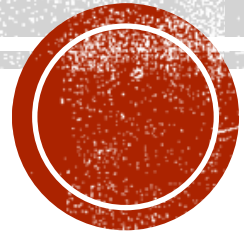
CH 2.2: IPC & SYNCHRONIZATION



INTER PROCESS COMMUNICATION AND SYNCHRONIZATION

- 1. How one process can pass the information to the another?
- 2. How to make sure two or more processes do not get into each other's way when engaging in critical activities?
  - Two Process trying to grab last seat on a plane.
- 3. How to maintain the proper sequence when dependencies are presents?
  - If process A produces data and process B tries to print them.

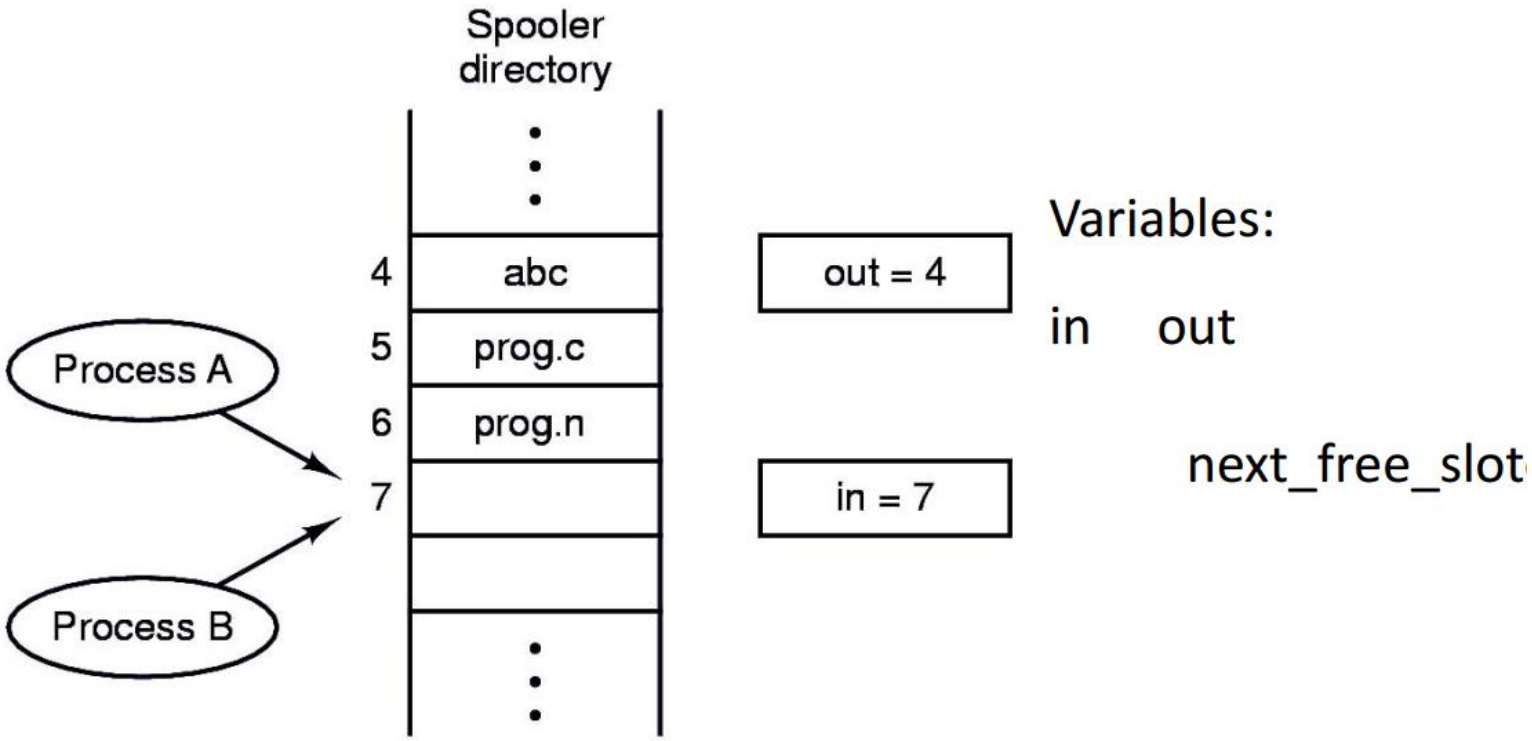
- In case of Threads,
  - Same problem(2&3) and same solution exits.

