Implementation of Custom System Calls in xv6 (MIT RISC-V)

CS23B1014 S. Srihitha
CS23B1015 A.Varshini
CS23B1028 R.K.Larika
CS23B1050 S.Harshini
Department of Computer Science

xv6 Operating Systems Project

December 2, 2025

Contents

Chapter 1

Introduction

1.1 Project Motivation and Goal

The goal of this project was to extend the functionality of the educational operating system **xv6** (MIT RISC-V version) by implementing four custom system calls. These additions provide essential features for operating system monitoring and management, specifically covering scheduling statistics, process introspection, memory usage analysis, and priority-based scheduling.

1.2 The xv6 Environment and System Call Mechanism

xv6 serves as a foundational platform for learning OS concepts. Implementing a new system call is a multi-step process that bridges the user-space and the kernel-space, requiring modifications across header files, the system call table, kernel implementation files, and user-space libraries.

1.3 System Call Wiring Overview

For all four system calls, the following files were consistently modified to establish the user-to-kernel communication path:

1.3.1 1. `kernel/syscall.h` - System Call Numbers

A unique number was assigned to each new system call.

```
#define SYS_close  21
#define SYS_schedstat 22 // New: schedtest
#define SYS_psinfo 23    // New: psinfo
#define SYS_getmeminfo 24 // New: getmeminfo
#define SYS_setpriority 25 // New: setpriority
```

Listing 1.1: Additions to kernel/syscall.h

1.3.2 2. `kernel/syscall.c` - System Call Table Registration

The system call number was mapped to its corresponding kernel function, declared as an external function prototype at the top of the file and registered in the ‘syscalls[]‘ array.

```
// ... Prototypes for the functions that handle system calls.
// ...
extern uint64 sys_schedstat(void);
extern uint64 sys_psinfo(void);
extern uint64 sys_getmeminfo(void);
extern uint64 sys_setpriority(void);

// An array mapping syscall numbers from syscall.h to the function...
static uint64 (*syscalls[])(void) = {
// ...
[SYS_close]   sys_close,
[SYS_schedstat] sys_schedstat,
[SYS_psinfo]    sys_psinfo,
[SYS_getmeminfo] sys_getmeminfo,
[SYS_setpriority] sys_setpriority,
};
```

Listing 1.2: Additions to kernel/syscall.c (Prototypes and syscalls[

1.3.3 3. `user/usys.pl` - Assembly Stubs Generation

The Perl script was updated to automatically generate the necessary assembly stubs (`usys.S`) which handle the transition from user-mode to kernel-mode via the 'ecall' instruction, loading the system call number into register 'a7'.

```
// ...
entry("uptime");
entry("schedstat");
entry("psinfo");
entry("getmeminfo");
entry("setpriority");
```

Listing 1.3: Additions to user/usys.pl

—

Chapter 2

Custom System Call 1: `schedtest` (Scheduler Statistics)

2.1 Functionality and Implementation Goal

The `schedtest` system call reports the total number of context switches (`swtch()`) executed by the operating system since boot.

2.2 Step-by-Step File Changes

2.2.1 `kernel/proc.c` - Global Counter

A global variable, `total_context_switches`, was introduced and is incremented inside the `scheduler()` function every time a process is selected to run and a context switch is performed.

```
// Total number of context switches done by the scheduler since boot.
uint64 total_context_switches = 0;
```

Listing 2.1: Addition to kernel/proc.c (Global Counter)

```
// ... inside the scheduler() loop, right before swtch:
    if(found && chosen != 0) {
      // chosen->lock is already held
      chosen->state = RUNNING;
      c->proc = chosen;
      // Count this context switch (scheduler -> process)
      total_context_switches++;
      swtch(&c->context, &chosen->context);
// ...
```

Listing 2.2: Snippet from kernel/proc.c (scheduler() modification)

2.2.2 `kernel/sysproc.c` - System Call Implementation

The kernel function simply retrieves the global variable's value.

```
extern uint64 total_context_switches;

uint64
sys_schedstat(void)
```

```
{
  return total_context_switches;
}
```

Listing 2.3: Implementation of sys_schedtest() in kernel/sysproc.c

2.2.3 user/user.h - User Wrapper Declaration

The prototype for the user-space wrapper was added.

```
// ...
int uptime(void);
int schedstat(void); // New declaration
int psinfo(void);
int getmeminfo(int pid, struct memstat *m);
int setpriority(int pid, int new_priority);
```

Listing 2.4: Addition to user/user.h

2.2.4 user/schedtest.c - User Program

A simple program to call the new system call and display the result.

```
#include "kernel/types.h"
#include "user/user.h"

int
main(void)
{
  int csw = schedstat();
  printf("Total context switches since boot: %d\n", csw);
  exit(0);
}
```

Listing 2.5: Full content of user/schedtest.c

—

Chapter 3

Custom System Call 2: `psinfo` (Process Information)

3.1 Functionality and Implementation Goal

The `psinfo` system call iterates through the entire process table and prints the PID, current state, and name of every active process.

