 # 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:

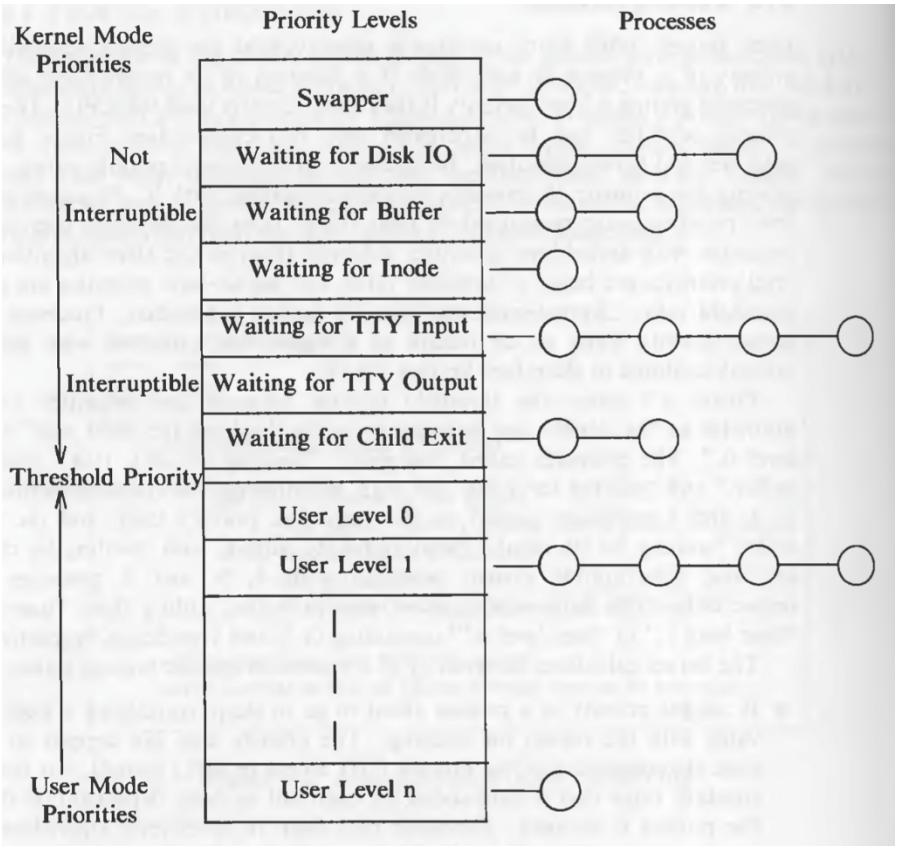
```
/*  Algorithm: schedule_process
 *  Input: none
 *  Output: none
 */

{
   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.

Scheduling Parameters

Each process table entry contains a priority field. The priority is a function of recent CPU usage, where the priority is lower if a process has recently used the CPU. The range of priorities can be partitioned in two classes: user priorities and kernel priorities. It is shown in the diagram below:



Each priority has a queue of processes logically associated with it. The processes with user-level priorities were preempted on their return from the kernel to user mode, and processes with kernel-level priorities achieved them in the **sleep** algorithm. User priorities are below a threshold value and kernel priorities are above a threshold value. Processes with low kernel priority wake up on receipt of a signal, but processes with high kernel priority continue to sleep. The user level 0 is the highest user level priority and user level n is the lowest.

The kernel calculates process priorities in these process states:

* It assigns priority to a process about to go to sleep. This priority solely depends on the reason for the sleep. Processes that sleep in lower-level algorithms tend to cause more system bottlenecks the longer they are inactive; hence they receive a higher priority than process that would cause fewer system bottlenecks. For instance, a process sleeping and waiting for the completion of disk I/O has a higher priority than a process waiting for a free buffer. Because the first process already has a buffer and it is possible that after the completion of I/O, it will release the buffer and other resources, resulting into more resource availability for the system.
* The kernel adjusts the priority of a process that returns from kernel mode to user mode. The priority must be lowered to a user level priority. The kernel penalizes the executing process in fairness to other processes, since it had just used valuable kernel resources.
* The clock handler adjusts the priorities of all processes in user mode at 1 second intervals (on System V) and causes the kernel to go through the scheduling algorithm to prevent a process from monopolizing use of the CPU.

When a process is running, every clock tick increments a field in the process table which records the recent CPU usage of the process. Once a second, the clock handler also adjusts the recent CPU usage of each process according to a decay function on system V:

`decay (CPU) = CPU / 2;`

When it recomputes recent CPU usage, the clock handler recalculates the priority of every process in the "preempted but ready-to-run" state according to the formula.

