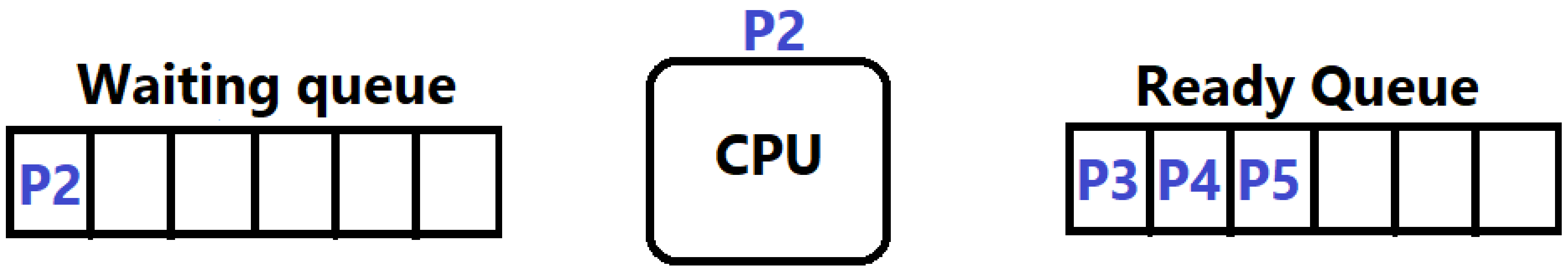
CSE308 Operating Systems

Concurrency & Synchronization

Dr S. Rajarajan

SASTRA

Concurrent Processes



Types of processes

- An **independent process** is one that executes independently without interacting or affecting other processes
- A **cooperating process** is one that can affect or be affected by other processes executing in the system
- Cooperating processes may get into conflicts

- Cooperating processes can
  - either **directly share a logical address space** that is, both code and data (**threads**)
  - or be allowed to **share data** only through files
  - or communicate through messages. (**pipes, shared memory, message queues**)
- In all the cases, concurrent access to **shared data** may result in **data inconsistency**.

- Shared resources
  - Files
  - IPC mediums – shared memory & message queue
  - Memory buffers
  - Variables
  - Hardware device e.g printer
  - Database

Concurrent Vs Parallel processing

- **Concurrent processing**
  - Under a uniprocessor, the scheduler keeps the CPU to switch from process to process to avoid the idle time of CPU.
- **Parallel processing**
  - Under a multi-core / multi-processor system either two separate processes or threads of a same process can be parallely executed on different **cores or processors** at a time.

- **concurrent or parallel execution** can contribute to issues involving the **integrity of data** shared by several processes.
- We will discuss **various mechanisms** to ensure the **orderly execution** of cooperating processes so that **data consistency** is maintained.

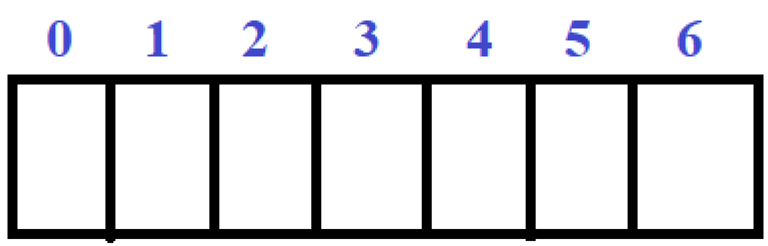
Bounded buffer producer-consumer problem

- **Producer** is a process which **produces data** and **puts them into a buffer.**
- **Consumer** is a process that **retrieves** and **consumes data** from the buffer.
- If **buffer is full** then **producer must not add** data.
- If **buffer is empty** then **consumer must not attempt to retrieve** data.

Producer

- Counter tells number of data in buffer & it is initialized to 0

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

    while (counter == BUFFER_SIZE);   //Buffer full
        /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```



If in is 6

(6+1) % 7 =0

Consumer

```
while (true) {
      while (counter == 0)      //Buffer empty
         ; /* do nothing */

      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;

      /* consume the item in next_consumed */
}
```

- The producer and consumer routines are **correct if** they are **executed sequentially**
- **But** they may function **incorrectly** when executed **concurrently**
- Suppose that the value of the variable **counter** is **5** and that the producer and consumer processes **concurrently execute** the statements **“counter++”** and **“counter--”.**
- The **correct result** is **counter = 5**, which is generated correctly if the producer and consumer execute separately
- But under concurrent execution of the two statements, the value of the variable counter may be **4, 5, or 6.**

We can show that the value of counter may become incorrect as follows

Counter 6 5

Producer

While(counter==
BUFFER_SIZE)

Counter++;

Blocked

Register

6 5

Consumer

While(counter==0)

Counter--;

Register

5 4

Context switch to Consumer

Context switch to Producer

- Note that the statement “counter++” may be implemented in machine language (on a typical machine) as follows:
- *register1 = counter*
- *register1 = register1 + 1*
- counter = *register1*

- *Similarly, the statement* “counter--” is implemented as follows:
- *register2 = counter*
- *register2 = register2 − 1*
- counter = *register2*
- Even if register1 and register2 are the same physical register (an accumulator, say), remember that the **contents of this register** will be **saved and restored** by context switching.

- The concurrent execution of “counter++” and “counter--” is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order.
- One such interleaving is the following:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

- Notice that we have arrived at the incorrect state “**counter == 4**”, indicating that four buffers are full, when, in fact, **five buffers** are full.
- If we reversed the order of the statements at *T4 and T5, we would arrive at the incorrect state* “**counter == 6**”.
- We would arrive at this incorrect state because we allowed both processes to **manipulate** the variable counter **concurrently.**

Multiprocessor System

Counter 5

Producer

While(counter==
BUFFER_SIZE)

Counter++;

Register

6

Consumer

While(counter==0)

Counter--;

Register

4

Race condition

- A **race condition** or **race hazard** is the condition of an software system where the system's substantive behaviour is dependent on the sequence or timing of other **uncontrolled events**, leading to **unexpected or inconsistent results.**
- Situations where several processes access and **manipulate** the same **data concurrently** and the **outcome** of the execution depends on the **particular order** in which the access takes place leads to a **race condition.**
- To guard against the race condition, we need to ensure that **only one process** at a time can be manipulating the variable counter.
- To make such a guarantee, we require that the **processes be synchronized** in some way

Sequential access

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Concurrent / Parallel access

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

What is synchronization ?

