Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 430-01

10/27/2021

Implementing process priority

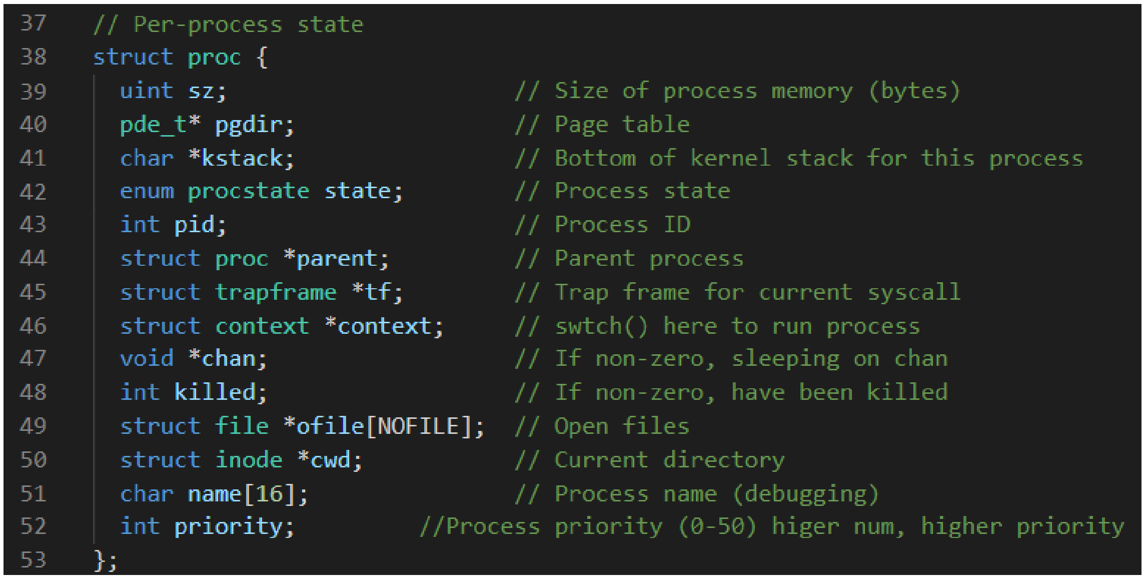
Purpose

The purpose of this lab is to give the students a way of assigning, changing, and viewing a process's priority. Also, students will reinforce some of the knowledge they have already learned by implementing new user and system calls. Furthermore, the system call that is to be implemented requires the use of a command line argument, which was learned in the previous labs.

