Concurrent Programming

Mid-Sem Examination

1. Markup/programming hybrid
   - Markup languages extended to support some programming
   - Examples: XHTML, MXML (Action Script)

```
<mx:Button id="btn" label="MyButton" height="100" />
```

```
var btn:Button = new Button();
btn.label = "MyButton";
btn.height = 100;
```

2. Heterogeneous Programming:
   1. For CPUs and accelerators like GPUs, FPGAs, Phis, ASICs
   2. OpenCL, CUDA, OpenACC, OpenMP
3. Orthogonality
   - A relatively small set of primitive constructs can be combined in a relatively small number of ways
   - Every possible combination is legal
   - C: function cannot return a static array, adding two pointers

Mid-Sem Examination

4. Tokens by Reg Exp, syntactic structure by CFG. Function parameters matching can be accepted by CFG. If-then-else cannot be accepted by CFG.
5. Shift-reduce parsers can accept left recursive grammars. Expression grammars as an example.
6. Binding ways – language design (operator), language implementation (float), compile time (variable), load time (static), runtime (dynamic)
7. Static scoping: m.x=7, m.y=8, Q() -> Q.x=3, Q.y=4, P(4) ->m.x=6, print(Q.x/3), print(m.x/6) -> (3,6)

   Dynamic Scoping: m.x=7, m.y=8, Q() -> Q.x=3, Q.y=4,

   P(4)-> Q.x=6, print(Q.x/6), print(m.x/7) -> (6,7)
8. E -> id { E.type = id.type }

   E -> int | float { E.type = int.type | float.type}

   E -> E1 + E2 { E.type = resultant (E1.type, E2.type)

Mid-Sem Examination

```
E → id[ E1 ]  { t1 = id.type;
                 if (t1 == ARRAY ^ E1.type == INTEGER)
                      E.type = id.type;
                   else
                      E.type = error;
               }
```

Test Program

```
int x, y;
function P(intn) {
        x = (n + 2) * (n – 3)
}

function Q() {
        int x, y;
       {
        x = 3; y = 4;
        P(y);
        print (x);
        }
}

main() {
        x = 7;y = 8;
        Q();
        print (x);
}
```

Syllabus

Lecture Series (hours)	Topics
1-4	Introduction and Motivation, Paradigms
5-10	Syntax and Semantics, BNF, Compilation
11-18	Data Types, Constructs, Functions, Activation Records, Names and Bindings
19-28	Concurrency, Functional PLs, Logical PLs, Lambda Calculus, Event driven programming
29-36	Virtual Machines, Managed Languages, JIT, Case study

Concurrency

Two or more sequences of events occur “in parallel”

- Multiprogramming
  - Single processor runs several programs at the same time
  - Each program proceeds sequentially
  - Actions of one program may occur between two steps of another
- Multiprocessors
  - Two or more processors
  - Programs on one processor communicate with programs on another
  - Actions may happen simultaneously

Process: program running on a processor

The Promise of Concurrency

- Speed
  - If a task takes time t on one processor, shouldn’t it take time t/n on n processors?
- Availability
  - If one process is busy, another may be ready to help
- Distribution
  - Processors in different locations can collaborate to solve a problem or work together
- Humans do it so why can’t computers?
  - Vision, cognition appear to be highly parallel activities

Example: Rendering a Web page

- Page is a shared resource
- Multiple concurrent activities in the Web browser
  - Thread for each image load
  - Thread for text rendering
  - Thread for user input (e.g., “Stop” button)
- Cannot all write to page simultaneously!
  - Big challenge in concurrent programming: managing access to shared resources

The Challenges of Concurrency

- Concurrent programs are harder to get right
  - Folklore: need at least an order of magnitude in speedup for concurrent program to be worth the effort
- Some problems are inherently sequential
  - Theory – circuit evaluation is P-complete
  - Practice – many problems need coordination and communication among sub-problems
- Specific issues
  - Communication – send or receive information
  - Synchronization – wait for another process to act
  - Atomicity – do not stop in the middle and leave a mess

Language Support for Concurrency

- Threads (or process)
  - Think of a thread as a system “object” containing the state of execution of a sequence of instructions
  - Each thread needs a separate run-time stack
- Communication abstractions
  - Synchronous communication
  - Asynchronous buffers that preserve message order
- Concurrency control
  - Locking and mutual exclusion
  - Atomicity is more abstract, less commonly provided

