SOS

Simple Operating System

30th May 2008
Advanced Operating Systems Course
Project Documentation

Andreas Kägi
akaegi@student.ethz.ch

Christian Oberholzer
coberhol@student.ethz.ch

Table of Contents

Chapter I: Overview....4

1. Design Decisions....4
2. System Architecture....5
3. SOS Servers....6

Chapter II: System Details....9

1. Boot Process....9
2. System Call Dispatching....10
3. Virtual Memory Management....12
4. Process Subsystem....14
5. Loading Binary Files using the Binary Server....17
6. I/O Subsystem....17
7. Networking Subsystem....20

Chapter III: Limitations....22

Chapter IV: Testing....24

1. Built-In Tests....24
2. Test Programs....24
3. Conclusion....25

Abstract

This report documents a simple operating system (SOS) that was built as a course project within an Advanced Operating Systems course. The goal was to build an operating system that builds upon the L4 micro kernel. SOS should feature process management with multiple processes residing in virtual memory, paging and swapping, console access (GNU Netcat), file I/O (NFS) and a clock driver. SOS was required to implement a simple POSIX-style client interface enabling the user to do I/O, process- and time management.

The decision was made to design SOS, very much in spirit of the L4 micro kernel, as a multi-server multi-threaded operating system. This lead to a more complex system but in return gave better modularity and maintainability. The complexity of the system was particularly apparent in the need to carefully design the interdependencies between different SOS server threads in order to prevent deadlocking the system. But the modularity also had some advantages: whereas in a monolithic framework it would be easy to block the root server with some page faults and difficult to resolve this problem, in our modular framework this was no problem at all, as all complex functionality lies outside of the root server.

All of the above listed features have been implemented. We conclude that, even though the initial effort was considerable (for example moving the networking subsystem out of the root server!), we are sure that the efforts payed off in the long run: during the whole project period we never had to significantly modify important aspects of the system due to design "errors".

It goes without saying that the system is far from being mature enough to be used in a productive environment. Moreover, some important components are missing such as a privilege management system for the SOS servers.

Chapter I: Overview

1 Design Decisions

The initial framework provided for the Advanced Operating Systems course had a monolithic single threaded design. All functionality resided within the root server.

But since L4 is a micro kernel and it looked like a cleaner solution, the decision was made to do a multi-server design. This has some advantages:

- The root server which is the only process in 1:1 mapped physical memory is quite small. In fact the root server consists of only ~4000 lines of code including debugging functionality. It does not do any dynamic memory allocation either which would otherwise be quite difficult to implement.
- Since the SOS subsystems are implemented within separate servers the overall system design and communication between the subsystems is clean and easy to understand.
- The servers implementing the OS subsystems are typically quite small and easy to manage and debug. They also reside within virtual memory enabling them to use dynamic memory allocation with malloc and free.
- Finally the idea of the micro kernel with its small trusted code base is carried from the kernel into the whole operating system.

It also has some drawbacks

- Since the design is heavily multi-threaded and parallel, care has to be taken to avoid deadlocks and race conditions.
- Interdependence between different servers may pose problems like deadlocks or the order in which to start the servers.
- It is more difficult to implement to implement a multi-server architecture, than a monolithic one.

Due to the advantages given the decision was made to design SOS using the multi-server paradigm.

2 System Architecture

As a result of the multi-server design approach, the system architecture is quite complex. The following diagram (Illustration 1) gives an overview of the architecture of SOS. It shows the different server threads and dependencies between them. A dependency A → B is given if thread A sends an IPC message to thread B in its lifetime. The dotted arrows denote dependencies that are less important in some sense: either a dependency on the synchronisation server or a dependency on the task server due to thread creation alone.

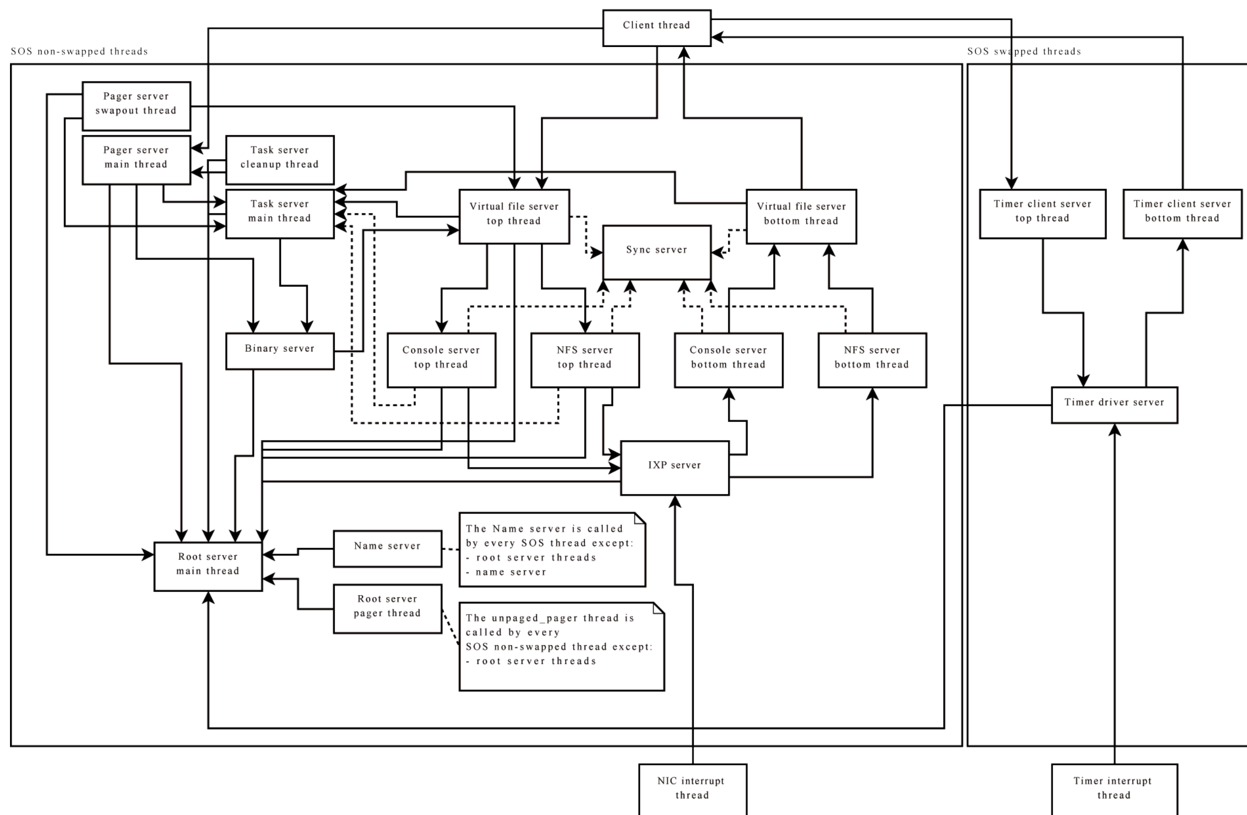

Illustration 1: Interdependence diagram of SOS server threads

Some noteworthy points:

