Introduction to Threads and Concurrency
Why is Concurrency Important?

Why study threads and concurrent programming in an OS class?
What is a thread?
Is multi-threaded programming easy?
If not, why not?
Threads

Processes have the following components:
- an address space
- a collection of operating system state
- a CPU context ... or thread of control

To use multiple CPUs on a multiprocessor system, a process would need several CPU contexts
- Thread fork creates new thread not memory space
- Multiple threads of control could run in the same memory space on a single CPU system too!
Threads

Threads share a process address space with zero or more other threads

Threads have their own CPU context
- PC, SP, register state,
- Stack

A traditional process could be viewed as a memory address space with a single thread
Single Thread in Address Space

User Address Space

stack
- routine1
- var1()
- var2()

text
- main()
- routine1()
- routine2()

data
- arrayA
- arrayB

heap

Stack Pointer
- Prgm. Counter
- Registers

Process ID
- Group ID
- User ID

Files
- Locks
- Sockets
Multiple Threads in Address Space

User Address Space

Thread 2
- stack
  - routine2()
  - var1
  - var2
  - var3

Thread 1
- stack
  - routine1()
  - var1
  - var2

- text
  - main()
  - routine1()
  - routine2()
  - ...

- data
  - arrayA
  - arrayB

- heap

Stack Pointer
- Prgrm. Counter Registers

Process ID
- User ID
- Group ID

Files
- Locks
- Sockets
What Is a Thread?

A thread executes a stream of instructions
- it is an abstraction for control-flow

Practically, it is a processor context and stack
- Allocated a CPU by a scheduler
- Executes in a memory address space
Private Per-Thread State

Things that define the state of a particular flow of control in an executing program

- Stack (local variables)
- Stack pointer
- Registers
- Scheduling properties (i.e., priority)
Shared State Among Threads

Things that relate to an instance of an executing program
- User ID, group ID, process ID
- Open files, sockets, locks
- Address space: Text, Data (off-stack global variables), Heap (dynamic data)
Concurrent Access to Shared State

Important: Changes made to shared state by one thread will be visible to the others!

Reading and writing memory locations requires synchronization!

This is a major topic for later ...
Programming With Threads

Split program into routines to execute in parallel
- True or pseudo (interleaved) parallelism

Alternative strategies for executing multiple routines
Why Use Threads?

Utilize multiple CPU’s concurrently

Low cost communication via shared memory

Overlap computation and blocking on a single CPU
  - Blocking due to I/O
  - Computation and communication

Handle asynchronous events
A word processor with three threads
Processes vs Threads

GET / HTTP/1.0

HTTPD

disk
Processes vs Threads

Why is this not a good web server design?
Processes vs Threads

GET / HTTP/1.0

disk
Processes vs Threads

GET / HTTP/1.0

HTTPD

GET / HTTP/1.0

disk
Processes vs Threads

GET / HTTP/1.0

HTTPD

GET / HTTP/1.0
GET / HTTP/1.0
GET / HTTP/1.0

disk
Pthreads: A Typical Thread API

Pthreads: POSIX standard threads
First thread exists in main(), creates the others

pthread_create (thread,attr,start_routine,arg)
- Returns new thread ID in “thread”
- Executes routine specified by “start_routine” with argument specified by “arg”
- Exits on return from routine or when told explicitly
Pthreads (continued)

pthread_exit (status)
- Terminates the thread and returns “status” to any joining thread

pthread_join (threadid,status)
- Blocks the calling thread until thread specified by “threadid” terminates
- Return status from pthread_exit is passed in “status”
- One way of synchronizing between threads

pthread_yield ()
- Thread gives up the CPU and enters the run queue
Using Create, Join and Exit
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    printf("%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++)
    {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}

Program Output
Creating thread 0
Creating thread 1
0: Hello World!
1: Hello World!
Creating thread 2
Creating thread 3
2: Hello World!
3: Hello World!
Creating thread 4
4: Hello World!

For more examples see: http://www.llnl.gov/computing/tutorials/pthreads
Pros & Cons of Threads

Pros:
- Overlap I/O with computation!
- Cheaper context switches
- Better mapping to multiprocessors

Cons:
- Potential thread interactions
- Complexity of debugging
- Complexity of multi-threaded programming
- Backwards compatibility with existing code
User-level threads

The idea of managing multiple abstract program counters above a single real one can be implemented using privileged or non-privileged code.
- Threads can be implemented in the OS or at user level

User level thread implementations
- Thread scheduler runs as user code (thread library)
- Manages thread contexts in user space
- The underlying OS sees only a traditional process above
Kernel-Level Threads

Thread-switching code is in the kernel
User-Level Threads Package

The thread-switching code is in user space
User-level threads

Advantages
- Cheap context switch costs among threads in the same process!
- Calls are procedure calls not system calls!
- User-programmable scheduling policy

Disadvantages
- How to deal with blocking system calls!
- How to overlap I/O and computation!
Concurrency
Sequential Programming

Sequential programming with processes

- Private memory
  - a program
  - data
  - stack
  - heap

- CPU context
  - program counter
  - stack pointer
  - registers
Sequential Programming Example

```
int i = 0
i = i + 1
print i
```

What output do you expect?
Why?
Concurrent Programming

Concurrent programming with threads

- Shared memory
  - a program
  - data
  - heap

- Private stack for each thread

- Private CPU context for each thread
  - program counter
  - stack pointer
  - registers
Concurrent Threads Example

```java
int i = 0

Thread 1:  i = i + 1
           print i
```

What output do you expect with 1 thread? Why?
Concurrent Threads Example

```java
int i = 0

Thread 1: i = i + 1
print i

Thread 2: i = i + 1
print i
```

What output do you expect with 2 threads? Why?
Race Conditions

How is $i = i + 1$ implemented?

