ADDING SYSTEM CALL, IMPLEMENTATION OF PRIORITY SCHEDULER AND COPY ON WRITE FORK IN XV6

REVIEW REPORT

Submitted by

Ankith Lagupudi (19BCE0478)

Simbothula Varun Kumar (19BCI0050)

Sighakolli Susmitha (19BCB0056)

Prepared For

OPERATING SYSTEMS (CSE2005)

PROJECT COMPONENT

School of Computer Science and Engineering

AIM:

The aim of the project is to add a system calls, to implement priority scheduler, copy on write fork on Operating system. Xv6 operating system, obtained from the github library, would be used, and we aim to add simple system calls into the existing code using an emulator (QEMU).

Abstract:

System calls are interfaces between the user and the services of the Operating system. They are required to execute processes that require memory spaces/ file accessing/ device connection etc. However, developers themselves don’t have direct access to system calls. System calls are used for process control, file manipulation, device management, communication and information management. In this project, we use a type 2 hypervisor (QEMU) to run a pre-existing Operating system Xv6 . We then add the code we develop for executing system calls to the operating system.

OPERATING SYSTEM USED:

Xv6 is a simple Operating system developed by MIT for its own Operating Systems course. It is developed from the sixth edition of Unix and is coded in the C language. It is an Open Source Software and is freely available for us to develop upon.

EMULATOR USED: QEMU

QEMU:

QEMU stands for quick emulator. QEMU is a generic open source machine emulator and virtualizer. It is able to emulate other operating systems on another operating systems. The performance of QEMU is far better than virtual box.It emulates the machine's processor through dynamic binary translation(where sequences of instructions are translated from a source instruction to the target instruction set).

Literature Review

1.PB Hansen - 1973 - dl.acm.org Operating system Principles
This introductory book was really helpful as it helped us get a grasp of the operating systems concept and explained the function of operating system. It elaborates on the principles it is built on, the main one being enforcing behavioural rules on users to enable sharing of computer systems. Safe methods that help in making large programs super efficient are explored and help us get a grasp on how different operatings systems operate on the basis of the same fundamental principles. It explains sequential and concurrent processes and also helps us get a solid foundation on the various theory concepts that we learnt - resource sharing and management,scheduling algorithms etc.

2. M Barabanov - 1997 - yodaiken.com- A linux-based real-time operating system
This book explores the idea of a real time linux. With respect to our project it helped us understand interrupt controllers and emulation as the OS functions on interrupt control emulation and user defined schedulers. It helped us understand better the concepts of Scheduling and interprocess communication that are critical for the functioning of any OS.

3. R Cox, MF Kaashoek, R Morris- xv6: a simple, Unix-like teaching operating system
This was one of the most important and useful resource for our project. It explains the working of xv6 and concepts of operating system and how they are employed in xv6.Interfaces, organization, page tables,traps and device drivers, locking , file systems all concepts are completely elaborated and code snippets from xv6 are used to explain them.This helped us in inserting and executing the system call concept.

4.M. Nakajima and S. Oikawa, "Effective I/O Processing with Exception-Less System Calls for Low-Latency Devices," 2015 Third International Symposium on Computing and Networking (CANDAR), Sapporo, 2015, pp. 604-606, doi: 10.1109/CANDAR.2015.91.
Latency is the turnaround time of execution of a request. To make our system calls more

effective, we referred to this document. However, while it was very insightful and helped us to learn about the future of input output systems, the execution process remained advanced for our execution.

5. S. Oikawa, "Delegating the kernel functions to an application program in UV6," 2012 IEEE International Conference on Signal Processing, Communication and Computing (ICSPCC 2012), Hong Kong, 2012, pp. 406-409, doi: 10.1109/ICSPCC.2012.6335626. UV6 improves upon xv6 and this paper clearly shows how assigning kernel functions to application programs helps in significantly increasing processor utilization and efficiency of the operating system increases multifold. It helped us to get new ideas to implement interrupts and system calls. The ideas also allowed us to think further about the future prospects of proxy kernels.

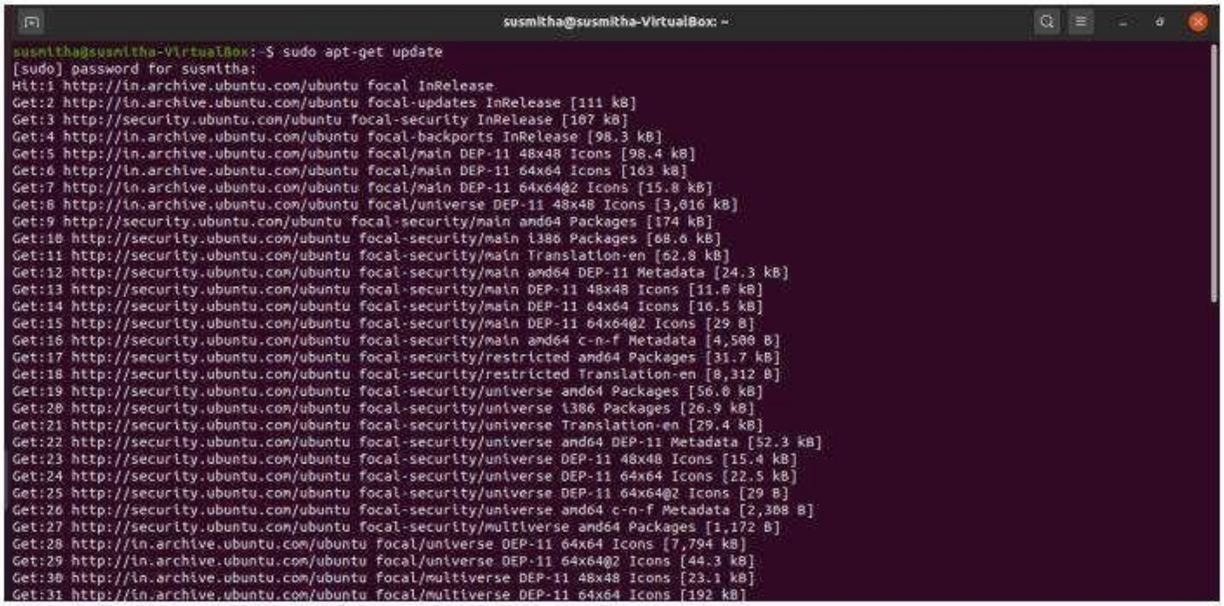
6. Copy on write file system consistency and block usage-David HitzMichael MalcolmJames LauByron Rakitzis
This paper significantly helped us realize the successful implementation of copy on write . It explains the importance of file system snapshots that help in preserving disk space while giving perfect information about the inode file without cloning it.

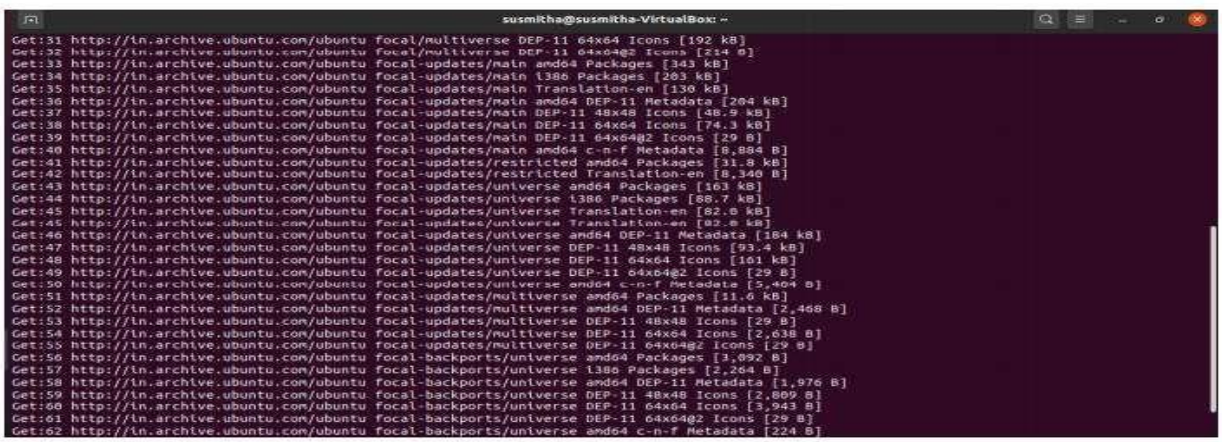
7. Systems and methods for adaptive copy on write- Darren P. SchackEric M. LemarNeal T. Fachan
This paper helped us dive deep into the concept of copy on write. It introduced to us the idea of poin-in-time-copy thats another way of recording file system state and taking the snapshot. It shows how various physically distributed systems can efficiently use copy on write function.

INSTALLATION PROCEDURE:

UPDATING: Before installing anything, we have to make sure that the ubuntu operating system is up to date. The Updated operating system makes our work easier and keeps our PC secured.**COMMAND**: sudo apt-get update. This command is used to update ubuntu operating system

SCREENSHOT:



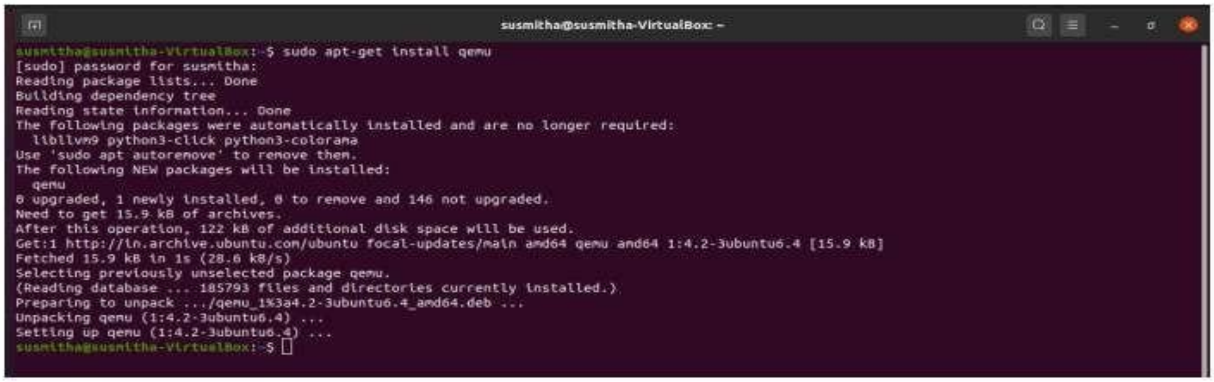


QEMU INSTALLATION:

COMMAND:

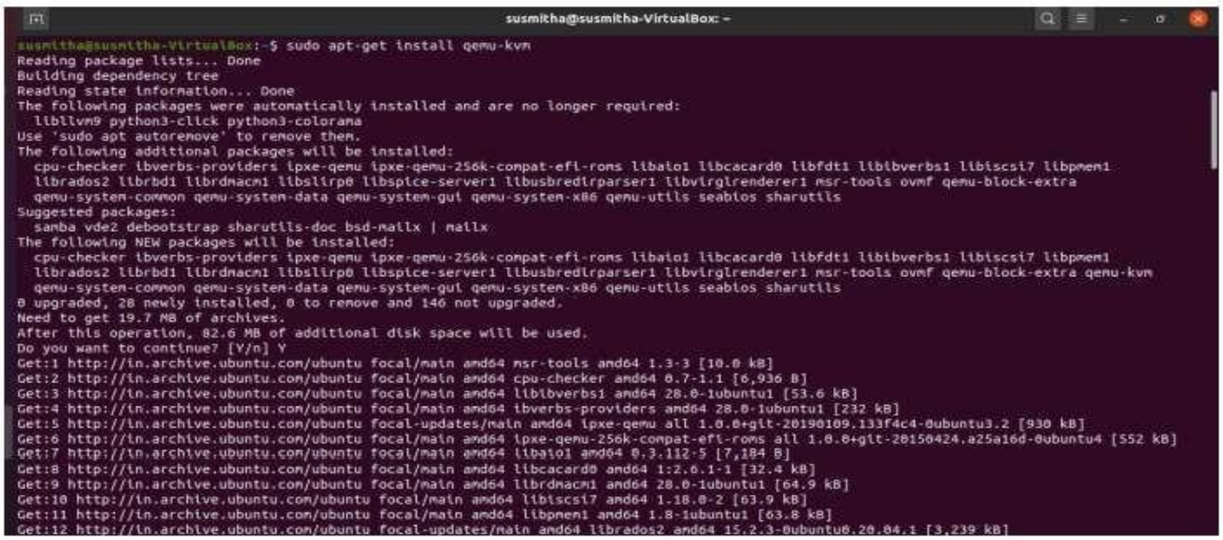
Sudo apt-get install qemu

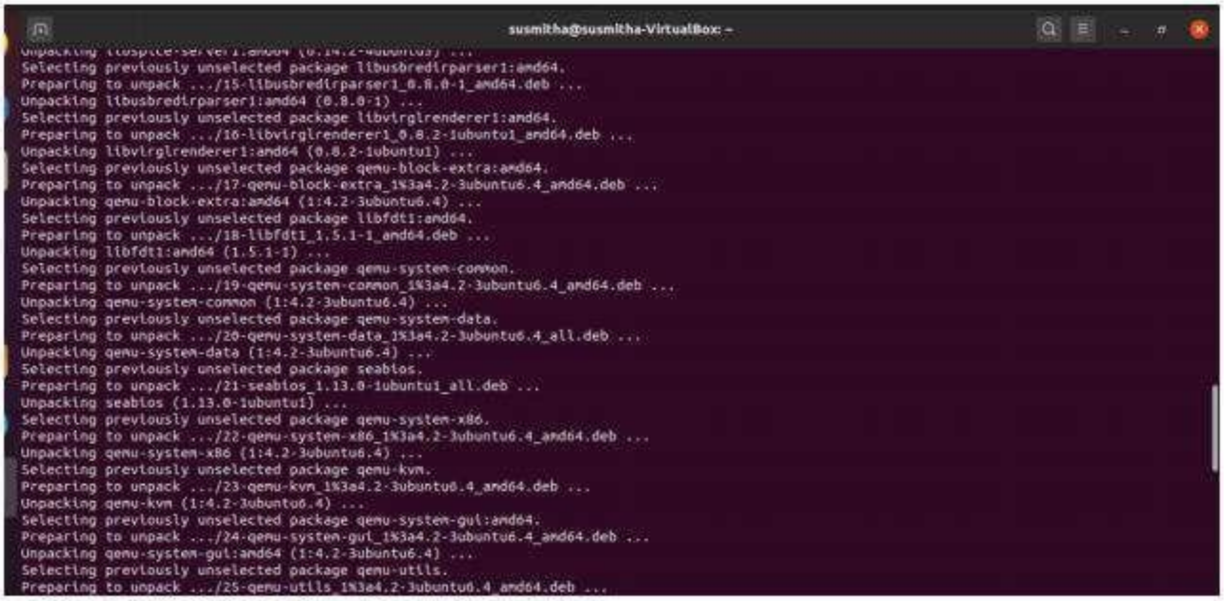
SCREENSHOT:

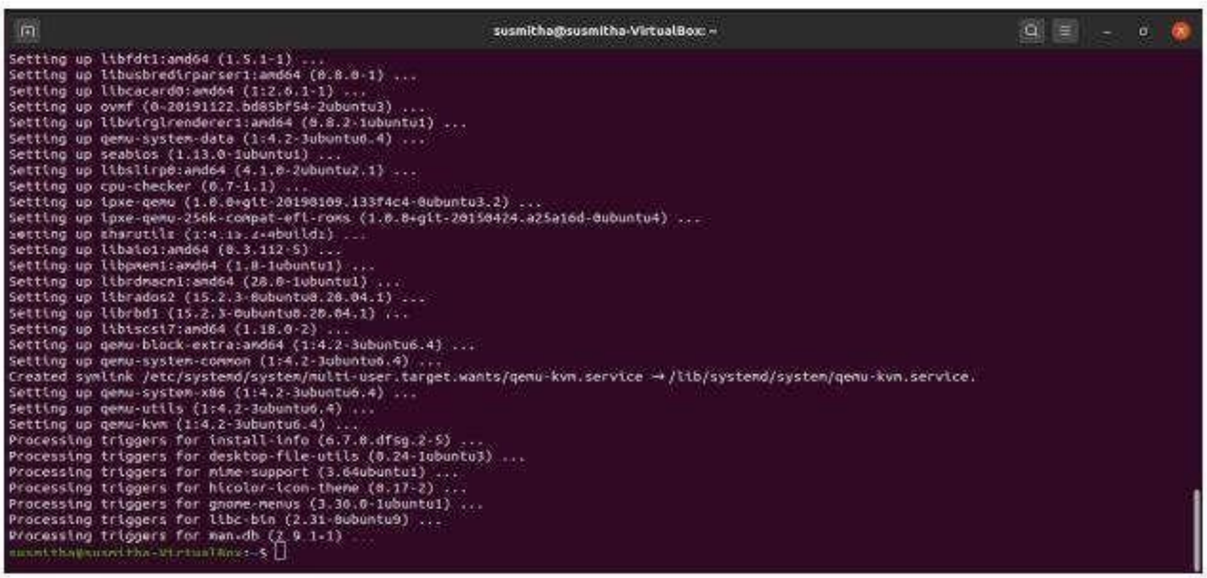


COMMAND:

sudo apt get-install qemu-kvm



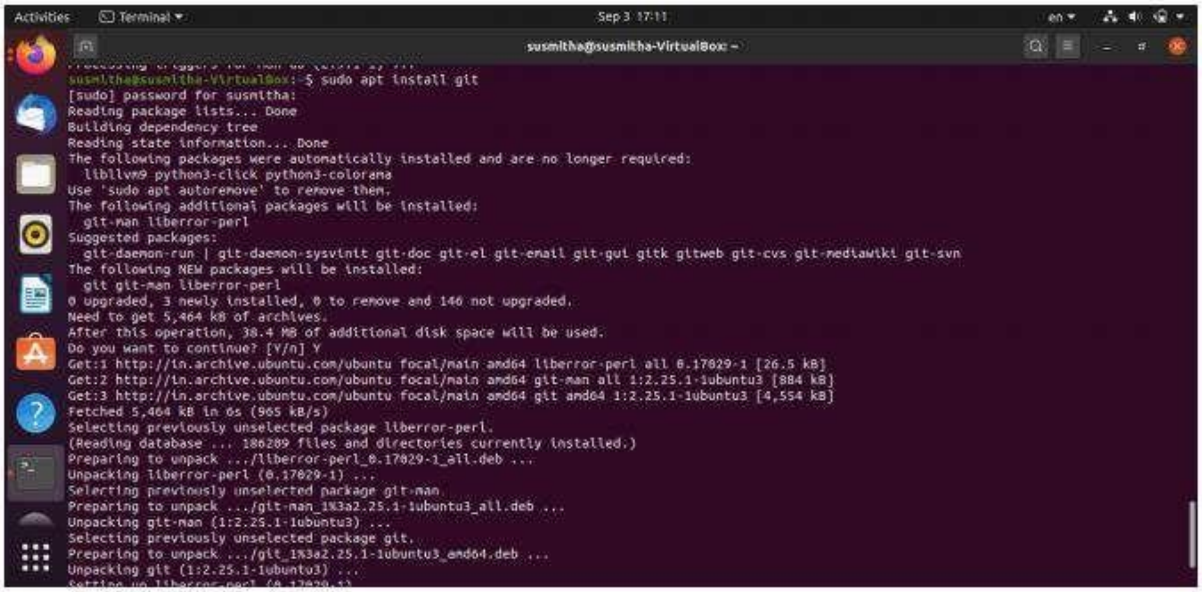

susmitha@susmitha-VirtualBox: ~
Selecting previously unselected package libusbredirparser1:amd64.
Preparing to unpack .../15-libusbredirparser1_0.8.0-1_amd64.deb ...
Unpacking libusbredirparser1:amd64 (0.8.0-1) ...
Selecting previously unselected package libvirglrenderer1:amd64.
Preparing to unpack .../16-libvirglrenderer1_0.8.2-1ubuntu1_amd64.deb ...
Unpacking libvirglrenderer1:amd64 (0.8.2-1ubuntu1) ...
Selecting previously unselected package qemu-block-extra:amd64.
Preparing to unpack .../17-qemu-block-extra_1%3a4.2-3ubuntu6.4_amd64.deb ...
Unpacking qemu-block-extra:amd64 (1:4.2-3ubuntu6.4) ...
Selecting previously unselected package libfdt1:amd64.
Preparing to unpack .../18-libfdt1_1.5.1-1_amd64.deb ...
Unpacking libfdt1:amd64 (1.5.1-1) ...
Selecting previously unselected package qemu-system-common.
Preparing to unpack .../19-qemu-system-common_1%3a4.2-3ubuntu6.4_amd64.deb ...
Unpacking qemu-system-common (1:4.2-3ubuntu6.4) ...
Selecting previously unselected package qemu-system-data.
Preparing to unpack .../20-qemu-system-data_1%3a4.2-3ubuntu6.4_all.deb ...
Unpacking qemu-system-data (1:4.2-3ubuntu6.4) ...
Selecting previously unselected package seabios.
Preparing to unpack .../21-seabios_1.13.0-1ubuntu1_all.deb ...
Unpacking seabios (1.13.0-1ubuntu1) ...
Selecting previously unselected package qemu-system-x86.
Preparing to unpack .../22-qemu-system-x86_1%3a4.2-3ubuntu6.4_amd64.deb ...
Unpacking qemu-system-x86 (1:4.2-3ubuntu6.4) ...
Selecting previously unselected package qemu-kvm.
Preparing to unpack .../23-qemu-kvm_1%3a4.2-3ubuntu6.4_amd64.deb ...
Unpacking qemu-kvm (1:4.2-3ubuntu6.4) ...
Selecting previously unselected package qemu-system-gui:amd64.
Preparing to unpack .../24-qemu-system-gui_1%3a4.2-3ubuntu6.4_amd64.deb ...
Unpacking qemu-system-gui:amd64 (1:4.2-3ubuntu6.4) ...
Selecting previously unselected package qemu-utils.
Preparing to unpack .../25-qemu-utils_1%3a4.2-3ubuntu6.4_amd64.deb ...

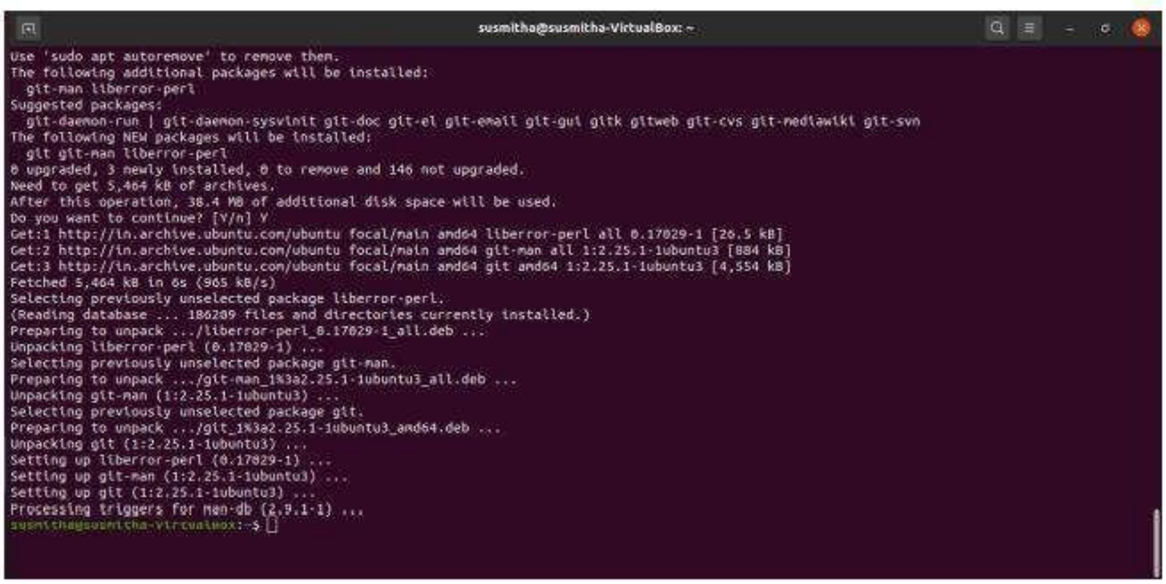

susmitha@susmitha-VirtualBox: ~
Setting up libfdt1:amd64 (1.5.1-1) ...
Setting up libusbredirparser1:amd64 (0.8.0-1) ...
Setting up libcacard0:amd64 (1:2.6.1-1) ...
Setting up ovmf (0~20191122.bd85bf54-2ubuntu3) ...
Setting up libvirglrenderer1:amd64 (0.8.2-1ubuntu1) ...
Setting up qemu-system-data (1:4.2-3ubuntu6.4) ...
Setting up seabios (1.13.0-1ubuntu1) ...
Setting up libslirp0:amd64 (4.1.0-2ubuntu2.1) ...
Setting up cpu-checker (0.7-1.1) ...
Setting up ipxe-qemu (1.0.0+git-20190109.133f4c4-0ubuntu3.2) ...
Setting up ipxe-qemu-256k-compat-efi-roms (1.0.0+git-20150424.a25a16d-0ubuntu4) ...
Setting up libaio1:amd64 (0.3.112-5) ...
Setting up libpmem1:amd64 (1.8-1ubuntu1) ...
Setting up librdmacm1:amd64 (28.0-1ubuntu1) ...
Setting up librados2 (15.2.3-0ubuntu0.20.04.1) ...
Setting up librbd1 (15.2.3-0ubuntu0.20.04.1) ...
Setting up libiscsi7:amd64 (1.18.0-2) ...
Setting up qemu-block-extra:amd64 (1:4.2-3ubuntu6.4) ...
Setting up qemu-system-common (1:4.2-3ubuntu6.4) ...
Created symlink /etc/systemd/system/multi-user.target.wants/qemu-kvm.service → /lib/systemd/system/qemu-kvm.service.
Setting up qemu-system-x86 (1:4.2-3ubuntu6.4) ...
Setting up qemu-utils (1:4.2-3ubuntu6.4) ...
Setting up qemu-kvm (1:4.2-3ubuntu6.4) ...
Processing triggers for install-info (6.7.0.dfsg.2-5) ...
Processing triggers for desktop-file-utils (0.24-1ubuntu3) ...
Processing triggers for mime-support (3.64ubuntu1) ...
Processing triggers for hicolor-icon-theme (0.17-2) ...
Processing triggers for gnome-menus (3.36.0-1ubuntu1) ...
Processing triggers for libc-bin (2.31-0ubuntu9) ...
Processing triggers for man-db (2.9.1-1) ...

<u>INSTALLING GIT REPOSITORY</u>:

Git repository is installed to clone XV6 from github.

<u>COMMAND:</u> sudo apt install git



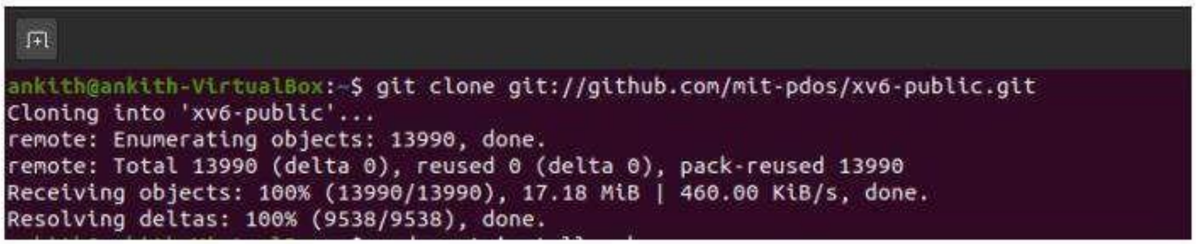


<u>CLONING XV6 FROM GITHUB:</u>

Using git and cloning XV6 OS from git://github.com/mit-pdos/xv6-public.git

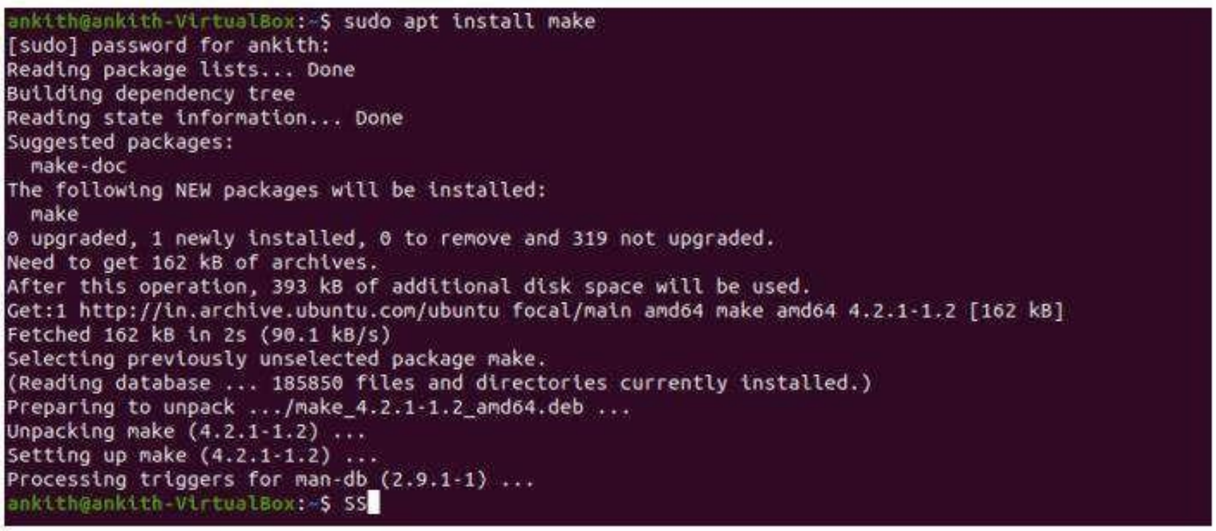
<u>COMMAND:</u>

git clone git://github.com/mit-pdos/xv6-public.git



<u>INSTALLING MAKE REPOSITORY</u>:

<u>COMMAND</u>: Sudo apt install make

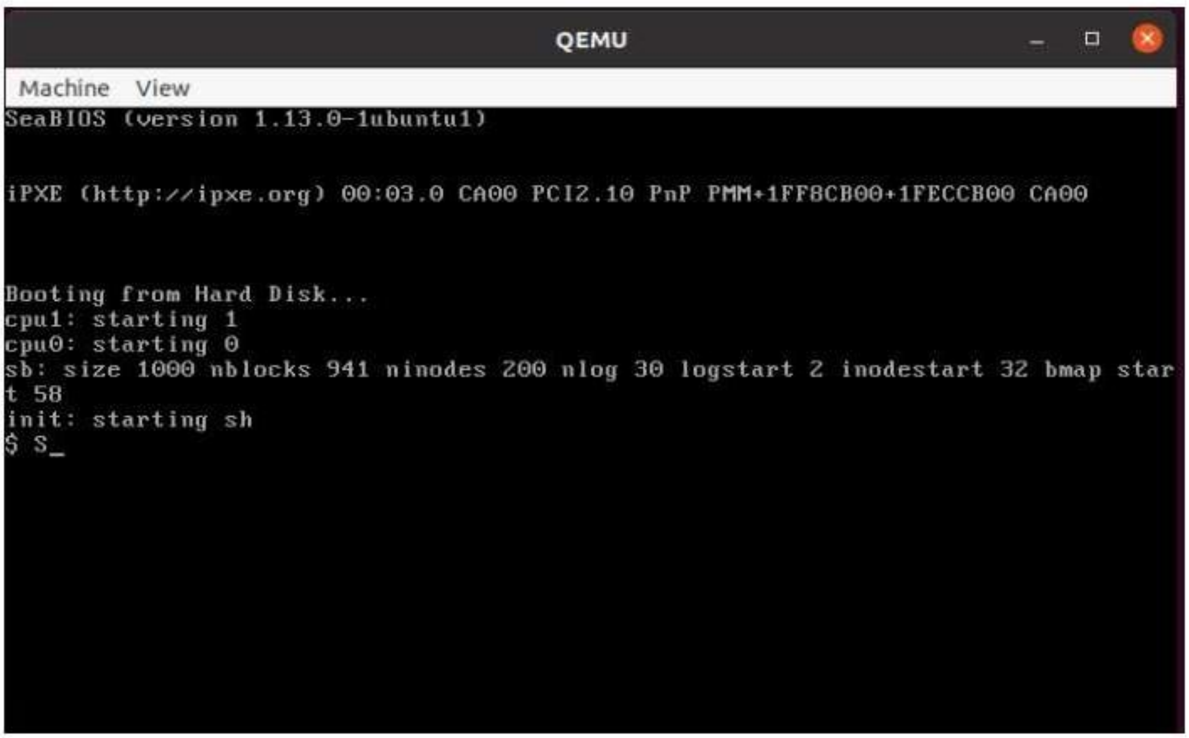


RUNNING XV6:

COMMAND:

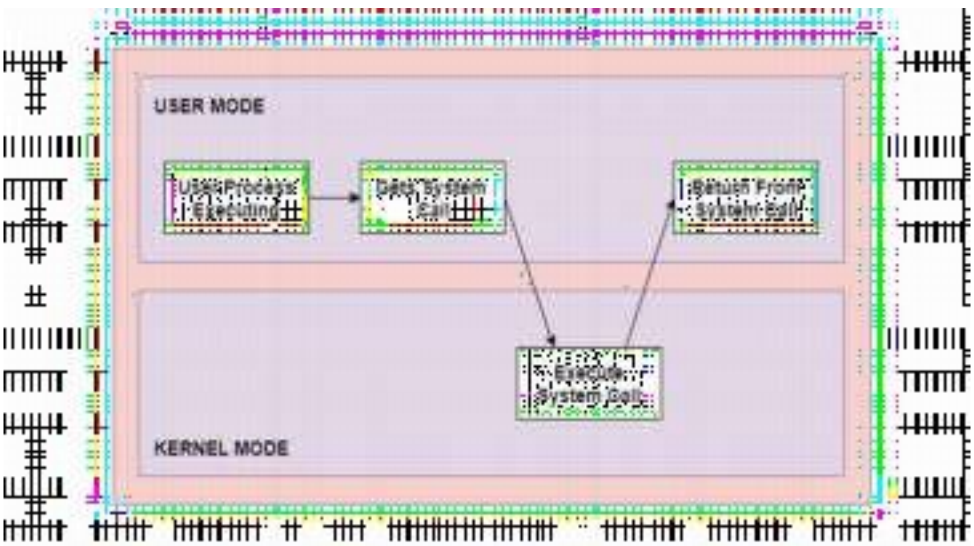
cd xv6-public

make qemu



System call:

A system call is way for programs to interact with operating system. System calls provide an interface to the services made available by an operating System. In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers. System calls are usually made when a process in user mode requires access to a resource or need service from kernel. Then it requests the kernel to provide the resource via a system call. A figure representing the execution of the system call is given as follows.



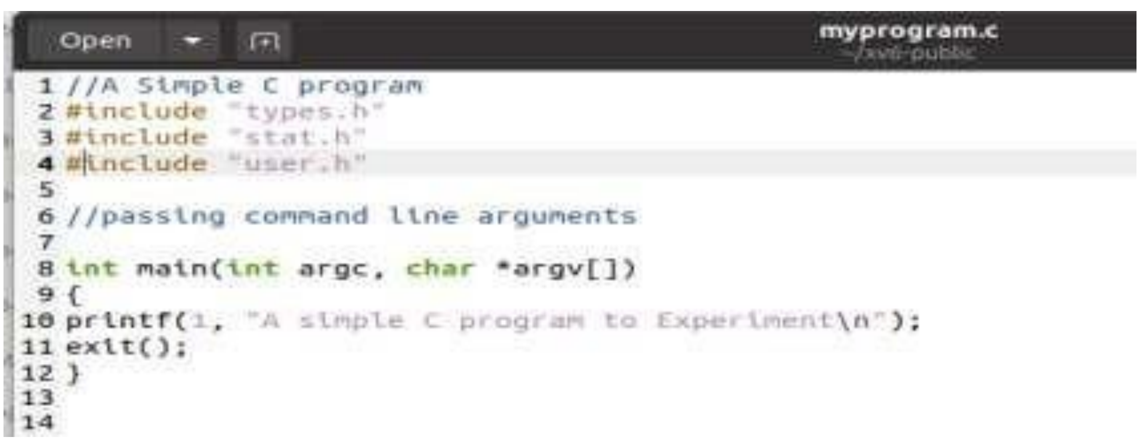
As can be seen from this diagram, the processes execute normally in the user mode until a system call interrupts this. Then the system call is executed on a priority basis in the kernel mode. After the execution of the system call, the control returns to the user mode and execution of user processes can be resumed.

Adding Simple User Program To Xv6:

First of all, We created a C program as shown in below image. We saved it inside the source code directory of xv6 operating system with the name myprogram.c

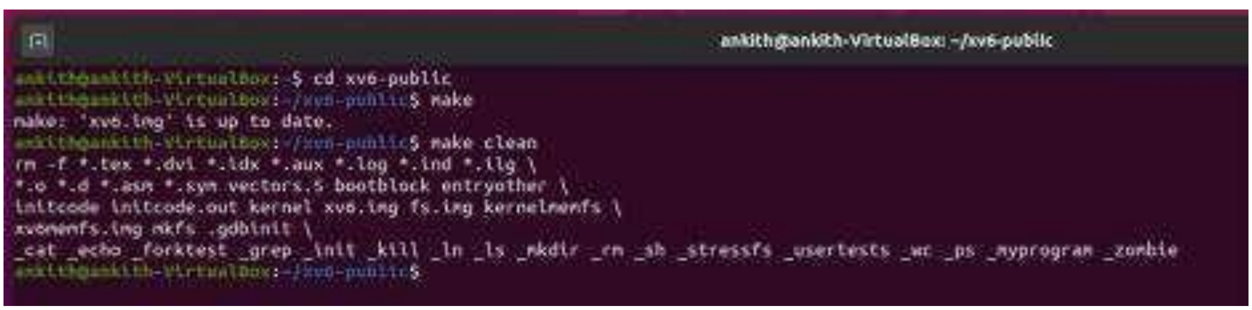
CODE:

```
#include “types.h”

#include “stat.h”

#include “user.h”

Int main(int argc,char *argv[ ])

{

      Printf(“a simple c program to experiment\n”);

      Exit();

}
```



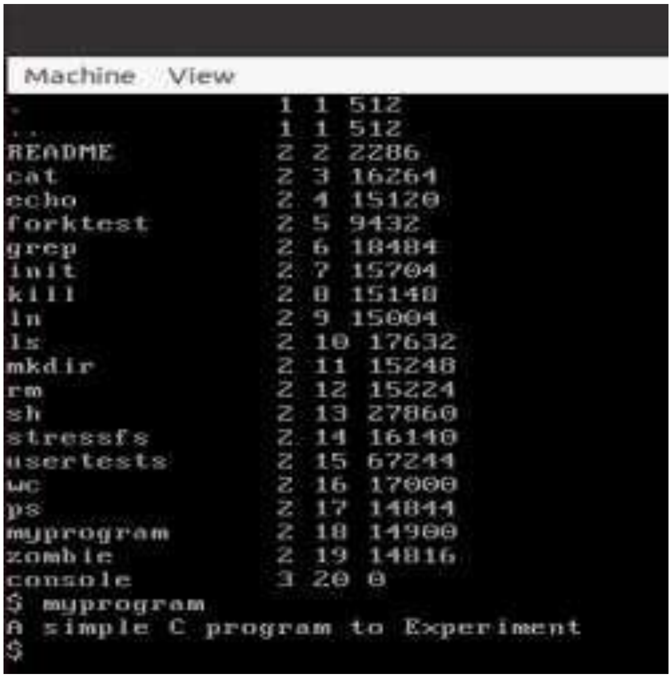
Makefile.c:

The Makefile needs to be edited to make our program available for the xv6 source code for compilation. The following sections of the Makefile needs to be edited to add our program myprogram.c



- Now, start xv6 system on QEMU and when it booted up, run ls command to check whether our program is available for the user.

- Here myprogram is availablein the list and by giving the name we can see the output of the Program In image below.



Adding New System Calls To xv6:

A system call is simply a kernel function that a user application can use to access or utilize system resources. Functions **fork**(), and **exec**() are well-known examples of system calls in UNIX and xv6. Here, we will use a simple example to walk you through the steps of adding a new system call to xv6. We name the system call **cps**(), which prints out the current running and sleeping processes.

An application signals the kernel it needs a service by issuing a software interrupt, a signal generated to notify the processor that it needs to stop its current task, and response to the signal request. Before switching to handling the new task, the processor has to save the current state, so that it can resume the execution in this context after the request has been handled. The following is a code that calls a system call in xv6 (found in *initcode.S*)

```
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL
```

Basically, it pushes the argument of the call to the stack, and puts the system call number, which is *$SYS_exec* in the example, into *%eax*. All the system call numbers are specified and saved in a table and the system calls of xv6 can be found in the file *syscall.h*. Next, the code *int $T_SYSCALL* generates a software interrupt, indexing the interrupt descriptor table to obtain the appropriate interrupt handler. The function **trap**() (in *trap.c*) is the specific code that finds the appropriate interrupt handler. It checks whether the trap number in the generated *trapframe* (a structure representing the processor's state at the time the trap happened) is equal to*T_SYSCALL*. If it is, it calls **syscall**(), the software interrupt handler that's available in *syscall.c*.

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

The function **syscall**() is the final function that checks out *%eax* to obtain the system call's number, which is used to index the table with the system call pointers, and to execute the code corresponding to that system call:

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

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

The following are the changes to be done to add our system call cps () to xv6:

1) Add name to *syscall.h* :

This defines the position of the system call vector that connects to the implementation.

CODE: #define sys_cps 22

Open *syscall.h ~/xv6-public

```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_cps    22
```

2) Add function prototype to *defs.h* :

This adds a forward declaration for the new system call. We add this function in proc.c

CODE: int cps(void);

```
// 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 );
```

3) Add function prototype to *user.h* :

It defines the function that can be called through the shell. We

add this function prototype in syscalls.

CODE: int cps(void);

```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
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 );
```

4) Add function call to *sysproc.c* :

We add the real implementation of our method here. We add a function sys_cps in the file sysproc.c which calls the function cps().

CODE:

Int sys_cps(void)

{

Return cps();

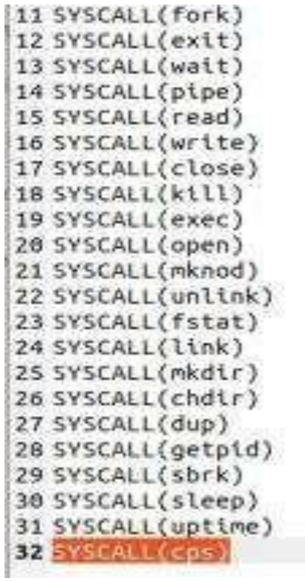
}

```
// return how many clock tick interrupts have occurred
// since start.
int
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}

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

5) Add call to *usys.S* :

It uses the macro to define connect the call of user to the system call function.

CODE: SYSCALL(cps)

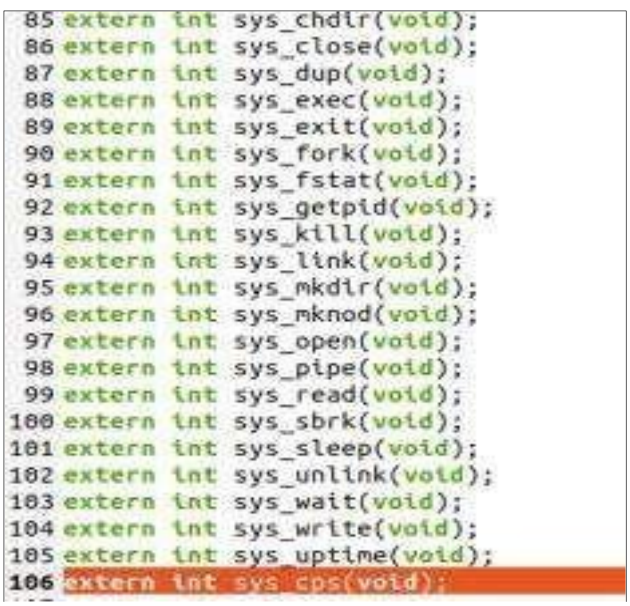


```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)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

6) Add call to *syscall.c* :

It defines the function that connects the kernel and the shell and by using the position defined in syscall.h it adds the function to the system call.

