A report on

Enhanced Shell Features in xv6

Assignment 1 | Resource Management in Operating System(COL633) | Semester II

Submitted by

Arindam Sal, 2024JCS2041

Date of Submission: 6 March 2025

---

1. Abstract

This report presents the implementation of key enhancements to the **`xv6`** shell, including a username-password login system for access control, a **`history`** command to track executed processes with memory usage, and **`block/unblock`** commands to disable specific system calls. Additionally, we implemented a **`chmod`** command to modify file permissions. Each feature was integrated by modifying system calls and kernel functionality while ensuring system stability. The report outlines our approach, challenges, and improvements beyond the requirements.

2. Introduction

`xv6` is a simple, UNIX-like operating system designed for educational purposes, providing a minimal yet functional environment to understand core OS concepts. It is a reimplementation of Sixth Edition UNIX, focusing on process management, memory handling, and file systems while keeping the codebase small and readable. Unlike modern operating systems, **`xv6`** is lightweight, making it an excellent platform for experimenting with system-level modifications.

Adding Shell Enhancement

The goal of this assignment was to enhance the xv6 shell by adding new functionalities that improve user interaction and system control. This involved implementing a secure login system, where users must authenticate before accessing the shell. Additionally, we introduced a **`history`** command to track executed processes, a **`block/unblock`** feature to restrict system calls, and a **`chmod`** command to modify file permissions. Each of these features required modifying the xv6 kernel by adding new system calls and handling process interactions efficiently.

By implementing these features, we gained deeper insights into how system calls work, how user-space and kernel-space interact, and how security and access control mechanisms can be integrated into an operating system. This report details the methodology behind these implementations, the challenges encountered, and additional improvements made beyond the initial requirements.

3. Implementation Methodology

3.1 Enhanced Shell with Login System

To enhance security in xv6, we implemented a username-password-based login system that restricts shell access to authorized users. This ensures that only a valid user can interact with the shell, adding an extra layer of protection to the system.

3.1.1 Approach

The username and password are defined as macros in the Makefile using:

```
USERNAME=arindam
PASSWORD=jcs242041
```

These values are passed as compile-time constants using the `-D` flag in `CFLAGS`, making them accessible in the code.

3.1.2 Authentication in `init.c`

- When the system boots, `init.c` prompts the user for credentials before launching the shell.
- The program reads the username first. If it matches, it then asks for the password.

```
if (strcmp(username, USERNAME) == 0)
{
      prompt_printf(1, "Enter password: ");
      gets(password, BUFFER_SIZE);
      password[strlen(password) - 1] = '\0';
      if (strcmp(password, PASSWORD) == 0)
      {
          printf(1, "Login successful\n");
          break;
       }
      else
         printf(1, "Incorrect password. Try again.\n");
 }
 else
      printf(1, "Incorrect username. Try again.\n");
```

- If authentication is successful, the shell starts. Otherwise, the user is given three attempts before the system exits.

3.2. `history` Command Implementation

The `history` command was implemented to allow users to track previously executed processes along with their PID, process name, and memory utilization. This required maintaining a history of all processes created in the system and retrieving this information when the history command is executed.

3.2.1 Approach

- Tracking Process Execution:

Modified `allocproc()` to log new processes using `add_to_history()`, which records process details in a fixed-size array.
Each entry in history stores:

`PID (Process ID)`
`Process Name`
`Memory Utilization` (including text, data, BSS, stack, and heap)

- Implementing a System Call (`sys_gethistory`) main underlying logic says

```
…
if (history_count < HISTORY_SIZE) {
        history[history_count].pid = p->pid;
        safestrcpy(history[history_count].name,
p->name, sizeof(p->name));

        // Calculate memory utilization
        history[history_count].mem_usage = p->sz;

        history_count++;
    }
…
```

A new system call `sys_gethistory()` was added to retrieve stored process execution history.
The system call returns a list of process details in ascending execution order.

- Creating the `history` Command:

A user-space command history was added, which invokes `sys_gethistory()`. It prints each process's PID, name, and memory usage.

