Operating Systems

Lecture: Synchronization

Anton Burtsev
Starting other CPUs
1317 main(void)
1318 {
...
1336   startothers(); // start other processors
1337   kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
1338   userinit(); // first user process
1339   mpmain();
1340 }
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())
```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));
    }
}
```

Start other CPUs

- Copy start code to 0x7000
- Start code is linked into the kernel
  - `_binary_entryother_start`
  - `_binary_entryother_size`
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

- Allocate a new kernel stack for each CPU

- What will be running on this stack?
  - Scheduler

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;
1394     stack = kalloc();
1395     *(void**)(code-4) = stack + KSTACKSIZE;
1396     *(void**)(code-8) = mpenter;
1397     *(int**)(code-12) = (void *) V2P(entrypgdir);
1399     lapicstartap(c->apicid, V2P(code));
Start other CPUs

- What is done here?
```c
1374 startothers(void) {
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;
1390   }
...
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));
```
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

- Send “magic” interrupt
- Wake up other CPUs
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
1162 movl %cr4, %eax
1163 orl $(CR4_PSE), %eax
1164 movl %eax, %cr4

# Use enterpgdir as our initial page table
1166 movl (start−12), %eax
1167 movl %eax, %cr3

# Turn on paging.
1169 movl %cr0, %eax
1171 orl $(CR0_PE|CR0_PG|CR0_WP), %eax
1172 movl %eax, %cr0
1173

# Switch to the stack allocated by startothers()
1174 movl (start−4), %esp

# Call mpenter()
1176 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
1163     movl %cr4, %eax
1164     orl $(CR4_PSE), %eax
1165     movl %eax, %cr4

# Use enterpgdir as our initial page table
1166     movl (start-12), %eax
1167     movl %eax, %cr3

# Turn on paging.
1169     movl %cr0, %eax
1171     orl $(CR0_PE|CR0_PG|CR0_WP), %eax
1172     movl %eax, %cr0

# Switch to the stack allocated by startothers()
1175     movl (start-4), %esp

# Call mpenter()
1177     call *(start-8)
static void mpenter(void)
{
  switchkvm();
  seginit();
  lapicinit();
  mpmain();
}
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

```c
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

}
How CPUs access memory?
Detour: Cache-coherence and memory hierarchy
Synchronization
Race conditions

• Example:
  • Disk driver maintains a list of outstanding requests
  • Each process can add requests to the list
List implementation (no locks)

- List
  - One data element
  - Pointer to the next 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 }
```
```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)

- Global head
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 }
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;
}
```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
  - Save data into that element
  - Insert into the list
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

l->next = list

l->next = list
CPU1 updates head pointer

list = 1
CPU2 updates head pointer

list = 1

Diagram: A diagram showing the update of a head pointer in a linked list. The diagram includes a CPU, a list, and elements with pointers.
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()?
Spinlock

• Spin until lock is 0
• Set it to 1

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

Still incorrect

- 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
Correct implementation

1573 void
1574 acquire(struct spinlock *lk)
1575 {
  ...
  // The xchg is atomic.
  1580   while(xchg(&lk->locked, 1) != 0)
  1581     ;
  ...
1592 }
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;
}
1573 void
1574 acquire(struct spinlock *lk)
1575 {

... 
1580   // The xchg is atomic.
1581   while(xchg(&lk->locked, 1) != 0)
1582     ;
1584   // Tell the C compiler and the processor to not move loads or stores
1585   // past this point, to ensure that the critical section’s memory
1586   // references happen after the lock is acquired.
1587   __sync_synchronize();

... 
1592 }
Deadlocks
Deadlocks

acquire(A)

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

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

acquire(B)
Lock ordering

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

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

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

Network interrupt
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

• Sends one pointer between two CPUs
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   while((p = q->ptr) == 0)
117     ;
118   q->ptr = 0;
119   return p;
120 }
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

```
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;
}
```

- 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 }
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 
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

Time

send

206 store p

207 wakeup

204 test

205 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);
}

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
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;
}

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

struct pipe {
    struct spinlock lock;
    char data[PIPSIZE];
    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
};
```
#define PIPESIZE 512

struct pipe {
    struct spinlock lock;
    char data[PIPEDSIZE];
    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**
  \[ p->nwrite = p->nread + \text{PIPSIZE} \]

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

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

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

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

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

- 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
}
```

Diagram:
- `piperead()`
- `pipewrite()`
- Free slots
- Unconsumed data
- Buffer full\[ p->nwrite = p->nread + PIPESIZE \]
- Buffer empty\[ p->nwrite = p->nread \]
piperead()

- Acquire pipe lock
- All pipe operations are protected with the lock
piperead()

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

- After reading some data from the buffer
- Wakeup the writer

```c
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;
}
```
pipewrite()

- If the buffer is full
- Wakeup reader
- Go to sleep

6530 pipewrite(struct pipe *p, char *addr, int n)
6531 {
6532   int i;
6533
6534   acquire(&p−>lock);
6535   for(i = 0; i < n; i++){
6536     while(p−>nwrite == p−>nread + PIPESIZE){
6537       if(p−>readopen == 0 || proc−>killed){
6538         release(&p−>lock);
6539         return −1;
6540       }
6541       wakeup(&p−>nread);
6542       sleep(&p−>nwrite, &p−>lock);
6543     }
6544     p−>data[p−>nwrite++ % PIPESIZE] = addr[i];
6545   }
6546   wakeup(&p−>nread);
6547   release(&p−>lock);
6548   return n;
6549 }
pipewrite()

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

```c
pipewrite(struct pipe *p, char *addr, int n)
{
    int i;

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

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