- Situations such as the one just described occur frequently in operating systems as different parts of the system **manipulate same resources**.
- The growing importance of **multi-core systems** has brought an increased emphasis on developing **multithreaded** applications.
- In such applications, **several threads** —which are quite possibly **sharing data** —are running in **parallel** on different processing cores
- When two processes want to access a **shared resource** there should be certain **discipline**, **monitoring** and **regulation** imposed while accessing that resource.

Starting point - Resource sharing

Synchronization – enforcing order on resource usage

When no Synchronization enforced

Consequence - conflicts

Rajendran

Limits on number of simultaneous users for a resource

Race condition - example

- Suppose that two processes, **P1** and **P2,** share the **global variable a**.
- At some point in its execution, **P1** updates **a=1**, and at some point in its execution, **P2** updates **a= 2.**
- Thus, the two tasks are **in a race** to write variable
- In this example the **"looser"** of the race (the process that **updates last**) determines the **final value of a.**

Example 1

int a

Process 1
a=1

Process 2
a=2

- If the sequence of execution of the two processes is **P1 followed by P2** then 'a' will be modified as
- P1: a=1 & P2: a=2
- So the final value of **'a' will be 2**
- If the sequence of execution of the two processes is **P1 followed by P2** then 'a' will be modified as
- P2: a=2 & P1: a=1
- Then the final value of **'a' will be 1**

Example 2

b=1 c=2

Process P3

b = b + c

Process P4

c = b + c

- Though the two processes update different variables still the final values of the b & c depends on the order in which two processes are executed.
- If the order is **P3 followed by P4** then b= 3, c= 5
- If the order is **P4 followed by P3** then b= 4, c=3

- Count is a global variable initialized with ‘0’

P1

```
If( Count == 0)
{
        ........
        Count++;
}
Count=0
```

P2

```
If( Count == 0)
{
        ........
        Count++;
}
Count=0
```

- Under **sequential execution**, only one process could increment Count and so it will become 1.
- But under **concurrent or parallel processing** count might become **2**

P1

```
If( Count == 0)
{
        ........
        Count++;
}
```

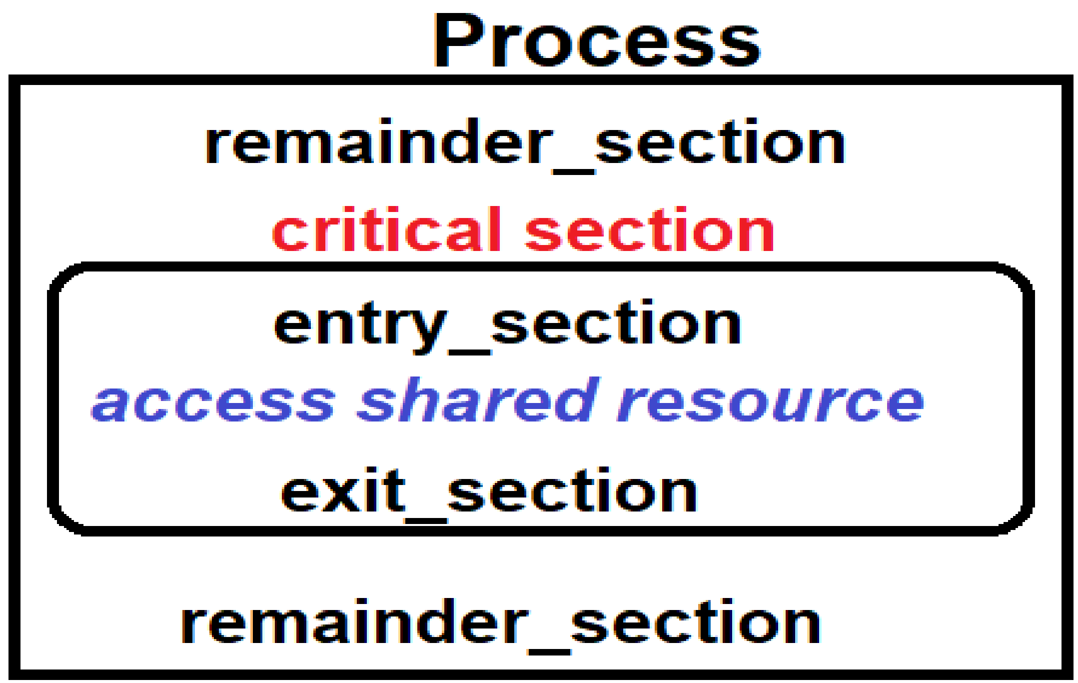
P2

```
If( Count == 0)
{
        ........
        Count++;
}
```

The Critical-Section

- Consider a system consisting of n processes {P0, P1, ..., Pn−1}.
- Each process has a **segment of code**, called a **critical section,** in which the process may be **changing common variables**, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is **executing in its critical section**, no other process is allowed to execute in its critical section.
- **That is, no two processes can be executing in their critical sections at the same time.**


यूनियन बैंक
एटीएम
ऑफ इंडिया
ATM
Union Bank
Of India
Chaayos
SHOETREE
PUBLIC
SOCIALNEWS.XYZ


Process
remainder_section
critical section
entry_section
access shared resource
exit_section
remainder_section