3.2.2 System Call Integration for sys_gethistory()

To integrate the `sys_gethistory` system call into xv6, several modifications were made across multiple files. First, in `syscall.h`, we assigned a unique system call number by adding `#define SYS_gethistory 22`. Next, in `syscall.c`, we mapped this identifier to its function implementation by adding `[SYS_gethistory] sys_gethistory,` in the `syscalls[]` array. To make this system call accessible from user programs, we updated `usys.S` by including the stub `SYSCALL(gethistory),` allowing user-space applications to invoke `gethistory()` seamlessly. These modifications collectively ensure that the history command can retrieve and display previously executed process details.

3.2.3 Handling Failed `exec()` Calls with execed Flag

In our implementation, we introduced an `execed` flag in the `proc structure` to ensure that only successfully executed processes are recorded in history. This flag is initialized to 0 in `allocproc()` and set to 1 inside `exec()` only if the program loads successfully. The exit() function now checks this flag before adding a process to history, preventing cases where failed `exec()` calls (e.g., due to nonexistent programs or permission errors) result in unnamed or unintended processes being recorded. This modification ensures accuracy in tracking process execution while avoiding redundant or misleading entries in the history system.

3.3 `block/unblock` System Call Implementation

3.3.1 Implementing Blocking Logic (`sys_block` in `sysproc.c`)

If a system call is critical (e.g., fork, exit), blocking is denied.
Otherwise, the requested system call is set to blocked (1).

```
int sys_block(void) {
   ...

    if (syscall_id == SYS_fork || syscall_id == SYS_exit)
        return -1; // Prevent blocking critical system calls

    struct proc *p = myproc();
    if (syscall_id >= 0 && syscall_id < NELEM(p->blocked_syscalls))
{
        p->blocked_syscalls[syscall_id] = 1;
        return 0;
    }
    return -1;
}
```

3.3.2 Implementing Unblocking Logic (`sys_unblock` in `sysproc`.c)

This function resets the flag for a previously blocked system call, allowing it to execute.

```
int sys_unblock(void) {
    int syscall_id;
    if (argint(0, &syscall_id) < 0)
        return -1;

    struct proc *p = myproc();
    if (syscall_id >= 0 && syscall_id <
NELEM(p->blocked_syscalls)) {
        p->blocked_syscalls[syscall_id] = 0;
        return 0;
    }
    return -1;
}
```

3.3.3 Special Treatment: Enforcing System Call Blocking in `syscall.c`

To enforce system call blocking, we modified syscall.c to check whether a system call is blocked before execution. When a process attempts to invoke a system call, the kernel first verifies if it has been marked as blocked in the `blocked_syscalls` array. If the system call is blocked, the kernel denies execution and prints a message indicating that the call is restricted. However, a special case is handled for `SYS_write`, allowing the shell (`sh`) to continue writing output even if `SYS_write` is blocked, ensuring smooth operation. This prevents the shell from becoming unresponsive while still enforcing restrictions on other processes.

```
if (curproc->blocked_syscalls[num]) {
    if (num == SYS_write && strcmp(curproc->name,
"sh") == 0) {
        // Allow shell to write output
    } else {
        cprintf("syscall %d is blocked\n", num);
        curproc->tf->eax = -1;
        return;
    }
}
```

3.3.4 System Call Integration for `Block/Unblock`

To integrate the `sys_block` and `sys_unblock` system calls into xv6, several modifications were made across multiple files. First, in syscall.h, we assigned unique system call numbers by adding `#define SYS_block 23` and `#define SYS_unblock 24`. Next, in syscall.c, we mapped these identifiers to their function implementations by adding `[SYS_block] sys_block`, and `[SYS_unblock] sys_unblock`, in the `syscalls[]` array. To make these system calls accessible from user programs, we updated usys.S by including the stubs `SYSCALL(block)` and `SYSCALL(unblock)`, allowing user-space applications to invoke `block()` and `unblock()` seamlessly. These modifications collectively enable processes to restrict or re-enable specific system calls dynamically while ensuring critical system operations remain unaffected.

3.4 Implementation of `chmod` and File Permission System in xv6

To enhance file security in xv6, we implemented a permission system that controls read, write, and execute operations for files. This was achieved by introducing a chmod system call and modifying file-related system calls (sys_read, sys_write) to enforce permission checks.

