ENHANCEMENT TO XOS OPERATING SYSTEM AND XFS FILE SYSTEM

A THESIS
Submitted by

In partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

Under the guidance of
DR. K MURALIKRISHNAN

DEPARTMENT OF COMPUTER ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY CALICUT
NIT CAMPUS PO, CALICUT
KERALA, INDIA 673601

May 11, 2015

ACKNOWLEDGEMENTS

Your acknowledgements

NAME

DECLARATION

"I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text".

Place:
Date:

Signature :
Name :
Reg.No:

CERTIFICATE

*This is to certify that the thesis entitled: "***ENHANCEMENT TO XOS OPERATING SYSTEM AND XFS FILE SYSTEM***" submitted by Sri/Smt/Ms to National Institute of Technology Calicut towards partial fulfillment of the requirements for the award of Degree of Bachelor of Technology in Computer Science Engineering is a bonafide record of the work carried out by him/her under my/our supervision and guidance.*

Signed by Thesis Supervisor(s) with name(s) and date

Place:
Date:

Signature of Head of the Department

Office Seal

Contents

Chapter

Abstract

Abstract here. Abstract should not exceed one page. if@

Tables

Figures

Figure

Chapter 1

Problem Definition

This project aims to modify and update the XOS operating sytem by

- Re-engineering the eXpFS system.
- Including inter process communication
- Redesigning process model
- Introducing shared memory model
- Adding more system calls thus broadening the features

The updated version is named as eXpOS/ Experimantal Operating System. Thus, adding the above features, eXpOS would become more advanced and closer to operating systems that are available in the market.

Chapter 2

Introduction

Project eXpOS or experimental Operating System is a educational platform to develop an operating system. Several instructional operating systems like Nachos, Pintos, GeekOS etc. have been developed by various universities for this purpose. Nachos has been one of the most popular instructional operating systems available and is being used in many institutes across the world. XOS, the previous version of eXpOS, was such a tool that helps undergraduate computer science students acquire an elementary understanding of the practical aspects of an operating system.

XOS had features like multiprogramming, process management, a primitive filesystem and virtual memory. In order to incorporate simplicity, OS features like interprocess communication, device management, file caching, file permissions etc. had been excluded from XOS. In the present version, eXpOS, some of these features such as inter process communication, asynchronous disk access, heap memory, etc had been included to make the student more proficient in Operating System concepts. The basic Machine model consists of memory, disk and the CPU. eXpOS includes specification for a set of system calls, the scheduler, the exception handler, device drivers etc. Further hardware support required includes the timer, disk controller, Input-output system etc.

The primary components of the project include a simulated machine hardware (XSM), file system (eXpFS) and the operating system (eXpOS). No code base

for the operating system is provided and the operating system is completely implemented by the student. Apart from the primary components, various tools are provided as part of the development environment. They include languages like Experimental Programming Language (ExpL) and System Programmer's Language (SPL) and their cross compilers to XSM instruction set, XSM debugger, and a UNIX-XFS interface to transfer files between a UNIX machine and the XFS disk (the eXpFS disk is itself a UNIX file). Generic hardware model is described below.

