Processor Scheduling

2023-24 COMP3230A

Time is money. – Ben Franklin

The best performance improvement is the transition from the non-working state to the working state. This is infinite speedup. – John Ousterhout

Contents

Related Learning Outcomes

- ILO 2a - explain how OS manages processes/threads and discuss the mechanisms and policies in **efficiently sharing** of CPU resources.

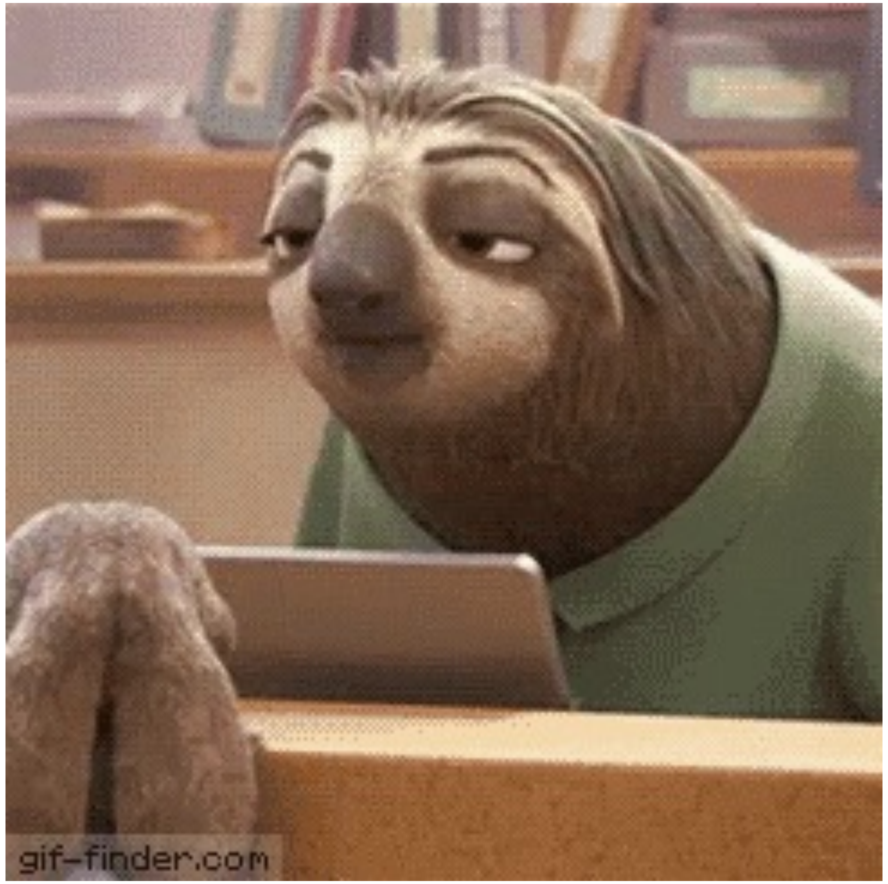
- ILO 3 [Performance] - analyze and evaluate the algorithms of . . . and explain the major performance issues . . .

Readings & Reference

- Required Readings
  - Chapter 7, Scheduling: Introduction
    - http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched.pdf
  - Chapter 8, Scheduling: The Multi-Level Feedback Queue
    - http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf
- References
  - Chapter 9, Scheduling: Proportional Share
    - http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-lottery.pdf
  - Chapter 10, Multiprocessor Scheduling
    - http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-multi.pdf

Scheduling

- First come, first served



Google/Amazon/Yahoo: 5-10% lose by 100 ms!

Scheduling: Trade-offs

- If overloaded:
  - Would implementing a different scheduling policy help, or hurt?
  - How much worse will the performance be if the # of users/jobs doubles again?
  - Should you turn away some users so that others will get acceptable performance?
  - Does it matter which users/jobs to turn away?
  - How much would it help if you run to get more resources (more CPUs)?

Processor Scheduling

- What is processor scheduling?
  - Is the activity of selecting the next process/**thread** to be serviced by the processor(s)

- Scheduling policy used by the OS would influence
  - Quality of service provided to users
    - A good scheduler can make a big difference in perceived performance and user satisfaction
  - Effective use of resources
    - Process switching is expensive; scheduling decision has an impact on the efficient use of CPU
  - System performance
    - We would like to maximize the number of processes that completed per unit time

Scheduling Levels

- High-level / Job / Long-term scheduling
  - Deals with creating a new process
  - **Controls number of processes** in system at one time
    i.e., degree of multiprogramming
- Intermediate-level / Medium-term scheduling
  - Determines which processes shall be allowed to compete for processors
    - **Deals with swapping processes in/out**
  - Responds to fluctuations in system load
- Low-level / Short-term scheduling
  - What process should we run next?
    - Assigns processors to processes

Scheduling Terms & Concepts

- CPU-bound (compute-bound) process
  - When running, tends to use all the processor time that allocated to it

- I/O-bound process
  - When running, tends to use the processor only briefly before generating I/O request and relinquishes the processor

- **Turnaround time** - amount of time to execute a particular process (***from submission to complete servicing***)
  - $T_{turnaround} = T_{completion} - T_{arrival}$
  - Objective: **minimize** turnaround time

- Waiting time - amount of time a process was ***waiting in the ready queue***

Scheduling Terms & Concepts (2)

- Response time - amount of time it takes from when a process (job) is submitted to the first time it is scheduled
  - An important performance metric of **interactive tasks**
  - $T_{response} = T_{firstrun} - T_{arrival}$
  - Objective: **minimize** response time

- Throughput - average # of processes (jobs) that complete their execution per time unit
  - Objective: **maximize** throughput

- Fairness - all similar processes (jobs) are treated the same, and even if processes are in different priority classes, **no** process should **suffer indefinite postponement** (starvation) due to scheduling

When to Schedule

- When a new process is created
- When a process exits
- When a process blocks for I/O or other event
- When a process invokes system call
- When an Interrupt occurs
  - I/O interrupt: a process blocked waiting for the I/O now be ready
  - Clock interrupt: a periodical signal to invoke the scheduler

