NachOS system-D

Nataša "Tipitaša" Vodopivec, Philippe "PhiL" Ledent, Benjamin "mrBen" Collet,
Antoine "Git Breaker" Colombier, Sayak "The Indian" Samaddar

January 19, 2018

Contents

1 Introduction

For this project, our team has high expectations on the expected result, in the one hand in term of quality, and in the other hand in term of features. Thus, we decided to implemente the required feature, plus several of additionnal feature in order to make our system safer, and more efficient. Lists and details of these features have been given in sections they are relating to.

2 Specifications

`void Halt () __attribute__((noreturn))`

Description :

Stop Nachos machine if it is the last process that is alive, and print out performance stats.

`void Exit (int status) __attribute__((noreturn))`

Description :

This user program is done (status = 0 means exited normally). Alias of `return <code>` in the `main` function.

*`int Join (SpaceId id, int* result_code_ptr)`

Description :

Join a process. Alias to wait on POSIX.

Parameters :

`id` only return once the the user program "id" has finished.

`result_code_ptr` pointer to store the result code. NULL to ignore it

Returns :

Returns 0 if the process has been joined, otherwise returns -1.

`int Create (const char *name, int perm)`

Description :

Create a file name at path "name".

Parameters :

`name` the path of the file to be created

`perm` new file permission in octal. You can use helpers O_R, O_W, 0_X, O_RW or O_RX to make it easier.

Returns :

Returns E_SUCCESS (0) on success and an error number on failure.

`OpenFileId Open (const char *name)`

Description :

Open a Nachos file "name" for reading and writing.

Parameters :

`name` the path of the file to be opened

Returns :

On success returns an "OpenFileId" that can be used to read and write to the file, returns NULL on failure.

`OpenFileId OpenDir (const char *name)`

Description :

Open Nachos directory "name" for reading.

Parameters :

`name` the path of the directory to be opened

Returns :

On success returns an "OpenFileId" that can be used to read the directory, returns NULL on failure.

`OpenFileId ParentDir (OpenFileId dir)`

Description :

Open the parent directory of a directory.

Parameters :

`dir` The child directory you want to find the parent of.

Returns :

Returns NULL if file has no parent (root directory) or if it is not a directory. This function is not implemented properly to be safe to use in multithreading so it's not recommended to use it.

`int Read (char *buffer, int size, OpenFileId id)`

Description :

Read "size" bytes from the a opened file into "buffer".

Parameters :

`buffer` address of the buffer where data is written

`size` size of data to read from the file and write to the buffer

`id` file descriptor

Returns :

If the return value was positive it corresponds to the number of bytes that were copied. If it is negative it is an error code.

`int Write (char *buffer, int size, OpenFileId id)`

Description :

Write "size" bytes from "buffer" to a file.

Parameters :

`buffer` address of the buffer where data is written

`size` size of data to read from the buffer and write to the file

`id` file descriptor

Returns :

If the return value was positive it corresponds to the number of bytes that were copied. If it is negative it is an error code.

`void Close (OpenFileId id)`

Description :

Close the related file if found, and deallocating it from the current address space. The file might still be opened in other address spaces.

Parameters :

`fd` file descriptor of the opened file

`void Yield ()`

Description :

Yield the CPU to another runnable thread, whether in this address space or not.

`void PutChar(char ch)`

Description :

Write a char to the NachOS console. This function acquires a lock on the nachOS console until the char has been written.

Parameters :

`ch` the char to write

`char GetChar()`

Description :

Reads a character in the NachOS input console. This function blocks until the char has been read.

Returns :

Returns the char read.

`void PutString(char *s)`

Description :

Write a string to the NachOS output console. This function blocks until the char has been written.

Parameters :

`s` the string to write. It has to be well typed, using the end of stream character `'\0'`. If the string exceeds in length MAX_STRING_SIZE then the string is concatenated to be valid.