- The dependencies on the root server thread are drastically reduced compared to a monolithic system. This is particularly apparent in this diagram. Other servers (task server and virtual file server) are comparably important (when the importance is measured by the number of dependencies on a thread).
- This diagram nicely illustrates how we tackle the problem of possible deadlocks. Deadlocks may occur when a thread A depends on a thread B (A wants to send IPC messages to B) and directly or indirectly thread B also depends on thread A. In this scenario it is possible that the two threads both wait for each other's reply message after having sent each other a message. The idea is that deadlocks cannot occur if no two threads are mutually dependent. A mutual dependency between two threads is seen in the diagram as a cycle. It is thus required that the interdependence diagram is cycle-free.

These deadlock problems emerged particularly in the following parts of the SOS system:

- I/O subsystem (subdivision of virtual file server, console server and NFS server into a bottom thread and a top thread). This is discussed in more detail in the section on the I/O subsystem.
- Task server/Pager server:
  Both the task server depends on the pager server (to notify the pager server to free frames for a

killed process) and the pager server depends on the task server (various dependencies, say to inform the task server about an allocated frame (stat statistics) or to create a new thread).
In this case the mutual dependency is resolved by making both servers two-threaded. After the split the dependencies are as follows:
Task server cleanup thread → Pager server main thread
Pager server main thread→ Task server main thread
Pager server swapout thread → Task server main thread
As can be seen, no cycles remain.

- Timer client server/Timer:
  Since the timer client server communicates with the timer driver server and vice versa, it was necessary to split up the timer client server into two threads to prevent deadlocks. Like the I/O servers it has a top thread and a bottom thread. The top thread receives requests from clients and delegate them to the timer driver server. The bottom thread receives the answers from the timer driver server and sends them back to the client.

The drawback of splitting up a server into multiple threads is the additional performance overhead due to synchronisation. First, it results in a lot of IPC messages to the synchronisation server[1]. But more importantly, it lowers the potential of possible parallelism by forcing sequentiality of the critical code sections.

3 SOS Servers

3.1 Root Server

L4 starts one process after it has successfully booted. This process is the only process allowed to execute privileged kernel calls. It is called the root server. Since SOS is designed as multi-server operating system the root server has only very few basic tasks.

First of all the root server provides functionality to access the privileged l4 kernel calls to other servers. It also manages access to these privileged calls to prevent not authorized access that may compromise system integrity.

Second the root server manages the physical memory and implements a basic pager for non-swapped processes.

Third the root server boots the systems. It starts all the other operating system servers. Thus booting the systems.

3.2 Name Server

To access SOS functionality it is necessary to send system calls to the appropriate system server. For example all the file operations have to be sent to the file server. To do that the user process needs to know the L4 thread id of the system server able to process the user request.

The name server manages these thread ids. He provides simple naming functionality. A unique name can be associated with whatever l4-thread-id the system chooses when a system server is started. The name servers purpose is similar to that of a DNS on the internet.

To ensure that the name server is always reachable it has a well known l4 thread id. The root server guarantees this.

3.3 Synchronisation Server

Within a heavily multi-threaded environment like SOS it is important to have powerful synchronisation

1 This problem could be weakened by using monitors or a similar synchronisation primitive that is implemented inside a process alone.

primitives. The synchronisation server provides two primitives allowing to synchronise processes and threads as desired. The primitives are:

- *sos_mutex_t*: A mutex can guard critical code sections that have to be synchronised between threads and processes.
- *sos_event_t*: Implements events between threads. Events are similiar to Windows events. One or multiple threads may wait for an event to be signalled by another thread. When signalling, the event may carry user specified data from sender to receiver.

SOS synchronises the system exclusively using these two primitives.

3.4 Task Server

The task server has the responsibility of creating and destroying processes and threads.

Additionally, user processes or other SOS servers may attach custom data to processes and threads using the task server.

3.5 Networking Server (IXP Server)

The terms networking server and IXP server may be used interchangeably throughout the document.

Originally the networking server was designed to encapsulate access to the intel networking driver provided by the project framework. But due to the high coupling between the intel networking driver, IP stack (lwip), libserial and the NFS (network file system) it turned out to be impossible to extract the libraries built on top of the intel driver out of this server within the small time frame given for the project.

The IXP server is therefore in contradiction to the chosen multi-server design a monolithic mixture of

- Intel networking driver
- IP stack (lwip)
- libserial
- NFS implementation

Despite the original design the IXP server encapsulates not one but all four mentioned libraries. Especially the NFS part of the server is quite large.

3.6 Virtual File Server

The purpose of the virtual file server is to provide an abstraction from any particular file system implementation. From a client perspective, it presents a uniform interface to access and modify files with functionality similar to the Standard C I/O API. From the perspective of a file system implementor, the virtual file server is a mediator that manages file meta data (file descriptors and open files table).

In SOS, there are two file system implementors: the console server and the NFS server.

3.7 Console Server

The console server is a simple backend to the virtual file server. It handles the files `"stdout"` to write and the file `"console"` to read and write a simple console. The console used is GNU Netcat. The Netcat console must be situated on host *192.168.0.1* and listen on UDP port *26706* (port *AOS06*).

Implementation-wise the serial library (the library communicating with Netcat) is part of the IXP server, so a write call is delegated to it, whereas a read from Netcat is received by the IXP server and delegated to the console server.

