Scheduling

Learning Objectives

- To understand the role of a scheduler in an operating system
- To understand the scheduling mechanism
- To understand scheduling strategies such as non-preemptive versus preemptive
- To be able to explain various scheduling algorithms like first-come first-served, round robin, priority-based etc
- To examine commonly using schedulers on Linux and Microsoft Windows

Scheduler

- Scheduler allocates CPU(s) to threads and processes. This action is known as scheduling.
- The scheduler is a part of the process manager code that handles scheduling.

Scheduling Mechanisms

- The *scheduling mechanism* determines when it is time to multiplex the CPU. It handles the removal of a running process from the CPU and the selection of another process based on a particular scheduling policy.
- The *scheduling policy* determines when it is time for a process to be removed from the CPU and which ready process should be allocated to the CPU next.
- A scheduler has three components: the enqueuer, the dispatcher, and the context switcher. The context switcher is different for voluntary versus involuntary process switching styles.
  - The *enqueuer* places a pointer to a process descriptor of a process that just got ready into the *ready list*.
  - The *context switcher* saves the contents of all processor registers for the process being removed from the CPU in its process descriptor. Is invoked either by the process or by the interrupt handler.
  - The *dispatcher* selects one of the several ready processes enqueued in the ready list and then allocates the CPU to the process by performing another context switch from itself to the selected process.

Non-preemptive Scheduling

- The process calls the `yield()` system call to relinquish the CPU. The `yield()` system call saves the address of the next instruction at some designated memory location and then branches to an arbitrary location.

```
yield(r,s) {
    memory[r] = PC;
    PC = memory[s];
}
```

- The address r is usually a function of the process's id. The address s is similarly related to the id of the process to which the first process yields.
- With several processes running, each could yield to the scheduler which can then yield to one of the processes it selects. This method of cooperative multiprogramming was used in some earlier operating systems.
- Identify problems with voluntary context-switching.

Preemptive Scheduling (1)

- The interrupt system enforces periodic involuntary interruption of any process using an *interval timer*.
- Examine context switches with /proc virtual filesystem. (See example wcs.sh)
- An interrupt occurs every k clock ticks, thus causing the hardware clock's controller to execute the logical equivalent of an `yield()` to invoke the interrupt handler. The interrupt handler invokes the scheduler to reschedule the CPU. Thus the scheduler is guaranteed to be invoked every k clock ticks.

Preemptive Scheduling (2)

- How to find out the clock interrupt frequency on your Linux system?
  - Start with `include/linux/sched.h` in the source code for Linux kernel and look for the keyword `HZ`.
  - Or try the following:

```
cat /boot/config-4.7.2.fc24.x86_64  | grep HZ
```

- **In-class Exercise**. Find the code for context switching in the Linux source code for an architecture of your choice. *Hint:* Search for `switch_to` using grep -r in the architecture specific code. For example `arch/x86` in the kernel source tree.

Kernel Timer Interval

- In Linux, the interval timer tick rate is defined as a constant `HZ` in `<asm/param.h>`. A generic value is 100 HZ that was used for a long time in various OSes. In Linux, this value can be configured.

- **In-class Exercise**. What are the advantages and disadvantages of a larger HZ?
- **A Tickless OS!** Most OSs have used regular timers for the last forty years. But it is possible to not have regular timers but have the kernel set the interval timer dynamically in accordance with pending timers. (Added in Linux kernel version 2.6.21)
- Read Documentation/timers/NO_HZ.txt in the Linux source code for more details on the tickless Linux kernel.

Scheduling Strategies

- Scheduling strategy depends on the goals of the particular operating system. Compare a real-time system versus a time sharing system.
- Other criteria could be the priorities of processes, maximizing throughput, minimizing turnaround time, minimizing response time, resource utilization, fairness etc.
- The *priority* for a process determines the order in which the dispatcher will select a ready process to execute when the CPU becomes available.
- For systems with interval timers, each process gets to run in units of *time quantums* or *time slices*. The time slice length may be less than the interval time if the process yields a CPU early because of blocking for I/O or for other reasons.
- Given a *static* set of processes, a preemptive scheduler, and a goal for scheduling, an *optimal* schedule can be calculated by considering various orderings. However there are several problems with this method.

Scheduling Terminology

Service Time $\tau(p_i)$ The amount of time a process needs to be in the running state before it is completed.

Wait Time $W(p_i)$ (a.k.a. Response Time) The time a process spends waiting in the ready state before its first transition to the running state.

Turnaround Time $T_{trnd}(p_i)$ The amount of time between a process enters the ready state and the moment the process exits the running state for the last time.

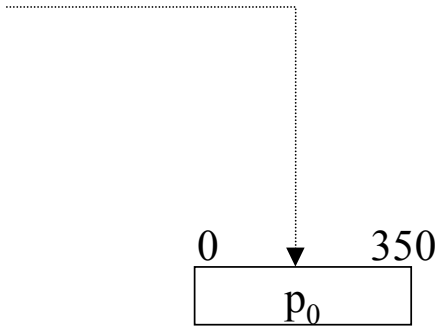
The *throughput rate* (in jobs per some unit of time) is the inverse of the average turnaround time.

