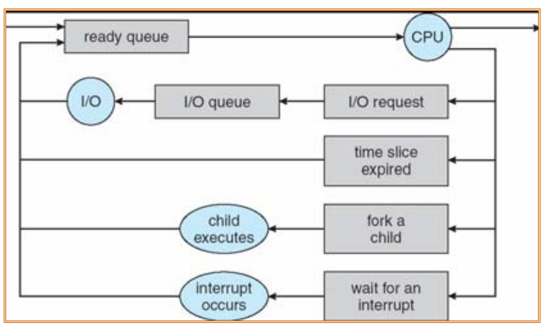
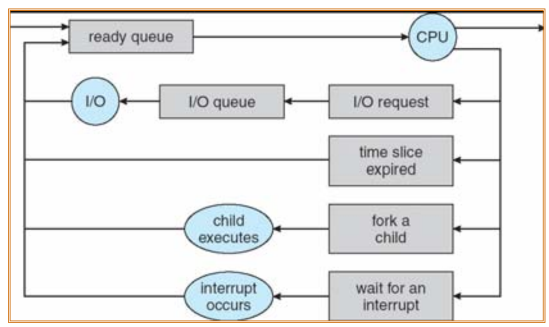
CPU Scheduling

Announcements

The scheduling big picture



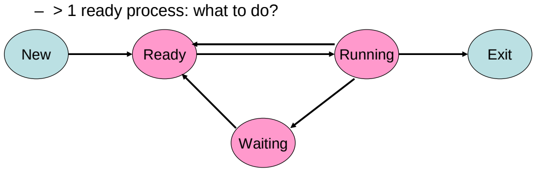
Today: how to schedule processes/threads in the ready queue



Process Scheduling

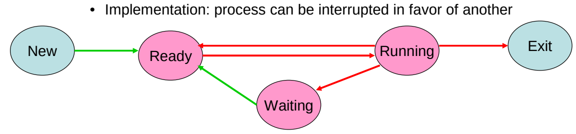
- “process” and “thread” used interchangeably

- Many processes in “ready” state
- Which ready process to pick to run on the CPU?
  - 0 ready processes: run idle loop
  - 1 ready process: easy!
  - > 1 ready process: what to do?



When does scheduler run?

- Non-preemptive minimum
  - Process runs until voluntarily relinquish CPU
    - process blocks on an event (e.g., I/O or synchronization)
    - process terminates
- Preemptive minimum
  - All of the above, plus:
    - Event completes: process moves from blocked to ready
    - Timer interrupts
    - Implementation: process can be interrupted in favor of another



Process Model

- Process alternates between CPU and I/O bursts
  - CPU-bound jobs: Long CPU bursts

  - I/O-bound: Short CPU bursts

  - I/O burst = process idle, switch to another "for free"
  - Problem: don't know job's type before running
- An underlying assumption:
  - "response time" most important for interactive jobs (I/O bound)

Scheduling Evaluation Metrics

- Many quantitative criteria for evaluating sched algo**:**
  - CPU utilization: percentage of time the CPU is not idle
  - Throughput: completed processes per time unit
  - Turnaround time: submission to completion
  - Waiting time: time spent on the ready queue
  - Response time: response latency
  - Predictability: variance in any of these measures
- The right metric depends on the context

"The perfect CPU scheduler"

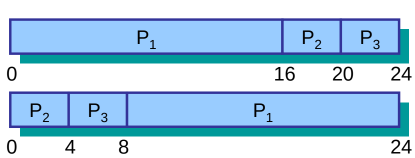
- Minimize latency: response or job completion time
- Maximize throughput: Maximize jobs / time.
- Maximize utilization: keep I/O devices busy.
  - Recurring theme with OS scheduling
- Fairness: everyone makes progress, no one starves

Problem Cases

- Blindness about job types
  - I/O goes idle
- Optimization involves favoring jobs of type "A" over "B".
  - Lots of A's? B's starve
- Interactive process trapped behind others.
  - Response time sucks for no fundamental reason
- Priorities: A's priority > B's.
  - B never runs

Scheduling Algorithms FCFS

- First-come First-served (FCFS) (FIFO)
  - Jobs are scheduled in order of arrival
  - Non-preemptive
- **Problem:**
  - Average waiting time depends on arrival order



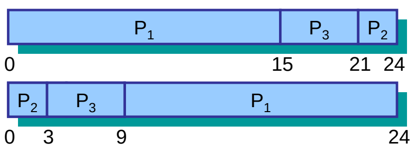
- **Advantage:** really simple!