CODE: extern int sys_cps(void);



```
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_cps(void);
```

<u>CODE:</u> [SYS_cps] sys_cps,

```
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,
```

<u>7) Add code to *proc.c* :</u>

We add this code to proc.c as written below.

It interrupts on the processor. It acquires a lock. It runs through the process table and checks whether the process is SLEEPING or RUNNING or RUNNABLE and then prints the same pid and status of the process. It releases the lock. It returns the syscall number which is 22.

CODE:

Int cps()

{

struct proc *p;   // Enable interrupts on this

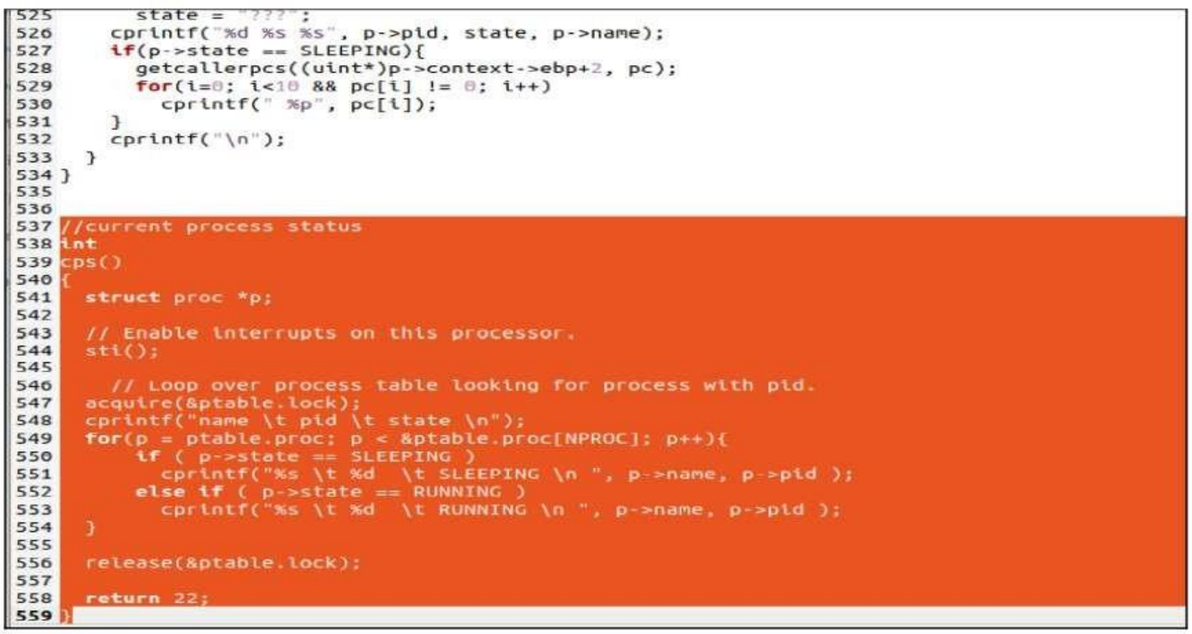
processor. sti();

// Loop over process table looking for process with pid.

acquire(&ptable.lock); // acquiring lock before use of critical

section

```
    cprintf("name \t pid \t state \n");

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) // checking process table

    {

      if(p->state == SLEEPING)

        cprintf("%s \t %d \t SLEEPING \n", p->name, p->pid); // printing pid,pname,state

      else if(p->state == RUNNING)

        cprintf("%s \t %d \t RUNNING \n", p->name, p->pid);

    }

    release(&ptable.lock);    // releasing acquired

    lock return 22;

}
```



```
      state = "???";
    cprintf("%d %s %s", p->pid, state, p->name);
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
    cprintf("\n");
  }
}


//current process status
int
cps()
{
  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 \n");
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if ( p->state == SLEEPING )
        cprintf("%s \t %d  \t SLEEPING \n ", p->name, p->pid );
      else if ( p->state == RUNNING )
        cprintf("%s \t %d  \t RUNNING \n ", p->name, p->pid );
  }

  release(&ptable.lock);

  return 22;
}
```

8) Create testing file *ps.c* with code shown below: CODE:

include “types.h”

#include “stat.h”

#include “user.h”

#include “fnctl.h”

Int main(int argc, char *argv[ ])

{

Cps(); // calling cps

function exit();

}

```
ps.c
~/xv6-public

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

int
main(int argc, char *argv[])
{
  cps();

  exit();
}
```

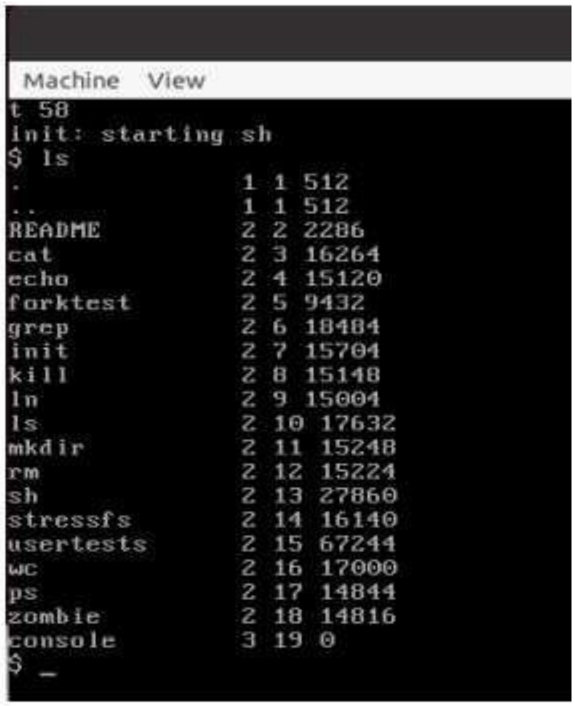
9) Modify* Makefile *:

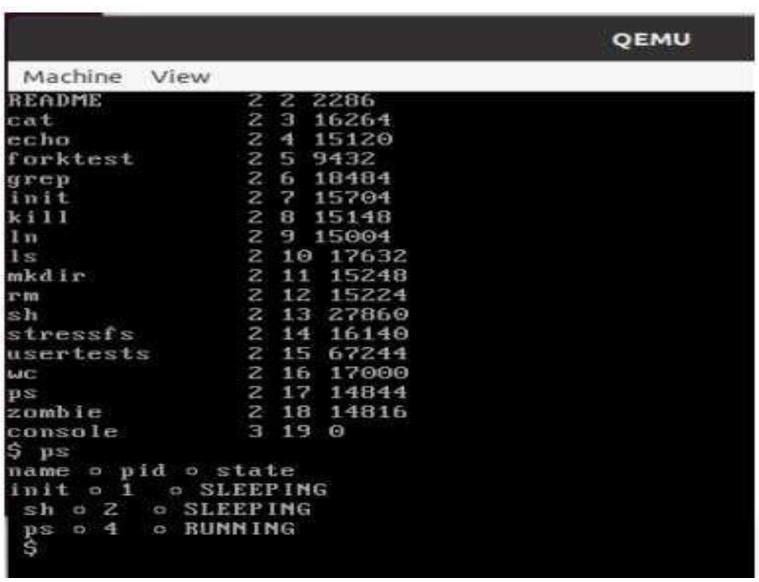
Then we make which compiles all changes we made inside the xv6 directories and subdirectories.

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

Output:

Now we will compile the whole code and execute the OS after the above changes are made.Our new syscall is now visible in the list:





IMPLEMENTATION OF PRIORITY SCHEDULING:

An overview of Priority scheduling:

Priority scheduling is one of the most common scheduling algorithms in batch systems. Priority scheduling is a method of scheduling processes based on priority. In this method, the scheduler chooses the tasks to work as per the priority. Each process is assigned a priority. Process with the highest priority is to be executed first and so on.

Processes with the same priority are executed on first come first served basis.

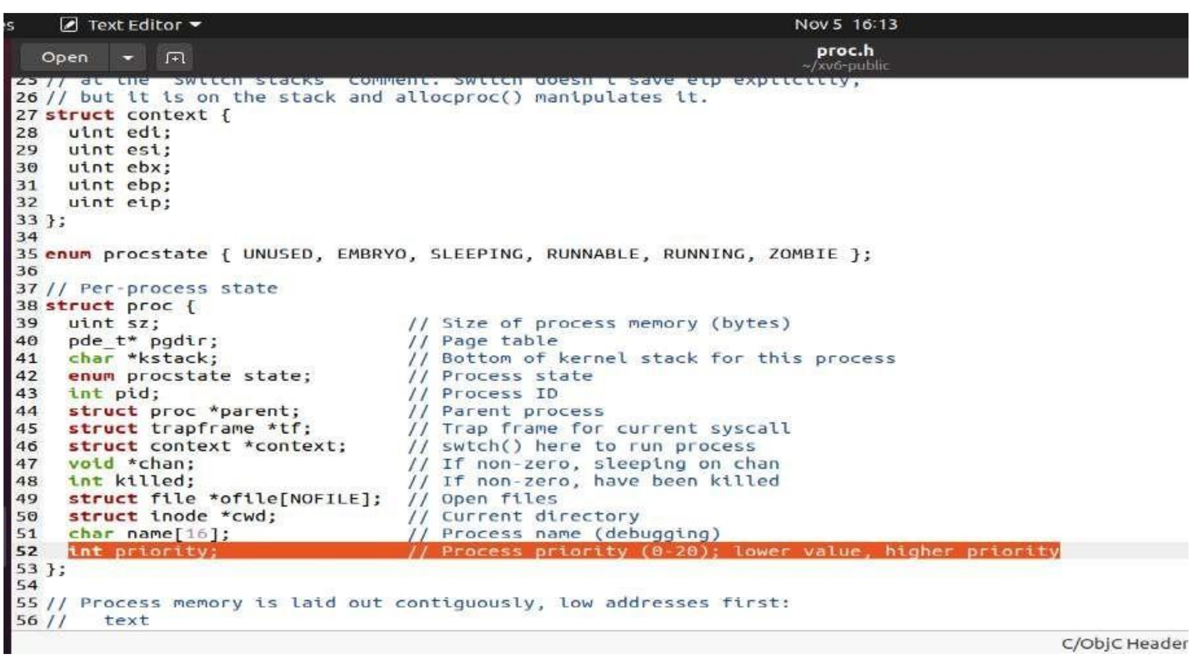
Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Default scheduling algorithm in XV6 operating system is round robin scheduling algorithm. It is not effective. It has more waiting time but priority scheduling algorithm reduces average waiting time.

1. **Add priority to struct proc in proc.h:**

Struct proc in proc.h is typically like PCB(process control block).It consists information about all the processes in the system. We add attribute priority in the struct proc which represents the priority of the process.

CODE: int priority; // process priority



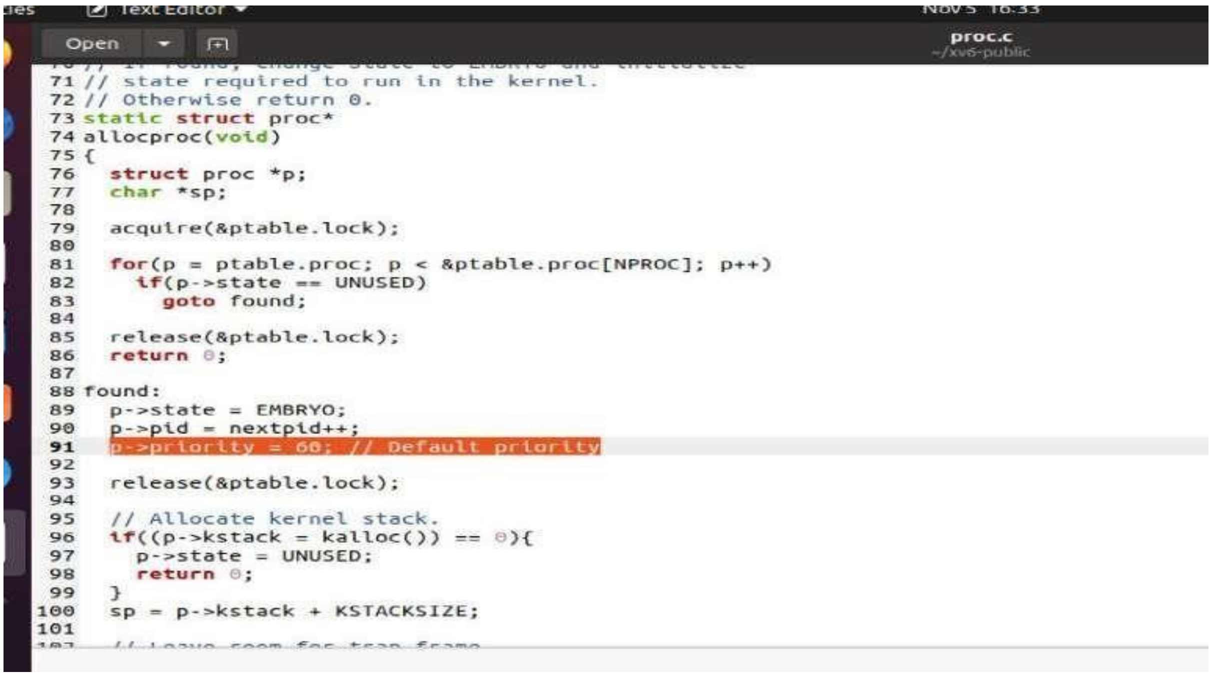

```
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

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 (0-20); lower value, higher priority
};

// Process memory is laid out contiguously, low addresses first:
//   text
```

2. **<u>Assign a default priority in proc.h:</u>**

allocproc is a function that allocates resources to new process. It scans the entire process table and if it finds an unused entry then it will assign pid and resources to process. Here we set default priority for all the process to 60 in allocproc function.

