

Process Synchronization

- Central to the design of modern Operating Systems is managing multiple processes
 - Multiprogramming
 - Multiprocessing
 - Distributed Processing
- Big Issue is Concurrency
 - Managing the interaction of all of these processes

Interleaving and Overlapping Processes

- Interleaving of processes on a uniprocessor system

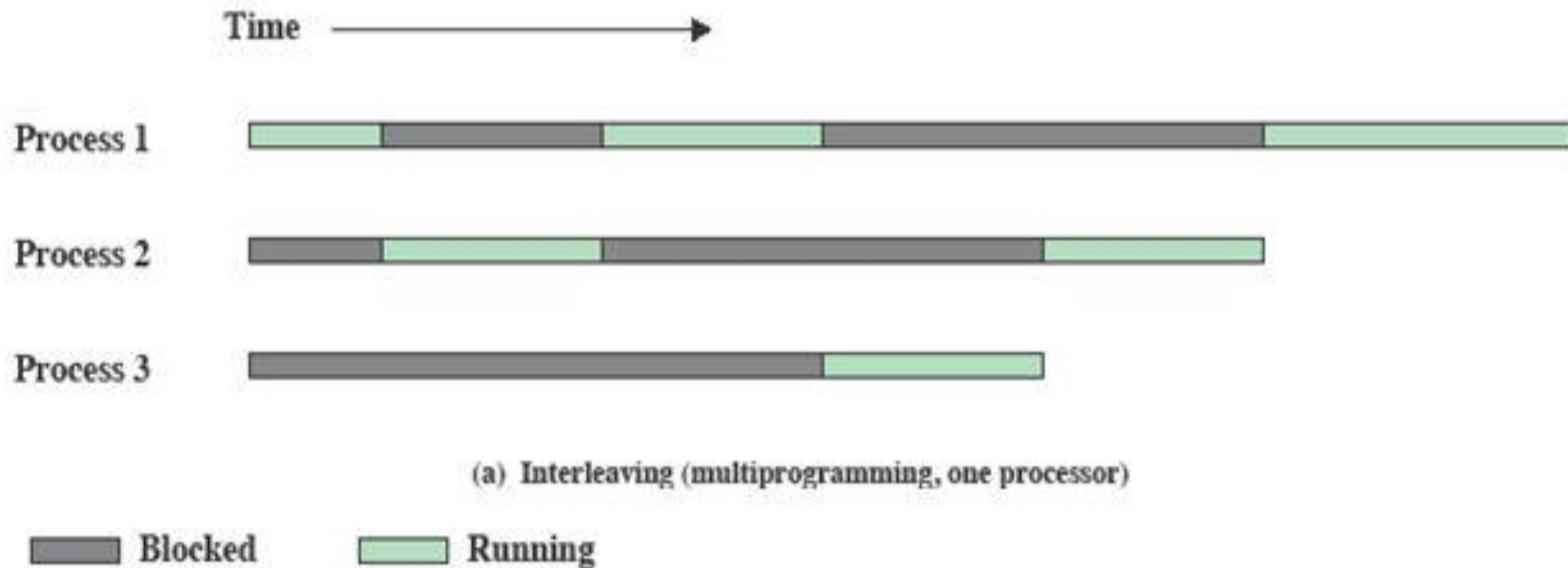


Figure 2.12 Multiprogramming and Multiprocessing

Interleaving and Overlapping Processes

- Not only interleaved but overlapped on multi-processors

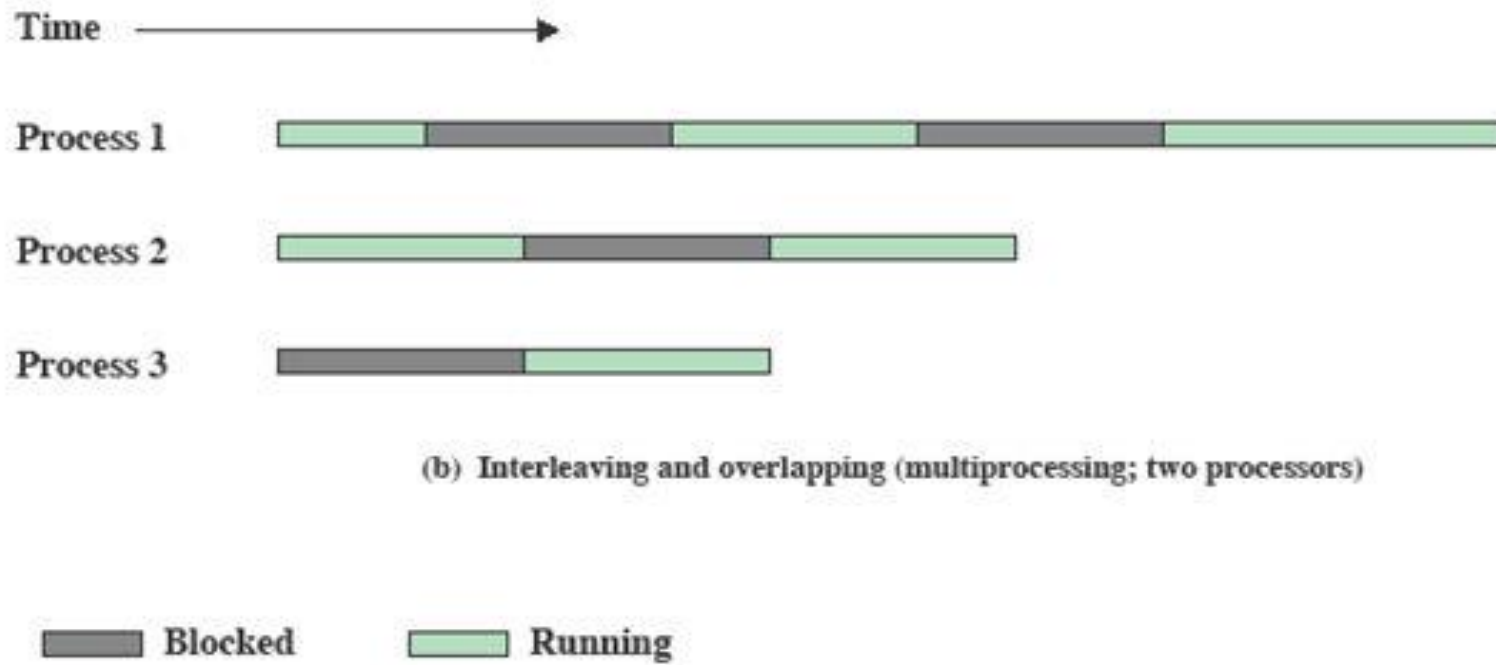


Figure 2.12 Multiprogramming and Multiprocessing

Concurrency

- Difficulties in concurrency

Eg: If two processes both make use of the same global variable and both perform reads and writes on that variable, then the order in which the reads and writes are executed is critical.

A Simple Example

(Consider the echo procedure)

```
void echo()  
{  
    chin = getchar(); // get a character from the keyboard  
                      // and store it to variable chin  
    chout = chin;     // transfer it to variable chout  
    putchar(chout);   // print the character to the display  
}
```

A Simple Example

(Consider the echo procedure)

- Input is obtained from a keyboard one keystroke at a time.
- Each input character is stored in variable chin. It is then transferred to variable chout and sent to the display.
- Instead of each application having it's own procedure **this procedure may be shared and only a copy is loaded to the memory global to all applications** thus saving space.
- Any program can call this procedure repeatedly to accept user input and display it on the user's screen.

Example : echo procedure

- **Consider a single processor multiprogramming system supporting a single user.**
- The user can be working on a number of applications simultaneously.
- Assume each of the application needs to use the procedure echo, which is shared by all the applications.
- However this sharing can lead to problems.

Example : echo procedure

- The sharing of main memory among processes is useful but could lead to problems.
- Consider the following sequence:

P1 invokes echo and is interrupted after executing

chin = getchar(); // assume x is entered

P2 is activated and invokes echo, which runs to completion

// assume y is entered and displayed

P1 is resumed and executes

chout = chin;

putchar(chout);

// What is the result?

- The result is that the character input to P1 is lost before being displayed
- And the character input to P2 is displayed by both P1 and P2.

Example : echo procedure

Problem

- The shared global variable chin and chout is used by multiple processes.
- If one process updates the global variable and then is interrupted, another process may alter this variable before the first process can use this value.

Race Condition

- **This is called as a Race Condition**

- A race condition occurs when

Two or more processes access a shared data and the outcome is dependent upon which process precisely runs when.

Example : echo procedure

- Suppose that we permit only one process at a time to be in that procedure.
- So, the following sequence may result:
 1. P1 invokes the *echo* procedure and is interrupted immediately after the conclusion of the input function.