3.8 NFS Server

The NFS server is the second back end to the virtual file server. It handles file I/O in combination with the IXP server. Due to the fact that it has been impossible to extract the NFS implementation from the IXP server this NFS server is quite minimalistic. It mainly forwards virtual file server calls to the IXP server and does some bookkeeping for opened files.

3.9 Binary Server

The binary server's task is to load ELF binary files from the file system and to map the code and data segments into the appropriate address space when the task server creates a new process.

3.10 Pager Server

To handle page swapping the operating system has to keep a relation between a process and all the pages owned by the process. It has to remember whether a given page has already been swapped out or if it still resides in physical memory. The implemented swapping algorithm may require the operating system to keep even more data about processes, pages and/or frames.

The root server is unable to keep this information because the data is a consolidation of root server and task server data. The solution to this problem is the pager server. It provides a more advanced pager which is used for any user process. Through this new pager it enables the operating system to swap pages from memory to disk storage and vice versa.

3.11 Timer Driver Server[2]

The timer driver server implements a timer driver (clock driver) for the timers that are part of NSLU2's hardware. It provides a

- *current real-time clock value* (time stamp), i.e. the current time in microseconds since booting. It is implemented using the Time-Stamp Timer register (`OST_TS`) that is part of the Intel® IXP42X Product Line.
- *client-programmable timer*: a client can register with the timer driver server to be woken-up after a certain delay (in microseconds). This is implemented using the General-Purpose Timer 1 registers (`OST_TIM1` and `OST_TIM1_RL`).

The timer driver server maintains a list of clients to be woken-up by IPC (priority queue). The hardware timer register `OST_TIM1` is always set for the earliest waiting client.

SOS clients do not directly communicate with the timer driver server. Instead the access is mediated by the timer client server (to abstract from this particular driver).

As a *bonus feature*, SOS loads the timer driver server from the file system. This is not much of a problem as the driver resides in virtual memory outside the root server anyway. The only concern is to how the driver can access the memory-mapped hardware registers. It is achieved by the functions `sos_iomap` and `sos_iounmap` that map a physical page uncached and one-to-one into a virtual address space.

3.12 Timer Client Server[3]

The timer client server provides an abstraction from the timer driver server. It provides uniform access to the timer functionality for clients. It ensures that the interface remains stable, even when the backend driver would change. Another reason for the separation of the timer client server from the timer driver server is the need for a synchronous client interface: `sos_timerserver_client_sleep` (timer client server) blocks the calling client until the sleep delay has expired, whereas `register_timer` (timer driver server) is an asynchronous function that returns immediately but sends a wakeup IPC once the delay has expired.

2 For historical reasons the timer driver server is named sos_timerserver in the source code.
3 For historical reasons the timer client server is named sos_timerserver_client in the source code.

Chapter II: System Details

1 Boot Process

Booting an operating system is not an easy task. Especially since some components of the operating systems may have circular dependencies on each other. The SOS boot process consists of roughly three parts. This chapter describes them in detail.

1.1 Booting the L4 Kernel (Pistachio)

As a first step RedBoot bootloader copies the SOS system image into local memory using an SFTP connection and starts the L4 kernel. The kernel boot process is not part of the advanced operating systems course but it is important nevertheless. After the kernel has started successfully it loads the root server and calls its main function. With the call to the root servers main function the boot process goes into its second state.

1.2 Initializing the Root Server

This step involves the initialization of the root server and the transition to the last boot step. The root server initialises two components:

- *The frame allocator* is responsible for physical memory management. With the aid of the frame allocator, the root server allocates physical memory for other processes.
- *The pager* is used to handle access violations of all SOS system servers.

When these components are initialised the root server starts a new thread. The so called *init thread*. This thread carries out the third boot step.

1.3 Creating and Initialising System Servers

This final step creates and initialises all SOS system servers. They are created in this order:

a) Name server

b) Synchronisation server

c) Task server

d) IXP server

e) Virtual file server

f) Console server

g) NFS server

h) Binary Server

i) Pager server

j) Timer server

k) Timer server client

l) SOSH

A description of most of the servers mentioned can be found in one of the following chapters. The servers are started in an order which satisfies the dependencies of any of the servers given. Even though special care has been taken to design the components well, it was impossible to avoid all problems (For example the

basic code to start processes still resides within the root server to start the first three servers).

With the last step l) the SOS shell is executed and the system ready to be used.

2 System Call Dispatching

The multi-server architecture of SOS makes system call (syscall) dispatching an easy task: The syscall IPC is just sent to the server that handles the syscall.

As an example consider a client thread issueing the syscall:

```
in = open("console", FM_READ | FM_WRITE);
```

Client syscalls are defined in libsos/sos.h and implemented in libsos/syscalls.c. Most of the time, the implementation just delegates the call to the client side implementation of the corresponding SOS server. This is the case in the example, open merely delegates the call to `sos_vfs_open` (libsos/vfileserver.c).

```
int sos_vfs_open(
                const char *path,
                fmode_t mode,
                fildes_t* new_file) {

        int res = 0;
        L4_ThreadId_t vfileserver;
        res = sos_vfileserver(&vfileserver);
        ...

        L4_ThreadId_t vfileserver_bottom;
        res = sos_vfileserver_bottom(&vfileserver_bottom);
        ...
        assert(sizeof(sos_syscall_vfileserver_open_ipc_t) ==
sizeof(sos_small_syscall_t));
        sos_syscall_vfileserver_open_ipc_t ipc;
        ipc.a.path_length = strlen(path);
        ipc.a.path = path;
        ipc.a.mode = mode;

        res = sos_small_syscall_split(
                        vfileserver,
                        vfileserver_bottom,
                        SOS_SYSCALL_VFILESERVER_OPEN,
                        &ipc
                );
        ...

        // unpack answer
        *new_file = ipc.r.new_file;
        return ipc.r.retval;
}
```

This is a very prototypical syscall. First, the corresponding server(s) is/are retrieved, then an IPC with appropriate arguments is fabricated an then delivered to corresponding server. Finally, the answer is unpacked and the result returned.

