CHAPTER 8 PROCESS SCHEDULING AND TIME

When an operating system supports multiprocessing (multitasking), it allocates the CPU to a process for specific period of time. This time is called as "***Time Slice***" or "***Time Quantum***". When this time for a process finishes it preempts (pre-emptying or stopping) this process and starts (or schedule) another process to execute. When the turn of stopped arrives, operating system reschedules it to execute.

On UNIX system, "***the relative time of execution***" is used as a parameter to the scheduling function. This function decides which process to schedule next when one's time expires.

Every active process has a specific "***scheduling priority***" and operating system makes context switch to that process which has highest priority at that time among all other active process currently in the memory.

Here 2 things must be kept in mind....

(1) The kernel always re-calculates the priority of a running process when running process returns from the kernel mode to user mode.

(2) The kernel priority re-adjusts the priority of every process, which is in "***ready to run***" state of the user mode.

The word "***periodically***" in above point tells us that kernel already has a time set, which does various tasks when the "***given period***" expires. This time concerns with the "***hardware clock***". The major thing to remember about this clock is that, "***it interrupts the CPU at a fixed but hardware dependent rate. This rate is typically set between 50-100 times/second.***" This interrupt is called as "***clock interrupt***" and each of its occurrence is called as "***clock tick***".

Process Scheduling

On UNIX as the schedule is known as "***Round robin with multilevel feedback***". This means that,

(a) Kernel allocates CPU to a process for a fixed time quantum.
(b) When this time quantum finishes, it pre-empts that process.
(c) The kernel puts this pre-empted process into priority queues.
(d) Then kernel makes a context switch to some another process whose "***scheduling priority***" is highest right now.
(e) When it returns to pre-empted process, it restores its context and proceeds the execution where it has been suspended.
(f) All these things are done after every context switch.

The schedule algorithm is as follows...

```
01 : algorithm : schedule – process
02 : input : nothing
03 : {
04 :            while(there is no process to pick to execute)
05 :            {
06 :                    for(every process on "run queue")
07 :                            pick-up the highest priority process present in memory;
08 :                    if(there is no process eligible to execute)
09 :                            idle the machine;
       /* Later on practically clock interrupt will take the machine out of this idle state */
10 :            }
```

```
11 :        remove chosen process from "run queue";
12 :        switch the context to that chosen process i.e. resume its execution;
13 : }
14 : output : nothing.
```

To understand the algorithm, first we must know that this algorithm is executed after every context switch.

So here we assume that one context switch is just concluded and kernel is executing this algorithm.

Kernel first chooses correct process to execute, which is in one of following 2 states only...

(a) The process, which is "***ready to run and loaded in memory***" state.

(b) The process, which is in "***preempted***" state.

This choice is important because there is no use to choose such a process, which is currently not loaded in memory. Because process else where than memory can never run. That process can only run which is already in memory or which is just "***swapped in***" in the memory.

While making choice of correct process to run there is yet another issue of "***race condition***". Means if many processes have same highest priority (i.e. tie), then kernel chooses that process which is "***ready to run, loaded in memory, for the longest time***". The longest time is decided on the basis of "***round robin scheduling policy***".

If there are no processes right now eligible to execute, CPU idles until next periodic clock interrupt (which will occur in one clock tick). When clock interrupt occurs, kernel first deals with the interrupt and then restarts the try for scheduling a process to run.

After making a choice, kernel removes chosen process from priority queue and makes context switch to that chosen process and starts its execution.

Scheduling Parameters

Each process table entry of a process contains a field concerned with process scheduling priority.

Priority of a process in user mode, is mathematically a function concerned with 2 entries...

(a) The process's current CPU usage (the process with which we concern)

(b) The process, which are getting lower priority because they have recently used the CPU. (i.e. their CPU usages)

The range of process priorities can be divided into 2 classes:

(a) User priorities (i.e. user mode priorities)

(b) Kernel priorities (i.e. kernel mode priorities)

Each above class has several priority values, but in turn each priority has a unique queue of processes logically concerned with that priority value.

When a process switches from kernel mode to user mode and if it has a user level priority, then kernel preempts this process. Also remember that kernel can preempts that process which gets kernel priority in sleep algorithm.

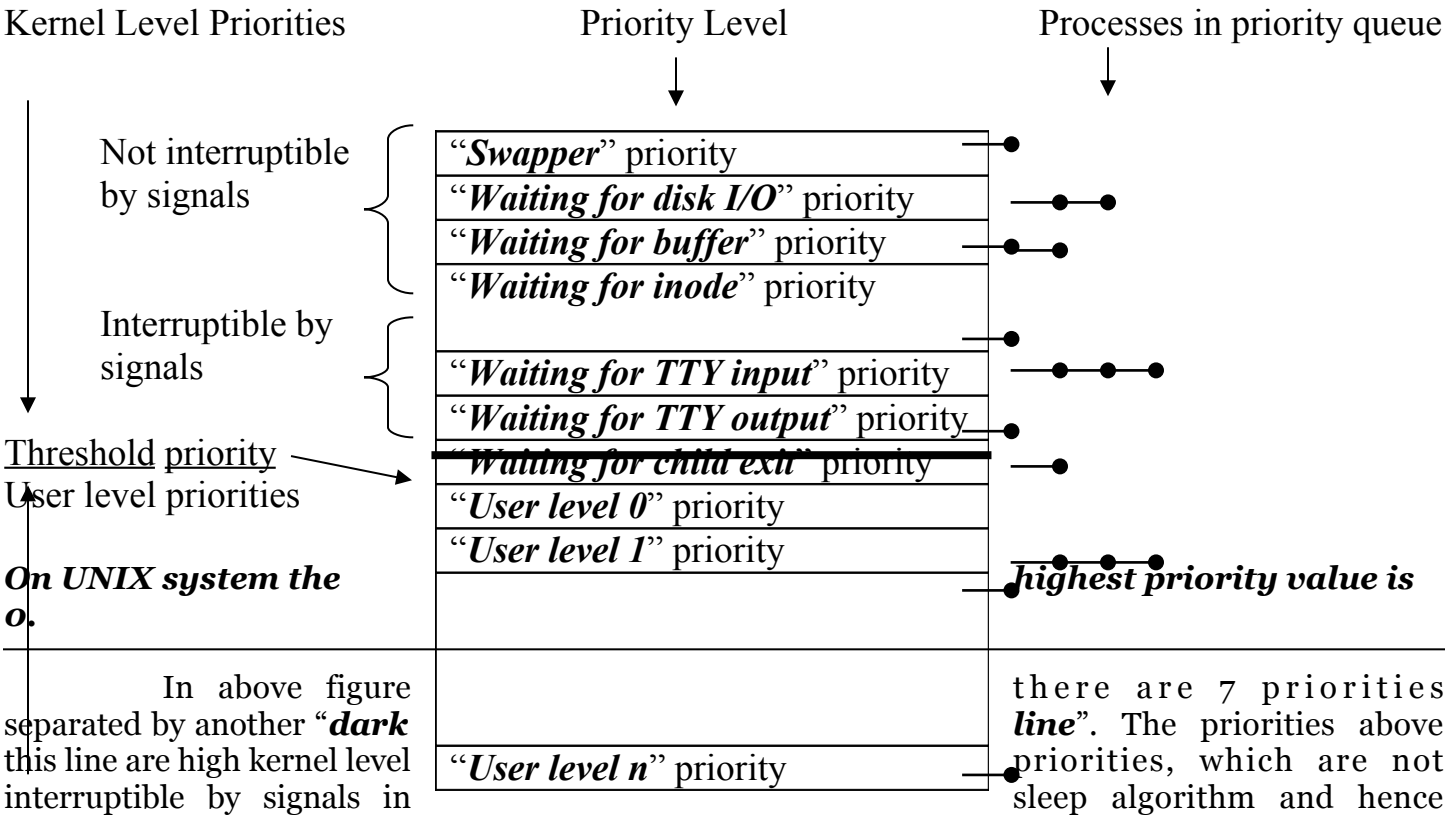
To distinguish user level priority and kernel level priority, there is a boundary priority called as ***threshold priority. User level priorities are below this threshold priority while kernel level priorities are above this threshold priority.***

Kernel level priorities are further subdivided into 2 categories:

(a) Low kernel level priority: - Process with this kernel level priority wakes-up in sleep algorithm on receipt of a signal.

(b) High kernel level priority: - Processes with this kernel level priority do not wake-up in sleep algorithm on receipt of a signal. Instead they continue their sleep algorithm and sleep.

Next figure shows "***user level priorities***" and "***kernel level priorities***", separate by "***dark lined***" threshold priority. Here the threshold priority is the priority between the "***user level o***" priority (which is user level priority) and the "***waiting for child exit***" priority (which is a kernel level priority). So by this figure we can say that the priorities called as "***swapper***" priority, "***waiting for disk I/O***" priority, "***waiting for buffer***" priority, "***waiting for inode***" priority, "***waiting for TTY input***" priority, "***waiting for TTY output***" priority and the last "***waiting for child exit***" priorities are kernel level priorities.



In above figure there are 7 priorities separated by another "***dark line***". The priorities above this line are high kernel level priorities, which are not interruptible by signals in sleep algorithm and hence continue to sleep. While priorities below this line are low kernel level priorities, which can be interrupted on occurrence of a signal in sleep algorithm and hence they wake-up.

Note that, kernel level priorities are in decreasing order. Means "***swapper***" priority is higher kernel level priority while "***waiting for child exit***" priority is the lowest kernel level priority.

The priorities below the threshold priority are "***user level priorities***" which are named as "***user level o***", "***user level 1***" up to "***user level n***" priorities.

Note that, though "***user level o***" has a value denoted by "***o***" and "***n***" will be something greater than 0, on priority level basis, the "***user level o***" priority is the highest user level while the "***user level n***" priority is the lowest user level priority. That is why the arrow denoted is indicator of increasing order user level priorities.

