OSD SKILL-7

1) A) VM.C Source Code

osd-190031187@team-osd:~/xv6-getpinfo

```
  GNU nano 2.3.1              File: vm.c

#include "param.h"
#include "types.h"
#include "defs.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "elf.h"

extern char data[];  // defined by kernel.ld
pde_t *kpgdir;  // for use in scheduler()

// Set up CPU's kernel segment descriptors.
// Run once on entry on each CPU.
void
seginit(void)
{
  struct cpu *c;

  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpuid()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
  lgdt(c->gdt, sizeof(c->gdt));
}

// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}

// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
```

```
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}

// There is one page table per process, plus one that's used when
// a CPU is not running any process (kpgdir). The kernel uses the
// current process's page table during system calls and interrupts;
// page protection bits prevent user code from using the kernel's
// mappings.
//
// setupkvm() and exec() set up every page table like this:
//
//   0..KERNBASE: user memory (text+data+stack+heap), mapped to
//                phys memory allocated by the kernel
//   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
//   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
//                for the kernel's instructions and r/o data
//   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
//                                  rw data + free physical memory
//   0xfe000000..0: mapped direct (devices such as ioapic)
//
// The kernel allocates physical memory for its heap and for user memory
// between V2P(end) and the end of physical memory (PHYSTOP)
// (directly addressable from end..P2V(PHYSTOP)).

// This table defines the kernel's mappings, which are present in
// every process's page table.
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
```

```
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};

// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}

// space for scheduler processes.
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}

// Switch h/w page table register to the kernel-only page table,
// for when no process is running.
void
switchkvm(void)
{
  lcr3(V2P(kpgdir));   // switch to the kernel page table
}

// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  if(p == 0)
    panic("switchuvm: no process");
  if(p->kstack == 0)
    panic("switchuvm: no kstack");
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");

  pushcli();
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->gdt[SEG_TSS].s = 0;
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  mycpu()->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}

// Load the initcode into address 0 of pgdir.
// sz must be less than a page.
void
```

```
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
  memmove(mem, init, sz);
}

// Load a program segment into pgdir.  addr must be page-aligned
// and the pages from addr to addr+sz must already be mapped.
int
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
  uint i, pa, n;
  pte_t *pte;

  if((uint) addr % PGSIZE != 0)
    panic("loaduvm: addr must be page aligned");
  for(i = 0; i < sz; i += PGSIZE){

  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if(sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if(readi(ip, P2V(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}

// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
```

```
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
  return newsz;
}

// Deallocate user pages to bring the process size from oldsz to
```

```
// process size.  Returns the new process size.
int
deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  pte_t *pte;
  uint a, pa;

  if(newsz >= oldsz)
    return oldsz;

  a = PGROUNDUP(newsz);
  for(; a  < oldsz; a += PGSIZE){
    pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte)
      a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
    else if((*pte & PTE_P) != 0){
      pa = PTE_ADDR(*pte);
      if(pa == 0)
        panic("kfree");
      char *v = P2V(pa);
      kfree(v);
      *pte = 0;
    }
  }
  return newsz;
}

// Free a page table and all the physical memory pages
// in the user part.
void
freevm(pde_t *pgdir)
{
  uint i;

  if(pgdir == 0)
    panic("freevm: no pgdir");
  deallocuvm(pgdir, KERNBASE, 0);
  for(i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P){
      char * v = P2V(PTE_ADDR(pgdir[i]));
      kfree(v);
    }
  }
  kfree((char*)pgdir);
```

```
// Clear PTE_U on a page. Used to create an inaccessible
// page beneath the user stack.
void
clearpteu(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, 0);
  if(pte == 0)
    panic("clearpteu");
  *pte &= ~PTE_U;
}

// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
      kfree(mem);
      goto bad;
    }
  }
  return d;

bad:
  freevm(d);
  return 0;
}

//PAGEBREAK!
// Map user virtual address to kernel address.
char*
uva2ka(pde_t *pgdir, char *uva)
{
  pte_t *pte;

  pte = walkpgdir(pgdir, uva, 0);
  if((*pte & PTE_P) == 0)
    return 0;
  if((*pte & PTE_U) == 0)
    return 0;
  return (char*)P2V(PTE_ADDR(*pte));
}

// Copy len bytes from p to user address va in page table pgdir.
// Most useful when pgdir is not the current page table.
// uva2ka ensures this only works for PTE_U pages.
int
copyout(pde_t *pgdir, uint va, void *p, uint len)
```

```
{
  char *buf, *pa0;
  uint n, va0;

  buf = (char*)p;
  while(len > 0){
    va0 = (uint)PGROUNDDOWN(va);
    pa0 = uva2ka(pgdir, (char*)va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (va - va0);
    if(n > len)
      n = len;
    memmove(pa0 + (va - va0), buf, n);
    len -= n;
    buf += n;
    va = va0 + PGSIZE;
  }
  return 0;
}

//PAGEBREAK!
// Blank page.
```

