THREADS & SYNCHRONIZATION

ISA 563: Fundamentals of Systems Programming
Announcements / Agenda

- Projects have been selected, check Project Link
  - Teams will meet with Prof. Locasto during class next week to discuss SOW & deliverables for project

- Homework 2 due in four weeks
  - Remember that Problem 2.2 is a design exercise

- Read APUE Chapter 11
  - Chapter 12 deals with advanced issues for configuring runtime thread and locking behavior (optional reading)

- Exercise: Producer-Consumer with pthreads
Major Thread Environments

- UNIX: pthreads library
- Java: Thread class, Runnable interface
- Intel Thread Building Blocks Library
Reasons to Use Threads

- Threads typically involve less overhead (memory & CPU time) than a full process
  - All threads “within” a process share the same memory as the containing process and each other
  - The overhead of “fork(2)” is avoided
  - Context switching (time for OS to “give” the CPU to another process) between multiple processes is avoided

- Threads can more naturally reflect independent but related subtasks of an algorithm or process
  - Potential for parallel execution & some speedup
High Level: What is a Thread?

- Threads represent an independent control flow within a process.
- How threads are implemented often depends on the underlying thread library and operating system.
- POSIX defines a standard thread API to manage the lifecycle of a thread as well as synchronization primitives.
Threads typically contain the following state:

- A thread ID *tid*
- Scheduling data: policy & priority
- A set of registers (i.e., CPU state), including:
  - A program counter (keep track of which instruction the thread is executing)
  - A stack (independent of the process’s stack and any other threads within the process)
- Their own *errno*
- Their own signal mask set
Mapping Threads to Code

- Threads execute code independently; more than 1 thread can simultaneously execute the same assembly instructions.

- In other words, source code doesn’t necessarily “belong” to any one thread.
  - The association of code to threads can change dynamically during runtime.
### Mapping Threads to Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>shr $4, %eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sall $4, %eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subl %eax, %esp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>movl $0, -4(%ebp)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Major Issue: Synchronized Access

- Two or more threads, in executing the same program statements simultaneously, might access (i.e., read or write) the same data items
  - Because thread execution ordering is unpredictable (just like process scheduling), consistency is unpredictable
  - Program correctness is then questionable
  - Thread APIs (pthreads, Java’s Thread object and synchronization primitives) often provide ways to control or synchronize access to shared data
Two Threads Sorting Same Data

Thread 1: shell_sort(data, len)

```c
for(gap = len / 2; gap > 0; gap /= 2)
    for(i = gap; i < len; i++)
        for(j = i - gap; j >= 0 && data[j] > data[j + gap]; j -= gap)
            swap(data[i], data[i + gap]);
```

Thread 2: shell_sort(data, len)

```c
for(gap = len / 2; gap > 0; gap /= 2)
    for(i = gap; i < len; i++)
        for(j = i - gap; j >= 0 && data[j] > data[j + gap]; j -= gap)
            swap(data[i], data[i + gap]);
```
The main idea is to provide atomic operations that govern permission to enter a critical section.

- **Atomic operations** are operations that execute in a single machine clock cycle and cannot be interrupted at any point in their execution.

- A **critical section** is a section of code that manipulates shared data items and must be made thread-safe in order to ensure program correctness.

- Specifics of how pthread library does it later...
A single line of C code corresponds to multiple assembly (machine) instructions

Even a single machine instruction may not execute in 1 clock cycle!
Mapping C to ASM Instructions

C Code

```c
int main(int argc,
    char *argv[])
{
    int c = c + 1;
    return c;
}
```

ASM Code

```
.text
.globl _main
_main:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
leal -12(%ebp), %eax
incl (%eax)
movl -12(%ebp), %eax
leave
ret
```
## Mapping ASM to Clock Cycles

<table>
<thead>
<tr>
<th>ASM Code</th>
<th>Instruction Mneumonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td></td>
</tr>
<tr>
<td>.globl _main</td>
<td></td>
</tr>
<tr>
<td>_main:</td>
<td></td>
</tr>
<tr>
<td>pushl %ebp</td>
<td>LEA: Load Effective</td>
</tr>
<tr>
<td>movl %esp, %ebp</td>
<td>Address: 2 cycles</td>
</tr>
<tr>
<td>subl $24, %esp</td>
<td>INC: Increment by 1:</td>
</tr>
<tr>
<td>leal -12(%ebp), %eax</td>
<td>1 or 2 cycles</td>
</tr>
<tr>
<td>incl (%eax)</td>
<td>MOV: Copy 2\textsuperscript{nd}</td>
</tr>
<tr>
<td>movl -12(%ebp), %eax</td>
<td>operand to 1\textsuperscript{st}</td>
</tr>
<tr>
<td>leave</td>
<td>operand: cycles</td>
</tr>
<tr>
<td>ret</td>
<td>vary</td>
</tr>
</tbody>
</table>
Highlights of Security Issues

- Privilege Separation between threads
- Information leaks & covert channels
- TOCTTOU (time-of-check-to-time-of-use) errors
- Memory leaks or double-free errors due to mismanagement of reference counters
- DoS due to deadlock (internal mismanagement of control paths leading to lock-acquiring mis-ordering)
Creating Threads

Tracking of Thread ID

Terminating & Joining Threads
### Comparing Process & Thread Lifecycles

<table>
<thead>
<tr>
<th>Process Functions</th>
<th>Thread Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork(2)</td>
<td>pthread_create</td>
</tr>
<tr>
<td>atexit(2)</td>
<td>pthread_cleanup_push</td>
</tr>
<tr>
<td>_exit(2)</td>
<td>pthread_exit</td>
</tr>
<tr>
<td>waitpid(2)</td>
<td>pthread_join</td>
</tr>
<tr>
<td>getpid(2)</td>
<td>pthread_self</td>
</tr>
</tbody>
</table>
Thread Creation

The ‘pthread_create(3)’ function is a pthread library function that instructs the operating system to create a thread in the current process’s context.

