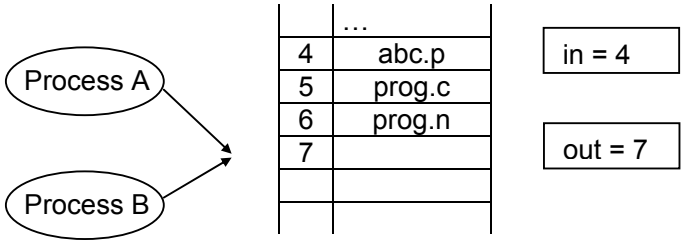
Chapter 6
Interprocess Communication

6.1 Introduction

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 down the line. Thus, there is a need for communication between processes, preferably in a well-structured way not using interrupts. Because, interrupts decrease system performance. That communication between processes in the control of the OS is called as *Interprocess Communication or simply IPC.*

In some operating systems, processes that are working together often share some common storage area that each can read and write. To see how IPC works in practice, let us consider a simple but common example, a *print spooler*. When a process wants to print a file, it enters the file name in a special *spooler directory*. Another process, the *printer daemon*, periodically checks to see if there are any files to be printed, and if there are, it sends them to printer and removes their names from the directory.

Assume that our spooler directory has a large number of slots, numbered 0, 1, 2, …, each one capable of holding a file name. Also suppose we have two shared variables, *out*, which points to the next file to be printed, and *in*, which points to the next free slot in the directory. These two variables might be kept on a two-word file available to all processes. Think of a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are dull (with the file names to be printed). More or less frequently, processes A and B decide they want to queue a file for printing as illustrated below.



Spooler Directory

The following might happen about the printing requests of two processes. Process A reads *in* and stores the value, 7, in a local variable *next_free_slot.* Just then a clock interrupt occurs and the processor decides that process A has run long enough, so it switches to process B. Process B also reads *in*, and also gets a 7, so 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*, in its local variable 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 get an output. 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** .

The key for preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else, is to find some way to prohibit 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* (some way of making sure that if one process is using a shared variable or file, the other process 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.

6.2 Critical Section Problem

The problem of avoiding race conditions can also be formulated in an abstract way. Part of the time, a process is busy doing internal computations and other things that do not lead to race conditions. However, sometimes a process may be accessing shared memory or files, or doing other critical things that can lead to races. That part of the program where the shared memory is accessed is called the **critical section (CS)**. If we could arrange matters such that no two processes were ever in their critical sections at the same time, we could avoid race conditions.

Although this requirement avoids race conditions, this 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 sections.
2. No assumptions may be made about speeds or the number of processors.
3. No process running outside its CS may block other processes.
4. No process should have to wait forever to enter its CS.

In this section, we will examine various proposals for achieving mutual exclusion, so that while one process is busy updating the shared memory in its CS, no other process will enter its own CS region and cause problem. In our discussions we will consider two process pi (i=0 and i=1) in the form:

```
{common variable declarations and  initializations}

Pi:
{
   while (TRUE) {
      {CS entry code}
      CS( ) ;
      {CS exit code}
      Non-CS( ) ;
   }
}
```

6.2.1 Disabling Interrupts

The simplest solution is to have each process disable all interrupts just after entering its CS and re-enable them just before leaving it. With interrupts disabled, the processor can not switch 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 intervene.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose one of them did it, and never turned them on again. That can be the end of the system. Furthermore, if the system has more than one processor, this method will fail again since the process can disable the interrupts of the processor it is being executed by.

6.2.2 Lock Variables

As a second attempt, let us look for a software solution. Consider having a single, shared lock variable initialized to 0. When a process wants to enter its CS, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the CS. If the lock is already 1, the process just waits until it becomes 0.

```
#define FALSE 0
#define TRUE   1
int lock=FALSE
```

```
PO:
{
   while (TRUE) {
      while (lock) { }; /* wait */
      lock=TRUE;
      CS( ) ;
      lock=FALSE;
      Non-CS( ) ;
   }
}
```

```
P1:
{
   while (TRUE) {
      while (lock) { }; /* wait */
      lock=TRUE;
      CS( ) ;
      lock=FALSE;
      Non-CS( ) ;
   }
}
```

Unfortunately, this idea contains a fatal flaw. Suppose 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 CSs at the same time. This is the situation we saw in our printer spooler example.