Code

ps command

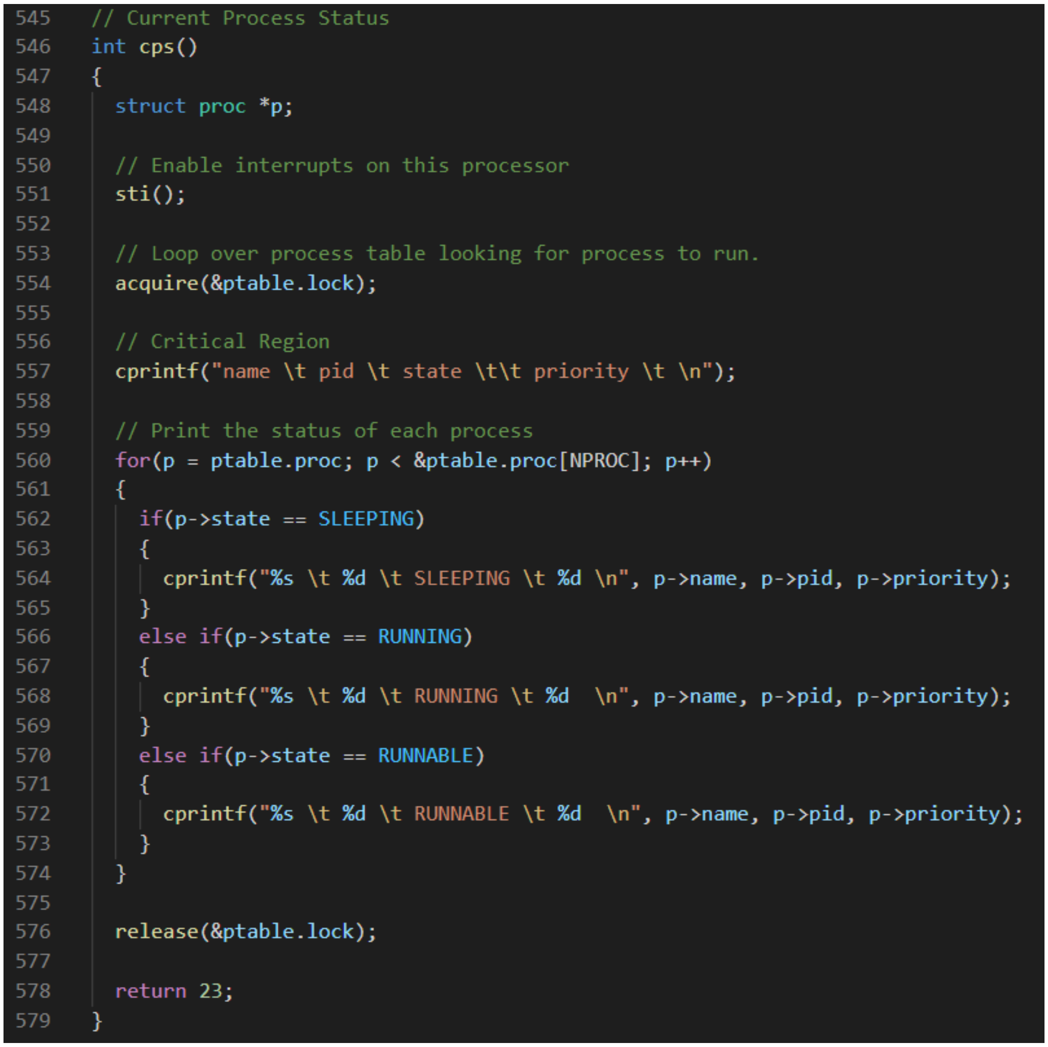
proc.h



```
// 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 priority;         //Process priority (0-50) higer num, higher priority
};
```

This file has been modified so that processes now have a priority level.

proc.c

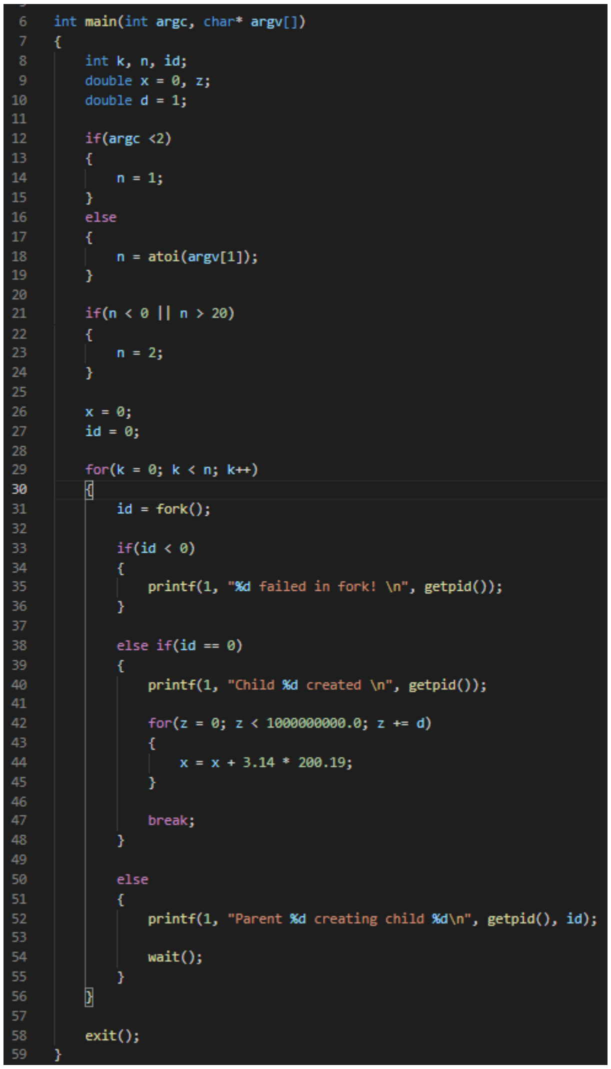


```c
// Current Process Status
int cps()
{
  struct proc *p;

  // Enable interrupts on this processor
  sti();

  // Loop over process table looking for process to run.
  acquire(&ptable.lock);

  // Critical Region
  cprintf("name \t pid \t state \t\t priority \t \n");

  // Print the status of each process
  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);
    }
    else if(p->state == RUNNABLE)
    {
      cprintf("%s \t %d \t RUNNABLE \t %d  \n", p->name, p->pid, p->priority);
    }
  }

  release(&ptable.lock);

  return 23;
}
```