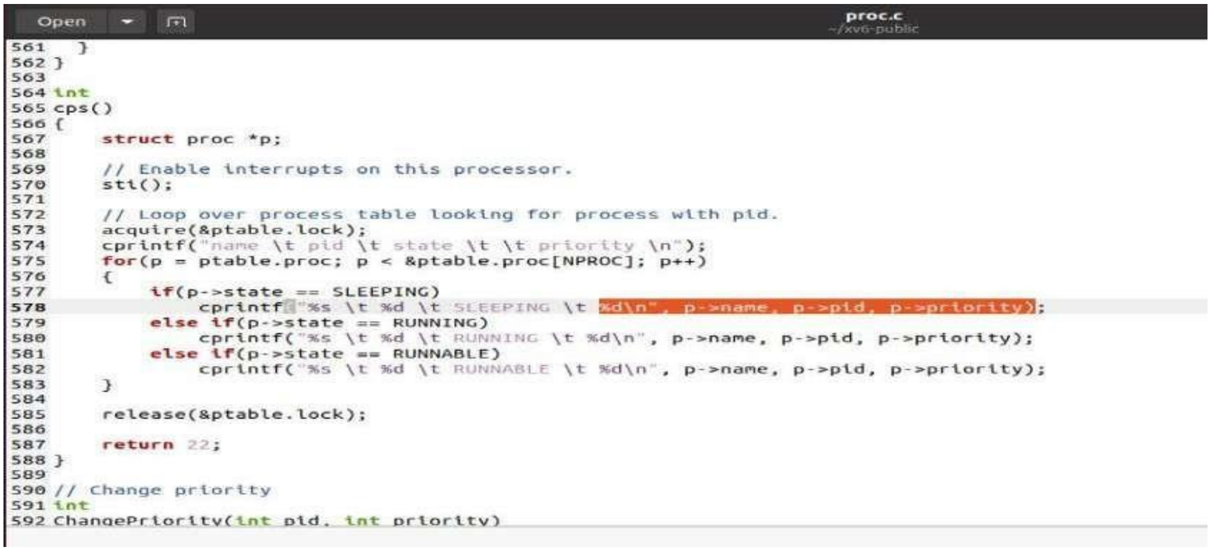
<u>CODE:</u>

```
p->priority=60;        //default priority set to 60
```



3. **<u>Adding code to print priority of process in cps in proc.c:</u>**

We have added code in cps function in proc.c to print priority of process along with process id, current process state and process name.



```
    }
}

int
cps()
{
    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 \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;
}

// Change priority
int
ChangePriority(int pid, int priority)
```

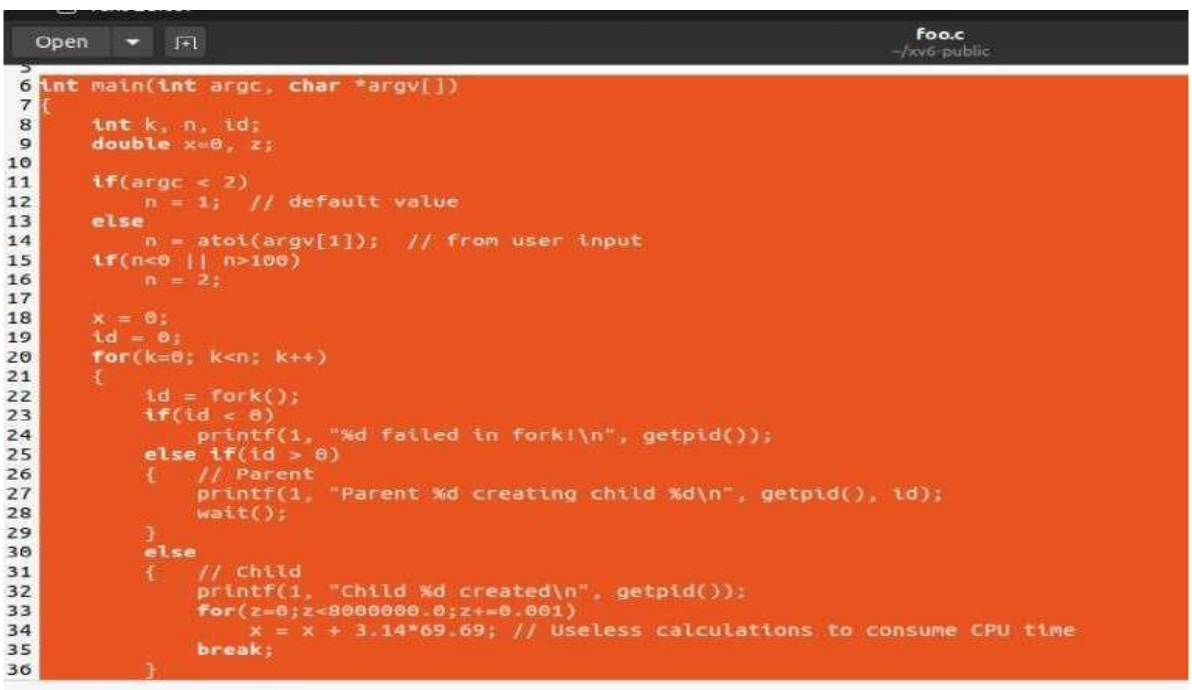
4. **Creation of dummy program foo.c:**

We create a dummy program named as foo.c and this dummy program will create a child and do sum dummy computations or calculations to waste CPU time. The main in this program take 2 arguments. The first argument is number of child processes that has to be created. We have used fork system call in this program to create child process.

CODE:

```
#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 user
   input if(n<0 || n>100)
      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+=0.001)
            x = x + 3.14*69.69;      // Useless calculations to consume CPU
        time break;
    }
}
```



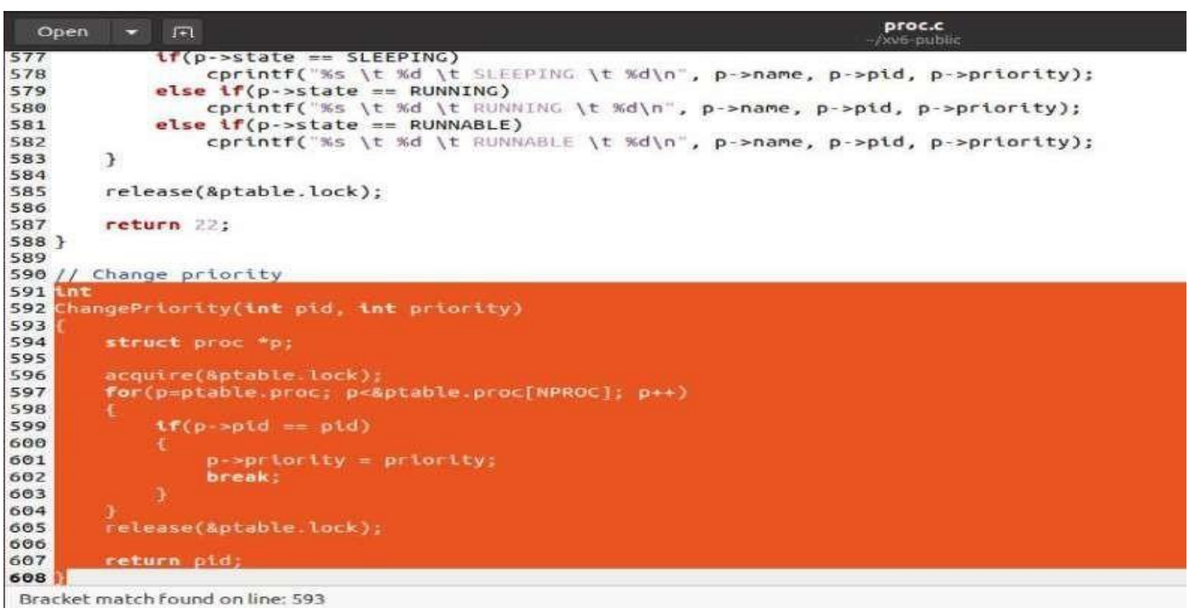

```
int main(int argc, char *argv[])
{
    int k, n, id;
    double x=0, z;

    if(argc < 2)
        n = 1;   // default value
    else
        n = atoi(argv[1]);   // from user input
    if(n<0 || n>100)
        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+=0.001)
                x = x + 3.14*69.69; // Useless calculations to consume CPU time
            break;
        }
```

5. Addition of new function chpr(change priority) to proc.c:

We add a new function named as chpr in proc.c file. This function takes two arguments. First argument is process id, and second argument is the priority, This function changes the priority of given process id.

CODE:

```
// Change priority
int
ChangePriority(int pid, int priority)
{
   struct proc *p;
   acquire(&ptable.lock);       // acquire lock to access critical
   section for(p=ptable.proc; p<&ptable.proc[NPROC]; p++)
   {
      if(p->pid == pid)    // checking process table
      {
         p->priority = priority;    //changing priority of
         process break;
      }
   }
   release(&ptable.lock);
   return pid;  }
```

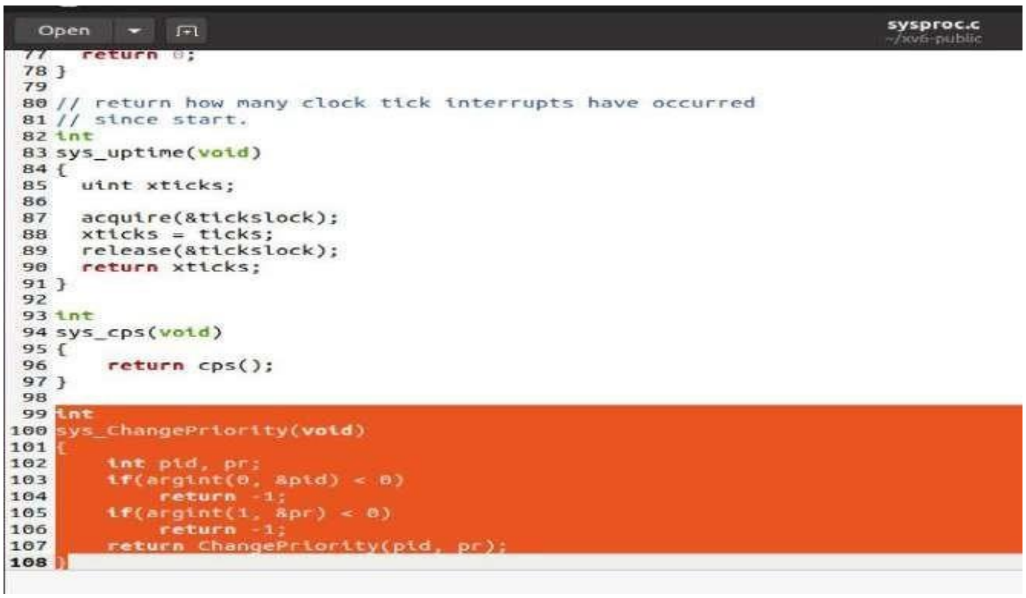


```
Open
proc.c
~/xv6-public
        if(p->state == SLEEPING)
            cprintf("%s \t %d \t SLEEPING \t %d\n", p->name, p->pid, p->priority);
        else if(p->state == RUNNING)
            cprintf("%s \t %d \t RUNNING \t %d\n", p->name, p->pid, p->priority);
        else if(p->state == RUNNABLE)
            cprintf("%s \t %d \t RUNNABLE \t %d\n", p->name, p->pid, p->priority);
    }

    release(&ptable.lock);

    return 22;
}

// Change priority
int
ChangePriority(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;
}
Bracket match found on line: 593
```

6. **<u>Adding system call(chpr):</u>**

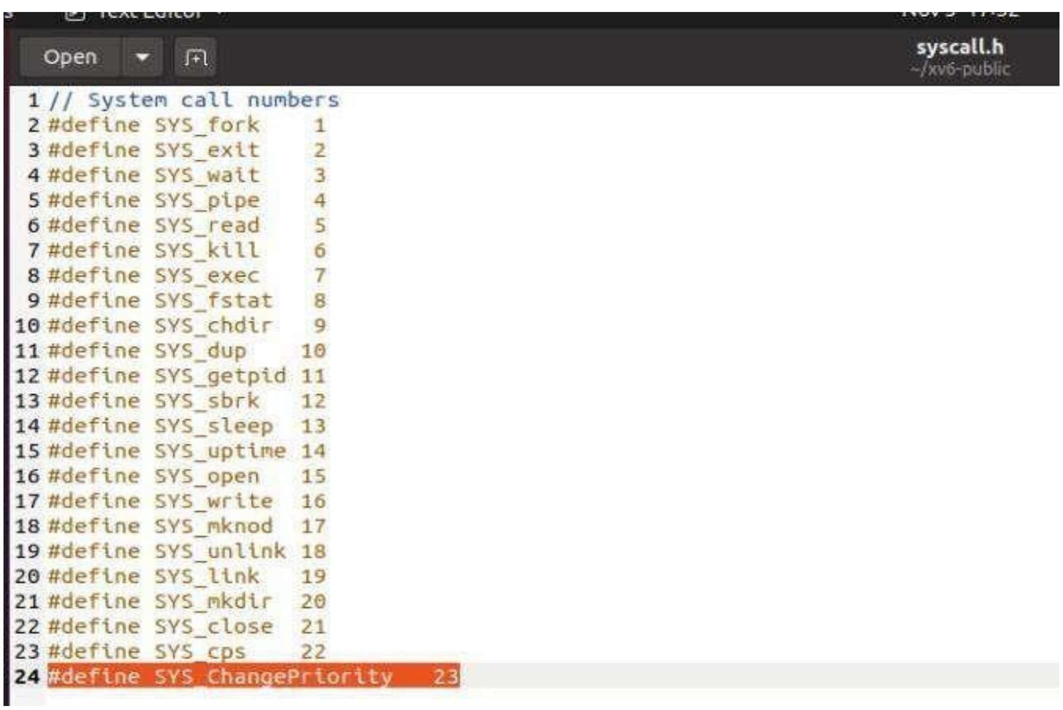
As we have added system call cps, we have to follow same steps to add system call chpr in xv6 operating system.

