Overview of Threads and Concurrency
Questions

- 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:
  - a CPU context ... or *thread* of control
  - an addressing context (address space)
  - a collection of operating system state

- On multiprocessor systems, with several CPUs, it would make sense for a process to have several CPU contexts (threads of control)

- Multiple threads of control could run in the same address space on a single CPU system too!
  - “thread of control” and “address space” are orthogonal concepts
Threads

- Threads share an address space with zero or more other threads
  - could be the kernel’s address space or that of a user level process

- Threads have their own
  - PC, SP, register state etc (CPU state)
  - Stack (memory)

- Why do these need to be private to each thread?
  - what other OS state should be private to threads?

- A traditional process can be viewed as an address space with a single thread
Single thread state within a process
Multiple threads in an address space
Shared state among related threads

- Open files, sockets, locks
- User ID, group ID, process/task ID
- Address space
  - Text
  - Data (off-stack global variables)
  - Heap (dynamic data)
- Changes made to shared state by one thread will be visible to the others!
  - Reading & writing shared memory requires synchronization!
Why program using 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
Why use threads? - example

- A WWW process

GET / HTTP/1.0

HTTPD
Why use threads? - example

- A WWW process

Why is this not a good web server design?
Why use threads? - example

- A WWW process

```
GET / HTTP/1.0
```

![Diagram showing HTTPD process and disk]
Why use threads? - example

- A WWW process

```
GET / HTTP/1.0
```

```
HTTPD
```

```
GET / HTTP/1.0
```

`disk`
Why use threads? - example

- A WWW process

GET / HTTP/1.0

HTTPD

GET / HTTP/1.0

GET / HTTP/1.0

GET / HTTP/1.0

GET / HTTP/1.0

disk
What does a typical thread API look like?

- POSIX standard threads (Pthreads)
- First thread exists in main(), typically 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
Thread API (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 primitives

Master Thread

Worker Thread

Worker Thread

pthread_create() → pthread_join() → pthread_exit()
An example Pthreads program

#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 shared memory multiprocessors

- **Cons**
  - Potential thread interactions due to concurrent access to memory
  - Complexity of debugging
  - Complexity of multi-threaded programming
  - Backwards compatibility with existing code
Concurrent programming

Assumptions:
- Two or more threads
- Each executes in (pseudo) parallel and can’t predict exact running speeds
- The threads can interact via access to shared variables

Example:
- One thread writes a variable
- The other thread reads from the same variable

Problem:
- The outcome depends on the order of these READs and WRITES!
Race conditions

- What is a race condition?
Race conditions

- What is a race condition?
  - two or more threads have an inconsistent view of a shared memory region (i.e., a variable)

- Why do race conditions occur?
Race conditions

- What is a race condition?
  - two or more threads have an inconsistent view of a shared memory region (i.e., a variable)

- Why do race conditions occur?
  - values of memory locations are replicated in registers during execution
  - context switches occur at arbitrary times during execution (or program runs on a multiprocessor)
  - threads can see “stale” memory values in registers
Race Conditions

- Race condition: whenever the output depends on the precise execution order of the threads!
- What solutions can we apply?
Race Conditions

- Race condition: whenever the output depends on the precise execution order of the threads!

- What solutions can we apply?
  - prevent context switches by preventing interrupts?
  - make threads coordinate with each other to ensure mutual exclusion in accessing critical sections of code
Synchronization by mutual exclusion

- Divide thread code into critical sections
  - Sections where shared data is accessed (read/written)
- Only allow one thread at a time in a critical section
Critical sections with mutual exclusion

Process A
A enters critical region
A leaves critical region

Process B
B attempts to enter critical region
B enters critical region
B leaves critical region

B blocked

Time

CS533 - Concepts of Operating Systems
How can we ensure mutual exclusion?

- What about using a binary “lock” variable in memory and having threads check it and set it before entry to critical regions?
Implementing locks

- A binary “lock” variable in memory does not work!

- Many computers have some limited hardware support for atomically testing and setting locks
  - “Atomic” Test and Set Lock instruction
  - “Atomic” compare and swap instruction

- These atomic instructions can be used to implement mutual exclusion (mutex) locks
Test-and-set-lock instruction (TSL, tset)

- A lock is a single word variable with two values
  - 0 = FALSE = not locked
  - 1 = TRUE = locked

- The test-and-set instruction does the following **atomically**:
  - Get the (old) value of lock
  - Set the new value of lock to TRUE
  - Return the old value

**If** the returned value was FALSE...
*Then you got the lock!!!*

**If** the returned value was TRUE...
*Then someone else has the lock*
* (so try again later)
Mutex locks

- An abstract data type built from the underlying atomic instructions provided by the CPU
- Used for mutual exclusion

- **Lock (mutex)**
  - Acquire the lock, if it is free
  - If the lock is not free, then wait until it can be acquired
  - Various different ways to “wait”

- **Unlock (mutex)**
  - Release the lock
  - If there are waiting threads, then wake up one of them
Building *spinning* mutex locks using TSL

**Mutex_lock:**

```
TSL REGISTER, MUTEX  \| copy mutex to register and set mutex to 1
CMP REGISTER, #0     \| was mutex zero?
JZE ok               \| if it was zero, mutex is unlocked, so return
JMP mutex_lock       \| try again later
Ok: RET              \| return to caller; enter critical section
```

**Mutex_unlock:**

```
MOVE MUTEX, #0       \| store a 0 in mutex
RET                  \| return to caller
```
Building *yielding* mutex locks using TSL

### Mutex_lock:

- **TSL REGISTER,MUTEX** | copy mutex to register and set mutex to 1
- **CMP REGISTER,#0**  | was mutex zero?
- **JZE ok**           | if it was zero, mutex is unlocked, so return
- **CALL thread_yield** | mutex is busy, so schedule another thread
- **JMP mutex_lock**   | try again later

**Ok:** **RET** | return to caller; enter critical section

### Mutex_unlock:

- **MOVE MUTEX,#0**  | store a 0 in mutex
- **RET**            | return to caller
To yield or not to yield?

