Operating Systems Practice

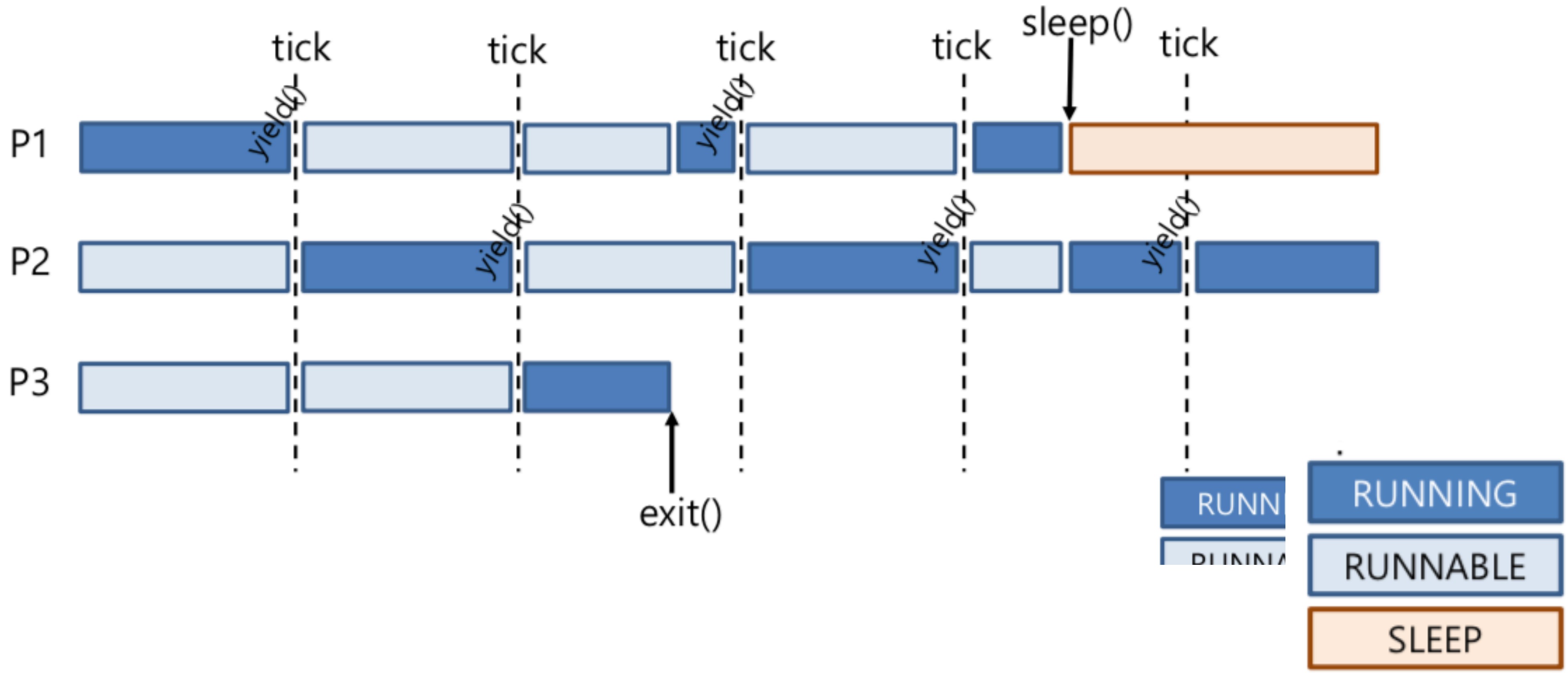
Scheduler

