3.4 PROCESS SCHEDULING

- Scheduling issues
- General Algorithms
  - FCFS
  - SJF
  - Round Robin
  - Priority
  - Multiple Queue
  - Guaranteed scheduling
  - Lottery scheduling

A. CPU BURST TIME

- almost all processes alternate bursts of computing and I/O requests
- **Burst time**
- how long a process requires the CPU

- **Compute Bound Process**
  - CPU bound processes
  - spend most of their time computing

- **I/O Bound Process**
  - processes that do lot of I/O
  - may spend most of the time waiting for I/O

B. SCHEDULERS

- **Long-term scheduler** (or job scheduler)
- selects which processes should be brought into the ready queue, based on the characteristics of the job
- invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
- controls the *degree of multiprogramming*

- **Short Term Scheduler** (or CPU scheduler)
- selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- also decides when to interrupt processes and the appropriate queue to move them to
- invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- CPU scheduling decisions may take place when a process:
  i. switches from running to waiting state
  ii. switches from running to ready state
  iii. switches from waiting to ready
  iv. terminates
  - scheduling can be **nonpreemptive** or
  - it can be **preemptive**

- **Dispatcher module**
- gives control of the CPU to the process selected by the short-term scheduler
- this involves:
  - 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 another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process from submission to completion
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output
- **Fairness to all jobs** – everyone has equal amount of CPU and I/O time

C. PROCESS SCHEDULING ALGORITHMS

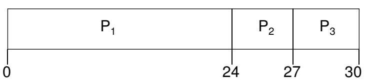
- process scheduler relies on a process scheduling algorithm to allocate the CPU
- early systems used non-preemptive policies
- most current systems emphasize interactive use and response time
- therefore use algorithms that takes care of immediate requests of interactive users

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

- historically used by many Oss
- jobs serviced according to arrival time in the ready queue
  - ➢ the earlier they arrive, the sooner they are served
- non preemptive
- very simple to use
- okay for batch systems
- unacceptable for interactive systems
- by definition the fairest algorithm

Process	Burst Time
P_1	24
P_2	3
P_3	3

- suppose that the processes arrive in the order: P_1 , P_2 , P_3
- Gantt Chart for the schedule is:

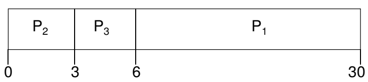


- waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- average waiting time: $(0 + 24 + 27)/3 = 17$

- now suppose that the processes arrive in the order

P_2 , P_3 , P_1

- Gantt chart for the schedule is:



- waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- average waiting time: $(6 + 0 + 3)/3 = 3$
- much better than previous case
- ***Convoy effect***: short process arriving after long process may have to wait a long time

2. Shortest-Job-First (SJF) Scheduling

- associate with each process the length of its ***next CPU burst***
- use these lengths to schedule the process with the shortest time
- SJF is ***optimal*** : it gives minimum average waiting time and shortest turnaround time for a given set of processes
- the difficulty is knowing the length of the next CPU request
  - can only estimate the length of next CPU burst
  - can be done by using the length of previous CPU bursts
- can be
  - preemptive (SRTF)
  - non preemptive
- processes with long burst times may **starve**

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Process	Burst Time	Arrival Time
P_1	7	0
P_2	4	2
P_3	1	4
P_4	4	5

3. Priority Scheduling

- a priority number (integer) is associated with each process
- CPU is allocated to the process with the highest priority (e.g. in UNIX smallest integer ≡ highest priority)
  - can be preemptive or
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- problem ≡ **Starvation** – low priority processes may never execute
- solution ≡ **Aging** – as time progresses increase the priority of the process

Process	Burst Time	Arrival Time	Priority
P_1	20	0	4
P_2	2	2	2
P_3	2	2	1

- A low number means a high priority

4. Round Robin (RR)

- each process gets a small unit of CPU time (***time quantum***), usually 10-100 milliseconds
- after this time has elapsed, the process is preempted and added to the end of the ready queue
- if there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once
  - ➢ no process waits more than $(n\text{-}1)q$ time units
- performance
  - q large ⇒ FIFO
  - q small ⇒ q must be large with respect to context switch, otherwise overhead is too high

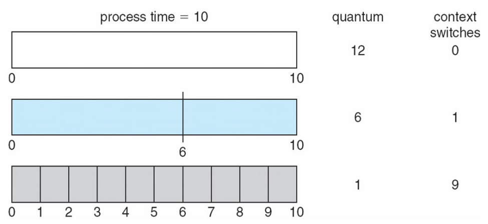
- example with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

