GROUP E

EK Sannara, Shakeel Ahmad Ahmad, Juan Vazquez

Final Report: NachOS

Final git commit: fff78fad971b6a1f1ae22fcf080e17e55db1a278 (tag: NachOSV1.2) on dev branch

MAIN FEATURES

We present our version of NachOS v2.0, an improved version of the original NachOS with the following characteristics:

- Multithreading and multiprocessing
- A File System that supports a directory three, multiple open files, and can manage files with size up to 119,75KB.

In addition, our system implements user semaphores and also some mechanisms to avoid common errors when writing multithread programmes, like forgetting to use a join before the main thread exits (and killing all the threads even if they have not finished) and doing a join to a thread that is not a child of the calling thread.
Also, NachOS v2.0 also incorporates a shell, with the most useful commands to manage directories, and run programmes.

SPECIFICATION

void PutChar(char ch)
Description: Puts a character in the NachOS console. The PutChar() acquires the lock of NachOS console till character has been written down.
Parameters: **ch:** the character to write

void PutString(char str)
Description: Puts a string in the NachOS output console.
Parameters: **str:** the string to write. If the string exceeds in length MAX_STRING_SIZE then the string is truncated.

char GetChar()
Description: Retrieves a character from the NachOS console. The GetChar() acquires the lock of NachOS console till character has been read out.
Returned Value: The character read.

Char GetString(char[] arr,int size)
Description: Retrieves the string from the NachOS console and stores it in the memory. The GetString() function acquires the lock until all n characters have been read out or until either there is a line break or an end of file.
Parameters: **arr:** The array of characters to be passed.
size: The size of arr. This value can't be more than MAX_STRING_SIZE

Void PutInt(int n)
Description: Writes an integer to the NachOS output console using ascii values.
Parameters: **n:** the integer to write on the NachOS output console.

void GetInt(int *n)
Description: Reads an integer as a string from the NachOS input console using ascii representation and converts it to an actual integer value.

Parameter: ***n:** pointer to an integer which is being read through the NachOS input console

int UserThreadCreate(void f(void *arg), void *arg)

Description: Creates a new thread in the calling process.

Parameters: **arg:** The argument to be passed to the routine f()

f(): The start routine to be executed by the new thread

Returned Value: The thread id of the newly created thread if created successfully otherwise returns -1. The thread id's are unique inside the same process, and they are not repeated in the NachOS session.

void UserThreadExit()

Description: Informs to the kernel that the thread has finished its work. Cleans and exits the current thread.

int UserThreadJoin(int threadID)

Description: Waits for the thread specified by **threadID** to terminate. If that thread is not currently running this function returns immediately. If that thread is not a child of the calling thread or the parameter **threadID** is not valid, then UserThreadJoin() also return immediately with a value of -1. This syscall can only be used to join threads from the same process.

Parameters: **threadID:** The thread ID number to be waited upon by the calling thread.

Return: Join attempt status of function (0 for success and -1 for failed)

void SemInit (sem_t *semID, int semCounter)

Description: Initializes the semaphore at the given address given by the parameter semID. The semaphore is then initialized to the desired value provided by the semCounter parameter

Parameters: **semID:** The address of the semaphore to be initialized.

semCounter: The number the semaphore will be initialized to.

void SemP (sem_t *semID)

Description: Does a wait on a semaphore at the given address specifiedd by the parameter **semID**. Decrements the value of the semaphore by 1. If the semaphore is initialized, a warning will be shown.

Parameters: **semID:** The address of the semaphore to be waited.

void SemV (sem_t *semID)

Description: Does a post on a semaphore at the given address specifiedd by the parameter **semID**. Increments the value of the semaphore by 1. If the semaphore is initialized, a warning will be shown.

Parameters: **semID:** The address of the semaphore to be posted.

int ForExec (char *name)

Description: Creates a new process and execute the program named **name** in the newly created process.

Parameters: **name:** The name of the program to be run in the new proces

Return: The process id, pid, of the new process. This pid is unique across the system and is not reused even if the process terminates.

void UserWaitPid (int pid)

Description: Waits for the process with id **pid** to finish execution.

