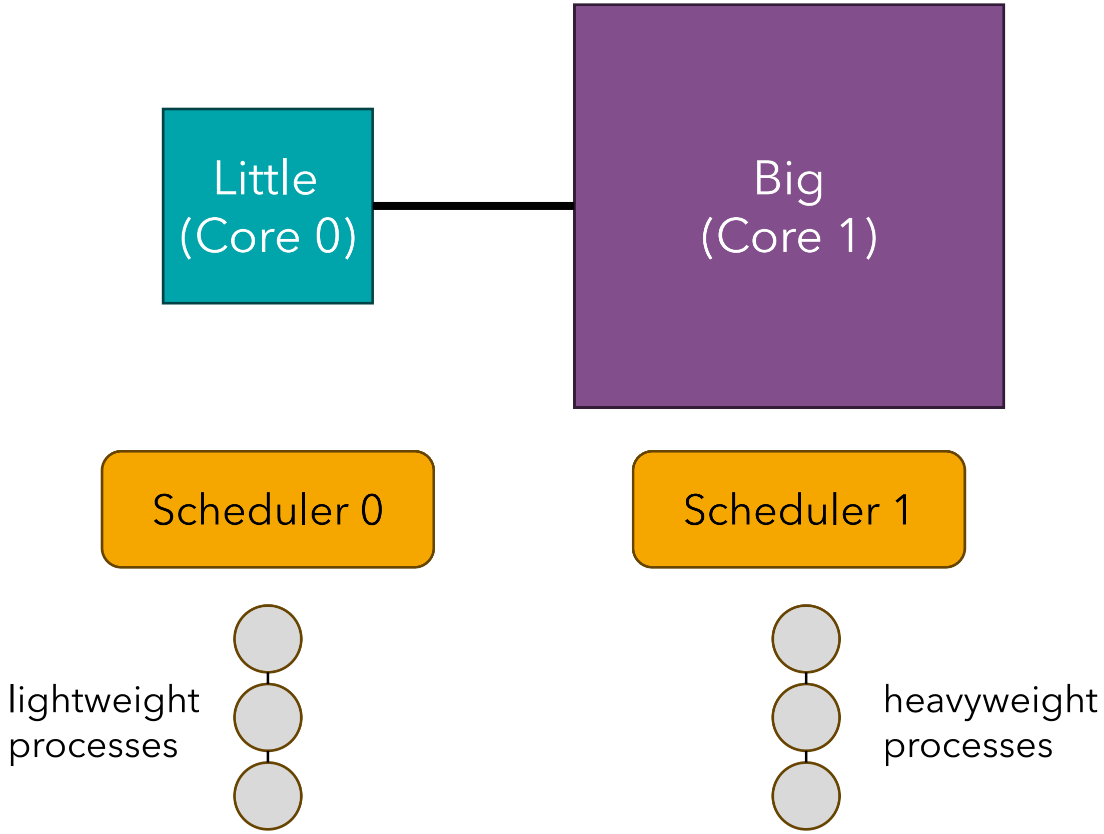
Operating Systems

Project 1 - Scheduler

Minjae Kim
jwya2149@dgist.ac.kr

Project #1

- Design & Implement a basic Big-Little CPU Scheduler
  - Modify the xv6 scheduler to a scheduler that manages workloads in a big-little multicore architecture



