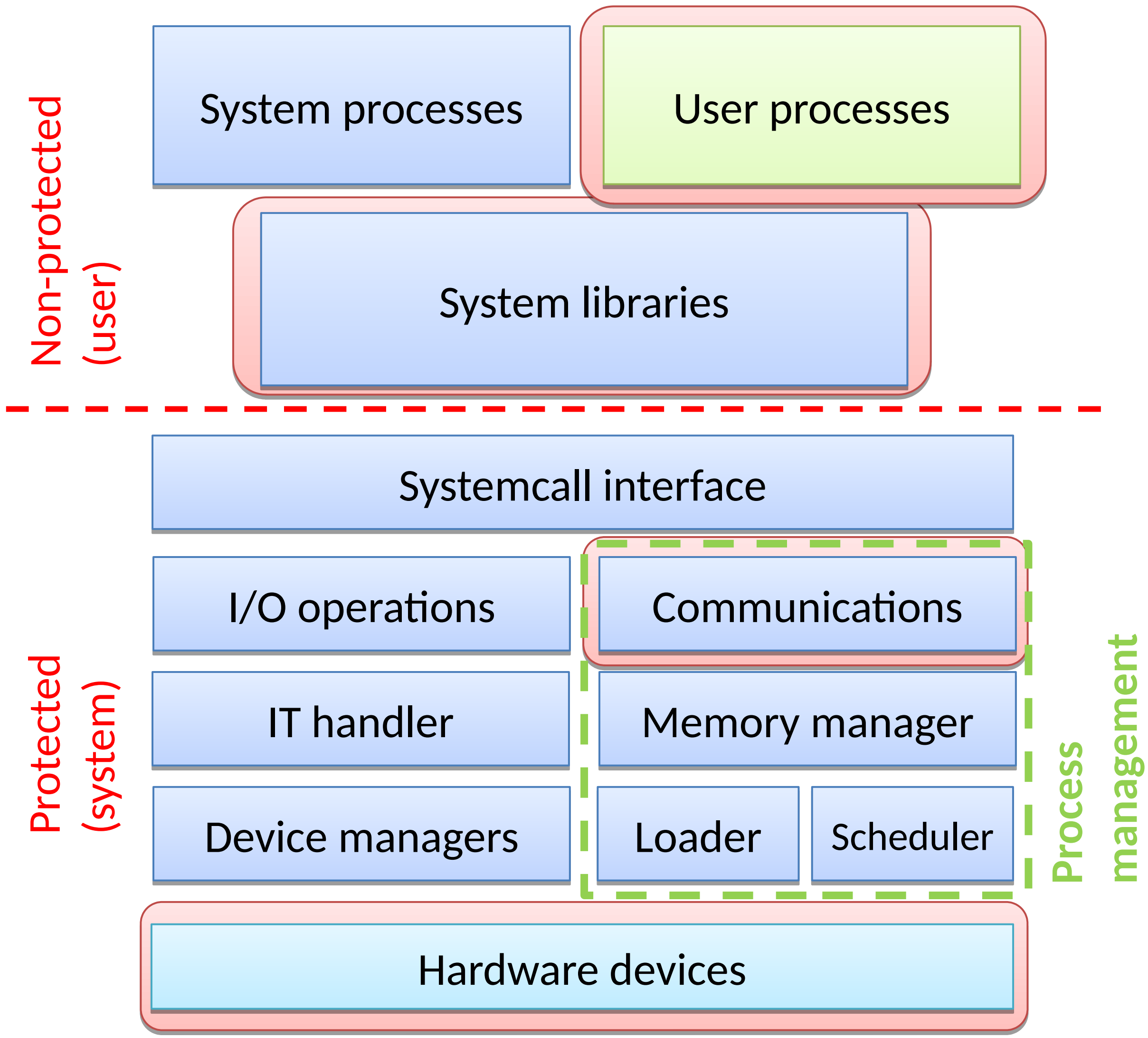
Operating Systems – Synchronization between tasks

Hussein Al-Rikabi
rhussein@mit.bme.hu

Budapest University of Technology and Economics (BME)
Department of Measurement and Information Systems (MIT)

The slides of the latest lecture will be on the course page. (https://www.mit.bme.hu/eng/oktatas/targyak/vimiab00)

The main blocks of the OS and the kernel (recap)