RACE CONDITION

- Imagine what might happen if two people tried to turn on the light using two different switches at exactly the same time.

Thread A		Thread B		
Instruction	Register	Instruction	Register	Count
		LOAD Count	10	10
LOAD Count	10			10
ADD #1	11			10
STORE Count	11			11
		SUB #1	9	11
		STORE Count	9	9

RACE CONDITION



RACE CONDITION

- Situation where two or more processes are reading or writing some shared data and the final result depends on who run precisely, are called **Race Conditions.**
- Debugging is most difficult.
- Problem occurs once in a while.

RACE CONDITION

- Possibilities of race:
  - many concurrent process read the same data.
  - one process reading and another process writing same data.
  - two or more process writing same data.

- Solution: *prohibiting more than one process from reading writing the same data at the same time-* ***Mutual Exclusion****.*
- Mutual Exclusion:
  Some way of making sure that if one process is using a shared variables or files, the other process will be excluded from doing the same thing.

CRITICAL REGIONS

- Mutual Exclusion:
  - Prohibit more than one process from reading and writing the shared data at the same time.
- critical region / *critical section.*
  - Part of the program where shared memory is accessed.
  - *Code executed by the process can be grouped into sections, some of which require access to shared resources, and other that do not. The section of the code that require access to shared resources are called critical section.*

