OS

Chapter – 2 "Interprocess Communication"

1) What is Interprocess Communication?
2) Race Conditions
3) Critical Regions
4) Mutual Exclusion with Busy Waiting
    a. Disabling Interrupts
    b. Lock Variables
    c. Strict Alternation
    d. Peterson's Solution
    e. Hardware Solution
5) The Producer-Consumer Problem
6) Semaphore
7) Solving Producer-Consumer Problem with Semaphore
8) Classical IPC Problems
    a. Dining Philosopher Problem
    b. Readers and Writers Problem

1) What is Interprocess Communication?

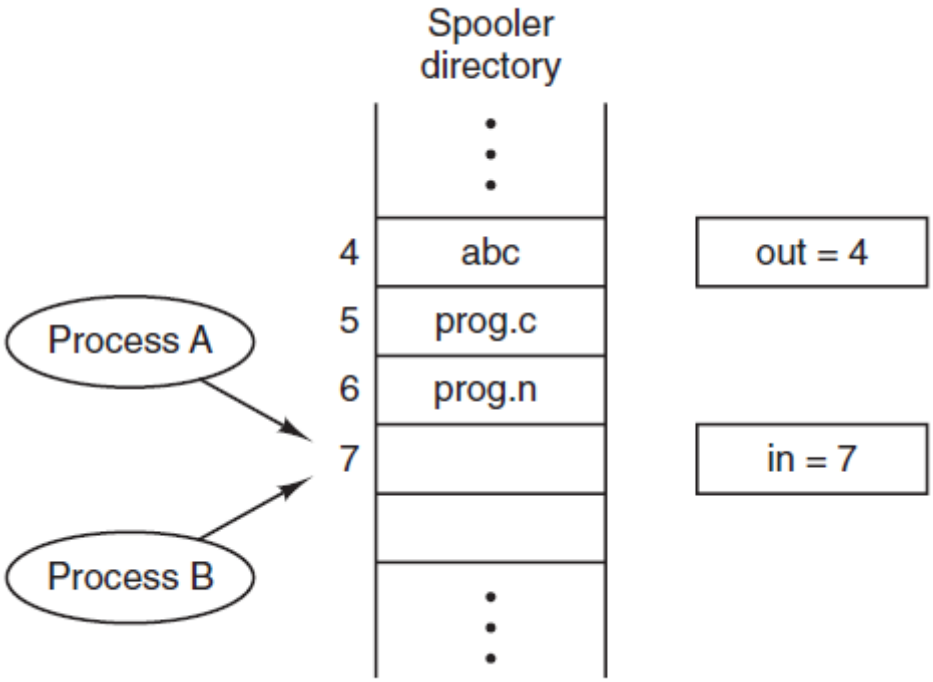
- Processes frequently need to communicate with other processes.
- For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on.
- Thus there is a need for communication between processes.
- 3 Issues:
    1) How one process can pass information to another.
    2) Making sure that two or more processes do not get in each other's way, when engaging in each other's activities.
    3) The third concerns proper sequencing when dependencies are present: if process *A* produces data and process *B* prints them, *B* has to wait until *A* produces some data before starts printing.

2) Race Conditions

- Processes that are working together may share some common storage that each one can read and write.
- To see how interprocess communication works in practice, let us now consider a simple but common example: a print spooler.
- **Spooler directory** : "When a process wants to print a file, it enters the file name in a special spooler directory".
- **Printer daemon** : "The **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory".
- Imagine that our spooler directory has a very large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name.
- Also imagine that there are two shared variables:
  out - Points to the next file to be printed
  in - Points to the next free slot in the directory.

- At a certain instant:

  Slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). This situation is shown in Fig. 2-21.
- More or less simultaneously, processes A and B decide they want to queue a file for printing. This situation is shown in Figure below:



- Process *A* reads *in* and stores the value, 7, in a local variable called *next_free_slot*.
- Just then a clock interrupt occurs and the CPU decides that process *A* has run long enough, so it switches to process *B*. Process *B* also reads *in* and also gets a 7. It, too, stores it in *its* local variable *next_free_slot*.
- At this instant both processes think that the next available slot is 7.
- Process *B* now continues to run. It stores the name of its file in slot 7 and updates *in* to be an 8. Then it goes off and does other things.
- Eventually, process *A* runs again, starting from the place it left off. It looks at *next_free_slot*, finds a 7 there, and writes its file name in

slot 7, **erasing the name that process *B* just put there**. Then it computes *next_free_slot* + 1, which is 8, and sets *in* to 8.