1) B) UMALLOC.C Source Code

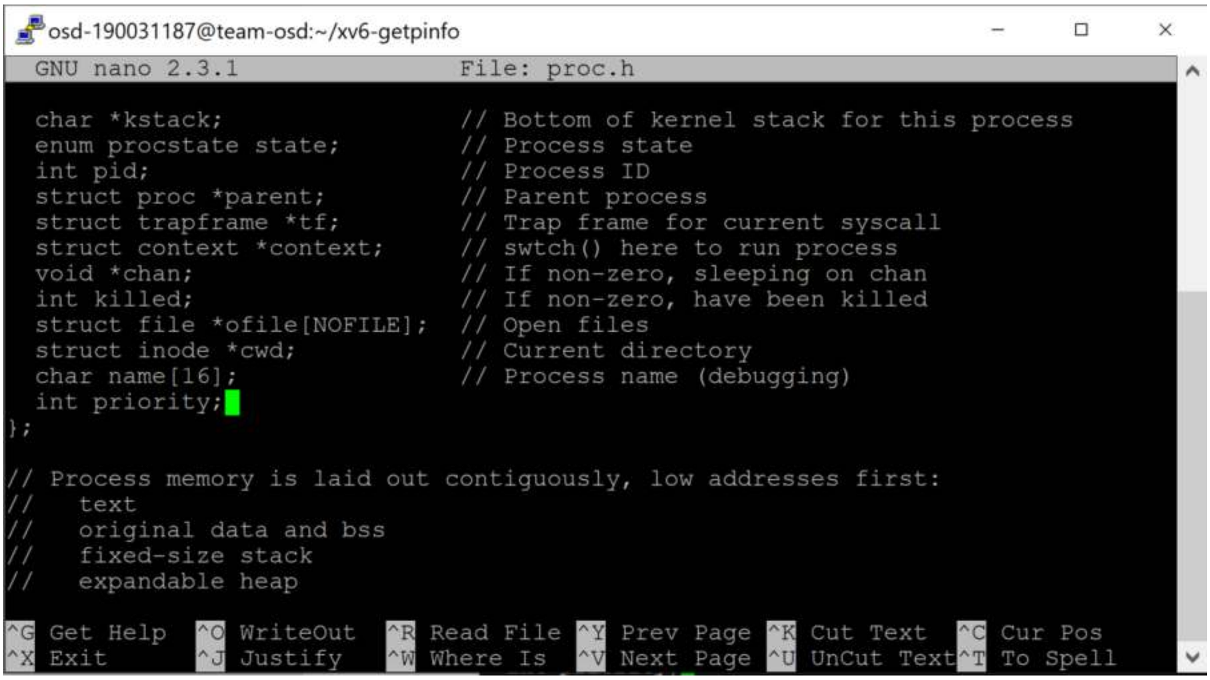
```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1              File: umalloc.c

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

// Memory allocator by Kernighan and Ritchie,
// The C programming Language, 2nd ed.  Section 8.7.

typedef long Align;

union header {
  struct {
    union header *ptr;
    uint size;
  } s;
  Align x;
};

typedef union header Header;

static Header base;
static Header *freep;

void
free(void *ap)
{
  Header *bp, *p;

  bp = (Header*)ap - 1;
  for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
    if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
      break;
  if(bp + bp->s.size == p->s.ptr){
    bp->s.size += p->s.ptr->s.size;
    bp->s.ptr = p->s.ptr->s.ptr;
  } else
    bp->s.ptr = p->s.ptr;
  if(p + p->s.size == bp){
    p->s.size += bp->s.size;
    p->s.ptr = bp->s.ptr;
  } else
    p->s.ptr = bp;
  freep = p;
}
```

```
static Header*
morecore(uint nu)
{
  char *p;
  Header *hp;

  if(nu < 4096)
    nu = 4096;
  p = sbrk(nu * sizeof(Header));
  if(p == (char*)-1)
    return 0;
  hp = (Header*)p;
  hp->s.size = nu;
  free((void*)(hp + 1));
  return freep;
}

void*
malloc(uint nbytes)
{
  Header *p, *prevp;
  uint nunits;

  nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
  if((prevp = freep) == 0){
    base.s.ptr = freep = prevp = &base;
    base.s.size = 0;
  }
  for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
    if(p->s.size >= nunits){
      if(p->s.size == nunits)
        prevp->s.ptr = p->s.ptr;
      else {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits;
      }
      freep = prevp;
      return (void*)(p + 1);
    }
    if(p == freep)
      if((p = morecore(nunits)) == 0)
        return 0;
  }
}
```

2) i) Priority scheduler in XV6 ii) chpr (XV6 Customization)

XV6 – Implementation ps, nice system calls & priority scheduling

The ps (i.e., process status) command is used to provide information about the currently running processes, including their process identification numbers (PIDs). A process, also referred to as a task, is an executing (i.e., running) instance of a program. The nice system call is used to change the priority of a given process.

