Enhancement to XOS educational operating system and XFS file system

Guide : Dr. K. Muralikrishnan

Members:	Kruthika Suresh Ved	B110300CS
	Sikha V Manoj	B110572CS
	Sonia V Mathew	B110495CS

What is eXpOS ?

- Educational platform to develop an OS.
- Undergraduate student can implement in one semester.
- Better insight of Operating System concepts.
- Minimum external supervision required.

Why eXpOS ?

- Previous version – XOS
- XOS has limited set of features
- Following features were added:
  - Non blocking disk access
  - Access Control
  - Dynamic Memory Allocation

Objective

- Specification,
- Design, and
- Documentation

of the eXpOS Kernel.

Specification

eXpFS File System

- File - a continuous stream of data
- Files organized in a single directory - root
- Types of eXpFS files - root, data and executable

Root File

- Root entry - file name, file-size and file-type
- Open, Close, Read and Seek

Data File

- Sequence of words
- Create, Delete, Open, Close, Read, Write, Seek, FLock and FUnLock

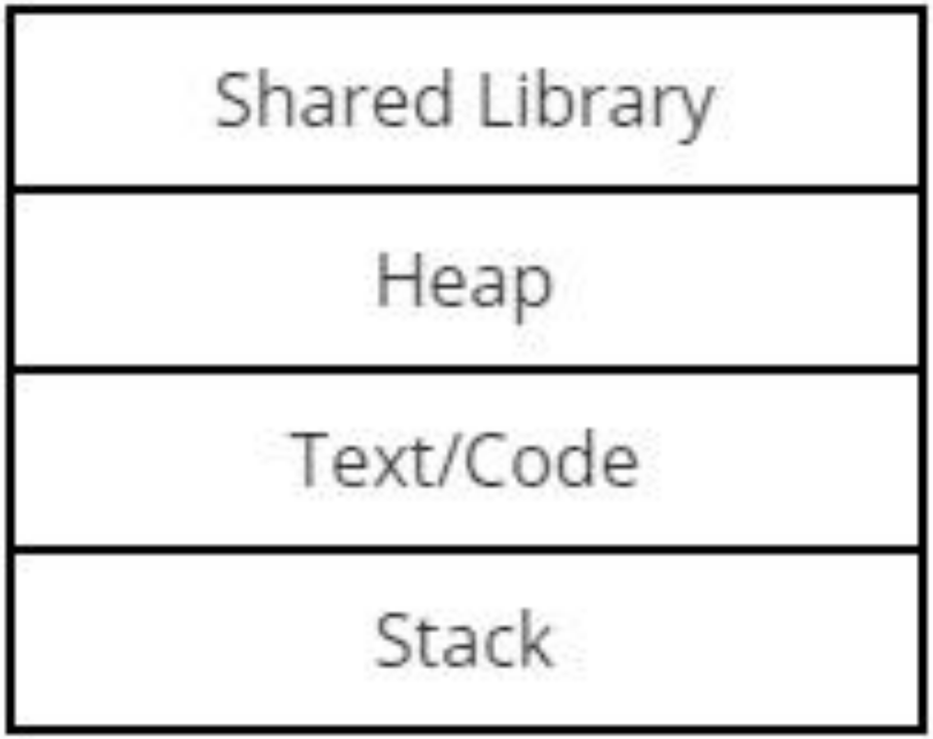
Executable File

- Cannot be created by application programs
- Loaded externally to disk
- XEXE format - header and the code section
- Cannot be read/written by user program

Process

- Process created when an existing process invokes Fork
- Init process created by OS startup code
- Unique process id for each process
- A process may open files or semaphores
- Fork, Exec, Exit, Wait, Signal, Getpid, Getppid.

Process - Virtual memory model



Synchronization and access control

- Synchronize execution – Wait and Signal
- Access control – file locks and binary semaphores
- FLock, FUnLock
- Semget, Semrelease, SemLock, SemUnLock

Hardware Interrupt Handlers

- Timer Interrupt Handler
  - Co-operative round robin scheduling
- Disk Interrupt Handler
  - Disk – block device
  - Disk controller raises the interrupt
- Terminal Interrupt Handler
  - Stream device
  - STDIN and STDOUT

System Call Interface

- Interface provided by kernel to application programmer
- File system calls, Process system calls and system calls for access control and synchronization
- System calls in XOS
  - Create, Delete, Open, Close, Read, Write, Seek
  - Fork, Exec, Exit, Getpid, Getppid
  - Wait, Signal
- System calls added
  - Shutdown
  - FLock, FUnLock
  - Semget, Semrelease, SemLock, SemUnLock