Parallel job execution

- The basic goal of the OS is to support user job execution
  - Jobs are executed by tasks (maybe more than one)
  - Executing jobs may require the executor tasks to **cooperate**
  - A modern OS is multiprogrammed → executing jobs parallel

- Task implementations: processes and threads
  - Threads in the same process are using shared memory → **competitive** environment
  - Processes are separated
    - Communication has to be synchronized
    - Parallel running processes may have to compete for the resources

- Current systems require parallel programming
  - The clock frequency is almost reached its technological boundary
  - Multithreaded execution is the design principle → Multicore CPUs
  - This also requires a new programming principle

Competition and cooperation between tasks

- Tasks may operate independently from each others
  - Not influencing each others operation
  - Asynchronous execution
  - This separation is made possible by the OS
  - The resources (CPU, RAM, HW devices) are used by more than one task
    - Conflicts may appear
    - Execution **dependencies** created
  - These conflicts have to be solved by the OS

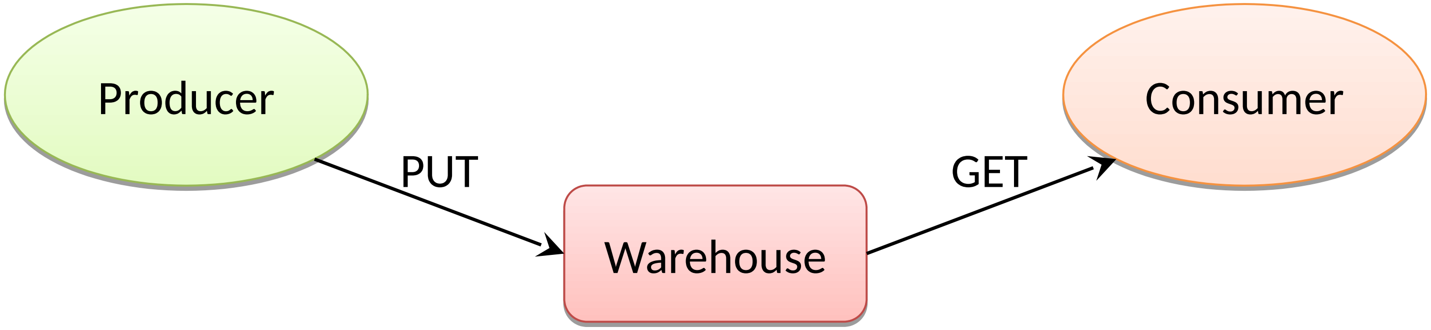
- The user jobs also require the tasks to cooperate
  - The job is decomposed to separate tasks
  - These tasks have to cooperate communicate and synchronize with each other
  - The OS provide services for this

- Remark: A single processor system is also a competitive environment that needs synch.

Simple example: producer – consumer problem

- The description of the problem
  - The producer creates a product which is stored in a warehouse (in a variable)
  - The consumer consumes the product from the warehouse
  - The producer and the consumer are working simultaneously
    - They may work separately in time
    - They may work with different rates
    - This can cause a problem.

- Problems to solve
  - Granting the consistency of the warehouse data structure
  - The consumer shouldn't check for products in an infinite loop
  - The producer shouldn't place new product in the warehouse while it's full



Synchronization between tasks

- Synchronization means coordination between tasks by constraining operation execution in time
  - The execution of specific tasks can be slowed down (temporally stopped) in order to achieve combined operation
- Basic application of synchronization
  - In competitive environments: using shared resources
  - In cooperation: communication
- The „price” of synchronization
  - It may cause performance degradation
  - The waiting tasks are not „useful”
    - They cannot wait for I/O operations, they are blocked
  - And if we have: **No sync**.: no waiting, erroneous behavior is possible
  - Bad sync. scheme: too much waiting, bad resource utilization

The basic forms of synchronization

- **Mutual exclusion**
  - **Critical section**: an instruction sequence of the tasks, which are cannot executed simultaneously
  - Shared resources are protected with this method, so **competitive situations** can be managed
  - Pessimistic method: locks the resource in every case
  - E.g.: while the printer is printing, cannot start a new print
- **Rendezvous**
  - Specific operations (part) of the tasks can start at the same time
  - **Cooperation** scheme to synchronize the operation of sub-tasks
  - E.g.: Send() and Receive() methods of direct (non buffered) messaging
- **Precedence**
  - The operations of the tasks should be executed in a predefined order
  - **Cooperation** scheme

The Critical-Section Problem

- The critical section is an instruction sequence with restricted execution: only one task can execute them at the same time
- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code like:
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design 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**