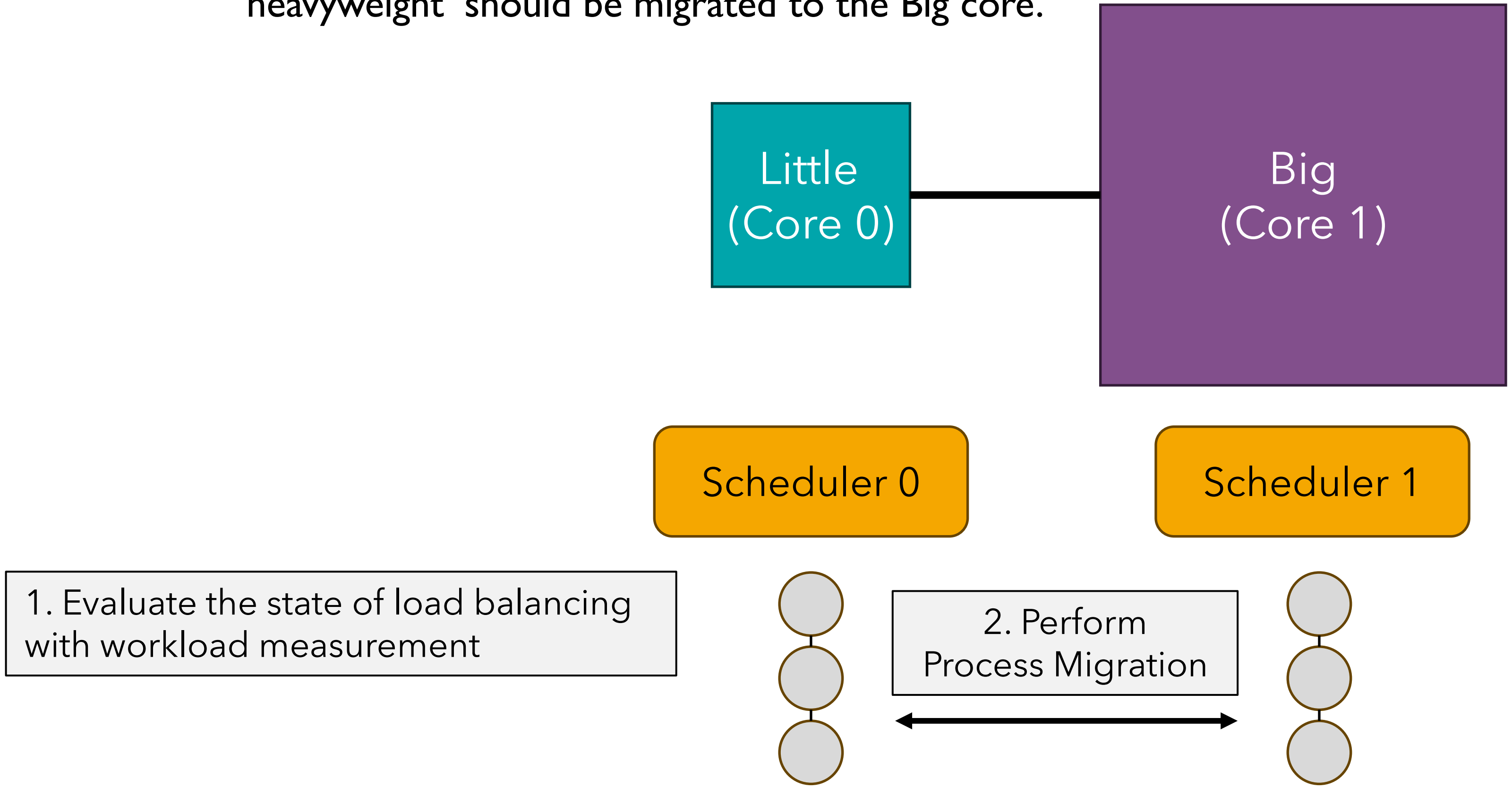
Project #1

- Objectives of this project
  - Understand how context-switches are performed in xv6 code
  - Investigate the functioning of the big-little core architecture
  - Explore how to differentiate between heavy/light processes based on their resource requirements

- Where to look and write code:
  - proc.c, proc.h (+ etc)

Project #1

- Implementation Approach in xv6
  - Assume there is one Little core and one Big core, with a per-CPU scheduler for each core
  - Newly created processes are assigned to the Little core
  - If the workload in the Little Scheduler exceeds a specified threshold, the processes that are classified as 'heavyweight' should be migrated to the Big core.



Project #1

- Criteria for process migration? (i.e., Little core is busy)
    - # of RUNNABLE processes
    - Total CPU usage time
    - I/O wait Time
    - …

e.g., when the number of RUNNABLE processes in the Little core is over 10,
select the process of the highest CPU runtime and migrate it to Big core.

- Criteria for heavyweight processes?
    - Execution time (i.e., CPU runtime)
    - Priority level
    - Memory usage
    - Historical performance data
    - …

Tasks for Students

- Step 1
  - Assign newly created processes to CPU 0
  - Implement a function to enable process migration between CPUs

- Step 2
  - Develop a system load measurement feature
  - Implement a load balancing mechanism to identify heavy processes and migrate them to Big core

Experimental Environments

- Benchmarks to be provided
  - Programs that generate multiple heavy and light processes
  - Measurements of scheduling metrics such as total time, turnaround time, response time, …
  - Verification of whether heavy processes are executed on the Big Core

- Big/Little Core Simulation
  - Direct modification of CPU performance is not possible in xv6
  - The provided benchmarks will ensure that heavy workloads continuously run on CPU 0 to simulate Little Core behavior

