Advanced Operating Systems 2010 Documentation

Release 1.0 final

Philipp Keller, Gerd Zellweger

December 17, 2010

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

This document is intended to provide insight about our implementation of the Advanced Operating System (AOS) project [1].

The two main parts of our project are the SOS server and a libsos library for user processes, which enables these to communicate with the SOS server. We chose to implement our SOS server in monolithic single threaded fashion, because we expected to get the least amount of trouble with this approach as the project would evolve.

- Chapter *Memory Management* will give insights about how the SOS server manages the memory of user processes.
- Chapter *System Call Interface* explains how user processes communicate with the SOS server.
- Chapter *I/O Subsystem* explains the code responsible for reading and writing files on the NFS (Network File System) server as well as reading and writing from and to the serial port.
- Chapter *Clock Driver* describes the driver code to access the CPU clock timer.
- Chapter *Process* contains information about how we manage user processes in the root server.

[1] AOS Project Website

CHAPTER

TWO

MEMORY MANAGEMENT

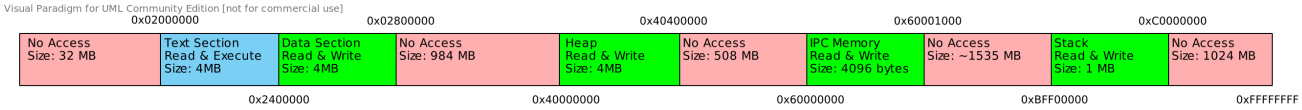
Memory management related code resides within the `src/sos/mm/` directory. It consists of three parts:

1. Frame Management
2. Pager (handles process page faults)
3. Swapping (swaps pages in and out of physical memory)

Hint: To test the Swapping it is recommended to reduce the number of physical frames artificially. This can be done by uncommenting `SWAP_TEST` (`frames.c:25`). The amount of frames is then defined through `SWAP_FRAMES_LIMIT` (`frames.c:30`).

2.1 Virtual Address Space Layout

The address space for a process is set up according to the following diagram:



You may notice that everything below `0x02000000` is not accessible. The reason for this is that loading of an ELF binary and copying text and data sections is done by a privileged piece of code running in the same address space but with the capability to access physical memory directly (i.e. everything below `0x02000000`). For more information refer to the *Process Creation* chapter.

2.2 Virtual Address Layout

We use a fixed page size of 4096 bytes which requires us to use a 12 bit offset in our virtual address. Since the CPU has a word size of 32 bit we can address 2^{20} frames with the remaining 20 bits (and since the NSLU2 has only 32 MB RAM this is more than enough). We decided to use a 2 level page table using the top 12 bits of an address as an index in the page directory (1st level) and the following 8 bits as an index in the second level page table. All page tables are stored in the heap memory of the SOS server (which we set up to use a reserved region of

5 MB in physical memory). This approach favors simplicity but it probably would have been wise to use physical frames for better scalability with multiple concurrent processes.

2.3 Functioning of the Pager

Implementing demand paging was probably the most difficult part of the project. In this chapter we try to give an overview of the interactions between a user space process, our SOS server and the network during page faults.

The first diagram (see *Access a page for the first time*) shows the steps taken in case a certain memory region is accessed for the first time (this means that the page is not swapped and there exists no mapping in the software or hardware page table for it). As we can see, once the system needs to swap out we select a page based on the second chance algorithm. We implemented second chance using a queue where we keep all the pages currently associated with a physical frame. Notice that in the diagrams the entities Pager, Swapper and NFS handler are not separate processes but rather refer to functions located in different code files. To give a better overview we decided to split it up anyways even though all of these functions are called in the SOS system call loop.

If one takes a look at the structure `page_queue_item` (`swapper.h:9`) he will notice that we store a bunch of information there to help us identify to which process a page belongs to, find the correct swap offset and corresponding page table entry. While a page is swapped out it's corresponding `page_queue_item` is inserted in the swapping queue. This is needed because swapping in and out leads to an asynchronous NFS callback in the server so in case a process is deleted while it's waiting for a page fault we need to be able to detect this in the callback (see also *Process Deletion*). The swap file stores only page content and can currently hold about 5000 pages (although this can easily be changed by setting `MAX_SWAP_ENTRIES` in `swapper.c:64`). The offset of a page in the swap file is stored in the page table entry during the time a page is swapped out and once it is back in memory gets copied into the corresponding `page_queue_item`.

The next diagram (see *Requesting a previously mapped page*) shows the steps taken in case a page is requested which was already mapped before but currently does not exist in the hardware page table.

As we can see, we have to distinguish two cases:

1. If the page is currently in the swap file, we need to read it back into physical memory. Notice also that we create the corresponding `page_queue_item` before we called `nfs_read()` so we can pass its address as an argument which is handed back later in the callback.
2. If the page is already in physical memory, we just update the hardware page table using `L4_MapFpage()`.

Now that we've seen the sequence patterns used to handle page faults, we will take a closer look at the states a page can be in and how they affect our data structures (see *Page as a State Machine*).