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

The Critical-Section Problem

- The rules of the restriction
  - Entering
    - It is forbidden to enter, when another task is in the critical section
    - If no tasks in the critical section, only then that task can enter, which were executing it's own instructions before the critical section
  - Exiting
    - The critical section should finished in finite time
    - Common programming mistake: the tasks don't leave the critical section (releasing the resources)

Critical-Section Problem Solution requirements

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** - There exists a bound, or limit, on the number of times a task, can enter their critical sections

There are many solutions to the Synchronization problems (Consumer-Producer, Critical-section)

TBCed

Peterson's Solution (Software-based solution)

- Good algorithmic description of solving the CS problem, but it may not work on modern computer architectures.
- The two processes **pi** and **pj** alternating and sharing 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 $\mathbf{P}_\mathtt{i}$ is ready!
  - **`flag[j] =`** ***`true`*** $\mathbf{P}_\mathtt{i}$ is ready

Peterson's Solution

int turn

Indicates whose turn it is to enter its critical section.

boolean flag [2]

Used to indicate if a process is ready to enter its critical section.

Structure of process P_i in Peterson's solution

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

        critical section

      flag [i] = false ;

        remainder section

  } while (TRUE) ;
```

What happens when both Pi and Pj wants to enter CS? Giving turns to each other?

Peterson's Solution

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

        `Pi` enters CS only if:

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

Hardware support

- Software-based solutions are **not** guaranteed to work on modern computer architectures
- Instead, we can have general solution using hardware (Locks)
  - That is, a process must acquire a lock before entering a critical section
  - releases the lock when it exits the critical section

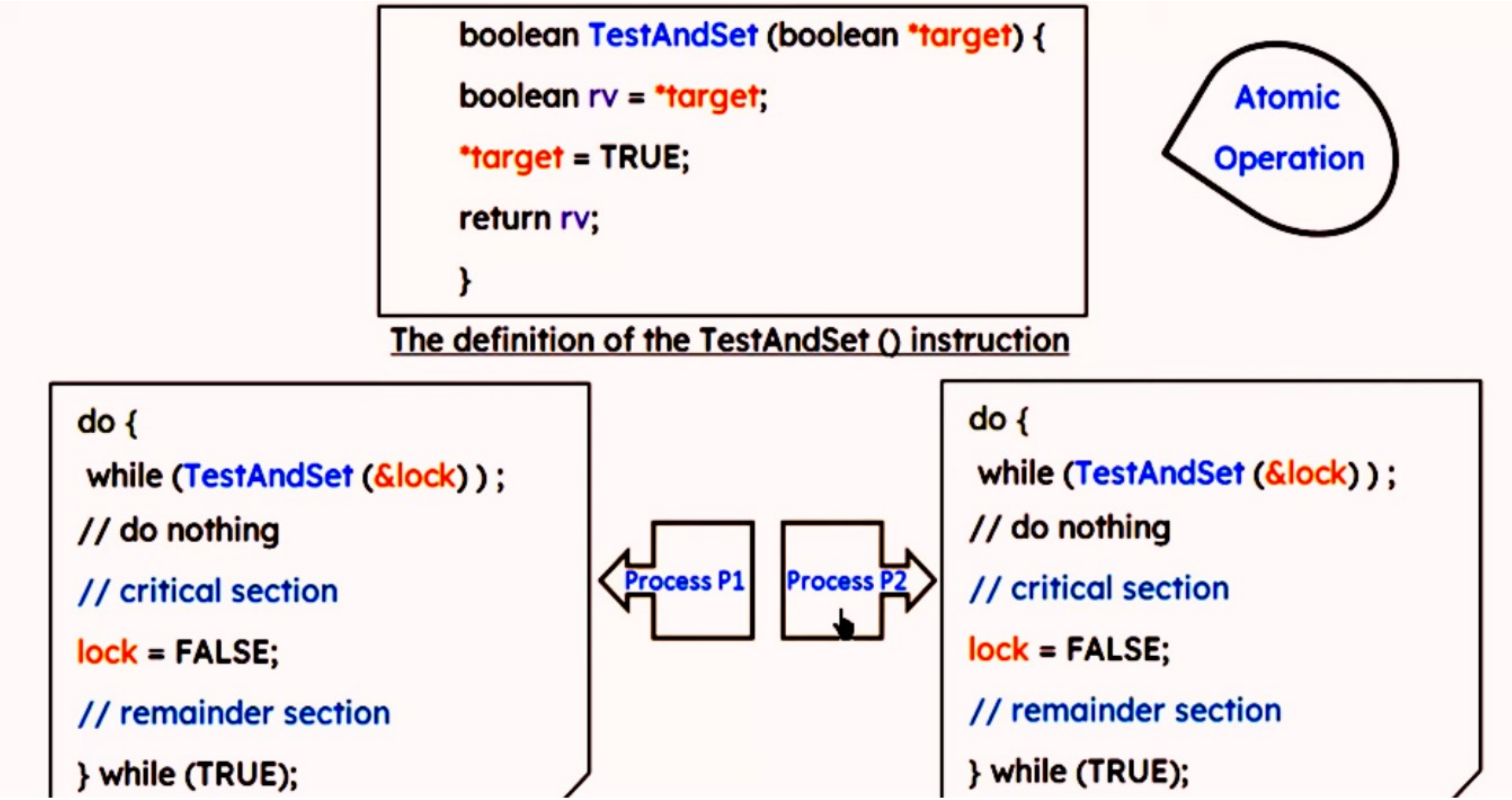
```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

