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**



**Drive Link for review 3 :** [https://drive.google.com/drive/folders/1Hk7VXN5NN9ewaSKb5gpTfVBiCUHzXj- o?](https://drive.google.com/drive/folders/1Hk7VXN5NN9ewaSKb5gpTfVBiCUHzXj-o?usp=sharing) [usp=sharing](https://drive.google.com/drive/folders/1Hk7VXN5NN9ewaSKb5gpTfVBiCUHzXj-o?usp=sharing)

# 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.

1. M Barabanov - 1997 - yodaiken.com- [A linux-based real-time operating system](http://www.yodaiken.com/papers/BarabanovThesis.pdf)

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.

1. R Cox, [MF Kaashoek](https://scholar.google.com/citations?user=YCoLskoAAAAJ&hl=en&oi=sra), [R Morris](https://scholar.google.com/citations?user=in6eBIwAAAAJ&hl=en&oi=sra)- 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.

1. 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.
2. Copy on write file system consistency and block usage-[David Hitz](https://patents.google.com/?inventor=David%2BHitz)[Michael](https://patents.google.com/?inventor=Michael%2BMalcolm) [Malcolm](https://patents.google.com/?inventor=Michael%2BMalcolm)[James Lau](https://patents.google.com/patent/US6892211B2/en)[Byron Rakitzis](https://patents.google.com/?inventor=Byron%2BRakitzis)

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.

1. Systems and methods for adaptive copy on write- [Darren P. Schack](https://patents.google.com/?inventor=Darren%2BP.%2BSchack)[Eric M.](https://patents.google.com/?inventor=Eric%2BM.%2BLemar) [Lemar](https://patents.google.com/?inventor=Eric%2BM.%2BLemar)[Neal T. Fachan](https://patents.google.com/?inventor=Neal%2BT.%2BFachan)

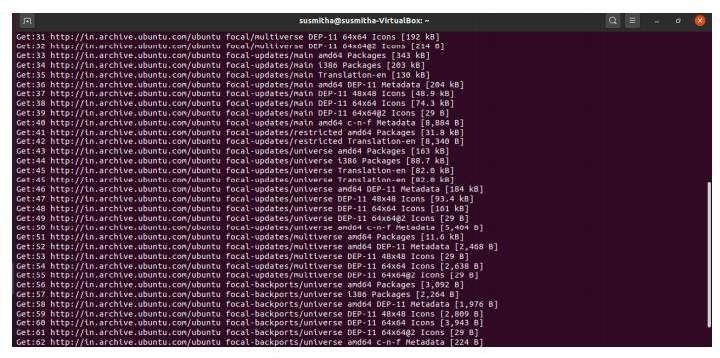
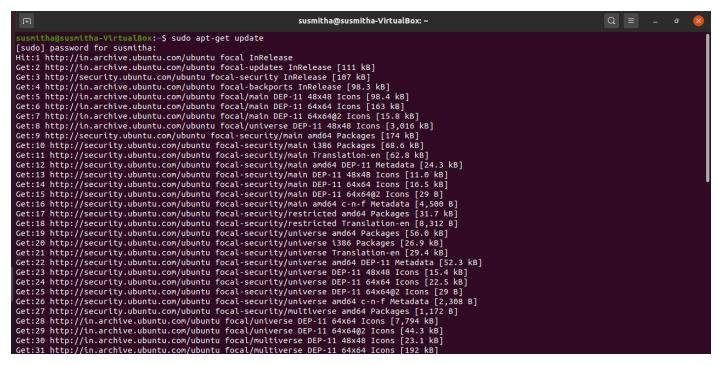
This paper helped us dive deep into the concept of copy on write. It introduced to us the idea of poin-in-time-copy thats another way of recording file system state and taking the snapshot. It shows how various physically distributed systems can efficiently use copy on write function.

# 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

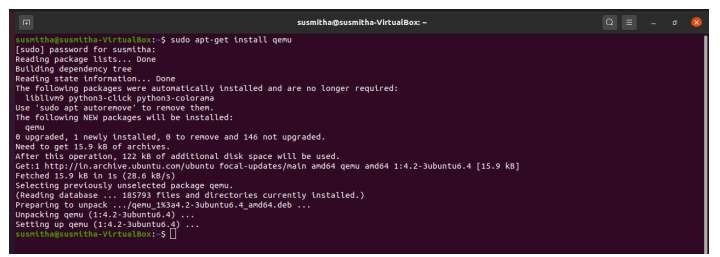
### SCREENSHOT:



**QEMU INSTALLATION**: **COMMAND**:

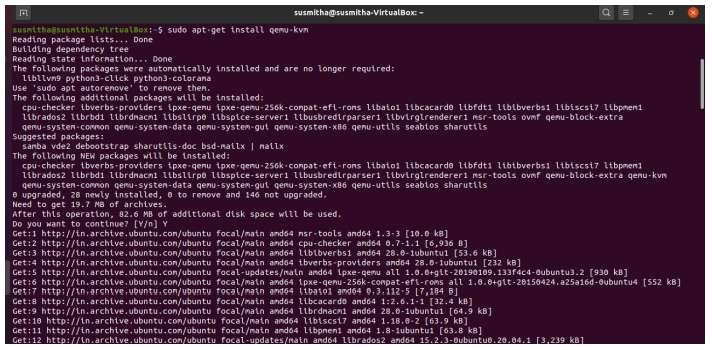
Sudo apt-get install qemu

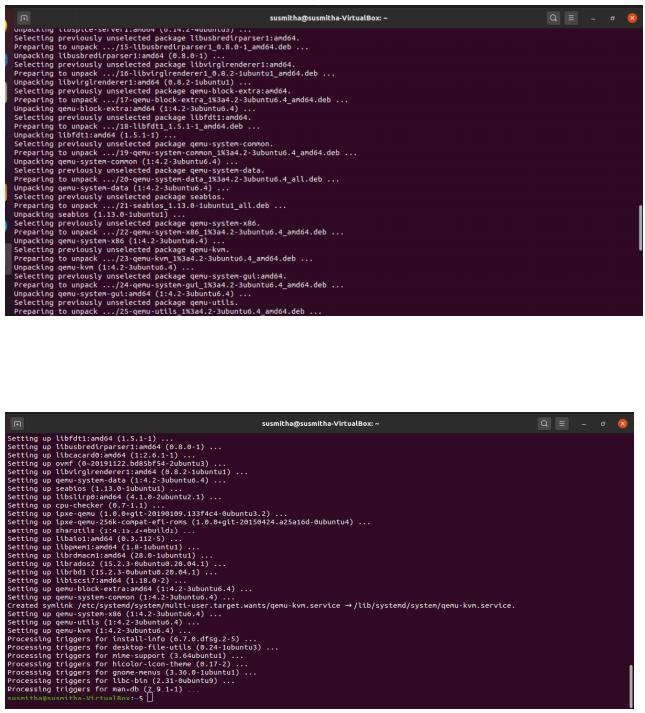
### SCREENSHOT:



