Scheduling Algorithms

Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems

Uppsala University

Spring 2008

Outline

Recall

Process Management

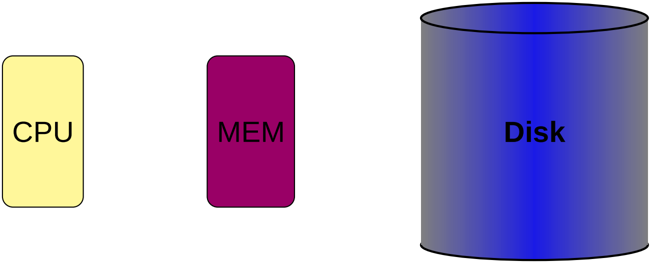
Memory Management

Storage Management

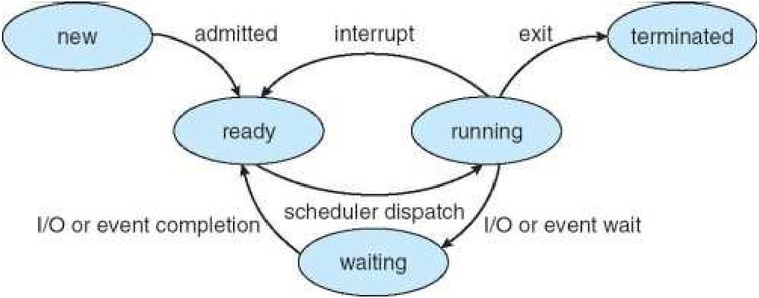
Protection and Security

Not in the class: Networking, GUI, Device Drivers

Hardware

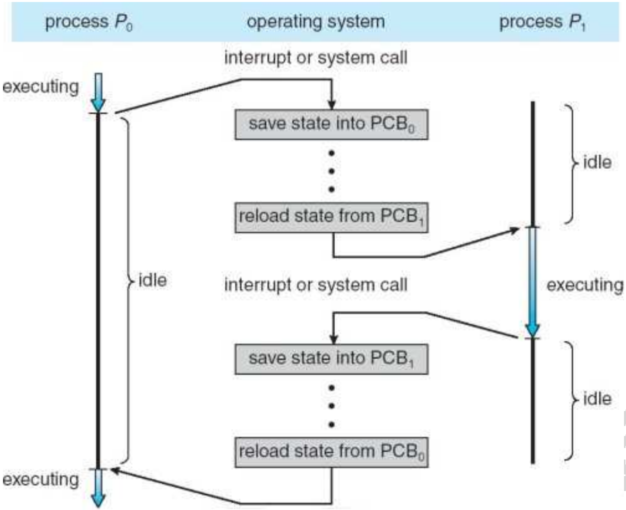


Recall...States & Transitions



- Interrupts
- Traps (software errors, illegal instructions)
- System calls

What happens at a transition?



Process Control Block & Queues

PCB
process state
process ID (number)
PC
Registers
memory information
open files
⋮ other resources

Job Queue

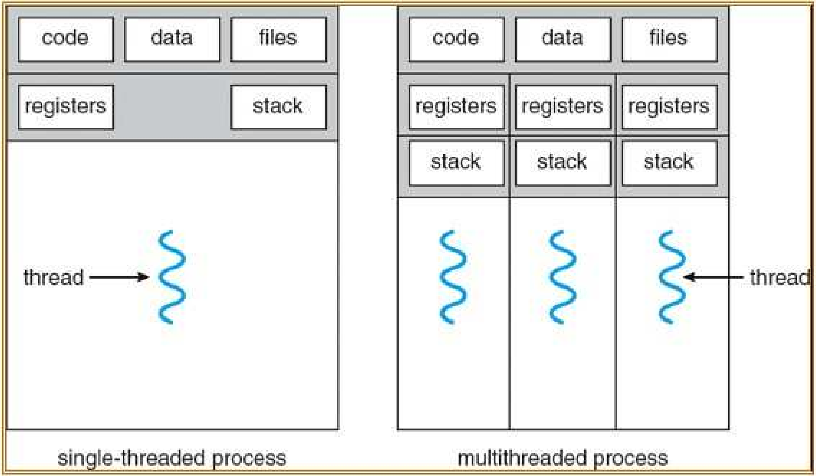
Linked list of PCBs

- (main) job queue
- ready queue
- device queues

Schedulers

- **Long-term/Job scheduler** (loads from disk)
- **Short-term/CPU scheduler** (dispatches from ready queue)

Threads


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

Note that...

