# Documentation for Customised system calls in xv6

## Contents

# 1 Introduction

This document outlines a detailed, step-by-step process for adding custom system calls to the **xv6-public** operating system. It begins by explaining the procedure for integrating a new system call into **xv6-public** and then showcases the implementation and functionality of the system calls we developed within this environment.

# 2 Adding a System Call

## 2.1 Defining the Function in `proc.c`

To implement the new system call, we first define the corresponding function in the file `proc.c`. In this example, we created the system call `cps()`, which lists all the current processes in the system. The code is shown below:

`proc.c`

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

    release(&ptable.lock);

    return 22;
}
```

## 2.2 Declaring the Function in `user.h` and `defs.h`

Next, the function must be declared in `user.h` and `defs.h`. Below is the declaration for the `cps` function:

`defs.h`

```
...
int cps(void);
...
```

`user.h`

```
...
int cps(int);
...
```

## 2.3 Defining the System Call Function

We then define the system call function in `sysproc.c`, which acts as a wrapper for the function implemented in `proc.c`. For the `cps` system call, we define a function named `sys_cps` that invokes the `cps` function:

`sysproc.c`

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

## 2.4 Declaring the System Call in `syscall.c`

Next, we declare the system call in the `syscalls` array within `syscall.c`. However, before doing so, we need to inform the compiler that the `sys_cps` function is defined in another file. Therefore, we declare it as an `extern` function:

`syscall.c`

```
...
extern int sys_cps(void);
...
```

Afterward, we add the new system call `SYS_cps` to the `syscalls` array as follows:

```
static int (*syscalls[])(void) = {
    ...
    [SYS_cps] sys_cps,
    ...
};
```

## 2.5 Assigning a Unique Identifier in `syscall.h`

Next, we assign a unique identifier (number) to the new system call in the `syscall.h` file, as shown below:

`syscall.h`

```
...
#define SYS_cps 22
...
```

This step ensures that the system call has a unique identifier within the operating system.

## 2.6 Adding a New User Program for Testing

To test the newly implemented system call, we add a user program to the system that invokes the system call to verify its correctness. For testing the `cps` system call, we create a simple program named `test_cps.c`:

`test_cps.c`

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

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

    // Start some background processes
    for (int i = 0; i < 3; i++) { // Create 3 background processes
        pid = fork();
        if (pid < 0) {
            printf(1, "Fork failed\n");
            exit();
        } else if (pid == 0) {
            // Child process
            // printf(1, "Child process %d started\n", getpid());
            for (volatile int j = 0; j < 1e8; j++); // Simulate work
            // printf(1, "Child process %d exiting\n", getpid());
            exit(); // End child process
        }
        // Parent continues to create more processes
    }
    cps();
    exit();
}
```

Once the user program is added, we need to include this file in the **EXTRAS** section of the **Makefile**, as shown below:

`Makefile`

```
...
EXTRA=\
        ...
    test_cps.c\
    ...
...
```

Finally, we add the corresponding executable to the **UPROGS** section of the **Makefile**:

Makefile

```
...
UPROGS=\
    ...
    _test_cps\
    ...
...
```

By following these steps, we ensure that the new system call is properly tested in the system environment.

## 2.7 Outputs

The following screenshots illustrate the functionality and correctness of the implemented system calls:

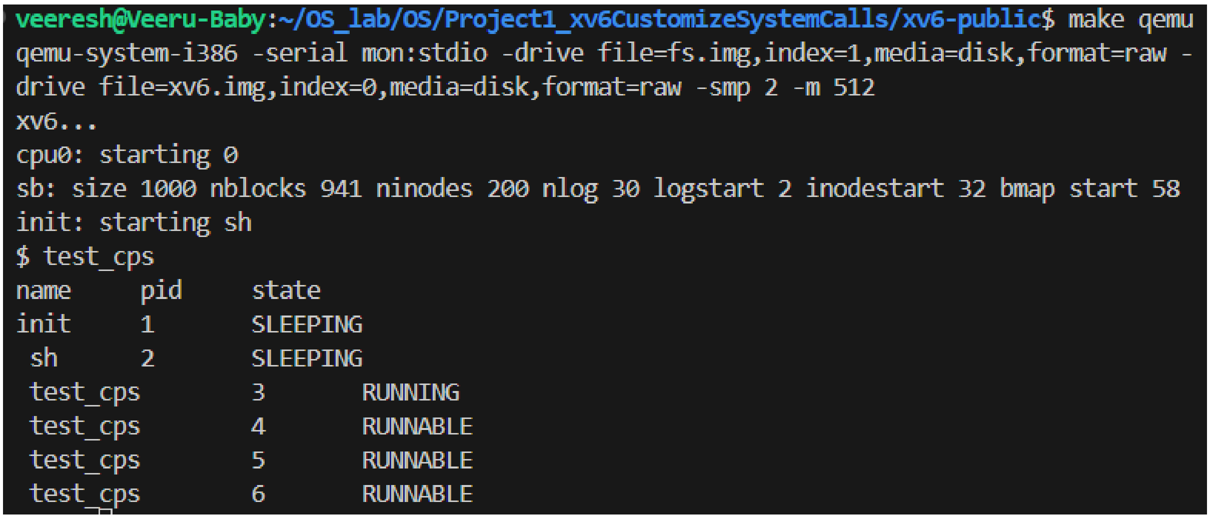



Figure 1: test results of cps

# 3 System Calls Added

We introduced the following custom system calls in the system environment:

1. **mem_usage():** Displays memory usage for processes.
2. **get_process_type():** Returns the type of the process whether it is a init, orphan, zombie or a normal process.
3. **getppid():** Returns the process id(pid) of the parent process.
4. **get_priority():** Returns the priority of the process.
5. **set_priority():** Allows setting process's priority.
6. **Semaphores:** Implemented semaphores for **process synchronization**, supporting **creation**, **initialization**, and **deallocation**, similar to those defined in `semaphore.h`.
7. **Priority Scheduler:** Enables scheduling of the processes based on the priority of the process.

## 3.1 Memory Usage for Each Process

### 3.1.1 Function Definition in `proc.c`

```
int mem_usage(void) {
    struct proc *p;
    int pid, size = 0, found = 0;

    // Get PID from argument
    if (argint(0, &pid) < 0) return -1;

    // Synchronization with lock
    acquire(&ptable.lock);

    // Locate process by PID
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            size = p->sz;
            found = 1;
            break;
        }
    }

    release(&ptable.lock);
```

```
    return found ? size : -1;
}
```

### 3.1.2 Declarations in user.h and defs.h

user.h:

```
...
int mem_usage(int);
...
```

defs.h:

```
...
int mem_usage(void);
...
```

### 3.1.3 System Call Implementation in sysproc.c

```
int sys_mem_usage(void) {
    return mem_usage();
}
```

### 3.1.4 Adding System Call in syscall.c

Declare the function as extern:

```
extern int sys_mem_usage(void);
```

Add the system call to the syscalls array:

```
static int (*syscalls[])(void) = {
    ...
    [SYS_mem_usage] sys_mem_usage,
    ...
};
```

### 3.1.5 Adding Identifier in syscall.h

Assign a unique number to the system call:

```
#define SYS_mem_usage 27
```

### 3.1.6 Adding user program for testing

To test the newly implemented system call, we add a user program to the system that invokes the system call to verify its correctness. For testing the mem_usage system call, we created a simple program named memusage.c:

`memusage.c`

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

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

    if(argc != 2) {
        printf(2, "Usage: memusage pid\n");
        exit();
    }

    //Get the PID from the command line argument
    pid = atoi(argv[1]);


    if(pid < 0) {
        printf(2, "Invalid PID\n");
        exit();
    }

    //Get the memory info using mem_usage system call
    mem = mem_usage(pid);

    if(mem < 0){
      printf(2, "Process %d not found or memory not accessible\n", pid);
    }

    else{
        printf(1, "Memory usage of process %d: %d bytes\n", pid, mem);
    }

    exit();
}
```

Once the user program is added, we need to include this file in the **EXTRAS** section of the **Makefile** and add the corresponding executable to the **UPROGS** section.

Makefile

