CSE 30341
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**

(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 System Diagram](image-url)
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**
  - Time
  - Tasks: $T_1$, $T_2$, $T_3$, $T_4$, $T_1$, $T_2$, $T_3$, $T_4$, $T_1$, ...

- **Parallelism on a multi-core system**
  - Time
  - Core 1: $T_1$, $T_3$, $T_1$, $T_3$, $T_1$, ...
  - Core 2: $T_2$, $T_4$, $T_2$, $T_4$, $T_2$, ...

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-F, all pages
  - Thread 2 does letters G-L, 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 sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

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.)

/* 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.

Pthreads Code for Joining 10 Threads

#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
  pthread_join(workers[i], NULL);

Figure 4.10 Pthread code for joining ten threads.
Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Examples:
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
  - Microsoft Threading Building Blocks (TBB)

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e., tasks could be scheduled to run periodically
- Windows API:
  ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadProc));
  ...
  static void ThreadProc(Object stateinfo) {
    ...
  }
OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies parallel regions – blocks of code that can run in parallel

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

int main(int argc, char *argv[])
{
    /* sequential code */

#pragma omp parallel
{
    /* I am a parallel region. */
}
    /* sequential code */
    return 0;
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in "^{}" - `{ printf("I am a block"); }`
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue
- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process, called main
    - Programmers can create additional serial queues within program
  - concurrent – removed in FIFO order but several may be removed at a time
    - Three system wide queues with priorities low, default, high

```c
dispatch.queue_t queue = dispatch.get_global.queue
    (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch.async(queue, "{ printf("I am a block."); }");
```
Threading Issues:
Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - When is duplicating all threads a really bad idea?
  - Some OSes have two versions of fork
  - POSIX: only the calling thread

- `Exec()` usually works as normal – replace the running process including all threads

Threading Issues:
Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process
Threading Issues:
Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

  ```c
  pthread_t tid;
  /* create the thread */
  pthread.create(&tid, 0, worker, NULL);
  ... 
  /* cancel the thread */
  pthread.cancel(tid);
  ```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state
  - `pthread_setcancelstate() -> enable/disable`
  - `Pthread_setcanceltype()`:

<table>
<thead>
<tr>
<th>Mode</th>
<th>State</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>Disabled</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>Deferred</td>
<td>Enabled</td>
<td>Deferred</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Enabled</td>
<td>Asynchronous</td>
</tr>
</tbody>
</table>

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
    - `pthread_testcancel()`
- Asynchronous: terminate immediately
Thread-Local Storage

• **Thread-local storage (TLS)** allows each thread to have its own copy of data

• Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

• Different from local variables
  – Local variables visible only during single function invocation
  – TLS visible across function invocations

• Similar to **static data**
  – TLS is unique to each thread

---

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?