 - x is stored in variable *chin*.
 2. P2 is activated and invokes the *echo* procedure.

However because P1 is still in the *echo* procedure, P2 is currently blocked from entering the procedure. Therefore P2 is suspended awaiting for the availability of the *echo* procedure.

Example : echo procedure

3. P1 resumes and completes execution of *echo*. The proper character x is displayed.
 4. When P1 exits *echo*, this removes the block on P2. When P2 later resumed, the echo is successfully invoked.
- Necessary to protect share global variables.
 - How?
 - Control the code that accesses the variable.

Example:

On a Multiprocessor system

- Previous Example : Assumption-Single processor, multiprogramming operating system.
- Same problem could occur on a multiprocessor system.
- Processes P1 and P2 are both executing each on a separate processor. Both processes invoke the echo procedure.

Process P1

```
.  
chin = getchar();  
.   
chout = chin;  
putchar(chout);  
.   
.
```

Process P2

```
.  
.   
chin = getchar();  
chout = chin;  
.   
putchar(chout);  
.   
.
```

On a Multiprocessor system

- The result is that the character input to P1 is lost before being displayed, and the character input to P2 is displayed by both P1 and P2.
- Suppose that we permit only one process at a time to be in that procedure.
- So, the following sequence may result:
 1. P1 & P2 run on separate processors
 2. P1 enters echo procedure first,
 3. P2 tries to enter but is blocked – P2 is suspended
 4. P1 completes execution
 5. P2 resumes and executes echo procedure

Analogy (Too much milk problem)

For example, consider the "too much milk" problem.

- Suppose you're sharing your room with a roomie.
- You realize there's no milk in the fridge, and being a good roommate, you decide to go and buy some.
- Your roommate isn't in the room when you go.
- A little while later, while you're still outside, the roommate, also being a nice guy, decides to go buy milk as well.
- **Now both of you buy milk and there's too much milk.**

How do we fix this? Well, you could leave a note on the fridge before going, in which case, both threads (you and your roommate) would have this code:

Analogy (Too much milk problem)

```
if (noMilk) {  
    if(noNote) {  
        leaveNote();  
        buyMilk();  
        removeNote();  
    }  
}
```


Analogy (Too much milk problem)

But this has a problem

- Suppose the first process is executing, and it checks there's no milk and before it checks the other if condition, the scheduler decides to switch to task B.
- Task B checks there's no milk, and that there's no note, and before it can execute the code inside the inner if, the scheduler switches again.
- Now task A checks that there's no note, and executes the statements and buys milk.
- Now the scheduler switches to B, which continues executing and now both of them buy milk.

It's a fail.

Critical Section & Mutual Section

- The portion in any program , which accesses a shared resource(such as a shared variable in memory) is called as the **CRITICAL SECTION** or **CRITICAL REGION**.

Solution to race condition.

When a process is in the critical section disallow any process to enter its critical section

i.e. No two processes should be allowed to be inside their critical region at the same time.