- **<u>Add sys_chpr to syspoc.c:</u>**




```
  return 0;
}

// return how many clock tick interrupts have occurred
// since start.
int
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}

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

int
sys_ChangePriority(void)
{
    int pid, pr;
    if(argint(0, &pid) < 0)
        return -1;
    if(argint(1, &pr) < 0)
        return -1;
    return ChangePriority(pid, pr);
}
```

- **<u>Adding to syscall.h:</u>**




```
// 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_ChangePriority   23
```

- **<u>Modify defs.h:</u>**

```c
void            picenable(int);
void            picinit(void);

// pipe.c
int             pipealloc(struct file**, struct file**);
void            pipeclose(struct pipe*, int);
int             piperead(struct pipe*, char*, int);
int             pipewrite(struct pipe*, char*, int);

//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             ChangePriority(int pid, int priority);

// swtch.S
```

- **<u>Modify user.h:</u>**

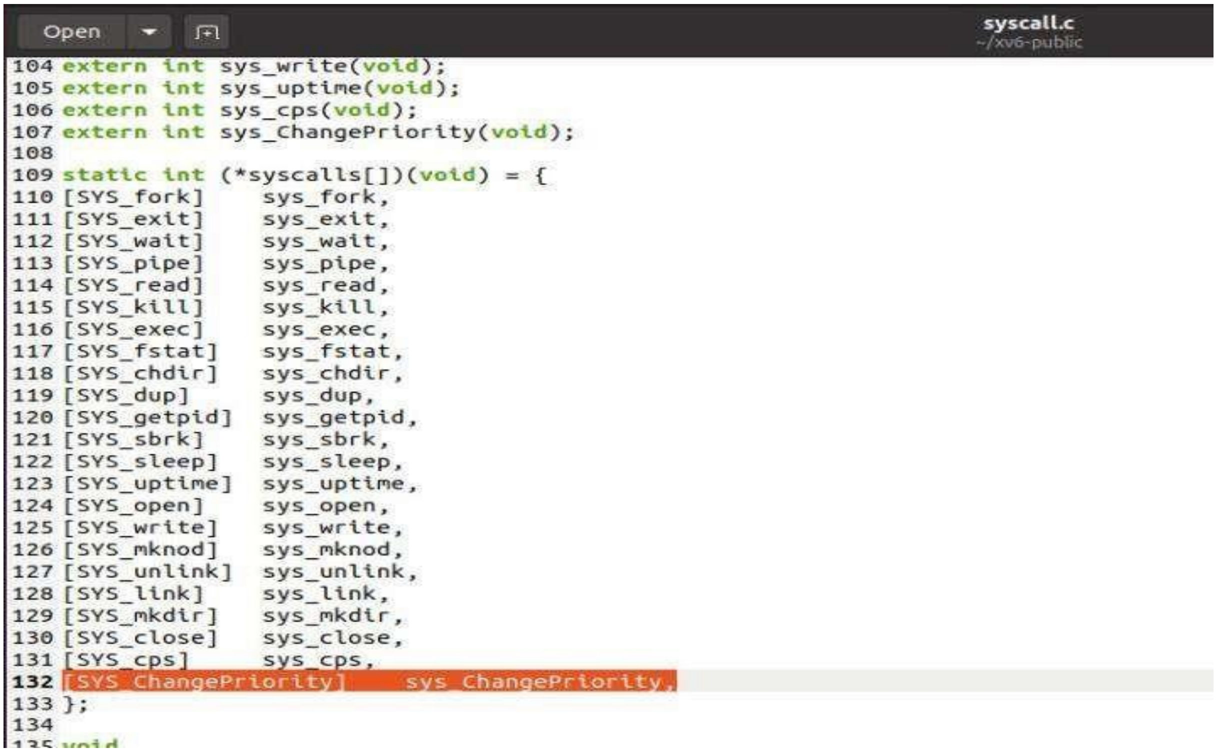
```c
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
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(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int cps(void);
int ChangePriority(int pid, int priority);

// ulib.c
int stat(const char*, struct stat*);
char* strcpy(char*, const char*);
void *memmove(void*, const void*, int);
```

- **Modify usys.s:**

```
usys.S
~/xv6-public
#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(ChangePriority)
```

Modify syscall.c:

```
syscall.c
~/xv6-public
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_cps(void);
extern int sys_ChangePriority(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_ChangePriority]    sys_ChangePriority,
};
```




```
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_cps(void);
extern int sys_ChangePriority(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_ChangePriority]    sys_ChangePriority,
};
```

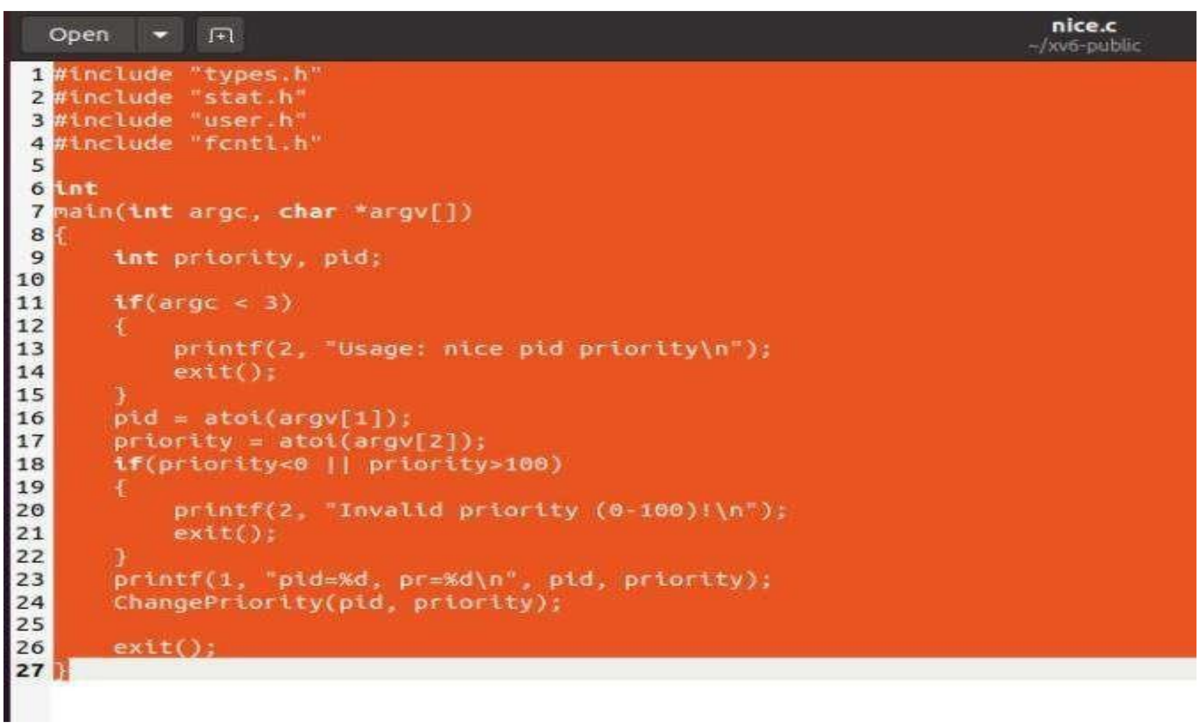
6. <u>Adding nice.c program:</u>

This user program will call system call chpr(change priority) to change priority of the process.

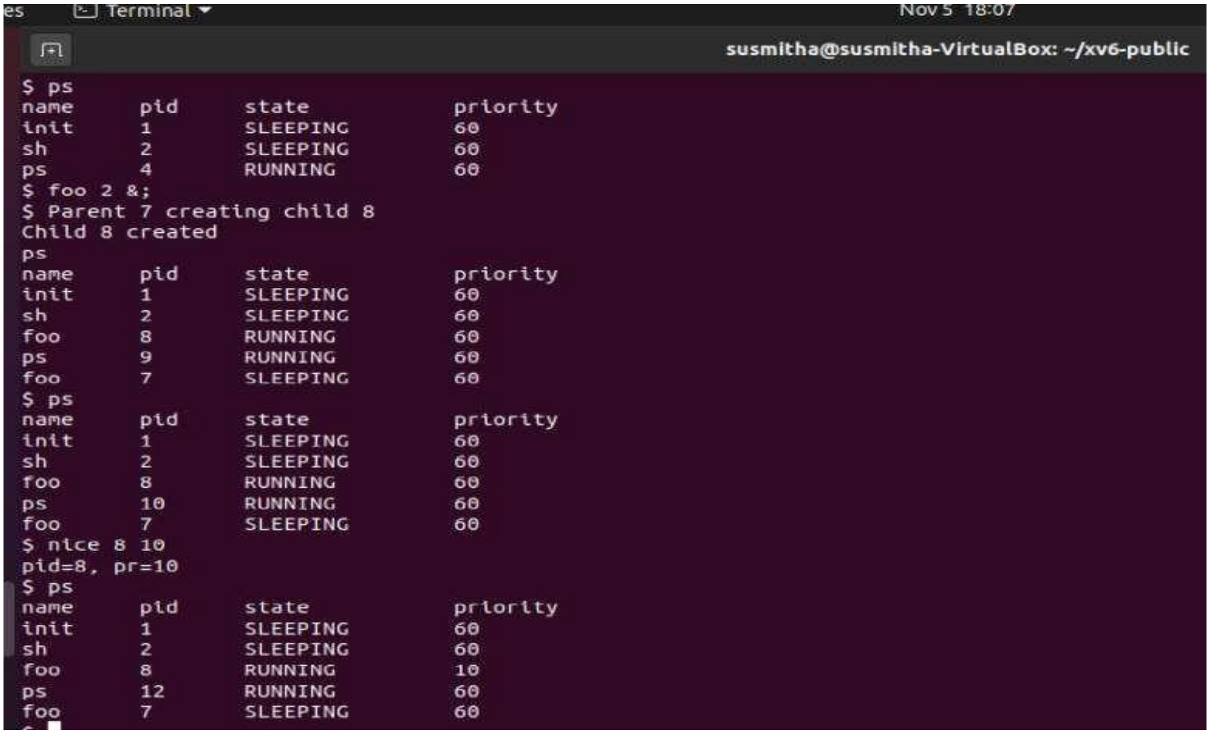
<u>CODE:</u>

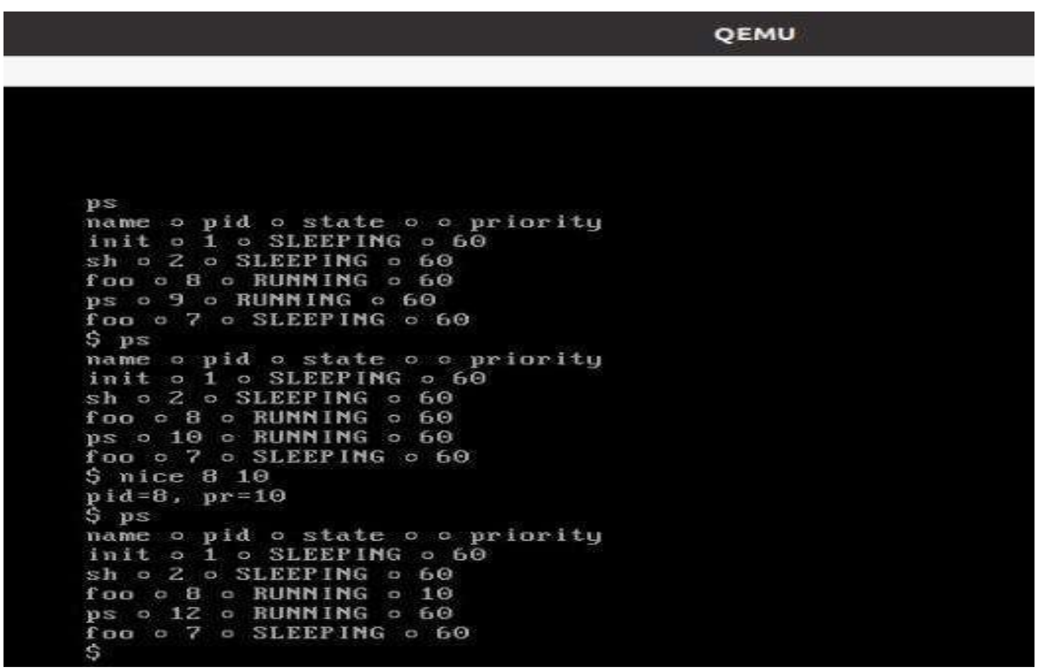
```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

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>100)
    {
        printf(2, "Invalid priority (0-100)!\n");
        exit();
    }
    printf(1, "pid=%d, pr=%d\n", pid, priority);
    ChangePriority(pid, priority);

    exit();
}
```



OUTPUT:


es
Terminal
Nov 5 18:07
susmitha@susmitha-VirtualBox: ~/xv6-public
$ ps
name pid state priority
init 1 SLEEPING 60
sh 2 SLEEPING 60
ps 4 RUNNING 60
$ foo 2 &;
$ Parent 7 creating child 8
Child 8 created
ps
name pid state priority
init 1 SLEEPING 60
sh 2 SLEEPING 60
foo 8 RUNNING 60
ps 9 RUNNING 60
foo 7 SLEEPING 60
$ ps
name pid state priority
init 1 SLEEPING 60
sh 2 SLEEPING 60
foo 8 RUNNING 60
ps 10 RUNNING 60
foo 7 SLEEPING 60
$ nice 8 10
pid=8, pr=10
$ ps
name pid state priority
init 1 SLEEPING 60
sh 2 SLEEPING 60
foo 8 RUNNING 10
ps 12 RUNNING 60
foo 7 SLEEPING 60


QEMU
ps
name pid state priority
init 1 SLEEPING 60
sh 2 SLEEPING 60
foo 8 RUNNING 60
ps 9 RUNNING 60
foo 7 SLEEPING 60
$ ps
name pid state priority
init 1 SLEEPING 60
sh 2 SLEEPING 60
foo 8 RUNNING 60
ps 10 RUNNING 60
foo 7 SLEEPING 60
$ nice 8 10
pid=8, pr=10
$ ps
name pid state priority
init 1 SLEEPING 60
sh 2 SLEEPING 60
foo 8 RUNNING 10
ps 12 RUNNING 60
foo 7 SLEEPING 60
$

Copy on write fork:

The fork() system call in xv6 copies all of the parent process's user-space memory into the child. If the parent is large, copying can take a long time.the work is often largely wasted; for example, a fork() followed by exec() in the child will cause the child to discard the copied memory, probably without ever using most of it

The goal of copy-on-write (COW) fork() is to defer allocating and copying physical memory pages for the child until the copies are actually needed, if ever. Copy-On-Write avoids this expense by being lazy. Rather than copy all the memory at once it pretends it was copied and only actually copies when the parent and child need to hold different values at the same address.

COW fork() creates just a pagetable for the child, with PTEs for user memory pointing to the parent's physical pages. COW fork() marks all the user PTEs in both parent and child as not writable. When either process tries to write one of these COW pages, the CPU will force a page fault. The kernel page-fault handler detects this case, allocates a page of physical memory for the faulting process, copies the original page into the new page, and modifies the relevant PTE in the faulting process to refer to the new page, this time with the PTE marked writeable. When the page fault handler returns, the user process will be able to write its copy of the page.