Eunji Lee

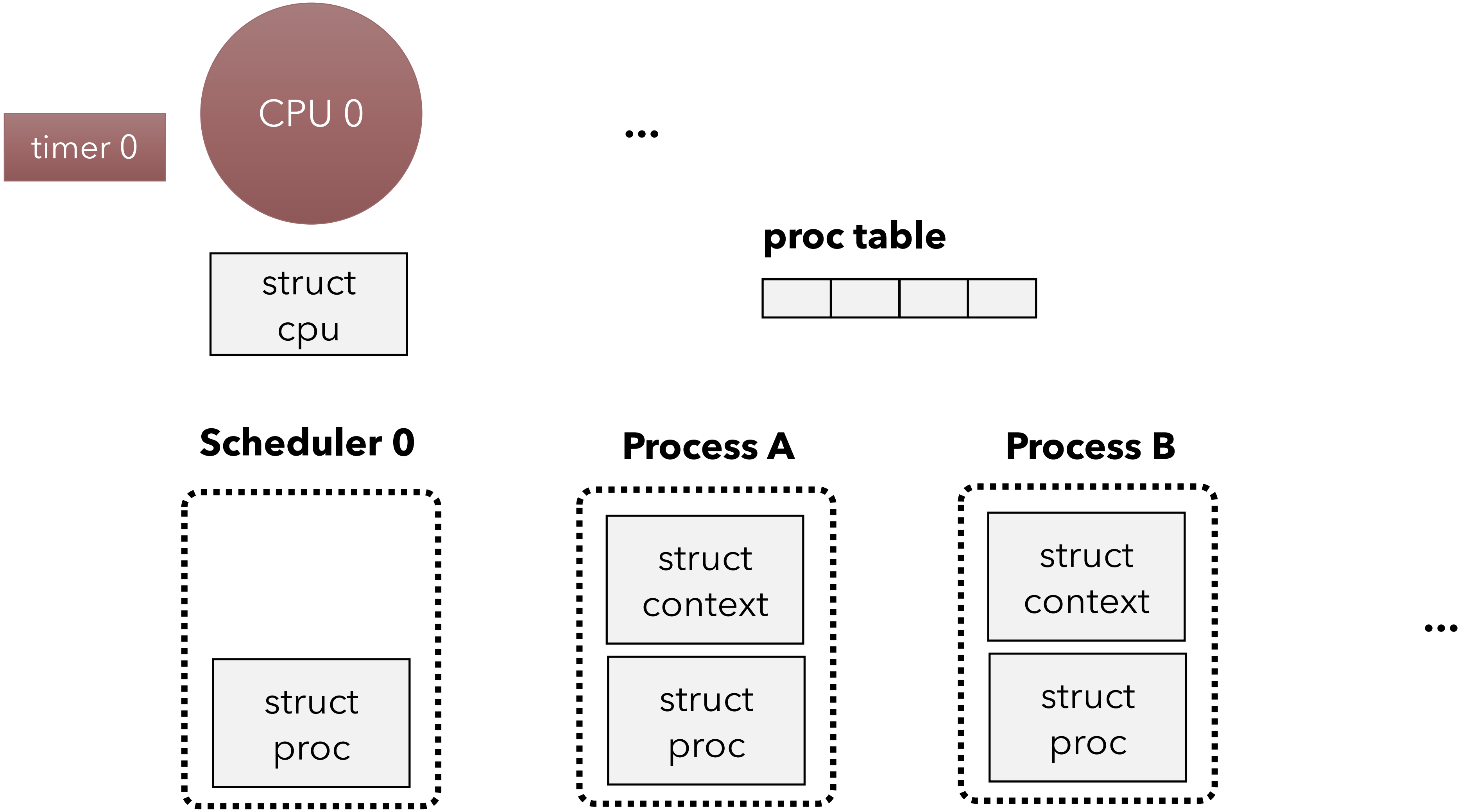
(ejlee@ssu.ac.kr)

