Scheduling

Chapters: 5.1, 5.2, 5.3, 5.4, 5.5, 5.7.1, 5.8

# Scheduling Basics

* Designs what runs in what order
  + Involves threads or processes
  + In **kernel** for **kernel-level threading**
* Scheduling selects task from “**ready**” queue
* When do we schedule? At **all times involving OS**
  + **Interrupts** (device completion, timer interrupt etc.)
  + **Syscalls** (voluntary process/thread termination or yields)
  + **Exceptions** (including involuntary termination)
* The scheduler makes decisions via **policies**, implemented by an **algorithm**
* Tasks are switches by **dispatcher**

## Dispatcher

* Module that gives control of CPU’s core scheduled by CPU scheduler. Involves:
  + Switching context from one process to another
  + Switching to user mode
  + Jumping to the proper location in user program to resume said program.
* Should be as fast as possible as it is invoked at **every context switch**
* **Dispatch Latency**: time taken to stop one process and start or continue running another**.**
* **Voluntary Context Switch**:when a process has given up control of the CPU as requires a currently unavailable resource (such as blocking for I/O)
* **Nonvoluntary Context Switch**:CPU has been taken away from a process because its time slice has expired, or a higher priority process needs CPU time.

## Policy or Thread Behaviour

* Diagram

  Description automatically generatedWhat are good policies for selecting tasks, and how do we decide if they are good?
* Process or thread execution consists of:
  + CPU execution: **CPU burst**
  + I/O wait: **IO Burst**
* CPU Burst distribution is **application dependent.**
  + For example, an AI mechanism would use a lot more CPU execution, whereas a web browser would focus on listening for user input.
* Scheduling should make the most of the resources we have
  + e.g. never leave CPU idle
* While a process or thread is waiting for IO, another can run on CPU
  + Focus on **single** CPU /core

Diagram, box and whisker chart

Description automatically generated

* We can visualise the bursts as shown above.
* As you can see, treating IO and CPU operations equally would be extremely inefficient.

## Scheduling Goals

### Performance

* There are many performance goals, some of which may conflict, including:
  + **Maximising** **CPU utilisation** (keep CPU as busy as possible)
  + **Maximise** **throughput** (processes completed per unit time)
  + **Minimise** **turnaround time** (time from submission of task to completion)
  + **Minimise** **waiting time** (all periods tasks spend waiting in ready queue from submission)
  + **Minimise** **response time** (time from submission of request to response produced)
  + **Minimise** **energy** (joules per instruction)
* In most cases we optimise the **average metric** by finding a **compromise**
* Sometimes minimise worst case e.g. response time.

### Fairness

* We want the OS to be **fair**, but there is no concrete definition or metric for this
  + We could look at equal CPU consumption, but would this be per user? Per process? Per thread?
  + What if one process is CPU bound and one is I/O bound?
* Sometimes we want to be **unfair**
  + We explicitly favour particular classes of requests: **priority systems**
    - e.g. a user paying premium access or driving a car and suddenly break is activated.
  + Avoid starvation: we want to ensure that every task gets some service

## Classes of Schedulers

* A picture containing text, sign, screenshot

  Description automatically generated**Batch**: Throughput/utilisation oriented e.g. audit inter-bank funds transfers each night, pixel rendering, Hadoop/MapReduce jobs
* **Interactive:** response time oriented, priority given to response time
* **Real-time**: deadline driven, we need to ensure that completion is made before deadline e.g. embedded systems (cars, aeroplanes.)

## Re-evaluating Scheduling Decisions

* CPU-scheduling decision may take place under the following circumstances:

1. A process switches from the running state to waiting state as result of I/O request or wait() command
2. Process switches from running to ready state because of interrupt
3. Process switches from waiting state to ready state.
4. Process terminates.

* Scenarios 1 and 4 describe n**on-pre-emptive scheduling**:
  + processes/threads execute until
    - Completion
    - I/O block ensues
  + Scheduler gets involved only at exit or on specific request i.e. decision irreversible
  + Clock interrupts do not affect running process
* Scenarios 2 and 3 describe **pre-emptive scheduling**
  + While process/thread executes, its thread may be paused and another process/thread resumes execution
  + Involves **involuntary process switching**
  + Every clock interrupt can suspend and switch to another process.

