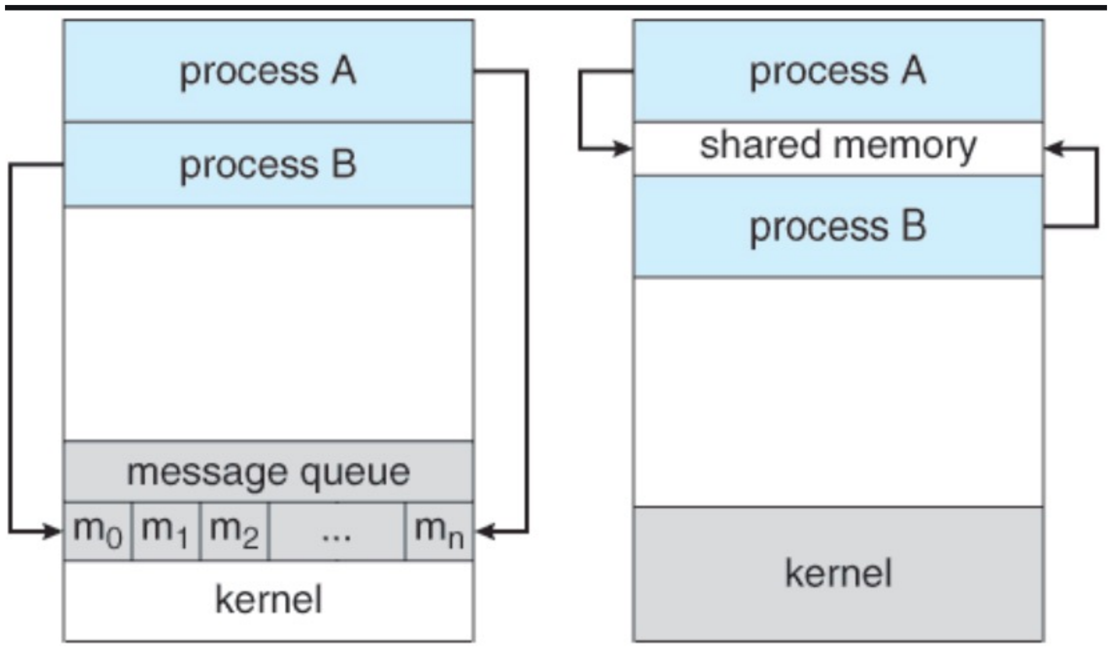
Chapter 6: Synchronization Tools

Background:

A cooperating process is one that can affect or be affected by other processes executing in the system.

- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through **shared memory** or **message passing**.
- Concurrent access to **shared data** may result in data **inconsistency**, however.
- Mechanisms to ensure the **orderly execution** of cooperating processes that share a logical address space.



(a) Message Passing (b) Shared Memory

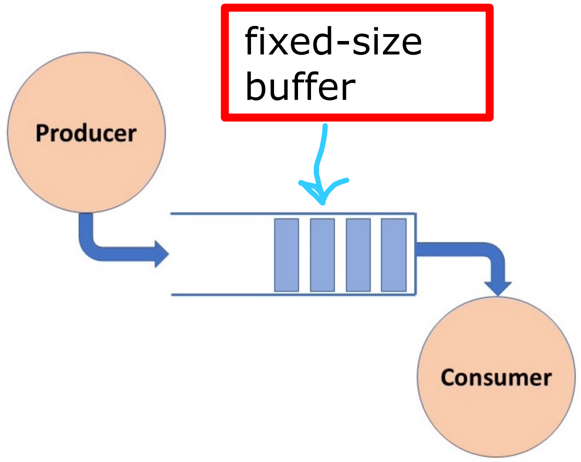
Chapter 6-Objectives

- Critical-section problem
- Race conditions
- Hardware solutions to the critical-section problem
- Mutex locks, semaphores, monitors, and condition variables to solve the critical section problem

Cooperating processes

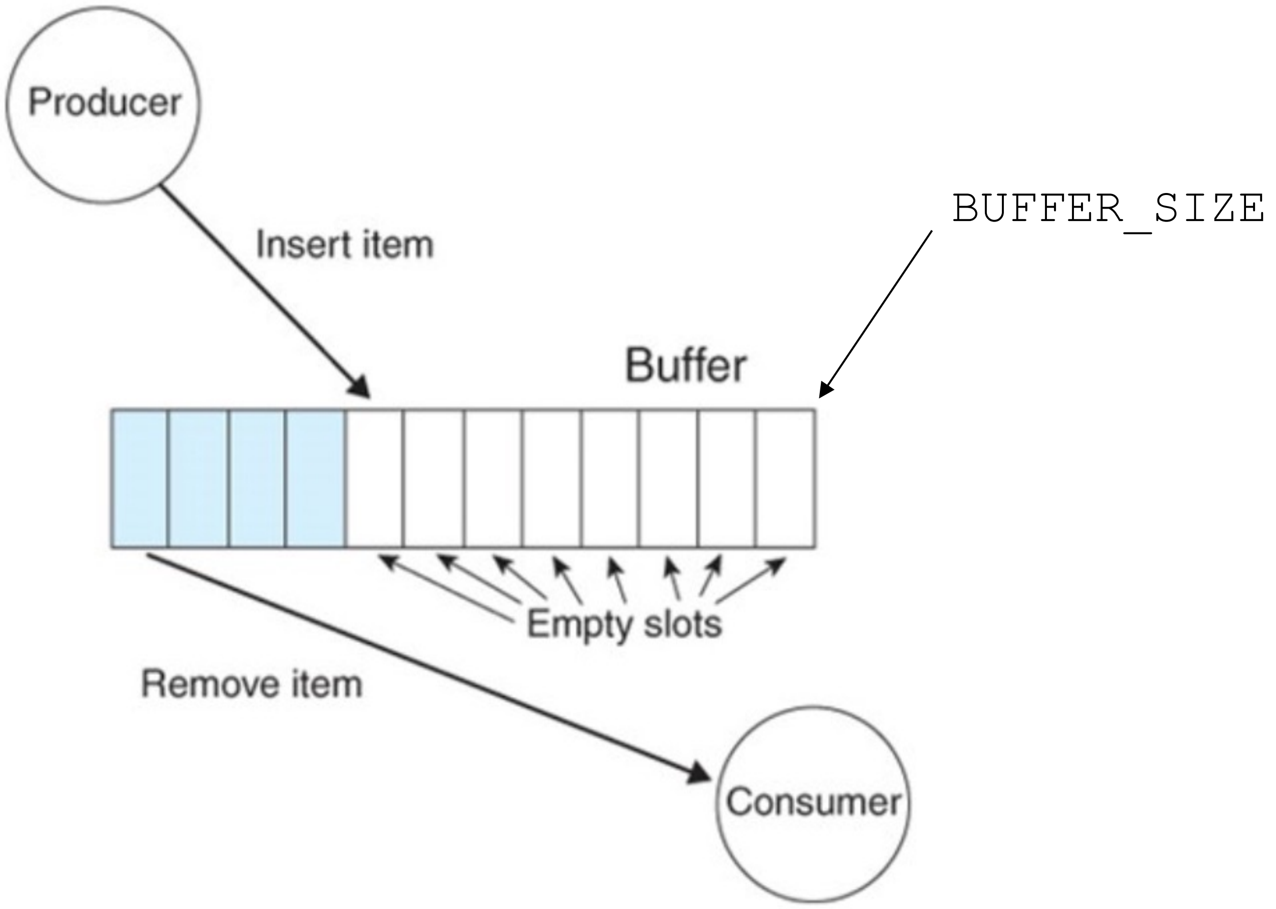
- CPU scheduler switches rapidly between processes to provide concurrent execution.
  - Processes may be interrupted **at any time.**
  - This means that one process may only partially complete execution before another process is scheduled.
  - Concurrent/parallel execution can contribute to issues involving the **integrity of data** shared by several processes
- Mechanisms to ensure the **orderly execution** of cooperating processes

