Yonatan Erez

HW4

Ps.c:

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

int main(int argc, char const *argv[])
{
    ps();
    exit();
}

```

Creates a terminal interface to the user.

Demosched.c:

demosched.c > main(int, char const * [])

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

int main(int argc, char const *argv[])
{
    getprio();
    ps();
    long num = 1 << 28;
    int pid = fork();
    if (pid==-1){
        printf(1, "\nError in fork\n");
        exit();
    }
    if (pid==0){
        setprio(7);
        ps();
        for (volatile long l1=0; l1<num; l1++){
            l1++;
            l1--;
        }
        printf(1, "\nChild exit\n");
        exit();
    }
    else{
        setprio(6);
        ps();
        for (volatile long l2=0; l2<num; l2++){
            l2++;
            l2--;
        }
        printf(1, "\nParent is waiting for child\n");
        wait();
    }
    ps();
    printf(1, "Parent exit\n");
    exit();
}

```

The script uses getprio, setprio and ps (for debug purposes).

```
init: starting sh
$ demosched

getprio: pid=3, priority=0

pid      state           priority        ticks_counter    required_ticks
1        SLEEPING        0               5                1
2        SLEEPING        0               3                1
3        RUNNING         0               1                1

setprio: pid=3, priority=6

pid      state           priority        ticks_counter    required_ticks
1        SLEEPING        0               5                1
2        SLEEPING        0               3                1
3        RUNNING         6               2                64
4        RUNNABLE        0               0                1

setprio: pid=4, priority=7

pid      state           priority        ticks_counter    required_ticks
1        SLEEPING        0               5                1
2        SLEEPING        0               3                1
3        RUNNABLE        6               128              64
4        RUNNING         7               0                128

Parent is waiting for child

Child exit
```

```
pid      state           priority        ticks_counter    required_ticks
1        SLEEPING        0               5                1
2        SLEEPING        0               3                1
3        RUNNING         6               128              64
Parent exit
```

Let's add another ps call.

```
pid      state           priority        ticks_counter    required_ticks
1        SLEEPING        0               5                1
2        SLEEPING        0               3                1
5        RUNNING         0               2                1
$
```

As we can see the sequence is:

1. Boot.
2. Demosched.
3. Getprio is called by the parent.
4. Fork (new process is added to that table).
5. Setprio(6) is called by the parent.
6. Setprio(6) is called by the child.
7. Parent is running till the required ticks are over.
8. Parent finished calculations and waits for child.
9. Child is running till the required ticks are over.
10. Child finished calculations and exits.
11. Parent exits.
12. Ps is called from the terminal and both parent and child do not exist.

Makefile:

```
181         _usertests\
182         _wc\
183         _zombie\
184         _demosched\
185         _ps\
186
187     fs.img: mkfs README $(UPROGS)
188         ./mkfs fs.img README $(UPROGS)
```

```
251
252     EXTRA=\
253         mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
254         ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
255         printf.c umalloc.c demosched.c ps.c\
256         README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
257         .gdbinit.tmpl gdbutil\
```

Both PS and Makefile were added to the makefile compilation lines.

Defs.h:

```
122  void            yield(void);
123  int             proc_ps(void);
124  int             proc_getprio(int);
125  int             proc_setprio(int, int);
126
```

The new syscalls implementation are decalred.

