The Experimental Operating System (eXpOS) Specification

Who should read this document?

This document must be read by anyone wishing to write eXpOS application programs in a high level language or assembly language. This OS specification is prepared in a programming language and target machine independent manner.

The document is also informative for system programmers like compiler designers who wishes to write the application programmer's interface for a high level language like APL to be run on eXpOS. The application binary interface is the must-read document for the compiler designer.

Finally, this document must be read by anyone who wants to implement eXpOS on some machine because one needs to know the specification for the OS before getting started with its implementation!

Introduction

This document gives application programmer's documentation for the eXperimental Operating System (**eXpOS**).

eXpOS is a tiny multiprogramming operating system. It has a very simple specification that allows a junior undergraduate computer science student to implement it in a few months, subject to availability of adequate hardware and programming platform support.

This specification is prepared from the perspective of the user/application programmer and is not hardware specific. Assumptions about the underlying hardware features necessary for the operating system to work are discussed in the first section of this document.

This document is divided into the following sections. Section 1 provides an overview of the system. Section 2 discusses the fundamental abstractions provided by the OS. Section 3 describes the logical file system of eXpOS called the Experimental File System (XFS), the various types of files supported by XFS and the operations on files. Section 4 describes the structure of eXpOS processes and the operations on processes. This section also discusses the special processes INIT and the shell. Section 5 describes the mechanisms for handling synchronization and resource access control (necessary when processes execute concurrently). Section 6 is a “miscellaneous” section that discusses special processes/routines like the INIT process, the shell, the scheduler, the exception handler, I/O and disk devices and handlers etc. Finally Section 7 provides a detailed specification of the application programmer's system call interface.

Section 1: Overview

Primitives:

It is assumed that the reader has some working familiarity with the following terms and concepts.

Machine: The hardware/machine (computer) on which the operating system is running.

(Physical) Memory: The physical memory (primary memory) of the machine.

Word: The fundamental unit of memory access/storage recognized by the XOS. A word is assumed to be able to store an integer or a character string.

Page: The memory is assumed to be divided into pages of contiguous memory words. It is assumed that the machine supports paging hardware.

Disk: The secondary storage where data and program files are stored. The disk operating system code also is stored in the disk and is loaded from the disk to the memory at the time of bootstrap.

Block: The basic unit of disk access. A block can store a sequence of words. The number of words per block is hardware dependent.

File: Each file is a sequence of words, stored in the disk. The most important file types are **program** files (or **executable** files or **application programs**) and **data** files.

Process: An application program under execution is called a process.

Kernel: The core part of the eXpOS operating system that forms a layer between the hardware and application programs. The kernel essentially is a collection of routines residing in the memory of the machine. In this document, the term "operating system" normally refers to the kernel.

System calls: These are kernel routines which application programs can invoke to do actions which only the OS reserves right to perform (Example: creating/modifying files or creating/destroying processes).

Multiprogramming: Multiple processes reside in the memory simultaneously and the OS time-shares the machine between the processes by **scheduling**. The processes are said to execute concurrently. An OS that supports concurrent execution of processes is called a **multiprogramming** OS.

Timer: The hardware device that interrupts the machine periodically. The scheduler is generally invoked by the timer interrupt service routine.

Assumptions about hardware.

The OS specification assumes a generic hardware model described below.

Figure here on the hardware model – abstract one – not depend on XSM.

The basic Machine model consists of memory, disk and the CPU. A small part of memory is assumed to contain a **bootstrap loader** stored permanently in ROM memory. These are machine instructions to load into the memory an **OS startup code** stored in a pre-defined area in the disk. The ROM code then transfers control to this newly loaded code. This code loads the operating system routines stored in (pre-defined areas of) the disk into memory and sets up the Operating system. This includes all the OS code for various system calls, the scheduler, the exception handler, device drivers etc. Further hardware support required like the timer, disk controller, Input-output system etc. are discussed later.