# Scheduling Algorithms

## FCFS- First-Come First-Served

* Processes are assigned to CPU in **order of request** (or **arrival**). Like concert scenario.
* Non-preemptive approach
* Some **abstract** metric of CPU time allocated to each task
* **Non-preemptive** approach

#### Example

* For example, say process P1, P2 and P3 arrive in this order, such that
  + P1 takes 24 units of CPU time
  + P2 takes 3 units
  + P3 takes 3 units
* Calculating the average turnaround time (time taken for job completion after submission/request):
  + P1 = 24
  + P2 = 27
  + P3 = 30
  + Average turnaround time =
* As we can see, short processes were delayed by long processes. This is the **convoy effect**
* If we change the order to P2, P3, P1, the turnaround times become:
  + P1 = 3
  + P2 = 6
  + P3 = 30
  + Average turnaround time =
* Clear improvement

### FCFS Drawbacks

* Average response time can be **poor** as seen by convoy effect
* May lead to **poor utilisation** of **other resources**, with poor overlap of CPU and I/O activity
  + E.g. CPU intensive job prevents I/O intensive job from completing small bit of computation
  + Keeps I/O on idle hence poor utilisation.

# SJF- Shortest Job First

* Associate each process with **length of** **CPU time**, and order in ascending
* Can be non-preemptive and preemptive:
  + Non-preemptive: process will run to completion, cannot be interrupted
  + Preemptive: new processes with a smaller CPU time than remaining time of current process will pre-empt. Also called **shortest remaining time left** (**SRTL**)

### Non-preemptive SJF Example

|  |  |  |
| --- | --- | --- |
| Process | Arrival Time | CPU Time |
| P1 | 0 | 7 |
| P2 | 2 | 4 |
| P3 | 4 | 1 |
| P4 | 5 | 4 |

* Table shows arrival and CPU times for processes.
* Order of Shortest time: P3, P2, P4, P1
* According to SJF:

1. P1 arrives alone and is complete
2. P2, P3 and P4 have arrived. P3 has smallest CPU time, so is carried out first
3. P3 is completed. P2 came first and is executed
4. P4 is executed

* Turnaround times:


  + Average =
* 3 CTX switches made
* This is perfect but requires complete view of the future
* Can try approximating, predicting next CPU burst as an **exponential average** of the measured lengths of previous CPU bursts using following formula:
  + : length of nth CPU burst
  + : most recent information
  + for define

### Preemptive SJF Example

|  |  |  |
| --- | --- | --- |
| Process | Arrival Time | CPU Time |
| P1 | 0 | 7 |
| P2 | 2 | 4 |
| P3 | 4 | 1 |
| P4 | 5 | 4 |

1. P1 starts running
2. P2 arrives after 2 units. P1 still has 5 units left, but P2 will only take 4. Switches to P2
3. P2 runs for 2 units until P3 arrives. P3 will take 1 units, 2 units remaining for P2. Switches to P3
4. Graphical user interface, application

   Description automatically generatedP3 is finished, P4 arrives. P2 has shortest time remaining, so P2 executed
5. P4 is next shortest (4 units) so it executes
6. P1 is executed.

* Turnaround time:

  + Average =
* 5 CTX (context) switches

### SJF Drawbacks

* Although it is **optimal**, it can only be approximated
  + Too complex to be implemented in practice
  + Not always possible to determine CPU/IO burst,
* As a result it provides a nice **upper bound**

## Round-Robin

* Each process is allowed to run for a specified time interval: a **quantum**
* After this time has elapsed

1. Process is pre-empted
2. Process added to end of ready queue
3. Next process scheduled

* If the process **terminates** or **blocks for IO** before this time:
  + added to wait queue
  + Next process is scheduled