Some remarks:

- `sos_vfileserver(&vfileserver)` retrieves the L4 thread id of the virtual file server: It is cached in a static variable; if not yet set, the thread id is retrieved by querying the nameserver for it using the syscall `sos_nameserver_resolve` with argument `"sos_vfileserver"`.
- IPC messages are normaly sent using the helper functions from libsos/sosutil.h, in this case `sos_small_syscall_split`. IPC messages are marshalled/unmarshalled by storing them in a special struct specific to every IPC message. In our example, the struct `sos_syscall_vfileserver_open_ipc_t` is defined as:

```
typedef union {

        struct {
                L4_Word_t path_length;
                const char* path;
                fmode_t mode;
        } a; // return
        struct {
                int retval;
                fildes_t new_file;
        } r; // return
        L4_Word_t regs[SOS_NUM_NATIVE_REGISTERS];
} sos_syscall_vfileserver_open_ipc_t;
```

- The sosutil helper functions just prepare `L4_Msg_t` message, load all data from the struct into it, load the message and send them with the appropriate L4 IPC operations.

For completeness the definition of `sos_small_syscall_split` follow:

```
int sos_small_syscall_split(
                L4_ThreadId_t component,
                L4_ThreadId_t reply_component,
                L4_Word_t label,
                void* ipc_ptr) {

        sos_small_syscall_t* ipc = ipc_ptr;

        L4_MsgTag_t tag;
        L4_Msg_t msg;
        L4_MsgClear(&msg);
        for (L4_Word_t i = 0; i < SOS_NUM_NATIVE_REGISTERS; ++i) {
                L4_MsgAppendWord(&msg, ipc->reg[i]);
        }
        L4_Set_MsgLabel(&msg, (label<<4));
        L4_MsgLoad(&msg);

        L4_ThreadId_t myself = L4_Myself();

        // Closed send
        tag = L4_Send(component);
        if (L4_IpcFailed(tag)) {
                ...
        }

        // Closed receive
        tag = L4_Receive(reply_component);
        if (L4_IpcFailed(tag)) {
                ...
        }

        // At this point we have received the answer. unpack it
        L4_MsgStore(tag, &msg); // Get the tag
        if (TAG_SYSLAB(tag) != label) {
                ...
        }

        for (L4_Word_t i = 0; i < SOS_NUM_NATIVE_REGISTERS; ++i) {
                ipc->reg[i] = L4_MsgWord(&msg, i);
        }

        return 0;
}
```

3 Virtual Memory Management

This chapter describes the SOS virtual memory management in greater detail. Virtual memory management mainly consists of the swapping functionality which has a great impact on the overall design of the operating system. The chapter is divided into three parts. The first part outlines the general design idea. The second part is dedicated to the memory bookkeeping structure the hat. Finally, the third part describes the implementation of the swapping functionality.

3.1 Separation of pagers

To avoid circular dependencies and difficult situations within the operating system core the design is built around two basic ideas

- Division into "user-level" and "kernel-level" processes. In general so called "kernel-level" servers are non-swappable. Therefore they can use the basic pager implemented within the root server. They are guaranteed to be never swapped out of main memory and, for this reason, do not create circular dependencies on page faults. "User-level" tasks may build upon this safe infrastructure. They use a new pager implemented by the pager server and they must never be involved within the process of swapping. Therefore file system drivers for example have to be implemented always on "kernel-level". Special care has to be taken to declare as few servers as possible as "kernel-level" servers since they occupy system memory that may never be swapped out. Illustration 1 visualises the distinction between "user-level" and "kernel-level" processes. The drawback is that from the perspective of resource usage there may be better solutions to solve the problem. But from a software engineering and debugging perspective this layered approach has the advantage of making everything easier to deal with.

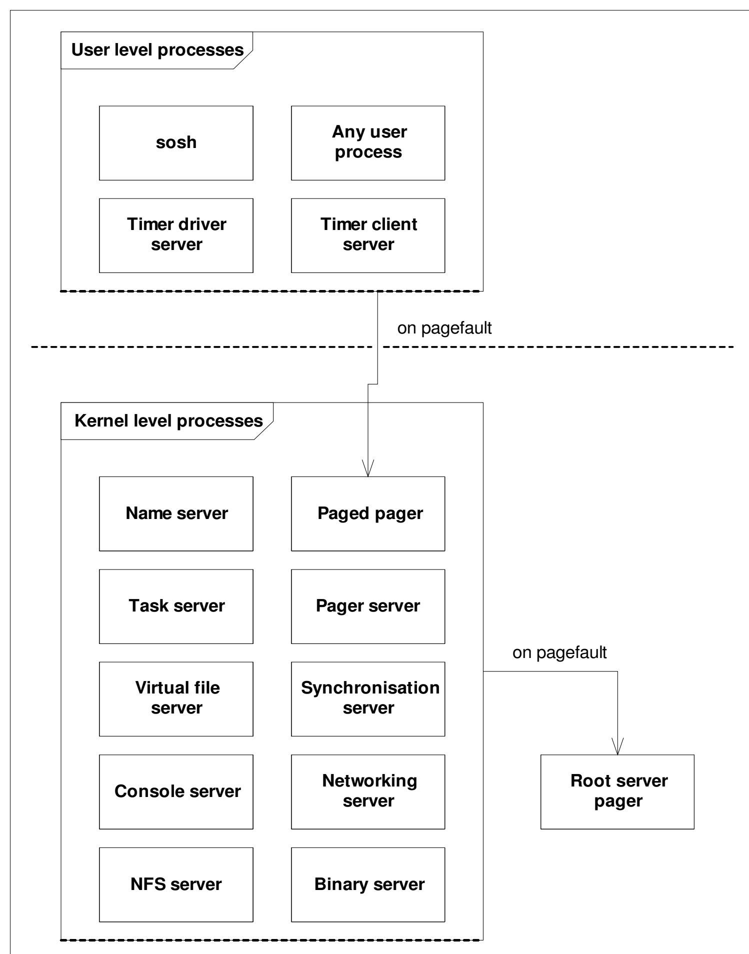

Illustration 2: Distinction between "user-level" and "kernel-level" processes

- The pager server always keeps a certain amount of spare frames. It starts to swap out frames when the amount of free system memory drops below a certain swapping threshold (for example 250 frames = 1 MiB). It may happen that a "user-level" process allocates memory faster then the system is able to swap out frames. In that case the system blocks the user thread whenever he tries to allocate frames below a certain minimum threshold (for example 125 frames = 0.5 MiB). This allows SOS to guarantee "kernel-level" threads the freedom to allocate memory during the swap-out process. The solution with two thresholds allows user processes to carry on without blocking whenever memory needs to be swapped out instead of blocking the whole system. This

should result in noticeable performance gains.