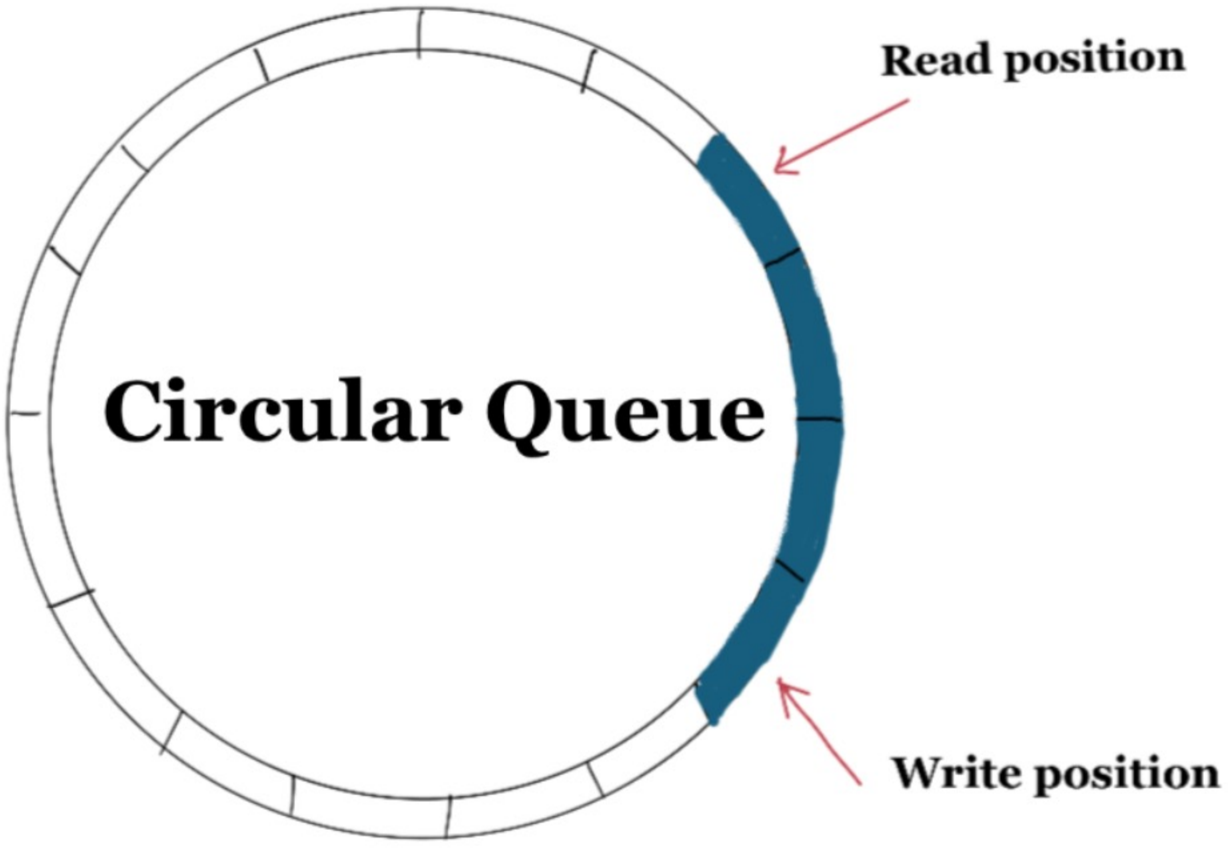
Producer-consumer problem



- Producer process creates an item and adds it to the shared buffer.
- Consumer process takes items out of the shared buffer and "consumes" them.

- Suppose that we wanted to provide a solution to the consumer-producer problem with a limited buffer size:
  - We can do so by having an integer **`counter`** that keeps track of the buffer.
  - Initially, **`counter`** is set to 0.
  - It is **incremented by the producer** after it produces a new item and is **decremented by the consumer** after it consumes an item.