3.4.1 Extending the `dinode` Structure (`fs.h`)

A new field perm (permission bits) was added to struct dinode to store file permissions.
The three permission bits work as follows:

```
Bit 0 (Read)
Bit 1 (Write)
Bit 2 (Execute)
```

```
struct dinode {
    ...
    uint perm;  //Permission bit
};
```

3.4.2 Implementing sys_chmod (`sysfile.c`)

Retrieves the file path and mode from user input.
Validates the mode (should be between 0-7, as it represents 3-bit permissions).
Updates the file's permission bits in its inode.

```
int sys_chmod(void) {
…
    if(argstr(0, &path) < 0 || argint(1, &mode) < 0 || mode <
0 || mode > 7)
        return -1;  // Invalid arguments
    struct inode *ip;
    begin_op();
    if((ip = namei(path)) == 0) {
        end_op();
        return -1;}
    ilock(ip);
    ip->dinode.perm = mode;  // Set new permissions
    iupdate(ip);
    iunlockput(ip);
    end_op();
    return 0;}
```

3.4.3 Enforcing Permissions in `sys_read` ,`sys_write` (`sysfile.c`) and `int exec() in exec.c` file

Bitwise AND (&) operator is used to check if a file has read or write permissions before performing the operation. If permission is denied, an error message is printed

```
int sys_read(void) {
  …
…

    if (!(f->ip->dinode.perm &
1)) {  // Check read permission
(bit 0)
        cprintf("Operation read
failed\n");
        return -1;
    }
    return fileread(f, p, n);
}
```

```
int sys_write(void) {
    …
    if(fd < 0 || fd >= NOFILE
|| (f = myproc()->ofile[fd]) ==
0)
        return -1;

    if (!(f->ip->dinode.perm &
2)) {  // Check write
permission (bit 1)
        cprintf("Operation
write failed\n");
        return -1;
    }
    return filewrite(f, p, n);}
```

```
int  exec(char *path, char **argv)
{
 ...
...
  if (!(ip->dinode.perm & 4)) {
    cprintf("Operation execute failed\n");
    iunlockput(ip);
    end_op();
    return -1;
  }
```

Why Inode is Commented Out in `file.h`?
In file.h, the struct `inode` was commented out because it already exists in `fs.h` as part of the file system's structure. Keeping a redundant copy could lead to inconsistencies. Instead, struct file maintains a pointer (`ip`) to the inode, allowing access to its details without duplication:

By referencing the existing inode structure, we avoid redundancy and ensure that all file permission changes are correctly applied across the system.

4. Conclusion

In this assignment, we successfully enhanced the xv6 operating system by implementing several key features:

- **Enhanced Shell with Login System:** We introduced a username-password authentication mechanism to the xv6 shell, ensuring that only authorized users can access the system. This involved modifying the init.c file to prompt for credentials and verify them against predefined values.

- **System Call Blocking Mechanism:** We developed block and unblock system calls, allowing the dynamic enabling or disabling of specific system calls during runtime. This feature enhances security and control within the operating system by permitting administrators to restrict certain functionalities as needed.

- **File Permission Management:** We implemented a chmod system call to manage file permissions, enabling users to set read, write, and execute permissions on files. This addition aligns xv6 more closely with traditional UNIX-like systems, offering granular control over file access.

- **Process History Tracking:** We added a gethistory system call to maintain a history of executed processes. This feature provides users with the ability to review previously run processes, aiding in system monitoring and debugging.

These enhancements collectively improve the functionality, security, and usability of the xv6 operating system, aligning it more closely with modern operating system standards.