Now, it may be thought that we could get around this problem by first reading the lock value, then checking it again just before storing into it However this solution really does not help. The race now occurs if the second process modifies the lock just after the first process has finished its second check.

```
int lock=FALSE

PO:
{
   while (TRUE) {
      lock=TRUE;
      while (lock) { }; /* wait */
      CS( ) ;
      lock=FALSE;
      Non-CS( ) ;
   }
}
```

```
P1:
{
   while (TRUE) {
      lock=TRUE;
      while (lock) { }; /* wait */
        CS( ) ;
      lock=FALSE;
      Non-CS( ) ;
   }
}
```

6.2.3 Strict Alternation

A third approach to the mutual exclusion problem is given below:

```
int turn=0

PO:
{
  while (TRUE) {
     while (turn ! = 0)  { } ; /* wait */
     CS( ) ;
     turn = 1;
     Non-CS( ) ;
  }
}
```

```
P1:
{
  while (TRUE) {
     while (turn != 1) { } ; /* wait */
    CS( ) ;
    turn = 0;
    Non-CS( ) ;
  }
}
```

Here, the integer variable *turn*, initially 0, keeps track of whose turn it is to enter the CS and examine or update the shared memory. Initially, process 0 inspects *turn*, finds it to be 0, and enters its CS. Process 1 also finds it to be 0, and therefore sits in a tight loop continually testing *turn* to see when it becomes 1. Continuously testing a variable waiting for some value to appear is called **busy waiting**. It should usually be avoided, since it wastes processor time.

When process 0 leaves the CS, it sets *turn* to 1, to allow process 1 to enter its CS. Suppose process 1 finishes its CS quickly, so both processes are in their non-CS sections, with *turn* set to 0. Now process 0 executes its whole loop quickly, coming back to its non-CS with *turn* set to 1. At this point, process 0 finishes its non-CS and goes back to the top of its loop. Unfortunately, it is not permitted to enter its CS now, because *turn* is 1 and process 1 is busy with its non-CS. This situation violates condition 3 set out before; process 0 is being blocked by a process not in its CS. Therefore, taking turns is not a good idea when one of the processes is much slower than the other.

6.2.4 Peterson's Solution

By combining the idea of taking turns with the idea of lock variables and warning variables, in 1965, a Dutch mathematician, T. Dekker, was the first one to devise a software solution to

the mutual exclusion problem that does not require strict alternation. In 1981, G.L. Peterson discovered a much simpler way to achieve mutual exclusion, thus rendering Dekker's algorithm obsolete.

Before entering its CS, each process calls CS_*entry* with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *CS_exit* to indicate that it is done and to allow the other process to enter, if it so desires.

Let us see how this solution works. Initially, neither process is in its CS. Now process 0 calls CS_*entry*. It indicates its interest by setting its array element, and sets *turn* to 0. Since process 1 is not interested, CS_*entry* returns immediately. If process 1 now calls CS_*entry*, it will hang there until *interested[0]* goes to *FALSE*, an event that only happens when process 0 calls *exit*.
Now consider the case that both processes call *enter_region* almost simultaneously. Both will store their process number in *turn*. Whicever store is done last is the one that counts; the first one is lost. Suppose process 1 stores last, so *turn* is 1. When both processes come to the while statement, process 0 executes it zero times, and enters its CS. Process 1 loops and does not enter its CS.

```
...                                          /* required header files */
#define FALSE 0
#define TRUE   1
#define N      2                             /* number of processes */

int turn;                                    /* whose turn is it? */
int interested[N]                            /* all elements initialized to FALSE */

void CS_entry (int process)                  /* process: Who is entering (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 )  { };   /* wait */
}

void CS_exit (int process)                   /* process: Who is leaving (0 or 1) ? */
{
        interested[process] = FALSE;         /* indicate departure from CS */
}
```

This method is correct for mutual exclusion but it wastes the processor time. Furthermore, it can have unexpected effects. Consider a system with two processes, H with high priority and L, with low priority. The scheduling rules are such that H is run whenever it is in ready state. At a certain moment, with L in CS, H becomes ready to run. H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its CS, so H loops forever. This situation is referred to as *priority inversion problem*.

