COS3 lektion 5

Nis Sarup

3. oktober 2010

4 Multithreaded Programming

4.1 Overview

- Thread = basic unit of CPU utilization.
  - Has it's own thread ID, program counter, register set and a stack.
  - Shares code section, data section and resources with other threads belonging to the same process.
- A multithreaded process can multitask.
- Benefits of using threads includes:
  - Better responsiveness
  - Resource sharing between threads is more efficient
  - Better economy as context switching threads is "cheaper"than context switching processes.
  - Better scaleability, single-thread processes can only be run on one CPU-core. Using threads more than one core can be used.

4.2 Multithreading Models

- Many-to-one model:
  - Many user threads maps to one kernel thread.
  - One thread can block the others.
  - No parallel running of threads on multi-core systems.
- One-to-one model:
  - One user thread maps to one kernel thread.
  - Not as efficient as creating one user thread will create a new kernel thread.
- Many-to-many model:

  - A pool of user threads maps to a pool of kernel thread of equal or lesser numbers
  - Cheaper than one-to-one.
  - True concurrency not achieved.
- Two-level model:
  - Combind many-to-many and one-to-one.
  - Threads can be assigned to the kerne-thread pool or get an individual kernel thread.

4.4 Threading Issues

- Fork works differently on multi-thread processes
- Exec do not (mostly)
- Threads can cancel other threads or threads can check whether or not it should be running and self-cancel.
- Thread-pools:
  - A number of threads are created at the start of a process.
  - Threads idle until works is assigned them.
  - If no threads is free, process wait for one to be freed.
  - Good on systems that cannot handle large numbers of threads.

5 Process Scheduling

5.1 Basic Concepts

- Scheduling is needed when multiple processes run on the same CPU.
- Scheduling decides which process in the ready queue gets to run.
- The goal is to have the CPU working at max all the time.
- Most processes uses the CPU only in small CPU-bursts lasting milliseconds.
- Processes in e.g. WAITing state is switched out, and a READY process can run.
- Scheduling decisions may take place:
  - When a process state switchesfrom RUNNING to WAIT.
  - When a process switches from RUNNING to READy state.
  - When a process switches from WAITing to READY state.
  - When a process terminates.

5.2 Scheduling Criteria

- What to measure to find the best Scheduling Algorithm:
  - CPU utilization: Is it working all the time?
  - Throughput: How many processes are completed per time unit?
  - Turnaround time: How long from process start to process finish.
  - Waiting time: Sum of periods waiting in the READY queue.
  - Response time: Time from process start to first response.

5.3 Scheduling Algorithms

- First-Come, First-Served:
  - Not very optimized.
  - Waiting time can be large, or small depending on which order the processes are started.
- Shortest-Job-First:
  - Gives the minimum average waiting time (Provably).
  - Hard to know the length of the next CPU burst of a given process.
- Priority
  - Low-priority processes can be blocked indefinitely.
  - Can be partially solved by using aging. Old processes get's a higher priority.
- Round-Robin
  - FIFO-queue where each process is, in turn, given access to the CPU for a given time period.
  - The process is then returned to the tail of the READY queue.
  - New processes are added to the tail as well.
  - Effectiveness varies according to time-slice-width and context-switch time.
- Multilevel Queue
  - Processes are added to different READY queues according.
  - Different queues have different priorities,
  - E.g. Foreground processes need higher priority than background processes, so the foreground queue have higher priority than the background queue.
  - Scheduling between queues can vary.

- Multilevel Feedback Queue
  - Several queues determined by time-slice-size.
  - New processes enter first queue and are getting a small time to run on the CPU.
  - If they take longer, they are preempted and sink to the next queue.
  - The next queue is run if the first queue is empty.
  - Long running processes is moved down to the last queue, FCFS.

5.4 Thread Scheduling

- User threads use process-contention scope
- Kernel threads use systemcontention scope

5.5 Multiple-Processor

- On systems with multiple CPUs load sharing becomes possible, which increases the complexity of the scheduling.
- Asymmetric multiprocessing:
  - One CPU for system
  - The rest for user processes
- Symmetric multiprocessing
  - Each CPU is self-scheduling.
  - Each CPU chooses a process from the READY queue and runs it.
- Processor Affinity:
  - Because of the high cost of repopulating caches, it is desirable for a process to run on the same CPU and not migrate to another CPU with a different cache.
- Load Balancing:
  - Push migration: A task looks at CPUs and pushed processes from an overworked CPU to an idle CPU.
  - Pull migration: Idle CPUs pulls processes from overworked CPUs.
- Memory stall can be prevented by having multiple hardware threads on each core.
- Virtualization:
  - A host OS creates virtual CPUs and assign them to guest any guest OS.

5.6 Operating System Examples

- Look in book.

5.7 Algoritm Evaluation

- Choose criteria, evaluate the given algorithms and choose the best one.
- Deterministic modeling demands known input values, but gives out nice concrete, easily comparable, values for each algorithm.
- Queuing models are only approximations of real systems.
- Simulation requires building a model of the system, but also gives nice comparable values.