`int GetString(char *s, int n)`

Description :

Read a string in NachOS input console. This function blocks until the n charracters has been read, or until either a line break or a end of file.

Parameters :

`s` the string to write

`n` the maximun number of characters. This value can't excceed MAX_STRING_SIZE

Returns :

Returns the number of charactes that were read. If negative, return a code control such as EOF.

`void PutInt(int n)`

Description :

Write an integer to the NachOS ouput console using ascii representation.

Parameters :

`n` the integer to put on the NachOS output console.

`void GetInt(int *n)`

Description :

Reads an integer in the NachOS input console as a string using ascii representation and converts it to an actual integer.

Parameters :

`*n` pointer to an integer for storing the read value

`ThreadId UserThreadCreate(void* f(void *arg), void *arg)`

Description :

Create a user thread.

Parameters :

`f` user pointer to the user function to execute. The function signature must have signature void* f(void *)

`arg` user pointer to the args of the function to execute.

Returns :

If the creation succeeded return the tid (thread id) of the new thread. If the creation failed return NULL_TID.

`void UserThreadExit(void* result_code) __attribute__((noreturn))`

Description :

Exit a user thread.

Parameters :

`result_code` result code to return to any waiting thread

`int UserThreadJoin(ThreadId tid, void* result_code_ptr)`

Description :

Wait for the specified thread to finish.

Parameters :

`tid` the thread identifier

`result_code_ptr` pointer to store the result code. Can be NULL to ignore it

Returns :

Returns E_SUCCESS if no error, anything else otherwise.

`void sem_init(sema_t* s, int e)`

Description :

User semaphore initialiser. This function must be called before any calls on this semaphore.

Parameters :

`s` the pointer to the semapore

`void sem_wait(sema_t s)`

Description :

User semaphore to wait.

Parameters :

`s` semaphore to wait on

`void sem_post(sema_t s)`

Description :

User semaphore post.

Parameters :

`s` semaphore to post on

`int ForkExec(char *s, char** args)`

Description :

Fork and run a process.

Parameters :

`*s` path to the executable

`**args` arguments to pass to the process. Must finish by a NULL arg. Can be NULL if no arguments

Returns :

Returns the pid of the created process on success, 0 on error.

`void Kill(SpaceId pid, char sig)`

Description :

Send a signal to a given process.

Parameters :

`pid` the process identifier that will receive the signa

`sig` the signal to send This call has not been implemented.

`int Tell (OpenFileId fd)`

Description :

Get the reading head position.

Parameters :

`fd` file descriptor

Returns :

Return the value.

`int Seek (OpenFileId fd, int offset)`

Description :

Move the reading head of the file.

Parameters :

`fd` file descriptor

`offset` the value where the head should be

Returns :

Return the value after beeing set. Might be different if there was a range error.

`int FileSystemInfo (fs_info_t *info)`

Description :

Get a structure containing the number of free block (free_block), the used block (used_block), and the block size(block_size) to the main file system.

Parameters :

`info` a pointer to the structure to write

Returns :

false is no error, anything else if one occurs.

`void Trunk (OpenFileId fd)`

Description :

Trunk the file to the current head position and deallocate the space already allocated after.

Parameters :

`fd` file descriptor

`int Move (char* old, char* new_)`

Description :

Move a file from the filesystem.

Parameters :

`old` the old file path

`new_` the new file path

Returns :

Return false if no error, anything else otherwise.

`int Remove (char* id)`

Description :

Delete a file from the filesystem.

Parameters :

`id` file path

Returns :

Return false if no error, anything else otherwise.

`int ReadDir (char *buffer, int size, OpenFileId id)`

Description :

Read at most "size" of next the file name.

Parameters :

`buffer` buffer address where next file is written

`size` size of data read and write

`id` file descriptor

Returns :

Return the number of bytes actually read – if the open file isn't long enough, or if it is an I/O device, and there aren't enough characters to read, return whatever is available (for I/O devices, you should always wait until you can return at least one character).

