CS4375-13948 Fall 2023
Denisse Loya Villalobos
dloyavilla@miners.utep.edu

Homework Report
Submitted on 23 Oct 2023

HW 3: Priority-based Scheduler for xv6

Task 1. Modify the provided ps command to print the priority of each process.

I added the "priority" field to the `struct proc` in `proc.h` to store process priorities and additional scheduling information.
In `sysproc.c`, I implemented the `getpriority()` and `setpriority()` system calls, allowing users to retrieve and set the priority of specific processes.
To make these system calls accessible to user-level programs, I added the necessary function prototypes and constants to `user.h` and updated `syscall.h`.
The changes were integrated into `syscall.c` to ensure that the new system calls are available for use by user-level programs.

```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID
  uint64 cputime;                    // cputime
  int priority;                // add priority
  int readytime;

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
```

```
3 [SYS_close]   sys_close,
4 [SYS_getprocs]   sys_getprocs,
5 [SYS_wait2]   sys_wait2,
6 [SYS_getpriority]   sys_getpriority,
7 [SYS_setpriority]   sys_setpriority,
8 };
9
```

```
28 int getprocs(struct pstat*);
29 int wait2(int*,struct rusage*);
30 int getpriority(void);
31 int setpriority(int);
32
33 // ulib.c
```

```
// getPriority()
uint64
sys_getpriority(void){
        return myproc()->priority;
}

// setPriority()
uint64
sys_setpriority(void){
        int priority;
        if(argint(0, &priority) < 0) {
                return -1;
        }
        myproc()->priority = priority;
        return 0;
}

// return the number of active processes in the system
```

```
24 #define SYS_wait2  23
25 #define SYS_getpriority 24
26 #define SYS_setpriority 25
```

Task 2. Add a readytime field to struct proc, initialize it correctly, and modify ps to print a process's age.

I added the "readytime" field to the `struct proc` in `proc.h`. This field is initialized to the current time whenever a process's state transitions from another state to RUNNABLE.

In `kernel/pstat.h`, I modified the `struct pstat` to include the "readytime" field. This change enables the `ps` command to access and display process ages for processes in the RUNNABLE state.
The `ps` command is updated to print the process's age when its state is RUNNABLE. The process age is calculated by subtracting the "readytime" of the process from the current time. The result is displayed in seconds.

To find the age of the process, subtract its "ready time" from the current time when you need to display the age. This difference represents the time elapsed since the process became ready to run.

```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID
  uint64 cputime;                     // cputime
  int priority;                // add priority
  int readytime;

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
```

```
struct pstat {
  int pid;     // Process ID
  enum procstate state;  // Process state
  uint64 size;     // Size of process memory (bytes)
  int ppid;        // Parent process ID
  int priority;
```

```
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/pstat.h"
#include "user/user.h"

int
main(int argc, char **argv)
{
  struct pstat uproc[NPROC];
  int nprocs;
  int i;
  char *state;
  static char *states[] = {
    [SLEEPING]  "sleeping",
    [RUNNABLE]  "runnable",
    [RUNNING]   "running ",
    [ZOMBIE]    "zombie  "
  };

  nprocs = getprocs(uproc);
  if (nprocs < 0)
    exit(-1);

  printf("pid\tstate\tsize\tage\tpriority\tcputime\tppid\tname\n");
  for (i=0; i<nprocs; i++) {
    int age = uptime() - uproc[i].readytime;
    state = states[uproc[i].state];
    printf("%d\t%s\t%d\t%d\t%d\t%d\t%d\t%s\n", uproc[i].pid, state,
                  uproc[i].size,age, uproc[i].priority,uproc[i].cputime, uproc[i].ppid, uproc[i].name);
  }

  exit(0);
}
```

Task 3. Implement a priority-based scheduler.