XV6 Scheduler – Round Robin

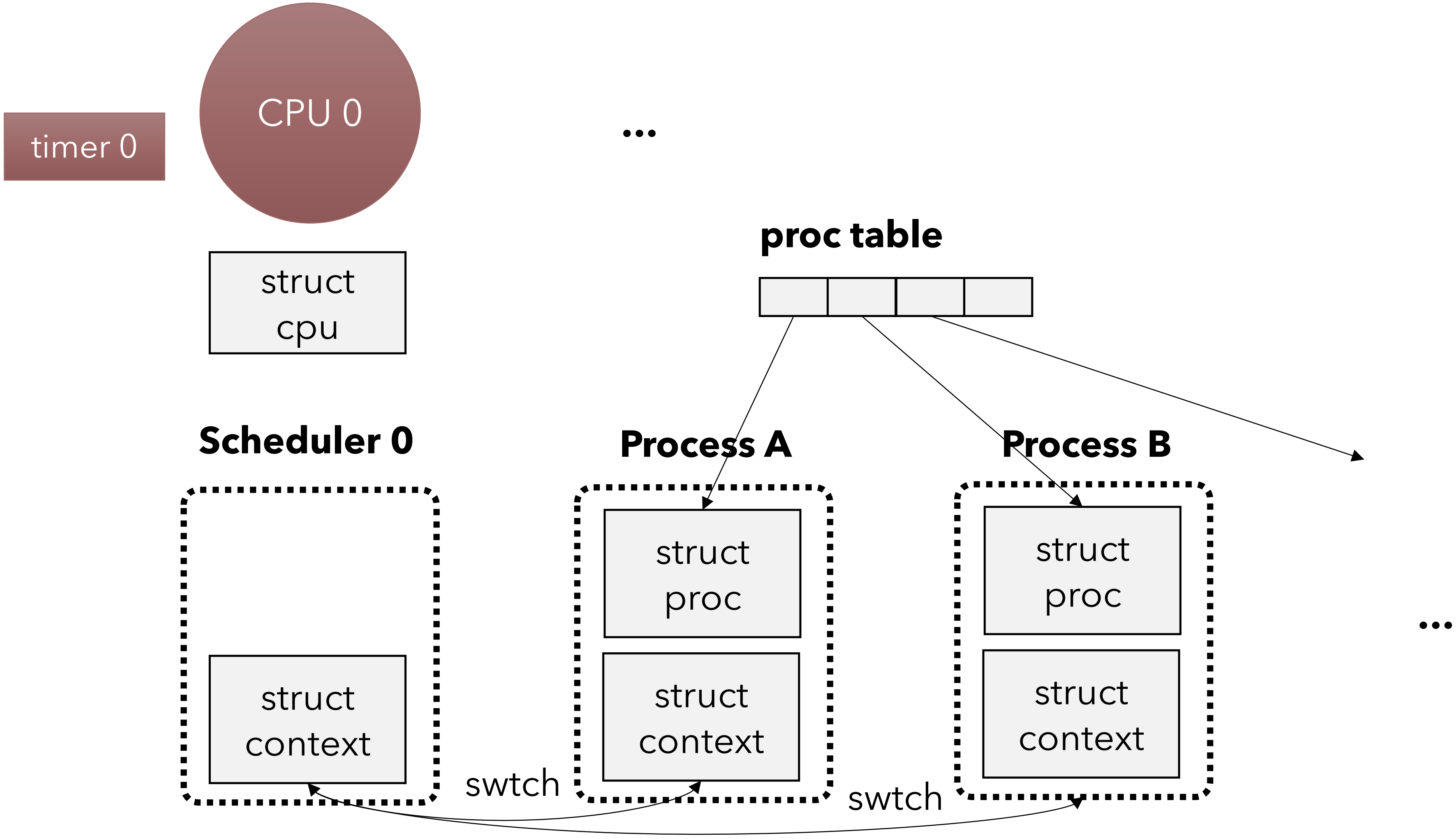
- Timer's interrupt request (IRQ) enforces an yield of a CPU
- A "RUNNABLE" process is chosen to be run in a round-robin manner



XV6 Data Structures for Scheduling



XV6 Data Structures for Scheduling



Process

- proc.h
- procstate
- struct proc

```c
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int priority;                // Process priority
};
```

Proc State

- UNUSED: Not used
- EMBRYO: Newly allocated (not ready for running yet)
- SLEEPING: Waiting for I/O, child process, or time
- RUNNABLE: Ready to run
- RUNNING: Running on CPU
- ZOMBIE: Exited

Process

- proc.h
- struct context

```c
//PAGEBREAK: 17
// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

Scheduler

- proc.h
- struct cpu

```c
// 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];
extern int ncpu;
```

Scheduler

main.c

```c
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come a
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

```c
// 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
}
```

proc.c

```c
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;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      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;
    }
    release(&ptable.lock);

  }
}
```

