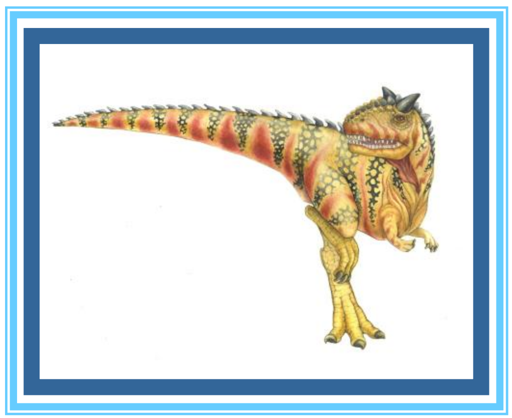
Xi'an Jiaotong-Liverpool University
西交利物浦大學

CPT104 - Operating Systems Concepts

Process Synchronization



Process Synchronization

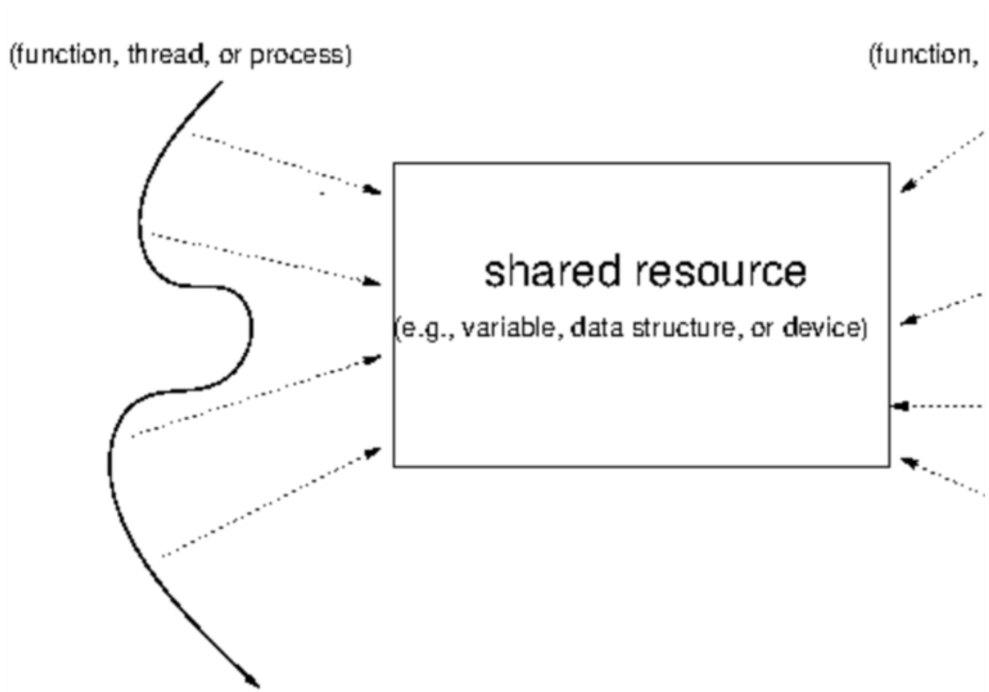
- Background
- The Critical-Section Problem
  - Peterson’s Solution
  - Synchronization Hardware
  - Mutex Locks / Mutual exclusion
  - Semaphores
- Classical Problems of Synchronization

Process Synchronization

- **Background**
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks / Mutual exclusion
- Semaphores
- Classical Problems of Synchronization

What is Process Synchronization (PS)?

- PS is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources, at one time.
- *n* processes all competing to use some shared resource.



Process Synchronization

- **Concurrent access** to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- **Race condition**: The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- **To prevent** race conditions, **concurrent processes must be synchronized**

Process Synchronization

- Background
- **The Critical-Section Problem**
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks / Mutual exclusion
- Semaphores
- Classical Problems of Synchronization

The Critical Section Problem

Critical Sections are sequences of code that cannot be interleaved among multiple threads/processes.

Each (concurrent) thread/process has a code segment, called ***Critical Section (CS),*** in which the shared data is accessed**.**

When using critical sections, the code can be broken down into the following sections:

Some arbitrary code (such as initialization).

code that tries to enter CS

Sequence of code = CS

.