- The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, **but process *B* will never receive any output**.
- User *B* will hang around the printer for years, hoping for output that never comes.
- **Race Conditions** :

  "Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions**".

3) Critical Regions

- How do we avoid race conditions? The key to preventing trouble here (and in many other situations involving shared memory, shared files, and shared everything else) is to find some way to not to allow more than one process from reading and writing the shared data at the same time.
- Put in other words, what we need is **mutual exclusion**, that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.
- The difficulty above occurred because process *B* started using one of the shared variables before process *A* was finished with it.
- **Critical region** : "The part of the program where the shared memory is accessed is called the **critical region** or **critical section**".
- If we could arrange matters such that no two processes were ever there in their critical regions at the same time, we could avoid races.

- Although this requirement avoids race conditions, it is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.
- We need four conditions to hold to have a good solution:

1. **No two processes** may be simultaneously inside their critical regions.
2. **No assumptions** may be made about speeds or the number of CPUs.
3. **No process running outside** its critical region may block any process.
4. No process should have to **wait forever** to enter its critical region.

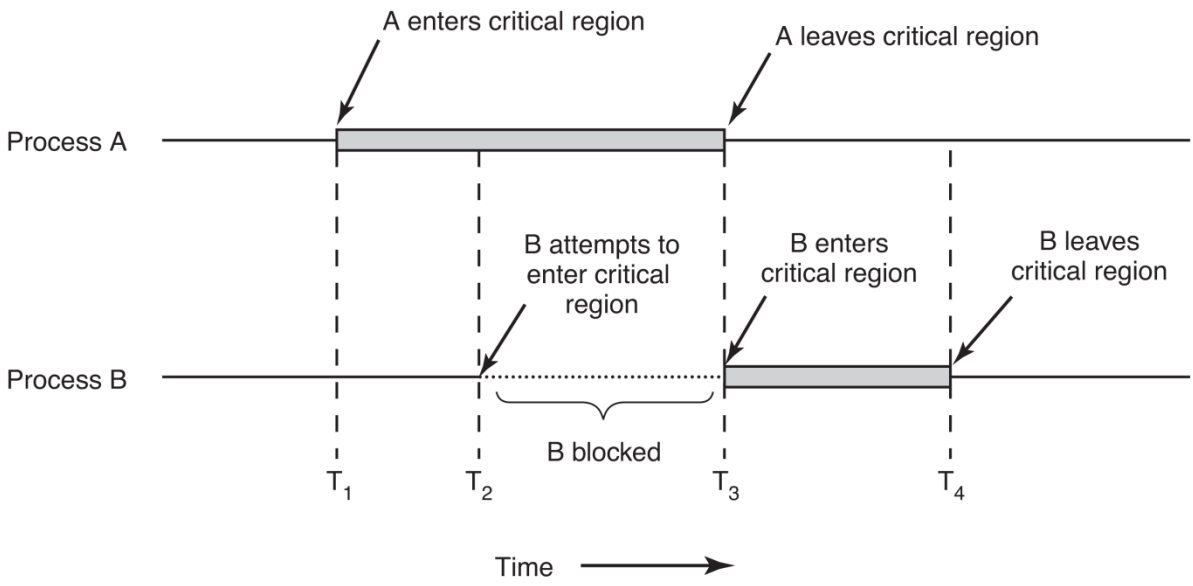


Figure: Mutual exclusion using critical regions

Example:

- Process A enters its critical region at time T1.
- A little later, at time T2 process **B attempts** to enter its critical region but fails because another process is already in its critical region and we allow only one at a time.
- Consequently, **B is temporarily suspended** until time T3.

- At Time T3, when A leaves its critical region, B is allowed to enter critical region.
- Eventually B leaves (at T4) and we are back to the original situation with no processes in their critical regions.

4) Mutual Exclusion with Busy Waiting

a) Disabling Interrupts
b) Lock Variables
c) Strict Alternation
d) Peterson's Solution
e) Hardware Solution (TSL instruction)

a) Disabling Interrupts

- The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it.
- With interrupts disabled, no clock interrupts can occur.
- As we know, the CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process.
- Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will interfere.