Non-Preemptive Strategies

- First-Come First-Served (FCFS)
- Shortest Job Next (SJN) (a.k.a. Shortest Job First (SJF))
- Priority Scheduling
- Deadline Scheduling (for real-time systems)

First-Come-First-Served

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

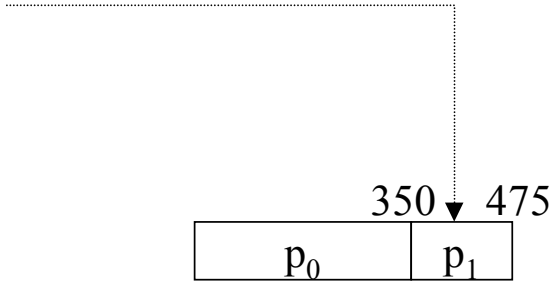


$T_{TRnd}(p_0) = \tau(p_0) = 350$

$W(p_0) = 0$

First-Come-First-Served

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_0) = \tau(p_0) = 350$

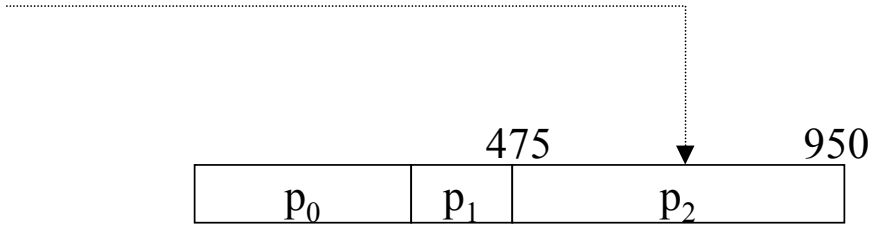
$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$

$W(p_0) = 0$

$W(p_1) = T_{TRnd}(p_0) = 350$

First-Come-First-Served

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

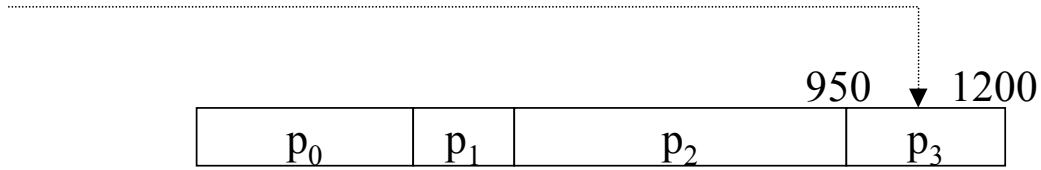


$T_{TRnd}(p_0) = \tau(p_0) = 350$
$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$
$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$

$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$
$W(p_2) = T_{TRnd}(p_1) = 475$

First-Come-First-Served

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

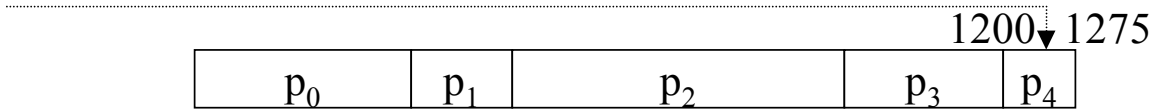


$T_{TRnd}(p_0) = \tau(p_0) = 350$
$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$
$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$
$T_{TRnd}(p_3) = (\tau(p_3) + T_{TRnd}(p_2)) = 250+950 = 1200$

$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$
$W(p_2) = T_{TRnd}(p_1) = 475$
$W(p_3) = T_{TRnd}(p_2) = 950$

First-Come-First-Served

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



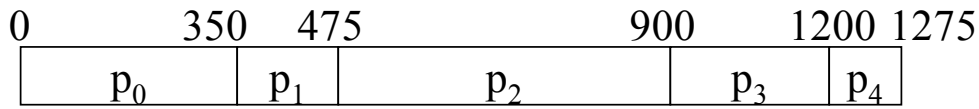
$T_{TRnd}(p_0) = \tau(p_0) = 350$
$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$
$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$
$T_{TRnd}(p_3) = (\tau(p_3) + T_{TRnd}(p_2)) = 250+950 = 1200$
$T_{TRnd}(p_4) = (\tau(p_4) + T_{TRnd}(p_3)) = 75+1200 = 1275$

$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$
$W(p_2) = T_{TRnd}(p_1) = 475$
$W(p_3) = T_{TRnd}(p_2) = 950$
$W(p_4) = T_{TRnd}(p_3) = 1200$

