CS399 New Beginnings

Jonathan Walpole
Threads & Concurrency
Threads

Processes have the following components:
- an address space
- a collection of operating system state
- a CPU context ... or thread of control

To use multiple CPUs on a multiprocessor system, a process would need several CPU contexts
- Thread fork creates new thread not memory space
- Multiple threads of control could run in the same memory space on a single CPU system too!
Threads

Threads share a process address space with zero or more other threads.

Threads have their own CPU context:
- PC, SP, register state,
- Stack

A traditional process could be viewed as a memory address space with a single thread.
Single Thread in Address Space
Multiple Threads in Address Space

- **User Address Space**
  - **Thread 1 stack**
    - `routine1()` `var1` `var2`
  - **Thread 2 stack**
    - `routine2()` `var1` `var2` `var3`
  - **Stack Pointer Prgrm. Counter Registers**
  - **Stack Pointer Prgrm. Counter Registers**

- **Text**
  - `main()` `routine1()` `routine2()` ...

- **Data**
  - `arrayA` `arrayB`

- **Heap**

- **Process ID**
  - **User ID**
  - **Group ID**

- **Files**
  - **Locks**
  - **Sockets**
What Is a Thread?

A thread executes a stream of instructions
- it is an abstraction for control-flow

Practically, it is a processor context and stack
- Allocated a CPU by a scheduler
- Executes in a memory address space
Private Per-Thread State

Things that define the state of a particular flow of control in an executing program

- Stack (local variables)
- Stack pointer
- Registers
- Scheduling properties (i.e., priority)
Shared State Among Threads

Things that relate to an instance of an executing program
- User ID, group ID, process ID
- Address space: Text, Data (off-stack global variables), Heap (dynamic data)
- Open files, sockets, locks

Important: Changes made to shared state by one thread will be visible to the others!
Reading and writing memory locations requires synchronization!
... a major topic for later ...
Concurrent Access to Shared State

**Important**: Changes made to shared state by one thread will be visible to the others!

Reading and writing memory locations requires synchronization!

This is a major topic for later ...
Programming With Threads

Split program into routines to execute in parallel
- True or pseudo (interleaved) parallelism

Alternative strategies for executing multiple routines
Why Use Threads?

Utilize multiple CPU’s concurrently

Low cost communication via shared memory

Overlap computation and blocking on a single CPU
- Blocking due to I/O
- Computation and communication

Handle asynchronous events
Typical Thread Usage

A word processor with three threads
Processes vs Threads

GET / HTTP/1.0

HTTPD

disk
Processes vs Threads

Why is this not a good web server design?
Processes vs Threads

GET / HTTP/1.0
Processes vs Threads
Processes vs Threads
System Structuring Options

<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>

Three ways to construct a server
Common Thread Strategies

Manager/worker
- Manager thread handles I/O
- Magaer assigns work to worker threads
- Worker threads created dynamically
- ... or allocated from a thread-pool

Pipeline
- Each thread handles a different stage of an assembly line
- Threads hand work off to each other in a producer-consumer relationship
Pthreads: A Typical Thread API

Pthreads: POSIX standard threads
First thread exists in main(), creates the others

pthread_create (thread, attr, start_routine, arg)
- Returns new thread ID in “thread”
- Executes routine specified by “start_routine” with argument specified by “arg”
- Exits on return from routine or when told explicitly
Pthreads (continued)

**pthread_exit (status)**
- Terminates the thread and returns “status” to any joining thread

**pthread_join (threadid, status)**
- Blocks the calling thread until thread specified by “threadid” terminates
- Return status from pthread_exit is passed in “status”
- One way of synchronizing between threads

**pthread_yield ()**
- Thread gives up the CPU and enters the run queue
Using Create, Join and Exit
An Example Pthreads Program

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    printf("%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++)
    {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Program Output

Creating thread 0
0: Hello World!
Creating thread 1
1: Hello World!
Creating thread 2
Creating thread 3
2: Hello World!
Creating thread 4
3: Hello World!
4: Hello World!

For more examples see: http://www.llnl.gov/computing/tutorials/pthreads
Pros & Cons of Threads

Pros:
- Overlap I/O with computation!
- Cheaper context switches
- Better mapping to multiprocessors

Cons:
- Potential thread interactions
- Complexity of debugging
- Complexity of multi-threaded programming
- Backwards compatibility with existing code
User-level threads

The idea of managing multiple abstract program counters above a single real one can be implemented using privileged or non-privileged code.

- Threads can be implemented in the OS or at user level

User level thread implementations

- Thread scheduler runs as user code (thread library)
- Manages thread contexts in user space
- The underlying OS sees only a traditional process above
Kernel-Level Threads

Thread-switching code is in the kernel
User-Level Threads Package

The thread-switching code is in user space
User-level threads

Advantages
- Cheap context switch costs among threads in the same process!
- Calls are procedure calls not system calls!
- User-programmable scheduling policy

Disadvantages
- How to deal with blocking system calls!
- How to overlap I/O and computation!
Quiz

1. What is the difference between a program and a process?
2. Is the Operating System a program?
3. Is the Operating System a process?
   - Does it have a process control block?
   - How is its state managed when it is not running?
4. What is the difference between processes and threads?
5. What tasks are involved in switching the CPU from one process to another?
   - Why is it called a context switch?
6. What tasks are involved in switching the CPU from one thread to another?
   - Why are threads “lightweight”?