Enhancement to eXpOS Operating System and eXpFS File System

Kruthika Suresh Ved B110300CS
Sikha V Manoj B110572CS
Sonia V Mathew B110495CS
Guided by: Dr.K.Muralikrishnan

March 1, 2015

Abstract

Project eXpOS or experimental Operating System is a educational platform to develop an operating system. It is an instructional tool for students to learn and implement OS data structures and functionalities on a simulated machine called XSM (eXperimental String Machine).The OS is programmed using a custom language known as SPL (System Programmer's Language) and application programs, which run on the OS, are programmed using eXpL (Application Programmer's Language).

1 Problem Definition

This project aims to modify and update the eXpOS operating system 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

Thus, adding the above features, eXpOS would become more advanced and closer to operating systems that are available in the market.

2 eXpOS Specification

eXpOS 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 OS specification is prepared in a manner independent of programming language and target machine.

2.1 eXperimental File System (eXpFS)

eXpOS uses eXpFS (eXperimental File System) which contains files organized into a single directory called the root. There are three types of eXpFS files: the root, data files and executable files. The root is also treated conceptually as a file.

2.1.1 Root

The root file has name **root** and contains information about the files stored in the file system. For each file stored in eXpFS, the root stores three words of information: file-name, file-size and file-type. This triple is called the root entry for the file. The first root entry is for the root itself.

2.1.2 Data File

A data file is a sequence of words. eXpFS expects the Operating System to display data files with an extension .dat. eXpFS treats this as a default file type, hence the application programs do not have to specify the extension .dat at the time of file creation. The operations allowed in data files are Create, Delete, Open, Close, Lock, Unlock, Read, Write, Seek.

2.1.3 Executable File

Executable files are essentially program files that must be loaded and run by the operating system.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 - Header and Code. In this implementation of eXpOS, static data is stored in stack pages.

2.2 Process Model

A program under execution is called a process. The eXpOS associates a virtual (memory) address space for each process. The eXpOS logically partitions the address space into four regions: library, code, stack and heap. These regions are mapped into physical memory using hardware mechanisms like paging/segmentation.

2.3 InterProcess Communication

eXpOS assumes a single processor multi programming environment. It allows processes to communicate with each other using mechanisms like semaphores and WaitSignal system calls. eXpOS provides (binary) semaphores to allow application programs to handle the critical section problem. eXpOS provides system calls like Semget, SemLock, SemUnlock, Semrelease for working with semaphores.

2.4 Shared Memory Model

Shared memory is an efficient means of sharing data between programs. In eXpOS, this sharing is done between the parent process and child pro-

cesses through heap. It is the responsibility of the programmer to ensure exclusive access to the shared resources for each process, to avoid data inconsistency. eXpOS helps programmer to realize data consistency with the help of semaphores.

2.5   System Calls

When a process invokes a system call, the process is interrupted and control goes to the kernel. Once the system call is carried out, the control goes back to user mode. There are system calls associated with processes, files and semaphores.
The following system calls are present in the system:

- File System Calls : Create, Delete, Open, Close, Read, Write, Seek
- Process System Calls : Fork, Exec, Exit, Getpid, Getppid, Shutdown
- System calls for access control and Synchronization: Wait, Signal, FLock, FUnLock, Semget, Semrelease, SemLock, SemUnLock.

2.6   Pre-Emptive Scheduling

In Pre-Emptive Scheduling, process can be paused before its time slice is over. This usually happens when a resource that the process requires is not available at the present. It puts itself to sleep and another process can be scheduled for execution.

2.7   Asynchronous disk operations

To minimize processor cycles spent on disk operations, disk operations were made asynchronous. This means that while a disk operation is being carried out, other processes which do not require the disk can be executed.

3   Design of eXpOS

3.1   Data Structures

It can be divided into - Disk Data structures and Memory (in-core) data structures. A copy of Disk Data Structures will be kept in the memory while the system is on, for reducing disk access.

3.2   Disk Data Structures

3.2.1   Inode Table

Inode Table is the record of the files stored in the disk. The entry of an Inode table has the following format:

3.2.2   Disk Free List

For each block in the disk there is an entry in the Disk Free List which contains a value of either 0 or 1 indicating whether the corresponding block in the disk is free or used, respectively.

FILE TYPE	FILE NAME	FILE SIZE	DATA BLOCK 1	DATA BLOCK 2	DATA BLOCK ..	DATA BLOCK n

Figure 1: Structure of the Inode Table

3.3 Memory Data Structures

3.3.1 Process Table

The Process Table contains an entry for each process. Each entry contains several fields that stores all the information pertaining to a single process and the maximum number of entries is equal to maximum number of processes allowed to exist at a single point of time in eXpOS.