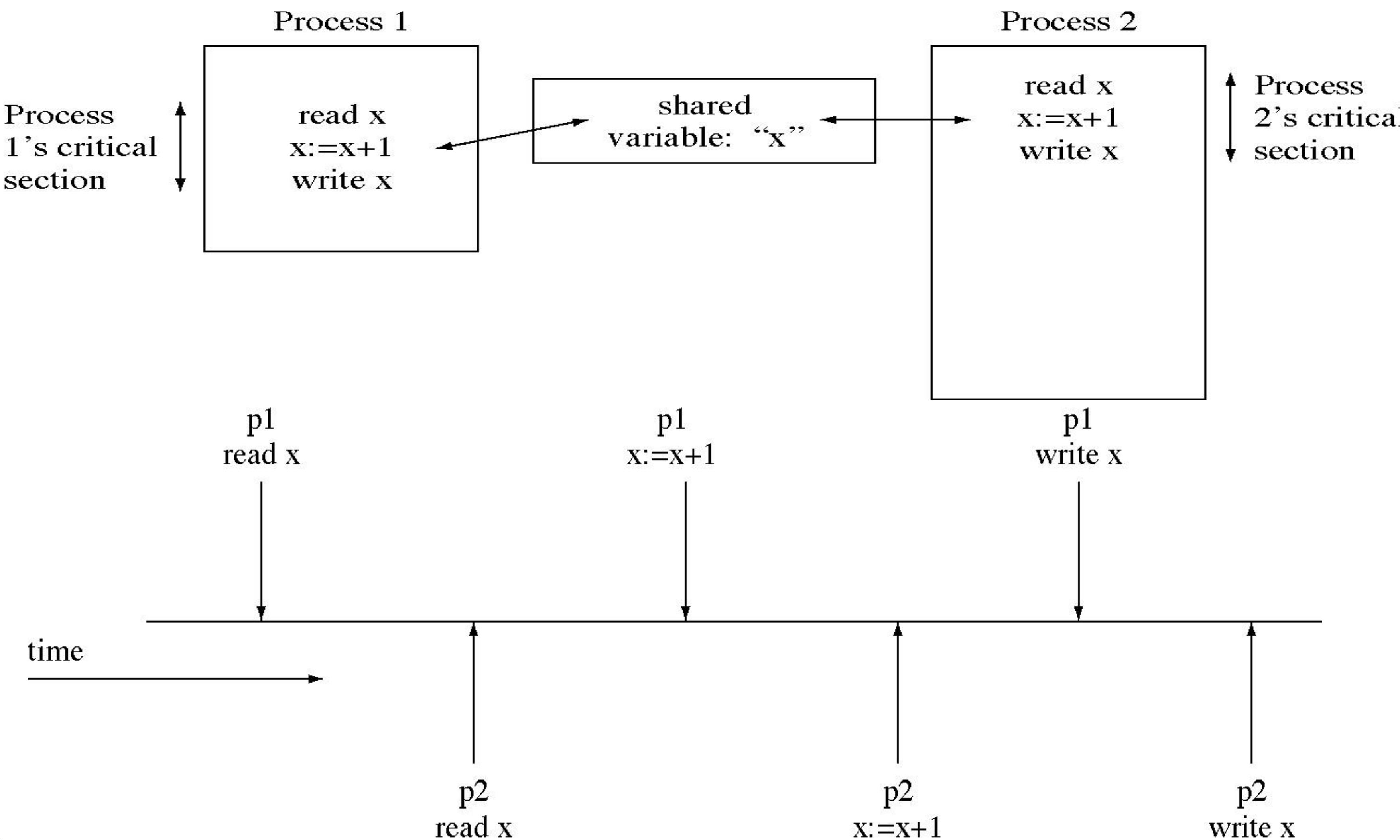
Code that leaves CS

can contain any code

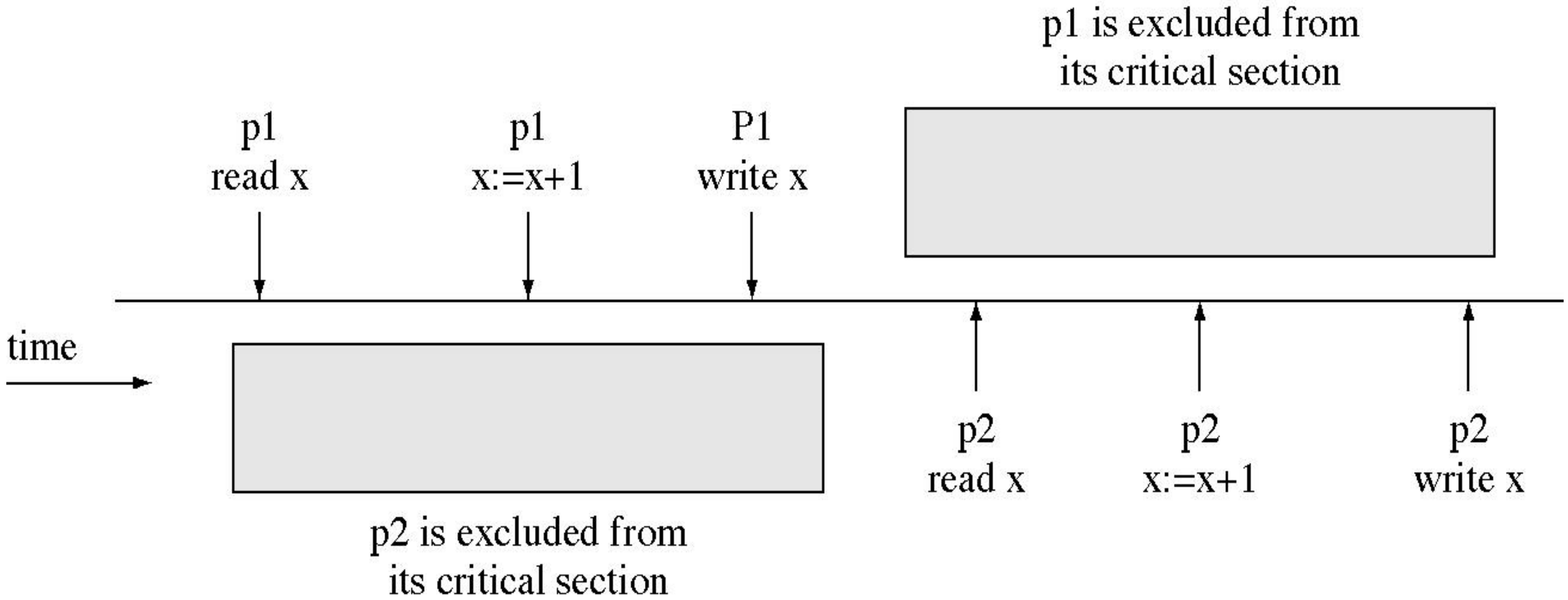
```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Race condition updating a variable

CS = codes that reference one variable in a “read-update-write” fashion



Critical section to prevent a race condition



- Multiprogramming allows logical parallelism (multiple programs to exist in memory at the same time) uses devices efficiently but we lose correctness when there is a race condition.

- Avoid/ forbid / deny execution in parallel inside critical section, even we lose some efficiency, but we gain correctness.

Solutions to CS problem

Concurrent processes come into conflict when they use the same resource (competitively or shared)

for example: I/O devices, memory, processor time, clock

There are 3 requirements that must stand for a correct solution:

- **Mutual exclusion**
- **Progress**
- **Bounded waiting**

The Critical-Section Problem

- **Mutual Exclusion**: When a process/thread is executing in its critical section, **no other process/threads** can be executing in their critical sections.

- **Progress**: If no process/thread is executing in its critical section, and if there are some processes/threads that wish to enter their critical sections, then one of these processes/threads will get into the critical section. **No process running outside** its critical region **may block any process.**

- **Bounded Waiting**: **No process**/thread should have **to wait forever** to enter into the critical section.

  - the waiting time of a process/thread outside a critical section should be **limited** (otherwise the process/thread could suffer from ***starvation***).

Types of solutions to CS problem

- **Software solutions**
  - algorithms whose correctness relies only on the assumption that only one process/thread at a time can access a memory location/resource
- **Hardware solutions**
  - rely on special machine instructions for “locking”
- **Operating System** and **Programming Language solutions** (e.g., Java)
  - provide specific functions and data structures for programmers to use for synchronization.

Software solutions

Peterson's Solution

- is used for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.
- The central problem is to design the **entry** and **exit** sections

```
do {
   entry section
    critical section - CS
   exit section
    remainder section - RS
} while(TRUE)
```

Peterson's Solution

- **Only 2 processes, P0 and P1**
- It was formulated by Gary L. Peterson in 1981.

Processes may share some common variables to synchronize their actions.

