CS399 New Beginnings

Jonathan Walpole
Concurrent Programming & Synchronization Primitives
Concurrency

Two or more threads
Parallel or pseudo parallel execution
Can’t predict the relative execution speeds
The threads access shared variables
Concurrency

Example: One thread writes a variable that another thread reads

Problem – non-determinism:

*The relative order of one thread’s reads and the other thread’s writes may determine the outcome!*
Race Conditions

What is a race condition?

Why do race conditions occur?
Race Conditions

A simple multithreaded program with a race:

```c
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 are 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 result 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
Mutual Exclusion Conditions

No two processes simultaneously in critical section

No assumptions about speeds or numbers of CPUs

No process running outside its critical section may block another process

No process waits forever to enter its critical section
Mutual Exclusion Example

- Process A enters the critical region at $T_1$ and leaves at $T_3$.
- Process B attempts to enter the critical region at $T_2$ and is blocked.
- Process A leaves the critical region at $T_3$.
- Process B enters the critical region at $T_3$.

Time: $T_1$, $T_2$, $T_3$, $T_4$
Enforcing Mutual Exclusion

What about using *locks*?

Locks can ensure 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

Free

Lock

Lock
Acquiring and Releasing Locks

Thread A

Set

Lock

Thread B

Thread C

Thread D
Acquiring and Releasing Locks

Thread A

Set

Lock

Thread B

Thread C

Thread D
Acquiring and Releasing Locks
Acquiring and Releasing Locks
Acquiring and Releasing Locks
Acquiring and Releasing Locks

Thread A

Thread B

Thread C

Thread D

Set

Lock

Lock

Lock

Lock
Acquiring and Releasing Locks
Acquiring and Releasing Locks

Thread A

Unlock

Thread B

Lock

Thread C

Lock

Thread D

Lock

Set

Lock
Acquiring and Releasing Locks
Acquiring and Releasing Locks
Acquiring and Releasing Locks

Thread A

Thread B

Thread C

Thread D

Free

Lock

Lock

Lock

Lock
Acquiring and Releasing Locks
Acquiring and Releasing Locks

Thread A

Thread B

Thread C

Thread D

Set

Lock

Lock

Lock

Lock
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 one up
Using a Mutex Lock

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
Implementing a Mutex Lock

If the lock was just a binary variable, how would we implement the lock and unlock procedures?
Implementing a Mutex Lock

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/exchange operation
- These can be used to implement mutex locks
Test-and-Set-Lock (TSL) Instruction

- A lock is a single word variable with two values:
  0 = FALSE = not locked ; 1 = TRUE = locked
- Test-and-set-lock does the following \textit{atomically}:
  - Get the (old) value
  - Set the lock to TRUE
  - Return the old value

\begin{itemize}
  \item \textbf{If} the returned value was FALSE…
    Then you got the lock!!!
  \item \textbf{If} the returned value was TRUE…
    Then someone else has the lock
    (so try again later)
\end{itemize}
Test and Set Lock

P1

FALSE

Lock
Test and Set Lock

FALSE = Lock Available!!
Test and Set Lock

P1

set

TRUE

Lock
Test and Set Lock

P1

P2
TRUE
TRUE

P3
TRUE
TRUE

P4
TRUE
TRUE

TRUE

Lock
Test and Set Lock

P1

P2

P3

P4

TRUE

TRUE

TRUE

TRUE

TRUE

TRUE

TRUE

TRUE

TRUE

TRUE

Lock
Test and Set Lock

P1

P2

TRUE

P3

TRUE

P4

TRUE

TRUE

TRUE

TRUE

Lock
Test and Set Lock

P1

P2

FALSE

Lock

P3

TRUE

P4

FALSE
Test and Set Lock

P1

P2

P3

P4

TRUE

FALSE

TRUE

Lock
Test and Set Lock
Test and Set Lock

P1

P2

TRUE

P3

TRUE

P4

TRUE

TRUE

TRUE

TRUE
Using TSL Directly

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 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
How Can We Implement Them?

In User Programs:
- System calls to the kernel

In Kernel:
- Calls to the thread scheduler routines
Synchronization in User Programs

User threads call sleep and wakeup system calls
Sleep and wakeup are system calls (in the kernel)
- 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...

ie. 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
Doesn’t Work on Multiprocessors

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
  - Disabling interrupts on one CPU didn’t prevent this!

Solution: use a mutex lock (based on TSL)
- Ensures mutual exclusion for all code that uses it
Some Other 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

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