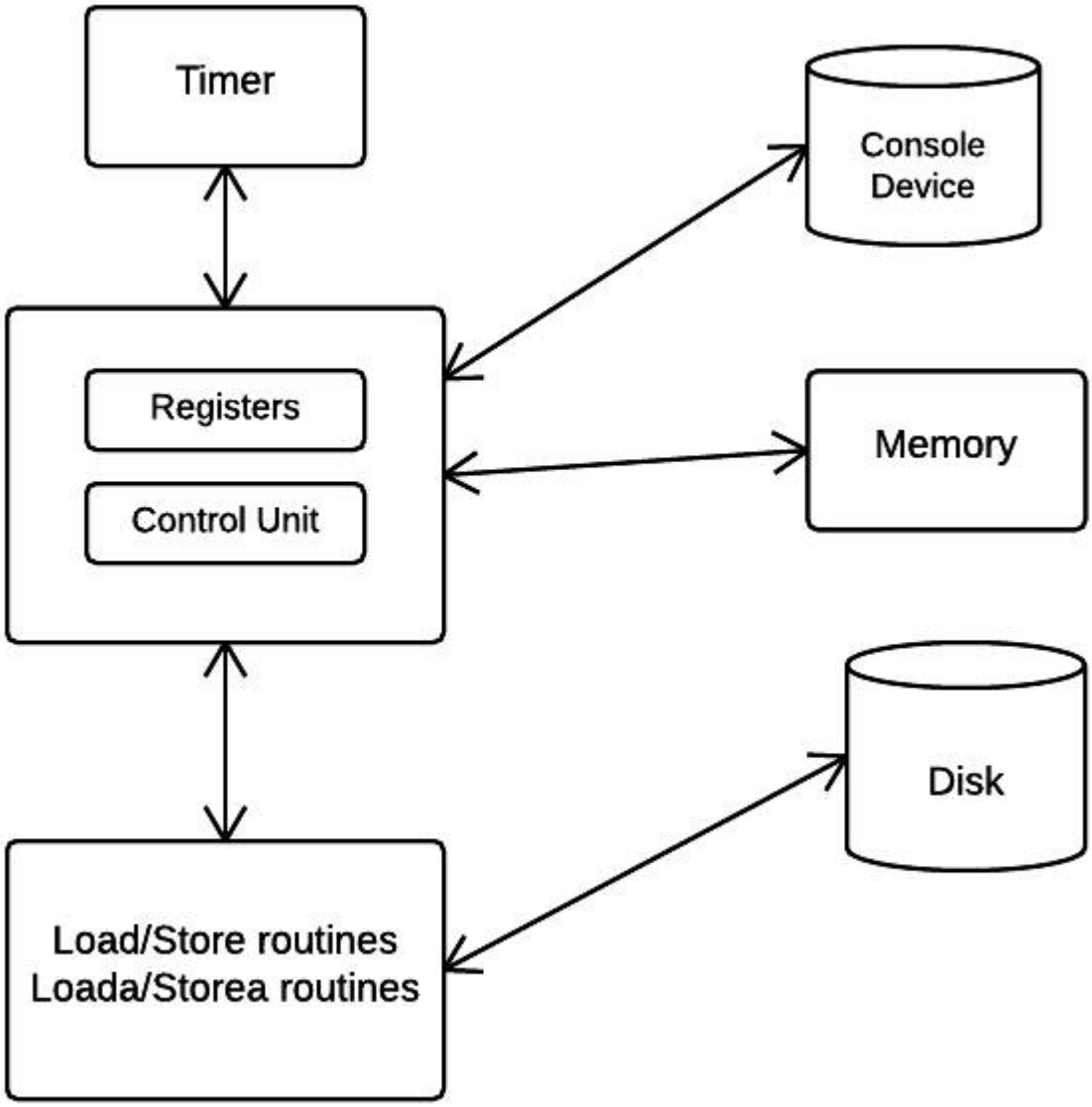

Figure 2.1: Hardware Model in eXpOS

Chapter 3

Experimental File System(eXpFS)

3.1 Introduction

The eXpFS logical file system provides a file abstraction that allows application programs to think of each data (or executable) file stored in the disk as a continuous stream of data (or machine instructions) without having to worry about the details of disk block allocation. eXpOS provides a sequence of file system calls through which application programs can create/read/write files. These system calls are OS routines that does the translation of the user request into physical disk block operations.

In addition to the eXpOS system call interface, the eXpFS specification also requires that there is an external interface through which executable and data files can be loaded into the file system externally. The external interface for eXpOS implementation on the XSM machine is the XFS Interface.

This section discusses the abstract logical view provided by eXpFS to the eXpOS application programmer.

3.2 eXpFS File system organization

The eXpFS logical file system comprises of files organized in a single directory called the root. The root is also treated conceptually as a file. Every eXpFS file is a sequence of words. Associated with each eXpFS file there are three attributes

- name, size and type, each attribute being one word long. The filename must be a string. Each file must have a unique name. The size of the file will be the total number of words stored in the file.(The maximum size of a file is operating system dependent). There are three types of eXpFS files - the root, data files and executable files. Each file in eXpFS has an entry in the root called its root entry.

3.2.1 The eXpFS Root File

The root file has the name root and contains meta-data about the files stored in the file system. For each file stored in eXpFS, the root stores three words of information - filename, file-size and file-type. This triple is called the root entry for the file. The first root entry is for the root itself. The order in which the remaining entries appear is not specified and can vary with the implementation. (The maximum size of the root file is defined by XFS_ROOTSIZE.).
Example: If the file system stores two files - a data file, file.dat, of size 700 words and an executable file, confirm this program.xexe, of 1025 words, the root file will contain the following information.

File name	File size	File Type
root	512	ROOT
file.dat	1024	DATA
program.xexe	1536	EXEC

Figure 3.1: Root File Entry

The operations on the root file are Open, Close, Read and Seek.

3.2.2 eXpFS Data files

A data file is a sequence of words. The maximum number of words permissible in a file is defined by the constant MAX_FILE_SIZE. (It is a recommended programming convention to use the extension ".dat" for data files). eXpFS treats

every file other than root and executable files as a data file. The Create system call automatically sets the file type field in the root entry for any file created through the create system call to DATA.

eXpOS allows an application program to perform the following operations on data files: Create, Delete, Open, Close, FLock, FUnlock, Read, Write, Seek. Application programs can create only data files using the Create system call. In addition to this, data files may be loaded into the eXpFS file system using the external interface.

3.2.3 eXpFS Executable files

These contain executable code for programs that can be loaded and run by the operating system. From the point of view of the eXpFS file system alone, executable files are just like data files except that file type is EXEC in the root entry. eXpFS specification does not allow executable files to be created by application programs. They can only be created externally and loaded using the external interface. However, application programs can read or modify executable files like data files.

Executable files are essentially program files that must be loaded and run by the operating system. Hence the Operating system imposes certain structure on these files (called the executable file format). Moreover, the instructions must execute on the machine on which the OS is running. Thus, there is dependency on the hardware as well. Typically, an application program written in a high level language (like ExpL) is compiled using a compiler that generates the executable file. The compiler generates executable file that is dependent on the operating system as well as the target machine.