```
int turn;              // indicates whose turn it is to enter the critical section.

boolean flag[2];       // initialized FALSE,
                       // indicates when a process wants to enter into their CS.
                       // flag[i] = true implies that process Pi is ready (i = 0,1)
```

NEED BOTH the **turn** and **flag[2]** to guarantee *Mutual Exclusion, Bounded waiting*, and *Progress*.

Peterson's Solution

Peterson's algorithm

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

        CRITICAL SECTION

    flag[i] = FALSE;

            REMAINDER SECTION
 }
while (TRUE);
```

Initialization: flag[0]:=flag[1]:=false

turn:= 0 or 1

Process P_0

do {

// Critical Section

flag[0] = true; *// It means P0 is ready to enter its critical section*

turn = 1; *// It means that if P1 wants to enter than allow it to enter and P0 will wait*

// Condition to check if the flag of P1 is true and turn == 1, this will only break when one of the conditions gets false.

while (flag[1] && turn == 1); *// do nothing*

/critical section/

// It sets the flag of P0 to false because it has completed its critical section.

flag[0] = false;

// Remainder Section

} while (true);

Process P_1

do {

// Critical Section

// It means process P0 is ready to enter its critical section

flag[1] = true;

turn = 0; *// It means that if P0 wants to enter than allow it to enter and P1 will wait*

// Condition to check if the flag of P0 is true and turn == 0, this will only break when one of the conditions gets false.

while (flag[0] && turn == 0); *// do nothing*

/critical section/

//it sets the flag of P1 to false because it has completed its critical section.

flag[1] = false;

// Remainder Section

} while (true);

This solution is correct:

The three CS requirements are met:

- **Mutual Exclusion** is assured as only one process can access the critical section at any time.

  each Pi enters its critical section only if either:

  flag[j] = false or *turn = i*

- **Progress** is also assured, as a process outside the critical section does not block other processes from entering the critical section.

- **Bounded Waiting** is preserved as every process gets a fair chance.

Hardware Solutions

Solution to CS Problem using **LOCKS**

Control the entry into CS and gets a **LOCK** on required resources.

do {

entry section

CRITICAL SECTION

exit section

remainder section

} while(true)

Remove the **LOCK** and let the others know that its CS is over.

Hardware Solutions

Single-processor environment - *could disable interrupts*

Effectively stops scheduling other processes.

TEST AND SET SOLUTION

Initially: **lock** value is set to **0**

Lock value = 0 means the critical section is currently vacant and no process is present inside it.

Lock value = 1 means the critical section is currently occupied and a process is present inside it.

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

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

- satisfy the mutual exclusion requirement, but unfortunately do not guarantee bounded waiting.

Hardware Solutions

Multi-processor environment

- provides special **atomic** hardware instructions. *Atomic* means *non-interruptable (i.e., the instruction executes as one unit)*

- a **global variable lock** is initialized to **0**.

- the only Pi that can enter CS is the one which finds **lock = 0**

- this Pi excludes all other Pj by setting **lock to 1**.

COMPARE AND SWAP SOLUTION

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

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Hardware Solution

Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable

Disadvantages

- **Busy-waiting** is when a process is waiting for access to a critical section it continues to consume processor time.
- **Starvation** is possible when a process executes its critical section, and more than one process is waiting for a long time.
- **Deadlock** is the *permanent* blocking of a set of processes waiting an event (the freeing up of CS) that can only be triggered by another blocked process in the set.

Operating Systems and
Programming Language Solutions

- ***Mutex***
- ***Semaphore***

1. Mutex Lock / Mutual exclusion

- A mutex is a **programming flag** used to grab and release an object.
- When **data processing** is started that **cannot be performed simultaneously** elsewhere in the system, **the mutex is set to lock** which **blocks other attempts to use it**.
- The mutex is set to **unlock** when the data are no longer needed, or the routine is finished.