Introduced constants in `param.h` to allow for the selection of scheduling policies at compile time. These constants determine the scheduling policy, such as whether the priority-based scheduler should be used.
Implemented a priority-based scheduler in the operating system. This scheduler selects the highest priority process for execution. In the case of a tie among the highest priority processes, the scheduler selects any one of them.
Modified `proc.h` and `proc.c` to include a "priority" field in the `struct proc` and initialized it for each process.
Modified the necessary system calls and their corresponding system call handlers to enable processes to set and get their priority values.
Developed test programs to validate the functionality of the priority-based scheduler. These programs include multiple processes with varying priorities.
The results of running the test programs indicate that the priority-based scheduler effectively prioritizes higher-priority processes, demonstrating the expected behavior of the scheduling policy. The use of the `pexec` program for testing helps showcase the priority-based execution of processes.

Difficulties: Implementing the scheduler needed extensive testing.

```
16 #define ROUNDROBIN   0
17 #define PRIORITY     1
18 #define DREFSCHED     PRIORITY
19 enum procstate {UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE};
```

```
496 void
497 scheduler(void)
498 {
499   struct proc *p;
500   struct cpu *c = mycpu();
501   struct proc *highProc;
502
503   c->proc = 0;
504   for(;;){
505     // Avoid deadlock by ensuring that devices can interrupt.
506     intr_on();
507     if(DREFSCHED == 0){
508         for(p = proc; p < &proc[NPROC]; p++) {
509                 acquire(&p->lock);
510                 if(p->state == RUNNABLE) {
511                 // Switch to chosen process.  It is the process's job
512                 // to release its lock and then reacquire it
513                 // before jumping back to us.
514                 p->state = RUNNING;
515                 c->proc = p;
516                 swtch(&c->context, &p->context);
517
518                 // Process is done running for now.
519                 // It should have changed its p->state before coming back.
520                 c->proc = 0;
521                 }
522                 release(&p->lock);
523         }
524     } else {
525         highProc = proc;
526         int highestProcess = 0;
527         for(p = proc; p < &proc[NPROC]; p++) {
528                 int age = sys_uptime() - p->readytime;
529                 int priorityCal = p->priority + age;
530                 acquire(&p->lock);
```

```
516                 swtch(&c->context, &p->context);
517
518                 // Process is done running for now.
519                 // It should have changed its p->state before coming back.
520                 c->proc = 0;
521                 }
522                 release(&p->lock);
523         }
524     } else {
525         highProc = proc;
526         int highestProcess = 0;
527         for(p = proc; p < &proc[NPROC]; p++) {
528                 int age = sys_uptime() - p->readytime;
529                 int priorityCal = p->priority + age;
530                 acquire(&p->lock);
531                 if(p->state == RUNNABLE) {
532                         if(priorityCal > highestProcess) {
533                                 highestProcess = priorityCal;
534                                 highProc = p;
535                         }
536                 }
537                 release(&p->lock);
538         }
539         acquire(&highProc ->lock);
540         if(highProc->state == RUNNABLE) {
541                 highProc->state = RUNNING;
542                 c->proc = highProc;
543                 swtch(&c->context, &highProc->context);
544                 c->proc = 0;
545         }
546         release(&highProc->lock);
547     }
548   }
```

```
init: starting sh
$ pexec 10 ps
pid     state   size    age     priority        cputime ppid    name
1       sleeping        12288   49      0       0       0       init
2       sleeping        16384   9       0       0       1       sh
3       sleeping        12288   5       10      0       2       pexec
4       running         12288   1       0       0       3       ps
$
```

Task 4. Add aging to your priority based scheduler.

The priority-based scheduler is modified to include an aging policy to prevent process starvation.

The scheduler periodically checks the age of processes in the RUNNABLE state.

Processes in the RUNNABLE state have their priorities gradually increased over time to prevent starvation. The exact aging algorithm specifics may vary based on the operating system's design.

Developed test programs to verify the aging policy's functionality, assessing the impact of aging on process scheduling.

The results of running these test programs demonstrate the effectiveness of the aging policy. Lower-priority processes are less likely to starve, leading to improved system performance and fair resource allocation.