`int ChMod (int perm, OpenFileId id)`

Description :

Change the permissions of an open item.

Parameters :

`perm` the new permission in octal. You can use O_R, O_W or O_X to make it easier.

`id` file descriptor

Returns :

Returns false if no error and modified with success, true on failure.

`void *Sbrk(int size)`

Description :

Move the break value of n pages.

Parameters :

`size` the number of pages to allocate if positive, or to free if negative.

Returns :

Returns the pointer to the first byte now available if size is positive or unavailable if size is negative. If positive size and no more memory available, NULL is returned.

`int FileInfo (file_info_t* info, OpenFileId fd)`

Description :

Get a structure containing the epoch timestamp of the last access (date), the permission in octal (perm), the size(size), and its type (file = 0, dir = 1, ...).

Parameters :

`info` pointer to the structure to write

`fd` file descriptor

Returns :
Returns false is no error, anything else if one occurs.

`OpenSocketId Socket(NetworkAddress machineId, MailBoxAddress port)`
Description :
Create a socket object and return its stream descriptor.
Parameters :
`machineId` the address of the remote machine
`port` the port of the remote machine
Returns :
Returns the stream descriptor on success, 0 on error.

`int Connect(unsigned int timeout, OpenSocketId sd)`
Description :
Block the socket until a server has proceeded to synchronisation with the given socket (by calling Accept).
Parameters :
`timeout` the maximum time to wait for. 0 to be none blocking
`sd` the socket descriptor
Returns :
Returns E_SUCCESS if a peer has syncronised, E_NOTFOUND if timeout.

`int Accept(remote_peer_t* client, unsigned int timeout, OpenSocketId sd)`
Description :
Block the socket until a client has proceeded to synchronisation with the given socket (by calling Connect).
Parameters :
`*client` the structure to store information about the client
`timeout` the maximum time to wait for. 0 to be none blocking
`sd` the socket descriptor
Returns :
Returns E_SUCCESS if a peer has syncronised, E_NOTFOUND if timeout.

2.1 User library : `userlib`

`int strcmp(char* s1, char* s2)`
Description :
Compare two strings.
Parameters :
`s1` the first string
`s2` the second string
Returns :
Returns 0 if strings are the same, else -1 if the s1 differs first or is smaller or 1.

`int strlen(const char* str)`
Description :
Calculate the length of the string s, excluding the terminating null byte (' \ 0').
Parameters :
`str` the string
Returns :
Returns the number of bytes in the string s.

`char* strrchr (char* str, int character);`
Description :
Locate character in string.
Parameters :
`str` the string
`character` the character to be located

Returns :

Returns a pointer to the last occurrence of the character in the string.

`int atoi(char* s)`

Description :

Convert a string to an integer.

Parameters :

`s` the string

Returns :

Converts the the string pointed to by s to int and returns the integer value.

`char* itoa(int value, char* str, int base)`

Description :

Convert an integer to string.

Parameters :

`value` value to be converted to string

`str` array in memory where to store the resulting non-terminating thread

`base` numerical base used to represent the value as a string

Returns :

Stores the result in the array given by the str parameter.

`void *memcpy(void *dest, const void *src, size_t n)`

Description :

Copy memory area. Copy n bytes from memory area src to memory area dest. Current POSIX standards.

Parameters :

`dest` the memory area destination

`src` the memory area source

`n` the number of bytes to be copied

Returns :

Returns a pointer to the destination string dest.

`void strcpy(char *dest, const char *src)`

Description :

Copy the string pointed to by src, including the terminating null byte (' \ 0'), to the buffer pointed to by dest.

Parameters :

`dest` the memory area destination

`src` the memory area source

Returns :

Returns a pointer to the destination string dest.

`void *memset(void *s, int c, size_t n)`

Description :

Fill memory with a constant byte. Fill the first n bytes of the memory area pointed to by s with the constant byte c. Current POSIX standards.

