MNNIT ALLAHABAD

OPERATING SYSTEM

SUBMITTED BY :
NAME : SHISHU
ROLL/REG ID : 2020CA089
SUBMIT DATE : 19/12/2021

SUBMITTED TO :
TEACHER _______________
DEPPT : COMPUTER SCIENCE
DEADLINE: 19/12/2021

Question 1.Adding Process Priority: In this lab, we will walk you through the steps of adding a priority attribute to a process in xv6 and changing its value. We assign a process with a value between 0 and 20, the smaller the value, the higher the priority. The default value is 10.

i. Add priority to struct proc in proc.h

ii. Assign default priority in allocproc() in proc.c

iii. Modify cps() in proc.c discussed in the last lab to include the print the priority

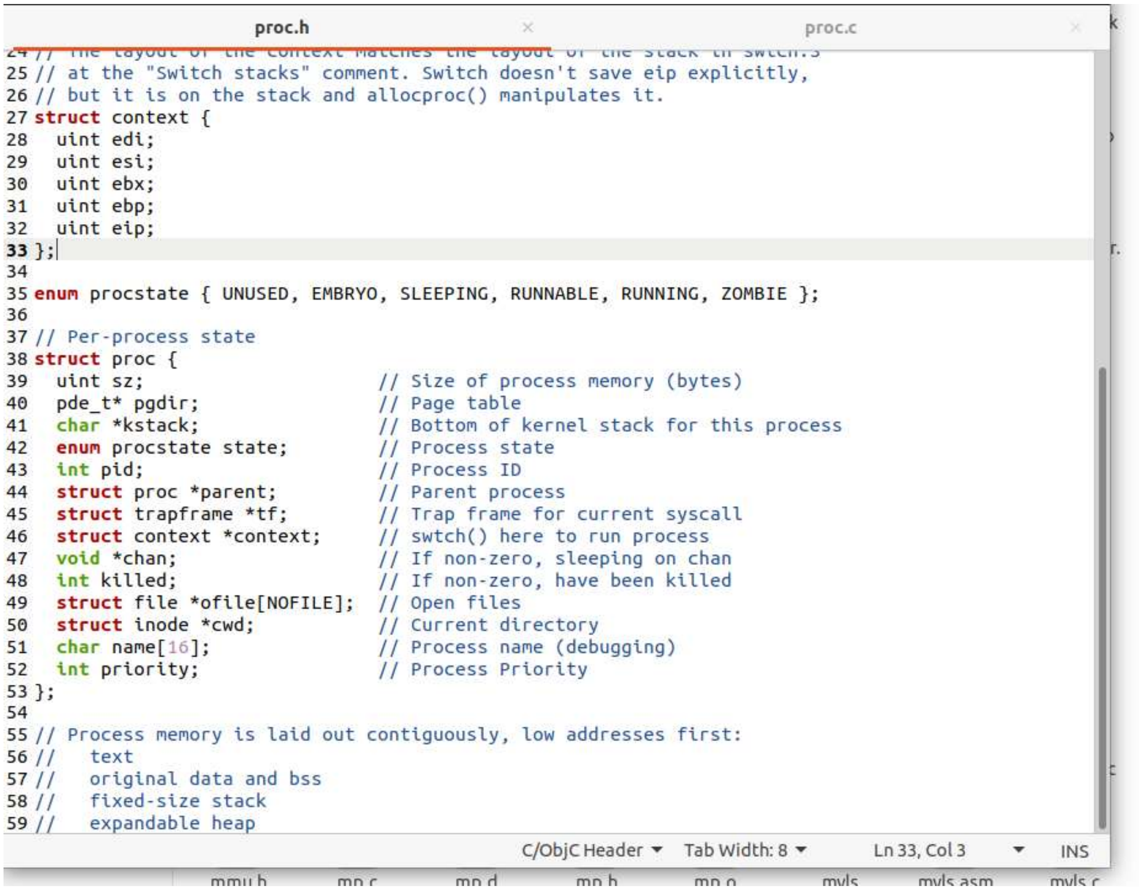
iv. Write a dummy program named foo.c that creates some child processes and consumes somecomputing time

