Project #2: Scheduling

Instructor: Sungyong Ahn

Process Control Block in xv6

- `struct proc` in proc.h

```
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)
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
```

Process Control Block in xv6

- **`struct ptable` in proc.c**

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

- **in param.h**

```
#define NPROC        64  // maximum number of processes
```

- **Process states (`procstate` in proc.h)**
  - 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

Xv6 Process Scheduler

- main() in main.c

```
int
main(void)
{
  userinit();          // first user process
  mpmain();            // finish this processor's setup
}
```

- mpmain() in main.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
}
```

Xv6 Process Scheduler (Cont'd)

- **`scheduler()` in proc.c**
  - Round-robin fashion

```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);

  }
}
```

Choose a next process

Start to execute chosen process

Xv6 Process Scheduler (Cont'd)

- `swtch()` in swtch.S

  Read **Ch.5 Scheduling** of xv6 commentary book!!

```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

in proc.h

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

Xv6 Entering Scheduler

- **When?**

1. Exiting process (`exit()` in proc.c)

```
// Jump into the scheduler, never to return.
curproc->state = ZOMBIE;
sched();
```

2. Sleeping process (`sleep()` in proc.c)

```
// Go to sleep.
p->chan = chan;
p->state = SLEEPING;

sched();
```

Xv6 Entering Scheduler (Cont'd)

- **When?**
  3. Yielding CPU due to timer interrupt
     - `trap()` in trap.c

```
  // 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();
```

     - `yield()` in proc.c

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

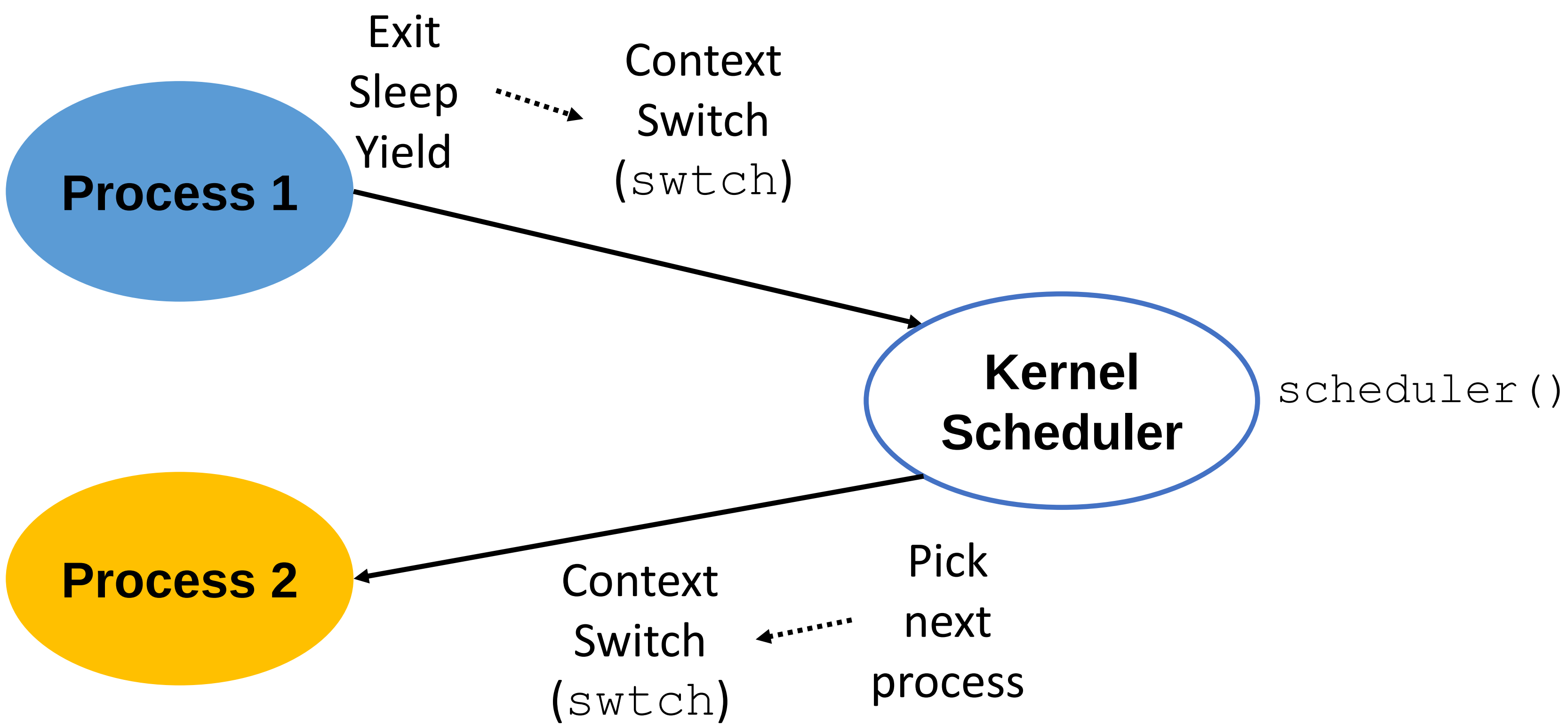
Xv6 Entering Scheduler (Cont'd)

