MID-SEMESTER EXAMINATION, 2015 SOLUTIONS

SUBJECT – OPERATING SYSTEM

1. Consider a typical computer system deadlock scenario and discuss all necessary and sufficient conditions for deadlock.

S: In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from starting

Example: DEADLOCK with mutex locks

```
/* Create and initialize mutex locks */

    pthread_mutex_t   first_mutex;

    pthread_mutex_t   second_mutex;

    pthread_mutex_init (&first_mutex, NULL);

    pthread_mutex_init (&second_mutex, NULL);
```

Next, two threads are created — thread_one and thread_two and both these threads have accers to mutex locks, thread_one and thread_two run in functions do_work_one() and do_work_two() respectively.

```
/*thread_one runs in this function */

void *do_work_one (void *param)

{ pthread_mutex_lock (&first_mutex);

  pthread_mutex_lock (&second_mutex);

  /* Do some work */

  pthread_mutex_unlock (&second_mutex);

  pthread_mutex_unlock (&first_mutex);

  pthread_exit (0);

}
```

```
/* thread two runs in this fn */

void *do_work_two (void *param)

{ pthread_mutex_lock (&second_mutex);

  pthread_mutex_lock (&first_mutex);

  /* Do some work */

  pthread_mutex_unlock (&first_mutex);

  pthread_mutex_unlock (&second_mutex);

  pthread_exit (0);

}
```

~~KAMLESH~~ Thread_one attempts to acquire locks in order (1) first_mutex (2) second mutex while thread_two does it in reverse order. Deadlock is possible if thread_one acquires first mutex while thread_two **aequits** second_mutex

Necessary conditions for Deadlock

1. Mutual exclusion → Atleast one resource must be held in a non-sharable mode; that is only one process at a time can use the resource. If another process requests for the resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait → A process must be holding atleast one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No preemption → Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait → A set $\{P_0, P_1, P_2 \ldots P_n\}$ of waiting process must exist such that P_0 is waiting for resource held by P_1, P_1 is waiting for resource held by P_2, ... P_{n-1} is waiting for resource held by P_n and P_n is waiting for resource held by P_0.

Q2. What is a System call? What are the different categories of System calls in an operating System? Discuss a system call implementation example with the help of a program.

Ans: System call is a routine/function (usually written in C/C++) built into a kernel which provides facility to access services provided by the operating System. A System call provides high level interface to the users to use the computer resources without having to know about functioning of resources. All activities related to file handling, process management and memory management are handled by System calls.

The different categories of System calls are as follows:

(i) Process control:
- end, abost e.g. exit()
- load, execute
- create process, terminate process e.g. fork()
- get process attributes, set-process attributes
- wait for time
- wait for signal e.g. wait()
- allocate & free memory

(ii) File management:
- Create file, delete file
- open, close e.g. open()
- read, write, reposition e.g. read()
- get file attributes, set file attributes

(iii) Device management:
- request device, release device
- read, write, reposition
- get device attributes, Set device attributes
- logically attach/detach devices

(iv) Information maintenance:
- get time & date, set time; date
- get System data, Set System data
- get process, file or device attributes
- Set process, file or device attributes

(v) Communications
- Create, delete comm. Connections
- Send, receive messages
- transfer status information
- attach/delete remote devices

(vi) Protection:

System call implementation example:

```
/* C program file to copy contents of file using read and write System calls */

#include <fcntl.h>      /* For O_RDONLY, O_WRONLY, O_CREAT etc. */
#include <sys/stat.h>   /* For S_IRUSR, S_IWUSR, S_IRGRP etc. */
#define BUFSIZE 1024
int main(void)
{
    int fd1, fd2;       /* file descriptors for read & write */
    int n;              /* No of characters returned by read */
    char buf[BUFSIZE];
    fd1 = open("/etc/passwd", O_RDONLY);
    fd2 = open("passwd.bak", O_WRONLY | O_CREAT | O_TRUNC);
    while ((n = read(fd1, buf, BUFSIZE)) > 0)
        write(fd2, buf, n);
    close(fd1);
    close(fd2);
    exit(0);
}
```

A:3

Classical Synchronization Problems

The Producer-Consumer Problem

In this problem, two processes, one called the *producer* and the other called the *consumer*, run concurrently and share a common buffer. The producer generates items that it must pass to the consumer, who is to consume them. The producer passes items to the consumer through the buffer. However, the producer must be certain that it does not deposit an item into the buffer when the buffer is full, and the consumer must not extract an item from an empty buffer. The two processes also must not access the buffer at the same time, for if the consumer tries to extract an item from the slot into which the producer is depositing an item, the consumer might get only part of the item. Any solution to this problem must ensure none of the above three events occur.