- You may notice that once a page is dereferenced, only page faults with read request can

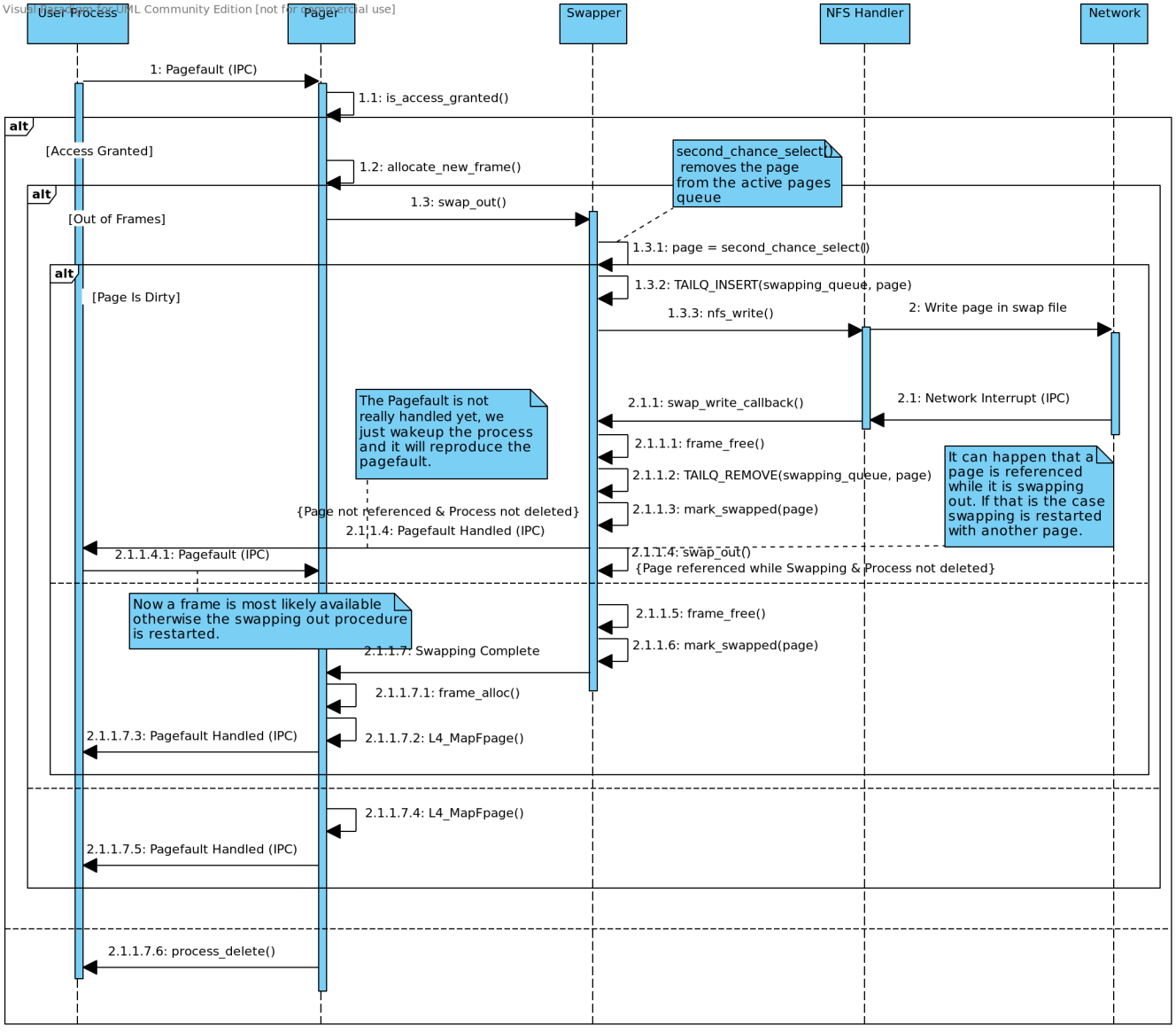


Figure 2.1: Access a page for the first time

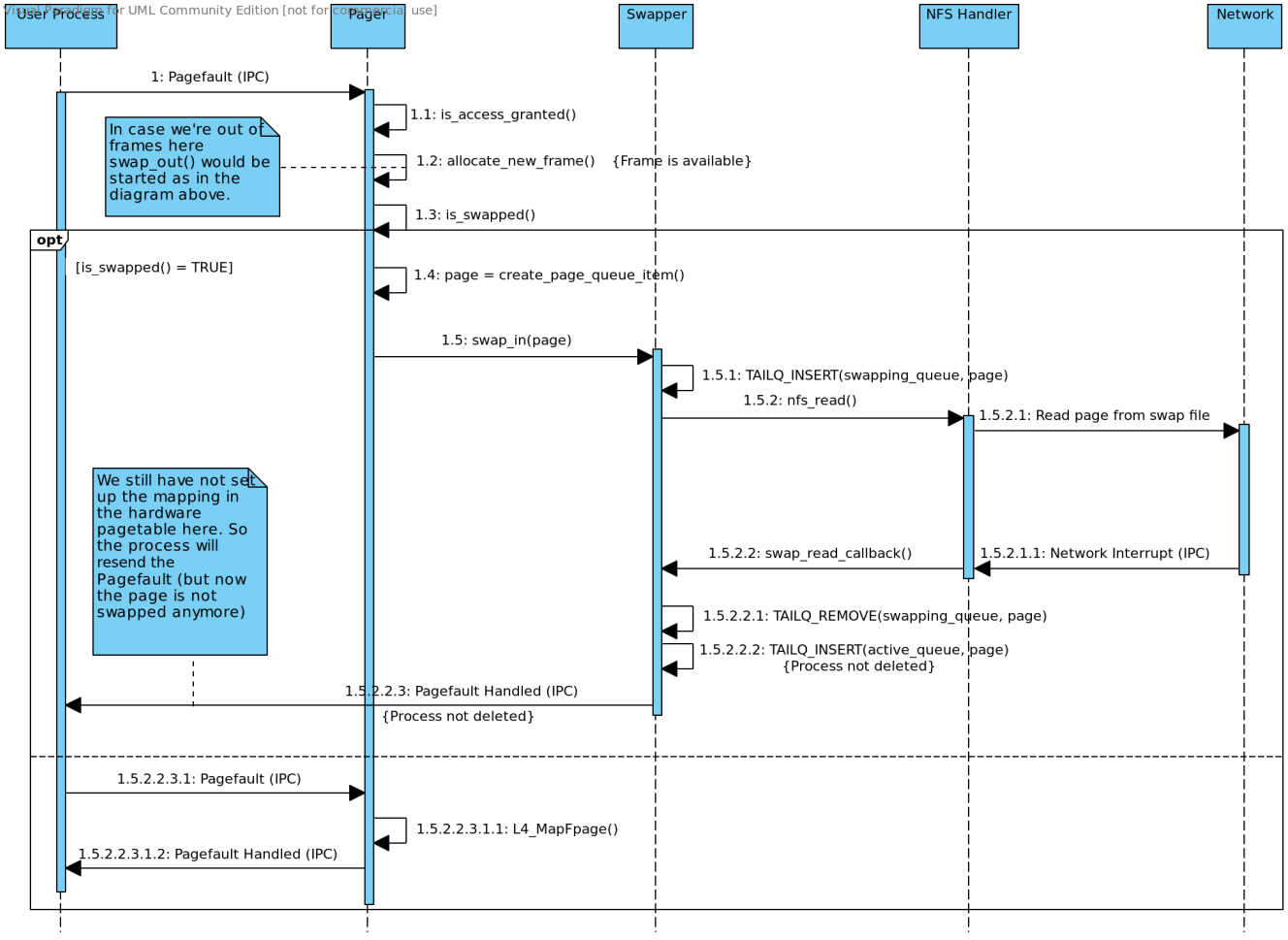


Figure 2.2: Requesting a previously mapped page

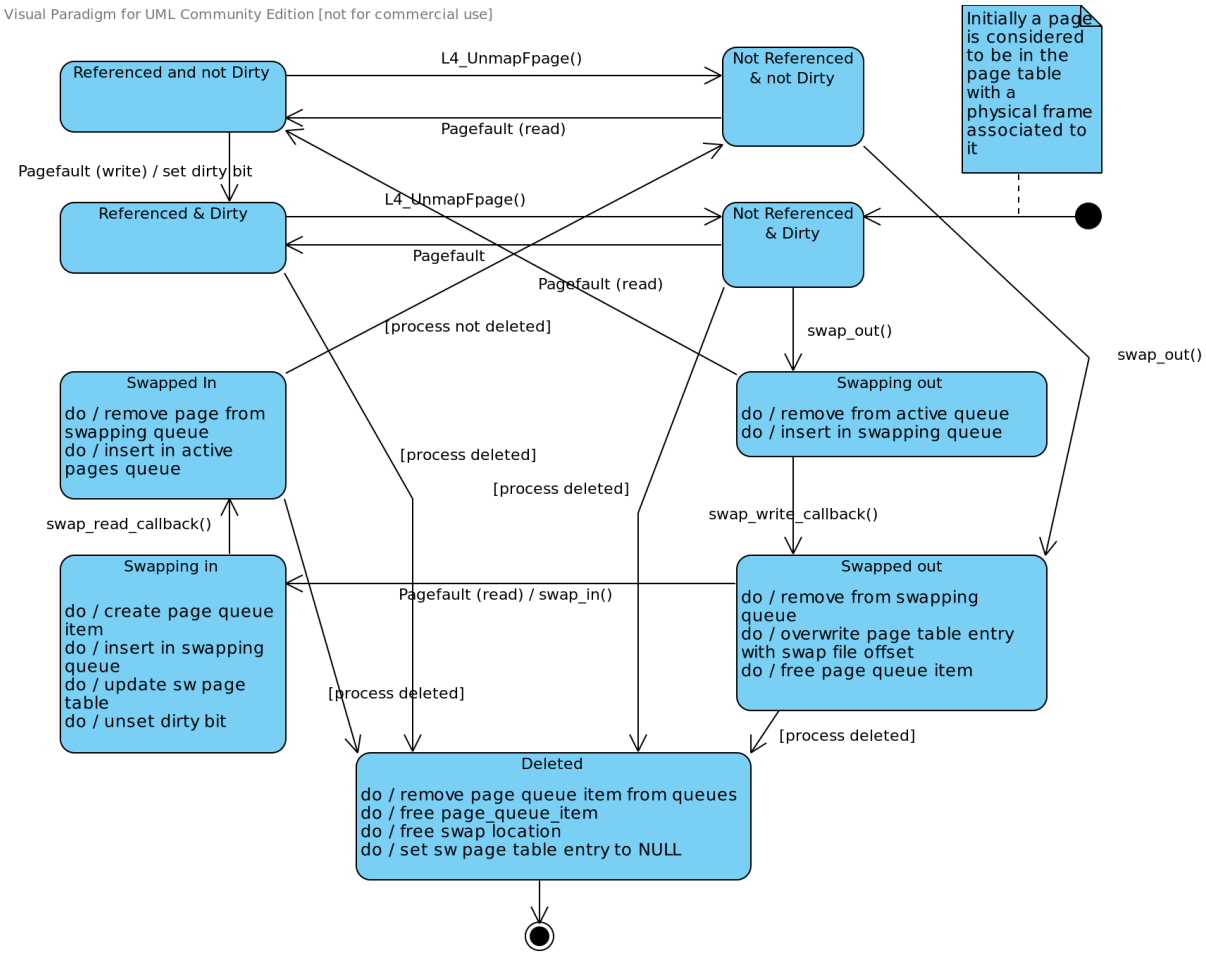


Figure 2.3: Page as a State Machine

happen. This is because in L4, the first access to an unmapped page will always deliver a read page fault, even if the fault is from a write instruction. Once a page is mapped, subsequent faults will report the correct access.

- Since the ARM CPU does not provide reference and dirty bits, we simulated the reference/dereference operation by mapping and unmapping the page in the hardware page table.
- The dirty bit is stored in the lower 12 bits of our software page table. A page referenced for the first time is always marked as dirty. This is needed because nothing is swapped out in the beginning. Once a page is swapped out, the dirty bit is cleared in the page table, but it is reset again if the page gets mapped with write access in the future.

CHAPTER

THREE

SYSTEM CALL INTERFACE

Our system uses a monolithic design where the SOS server functions as a root task and handles all incoming system calls. Since it is the only privileged thread in our system we can also use any kind of L4 calls there.

Therefore, whenever a process wants to use some hardware or privileged L4 system calls, they have to ask the SOS server to do it for them. The System Call Interface is the way how user programs communicate with the SOS server in such situations.