**COMMAND**:

sudo apt get-install qemu-kvm

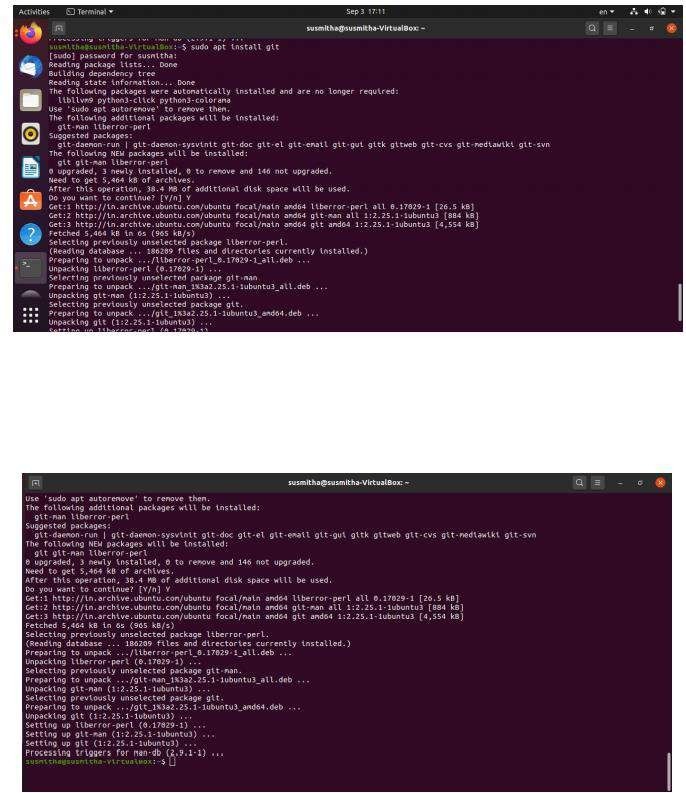




### INSTALLING GIT REPOSITORY:

Git repository is installed to clone XV6 from github.

**COMMAND:** sudo apt install git

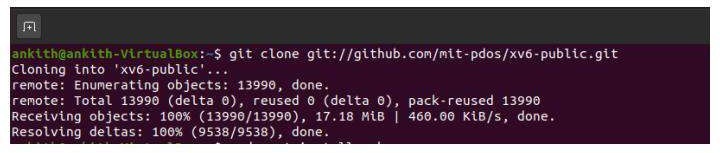


### CLONING XV6 FROM GITHUB:

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

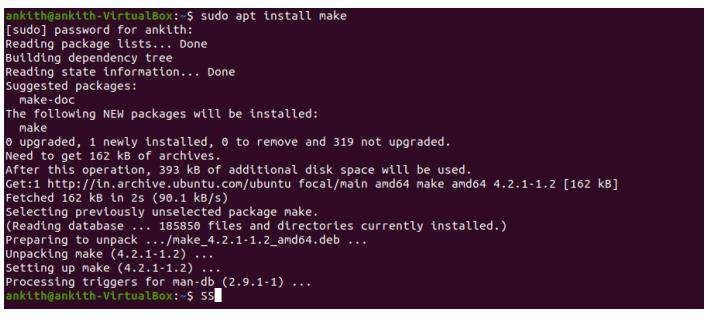
### COMMAND:

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



### INSTALLING MAKE REPOSITORY:

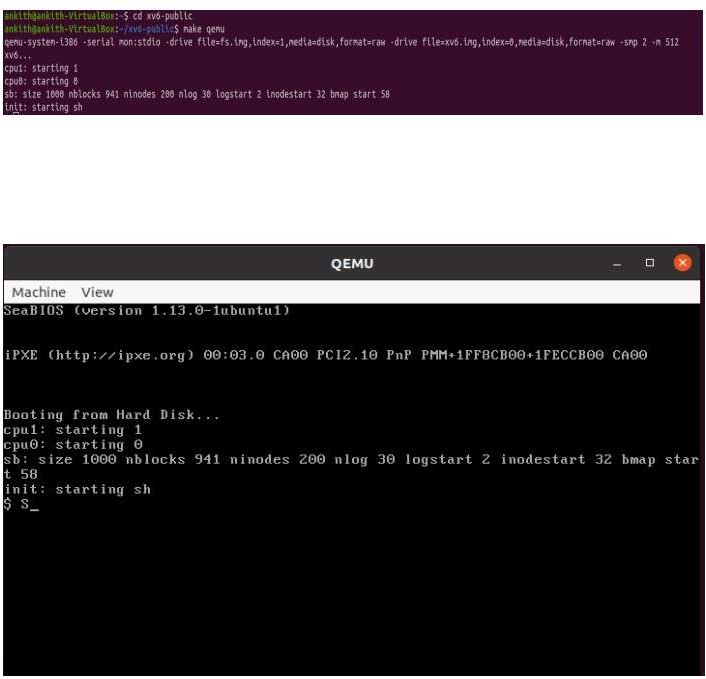
**COMMAND**: Sudo apt install make



### RUNNING XV6:

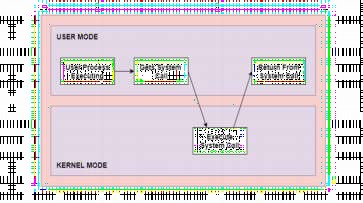
**COMMAND:**

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();

}



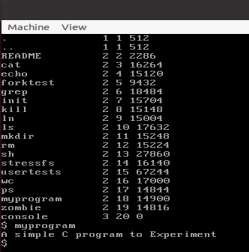
# Makefile.c:

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





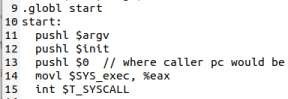
* Now, start xv6 system on QEMU and when it booted up, run ls command to check whether our program is available for the user.
* Here myprogram is availablein the list and by giving the name we can see the output of the Program In image below.



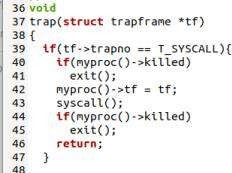
# Adding New System Calls To xv6:

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.