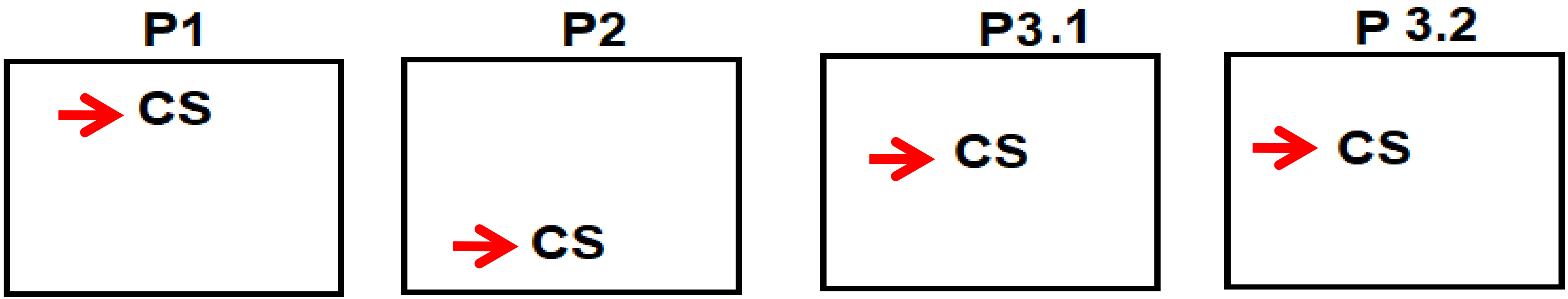
```
while (true) {
      /* produce an item in next_produced */

      while (counter == BUFFER_SIZE);
           /* do nothing */

      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
CS    counter++;
}
```

```
while (true) {
      while (counter == 0)
         ; /* do nothing */

      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
CS    counter--;

      /* consume the item in next_consumed */
}
```

Requirements

- The ***critical-section problem is to design a*** protocol that coordinates the cooperative processes.
- Each process **must request permission** to enter its critical section.
- **Entry section-** The section of code that begins to access the shared resource
- **Exit section-** Section of code following the shared resource usage
- The remaining code is the **remainder section**

E.g. Critical section for resource R1



Strategy

- ***Critical Resource:-*** A resource that may lead to conflict between processes.
  - It can be a variable or file or hardware device or memory region
- **Critical Section**:-The portion of the program/process that accesses the critical resource
- **Mutual exclusion:-** Only one process at a time to be allowed to enter into its critical section

Critical section conditions

- A solution to the critical-section problem must satisfy the following three 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 in its critical section then an intending process should be able to enter provided it is the only process entering. This selection cannot be postponed indefinitely.
- **3. Bounded waiting.** There exists a bound, or limit, 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 (***no starvation***)

Kernel level race conditions

- At a given point in time, many kernel-mode processes may be active in the operating system.
- As a result, the code implementing an operating system (**kernel code)** is subject to several possible race conditions.
- E.g file management – two kernel modules updating same file
- Two general approaches are used in operating systems:
  preemptive kernels and **non-preemptive kernels.**
- A **preemptive kernel** allows a process to be preempted while it is running in kernel mode.
- A **non-preemptive kernel** does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

- Obviously, a **non-preemptive kernel** is essentially **free from race conditions** on kernel data structures, as **only one process is active** in the kernel **at a time** (no context switching among concurrent processes).
- We cannot say the same about **preemptive kernels**, so they must be **carefully designed** to ensure that shared kernel data are free from race conditions.
- **Preemptive** kernels are especially **difficult** to design for **SMP architectures**, since in these environments it is possible for **two kernel-mode processes** to run simultaneously on **different processors**

1. Peterson's Solution

- A classic **software-based** solution to the critical-section.
- Because of the way **modern computer architectures** perform machine-language instructions- **load and store**, there are **no guarantees** that Peterson's solution will work correctly on such architectures.
- However, it provides a **good algorithmic description** of solving the **critical-section problem** and illustrates some of the **complexities involved** in designing software that addresses the requirements of **mutual exclusion, progress, and bounded waiting**

- Peterson's solution is **restricted to two processes** that alternate execution between their critical sections and remainder sections.
- The processes are numbered ***P_0 and P_1***.
- *For convenience, when presenting P_i , we use Pj to* denote the other process; that is, j equals 1 – i.
- Peterson's solution requires the two processes to share two data items:
  - **int turn;**
  - **boolean flag[2];**
- The variable **turn indicates whose turn** it is to enter its critical section.
- That is, if **turn == i**, then process ***Pi is allowed*** *to execute in its critical section.*

- The **flag array** is used to **indicate** if a **process wants** to enter its **critical section**.
- For example, if **flag[i] is true**, this value indicates that ***Pi is willing*** *to enter* its critical section.
- To enter the critical section, P_i,
  - *first* ***sets flag[i]*** *to be* ***true***
  - *and* then sets **turn to the value j**, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- If **both processes try** to enter at the same time, turn will be set to **both i and j** at roughly the same time.
- Only one of these assignments will last; the other will occur but will be **overwritten immediately**.
- The **eventual value of turn determines** which of the two processes is allowed to enter its critical section first.

Peterson’s algorithm

```
do {

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

If condition is false then process enters the critical section

Both P0 & P1 wants to enter CS

	0	1
Flag	false / true	false / true
turn	1 / 0	

```
P0:       flag[0] = true;
P0_gate:  turn = 1;
          while (flag[1] == true && turn == 1); False
                                    // busy wait
          // critical section
          ...
          // end of critical section
          flag[0] = false;
```

```
P1:       flag[1] = true;
P1_gate:  turn = 0;
          while (flag[0] == true && turn == 0); False / True
                                    // busy wait
          // critical section
          ...
          // end of critical section
          flag[1] = false;
```

P1 in CS & P0 wants to enter

	0	1
Flag	false true	true
turn	0 1	

```
P0:      flag[0] = true;
P0_gate: turn = 1;
         while (flag[1] == true && turn == 1); True
                                   // busy wait

          // critical section

         ...
         // end of critical section
         flag[0] = false;
```

```
P1:      flag[1] = true;
P1_gate: turn = 0;
         while (flag[0] == true && turn == 0);
                                   // busy wait

         // critical section

         ...
         // end of critical section
         flag[1] = false;
```

Only P0 wants to enter

	0	1
Flag	false true	false
turn	0 1	

```
P0:      flag[0] = true;
P0_gate: turn = 1;
         while (flag[1] == true && turn == 1);  False
                                   // busy wait
  →       // critical section
         ...
         // end of critical section
         flag[0] = false;
```

```
P1:      flag[1] = true;
P1_gate: turn = 0;
         while (flag[0] == true && turn == 0);
                                   // busy wait
         // critical section
         ...
         // end of critical section
         flag[1] = false;
```

Wrong turn

Both P0 & P1 wants to enter CS

	0	1
Flag	false / true	false / true
turn	0 / 1	