- Spin-locks do *busy waiting*
  - wastes CPU cycles on uni-processors
  - Why?

- Yielding locks give up the CPU
  - may waste CPU cycles on multi-processors
  - Why?

- Yielding is not the same as blocking!
An Example using a Mutex

**Shared data:**

```
Mutex myLock;
```

1 repeat
2  `Lock(myLock);`
3   `critical section`
4  `Unlock(myLock);`
5   `remainder section`
6 until FALSE

1 repeat
2  `Lock(myLock);`
3   `critical section`
4  `Unlock(myLock);`
5   `remainder section`
6 until FALSE
Enforcing mutual exclusion

- **Assumptions:**
  - Every thread sets the lock before accessing shared data!
  - Every thread releases the lock after it is done!

- **Only works if you follow these programming conventions all the time!**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>Lock</td>
<td></td>
</tr>
<tr>
<td>$A = 2$</td>
<td>$A = A + 1$</td>
<td>$A = A \times B$</td>
</tr>
<tr>
<td>Unlock</td>
<td>Unlock</td>
<td></td>
</tr>
</tbody>
</table>
Using Pthread mutex variables

- **Pthread_mutex_lock (mutex)**
  - Acquire the lock or block until it is acquired

- **Pthread_mutex_trylock (mutex)**
  - Acquire the lock or return with “busy” error code

- **Pthread_mutex_unlock (mutex)**
  - Free the lock
Invariant of a mutex

- The mutex “invariant” is the condition that must be restored before:
  - The mutex is released

- Example
  - Invariant $A=B$
    - always holds outside the critical section
  - Critical section updates $A$ and $B$
What does “thread-safe” mean?
What does “thread-safe” mean?

- A piece of code (library) is “thread-safe” if it defines critical sections and uses synchronization to control access to them.
- All entry points must be re-entrant.
- Results not returned in shared global variables nor global statically allocated storage.
- All calls should be synchronous.
Reentrant code

- A function/method is said to be **reentrant** if...

  A function that has been invoked may be invoked again before the first invocation has returned, and will still work correctly

- Recursive routines are reentrant

- In the context of concurrent programming...

  A reentrant function can be executed simultaneously by more than one thread, with no ill effects
Reentrant Code

- Consider this function...

```plaintext
var count: int = 0
function GetUnique () returns int
    count = count + 1
    return count
endFunction
```

- What happens if it is executed by different threads concurrently?
Reentrant Code

- Consider this function...

```
var count: int = 0
function GetUnique () returns int
    count = count + 1
    return count
endFunction
```

- What happens if it is executed by different threads concurrently?
  - The results may be incorrect!
  - This routine is not reentrant!
When is code reentrant?

- Some variables are
  - “local” -- to the function/method/routine
  - “global” -- sometimes called “static”

- Access to local variables?
  - A new stack frame is created for each invocation
  - Each thread has its own stack

- What about access to global variables?
  - Must use synchronization!
Making this function reentrant

```plaintext
var count: int = 0
myLock: Mutex

function GetUnique () returns int
    var i: int
    myLock.Lock()
    count = count + 1
    i = count
    myLock.Unlock()
    return i
endFunction
```
Question

- What is the difference between mutual exclusion and condition synchronization?
Question

- What is the difference between mutual exclusion and condition synchronization?

- **Mutual exclusion**
  - only one at a time in a critical section

- **Condition synchronization**
  - wait until some condition holds before proceeding
  - signal when condition holds so others may proceed
Condition variables

- Mutex locks allow threads to synchronize before accessing the data
- Condition variables allow synchronization based on the value of the data
  - Used in conjunction with a mutex lock
  - Allows a thread in a critical section to wait for a condition to become true or signal that a condition is true

Acquire mutex lock (enter critical section)
...
Block until condition becomes true (frees mutex lock)
...
Free mutex lock (leave critical section)
Pthread condition variables

- `pthread_cond_wait (condition, mutex)`
  - Releases “mutex” and blocks until “condition” is signaled

- `pthread_cond_signal (condition)`
  - Signals “condition” which wakes up a thread blocked on “condition”

- `pthread_cond_broadcast (condition)`
  - Signals “condition” and wakes up all threads blocked on “condition”
Semantics of condition variables

- How many blocked threads should be woken on a signal?
- Which blocked thread should be woken on a signal?
- In what order should newly awoken threads acquire the mutex?
- Should the signaler immediately free the mutex?
  - If so, what if it has more work to do?
  - If not, how can the signaled process continue?
- What if signal is called before the first wait?
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

- Occurs in priority scheduling
- Starvation of high priority threads

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

modified (race?)

First thread

rd, wr

rd

wr, new thread

wr, new thread

rd, new thread

wr

wr
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?
Questions

- What is “lock contention”?  
  - Why is it worse on multiprocessors than uniprocessors?  
  - What is the solution? ... and its cost?

- What else might cause performance to degrade when you use multiple threads?
Why is multi-threaded programming hard?

- Many possible interleavings at the instruction level that make it hard to reason about correctness.