Convoy Effect

- A CPU bound job will hold CPU until done,
  - or it causes an I/O burst
    - rare occurrence, since the thread is CPU-bound
  - ⇒ long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization
- Example: one CPU bound job, many I/O bound
    - CPU bound runs (I/O devices idle)
    - CPU bound blocks
    - I/O bound job(s) run, quickly block on I/O
    - CPU bound runs again
    - I/O completes
    - CPU bound still runs while I/O devices idle (continues…)
  - Simple hack: run process whose I/O completed?
    - What is a potential problem?

Scheduling Algorithms: SJF

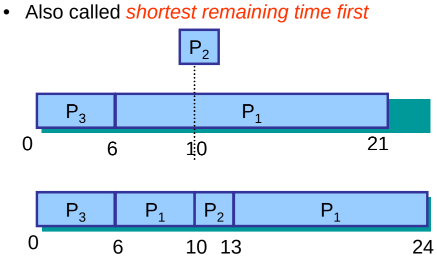
- **Shortest Job First (SJF)**
  - Choose the job with the shortest next CPU burst
  - Provably optimal for minimizing average waiting time



- **Problem:**
  - Impossible to know the length of the next CPU burst

Scheduling Algorithms: Preemptive SJF

- SJF can be either preemptive or non-preemptive
  - New, short job arrives; current process has long time to execute
- Also called *shortest remaining time first*



Shortest Job First Prediction

- Approximate next CPU-burst duration
  - from the durations of the previous bursts
    - The past can be a good predictor of the future
- No need to remember entire past history
- Use exponential average:

  t_n duration of the n^{th} CPU burst

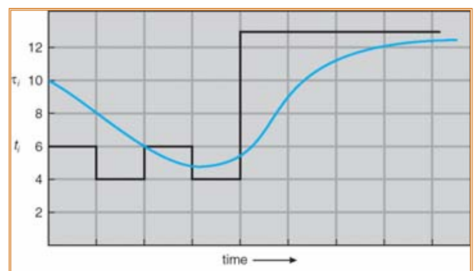
  τ_{n+1} predicted duration of the $(n+1)^{st}$ CPU burst

  $$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\, \tau_n$$

  where $0 \le \alpha \le 1$

  α determines the weight placed on past behavior

What is dumb about this picture?



CPU burst (t_i)		6	4	6	4	13	13	13 …
"guess" (τ_i)	10	8	6	6	5	9	11	12 …

Priority Scheduling

- **Priority Scheduling**
  - Choose next job based on priority
  - For SJF, priority = expected CPU burst
  - Can be either preemptive or non-preemptive
- **Problem:**
  - Starvation: jobs can wait indefinitely
- **Solution to starvation**
  - Age processes: increase priority as a function of waiting time

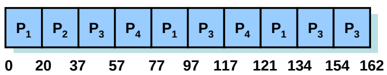
Round Robin

- **Round Robin (RR)**
  - Often used for timesharing
  - Ready queue is treated as a circular queue (FIFO)
  - Each process is given a time slice called a *quantum*
  - It is run for the quantum or until it blocks
  - RR allocates the CPU uniformly (fairly) across participants.
  - If average queue length is n, each participant gets 1/n

RR with Time Quantum = 20

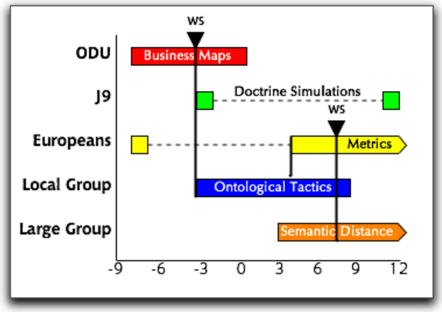
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

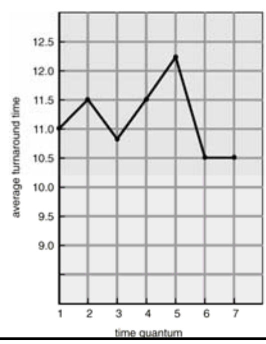


- Higher average turnaround than SJF,
- But better response time

Another Gantt Chart



What is dumb about this picture?



