Scheduling

Youjip Won

KAIST EE
outline

- sharing CPU
- swtch
- sched
- switch
- sleep and wakeup
Start scheduler (Cont.)

- scheduler() is started in mpmain().

```c
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
}
```
scheduler()

- The scheduler loops over the process table looking for a runnable process.

- Once it finds a process
  - it sets the per-CPU current process variable proc.
  - switches to the process’s page table with switchuvm().
  - marks the process as RUNNING.
  - and then calls swtch() to start running it.
void scheduler(void) {
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for (;;) {
        // Enable interrupts on this processor.
        sti();

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            ...
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;

            swtch(&(c->scheduler), p->context);
            switchkvm();

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            c->proc = 0;
        }

        ...
    }
}
void scheduler(void) {
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        // Enable interrupts on this processor.
        sti();

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;
            ...

            swtch(&(c->scheduler), p->context);
            switchkvm();
            ...
        }
        release(&ptable.lock);
    }
}
Interrupt in the `scheduler()`

- **Interrupt**
  - is enabled in the outer loop.
  - if the scheduler left interrupts disabled all the time, the I/O would never arrive.

- **ptable lock**
  - is released at the end of each iteration.
  - If an idling scheduler on one CPU is looping with the lock held,
    - no other CPU cannot perform a context switch or any process-related system call.
    - can never mark a process as **RUNNABLE** so as to break the idling CPU out of its scheduling loop.
void yield(void)
{
    acquire(&ptable.lock);
    myproc()->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}

void sleep(void *chan, struct spinlock){
    ...
    sched();
    p->chan = 0;
    if(lk != &ptable.lock){
        release(&ptable.lock);
        acquire(lk);
    }
}
ptable.lock

- lock in one thread (often in yield) and releases the lock in another thread.
- Once scheduler starts to convert a runnable process to **RUNNING**, the lock cannot be released until the kernel thread completes running (after the **switch**, e.g. in **yield**).
mycpu() and myproc()

- struct cpu
  - contains per-processor state: the currently running process, hardware id for that processor (apicid).
  - When a processor must find its per-cpu state, it reads its identifier from its local APIC and uses that identifier to find its state in the array.

// 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
};
mycpu() scans array of a struct CPU and returns the address of struct cpu

inefficient!!!

```c
struct cpu* mycpu(void) {
    int apicid, i;
    if(readeflags() & FL_IF)
        panic("mycpu called with interrupts enabled\n");

    apicid = lapicid();

    for (i = 0; i < ncpu; ++i) {
        if (cpus[i].apicid == apicid)
            return &cpus[i];
    }
    panic("unknown apicid\n");
}
```
myproc()

- **find the struct proc** for the process that is running on the current processor.
- **myproc()** disables interrupts, and invokes mycpu().

```
struct proc* myproc(void) {
    struct cpu *c;
    struct proc *p;
    pushcli();
    c = mycpu();
    p = c->proc;
    popcli();
    return p;
}
```
sleep and wakeup

- Let the processes to interact with each other!
- Sleep and wakeup allows one process to sleep waiting for an event and another process to wake it up once the event has happened.
- Sleep and wakeup are often called sequence coordination or conditional synchronization mechanisms.
- Make sure that they do not miss each other!!!
### producer/consumer with busy waiting

**Operation**
- **send**: loops until the queue is empty and then puts the pointer `p` in the queue.
- **recv**: loops until the queue is non-empty and takes the pointer out.

**Problem: waste of CPU**
- If the sender sends rarely, the receiver will spend most of its time spinning (busy).

```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 }
```

```c
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 }
```
**Problem: lost wakeup**

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

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

![Diagram showing the timeline of recv and send functions with test and sleep scenarios.](image-url)
Incorrect solution to lost wakup: deadlock

While receiver is waiting, the send cannot send.

```
300     struct q {
301         struct spinlock lock;
302         void *ptr;
303     };
304
305     void*
306     send(struct q *q, void *p)
307     {
308         acquire(&q->lock);  
309         while(q->ptr != 0)  
310             ;
311         q->ptr = p;
312         wakeup(q);
313         release(&q->lock);
314     }
315
316     void*
317     recv(struct q *q)
318     {
319         void *p;
320
321         acquire(&q->lock);
322         while((p = q->ptr) == 0)
323             sleep(q);  
324         q->ptr = 0;
325         release(&q->lock);
326         return p;
327     }
```
producer/consumer with sleep and wakeup

within sleep, release the lock before it sets the process to sleep.

400 struct q {  
401    struct spinlock lock;  
402    void *ptr;  
403 };  
404  
405 void*
406 send(struct q *q, void *p)  
407 {  
408    acquire(&q->lock);  
409    while(q->ptr != 0)  
410        ;  
411    q->ptr = p;  
412    wakeup(q);  
413    release(&q->lock);  
414 }

415 void*
416 recv(struct q *q)  
417 {  
418    void *p;  
419    acquire(&q->lock);  
420    while((p = q->ptr) == 0)  
421        sleep(q, &q->lock);  
422    q->ptr = 0;  
423    release(&q->lock);  
424    return p;  
425 }

407 void send(struct q *q, void *p) {  
408    acquire(&q->lock);  
409    while(q->ptr != 0)  
410        ;  
411    q->ptr = p;  
412    wakeup(q);  
413    release(&q->lock);  
414 }  
415 void recv(struct q *q) {  
416    void *p;  
417    acquire(&q->lock);  
418    while((p = q->ptr) == 0)  
419        sleep(q, &q->lock);  
420    q->ptr = 0;  
421    release(&q->lock);  
422    return p;  
423 }
Code: sleep

- sleep: mark the current process as SLEEPING and then call `sched()` to release the processor.
2873 void
2874 sleep(void *chan, struct spinlock *lk) {
2876     struct proc *p = myproc();
2877
2878     if(p == 0)
2879         panic("sleep");
2881     if(lk == 0)
2882         panic("sleep without lk");
2890     if(lk != &ptable.lock){
2891         acquire(&ptable.lock);
2892         release(lk);//release the lock.
2893     }
2895     p->chan = chan;
2896     p->state = SLEEPING;
2898     sched();
2901     p->chan = 0;
2904     if(lk != &ptable.lock){
2905         release(&ptable.lock);
2906         acquire(lk);//acquire the lock.
2907     }
2908 }

Hand CPU over.

Get back from sleep and wake up.
wakeup looks for a process sleeping on the given wait channel and marks it as RUNNABLE.

```
2962 // Wake up all processes sleeping on chan.
2963 void
2964    wakeup(void *chan)
2965 {
2966       acquire(&ptable.lock);
2967       wakeup1(chan);
2968       release(&ptable.lock);
2969   }
```
2950 // Wake up all processes sleeping on chan.
2951 // The ptable lock must be held.
2952 static void
2953 wakeup1(void *chan)
2954 {
2955      struct proc *p;
2956
2957      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2958          if(p->state == SLEEPING && p->chan == chan)
2959              p->state = RUNNABLE;
2960 }

- Put all threads waiting for the channel to **RUNNABLE**.
- Inefficient: O(n)
**summary**

- **switch**: It switch the process to another process for running on the CPU.
- **sched()**: Release the CPU and switch to scheduler thread.
- **schedule()**: Called by Independent thread by each CPU, switch to new RUNNABLE thread
- **sleep** and **wakeup**