Now look at the right hand side of the figure. The black circles are indicator of processes concerned with that specific priority. As stated before, the two classes "***kernel level priorities***" and "***user level priorities***" can have several priority values (in this figure there are 7 kernel level priority values and n user level priority values) and each of this value has its unique queue of processes. Thus the chained black circles indicate the process queue, unique for that particular priority value.

Owing to this figure...

- "***Swapper***" priority has only one process in its queue.
- "***Waiting for disk I/O***" priority has 3 processes in its queue.
- "***Waiting for TTY output***" priority does not have any process in its queue, while
- "***Waiting for TTY input***" kernel level priority and the

- "***User level 1***" user level priority have 4 processes in their respective queues.

Where these priority values come from? It is the kernel, which calculates these priority values for processes. But note that kernel does this in 3 particular situations…

(1) Kernel assigns a priority value to a process when the process is about to go to sleep. This priority value is not any arbitrary value but a fixed value, which has relationship with the reason of sleeping. Thus priority does not much concern with the running characteristics of process but does much concern with the "***cause of sleep***" of process. Means, the processes sleep in low-level algorithm, like buffer cache algorithms or process address space manipulation algorithms (i.e. "***region***" algorithm) are much prone to undergo deadlocks, race conditions and system crash. Hence such processes are assigned by a high level priority by kernel while they are about to go to sleep. On other hand the processes, which sleep in comparatively high level algorithms, such as file system algorithms or process control algorithms, are less prone to cause system bottlenecks and hence they are assigned by comparatively low priorities by kernel while they are about to go to sleep.

Let us take one example, suppose one process say "**A**" is slept and waiting for completion of a disk I/O while other process, say "**B**" is slept and waiting for a buffer to get free. Now kernel will assign higher to **A** than **B**, because…

- As process A is waiting for completion of disk I/O, means it already has a buffer. When it wakes up after completion of disk I/O, it will free the buffer and all other processes will wake up which were slept for this buffer to get free. Means completion of one task of one process has a chance to wake one or many processes.
- On the other hand, process B is waiting for a free buffer, means some other process already holds the buffer. If the other process is like **A** means waiting for completion of I/O, then it too is in sleep. So when I/O of other process completes, both B and that other process are going to wakeup and rush for the same buffer, because both were slept on same "***sleep address***". Now suppose B is given a higher priority then that of other process, then B will sleep again after wakeup, because other process's I/O may yet not be completed. That is why B should be given less priority than that other process.

So, now we know, when kernel assigns priority to a process. This does not mean that this value is going to remain same all the time.

Rather kernel adjusts priority of a process when that process returns from kernel mode to use mode. This is very important. Because when the process was in kernel mode, it might be in sleep. Means in sleep it was having a kernel level priority. When it returns to user mode, its past kernel level priority must be changed to some user level priority as now on wards the process is going to run in user mode. Kernel does this because, as this process just returned from the kernel mode, it had already used valuable kernel resources in that mode. So to give chance to other processes, kernel gives penalty to this process by lowering its priority from kernel level priority to user level priority.

(2) Not only the kernel, but the clock interrupt handler also adjusts priorities of all processes in user mode at every 1-second interval on system V. This step forces the kernel to re-enter in scheduling algorithm for that process so as to prevent monopolization of CPU by that process.

Clock And Priority

During the total time quantum of a process, the clock may interrupt that process for several times. During such every clock interrupt, the clock handler itself increments a field in the process table slot of that process. This field is concerned with the recent CPU usage of that process.

Besides that activity clock handler also adjust the recent CPU usage of every process (obviously including above one) according to the decay function.

This decay function is…

Decay (CPU) = CPU / 2 **or** } **for system V**
CPU = decay (CPU) = CPU / 2 }

While doing this (means while adjusting recent CPU usage of every process), the clock handler also re-calculates priority of every process (obviously per second) which is in "***preempted but ready to run***" state only.

Priority = (recent CPU usage / 2) + base level user priority where "***base level user priority***" is the threshold priority between kernel mode priority and user mode priority as shown in previous figure.

Again note that, numerically low value is for high priority and numerically high value is for low priority.

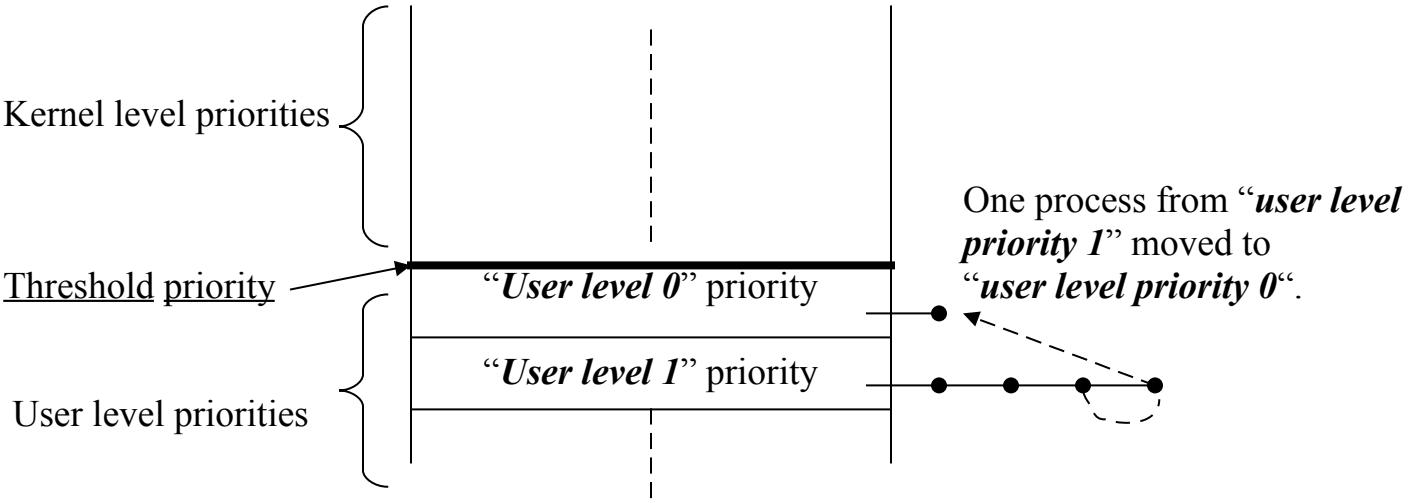
When we say that a process is in "***preempted but ready to run***" state, it is implied that process is returning from kernel mode to user (either after system call or after interrupt) mode, means this process had already used kernel resources. Thus their kernel usage time i.e. recent CPU usage is going to be high. Obviously recent CPU / 2 is also high and addition of recent CPU/2 + base level user priority is also high. So priority of such a process in "***preempted but ready to run***" state is comparatively high.

Now to reach the level of "***base level user priority***", recent CPU usage/2 division should be nearer to zero, means recent CPU usage should be nearer to zero.

That is why priority of "***preempted but ready to run***" state processes is recalculated, to see whether the process is jumping from one priority to other.

Thus processes change their queue when their priority changes from one value to other value in user level priorities.

Compare following figure with the previous figure...



Movement of A process in user priority queues

In figure on page (278) "***user level priority 1***" value has its unique queue of 4 processes, while "***user level priority 0***" has no process in its queue.

Now suppose recalculation of priority, gives higher priority to a process in "***user level1***" priority. Due to this now on wards, this process will not stay in the queue to "***user priority 1***", but will move to the queue of "***user level 0***" priority's queue. That is why, after such removal now "***user level 1***" priority queue will have 3 processes (which was having 4 before) and the "***use level 0***" priority queue will have 1 process (which was not having any process before).

But story remains true for "***user level priorities***" only means..

(a) Kernel never changes priority of a process when process is in kernel mode, and...

(b) Kernel never allows a process having "***user level priority***" right now, to cross the threshold line priority and acquire a "***kernel level priority***" unless the process makes a system call and goes to sleep.

(3) Kernel (like clock interrupt handler), also recalculate the priority of all active processes once a second. But this time (i.e. once a second) may vary slightly. Because before or at "***one second***" interval, if clock handler arrives and at that time if kernel is busy in doing some "***critical section***" code, then kernel does not recalculate priorities of active processes though "***one second***" interval is passed. Because to do this kernel has to divert from critical section, go to process table to search every active process, and then recalculate the priority. This lot of work keeps in the critical region for too long time. To avoid this kernel just remembers that it had missed one opportunity of priority recalculation and thus completes it before next "***one second***" interval.

But here you must keep in mind about the current processor execution level. When kernel was in "***critical section***", all interrupts, except the "***clock***" were blocked, that is why clock handler can arrive in the picture at that moment. For the next "***one second***" interval, when kernel will try to recalculate priorities, this processor execution level must be sufficiently low, so that kernel can be able to move to priority recalculation task.

*This periodic re-calculation of process priority, gives assurance of "**round robin scheduling policy**" to the processes, which are now in user mode.*

About the hard coded "*one second*" interval

The time interval "***1 second***" used by both kernel and clock interrupt handler, is hard coded on UNIX system V. The benefit is that, kernel requires less context switches as it can concentrate on the current task for that "***whole second***". (Note: The 1 second time, though sounds very minute for day to day human, it is very…very big time for computer activities). The drawback is that, kernel has to wait for completion of this interval to recalculate priorities, though it seems sometimes possible to do recalculation before 1-second interval.

On the other hand, some other operating system use different scheduling policy, in which they dynamically can change this time interval between 0 to 1 second, to allow kernel to do priority recalculation before completion of 1 second interval. This is useful when system overload is less. Otherwise, means if system load is high, the default static 1-second interval is used. Means waiting for "***1 second***" is not must "***every time***".

The benefit of this dynamic decision-making is that, kernel gives quick response to process as they get quickly (comparatively) scheduled. The drawback is that, context switches become extra due to rapid process scheduling one after the other and thus kernel has to perform more overheads.