Design

- Data Structures
- Algorithms

Disk Data Structures

- Inode Table

FILE TYPE	FILE NAME	FILE SIZE	DATA BLOCK 1	DATA BLOCK 2	DATA BLOCK 3	DATA BLOCK 4	Unused

- Disk Free List
    - Information about the disk block usage, 512 entries

- Root File

FILE NAME	FILE SIZE	FILE TYPE	Unused

Memory Data Structures

•File Table

INODE INDEX	FILE OPEN COUNT	ULOCK	KLOCK	Unused

•Semaphore Table

LOCKING PID	PROCESS COUNT	Unused

•Buffer Table

BLOCK NUMBER	DIRTY BIT	LOCKING PID	Unused

- System Status Table

MEM_FREE_COUNT	WAIT_MEM_COUNT	SWAPPED_COUNT	Unused

- Terminal Status Table

STATUS	PID	Unused

- Disk Status Table

STATUS	LOAD/STORE BIT	PAGE NUMBER	BLOCK NUMBER	PID	Unused

- Memory Free List
  - Information about memory page usage, 512 entries

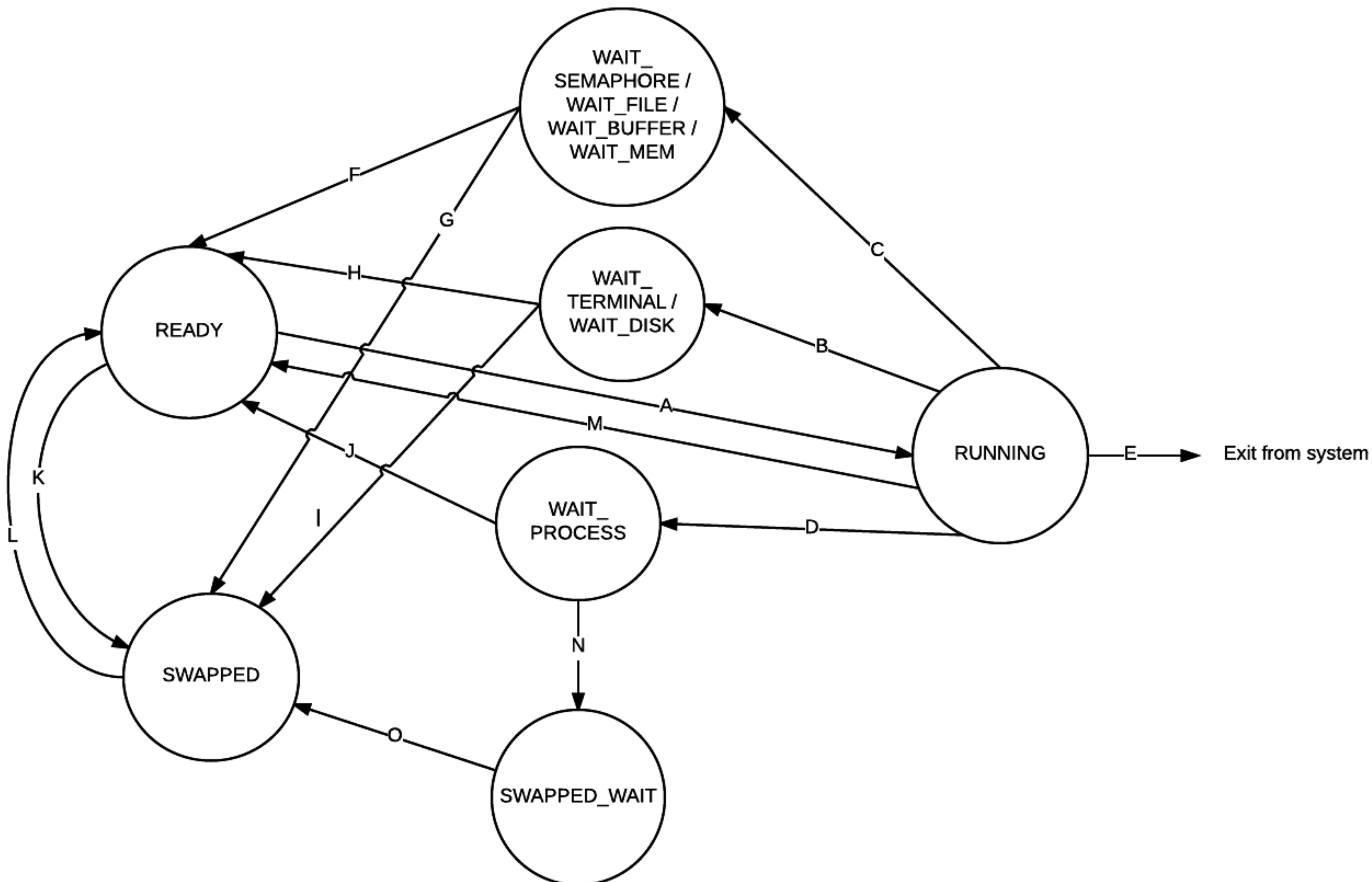
- Process Table

TICK	PID	PPID	STATE	PER-PROCESS RESOURCE TABLE	INODE INDEX	INPUT BUFFER	MODE FLAG	KERNEL RE-ENTRY POINT	PTBR	MACHINE STATE POINTER	Unused

- Machine State

SP	BP	IP	PTBR	PTLR	REGISTERS	Unused

State Transitions



- Per-Process Resource Table

Index of File Table/ Semaphore Table entry (1 word)	LSEEK (1 word)

- Per-Process Page Table

PHYSICAL PAGE NUMBER	REFERENCE BIT	VALID BIT	WRITE BIT

Create System Call

Algorithm 1 Create System Call

Input: Filename

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

If the file is present in the system, return 0.

Find the index of a free entry in the Inode Table. If no free entry found, return -1.

Allocate the Inode Table entry to the file.

Update the Root file by adding an entry for the new file.

return 0

Delete System Call

Algorithm 2 Delete System Call

Input: Filename

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

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

Open System Call

Algorithm 3 Open System Call

Input: Filename

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

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

Close System Call

Algorithm 4 Close System Call

Input: File Descriptor

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

If file descriptor does not correspond to valid entry in Per Process Resource Table, return -1

Get the index of the File Table entry from Per-Process Resource Table entry.

If file is locked by the current process, return -2.

In the File Table Entry found above, decrement the File Open Count. If the count becomes 0, invalidate the entry.

Invalidate the Per-Process Resource Table entry.

return 0

Read System Call

Algorithm 5 Read System Call

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

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

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

Write System Call

Algorithm 6 Write System Call

Input: File Descriptor and a Word to be written

Output: 0 (Success) or -1 (File Descriptor given is invalid) or -2 (No disk space) or -3 (File is of type ROOT).

```
if word is to be written to standard output then
    Issue the machine instruction to output the given word and return 0.
end if