TICK	PID	PPID	STATE	MACHINE STATE	PTBR	PTLR	PER-PROCESS RESOURCE TABLE	INODE INDEX	KERNEL STACK POINTER

Figure 2: Structure of the Process Table

The first entry Tick keeps track of how long the process was in memory. The second column is PID or Process ID which is the unique process descriptor, a number that is unique to each process.

The third column gives the process descriptor of the parent process or the PPID.

Next column, State , consists of a two tuple that describes the current state of the process.

The fifth column is the pointer to a structure that gives the Machine State when the process was last executed. This part is machine dependent.

The next two columns are regarding page table of the process. The first one (PTBR or Page Table Base Register) stores the starting address of the page table of a process while the next one (PTLR or Page Table Length Register) stores the number of entries in the page table of a process and determines the size of the virtual address space of the process.

The next column is a pointer to a table, the Per-Process Resource Table that contains information about the files opened by the process as well as semaphores used by the process.The index of the per-process resource table entry for each resource is known as the resource descriptor.

Inode Index is a reference to the Inode entry of the executable file. It could be used to access the code pages of the process.

Each process has its own kernel stack whose pointer is given in the Kernel Stack Pointer column. A process uses its kernel stack to save the return address when it decides to voluntarily schedule out itself out when entering a blocking system call.

4

3.3.2 File Table

File Table has information about all the files that are currently open. It is initialized by the OS startup code. Initially, there are no open files.

Pointer to the Inode entry	Number of Open Instances of the file	PID of the process which has currently locked the file

Figure 3: Structure of the File Table

3.3.3 Semaphore Table

Semaphore Table contains details about all the semaphores used by the processes. For every semaphore opened by a process, there is an entry in the per-process resource table (which is discussed later), and this entry points to a corresponding entry in the semaphore table.

PID of the process locking the semaphore	Number of processes using the semaphore

Figure 4: Structure of the Semaphore Table

3.3.4 Disk Status Table

eXpOS makes use of Disk Status Table to keep track of load and store operations. It consists of four entries which is given in Figure 4.

LOAD/STORE BIT	PAGE NUMBER	BLOCK NUMBER	PID of the process who invoked the device

Figure 5: Structure of the Disk Status Table

3.3.5 Buffer Table

To minimize the number of load and store operations, eXpOS provides a buffer cache in memory which would temporarily store disk blocks. Each disk block is mapped to a unique buffer. The disk blocks will be stored

back to the disk when some other blocks replace it. Only modified blocks are written back to disk. Buffer Table keeps the information about disk blocks present in the buffer.

Block number of the disk block stored in buffer	DIRTY BIT which indicates whether the block stored in the buffer has been modified	The PID of the process which has currently locked the buffer.

Figure 6: Structure of the Buffer Table

3.3.6   System Status Table

System Status Table keeps the information about number of free pages in memory (Mem_free_count), number of processes waiting for memory pages (Wait_mem_count) , and also the number of processes in swapped state (Swapped_count).

3.3.7   Memory Free List

The Memory free list is a data structure used for keeping track of used and unused pages in the memory. Each entry of the free list contains a value of either 0 indicating whether the corresponding page in the memory is free or number indicating the number of processes that share the page.

3.4   Algorithms

3.4.1   File System Calls

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.

Arguments : Filename

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

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.

Arguments : Filename

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

6

Algorithm 1 Create system call

---

if file is present in Inode Table **then**
    return 0
end if
if no free entry in Inode Table **then**
    return -1
else
    store index of free entry in InodeIndex
end if
In the Inode Table entry corresponding to InodeIndex, set file name to Filename, file size to 0 and file type to DATA .
In the root file entry corresponding to InodeIndex, set file name to Filename, file size to 0 and file type to DATA .
return 0

Algorithm 2 Delete system call

---

if file is not present in Inode table **then**
    return -1
else
    store index of file in InodeIndex
end if
if file is open **then**
    return -2
end if
Free all the blocks allocated to the file
Invalidate the Inode Table entry corresponding to Inodeindex
Remove root file entry corresponding to Inodeindex
return 0

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).