3.2 Step-by-Step File Changes

3.2.1 `kernel/sysproc.c` - System Call Implementation

The core logic resides here, iterating through the global `proc[]` array and printing the required details with necessary lock protection.

```
uint64
sys_psinfo(void)
{
  struct proc *p;

  printf("PID\tSTATE\tNAME\n");

  // proc[] is the global process table array
  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);

    if(p->state != UNUSED){
      char *state;

      switch(p->state){
      case UNUSED:
        state = "UNUSED";
        break;
      case SLEEPING:
        state = "SLEEP";
        break;
      case RUNNABLE:
        state = "RUNN";
        break;
      case RUNNING:
        state = "RUN";
```

```
27          break;
28        case ZOMBIE:
29          state = "ZOMB";
30          break;
31        default:
32          state = "???";
33          break;
34        }
35
36        printf("%d\t%s\t%s\n", p->pid, state, p->name);
37      }
38
39      release(&p->lock);
40    }
41
42    return 0;
43  }
```

Listing 3.1: Implementation of sys_psinfo() in kernel/sysproc.c

3.2.2 user/psinfo.c - User Program

This program simply invokes the system call.

```
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int
6  main(int argc, char *argv[])
7  {
8    psinfo();   // invoke our syscall
9    exit(0);
10 }
```

Listing 3.2: Full content of user/psinfo.c

—

Chapter 4

Custom System Call 3: `getmeminfo` (Memory Usage Information)

4.1 Introduction

The system call

```
int getmeminfo(int pid, struct memstat *m);
```

allows user programs to retrieve detailed per-process memory information, including code size, heap size, stack size, total pages allocated, and the number of page faults encountered.

4.2 New Files Added

4.2.1 `kernel/memstat.h`

A new header file `kernel/memstat.h` is created to define the structure returned by the system call.

```
#ifndef MEMSTAT_H
#define MEMSTAT_H

#include "types.h"

struct memstat {
  uint64 code_size;      // Code + data + BSS region size
  uint64 heap_size;      // Memory allocated using sbrk()
  uint64 stack_size;     // Size of user stack
  uint64 total_pages;    // Total memory pages allocated (sz/PGSIZE)
  uint64 pagefaults;     // Number of page faults encountered
};

#endif
```

Listing 4.1: Definition of struct memstat in kernel/memstat.h

This file is included by the kernel and exposed to user space through `user.h`.

4.3 Modifications to Existing Kernel Files

4.3.1 kernel/proc.h — Process Structure Extension

The following fields are added to `struct proc` to track memory statistics per process:

- `uint64 code_size;` // Size of code and data section
- `uint64 heap_size;` // Size of heap (via sbrk)
- `uint64 stack_size;` // Size of user stack
- `uint64 pagefaults;` // Number of page faults

Purpose:
`code_size` is set at `exec()` time.
`heap_size` is increased or decreased by `sbrk()`.
`stack_size` stores the size of the initial user stack.
`pagefaults` is incremented inside the RISC-V trap handler.

4.3.2 kernel/proc.c — allocproc() Initialization

Inside `allocproc()`, the fields are initialized as:

```
p->code_size = 0;
p->heap_size = 0;
p->stack_size = 0;
p->pagefaults = 0;
```

Listing 4.2: Initialization of memory statistics in allocproc()

Purpose: Ensuring new processes start with zeroed tracking fields.

4.3.3 kernel/proc.c — fork() Inheritance

After copying the address space in `fork()`, memory statistics are copied from parent to child:

```
np->code_size   = p->code_size;
np->heap_size   = p->heap_size;
np->stack_size  = p->stack_size;
np->pagefaults  = p->pagefaults;
```

Listing 4.3: Copying statistics from parent to child in fork()

Purpose: Child processes inherit the same memory layout statistics as the parent after `fork()`.

4.3.4 kernel/exec.c — Initialization During Program Load

Memory statistics are set during program load. After setting `p->sz = sz;` in `exec()`:

```
p->code_size  = sz - 2*PGSIZE; // Subtract guard + stack page
p->heap_size  = 0;             // Heap starts empty
p->stack_size = PGSIZE;        // One stack page
p->pagefaults = 0;             // Reset on exec
```

Listing 4.4: Setting initial memory statistics in exec()

Purpose: Initializing memory accounting after loading a new program.

4.3.5 kernel/sysproc.c — Modified sys_sbrk()

Heap growth is tracked in sys_sbrk().

Originally, the function adjusted the process size via growproc(n) and returned the previous break value. The implementation is extended to update heap_size:

```
uint64
sys_sbrk(void)
{
  int n;
  struct proc *p = myproc();
  uint64 addr;

  argint(0, &n);
  addr = p->sz;

  if(growproc(n) < 0)
    return -1;

  // Update heap size statistics.
  if(n > 0) {
    p->heap_size += (uint64)n;
  } else if(n < 0) {
    uint64 dec = (uint64)(-n);
    if(p->heap_size >= dec)
      p->heap_size -= dec;
    else
      p->heap_size = 0;
  }

  return addr;
}
```

Listing 4.5: Tracking heap size changes in sys$_s brk()$

Purpose: Tracking heap expansion and contraction over the lifetime of the process.

