Multiple Processes in One World

Safe? Not automatically.

Suppose one process is modifying shared state information.

- Can another process safely modify at the same time?
- Can another process safely read at the same time?

Are there inherently safe sharing scenarios?

Errors may be irreproducible.

Consider a shared counter…

So why share?

Cost: buy limited # of resource units and share among many processes

Information: need results from other processes

- promotes speed; parallel threads working on same state
- promotes modularity; one process may serve as front-end for others

Race Conditions

race condition: timing-dependent error involving shared state.

occurrence depends on vagaries of scheduling

must make sure that ALL possible schedules are safe

number of possible schedules may be HUGE

```
if ( n == stack_size )   // A
   return FULL;          // B
stack[n] = v;            // C
n = n + 1;               // D
```


Race Conditions


```
// Thread a
i = 0;
while ( i < 10 )
   i = i + 1;
print "A won!";
```

```
// Thread b
i = 0;
while ( i > -10 )
   i = i - 1;
print "B won!";
```

Note: variable `i` is shared

Who wins?

Must one win?

Responses

Do nothing…

shared counter may not require a precise result

but often precision does matter

Avoid sharing…

whenever possible, duplicate or partition state

can't always avoid sharing

What's the root of the problem?

bad interleaving of processes…

… so prevent them from occurring

Atomicity

atomic unit: instruction sequence guaranteed to execute indivisibly

critical section: instruction sequence in which shared state must be accessed atomically

If two threads attempt to execute the same atomic unit at the same time, one thread will execute the whole sequence before the other thread begins it.

But how do we make multiple instructions seem like atomic ones?

Atomicity on a Uniprocessor

The only requirement is that when a thread is in a critical section it will not be preempted.

OS traditional approach is that threads will disable/enable interrupts.

```
old = intr_disable();
hits = hits + 1;
intr_set_level( old );
```

Dangerous… programmer error can disrupt entire system, not merely lead to incorrect results for the threads that are sharing state.

Could have the scheduler check the thread's PC versus a table of critical section addresses… requires compiler support and extra work.

Atomicity on a Multiprocessor

Simply avoiding certain preemptions is no longer sufficient…

Hardware support?

- e.g, atomic increment operation
- not a general solution
- can be used to create atomic primitives that do provide a solution

General solution:

- when a thread enters a critical section, it sets a lock
- no other thread can enter the critical section without holding the lock
- when a thread leaves a critical section, it unsets the lock

Locks

A lock is shared amongst the relevant threads

acquire(lock):	acquire exclusive access to lock;
	if lock is already acquired, wait for it
release(lock)	release exclusive access to lock

Pattern is obvious: bracket critical section in lock/unlock calls

This type of lock is often called a *mutex*

Locks let us create big atomic units (critical sections) from small ones (locks)

Lock Rules

Simple rules for easy concurrency:

- every shared variable is protected by a lock
- every thread must hold the relevant lock before it touches the shared vble

Must have a guarantee that two threads cannot hold the same lock at the same time.

Implementing Locks

A simple attempt:

```
acquire( Lock& L ) {
   while ( L == 0 ) continue;
   L = 0;
}
release( Lock& L ) {
   L = 1;
}
```


Second Attempt

Focus on the uniprocessor issue for now:

```
acquire( Lock& L ) {
   disable_preemption();
      while ( L == 0 ) continue;
      L = 0;
   enable_preemption();
}
release( Lock& L ) {
   L = 1;
}
```

Third Attempt

Focus on the uniprocessor issue for now:

```
acquire( Lock& L ) {
   acquired = 0;
   while ( !acquired ) {
      disable_preemption();
      if ( L == 1 ) {
         acquired = 1;
         L = 0;
      }
      enable_preemption();
   }
}
```


Multiprocessing Locks

Turning off other processors is too costly… or impossible.

Atomic primitives can be used to build locks

- test and set instruction
- atomic swap (atomically swap values in register and memory)

```
acquire( Lock& L ) {
   acquired = 0;
   while ( !acquired )
      aswap( acquired, L);
}
```

Spin or block?

Blocking isn't free. Best choice depends on how long thread must wait.

One strategy:

- spin for length of block cost
- if lock not available, then block

Performance is within a factor of 2 of optimal.

General Observations

There are many other workable solutions

All share three common safety requirements:

- mutual exclusion
- progress (deadlock free)
- bounded (starvation free)

Some "nice" properties:

- fair (don't make some wait longer than others)
- efficient (don't waste resources while waiting)
- simple