J. Integr. Sci. Technol. 2025, 13(3), 1053

Journal of Integrated
SCIENCE & TECHNOLOGY

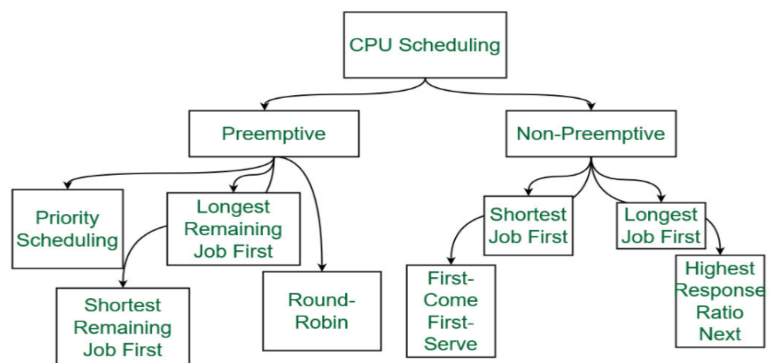
CPU scheduling algorithms performance analysis in the RISC-V xv6 operating system environment

Madan H T[1], Manjunatha H M[2*], Nagaraja Rao Pradeep[3] , Vidyashankar M[4]

[1]NITTE (Deemed to be University), Department of Electronics and Communication (Advanced Communication Technology), NMAM Institute of Technology, Nitte, Karkala Taluk, Udupi, 574110, Karnataka, India. [2]Electrical and Electronics Engineering, Bapuji Institute of Engineering and Technology, Davanagere-577004, Karnataka, India. [3]Electrical and Electronics Engineering, Sri Siddhartha Institute of Technology, Sri Siddhartha Academy of Higher Education, Tumakuru-572105, Karnataka, India. [4]Electrical and Electronics Engineering, J N N College of Engineering, Shimoga, 577205, Karnataka, INDIA

Submitted on: 03-Sep-2024, Accepted and Published on: 10-Dec-2024 | Article

ABSTRACT



Process scheduling is a crucial element of operating systems, which has a significant impact on system performance and the efficient use of resources. This paper includes a study of scheduling algorithms implemented in xv6, which is a Unix-like operating system designed specifically for educational purposes. This paper examines the specificities and operational attributes of Round Robin, First-Come-First-Serve, and Priority-Based Scheduling. The functionality of xv6 is enhanced by integrating system call tracing, a debugging tool that intercepts and logs system calls invoked by programs during execution. The strace system function can be implemented in a comprehensive manner, allowing for selective tracing depending on a mask given by the user. The procdump software offers detailed information on the present condition of running processes, encompassing their execution durations, waiting durations, and queue durations. Benchmarking and performance analysis provide a quantitative evaluation of the trade-offs linked to each scheduling policy. The results indicate that the First-Come-First-Serve scheduler has the lowest average waiting time, while the Round Robin and Priority-Based Scheduling schedulers have larger waiting times due to the additional overhead of preemption and priority-based selection.

Keywords: Process scheduling, xv6 Operating System, RISC-V Architecture, Round Robin First-Come-First-Serve, Priority-Based Scheduling.

INTRODUCTION

Process scheduling is an essential element in contemporary operating systems as it determines how CPU time is distributed among competing processes. Efficient scheduling algorithms are crucial for optimizing resource use, promoting fairness, and enhancing responsiveness in multitasking situations. The choice of a suitable scheduling policy has a substantial impact on system performance, throughput, turnaround time, and overall user experience.

The primary objective of this work is to implement and assess scheduling algorithms in xv6, which is a Unix-like operating system specifically, developed for educational use. xv6 is a C-based re-implementation of the original Unix V6 operating system, specifically designed for the Reduced Instruction Set Computer V (RISC-V) instruction set architecture.[1] RISC-V is an unencumbered and complimentary set of instructions that has garnered substantial popularity in educational and scientific circles.[2,3]

Round Robin (RR)[4] is a preemptive, time-sharing strategy that allocates set time intervals to processes in a circular queue. When the allotted time for a process to run is over, it is interrupted, and the next process in line is given an opportunity to execute. The objective of this algorithm is to ensure equitable distribution of

*Corresponding Author: Dr. Manjunatha H.M., Assistant Professor, Dept. of EEE, BIET, Davanagere, Karnataka, India
Tel: +919741677070; Email: Manjunath.hm1986@bietdvg.edu

Cite as: *J. Integr. Sci. Technol., 2025, 13(3), 1053.*
URN:NBN:sciencein.jist.2025.v13.1053
DOI:10.62110/sciencein.jist.2025.v13.1053

©Authors CC4-NC-ND, ScienceIN https://pubs.thesciencein.org/jist

CPU resources and prevent any individual process from dominating the CPU for a prolonged duration.

First-Come-First-Serve (FCFS)[5] scheduling algorithm is a method where tasks are executed in the order they arrive.[3] A non-preemptive algorithm that executes processes in the order of their arrival in the ready queue. Once a process commences execution, it persists until it either ends or becomes blocked, ensuring uninterrupted execution. Although the implementation of FCFS is straightforward, it can result in starvation if the CPU is consistently occupied by long-running processes.

Priority-Based Scheduling (PBS)[6] is a scheduling approach that incorporates priorities into the decision-making process. Activities with elevated priorities are allocated CPU time ahead of activities with lower priorities. PBS has the capability to preempt a lower-priority process that is already executing, in order to allow a higher-priority process to take its place. This technique is very valuable in real-time systems when specific processes have stringent temporal limitations.

