InterProcess Communication

Reading: Section 2.3 from TextBook (Tanenbaum)

–How one process can pass the information to the another?

–How to make sure two or more processes do not get into each other's way when engaging in critical activities?

–How to maintain the proper sequence when dependencies are presents?

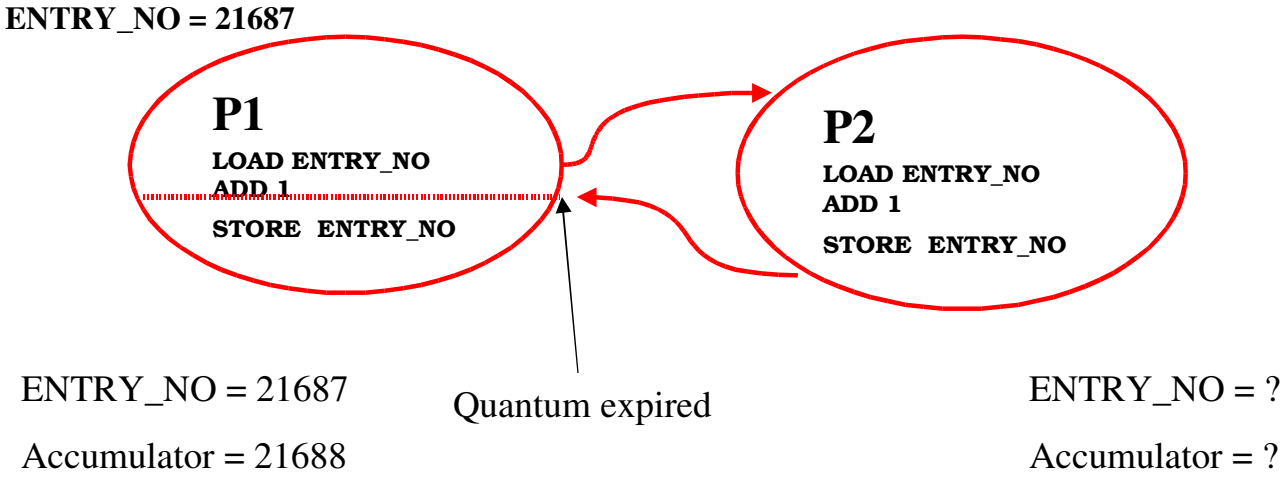
Thus, InterProcess Communication (IPC) and synchronization

IPC provide the mechanism to allow the processes to communicate and to synchronize their actions.

Race Conditions

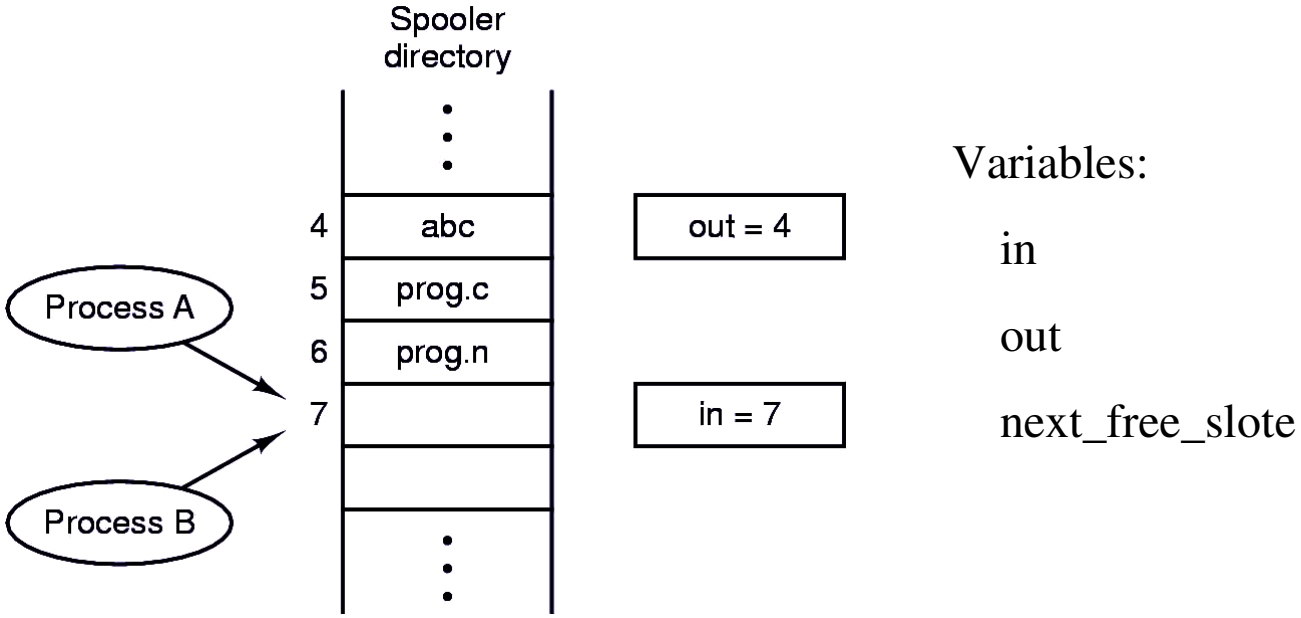
Scenario1: Suppose two user working in a computer, there is a mechanism to monitor continuously the total number of lines each users have entered, each user entry is recorded globally, different process monitoring different users .