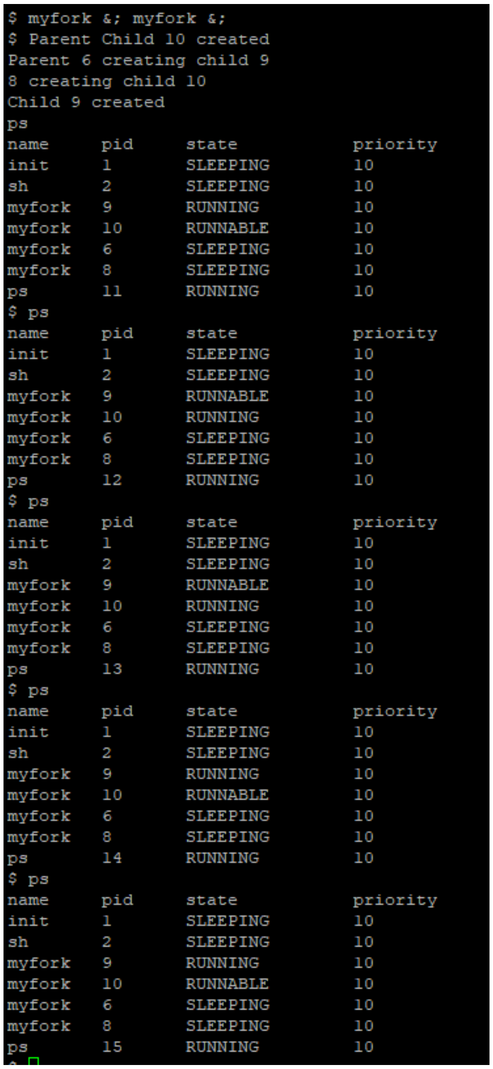
the cps() function has been modified to now display each processes priority level on execution.

myfork command

myfork.c



```c
int main(int argc, char* argv[])
{
    int k, n, id;
    double x = 0, z;
    double d = 1;

    if(argc <2)
    {
        n = 1;
    }
    else
    {
        n = atoi(argv[1]);
    }

    if(n < 0 || n > 20)
    {
        n = 2;
    }

    x = 0;
    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)
        {
            printf(1, "Child %d created \n", getpid());

            for(z = 0; z < 1000000000.0; z += d)
            {
                x = x + 3.14 * 200.19;
            }

            break;
        }

        else
        {
            printf(1, "Parent %d creating child %d\n", getpid(), id);

            wait();
        }
    }

    exit();
}
```

