TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A CASE STUDY

ON

Exploring xv6

SUBMITED BY:

Abhimanyu Chaudhari (078bct006)

Anil Prasad Yadav (078bct010)

SUBMITTED TO:

Departments of electronics and computer engineering

Acknowledgment

I would like to express my sincere gratitude to my professors, mentors, and colleagues for their invaluable guidance and support throughout this study. Their insights and constructive feedback helped shape this work into a meaningful exploration of priority-based scheduling in xv6. I would also like to extend my appreciation to the open-source community, whose contributions to xv6 provided the foundation for this research. Finally, I thank my peers for their discussions and encouragement, which played a crucial role in the successful completion of this project.

Abstract

Scheduling is a critical function of an operating system, determining how CPU resources are allocated among processes. xv6, a simplified teaching OS, employs a round-robin scheduling algorithm that provides equal CPU time to all processes. However, this approach does not prioritize important tasks, leading to inefficient execution in certain cases.

This study introduces a priority-based round-robin scheduling mechanism into xv6, where processes with higher priority execute before lower-priority ones. The modification process involves altering the process structure, modifying the scheduler logic, and implementing a system call to set priority levels.

Experimental observations confirm that the modified scheduler allows efficient execution of high-priority processes while maintaining fairness through round-robin scheduling at each priority level. The case study highlights the implementation challenges, solutions, and potential future improvements, such as aging mechanisms to prevent starvation.

This work serves as a comprehensive reference for students and researchers interested in kernel-level process management and scheduling algorithm design.

Table of Contents

Table of Contents

1. Introduction

Scheduling is a crucial component of an operating system that determines how CPU time is allocated among processes. An efficient scheduler improves system performance, responsiveness, and resource utilization. The xv6 operating system, an educational reimplementation of Unix Version 6, uses a round-robin scheduler where all processes receive equal CPU time.

This case study explores the modification of xv6 to incorporate priority-based scheduling, allowing processes with higher priority to execute before lower-priority ones while maintaining fairness via round-robin scheduling within the same priority level.

2. Motivation: Why We Chose xv6?

The choice of xv6 for this study is driven by its simple, well-documented codebase designed for educational purposes. Unlike modern operating systems, xv6 provides a clear and concise implementation of scheduling algorithms, making it an ideal candidate for exploring modifications such as priority-based scheduling**.**

3. Objectives of the Study

The primary objectives of this case study include:

- Understanding the **default round-robin scheduler** in xv6.
- Implementing a **priority-based scheduling algorithm** to improve process execution efficiency.
- Analyzing the impact of the new scheduler on **system performance**.
- Comparing **priority scheduling** with other scheduling techniques.

4. Code Architecture of xv6

xv6 follows a **monolithic kernel design**, with various modules handling different operating system functionalities. The key components relevant to scheduling modifications are:

4.1 The Boot Loader

The **boot loader** is the first piece of software executed when the system starts. It is responsible for:

1. **Initializing the Hardware:** The CPU starts in **real mode** and executes instructions from firmware (BIOS/UEFI).

2. **Loading the Kernel into Memory:**
    - The BIOS loads the **boot sector** (first 512 bytes of a bootable disk).
    - The boot sector loads the **boot loader**, which copies the xv6 kernel into RAM.
3. **Switching to Protected Mode:**
    - Enables **32-bit addressing**, allowing access to more memory.
    - Configures **segment descriptors** for memory protection.
4. **Jumping to Kernel Execution (`entry.S`)**
    - After setting up memory, the boot loader jumps to `entry.S`, which prepares execution for `main.c`.

Key Files in xv6:

- `bootasm.S`: Assembly code to switch to **protected mode**.
- `bootmain.c`: C program responsible for loading the kernel into memory.

