Scheduling

Youjip Won

KAIST EE
outline

- sharing CPU
- swtch
- sched
- switch
- sleep and wakeup
Let's share CPU.

In Reality

What you think. 😅
Multiplexing

- number of processes >> number of processors
- multiplexing the processes onto the hardware processors

Way to release CPU

- Releasing by self: voluntary context switch
  - Based on sleep & wakeup
  - waits for device or pipe IO to complete
  - waits for a child to exit

- Releasing involuntarily: involuntary context switch
  - Based on timer interrupt
  - Multiplexing by scheduler
Issues in CPU multiplexing

1. context switch mechanism: Which informations are saved and restored during switching?
2. How to do it transparently: timer interrupt.
3. Avoid race condition: Many CPU’s perform context switches concurrently. How to avoid race condition when multiple processors are switching processes?
4. Release resources: How can exiting process release its resources?
5. Maintain information on current processes
Switching the processes

- Every process has its own kernel stack and register set.
- Each CPU has a separate scheduler thread.
- Switching from one thread to another
- Saving the old thread’s CPU registers
- Restoring the previously-saved registers of the new thread.

```
sched()
swtch
context
```
switching the processes

% cat

User Space

Kernel Space

shell

cat

① save

Kstack(shell) Kstack(scheduler) Kstack(cat)

② swtch() ③ swtch() ④ restore

Context switch

mode switch

system call
interrupt
trap

Youjip Won
switching the processes

1. stack switch from user to kernel: old process
2. A context switch to the local CPU’s scheduler thread.
3. A context switch to a new process’s kernel thread.
4. stack switch from kernel to user: new process (A trap return)
separate scheduler thread in xv6

- Xv6 scheduler runs on its own thread.
- The process switch accompanies two context switch.

why?
- simplify the procedure of cleaning up user processes.
- exit
- kill
**struct context**

- `swtch()` saves and restores the register sets, called **contexts**.
- When it is time for a process to give up the CPU, the process’s kernel thread calls `swtch()` **to save its own context and return to the scheduler context**.
- `struct context*`. It points to a structure stored on the **kernel stack** involved.

```c
struct context {
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
};
```
swtch(void **old, void* new)
swtch(struct context **old, struct context *new)

- pushes the current CPU register onto the kernel stack and saves the stack pointer in *old.
- copies new to %esp, pops previously saved registers, and returns.
Switch ()

Coroutine

Process A

Scheduler

Process B

Switch (A, Scheduler)

Switch (Scheduler, B)

Switch (Scheduler, A)

Switch (B, scheduler);
swtch(struct context **old, struct context *new)

# load arguments to %eax and %edx
movl 4(%esp), %eax    //old
movl 8(%esp), %edx    //new

# Pushes the registers (Save context of old)
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stack.
movl %esp, (%eax) //Saves %esp at *old
movl %edx, %esp   //Restore %esp from new

# Load registers (Restore context of new)
popl %edi
popl %esi
popl %ebx
popl %ebp

# make return address as %eip
ret
swtch(struct context **old, struct context *new)

# load arguments to %eax and %edx
movl 4(%esp), %eax  //old
movl 8(%esp), %edx  //new

Kernel stack of old thread
Kernel stack of new thread

**old

%esp
ret addr

*new

%ebp
%ebx
%esi
%esi

%eax %edx

*new

**old

ret addr

%ebp
%ebx
%esi
%esi
swtch(struct context **old, struct context *new)

# load arguments to %eax and %edx
movl 4(%esp), %eax    //old
movl 8(%esp), %edx    //new

# Pushes the registers (Save context of old)
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stack.
movl %esp, (%eax) //Saves %esp at *old
movl %edx, %esp    //Restore %esp from new

# Load registers (Restore context of new)
popl %edi
popp %esi
popp %ebx
popp %ebp

# make return address as %eip
ret
swtch(struct context **old, struct context *new)

# load arguments to %eax and %edx
movl 4(%esp), %eax    //old
movl 8(%esp), %edx    //new

# Pushes the registers (Save context of old)
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stack.
movl %esp, (%eax) //Saves %esp at *old
movl %edx, %esp   //Restore %esp from new

# Load registers (Restore context of new)
popl %edi
popl %esi
popl %ebx
popl %ebp

# make return address as %eip
ret
swtch(struct context **old, struct context *new)

# load arguments to %eax and %edx
movl 4(%esp), %eax    //old
movl 8(%esp), %edx    //new

# Pushes the registers (Save context of old)
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stack.
movl %esp, (%eax) //Saves %esp at *old
movl %edx, %esp   //Restore %esp from new

# Load registers (Restore context of new)
popl %edi
popl %esi
popl %ebx
popl %ebp

# make return address as %eip
ret
swtch(struct context **old, struct context *new)

# load arguments to %eax and %edx
movl 4(%esp), %eax    // old
movl 8(%esp), %edx    // new

# Pushes the registers (Save context of old)
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stack.
movl %esp, (%eax)    // Saves %esp at *old
movl %edx, %esp      // Restore %esp from new

# Load registers (Restore context of new)
popl %edi
popl %esi
popl %ebx
popl %ebp

# make return address as %eip
ret

It is returned to sched() or scheduler()
The function to release the CPU

```
sched()
```

Cases to release the CPU

- Determined by process’s action: `yield()`
- Exit of process: `exit()`
- Wait for something such as IO completion or pipe IO: `sleep()`

Actually doing

- Switch to `scheduler()`
Releasing the CPU

A process that wants to give up the CPU must

- acquire the process table lock `ptable.lock`
- release any other locks it is holding
- update its own state (`proc->state`) to `RUNNABLE`
- and then call `sched()`.