Arguments : Filename

Return Value : 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)

Algorithm 3 Open system call

---

  if file is not found in Inode Table **then**
    return -1
  end if
  if file type is not DATA or ROOT **then**
    return -1
  end if
  if no free entry in Per-Process Resource table **then**
    return -2
  else
    store index of free entry in *PerProcessIndex*
  end if
  if file is already open **then**
    store index of file table entry in *FTIndex*
  else
    if no free entry in FT **then**
      return -2
    else
      store index of free entry in *FTIndex*
    end if
  end if
  In entry corresponding to *PerProcessIndex*, set pointer to FT as *FTIndex* and LSEEK as 0
  In entry corresponding to *FTIndex*, set pointer to Inode Table as *InodeIndex*
  Increment file open count, set lock status to free
  return *PerProcessIndex*

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.

Arguments : File Descriptor

Return Value : 0 (Success) or -1 (File Descriptor is invalid)

Algorithm 4 Close system call

---

if File Descriptor is not valid **then**
   return -1
end if
if entry in Per-Process Resource table is not valid **then**
   return -1
else
   store the pointer to FIle Table in *FTIndex*
end if
Decrement file open count
if file is locked by current process **then**
   unlock the file
end if
if file open count becomes zero **then**
   Invalidate the File Table entry
end if
Invalidate PerProcess Table entry
return 0

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.

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

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

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.

Arguments :File Descriptor and a Word to be written

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

Algorithm 5 Read system call

___

   if File Descriptor is not valid **then**
      return -1
   end if
   if entry in Per-Process Resource table is not valid **then**
      return -1
   else
      store the pointer to FIle Table in *FTIndex*
      store LSEEK in *lseek*
   end if
   while file is locked **do**
      put the current process to sleep
      call scheduler
   end while
   lock the file
   store pointer to Inode Table in *InodeIndex*
   if file pointer is at the end of the file **then**
      return -2
   else
      store *lseek*/XFS_MAXBSIZE in *BlockNum*
      store *lseek*%XFS_MAXBSIZE in *Offset*
   end if
   find the buffer to which the block is mapped
   if buffer is locked **then**
      if current process has not locked the buffer **then**
         while buffer is locked **do**
            put the process to sleep
            call scheduler
         end while
         lock the buffer
      end if
   else
      lock the buffer
   end if
   if buffer does not have required block **then**
      if buffer contains a block and dirty bit is set **then**
         store the block in buffer to disk
      end if
      load the required block to the buffer
   end if
   Read the data
   Increment the file pointer
   unlock the buffer and wake all processes waiting for the buffer
   Unlock the file and wake all processes waiting for the file
   return 0

___

Algorithm 6 Write system call

---

if File Descriptor is not valid **then**
    return -1
end if
if entry in Per-Process Resource table is not valid **then**
    return -1
else
    store the pointer to FIle Table in *FTIndex* and LSEEK in *lseek*
end if
while file is locked **do**
    put the current process to sleep and call scheduler
end while
lock the file
store pointer to Inode Table in *InodeIndex*
store *lseek*/XFS_MAXBSIZE in *BlockNum* and *lseek*%XFS_MAXBSIZE in *Offset*
if entry corresponding to *BlockNum* is invalid **then**
    if no free block in disk **then**
        return -2
    else
        allocate a free block to the file
        increment file size in Inode Table and root file
    end if
end if
find the buffer to which the block is mapped
if buffer is locked **then**
    if current process has not locked the buffer **then**
        while buffer is locked **do**
            put the process to sleep
            call scheduler
        end while
        lock the buffer
    end if
end if
lock the buffer
if buffer does not have required block **then**
    if buffer contains a block and dirty bit is set **then**
        store the block in buffer to disk
    end if
    load the required block to the buffer
end if
Write the data
unlock the buffer and wake all processes waiting for the buffer
Unlock the file and wake all processes waiting for the file
return 0

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.

Arguments : File Descriptor and Offset

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

Algorithm 7 Seek system call

---
if File Descriptor is not valid **then**
   return -1
end if
if entry in Per-Process Resource table is not valid **then**
   return -1
else
   store the pointer to FIle Table in *FTIndex*
   store LSEEK in *lseek*
end if
while file is locked **do**
   put the current process to sleep
   call scheduler
end while
lock the file
store pointer to Inode Table in *InodeIndex*
if new file pointer is not valid **then**
   return -2
end if
if Offset is zero **then**
   set LSEEK to beginning of file
else
   set LSEEK to LSEEK+Offset
end if
Unlock the file and wake all processes waiting for the file
return 0

3.4.2 Process System Calls

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.