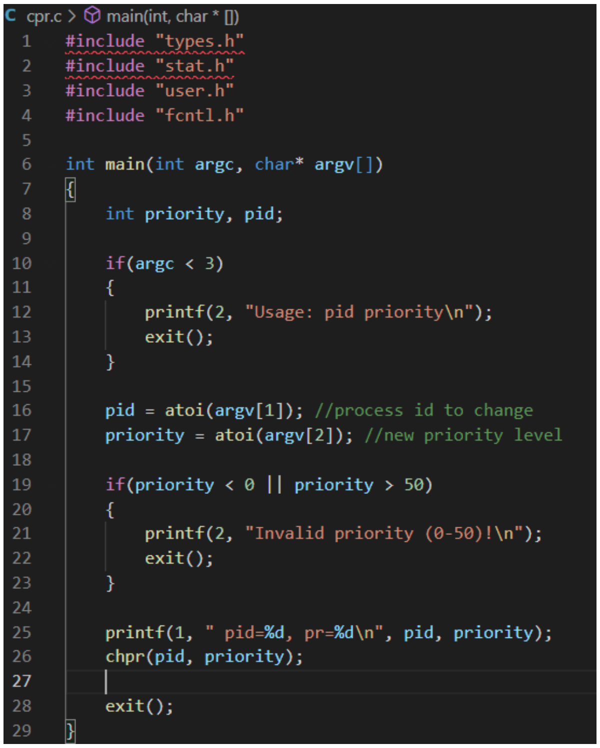


```
$ myfork &; myfork &;
$ Parent Child 10 created
Parent 6 creating child 9
8 creating child 10
Child 9 created
ps
name     pid     state           priority
init     1       SLEEPING        10
sh       2       SLEEPING        10
myfork   9       RUNNING         10
myfork   10      RUNNABLE        10
myfork   6       SLEEPING        10
myfork   8       SLEEPING        10
ps       11      RUNNING         10
$ ps
name     pid     state           priority
init     1       SLEEPING        10
sh       2       SLEEPING        10
myfork   9       RUNNABLE        10
myfork   10      RUNNING         10
myfork   6       SLEEPING        10
myfork   8       SLEEPING        10
ps       12      RUNNING         10
$ ps
name     pid     state           priority
init     1       SLEEPING        10
sh       2       SLEEPING        10
myfork   9       RUNNABLE        10
myfork   10      RUNNING         10
myfork   6       SLEEPING        10
myfork   8       SLEEPING        10
ps       13      RUNNING         10
$ ps
name     pid     state           priority
init     1       SLEEPING        10
sh       2       SLEEPING        10
myfork   9       RUNNING         10
myfork   10      RUNNABLE        10
myfork   6       SLEEPING        10
myfork   8       SLEEPING        10
ps       14      RUNNING         10
$ ps
name     pid     state           priority
init     1       SLEEPING        10
sh       2       SLEEPING        10
myfork   9       RUNNING         10
myfork   10      RUNNABLE        10
myfork   6       SLEEPING        10
myfork   8       SLEEPING        10
ps       15      RUNNING         10
```

myfork will duplicate a parent process n number of times, up to 20. Each child process will be delayed with its execution because it will perform a trivial calculation. The parent must then wait for the child's operation before continuing. This will keep the processes active, allowing for some tests on scheduling and process creation.