–**To enforce mutex** **at the kernel level** and prevent the corruption of shared data structures - **disable interrupts** for the smallest number of instructions is the best way.

–**To enforce mutex** **in the software areas** – use the **busy-wait mechanism**

__busy-waiting mechanism__ is a mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

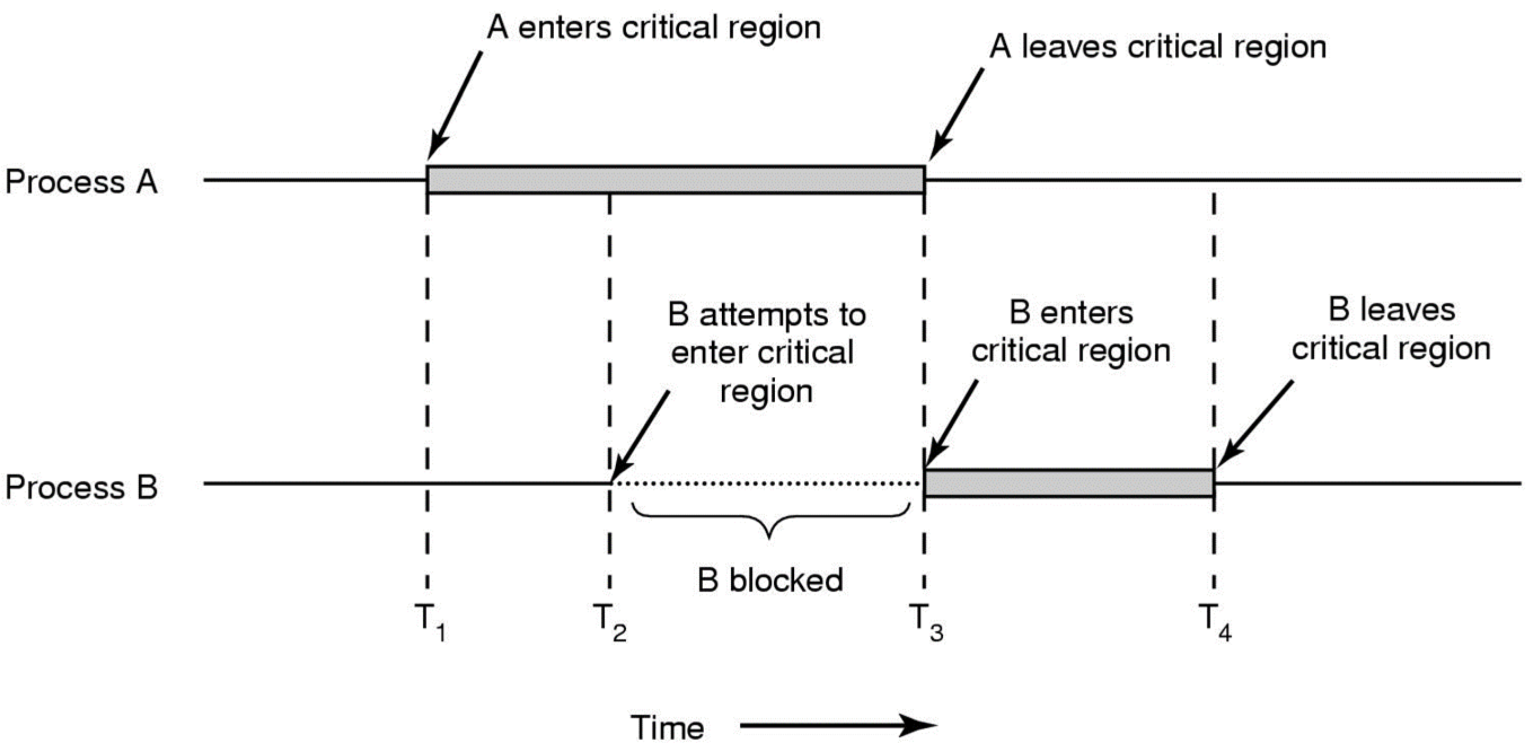
1. Mutex Lock / Mutual exclusion

- using mutex is to ***acquire*** a lock prior to entering a critical section, and to ***release*** it when exiting

```
do {
    Acquire Lock

        CRITICAL SECTION

    Release Lock

        REMAINDER SECTION

} while (TRUE);
```

- Mutex object is **locked** or **unlocked** by the process requesting or releasing the resource

1. Mutex Locks / Mutual exclusion



This type of mutex lock is called a **spinlock** because the process "spins" while waiting for the lock to become available.