Each process : LOAD ENTRY_NO
ADD 1
STORE ENTRY_NO



Race Conditions

Scenario2: a print spooler: when a process wants to print a file, it enters the file name in a special spooler directory, another process, print daemon, periodically checks to see if there are any files to be printed, print them and removes their names from the directory.



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

Mutual Exclusion

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 Section

Problem: How to avoid race?

Fact: The part of the time, process is busy doing internal computations and other things that do not lead to the race condition.

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.

Critical Section

General structure of process P_i (other process P_j)

```
while(true){
        entry_section
                critical_section
        exit_section
                reminder_section
        }
```

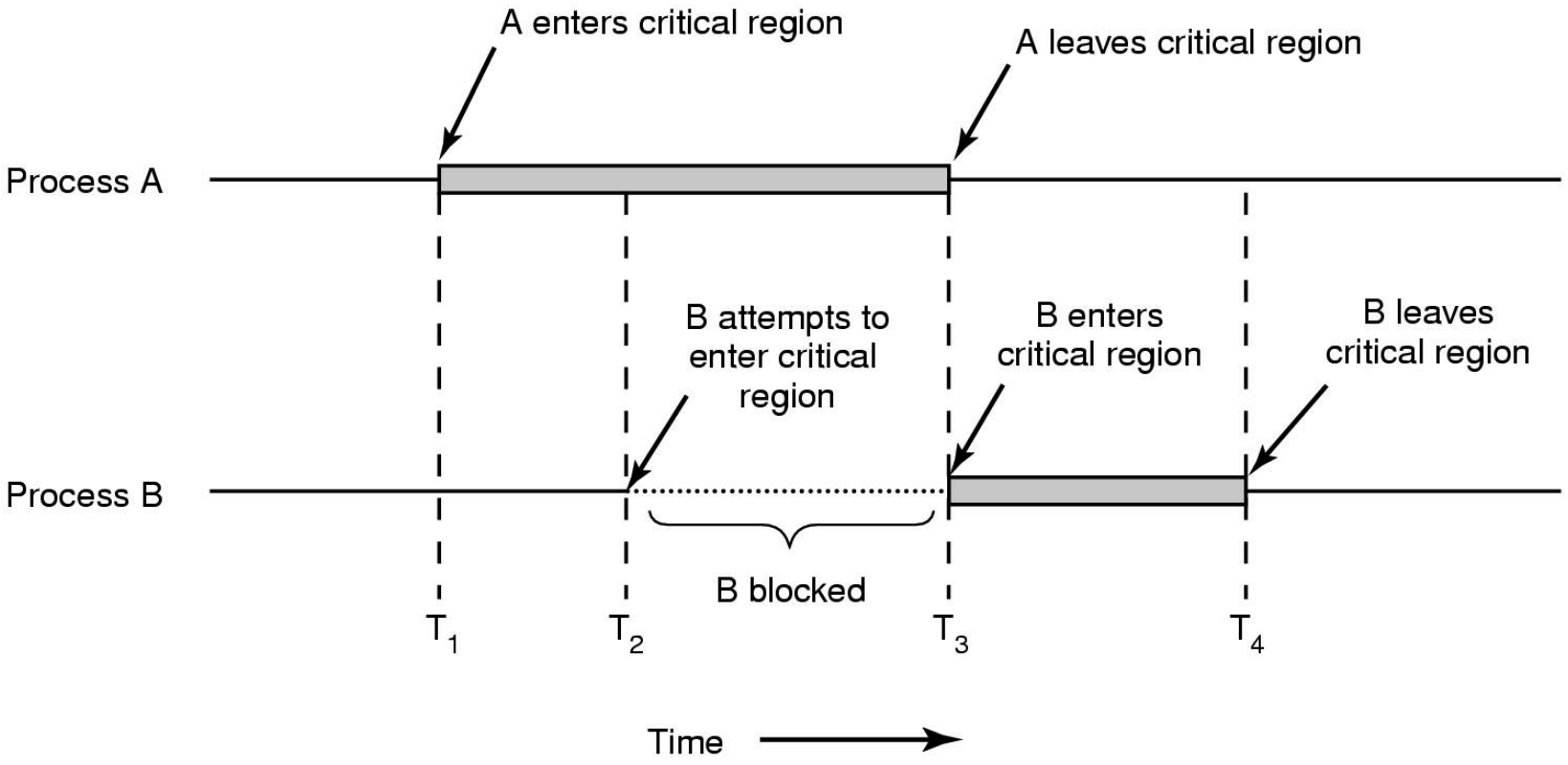
- When a process is accessing a shared modifiable data, the process is said to be in critical section.
- All other processes (those access the same data) are excluded from their own critical region.
- All other processes may continue executing outside their CR.
- When a process leaves its critical section, then another processes waiting to enter its own CR should be allowed to proceed.

