Frequently asked questions from the previous class survey

- Thread table in kernel space: Also for mapping user threads?
- How can multiple threads execute concurrently, when programs execute linearly?
- Why is the many-to-many model not a good idea?
- Ever useful to have a many-to-one mapping model?
- What is being scheduled?
- Which thread model is the best?
- How can a process execute on multiple cores?
- Does a process wait for one of its threads to complete?
- Difference between pthreads cancel vs kill
- Difference between threads in Java and pthreads
- Where is the Thread Pool allocated?
- In ARM-based architectures, does the active process own the L1 cache?

Topics covered in this lecture

- CPU Scheduling
- Scheduling Criteria
- Scheduling Algorithms
  - First Come First Serve (FCFS)
  - Shortest Job First
  - Round robin scheduling

Multiprogramming organizes jobs so that the CPU always has one to execute

- A single program (generally) cannot keep CPU & I/O devices busy at all times
- A user frequently runs multiple programs
- When a job needs to wait, the CPU switches to another job
- Utilizes resources effectively
  - CPU, memory, and peripheral devices

Observed Property of Process execution: CPU-I/O burst cycle

Processes alternate between CPU-I/O bursts

- load store
- add store
- read from file
- CPU burst
- wait for I/O
- I/O burst
- store increment
- index write to file
- CPU burst
- wait for I/O
- I/O burst
- load store
- add store
- read from file
- CPU burst
- wait for I/O
- I/O burst
SLIDES CREATED BY: SHRIDEEP PALICKARA

**Distribution of the duration of CPU bursts**
- Large number of short CPU bursts
  - A typical I/O bound process
- Small number of long CPU bursts
  - A typical CPU-bound process

**Bursts of CPU usage alternate with periods of waiting for I/O**

**As CPUs get faster ...**
- Processes tend to get more I/O bound
  - CPUs are improving faster than disks
- Scheduling of I/O bound processes will continue to be important

**When CPU is idle, OS selects one of the processes in the ready queue to execute**
- Records in the ready queue are process control blocks (PCB)

**The Process Control Block (PCB)**
- When a process is not running,
  - The kernel maintains the hardware execution state of a process within the PCB
    - Program counter, stack pointer, registers, etc.
- When a process is being context-switched away from the CPU
  - The hardware state is transferred into the PCB

**The Process Control Block (PCB) is a data structure with several fields**
- Includes process ID, execution state, program counter, registers, priority, accounting information, etc.

- In Linux:
  - Kernel stores the list of tasks in a circular doubly linked list called the task list
  - Each element in the task list is a process descriptor of the type struct task_struct, which is defined in `<linux/sched.h>`
  - Relatively large data structure: 1.7 KB on a 32-bit machine with ~100 fields
CPU scheduling takes places under the following circumstances:

- new
- ready
- running
- waiting
- I/O or event completion
- terminated
- No scheduling choice (UA)

Non-preemptive or cooperative scheduling:
- Process keeps CPU until it relinquishes it when:
  1. It terminates
  2. It switches to the waiting state
- Sometimes the only method on certain hardware platforms
  - E.g., when they don't have a hardware timer
- Used by initial versions of OS
  - Windows: Windows 3.x
  - Mac OS

Preemptive scheduling:
- Pick a process and let it run for a maximum of some fixed time
- If it is still running at the end of time interval?
  - Suspend it...
- Pick another process to run

Preemptive scheduling requires:
- A clock interrupt at the end of the time interval to give control of CPU back to the scheduler
- If no hardware timer is available?
  - Non-preemptive scheduling is the only option

Preemptive scheduling impacts:
- Concurrency management
- Design of the OS
- Interrupt processing

Preemptive scheduling incurs some costs:
- Access to shared data
  - Processes A and B share data
  - Process A is updating when it is preempted to let Process B run
  - Process B tries to read data, which is now in an inconsistent state
Preemptive scheduling incurs some costs:
Affects the design of the OS

- System call processing
  - Kernel may be changing kernel data structure (I/O queue)
- Process preempted in the middle AND
  - Kernel needs to read/modify same structure?
- SOLUTION: Before context switch
  - Wait for system call to complete OR
  - I/O blocking to occur