v. Add the function chpr() (meaning change priority) in proc.c

vi. Add sys_chpr() in sysproc.c

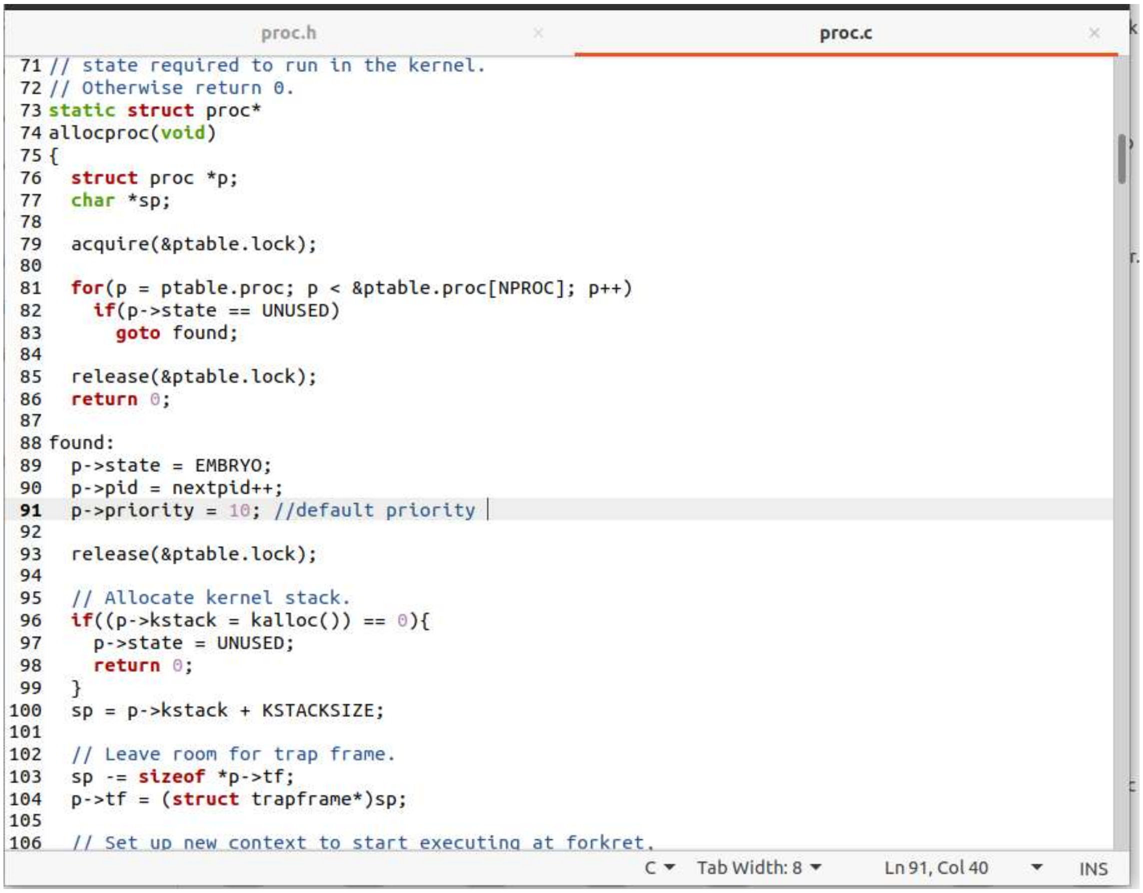
vii. Add chpr() as a system call to xv6

viii. Create the user file nice.c with which calls chprix. Test nice



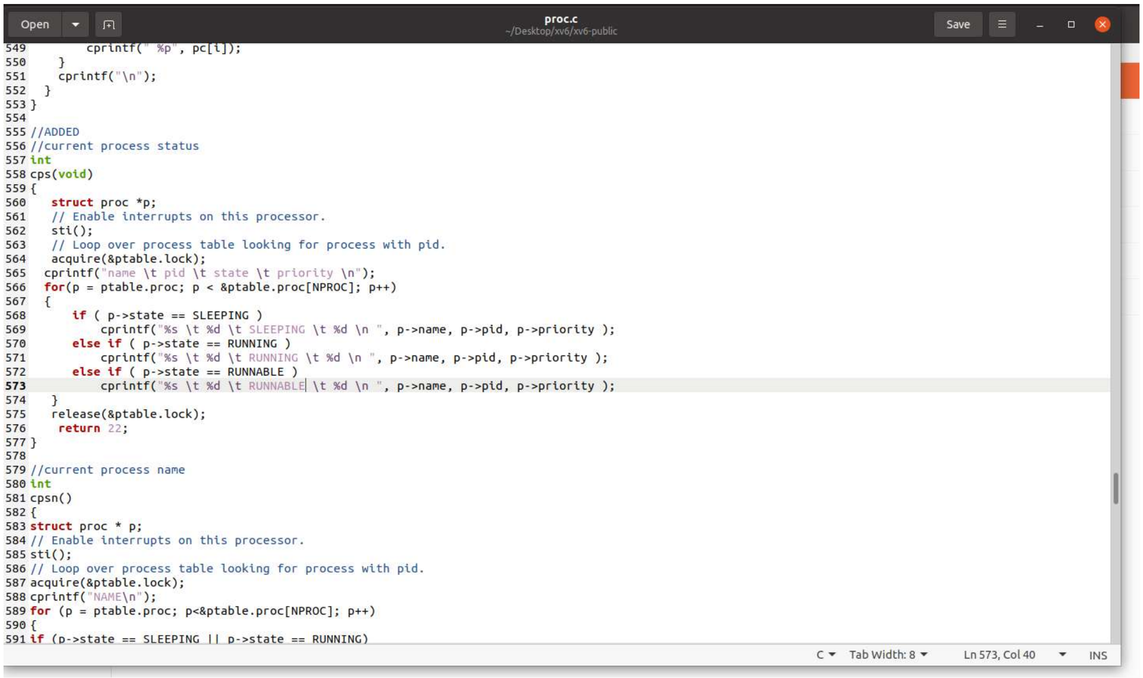
```c
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// 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
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
```

Adding priority in proc.h



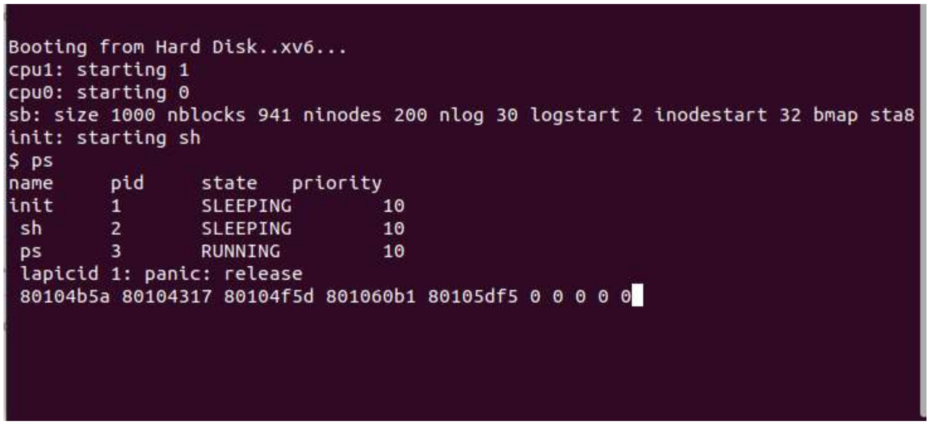
```c
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority = 10; //default priority

  release(&ptable.lock);

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

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret.
```

Assigning default priority in allocproc()



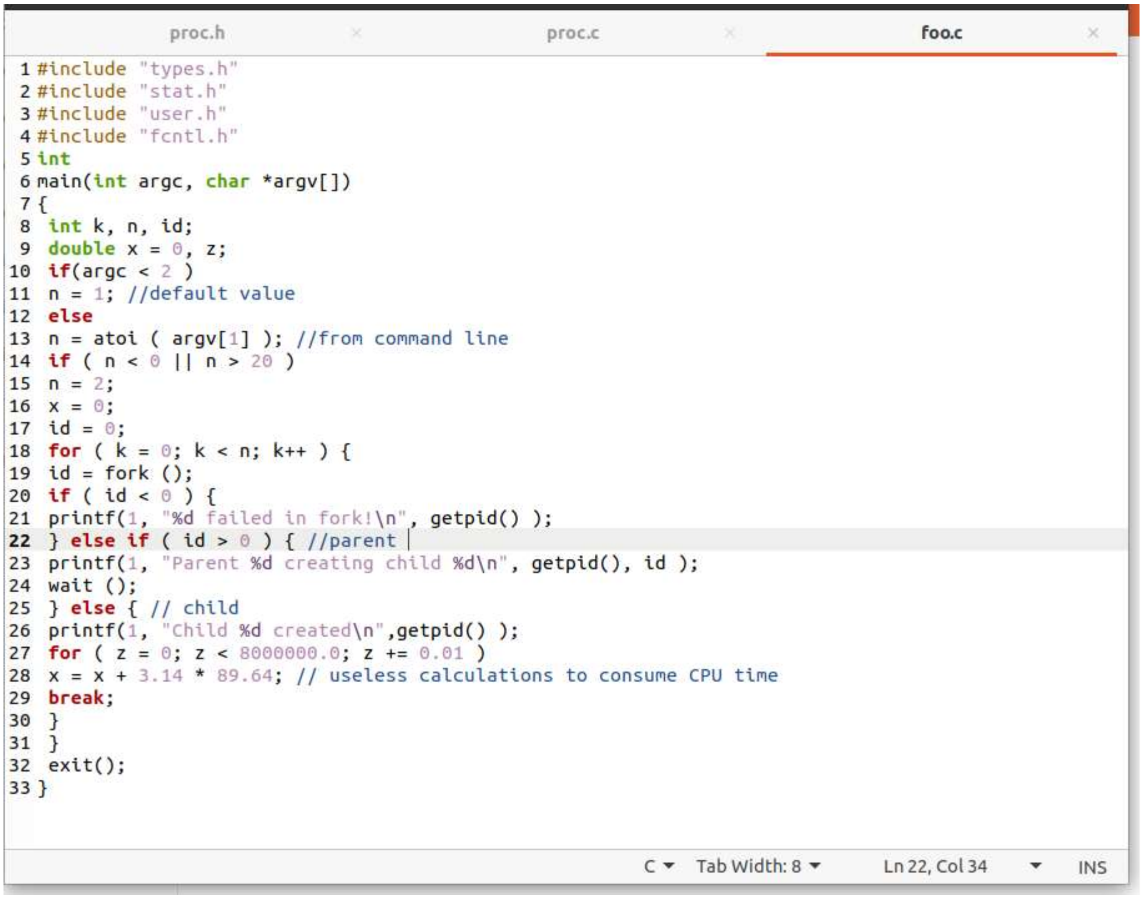
```c
        cprintf(" %p", pc[i]);
    }
    cprintf("\n");
  }
}

//ADDED
//current process status
int
cps(void)
{
   struct proc *p;
   // Enable interrupts on this processor.
   sti();
   // Loop over process table looking for process with pid.
   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 );
      else if ( p->state == RUNNABLE )
          cprintf("%s \t %d \t RUNNABLE \t %d \n ", p->name, p->pid, p->priority );
   }
   release(&ptable.lock);
    return 22;
}

//current process name
int
cpsn()
{
struct proc * p;
// Enable interrupts on this processor.
sti();
// Loop over process table looking for process with pid.
acquire(&ptable.lock);
cprintf("NAME\n");
for (p = ptable.proc; p<&ptable.proc[NPROC]; p++)
{
if (p->state == SLEEPING || p->state == RUNNING)
```