Parameters :

`s` memory area to set

`c` the byte to be used as a constant

`n` the number of bytes to fill in the memory area

Returns :

Returns a pointer to the memory area s.

`char *strtok (char *str, char *delimiters )`

Description :

Extract tokens from strings.

Parameters :

`str` the string to be parsed
`delimiters` a set of bytes that delimit the tokens in the parsed string

Returns :

Returns a pointer to the next token, or NULL if there are no more tokens.

`int simple_strftime(char *str, size_t size, int time)`

Description :

Format date and time in a predefined nachos format and store it into a sharacter array.

Parameters :

`str` the character array to store the formated date and time
`size` the maximum size of the character array str
`time` the time to be broken down

Returns :

Returns 0 at success, otherwise -1 or 1

`char* strpad(char* str, size_t i, char padder)`

Description :

Add padding to current string so that it reaches the wanted lenght. Padding is added at the end of the string.

Parameters :

`str` the string to pad
`i` the final size of the padded string
`padder` the character to be used as padding

Returns :

Returns the pointer to padded string.

`char* reverse(char *str)`

Description :

Reverse the string.

Parameters :

`str` the string to be reversed

Returns :

Returns the pointer to the reversed string.

3 User tests

One month for programming an operating system is quite short so we didn't have time to create robust tests for all functions. We have decided to focus as much as possible on the robustness of the user-space so prevent the user from breaking the system. There are many functions that are executed in kernel space that would be worthy of more strict testing but we don't have time for that.

3.1 I/O Console

The input and output user-functions are very basic and allow users to read and write from a NachOS console. They consist in `PutChar`, `GetChar`, `PutString`, `GetString`, `PutInt` and `GetInt`. The most basic functions `PutChar` and `GetChar` don't have much testing to do except checking that they really do read and write properly. For strings there are more things to do. A string has a maximal size `STRING_MAX_SIZE = 512` and `GetString` has a parameter `int size` that indicates the maximum number of characters that should be read. There are therefore two straightforward tests.

Consider that `GetString` has a parameter `size = n` with `n` $<$ `STRING_MAX_SIZE` $- 1$. Consider `str` the string to read from the console of length `m`. In one case `m` $<$ `n` and `str` should be read properly. In the other case `n` $<=$ `m` and `str` should be truncated to its substring of size `n`. The remaining characters of `str` should remain in the Console and wait to be read by another function.
These tests were useful to reveal problems with missing \0 characters in strings and length issues.

More I/O tests have been performed one the synchronisation and multi-threading have been implemented.

For instance if multiple threads try to read at the same time they should not be duplications and if multiple threads try to write at the same time each string should be entirely written before another thread can write.

Apart form the I/O possibilities our Console implements some Unix commands that can be obtained with the command `ls /bin` in the `NachOS_shell`. These commands are `ls (-l), mv, cat, chmod, df, ls, mkdir, rm, touch`.

3.2 Synchronization

In terms of sychronisation the user is only provided with semaphores. We have however implemented the mutex locks using semaphores, semaphores and conditional variables. We have only tested locks in the kernel space and semaphores. The test on locks can be run with `./nachos-threads -rs`. Along with the provided synchronisation test that is provided our own synchronisation test on locks will be launched. As for the user semaphores we have a producer-consumer called `prod_cons` that checks our semaphores.

3.3 Multi-threading

The user is provided with `UserThreadCreate`, `UserThreadJoin` and `UserThreadExit`. Thread creation fails when for a given address space the number of threads reaches `MAX_THREADS` or if there is no space left in the memory ie the stack and the heap are colliding.

3.4 Virtual memory

The user should not have direct access to the memory so the tests have been done in a user program via special console-like interface. The user has a few commands that `[a]`llocate memory blocks, `[f]`ree memory blocks, `[p]`rint the state of the memory and get `[i]`nformation about blocks.