For eXpOS to work, the machine should support two **privilege levels** of program execution. These are called **the user mode** (unprivileged mode) and the **system mode** (privileged mode). User programs (or application programs) run in user mode whereas OS routines run in system mode. The collection of OS routines that run in system mode is called the **kernel** of the operating system. A user program in execution is called a **process**. (Sometimes the term "program" may be (ab)used to refer to the corresponding process). An application process has access only to a limited set of machine instructions and can only access a limited set of memory addresses called the **virtual address space** of the process. This restricted machine model provided by the OS (of course, using the support from the machine architecture) is called the **virtual machine model.**

The OS logically divides virtual address space of a process into **library**, **code, data, stack and heap** regions. eXpOS assumes that the machine provides **paging** hardware to implement the mapping of virtual address space into the physical memory when the program is loaded into memory for execution. If the machine supports segmented paging, the feature can be used profitably. However, the discussion here assumes no segmentation support.

Data and program files are stored in the disk. The disk is typically divided into **blocks** and the machine provides instructions to transfer blocks into and out of memory. These instructions can be accessed only in the system mode. The specific mechanisms available are hardware dependent. Note that since processes run in user mode, they can access disk files only through invoking the designated system calls for the purpose.

User programs are generally stored as **executable files** in the disk. Typically the user writes the application programs in a high level language and a compiler generates the executable file. The OS expects that executable files follow certain format and compilers must adhere to the format. This allows the OS to figure out where and how much space must be allocated for library, stack, code, data and heap in the virtual address space when the program is loaded into memory for execution. The virtual machine model as well as the executable file format for eXpOS are described in the eXpOS **application binary interface** documentation.

At the end of the bootstrap process, the OS startup code hand-creates the first user process called the **INIT process** in memory. Thereafter, new processes can be created by existing processes by invoking the OS system calls for the purpose. Recall that the system call routines would have been set up in the memory during the bootstrap process. The INIT process creates a special user process called the **shell process** by loading and executing a **shell program** from the disk. The shell program repetitively reads user commands from the input and executes programs specified by the user and the OS becomes functional. The specification of the INIT and the shell process are described

eXpOS treats the standard input and output devices just like special files. Hence user processes must use the system calls to read/write files to perform I/O operations. The underlying implementation details are hardware dependent and are left unspecified in the OS specification.

Finally, eXpOS assumes that the machine is equipped with a **hardware timer** device that can interrupt the machine at pre-defined regular intervals. The timer is crucial for multiprogramming. The timer interrupt handler is responsible for invoking the eXpOS kernel's **scheduler** routine which is responsible for **timesharing** the machine between processes. Similarly, disk/input-output devices may require handler interrupt service routines. These are hardware dependent and hence left unspecified in the OS specification. A discussion of these issues is taken up in Section 6.

Section 2: The fundamental eXpOS Abstractions

eXpOS provides the following fundamental abstractions to an application program: 1) The XFS **logical file system** 2) The **process** abstraction for programs in execution 3) Methods of resource sharing 4) Primitives for concurrent access control and process synchronization and 5) The **system call interface** that specifies the interface through which application programs can invoke the system calls and access the OS services.

The XFS logical file system.

The eXpOS kernal provides a hardware independent logical file system model (called the **experimental file system or XFS**) for the application programs. The application program views files as being organized and stored in the XFS logical file system. Application programs are not permitted to access files directly. Instead, they must invoke the appropriate **file system call** for creating, modifying or accessing files. The OS routine implementing each system call internally translates the request into disk block operations, hiding the hardware details from the application program.

XFS support three kinds of files – **data files, program files** (executable files) and a special file called the **root file**. The root file is a meta-data file that contains the list of all files in the file system. A data file consists of a sequence of words. A program contains the sequence of machine instructions of the program together with the static data of the program and a header.

eXpOS does not provide any mechanisms for application programs to create **executable files**. Executable files have to be pre-loaded into the disk using some other external disk access mechanism before OS bootstrap. Since such mechanisms are implementation dependent, they are not part of the OS specification. For instance, the **XFS-Interface** tool for eXpOS implementation on the XSM machine is one such mechanism.

Executable files follow certain format called the **experimental executable format or XEXE format**. The OS will execute only program files stored in the file system in the XEXE format. Hence system programs like compilers that translate high level application programs must ensure that the executable files adhere to the XEXE format.