6.2.5 Special Hardware Instructions

Followings are special hardware instructions that may be used for the solution of CS problem. These operations are assumed to be atomically executed (in one machine cycle).

Test-and-set

The instruction test-and-set(int lock) returns true if lock is true, otherwise it first sets the value of lock to true and returns false. It can be used for the solution of the CS problem as follows:

```
int  lock=0;     /*global */

PO:                                          P1:
{                                            {
  while (TRUE) {                               while (TRUE) {
     while test-and-set(lock) { } ;               while test-and-set(lock) { };
     CS                                           CS
     lock = FALSE;                                lock = FALSE;
     non-CS;                                      non-CS;
  }                                            }
}                                            }
```

Swap

The instruction swap (int lock, int key) interchanges the content of its parameters. Again, lock is used as a global boolean variable, initialized to false.

```
int  lock=FALSE;     /*global */

PO:                                          P1:
{ int  key                                   { int  key
  while (TRUE) {                               while (TRUE) {
     key=TRUE                                     key=TRUE
     while (KEY) swap (lock,key);                 while (LOCK) swap (lock,key);
        CS                                        CS
     lock = FALSE;                                lock = FALSE;
     non-CS;                                      non-CS;
  }                                            }
}                                            }
```

6.2.6. Semaphores

E. W. Dijkstra suggested using an integer variable for IPC problems. In his proposal, a new variable type, called a **semaphore,** was introduced. Dijkstra proposed having *two atomic operations*, DOWN and UP (P and V in Dijkstra's original paper). The DOWN operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value and

just continues. If the value is 0, the process is put to sleep. Checking the value, changing it, and possibly going to sleep is all done as a single, indivisible action (this is why these operations are called *atomic*.). 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 solving synchronization problems and avoiding race conditions. The UP operation increments the value of the semaphore addressed.

```
typedef int semaphore;   /* semaphores are a special kind of int */
semaphore s=0;

void down(semaphore &s)
{
        while (s <= 0) { }  ;
        s = s – 1;
}

void up(semaphore &s)
{
        s=s+1;
}
```

If two processes try to execute up(s) or down(s) simultaneously, these operations will be executed sequentially in an arbitrary order.

Semaphores can be used for CS problem as follows

```
semaphore mutex = 1;              /* controls access to CS */

 proces_i (void)                  /* n-processes */
 {   while (TRUE) {
        down(mutex) ;             /* CS_entry */
        CS
        up (mutex) ;              /* CS_exit */
        non-CS
     }
  }
```

6.3 Classical IPC Problems

Besides its usage for CS problem, semaphores can also be used for synchronisation of the processes. For example consider two concurrent processes: p1, p2. We require that statement s2 of p2 will be executed after statement s1 of p1. Let p1 and p2 share a common semaphore 'synch', initialized to 0. Semaphores can be used for this synchronisation problem as follows :

```
semaphore synch=0;

void P1(void)
{ ...
   down(synch)
   S1
}
```

```
semaphore synch=0;

void P1(void)
{ ...

   S1
   up(synch)
}
```

In this section we will examine some well-known IPC problems and their solutions using semaphores.

6.3.1 The Bounded Buffer (Producer-Consumer) Problem

Assume there are n slots capable of holding one item. Process producer will produce items to fill slots and process consumer will consume the items in these slots. There is no information on the relative speeds of processes. Devise a protocol which will allow these processes to run concurrently. A common buffer whose elements (slots) will be filled/emptied by the producer/consumer is needed. The consumer should not try to consume items which have not been produced yet (i.e. the consumer can not consume empty slots). The producer should not try to put item into filled slots.

```
# define  N  100                        /* number of slots in the buffer */

typedef int semaphore;                  /* semaphores are a special kind of int */

semaphore mutex = 1;                    /* controls access to CS */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
        int item;
        while (TRUE)                    /* infinite loop */
        {
                produce_item(item)      /* generate something to put into buffer */
                down(empty);            /* decrement empty count */
                down(mutex);            /* enter CS */
                enter_item(item);       /* put new item in buffer */
                up(mutex);              /* leave CS */
                up(full);               /* increment count of full slots */
        }
}

void consumer(void)
{
        int item;
        while (TRUE)                    /* infinite loop */
        {
                down(full);             /* decrement full count */
                down(mutex);            /* enter CS */
                remove_item(item);      /* take item from buffer */
                up(mutex);              /* leave CS */
                up(empty);              /* increment count of empty slots */
                consume_item(item);     /* do something with the item */
        }
}
```

