CS 333
Introduction to Operating Systems

Class 4 - Concurrent Programming and Synchronization Primitives

Jonathan Walpole
Computer Science
Portland State University
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
An example Pthreads program

#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    printf("\n%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 concurrency
  - Complexity of debugging
  - Complexity of multi-threaded programming
  - Backwards compatibility with existing code
Concurrency

Assumptions:
- Two or more threads
- Each executes in (pseudo) parallel
- We 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 - non-determinism:
  - The relative order of one thread’s reads and the other thread’s writes determines the end result!
Race conditions

- What is a race condition?
- Why do race conditions occur?
Race conditions

- A simple multithreaded program with a race:

```csharp
i++;
```
Race conditions

- A simple multithreaded program with a race:

... load i to register; increment register; store register to i; ...

...
Race conditions

- Why did this race condition occur?
  - two or more threads have an inconsistent view of a shared memory region (i.e., a variable)
  - values of memory locations replicated in registers during execution
  - context switches at arbitrary times during execution
  - threads can see “stale” memory values in registers
Race Conditions

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

- 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
Mutual exclusion conditions

- No two processes simultaneously in critical section
- No assumptions made about speeds or numbers of CPUs
- No process running outside its critical section may block another process
- No process must wait forever to enter its critical section
Using mutual exclusion for critical sections
How can we enforce mutual exclusion?

- What about using *locks*?
  - Locks solve the problem of exclusive access to shared data.
    - Acquiring a lock prevents concurrent access
    - Expresses intention to enter critical section

- Assumption:
  - Each shared data item has an associated lock
  - All threads set the lock before accessing the shared data
  - Every thread releases the lock after it is done
Acquiring and releasing locks

Thread A
Thread B
Thread C
Thread D

Free
Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Lock

Free

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Lock

Set

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Set

Lock

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Set Lock
Acquiring and releasing locks
Acquiring and releasing locks

Thread A

Thread B

Lock

Thread C

Set

Lock

Thread D
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Lock

Lock

Lock

Set

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Set

Lock

Lock

Lock

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Set

Lock

Unlock

Lock

Lock

Lock
Acquiring and releasing locks

Thread A

Lock

Unlock

Thread B

Lock

Thread C

Lock

Thread D

Lock

Set

Lock
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D

Free

Lock
Acquiring and releasing locks

Thread A

Thread B

Lock

Free

Lock

Thread C

Lock

Thread D

Lock
Acquiring and releasing locks
Acquiring and releasing locks

Thread A

Thread B

Thread C

Thread D
Acquiring and releasing locks
Mutual exclusion (mutex) locks

- An abstract data type
- Used for synchronization
- The mutex is either:
  - Locked ("the lock is held")
  - Unlocked ("the lock is free")
Mutex lock operations

- **Lock** *(mutex)*
  - Acquire the lock if it is free ... and continue
  - Otherwise wait until it can be acquired

- **Unlock** *(mutex)*
  - Release the lock
  - If there are waiting threads wake up one of them
How to use a mutex?

Shared data:

Mutex myLock;

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

```plaintext
1 repeat
2   Lock(myLock);
3   critical section
4   Unlock(myLock);
5   remainder section
6 until FALSE
```
But how can we implement a mutex?

- What if the lock is a binary variable
- How would we implement the lock and unlock procedures?
But how can we implement a mutex?

- **Lock** and **Unlock** operations must be *atomic*!

- Many computers have *some limited* hardware support for setting locks
  - Atomic Test and Set Lock instruction
  - Atomic compare and swap operation

- These can be used to implement 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

- Test-and-set does the following *atomically*:
  - Get the (old) value
  - Set the 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)
Test and set lock

P1

FALSE

Lock
Test and set lock

FALSE = Lock Available!!

