CS 370: OPERATING SYSTEMS

[THREADS]

Shrideep Pallickara
Computer Science
Colorado State University

February 9, 2016

Topics covered in this lecture

- Classical thread model
- User- and kernel-level threads
- Thread Models

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, pending alarms, signal handlers, etc.

Frequently asked questions from the previous class survey

- Objective C (early 1980s) and C++
- SmallTalk (1972), C++ (1983)
- Pipe
- Does a child inherit closed files?
- Does data travel faster with using two pipes (say the data goes in the same direction)?
- Blocking
- Endian representations
- How many threads can a process spawn?
- If a thread is blocked what happens to the calling process?
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

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 and Processes

- Three processes, each with one thread
- One process with three threads

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.
- User created threads so that they can cooperate ... not fight.

Contrasting items unique & shared across threads:

<table>
<thead>
<tr>
<th>Per process Items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Shared by threads within a process)</td>
<td>(Items unique to a thread)</td>
</tr>
<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>Signals and signal handlers</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:

Traditional Heavy weight process

Process with multiple threads:

Why each thread needs its own stack (1):
- 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 (2)

- Procedure X calls procedure 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

- The rationale for threads
  - Process creation is:
    - Time consuming
    - Resource intensive
  - If new process performs same tasks as 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

1. Responsiveness
2. Resource Sharing
3. Economy
4. Scalability

Multithreaded programming: Benefit #1 Responsiveness

- 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
- Solaris: Process vs. Threads
  - Process creation is 30 times slower
  - Process context switching is 5 times slower

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

Comparing thread executions on single core and dual core systems

Time

Single core: Thread executions are interleaved on a single core

Time

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)

- **Subdivide** functionality into multiple separate & concurrent tasks
- Ensure tasks perform equal work of equal value

Going about writing multithreaded programs (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

**Complications introduced by threads**

Semantics of `fork()` and `exec()` with a multithreaded program

- If one thread calls `fork()`
  - Does new thread duplicate all threads?
  - Is the new process single-threaded?
- Depends on when/if `exec()` is called
  - If immediate: Duplicating all threads unnecessary
  - If NOT: Separate process should duplicate all threads

If the child process gets as many threads as the parent

- What happens if a thread in the parent was blocked on a **read** system call?
  - Say from the keyboard
- Are there two threads blocked on the keyboard?
  - When a line is typed, do both threads get a copy?
  - Same issue with open network connections
Problems relating to sharing data structures

- What if one thread closes a file ...
  - When another thread is reading from it?
- A thread notices that there is little memory
  - Starts allocating more memory
  - Midway in the allocation, a thread-switch occurs
  - New thread notices there is too little memory
    - Starts allocating more memory
    - Memory gets allocated twice!

Support for threads

- Kernel threads
  - Supported & managed by the OS
- User threads
  - User level
  - Above the kernel
- A relationship must exist between user threads and kernel threads

Summarizing threading models

- Many-to-One
- One-to-one
- Many-to-Many
- Two-level

User-level threads
User-level threads: Overview

- Process
- Thread
- Runtime System

Kernel

User space

Thread table

Process table

User threads are invisible to the kernel and have low overhead

- **Compete among themselves** for resources allocated to their encapsulating process
- Scheduled by a **thread runtime** system that is part of the process code
- Programs link to a special library
  - Each function is enclosed by a **jacket**
  - Jacket function calls thread runtime to do thread management
    - Before (and possibly after) calling jacketed library function.

User threads are invisible to the kernel and have low overhead

- **Compete among themselves** for resources allocated to their encapsulating process
- Scheduled by a **thread runtime** system that is part of the process code
- Programs link to a special library
  - Each function is enclosed by a **jacket**
  - Jacket function calls thread runtime to do thread management
    - Before (and possibly after) calling jacketed library function.

User level thread libraries: Managing blocking calls

- **Replace** potentially blocking calls with non-blocking ones
- If a call does not block, the runtime invokes it
- If the call **may block**
  1. Place thread on a list of **waiting** threads
  2. Add call to list of actions to try later
  3. Pick another thread to run
- **All control is invisible to user and OS**

Disadvantages of the user level threads model (1)

- Assumes that the runtime will eventually regain control, this is thwarted by:
  - CPU bound threads
  - Thread that rarely perform library calls ...
    - Runtime can’t regain control to schedule other threads
- **Programmer must avoid** lockout situations
  - Force CPU-bound thread to yield control

Disadvantages of the user level threads model (2)

- Can only share processor resources allocated to encapsulating process
  - **Limits** available parallelism

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