238P: Operating Systems

Lecture 11: Synchronization

Anton Burtsev
May, 2019
Starting other CPUs
main(void)
{
...

startothers(); // start other processors
kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
userinit(); // first user process
mpmain();
}
Starting other CPUs

- Copy start code in a good location
  - 0x7000 (remember same as the one used by boot loader)
- Pass start parameters on the stack
  - Allocate a new stack for each CPU
  - Send a magic inter-processor interrupt (IPI) with the entry point (mpenter())
Start other CPUs

startothers(void)
{
  code = P2V(0x7000);
  memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == cpus+cpunum()) // We’ve started already.
      continue;
    stack = kalloc();
    *(void**)(code−4) = stack + KSTACKSIZE;
    *(void**)(code−8) = mpenter;
    *(int**)(code−12) = (void *) V2P(entrypgdir);
    lapicstartap(c−>apicid, V2P(code));
}
startothers(void)
{
    code = P2V(0x7000);
    memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);

    for(c = cpus; c < cpus+ncpu; c++){
        if(c == cpus+cpunum()) // We’ve started already.
            continue;
        ...
        stack = kalloc();
        *(void**)(code-4) = stack + KSTACKSIZE;
        *(void**)(code-8) = mpenter;
        *(int**)(code-12) = (void *) V2P(entrypgdir);
        lapicstartap(c->apicid, V2P(code));
    }
}
Start other CPUs

- Allocate a new kernel stack for each CPU
- What will be running on this stack?
  - Scheduler

```c
startothers(void)
{
  code = P2V(0x7000);
  memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == cpus+cpunum()) // We’ve started already.
      continue;
    ...

    stack = kalloc();
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void**)(code-8) = mpenter;
    *(int**)(code-12) = (void *) V2P(entrypgdir);
    lapicstartap(c->apicid, V2P(code));
}
```
```c
1374 startothers(void)
1375 {
1384   code = P2V(0x7000);
1385   memmove(code, _binary_entryother_start, (uint)_binary_entryother_size);
1386
1387   for(c = cpus; c < cpus+ncpu; c++){
1388     if(c == cpus+cpunum()) // We’ve started already.
1389       continue;
1389+  continue;
1394+  stack = kalloc();
1395+   *(void**)(code-4) = stack + KSTACKSIZE;
1396+   *(void**)(code-8) = mpenter;
1397+   *(int**)(code-12) = (void *) V2P(entrypgdir);
1398
1399   lapicstartap(c->apicid, V2P(code));
```

Start other CPUs

- What is done here?
startothers(void)
{
    code = P2V(0x7000);
    memmove(code, _binary_entryother_start,
            (uint)_binary_entryother_size);

    for(c = cpus; c < cpus+ncpu; c++){
        if(c == cpus+cpunum()) // We’ve started already.
            continue;
        stack = kalloc();
        *(void**)(code-4) = stack + KSTACKSIZE;
        *(void**)(code-8) = mpenter;
        *(int**)(code-12) = (void *) V2P(entrypgdir);
        lapicstartap(c->apicid, V2P(code));
    }
}
Start other CPUs

- Send “magic” interrupt
- Wake up other CPUs

```c
1374 startothers(void) {
1375   code = P2V(0x7000);
1376   memmove(code, _binary_entryother_start,
1377            (uint)_binary_entryother_size);
1378
1379   for(c = cpus; c < cpus+ncpu; c++){
1380     if(c == cpus+cpunum()) // We’ve started already.
1381       continue;
1382     stack = kalloc();
1383     *(void**)(code−4) = stack + KSTACKSIZE;
1384     *(void**)(code−8) = mpenter;
1385     *(int**)(code−12) = (void *) V2P(entrypgdir);
1386     lapicstartap(c−>apicid, V2P(code));
```
entryother.S

- Disable interrupts
- Init segments with 0
entryother.S

• Load GDT
• Switch to 32bit mode
  • Long jump to start32
• Load segments
# Turn on page size extension for 4Mbyte pages
movl %cr4, %eax
orl $(CR4_PSE), %eax
movl %eax, %cr4

# Use enterpgdir as our initial page table
movl (start-12), %eax
movl %eax, %cr3

# Turn on paging.
movl %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl %eax, %cr0

# Switch to the stack allocated by startothers()
movl (start-4), %esp

# Call mpenter()
call *(start-8)
# Turn on page size extension for 4Mbyte pages
movl %cr4, %eax
orl $(CR4_PSE), %eax
movl %eax, %cr4