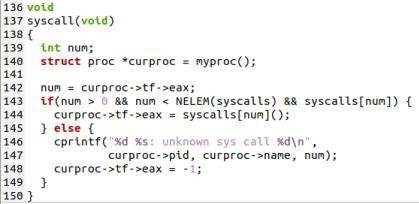
An application signals the kernel it needs a service by issuing a software interrupt, a signal generated to notify the processor that it needs to stop its current task, and response to the signal request. Before switching to handling the new task, the processor has to save the current state, so that 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*)



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*.



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:

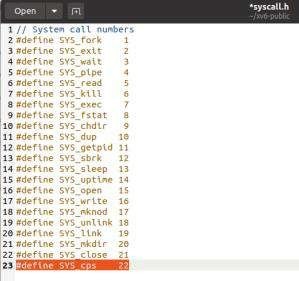


# The following are the changes to be done to add our system call cps () to xv6:

* 1. **Add name to *syscall.h* :**

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

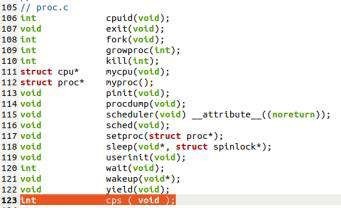
**CODE:** #define sys\_cps 22



* 1. **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);



* 1. **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);



* 1. **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().

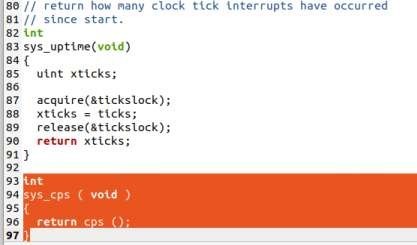
### CODE:

Int sys\_cps(void)

{

Return cps();

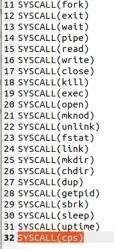
}



* 1. **Add call to *usys.S* :**

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

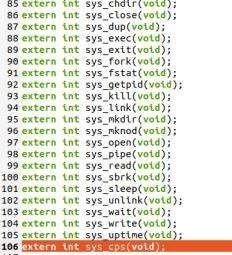
**CODE:** SYSCALL(cps)



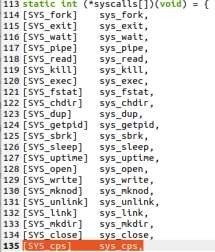
* 1. **Add call to *syscall.c* :**

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);



**CODE:** [SYS\_cps] sys\_cps,



* 1. **Add code to *proc.c* :**

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)

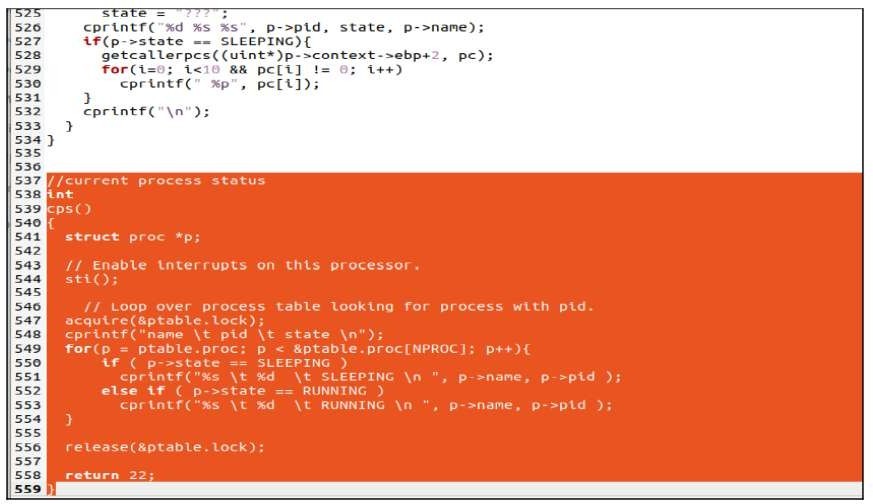
cprintf("%s \t %d \t SLEEPING \n", p->name, p->pid); // printing pid,pname,state else if(p->state == RUNNING)

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

}

release(&ptable.lock); // releasing acquired lock return 22;

}



### 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[ ])

{

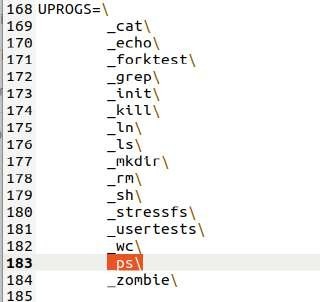
Cps(); // calling cps function exit();

}



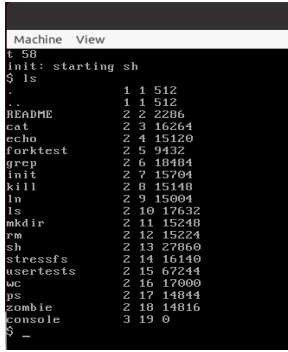
* 1. ***Modify* Makefile *:***

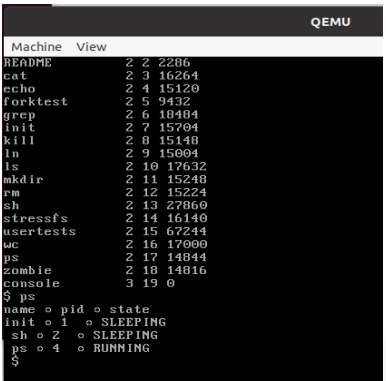
Then we make which compiles all changes we made inside the xv6 directories and subdirectories.



# 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.

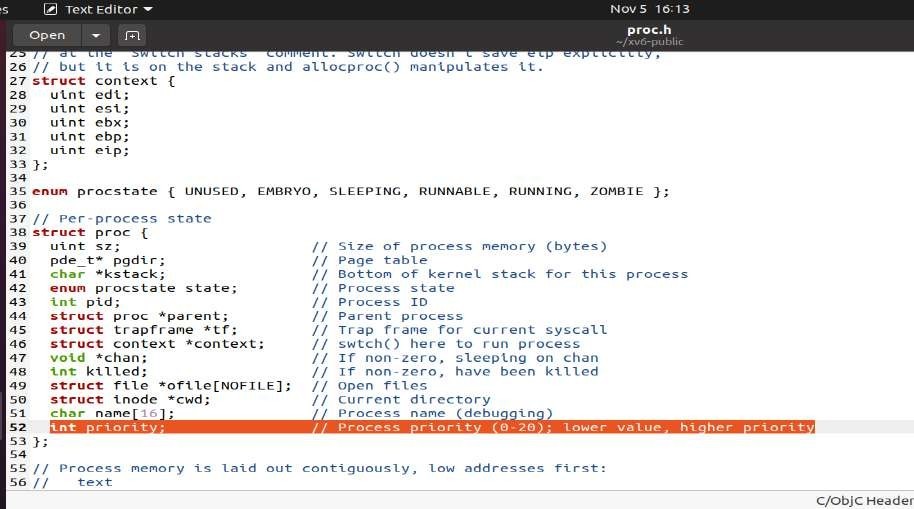
Processes with the same priority are executed on first come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

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.

