CPU Scheduling Overview

• Previously, we have only glossed over the details of which thread is chosen from the ready queue.
• Making this decision is called CPU scheduling.
• In this lecture, we’ll look at:
  ♦ Goals of scheduling
  ♦ Various well-known scheduling algorithms
• We’ll discuss scheduling algorithms in two contexts:
  ♦ In preemptive systems, the scheduler can interrupt a running job.
  ♦ In non-preemptive systems, the scheduler waits for a running job to explicitly block.
OS as a Resource Manager: Allocation vs. Scheduling

- **Allocation (spatial)**
  - **Who gets what.** Given a set of requests for resources (e.g. memory), which processes should be given which resources (e.g. how much memory & where) for best utilization

- **Scheduling (temporal)**
  - **How long can they keep it.** When more resources (e.g. 10 CPUs) are requested than can be granted (e.g. 1 CPU), in what order can they be serviced?
CPU Scheduling

• CPU scheduling is the basis of multiprogrammed operating systems

• By switching a CPU core among multiple processes, the OS can make the CPU maximally utilized
Non-Preemptive Scheduling

- OS only has a chance to schedule threads on a core when the current running thread leaves its running state:
  - Yield, terminate, blocked by I/O, etc.

- How can we force a thread off its running state?
Timer Interrupts

- Using timer interrupt to do CPU management

Timer interrupt
- generated by hardware
- setting requires privilege
- delivered to the OS

- before moving a process to running, OS sets timer
- if process yields/blocks, clear timer, go to scheduler
- If timer expires, go to scheduler
Preemptive Scheduling

A running process is interrupted by the timer, and CPU is switched to run another process.

- **Create** → **Ready**
- **Scheduler dispatch** → **Running**
- **Yield, Timer Interrupt** (call scheduler) → **Ready**
- **Terminate** (call scheduler) → **Ready**
- **Block for resource** (call scheduler) → **Blocked**
- **I/O completion interrupt** (move to ready queue) → **Ready**
Separating Policy from Mechanism

Mechanism – tool to achieve some effect

Policy – decisions on how to use tool

examples:
- All users treated equally
- All program instances treated equally
- Preferred users treated better

Separation leads to flexibility
**CPU Scheduling Mechanism**

- **Mechanisms** are relatively easy
- Context switching
- Process queues and process states
CPU Scheduling Policy

• Now we’ll look at the policies
  ♦ Which process (thread) to run, for how long, etc.
• We’ll refer to schedulable entities as jobs (standard usage) – could be processes, threads, people, etc.
Challenges in Policy

• Flexibility - variability in job types
  ♦ Long vs. short
  ♦ Interactive vs. non-interactive
  ♦ I/O-bound vs. compute-bound

• Issues
  ♦ Short jobs shouldn’t suffer
  ♦ (Interactive) Users shouldn’t be annoyed
Challenges in Policy (cont)

- Fairness
  - All users should get access to CPU
  - Amount of CPU should be roughly even?

- Issue
  - Short-term vs. long-term fairness
Goals and Assumptions

- **Goals (Performance metrics)**
  - Minimize turnaround time
    - avg time to complete a job
    - \( T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}} \)
  - Maximize throughput
    - operations (jobs) per second
    - Minimize overhead of context switches: infrequent switch
    - Efficient utilization (CPU, memory, disk etc)
  - Short response time
    - \( T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}} \)
    - type on a keyboard
    - Frequent switch
  - Fairness
    - fair, no starvation, no deadlock
Scheduling policies

• Goals often conflict
  ♦ Response time vs. throughput
  ♦ fairness vs. avg turnaround time?

• Is there an optimal scheduling policy?
• Even if we narrow down to one goal?

• But we don’t know about future
  ♦ Offline vs. online
Scheduling policies

- FIFO
- Round Robin
- SJCF
- SRTCF
FCFS/FIFO (Non-Preemptive)

• First-come first-served (FCFS), first-in first-out (FIFO)
  ♦ Jobs are scheduled in order of arrival to ready Q
  ♦ “Real-world” scheduling of people in lines (e.g., supermarket)

• Advantages
  ♦ Simple, minimal context switch overhead, no starvation

• Problem?
  ♦ Average waiting time can be large if small jobs wait behind long ones (high turnaround and response time)
  » You have a basket, but you’re stuck behind someone with a cart
Round Robin (RR) (Preemptive)

• Each job runs a time slice or quantum
  ♦ Ready queue is treated as a circular queue
  ♦ A job executes for the duration of the quantum, or until it blocks or is interrupted
  ♦ Short response time, no starvation
• How do you choose time slice?
  ♦ Overhead vs. response time
• Problem?
Is Fairness Always Good?