Inter-Process Communication

- Processes may need to communicate
  - Process requires exclusive access to some resources
  - Process need to exchange data with another process
- Can communicate via:
  - Shared variables
  - Message passing
  - Parameters

Explicit vs. Implicit Concurrency

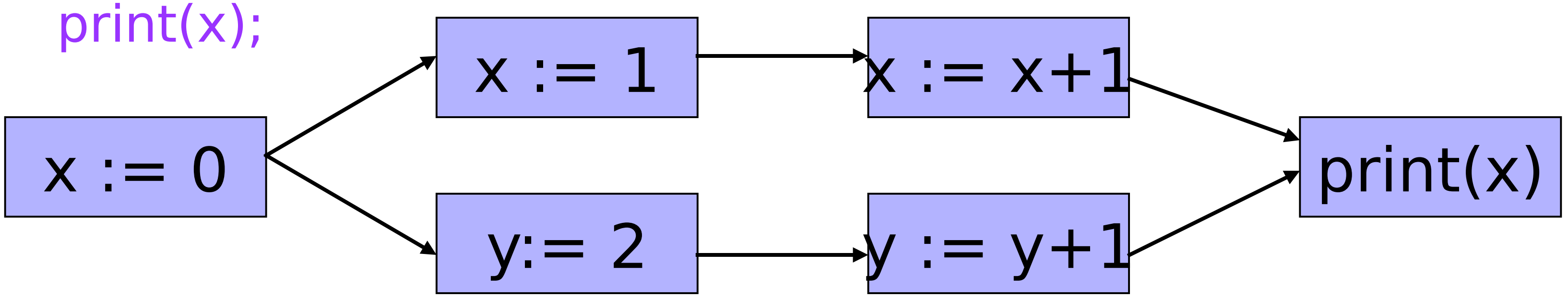
- Explicit concurrency
  - Fork or create threads / processes explicitly
  - Explicit communication between processes
    - Producer computes useful value
    - Consumer requests or waits for producer
- Implicit concurrency
  - Rely on compiler to identify potential parallelism
  - Instruction-level and loop-level parallelism can be inferred, but inferring subroutine-level parallelism has had less success

cobegin / coend

- Limited concurrency primitive
  - Concurrent Pascal [Per Brinch Hansen, 1970s]

```
x := 0; y = 0;
cobegin
   begin x := 1; x := x+1 end;
   begin y := 2; y := y+1 end;
coend;
print(x);
```

execute sequential blocks in parallel



Atomicity at level of assignment statement

Properties of cobegin/coend

- Simple way to create concurrent threads of execution (or processes)
- Communication by shared variables
- No mutual exclusion
- No atomicity
- Number of processes fixed by program structure
- Cannot abort processes
  - All must complete before parent process can go on

Race Conditions

- Race condition occurs when the value of a variable depends on the execution order of two or more concurrent processes
- Example

```
procedure signup(person)
   begin
     number := number + 1;
     list[number] := person;
end;
signup(joe) || signup(bill)
```

Critical Section

- Two concurrent processes may access a shared resource
- Inconsistent behavior if processes are interleaved
- Allow only one process in critical section
- Issues
  - How to select which process is allowed to access the critical section?
  - What happens to the other process?

Locks and Waiting

<initialize concurrency control>

Process 1:

```
        <wait>
        signup(joe);  // critical section
        <signal>
```

Process 2:

```
        <wait>
   signup(bill);   // critical section
        <signal>
```

Need atomic operations to implement wait

Deadlock

- Deadlock occurs when a process is waiting for an event that will never happen
- Necessary conditions for a deadlock to exist:
  - Processes claim exclusive access to resources
  - Processes hold some resources while waiting for others
  - Resources may not be removed from waiting processes
  - There exists a circular chain of processes in which each process holds a resource needed by the next process in the chain
- Example: “dining philosophers”

Implementing Mutual Exclusion

- Atomic test-and-set
  - Instruction atomically reads and writes some location
  - Common hardware instruction
  - Combine with busy-waiting loop to implement mutex
- Semaphore
  - Keep queue of waiting processes
    - Avoid busy-waiting loop
  - Scheduler has access to semaphore; process sleeps
  - Disable interrupts during semaphore operations
    - OK since operations are short