Application programs can create, modify and delete data files using appropriate OS system calls.

The eXpOS process Abstraction.

It was noted earlier that at the end of bootstrap, eXpOS loads into memory a program stored in a pre-determined part of the disk and create the first process called the INIT process. Once a process is created, it can spawn new processes using the **fork** system call. When a process spawns a new process, the former is called the **parent process** and the later is called the **child process**. A process can decide to terminate itself using the **exit** system call.

Associated with each process, there is a (virtual) address space (or logical memory space). This address space is a sequence of memory locations, each of which can store a word. XOS logically divides the address space of a process is into four regions – (shared) library, code, stack and heap.

When a process is created using the fork system call, the OS creates a virtual address space for the new process. Each process is given a view that it has its own virtual address space containing its code, data, library, stack and heap. The virtual address space is a continuous address space starting from address 0 up to a maximum limit that is implementation dependent. Internally, the OS maps the virtual address space into the machine memory using hardware mechanisms available in the machine like paging/segmentation.

The code region of a process contains the machine instructions that are to be executed. This code consists of the instructions stored in some executable file in the file system. When a new process is created using the fork system call, the child process shares the library, code and heap with the parent. This means that any modifications to memory words in these regions by one process will result in modification of the contents for both processes. The stack and the data regions of the parent and the child will be separate. The parent and the child concurrently proceeds execution from the instruction immediately after the fork system call in the code.

The stack and the data regions of a process stores the variables (including static variables) and stack frames during the execution of the program. Dynamic memory allocation is normally done from the heap region. Variables to be shared between different processes could also be allocated in the heap. Finally all (standard) library code (which is typically shared by all applications) is mapped to the library region.

A process can load a XEXE executable file from the file system into the virtual address space (of the calling process) using the **exec** system call. The exec system call determines the size of the code, stack and heap regions for the loaded program using the information in the **header** of the executable file. During such loading, the original code, data and heap regions are overlayed by those of the newly loaded program. However, if the original process had shared its heap with its parent process (or any other process), the OS ensures that the heap area is allocated separately so that other processes do not lose their shared heap data. Finally, the (shared) library is common to every application.

The OS expects that executable files respect the programming conventions laid down by the OS (called the **Application Binary Interface – ABI**) like the division of memory into stack, data, code, heap and library. Each XEXE executable file must have a header which specifies the how much size must be allocated by the OS for each region when the program is loaded for execution. Application programs are typically written in high level languages like APL. eXpOS expects the compiler to generate code respecting the ABI specification.

It must be noted here that an application program is free to violate the ABI conventions and decide to use its virtual address space in its own way. It is only required that the executable file follows XEXE format in order to ensure that exec system call does not fail. As long as such a process operates within its own address space, the OS permits the process to execute. However, if at any point during its execution, the process generates a virtual address beyond its permitted virtual address space, a hardware exception will be generated and the OS routine handling the exception will terminate the process.

Access Control and Synchronization

Two concurrently executing processes sharing resources like files or memory (for example, parent and child processes sharing the heap) would like to ensure that only one of them execute critical section code that access/modify a resource that is shared between them.

A classical solution to this problem is using **semaphores**. A process can acquire a semaphore using the semget system call and share it with its child (or later generation) processes. A semaphore can be locked by any of these sharing processes using the semlock system call and the execution of all other processes trying to lock the same semaphore subsequently will be suspended (**blocked**) by the OS until the locking process unlocks the semaphore using the semunlock system call.

A process can lock a (data) file using the **flock** system call. Once locked, every other process trying to execute any system call on the file will be blocked by the OS till the file is unlocked by the original process using the **funlock** system call.

The **wait** system call allows a process to suspend its own execution until another process wakes it up using the **signal** system call. This primitive is useful when a process must be made to wait at a point during its execution until another related process signals it to continue.

Resource Sharing in eXpOS

A process can open data files using the **open** system call. The open system call returns a **file handle**. The file handle advances as data is read from the file. Similarly, a process can acquire a **semaphore** using the **semget** system call.

