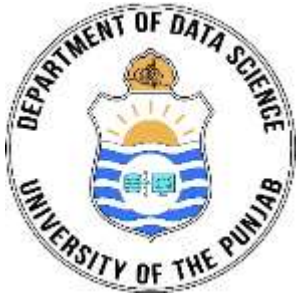


CMP325
Operating Systems
Lecture 09, 10, 11
CPU Scheduling Algorithms

Muhammad Arif Butt, PhD

Note:
Some slides and/or pictures are adapted from course text book and Lecture slides of

- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice following video lectures:

OS with Linux:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8

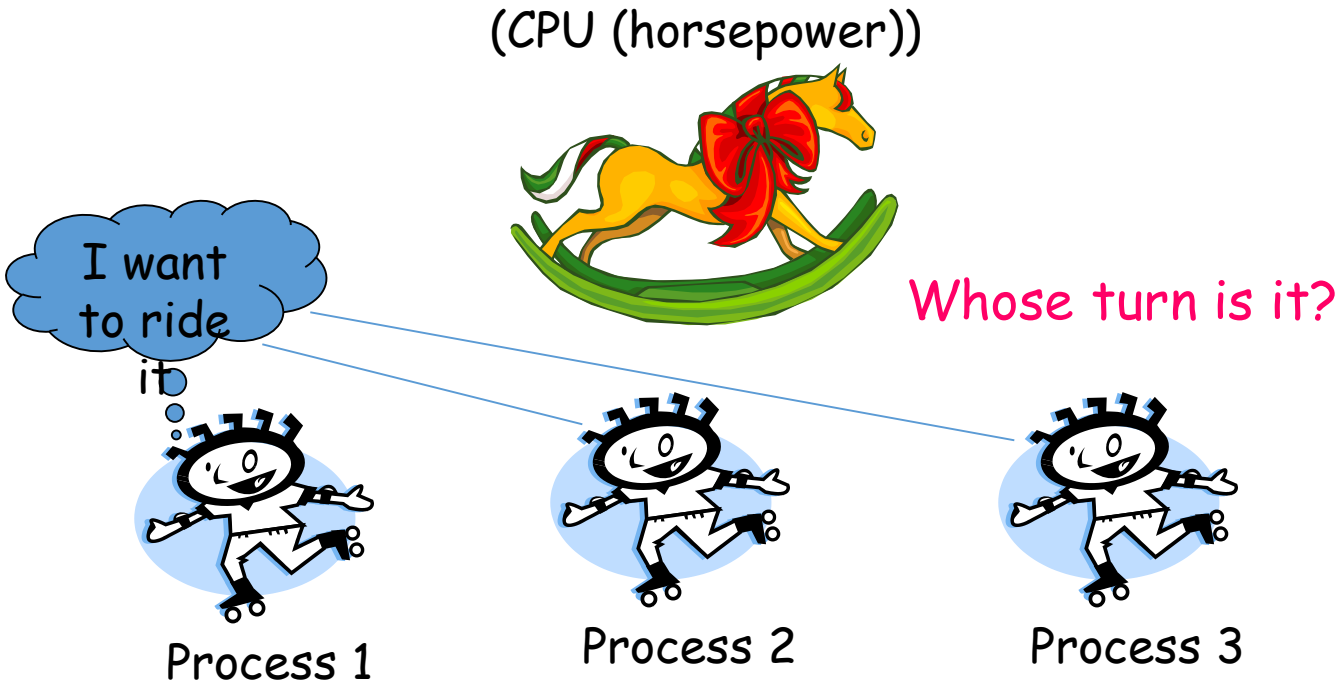
System Programming:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW

Today's Agenda

- Review of previous Lecture
- Process Scheduler & Dispatcher
- Preemptive vs Non-Preemptive Scheduler
- CPU Scheduling and Scheduling Criteria
- Scheduling Algorithms
  - FCFS Scheduling
  - SJF & SRTF Scheduling
  - Priority Scheduling
  - Round Robin Scheduling
  - MQ & MFQ Scheduling
  - Rotating Staircase Dead Line Scheduling
  - SVR3 Scheduling
- Changing Scheduling parameters on a Linux Shell