About "*Data Entry*" processes

Some programs are "***Data entry***" programs, where system has to wait on the user's action of data entry. Obviously much time goes idle for such processes. Consequently such processes high "***Idle time: CPU usage ratio***" and thus they get high priority values (numerically low) step by step as they become "***ready to run***".

An example of process scheduling: There are 3 processes A, B, And C.
Assumptions:

(1) All these 3 processes are created simultaneously.
(2) All these 3 processes have initial priority 60.
(3) The highest user level priority is 60.
(4) The clock interrupts the system 60 times/sec.
(5) All these 3 processes will not make any system call.
(6) At completion of 1-second kernel will recalculate CPU usage and priority and will make a context switch.
(7) Besides these 3 processes there is no "***ready to run***" process.
(8) The formula for calculating CPU usage is….

CPU = decay (CPU) = CPU /2.

(9) The formula for calculating process priority...

priority = (CPU/2) + 60 where 60 is base user level priority as in assumption (3).

	Process A			Process B			Process C	
	Time	Priority	CPU usage	Priority	CPU usage		Priority	CPU usage
0 sec		60	0	60	0		60	0
			1					
			¦					
			60					
1 sec		75	30	60	0		60	0
						1		
					¦			
					60			
2 sec		67	15	75	30		60	0
								1
								¦
								60
3 sec		63	7	67	15		75	30
			8					
			¦					
			67					
4 sec		76	33	63	7		67	15
					8			
					¦			
					67			
5 sec		68	16	76	33		63	7

Clock interrupt is going to increment (i.e. ++) the CPU usage count.

Column of process A

Now assume that process A is the first process to run within its given time quantum. Suppose it runs for 1 second. According to our assumption the clock will interrupt the system for 60

times in this one second. At the 0 second mark, i.e. when process begins to run clock will interrupt 60 times and thus will increment CPU usage field 60 times, to 60 (i.e. 0, 1, 2, 3...to 60). At one-second mark when CPU usage is recalculated, it is CPU/2 i.e. 60/2 means 30. So at one second mark the priority will be 30/2 + 60 i.e. 75.

At one-second mark, kernel does context switch from A to B. Obviously there is no incrementation of CPU usage count by clock interrupt during this whole gap of 1-second (i.e. 1st second to 2nd second) When 2nd second quantum finishes, the CPU usage is taken the same as that of 1 second mark i.e. 30 (because there is no activity of A) and recalculation of CPU usage gives 30/2 = 15. So at 2-second mark, the priority of inactive process A is 15/2 + 60 = 67.

At three-second mark, kernel does context switch from B to C. Obviously process A is still inactive. Thus there is no incrementation of its CPU usage count by clock interrupt during this whole gap of one second. (i.e. from 2nd second to 3rd second). When 3rd second quantum finishes, the CPU usage is taken the same as that of 2-second mark, i.e. 15. (Because still A is not scheduled) and recalculation of CPU usage gives 15/2 = 7 (integer division only). So at three second mark the priority of inactive (still) process A is 7/2 + 60 = 63.

At four-second mark, kernel again does context switch from C to A. Now the idle process A gets scheduled and thus clock interrupt starts incrementing its CPU usage count field from previous value. The previous value (i.e. the value at 3 second mark) is 7 so incrementation of it for 60 times in this gap of whole second will yield 67 (i.e. 6..7..8..67). Thus at 4 second mark, the CPU usage is recalculated as 67/2 = 33 and priority becomes 33/2+60 = 76. This happens within the one-second gap of 3rd second to 4th second. When this 1-second finishes kernel does context switch from A to B and whole story repeats. So during the gap of 4th second to 5th second, process A remains inactive and thus there is no incrementation of its CPU usage count. Hence at 5-second mark, the previous CPU usage count value i.e. 33 will be taken as it is and CPU usage is recalculated as 33/2 = 16 and priority becomes 16/2 + 60 = 68.

Column of process B

The logic is same as above. Just there are differences in the sequence of scheduling.

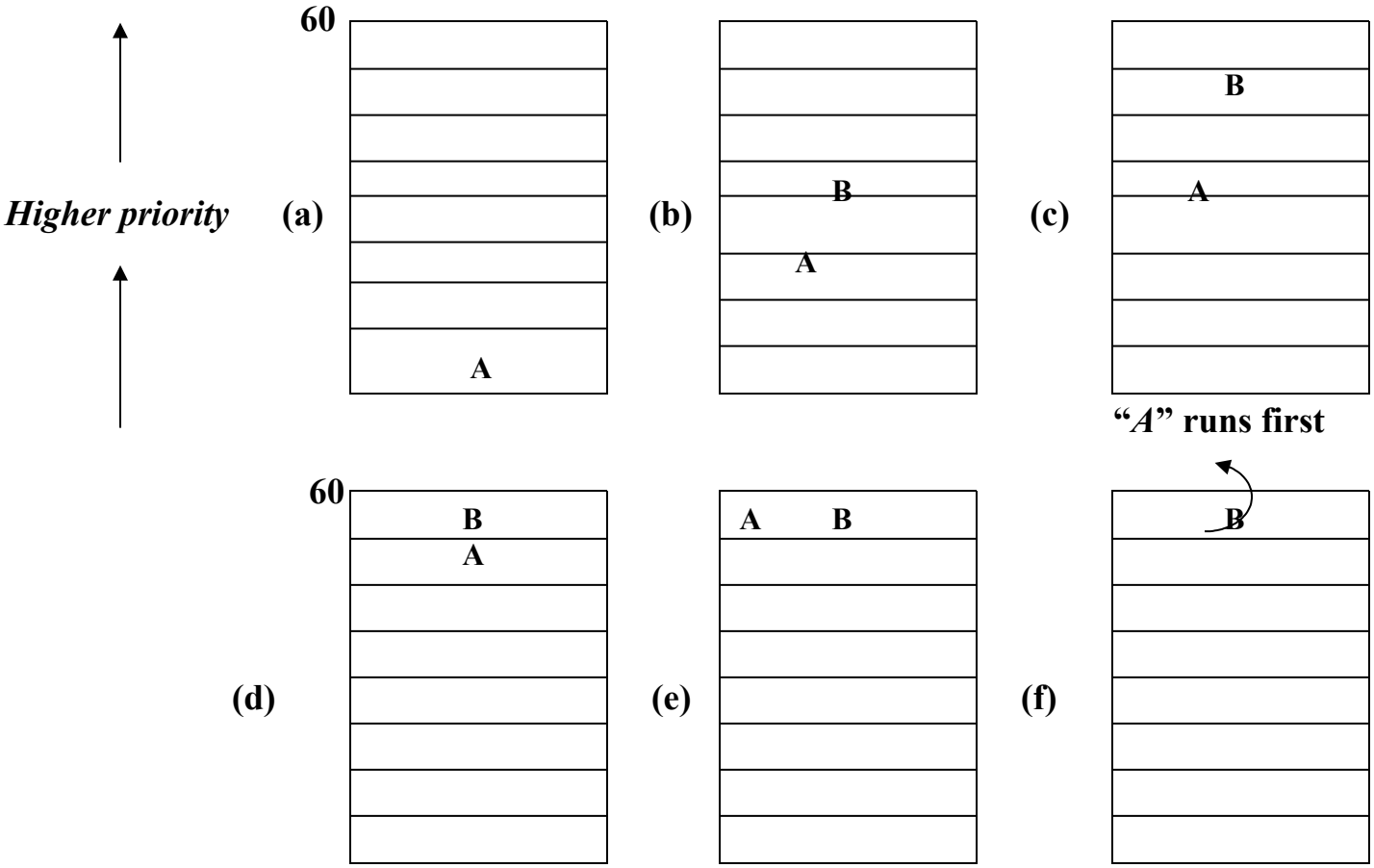
At 0th second mark B has CPU usage count 0 and process B is having priority 60. As process A is first getting scheduled to run, there is no activity of B from 0th second mark to 1st second mark. So at 1st second mark, its CPU usage count remains same as 0 and thus priority also remains same i.e. 60. At 1st second mark kernel makes context switch from A to B and thus B process's CPU usage count starts incrementing due to clock interrupt from 0 to 60. So when this whole second finishes then at 2nd second mark, the recalculated CPU usage of process B is 60/2 = 30 and priority is 30/2 + 60 = 75. At 2nd second mark, kernel does context switch from B to C and B becomes idle. So for the gap of 2nd second to 3rd second, there is no incrementation of CPU usage count and hence at 3rd second mark the count remains the same as before i.e. 30 and recalculated CPU usage thus becomes 30/2 = 15 at 3rd second mark. Obviously at this point priority of B is 3rd second mark. Obviously at this point priority of B is 15/2 + 60 = 67. At 3rd second mark kernel does context switch from C to A and hence still B remains idle for this whole second gap. Thus at 4th second mark its CPU usage count is taken same as its previous value 15 and recalculated CPU usage becomes 15/2 = 7 and the recalculated priority becomes 7/2 + 60 = 63. At 4th second mark, kernel does context switch from A to B. Now idle B becomes active and its CPU usage starts getting incremented from 7 to 67 (i.e. 60 times) by clock interrupt. So after completion of this whole second i.e. at 5th second mark, the CPU usage count is 67, recalculated CPU usage is 67/2 = 33 and recalculated priority is 33/2 + 60 = 76.

Column of process C

As usual, at 0th second mark, the priority of process C is 60 and its CPU usage count is 0. As process A is scheduled at 0th second mark, there is no change in CPU usage count and priority of

process B at 1st second mark. Means they will remain same 0 to 60 respectively. At 1st second mark, kernel does context switch from A to B. So C still remain idle and thus after this one second at 2nd second mark its CPU usage count and priority will remain same i.e. 0 to 60 respectively. At 2nd second mark, kernel does context switch from B to C and thus now C becomes active and its CPU usage count starts getting incremented from 0 to 60 (i.e. 60 times). So after this one second i.e. 3rd second mark, the CPU usage count is 60, recalculated CPU usage becomes 60/2 = 30, and priority becomes 30/2 + 60 = 75. At 3rd second mark kernel does context switch from C to A so after one second, i.e. 4th second mark there is no change in CPU usage count and thus remains same as 30. But when CPU usage is recalculated it becomes 30/2 = 15 and thus priority becomes 15/2 + 60 = 67. At 4th second mark kernel does context switch from A to B, thus C remains idle. No change will occur in CPU usage count and priority for this one-second gap and when 5th second mark it reached, recalculated CPU usage count becomes 15/2 = 7 and priority becomes 7/2 + 60 = 63.