This is called as **MUTUAL EXCLUSION**

Critical Section & Mutual Section

To summarize

- Each communicating process has a sequence of instructions that modify shared data. This sequence of instructions that modify the shared data is called a
- CRITICAL SECTION.
- If 2 processes are executing in their respective critical section at the same time, then they interfere with each other and the result of execution depends on the order in which they are executing.

This is called a RACE CONDITION and should be avoided.

- The way to avoid race condition is to allow only one process at a time to be executing in its critical section.

This is called as MUTUAL EXCLUSION

Primitives of mutual exclusion

- So each process must first request permission to enter its critical section.
- The section of code implementing this request is called the Entry Section (ES).
- The critical section (CS) might be followed by a Leave/Exit Section (LS).
- General structure of a process:

entry section

critical section

exit section

remainder section

Requirements for a valid solution to the critical section problem

The solutions for the CS problem should satisfy the following characteristics :

1. **Mutual Exclusion :**

At any time, at most one process can be in its critical section (CS)

2. **Progress:**

If there are some processes in the queue waiting to get into their CR and currently no process is inside the CR, then only the waiting processes must be granted permission to execute in the CR

i.e. The CS will not be reserved for a process that is currently in a non-critical section.

3. **Bounded wait :**

A process in its CS remains there only for a finite time because generally the execution does not take much time, so the waiting processes in the queue need not wait for a long time.

Process using a CS

- Eg: Process using a CS

do {

Entry section

CRITICAL SECTION

Exit section

Remainder section

} while()

In this eg. The CS is situated in a loop.

In each iteration of the loop, the process uses the CS and also performs other computations called Remainder section

Approaches to achieving Mutual Exclusion

Two Process Solution

Attempt 1

- Processes enter their CSs according to the value of *turn(shared variable)* and in strict alternation

```
do {  
    while (turn = 2) do {nothing};  
  
    {CRITICAL SECTION}  
  
    turn:=2  
  
    Remainder Section  
} while ()
```

Process 1

```
do {  
    while (turn = 1) do {nothing};  
  
    {CRITICAL SECTION}  
  
    turn:=1  
  
    Remainder Section  
} while ()
```

Process 2

Two Process Solution: Attempt 1

- turn is a shared variable.
- It is initialized to 1 before processes p1 and p2 are created.
- Each of these processes contains a CS for some shared data d.
- The shared variable turn is used to indicate which process can enter the CS next.
- Let $\text{turn}=1$, P1 enters CS.
- After completing CS , it sets turn to 2 so that P2 can enter CS

Two Process Solution: Attempt 1

Adv:

- Achieves Mutual Exclusion

Drawback

- Suffers from busy waiting(technique in which a process repeatedly checks a condition) i.e entire time slice allocated to the process is wasted in waiting.
- Let P1 be in CS and process P2 be in remainder section. If P1 exits from CS, and finishes its remainder section and wishes to enter the CS again, it would encounter a busy wait until P2 uses the CS.

Here the PROGRESS condition is violated since P1 is currently **the only process** interested in using the CS, however it is not able to enter the CS.

Two Process Solution: Attempt 2

Attempt 2

- Drawback of the first algorithm is that it does not know the state of the executing process(whether in CS/RS)
- To store the state of the process 2 more variables are introduced, state_flag_P1 and state_flag_P2.
- If a process enters its CR, it first sets its state flag to 0, and after exiting it sets it to 1.
- Use of two shared variables eliminates the problem of progress in the first algorithm.
- Does not require strict alternation of entry into CR

Two Process Solution: Attempt 2

- Attempt 2

Process 1

```
{  
state_flag_P1=1;  
do  
{  
while(state_flag_P2 ==0);  
state_flag_P1=0;
```

CR

```
state_flag_P1=1;
```

```
Remainder section  
} while(true)
```

Process 2

```
{  
state_flag_P2=1;  
do  
{  
while(state_flag_P1 ==0);  
state_flag_P2=0;
```

CR

```
state_flag_P2=1;
```

```
Remainder section  
} while(true)
```

Two Process Solution: Attempt 2

- Violates mutual exclusion property.
- Assume currently **state_flag_P2=1**;
- P1 wants to enter CS.
- P1 skips the while loop.
- Before it could make **state_flag_P1=0**, it is interrupted.
- P2 is scheduled, it finds state_flag_P1=1 and enters CS.
- Hence both processes are in CS
- Hence violates mutual exclusion property.

Two Process Solution: Peterson's algorithm

Peterson's algorithm

- Uses a boolean array `process_flag[]` which contains one flag for each process. These flags are equivalent to status variables **state_flag_P1** **state_flag_P2** of the earlier algorithm.
- `process_flag[0]` is for P1 and `process_flag[1]` is for P2.
- If the process P1 wants to enter the CR it sets `process_flag[0]` to true. If it exits CR then it sets `process_flag[0]` to false.
- Similarly If the process P2 wants to enter the CR it sets `process_flag[1]` to true. If it exits CR then it sets `process_flag[1]` to false.