Disadvantage:

- This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. What if one of them did it, and never turned them on again? That could be the end of the system.

b) Lock Variables :

- As a second attempt, let us look for a software solution.
- Consider having a single, shared (lock) variable, initially 0.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0 ➔ the process sets it to 1 and enters the critical region.
- If the lock is 1 ➔ the process just waits until it becomes 0 because Other process is already there in the critical section.
- Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Disadvantage:

- Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory.
- Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

c) Strict Alternation

Process 0	Process 1
while (TRUE) **{** **while (turn != 0) ;** **critical region( );** **turn = 1;** **noncritical region( );** **}**	**while (TRUE)** **{** **while (turn != 1) ;** **critical region( );** **turn = 0;** **noncritical region( );** **}**

- Integer variable *turn* ➔ keeps track of whose turn it is to enter the critical region and examine or update the shared memory.
- Initial value of turn is 0.
- Initially, process 0 inspects *turn*, finds it to be 0, and enters its critical region.
- Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing *turn* to see when it becomes 1.
- **Busy waiting** : "Continuously testing a variable until some value appears is called **Busy waiting**".
- (Busy waiting should usually be avoided, since it wastes CPU time).
- When process 0 leaves the critical region, it sets *turn* to 1, to allow process 1 to enter its critical region.
- Suppose that process 1 finishes its critical region quickly, and sets turn to 0.
- Now both processes are in their noncritical regions, with *turn* set to 0.

Disadvantage:

- Consider that process 0 enters its critical region, completes the task in critical region and leaves the critical region. It sets *turn* to 1, to allow process 1 to enter its critical region.
- At this point both processes are executing in their noncritical regions.
- Now as the turn is set to 1 only process 1 can enter in the critical region, not process 0.
- Now, process 0 wants to enter critical region but unfortunately, it is not permitted to enter its critical region, because *turn* is 1 and process 1 is busy with its noncritical region.
- Thus, Process 0 hangs in its while loop until process 1 sets *turn* to 0.
- This situation violates condition 3 set out above: process 0 is being blocked by a process not in its critical region.

- In fact, this solution requires that the two processes **strictly alternate** in entering their critical regions, which is not a good idea.

d) Peterson's Solution (from Galvin & Greg)

- A classic software-based solution to the critical-section problem known as **Peterson's solution.**
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered P_o and P_1.
- We use *Pj* to denote the other process; that is, **j** equals **1 - i**.
- if i=0 ➔ j=1
- if i=1 ➔ j=0
- Peterson's solution requires two data items to be shared between the two processes

  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 is ready to enter its critical section. For example, if flag[i] is true, this value indicates that Pi is ready to enter its critical section.

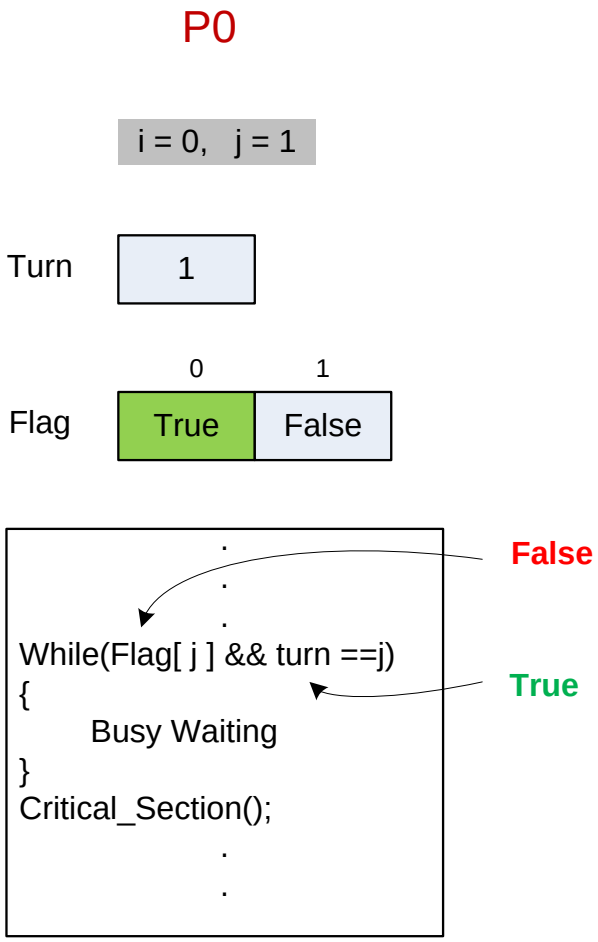
```
do
{
   flag[i] = TRUE;
   turn = j ;

   while (flag[j] && turn == j )
   {
         // Busy Waiting
   }

   Critical_Section();
   flag[i] = FALSE;
   Non_Critical_Section();

} while (TRUE);
```

Figure: The structure of process *Pi,* in Peterson's solution.