Modifying the cps in proc.c that now also prints the priority



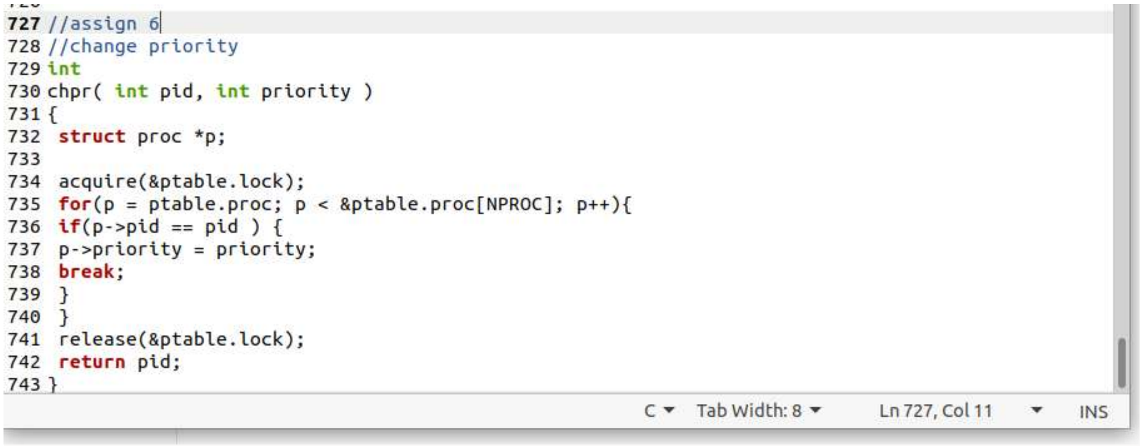
```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ ps
name     pid     state    priority
init     1       SLEEPING        10
 sh      2       SLEEPING        10
 ps      3       RUNNING         10
 lapicid 1: panic: release
 80104b5a 80104317 80104f5d 801060b1 80105df5 0 0 0 0 0
```

Dummy foo.c



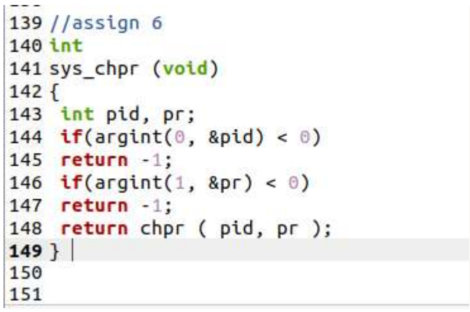
```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
int
main(int argc, char *argv[])
{
 int k, n, id;
 double x = 0, z;
 if(argc < 2 )
 n = 1; //default value
 else
 n = atoi ( argv[1] ); //from command line
 if ( n < 0 || n > 20 )
 n = 2;
 x = 0;
 id = 0;
 for ( k = 0; k < n; k++ ) {
 id = fork ();
 if ( id < 0 ) {
 printf(1, "%d failed in fork!\n", getpid() );
 } else if ( id > 0 ) { //parent 
 printf(1, "Parent %d creating child %d\n", getpid(), id );
 wait ();
 } else { // child
 printf(1, "Child %d created\n",getpid() );
 for ( z = 0; z < 8000000.0; z += 0.01 )
 x = x + 3.14 * 89.64; // useless calculations to consume CPU time
 break;
 }
 }
 exit();
}
```

Added chpr() in proc.c



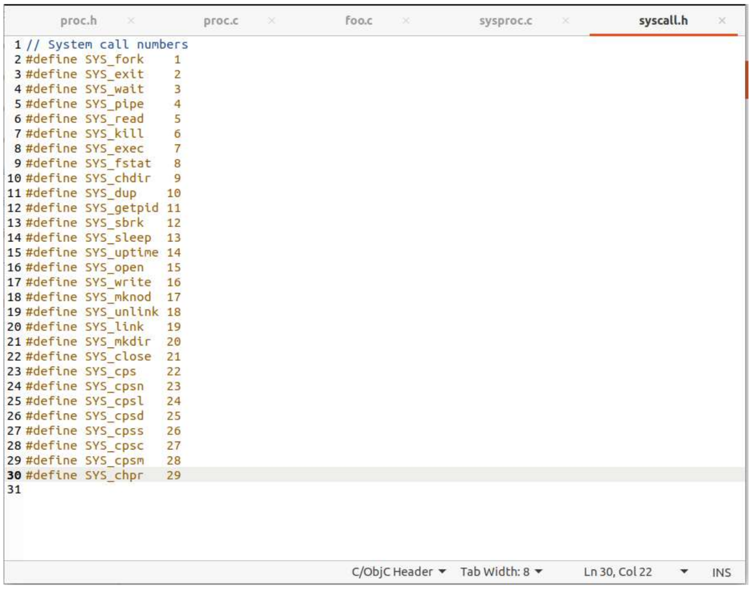
```c
//assign 6
//change priority
int
chpr( int pid, int priority )
{
 struct proc *p;

 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;
}
```

Added sys_chpr() in sysproc.c

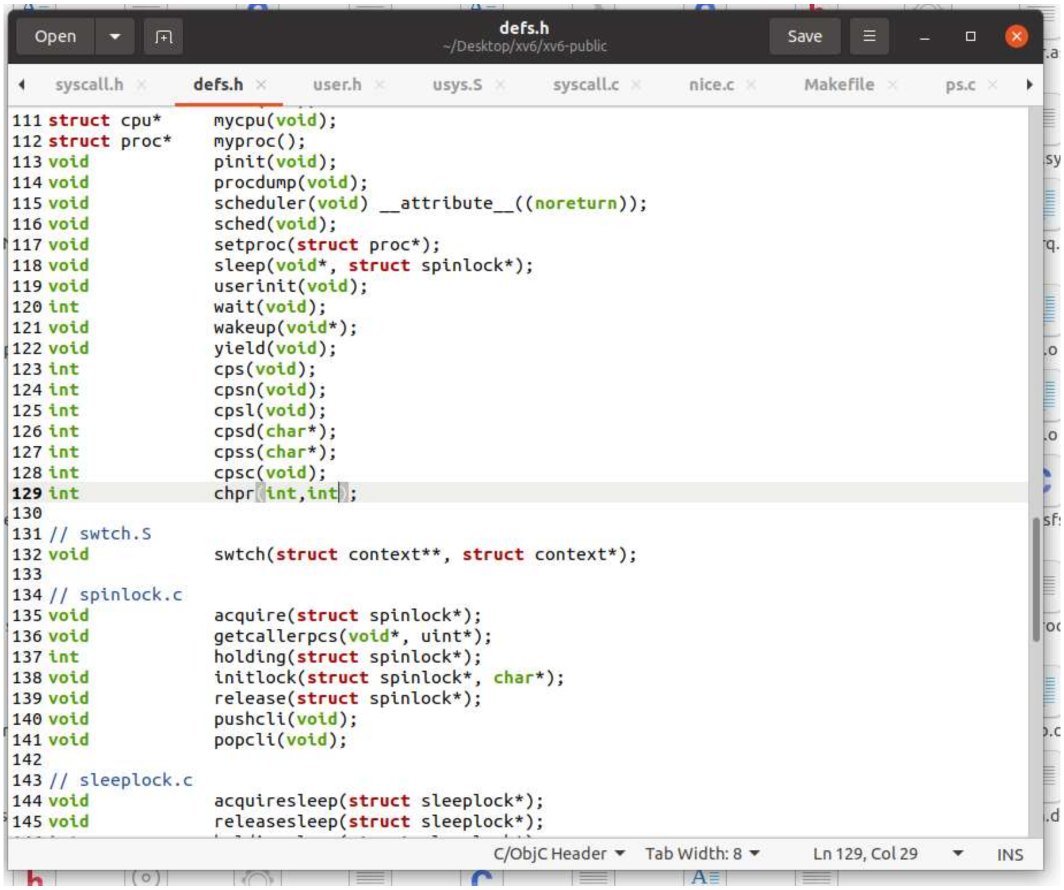


```c
//assign 6
int
sys_chpr (void)
{
 int pid, pr;
 if(argint(0, &pid) < 0)
 return -1;
 if(argint(1, &pr) < 0)
 return -1;
 return chpr ( pid, pr );
}
```