```
P0:       flag[0] = true;
P0_gate:  turn = 0;
          while (flag[1] == true && turn == 1);   False
                                   // busy wait
   →  // critical section
          ...
          // end of critical section
          flag[0] = false;
```

```
P1:       flag[1] = true;
P1_gate:  turn = 1;
          while (flag[0] == true && turn == 0);   False
                                   // busy wait
   →  // critical section
          ...
          // end of critical section
          flag[1] = false;
```

Wrong turns

Both P0 & P1 wants to enter CS

	0	1
Flag	false / true	false / true
turn	0 / 1	

```
P0:      flag[0] = true;
P0_gate: turn = 0;
         while (flag[1] == true && turn == 0);   True
                                   // busy wait
          // critical section
         ...
         // end of critical section
         flag[0] = false;
```

```
P1:      flag[1] = true;
P1_gate: turn = 1 ;
         while (flag[0] == true && turn == 1);   True
                                   // busy wait
         // critical section
         ...
         // end of critical section
         flag[1] = false;
```

1. Situation 1: Currently turn is 0 , P1 has no interest in CS & P0 wants to enter CS ?

   That means P0 changes Flag[0]=true, changes turn=1 and runs while (Flag[1] && turn =1). But since P1 has no interest, its Flag[1] is false. So P0 breaks while loop and enters CS

2. Situation 2: Currently turn is 0 , P1 has entered CS & P0 wants to enter CS ?

   That means P0 changes Flag[0]=true, changes turn=1 and runs while (Flag[1] && turn =1). But since P1 has interest, its Flag[1] is true. Which means P1 is already in its CS. So P0 struck in while loop .

3. Situation 3: Currently turn is 0 , P1 has interest in CS and has changed Flag[1]= true and it about to change turn=1. Meanwhile P0 changes Flag[0]=true and it is about to change turn=1. Which will enter CS ?

   This is a typical race condition. Depending upon who changes turn at last, its value will be either 0 or 1. If turn becomes 0 then P0 enters CS else P1 enters.

- We now prove that this solution is correct. We need to show that:
- 1. Mutual exclusion is preserved.
- 2. The progress requirement is satisfied.
- 3. The bounded-waiting requirement is met.

Mutual exclusion proof

- Both P0 and P1 can never be in the critical section at the same time
- If both processes are in their critical sections then the state must satisfy **flag[0] and flag[1] are true** and **turn = 0 and turn = 1**.
- **No state** can satisfy both **turn = 0 and turn = 1**, so there can be no state where both processes are in their critical sections.

Progress requirement proof

- To prove properties 2 and 3,we note that a process ***Pi has to be prevented*** *from* entering the critical section then it is only if gets stuck in the while loop with the condition **flag[j] == true and turn == j;** this loop is the only one possiblity.
- If ***Pj is not*** **ready** to enter the critical section, then **flag[j] == false**, and *Pi can enter its* critical section.
- If ***Pj has set flag[j] to true*** *and is also executing in its while* statement, then either **turn == i**.
- The **Pi** changes **turn to j** and **Pj enters CS**

- If Pj enter critical section then when it *exits its critical section, it will* ***reset flag[j] to false and also set turn to i*** *allowing Pi to* enter its critical section.
- So progress is guaranteed.

Bounded waiting proof

- Bounded waiting means that **the number of times a process is bypassed** by another process after it has indicated its desire to enter the critical section is bounded by a function of the number of processes in the system.
- In Peterson's algorithm, a process will **never wait longer than one turn** for entrance to the critical section.

2. Synchronization Hardware

- Software-based solutions such as Peterson’s are **not guaranteed** to work on **modern computer architectures.**
- we explore several more solutions to the critical-section problem using techniques ranging from **hardware to software-based APIs.**
- All these solutions are based on the premise of **locking —** that is, protecting critical regions through the use of locks.

2.1 Interrupt disabling

- Critical-section problem could be solved easily in a **single-processor** environment if we could **prevent interrupts** from occurring while a **shared variable** was being modified – ***non-preemptive.***
- In this way, we could be sure that the current sequence of instructions would be allowed to **execute in order** without **preemption** and **context switching**.
- This is often the approach taken by **non-preemptive kernels.**
- Unfortunately, this solution is not as feasible in a **multiprocessor** environment

Pseudo-Code

```
while (true) {
/* disable interrupts */;

/* critical section */;

/* enable interrupts */;

/* remainder */;

}
```

2.2 Special hardware instructions

- Many modern computer systems provide special hardware instructions that allow us either
  - to test and modify the content of a word
  - or to swap the contents of two words **atomically**—that is, as one **uninterruptible** unit.
- We can use these special instructions to solve the critical-section problem in a relatively simple manner
- E.g. **test and set()** and **compare and swap()**
- instructions.

Compare & Swap Instruction

Checks a local value ***word** against a **testval**, it they are same, replaces with a newval

```
int compare_and_swap(int *word, int testval, int newval)
{       int oldval;
        oldval = *word;

        if (oldval == testval) *word = newval;

        return oldval;

}
```

```
void p(){ while((c_a_s(bolt,0,1)==1); CS }
```

The process that finds Bolt as 0 enters into the critical section

```
int Compare_and_Swap(int *word, int testval, int newval)
                           0            0          1
{
        int oldval;
        oldval = *word;
        if ( oldval == testval ) *word = newval;   1
             0         0
        return Oldval;
               0;
}
const int n= /* number of processes */
int bolt;
void P( In t i )
           1
{
        while(    Compare_and_Swap(bolt, 0, 1)==1);
                          0          0
              /* do nothing */
        /* Critical Section */
        bolt =0 ;
        /* remaining code */
}
void main()
{
        bolt =0 ;
        parbegin(P(1), P(2), ...P(n));
}
```

bolt 0 1

```
int Compare_and_Swap(int *word, int testval, int newval)
{                                  1          0          1
        int oldval;
        oldval = *word;      1
        if ( oldval 1 == testval 0 ) *word = newval;
        return    1; oldval;
}
```

bolt $\boxed{1}$