Workload Assumptions

- Here are the assumptions about processes running in our evaluation system
  - All processes (jobs) **only use** the CPU, i.e. no I/O
    - Although not realistic, ***jobs with I/O can be treated with each CPU burst (separate by I/O) as an independent job***
  - The duration of runtime (CPU burst) of each process (job) is **known beforehand**

Algorithm Evaluation

- Deterministic modeling
  - Given a predetermined workload and evaluate the performance of each algorithm
- Example for performance analysis of different algorithms

Assume that new processes **always arrive just before** the clock tick

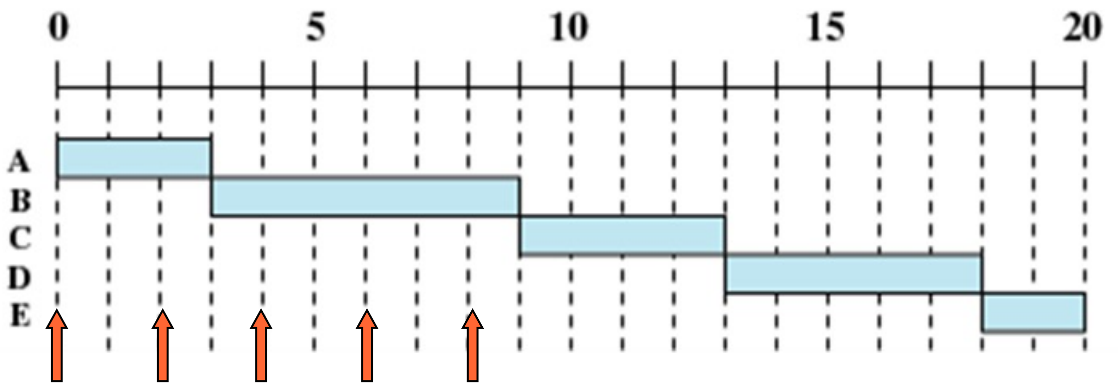
Process	Arrival Time	CPU burst
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

First-In-First-Out (FIFO) Scheduling

- Also known as FCFS (First Come First Served)
- Processes dispatched according to arrival time
  - Order in a list (queue) according to the arrival time
- Is a **non-preemptive** scheme
  - Once given the CPU, run until completion or voluntary release of CPU
- Advantages
  - **Fair** - all processes are treated equally
  - Easy to implement

Process	Arrival Time	CPU burst
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

FIFO Scheduling



Time	Queue	CPU
0	A	
0		A
2	B	A
3		B
4	C	B
6	C←D	B
8	C←D←E	B
9	D←E	C
13	E	D
18		E

- Turnaround time for
  - A = 3 – 0 = 3
  - B = 9 – 2 = 7
  - C = 13 – 4 = 9
  - D = 18 – 6 = 12
  - E = 20 – 8 = 12

- Average turnaround time: (3+7+9+12+12)/5 = **8.6**

FIFO Scheduling

- Disadvantages
  - Short processes may have to wait relatively longer time when they are queued behind a CPU-bound process
  - Not good for interactive processes

Response time = Wait time

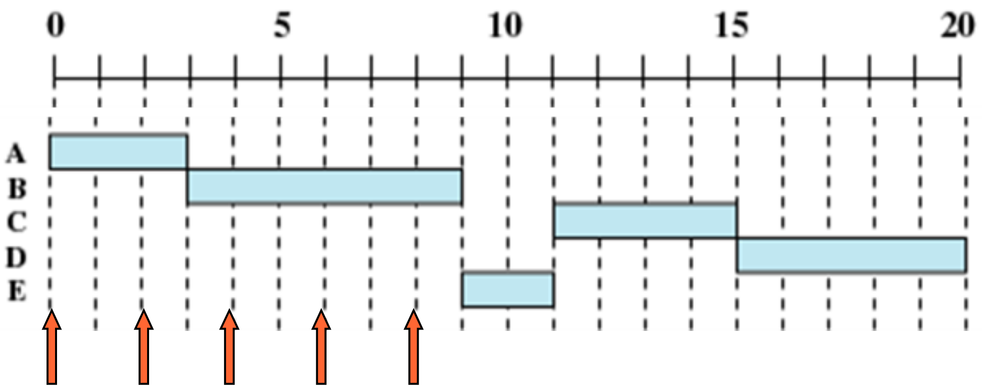
Response time of
A=0
B=1
C=5
D=7
E=10

Shortest-Job-First (SJF) Scheduling

- Also known as Shortest-Process-First
- A non-preemptive scheme that selects process with shortest "CPU burst" to run next
- Advantage
  - Lower average wait time than FIFO
    - Reduces the number of waiting processes
    - Known to be better in terms of average waiting time

SJF Scheduling

Process	Arrival Time	CPU burst
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



Time	Queue	CPU
0		A
2	B	A
3		B
4	C	B
6	C←D	B
8	E←C←D	B
9	C←D	E
11	D	C
15		D

- Average turnaround time: (3+7+11+14+3)/5 = **7.6**

Response time of
A=0
B=1
C=7
D=9
E=1

Shortest-Job-First (SJF) Scheduling

- Disadvantages
  - Because of **non-preemptive nature**, newly short/interactive jobs may be forced to wait for a running CPU-bound job
    - may results in slow response times
  - Possibility of starvation for longer processes
    - If there always have short interactive processes, longer processes have to wait longer
    - Potentially large variance in wait times
- These policies are not good for interactive jobs which demand to have good response time