FCFS Average Wait Time

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

- Easy to implement
- Ignores service time, etc
- Not a great performer



$T_{TRnd}(p_0) = \tau(p_0) = 350$
$T_{TRnd}(p_1) = (\tau(p_1) + T_{TRnd}(p_0)) = 125+350 = 475$
$T_{TRnd}(p_2) = (\tau(p_2) + T_{TRnd}(p_1)) = 475+475 = 950$
$T_{TRnd}(p_3) = (\tau(p_3) + T_{TRnd}(p_2)) = 250+950 = 1200$
$T_{TRnd}(p_4) = (\tau(p_4) + T_{TRnd}(p_3)) = 75+1200 = 1275$

$W(p_0) = 0$
$W(p_1) = T_{TRnd}(p_0) = 350$
$W(p_2) = T_{TRnd}(p_1) = 475$
$W(p_3) = T_{TRnd}(p_2) = 950$
$W(p_4) = T_{TRnd}(p_3) = 1200$

$$W_{avg} = (0+350+475+950+1200)/5 = 2974/5 = 595$$

Shortest Job Next

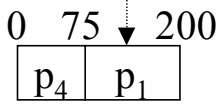
i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

$T_{TRnd}(p_4) = \tau(p_4) = 75$

$W(p_4) = 0$

Shortest Job Next

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

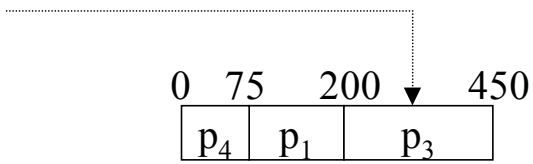


$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_4) = 125+75 = 200$ $W(p_1) = 75$

$T_{TRnd}(p_4) = \tau(p_4) = 75$ $W(p_4) = 0$

Shortest Job Next

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



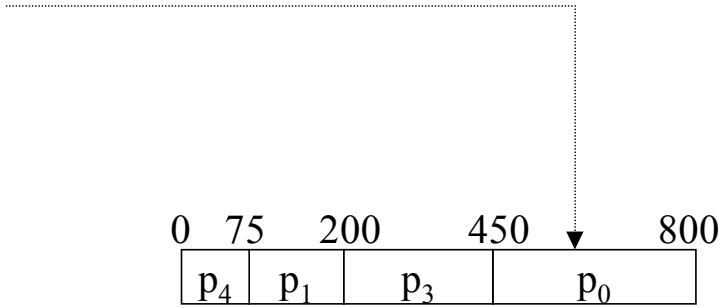
$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_4) = 125+75 = 200$ $W(p_1) = 75$

$T_{TRnd}(p_3) = \tau(p_3)+\tau(p_1)+\tau(p_4) = 250+125+75 = 450$ $W(p_3) = 200$

$T_{TRnd}(p_4) = \tau(p_4) = 75$ $W(p_4) = 0$

Shortest Job Next

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_0) = \tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 350+250+125+75 = 800$ $W(p_0) = 450$

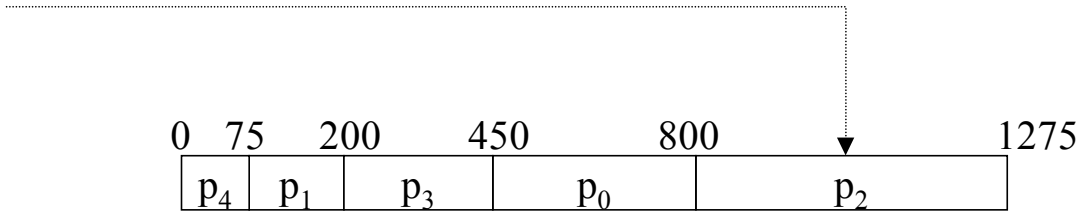
$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_4) = 125+75 = 200$ $W(p_1) = 75$

$T_{TRnd}(p_3) = \tau(p_3)+\tau(p_1)+\tau(p_4) = 250+125+75 = 450$ $W(p_3) = 200$

$T_{TRnd}(p_4) = \tau(p_4) = 75$ $W(p_4) = 0$

Shortest Job Next

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_0) = \tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 350+250+125+75 = 800$ $W(p_0) = 450$

$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_4) = 125+75 = 200$ $W(p_1) = 75$

$T_{TRnd}(p_2) = \tau(p_2)+\tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 475+350+250+125+75 = 1275$ $W(p_2) = 800$

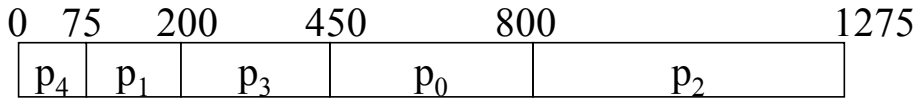
$T_{TRnd}(p_3) = \tau(p_3)+\tau(p_1)+\tau(p_4) = 250+125+75 = 450$ $W(p_3) = 200$

$T_{TRnd}(p_4) = \tau(p_4) = 75$ $W(p_4) = 0$

Shortest Job Next

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

- Minimizes wait time
- May starve large jobs
- Must know service times



$T_{TRnd}(p_0) = \tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 350+250+125+75 = 800$ $W(p_0) = 450$

$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_4) = 125+75 = 200$ $W(p_1) = 75$