```
const int n= /* number of processes */
int bolt;
void P(  Int 2 i  )
{
        while(   Compare_and_Swap( bolt 1, 0, 1)==1) 1 ;
                /* do nothing */
        /* Critical Section */
        bolt =0  ;
        /* remaining code */
}
void main()
{
        bolt = 0;
        parbegin(P(1), P(2), ...P(n));
}
```

test and set()

- The important characteristic of this instruction is that it is **executed atomically.**
- Thus, if two test and set() instructions are executed simultaneously (each on a different CPU), they will be **executed sequentially** in some **arbitrary order**.
- If the machine supports the test and set() instruction, then we can implement **mutual exclusion** by declaring a **boolean variable lock, initialized to false**.

Lock false true

```
boolean test and set(boolean *target) {
    boolean rv = *target;     false true
    *target = true;
    return rv;   false true
}
do { while (   test and set(&lock  )); /* do nothing */   false true
/* critical section */
lock = false;
/* remainder section */
} while (true);
```

- Although these algorithms satisfy the mutual-exclusion requirement, they **do not satisfy the bounded-waiting** requirement.
- We present another algorithm using the test and set() instruction that satisfies all the critical-section requirements.

```
do {
        waiting[i] = true;
        key = true;
        while (waiting[i]  &&  key)
                key = test and set(&lock);    false
        waiting[i] = false;
        /* critical section */
        j = (i + 1) % n;   / To give the chance to next waiting process
        while ((j != i) && !waiting[j])   / J is not waiting for CS
                j = (j + 1) % n;    / Continue search
        if (j == i)  / Unable to find a process
                lock = false;
        else // Next process 'j' to enter CS found
                waiting[j] = false;
} while (true);
```

Mutual exclusion proof

- Process ***Pi can enter its critical*** **section** only if either **waiting[i] == false** or **key == false**.
- The value of **key can become false** only if the **test and set() is executed**.
- The **first process** to execute the **test and set()** will **find key == false**; all others must wait.
- The variable **waiting[i]** can become **false** only if another process leaves its critical section; **only one waiting[i]** is **set to false**, maintaining the mutual-exclusion requirement

Progress proof

- The arguments presented for mutual exclusion also apply here.
- Since a process exiting the critical section either **sets lock to false** or **sets waiting[j] to false**.
- Both allow a process that is waiting to enter its critical section to proceed.

Bounded-waiting proof

- When a process leaves its critical section, it scans the array waiting in the cyclic ordering (*$i + 1, i + 2, \ldots, n - 1, 0, \ldots, i - 1$*).
- *It designates the* ***first process*** *in this* ordering that is in the entry section (**waiting[j] == true**) as **the next one to enter** the critical section.
- Any process waiting to enter its critical section will thus do so within *$n - 1$ turns.*

3. Mutex Locks

- Hardware-based solutions are **complicated** as well as generally **inaccessible** to application **programmers**.
- Instead, ***operating-systems*** designers build software tools to solve the **critical-section problem**