Highest-Response-Ratio-Next (HRRN) Scheduling

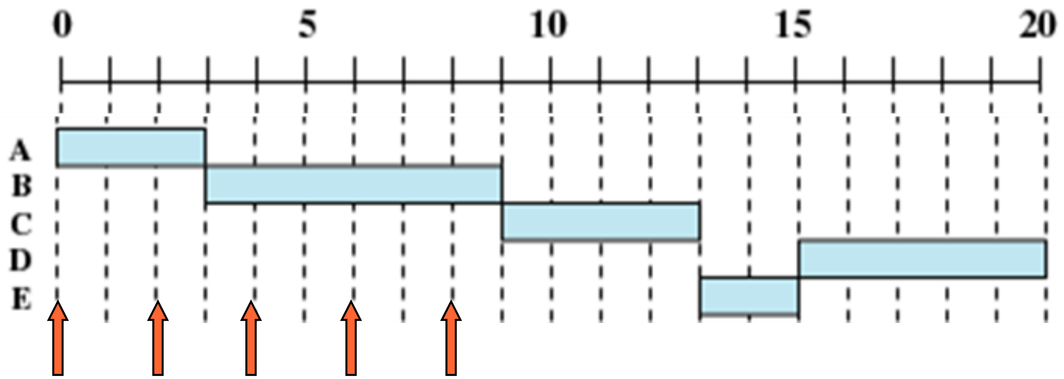
- A **non-preemptive** scheme; is an improvement of SJF scheduling
- We take into consideration of how long process has been waiting
- Calculate the priority based on predicted service time as well as how long has this process been waiting

$$priority = \frac{time\ waiting + service\ time}{service\ time}$$

- Shorter processes are favored for scheduling
- **Aging effect**; prevents indefinite postponement

Process	Arrival Time	CPU burst
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

HRRN



Time	Queue	CPU
0		A
2	B	A
3		B
4	C	B
6	C←D	B
8	C←D←E	B
9	D←E	C
13	D	E
15		D

- Average turnaround time: (3+7+9+14+7)/5 = **8**

Response time of
A=0
B=1
C=5
D=9
E=5

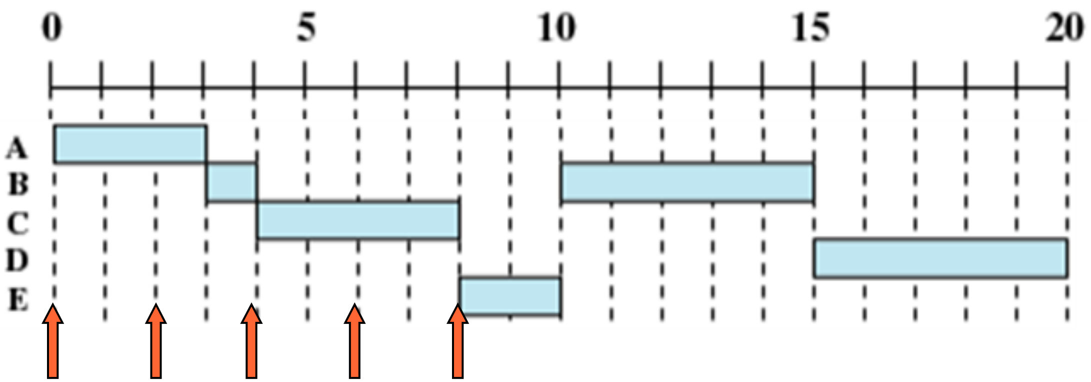
Shortest Time-to-Completion First (STCF) Scheduling

- **Preemptive** version of SJF
  - Running process may be interrupted and switched to Ready state by the scheduler

- **Arriving processes** will trigger scheduler to check against the **remaining running time** of existing and new processes, and schedules the process with the shortest run-time-to-completion
  - Remaining = CPU burst – elapsed service time
  - Must maintain information about the elapsed service time

STCF Scheduling

Process	Arrival Time	CPU burst
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



Time	Queue	CPU
0		A
2	B	A (1)
3		B
4	B	C
6	B←D	C (2)
8	B←D	E
10	D	B
15		D

- Average turnaround time: (3+13+4+14+2)/5 = **7.2**

Response time of
A=0
B=1
C=0
D=9
E=0

Waiting time of
A=0
B=7
C=0
D=9
E=0

STCF Scheduling

- Disadvantages
  - Very large variance of waiting times: long processes may wait even longer than under SJF; because with SJF, long process won’t be preempted as being scheduled
  - Not always optimal
    - Short incoming process can preempt a running process that is near completion
    - Would experience more context switches and affect overall performance

Round-Robin (RR) Scheduling

- Keep processes in ready queue in FIFO
  - New processes are added to the tail of the ready queue

- Processes run only for a limited amount of time called a **time slice** or a **time quantum**

- **Preemption**
  - Upon clock interrupt, if currently running process has its quantum expires, place it to the tail of ready queue; next ready process is selected and dispatched

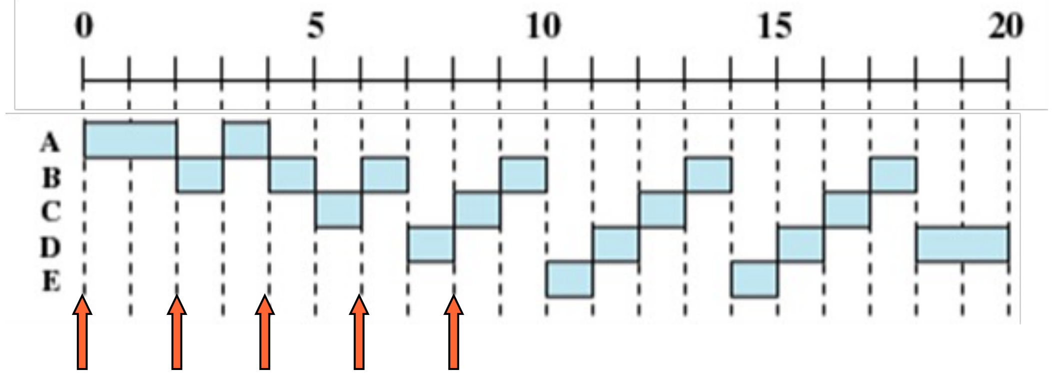
- Advantages
  - Fair
  - Good for interactive processes

RR Scheduling

	Average Turnaround	Average Waiting	Average Response
FIFO	8.6	4.6	4.6
SJF	7.6	3.6	3.6
HRRN	8	4	4
STCF	7.2	3.2	2
RR	10.8	6.8	0.8

Process	Arrival Time	CPU burst
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Time Quantum = 1



