Lecture 19: Shared Memory & Synchronization

COMP 524 Programming Language Concepts
Stephen Olivier
April 16, 2009

Based on notes by A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts

Forking

```
int pid;
pid = fork();
// Error occurred
if (pid < 0) {
    cerr << "main: Fork failed!" << endl;
    exit(-1);
} else if (pid == 0) {
    cout << "Main Thread" << endl;
} else {
    cout << "Child start" << endl;
    cout << "Child complete" << endl;
    exit(0);
}
```

Synchronization

- One of the most fundamental issues in concurrent systems is how to ensure that different threads do not interfere with each other.

Atomic instructions

- One of the most important tools for implementing synchronizations protocols are atomic instructions.
- **Atomic instructions** are multiple instructions that are treated as one.
- For example, **Test-and-set** sets a boolean variable to true and returns the previous value.

Busy Waiting

- Under **Busy Waiting** a process continually attempts to access a “critical section” until it is free.
- Busy waiting is often implemented by a **spin lock**.

Barriers

- Barriers stop all threads (or a set of threads) until they reach a certain point.
- Busy waiting is one way implement these.
  - There are some performance issues
- Tree-based barriers for O(log(n)) time

Semaphores

- Semaphore is the first synchronization method.
- A Semaphore has one of two states, up or down.
- If the semaphore is up, then a process can acquire the semaphore and change its state to down.
- If a semaphore is down, then no process can acquire the semaphore.
- There can exist semaphores that have “multiple ups”

Deadlock

- Deadlock occurs when two processes attempt to acquire “nested” resources.
- e.g., Task one requests “A then B” and task two requests “B then A”.
- Djikstra calls this “the deadly embrace.”

Monitors

- Monitors are similar to semaphores, except that they are directly associated with resources and a set of procedures.

```
monitor account {
  int balance := 0
    function withdraw(int amount) {
    if amount < 0 then error "Amount may not be negative"
    else if balance < amount then error "Insufficient funds"
    else balance := balance - amount
  }
  function deposit(int amount) {
    if amount < 0 then error "Amount may not be negative"
    else balance := balance + amount
  }
}
```

Conditional Critical Regions

- Conditional critical Regions are similar to monitors, except that they specify regions of code over which only one process may execute.

```
region protected_variable when Boolean_condition do
...
end region.
```

Java Synchronization

- Before Java 5, only through use of the synchronized construct
  - Controls access to an object

```
class class_name {
   type method_name() {
      synchronized (object) {
         statement block
      }
   }
}
```

Java Synchronization

- Syntactic sugar lets us specify an entire method as synchronized at definition
  - Implicit object is **this**

```
rclass class_name {
    synchronized type method_name() {
        statement block
    }
}
```

Java 5 Synchronization

- Now Java has locks:

```
Lock l = ...;
    l.lock();
    try {
        // access the resource protected by this lock
    } finally {
        l.unlock();
    }
```

Java 5 Synchronization

- Condition variables also built in now:

```
Condition conditionVariable = l.newCondition();

...

boolean somecondition; //evaluate your wait criteria
while(somecondition){
  conditionVariable.await();
  //re-evaluate somecondition
}
```

Pthreads

- POSIX threading library for unix-based systems
  - Windows variants exist
- Used in conjunction with C/C++
- Relatively low level of abstraction
  - Supports explicit thread creation, management, synchronization, scheduling

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;
     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

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

- **thread** - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
- **attr** - Set to NULL if default thread attributes are used. (else define members of the struct pthread_attr_t defined in bits/pthreadtypes.h)
- Attributes include:
  - **detached state** (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
  - **scheduling policy** (real-time? PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED, SCHED_OTHER)
  - **scheduling parameter**
  - **inherit sched attribute** (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
  - **scope** (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
  - **guard size**
  - **stack address** (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
  - **stack size** (default minimum PTHREAD_STACK_SIZE set in pthread.h),
- **void * (*start_routine)** - pointer to the function to be threaded. Function has a single argument: pointer to void.
- ***arg** - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

```
void pthread_exit(void *retval);
```

- **retval** - “Return” value of thread.
- Pthreads don’t return values, but if the thread isn’t detached, then the thread ID and return value may be examined by another thread using “pthread_join.”
- *retval must not be local, otherwise it would cease to exist one the thread terminates

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *functionC();

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;
   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) ){
      printf("Thread creation failed: %d\n", rc1); }
   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) ){
      printf("Thread creation failed: %d\n", rc2);}
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);
   exit(0);
}
void *functionC()
{
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

```
#include <stdio.h>
#include <pthread.h>
#define NTHREADS 10
void *thread_function(void *);

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;
main()
{
   pthread_t thread_id[NTHREADS]; int i, j;
   for(i=0; i < NTHREADS; i++) {
      pthread_create( &thread_id[i], NULL, thread_function, NULL ); }
   for(j=0; j < NTHREADS; j++) {
      pthread_join( thread_id[j], NULL); }
   printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr)
{
   printf("Thread number %ld\n", pthread_self());
   pthread_mutex_lock( &mutex1 );
   counter++;
   pthread_mutex_unlock( &mutex1 );
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t count_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;
void *functionCount1();
void *functionCount2();
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6

main()
{
   pthread_t thread1, thread2;
   pthread_create( &thread1, NULL, &functionCount1, NULL);
   pthread_create( &thread2, NULL, &functionCount2, NULL);
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);
   exit(0);
}
```

```
void *functionCount1() {
   for(;;) {
      pthread_mutex_lock( &condition_mutex );
      while( count >= COUNT_HALT1 && count <= COUNT_HALT2 ){
         pthread_cond_wait( &condition_cond, &condition_mutex );}
      pthread_mutex_unlock( &condition_mutex );
      pthread_mutex_lock( &count_mutex );
      count++;
      printf("Counter value functionCount1: %d\n",count);
      pthread_mutex_unlock( &count_mutex );
      if(count >= COUNT_DONE) return(NULL);}
}

void *functionCount2(){
    for(;;) {
       pthread_mutex_lock( &condition_mutex );
       if( count < COUNT_HALT1 || count > COUNT_HALT2 ){
          pthread_cond_signal( &condition_cond );}
       pthread_mutex_unlock( &condition_mutex );
       pthread_mutex_lock( &count_mutex );
       count++;
       printf("Counter value functionCount2: %d\n",count);
       pthread_mutex_unlock( &count_mutex );
       if(count >= COUNT_DONE) return(NULL);}
}
```

Remote Procedure Call (RPC)

- Message passing (rather than shared memory) approach to communication
  - Works across distributed systems
- Allows higher level of abstraction than network API's like sockets
  - Leverage type checking and/or OO programming
- Main problem is packing, sending, and unpacking parameters efficiently while preserving semantics
  - This is called **marshalling**

RPC Implementations

- CORBA is a language & OS independent solution
  - Free & commercial implementations
- Microsoft DCOM and later .NET remoting
- Java Remote Method Invocation (RMI)