CPU Scheduling

CS 256/456

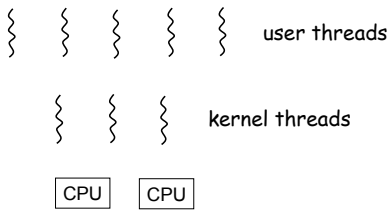
Dept. of Computer Science, University of Rochester

User/Kernel Threads

- User threads
  - Thread data structure is in user-mode memory
  - scheduling/switching done at user mode
- Kernel threads
  - Thread data structure is in kernel memory
  - scheduling/switching done by the OS kernel
- Benefits of user threads
  - lightweight – less context switching overhead
  - flexibility – allow application-controlled scheduling
- Problems of user threads
  - can't use more than one processor
  - oblivious to kernel events, e.g., all threads in a process are put to wait when only one of them does I/O

Mixed User/Kernel Threads

- M user threads run on N kernel threads (M≥N)
  - N=1: pure user threads
  - M=N: pure kernel threads
  - M>N>1: mixed model



Solaris/Linux Threads

- Solaris
  - supports mixed model
- Linux
  - No standard user threads on Linux
  - Processes are treated similarly with threads (both called tasks)
  - Processes are tasks with exclusive address space
  - Tasks can also share the address space, open files, …

Pthreads

- Different OS has its own thread package with different Application Programming Interfaces ⇒ poor portability.
- Pthreads
  - A POSIX standard API for thread management and synchronization.
  - API specifies behavior of the thread library, not the implementation.
  - Commonly supported in UNIX operating systems.

CPU Scheduling

- Selects from among the processes/threads that are ready to execute, and allocates the CPU to it.
- CPU scheduling may take place at:
  1. Hardware interrupt/software exception.
  2. System calls.
- *Nonpreemptive:*
  - Scheduling only when the current process terminates or not able to run further
- *Preemptive:*
  - Scheduling can occur at any opportunity possible

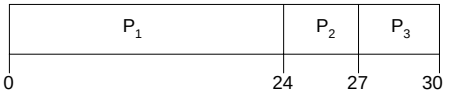
Scheduling Criteria

- Minimize turnaround time – amount of time to execute a particular process
- Maximize throughput – # of processes that complete their execution per time unit
- Maximize CPU utilization – the proportion of the CPU that is not idle
- Minimize response time – amount of time it takes from when a request was submitted until the first response is produced (interactivity)
- Fairness: avoid starvation

First-Come, First-Served (FCFS) Scheduling

Process	CPU Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3 The schedule is:



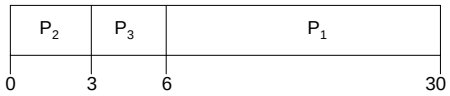
- Turnaround time for P_1 = 24; P_2 = 27; P_3= 30
- Average turnaround time: (24 + 27 + 30)/3 = 27

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

P_2 , P_3 , P_1 .

- The schedule is:



- Turnaround time for P_1 = 30, P_2 = 3, P_3 = 6
- Average turnaround time: (30 + 3 + 6)/3 = 13
- Much better than previous case.
- Short process delayed by long process: *Convoy effect*

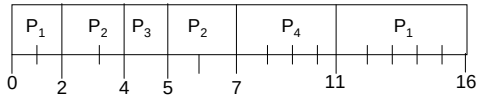
Shortest-Job-First (SJR) Scheduling

- Associate with each process the length of its CPU time. Use these lengths to schedule the process with the shortest CPU time.
- Two variations:
  - nonpreemptive – once CPU given to the process it cannot be taken away until it completes.
  - preemptive – if a new process arrives with CPU time less than remaining time of current executing process, preempt.
- Preemptive SJF is optimal – gives minimum average turnaround time for a given set of processes.
- Problem:
  - don't know the process CPU time ahead of time.

Example of Preemptive SJF

Process	Arrival Time	CPU Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average turnaround time = (16 + 5 + 1 +6)/4 = 7

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority.
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted CPU time.
- Problem: Starvation – low priority processes may never execute.
- Solution: Aging – as time progresses increase the priority of the process.

Round Robin (RR)

- Each process gets a fixed unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* small ⇒ fair, starvation-free, better interactivity
  - *q* large ⇒ FIFO
  - *q* must be large with respect to context switch cost, otherwise overhead is too high.

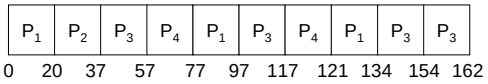
Cost of Context Switch

- Direct overhead of context switch
  - saving old contexts, restoring new contexts, … …
- Indirect overhead of context switch
  - caching, memory management overhead

Example of RR with Quantum = 20

Process	CPU Time
P_1	53
P_2	17
P_3	68
P_4	24

- The schedule is:



- Typically, higher average turnaround than SJF, but better *response*.

Multilevel Scheduling

- Ready tasks are partitioned into separate classes:
  foreground (interactive)
  background (batch)
- Each class has its own scheduling algorithm,
  foreground – RR
  background – FCFS
- Scheduling must be done between the classes.
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each class gets a certain amount of CPU time which it can schedule amongst its processes; e.g.,
    - 80% to foreground in RR
    - 20% to background in FCFS

Real-Time Scheduling

- *Hard real-time systems* – required to complete a critical task within a guaranteed amount of time.
- *Soft real-time computing* – requires that critical processes receive priority over less fortunate ones.
- EDF – Earliest Deadline First Scheduling.

Linux Task Scheduling

- Linux uses two task-scheduling classes:
  - A time-sharing class for fair preemptive scheduling.
  - A real-time class that conforms to POSIX real-time standard.
- For time-sharing tasks, Linux 2.4 uses a prioritized, credit based algorithm.
  - Each task carries an static priority, a dynamic credit; initially a task's credit is its priority
  - Scheduling is epoch-based. At each epoch
    - scheduling is ordered on the initial credit
    - Credit of the running task decrements by one at every clock tick
  - Epoch ends when no runnable tasks have any credit – recrediting

$$\text{credits} = \frac{\text{credits}}{2} + \text{priority}$$

  - This crediting system automatically prioritizes interactive or I/O-bound tasks.

CPU Scheduling on Multi-Processors

- Cache affinity
  - keep a task on a particular processor as much as possible
- Resource contention
  - prevent resource-conflicting tasks run simultaneously on sibling processors

Disclaimer

- Parts of the lecture slides contain original work of Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).