Recap: Race conditions

- Disk driver maintains a list of outstanding requests
- Each process can add requests to the list
Request queue (e.g. incoming network packets)

- Linked list, list is pointer to the first element
List implementation with locks

- Critical section

```c
9    insert(int data)
10   {
11      struct list *l;
13      l = malloc(sizeof *l);
   acquire(&listlock);
14      l->data = data;
15      l->next = list;
16      list = l;
   release(&listlock);
17   }
```
Xchg instruction

• Swap a word in memory with a new value
  • Atomic!
  • Return old value
Correct implementation

1573 void
1574 acquire(struct spinlock *lk)
1575 {
...
1580   // The xchg is atomic.
1581   while(xchg(&lk->locked, 1) != 0)
1582     ;
...
1592 }
insert(int data)
{
    struct list *l;
    l = malloc(sizeof *l);
    acquire(&listlock);
    l->data = data;
    l->next = list;
    list = l;
    release(&listlock);
}

One last detail...
void acquire(struct spinlock *lk) {
  ...
  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0) ;
  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section’s memory
  // references happen after the lock is acquired.
  __sync_synchronize();
  ...
}
Locks and interprocess communication
Send/receive queue

100 struct q {
101   void *ptr;
102 };;

103 void*
104 send(struct q *q, void *p)
105 {
106   while(q->ptr != 0)
107     ;
108   q->ptr = p;
109 }

110 void*
111 recv(struct q *q)
112 {
113   void *p;
114   void *p;
115   while((p = q->ptr) == 0)
116     ;
117   q->ptr = 0;
118   return p;
119 }

120 }

• Sends one pointer between two CPUs
Send/receive queue

```c
100 struct q {
101   void *ptr;
102 };  
103
104 void*
105 send(struct q *q, void *p)
106 {
107   while(q->ptr != 0)
108     ;
109   q->ptr = p;
110 }

112 void*
113 recv(struct q *q)
114 {
115   void *p;
116
117   while((p = q->ptr) == 0)
118     ;
119   q->ptr = 0;
120   return p;
121 }
```
Send/receive queue

```c
struct q {
    void *ptr;
};

void*
send(struct q *q, void *p)
{
    while(q->ptr != 0)
        ;
    q->ptr = p;
}

void*
recv(struct q *q)
{
    void *p;
    while((p = q->ptr) == 0)
        ;
    q->ptr = 0;
    return p;
}
```
Send/receive queue

100 struct q {
101   void *ptr;
102 }
103
104 void*
105 send(struct q *q, void *p)
106 {
107   while(q->ptr != 0)
108     ;
109   q->ptr = p;
110 }
112 void*
113 recv(struct q *q)
114 {
115   void *p;
116
117   while((p = q->ptr) == 0)
118     ;
119   q->ptr = 0;
120   return p;
121 }

• Works well, but expensive if communication is rare
• Receiver wastes CPU cycles
Sleep and wakeup

- **sleep(channel)**
  - Put calling process to sleep
  - Release CPU for other work

- **wakeup(channel)**
  - Wakes all processes sleeping on a channel
    - If any
  - i.e., causes sleep() calls to return
201 void*
202 send(struct q *q, void *p)
203 {
204   while(q->ptr != 0)
205     ;
206   q->ptr = p;
207   wakeup(q); /*wake recv*/
208 }

210 void*
211 recv(struct q *q)
212 {
213   void *p;
214   while((p = q->ptr) == 0)
215     sleep(q);
216   q->ptr = 0;
217   return p;
218 }
219 }
Send/receive queue

201 void*
202 send(struct q *q, void *p)
203 {
204   while(q->ptr != 0)
205     ;
206   q->ptr = p;
207   wakeup(q); /*wake recv*/
208 }

210 void*
211 recv(struct q *q)
212 {
213   void *p;
214   while((p = q->ptr) == 0)
215     sleep(q);
216   q->ptr = 0;
217   return p;
218 }

