Synchronization *con't*

+ Condition Variables

- **Locking is a simple kind of resource scheduling -- one thread at a time may enter a critical section.**
- **What about more complicated scheduling policy?**
  - Supposed we need a mechanism to block thread(s) until some condition is true?
- **Condition variables are synchronization variables that are used for *signaling* that some *condition* is met and that any *waiting threads* can proceed.**

+ Pthread Condition Variable Functions

- **Pthreads defines three basic operations on condition variables.**
  - **`int pthread_cond_init(cond, …)`**
    - **Takes two arguments, the first of which is the condition variable itself. The second we don't care about.**
  - **`int pthread_cond_wait(cond, mutex)`**
    - **The calling thread will wait until the condition represented by the `cond` variable is met.**
  - **`int pthread_cond_signal(cond)`**
    - **Sends a signal that wakes up exactly one thread that is waiting due to a call to `pthread_cond_wait`.**

+ Condition Variable Example

- **Example: Three threads collaborating**
  - Two threads that increment a global counter.
  - One is waiting for a signal that the work is done.
  - See lecture27/condition_vars/cond_var.c

+ Waiting on a Condition

- **Another example: suppose we want one function on one thread to produce a value and another function on another thread to consume that value?**

```c
typedef struct {
  int* val;
} channel;

static channel c;

void send(int* v) {
  if (c->val == NULL) {
    c->val = v;
  } else {
    // wait until null
  }
}
```

```c
int* receive() {
  if (c->val != NULL) {
    int *v = c->val;
    c->val = NULL;
    return v;
  } else {
    // wait until non-null
  }
}
```

+ Condition Variable Example

```c
typedef struct {
  int* val;
} channel;

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

static channel c;

void send(int* v) {
   pthread_mutex_lock(&m);
   while (c.val != NULL) {
     pthread_cond_wait(&cv, &m);
   }
   c.val = v;
   pthread_mutex_unlock(&m);
}
```

```c
int* receive() {
   pthread_mutex_lock(&m);
   if (c.val) {
     int* v = c->val;
     c.val = NULL;
     pthread_cond_signal(&cv);
     pthread_mutex_unlock(&m);
     return v;
   }else {
     pthread_mutex_unlock(&m);
     return NULL;
   }
}
```

- **See lecture27/condition_vars/cond_channel.c**

Conditional Variable Usage

- **General pattern:**
  - T1:

```
lock(&m);
while (condition != true)
 cond_wait(&cv, &m)
... do stuff...
unlock(&m)
```

  - T2:

```
lock(&m)
condition = true
cond_signal(&cv)
unlock(&m)
```

+ Barrier Synchronization

- **A *barrier* is another type of synchronization method.**
- **A barrier for a group of threads means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.**
- **Barriers are used in concurrent programs whose threads must progress at roughly the same rate.**
- **Imagine we wanted to parallelize a sorting algorithm, like merge sort.**
  - We would need threads waiting for each other in order to do the merging!

+ Barrier Synchronization Implementation

- **Using condition variables we can implement our own barrier library pretty easily.**
- **One new pthreads method we need to know, however.**
  `pthread_cond_broadcast(&cond);`
  - Similar to `pthread_cond_signal`, except it wakes up all threads, not just one.
- **See lecture27/barrier/***

+ Semaphores

- **Semaphore: non-negative global integer synchronization variable. Manipulated by P and V operations.**
- **P(s)**
  - If s is nonzero, then decrement by 1 and return atomically.
  - If s is zero, then suspend thread until s becomes nonzero and the thread is restarted by a V operation.
    - After restarting, the P operation decrements s and returns control to the caller.
- **V(s):**
  - Increment s by 1 atomically
  - If there are any threads blocked in a P operation waiting for s to become non-zero, then restart exactly one of those threads, which then completes its P operation by decrementing s.
- **Can be used to synchronize *processes* in addition to threads**

+ C Semaphore Functions

- **There are three basic operations defined for a semaphore.**
  - **`int sem_init(sem_t *s, 0, unsigned int val);`**
    - A program initializes a semaphore by calling this function.
    - Initializes semaphore sem to value
  - **`int sem_wait(sem_t *s);  /* P(s) */`**
    - `P` operation
  - **`int sem_post(sem_t *s);  /* V(s) */`**
    - `V` operation

+ Semaphores vs Mutexes

- **Mutex: exclusive access to a resource**
- **Semaphore: n-party access to a resource**
- **Semaphores can be used for mutual exclusion:**

```
sem_init(&s,..,1);
sem_wait(); // lock()
// critical section
sem_post()  // lock()
```

- **Semaphores can be used in place of conditional variable as well**
- **Prefer mutexes and conditional variables rather than semaphores, they lead to simpler and more readable code.**

Thread-safety

+ Crucial concept: Thread Safety

- **Functions called from a thread must be thread-safe**
- **Def: A function is thread-safe iff it will always produce correct results when called repeatedly from multiple concurrent threads**
- **Classes of thread-unsafe functions:**
  - Class 1: Functions that do not protect shared variables
  - Class 2: Functions that keep state across multiple invocations
  - Class 3: Functions that return a pointer to a static variable
  - Class 4: Functions that call thread-unsafe functions ☺

Thread-Unsafe Functions (Class 1)

- **Failing to protect shared variables**
  - Fix: Use `lock` and `unlock` mutex operations
  - Issue: Synchronization operations will slow down code

