CSL373: Lecture 4
Abstracting synchronization and some subtleties

Past & Present

- Shared resources often require “mutual exclusion”
    - a mutual exclusion between two events is a requirement that they do not overlap in time

    - Conceptually: a thread is assigned exclusive use of a resource until it is done performing a critical set of operations.
- Today:
    - some details about threads
    - how to simplify the construction of critical regions using semaphores and monitors (Chapter 4.6, 6)
    - some nuances

Spinning tricks

- Initial spin was pretty simplistic:

```
lock(L)  {
      for (acquired = 0; !acquired; )
                  aswap acquired, L;
}
```

- Atomic instructions are costly (need to lock the hardware bus), so want to avoid

```
lock(L) {
            for (acquired = 0; !acquired; )
                  while (!L);
                  aswap acquired, L;
}
```

- Problem: what happens if L put in register?

Locking variations

- Recursive locks

```
recursive_lock(l) {
   if(l->owner == cur_thread)
         l->count++;
   else
         lock(l->lock);
         l->count = 0;
         l->owner = cur_thread;
```

```
recursive_unlock(l) {
   if(l->owner != cur_thread
   || l->count < 0)
         fatal(bogus release!);
   l->count--;
   if(!l->count)
         l->owner = -1;
      unlock(l->lock);
```

Why? Synchronization modularity

Can we swap lock() and l->owner assignment statements?

- "trylocks": non-blocking lock acquisition

```
if (!try_lock(l))
         return RESOURCE_BLOCKED;
```

- Exercise: Implement try_lock() and try_unlock() using:
  - aswap instruction
  - lock() and unlock

Blocking problems

- yield: if another thread on run queue, take off, put current on run queue, switch

```
void  yield()  {
        lock(runq);            /* lock run q and dequeue */
        new = deq(runq);
        if (!new)
                unlock(runq);        /* no thread, continue */
        else
                old = current_thread;
                current_thread = new;
                enq(runq, old);            /* put current on runq */
                unlock(runq);
                switch(old, new);   /* context switch */
}
```

- yield() puts the thread back on run queue. What's a problem here? Can we do better?

Blocking on a lock

- sleep(L): Put the current thread in BLOCKED state waiting on lock L.
- wake_sleepers(L): wake-up threads sleeping on L.

```
void sleep(l)  {
    lock(runq);
    new = deq(runq);
    old = current_thread;
    current_thread = new;
    lock(l->sleepers);
    enq(l->sleepers, old);
    unlock(l->sleepers);
    unlock(runq);
    switch(old, new);
}
```

```
void wake_sleepers(l)  {
        lock(runq);
        lock(l->sleepers);
        while (t = deq(l->sleepers)) {
                enq(runq, t)
         }
         unlock(l->sleepers);
         unlock(runq);
}
```

- What’s the difference between yield() and sleep()?

Blocking mechanics

- Producer/consumers: producer puts characters in an infinite buffer, consumers pull out

```
char buf[];  /* infinite buf */
int head = 0, n = 0, tail = 0;
lock l;
void put(char c)
        lock(l);
        buf[head++] = c;
        n++;
        unlock(l);
        wake_sleepers(l);
```

```
int get(void) {
        lock(l);
        if(!n)
                unlock(l);
                sleep(l);
                lock(l);
        c = buf[tail++];
        n--;
        unlock(l);
        return c;
```

- What are some problems? Does it work with one consumer? Does it work with n consumers? What if two consumers are sleeping and get woken up simultaneously?
- How to simplify?

Semaphores

- Synchronization variable [Dijkstra, 1960s]
  - A non-negative integer counter with atomic increment and decrement. Blocks rather than going negative.
  - Used for mutual exclusion and scheduling
- Two operations on semaphore:

  P(sem): decrement counter “sem”. If sem = 0, then block until greater than zero.   Also called wait().

  V(sem): increment counter “sem” by one and wake 1 waiting process (if any). Also called signal().

  Classic semaphores have no other operations.
- Key:

  Semaphores are higher-level than locks (makes code simpler) but not too high level (keeps them relatively inexpensive).