Difficulties: Making the necessary adjustments to the code, so it can function correctly

```
496 void
497 scheduler(void)
498 {
499   struct proc *p;
500   struct cpu *c = mycpu();
501   struct proc *highProc;
502
503   c->proc = 0;
504   for(;;){
505     // Avoid deadlock by ensuring that devices can interrupt.
506     intr_on();
507     if(DREFSCHED == 0){
508         for(p = proc; p < &proc[NPROC]; p++) {
509                 acquire(&p->lock);
510                 if(p->state == RUNNABLE) {
511                 // Switch to chosen process.  It is the process's job
512                 // to release its lock and then reacquire it
513                 // before jumping back to us.
514                 p->state = RUNNING;
515                 c->proc = p;
516                 swtch(&c->context, &p->context);
517
518                 // Process is done running for now.
519                 // It should have changed its p->state before coming back.
520                 c->proc = 0;
521                 }
522                 release(&p->lock);
523         }
524     } else {
525         highProc = proc;
526         int highestProcess = 0;
527         for(p = proc; p < &proc[NPROC]; p++) {
528                 int age = sys_uptime() - p->readytime;
529                 int priorityCal = p->priority + age;
530                 acquire(&p->lock);
```

```
516                 swtch(&c->context, &p->context);
517
518                 // Process is done running for now.
519                 // It should have changed its p->state before coming back.
520                 c->proc = 0;
521                 }
522                 release(&p->lock);
523         }
524     } else {
525         highProc = proc;
526         int highestProcess = 0;
527         for(p = proc; p < &proc[NPROC]; p++) {
528                 int age = sys_uptime() - p->readytime;
529                 int priorityCal = p->priority + age;
530                 acquire(&p->lock);
531                 if(p->state == RUNNABLE) {
532                         if(priorityCal > highestProcess) {
533                                 highestProcess = priorityCal;
534                                 highProc = p;
535                         }
536                 }
537                 release(&p->lock);
538         }
539         acquire(&highProc ->lock);
540         if(highProc->state == RUNNABLE) {
541                 highProc->state = RUNNING;
542                 c->proc = highProc;
543                 swtch(&c->context, &highProc->context);
544                 c->proc = 0;
545         }
546         release(&highProc->lock);
547     }
548   }
```

```c
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/pstat.h"
#include "user/user.h"

int
main(int argc, char **argv)
{
  struct pstat uproc[NPROC];
  int nprocs;
  int i;
  char *state;
  static char *states[] = {
    [SLEEPING]  "sleeping",
    [RUNNABLE]  "runnable",
    [RUNNING]   "running ",
    [ZOMBIE]    "zombie  "
  };

  nprocs = getprocs(uproc);
  if (nprocs < 0)
    exit(-1);

  printf("pid\tstate\tsize\tage\tpriority\tcputime\tppid\tname\n");
  for (i=0; i<nprocs; i++) {
    int age = uptime() - uproc[i].readytime;
    state = states[uproc[i].state];
    printf("%d\t%s\t%d\t%d\t%d\t%d\t%d\t%s\n", uproc[i].pid, state,
                  uproc[i].size,age, uproc[i].priority,uproc[i].cputime, uproc[i].ppid, uproc[i].name);
  }

  exit(0);
}
```

```
xv6 kernel is booting

init: starting sh
$ pexec 5 matmul 5 &; matmul 10 &
$ pexec 10 ps
pid     state           size    ppid    name    priority cputime age
1       sleeping        12288   0       init    0
2       sleeping        16384   1       sh      0
7       runnable        12288   5       matmul  0
6       runnable        12288   1       matmul  0
5       sleeping        12288   1       pexec   0
8       sleeping        12288   2       pexec   0
9       running         12288   8       ps      0       0       177
$ Time: 88 ticks
Time: 184 ticks
pexec 10 ps
exec pexec failed
$ pexec 10 ps
pid     state           size    ppid    name    priority cputime age
1       sleeping        12288   0       init    0
2       sleeping        16384   1       sh      0
11      sleeping        12288   2       pexec   0
12      running         12288   11      ps      0       0       557
$
```