SYNCHRONIZATION

Slides adapted from Tom Anderson's "Operating Systems: Principles and Practice"

How can threads communicate?

1. Message passing
   - Communication is explicit
   - + Easier to reason about
   - – Copy overhead

2. Shared memory
   - Communication is implicit on data access
   - + No copy overhead
   - – Correctness often requires explicit thread synchronization

Multi versus single threaded programs

- Execution may depend on the possible interleavings of the thread's access to shared data

- Execution may be non-deterministic

- More sensible to hardware and compiler instruction reordering optimizations

Synchronization Motivation

Thread 1

```
p = someFn();
isInitialized = true;
```

Thread 2

```
while (! isInitialized )
    ;
q = aFn(p);

if q != aFn(someFn())
   panic
```

Definitions

- **Race condition:**
  - Output of a concurrent program depends on the order of operations between threads
- **Data race:**
  - Two threads are accessing shared data and at least one of them is performing a write operation
- **Critical section:**
  - Piece of code that only one thread can execute at once
- **Mutual exclusion:**
  - Only one thread does a particular thing at a time
- **Lock:**
  - Prevent someone from doing something
    - Lock before entering critical section, before accessing shared data
    - unlock when leaving, after done accessing shared data
    - wait if locked (all synch involves waiting!)

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Too Much Milk, Try #1

- Correctness property
  - Someone buys if needed (**liveness**)
  - At most one person buys (**safety**)

- Try #1: leave a note

```
if !note
  if !milk {
        leave note
        buy milk
        remove note
   }
```

Safety sensible to context switch

Too Much Milk, Try #2

Thread A	Thread B

Thread A

```
leave note A
if (!note B) {
  if (!milk)
    buy milk
  }
remove note A
```

Thread B

```
leave note B
if (!noteA) {
  if (!milk)
    buy milk
  }
remove note B
```

Liveness sensible to context switch

Too Much Milk, Try #3

Thread A

```
leave note A
while (note B) // X
   do nothing;
if (!milk)
   buy milk;
remove note A
```

Thread B

```
leave note B
if (!noteA) {   // Y
  if (!milk)
    buy milk
  }
remove note B
```

Can guarantee at X and Y that either:

1. Safe for me to buy
2. Other will buy, ok to quit

Lessons

- Solution is complicated
  - “obvious” code often has bugs
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult
  - Memory barriers are needed
- Generalizing to many threads/processors
  - Peterson’s algorithm: even more complex

Locks

- lock_acquire
  - wait until lock is free, then take it

- lock_release
  - release lock, waking up anyone waiting for it

- At most one lock holder at a time (safety)
- If no one holding, acquire gets lock (progress)
- If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Too Much Milk, #4

- Locks allow concurrent code to be much simpler:

```
lock_acquire()
if (!milk) buy milk
lock_release()
```

Rules for Using Locks

- Lock is initially free

- Always acquire before accessing shared data structure
  - Beginning of procedure!

- Always release after finishing with shared data
  - End of procedure!
  - DO NOT throw lock for someone else to release

- Never access shared data without lock
  - Danger!

Condition Variables

- Called only when holding a lock

- **Wait**: atomically release lock and relinquish processor until signaled
- **Signal**: wake up a waiter, if any
- **Broadcast**: wake up all waiters, if any

Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up
- Wait atomically releases lock

Condition Variables

- When a thread is woken up from wait, it may not run immediately
  - Mesa semantics
    - Signal puts waiter on ready list
    - Signaler keeps lock and processor
  - Hoare semantics
    - Signal gives processor and lock to waiter
    - When waiter finishes, processor/lock given back to signaler
    - Nested signals possible!

- Under Mesa semantics wait MUST be in a loop

```
while (needToWait())
    condition.Wait(lock);
```

- Mesa semantics simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Java Manual

- When waiting upon a Condition, a “spurious wakeup” is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for.