3.2 Page-Table Structure

SOS's page table resides in virtual memory inside the pager server. The ability to do swapping is the only reason that makes managing a separate page table for SOS necessary. Without swapping, one could rely on the page table provided by L4 and use `L4_GetStatus` to retrieve the physical address corresponding to a virtual address. Indeed, this is done in the unpaged_pager (non-swapping pager) within the root server.

Within the paged_pager (swapping pager) the page table is mainly used to keep track of which pages are swapped out and which are not.

The page table is realised as a hash table with the following key and value types:

```
typedef struct {
        sos_pid_t process;
        L4_Word_t fpage_base;⁴
} sos_hat_key_t;


typedef struct {
        L4_Word_t frame_addr;
        /*
         * if swapfile_index == 0
         *            frame not paged out
         * if swapfile_index > 0
         *            frame paged out and index
         *            into swapfile == swapfile_index-1
         */
        L4_Word_t swapfile_index;
} sos_hat_value_t;
```

The page table was originally called a HAT (hash-anchor table) as it was used to do the inverse lookup (virtual address → physical address) for the inverted page table (frame table) that was managed in the root server.

Currently it is used only within the paged_pager to retrieve the frame address (physical address) of a given virtual address and to determine whether that frame is swapped out or not.

One could implement the page table for each process separately. This would result in a more complex implementation but would have the advantage of a speed-up in the implementation of `sos_pagerserver_hat_remove`: This function is called when a process is deleted to remove all of its entries from the page table. Currently, the whole page table has to be traversed for this.

3.3 Swapping

Swapping is done according to the second-chance page-replacement algorithm. This algorithm inserts all pages residing in physical memory into a so called page queue. Whenever a new page is allocated it is inserted at the end of the queue and marked as referenced. Whenever the pager server needs to swap out a page it performs the following algorithm:

m) Take and remove the page at the front of the page queue.

n) If the element is marked as referenced erase the referenced bit and insert the page at the end of the page queue.

o) If the element is not marked as referenced, swap it out.

p) Go back to a) if either no page has been swapped out or more pages have to be swapped out.

The actual work of swapping out a page is delegated to the so called swap out thread. The pager server itself only swaps in pages or issues swap out order. Swap orders are delivered from the pager server to the swap out thread using a synchronised queue. Pages residing within the swap out queue have to be handled

4 Virtual page number (Virtual address without the page offset bits)

specially. If they are referenced before they are actually swapped out by the swap out thread they are later reinserted into the page queue instead of swapping them out.

4 Process Subsystem

The Process subsystem of SOS the basic functions `process_create`, `process_delete`, `process_wait`, `process_wait_2`, `process_status`, `my_id`, `thread_create`, `thread_delete`, `thread_wait`, `thread_wait_2` and `my_thread_id`. These functions are an extended set of deliverables for the SOS process management. The extensions are kernel level threads and exit codes for processes and threads. The functions are implemented within the statically linked SOS-library. They are implemented using specific task server system calls. This chapter is further divided into an overview and detailed design descriptions.

4.1 Overview

A process (or task) is designed within SOS to represent a virtual address space and to be a container for a collection of threads running within this virtual address space. The address space is divided into a region containing the executable binary including its global data, a region to map memory pages from other processes, a region for the heap and finally a region for every threads stack.

To run code within the process a new thread has to be started. The user may start as many threads within the process as he would like (up to the limit of available thread ids).

