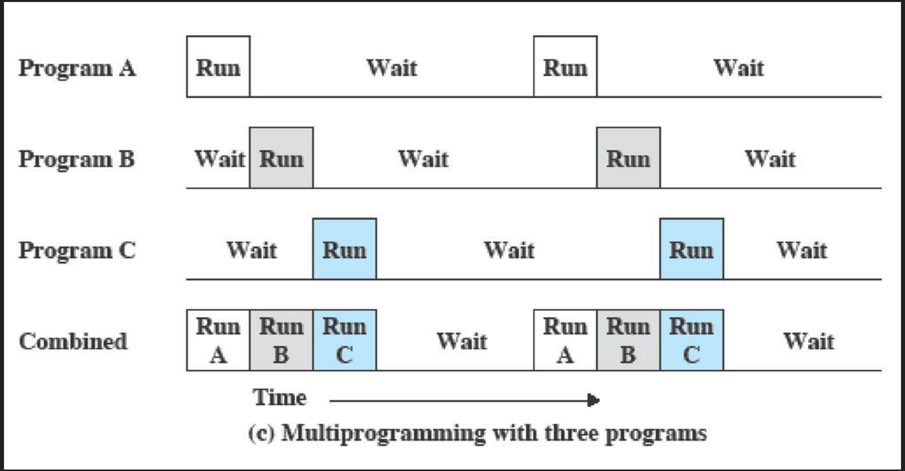
05 - CPU Scheduling

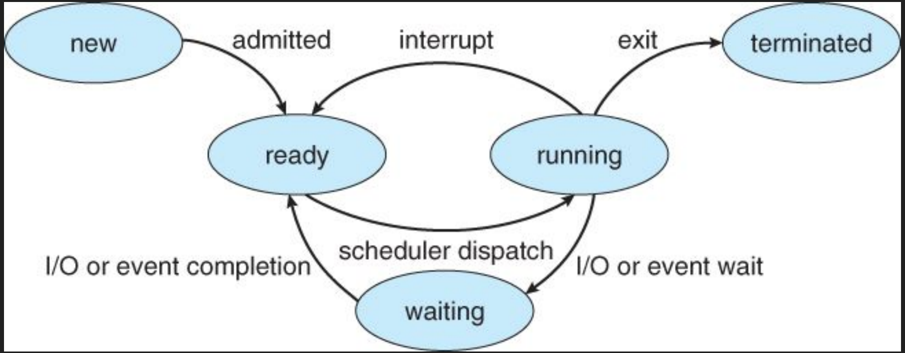
CEG 4350/5350 Operating Systems Internals and Design
Max Gilson

Scheduling Processes

- Multiprocessing allows us to better utilize the CPU and execute multiple processes “at once”
  - In reality, only one process is running on the CPU at a time
  - The OS switches between multiple processes very quickly
  - If our hardware has multiple CPU cores we can execute multiple processes simultaneously
- If a process needs to wait for I/O it can give another process a chance to run
- All of our processes reside in a state queue
  - i.e. all ready processes live inside a “ready” queue that the OS will pull from when choosing to execute a new process



(c) Multiprogramming with three programs



Scheduling Processes (cont.)

- Long Term Scheduling
  - How many processes will the OS allow to exist
  - We are limited by memory so we can’t have an infinite amount
- Short Term Scheduling
  - How should the OS select a “ready” process from the ready queue

Short Term Scheduling

- The process scheduler resides in the kernel and can make a decision when:
  - A process switches from running to waiting
    - i.e. if the process wants to access I/O
  - An interrupt occurs
    - i.e. if a process wants the keyboard data and the keyboard sends and interrupt
  - A processes is created or terminated
- Non-preemptive scheduling method must wait for one of the above events to occur before switching processes
- Preemptive scheduling method allows the scheduler to interrupt a process

What Makes a Good Scheduling Algorithm?

