Past and Present

- Previous: isolated processes
  - **modularize system**
  - **share resources**
  - **speed**

- Now: safe non-isolated processes
  - **Processes share state (computer, files, memory).**
  - **Concurrent access = bugs.**
    - **Example: single lane road, two approaching cars**
- Readings:
  - **Silbershatz/Galvin: 7th Ed – ch. 6; 6th Ed - ch. 7**

Multiple processes, one world: safe?

- No. Bugs if one process writes state that could be simultaneously read/written by another.
  - **emacs writes out file while gcc compiling it.**

  - **Result? Hard to predict, except that its probably not going to be correct (or repeatable: have fun).**
  - **Always dangerous? (No. More later.) But often enough that you better think carefully.**
- When safe? Isolated processes
  - **"isolated" = shares no state with any other process**
  - **doesn't really exist: share file system, memory, …**

Isolated vs non-isolated processes

- isolated: no shared data between processes
  - **If P produces result x, then running any other set of independent processes P', P", … wouldn't change it.**
  - **Scheduling independence: any order = same result**
  - **Consider: internet, lots of independent machines. If don't share state, doesn't matter what other people do.**
- Non-isolated: share state
  - **Result can be influenced by what other processes running**
  - **Scheduling can alter results**
  - **Big problem: non-deterministic. Same inputs != same result. Makes debugging very very hard.**

Why share? Two core reasons

- Cost: buy m, amortize cost by letting n share (n > m)
  - **One computer, many jobs; one road, many cars; this classroom, many other classes.**
- Information: need results from other processes

  - **Gives speed: parallel threads working on same state**
  - **Gives modularity(?!): ability to share state lets us split tasks into isolated processes (gcc, emacs) and communicate just what is necessary**
  - **Sharing information hugely important. Consider impact of new ways to share information (print, telephone, internet, www, human voice)**

Example: two threads, one counter

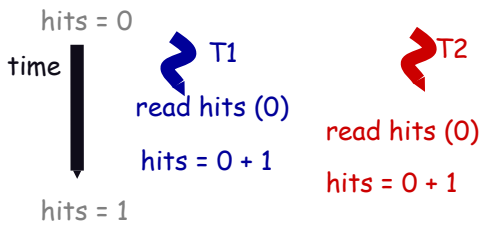
- Assume a popular web server. Uses multiple threads (on multiple processors) to speed things up.
- Simple shared state error: each thread increments a shared counter to track the number of hits today:

```
hits = hits + 1;
```

- What happens when two threads execute this code concurrently?

Fun with shared counters

- One possible result: lost update!



- **One other possible result: everything works.**
  - **Bugs in parallel code are frequently intermittent. Makes debugging hard.**
- Called a "race condition"

Race conditions

- Race condition: timing dependent error involving shared state.
  Whether it happens depends on how threads scheduled
- *Hard* because:
  Must make sure all possible schedules are safe. Number of possible schedules permutations is huge.

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

Some bad schedules? Some that will work sometimes?
They are intermittent. Timing dependent = small changes (printf's, different machine) can hide bug.

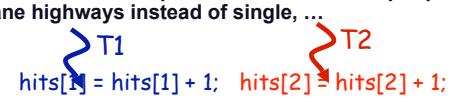
More race condition fun

```
Thread a:              Thread b:
i = 0;                 i = 0;
while(i < 10)          while(i > -10)
  i = i +1;              i = i – 1;
print 'A won!';         print 'B won!';
```

Who wins?
Guaranteed that someone wins?
What if both threads run on own identical speed CPU executing in parallel? (Guaranteed to go on forever?)
What to do???

Dealing with race conditions

- Nothing. Can be a fine response
  if "hits" a perf. counter, lost updates may not matter.
  Pros: simple, fast. Cons: usually doesn't help.
- Don't share: duplicate state, or partition:
  Do this whenever possible! One counter per process, two lane highways instead of single, …

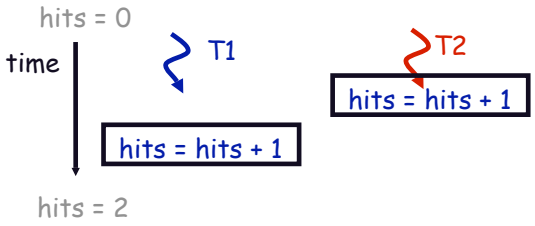


Pros: simple again. Cons: never enough to go around or may have to share (gcc eventually needs to compile file)
(ee/architecture note: cache interference)

- Is there a general solution? Yes!
  What was our problem? Bad interleavings. So prevent!

Atomicity: controlling race conditions

- atomic unit = instruction sequence guaranteed to execute indivisibly (also, a "critical section").
  If two threads execute the same atomic unit at the same time, one thread will execute the whole sequence before the other begins.



How to make multiple inst's seem like one atomic one??

Making atomicity: Uniprocessor

- Only req': thread not preempted in critical section.
  Have scheduler check thread's program counter:

```
while(1) {      /* naïve dispatcher loop */
      interrupt thread;
      if pc != critical section
            save old thread state
            pick thread
            load new thread state
      jump to thread
```

Pro: fast atomicity. Con: need compiler support.

- OS Traditional: threads disable/enable interrupts:

```
/* pintos */
old = intr_disable();
hits = hits + 1;
intr_set_level(old);
```

```
/* openbsd */
int s = splhigh();
hits = hits + 1;
splx(s);
```

```
save_flags(flags); /* linux */
cli()
hits = hits + 1;
restore_flags(flags);
```

Pro: works. Con: infinite loop = stop the world.