Preemptive scheduling incurs some costs:
Interrupt processing

- Interrupts can occur at any time
  - Cannot always be ignored by kernel
    - Consequences: Inputs lost or outputs overwritten
- Guard code affected by interrupts from simultaneous use:
  - Disable interrupts during entry
  - Enable interrupts at exit
  - CAVEAT: Should not be done often, and critical section must contain few instructions

The dispatcher is invoked during every process switch

- Gives control of CPU to process selected by the scheduler
- Operations performed:
  - Switch context
  - Switch to user mode
  - Restart program at the right location
- Dispatch latency
  - Time to stop one process and start another

Scheduling Criteria

- Fairness
- Policy Enforcement
- Balance
- All Systems
  - Throughput
  - Turnaround time
  - CPU Utilization
  - Response time
  - Proportionality
  - Meeting deadlines
  - Predictability
- Interactive Systems
- Real-time systems

CPU Utilization

- Difference between elapsed time and idle time
- Average over a period of time
  - Meaningful only within a context
Scheduling Criteria: Choice of scheduling algorithm may favor one over another

- **CPU Utilization**: Keep CPU as busy as possible
  - 40% for lightly loaded system
  - 90% for heavily loaded system

- **Throughput**: Number of completed processes per time unit
  - Long processes: 1/hour
  - Short processes: 10/second

Scheduling Criteria: Choice of scheduling algorithm may favor one over another

- **Turnaround time**:
  - $t_{\text{completion}} - t_{\text{submission}}$
- **Waiting time**:
  - Total time spent waiting in the ready queue
- **Response time**:
  - Time to start responding
  - $t_{\text{first response}} - t_{\text{submission}}$
  - Generally limited by speed of output device

What are we trying to achieve?

- **Objective is to maximize the average measure**
- Sometimes averages are not enough
  - Desirable to optimize minimum & maximum values
  - For good service put a ceiling on maximum response time
  - **Minimize the variance** instead of the average
    - Predictability more important
    - High variability, but faster on average, not desirable

Scheduling Algorithms

- **Decides** which process in the ready queue is allocated the CPU
- Could be preemptive or nonpreemptive
- **Optimize measure of interest**
- We will use **Gantt charts** to illustrate schedules
  - Bar chart with start and finish times for processes

First-Come, First-Served Scheduling (FCFS)

- Process requesting CPU first, gets it first
- Managed with a FIFO queue
  - When process enters ready queue?
    - PCB is tacked to the tail of the queue
  - When CPU is free?
    - It is allocated to process at the head of the queue
- Simple to write and understand
Average waiting times in FCFS depend on the order in which processes arrive.

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

Wait time = \( (0 + 24 + 27)/3 = 17 \)

Wait time = \( (6 + 0 + 3)/3 = 3 \)

Disadvantages of the FCFS scheme (1)
- Once a process gets the CPU, it keeps it
  - Till it terminates or does I/O
  - Unsuitable for time-sharing systems
- Average waiting time is generally not minimal
  - Varies substantially if CPU burst times vary greatly

Disadvantages of the FCFS scheme (2)
- Poor performance in certain situations
  - 1 CPU-bound process and many I/O-bound processes
  - Convoy effect: Smaller processes wait for the one big process to get off the CPU

Shortest Job First (SJF) scheduling algorithm
- When CPU is available it is assigned to process with smallest CPU burst
- Moving a short process before a long process?
  - Reduction in waiting time for short process
    - Greater than increase in waiting time for long process
  - Gives us minimum average waiting time for a set of processes that arrived simultaneously
  - Provable Optimal

Shortest Job First (SJF) in action

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>9</td>
</tr>
<tr>
<td>P3</td>
<td>16</td>
</tr>
<tr>
<td>P2</td>
<td>24</td>
</tr>
</tbody>
</table>

Wait time = \( (3 + 16 + 9 + 0)/4 = 7 \)
SJF is optimal ONLY when ALL the jobs are available simultaneously

- Consider 5 processes A, B, C, D and E
  - Run times: 2, 4, 1, 1
  - Arrival times: 0, 0, 3, 3
- SJF will run jobs: A, B, C, D and E
  - Average wait time: \((2 + 3 + 4 + 5)/5 = 2.8\)
  - But if you run B, C, D, E and A?
    - Average wait time: \((7 + 0 + 1 + 2 + 3)/5 = 2.6\)

Preemptive SJF

- A new process arrives in the ready queue
  - If it is shorter than the currently executing process
    - Preemptive SJF will preempt the current process

The SJF algorithm and short term schedulers

- No way to know the length of the next CPU burst
- So try to predict it
- Processes scheduled based on predicted CPU bursts

Use of SJF in long term schedulers

- Length of the process time limit
  - Used as CPU burst estimate
- Motivate users to accurately estimate time limit
  - Lower value will give faster response times
  - Too low a value?
    - Time limit exceeded error
    - Requires resubmission!