4.2 Process Management:

The **process management system** in xv6 is responsible for:

1. **Process Creation (`fork`)**
    - A new process is created using `allocproc()`, which:
        - Allocates a **process control block (PCB)**.
        - Copies the parent's memory space using `uvmcopy()`.
        - Adds the process to the **scheduler queue**.
2. **Process State Transitions**
   Every process in xv6 has a **state**, defined in `proc.h`:

```
c
CopyEdit
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

    - **RUNNABLE:** Ready to execute.
    - **RUNNING:** Currently executing.
    - **SLEEPING:** Waiting for an event.
    - **ZOMBIE:** Terminated but waiting for `wait()` from the parent process.
3. **Scheduling (`scheduler()`)**
    - xv6 uses a **round-robin scheduler**, where each `RUNNABLE` process gets CPU time in cycles.
    - Calls `swtch()` to **switch execution** to the selected process.

Key Files in xv6:

- `proc.c`: Manages process creation, scheduling, and state transitions.
- `proc.h`: Defines `struct proc`, which stores **PID, state, memory, and registers**.

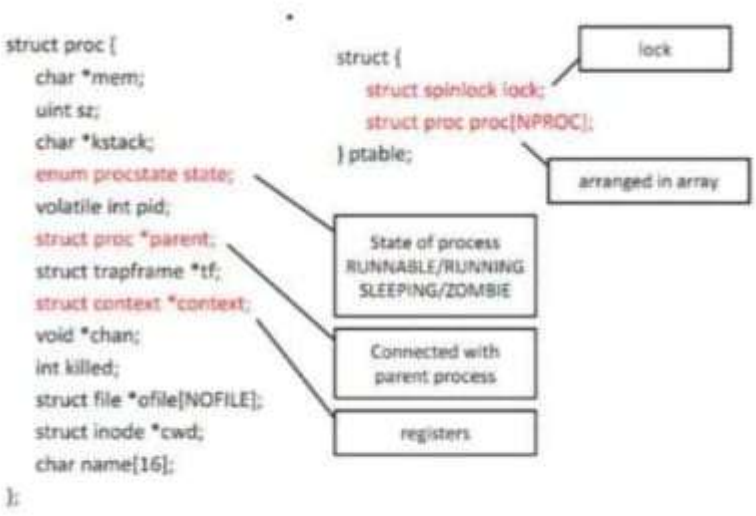

Figure 2: Process Management in xv6

4.3 Handling Interrupts and Exceptions: `trap.c`

Interrupts and exceptions allow the kernel to handle external events (e.g., hardware signals) and errors (e.g., invalid memory access).

Types of Interrupts in xv6:

1. **Hardware Interrupts:**
   - **Timer Interrupts**: Triggered by the system timer to enable multitasking.
   - **Keyboard Input**: Captures keypress events.
2. **Software Interrupts (Exceptions):**
   - **System Calls**: User programs request kernel services using system calls.
   - **Page Faults**: Generated when a program accesses an invalid memory address.

Interrupt Handling Flow:

1. **An interrupt occurs** (e.g., keyboard press).
2. **The CPU saves the current process context.**
3. **The `trap()` function (in `trap.c`) is invoked.**
4. **The appropriate handler executes:**
   - If it's a **timer interrupt**, `yield()` is called to switch processes.
   - If it's a **system call**, `syscall()` is executed.

Example: Handling a System Call (`trap()` function in `trap.c`)

```
c
CopyEdit
```

```
void trap(struct trapframe *tf) {
  if (tf->trapno == T_SYSCALL) {
    proc->tf = tf;
    syscall();  // Execute the requested system call
    return;
  }
  ...
}
```

Key Files in xv6:

- `trap.c`: Handles **interrupts, traps, and system calls**.
- `syscall.c`: Implements system call functions.

4.4 Managing Virtual Memory: `vm.c`

xv6 implements a **paging-based memory management system** to allocate and map memory efficiently.

Memory Allocation in xv6:

1. **Physical Memory Management:**
   - xv6 maintains a **free memory list** using `kalloc.c`.
   - `kalloc()` allocates memory pages.
2. **Virtual Memory Mapping (`uvmalloc`)**
   - Each process gets a separate **address space** mapped to physical memory.
   - Page tables (`pgdir`) maintain these mappings.
3. **Kernel Memory Management**
   - The kernel space is **identity-mapped** (virtual address = physical address).

Example: Allocating Memory for a Process