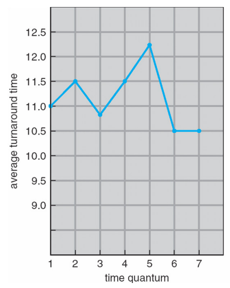
Process	Burst Time	Arrival Time
P_1	7	0
P_2	4	4
P_3	1	5
P_4	1	9
P_5	3	12

Use a time quantum of 3

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum

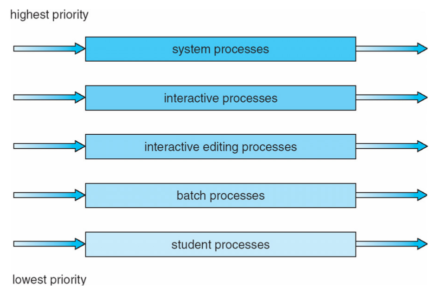


process	time
P_1	6
P_2	3
P_3	1
P_4	7

5. Multilevel Queue

- ready queue is partitioned into ***separate queues***
- a new job is placed in one of the queues
- each queue has its own scheduling algorithm, for example
  - foreground (interactive) – RR
  - background (batch) – FCFS
- scheduling must be done between the queues
  - fixed priority scheduling; (i.e., serve all from foreground then from background) - possibility of starvation
  - time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes

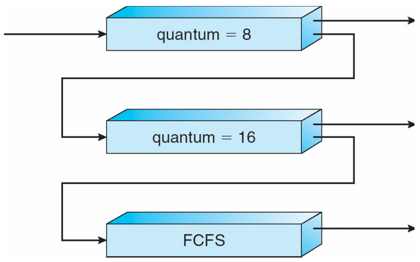
    i.e., 80% to foreground in RR, 20% to background in FCFS



Multilevel Feedback Queue

- a process can move between the various queues
  - ➢aging can be implemented this way
- assumption is that a process in interactive, therefore placed in the highest priority queue
- as the process uses its time quantum, it moves to other queus
- multilevel-feedback-queue scheduler defined by the following parameters:
  - 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

- Example
- three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- scheduling
  - a new job enters queue Q_0 which is served FCFS
  - when it gains CPU, job receives 8 milliseconds
  - if it does not finish in 8 milliseconds, job is moved to queue Q_1
  - at Q_1 job is again served FCFS and receives 16 additional milliseconds
  - if it still does not complete, it is preempted and moved to queue Q_2



6. Guaranteed Scheduling

- what if we want to *guarantee that a* process get x% of the CPU?
  - ➢ How do we write the scheduler?
- scheduling algorithm would compute the ratio of fraction of
  - CPU time a process has used since the process began
  - CPU time it is supposed to have
- process with the lowest ratio would run next
- difficult to implement

7. Lottery

- issues lottery tickets to processes for various resources
- the scheduler then randomly selects a lottery number
- the winning process gets to run
- the more lottery tickets you have, the better your chance of "winning"
- processes can give (or lend) their tickets to their children or to other processes
- more important processes can be given more tickets

D. ALGORITHM EVALUATION

- many scheduling algorithms
- need criteria to select one for a system
- also need to evaluate algorithms
- four evaluation methods
  - deterministic modeling
  - queuing models
  - simulations
  - implementation

1. Deterministic modeling

- also called ***discreet modeling***
- takes a particular predetermined workload and *defines the performance of each algorithm for that workload*

➢ e.g. given a set of processes and their burst times, which of the following algorithms gives minimum average waiting time: FCFS, SJF, RR(quantum = 10)? Assume all processes arrive at time 0 and in the order given.

Process	Burst time
P1	10
P2	29
P3	3
P4	7
P5	12

- advantages:
  - simple and fast
  - gives exact numbers for comparison
  - useful if the same processes will be run many times
  - can identify trends (e.g. SJF results in minimum waiting time)
- disadvantages:
  - requires exact numbers
  - results only apply to the cases that were used to evaluate the algorithms

2. **<u>Queuing Models</u>**

- on many systems, many different types of processes run, therefore deterministic modeling cannot be used
- however, we can *analyze characteristics of the processes*, e.g.
  - CPU times
  - I/O bursts times
  - arrival times
  - service rates
- these characteristics are represented by distributions (mathematical formulas)
- once we have these distributions, we can compute the throughput, utilization, average queue lengths, waiting times, etc.

- let
  - W=average waiting time
  - λ =average arrival rate of new processes
  - n =average queue length, excluding process running
- then in a steady state (i.e. number of processes leaving the queue equals the number of processes arriving)
  - ➢ **n = λ x W** (this is known as **Little's formula**)

i.e. during W , λ *W processes arrive in the ready queue

- ***Little's formula*** is valid for any scheduling algorithm and arrival distribution
- supposing we want to find W and know that:
  - n =14, and λ =7 processes/second
  - What is the average waiting time?