2. Semaphore

- **Semaphore** was proposed by Dijkstra in 1965.

- is a technique to **manage concurrent processes by using a simple non-negative integer value** and **shared between threads / processes**.

- Only **three** atomic **operations** may be performed on a semaphore: **initialize**, **decrement**, and **increment**.

  - the decrement operation may result in the blocking of a process, and

  - the increment operation may result in the unblocking of a process.

This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

2. Semaphore

A semaphore S may be initialized to a **non-negative integer value**.

- is accessed only through two standard atomic operations: **wait()** and **signal().**

- **wait()** operation **decrements** the semaphore value

If the S<0, then the process executing the **wait()** is blocked. Otherwise, the process continues execution**.**

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

- **signal()** operation **increments** the semaphore value.

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

Using semaphores for solving CS Problem

- For ***n*** processes
- **Initialize** semaphore **S** to **1**
- Then only one process is allowed into CS (mutual exclusion)
- To allow **k** processes into CS at a time, simply initialize mutex to **k**

```
Process Pi:

do {
   wait(S);
    CRITICAL SECTION
   signal(S);
    RS
} while(true)
```

Semaphore

There are two main types of semaphores:

- **Counting Semaphore** – allow an arbitrary ***resource count***. Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.
- **Binary Semaphore** – similar to ***mutex lock***. It can have only two values: 0 and 1.

  Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.

COUNTING Semaphores

- The **semaphore S** is initialized to **the number of available resources**.
- Each process that uses a resource, it performs a **WAIT()** operation on the semaphore (thereby ***decrementing the number of available resources***).
- When a process releases a resource, it performs a **SIGNAL()** operation (***incrementing the number of available resources***).
- When **the count for the semaphore goes to 0**, **all resources are being used**. After that, processes that wish to use a resource will be block until the count becomes greater than 0.

BINARY Semaphores

A binary semaphore may only take on the values **0** and **1**.

1. A binary semaphore may be **initialized** to **1**.

2. The **WAIT()** operation (***decrementing***) checks the semaphore value.

    - If the value is **0**, then the process executing the **wait()** is **blocked**.
    - If the value is **1**, then the value is changed to 0 and the process **continues** execution.

3. The **SIGNAL()** operation (***incrementing)*** checks to see if any processes are blocked on this semaphore (semaphore value equals 0).

    - <u>If so</u>, then a process blocked by a **signal()** operation is **unblocked**.
    - If <u>no processes are blocked</u>, then the value of the semaphore is set to **1**.

Mutex vs. Binary semaphore

A **key difference** between the a **mutex** and a **binary semaphore** is that the process that locks the mutex (sets the value to **zero**) must be the one to unlock it (sets the value to **1**).

In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it. *(example in the tutorial)*

Same issues of semaphore

- ***Starvation*** - when the processes that require a resource are delayed for a long time. Process with high priorities continuously uses the resources preventing low priority process to acquire the resources.

- **Deadlock** is a condition where no process proceeds for execution, and each waits for resources that have been acquired by the other processes.

