Introduction to PThreads and Basic Synchronization

Michael Jantz, Dr. Prasad Kulkarni
Dr. Douglas Niehaus

Introduction

- In this lab, we will learn about some basic synchronization issues using PThreads.
- Go ahead and make the starter code for this lab.
  - cd pthreads_intro/; make
- This creates the ptcount executable
  - Currently this does very little
- We will discuss some PThreads issues first, and then consider how to add the missing pieces to ptcount.c

Introduction

- Helpful man pages:
  - Pthreads(7): Overview of Pthreads programming model
  - pthread_create: Routine to create a new thread
  - pthread_join: One thread waits for another to terminate
  - pthread_mutex_init: Initialize a pthread_mutex_t mutex
  - pthread_mutex_lock: lock a mutex
    - Also presents pthread_mutex_unlock and pthread_mutex_trylock
- These functions are all you should need to make the code modifications required for the lab

Threads vs. Processes

- Multiple processes work well when the application tasks performed by each process are: (a) unrelated, or (b) require communication well suited to pipes or sockets
- More intimate, complex, or fine-grained collaboration among concurrent parts of an application is often difficult or impossible when multiple processes are used
- With an application's work divided among multiple processes, the context switch overhead generally decreases performance
- The thread model was invented to address these issues
  - Multiple threads sharing an address space decreases several sources of overhead

Threads vs. Processes

- Sharing of significant state information among processes is problematic because:
  - Multiple copies of state data are created or a shared segment must be declared and managed
  - Exchanging state information requires expensive IPC mechanisms and complete consistency among multiple copies is often not possible
  - Using semaphores among processes involves system call overhead
- Under the thread model context switch overhead is lower, all global variables are shared, and implementing mutexes is cheaper

Threads vs. Processes

- When you create a new process, an entire copy of the parent process' process control block is copied to the child
- When you create a new thread, only the components of the process control block that are *necessary to create a new thread of control* are actually allocated for the child
  - A new PC, registers image, user and kernel stack, and some other misc. info are allocated for the new thread.
  - The code and data regions of the address space are not copied, but are now shared between the parent and the child
- For these reasons, threads are often thought of as lightweight processes
  - Lower context switch and lower memory use compared to an equivalent set of processes

Thread Pros and Cons

- By allowing threads of control to share a common address space, the thread model provides an efficient way to multi-task a job over several threads of control
  - State information available to each thread is updated in a common copy without the use of IPC mechanisms
  - Context switching from thread to thread is more efficient than switching the context of an entire process because much of the address space remains the same
- These advantages, however, come with their own drawbacks:
  - The use of shared data must be synchronized among threads
  - Functions used by multiple threads of control must be *reentrant*
  - A reentrant function must not hold static data over successive calls or return a pointer to static data

Creating Threads

- In the POSIX API, the *fork()* system call creates processes
- Under Linux*, fork()* makes use of the *clone()* system call :
  - int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...)
- *Clone()* provides a fine-grain interface for controlling what resources are shared among parent and child
  - The *flags* argument controls which resources are shared and which copied
  - For example, calling *clone()* with the CLONE_FILES bit set in the *flags* argument will create a child which shares the parent's open file table (as opposed to creating a copy of the parent's file descriptor table for the child)
  - Process creation uses flags that copy the parent address space
  - Thread creation uses flags that share the address space

Thread Libraries

- For many reasons, thread standards are often implemented and distributed as libraries
  - A common standard allows for portable software
  - Libraries are an efficient way of distributing widely used user level code
  - Depending on which metrics the programmer cares about, different user level designs of the thread system will yield much different performance
- In today's lab, we will the Native POSIX Threads Library, which implements the POSIX Thread API standard for Linux

The POSIX Standard

- The POSIX standard specifies a set of interfaces (i.e. functions and header files) that can be used for threaded programming
- These interfaces require that the underlying implementations meet a certain set of criteria:
  - A single process may contain multiple threads, all of which execute the same program, i.e. share the same text segment
  - Threads share the same global memory (data and heap segments).
  - Each thread has its own stack (automatic variables)
- There are several other characteristics specified by the standard.
  - Threads share a Process ID, but have a unique Thread ID.
  - See the pthreads(7) manual page for a more complete description

