Atomicity

Atomicity: the property of a piece of code that it executes **alone and without interference**. Actually two properties:

Either the operation succeeds or it doesn't. There is no such thing as "part done".
If it doesn't succeed, it:

- leaves the state of the process as if it weren't called.
- returns an error code.

Really two kinds:

- **Kernel atomicity**: defines operations that are indivisible in any context.
- **Application atomicity**: defines operations that are indivisible in the context of the application.

A section of code that should be atomic with respect to others in the same application is called a **critical section**.

A **lock** is a mechanism whereby code is forced to wait for **exclusive access to a critical section**.

**Mutex** = mutual exclusion = exclusive access.

Today's installment: **things that support atomic operations**: mutexes, semaphores, and others.
Kinds of atomicity:

**synthetic (voluntary) atomicity**: programs and/or threads obey locking to avoid operating at the same time.

**kernel atomicity**: when a system call is executing, the scheduler won't interrupt it.

**instruction atomicity**: some machine instructions preclude concurrent operations.

**special-purpose system calls**: some specific system calls execute program instructions in an atomic (uninterruptible) fashion.

The game of operating systems
Construct mechanisms for synthetic atomicity (in processes)
from low-level atomic building blocks (instructions, system calls).
Facts about locking

Locks slow down code in order to avoid state errors. Locking has overhead, i.e., it takes time to lock and unlock.

Errors in locking manifest as extremely rare execution errors.

Important note:

Correctness of a locking scheme is not testable, but instead must be mathematically proven. (You can demonstrate that a locking scheme is inadequate via testing, but cannot demonstrate that it is adequate.)
Kinds of locks

Between threads: mutexes
  user mode (except when they block!)
  exploit **shared memory**
  **binary: locked or unlocked**
Between threads: **unnamed semaphores**
  user mode (except when they block!)
  exploit shared memory
  they're integers
Between processes: **named semaphores**
  kernel mode: live in kernel memory.
  no shared **memory**.
  They're integers.
Between machines: **network semaphores**
  network objects
  use network communications
  no shared **architecture**.
What's a mutex?

Basically, one word of user memory.
Utilize a simple race condition to acquire lock, based upon atomicity of operations in assembly language
If the word of memory is > 0, then a lock has been acquired.
If the word of memory == 0, then there is no lock.
Testset: if word==0 then word=1 as an atomic operation.

Guarantees and contracts

After a lock,
  exactly one thread will acquire the lock at a time.
  Others will be queued ("strong" semantics)

The catch:
  Strong semantics require that we block the calling thread and queue it.
  This requires the kernel. Can't be done in user space.
pthread_mutex_t mutex; // in shared memory
...
pthread_mutex_init(&mutex);
...
pthread_mutex_lock(&mutex);
...
pthread_mutex_unlock(&mutex);
...
pthread_mutex_destroy(&mutex);
Implementing mutexes

- A mutex lock is implemented with some sort of atomic operation, e.g., a "testset" operation, like
  
  ```c
  if (mutex==0) mutex=1; // as an atomic operation
  ```

  If two of these race, one wins; race is disambiguated by atomicity.

- A mutex unlock simply resets a mutex that is 1, and does not invoke a race:
  
  ```c
  if (mutex==1) mutex=0;
  ```

- But, after an unlock, the scheduler is notified that any thread waiting for a mutex lock is ready to run, after which

  ```
  mutexes have strong ordering:
  ```

- A mutex has **strong ordering** if threads obtain locks in **first-come, first-served order (FCFS)**.

A clever dance of kernel and user objects

The memory location is in userspace.

The testset operation occurs in userspace.

A system call queues the losers of the race.
A semaphore is an integer to which access is atomic. Initialized to some integer value. locks decrement it. unlocks increment it. value determines number of concurrent locks available.

Several kinds of semaphores:
Thread semaphore: shared memory, between threads.
Named semaphore: kernel memory, between processes.
Network semaphore: network level, between machines.
mutexes                  shared memory semaphores
pthread_mutex_init(&mut,NULL) sem_init(&sem,0,value)
pthread_mutex_lock(&mut)    sem_wait(&sem)
pthread_mutex_unlock(&mut)  sem_post(&sem)
pthread_mutex_destroy(&mut) sem_destroy(&sem)

wait will proceed and decrement the semaphore value if that value >0. Otherwise it blocks.
post will increment the semaphore value and never blocks. It's an unlock.

