William Stallings Book: Chapter 5

OS – has processes and threads

Multiprogramming: The management of multiple processes within in a uniprocessor system

Distributed processing: The management of multiple processes executing on multiple distributed computer systems.

Fundamental to OS is concurrency…

Concurrency:

- Communication among processes
- Sharing and competing for resources
- Synchronization of the activities of multiple processes

Concurrency arises in three different contexts:

- Multiple applications (processing time shared)
- Structured applications (set of concurrent processes)
- OS structures (set of processes or threads)

Principles of concurrency:

- Single processor multiprogramming (processes are interleaved in time)

- Multiple processor multiprogramming (processes are overlapped)

(Interleaving vs overlapping)

- Sharing of global resources
- It is difficult to manage resources optimally
- Difficult to locate programming errors

Example:

Process 1

```
void echo()
{
   chin = getchar();
   chout = chin;
  putchar(chout);
}
```

Process 2

```
      void echo()
  {
              chin = getchar();
              chout = chin;
               putchar(chout);
}
```

Single processor and single user: User can jump from one program to another; each application uses the same keyboard and monitor

P1 – invokes echo, interrupted after getchar(), character x is in chin variable

P2 – invokes echo, runs to completion, read a character y, and printed y

P1 – resumes execution, chin contains y, now y is printed

First character read by P1 is lost. We display ‘y’ twice.

Problem: chin is a shared variable

Solution: Only one process at a time should be in the above procedure. They should run one after the other to avoid this problem.

It is necessary to protect shared resources or global variables. Single processor or multiprocessors both have same problems….

Processor 1

```
void echo()
{
   chin = getchar();
  chout = chin;  (LOST)
```

Processor 2

```
      void echo()
  {
```

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

```
chout = chin;
putchar(chout);
}
```

Control access to shared variables is needed in both cases……

OS Concerns:

Design and manage issues related to concurrency!

(1) OS must be able to keep track of the various active processes using PCB

(2) Os must allocate and de-allocate various resources for each process

   a. Process time
   b. Memory
   c. Files
   d. I/O devices

(3) The OS must protect the data and physical resources of each process against unintended interface by other processes

(4) The results of a process must be independent of the speed at which the execution is carried out relative to the speed of other concurrent processes

Process Interaction:

(1) Processes unaware of each other; independent processes that are not intended to work together (batch, interactive, mixed, may require same disk, printer; OS must regulate these processes)

(2) Processes indirectly aware of each other (share access to same object; I/O buffer may be shared; exhibit cooperation in sharing common object

(3) Processes directly aware of each other (able to communicate with PID and designed to work jointly on the same activity; exhibit cooperation)

Competition among processes for resources

- I/O devices
- Memory
- Processor time
- Clocks

Three control problems must be defined:

A. Mutual exclusion (ME)
B. Deadlocks
C. Starvation

No exchange of information between the competing processes…

Critical Resource (CR): printer, memory, device

Critical Section (CS): portion of the program that uses critical resources

A. **Mutual Exclusion (ME):** Only one program at a time is allowed in a critical section.

(We can't rely on OS to enforce this restriction as application programmer designed concurrent code!)

B. Deadlock:

P1	P2
(r1)	(r2)

P1 needs r2 and P2 needs r1; each process is waiting for the other resource

C. Starvation:

P1 P2 P3

P1 – r1

P2, P3 are delayed; P1 and P3 may get r1 but P2 is starved

//process

```
const int n = /*no of processes*/ ;
void P(int i)
{
      while (true)
       {
            entercritical(i);
            /*critical section */
            exitcritical(i);
            /* remainder*/
      }
}
```

//main program

```
void main()
{
      parabegin ((P(r1), P(r2), …., P(rn))
}
```

- Cooperation among processes by sharing resources
- Interact with each other without being explicitly aware of them, sheared variables or files
- ME, Deadlock and Starvation need to be addressed

Data may be accessed in two forms: **reading** and **writing**

causes data coherence…

Data Coherence

Two data items “a” and “b” are to be maintained in the relationship **a = b**;

A program that updates one value must update another….

P1 a = a+ 1; (1)

b = b+1; (2)

P2 b = 2 * b; (3)

a = 2 * a; (4)

If each process executes separately; it is consistent.

If P1 and P2 execute concurrently; assume a = 4; b = 4; the steps may run in any order of execution

a = a+ 1; (1) a=5

b = 2*b; (3) b=8

b=b+1; (2) b=9

a = 2*a ; (4) a=10

a != b Inconsistent

Cooperation among processes by communication

- Messages of some sort
- Primitives of sending and receiving may be provided as part of a programming language or kernel
- No ME is required; but deadlock and starvation exists.