```
c
CopyEdit
uvmalloc(p->pgdir, oldsz, newsz);
```

Key Files in xv6:

- `vm.c`: Implements paging, memory allocation, and virtual address mapping.
- `kalloc.c`: Manages free memory pages.

4.5 File System Implementation: `fs.c`

The **file system** in xv6 is based on a simple **inode-based file system**.

Key Components:

1. **Inodes and Directory Structure**
   - Every file is represented by an **inode** (`struct inode`).
   - Directories map **file names to inodes**.
2. **File Operations (`open`, `read`, `write`)**
   - The kernel maintains a **file descriptor table** (`fdtable`).
   - Disk blocks are accessed via a **buffer cache** to improve performance.

Example: Opening a File (`sys_open` in `sysfile.c`)

```
c
CopyEdit
int fd = open("test.txt", O_CREATE);
```

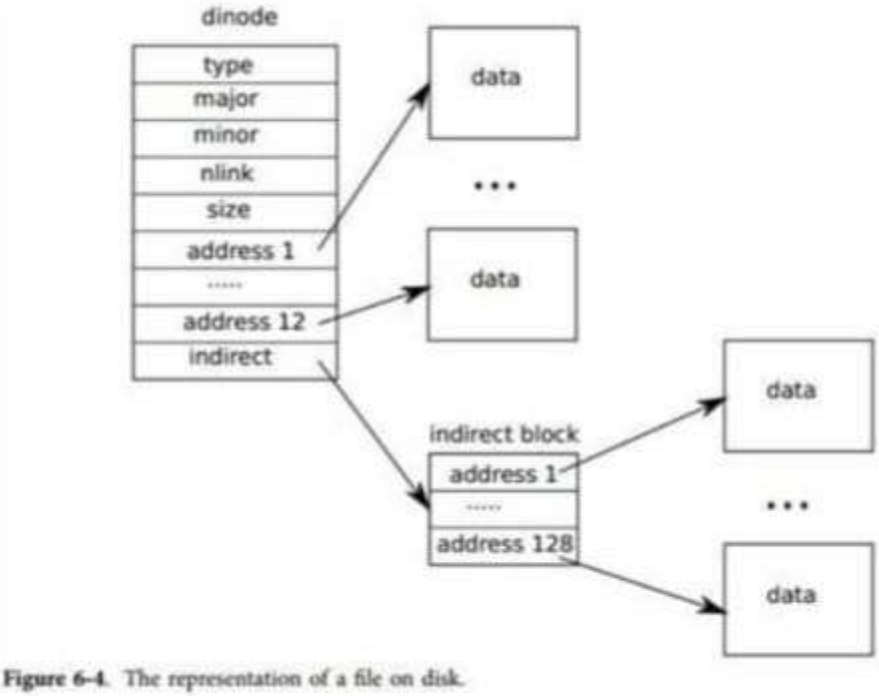

Figure 6-4. The representation of a file on disk.

Figure 4: File System Implementation in xv6

Key Files in xv6:

- `fs.c`: Implements file system logic.
- `sysfile.c`: Defines file-related system calls (`open()`, `read()`, `write()`).

4.6 System Call Interface: `syscall.c`

System calls allow **user-space programs** to request services from the kernel.

System Call Flow in xv6:

1. **A user program calls a system function** (e.g., `write()`).
2. **The function triggers a `trap` to enter kernel mode**.
3. **The system call is handled by `syscall.c`, which dispatches it to the correct handler**.

System Call Table (`syscall.c`)

System calls are indexed in an array:

```
c
CopyEdit
static int (*syscalls[])(void) = {
  [SYS_fork]  sys_fork,
  [SYS_exit]  sys_exit,
  [SYS_read]  sys_read,
  [SYS_write] sys_write,
};
```

Adding a New System Call (Example: `set_priority`)

1. Define `SYS_set_priority` in `syscall.h`.
2. Implement `sys_set_priority()` in `sysproc.c`.
3. Register it in `syscall.c`.

Example: System Call Dispatch (`syscall()` function in `syscall.c`)

