Implement priority scheduling

陈建腾, 1120213582, 单人完成
jianteng.chen@bit.edu.cn

1. 实验要求

In this part of the assignment, you will change the scheduler from a simple round-robin to a priority scheduler. Add a priority value to each process (lets say taking a range between 0 to 31). The range does not matter, it is just a proof of concept. When scheduling from the ready list you will always schedule the highest priority thread/process first.

Add a system call to change the priority of a process. A process can change its priority at any time. If the priority becomes lower than any process on the ready list, you must switch to that process.

To get started, look at the file `proc.c` Implement one of the next three items. If you implement more, each is a 5% bonus. To get credit for a bonus part, you must also develop a user test that will illustrate it.

- To avoid starvation, implement aging of priority. If a process waits increase its priority. When it runs, decrease it. (Possible Bonus 1)
- Implement priority donation/priority inheritence. (Possible Bonus 2)
- Add fields to track the scheduling performance of each process. These values should allow you to compute the turnaround time and wait time for each process. Add a system call to extract these values or alternatively print them out when the process exits. (Possible Bonus 3)

Goals: Understand how the scheduler works. Understand how to implement a scheduling policy and characterize its impact on performance. Understand priority inversion and a possible solution for it.

2. 实验环境

1. Linux Environment

- OS: Ubuntu 22.04.2 LTS on Windows 10 x86_64
- Kernel: 5.15.90.1-microsoft-standard-WSL2
- Shell: zsh 5.8.1
- CPU: AMD Ryzen 7 5800H with Radeon Graphics (8) @ 3.194GHz
- GPU: af4e:00:00.0 Microsoft Corporation Device 008e

2. xv6 Environment

通过以下代码在Linux环境上安装xv6：

```
sudo apt install qemu-system-x86
cd ~
git clone https://github.com/mit-pdos/xv6-public.git xv6
cd xv6
make
echo "add-auto-load-safe-path $HOME/xv6/.gdbinit" > ~/.gdbinit
```

3. Start the xv6

分裂终端，在一个终端内输入以下命令：

```
make-nox-gdb
```

在另一终端内打开gdb即可。

3. 实验过程

1. 修改 `proc.h` 中的PCB内容，加入需要维护的数据结构。

```
// 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 rtime;
  int wtime;
  int stime;
  int priority;
};
```

2. 在 `proc.c` 中做相关修改.

在进程初始化时设置 `p->init_time` 与 `p->priority`，考虑到第一个用户进程不是由fork产生，我们将初始化放置于 `allocproc` 中。

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

  p->priority = 10;
  p->rtime = 0;
  p->wtime = 0;
  p->stime = 0;

  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
```

在父进程调用 `wait` 回收进程相应资源时打印进程信息并清空变量。

```
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->rtime = 0;
        p->wtime = 0;
        p->stime = 0;
        p->priority = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
```

调度器的内层循环用指针遍历整个进程表，修改前每一次找到一个 RUNNABLE 进程时将切换到该进程。修改 scheduler，选择执行更高优先级的进程。每次调度一个进程后返回调度器指针指向下一个 RUNNABLE 进程时，highPriorityProcess 将首先指向这个进程，然后遍历整个进程表，如有更高级的进程，指针会指向接下来第一个优先级最高的进程；如果没有，则下一个进程就是表中下一个可执行进程。

为了避免进程出现"饿死"现象，我们可以监测每个进程的运行时间、等待时间和休眠时间。当一个进程的运行时间过长时，我们可以适当降低其优先级；当一个进程的等待时间过长时，我们可以适当提高其优先级。这样可以确保所有进程都有机会被执行，避免某些进程因为长时间运行或等待而无法得到资源的情况。

```
void
scheduler(void)
{
  struct proc *p;
  struct proc *p1;
  int seg = 500000;
  struct cpu *c = mycpu();
  for(;;){
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state == SLEEPING){
        p->stime++;
        p->rtime = 0;
        p->wtime = 0;
      }
      else if(p->state == RUNNABLE){
        p->wtime++;
        p->rtime = 0;
        p->stime = 0;
        if(p->wtime > seg){
          p->wtime = 0;
          if(p->priority < 15){
            p->priority ++;
          }
        }
      }
      else if(p->state == RUNNING){
        p->rtime++;
        p->stime = 0;
        p->wtime = 0;
        if(p->rtime > seg){
          p->rtime = 0;
          if(p->priority > 5){
            p->priority --;
          }
        }
      }
    }
    // Enable interrupts on this processor.
    sti();
    struct proc *highPriorityProcess = 0;
    // 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;                  //find a runnable process first here
      highPriorityProcess = p;
      for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
        if(p1->state != RUNNABLE)
          continue;
        if ( highPriorityProcess->priority < p1->priority )
         highPriorityProcess = p1;    //find the runnable process with highest priority
      }
      p = highPriorityProcess;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;   //change the state from RUNNABLE to 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);

