# PROCESS SCHEDULING AND TIME

1

The scheduler function on the UNIX system uses relative time of execution as a parameter for scheduling.

Every active process has a scheduling priority.

The kernel switches context to the process with the highest priority.

The kernel recalculates the priority of the running process when it returns from kernel mode to user mode, and it periodically readjusts the priority of every “ready-to-run” process in user mode.

2

PROCESS SCHEDULING

The scheduler on the UNIX system belongs to the general class of operating system schedulers known as ***round robin with multilevel feedback.***

1. The kernel allocates the CPU to a process for a time quantum.
2. Preempts a process that exceeds its time quantum.
3. Feeds it back into one of several priority queues.

3

```
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;
}
```

4

## Process Scheduling Algorithm

At the conclusion of a context switch, the kernel executes the algorithm to schedule a process.

Selects the highest priority process from those in the states "ready to run and loaded in memory" and "preempted".

It does not select a process that is not in main memory.

If several processes tie for highest priority, the kernel picks the one that has been "ready to run" for the longest time.

If there are no processes eligible for execution, the processor idles until the next Interrupt.

5

## Scheduling Parameters

Each process table entry contains a priority field for process scheduling.

The priority of a process in user mode is a function of its recent CPU usage.

lower priority if a process has recently used the CPU.

The range of process priorities are partitioned into two classes:

user priorities and kernel priorities.

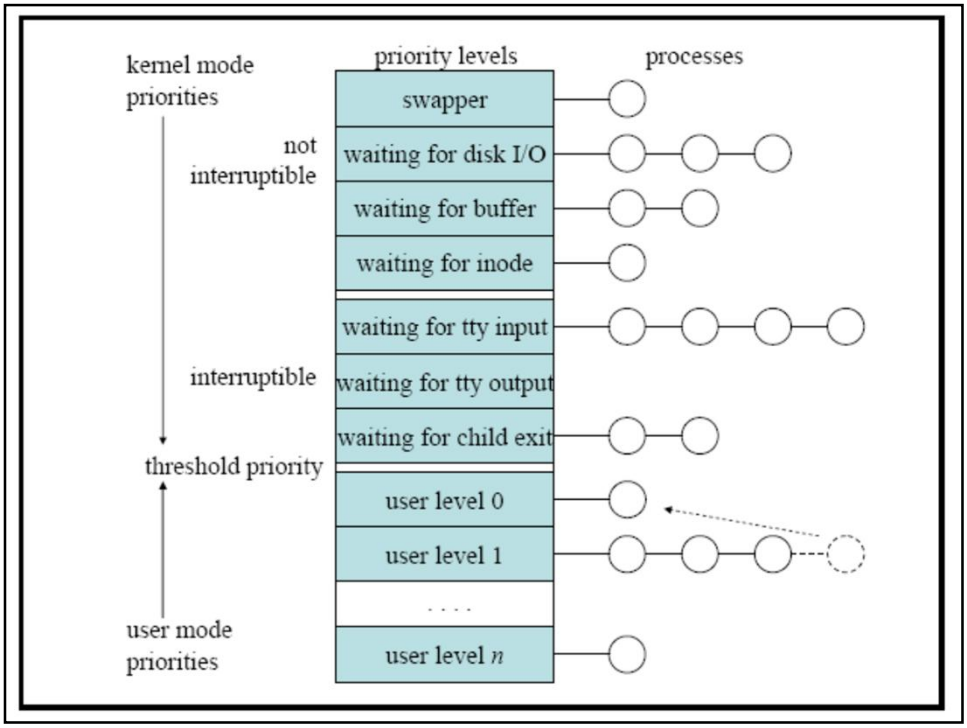
Each class contains several priority values.

Each priority has a queue of processes logically associated with it.

User level priorities are below a threshold value, and kernel-level priorities are above the threshold value.

Processes with user level priorities were preempted on their return from the kernel to user mode.

6




7

Processes with kernel-level priorities achieved them in the *sleep* algorithm.

Kernel-level priorities are further subdivided:

Processes with low kernel priority wake up on receipt of a signal, but processes with high kernel priority continue to sleep.

The priorities called “swapper”, “waiting for disk *I/O”, “*waiting for buffer”, and "waiting for inode" are high, noninterruptible system priorities.

8

The kernel calculates the priority of a process in specific process states:

1. It assigns priority to a process about to go to sleep, correlating a fixed, priority value with the reason for sleeping.

   The priority does not depend on the runtime characteristics of the process.

   hard-coded for each call to sleep, dependent on the reason the process is sleeping.

   Processes that sleep in lower-level algorithms has high priorities.

2. The kernel adjusts the priority of a process that returns from kernel mode to user mode.

   The process may have previously entered the sleep state, changing its priority to a kernel-level priority that must be lowered to a user-level priority when returning to user mode.

   kernel penalizes the executing process in fairness to other processes, since it had just used valuable kernel resources.

9

3. The clock handler adjusts the priorities of all processes in user mode at 1 second intervals (on System V).

   At every clock interrupt, the clock handler increments a field in the process table that records the recent CPU usage of the process.