```
...
UPROGS=\
    ...
    _memusage\
    ...
...
EXTRA=\
        ...
    memusage.c\
    ...
...
```

### 3.1.7 Outputs

The following screenshots illustrate the functionality and correctness of the implemented system calls:

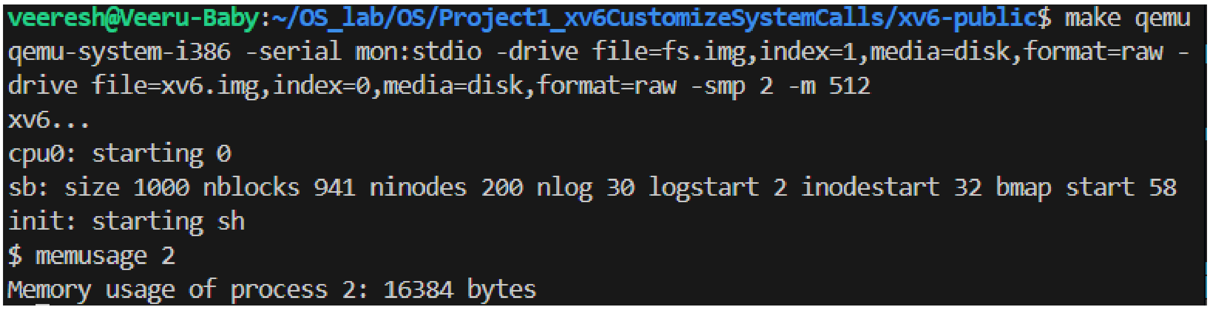



Figure 2: test results of memory usage

## 3.2 Finds the process type of the process

### 3.2.1 Function Definition in proc.c

```c
int get_process_type(void){
  int pid;
  struct proc *p;
  //Get the pid (given as argument) from argint.
  if(argint(0, &pid) < 0){
    return -1;
  }

  if(pid == 1){
    //init process
    return 3;
  }
  //Acquire the lock for synchronization.
  acquire(&ptable.lock);
  //Going throught the process table for finding the process with the
  //given pid.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      //Checking whether the process is a zombie process or not.
      if(p->state == ZOMBIE){
        //cprintf("Process type: %s\n", p->state);
        //release the lock before returning.
        release(&ptable.lock);
        return 1;
      }
      //Checking whether the process is orphan or not.
      else if(p->parent && p->parent->pid == 1 && p->state != ZOMBIE){
        //cprintf("Process type: %s\n", p->state);
        //As init processes adopt orphan processes.
        release(&ptable.lock);
        return 0;
      }else{
        //cprintf("Process type: %s\n", p->state);
        //Else it is a normal process.
        release(&ptable.lock);
        return 2;
      }
    }
  }
  //If no such process exists return -1
  release(&ptable.lock);
  return -1;
}
```

### 3.2.2 Declarations in `user.h` and `defs.h`

`user.h`:

```
...
int get_process_type(int);
...
```

`defs.h`:

```
...
int get_process_type(void);
...
```

### 3.2.3 System Call Implementation in `sysproc.c`

```
int sys_get_process_type(void){
  return get_process_type();
}
```

### 3.2.4 Adding System Call in `syscall.c`

Declare the function as `extern`:

```
extern int sys_get_process_type(void);
```

Add the system call to the `syscalls` array:

```
static int (*syscalls[])(void) = {
    ...
    [SYS_get_process_type] sys_get_process_type,
    ...
};
```

### 3.2.5 Adding Identifier in `syscall.h`

Assign a unique number to the system call:

```
#define SYS_get_process_type 24
```

### 3.2.6 Adding user program for testing

To test the newly implemented system call, we add a user program to the system that invokes the system call to verify its correctness. For testing the `get_process_type` system call, we created a simple program named `getprocesstype.c`:

getprocesstype.c

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

int main() {
    int zombie_pid, orphan_pid;
    // Create a zombie process
    zombie_pid = fork();
    if (zombie_pid == 0) {
        // Child process immediately exits
        printf(1, "Child (Zombie PID: %d) exiting\n", getpid());
        exit();
    } else if (zombie_pid > 0) {
        // Parent does NOT call wait() to retrieve the child's status
        sleep(50); // Allow time for the child to become a zombie
    }
    // Check if the child process is a zombie
    int type = get_process_type(zombie_pid);
    if (type == 1) {
        printf(1, "Process %d is a zombie\n", zombie_pid);
    } else {
        printf(1, "Process %d is NOT a zombie (Type: %d)\n",
                zombie_pid, type);
    }
    // Create an orphan process
    orphan_pid = fork();
    if (orphan_pid == 0) {
        // Child process will outlive its parent
        printf(1, "Child (Orphan PID: %d) created\n", getpid());
        sleep(100); // Simulate some work
        // Check if the child has become an orphan
        type = get_process_type(getpid());
        if (type == 0) {
            printf(1, "Child (PID: %d) is now an orphan\n", getpid());
        }
        exit();
    } else if (orphan_pid > 0) {
        // Parent process exits before the child
        printf(1, "Parent process (PID: %d) exiting, making child an
                orphan\n", getpid());
        exit();
    }
    return 0;
}
```