Making atomicity: Multiprocessor

- Must prevent any other thread from executing critical section
- Hardware support: could wire in atomic increment
  pro: works. Con: not a general approach
  instead, we do a variant: provide a hardware building block that can construct atomic primitives
- General solution: locks (just like on door)
  when thread enters critical section, locks it so no other thread can enter. when it leaves, thread unlocks it.

Pro: general. Con: manual, low level (better later…)

Locks: making code atomic.

- Lock: shared variable, two operations:
  - **"acquire" (lock): acquire exclusive access to lock, wait if lock already acquired.**
  - **"release" (unlock): release exclusive access to lock.**
- How to use? Bracket critical section in lock/unlock:

```
lock hit_lock;
...
lock(hit_lock); hit = hit + 1; unlock(hit_lock);
```

Result: only one thread updating counter at a time.

Access is "mutually exclusive": locks in this way called "mutexes"

What have we done? Bootstrap big atomic units from smaller ones (locks)

Lock rules for easy concurrency

- Every shared variable protected by a lock
  - **shared = touched by more than one thread**

```
int stack[ ], n;
lock s_l, n_l;
```

- Must hold lock for a shared variable before you touch
  - **essential property: two threads can't hold same lock at same time**
- Atomic operation on several shared variables:
  - **Acquire all locks before touching, don't release until done**

```
lock(s_l); lock(n_l);
stack[n++] = v;
unlock(s_l); unlock(n_l);
```

Implementing locks. Try #1

- A simple implementation:

```
lock( L )
{
  while( L == 0 ) continue;
  L = 0;
}

unlock( L )
{
  L = 1;
}
```

- Does this work?

Implementing locks. Try #2

- Lets try to get a uniprocessor version right first:

```
lock( L ) {
    disable_preemption();
    while( L == 0 ) continue;
    L = 0;
    enable_preemption();
}

unlock( L ) {
  L = 1;
}
```

- Works?

Implementing locks. Try #3

- Uniprocessor correct:

```
lock(L) {
     acquired = 0;
     while(!acquired)
          disable_preemption();
                if (L == 1)
             acquired = 1;
                L = 0;
              enable_preemption();
}
unlock( L ) {  L = 1; }
```

- Issues?
  - **What's a better thing to do if lock already acquired?**

Implementing multiprocessing locks

- How?
  - **Turning off other processors probably too expensive. Or impossible (OSes don't let user-level threads do so)**
  - **Instead: have hardware support.**
    - **Do we need a hardware lock instruction? No. Can build locks from more primitive instructions.**
    - **Common primitives: test & set, atomic swap, ...**
- Example instruction: atomic swap (aswap):
  - **aswap mem, R: atomically swap values in reg and memory**

```
temp = R; R = mem; mem = temp;
```

  - **Hardware guarantees the two assignments are atomic.**
  - **This primitive lets us implement any other concurrency primitive!**

A multiprocessor lock using aswap

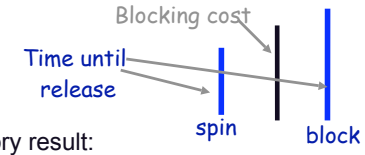
- A aswap-based lock:

```
lock(L) {
     acquired = 0;
     while(!acquired)
          aswap acquired, L;
}
unlock(L) {
     L = 1;
}
```

- Called a "spin lock": thread spins until it acquires lock.
- Problem with spinning?

Spin or block?

- Blocking is not free, so correct action depends on how long before lock released.
  - **Released "quickly": spin-wait.**
  - **Released "slowly": block (yield)**



- Pretty theory result:
  - **Spin for length of block cost**
  - **If lock not available, then block.**
  - **Performance always within a factor of two of optimal!**

Optimality intuition

- Let cost of block = n cycles.
- If we acquire lock after m cycles of spinning (m <= n) then m is the optimal cost
  - **nothing else would have been faster: blocking immediately would have cost n and m <= n.**
- If we spin for n cycles, then block, cost = 2n.
  - **If we blocked immediately, would cost n.**
  - **Therefore, within 2 of optimal.**
- Same strategy works any situation where you have two solutions:
  - **one with an incremental cost**
  - **one with an up front cost.**
  - **Pay incremental cost until = up front cost, then switch**

General atomicity requirements

- We've shown one way to implement critical sections (locks). There are many others. However, they all share three common safety requirements:
  - **Mutual exclusion: at most one process at a time is in the critical section**
  - **Progress (deadlock free)**
    - **If several simultaneous requests, must allow one to proceed.**
    - **Must not depend on processes outside critical section**
  - **Bounded (starvation free): A process attempting to enter critical section will eventually succeed.**
- Some nice properties (to strive for):
  - **fair: don't make some wait longer than others.**
  - **efficient: don't waste resources waiting.**
  - **oh yeah: and simple.**

Summary

- Many threads + sharing state = race conditions
  - **one thread modifying while others reading/writing**
- How to solve? Intuition:
  - **private state doesn't need to be protected**
  - **multiple threads run one after another can't have race conditions**
  - **SO: to make multiple threads behave like one safe sequential thread, force only one thread at a time to use shared state.**
  - **General solution is to use locks. Let us bootstrap to arbitrarily sized atomic units.**
- Next: higher-level mutual exclusion primitives
  (Read Birrell paper and Ch. 6 if you haven't yet.)