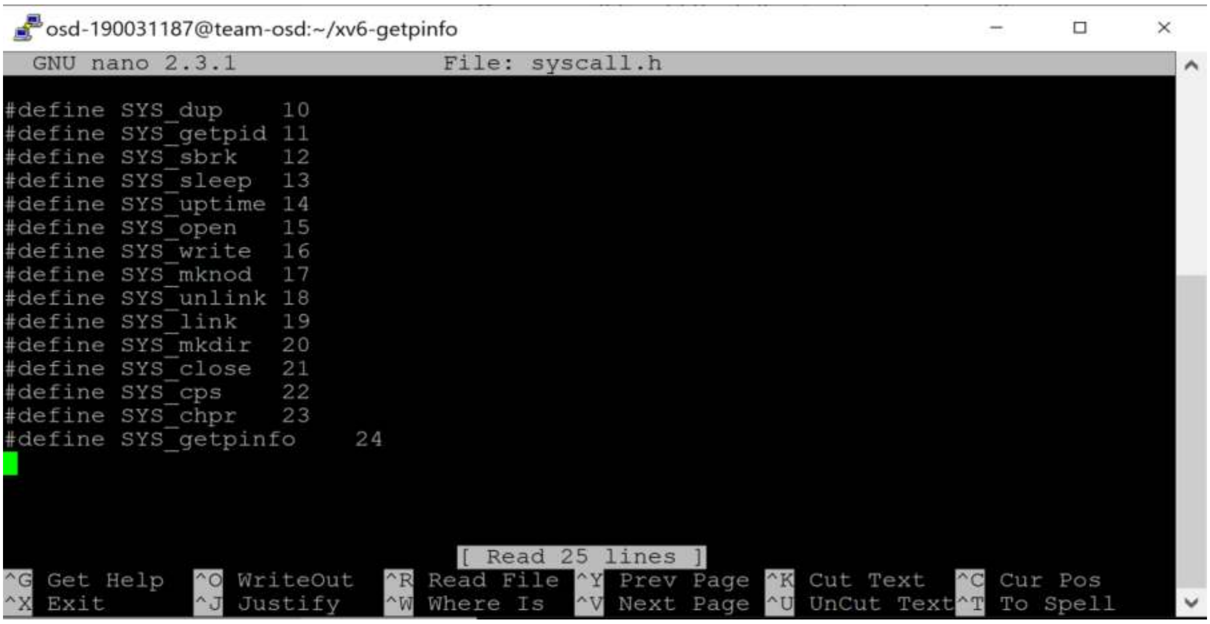


The PCB of the process is stored in proc.h file. In the struct proc in the proc.h file, add a new attribute 'priority' of int data type.

Step 1: The cps is for the ps system call and chpr (change priority) is for the nice system call. Open syscall.h, add the following two system calls:

#define SYS_cps 22

#define SYS_chpr 23



Step 2:

Next, in the PCB of the process, we have to add a new attribute ‘priority’.
The PCB of the process is stored in proc.h file.
In the struct proc in the proc.h file, add a new attribute ‘priority’ of int data type.
Next, we have to include the declaration of these functions in defs.h and user.h files

```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1                File: user.h

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 getpinfo(void);
int cps(void);
int chpr(int pid,int priority);

// 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*);

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1                File: defs.h

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             getpinfo (void);
int             cps(void);
int             chpr(int pid,int priority);

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

// spinlock.c
void            acquire(struct spinlock*);
void            getcallerpcs(void*, uint*);
int             holding(struct spinlock*);

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Step 3:

Next, we have to include the definition of the cps and chpr functions in proc.c
//Add this in the end of the proc.c file

osd-190031187@team-osd:~/xv6-getpinfo

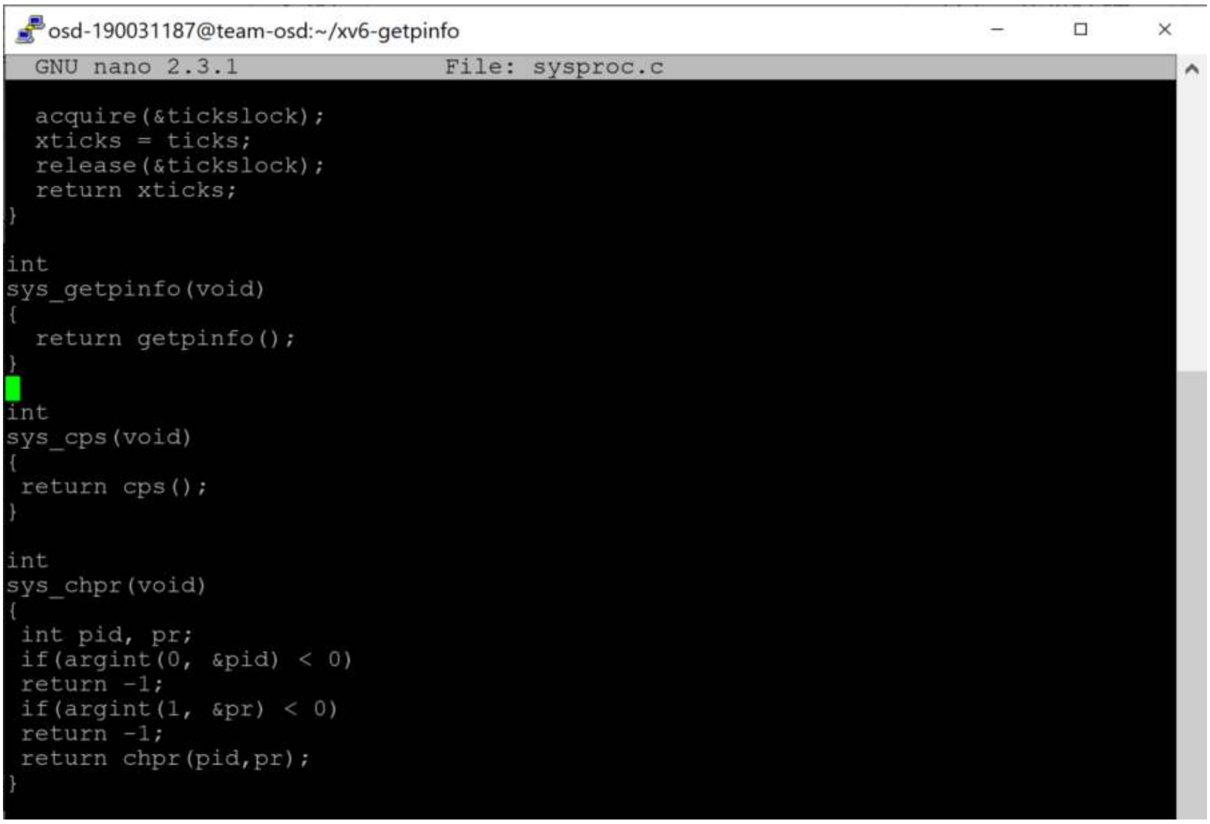
```
  GNU nano 2.3.1                 File: proc.c                              Modified

    if(p->state == SLEEPING)
        cprintf("%s \t %d \t SLEEPING \t  \n ", p->name,p->pid);
    else if(p->state == RUNNING)
        cprintf("%s \t %d \t RUNNING \t \n ", p->name,p->pid);
    else if(p->state == RUNNABLE)
        cprintf("%s \t %d \t RUNNABLE \t  \n ", p->name,p->pid);
   }
  release(&ptable.lock);
  return 22;
}

int cps()
{
struct proc *p;
//Enables 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;
}
```

osd-190031187@team-osd:~/xv6-getpinfo

```
  GNU nano 2.3.1                 File: proc.c                              Modified