  }
}
```

3. 实现系统调用 chpr 与 cps 函数，用于修改对应进程的优先级与打印现有进程信息。并类似 lab3 中操作添加相应代码段。

```
int
chpr(int pidNum, int priority)
{
  struct proc *p;
  acquire(&ptable.lock);
  for(p=ptable.proc;p < &ptable.proc[NPROC];p++)
  {
    if(p->pid == pidNum)
    {
        p->priority = priority;
        break;
    }
  }
  release(&ptable.lock);
  return pidNum;
}
int
cps()
{
    struct proc*p;
    sti();
    acquire(&ptable.lock);
    cprintf("name \t pid \t state \t       priority\n");
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state ==SLEEPING)
          cprintf("%s \t %d \t SLEEPING \t %d\n",p->name,p->pid,p->priority);
      else if(p->state ==RUNNING)
          cprintf("%s \t %d \t RUNNING \t %d\n",p->name,p->pid,p->priority-5);
      else if(p->state ==RUNNABLE)
          cprintf("%s \t %d \t RUNNABLE \t %d\n",p->name,p->pid,p->priority+5);

    }

    release(&ptable.lock);

    return 23;
}
```

4. 创建系统调用，用于修改进程优先级与打印进程信息。

```
// sysproc.c
int
sys_chpr(void)
{
    int pid,pr;
    if(argint(0,&pid) < 0)
    {
      return -1;
    }
    if(argint(1,&pr)<0)
    {
      return -1;
    }
    return chpr(pid,pr);
}

int sys_cps(void)
{
  return cps();
}
```

5. 创建用户态程序 `myfork.c` , `cpr.c` , `ps.c` 并对应修改 `Makefile` 文件

```
// myfork.c

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



int
main(int argc, char *argv[])
{
    int k, n, id;
    int a = 0;
    double b = 1;
    if(argc < 2) //if user does not provide a value
        n = 1;     else
        n = atoi ( argv[1] );

    //if user enters a negative value for no. of forks
    //or a number above 20 we default it to 2
    if(n<0 || n>20) {n = 2;}      id = 0;
    for ( k = 0; k < n; k++ ) {
        id = fork ();
        if ( id < 0 ) {
            printf(1, "%d failed in fork!\n", getpid() );
        }else if(id == 0 ) { // child
            printf(1, "Child %d created\n",getpid() );
            //USELESS calculations
            for ( a = 0; a < 1000000000001; a += 1 )
            {
                b+=0.001;
                b = b * 101010101.1 - 0.005 / 10.0;
            }
            //USELESS calculations end
            break;
        }else { //parent
            printf(1, "Parent %d creating child %d\n", getpid(), id );
            wait ();
        }
    }
    exit();
}
```