Semaphores

- Semaphore is an integer variable and an associated process queue
- Operations:
  - P(s) if s > 0 then s--
    else enqueue process
  - V(s) if a process is enqueued then dequeue it
    else s++
- Binary semaphore
- Counting semaphore

Simple Producer-Consumer

```
program
    SimpleProducerConsumer;
var buffer : string;
    full : semaphore = 0;
    empty : semaphore = 1;
begin
    cobegin
        Producer; Consumer;
    coend;
end.
```

```
procedure Producer;
var tmp : string
begin
  while (true) do begin
    produce(tmp);
    P(empty);  { begin critical section }
    buffer := tmp;
    V(full);   { end critical section }
  end;
end;
```

```
procedure Consumer;
var tmp : string
begin
  while (true) do begin
    P(full);   { begin critical section }
    tmp := buffer;
    V(empty);  { end critical section }
    consume(tmp);
  end;
end;
```

Producer-Consumer

```
program ProducerConsumer;
const size = 5;
var buffer : array[1..size] of string;
    inn    : integer = 0;
    out    : integer = 0;
    lock   : semaphore = 1;
    nonfull : semaphore = size;
    nonempty : semaphore = 0;  …
```

```
procedure Produce
var tmp : string
begin
    while (true) do begin
        produce(tmp);
        P(nonfull);
        P(lock);   { begin critical section }
        inn := inn mod size + 1;
        buffer[inn] := tmp;
        V(lock);   { end critical section }
        V(nonempty);
    end;
end;
```

```
procedure Consumer;
var tmp : string
begin
    while (true) do begin
        P(nonempty);
        P(lock);   { begin critical section }
        out = out mod size + 1;
        tmp := buffer[out];
        V(lock);   { end critical section }
        V(nonfull);
        consume(tmp);
    end;
end;
```

Monitors

- Monitor encapsulates a shared resource (monitor = “synchronized object”)
  - Private data
  - Set of access procedures (methods)
  - Locking is automatic
    - At most one process may execute a monitor procedure at a time (this process is “in” the monitor)
    - If one process is in the monitor, any other process that calls a monitor procedure will be delayed

Example of a Monitor

```
monitor Buffer;
const size = 5;
var buffer : array[1..size] of string;
    in      : integer = 0;
    out     : integer = 0;
    count   : integer = 0;
    nonfull : condition;
    nonempty : condition;  …
```

```
procedure put(s : string);

begin
    if (count = size) then
      wait(nonfull);
    in := in mod size + 1;
    buffer[in] := tmp;
    count := count + 1;
    signal(nonempty);
end;
```

```
function get : string;
var tmp : string
begin
    if (count = 0) then wait(nonempty);
    out = out mod size + 1;
    tmp := buffer[out];
    count := count - 1;
    signal(nonfull);
    get := tmp;
end;
```

Java Threads

- Thread
  - Set of instructions to be executed one at a time, in a specified order
  - Special Thread class is part of the core language
    - In C/C++, threads are part of an “add-on” library
- Methods of class Thread
  - start : method called to spawn a new thread
    - Causes JVM to call run() method on object
  - suspend : freeze execution (requires context switch)
  - interrupt : freeze and throw exception to thread
  - stop : forcibly cause thread to halt

java.lang.Thread

```
public class Thread implements Runnable {
      private char name[];
      private Runnable target;
      ...
      public final static int MIN_PRIORITY = 1;
      public final static int NORM_PRIORITY = 5;
      public final static int MAX_PRIORITY = 10;

      private void init(ThreadGroup g, Runnable target, String name) {...}

      public Thread() { init(null, null, "Thread-" + nextThreadNum()); }
      public Thread(Runnable target) {
         init(null, target, "Thread-" + nextThreadNum());
     }
      public Thread(Runnable target, String name) { init(null, target, name); }

      public synchronized native void start();

      public void run() {
         if (target != nul
             target.run();
         }
     }
}
```

What does this mean?

Creates execution environment for the thread
(sets up a separate run-time stack, etc.

