

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], 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;
}

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

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);
}

int get(char c) {
    lock(l);
    if(!n)
        unlock(l);
        sleep(l);
    lock(l);
    c = buf[tail++];
    n--;
    unlock(l);
    return c;
}
```
- what are some problems?

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()/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:

```
Initializing a semaphore with one
Lock_Acquire becomes P(semaphore)
Lock_Release becomes V(semaphore)

char buf[]; /* infinite buf */
int head = 0, n = 0, tail = 0;
sem mutex = 1;
void put(char c) {
    P(mutex);
    buf[head++] = c;
    n++;
    V(mutex);
}
```

Infinite buffer w/ locks vs w/ 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;
}

```

problems?

Scheduling with semaphores

- In general, scheduling dependencies between threads T_1, T_2, \dots, T_n can be enforced with $n-1$ semaphores, S_1, S_2, \dots, S_{n-1} used as follows:

T_1 runs and signals $V(S_1)$ when done.

T_m waits on S_{m-1} (using P) and signals $V(S_m)$ when done.

- (contrived) example: schedule $\text{print}(f(x,y))$

```

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

```

Common semaphore usage idioms

- Waiting for an activity to finish:


```

sema_init(sema,0);
thread_create(sema);
sema_down(sema);

```

 In new thread:


```

... do something ...
sema_up(sema);

```
- Mutual exclusion/controlling access:


```

sema_init(sema,1);
sema_down(sema);
... do something ...
sema_up(sema);

```

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

Condition variables: blocking in a monitor

- Three basic atomic operations on condition variables
 - condition x, y ;
- $\text{wait}(\text{condition})$:
 - release monitor lock, sleep, re-acquire lock when woken
 - usage: `while(!exper) wait(condition);`
- $\text{signal}(\text{condition})$:
 - 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. c_0 tries to take char, blocks on `not_empty` (releasing monitor lock)
2. p_0 puts a char ($n = 1$), signals `not_empty`
3. c_0 is put on run queue
4. Before c_0 runs, another consumer thread c_1 enters and takes character ($n = 0$)
5. c_0 runs.

What are the possible fixes?

More mesa-style subtleties

```

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)
        wait(not_full);
    buf[head] = ch;
    head = (head+1)%N;
    n++;
    signal(not_full);
char get()
    while(n == 0)
        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


```

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++;
}
char 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. 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**