Producer
Insert item
Buffer
BUFFER_SIZE
Empty slots
Remove item
Consumer


Read position
Circular Queue
Write position

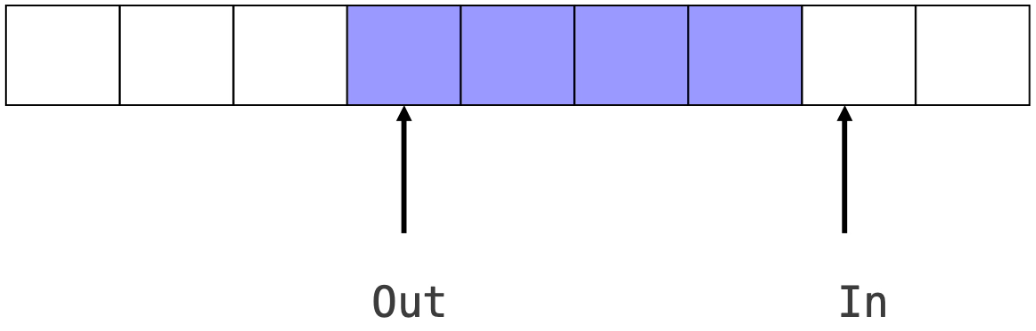
Producer

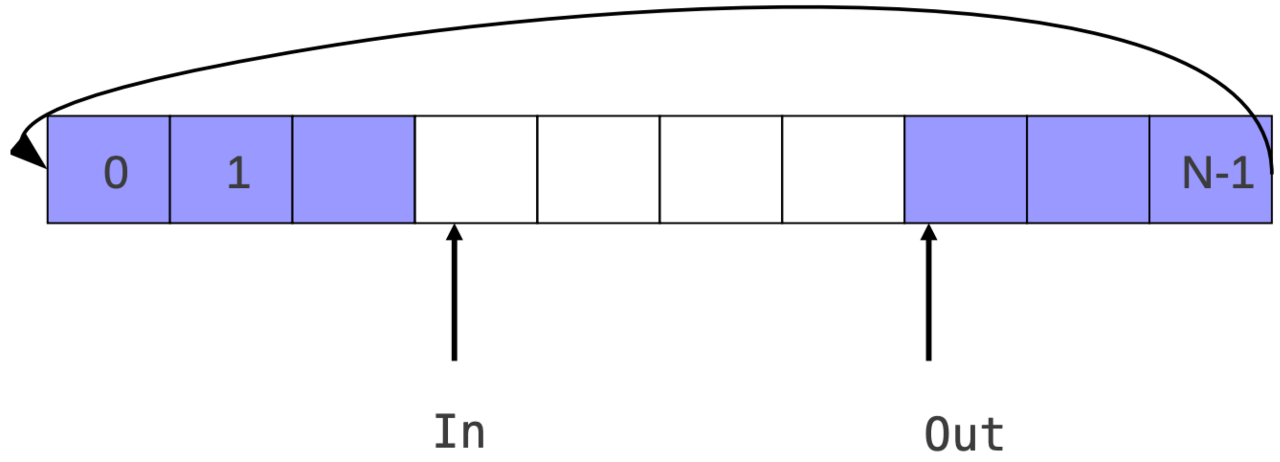
```
while (true) {
     /* produce an item in next produced */

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

Circular buffer/queue

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```




0
1
N-1
In
Out

Race Condition

- `counter++` could be implemented as:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

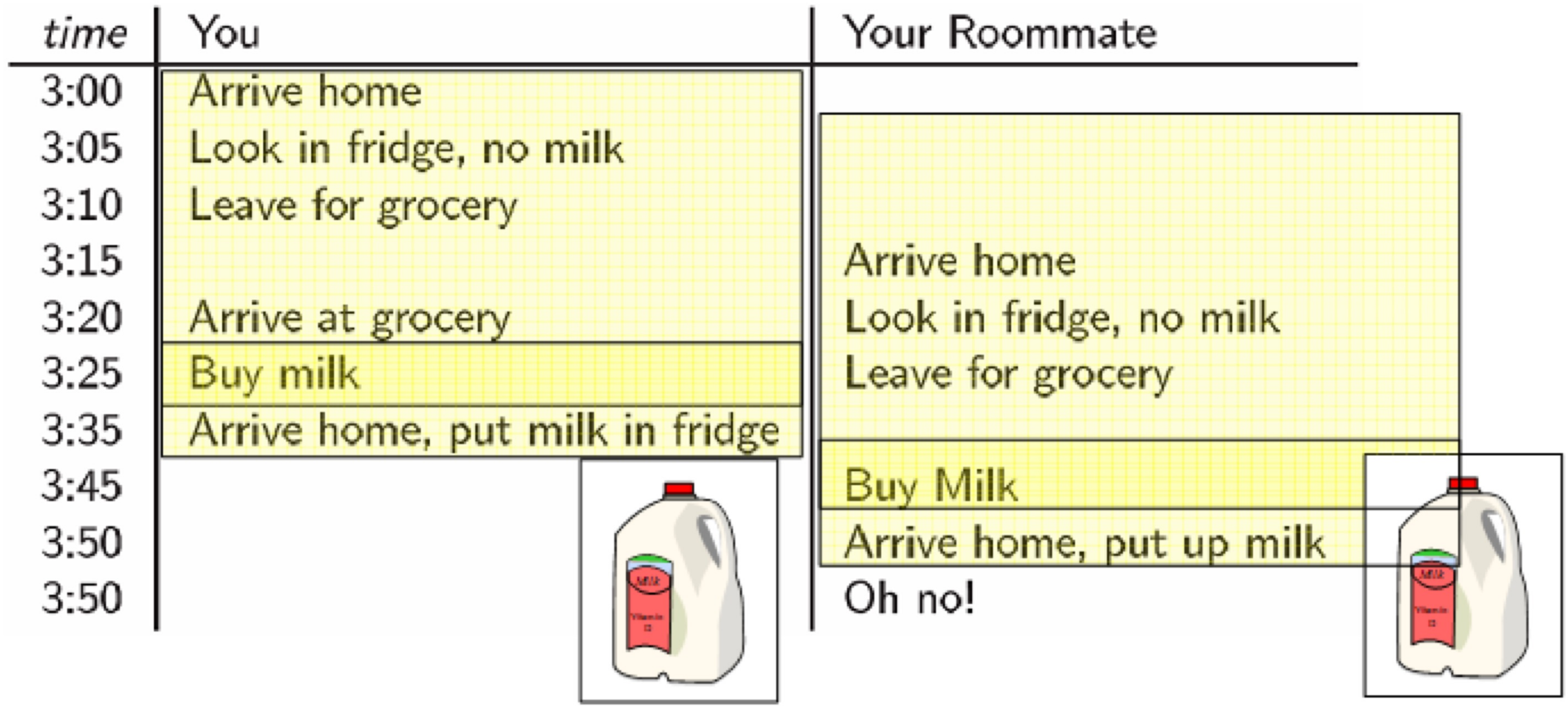
outcome of the execution depends on the particular order in which the access takes place!

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