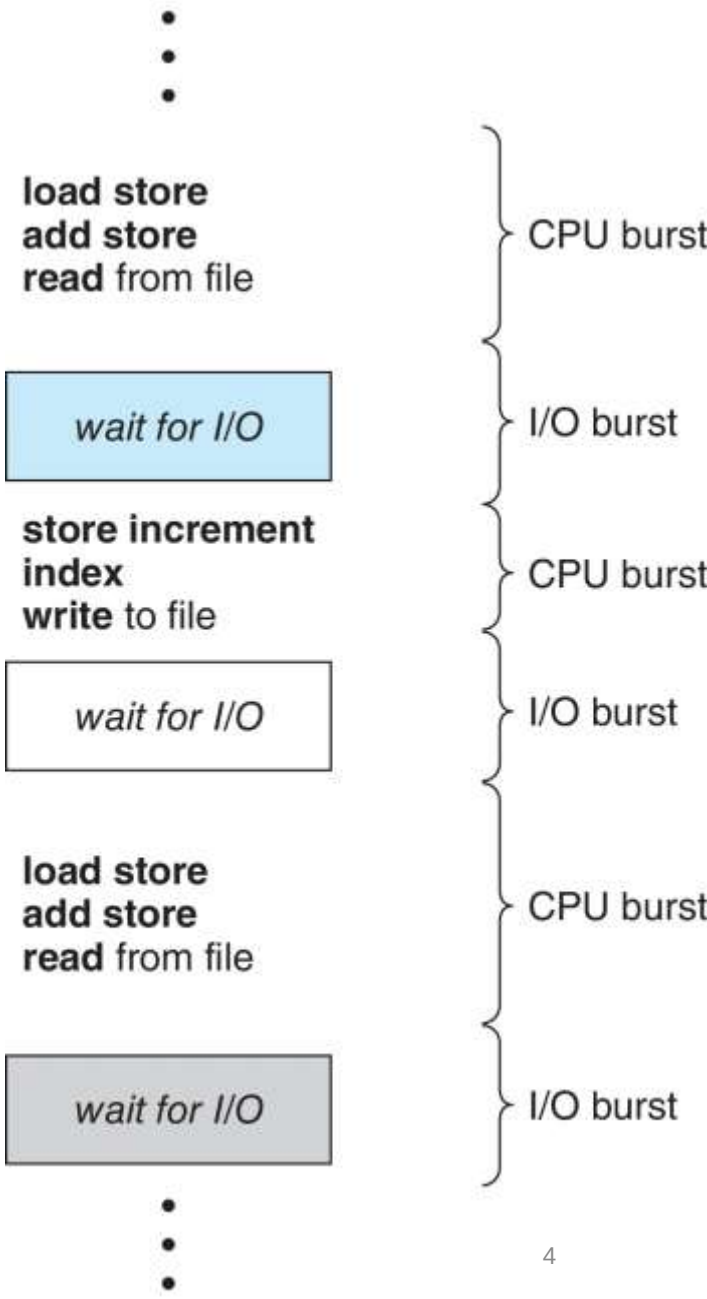
Scheduling

- Deciding which process/thread should occupy a resource (CPU, disk, etc.)



CPU – I/O Burst Cycle

- Process execution consists of a cycle of CPU execution and I/O wait
- Processes move back & forth between these two states
- A process execution begins with a CPU burst, followed by an I/O burst and then another CPU burst and so on



Scheduling

- Scheduling is a matter of managing queues to minimize queueing delay and to optimize performance in a queueing environment
- We have already discussed three types of schedulers in the previous lectures:
  - Long term scheduler
  - Medium term scheduler
  - Short term scheduler
- A **short term scheduler** is also called process scheduler or CPU scheduler. When CPU becomes idle the OS must select one of the processes from the Ready Queue to be executed by the CPU

CPU Scheduler

- **CPU Scheduler** is a kernel component that decides which process runs when and for how long.
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Terminates
  3. Switches from running to ready state. (e.g. when time slice of a process expires or an interrupt occurs)
  4. Switches from waiting to ready state. (e.g. on completion of I/O)
  5. On arrival of a new process

Dispatcher

- Dispatcher is an important component involved in CPU scheduling
- The OS code that takes the CPU away from the current process and hands it over to the newly scheduled process is known as the dispatcher
- Dispatcher gives control of the CPU to the process selected by the short-term scheduler
- Dispatcher performs following functions:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start executing another

Main Categories of Process Schedulers

In **Pre-emptive scheduling** the scheduler decides when a process is to cease running and a new process is to begin running. The duration a process executes, before it is preempted is usually fixed and is called the time slice of that process. On many modern OSs, the time slice is dynamically calculated as a function of process behavior and configurable system policy.

In **Non-preemptive Scheduling**, the running process can only lose the processor voluntarily by terminating or by requesting an I/O. OR, Once CPU given to a process it cannot be preempted until the process completes its CPU burst

Preemptive vs Non Preemptive Kernels

At any instant of time a system can either be executing in user mode (executing LOCs written by programmer) or kernel mode (executing LOCs written by the kernel developer). This can happen

A) In process context (a system call made by programmer)

B) In interrupt context

The three types of OS kernel are:

- **Preemptive Kernel**, a kernel that can be preempted both in A and B
- **Reentrant Kernel**, a kernel that can be preempted in A only
- **Nonpreemptive Kernel**, a kernel cannot be preempted

Scheduling Objectives

- Fairness (nobody cries)
- Priority (ladies first)
- Efficiency (make best use of equipment)
- Encourage good behavior (good boy/girl)
- Support heavy loads (degrade gracefully)

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Waiting time** – amount of time a process has been waiting in the ready queue
  - For Non preemptive Algos = S.T – A.T
  - For Preemptive Algos = F.T – A.T – B.T
- **Turnaround time** – amount of time to execute a particular process.
  - W.T to get into memory + W.T in Ready Q + Execution time + I/O time
  - (FinishTime – Arrival Time)
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Classification of Processes

- When speaking about process scheduling, processes are traditionally classified into three different classes:
  - **Interactive Processes:** These interact constantly with their users. When input is received, the average delay must fall between 50-150 ms, otherwise the user will find the system to be unresponsive. Typical interactive programs are command shells, text editors and graphical applications
  - **Batch Processes:** These do not need user interaction and often execute in the back ground and are often penalized by the scheduler. Typical batch programs are programming language compilers, database search engines and scientific computations
  - **Real-time Processes:** These processes should have a short guaranteed response time with a minimum variance. Typical real-time programs are multimedia applications, robot controllers, and programs that collect data from physical sensors

Classification of Processes

There are two more classifications of our interest when talking about scheduling algorithms:

- **CPU bound processes:** These processes spend more time doing computation and try to hold CPU for longer durations, e.g., image processing applications, network traffic simulators
- **I/O bound processes:** These processes spend more time doing I/O and try to hold I/O devices for longer durations, e.g., text editors, printing application like billing system

These two classifications are not mutually exclusive, a process can exhibit both behaviors. For example, a word processor (an I/O bound process) can become CPU bound when doing spell checking or macro calculations.