Once the user program is added, we need to include this file in the **EXTRAS** section of the **Makefile** and add the corresponding executable to the **UPROGS** section. `Makefile`

```
...
UPROGS=\
    ...
    _getprocesstype\
    ...
...
EXTRA=\
        ...
    getprocesstype.c\
    ...
...
```

### 3.2.7 Outputs

The following screenshots illustrate the functionality and correctness of the implemented system calls:

```
qemu-system-i386 -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
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ getprocesstype
Child (Zombie PID: 4) exiting
Process 4 is a zombie
Parent process (PID: 3) exiting, making child an orphan
 $ Child (Orphan PID: 5) created
Child (PID: 5) is now an orphan
```

Figure 3: test results of get process type

## 3.3 Finds the process id of parent process

### 3.3.1 Function Definition in `proc.c`

```
int getppid(void){
  int pid;
  struct proc *p;

  if(argint(0, &pid) < 0){
    return -1;
  }

  acquire(&ptable.lock);
  // Find process with the specified PID
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->pid == pid) {
          break;
      }
  }

  if(p >= &ptable.proc[NPROC]){
    release(&ptable.lock);
    cprintf("Invalid pid..\n");
  }

  release(&ptable.lock);
  return p->parent->pid;

}
```

### 3.3.2 Declarations in `user.h` and `defs.h`

`user.h`:

```
...
int getppid(int);
...
```

`defs.h`:

```
...
int getppid(void);
...
```

### 3.3.3 System Call Implementation in `sysproc.c`

```c
int sys_getppid(void){
  return getppid();
}
```

### 3.3.4 Adding System Call in `syscall.c`

Declare the function as `extern`:

```c
extern int sys_getppid(void);
```

Add the system call to the `syscalls` array:

```c
static int (*syscalls[])(void) = {
    ...
    [SYS_getppid] sys_getppid,
    ...
};
```

### 3.3.5 Adding Identifier in `syscall.h`

Assign a unique number to the system call:

```c
#define SYS_getppid 34
```

### 3.3.6 Adding user program for testing

To test the newly implemented system call, we add a user program to the system that invokes the system call to verify its correctness. For testing the `getppid` system call, we created a simple program named `test_getppid.c`:

test$_{g}etppid.c$

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

int main() {
    int parent_pid = getpid(); // Get the parent process ID
    printf(1,"Parent:My id is %d\n",parent_pid);
    int pid = fork();          // Fork a child process

    if (pid < 0) {
        printf(1,"Fork failed\n");
        exit();
    }

    if (pid == 0) {
        // Child process
        int ppid = getppid(getpid());
        printf(1,"Child: My parent PID is %d, expected %d\n", ppid, parent_pid);
        if (ppid != parent_pid) {
            printf(1,"Test failed: Parent PID mismatch\n");
            exit();
        } else {
            printf(1,"Test passed\n");
            exit();
        }
    } else {
        // Parent process
        wait(); // Wait for the child to finish
    }

    exit();
}
```

Once the user program is added, we need to include this file in the EXTRAS section of the Makefile and add the corresponding executable to the UPROGS section. Makefile

```
...
UPROGS=\
    ...
    _test_getppid\
    ...
...
EXTRA=\
    ...
    test_getppid.c\
```

```
    ...
...
```

### 3.3.7 Outputs

The following screenshots illustrate the functionality and correctness of the implemented system calls:

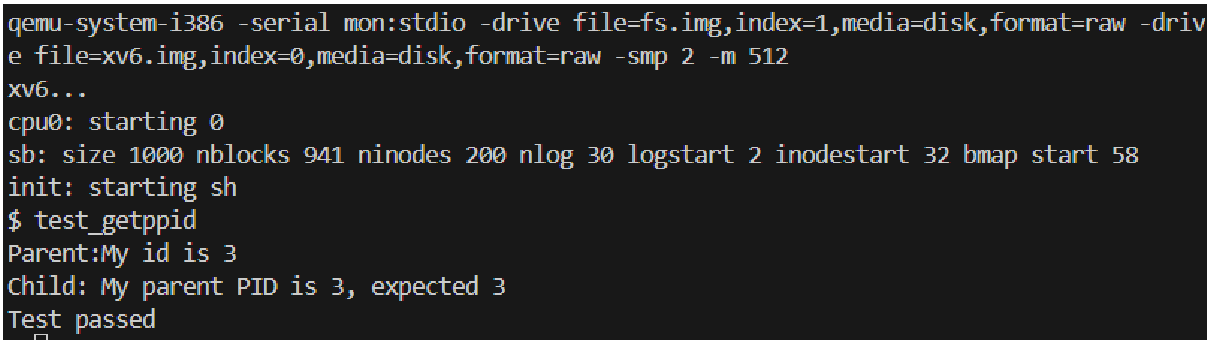



Figure 4: test results of get parent process id

## 3.4 Setting Process Priority

### 3.4.1 Function Definition in proc.c

