CSCI-GA.2250-001

Operating Systems

Lecture 3:
Processes and Threads - Part 2
Scheduling

Hubertus Franke
frankeh@cs.nyu.edu
Processes Vs Threads

• The unit of dispatching is referred to as a *thread* or *lightweight process*
• The unit of resource ownership is referred to as a *process* or *task*
• *Multithreading* - The ability of an OS to support multiple, concurrent paths of execution within a single process
Processes Vs Threads

• Process is the unit for resource allocation and a unit of protection.
• Process has its own address space.
• A thread has:
  – an execution state (Running, Ready, etc.)
  – saved thread context when not running
  – an execution stack
  – some per-thread static storage for local variables
  – access to the memory and resources of its process (all threads of a process share this)
Processes Vs Threads

Single-Threaded Process Model

- Process Control Block
- User Address Space
- User Stack
- Kernel Stack

Multithreaded Process Model

- Process Control Block
- User Address Space
- User Stack
- Kernel Stack
- Thread Control Block
- User Stack
- Kernel Stack
- Thread Control Block
- User Stack
- Kernel Stack
- Thread Control Block
- User Stack
A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach ... Example: MS-DOS

A Java run-time environment is an example of a system of one process with multiple threads.
Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs
Multithreading on Uniprocessor System

Diagram showing the execution of threads A, B, and C on a uniprocessor system, illustrating the concepts of I/O request, request complete, time quantum expires, blocked, ready, and running states.
User-Level Threads (ULT)

- All thread management is done by the application
- The kernel is not aware of the existence of threads
User-Level Threads (ULTs)

- The kernel continues to schedule the process as a unit and assigns a single execution state.
User-Level Threads (ULTs)

Advantages
- Thread switch does not require kernel-mode.
- Scheduling (of threads) can be application specific.
- Can run on any OS.

Disadvantages
- A system-call by one thread can block all threads of that process.
- In pure ULT, multithreading cannot take advantage of multiprocessing
Kernel-Level Threads (KLTs)

- Thread management is done by the kernel
- no thread management is done by the application
- Windows OS is an example of this approach
Kernel-Level Threads (KLTs)

Advantages

• The kernel can simultaneously schedule multiple threads from the same process on multiple processors

• If one thread in a process is blocked, the kernel can schedule another thread of the same process

• Kernel routines can be multithreaded

Disadvantages

• The transfer of control from one thread to another within the same process requires a mode switch to the kernel
Combined (Hybrid) Approach

- Thread creation is done completely in user space.
- Bulk of scheduling and synchronization of threads is by the application (i.e. user space).
- Multiple ULTs from a single application are mapped onto (smaller or equal) number of KLTs.
- Solaris is an example
# Threads and Processes Relationship

<table>
<thead>
<tr>
<th>Threads:Processes</th>
<th>Description</th>
<th>Example Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:1</td>
<td>Each thread of execution is a unique process with its own address space and resources.</td>
<td>Traditional UNIX implementations</td>
</tr>
<tr>
<td>M:1</td>
<td>A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.</td>
<td>Windows NT, Solaris, Linux, OS/2, OS/390, MACH</td>
</tr>
<tr>
<td>1:M</td>
<td>A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.</td>
<td>Ra (Clouds), Emerald</td>
</tr>
<tr>
<td>M:N</td>
<td>Combines attributes of M:1 and 1:M cases.</td>
<td>TRIX</td>
</tr>
</tbody>
</table>
Scheduling

Given a group of ready processes, which process to run?
Scheduling

Given a group of ready processes, which process to run?

- When to schedule?
- Categories of scheduling algs
  - Preemptive vs Non-preemptive
  - Batch
  - Interactive
  - Real-time
- Scheduling algs goals
When to Schedule?

- When a process is created
- When a process exits
- When a process blocks
- When an I/O interrupt occurs
Categories of Scheduling Algorithms

• Batch
  – No users impatiently waiting
  – mostly nonpreemptive, or preemptive with long period for each process

• Interactive
  – preemption is essential

• Real-time
  – deadlines
How/what to measure “Scheduling”

- Turn Around Time (Batch)
- Throughput (e.g. jobs per second)
- Response Time (Interactive)
- Average wait times
Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.
Almost ALL scheduling Algorithms can be described by the following state diagram

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
Scheduling Algorithms Goals

All systems
Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems
Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems
Response time - respond to requests quickly
Proportionality - meet users’ expectations

Real-time systems
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems
Scheduling in Batch Systems: First-Come First-Served

- Nonpreemptive
- Processes ordered as queue
- A new process added to the end of the queue
- A blocked process that becomes ready added to the end of the queue
- Main disadv: Can hurt I/O bound processes
Scheduling in Batch Systems: Shortest Job First

- Nonpreemptive
- Assumes runtime is known in advance
- Is only optimal when all the jobs are available simultaneously

![Diagram](a) Run in original order

![Diagram](b) Run in shortest job first
Scheduling in Interactive Systems: Round-Robin