Structured Synchronization

1. Identify objects or data structures that can be accessed by multiple threads concurrently
2. Add locks to object/module
   - Grab lock on start to every method/procedure
   - Release lock on finish
3. If need to wait
   - `while(needToWait()) condition.wait(lock);`
   - Do not assume when you wake up, signaler just ran
4. If do something that might wake someone up
   - Signal or Broadcast
5. Always leave shared state variables in a consistent state
   - When lock is released, or when waiting

Implementing Synchronization

Concurrent Applications

Semaphores | Locks | Condition Variables

Interrupt Disable | Atomic Read/Modify/Write Instructions

Multiple Processors | Hardware Interrupts

Lock Implementation, Uniprocessor

```
LockAcquire(){
  disableInterrupts ();
  if (value == BUSY) {
    waiting.add(
        current TCB);
    scheduler.suspend();
  }
  else
     value = BUSY;
  enableInterrupts ();
}
```

```
LockRelease() {
  disableInterrupts ();
  if (!waiting.Empty()) {
    thread =
        waiting.remove();
    readyList.
        append(thread);
  }
  else
   value = FREE;
  enableInterrupts ();
}
```

Multiprocessor

- Read-modify-write instructions
  - Atomically read a value from memory, operate on it, and then write it back to memory
  - Intervening instructions prevented in hardware
- Examples
  - Test and set
  - Intel: xchgb, lock prefix
  - Compare and swap
- Does it matter which type of RMW instruction we use?
  - Not for implementing locks and condition variables!

Spinlocks

- Lock where the processor waits in a loop for the lock to become free
  - Assumes lock will be held for a short time
  - Used to protect ready list to implement locks

```
SpinlockAcquire() {
   while (testAndSet(&lockValue) == BUSY)
      ;
}

SpinlockRelease() {
   lockValue = FREE;
}
```

Lock Implementation, Multiprocessor

```
LockAcquire(){
  spinLock.acquire();
  if (value == BUSY){
    waiting.
      add(current TCB);
    scheduler.
      suspend(&spinLock);
  }
  else {
    value = BUSY;
    spinLock.release();
  }
}
```

```
LockRelease() {
  TCB *next;

  spinLock.acquire();
  if (!waiting.Empty()){
    next = waiting.remove();
    scheduler.makeReady(next);
  }
  else {
    value = FREE;
  }
  spinLock.release();
}
```

Lock Implementation, Multiprocessor

```
Scheduler:
  Queue readyList;
  SpinLock schedSpinLock;

  makeReady(TCB *thread){
    disableInterrupts();
    schedSpinLock.acquire();
    readList.add(thread);
    thread->state = READY;
    schedSpinLock.release();
    enableInterrupts();
  }
```

Lock Implementation, Multiprocessor

```
suspend(SpinLock *lock){
  TCB *chosenTCB;

  disableInterrupts();
  schedSpinLock.acquire();
  lock->release();
  runningThread->state = WAITING;
  chosenTCB = readList.getNext();
  thread_switch(runningThread, chosenTCB);
  chosenTCB ->state = RUNNING;
  schedSpinLock.release();
  enableInterrupts();
}
```

Lock Implementation, Linux

- Fast path
  - If lock is FREE, and no one is waiting, test&set
- Slow path
  - If lock is BUSY or someone is waiting, see previous slide

- User-level locks
  - Fast path: acquire lock using test&set
  - Slow path: system call to kernel, to use kernel lock

Futexes

- Safe, efficient kernel conditional queueing in Linux
- All operations performed atomically
  - `futex_wait(futex_t *futex, int val)`
    - if futex->val is equal to val, then sleep
    - otherwise return
  - `futex_wake(futex_t *futex)`
    - wake up one thread from futex's wait queue, if there are any waiting threads

- For more information:

  http://people.redhat.com/drepper/futex.pdf

Ancillary Functions

- `int atomic_inc(int *val)`
  - add 1 to *val, return its original value
- `int atomic_dec(int *val)`
  - subtract 1 from *val, return its original value

Attempt 1

```
void lock(futex_t *futex) {
  int c;
  while ((c = atomic_inc(&futex->val)) != 0)
    futex_wait(futex, c+1);
}

void unlock(futex_t *futex) {
  futex->val = 0;
  futex_wake(futex);
}
```

Attempt 2

State:
- 0 – unlocked
- 1 – No waiting threads
- 2 – Waiting threads

```
void lock(futex_t *futex) {
  int c;
  if ((c = CAS(&futex->val, 0, 1) != 0)
    do {
      if (c == 2 || (CAS(&futex->val, 1, 2) != 1))
        futex_wait(futex, 2);
    }
    while ((c = CAS(&futex->val, 0, 2)) != 0))
}

void unlock(futex_t *futex) {
  if (atomic_dec(&futex->val) != 1) {
    futex->val = 0;
    futex_wake(futex);
  }
}
```

Condition Variables and Semaphores

- The implementation follows the same reasoning for lock implementation