The pattern shown in these 3 columns goes on repeating for processes A, B, & C over time.



Now consider above figure. It shows "***Round Robin***" scheduling and its relationship with the priority of process.

Suppose there are 2 processes A and B, and A was already running and over time received many time quanta for it. Now it is returning from kernel mode to user mode and thus kernel can preempts it (see process transition diagram in ***6th*** chapter of the notes).

Thus suppose kernel preempts A and keeps it in "***ready to run***" state (state 7 of state transition diagram i.e. "***preempted***" = similar to "***ready to run***"). As up till now process A received many time quanta (as per our assumption), its priority will be very low and hence shown at bottom row in diagram (a).

Now suppose process B enters in the picture and is in "***ready to run***" state. Also assume that its priority is higher than that of process A. Thus in diagram (b), after some "***time elapse***", A is advanced but B is on its top due to its high priority.

Supposing that there are other running processes in the system and hence kernel may be busy in dealing with them and thus may not schedule either of A and B. According to our first example, though processes not scheduled right now, but scheduled previously at some time, their CPU usage and priorities are recalculated at every context switch (even though context switch is not concerned with either A or B). Thus by this recalculation mechanism A and B go on marching towards higher priority, as shown in diagram (c) and (d). Here B may reach at top priority first as it was originally had higher priority than A. But kernel will not schedule B to run (why? Answer is coming soon) and thus A also reaches the top priority after some time. Now both A and B are in the row of top priority as shown in diagram (e).

Here we know both A and B in the state "***ready to run***" and both are at high priority and hence one must be scheduled to run.

In this tie situation, though B originally had higher priority than A and though it reached the top level first, it is not scheduled to run. But actually A is scheduled to run and B remains in the top row un-scheduled as shown in diagram (f).

*This is because; A was in "**ready to run**" state for longer time than B. So this is the "**tie-breaker**" rule for processes with equal priority.*

Recall from chapter (6) that, there can be a context switch when

(a) Process goes to sleep with which kernel is right now dealing.
(b) A process exits, and
(c) There may be context switch when a process returns from kernel mode to user mode.

Out of these cases consider the third case means "***kernel can do context switch from current process to some other process when current process is returning from kernel mode to user mode.***" Here a question arises which other process will kernel choose to do context switch? As we know in the previous situation the current process is preempted. But this will happen only when kernel finds that there is some other process having higher priority than the current process. If there exists such a process then only kernel preempts current process, does context switch to other (i.e. having higher priority than current) process and schedules this other process to run.

This raises a new question; in which circumstances such "***other***" process with higher priority than current process will arrive? This may happen in 2 situations.

(a) If kernel awaked such a process which has higher priority than the current running process.
(b) If clock handler (as its regular behavior) change i.e. recalculate and change priorities of all "***ready to run*** " processes and one appear in picture which incidentally has higher priority than current process.

Now why in case (a), kernel cannot continue with current process? Why its must to make the context switch? Answer is that, current process is returning to user mode, means hereafter it will run in user mode. But as the kernel awakens a new process with higher priority and as this newly awakened process is right now in kernel mode, kernel can not allow "***user mode***" process to run when there already exists another "***kernel mode***" high priority process.

In case (b), clock handler realizes that current process has used considerable part of its time quantum (as it is returning to user mode, it is clear that it already spent considerable time in kernel mode) and recalculation of all "***ready to run***" process priorities co-incidentally shows that there is some another process with high priority than the current process and yet not used its time quantum. So to give fair chance to this newly "***high priority***" gained process, kernel preempts current process and makes context switch to this "***high priority***" process.

Controlling process priorities on program level

Programmatically, a program (or say process) can change its scheduling priority. But note that, this is not "***fine tuning***" of priority. Rather this is crude tuning, as programmer cannot know scheduling algorithm flow dynamically.

Programmatic change in process priority can be done by using nice() system call whose syntax is..

nice(value);

where the passed value is added to the equation of priority as shown on the pages (280) and (283). The actual formula or equation of priority is..

priority = (CPU usage / constant) + base priority + nice value.

In the equations on page (280) and (283), the constant is assumed as 2 and nice value is assumed as 0, and base priority was assumed as 60, hence it was mentioned there as ***CPU usage/2 + 60.***

Actually when nice() system call is used, the value passed as parameter is kept in the process table slot of this process..

It is assumed that, programmer will call nice() system call to lower the priority of this process so as to give other processes a chance to get scheduled than this process. This "***nice***" approach of the programmer, thinking more about other users and system performance, gave opportunity to library designers to give name to this system call as "***nice()***".

Important points about nice() system call

(1) Only super user or super user privilege can give such a nice value, which will increase the priority of a process.
(2) Only super user or super user privilege can give such a nice value, which is below a system decided threshold priority.
(3) The process table slot of a process has a field called as nice value with its default value. So when nice() system call is used, the value passed as parameter, is used to increment (if value is +) or decrement(if value is -) the nice value in process table slot by the amount of supplied parameter value.
(4) It is clear from (1) and (2) that ordinary user (or programmer) with neither super user status nor super user privilege, can only lower the process priority by nice() system call. (But not below particular threshold priority).
(5) Process, at its beginning, inherits the default nice value of its parent during fork() system call.
(6) nice() system call works only for current process. Means a process cannot reset or change priority of any other process in any circumstances. This is applicable to super user also. Means admin or super user also cannot change or reset priorities of another processes, even those other processes are consuming too much time. If he/she has to do it so, there is only one way to kill such processes.

Fair share scheduler

We know that in UNIX system, the users can be grouped i.e. some users can be included in one group and according to that group file or process permissions can be manipulated (recall: UGO, where "***G***" stands for "***group***" class of user).

If we consider a "***computer center environment***", there are many users and some of them may form a group and then demand half of the total CPU time of machine for their related process, then the "***scheduling algorithm***", that we saw up till now, can not fulfill this need. Because the "***scheduler algorithm***" we saw, cannot differentiate between classes of users.

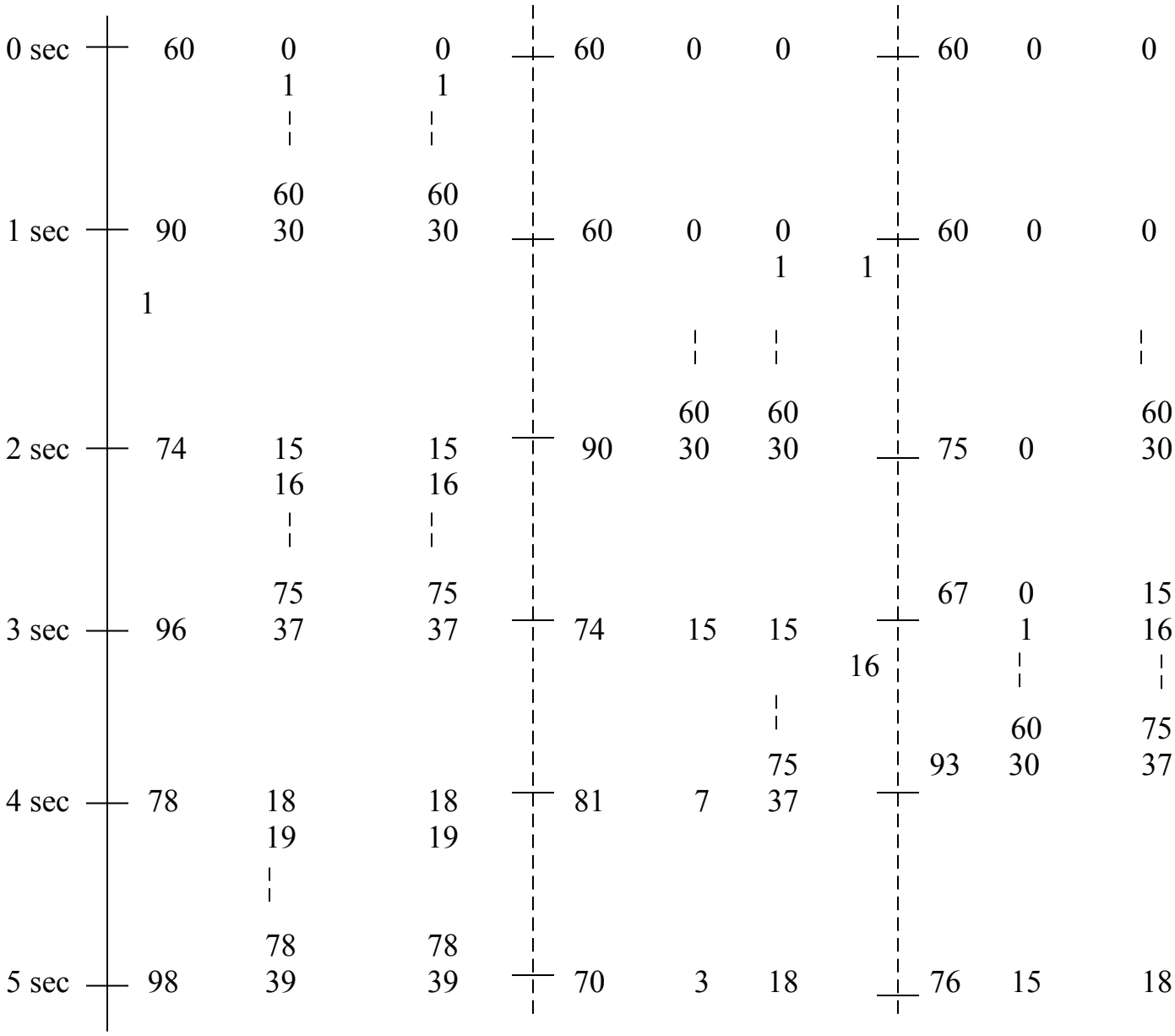
Owing to this need, "***AT&T Bell Laboratories Indian Hill Computer Center***" implemented a scheme called as "***Fair Share Scheduler***". The principle of this scheme is to divide the user community into a set of fair share groups, such that, the members of each group belong to the constraints of regular process scheduler relative to other processes in that same group. But it is important to note that; system will allocate CPU time proportionally to each group irrespective of number of process in that group. For example, suppose there is a 4 fair share group P, Q, R, S and P contains one process, Q contains 2, R contains 3 and S contains 4 processes. Supposing that there are only these 10 processes (i.e. 1 + 2 + 3 + 4 = 10) in the system 100% of CPU time will be divided as 10% to each process. This division does not concern with groups by any way. Now if we further assume that all these 10 processes are continuously running (i.e. they do not exit themselves), because they all have infinite loop in their code, then the CPU time of all processes in a single group should be equal over time as they all are in infinite loop.