First Come First Serve

- ***Process that requests the CPU FIRST is allocated the CPU FIRST***
- Called "FIFO", **Non-preemptive**, Used in Batch Systems
- Real life analogy: Any ticket counter

- Implementation: FIFO queues
  - A new process enters the **tail** of the queue
  - The scheduler selects from the **head** of the queue.
- Performance Metric: **Average Waiting Time** (AWT)
- Given Parameters:
  - Burst Time (in ms), Arrival Time and Order
  - Can be generalized to processes with alternate CPU and I/O bursts: blocking process goes to queue's tail

First Come First Serve (cont…)

- Simplest CPU scheduling algorithm
- Non preemptive
- The process that requests the CPU first is allocated the CPU first
- Implemented with a FIFO queue. When a process enters the Ready Queue, its PCB is linked on to the tail of the Queue. When the CPU is free it is allocated to the process at the head of the Queue
- **Limitations**
  - FCFS favor long processes as compared to short ones. (Convoy effect)
  - FCFS tends to favor processor bound processes over I/O bound processes
  - Average waiting time is often quite long
  - FCFS is non-preemptive, so it is trouble some for time sharing systems

Convoy Effect

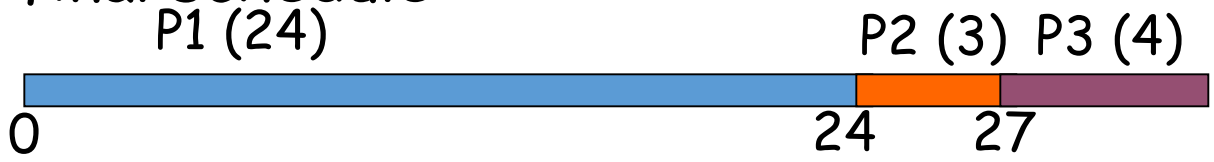
- Consider n-1 jobs in system that are I/O bound and 1 job that is CPU bound
- I/O bound jobs pass quickly through the ready queue and suspend themselves waiting for I/O
- CPU bound job arrives at head of queue and executes until complete
- I/O bound jobs rejoin ready queue and wait for CPU bound job to complete
- I/O devices idle until CPU bound job completes
- When CPU bound job completes, other processes rush to wait on I/O again
- CPU becomes idle

"A convoy effect happens when a set of processes need to use a resource for a short time, and one process holds the resource for a long time, blocking all of the other processes. Causes poor utilization of the other resources in the system"

FCFS – Example 1

Process	Duration/B.T	Order	Arrival Time
P1	24	1	0
P2	3	2	3
P3	4	3	4

The final schedule:



P1 waiting time: 0-0
P2 waiting time: 24-3
P3 waiting time: 27-4

The average waiting time:
(0+21+23)/3 = 14.667

FCFS – Example 2

- Draw the graph (Gantt chart) and compute average waiting time for the following processes using FCFS Scheduling algorithm.

Process	Arrival time	Burst Time
P1	1	16
P2	5	3
P3	6	4
P4	9	2

SJF & SRTF Scheduling…

- When the CPU is available it is assigned to the process that has the smallest next CPU burst
- If two processes have the same length next CPU bursts, FCFS scheduling is used to break the tie

Comes in two flavors

- **Shortest Job First (SJF)**
  - It's a non preemptive algorithm. When a new process arrives having a shorter next CPU burst than what is left of the currently executing process, it ***allows*** the currently running process to finish its CPU burst
- **Shortest Remaining Time First (SRTF)**
  - It's a Preemptive algorithm. When a new process arrives having a shorter next CPU burst than what is left of the currently executing process, it ***preempts*** the currently running process

SJF Example 3

Process	Duration/B.T	Order	Arrival Time
P1	6	1	0
P2	8	2	0
P3	7	3	0
P4	3	4	0

P4 waiting time: 0-0
P1 waiting time: 3-0
P3 waiting time: 9-0
P2 waiting time: 16-0

The total running time is: 24
The average waiting time (AWT):
(0+3+9+16)/4 = 7 time units

SRTF Example 4

Process	Duration	Order	Arrival Time
P1	10	1	0
P2	2	2	2

P1 waiting time: 4-2 =2
P2 waiting time: 0

The average waiting time (AWT):
(0+2)/2 = 1

Now run this using SJF!

SJF & SRTF – Example 5

Draw the graph (Gantt chart) and compute waiting time and turn around time for the following processes using **SJF** & **SRTF** Scheduling algorithm. For SJF consider no arrival time and consider all processes arrive at time 0 in sequence P1, P2, P3, P4.

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

NOTE: For pre emptive algos, we calculate Turn Around Time. Whenever a process enters Ready Queue more than once we use turn around time instead of waiting time.

- For non-emptive algo: Waiting time = Start time – Arrival time
- For preemptive algo: Waiting time = Finish time – Burst time – ArrivalTime
- Turn around time = Finish time – Arrival time

SJF & SRTF – Example 6

Draw the graph (Gantt chart) and compute waiting time and turn around time for the following processes using **SJF** & **SRTF** Scheduling algorithm.

Process	Arrival Time	Burst Time
P1	0	5
P2	1	2
P3	2	3
P4	3	1

SJF & SRTF – Example 7

Draw the graph (Gantt chart) and compute waiting time and turn around time for the following processes using **SJF** & **SRTF** Scheduling algorithm.

Process	Arrival Time	Burst Time
P1	0	9
P2	3	6
P3	6	2
P4	9	1

SJF & SRTF Scheduling

$100 QUESTION

How to compute the next CPU burst?

- This algorithm cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst
- One approach is to try to approximate. We may not *know* the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. Thus, by computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst

SJF & SRTF Scheduling…

Exponential Averaging

Estimation based on historical data

t_n = Actual length of n^{th} CPU burst

τ_n = Estimate for n^{th} CPU burst

τ_{n+1} = Estimate for $n+1^{st}$ CPU burst

α, $0 \leq \alpha \leq 1$

$$\tau_{n+1} = \alpha t_n + (1- \alpha)\, \tau_n$$

Exponential Averaging…

$$\tau_{n+1} = \alpha t_n + (1- \alpha)\, \tau_n$$

- Plugging in value for τ_n, we get

$$\tau_{n+1} = \alpha t_n + (1- \alpha)[\alpha t_{n-1} + (1- \alpha)\tau_{n-1}]$$

$$\tau_{n+1} = \alpha t_n + (1- \alpha)\alpha t_{n-1} + (1- \alpha)^2 \tau_{n-1}$$

