

Process Management

From Simulation to Implementation

COSC519 – Operating Systems

Dr. Ramesh Karne

Towson University

Fall 2010

GROUP 1

Chuck Smith

Jeremy Dixon

Ron Erdman

Deema Alsekait

Dean Monostori



Table of Contents

Section 1 - Introduction	3
Section 2 – Group Description	3
Section 3 – Overall Description	3
Section 4 – System Requirements	5
Section 5 – Design and Implementation Constraints	5
Section 6 – User Interface:	6
Section 7 – Simulator Structure	7
Section 8 – Goals of the simulation:	8
Section 7 – Scheduling Methodology:	9



Section 1 - Introduction

PROCESS MANAGEMENT FROM SIMULATION TO IMPLEMENTATION

Modern day operating systems are required to manage a wide variety of things. One of the primary responsibilities in these systems includes process management. Processes are instances of programs or applications that are currently running or executed. The operating system is responsible for allocating processor time to run these processes. To help examine and better understand the process management functions, this team completed a multi-stage project which included the analysis, planning, development and implementation of a process management simulation. This effort also included a thoughtful analysis of the results, and attempt to identify, understand, and/or modify the process management function within an existing, open-source operating system.

Section 2 – Group Description

GROUP 1 THE PROJECT TEAM

The following list details the members of our team and the description of their responsibilities as defined as part of this project.

Jeremy Dixon: Initial proposal, Requirements Specifications, Presentation

Dean Monostori: Final Requirements Spec, Initial Simulation Design

Ron Erdman: Final Proposal, Final Simulation Design, Initial Simulation Coding

Deema AlSekait: Final Simulation Testing, Initial Observations & Conclusion (O&C)

Chuck Smith – Final O&C, Compare to Existing Open Source OS

The Entire Team: Final Project Report

Section 3 – Overall Description

SIMULATION DEVELOPMENT PROCESS

In order to successfully simulate the process management aspects of the

operating system, several software development steps were completed. First, a clear and concise set of requirements were generated in order to ensure the simulation is complete. After the requirements were created and analyzed, the simulator design was documented. The design document explains the software solutions and the required algorithm components. Standard outputs from this stage of the development commonly included abstractions, software architecture, data structures, and modularity. After the design of the simulator was completed, the implementation of the system began. The implementation is the step where the actual code was generated and the software was programmed. With the completion of the programming, the testing step commenced. Additional requirements modifications, design changes, or programming were required due to testing results (Summerville, 2006).

DESIGN CONSIDERATIONS

There are several aspects of processes were addressed as part of the simulator. First, the simulator has to establish a concept of process state. These states include new, running, waiting, ready, or terminated. Figure 1 shows a diagram of process states.

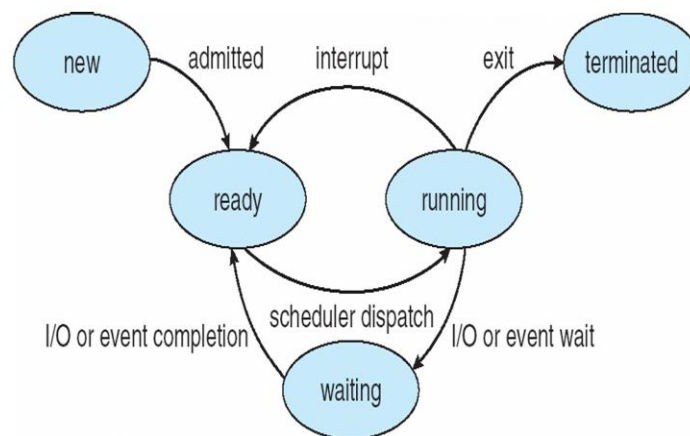



Figure 1. Diagram of Process States (Silberschatz et al. 2008)

Secondly, the process management simulator is required to manage the process control block, the ready queue, and the scheduler. As part of the processes creation, there will also be simulated forking and process termination considerations (Silberschatz et al. 2008).



Process management implementation has to be especially considerate of inter-process communication and synchronization issues that may arise due to resource limitations or competition. The simulator will manage these limitations as required.