Create a semaphore with value of 1 === mutex.
Use of semaphores

Wednesday, October 14, 2015  4:58 PM

Browsers need n threads for downloading.
Set semaphore to n
Decrement when starting a thread.
Increment when a thread dies.
Value of semaphore is # of active threads.
Mutexes versus semaphores

- Mutexes can only be locked or unlocked; they are sometimes called "binary semaphores".
- Semaphores have an integer value that is the number of locks available. Can read a semaphore's value via `sem_getvalue(&sem, &val)` where `val` is an int.
What's a (kernel) semaphore?

- Basically, a word of **kernel** memory.
- System calls increment and decrement the word of memory;
- these are **atomic** because during the call, the **scheduler** is not allowed to interrupt execution!
- So processes that are not sharing memory can share kernel semaphores.
### Using a semaphore

<table>
<thead>
<tr>
<th>Unnamed semaphore</th>
<th>Named semaphore (kernel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem_t sem;</td>
<td>sem_t *sep;</td>
</tr>
<tr>
<td>sem_init(&amp;sem,0,1)</td>
<td>For creator:</td>
</tr>
<tr>
<td></td>
<td>sep=sem_open(&quot;/name&quot;, O_CREAT,0644,1);</td>
</tr>
<tr>
<td></td>
<td>For another process:</td>
</tr>
<tr>
<td></td>
<td>sep=sem_open(&quot;/name&quot;, 0);</td>
</tr>
<tr>
<td>sem_destroy(&amp;sem);</td>
<td>To stop using it:</td>
</tr>
<tr>
<td></td>
<td>sem_close(sep);</td>
</tr>
<tr>
<td></td>
<td>To remove it entirely:</td>
</tr>
<tr>
<td></td>
<td>sem_unlink(&quot;/name&quot;);</td>
</tr>
</tbody>
</table>
Initiating process creates a file of type semaphore:

```c
sem = sem_open("/name", O_CREAT, 0644, 1);
"/name": name of the semaphore as a file.
O_CREAT: create it if it doesn't exist.
0644: file protection in octal: everyone can read it, owner can change it.
1: initial value of semaphore when created.
```
Dumb example: one process stops executing when another tells it to: First, the listener: "go.c"

```c
sem_t* shutup;

main()
{
    pthread_t thread1, thread2;
    void *retptr;
    int i;
    shutup = sem_open("/shutup_semaphore", O_CREAT, 0755, 1);
    if (!shutup) {
        fprintf(stderr, "go: no semaphore found!\n");
        exit(1);
    }
    while (1) {
        int val;
        sem_getvalue(shutup, &val);
        fprintf(stderr, "go: got value %d\n", val);
        if (val == 0) break;
        sleep(1);
    }
    sem_close(shutup);
    sem_unlink("/shutup_semaphore");
    fprintf(stderr, "bye from go\n");
}
```

0644 means:
6 for me = 4+2 = 4 (I can read) + 2 (I can write: post, wait)
4 for group = 4 + 0 = 4 (group can read).
4 for everyone else = 4 + 0 = 4 (others can read).
Another process can open the listener's semaphore:

```c
sem_t* shutup;

main()
{
    int val;
    shutup = sem_open("/shutup_semaphore", 0);
    if (!shutup) {
        fprintf(stderr, "stop: no semaphore found!
        ");
        exit(1);
    }
    sem_getvalue(shutup, &val);
    if (val>0) { // not locked already
        sem_wait(shutup);
        sem_getvalue(shutup, &val);
        fprintf(stderr,"stop: got value %d
", val);
    } else {
        fprintf(stderr, "stop: value is already 0
", val);
    }
    sem_close(shutup);
    fprintf(stderr,"bye from stop\n");
}
```

Pasted from `<http://www.cs.tufts.edu/comp/111/examples/Locks/stop.c>`
What's a network semaphore?