process	time
P_1	6
P_2	3
P_3	1
P_4	7

RR: Choice of Time Quantum

- Performance depends on length of the timeslice
  - Context switching isn't a free operation.
  - If timeslice time is set too high
    - attempting to amortize context switch cost, you get FCFS.
    - i.e. processes will finish or block before their slice is up anyway
  - If it's set too low
    - you're spending all of your time context switching between threads.
  - Timeslice frequently set to ~100 milliseconds
  - Context switches typically cost < 1 millisecond

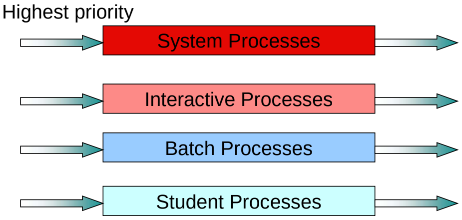
Moral:
Context switch is usually negligible (< 1% per timeslice) unless you context switch too frequently and lose all productivity

Scheduling Algorithms

- Multi-level Queue Scheduling
- Implement multiple ready queues based on job "type"
  - interactive processes
  - CPU-bound processes
  - batch jobs
  - system processes
  - student programs
- Different queues may be scheduled using different algos
- Intra-queue CPU allocation is either strict or proportional
- Problem: Classifying jobs into queues is difficult
  - A process may have CPU-bound phases as well as interactive ones

Multilevel Queue Scheduling

Highest priority

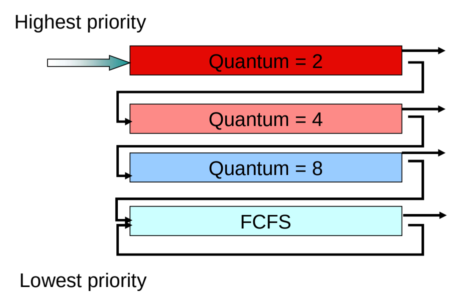


Lowest priority

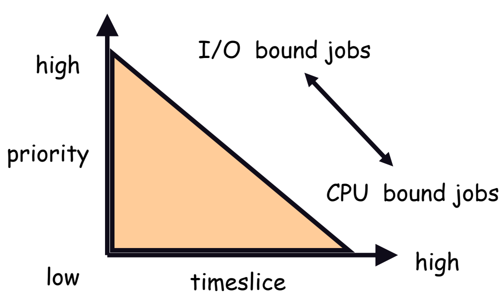
Scheduling Algorithms

- Multi-level Feedback Queues
- Implement multiple ready queues
  - Different queues may be scheduled using different algorithms
  - Just like multilevel queue scheduling, but assignments are not static
- Jobs move from queue to queue based on feedback
  - Feedback = The behavior of the job,
    - e.g. does it require the full quantum for computation, or
    - does it perform frequent I/O ?
- Very general algorithm
- Need to select parameters for:
  - Number of queues
  - Scheduling algorithm within each queue
  - When to upgrade and downgrade a job

Multilevel Feedback Queues



A Multi-level System



Thread Scheduling

Since all threads share code & data segments

- Option 1: Ignore this fact
- Option 2: Gang scheduling
  - run all threads belonging to a process together (multiprocessor only)
  - if a thread needs to synchronize with another thread
    - the other one is available and active
- Option 3: Two-level scheduling:
  - Medium level scheduler
  - schedule processes, and within each process, schedule threads
  - reduce context switching overhead and improve cache hit ratio
- Option 4: Space-based affinity:
  - assign threads to processors (multiprocessor only)
  - improve cache hit ratio, but can bite under low-load condition

Real-time Scheduling

- Real-time processes have timing constraints
  - Expressed as deadlines or rate requirements
- Common RT scheduling policies
  - Rate monotonic
    - Just one scalar priority related to the periodicity of the job
    - Priority = 1/rate
    - Static
  - Earliest deadline first (EDF)
    - Dynamic but more complex
    - Priority = deadline
- Both require admission control to provide guarantees

Actual OS algorithms

- All use preemption
- All have priorities
  - Normally along real-time, interactive, system lines
- All have different time-slice sizes for different priorities
- But the details vary tremendously