```
// cpr.c

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

int
main(int argc, char *argv[])
{
    int priority, pid;
    if(argc < 3 ){
        printf(2,"Invalid command!\n");
        printf(2, "Usage: cpr pid priority\n" );
    }else
    {
        pid = atoi ( argv[1] );
        priority = atoi ( argv[2] );
        if ( priority < 0 || priority > 31 ) {
            printf(2, "Priority needs to be between 0-31. \n" );
        }else
        {
            chpr ( pid, priority );
        }
    }
    exit();
}
```

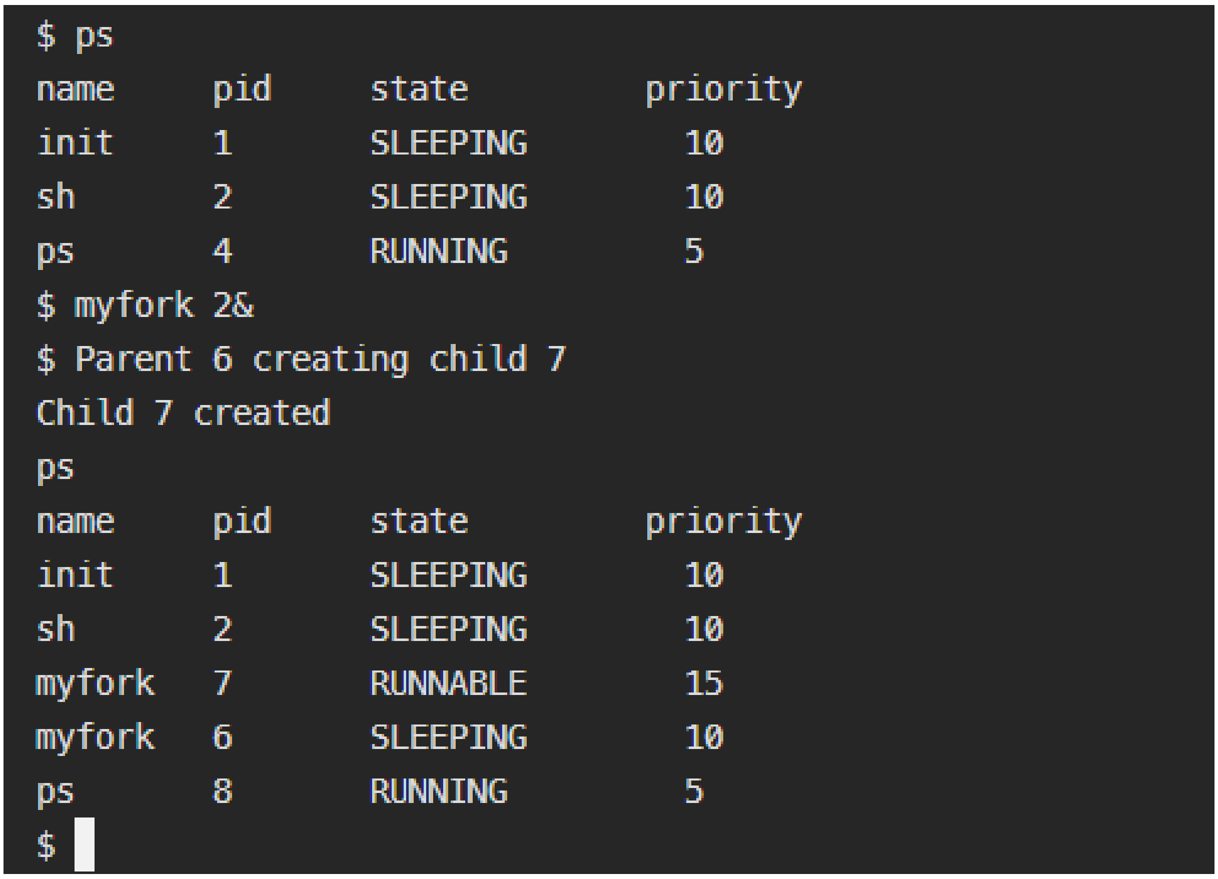
```
// ps.c