On Operating Systems which support threads,
it is kernel-level threads – *not processes* –
that are being scheduled.

However, *process* sheduling $\approx$ *thread* scheduling.

CPU and IO Bursts

⋮

load, store,
add, store,
read from file

Wait for IO

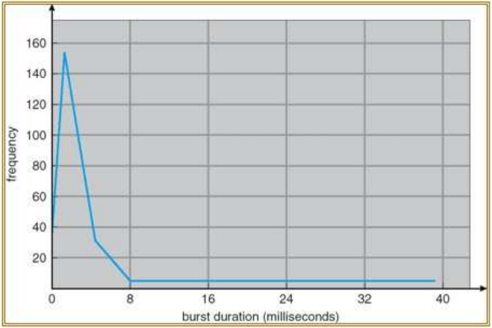
store,increment,
branch, write to file

Wait for IO

load, store,
read from file

Wait for IO

⋮



CPU Burst cycles

Intervals with no I/O usage

Waiting time

Sum of time waiting in ready queue

When should we schedule a process?

- From *running* state to *waiting* state
- From *running* state to *ready* state
- From *waiting* state to *ready* state
- Terminates

Scheme

non-preemptive
or cooperative

Scheme

preemptive

How do we select the next process?

- CPU utilization

  CPU as busy as possible
- Throughput

  Number of process that are completed per time unit
- Turnaround time

  Time between submisson and completion
- Waiting time

  Scheduling affects only waiting time
- Response time

  Time between submisson and first response

First Come, First Served (FCFS)

- Non-preemptive
- Treats ready queue as FIFO.
- Simple, but typically long/varying waiting time.

First Come, First Served (FCFS)

Example

Process	Burst time	Arrival
P_1	24	0
P_2	3	0
P_3	3	0

Gantt chart: Order P_1, P_2, P_3

P_1	P_2	P_3
0 – 24	24 – 27	27 – 30

0 24 27 30

Average waiting time: (0+24+27)/3 = 17

First Come, First Served (FCFS)

Example

Process	Burst time	Arrival
P_1	24	0
P_2	3	0
P_3	3	0

Gantt chart: Order P_2, P_3, P_1

P_2	P_3	P_1

0	3	6	30

Average waiting time: (0+3+6)/3 = 3

Convoy effect

Consider :

- P_1: CPU-bound
- P_2, P_3, P_4: I/O-bound

Convoy effect

- P_2, P_3 and P_4 could quickly finish their IO request $\Rightarrow$ ready queue, waiting for CPU.
- Note: IO devices are idle then.
- then P_1 finishes its CPU burst and move to an IO device.
- P_2, P_3, P_4, which have short CPU bursts, finish quickly $\Rightarrow$ back to IO queue.
- Note: CPU is idle then.
- P_1 moves then back to ready queue is gets allocated CPU time.
- Again P_2, P_3, P_4 wait behind P_1 when they request CPU time.

One cause: FCFS is non-preemptive

P_1 keeps the CPU as long as it needs

Shortest Job First (SJF)

- Give CPU to the process with the shortest next burst
- If equal, use FCFS
- Better name: shortest next cpu burst first

Assumption

Know the length of the next CPU burst of each process in Ready Queue

Short Job First (SJF)

Example

Process	**Burst time**	**Arrival**
P_1	6	0
P_2	8	0
P_3	7	0
P_4	3	0

Gantt chart: Order P_1, P_2, P_3, P_4

P_4	P_1	P_3	P_2
0 – 3	3 – 9	9 – 16	16 – 24

Average waiting time: (0+3+16+9)/4 = 7

With FCFS: (0+6+(6+8)+(6+8+7))/4 = 10.25

SJF – Characteristics

Optimal wrt. waiting time!