- Let’s consider 3 various scenarios to check whether mutual exclusion is achieved through Peterson’s Solution.

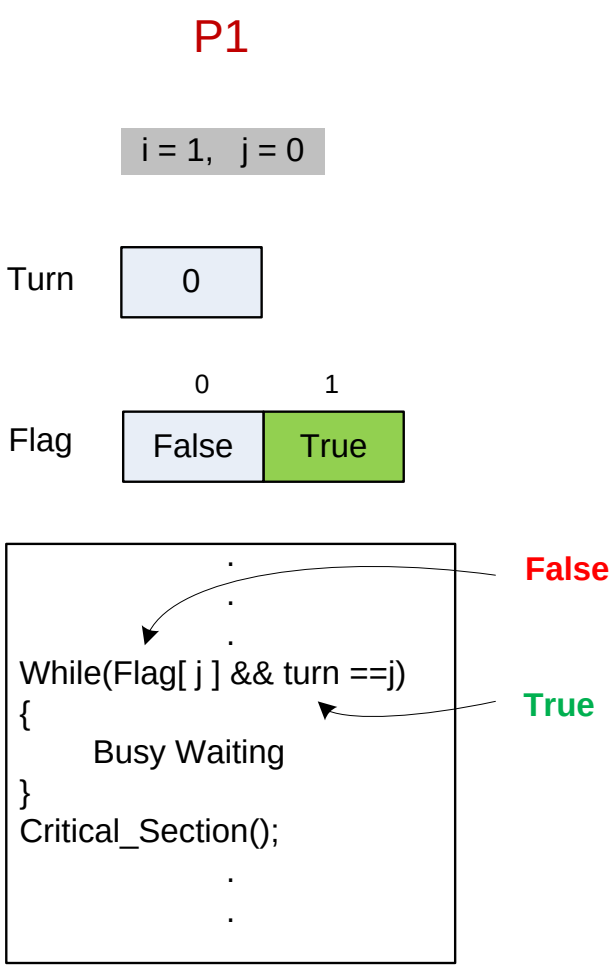
CASE 1 : Only Po is ready to access Critical Section, Not P1.

- In this case, the While loop will be false as shown in the below figure.
- So P0 will not have to wait, and can directly access the Critical_Section.



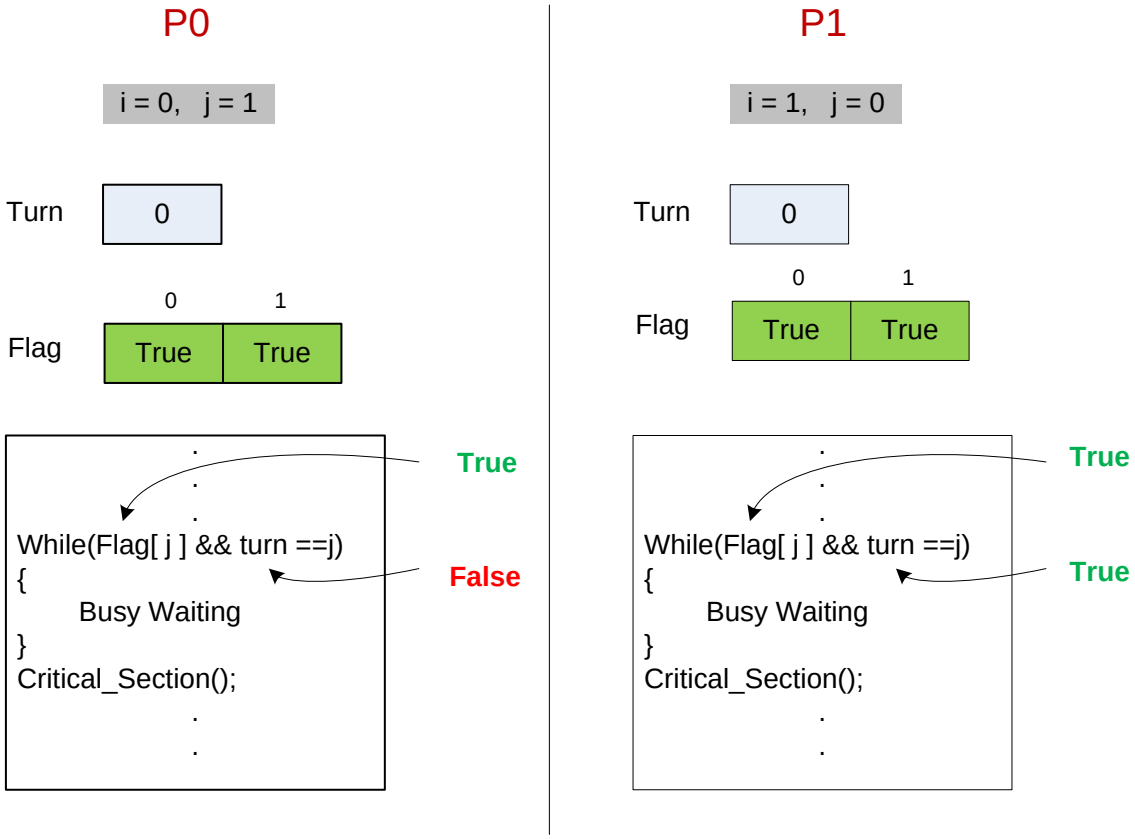
CASE 2 : Only P1 is ready to access Critical Section, Not P0.