Hardware support for synchronization

- **Simple solution**: disable interrupts when a task executes the critical section
  - no preemption is possible, mutual exclusion is realized
  - It makes the system cooperative (non preemptive)
  - It may used in single processor systems
  - Disabling the interrupts may lead to omitting important events
  - Cannot be used in a multiprocessor environment
    - Other operations also disabled on the other CPU-s
- **Good solution**: **atomic** (non interruptible) memory instruction pairs
  - **Test-and-Set-lock (TSL)**: Sets the lock value to true
    - `while( TSL(lock) ) { }`
    - Waits until the lock is released and engages the new lock for the current task
    - The lock is usually a binary variable (true-false)
  - **Compare-and-swap (CAS)**: A variable is only modified when it has a specified value
    - `while ( CAS(var, a, b) == a ) {}`
    - Waits until `var` value is a, then sets it to b
    - Works on larger variables also (e.g. arrays)

TSL Instruction (Definition and Solution)

Shared Boolean variable lock, initialized to FALSE



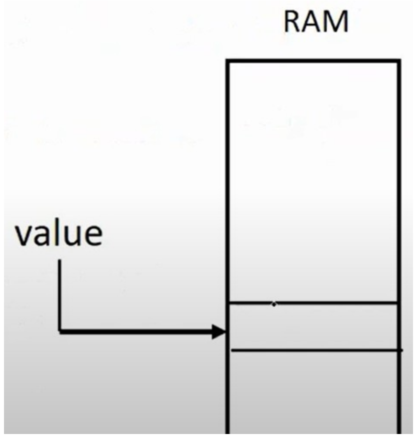
Evaluation of TSL

- It satisfies the Mutual exclusion
- Bounded-waiting requirement is not met
- When a process Px make a request to enter the critical section, there might be a lot of other processes that keep entering their CS, so this will make Px to be **starved**.

Compare-And-Swap CAS Instruction

Definition: --- Atomic --- Shared integer "lock" initialized to 0;

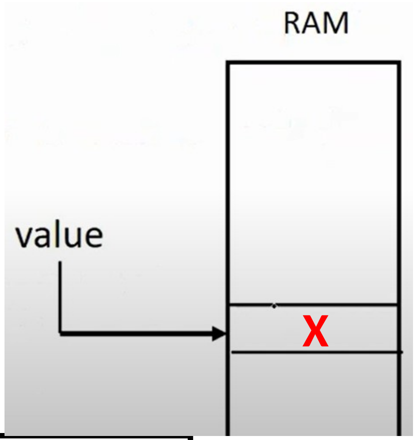
```
int Compare_And_Swap(int *value, int expected, int new_value)
    {
        int temp = *value;

        if (*value == expected)
           *value = new_value;
     return temp;
     }
```



CAS Instruction

Shared integer "lock" initialized to 0;

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

        if (*value == expected)
           *value = new_value;
     return temp;
    }
```




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

Evaluation of CAS

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”. That is, the **swap** takes place only under this condition.

Definition of CAS

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

    if (*value == expected)
        *value = new_value;

    return temp;
}
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- Software tools designed to solve critical section problem
- Simplest is **mutex** lock
- 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
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Mutex Locks