If file descriptor does not correspond to valid entry in Per Process Resource
Table, return -1.
If the file descriptor corresponds to Root file, return -3.
Get the index of the File Table entry and lseek position from the Per Process
Resource Table entry.
If lseek is equal to MAX_FILE_SIZE - 1, return -2.
Find the block and position in the block to which data has to be written.
while the file is locked by a process other than the current process do
    put the process to sleep and call the scheduler.
end while
Lock the file.
Get the buffer page number from block number.
while the buffer is locked by a process other than the current process do
    put the process to sleep and call the scheduler.
end while
Lock the buffer page.
if the buffer contains a block other than the required block then
    Bring the block to the buffer from the disk
end if
Copy the word passed as argument to the offset position in the buffer.
Set dirty bit to 1 in the Buffer Table entry and increment lseek position.
Increment file size in the Inode Table entry and Root file entry.
Unlock the buffer and wake up all processes waiting for buffer.
Unlock the file and wake up all processes waiting for the file.
return 0.
```

Seek System Call

Algorithm 7 Seek System Call

Input: File Descriptor and Offset

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

If file descriptor does not correspond to valid entry in Per Process Resource Table, return -1.

Get the index of the File Table entry and lseek position from the Per Process Resource Table entry.

Check the validity of the given offset

if given offset is 0 **then**

  Set lseek value in the Per-Process Resource Table entry to 0.

else

  Change the lseek value in the Per-Process Resource Table entry to lseek+offset.

end if

return 0

Fork System Call

Algorithm 8 Fork System Call

Input: None

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

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

Exec System Call

Algorithm 9 Exec System Call

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

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

Exit System Call

Algorithm 10 Exit System Call

Input: None

Output: None

→If no more processes to schedule, shutdown the machine.

Unlock and close all files opened by the current process.

→ Release all the semaphores used by the current process.

Wake up all processes waiting for the current process.

Invalidate the Process Table entry and the page table entries of the current process

Invoke the scheduler to schedule the next process.

Getpid and Getppid

Algorithm 11 Getpid System Call

Input: None
Output: Process Identifier (Success)

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

Algorithm 12 Getppid System Call

Input: None
Output: Process Identifier (Success)

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

Wait and Signal

Algorithm 13 Wait System Call

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

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

If process is intending to wait for itself or for a non-existent process, return -1.