- Average turnaround time: (4+16+13+14+7)/5 = **10.8**

Response time of	Waiting time of
A=0	A=1
B=0	B=10
C=1	C=9
D=1	D=9
E=2	E=5

Time	Queue	CPU
0		A
1		A
2	A	B
3	B	A
4	C	B
5	B	C
6	D←C	B
7	C←B	D
8	B←E←D	C
9	E←D←C	B
10	D←C←B	E
11	C←B←E	D
12	B←E←D	C
13	E←D←C	B
14	D←C←B	E
15	C←B	D
16	B←D	C
17	D	B
18		D

B←C

B←D

RR Scheduling

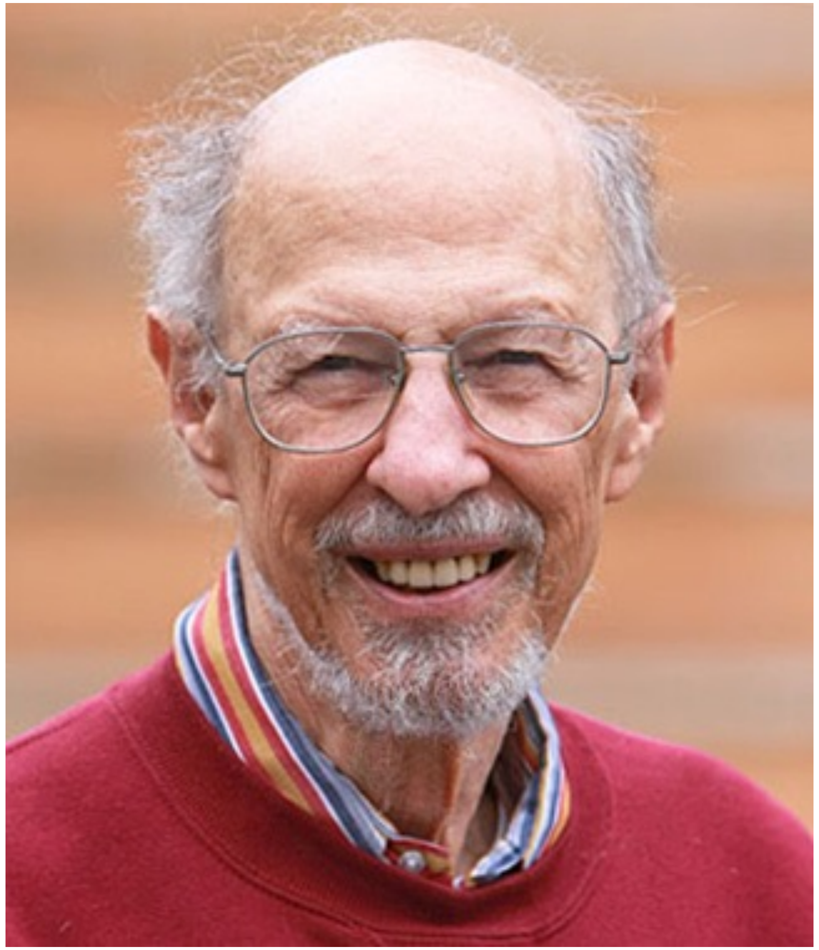
	Average Turnaround	Average Waiting	Average Response
FIFO	8.6	4.6	4.6
SJF	7.6	3.6	3.6
HRRN	8	4	4
STCF	7.2	3.2	2
RR (1)	10.8	6.8	0.8
RR (2)	10	6	1.8

- Disadvantage
  - The RR policy does not go well in terms of turnaround time
- Quantum size
  - Determines response time to interactive requests
  - Very large quantum size
    - Processes run for long periods
    - Degenerates to FIFO
  - Very small quantum size
    - System spends more time context switching than running processes
  - Middle-ground
    - Long enough for interactive processes to issue I/O request
    - Long processes still get majority of processor time

The Crux

- How can we design a scheduler that both
  - Minimizes response time for interactive jobs
    - Like RR, make the system be responsive to interactive users
  - Minimizes turnaround time without a priori knowledge of CPU burst time
    - Like SJF or STCF, gives preference to shorter processes

Multilevel Feedback Queues (MLFQ)



- One of the most well-known scheduling algorithms
- Used in most commercial OSs: Windows, MacOS, Linux

Fernando José Corbató

ACM Turing Award 1990

MIT CTSS & Multics

Priority

- A priority number is assigned to each process
  - Scheduler chooses a process of higher priority over one with lower priority
  - i.e., selection of tasks other than just using the arrival order
- Static priorities - priority does not change during process's lifetime
  - Adv: easy to implement and low overhead
  - Disadv: not responsive to changes in environment
  - Disadv: lower-priority processes may suffer starvation
- Dynamic priorities
  - Adv: responsive to change
  - Disadv: incur more overhead than static priorities as scheduler needs to determine the priority of all processes when making scheduling decision

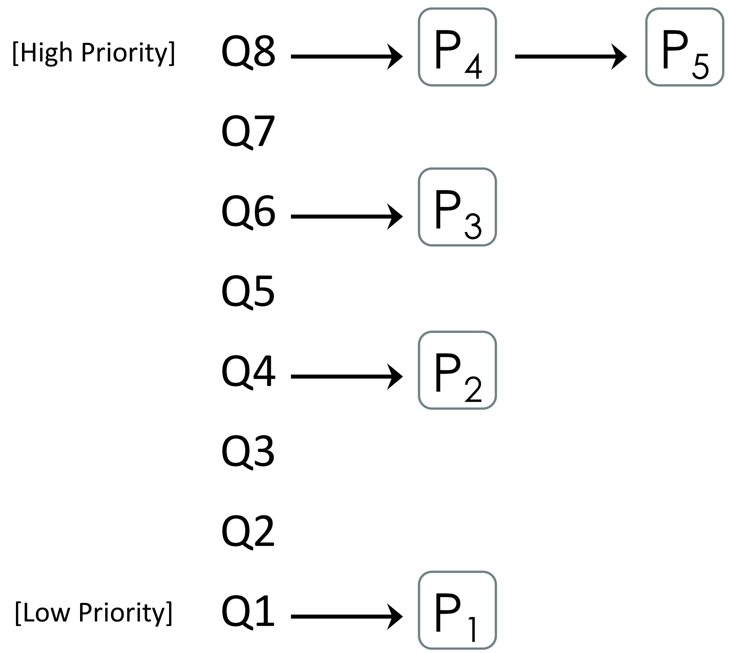
Multilevel Feedback Queues (MLFQ) Scheduling