- CPU Utilization
  - How much is the CPU being utilized to its fullest potential?
  - Ideal: as close to 100% as possible
- I/O Utilization
  - How much of our I/O or storage is being used to its fullest potential?
  - Ideal: as close to 100% as possible
- Throughput
  - How fast are the processes completing?
  - Ideal: very fast
- Turnaround Time
  - How much time are processes taking to complete from “new” to “terminated”?
  - Ideal: very short
- Waiting Time
  - How much time do processes stay in the “ready” queue?
  - Ideal: very short
- Response Time
  - How much time between a process being “ready” and its next I/O request?
  - Ideal: very short

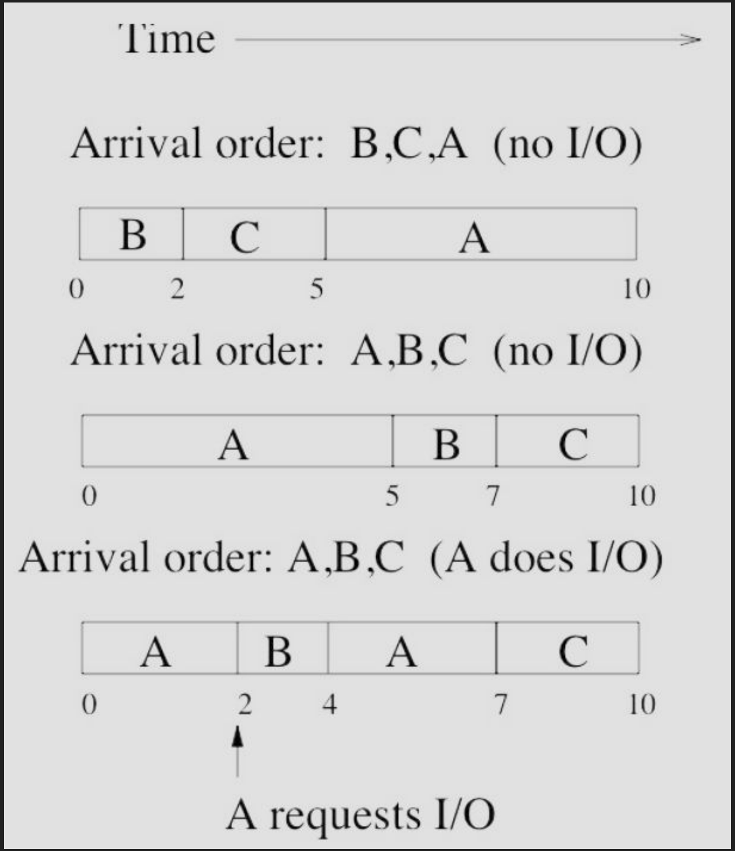
Scheduling Policies

- Ideally, you want a process scheduler to optimize all criteria but this is not realistic
- Instead, choose a scheduler that optimizes your most important metric(s):
    - Shortest response time
        - Good for when the user has to interact with the system (i.e. moving the mouse, typing on keyboard, loading screens, etc.)
    - Lowest response time *variance*
        - Having a consistent response time might make the system less frustrating to work with
    - Maximize throughput
        - Minimize OS overhead, context switching overhead, and efficiently use I/O and resources
    - Minimize waiting time
        - Give each process the same amount of CPU time but this may increase response time

First In First Out (FIFO)

- The scheduler will execute processes <u>in the order in which they are created or arrive</u>
  - Not necessarily the order that they are added to the ready queue
- Assume processes only release the CPU when they are waiting for I/O
- Notice, A still gets to finish before C
  - Even though A had to wait for I/O, it still arrived before C
- This method is <u>cooperative</u> (requires process to release the CPU)
- Advantage: simple to implement
- Disadvantage: process’s time spent waiting is highly variable
- Disadvantage: I/O bound processes are not prioritized and forced to wait for CPU bound processes