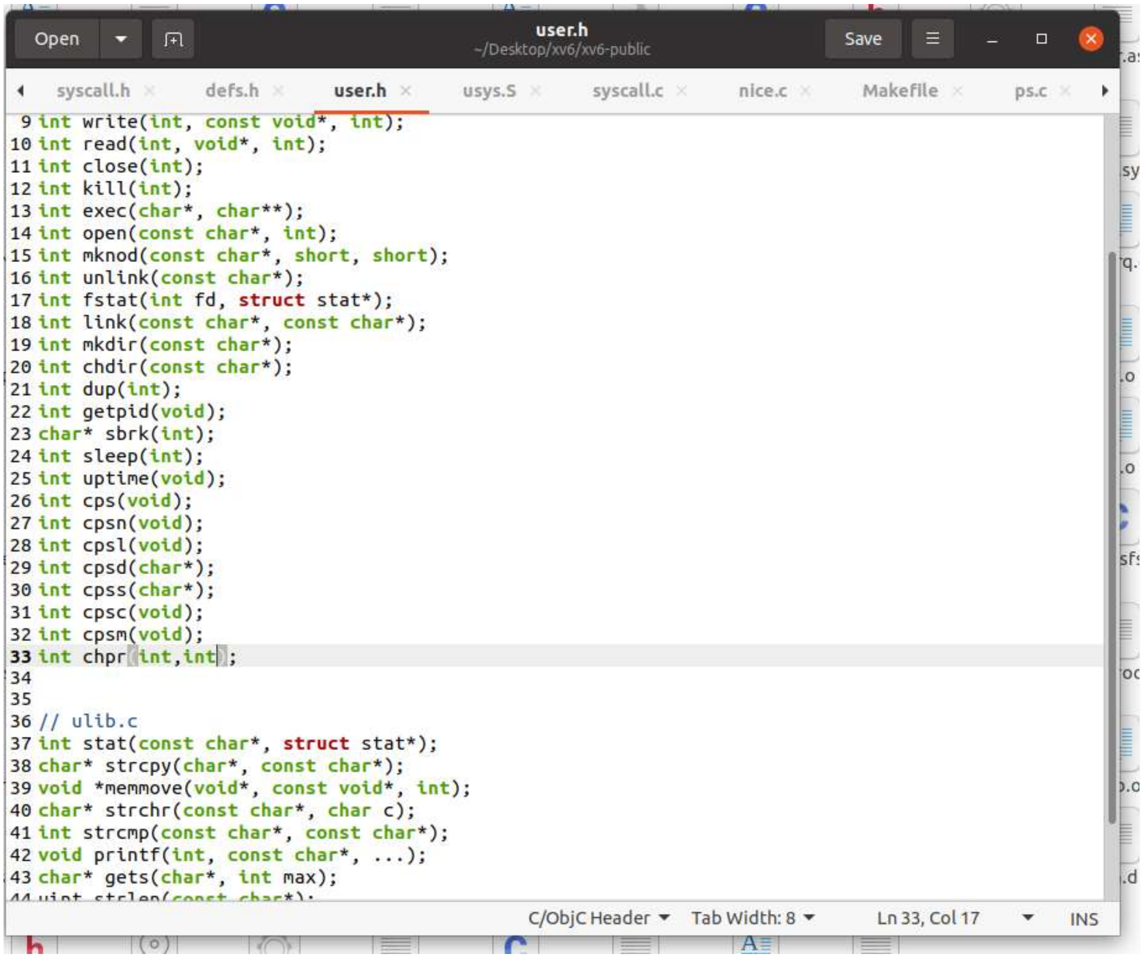
Now in the below screenshots we’re just adding chpr() as a system call just like we did for cps()



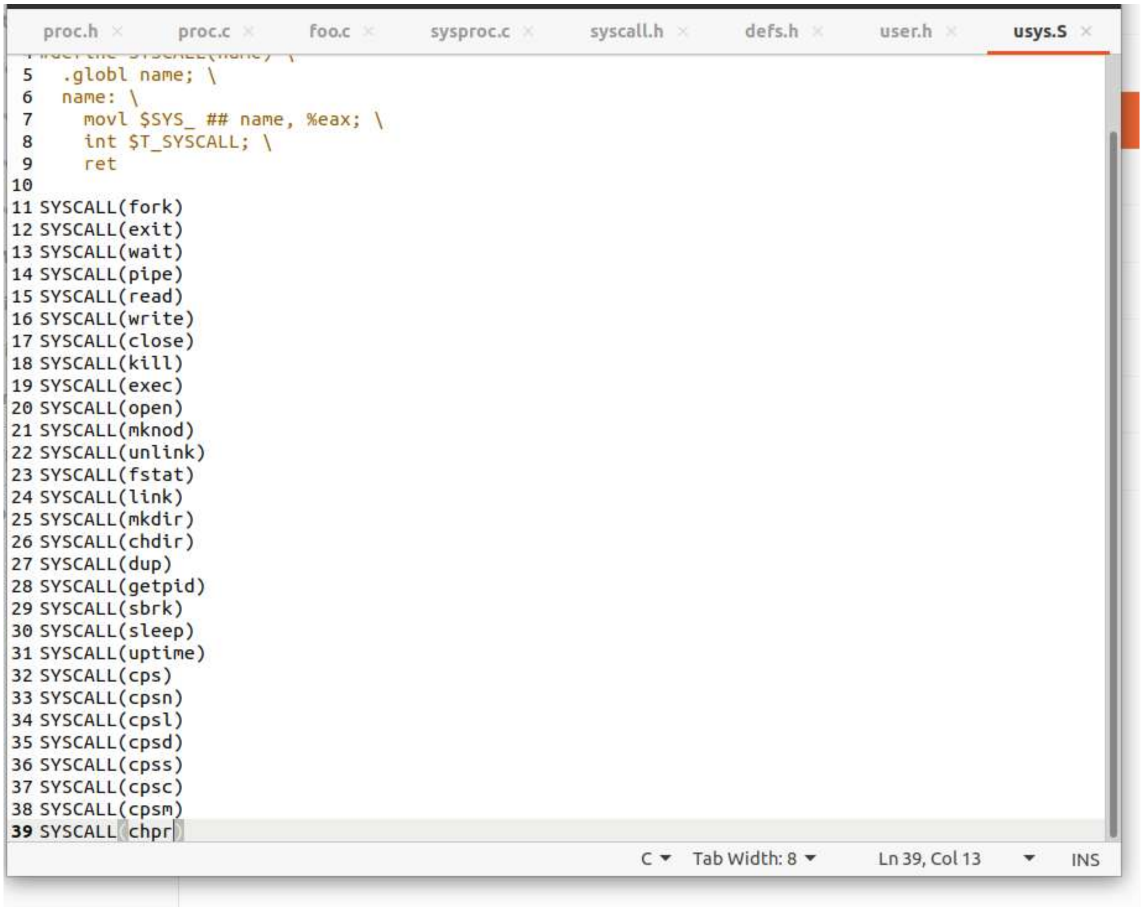
```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  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_cps    22
#define SYS_cpsn   23
#define SYS_cpsl   24
#define SYS_cpsd   25
#define SYS_cpss   26
#define SYS_cpsc   27
#define SYS_cpsm   28
#define SYS_chpr   29

