Scheduling

Problem: Which process ready to execute commands gets the CPU?
key function of the operating system

Prerequisites for successful scheduling:

1.) CPU-I/O-Burst Cycle

Experience shows: I/O occurs after fixed amount of time in $\geq 90\%$
$\Rightarrow$ appropriate time for re-scheduling

2.) Preemptive Scheduling: Processes can be forced to relinquish processor

Scheduling Criteria

Have various, often conflicting criteria to measure success of scheduling:

- CPU utilisation
- Throughput: Number of processes completed within a given time
- Turnaround time: Time it takes for each process to be executed
- Waiting time: Amount of time spent in the ready-queue
- Response time: time between submission of request and production of first response

Scheduling algorithms

1.) First-Come, First-Served (FCFS)

Jobs are put in a queue, and served according to arrival time

Easy to implement but CPU-intensive processes can cause long waiting time.

FCFS with preemption is called Round-Robin
standard method in time sharing systems

Problem: get the time quantum (time before preemption) right.
- too short: too many context switches
- too long: Process can monopolise CPU

Shortest Job First

Next job is one with shortest CPU-burst time (shortest CPU-time before next I/O-operation)

Not implementable, but this is algorithm with the smallest average waiting time
$\Rightarrow$ Strategy against which to measure other ones
Approximation: Can we predict the burst-time?
Only hope is extrapolation from previous behaviour
done by weighting recent times more than older ones.

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

Priority Scheduling

Assumption: A priority is associated with each process
CPU is allocated to process with highest priority
Equal-priority processes scheduled according to FCFS

Two variations:

- With preemption: newly-arrived process with higher priority may gain processor immediately if process with lower priority is running
- Without preemption: newly arrived process always waits

Preemption good for ensuring quick response time for high-priority processes
Disadvantage: Starvation of low-priority processes possible
Solution: Increase priority of processes after a while (Ageing)

Multilevel Queue Scheduling

Applicable when processes can be partitioned into groups (eg interactive and batch processes):

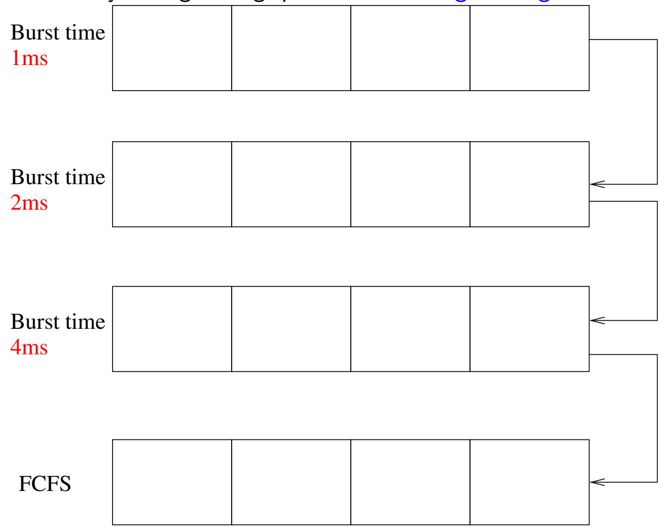
Split ready-queue into several separate queues, with separate scheduling algorithm

Scheduling between queues usually implemented as pre-emptive priority scheduling

Possible setup of queues:

- System processes
- Interactive processes
- Interactive editing processes
- Batch processes

Other way of organising queues: according to length of CPU-burst



Scheduling for Multiprocessor Systems

CPU scheduling more complex when multiple CPU's are available

Most common case: Symmetric multiprocessing (SMP):

- all processors are identical, can be scheduled independently
- have separate ready-queue for each processor (Linux), or shared ready-queue

Processor Affinity

Process affinity for CPU on which it is currently running

- Soft Affinity current CPU only preferred when re-scheduled
- Hard Affinity Process may be bound to specific CPU
  Advantage: caches remain valid, avoiding time-consuming cache invalidation and recovery

Load Balancing

Idea: use all CPU's equally (goes against processor affinity)

- Push migration: periodically check load, and push processes to less loaded CPU's
- Pull migration: idle CPU's pull processes from busy CPU's

Linux Implementation

Several schedulers may co-exist, assign fixed percentage of CPU-time to each scheduler

Important schedulers:

- Round-robin scheduler with priorities (the default scheduler)
- Real-time scheduler (process needs to be assigned explicitly to this one) (typically FIFO)

Round-Robin Scheduler with priorities

implemented in an interesting way:
maintain tree of processed ordered by runtime allocated so far
pick next process as one with least runtime allocated so far
insert new process in ready queue at appropriate place in tree
Priorities handled by giving weights to run-times.