Operating Systems can do this (i.e., map threads to OS processes) in many ways. Linux uses clone(), so 1 thread per process.
Thread Creation & Running

- Threads do not follow the fork/exec pattern for Unix processes

- Instead, when they are created, they are explicitly assigned a section of code to begin executing via the 3rd argument of pthread_create, a function pointer
Thread Identification

- int pthread_equal(pthread_t tid1, pthread_t tid2);
- pthread_t pthread_self(void);

Why a function to compare pthread IDs?
Because the pthread_t type can be a structure
(not necessarily an integer like pid_t)
Terminating Threads

- Use `pthread_exit`: extinguish current thread
- Use `pthread_join`: extinguish target thread (i.e., join with caller)
- Use `pthread_cancel` to request that another target thread be extinguished
- Threads can register shutdown hooks via:
  - Using `pthread_cleanup_push()`
  - Using `pthread_cleanup_pop()`
Using `pthread_exit`

- Allows a thread to terminate itself
- Can pass back a pointer to a return value:
  - `pthread_exit((void*)RETURN_CODE);`
- Return value can also be a structure
  - But be careful that it is a valid pointer!
  - For example, variables local to the thread stack may be destroyed by the time the caller uses the thread’s return structure value
- See Figure 11.4, page 362..364
Thread Shutdown Hooks

- Similar to atexit(3) process exit handlers
- Calls to pthread_cleanup_push and pthread_cleanup_pop must match in the source code
- These might be implemented as macros
- Figure 11.5 in APUE
Create a child thread that randomly perturbs a piece of global state

Parent sleeps for configurable (via cmdline param) amount of time
Synchronization Mechanisms

Mutexes

Reader-writer locks (shared-exclusive locks)

Condition variables
Synchronization with Mutexes

- **Mutual Exclusion: mutex**
  - A property whereby a resource is available to only 1 thread at a time. A mutex is a data item that represents a ‘lock’ on a resource

- Threads must acquire the mutex before manipulating the resource
  - This is a convention only: the OS and hardware do not enforce access on a data item --- the calling thread must be cooperative and include the calls to the mutex acquisition routines!
Caveats

- Threads can ignore mutexes and just access the data

- Threads can race to acquire the mutex itself

- Ordering of mutex acquisition and release must be the same across potentially many code paths; deadlock can occur when an infrequently-exercised code path (and thus series of mutex acquisitions) is executed by multiple threads
Using pthread Mutex Variables

- **Static Allocation:**
  
  ```c
  pthread_mutex_t mlock = PTHREAD_MUTEX_INITIALIZER;
  ```

- **Dynamic Allocation:**
  
  Use `pthread_mutex_init` after `malloc` of a `pthread_mutex_t` pointer
  
  Must use `pthread_mutex_destroy` before freeing mutex pointer

- **Lock / Unlock**
  
  ```c
  pthread_mutex_lock(&mlock);
  //critical section, update shared data
  pthread_mutex_unlock(&mlock);
  ```
Can’t Afford to Block?

- Use `pthread_mutex_trylock`
  - The calling thread can return without block from this function
  - It can then decide whether to try again, essentially looping on the mutex, or continue on some other processing path
captureflag demo

captureflag.c

captureflag-nolock.c
Reader-Writer Locks

- A better name is “shared-exclusive”
  - Three modes of access:
    - “read”: multiple threads can read this resource
    - “write”: a single thread locks resource to write to it
    - “open”: unlocked

- Finer-grained than unlocked/locked of mutexes
  - But has potential to starve writers if a high rate of readers occurs; some implementations handle this
  - Suitable for data structures that are read more often than they are updated
Condition Variables

- Customize locking based on state of the shared data
- When condition is satisfied, a signal is sent to interested threads
Summary: Take-Home Message

- Threads provide a mechanism for allowing a single, monolithic piece of source code to accomplish multiple independent or dependent subtasks concurrently.

- Concurrency introduces challenges with regards to consistency of critical data items.
  - Synchronization primitives provide a means to protect critical sections of code, but the burden rests on the programmer to use them correctly.
Summary: Things to Consider

- Why use threads instead of fork?
- Do threads guarantee mutual exclusion?
- How would you find bugs (e.g., TOCTTOU) in multi-threaded code?
- Do threads *always* require locking?
- Can a single thread cause the entire process to terminate?
Threads Exercise

Producer-Consumer: Create two threads. One thread injects random data into a random position in the data collection (i.e., array) and the other thread attempts to sort the data collection. Synchronize access to the data collection. Is the data collection ever fully sorted?
Threads Exercise (Alternate)

Competitor Sort: Create two threads that compete to sort a data collection from greatest to least and from least to greatest, respectively.