Two Process Solution: Peterson's algorithm

- Peterson's Algorithm

Process P1

```
{
do {
process_flag[0]=true;
process_turn=1;
while(process_flag[1] &&
           process_turn==1);
```

CR

```
process _flag[0]=false;
```

```
Remainder section
} while(true)
```

Process P2

```
{
do {
process_flag[1]=true;
process_turn=0;
while(process_flag[0] &&
           process_turn==0);
```

CR

```
process _flag[1]=false;
```

```
Remainder section
} while(true)
```

Two Process Solution: Peterson's algorithm

- In addition there is one more shared variable `process_turn`, which takes the value of 0 for P1 and 1 for P2.
- The variable `process_turn` maintains mutual exclusion and `process_flag []` maintains the state of the process.
- Assume both the processes want to enter the CR, so both are making their flags true, but to maintain mutual exclusion, the processes before entering the CR allow other processes to run.
- The process that satisfies both the criteria, that its `process_flag` is true and it is its process turn will enter the CR.
- After exiting the CR, the process makes its flag false, so that the other can enter CR, if it wants

Two Process Solution: Peterson's algorithm

Eg:1

- Assume initially both `process_flag[0]` and `process_flag[1]` are false.
- Assume P1 is about to enter the CR hence it makes `process_flag[0]=true;`
`process_turn=1;`
- It executes while statement , but since `process_flag[1]=false` , P1 will enter the CR, execute CR and after exiting it will make `process_flag[0] =false`.
- Now if P2 wants to enter CR, it makes
`process_flag[1]=true; process_turn=0.`
- It executes while statement , but since `process_flag[0]=false` , P2 will enter the CR,

Two Process Solution: Peterson's algorithm

- P2 executes CR and after exiting it will make `process_flag[1] = false`.

Eg : 2

- Assume initially both `process_flag[0]` and `process_flag[1]` are false.
- P1 wants to enter CR, `process_flag[0] = true`; `process_turn = 1`;
- It executes while statement, but since `process_flag[1] = false`, P1 will enter the CR.
- Now assume control switches to P2 and P2 wants to enter CR

Two Process Solution: Peterson's algorithm

- P2 makes `process_flag[1]=true`; `process_turn=0`.
- Executes the while loop and since `process_flag[0]` is true and `turn =0` it waits in the while loop and not allowed to enter CR.
- Now if CPU switches back to P1, P1 continues with CR and after exiting makes `process_flag[0]` to false.
- And now when CPU switches back to P2, P2 can enter CR.
- Hence achieves mutual exclusion.

Two Process Solution: Peterson's algorithm

Ex 3:

- Assume P1 starts and execute `process_flag[0]=true` and `process_turn=1`;
- At this time P2 interrupts and gets execution and executes `process_flag[1]=true`;
- And continues with `process_turn=0`;
- At this time if P1 is able to get back the execution(CPU), then it can continue because P2 has given P1 another chance to execute by making `process_turn=0`.
- Hence P1 will not busy wait in the while loop and enter CR.

Hardware approaches

1. TSL instruction

- Many computers have a special instruction called “Test and Set Lock” (TSL) instr.
- The instruction has the format -----→

TSL ACC, IND

ACC-accumulator reg

IND- name of a memory location which hold a character(F/N)

- The following actions are taken after the instruction is executed---→
 1. Copy the contents of IND to ACC.
 2. Set the contents of IND to “N”.

Hardware approaches

- This instruction is an indivisible instruction, which means that it cannot be interrupted during its execution consisting of these 2 steps.
- Hence process switch cannot take place during the execution of this TSL instruction.
- It will either be fully executed or will not be executed at all.
- How can we use this TSL instruction to implement mutual exclusion?

1. IND can take on values N ---- Not free (CR)

F ---- Free (CR)

If $IND = N$ no process can enter its CR because some process is in CR.

Hardware approaches

2. There are 2 routines
1. ENTER-CRITICALREGION
 2. EXIT-CRITICALREGION

The CR is encapsulated between these 2 routines.

ENTER-CRITICALREGION

EN.0	TSL ACC,IND	IND → ACC, "N" → IND
	CMP ACC,F	Check if CR is free
	BU EN.0	Branch to EN.0 if unequal
	RTN	Return to the caller and enter CR

Hardware approaches

EXIT-CRITICALREGION

MOV IND,"F"

RTN

Begin

Initial Section

Call ENTER-CRITICALREGION

CR

Call EXIT-CRITICALREGION

Remainder Section

End

Hardware approaches

- Let IND="F"
- PA is scheduled. PA executes ENTER-CR routine.
- ACC becomes "F" and IND becomes "N".
- Contents of ACC are compared with "F".
- Since it is equal process PA enters its CR.
- Assume process PA loses the control of CPU due to context switch to PB when in CR.
- PB executes ENTER-CR routine. PB executes EN.0.
- IND which is "N" is copied to ACC and IND becomes "N". ACC=N.
- The comparison fails and loops back to EN.0 and therefore does not execute its CR.

- Assume that PA is rescheduled and it completes its execution of CR and then executes the Exit-CR routine where IND becomes “F”.
- Hence now since IND becomes “F” and eventually ACC becomes “F”, the next process say PB if scheduled again can execute its CR.

Hardware approaches

2. Interrupt Disabling/Enabling

- When a process enters a CR , it should complete the CR without interruption.
- A process switch after a certain time slice happens due to an interrupt generated by the timer hardware.
- One solution is to disable all interrupts before any process enters the CR.
- If the interrupt is disabled, the time slice of the process which has entered its CR will never get over until it has come out of its CR completely.
- Hence no other process will be able to enter the CR simultaneously.

Hardware approaches

Other Instructions

DI

CR

EI

Remaining Sections

Drawback:

1. Dangerous approach since it gives user processes the power to turn off the interrupts.

A process in the CR executing an infinite loop.

The interrupt will never be enabled and no other process can ever proceed

2. The approach fails for a multiprocessor system i.e if there are 2 or more CPUs disabling interrupt affects only the CPU that executed the disable instruction

Semaphores

- Used for synchronization
- Used to protect any resource such as shared global memory that needs to be accessed and updated by many processes simultaneously.
- Semaphore acts as a guard or lock on that resource.
- Whenever a process needs to access the resource, it first needs to take permission from the semaphore.
- If the resource is free , ie. no process is accessing or updating it, the process will be allowed , otherwise permission is denied.
- In case of denial the requesting process needs to wait, until semaphore permits it.