- Multilevel Feedback Queues
  - The system consists of a number of ready queues, each has a different priority level
  - Scheduler always select a process that appears in the highest priority queue to run first
    - Processes in lower-priority queues will run only when higher-priority queues are empty
  - Within the same priority queue, scheduler selects process using round-robin scheduling

- Principle of assigning priority
  - Using dynamic priority – process's priority is changing
  - MLFQ **varies the priority** of a process based on applications' **runtime characteristic**
    - MLFQ tries to learn about processes as they run, and uses the history of a process to predict its future behavior

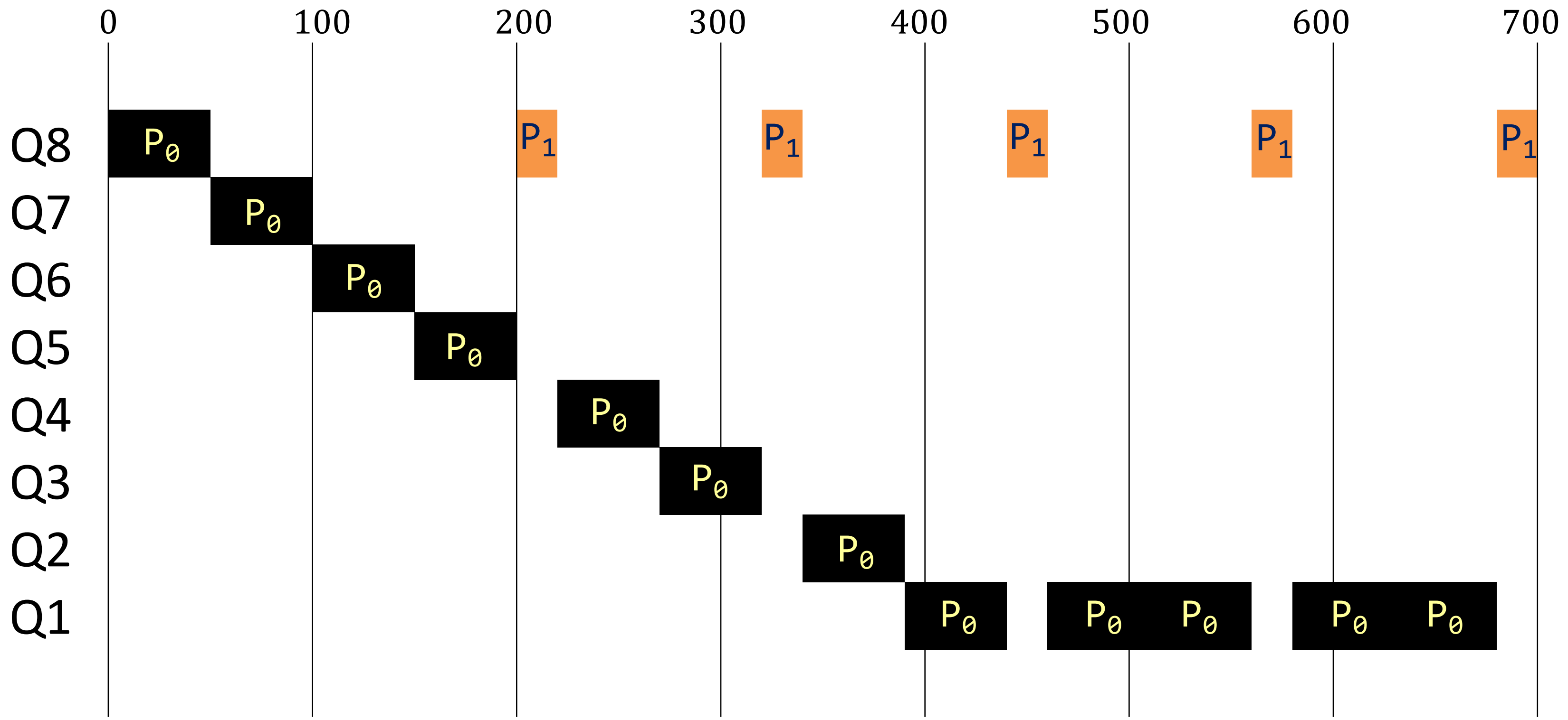
MLFQ – Assigning Priority

- New submitted processes enter the highest-priority queue
- If a process uses up its quantum, it is preempted and positioned at the end of the next lower level queue
  - Long processes repeatedly descend into lower priority levels
- If a process relinquishes the CPU before quantum expires, it stays at the same priority level
  - We don't want to penalize interactive job and keep them at the same priority level



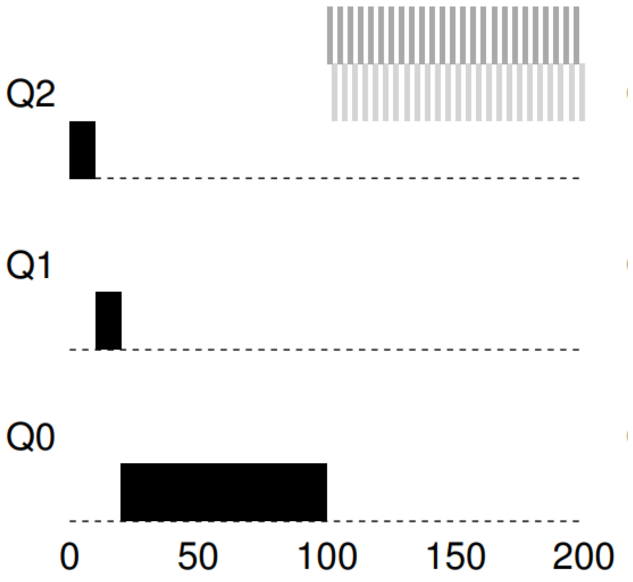
MLFQ – in action

P_0 is a long process and arrives at T = 0
P_1 is an interactive process, which arrives at T = 200 and uses the CPU for 20 time units and blocks for I/O for 100 time units
Assume the time quantum at each level is 50 time units