- Again plugging in value for τ_{n-1}, we get

$$\tau_{n+1} = \alpha t_n + (1- \alpha)\alpha t_{n-1} + (1- \alpha)^2 [\alpha t_{n-2} + (1- \alpha)\tau_{n-2}]$$

$$\tau_{n+1} = \alpha t_n + (1- \alpha)\alpha t_{n-1} + (1- \alpha)^2\, \alpha t_{n-2}\, (1- \alpha)^3 \tau_{n-2}$$

- Continuing like this results in

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + \ldots + (1 - \alpha)^j\, \alpha\, t_{n-j} + \ldots \quad + (1 - \alpha)^{n+1}\, \tau_0$$

If $\alpha = 1/2$

$$\tau_{n+1} = t_n/2 + t_{n-1}/2^2 + t_{n-2}/2^3 + t_{n-3}/2^4 + \ldots$$

Exponential Averaging…

$$\tau_{n+1} = \alpha t_n + (1- \alpha)\, \tau_n$$

Lets take two extreme values of α

If $\alpha = 0$

$$\tau_{n+1} = \tau_n$$

Next CPU burst estimate will be exactly equal to previous CPU burst estimate.

If $\alpha = 1$

$$\tau_{n+1} = t_n$$

Next CPU burst estimate will be equal to previous actual CPU burst.

Exponential Averaging…

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + \ldots + (1 - \alpha)^j\, \alpha\, t_{n-j} + \ldots \qquad + (1 - \alpha)^{n+1}\, \tau_0$$

Typical value used for α is $\frac{1}{2}$. With this value, our $(n+1)^{st}$ estimate is

$$\tau_{n+1} = t_n/2 + t_{n-1}/2^2 + t_{n-2}/2^3 + t_{n-3}/2^4 + \ldots$$

Note that as we move back in time we are giving less weight-age to the previous CPU bursts. Older histories are given exponentially less weight-age

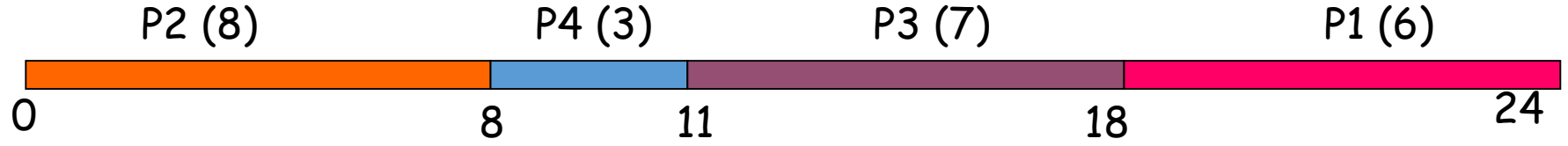
Priority Scheduling

- A priority number (integer) is associated with each process and the CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
- A priority scheduling algorithm can be preemptive or non preemptive
  - A **Preemptive** priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. E.g. SRTF is a priority scheduling algorithm where priority is the predicted next CPU burst time
  - A **Non preemptive** priority scheduling algorithm will simply put the new process at the tail of the Ready Queue
- Problem ≡ Starvation / Indefinite Blocking; i.e. Low priority processes may never execute
- Solution ≡ *Aging;* It is a technique of gradually increasing the priority of processes that wait in the system for long time

Priority Scheduling – Example 8

Lower priority # == More important

Process	Duration	Priority #	Arrival Time
P1	6	4	0
P2	8	1	0
P3	7	3	0
P4	3	2	0



P2 waiting time: 0
P4 waiting time: 8
P3 waiting time: 11
P1 waiting time: 18

The average waiting time (AWT):
(0+8+11+18)/4 = 9.25
(worse than SJF's)

Round Robin (RR) Scheduling

- RR is preemptive and designed especially for time sharing systems.
- A clock interrupt is generated at periodic intervals called time quantum or time slice.
- To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is greater than 1 time quantum; the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process from the ready queue.

Round Robin (RR) Scheduling…

- The performance of the RR algorithm depends heavily on the size of the time quantum. Principal design issue in the length of the time quantum to be used are:
  - If the time quantum is very large (larger than the largest CPU burst of any process), the RR policy become the FCFS policy.
  - If the time quantum is very short, the RR approach is called processor sharing. Short processes will move through the system relatively quickly.

Round Robin (RR) Scheduling (cont..)

Limitation

- Processor bound processes tend to receive an unfair portion of processor time, which result in poor performance of I/O bound processes.

Effect of context switching on the performance of RR scheduling

- Let us assume that we have only 1 process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process will require 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then 9 context switches will occur, slowing the execution of the process accordingly.
- Thus, the time quantum should be large with respect to the context switch time. If the context switch time is approximately 5 percent of the time quantum, then about 5 percent of the CPU time will be spent in context switching.

RR – Example 9

Draw the graph (Gantt chart) and compute turn around time for the following processes using RR Scheduling algorithm. Consider a time slice of 4 sec.

Process	Arrival Time	Burst Time
P1	0	16
P2	1	3
P3	2	3

RR – Example 10

Draw the graph (Gantt chart) and compute turn around time for the following processes using RR Scheduling algorithm. Consider a time slice of 3 sec

Process	Arrival Time	Burst Time
P1	0	8
P2	2	4
P3	3	2
P4	6	5

NOTE: Priority of placing a process in Ready Queue

- New Process.
- Running Process.

RR with I/O – Example 11

Draw the graph (Gantt chart) and compute turn around time for the following processes using **RR** Scheduling algorithm. Consider a time slice of 3 sec. Every even number process perform I/O after every 2 sec of its running life. I/O takes 10 seconds.

Process	Arrival Time	Burst Time
P1	0	8
P2	2	3
P3	3	5
P4	6	4

NOTE: Priority of placing a process in Ready Queue

- New Process.
- Blocked Process or I/O process.
- Running Process.

RR with I/O – Example 12

Draw the graph (Gantt chart) and compute turn around time for the following processes using RR Scheduling algorithm. Consider a time slice of 3 sec. Every odd number process perform I/O after every 2 sec of its running life. I/O takes 10 seconds.