```



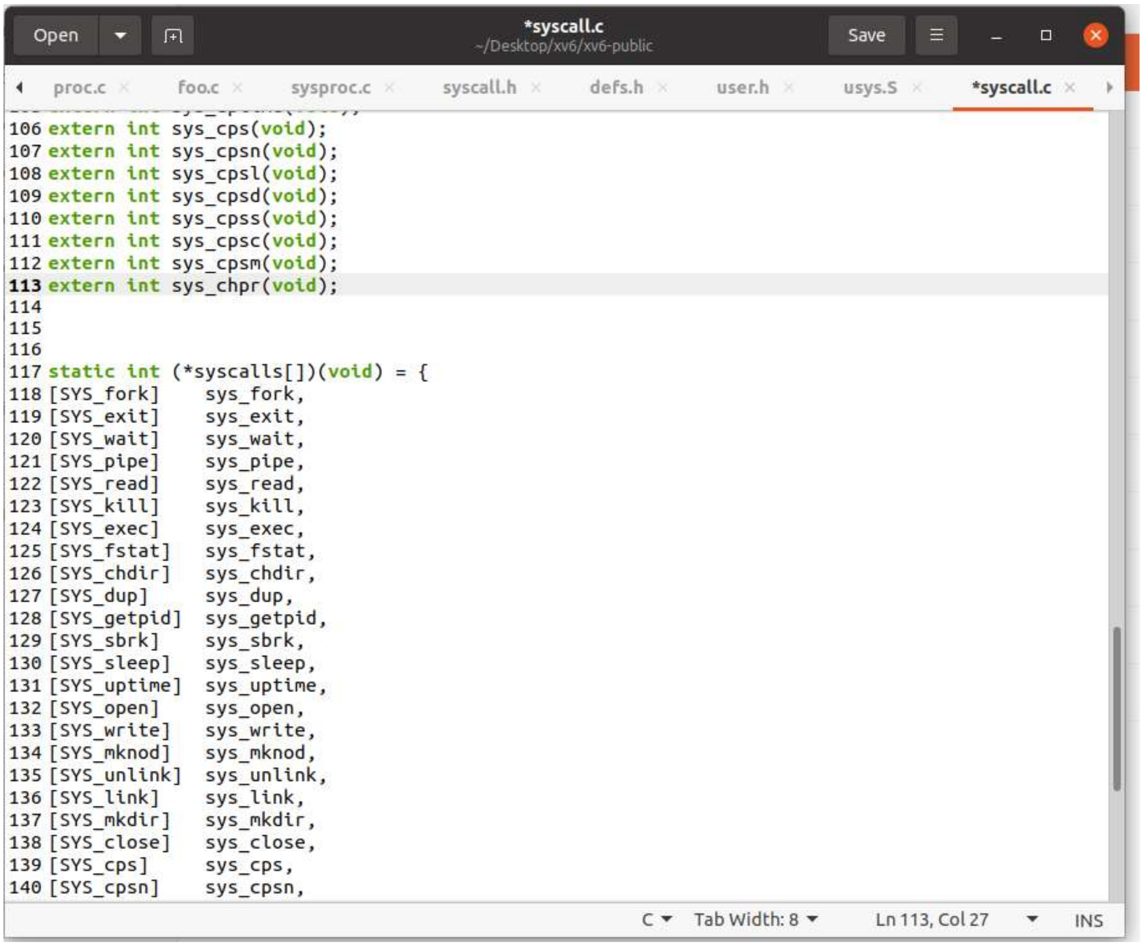
```
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            setproc(struct proc*);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);
int             cps(void);
int             cpsn(void);
int             cpsl(void);
int             cpsd(char*);
int             cpss(char*);
int             cpsc(void);
int             chpr(int,int);

// swtch.S
void            swtch(struct context**, struct context*);

// spinlock.c
void            acquire(struct spinlock*);
void            getcallerpcs(void*, uint*);
int             holding(struct spinlock*);
void            initlock(struct spinlock*, char*);
void            release(struct spinlock*);
void            pushcli(void);
void            popcli(void);

// sleeplock.c
void            acquiresleep(struct sleeplock*);
void            releasesleep(struct sleeplock*);
```



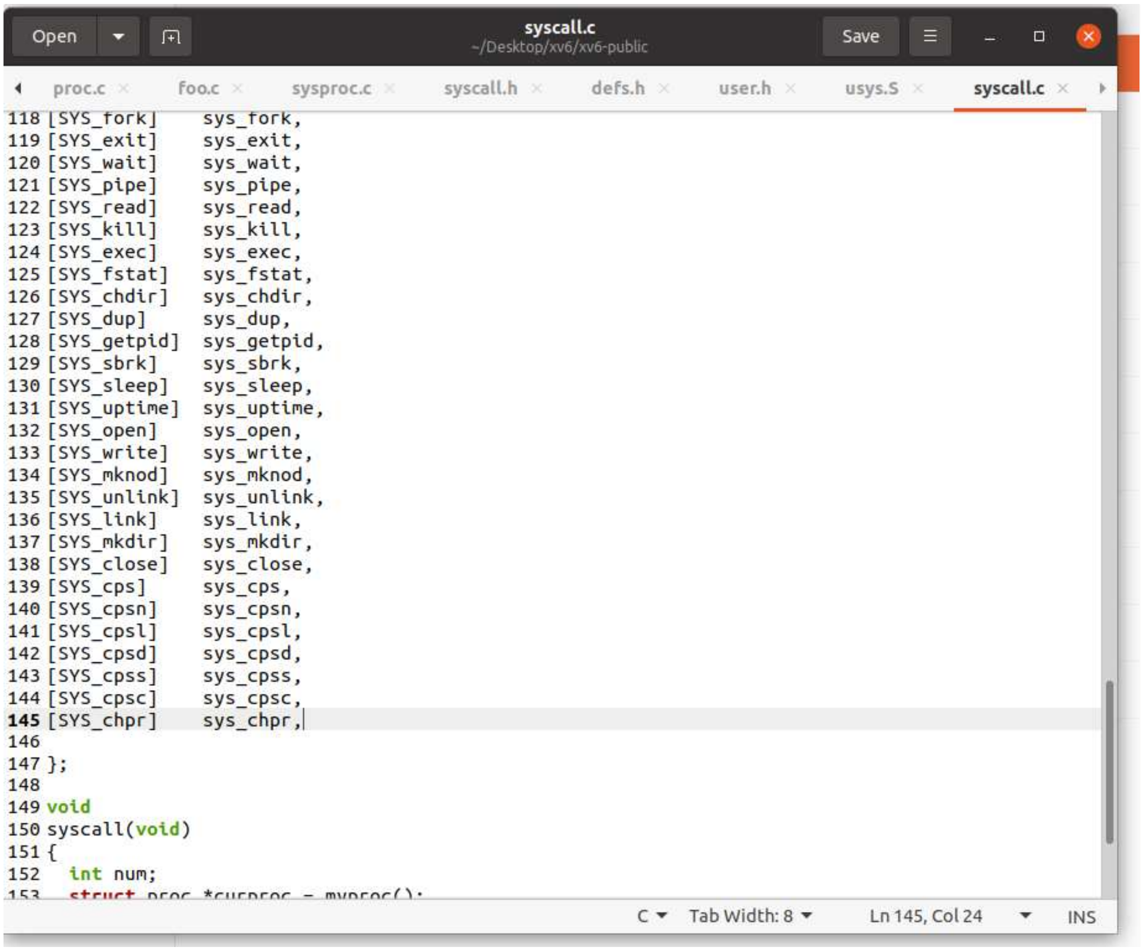
```
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int cps(void);
int cpsn(void);
int cpsl(void);
int cpsd(char*);
int cpss(char*);
int cpsc(void);
int cpsm(void);
int chpr(int,int);



// ulib.c
int stat(const char*, struct stat*);
char* strcpy(char*, const char*);
void *memmove(void*, const void*, int);
char* strchr(const char*, char c);
int strcmp(const char*, const char*);
void printf(int, const char*, ...);
char* gets(char*, int max);
uint strlen(const char*);
```



```
#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(cps)
SYSCALL(cpsn)
SYSCALL(cpsl)
SYSCALL(cpsd)
SYSCALL(cpss)
SYSCALL(cpsc)
SYSCALL(cpsm)
SYSCALL(chpr)
```