$T_{TRnd}(p_2) = \tau(p_2)+\tau(p_0)+\tau(p_3)+\tau(p_1)+\tau(p_4) = 475+350+250+125+75 = 1275$ $W(p_2) = 800$

$T_{TRnd}(p_3) = \tau(p_3)+\tau(p_1)+\tau(p_4) = 250+125+75 = 450$ $W(p_3) = 200$

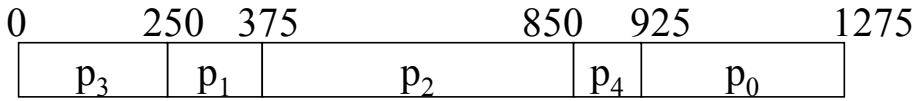
$T_{TRnd}(p_4) = \tau(p_4) = 75$ $W(p_4) = 0$

$$W_{avg} = (450+75+800+200+0)/5 = 1525/5 = 305$$

Priority Scheduling

i	$\tau(p_i)$	Pri
0	350	5
1	125	2
2	475	3
3	250	1
4	75	4

- Reflects importance of external use
- May cause starvation
- Can address starvation with aging



$T_{TRnd}(p_0) = \tau(p_0)+\tau(p_4)+\tau(p_2)+\tau(p_1))+\tau(p_3) = 350+75+475+125+250 = 1275$

$T_{TRnd}(p_1) = \tau(p_1)+\tau(p_3) = 125+250 = 375$

$T_{TRnd}(p_2) = \tau(p_2)+\tau(p_1)+\tau(p_3) = 475+125+250 = 850$

$T_{TRnd}(p_3) = \tau(p_3) = 250$

$T_{TRnd}(p_4) = \tau(p_4)+ \tau(p_2)+ \tau(p_1)+\tau(p_3) = 75+475+125+250 = 925$

$W(p_0) = 925$
$W(p_1) = 250$
$W(p_2) = 375$

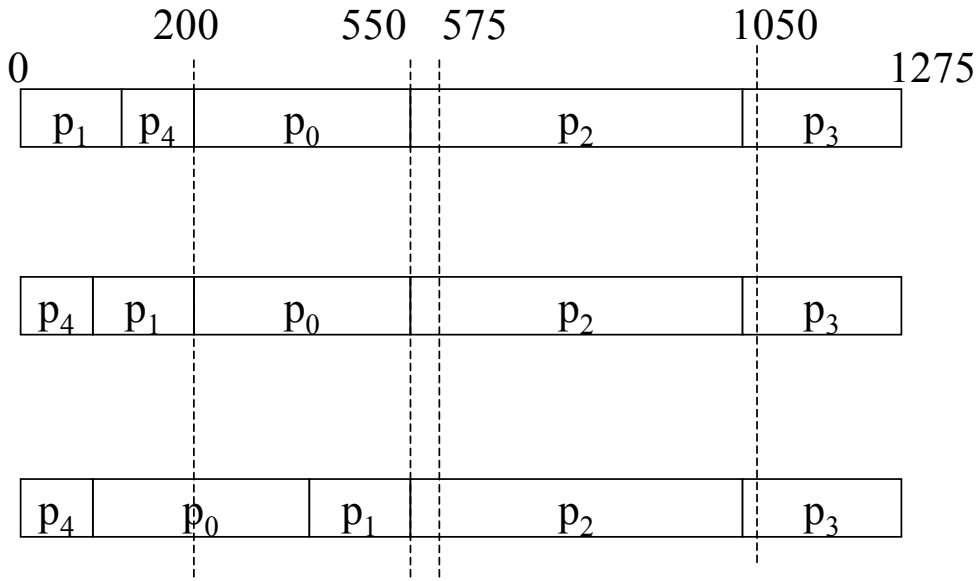
$W(p_3) = 0$
$W(p_4) = 850$

$$W_{avg} = (925+250+375+0+850)/5 = 2400/5 = 480$$

Deadline Scheduling

i	$\tau(p_i)$	Deadline
0	350	575
1	125	550
2	475	1050
3	250	(none)
4	75	200

- *Must* receive service by deadline
- May not be feasible



In-class Exercises

- Suppose you do your homework assignments in SJF-order. After all, you feel like you are making a lot of progress! What might go wrong?

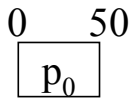
- Devise a workload where FCFS is pessimal— it does the worst possible scheduling choices— for average response time.

Preemptive Strategies

- Preemptive strategies are useful to ensure quick response times to higher priority processes or to ensure fair sharing of the CPU.
  - FCFS, SJN, Deadline scheduling have a corresponding preemptive version.
  - Priority Scheduling.
  - Round Robin (RR).
  - Multiple-level Queues.
- Cost of context-switching can be significant. If time for context-switch is c and we have n processes in RR scheduling, then each process gets q units of time every $n(q+c)$ units of real time.
- Timer interrupts go hand in hand with RR scheduling.