Problem: how to know the next burst?

- User specifies (e.g. for batch system)
- Guess/predict based on earlier bursts, using exponential average:
  $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
  t_n: most recent information
  τ_n: past history

Can be preemptive or not

SJF with Preemption

Shortest Remaining Time First

When a process arrives to RQ, sort it in and select the SJF including the running process, possibly interrupting it

(Remember: SJF schedules a new process only when the running is finished)

SJF with Preemption

Example

Process	Burst time	Arrival
P_1	8	0
P_2	4	1
P_3	9	2
P_4	5	3

Gantt chart

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

Average waiting time: ((10-1)+(1-1)+(17-2)+(5-3))/4 = 6.5

With SJF: (0+4+(4+5)+(4+5+8))/4 = 7.75

Priority Scheduling Algorithms

- Priority associated with each process
- CPU allocated to the process with highest priority
- If equal, use FCFS

Note: SJF is a priority scheduling algorithm with

$$p = \frac{1}{(predicted)\ next\ CPU\ burst}$$

Priority Scheduling Algorithms

Example

Process	Burst time	Arrival	Priority
P_1	10	0	3
P_2	1	0	1
P_3	2	0	4
P_4	1	0	5
P_5	5	0	2

Gantt chart

P_2	P_5	P_1	P_3	P_4
0–1	1–6	6–16	16–18	18–19

Average waiting time: (0+1+6+16+18)/5 = 8.2

Priority Criteria

- Internal Priority
  time limits, mem requirements, number of open files,
  ratio $\frac{Average\ IO\ burst}{Average\ CPU\ burst}$
- External Priority
  Critera outside the OS. Choice related to computer usage.

- Can be preemptive or not

- Problem: Starvation (or Indefinite Blocking)
- Solution: Aging

Round-Robin (RR)

- FCFS with Preemption
- Time quantum (or time slice)
- Ready Queue treated as circular queue

Round-Robin (RR)

Example

Quantum $q = 4$

Process	Burst time	Arrival
P_1	24	0
P_2	3	0
P_3	3	0

Gantt chart

P_1	P_2	P_3	P_1	...	P_1

0, 4, 7, 10, 14, 26, 30

Average waiting time: (0+4+7+(10-4))/3 = 5.66
With FCFS: (0+24+27)/3 = 17

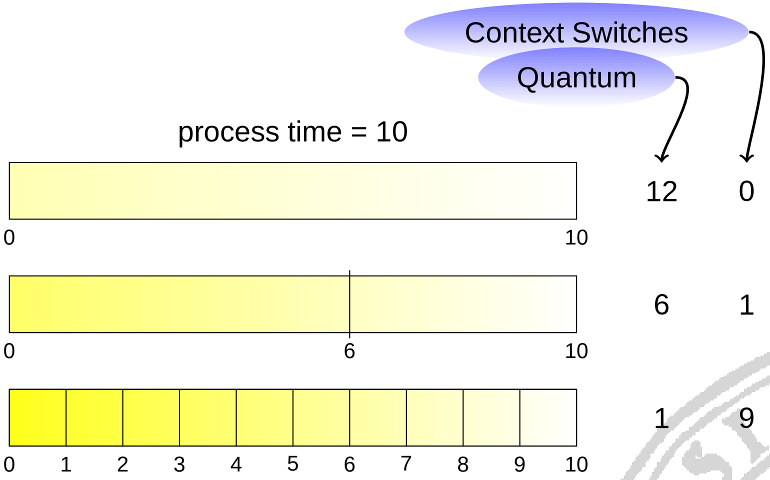
RR – Characteristics

- Turnaround time typically larger than SRTF but better response time
- Performance depends on quantum *q*
  - Small *q*: Overhead due to context switches (& scheduling)

    q should be large wrt context-switching time
  - Large *q*: Behaves like FCFS

    rule of thumb: 80% of bursts should be shorter than *q* (also improves turnaround time)

Quantum vs Context switches



Multilevel Queue Scheduling

Observation

Different algorithms suit different types of processes
(e.g. interactive *vs* batch/background processes)
and systems are often not only running interactive or "batch" processes.