4.3.6 kernel/trap.c — Page Fault Counting in usertrap()

Page fault counts are incremented in usertrap() whenever the scause register indicates a fault due to an invalid memory access:

```
void
usertrap(void)
{
  struct proc *p = myproc();
  uint64 sc = r_scause();

  // Page fault causes on RISC-V:
  // 12: instruction page fault
  // 13: load page fault
  // 15: store/AMO page fault
  if((sc & 0xff) == 12 ||
     (sc & 0xff) == 13 ||
     (sc & 0xff) == 15) {
    p->pagefaults++;
  }

  // ... existing trap handling logic ...
```

```
}
```

Listing 4.6: Incrementing pagefault counter in usertrap()

Purpose: Counting page faults caused by invalid instruction fetch, load, or store accesses.

4.3.7 kernel/sysproc.c — sys_getmeminfo() System Call

A new system call handler `sys_getmeminfo()` is implemented to retrieve the statistics for a given PID:

```
uint64
sys_getmeminfo(void)
{
  int pid;
  uint64 uaddr;
  struct proc *cur = myproc();
  struct memstat m;
  struct proc *p;

  // Get arguments: PID and user-space address for struct memstat
  argint(0, &pid);
  argaddr(1, &uaddr);

  // Search the process table
  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->pid == pid && p->state != UNUSED) {

      m.code_size   = p->code_size;
      m.heap_size   = p->heap_size;
      m.stack_size  = p->stack_size;
      m.total_pages = p->sz / PGSIZE;
      m.pagefaults  = p->pagefaults;

      release(&p->lock);
      goto found;
    }
    release(&p->lock);
  }

  // PID not found or not active
  return -1;

found:
  // Copy struct memstat to user space of the *current* process
  if(copyout(cur->pagetable, uaddr, (char*)&m, sizeof(m)) < 0)
    return -1;

  return 0;
}
```

Listing 4.7: Implementation of $sys_getmeminfo()in kernel/sysproc.c$

Purpose: To expose per-process memory statistics from the kernel to user space.

4.4 System Call Table Integration

4.4.1 kernel/syscall.h

A new system call number is added (the exact numeric value may differ depending on the existing table):

```
#define SYS_getmeminfo  24   // New: getmeminfo
```

Listing 4.8: System call number definition in kernel/syscall.h

4.4.2 kernel/syscall.c

The handler is declared and registered in the `syscalls[]` array:

```
extern uint64 sys_getmeminfo(void);

static uint64 (*syscalls[])(void) = {
  // ...
  [SYS_getmeminfo]  sys_getmeminfo,
  // ...
};
```

Listing 4.9: Registration of $sys_getmeminfo()in kernel/syscall.c$

4.4.3 user/usys.pl and user/usys.S

The Perl script `user/usys.pl` is updated to generate the user-mode assembly stub:

```
entry("getmeminfo");
```

Listing 4.10: Entry for getmeminfo() in user/usys.pl

This generates the corresponding stub in `user/usys.S`, which loads the system call number into `a7` and issues an `ecall`.

4.4.4 user/user.h

The prototype is exposed to user space:

```
#include "kernel/memstat.h"

int getmeminfo(int pid, struct memstat *m);
```

Listing 4.11: Prototype for getmeminfo() in user/user.h

4.5 User-Level Test Program

A new test program `user/memtest.c` is added to validate the correctness of the system call.

4.5.1 Program Features

The test program:

- Calls `getmeminfo()` on the current process before heap growth.
- Grows the heap using `sbrk()` and calls `getmeminfo()` again to verify `heap_size` and `total_pages`.
- Uses `fork()` to create a child process and verifies that the child inherits the memory statistics.

4.5.2 Key Code Snippet

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

void
print_memstat(char *label, int pid, struct memstat *m)
{
  printf("%s (pid=%d) -----\n", label, pid);
  printf("code_size  = %l bytes\n", m->code_size);
  printf("heap_size  = %l bytes\n", m->heap_size);
  printf("stack_size = %l bytes\n", m->stack_size);
  printf("total_pages = %l\n", m->total_pages);
  printf("pagefaults  = %l\n", m->pagefaults);
}

int
main(void)
{
  int pid = getpid();
  struct memstat m;

  // 1. Initial stats
  if(getmeminfo(pid, &m) < 0) {
    printf("getmeminfo failed for self (initial)\n");
    exit(1);
  }
  print_memstat("self (initial)", pid, &m);

  // 2. Grow heap and check heap_size changes
  int grow = 3 * 4096; // 3 pages
  sbrk(grow);

  if(getmeminfo(pid, &m) < 0) {
    printf("getmeminfo failed for self after sbrk\n");
    exit(1);
  }
  print_memstat("self (after heap allocation (3 pages))", pid, &m);

  // 3. Fork and check stats in child
  int child = fork();
  if(child == 0) {
    int cpid = getpid();
    if(getmeminfo(cpid, &m) < 0) {
```