Process	Arrival Time	Burst Time
P1	0	8
P2	2	3
P3	3	5
P4	6	4

RR with I/O – Example 13

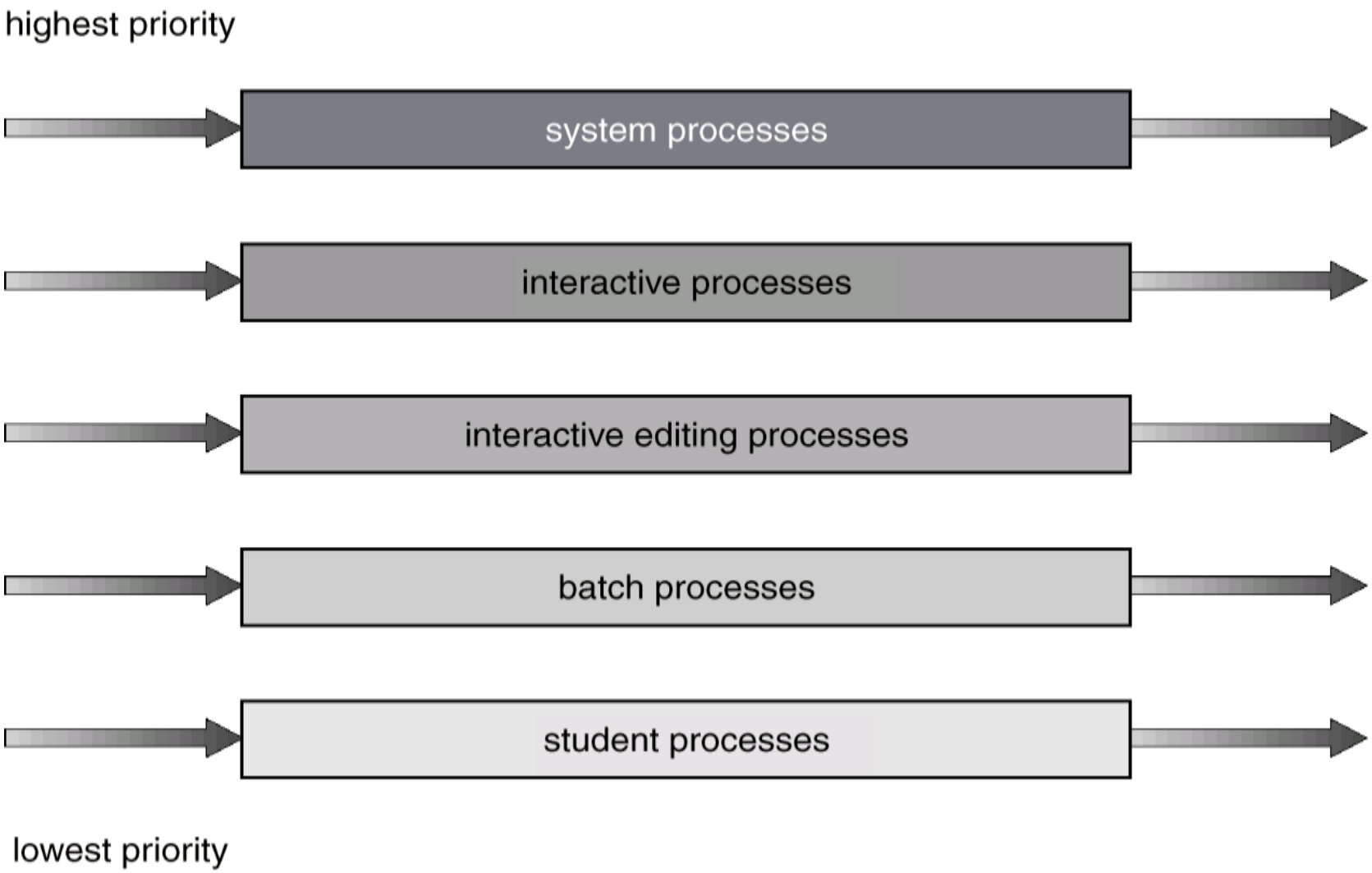
Schedule the following processes using RR. The processes P1, P2 and P3 have arrived at time units 0, 1 and 2 respectively. The process P1 demands CPU for 3 time quantum before going for I/O for 6 time quantum, then again demand CPU for 2 quantum and then goes for I/O for 4 quantum and finally demand CPU for 4 CPU quantum before it terminate. Assume RR time slice of 4 time units.

Processes	CPU Burst	I/O Burst	CPU Burst	I/O Burst	CPU Burst
P1	3	6	2	4	4
P2	5	4	3		
P3	6	8	5	7	2

Multilevel Queue Scheduling

- A multilevel queue scheduling algorithm partitions the Ready Queue into several separate queues:
  - Foreground (interactive)
  - Background (batch)
- Processes are permanently assigned to a queue on entry to the system (based on some property of the process, e.g. memory size, process priority, process type).
- Processes do not move between queues.
- Each queue has its own scheduling algorithm,
  - Foreground – RR
  - Background – FCFS
- More over there must be scheduling between the queues. E.g. foreground queue may have absolute priority over back ground queue.

Multilevel Queue Scheduling



MQ Scheduling – Example 14

Draw the graph (Gantt chart) for the following processes using **MQ** Scheduling algorithm.

If (CPU time < 4) **then** Q1 (FCFS)

else **if** (4 <= CPU time < 7) **then** Q2 (FCFS)

else Q3 (FCFS)

Process	Arrival Time	Burst Time
P1	0	8
P2	2	2
P3	3	4
P4	5	6
P5	7	9
P6	9	3
P7	10	5

MQ Scheduling – Example 15

Draw the graph (Gantt chart) for the following processes using **MQ** Scheduling algorithm.