Multilevel queues

We split the Ready Queue in several queues,
each with its own scheduling algorithm

Example

interactive processes: RR
background processes: FCFS/SRTF

Multilevel Queue – Scheduling among Queues

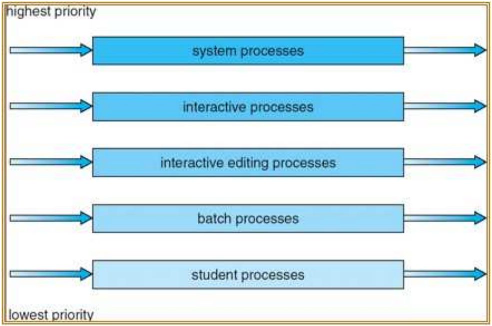
One more dimension

We need scheduling between the Ready Queues

Example (Common implementation)

Fixed-priority preemption (with priority to interactive processes)

Multilevel Queue – More complex example



where each queue has absolute priority over lower-priority queues.

No process in low-priority queues can run if high-priority queues are not empty

So, if a lower-priority queue is only used when all higher-priority RQs are empty & higher-priority processes preempt lower-priority ones, we risk starvation.

Possible solution: give time-slices to each Ready Queue

(basically RR between the queues, with different quanta for each queue)

⇒ Each queue gets a certain guaranteed slice of the CPU time.

Multi-Level Feedback Queue Scheduling (MLFQ)

With MLQ, each process is permanently assigned to one queue (based on type, priority etc).

MLFQ

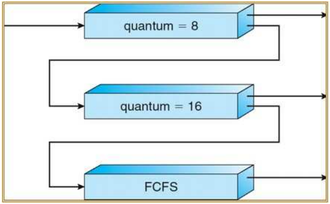
allow processes to move between queues

Idea: Separate processes according to their CPU bursts.

Example

- Let processes with long CPU bursts move down in the queue levels
- Leave I/O bound and interactive processes in high-priority queues
- Combine with aging principle to prevent starvation

MLFQ – Example



1. Round-Robin with quantum 8
2. Round-Robin with quantum 16
3. FCFS

Q_i has priority over, and preempts, Q_{i+1}.
New processes are added to Q_1.
If a process in Q_1 or Q_2 does not finish within its quantum, it is moved down to the next queue.
Thus:

- short bursts (I/O bound and interactive proc) are served quickly;
- slightly longer are also served quickly but with less priority;
- long (CPU bound processes) are served when there is CPU to be spared.

Symmetry / Asymmetry

Asymmetric MPs scheduling

One Master Server does all scheduling.
Others execute only user code

Symmetric MPs (SMP) scheduling

Each processor does scheduling.
(whether CPUs have a common or private Ready Queues)

Processor Affinity

Try to keep a process on the same processor as last time, because of Geographical Locality

(Moving the process to another CPU causes cache misses)

- Soft affinity

  The process *may move* to another processor

- Hard affinity

  The process *must stay* on the same processor

Load Balancing

Keep the workload evenly distributed over the processors

- push migration
  periodically check the load, and "push" processes to less loaded queues.
- pull migration
  idle processors "pull" processes from busy processors

Note: Load balancing goes against processor affinity.

Hyperthreaded CPUs

CPUs with multiple "cores"

Sharing cache and bus influences affinity concept and thus scheduling.

The OS can view each core as a CPU, but can make additional benefits with threads