```
int get_priority(void){
  int pid;
  int priority = -1;
  if(argint(0, &pid) < 0){
    return -1;
  }

  struct proc *p;

  acquire(&ptable.lock);
  //Find the process with the given pid
  for(p = ptable.proc; p<&ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      priority =  p->priority;
      break;
    }
  }
  release(&ptable.lock);
  if(priority==-1){
    return -2; // pid not found
  }

  return priority;
}

int set_priority(void) {
    int pid, priority, setPrior = -1;

    if (argint(0, &pid) || argint(1, &priority)) return -1;

    acquire(&ptable.lock);

    struct proc *p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            p->priority = priority;
            setPrior = 0;
        }
    }

    release(&ptable.lock);
    return setPrior; // -1 for failure, 0 for success
}
```

### 3.4.2 Declarations in user.h and defs.h

user.h:

```
...
int get_priority(int);
int set_priority(int, int);
...
```

defs.h:

```
...
int get_priority(void);
int set_priority(void);
...
```

### 3.4.3 System Call Implementation in sysproc.c

```
int sys_get_priority(void){
  return get_priority();
}
int sys_set_priority(void) {
    return set_priority();
}
```

### 3.4.4 Adding System Call in syscall.c

Declare the function as extern:

```
extern int sys_get_priority(void);
extern int sys_set_priority(void);
```

Add the system call to the syscalls array:

```
static int (*syscalls[])(void) = {
    ...
    [SYS_get_priority] sys_get_priority,
    [SYS_set_priority] sys_set_priority,
    ...
};
```

### 3.4.5 Adding Identifier in syscall.h

Assign a unique number to the system call:

```
#define SYS_get_priority 28
#define SYS_set_priority 29
```

### 3.4.6 Adding user program for testing

To test the newly implemented system call, we add a user program to the system that invokes the system call to verify its correctness. For testing the set_priority system call, we created a simple program named test_get_set_p.c.c:

test_get_set_p.c

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

int main() {
    int pid = getpid();
    int priority = get_priority(pid);

    printf(1, "Process: PID = %d, Priority = %d\n", pid, priority);

    printf(1,"Setting the priority\n");
    set_priority(pid, 20);

    priority = get_priority(pid);

    printf(1, "Process: PID = %d, Priority = %d\n", pid, priority);

    exit();
}
```

Once the user program is added, we need to include this file in the EXTRAS section of the Makefile and add the corresponding executable to the UPROGS section.

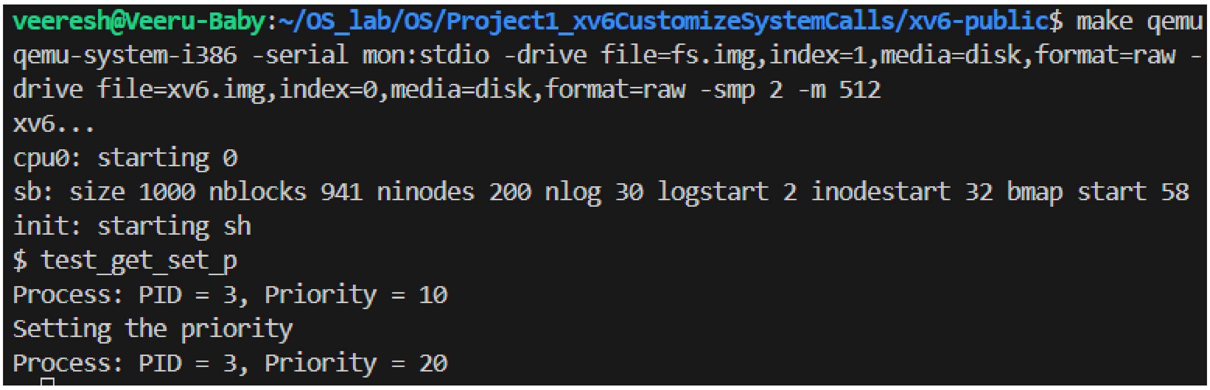
Makefile

```
...
UPROGS=\
    ...
    _test_get_set_p\
    ...
...

EXTRA=\
        ...
    test_get_set_p.c\
    ...
...
```