- The simplest of these tools is the **mutex lock** (the term ***mutex*** *is short for* ***mutual exclusion****)*.
- That is, a **process must acquire the lock** before entering a critical section; it **releases the lock** when it exits the critical section.

- The **acquire() function** acquires the lock, and the **release() function** releases the lock.
- mutex lock has a **boolean variable** available whose value indicates if the **lock is available or not**.
- If the **lock is available**, a call to **acquire() succeeds**, and the lock is then considered unavailable.
- A process that attempts to acquire an **unavailable** lock is **made to wait until the lock** is released

- The definition of acquire() is as follows:

available $\boxed{\text{true}}$ $\boxed{\text{false}}$

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

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

Applying mutex lock for CS

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

- Calls to either acquire() or release() must be performed **atomically.**
- Thus, mutex locks are often implemented using one of the **hardware mechanisms** described

Drawback - Busy waiting

- The main disadvantage of the implementation given here is that it requires **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().
- In fact, this type of mutex lock is also called a **spinlock because the process** “spins” while waiting for the lock to become available.
- We see the **same issue with** the code examples illustrating the **test and set()** instruction and the **compare and swap()** instruction.
- Busy waiting **wastes CPU cycles** that some other process might be able to use productively.

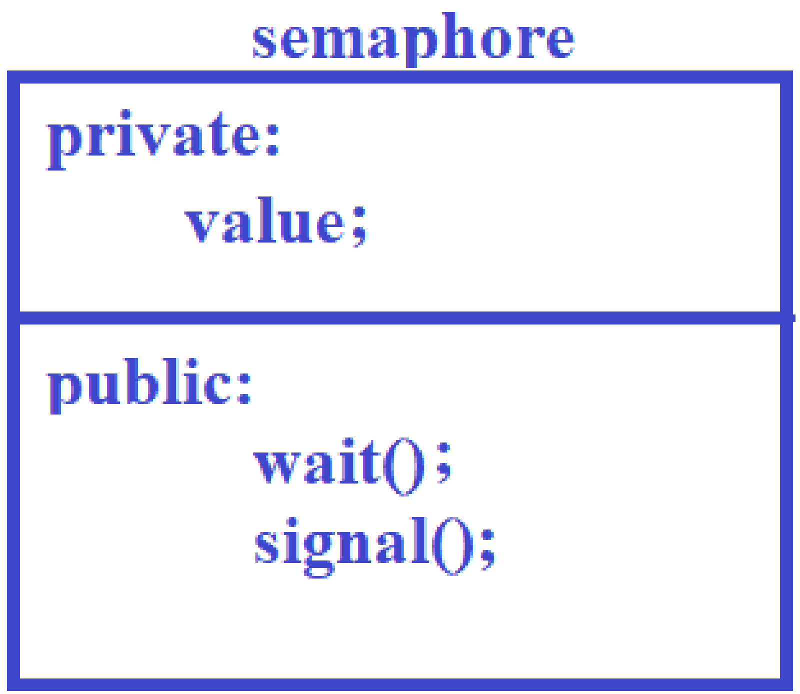
- **Spinlocks** do have an **advantage**.
- **No context switch is required** when a process must wait on a lock, and a context switch may take considerable time.
- Thus, when **locks are expected** to be held for **short times**, spinlocks are useful.
- They are often employed on **multiprocessor systems.**
- One thread can **“spin” on one processor** while another **thread performs its critical section** on another **processor**.

4. SEMAPHORE

Semaphores

- A more robust tool that can behave similarly to a mutex lock but **also provide more sophisticated ways** for processes to synchronize their activities.
- A **semaphore S** is an **integer** variable
  - that can be **initialized**, **incremented** and **decremented.**
  - and is accessed only through two standard **atomic** operations: **wait() (semWait)**and **signal() (semSignal).**
- All modifications to the integer value of the semaphore in the **wait() and signal()** operations must be **executed indivisibly**.
- That is, when one process modifies the semaphore value**, no** other process can **simultaneously modify** that same semaphore value

- Semaphore can be treated **like an object** with s.value as its private data member and wait and signal as its public methods



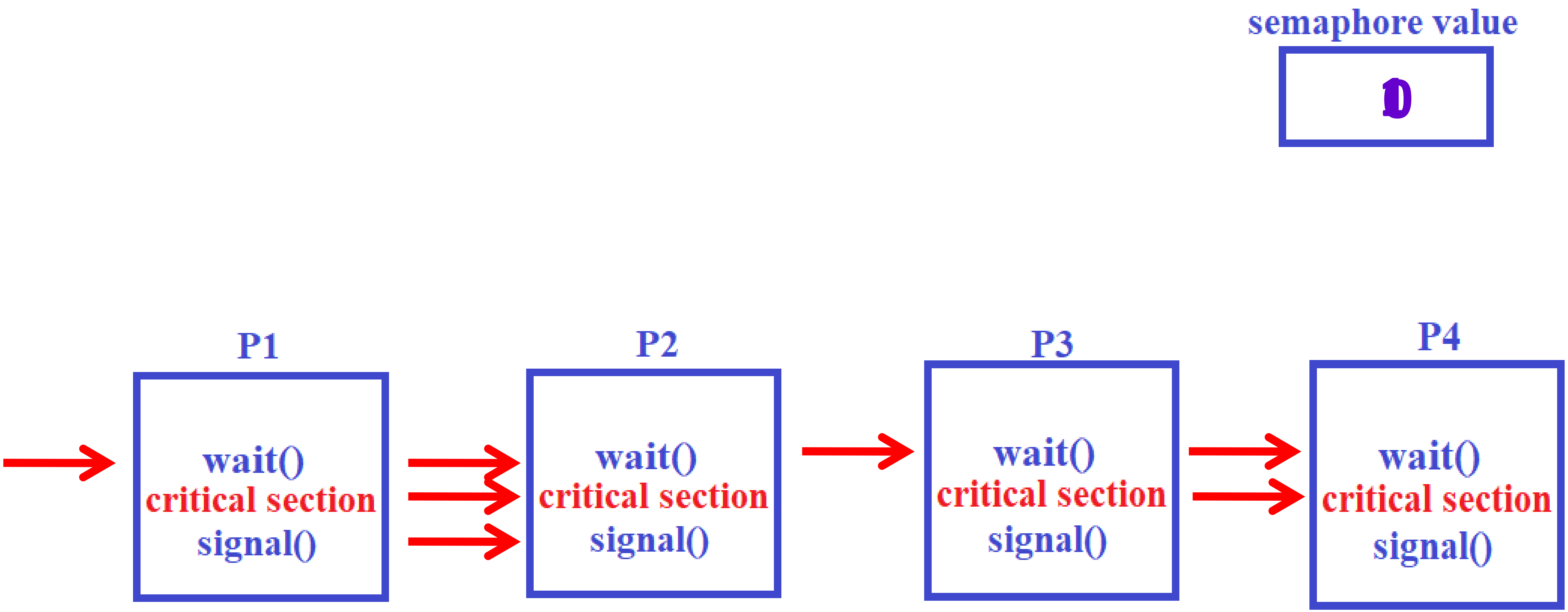
- The definition of wait() is as follows:
- S=1

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

The definition of signal() is as follows:

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

- Each process that wishes **to use a resource** performs a **wait() operation** on the semaphore (thereby decrementing the count).