# Use enterpgdir as our initial page table
movl (start-12), %eax
movl %eax, %cr3

# Turn on paging.
movl %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl %eax, %cr0

# Switch to the stack allocated by startothers()
movl (start-4), %esp

# Call mpenter()
call *(start-8)
# Turn on page size extension for 4Mbyte pages
movl %cr4, %eax
orl $(CR4_PSE), %eax
movl %eax, %cr4

# Use enterpgdir as our initial page table
movl (start-12), %eax
movl %eax, %cr3

# Turn on paging.
movl %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl %eax, %cr0

# Switch to the stack allocated by startothers()
movl (start-4), %esp

# Call mpenter()
call *(start-8)
# Turn on page size extension for 4Mbyte pages
movl %cr4, %eax
orl $(CR4_PSE), %eax
movl %eax, %cr4

# Use enterpgdir as our initial page table
movl (start-12), %eax
movl %eax, %cr3

# Turn on paging.
movl %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl %eax, %cr0

# Switch to the stack allocated by startothers()
movl (start-4), %esp

# Call mpenter()
call *(start-8)
1251 static void 
1252 mpenter(void)
1253 {
1254   switchkvm();
1255   seginit();
1256   lapicinit();
1257   mpmain();
1258 }
static void mpenter(void) {
  switchkvm();
  seginit();
  lapicinit();
  mpmain();
}
seginit(void)
{
    struct cpu *c;

    // Map "logical" addresses to virtual addresses using identity map. 
    // Cannot share a CODE descriptor for both kernel and user 
    // because it would have to have DPL_USR, but the CPU forbids 
    // an interrupt from CPL=0 to DPL=3.
    c = &cpus[cpuid()];
    c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
    c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
    c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
    c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
    lgdt(c->gdt, sizeof(c->gdt));
}

Init segments
Per-CPU variables

- Variables private to each CPU
Per-CPU variables

- Variables private to each CPU
  - Current running process
  - Kernel stack for interrupts
    - Hence, TSS that stores that stack

struct cpu cpus[NCPU];
// Per-CPU state

struct cpu {
    uchar apicid; // Local APIC ID
    struct context *scheduler; // swtch() here to enter scheduler
    struct taskstate ts; // Used by x86 to find stack for interrupt
    struct segdesc gdt[NSEGS]; // x86 global descriptor table
    volatile uint started; // Has the CPU started?
    int ncli; // Depth of pushcli nesting.
    int intena; // Were interrupts enabled before pushcli?
    struct proc *proc; // The process running on this cpu or null
};

extern struct cpu cpus[NCPU];
// Must be called with interrupts disabled
int cpuid() {
    return mycpu()-cpus;
}

struct cpu* mycpu(void)
{
    int apicid, i;

    if(readeflags()&FL_IF)
        panic("mycpu called with interrupts enabled\n");

    apicid = lapicid();
    // APIC IDs are not guaranteed to be contiguous. Maybe we should have
    // a reverse map, or reserve a register to store &cpus[i].
    for (i = 0; i < ncpu; ++i) {
        if (cpus[i].apicid == apicid)
            return &cpus[i];
    }
    panic("unknown apicid\n");
}
// Common CPU setup code.

static void mpmain(void)
{

cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
idtinit();    // load idt register
xchg((&(mycpu())->started), 1);    // tell startothers() we’re up
scheduler();    // start running processes
}
Synchronization
Race conditions

• Example:
  • Disk driver maintains a list of outstanding requests
  • Each process can add requests to the list
```c
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;
}
```

### List implementation (no locks)

- **List**
  - One data element
  - Pointer to the next element
List implementation (no locks)

- Global head

```c
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;
}
```
List implementation (no locks)

- Insertion
  - Allocate new list element

```c
1 struct list {
2   int data;
3   struct list *next;
4 }
...
6 struct list *list = 0;
...
9 insert(int data)
10 {
11   struct list *l;
12
13   l = malloc(sizeof *l);
14   l->data = data;
15   l->next = list;
16   list = l;
17 }
```
List implementation
(no locks)

- Insertion
  - Allocate new list element
  - Save data into that element
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;
}
Now what happens when two CPUs access the same list
Request queue (e.g. pending disk requests)