Requirements for ME

1. ME must be enforced; only one process at a time is allowed in the CS
2. A process that halts in a non CS must do so without interfering with other processes
3. It must not be possible for a process requiring access to a CS to be delayed indefinitely; no deadlock or starvation
4. When no process is in CS, any process that requests entry to its CS must be permitted to enter
5. No assumptions are made about relative process speeds or number of processors
6. A process remains inside its CS for a finite time delay.

Implementation of ME

- Software Approach (No support from OS or programming languages)
- Use of special purpose machine instructions
- Semaphores

Software Approaches

First Attempt

Assumption: Only one access to a memory location can be made at a time.

Global memory location "**turn**" is reserved for shared variable

Process 0

```
....
.....
while (turn != 0)
    /*do nothing*/;
/*CS*/
turn = 1;
…….
……
```

Process 1

```
.....
……
.....
while (turn != 1)
    /* do nothing*/;
/*CS*/
turn = 0;
…….
```

- Guarantees ME

- Has two problems:
    - Processes must strictly alternate in their use of their CS; pace is dictated by the slower process

    - If one process fails, the other one is permanently blocked; whether in CS or not.

Second Attempt

Need state information about both processes.

flag[0] for P0

flag[1] for P1

(Boolean vector flag;

when one fails, the other can still access CS)

Each process may examine the other's flag, but may not alter it…

When a process P0 wishes to enter CS, it periodically checks P1s flag until that flag has FALSE, indicating that it is not in CS.

```
enum boolean {FALSE=0; TRUE=1};
boolean flag[2] = {FALSE, FALSE};
```

Process 0

```
…….
while (flag[1])
    /*do nothing*/;
flag[0] = TRUE;
/*CS*/
flag[0] = FALSE;
…….
…….
```

Process 1

```
……..
while (flag[0])
    /*do nothing*/;
flag[1] = TRUE:
/*CS*/
flag[1] = FALSE;
…….
…….
```

P0 executes the while flag[1] false

P1 executes the while flag[0] false

BOTH can enter the CS at the same time…..!!!!!!!

If flag[1] = TRUE and P1 fails; then system hangs

If flag[1]=FALSE and P1 fails; then it is ok .

NOT independent of relative process execution speeds….

A process can change its state after the other process has checked it, but before the other process can enter into critical section…

Third Attempt

Process 0	**Process 1**
…….	…….
flag[0] = TRUE;	flag[1] = TRUE;
while (flag[1])	while (flag[0])
/*do nothing*/;	/*do nothing*/;
/*CS*/	/*CS*/
flag[0] = FALSE;	flag[1] = FALSE;
…….	…….
…….	…….

- If both processes set their flags to TRUE at the same time, then they are in a loop for ever
- If a process fails inside a CS, then the other process is blocked
- If a process fails outside the CS, then is not blocked (after reset)

  A process sets its flag without knowing other process's status!!

Fourth Attempt

Process 0

```
…….
flag[0] = TRUE;
while (flag[1])
  {
    flag[0] = FALSE;
    /* delay */;
   flag[0] = TRUE;
 }
/* CS */
flag[0] = FALSE;
……..
……
```

Process 1

```
……..
flag[1] = TRUE;
while (flag[0])
 {
   flag[1] = FALSE;
   /*delay*/;
  flag[1] = TRUE;
   }
/*CS */
flag[1] = FALSE;
……..
……..
```

- P0 sets flag[0] to TRUE
- P1 sets flag[0] to TRUE
- P0 checks flag[1] FALSE
- P1 checks flag[0] FALSE
- P0 sets flag[0] TRUE
- P1 sets flag[1] TRUE

The above sequences could be extended indefinitely.
Neither process could get into CS
It is not a deadlock! It is a livelock!

Any alteration in relative speeds of processes could make one process enter into CS.