Arguments : None

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

Algorithm 8 Fork system call

---

if no free entry in process table **then**
  return -1
else
  store index of free entry in *ChildPID*
  store pid of parent process in *ParentPID*
end if
set the PPID field of child process to *ParentPID*
count the number of stack pages of parent
while equal number of free pages are not present in memory **do**
  put the process to sleep
  call scheduler
end while
Allocate free pages to the child
Copy the parent's stack to child's stack
Copy the page table entries of parent, except stack entries, to the page table of child
Copy the machine state and perprocess resource table to child
For every open file of the parent, increment the file opern count
For every semaphore acquired by the parent, increment process count
set state of child to ready
return 0 to the child process and *ChildPID* to the parent process

---

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.

Arguments : Filename

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

Algorithm 9 Exec system call

---

  if file not found in Inode Table **then**
    return -1
  else
    if file type is not EXEC **then**
      return -1
    else
      store index of Inode Table entry in *InodeIndex*
      store the ode block numbers of the file in *Block1* and *Block2*.
    end if
  end if
  In the page table of current process, set code page entries to *Block1* and *Block2*.
  Include the page numbers of shared library in the page table.
  Set the auxillary information to invalid and unreferenced.
  Invalidate the entry for heap pages
  Close all files opened by the current process
  Release all semaphores held by the parent process.
  Set SP and IP values to valid locations.
  return 0

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.

Arguments : None

Return Value : -1 (Failure)

Getpid System Call

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

Arguments : None

Return Value : Process Identifier (Success)

Algorithm 10 Exit system call

  if no more processes to schedule **then**
    shutdown the machine
  else
    store the pid of the next ready process in *NextPID*
  end if
  Close all files opened by the current process
  Release all the semaphores used by the current process
  Memory pages of the current process are freed
  Invalidate the page table entry
  Wake up all processes waiting for the current process
  Schedule the process with pid *NextPID*
  return

Algorithm 11 Getpid system call

  Find the PID of the current process by using PTBR value.
  return PID of current process

Getppid System Call

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

Arguments : None

Return Value : Process Identifier (Success)

Algorithm 12 Getppid system call

  Find the PID of the current process by using PTBR value.
  From the Process Table entry of the current process, find the PPID
  return PPID of current process

Shutdown System Call

Shutdown system call terminates all processes and halts the machine.

Arguments : None

Return Value : None

Algorithm 13 Shutdown system call

---
while disk is not free **do**
   put the process to sleep
   call scheduler
end while
store Inode Table to the disk
store dirty pages to disk
store Disk Free List to the disk
Halt the machine
return

3.4.3   System calls for 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. Note that the system call will fail if a process attempts to wait for itself.

Arguments : Process Identifier of the process for which the current process has to wait.

Return Value : 0 (Success) or -1 (Given process identifier is invalid or it is the pid of the invoking process)

Algorithm 14 Wait system call

---
if process is intending to wait for itself or for a terminated process **then**
   return -1
end if
put the current process to sleep
call scheduler
return 0

Signal System Call

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

Arguments : None

Return Value : 0 (Success)

Algorithm 15 Signal system call

---
wake up all processes waiting for the current process
return

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.

Arguments : File Descriptor

Return Value : 0 (Success) or -1 (File Descriptor is invalid)

Algorithm 16 FLock system call

---

if File Descriptor is not valid **then**
  return -1
end if
if entry in Per-Process Resource table is not valid **then**
  return -1
else
  store the pointer to FIle Table in *FTIndex*
end if
if file is locked **then**
  if current process has not locked the file **then**
    while file is locked **do**
      put the current process to sleep
      call scheduler
    end while
    lock the file
  end if
else
  lock the file
end if
return 0

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.

Arguments : File Descriptor

Return Value : 0 (Success) or -1 (File Descriptor is invalid) or -2 (File was not locked by the calling process)

Algorithm 17 FUnLock system call

---

if File Descriptor is not valid **then**
    return -1
end if
if entry in Per-Process Resource table is not valid **then**
    return -1
else
    store the pointer to FIle Table in *FTIndex*
end if
if file is locked **then**
    if current process has locked the file **then**
        unlock the file
        wake up all the processes waiting for the file
    else
        return -2
    end if
end if
return 0

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.