### 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

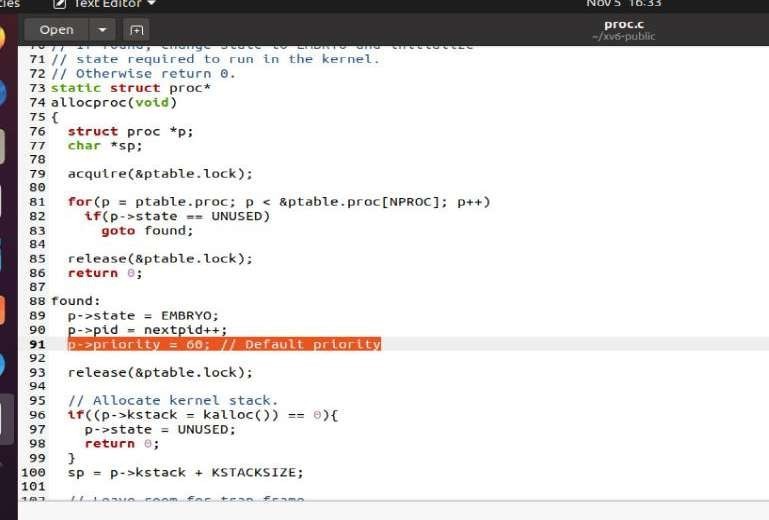


### Assign a default priority in proc.h:

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.

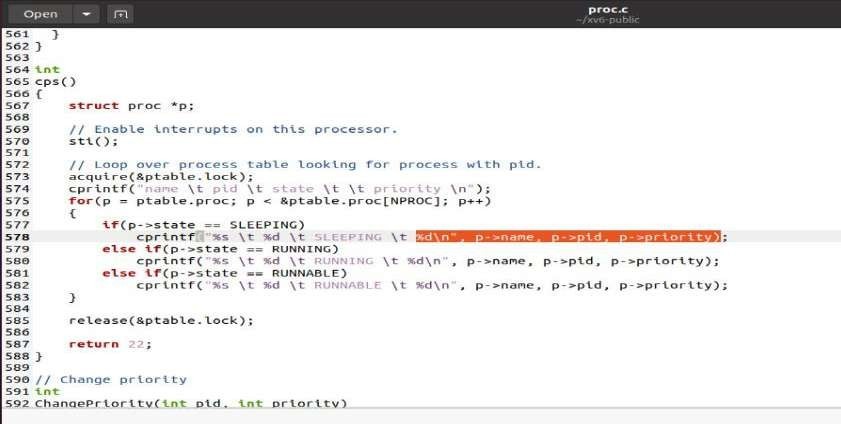
### CODE:

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



### Adding code to print priority of process in cps in proc.c:

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.



### 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

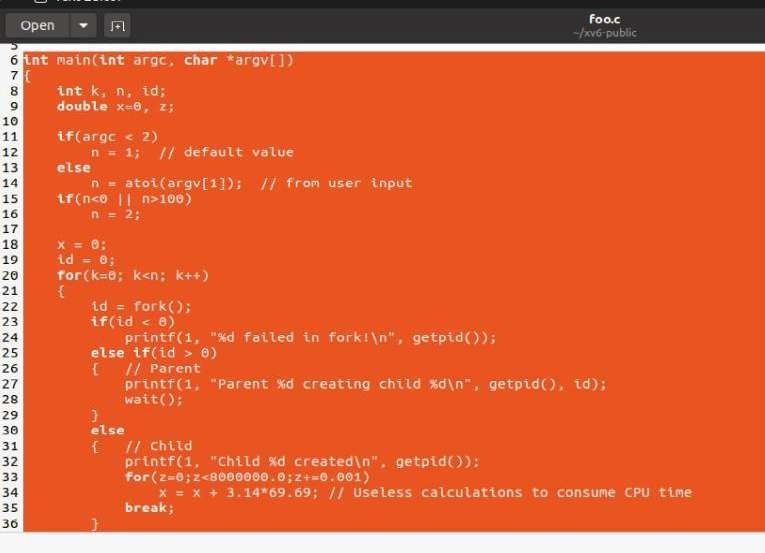
{ // Child

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;

}

}



### 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

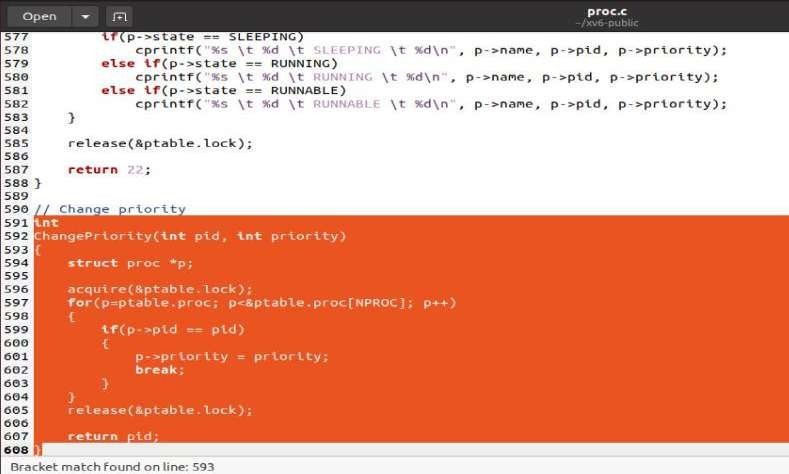
{

p->priority = priority; //changing priority of process break;