When a process executes the fork system call, the child process shares with the parent all the file handles and semaphores previously opened by the parent. Thus, when either the parent or the child process write into the file after the fork, the file pointers of both process get updated. This allows processes to synchronize access. Note that semaphores and files handles acquired subsequent to the fork operation by either the child or the parent will be exclusive to the respective processes and will not be shared with the parent.

It was already noted that the child process shares the **heap** of the parent process. Hence memory allocated in the heap will be a **shared memory** between both the processes. However, if either the parent or the child process loads another program into its virtual address space using the exec system call, then the shared heap is detached from that process and the surviving process will have the heap intact.

Thus, eXpOS does not support any explicit primitives for file or memory sharing, but instead allows related processes to share these resources implicitly using the fork system call semantics.

Figures to be added.

The System Call Interface.

The XOS system calls are software interrupt routines of the eXpOS kernal which are loaded to the memory when the OS is bootstrapped. These routines define the services provided by the OS to application programs. These services include accessing files and semaphores, creating a new processes , sending a signal to another process etc.

Application programs are not permitted to directly access files/semaphores or create new processes. Instead they must invoke the corresponding system call routines. The system calls are kernal routines and operate in **privileged or system mode**. Thus when an application program invokes a system call (by invoking the corresponding software interrupt), a change of mode from unprivileged mode to privileged mode occurs. The system call code checks whether the request is valid and the process has permission to the resources/actions requested and then perform the request. Upon completion of the interrupt service routine, control is transferred back to the user process with a switch back to the unprivileged mode.

eXpOS system calls can be classified as file system calls, process system calls and system calls for access control and synchronization. The following table lists the system calls.

File system calls:

System Call	**Functionality**
Create	Create an XFS file
Delete	Delete an XFS file
Open	Open an XFS file and return a file handle to the calling process
Close	Close an XFS file already opened by the calling process
Read	Read one word from the location pointed to by the file handle and advance the file handle to the next word in the file.
Write	Write one word from the location pointed to by the file handle and advance the file handle to the next word in the file.
Lseek	Change the position of the file handle.

Process System calls:

System Call	Functionality
Fork	Create a child process allocating a new address space.
Exec	Load and execute an XFS executable file into the virtual address space of the present process.
Exit	Destroy the process invoking the call.
Getpid	Get the Process ID of the invoking Process
Getppid	Get the process ID of the parent process of the invoking process.

System calls for access control and synchronization:

System Call	Functionality
Signal	Send a signal to a process specified in the call.
Wait	Suspend execution of the current process until a signal is received from the process specified or the process specified terminates.
Flock	Wait until an exclusive access permission to file is obtained.
Funlock	Release the lock on a file already acquired.
Semget	Acquire a new semaphore
Semrelease	Release a semaphore already acquired by the process
Semlock	Wait till the process get exclusive lock of the semaphore
Semunlock	Release the lock on a semaphore already acquired

<u>Section 3: The Experimental File System (XFS)</u>

eXpOS assumes that the disk is a sequence of blocks, where each block can store a sequence of words. The number of words in a block is hardware dependent. Generally, the hardware comes with machine instructions to transfer a block from the disk to a contiguous space in memory and back.

The XFS 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. Thus XFS hides the details of physical storage from application programs.

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 XFS specification also requires that there is an **external interface** through which executable and data files can be loaded into the file system externally. The details of the external interface are implementation specific.

In this section we discuss the abstract logical view provided by XFS to the eXpOS application programmer.

XFS File system organization

The XFS logical file system comprises of files organized in a single directory called the **root.** The root is also treated conceptually as a file. The structure and operations on XFS files including the root are described below.

As noted already, Every XFS file is a sequence of words. Associated with each XFS 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 XFS files – the **root, data files** and **executable files**. Each file in XFS has an entry in the root called the **root entry**. The various file types and file operations are described below:

The XFS Root File

The root file has name **root** and contains the meta-data about the files stored in the file system. For each file stored in XFS, 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, program.xsm, of 1025 words, the root file will contain the following information assuming block size of 512.

File name	File size	File Type
root	9	ROOT
file.dat	700	DATA
program.xsm	1025	EXEC

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

Since the operations on the root file is a subset of the operations on data files, with the same syntax and semantics, these operations are discussed together with other operations on data files.

