Announcements

- Project 1 is due Feb 27, 11:59pm
- I’m not getting enough questions
- Project 1 Repeat Help Session
  - Wednesday (tonight) MCB 126, 7pm
- *nix Crash Course offered: Feb 9, 8:30pm
  - tomorrow
- Reading: Section 5.1 through 5.4

Recap: Disabling Interrupts

- (this applies to all variations)
- Works for Critical Section, but sledgehammer solution
  - Infinite loop inside CS means machine locks up
  - If you have to block (give up CPU) mutual exclusion with other threads is not guaranteed
  - Any function that transitively calls thread_block() may block
- Use this to protect data structures from concurrent access by interrupt handlers
  - Keep sections of code where irqs are disabled minimal (nothing else can happen until irqs are reenabled – latency penalty!)
- Want something more fine-grained
  - Key insight: don’t exclude everybody else, only those contending for the same critical section

Critical Section Problem

- A solution for the CS Problem must
  1) Provide mutual exclusion: at most one thread can be inside CS
  2) Guarantee Progress: (no deadlock)
     - if more than one threads attempt to enter, one will succeed
     - ability to enter should not depend on activity of other threads not currently in CS
  3) Bounded Waiting: (no starvation)
     - A thread attempting to enter critical section eventually will
       (assuming no thread spends unbounded amount of time inside CS)
  - A solution for CS problem should be
    - Fair (make sure waiting times are balanced)
    - Efficient (not waste resources)
    - Simple

Locks

- Thread that enters CS locks it
  - Others can’t get in and have to wait
- Thread unlocks CS when leaving it
  - Lets in next thread
    - which one?
      - FIFO guarantees bounded waiting
      - Highest priority in Proj1
- Lock is an abstract data type
  - Provides (at least) init, acquire, release

Implementing Locks

- Let’s discuss how to implement locks to solve CS problem
- Later talk about semaphores
- Different solutions exist to implement locks for uniprocessor and multiprocessors
- Will talk about how to implement locks for uniprocessors first – next slides all assume uniprocessor
• Does this work?

No – does not guarantee mutual exclusion property – more than one thread may see "state" in UNLOCKED state and break out of while loop. This implementation has itself a race condition.

• Associate each shared variable with lock L

– One lock only protects one (or a small set of) variables – how to pick that set?

• Ideal, want fine-grained locking

– Sometimes used when retrofitting non-threaded code into threaded framework

• Examples:

  • "KBL" Big Kernel Lock in Linux
  • fslock in Pintos Project 2

• Ideally, want fine-grained locking

  – One lock only protects one (or a small set of) variables – how to pick that set?

Implementing Locks, Take 1

| lock_acquire(struct lock *) |
| while (l->state == LOCKED) |
| continue; |
| l->state = LOCKED; |

lock_release(struct lock *)

l->state = UNLOCKED;

Does this work?

No – does not guarantee mutual exclusion property – more than one thread may see "state" in UNLOCKED state and break out of while loop. This implementation has itself a race condition.

Implementing Locks, Take 2

lock_acquire(struct lock *)

l->state = LOCKED;

disable_preemption();

while (l->state == LOCKED)

l->state = LOCKED;

disable_preemption();

l->state = LOCKED;

enable_preemption();

lock_release(struct lock *)

l->state = UNLOCKED;

disable_preemption();

l->state = UNLOCKED;

enable_preemption();

Does this work?

No – does not guarantee progress property. If one thread enters the while loop, no other thread will ever be scheduled since preemption is disabled – in particular, no thread that would call lock_release will ever be scheduled.

Implementing Locks, Take 3

lock_acquire(struct lock *)

while (true) |

disable_preemption();

if (l->state == UNLOCKED) |

return;

l->state = LOCKED;

enable_preemption();

lock_release(struct lock *)

l->state = UNLOCKED;

disable_preemption();

l->state = UNLOCKED;

enable_preemption();

Correct & uses proper blocking.

Note that thread doing the unlock performs the work of unblocking the first waiting thread.

Does this work?

Yes, this works – but is grossly inefficient. A thread that encounters the lock in the LOCKED state will busy wait until it is unlocked, needlessly using up CPU time.

Implementing Locks, Take 4

lock_acquire(struct lock *)

l->state = LOCKED;

disable_preemption();

while (l->state == LOCKED)

l->state = LOCKED;

disable_preemption();

l->state = LOCKED;

enable_preemption();

lock_release(struct lock *)

l->state = UNLOCKED;

disable_preemption();

l->state = UNLOCKED;

enable_preemption();

Using Locks

• Associate each shared variable with lock L

  "lock L protects that variable"

  • Example:

    static struct list usedlist; /* List of used blocks */
    static struct list freelist; /* List of free blocks */
    static struct lock listlock; /* Protects usedlist & freelist */

void *mem_alloc(…) |

b = alloc_block_from_freelist();

insert_into_usedlist(&usedlist, b); |

lock_acquire(&listlock);

lock_release(&listlock);

return b->data;

void mem_free(block *b) |

list_remove(&b->elem);

coalesce_into_freelist(&freelist, b);

lock_acquire(&listlock);

lock_release(&listlock);

return b->data;

How many locks should I use?

• Could use one lock for all shared variables

  – Disadvantage: if a thread holding the lock blocks, no other thread can access any shared variable, even unrelated ones

  – Sometimes used when retrofitting non-threaded code into threaded framework

  – Examples:

    • "KBL" Big Kernel Lock in Linux
    • fslock in Pintos Project 2

• Ideally, want fine-grained locking

  – One lock only protects one (or a small set of) variables – how to pick that set?
Multiple locks, the wrong way

```c
static struct list usedlist; /* List of used blocks */
static struct lock freelock; /* Protects deallocations */
static struct list freelist;  /* List of free blocks */
...

void *mem_alloc(...) {
   ...
    lock_acquire(&freelock);
    lock_acquire(&usedlock);
    insert_into_usedlist(&usedlist, b);
    return b->data;
}
```

Wrong: locks protect data structures, not code blocks! Allocating thread & deallocating thread could collide

Conclusion

- Choosing which lock should protect which shared variable(s) is not easy – must weigh:
  - Whether all variables are always accessed together (use one lock if so)
  - Whether code inside critical section can block (if not, no throughput gain on uniprocessor)
  - Whether there is a consistency requirement if multiple variables are accessed in related sequence (must hold single lock if so)
  - Cost of multiple calls to lock/unlock (gains may be offset by those costs)

Rules for easy locking

- Every shared variable must be protected by a lock
  - Acquire lock before touching (reading or writing) variable
  - Release when done, on all paths
  - One lock may protect more than one variable, but not too many
- If manipulating multiple variables, acquire locks protecting each
  - Acquire locks always in same order (doesn't matter which order, but must be same)
  - Release in opposite order
  - Don’t mix acquire & release (two-phase locking)