- load $i$ to register
- increment register
- store register value to $i$

Registers are part of each thread’s *private* CPU context.
Race Conditions

Thread 1

load i to regn
inc regn
store regn to i

Thread 2

load i to regn
inc regn
store regn to i
Critical Sections

What is the danger in the previous example?
How can we make it safe?
What price do we pay?
Mutual Exclusion

- A enters critical region
- A leaves critical region
- B attempts to enter critical region
- B enters critical region
- B leaves critical region
- B blocked

Time T1, T2, T3, T4
Mutual Exclusion

How can we implement it?
Locks

Each shared data has a unique lock associated with it. Threads acquire the lock before accessing the data. Threads release the lock after they are finished with the data. The lock can only be held by one thread at a time.
Locks - Implementation

How can we implement a lock?
How do we test to see if its held?
How do we acquire it?
How do we release it?
How do we block/wait if it is already held when we test?
Does this work?

```cpp
bool lock = false

while lock = true; /* wait */
lock = true; /* lock */
critical section
lock = false; /* unlock */
```
Atomicity

Lock and unlock operations must be atomic.
Modern hardware provides a few simple atomic instructions that can be used to build atomic lock and unlock primitives.
Atomic Instructions

Atomic "test and set" (TSL)
Compare and swap (CAS)
Load-linked, store conditional (ll/sc)
Atomic Test and Set

TSL performs the following in a single atomic step:
- set lock and return its previous value

Using TSL in a lock operation
- if the return value is false then you got the lock
- if the return value is true then you did not
- either way, the lock is set
Spin Locks

while TSL (lock); /* spin while return value is true */
critical section
lock = false
Spin Locks

What price do we pay for mutual exclusion?

How well will this work on uniprocessor?
Blocking Locks

How can we avoid wasting CPU cycles?
How can we implement sleep and wakeup?
  - context switch when acquire finds the lock held
  - check and potential wakeup on lock release
  - system calls to acquire and release lock

But how can we make these system calls atomic?
Blocking Locks

Is this better than a spinlock on a uniprocessor?
Is this better than a spinlock on a multiprocessor?
When would you use a spinlock vs a blocking lock on a multiprocessor?
Tricky Issues With Locks

Global variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char buf[n]</td>
<td>char buffer</td>
</tr>
<tr>
<td>int InP = 0</td>
<td>place to add</td>
</tr>
<tr>
<td>int OutP = 0</td>
<td>place to get</td>
</tr>
<tr>
<td>int count</td>
<td></td>
</tr>
</tbody>
</table>

0 thread producer {
  while(1) {
    // Produce char c
    if (count==n) {
      sleep(full)
    }
    buf[InP] = c;
    InP = InP + 1 mod n
    count++
    if (count == 1)
      wakeup(empty)
  }
}

0 thread consumer {
  while(1) {
    // Consume char
    if (count==0) {
      sleep(empty)
    }
    c = buf[OutP]
    OutP = OutP + 1 mod n
    count--;
    if (count == n-1)
      wakeup(full)
    // Consume char
  }
}

n-1
0
1
2
...
Conditional Waiting

Sleeping while holding the lock leads to deadlock
Releasing the lock then sleeping opens up a window for a race
Need to atomically release the lock and sleep
Semaphores

Semaphore S has a value, S.val, and a thread list, S.list.

**Down (S)**

S.val = S.val - 1

If S.val < 0
  add calling thread to S.list;
  sleep;

**Up (S)**

S.val = S.val + 1

If S.val <= 0
  remove a thread T from S.list;
  wakeup (T);
Semaphores

Down and up are assumed to be atomic
How can we implement them?
  - on a uniprocessor?
  - on a multiprocessor?
Semaphores in Producer-Consumer

Global variables

```c
semaphore full_buffs = 0;
semaphore empty_buffs = n;
char buff[n];
int InP, OutP;
```

```c
0 thread producer {
    while(1){
        // Produce char c...
        down(empty_buffs)
        buf[InP] = c
        InP = InP + 1 mod n
        up(full_buffs)
    }
}
```

```c
0 thread consumer {
    while(1){
        down(full_buffs)
        c = buf[OutP]
        OutP = OutP + 1 mod n
        up(empty_buffs)
        // Consume char...
    }
}
```
Monitors and Condition Variables

Correct synchronization is tricky
What synchronization rules can we automatically enforce?
- encapsulation and mutual exclusion
- conditional waiting
Condition Variables

Condition variables (cv) for use within monitors

cv.wait(mon-mutex)
  - thread blocked (queued) until condition holds
  - Must not block while holding mutex!
  - Monitor’s mutex must be released!
  - Monitor mutex need not be specified by programmer if compiler is enforcing mutual exclusion

cv.signal()
  - signals the condition and unblocks (dequeues) a thread
Condition Variables – Semantics

What can I assume about the state of the shared data?