```
S0: producer execute register1 = counter          {register1 = 5}
S1: producer execute register1 = register1 + 1    {register1 = 6}
S2: consumer execute register2 = counter          {register2 = 5}
S3: consumer execute register2 = register2 - 1    {register2 = 4}
S4: producer execute counter = register1          {counter = 6 }
S5: consumer execute counter = register2          {counter = 4}
```

Two processes manipulate the variable counter concurrently!

Too much milk problem



time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home, put milk in fridge	
3:45		Buy Milk
3:50		Arrive home, put up milk
3:50		Oh no!

“Withdraw at the same time from a joint account” Problem

Balance = $300

ATM -1

ATM-2

$300 ←Check Balance

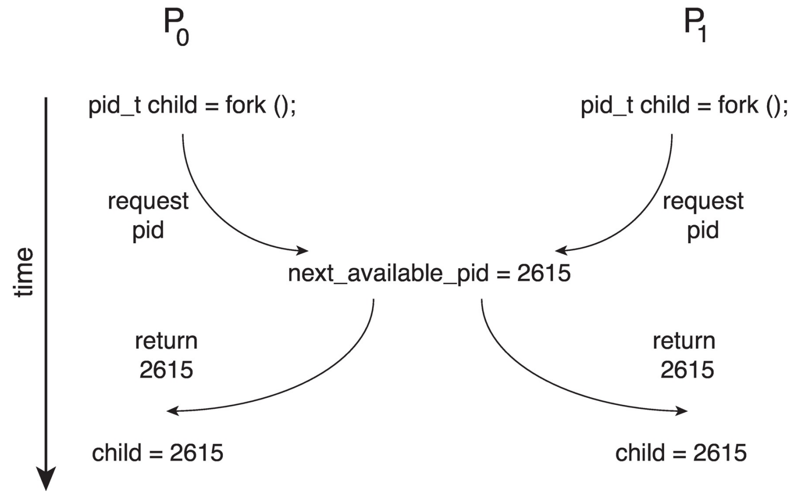
Check Balance → $300

??? ←Withdraw $300

Withdraw $300 → ???

Race Condition – kernel level example

- Processes P_0 and P_1 are creating child processs using the `fork()` system call.
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid).



- Unless there is **mutual exclusion**, the **same `pid`** could be assigned to two different processes!

Critical Section Problem

- Consider a system of ***n*** processes {p_0, p_1, … p_{n-1}}.
- Each process has **critical section** segment of code:
  - Process may be changing common/shared variables, updating table, writing file, etc
  - When one process in critical section, **no other process** may be in its critical section.
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section.**

Critical Section

General structure of process P_i

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

Critical-Section Handling in OS

- The critical-section problem could be solved simply in a **single-core** environment if we could **prevent interrupts from occurring** while a shared variable was being modified.
- Unfortunately, this solution is **not feasible** in a **multiprocessor/multicore** environment.
  - Disabling interrupts on such systems can be **time consuming**, since this message must be passed to all the processors.

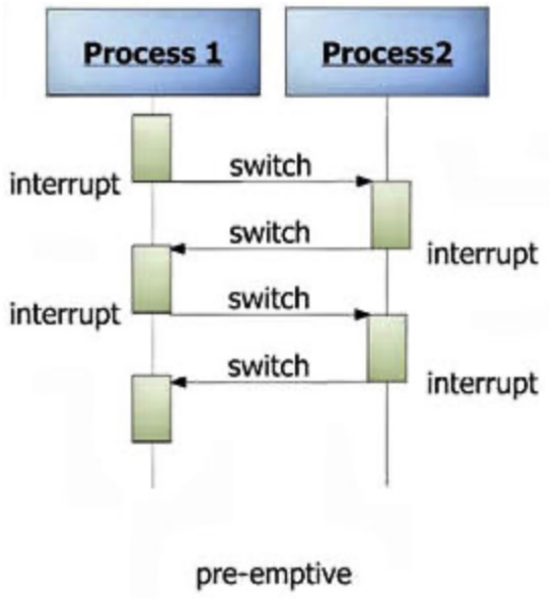
Solution to Critical-Section Problem

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

Critical-Section Handling in OS

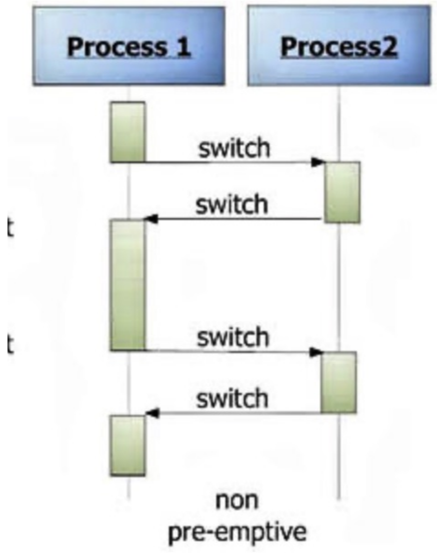
Two approaches:

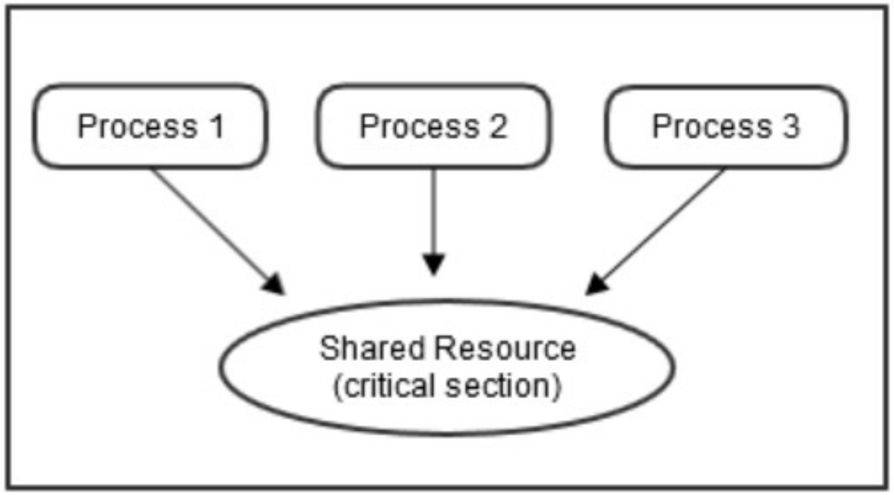
- **Preemptive kernel** – allows preemption of a process when running in kernel mode.
    - It is possible for two kernel-mode processes to run simultaneously on different CPU cores.

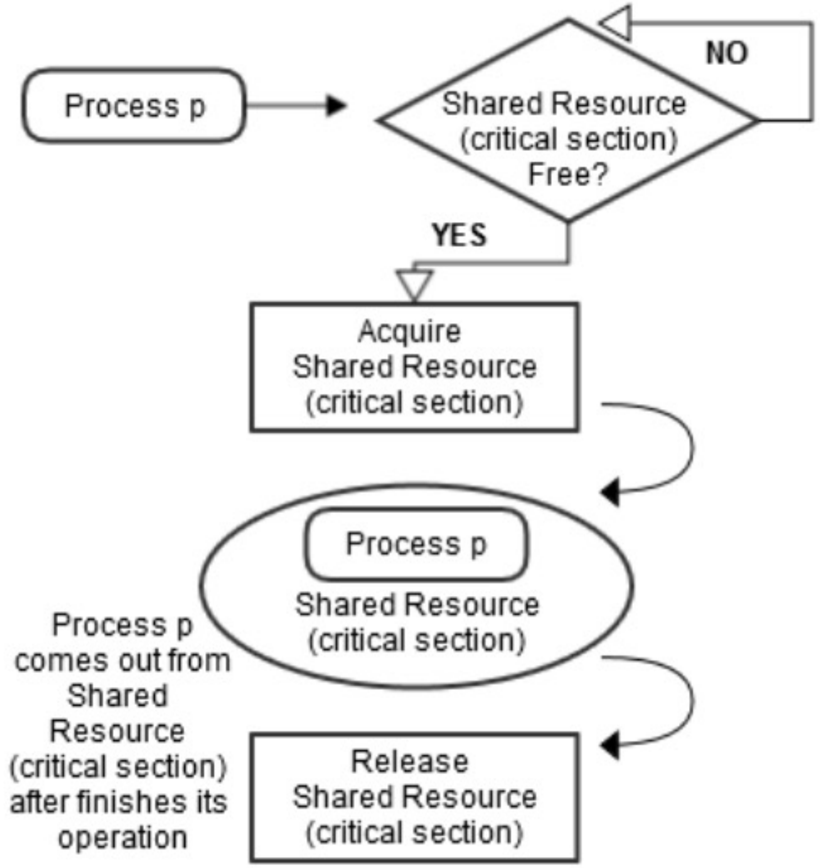


Critical-Section Handling in OS

- **Non-preemptive kernel** – process runs until exits kernel mode, blocks, or voluntarily yields the CPU
    - Essentially **free of race** conditions in kernel mode as only one process is active in the kernel at a time.
    - Non-preemptive kernels cannot be used for real-time scheduling or multiprocessor systems




Process 1
Process 2
Process 3
Shared Resource
(critical section)


Process p
Shared Resource (critical section) Free?
NO
YES
Acquire Shared Resource (critical section)
Process p
Shared Resource (critical section)
Process p comes out from Shared Resource (critical section) after finishes its operation
Release Shared Resource (critical section)

Peterson's Solution

- Not guaranteed to work on modern architectures (but good algorithmic description of solving the problem)!

- 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

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;
```

```
        /* critical section */
```

```
        flag[i] = false;
```

```
        /* remainder section */
```

```
}
```