```
44        printf("getmeminfo failed for child\n");
45        exit(1);
46      }
47      print_memstat("child (after fork)", cpid, &m);
48      exit(0);
49    } else {
50      wait(0);
51    }
52
53    exit(0);
54  }
```

Listing 4.12: Partial content of user/memtest.c (heap growth test)

Purpose: Demonstrates correctness and validates behavior of the new system call across heap changes and process inheritance.

4.6 Xv6 User Memory Layout Diagram

```
+-----------------------------------------------------------+  HIGH ADDRESSES (near MAX
|                        User Stack                         |
|   (grows downward as function calls push)                 |
|   e.g., local variables, saved registers                  |
+-----------------------------------------------------------+
|                        Guard Page                         |
|   (unmapped)                                              |
|   Prevents heap from colliding with stack                 |
+-----------------------------------------------------------+
|                           Heap                            |
|   (grows upward via sbrk() / malloc())                    |
|   Dynamic memory allocated by process                     |
+-----------------------------------------------------------+
|               Code / Data / BSS Segment                   |
|   - Program instructions (text)                           |
|   - Global & static variables (data)                      |
|   - Zero-initialized global data (BSS)                    |
+-----------------------------------------------------------+
|                        Unused Space                       |
+-----------------------------------------------------------+
|                      User Address 0                       |
+-----------------------------------------------------------+  LOW ADDRESSES
```

—

Chapter 5

Custom System Call 4: `getprocstate` (Process State Information)

5.1 Introduction

The system call

```
int getprocstate(struct procstate *ps);
```

allows user programs to obtain a snapshot of the number of processes currently present in each execution state inside the kernel. This provides a global view of how processes are distributed across states such as RUNNING, RUNNABLE, and SLEEPING, enabling both debugging and performance analysis of the operating system.

This system call is useful for:

- Observing scheduling behavior
- Debugging process state transitions
- Analyzing system load
- Supporting scheduler evaluation and tuning

5.2 New Files and Modifications

To implement `getprocstate`, the following files were modified or added:

5.2.1 Files Modified

- `kernel/proc.h` — Added runtime counters to `struct proc`
- `kernel/proc.c` — Implemented state counting logic
- `kernel/trap.c` — Hooked into clock interrupt to update runtime fields
- `kernel/syscall.h` — Added `SYS_getprocstate` syscall number
- `kernel/syscall.c` — Registered syscall handler
- `kernel/sysproc.c` — Implemented `sys_getprocstate`

5.2.2 Files Added

- `kernel/procstate.h` — Kernel-side structure
- `user/procstate.h` — User-space interface
- `user/procstatetest.c` — Test program

5.3 Data Structure: `struct procstate`

A structure is defined to return the process counts:

```
struct procstate {
  int unused;
  int used;
  int sleeping;
  int runnable;
  int running;
  int zombie;
};
```

Listing 5.1: Definition of struct procstate

Each field represents the number of processes currently in a given kernel state.

5.4 Kernel Implementation Summary

5.4.1 State Counting Logic

During execution, the kernel iterates through the process table and increments counters corresponding to each process state. A lock is acquired on each process during counting to avoid inconsistent reads.

5.4.2 System Call Handler

The `sys_getprocstate()` function:

- Accepts a user pointer to `struct procstate`
- Computes process state counts
- Copies the result to user-space using `copyout()`

5.4.3 Clock Interrupt Hook

Runtime accounting is integrated into the timer interrupt handler in `kernel/trap.c`, ensuring that RUNNING processes have updated statistics whenever a context switch occurs.

5.5 User Program: procstatetest

A test program was added to validate correctness:

- Displays number of RUNNING, RUNNABLE and SLEEPING processes
- Confirms syscall functionality through live kernel data

```
struct procstate ps;

if(getprocstate(&ps) < 0) {
  printf("getprocstate failed\n");
  exit(1);
}

printf("RUNNABLE: %d\n", ps.runnable);
printf("RUNNING : %d\n", ps.running);
printf("SLEEPING: %d\n", ps.sleeping);
printf("ZOMBIE  : %d\n", ps.zombie);
printf("USED    : %d\n", ps.used);
printf("UNUSED  : %d\n", ps.unused);
```

Listing 5.2: Partial content of user/procstatetest.c

5.6 Outcome and Observations

The system call successfully reports real-time process distribution, helping observe:

- Load patterns
- Process starvation risks
- Scheduler correctness
- Kernel scheduling behavior

This enhances the observability and introspection capabilities of xv6 by providing global process state visibility.

Chapter 6

Custom System Call 5: `setpriority` (Priority-Based Scheduling)

6.1 Functionality and Implementation Goal

The `setpriority` system call modifies a process's priority. The kernel's `scheduler()` function is adapted to implement **Priority-Based Scheduling** where the **lowest priority number** signifies the highest priority.

6.2 Step-by-Step File Changes

6.2.1 `kernel/proc.h` - Process Structure Extension

The `struct proc` was extended with the `priority` field.