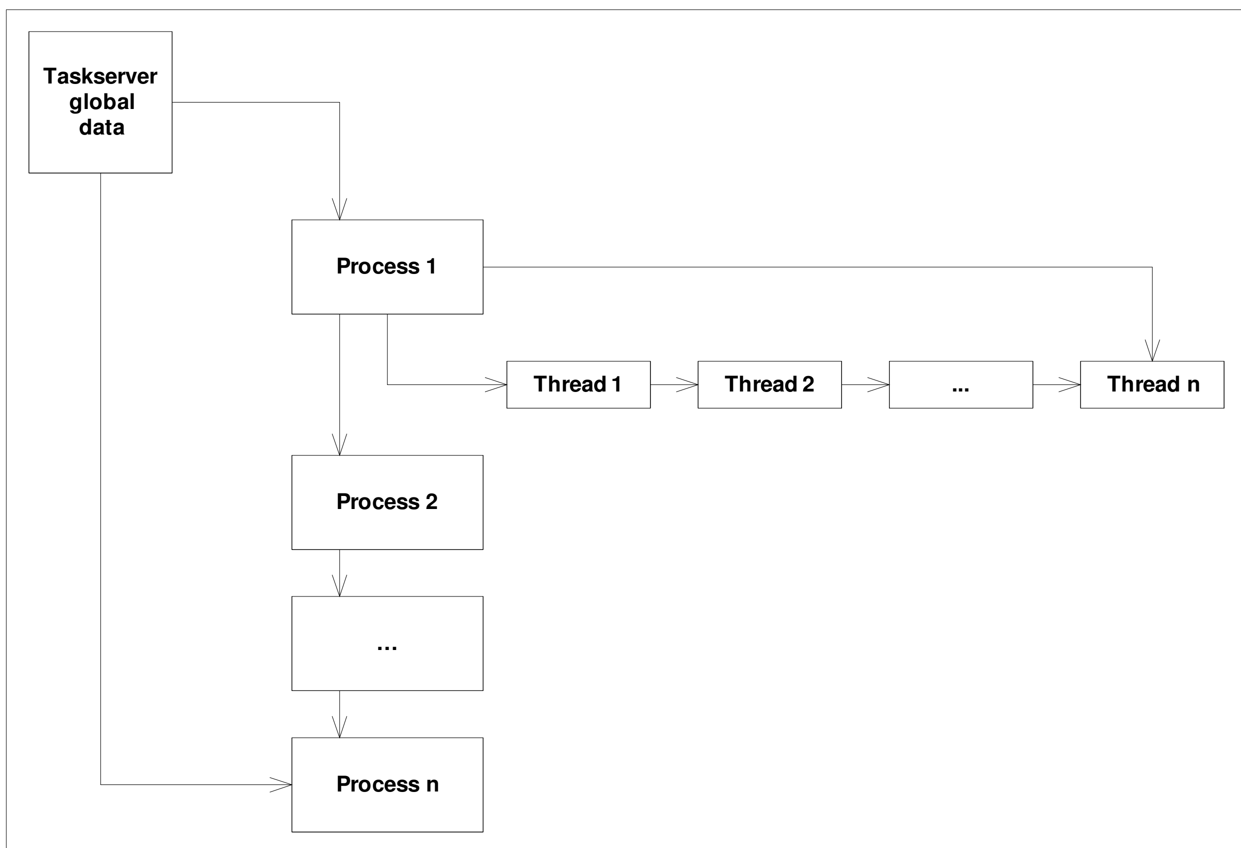

Illustration 3: relationship between processes and threads

Illustration 3 shows the relation between processes and threads as described above. Be aware that processes are semantically different inside SOS to what they appear to be when looking at the `process_create` system call. The differentiation between process and threads running within the process allow SOS to implement kernel level threads quite easily.

4.2 Clean-up Thread

Whenever a process is killed the operating system has to clean up some resources like frames allocated to the process or files opened (and not yet closed) for the process. The clean-up thread is used for this purpose. Whenever a process is killed, the task server enqueues a clean-up action to be processed by the clean-up thread.

Currently the clean-up thread only frees frames allocated to the deleted process.

4.3 Processes

For each process the task server allocates a new structure and links it into a global linked list of processes. The process structure is

```
struct sos_taskserver_process {
        sos_pid_t pid;
        L4_Word_t is_paged;
        L4_Word_t first_thread_id;
        L4_Word_t current_thread_id;
        sos_vector_t free_thread_ids;
        sos_hashtable_t associated_values;
        char* exec_name;
        L4_Word_t page_count;
        sos_event_t wait_event;
        L4_Word_t entry_point;
        L4_Word_t text_address;
        L4_Word_t text_size;
        L4_Word_t rodata_address;
        L4_Word_t rodata_size;
        L4_Word_t data_address;
        L4_Word_t data_size;
        L4_Word_t bss_address;
        L4_Word_t bss_size;
        // double linked list of all threads contained within this task
        sos_taskserver_thread_t* first_thread;
        sos_taskserver_thread_t* last_thread;
        // links to prev/next task in global tasklist
        sos_taskserver_process_t* next_task;
        sos_taskserver_process_t* prev_task;
};
```

The fields are

Field Name	Usage
pid	SOS Process Id
is_paged	Flag indicating if this is a non-swapped or swapped process. This flag indicates which pager is later assigned to any thread within this process.
first_thread_id	Next process local thread id to assign to a new thread created within the process.
free_thread_ids	A list of currently free process local thread ids.
associated_values	A hashtable containing the process local storage. If the user wants to associate some information with the process this information is stored within this table.
exec_name	Name of the executable binary mapped into this process.
page_count	The number of pages currently occupied by the process.
wait_event	Any thread executing process_wait waits for the process using this event.
entry_point	Pointer to the processes entry point.
text_address, text_size, rodata_address, rodata_size, data_address, data_size, bss_address, bss_size	These fields store information about the layout of this processes executable binary.
first_thread, last_thread	Linked list of threads contained within the process

Field Name	Usage
next_task, prev_task	Links to the previous respectively next process within the global process list.

The system calls to manipulate processes are:

- `sos_taskserver_process_create`
- `sos_taskserver_process_create_paged`
- `sos_taskserver_process_nonpaged`
- `sos_taskserver_process_kill`
- `sos_taskserver_process_get_wait_event`
- `sos_taskserver_process_get_id`
- `sos_taskserver_process_ls_store`
- `sos_taskserver_process_ls_lookup`
- `sos_taskserver_process_ls_delete`
- `sos_taskserver_process_set_binary_layout`
- `sos_taskserver_process_get_binary_layout`

4.4 Threads

SOS describes every runnable entity within the environment using a thread structure. The thread is embedded within a process as desccribed above. The thread structure is

```
struct sos_taskserver_thread {
        sos_taskserver_process_t* task_info;
        sos_tid_t tid;
        L4_Word_t local_thread_id;
        L4_ThreadId_t l4_thread_id;
        sos_hashtable_t associated_values;
        sos_event_t wait_event;
        // links to prev/next thread in tasks threadlist
        sos_taskserver_thread_t* next_thread;
        sos_taskserver_thread_t* prev_thread;
}
```

The fields are

Field Name	Usage
task_info	Pointer to the process description of the process owning this thread
tid	SOS thread id
local_thread_id	Local thread id. This id is zero based and only valid within the process context.
l4_thread_id	The L4 thread id given to the L4 kernel.
associated_values	A hashtable containing the thread local storage. If the user wants to associate some information with the thread this information is stored within this table.
wait_event	Any thread executing thread_wait waits for the thread using this event.
next_thread, prev_thread	Links to the previous respectively next thread within the process thread list.

4.5 L4 Thread Id

L4 thread ids are composed of two parts. One part is the global id and the other part is a version field. The task server manages global ids for SOS. It also misuses the version field to store the local thread id within that field. The local id is needed to calculate a thread's stack position.

5 Loading Binary Files using the Binary Server

When creating a new process, SOS has to ensure that the corresponding code and data segments from the ELF file are mapped into the virtual address space of the new process at the correct addresses. However, SOS allows creating an empty process with no threads in them (`sos_taskserver_process_create`). Thus, it is easiest to let an external server do the work of loading and mapping the segments. This is the task of the binary server.

A client syscall `process_create` performs the following steps:

1. It creates an empty process using the syscall `sos_taskserver_process_create`.
2. It loads the binary into the binary server using the syscall `sos_binaryserver_load_binary`.
3. The binary layout returned by the binary server is reported to the taskserver.
4. A thread is created (and started) within this process: `sos_taskserver_tread_create`.

The binary server is only responsible for step 2: It is given the name of an ELF binary and in turn has to report back the layout of it (the start address and size of the code and data segments). First, it opens the file, determines its size and allocates (private) memory for it. It then loads it into its private memory (using libelf) and stores a reference to this loaded binary in an associative array (key = process id). Finally, it reports back the desired layout information.

Later on, when the thread reads from its code section for the first time and thus triggers a page fault to the swapping pager, the pager instructs the binary server to move the code and data segments into the faulting address space: `sos_binaryserver_move_to_l4_addr_space`.