6.3.2 The Readers and Writers Problem

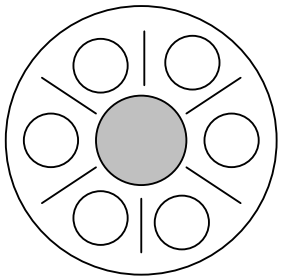
Imagine a big database, such as an airline reservation system, with many competing processes wishing to read and write. It is acceptable to have multiple processes reading the database at the same time, if one process is writing to the database, no other processes may have access to the database, not even readers. Following is a solution for this case.

```
typedef int semaphore;
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1 ;                      /* controls access to db */
int rc = 0 ;                            /* no. of processes reading or writing */

void reader(void)
{
        while (TRUE)
        {
                down(mutex);            /* get exclusive access to rc */
                rc = rc + 1;            /* one reader more now */
                if (rc == 1) down(db);  /* whether this is the first reader */
                up(mutex);              /* release exclusive access to rc */
                read_database();        /* access the data */
                down(mutex);            /* get exclusive access to rc */
                rc = rc – 1;            /* one reader fewer now */
                if (rc == 0) up(db);    /* whether this is the last reader */
                up (mutex);             /* release exclusive access to rc */
                use_data_read();        /* non-CS */
        }
}

void writer(void)
{
        while (TRUE)
        {
                think_up_data();        /* non-CS */
                down (db);              /* get exclusive access */
                write_database();       /* update the database */
                up(db);                 /* release exclusive access */
        }
}
```

It is seen in this solution that the readers have priority over writers. If a writer appears while several readers are in the database, the writer must wait

6.3.3 The Dining Philosophers Problem

There are N philosophers spending their lives thinking and eating in a room. In their round table there is a plate of infinite rice and N chopsticks. From time to time, a philosopher gets hungry. He tries to pick up the two chopsticks that are on his right and his left. A philosopher that picks both chopsticks successfully (one at a time) starts eating. A philosopher may pick one chopstick at a time. When a philosopher finishes eating, he puts down both of his chopsticks to their original position and he starts thinking again. The question is to write a program which does not let any philosopher to die due to hunger (i.e. no deadlocks).



```
#define  N  6                          /* number of philosophers */
semaphore chopstick[N]                 /* a semaphore for each chopstick,
                                          each to be initialized to 1 */
void philosopher(int i)                /* which philosopher (0 to N-1) ? */
{
        while (TRUE)
        {
                think();               /* philosopher is thinking */
                down(chopstick[i]);    /* take left chopstick */
                down(chopstick [(i+1) % N]);   /* take right chopstick */
                eat();                 /* yum-yum, rice */
                up(chopstick[i]);      /* put left chopstick */
                up(chopstick [(i+1) % N]);     /* put left chopstick */
        }
}
```

Unfortunately, this program fails in the situation when all philosophers take their left chopsticks simultaneously. None will able to take their right chopsticks, there will be a deadlock, and all of them will die because of hunger.

We can modify the program so that after taking the left chopstick, the program checks whether the right chopstick is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left chopsticks, seeing that their right chopsticks are not available, putting down their left chopsticks, waiting, picking up their left chopsticks again simultaneously, and so on till death. We have defined this situation in former chapters. This is the situation in which all the programs continue to run indefinitely but fail to make any progress, namely **starvation**.