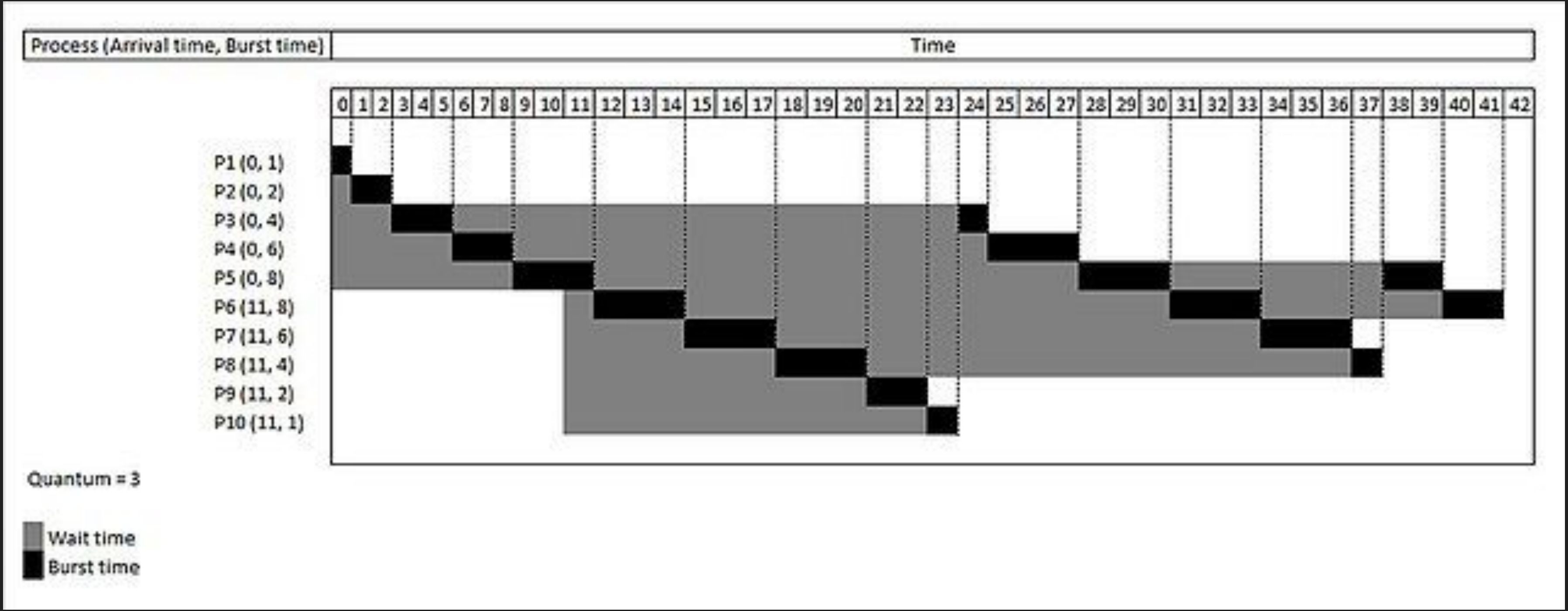
First In First Out (FIFO) (cont.)


Time
Arrival order: B,C,A (no I/O)
B C A
0 2 5 10
Arrival order: A,B,C (no I/O)
A B C
0 5 7 10
Arrival order: A,B,C (A does I/O)
A B A C
0 2 4 7 10
A requests I/O

Round Robin (RR)

- The scheduler schedules processes evenly or fairly giving each process a *quanta* of time to run on the CPU
  - The quantum is a predefined slice of time (e.g. 100 clock cycles)
  - The quanta is how many of these slices a process may use at a maximum (e.g. 3 quanta max. = 300 clock cycles max. CPU time)
- This method is <u>preemptive</u> (uses a clock to force processes to stop)
- Quanta too large - processes spend way too much time waiting
- Quanta too small - most of your time is spent context switching
  - Try to find a quanta where context switching is 1% of the total quanta
- Advantage: Consistent response time, all processes have equal time to access CPU
- Disadvantage: Process’s average time spent waiting can be long (system may feel slow with 100’s of processes running)