5. References

1. Adding-System-Calls-to-xv6 by nicolefrumkin on GitHub: github
2. Xv6 module implementation document on Studocu: Studocu
3. how do i add a system call / utility in xv6 on Stack Overflow: StackOverflow
4. Xv6 Operating System - adding a new system call on GeeksforGeeks: GeeksforGeeks
5. Lab 2: System Calls by KUL: DistrinetResearch

Implementation of

Signal Handling and Priority Based Scheduler in xv6

Resource Management in Computer System (COL633) |Assignment-2 |2024–2025

Submitted by

Arindam Sal
Entry Number: 2024JCS2041

Ayan Shil
Entry Number: 2024JCS2048

Submitted on: 10th April, 2025

Indian Institute of Technology Delhi

M.Tech in Cyber Security

Contents

Abstract

This report presents the design and implementation of two key functionalities within the xv6 operating system: a signal handling mechanism for user-triggered process control and a dynamic priority-based scheduler with profiling capabilities. The signal handling mechanism enables the operating system to respond to specific keyboard inputs (Ctrl+C, Ctrl+B, Ctrl+F, and Ctrl+G) to terminate, suspend, resume, or trigger custom signal handlers for processes running in the foreground. The scheduling enhancement introduces a profiler that computes turnaround time, waiting time, response time, and context switches for each process. It further incorporates a dynamic priority scheduler where the scheduling decision is influenced by tunable parameters α and β, alongside starvation prevention through priority boosting. Together, these modifications enrich xv6 with interactive process control and advanced scheduling logic, simulating realistic OS behavior for resource management.

1 Introduction

Operating systems play a critical role in managing system resources and ensuring smooth interaction between hardware and software. xv6, a Unix-like teaching operating system developed by MIT, provides a minimal yet functional framework for exploring OS internals. This assignment focuses on enhancing xv6 with two important capabilities: signal handling for interactive process control and a priority-based scheduler for efficient CPU management. The first part involves implementing support for user-generated signals triggered via keyboard shortcuts (Ctrl+C, Ctrl+B, Ctrl+F, Ctrl+G), enabling process termination, suspension, resumption, and custom handler invocation. The second part extends the default round-robin scheduler by incorporating a profiling mechanism and a dynamic priority scheduler that responds to runtime and wait-time characteristics of processes. These enhancements provide deeper insight into process management, scheduling policies, and user-kernel interactions in a real operating system environment.

2 Implementation Methodology

2.1 Part 1: Signal Handling Mechanism in xv6

This part implements support for handling user-generated signals via keyboard inputs: Ctrl+C (terminate), Ctrl+B (suspend), Ctrl+F (resume in foreground), and Ctrl+G (custom handler). The implementation involves modifications to `console.c` for capturing

keypresses, `proc.c` for signal delivery, `syscall.c` for system call extensions, and relevant user-level handling in `user.h` and `usys.S`.

2.1.1 Control Flow: Ctrl+C (SIGINT)

When the user presses Ctrl+C, a check in `console.c` identifies the keypress and calls the system call associated with SIGINT. This invokes the `fgproc()` function in `proc.c`, which iterates over the process table and marks all active foreground processes (except `initproc` and the shell) as `killed = 1`. If a process is sleeping, it is transitioned to `RUNNABLE` so it can exit cleanly.

```
if (p != initproc && p->pid != 2 && p->state != UNUSED)
    p->killed = 1;
if (p->state == SLEEPING)
    p->state = RUNNABLE;
```

This ensures the process wakes up, checks the killed flag, and invokes `exit()`. A wakeup is also issued on `&input.r` to unblock the shell if needed.

2.1.2 Control Flow: Ctrl+B (SIGSTOP)

Ctrl+B triggers the SIGBG signal. Inside the `sendsig()` function in `proc.c`, all user processes (excluding `initproc` and the shell) are marked as suspended by setting a custom flag `sigsuspended = 1`. Their parent is also set to `initproc`.

```
if (p->pid > 2)
    p->sigsuspended = 1;
    p->parent = initproc;
```

This effectively suspends all user processes while keeping the shell responsive.

2.1.3 Control Flow: Ctrl+F (Foreground)

Ctrl+F invokes SIGFG through `sendsig()`, which resumes previously suspended background processes by resetting the `sigsuspended` flag to 0 for each.

```
if (p->sigsuspended)
    p->sigsuspended = 0;
```

This allows these processes to be scheduled again by the kernel.

2.1.4 Control Flow: Ctrl+G (SIGCUSTOM)

Ctrl+G initiates the SIGCUSTOM signal to invoke user-registered signal handlers. In `sendsig()`, processes with a valid handler are marked with `sigcustom_pending = 1`.