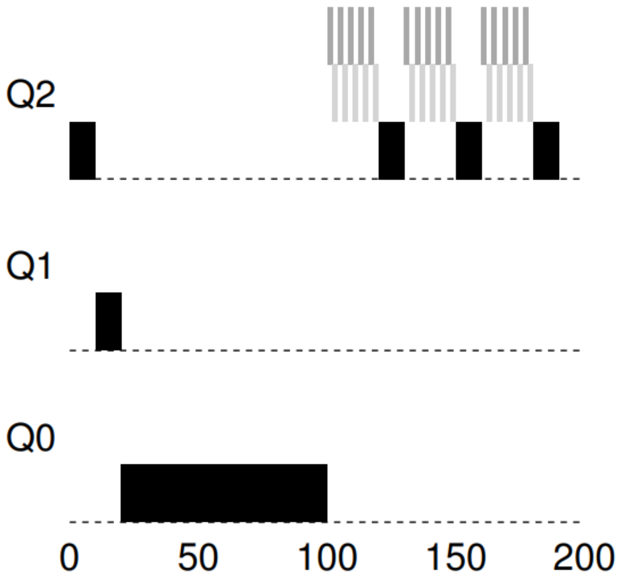


MLFQ - Deficiency

- Process in lower queue can suffer indefinite postponement (starvation)
  - If too many interactive processes, they will eat up CPU time and long processes will be starved
  - A **serious issue** for **priority-based** scheduling scheme (especially with static priority)
- Process may change its behavior from CPU-bound to I/O-bound in its lifetime
  - Unfortunately, a CPU-bound process has descended to the lowest queue; cannot be treated as interactive process



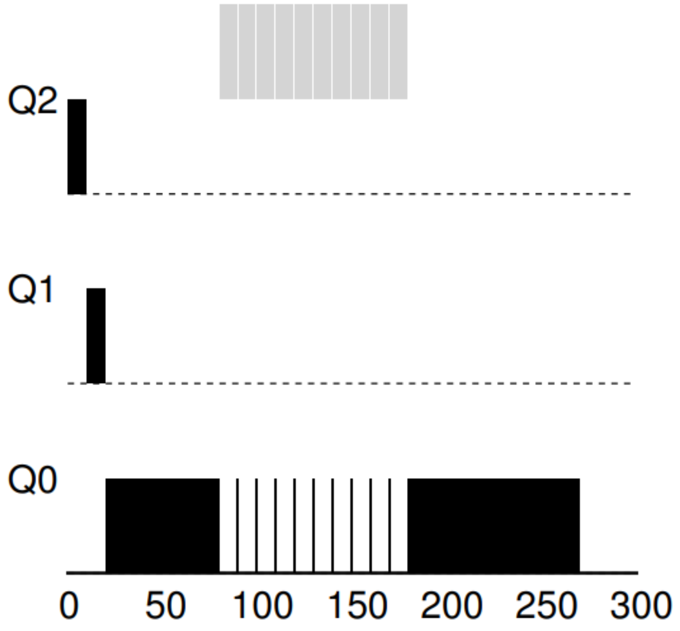
Without Priority Boost



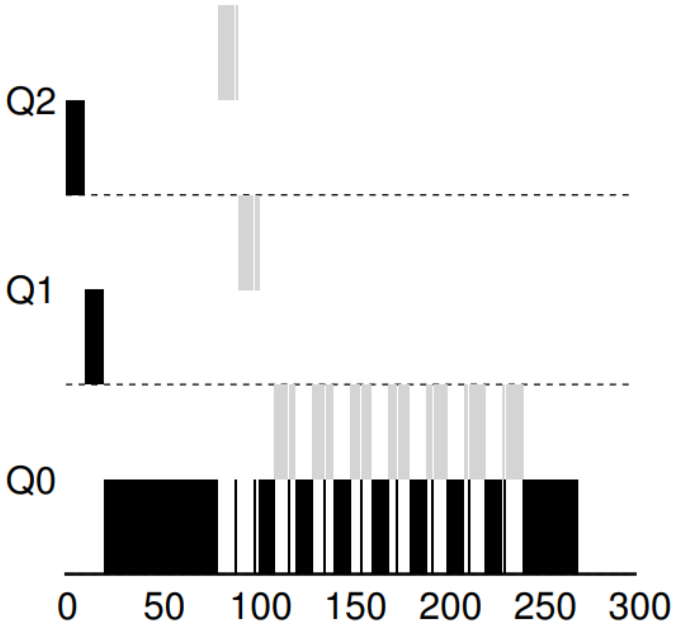
With Priority Boost

MLFQ - Deficiency

- Misbehaved processes can steal lots of CPU time
  - Process can pretend to be a short job by voluntarily relinquishing CPU before expiration of time quantum
  - Just to trick the scheduler



Without Gaming Tolerance



With Gaming Tolerance

MLFQ - Adjustment

- Starvation
  - Periodically boost the priority of all processes in system
  - Example: Every time period S, move all processes to the topmost queue
  - Long processes will get the chance to run AND
  - A CPU-bound process that turns interactive can be treated correctly
- Unfair usage by misbehaved process
  - Memorize a process CPU time at each level
  - When a process has used its time allotment at a given level, it is demoted to the next priority queue
- Different time quantum length for different queues
  - The scheduler increases a process's quantum size as the process moves to each lower-level queue
  - Example: weight = 2^{k-i} (simply double the quantum, where i is current level and k is the no. of priority level)
    - High-priority queues are given short time slices

Multilevel Feedback Queue Summary

- **Rule 1**: If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2**: If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
- **Rule 3**: When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4**: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5**: After some time period S, move all the jobs in the system to the topmost queue.

Multilevel Feedback Queues

- A good example of an adaptive mechanism
  - Responds to changing behavior of the environment
    - Move processes to different queues as they alternate between interactive and batch behavior
  - Incur more overhead but system has increased its sensitivity
- Refined over time and used in modern operating systems

- Parameterize MLFQ in real implementations
  - How many queues?
  - How big time slice per queue?
  - How often for priority boost?