XFS Data files

A data file is a sequence of words. The maximum number of words permissible in a file is defined by the constant **XFS_DATASIZE**. (It is a recommended programming convention to use the extension ".dat" for data files.) XFS treats every file other than root and executable files (will be described later) as a data file. The **Create** system call automatically sets the file type field in the root entry for any file created through the open system call to DATA (.dat).

XOS allows an application program to perform the following operations (by invoking appropriate system calls) 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 XFS file system using the external interface.

In addition to the above, the description of the operations **FLock** and **FUnlock** are discussed in Section 5.

XFS 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 XFS file system alone, executable files are just like data files except that file type EXEC in the root file entry. XFS 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 APL is compiled using a compiler that generates the executable file. The compiler generates executable file that is dependent on the operating system as well the target machine.

An OS implementation on a particular machine specifies an **application binary interface** (**ABI**). The ABI defines the following:

1. The machine model – that is, the **instruction set** and the **virtual memory address space** to which a compiler generating executable file must generate target code for. This is very much architecture specific.

2. A logical **division of the memory address space into regions - text, data, stack, heap and library** and the **low level (assembly/machine level) system call interface**. This part is dependent on both the OS and the architecture.

3. The file format to be followed for creating executable files by the compiler (**the XEXE executable file format**). This part is generally OS specific and architecture independent.

Important Note: Application programs are typically written in a high level language like APL. A high level language implementation for an OS comes with an **Application Programmers Interface** (**API**) for the OS system calls. This document describes the library functions which the application programs must invoke for each operating system call. The compiler will translate the library call to corresponding low level interrupt calls as specified in the ABI. Thus, application programmers need to know only API. The APL compiler for eXpOS running on the XSM machine generates target code based on the ABI specification for eXpOS on XSM. Thus the ABI becomes the most important document for the compiler back end design.

The executable file format recognized by eXpOS is called the **Experimental executable file (XEXE) format.** In this format, an executable file is divided into three sections. The first section is called **header**, the second section called the **code** (or text) **section** and third called the (static) **data section**. The code section contains the program instructions. The data section contains the static data. 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 correct XEXE format into memory for execution.

Section 4: The eXpOS 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 “hand 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.

The structure of Processes:

The OS associates a **virtual (memory) address space** for each process. The address space of a process is a contiguous sequence of memory addresses staring from zero accessible to a process. The maximum limit on the address space of a process is specified by the constant **XOS_MAXPSIZE**. The OS logically partition the address space into five regions – **library, code, data, stack and heap**. 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 XFS 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.

Apart from the process id and the virtual address space, 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. Hence, they are part of this documentation.

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.

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.

When a process executes a **exec** system call with an executable file name as a parameter, the following are the actions taking place which are relevant to application programs.

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. (what happens to the original address space is irrelevant to the application/system programmer. But it is important to know that any other process sharing code/heap/library regions with this process will not be affected – this is discussed in Section 5.).

2. The code and static data of the executable file are loaded into the code and data 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. (These details are hidden from the application programmer by the compiler if a high level language is used for writing the application.)

From here, execution continues with the newly loaded program.

When a process executes the **fork** system call, the following sequence of events occur.

Semantics of fork operation:

1. A new process child with a new process id and address space is created which is an exact replica of the original process with the same library, code, data, stack and heap regions. (The OS assigns a new process id of 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.** This means, any modification of contents of these regions by one process during subsequent execution will change the other as well. Note that both processes are in concurrent execution subsequent to the fork operation. Data and stack are separate for the child and are 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 get automatically updated. See further discussion in Section 5 . 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.

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. Wait and Signal operations are discussed in more detail in the next section.

Section 5: Synchronization and Access control.

eXpOS assumes a single processor multi programming environment. This means that all processes exist concurrently in the machine and the OS time shares the machine between various processes. The OS specification requires that **round robin scheduling is used with co-operative time-sharing**. This is discussed in Section 6. The OS does not provide any promise to the application program about the order in which processes will be executed.

However, application programs often need to stop and wait for another process to execute certain operations before proceeding. The OS provides system calls that allow user processes to **synchronize execution**.

