CMSC 216
Introduction to Computer Systems
Lecture 22
Concurrent Programming

Administrivia

- Read Chapter 12 on Concurrent Programming

Chapter 12, Bryant and O'Hallaron

CONCURRENT PROGRAMMING (CONT)

Semaphore vs. Mutex

- Semaphores – counted or binary
- Looked at <semaphore.h>
- Alternative is mutex – only binary semaphore and defined in pthread.h
- mutex = “mutual exclusion” lock

Pthread Synchronization

- TO initialize – these are equivalent
- pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

or

- pthread_mutex_init(&mutex, NULL);
  - Returns 0 on success (non-zero on failure)
- pthread_mutex_lock(*pthread_mutex)
  - Returns 0 on success (non-zero on failure)
- pthread_mutex_unlock(*pthread_mutex)
  - Returns 0 on success (non-zero on failure)
- pthread_mutex_destroy(*pthread_mutex)
  - Returns 0 on success (non-zero on failure)

Thread safety

- Thread-safe functions are those that always produce correct results when called by concurrent threads
- One class of thread-unsafe functions is those functions that don't protect shared variables
- Use of semaphores to protect access to shared variables is one way we can make thread-unsafe functions thread-safe
- The fixed example will now report a final count of 20000000, as we expect, but it runs significantly slower

Thread safety, cont.

- Functions that keep state between invocations are also thread-unsafe
  - state can be held by global and/or static vars
- Consider this implementation of a pseudo-random number generator:

```
unsigned int last;

int rand(void) {
  last = last*1103515245 + 12345;
  return (unsigned int) (last / 65536) % 32768;
}

void srand(unsigned int seed) {
  last = seed;
}
```

Thread safety, cont.

- We can only fix this by requiring callers to supply the seed value on each call, eliminating use of global/static data:

```
int rand(unsigned int *last) {
  *last = *last * 1103515245 + 12345;
  return (unsigned int) (*last / 65536) % 32768;
}
```

- Although now all calls to this function require changing, which could lead to bugs, especially in larger programs
- If your code is threaded and relies on a dependable sequence of results, you cannot have global (or static) data accessed in your functions called by threads
  - protecting the shared data with semaphores helps, but still doesn’t ensure ordering across threads

Thread safety, cont.

- Some functions also return pointers to static data:

```
char *itoa(int n) {
  static char buffer[50];
  sprintf(buffer, "%d", n);
  return buffer;
}
```

- This approach breaks down if multiple threads use the function, as one thread could end up using another thread's results
- Solution: protect calls to these functions with semaphores, and make **deep** copies before allowing other threads access to the function

Reentrancy

- A reentrant function relies on no shared data at all
- "thread safe" ≠ "reentrant"
  - "x is reentrant" implies "x is thread safe"
- We made a reentrant form of the rand() function earlier
- Unix systems provide reentrant versions of most thread-unsafe functions, but their interfaces sometimes vary across platforms

Race condition

- A race condition occurs when a program depends on one thread reaching a point x before another threads reaches a point y
- The following program is supposed to print four points on the line **`y = 3x + 2`**, but instead, prints the same point **`(3,11)`** 4 times, due to a race condition

Eliminating the race condition

- The struct being passed to the peer threads is changed by the main thread before the peer threads access it!
- We can use dynamically allocated memory to make sure each thread has a struct dedicated to it
- The main thread will create the struct, and each thread will destroy its struct once it has obtained all the needed values from the struct

Sections 7.6-7.13, Bryant and O'Hallaron

LIBRARIES

Motivation

- Suppose we wrote some really useful functions to do something, and want to distribute them to clients or to other programmers. How can we do this?
  - give out the source code
  - give out the object code
  - in the form of a library
- What's a library? Basically a collection of object files that provide compiled functions performing some related tasks (often utility functions)
- Libraries can be linked into programs
  - linking can happen prior to execution (at compilation)
  - linking can be done **during** program execution

Comparison

- Giving out source code is platform-independent, but it needs to be recompiled and relinked by every client or user. It exposes our intellectual property or trade secrets, which we may not want to do
  - it also makes details of the implementation visible, and clients may come to rely on that
- Giving out object code doesn't require recompilation of that object code, but it requires relinking of the application which is going to use it
- Giving out either object code or a library is platform-dependent, but all we have to provide besides the object code or library is the header file (at most), not the source code
- Giving out some types of libraries doesn't even require relinking of the application using it

Object code vs. library

- In UNIX systems the linker includes an entire object file in an executable, even if not all the functions in it are used. With libraries, the linker can include the code for only the functions from the library that are actually called by a program
- The linker has to search through an object file to find each function, but a library can be indexed for faster lookup by the linker, so compilation is slightly faster
- Some types of libraries allow different executables to share the same library code, saving disk and memory space
- The UNIX utility **`nm`** lists the symbols (functions and other names) in a library

Types of libraries

- Static libraries (extension **`.a`**, for "archive")
  - are linked into a program as part of the linking phase of compilation
  - require space in each executable that uses them, which uses disk space, and memory space during execution
  - updating a library requires recompiling (relinking) all applications using it
  - are easy to use
- Shared libraries (extension **`.so`**, for "shared object")
  - are linked into a program at program startup, or during execution
  - require only one copy for the entire system
  - libraries can be updated independent of applications
  - must have version numbers associated with them, to control which version works with which applications

Dynamically loading libraries

- Functions in a library that is dynamically loaded can be loaded into an application during execution, not just at program startup
- This enables an application to load different libraries (functions) depending upon input while it's running
- This allows for things like skins, browser plugins, etc.
- Dynamically loading a library is more work for the programmer - using a shared library in the normal way doesn't require writing code specially, but dynamically loading a library requires that it first be explicitly opened by the program, and everything from the library that is then used must be explicitly looked up and loaded into memory

Creating a static library

- To create a static library:
  - the UNIX utility **`ar`** creates a library from a group of object files
  - example rules in a Makefile to create a library **`libavl.a`** from two object files **`avl.o`** and **`node.o`**:

```
LIBRARY_TO_CREATE = libavl.a
OBJS = avl.o node.o
...
ar cru $(LIBRARY_TO_CREATE) $(OBJS)
```