Peterson's Solution (Cont.)

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

Two-thread example

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag)
    ;
print x
```

- Thread 2 performs

```
x = 100;
flag = true
```

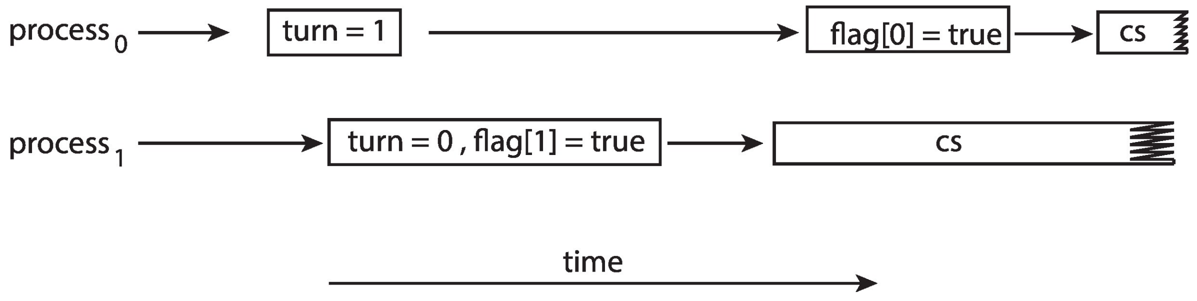
- What is the expected output?

Peterson's Solution

Operations for Thread 2 may be **reordered**:

```
flag = true;
x = 100;
```

- If this occurs, the output may be 0!
- The effects of instruction reordering in Peterson's Solution:



- This allows both processes to be in their critical section at the same time!

***To improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies.**

Synchronization Hardware

- Software-based solutions are **not guaranteed** to work on modern computer architectures.
- Many systems provide **hardware support** for implementing the critical section code.
  - These primitive operations can be used **directly** as synchronization tools, or they can be used to form the more abstract synchronization mechanisms.
- Three forms of hardware support:

  1. Memory barriers

  2. Hardware instructions

  3. Atomic variables

Memory Barriers

- Instruction that forces any change in memory to be propagated (made visible) to all other processors.
- When a memory barrier instruction is performed, the system ensures that all **loads and stores** are completed **before any subsequent load or store** operations are performed.

Memory Barrier

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100 (the correct value):
- Thread 1:

```
while (!flag)
   memory_barrier();
print x
```

- Thread 2:

```
x = 100;
memory_barrier();
flag = true
```

* Low-level operations and are typically only used by kernel developers

Hardware Instructions

- Special hardware instructions that allow us to either ***test-and-modify*** the content of a word, or two *swap* the contents of two words **atomically** (uninterruptible)
- We can use these special instructions to solve the critical-section problem
- **Test-and-Set** instruction
- **Compare-and-Swap** instruction

test_and_set Instruction

```
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`**

If two `test_and_set()` instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order .

Solution using test_and_set()

- Shared boolean variable **lock**, initialized to **false**
- Solution for a process P_i:

```
do {
     while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

      lock = false;
      /* remainder section */
} while (true);
```

compare_and_swap Instruction

```
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 the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer **`lock`** initialized to 0;
- Solution:

```
while (true){
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

The first process that invokes compare_and_swap() will set lock to 1. It will then enter its critical section. Subsequent calls to compare_and_swap() will not succeed, because lock now is not equal to the expected value of 0.

Does not satisfy the bounded-waiting requirement

- **Intel processors: cmpxchg**

Atomic Variables

- **Atomic variable:** provides *atomic* (**uninterruptible**) updates on basic data types such as integers and booleans.
- For example, the **`increment()`** operation on the atomic variable **`sequence`** ensures **`sequence`** is incremented without interruption:

```
increment(&sequence);
```

- Atomic variables can be used to ensure **mutual exclusion** in situations where there may be a race on a single variable while it is being updated (e.g. a **counter** is incremented).

Atomic Variables

- The `increment()` function can be implemented as follows:

```
void increment(atomic_int *v)
{
     int temp;

     do {
            temp = *v;
     }
     while (temp != (compare_and_swap(v,temp,temp+1));
}
```

They do not entirely solve race conditions in all circumstances.

Mutex Locks

Previous solutions are **complicated** and **inaccessible** to application programmers. OS designers build **software tools** to solve **critical section** problem:

- 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 such as compare-and-swap.
- ***Busy waiting:*** While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to `acquire()`.
- This lock therefore called a ***spinlock***

Solution to Critical-section Problem Using Locks

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

Mutex Lock Definitions

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

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

These two functions must be implemented atomically.
Both `test-and-set` and `compare-and-swap` can be used to implement these functions.

Semaphore

Busy waiting: while a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to `acquire()`.