An OS implementation on a particular machine specifies an application binary interface (ABI). The eXpOS ABI for XSM machine is described in Appendix A.

Application programs are typically written in a high level language like ExpL. The ExpL compiler for eXpOS running on the XSM machine generates target code based on the ABI specification for eXpOS on XSM.

The executable file format recognized by eXpOS is called the Experimental executable file (XEXE) format. In this format, an executable file is divided into two sections. The first section is called header and the second section called the code (or text) section. The code section contains the program instructions. The header section contains information like the size of the text and data segments in the file, the space to be allocated for stack and heap areas when the program is loaded for execution etc. This information is used by the OS loader to map the file into a virtual address space and create a process in memory for executing the program.

An application program can read/modify an executable file like a data file using the standard system calls on data files. In addition to this, the Exec system call invokes the eXpOS loader that loads an executable file in XEXE format into memory for execution.

Chapter 4

Process Model

A program under execution is called a process. A process is newly created when a process already in execution invokes the Fork system call. The first process, the INIT process, is created by the OS during bootstrap by loading a code stored in a pre-defined disk location to memory and setting up a process. The OS assigns a unique integer identifier called process id for each process when it is created. The process id does not change during the lifetime of the process. The process that creates the new process is called the parent process of the newly created process.

4.1 The Structure of Processes

eXpOS associates a virtual (memory) address space for each process.

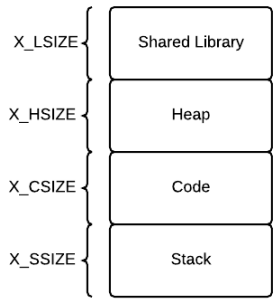

Figure 4.1: Process Model in eXpOS

The address space of a process is a contiguous sequence of memory ad-

dresses, starting from zero, accessible to a process. The eXpOS logically partitions the address space into four regions library, heap, code and stack. These regions are mapped into physical memory using hardware mechanisms like paging/segmentation.

Every process corresponds to some executable file stored in XEXE format stored in the eXpFS file system. The XEXE header of the executable file contains the information about how much space must be allocated for the various memory regions. When the program is loaded into memory by the operating system, the OS reads the header and sets up the regions of the virtual address space accordingly. Once the layout of the virtual address space is clear, the OS maps the virtual address space into the physical memory. The part of the OS which does all these tasks is called the OS loader. The eXpOS loader is the interrupt service routine corresponding to the Exec system call.

A process may open files or semaphores. The OS associates a file handle with each open instance of a file. Similarly, the OS assigns a semaphore identifier (semid) for each semaphore acquired by the process. The file handles and semids acquired by a process are also attributes of a process.

Note: In addition to the above attributes of a process that are visible to application/system programs, a process under execution at any given point of time has an execution context. The context of a process refers to the contents of the registers, instruction pointer, contents of the memory etc. These are hardware dependent and are managed internally by the OS. In fact, managing the execution contexts of multiple processes simultaneously and running them all in one machine is the main challenge in the design and implementation of a multiprogramming OS. However, the OS hides these internal details from the application programs as well as system programs like compilers.

4.2 Operations on Processes

The two most fundamental operations associated with process are Fork and Exec. The remaining operations are Exit, Wait, Signal, Getpid and Getppid.

4.2.1 Semantics of Exec operation

(1) The OS closes all files and semaphores opened by the process. A new address space is created replacing the existing one. The new process inherits the process id of the calling process.

(2) The code (and static data, if any) of the executable file are loaded into the code (and stack) regions of the new address space. The system library is mapped to the library region and stack is initialized to empty.

(3) The machine instruction pointer is set to the location specified in the executable header. The machine stack pointer is initialized to the beginning of the stack. From here, execution continues with the newly loaded program.

4.2.2 Semantics of Fork operation

(1) A new child process with a new process id and address space is created which is an exact replica of the original process with the same library, code,stack and heap regions. (The OS assigns a new process id for the child and returns this value to the parent as the return parameter of the fork system call.) The heap, code and library regions of the parent are shared by the child. Stack is separate for the child and is not shared.

(2) All open file handles and semaphores are shared by the parent and the child. This means that if one of the processes write/read into/from or adjust the file handle using the lseek system call, the corresponding file

handle of the other process also gets automatically updated. Note that file handles (or semaphore identifiers) of files (or semaphores) that are opened (or created) subsequent to the fork operation by the parent or the child will be exclusive to the particular process and will not be shared. The parent and the child continue execution from here on.

4.2.3 Other Operations

The Exit system call terminates a process after closing all files and semaphores. The Wait system call suspends the execution of a process till another process exits or executes a Signal system call. The Signal system call resumes the execution of a process that was suspended by wait.

A process can get its process id using the Getpid system call. The pid of the parent process can be obtained using the Getppid system call.

4.3 Special Processes in eXpOS

eXpOS specifies two special processes, the idle and the init process. These are stored in a predefined location in the disk and loaded to the memory by the bootstrap loader. The main purpose of idle process is to run as a background process in an infinte loop. This is demanded by the OS so that the scheduler will always have a process to schedule. The init process is the first process executed by the OS. The process identifiers for the idle and init processes are fixed as 0 and 1 respectively.