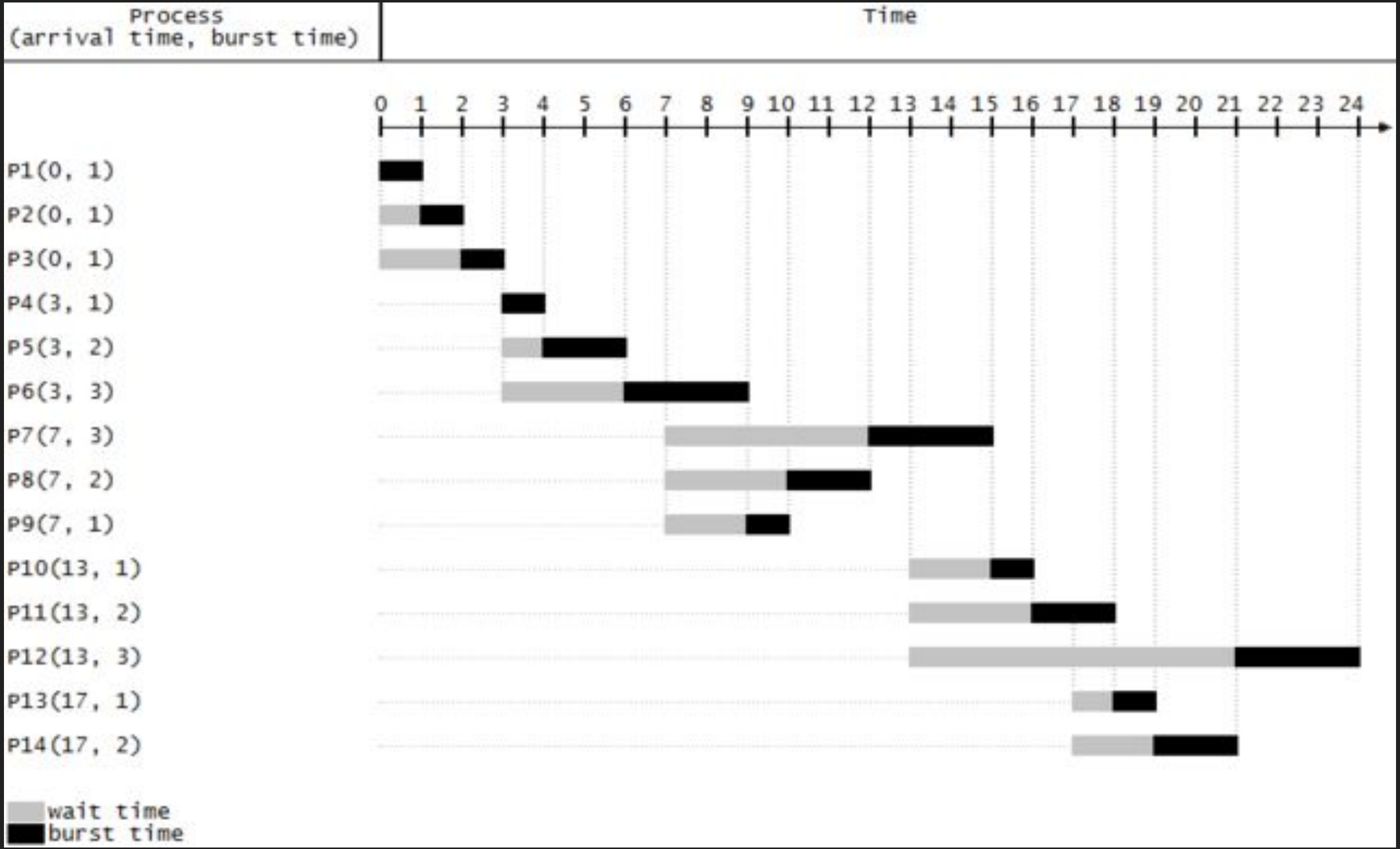
Round Robin (RR) (cont.)



Shortest Process Next (SPN)

- The scheduler schedules the process with the least (expected) amount of work (CPU time) to run until the process has an I/O request or terminates
- Can be preemptive or cooperative
  - Preemptive version uses shortest remaining time first
  - Preemptive version prioritizes I/O bound jobs over CPU bound jobs
- Advantage: Optimally minimizes (on average) process’s waiting time
  - This is an optimal solution for this metric
- Disadvantage: impossible to predict how long a process needs to run
- Disadvantage: long processes may never get a chance to run

Shortest Process First (cont.)

