CS533 Concepts of Operating Systems

Jonathan Walpole
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
Multiple Threads in Address Space

User Address Space

Thread 2
stack

routine2() var1
    var2
    var3

Stack Pointer
Prgrm. Counter
Registers

Thread 1
stack

routine1() var1
    var2

Stack Pointer
Prgrm. Counter
Registers

text

main()
routine1()
routine2()
...

data

arrayA
arrayB

heap

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)

The memory address space is shared with other threads in that process
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
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
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

HTTPD

HTTPD

disk
Processes vs Threads

GET / HTTP/1.0

HTTPD

GET / HTTP/1.0

disk
Processes vs Threads
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
An Example Pthreads Program

```c
#include <stdio.h>
#include <string.h>
#define MAX_THREADS 5

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

int main (int argc, char *argv[])
{
    pthread_t threads[MAX_THREADS];
    int rc, t;
    for(t=0; t<MAX_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
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.
Implementing Threads

When a thread is created, what needs to happen?
When a thread exits, what needs to happen?
What will cause a thread switch to occur?
What will happen when a thread switch occurs?
Is the kernel really needed?
  - can multiple CPUs access the same memory?
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

```plaintext
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   Thread 2:   i = i + 1
print i      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 dangerous in the previous example?
How should we reason about this kind of code?
What property did we have in sequential programming, which we lost in concurrent programming?
Why was that property important?
Memory Invariance

Sequential programs have the property that memory values do not change unless the control flow changes them.
Hence, we can reason about the effects of a control flow.
How can we regain this memory invariance property in concurrent programs?
Mutual Exclusion

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

Time progression:
- **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?

```c
bool lock = false
while lock = true; /* wait */
lock = true; /* lock */
critical section
lock = false; /* unlock */
```
What is the Problem?

The memory invariance property does not hold for the code that implements the lock acquisition!

How can we proceed with out it?

Lock and unlock operations must be made atomic, i.e., indivisible!

Modern hardware provides a few simple atomic instructions that can be used to build atomic lock and unlock primitives

Programming with these primitives requires a different way of thinking!
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:

char buf[n]
int InP = 0 // place to add
int OutP = 0 // place to get
int count

0 thread producer {
1   while(1) {
2     // Produce char c
3     if (count==n) {
4         sleep(full)
5     }
6     buf[InP] = c;
7     InP = InP + 1 mod n
8     count++
9     if (count == 1)
10        wakeup(empty)
11   }
12 }

0 thread consumer {
1   while(1) {
2     if(count==0) {
3         sleep(empty)
4     }
5     c = buf[OutP]
6     OutP = OutP + 1 mod n
7     count--;
8     if (count == n-1)
9        wakeup(full)
10     // Consume char
11   }
12 }

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

0 thread producer {
1    while(1){
2        // Produce char c...
3        down(empty_buffs)
4        buf[InP] = c
5        InP = InP + 1 mod n
6        up(full_buffs)
7    }
8 }

0 thread consumer {
1    while(1){
2        down(full_buffs)
3        c = buf[OutP]
4        OutP = OutP + 1 mod n
5        up(empty_buffs)
6        // Consume char...
7    }
8 }

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

Let’s revisit the memory invariance property in the context of monitors.

Can we assume memory invariance when reasoning about the effects of threads?

What can I assume about the state of the shared data?
- while I am executing in the monitor?
- when I wake up from a wait?
- after I have issued 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 cannot 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?
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?
What is memory invariance and why is it important?
How can we reason about concurrent code?
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?