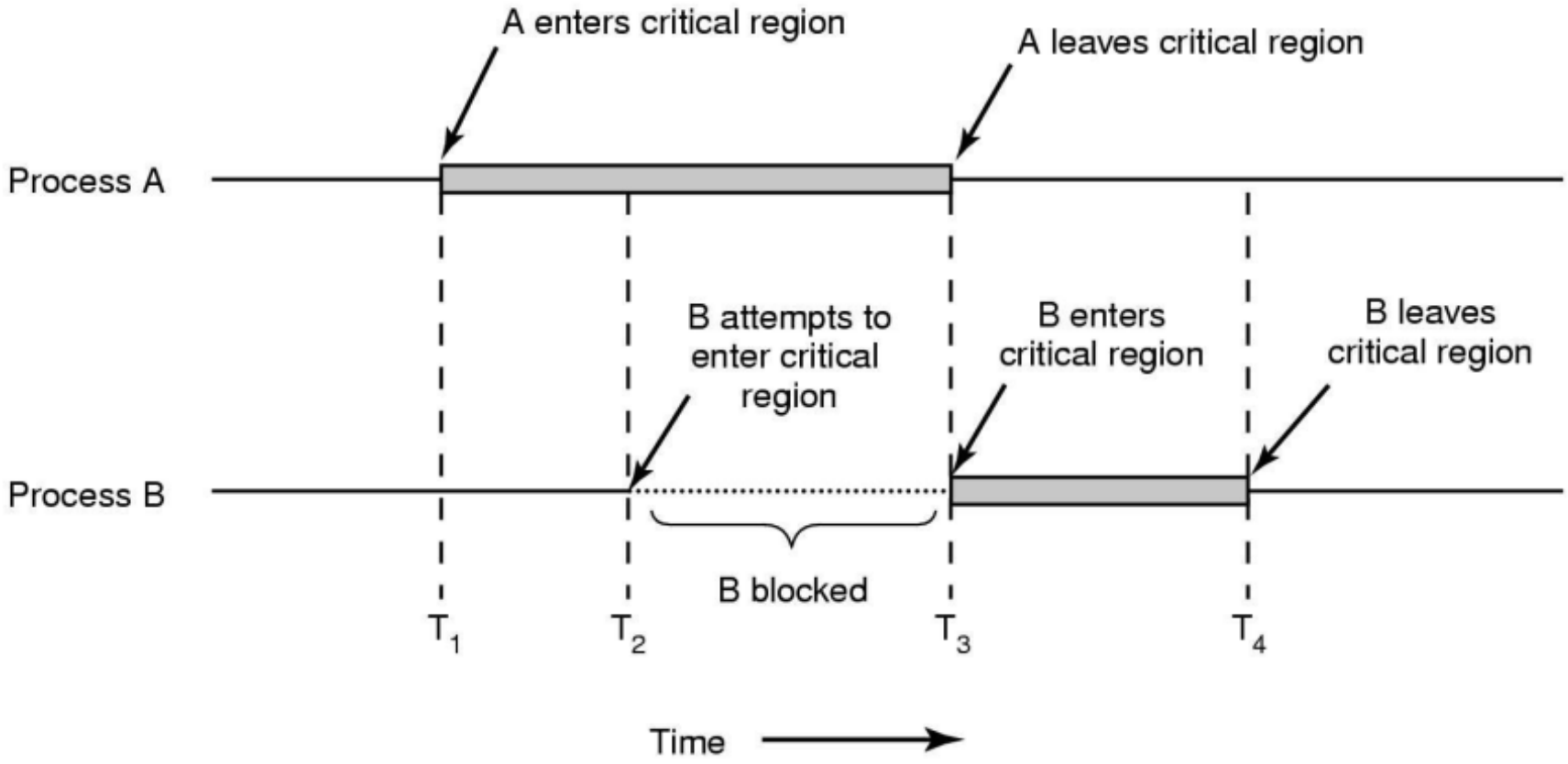
GENERAL STRUCTURE

- while(true){
  entry_section
  critical_section
  exit_section
  reminder_section
  }

SOLUTION TO AVOID RACE CONDITION

- 1. No two processes may be simultaneously inside their CRs(mutual exclusion).
- 2. No assumptions may be made about the speeds or number of CPUs.
- 3. No process running outside its CR may block other process.
- 4. No process should have to wait forever to enter its CR.

MUTUAL EXCLUSION TO AVOID RACE CONDITION



Mutual exclusion using critical regions

VARIOUS METHODS TO ACHIEVE ME(BUSY WAITING)

- Disabling Interrupts
- Lock Variables
- Strict Alternation
- Peterson's Solutions
- Test and Set Lock (TSL)

DISABLING INTERRUPTS

- *Each process disable all interrupts just after entering its CR and reenable them just before leaving it.*
- No clock interrupt, no other interrupt, no CPU switching to other process until the process turn on the interrupt.
-

```
DisableInterrupt()
        // perform CR task
EnableInterrupt()
```

- Advantages:
  - Mutual exclusion can be achieved by implementing OS primitives to disable and enable interrupt.
- Problems:
  - allow the power of interrupt handling to the user.
  - The chances of never turned on – is a disaster.
  - it only works in single processor environment.

LOCK VARIABLES

- *A single, shared (lock) variable, initially 0.*
- *When a process wants to enter its CR, it first test the lock. If the lock is 0, the process set it to 1 and enters the CR. If the lock is already 1, the process just waits until it becomes 0.*
- Advantages:
  - seems no problems.
- Problems:
  - problem like spooler directory;
  - suppose that one process reads the lock and sees that it is 0, before it can set lock to 1, another process scheduled, enter the CR, set lock to 1 and can have two process at CR (violates mutual exclusion).

STRICT ALTERNATION

- *Processes share a common integer variable **turn**. If turn == i then process Pi is allowed to execute in its CR, if turn == j then process Pj is allowed to execute.*

- While (true){
  while(turn!=i); /*loop */
  critical_section();
  turn = j;
  noncritical_section();
  }

  Process Pi

  While (true){
  while(turn!=j); /*loop*/
  critical_section();
  turn = i;
  noncritical_section();
  }

  Process Pj

- **Advantages:** Ensures that only one process at a time can be in its CR.
- **Problems:** strict alternation of processes in the execution of the CR.

  What happens if process i just finished CR and again need to enter CR and the process j is still busy at non-CR work? (violate condition 3)

PETERSON'S SOLUTION

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

PETERSON'S SOLUTION

- Before entering its CR, each process call *enter_region* with its own process number, 0 or 1 as parameter.
- Call will cause to wait, if need be, until it is safe to enter.
- When leaving CR, the process calls *leave_region* to indicate that it is done and to allow other process to enter CR.
- Advantages: Preserves all conditions.
- **Problems:** difficult to program for n-processes system and less efficient.

