Announcements

• Project 1 is due Feb 27, 11:59pm
  – 3 days left
  – priority donation: extra credit for handling & testing
donation + priority change
• Office hours today 4-5 F
• Next week: Project 2 Help Sessions Wed Mar 1 & Th Mar 2 from 7 to 9pm in MCB 126
• Reading assignments:
  – Stallings Chapter 9.1-9.4, plus will post additional
    reading over the weekend

Scheduling

Resource Allocation & Scheduling

• Resource management is primary OS function
• Involves resource allocation & scheduling
  – Who gets to use what resource and for how long
• Resources we consider now
  – CPU time
  – Disk bandwidth
  – Network bandwidth
  – RAM
  – Disk space
• Processes are the principals that use resources
  – often on behalf of users

CPU vs. Other Resources

• CPU is not the only resource that needs to be
  scheduled
• Overall system performance depends on
  efficient use of all resources
  – Resource can be in use (busy) or be unused (idle)
    • Duty cycle: portion of time busy
  – Consider I/O device: busy after receiving I/O request
    – if CPU scheduler delays process that will issue I/O
      request, I/O device is underutilized
• Ideal: want to keep all devices busy

Preemptible vs Nonpreemptible Resources

• Nonpreemptible resources:
  – Once allocated, can’t easily ask for them back
    – must wait until process returns them (or
      exits)
    • Examples: Locks, Disk Space, Control of terminal
• Preemptible resources:
  – Can be taken away (“preempted”) and
    returned without the process noticing it
  • Examples: CPU, Memory
Physical vs Virtual Memory

- Classification of a resource as preemptible depends on price one is willing to pay to preempt it.
  - Can theoretically preempt most resources via copying & indirection.
- Virtual Memory: mechanism to make physical memory preemptible.
  - Take away by swapping to disk, return by reading from disk (possibly swapping out others).
- Not always tolerable:
  - Resident portions of kernel
  - Pintos kernel stack pages.

Space Sharing vs Time Sharing

- Space Sharing: Allocation ("how much?")
  - Use if resource can be split (multiple CPUs, memory, etc.).
  - Use if resource is non-preemptible.
- Time Sharing: Scheduling ("how long?")
  - Use if resource can’t be split
  - Use if resource is easily preemptible.

CPU Scheduling Terminology (1)

- A job (sometimes called a task, or a job instance):
  - Activity that’s scheduled: process or part of a process.
- Arrival time: time when job arrives.
- Start time: time when job actually starts.
- Finish time: time when job is done.
- Completion time (aka Turn-around time):
  - Finish time – Arrival time.
- Response time:
  - Time when user sees response – Arrival time.
- Execution time (aka cost): time a job need to execute.

Static vs Dynamic Scheduling

- Static:
  - All jobs, their arrival & execution times are known in advance, create a schedule, execute it.
  - Used in statically configured systems, such as embedded real-time systems.
- Dynamic or Online Scheduling:
  - Jobs are not known in advance, scheduler must make online decision whenever jobs arrives or leaves.
  - Execution time may or may not be known.
  - Behavior can be modeled by making assumptions about nature of arrival process.

CPU Scheduling Model

- Process alternates between CPU burst & I/O burst.

CPU Scheduling Model (2)

- If these were executed on the same CPU:
  - I/O Bound Process
  - CPU Bound Process
  - Waiting
  - CPU
  - I/O.
CPU Scheduling Terminology (2)

- Waiting time = time when job was ready-to-run
  - didn’t run because CPU scheduler picked another job
- Blocked time = time when job was blocked
  - while I/O device is in use
- Completion time
  - Execution time + Waiting time + Blocked time

Preemptive vs Nonpreemptive Scheduling

- Q.: when is scheduler asked to pick a thread from ready queue?
  - Nonpreemptive:
    - Only when RUNNING → BLOCKED transition
    - Or RUNNING → EXIT
    - Or voluntary yield: RUNNING → READY
  - Preemptive:
    - Also when BLOCKED → READY transition
    - Also on timer

CPU Scheduling Goals

- Minimize latency
  - Can mean (avg) completion time
  - Can mean (avg) response time
- Maximize throughput
  - Throughput: number of finished jobs per time-unit
  - Implies minimizing overhead (for context-switching, for scheduling algorithm itself)
  - Requires efficient use of non-CPU resources
- Fairness
  - Minimize variance in waiting time/completion time

Scheduling Constraints

- Reaching those goals is difficult, because
  - Goals are conflicting:
    - Latency vs. throughput
    - Fairness vs. low overhead
  - Scheduler must operate with incomplete knowledge
    - Execution time may not be known
    - I/O device use may not be known
  - Scheduler must make decision fast
    - Approximate best solution from huge solution space

First Come First Serve

- Schedule processes in the order in which they arrive
  - Run until completion (or until they block)
- Simple!
- Example:

<table>
<thead>
<tr>
<th>Q.: what is the average completion time?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 7 20 22 27</td>
</tr>
</tbody>
</table>

FCFS (cont’d)

- Disadvantage: completion time depends on arrival order
  - Unfair to short jobs
- Possible Convo Effect:
  - 1 CPU bound (long CPU bursts, infrequent I/O bursts), multiple I/O bound jobs (frequent I/O bursts, short CPU bursts).
  - CPU bound process monopolizes CPU; I/O devices are idle
  - New I/O requests by I/O bound jobs are only issued when CPU bound job blocks – CPU bound job “leads” convoy of I/O bound processes
- FCFS not usually used for CPU scheduling, but often used for other resources (network device)
Round-Robin

- Run process for a timeslice (quantum), then move on to next process, repeat
- Decreases avg completion if jobs are of different lengths
- No more unfairness to short jobs!

Q.: what is the average completion time?

Round Robin (2)

- What if there are no “short” jobs?

What would it be under FCFS?

Round Robin – Cost of Time Slicing

- Context switching incurs a cost
  - Direct cost (execute scheduler & context switch) + indirect cost (cache & TLB misses)
- Long time slices → lower overhead, but approaches FCFS if processes finish before timeslice expires
- Short time slices → lots of context switches, high overhead
- Typical cost: context switch < 1ms
- Time slice typical around 100ms
  - Linux: 100ms default, adjust to between 10ms & 300ms
- Note: time slice length = interval between timer interrupts (as you know from Pintos…)
  - Timer frequency usually 1000Hz

Shortest Process Next (SPN)

- Idea: remove unfairness towards short processes by always picking the shortest job
- If done nonpreemptively also known as:
  - Shortest Job First (SJF), Shortest Time to Completion First (STCF)
- If done preemptively known as:
  - Shortest Remaining Time (SRT), Shortest Remaining Time to Completion First (SRTCF)

SPN (cont’d)

- Provably optimal with respect to completion time:
  - Moving shorter job up reduces its completion time more than it delays completion of longer job that follows
- Advantage: Good I/O utilization
- Disadvantage: Can starve long jobs

Big Q: How do we know the length of a job?

Practical SPN

- Usually don’t know (remaining) execution time
  - Exception: profiled code in real-time system; or worst-case execution time analysis (WCET)
- Idea: determine future from past:
  - Assume next CPU burst will be as long as previous CPU burst
  - Or: weigh history using (potentially exponential) average: more recent burst lengths more predictive than past CPU bursts
- Note: for some resources, we know or can compute length of next “job”:
  - Example: disk scheduling (shortest-seek time first)