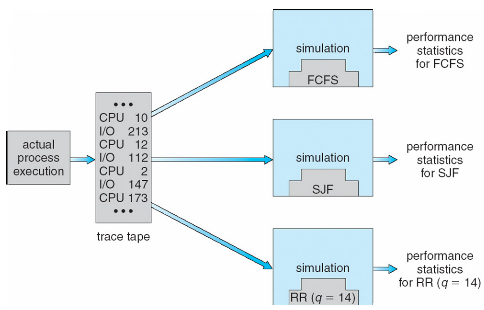
- advantages:
  - very fast way of analyzing systems
- disadvantages:
  - mathematics can become complex, so processes and systems are usually modeled in unrealistic but tractable ways
  - many assumptions may have to be made
  - accuracy is therefore questionable

3. **<u>Simulation</u>**

- program a model of a computer system
- software data structures represent system components
- a clock is maintained
- as the clock is advanced, the state of the various components in the system are modified
- if a process in the real system requires 5 seconds of CPU time, it will actually execute in a moment in the simulation (the clock will simply be advanced)
- the simulator keeps track of variables in order to calculate statistics for utilization, throughput, waiting time, etc.

- can use random number generation
  - to model arrival times, CPU times, etc.
  - does not capture the order of events
- a trace from a real system can be used
  - order of events is preserved
  - but trace can be very large
- advantages:
  - very accurate
  - level of detail is controlled
- disadvantages:
  - accuracy may require very complex models to be programmed, which takes a long time
  - traces may require large amounts of storage

Evaluation of CPU schedulers by Simulation



4. **Implementation**

- actually build the system with desired features: most accurate way of evaluating performance.
- use a benchmark to measure performance
  - a workload is run and performance is measured
  - the workload mimics processes behavior of actual processes that will run on the system
- big disadvantage: very expensive

E. MULTIPROCESSOR SCHEDULING

- concentrate on **homogeneous** processors within a multiprocessor
- CPU scheduling more complex
  - ➢ load sharing needed
- **Asymmetric multiprocessing**
  - master server handles all scheduling decisions
- **Symmetric multiprocessing (SMP)**
  - each processor is self-scheduling
  - could have all processes in a common ready queue
  - or each CPU could have its own private queue of ready processes

- some issues concerning SMP include

- **Processor affinity**
- process has affinity for processor on which it is currently running; avoid migration of processes
- a process has a value that indicates preference
  - soft affinity – possible migration
  - hard affinity – no migration

- **Load balancing**
- keep workload evenly distributed across all processors
  - push migration
  - pull migration

F. THREAD SCHEDULING

- distinction between user-level and kernel-level threads
- many-to-one and many-to-many models
  - thread library schedules user-level threads to run on LWP
  - use **process-contention scope (PCS)**
  - scheduling competition is within the process
- for one-to-one
  - kernel thread scheduled onto available CPU using **system-contention scope (SCS)**
  - competition among all threads in system

Java Thread Scheduling

- each thread has a ***priority*** ranging between 1 and 10
- a thread is given a ***default priority of 5*** when created
- higher priority processes have preference
  - ➢ FIFO used if there are multiple threads with the same priority
- Thread.MIN_PRIORITY Minimum Thread Priority (1)
- Thread.MAX_PRIORITY Maximum Thread Priority (10)
- Thread.NORM_PRIORITY Default Thread Priority (5)
- priorities are not adjusted dynamically
  - ➢ priority will only change if this is done explicitly in the program
  - ➢ using setPriority() method:

```
setPriority(Thread.NORM_PRIORITY + 2);
```


- loosely-defined scheduling policy
- a thread runs until:
  - it's time quantum expires
  - it blocks for I/O
  - it exits its run() method
- some systems *may* support preemption
- JVM schedules a thread to run when
  - the currently running thread exits the Runnable state
  - a higher priority thread enters the Runnable state

- since the JVM doesn't ensure time-slicing, the yield() method may be used:

➢ this yields control to another thread of equal priority

```
while (true) {
        // perform CPU-intensive task
        . . .
        Thread.yield();
}
```


Relationship between Java and Win32 Priorities

Java priority	Win32 priority
1 (MIN_PRIORITY)	LOWEST
2	LOWEST
3	BELOW_NORMAL
4	BELOW_NORMAL
5 (NORM_PRIORITY)	NORMAL
6	ABOVE_NORMAL
7	ABOVE_NORMAL
8	HIGHEST
9	HIGHEST
10 (MAX_PRIORITY)	TIME_CRITICAL