The binary server looks up the binary for the given process (from the associative array). It then maps the code and data segments of the client address space into its own address space (`sos_map_safe`) and copies the segments to the target location.

6 I/O Subsystem

The I/O subsystem of SOS is built in a POSIX-like style: It provides the familiar functions `open`, `close`, `read` and `write` to access and modify files. All of the client I/O functions are synchronous, blocking operations.

`open` returns a file descriptor, a small non-negative integer, that can be used in subsequent I/O calls, like `read` or `write`. A file descriptor is an indirect index into the system-wide table of open files (OFT). File descriptors and the open files table are both managed by the central component of the IO subsystem: the virtual file server.

The virtual file server is also the component that implements the above mentioned client I/O interface (see Appendix for details), namely:

- `sos_vfs_open`, `sos_vfs_close`
- `sos_vfs_read`, `sos_vfs_write`
- `sos_vfs_seek`, `sos_vfs_tell`
- `sos_vfs_dirent`
- `sos_vfs_stat`

Moreover, the virtual file server provides a second interface for implementors of a particular file system:

- `sos_vfs_add_implementor`
- `sos_vfs_remove_implementor`

Thus the virtual file server maintains a list of implementors. It then assigns an implementor to each opened file to which it delegates all client requests that operate on that particular file. The assignment is done by sending a `can_handle_file` IPC (containing the file name) to every known implementor. The first implementor returning `true` is taken as the file's implementor.

6.1 I/O dispatching using the virtual file server

The functioning of the virtual file server is best explained by example:

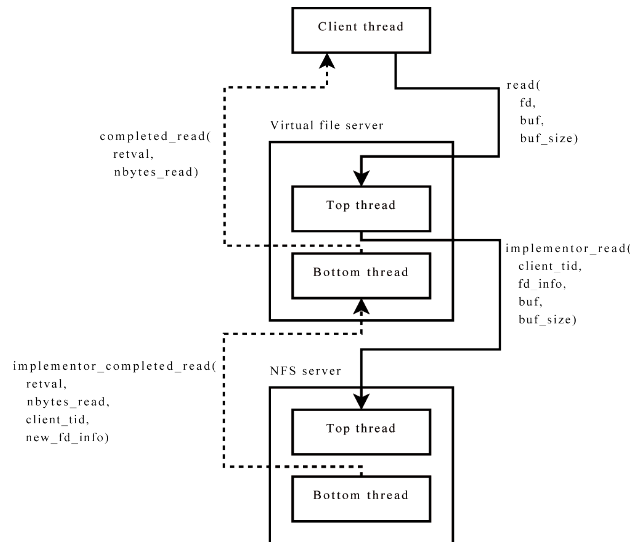

Illustration 4: Example dispatching of a read syscall

The example illustrates the dispatching of a read system call issued by a client thread. It assumes that the client has already opened the file beforehand and holds a valid file descriptor (fd) for the corresponding file. Some noteworthy comments are:

- The virtual file server passes a `fildes_info_t struct` to the implementor. This allows the implementor to store some process-specific information needed to handle the file with the file descriptor. The info struct also contains the global open file index allowing the implementor to uniquely refer to the file denoted by the descriptor.
- Note how the synchronous blocking I/O client interface (POSIX style) is converted into asynchronous operations by the virtual file server: Its top thread delegates the client read request to the NFS server but sends no reply to the client. Instead, the reply is generated asynchronously by the NFS server that passes it to the virtual file server (bottom thread!) which in turn passes it to the client. This requires that
  - the client thread id is passed to the implementor and back from the implementor to the virtual file server.[5]
  - the client threads sends its IPC to the top thread (IPC send) and receives the reply from the bottom thread (closed IPC receive) instead of a normal IPC call.

6.2 File descriptors and the open files table

5 One could instead share the client thread ids in an associative array in the virtual file server. Beside an additional managing overhead this would result in additional synchronisation between the two threads of the virtual file server.

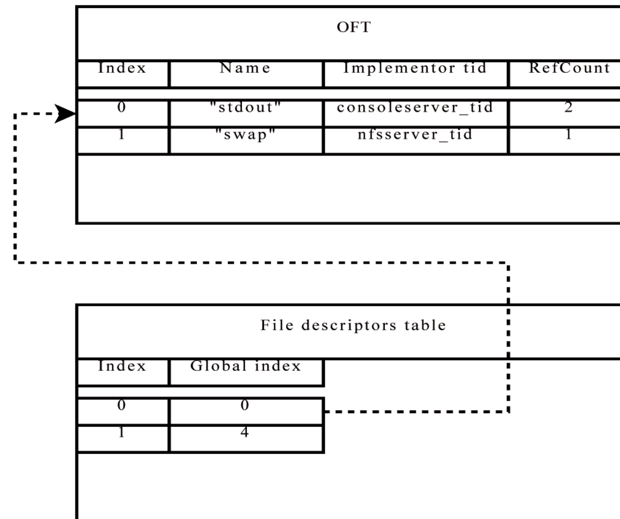

Illustration 5: File descriptors table and OFT

File descriptors are indirect indices into the open files table (OFT). Indirect meaning that the file descriptor is an index into the process-local file descriptors table that contains the global index into the OFT. The file descriptors table is maintained by the virtual file server for every process. Currently, it is of fixed size (`PROCESS_MAX_FILES`, defined in `<sosh.h>`). It resides in the virtual memory of the virtual file server. It is linked to the process by means of the process-specific local storage area provided by the task server (`sos_taskserver_ thread_ls_store`).

The OFT is a system-wide table storing information for all opened files. It resides in the virtual memory of the virtual file server and is shared between its top and bottom threads. The OFT is manipulated by the following operations:

Operation	Location	Description
open	Top thread	Creates a new OFT entry if none is present for the corresponding file, otherwise it increments the reference count.
read/write	Top thread	Look up the implementor tid and delegating the call to the implementor.
add_implementor	Bottom thread	Sets the implementor tid of the OFT entries stdāstdoutÒ and āstderrÒ to the new implementor if this is an stdio_handler.
remove_implementor	Bottom thread	Sets the implementor tid to `0` for any OFT entry having referenced the implementor to remove.
completed_open	Bottom thread	If the open failed and the reference count of the OFT entry is 0, it is removed.
completed_close	Bottom thread	If the reference count of the OFT entry is 0, it is removed.