Fair Share Scheduling

- Previous scheduling policies apply to individual processes
  - Consider an application may spawn many processes which gives this application more CPU attention than others. Would that be fair?
  - The same issue happens when an user starts many application processes and gets more CPU share
  - How about a group of students running many processes and affects others using their share?
- Fair Share Scheduling supports scheduling decisions based on process groups
  - Users (or groups) are assigned a weighting that **indicates their share** of system resource as a fraction of the total resource capacity
  - FSS tries to **monitor resource usage** to give fewer resources to users (or groups) who have had more than their fair share and more to those who have had less than their fair share

Lottery Scheduling

- Implementing the fair sharing concept by using **random** allocation
  - On long run, the resource consumption rates of users (or groups) are proportional to the relative shares that they are allocated
- Lottery **tickets** are distributed to all processes (or users or groups)
  - On the basis of their fair share of CPU time
    - A process/user/group is given 10 tickets out of 100 if its fair share is 10 percent
- When scheduling
  - A lottery ticket is chosen at RANDOM
  - The process holding the winning ticket is allocated the CPU
- Why is fair?
  - More important processes can be given extra tickets to increase their odds of winning
    - The more tickets a process has, the higher the chance
  - On long run, a process holding a fraction of *f* of the tickets will get about a fraction of *f* of the CPU resource

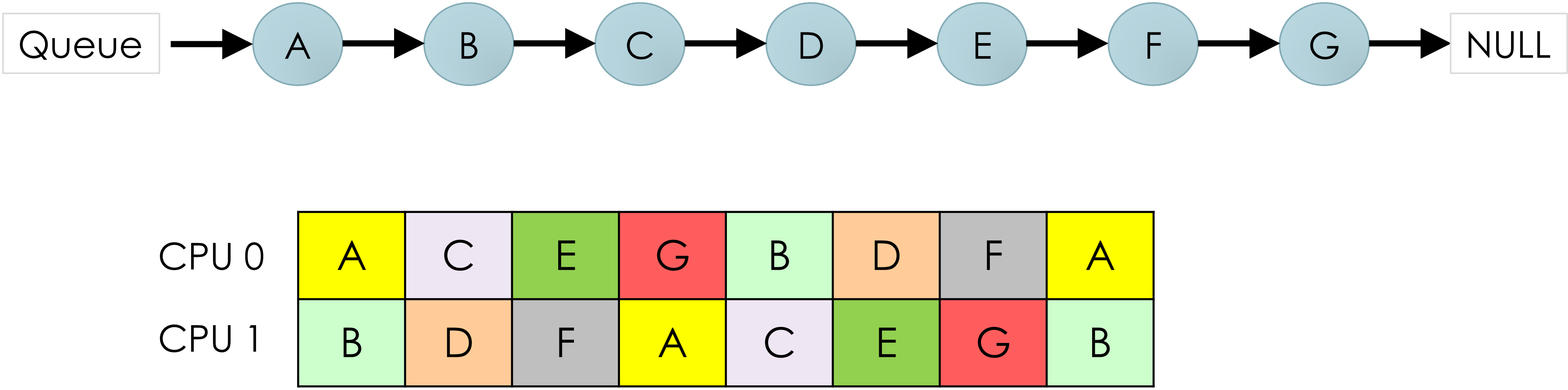
Stride Scheduling

- Fair share: Probabilistic → Deterministic
- Two states of each process:
  - **stride**: Inverse to its ticket
  - **pass**: a counter increased by stride every time it runs
- Scheduling:
  - Pick the process to run that has the lowest pass value
  - Increase its pass by its stride when it is scheduled

- Stride vs. Lottery Scheduling
  - Global states
  - New/blocked processes

Multiprocessor Scheduling

- How to schedule jobs on multicore machines?
- Do the same old techniques work?
  - e.g., A single-queue scheduler for all cores

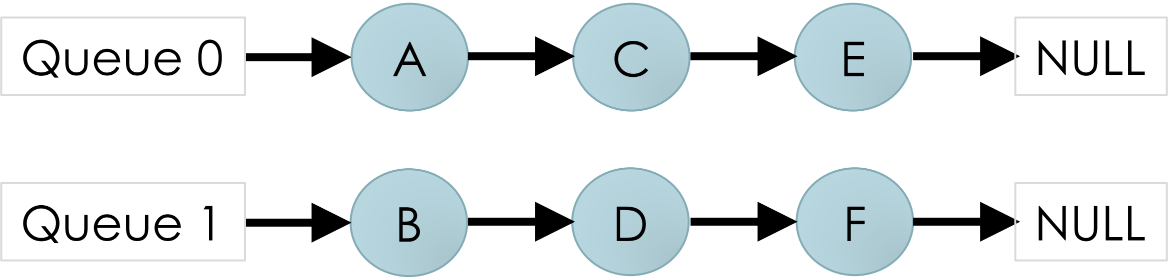


Single-queue Multi-Core Scheduling

- Deficiencies
  - Poor scalability
    - To avoid the concurrency issue, access to the queue structure must be guarded
    - ↑ cores ⇨ ↑ contention ⇨ ↓ performance
  - Cache affinity
    - A process when runs on a core would have a fair bit of state in the caches of the CPU
    - It would be advantageous to run the process on the same core
    - With single-queue scheduler, a job may end up bouncing around from core to core, and it has to reload the state each time it runs!!!