The actual invocation occurs later inside `trap()` in `trap.c`:

```
if (proc->sigcustom_pending && proc->signal_handler) {
    proc->sigcustom_pending = 0;
    ... // Stack setup
    proc->tf->eip = (uint)proc->signal_handler;
}
```

The kernel sets up the user stack to jump to the registered handler, simulating user-level signal processing.

2.1.5 Additional Notes

All Ctrl key combinations are intercepted in `console.c` by checking input characters and dispatching the respective system calls. Signal-related flags like `sigsuspended` and `sigcustom_pending` are introduced in `struct proc`. System calls and helpers are implemented in `sysproc.c`, and user-space wrappers in `usys.S` and `user.h`.

2.2 Part 2: Scheduler Enhancements in xv6

This part focuses on enhancing the xv6 scheduler by integrating a profiler for performance metrics and implementing a dynamic priority-based scheduling mechanism with starvation prevention. Key modifications span across `proc.c`, `sysproc.c`, and the scheduler loop. Additional fields were added to `struct proc`, including `creation_time`, `end_time`, `response_time`, `ticks_run`, `dyn_priority`, `base_priority`, and `context_switches`.

2.2.1 Scheduler Profiler

Upon process termination, `exit()` computes and prints profiling metrics: - **Turnaround Time (TAT)** = `end_time - creation_time` - **Waiting Time (WT)** = TAT - `ticks_run` - **Response Time (RT)** = first scheduled time - `creation_time` - **Context Switches (CS)** = counter incremented each time the process is scheduled

```
curproc->end_time = ticks;\textbf{}
cprintf("TAT: %d\n", curproc->end_time - curproc->creation_time);
cprintf("WT: %d\n", (curproc->end_time - curproc->creation_time) - curproc->
    ↪ ticks_run);
cprintf("RT: %d\n", curproc->response_time);
cprintf("#CS: %d\n", curproc->context_switches);
```

2.2.2 Dynamic Priority Calculation

The scheduler computes a dynamic priority using the formula:

$$\text{priority} = \text{base\_priority} - \alpha \cdot \text{run\_time} + \beta \cdot \text{wait\_time}$$

This balances CPU usage with wait fairness. The process with the highest computed priority is scheduled. In case of a tie, the process with the lower PID is chosen.

```
curr_priority = p->base_priority - ALPHA * p->ticks_run + BETA * wait;
```

2.2.3 Starvation Avoidance: WAIT_THRESHOLD Boosting

To prevent starvation, a boosting mechanism was considered for processes whose wait time exceeds `WAIT_THRESHOLD`, though not enabled by default. It assigns a very high priority to long-waiting processes.

```
// if (wait > WAIT_THRESHOLD)
// curr_priority = MAX_PRIORITY + 1;
```

This mechanism ensures fairness by eventually selecting even low-priority processes that have waited long enough.

2.2.4 Control Flow Summary of Scheduler

The scheduler iterates over the process table, skips suspended or completed processes, computes priority, and chooses the best one to run. It switches context using `swtch()` and updates runtime and profiling stats.

```
best->context_switches++;
if (best->response_time == -1)
    best->response_time = ticks - best->creation_time;

int start_ticks = ticks;
swtch(&(c->scheduler), best->context);
int end_ticks = ticks;

best->ticks_run += (end_ticks - start_ticks);
```

Processes spawned via `custom_fork()` start later and run for a specified time, enabling controlled experiments. They are awakened by calling `scheduler_start()`.

```
if (p->is_custom && p->start_later && p->state == SLEEPING)
    p->state = RUNNABLE;
```

3 Discussion on α and β Effects

The parameters α and β directly influence the dynamic priority calculation in the scheduler. A higher α value penalizes processes with longer run times, thus promoting short CPU-bound tasks. Conversely, a higher β favors processes that have been waiting longer, improving fairness for I/O-bound or delayed tasks.

Our profiler results show that:

- Increasing α leads to reduced average turnaround time (TAT) for short jobs but may increase waiting time (WT) for longer jobs.
- Increasing β significantly reduces WT and response time (RT) for processes that would otherwise starve, at the cost of scheduling longer-waiting processes more aggressively.