Now, you may think, “If the philosophers would just wait a random time instead of the same time after failing to acquire the right chopstick.” That is true, but in some application one would prefer a solution that always works and cannot fail due to an unlikely series of random numbers. (Think about safety control in a nuclear plant)

The following program uses an array *state,* to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire chopsticks). A philosopher may move into eating state only if neither neighbour is eating. The neighbors are defined by the macros LEFT and RIGHT.

```
#define N               6                       /* number of philosophers */
#define LEFT            (i-1) % N               /* number i's left neighbor */
#define RIGHT           (i+1) % N               /* number i's right neighbor */
#define THINKING        0                       /* mode of thinking */
#define HUNGRY          1                       /* mode of hunger */
#define EATING          2                       /* mode of eating */

typedef int semaphore;                          /* semaphores are a special kind of int */
int state[N];                                   /* array to keep track of states */
semaphore mutex = 1;                            /* mutual exclusion for CS */
semaphore s[N] ;                                /* one semaphore per philosopher */

void philosopher(int i)                         /* i : Which philosopher (0 to N-1) ? */
{
        while (TRUE)                            /* infinite loop */
        {
                think();                        /* philosopher is thinking */
                take_sticks(i);                 /* acquire two chopsticks or block */
                eat();                          /* yum-yum, rice */
                put_sticks(i);                  /* put both chopsticks back */
        }
}

void take_sticks(int i)                         /* i : Which philosopher (0 to N-1) ? */
{
        down(mutex);                            /* enter CS */
        state[i] = HUNGRY;                      /* record that the philosopher is hungry */
        test(i);                                /* try to acquire 2 chopsticks */
        up(mutex);                              /* leave CS */
        down(s[i]) ;                            /* block if chopsticks were not acquired */
}

void put_sticks(int i)                          /* i : Which philosopher (0 to N-1) ? */
{
        down(mutex);                            /* enter CS */
        state[i] = THINKING;                    /* philosopher has finished eating */
        test(LEFT);                             /* see if the left neighbor can eat now */
        test(RIGHT);                            /* see if the right neighbor can eat now */
        up(mutex);                              /* leave CS */
}

void test(int i)                                /* i : Which philosopher (0 to N-1) ? */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
          {
                state[i] = EATING;
                up(s[i]);
          }
}
```

6.3.4 The Sleeping Barber Problem

The barber shop has one barber, one barber chair, and N chairs for waiting customers, if any, to sit in. If there is no customer at present, the barber sits down in the barber chair and falls asleep. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there is an empty chair) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions.

```
#define CHAIRS 5                    /* number of chairs for waiting customers */

typedef int semaphore;

semaphore customers = 0;            /* number of waiting customers */
semaphore barbers = 0;              /* number of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting not being haircut */

void Barber(void)
{
        while (TRUE)
        {
                down(customers);    /* go to sleep if number of customers is 0 */
                down(mutex);        /* acquire access to 'waiting' */
                waiting = waiting – 1;  /* decrement count of waiting customers */
                up(barbers);        /* one barber is now ready to cut hair */
                up(mutex);          /* release 'waiting' */
                cut_hair();         /* cut hair, non-CS */
        }
}

void customer(void)
{
        down(mutex);                /* enter CS */
        if (waiting < CHAIRS)
        {
                waiting = waiting + 1;  /* increment count of waiting customers */
                up(customers);      /* wake up barber if necessary */
                up(mutex);          /* release access to 'waiting' */
                down(barbers);      /* wait if no free barbers */
                get_haircut();      /* non-CS */
        }
        else
        {
                up(mutex);          /* shop is full, do not wait */
        }
}
```

Our solution uses three semaphores: *customers*, which counts waiting customers (excluding the one being served), *barbers*, the number of idle barbers (0 or 1), and *mutex* for mutual exclusion. The variable *waiting* is essentially a copy of *customers* and it is required since there is no way to read the current value of a semaphore.

QUESTIONS

1. A counting semaphore pair allows the down and up primitives to operate on two counting semaphores simultaneously. It may be useful for getting and releasing two resources in one atomic operation. The down primitive for a counting semaphore pair can be defined as follows:

```
void down(sermaphore s1, semaphore s2)
{
        while (s1<=0)or(s2<=0) do (*nothing*);
         s1:=s1-1;
         s2:=s2-1;
}
```

Show how a counting semaphore pair can be implemented using regular down(s) and up(s) primitives.