Now if we think on group basis, we will find that as group P contains only one process, it will get double CPU time than group Q (as group Q has two processes), also it will get triple CPU time than R (as group R contains three processes) and also will get quadruple CPU time than S (as group S contains 4 processes).

To implement this scheme, a new term called as "***Fair Share Group Priority***" is added into the original equation of priority. So now "***U area***" of each process has a new field which points to "***Fair Share Group CPU Usage***" field and in turn this field will be visible to all processes in that group only. Simplifying this, we can say that pointer to "***Fair Share Group Usage***" field will be kept locally to every process in its U area but the field itself is visible i.e. sharable to all processes in that group. Now clock handler has additional task to do. On each tick, it will increment the running process's "***CPU usage field***" (according to the diagram on page (283) as well as will increment "***Fair Share Group CPU Usage***" field also. Also there will be recalculation of "***Fair Share Group CPU Usage Count***" after every second as we do for our regular "***CPU Usage Count***". Obviously after every second, when priority is recalculated, the new term "***Fair Share Group Priority***" will be added to the priority, which will be according to the "***Fair Share Group CPU Usage***".

Obviously when processes in a group receive more CPU time, then their "***Fair Share Group CPU Usage***" will be more and hence their priority will also be numerically higher and thus actual priority of the processes in that group will be lower. (Again recall that numerically high value means low priority).

	Process A		**Process B**		**Process C**	
Time	Priority	CPU Usage	Group CPU Usage	Priority	CPU Usage	Group CPU Usage	Priority	CPU Usage	Group CPU Usage



The example that we are going to consider now is actually an extension to the example we saw on page (283). Logic is same as that example. But you will see 2 major differences. One is in the value of priority on each second mark, which are quite different than old values and other is the addition of new field "***Group CPU Usage***" and its progression.

For this example assume that process A belongs to one group say P and processes B and C belong to other group say Q. Again assume that kernel is going to schedule process A i.e. group P and when time of group Q will occur, out of B and C, it will schedule B first.

When process A gets scheduled as like its CPU usage count, its "***Group CPU usage count***" will increment 60 times in that second and on 1 sec mark recalculation of group CPU usage i.e. 60/2 it will be 30. So priority will be CPU usage/2 + Group CPU Usage/2 + base priority means 30/2 + 30/2 + 60 = 90.

At this time as B and C of group Q are idle, their priority remains 60 (numerically low than A's 90) and thus both will have high priorities. According to out assumption, now B gets scheduled. Hence clock handler will obviously increment both CPU usage and Group CPU usage fields of it. The new thing is that, though process C of this group is going to be idle when kernel deals with B, and thus though its CPU usage field is not incremented, its Group CPU Usage field will be incremented as B And C belong to same group Q. This is very important. Hence at 2 sec mark, when

recalculations occur, CPU usage field and group CPU usage field of B will be 60/2 and 60/2 hence 30 and 30 and thus its priority will be 90. But for process C as there is no increment in its CPU usage field, it will remain Q, but as there is increment in its group CPU usage field, on recalculation its group CPU usage field will be 60/2 i.e. 30 and hence its priority will be 0/2 + 30/2 + base priority 60 i.e. 0 + 15 + 60 = 75. So at 2nd second mark process B will have priority 90 and process c will have priority 75.

According to our previous example of page (283), now we will expect that after B, kernel will schedule process C. But no! Here kernel will schedule process A. How and why? For this we must first consider what happens to process A within this gap of one second from 1 sec. mark to 2 sec mark. Obviously as kernel is now dealing with B, A is idle. So at 2nd sec mark when recalculation is done, its CPU usage count is same as before means 30 and its Group CPU usage count is also same as before i.e. 30. Thus on recalculation both these counts will be 30/2 and 30/2 means 15 and 15. So the priority of process A will be 15/2 + 15/2 + 60 i.e. 74.

Now if we compare priorities of 3 processes i.e. A's 74, B's 90 and C's 75, then we come to know that A is at the highest priority (because it is numerically lowest) and hence process A gets scheduled.

This applies to all further intervals and kernel schedules the processes in the order: (A, B), (A, C), (A, B) and so on.

Real time processing

By definition, real time processing is the capability of system to give immediate response to specific external events and according to that event, schedule a particular process to run within a specific time limit after occurrence of that specific event.

Here the term "***specific time limit***" does not mean the "***time quantum***" of a process, but it actually means a predefined "***time limit***" within which the kernel must schedule and run particular process after occurrence of particular external event.

For example, there is a computer system situated in a hospital and watching continuously on "***critical status***" of all intensive care unit patients. When an event of "***change in critical status***" of a patient below threshold level, the computer system should acknowledge this external event and within a specific time (say 1 sec) should schedule and run such a process which will start ringing all "***alert bells***" in the hospital for immediate access to that specific health critical patient.

Processes such as text editors are not real time processes. Because though response for events in text editors are quick, they are not quick enough as required in real time processing.

The scheduler algorithms described up till now (i.e. the regular scheduler algorithm and the fair share scheduler algorithm) were not designed for real time environment but are designed for "***time sharing***" environment. This is because it is not guaranteed that kernel will schedule and run a particular process within a fixed time limit after occurrence of particular event.

Recall from the 1st chapter, where we told that "***though Unix is preemptive, its kernel is actually non preemptive***". This is another hurdle in real time processing. Because as non-preemptive kernel can not schedule and run real time process in user mode if it already running other kernel mode process. To allow happening this, major changes has to be made for real time user mode process.

Currently, to achieve real time processing the programmer must insert real time process into the kernel manually. But this is not a true solution because real time processes has to patch in kernel hard coded. The true solution is that, such processes can be allowed exist dynamically at any time and when they run, they will tell the kernel about the real time requirements and then kernel will take appropriate real time actions.

It is very important to note that no Unix system, currently available, is capable of this dynamic real time process patching. In other words Unix is not RTOS (real time operating system) but from initial design Unix variant can be newly design to be RTOS.

System call for time

The important system calls, those deal with time are .....

1) stime()
2) time()
3) times()
4) alarm()

These 2 deal with global system time.

These 2 deal with time for "***individual processes***".

1) **stime()** : - This allows the super user to set a global kernel variable to a value that gives the current time. The syntax is...

stime(pValue);

Where *pValue* is pointer to a long integer that gives the time measured in seconds form the midnight 00:00:00 (h:m:s) of 1st Jan , 1970 GMT (**G**reenwich **M**ean **T**ime).

2) **time()**: - This system call returns/retrieves the time set by stime(). The syntax is ...

time(ptlocation);

where ptlocation is a pointer to a location in the user process in which the time value is filled by the system. Not only this but time () system call returns this time value as its return value, too. The shell command like "***date***" uses this system call internally to determine the current time for end user.

3) **times()**: - This system call retrieves the cumulative (i.e. additive) times which is spent by the calling process in user mode and kernel mode. Besides this, it also retrieves cumulative times, which is spent by zombie children in user mode and kernel mode. The syntax is....

times(pTBuffer); /* struct tms *ptbuffer */

Where ptbuffer is a pointer to tms structure. The tms structure is defined as follows....

```
Struct tms /* time_t is the data structure for time */
{
        time_t tms_utime ; /* time of a process in user mode */
        time_t tms_stime ; /* time of a process in kernel mode */
        time_t tms_cutime ; /* time of a children in user mode */
        time_t tms_cstime ; /* time of a children in kernel mode */
};
```

Here an important point to remember that the times retrieved/returned by times () system call is the time started from system boot. We will consider one example (code for demonstration)

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

void main(void)
{
        int I;
        struct tms  tms1 , tms2;
        long t1 , t2;
        /* Code*/
```