Semaphores

- Semaphore can be a “Counting Semaphore” where it can take any integer value or a “Binary semaphore” where it can take on values 0 or 1
- The semaphore is accessed by only 2 **indivisible** operation known as wait and signal operations, denoted as P and V respectively.
- P- proberen(Dutch word)/wait /down
V- verogen(Dutch word)/signal /up
- P and V form the mutual exclusion primitives for any process.
- Hence if a process has a CS, it has to be encapsulated between these 2 operations

Semaphores

- The general structure of such a process becomes --→

Initial routine

P(S)

CS

V(S)

Remainder section

- The P and V primitives ensure that only one process is in its CS.

P(S)

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

V(S)

```
{  
S++;  
}
```

- P and V routine must be executed indivisibly.

Drawback

- Requires busy waiting. If a process(PA) is in its CS and is interrupted by another process(PB).When PB tries to execute P(S) it loops continuously in the entry code(busy waiting).

Busy waiting is undesirable since it wastes the CPU time.

To overcome busy waiting, P and V are modified, when a process executes P(S) and finds the semaphore value 0, rather than busy waiting, the process can go to the blocked state.

i.e the process goes from running state to blocked state and is put in a queue of waiting processes called the semaphore queue

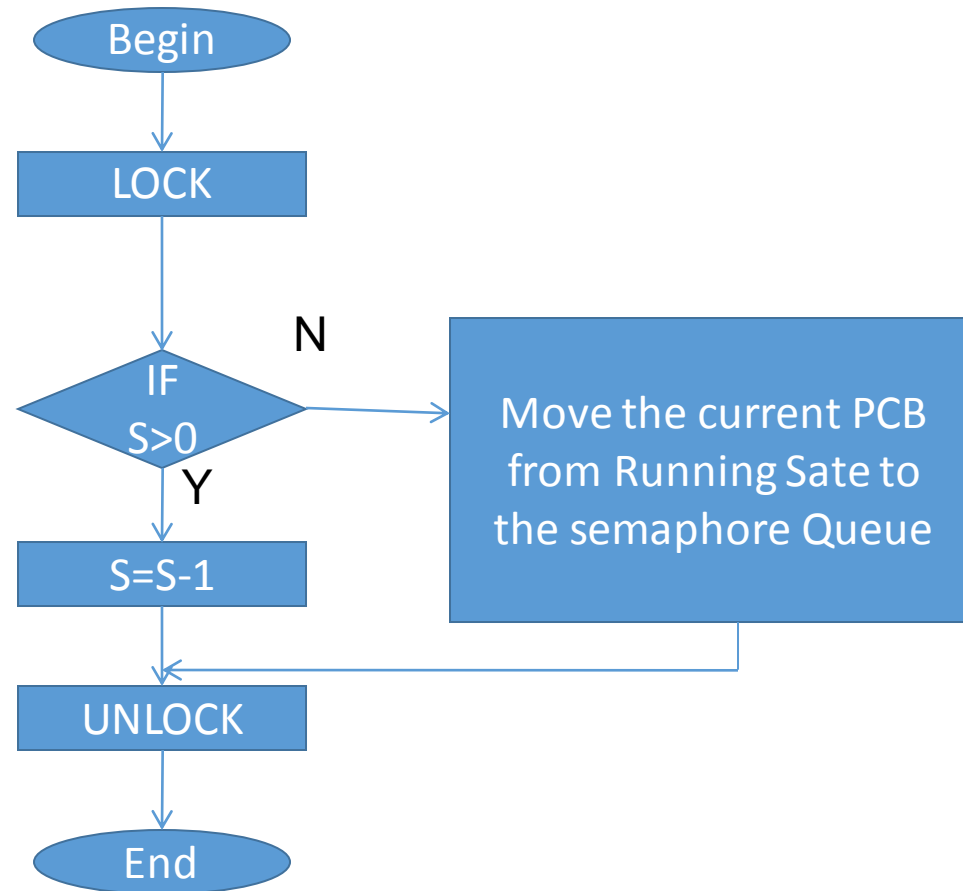
- **Semaphore queue**

Queue of all processes wanting to enter the CS.

- The semaphore queue is a chain of all PCBs of processes waiting for the CS to get free.
- Only when a process which is in its CS comes out of it, should the OS allow a new process to be released from the semaphore queue.

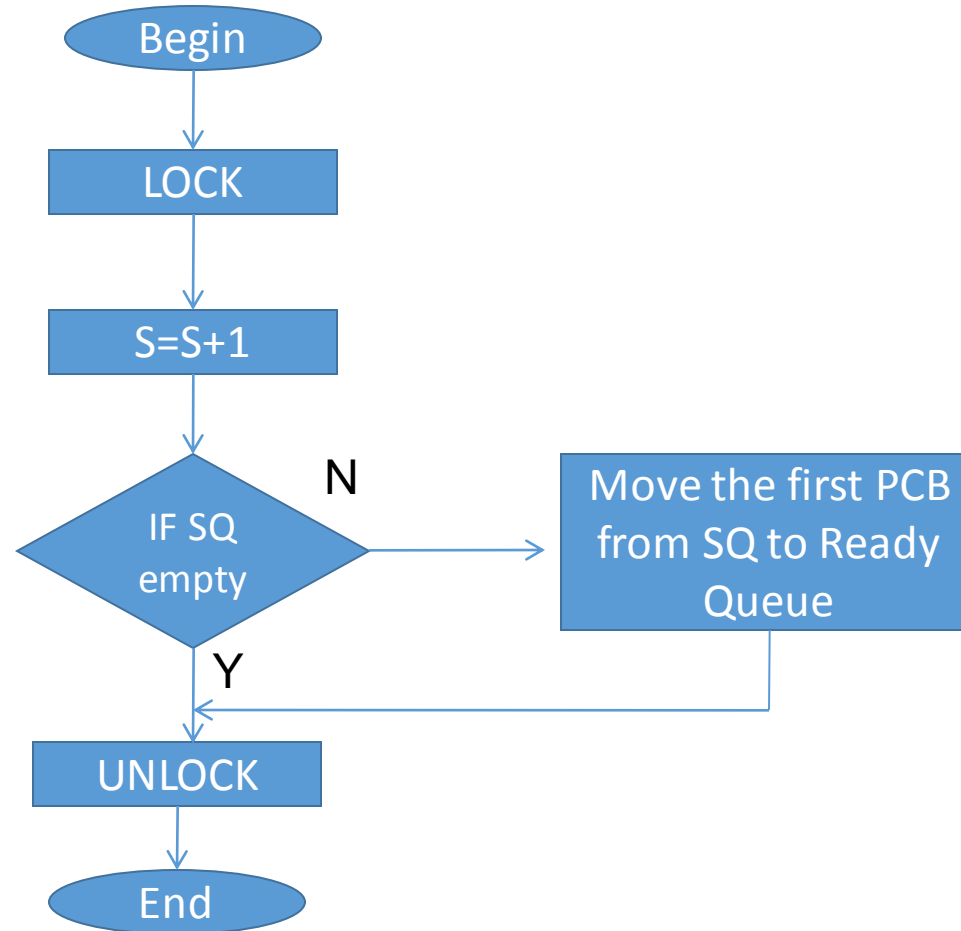
Semaphores(Binary)