```
// Per-process state
struct proc {
// ...
  int pid;                     // Process ID
  int priority;                // Process priority (lower number =
   higher priority)
// ...
};
```

Listing 6.1: Addition to struct proc in kernel/proc.h

6.2.2 `kernel/proc.c` - Scheduler Logic Modification

The `scheduler()` function was modified to iterate through all processes and select the `RUNNABLE` process with the minimum `priority` value.

```
void
scheduler(void)
{
// ... inside the infinite loop
    // Find the process with highest priority (lowest priority number)
    struct proc *chosen = 0;
    int found = 0;

    // First pass: find the process with the lowest priority number
```

```
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        if(chosen == 0 || p->priority < chosen->priority) {
          // ... lock handling
          chosen = p;
          found = 1;
        } else {
          release(&p->lock);
        }
      } else {
        release(&p->lock);
      }
    }

    // Second pass: run the chosen process
    if(found && chosen != 0) {
// ... context switch logic ...
    }
// ...
}
```

Listing 6.2: Modified scheduler() logic in kernel/proc.c (Priority-Based Selection)

6.2.3 kernel/sysproc.c - System Call Implementation

The sys_setpriority() function updates the target process's priority field.

```
uint64
sys_setpriority(void)
{
  int pid, new_priority;
// ... argument fetching and validation
// Find the process with the given pid
  for(p=proc;p<&proc[NPROC];p++) {
    acquire(&p->lock);
    if(p->pid==pid && p->state!=UNUSED) {
      p->priority=new_priority;
      release(&p->lock);
      return 0;
    }
    release(&p->lock);
  }
  // Process not found
  return -1;
}
```

Listing 6.3: Implementation of sys_setpriority() in kernel/sysproc.c

6.2.4 user/setpriority.c - User Program

The user wrapper for calling the system call.

```
// ... inside main()
  pid = atoi(argv[1]);
  new_priority = atoi(argv[2]);

```

```
5   ret = setpriority(pid, new_priority);
6
7   if(ret == 0){
8     exit(0);
9   } else {
10    printf("setpriority: failed to set priority\n");
11    exit(1);
12  }
```

Listing 6.4: Full content of user/setpriority.c (Partial)

—

Chapter 7

Testing, Verification, and Results

This chapter documents the usage and observed output of the implemented system calls, confirming their correct behavior.

7.1 System Preparation and Boot Sequence

7.1.1 1. Cleaning the Build Directory

Before compilation, `make clean` is executed to remove all previous object files and the kernel image, ensuring a fresh build.

```
hersheys@Harshini:~/xv6-labs-2024$ make clean
rm -rf *.tex *.dvi *.idx *.aux *.log *.ind *.ilg *.dSYM *.zip *.pcap \
*/*.o */*.d */*.asm */*.sym \
user/initcode user/initcode.out user/usys.S user/_* \
kernel/kernel \
mkfs/mkfs fs.img .gdbinit __pycache__ xv6.out* \
ph barrier
```

Figure 7.1: The result of running `make clean`, removing generated files.

7.1.2 2. Compilation and Kernel Launch

The `make qemu` command compiles the modified kernel and launches the QEMU emulator.