A shell is an ExpL program which takes the name of an executable file as input and executes it. The shell process Forks itself and the child process invokes the Exec system call with the executable file as argument. The shell runs until the user stops the process.

Chapter 5

Hardware Interrupts and Exception Handlers

5.1 Introduction

When the machine is powered on, the system is configured to start executing a ROM code in privileged mode. This code is called the bootstrap loader. This ROM code loads the first block of the disk into a pre-defined area in memory and transfers control to the newly loaded code. This code is called the OS startup code.

The OS startup code loads the system call routines into memory. These routines are loaded as software interrupt handlers. In addition to these, there are three hardware interrupt / exception handler modules that are to be loaded - the timer interrupt handler, the exception handler, the disk interrupt handler and the terminal handler. If the architecture supports other devices, then the corresponding device interrupt handlers also must be loaded by the OS Startup code.

5.2 Timer interrupt handler

The hardware requirement specification for eXpOS assumes that the machine is equipped with a timer device that sends periodic hardware interrupts. The OS scheduler is invoked by the hardware timer interrupt handler. eXpOS specification suggest that a co-operative multitasking round robin scheduling is employed. This means that a round robin scheduling is employed, but a process may go to sleep

inside a system call when:

- The resource which the process is trying to access (like a file or semaphore) is locked by another process (or even internally locked by another OS system call in concurrent execution).

- There is a disk or I/O device access in a system call which is slow. If the wait for the device access is to be avoided, there must be hardware support from the device to send a hardware interrupt when device operation is finished. This allows the OS to put the process on sleep for now, continue scheduling the remaining processes in round robin fashion and then wake up the sleeping process when the device sends the interrupt.

5.3 Exception handler

If a process generates an illegal instruction, an invalid address (outside its virtual address space) or do a division by zero (or other faulty conditions which are machine dependent), the machine will generate an exception. The exception handler must terminate the process, wake up all processes waiting for it (or resources locked by it) and invoke the scheduler to continue round robin scheduling the remaining processes.

The exception handler is invoked when a page required by the process is not present in the memory. This condition is known as a page fault. The module which handles demand paging (if the machine hardware supports demand paging) is invoked by the exception handler when there is a page fault. eXpOS specification does not require implementation of demand paging. However, most machines (including XSM) are equipped with hardware support for demand paging and using the feature can improve machine throughput considerably.

5.4 Disk Controller

eXpOS treats the disk as a special block device and assumes that the hardware provides low level block transfer routines to transfer disk blocks to memory (pages) and back. The block transfer routines contain instructions to initiate block-memory transfer by the disk controller hardware. After initiating the disk-memory transfer, the block transfer routine normally returns to the calling program, which sleeps for the disk operation to complete.

When the disk-memory transfer is complete, the disk controller raises a hardware interrupt. The interrupt service routine (handler) must be part of the OS code to be set up during the bootstrap. The disk interrupt handler is responsible for waking processes that went into sleep awaiting completion of the disk operation.

5.5 Terminal and other Device Handlers

All other data handling devices (other than the disk) are treated as stream devices. This means that each device allows transfer of only one word from memory to the device or back at a time. Some devices may permit only write (like a printer) whereas some devices may permit only read. It is assumed that for each device there are associated low level routines that can be invoked by the OS to transfer data and control instructions. Some of these devices may raise a hardware interrupt when the transfer is complete. Thus, for each device that raises an interrupt, there must be a corresponding device interrupt handler.

For each device part of the hardware, the OS assigns a unique device identifier (devid) which is announced to the application programmer. (The device identifiers are specific to the particular installation). It is assumed that device identifiers are distinguishable from file handles. A user program can write a word into a device using the write system call. The read system call is used when the device allows a word to be read. Read and Write are the only system calls associated with devices.

The standard input and the standard output are two special stream devices with predefined identifiers STDIN = -1 and STDOUT = -2. Standard output permits only Write and standard input permits only Read. The Read operation typically puts the process executing the operation to sleep for console input from the user. When the user inputs data, the console device must send a hardware interrupt. The corresponding handler routine is called terminal handler. The terminal handler is responsible for waking up processes that are blocked for input console

Chapter 6

System Call Interface

Application programmers interact with the Operating System using the system calls. System calls are stored in the disk and are loaded into memory when the OS is loaded by the bootstrap loader. When a process invokes a system call, the process is interrupted and control goes to the corresponding interrupt service routine of the kernel, resulting in a switch from user mode to kernel mode. Once the system call is carried out, the control goes back to the application program, with a switch back to the user mode.

The system calls of eXpOS are classified into file system calls, process system calls and system calls for access control and synchronization.

6.1 File System Calls

(1) Create System Call

The Create operation takes as input a filename and creates an empty file by that name. If a root entry for the file already exists, then the system call returns 0 (success). Otherwise, it creates a root entry for the file name, sets the file type to DATA and file size to 0. Note that the file name must be a character string and must not be root.

(2) Delete System Call

Delete removes the file from the file system and removes its root entry. A

file that is currently opened by any application cannot be deleted. Root file also cannot be deleted.