- In this case, the While loop will be false as shown in the below figure.
- So P1 will not have to wait, and can directly access the Critical_Section.



CASE 3 : Both P0 and P1 are ready to access Critical Section.

- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- Whichever store is done last is the one that counts; the first one is overwritten and lost. Suppose that P1 stores last, which will set Turn to 0.
- When both processes come to the while statement, P0 executes it zero times and enters its critical region. (as shown in the below figure).
- P1 loops and does not enter its critical region until P0 exits its critical region.



e) Hardware Solution

1. TSL instruction
2. XCHG instruction

1. TSL instruction

- Some computers have an instruction like
- **TSL RX,LOCK**

(Test and Set Lock) that works as follows:

```
enter_region:
        TSL REGISTER,  LOCK
        CMP REGISTER,  #0
        JNE enter_region
RET
leave_region:
        MOVE LOCK,  #0
RET
```

1) TSL REGISTER, LOCK

⇨ Test and Set Lock

⇨ It will move the value of LOCK to REGISTER and will set lock =1.

2) CMP REGISTER, #0

⇨ CMP = Compare

⇨ It will compare the value of REGISTER with 0.

3) JNE enter region

⇨ JNE = Jump if Not Equal to zero.

⇨ if REGISTER was not 0 ➔ then LOCK was not 0

➔ i.e. LOCK is 1

➔ So, a process is already inside critical region, so other process will be prevented from accessing critical region. and process will have to call critical region again, until it gets free.

NOTES :

1) The operations of reading the word and storing into it are guaranteed to be **indivisible**—no other processor can access the memory word until the instruction is finished.
2) When *lock* is 0 ➔ any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets *lock* back to 0 using an ordinary move instruction.
3) How can this instruction be used to prevent two processes from simultaneously entering their critical regions? Before entering its critical region, a process calls *enter region*, which does busy waiting until the lock is free; then it acquires the lock and returns. After leaving the critical region the process calls *leave region*, which stores a 0 in *lock*.

2. XCHG instruction

- An alternative instruction to TSL is XCHG, which exchanges the contents of two locations atomically, for example, a register and a memory word.
- The code same as the solution with TSL.

```
enter_region:
        MOVE REGISTER,#1
        XCHG REGISTER,LOCK
        CMP REGISTER,#0
        JNE enter_region
    RET
leave_region:
        MOVE LOCK,#0
    RET
```

5) The Producer-Consumer Problem

- As an example of how these primitives (sleep and wakeup) can be used, let us consider the producer-consumer problem.
- It is also known as the "bounded-buffer" problem.