Parameters: **pid:** The process id of the process to wait.

void cpunix(const char *from, const char *to)

Description: Copies the file **from** located in the unix file system to the file **to** in the NachOs file system

Parameters: **from:** The name of the file in the unix file system

to: The name of the file in the NachOs file system

void filepr(char *name)
Description: Print the contents of the file **name.**
Parameters: **name:** The name of the file to be printed.

void DIR ()
Description: Prints the contents of the entire file system

int rm(const char *name)
Description: Remove the file **name.** A file that is currently open cannot be removed.
Parameters: **name:** The name of the file to be removed.
Return: If the file was correctly removed, returns 0. Otherwise, returns -1.

void ls()
Description: Lists the contents of the directory

int mkdir(const char *name)
Description: Creates a directory with named **name.**
Parameters: **name:** The name of the directory to be created
Return: If the directory was correctly created, returns 0. Otherwise, returns -1.

int cd (const char *name)
Description: Changes the directory to **name.**
Parameters: **name:** The name of the target directory
Return: On success returns 0, otherwise returns -1.

int open(const char *name, int create, int size)
Description: Open or creates a file named **name**. The flag **create** is used to signal if the file should be created if it doesn't exist. If the file is created, it will have a size given by **size.**
Parameters: **name:** The name of the file to be open or created
create: 1 to create a file if it doesn't exist, 0 if a file shouldn't be created.
size: The size of the file in case it is created.
Return: On success returns the file descriptor of the newly open file, otherwise returns -1.

int read(int fileDescriptor, int count)
Description: Reads and prints in the console **count** bytes of the file corresponding to **fileDescriptor**. The seek position of the file being read is incremented by **count.**
Parameters: **fileDescriptor**: the file descriptor of the file to be read.
count: The intended number of bytes to be read.
Return: The number of bytes actually read or -1 if there is an error.

int write(int fileDescriptor, const char* buf, int size)
Description: Write **size** bytes from **buf** in the file referred by **fileDescriptor.** The seek position in the file is incremented
Parameters: **fileDescriptor:** The file descriptor of the file to be written
buf: A pointer to the data to be written
size: The number of bytes to be written
Return: The number of bytes actually written or -1 if there is an error.

int lseek(int fileDescriptor, int offset)

Description: Changes the current location to **offset** within the file referenced by **fileDescriptor.** Thus, the point of the next read/write will be at **offset**.
Parameters: **fileDescriptor:** The file descriptor of the file to be written
offset: The location within the file for the next read/write
Return: 0 on success or -1 if there is an error.

int close(int fileDescriptor)
Description: Closes a file referenced by **fileDescriptor.**
Parameters: **fileDescriptor:** The file descriptor of the file to be close.
Return: 0 on success or -1 if there is an error.

TEST PROGRAMS

Test on multithreading part 1
Commands: ./nachos-withstub -rs -x d1mtwj
Details: The main thread will create threads VIA (*UserThreadCreate*) in a "for loop". In each iteration, the main will pass the current *i* counter as an argument to the threads. Each of these threads will print the argument that has been passed. *UserThreadJoin* is called upon after every *UserThreadCreate,* the results are that the printed message from the threads will be in an ascending order matter while as without the join, the threads printing would be random.

Test on multithreading part 2
Commands: ./nachos-withstub -rs -x d1mtwoj
Details: The main thread creates 2 threads. These 2 threads, each creates another 2 threads of their own. These threads each print a message to indicate who they are.

Test on semaphores
Commands: ./nachos-withstub -rs -x d1s
Details: We have set up a producer-consumer scenario where the main has created 2 threads. A total of 3 semaphores are used, (*fullcount* initialized at 0, *emptyCount* initialized at 4, *mutex* initialized at 1). The buffer has a size of 4.
The producer thread will attempt to put an incrementing number which starts from 0 to 10. SemP is called on *emptyCount* to check if the buffer is full and then calls SemP on *mutex* to secure a lock. *mutex* and *emptyCount* will then be posted VIA the system call SemV after the producer task has been completed.
The consumer thread will, on the other hand, takes 10 number that is in the buffer and print what was taken out. Initially, SemP is called on *fullCount* to check if the buffer is empty and then mutex to secure the lock. Once its work is done, SemV will be called on *mutex* and *emptyCount* to signify the release of the lock and that one value of the buffer has been consumed.

Test on multiprocessing part 1
Commands: ./nachos-withstub -rs -x d2p0
Details: 3 processes are created. Processes 1 and 2 will attempt to open a valid filename. Each process will make 2 threads which print out a message which indicates from where they are from printing from.
The 3rd process on the other hand will attempt to do a forkExec with a non-existent file name. Thus the call forkExec here will return -1, which will be then captured by main which will then print out an error of the mistake for the user.

Test on multiprocessing part 2
Commands: ./nachos-withstub -rs -x d3p1

Details: The main will attempt to create a new process which will print out a message, this attempt is wrapped around a "for loop" with 1000 iteration. The process will then print out its counter of the "for loop" to indicate it has finished its work. UserWaitId is invoked after every forkExec, thus the main will wait for each process to finish before proceeding and incrementing the counter of the loop and creating another process. The achieved result is that the processed are created in an ascending order manner.