• Each process is assigned a time interval: quantum
• After this quantum, the CPU is given to another process
• What is the length of this quantum?
  – too short -> too many context switches -> lower CPU efficiency
  – too long -> poor response to short interactive
  – quantum longer than CPU burst is good (why?)
Scheduling in Batch Systems: Shortest Remaining Time Next

• Preemptive
• Scheduler always chooses the process whose remaining time is the shortest
• Runtime has to be known in advance
Round-Robin Scheduling

- Limits the time a process can run at one time
  → quantum
- Increases context switching → Overhead
- Promotes Fairness

Figure 2-41. Round-robin scheduling.
(a) The list of runnable processes.
(b) The list of runnable processes after B uses up its quantum.
Scheduling in Interactive Systems: Multiple Queues

Queue headers

Priority 4
Priority 3
Priority 2
Priority 1

Runnable processes

(Highest priority)

(Lowest priority)
Scheduling in Interactive Systems: Priority Scheduling

- Each process is assigned a priority
- Runnable process with the highest priority is allowed to run
- Priorities are assigned statically or dynamically
- Must not allow a process to run forever
  - Can decrease the priority of the currently running process
  - Use time quantum for each process
Multi-Level Feedback Queues

- Multiple levels of priority
- Each level is run round-robin
- If process has to be preempted, moves to worse priority.
- What kind of process should be in bottom queue?
Lottery Scheduling

• Each runnable entity is given a certain number of tickets.

• The more tickets you have, the higher your odds of winning.

• Trade tickets?

• Problems?
Fair Share Scheduler

- Schedule not only based on individual process, but process's owner.
- $N$ users, each user may have different # of processes.
- Does this make sense on a PC?
Policy versus Mechanism

• Separate what is **allowed** to be done with **how** it is done
  – a process knows which of its children threads are important and need priority

• Scheduling algorithm parameterized
  – mechanism in the kernel

• Parameters filled in by user processes
  – policy set by user process
Scheduling in Real-Time

• Process must respond to an event within a deadline
• Hard real-time vs soft real-time
• Periodic vs aperiodic events
• Processes must be schedulable
• Scheduling algorithms can be static or dynamic
Thread Scheduling

- Two levels of parallelism: processes and threads within processes
- Kernel-bases vs user-space
Conclusion

• Threads and processes are crucial concepts in OS design.
• As OS designer, you must make decision regarding: process table, threading, scheduling, etc.
• We have covered more stuff than the book so you may find information here more than the book (especially in mutual exclusion part).
Priority Scheduling

• We can assign an importance factor to our processes.

• Priority can be static or dynamic
  Why would we ever change priority?

• Higher priority equals longer time quota?
  Or
  All good priorities should finish first?
Priority Scheduling

Figure 2-42. A scheduling algorithm with four priority classes.
Figure 2-43. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.
Figure 2-43. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).
Example Linux Scheduling

• Implementation has changed multiple times
• Dynamic Priority-Based Scheduling
• Two Priority Ranges
  • Nice value -20 to +19, default 0. Larger values are lower priority. Nice value of -20 receives maximum timeslice, +19 minimum.
  • Real-time priority. By default values range 0 to 99. Real-time processes have a higher priority than normal processes.
Linux Timeslice

Lower priority or less interactive  Higher priority or more interactive

Default 100ms  Minimum 5ms  Maximum 800ms
Scheduler Goals

• O(1) scheduling - constant time
• SMP - each processor has its own locking and individual runqueue
• SMP Affinity. Only migrate process from one CPU to another if imbalanced runqueues.
• Good interactive performance
• Fairness
• Optimized for one or two processes but scales
Runqueues
<kernel/sched.c> struct runqueue

*active - active priority array
*expired - expired priority array
arrays[2] - priority arrays
*migration_thread
migration_queue
nr_iowait - number of tasks waiting on I/O
Priority Arrays

Each runqueue has two priority arrays - active expired

Each priority array contains a bitmap
If bit is set in bitmap, it indicates there are processes with a given priority. (There is also a count.)

Allows constant retrieval algorithm to find highest set bit
Scheduler Algorithm

- bit 0 priority 0
- bit 139 priority 139
- List of runnable tasks by priority
- Bit 7 priority 7
- Run the first process in the list
- 140 bit priority array
- Bit 139 priority 139
- List of runnable tasks for priority 7
Calculating Priority and Timeslice

effective_prio() returns task’s dynamic priority.
nice value + or - bonus in range -5 to +5
Interactivity measure by how much time a process sleeps. Indicates I/O activity.
sleep_avg incremented to max_sleep_avg (10 millisecs) every time does I/O. If no I/O, decremented.
Load Balancing
LAB 2 discussion
Lab instructions

What the user asked for
How the analyst saw it
How the system was designed
As the programmer wrote it
What the user really wanted
How it actually works

Possible solutions

3. Find x.

Maths question for engineers

Here it is