2. Using up and down operations on semaphores,

a. Present an incorrect solution to the critical section problem that will cause a deadlock involving only one process.

b. Repeat **a.** for case involving at least two processes.

3. consider the following processes executing concurrently:

```
void P1(void)
{  while (TRUE) {
     st_a
     st_b
     st_c
     st_d
   }
}
```

```
void P2(void)
{  while (TRUE) {
     st_e
     st_f
     st_g
   }
}
```

```
void P3(void)
{  while (TRUE) {
     st_h
     st_i
   }
}
```

Give a solution to synchronize P1, P2 and P3 such that the following order of execution across the statements are is satisfied:

statement a before statement f, statement e before statement h,
statement g before statement c, statement g before statement i.

4. A version of readers/writers problem is defined as follows: 'A reader can enter its critical section only if there is no writer process executing, and there is no writer process waiting.' Device a concurrent solution for the second readers/writers problem. That is, define the shared variables and semaphores needed for your solution, and write the general structure for:

a. a reader process,
b. a writer process,
c. the initialization code.

Write a comment for each statement in your solution.

5. Assume that there are two process P1 and P2 executing on processors Pi and Pj in a distributed system. P1 and P2 are synchronized with a binary semaphore 'flag':

```
void P1(void)
{  while (TRUE) {
     st_a
     up(flag)
     st_b
   }
}
```

```
void P2(void)
{  while (TRUE) {
     st_c
     down(flag)
     st_d
   }
}
```

a. What is the resulting order of execution among statements a, b, c, and d with above code?

Now assume that down(flag) is implemented as [wait until you receive a 'proceed on flag' message from a process executing on any processor], and up(flag) is implemented as a [send a 'proceed on flag' message to all processors].

b. Is there any danger of violating the order of execution of part **a.** with this implementation.

c. If two or more processes use this implementation of down and up primitives in accessing shared data, is there any danger of violating the mutual exclusion condition ?

d. Is there any danger of deadlocks in this implementation?

6. Consider a concurrent system with two processes p1 and p2, and two semaphores s1 and s2. Explain why the following use of semaphores s1 and s2 may create a deadlock.

semaphore s1=1, s2=1

```
void P1(void)
{  down(s1)
   use_device1
   down(s2)
   use_device2
   up(s1)
   up(s2)
}
```

```
void P2(void)
{  down(s2)
   use_device2
   down(s1)
   use_device1
   up(s1)
   up(s2)
}
```

7. Present a correct solution for concurrent bounded buffer problem assuming there are two buffers, buffer A and buffer B available each of size n. Assume you are given procedures which add an item to a buffer, and remove an item from a buffer, given the buffer name explicitly. Give the structure of the consumer, and the producer processes, and the initialization code. Clearly explain the need for every semaphore variable and shared variable you use in your solution.

8. Suppose we have a computer system with n processes sharing 3 identical printers. One process may use only one printer at a time. In order to use a printer, a process should call procedure "request_printer(p)". This procedure allocates any available printer to the calling

process, and returns the allocated printer's id in variable p. If no printers are available, the requesting printer is blocked, and it waits until one becomes available. When a process is finished with printer p, it calls procedure "release_printer(p)".

Write procedures request_printer(p) and release_printer(p) in C, for this system, using semaphores.

9. It is claimed that the following code for producer and consumer processes is a correct solution for the bounded buffer problem :

```
semaphore mutex=1, empty=N, full=0

void producer(void)
{
        int item;
        while (TRUE)
        {
                produce_item(item)
                down(mutex);
                down(empty);
                enter_item(item);
                up(mutex);
                up(full);
        }
}
```

```
void consumer(void)
{
        int item;
        while (TRUE)
        {
                down(mutex);
                down(full);
                remove_item(item);
                up(mutex);
                up(empty);
                consume_item(item);
        }
}
```

Is this solution deadlock-free? If yes, prove your answer by using any deadlock detection algorithm you wish. If no, modify it so that it becomes deadlock-free.

10. A street vendor prepares cheese potatoes according to the following rule:

i. Baked potatoes should be ready;
ii. Grated cheese should be ready;
iii. When there are n customers waiting (n=0,1,..), up to n+1 cheese potatoes may be prepared.