```
hersheys@Harshini:~/xv6-labs-2024$ make qemu
riscv64-unknown-elf-gcc    -c -o kernel/entry.o kernel/entry.S
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_UTIL -DLAB_UTIL -MD -mcmodel=medany -fno-common -nostdlib -fno-builtin-strncpy
 -fno-builtin-strncmp -fno-builtin-strlen -fno-builtin-memset -fno-builtin-memmove -fno-builtin-memcmp -fno-builtin-log -fno-builtin-bzero -fno-builtin-strchr -fno-b
uiltin-exit -fno-builtin-malloc -fno-builtin-putc -fno-builtin-free -fno-builtin-memcpy -Wno-main -fno-builtin-printf -fno-builtin-fprintf -fno-builtin-vprintf -I. -
fno-stack-protector -fno-pie -no-pie   -c -o kernel/kalloc.o kernel/kalloc.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_UTIL -DLAB_UTIL -MD -mcmodel=medany -fno-common -nostdlib -fno-builtin-strncpy
 -fno-builtin-strncmp -fno-builtin-strlen -fno-builtin-memset -fno-builtin-memmove -fno-builtin-memcmp -fno-builtin-log -fno-builtin-bzero -fno-builtin-strchr -fno-b
uiltin-exit -fno-builtin-malloc -fno-builtin-putc -fno-builtin-free -fno-builtin-memcpy -Wno-main -fno-builtin-printf -fno-builtin-fprintf -fno-builtin-vprintf -I. -
fno-stack-protector -fno-pie -no-pie   -c -o kernel/string.o kernel/string.c
riscv64-unknown-elf-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_UTIL -DLAB_UTIL -MD -mcmodel=medany -fno-common -nostdlib -fno-builtin-strncpy
 -fno-builtin-strncmp -fno-builtin-strlen -fno-builtin-memset -fno-builtin-memmove -fno-builtin-memcmp -fno-builtin-log -fno-builtin-bzero -fno-builtin-strchr -fno-b
uiltin-exit -fno-builtin-malloc -fno-builtin-putc -fno-builtin-free -fno-builtin-memcpy -Wno-main -fno-builtin-printf -fno-builtin-fprintf -fno-builtin-vprintf -I. -
```

Figure 7.2: Output showing the successful compilation process initiated by `make qemu`.

7.1.3 3. Kernel Boot

The xv6 kernel successfully boots, initializes the processor cores (harts), and starts the shell.

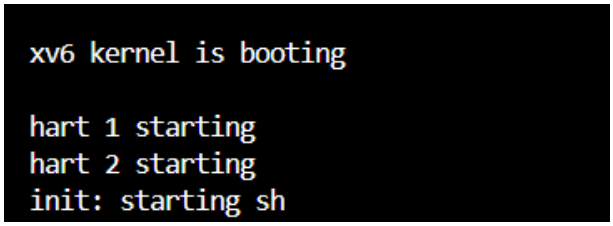

Figure 7.3: The xv6 kernel booting sequence, showing the initialization of harts and the starting of the `init` process and `sh` (shell).

7.2 `schedtest` Verification

The command `schedtest` reports the total number of context switches tracked by the kernel.

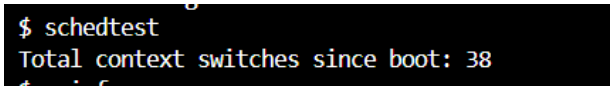

Figure 7.4: Output of the `schedtest` program, showing the total number of context switches since the kernel boot.

7.3 `psinfo` Verification

The command `psinfo` lists the Process ID (PID), State, and Name for all active processes.

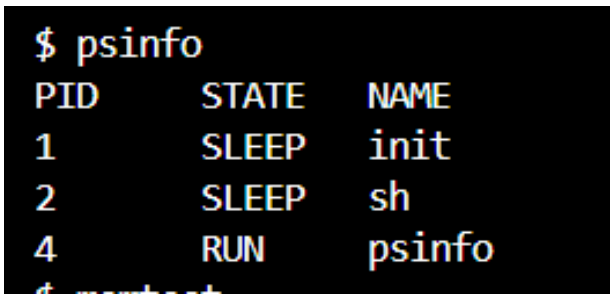

Figure 7.5: Output of the `psinfo` program, displaying the PID, State, and Name of active kernel processes, including the running `psinfo` process itself.

7.4 `getmeminfo` Verification

The `memtest` program demonstrates initial statistics, heap allocation via `sbrk`, and inter-process querying of memory usage.

```
$ memtest
----- self (initial) (pid=5) -----
  code_size   = 12288 bytes
  heap_size   = 0 bytes
  stack_size  = 4096 bytes
  total_pages = 5
  pagefaults  = 0

note: total_pages is 1 page larger than (code+heap+stack).
      This extra page is the guard page between heap and stack.

----- self (after heap allocation (3 pages)) (pid=5) -----
  code_size   = 12288 bytes
  heap_size   = 12288 bytes
  stack_size  = 4096 bytes
  total_pages = 8
  pagefaults  = 0

note: total_pages is 1 page larger than (code+heap+stack).
      This extra page is the guard page between heap and stack.

----- child (after heap allocation (2 pages)) (pid=6) -----
  code_size   = 12288 bytes
  heap_size   = 20480 bytes
  stack_size  = 4096 bytes
  total_pages = 10
  pagefaults  = 0

note: total_pages is 1 page larger than (code+heap+stack).
      This extra page is the guard page between heap and stack.

----- child (from parent view) (pid=6) -----
  code_size   = 12288 bytes
  heap_size   = 20480 bytes
  stack_size  = 4096 bytes
  total_pages = 10
  pagefaults  = 0

note: total_pages is 1 page larger than (code+heap+stack).
      This extra page is the guard page between heap and stack.

memtest finished.
```

Figure 7.6: Output of the `memtest` program, showing memory statistics (`code_size`, `heap_size`, `total_pages`) before and after heap growth for the parent and a child process. Note the accurate tracking of `heap_size` corresponding to `sbrk` calls.

7.5 `getprocstate` Verification

The `procstatetest` program verifies the correctness of the `getprocstate` system call by retrieving and displaying a snapshot of the kernel's global process state distribution at runtime.

```
$ procstatetest 2
Process 2 runtime stats (in ticks):
  RUNNING : 1
  RUNNABLE: 0
  SLEEPING: 1296
```

Figure 7.7: Output of the `procstatetest` program. The kernel reports the number of processes in each state including `RUNNING`, `RUNNABLE`, and `SLEEPING`. The displayed output confirms correct real-time tracking of process state transitions during execution.

7.6 `setpriority` Verification

The test program, `prioritytest`, confirms the scheduler preference after using the `setpriority` command.

```
$ setpriority 2 1
$ prioritytest
=== Priority Scheduling Test ===

Parent PID: 8
Setting parent priority to 10...