G. LINUX SHEDULING

- tries to have ***good response time and throughput***
- increased support for SMP, as well as processor affinity and load balancing
- ***preemptive***, ***priority-based*** scheduling algorithm
- good interactive performance: even during high load, interactive tasks should be scheduled immediately
- ***fairness***: no process should be deprived of a time slice for a reasonable amount of time; no process should get an unfairly high amount of time slice

i. Priorities

- two priority ranges: real-time and nice values
- **real-time** range from 0 to 99, **nice** value from 100 to 140
- higher priority tasks have longer time quanta
- lower priority tasks have shorter time quanta
- task are eligible for execution on the CPU as long as it has time remaining in its time slice
- when time slice is exhausted, the task is considered exhausted and is not eligible to run until all other tasks have exhausted their time slices

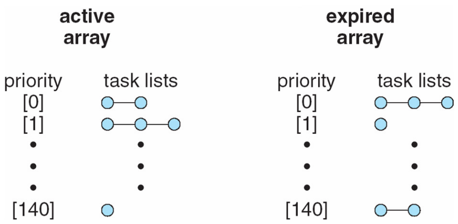
Priorities and time slice length

numeric priority	relative priority		time quantum
0 • • • 99	highest	real-time tasks	200 ms
100 • • • 140	lowest	other tasks	10 ms

ii. Scheduling Data Structures

- a structure called ***runqueue*** is used to store all runnable processes
- each processor has its own *runqueue* data structure and schedules itself independently
- *runqueue* has an *active array* and an *expired array*
- *active* array has all tasks with time remaining in their time slices
- *expired* array contains all expired tasks
- processes in both arrays are indexed by priority
- scheduler picks highest priority task from the active array
- if the process exhaust its time slice, it is placed in the expired array
- when the active array is empty, the two arrays are swapped

List of Tasks Indexed According to Priorities



iii. **Real-time scheduling**

- real-time tasks have static priorities
- all other tasks have dynamic priorities based on *nice values +/- 5*
- interactivity determines whether we will have nice+5 or nice-5
- if a task has been sleeping a long time for I/O, then it is nice-5 (since it is more interactive)
- when a task has exhausted its time slice its dynamic priority is recalculated
- thus when active and expired arrays are exchanged, all tasks already have new priorities

iv. **Recalculating Time Slices**

- many OS have an explicit method of recalculating time slices when they all reach zero (including older Linux versions)

```
for(each task in the system) {
    recalculate priority
    recalculate time slice
}
```

- it can take a long time
- non-determinism of a randomly occurring recalculation of the time slice is a problem with deterministic real-time programs
- with new O(1) scheduler, recalculation is as simple as switching the active and expired arrays

v. **Linux 2.4 Scheduler**

- processes are assigned CPU time once each **epoch**
- a new epoch begins when no ready job has any CPU time left
- process can carry over half its unused CPU time from last epoch
- priority (known as **goodness)** is recalculated each time the scheduler runs
- goodness is largely determined by the unused CPU time
- preference is given to a process if it uses the same memory space as the last process
- so the memory management cache doesn't need to be cleared
- in a multiprocessor machine, preference given to process if it last ran on the current processor.
- improves cache hits

vi. **Linux 2.6 Scheduler**

- 140 levels : first 100 are "real time" , last 40 for "user"
- *Active vs. expired arrays*
- active array holds processes to be scheduled
- when user process uses up its quantum it moves to the expired array
- priority is then recalculated based on "interactivity":
  - ratio of how much it executed compared to how much it slept
  - adjusts priority ±5.
  - quantum is based on priority
    - better priority has longer quantum

  (Note: different sources quote different ranges…)
- queues are swapped when no active user process left.
- like the 2.4 scheduler this allows low priority processes to get a chance.
- separate structures for each cpu, but migration is possible

H. WINDOWS XP SCHEDULING

- schedules threads rather than processes
- uses quantum based preemptive priority scheduling
- **32 priority levels** : 31 is the highest, O is the lowest
  - 0 is reserved for the thread used for memory management
  - 1-15 is the *variable* class, for normal applications
  - and 26-31 is the *real time* class for real time processes
- there is a queue for each priority
  - scheduler traverse queues from highest to lowest until it finds a thread to run
  - if there is no thread to run, the idle thread is executed

- when the thread's quantum expires, its priority is lowered
- when a thread becomes ready after waiting, it is given a priority boost
- also viewed as multiple feedback queue algorithm
- a thread is preempted if
  - a higher priority thread becomes ready
  - thread quantum expires
  - thread blocks