Race conditions

- Disk driver maintains a list of outstanding requests
- Each process can add requests to the list
struct list {
    int data;
    struct list *next;
};

... 

struct list *list = 0;
...

insert(int data) {
    struct list *l;

    l = malloc(sizeof *l);
    l->data = data;
    l->next = list;
    list = l;
}
Request queue (e.g. incoming network packets)

- Linked list, list is pointer to the first element
CPU1 allocates new request
CPU2 allocates new request
CPUs 1 and 2 update next pointer
CPU updates head pointer

list = l
CPU2 updates head pointer
State after the race

list = 1
List implementation with locks

```c
1 struct list {
2   int data;
3   struct list *next;
4 };
6 struct list *list = 0;
    struct lock listlock;
9 insert(int data)
10 {
11   struct list *l;
13   l = malloc(sizeof *l);
14     acquire(&listlock);
15     l->data = data;
16     l->next = list;
17     list = l;
18     release(&listlock);
19 }
```
void acquire(struct spinlock *lk) {
  for(;;) {
    if(!lk->locked) {
      lk->locked = 1;
      break;
    }
  }
}
Still incorrect

21 void
22 acquire(struct spinlock *lk)  
23 {  
24    for(;;) {  
25        if(!lk->locked) {  
26            lk->locked = 1;  
27            break;  
28        }  
29    }  
30 }  

- Two CPUs can reach line #25 at the same time  
  - See not locked, and  
  - Acquire the lock  
- Lines #25 and #26 need to be atomic  
  - I.e. indivisible
Compare and swap: xchg

- We switch between processes now
void acquire(struct spinlock *lk)
{
...

// The xchg is atomic.
// It also serializes, so that reads after acquire are not reordered before it.
while(xchg(&lk->locked, 1) != 0)
;

...
Compare and swap

static inline uint xchg(volatile uint *addr, uint newval) {
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
                   "+m" (*addr), "=a" (result) :
                   "1" (newval) :
                   "cc");

  return result;
}
Deadlocks

acquire(A) → acquire(B)

acquire(B) { 
  while(xchg(&B->locked, 1) != 0) 
}

acquire(A) { 
  while(xchg(&A->locked, 1) != 0) 
}
Lock ordering

• Locks need to be acquired in the same order
Locks and interrupts

```
network_packet()
    ....
    insert() {
        acquire(A)
    }
```

Network interrupt

```
network_packet()
    ....
    insert() {
        acquire(A)
    }
```
Locks and interrupts

• Never hold a lock with interrupts enabled
void acquire(struct spinlock *lk) {
    pushcli(); // disable interrupts to avoid deadlock.
    while(xchg(&lk->locked, 1) != 0) ;
    // The xchg is atomic.
    // It also serializes, so that reads after acquire are not reordered before it.
}

Disabling interrupts
Simple disable/enable is not enough

- If two locks are acquired
- Interrupts should be re-enabled only after the second lock is released

- Pushcli() uses a counter
void pushcli(void) {
    int eflags;
    eflags = readeflags();
    cli();
    if(cpu->ncli++ == 0)
        cpu->intena = eflags & FL_IF;
}
void popcli(void)
{
...

    if(--cpu->ncli < 0)
        panic("popcli");
    if(cpu->ncli == 0 && cpu->intena)
        sti();
}

popcli()
Problems with locks
Problems with locks

- Deadlock
  - Locks break modularity of interfaces, easy to get wrong
- Priority inversion
  - Low-priority task holds a lock required by a higher priority task
  - Priority inheritance can be a solution, but can also result in errors (see What really happened on Mars)
Problems with locks

- Convoying
  - Several tasks need the locks in roughly the same order
  - One slow task acquires the lock first
  - Everyone slows to the speed of this slow task

- Signal safety
  - Similar to interrupts, but for user processes
  - Can't be disabled, thus can't use locks
Problems with locks

- Kill safety
  - What if a task is killed or crashed while holding a lock?
- Preemption safety
  - What happens if a task is preempted while holding a lock?
Optimistic concurrency
Optimistic concurrency: main idea

• Instead of acquiring a lock try updating a data structure
  • When done, try committing changes
  • If there is a conflict, retry

• Similar to database transactions
Example: lock-free stack(), aka FIFO queue

class Node {
    Node * next;
    int data;
};

// ‘head of list’
Node * head;
Lock-free push()

```c
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}
```
bool pop(int& t) {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current)) {
            t = current->data;
            return true;
        }
    }
    return false;
}
The ABA problem

- The value of a variable is changed from A to B and then back to A
- In our example the variable is a pointer to a stack element
- What if the head gets deallocated with free(), and allocated again?
  - There is a good chance that head will have the same pointer value
    - Memory allocators often choose recently deallocated values
  - But really this is a different stack element
ABA example

Thread 1: pop()
read A from head
store A.next `somewhere`

cas with A succeeds

Thread 2:

pop()
Pops A, discards it
First element becomes B
pop(): pops B
push():
Memory manager recycles A to hold a new variable

cas with A succeeds
ABA workaround

- Keep an `update counter' along with a pointer
  - Needs a double word CAS
- Don't recycle memory too soon
Nontrivial lock-free data structures

• For example, a linked list
  • Much more complex
    – Operations on two pointers
  • Insert
    – What if predecessor is removed?
Thank you!