P1

FALSE

Lock
Test and set lock

P1

set

TRUE

Lock
Test and set lock
Test and set lock
Test and set lock
Test and set lock

P1

P2

P3

P4

FALSE

TRUE

Lock
Test and set lock

[Diagram with nodes P1, P2, P3, P4 and arrows indicating TRUE, FALSE, and Lock connections]
Test and set lock

P1

P2

P3

P4

TRUE

FALSE

TRUE

Lock
Test and set lock
Using TSL directly for critical sections

1 repeat
2 while(TSL(lock))
3 no-op;
4 critical section
5 Lock = FALSE;
6 remainder section
7 until FALSE

Guarantees that only one thread at a time will enter its critical section
Implementing a mutex with TSL

1 repeat
2  while(TSL(mylock))
3     no-op;
4 critical section
5 mylock = FALSE;
6 remainder section
7 until FALSE

- Note that processes are busy while waiting
  - this kind of mutex is called a spin lock
Busy waiting

- Also called polling or spinning
  - The thread consumes CPU cycles to evaluate when the lock becomes free!

- Problem on a single CPU system...
  - A busy-waiting thread can prevent the lock holder from running & completing its critical section & releasing the lock!
    - time spent spinning is wasted on a single CPU system
  - Why not block instead of busy wait?
Blocking synchronization primitives

- **Sleep**
  - Put a thread to sleep
  - Thread becomes BLOCKED

- **Wakeup**
  - Move a BLOCKED thread back onto “Ready List”
  - Thread becomes READY (or RUNNING)

- **Yield**
  - Put calling thread on ready list and schedule next thread
  - Does not BLOCK the calling thread!
    - *Just gives up the current time-slice*
But how can these be implemented?

- **In User Programs:**
  - System calls to the kernel

- **In Kernel:**
  - Calls to the thread scheduler routines
Concurrency control in user programs

- User threads call sleep and wakeup system calls
- Scheduler routines in the kernel implement sleep and wakeup
  - they manipulate the “ready list”
  - but the ready list is shared data
  - the code that manipulates it is a critical section
    - What if a timer interrupt occurs during a sleep or wakeup call?
- Problem:
  - How can scheduler routines be programmed to execute correctly in the face of concurrency?
Concurrency in the kernel

Solution 1: Disable interrupts during critical sections
- Ensures that interrupt handling code will not run
- ... but what if there are multiple CPUs?

Solution 2: Use mutex locks based on TSL for critical sections
- Ensures mutual exclusion for all code that follows that convention
- ... but what if your hardware doesn’t have TSL?
Disabling interrupts

- Disabling interrupts in the OS vs disabling interrupts in user processes
  - why not allow user processes to disable interrupts?
  - is it ok to disable interrupts in the OS?
  - what precautions should you take?
Disabling interrupts in the kernel

Scenario 1:

A thread is running; wants to access shared data

- Disable interrupts
- Access shared data ("critical section")
- Enable interrupts
Disabling interrupts in the kernel

Problem:

Interrupts are already disabled and a thread wants to access the critical section...using the above sequence...

- I.e. One critical section gets nested inside another
Disabling interrupts in the kernel

Problem: Interrupts are already disabled.
  • Thread wants to access critical section using the previous sequence...

  Save previous interrupt status (enabled/disabled)
  Disable interrupts
  Access shared data ("critical section")
  Restore interrupt status to what it was before
Disabling interrupts is not enough on MPs...

- Disabling interrupts during critical sections
  - Ensures that interrupt handling code will not run
  - But what if there are multiple CPUs?
  - A thread on a different CPU might make a system call which invokes code that manipulates the ready queue

- Using a mutex lock (based on TSL) for critical sections
  - Ensures mutual exclusion for all code that follows that convention
Some tricky issues ...

- The interrupt handling code that saves interrupted state is a critical section
  - It could be executed concurrently if multiple almost simultaneous interrupts happen
  - Interrupts must be disabled during this (short) time period to ensure critical state is not lost

- What if this interrupt handling code attempts to lock a mutex that is held?
  - What happens if we sleep with interrupts disabled?
  - What happens if we busy wait (spin) with interrupts disabled?
Implementing mutex locks without TSL

- If your CPU did not have TSL, how would you implement blocking mutex lock and unlock calls using interrupt disabling?
  - … this is your next Blitz project!
Quiz

- What is a race condition?
- How can we protect against race conditions?
- Can locks be implemented simply by reading and writing to a binary variable in memory?
- How can a kernel make synchronization-related system calls atomic on a uniprocessor?
  - Why wouldn’t this work on a multiprocessor?
- Why is it better to block rather than spin on a uniprocessor?
- Why is it sometimes better to spin rather than block on a multiprocessor?