Test on filesystem
Commands: ./nachos-step5 -rs -x d5filesys
Details: The main creates a thread. This thread creates a file named "hola" with the size of 40 and writes the sentence "Hello from group E\n". The main calls a *userThreadJoin* immediately after the thread creation to wait for its completion. After the main has waited, it will then attempt to open the file named "hola" and read its content, which should print out the message which was written in the thread.

Shell
Commands: ./nashos-step5 -cp powershell shell, and then ./nachos-step5 -rs -x shell
Details: Runs a shell that allows launching a program. The program has to reside inside NachOS file system. Also implements the commands *cd, makedir, ls* and *rm* to manage directories in NachOS file system. Additionally, commands to copy files from unix file system to NachOS file system, print the contents of a file and print the contents of the entiere file system are also implemented. The command *help* will show information about the commands implemented in the shell.

IMPLEMENTATION

MULTITHREAD AND MULTIPROCESS

Multithread and Multiprocess Model

Our implementation of Nachos supports multiple processes, and each process can run several threads as well. When creating a thread, each thread is assigned a thread id that is unique across the address space. Also, thread id's are not reused even if the thread has terminated. When creating a process using the ForkExec system call, each process gets a unique process id across the system.

MULTITHREAD SYSTEM

Each user thread is supported by a kernel thread, so it is a 1:1 mapping. The threads inside a process share the same address space, but they have their own registers and stack.

Supporting Structures

Since the address space is shared among threads running in the same process, there are several fields in the class AddrSpace that are used to manage multi-threading. The most important fields are:

- A counter, *tidcounter*, that is incremented each time a thread is created. The value of this counter is used to assign the thread id or *tid* to the newly created threads.
- A counter, *livethreads*, that is used to count the number of threads that are running in the address space. Each time a thread is created that counter is incremented, and each time a thread finishes that counter is decremented.
- A bit map, *stackFrames*, used to keep track which frames from memory are free in order to allocate them as the per-thread stack.
- An array of semaphores, called *createdThreads*, that is used to implement the UserThreadJoin syscall.
- Semaphores used as locks to protect the global variables.

In addition, the class *Thread* has a new filed called *tid*. This field will hold the thread id.

Stack Management

In NachOS each process is assigned a stack, the user stack, with a fixed size given by the constant *UserStackSize*. In order to allocate the stack for each thread inside a process, the user stack is divided into frames with a fixed size, thus avoiding external fragmentation. Each time a thread is created, a frame is assigned allocated to that thread, and when the thread finishes, the frame is deallocated. The bit map *stackFrames* is used to help in the allocation and deallocation of frames. On Figure on the right, there is an example of a process with the user stack divided and allocated for several threads.

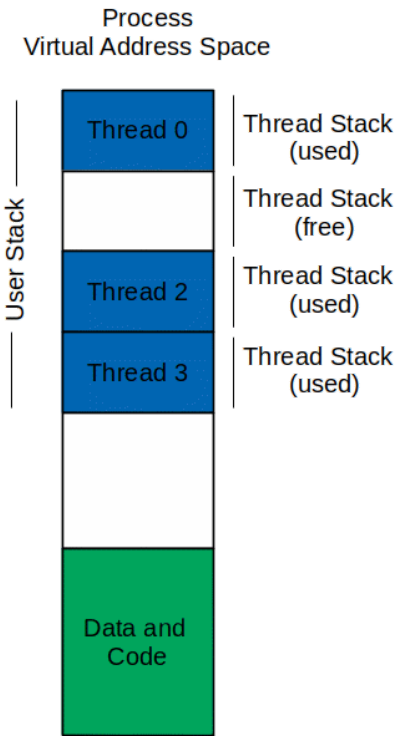


Figure: Stack Management

Managing the Finishing of the Threads and the Process

If a process exits before all of its child threads finishes, the machine will shut down and those threads will not have the chance to terminate. A process exits when the main function returns. When this happens, the syscall *Exit* is executed.

We created a handler for the Exit syscall, in order to synchronize the termination of the process, i.e. the *main* thread and its child threads. In order to do this, when the Exit syscall occurs, the value of the variable *livethreads* is checked. Recall that each time a thread is created the variable *livethreads* is incremented, and each time a thread finishes the variable is decremented. If *livethreads* is greater than 0, the main thread yields the CPU, allowing the other threads to run. When the main thread regains the control of the CPU, it checks the value of *livethreads* again. This loop is repeated until the value of *livethreads* is equal to 0, and when this happens, the main thread, and thus the process, is allowed to terminate.

