Lecture Topics

- Today: Advanced Scheduling  
  (Stallings, chapter 10.1-10.4)
- Next: I/O Management  
  (Stallings, chapter 11.1-11.9)

Announcements

- Self-Study Exercise #10
- Project #9 (due 11/29)
Case Study: MSU HPCC

- MSU's High Performance Computing Center has facilities for compute-intensive programs
- Users submit batch jobs (shell scripts), must declare upper bound on resources such as CPU time and memory usage
- System schedules the jobs onto the resources

Overview: MSU HPCC

- Torque: manages resources, interacts with users
- Moab: schedules jobs onto resources
- Job which exceeds stated limits is terminated
Overview: MSU HPCC

From the HPCC website:

• The smaller the job, the more opportunities there are for resources to become available.

• The system tries to schedule large jobs (number of CPUs and memory) and then tries to fit the smaller jobs in around them.

• Given two equivalent jobs, the scheduler will give more priority to the user who has used the system less in the last week.

Example Script: MSU HPCC

#!/bin/bash -login

### walltime - how long you expect the job to run
#PBS -l walltime=00:01:00

### nodes:ppn - how many nodes & cores per node (ppn) that you require
#PBS -l nodes=5:ppn=1

### mem: amount of memory that the job will need
#PBS -l mem=2gb

### call your executable
mpirun -np 5 <executable>
Multiprocessor Systems

- On-going theme in computing: exploit parallelism to improve performance
- Instruction-level parallelism: concurrent execution of individual instructions (pipelined and superscalar architectures)
- Task-level parallelism: concurrent execution of processes or threads

Flynn's Taxonomy (1966)

- SISD: single instruction stream, single data stream
- SIMD: single instruction stream, multiple data streams
- MISD: multiple instruction streams, single data stream
- MIMD: multiple instruction streams, multiple data streams
Multiprocessor Systems

- Multiprocessor systems support execution of more than one task simultaneously (more than one processor available)
- Loosely coupled multiprocessor (cluster): multiple standalone machines interconnected by a high-speed network
- Tightly coupled multiprocessor: multiple processors interconnected at the bus level
Loosely Coupled System (Cluster)

- Collection of interconnected autonomous computers (each computer has its own processor and memory)
- Each computer is called a *node*
- Cluster as a whole can be viewed as a machine with multiple processors
- High-speed interconnection is critical (communication via message passing)
Tightly Coupled System

- Collection of processors, each of which has its own Level 1 cache
- Processors share RAM (and perhaps some levels of cache)
- Interconnected at the bus level
- Single operating system controls all processors and peripherals
Variations

- **UMA (Uniform Memory Access):** access time to memory is the same for all processors
- **NUMA (Non-Uniform Memory Access):** access time to memory may differ (access to *local memory* is faster than to *remote memory*)
- **SMP (Symmetric Multiprocessor):** another term for UMA processor

**UMA architecture**

![Diagram of UMA architecture](image)
NUMA architecture

Multicore Systems

- Multiple processors (cores) on the same die
- Tightly-coupled (shared memory) systems (can be either UMA or NUMA)
- Example: Intel Core i7
  - split Level 1 cache (per core)
  - Level 2 cache (per core)
  - Level 3 cache (shared)
  - RAM (shared)
Example: Intel Core i7

Granularity of Parallelism

- Another viewpoint: what is the frequency of synchronization of concurrent processes?
- Independent processes (no synchronization): benefit from a multiprocessor system since additional processors are available
- Cooperating processes: frequency of synchronization varies (table)
Granularity of Parallelism

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Description</th>
<th>Synchronization interval (in instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent</td>
<td>Multiple unrelated processes</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Very Coarse</td>
<td>Distributed processing across network</td>
<td>2,000 – 1,000,000</td>
</tr>
<tr>
<td>Coarse</td>
<td>Concurrent processes in multiprogramming system</td>
<td>200 – 2,000</td>
</tr>
<tr>
<td>Medium</td>
<td>Multitasking within single application</td>
<td>20 - 200</td>
</tr>
<tr>
<td>Fine</td>
<td>Instruction-level parallelism</td>
<td>&lt; 20</td>
</tr>
</tbody>
</table>

Granularity of Parallelism

- **Coarse**: concurrent processes, can run on uniprocessor system or multiprocessor system
- **Medium**: multiple threads within a single application
Multiprocessor Scheduling

- Three main issues:
  - assignment of processes to processors
  - individual processors multiprogrammed or not
  - process dispatch

- Degree of granularity of applications and number of processors available influence details of approach used

Assignment of Processes to Processors

- Static assignment: a process is permanently assigned to a specific processor (queue of processes for each processor)
  - simple, low overhead
  - one processor may sit idle while others are busy

- Dynamic assignment: a process may run on different processors (global queue or load balancing)
Assignment of Processes to Processors

Related issue: who does the assignment?

- Master / slave approach: one processor runs kernel, other processors run user processes
- Peer approach: kernel can run on any processor; each processor schedules itself (requires additional overhead to control access to shared resources)

Multiprogramming on Processors

If static assignment is used, should the processor be multiprogrammed?