Critical Section

CR must satisfy the following conditions:

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.

Critical Section



Mutual exclusion using critical regions

ME with Busy Waiting

Interrupt Disabling

Each process disable all interrupts just after entering its CR and re-enable 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.

ME with Busy Waiting

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).

ME with Busy Waiting

Strict Alternation

Processes share a common integer variable turn. If turn == i then process P_i is allowed to execute in its CR, if turn == j then process P_j is allowed to execute.

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

Process P_i

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

Process P_j

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)

ME with Busy Waiting
Peterson's Algorithm

```
#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 */
}
```

ME with Busy Waiting
Peterson's Algorithm

- 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.

Think: *How it preserves all conditions?*

ME with Busy Waiting
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.
- This automatically as in a un-interruptible time unit.
- 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.
- When it is done the process lock back to 0.

ME with Busy Waiting
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        |if it was non 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.*

Busy Waiting Alternate?

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!

Possibility of *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.

Sleep and Wakeup

Producer-Consumer Problem

```
#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 empty*/
   }
  }
```

```
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 full ?*/
  consume_item();      /*print item*/
}
}
```

Sleep and Wakeup

Producer-Consumer Problem

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.
- Checking value, changing it, and possibly going to sleep, is all done as single action.

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.

Semaphores

```
typedef int semaphore S;
void down(S)
        { if(S> 0)  S--;
          else      sleep();
        }
void up(S)
        {if(one or more processes are sleeping on S)
                  one of these process is proceed;
         else     S++;
        }
while(TRUE){
        down(mutex)
        critical_region();
        up(mutex);
        noncritical_region();
        }
```

Semaphores

Producer-Consumer using Semaphore

```
#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*/
        }
}
```

Semaphores

Producer-Consumer using Semaphore

```
void consumer(void)
{
      int item
```

Use of Semaphore

1. *To deal with n-process critical-section problem.*

   The n processes share a semaphore, (e. g. mutex) initialized to 1.

2. *To solve the various synchronizations problems.*

   For example two concurrently running processes: P1 with statement S1 and P2 with statement S2, suppose, it required that S2 must be executed after S1 has completed. This problem can be implemented by using a common semaphore, synch, initialized to 0.

```
P1:   S1;
      up(synch);
P2:   down(synch);
      S2;
```

Criticality Using Semaphores

All process share a common semaphore variable mutex, initialize to 1. Each process must execute down(mutex) before entering CR, and up(mutex) afterward. *What happens when this sequence is not observed?*

1. When a process interchange the order of *down* and *up* operation: causes the multiple processes in CR simultaneously.
2. When a process replace *up* by *down*: causes the dead lock.
3. When a process omits *down* or *up* or both: violated mutual exclusion and deadlock occurs.

A subtle error is capable to bring whole system grinding halt!!

Monitors

Higher level synchronization primitive.
A monitor is a programming language construct that guarantees appropriate access to the CR. It is a collection of procedures, variables , and data structures that are all grouped together in a special kinds of module or package.
Processes that wish to access the shared data, do through the execution of monitor functions.
Only one process can be active in a monitor at any instant.
Compiler level management.

```
Monitor monitor_name
   {
     shared variable declarations;
     procedure p1(){ .....  }
     procedure p2(){ ....  }
                    ....                    .
     procedure pn(){ ...}
     {initialization code;}
  }
```

Producer Consumer Using Monitors

```
Monitor ProducerConsumer
  {
    int count;
    condition full, empty;
    void insert(int item){
          if (count == N ) wait(full);
          insert_item(item);
          count++;
         if (count ==1) signal(empty)
         }
     void remove(){
          if(count ==0) wait(empty);
          remove_item();
          count--;
          if(count ==N-1) signal(full);
          }
    count =0;
    };
```

```
Void producer(){
      while(TRUE){
            item = produce_item();
            ProcedureConsumer.insert(item);
             }
      }
void consumer(){
      while(TRUE){
           item = ProduceConsumer.remove();
            consume_item();
            }
      }
```

Problems with Monitors

1. Lack of implementation in most commonly used programming languages.

   How to implement in C?