   The clock handler also adjusts the recent CPU usage of each process according to a decay function:

   decay(CPU) = CPU/2;

   the clock handler also 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.

10

The kernel does not change the priority of processes in kernel mode, nor does it allow processes with user-level priority to cross the threshold and attain kernel-level priority, unless they make a system call and go to sleep.

The kernel attempts to recompute the priority of all active processes once a second, but the interval can vary slightly.

--If the clock interrupt had come while the kernel was executing a critical region of code.

The kernel responds naturally to interactive requests such as for text editors or form entry programs: such processes have a high idle-time-to-CPU usage ratio, and consequently their priority value naturally rises when they are ready for execution

11

## Example of Process scheduling

scheduling priorities on System V for 3 processes A, B, and C, under the following assumptions:

-- Initial priority 60
-- The clock interrupts the system 60 times a second.
-- The processes make no system calls.
-- No other processes are ready to run.

The kernel calculates the decay of the CPU usage by:

CPU = decay(CPU) = CPU/2;

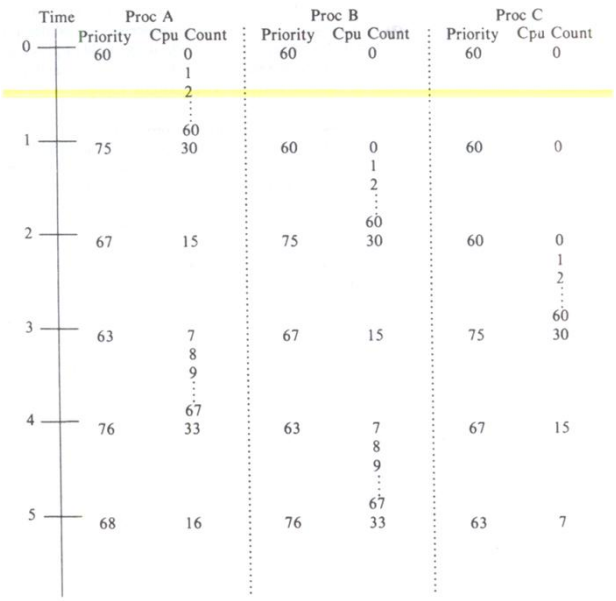
and the process priority as

priority = (CPU/2) + 60;

Assuming process A is the first to run and that it starts running at the beginning of a time quantum, it runs for 1 second: During that time the clock interrupts the system 60 times and the interrupt handler increments the CPU usage field of process A 60 times (to 60).

The kernel forces a context switch at the I-second mark & recalculates priority.

12




13

## Controlling Process Priorities

Processes can exercise crude control of their scheduling priority by using the *nice* system call:

```
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 of the parameter.

only the superuser can supply *nice* values that increase the process priority.

Processes inherit the *nice* value of their parent during the *fork* system call.

The *nice* system call works for the running process only; a process cannot reset the *nice* value of another process.

14

## Fair Share Scheduler

The scheduler algorithm described before does not differentiate between classes of users.

-- it is not possible to allocate a fraction of CPU time to a specific set of process.

However, such considerations are important in a computer center environment, where a set of users may want to buy half of the CPU time of a machine on a guaranteed basis, to ensure a certain level of response.

Fair Share Scheduler overcomes this problem.

15

## Clock

- The function of the clock interrupt hander 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.

16

## Clock

- At every clock interrupt, the clock handler decrements the time field of the first entry.
- If the time field of the first entry is less than or equal to 0, then the specified function should be invoked.
  - the clock handler does not invoke the function directly.
- The clock handler schedules the function by causing a "software interrupt."
  - because software interrupts are at a lower priority level than other interrupts, they are blocked until the kernel finishes handling all other interrupts.
  - clock interrupts could occur until the software interrupt occurs and, therefore, the time field of the first entry can have a negative value.

17

## Clock

- Some kernel operations such as network protocols require invocation of kernel functions on a real-time basis.
- The kernel stores the necessary information in the *callout* table.
  - the table consists of the function to be called when time expires, a parameter for the function, and the time in clock ticks until the function should be called.
- The kernel sorts entries in the callout table according to their respective "time to fire".
  - the time field for each entry is stored as the amount of time to fire after the previous element fires.

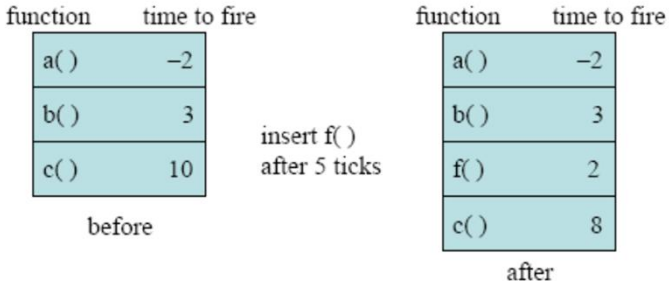
18

```
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 (1second 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;
      } /* for */
      wakeup swapper process
         if necessary;
   } /* if (1second...) */
}
```

19

# Callout Table Example




20

End

21