### 3.4.7 Outputs

The following screenshots illustrate the functionality and correctness of the implemented system calls:



```
veeresh@Veeru-Baby:~/OS_lab/OS/Project1_xv6CustomizeSystemCalls/xv6-public$ make qemu
qemu-system-i386 -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
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test_get_set_p
Process: PID = 3, Priority = 10
Setting the priority
Process: PID = 3, Priority = 20
```

Figure 5: test results of get and set priority

## 3.5 Semaphores for Process Synchronization

### 3.5.1 Functions Defined in proc.c

```c
int sem_init(int sem, int value) {
    acquire(&sema[sem].lock);
    if (sema[sem].active == 0) {
        sema[sem].active = 1;
        sema[sem].value = value;
    } else {
        return -1;
    }

    release(&sema[sem].lock);
    return 0;
}

int sem_destroy(int sem) {
    acquire(&sema[sem].lock);
    sema[sem].active = 0;
    release(&sema[sem].lock);
    return 0;
}

int sem_wait(int sem, int count) {
    acquire(&sema[sem].lock);

    if (sema[sem].value >= count) {
        sema[sem].value -= count;
    } else {
        while (sema[sem].value < count) {
            sleep(&sema[sem], &sema[sem].lock);
        }
        sema[sem].value -= count;
    }

    release(&sema[sem].lock);
    return 0;
}

int sem_signal(int sem, int count) {
    acquire(&sema[sem].lock);
    sema[sem].value += count;
    wakeup(&sema[sem]);
    release(&sema[sem].lock);
    return 0;
}
```

### 3.5.2 Declarations in user.h and defs.h

user.h:

```
...
int sem_init(int, int);
int sem_destroy(int);
int sem_wait(int, int);
int sem_signal(int, int);
...
```

defs.h:

```
...
int sem_init(void);
int sem_destroy(void);
int sem_wait(void);
int sem_signal(void);
...
```

### 3.5.3 System Call Implementation in sysproc.c

```
int sys_sem_init(void) {
    int sem, value;
    if (argint(0, &sem) < 0 || argint(1, &value) < 0) return -1;
    return sem_init(sem, value);
}

int sys_sem_destroy(void) {
    int sem;
    if (argint(0, &sem) < 0) return -1;
    return sem_destroy(sem);
}

int sys_sem_wait(void) {
    int sem, count;
    if (argint(0, &sem) < 0 || argint(1, &count) < 0) return -1;
    return sem_wait(sem, count);
}

int sys_sem_signal(void) {
    int sem, count;
    if (argint(0, &sem) < 0 || argint(1, &count) < 0) return -1;
    return sem_signal(sem, count);
}
```

### 3.5.4 Adding System Call in syscall.c

Declare the functions as extern:

```
extern int sys_sem_init(void);
extern int sys_sem_destroy(void);
extern int sys_sem_wait(void);
extern int sys_sem_signal(void);
```

Add system calls to the syscalls array:

```
static int (*syscalls[])(void) = {
    ...
    [SYS_sem_init] sys_sem_init,
    [SYS_sem_destroy] sys_sem_destroy,
    [SYS_sem_wait] sys_sem_wait,
    [SYS_sem_signal] sys_sem_signal,
    ...
};
```

### 3.5.5 Adding Identifiers in syscall.h

```
#define SYS_sem_init 30
#define SYS_sem_destroy 31
#define SYS_sem_wait 32
#define SYS_sem_signal 33
```

### 3.5.6 Adding user program for testing

For testing semaphores, we created a simple program named test_semaphore.c to verify its correctness:

test_semaphore.c

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

#define SEM_MUTEX1 0
#define SEM_MUTEX2 1

void thread_func1() {
    printf(1, "Statement A1\n");
    sem_wait(SEM_MUTEX1, 1);
    printf(1, "Statement A2\n");
    sem_signal(SEM_MUTEX2, 1);
}

void thread_func2() {
    printf(1, "Statement B1\n");
    sem_signal(SEM_MUTEX1, 1);
    sem_wait(SEM_MUTEX2, 1);
    printf(1, "Statement B2\n");
}

int main(void)
{
    // Initialize semaphores to 0
    sem_init(SEM_MUTEX1, 0);
    sem_init(SEM_MUTEX2, 0);