Careful tuning of α and β is required to balance throughput and fairness based on workload characteristics.

4 Conclusion

This assignment provided practical experience in modifying core components of the xv6 operating system. The implementation of signal handling enabled responsive process control through user interactions, while the scheduler enhancements introduced a more realistic model of CPU allocation using tunable dynamic priorities and profiling.

Key challenges included correctly capturing keyboard inputs, preserving process states during signal delivery, and designing a fair scheduling policy without introducing starvation. These tasks offered valuable insights into kernel-level process management and real-world scheduling strategies.

Future improvements could include adding support for multiple custom signal types, visualizing scheduling decisions over time, and experimenting with adaptive parameter tuning for α and β.

References

[1] Russ Cox, Frans Kaashoek, and Robert Morris, *xv6: a simple, Unix-like teaching operating system*, MIT, Revision 12, 2016.
`https://pdos.csail.mit.edu/6.828/2016/xv6/xv6-rev12.pdf`

[2] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, *Operating System Concepts*, 10th Edition, Wiley, 2018.

[3] Linux Programmer's Manual, *Signal(7) - Overview of signals*,
`https://man7.org/linux/man-pages/man7/signal.7.html`

[4] Wikipedia contributors, *CPU scheduling*, Wikipedia, The Free Encyclopedia,
`https://en.wikipedia.org/wiki/Scheduling_(computing)`

[5] John Kubiatowicz, *Process and Signal Control in Operating Systems*, UC Berkeley Lecture Notes.
`https://people.eecs.berkeley.edu/~kubitron/courses/cs162/`

[6] Andrew S. Tanenbaum and Herbert Bos, *Modern Operating Systems*, 4th Edition, Pearson, 2014.

[7] David A. Curry, *UNIX Systems Programming for SVR4*, O'Reilly Media, 1996. (Chapter 8: Signals)

Assignment Solution on

Real-Time Page Statistics and Swap Management in xv6 Kernel

Resource Management in Computer Systems (COL633) |Assignment-3(Part 1 & 2) |2024–2025

Submitted by

Arindam Sal
Entry Number: 2024JCS2041
Ayan Shil
Entry Number: 2024JCS2048

Submitted on: 30th April, 2025

Indian Institute of Technology Delhi

M.Tech in Cyber Security

Contents

1 Objective

The goal of this part of the assignment was to enhance the xv6 kernel with a signal-based handler that responds to the **Ctrl+I** key combination. The handler should:

- Detect the Ctrl+I signal during shell runtime.
- Print the number of user-space pages residing in RAM for each process.
- Filter processes that are in `SLEEPING`, `RUNNABLE`, or `RUNNING` state and have `pid >= 1`.
- Resume normal shell operation after executing the handler.

2 System Used

- **QEMU Emulator**: xv6 runs in qemu via `make qemu`.
- **xv6 Public Repository**: Modified to include Ctrl+I handler.
- **Linux Host OS**: Ubuntu 22.04 on x86-64.
- **SMP Enabled**: `-smp 2` used to simulate concurrency.

3 Design and Approach

The feature was implemented by hooking into the xv6 keyboard interrupt mechanism within the `consoleintr()` function in `console.c`. Special care was taken to:

- Avoid deadlocks by deferring the memory printing outside the spinlock (`cons.lock`).
- Use `p->sz` to compute the number of pages as `(sz + PGSIZE - 1) / PGSIZE`.
- Avoid printing for processes that were in `ZOMBIE` or `UNUSED` state.

4 Implementation Details

4.1 console.c - Detect Ctrl+I

```
case C('I'): // Ctrl+I doprintmem = 1; // defer printing outside lock break;
...  if(doprintmem)  print_log_("Ctrl+I is detected by xv6
n"); print_mem_stats();
```

4.2 proc.c - print_mem_stats()

This function is responsible for printing the number of user-space pages in RAM for each active process.