(3) Open System Call

For a process to read/write a file, it must first open the file. Only data and root files can be opened. The Open operation returns a file descriptor. An application can open the same file several times and each time, a different descriptor will be returned by the Open operation. The file descriptor must be passed as argument to other file system calls, to identify the open instance of the file.

The OS associates a file pointer with every open instance of a file. The file pointer indicates the current location of file access (read/write). The Open system call sets the file pointer to 0 (beginning of the file).

(4) Close System Call

After all the operations are done, the user closes the file using the Close system call. The file descriptor ceases to be valid once the close system call is invoked.

(5) Read System Call

The file descriptor is used to identify an open instance of the file. The Read operation reads one word from the position pointed by the file pointer and stores it into the buffer. After each read operation, the file pointer advances to the next word in the file.

(6) Write System Call

The file descriptor is used to identify an open instance of the file. The Write operation writes the word stored in the buffer to the position pointed by

the file pointer of the file. After each Write operation, the file pointer advances to the next word in the file.

(7) Seek System Call

The Seek operation allows the application program to change the value of the file pointer so that subsequent Read/Write is performed from a new position in the file. The new value of the file pointer is determined by adding the offset to the current value. (A negative Offset will move the pointer backwards). An Offset of 0 will reset the pointer to the beginning of the file.

6.2 Process System Calls

(1) Fork System Call

Replicates the process invoking the system call. The heap, code and library regions of the parent are shared by the child. A new stack is allocated to the child and the parent's stack is copied into the child's stack.

When a process executes the Fork system call, the child process shares with the parent all the file and semaphore descriptors previously acquired by the parent. Semaphore/file descriptors acquired subsequent to the fork operation by either the child or the parent will be exclusive to the respective process and will not be shared.

(2) Exec System Call

Exec destroys the present process and loads the executable file given as input into a new memory address space. A successful Exec operation results in the extinction of the invoking application and hence never returns to it. All open instances of file and semaphores of the parent process are closed. However, the newly created process will inherit the PID of the

calling process.

(3) Exit System Call

Exit system call terminates the execution of the process which invoked it and destroys its memory address space. The calling application ceases to exist after the system call and hence the system call never returns.

(4) Getpid System Call

Returns the process identifier of the invoking process. The system call does not fail.

(5) Getppid System Call

Returns to the calling process the value of the process identifier of its parent. The system call does not fail.

(6) Shutdown System Call

Shutdown system call terminates all processes and halts the machine.

Arguments : None

Return Value : None

6.3 System calls for access control and synchronization

(1) Wait System Call

The current process is blocked till the process with PID given as argument executes a Signal system call or exits. Note that the system call will fail if a process attempts to wait for itself.

(2) Signal System Call

All processes waiting for the signaling process are resumed. The system call does not fail.

(3) FLock System Call

To lock a file so that other applications running concurrently are not permitted to access the file till the calling process unlocks it. If the file is already locked by some other process, the system call waits for the file to be unlocked, locks it, and returns to the calling process.

(4) FUnLock System Call

FUnLock operation allows an application program to unlock a file which the application had locked earlier, so that other applications are no longer restricted from accessing the file.

(5) Semget System Call

This system call is used to obtain a binary semaphore. eXpOS has a fixed number of semaphores. The calling process can share the semaphore with its child processes using the fork system call.

(6) Semrelease System Call

This system call is used to release a semaphore descriptor held by the process.

(7) SemLock System Call

This system call is used to lock the semaphore. If the semaphore is already locked by some other process, then the calling process goes to sleep and wakes up only when the semaphore is unlocked. Otherwise, it locks the semaphore and continues execution.

(8) SemUnLock System Call

This system call is used to unlock a semaphore that was previously locked by the calling process. It wakes up all the processes which went to sleep trying to lock the semaphore while the semaphore was locked by the calling

process.

Chapter 7

Algorithms Used

Algorithm 1 Create System Call

Input: Filename

Output: 0 (Success) or -1 (No Space for file)

If the file is present in the system, return 0.
Find the index of a free entry in the Inode Table. If no free entry found, return -1.
Allocate the Inode Table entry to the file.
Update the Root file by adding an entry for the new file.
return 0

Algorithm 2 Delete System Call

Input: Filename

Output: 0 (Success) or -1 (File not found) or -2 (File is open)

If file is not present in Inode Table or if it is not a DATA file, return -1.
Find the index of the file in the Inode Table.
If File Table entry exists with the same index as found above, return -2.
Update Inode Table.
Update the Root file by invalidating the root entry for the file.
return 0

Algorithm 3 Open System Call

Input: Filename

Output: File Descriptor (Success) or -1 (File not found) or -2 (Process has reached its limit of resources) or -3 (System has reached its limit of open files)

If file is not present in Inode Table or if it is of type EXEC, return -1.
Find the index of the Inode Table entry of the file.
Allocate entry in Per Process Resource Table
if file is already open **then**
  Get the File Table entry of the file and increment the File Open Count.
else
  Find a free entry in the File Table. If there are no free entries, return -3.
end if
Allocate the Per-Process Resource Table entry to the file.
Update the File Table entry.
return from system call with the index of the Per-Process Resource Table entry.