In L4, the only way in which different threads can communicate with each other is through IPC (interprocess communication), therefore IPC is also used to implement the functionality of the system calls. Implementing IPC was the goal of milestone 0. To solve it, we more or less only put together all the L4 calls necessary to send one message. But already during work for milestone 1, we realized that because of the fact, that we need to send an IPC message for every system call, there would be a lot of dupicated code for all the different system call functions.

So we designed a more generalized way of handling the system calls, which has parts in both the SOS server code and the libsos library code. But before describing each of these parts, we need to look at how they will be able to understand each other first.

3.1 Interprocess Communication

We implemented IPC to use shared memory for transmitting the bigger chunks of data like strings and buffers in general, but used the L4 message registers to send anything which is small enough to fit in a single register (i.e. 4 byte words). For shared memory each user process has one page mapped with caching disabled [1] at a fixed virtual address. Before sending a message to the server, the user process can copy data into that page. Then upon recieving a message, the server does an address lookup in the user process' page table to acquire access to the frame corresponding to the user process' IPC page. The server can read the data, processes the request and writes a result back into the frame. Then he sends a IPC reply to wake up the user thread which is able to read the result data in its own address space again.

[1] Enabled caching for these pages leads to cache inconsistency if the page is accessed in the SOS server and the user process.

3.2 System Call Numbers

The identifiers for all the system calls we implemented are defined in a header file `syscalls.h`, that is included in both the SOS server and the libsos library. It defines the following system calls:

System Call Type	Description
SOS_OPEN	Opens a file
SOS_READ	Reads from an open file
SOS_WRITE	Writes to an open file
SOS_CLOSE	Closes a file
SOS_GETDIRENT	Gets name of directory a directory entry
SOS_STAT	Gets file status (access rights, size, creation time, ...)
SOS_PROCESS_CREATE	Creates a new process from an executable file
SOS_PROCESS_START	Starts a process
SOS_PROCESS_DELETE	Deletes a process
SOS_PROCESS_ID	Return process ID of the calling process
SOS_PROCESS_STATUS	Get status information of all running processes
SOS_PROCESS_WAIT	Wait for another process to exit
SOS_PROCESS_GET_NAME	Get the command string that startet the process
SOS_SLEEP	Sleep for a number of milliseconds
SOS_TIMESTAMP	Get uptime of the system
SOS_UNMAP_ALL	Unmap all memory pages (for debugging)

Note: This header file resides in `src/libs/sos_shared`, a library which only contains structs, defines, typedefs and some macros but is accessable on both sides - the SOS server and user processes.