- `sched()` in proc.c

```c
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state. Saves and restores
// intena because intena is a property of this
// kernel thread, not this CPU. It should
// be proc->intena and proc->ncli, but that would
// break in the few places where a lock is held but
// there's no process.
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;
}
```

Xv6 Scheduler Summary



Project #2. Priority Scheduler

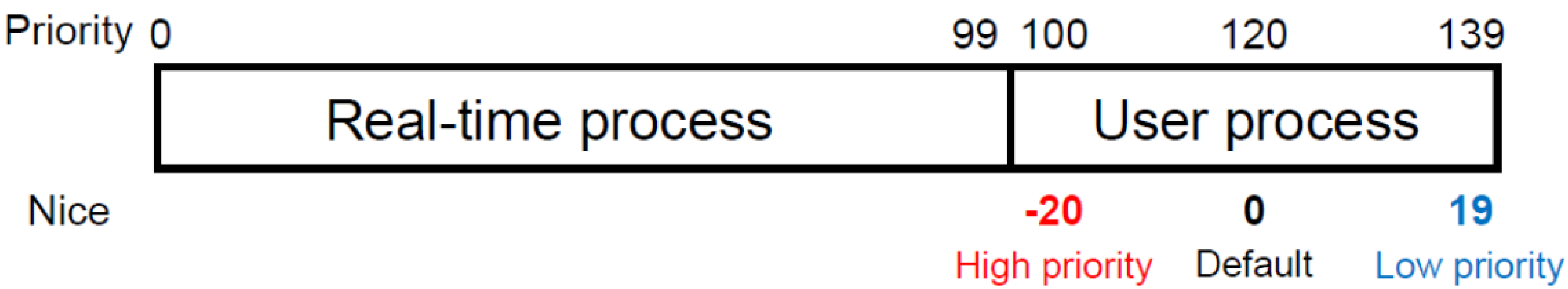
1. **Implement system calls related to process priority**
   - `setnice(), getnice(), ps()`

2. **Implement priority-based scheduler on xv6**
   - The lower nice value, the higher priority
   - The highest priority process is selected for next running
     - Tiebreak: FIFO fashion

Project #2-1: Process Priority

- For represent weight between processes

- In Linux



- In Xv6
  - Not implemented!

Project #2-1: Process Priority

- **Make three system calls (`getnice`, `setnice`, `ps`)**

- **Description of nice value**
  - The default nice value is 20.
  - The range of valid nice value is [0, 40].
  - Lower nice values cause more favorable scheduling.
  - When a process calls `fork()` system call, the nice value of child process is same as its parent process.