- Semaphore ***S*** – integer variable: can only be accessed via **two indivisible (atomic)** operations
  - **`wait()`** and **`signal()`**

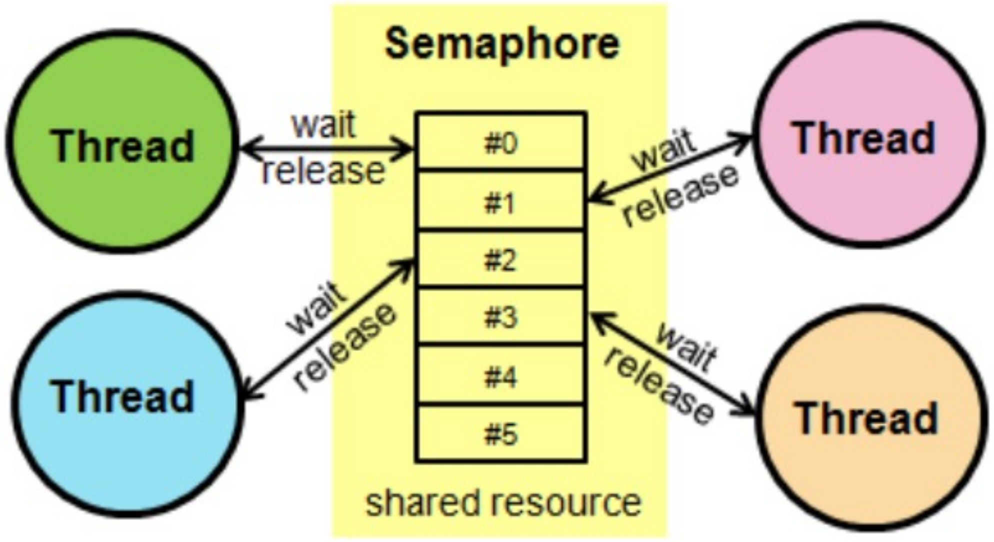
Definition of the **`wait()`** operation:

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

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed atomically

Definition of the **`signal()`** operation:

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


Semaphore
Thread
wait
release
#0
#1
#2
#3
#4
#5
shared resource
Thread
wait
release
Thread
wait
release
Thread
wait
release

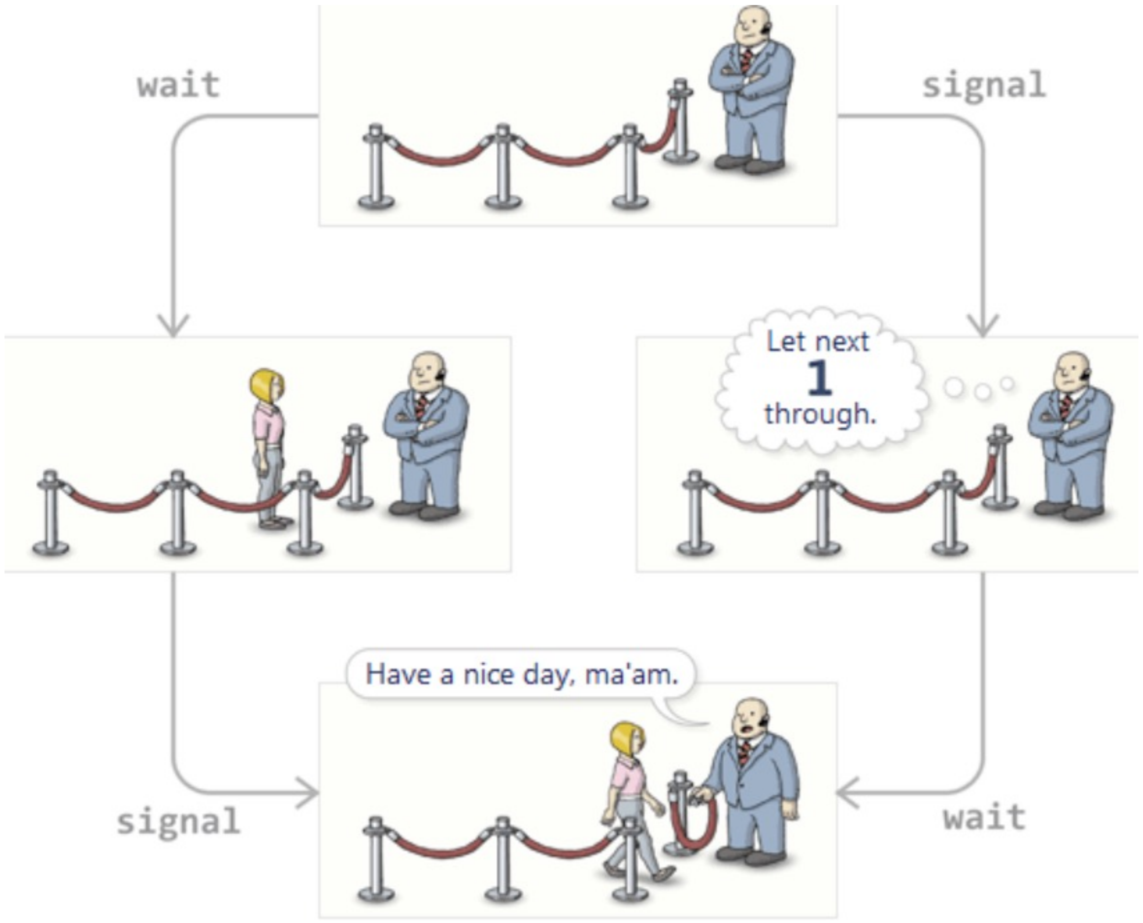
Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
- Consider ***P₁*** and ***P₂*** that require ***S₁*** to happen before ***S₂***

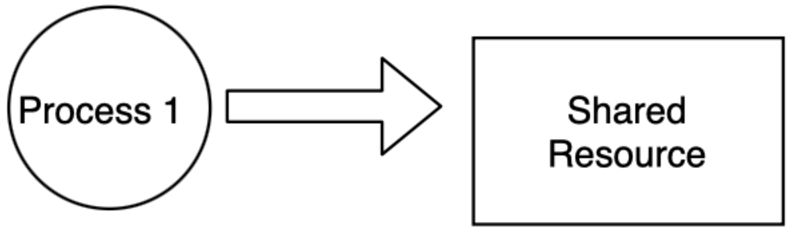
  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);