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

```
list = l
```

![Diagram showing list and CPU connections]
CPU2 updates head pointer

```
list = l
```

Diagram showing a list structure with an update from CPU2.
State after the race (red element is lost)
Mutual exclusion

- Only one CPU can update list at a time
List implementation with locks

- Critical section
• How can we implement acquire()?
21 void
22 acquire(struct spinlock *lk)
23 {
24   for(;;) {
25     if(!lk->locked) {
26       lk->locked = 1;
27       break;
28     }
29   }
30 }
void acquire(struct spinlock *lk)
{
    for(;;) {
        if(!lk->locked) {
            lk->locked = 1;
            break;
        }
    }
}

- 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

- Swap a word in memory with a new value
  - Return old value
void acquire(struct spinlock *lk) {

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0) ;

  ...
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;
}
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();
...
}
Deadlocks
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_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.
    if(holding(lk))
        panic("acquire");
    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;
    __sync_synchronize();

    // 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
pushcli(void) {
  int eflags;
  eflags = readeflags();
  cli();
  if(cpu->ncli == 0) cpu->intena = eflags & FL_IF;
  cpu->ncli += 1;
}
popcli(void)
{
  if(readeflags() & FL_IF)
    panic("popcli − interruptible");
  if(--cpu->ncli < 0)
    panic("popcli");
  if(cpu->ncli == 0 && cpu->intena)
    sti();
}
Locks and interprocess communication
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;
}
```

- Sends one pointer between two CPUs
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*send(struct q *q, void *p)
105 {
106   while(q->ptr != 0)
107     ;
108   q->ptr = p;
109 }  
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
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   while((p = q->ptr) == 0)
117     ;
118   q->ptr = 0;
119   return p;
120 }
```

- 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
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  }
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
215   while((p = q->ptr) == 0)
216     sleep(q);
217   q->ptr = 0;
218   return p;
219 }

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

recv

215 test
216 sleep wait for wakeup forever

send

206 store p 207 wakeup
204 test 205 spin forever

Time
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->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)
{
...

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
```c
2809 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**
- **Release lk**
- **Why is it safe?**
2809 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
- **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;
}

wakeup(void *chan) {
    acquire(&ptable.lock);
    wakeup1(chan);
    release(&ptable.lock);
}
Pipes
Pipe

#define PIPESIZE 512

struct pipe {
    struct spinlock lock;
    char data[PIPE SIZE];
    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

```c
#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**
  \[ p->nwrite = p->nread + PIPESIZE \]

- **Buffer empty**
  \[ p->nwrite = p->nread \]
Pipe buffer

- Buffer full
  \[ p->nwrite = p->nread + PIPESIZE \]

- Buffer empty
  \[ p->nwrite = p->nread \]

```
struct pipe {
    nread
    nwrite
}
```

Free slots

Unconsumed data
Pipe buffer

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

- **Buffer empty**
  \[ p->nwrite == p->nread \]

```
struct pipe {
    nread
    nwrite
}
```

Diagram:
- `piperead()`
- `pipewrite()`
- Unconsumed data
- Free slots
- Buffers occupied
- Buffer size: 512
Pipe buffer

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

- Buffer empty
  \[ p->nwrite == p->nread \]

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

```
piperead()
```

```
pipewrite()
```

```
Free slots
```

```
Unconsumed data
```
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;
}
piperead(){}

- If the buffer is empty && the write end is still open
- Go to sleep

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;
}
piperead

- After reading some data from the buffer
- Wakeup the writer
pipewrite()

- If the buffer is full
- Wakeup reader
- Go to sleep
pipewrite() (6530-6549)

- If the buffer is full
- Wakeup reader
- Go to sleep
- However if the read end is closed
- Return an error
- (-1)
pipewrite()

- Otherwise keep writing bytes into the pipe
- When done
  - Wakeup reader

```c
int i;

acquire(&p->lock);
for(i = 0; i < n; i++){
  while(p->nwrite == p->nread + PIPESIZE){
    if(p->readopen == 0 || proc->killed){
      release(&p->lock);
      return -1;
    }
    wakeup(&p->nwrite);
    sleep(&p->nwrite, &p->lock);
  }
  p->data[p->nwrite++ % PIPESIZE] = addr[i];
}
wakeup(&p->nread);
release(&p->lock);
return n;
```
Thank you!