3.3 Server

In the SOS server, we have created a function table that maps each of these constants to an actual handler function. The switch statement in the system call loop can then do a lookup in this table based on the IPC label it receives.

The handler functions have the following signature:

```
typedef int(*syscall_function_ptr)(L4_ThreadId_t, L4_Msg_t*, data_ptr);
```

Inputs are the thread ID of the sender (a user process), the IPC message recieved from that process and a physical memory address pointing to the user threads dedicated IPC page (see *Interprocess Communication*). The return value is an integer that specifies if the system call loop is supposed to immediately send a reply message (which has to be loaded into the message registers by the handler function before) to the user thread (return is 1) or not (return is 0). This is done to be able to use L4s special IPC speedup function `L4_ReplyWait`.

Important: If a system call handler function does not immediately reply to the user thread (usually because the server has to wait for a callback to return) we need to make sure to reply in

the callback function itself, otherwise the user thread will wait until it receives a reply message forever.

A handler function can be registered in the function table by calling

```
void register_syscall(int ident, syscall_function_ptr func);
```

with one of the system call identifiers as the first and the respective handler function pointer as second parameter. For example like this:

```
register_syscall(SOS_OPEN, &open_file);
```

3.4 User processes

On the side of the user process, there is the libsos library that contains all the necessary functions to interface with the SOS server. The library features a generic function that takes care of all the assembling of the IPC message that is needed for a system call:

```
L4_MsgTag_t system_call(int type, L4_Msg_t* msg_p, int args, ...) {
    assert(args < IPC_MAX_WORDS);

    L4_Accept(L4_UntypedWordsAcceptor);
    L4_MsgClear(msg_p);

    // Appending Data Words
    va_list ap;
    va_start(ap, args);
    for(int i = 0; i < args; i++) {
        L4_MsgAppendWord(msg_p, va_arg(ap, L4_Word_t));
    }
    va_end(ap);

    // Set Label and prepare message
    L4_Set_MsgLabel(msg_p, CREATE_SYSCALL_NR(type));
    L4_MsgLoad(msg_p);

    // Sending Message
    L4_MsgTag_t tag = L4_Call(L4_Pager());
    if(L4_IpcFailed(tag)) {
        printf("System Call# %d has failed.", type);
    }

    L4_MsgStore(tag, msg_p);

    return tag;
}
```

Inputs are the system call type (one of the defines in the *shared header file*), a pointer to a L4 message and the amount of following `L4_Word_t` values (it's a variable input function). The message pointer is used to assemble the outgoing message and to store the incoming mes-

sage and therefore does not need to be passed initialized (the `system_call` function will initialize it). The function clears the message (`L4_MsgClear`) and appends all the provided `L4_Word_t` values to it. It's assumed that the message data has already been copied into the shared memory region before `system_call` is called. The function then loads the message and calls `L4_Call`, which sends the IPC message and blocks the thread until the reply message arrives.

The libsos library consists of functions that nicely wrap up the different system calls. One of these, as an example on how to call the `system_call` function, is open:

```
/**
 * Opens a file for a given access mode.
 * The file path is copied over IPC shared memory.
 * The mode is passed in the first IPC message register.
 *
 * @param path path to the file
 * @param mode access rights
 * @return File descriptor or -1 on error.
 */
fildes_t open(const char *path, fmode_t mode) {
    if(path == NULL)
        return -1;

    // preconditions (client responsibility)
    assert(strlen(path) <= MAX_PATH_LENGTH);
    assert(mode == O_RDONLY || mode == O_WRONLY || mode == O_RDWR);

    //  copy path in IPC memory
    strcpy((char*) ipc_memory_start, path);

    L4_Msg_t msg;
    L4_MsgTag_t tag = system_call(SOS_OPEN, &msg, 1, mode);
    assert(L4_UntypedWords(tag) == 1);

    fildes_t fd = L4_MsgWord(&msg, 0);

    return fd;
}
```

Warning: Special care has to be taken if you use `printf()` as a debugging method in system calls. Since `printf()` uses our write system call internally it will at least destroy your data in the shared page and most likely lead to weird result data in your L4 message.

3.5 Interface Changes

We tried to keep changes in the given interface description as little as possible and made only three small changes in the behaviour of the system calls during the project:

1. The `time_stamp()` call returns `uint_64` (8 byte) instead of `long` (4 byte). Since

we store the time stamp in the SOS server already as a 8 byte value we did not see much point in removing the upper 4 bytes here.

2. `process_wait()` returns -1 if a invalid pid is specified.
3. `process_create()` we made this call a bit more specific in case an error occurs. Basically it will return a value less than zero in case of an error. The exact error types are defined in `process_shared.h:6`. We also do busy waiting here in case the process table is full.

In addition to these changes we introduced some new system calls `process_start()` and `process_get_name()`. However they are not very useful for user processes since they can only be used by the initializer (see *Process Creation* for a more detailed explanation).

3.6 Known Limitations

The read and write system calls are not guaranteed to work properly in case the parameter `nbyte` is larger than 512 bytes. This is because of limitations in the NFS library which does not split packets.