P2:
    wait(synch);
    S2;
```

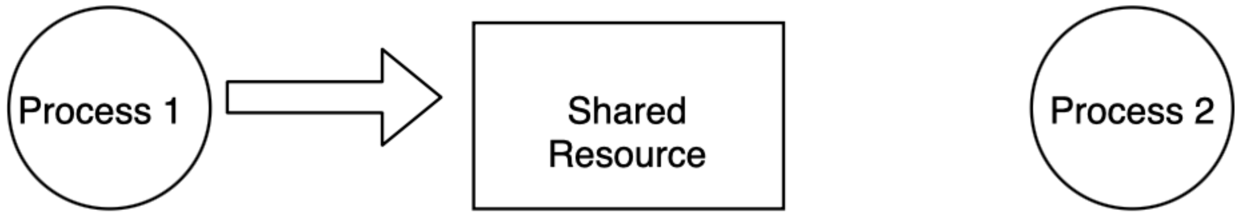
P1 and P2 share a common semaphore `synch`, initialized to 0.


wait
signal
Let next 1 through.
Have a nice day, ma'am.
signal
wait

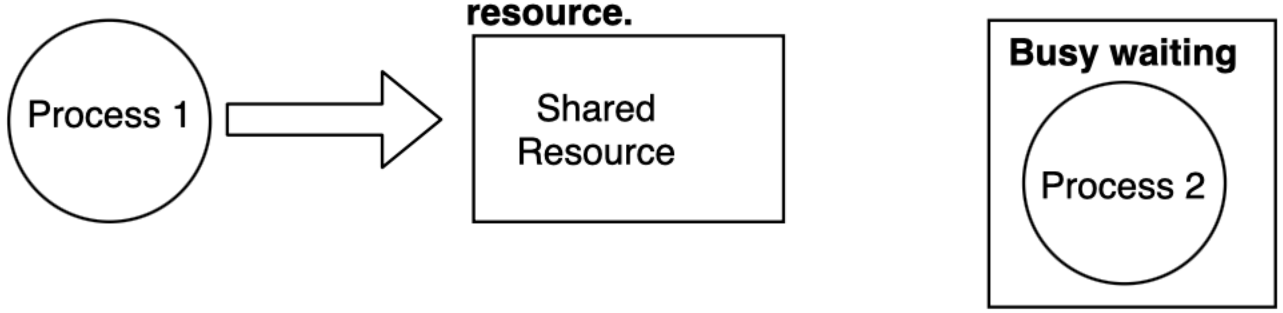
1. Process 1 is using the shared resource.



2. Process 2 is now in need of the shared resources.



3. Process 2 enters busy waiting till it has access to the shared resource.



Semaphore Implementation

- Rather than engaging in busy waiting, the process can **suspend** itself.
- Process is placed into a **waiting queue** associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

- A process that is suspended, waiting on a semaphore S, should be **restarted** when some other process executes a `signal()` operation.
  - The process is restarted by a `wakeup()` operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **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

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

Implementation with no Busy waiting (Cont.)

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

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

Monitors

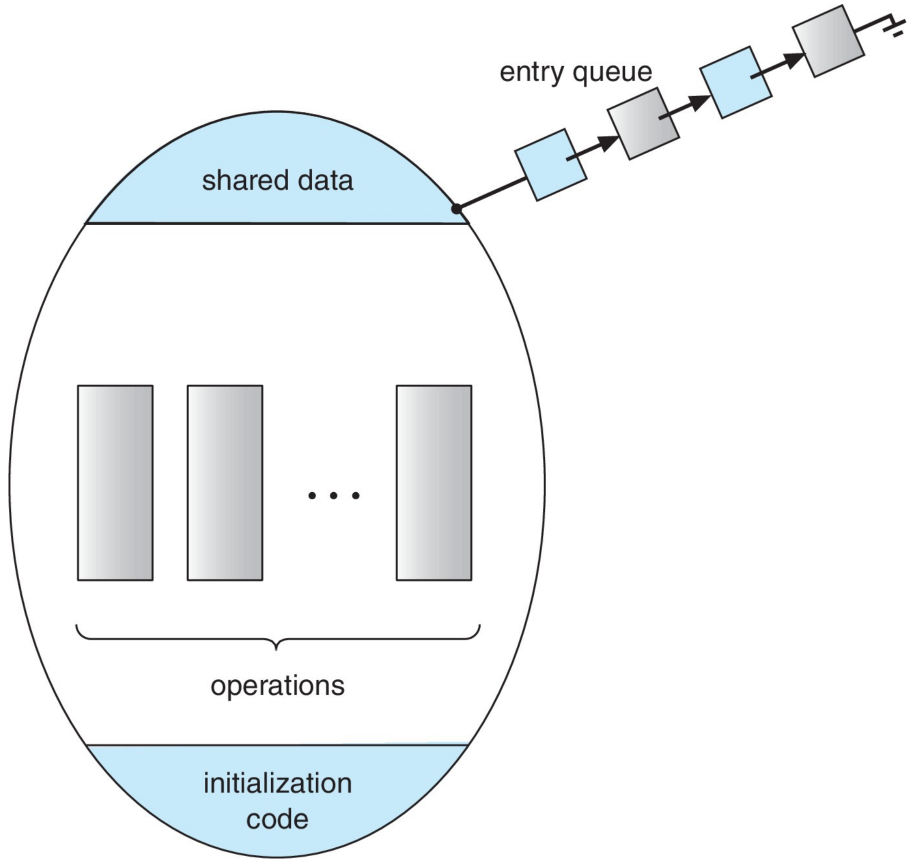
- Hide mutual exclusion: provides a convenient and effective mechanism for process synchronization
  - *Abstract data type*, internal variables only accessible by code within the procedure
  - Only one process may be active within the monitor at a time

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

  function P2 (…) { …. }

  function Pn (…) {……}

  initialization code (…) { … }
}
```

A function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.

- Similarly, the local variables of a monitor can be accessed by only the local functions.