This has been very useful to see allocation problems when the memory is full, when it is emptied and when it is filled again, when the break between the heap and the stack was not respected.

3.5 File System

The file system has been tested with a quick test unit done on the kernel side that can still be launched using the embedded command on the `main.cc` file. This test is similar to a stressing test, and tries to fill up a file as much as possible, delete a row, delete while open elsewhere, and so on...

Furthermore it has been tested on the user side with opening closing of the files. The file merging test was also carried out where the contents of two files are merged into the third one. This tests the reading, writing and also validates the functionality of GetChar and PutChar.

3.6 Network

The network has not been tested as it doesn't really work properly. We have written a file transfer program in the userspace, but we cannot launch it because the network sometimes(often) crashes. For more details, see the last paragraph of the network implementation (subsection 4.6).

4 Implementation

4.1 I/O Console

We decided to follow the C++ standards and used inheritance of Console to implement the SynchConsole, by not replacing the Console but wrapping it.

4.2 Synchronization

The Locks implemented don't allow busy waiting under any circumstances. The implementation follows these steps: Interupts are disabled in the first place. If semaphore is available it consumes the value else it is appended to the queue where it waits for the the present thread to be executed.

After the previous thread is executed it actually broadcasts and wakes up all the waiting threads, and the next thread is executed sequentially from the list. There is a sync list which helps in keeping track of the threads waiting to be executed.

Conditions are provided to check whether the lock is held by the current thread or not and whether the next thread waiting is ready to run or not. If there are certain problems or exceptions ocurred in the waiting thread or the thread holding the lock then the synchlist is updated and refreshed, maintaining the same sequence of the previous thread.

4.3 Multi-threading and Halting

The two things we want to emphasize here are the way we handle problems that might arise at user thread creation and how we handle the halting of the machine.

When we create a new user thread we first check two things - whether the maximum number of threads has been reached or if the current address space doesn't have any more free pages to hold the new thread's stack. In these cases the creation is canceled and the call for `UserThreadCreate` returns a `NULL_TID`. Otherwise, we append it to the `mThreadList` which is the list of current threads in the address space. We give it a `TID` (automatically increases for each thread) and allocate stack for it. Then we start a user thread with it's function, args and exit_handler which holds the address of the system call `UserThreadExit`, that cleanly exits the thread after it returns, which we think is quite nice.

As for halting the machine, we took care that the `Halt()` doesn't abruptly shut down the machine and at the same time, we also took care of automatic termination. When `UserProcessExit` is called one of the two things happens: if there are other processes running on the machine we just finish the calling one; if it is the last one we route it to calling the `UserHalt`. Here we provide safety again. If a process calls `UserHalt` out of the blue we check if it is actually the last process running. If it is not, we only finish this calling process. If it is indeed the last process, it is allowed to halt the machine but only after all of the threads inside of the address space have been finished and joined and only then is machine allowed to shut down and actually does.

4.4 Virtual memory

One of the things we would like to emphasize here is our implementation of the pages access management. The `code` and the `data` segments are made `read only` and at a write access an exception is raised. The `bss` segment has `read` and `write` permissions. The main problem we had to face was that the limit between data section and BSS can be on the same page. It order to solve that and not to brake the access linkage, we cannot shift BSS to the next page. So we decided to make the shared page writtable, which is potentially as security breach.

Second interesting point is the ability of a process to get parameters out of the standard argc and argv. As said in the ForkTest syscall documentation, the call get a list of parameters that the system load on the first stack. At first, it allocates every arguments `(char*)` and keeps track of its address. Afterward, it writes an array of those arguments address `(char**)`. Finnaly, it initialise the value of register 4 with the value of `argc`, and the address of the `char*` array `(char**)` previously written in the stack into register 5. After that, the user get ran as usual.