}

}

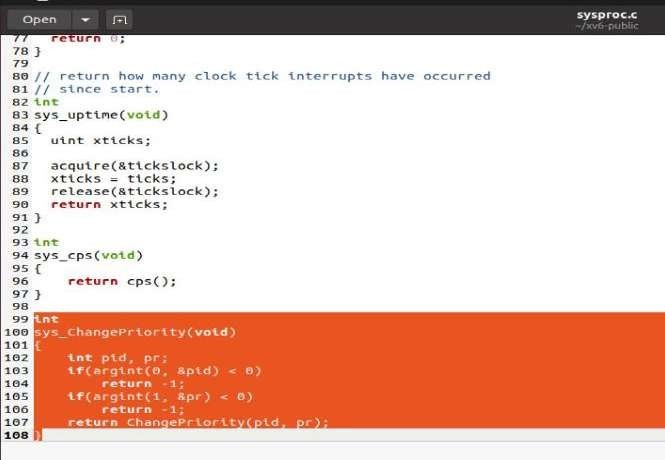
release(&ptable.lock); return pid; }



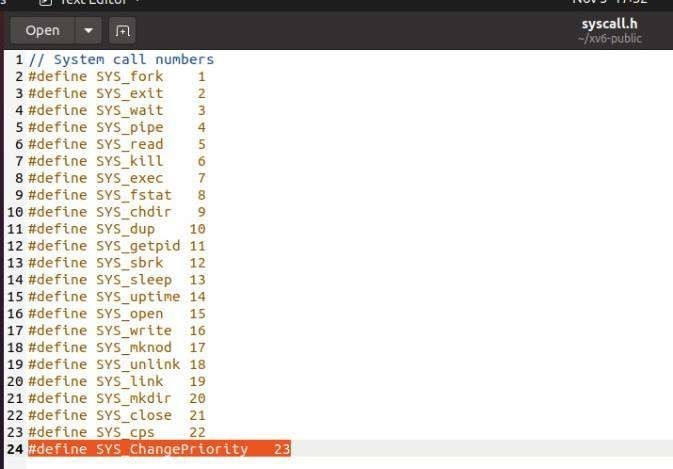
### Adding system call(chpr):

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

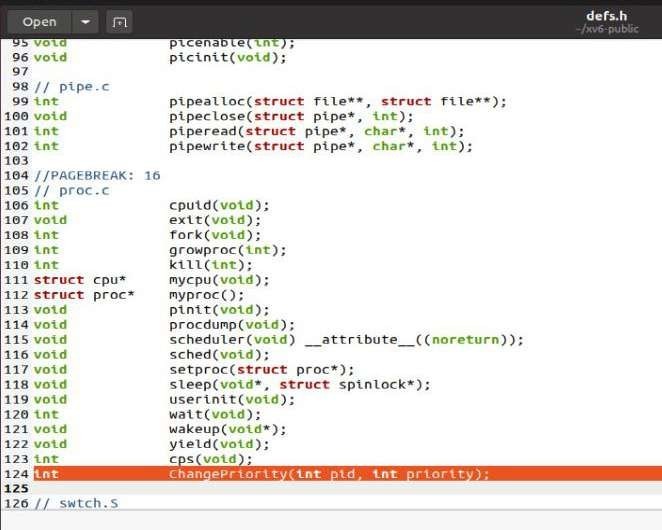
### Add sys\_chpr to syspoc.c:



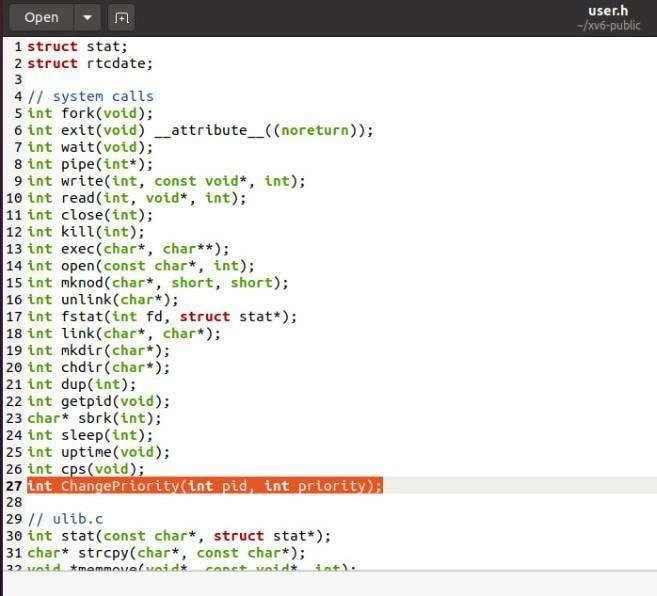
* + **Adding to syscall.h:**



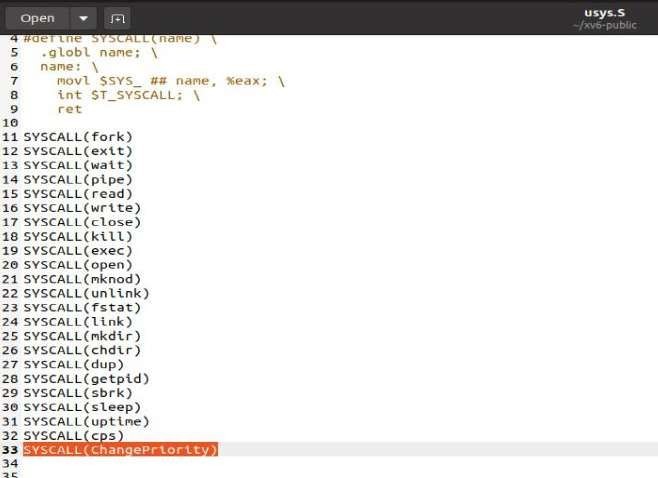
### Modify defs.h:



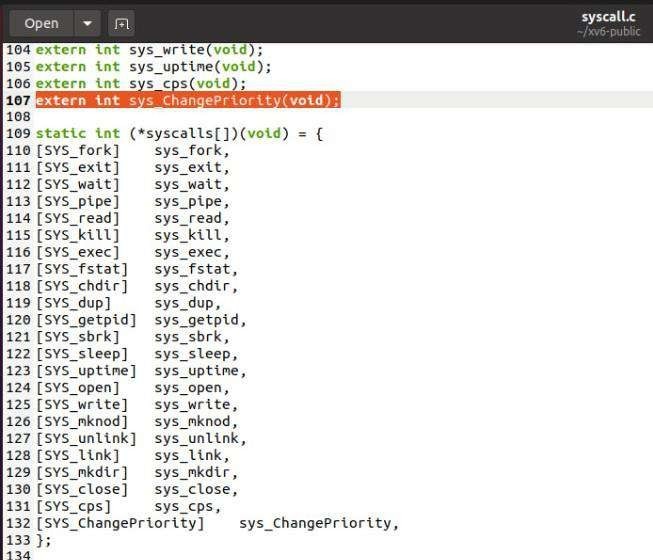
* **Modify user.h:**

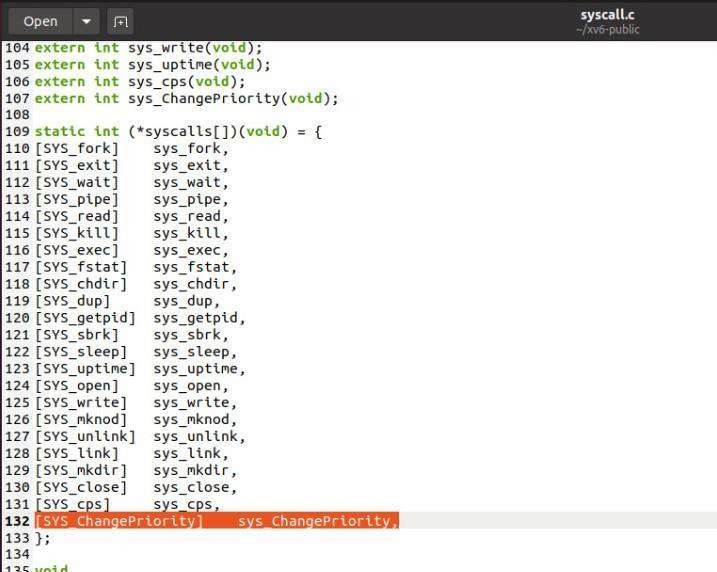


### Modify usys.s:



**Modify syscall.c:**





### 6. Adding nice.c program:

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

### CODE:

#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)

{

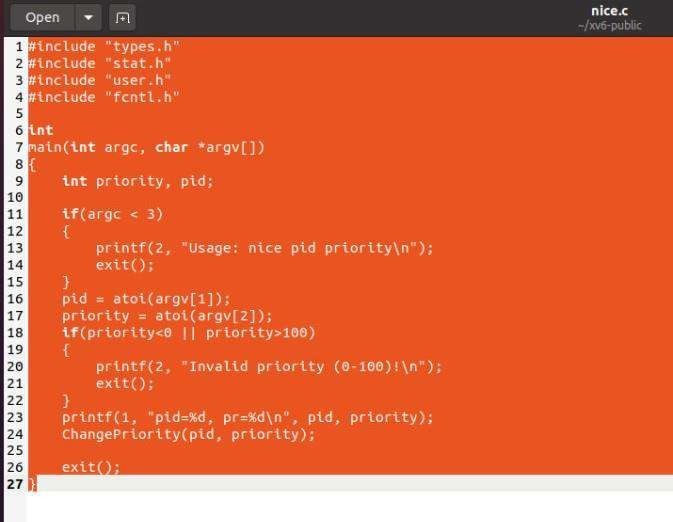
printf(2, "Invalid priority (0-100)!\n"); exit();

}

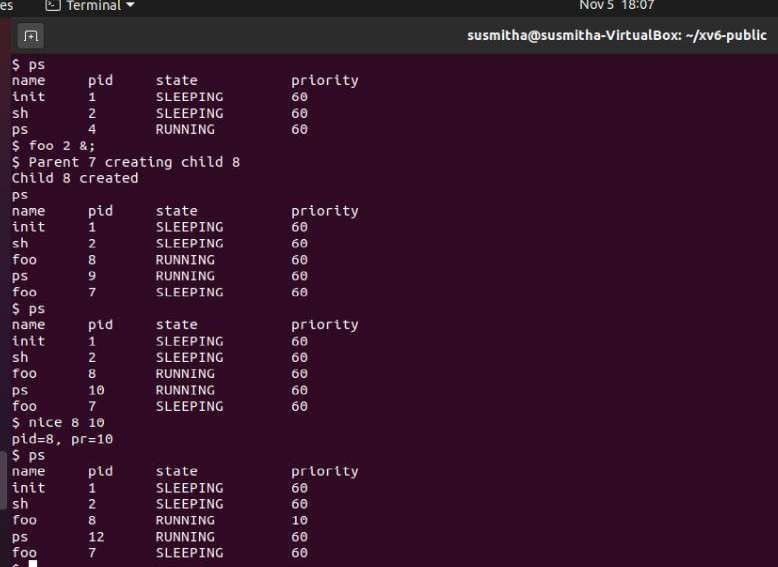
printf(1, "pid=%d, pr=%d\n", pid, priority); ChangePriority(pid, priority);

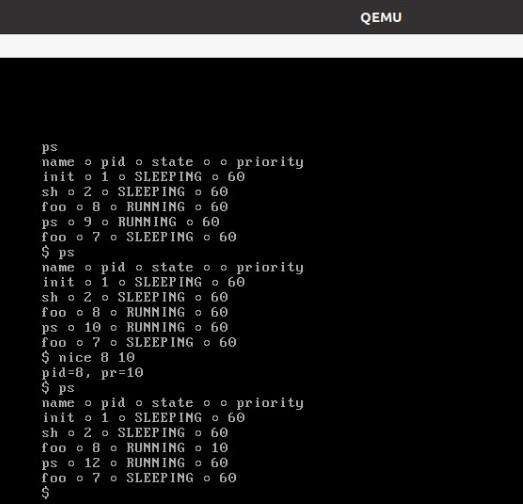
exit();

}



### OUTPUT:





**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.

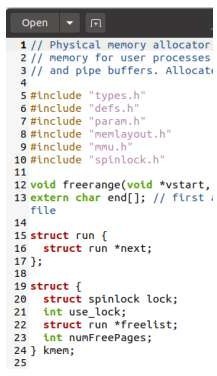
### Implementation of Copy-on-write System Call: Adding getNumFreePages system call:

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

In kalloc.c add the entry in struct kmem:

### int numFreePages;

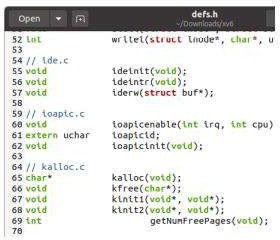
The system call getNumFreePages() should return the total number of free pages in the system. This system call will help you see when pages are consumed, and can help you debug your CoW implementation. You must add code to maintain and track freepages in kalloc.c, and access this information when this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



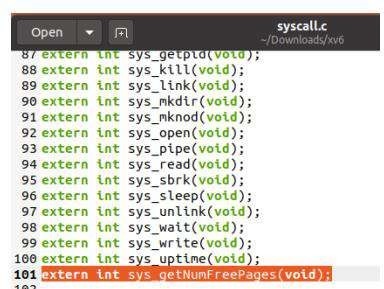
**Adding given system calls**: add following code in defs.h int

**getNumFreePages(void);**

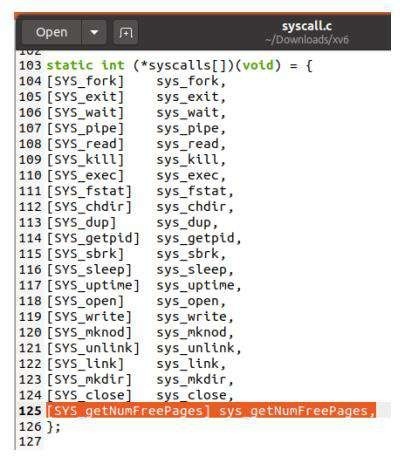
The file defs.h acts as the header file for several parts of the kernel code.