`yield()`, `sleep()` and `exit()` follow this convention.
yield()

// Give up the CPU for one scheduling round.
void yield(void) {
    acquire(&ptable.lock);  //DOC: yieldlock
    myproc()->state = Runnable;
    sched();
    release(&ptable.lock);
}
A thread in execution can call `sched()` in three cases:
- `exit`
- `yield`
- `sleep`
sched()  

- hand the control over to the scheduler function.
- be sure that interrupts are disabled.
- save the current context in `proc->context` and switch to the scheduler context in `cpu->scheduler`. 
void sched(void) {
    int intena;
    struct proc *p = myproc();

    if(!holding(&ptable.lock)) // make sure the ptable is locked.
        panic("sched ptable.lock");
    if(mycpu()->ncli != 1) // make sure interrupt is disabled.
        panic("sched locks");
    if(p->state == RUNNING) // sleep, yield, exit
        panic("sched running");
    if(readeflags()&FL_IF)
        panic("sched interruptible");
    intena = mycpu()->intena;
    swtch(&p->context, mycpu()->scheduler);
    mycpu()->intena = intena;
}
scheduler()

- Independent thread by each CPU
- Select CPU to run and switch to the thread
void scheduler(void) {
    ...
    for(;;) {
        ...
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if(p->state != RUNNABLE)
                continue;
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;
            swtch(&(c->scheduler), p->context);
            switchkvm();
            ...
        }
    }
}
Switching the address space

- `scheduler()` is responsible for switching the address space.
- When the scheduler() switches to the user process,
- Select the process to run. → Address space switch (switchuvm()) → Call `swtch()`
- Once it has returned from `swtch()` → address space switch (switchkvm())
// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
    if (p == 0)
        panic("switchuvm: no process");
    if (p->kstack == 0)
        panic("switchuvm: no kstack");
    if (p->pgdir == 0)
        panic("switchuvm: no pgdir");

    pushcli();
    mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
        sizeof(mycpu()->ts)-1, 0);
    mycpu()->gdt[SEG_TSS].s = 0;
    mycpu()->ts.ss0 = SEG_KDATA << 3;
    mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
    // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
    // forbids I/O instructions (e.g., inb and outb) from user space
    mycpu()->ts.iomb = (ushort)0xFFFF;
lcr3(V2P(p->pgdir)); // switch to process's address space
    popcli();
}

switchkvm(void)
{
lcr3(V2P(kpgdir)); // switch to the kernel page table
}

// Switch TSS and h/w page table to correspond to process p.
void scheduler(void) {
    ... for(;;){
        ... for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;

            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;
        }
    
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

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

    ...}
}
acquire & release for ptable.lock

```c
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);
    }
}
```

Holding ptable.lock
Managing `ptable.lock`

- Generally, the thread that has acquired a lock is responsible for releasing.
- However, thread that is calling `sched()` is not.
- `ptable.lock` is acquired before `swtch()` calling
- the thread that is scheduled newly release `ptable.lock`.

```c
void yield(void){
    acquire(&ptable.lock);
    myproc()->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}
```
Scheduling pattern

yield()
yield()
exit() -> sched() -> swtch() -> scheduler -> swtch() -> sched() ->
exit()
sleep()
sleep()

```c
void yield(void) {
    acquire(&ptable.lock);
    myproc()->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}
```

```c
void sched(void) {
    ...
    swtch(&p->context, mycpu()->scheduler);
    mycpu()->intena = intena;
}
```
Scheduling pattern

yield()
yield()
exit()  -> sched() -> swtch() -> scheduler -> swtch() -> sched() ->
exitsleep()
sleep()

void sched(void)
{
    ...
    swtch(&p->context, mycpu()->scheduler);
    mycpu()->intena = intena;
}

swtch:  
... ...
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
Scheduling pattern

```c
void scheduler(void) {
    ...  
    for(;;){  
        ...  
        for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
            ...  
            swtch(&c->scheduler), p->context);
            switchkvm();
            ...  
        }
        ...  
    }
    ...  
}
```
Scheduling pattern

\[\text{yield()} \rightarrow \text{yield()} \rightarrow \text{exit()} \rightarrow \text{sched()} \rightarrow \text{swtch()} \rightarrow \text{scheduler} \rightarrow \text{swtch()} \rightarrow \text{sched()} \rightarrow \text{exit()} \rightarrow \text{sleep()} \rightarrow \text{sleep()}\]

```c
void scheduler(void) {
    ...
    for(;;){
        ...
        for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
            ...
            swtch(&c->scheduler, p->context);
            switchkvm();
            ...
        }
    }
}
```

```
swtch:
    ...
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```
Scheduling pattern

```
yield()
yield()
exit()  -> sched() -> swtch() -> scheduler -> swtch() -> sched() ->
exit()
sleep()
sleep()
```

```c
void sched(void)
{
    ... ...
    swtch(&p->context, mycpu()->scheduler);
    mycpu()->intena = intena;
    
    ret
}
```
start scheduler

- `mpmain()` is started in `main()`.

```c
int
main(void)
{
    ...
    startothers();  // start other processors
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP));  // must come after startothers()
    userinit();    // first user process
    mpmain();      // finish this processor's setup
}
```
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
}
```
The scheduler loops over the process table looking for a runnable process, one that has p->state == RUNNABLE.

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);
    }
}
**scheduler()**

**Outer loop**

- Interrupt is enabled.: 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 looped with the lock continuously held, no other CPU that was running a process could ever 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.