Change the status from (RUNNING, _) to (WAIT_PROCESS, Argument_PID) in the Process Table.

Invoke the Scheduler to schedule the next process.

return 0

Algorithm 14 Signal System Call

Input: None

Output: 0 (Success)

Wake up all processes waiting for the current process.

return 0

FLock System Call

Algorithm 15 FLock System Call

Input: File Descriptor

Output: 0 (Success) or -1 (File Descriptor is invalid) or -2 (File is of type ROOT).

If file descriptor does not correspond to valid entry in Per Process Resource Table, return -1.

If the file descriptor corresponds to Root file, return -2.

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

Change the state to (WAIT_FILE, ftindex) where ftindex is the File Table index of the locked file and call the Scheduler.

end while

Change the ULock field of the file table to PID of current process.

return 0

FUnlock System Call

Algorithm 16 FUnLock System Call

Input: File Descriptor

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

If file descriptor does not correspond to a valid entry in Per Process Resource Table, return -1.

if file is locked **then**

  If current process has not locked the file, return -2.

  Unlock the file.

  Wake up all processes waiting for the file.

end if

return 0

Semget System Call

Algorithm 17 Semget System Call

Input: None

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

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

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

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

return Semaphore Table entry index.

Semrelease System Call

Algorithm 18 Semrelease System Call

Input: Semaphore Descriptor

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

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

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

Unlock the semaphore before release.

end if

Decrement the process count of the semaphore.

Invalidate the Per-Process resource table entry.

return 0

SemLock System Call

Algorithm 19 SemLock System Call

Input: Semaphore Descriptor

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

```
If Semaphore descriptor is not valid or the entry in Per Process Resource Table
is not valid , return -1.
while the semaphore is locked by a process other than the current process do
    Put the current process to sleep
    Call scheduler
end while
Change the Locking PID to PID of the current process in the Semaphore Table.
return 0
```

SemUnLock System Call

Algorithm 20 SemUnLock System Call

Input: Semaphore Descriptor

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

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

if semaphore is locked **then**

  If current process has not locked the semaphore, return -2.

  Unlock the semaphore.

  Wake up all processes waiting for the semaphore.

end if

return 0

Timer Interrupt Handler

Algorithm 21 Timer Interrupt Handler

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

Disk Interrupt Handler

Algorithm 22 Disk Interrupt Handler

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

Exception Handler

Algorithm 23 Exception Handler

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

Terminal Interrupt Handler

Algorithm 24 Terminal Interrupt Handler

Set the status field in Terminal Status Table to 0.

Locate the Process Table entry of the process that read the data.

Copy the word read from standard input to the Input Buffer field in the Process Table entry.

Set the PID field of Terminal Status Table to -1.

Wake up all processes waiting for terminal.

OS Startup Code and Shell process

Algorithm 25 OS Startup Code

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

Algorithm 27 Shell process

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

Shutdown System Call

Algorithm 26 Shutdown System Call

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

Idle Process

Algorithm 28 Idle process

```
while true do
end while
```

Other Components of eXpOS

- XSM – Underlying architecture
- Support Tools :
  - SPL – System Programmer's Language.
  - ExpL – For writing application programs.
  - XFS Interface – For loading files externally.

Work Done

- Redesign of existing OS data structures.
- Design of new data structures.
- Design of executable file format.
- Design of algorithms for system calls and interrupt handlers.
- Webpage for Project eXpOS was created http://exposnitc.github.io/

Future Work

•Building a Roadmap for guidance.
•Implementing and testing the algorithms designed
•Adding more features like file permissions and Super-User previleges.
•Adding Multi-Threading feature.
•Adding a Directory structure in eXpFS.

Conclusion

- Easy to implement
- More informative
- Better comprehension of OS concepts

References

1.Shamil C. M., Sreeraj S, and Vivek Anand T.Kallampally, ”XOS - An Experimental Operating System”, http://xosnitc.github.io/
2.Saman Hadiani, Niklas Dahlbck, and Uwe Assmann, ”Nachos Beginner’s Guide”, http://www.ida.liu.se/TDDB63/material/begguide/
3.George Fankhauser, Christian Conrad, Eckart Zitzler, Bernhard Plattner, ”Topsy - A Teachable Operating System”, Version 1.1, http://www.tik.ee.ethz.ch/topsy/Book/Topsy1.1.pdf
4.Maurice J. Bach, ”The Design of Unix Operating System”.

THANK YOU