Studying scheduling methods in xv6 offers useful insights for students and researchers. The streamlined yet authentic setting facilitates practical investigation of process management and scheduling principles. By incorporating different algorithms into xv6, one can get a more profound comprehension of their principles, trade-offs, and practical consequences, effectively connecting theory and practice.

The research incorporates a customized bootloader into xv6, hence improving the system's initialization procedure. This bootloader initializes the stack, configures UART connectivity, alerts all active cores, and transfers control to the kernel logic. Implementing a custom bootloader offers a deeper understanding of the initial phases of system starting and the many nuances of hardware-software interaction.

Assessing these algorithms in the context of xv6 RISC-V provides valuable information about their advantages, disadvantages, and compromises in terms of performance measures such as average waiting time, turnaround time, and fairness.[7] This evaluation entails the use of thorough benchmarking methodologies and meticulously planned experiments to evaluate the effects of different scheduling policies on system performance across a range of workload conditions.

The study encompasses an examination of current literature about process scheduling methods and their implementations across different operating systems. This review provides a contextual analysis of the research contributions and emphasizes the distinctive characteristics of investigating scheduling inside the xv6 system. This text encompasses the fundamental principles, real-world applications, and latest developments in scheduling algorithms across several operating systems.

This paper extensively examines the implementation specifics, which encompass alterations made to the xv6 kernel, the utilization of data structures to support various scheduling policies, and the incorporation of new system calls to enable the selection and adjustment of scheduling algorithms. The experimental setup is thoroughly explained, providing a comprehensive overview of the hardware and software configurations employed for testing and assessment purposes.

RELATED WORK

Process scheduling algorithms and their implementations represent an active research area across various operating systems and educational platforms. This section reviews relevant literature and foundational work, providing context for the current research.

The xv6 operating system, developed by Cox et al. [1] at MIT, serves as the basis for this research. Xv6, a re-implementation of Unix V6 in C for RISC-V architecture, provides a simplified environment for exploring OS concepts like process management, scheduling, and memory management. Its design philosophy emphasizes clarity and simplicity, making it an ideal platform for educational purposes and experimental implementations of operating system concepts.

The xv6 codebase and accompanying lectures[8] offered valuable insights into the existing RR scheduling implementation. This understanding supported the addition of new scheduling policies and modifications to the kernel's process management subsystem. The codebase's modular structure facilitated the integration of additional scheduling algorithms without compromising the overall system integrity.

Process Scheduling Algorithms lecturers[9] and R. H. Arpaci-Dusseau et.al., (2014)[10] provide comprehensive theoretical background on scheduling algorithms. These resources cover RR, FCFS, and PBS, detailing their operational principles, advantages, and potential drawbacks in various computing scenarios. The material also discusses concepts such as preemption, time quantum selection, and starvation prevention, which are crucial for understanding the trade-offs between different scheduling policies.

Bibu and Nwankwo's (2019)[11] comparative analysis of FCFS and Shortest-Job-First (SJF) algorithms informed the understanding of FCFS characteristics in the xv6 context. Their work highlights the simplicity and fairness of FCFS, as well as its potential for convoy effect in certain workloads. This analysis provided valuable insights for predicting and interpreting the performance of FCFS when implemented in xv6.

The RISC-V instruction set architecture, central to this work, is thoroughly documented in the RISC-V instruction set manuals [12,13] and the RISC-V Reader by Patterson and Waterman (2017)[14]. These resources detail the RISC-V instruction formats, addressing modes, and system-level operations, which are essential for low-level operating system development. Understanding the RISC-V architecture is crucial for optimizing scheduler performance and ensuring correct interaction between the operating system and the underlying hardware.

Inspiration for implementing scheduling algorithms in educational operating systems comes from studies like Shree's exploration of priority scheduling and system calls in xv6.[15] This work demonstrates practical approaches to modifying xv6's process management subsystem, including the addition of new system calls and the integration of priority-based scheduling. Such studies provide valuable insights into the challenges and best practices for extending xv6's functionality while maintaining its educational value.

Vutukuru (2024)[16] presented on operating systems at the Indian Institute of Technology Bombay, which covered various

scheduling algorithms and their implementations. These lectures complemented our understanding of the theoretical concepts and practical considerations involved in implementing scheduling policies. Chandan Shrivastava (2021)[17] tweaked-xv6 is a valuable resource that complements this work on implementing and evaluating scheduling algorithms in the xv6 operating system environment. This contains modifications and enhancements to the original xv6 codebase, providing additional features and functionalities relevant to this research. Engel's presentation (2023)[18] on porting the xv6 operating system to the Nezha D1 RISC-V board highlights the process of adapting and running xv6 on a specific RISC-V hardware platform, which can be valuable for understanding the practical considerations involved in deploying and testing scheduling algorithms in a RISC-V environment

The expanded research context encompasses not only the theoretical foundations of process scheduling but also practical considerations for implementation in a teaching operating system. This comprehensive approach aims to bridge the gap between theoretical concepts and their real-world application in operating system design and development.

This article seeks to enhance the current understanding of scheduling algorithm performance and trade-offs in the context of the xv6 RISC-V operating system environment. It builds upon previous work in the xv6 operating system, process scheduling algorithms, and the RISC-V architecture. The study commences by doing a comprehensive analysis of the xv6 codebase, with a specific emphasis on the pre-existing process management and scheduling methods. Establishes a strong basis for the implementation of novel scheduling algorithms and the modification of the kernel to accommodate multiple scheduling policies. The study subsequently applies three separate scheduling algorithms: RR, FCFS, and PBS. Every implementation requires meticulous deliberation of data structures, administration of process state, and integration with the preexisting components of the xv6 kernel. The work also involves the deployment of a bespoke bootloader for xv6. This bootloader improves the system setup process by correctly configuring the execution environment prior to handing over control to the kernel. The development of the bootloader offers a detailed understanding of the low-level interaction between hardware and software that occurs during the initial phases of system startup. The results of this extensive investigation offer significant insights into the behavior and performance traits of several scheduling algorithms within the framework of a basic, educational operating system operating on RISC-V hardware. These observations can provide guidance for the development of future operating systems, specifically for embedded systems and educational platforms that utilize the RISC-V architecture.