ptcount.c

- Open ptcount.c and read through it to get a feel for the program's structure. As it is now, the program doesn't do much.
- The goal of this lab is to learn how PThreads work by modifying this program to create some number of threads executing the same routine
  - NUM_THREADS controls the number of threads created
- The routine each thread executes (inc_count) will spin in a loop (bound by the LOOP_BOUND command line arg), incrementing some global integer (by the scalar specified by the INCREMENT command line arg) for each iteration

pthread_create

- To create each pthread, use:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
```

- The arguments are as follows:
  - **thread** is an address pointing to the allocated thread object, filled in on return
  - **attr** is a pointer to the attribute object for this thread. Attribute objects provide a mechanism for changing the configurable aspects of a thread.
  - **start_routine** is a pointer to the routine in which the thread should start its execution. For this lab, the start routine for all of our threads will be **inc_count**
  - **arg** is a pointer to the arguments passed to the start routine. Each thread start_routine is only allowed one argument. Thus, multiple arguments are packaged in a struct, whose address is passed as the argument.

pthread_create (cont.)

- The easiest way to create all of our threads is in a for loop
- In order to save time, we have provided the code you should use. Make sure you understand where to put it and what each of its arguments means:

```
for ( i = 0; i < NUM_THREADS; i++)
    . . .
    pthread_create(&threads[i], &attr, inc_count, (void *)targs);
}
```

- When pthread_create returns, the thread described by &threads[i] are live (i.e. ready to run and waiting to be scheduled or has started running inc_count code already)

Waiting on the Threads

- You'll recall from the IPC lab that we used waitpid to have the parent process wait for its children to complete and exit
- We achieve similar behavior for threads with **pthread_join**:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- **pthread_join** will suspend execution of the calling process (or thread) until the target thread, the *thread* referred to by the pthread_t instance in argument *thread*, has terminated.
- **value_ptr** is a pointer to the data returned by the pthread_exit() library call. We do not use this data, so you can just pass in NULL here.
- Again, we use **pthread_join** in a for loop:

```
for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
}
```

Testing the Threaded Program

- At this point you have the parent creating NUM_THREADS threads and then waiting for the threads created to finish executing **inc_count**
- Now is a good time to go ahead and test the threaded program. To run the program, use:

```
-bash-3.2$ ./ptcount LOOP_BOUND INCREMENT
```

- Where LOOP_BOUND is the number of times the loop in each thread should iterate, and INCREMENT is the amount to increment count for each iteration.
- If you created and joined the PThreads correctly, you should see something like:

```
-bash-3.2$ ./ptcount 100 1
Thread: 0 finished. Counted: 100
Thread: 1 finished. Counted: 100
Thread: 2 finished. Counted: 100
Main(): Waited on 3 threads. Final value of count = 300. Done.
```

- Which is exactly what we wanted. Each thread increments the count 100 times (by 1 each time) and the final value of count is 300 because we have three threads.

Puzzling Behavior

- Though these initial tests seem to show that *ptcount* is working correctly, but running *ptcount* with a larger loop bound results in puzzling behavior
- The makefile has a target *test* runs *ptcount* with a large loop bound. The output should look something like this:

```
-bash-3.2$ make test
./ptcount 100000 1
Thread: 0 finished. Counted: 100000
Thread: 1 finished. Counted: 100000
Thread: 2 finished. Counted: 100000
Main(): Waited on 3 threads. Final value of count = 288284. Done.
```

- Each thread reports that it counts up to 100000, but the final count is significantly less than 300000. Note that in ptcount.c each thread prints out a *local* variable to report its own count.
- Furthermore, if you run the test again, the final count changes, but it is still not the expected result. What is going on?

Incrementing Count

- Each thread updates the count using the = and + operator in C:

```
count = count + my_args->inc;
```

- Observe that this one line instruction in C, is actually implemented by three instructions in the x86 hardware:

```
mov     $(C_ADDR), %eax      ; Move count into a register
add     %ebx, %eax           ; Add inc to count
mov     %eax, $(C_ADDR)      ; Store count back into memory
```

- Now, recall that each thread is sharing only the global instance of *count*, which is loaded from memory and stored again each time any thread increments it
- While this does not explain observed behavior yet, it's necessary for understanding the solution

A Preemptive Scheduler

- Recall from lecture that the Linux scheduler is preemptive.
  - The scheduler has the ability to stop a running process, with the intention of resuming it later, and switch context to another process in the READY state.
- Now, imagine a scenario where the scheduler just happened to interleave two running threads in the following way:

	T1	T2
1.	mov $(C_ADDR), %eax	
2.		mov $(C_ADDR), %eax
3.		add %ebx, %eax
4.		mov %eax, $(C_ADDR)
5.	add %ebx, %eax	
6.	mov %eax, $(C_ADDR)	

An Interleaving Problem

- Observe that:
  - At time 1, T1 loads the count value from memory.
  - Next, T1 is preempted and T2 begins executing. In its execution, it loads the *same value* from memory. T2 continues, incrementing the value it had loaded, and stores it back into memory.
  - Now, at time 5, T2 is preempted by T1. At this time, T1 still holds the value it loaded *at time 1* into %eax. T1 proceeds to increment *this value*, and stores it back into memory.
  - When T1 stores its value of count back into memory, the work done by T2 is essentially lost.
- This explains why the value read by the parent process after the threads had finished incrementing is substantially lower than what you might expect
  - Each instance of such an interleaving loses an increment of count by a thread
- Probability of inconsistencies in shared data, such as *count*, depends on the length of the code section using it, the number of threads sharing it, and the number of times threads execute the code section operating on shared data.

Fixing the Problem

- Regions of code that update shared data are known as *critical sections*.
- In order to ensure data shared among cooperating threads remains correct, we need a way of ensuring that only one thread may execute inside a critical section at a time:
  - This is called *mutual exclusion*
- Policy Goal:
  - All updates to shared data must be executed atomically with respect to other operations on that data.

pthread_mutex_lock

- Library calls for lock and unlock:

  int pthread_mutex_lock(pthread_mutex_t *mutex);

  int pthread_mutex_unlock(pthread_mutex_t *mutex);

- The implementation of these routines permits only one thread to lock a given mutex at a time
  - A *mechanism* for mutual exclusion on critical section
- When a thread executes **pthread_mutex_lock** and the mutex is already locked, the operating system blocks the calling thread.
- All threads blocking on the mutex are awakened when the lock is released -> the owner calls **pthread_mutex_unlock**

Mutex Solution

- Place mutex lock/unlock around the critical section in ptcount.c to ensure mutually exclusive access to the *count* variable
  - Note that the code already declares an instance of pthread_mutex_t count_mutex
- When you have built *ptcount* with the locks in place, *ptcount* should produce the following output:

```
-bash-3.2$ make test

./ptcount 1000000 1
Thread: 0 finished. Counted: 1000000
Thread: 2 finished. Counted: 1000000
Thread: 1 finished. Counted: 1000000
Main(): Waited on 3 threads. Final value of count = 3000000. Done.
```

Atomic Add Solution

- The pthread mutex solution works and is portable but is quite expensive for a simple add operation.
- Far more efficient would be to use the GCC specific built-in function specifically for incrementing values:

  __atomic_add_fetch(type* ptr, type val, int memorder)

  - **type** is an overloaded integral or pointer type referring to memory sizes of 1, 2, 4, or 8 bytes. This is determined automatically by the compiler.
  - **ptr** is a pointer to the field to increment
  - **val** is the amount to add to the current value dereferenced by ptr
  - **memorder** is a hint to the compiler about code motion in optimization. In this lab we will use the value __ATOMIC_RELAXED for this argument.
  - Documentation for GCC atomics

Built-In Functions

- Built-in functions are functions provided by the compiler. They often times give access to an optimized set of assembly instructions that are normally hard to express from standard C code.
- The __atomic_add_fetch() built-in function gives access to the “lock add” x86 assembly instruction which performs the load, increment, and store atomically.
- Generally, a fallback implementation should be provided for compiling with other compilers that may not have the same built-in function. For GCC this is outlined in the following example:

```
#ifdef __GNUC__
   __atomic_add_fetch(/* arguments */);
#else
   /* pthread mutex solution */
#endif
```

Conclusion

- PThreads provides a standard interface for multi-threaded programming available on many platforms
- Multi-threaded software architectures have many advantages over multi-process architectures but:
  - Concurrent access to shared data creates critical sections which can cause incorrect shared data
- Concurrency control in the form of mutex locks, also called semaphores, is the most common way to ensure correct multi-threaded programs
- Other types of concurrency control exist, with various advantages and disadvantages

Conclusion

- Creation, exposure and use of concurrency arises in many different situations
- Shared data access must be properly controlled for correct programs
- The PThreads mutex you have seen here is a classic example
- Principles of concurrent programming in PThreads also apply in a wide range of other situations
  - Linux Kernel internals among others
  - There are at least 7 different kinds of concurrency control in the Linux Kernel