Past & Present

- Shared resources often require "mutual exclusion"
  A mutual exclusion constraint between two events is a requirement that they not overlap in time.

Conceptually: a thread is assigned exclusive use of a resource until it is done performing a critical set of operations.

- Next:
  Some details about threads
  How to simplify the construction of critical regions using semaphores and monitors
  Reading: (7th ed.: Ch. 3.5,6) , 6th ed.: Ch. 4.5, 7

Spinning tricks

- Initial spin was pretty simplistic:
  ```
  lock(L) {
    for(acquired = 0; !acquired; )
      aswap acquired, L;
  }
  ```

- Atomic instructions are costly, 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;
  }
  ```

- Why? Synchronization modularity

- "trylocks": non-blocking lock acquisition
  ```
  if ( !try_lock( l ) )
    return RESOURCE_BLOCKED;
  ```

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);
  ```

- Critical Sections with Semaphores

- Emulate a lock:
  ```
  initializing a semaphore with one
  lock_acquire becomes P(seaphore)
  lock_release becomes V(seaphore)
  ```

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

Critical Sections with Semaphores

- Emulate a lock:
  ```
  P(mutex);
  buf[head++] = c;
  ```
Infinite buffer w/ locks vs w/ semaphores

```c
char buf[N];
int head = 0, tail = 0, n = 0;
lock;
void put(char ch) {
    lock(lock);
    buf[(head++) % N] = ch;
    n++;
    unlock(lock);
    return ch;
}
void get(void) {
    lock(lock);
    while(n)
        P(holes);
    c = buf[(tail++) % N];
    n--;
    unlock(lock);
    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 on Sm-1 (using P) and signals V(Sm) when done.

  (contrived) example: schedule print(f(x,y))
  ```c
  float x, y, z;
  sem Sx = 0, Sy = 0, Sz = 0;
  T1:
  x = ...
  P(Sx);
  y = ...
  V(Sy);
  T2:
  T3:
  z = f(x,y);
  V(Sz);
  print(z);
  ```

Common semaphore usage idioms

- Waiting for an activity to finish:
  ```c
  sema_init(sema,0);
  thread_create(sema);
  sema_down(sema);
  In new thread:
  ... do something ...
  sema_up(sema);
  ```

- Mutual exclusion/controlling access:
  ```c
  sema_init(sema,1);
  sema_down(sema);
  ... do something ...
  sema_up(sema);
  ```

Condition variables: blocking in a monitor

- Three basic atomic operations on condition variables condition x, y:
  - wait(condition):
    ```c
    release monitor lock, sleep, re-acquire lock when woken usage: while(!exper) wait(condition);
    ```
  - signal(condition):
    ```c
    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):
    ```c
    wake "all" processes waiting on condition useful when waiters checking different expressions.
    ```

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
  ```c
  // 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:
  ```c
  Mesa/Cedar from Xerox PARC
  ```

  Monitors easier and safer than semaphores
  ```c
  Compiler can check, lock implicit (cannot be forgotten)
  ```

Mesa-style monitor subtleties

```c
char buf[N];
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
    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 are the possible fixes?
    ```
```
More mesa-style subtleties

```c
char buf[N];                     // producer/consumer with monitors
int n = 0, tail = 0, head = 0;
condition not_empty, not_full;
void put(char ch) {
    while(n == N) { // producer
        wait(not_full);
        buf[head] = ch;
        head = (head+1)%N;
        n++;
    }
    signal(not_full);
}

char get() {
    while(n == 0) { // consumer
        wait(not_empty);
        ch = buf[tail];
        tail = (tail+1) % N;
        n--;
    }
    signal(not_full);
    return ch;
}
```

When can we replace "while" with if?

Eliminating locks

- One use of locks is to coordinate multiple updates of 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?

Lock free synch: 1 consumer, 1 producer

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

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
gcc emacs save c.c c.c read gcc
doc appointment vs. emergency room classroom scheduling vs. bathroom stall
dinner reservation vs. showing up
  - run processes sequentially vs. acquire locks
  - Tradeoffs?

Non-blocking/wait free synchronization

- How about getting correct interleaving by detecting and retrying when a bad interleaving occurred?
  - Don't need locks to synchronize.
- Example: hits = hits + 1;
  - A) Read hits into register R1.
  - B) Add 1 to R1 and store it in R2.
  - C) Atomically store R2 in hits only if hits==R1.(i.e. CAS) If store didn't write goto A)
- Can be extended to any data structure:
  - A) Make copy of data structure, modify copy.
  - B) Use atomic word compare-and-swap to update pointer.
  - C) Goto A if some other thread beat you to the update.

Non-Blocking synchronization (2)

- Other names:
  - Wait free synchronization, Lock free synchronization.
  - Optimistic concurrency control.
- Modern machine have support for it:
  - x86 CMPXCHG, CMPXCHG8B – Compare and Exchange.
  - Someone wrote an entire OS with no locks!
- Useful properties:
  - Synchronizes with interrupt handlers.
  - Remove overhead (CPU/memory) locks.
  - Deals with failures better. (e.g. process dies with locks)
- Issues:
  - Lots of retrying under high load.
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:
- use locks, semaphores, monitors to enforce mutual exclusion