User Thread Join

We implemented a syscall, *UserThreadJoin*, to allow the users to wait that a given thread finishes. In order to do this, the array *createdThreads* is used. This is an array of semaphores, and when the address space is created this array will be initialized with all of its semaphores with value 0. When a thread finishes, it will post on the semaphore that corresponds to its thread id (*tid*). For example, a thread with tid = 3 will post on the semaphore located in *createdThreads[3]*, i.e the instruction *createdThreads[3]->V()* is executed.

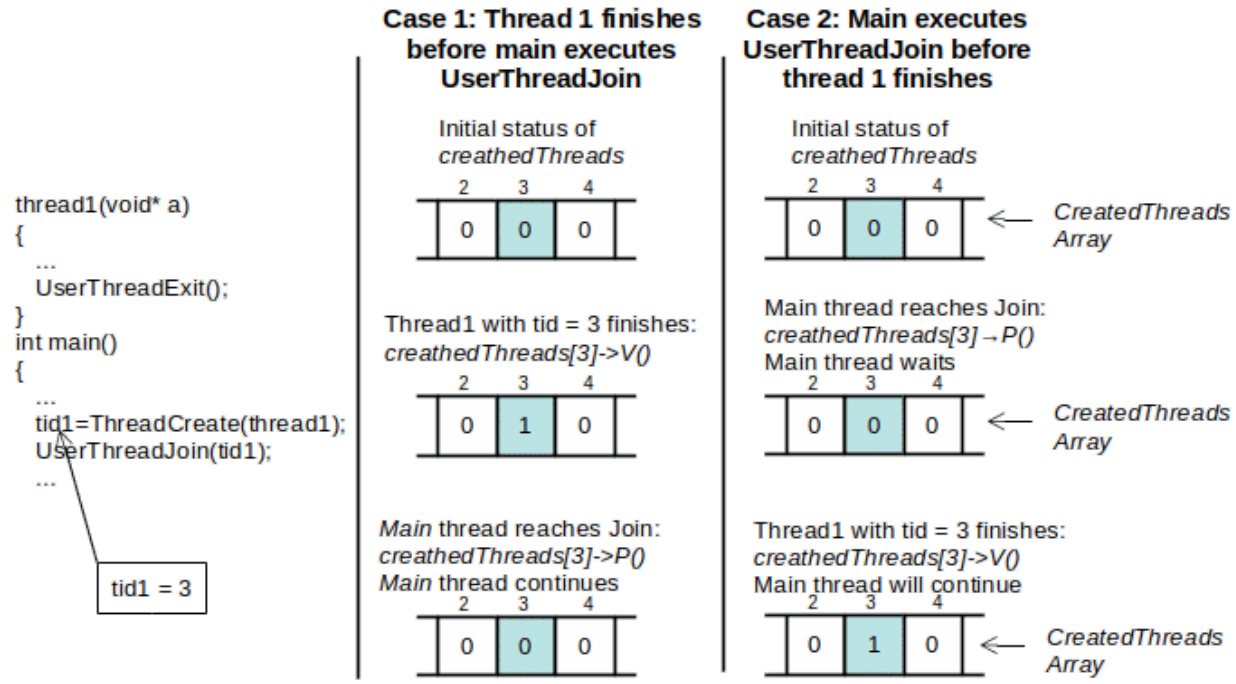
If a user thread calls the *UserThreadJoin* syscall, it passes to the kernel as a parameter the tid of the thread that wants to wait. Then, the kernel will wait on the semaphore at index *tid* in the array *createdThreads*. For instance, if the user called *UserThreadJoin(3)*, the kernel will execute the instruction *createdThreads[3]->P()*. Therefore, if the thread with thread id 3 has finished and posted to the *createdThreads[3]* semaphore, the thread calling *UserThreadJoin* will be allowed to continue. Otherwise, the calling thread will wait for the thread with tid = 3 to post in the *createdThreads[3]* semaphore.

Using the *createdThreads* array has a huge disadvantage: it means that the number of threads that can be created during the lifetime of the system is limited to the size of the array. This can be corrected by using a different data structure, for example, a linked list. Due to time constraints, we were not able to change from an array to another structure.