```
        t1 = times(&tms1);
        for(I=0; I<10; I++)
        {
                if(fork() == 0)
                        child(i);
        }
        for(I=0; I< 10; I++)
                wait((int *) 0);
        t2 = times(&tms2);
        printf("Parent real %u : user %u  : sys  %u cuser %u : csys %u \n" , t2 – t1 ,
                cuser %u : csys %u \n" , t2 – t1 ,
                tms2.tms – utime – tms1.tms – utime ,
                tms2.tms – stime – tms1.tms – stime ,
                tms2.tms – cutime – tms1.tms.cutime,
                tms2.tms – ctime – tms1.tms – ctime);
}

void child(int n)
{
        int I;
        struct tms ctms1 , ctms2;
        long ct1 , ct2;
        /* Code*/

        ct1 = times(&ctms1);
        for(I=0;I<10000;I++); /* null loop*/
        ct2 = times(&ctms2);
        printf("Child %d -> real %u : user %u : sys %u \n" , ct2 – ct1 , ctms2.tms – utime
                – ctms1.tms.utime , ctms2.tms- stime – ctms1.tms- stime);
        exit();
}
```

In above program, first initial times are stored and 10 child processes are created and process waits until exit of all children. Finally again the new time say final time is obtained and in the printf() these times i.e. initial time, final time and their differences with respect to its own and its child's time are calculated for both user and kernel mode.

In the child creating function (which is called ten times by the parent) 1st initial time is stored, then child looks for 10000 times (via a null loop) and again final time is calculated. Child also prints (as child itself is an independent process, it has its own times) its real, user mode and kernel mode times and then exits.

Now the point is that, we may expect that the time given by "***child time***" fields (i.e. cutime & cstime) in parent process should be equal to the respective times given by printf in child's code (utime & stime). But this is not the real story. As child executes after the call to fork (), there is no consideration of time spent by fork () system call. Also when child start executing and parent waits, there is context switch from parent to child and when child exits, there is again context switch from child to parent. times() does not consider this time spent in various context switches and interrupt handling in between.

We also may expect that some of real times of 10 children should be equal to the real time of parent as parent does not have any other task than spawning 10 children and waiting for their exits. But here again it is found that this quality not there and the reason is same as above.

Alarm

User process can use alarm() system call to send alarm signals. Consider the following code...

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/signal.h>

void main(int argc, char * argv[])
{
        extern unsigned alarm(int)
        extern void wakeup(void);
        struct stat statuffer;
        time_t time;

        /* code */

        if(argc !=2)
        {
                printf("only one Argument\n");
                exit();
         }
         time = (time_t)0;
        for( ; ; ) /* infinite loop */
        {
                /* find out file access time */
                if(stat (argv[1],&statbuffer) == -1)
                {
                        printf("file %s does not exist \n ",argv[1]);
                        exit();
                }
                 if(time != statbuffer.st_atime) /* file is accessed */
                {
                        printf("file %s is accessed \n",argv[1]);
                        time = statbuffer.st_atime;
                }
                        signal(SIGALARM, wakeup); /* reset*/
                        alarm(60);
                        pause(); /* sleep until signal occurs */
        }
}

void wakeup(void)
{
        /* no code*/
}
```

This program is executed by the passing a file name as its command line argument. The time variable is initialized to o. Then an infinite loop is started. In this loop, inode of this file is retrieved in a buffer by using stat () system call. Then the initial value of time is checked against the access time in

file's inode buffer (i.e. statbuffer). If both the times are equal, then file is not access & if both times are unequal then the file is accessed.

If the file is accessed, then time variable is reset to new file access time in statbuffer so that when it will be access again, then next access time will be compared with previous access time (which is now in time variable) to check the resent access. Beside this program allows catching of SIGALARM signal and executes wakeup () as its signal catches function. This is by signal() system call. Then it sends alarm signal to itself every after a minute (i.e. 60 seconds) and sleeps until signal occurs. All above task are inside this infinite loop. So this program will never terminate until some external interrupt measures, like kill.

Now the reason of demonstrating this code is that, the pause () system call sleeps this process and allows kernel to schedule another process, which may access that file, whose access time is under watch of slept process. If file is accessed, then program prints a message and if not code proceeds. After every minute, signal alarm is send to the process, process wakes up due to wakeup () signal catcher function (though wakeup () in this program has no code, it is just a demo) and returns to the position after the pause () system call and loop back again.

The purpose of this program is demonstration of alarm signal, which is another time related system call.

All these 4-system calls for time are heavily depends upon the system clock. Kernel itself manipulates many time counters in its own code, when it deals with clock interrupt.

The Clock

As we saw up till now in scheduling and memory management, the clock is critical part of hardware of a system & the operating system must write its clock interrupt handler to do various functions such as..

1) Restart the clock.
2) Schedule and execute those kernel's internal functions, which are based on the internal timers.
3) Gives "***execution profiling***" capability to the kernel & to the user processes.
4) Gathers system & process's accounting statistics.
5) Keeps track of time.
6) On request, send alarm signal to processes.
7) Wakes up swapper process periodically, which is crucial in "***swapping***" based memory management.
8) Control process scheduling, which is crucial in multitasking, multi-user & multithreading systems.

Out of above functions, some are done on every clock interrupt while some are done after several clock ticks. *Clock handler function has its processor execution level, set high, so as to prevent other peripheral device interrupts, while handler function is actively running.*
That is why clock handler function execution is quite fast made the time gap as small as possible. Obviously this can be achieved better and better when other interrupts are blocked and thus clock interrupts having full freedom.

The algorithm of Clock Interrupt Handler is .....

```
01 : algorithm : clock()
02 : input : none
03 : {
04 :        restart the clock; /* so that it will interrupt again and again */
05 :        if(callout table is not empty)
06 :        {
```

```
07 :            adjust callout times accordingly;
08 :            schedule callout function, if time is elapsed;
09 :      }
10 :      if(kernel profiling is set on)
11 :            note the program counter at time of interrupt;
12 :      if(user profiling is set on)
13 :            note the program counter at time of interrupt;
14 :      gather system statistics;
15 :      gather per process statistics;
16 :      adjust the measurement of process's CPU utilization;
17 :      if(1 second or more time is elapsed since last time here and this interrupt is
18:         not in critical region of code)
19 :      {
20 :            for(all processes currently in system)
21 :            {
22 :                  adjust alarm time if alarm signal is active by any process;
23 :                  adjust measure of CPU utilization;
24 :                  if(the process which is to be executed is in user mode)
25 :                        adjust that process's priority;
26 :            }
27 :            wake up swapper process if necessary;
28 :      }
29 : }
30 : output : none
```

Most of tasks in this algorithm are general functions of clock interrupt handler as explained early. But some points in this algorithms like restarting, callouts, profiling, accounting & statistics are going to be discuss separately here after…

Restarting the clock

Clock interrupt must occur again and again to do its all-critical system life-maintaining task. So when the clock interrupt occur once, it is expected that clock interrupt handler function, besides all its other tasks must restart the clock again so that it will interrupt again after suitable (i.e. system dicided) time interval is elapsed.

But the instruction to do so, are heavily hardware dependent and done by assembler code, and obviously different from hardware to hardware.

Internal System Timeouts/Callouts

It is very important to note that, device drivers and network protocols require execution of kernel function on real time basis. For example, a process may keep a terminal in raw mode means terminal is reading user's input form console read () system call will satisfy user's request after certain time period, not when enter is pressed! (Usually system waits for input to satisfy until user press enter key. But if user spends much time, then its share wastage of CPU time. DOS works like that. But real time response cannot spare such wastage).

For situation like above kernel keeps a separate data structure called as callout table. Each entry in this table is made up of 2 elements…

1) Pointer to function along with its parameters which to be invoked after expiration of certain time.

2) Time in "***clock ticks***" until the function should be called. This is called as "***time to fire***".

User processes do not have direct control on entries in this table. Kernel algorithms only can make entries as needed. The order of entries in this table does not depend upon "***by which order the entries are made***", but actually depends upon "***time of fire***" field. Thus the enters in this table are "***timely ordered***" and that is why the time field for each entry in the callout table is stored as the amount time to fire after the previous element fires. The total to fire for a given element in the table is the sum of "***times to fire***" of all entries "***up to***" and "***time to fire***" current element.

From (1) to

F^n Name	Time to fire
a()	**-2**
b()	**3**
c()	**10**

from (2) to

F^n name	Time to fire
a()	**-2**
b()	**3**
f()	**2**
c()	**8**

To explain above idea in somewhat more detail, consider this figure of callout tables. Figure (1) is the callout table before making new entry and figure (2) is the callout table after making new entry of function f().

Before explaining the figure, let us see some important theory. At every clock interrupt, the clock interrupt handler checks for any entry in the callout table (Line 5 of algorithm). If there are any entries in the callout table, decrements the time of the first positive entry in the table by 1 (Positive means whose time field is +ve).

Here a question may arise, if the time field is the number of ticks after which that function should be invoked, then how can it be –ve?

Answers: Actually, handler schedules and runs that function whose time field is less than or equal to 0 (Line 8 of algorithm). It does not call that function to run directly, but does it indirectly. Because if function gets too long to complete (greater than next tick) then next tick will not occur as clock gets block in handler algorithm, (because first statements in algorithms is about restarting the clock, but this will not make its effect until handler function algorithm completes) as algorithm blocks next clock interrupt until algorithm's completion, by setting processor execution level to block interrupt. That is why instead of calling schedule function directly, algorithm schedules the function to run causing "***software interrupt***" also called as "***programmed interrupt***". This interrupt is caused by hardware dependent machine instruction. This intelligently delay "***function call***" during the execution of algorithm and thus algorithm proceeds smoothly without mixing with function call. How? This happens because we know that "***software interrupts***" are at the bottom (of low priority) of our "***processor execution level concerned interrupt diagram***" which was seen in chapter 1, now as clock interrupts are at high priority and we already block it in algorithm, "***software interrupts***" also are blocked and will not occur until algorithm complete. When algorithm completes and when a processor execution drops, the machine instruction of software interrupt will occur and function will be invoked.

Coming back to our question, within this period (i.e. occurrence of software interrupt) many interrupts including clock interrupt thus can occur and hence "***the exact time decided to schedule function and the actual time when function gets schedule*** " may differ.

Means suppose function a() has its time field in callout entry 0. Hence it is the function, which is to be schedule to invoke. Handler decides it to run and tells this to "***software interrupt***". As per our discussion in above paragraph suppose 2 more ticks occur until "***software interrupt*** " occurs here the time field will be –2 for a().

In other words, negative time field for a function indicates function was schedule to invoke in past, but yet not invoke and more clock ticks are occurred in between.

This makes the idea clear why time field for a entry in callout table can be negative.

Finally, when actual "***software interrupt***" occurs, the handler removes the entry for that function from the callout table and invokes the function. Obviously this can occur after some ticks already passed from the actual scheduled time. By above example a() will get executed (possibly) when algorithm runs 3rd time that 2 ticks late.