```
•   acquire() {
        while (!available)
            ; /* busy wait */
        available = false;
     }
•   release() {
        available = true;
     }
•   do {
     acquire lock
        critical section
     release lock
       remainder section
  } while (true);
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – non-negative integer variable … that is **shared** between threads.
- Apart from initializations , Can only be accessed via two indivisible (**atomic**) operations
  - **`wait()`** and **`signal()`**
    - Originally from Dutch was called called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

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

Checking if S is less or equal to zero,
S is the Semaphore, True ⇒ waiting
Faulse ⇒ break the while and do (S - -)
Why this Decrement? **If S was 1**

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

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

Increasing S when a process was using a Semaphore finished the operation. Denoting that it is releasing it.

Remark; when a process modifies S, no other process can modify it simultaneously.

Types of Semaphore

- Binary Semaphore – called also Mutex Locks
  - How is it Mutex, see the waint,signal with s=0/1. initial value of S is 1.

```
P (Semaphore S) {
    while (S <= 0)
      ; // no operation
    S-- ;
}
```

```
V (Semaphore S) {
    S++ ;
}
```

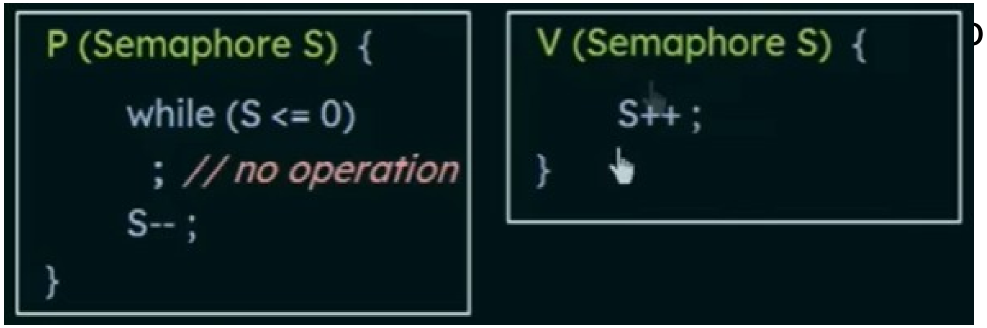
P1:	P2:
P(S)	P(S)
CS	
V(S)	
	CS
	V(S)

- A process:

```
do {
   wait(mutex);        S

      // critical section

   signal(mutex);      S

      // remainder section
} while (TRUE);
```

Types of Semaphore

- Counting Semaphore ▭ S value can range over unrestricted domain that it can be 0, 1, 2, 3 …
- Controlling the access to a resource that has multiple instances
- like system with 3 CPUs, U1,U2 and U3, and you have 3 processes, by this method P1 can use U1 and P2 can use U2…
- But the Resources are limited, here 3 CPUs, so we set the S initially to 3. so that it will be
  - 2 ▭ one process running
  - 1 ▭ two process running
  - 0 ▭ three process running – which means no more resource to use (**wait**)
  - When one is ource is free.



Disadvantages of semaphore

```
P (Semaphore S) {
    while (S <= 0)
     ; // no operation
    S-- ;
}
```

- It requires **Busy-Waiting**; If one process on a a CPU is trying to enter CS, it will be stuck in that true while loop, until it gets CS access.
- This type of Semaphore is call **SpinLock**, P is spinning in this While.
- Solving this issue by Modifying the wait/signal functions.
  - We can remove the waiting command
  - The process can block itself instead and
  - Switch it to **waiting state**. And
  - Placed in a special queue called **waiting queue**
  - The scheduler then can execute other tasks.
  - Can cause Starvation, **Deadlocks**

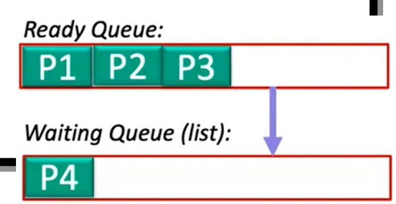
Implementation of semaphore without Busy-Waiting

- We define a semaphore as a Struct: S:
  - Value
  - List

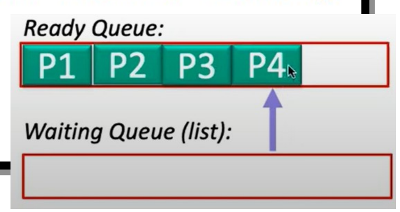
```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

- When a process must wait on a semaphore
  - Block: → Wait Q
- Signal: Resource Available
  - WakeUp: Wait Q → Ready