3.7 Testing

We did blackbox testing of all the system calls. The test program is located at `src/tests/syscalls`.

CHAPTER

FOUR

I/O SUBSYSTEM

The Input Output Subsystem uses a NFS library to access files over the network and a serial library to support a console.

4.1 File System

To understand the Serial I/O and NFS parts, we need to explain the design of our file system first. Not only the real files that we can access over the NFS, but also the "console", the only way a user can interact with our system, is modeled as a file residing in our file system. If a user program wants to write to the console or read from it, it simply opens the file called "console" like any other file and performs normal file writing/reading operations on it (see *Serial I/O* for details).

4.1.1 Server side implementation

File system related code resides in the `src/sos/io` directory. Our file system has no support for directories and can only handle a limited number of files (limited to `DIR_CACHE_SIZE` defined in `io.h:32`). We maintain a file cache (a static array called `file_cache`) that allows `DIR_CACHE_SIZE` possible entries of type `file_info`. At boot time, the system uses NFS to find files in a specific directory of the connected host and creates a `file_info` entry in the file_cache for each file found. The system is not able to detect changes to files on the host or new added files, unless they are created by our own file system of course.

If we take a closer look at the `file_info` type (see `io.h:14`) we can see that we store function pointers for all the file operations here. This allows us to be able to specify a different behaviour for every file. We use this form of dynamic dispatching to figure out at runtime (based on the file) if the serial I/O functions should be invoked or rather the ones that deal with NFS.

The second data structure defined in `io.h:36` is a structure called `file_table_entry`. It is used to hold information about files which are currently opened by a process. Each user process has its own file table in the process descriptor. It contains pointers to the type `file_table_entry`. If a file is created/opened, one of these structure is allocated and stored in the file table.

To enable user processes to access the file system, there are system call handlers defined in `io.c` for all the different file operations. These function do some general checks before dispatching to the corresponding function in the `file_info` structure. The specialized handler functions are located in `io_serial.c` and `io_nfs.c` respectively. An example of an I/O system call is shown in the diagram *Flow of events during a read operation*.

4.2 Serial I/O

The serial I/O system is a very important part of the system, because it ultimately enables the user to communicate with our system. Since the NSLU2 system does not have a display but only a network adapter, it has to send its output over the network to the host computer. This is also the way in which a program running on our system can get inputs from the user.

Reading from and writing to this kind of "console" are modeled as normal file access operations, but on a specially designed, virtual file. At boot time, we initialize the first entry of the file cache to become that virtual file with the name "console", creation and access time 0, read- and writable and so on (`io_init()` in `io.c`). But most importantly we set its file operation function pointers to different functions than we would for one of the "real" files. The four pointers are set to `open_serial()`, `read_serial()`, `write_serial()` and `close_serial()`, which can all be found in the file `io_serial.c`.

The function `open_serial()` is different from its pendant `open_nfs()` in the sense, that for the console we have to ensure that only one process at a time is allowed to open the "file" in read mode. The reason for this is that if we enter something into the console, it is not reasonable to send the same input to several different processes. The `close_serial()` function ensures, that the exclusive read right is reset correctly, so that at a later time another process can acquire it.

The function `write_serial()` basically invokes the serial drivers `serial_send` function, that sends strings over the network to the net cat instance that is listening on port 26706. If sending fails, `write_serial()` tries to resend the buffer 20 times before it is discarded.

The function `read_serial()` reads characters from the circular buffer in the file info structure of the console file. The circular buffer gets filled by the callback function `serial_receive_handler()` which is registered to the serial driver in `console_init` which is called in `io_init()` (line 194) on system boot time.

Every process gets created with a pre-initialized file descriptor 0 which has write capability for the console file. In milestone 0, we were supposed to implement the two user space functions `sos_write()` and `sos_read()` which are used by the C library for its I/O functions. Back then, they just called the respective system calls directly, but now we replaced that code so it just performs the needed file operations on the console file.

4.3 Network File System

The code for accessing the real files over NFS on the host PC is found in the file `io_nfs.c`. It contains the handlers for read, write, open, getdirent and stat system calls specific to the NFS.

The usual behaviour is that the `create_nfs()`, `read_nfs()` and `write_nfs()` call the NFS library function which (after a reply has been received from the network) calls a supplied callback function in return. We use the token value as a pointer to the file handle and store all the information about what we need to do on a callback in the file handle.

Note that `open_nfs()` does reply immediately. We don't need to do a NFS call over network first because we already cache the NFS file handle in the file cache.