HARDWARE SOLUTION - TSL

- ***help from the hardware***

- **Test and Set Lock (TSL)**
  - instruction reads the contents of the memory word lock (shared variable) into the register and then stores nonzero value at the memory address lock.
  - Reading and storing are indivisible.
  - The CPU executing TSL locks the memory bus to prohibit other CPUs from accessing memory until it is done.
  - When lock is 0, any process may set it to 1 using the TSL instruction.

HARDWARE SOLUTION - TSL

- *enter_region:*

```
TSL register, lock      | copy lock to register and set lock to 1
CMP register, #0        | was lock 0?
JNE enter_region        | Jump if not equal to zero, lock was set, so loop
RET                     | return to caller;
```

  - *critical_region();*

- *leave_region:*

```
MOVE lock, #0           | store a 0 in lock
RET                     | return to caller
```

  - *noncritical_section();*

- **Advantages:** *Preserves all condition, easier programming task and improve system efficiency.*
- ***Problems:*** *difficulty in hardware design.*

ALTERNATE COMMAND TO TSL: XCHG

- enter_ region:

```
        MOVE REGISTERS          | put a 1 in the register
        XCHG REGISTER.LOCK      | swap the contents of the register and lock variable
        CMP REGISTER,#0         | was lock zero?
        JNE enter.region        | if it was non zero, lock was set, so loop
        RET                     | return to caller; critical region entered
```

leave_region:

```
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

BUSY WAITING ALTERNATIVES???

- **Busy waiting:**
  When a process want to enter its CR, it checks to see if the entry is allowed, if it is not, the process just sits in a tight loop waiting until it is.

- **Waste of CPU time for NOTHING!**
- **Priority Inversion Problem**
  H with high priority and L with low priority.
  L is in CR and H becomes ready to run after I/O completion
  H is scheduled and keep busy waiting loop
  L never gets chance to leave CR

- ***Sleep* and *Wakeup*** pair instead of waiting.
  Sleep causes to caller to block, until another process wakes it up.

SLEEP AND WAKEUP

- **Producer-Consumer Problem**
- Two process share a common, fixed sized buffer.
- Suppose one process, producer, is generating information that the second process, consumer, is using.
- Their speed may be mismatched, if producer insert item rapidly, the buffer will full and go to the sleep until consumer consumes some item, while consumer consumes rapidly, the buffer will empty and go to sleep until producer put something in the buffer.

```
#define N = 100 /* number of slots in the buffer*/
int count = 0; /* number of item in the buffer*/
void producer(void)
{
        int item;
        while(TRUE){                                    /* repeat forever*/
                item = produce_item();                  /*generate next item*/
                if(count == N) sleep();                 /* if buffer is full go to sleep*/
                insert_item(item);                      /* put item in buffer */
                count = count +1;                       /*increment count */
                if (count == 1)wakeup(consumer);        /*was buffer full ?*/
        }
}
```

```
void consumer(void)

{
        int item;
        while(TRUE){                                    /* repeat forever*/
                if(count == 0)sleep();                  /*if buffer is empty go to sleep*/
                item = remove_item();                   /*take item out of buffer*/
                count = count -1;                       /*decrement count*/
                if (count == N-1) wakeup(producer);     /* was buffer empty*/
                consume_item();                         /*print item*/
        }
}
```

PRODUCER CONSUMER PROBLEM

- leads to race as in spooler directory.

- What happen if
  When the **buffer is empty, the consumer just reads count and quantum is expired**, the producer inserts an item in the buffer, increments count and wake up consumer. The consumer not yet asleep, so the wakeup signal is lost, the consumer has the count value 0 from the last read so go to the sleep. Producer keeps on producing and fill the buffer and go to sleep, both will sleep forever.

- *Think: If we were able to save the wakeup signal that was lost.....*

SEMAPHORES

- *E.W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups, called a semaphore.*
- It could have the value 0, indicating no wakeups ware saved, or some positive value if one or more wakeups were pending.
- **Operations:** *Down* and *Up* (originally he proposed *P* and *V* in Dutch and sometimes known as wait and signal)
- ***Down()*:** checks if the value greater than 0
  - yes- decrement the value (i.e. Uses one stored wakeup) and continues.
  - No- process is put to sleep without completing down.
  - **Indivisible Atomic Action**
    - Checking value, changing it, and possibly going to sleep, is all done as single action.
    - Implemented as System Call and disable interrupt or use TSL if multiprocessor are used.
- ***Up():*** increments the value;
  - if one or more processes were sleeping, unable to complete earlier down operation, one of them is chosen and is allowed to complete its down.
  - incrementing the semaphore and waking up one process is also indivisible.
- Semaphores which allow an arbitrary resource count are called **counting semaphores**, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores(Mutex)** and are used to implement locks.

SEMAPHORES: PRODUCER CONSUMER