```
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);
        }
}
```



Deadlock and Starvation

- **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

```
        P0                              P1
          wait(S); <---same time ---> wait(Q);
          wait(Q);                     wait(S);
    ...              ...
          signal(S);                  signal(Q);
          signal(Q);                  signal(S);
```

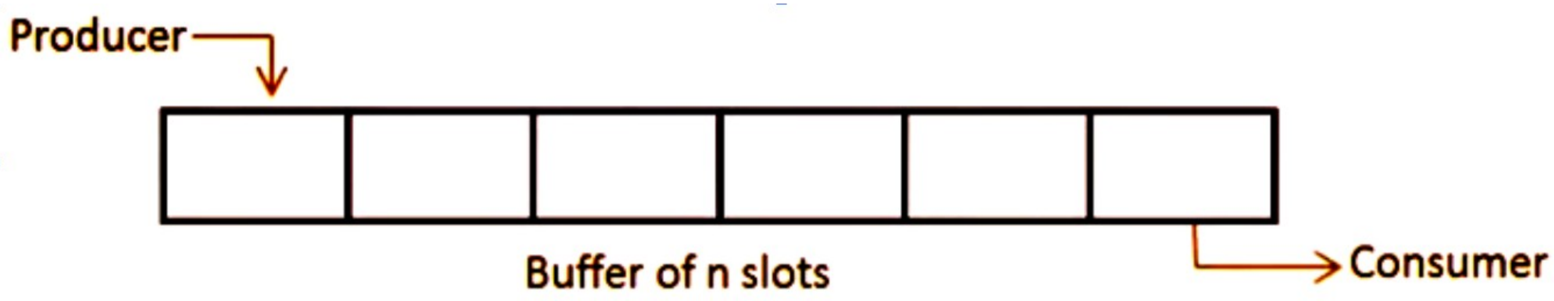
- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Classic Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem (producer-consumer problem)
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Producer-Consumer problem (Semaphores Solution)

- ***n*** (Slots) buffer, each can hold one item. The problem is:
  - When buffer is full, no incretion done by producer
  - Consumer should not remove date When buffer is empty
  - They should not add/remove simultaneously
- Semaphore **`mutex(m)`**`:acquire/Release lock`
  - initialized to the value 1 (binary semaphore)
- Semaphore **`full`** initialized to the value 0 (counting semaphore)
- Semaphore **`empty`** initialized to the value n (counting semaphore)



Producer-Consumer problem (Semaphores Solution)

Producer

```
do {
  wait (empty); // wait until empty>0
                   and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```

Consumer

```
do {
  wait (full); // wait until full>0 and
                   then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty);  // increment 'empty'
} while(TRUE)
```

```
P (Semaphore S) {
      while (S <= 0)
        ; // no operation
      S-- ;
}
```

```
V (Semaphore S) {
      S++ ;
}
```

Readers and Writers Problem (Semaphores Solution)

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

- To solve this problem we need to make writers to have exclusive access
- Shared Data
  - Integer `readcount` initialized to 0 (how many processing are reading)
  - Semaphore `wrt` initialized to 1 (Common for W/R) (0/1)
  - Semaphore `mutex` initialized to 1 (Mutual exclusion for readcount) (0/1)

Readers and Writers Problem (Semaphores Solution)

Writer Process

```
do {
/* writer requests for critical
section */
    wait(wrt);
   /* performs the write  */
    // leaves the critical section
    signal(wrt);
} while(true);
```

```
P (Semaphore S) {
      while (S <= 0)
        ; // no operation
      S-- ;
}
```

```
V (Semaphore S) {
      S++ ;
}
```

Overview of the locking methods

- Lock bit
  - Single bit, accessing it is atomic, e.g.: TSL
- Mutex (mutual exclusion lock)
  - A tool for implementing critical sections
- Semaphore
  - A data structure with two atomic operations: `wait(P)` and `signal(V)`
- Spinlock (spinning lock)
  - Busy waiting lock bit, mutex or semaphore
  - E.g.: TSL and CAS instructions
  - It should be used only when the lock is used for a short time
- ReaderWriterLock
  - Any number of readers may enter the critical section
  - If a writer is entering, it will be blocked until all of the readers are left the critical section
- RecursiveLock
  - The task which has the lock can re-lock the same lock without blocking
  - It is useful for recursive functions

Thank You – To be continued