In the subsequent section of this study, Section 2 outlines the system model for this research centers on the xv6 operating system running on RISC-V architecture. The implementation involves modifying the xv6 kernel to support multiple scheduling algorithms is discussed in Section 3. Section 4, evaluate the performance of the implemented RR, FCFS, and PBS algorithms. The conclusions are reported in Section 5.

SYSTEM MODEL

The system model for this research centers on the xv6 operating system running on RISC-V architecture. It encompasses the implementation of multiple scheduling algorithms (RR, FCFS, PBS) within xv6, along with system call tracing, process state monitoring, and a custom bootloader for enhanced initialization.

A. RISC-V Architecture

The RISC-V is an open-source instruction set architecture (ISA) [12,13] designed to support a wide range of computing systems, from embedded devices to high-performance servers. It is a load/store architecture, meaning that all instructions operate on data in registers, and memory is accessed through explicit load and store instructions.

The RISC-V instruction format is divided into several types, as shown in Figure 1. The R-type instructions are used for register-to-register operations, while the I-type instructions handle register-immediate operations. The S-type instructions are used for memory store operations, and the U-type instructions are primarily used for upper immediate values and PC-relative addressing.

The RISC-V architecture supports multiple privilege levels , as illustrated in Figure 2.[19] The User Mode is the least privileged level, where user applications run. The Supervisor Mode is responsible for executing operating system kernel tasks, such as system call handling and memory management. The Machine Mode is the most privileged level, reserved for low-level hardware control and initialization.

The transitions between these privilege levels are facilitated by instructions like ecall (environment call), sret (supervisor return), and mret (machine return), ensuring proper isolation and security.

B. xv6 Kernel

xv6 is a re-implementation of the classic Unix V6 operating system, written in C and tailored for the RISC-V instruction set architecture. It is a simplified yet functional operating system designed for educational purposes, allowing students and researchers to explore and experiment with various operating system concepts, including process management, memory management, file systems, and device drivers. Figure 3 illustrates the high-level architecture of the xv6 operating system, highlighting its key components and their interactions.[8]

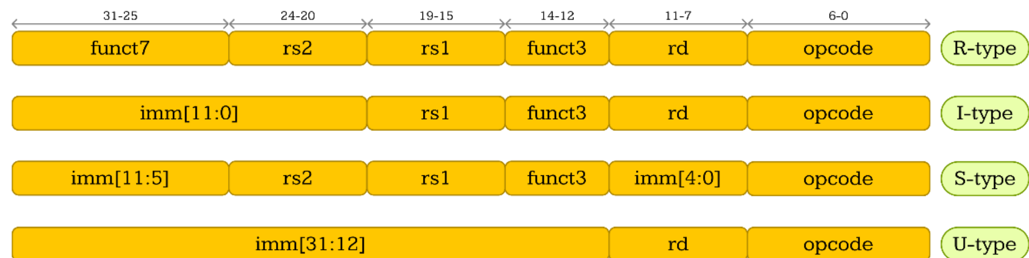


Figure 1. RISC-V Instruction Format

The xv6 kernel is monolithic, meaning that all kernel services and device drivers are integrated into a single executable image. The User Space contains user processes, while the Kernel Space comprises various components responsible for managing system resources and facilitating interactions with hardware. The user processes interact with the kernel through system calls, enabling controlled access to kernel services and resources.