Section 4 – System Requirements

- 3.1. The system shall be run using Windows XP, Windows 7 or Linux.
- 3.2. The system shall be compiled using a standard C++ compiler.
- 3.3. The system shall be coded using Visual Studio 2008.

Section 5 – Design and Implementation Constraints

- 4.1. The simulator shall be coded using C++ or a derivative thereof.
- 4.2. The simulator shall simulate the process management of an operating system.
- 4.3. The simulator shall not create actual processes using the fork command.
- 4.4. The simulator shall simulate the creation of processes.
- 4.5. The simulated processes will be considered ready to process.
- 4.6. The simulated processes will not require I/O for functionality.
- 4.7. The simulated processes will be scheduled using one or Shortest Job First (SJF), Round-Robin (RR), or Preemptive Priority.
- 4.8. The simulator shall start with the ready Q at time 0.
- 4.9. The simulator shall display a line of output for process creation.
- 4.10. The simulator shall display a line of output for the context switch.
- 4.11. The simulator shall display a line of output for the first CPU usage for each process (PID, initial wait time).
- 4.12. The simulator shall display a line of output for the process termination (PID, total time, total wait time).
- 4.13. The simulator shall display the current elapsed time.
- 4.14. The simulator shall display, when the simulation is finished running, the minimum, average, and maximum turnaround time.
- 4.15. The simulator shall display, when the simulation is finished running, the minimum, average, and maximum wait times.
- 4.16. The simulator shall display all output to a maximum of three decimal places.

- 4.17. The simulator shall request the number of processes to simulate.
- 4.18. The simulator shall request the type of scheduler the simulator shall use.
- 4.19. The simulator shall randomly create the processes.

TECHNICAL DETAILS

Language:	C++
Platform:	Windows 7 (Simulation) Linux (open source O/S Investigation)
Measurable:	Rates (Number of Processes/ Second), including: Process Creation Rate Process Queuing Rate Process Execution Rate Variables & Constraints, including: Process Size/Duration (modeled as delays) Starting Configuration (empty queue, full queue) Hardware in Use Process Priority

Section 6 – User Interface:

The purpose of this section is to describe the Simulator's User Interface, delineate the navigation of the system, the system requirements, and to demonstrate a non-functioning prototype of the system through scenarios.

Figure 2 below will explain process data-flow in the system

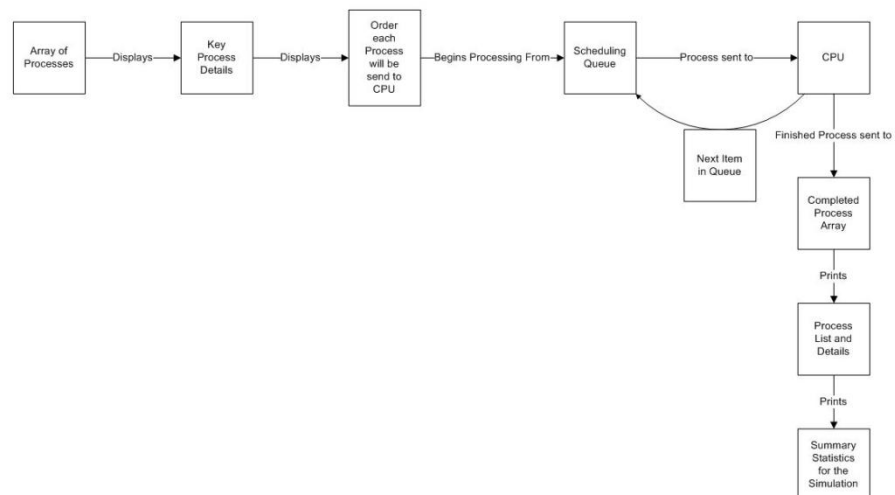


FIGURE 2 – PROCESS FLOW DESIGN

Section 7 – Simulator Structure

The Simulator component allows for several key functionalities relating to the user objects. The major functionalities include the inputs below.