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;
}

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;
}
```

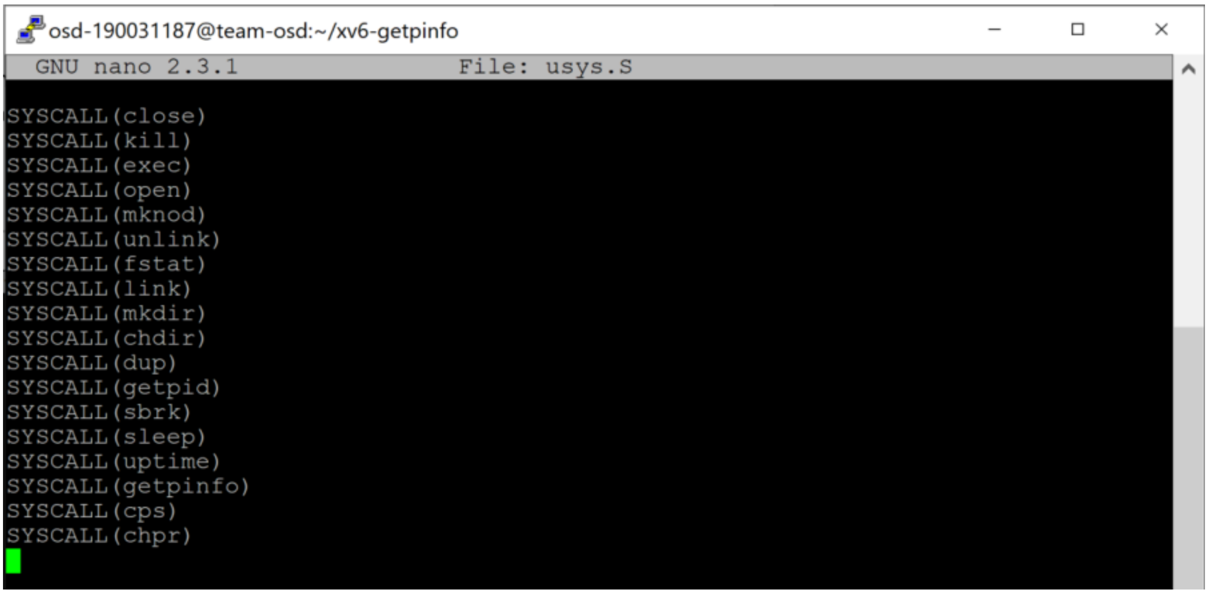
Step 4:

Next, in sysproc.c, we have to define a function in which our cps and chpr functions will be called.
//Add this in the end of the sysproc.c file



Step 5:

Next, we have to make some minor changes in the usys.S file. The '.S' extension indicates that this file has assembly level code and this file interacts with the hardware of the system.
//Add this in the end of the usys.S file
SYSCALL(cps)
SYSCALL(chpr)



Step 6:

Next, we open the syscall.c file and add the two system calls.
//Add this where the other system calls are defined in syscall.c
extern int sys_cps(void);
extern int sys_chpr(void);

```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1              File: syscall.c                        Modified

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_getpinfo(void);
extern int sys_cps(void);
extern int sys_chpr(void);
```

//Add this inside static int (*syscalls[])(void)
[SYS_cps] sys_cps,
[SYS_chpr] sys_chpr,

```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1              File: syscall.c                        Modified

[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_getpinfo]      sys_getpinfo,
[SYS_cps] sys_cps,
[SYS_chpr] sys_chpr,
};
```

Step 7:

Next, we have to create a ps.c and nice.c file in which our cps and chpr functions will be called respectively.
// ps.c

```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1              File: ps.c

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
int main(void){
 cps();
 exit();
}
```

// nice.c

```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1              File: nice.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();
}
```

Step 8:

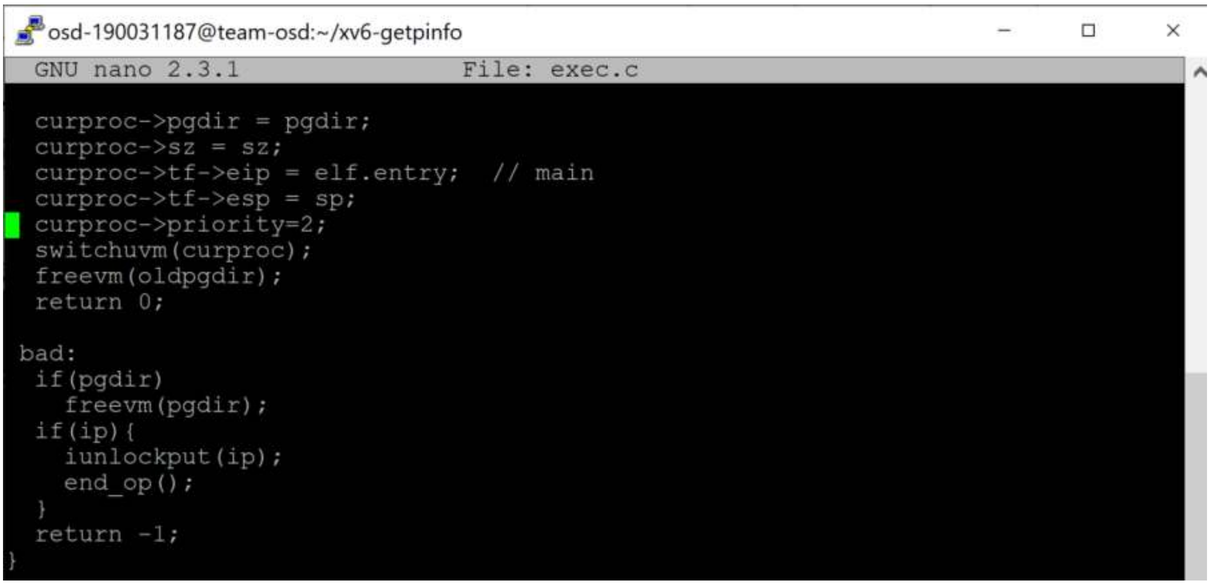
Now that we have our system calls done, we have to work on the process priority assignment. For this, firstly we define the default priority of a process in the allocproc function in the proc.c file. Here, I have assumed higher the number, lower is the priority of the process.
//Add this under the "found:" part of the allocproc function in proc.c

found:
 p->state = EMBRYO;
 p->pid = nextpid++;
 p->priority = 10; //Default Priority of a process is set to be 10