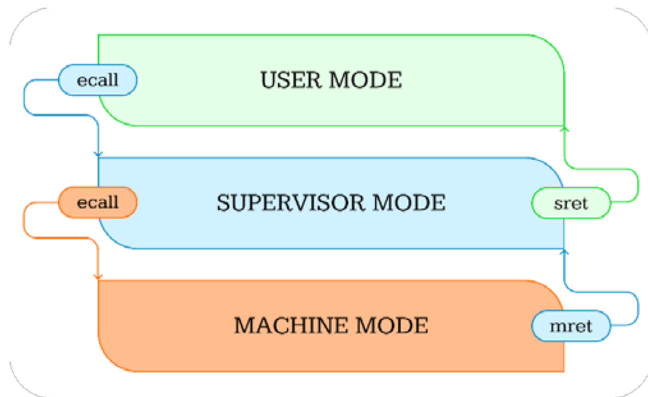


Figure 2. RISC-V Privilege Levels

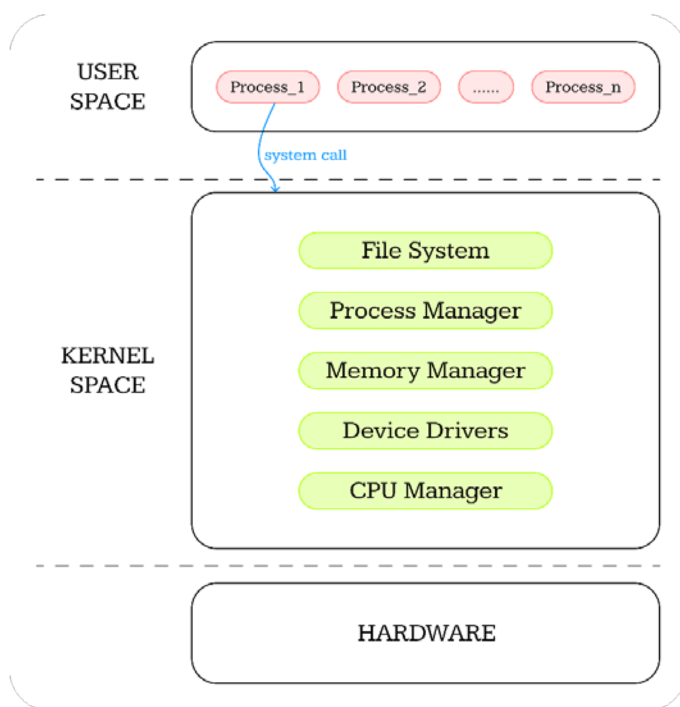


Figure 3. xv6 Kernel Architecture

The File System component manages the hierarchical file system and provides file-related operations. The Process Manager is responsible for creating, scheduling, and managing processes. The Memory Manager handles memory allocation and management tasks. The device drivers facilitate communication with various hardware devices, such as disks, terminals, and networks. Finally, the CPU Manager is responsible for implementing and managing the scheduling algorithms, which are the focus of this research work.

By leveraging the RISC-V architecture and the xv6 operating system, this research aims to contribute a comprehensive analysis of scheduling algorithm performance and trade-offs in a resource-constrained, educational environment. The next section will delve into the specific implementation employed in this study, including the experimental setup, scheduling algorithm implementations, and performance evaluation techniques.

C. Custom Bootloader

Our implementation includes a custom bootloader that performs several crucial initialization tasks before handing control to the kernel. The bootloader's main functions are:

- Setting up the stack: The bootloader initializes the stack for each CPU core, allocating 4096 bytes of stack space per core.
- Configuring UART for communication: It sets up the Universal Asynchronous Receiver/Transmitter (UART) for serial communication , enabling debugging output and system messages.20
- Notifying all running cores: The bootloader announces the booting process for each hardware thread (HART), flushing this information to the terminal.
- Jumping to kernel logic: After initialization, the bootloader transfers control to the kernel's entry point in start.c. This custom bootloader enhances the system's initialization process and provides better visibility into the early boot stages.

IMPLEMENTATION

The implementation involves modifying the xv6 kernel to support multiple scheduling algorithms (RR, FCFS, PBS) and adding new system calls for algorithm selection and process monitoring. It also includes developing a custom bootloader and implementing utilities like 'strace' and 'procdump' for system analysis and debugging.

A. Experimental Setup

In order to conduct a thorough evaluation of the different CPU scheduling algorithms, this research project has utilized the xv6 operating system environment. xv6 is a simplified yet realistic Unix-like system that is specifically designed for educational purposes. The xv6 codebase underwent modifications and extensions to incorporate the implementation of the RR, FCFS, and PBS scheduling algorithms.

The experiments were conducted on a RISC-V-based system emulated using the QEMU virtualization software, version 9.1.0. The virtual machine had been configured with 256 MB of RAM and a single-core RISC-V processor running at 1 GHz, faithfully reproducing a standard embedded or resource-constrained environment. The xv6 operating system, version 6.2, was compiled and installed on the virtual machine, providing a controlled and reproducible environment for the experiments. [21]

In order to facilitate debugging and performance analysis, two crucial features were implemented into the xv6 codebase. Initially, the strace system call was implemented to enable selective tracing of system calls invoked by processes during their execution. This powerful tool allowed for the monitoring and recording of system

call activity, offering valuable insights into process behavior and system interactions.

Additionally, the procdump utility was developed, which serves as a valuable debugging aid by providing a comprehensive snapshot of active processes in the system. The procdump output includes detailed information like process IDs, priorities, states, run times, wait times, and queue residencies for the MLFQ.[22] This utility was extremely helpful in examining the system's state and confirming the proper functioning of the implemented scheduling algorithms.