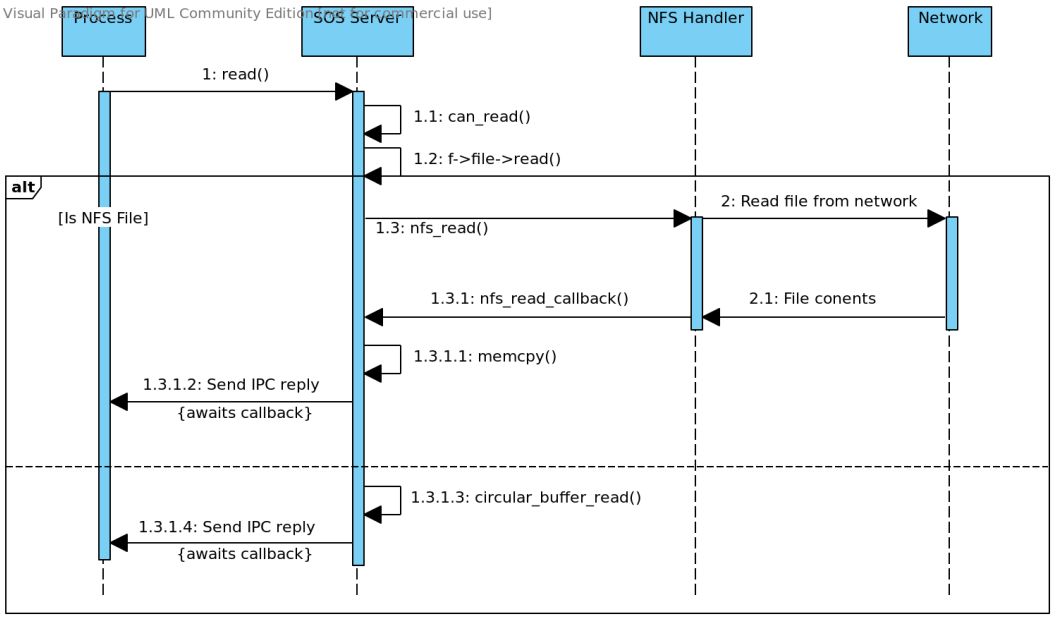


Figure 4.1: Flow of events during a read operation

CHAPTER

FIVE

CLOCK DRIVER

For milestone 5, we were supposed to create a driver for the IXP42X timer. The driver should have enabled printing of the the system uptime and running sleep timers. With these tools we were then supposed to do some benchmarking on the performance of the NFS functionality.

5.1 Clock System Calls

All the SOS server code associated with the clock driver is inside the two files `clock.h` and `clock.c`. On the user space side we just have two system call functions (`time_stamp` and `sleep`, implemented in `sos.h` and `syscall_timer.c`) and of course the whole benchmarking programm (`src/sosh/benchmark.c`).

For the time stamp measurement we use the time stamp timer register of the IXP42X hardware and for sleep calls we use the general purpose timer 0.

The timer needs of course to be initialized before it can be used. We do that in the function `start_timer` which maps the memory region where the timer registers are in uncached mode, start the time stamp timer register and enable the interrupts for our time stamp timer and the general purpose timer 0.

The timer interrupts are detected in the system call loop in `main.c`.

5.1.1 Time Stamp

The timestamp timer will give an interrupt whenever its 32 bit value overflows. The interrupt recieved in the system call loop then comes from thread number `NSLU2_TIMESTAMP_IRQ`. The system call loop calls the handler function `timer_overflow_irq` which increments our internal high word of the time stamp value and resets the timestamp timer (the low word needs to restart counting).

The system call handler function for the time stamp retrieval function is called `send_timestamp`. It calls the internal function `get_time_stamp` which returns a 64 bit integer assembled from the high word value in `clock.c` and the lower word read from the actual hardware timer. To send this number with IPC message registers we split it up into two 32 bit integers that are assembled back into the 64 bit integer in the user space function that did the system call.

5.1.2 Sleep Timers

The general purpose timer 0 is programmed to give an interrupt whenever it reaches zero (in the system call loop, the received interrupt then comes from thread number `NSLU2_TIMER0_IRQ`). In that case `timer0_irq` is called as the interrupt handler, which removes the head of the timer queue and restarts the timer with the finish time of the next queue entry. `timer0_irq` of course also calls the handler function associated with that timer. At the moment this is always the function `wakeup` which just sends the IPC reply to the sleeping user process to wake it up. But the way we implemented it, we could extend this easily.

The sleep system call basically uses the function `register_timer` of the clock driver interface. `register_timer` will insert an `alarm_timer` structure into a priority queue and if the new timer has been inserted as the new queue head because it has a finishing time before the time of the old head of the queue, the timer is restarted with the new time.

Additionally, we have the functions `remove_timers` and `stop_timer`. The first one is used at process deletion time, it basically walks through the timer queue and deletes all timer entries that belong to the deleted process. It also restarts the timer if the pending timer queue head belonged to the deleted process. The second function stops all timers and deletes all entries from the timer queue.

CHAPTER

SIX

PROCESS

6.1 Process Table

We use a hash table of fixed size to store our process descriptor (see `process.h:13` type `process`). This means that we limit the number of concurrently running processes to the size of this table (which at the moment is 128). If a higher amount would be required here, the value `MAX_RUNNING_PROCESS` in `process.h:29` can be adjusted. However, since the key to find a process in the table is the corresponding L4 thread ID it also requires careful adjustment of the `tid2pid()` helper function.