Process Synchronization

eXpOS provides the **wait** and **signal** system calls for process synchronization. When a process executes the wait system call specifying the process id of another process as argument, the OS puts the calling process to **sleep**. This means, the OS will schedule the process out and won't execute it until one of the following events occur:

1. The process specified as argument terminates by the **exit** system call.

2. The process specified executes a **signal** system call specifying

the original process id as argument.

Access Control

A second major concurrency related requirement is that when multiple processes access the same data (in memory or files), it is often required to have some kind of locking mechanism to ensure that when one processes is accessing the shared data, no other process is allowed to modify the same. This is to ensure data integrity and this issue is called the **critical section problem** .

eXpOS provides (binary) **semaphores** to allow application programs to handle the critical section problem. A (binary) semaphore is conceptually a binary-valued variable whose value can be set or reset only by the OS through designated system call. (Internally how it is implemented is an OS concern and is oblivious to the application programmer).

A process can acquire a semaphore using the **semget** system call, which returns a unique **semid** for the semaphore. The semaphore is initially not set. A process may open several files and semaphores and execute the fork system call multiple times to create an several process that shares all the opened file handles, the heap memory region and all the semaphores acquired by the parent.

Any of these processes sharing a semaphore identifier can set the semaphore (called locking) using the **semlock** system call giving the semid as an argument.

Semantics of Semlock

1. If the semaphore is already locked, the process goes to sleep and wakes up only when the semaphore is reset.
2. Otherwise, lock the semaphore and continue execution.

Semunlock will reset (called release) the semaphore waking up all other processes which went to sleep trying to lock the semaphore.

Just like semaphores, files can also be locked using the **flock** and **funlock** system calls.

Section 6: Scheduler, Exception Handler and Device Interface

Parts of the OS specification which does not fit into other sections of this documentation has been collected here. The topics discussed here are internal to the operating system, but part of the OS specification. Some of these details are hidden from application/system programs and have more hardware dependency than most other parts of this documentation.

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 new loaded code. This code is called the **OS startup code**.

The OS startup code loads the system call routines into memory and sets up the system call interrupt handlers. In addition to these, there are three special modules that are to be loaded – 1. The timer interrupt handler 2.The exception handler(s) and 3. Device interrupt handler(s)

1. **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:

a) 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).

b) There is a disk or I/O device access in a system call which is slow. If wait for the device access is to be avoided, there must be hardware support from the device to send a hardware interrupt upon completion of the device operation so that the OS can 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. Such hardware support is desirable, but not necessary to implement eXpOS.

2. **Exception handler:** If a process generates an illegal instruction, generates 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 **swapper** 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 are equipped with hardware support for demand paging and using the feature can improve machine throughput considerably.

Device Handlers:

eXpOS treats the disk as a special **block device** and assumes that the hardware provides **block transfer instructions** to transfer a disk blocks to memory (pages) and back. The details are hardware specific and not part of the OS specification. The OS system call routines are required to do the interface with the disk device drivers or low level disk routines for handling these.

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 driver routines that that can be invoked by the OS to transfer words. The low level details are hardware specific and are to be handled by the OS routines in an implementation specific way. Here we are concerned about the device interface to application programmers.

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.

The **standard input** and the **standard output** are two special stream devices with predefined identifiers XOS_STDIN and XOS_STDOUT. Standard output only permits **write** and standard input only permits **read**.

Section 7: Application Programmers Interface

Application programmer interacts with the Operating System using system calls. System calls are stored in the disk and loaded into memory when the OS is loaded by bootstrap loader. When a process invokes a system call, the process is interrupted and control goes to the kernel. Therefore, system calls can also be considered as interrupts. Once the system call is carried out, the control goes back to user mode.

There are system calls associated with processes, files and semaphores. In eXpOS, there are three process system calls, nine file system calls and four system calls related to semaphores.

As said before, when the OS is loaded, the first process executed is INIT process. For a user to implement his/her applications, he/she can either create a new process using **Fork** or change the existing process using **Exec**.

Fork system call is used to replicate the process which invoked it. The new process which is created is known as the child and the process which invoked this system call is known as its parent. The heap and the code area will be shared but the stack will get duplicated.