2. Both semaphore and monitors are used to hold mutual exclusion in multiple CPUs that all have a common memory, but in distributed system consisting multiple CPUs with its own private memory, connected by LAN, non of these primitives are applicable.

Semaphores are too low level and monitors are not usable except in a few programming language.

Message Passing

With the trend of distributed operating system, many OS are used to communicate through Internet, intranet, remote data processing etc.

Interprocess communication based on two primitives: send and receive.

send(destination, &message);

receive(source, &message);

The send and receive calls are normally implemented as operating system calls accessible from many programming language environments.

Producer-Consumer with 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 */
              receive(consumer, &m);     /*wait for an empty to arrive*/
              build_message(&m, item);  /*construct a message to send*/
              send(consumer, &m); }
}
void consumer(void)
{      int item;
       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 item*/
       }}
```

Message Passing

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.

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.

Classical IPC Problems

The Dining Philosophers Problem

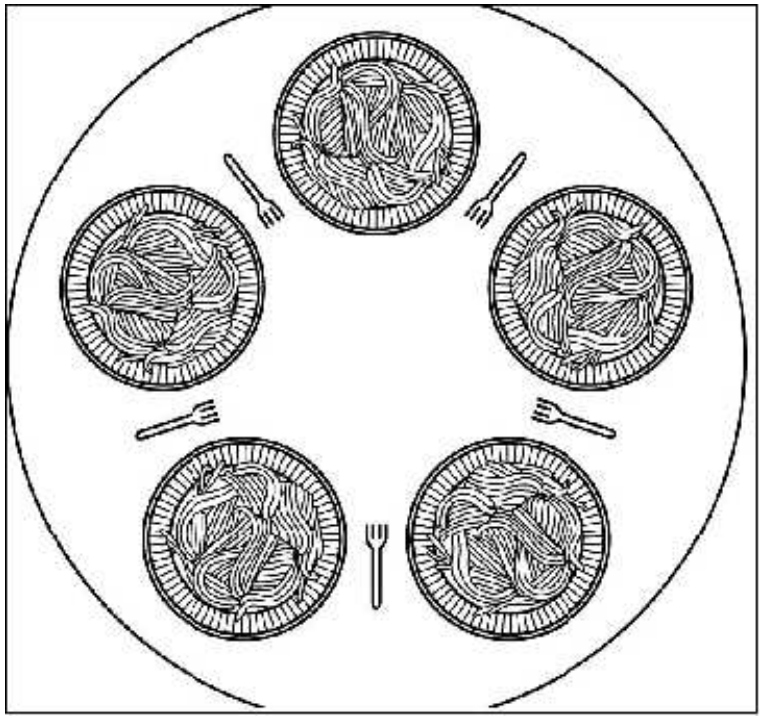
Scenario: consider the five philosophers are seated in a common round table for their lunch, each philosopher has a plate of spaghetti and there is a forks between two plates, a philosopher needs two forks to eat it. They alternates thinking and eating.

What is the solution (program) for each philosopher that does what is supposed to do and never got stuck?

Solution

Attempt 1: 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 happen, if all five philosophers take their left fork simultaneously?



Classical IPC Problems

The Dining Philosophers Problem

Attempt 2: After taking the fork, checks for right fork. If it is not available, the philosopher puts down the left one, waits for some time, and then repeats the whole process.

Problem: What happen, if all five philosophers take their left fork simultaneously?

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.

Attempt 4: Using semaphore for each philosopher, a philosopher move only in eating state if neither neighbor is eating. -*perfect solution*.

Read:

Readers and writer, sleeping barber and Cigarette-smokers.

Home Works

HW #4

1. Textbook (Tanenbaum): 18, 19, 20, 21, 22, 23.
2. What is the meaning of busy waiting? What others kinds of waiting are in OS? Compare each types on their applicability and relative merits.
3. Show the Peterson's algorithm preserve mutual exclusion, indefinite postponement and dead lock.
4. Compare the use of monitor and semaphore operations.

Reading: Section 2.5 of Textbook (Tanenbaum)