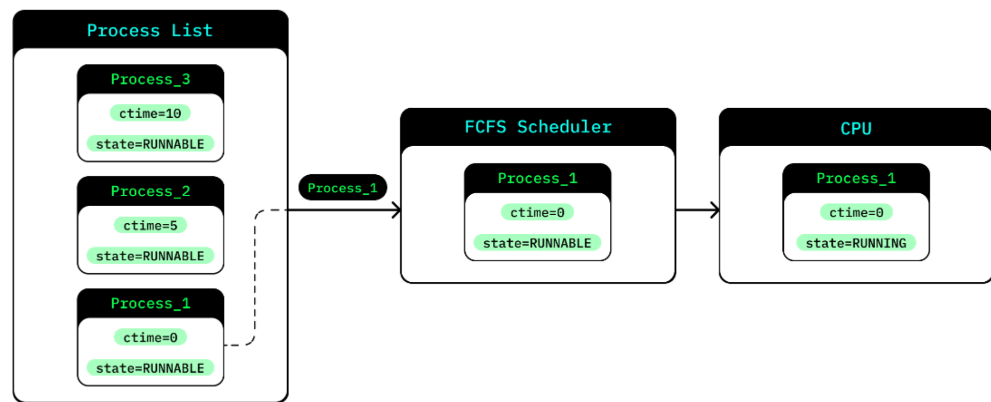


Figure 4. First Come First Serve

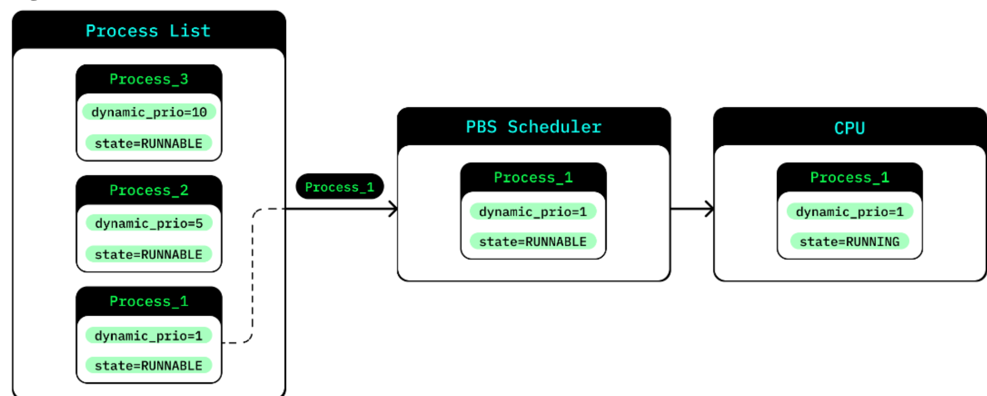


Figure 5. Priority Based Scheduling

B. System Call Tracing and Procdump Implementation

System Call Tracing (strace) was implemented in xv6 to enable selective tracing of system calls during process execution. This feature allows users to specify a mask indicating which system calls should be traced. The implementation involved several steps. First, a mask field was added to the proc structure in proc.h to store the trace mask for each process. The fork function in proc.c was then modified to copy the trace mask from the parent to the child process, ensuring that tracing behavior is inherited by child processes.The sys_trace function was implemented in sysproc.c to handle the strace system call, retrieving and storing the trace mask in the current process's mask field. Additionally, the syscall function in syscall.c was updated to check the trace mask and print trace information for specified system calls. To facilitate trace information printing, syscall_names and syscall_argc arrays were created to store system call names and argument counts.A user-space program, strace.c, was developed to provide a command-line interface for the strace system call, allowing users to specify the trace mask and the command to be traced. Finally, entries were added to user.h, usys.pl, and syscall.h to declare and register the strace system call within the xv6 operating system.

The procdump utility in xv6 was enhanced to provide additional process information relevant to the scheduling algorithms under study. The enhanced procdump output now includes the process ID (PID), process state (e.g., RUNNING, RUNNABLE, SLEEPING), process name, total runtime (rtime), total waiting time (wtime), and the number of times the process was scheduled (nrun). Additionally, for the PBS algorithm, the output includes the current dynamic priority. These enhancements make the procdump utility a powerful tool for inspecting the system's state and verifying the correct operation of the implemented scheduling algorithms.

C. Scheduling Algorithm Implementations

The xv6 operating system comes with a default implementation of the RR scheduling algorithm. This preemptive, time-sharing approach assigns a fixed time slice to each process in a circular queue. When a process's time slice expires, it is preempted, and the next process in the queue is given a chance to execute. This algorithm aims to provide fair CPU sharing and prevent any single process from monopolizing the CPU for an extended period.

The FCFS scheduling algorithm was implemented in the xv6 kernel. FCFS is a non-preemptive approach that executes processes in the order they arrive in the ready queue. Once a process begins executing, it carries on until it terminates or blocks, ensuring uninterrupted execution.[23,24]

In figure 4, it is demonstrated that the FCFS implementation in xv6 requires traversing the process list to locate the process with the lowest creation time (ctime), which indicates the tick number at which the process was created. The selected process is then executed until it is no longer in need of CPU time. In order to avoid preemption, the clock interrupt handling mechanism was disabled in the trap.c file. The core logic of the FCFS scheduler involves acquiring locks on processes, checking their states, and selecting the process with the minimum creation time for execution.

A non-preemptive priority-based scheduler has been implemented to select the process with the highest priority for execution. If two or more processes have the same priority, the tie is broken based on the number of times the process has been scheduled (n_run). In the event of a tie, the start-time of the process

(ctime) is utilized to resolve it, ensuring that processes with lower start times are scheduled later.

The static priority (SP) of a process can vary from 0 to 100, with lower values indicating higher priorities. The default priority has been set to 60. The system call, setpriority(), has been implemented to enable users to modify the static priority of a process. The dynamic priority (DP) is derived from the static priority and a niceness value, which quantifies the amount of time the process spent in a sleeping state.

As depicted in figure 5, the core logic of the PBS scheduler. The process list is iterated through, and the dynamic priority (DP) is calculated based on the static priority and niceness value. Then, the process with the minimum dynamic priority is selected for execution, taking into account the number of times the process has been scheduled (n_run) and the creation time (ctime) to break ties.

In both FCFS and PBS implementations, the process struct in proc.h was adjusted to include extra details about the process. This included information like creation time, run time, wait time, and priority. The allocproc() function in proc.c has been updated to initialize these values when a new process is created. The scheduling algorithms were implemented in the scheduler() function in proc.c, triggered using preprocessor directives.[25]

While implementing these scheduling algorithms, challenges were encountered related to process synchronization and ensuring the correctness of the scheduling decisions. The challenges were overcome by acquiring locks on the process structs before modifying their states or accessing sensitive information. This approach avoided race conditions and guaranteed the integrity of the scheduling process.

Results and Discussion

Round Robin (RR), First-Come-First-Serve (FCFS), and Priority-Based Scheduling (PBS) Algorithm were evaluated by using standardized experimental configuration. These three algorithms were chosen to analyze the scheduling mechanism because of their fundamental difference in scheduling and resource handling strategy. Each algorithm was tested under the consistent workload in order to reason about them properly. The workload consists of equally divided I/O-bound and CPU-bound processes and the evaluation was done by two metrics such as Average Waiting Time and Average Runtime. Both were measured in terms of CPU ticks. The Average Waiting Time is the amount of time that the process spend in the ready queue before it starts executing and Average Running Time is the amount of time at which the process gets executed before its completion and interruption.