If (CPU time < 4) **then** Q1 (FCFS)

else **if** (4 <= CPU time < 7) **then** Q2 (RR – 3sec)

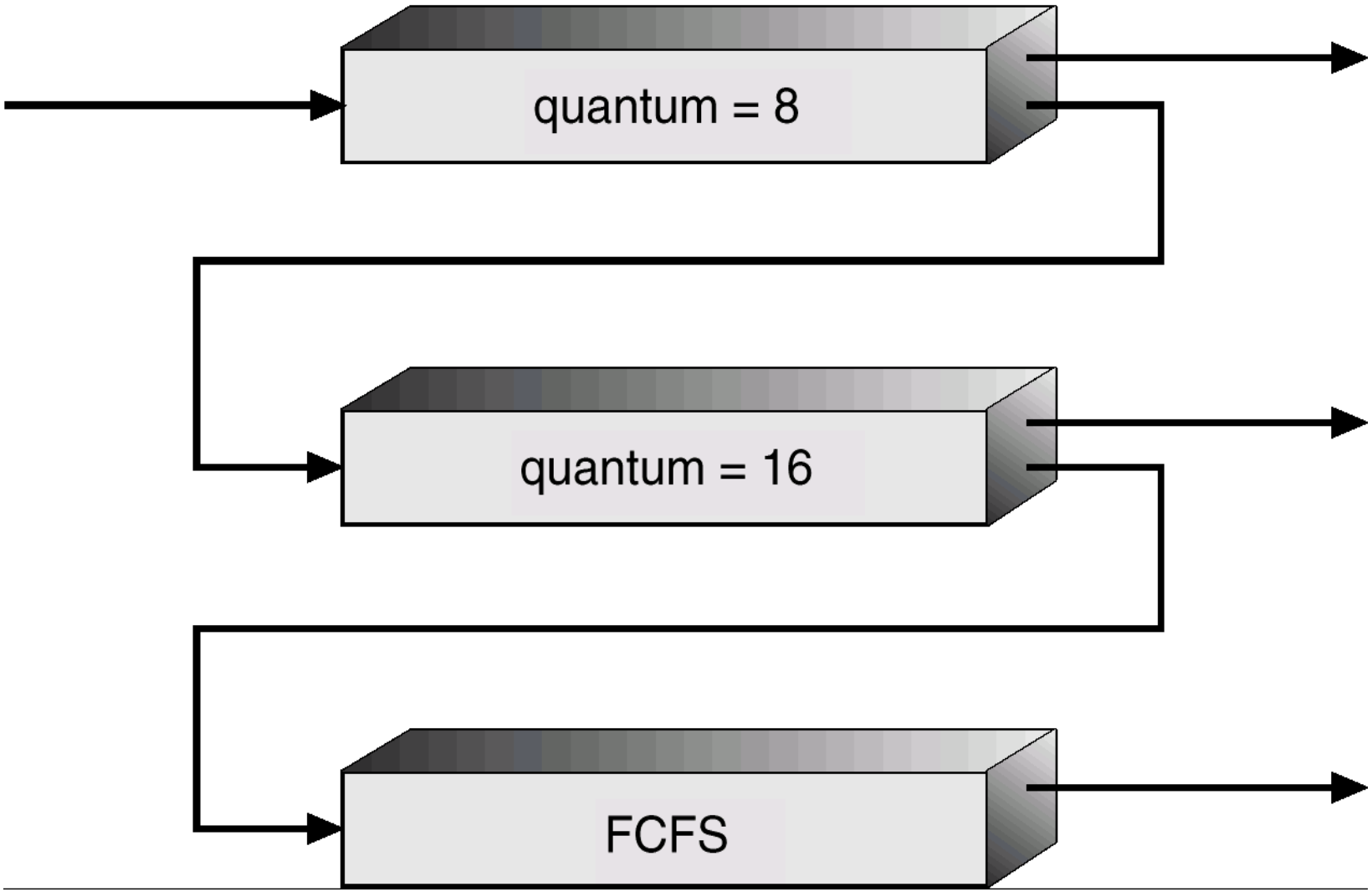
else Q3 (FCFS)

Process	Arrival Time	Burst Time
P1	0	8
P2	2	2
P3	3	4
P4	5	6
P5	7	9
P6	9	3
P7	10	5

Multilevel Feedback Queue (MFQ) Scheduling

- Multilevel feed back queue scheduling allows a process to move between various queues.
- Processes that uses too much CPU time is moved to a lower priority queue thus leaving the interactive and I/O bound processes in the higher priority queue.
- Similarly Aging can be done to prevent starvation i.e. the processes that waits too long in the lower priority queue are moved to a higher priority queue.
- Parameters of a Multilevel-feedback-queue scheduler are:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service

Multilevel Feedback Queues



MFQ Scheduling – Example 16

Draw the graph (Gantt chart) for the following processes using **MFQ** Scheduling algorithm.