Fork system call

Input: None
Output: Integer value corresponding to the following:

Return Value	**Action**
0- MAX_PROC_NUM -1	Success, the return value is the **process descriptor** of the child process to the parent.
XOS_CFORK_RET	Success, the value returned to the child process.
XOS_PROCLIMIT	Failure, Number of process has reached its maximum, thus cannot create a new one.
XOS_NOFREEPAGE	Failure, No free pages available in memory.

When a process executes the **Fork** system call, the child process shares with the parent all the file handles and semaphores previously opened by the parent. Note that semaphores and the files handles 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.

In order to change the existing process, we use Exec system call.**Exec** loads the executable program into the memory address space of the calling process, overlaying the calling application with the newly loaded program. After the fork system call, if either the parent or the child process loads another program into its virtual address space using the exec system call, then the shared heap is detached from that process and the surviving process will have the heap intact. A successful **Exec** operation results in extinction of the calling application and never returns to the caller application.

Exec system call

Input: Filename which is of the XEXE format. (See Executable file Structure given above)
Output: Integer value corresponding to the following :

Return Value	**Action**
XFS_INVFILNAM	Failure, Filename does not exist or it is not of XEXE type.
XOS_NOFREEPAGE	Failure, Not enough free pages available in memory.
XOS_INVEXEC	Not able to recognize format of the header.

If the user wants to terminate a process, the **Exit** system call is used. **Exit** system call terminates the execution of the process which invoked it and removes it from the memory. It schedules the next ready process and starts executing it. When there is no other ready process to run, it halts the machine.

Exit system call

Input: None
Output: None, The process terminates and it wakes up all the processes waiting for it.

Any process can use multiple resources. Resources can be either files or semaphores. A process can create a new data file in the disk using **Create** system call and delete them using **Delete**. Analogously, a process can create a new semaphore using **Semget** and delete/release the semaphore using **Semrelease**.

The **Create** operation takes as input a filename and creates an empty file by that name. It also creates the root entry for the file and makes sure that every file has a unique name. Note that the file name must be a character string and must not be named "root".

Create system call

Input:A word specifying the file name to be created.

Output: An integer specifying one of the following conditions:

Return Value	**Action**
0	Success. A root entry created with the file name, size=0 and type=DATA.
XFS_NOENTSP	No space for new entry in the root file.
XFS_NODSKSP	No space in disk for a new file.
XFS_INVFILNAM	File name already exists or invalid.

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.

Delete system call

Input: A word specifying the name of the file to be deleted.
Output: Integer value corresponding to the following:

Return Value	**Action**
0	Success
XFS_INVFILNAM	Invalid file name or filename is root.
XFS_FILOPEN	There is some application that has not closed this file.

Semget does not take any input and gives the semaphore handle as the output. eXpOS has a fixed number of semaphores defined by MAX_SEM_COUNT. An error code is produced if there are no available semaphores in the system.

Semget system call

Input: None
Output: Integer value corresponding to following values:

Return Value	Action
0- MAX_SEM_COUNT	Success, returns the ID of free semaphore
XOS_SEMLIMIT	Failure, no free semaphore available
XOS_RESFULL	Process has reached its maximum limit of resources.

Semrelease releases the semaphore so that it can be reused by another process.

Semrelease system call
Input: semaphore id
Output : Integer value corresponding to following values:

Return Value	Action
0	Success
XOS_INVSEM	Failure, semaphore ID does not exist

For a process to read/write a file, it must first open the file. The **Open** operation returns a file descriptor. The file descriptor conceptually refers to the location of the file which will be referred to by other file operations. The file pointer initially points to the first word in the file when **Open** system call is completed. An application can open the same file several times and each time a different descriptor will be returned by the **Open** operation.

Open system call
Input:A word specifying the name of the file to be opened.
Output: An integer value specifying one of the following:

Return Value	Action
0 - XFS_MAXFILDES	Success, the return value is the **file descriptor** for the opened file.

Return Value	Action
XFS_INVFILNAM	Invalid file name. (file is not a data file or the root file).
XFS_INVFILDES	Invalid file descriptor. (Process has reached its maximum limit of open files – this limit is set by the Operating System.)

