## Lecture 7: Scheduling

- definitions, assumptions
- scheduling policy goals and metrics
- scheduling algorithms:
  - First Come First Served (FCFS)
  - Round Robin (RR)
  - Shortest Job First (SJF)
  - Shortest Remainder First (SRF)
  - priority
  - multilevel queue
  - multilevel feedback queue
- Unix scheduling

## Types of CPU schedulers

- The *CPU scheduler* (sometimes called the *dispatcher* or *short-term scheduler*): selects a process from the ready queue and lets it run on the CPU
- Types of schedulers*:*
  - Non-preemptive
    - Executes only when:
      - Process is terminated
      - Process switches from running to blocked
    - simple to implement but unsuitable for time-sharing systems
  - Preemptive
    - Executes at times above, also when:
      - Process is created
      - Blocked process becomes ready
      - A timer interrupt occurs
    - More overhead, but keeps long processes from monopolizing CPU
    - Must not preempt OS kernel while it's servicing a system call (e.g., reading a file) or otherwise OS may end up in an inconsistent state

## Scheduling: assumptions

- Assumptions:
  - Assumes all processes are in memory (swapping and process admission is handled by medium-term scheduler), and one of those is executing on the CPU
  - One process per user
  - One thread per process
  - Processes are independent, and compete for resources (including the CPU)
- Processes the following cycle:
  - compute for a while (on CPU)
  - wait for some some I/O
- Processes can be loosely classified into:
  - CPU-bound — does mostly computation (long CPU burst), and very little I/O
  - I/O-bound — does mostly I/O, and very little computation (short CPU burst)

## CPU scheduling policy

- CPU scheduler must decide:
  - How long a process executes (if preemptive)
  - In which order processes ready to run will execute
- User-oriented scheduling policy goals (behavior of the system as perceived by the user):
  - Minimize average *response time* (time from request received until response starts)
  - Minimize *turnaround time* (time from process start until completion) - Execution time plus waiting time (in ready queue)
  - Minimize variance of average response time
    - Predictability is important
    - Process should always run in (roughly) same amount of time regardless of the load on the system

## CPU scheduling policy (cont.)

- System-oriented scheduling policy goals:
  - Performance related:
    - Maximize *throughput* (number of processes that complete in unit time)
    - Maximize processor utilization (percentage of time CPU is busy)
  - Non-performance related:
    - *Fairness* — in the absence of guidance from the user or the OS, processes should be treated the same, and no process should suffer *starvation* (being infinitely denied service)
      May have to be less fair in order to minimize average response time!
    - Balance resources — keep all resources of the system (CPU, memory, disk, I/O) busy
      Favor processes that underuse stressed resources

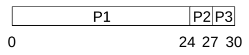
## First Come First Served (FCFS)

- Policy:
  - Choose process from ready queue in the order of its arrival, and run that process non-preemptively
    - Early FCFS schedulers were overly non-preemptive: the process did not relinquish the CPU until it was finished, even when it was doing I/O
    - Now, non-preemptive means the scheduler chooses another process when the first one terminates or blocks
- Implement using FIFO queue (add to tail, take from head)
- Used in Nachos (as distributed)

## FCFS examples

Example 1

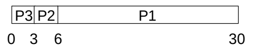
| Process (Arrival Order) | P1 | P2 | P3 |
|---|---|---|---|
| **Burst Time** | 24 | 3 | 3 |
| **Arrival Time** | 0 | 0 | 0 |



average waiting time =
(0 + 24 + 27) / 3 = 17

Example 2

| Process (Arrival Order) | P3 | P2 | P1 |
|---|---|---|---|
| **Burst Time** | 3 | 3 | 24 |
| **Arrival Time** | 0 | 0 | 0 |



average waiting time =
(0 + 3 + 6) / 3 = 3

## FCFS evaluation

- Non-preemptive
- Response time — slow if there is a large variance in process execution times
  - If one long process is followed by many short processes, short processes have to wait a long time
  - If one CPU-bound process is followed by many I/O-bound processes, there's a "convoy effect"
    - Low CPU and I/O device utilization
- Throughput — not emphasized
- Fairness —penalizes short processes and I/O bound processes
- Starvation — not possible
- Overhead — minimal

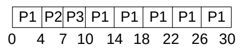
## Round-robin scheduling (RR)

- Preemptive version of FCFS
- Policy:
  - Define a fixed *time slice* (also called a *time quantum*)
  - Choose process from head of ready queue
  - Run that process for at most one time slice, and if it hasn't completed by then, add it to the tail of the ready queue
  - If that process terminates or blocks before its time slice is up, choose another process from the head of the ready queue, and run that process for at most one time slice…
- Implement using:
  - Hardware timer that interrupts at periodic intervals
  - FIFO ready queue (add to tail, take from head)

## RR examples

Example 1 time slice = 4

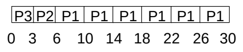
| Process (Arrival Order) | P1 | P2 | P3 |
|---|---|---|---|
| **Burst Time** | 24 | 3 | 3 |
| **Arrival Time** | 0 | 0 | 0 |



average waiting time=
(4 + 7 + (10–4)) / 3 = 5.66

Example 2

| Process (Arrival Order) | P3 | P2 | P1 |
|---|---|---|---|
| **Burst Time** | 3 | 3 | 24 |
| **Arrival Time** | 0 | 0 | 0 |



average waiting time =
(0 + 3 + 6) / 3 = 3

## RR evaluation

- Preemptive (at end of time slice)
- Response time — good for short processes
  - Long processes may have to wait *n***q* time units for another time slice
    - *n* = number of other processes,
      *q* = length of time slice
- Throughput — depends on time slice
  - Too small — too many context switches
  - Too large — approximates FCFS
- Fairness — penalizes I/O-bound processes (may not use full time slice)
- Starvation — not possible
- Overhead — low