1) Two processes share a common, fixed-size buffer.
2) One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.
3) Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, and Producer can be awakened when the consumer has removed one or more items.
4) Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep. The consumer can be awakened when the producer produces one or more items.
5) This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory.
6) To keep track of the number of items in the buffer, we will need a variable, count.
   Count = No. of items in the buffer
   N = Maximum no. of items the buffer can hold
7) The producer's code will test to see if count is N. (**Count = N** indicates that all the buffers are filled, no buffer is empty). If it is, the producer cannot insert any item and will go to sleep. if it is not, the producer will add an item and increment count.
8) The Consumer's code will test to see if count is 0. (**Count = 0** indicates that all the buffers are empty, no buffer is filled). If it is, the consumer cannot remove any item and will go to sleep. If it is nonzero, remove an item and decrement the counter.

```
#define N 100
int count = 0;
void producer(void)
{
        int item;
        while (TRUE)
        {
                item = produce_item( );
                if (count == N)
                {
                        sleep( );
                }
                insert_item(item);
                count = count + 1;
                if (count == 1)
                {
                        wakeup(consumer);
                }
        }
}
void consumer(void)
{
        int item;
        while (TRUE)
        {
                if (count == 0)
                {
                        sleep( );
                }
                item = remove_item( );
                count = count – 1;
```

```
            if ( count == N-1 )
            {
                  wakeup(producer);
            }
            Consume_item(item);
      }
}
```

PROBLEM:

1) Now let us get back to the race condition. It can occur because access to count is unconstrained. As a consequence, the following situation could possibly occur.
2) The buffer is empty and the consumer has just read count to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer.
3) The producer
   - Inserts an item in the buffer,
   - Increments count, notices that it is now 1.
   - Producer thinks that the consumer must be sleeping, and wakes the consumer up.
4) Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost.
5) When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep.
6) Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.
7) "The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost". If it were not lost, everything would work.

SOLUTION:

1) A quick fix is to modify the rules to add a wakeup waiting bit to the picture.
2) When a wakeup is sent to a process that is not yet sleeping, this bit is set.
3) Later, when the process tries to go to sleep, and if the wakeup waiting bit is **on**, then the process will stay awake and wakeup waiting bit will be turned **off**.

6) Semaphores

⇨ **Semaphore:** "An integer variable, which is used to count the no. of wakeups saved for future use"

⇨ **Operations on Semaphores :**

1) down operation on Semaphores

2) up operation on Semaphores

1) down operation on Semaphore

- The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value.
- If the value is 0, the process is put to sleep without completing the down for the moment.
- **"Checking the value,**
  changing it, and
  possibly going to sleep,
  are all done as a single, indivisible atomic action".

- It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked.
- This atomicity is absolutely essential to solve synchronization problems and avoiding race conditions.

2) up operation on Semaphore

- The up operation increments the value of the semaphore.
- If more processes were sleeping on that semaphore, (unable to complete an earlier down operation) one of them is chosen by the system (e.g., at random) and is allowed to complete its down.
- **"Incrementing the Semaphore and waking up one process is also a single, indivisible atomic action".**

⇨ **<u>Types of Semaphores :</u>**

1) Binary Semaphores

2) Counting Semaphores

1) Binary Semaphores:

- It can have values 0/1.
- Two methods associated with it: up, down or lock, unlock

2) Counting Semaphores:

- It is typically used for two things:
  a) Counting Events
  b) Resource Management

a) Counting Events

- Event handler will 'give' a semaphore to a process when it occurs and will 'take' a semaphore when a process is completed.

Count = No. of Events Occurred – No. of Events Completed

- Ex.
  Count = 5 – 3
  5 events occurred,
  Out of 5, 3 processes completed with the resource
- Desirable Count value is Zero.

b) Resource Management
- In this, Count value indicates no. of resources available.
- To obtain control of resource a process must first obtain a semaphore, decrementing the semaphore count value.
- When semaphore value reaches to Zero, there are no free resources.
- When a Process is completed with the resource, it increments the semaphore count value.
- Desirable Count value is Maximum no. of resources.

7) Solving Producer-Consumer Problem with Semaphore

```
#define N 100
typedef int semaphore;

semaphore mutex = 1;
semaphore empty = N;

semaphore full = 0;

void producer(void)
{
        int item;
        while (TRUE)
        {
                item = produce_item( );

                down(&empty);
                down(&mutex);

                insert_item(item);

                up(&mutex);
                up(&full);
        }
}
```

```
void consumer(void)
{
        int item;
        while (TRUE)
        {
                down(&full);
                down(&mutex);

                item = remove_item( );

                up(&mutex);
                up(&empty);

                consume_item(item);
        }
}
```

- Semaphores solve the lost-wakeup problem.
- To make them work correctly, it is essential that they be implemented in an ***indivisible way***.
- The normal way is to implement up and down as system calls, with the OS briefly ***disabling all interrupts*** while
  - it is testing the semaphore,
  - updating it,
  - and putting the process to sleep, if necessary.
- As all of these actions take only a few instructions, no harm is done in disabling interrupts.
- This solution uses three semaphores:
  1) full - for counting the number of slots that are full,
  2) empty - for counting the number of slots that are empty,
  3) mutex - to make sure the producer and consumer do not access the buffer at the same time.
- Initially

  full = 0,

empty = N,
mutex = 1. Semaphores that are initialized to 1

- **binary semaphores**: The Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called binary semaphores.