- Independent: yes
- Coarse: yes
- Medium: maybe

Applications with multiple threads may have poor performance unless all threads are running simultaneously
### Process Dispatch

- Uniprocessor system: important to make good choice to keep processor busy, so sophisticated algorithm used
- Multiprocessor system: more CPU cycles available, so less important to make good choice – simpler algorithm with less overhead is sufficient

### Process and Thread Scheduling

- Process scheduling: most multiprocessor systems use a common pool of processors and dynamically allocate processes to processors
- Thread scheduling: to exploit parallelism in a threaded application, the threads must be executing simultaneously; scheduling and processor assignment can have a large impact
Approaches to Thread Scheduling

- Load sharing: one queue of threads in Ready state, each processor selects from that queue
- Gang scheduling: set of related threads scheduled onto a specific set of processors
- Dedicated processor assignment: each thread scheduled onto a specific processor
- Dynamic scheduling: number of threads in a process can change during execution

Multicore Thread Scheduling

- Typical approach: same as other multiprocessor environments
- Does not recognize that multiple levels of cache in multicore environment imposes different constraints
- Threads which share data should be scheduled based on cache organization (number of levels, how shared)
Real-Time Systems

- Tasks or processes attempt to control or react to events that take place in the outside world.
- These events occur in “real time” and tasks must be able to keep up with them.
- Correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced.

Examples of Real-Time Systems

- Control of laboratory experiments
- Process control in industrial plants
- Robotics
- Air traffic control
- Telecommunications
- Military command and control systems
Characteristics of Real-Time OS

- **Determinism**: focus is on how long it takes the OS to acknowledge an interrupt
  - Operations must be performed at fixed, predetermined times or within predetermined time intervals
  - Concerned with how long the operating system delays before acknowledging an interrupt and that there is sufficient capacity to handle all of the requests within the required time

Characteristics of Real-Time OS

- **Responsiveness**: focus is on how long it takes the operating system to service an interrupt (after acknowledging it)
  - Includes amount of time to initially handle the interrupt and start the ISR (longer if process switch is required)
  - Includes amount of time to run the interrupt service routine
  - Nested interrupt handling may cause delays
Characteristics of Real-Time OS

- **User control**
  - User given finer-grain control over priority of processes

- **Reliability**
  - Degradation of performance may have catastrophic consequences

- **Fail-soft operation**
  - Ability of a system to fail in such a way as to preserve as much capability and data as possible

Features of Real-Time Operating Systems

- **Ability to respond to external interrupts quickly**
- **Preemptive scheduling based on priority**
- **Fast process (or thread) switch**
- **Minimization of intervals during which interrupts are disabled**
- **Multitasking with interprocess communication tools such as semaphores and signals**
### UNIX SVR4 Scheduling

- Highest preference to real-time processes
- Next highest preference to kernel-mode processes
- Lowest preference to other user-mode processes

![Priority Class Table]

### Linux Scheduling

- Unit of execution is the *thread*
- By default, OS uses load sharing to maximize processor utilization
- User can influence scheduling decisions (utility programs, thread library functions)
- Recent releases of Linux (2.6 and up) support both real-time scheduling and normal scheduling
Linux Scheduling Goals

- Algorithms in scheduler to run quickly, regardless of number of threads or processors
- Perfect scalability with respect to number of processors
- Improved affinity between threads and processors (keep thread on same processor whenever possible)

Linux Scheduling Goals

- Provide good interactive service, even when system is heavily loaded
- Provide fairness (no thread should be starved, no thread should receive too many CPU cycles)
- Optimize for common case (only a few runnable threads), but scale up to handle heavy workloads
Real-time policies
- SCHED_FIFO
- SCHED_RR

Normal policies
- SCHED_OTHER
- SCHED_BATCH
- SCHED_IDLE

Real-Time Scheduling: SCHED_FIFO

- Static priority scheduling: each thread given a priority between 1 and 99 (lowest priority)
- Scheduler scans priority queues, selects thread with highest priority (FCFS, if more than one thread with same priority)
- Thread runs until it blocks, exits, or is preempted by a higher priority thread
Real-Time Scheduling: SCHED_RR

- Variation on SCHED_FIFO: each thread given a priority between 1 and 99
- Scheduler scans priority queues, selects thread with highest priority (FCFS, if more than one thread with same priority)
- Thread runs until it blocks, exits, *uses up its time slice*, or is preempted by a higher priority thread

Normal Scheduling: SCHED_OTHER

- Default is SCHED_OTHER: each thread given a static priority between 100 and 139 (120 by default)
- Scheduler calculates dynamic priority based on static priority and execution behavior
- A non-real-time thread is only selected if there are no real-time threads in the Ready state
Linux Scheduling: the O(1) scheduler

- Each processor has its own scheduling data structures
- Two data structures: set of active queues, set of expired queues
- Separate queue for each priority level (bitmap indicates which queues are not empty)
Linux Scheduling: the O(1) scheduler

- Scheduler selects thread at front of highest priority queue which is not empty (constant time using bitmap)
- When all active queues are empty, flip-flop the active queues and the expired queues by changing two pointers (the expired queues become the active queues)

Normal Threads

- Thread preempted: returns to front of active queue (keeps remaining time slice)
- Thread uses full time slice: placed at back of expired queue (given new time slice)
- Priority recalculated when thread placed in queue (based on static priority and run-time behavior)
- Time slice based on priority
Real-Time Threads

- Thread preempted: returns to *front* of active queue (keeps remaining time slice)
- Thread uses full time slice: placed at *back* of active queue (given new time slice)
- Real-time threads are never moved to the set of expired queues
- Priority never changes
- Time slice based on priority

Priority Calculations

- Priority of normal thread recalculated whenever it is placed in a queue
- Based on how long it is blocked vs. how long it is runnable (I/O bound threads will be blocked for long periods of time)
- Favors I/O bound threads, but doesn't penalize threads which have occasional long CPU bursts
The O(1) scheduler has become bloated (complex code for heuristics)

Recent alternative: Completely Fair Scheduler (CFS)

Uses red-black tree instead of run queues

Identifying next process is $O(1)$, but insertion back into the red-black tree is $O(\log N)$