- recv() gives up the CPU to other processes
- But there is a problem...
Lost wakeup problem

recv

215

216

send

store p

206

207

204

205

215

test

sleep

wakeup

wait for wakeup forever

test

spin forever
struct q {
  struct spinlock lock;
  void *ptr;
};

void*
send(struct q *q, void *p)
{
  acquire(&q->lock);
  while(q->ptr != 0)
    ;
  q->ptr = p;
  wakeup(q);
  release(&q->lock);
}

Lock the queue

void*
recv(struct q *q)
{
  void *p;
  acquire(&q->lock);
  while((p = q->ptr) == 0)
    sleep(q);
  q->ptr = 0;
  release(&q->lock);
  return p;
}
• Doesn't work either: deadlocks
  • Holds a lock while sleeping
struct q {
  struct spinlock lock;
  void *ptr;
};

void*
send(struct q *q, void *p)
{
  acquire(&q->lock);
  while(q->ptr != 0)
    ;
  q->ptr = p;
  wakeup(q);
  release(&q->lock);
}

void *
recv(struct q *q)
{
  void *p;
  acquire(&q->lock);
  while((p = q->ptr) == 0)
    sleep(q, &q->lock);
  q->ptr = 0;
  release(&q->lock);
  return p;
}
sleep(void *chan, struct spinlock *lk) {
...

2823 if(lk != &ptable.lock){
2824   acquire(&ptable.lock);
2825   release(lk);
2826 }
2827
2828 // Go to sleep.
2829 proc->chan = chan;
2830 proc->state = SLEEPING;
2831 sched();
...

2836 // Reacquire original lock.
2837 if(lk != &ptable.lock){
2838   release(&ptable.lock);
2839   acquire(lk);
2840 }
2841 }

sleep()

• Acquire ptable.lock
• All process operations are protected with ptable.lock
sleep(void *chan, struct spinlock *lk)
{
  ...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();
  ...
  // Reacquire original lock.
  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
}

sleep()

- Acquire ptable.lock
- All process operations are protected with ptable.lock
- Release lk
- Why is it safe?
sleep(void *chan, struct spinlock *lk)
{
...
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }

  // Go to sleep.
  proc−>chan = chan;
  proc−>state = SLEEPING;
  sched();
  ...

  // Reacquire original lock.
  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
}

sleep()

- Acquire ptable.lock
- All process operations are protected with ptable.lock
- Release lk
- Why is it safe?
- Even if new wakeup starts at this point, it cannot proceed
- Sleep() holds ptable.lock
wakeup1(void *chan)
{
    struct proc *p;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;

    acquire(&ptable.lock);
    wakeup1(chan);
    release(&ptable.lock);
}

wakeup(void *chan)
{
    acquire(&ptable.lock);
    wakeup1(chan);
    release(&ptable.lock);
}
Pipes
#define PIPESIZE 512

struct pipe {
    struct spinlock lock;
    char data[PIPESIZE];
    uint nread; // number of bytes read
    uint nwrite; // number of bytes written
    int readopen; // read fd is still open
    int writeopen; // write fd is still open
};
Pipe buffer

- Buffer full
  \[p->nwrite == p->nread + \text{PIPESIZE}\]
- Buffer empty
  \[p->nwrite == p->nread\]

```c
struct pipe {
    nread
    nwrite
};
```

Free slots

Unconsumed data
pipewrite()
piperead(struct pipe *p, char *addr, int n)
{
    int i;

    acquire(&p->lock);
    while(p->nread == p->nwrite && p->writeopen){
        if(proc->killed){
            release(&p->lock);
            return -1;
        }
        sleep(&p->nread, &p->lock);
    }

    for(i = 0; i < n; i++){
        if(p->nread == p->nwrite)
            break;
        addr[i] = p->data[p->nread++ % PIPESIZE];
    }

    wakeup(&p->nwrite);
    release(&p->lock);
    return i;
}
Thank you!