The following two figures will show how the address space is organised. On Figure 1 we see that at the beginning we have two sections that are read-only for code and data. Then comes `bss`. Now if `bss` starts on the same page as the data finishes only the bytes for data are read-only. Then comes a padding so that the heap starts on a clean page. The break `brk` is not allocated in the memory. It is just a pointer that indicates the limit of the allocated blocks in the heap.

In figure 2 we see the memory when it is full. When `mem_alloc` is called if there is no space to allocate the new block before `brk` then a `syscall` is triggered to check if the heap can be increased. If it can it does. In this case it cannot because there would be a collision between the heap and the stack. We then trigger a `syscall` to point out the error. Similarly the stack grows with new threads but the stack will not overflow if there is no room. The memory can fill up very fast so because each thread has a stack of 6 pages.

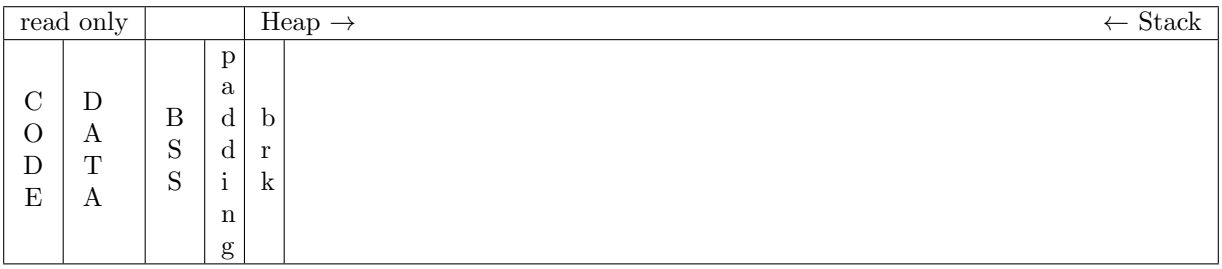


Figure 1: Address space when it is ready to run programs

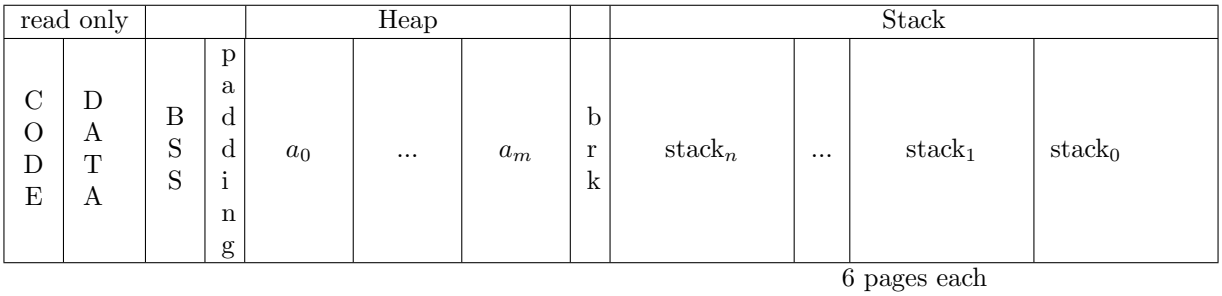


Figure 2: Full address space

4.5 File System

The file system is implemented with two layers of abstraction. Files and directory use the same main first level entry type, which is contained on one sector disk. This entity holds 5 data:

- A flag, which gives :
  - The last access timestamp
  - The permission
  - A Type which define which kind of item it does represent (Only directory and file were implemented so far)
- An effective size which corresponds to the used size for data
- The number of data sector it does use
- A list of extended sectors

An extended sector contains only a list of sector which refer to data.

When if file is written after is current size, it automatically grown up its allocated memory (and/or its effective size)

The concurrency is ensured by two principles:

- Many OpenFile object can be instantiated to a same file, but they will all use only one FileHdr instance, which will ensure that neither double allocation is done, nor deleting while other use it. The System keeps thus a list of all the current use FileHdr. This means that, as UNIX system, processes can overwritten content of the file if they writing on the same space.
- The address space keeps track of all the open files, that may be shared between the different threads of the process.