A. First-Come-First-Serve Scheduling

First-Come-First-Serve algorithm is a simple scheduling algorithm that follows a very sequential execution paradigm. In the performed study, to evaluate the performance of FCFS different configuration workloads were provided. The investigation includes different workloads comprising 10 processes, equally distributed between I/O-bound and CPU-bound categories. Table 1 illustrates experimental configuration highlighting the equal process distribution and arrival times.

To understand the FCFS scheduling algorithm's performance, a detailed breakdown of individual process readings provides good insights. Table 2 presents a granular breakdown of process execution parameters across the experimental workload.

First, the initial I/O-bound processes (PIDs 4, 5, and 6) experienced minimal wait times, completing within the first 31 CPU ticks. This demonstrates the algorithm's initial efficiency with short-duration tasks. Conversely, subsequent I/O-bound and CPU-bound processes encountered progressively increasing wait times, ranging from 31 to 91 CPU ticks.

The execution times remained remarkably consistent across all processes, with each process consuming approximately 30-31 CPU ticks. However, the wait times exhibited significant variation, directly correlated with the process's position in the execution queue. This variation underscores the FCFS algorithm's inherent sensitivity to process arrival order and duration.

The most pronounced impact is evident in the CPU-bound processes (PIDs 9, 10, 11, 12, and 13), which experienced substantial wait times ranging from 61 to 91 CPU ticks. A critical observation emerged regarding the "convoy effect," where longer-running or earlier-arrived processes significantly delay subsequent task executions.

The performance evaluation revealed nuanced insights into FCFS scheduling dynamics. The aggregate metrics, presented in Table 3, provide a quantitative representation of the algorithm's operational characteristics.

B. Round Robin Scheduling

To evaluate RR performance, workloads similar to the FCFS study were used, comprising 10 processes distributed equally between I/O-bound and CPU-bound categories. Table 1 describes the experimental configuration.

The Round-Robin scheduling simulation revealed distinct execution patterns across I/O-bound and CPU-bound processes, as summarized in Table 2. The data demonstrates a balance between runtime efficiency and wait times for I/O-bound tasks, while CPU-bound processes experienced predictable outcomes due to their uninterrupted quantum utilization.

Processes 4, 5, and 6, all I/O-bound, started execution simultaneously at time 0. They completed at tick 49 with runtimes of 25, 15, and 17 ticks, respectively. Their wait times varied between 21 and 32 ticks, reflecting the algorithm's time-sharing behavior, where shorter bursts of computation and sleep cycles contributed to moderate queue delays.

Processes 7 and 8, also I/O-bound, entered the system at tick 1. They completed slightly later at ticks 49 and 50, respectively. Process 7 exhibited a runtime of 23 ticks and a wait time of 25 ticks, whereas Process 8 required 27 ticks of runtime and experienced a shorter wait time of 22 ticks. The slight stagger in completion times illustrates the effects of quantum-based interruptions and the advantage of starting just after the initial batch.

CPU-bound processes (PIDs 9 through 13) followed a notably different pattern. All these processes began execution at tick 1 and completed only after their extensive computational requirements were satisfied. Each CPU-bound process maintained a runtime of 0 to 1 tick per quantum, with total runtimes stretching to 201 ticks

due to the repetitive allocation of quantum slices. Wait times for these processes were remarkably consistent, averaging around 200 ticks, highlighting the algorithm's cyclical fairness but also its inefficiency in prioritizing longer tasks. Aggregate performance metrics derived from the experiment are summarized in Table 4.

C. Priority-Based Scheduling

The implementation and analysis of Priority-Based Scheduling (PBS) in XV6 reveals sophisticated process management capabilities and efficient resource utilization. As illustrated in Table 5/4, our experimental setup consisted of 10 processes: 5 I/O-bound and 5 CPU-bound processes, with priorities ranging from 60 to 85.

As shown in Table 2, the I/O process behavior exhibited early completion times ranging from 38 to 76 ticks, with a consistent runtime of 37 to 38 ticks. Notably, the initial processes, specifically PIDs 4 to 6, experienced zero wait time, while the later processes, PIDs 7 and 8, encountered wait times of 37 ticks. In terms of CPU process behavior, all processes completed at a total of 201 ticks, demonstrating minimal runtime between 0 to 1 tick but experiencing high wait times ranging from 200 to 201 ticks. This resulted in a consistent completion pattern across the board. Regarding priority impact, I/O processes with priorities between 80 and 85 were completed first, despite their lower priority compared to CPU processes that had a priority of 60 and showed longer wait times. Overall, the system effectively maintained a priority-based execution approach throughout the processes.

Table 1. Experimental Workload Configuration

Process Category	Quantity	Arrival Characteristics	Sleep Duration	Computational Profile
I/O-bound	5	Simultaneous	200 CPU ticks	Periodic interruption
CPU-bound	5	Simultaneous	N/A	Continuous execution

Table 2. Detailed Process Execution Characteristics

	Process ID	4	5	6	7	8	9	10	11	12	13
	Process Type	I/O-bound					CPU-bound				
FCFS	Completion Time	31	31	31	61	62	62	91	92	93	121
	Runtime	30	30	30	30	31	31	30	30	31	30
	Wait Time	1	1	1	31	31	31	61	62	62	91
RR	Completion Time	49	49	49	49	50	201				
	Runtime	25	15	17	23	27	0-1				
	Wait Time	21	32	31	25	22	200-201				
PBS	Completion Time	38	38	39	75	76	201				
	Runtime	37	37	38	37	38	0-1				
	Wait Time	0	0	0	37	37	200-201				

Table 3. FCFS Performance Metrics

Metric	Quantitative Value	Description
Total Runtime	303 CPU ticks	Comprehensive system utilization
Total Wait Time	372 CPU ticks	Aggregate queuing duration
Average Runtime	30 CPU ticks	Per-process computational allocation
Average Wait Time	37 CPU ticks	Average queue retention period