Round Robin (TQ=50)

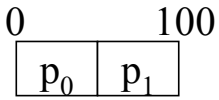
i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$W(p_0) = 0$

Round Robin (TQ=50)

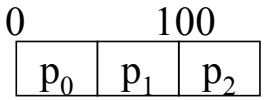
i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$W(p_0) = 0$
$W(p_1) = 50$

Round Robin (TQ=50)

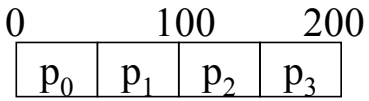
i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$

Round Robin (TQ=50)

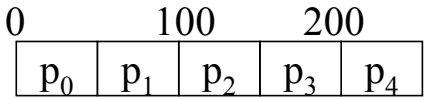
i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$

Round Robin (TQ=50)

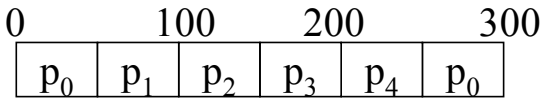
i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$

Round Robin (TQ=50)

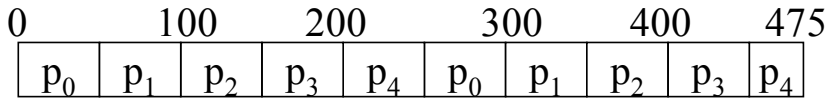
i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$

Round Robin (TQ=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

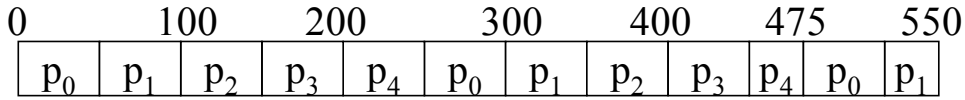


$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$

$T_{TRnd}(p_4) = 475$

Round Robin (TQ=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



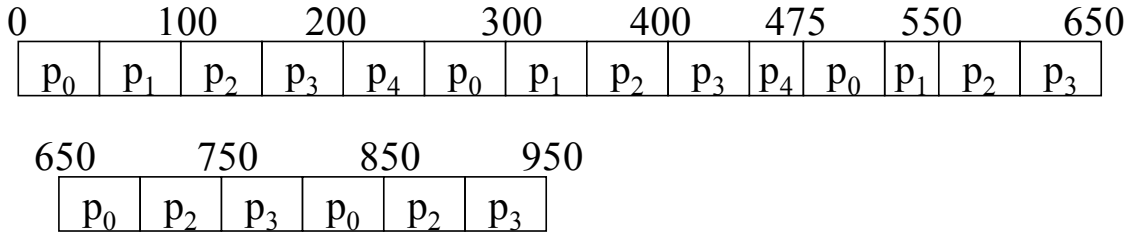
$T_{TRnd}(p_1) = 550$

$T_{TRnd}(p_4) = 475$

$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$

Round Robin (TQ=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



$T_{TRnd}(p_1) = 550$

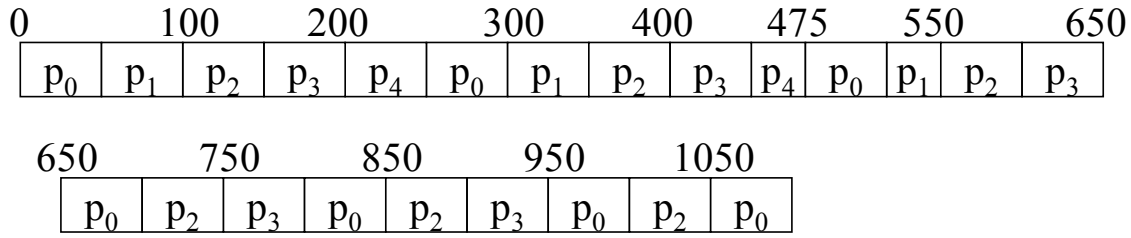
$T_{TRnd}(p_3) = 950$

$T_{TRnd}(p_4) = 475$

$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$

Round Robin (TQ=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



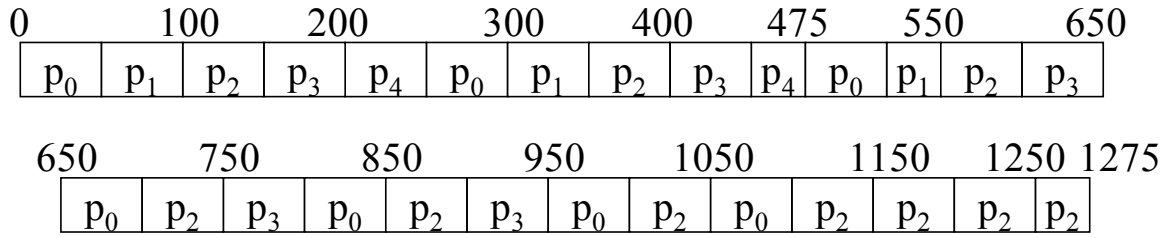
$T_{TRnd}(p_0) = 1100$
$T_{TRnd}(p_1) = 550$