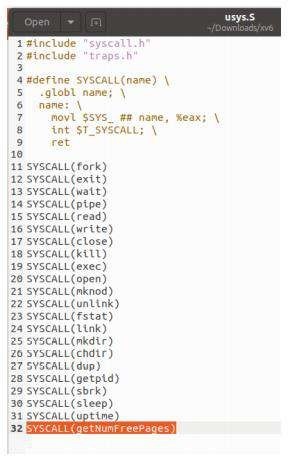
### Add following code in sysproc.c:

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

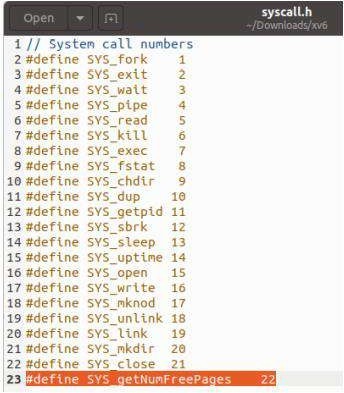
### Add following fields in the same file like other system calls in syscall.c:



**Add the following lines in usys.S:**

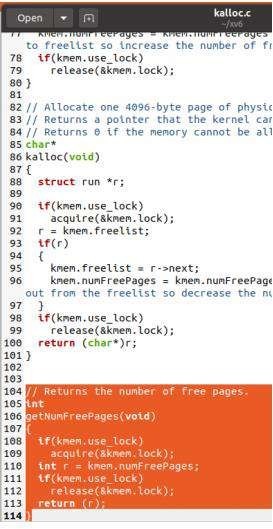


### Add following lines in syscall.h:



**Add the function body of getNumFreePages in kalloc.c:**

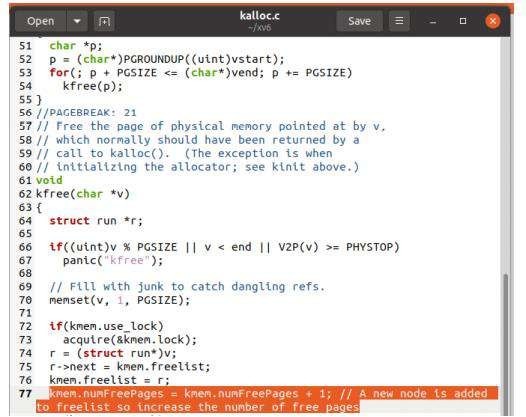
## The files vm.c and kalloc.c contain most of the logic for memory management in the xv6 kernel



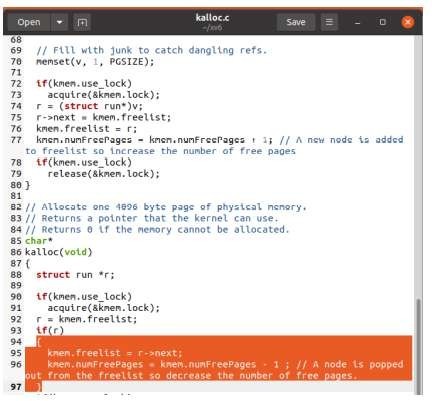
### Initialize numFreePages in kinit1:



**In kfree:**



### In kalloc:



**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

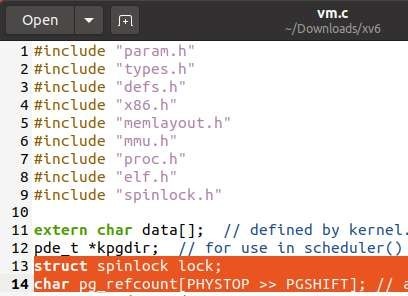
### Keeping track of the reference count of pages:

**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

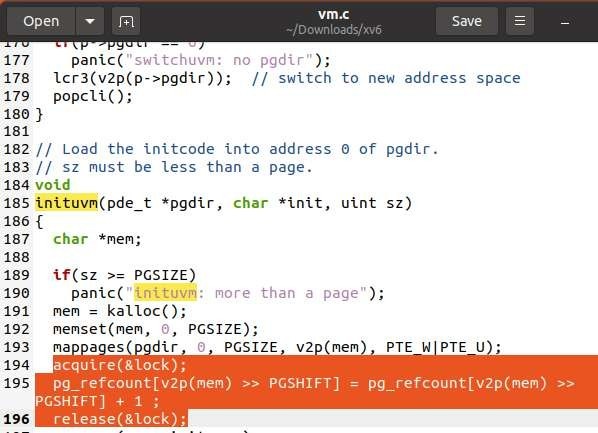


### In inituvm function:

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);

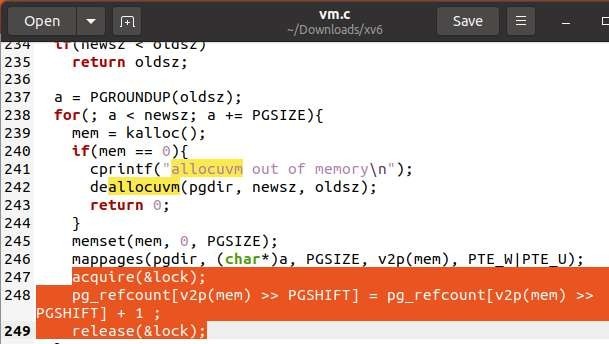
pg\_refcount[v2p(mem) >> PGSHIFT] = pg\_refcount[v2p(mem) >> PGSHIFT] +1 ; release(&lock);



### In allocuvm function:

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);

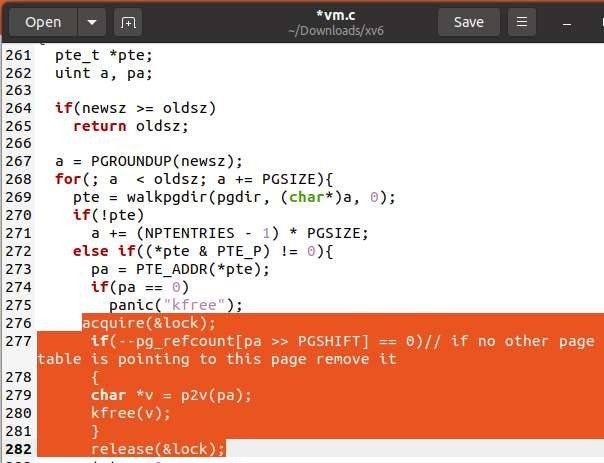
if(--pg\_refcount[pa >> PGSHIFT] == 0)// if no other page table is pointing to this page remove it