Algorithm 4 Close System Call

Input: File Descriptor

Output: 0 (Success) or -1 (File Descriptor is invalid) or -2 (File is locked by calling process)

If file descriptor does not correspond to valid entry in Per Process Resource Table, return -1
Get the index of the File Table entry from Per-Process Resource Table entry.
If file is locked by the current process, return -2.
In the File Table Entry found above, decrement the File Open Count. If the count becomes 0, invalidate the entry.
Invalidate the Per-Process Resource Table entry.
return 0

Algorithm 5 Read System Call

Input: File Descriptor and a Buffer (a String/Integer variable) into which a word is to be read from the file

Output: 0 (Success) or -1 (File Descriptor is invalid) or -2 (File pointer has reached the end of file)

if input is to be read from terminal **then**
  Wait till terminal gets the input for the current process.
  Copy the word from the input buffer of the Process Table to the buffer passed as argument and return with 0.
end if
If file descriptor does not correspond to a valid entry in Per Process Resource Table, return -1.
Check the validity of the File pointer
Find the block and the position in the block from which input is read.
while the file is locked by a process other than the current process **do**
  put the process to sleep and call the scheduler.
end while
Lock the file.
Get the buffer page number from block number.
while the buffer is locked by a process other than the current process **do**
  put the process to sleep and call the scheduler.
end while
Lock the buffer page.
if the buffer contains a block other than the required block **then**
  Bring the block to the buffer from the disk
end if
Copy the word at the offset position of the block into the buffer passed as argument and Increment lseek position.
Unlock the buffer and wake up all processes waiting for the buffer.
Unlock the file and wake up all processes waiting for the file.
return 0

Algorithm 6 Write System Call

Input: File Descriptor and a Word to be written

Output: 0 (Success) or -1 (File Descriptor given is invalid) or -2 (No disk space)

if word is to be written to standard output **then**
Issue the machine instruction to output the given word and return 0.
end if
If file descriptor does not correspond to valid entry in Per Process Resource Table, return -1.
Get the index of the File Table entry and lseek position from the Per Process Resource Table entry.
If lseek is equal to MAX_FILE_SIZE - 1, return -2.
Find the block and position in the block to which data has to be written.
while the file is locked by a process other than the current process **do**
put the process to sleep and call the scheduler.
end while
Lock the file.
Get the buffer page number from block number.
while the buffer is locked by a process other than the current process **do**
put the process to sleep and call the scheduler.
end while
Lock the buffer page.
if the buffer contains a block other than the required block **then**
Bring the block to the buffer from the disk
end if
Copy the word passed as argument to the offset position in the buffer.
Set dirty bit to 1 in the Buffer Table entry and increment lseek position.
Increment file size in the Inode Table entry and Root file entry.
Unlock the buffer and wake up all processes waiting for buffer.
Unlock the file and wake up all processes waiting for the file.
return 0.

Algorithm 7 Seek System Call

Input: File Descriptor and Offset

Output: 0 (Success) or -1 (File Descriptor given is invalid) or -2 (Offset value moves the file pointer to a position outside the file)

If file descriptor does not correspond to valid entry in Per Process Resource Table, return -1.
Get the index of the File Table entry and lseek position from the Per Process Resource Table entry.
Check the validity of the given offset
if given offset is 0 **then**
  Set lseek value in the Per-Process Resource Table entry to 0.
else
  Change the lseek value in the Per-Process Resource Table entry to lseek+offset.
end if
return 0

Algorithm 8 Fork System Call

Input: None

Output: Process Identifier to the parent process and 0 to child process (Success) or -1 (Number of processes has reached its maximum, returned to parent)

If no free entry in the Process Table, return -1.
Find the index of a free entry in the Process Table and set the PID and PPID field.
Count the number of valid stack pages from the Page Table of the parent process.
If sufficient number of free pages are not present in memory, then increment the WAIT_MEM_COUNT in the System Status Table.
while sufficient number of free pages are not present in memory **do**
  put the process to sleep and call the scheduler.
end while
for each stack page of the parent process **do**
  if the stack page is valid **then**
    Allocate a free page to the child.
    Copy the stack page of the parent into the child stack page.
  else
    Share the swap block containing the stack page with the child.
  end if
end for
Construct the context of the child process.
Set the return value to 0 for the child process.
The PID of the child process is set as the return value for the parent process.
return to the parent process.

Algorithm 9 Exec System Call

Input: Filename

Output: -1 (File not found or file is of invalid type)

If file not found in system or file type is not EXEC, return -1.
For each page of the current process that is swapped out, find the swap block and decrement its entry in the Disk Free List.
Setup code and library pages.
Free the heap pages
In the Process Table entry of the current process, set the Inode Index field to the index of Inode Table entry for the file.
Close all files opened by the current process.
Release all semaphores held by the current process.
Set SP to the start of the stack region and IP to the start of the code region.
return from system call to newly loaded process.

Algorithm 10 Exit System Call

Input: None

Output: None

If no more processes to schedule, shutdown the machine.
Unlock and close all files opened by the current process.
Release all the semaphores used by the current process.
Wake up all processes waiting for the current process.
Invalidate the Process Table entry and the page table entries of the current process
Invoke the scheduler to schedule the next process.