Write concurrent processes that use only semaphores to control concurrent processes

a. Potato_baker, which bakes 4 potatoes at a time and should run until out_of_potato

b. Cheese_grater, grates a single portion cheese at a time and should run until out_of_chese

c. Cheese_potato_vendor, prepares one cheese potato at a time,
d. Customer_arrival

Do not forget to indicate the initial values of the semaphores.

11. In the IsBank, METU, there is a single customer queue, and four bank servers.

Write concurrent processes that use only semaphores to control concurrent processes

a. customer queue
b. bank servers

Do not forget to indicate the initial values of the semaphores.

12. Two processes P1 and P2 are being executed concurrently on a single processor system. The code for P1 and P2 is given as:

{common variable declarations and initializations}

```
void P1(void)
{
   while (TRUE) {
     {CS1 entry code}
     CS1( ) ;
     {CS1 exit code}
     Non-CS( ) ;
   }
}
```

```
void P2(void)
{
   while (TRUE) {
     {CS2 entry code}
     CS2( ) ;
     {CS2 exit code}
     Non-CS( ) ;
   }
}
```

a. Write the CS entry and exit codes for P1 and P2 using down and up operations on semaphores to satisfy the following condition: 'CS1 will be executed exactly once after CS2 is executed exactly once' (i.e. init | CS2 | CS1 | CS2 | CS1 | CS2 | CS1 | ....). Show the initial values for semaphores as well.

b. Repeat part **a.** for the following condition: 'CS1 will be executed exactly once, after CS2 is executed exactly two times.' (i.e. init | CS2 | CS2 | CS1 | CS2 | CS2 | CS1 |...)

13. Five dining philosophers are in the graduation ball of the university. Again the menu is rice to be eaten by chopsticks. However here they spend their time also for dancing in addition to thinking and eating. Thinking and eating are activities performed while sitting. They are sitting around a table having five plates and one chopstick between each plate.
From time to time a philosopher gets hungry. He tries to pick up the two chopsticks that are on his right and on his left. A philosopher that picks both chopsticks successfully, starts eating without releasing his chopsticks. A philosopher may only pick up one chopstick at a time. When a philosopher finishes eating he puts down both of his chopsticks to their original position and he starts thinking again.

There are two ladies in the ballroom who are always willing to dance with the philosophers, and they never let more than four philosophers to be sitting at the same time. They may invite a philosopher to dance only if he is not eating , but he is thinking. A philosopher cannot deny the invitation of a lady if he is thinking.

Write concurrent processes for philosophers and ladies, that use semaphores to control synchronization over critical section code. Discuss deadlock and starvation issues, and propose deadlock-free and nonstarving solutions if possible.

14. a. Find an execution order of the statements of the master and slave processes causing deadlock:

semaphore mutex=1, clean=1, dirty=0;

```
        void master(void)
       {
m1:        down (mutex);
m2:        down (clean);
m3:        drink();
m4:        up(dirty)
m5:        up(mutex)
       }
```

```
        void slave(void)
       {
s1:        down (mutex);
s2:        down (dirty);
s3:        wash();
s4:        up(clean)
s5:        up(mutex)
       }
```

b. How can you overcome this deadlock possibility with no change on the initial values?

15. Our five philosophers in the coffee room have realized that it is not easy to wash dishes. So they have decided to have a servant. Furthermore they bought some more cups. Now they have 3 cups, 1 pot and a servant. All the cups are clean initially and pot is full of coffee. The philosophers either drink coffee or chat. To drink coffee, a philosopher should grab a clean cup, and have to wait the servant to pour some coffee in it. Then he drinks the coffee and puts empty cup on the table. On the other hand, the servant is responsible from washing it if there is any dirty cup, pouring coffee if there is any philosopher has grabbed a cup. The servant is filling the pot if the coffee in the pot have been finished, so assume the coffee in the pot is infinity. Write concurrent procedures for the philosophers and the wash-cup, pour-coffee of the servant by using semaphores such that it should be deadlock free and any process in non-critical section should not cause any other process to wait unneccessarily.