Title : Study and implementation of time, sleep and clock management.
Aim : To demonstrate system call for time and clock.

Objective: This lab describes what are the process scheduling, time and clock management in Unix.

Process Scheduling and Time

On a time sharing system, the kernel allocates CPU to a process for a period of time called the time slice or time quantum. After the time quantum expires, it preempts the process and schedules another one. The scheduler in UNIX uses relative time of execution as a parameter to determine which process to schedule next. Every process has a priority associated with it. Priority is also a parameter in deciding which process to schedule next. The kernel recalculates the priority of the running process when it comes to user mode from kernel mode, and it periodically re-adjusts the priority of every "ready-to-run" process in user mode.

The hardware clock interrupts the CPU at a fixed, hardware dependent rate. Each occurrence of the clock interrupt is called a *clock tick*.

Process Scheduling

The scheduler on the UNIX system belongs to the general class of operating system schedulers knows as *round robin with multilevel feedback*. That means, when kernel schedules a process and the time quantum expires, it preempts the process and adds it to one of the several priority queues.

The algorithm *schedule_process* is given below:

```
/*  Algorithm: schedule_process
 *  Input: none
 *  Output: none
 */

{
        while (no process picked to execute)
        {
                for (every process on run queue)
                        pick highest priority process that is loaded in memory;
                if (no process eligible to execute)
                        idle the machine;
                        // interrupt takes machine out of idle state
        }
        remove chosen process from run queue;
        switch context to that of chosen process, resume its execution;
}
```

This algorithm is executed at the conclusion of a context switch. It selects the highest priority process from the states "ready to run, loaded in memory" and "preempted". If several processes have the same priority, it schedules the one which is "ready to run" for a long time.

System Calls for Time

There are several time-related system calls, *stime*, *time*, *times*, and *alarm*. The first two deal with global system time, and the latter two deal with time for individual processes.

stime allows the suepruser to set a global kernel variable to a value that gives the current time:

`stime(pvalue);`

where `pvalue` points to a long integer that gives the time as measured in seconds from midnight before (00:00:00) January 1, 1970, GMT. The clock interrupt handler increments the kernel variable once a second.

time retrieves the time as set by *stime*.

`time(tloc);`

where `tloc` points to a location in the user process for the return value. *time* returns this value from the system call, too.

times retrieves the cumulative times that the calling process spent executing in user mode and kernel mode and the cumulative times that all zombie children had executed in user mode and kernel mode.

```
struct tms *tbuffer;
times(tbuffer);
```

where the structure `tms` contains the retrieved times, and is defined by:

```
struct tms {
   // time_t is the data structure for time
   time_t tims_utime;      // user time of process
   time_t tm_stime;          // kernel time of process
   time_t tms_cutime;      // user time of children
   time_t tms_cstime;      // kernel time of children
}
```

times returns the elapsed time "from an arbitrary point in the past", usually the time of system boot.

The following program forks 10 children and uses *times* to get the time related statistics:

```
#include <sys/types.h>
#include <sys/times.h>
extern long times();

main()
{
   int i;
   // tms is data structure containing the 4 time elements
   struct tms pb1, pb2;
   long pt1, pt2;

   pt1 = times(&pb1);
   for (i = 0; i < 10; i++)
      if (fork() == 0)
```

```
            child(i);

    for (i = 0; i < 10; i++)
        wait((int *) 0);
    pt2 = times(&pb2);
    printf("parent real %u user %u sys %u csys %u", pt2 - pt1,
            pb2.tms_utime - pb1.tms_utime, pb2.tms_stime - pb1.tms_stime,  pb2.tms_cutime
- pb1.tms_cutime, pb2.tms_cstime - pb1.tms_cstime);
}

child(n)
    int n;
{
    int i;
    struct tms cb1, cb2;
    long t1, t2;

    t1 = times(&cb1);
    for (i = 0; i < 10000; i++)
        ;
    t2 = times(&cb2);
    printf("child %d: real %u user %u sys %u", n, t2 - t1,
            cb2.tms_utime - cb1.tms_utime, cb2.tms_stime - cb1.tms_stime);
    exit();
}
```

One would naively expect the parent **child user** and **child system** times to equal the respective sums of the child processes' **user** and **system** times, and the parent **real time** to equal the sum of the child processes' **real time**. However, child times do not include times spent in the **fork** and **exit** system calls, and all times can be distorted by time spent handling interrupts or doing context switches.

User processes can schedule alarm signals using the **alarm** system call.

Clock

The functions of the clock interrupt handler are to:

* restart the clock
* schedule invocation of internal kernel functions based on internal timers
* provide execution profiling capability for the kernel and for user processes
* gather system and process accounting statistics,
* keep track of time
* send alarm signals to processes on request
* periodically wake up the swapper process
* control process scheduling

Some operations are done every clock interrupt, whereas others are done after several clock ticks. The clock handler runs with processor execution level set high. The algorithm **clock** is given below:

```
/*  Algorithm: clock
 *  Input: none
 *  Output: none
```

```
 */

{
   restart clock;      // so that it will interrupt again
   if (callout table not empty)
   {
      adjust callout times;
      schedule callout function if time elapsed;
   }
   if (kernel profiling on)
      note program counter at time of interrupt;
   if (user profiling on)
      note program counter at time of interrupt;
   gather system statistics;
   gather statistics per process;
   adjust measure of process CPU utilization;
   if (1 second or more since last here and interrupt not in critical region of code)
   {
      for (all processes in the system)
      {
         adjust alarm time if active;
         adjust measure of CPU utilization;
         if (process to execute in user mode)
            adjust process priority;
      }
      wakeup swapper process if necessary;
   }
}
```