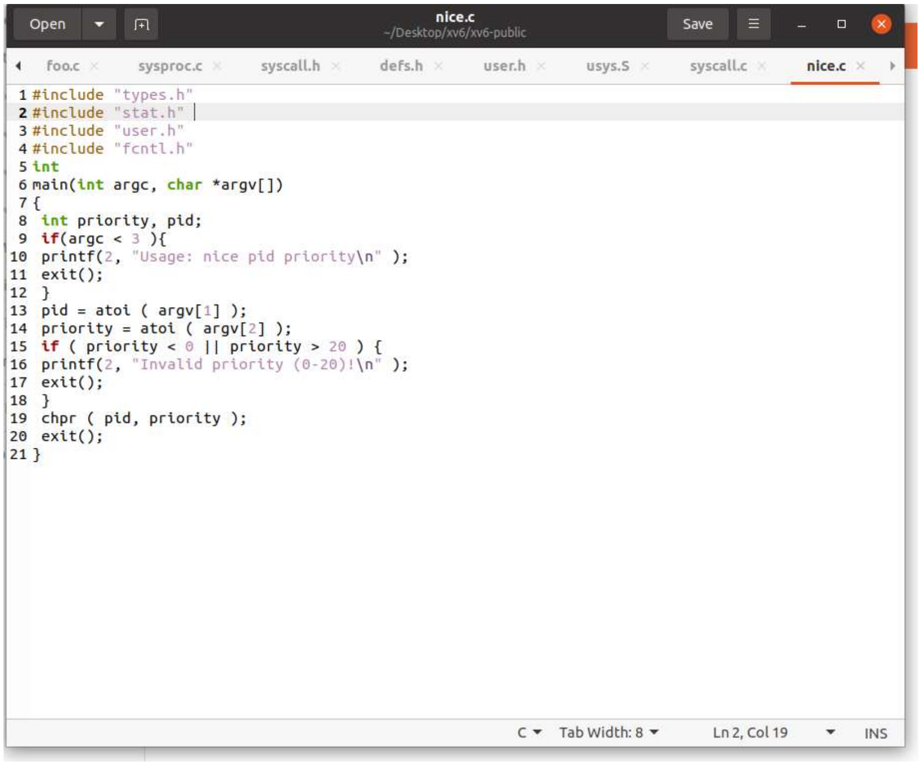


```c
extern int sys_cps(void);
extern int sys_cpsn(void);
extern int sys_cpsl(void);
extern int sys_cpsd(void);
extern int sys_cpss(void);
extern int sys_cpsc(void);
extern int sys_cpsm(void);
extern int sys_chpr(void);



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_cps]     sys_cps,
[SYS_cpsn]    sys_cpsn,
```



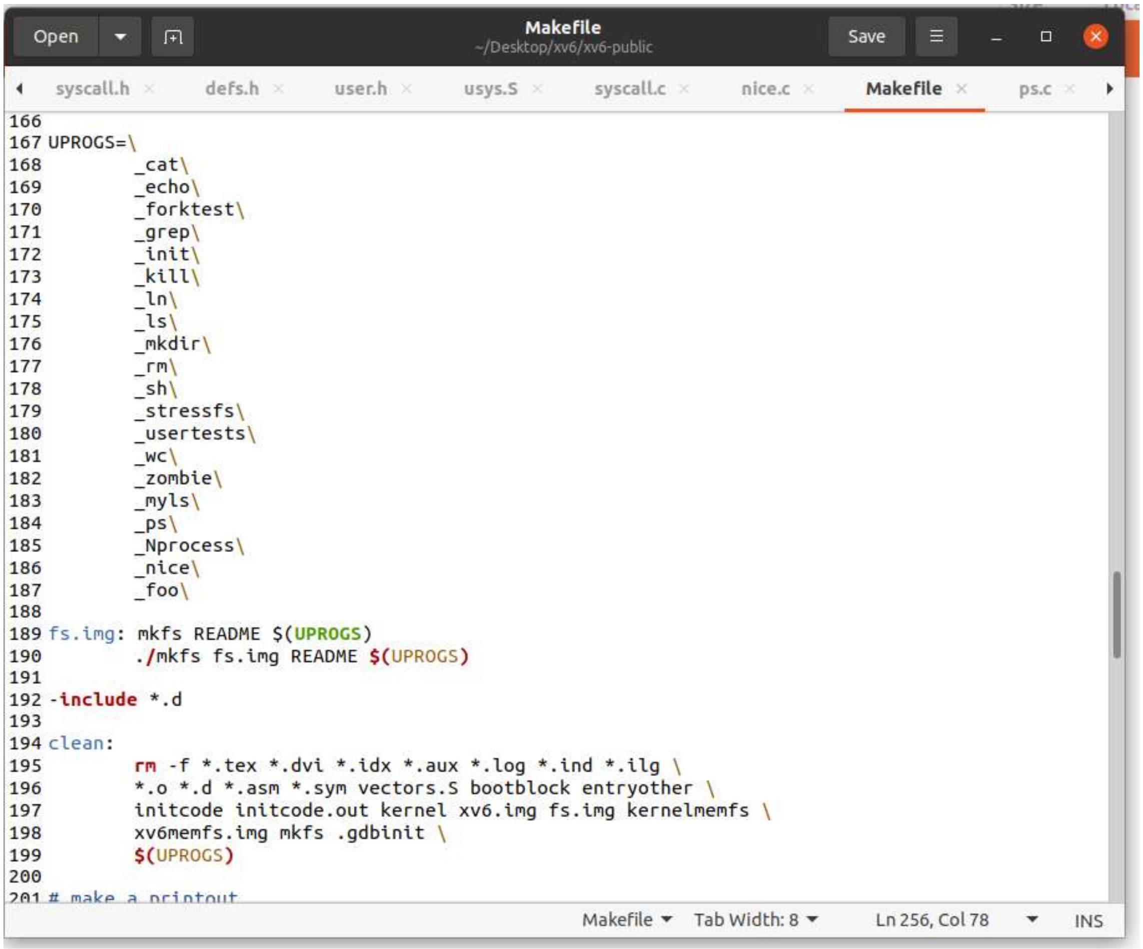
```
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   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_cps]     sys_cps,
[SYS_cpsn]    sys_cpsn,
[SYS_cpsl]    sys_cpsl,
[SYS_cpsd]    sys_cpsd,
[SYS_cpss]    sys_cpss,
[SYS_cpsc]    sys_cpsc,
[SYS_chpr]    sys_chpr,

};

void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();
```

Now creating a new user file nice.c which will call chpr



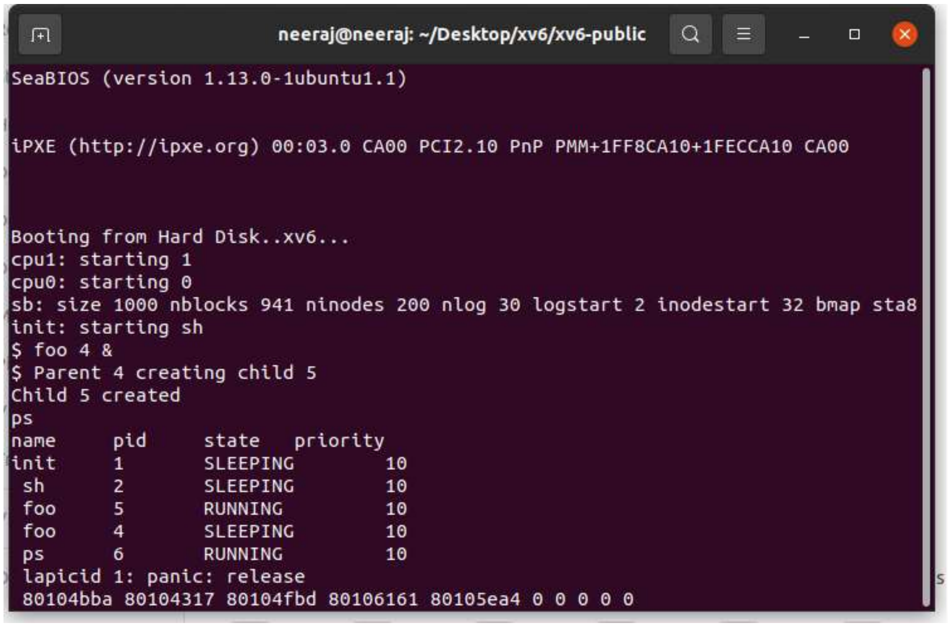
```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
int
main(int argc, char *argv[])
{
 int priority, pid;
 if(argc < 3 ){
 printf(2, "Usage: nice pid priority\n" );
 exit();
 }
 pid = atoi ( argv[1] );
 priority = atoi ( argv[2] );
 if ( priority < 0 || priority > 20 ) {
 printf(2, "Invalid priority (0-20)!\n" );
 exit();
 }
 chpr ( pid, priority );
 exit();
}
```