```
#define N 100                                   /*number of slots in buffer*/
typedef int semaphore;                          /*defining semaphore*/
semaphore mutex = 1;                            /* control access to the CR */
semaphore empty = N;                            /*counts empty buffer slots*/
semaphore full = 0;                             /*counts full buffer slots*/

void producer(void)
{
        int item;
        while(TRUE){                                    /*repeat forever */
                item = produce_item();                  /*generate something */
                down(empty);                            /*decrement empty count*/
                down(mutex);                            /*enter CR */
                insert_item();                          /* put new item in buffer*/
                up(mutex);                              /* leave CR*/
                up(full);                               /*increment count of full slots*/
        }
}
```

```
void consumer(void)
{
        int item;
        while(TRUE){                                    /*repeat forever*/
                down(full);                             /*decrement full count */
                down(mutex);                            /*enter CR*/
                item = remove_item();                   /*take item from buffer*/
                up(mutex);                              /*leave CR*/
                up(empty);                              /*increment count of empty slots*/
                consume_item();                         /* print item*/
        }
}
```

MESSAGE PASSING

- Based on two primitives: Send and Receive
- Can be easily implemented in library procedure as **System Calls**
  - **send**(destination, &message);
    receive(source, &message);
  - former call sends a message to a given destination and the latter one receives a message from a given source
- accessible from many programming language environments
- No shared memory.
- Messages sent but not yet received are buffered automatically by OS, it can save N messages.
- The total number of messages in the system remains constant, so they can be stored in given amount of memory known in advance.

DESIGN ISSUES FOR MESSAGE-PASSING SYSTEMS

- What if two different processes are in different m/c connected by network.
  - Message can be lost in network
  - Require Special ACK msg
  - Retransmits if ACK not received
- What if ACK packet is lost
  - Sender retransmits message
  - Receiver must be able to distinguish between retransmit and new message
  - Sequence Number
- Message systems also have to deal with the question of how processes are named, so that the process specified in a send or receive call is unambiguous.
- Authentication is also an issue in message systems: how can the client tell that it is communicating with the real file server, and not with an imposter?
- Performance issues when the sender and receiver are on the same machine
- Copying messages from one process to another is always slower than doing a semaphore operation

PRODUCER CONSUMER PROBLEM USING MESSAGE PASSING

```
#define N 100                                   /* number of slots in the buffer */
void producer(void)
{
        int item;
        message m;                              /* message buffer */
        while (TRUE) {
                item = produce.item();          /* generate something to put in buffer */
                receive(consumer, &m);          /* wait for an empty to arrive */
                build_message(&m, item) ;       /*construct a messgae to send*/
                send(consumer, &m);             */ send item consumer */
        }
}
```