    // Fork first thread
    if (fork() == 0) {
        thread_func1();
        exit();
    }
    // Fork second thread
    if (fork() == 0) {
        thread_func2();
        exit();
    }
    // Wait for both child processes
    wait();
    wait();

    sem_destroy(SEM_MUTEX1);
    sem_destroy(SEM_MUTEX2);

    exit();
}
```

Once the user program is added, we need to include this file in the EXTRAS section of the Makefile and add the corresponding executable to the UPROGS section.

Makefile

```
...
UPROGS=\
    ...
    _test_semaphore\
    ...
...
...
EXTRA=\
        ...
    test_semaphore.c\
    ...
...
```

### 3.5.7 Outputs

The following screenshots illustrate the functionality and correctness of the implemented semaphores:

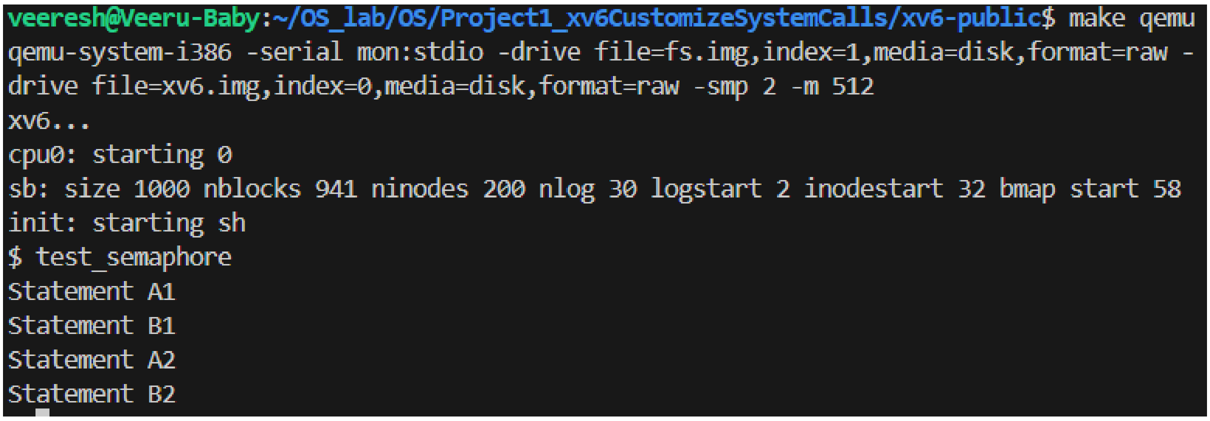



Figure 6: test results of semaphore

## 3.6 Priority Scheduler

### 3.6.1 Function Definition in proc.c

The priority of the process is defined during the process allocation to a default value.Then during the scheduling the scheduler will schedule the process according to the priority.The scheduler will first find the highest priority process then save the current process state and execite the highest priority process.

Changes in allocproc function

```c
static struct proc* allocproc(void)
{
    struct proc *p;
    char *sp;
    acquire(&ptable.lock);

    // Loop to find an UNUSED process slot.
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state == UNUSED) {
          // Initialize the waiting_for field
            p->waiting_for = -1;
            goto found;
        }
    }

    release(&ptable.lock);
    return 0;

found:
    p->state = EMBRYO;
    p->pid = nextpid++;
    p->priority = 10; //this is the default priority for every process

    release(&ptable.lock);

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

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

    // Set up new context to start executing at forkret,
    //which returns to trapret.
    sp -= 4;
    *(uint*)sp = (uint)trapret;
    sp -= sizeof *p->context;
    p->context = (struct context*)sp;
    memset(p->context, 0, sizeof *p->context);
    p->context->eip = (uint)forkret;
    return p;
}
```

Changes in scheduler function

```
void scheduler(void)
{
  struct proc *p,*high_p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

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

    high_p = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      if(high_p == 0 || p->priority > high_p->priority){
        high_p = p;
      }
    }

    if(high_p!=0){
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state == RUNNING && p->priority < high_p->priority){
          // Save the state of current running process
          p->state = RUNNABLE;
          // Mark it as RUNNABLE so it can run again later
        }
      }
      c->proc = high_p;
      switchuvm(high_p);
      high_p->state = RUNNING;

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

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

### 3.6.2 Process struct change in proc.h

proc.h:

```
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 waiting_for;
  int wait_state;
  int priority;                //priority of the process
};
```

### 3.6.3 Adding user program for testing

For testing the Priority Scheduling, we created a simple program named test_.priority.c to verify its correctness:

test_priority.c

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

void spin(int duration) {
    uint start = uptime();
    while(uptime() - start < duration) { }
}

int main(void) {
    int pid1, pid2, pid3;
    int orig_priority;

