Enhancement to XOS Operating System and XFS File System

- Kruthika Suresh Ved
Sikha V Manoj
Sonia V Mathew
Guided by: Dr.K.Muralikrishnan

What is eXpOS?

- Simpler version of OS
- Multiprogramming
- Educational tool to familiarize students with the working of OS

Work done during last semester

- Redesigned the XFS system
- Included Inter process communication
- Redesigned process model
- Redesigned existing data structures and added new ones.

Work done

- Algorithms for system calls were designed.
- Modifications were made to the existing system calls to incorporate asynchronous disk access
- New system calls for synchronization were also added.
- A new website for eXpOS had been created to guide the students through the educational tool.

File System Calls

- Create
- Delete
- Open
- Close
- Read
- Write
- Seek

Create System Call

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 .
Increment the root file size
return 0

Delete System Call

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
Decrement the file size of root file
return 0
```

Open System Call

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 File Table then
        return -2
    else
        Store index of free entry in FTIndex
    end if
end if
In entry corresponding to PerProcessIndex, set pointer to File Table 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

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
if file is locked by current process then
    Unlock the file
end if
Decrement file open count
if file open count becomes zero then
    Invalidate the File Table entry
end if
Invalidate Per-Process Table entry
return 0
```

Read System Call

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
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
  end if
end if
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/block-size in BlockNum
  Store lseek%block-size in Offset
end if
if buffer to which the block is mapped 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
  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
Read the data and 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
```

Write System Call

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
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 and call scheduler
    end while
  end if
end if
Lock the file
Store pointer to Inode Table in InodeIndex
Store lseek/block-size in BlockNum and lseek%block-size in Offset
if entry in Inode Table 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
if buffer to which the block is mapped is locked then
  if current process has not locked the buffer then
    while buffer is locked do
      Put the process to sleep and call scheduler
    end while
  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 and increment 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
```

Seek System Call

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
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
   end if
end if
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
```

Process System Calls

- Fork
- Exec
- Exit
- GetPid
- GetPPid
- Shutdown

Algorithm 11 Getpid system call

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

return PID of current process

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

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

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 one free page to the child for each stack page of the parent
Copy the parent's stack to child's stack
Copy the page table entries, except stack entries, of parent to the page table of child
Copy the parent's machine state and Per-Process resource table to the child
Copy the inode index from parent to child
For every open file of the parent, increment the file open 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
```

Algorithm 9 2. 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 code 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*.
Set the auxiliary information of code pages to invalid and unreferenced.
Include the page numbers of shared library in the page table.
Invalidate the entry for heap pages
In the process table of current process, set the pointer to Inode Table as *InodeIndex*
Close all files opened by the current process
Release all semaphores held by the current process.
Set SP and IP values to valid locations.
return 0

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

System calls for access control and synchronization

- Wait
- Signal
- Flock
- FunLock
- Semget
- Semrelease
- SemLock
- SemUnLock

Wait and Signal

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
```

Algorithm 15 Signal system call

```
Wake up all processes waiting for the current process
return
```

FLock and FUnLock

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
  end if
end if
lock the file
return 0
```

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 and Semrelease

Algorithm 18 Semget system call

```
if no free entry in Per-Process 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 Per-Process table entry corresponding to PerProcessIndex.
return STIndex
```

Algorithm 19 Semrelease system call

```
if Semaphore Descriptor is not valid then
    return -1
end if
if entry in Per-Process Resource table is not valid 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 Per-Process resource table entry
return 0
```

SemLock

Algorithm 20 SemLock system call

```
if Semaphore Descriptor is not valid then
  return -1
end if
if entry in Per-Process Resource table is not valid 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
  end if
end if
Lock the semaphore
return 0
```

SemUnLock

Algorithm 21 SemUnLock system call

```
if Semaphore Descriptor is not valid then
   return -1
end if
if entry in Per-Process Resource table is not valid 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
```

Miscellaneous System Calls

- Timer interrupt
- Exception Handler
- Disk Interrupt Handler

Timer Interrupt

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 code page then
      Using pointer to Inode Table, find the corresponding code block
      Set the page table entry of the selected page to the code block number
      Set auxiliary information to unreferenced and invalid
    end if
    if the unreferenced page is a stack or heap page then
      Find a free block in swap area
      Store the selected page in the swap block
      Set the page table entry to the swap block number and auxiliary information to unreferenced
      and invalid
      if selected page is the stack page where process stopped execution then
        Set the state of the process to swapped
      end if
    end if
  end if
end if
Schedule the next ready process
return
```

Exception Handler

Algorithm 22 Exception Handler

```
if exception is not caused by page fault then
    Display the cause and exit the process
end if
if reference to an invalid address was made then
    Display the error 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 FreePage in memory.
    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

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
```

Future Work

- The algorithms that has been designed has to be implemented and its compliance with the OS specification and XSM architecture has to be tested for.
- Addition of more features like directory structure, user and super user , etc.

Thank You :)