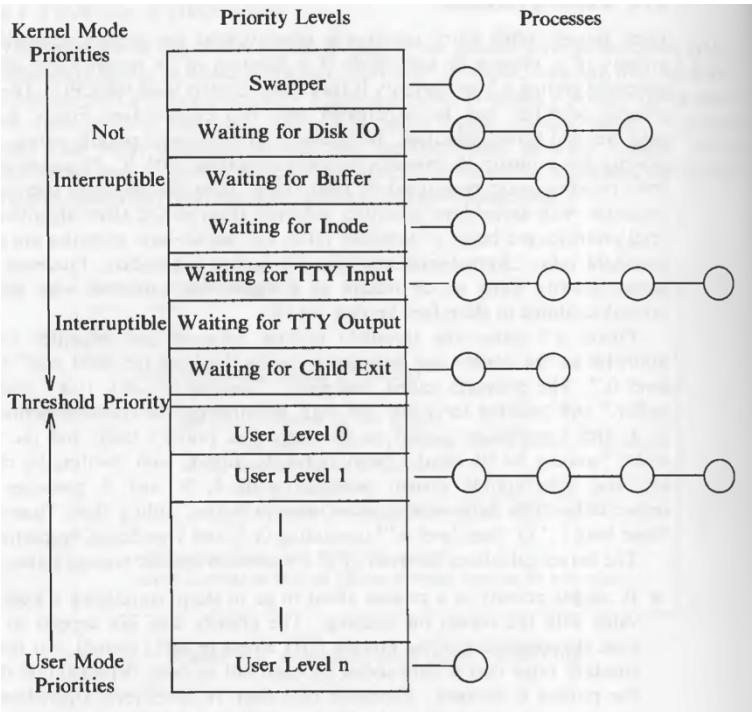
`priority = ("recent CPU usage" / 2) + base level user priority;`

where `base level user priority` is the threshold priority between kernel and user mode. A numerically low value implies a high scheduling priority. As the effect of once a second recalculation, processes with user-level priorities move between the priority queues. The priorities in kernel mode do not change.

If a process is in critical region of the kernel (i.e. the process execution level is risen), the kernel does not recompute the priorities on the one second clock tick, it recomputes the priorities at the next clock tick after the critical region is finished.

****Examples of Process Scheduling****

Consider the following time diagram:



Process A, B, and C are created and initially given the priority 60, which is the highest user-level priority. Assuming that the processes make no system calls, and process A gets scheduled first, after one second the CPU count of A becomes 60. And when it is recalculated, it becomes 30 (decay = 60 / 2). And the priority becomes 75 (priority = 30 / 2 + 60). Then B gets scheduled and the calculation continues every second.

Controlling Process Priorities

Processes can use the **nice** system call to crude control their scheduling priorities.

`nice(value);`

where `value` is added in the calculation of process priority

`priority = ("recent CPU usage" / constant) + (base priority) + (nice value);`

The **nice** system call increments or decrements the **nice** field in the process table by the value parameter, although only the superuser can supply **nice** values that increase the process priority. The term "nice" is used because decreasing the priority (which is the usual use-case) is being "nice" to other processes. Processes inherit their **nice** value of their parent during the **fork** system call. **nice** system call works for running processes only; a process cannot reset the **nice** value of another process.

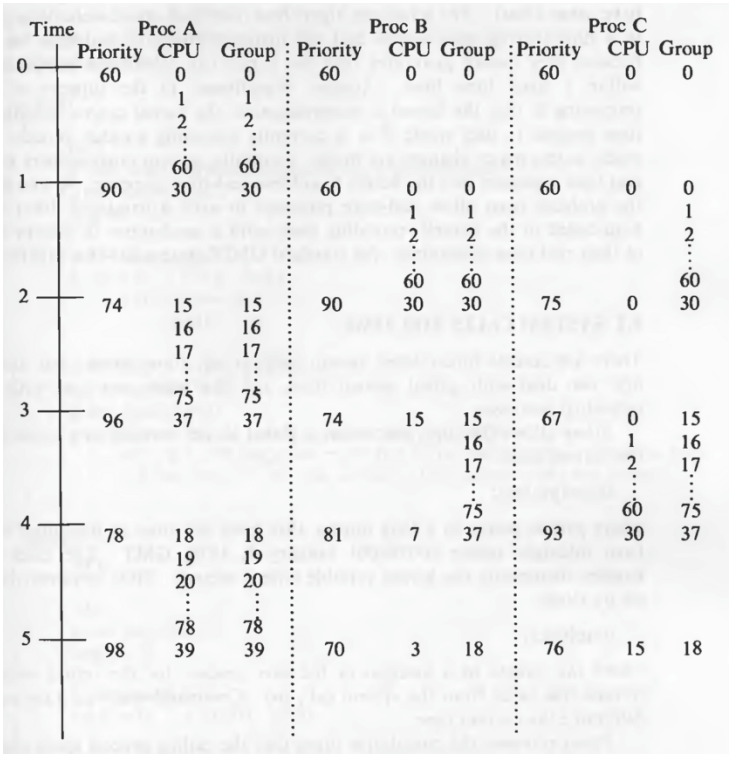
Fair Share Scheduler

The scheduler algorithm described above does not differentiate between classes of users. In **fair share scheduling**, there are groups of processes and the time quantum is allocated equally to all the groups, even if number of processes in each group might be different. For example, if there are 4 fair share groups with 1, 2, 3, and 4 processes

respectively, then the one process in group 1 will get twice time quantum than a process in group 2, thrice than a process in group 3 and four times than a process in group 4.

To implement this scheme, another field is added in the u-area for "fair share group priority". It is shared by all the processes in the same fair share group. The clock interrupt handler increments the fair share group CPU usage field for the running process, just as it increments the CPU usage field of the running process and decays the values of all fair share group CPU usage fields once a second. When calculating priorities, a new component of the calculation is the group CPU usage, normalized according to to the amount of CPU time allocated to the fair share group.

For example, consider the processes in the following diagram:



Process A belongs to one group and processes B and C belong to another group. Therefore, the "group" value is shared between B and C. The priorities will be calculated by this formula:

`priority = (CPU usage / 2) + (Group CPU usage / 2) + base priority`

That is why processes execute in this manner: A, B, A, C, A, B, A, C, and so on...