- Assume 10 jobs waiting to be scheduled, each taking 100 seconds
- Assume no scheduling or context switching overhead
- Total CPU time? 1000 seconds, always

- Implications?
  - Last job always finishes at 1000 seconds
  - So what’s the point of scheduling?
FIFO Example

• Job 1 – start 0, end 100
• Job 2 – start 100, end 200
• ...
• Job 10 – start 900, end 1000

• Average turnaround time = \( \frac{100 + 200 + \ldots}{N} = 550 \text{ sec} \)
Round Robin Example

• Assume each quantum is 1 second
• Job 0 – 0, 10, 20, 30, 40,…, 990
• Job 1 – 1, 11, 21, 31,…, 991
• Job 2 – 2, 12, 22, 32,…, 992
• …

• Avg turnaround time = \(\frac{990+991+…}{N} = 995\)
Like, Whoa!

- Unfair policy was faster!
- Job 10 always ended at the same time
- Round-Robin just hurt jobs 1-9 with no gain
So Why Use Round-Robin?

• Imagine 10 jobs
• Jobs 1-9 are 100 seconds
• Job 10 is 10 seconds

• Which policy is better now?
FIFO again

- Jobs 1-9 are 100 seconds
- Job 10 is 10 seconds

- Job 0 – start 0, end 100
- Job 1 – start 100, end 200
- Job 10 – start 900, end 910

- Avg turnaround time = \(\frac{100+200+\ldots+910}{N} = 541\)
Round-robin again

- Jobs 1-9 are 100 seconds
- Job 10 is 10 seconds

- Job 0 – 0, 10, 20, …, 900
- Job 1 – 1, 11, 21, …, 901
- Job 10 – 9, 19, 29, …, 99

Avg turnaround time = \( \frac{900 + 901 + 908 + 99}{10} = 824 \)

9% work drop ➔ 2% avg turnaround drop for FIFO
17% avg turnaround drop for RR
So Why Use Round-Robin?

• Imagine 10 jobs
• Jobs 1 is 100 seconds
• Job 2-10 is 10 seconds

• Which policy is better now?
  ♦ FIFO: average turnaround 145
  ♦ RR: average turnaround 105
SJF – Shortest Job First (Non-Preemptive)

• What shall we do if we care about turn-around time?
  ♦ FIFO can be bad

• STCF/SJF
  ♦ schedule shortest (total completion time) job first
SJF – Pros and Cons

• Can we do better than Shortest Job First in terms of average turnaround time?
  ♦ Assume all jobs arrive at the beginning

• In fact, SJF can be proved to be the optimal scheduling algorithm with the above assumption
  ♦ But we are not going to prove it, since this is not a theory class 😊

• SJF Advantage
  ♦ Minimal average turnaround time

• Disadvantage
  ♦ Difficult to know the future, has to run until finish
SJF vs. SRTCF

- Shortest job first (non-preemptive)

- Shortest remaining time to completion first (preemptive)

Any potential problems?  
- Can cause starvation!
Observations so far

• Need to accommodate interactive jobs
  ♦ Need some kind of RR

• Diversity in jobs – job length, I/O mix
  ♦ RR also appears to help

• SJF also has virtue
  ♦ Reduce avg. turnaround time

• Can we accommodate all?
Scheduling policies

- FIFO
  - Response time
- RR
  - Throughput
  - Avg. turnaround time
- SJF
  - Fairness
Multiple Queue Scheduling

- Motivation: processes may be of different nature and can be easily classified
  - e.g. foreground jobs vs. background jobs

- The method:
  - Processes permanently assigned to one queue, based on processes priority / type
    - Preference to jobs with higher priorities
  - Each queue can have its own scheduling algorithm
    - e.g. RR for foreground queue, FCFS for background queue

- Need a scheduling among the queues
  - e.g. fixed priority preemptive scheduling (high-pri queue trumps other)
    - e.g. time-slice between queues
Scheduling Overhead

• Operating systems aim to minimize overhead
  ♦ Context switching is not doing any useful work and is pure overhead
  ♦ Overhead includes context switch + making a scheduling decision

• Modern time-sharing OSes (Unix, Windows, …) time-slice processes in ready list
  ♦ A process runs for its quantum, OS context switches to another, next process runs, etc.
  ♦ A CPU-bound process will use its entire quantum (e.g., 10ms)
  ♦ An IO-bound process will use part (e.g., 1ms), then issue IO
  ♦ The IO-bound process goes on a wait queue, the OS switches to the next process to run, the IO-bound process goes back on the ready list when the IO completes
What about I/Os?

