Threads

CSE 2431: Introduction to Operating Systems
Reading: Chap. 4, [OSC]
Contents

• What Are Threads?
• Why Use Threads?
• Thread Usage
• Thread Review
• Thread Implementations
• Issues Making Single-Threaded Code Multi-threaded
Process Review

• What Is A Process?
  – It’s one executing instance of a “program”
  – It’s separate from other instances
  – It can start (“launch”) other processes

• What’s in a process?
  – Code (text), data, stack, heap
  – Process control block
    • Process state, priority, accounting
    • Program counter, register variables, stack pointers, etc
    • Open files and devices
Process Model – Another Perspective

• Resource grouping
  – Address space (text, data, etc)
  – Opened files, signal handlers
  – Accounting info
  – …

• A thread of execution
  – Program counter
  – Registers
  – Stack
  – State
Threads: “Lightweight Processes”

(a) Three processes with one thread each
(b) One process with three threads
Thread Model

• Shared information (per process items)
  – Process info: parent process, time, etc.
  – Memory: segments, page table, etc.
  – I/O and files: comm ports, directories and file descriptors, etc.

• Private state (per thread items)
  – State (ready, running and blocked)
  – Registers
  – Program counter
  – Execution stack

• Each thread execute separately

• *Any protection between threads?*
Why Is Stack Private?

![Diagram showing thread stacks and kernel]

Thread 1's stack

Thread 2

Thread 3

Thread 3's stack

Process

Kernel
An Example Program

#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t pid; // stores the new thread ID
    pthread_create(&tid, NULL, thread, NULL); // create a new thread
    pthread_join(tid, NULL); // main thread waits for
    // the other thread to terminate
    exit(0); /* main thread exits */
}

void *thread(void *vargp) /* thread routing */
{
    printf(``Hello, world! \n'');
    return NULL;
}
Windows Threads
Discussion

• Processes vs. Threads
  – Similarities
  – Differences

• Real life analogies?
Real Life Analogy

• Process
  – Working on Lab1 assignment

• Threads
  – Looking for references
  – Coding
  – Running & Testing
  – Writing report

• Share
  – The same context (Lab1)

• Some may concurrently execute

• May affect each other
Contents

• What Are Threads?
• Why Use Threads?
• Thread Usage
• Thread Review
• Thread Implementations
• Issues Making Single-Threaded Code Multi-threaded
Why Threads?

• Good things about threads
  – Light-weight
  – Easy for sharing

• Thread usage examples
  – Word processor
  – Web server
Thread Usage: Word Processor

- A thread can wait for I/O, while the other threads are still running.
- What if it is single-threaded?
Thread Usage: Web Server (1)
Thread Usage: Web Server (2)

(a) Dispatcher thread

(b) Worker thread

How do you do it with a single thread? What’s the tradeoff?

How do you do it with a single thread while still providing parallelism?
Tradeoffs

Three ways to construct a server

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls, interrupts</td>
</tr>
</tbody>
</table>
Thread Review

• A lightweight process
  – Efficient for creation
  – Easy for sharing

• *Sharing resources* with other threads in the same process
  – Address space (text, data, etc.)
  – Opened files, signal handlers
  – Accounting info
  – …

• Thread-private items (for *execution*)
  – Program counter
  – Registers
  – Stack
  – Execution state
Contents

• What Are Threads?
• Why Use Threads?
• Thread Usage
• Thread Implementations
• Issues Making Single-Threaded Code Multi-threaded
Thread Implementations

- User-level threads
- Kernel-level threads
- Hybrid implementation
Implementing Threads in User Space

A user-level thread package
Challenges

• How to implement a thread package at user-level (without modifying the OS)?

• Is your implementation most efficient? Any performance problem? How to address it?
  – Blocking I/O?
  – Page faults?
  – Infinite loop?
Blocking Vs. Non-blocking System Calls

• Blocking system calls
  – Usually I/O related: `read()`, `fread()`, `getc()`, `write()`
  – Doesn’t return until the call completes
  – The process/thread is switched to blocked state
  – When the I/O completes, the process/thread becomes ready
  – Simple
    – *Real life example: a phone call*

• Using non-blocking system calls for I/O
  – Asynchronous I/O
  – Complicated
  – The call returns once the I/O is initiated, and the caller continues
  – Once the I/O completes, an interrupt is delivered to the caller
  – *Real life example: registered mail*
Implementing Threads in the Kernel

A thread package managed by the kernel
User-Level vs. Kernel-Level

• What is the difference between user-level and kernel-level threads?

• What are the tradeoffs?
  – When a user-level thread is blocked on an I/O event, the whole process is blocked
  – A context switch of kernel-level threads is expensive

• Can we do better?
Hybrid Implementations

Multiple user threads on a kernel thread

Multiplexing user-level threads onto some kernel-level threads
Scheduler Activations

• Another way to combine advantages of user-level and kernel-level threads

• A little complicated
  – Involving upcalls
  – Violating some fundamental principles

• Read Section 4.4.6 or

Contents

• What Are Threads?
• Why Use Threads?
• Thread Usage
• Thread Review
• Thread Implementations
• Issues Making Single-Threaded Code Multi-threaded
Issues of Making Single-Threaded Code Multi-Threaded

• Sharing of Global Variables

• Non-reentrant library procedures

• Signals

• Stack Management
An Example of Sharing Global Variable Problem

Conflicts between threads over the use of a global variable
Solution: Private Global Variables

Diagram:

- Thread 1's code
- Thread 2's code
- Thread 1's stack
- Thread 2's stack
- Thread 1's globals
- Thread 2's globals
## Non-Reentrant Library Routines

### addNodeToTail(List l, Node m)
```c
{
    tail = getTail(l);
    // interrupted
    tail->next = m;
    l->tail = m;
}
```

### addNodeToTail(List l, Node m)
```c
{
    tail = getTail(l);
    tail->next = m;
    l->tail = m;
}
```
Signals

• Delivered the signal to
  – One thread?
  – A set of threads?
  – All threads?

• Confliction of signal handler registration
Stack Management

• Single-threaded process
  – Automatically grow once the stack is overflowed (until some point)

• How about multi-threaded process?
  – Kernel may not be aware of multiple threads in one process
Summary

• What Is Thread?
• Why Thread?
• Thread Usages
• Thread Review
• Thread Implementations
• Issues of Making Single-Threaded Code Multi-threaded