Arguments : None

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

Algorithm 18 Semget system call

---

if no free entry in perprocess resource table **then**
    return -1
else
    store the index of the free entry in *PerProcessIndex.*
end if
if no free entry in semaphore table **then**
    return -2
else
    store index of free entry in *STIndex.*
    increment process count
end if
Store *STIndex* in the perprocess table entry corresponding to *PerProcessIndex.*
return *STIndex*

18

Semrelease System Call

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

Arguments : Semaphore Descriptor

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

Algorithm 19 Semrelease system call

---

if Semaphore Descriptor is not valid **then**
   return -1
end if
if process has not acquired the semaphore **then**
   return -1
end if
if semaphore is locked by current process **then**
   unlock the semaphore
end if
decrement the process count of semaphore
invalidate the perprocess resource table entry
return 0

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.

Arguments : Semaphore Descriptor

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

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.

Arguments : Semaphore Descriptor

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

Algorithm 20 SemLock system call

---

  if Semaphore Descriptor is not valid **then**
    return -1
  end if
  if process has not acquired the semaphore **then**
    return -1
  end if
  if semaphore is locked **then**
    if current process has not locked the semaphore **then**
      while semaphore is locked **do**
        put the current process to sleep
        call scheduler
      end while
      lock the semaphore
    end if
  else
    lock the semaphore
  end if
  return 0

Algorithm 21 SemUnLock system call

---

  if Semaphore Descriptor is not valid **then**
    return -1
  end if
  if process has not acquired the semaphore **then**
    return -1
  end if
  if semaphore is locked by current process **then**
    unlock the semaphore
    wake up all processes waiting for the semaphore
  else
    return -2
  end if
  return 0

3.4.4 Miscellaneous

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 condi-
tions which are machine dependent) or if a page fault occurs, 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 con-
tinue round robin scheduling the remaining processes.

Arguments : None

Return Value : None

Algorithm 22 Exception Handler

---

  if exception is not caused by page fault **then**
    display the cause and exit the process
  end if
  if Exception Page Number register is -1 **then**
    display error XOS_INVPAGENUM and exit the process
  else
    store the logical page number causing exception in *LogicalPage*.
    while no free page in memory **do**
      put the current process to sleep
      call scheduler
    end while
    store a free page number in *FreePage*.
    if page corresponding to *LogicalPage* is in disk **then**
      load the block to memory, to *FreePage*.
    end if
    Set the page table entry corresponding to *LogicalPage* with *FreePage*.
    set the auxiliary information as referenced and valid
  end if
  return

Disk Interrupt Handler

Once a disk transfer is completed, the disk controller produces an interrupt to let the processor know that the disk transfer is done. Disk Interrupt Handler wakes up all processes that went to sleep waiting for the disk while the transfer was going on.

Arguments : None

Return Value : None

Algorithm 23 Disk Interrupt Handler

---

set disk status register to 0
if disk transfer was initiated by a process **then**
  wake up all processes waiting for the disk
end if
if disk transfer was initiated by scheduler **then**
  if disk operation was store **then**
    free the stored memory page
  else
    if loaded block is in swap area **then**
      free the loaded block
    end if
  end if
end if
return

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. This scheduler uses Round Robin scheduling to schedule the next process for execution.

Arguments : None

Return Value : None

Algorithm 24 Timer Interrupt Handler

Find the next process to be scheduled and the senior most swapped process
Save the context of the current process
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 there are swapped processes **then**
        for the senior most swapped process, find the block which has to be swapped in
        find a free page in memory
        load the block from disk to free page in memory
        set the state of the process to ready
      end if
    end if
else
    if there are sleeping processes that requires memory pages **then**
      use second chance algorithm to find an unreferenced page
      if the unreferenced page is a stack or heap page **then**
        find a free block in swap area
        store the page in the swap block
        set the page table entry to the swap block number and auxillary information to unreferenced and invalid
        if stack page is needed for execution by the process **then**
          set the state of the process to swapped
        end if
      end if
    end if
end if
schedule the next ready process
return

4   Work Done

- The existing OS data structures were redesigned to incorporate the changes done.
- New data structures like Buffer Table, Disk Status Table, Semaphore Table, System Status Table etc were designed.
- System calls were redesigned to incorporate asynchronous operations, buffer cache and pre-emptive scheduling.
- Directory structure was introduced in eXpFS.
- Executable file format was designed.
- Algorithms for system calls and other interrupt handlers were designed.
- Webpage for eXpOS was created. *http* : *//exposnitc.github.io/*

5   Future Work

Future work includes implementation of all the features mentioned above, testing of the system and building a roadmap for anyone wishing to build eXpOS.

6   Conclusion

This project aims to create a simpler version of an operating system which allows students to acquire insight into the working of a real operating system.

References

[1] *http* : *//xosnitc.github.io/*

[2] The Design of Unix Operating System, By Maurice J. Bach