Simulator's Outputs:

- Login to the System
- Logout of the System
- Display process creation
- Display context switch
- Create a processes
- Display the first CPU usage (PID, initial wait time) for each process
- Display process termination (PID, total time, and total wait time).
- Provide the minimum, average, and maximum turnaround time and wait times
- Display all outputs

However, in order for the simulator to provide us with all the information listed above it requires the following inputs from the user:

User Inputs:

- Number of process to simulate
- Type of scheduler to use

Figure 3 below explains how the user inputs the commands according to the simulator.

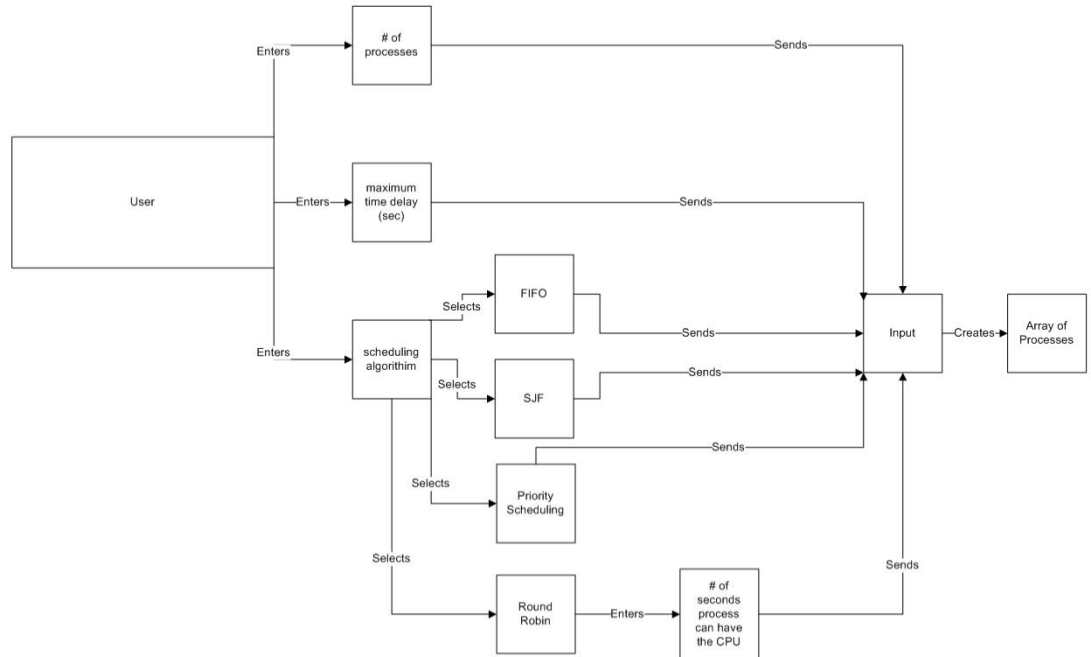


Figure 3 – Simulation Design Flow

SIMULATOR RESTRICTIONS

The simulator limits the number of users on a machine to one while displaying the simulator screen. Furthermore, the simulator will only perform on a ##### of processes.

Section 8 – Goals of the simulation:

Process management is very important in a system. Our goal is to demonstrate the efficiency or lack of using different scheduling types such as, RR, SJF, FCFS and preemptive priority.

COMPONENT TEMPLATE DESCRIPTION

NOTE: This section is the main focus in version 2.0 of the SDS, the detailed design. This section will provide most of the basis for implementing the product.

Identification	The unique name for the component and the location of the component in the system.
Scheduler	A process scheduler capable of switching between 4 scheduler algorithms housed within the simulator
Simulator	The system used to test and evaluate process generation, scheduling and completion. Encompasses the whole system
Random Process Generator	The component of the simulator responsible for generating random processes Housed within the simulator.
Interfaces	All user input is made via CLI
Data	All output is displayed as line(s) of data on the screen of the PC


Section 7 – Scheduling Methodology:

FIFO (First In First Out) which is non pre-emptive tends to favor processor-bound over I/O bound processes, this will result in inefficient use of both the processor and the I/O devices as whenever a processor-bound process is running, the I/O events will be idle even if there is potential work for them to do (Aber, 2002).

