COP 4610
Operating System Principles

Lecture 5 – Processes / Threads

Recap

• Processes
  – What is a process?
  – What is in a process control block?
  – Contrast stack, heap, data, text.
  – What are process states?
  – Which queues are used in an OS?
  – What does the scheduler do?
  – What is a context switch?
  – What is the producer/consumer problem?
  – What is IPC?
Lecture Overview: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Definition

- Process: group resources together
- Thread: entity scheduled for execution in a process
- “Single sequential stream of instructions within a process”
- “Lightweight process”
Thread of Execution

- Code
- Data
- Files
- Registers
- Stack

Thread vs. Process

- Threads have their own:
  - Thread ID (TID) (compare to PID)
  - Program counter (PC)
  - Register set
  - Stack

- Threads commonly share:
  - Code section (text)
  - Data section
  - Resources (files, signals, etc.)
Why Threads?

- Enable **multi-tasking** within an app
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Reduced **cost** ("lightweight" process)
  - Processes are heavy to create
  - IPC for threads cheaper/easier than processes
- Can “simplify” code & increase efficiency
- Kernels are generally multithreaded (different threads provide different OS services)

Multi-Threaded Server

1. Request
2. Create new thread to service the request
3. Resume listening for additional client requests

(client) → (server) → (thread)
(thread pool)
Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures

Multicore Systems
Multicore Programming

- **Multicore** systems putting pressure on programmers; challenges include:
  - **Dividing activities** (which tasks to parallelize)
  - **Balance** (if/how to parallelize tasks)
  - **Data splitting** (how to divide data)
  - **Data dependency** (thread synchronization)
  - **Testing and debugging** (how to test different execution paths)

- **Parallelism** implies a system can perform more than one task simultaneously

- **Concurrency** supports more than one task making progress
  - Single processor/core, scheduler providing concurrency

Concurrency vs. Parallelism

- **Concurrent execution on single-core system**

  single core

  T1 T2 T3 T4 T1 T2 T3 T4 T1 ...

  time

- **Parallelism on a multi-core system**

  core 1

  T1 T3 T1 T3 T1 ...

  core 2

  T2 T4 T2 T4 T2 ...

  time
Multicore Programming

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

- As # of threads grows, so does architectural support for threading (“hyperthreading”)
  - CPUs have cores as well as **hardware threads**
  - Consider Oracle SPARC T4 with 8 cores and 8 hardware threads per core

Data vs. Task Parallelism

- Count number of times each character in alphabet occurs
- **Data Parallelism**
  - Thread 1 does page 1-100
  - Thread 2 does page 100-200
- **Task Parallelism**
  - Thread 1 does letters A-M, all pages
  - Thread 2 does letters N-Z, all pages
Single and Multithreaded Processes

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

- **Kernel threads** - Supported by the Kernel, “schedulable entity”
- Examples – virtually all general-purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

Examples
- Windows NT/XP/2000
- Linux
- Solaris 9 and later

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package
Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
Pthreads

- May be provided either as user-level or kernel-level

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- **Specification**, not **implementation**

- API specifies behavior of the thread library, implementation is up to development of the library

- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

---

Pthreads Example

```c
#include <pthread.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */
    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }
    return 0;
}
```
Pthreads Example (Cont.)

```c
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);
printf("sum = %d\n",sum);
}
/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for (i = 1; i <= upper; i++)
        sum += i;
    pthread_exit(0);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

Linux Threads

- Linux refers to them as **tasks** rather than **threads**
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

<table>
<thead>
<tr>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLONE_FS</td>
<td>File-system information is shared.</td>
</tr>
<tr>
<td>CLONE_VM</td>
<td>The same memory space is shared.</td>
</tr>
<tr>
<td>CLONE_SIGHAND</td>
<td>Signal handlers are shared.</td>
</tr>
<tr>
<td>CLONE_FILES</td>
<td>The set of open files is shared.</td>
</tr>
</tbody>
</table>
Recap

• What is a thread? Why would one use a thread?
• How does a thread differ from a process?
• What are pthreads?
• What is a kernel thread?
• How does task parallelism differ from data parallelism?