Prediction of CPU bursts:
Make estimates based on past behavior

- \(t_n\): Length of the n\textsuperscript{th} CPU burst
- \(\tau_n\): Estimate for the n\textsuperscript{th} CPU burst
- \(\alpha\): Controls weight of recent and past history
- \(\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n\)
- Burst is predicted as an exponential average of the measured lengths of previous CPU bursts
α controls the relative weight of recent and past history

- \( \tau_{n+1} = \alpha t_n + (1-\alpha) \tau_n \)
- Value of \( t_n \) contains our most recent information, while \( \tau_n \) stores the past history
- \( \tau_{n+1} = \alpha t_n + (1-\alpha) t_{n-1} + \ldots + (1-\alpha)^j t_{n-j} + \ldots + (1-\alpha)^{n+1} \alpha \tau_0 \)
- \( \alpha \) is less than 1, \((1-\alpha)\) is also less than one
- Each successive term has less weight than its predecessor

The choice of \( \alpha \) in our predictive equation

- If \( \alpha = 1/2 \)
  - Recent history and past history are equally weighted
- With \( \alpha = 1/2 \), successive estimates of \( \tau \)
  - \( t_0/2, t_0/2 + t_1/2, t_0/8 + t_1/4 + t_2/2 \)
  - By the 3rd estimate, weight of \( t_0 \) has dropped to 1/8.

Priority Scheduling

- Priority associated with each process
- CPU allocated to process with highest priority
- Can be preemptive or nonpreemptive
  - If preemptive: Preempt CPU from a lower priority process when a higher one is ready

Depiction of priority scheduling in action

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Wait time = (6 + 0 + 16 + 18 + 1)/5 = 8.2
How priorities are set

- Internally defined priorities based on:
  - Measured quantities
  - Time limits, memory requirements, # of open files, ratio (averages) of I/O to CPU burst

- External priorities
  - Criteria outside the purview of the OS
  - Importance of process, $ paid for usage, politics etc

Issue with priority scheduling

- Can leave lower priority processes waiting indefinitely
- Perhaps apocryphal tale:
  - MIT's IBM 7094 shutdown (1973) found processes from 1967!

Coping with issues in priority scheduling: Aging

- Gradually increase priority of processes that wait for a long time
- Example:
  - Process with priority of 127 and increments every 15 minutes
  - Becomes 0 in no more than 32 hours

Can SJF be thought as a priority algorithm?

- Priority is inverse of CPU burst
- The larger the burst, the lower the priority

Round-Robin Scheduling

- Similar to FCFS scheduling
  - Preemption to enable switch between processes
- Ready queue is implemented as FIFO
  - Process Entry: PCB at tail of queue
  - Process chosen: From head of the queue
- CPU scheduler goes around ready queue
  - Allocates CPU to each process one after the other
    - CPU-bound up to a maximum of 1 quantum
Round Robin: Choosing the quantum

- Context switch is time consuming
  - Saving and loading registers and memory maps
  - Updating tables
  - Flushing and reloading memory cache
- What if quantum is 4 ms and context switch overhead is 1 ms?
  - 20% of CPU time thrown away in administrative overhead

February 16, 2016

Round Robin: Improving efficiency by increasing quantum

- Let’s say quantum is 100 ms and context-switch is 1 ms
  - Now wasted time is only 1%
- But what if 50 concurrent requests come in?
  - Each with widely varying CPU requirements
  - 1st one starts immediately, 2nd one 100 ms later, …
  - The last one may have to wait for 5 seconds!
  - A shorter quantum would have given them better service

February 16, 2016

If quantum is set longer than mean CPU burst?

- Preemption will not happen very often
- Most processes will perform a blocking operation before quantum runs out
- Switches happens only when process blocks and cannot continue

February 16, 2016

Quantum: Summarizing the possibilities

- Too short?
  - Too many context switches
  - Lowers CPU efficiency
- Too long?
  - Poor responses to interactive requests

February 16, 2016

The contents of this slide-set are based on the following references