Schematic view of a Monitor



only one process at a time is active within the monitor

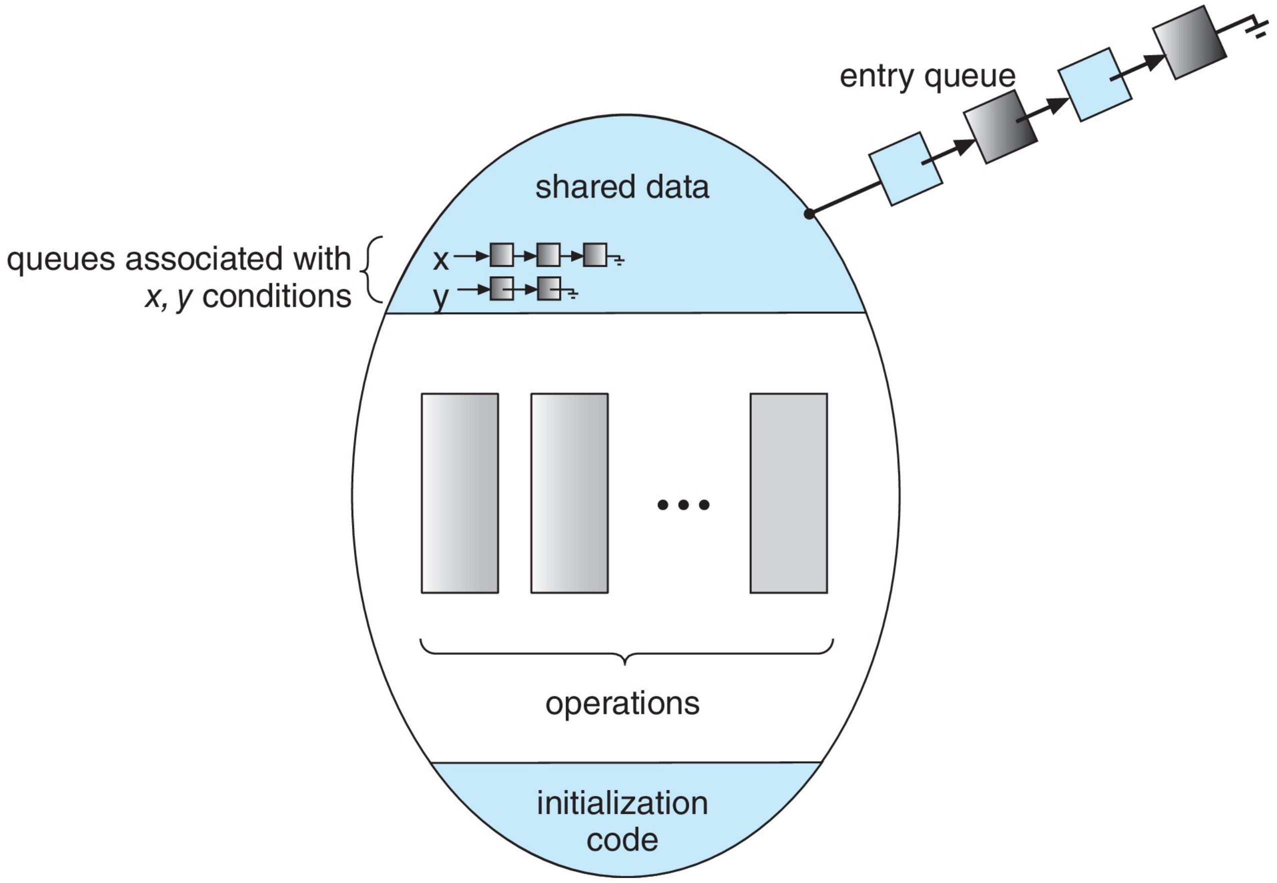
Condition Variables

- Monitors: insufficient for modeling some synchronization schemes

```
        condition x, y;
```

- Two operations are allowed on a condition variable:
  - `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 invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a **mutex lock** or **semaphore**.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a **liveness failure**.

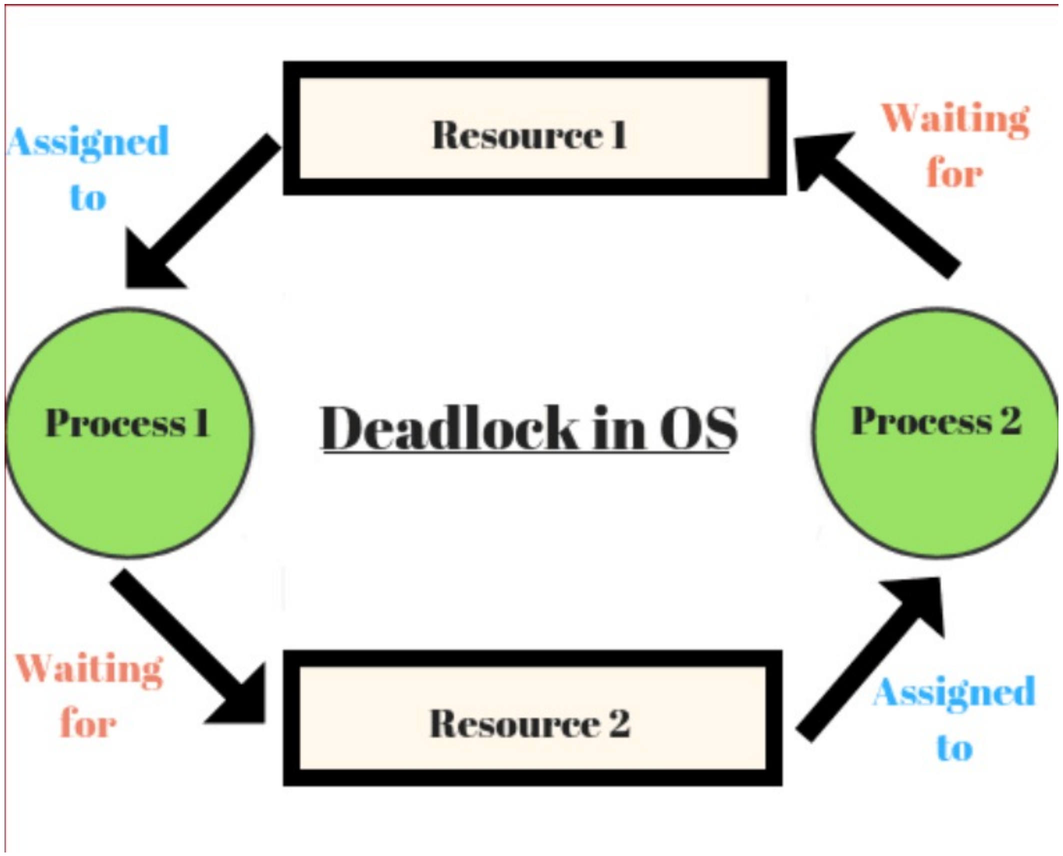
Liveness

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

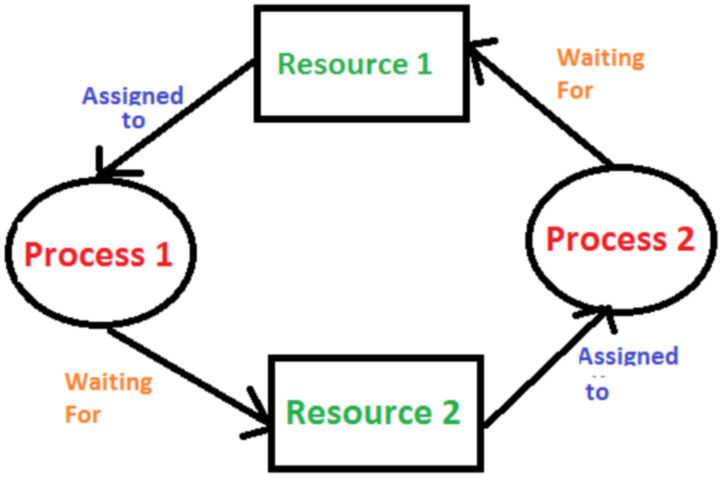
P_0 $\qquad\qquad\qquad\qquad$ P_1

```
   wait(S);              wait(Q);
   wait(Q);              wait(S);
    ...                   ...
   signal(S);            signal(Q);
   signal(Q);            signal(S);
```

- Consider if P_0 executes wait(S) and P_1 wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q)
- However, P_1 is waiting until P_0 execute signal(S).
- Since these signal() operations will never be executed, P_0 and P_1 are **deadlocked**.


Resource 1
Assigned to
Waiting for
Process 1
Deadlock in OS
Process 2
Waiting for
Resource 2
Assigned to

Liveness



- Other forms of deadlock:
- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**