Q1 - RR - 8 sec

Q2 - RR - 16 sec

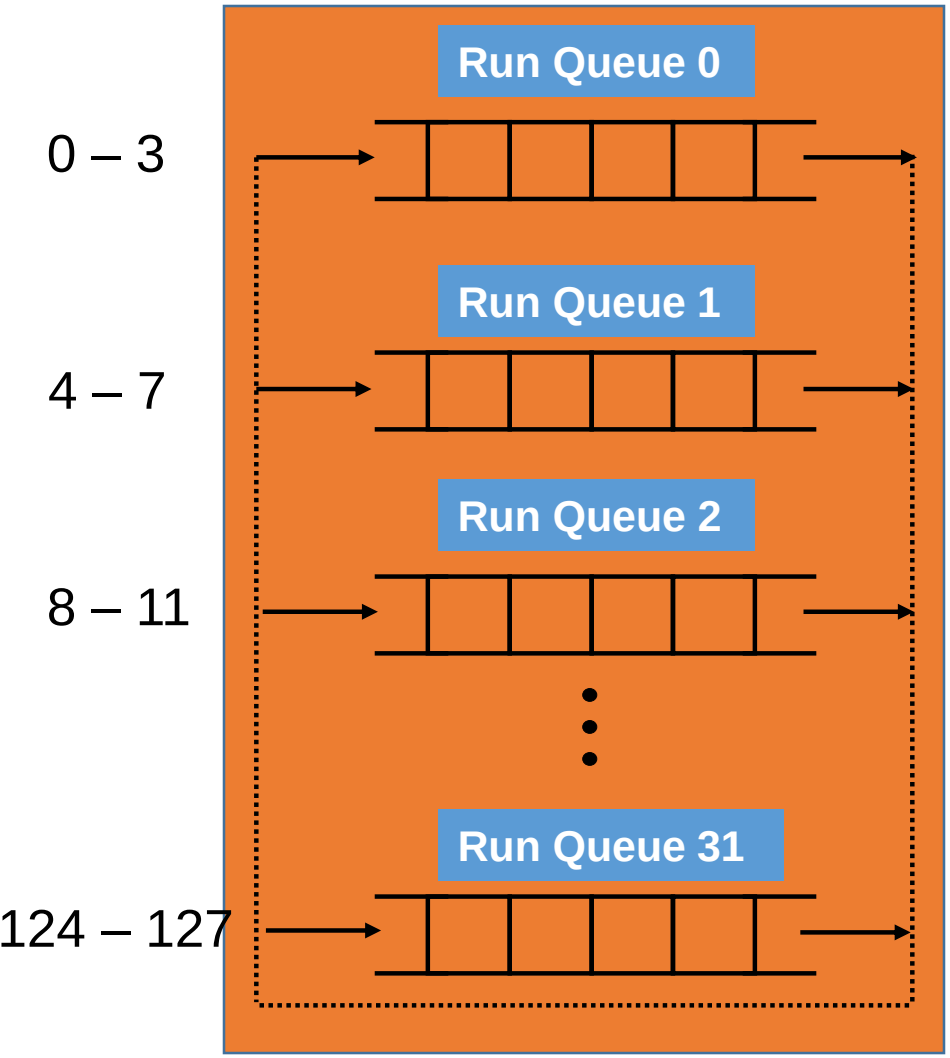
Q3 - FCFS

Process	Arrival Time	Burst Time
P1	1	10
P2	5	8
P3	8	22
P4	12	16
P5	15	30

Rotating Staircase Dead Line Scheduler (RSDL)

- This scheduler uses multilevel queues. Each process has a quota of time it is allowed to execute at a particular level. Once a process has consumed its quota at a given priority level, it is dropped down to the next priority queue and given a new quota. As a process move down the stair case it has to contend with the lower priority processes who are patiently waiting on the lower levels. Each priority level has also a quota of its own. Once highest priority level has used its quota, all processes running at that level are pushed down to next lower level regardless of whether the process have consumed their individual quotas or not. So there is no starvation in this scheduler

UNIX SVR3 Scheduling Algorithm



- 128 Priority values
  - 0–49: Kernel
  - 50–127: User level programs

$$usrpri_j(i) = Base_j + cpu_j(i) + nice_j$$

Where $Base_j = 50$
$cpu_j(i) = DR * cpu_j(i-1)$
$nice_j$ = -20 to +19

UNIX SVR3 Scheduling Algorithm (…)

- The Traditional UNIX scheduler employs thirty-two multi-level feed back queues implementing round robin algorithm with a fixed time quantum of 100 ms
- There are total of 128 different priority values, 0 to 49 for kernel processes and rest for user level programs. Four priority values are mapped on each queue
- A process enters in an appropriate queue based on its priority value, which is computed by a formula and is recomputed every second (not inherited)
- When it comes to scheduling the process in the smallest priority number queue is selected. After every second, the priorities of all the processes are recalculated and they are promoted or demoted in the queues accordingly

UNIX SVR3 Scheduling Algorithm (…)

- The priority of a process is calculated as the sum of three terms as shown in the formula:

$$\mathbf{usrpri_j(i) = Base_j + CPU_j(i) + nice_j}$$

- **$Base_j$** means base value for process j, which differentiate between user and kernel priorities. For user processes its value is 50-127, while for kernel processes its value is 0-49
- ***$CPU_j(i)$***, means the CPU utilization of process j through interval i. It is calculated by multiplying the previous cpu utilization with a decay rate. In SVR3 the decay rate is $\frac{1}{2}$
- The nice value (a per process attribute) is a user controllable adjustment factor. It is called nice because a process increases its nice value and in turn reduces its priority and show nice behavior to other processes by giving them the opportunity to run. A user can change the nice value of a process by `nice(1)` and `renice(1)` commands. The nice values range from -20 to 19 with a default value of 0

UNIX SVR3 Scheduling Algorithm (…)

Limitations

- With large number of processes, overhead of re-computing process priorities every second is very high

- Since the kernel itself is non-preemptive, high priority processes may have to wait for low priority processes executing in kernel mode

Changing Priority of Processes: nice(1)

- When a process is executed (e.g., by the shell), it inherits the nice value of its parent. (default nice value is zero)
- If you want to execute a process with a different nice value, use following command

```
$ nice -val cmd [args]
```

- You need to be root if you want to run a process with a nice value of less than zero
- If you want to change the nnice value of an already runninng process, use following command

```
$ renice val <PID>
```

Use above commands to and check the impact of nice value on the priority of a process using the ps -l command

CPU Affinity

- In a multi-processor / multi-core system, when a process is rescheduled, it does not necessarily run on the same CPU on which it ran previously
- If a process moves from one CPU to the other, the cache of the first CPU must be invalidated and the cache of the second CPU must be populated with the process data
- To achieve performance gains in cache optimization the concept of CPU affinity is introduced in Linux Kernel
- Scheduler in today's Linux Kernel (CFS) tries to ensure **soft CPU affinity**, i.e., tries to run the task on the same CPU on which it ran previously
- We can ensure **hard CPU affinity** using a per process attribute cpus_allowed, which is a 32 bit mask having one bit per CPU or core in the system
- When a process is created it inherits the affinity mask of its parent, later a process can change it using schedtool(1)

Schedtool

- The `schedtool` is an interface to Linux's scheduler used to query and change all CPU scheduling parameters policies under Linux
- Use `sudo apt-get install schedtool` command to install it on your machine

