Frequently asked questions from the previous class survey

- When a process is waiting, does it get penalized later on when it executes?
- Difference between tasks and processes?
- Pipes
  - In memory filesystem vs. disk files
  - The shell example, how do the child communicate using the pipe? [child-child]
  - Does a process group have a default pipe to communicate over?
  - Garbage collected when the process terminates?
- How are rights transferred between queues?
- Distributed objects and RPCs: Sockets

Topics covered in this lecture

- Background
- Rationale for threads
- Thread model
- Benefits of multithreaded programming

Some background on threading

- Exploited to make programs easier to write
  - Split programs into separate tasks
- Took off when GUIs became standard
  - User perceives better performance
  - Programs did not run faster; this was an illusion
- Dedicated thread to service input OR display output
- Growing trend to exploit available processors on a machine

What are threads?

- Miniprocesses or lightweight processes
- Deja vu all over again?
  - Why would anyone want to have a kind of process within a process?
The main reason for using threads

- In many applications multiple activities are going on at once
  - Some of these may block from time to time
- Decompose application into multiple sequential threads
  - Running in quasi-parallel

Isn’t this precisely the argument for processes?

- Yes, but there is a new dimension …
- Threads have the ability to share the address space (and all of its data) among themselves
- For several applications
  - Processes (with their separate address spaces) don’t work

Threads are also lighter weight than processes

- Faster to create and destroy than processes
- In many systems thread creation is 10-100 times faster
- When number of threads that are needed changes dynamically and rapidly?
  - Lightweight property is very useful

Threads: The performance argument

- When all threads are CPU bound all the time?
  - Threads yield no performance gain
- But when there is substantial computing and substantial I/O
  - Having threads allows activities to overlap
  - Speeds up the application

AN EXAMPLE APPLICATION
Word Processor

- Displays document being created on the screen
- Document formatted exactly as it will appear on a printed page
Let's take a look at someone editing a 800-page document

- User deletes one sentence from Page-1 of an 800-page document
- Now user wants to make a change on page 600
  - Either go to that page or search for term that only appears there

Page 600 after the edit on Page 1

- Word processor does not know what's the first line on page 600
- Word processor has to reformat entire book up to page 600
- Threads could help here …

Suppose the word processor is written as a 2-threaded program

- One thread interacts with the user
- The second thread handles formatting in the background
- As soon as the sentence is deleted
  - Interactive thread tells formatter thread to format the book

While we are at it, why not add a third thread?

- Automatically save file every few minutes
- Handle disk backups without interfering with the other 2 threads

What if the program were single threaded?

- Whenever disk backup started
  - Commands from keyboard/mouse would be ignored till backup was finished
  - User perceives sluggish performance
- Alternatively, keyboard/mouse events could interrupt the disk backup
  - Good performance
  - Complex, interrupt-driven programming

With 3 threads the programming model is simpler

- First thread interacts with the user
- Second thread reforms when told to
- Third thread writes contents of RAM on to disk periodically
Three separate processes WOULD NOT work here

- All three threads need to operate on document
- By having 3 threads instead of 3 processes
  1. The threads share a common memory
  2. Have access to document being edited

Applications are typically implemented as a process with multiple threads of control

- Perform different tasks in the application
  - Web browser
    - Thread A: Render images and text
    - Thread B: Fetch network data
  - Assist in the performance of several similar tasks
    - Web Server: Manages requests for web content
      - Single threaded model: One client at a time
      - Poor response times
      - Multithreaded model: Multiple clients served concurrently

The process model is based on two independent concepts

- Resource grouping
- Execution

A process can be thought of as a way to group related resources together

- Address space containing program text and data
- Other resources
  - Open files, child processes, signal handlers, etc.

A process also has a thread-of-execution

- Usually shortened to just thread
- The thread has
  1. Program counter
  2. Registers: Current working variables
  3. Stack: Contains execution history
    - One frame for each procedure called, but not returned from

CLASSICAL THREAD MODEL
Although a thread must execute in some process

- The process and thread are different concepts
- Can be treated separately
- Processes are used to group resources together
- Threads are entities scheduled for execution on the CPU

Threads & Processes

- Threads extend the process model by allowing multiple executions in the same process
- Multiple threads in parallel in one process?
  - Analogous to multiple processes running in parallel on one computer

Threads & Processes

- Threads and processes

<table>
<thead>
<tr>
<th>User Space</th>
<th>Kernel Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three processes, each with one thread</td>
<td>One process with three threads</td>
</tr>
</tbody>
</table>

Different threads in a process are NOT AS INDEPENDENT as different processes

- All threads within a process have the same address space
  - Share the same global variables
- Every thread can access every memory address within the process' address space
  - Read
  - Write
  - Wipe out another thread's stack

There is no protection between threads, because ...