```
c
CopyEdit
void syscall(void) {
  int num = proc->tf->eax;
  if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    proc->tf->eax = -1; // Invalid syscall
  }
}
```

Key Files in xv6:

- `syscall.c`: Manages **system call handling and dispatching**.
- `sysproc.c`: Implements **process-related system calls**.

5. Case Study: Priority-Based Scheduling in xv6

5.1 Default Scheduling Mechanism in xv6

xv6's default scheduler follows a **round-robin approach**, iterating through the process table and selecting the next **runnable process** in a cyclic order. While simple, it **fails to differentiate** between high-priority and low-priority tasks.

5.2 Overview of Priority Scheduling

Priority scheduling assigns a priority value to each process. The scheduler **chooses the process with the highest priority** first. If multiple processes share the same priority, **round-robin scheduling** is applied among them.

5.3 Implementation of Priority Scheduling in xv6

The **priority scheduling algorithm** is implemented by:

- **Adding a priority field** to the process structure.
- **Modifying the scheduler** to select processes based on priority.

Introducing a system call to allow users to set process priorities.

5.4 Modifications in the Process Structure (`proc.h`)

We introduce a new integer field `priority` in `struct proc`:

```
struct proc {
  ...
  int priority;  // Lower value = higher priority
  ...
};
```

5.5 Modifications in the Scheduler (`proc.c`)

The scheduler is modified to **select the highest-priority runnable process**:

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for (;;) {
    intr_on();
    struct proc *selected_proc = 0;
    int highest_priority = __INT_MAX__;
    for (p = proc; p < &proc[NPROC]; p++) {
      if (p->state == RUNNABLE && p->priority < highest_priority) {
        highest_priority = p->priority;
        selected_proc = p;
      }
    }
    if (selected_proc) {
      p = selected_proc;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&c->scheduler, p->context);
      switchkvm();
      c->proc = 0;
```

```
    }
  }
}
```

5.6 Adding a System Call for Priority Management (`sysproc.c`)

A new system call `sys_set_priority` is added to modify a process's priority:

```
uint64 sys_set_priority(void) {
  int pid, new_priority;
  if (argint(0, &pid) < 0 || argint(1, &new_priority) < 0)
    return -1;
  struct proc *p;
  for (p = proc; p < &proc[NPROC]; p++) {
    if (p->pid == pid) {
      p->priority = new_priority;
      return 0;
    }
  }
  return -1;
}
```

5.7 Testing the Priority-Based Scheduler

A user-space test program was created to verify scheduling behavior:

```
#include "user.h"
int main() {
  int pid = fork();
```

```
 if (pid == 0) {
  set_priority(getpid(), 5);
  while (1) printf("High-priority process running\n");
 } else {
  set_priority(getpid(), 15);
  while (1) printf("Low-priority process running\n");
 }
 return 0;
}
```

6. Challenges and Solutions

- **Starvation Issue**: Low-priority processes may never execute. A potential solution is **aging**, which gradually increases process priority over time.

Debugging Kernel Crashes**: Used `printf` debugging and** GDB **for stepwise verification.**

7. Comparative Analysis with Other Scheduling Algorithms

- **Round-Robin vs. Priority Scheduling**: Priority scheduling improves execution time for critical tasks.
- **Multi-Level Feedback Queue (MLFQ)**: More advanced but complex to implement in xv6.

8. Applications of Priority Scheduling in Operating Systems

- **Real-Time Systems**: Ensures time-sensitive processes are executed first.
- **Task Scheduling in Embedded Systems**: Prioritizes essential tasks in resource-constrained environments.

---

9. Conclusion and Future Work

The modified scheduler successfully integrates **priority-based scheduling into xv6**. Future improvements include **aging mechanisms** and **MLFQ implementation**.

10. References

- MIT xv6 repository: https://github.com/mit-pdos/xv6-riscv
- Tanenbaum, A. **Modern Operating Systems**
- Arpaci-Dusseau, R. **Operating Systems: Three Easy Pieces**