This implementation means that if a thread calls *UserThreadJoin* on a tid that has never been assigned to any thread, it will wait forever, or at least until a thread with the same tid is created and finishes. In addition, it is

possible for a thread to wait on any other thread, even if such thread is not a child of calling thread. We corrected this behavior and implemented some restrictions that we believe it is useful to avoid bugs in user programs. Thus, some mechanisms were put in place so if a user calls *UserThreadJoin* to wait for a thread that has never existed or if a user calls *UserThreadJoin* to wait for a thread that is not a child of the calling thread, the *UserThreadJoin* syscall will return immediately with a return value of -1.

This behaviour is achieved by keeping track of all the parent thread ID of each threads that are made with an integer typed array named *activeThreads[]* in the address space. For the creation of each threads, the parent's thread id is saved and stored in *activeThreads[],* where the index would be the thread ID of the newly created thread. Thus, when a thread wishes to join another thread, it checks if the value of *activeThreads[toBeJoinedThreadID]* is the same as the thread ID invoking the call. The warning is then invoked if the value is mismatched.



User Semaphores

In order to allow user level semaphore access, a new data type based on the int, *sem_t*, is introduced for user level uses. In the address space, A semaphores typed array, *UserSemaphore[]* is used to manage all the semaphores declared and utilized by the users. While an int variable *UserSemCounter* is used to assign unique IDs in a process to new *sem_t* variables.

The *sem_t* data structure will hold a unique semaphores ID for the kernel after each user level initialization through the system call *SemInit* takes a sem_t argument and gives it a unique semaphore ID in the process. This unique ID will then be used as index to *UserSemaphore[],* which will then at its index initialize a semaphore within the kernel.

For waiting on a semaphore, *SemP* calls on semaphore wait function provided by the kernal named *V()* on *UserSemaphores* at the index obtained from arg. Ex *UserSemaphores[semID]->(V)).* While as a post is done with the call SemV and uses the kernal provided function *P()*. Ex *UserSemaphores[semID]->(P)).*

MULTIPROCESS SYSTEM

Our version of NachOS supports multiprocessing. This means that it is possible to run multiple threads that belong to different address spaces.

Supporting Structures

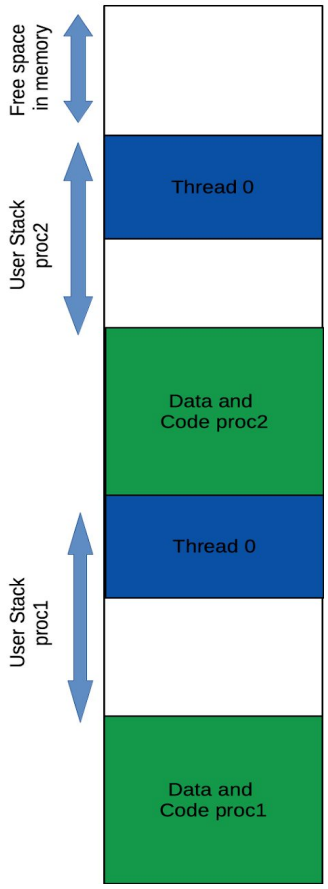
In order to support multiprocessing, some variable and structures were made available system-wide, by declaring them in *systems.h* and initializing them in the *Initialize* function inside *system.cc*. The most important variables and structures to support multiprocessing are:

- A counter *procounter*, that is incremented every time a new process is created. The value of this counter is used to assign a process id or *pid* to the newly created processes.
- A counter *livepro*, that is used to count the number of processes that are running in the system. Each time a process is created *livepro* is incremented and every time a process exits *livepro* is decremented.
- An object named *frameProvider* of type *FrameProvider*. The class *FrameProvider* implements a bit map, and the object *frameProvider* is used to allocate frames from Main Memory for the process.
- An array of semaphores, called *createdPro*, that is used to implement the UserWaitPid syscall.
- Semaphores used as locks to protect the global variables.

In addition, the class *AddrSpace* has a new field, called *pro* that will hold the unique number identifying the address space. This number corresponds to the process id or *pid*.

Creation of a New Process

To support multiprocess in NachOS, we implemented the *ForkExec* syscall. This syscall creates a new process in which a new program will run, we will call this new program as the *target program*. The name of the file containing the compiled target program is passed as a parameter for the syscall.