As a result : no data allocation overlapping is ensured by a transaction system on the filesystem: Such as some DBMS, the filesystem get frozen (in terms of sector allocation, other file can still be written as long as they do not require allocation/deallocation), new sectors get allocated/deallocated, then the change are committed on the filesystem.

4.6 Network

The reliable network is implemented using abstraction layers. The provided code gives us address and port, and we have used that system to encapsulate a "lite TCP" that we have defined. In this protocol, there are a server that waits for a connection and a client that connects to a server. When both parts are connected, they can both send and receive data (full duplex) and when they have finished, one of them can close the connection. The data that is transfered is bytes stream, so we encapsulate some higher level protocols. Our protocol should ensure reliability or at least notify the userspace if something cannot be delivered.

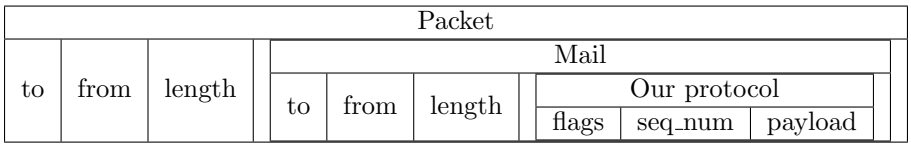
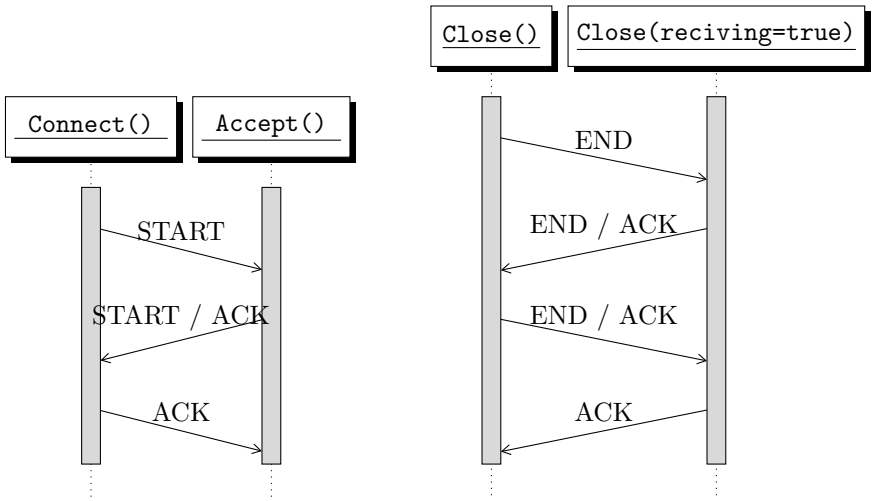


Figure 3: encapsulation of our protocol inside the existing layers

Packets and mails were provided, they provide us with an address and port system. The aim of our protocol is to add reliability for the packet delivery. As you can see in Figure 3, our header is composed of flags and a sequence number. The available flags are ACK, START, END and RESET. Once connected, two machines can send data to each other.

The sender sends a message with no flag and a sequence number, and then waits for an empty message with the same sequence number and the flag ACK (acknowledgment). If it didn't receive the ACK or if the sequence number is wrong, it means that the data has not been delivered properly, either the message has been lost or the ACK has been lost. In that case, the sender will retry to send the message. The waiting time and the number of retries are bound. The timeout is handled at the Mail level. As the `mailbox` are a synchronization queue, when we want a timeout, we schedule an interrupt that will put a `nullptr` inside the queue. If we receive a `nullptr` instead of some data, we know that the reception of the ACK had a timeout and we migth need to retry.

As you can guess, the receiver waits for a message to come and then sends an ACK with the same sequence number. If a message comes with the same sequence number as the previous one, it means that the ACK has been lost. The receiver then replaces the previous data with the one just received and sends the ACK. Send and Receive are available in the userspace with the system calls `Write` and `Read` used with a socket descriptor instead of a file descriptor.