Implementation of Copy-on-write System Call:

Adding getNumFreePages system call:

Add the variable numFreePages to implement the given system call in kalloc.c :

In kalloc.c add the entry in struct kmem:

```
int numFreePages;
```

The system call getNumFreePages() should return the total number of free pages in the system. This system call will help you see when pages are consumed, and can help you debug your CoW implementation. You must add code to maintain and track freepages in kalloc.c, and access this information when this system call is invoked.

```
Open
// Physical memory allocator
// memory for user processes
// and pipe buffers. Allocat

#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "spinlock.h"

void freerange(void *vstart,
extern char end[]; // first
file

struct run {
  struct run *next;
};

struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
  int numFreePages;
} kmem;

```

<u>**Adding given system calls**</u>: add following code in defs.h int

getNumFreePages(void);

The file defs.h acts as the header file for several parts of the kernel code.

```
Open                    defs.h
                  ~/Downloads/xv6
int             writei(struct inode*, char*, u

// ide.c
void            ideinit(void);
void            ideintr(void);
void            iderw(struct buf*);

// ioapic.c
void            ioapicenable(int irq, int cpu)
extern uchar    ioapicid;
void            ioapicinit(void);

// kalloc.c
char*           kalloc(void);
void            kfree(char*);
void            kinit1(void*, void*);
void            kinit2(void*, void*);
int                    getNumFreePages(void);

```

Add following code in sysproc.c:

```
int
sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(proc->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}

// return how many clock tick interrupts have
// since start.
int
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}

int
sys_getNumFreePages(void)
{
  return getNumFreePages();
}
```

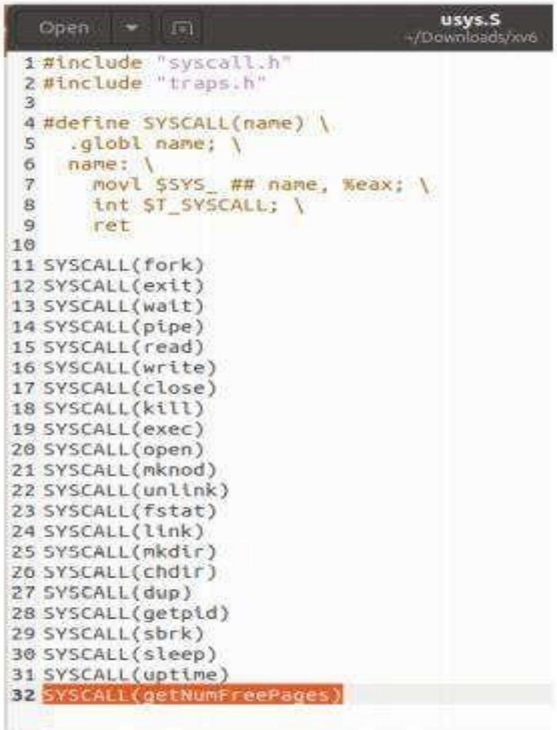
Add the following declaration along with other system calls in syscall.c:

```
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_getNumFreePages(void);
```

Add following fields in the same file like other system calls in syscall.c:

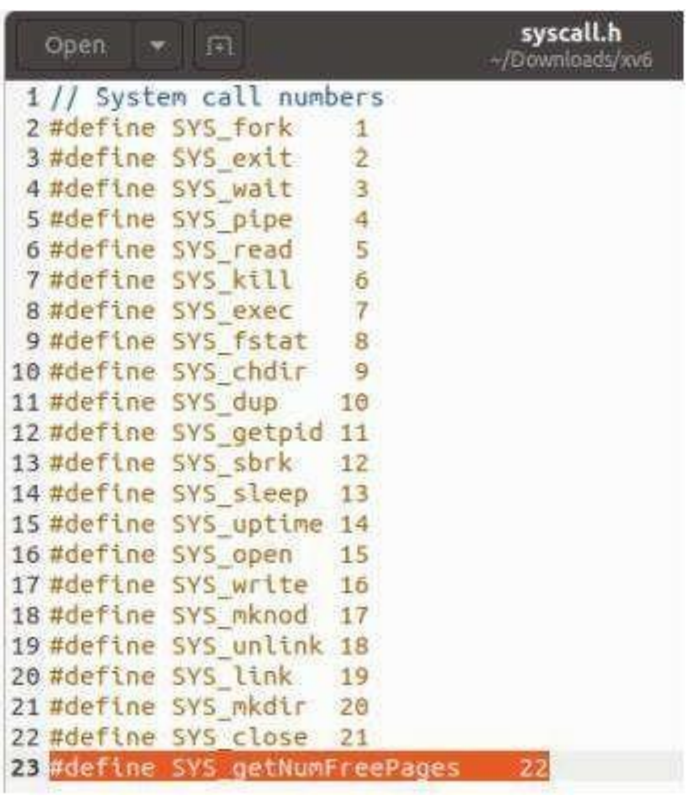
```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[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_getNumFreePages] sys_getNumFreePages,
};
```

Add the following lines in 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(getNumFreePages)
```

Add following lines in syscall.h:



```
// 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_getNumFreePages    22
```

Add the function body of getNumFreePages in kalloc.c:

The files vm.c and kalloc.c contain most of the logic for memory management in the xv6 kernel

```
  kmem.numFreePages = kmem.numFreePages
to freelist so increase the number of fr
  if(kmem.use_lock)
    release(&kmem.lock);
}

// Allocate one 4096-byte page of physic
// Returns a pointer that the kernel car
// Returns 0 if the memory cannot be all
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
  {
    kmem.freelist = r->next;
    kmem.numFreePages = kmem.numFreePage
out from the freelist so decrease the nu
  }
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}


// Returns the number of free pages.
int
getNumFreePages(void)
{
  if(kmem.use_lock)
    acquire(&kmem.lock);
  int r = kmem.numFreePages;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (r);
}
```

Initialize numFreePages in kinit1:

```
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
  int numFreePages;
} kmem;

// Initialization happens in two phases.
// 1. main() calls kinit1() while still
just
// the pages mapped by entrypgdir on fre
// 2. main() calls kinit2() with the res
// after installing a full page table th
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  kmem.numFreePages = 0;
  freerange(vstart, vend);
}

```

In kfree:

```
kalloc.c
~/xv6
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
//PAGEBREAK: 21
// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  kmem.numFreePages = kmem.numFreePages + 1; // A new node is added to freelist so increase the number of free pages
```

In kalloc:

```
kalloc.c
~/xv6

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  kmem.numFreePages = kmem.numFreePages + 1; // A new node is added to freelist so increase the number of free pages
  if(kmem.use_lock)
    release(&kmem.lock);
}

// Allocate one 4096 byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
  {
    kmem.freelist = r->next;
    kmem.numFreePages = kmem.numFreePages - 1 ; // A node is popped out from the freelist so decrease the number of free pages.
  }
```

Reinstalling of page table:

Whenever the flags are changed in copyuvm function the page table must be reinstalled using:

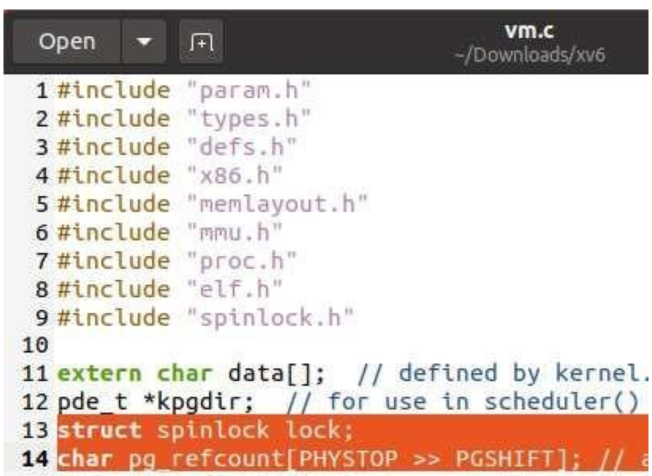
```
lcr3(v2p(pgdir)); // reinstall the page table
```

Keeping track of the reference count of pages:

In vm.c add the declaration of array and lock:

Begin with changes to kalloc.c. To correctly implement CoW fork, you must track reference counts of memory pages. A reference count of a page should indicate the number of processes that map the page into their virtual address space. The reference count of a page is set to one when a freepage is allocated for use by some process.

```
struct spinlock lock;
char pg_refcount[PHYSTOP >> PGSHIFT]; // array to store refcount
```

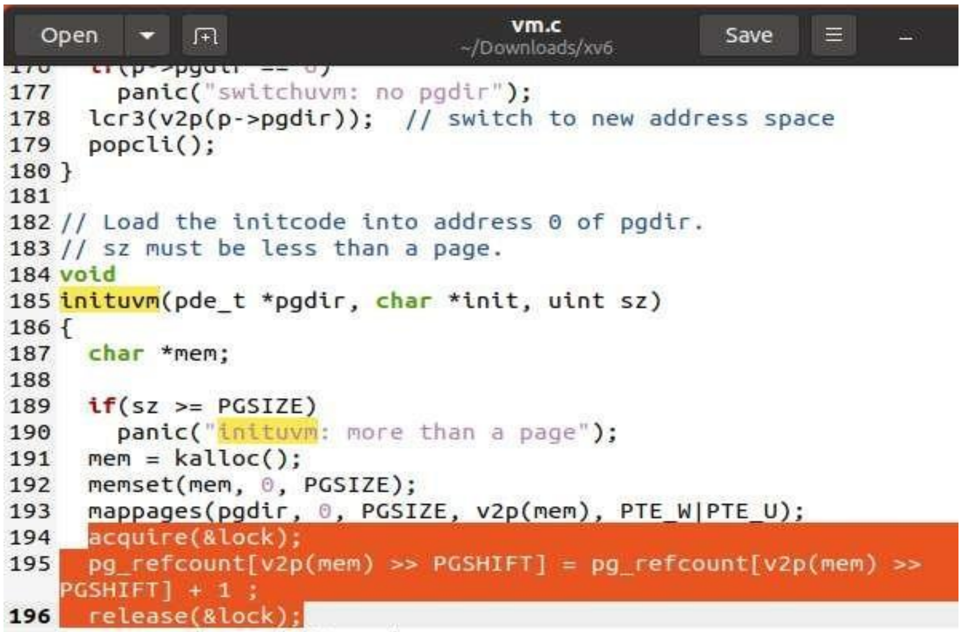



```
#include "param.h"
#include "types.h"
#include "defs.h"
#include "x86.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "elf.h"
#include "spinlock.h"

extern char data[];  // defined by kernel.
pde_t *kpgdir;  // for use in scheduler()
struct spinlock lock;
char pg_refcount[PHYSTOP >> PGSHIFT]; // a
```

In inituvm function:

When a freepage is allocated for use by some process. Whenever an additional process points to an already existing page (e.g., when parent forks a child and both share the same memory page), the reference count must be incremented.

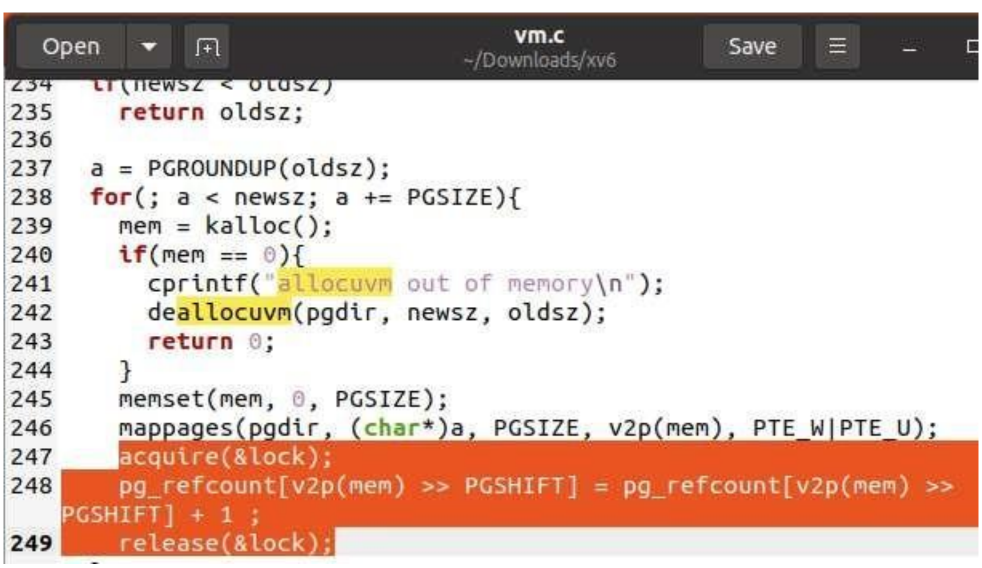
acquire(&lock);
pg_refcount[v2p(mem) >> PGSHIFT] = pg_refcount[v2p(mem) >> PGSHIFT] +1 ;
release(&lock);




```
    panic("switchuvm: no pgdir");
  lcr3(v2p(p->pgdir));  // switch to new 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);
  acquire(&lock);
  pg_refcount[v2p(mem) >> PGSHIFT] = pg_refcount[v2p(mem) >>
PGSHIFT] + 1 ;
  release(&lock);
```

<u>**In allocuvm function:**</u>

acquire(&lock);
pg_refcount[v2p(mem) >> PGSHIFT] = pg_refcount[v2p(mem) >> PGSHIFT] +1 ;
release(&lock);

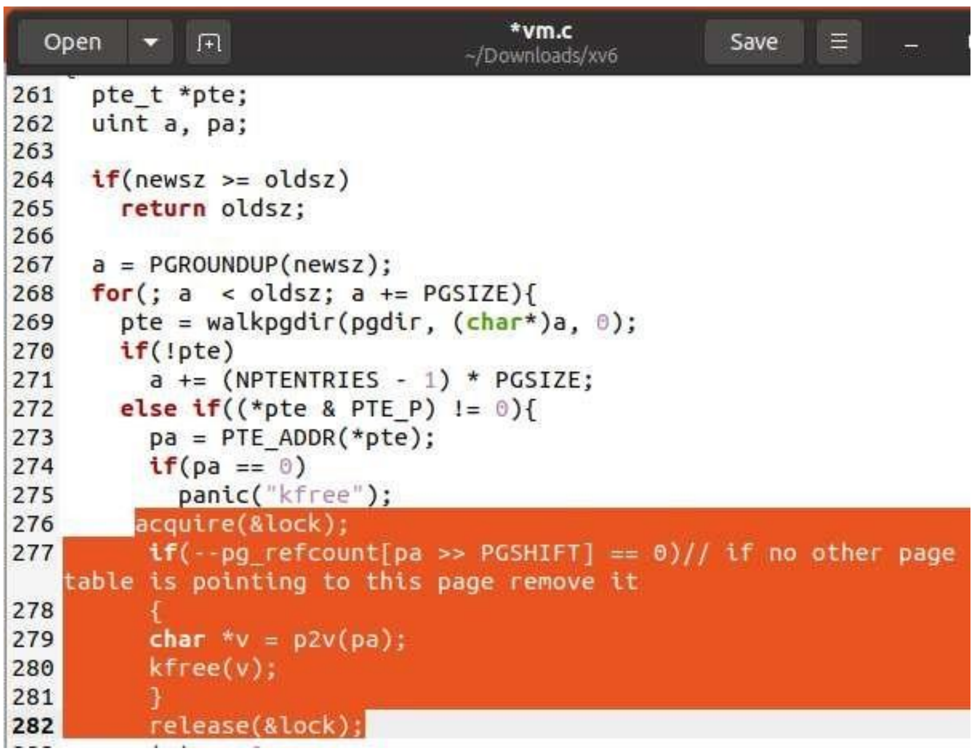