After opening the file, if the user wants to read the file, **Read** system call can be used. **Read** reads one word into a string/integer variable from the position pointed by file pointer in the file specified by the input file descriptor. After each read is performed, the file pointer advances to the next word in the file. Thus file access is sequential.

Read system call
Input: 1) An integer specifying the file descriptor and
2) a memory address for storing the word read.
Output: An integer value specifying the following:

Return Value	Action
0	Success. The next word in the file is copied into the memory address and file descriptor advances to the next word in the file.
XFS_INVFILDES	Invalid file descriptor.
XFS_INVBUFFER	Invalid memory reference.

If the user wants to write into the file, **Write** system call is used. **Write** writes one word from a string/integer variable provided by the user, to the position pointed by input file pointer. After each write is performed, the file pointer advances to the next word in the file.

Write system call
Input: 1) An integer specifying the file desciptor
2) Memory address of the word to be written into the file.
Output: An integer value specifying the following:

Return Value	Action
0	Success
XFS_INVFILDES	Invalid file descriptor.
XFS_INVBUFFER	Invalid memory reference.
XFS_NODSKSP	Beyond maximum size permitted for a file or no disk space.

If user wants to change the file pointer without reading/writing, **Seek** system call is used. The **Seek** operation allows the application program to change the location pointed to by the file pointer from the current position to a different position in the file so that next Read/Write is performed from the new position.

Seek system call

Input : 1) An integer specifying the file descriptor

2) An integer offset specifying the number of words ahead/behind the file pointer has to shift. (A negative offset shits backwards).

Output: An integer value specifying the following:

Return Value	Action
0	Success.
XFS_INVFILDES	Invalid file descriptor.
XFS_INVINT	Invalid offset – beyond file limits.

After all the operations are done, the user closes the file using **Close** system call.

Close system call

Input:An integer corresponding to the file descriptor.

Output: Integer value corresponding to the following:

Return value	Action
0	File closed successfully.

Return value	Action
XFS_INVFILDES	Invalid file descriptor.

To prevent concurrency issues, the process can lock (or unlock) its resources. The **FLock** and **FUnlock** are used for files while **SemLock** and **SemUnlock** are used for semaphores.

The Lock operations (**FLock** and **SemLock**) allow an application program to lock a resource (file or semaphore) so that other applications running concurrently are not permitted to access the resource till it is unlocked. Any operation on a resource already locked by another process will cause the process invoking the operation to be blocked (i.e. Process will go to sleep) till the resource is unlocked by the process that had locked it.

FLock system call
Input : The file descriptor of the file to be locked.
Output : An integer value specifying the following:

Return Value	Action
0	Success
XFS_INVFILDES	Invalid file descriptor.

Semlock system call
Input: semaphore id
Output : Integer value corresponding to following values:

Return Value	Action
0	Success
XOS_INVSEM	Failure, semaphore ID does not exist

The Unlock operation (**FUnlock** and **SemUnlock**) allow an application program to unlock a resource the application had locked earlier so that other applications are no longer restricted from accessing the resource. Applications waiting for the resource to be unlocked is unblocked by the unlock operation.

FUnlock system call

Input: The file descriptor.
Output: An integer value specifying the following:

Return Value	**Action**
0	Success or the file is already unlocked.
XFS_INVFILDES	Invalid file descriptor.
XFS_NOLOCK	File is not locked by the current process

SemUnlock system call

Input: semaphore id
Output : Integer value corresponding to following values:

Return Value	**Action**
0	Success
XOS_INVSEM	Failure, semaphore ID does not exist

As mentioned before, eXpOS provides two system calls **wait** and **signal**, to implement process synchronization (Read Access control and synchronization).

Wait System Call

The current process is blocked till the process with PID given as argument executes a Signal system call or exits.

Input:ProcessId
Output: Integer value corresponding to following values:

Return Value	**Action**
0	Success
XOS_INVPID	Process does not exist or Process intends to waits

Return Value	Action
	for itself.
XOS_NOLOCK	File is not locked by the current process

Signal System Call :

All processes waiting for the current process are resumed.
Input: None

Output: 0 indicating success