When a user calls *ForkExec*, the following steps will be executed:

- The function that handles the *ForkExec* syscall is called *do_ForkExec*. This function performs the following tasks:
  - After recovering from memory the name of the target program passed as a parameter, the kernel will open the respective file. In addition, using the *frameProvider->NumAvailFrame()* function that returns the number of free memory frames, we check if there is enough space in memory to create the new process.
  - The counter *procounter*, that holds the number of processes created, is incremented.
  - A new object called *space* of type AddrSpace is created. The constructor of AddrSpace will fill the Page Table using the function *frameProvider->GetEmptyFrame()*, which returns the number of a free frame in memory and marks it as used. This way each entry of the Page Table will have a number corresponding to the assigned physical frame. In addition, the code of the target program is loaded to memory, using the mapping just created in the Page Table. Finally, the counters and structures described in the subsection *Supporting Structures* of the section *Multithread System* are initialized. In the figure on the left, we can see an example of a layout of two processes in memory, where the assignation of the frames was done in order.

- A new thread is created and forked. The name of the procedure passed as a parameter for the Fork is *StartForkedProcess*. The arguments passed for this procedure are the pointer *space* to the AddrSpace object recently created, and the value of *procounter*.

At some point, the procedure *StartForkedProcess* will run. We shall remark that this procedure is running in a new thread, different from the one used to create it. In this procedure, the pointer *space* of the current thread is made to point the new address space previously created in *do_ForkExec*, that was passed as a parameter for *StartForkedProcess*. Also, the *pro* field of the new address space is set to the value of *procounter* that was passed as a parameter as well. Recall, the *pro* field will hold the process id, or *pid*. Finally, the machine registers are initialized and the function *machine->Run()* is called, thus running the target program in a new address space (i.e. in a new process).

In the next figure, we can see several threads that belong to two different address spaces. We can see that the *space* pointer inside each thread points to the corresponding address space.

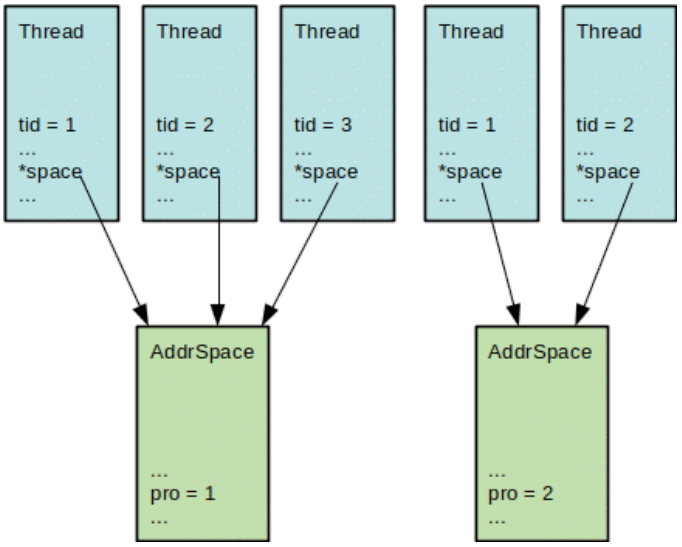


Figure: Threads belonging to two different address spaces

Managing the exit of the processes

If a process exits before all of the concurrent processes exits, the machine will halt and those processes will not have the chance to terminate. Therefore, a mechanism to avoid this was implemented. Remember that when a process exists, the syscall *Exit* is executed. Thus in the *Exit* syscall handler, the value of the variable *livepro* is checked. We need to remember that each time a process is created the variable *livepro* is incremented, and each time a process exits that variable is decremented. If *livepro* is greater than 0, it means that the ending process is not the last one, so the process just releases the resources, for example it deallocates its memory, and the supporting kernel thread finishes, without halting the system. On the other hand, if *livepro* is equal to 0 the machine will be halted.

User Wait Pid

The syscall *UserWaitPid* allows the users to wait that a given process finishes. The mechanisms used for this siscall is very similar to the mechanism used in *UserThreadJoin*. That is, an array of semaphores is used, in this case, the *createdPro* array. Everytime a process finishes it will post in the semaphore inside *createdPro* indexed by the value of the process id of the exiting process. The caller of *UserWaitPid(pid)* will wait on the semaphore on the *createdPro* array indexed by *pid*. Therefore, if a process calls *UserWaitPid(3)* it will wait for a process with id = 3 to finish by executing *createdPro[3]->P()*. On the other hand, when a process with pid equal to 3 exits, it will execute *createdPro[3]->V()*, thus signaling the calling of *UserWaitPid(3)* to continue.

FILE SYSTEM

Our implementation of the filesystem supports a directory tree, the ability to have multiple open files, and files with size up to 119,75KB. In the implemented File System, a file can not be opened by more than one process, but it can be opened by several threads inside the same process. The user is responsible for proper management if a file is open multiple times in a process.