```
void consumer(void)
{
        int item, i;
        message m;
        for (i = 0; i < N; i++) send(producer, &m);        /* send N empties */
        while (TRUE) {
                receive(producer, &m);                     /* get message containing item */
                item = extract_item(&m);                   /* extract item from message */ *
                send(producer, &m);                        /* send back empty reply */
                consume_item(item);                        /* do something with the item */
        }
}
```

IMPLEMENTING MESSAGE PASSING:

- *Direct addressing:*
  - provide ID of destination.
- *Indirect addressing:*
  Send to a mailbox.
- *Mail box:* It is a message queue that can share by multiple senders and receivers, senders sends the message to the mailbox while the receiver pick ups the message from the mailbox.

DINING PHILOSOPHER PROBLEM

- Scenario:
  - The five philosophers are seated in a common round table for their lunch, each philosopher has a plate of spaghetti and there is a forks between two plates, a philosopher needs two forks to eat it. They alternates thinking and eating.
  - *What is the solution (program) for each philosopher that does what is supposed to do and never got stuck?*

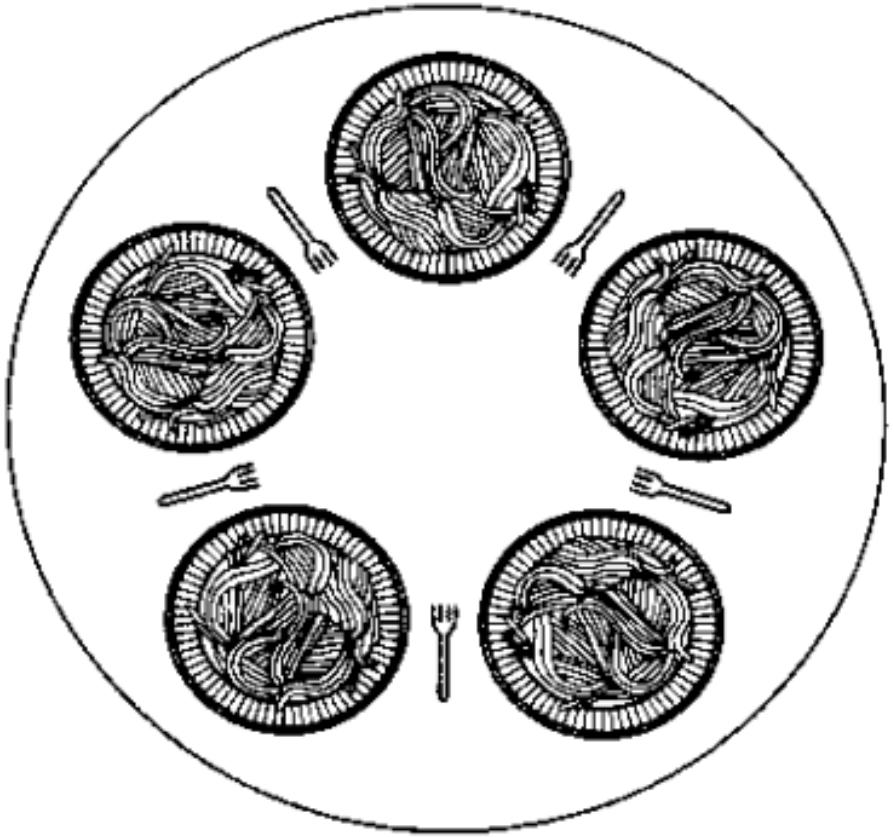


Figure 2-31. Lunch time in the Philosophy Department.

DINING PHILOSOPHER PROBLEM

- **Attempt 1:** The Obvious Solution.
- When the philosopher is hungry it picks up a fork and wait for another fork, when get, it eats for a while and put both forks back to the table.
- *Problem: What happens, if all five philosophers take their left fork simultaneously?*

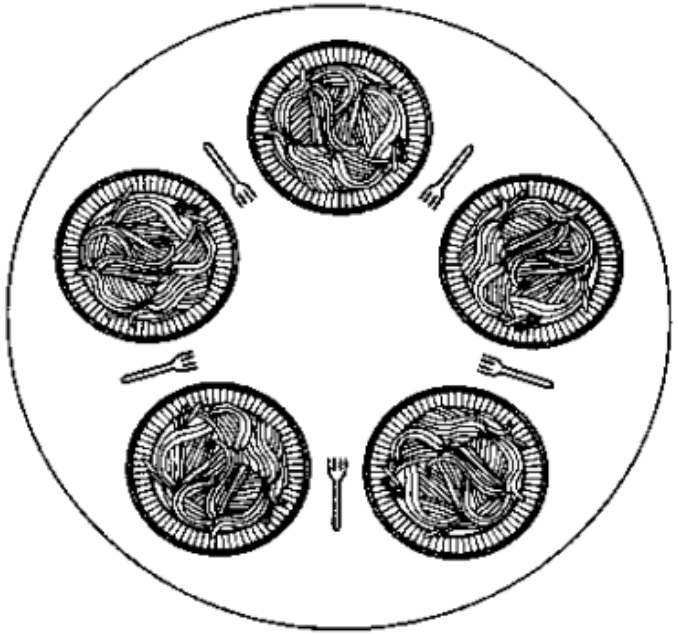


Figure 2-31. Lunch time in the Philosophy Department.

A NON SOLUTION DINING PHILOSOPHER PROBLEM