(a) connection establishment (b) connection termination

Figure 4: Our connection protocol

Before being able to send and receive data, the two machines need to connect to each other. The connection is done on a socket, witch is a local port number, a remote address and a remote port number. These sockets are created by the `Socket` system call.

To initialize a connection, a "server" will create a socket with only a local port number and will wait on that port for a message with a START flag on it, it then send back the START and ACK flags and wait for an ACK, as seen in figure 4. (a). This is done by the `Accept` systrem call. If anything happen not properly during this initialisation, *e.g.* something else than a START come first or the last ACK get lost and a timeout, the "server" send the flag RESET and start the connection establishment again. The "client" will create a socket with a remote address and port number, and the `Connect` system call will chose a non used local port.

To close the connection, any side might send a END flag. This is done by the `Close` system call (the same that is use to close file, but with a socket descriptor instead). If a MailBox (the layer that was provided) receive a END flag, it will call the same `Close` function, but with a parameter `receiving` to `true`. The figure 4. (b) is self-explainatory and work almost like the connection establishment, except that we retry to send END and ACK if something happend (instead of using RESET). Once the last END and ACK is receive, both machine consider the connection as close, even if the last ACK is lost.

This protocol that we have define should in theory ensure a good reliability and some ease of use for the user space (as the `Accept` can wait and we don't need to launch two nachos at the same time). In practice, we've fully implemented this protocol but we sometime get a critical error. They are cause by packets that are receive twice (because ACK get lost) and we cannot get rid of them because we might lose data. The main problem is that this protocol was too ambitious, too complex, for the time we got to implement it, and still too simple to be really reliable. We should have stick with our first protocol that was far simpler, and moreover, we should not have try to redesign the protocol during the last week.

4.7 User Exception Management

We extended our program to care about many types of exceptions that a user code might trigger and that CPU cannot handle. We assume that the names of the exceptions are self explanatory about what they handle. We handle the following exceptions: `BusErrorException`, `PageFaultException`, `ReadOnlyException`, `OverflowException`, `IllegalInstrException`, `AddressErrorException`. We handle them by killing the process that caused the exception instead of calling it in a loop.

5 Work planning

We tried to follow the pre-given planning.

Criticism: There are many parts of the nachOS code we received that should have been updated a long time ago and caused unnecessary problems, that shouldn't have been a part of this project. At the same time, we did not appreciate the many mistakes, typos, false and outdated instructions in the English project instructions. Last but not least, we were told to ask questions online instead on private emails, and we received answers way after having already fixed issues by ourselves. Because of the time limitations to this project, we think this things should be improved greatly.

6 Appendix

6.1 System calls

```
#define SC_Halt     0
#define SC_Exit     1
#define SC_Join     3
#define SC_Create   4
#define SC_Open     5
#define SC_OpenDir 33
#define SC_ParentDir 36
#define SC_Read     6
#define SC_Write    7
#define SC_Close    8
#define SC_Yield   10

// Input / Output
#define SC_PutChar 11
#define SC_GetChar 12
#define SC_PutString 13
#define SC_GetString 14
#define SC_PutInt  15
#define SC_GetInt  16

// User Threads
#define SC_CreateUserThread 17
#define SC_ExitUserThread 18
#define SC_JoinUserThread 19
```

```
// Semaphores
#define SC_SemaInit 20
#define SC_SemaWait 21
#define SC_SemaPost 22

// Processes
#define SC_ForkExec 23
#define SC_Kill   24
#define SC_Tell    25
#define SC_Seek    26
#define SC_FSInfo  27
#define SC_FileTrunk 35

// File System
#define SC_Move    28
#define SC_Remove  29
#define SC_ReadDir 30
#define SC_Changemod 31

// Memory
#define SC_Sbrk   32
#define SC_FileInfo 34
```