CPU

A B A B A B A

I/O

A A A A A

B
CPU Utilization

- CPU utilization is the fraction of time the system is doing useful work (e.g., not context switching)

- If the system has
  - Quantum of 10ms + context-switch and decision making overhead of 0.1ms
  - 3 CPU-bound processes + round-robin scheduling

- In steady-state, time is spent as follows:
  - 10ms + 0.1ms + 10ms + 0.1ms + 10ms + 0.1ms
  - CPU utilization = time doing useful work / total time
  - CPU utilization = (3*10ms) / (3*10ms + 3*0.1ms) = 30/30.3

- If one process is IO-bound, it will not use full quantum
  - 10ms + 0.1ms + 10ms + 0.1ms + 1ms + 0.1ms
  - CPU util = (2*10 + 1) / (2*10 + 1 + 3*0.1) = 21/21.3
Scheduling Summary

• Scheduler (dispatcher) is the module that gets invoked when a context switch needs to happen
• Scheduling algorithm determines which process runs, where processes are placed on queues
• Many potential goals of scheduling algorithms
  ♦ Utilization, throughput, wait time, response time, etc.
• Various algorithms to meet these goals
  ♦ FCFS/FIFO, SJF, Priority, RR
• Can combine algorithms
  ♦ Multiple-level feedback queues (backup slides, not in exam)
## Scheduling Algorithms in OSes

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Preemption</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 3.1x</td>
<td>None</td>
<td>Cooperative Scheduler</td>
</tr>
<tr>
<td>Windows 95, 98, Me</td>
<td>Half</td>
<td>Preemptive for 32-bit processes, Cooperative Scheduler for 16-bit processes</td>
</tr>
<tr>
<td>Windows NT (2000, XP, Vista, 7, and Server)</td>
<td>Yes</td>
<td>Multilevel feedback queue</td>
</tr>
<tr>
<td>Mac OS pre-9</td>
<td>None</td>
<td>Cooperative Scheduler</td>
</tr>
<tr>
<td>Mac OS 9</td>
<td>Some</td>
<td>Preemptive for MP tasks, Cooperative Scheduler for processes and threads</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>Yes</td>
<td>Multilevel feedback queue</td>
</tr>
<tr>
<td>Linux pre-2.6</td>
<td>Yes</td>
<td>Multilevel feedback queue</td>
</tr>
<tr>
<td>Linux 2.6-2.6.23</td>
<td>Yes</td>
<td>O(1) scheduler</td>
</tr>
<tr>
<td>Linux post-2.6.23</td>
<td>Yes</td>
<td>Completely Fair Scheduler</td>
</tr>
<tr>
<td>Solaris</td>
<td>Yes</td>
<td>Multilevel feedback queue</td>
</tr>
<tr>
<td>NetBSD</td>
<td>Yes</td>
<td>Multilevel feedback queue</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>Yes</td>
<td>Multilevel feedback queue</td>
</tr>
</tbody>
</table>
Next time...

- Memory replacement (another type of policies)
Priority Scheduling

• To accommodate the spirits of SJF/RR/FIFO

• The method
  ♦ Assign each process a *priority*
  ♦ Run the process with highest priority in ready queue first
    » Use FIFO for processes with equal priority
  ♦ Adjust priority dynamically
    » To deal with *all* issues: e.g. aging, I/O wait raises priority

• Advantage
  ♦ Flexibility: Not all processes are “born” equal
Priority Scheduling (cont)

• Who sets the priorities
  ♦ Internally by OS
  ♦ Externally by users/sysadm
    » e.g., Importance, funds paid for

• Dynamically adjustment is tricky
Multiple Queue Scheduling

• Motivation: processes may be of different nature and can be easily classified
  ♦ e.g. foreground jobs vs. background jobs

• The method:
  ♦ Processes permanently assigned to one queue, based on processes priority / type
    » Preference to jobs with higher priorities
  ♦ Each queue can have its own scheduling algorithm
    » e.g. RR for foreground queue, FCFS for background queue
  ♦ Need a scheduling among the queues
    » e.g. fixed priority preemptive scheduling (high-pri queue trumps other)
    » e.g. time-slice between queues
Multiple Queue Example

[High Priority]
Q8 ➔ A ➔ B
Q7
Q6
Q5
Q4 ➔ C
Q3
Q2

[Low Priority]
Q1 ➔ D
Pros/Cons of Multiple Queue Scheduling

• **Pros:**
  - Jobs do not move across queues
    - Lower scheduling overhead

• **Cons:**
  - Processes permanently assigned to one queue – not flexible
    - Program behavior may change
    - E.g. can switch between I/O bound and CPU bound
      → Need some learning/adaptation at runtime

  - Starvation cannot be easily handled
    → Need some learning/adaptation at runtime
Multilevel Feedback Queue (MLFQ)