- when I wake up from a wait?
- when I issue a signal?
Hoare Semantics

Signaling thread hands monitor mutex directly to signaled thread

Signaled thread can assume condition tested by signaling thread holds
Mesa Semantics

Signaled thread eventually wakes up, but signaling thread and other threads may have run in the meantime

Signaled thread can not assume condition tested by signaling thread holds
- signals are a hint

Broadcast signal makes sense with MESA semantics, but not Hoare semantics
Memory Invariance

A thread executing a sequential program can assume that memory only changes as a result of the program statements

- can reason about correctness based on pre and post conditions and program logic

A thread executing a concurrent program must take into account the points at which memory invariants may be broken

- what points are those?
Subtle Race Conditions

Why does wait on a condition variable need to “atomically” unlock the mutex and block the thread?

Why does the thread need to re-lock the mutex when it wakes up from wait?

Can it assume that the condition it waited on now holds?
Deadlock

Thread A locks mutex 1
Thread B locks mutex 2
Thread A blocks trying to lock mutex 2
Thread B blocks trying to lock mutex 1

Can also occur with condition variables
Nested monitor problem (p. 20)
Deadlock (nested monitor problem)

Procedure Get();
BEGIN
    LOCK a DO
        LOCK b DO
            WHILE NOT ready DO wait(b,c) END;
        END;
    END;
END Get;

Procedure Give();
BEGIN
    LOCK a DO
        LOCK b DO
            ready := TRUE; signal(c);
        END;
    END;
END Give;
Deadlock in layered systems

High layer: Lock M; Call lower layer; Release M;

Low layer: Lock M; Do work; Release M; return;

Result – thread deadlocks with itself!
Layer boundaries are supposed to be opaque
Deadlock

Why is it better to have a deadlock than a race?
Deadlock

Why is it better to have a deadlock than a race?

Deadlock can be prevented by imposing a global order on resources managed by mutexes and condition variables, i.e., all threads acquire mutexes in the same order.

Mutex ordering can be based on layering. Allowing upcalls breaks this defense.
Priority Inversion

Starvation of high priority threads (occurs in priority scheduling)

- Low priority thread C locks M
- Medium priority thread B pre-empts C
- High priority thread A preempts B then blocks on M
- B resumes and enters long computation

Result:

- C never runs so can’t unlock M, therefore A never runs

Solution? – priority inheritance
Dangers of Blocking in a Critical Section

Blocking while holding M prevents progress of other threads that need M

Blocking on another mutex may lead to deadlock

Why not release the mutex before blocking?
  Must restore the mutex invariant
  Must reacquire the mutex on return!

Things may have changed while you were gone ...
Reader/Writer Locking

Writers exclude readers and writers
Readers exclude writers but not readers

Example, page 15

Good use of broadcast in ReleaseExclusive()
Results in “spurious wake-ups”
... and “spurious lock conflicts”

How could you use signal instead?
Move signal/broadcast call after release of mutex?

Advantages? Disadvantages?

Can we avoid writer starvation?
Useful Programming Conventions

All access to shared data must be protected by a mutex
- All shared variables have a lock
- The lock is held by the thread that accesses the variable

How can this be checked?
- Statically?
- Dynamically?
Automated Checking of Conventions

Eraser

A dynamic checker that uses binary re-writing techniques
Gathers an “execution history” of reads, writes and lock acquisitions
Evaluates consistency with rules

Is it enough to simply check that some lock is held whenever a global variable is accessed?
Automated Checking of Conventions

Eraser doesn’t know ahead of time which locks protect which variables

It infers which locks protect which variables using a lock-set algorithm

Assume all locks are candidates for a variable
( C(v) is full)

For each access take intersection of C(v) and locks held by thread and make these the candidate set C(v)

If C(v) becomes empty, issue warning
Improving the Locking Discipline

The standard approach produces many false positives that arise due to special cases:

Initialization
   No need to lock if no thread has a reference yet
Read sharing
   No need to lock if all threads are readers
Reader/writer locking
   Distinguish concurrent readers from concurrent readers and writers
Improved Algorithm

virgin

exclusive

shared

First thread

rd, wr

wr, new thread

rd

rd, new thread

wr, new thread

wr

rd

wr

modified (race?)
Questions

Why are threads “lightweight”?  
Why associate thread lifetime with a procedure?  
Why block instead of spin waiting for a mutex?  
If a mutex is a resource scheduling mechanism
   What is the resource being scheduled?  
   What is the scheduling policy and where is it defined?  
Why do “alerts” result in complicated programs?  
What is coarse-grain locking?  
   What effect does it have on program complexity?  
   What effect does it have on performance?
Is multi-threaded programming hard?

If so, why?