Table 4. RR Performance Metrics

Metric	Quantitative Value	Description
Total Runtime	109 CPU ticks	Aggregate CPU usage across processes
Total Wait Time	1132 CPU ticks	Time spent in the ready queue
Average Runtime	10 CPU ticks	CPU usage per process
Average Wait Time	113 CPU ticks	Queue duration per process

Table 5. Experimental Workload Configuration

Process Category	Quantity	Priority	Sleep Duration	Computational Profile
I/O-bound	5	80-85	200 CPU ticks	Periodic interruption
CPU-bound	5	60	N/A	Continuous execution

Table 6. PBS Performance Metrics

Metric	Quantitative Value	Description
Total Runtime	189 CPU ticks	Aggregate CPU usage across processes
Total Wait Time	1075 CPU ticks	Time spent in the ready queue
Average Runtime	18 CPU ticks	CPU usage per process
Average Wait Time	107 CPU ticks	Queue duration per process

Our analysis revealed an intriguing phenomenon where I/O-bound processes, despite their lower priority (80-85), completed before CPU-bound processes with higher priority (60). This behavior stems from the fundamental difference in resource utilization patterns. I/O processes voluntarily release the CPU during their sleep cycles, requiring minimal CPU time for completion. Conversely, CPU-bound processes require continuous processor access, making their completion time more dependent on

priority-based scheduling decisions. The overall performance metrics obtained from the experiment are presented in Table 6.

The comparative analysis of FCFS, RR, and PBS uncovers distinct behavior in process management and system resource allocation. FCFS achieved lowest total wait time of 372 CPU ticks along with the efficient execution model but it was suffered from the "convoy effect" where longer process significantly delayed the execution of subsequent processes. RR showed a balanced timesharing approach with total wait time of 1132 ticks but offered a fair time for execution, which is beneficial for interactive systems. PBS balanced between RR and FCFS sharing 1075 total wait time ticks and demonstrated a sophisticated priority management functionality. However, these tests were tested in emulation environment using QEMU. QEMU's inherent limitations is, it cannot mirror the real hardware behavior. Emulations are not designed to do the fine-grained things like hardware do such as context-switching, interrupt handling etc. Real-world factors like, interaction with cache, interrupt arrival time introduces a subtle effect in scheduling behaviors of these algorithms.

This work enhances the understanding of CPU scheduling algorithms with a novel performance analysis within the context of the RISC-V xv6 operating system. A comprehensive review of the FCFS, RR, and PBS algorithms in a developing open-source framework provides insights beyond conventional x86-based reviews. The work connects theory with evidence, thereby setting up a crucial methodological framework for future investigations on the optimization of scheduling in open-source architecture, where the technological relevance of RISC-V architectures continues to grow.

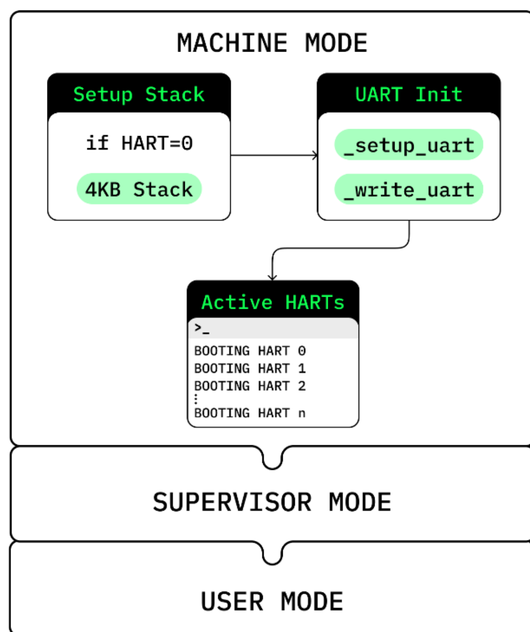


Figure 6. Custom Bootloader

D. Custom Bootloader Implementation

The custom bootloader was implemented in assembly language, tailored for the RISC-V architecture as shown in Figure 6. Key components of the bootloader include:

- Stack setup: The bootloader calculates and sets the stack pointer (sp) for each CPU core based on its hart ID.
- UART configuration: It disables interrupts on the UART and sets the output to 8 bits.
- Core announcement: Each core announces itself using the UART, printing "Booting HART x" where x is the core number.
- Synchronization: A simple locking mechanism ensures that cores announce themselves in order.
- Kernel handoff: After initialization, the bootloader jumps to the main kernel entry point.

Conclusion

This research provides a comprehensive analysis of process scheduling algorithms within the xv6 operating system on RISC-V architecture. By implementing and evaluating RR, FCFS, and PBS algorithms, along with system call tracing and process monitoring utilities, the study offers valuable insights into the performance trade-offs and behavioral characteristics of these scheduling policies in a simplified yet realistic environment. From the results it is observed that, FCFS demonstrated the lowest average waiting time, making it suitable for workloads where minimizing waiting time is crucial. RR and PBS offered better responsiveness but showed higher average waiting times due to preemption and prioritization overhead. The implementation of system call tracing, an enhanced procdump utility, and a custom bootloader improved debugging capabilities and system initialization. Future work could focus on optimizing PBS, exploring hybrid scheduling approaches, and extending support for real-time and multi-core scheduling. Investigating power-aware algorithms and applying machine learning techniques to predict workload patterns could lead to more efficient and adaptive operating system behavior. Enhancing the custom bootloader with advanced features like multi-stage booting or secure boot integration could further improve system security and flexibility.