Modifying the makefile

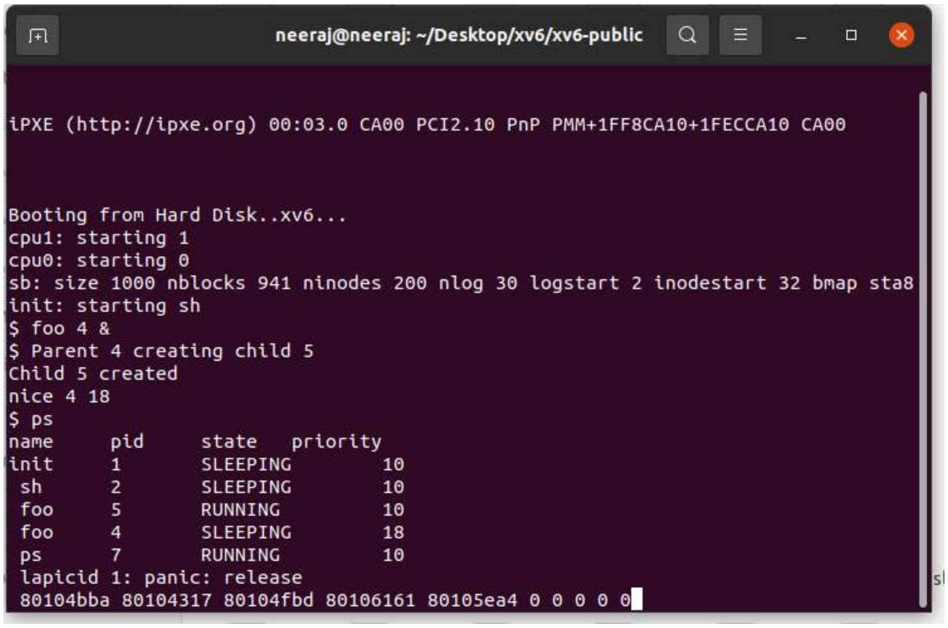


```

UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_myls\
	_ps\
	_Nprocess\
	_nice\
	_foo\

fs.img: mkfs README $(UPROGS)
	./mkfs fs.img README $(UPROGS)

-include *.d

clean:
	rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
	*.o *.d *.asm *.sym vectors.S bootblock entryother \
	initcode initcode.out kernel xv6.img fs.img kernelmemfs \
	xv6memfs.img mkfs .gdbinit \
	$(UPROGS)

# make a printout
```

Now making some dummy processes using foo command and then changing the priority of process with pid 4 to 18 using nice command



```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ foo 4 &
$ Parent 4 creating child 5
Child 5 created
ps
name     pid     state   priority
init     1       SLEEPING        10
 sh      2       SLEEPING        10
 foo     5       RUNNING         10
 foo     4       SLEEPING        10
 ps      6       RUNNING         10
 lapicid 1: panic: release
 80104bba 80104317 80104fbd 80106161 80105ea4 0 0 0 0 0
```

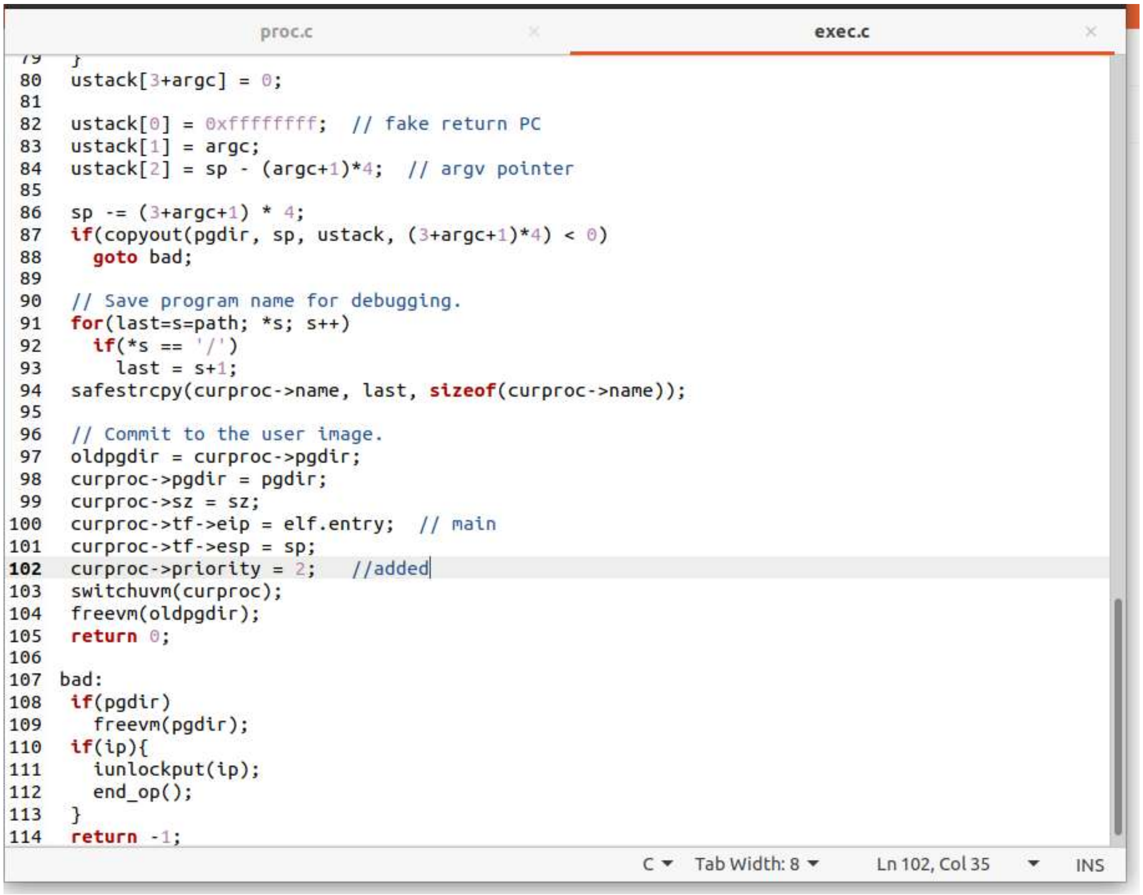


```

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ foo 4 &
$ Parent 4 creating child 5
Child 5 created
nice 4 18
$ ps
name     pid     state   priority
init     1       SLEEPING        10
 sh      2       SLEEPING        10
 foo     5       RUNNING         10
 foo     4       SLEEPING        18
 ps      7       RUNNING         10
 lapicid 1: panic: release
 80104bba 80104317 80104fbd 80106161 80105ea4 0 0 0 0 0
```

Question 2. XV6 Process Priority Scheduling: In the previous question, we have learned how to change the priority of a process. In this question, we will implement a very simple priority scheduling policy. We simply choose a runnable process with the highest priority to run. (In practice, multilevel queues are often used to put processes into groups with similar priorities.) As we have done in the previous question, we assume that a process has a value between 0 and 20, the smaller the value, the higher the priority. The default value is 10. The program nice that we implemented in the previous question is used to change the priority of a process.

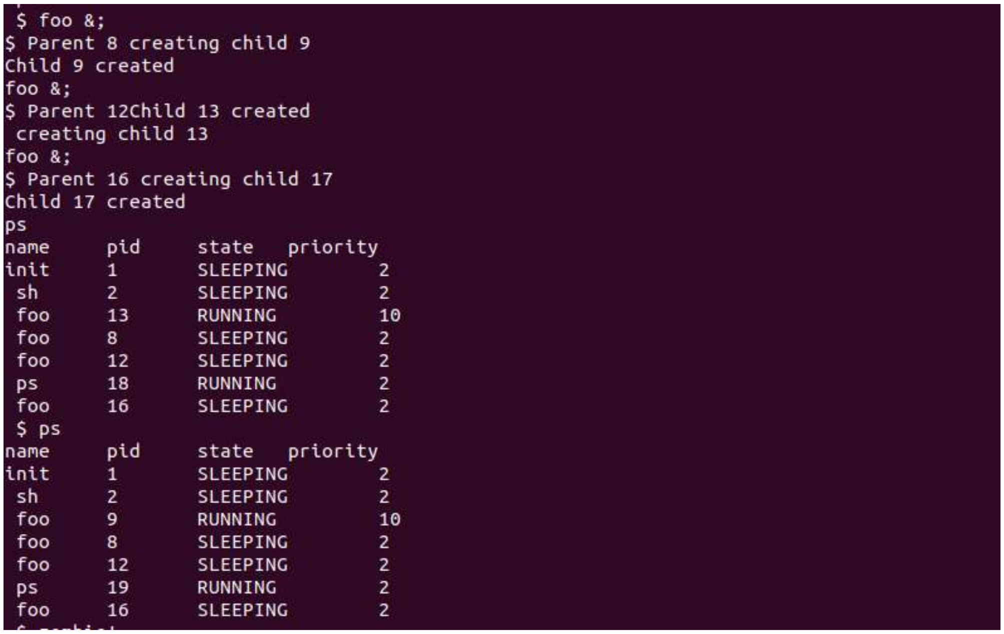
i. Give high priority to a newly loaded process by adding a priority statement in exec.c
ii. ii. Modify foo.c so that the parent waits for the children and adjust the loop for your convenience
iii. iii. Observe the default round-robin (RR) scheduling
iv. iv. Implement Priority Scheduling
v. v. Observe the priority scheduling