Algorithm 11 Getpid System Call

Input: None

Output: Process Identifier (Success)

Find the PID of the current process from the Process Table.
return the PID of current process.

Algorithm 12 Getppid System Call

Input: None

Output: Process Identifier (Success)

Find the PPID of the current process from the Process Table.
return PPID.

Algorithm 13 Wait System Call

Input: Process Identifier of the process for which the current process has to wait.
Output: 0 (Success) or -1 (Given process identifier is invalid or it is the pid of the invoking process)
If process is intending to wait for itself or for a non-existent process, return -1.
Change the status from (RUNNING, _) to (WAIT_PROCESS, Argument_PID) in the Process Table.
Invoke the Scheduler to schedule the next process.
return 0

Algorithm 14 Signal System Call

Input: None
Output: 0 (Success)
Wake up all processes waiting for the current process.
return 0

Algorithm 15 FLock System Call

Input: File Descriptor
Output: 0 (Success) or -1 (File Descriptor is invalid)
If file descriptor does not correspond to valid entry in Per Process Resource Table, return -1.
while the file is locked by a process other than the current process **do**
Change the state to (WAIT_FILE, ftindex) where ftindex is the File Table index of the locked file and call the Scheduler.
end while
Change the ULock field of the file table to PID of current process.
return 0

Algorithm 16 FUnLock System Call

Input: File Descriptor
Output: 0 (Success) or -1 (File Descriptor is invalid) or -2 (File was not locked by the calling process)
If file descriptor does not correspond to a valid entry in Per Process Resource Table, return -1.
if file is locked **then**
If current process has not locked the file, return -2.
Unlock the file.
Wake up all processes waiting for the file.
end if
return 0

Algorithm 17 Semget System Call

Input: None

Output: semaphore descriptor (Success) or -1 (Process has reached its limit of resources) or -2 (Number of semaphores has reached its maximum)

Find the index of a free entry in Per Process Resource Table. If no free entry, then return -1.

Find the index of a free entry in Semaphore table and increment the process count. If no free entry, return -2.

Store the index of the Semaphore table entry in the Per Process Resource Table entry.

return Semaphore Table entry index.

Algorithm 18 Semrelease System Call

Input: Semaphore Descriptor

Output: 0 (Success) or -1 (Semaphore Descriptor is invalid)

If Semaphore descriptor is not valid or the entry in Per Process Resource Table is not valid , return -1.

if semaphore is locked by the current process **then**

Unlock the semaphore before release.

end if

Decrement the process count of the semaphore.

Invalidate the Per-Process resource table entry.

return 0

Algorithm 19 SemLock System Call

Input: Semaphore Descriptor

Output: 0 (Success or the semaphore is already locked by the current process) or -1 (Semaphore Descriptor is invalid)

If Semaphore descriptor is not valid or the entry in Per Process Resource Table is not valid , return -1.

while the semaphore is locked by a process other than the current process **do**

Put the current process to sleep

Call scheduler

end while

Change the Locking PID to PID of the current process in the Semaphore Table .

return 0

Algorithm 20 SemUnLock System Call

Input: Semaphore Descriptor

Output: 0 (Success) or -1 (Semaphore Descriptor is invalid) or -2 (Semaphore was not locked by the calling process)

```
If Semaphore descriptor not valid or the entry in Per Process Resource Table is not valid , return -1.
if semaphore is locked then
    If current process has not locked the semaphore, return -2.
    Unlock the semaphore.
    Wake up all processes waiting for the semaphore.
end if
return 0
```

Algorithm 21 Timer Interrupt Handler

```
Save the context of current process and mark it as ready.
Increment TICK
if free memory pages present then
    if there are sleeping processes that requires memory pages then
        Wake up all processes that requires memory pages.
    else
        if disk is free and there are swapped processes then
            Swap in the seniormost swapped out process.
        end if
    end if
else
    if disk is free and there are processes that require memory pages then
        Use the modified second chance algorithm to find an unreferenced page
        if the unreferenced page is a code page then
            In the page table entry of the unreferenced page, store the code block number.
        else
            If the unreferenced page is a stack or heap page, swap the page to a free swap block in the disk.
        end if
        if unreferenced page is pointed to by the stack pointer of the process then
            Mark the process that owns the page as swapped out.
        end if
    end if
end if
Schedule the next ready process
```

Algorithm 22 Disk Interrupt Handler

In the Disk Status Table, set the STATUS field to 0, indicating that the disk is no longer busy.
if load/store was issued by a process **then**
  Wake up all processes waiting for the disk.
  If the loaded block was a swap block, decrement the corresponding Disk Free List entry.
else
  Get the PID of the process for which the scheduler had issued the load/store instruction.
  if the disk operation was a load operation **then**
    Update Disk Free List.
    Update the Page Table of the process.
    Mark the process as ready.
    Decrement SWAPPED_COUNT in System Status Table.
  else
    Update Memory Free List.
    Update Page Table of the process.
    Increment the MEM_FREE_COUNT in the System Status Table.
  end if
end if

Algorithm 23 Exception Handler

If the exception is not caused by a page fault, display the cause and exit the process.
Find the page table entry of the page causing the exception.
If there are no free pages in the memory, then increment the WAIT_MEM_COUNT in the System Status Table.
while there are no free pages in memory **do**
  Put the current process to sleep and call scheduler
end while
Find a free page in memory.
if the page that caused exception is stored in the disk **then**
  while the disk is busy **do**
    Put the current process to sleep and call scheduler
  end while
  Load the block to the memory page found above.
  Put the current process to sleep and call scheduler.
end if
Update the Page Table entry of the page that caused exception.
return to the user process.