A practical example of this problem is electronic mail. The process you use to send the mail must not insert the letter into a full mailbox (otherwise the recipient will never see it); similarly, the recipient must not read a letter from an empty mailbox (or he might obtain something meaningless but that looks like a letter). Also, the sender (producer) must not deposit a letter in the mailbox at the same time the recipient extracts a letter from the mailbox; otherwise, the state of the mailbox will be uncertain.

Because the buffer has a maximum size, this problem is often called the *bounded buffer problem*. A (less common) variant of it is the unbounded buffer problem, which assumes the buffer is infinite. This eliminates the problem of the producer having to worry about the buffer filling up, but the other two concerns must be dealt with.

The Readers-Writers Problem

In this problem, a number of concurrent processes require access to some object (such as a file.) Some processes extract information from the object and are called *readers*; others change or insert information in the object and are called

writers. The Bernstein conditions state that many readers may access the object concurrently, but if a writer is accessing the object, no other processes (readers or writers) may access the object. There are two possible policies for doing this:

1. *First Readers-Writers Problem*. Readers have priority over writers; that is, unless a writer has permission to access the object, any reader requesting access to the object will get it. Note this may result in a writer waiting indefinitely to access the object.
2. *Second Readers-Writers Problem*. Writers have priority over readers; that is, when a writer wishes to access the object, only readers which have already obtained permission to access the object are allowed to complete their access; any readers that request access after the writer has done so must wait until the writer is done. Note this may result in readers waiting indefinitely to access the object.

So there are two concerns: first, enforce the Bernstein conditions among the processes, and secondly, enforcing the appropriate policy of whether the readers or the writers have priority. A typical example of this occurs with databases, when several processes are accessing data; some will want only to read the data, others to change it. The database must implement some mechanism that solves the readers-writers problem.

The Dining Philosophers Problem

In this problem, five philosophers sit around a circular table eating spaghetti and thinking. In front of each philosopher is a plate and to the left of each plate is a fork (so there are five forks, one to the right and one to the left of each philosopher's plate; see the figure). When a philosopher wishes to eat, he picks up the forks to the right and to the left of his plate. When done, he puts both forks back on the table. The problem is to ensure that no philosopher will be allowed to starve because he cannot ever pick up both forks.

There are two issues here: first, deadlock (where each philosopher picks up one fork so none can get the second) must never occur; and second, no set of philosophers should be able to act to prevent another philosopher from ever eating. A solution must prevent both.

8:4 Define process and thread. How a multithreaded model is better as compared to multiprocess model having single thread. What are kernel and user threads? Explain the mapping between them.

Soln: (a) Process — It is an executing instance of a program

Threads — Basic unit to which the operating system allocates processor time. A thread is a path of execution within a process.

Threads within the same process share the same address space, whereas different processes do not. This allows threads to write and read to the same data structures and variables, and also facilitates communication between threads.

Threads are considered lightweight because they use far less resources than processes.

A process can have multiple threads.

(b) Benefits of Multi-threading

(i) Parallelism — In machines with multiple processors, threads provide an efficient way to achieve true parallelism. As each thread receives its own processor individually and is schedulable entity, multiple threads may run on multiple processors at the same time, improving a system's throughput.

(ii) Programming abstraction — Dividing up the work and assigning each division to ~~each~~ a unit of execution is a natural approach to many problems.

(iii) Blocking I/O — Without threads, blocking I/O halts the whole process. This negatively affects throughput and latency. In a multithreaded process individual threads may block, waiting for I/O, while other threads make progress.

(c)

User and kernel Level Threads

1. Kernel-Level Threads:

In this method, the kernel knows about and manages the threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.

Advantages:

Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads. Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.
Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

2. User-Level Threads

Kernel-Level threads make concurrency much cheaper than process because, much less state to allocate and initialize. However, for fine-grained concurrency, kernel-level threads still suffer from too much overhead. Thread operations still require system calls. Ideally, we require thread operations to be as fast as a procedure call. Kernel-Level threads have to be general to support the needs of all programmers, languages, runtimes, etc. For such fine grained concurrency we need still "cheaper" threads. To make threads cheap and fast, they need to be implemented at user level. User-Level threads are managed entirely by the run-time system (user-level library).The kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.User-Level threads are small and fast, each thread is represented by a PC,register,stack, and small thread control block. Creating a new thread, switiching between threads, and synchronizing threads are done via procedure call. i.e no kernel involvement. User-Level threads are hundred times faster than Kernel-Level threads.

Advantages:

The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads.
User-level threads does not require modification to operating systems.