1. It is impossible
2. It should not be necessary

Unlike processes which may be from different users

- A process is always owned by a single user
- The user created threads so that they can cooperate ... not fight
Contrasting items unique & shared across threads

<table>
<thead>
<tr>
<th>Per process items (Shared by threads within a process)</th>
<th>Per thread items (Items unique to a thread)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program Counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child Processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting Information</td>
<td></td>
</tr>
</tbody>
</table>

A thread is a basic unit of CPU utilization

- Thread ID
- Program Counter
- Register Set
- Stack
- State

Sharing among threads belonging to a given process

- Code section
- Data section
- OS resources
  - Open files
  - Signals

A process with multiple threads of control can perform more than 1 task at a time

Why each thread needs its own stack

- Stack contains one frame for each procedure called but not returned from
- Frame contains
  - Local variables
  - Procedure’s return address

Why each thread needs its own stack

- Procedure X calls procedure Y, Y then calls Z
- When Z is executing?
  - Frames for X, Y and Z will be on the stack
- Each thread calls different procedures
  - So has a different execution history
Each thread has its own stack

Thread states are similar to processes
- Running
- Blocked
- Ready
- Terminated

Benefits of multithreaded programming
- Responsiveness
- Resource Sharing
- Economy
- Scalability

The rationale for threads
- Process creation is time consuming and resource intensive.
- If a new process performs the same tasks as an existing process, why incur this overhead?
- Much more efficient to use multiple threads in the process.

Threads have made inroads into the OS itself
- Most OS kernels are now multithreaded:
  - Perform specific tasks
  - Interrupt handling
  - Device management
- Solaris OS:
  - Multiple threads in the kernel for interrupt handling
- Linux:
  - Kernel thread manages system’s free memory

Benefits of multithreaded programming
- Responsiveness
- Resource Sharing
- Economy
- Scalability
Multithreaded programming: Benefit #1
Responsiveness
- Shifting work to run in the background
- Interactive multithreaded application
  - Parts of program may be blocked or slow
  - Remainder of program may still chug along
  - E.g., Web browser
    - You may read text, while high-resolution image is being downloaded

Multithreaded programming: Benefit #2
Resource Sharing
- Programmer arranges sharing between processes
  - Shared memory & message passing
- Threads within a process share its resources
  - Memory, code, and data
  - Allows several different threads of activity within the same process

Multithreaded programming: Benefit #3
Economy
- Process creation is memory and resource intensive
- Threads share process' resources
  - Economical to create and context-switch threads

Multithreaded programming: Benefit #4
Scalability
- A single threaded process can ONLY run on 1 processor
  - Regardless of how many are available
  - Underutilization of compute resource
- Programs can use threads on a multiprocessor to do work in parallel
  - Do the same work in less time OR
  - Do more work in the same elapsed time

Comparing thread executions on single core and dual core systems

<table>
<thead>
<tr>
<th>Time</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1</td>
<td>T3</td>
</tr>
<tr>
<td></td>
<td>T2</td>
<td>T4</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td></td>
<td>T4</td>
<td>T3</td>
</tr>
</tbody>
</table>

Single core: Thread executions are interleaved on a single core

<table>
<thead>
<tr>
<th>Time</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T1</td>
<td>T3</td>
</tr>
<tr>
<td></td>
<td>T2</td>
<td>T4</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td></td>
<td>T4</td>
<td>T3</td>
</tr>
</tbody>
</table>

True concurrency: Threads execute in parallel on different cores

Demand pulls of multicore systems
- OS designers
  - Scheduling algorithms to harness multiple cores
- Application Programmers
  - Modify existing non-threaded programs
    - Daunting!
  - Design multithreaded programs
Going about writing multithreaded programs [1/2]

- The key idea is to write a concurrent program — one with many simultaneous activities
  - As a set of sequential streams of execution, or threads, that interact and share results in very precise ways
- Subdivide functionality into multiple separate & concurrent tasks
- Threads let us define a set of tasks that run concurrently while the code for each task is sequential.

Going about writing multithreaded programs [2/2]

- Managing data manipulated by tasks
  - Split to run on separate cores. BUT
    - Examine data dependencies between the tasks
- Threaded programs on many core systems have many different execution paths
  - Which may or may not reveal bugs
  - Testing and debugging is inherently harder

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