cpr command

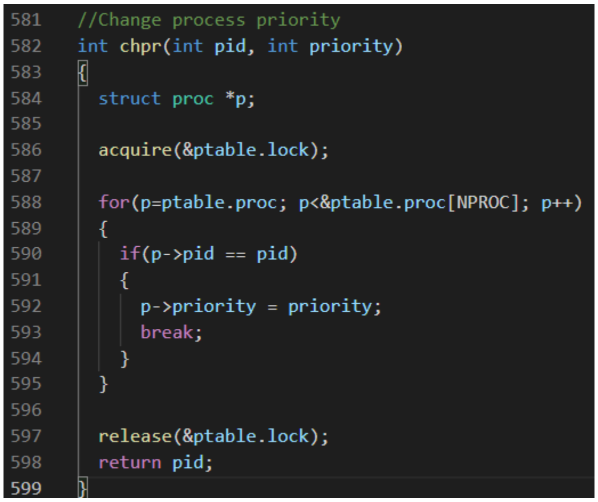
cpr.c



```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, "Usage: pid priority\n");
        exit();
    }

    pid = atoi(argv[1]); //process id to change
    priority = atoi(argv[2]); //new priority level

    if(priority < 0 || priority > 50)
    {
        printf(2, "Invalid priority (0-50)!\n");
        exit();
    }

    printf(1, " pid=%d, pr=%d\n", pid, priority);
    chpr(pid, priority);

    exit();
}
```

This file will take in two arguments and change a processes priority. The first argument is the process id to change, and the second argument is the priority to assign to the process. If both values are valid, then chpr will be called to assign the new priority.

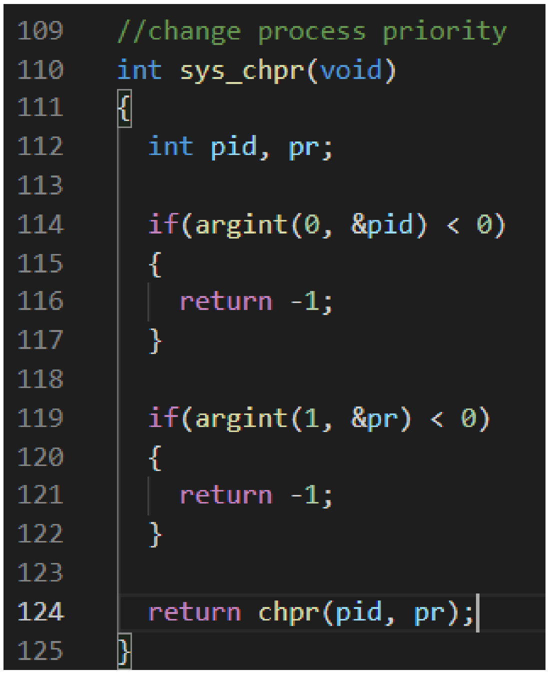
proc.c



```c
//Change process priority
int chpr(int pid, int priority)
{
  struct proc *p;

  acquire(&ptable.lock);

  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++)
  {
    if(p->pid == pid)
    {
      p->priority = priority;
      break;
    }
  }

  release(&ptable.lock);
  return pid;
}
```

The function call that acquires a lock on a process and changes its priority. It will return the pid of the current process. This function is used internally by the OS and not directly called by the user.

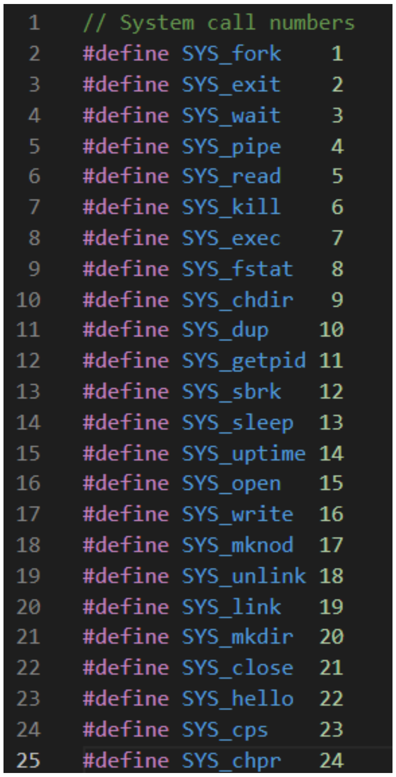
sysproc.c



```c
//change process priority
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);
}
```

The actual system call by the kernel to change a process' priority. It changes the priority if two valid arguments are supplied using the argint() function. Then it calls chpr().