Methods of Thread Class

```
public class Thread implements Runnable {
      ...
      public static native Thread currentThread();
      public static native void yield();
      public static native void sleep(long millis) throws InterruptedException;
      public static int enumerate(Thread tarray[])

      public static boolean interrupted() { ... }
     public boolean isInterrupted() { ... }
      public final native boolean isAlive();
      public String toString() {
      public void interrupt() { ... }
      public void interrupt() { ... }
      public final void stop() { ... }
      public final void suspend() { ... }
      public final void resume() { ... }
      public final void setPriority(int newPriority) {
      public final int getPriority() {
      public final void setName(String name) { ... }
      public final String getName() { return String.valueOf(name); }
      public native int countStackFrames();
      public final synchronized void join() throws InterruptedException {...}
      public void destroy() { throw new NoSuchMethodError(); }
}
```

Runnable Interface

- Thread class implements Runnable interface
- Single abstract (pure virtual) method run()

```
public interface Runnable {
    public void run(); }
```

- Any implementation of Runnable must provide an implementation of the run() method

```
public class ConcurrentReader implements Runnable {
    …
    public void run() { …
        … code here executes concurrently with caller
    … }
}
```

Two Ways to Start a Thread

- Construct a thread with a runnable object

```
ConcurrentReader readerThread = new ConcurrentReader();
Thread t = new Thread(readerThread);
t.start();   // calls ConcurrReader.run() automatically
```

… OR …

- Instantiate a subclass of Thread

```
class ConcurrWriter extends Thread { …
    public void run() { … } }
ConcurrWriter writerThread = new ConcurrWriter();
writerThread.start();   // calls ConcurrWriter.run()
```

- What happens if you can just call readerThread.run();

Why Two Ways?

- Java only has single inheritance
- Can inherit from some class, but also implement Runnable interface so that can run as a thread

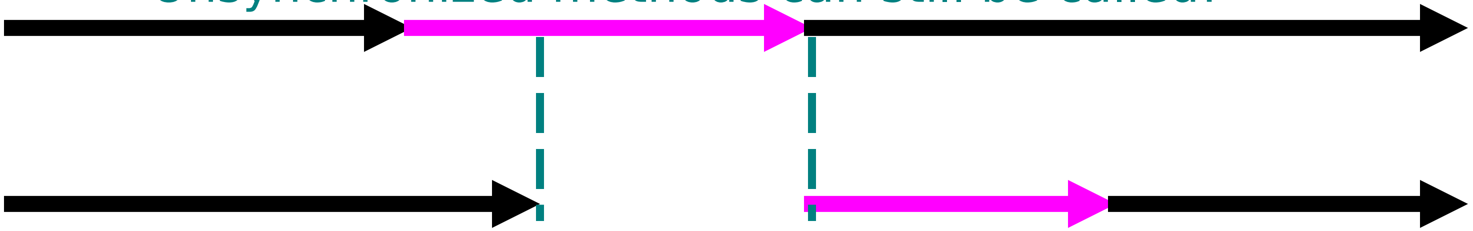
```
class X extends Y implements Runnable { …
    public synchronized void doSomething() { … }
    public void run() { doSomething(); }
}
X obj = new X();
obj.doSomething(); // runs sequentially in current thread
Thread t = new Thread(new X()); // new thread
t.start();   // calls run() which calls doSomething()
```

Interaction Between Threads

- Shared variables and method calls
  - Two threads may assign/read the same variable
    - Programmer is responsible for avoiding race conditions by explicit synchronization!
  - Two threads may call methods on the same object
- Synchronization primitives
  - All objects have an internal lock (inherited from Object)
  - Synchronized method locks the object
    - While it is active, no other thread can execute inside object
  - Synchronization operations (inherited from Object)
    - Wait: pause current thread until another thread calls Notify
    - Notify: wake up waiting thread

Synchronized Methods

- Provide mutual exclusion
  - If a thread calls a synchronized method, object is locked
  - If another thread calls a synchronized method on the same object, this thread blocks until object is unlocked
    - Unsynchronized methods can still be called!