RR (Round Robin): RR is sometimes referred to as "time- slicing" (Charles H.Sauer et al, Computer systems performance modeling) .With this pre-emptive algorithm, clock interrupts are generated at periodic interval. When the interrupt occurs, the currently running process is placed in the ready queue and the most ready job is selected on a first come first serve basis. The process is given a time slice before being pre-empted.

SJF/SPN (Shortest Job First/Shortest Process Next): This is another approach to reducing the bias in favoring of long processes inherent in FIFO. This is a Non pre-emptive policy in which the process with the shortest processing time is run next. Short processes jump to the head of the queue past longer jobs (Bin Muhammad, 2010).

Preemptive Priority (PP) A preemptive priority algorithm will preempt the CPU if the priority of the newly arrival process is higher than the priority of the currently running process.



Command Line Interface (CLI) a user interface common to MS-DOS computers. The user sees the command line on the monitor and a prompt that is waiting to accept instructions from the user. The user types in the command, the computer acts on that command and then issues a new prompt for the next instruction from the user.

Each of these algorithms has distinctive advantages and disadvantages. The main areas of observation that allow for the efficiency of the algorithms to be measured consist of the following:

- CPU utilization – This can range from 0 to 100 percent. It is desirable to have a higher percent of CPU utilization, because it indicates that the hardware is not being wasted.
- Throughput – The number of processes completed per unit of time is the throughput.


This is a measurement of the work being done by the CPU. It is not always a good indicator, however, because it depends directly on the length of the processes being executed.

- Turnaround time – This measures the amount of time it takes between the submission of a process and the time the process completes its execution. It is measured for each process and takes all the time the process spends waiting and executing into account.
- Waiting time – This is a good measure of how efficiently the scheduling algorithm is allocating the CPU to ready processes. It measures the sum of the time periods a process spends waiting in the ready queue.
- Response time – This is measured only in interactive systems, and is the amount of time it takes to start responding to a request. The speed of the output device usually limits the response time.

Comparison To Other Schedulers:

THE UNIX SYSTEM

Here we will review the schedulers of Open Sources operating systems namely



Linux and BSD to explore how scheduling is implemented in a real world scenario. Both Linux and BSD are based upon the Unix System which has its roots in the MULTICS operating system. MULTICS, a collaborative effort between Bell Labs, General Electric, and Massachusetts Institute of Technology, was a research project designed to prove the feasibility of a general purpose time sharing system.

At the time most operating systems were batch oriented, and MULTICS was a dramatic departure from this paradigm. Although the first implementation of The Unix System in 1970 was single user, by 1974 it was described as "The Unix Time-Sharing System." As a time-sharing system, the goal of the scheduler was to provide quick response time to interactive programs and fair distribution of CPU to processor intensive processes. The Unix System never has actually distinguished interactive processes; instead, I/O intensive processes have always been assumed to be interactive. To accomplish this goal, The Unix System, described in K. Thompson, "UNIX Implementation", "The Bell System Technical Journal," vol 57, No. 6, July-August 1978, pg. 1931, implemented a dynamic priority based scheduling system with two classes of processes. System or kernel processes were given one set of priorities, user processes given a lower set of priorities. This ensured that system processes would always preempt user processes. System process priorities are fixed; however, user process priorities are dynamically adjusted based upon how much CPU time the process has received. Processes using a lot of CPU time have their priorities lowered while system using little CPU time have their priorities raised. This ensures that I/O bound processes assumed to be interactive will always have a higher priority improving interactive response time, and that even processes with very low priorities will eventually increase in priority over time to prevent starvation. Since all of this dynamic adjustment of priority is expensive, it is only done once a second and the most of the time the scheduler only picks the job of the ready queue with the highest priority (Thompson, 1978).