Here's how it works:

- It acquires the `ptable.lock` to safely iterate over all processes.
- It prints a header line: `PID NUM_PAGES`.
- Then, it traverses the global process table and filters only those processes that:
  - Have a `pid >= 1`
  - Are in the `SLEEPING`, `RUNNABLE`, or `RUNNING` state
- For each valid process, it calculates the number of user pages using the formula:

  ```
  (p->sz + PGSIZE - 1) / PGSIZE
  ```

  where `p->sz` is the size of the process memory in bytes.
- It then prints the result using a helper function `print_log_pid(pid, num_pages)`.
- Finally, it releases the process table lock.

This function is invoked when the user presses `Ctrl+I`, allowing the kernel to report current memory usage of active processes.

4.3 Custom Logging Function

```
void print_log_(const char *msg)  cprintf("
void print_log_pid(int pid, int pages)  cprintf("
```

5 Testing Strategy

The feature was validated using:

- Built-in commands like `stressfs` and `usertests`.
- Triggering `Ctrl+I` while multiple processes were active.
- Verifying no kernel panic due to deadlock by deferring output logic.
- Ensuring printed output matched the required format.

Sample Output

```
Ctrl+I is detected by xv6 PID NUM_PAGES 1 3 2 4 3 13 ...
```

6 Challenges Faced

- Initially experienced a kernel panic due to acquiring `cons.lock` while `print_mem_stats()` acquired `ptable.lock`.
- Solved by deferring execution logic using flags (`doprintmem`) outside the lock block.
- Another challenge was ensuring output formatting exactly as required, especially under concurrent process creation.

7 Conclusion

This assignment successfully introduced a kernel-level signal handler in xv6. The implementation carefully handled interrupt safety, synchronization, and memory computation. The work demonstrates:

- A clear understanding of process state management in xv6.
- Safe signal-based interrupt handling and lock synchronization.
- Real-time memory statistics printing from kernel space.

8 Part 2: Page Swapping and Adaptive Replacement in xv6

8.1 Objective

The second part of the assignment aimed to extend the xv6 kernel with a complete swapping system using disk-backed memory pages. The key objectives were:

- Modify xv6 to support page swapping using a designated disk region.
- Track and replace memory pages dynamically based on access patterns.
- Limit RAM usage to 4MB and evict pages when free memory drops below a threshold.
- Implement an **adaptive page replacement strategy** using tunable parameters `ALPHA` and `BETA`.
- Handle page faults and swap pages back into memory upon access.
- Ensure correctness using provided test program `memtest`.

8.2 System Setup

- xv6-public with QEMU and SMP enabled.
- Compilation parameters modified: `PHYSTOP = 0x400000` to restrict RAM.
- Disk layout altered to include 800 swap slots, each storing 1 page using 8 blocks.
- ALPHA and BETA parameters configured via Makefile.

8.3 Design Overview

8.3.1 Modified Disk Layout

The disk layout was updated to insert a dedicated swap area immediately after the superblock. Each swap slot consists of 8 disk blocks (512 bytes/block) to store a single 4KB page. A structure `swap_slot` was created with fields for page permissions and slot status:

```
struct swap_slot  int page_perm; // Page flags like PTE_U, PTE_W int is_free;
// Whether the slot is available ;
```

8.3.2 Swap Management

- Swapped-out pages are written directly to disk bypassing the log layer (`fs.c`).
- Swap slots are initialized at boot using `swap_init()` and tracked using a spinlock-protected array.
- Slot allocation uses `find_free_slot()`, and cleanup is triggered via `swap_cleanup()` during process exit.

8.3.3 Page Replacement Strategy

To maintain performance under limited memory, an adaptive policy was introduced:

- If free pages $\leq$ threshold, `check_and_swap()` is invoked.
- The victim process is chosen as the one with the highest number of resident pages (RSS).
- Among its pages, one with the PTE_A flag unset is selected as the victim.
- If no such page is found, the PTE_A bit is cleared on all pages and the scan is repeated.
- Threshold and swap count are updated dynamically using:

```
Th = Th * (1 - BETA/100)
Npg = min(LIMIT, Npg * (1 + ALPHA/100))
```

8.3.4 Swapping In and Out

- On allocation failure, `kalloc()` triggers swap-out attempts.
- `swappage_out()` selects a victim and writes the page to disk.
- PTE is updated to encode swap slot index, clearing PTE_P.
- On page fault (trap `T_PGFLT`), `swappage_in()` restores the page from swap.
- Page table entries are updated and RSS counters are adjusted.