Schedtool

$ schedtool

```
get/set scheduling policies - v1.3.0, GPL'd, NO WARRANTY
USAGE: schedtool PIDS                          - query PIDS
       schedtool [OPTIONS] PIDS                - set PIDS
       schedtool [OPTIONS] -e COMMAND          - exec COMMAND
set scheduling policies:
    -N                    for SCHED_NORMAL
    -F -p PRIO            for SCHED_FIFO       only as root
    -R -p PRIO            for SCHED_RR         only as root
    -B                    for SCHED_BATCH
    -I -p PRIO            for SCHED_ISO
    -D                    for SCHED_IDLEPRIO
    -M POLICY             for manual mode; raw number for POLICY
    -p STATIC_PRIORITY    usually 1-99; only for FIFO or RR
                          higher numbers means higher priority
    -n NICE_LEVEL         set niceness to NICE_LEVEL
    -a AFFINITY_MASK      set CPU-affinity to bitmask or list

    -e COMMAND [ARGS]     start COMMAND with specified policy/priority
    -r                    display priority min/max for each policy
    -v                    be verbose
```

Schedtool

Query Processes:

```
$ schedtool 8991

PID  8991: PRIO   0, POLICY N: SCHED_NORMAL  ,
NICE   0,AFFINITY 0x1

$ schedtool -r

N: SCHED_NORMAL  : prio_min 0, prio_max 0
F: SCHED_FIFO    : prio_min 1, prio_max 99
R: SCHED_RR      : prio_min 1, prio_max 99
B: SCHED_BATCH   : prio_min 0, prio_max 0
I: SCHED_ISO     : policy not implemented
D: SCHED_IDLEPRIO: prio_min 0, prio_max 0
```

Schedtool

Set Processes:

(1) Scheduling Policy

```
$ schedtool -<NFRBID> <PIDs>
```

(2) Nice Level

```
$ schedtool -n 10 <PIDs>
```

(3) Static Priority

```
$ schedtool -p 20 <PIDs>
```

(4) CPU Aff nity

```
$ schedtool -a 0x3 <PIDs>
```

Schedtool

- We can execute a process with specific CPU-affinity using `schedtool`. The value used with -a switch is used as a simple bitmask, the bit set to 1 denoting the PID may run on that CPU, the bit unset (0) denoting it MUST NOT.
- This will execute `command1` on *CPU0* only

```
$  schedtool -a 0x1 command1
```

- This will execute `command1` on *CPU0 and CPU3* only

```
$  schedtool -a 0x5 command1
```

- This will execute `command1` on *CPU3* only

```
$  schedtool -a 0x4 command1
```

- This will execute `command1` on *CPU0 to CPU3* only

```
$  schedtool -a 0xF command1
```

Schedtool

Execute a new Processes:

The following will execute “command -arg1 -arg2 f le” with the mentioned parameters

```
$ schedtool -<SCHED_PARAMETERS> -e cmd -arg1 -arg2 file
```

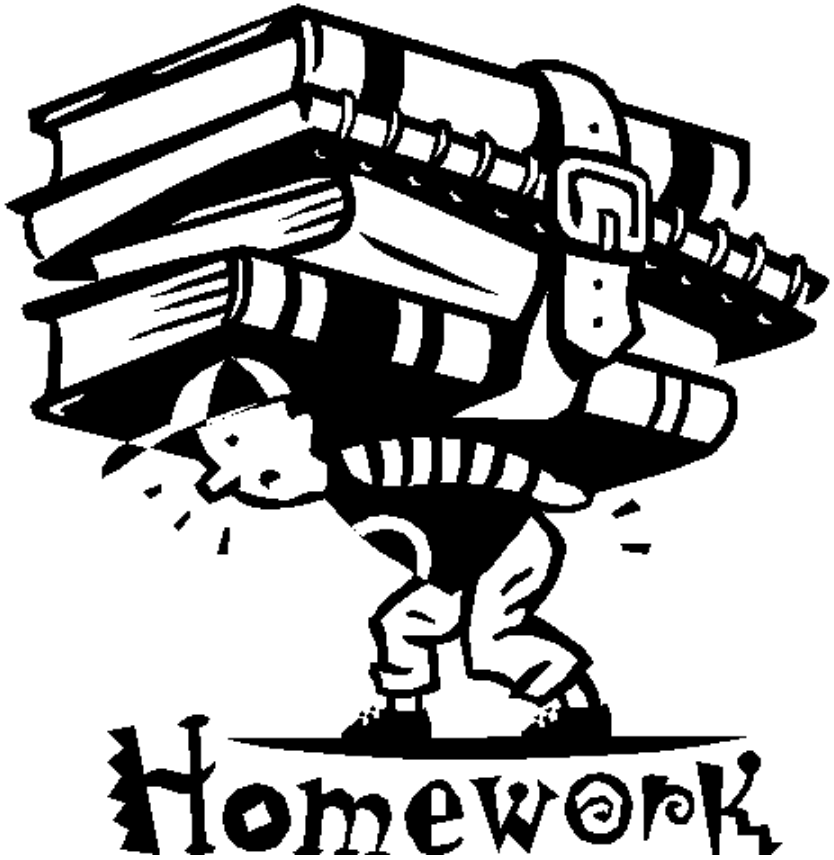
The following example will execute `mplayer file.avi` with SCHED_RR, static priority 50 and aff nity 0x2 (run only on CPU1)

```
$ schedtool -R -p 50 -a 0x2 mplayer file.avi
```

Evaluation of Scheduling Algorithms

- Deterministic Modeling
- Queuing Model
- Simulation
- Implementation

COMING SOON



SCHEDULING ALGORITHMS

We're done for now, but Todo's for you after this lecture…

- Go through the slides and Book Sections: 5.1 to 5.3, 5.7
- Practice sample problems discussed and solved in class.
- Assignment for Scheduling Algorithms will be uploaded soon, start doing it at your earliest to get it done by the given dead line.

If you have problems visit me in counseling hours. . . .