- When a process **leaves critical section**, it performs a **signal()** operation (incrementing the count).
- As long as the count for the semaphore goes to >=0, process invoking wait is allowed to enter critical section
- After count becomes <0, processes that wish to use a resource will busy wait until the count becomes greater than 0.


semaphore value
0
P1
wait()
critical section
signal()
P2
wait()
critical section
signal()
P3
wait()
critical section
signal()
P4
wait()
critical section
signal()

Blocking instead of busy waiting

- Recall that the implementations of **mutex, peterson, compare & swap, test & test** are all suffering from **busy waiting**.
- The definitions of the **wait() and signal()** semaphore operations just described present **the same problem**.
- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows:
  - When a process **executes the wait()** operation and finds that the semaphore value is not positive, it must wait.
  - However, 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**.
- Then control is transferred to the **CPU scheduler**, which **selects another process** to execute – **context switch**.

- A process that is **blocked**, 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**.
- To implement semaphores under this definition, we define a semaphore as follows:

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

- Now, the **wait()** semaphore operation can be defined as:

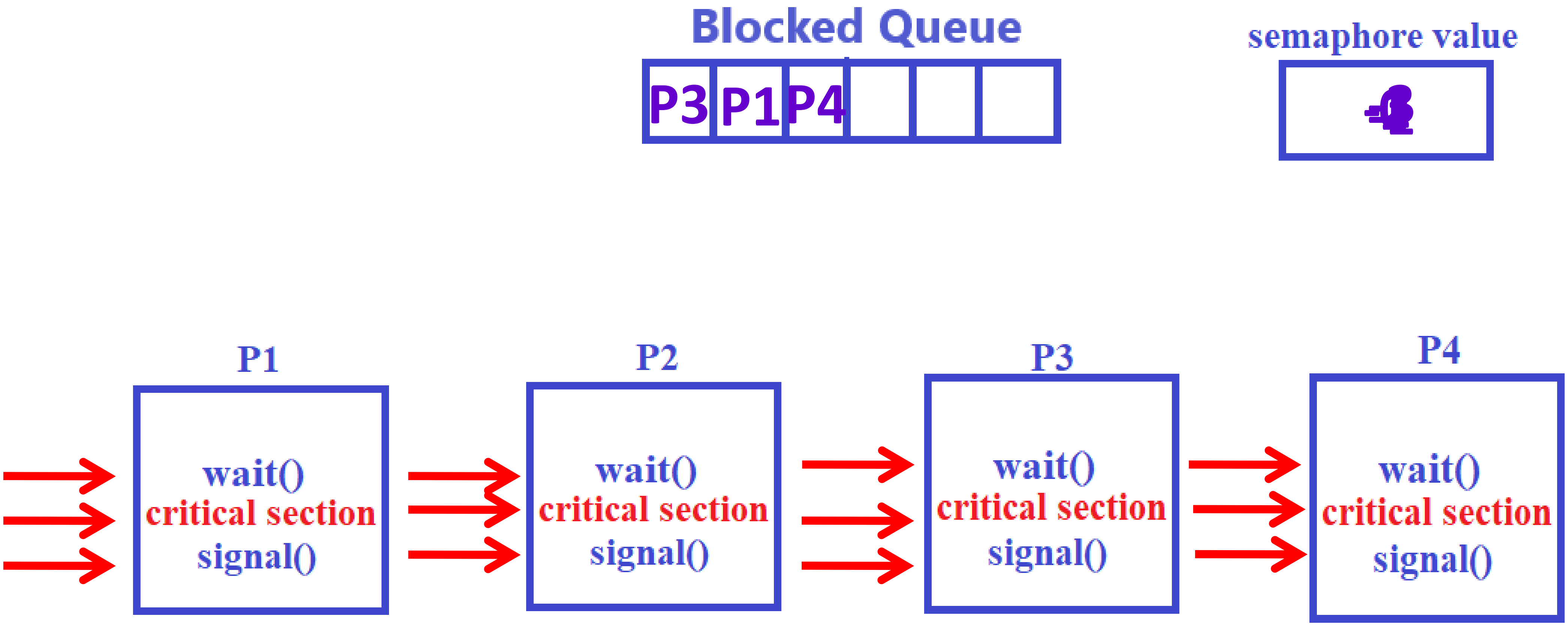
```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) { add this process to S->list;
         block();
   }
}
```

- and the **signal()** semaphore operation can be defined as

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

- The **block()** operation **suspends the process** that invokes it.
- The **wakeup(P**) operation **resumes the execution** of a blocked process P.
- These two operations are provided by the operating system as **basic system calls**

- In this implementation, semaphore values **may be negative**.
- If a semaphore value is **negative**, its **magnitude** is the **number of processes waiting** on that semaphore.
- The list of waiting processes can be easily implemented by a link field in each **process control block (PCB).**
- Each semaphore contains an integer value and a pointer to a list of PCBs.
- One way to add and remove processes from the list so as to ensure bounded waiting is to use a **FIFO queue**, where the semaphore contains both head and tail pointers to the queue.
- In general, however, the list can use any queueing strategy



Blocked Queue
P3
P1
P4
semaphore value
P1
wait()
critical section
signal()
P2
wait()
critical section
signal()
P3
wait()
critical section
signal()
P4
wait()
critical section
signal()

Sequencing instructions using Semaphores

- We can also use semaphores **to solve various other synchronization** problems.
- For example, consider two concurrently running processes: ***P1 with a statement S1*** *and* ***P2 with a statement S2***.
- *Suppose we require that* ***S2 be executed only*** **after** ***S1 has completed***.
- *We can implement this scheme readily by letting P1* and *P2 share a common semaphore* ***synch, initialized to 0***.

P2 should execute s2 only after P1 completes s2

P1