- Problem: how to change priority?
- Jobs start at highest priority queue
- Feedback
  - If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
  - If a job gives up the CPU before the time slice is up, it stays at the same priority level.
  - After a long time period, move all the jobs in the system to the topmost queue (aging)
MLFQ Example – Single long job

Time Slice

Q2
C

Q1
C

Q0
C
MLFQ Example – a long job + short jobs in between

Time Slice

Leave I/O bound and interactive processes in higher-priority queue

Potential problem? Starvation
MLFQ Example – a long job+short jobs, with boost

![Diagram showing MLFQ example]

- **Time Slice**
  - Q2: A, C, B
  - Q1: C, B
  - Q0: C, C, C

- **Boost Time**
Let’s look at fairness again

- Proportional share: another view of fairness
  - Each job gets a (fair) proportional of CPU time
  - Goals here are not turnout time or response time

- How to share CPU proportionally?
  - Idea: proportional => probabilistic
Lottery Scheduling [OSDI 94]

- **Motivations**
  - SJF does well with avg turnaround time, but unfair
  - Priority scheduling is implemented by adjusting priorities, adjusting priority is a bit ad hoc.

- **Lottery method**: using **probabilistic** to assign CPU time
  - Give each job a number of tickets
  - Randomly pick a winning tickets => jobs with more tickets have higher chance to win (get CPU)
  - To approximate priority scheduling, high priority jobs get more tickets
  - To approximate SRTCF, short jobs get more tickets
  - To avoid starvation, give each job at least one ticket
Best thing about lottery scheduling

• Easy to implement!
Real-Time Scheduling

• Two types of real-time
  ♦ Hard deadline: must meet, otherwise can cause fatal error
  ♦ Soft headline: meet most of the time, but not mandatory

• Characteristics
  ♦ User control: provide users with abilities to control and specify
  ♦ Deterministic: upper bound on when to get services on an I/O
  ♦ Responsive: how long does OS delay before ack an interrupt
Deadline Scheduling

- Admission control
  - Take a job only if the system can guarantee real-time

- Information needed
  - Ready time: time at which task becomes ready
  - Starting deadline: time by which a task must begin
  - Completion deadline: time by which a task must complete
  - Processing time: time required to execute the task to completion
  - Resource requirements
  - Priority
  - Subtask structure
Multiprocessor and Cluster

Multiprocessor architecture
• L2 cache coherence
• A single “image” OS

Cluster/Multicomputer
• Distributed memory
• An OS on each box
Multiprocessor/Cluster Scheduling

• New design issue: process/thread interdependence
  ♦ Threads of the same process may synchronize
  ♦ Processes of the same job may send/recv messages
Multiprocessor/Cluster Scheduling: Example Approach

• Gang scheduling (coscheduling)
  ♦ Threads of same process will run together on multiprocessor
  ♦ Processes of same application run together on cluster

• Dedicated processor assignment
  ♦ Threads will be running on specific processors to completion
  ♦ Pros / cons?
    » Good for reducing cache misses
    » Bad for load balance / fairness
Case Study: Windows NT Scheduling

- Classes and priorities
  - Real time: 16 static priorities
  - Variable: 16 variable priorities, start at a base priority
    » If a process has used up its quantum, lower its priority
    » If a process waits for an I/O event, raise its priority
- Priority-driven scheduler
  - For real-time class, do round robin within each priority
  - For variable class, multiple queue feedback
- Multiprocessor scheduling
  - For N processors, run N-1 highest priority threads on N-1 processors and run remaining threads on a single processor
  - A thread will wait for processors in its affinity set, if there are other threads available (for variable priorities)
Case Study: Unix Scheduler

• The canonical Unix scheduler uses a MLFQ
  ♦ 3-4 classes spanning ~170 priority levels
    » Timesharing: first 60 priorities
    » System: next 40 priorities
    » Real-time: next 60 priorities
    » Interrupt: next 10 (Solaris)

• Priority scheduling across queues, RR within a queue
  ♦ The process with the highest priority always runs
  ♦ Processes with the same priority are scheduled RR

• Processes dynamically change priority
  ♦ Increases over time if process blocks before end of quantum
  ♦ Decreases over time if process uses entire quantum
Motivation of Unix Scheduler

• The idea behind the Unix scheduler is to reward interactive processes over CPU hogs
• Interactive processes (shell, editor, etc.) typically run using short CPU bursts
  ♦ They do not finish quantum before waiting for more input
• Want to minimize response time
  ♦ Time from keystroke (putting process on ready queue) to executing keystroke handler (process running)
  ♦ Don’t want editor to wait until CPU hog finishes quantum
• This policy delays execution of CPU-bound jobs
  ♦ But that’s ok