#### Example

|  |  |
| --- | --- |
| Process | CPU time |
| P1 | 53 |
| P2 | 8 |
| P3 | 68 |
| P4 | 24 |

* Time Quantum = 20
* All processes requested at 0

1. P1 runs for 20 units. CTX switch with 33 units left
2. P2 runs to completion. P3 begins
3. P3 runs for 20 units, CTX switch with 48 units left.
4. P4 runs for 20 units, CTX switch with 4 units left
5. P1 runs for 20 units. CTX switch with 13 units left
6. P3 runs for 20 units, CTX switch with 28 units left.
7. A picture containing graphical user interface

   Description automatically generatedP4 runs for 4 units to completion. CTX switch.
8. P1 runs for 13 units to completion. CTX switch
9. P3 runs for 20 units. CTX switch with 28 units left, CTX switch, runs again with additional switch to completion

* Waiting Times:
  + Average waiting time:
* Average turnaround time:

### Choosing Time Quantum

* **Context switching** will impact choice of time quantum
  + E.g. CTX: 1ms, time quantum: 4ms, 20% time spent on quantum switching alone
  + On the opposite side long quantum times long processes will prevent other smaller processes from running
* Typical numbers:
  + context switches in order of 10 of microseconds
  + Quantum/time slide is 1KHz every 1ms
* Lots of processes would result in a **poor response time**

### Pros and Cons

* Advantages:
  + Solution to fairness and starvation
  + Fair allocation of CPU across jobs
  + Low average waiting times when jobs vary in length
  + good for responsiveness (interactivity) if small number of jobs
* Disadvantages
  + Context switching time may accumulate for longer jobs

## Graphical user interface, application, table, Excel Description automatically generatedText, table Description automatically generatedFCFS vs RR

pages, with bit offset

## PRIO- Priority

* Diagram, shape

  Description automatically generatedProcesses are organised into priority categories.
* Executed in categories on a FCFS basis
* Problems:
  + Starvation: of lower priority jobs, as higher priority jobs always running
  + Deadlock: if a high priority job is waiting on low priority job, it won’t run as lower priority jobs struggle to run.
  + Leads to **indefinite blocking** or **starvation**0
* Solution: **aging**: increase the priority of processes waiting in the system for a significant amount of time.

### Graphical user interface Description automatically generated with low confidencePriority Example

### Assigning Priorities

* Statically based on:
  + process type
  + user
  + how much user is “paid
* Dynamically, based on how much they run vs IO
  + priority – 1/f, where f = size of quantum used last
    - Longer a process ran, lower its priority
    - Process that runs shortest gets highest priority to run next

## MQ- Round Robin PRIO/Multiple Queues

Diagram

Description automatically generated

Example:

Table

Description automatically generated

## MLFQ - Multilevel Feedback Queue

* Execution plan:
  + Same as MQ, except each queue has different time quanta:
    - shortest for high-priority, longest for low-priority
  + Processes start at highest priority
    - When process exceeds its quanta, its moved to lower priority
    - When a problem becomes interactive, its moved to higher priority
* Graphical user interface, application, table

  Description automatically generatedProblem with this system is if the user discovers how to make their tasks interactive, they can mess with the system
* Example:
* In general a MLFQ is defined by the following parameters:
  + number of queues
  + Scheduling algorithm per queue
  + Method used to determine when to upgrade a process to a higher priority queue
  + Method used to demote a process to a lower priority queue.
  + Method to determine which queue a process will enter when said process needs service.

# Thread Scheduling

* Most OS schedule **kernel-level threads**.
* User level threads must be mapped to kernel level threads to be scheduled.

## Contention Scope

* Systems using one-to-one or many-to-many mapping must schedule user-threads to run on lightweight processes (LWPs).
* **Process-Contention Scope** (**PCS**): where the thread library schedules user-level threads to run on available LWPs. Note that these threads would belong to a process, and that to actually run on the CPU, the LWP must still be scheduled by the OS**.**
  + User-level thread priorities set by programmer and not adjusted by thread library. some libraries may facilitate priority alteration of a thread.
  + PCS will also pre-empt the thread currently running in favour of a higher-priority thread, but there is no guarantee of time slicing among threads of equal priority.
* **System Contention Scope** (**SCS**): all threads compete for CPU. One-to-one mapping requires only this.

