (Semaphore S) {
        S.count = S.count + 1;
}
```

Short Questions - Solution

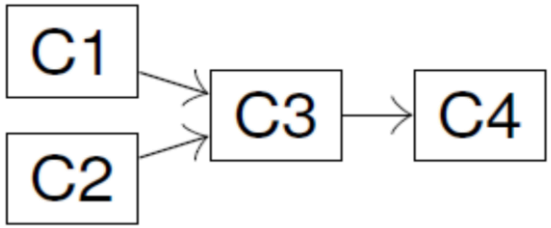
- Describe a scenario of context switches where two threads, T1 and T2, can both enter a critical section guarded by a single mutex semaphore as a result of a lack of atomicity.

void Wait (Semaphore S) {
while (S.count <= 0) {}
S.count = S.count - 1;
}
	void Signal (Semaphore S) {
	S.count = S.count + 1;
	}

- Assume that the semaphore is initialized with count = 1.
- T1 calls Wait, executes the while loop, and breaks out because count is positive.
- Then a context switch occurs to T2 before T1 can decrement count.
- T2 also calls Wait, executes the while loop, decrements count, and returns and enters the critical section.
- Another context switch occurs, T1 decrements count, and also enters the critical section. Mutual exclusion is therefore violated as a result of a lack of atomicity.

Exercise

- Consider a program with four concurrent threads $T_1 \ldots T_4$, where each thread T_i executes a single computation C_i . We want this program to comply with the precedence graph below:

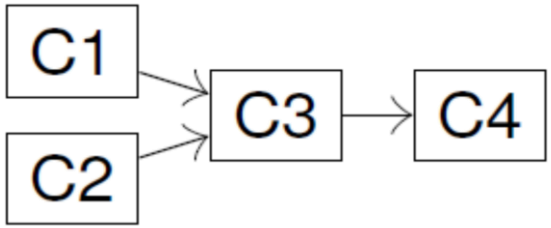


- Write the code for each thread (i.e., computation and synchronization) so that all precedence constraints are satisfied. Please also specify the initial value of each semaphore.

Initial Conditions	Thread1	Thread2	Thread 3	Thread4

Exercise - Solution

- Consider a program with four concurrent threads $T_1 \ldots T_4$, where each thread T_i executes a single computation C_i . We want this program to comply with the precedence graph below:



- Write the code for each thread (i.e., computation and synchronization) so that all precedence constraints are satisfied. Please also specify the initial value of each semaphore.

Initial Conditions	***Thread1***	***Thread2***	***Thread 3***	***Thread4***
$S_0 = 0$ $S_1 = 0$ $S_2 = 0$	C_1 $Signal\ (S_0)$	C_2 $Signal(S_1)$	$Wait(S_0)$ $Wait(S_1)$ C_3 $Signal(S_2)$	$Wait(S_2)$ C_4