- “synchronized” is <u>not</u> part of method signature
  - Subclass may replace a synchronized method with unsynchronized method

Wait, Notify, NotifyAll

```
public class Object {
      ...
      public final native void notify();
      public final native void notifyAll();

      public final native void wait(long timeout) throws InterruptedException;
      public final void wait() throws InterruptedException { wait(0); }
      public final void wait(long timeout, int nanos)
         throws InterruptedException { ... }
}
```

- wait() releases object lock, thread waits on internal queue
- notify() wakes the highest-priority thread closest to the front of the object's internal queue
- notifyAll() wakes up all waiting threads
  - Threads non-deterministically compete for access to object
  - May not be fair (low-priority threads may never get access)
- May only be called when object is locked (when is that?)

Using Synchronization

```
public synchronized void consume() {
    while (!consumable()) {
        wait();  }    // release lock and wait for resource
    ...   // have exclusive access to resource, can consume
}

public synchronized void produce() {
    ... // do something that makes consumable() true
    notifyAll();    // tell all waiting threads to try consuming
    //  can also call notify() and notify one thread at a time
}
```

Example: Shared Queue

```
class SharedQueue {
    private Element head, tail;

    public boolean empty() { return head == tail; }

    public synchronized Element remove() {
      try { while (empty()) wait(); } // wait for an element in the queue
      catch (InterruptedException e) { return null; }
      Element p = head; head = head.next;
      if (head == null) tail == null;
      return p;
    }

    public synchronized void insert(Element p)
      if (tail == null) head = p;
      else tail.next = p;
      p.next = null;
      tail = p;
      notify();  // let one waiter know something is in the queue
    }
}
```

POSIX Threads

- Pthreads library for C

pthread_create - create a new thread giving it a "starting" procedure to run along with a single argument.
pthread_self - ask the currently running thread for its thread id.
pthread_join - join with a thread using its thread id (an integer value)

pthread_mutex_init - initialize a mutex structure
pthread_mutex_destroy - destroy a mutex structure
pthread_mutex_lock - lock an initialized mutex, if already locked suspend execution and wait
pthread_mutex_trylock - try to lock a mutex and if unsucessful, do not suspend execution
pthread_mutex_unlock - unlock a mutex that was locked by the current thread

pthread_cond_init - initialize a condition variable structure
pthread_cond_destroy - destroy a condition variable structure
pthread_cond_wait - block the currently running thread on a condition variable indefintely
pthread_cond_timedwait - block the currently running thread on a condition variable for a specific time
pthread_cond_signal - wakeup one thread blocked on a condition variable
pthread_cond_broadcast - wakeup all threads blocked on a condition variable