"If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed.**"**

8) Classical IPC Problems

a. **Dining Philosopher Problem**
b. **Readers and Writers Problem**

a. Dining Philosopher Problem

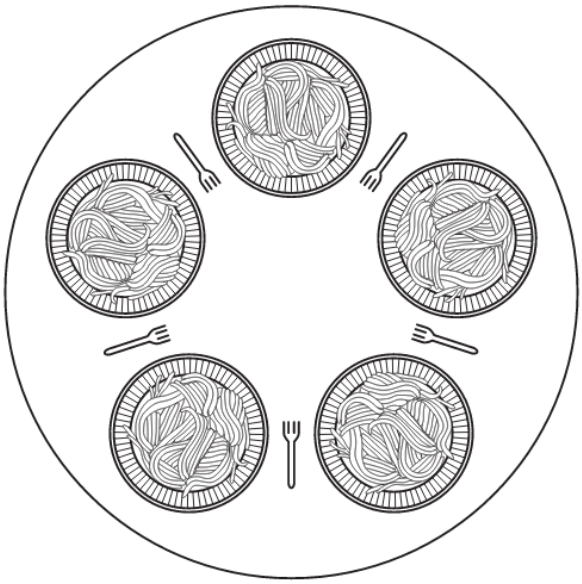


Fig : Lunch time in philosophy department

1) The problem can be stated as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates there is one fork.
2) The life of a philosopher consists of alternating periods of eating and thinking.
3) When a philosopher gets hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think.
4) The key question is: Can you write a program for each philosopher that does what it is supposed to do and never gets stuck?
5) The code shows the obvious solution.
6) The procedure take_fork() **waits** if the specified fork is not available and when it becomes available then it **takes** it.

```
#define N 5
Void philosopher(int i)
{
        While(TRUE)
        {
                think();
                take_fork(i);
                take_fork((i+1)%N);
                eat():
                put_fork(i);
                put_fork((i+1)%N);
        }
}
```

PROBLEM:

Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

SOLUTION:

Several possible solutions to the deadlock problem are listed next.

1) Allow at most four philosophers to be sitting simultaneously at the table.
2) Allow a philosopher to pick up her forks only if both forks are available.
3) Use an asymmetric solution; that is, an odd philosopher picks up first her left fork and then her right fork, whereas an even philosopher picks up her right fork and then her left fork.

b. The Readers and Writers Problem

1) Another famous problem is the readers and writers problem.
2) A database is to be shared among several processes. Some of these processes may want only to read the database, whereas others may want to update (write) the database.
3) It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers.
4) The question is how do you program the readers and the writers?
5) One solution is shown in the following code:

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
	while (TRUE)
	{
		down(&mutex);
		rc = rc + 1;
		if (rc == 1)
		{
			down(&db);
		}
		up(&mutex);

		read_data_base( );

		down(&mutex);
		rc = rc −1;
		if (rc == 0)
		{
			up(&db);
		}
		up(&mutex);

		use_data_read( );
	}
}
```

```
void writer(void)
{
      while (TRUE)
      {
            Think_up_data( );
            down(&db);
            write_data_base( );
            up(&db);
      }
}
```

Code Description:

1) In this solution, the **first** reader to get access to the database does a down on the semaphore *db*. Subsequent readers merely increment a counter, *rc*. As readers leave, they decrement the counter, and the **last** to leave does an up on the semaphore, allowing a blocked writer, if there is one, to get in.
2) Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. Additional readers can also be admitted if they come along.
3) Now suppose a writer shows up. The writer may not be admitted to the database, since writers must have exclusive access, so the writer is suspended. Later, additional readers come along. As long as at least one reader is still active, subsequent readers are admitted.
4) As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present.

5) If a new reader arrives, say, every 2 sec, and each reader takes 5 sec to do its work, the writer will never get in.
6) To avoid this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately.
7) In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it.