Multi-Queue Scheduling

- One scheduling queue per core
  - When jobs enter the system, they are allocated to exactly one queue according to some heuristic
    - e.g., random or the queue with less jobs
- Pros
  - Less concurrency issue
  - Provides cache affinity



CPU 0	A	C	E	A	C	E	A	C
CPU 1	B	D	F	B	D	F	B	D

Multi-Queue Scheduling

- Major issue – Load imbalance
  - What if Q1 has one job and Q2 has 3 jobs

CPU 0	A	A	A	A	A	A	A	A
CPU 1	B	D	F	B	D	F	B	D

- Solution – Migration of jobs using work stealing approach
  - A queue that is low on jobs occasionally peeks at another queue
  - If other queue has more jobs, it steals one or more jobs from that queue
  - How much jobs is considered as low?
    - Finding the right threshold remains a black art

Case Study
Linux Scheduler

Not for examination

Linux Scheduling

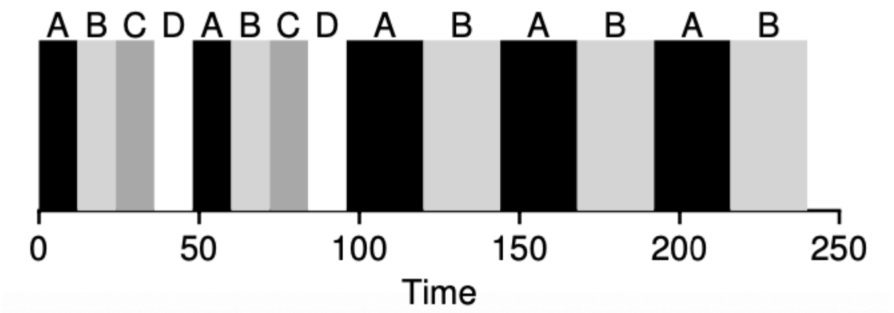
- A preemptive, priority-based algorithm
- There are two separate priority classes:
  - Real-time and non-real-time (normal)
- Real-time priority: 1 (low) to 99 (high)
  - SCHED_FIFO: First-in-first-out real-time process
    - Run until its exits, sleeps, blocks, or is preempted by another newly arrived higher priority process
  - SCHED_RR: Round-robin real-time process
    - Using RR scheme on real-time processes with the same priority
  - Real-time processes can indefinitely postpone other processes, so normal users cannot create a real-time process

Linux Scheduling – Completely Fair Scheduler (CFS)

- For non-real-time class from kernel 2.6.23 and after
- This scheme does not label a process as interactive or CPU-bound; every process looks the same initially
- The main idea behind the CFS is to **maintain fairness** in providing processor time to tasks

- Fairness
  - Divide the CPU resource fairly by the available runnable processes
    - If currently has N processes, each gets 1/N units of CPU time ← **Dynamic time quantum**!
  - Priority is given by the weighting factor called **niceness**
    - 8 processes of 1 unit weighting and 1 process of 2 units of weighting
    - each process has 1/10 units of CPU time except the higher priority process which receives 2/10 units

CFS Basic Operation



- Upon scheduling, pick the process with lowest **vruntime**
  - **Virtual runtime**: a process's accumulated (real) runtime
- When to schedule (stop one process and run next)?
  - Tradeoff fairness vs. performance
  - **sched_latency** / N (# of **runnable** processes): e.g., **48 ms** / 4 =12 ms
  - max(sched_latency / N, **min_granularity**): e.g., 48 ms/20=2.4ms < **6ms**
- **Niceness**: process priority, ranges [-20, 19] and defaults 0

$$\text{time\_slice} = \frac{w_i}{\sum_{j=0}^{N-1} w_j} \times \text{sched\_latency}$$

$$\text{vruntime}_i \mathrel{+}= \frac{w_0}{w_i} \times \text{runtime}_i$$

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

Linux Scheduling – CFS

- Deal with New/Blocked Processes
  - A new process? A process that sleeps for a long time?
  - Set vrutime = min(vruntime_i). ← *Perfect??*
- Efficiency: Red-Black Tree
  - Use a time-ordered Red-Black tree to order runnable (ready) tasks
    - Red-Black Tree - Balance binary tree of $O(\log_2 n)$ time complexity of operations
  - By order the tree nodes by their virtual runtime values, CFS can quickly locate the process with the "gravest CPU need", which is the leftmost node in the tree

- Many other features to make it the most widely used fair-share scheduler in existence today.

Linux Multiprocessor Schedulers

- Three different schedulers
  - Completely Fair Scheduler
    - Proportional share, multi-queue
  - O(1) Scheduler
    - Priority-based (like MLFQ), multi-queue
  - BF Scheduler
    - Proportional share, single-queue
- No common solution (yet)

Summary

- Scheduling policies (algorithms)
  - Decide when and for how long each process runs
- Make choices about
  - Preemptibility and time quantum – These affect the responsiveness
  - Priority – how to apply priority to processes and how to enforce priority
  - Turnaround time – the smaller the better
  - Fairness – No process should suffer starvation
- MLFQ is a good example of a system that learns from the past to predict the future
- FSS monitors resource usage and try to provide a guarantee that each process obtains a certain percentage of CPU time
- Modern OSs use one queue per core to provide cache affinity but load balancing is an issue

Operating Systems

- **Virtualization**
  - CPU Virtualization
    - Process Abstract
      - Address space
      - Process states
      - Process control block
      - Process operations API
      - Signals
    - Limited Direct Execution
      - System calls
      - Context switch
      - Interrupts
    - Scheduling
      - Scheduling metrics
      - FIFO, SJF, HRRN, STCF, RR, MLFQ
      - Multi-core scheduling, Linux CFS
  - Memory Virtualization
    - Address space
    - Address translation: dynamic relocation
    - Segmentation
    - Paging
    - TLB
    - Multi-level paging
    - Inverted page table
    - Swap space
    - Page replacement policy: FIFO, LFR, LRU, Clock
    - Thrashing
- **Concurrency**
  - Thread
    - POSIX threads (pthreads)
    - Race conditions, critical sections, mutual exclusion, atomic operations, synchronization
  - Locks
    - Atomic instructions: test-and-set, compare-and-swap
    - Mutex locks
  - Condition Variables
    - Pthread CVs
    - Producer-Consumer problem
  - Semaphores
    - Binary Semaphores
    - Counting Semaphores
    - Ordering
    - Readers-Writers problem
  - Deadlock
    - Dining philosophers' problem
    - Four necessary conditions
    - Deadlock prevention, avoidance, detection&recovery
- **Persistence**
  - I/O devices (HDD, SSD)
  - Files and Directories
    - Inode
    - File descriptor
    - Hard/Symbolic links
  - File System Implementation
    - On-disk data structure
      - Superblock, Bitmap, Inodes, Data blocks
    - Free space management
      - Bitmap, linked-list, block-list
    - Caching and buffering
    - Access control and protection
    - Journaling file system
      - Data journaling
      - Metadata journaling
- ***Advanced Topics***