P(S)



Semaphores(Binary)

V(S)



Semaphores(Binary)

- Algorithms

P.0 **Disable interrupt**

P.1 If $S > 0$

P.2 then $S = S - 1$

P.3 else wait on S

P.4 Endif

P.5 **Enable interrupts**

End

U.0 **Disable interrupt**

U.1 $S = S + 1$

U.2 If SQ NOT empty

U.3 then release a process

U.4 Endif

U.5 **Enable interrupts**

End

Semaphores(Binary)

Principles of operation

- Assume $S=1$ and 4 processes PA PB PC PD in the Ready Queue. Each of these processes have a CS encapsulated between P(S) and V(S).
- Assume PA gets scheduled(control of CPU)
 1. PA executes the initial routine , if it has one.
 2. It executes P(S) routine.
 3. Disables interrupt. Checks if $S>0$. As $S=1$, it decrements S and S becomes 0. And enables interrupt.
 4. PA enters the CS.
 5. Assume time slice for PA gets over while it is in the CS and PA is moved from 'running' state to 'ready' state.

Semaphores(Binary)

6. Next PB is scheduled.
7. PB executes P.0.
8. It checks for $S > 0$. But $S = 0$ and therefore check fails. It skips P.2 and PCB of PB is added to the semaphore queue. It is no more running.
9. Next if PC is scheduled, it will undergo the same steps as PB and its PCB will also be added to the SQ because still $S = 0$.
10. When will S become 1?
When PA is rescheduled it will complete its CS and then call the V(S) routine.
11. Assume PA is rescheduled. It disables interrupt at V.0

Semaphores(Binary)

12. Increments S by 1, checks SQ and finds it is not empty.

13. It releases PB ie. It moves it from SQ to RQ . Hence now PB can enter CS after it executes $P(S)$.

Semaphore(counting)

Wait(s.count)

Begin

 s.count:=s.count-1;

 if s.count <0 then

 begin

 place the process in s.queue;

 block this process

 end;

end;

Semaphore(counting)

Signal(s.count):

 s.count:=s.count+1

 if s.count<= 0 then

 Begin

 remove a process from s.queue;

 place this process on ready list

 end;

Note:

- s.count>= 0, s.count is number of processes that can execute wait(s) without blocking
- s.count<=0, the magnitude of s.count is number of processes blocked waiting in s.queue

Time	Current value of Semaphore	Name of the process	Modified value of the semaphore	Current Status
1	1	P1 needs to access	0	P1 enters CS
2	0	P1 interrupted P2 is scheduled	-1	Process P2 is waiting in SQ <div> <div>P2</div> <div></div> <div></div> </div>
3	-1	P3 is scheduled	-2	P2 and P3 in SQ <div> <div>P2</div> <div>P3</div> </div>
4	-2	P1 scheduled. P1 exits CS	-1	P2 is moved from SQ to RQ <div> <div>P3</div> </div>
5	-1	P2 is scheduled Enters CS, exits CS	0	P3 is moved from SQ to RQ SQ is empty
6	0	P3 is scheduled	1	No process in CS

Classic synchronization problem(Producer consumer problem)

Producer consumer problem(Bounded buffer problem)

Eg-1 Compilers can be considered producer process
and assemblers as consumer process.

A compiler produces the object code and an
assembler consumes the object code.

Eg-2 Process that gives a command for printing a file is a producer process, and the process for printing the file on the printer is a consumer process.

- There is a buffer in an application maintained by 2 processes.
- One process is called a producer that produces some data and fills the buffer.
- Another process is called a consumer that needs data produced in the buffer and consumes it.

Classic synchronization problem(Producer consumer problem)

Producer Consumer problem solution using SEMAPHORE

To solve the producer consumer problem using semaphore, the following requirement should be met :

1. The producer process should not produce an item when the buffer is full.
2. The consumer process should not consume an item when the buffer is empty.
3. The producer and consumer process should not try to access and update the buffer at the same time.
4. When a producer process is ready to produce an item and the buffer is full, the item should not be lost i.e the producer must be blocked and wait for the consumer to consume an item.

Classic synchronization problem(Producer consumer problem)

5. When a consumer process is ready to consume an item and the buffer is empty, consumer must be blocked and wait for the producer to produce item.
6. When a consumer process consumes an item i.e a slot in the buffer is created, the blocked producer process must be signaled about it.
7. When a producer process produces an item in the empty buffer, the blocked consumer process must be signaled about it.