```
    return oldsz;

  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);
    mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
    acquire(&lock);
    pg_refcount[v2p(mem) >> PGSHIFT] = pg_refcount[v2p(mem) >>
PGSHIFT] + 1 ;
    release(&lock);
```

In deallocuvm free the page only when no other page table is pointing it:

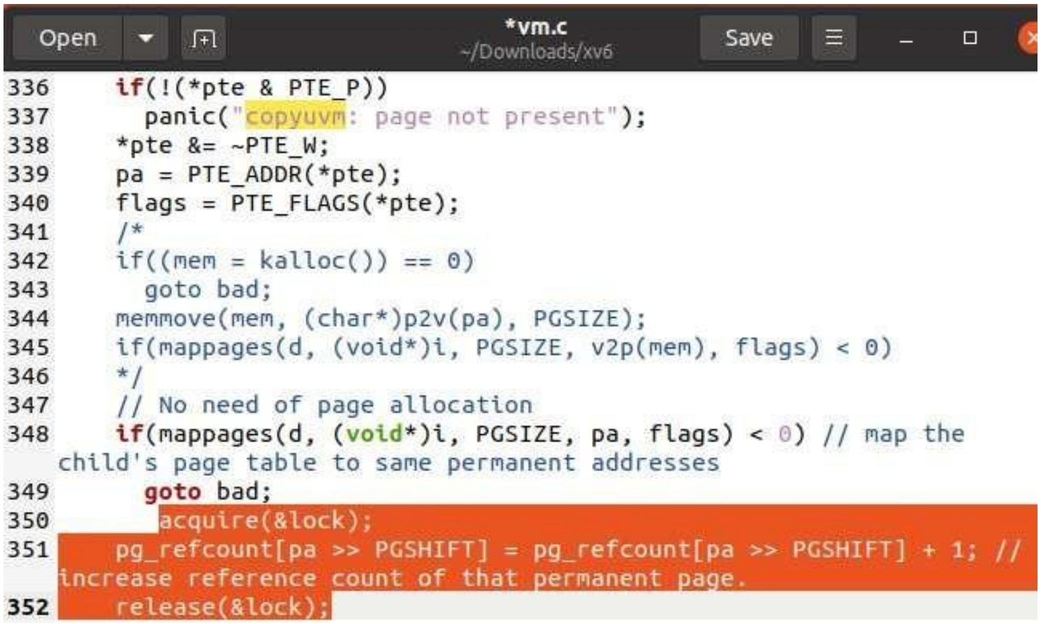
The reference count must be decremented when a process no longer points to the page from its page table. A page can be freed up and returned to the freelist only when there are no active references to it, i.e., when its reference count is zero.

```
acquire(&lock);
  if(--pg_refcount[pa >> PGSHIFT] == 0)// if no other page table
  is pointing to this page remove it
  {
  char *v = p2v(pa);
  kfree(v);
  }
  release(&lock);
```



In copyuvm function when a process is forked the refcount of that permanent address should be incremented:

```
acquire(&lock);
pg_refcount[pa >> PGSHIFT] = pg_refcount[pa >> PGSHIFT] + 1; //
increase reference count of that permanent page.
release(&lock);
```



<u>Change of copyuvm function:</u>

Make the pagetable unwritable and then assign the same permanent addresses to the new page table

```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
pde_t *d;
pte_t *pte;
uint pa, i, flags;
//char *mem; //No need to allocate new memory
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");
*pte &= ~PTE_W; // make this page table unwritable
pa = PTE_ADDR(*pte);
flags = PTE_FLAGS(*pte);
```

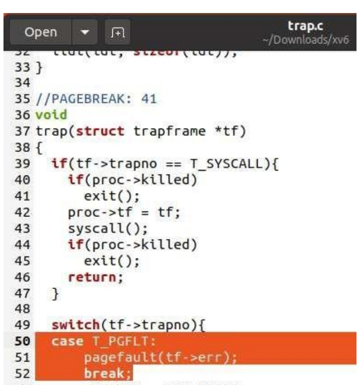
```
// No need of page allocation
if(mappages(d, (void*)i, PGSIZE, pa, flags) < 0) // map the
child's page table to same permanent addresses
goto bad;
acquire(&lock);
pg_refcount[pa >> PGSHIFT] = pg_refcount[pa >> PGSHIFT] + 1; //
increase reference count of that permanent page.
release(&lock);
}
lcr3(v2p(pgdir)); // reinstall the page table
return d;
bad:
freevm(d);
lcr3(v2p(pgdir)); // reinstall the page table
return 0;
}
```

Adding trap handler to handle pagefaults:

In trap.c:

Once you have changed the fork implementation as described above, both parent and child will execute over the same read-only memory image. Now, when the parent or child processes attempt to write to a page marked read-only, a page fault occurs. The trap handling code in xv6 does not currently handle the T_PGFLT exception (that is defined already, but not caught). You must write a trap handler to handle page faults in trap.c. You can simply print an error message initially, but eventually this trap handling code must call the function that makes a copy of user memory.

```
case T_PGFLT:
pagefault(tf->err);
break;
```




```
}

//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_PGFLT:
      pagefault(tf->err);
      break;
```

In vm.c:

```
void pagefault(uint err_code)
{
cprintf("Pagefault occured");
return;
}
```

Adding trap handling function to make copy of user memory:

In vm.c:

The bulk of your changes will be in this new function you will write to handle page faults. When a page fault occurs, the CR2 register holds the faulting virtual address, which you can get using the xv6 function call rcr2(). You must now look at this virtual address and decide what must be done about it. If this address is in an illegal range of virtual addresses that are not mapped in the page table of the process, you must print an error message and kill the process. Otherwise, if this trap was generated due to the CoW pages that were marked as read-only, you must proceed to make copies of the pages as needed.

Note that between the parent and the child, the first one that tries to write to a page should get a new memory page allocated to it. This new page's content must be copied from the contents of the original page pointed to by the virtual address. Even after this copy is made, note that the page is still marked as read only in the page table of the second process, and it will soon trap as well when it attempts to write to the read-only page. When the second process traps, no new pages need to be allocated; it suffices to remove the read-only restriction on the trapping page, since the first process already has its copy. Your page fault handling code should distinguish between these two cases using the reference count variable, and handle them suitably. Make sure you modify the reference counts correctly, and remember to flush the TLB whenever you change page table entries.

```
void pagefault(uint err_code)
{
uint va = rcr2();
uint pa;
pte_t *pte;
char *mem;
if(va >= KERNBASE)
{
cprintf("pid %d %s: Illegal memory access on CPU %d due
to virtual address 0x%x is mapped to kernel code. So killing
the process\n", proc->pid, proc->name, cpu->id, va);
proc->killed = 1;
return;
}
if((pte = walkpgdir(proc->pgdir, (void*)va, 0))==0)
{
cprintf("pid %d %s: Illegal memory access on CPU %d due
to virtual address 0x%x is mapped to NULL pte. So killing the
process\n", proc->pid, proc->name, cpu->id, va);
proc->killed = 1;
return;
5
}
if(!(*pte & PTE_P))
{
cprintf("pid %d %s: Illegal memory access on CPU %d due
to virtual address 0x%x is mapped to pte which is not
present.
So killing the process\n", proc->pid, proc->name, cpu->id,

va);
proc->killed = 1;
return;
}
if(!(*pte & PTE_U))
{
cprintf("pid %d %s: Illegal memory access on CPU %d due
to virtual address 0x%x is mapped to pte which is not
accessible to user. So killing the process\n", proc->pid,
proc->name, cpu->id, va);
proc->killed = 1;
```

```
return;
}
if(*pte & PTE_W)
{
panic("Unknown page fault due to a writable pte");
}
else
{
pa = PTE_ADDR(*pte);
acquire(&lock);
if(pg_refcount[pa >> PGSHIFT] == 1)
{
release(&lock);
*pte |= PTE_W;
}
else
{
if(pg_refcount[pa >> PGSHIFT] > 1)
{
release(&lock);
if((mem = kalloc()) == 0)
{
cprintf("pid %d %s: Pagefault due to out of
memory", proc->pid, proc->name);
proc->killed = 1;
return;
}
memmove(mem, (char*)p2v(pa), PGSIZE);
acquire(&lock);
```

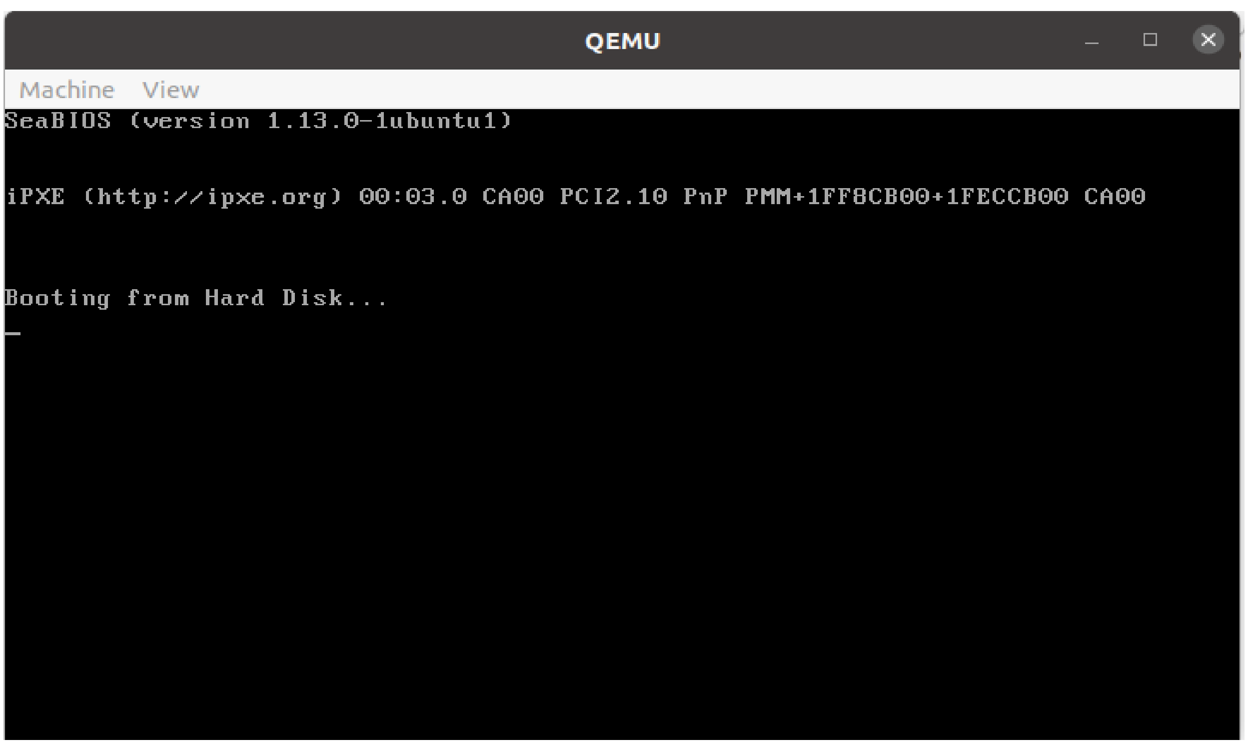
```
pg_refcount[pa >> PGSHIFT] = pg_refcount[pa >> PGSHIFT]
- 1;
pg_refcount[v2p(mem) >> PGSHIFT] = pg_refcount[v2p(mem) >>
PGSHIFT] + 1;
release(&lock);
*pte = v2p(mem) | PTE_P | PTE_W | PTE_U;
}
else
{
release(&lock);
panic("Pagefault due to wrong ref count");
}
}
lcr3(v2p(proc->pgdir));
}
}
```

<u>Test case testcow</u>:

testcow.c
~/Downloads/xv6

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

int i = 3;
int main(void)
{
        int pid;
        pid = fork();
        if(pid == 0)
        {
                printf(1,"Number of free pages in child 1 before changing variable are: %d\n", getNumFreePages());
                i = 4;
                printf(1,"Number of free pages in child 1 after changing variable due to copy are: %d\n", getNumFreePages());
        }
        else
        {
                wait();
                pid = fork();
                if(pid == 0)
                {
                        printf(1,"Number of free pages in child 2 before changing variable are: %d\n", getNumFreePages());
                        i = 4;
                        printf(1,"Number of free pages in child 2 after changing variable due to copy are: %d\n", getNumFreePages());
                }
                else
                {
                        printf(1,"Number of free pages in parent are: %d\n", getNumFreePages());
                        wait();
                }
        }
        exit();
}
```


ankith@ankith-VirtualBox: ~/xv6
ankith@ankith-VirtualBox:~$ cd xv6
ankith@ankith-VirtualBox:~/xv6$ make qemu
qemu-system-x86_64 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512


QEMU
Machine View
SeaBIOS (version 1.13.0-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00
Booting from Hard Disk...

Result:

Output of above test case is :

$ testcow

Number of free pages in child 1 before changing variable are:

56710 Number of free pages in child 1 after changing variable due

to copy are: 56709

Number of free pages in parent are: 56710

Number of free pages in child 2 before changing variable are:

56710 Number of free pages in child 2 after changing variable due

to copy are: 56709

References :

- https://stackoverflow.com/questions/21653195/xv6-add-a-system-call-that-counts-system-calls
- https://www.youtube.com/watch?v=21SVYiKhcwM 2. https://medium.com/@silvamatteus/adding-new-system-calls-to-xv6-217b7daefbe1
- https://medium.com/@viduniwickramarachchi/add-a-new-system-call-in-xv6- 5486c2437573.
- https://stackoverflow.com/questions/8021774/how-do-i-add-a-system-call-utility-in-xv6
- https://stackoverflow.com/questions/21653195/xv6-add-a-system-call-that-countssystem-calls
- https://arjunkrishnababu96.gitlab.io/post/xv6-system-call/
- https://github.com/sayak119/xv6_scheduler
- https://www.cse.iitb.ac.in/~mythili/teaching/cs347_autumn2016/ index.html
- https://www.reddit.com/r/osdev/comments/32dtz0/ copy_on_write_fork_are_the_page_tables_hierarchy/

- https://medium.com/@viduniwickramarachchi/add-a-new system-call-in-xv6-5486c243757