$T_{TRnd}(p_3) = 950$
$T_{TRnd}(p_4) = 475$

$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$

Round Robin (TQ=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75



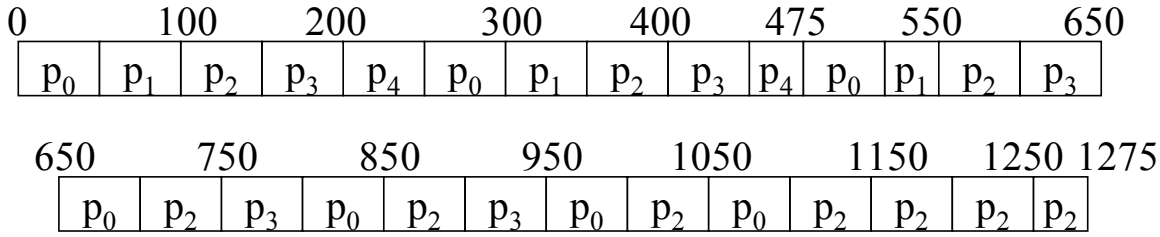
$T_{TRnd}(p_0) = 1100$
$T_{TRnd}(p_1) = 550$
$T_{TRnd}(p_2) = 1275$
$T_{TRnd}(p_3) = 950$
$T_{TRnd}(p_4) = 475$

$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$

Round Robin (TQ=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

- Equitable
- Most widely-used
- Fits naturally with interval timer



$T_{TRnd}(p_0) = 1100$
$T_{TRnd}(p_1) = 550$
$T_{TRnd}(p_2) = 1275$
$T_{TRnd}(p_3) = 950$
$T_{TRnd}(p_4) = 475$

$W(p_0) = 0$
$W(p_1) = 50$
$W(p_2) = 100$
$W(p_3) = 150$
$W(p_4) = 200$

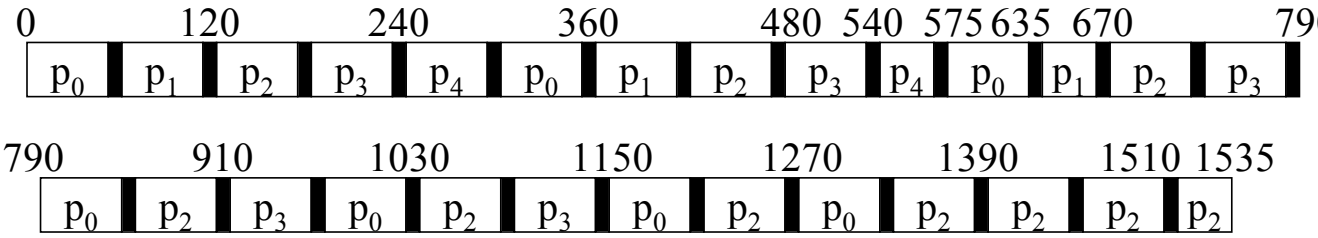
$T_{TRnd\_avg} = (1100+550+1275+950+475)/5 = 4350/5 = 870$

$W_{avg} = (0+50+100+150+200)/5 = 500/5 = 100$

RR with Overhead=10 (TQ=50)

i	$\tau(p_i)$
0	350
1	125
2	475
3	250
4	75

- Overhead must be considered



$T_{TRnd}(p_0) = 1320$ $W(p_0) = 0$
$T_{TRnd}(p_1) = 660$ $W(p_1) = 60$
$T_{TRnd}(p_2) = 1535$ $W(p_2) = 120$
$T_{TRnd}(p_3) = 1140$ $W(p_3) = 180$
$T_{TRnd}(p_4) = 565$ $W(p_4) = 240$

$T_{TRnd\_avg} = (1320+660+1535+1140+565)/5 = 5220/5 = 1044$

$W_{avg} = (0+60+120+180+240)/5 = 600/5 = 120$

In-class Exercises

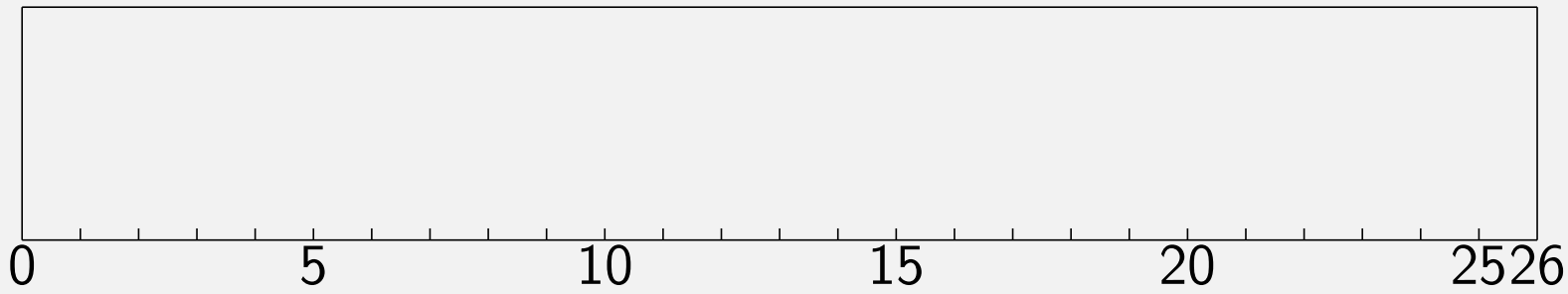
- Round Robin scheduling algorithm can be implemented without using any timer interrupts. **TRUE** **FALSE**
- Many CPU scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Some of these algorithms are related to one another (for example, FCFS is RR with infinite time quantum). What (if any) relation holds among the following pairs of algorithms? Please provide a concise (but complete) answer.
  1. Priority and SJF
  2. Priority and FCFS
  3. RR and SJF