Simple Representation: Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
Simple Management: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without
intervention ofthe kernel.
Fast and Efficient: Thread switching is not much more expensive than a procedure call.

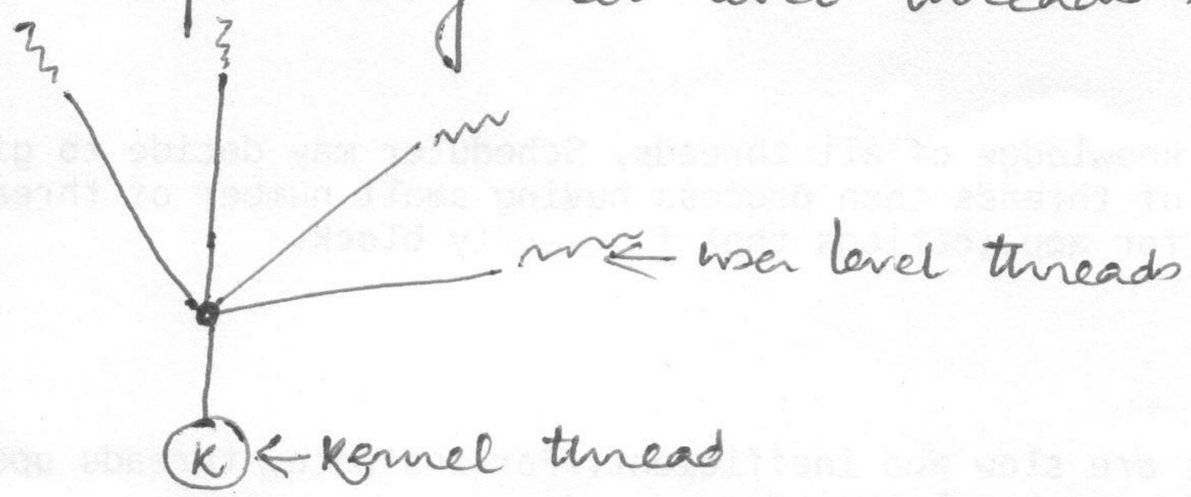
Disadvantages:

User-Level threads are not a perfect solution as with everything else, they are a trade off. Since, User-Level threads are invisible to the OS they are not well integrated with the OS. As a result, Os can make poor decisions like scheduling a process with idle threads, blocking a process whose thread initiated an I/O even though the process has other threads that can run and unscheduling a process with a thread holding a lock. Solving this requires communication between between kernel and user-level thread manager.There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespect of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

(d) There exists a relationship between user threads and kernel threads. There are three ways to establish this relationship

(i) Many-to-One model

This model maps many user-level threads to one kernel thread



Thread management is done by thread library in user space & thus is efficient. but a blocking call by one thread blocks the entire process. Since only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

(ii) One-One model.

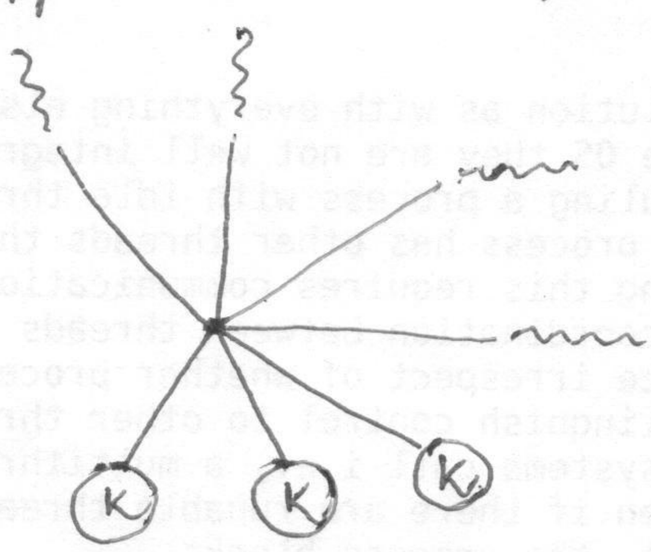
Maps each user thread to a kernel thread.

Advantage: Another thread runs when a thread makes blocking System call.

Disadvantage: Creating a user thread requires creating the corresponding kernel thread.

(iii) Many-to-many model

Multiplexes many user level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to a particular application or a particular machine.



In this model developers can create as many user threads as necessary and the corresponding kernel threads can run in parallel on a multiprocessor.

5/a. Grep.

This command is used to search for a particular pattern from a file or from standard input and display those lines on the standard output. Grep stands for "global search for regular expressions"

Syntax: grep [options] PATTERN [FILE...]

options available with grep command.

-v displays only those lines which do not match the line specified

-n displays matched lines with line numbers

-c displays only the count of lines which match the pattern specified

-i displays matched pattern ignoring case distinction

b. PIPE.