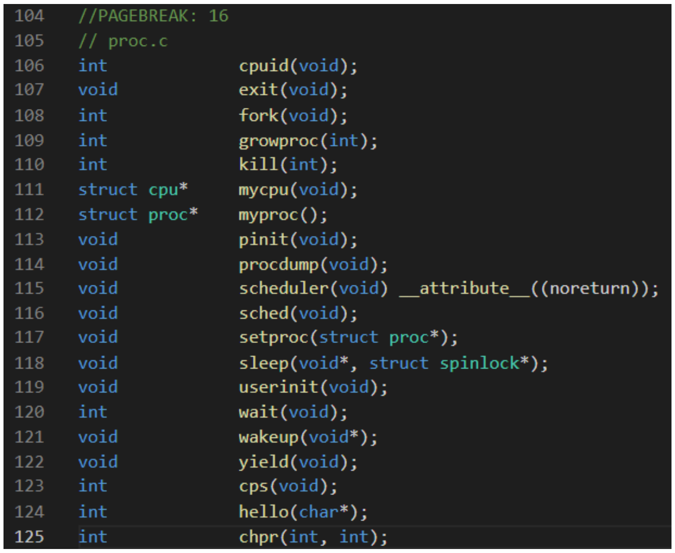
syscall.h



```c
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_hello  22
#define SYS_cps    23
#define SYS_chpr   24
```

The SYS_chpr is added as a usable system call with the call number 24.

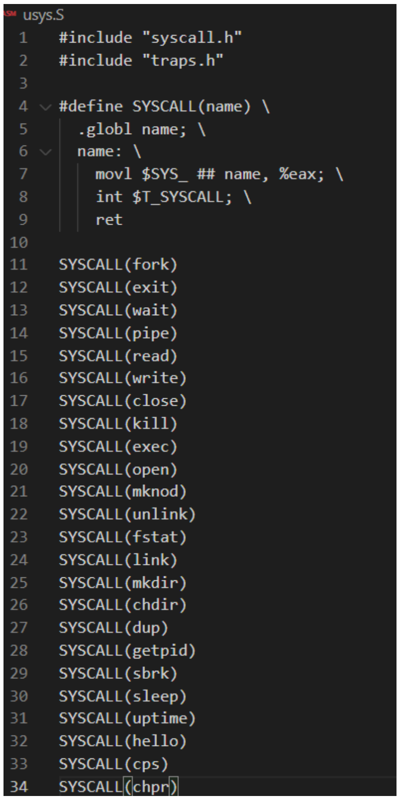
defs.h



```
//PAGEBREAK: 16
// proc.c
int             cpuid(void);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            setproc(struct proc*);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);
int             cps(void);
int             hello(char*);
int             chpr(int, int);
```

chpr() is declared to be a function that will be defined in proc.c.