A Correct Solution: Decker's Algorithm

Need to observe the state of both processes, which process has the right to insist on entering into CS.

```
boolean flag[2];
int turn;
```

```
void P0()
    {
        while(TRUE)   //main while loop
        {
           flag[0] = TRUE; //turn on your own flag
          while(flag[1])   //test for P1's flag
              {
                   If (turn == 1)  //if P1 is active then
                     {
                        flag[0] = FALSE;  //turn off your own flag
                        while (turn == 1) //wait for P1 to release
                            /*do nothing, stay here*/;
                        flag[0] = TRUE;    //you got the turn  now
                     }
                  /*        CS          */;
                  turn = 1;               //give turn to other guy
                  flag[0] = FALSE;    //turn off your flag
                 /* remainder       */
              }  //end of test for P1
        } //end of main while
     } //end of P0
```

```
void P1()
        {
                while(TRUE)   //main while loop
                {
                   flag[1] = TRUE; //turn on your own flag
                  while(flag[0])   //test for P0's flag
                        {
                             If (turn == 0)  //if P0 is active then
                                 {
                                    flag[1] = FALSE;  //turn off your own flag
                                    while (turn == 0) //wait for P0 to release
                                          /*do nothing, stay here*/;
                                    flag[1] = TRUE;    //you got the turn now
                                 }
                             /*        CS          */;
                            turn = 0;                    //give turn to other guy
                            flag[1] = FALSE;             //turn off your flag
                           /* remainder       */
                        }  //end of test for P0
                } //end of main while
        } //end of P1

//MAIN PROGRAM
void main()
{
  flag[0] = FALSE;
  flag[1] = FALSE;
  turn = 0;
  parabegin (P0, P1);
}
```

A Correct Solution: Peterson's Algorithm

```
boolean flag[2];
int turn;

void P0()
      {
            while(TRUE)   //main while loop
            {
                  flag[0] = TRUE; //turn on your own flag
                                    // so that P1 can't get in
                  turn = 1;           //Give a chance to P1
                  while(flag[1] && turn == 1)

                        //test for P1's flag and his turn

                        {
                        /*do nothing, stay here*/;
                        }
                  /*        CS          */;
                  flag[0] = FALSE;          //turn off your flag
                  /* remainder       */
            } //end of main while
      } //end of P0
```

```
void P1()
    {
        while(TRUE)   //main while loop
        {
            flag[1] = TRUE; //turn on your own flag
                            // so that P0 can't get in
            turn = 0;          //Give a chance to P0
            while(flag[0] && turn == 0)
                //test for P0's flag and his turn
                {
                /*do nothing, stay here*/;
                }
            /*      CS        */;
            flag[1] = FALSE;          //turn off your flag
            /* remainder      */
        } //end of main while
    } //end of P1

//MAIN PROGRAM
void main()
{
  flag[0] = FALSE;
  flag[1] = FALSE;
  turn = 0;
  parabegin (P0, P1);
}
```

Hardware Support

Mutual Exclusion

Interrupt disabling: In a uniprocessor environment concurrent processes can't be overlapped, but can be interleaved…

A process will continue to run until it invokes an OS service or until it is interrupted.

It is sufficient for ME to prevent a process being interrupted…

```
while(TRUE)
{
    /* disable interrupts */
    *   CS   */
    /*enable interrupts */
    /*remainder*/
}
```

Because, CS can't be interrupted, ME is guaranteed…

Special Machine Instructions

Test and Set Instruction

Reading and writing with one instruction in an atomic fashion…..

```
boolean testset (int i)
{
    If (i==0)
        {
            i = 1;
            return TRUE;
        }
    else
        {
            return FALSE;
        }
};
```

```
/*program ME */
const int n = /*number of processes */
int bolt;
void P(int i)
{
        while (TRUE)
                {
                        while (!testset(bolt))
                                /* do nothing, testset was not successful*/;
                        /*   CS        testset was successful */
                        bolt = 0;  /* release the bolt*/
                        /*remainder*/;
                } //end of main while
}//end of P
void main()
{
        Bolt = 0;
        parabegin ((P(1), P(2), …, P(n));
}
```

testset() is atomic, not subject to interrupts…

Properties of the machine instruction approach

Advantages:

- It is applicable to any number of processes on either a single or multiple processors sharing main memory
- It is simple and therefore easy to implement
- It can be used to support multiple CSs; each can be defined by its own variable

Disadvantages:

- Busy waiting is employed; while waiting for a CS, it consumes CPU time
- Starvation is possible, selection of a waiting process is arbitrary
- Deadlock—
    - P1 in CS – low priority – needs CR r1
    - P2 high priority, need CR r1; then P2 can’t complete

Semaphores

1965 Dijstra

Design of OS as a collection of cooperating sequential processes.

A process can stop at a specified place until it has received a specific signal.

For signaling, special variables called Semaphores are used.

1. A semaphore may be initialized to a non-negative value

2. The wait() operation decrements the semaphore value. If the value becomes negative, then the process executing the wait() is blocked.

3. The signal() operation increments the semaphore value. If the value is not positive, then a process is unblocked.

```
struct semaphore {
        int count;
        queuetype queue;
    }
```

```
void wait (semaphore s)
{
        s.count --; //decrement
        if (s.count < 0)
                {
                        place this process in s.queue;
                        block this process;
                }
}
void signal (semaphore s)
{
        s.count ++;   //increment
        if (s.count < = 0)
                {
                        remove a process P from  s.queue;
                        place P on ready queue;
                }
}
```

Queue is FIFO

The process that has been blocked the longest is released from the Queue first; strong semaphore.