#include "types.h"
```

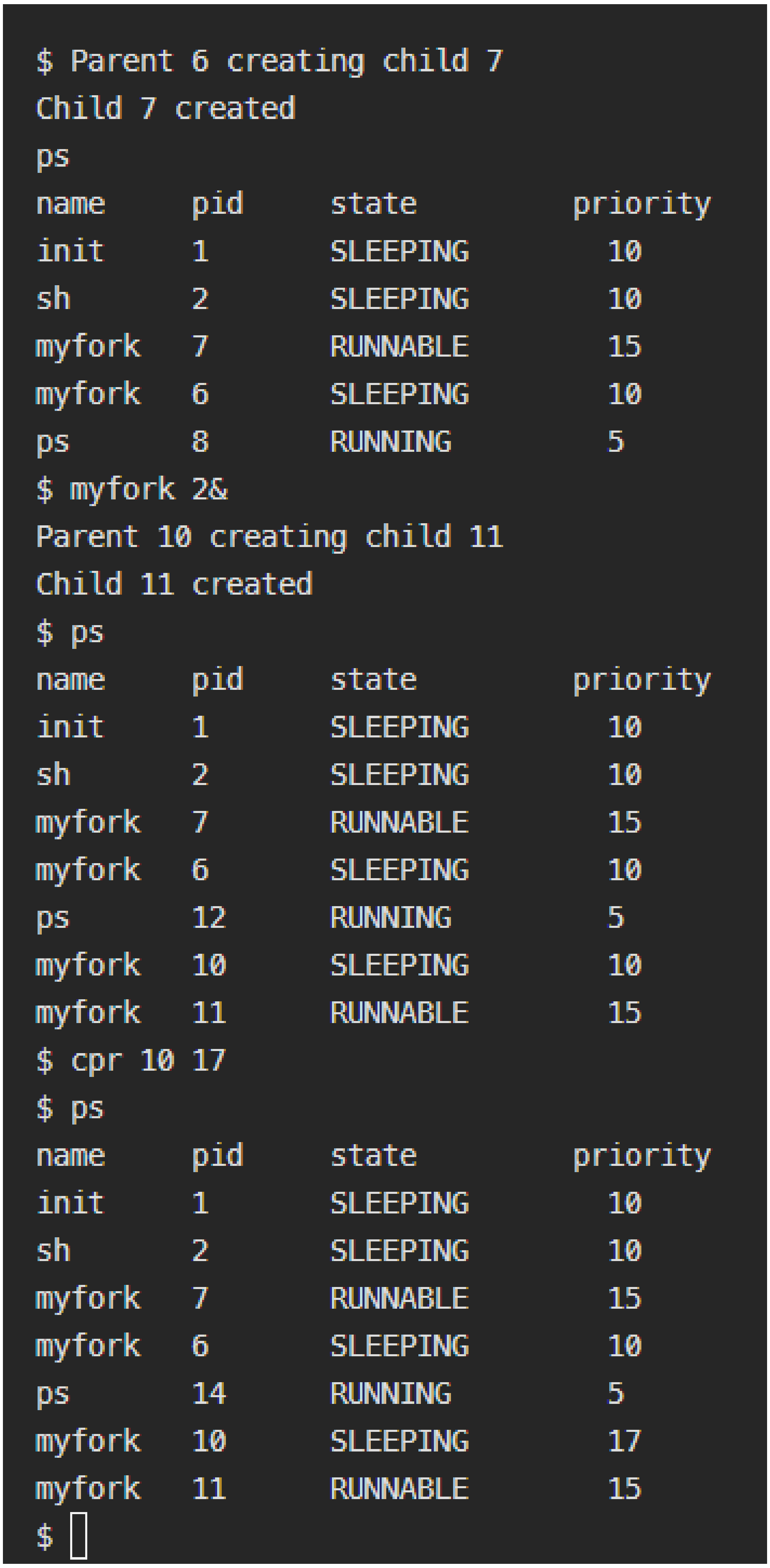
```c
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int
main(int args,char *argv[])
{
    cps();
    exit();
}
```

4. 运行结果



```
$ ps
name      pid     state          priority
init      1       SLEEPING         10
sh        2       SLEEPING         10
ps        4       RUNNING          5
$ myfork 2&
$ Parent 6 creating child 7
Child 7 created
ps
name      pid     state          priority
init      1       SLEEPING         10
sh        2       SLEEPING         10
myfork    7       RUNNABLE         15
myfork    6       SLEEPING         10
ps        8       RUNNING          5
$
```



```
$ Parent 6 creating child 7
Child 7 created
ps
name      pid      state         priority
init      1        SLEEPING        10
sh        2        SLEEPING        10
myfork    7        RUNNABLE        15
myfork    6        SLEEPING        10
ps        8        RUNNING         5
$ myfork 2&
Parent 10 creating child 11
Child 11 created
$ ps
name      pid      state         priority
init      1        SLEEPING        10
sh        2        SLEEPING        10
myfork    7        RUNNABLE        15
myfork    6        SLEEPING        10
ps        12       RUNNING         5
myfork    10       SLEEPING        10
myfork    11       RUNNABLE        15
$ cpr 10 17
$ ps
name      pid      state         priority
init      1        SLEEPING        10
sh        2        SLEEPING        10
myfork    7        RUNNABLE        15
myfork    6        SLEEPING        10
ps        14       RUNNING         5
myfork    10       SLEEPING        17
myfork    11       RUNNABLE        15
$ 
```