Start to execute chosen process

When to Schedule

- exit(), sleep()
- timer interrupt (yield())

trap.c

```
void
trap(struct trapframe *tf)
{
```

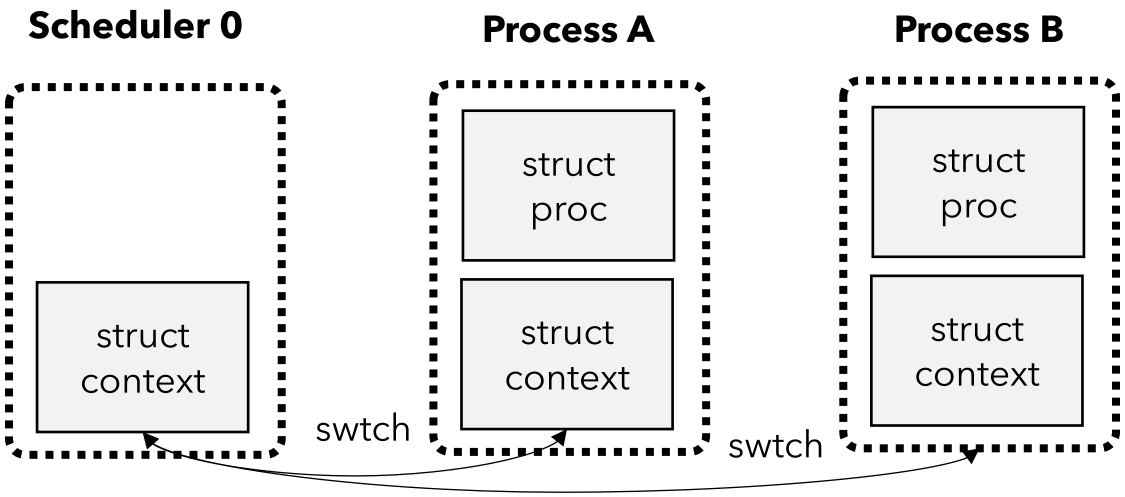
```
  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(myproc() && myproc()->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
```

How Scheduler works

```c
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}

// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```



How Scheduler works

```c
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;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      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;
    }
    release(&ptable.lock);

  }
}
```

Return from here

Project 2. Priority Scheduling

- Implement priority scheduler in xv6
  - The lower nice value, the higher priority
  - The highest priority process should be chosen for next running
    - Tiebreak: Arbitrary

- Scheduler runs only when a change occurs in process priorities
  - DO NOT call the scheduler on the timer interrupt
  - When a process calls fork(), the nice value of child process is set to 5.

test_sched.c

- Add test_sched.c
- ./test_sched

```c
#include "types.h"
#include "stat.h"
#include "user.h"


int main(int argc, char** argv)
{
    int pid;
    int mypid;

    // Change the priority of init processes.
    setnice(1, 10);

    // Change the priority of current processes.
    setnice(getpid(), 2);

    // Create a child process
    pid = fork();

    if(pid == 0) {
        printf(1, "#### State 2 ####\n");
    } else {
        printf(1, "#### State 1 ####\n");

        // Change the priority of parent process.
        setnice(pid, 10);
        wait(); // Yield CPU

        printf(1, "#### State 3 ####\n");
    }

    mypid = getpid();
    printf(1, "PID %d is finished\n", mypid);

    exit();

}
```

```
$ test_sched
#### State 1 ####
#### State 2 ####
PID 4 is finished
#### State 3 ####
PID 3 is finished
```

Hand-in Procedures (1/2)

- Download template
  - https://github.com/eunjicious/xv6-ssu.git (pull or clone)
  - `tar xvzf xv6_ssu_syscall.tar.gz`

- Rename directory
  - `mv xv6_ssu_syscall xv6_ssu_sched`

- Add test_sched.c to your codes and modify Makefile properly
- Build with CPUS=1 flag
  - Makefile

```
ifndef CPUS
CPUS := 1
endif
```

Hand-in Procedures (2/2)

- Compress your code (ID: 20201234)
  - `$tar cvzf xv6_ssu_sched_20201234.tar.gz xv6_ssu_sched`
  - Please command `$make clean` before compressing

- Submit your `tar.gz` file through myclass.ssu.ac.kr

- NO DELAY is allowed !!
- PLEASE DO NOT COPY !!