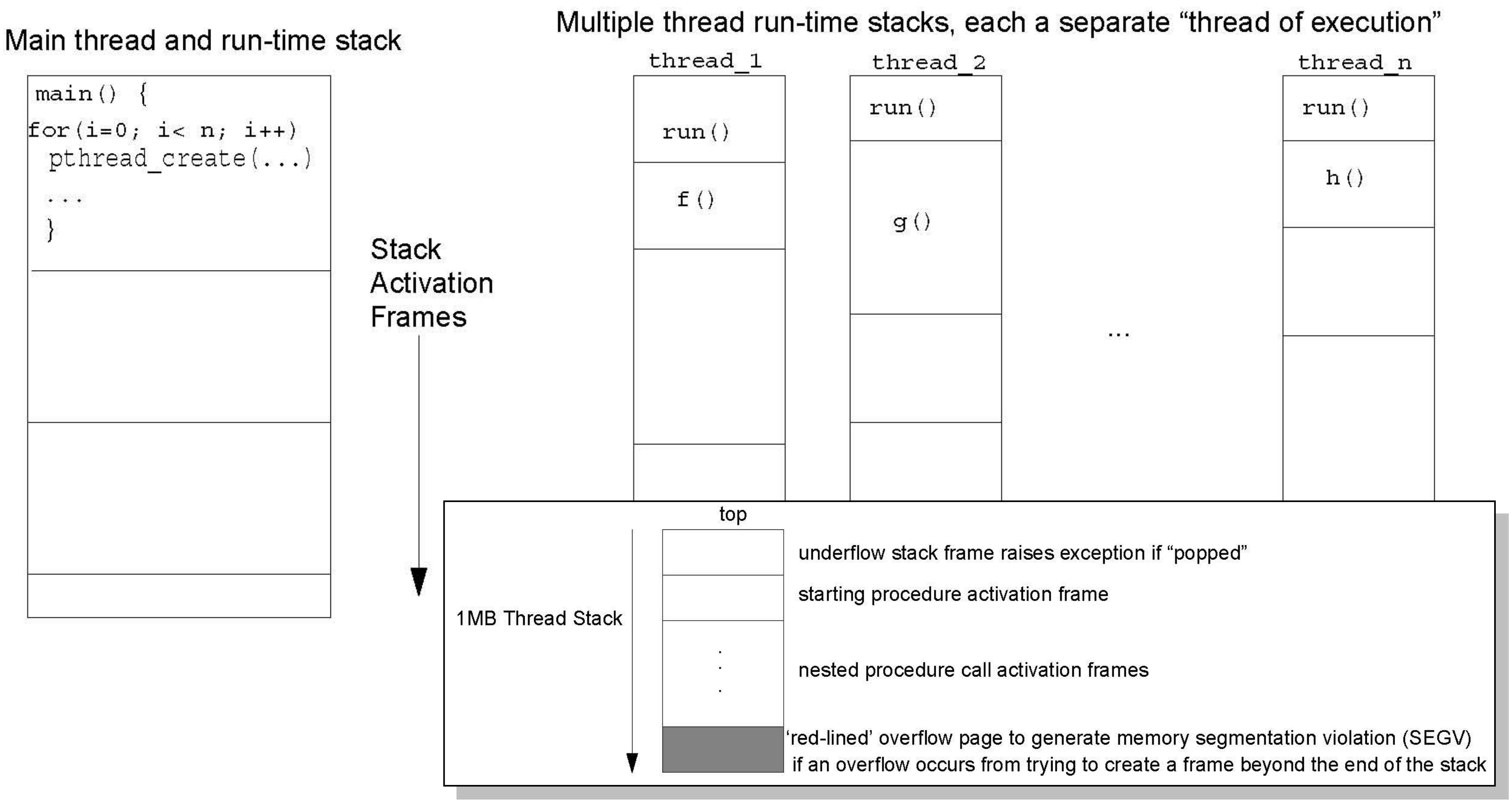
Example of Using POSIX Threads

```
#include <pthread.h>
#include <unistd.h>  /* sleep declaration */
#include <stdio.h>  /* printf declaration */
const int NUM_THREADS = 5;

void* sleeping(void* st)
{
    int sleep_time = (int) st; /* cast void* to an int */
    printf ("thread %d sleeping %d seconds ...\n", pthread_self(), sleep_time);
    sleep(sleep_time);
    printf ("\nthread %d awakening\n", pthread_self());
}

main( int argc, char *argv[] )
{
     pthread_t tid[NUM_THREADS];      /* array of thread IDs */
     int i;

     for ( i = 0; i < NUM_THREADS; i++)
       pthread_create (&tid[i], NULL, sleeping, i+2);

     for ( i = 0; i < NUM_THREADS; i++)
       pthread_join (tid[i], NULL);

     printf ("main() reporting that all %d threads have terminated\n", i);
}  /* main */
```

Create several child threads

Wait for children to finish

Thread Stacks



Thread Safety of Classes

- Fields of an object or class must always be in a valid state, even when used concurrently by multiple threads
  - What's a "valid state"? Serializability …
- Classes are designed so that each method preserves state invariants on entry and exit
  - Example: priority queues represented as sorted lists
  - If invariant fails in the middle of a method call, concurrent execution of another method call will observe an inconsistent state

Making Classes Thread-Safe

- Synchronize critical sections
  - Make fields private, synchronize access to them
- Make objects immutable
  - State cannot be changed after object is created

```
public RGBColor invert() {
    RGBColor retVal = new RGBColor(255 - r, 255 - g, 255 - b);
    return retVal; }
```

  - Examples: Java String and primitive type wrappers Integer, Long, Float, etc.
  - Pure functions are always re-entrant!
- Use a thread-safe wrapper

