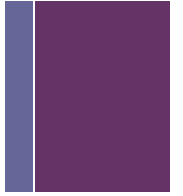




+

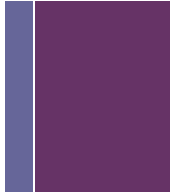
Synchronization *con't*

+ Condition Variables



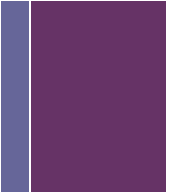
- Locking is a simple kind of resource scheduling -- one thread at a time may enter a critical section.
- What about more complicated scheduling policy?
 - Supposed we need a mechanism to block thread(s) until some condition is true?
- Condition variables are synchronization variables that are used for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

+ Pthread Condition Variable Functions



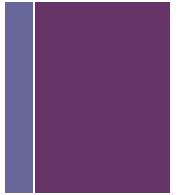
- Pthreads defines three basic operations on condition variables.
- `int pthread_cond_init(cond, ...)`
 - Takes two arguments, the first of which is the condition variable itself. The second we don't care about.
- `int pthread_cond_wait(cond, mutex)`
 - The calling thread will wait until the condition 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

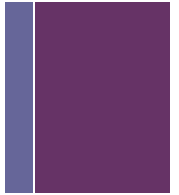


- Another example: suppose we want one function on one thread to produce a value and another function on another thread to consume that value?

```
1  typedef struct {
2      int* val;
3  } channel;
4
5  static channel c;
6
7  void send(int* v) {
8      if (c->val == NULL) {
9          c->val = v;
10     } else {
11         // wait until null
12     }
13 }
```

```
15  int* receive() {
16      if (c->val != NULL) {
17          int *v = c->val;
18          c->val = NULL;
19          return v;
20      } else {
21          // wait until non-null
22      }
23 }
```

+ Condition Variable Example

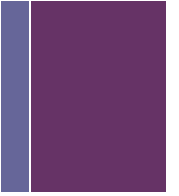


```
1  typedef struct {
2      int* val;
3  } channel;
4
5  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
6  pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
7
8  static channel c;
9
10 void send(int* v) {
11     pthread_mutex_lock(&m);
12     while (c.val != NULL) {
13         pthread_cond_wait(&cv, &m);
14     }
15     c.val = v;
16     pthread_mutex_unlock(&m);
17 }
```

```
19 int* receive() {
20     pthread_mutex_lock(&m);
21     if (c.val) {
22         int* v = c->val;
23         c.val = NULL;
24         pthread_cond_signal(&cv);
25         pthread_mutex_unlock(&m);
26         return v;
27     } else {
28         pthread_mutex_unlock(&m);
29         return NULL;
30     }
31 }
```

- See [lecture27/condition_vars/cond_channel.c](#)

+ Conditional Variable Usage



- **General pattern:**

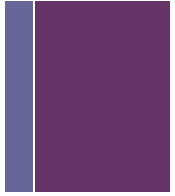
- T1:

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

- T2:

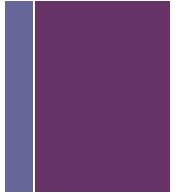
```
lock(&m)  
condition = true  
cond_signal(&cv)  
unlock(&m)
```

+ Barrier Synchronization



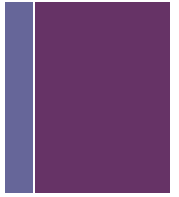
- A *barrier* is another type of synchronization method.
- A barrier for a group of threads means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.
- Barriers are used in concurrent programs whose threads must progress at roughly the same 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



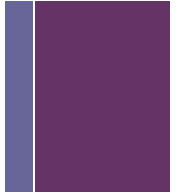
- Using condition variables we can implement our own barrier library pretty easily.
- One new pthreads method we need to know, however.
`pthread_cond_broadcast(&cond) ;`
 - Similar to `pthread_cond_signal`, except it wakes up all threads, not just one.
- See `lecture27/barrier/*`

+ Semaphores



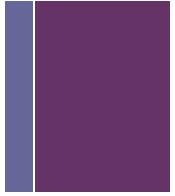
- **Semaphore: non-negative global integer synchronization variable. Manipulated by P and V operations.**
- **P(s)**
 - If s is nonzero, then decrement by 1 and return atomically.
 - If s is zero, then suspend thread until s becomes nonzero and the thread is restarted by a V operation.
 - After restarting, 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()
```

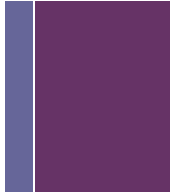
- **Semaphores can be used in place of conditional variable as well**
- **Prefer mutexes and conditional variables rather than semaphores, they lead to simpler and more readable code.**



+

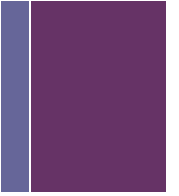
Thread-safety

+ Crucial concept: Thread Safety



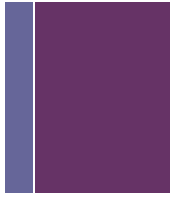
- **Functions called from a thread must be thread-safe**
- **Def: A function is thread-safe iff it will always produce correct results when called repeatedly from multiple concurrent threads**
- **Classes of thread-unsafe functions:**
 - Class 1: Functions that do not protect shared variables
 - Class 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

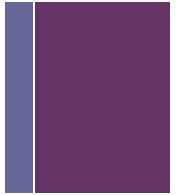
+ Thread-Unsafe Functions (Class 2)



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