Again remember that, whole above story (i.e. of negative time value) is possible only for first entry in callout table.

Also remember that, the "***decrement by 1***" function of algorithm (as explained on the previous page) is concerned only with first positive entry. The negative nature of first entry is not due to "***decrement***" but due to adjustment for showing how many ticks the function gets late to execute. So when algorithm arrives at the table and finds that first entry is already negative then it does not decrement it. Instead, it searches for first positive entry and then decrements it by 1.

This concludes the theory part clock handler's dealing with the callout table. Now we will discuss about the figures on previous page.

Out of the two diagrams in figure, 1st diagram is the callout table before entry of function f(). The first entry is for function a() whose time field is –2, means this function was expected to get scheduled 2 ticks ago.(In other words, scheduling of function a() is delayed by 2 ticks). The second entry is for function b() , whose time field is 3, means this function is expected to get scheduled after 3 ticks. Here a confusion may occur that, the total time of fire will be 3 + (-2) = 1 tick. But no! Negative entry is not considered in total time's summation. The third entry is for function c(), whose time field is 10, means this function is expected to get scheduled after 13 ticks. Because 3 of b() + 10 of c() is 13. Here again note that –2 of a() is not considered in summation.

Now consider the 2nd diagram and assume that entry of f() is to be made which is expected to get scheduled after 5 ticks. To make this entry algorithm starts finding of correct slot in callout table. It can not put it on the top of a(), because a() is expected to schedule (rather its already late for a()). It can not put it below a() & above b(), because b() has its time field 3 and expected to get scheduled after 3 ticks, while our f() is expected to get scheduled after 5 ticks. It can put the entry of f() below b() & above c(), because c() is expected to get scheduled after 10 ticks, i.e. later than 5 ticks of f(). So the correct slot for entry of f() is between b() &c(). Thus in the "***function***" field of entry, function f() entry is made. Now what about its time field? We cannot write 5 there. Because entering 5 in time field tells that f() will get scheduled after 3 ticks of b() and 5 ticks of itself. This gives total of 8 ticks. This is obviously not expected. To allow f() to get scheduled after 5 ticks, we have to make sum of entries up to f(). First a() is skipped , as it is negative. Next is b(), whose time field is 3. Now to make total ticks of f() as 5, time field of f() should have valued, 5-3 i.e. 2. Hence f() will get scheduled after 5 ticks, out of which 3 are of b() and 2 are of itself.

Now what the time field of c(), whether its value will change or not? Certainly its value will be changed. As c() is expected to get scheduled after 13 ticks , if we keep 10 as it is, then total will give 3 of b() + 2 of f() + 10 of c() (as a() has –2, skipped) total of 15 ticks which is wrong. To allow c() to get scheduled after 13 ticks, the valued should be 13 – (3 + 2) = 8. Thus time field of c() changes to 8. Now we can say that function c() will get schedule, when 3 of b(), 2 of f() & 8 of c() will complete, which is correct 13 ticks. (i.e. 3 + 2 + 8 = 13). This makes one more pint clear that, whenever a new entry is made all the entries below that will modify their time fields accordingly.

On implementation level, the callout table can be either implemented by using linked list or by using array. Though it seems (and actually it is) that using linked list is better, array implementation will also do well, by pushing later entries further, when a new entry is made. But array implementation will be only effective if kernel does not use callout table too much.

Profiling

Kernel profiling gives a measurement of

(a) How much time kernel spends executing individual routines (functions) of the kernel code. (E.g. bread, bwrite, getblk, etc)

(b) How much time the system is executing in user mode versus kernel mode.

To allow this to work, kernel requires a special driver called as kernel profile drivers. This driver works by monitoring the relative performance of "***kernel modules***" which is done by sampling "***system activity***" of the time of a "***clock interrupt***".

Kernel profile Driver already has a record of all kernel addresses, which are going to be sampled. These address are usually of kernel's internal functions. The driver writer keeps knowledge of these kernel addresses while writing this driver (Design issue).

When we need profiling, we have to enable the kernel profiling. When it gets enabled, the clock interrupt handler invokes the interrupt handler function of **K**ernel **P**rofile **D**river (KPD) (lines (10) to (13) in algorithm). This KPD'S interrupt handler first determines whether the "***processor mode***" is currently the "u***ser mode***" or "***the kernel mode***". This can be easily known by knowing the value of "***program counter***" at the time of "***clock interrupt***". If the mode is "***user mode***", the KPD'S handler increments " ***a special count variable***", which is later used to compute "***user profiling***", but if the mode is "***kernel mode***", it increments "***an internal counter corresponding to the program counter***".

User processes can read the profile driver to obtain above "***counter***" variable and then can use them in statistical measurements.

For example we will take some hypothetical addresses of kernel routines, which are used during execution of processes.

No.	Algorithm Name	Hypothetical Name	Count
1	bread()	100	5
2	breada()	150	0
3	bwrite()	200	0
4	brelease()	300	2
5	getblk()	400	1
6	user	-	2

Assume that program counter is sampled over 10 clock interrupts at 110, 330, 145, 125, 440, 130, 32 and 104 addresses in user space then the kernel will save "***count***" as shown in above figure.

Above figure also tells us range of addresses for kernel routines. The addresses shown in figure are starting addresses of those kernel routines. Thus hypothetically we can say that kernel routine bread start at 100th address and ends at 149th address, because breada () starts at 150th address. Then kernel routine breada () starts to 150th address and ends at 199th address, because bwrite () starts at 200th address. Then bwrite () starts at 200th address and ends at 299th address, because brelease () starts at 300th address. The brelease () starts at 300th address and ends at 400th address. Then getblk () starts at 400th address and rages up to the user address which is not specified in the figure. Finally users made udfs are called while addresses are not specified in the figure.

This figure shows that out of 10 clock interrupts 5 occurred for read () means consume 50% of time, 20% by brelease (), 10% by getblk () and 20% in user mode.

If we observe these range of kernel routines, we come to know that clock interrupts sampled for bread () occurred for 5 times i.e. at 110, 145, 125, 130 and 104. Because all these addresses fall into the range of bread () form 100 to 149. Thus count is 5.

Then in the given list of addresses, no clock interrupts are occurring between the address range of breada () i.e. form 150 to 199. Therefore count is 0. Same that for bwrite () too.

Then clock interrupts sampled for brelease () are occurring 2 times, because brelease () ranges from 300 to 399 and in the given list addresses 330& 320 are sampled. Hence count is 2.

For the range of getblk (), i.e. from 400th onwards, address 440 is sampled in the list. So clock interrupt occurred only once, hence count is 1.

By common sense, it is also easy to determine that calls to kernel routines bread (), breada (), brelease () & getblk () will be called in read () system call and bwrite () & again brelease () will be called in write () system call. So user process called read () & write () for which above story is happening. Obviously user process will call read (), and there will be mode switch from user to kernel. When read () completes, mode switches back to user space. Suppose one clock interrupt occurred here. When write () will be called mode again switches to kernel mode and when write () completes it switch back to user mode. Suppose again one clock interrupt occurs. Summing up, we can say that 2 clock interrupts occur in user space and thus count is 2.

Note that, the time measured during kernel profiling does not include the time spent during execution of (a) clock interrupt handler itself and (b) the code which blocks clock level interrupts. Here for the case (b), clock cannot interrupt such code, because as interrupts are blocked at the level of clock, for clock too, it is a critical section and thus even interrupt handler cannot execute during this critical section and thus kernel profiling cannot be done for this code execution time. Unfortunately such codes are actually real candidates of kernel profiling measurement. Thus the kernel profiling we received must be considered with this flaw.

Weinberger developed a scheme for counting into code blocks like if…else… statements to give how many times they are executed. But the scheme given by him increases CPU time form 50% to 200% and hence his method is considered to be non-practical and hence not used.

On user level, a user can profile execution of processes at user level, by using *profil* system call. Whose syntax is….

profil(buffer , buffer-size , offset , scale);

Where

a) ***buffer*** : It is the address of a buffer in user space, which gets filled (on return) with accumulated frequency counts of execution in different addresses of the process. The contents of this buffer depend upon buffer-size and scale.
b) ***buffer-size*** : This the size of buffer array, i.e. in other words it is the size of (a).
c) ***offset*** : This is the starting address(virtual) of the routine or function in the process, of which we want to calculate profile. This address is given usually of the main entry point function of a process. Because we commonly need to profile whole process not a single function in it.
d) ***scale*** : This is a "***factor***" which maps virtual addresses of process (or say given virtual address in "***offset***" parameter) into the buffer i.e. into (a).

! If the value 0xfff is used as "***scale***", then there will be one to one mapping of virtual address to the address in buffer. Obviously bigger the process, programmer should set buffer size bigger, to correctly map virtual address of required process to the buffer. E.g. suppose virtual address of a function in process is 4000 (in decimal) and buffer size is buffer [2048], then address 4000 cannot be mapped with this scale of one to one mapping.

! If the value 0x7fff is used as "***scale***", then two virtual addresses will be mapped to one (2:1) to single address in buffer. E.g. continuing above example, the virtual address 4000 (in decimal) now can be mapped into buffer whose size is 2048. Because as there is 2 : 1 ratio, the buffer size 2048 can map virtual addresses 2048 * 2 i.e. 4086.

! If the value 0x3fff is used as "***scale***", then four virtual addresses will be mapped to one address in buffer (4:1). So to map virtual address 4000 now buffer size can be just 1024. It can map 1024 * 4 = 4096 addresses.