Java-Style Synchronization in C++

```
class Synchronized {
    pthread_mutex_t m;  // mutex variable
    pthread_cond_t  c;  // condition variable
protected:

    /* use this class to associate the mutex lock/unlock with the scope of a procedure */
    class Scope {
        Synchronized* obj;
    public:
        Scope(Synchronized* s) : obj(s)  { pthread_mutex_lock(&obj->m); }
        ~Scope() { pthread_mutex_unlock(&obj->m); }
    };

public:

    Synchronized() { // initialize the mutex and condvar on construction
       pthread_mutex_init(&m, 0);
       pthread_cond_init(&c, 0);
    }

    ~Synchronized() { // destroy the mutex and condvar on destruction
        pthread_mutex_destroy(&m);
        pthread_cond_destroy(&c);
    }

    // map Java-like wait, notify and notifyAll onto pthread equivalents

    void wait() { pthread_cond_wait(&c, &m); }
    void notify() { pthread_cond_signal(&c); }
    void notifyAll() { pthread_cond_broadcast(&c); }
};
```

Thread-Safe Wrapper

- Define new class which has objects of original class as fields, provides methods to access them

```
public synchronized void setColor(int r, int g, int b) {
    color.setColor(r, g, b);
}
public synchronized int[] getColor() {
    return color.getColor();
}
public synchronized void invert() {
    color.invert();
}
```

Comparison

- Synchronizing critical sections
  - Good way to build thread-safe classes from scratch
  - Only way to allow wait() and notify()
- Using immutable objects
  - Good if objects are small, simple abstract data types
  - Benefits: pass without aliasing, unexpected side effects
- Using wrapper objects
  - Works with existing classes, gives users choice between thread-safe version and original (unsafe) one
    - Example: Java 1.2 collections library – classes not thread-safe, but some have methods to enclose objects in safe wrapper

Why Not Synchronize Everything?

- Performance costs
  - Current Sun JVM – synchronized methods are 4 to 6 times slower than non-synchronized
- Risk of deadlock from too much locking
- Unnecessary blocking and unblocking of threads can reduce concurrency
- Alternative: immutable objects
  - Issue: often short-lived, increase garbage collection

Syllabus

Lecture Series (hours)	Topics
1-4	Introduction and Motivation, Paradigms
5-10	Syntax and Semantics, BNF, Compilation
11-18	Data Types, Constructs, Functions, Activation Records, Names and Bindings
19-28	Concurrency, Functional PLs, Logical PLs, Lambda Calculus, Event driven programming
29-36	Virtual Machines, Managed Languages, JIT, Case study

Atomicity

[Flanaghan]

An easier-to-use and harder-to-implement primitive:

```
void deposit(int x)
{
synchronized(this)
{
  int tmp =
balance;
  tmp += x;
  balance = tmp;
}}
```

semantics:
lock acquire/release

```
void deposit(int x)
{
atomic {
  int tmp =
balance;
  tmp += x;
  balance = tmp;
}}
```

semantics:
(behave as if)
no interleaved execution

No fancy hardware, code restrictions, deadlock, or unfair scheduling (e.g., disabling interrupts)

AtomJava

[Grossman]

- New prototype from the University of Washington
  - Based on source-to-source translation for Java
- Atomicity via locking (object ownership)
  - Poll for contention and rollback
  - No support for parallel readers yet
- Key pieces of the implementation
  - All writes logged when an atomic block is executed
  - If thread is pre-empted in atomic, rollback the thread
  - Duplicate so non-atomic code is not slowed by logging
  - Smooth interaction with GC

Example: RGBColor Class

```
public class RGBColor {
    private int r; private int g; private int b;
    public RGBColor(int r, int g, int b) {
        checkRGBVals(r, g, b);
        this.r = r; this.g = g; this.b = b;
    }

private static void checkRGBVals(int r, int g,
int b) {
        if (r < 0 || r > 255 || g < 0 || g > 255 ||
            b < 0 || b > 255) {
            throw new
IllegalArgumentException();
        }
    }
}
```

What goes wrong with multi-threaded use of this class?

```
public void setColor(int r, int g, int b) {
        checkRGBVals(r, g, b);
        this.r = r; this.g = g; this.b = b;
    }

public int[] getColor() {
        //  returns array of three ints: R,
G, B
        int[] retVal = new int[3];
        retVal[0] = r;
        retVal[1] = g;
        retVal[2] = b;
        return retVal;
    }

public void invert() {
        r = 255 - r; g = 255 - g; b = 255 -
b;
```

Problems with RGBColor Class

- Write/write conflicts
  - If two threads try to write different colors, result may be a “mix” of R,G,B from two different colors
- Read/write conflicts
  - If one thread reads while another writes, the color that is read may not match the color before <u>or</u> after