```
s1;
s2;
```

P2

```
s1;
s2;
```

synch = 0

P1

```
s1;
signal(synch)
s2;
```

P2

```
s1;
wait(synch)
s2;
```

- Initialize ***synch =0***
- ***In process P1****, we* insert the statements:

S1;

signal(synch);

- **In process *P2****, we insert the statements*

wait(synch);

S2;

- Because synch is initialized to 0, *P2 will execute S2 only after P1 has invoked* signal(synch), which is after statement *S1 has been executed.*

Semaphore Types

- Operating systems often distinguish between counting and binary semaphores
- The value of a **counting semaphore** can range over an unrestricted domain
- The value of a **binary semaphore** can range only between 0 and 1

Advantage of Counting semaphore

- Counting semaphores can be used to control access to a given resource consisting of a **finite number of instances**.
- The semaphore is **initialized to the number of resources** available.
- Each process that wishes to use a resource **performs a wait() operation** on the semaphore (thereby decrementing the count).
- When a process releases a resource, it **performs a signal()** operation (incrementing the count).
- When the count for the **semaphore goes to 0**, all resources are being used.

- After that, processes that wish to use a resource **will block** until the count becomes greater than 0.
- Whereas **binary semaphores** can only be used when the **number of instances of the resource is exactly 1.**
- With counting semaphore, we may exactly **know number of processes waiting** for the critical section.

Ensuring Atomicity

- It is critical that semaphore operations be executed atomically.
- We must guarantee that **no two processes can execute wait() and signal() operation**s on the same semaphore at the same time.
- This is a critical-section problem; and in a **single-processo**r environment, we can solve it by simply **disabling interrupts** during the time the wait() and signal() operations are executing.
- This scheme **works in a single-processor** environment because, once interrupts are inhibited, instructions from **different processes cannot be interleaved**.
- Only the currently running process executes until interrupts are **re-enable**d and the scheduler can regain control.

- In a **multiprocessor environment**, interrupts must be **disabled on every processor**.
- Disabling interrupts on every processor can be a **difficult task** and furthermore can seriously **diminish performance**.
- Therefore, SMP systems must provide alternative locking techniques

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- When such a state is reached, these processes are said to be **deadlocked.**
- We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

P_0 P_1

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

- Another problem related to deadlocks is **indefinite blocking or starvation,** a situation in which processes wait indefinitely within the semaphore.
- Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Semaphore Primitives

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

Figure 5.3 A Definition of Semaphore Primitives

s.count -1

P1

SemWait()
Critical_Section
SemSignal()

P2

SemWait()
Critical_Section
SemSignal

Binary Semaphore Primitives

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
               /* place this process in s.queue */;
               /* block this process */;
     }
}
void semSignalB(semaphore s)
{
     if (s.queue is empty())
          s.value = one;
     else {
               /* remove a process P from s.queue */;
               /* place process P on ready list */;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

Priority Inversion

- A scheduling challenge arises when a **higher-priority process** needs to **access a resource** that is currently being accessed **by a lower-priority process**
- Since **shared resources** are typically protected with a **lock**, the **higher-priority process** will have to **wait** for a **lower-priority** one to finish with the resource
- The situation becomes more complicated **if the lower-priority process is preempted** in favor of another process with a higher priority.

- As an example, assume we have three processes—L, M, and H—whose priorities follow the **order L < M < H.**
- Assume that process **H requires resource R**, which is currently being **accessed by process L.**
- Ordinarily, **process H would wait for L** to finish using resource R.
- However, now suppose that process **M becomes runnable**, thereby **preempting process L.**
- Indirectly, a process with a **lower priority—process M—has affected** how long process **H must wait** for L to relinquish resource R.
- This problem is known as **priority inversion**

A high priory Ambulance vehicle is forced to wait behind the low priority auto-rickshaws due to traffic

Priority-inheritance protocol

- Typically these systems solve the problem by implementing a **priority-inheritance protocol.**
- According to this 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.
- When they are finished, their **priorities revert** to their original values
- In the example above, a priority-inheritance protocol would allow process *L to temporarily inherit the priority of process H, thereby preventing process Mfrom preempting its execution*

Classical Problems of Synchronization

- Classical problems that involve critical section problem:
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Producer - Consumer Problem

- **General Situation:**
  - One or more producers are **generating data** and **placing these** in a **buffer**
  - A **single consumer** is **taking items** out of the buffer one at time
  - **Only one producer or consumer** may access the buffer at any one time.
- **Conditions:**
  - Either the producer or the consumer should access the buffer at a time - ***Mutual exclusion***
  - Ensure that the ***Producer*** can’t add data into ***full buffer*** and ***consumer*** can’t remove data from ***empty buffer***.

Two variants of the P-C problem

- **Unbounded buffer**
  - Required conditions – ME, Empty buffer
- **Bounded buffer**
  - Required conditions – ME, Empty buffer, Full buffer

Solution for Bounded buffer problem

- In our problem, the producer and consumer processes share the following data structures:
- int n;
- **semaphore mutex = 1;** // to enforce mutual exclusion
- **semaphore empty = n;** // to prevent producer when buffer full
- **semaphore full = 0 ;** // to prevent consumer when buffer empty

mutex 1

Consumer

empty 5 full -1

```
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);
```

Consumer blocked on queue of Full semaphore mutex 1

Producer

empty 4 full 0

```
do{
    wait(empty) ;
    wait(mutex);
    ..................
    /* add an item into the buffer */
    ..................
    signal(mutex);
    signal(full );
}while(true);
```

Consumer unblocked from of Full semaphore queue

mutex 0 1

empty 4 5 full 0

Consumer

```
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);
```

The Readers–Writers Problem

- Suppose that a **database** or **file** is to be **shared** among several concurrent processes.
- Some of these processes may **want only to read** the database, whereas others may **want to update** (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as ***readers*** and to the latter as ***writers.***
- if **two readers access** the shared data simultaneously, **no adverse effects** will result.
- However, **if a writer** and some other process (either a reader or a writer) access the database simultaneously, **chaos may ensue**.

- To ensure that these difficulties do not arise, we require that the **writers have exclusive access** to the shared database while writing to the database.
  – Mutual exclusion for writer
  – No mutual exclusion among readers
- This synchronization problem is referred to as the **readers–writers problem.**

Producer & Consumer Vs Reader & Writer

P- C	R-W
The buffer is filled by producer and it is emptied by consumer	Reader simply reads the values of a file and the writer updates the values
Mutual exclusion needed among any two processes whether two producers or two consumers or a producer and a consumer	No mutual exclusion required among readers since they only read, whereas writer requires exclusive access.
Two variants – bounded and unbounded buffer	Two variants – reader priority and writer priority

- **Variants**
  - ***first readers–writers problem,*** requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object.
  - **second readers –writers problem** requires that, once a writer is ready, that writer perform its write as soon as possible.
- A solution to either problem may result in **starvation**
- In the first case, **writers may starve**; in the second case, **readers may starve**.
- For this reason, other variants of the problem have been proposed

- In the solution to the first readers–writers problem, the reader processes share the following data structures:
- **semaphore rw_mutex = 1;**
- **semaphore mutex = 1;**
- **int read count = 0;**
- The semaphore **rw mutex** is common to both reader and writer processes
- The **mutex** semaphore is used to ensure mutual exclusion when the variable read count is updated
- The **read_count** variable keeps track of how many processes are currently reading the object.

Readers have Priority

mutex – to prevent multiple readers accessing read_count variable
rw_mutex – to enforce ME among readers and writer

READER

```
do{
    wait(mutex);
    read_count++;
    if(read_count==1)
            wait(rw_mutex);
    signal(mutex)
    ......................................
    /*reading is performed */
    ......................................
    wait(mutex);
    read_count--;
    if(read_count==0)
            signal(mutex);
}while(true);
```

WRITER

```
do{
    wait(rw_mutex);
    ..............................
    /*writting is performed */
    ..........................
    signal(rw_mutex);

}while(true)
```

mutex		readcount		rw_mutex	
	1		0		-1

READER

```
do{
    wait(mutex);
    read_count++;
    if(read_count==1)
        wait(rw_mutex);
    signal(mutex)
    ..................................
    /*reading is performed */
    ..................................
    wait(mutex);
    read_count--;
    signal(mutex)
    if(read_count==0)        ;
        signal(rw_mutex)
}while(true);
```

Reader 1 -> CS

Reader 2 -> CS

Writer 1 -> CS

WRITER

```
do{
    wait(rw_mutex);
    ..............................
    /*writting is performed */
    ..........................
    signal(rw_mutex);

}while(true)
```

Blocked Queue

W1				