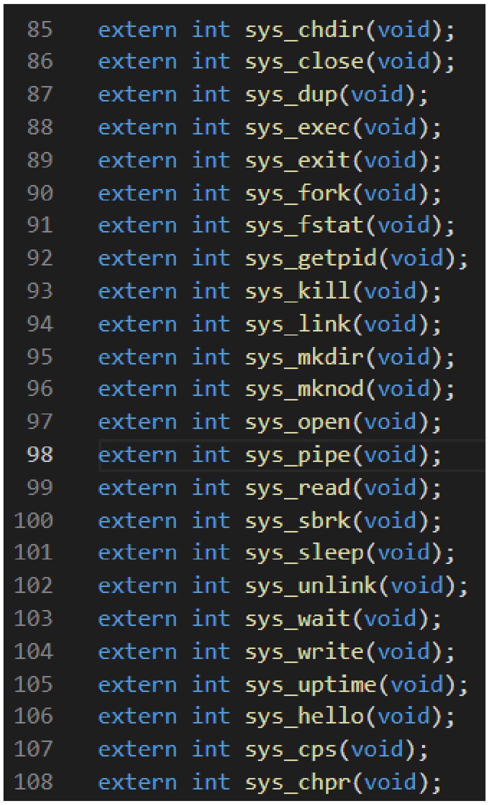
usys.S



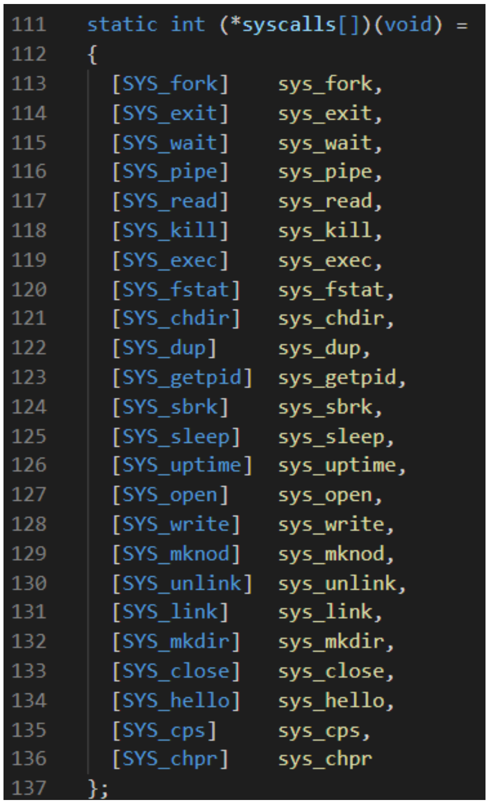
```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(hello)
SYSCALL(cps)
SYSCALL(chpr)
```

chpr is added to the list of systemcalls.

syscall.c



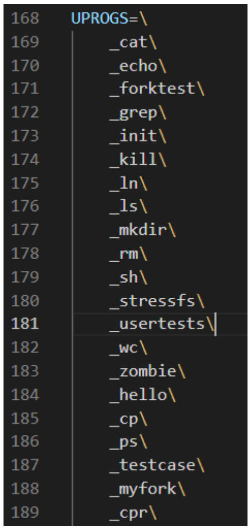
```
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_hello(void);
extern int sys_cps(void);
extern int sys_chpr(void);
```



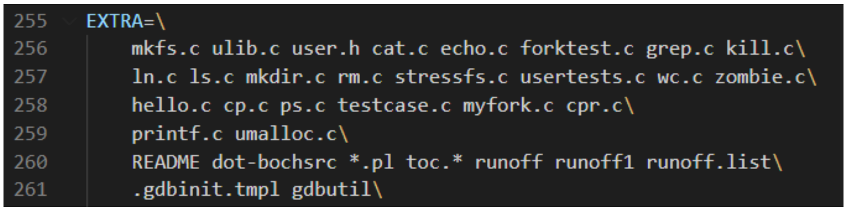
```
static int (*syscalls[])(void) =
{
  [SYS_fork]    sys_fork,
  [SYS_exit]    sys_exit,
  [SYS_wait]    sys_wait,
  [SYS_pipe]    sys_pipe,
  [SYS_read]    sys_read,
  [SYS_kill]    sys_kill,
  [SYS_exec]    sys_exec,
  [SYS_fstat]   sys_fstat,
  [SYS_chdir]   sys_chdir,
  [SYS_dup]     sys_dup,
  [SYS_getpid]  sys_getpid,
  [SYS_sbrk]    sys_sbrk,
  [SYS_sleep]   sys_sleep,
  [SYS_uptime]  sys_uptime,
  [SYS_open]    sys_open,
  [SYS_write]   sys_write,
  [SYS_mknod]   sys_mknod,
  [SYS_unlink]  sys_unlink,
  [SYS_link]    sys_link,
  [SYS_mkdir]   sys_mkdir,
  [SYS_close]   sys_close,
  [SYS_hello]   sys_hello,
  [SYS_cps]     sys_cps,
  [SYS_chpr]    sys_chpr
};
```

The system call that calls chpr is declared to be defined elsewhere and the system call itself is added to the array of system calls.