## Pthread Scheduling

* Involves specifying PCS and SCS during thread creation.
* Ptherads identifies following scope values:
  + PTHREAD\_SCOPE\_PROCESS: Schedules threads using PCS scheduling
  + PTHREAD\_SCOPE\_SYSTEM: Schedules threads using SCS scheduling
* On system using many-to-many model:
  + PTHREAD\_SCOPE\_PROCESS policy schedules threads onto available LWPs (Lightweight Processes).
    - Number of LWP’s maintained by thread library using scheme e.g. scheduler activations
  + PTHREAD\_SCOPE\_SYSTEM scheduling policy will create and bind LWP for each user-level thread, effectively mapping threads using one-to-one policy.
* Two functions for getting and setting contention scope policy:
  + pthread\_attr\_setscope(pthread\_attr\_t \*attr, int scope)
  + pthread\_attr\_getscope(pthread\_attr\_t \*attr, int scope)
* Parameters:
  + pthread\_attr\_t \*attr: pointer to attribute set of thread
  + int scope: How contention scope is to be set.
* Error of functions yields nonzero value.
* Linux and macOS only allow PTHREAD\_SCOPE\_SYSTEM

# Linux Scheduling

* Schedules **tasks**
* Default scheduler: **Completely Fair Scheduler** (**CFS**)
* Based on **scheduling classes**, where each class is assigned to a specific priority.
* Allows kernel to accommodate different scheduling algorithms based on needs of system and its processes
* Standard Linux kernels use **two classes**

## CFS Scheduling (Default)

* Assigns a proportion of CPU processing time to each task.
* Based on **nice value** assigned to each task:
  + Nice values range from -20 to 10. The lower the nice value, the higher the priority.
  + Nice processes finish last
* Identifies **targeted latency**: an interval of time during which every runnable task should run at least once.
* Does not use priorities, but instead selects task with next smallest **virtual run time**.
  + Associated with a decay factor based on the priority of the task. If a task physically runs for 200ms:
    - Task with Nice value > 0 will have vruntime > 200
    - Task with Nice value = 0 will have vruntime = 200
    - Task with Nice value < 0 will have vruntime < 200
* Also supports load balancing by using a metric that combines a thread’s priority with its average rate of CPU utilisation:
  + I/O task that is high priority but mostly I/O bound will have similar load to lower priority CPU bound task.

## Realtime Scheduling

* Implemented using POSIX standard.
* Any task scheduled using either FIFO or RR runs at higher priority than normal, non-realtime tasks.
* Linux uses two separate priority ranges:
  + 0-99: Real-time
  + 100-139: Normal
* Normal tasks priorities based on nice values: -20 = 100, +19 = 139.

# Algorithm Evaluation

## Analytic Evaluation

* Use given algorithm and system workload to produce a formula/number to evaluate the performance of the algorithm for that workload.
* **Deterministic modelling**: takes particular predetermined workload and defines performance for each algorithm on the workload.
* Simple, fast method giving exact measurements to compare the algorithms.
* Requires exact numbers of input, and answers only apply to these specific cases.
* Mainly used for describing scheduling algorithms, and providing examples.
* Over a set of example, deterministic modelling may indicate trends that can be analysed and proven separately.

## Queue Models

* Uses distribution of CPU and I/O bursts to create distribution via approximation or estimation.
* Distributions are commonly exponential and described by their means.
* Can also describe distribution of times when processes arrive in the system.
* Can compute average throughput, utilisation, waiting time and other properties for most algorithms via distributions.
* Little’s formula:
  + where
    - n average long-term queue length
    - average arrival rate for new processes in the queue
    - average waiting time in the queue
* Drawbacks:
  + Mathematics behind distributions can be complex, thus arrival and service distributions are often mathematically tractable but unrealistic.
  + Queuing models resultantly are only approximations of real systems.