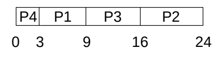
## Shortest Job First (SJF)

- Other names:
  - Shortest-Process-Next (SPN)
- Policy:
  - Choose the process that has the smallest next CPU burst, and run that process non-preemptively (until termination or blocking)
  - In case of a tie, FCFS is used to break the tie
- Difficulty: determining length of next CPU burst
  - Approximation — predict length, based on past performance of the process, and on past predictions

## SJF examples

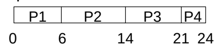
| Process (Arrival Order) | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| **Burst Time** | 6 | 8 | 7 | 3 |
| **Arrival Time** | 0 | 0 | 0 | 0 |

SJF



average waiting time = (0 + 3 + 9 + 16) / 4 = 7

FCFS, same example



average waiting time = (0 + 6 + 14 + 21) / 4 = 10.25

## SJF evaluation

- Non-preemptive
- Response time — good for short processes
  - Long processes may have to wait until a large number of short processes finish
  - Provably *optimal* — minimizes average waiting time for a given set of processes (if preemption is not considered)
- Throughput — high
- Fairness — penalizes long processes
- Starvation — possible for long processes
- Overhead — can be high (recording and estimating CPU burst times)

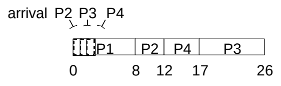
## Shortest Remaining Time (SRT)

- SRT is a preemptive version of SJF
- Policy:
  - Choose the process that has the smallest next CPU burst, and run that process preemptively…
    - (until termination or blocking, or
    - until a process enters the ready queue (either a new process or a previously blocked process))
  - At that point, choose another process to run if one has a smaller expected CPU burst than what is left of the current process' CPU burst

## SRT examples

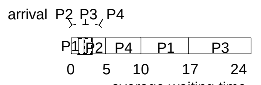
| Process (Arrival Order) | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| **Burst Time** | 8 | 4 | 9 | 5 |
| **Arrival Time** | 0 | 1 | 2 | 3 |

SJF



P1 P2 P3 P4
average waiting time = (0 + (8–1) + (12–3) + (17–2)) / 4 = 7.75

SRT



P1 P2 P3 P4
average waiting time = ((0+(10–1)) + (1–1) + (17–2) + (5–3)) / 4 = 6.5

## SRT evaluation

- Preemptive (at arrival of process into ready queue)
- Response time — good
  - Provably *optimal wait time* — minimizes average waiting time for a given set of processes
- Throughput — high
- Fairness — penalizes long processes
  - Note that long processes eventually become short processes
- Starvation — possible for long processes
- Overhead — can be high (recording and estimating CPU burst times)

## Priority Scheduling

- Policy:
  - Associate a priority with each process
    - Externally defined, based on importance, money, politics, etc.
    - Internally defined, based on memory requirements, file requirements, CPU requirements vs. I/O requirements, etc.
    - SJF is priority scheduling, where priority is inversely proportional to length of next CPU burst
  - Choose the process that has the highest priority, and run that process either:
    - preemptively, or
    - non-preemptively
- Evaluation
  - Starvation — possible for low-priority processes
    - Can avoid by *aging* processes: increase priority as they spend time in the system

## Multilevel queue scheduling

- Policy:
  - Use several ready queues, and associate a different priority with each queue
  - Choose the process from the occupied queue that has the highest priority, and run that process either:
    - preemptively, or
    - non-preemptively
  - Assign new processes permanently to a particular queue
    - Foreground, background
    - System, interactive, editing, computing
  - Each queue can have a different scheduling policy
    - Example: preemptive, using timer
      - 80% of CPU time to foreground, using RR
      - 20% of CPU time to background, using FCFS
- Problem: processes at low level queues may starve

## Multilevel feedback queue

- Policy:
  - Use several ready queues, and associate a different priority with each queue
  - Choose the process from the occupied queue with the highest priority, and run that process either:
    - preemptively, or
    - non-preemptively
  - Each queue can have a different scheduling policy
  - Allow scheduler to move processes between queues
    - Start each process in a high-priority queue; as it finishes each CPU burst, move it to a lower-priority queue
    - Aging — move older processes to higher-priority queues
    - Feedback = use the past to predict the future — favor jobs that haven't used the CPU much in the past   close to SRT!

## Unix CPU Scheduling

- Policy:
  - Multiple queues (32), each with a priority value - 0-127 (low value = high priority):
    - Kernel processes (or user processes in kernel mode) the lower values (0-49) - kernel processes are not preemptible!
    - User processes have higher value (50-127)
  - Choose the process from the occupied queue with the highest priority, and run that process preemptively, using a timer (time slice typically around 100ms)
    - Round-robin scheduling in each queue
  - Move processes between queues
    - Keep track of clock ticks (60/second)
    - Once per second, add clock ticks to priority value
    - Also change priority based on whether or not process has used more than it's "fair share" of CPU time (compared to others)
  - users can decrease priority