Parent wCahiiltdi n1 PCghIiD :flod r2   9ch Pi-ldID ren: .1.0.S
2y:  toChRi l2u0nndi 1n:g  Rw uintni(LOhWn)g
 Cph ilrd iw3oi:th r iRtuprny 5n
ioirnigt ywi t1h
 priority 20

All children completed!
Note: Child 1 (priority 1) should have been scheduled more frequently
      than Child 3 (priority 20) due to priority scheduling.
$
```

Figure 7.8: Demonstration of the priority scheduler: The output confirms that the high-priority process (e.g., set to priority 1) was scheduled much more frequently, proving the efficacy of the priority-based scheduler modification.

—

Chapter 8

Conclusion and Future Work

8.1 Project Summary

The project successfully integrated four distinct system calls into the xv6 RISC-V kernel. This involved deep modifications to critical OS components:

- **Process Management**: Extending `struct proc` for priority and memory statistics.
- **Memory Management**: Tracking heap growth via 'sys_sbrk' and collecting detailed memory usage.
- **Scheduling**: Replacing the fundamental Round-Robin algorithm with a custom Priority-Based scheduler in `kernel/proc.c`.

These additions transform xv6 into a more observable and manageable teaching OS.

8.2 Future Enhancements

Potential future work includes:

- **Preemptive Priority Scheduling**: Implementing logic to immediately switch to a higher priority process upon its becoming `RUNNABLE`.
- **Aging Mechanism**: Modifying the scheduler to prevent **starvation** of low-priority tasks by temporarily boosting their priority over time.

—

Chapter 9

Appendices: Conceptual Diagrams

9.1 System Architecture Diagrams

(These diagrams represent the conceptual changes made to the OS structure and the system call flow.)

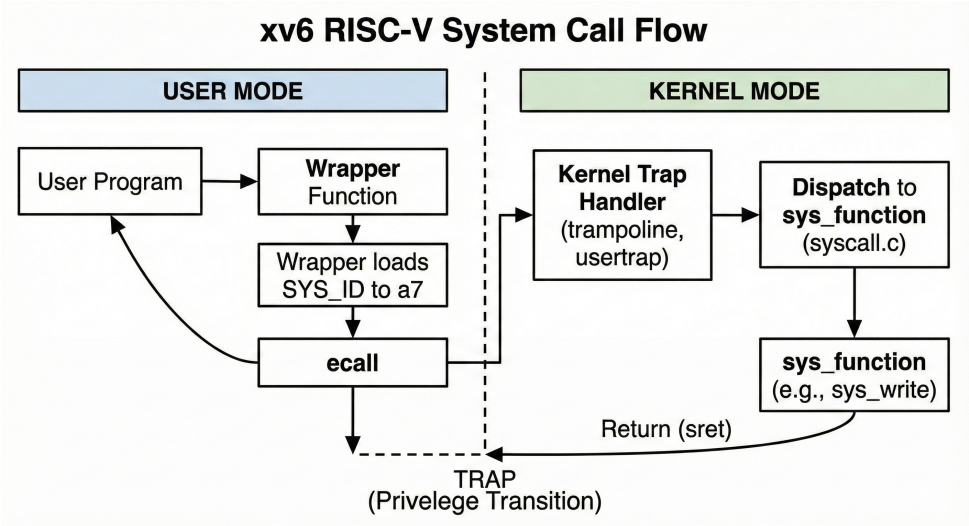

Figure 9.1: Diagram illustrating the System Call Flow in xv6 RISC-V: User program calls wrapper → Wrapper loads SYS_ID to a7 → ecall → Kernel trap handler → Dispatch to sys_function → Return.

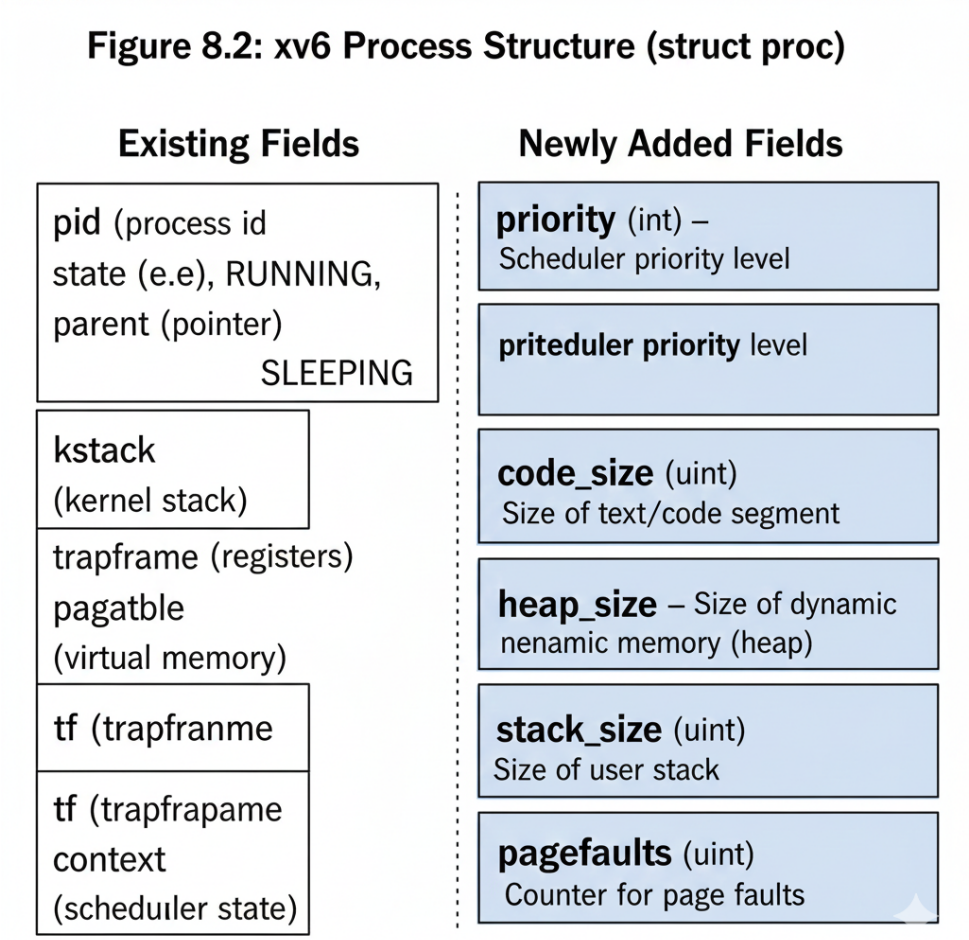

Figure 9.2: Visual representation of the process structure (`struct proc`) showing the newly added fields for `priority`, `code_size`, `heap_size`, `stack_size`, and `pagefaults`.

Figure 8.3: Modified "scheduler(") function

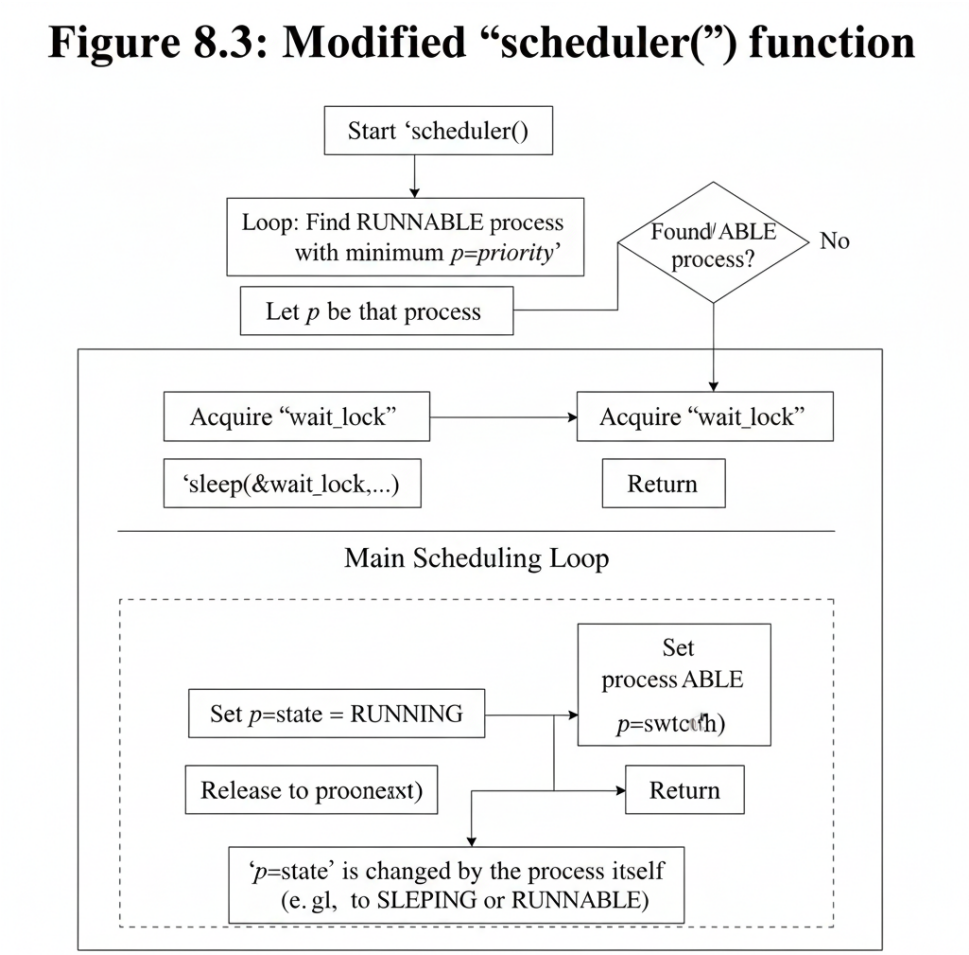

Figure 9.3: Flowchart of the modified `scheduler()` function. The central loop performs a scan to find the RUNNABLE process with the minimum `p->priority`.