A pipe is a mechanism which takes the output of a command as its input for the next command

ex: $ who | wc -l

c. ps.

Displays information about a selection of the active processes

Syntax: ps [options]

Examples: ps -e : To see every process on the system

ps aux : To see every process on the system using BSD synt

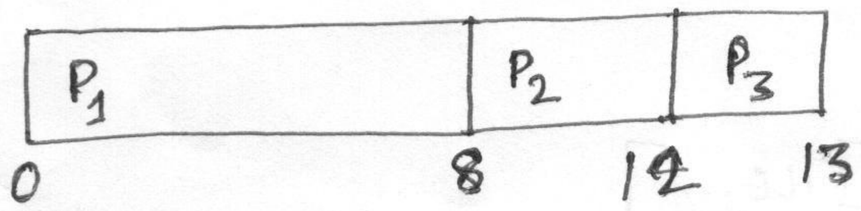
ps -ejH : To print a process tree

```
# A shell program to perform addition/subtraction/multiplication/division using CASE statement


exit_function()
{
   clear
   exit
}


while true
do

   echo -n "Enter your choice : "
   read choice

case $choice in
     1) clear
        echo  "Enter two numbers for Addition : "
        echo -n "Number1: "
        read num1
        echo -n "Number2: "
        read num2
        echo "$num1 + $num2 = `expr $num1 + $num2`"
        exit_function;;
     2) clear
        echo "Enter two numbers for Subtraction : "
        echo -n "Number1: "
        read num1
        echo -n "Number2: "
        read num2
        echo "$num1 - $num2 = `expr $num1 + $num2`"
        exit_function;;

     3) clear
        echo "Enter two numbers for Multiplication : "
        echo -n "Number1: "
        read num1
        echo -n "Number2: "
        read num2
        echo "$num1 * $num2 = `echo "$num1*$num2"|bc`"
        exit_function;;
      4)  clear
        echo "Enter two numbers for Division : "
        echo -n "Number1: "
        read num1
        echo -n "Number2: "
        read num2

        echo "$num1 / $num2 = $div"
        exit_function;;
   esac

done
```

	AT	BT	CT	TAT	WT
P_1	0	8	8	8	0
P_2	0.4	4	12	11.6	7.6
P_3	1	1	13	12	11

Avg = 18.6/3 = (6.2)

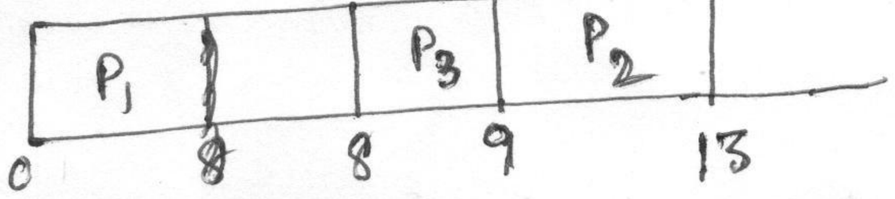
FCFS:



SJF (without preemption)

	AT	BT	CT	TAT	WT
P_1	0	8	8	8	0
P_2	0.4	4	13	12.6	8.6
P_3	1	1	9	8	7

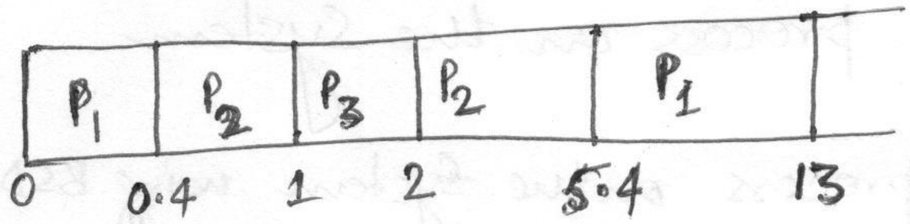
Avg = 15.6/3 = (5.2)



SJF (with preemption)

	AT	BT	CT	TAT	WT
P_1	0	8	13	13	5
P_2	0.4	4	5.4	5	1
P_3	1	1	2	1	0

Avg = 6/3 = (2) ← MINIMUM



Round Robin.

TQ = 1.

	AT	BT	CT	TAT	WT
P_1	0	~~8 7 6 5~~ 4	13	13	5
P_2	0.4	~~4 3 2 1~~ 0	9	8.6	4.6
P_3	1	~~1~~ 0	3	2	1

Avg = 10.6/3 = (3.5)

Queue:

~~P_1 P_2 P_3 P_1 P_2 P_1 P_2 P_1 P_2 P_1~~