*The 3 values of scale i.e. 0xffff, 0x7fff, 0x3fff are just examples. You can extend this idea further and can give your own "***Scale Factor***". But for clarity usually one to one mapping i.e. 0xffff scale factor is used.*

How profil() works? When clock interrupts the process in user mode, the clock interrupt handler examines the "***user program counter at the time of interrupt***" (recall that, user stack frame has program counter value as one of its contents. See chapter 2) and compares its value with the "***offset***" parameter of profil () system call and then looking into the buffer (parameter (a)) increments one of according location inside that buffer whose address (i.e. location in buffer) is given by "***scale factor***". Means suppose "***offset***" is given the address of temp () function in a process (e.g.: 400). Now suppose when clock interrupt occurs, the program counter is at 409. Suppose there is "***one to one mapping***" scale factor. Means indirectly location 400 in buffer is mapped to virtual address 400 of temp (). So when program counter is at 409 in temp () and as clock interrupt occurred here, the location 409 in buffer will be incremented to 411 or 413. Here 411 value assumes that "***int***" size is 2 bytes and 413 value assumes that "***int***" size is 4 bytes. (i.e. 409 + 2 = 411 or 409 + 4 = 413).

We will take a complete example to demonstrate the usage of profil() system call. Assume that programmer had written a program and wanted to profile itself when program runs as process. The buffer size in program is made 4096 to account whole size in program safely. It is also assumed that the size of "***int***" is of 4 bytes. And the program runs for about 10 seconds (because it has infinite loop).

The sample program in Bach was run on AT&T 3 B20 model:

```
# include <signal.h>
int buffer[4096];
void main(void)
{
 /* declarations */
 extern void theend(void);
 extern void f(void);
 extern void g(void);
 int offset , endof , scale , af , ag , text;
 /* code */
 signal(SIGINT , theend);
 endof = (int) theend;
 offset = (int) main;
 /* calculate number of bytes for text of program */
 text = (endof – offset + sizeof(int) –1) / sizeof(int);
 /* build parameters of profil() */
 scale = 0xffff; /* one to one mapping */
 printf("offset %d endof %d text %d\n" , offset , endof , text);
 af = (int) f;
 ag = (int) g;
 printf("f %d g %d f_diff %d g_diff %d\n" , af , ag , af_offset , ag_offset);
 /* call profil() */
 profil(buffer , sizeof(int) * text , offset , scale);
 for(;;) /* infinite loop*/
 {
        f();
        g();
 }
```

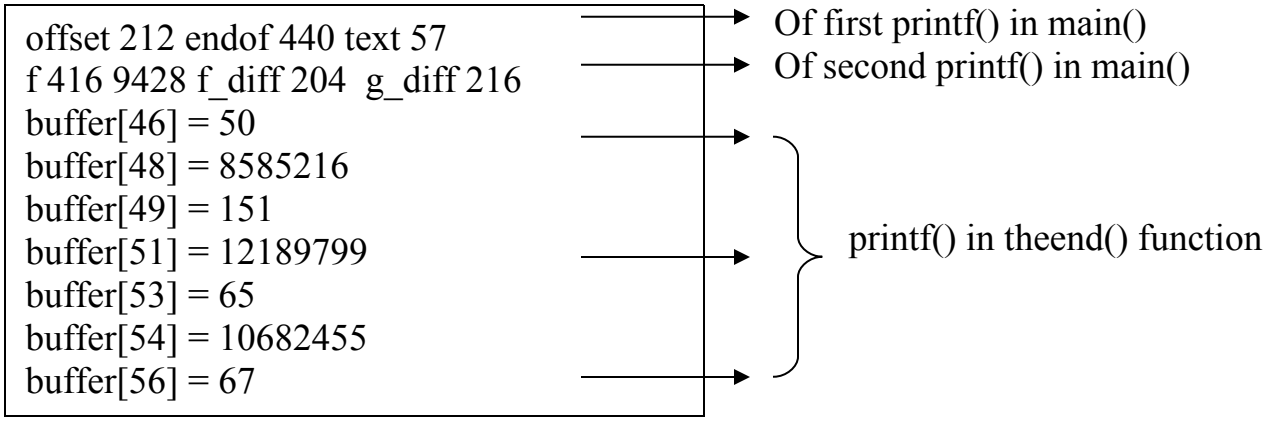
```
}

void f(void)
{
}

void g(void)
{
}

void theend(void)
{
 /* declarations */
 int I;
 /* code */
 for(I=0;I<4096;I++)
 {
        if(buffer[I]);
                printf("buffer[%d] = %d\n" , I , buffer[I]);
 }
 exit();
}
```

And the sample out is (possible output)



In the beginning portion of code, program calls signal system call for SIGINT (i.e. interrupt) value and specifies theend () as its signal catcher function when interrupt will occur, system will call theend function which in turn runs a loop, prints contents of buffer and then exits the process.

The endof & offset variables are assigned buy virtual addresses of theend () & main() respectively. (Recall that "***offset***" parameter is usually specified by the main () entry point function of process.). For example purpose suppose that virtual address of main () is 212 and that of theend () is 440. So variable offset will have value 212 and variable endof will have value 440.

Then the "***text***" variable is calculated for which we want to measure profile. Supposing that program wants to profile the execution form address of main () to address of profil (), then the "***text***" will be difference between the addresses of theend () & main (), i.e. 440 - 212 which is 228. The "***word***" is a unit of 4 bytes. So the same "***text***" variable in the form of words will be 228/4 = 57. The first printf() statement in main() prints these 3 things means "***offset 212 endof 440 text 57***".

The variables af (i.e. for address of f ()) and ag (i.e. for address of g ()) are assigned by virtual addresses of function's f () & g () respectively. Then "***address difference***" between virtual address of f () & virtual address of main () is calculated as f_deff, which is, 416-212 i.e. 204 (supposing f () has its starting virtual address 416. then "***address difference***" between virtual address of g () & virtual address of main () is calculated as g_diff, which is, 428 – 212 = 216 (supposing that g() has its starting virtual address 428). These 4 things i.e. virtual address of f (), virtual address of g (), address difference between f () and main () and address difference between g () and main () are printed by second printf() in main as "***f416 g 428 f_deff 204 g_diff 216***".

As there is one to one mapping, between locations in buffer and virtual addresses in process, the addresses in f () function will get mapped to buffer entries 51, 52, & 53. How? Recall that difference between virtual addresses of f () & main () i.e. f_diff is 204, and our "***int***" size is 4, if we divide 204/4 it becomes 51. So we can say that up to buffer entry 50, main () resides and from 51, f () function starts. Similarly we know that g_diff is 216, so difference between storing addressees g () function and main () is 216. So g () will start from buffer entry 54 because 216/4 is 54.

Now we know starting of f () in buffer is 51 and storing of g () is 54. Thus we can say f () extends from 51 to 53, means it got 3 entries in buffer and its size is 12 bytes (as 3 buffer entries *4 =12 or 216 −204 =12).

Similarly g () will also have 3 buffer entries from 54 to 56 and its size is also 12 bytes. As main () extends up to 200, buffer entries 46, 47, 48, 49, 50 will be for addresses in "***for***" loop of main (). Because 46 * 4 = 184, 47 * 4 = 188, 48 * 4 = 192, 49 * 4 = 196, 50 * 4 = 200.

Though above example shows demonstration of usage of profil() system call, on practical level, users are not supposed to call profil() directly in their program because of its complicated nature. Rather "***C***" compiler option is used to generate code to profile that process/ program.

Accounting & statistics

When clock interrupts the system, the system itself may be in one of the 3 situation i.e. (a) Either running in kernel mode *or*
(b) Running in user mode *or*
(c) Idle.
The system is said to be "***idle***" when all processes currently in the system are sleeping means waiting for occurrence of an event. During this stage system is not executing any process.

Kernel keeps internal counters for these 3 processor states of the system itself. And adjust them during each clock interrupt. It also notes current mode (out of above 3) of the machine. Then user process, later, can gather & analyze these counter stored in kernel to maintain the statistics.

Every process has 2 fields in its "***u area***" to keep record of elapsed kernel time and user time. So while handling clock interrupts, the kernel updates the appropriate field (out of the above 2) for the executing process, depending upon whether the process is currently running in user mode or kernel mode.

Parent processes can gather statistics of their children during the wait () system call. (recall that by wait () parent waits up to child's death i.e. exit)

Also every process has one field has one filed in its "***u area***" to allow kernel to write its own memory usage (made for this process). When clock interrupts running process, the kernel calculates the total memory used by a process. This is done by taking "***private***" and "***shared***" regions of the process, into consideration. For example, suppose a process, has its text region of 50KB , shared with four other processes (i.e. total 5, one this & four other), then it has it private data region of 25KB and private stack region of 40KB. Then kernel charges this process for 75KB of memory usage. That is 50/5 + 25 + 40 = 75. (Note the billing for shared text region). Above scheme is 0K for swapping systems. But for paging system, kernel calculates the memory usage by counting the number of valid pages in each region. Thus if a processes uses 2 private region (i.e. data and stack) and one shared region (i.e. text) which is shared with one another process (i.e. text region is shared by total 2

processes) then kernel charges for number of valid pages in both private regions + half the number of valid pages in the shared region. ("***Half***" because there are 2 processes using /sharing this shared region).

The value 75k obtained above, or whatever value obtained for valid pages, it logged i.e. written to this field in "***u area***". When process exits, value from this field is written into the process table slot of this process (which remains in process table though process exits. That's why exiting process is called as Zombie). Later administrator can gather information about this process's accounting & statistics from its process table slot (which is still there, though process already exited) and uses for customer billing.

Keeping Time

Kernel has a "***timer variable***" which is initialized when system boots and is incremented at every clock interrupt. This is the same timer variable, whose value (current) is returned as return value of time () system call. This is the same timer variable used by the kernel to calculate the "***total real time***" of execution of a process.

The kernel, during fork () system call for creating a process, saves "***time of creation of process*** " in the "***u area***" of newly created process. When process exits, the current time when process exits, the current time when process exits is noted and subtraction of these 2 times gives the "***total real execution time of a process***".

There is another "***timer variable***" which is set by stime() system call and is updated once a second and thus gives the system's calendar time to which we call whole system time, that is returned by sysdate & time related functions.