Evaluation

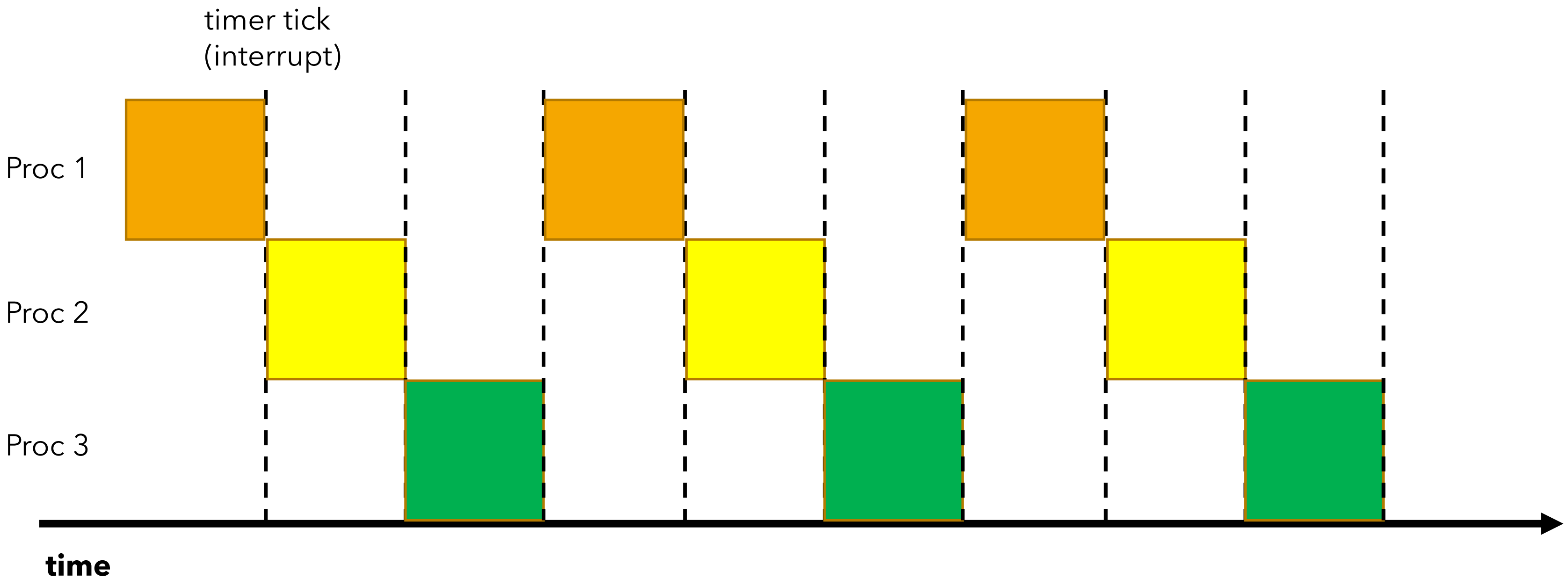
- Proper Functionality
  - Ensure the system operates without kernel panics
  - Verify that the system functions reasonably and efficiently
- Process Classification
  - Evaluate how effectively heavy and light processes are distinguished
- Report
  - Implementation Clarity: Assesses the clarity and thoroughness of your implementation description
  - Results Analysis: Analyze the results effectively, including insights on performance and behavior

Project #1

- Deadline
  - ~ 2024.10.30 (Wed) 23:59
- Hand-in procedure
  - proj1_202212345.patch (student number)
    - Run the following command and upload proj1_202212345.patch
      - `git diff > proj1_202212345.patch`
    - Check the patch file with Notepad and confirm your modifications are in the patch file
  - Report
    - Submit a report (No page limit, but 2~3 pages are enough)
      - Free format (Korean/English)
      - Description of your implementation
      - Analysis of benchmark programs

Project #1

- What is the default xv6 scheduler?
  - Round-robin scheduler
  - For each timer tick (~10ms), a timer interrupt occurs to incur a context switch
  - Don't need to change this scheduling policy



Project #1

- Functions to create a process in xv6
  - proc.c: fork(), allocproc()

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

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

...

```c

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

```c
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  release(&ptable.lock);
```

Project #1

- Core function: *void scheduler() in proc.c*
- First *for loop* is looping forever
  - *This function never returns*
  - *Find a new process to be scheduled, run it until it yields*

Infinite loop (1 tick)

- Scan ptable to find RUNNABLE process

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

- Switch from scheduler to the process
  - After the process yields by timer interrupt, come back to swtch()

<default round-robin scheduler code>

```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);
  }
}
```

Project #1

- 10ms after the user process occupies the CPU, timer interrupt happens

in trap.c, trap() calls yield()

```c
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
```

…

```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();
```

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

```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;
}
```

back to scheduler(), starts at line 345

Project #1

- per-process structure: struct proc in proc.h

```c
// Per-process state
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
rocess
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

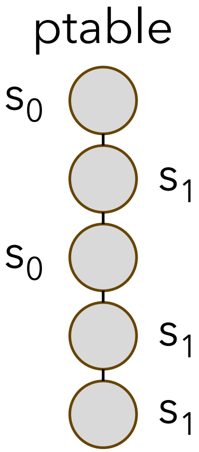
- You can add member variables for per-process variable

  e.g.,) runtime, cpu_affinity, …

Project #1

Scheduler 0

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

  cprintf("Scheduler id: %d\n", c->apicid);

  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);

  }
}
```



Scheduler 1


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

  cprintf("Scheduler id: %d\n", c->apicid);

  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);

  }
}
```

Project #1

- References
  - To understand more detail about xv6 scheduler, study Chapter 5 in the xv6 book (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)

- Build xv6 with 'CPUS = 2' flag to test easier (In Makefile)

```
ifndef CPUS
CPUS := 2
endif
```

mpmain() called twice?

```
// 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
}
```

Finally…

Do NOT hesitate to ask questions!