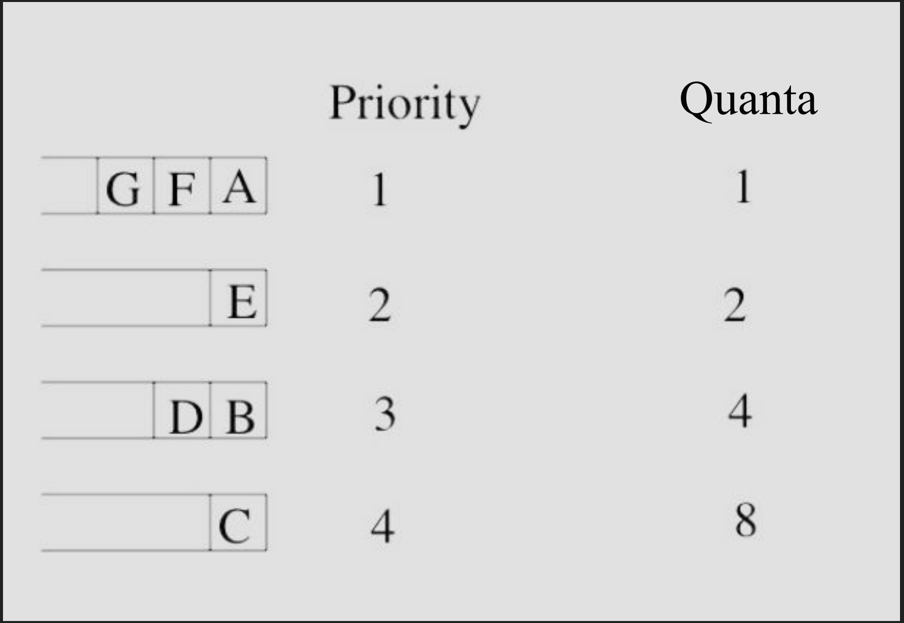


Multilevel Feedback Queue (MLFQ)

- The scheduler schedules based on past behavior of the process to attempt to predict the future and assign process priorities
- Used in most modern UNIX like systems
  - Overcomes the limitations of preemptive SPN
- If a process was I/O bound in the past, it is likely to be I/O bound in the future
- The scheduler can favor jobs that have used the least amount of CPU time (I/O bound processes), thus approximating SPN
- This policy is adaptive because it adapts to how the processes run and changes it scheduling behavior based on this past history
- The kernel keeps track of how often a process waits for I/O and prioritizes the processes that often access I/O

Multilevel Feedback Queue (MLFQ) (cont.)

- Multiple queues with priority based on predicted run time
- Use RR scheduling for each priority queue
  - Once finished, run the next priority level queue with RR
    - This can lead to starvation!
- Increase RR quanta exponentially for each priority level
  - This gives CPU bound processes more time to get stuff done



Multilevel Feedback Queue (MLFQ) (cont.)

- New processes start in the highest priority queue
- If the process wants to exceed its quanta, decrease the process’s priority level by 1
  - The process is using more CPU time than expected
- If the process does not exceed its quanta, increase the priority level by 1 up to the highest priority level
  - The process is using less CPU time than expected
  - This can happen if the process begins waiting for I/O very quickly
- I/O bound jobs become higher priority
- CPU bound processes become lower priority

Improving Fairness

- Since SPN is optimal, but unfair and can starve long processes, increasing fairness must increase the waiting time
- Possible solutions:
    - Give each queue a fraction of CPU time
        - This is only fair if priority level queues have the same number of processes
    - Adjust the priority of processes if they are not getting ran
        - Unix originally took this approach
        - Avoids starvation but waiting time suffers when system is overloaded
            - This is because every process becomes high priority!

Lottery Scheduling

- Give each process a set of tickets
    - Assign more tickets to short running processes
    - Assign fewer tickets to long running processes
- Each quantum, randomly pick a winning ticket
- On average, CPU time is proportional to the number of tickets given to a process
- This approximates SPN while avoiding starvation since every process has a ticket that can be picked to run
- As the CPU load changes, adding or removing processes affects all other processes proportionately
    - Regardless of how many tickets each process has

Lottery Scheduling Performance

- In the example to the right, assume:
  - Short jobs get 10 tickets
  - Long jobs get 1 ticket
- How do we know how long the processes run?
  - Look at past history and estimate
- How do we determine how many tickets to give out?
  - Let the user decide (basically allows user to choose priority)
  - Let the OS determine based on the time

# of short processes / # of long processes	% of CPU each short process gets	% of CPU each long process gets
1/1	91% (10/11)	9% (1/11)
0/2	N/A	50% (1/2)
2/0	50% (10/20)	N/A
10/1	~10% (10/101)	< 1% (1/101)
1/10	50% (10/20)	5% (1/20)

Scheduling Algorithm Summary

- FIFO
  - Not fair, and average waiting time is poor
- Round Robin
  - Fair, response time variance minimized, but average waiting time is poor
- SPN
  - Not fair, but average waiting time is minimized assuming we can accurately predict the length of the next CPU burst
  - Starvation is possible
- Multilevel Queuing
  - An implementation (approximation) of SJF.
- Lottery Scheduling
  - Fairer with a low average waiting time, but less predictable.