Algorithm 24 Terminal Interrupt Handler

Set the status field in Terminal Status Table to 0.
Locate the Process Table entry of the process that read the data.
Copy the word read from standard input to the Input Buffer field in the Process Table entry.
Set the PID field of Terminal Status Table to -1.
Wake up all processes waiting for terminal.

Algorithm 25 OS Startup Code

Load the software interrupts and handler routines to memory.
Load the init program to memory and set up its machine context.
Load the idle program to memory and set up its machine context.
Load the disk data structures to the memory.
Initialize all the memory data structures.
Schedule the init process for execution.

Algorithm 26 Shutdown System Call

while disk is not free **do**
  Put the current process to sleep and call scheduler
end while
Store Inode Table to the disk.
Store dirty pages contained in buffer to the disk.
Store Disk Free List to the disk.
Halt the machine.

Algorithm 27 Shell process

while true **do**
  Get the program to be executed using read system call
  If the shutdown instruction is received, invoke shutdown system call.
  childPID = fork();
  if childPID == 0 **then**
    Execute the program given by read.
  else
    Wait for child to finish execution.
  end if
end while

Algorithm 28 Idle process

while true **do**
end while

Bibliography

Appendix A

Appendix

A.1 State Transition Diagram in eXpOS

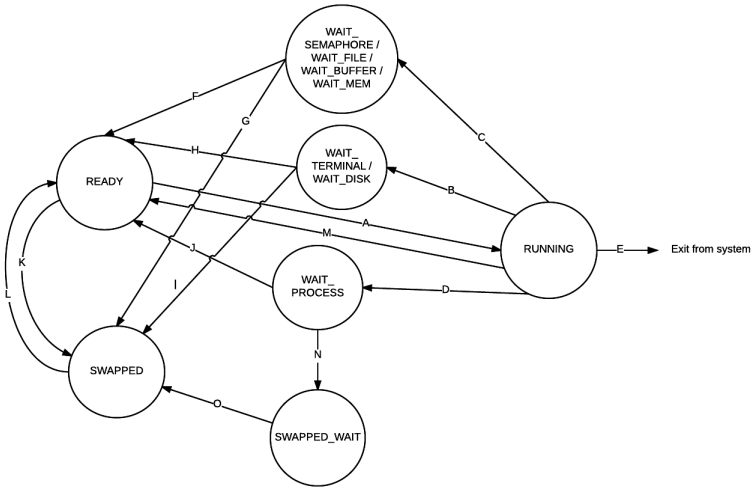

Figure A.1: State Transitions

A

READY → RUNNING : The scheduler has scheduled the process for execution.

B

RUNNING → WAIT_TERMINAL : The process is either waiting for access to the terminal or for data to be inputted through terminal.

RUNNING → WAIT_DISK : The process is either waiting for access to the disk

or for the disk operation to finish.

C

RUNNING → WAIT_SEMAPHORE : The semaphore which the process is trying to use, is found to be locked.

RUNNING → WAIT_FILE : The file which the process is trying to read/write, is found to be locked.

RUNNING → WAIT_BUFFER : The buffer which the process is trying to use, is found to be locked.

RUNNING → WAIT_MEM : The process requires a free memory page but there are none in the memory.

D

RUNNING → WAIT_PROCESS : The process is waiting for another process to either exit or to signal it.

E

RUNNING → Exit from system : The process has either completed execution or has invoked an Exit System Call.

F

WAIT_SEMAPHORE → READY : The semaphore for which the process was waiting, is now unlocked.

WAIT_FILE → READY : The file for which the process was waiting, is now unlocked.

WAIT_BUFFER → READY : The buffer for which the process was waiting, is now unlocked.

WAIT_MEM → READY : There are free pages in memory.

G

WAIT_SEMAPHORE / WAIT_FILE / WAIT_BUFFER / WAIT_MEM → SWAPPED : The current stack page of the process has been swapped out.

H

WAIT_TERMINAL → READY : The input data has been read from terminal and terminal is free to be used by any process.
WAIT_DISK → READY : The disk operation is complete.

I

WAIT_TERMINAL / WAIT_DISK → SWAPPED : The current stack page of the process has been swapped out.

J

WAIT_PROCESS → READY : The process has either received a signal from the process it was waiting for or the latter has exited the system.

K

READY → SWAPPED : The current stack page of the process has been swapped out.

L

SWAPPED → READY : The stack page required by the process to continue execution was swapped in.

M

RUNNING → READY : Context switch caused by the timer interrupt routine.

N

WAIT_PROCESS → SWAPPED_WAIT : The current stack page of the process was swapped out while waiting for another process.

O

SWAPPED_WAIT → SWAPPED : The process in SWAPPED_WAIT state has either received a signal from the process it was waiting for or the latter has exited.