8.4 Implementation Highlights

8.4.1 Key Snippet: Victim Page Selection

```
for (addr = 0; addr < KERNBASE; addr += PGSIZE)  entry = walkpgdir(pgdir,
(void*)addr, 0); if (entry && (*entry & PTE_P) && (*entry & PTE_U))  if
(!(*entry & PTE_A))  *va_out = addr; return PTE_ADDR(*entry);
```

This logic checks for unaccessed pages to minimize disruption. If not found, all pages are unmarked and the scan is repeated.

8.4.2 Modular Strategy

Helper abstractions were introduced:

- `attempt_single_swap()`: Encapsulates the logic for evicting a single page during a swap-out cycle.
- `resolve_swap_fault()`: Serves as a wrapper to handle page faults and initiate swap-in.
- `trigger_swap_policy()`: Abstracts `check_and_swap()` for cleaner usage in functions like `kalloc()`.

8.4.3 Trap Handler

```
if (trapframe->trapno == T_PGFLT)  uint fault_addr = rcr2(); if
(resolve_swap_fault(myproc()->pgdir, (void*)fault_addr) == 0) return;
```

The page fault handler calls into the swap system when a page is missing from memory.

8.5 Testing and Output

The kernel was tested using the provided `memtest` utility with expected output:

```
Current Threshold = 100, Swapping 4 pages Current Threshold = 90, Swapping 5
pages Current Threshold = 81, Swapping 6 pages ...  Memtest Passed
```

8.6 Challenges and Debugging

- Resolving page faults in a safe context to avoid `panic:  sched locks`.
- Avoiding nested lock conflicts between swap slots and page table locks.
- Correctly encoding and decoding PTE flags with swap slot index.
- Ensuring TLB flush consistency using `lcr3()`.
- Tracing and fixing double-swapping scenarios in `copyuvm()`.

8.7 Impact of α and β on Swapping Efficiency

The adaptive page replacement mechanism dynamically adjusts the number of pages to be swapped out (`Npg`) and the threshold for free pages (`Th`) using the parameters α and β, respectively.

- α (ALPHA): Controls the rate at which the number of pages to be swapped out increases. A higher α leads to more aggressive swapping in future iterations.
- β (BETA): Dictates the rate at which the memory pressure threshold `Th` decreases. A larger β value causes the threshold to reduce faster, thereby initiating swap-outs sooner in future.

This strategy ensures the system reacts based on recent memory demand. The update equations are:

$$\begin{aligned} \texttt{Th} &\leftarrow \texttt{Th} \cdot (1 - \tfrac{\beta}{100}) \\ \texttt{Npg} &\leftarrow \min(\texttt{LIMIT}, \texttt{Npg} \cdot (1 + \tfrac{\alpha}{100})) \end{aligned}$$

Efficiency Trade-offs

- **Low α, low β**: Results in slow adaptation, safer but less responsive under pressure.
- **High α, low β**: Increases swap-out intensity without increasing sensitivity—may overshoot memory needs.
- **Low α, high β**: Threshold drops quickly but not enough pages are evicted—might not prevent out-of-memory errors.
- **High α, high β**: Highly responsive but can lead to excessive page eviction, risking unnecessary I/O and performance degradation.

Optimal Choice

In practice, moderate values such as $\alpha = 25$, $\beta = 10$ provide a good balance—allowing the system to scale swap activity gradually while reacting to memory pressure in a timely manner. These values prevent frequent thrashing and ensure efficient use of swap space with minimal disk overhead.

8.8 Conclusion

This part added a working swapping mechanism with adaptive replacement to xv6. It simulated realistic memory pressure and enforced disciplined memory reuse. The implementation involved changes to memory allocators, page table logic, trap handling, and disk block management.

- Demonstrated deep understanding of virtual memory internals.
- Correctly handled concurrency and locking under swap pressure.
- Achieved expected test output and performance under constrained memory.