    // Test : Priority Scheduling
    printf(1, "\nTest: Testing priority-based scheduling\n");

    // Create first child with low priority (higher number)
    if((pid1 = fork()) == 0) {
```

```
        int cur_pid1 = getpid();
        set_priority(cur_pid1,12);  // Lower priority
        sleep(20); //buffer for the priority to be set
        printf(1, "Low priority process (pid %d) starting\n", getpid());
        spin(100);  // Spin for some time
        printf(1, "Low priority process (pid %d) finishing\n", getpid());
        exit();
    }

    // Create second child with medium priority
    if((pid2 = fork()) == 0) {
        int cur_pid2 = getpid();
        set_priority(cur_pid2,15);  // Medium priority
        sleep(20); //buffer for prirotiy to be set
        printf(1, "Medium priority process (pid %d) starting\n",
                 getpid());
        spin(100);  // Spin for some time
        printf(1, "Medium priority process (pid %d) finishing\n",
               getpid());
         exit();
    }

    // Create third child with high priority
    if((pid3 = fork()) == 0) {
         int cur_pid3 = getpid();
         set_priority(cur_pid3,20);   // Higher priority (lower number)
         sleep(20); //buffer for the priroity to be set
         printf(1, "High priority process (pid %d) starting\n",
             getpid());
         spin(100);  // Spin for some time
         printf(1, "High priority process (pid %d) finishing\n",
               getpid());
         exit();
    }

    // Wait for all children to complete
    wait();
    wait();
    wait();

    printf(1, "All tests completed!\n");
    exit();
}
```

Once the user program is added, we need to include this file in the EXTRAS section of the Makefile and add the corresponding executable to the UPROGS section.

Makefile

```
...
UPROGS=\
    ...
    _test_priority\
    ...
...
EXTRA=\
        ...
    test_priority.c\
    ...
...
```

#### 3.6.4 Outputs

The following screenshots illustrate the functionality and correctness of the implemented system calls:

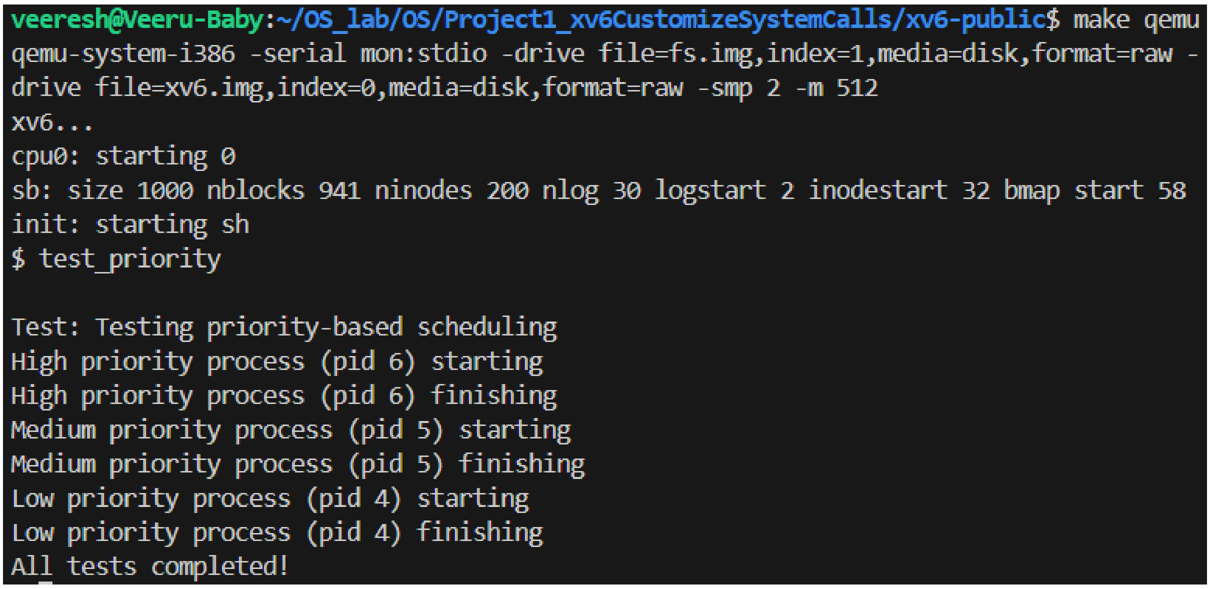



Figure 7: test results of priority scheduling