Conflict of Interest Statement

Authors declare that there is no conflict of interest for this study

References

1. R. Cox, M.F. Kaashoek, and R.T. Morris, "Xv6, a Simple Unix-Like Teaching Operating System," Massachusetts Institute of Technology, Cambridge, MA, USA, 2021. Accessed on: April 10, 2024. [Online]. Available: https://pdos.csail.mit.edu/6.828/2021/xv6.html
2. Q. Zhang, J. Qiao, Q. Meng, Y. Chen. Automatic kernel code synthesis and verification. *Comput. Secur.* **2020**, 91, 101733.
3. G. Liu, J. Huang, X. Liu. An OS Kernel Based on RISC-V Architecture. In *Communications in Computer and Information Science*; Springer Nature Singapore, Singapore, **2024**; Vol. 1899, pp 3–16.
4. Y. Eni, S. Greenberg, Y. Ben-Shimol. Efficient Hint-Based Event (EHE) Issue Scheduling for Hardware Multithreaded RISC-V Pipeline. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2022**, 69 (2), 735–745.
5. C. Wulf, M. Willig, D. Goehringer. RTOS-supported low power scheduling of periodic hardware tasks in flash-based FPGAs. *Microprocess. Microsyst.* **2022**, 92, 104566.
6. L. Han. Resource Scheduling Algorithm for Heterogeneous High-Performance Clusters. In *Communications in Computer and Information Science*; Springer Nature Singapore, Singapore, **2022**; Vol. 1714 CCIS, pp 307–321.

7. R. Lowe. A Flexible and Broad Operating System Project. In *Proceedings of the 2024 ACM Southeast Conference, ACMSE 2024*; **2024**; pp 27–34.
8. "The xv6 Kernel," YouTube, Feb.3,2022 [Video file]. Available: https://www.youtube.com/playlist?list=PLbtzT1TYeoMhTPzyTZboW_j7TPAnjv9XB. [Accessed: Apr. 10, 2024].
9. OsLecturesForAll, "Operating Systems - NPTEL IITD," YouTube, Apr.6,2017 [Video file]. Available: https://www.youtube.com/playlist?list=PLsylUObW5M3CAGT6OdubyH6FztKfJCcFB.
10. R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "Operating Systems: Three Easy Pieces," [Chapter: Scheduling Introduction], Arpaci-Dusseau Books, 2014.
11. G. Dadik Bibu; Gloria, C. Nwankwo. Comparative Analysis Between First-Come-First-Serve (Fcfs) and Shortest-Job-First (Sjf) Scheduling Algorithms. *Int. J. Comput. Sci. Mob. Comput.* **2019**, 8 (5), 176–181.
12. Waterman, Andrew, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanovic. "The RISC-V instruction set manual volume II: Privileged architecture version 1.7." EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-49 (2015).
13. Andrew Waterman, Krste Asanovi´c and SiFive Inc., "The RISC-V instruction set manual Volume I: unprivileged ISA." https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf
14. Patterson, D, Waterman, A. The RISC-V Reader: An Open Architecture Atlas. Strawberry Canyon LLC: San Francisco, CA, USA, 2017.
15. H. Shree, "Xv6 Implementing PS, Nice System Calls, and Priority Scheduling," Medium, 2020. Accessed on: April 10, 2024. [Online]. Available: https://medium.com/@harshalshree03/xv6-implementing-ps-nice-system-calls-and-priority-scheduling-b12fa10494e4.
16. M. Vutukuru, "Lectures on Operating Systems," Indian Institute of Technology Bombay, 2018. Accessed on: April 10, 2024. [Online]. Available: https://www.cse.iitb.ac.in/~mythili/os/.
17. Chandan Shrivastava, 2021. tweaked-xv6. GitHub. https://github.com/chandan-shrivastava/tweaked-xv6. [Accessed: Apr. 10, 2024].
18. M. Engel, "Porting the xv6 OS to the Nezha D1 RISC-V Board," 2023. [Online]. Available: https://www.uni-bamberg.de/fileadmin/sysnap/slides/xv6-riscv.pdf. [Accessed: Apr. 10, 2024].
19. D. Mangum, "RISC-V Bytes: Privilege Levels,"2023. [Online]. Available: https://danielmangum.com/posts/risc-v-bytes-privilege-levels/. [Accessed: Jul. 13, 2024].
20. D. Mangum, "RISC-V Bytes: Accessing Pinecil UART with Picoprobe," 2023. [Online]. Available: https://danielmangum.com/posts/risc-v-bytes-accessing-pinecil-uart-picoprobe/. [Accessed: Jul. 13, 2024].
21. Atish Patra and Anup Patel "RISC-V Boot Flow," RISC-V International, Dec. 2019. [Online]. Available: https://riscv.org/wp-content/uploads/2019/12/Summit_bootflow.pdf. [Accessed: Jul. 13, 2024].
22. G. J. Popek, "CS537 Project 2b," University of Wisconsin-Madison, [Online]. Available: https://pages.cs.wisc.edu/~gerald/cs537/Fall17/projects/p2b.html. [Accessed: Jul. 13, 2024].
23. M. Fontana, "xv6-scheduling," GitHub repository, [Online]. Available: https://github.com/marf/xv6-scheduling. [Accessed: Jul. 13, 2024].
24. Muraleedharan, "enhanced-xv6-riscv," GitHub repository, [Online]. Available: https://github.com/coniferousdyer/enhanced-xv6-riscv. [Accessed: Jul.13, 2024].
25. Cloud-V, "Developing xv6 Operating System with Cloud-V: Testimonial 1," [Online]. Available: https://cloud-v.co/blog/risc-v-1/developing-xv6-operating-system-with-cloud-v-testimonial-1. [Accessed: Jul. 13, 2024].