In-class Exercises

- Five processes arrived at the same time with the following burst times (that is the CPU time for which they will run before blocking for I/O).

Process Name	Burst Time
P1	10
P2	7
P3	5
P4	1
P5	3

Show how the processes are scheduled by filling in the following Gantt chart and calculate the average response and average turnaround time if the scheduling policy is round-robin with time quantum of 3 time units.



Average Response Time =

Average Turnaround Time =

Multiple-level Feedback Queues

- One queue for higher priority foreground jobs and another for lower priority background jobs.
- Interrupt handler runs at priority 1, device drivers at priority 2, interactive compute jobs at priority 3, interactive editing jobs at priority 4, normal batch jobs at priority 5, and “long” batch jobs at priority 6. We have one queue for each priority. Processes can move up or down the multiple-level ready queues if they change their behavior.

BSD Unix Scheduling

- BSD Unix scheduling: Uses 32 queues for priorities from 0 through 31. System processes use run queues 0 through 7, and user processes use run queues from 8 through 31.
  - Each process has an external `nice` priority used to influence (but not solely) which run queue it ends up in whenever it becomes ready.
  - The default nice value is 0 and the range is from -20 (the highest) to 20 (the lowest).
  - Once every time quantum, the scheduler recomputes the priorities for all processes depending on the `nice` value and recent demand on the CPU.

Example using `nice` values

- The command `nice` allows a user to run a program with modified scheduling priority. Normal users can only set positive nice values, thus lowering the priority from the default value. Super-users can set any value from -20 (highest priority) to +19 (lowest priority).
  `nice -10 program arguments`
  where the program will run with nice value of 10 more than the default.
- The command `renice` alters the priority of a running process.
  `renice priority [[-p] pid ...] [[-g] pgrp ...] [[-u] user ...]`
  `renice +15 4751`
  where 4751 is the process id of a process.