```
#define N 5;                    /* No of philosophers */

void philosopher(int i)  /*i is philosopher number from 0~4*/
{
        while(true){
        think();                        /* philosopher is thinking*/
        take_fork(i);                   /*take left fork*/
        take_fork((i+1)%N)      /*take right fork % is modulo*/
        eat();
        put_fork(i);                    /*put left fork on the table*/
        put_fork((i+1)%N);      /*put right fork on the table*/

        }

}
```

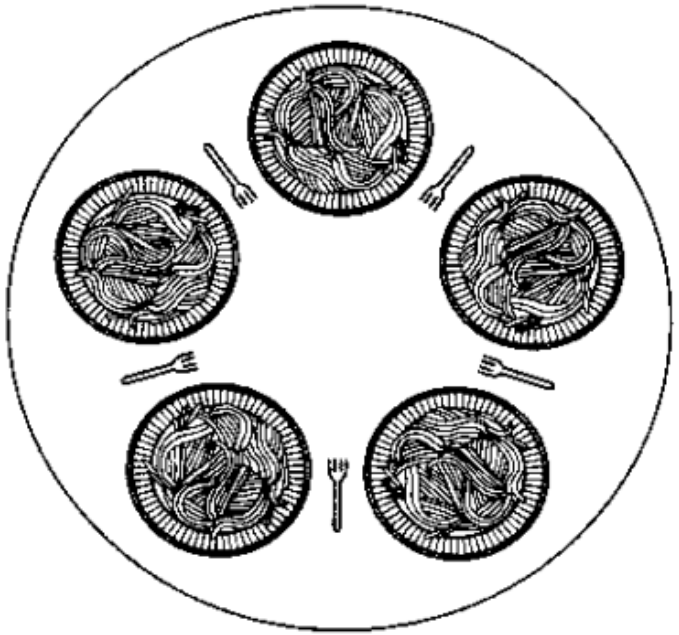


Figure 2-31. Lunch time in the Philosophy Department.

DINING PHILOSOPHER PROBLEM

- **Attempt 2: Little bit modification to** The Obvious Solution.
- When the philosopher is hungry it picks up a fork and wait for another fork, when she doesn't get right, she puts left fork back to table and try again later after a while.

- *Problem: What happens, if all five philosophers take their left fork simultaneously and put back together and unfortunately try again after some time at the same time.*