Makefile



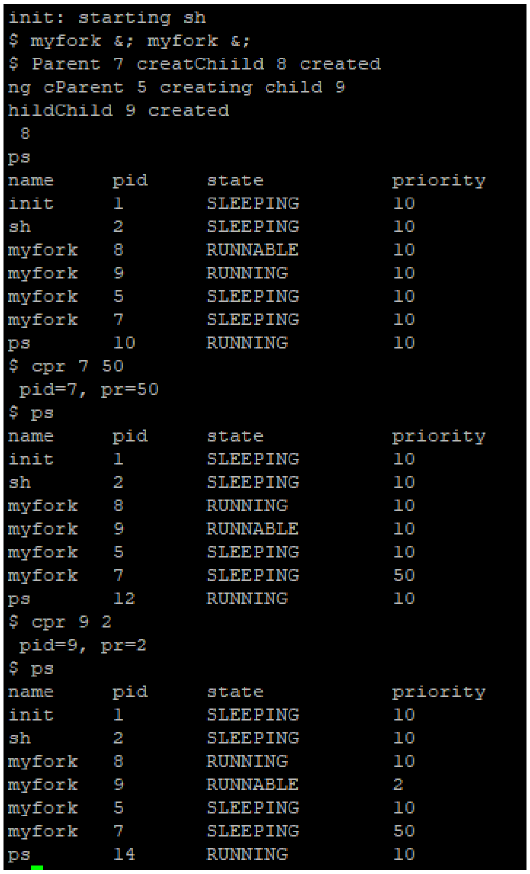
```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_hello\
	_cp\
	_ps\
	_testcase\
	_myfork\
	_cpr\
```



```
EXTRA=\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
	hello.c cp.c ps.c testcase.c myfork.c cpr.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

cpr.c is added to the list of files that need to be compiled and linked.

Code Execution



```
init: starting sh
$ myfork &; myfork &;
$ Parent 7 creatChiild 8 created
ng cParent 5 creating child 9
hildChild 9 created
 8
ps
name     pid     state          priority
init     1       SLEEPING       10
sh       2       SLEEPING       10
myfork   8       RUNNABLE       10
myfork   9       RUNNING        10
myfork   5       SLEEPING       10
myfork   7       SLEEPING       10
ps       10      RUNNING        10
$ cpr 7 50
 pid=7, pr=50
$ ps
name     pid     state          priority
init     1       SLEEPING       10
sh       2       SLEEPING       10
myfork   8       RUNNING        10
myfork   9       RUNNABLE       10
myfork   5       SLEEPING       10
myfork   7       SLEEPING       50
ps       12      RUNNING        10
$ cpr 9 2
 pid=9, pr=2
$ ps
name     pid     state          priority
init     1       SLEEPING       10
sh       2       SLEEPING       10
myfork   8       RUNNING        10
myfork   9       RUNNABLE       2
myfork   5       SLEEPING       10
myfork   7       SLEEPING       50
ps       14      RUNNING        10
```

In the execution, first myfork is called twice to create some processes that are running and want to run. Then ps is called to show what processes are running and what the process' priority are. Finally, some of the processes have their priorities changed by using their pid and a new priority value. The results of the change are reflected with the use of the ps command again.

Conclusion

This lab was useful for understanding how the OS handles processes. It also was good to continue working with system calls and user calls. By repeating these tasks, I am more likely to remember how to do them in the future. Again, it was good to use a command line argument with a system call because this will be useful for other system calls. Lastly, this lab was most useful for setting up scheduling algorithms. It didn't directly do anything with the scheduler, but if the scheduler does start to use the priority level of processes, then processes can be deemed more important than others for whatever reason.