+ Thread-Unsafe Functions (Class 2)

- **Relying on persistent state across multiple function invocations**
  - Example: Random number generator that relies on static state

```c
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Thread-Safe Random Number Generator

- **Pass state as part of argument**
  - and, thereby, eliminate global state

```c
/* rand_r - return pseudo-random integer on 0..32767 */

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

- **Consequence: programmer using rand_r must provide seed**

+ Thread-Unsafe Functions (Class 3)

- **Returning a pointer to a static variable**
- **Fix 1. Rewrite function so caller passes address of variable to store result**
  - Requires changes in caller and callee
- **Fix 2. Lock-and-copy**
  - Requires simple changes in caller (and none in callee)

```c
/* lock-and-copy version */
char *ctime_ts(const time_t* timep,
                     char* privatep)
{
    pthread_mutex_lock(&mutex);
    // ctime returns ptr to static
    char* sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    pthread_mutex_unlock(&mutex);
    return privatep;
}
```

+ Thread-Unsafe Functions (Class 4)

- **Calling thread-unsafe functions**
  - Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - Fix: Modify the function so it calls only thread-safe functions or not use it!

+ Thread-Safe Library Functions

- **All functions in the Standard C Library are thread-safe**
  - Examples: `malloc, free, printf, scanf`
- **Most Unix system calls are thread-safe, with a few exceptions:**

Thread-unsafe function	Class	Safe version
`asctime`	3	`asctime_r`
`ctime`	3	`ctime_r`
`gethostbyaddr`	3	`gethostbyaddr_r`
`gethostbyname`	3	`gethostbyname_r`
`inet_ntoa`	3	`(none)`
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`

+

Concurrent Programming in Java

+ Concurrent Programming in Java

- **The Java platform has a number of constructs to support concurrent programming.**
  - Thread model still prevalent
  - Other mechanisms introduced in Java 5 with the `java.util.concurrent` packages.
- **There is a lot more than we can cover in a few slides, but we'll try to give you a taste.**

+ Processes & Threads

- **A Java application can create additional processes using a `ProcessBuilder` object.**
  - We are not going to talk any more about that since…
- **In Java, concurrent programming is mostly concerned with threads.**
- **Like pthreads…**
  - Every running program has at least one thread — or several, if you count JVM "system" threads that do things like memory management and signal handling.
  - From the programmer's point of view, you start with just one thread, called the main thread.
  - The main thread has the ability to create additional threads.

Defining & Starting a Thread: Runnable

- **Provide a `Runnable` object**
  - The `Runnable` interface defines a single method, `run`, meant to contain the code executed in the thread

```java
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

+ Defining & Starting a Thread: Runnable

- **Subclass Thread.**
  - The `Thread` class itself implements `Runnable`, though its `run` method does nothing.

```java
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

+ Basic Thread Behaviors

- **`sleep`**
  - `Thread.sleep` causes the current thread to suspend execution.
  - Is an efficient means of making processor time available to the other threads.
- **`interrupted`**
  - An interrupt is an indication to a thread that it should stop what it is doing and do something else. Program-specific semantics.
  - It's up to the programmer to decide how to respond an interrupt.
  - An interrupt can occur in the form of `InterruptedException`
- **`join`**
  - The `join` method allows one thread to wait for the completion of another.

+ Java Thread Example

- **An example Java program that creates some threads and makes use of the basic features we've covered so far.**
  - See lecture27/java/SimpleThreads.java

+ Synchronization

- **Java provides two synchronization idioms: synchronized methods and statements.**
- **To make a method synchronized, add the synchronized keyword to its declaration:**

```java
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized int value() {
        return c;
    }
}
```

+ Synchronized Methods

- **It is not possible for two invocations of synchronized methods on the same object to interleave.**
- **When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block until the first thread is done.**
- **What is the mutex variable? The object itself.**
  - Known as an *intrinsic lock* or *monitor*
- **Synchronized method are effective, but can present problems with *liveness*.**
  - *Coarse-grained synchronization*

Synchronized Statements

- **Another way to create synchronized code is with synchronized statements.**
- **Synchronized statements must specify the object acts as the mutex variable.**

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

- **Synchronized statements are useful for improving concurrency with *fine-grained synchronization*.**

Java Synchronization Example

- **An example Java class utilizes synchronized methods and blocks.**
  - See lecture27/java/SynchronizedExample.java

+ Other Concurrency Features

- **Guarded Blocks**
  - Like condition variables
- **CyclicBarriers**
  - Like barriers
- **Semaphore**
  - Like, well, semaphores
- **Concurrent Collections**
  - Thread-safe data structures!
  - Ex. `BlockingQueue` defines a first-in-first-out data structure where all operations are atomic.
- **Atomic Variables**
  - Classes that support atomic operations on single variables
  - Ex. `AtomicInteger` provides an atomic `incrementAndGet` method

Modern Concurrency

- **Many modern programming languages are building in concurrency into their languages as a first principle.**
- **Here is an example in Scala, which is another language that runs on the JVM.**

```scala
object ParallelSum extends App {

  // Build a list of 10 million integers
  val integers = (1L to 100000000L).toList
  // Concurrency is built into the Scala collections API
  // using .par after the collection name, the workf for any methods
  // that are executed thereafter are distributed across threads.
  val sum = integers.par.sum
  // Print
  println(s"sum = $sum")

}
```