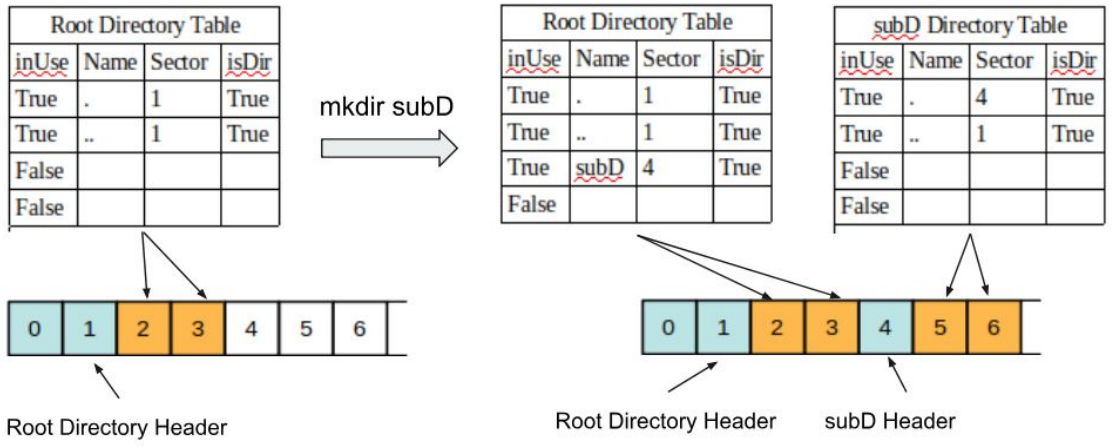
Directory Tree

In order to support a directory tree, the syscalls *mkdir* (make a directory) and *cd* (change directory) were implemented. For this, a new field in the *Directory Entry* class was added: *idDir*, which is used to differentiate if the corresponding entry is a file or a directory.

When the user executes the *mkdir* syscall the following actions take place:

- A new file header is created, that points to the data blocks containing the Directory Table that will be created in the next step
- A new Directory Table is created. By default it will have two entries: '.' that points to the header of the new directory that is being created, and '..' that points to the header of the parent directory.
- In the Directory Table of the current directory, a new entry with the *isDir* set to TRUE is added. This entry will also have the name of the new directory and the sector where its header is located.
- The bit map will be updated accordingly.

In the following figure, we can see an example of this process. The user executes the *mkdir subD* command from the *root* directory, and then a new header for *subD* is allocated in sector 4, a new Directory Table for *subD* is created with the default entries '.' and '..' pointing to *subD* and *root's* headers respectively, and a new entry with the information of *subD* is added to the *root's* Directory Table.



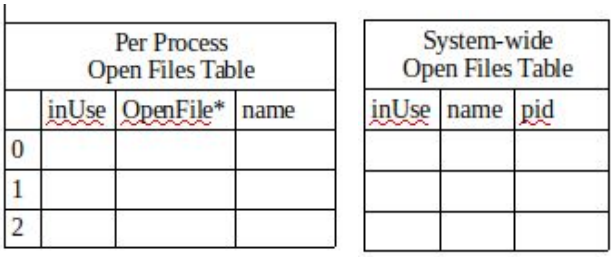
The syscall *cd* is used to change directory, so if for example, the user created the subdirectory *subD*, performing the command *cd subD* will make the system to change directory to *subD*. In order to do this, two global variables are used: one of type OpenFile called *directoryFile* (already present in the original version of NachOS) and one of type int called *currentSector*. The first one, *directoryFile* will reference to the data, i.e. the Directory Table, of the current directory. The second one, *currentSector,* is the sector number where the header of the current directory is located.

When a user performs the *cd* syscall, the variables *directoryFile* and *currentSector* will be made to point the target directory. For instance following the previous example, if the user executes *cd subD* from the *root* directory, *directoryFile* will reference to the open file of the *subD* header (i.e. it will refer to the *subD* Directory Table) and *currentSector* will be set to the sector where the header of subD is located, that is sector 4. Therefore, any subsequent commands regarding the file system, like creating new files or directories, will modify the *subD* Directory Table.

Open Files Table and Concurrent Access to Files

Two allow concurrent access to files, an open files table at system level was implemented. In addition, an open files table for each process was created. To manage files, the syscalls *open, read, write, close* and *lseek* were implemented.

The two previously described tables are shown in the following figure . The system-wide table is used to register if a file is open in a process. This way, we are able to impede that two different processes open the same file. The per-process table holds the actual OpenFile reference of the file. The index of the entry where the OpenFile reference is stored in the per-process table is returned to the user as the file descriptor.