Proc.c:

```
545
546  int
547  proc_getprio(int pid)
548  {
549    int priority = -1;
550    struct proc* p;
551    acquire(&ptable.lock);
552    for (p=ptable.proc; p<&ptable.proc[NPROC]; p++){
553      if (p->pid == pid){
554        priority = p->priority;
555        cprintf("\n");
556        cprintf("getprio: pid=%d, priority=%d\n", p->pid, p->priority);
557        break;
558      }
559    }
560    release(&ptable.lock);
561    return priority;
562  }
```

```
566
567  int
568  proc_setprio(int pid, int priority)
569  {
570    struct proc* p;
571    acquire(&ptable.lock);
572    for (p=ptable.proc; p<&ptable.proc[NPROC]; p++){
573      if (p->pid == pid){
574        p->priority = priority;
575        cprintf("\n");
576        cprintf("setprio: pid=%d, priority=%d\n", p->pid, p->priority);
577        break;
578      }
579    }
580    release(&ptable.lock);
581    return 0;
582  }
```

```c
int
proc_ps()
{
  struct proc* p;
  acquire(&ptable.lock);
  cprintf("\n");
  cprintf("pid \t state \t         priority \t ticks_counter \t  required_ticks\n");
  for (p=ptable.proc; p<&ptable.proc[NPROC]; p++){
    if (p->state == UNUSED){
      continue;
    }
    if (p->state == EMBRYO){
      cprintf("%d \t EMBRYO     \t %d \t\t %d \t          %d \n", p->pid, p->priority, p->ticks_counter, 1 << p->priori
      continue;
    }
    if (p->state == SLEEPING){
      cprintf("%d \t SLEEPING   \t %d \t\t %d \t          %d \n", p->pid, p->priority, p->ticks_counter, 1 << p->priori
      continue;
    }
    if (p->state == RUNNABLE){
      cprintf("%d \t RUNNABLE   \t %d \t\t %d \t          %d \n", p->pid, p->priority, p->ticks_counter, 1 << p->priori
      continue;
    }
    if (p->state == RUNNING){
      cprintf("%d \t RUNNING    \t %d \t\t %d \t          %d \n", p->pid, p->priority, p->ticks_counter, 1 << p->priori
      continue;    }
    if (p->state == ZOMBIE){
      cprintf("%d \t ZOMBIE     \t %d \t\t %d \t          %d \n", p->pid, p->priority, p->ticks_counter, 1 << p->priori
      continue;
    }
  }
  release(&ptable.lock);
  return 0;
}
```

The implementations are added to proc.c file.

```c
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      volatile int* tick_counter_ptr = &p->ticks_counter;
      while (*tick_counter_ptr < (1 << p->priority))
      {
        if (p->state != RUNNING){
          break;
        }
        continue;
      }

      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```

The scheduler picks the lowest pid which did not finish ticking.

Proc.h:

```
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52    int priority;                // Process priority
53    int ticks_counter;           // Process ticks counter
54  };
```

The new attributes are added to the proc struct.

Syscall.c:

```
104  extern int sys_write(void);
105  extern int sys_uptime(void);
106  extern int sys_getprio(void);
107  extern int sys_setprio(void);
108  extern int sys_ps(void);
109
```

```
129  [SYS_link]    sys_link,
130  [SYS_mkdir]   sys_mkdir,
131  [SYS_close]   sys_close,
132  [SYS_getprio] sys_getprio,
133  [SYS_setprio] sys_setprio,
134  [SYS_ps]      sys_ps
135  };
136
137  void
```

Added the new syscalls to the list.

Proc.h:

```
19  #define SYS_unlink 18
20  #define SYS_link   19
21  #define SYS_mkdir  20
22  #define SYS_close  21
23  #define SYS_getprio 22
24  #define SYS_setprio 23
25  #define SYS_ps 24
26
```

The syscalls number are added.

Sysproc.c:

```
93
94  int
95  sys_getprio(void){
96    return proc_getprio(myproc()->pid);
97  }
98
99
100 int
101 sys_setprio(void){
102   int priority;
103   if (argint(0, &priority) < 0){
104     return -1;
105   }
106   return proc_setprio(myproc()->pid, priority);
107 }
108
109
110 int
111 sys_ps(void){
112   return proc_ps();
113 }
```

The implementations are being called by the syscalls.

Trap.c:

```
102
103   // Force process to give up CPU on clock tick.
104   // If interrupts were on while locks held, would need to check nlock.
105   if(myproc() && myproc()->state == RUNNING &&
106      tf->trapno == T_IRQ0+IRQ_TIMER){
107     myproc()->ticks_counter++;
108     if (myproc()->ticks_counter > (1<<7)){
109       myproc()->ticks_counter = 1<<7;
110       yield();
111     }
112   }
113
114   // Check if the process has been killed since we yielded
115   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
116     exit();
117 }
118
```

On every iteration the ticks_counter is incremented till max value of $2^7 = 128$.

If the counter reaches the $counter\_ticks = 2^{priority}$.

User.h:

```
22  int getpid(void);
23  char* sbrk(int);
24  int sleep(int);
25  int uptime(void);
26  int getprio(void);
27  int setprio(int);
28  int ps(void);
29
30  // ulib.c
```

The new system calls are added to the user space.

Usys.S:

```
30  SYSCALL(sleep)
31  SYSCALL(uptime)
32  SYSCALL(getprio)
33  SYSCALL(setprio)
34  SYSCALL(ps)
35
```

The syscalls are added to the list.