- ***Starvation: A situation in which where all the programs continue to run indefinitely but fail to make any progress due to unavailability of resources is called starvation.***

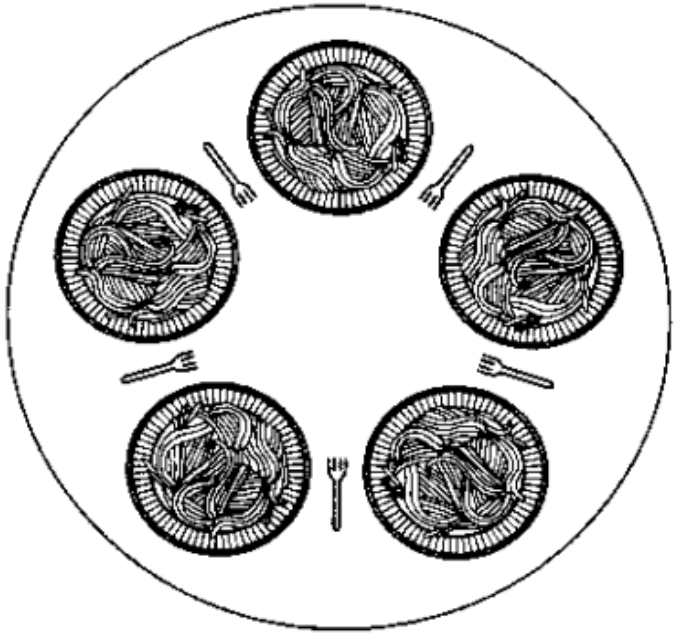


Figure 2-31. Lunch time in the Philosophy Department.

DINING PHILOSOPHER PROBLEM

- **Attempt 3**:
- Using semaphore, before starting to acquire a fork he would do a down on mutex, after replacing the forks, he would do up on mutex.

- ***Problem:*** *adequate but not perfect: only one philosopher can be eating at any instant.*

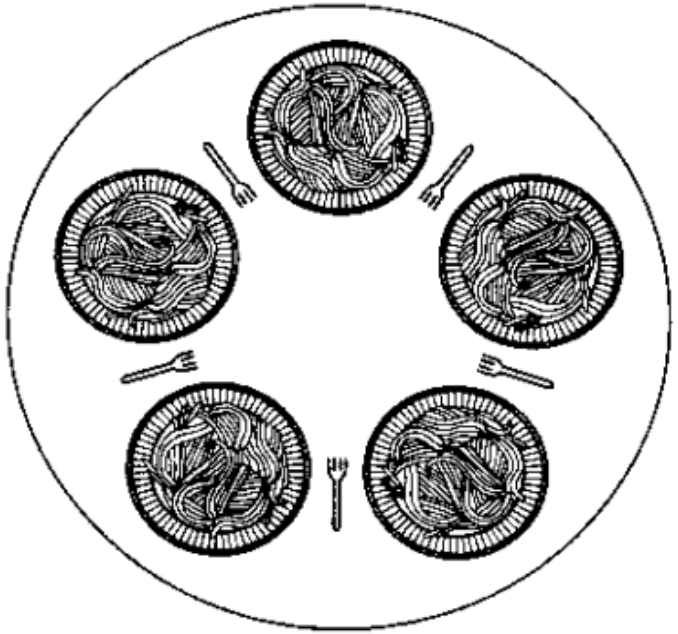


Figure 2-31. Lunch time in the Philosophy Department.

DINING PHILOSOPHER PROBLEM

- **Attempt 4:**
- 3 states
  - EATING
  - THINKING
  - HUNGRY

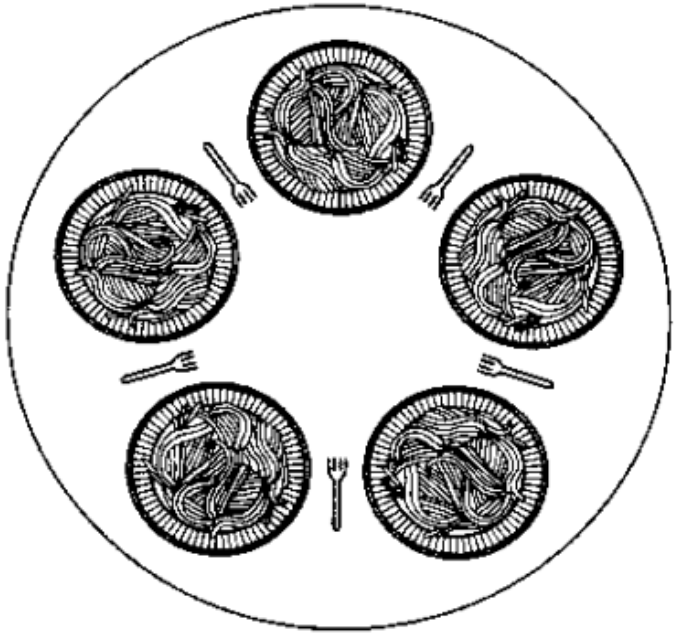


Figure 2-31. Lunch time in the Philosophy Department.

  - A Philosopher goes to eating state if neither neighbors are in eating state.
  - Philosophers neighbors are defined by macros left and right. If I is 2 left is 1 and right is 3.