```
1  static unsigned int next = 1;
2
3  /* rand: return pseudo-random integer on 0..32767 */
4  int rand(void)
5  {
6      next = next*1103515245 + 12345;
7      return (unsigned int)(next/65536) % 32768;
8  }
9
10 /* srand: set seed for rand() */
11 void srand(unsigned int seed)
12 {
13     next = seed;
14 }
```


+ Thread-Safe Random Number Generator

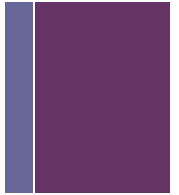


- Pass state as part of argument
 - and, thereby, eliminate global state

```
1  /* rand_r - return pseudo-random integer on 0..32767 */
2
3  int rand_r(int *nextp)
4  {
5      *nextp = *nextp * 1103515245 + 12345;
6      return (unsigned int)(*nextp/65536) % 32768;
7  }
```

- Consequence: programmer using rand_r must provide seed

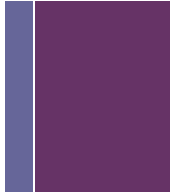
+ Thread-Unsafe Functions (Class 3)



- 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)

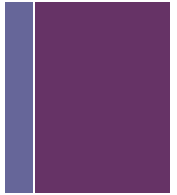
```
1  /* lock-and-copy version */
2  char *ctime_ts(const time_t* timep,
3                  char* privatep)
4  {
5      pthread_mutex_lock(&mutex);
6      // ctime returns ptr to static
7      char* sharedp = ctime(timep);
8      strcpy(privatep, sharedp);
9      pthread_mutex_unlock(&mutex);
10     return privatep;
11 }
```

+ 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



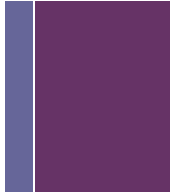
- All functions in the Standard C Library are thread-safe
 - Examples: `malloc`, `free`, `printf`, `scanf`
- Most Unix system calls are thread-safe, with a few exceptions:

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



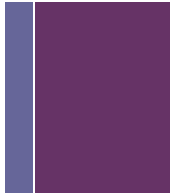
Concurrent Programming in Java

+ Concurrent Programming in Java



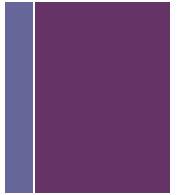
- **The Java platform has a number of constructs to support concurrent programming.**
 - Thread model still prevalent
 - Other mechanisms introduced in Java 5 with the `java.util.concurrent` packages.
- **There is a lot more than we can cover in a few slides, but we'll try to give you a taste.**

+ 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.

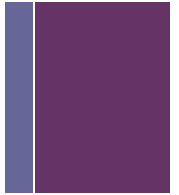
+ Defining & Starting a Thread: Runnable



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

```
1  public class HelloRunnable implements Runnable {  
2      public void run() {  
3          System.out.println("Hello from a thread!");  
4      }  
5  
6      public static void main(String args[]) {  
7          (new Thread(new HelloRunnable())).start();  
8      }  
9  }
```


+ Defining & Starting a Thread: Runnable

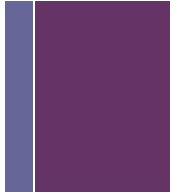


- **Subclass Thread.**

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

```
1  public class HelloThread extends Thread {
2      public void run() {
3          System.out.println("Hello from a thread!");
4      }
5
6      public static void main(String args[]) {
7          (new HelloThread()).start();
8      }
9  }
```

+ 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.

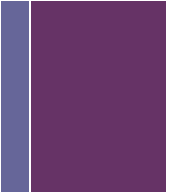
- **interrupted**

- 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**

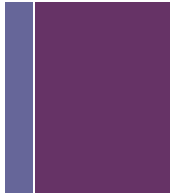
- The `join` method allows one thread to wait for the completion of another.

+ 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`

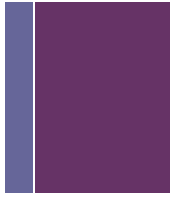
+ Synchronization



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

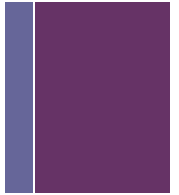
```
1  public class SynchronizedCounter {  
2      private int c = 0;  
3  
4      public synchronized void increment() {  
5          c++;  
6      }  
7  
8      public synchronized int value() {  
9          return c;  
10     }  
11 }
```

+ 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 methods are effective, but can present problems with *liveness*.
 - *Coarse-grained synchronization*

+ Synchronized Statements

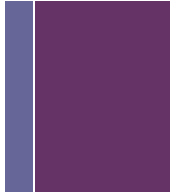


- Another way to create synchronized code is with synchronized statements.
- Synchronized statements must specify the object acts as the mutex variable.

```
1  public void addName(String name) {  
2      synchronized(this) {  
3          lastName = name;  
4          nameCount++;  
5      }  
6      nameList.add(name);  
7  }
```

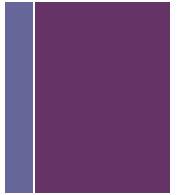
- Synchronized statements are useful for improving concurrency with *fine-grained synchronization*.

+ Java Synchronization Example



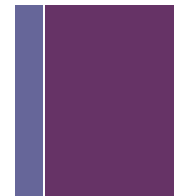
- **An example Java class utilizes synchronized methods and blocks.**
 - See `lecture27/java/SynchronizedExample.java`

+ 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.

```
1  object ParallelSum extends App {  
2  
3      // Build a list of 10 million integers  
4      val integers = (1L to 100000000L).toList  
5      // Concurrency is built into the Scala collections API  
6      // using .par after the collection name, the work for any methods  
7      // that are executed thereafter are distributed across threads.  
8      val sum = integers.par.sum  
9      // Print  
10     println(s"sum = $sum")  
11  
12 }
```