Classic synchronization problem(Producer consumer problem)

Producer Consumer problem solution using SEMAPHORE

- In the Producer-Consumer problem, semaphores are used for two purpose:
 - mutual exclusion and
 - synchronization.
- In the following example there are three semaphores:
 1. *full*, used for counting the number of slots that are full;
 2. *empty*, used for counting the number of slots that are empty; and
 3. *Buffer_access*, used to enforce mutual exclusion.
- Let `BufferSize = 3;`
`semaphore Buffer_access = 1; // Controls access to critical section`
`semaphore empty = BufferSize;`
`// counts number of empty buffer slots, semaphore full = 0; // counts number of full buffer slots`

Classic synchronization problem(Producer consumer problem)

Producer()

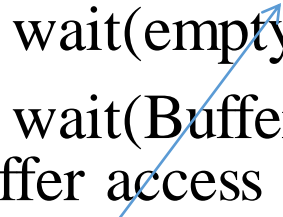
```
{  while (TRUE) {  
    Produce an item;  
    wait(empty);  
    wait(Buffer_access); // enter critical section  
    //buffer access  
    signal(Buffer_access); // leave critical section  
    signal(full); // increment the full semaphore } }
```

The producer must wait
for an empty space in
the buffer

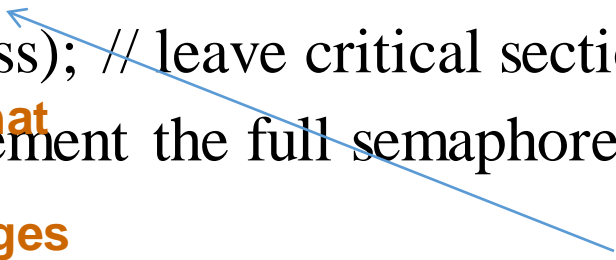


Add item to the buffer;

We must make sure that
the producer and the
consumer make changes
to the shared buffer in a
mutually exclusive
manner



Wake up a blocked
consumer process




Classic synchronization problem(Producer consumer problem)

```
Consumer()
{ while (TRUE)
{
    wait(full); // decrement the full semaphore
    wait(Buffer_access); // enter critical section
    Consume item from buffer ;
    signal(Buffer_access); // leave critical section
    signal(empty); // increment the empty semaphore
}
}
```

**The consumer must wait
for a space in the buffer**



**We must make sure that the producer
and the consumer make changes to
the shared buffer in a mutually
exclusive manner**



**Wake up a blocked
producer process**



Classic synchronization problem(Producer consumer problem)

Eg: let empty=3 full =0 Buffer_access =1

Assume producer is scheduled(allocated the CPU).

- It produces an item
- Executes wait(empty), empty becomes 2
- Executes wait (Buffer_access), Buffer_access becomes 0.
- Adds item to the buffer
- Executes signal(Buffer_access) , Buffer_access becomes 1.
- Executes signal(full), full becomes 1

Classic synchronization problem(Producer consumer problem)

Assuming time slice of producer process is not yet over....

- Producer produces the next item.
- Empty=1
- Buffer_access=0
- Producer adds item to the buffer.
- Buffer_access =1
- full=2

Classic synchronization problem(Producer consumer problem)

Continuing.....

- Producer produces the next item.
- Empty=0
- Buffer_access=0
- Producer adds item to the buffer.
- Buffer_access =1
- full=3

Classic synchronization problem(Producer consumer problem)

Continuing.....

- Producer produces the next item.
- Empty=-1

PRODUCER PROCESS GETS BLOCKED

Classic synchronization problem(Producer consumer problem)

- Assume the consumer is scheduled to run
- Consumer executes wait(full) , full becomes 2
- Executes wait(Buffer_access), Buffer_access becomes 0.
- Consumes item from the buffer.
- Executes signal(Buffer_access), Buffer_access=1.
- Executes signal(empty), empty becomes 0 and it wakes up the producer process(removes it from the blocked state).

Classic synchronization problem(Producer consumer problem)

- After this if the producer process is scheduled...
- It will start from wait(Buffer_access), Buffer_access=0.
- It add an item to the buffer.
- Buffer_access=1.
- full becomes 3.

Classic synchronization problem(Producer consumer problem)

Eg: 2

Assume empty=3 full =0 Buffer_access=1

And consumer process is scheduled

- Consumer executes wait(full), full becomes -1.

CONSUMER PROCESS GETS BLOCKED

Classic synchronization problem(Reader-Writer problem)

- Assume a data item being a shared by a number of processes.
- Some processes read the data item -reader processes
- Some processes write/update the data item -writer processes
- Eg- In an airline reservation system, there is a shared data where the status of seats is maintained.
- If a person needs to enquire about the reservation status, then the reader process will read the shared data and get the information.
- On the other hand if a person wishes to reserve a seat, then the writer process will update the shared data.

Classic synchronization problem(Reader-Writer problem)

- Can multiple readers access the shared data simultaneously

YES

- Can multiple writers access the shared data simultaneously

NO

i.e if one writer is writing, other writers must wait.

Also when a writer is writing , a reader is not allowed to access the data item.

- This synchronization problem cannot be solved by simply providing mutual exclusion on the shared data area, because it will lead to a situation where a reader will also wait while another reader is reading.
- i.e the people enquiring reservation status should be given simultaneous access, otherwise there will be unnecessary delays.

Classic synchronization problem(Reader-Writer problem)

Key features of the readers-writers problem are:

- Many readers can simultaneously read.
- Only one writer can write at a time. Readers and writers must wait if a writer is writing. When the writer exits either all waiting readers should be activated or one waiting writer should be activated.
- A writer must wait if a reader is reading. It must be activated when the last reader exits.