```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1              File: proc.c

  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;
  release(&ptable.lock);

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

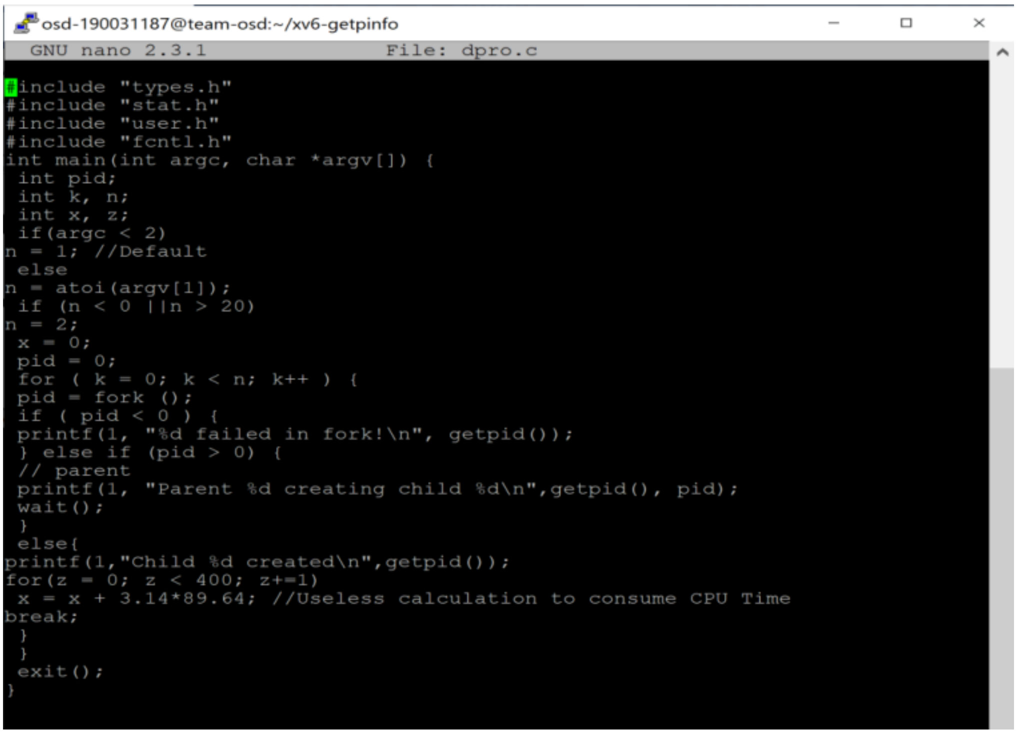
Step 9:

Now, the child process is expected to have higher priority than the parent process. So, we have to change the priority of child process when it is created. For this, we will make the changes in exec.c file.
/* Add this above the "bad:" part in the exec.c file where all other child process attributes are mentioned */
curproc->priority = 2; //Giving child process default priority of 2


```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1                File: exec.c

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

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

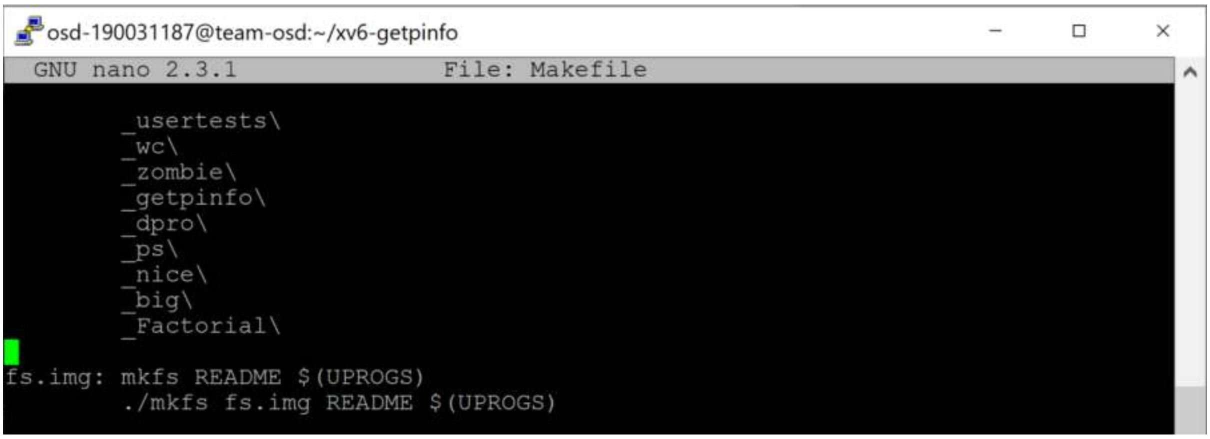
Step 10:

Now, we have to create a c program which creates a number of child process as mentioned by the user and consumes CPU time for testing our system calls and scheduling. So, we create a new file dpro.c(dummy program)and write the following code:

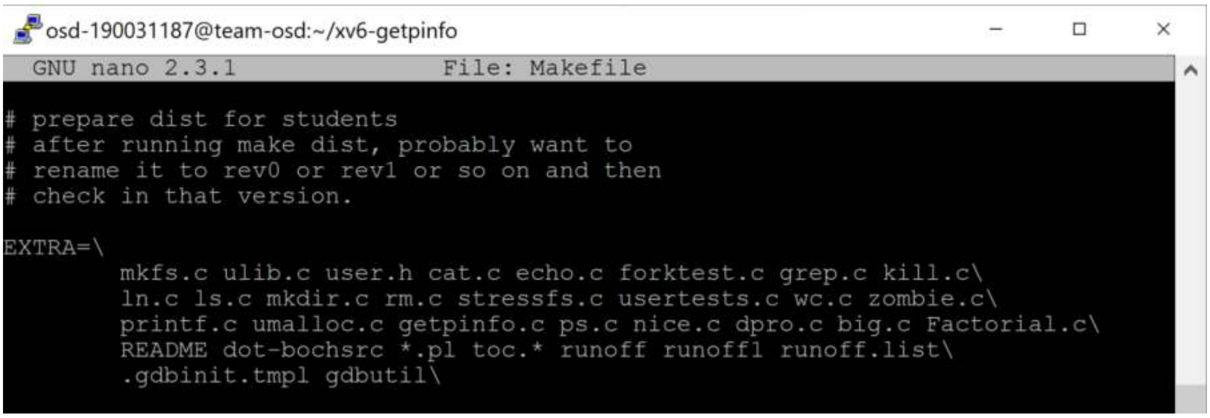

```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1                File: dpro.c

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
int main(int argc, char *argv[]) {
 int pid;
 int k, n;
 int x, z;
 if(argc < 2)
n = 1; //Default
 else
n = atoi(argv[1]);
 if (n < 0 ||n > 20)
n = 2;
 x = 0;
 pid = 0;
 for ( k = 0; k < n; k++ ) {
 pid = fork ();
 if ( pid < 0 ) {
 printf(1, "%d failed in fork!\n", getpid());
 } else if (pid > 0) {
 // parent
 printf(1, "Parent %d creating child %d\n",getpid(), pid);
 wait();
 }
 else{
printf(1,"Child %d created\n",getpid());
for(z = 0; z < 400; z+=1)
 x = x + 3.14*89.64; //Useless calculation to consume CPU Time
break;
 }
 }
 exit();
}
```

Step 11:

Now, we make the appropriate changes in the Makefile. In Makefile, under 'UPROGS', add the following: _ps\ _dpro\ _nice\ Also in the EXTRAS section of the Makefile, add nice.c, dpro.c and ps.c.



```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1              File: Makefile

        _usertests\
        _wc\
        _zombie\
        _getpinfo\
        _dpro\
        _ps\
        _nice\
        _big\
        _Factorial\

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



```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1              File: Makefile

# prepare dist for students
# after running make dist, probably want to
# rename it to rev0 or rev1 or so on and then
# check in that version.

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\
        printf.c umalloc.c getpinfo.c ps.c nice.c dpro.c big.c Factorial.c\
        README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
        .gdbinit.tmpl gdbutil\
```