Infinite buffer with locks vs with semaphores

```
char buf[];
int head = 0, tail = 0, n = 0;
lock lock;
void put(char c)
        lock(lock);
        buf[head++] = c;
        n++;
        unlock(lock);
void get(void)
        lock(lock);
        while(!n)
                unlock(lock);
                yield();
                lock(lock);
        c = buf[tail++];
        n--;
        unlock(lock);
        return c;
```

```
char buf[];
int head = 0, tail = 0;
sem holes = N, chars = 0;
void put(char c)
        P(holes);
        buf[head++] = c;
        V(chars);

void get(void)

        P(chars);




        c = buf[tail++];
        V(holes);

        return c;
```

Scheduling with semaphores

- In general, scheduling dependencies between threads
  - T1, T2, …, Tn can be enforced with n-1 semaphores
  - S1, S2, …, Sn-1 used as follows:
    - T1 runs and signals V(S1) when done.
    - Tm waits for Sm-1 (using P) and signals V(Sm) when done.
- (contrived) example: schedule print(f(x,y))

```
float  x, y, z;
sem   Sx = 0, Sy = 0, Sz = 0;
T1:                          T2:                        T3:
  x = …;                        P(Sx);                     P(Sz);
  V(Sx);                        P(Sy);                     print(z);
   y = …;                       z = f(x,y);                …
   V(Sy);                       V(Sz);
   …                             …
```

Monitors

- High-level data abstraction that unifies handling of:
  - Shared data, operations on it, synch and scheduling
  - All operations on data structure have single (implicit) lock
  - An operation can relinquish control and wait on condition

```
// only one process at time can update instance of Q
Class Q   {
          int head, tail;                    //  shared data
          void enq(val)  { locked access to Q instance }
          int deq()  { locked access to Q instance }
}
```

  Can be embedded in programming language:
    - Mesa/Cedar from Xerox PARC
    - Java “synchronized” keyword
- Monitors easier and safer than semaphores
  - Compiler can check, lock implicit (cannot be forgotten)

(Read Ch. 6.7)

Monitors. Try #1

```
synchronized class Queue  {
    int head, tail;   //  shared data
    int *buf;         // (assume) infinite buffer
    void init() {
        head = tail = 0;
     }
     void enq(val)  {
        buf[head++] = val
    }
     int deq()  {
          return buf[tail++];
     }
}
```

- Correct? What do we need?

Monitors. Try #2

```
synchronized class Queue  {
    int head, tail;   //  shared data
    int *buf;         // (assume) infinite buffer
    void init() {
        head = tail = 0;
     }
     void enq(val)  {
        buf[head++] = val
    }
     int deq()  {
          while (tail == head) continue;
          return buf[tail++];
     }
}
```

- Correct? What do I need?

Condition variables: blocking in a monitor

- Three basic atomic operations on condition variables

  condition x, y;
- wait(condition):

  release monitor lock, sleep, re-acquire lock when woken

  usage: while (!exper) wait(condition);
- signal(condition):

  wake *one* process waiting on condition (if there is one)

  Hoare: signaler immediately gives lock to waiter (theory)

  Mesa: signaler keeps lock and processor (practice)

  No history in condition variable (unlike semaphore)
- broadcast(condition)

  wake *all* processes waiting on condition

  Useful when waiters checking different expressions.

Mesa-style monitor subtleties

```
char buf[N];                    // producer/consumer with monitors
int n = 0, tail = 0, head = 0;
condition not_empty, not_full;
void put(char ch)
        if(n == N)
                wait(not_full);
        buf[head%N] = ch;
        head++;
        n++;
     signal(not_empty);
char get()
        if(n == 0)
                wait(not_empty);
        ch = buf[tail%N];
        tail++;
        n--;
        signal(not_full);
        return ch;
```

Consider the following time line:

0. initial condition: n = 0
1. c0 tries to take char, blocks on not_empty (releasing monitor lock)
2. p0 puts a char (n = 1), signals not_empty
3. c0 is put on run queue
4. Before c0 runs, another consumer thread c1 enters and takes character (n = 0)
5. c0 runs.

What is a possible fix?

This code would be correct under Hoare semantics, but incorrect under Mesa

Implementing Condition Variables using Semaphores

```
struct condition {
    int waiting;
    semaphore *sema;
}
```

```
wait(condition *c, lock* l)
{
    waiting++;
    l->release();
    sema->P();
    l->acquire();
}
```

```
signal(condition *c, lock* l)
{
    if (waiting > 0) {
        sema->V();
        waiting--;
    }
}
```

- Why is this solution incorrect? Read Birrell paper for correct solution

Eliminating locks

- One use of locks is to coordinate multiple updates of a single piece of state. How to remove locks here?
  - Duplicate state so each instance only has a single writer
  - (Assumption: assignment is atomic)

- Circular buffer:
  - Why do we need lock in circular buffer?
    - To prevent loss of update to buf.n. No other reason.
  - What is buf.n good for?
    - signaling buf full and empty.
  - How else to check this?
    - Full: (buf.head – buf.tail) == N
    - Empty: buf.head == buf.tail
  - Can we use these facts to eliminate locks in get/put? Exercise.