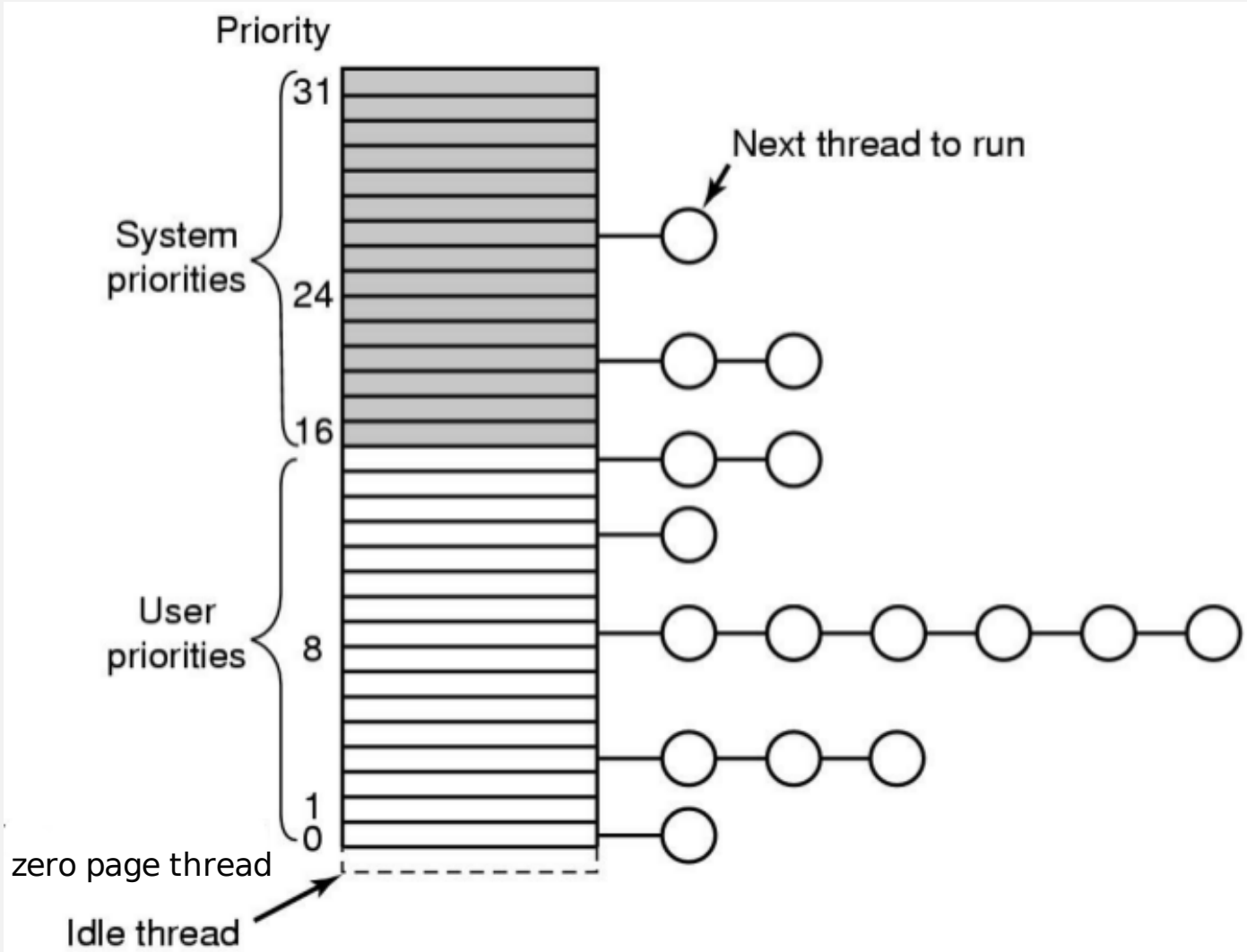
Real Time Scheduling

- A real-time system has expectations of response time as well as correctness. A hard real-time system requires a guaranteed worst-case response time, whereas a soft real-time requires expectations on response time but it isn't as critical.
- Traditional real time operating systems (RTOses) include LynxOS and QNX.
- Linux is being reworked to be more real time in response to demand from developers. One way to get soft real-time performance is to run an application under a FIFO (FCFS) scheduler. This does require superuser privileges. See example scheduling/soft-realtime-example.c.
- MS Windows also has added support for soft real-time. Try the command `start /realtime notepad` under MS Windows and check its priority in the Task Explorer versus other applications. Note that you may have to start the command console in *Administrator* mode for this example to work.
- Systems like RTLinux and RTAI (RealTime Application Interface) provide multi-microsecond worst case response time by piggy-backing onto Linux.

MS Windows Scheduler

- The scheduler is priority-driven, preemptive scheduling system modeled after the BSD scheduling system.
- The scheduler schedules at the thread granularity.
- The quantum on professional version is 2 times the clock (typically 20 msecs) and 12 times the clock (typically 120 msecs) on uniprocessor servers and various values on multiprocessor servers.

MS Windows Scheduler



MS Windows Scheduler

		Win32 process class priorities					
		Realtime	**High**	**Above Normal**	**Normal**	**Below Normal**	**Idle**
Win32 thread priorities	**Time critical**	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

O(1) Linux 2.6 Scheduler

- $O(1)$ average-time scheduler. Default quantum is 1ms.
- There are 140 queues per processor (one for each possible level of priority). The individual queues are FIFO lists.
- A bitmap indicates which queues are empty or not. Thus we can execute an efficient find-first-bit instruction over a set of 32-bit bitmaps and then take the first task off the indicated queue each time.
- Load Balancing. Every 200ms a processor checks to see if any other processor is out of balance and tries to balance the workload. If a processor is idle, it checks every 1ms so as to get started on a real task as soon as possible.

Used with modifications until version 2.6.23. Worked well on large servers but not as well on desktops with interactive processes. Developers added the *Rotating Staircase Deadline* scheduler, which used the concept of fair scheduling, which further led to the current scheduler in use (on next slide).

CFS: Completely Fair Scheduler for Linux 2.6.23+

- CFS does not directly assign a timeslice to a process. Instead CFS assigns a *proportion* of the processor. This is further affected by the nice value for the process.
- The scheduler uses *scheduler classes*, which enables different pluggable algorithms to coexist, scheduling their own type of processes. CFS is the registered scheduler for the class `SCHED_NORMAL`. CFS is defined in the file `kernel/sched/fair.c`
- Older style schedulers use constant timeslices that give a constant switching rate but variable fairness.
- Nice values are geometric instead of additive. Nice values are mapped to timeslices using a measurement decoupled from the timer. Going further, CFS does away with timeslices completely. CFS thus yields constant fairness but a variable switching rate. It does have a lower *granularity* limit of 1 ms for how long a process will run.
- *Virtual runtime* is the actual runtime normalized by the number of runnable processes. It is measured in nanoseconds. *CFS always picks the process with the smallest virtual runtime to run next*.
- CFS uses a red-black tree (self-balancing binary tree) to manage the list of runnable processes. It is referred to as *restorer* in Linux. See `include/linux/rbtree.h` and `include/linux/rbtree_augmented.h` in the kernel source code.