Project #2-1: Process Priority

- **Make three system calls (`getnice`, `setnice`, `ps`)**

- **`int setnice(int pid, int nice_value)`**
  - Get the nice value of process (min: 0 / max: 40)
  - Return 0 on success.
  - Return -1 if there is no process corresponding to the pid or the nice value is invalid.

- **`int getnice(int pid)`**
  - Return the nice value of pid(process)
  - Return -1 if there is no process corresponding to the pid.

Project #2-1: Process Priority

- **`void ps(int pid)`**
  - The `ps` function prints out process(s)'s information, which includes **pid, ppid, nice, status,** and **name of each process**. If the pid is 0, print out all processes' information. Otherwise, print out corresponding process's information. If there is no process corresponding to the pid, print out nothing.
  - No return value
  - Make a user program (**`minitop`**) displays process status using `ps` system call

```
qemu-system-i386 -nographic -drive file=
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nl
init: starting sh
$ minitop
pid     ppid    prio    state   name
1       1       20      sleep   init
2       1       20      sleep   sh
3       2       20      run     minitop
$
```

- **Reading Chap. 3 of xv6-commentary will help your project**

Project #2-1: Process Priority

- **To-do list for process priority**
  - Create a variable for a nice value
  - Initialization code for a nice value
    - What is default nice value?
    - How can child process inherit the nice value of parent process?
- **Then, implement three system calls related to nice value**
  - `setnice()`, `getnice()`, `ps()`

fork()

- in proc.c

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;
```

Process Dump in proc.c

```
//PAGEBREAK: 36
// Print a process listing to console.  For debugging.
// Runs when user types ^P on console.
// No lock to avoid wedging a stuck machine further.
void
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  int i;
  struct proc *p;
  char *state;
  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
    cprintf("%d %s %s", p->pid, state, p->name);
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
    cprintf("\n");
  }
}
```

Process Dump in proc.c

```
user@ubuntu:~/xv6-SSE3044$ make qemu-nox
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.01316 s, 389 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000992285 s, 516 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
334+1 records in
334+1 records out
171164 bytes (171 kB, 167 KiB) copied, 0.000756917 s, 226 MB/s
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive f
ormat=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ 1 sleep  init 80103dc7 80103e69 80104837 801058d9 80105638
2 sleep  sh 80103d8c 801002c2 80100f8c 80104b32 80104837 801058d9 80105638

$ 
```

Project #2-2. Priority Scheduler

- **Ignore to yield CPU on clock tick (`trap()` in trap.c)**

```
// 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();
*/
```

- **Entering scheduler when**
  1. Exiting process
  2. Sleeping process
  3. Yielding CPU
  4. Changing priority (setnice)

Sample Test Program

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

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

    setnice(1, 19);
    setnice(getpid(), 2);

    pid = fork();

    if(pid == 0){    //Child
        printf(1, "##### State 2 #####\n");
    }
    else{            //Parent
        setnice(pid, 10);   //Set nice value of Child
        printf(1, "##### State 1 #####\n");
        wait();             //Scheduling
        printf(1, "##### State 3 #####\n");
    }

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

    exit();
}
```

```
$ test2
##### State 1 #####
##### State 2 #####
PID 4 is finished
##### State 3 #####
PID 3 is finished
```

Project #2. Template Code

- Download **xv6-pnu-p2.tar.gz** from PLATO
- **Modifications**
  - CPUS=1 (`Makefile`)

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

  - Ignore to yield CPU on clock tick
  - `yield` system call
    - Yield CPU

Submission

- **Compress your xv6 folder as `StudentID-2.tar.gz`**
  - `$make clean`
  - `$tar -czvf StudentID-2.tar.gz ./xv6-pnu-p2`
  - Please command `$make clean` before compressing

- **Submit your `tar.gz` file through PLATO**

- **Due date: 4/20, 23:59**
  - -25% penalty of total mark per day

- **PLEASE DO NOT COPY !!**
  - YOU WILL GET F IF YOU COPIED