Adding high priority by adding a priority statement in exec.c



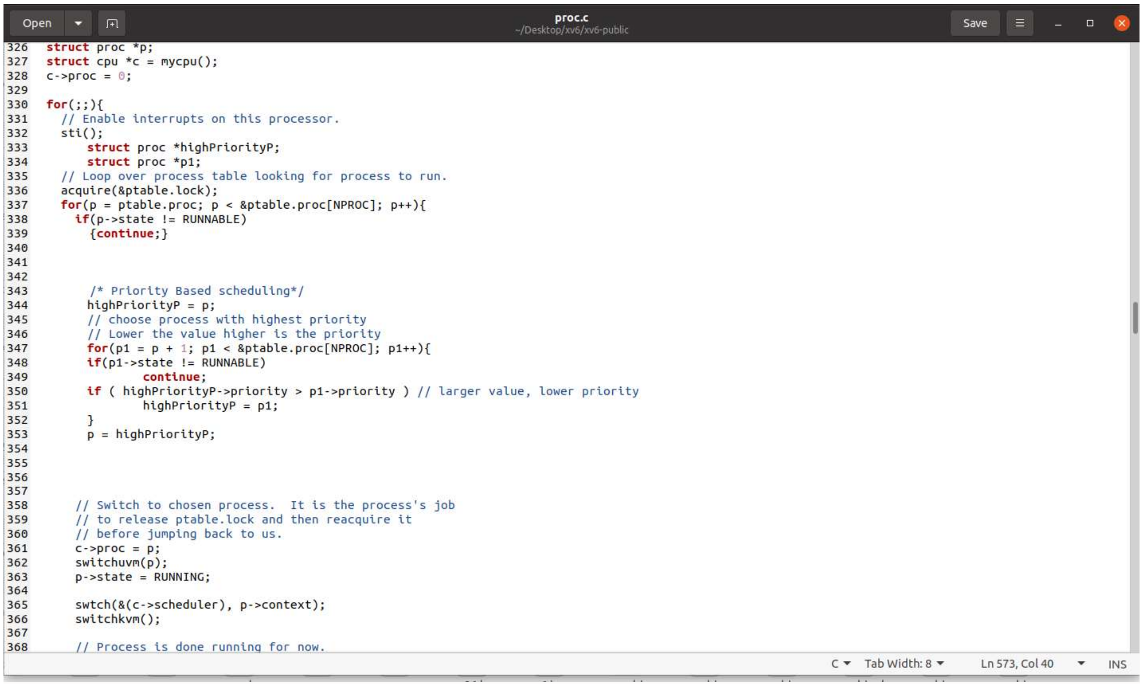
```
  }
  ustack[3+argc] = 0;

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
  ustack[2] = sp - (argc+1)*4;  // argv pointer

  sp -= (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(curproc->name, last, sizeof(curproc->name));

  // Commit to the user image.
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  curproc->priority = 2;    //added
  switchuvm(curproc);
  freevm(oldpgdir);
  return 0;

 bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
```

Observing the default round-robin scheduling



```
 $ foo &;
$ Parent 8 creating child 9
Child 9 created
foo &;
$ Parent 12Child 13 created
 creating child 13
foo &;
$ Parent 16 creating child 17
Child 17 created
ps
name     pid     state   priority
init     1       SLEEPING       2
 sh      2       SLEEPING       2
 foo     13      RUNNING        10
 foo     8       SLEEPING       2
 foo     12      SLEEPING       2
 ps      18      RUNNING        2
 foo     16      SLEEPING       2
 $ ps
name     pid     state   priority
init     1       SLEEPING       2
 sh      2       SLEEPING       2
 foo     9       RUNNING        10
 foo     8       SLEEPING       2
 foo     12      SLEEPING       2
 ps      19      RUNNING        2
 foo     16      SLEEPING       2
```

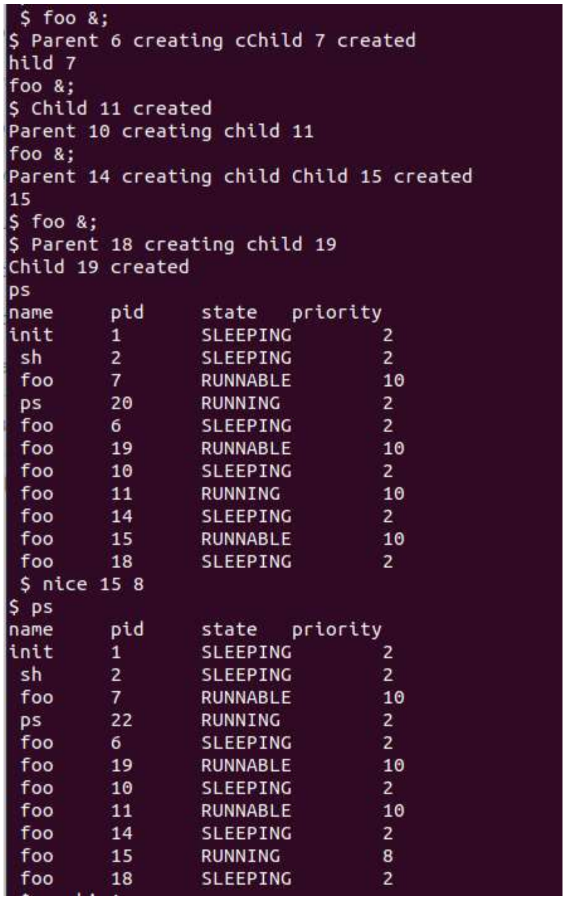
Modifying the schedule function in proc.c to select the highest priority runnable process



```c
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();
        struct proc *highPriorityP;
        struct proc *p1;
    // 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;}



        /* Priority Based scheduling*/
        highPriorityP = p;
        // choose process with highest priority
        // Lower the value higher is the priority
        for(p1 = p + 1; p1 < &ptable.proc[NPROC]; p1++){
        if(p1->state != RUNNABLE)
                continue;
        if ( highPriorityP->priority > p1->priority ) // larger value, lower priority
                highPriorityP = p1;
        }
        p = highPriorityP;




      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
```

Now first we’re creating some dummy processes using our foo command and then observe the priorities of the different processes.

Now we’re changing the priority of the runnable process with pid =15 from 10 to 8 so that it now become ready to run and then in the next ps command its status changes from runnable to running



```
 $ foo &;
$ Parent 6 creating cChild 7 created
hild 7
foo &;
$ Child 11 created
Parent 10 creating child 11
foo &;
Parent 14 creating child Child 15 created
15
$ foo &;
$ Parent 18 creating child 19
Child 19 created
ps
name     pid     state   priority
init     1       SLEEPING        2
 sh      2       SLEEPING        2
 foo     7       RUNNABLE        10
 ps      20      RUNNING         2
 foo     6       SLEEPING        2
 foo     19      RUNNABLE        10
 foo     10      SLEEPING        2
 foo     11      RUNNING         10
 foo     14      SLEEPING        2
 foo     15      RUNNABLE        10
 foo     18      SLEEPING        2
 $ nice 15 8
$ ps
name     pid     state   priority
init     1       SLEEPING        2
 sh      2       SLEEPING        2
 foo     7       RUNNABLE        10
 ps      22      RUNNING         2
 foo     6       SLEEPING        2
 foo     19      RUNNABLE        10
 foo     10      SLEEPING        2
 foo     11      RUNNABLE        10
 foo     14      SLEEPING        2
 foo     15      RUNNING         8
 foo     18      SLEEPING        2
```