Priority based round robin scheduling walkthrough

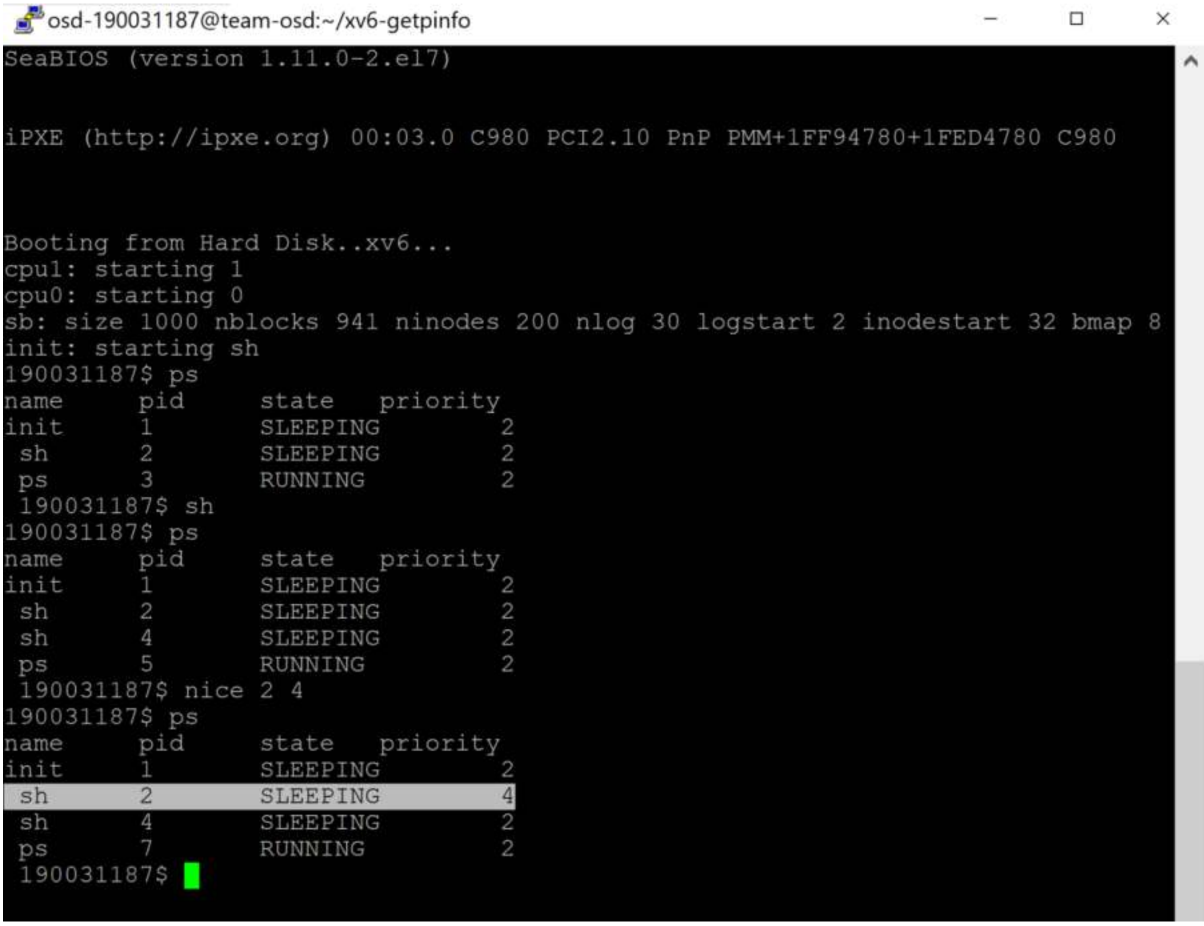
Priority based Round-Robin CPU Scheduling algorithm is based on the integration of round-robin and priority scheduling algorithm. It retains the advantage of round robin in reducing starvation and also integrates the advantage of priority scheduling. Existing round robin CPU scheduling algorithm cannot be implemented in real time operating system due to their high context switch rates, large waiting time, large response time, and large turnaround time and less throughput. The proposed algorithm improves all the drawbacks of round robin scheduling algorithm.

Step 1:

For implementing this, we make the required changes in scheduler function in proc.c file.
//Replace the scheduler function with the one below for priority round robin scheduling

```
osd-190031187@team-osd:~/xv6-getpinfo
  GNU nano 2.3.1              File: proc.c

//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
 struct proc *p, *p1;
 struct cpu *c = mycpu();
 c->proc = 0;

 for(;;){
 // Enable interrupts on this processor.
 sti();
 struct proc *highP;
 // 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;
 // Switch to chosen process. It is the process's job
 // to release ptable.lock and then reacquire it
 // before jumping back to us.
 highP = p;
 //choose one with highest priority
 for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
if(p1->state != RUNNABLE)
 continue;
if(highP->priority > p1->priority) //larger value, lower priority
 highP = p1;
 }
 p = highP;
 c->proc = p;
 switchuvm(p);
 p->state = 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);
 }

}
```

Output:



```
osd-190031187@team-osd:~/xv6-getpinfo
SeaBIOS (version 1.11.0-2.el7)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF94780+1FED4780 C980



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 8
init: starting sh
190031187$ ps
name     pid     state   priority
init     1       SLEEPING        2
 sh      2       SLEEPING        2
 ps      3       RUNNING         2
 190031187$ sh
190031187$ ps
name     pid     state   priority
init     1       SLEEPING        2
 sh      2       SLEEPING        2
 sh      4       SLEEPING        2
 ps      5       RUNNING         2
 190031187$ nice 2 4
190031187$ ps
name     pid     state   priority
init     1       SLEEPING        2
 sh      2       SLEEPING        4
 sh      4       SLEEPING        2
 ps      7       RUNNING         2
 190031187$
```