{

char \*v = p2v(pa); kfree(v);

}

release(&lock);

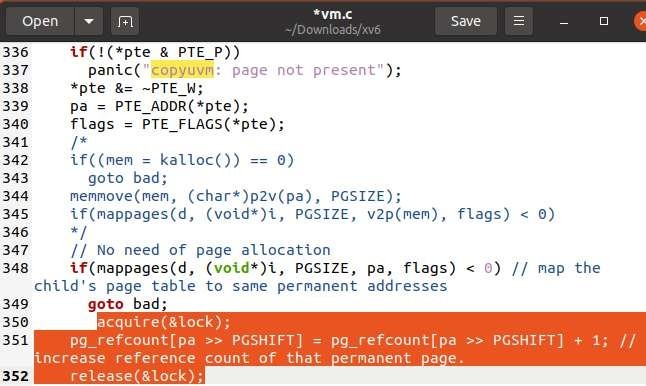


### In copyuvm function when a process is forked the refcount of that permanent address should be incremented:

acquire(&lock);

pg\_refcount[pa >> PGSHIFT] = pg\_refcount[pa >> PGSHIFT] + 1; // increase reference count of that permanent page.

release(&lock);



### Change of copyuvm function:

**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;



### 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");

}

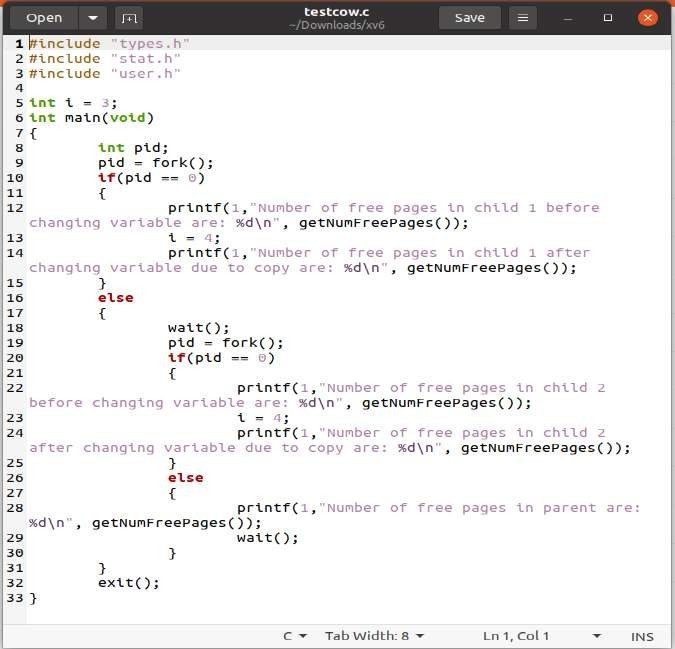
}

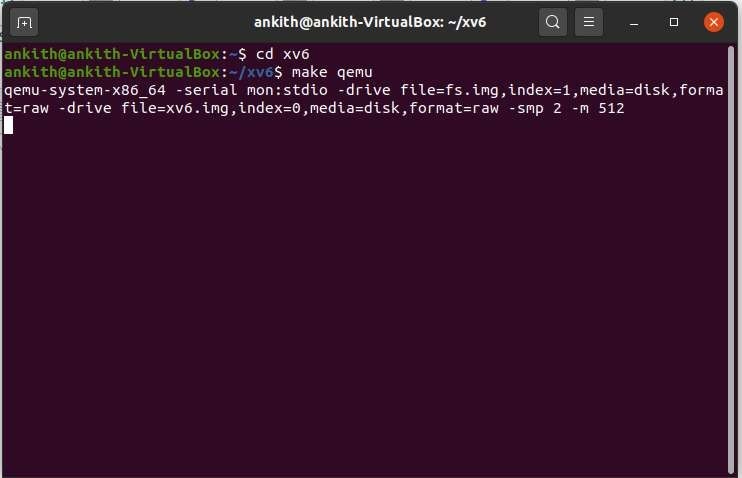
lcr3(v2p(proc->pgdir));

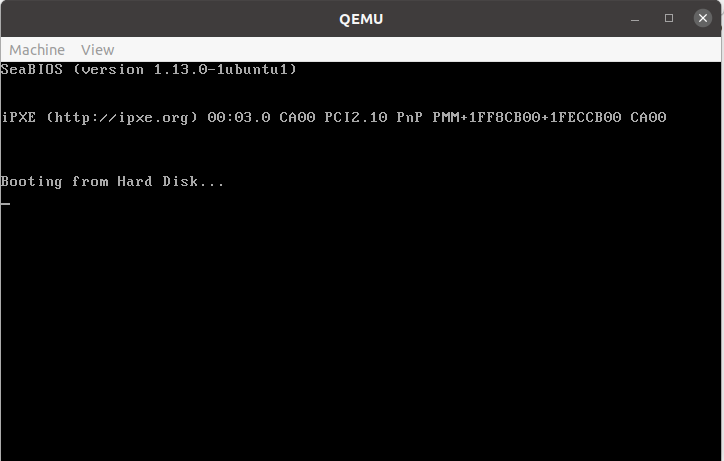
}

}

### Test case testcow:







**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-](https://stackoverflow.com/questions/21653195/xv6-add-a-system-call-that-counts-system-calls) [that-counts-system-calls](https://stackoverflow.com/questions/21653195/xv6-add-a-system-call-that-counts-system-calls)
* https://[www.youtube.com/watch?v=21SVYiKhcwM 2.](http://www.youtube.com/watch?v=21SVYiKhcwM2) [https://medium.com/@silvamatteus/adding-new-system-calls-to-xv6-](https://medium.com/%40silvamatteus/adding-new-system-calls-to-xv6-) 217b7daefbe1
* [https://medium.com/@viduniwickramarachchi/add-a-new-system-call-](https://medium.com/%40viduniwickramarachchi/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/](http://www.cse.iitb.ac.in/~mythili/teaching/cs347_autumn2016/) index.html
* https://[www.reddit.com/r/osdev/comments/32dtz0/](http://www.reddit.com/r/osdev/comments/32dtz0/) copy\_on\_write\_fork\_are\_the\_page\_tables\_hierarchy/
* [**https://medium.com/@viduniwickramarachchi/add-a-new system-**](https://medium.com/%40viduniwickramarachchi/add-a-newsystem-) **call-in-xv6-5486c243757**