Classic synchronization problem(Reader-Writer problem)

Reader writer problem using semaphores

- **Shared data:**

`wrt`

`int readcount`

`semaphore mutex,`

- **Initially:**


`wrt = 1, readcount = 0`

`mutex = 1,`

Classic synchronization problem(Reader-Writer problem)

```
do {  
    P(wrt);  
    ...  
    writing is performed  
    ...  
    V(wrt);  
  
} while (TRUE);
```

**A writer will wait if either
another writer is
currently writing or one
or more readers are
currently reading**



Classic synchronization problem(Reader-Writer problem)

```
do{
  P(mutex) ;
  readcount++;
  if (readcount == 1)
    P(wrt) ;
  V(mutex) ;
  ...
  P(mutex) ;
  readcount--;
  if (readcount == 0)
    V(wrt) ;
  V(mutex) ;
} while(TRUE) ;
```

reading is performed

We must make sure that readers update the shared variable readcount in a mutually exclusive manner

A reader will wait only if a writer is currently writing. Note that if readcount == 1, no reader is currently reading and thus that is the only time that a reader has to make sure that no writer is currently writing (i.e., if readcount > 1, there is at least one reader reading and thus the new reader does not have to wait)

The Dining-Philosophers Problem

- The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices.
- Consider five philosophers who spend their lives thinking and eating.
- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks`



The Dining-Philosophers Problem

- When a philosopher thinks, she does not interact with her colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.

The Dining-Philosophers Problem

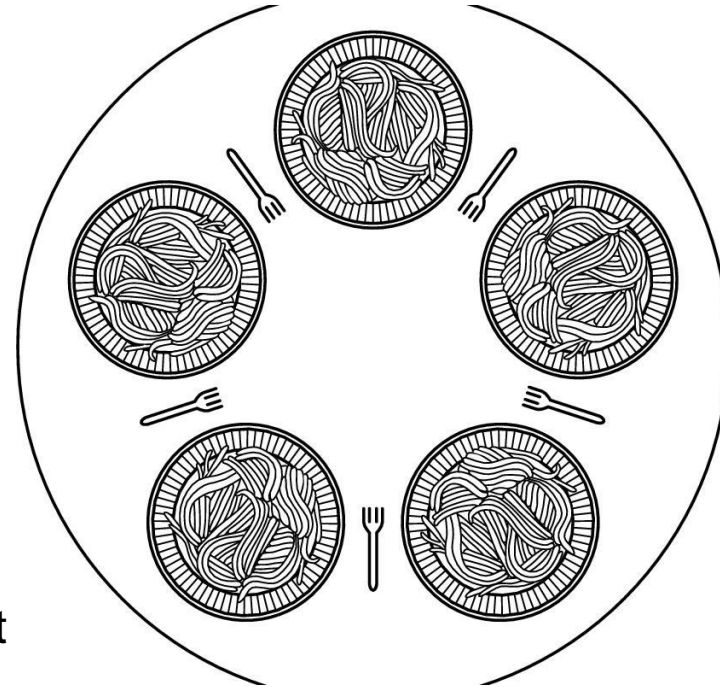
- When she is finished eating, she puts down both of her chopsticks and starts thinking again.
- The dining-philosophers problem is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

The Dining-Philosophers Problem

- Solution 1 `Shared: semaphore fork[5];`
`Init: fork[i] = 1 for all i=0 .. 4`

Philosopher i

```
do {  
    P(fork[i]);  
    P(fork[i+1]);  
  
    /* eat */  
  
    V(fork[i]);  
    V(fork[i+1]);  
  
    /* think */  
} while(true);
```



Oops! Subject
to deadlock if
they all pick up
their "right" fork
simultaneously!

The Dining-Philosophers Problem

- Another solution given by Tanenbaum :
- Uses binary semaphores.
- He defines the states of philosopher.
- The possible states are thinking, eating and hungry.
- There is an array `state[5]` which stores the state of each philosopher.
- This array will be used by all the philosophers hence it has to be accessed in a mutually exclusive manner.
- Hence a semaphore is defined `sem_state`.
- A semaphore is taken for all the philosophers(`philosopher[5]`), on which they wait to start eating.
- The philosopher can start eating if neither of their neighbour is eating.
- The state of the philosopher is checked and updated.

The Dining-Philosophers Problem

- Solution

```
void Philosopher(int n)
{  do {
    think()
    get_spoons(n);
    eat();
    put_spoons(n);
  }while(true)
}
```

The Dining-Philosophers Problem

```
void get_spoons(int n)
{
    wait(Sem_state);
    state[n]=Hungry;
    test_state(n);
    signal(Sem_state);
    wait(philosopher[n]);
}
```

The Dining-Philosophers Problem

```
void put_spoons(int n)
{
    wait(Sem_state);
    state[n]=Thinking;
    test_state(LEFT);
    test_state(RIGHT);
    signal(Sem_state);
}
```

The Dining-Philosophers Problem

```
Void test_state(int n)
{ if (state[n]=Hungry && state[LEFT]!=Eating && state[RIGHT]!=Eating )
    { state[n]=Eating;
      signal(philosopher[n]);
    }
}
```

Solution

```
void Philosopher(int n)
{
    do {
        think()
        get_spoons(n);
        eat();
        put_spoons(n);
    } while(true)
}

void test_state(int n)
{
    if (state[n]=Hungry && state[LEFT]!=Eating &&
        state[RIGHT]!=Eating )
    {
        state[n]=Eating;
        signal(philosopher[n]);
    }
}

void get_spoons(int n)
{
    wait(Sem_state);
    state[n]=Hungry;
    test_state(n);
    signal(Sem_state);
    wait(philosopher[n]);
}

void put_spoons(int n)
{
    wait(Sem_state);
    state[n]=Thinking;
    test_state(LEFT);
    test_state(RIGHT);
    signal(Sem_state);
}
```

File and record locking

- Mutual exclusion is required in file processing where several users are accessing the same file (databases)
- Example airline booking system, one seat left two agents access at same time before file is updated and seat is double booked.
- Solution is file locking preventing access to the file while it is being updated