THE BERKELEY SOFTWARE DISTRIBUTION

The Unix System quickly found its way into the University of California Berkeley (UCB) and other academic institutions. UCB added things like a visual editor, virtual memory, and TCP/IP Networking. UCB released the source code for their networking implementation which began the reference implementation of TCP/IP. Eventually UCB release 4.4-Lite2 in 1995 completely free of proprietary code from AT&T. This became the basis for many modern BSD

implementations including NetBSD, FreeBSD, and MAC OSX. The scheduling goals of BSD Unix were the same as the Unix System. To improve efficiency BSD implemented a ready queue for each priority. Were executed in a round robin fashion at each priority level until that queue is empty then it moves on to the next lower priority level. A processes priority is dynamic. And computed using the following formula:

$$p_{usrpri} = PUSER \cdot \left[\frac{p_{estcpu}}{4} \right] \cdot 2^{p_{nice}}$$


where PUSER is the value of the process when not in kernel mode, p_estcpu is an estimation of the previous user of the CPU, and p_nice is a user-specified priority value. Back in 1974 a second was a very short period of time by 1995 a second became a very long period of time. BSD recalculated priorities at the end of every time quantum which is defined based upon the system clock (Giza, 1987).

MINIX

Minix, or mini-Unix, was created in 1987 by Andrew S. Tanenbaum for use in the Operating Systems class he taught at Vrije Universiteit in Amsterdam. It was originally released with his text book, Operating Systems: Design and Implementation. It was a complete rewrite that was system-call compatible with Seventh Edition Unix (Tanenbaum, 1987). It was specifically designed to be simple and run on the IBM PC and IBM PC/AT microcomputers. Minix implemented round robin scheduling with three queues: the user queue, the server queue, and the task queue. The scheduler was implemented with two functions sched() and pick_proc(). Sched() rotated the current process to the end of the current queue. Pick_proc() selected the first task in the first available queue.

LINUX

In 1991 Linus Benedict Torvalds having worked with MINIX started working on Linux originally to learn more about the Intel 386 processor. Initially the scheduler was fairly simple. At the beginning of every quantum the sys_setpriority() function would go through the ready queue adjusting the priorities of each process implementing the policies in use since mid seventies of improving the priority of I/O bound processes and degrading the priority of CPU bound processes. During the time quantum, the schedule() function walks through the ready queue selecting the highest priority process to run next. One very interesting thing about the Linux implementation is that the priority



becomes the time slice for the process for that particular time quantum. When it is has executed for the time specified by its priority, it is preempted and the priority is degraded. Over the years the scheduler has grown dramatically adding support real time processes and multiple scheduling policies either first in first out, or round robin. The ready queue has been converted into a sorted linked list reducing the scheduler from an $O(n)$ algorithm to an $O(1)$ algorithm (Bovet, 2000).

REFERENCES

- Aber, Jennifer B. (2002). Process Scheduling – Scheduling Algorithm. Retrieved on 12/8/2010 from <http://homepages.uel.ac.uk/u0214248/>
- Bim Muhammad, Rashid. (2010). Operating Systems – Scheduling Algorithms. Retrieved on 12/8/2010 from <http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/fcfsSchedule.htm>
- Bovet, Daniel P. and Cesati, Marco (2000) *Understanding the Linux Kernel*, Retrieved on 12/12/2010 from <http://oreilly.com/catalog/linuxkernel/chapter/ch10.html>
- Giza, Jon P. (1998) *The FreeBSD Process Scheduler*, Retrieved on 12/12/2010 from http://www.thehackerademy.net/madchat/ebooks/sched/FreeBSD_the_FreeBSD_process_scheduler.pdf
- Silberschatz, Abraham, Galvin, Peter B., and Gagne, Greg. 2008. *Operating Systems Concepts, 8th Edition*. Wiley Publishing. 978-0470128725
- Summerville, Ian. 2006. *Software Engineering, 8th Edition*. Addison-Wesley. 978-0321313799
- Thompson, Ken. (1978) "UNIX Implementation", "The Bell System Technical Journal," vol 57, No. 6, July-August 1978, pg. 1931
- Tanenbaum, Andrew S. (1987) *Operating Systems: Design and Implementation*. Prentice-Hall 013601959