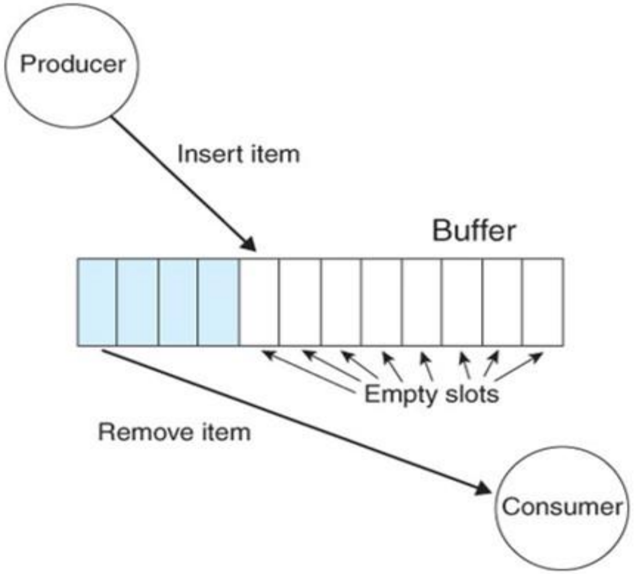
Classical Problems of Synchronization

- The Bounded-Buffer / Producer-Consumer Problem
- The Readers–Writers Problem
- The Dining-Philosophers Problem

Classic Problems of Synchronization

❑ The Bounded-Buffer / Producer-Consumer Problem

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```



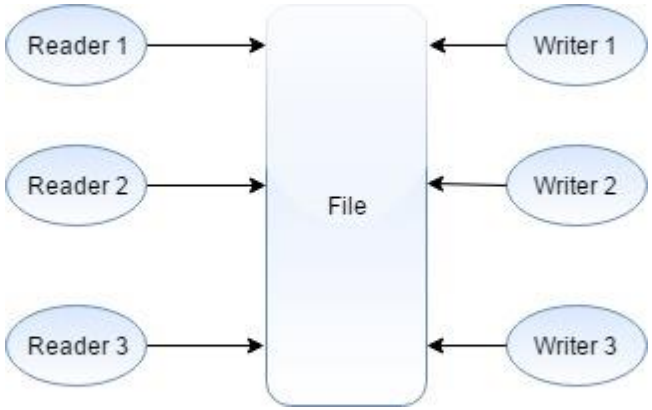
The **mutex** binary semaphore provides mutual exclusion for accesses to the buffer pool and is **initialized to the value 1**.

The *empty* and *full* semaphores count the number of empty and full buffers.

- the semaphore **empty is initialized to the value n**;
- the semaphore **full is initialized to the value 0**.

Classic Problems of Synchronization

❑ The Readers–Writers Problem



A data set is shared among a number of concurrent processes.

• Only one single writer can access the shared data at the same time, any other writers or readers must be blocked.

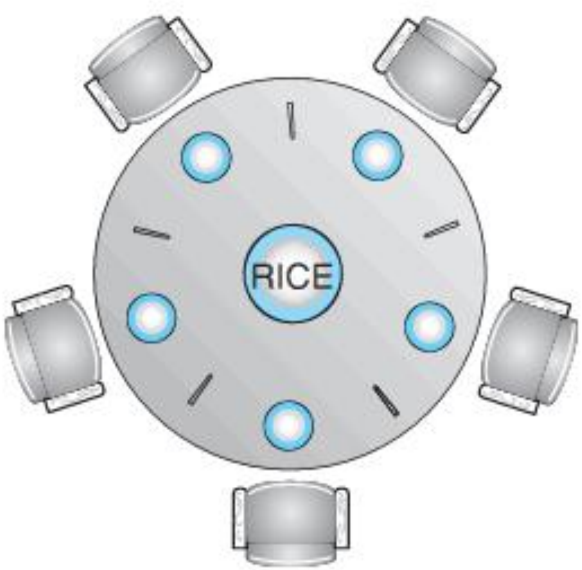
• Allow multiple readers to read at the same time, any writers must be blocked.

Solution: Acquiring a reader–writer lock requires specifying the mode of the lock: either *read* or *write* access.

Classic Problems of Synchronization

The Dining-Philosophers Problem

How to allocate several resources among several processes.



Several solutions are possible:

- Allow only **4** philosophers to be hungry at a time.
- Allow pickup only **if** both chopsticks are available. (Done in critical section)
- **Odd** # philosopher always picks up left chopstick 1st,
- **Even** # philosopher always picks up right chopstick 1st.

End of Lecture

- **Summary**

  - Background
  - The Critical-Section Problem
    - Peterson's Solution
    - Synchronization Hardware
    - Mutex Locks / Mutual exclusion
    - Semaphores
  - Classical Problems of Synchronization

- **Reading**
  - Textbook 9th edition, **chapter 5 of the module textbook**