Lab 10:

Designing & Deploying New Scheduling Algorithms in XV6

Omar Harb 900201063
Sara Mohamed 900203032
Mohamed Shaalan 900201539

Roles:

Omar Harb: schedulers

Sara Mohamed: testing of schedulers

Mohamed Shaalan: system calls & their user programs

Priority Ranges and Semantics:

Range Chosen:

1 - 40

Min Value: 40, Max Value: 1, Default Value: 20

Why?

All Linux and UNIX based systems use range from -20 to 19, have minimum priority for maximum value, maximum priority for minimum value, and default value as 0, so we modeled that same behaviour in xv6.

System Calls and User Application Verifications

printptable():

for system call:

loops through process table and prints out the processes - no particular verification added

for user program:

verifies that no command line arguments are input, and then calls the system call

set_priority():

for system call:

loops through process table to find the process of the given pid to set its priority - if found returns old priority, if not, returns -1

for user program:

verifies that two command line arguments are input, then verifies that the value given for the pid corresponds to a real process, then calls the system call

System Calls and User Application Test Programs

printptable():

testing:

program forks a few children, then calls printptable() to view full process table

set_priority():

testing:

program uses set_priority() to set a non-default priority to process, then calls printptable() to show the updated priority

Scheduling Algorithms: Descriptions

Priority Based: loops over process table to find the runnable process with the highest priority; and then sets it to running

Decaying: loops over process table to find the runnable process with the highest priority; and then lowers its priority by adding 2, then sets it to running

Pseudocode: Priority-Based Scheduler

```
scheduler(){

      process *p;
      cpu *c = mycpu();
      loop infinitely{
      struct cpu *c = mycpu();
      c->proc = 0; //no process running on cpu now
      loop infinitely{
         // Enable interrupts on this processor.
         sti();
         process *highestPriority = 0;//to find proc with
                                      //highest priority
         acquire ptable lock;
         for(each process p in ptable){
           if(p->state != RUNNABLE)
             continue;
           else
             if (highestPriority == 0)
               highestPriority = p;
             else
               if (p->priority < highestPriority->priority)
                 highestPriority = p;
         }

      if (highestPriority != 0)//if found a runnable process
      {
             c->proc = highestPriority;//set that process to run on cpu
             highestPriority->state = RUNNING;//set its state to running
             switchuvm(highestPriority);//switch the TSS and h/w page table
                                        // to correspond to process
             swtch(&(c->scheduler), highestPriority->context);
      }

      switchkvm(); //switch h/w page table register to the kernel-only page
                   //table


      c->proc = 0; //no process running on cpu now
      release(&ptable.lock);

}
```

Pseudocode: Decay-Based Scheduler

```
scheduler(){

      process *p;
      cpu *c = mycpu();
      loop infinitely{
      struct cpu *c = mycpu();
      c->proc = 0; //no process running on cpu now
      loop infinitely{
         // Enable interrupts on this processor.
         sti();
         process *highestPriority = 0;//to find proc with
                                      //highest priority
         acquire ptable lock;
         for(each process p in ptable){
           if(p->state != RUNNABLE)
             continue;
           else
             if (highestPriority == 0)
               highestPriority = p;
             else
               if (p->priority < highestPriority->priority)
                 highestPriority = p;
         }

      if (highestPriority != 0)//if found a runnable process
      {
             c->proc = highestPriority;//set that process to run on cpu
             highestPriority->state = RUNNING;//set its state to running
             add 2 to priority, if priority has not yet reached 40;
             switchuvm(highestPriority);//switch the TSS and h/w page table
                                        // to correspond to process
             swtch(&(c->scheduler), highestPriority->context);
      }

      switchkvm(); //switch h/w page table register to the kernel-only page
                   //table


      c->proc = 0; //no process running on cpu now
      release(&ptable.lock);

}
```

Decay

Decay Factor and Frequency:

+2 priority per scheduler's decision to schedule

```
  if (highestPriority != 0){
  c->proc = highestPriority;
  highestPriority->state = RUNNING;
  if ((highestPriority->priority) + 2 <= 40){
    highestPriority->priority = (highestPriority->priority + 2);
  }
  switchuvm(highestPriority);
  swtch(&(c->scheduler), highestPriority->context);
  }
  switchkvm();
  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
release(&ptable.lock);
```