Lock free synch: 1 consumer, 1 producer

```
int head = 0, tail = 0;
char buf[N];
void put(char c) {
        while((buf.head - buf.tail) == N)
                wait();
        buf.buf[buf.head % N] = c;
        buf.head++;
        }
void get(void) {
        char c;
        while(buf.tail == buf.head)
                wait();
        c = buf.buf[buf.tail % N];
        buf.tail++;
        return c;
}
```

All shared variables have single writer (no lock needed):

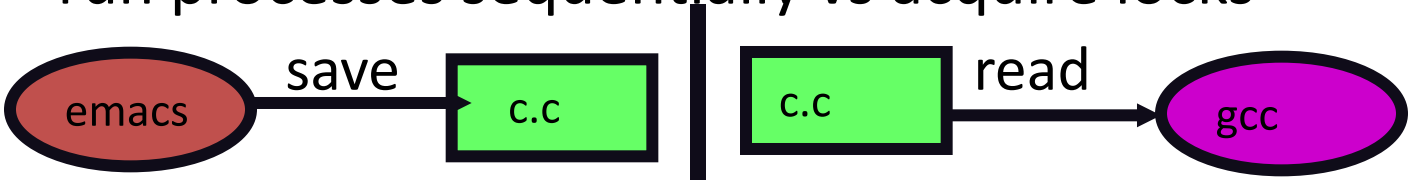
- **head - producer**
- **tail - consumer**
- **buffer:**
  - **head != tail then no overlap and**
    - **buf[head] - producer**
    - **buf[tail] - consumer**
  - **head = tail then empty and consumer waits until head != tail**

invariants:

- **not full: once not full true, can only be changed by producer**
- **not empty: once not empty can only be changed by consumer**

Locks vs explicit scheduling

- Race condition = bad interleaving of processes.
  - We've used locks to prevent bad interleavings
  - Could use scheduler to enforce legal schedules.
- Examples:
  - run processes sequentially vs acquire locks



doc appointment vs emergency room
classroom scheduling vs hostel bathroom
dinner reservation vs showing up
run processes sequentially vs acquire locks

- Tradeoffs?

Transactions

- Mr. X deposits money to a shared bank account
- Mrs. X withdraws the money from the bank account *at the same time*.
- Solution with locks?

```
deposit(account) {
    lock(account);
    <lot of processing
    on account.money>
    account.money++;
    unlock(account);
}
```

```
withdraw(account) {
    lock(account);
    <lot of processing on
    account.money>
    account.money--;
    unlock(account);
}
```

What is the common case? Can we do better for the common case at (maybe) the expense of the uncommon case?

Transactions (aka optimistic concurrency control)

```
deposit(account) {
    m = account.money;
    m++;
    If (no_error)
        commit change
    Else
        rollback & try again
}
```

atomic

Error routine might check if nobody else modified the value of money while it was operating on it.

Rollback might throw away all computed results

Non-Blocking Synchronization (LL/SC) [RISC]

- Semantics of LL:
  - Load memory location into register and mark it as loaded by this processor. Can be marked loaded by more than one
- Semantics of SC:
  - If the memory location is marked as loaded by *this* processor, store the new value and remove all marks from the memory location. Otherwise, don't perform the store. Return whether or not the store succeeded.

```
Lock(lock) :
   while (1) {
      LL  r1, lock
      if (r1 == 1) {
         mov $0, r2
         if (SC r2, lock) break;
      }
   }
```

```
Unlock():
   mov $1, r1
   st   r1, mem
```

LL/SC to implement some operations directly

- e.g. increment mem:

```
while (1) {
    LL r1, mem
    ADDI r1, 1, r1
    if (SC r1, lock) break;
}
```

- Increment operation is now non-blocking: If two threads start to perform the increment at the same time, neither will block – both will complete the add and only one will successfully perform the SC. The other will retry.
- Eliminates problems with locking like: one thread acquires locks and dies, or one thread acquires locks and is suspended for a long time

Synchronization in the real world

- Synchronization whenever >1 user of resource
  - Use same solutions in real world: lock (on door), scheduling (appointments), duplicate resource (everyone has laptop)
- Examples:
  - Contagious disease race conditions
  - One road, multiple cars: traffic lights (scheduling-based synchronization), two lanes ("duplicate" state – trade less utilization for simpler coordination)
  - Bathroom: door(lock), male/female (duplicate state)
  - You & partner: lock = "hacking thread.c" unlock = "done"
  - Parking space: car parked (lock), not parked (unlocked). Parking assignment (lock always, no concurency = bad utilization)

Summary

- Concurrency errors:

  One way to view: thread checks condition(s)/examines value(s) and continues with the implicit assumption that this result still holds while another thread modifies.

- Simplest fixes?

  Run threads sequentially (poor utilization or impossible)

  Do not share state (may be impossible)

- More complex fixes:

  Use locks, semaphores, monitors to enforce mutual exclusion

  Use transactions.