6.2 Process Creation

For process creation we tried to "cheat" a little bit to make it a lot simpler: In a normal system the SOS server would probably be responsible for loading the file from the NFS file system, parse the elf binary and map the contents to the correct pages. But because in L4 the unit of protection is the address space - and one can control what is running something when in that address space - we first create a new address space and run an initializer program in it. The initializer appears as a (more privileged) user process in the server and resides within the boot image. This means that in order to read the ELF binary we can reuse our file system calls. To set up the pages containing the data and text section of the binary we are also able to use the `elf_loadFile()` function which copies the contents at the correct offsets using `memcpy()`. This means that all the copying to the uninitialized memory regions is handled through the pager who allocates the frames and sets up the page table correctly. When `elf_loadFile()` finishes, we just tell the thread to stop and restart it by setting the instruction pointer to our process entry point. The downside of this approach is, that pages containing code are not linked to the original binary and get swapped out in the swap file. However, this is only a performance issue and does not affect the functioning of the system in general. The described sequence is also shown in the diagram below (see *Creating a new process*).

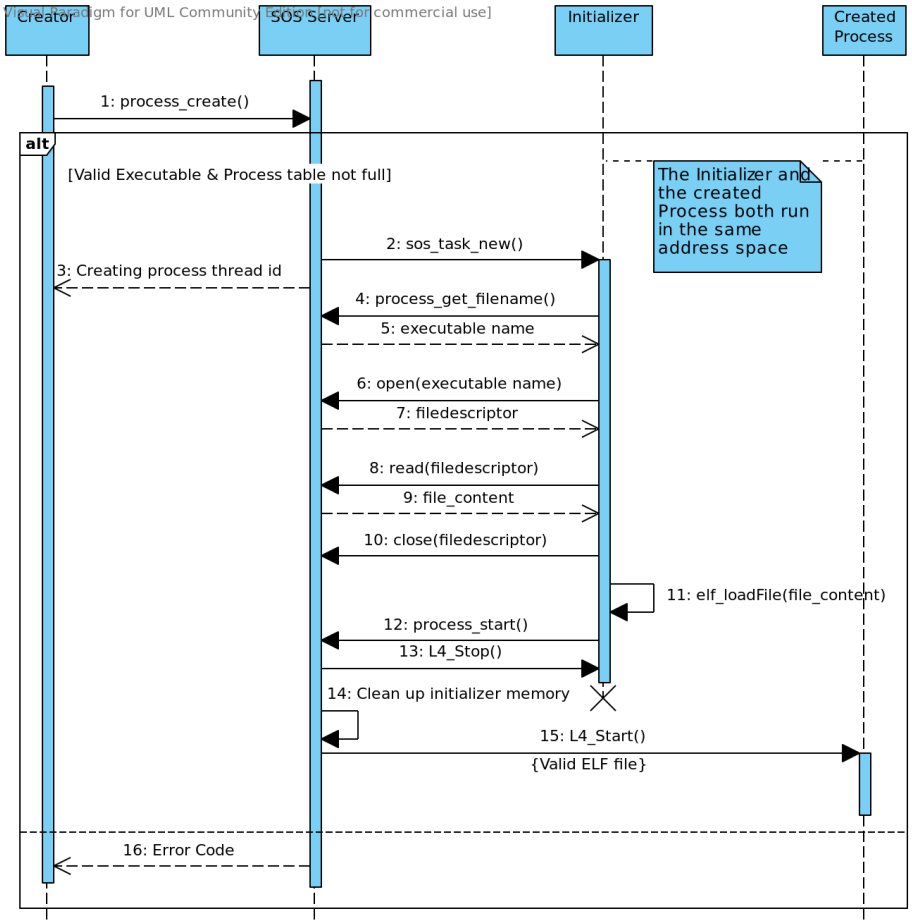


Figure 6.1: Creating a new process

6.3 Process Deletion

Process deletion basically just stops the thread using `L4_Stop()`, cleans the mappings in the hardware page table and frees data structures belonging to this process in the SOS server. The biggest problem here is, that a process could be deleted while waiting for a system call to return and the SOS server would not detect that the process is deleted and try to reply. Since our server is single threaded and `process_delete()` is also a system call, this can only happen whenever we have a system call which does not immediately (i.e. not in the same loop iteration) return back to the client.

With our implementation this is the case in the following situations:

1. Creating a file on the NFS file system
2. Reading and writing to a file
3. Waiting for a process to finish (`process_wait()`)
4. Swapping pages in and out of physical memory
5. Sleeping

For the third case the fix is pretty easy, since the ID on which a thread waits is stored in the process descriptor. So removing the process from the process table would already make sure that we don't call back anymore. Sleep timers are easily removed by walking through the priority queue of all currently registered timers. In all the other cases we call back to the process in the callback function invoked by the NFS library. So the solution for all those cases look similar. The NFS library allows you to define a token (a 4 byte word) which is passed back to you as an argument to the callback function. For example in `swap_in` and `swap_out`, we pass on the corresponding `page_queue_item` address. Now all we have to do on process deletion, is to identify which pages are currently being swapped out/in (this is why we have a queue containing all the swapping page items) and mark them so we can detect that the corresponding process was actually deleted and act appropriately.