When a user opens a file by using the syscall *open(),* the following actions take place:

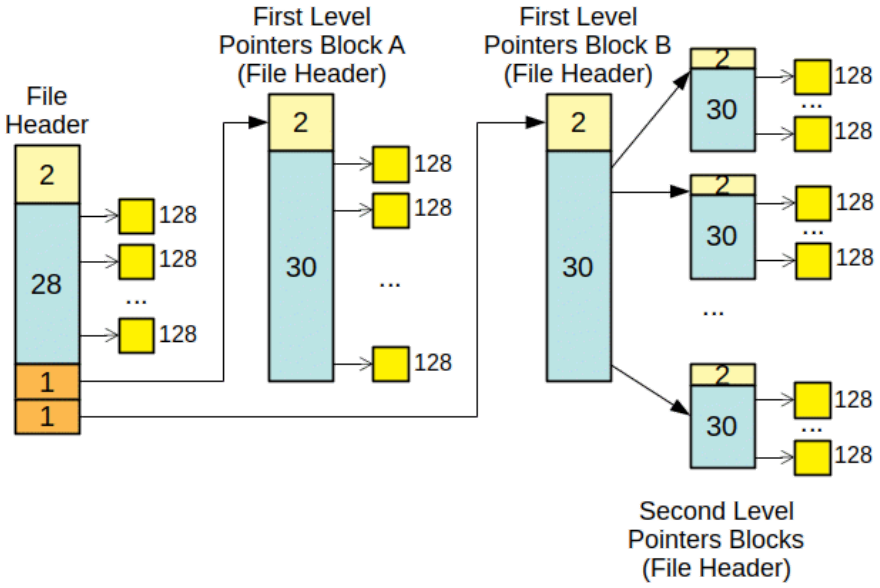
- The kernel checks if the file is already present in the system-wide open files table. If it is present, it checks in which process the file has been opened. If it was opened by another process, the operation fails. On the other hand, if it was opened by the same process, or if the if the file is not present in the system-wide open files table at all, the operation continues.
- If the file does not exsts, and the user asked to create a new one, a new file is created using the function *filesys->Create()*.
- The file is opened using the function *filesys->Open()*, that will create an OpenFile reference.
- If the file was not present in the system-wide open files table, a new entry for the file is created and filled.
- A new entry is created in the per-process open files table and filled with the information of the file that is being opened. It is worth to point that every time the user calls *open()* a new entry is created in this table, even if there is another entry for the same file. In this case, all those entries will refer to the same file, but the OpenFile object will be different.
- The index of the newly created entry in the per-process open files table will be returned as a file descriptor.

The syscalls *read, write* and *lseek* use as a parameter the file descriptor returned by *open*. Thus, whenever the user wants to read, write or perform a seek action in a file, the kernel will retrieve OpenFile reference from the per-process open files table using the passed file descriptor as an index for that table, and then use the procedures from the *OpenFile* class to read, write and seek files. The syscall *close* uses a file descriptor as a parameter as well, but in this case, the corresponding entry in the per-process open files table is marked as free. Also, if this was the last reference to the file in the process, the respective entry in the system-wide open files table will be marked as free.

Finally, some checkings were added to the *Remove* procedure in the *FileSystem* class in order to avoid that an open file can be deleted. This is simply done by checking if the file has an entry in the system-wide open files table, and if it does, it means that the file is open by at least one thread and can not be deleted.

Increasing the Maximum File Size

To increase the maximum supported file size, we used a three-level indirection, that is shown in the following figure. In this case, the maximum file size is be $(28 + 30 + 30^2)$x128, or 119.75 KB.



In order to do this, the procedures *Allocate* and *Deallocate* in the *FileHeader* class were modified. In the procedure *Allocate*, we first check the size of the file to determine if it can be handled with direct pointers, one level of indirection or with two levels of indirection. If we need one or two levels of allocation, the procedures *AllocateIndirection1* and *AllocateIndirection2* will be called in order to allocate the sectors for the indirect pointer blocks.

CONCLUSIONS

- In terms of organization, all the group member worked in the basic parts of Step 2: I/O, Step 3: Multithreading and Step 4: Virtual Memory (Multiprocessing). We did it this way because we wanted to understand the basic parts of the operating system that we were building. Then, some members of the group worked on finishing the extra parts on those steps, while one member worked in Step 5: File Systems
- The project timing provided was good, and it gave us a good idea on how to keep up and not fall behind schedule
- In terms of the project itself, we find that many improvements can be done, but due to time constraints, we were not able to implement them.
- Overall we found this project very interesting, and it helped us to gain a deeper understanding of how an Operating System works.