A semaphore that does not specify the order in which processes are removed from the Queue is a weak semaphore.

Mutual Exclusion Using Semaphores

n processes

a wait(s) is executed first before entering into CS

```
/*program ME */
const int n= /*no of processes */
semaphore s=1;
```

```
void P(int i)
{
	While (TRUE)
	{
		wait(s);	/* if any one there, then it will wait*/
		/*	CS	*/
		signal(s);
		/* remainder	*/
	}
}


void main()
{
	parabegin((P(1), P(2), …., P(n));
}
```

s = 1

First process will enter CS immediately. Other processes will be blocked setting “s” to negative value.

Any number of processes may attempt to enter into CS, each will result in decrementing “s” and will be blocked.

When a process leaves a CS, “s” is incremented and the blocked process is removed from Queue for execution.

Initializing the semaphore to a specific value ‘n’ will allow ‘n’ number of processes to be in CS concurrently.

s.count >= 0 is the number of processes that can execute wait(s) without suspension. (When s=1, only one process can be in CS)

s.count <0 is the magnitude of no of processes waiting in s.queue. (when s = -2, two processes are waiting)

Producer/Consumer Problem using Semaphores

One or more processes generating data and placing in a buffer. There is a single consumer that is taking data out of the buffer one at a time. (prevent overlap of buffer operations)

Assume infinite buffer…

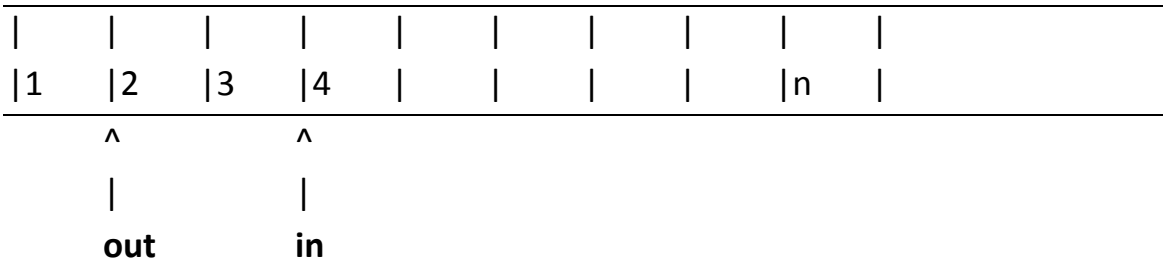
Producer

```
While (TRUE)
    {
    /*produce an item v */;
    b[in] = v;
    in++;
    }
```

Consumer

```
while(TRUE)
    {
    while (in <= out)
            /*do nothing*/;
    w = b[out];
    /*consume item v*/;
    }
```

b



```
/*program, Producer/Consumer */
semaphore n = 0;  /* no of items in the buffer */
semaphore s = 1;  /* communicating semaphore */

void Producer ()
{
        while(TRUE)
        {
                produce();
                wait(s);
                append();
                signal(s);
                signal(n);
        }
}

void Consumer ()
{
        while(TRUE)
        {
                wait(n);
                wait(s);
                take();
                signal(s);
                consume();
        }
}
void main()
{
        parabegin (Producer, Consumer);
}
```

Readers/Writers Problem (Reader's Priority)

1. Any number of readers may simultaneously read a file.

2. Only one writer at a time may write to the file.

3. If a writer is writing to a file, no reader may read it.

4. Once a single reader access a file, then all other readers can retain control to access the file.

   (Readers have priority).

Producer and consumer problem is not same as reader/writer problem. Producer must read Queue pointers to determine where to write the next item and also determine if the buffer is full.

Consumer is not just a reader, it must adjust a Queue pointer to show that it has removed a unit from the buffer…

/*program Readers/Writers*/

int readcount;

semaphore x=1 (readers control); wsem=1 (ME);

```
void Reader()
{
        while(TRUE)
        {
                wait(x);
                readcount++;
                If (readcount == 1)  //first reader
                        wait(wsem);   //writers may be starved
                signal(x);
                READUNIT();
                wait(x);
                readcount--;
                If (readcount == 0) //last reader
                        signal(wsem);
                signal(x);
        } //end of while
} //end of reader
```

```
void Writer()
{
        while (TRUE)
                {
                        //as long as a writer is accessing data, no reader or writer
                        //       can access it; see wait(wsem) in the readers
                        wait(wsem);
                        WRITEUNIT();
                        signal(wsem);
                }//end of while
} //end of writer


void main()
{
        readcount = 0;
        parabegin(Reader, Writer);
}
```