Priority-Based Scheduler Test Cases

```
================
START OF TESTING
================
child process 4: with pid 7 created.
child process 3: with pid 6 created.
child process 2: with pid 5 created.
child process 4: with pid 7 finished.
child process 3: with pid 6 finished.
child process 2: with pid 5 finished.
child process 1: with pid 4 created.
child process 1: with pid 4 finished.
================
END OF TESTING
================
```

```c
int main(int argc, char *argv[])
{
    printf(1, "================\nSTART OF TESTING\n================\n");

    for (int i = 0; i < 4; i++)
    {
        if (fork() == 0)
        {
            setpriority(getpid(), 39 - i);
            printf(1, "child process %d: with pid %d created.\n", i+1, getpid());
            int temp = 1;
            for (int a = 0; a < 10000; a++)
            {
                temp = temp * 2;
                yield();
            }
            printf(1, "child process %d: with pid %d finished.\n", i+1, getpid());
            exit();
        }
    }

    while (wait() > 0)
        ;
    printf(1, "================\nEND OF TESTING\n================\n");

    exit();
}
```

Explanation:

Parent forks 4 children, which we assign increasing priority using set_priority(), e.g. the second process is of higher priority than the second, etc. Upon creation, each child enters a time-consuming loop in which every iteration contains a yield() instruction, giving up control to the cpu. Thus, the process with the highest priority (the one created last), is scheduled and finishes first, and the others follow in descending order.

Decay-Based Scheduler Test Cases

```
================
START OF TESTING
================
child process 1: with pid 4 created.
child process 2: with pid 5 created.
child process 3: with pid 6 created.
child process 4: with pid 7 created.
child process 1: with pid 4 finished.
child process 2: with pid 5 finished.
child process 3: with pid 6 finished.
child process 4: with pid 7 finished.
================
END OF TESTING
================
```

```c
int main(int argc, char *argv[])
{
    printf(1, "================\nSTART OF TESTING\n================\n");

    for (int i = 0; i < 4; i++)
    {
        if (fork() == 0)
        {
            printf(1, "child process %d: with pid %d created.\n", i+1, getpid());
            int temp = 1;
            for (int a = 0; a < 100000; a++)
            {
                temp = temp * 2;
                yield();
            }
            printf(1, "child process %d: with pid %d finished.\n", i+1, getpid());
            exit();
        }
    }

    while (wait() > 0)
        ;
    printf(1, "================\nEND OF TESTING\n================\n");

    exit();
}
```

Explanation:

Parent forks 4 children, each of which get the default priority. Upon creation, each child enters a time-consuming loop in which every iteration contains a yield() instruction, giving up control to the cpu. Thus, eventually all children decay to the lowest priority; and they finish in order of creation. We cannot make them execute out of order in this case, as we did in the no-decay case.

XV6 Changes

```
#define SYS_setpriority 22
#define SYS_printptable 23
#define SYS_yield 24
```

new macros in syscall.h

```
extern int sys_setpriority(void);
extern int sys_printptable(void);
extern int sys_yield(void);
```

externs in syscall.c

```
[SYS_setpriority] sys_setpriority,
[SYS_printptable] sys_printptable,
[SYS_yield] sys_yield,
```

function pointers to syscall pointer array in syscall.c

```
SYSCALL(setpriority)
SYSCALL(printptable)
SYSCALL(yield)
```

SYSCALLS for macros in usys.S

```
int setpriority(int, int);
int printptable(void);
void yield(void);
```

function prototypes in user.h

```
void            yield(void);
int             setpriority(int, int);
int             printptable(void);
```

function prototypes in defs.h

```
int
sys_setpriority(void)
{
  int pid;
  int priority;
  if(argint(0, &pid) < 0)
    return -1;
  if (argint(1, &priority) < 0)
    return -1;
  return setpriority(pid, priority);
}
int sys_printptable(void)
{
  return printptable();
}
int sys_yield(void)
{
  yield();
  return 0;
}
```

function calls in sysproc.c

added all function definitions in proc.c,

XV6 Changes

proc.h:

- Added a numerical priority variable to the proc struct.

proc.c:

- changed the scheduler() function in two instances of xv6 for each scheduling algorithm. The changes include the implementation of the two algorithms.

- assigned a default value to children (20) by initializing priority in the allocproc() function, as children need to have a higher priority than the init process.

- assigned a lower initial priority (30) to the init process by initializing its priority the userinit() function.

- changed the procdump() function to list the same table as printptable(), so we can use the Ctrl+p shortcut to print table as the program runs (for extra testing)

yield() system call:

Turned the already present yield() function into a system call to allow the test user programs to call it. Each child calls yield() to voluntarily give up the CPU, turning its state from ‘RUNNING’ to ‘RUNNABLE’.

XV6 Changes

params.h:

CPUs number lowered from 8 to 1 for the purpose of slowing down the scheduling for it to be easier to trace.