- A data structure shared among processes on different computers.
- Can be implemented many different ways.
- Behavior same as kernel semaphore, but much slower in execution (time between someone's unlock() and someone else's lock() is longer).
The astute programmer might realize that there are two interfaces to semaphores:

- `sem_wait` and `sem_post`: work on **one semaphore**
- `semop`: works on **a group of semaphores**.

reason: sometimes one needs atomicity for several operations at the same time.
#define FREE 0
#define USED 1
int group;
...
group = semget(IPC_PRIVATE, 2, 0755);
struct sembuf sb[2] = {
  // num    initial  mode: 0=set, 1=add
  { FREE, BUFSIZE-1, 0 },  // free space in buffer
  { USED,         0, 0 },  // elements in queue
};
semop(group, sb, 2); // initial setup
...
sb[0].sem_num=USED ;
sb[0].sem_op=1;     // add one
sb[0].sem_flg=1;    // relative: add sem_op to value
sb[1].sem_num=FREE ;
sb[1].sem_op=-1;    // subtract one
sb[1].sem_flg=1;    // relative: add sem_op to value
semop(group, sb, 2); // do both of these!
// blocks until both are done!

Pasted from <http://www.cs.tufts.edu/comp/111/examples/Locks/semop.c>
POSIX standard: sem_wait, sem_post.
System-V: semop
Reason System-V survived is **atomicity:**
   one can assure that a group of operations occur concurrently.

Compare
   sem_wait(&avail); sem_post(&used);
against
   semop(...)
that I just did, that does both at the same time!

A deep theoretical result: **if every process asks for every lock it needs at the same time, then deadlocks due to locks cannot occur.**

More about this later.
A deadlock is a state in which neither of two or more processes can proceed.

<table>
<thead>
<tr>
<th>Lock A</th>
<th>Lock B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lock A</td>
</tr>
</tbody>
</table>

Why deadlocks occur with locks
We will see that the order of locking influences -- in a theoretical way -- whether deadlock can occur.

But atomic allocations of multiple locks cannot deadlock in the way that sequences of single locks can.
Problem: outsourcing websites to the third world produces sucky code.  
Main issue: site deadlocks due to messed up locking.  
Proposed solution: track schedule of the processes that caused the deadlock, and prevent that schedule in the future.

Two possibilities:
- If code is not awful, preventing the deadlock schedules "fixes" the code.  
- Otherwise, all schedules are prevented!
Assuring Atomicity of an operation:
  o Switch to kernel mode
    (Switch to kernel memory map).
  o Get (user) address of thing to change
  o Translate into kernel address of thing to change.
  o Disable interrupts and scheduler.
  o Do instruction on translated address.
  o Enable interrupts and scheduler.
  o Switch back to user mode.
    (Switch back to user memory map).
  o Continue user-mode execution
int i; atomic_inc((atomic_t *)&i); // linux

**Diagram:**

```
User

Program

user memory map

0x3000

atomic_inc((atomic_t *)&i)

next mutex

Kernel

kernel memory map (case 1)

0x7000

OS

enable kernel map

got pointer to i...

(0x3000)

translate into kernel address (0x7000)

disable uts, sched ut (0x7000)

disable uts, sched

disable user map

return.
```
Linux used to provide:

```c
atomic_inc(atomic_t *p);
atomic_dec(atomic_t *p);
atomic_add(int i, atomic_t *p);
Etc.
```
Uses of primitive atomic actions
Implementing semaphores in local domain.
Avoiding simple races involving integers.

Problem:
Atomic operations are system-specific (and relatively slow).
Semaphores, Monitors, Mutex's are system-independent (and relatively fast).
Named semaphores:
  - Kernel object.
  - Semaphore is simply an array offset in a kernel table.
  - **Limited number available.**

Unnamed semaphore:
  - User object (with kernel queue)
  - A piece of shared memory.
  - Unlimited number available.

Mutex:
  - Special case of unnamed semaphore.
  - User/thread object.
  - Unlimited number available.
How to do bounded buffer *without locking*. Based upon *atomic swap instruction*. 