All operations on the OFT are synchronised since the OFT is accessed by multiple threads.

The OFT is implemented using three data structures:

- a hash table mapping file names to global indices
- a resizable vector (array) representing the actual open files table
- a helper stack with free global indices

6.3 Handling of stdin, stdout and stderr

In SOS the standard file descriptors are named `stdin_fd` (fd 0), `stdout_fd` (fd 1) and `stderr_fd` (fd 2).

For `stdout_fd` an entry "stdout" is always present in the OFT. Initially, this entry references a dummy

implementor (`stdout_null`) that discards all output. As soon as an stdio-handling implementor is registered with the virtual file server, the implementor tid of the "stdout" entry is updated.

`stdin_fd` is currently not supported. Instead, a client process has to open the file "console" explicitly and use the returned file descriptor for subsequent read syscalls.

`stderr_fd` is currently not supported either.

Both `stdin_fd` and `stderr_fd` were not part of the original libsos/sos.h interface and were planned extensions that did not made it into SOS.

7 Networking Subsystem

The networking subsystem consists mainly of the IXP server and the NFS server. As mentioned within the chapter about the different SOS servers, there is no clear line to draw between the tasks of the NFS server and the IXP server due to the fact that it was impossible to factor the NFS library out of the IXP server. File handling is covered by the chapter about the I/O subsystem. Therefore this chapter presents only two special topics.

7.1 Migrate the Intel networking driver to a virtual address space

To port the Intel networking driver the following things need to be considered:

- Changes have to be done inside the folder `ixp_osal/os/l4aos`.
- Privileged calls to L4 have to be changed and delegated to the root server. The L4 calls typically fail silently so care has to be taken to exchange all of them.
- Finally the driver assumes a 1:1 physical to virtual memory mapping to do DMA transfers. This leaves two choices. Either memory for DMA transfer may be allocated directly from the root server still using 1:1 mapping like it is done within SOS or `IX_OSAL_OS_MMU_VIRT_TO_PHYS` and `IX_OSAL_OS_MMU_PHYS_TO_VIRT` in `IxOsalOs.h` have to be changed to implement the correct address translation.

7.2 Splitting large read calls

Since the lwip stack is really simple it does not support splitting packets. Therefore the IXP server may issue only read or write requests filling at most one packet at the time. To limit the user to read at most blocks of packet size limits the read/write throughput to a few kilobytes per second. This is clearly unsatisfactory.

Therefore SOS implements an optimization for large read requests. If the user process requests to read more bytes than a packet can hold the request is split into several read fragments (as shown in illustration 6). For each fragment the IXP server issues a read call to the NFS server. Within the callback function the IXP server fills the buffer from successfully read fragments and repeats read calls for failed nfs_read calls. After the last fragment is

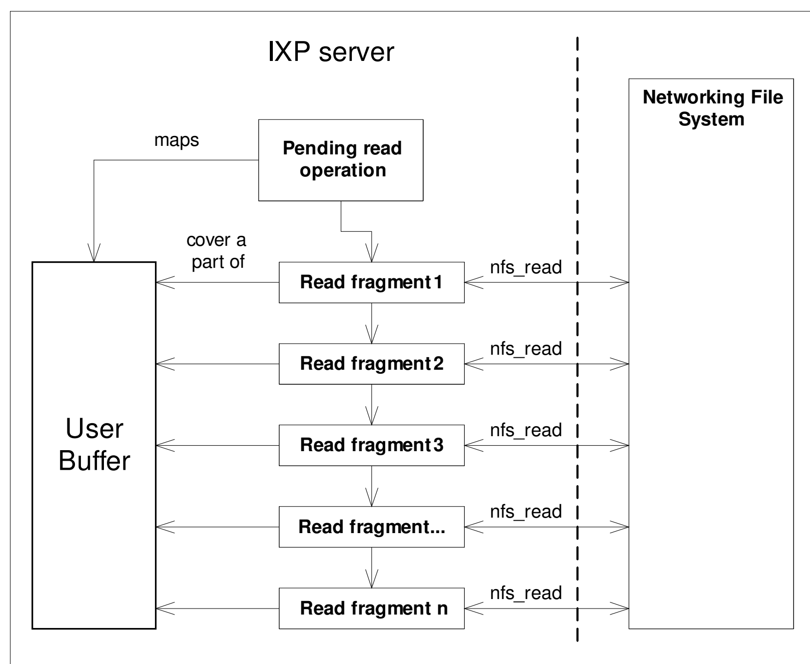

Illustration 6: Split concept

successfully completed, the call will return.

Chapter III: Limitations

This section describes some shortcomings of the current SOS system. The limitations are loosely ordered by importance.

Limitation	Description
Missing priviledges	Currently, all SOS server threads have full access to the root server. That is, they can call any priviledges operation that the root server exposes as a syscall. This is clearly unsatisfactory. Instead, SOS should have some privilege management system that give certain restricted priviledge to the servers and enforces them to not do other unallowed priviledged calls.
Overburden IXP server	The IXP server currently contains code for the NIC driver (ixp_osal), the IP stack (lwip), NFS and the serial library. The goal would be to split it, such that the IP stack resided in its separate server, independent of any network device. Moreover, the NFS and serial libaries should be modified such that they communicate with the IP stack server via IPC messages (and not with the IXP server).
ixp_osal DMA memory hack	Currently, ixp_osal relies on the availability of physical memory (consequtive memory) of arbitrary size. This resulted in a special treatment in the root server: `sos_rootserver_dma_malloc_hack`
Exit codes and `process_wait`	It would be clearly desirable, if `process_wait` returned the exit code of the process's main function. For this purpose, SOS provides `process_wait_2` that returns the exit code. The problem arises when the called process terminates *before* the calling process can initiate `process_wait_2`. In this case, the exit code is lost. With the current implementation it is not possible to resolve this issue, so a new reimplementation would be necessary.
Maximum file name length	In SOS, the length of a file name is restricted to be smaller or equal to `MAX_FILENAME_LENGTH`. This constant is defined in libsos/vfileserver.h
Maximum number of entries in the file descriptor table	The size of the file descriptor table is currently limited. No more than `PROCESS_MAX_FILES` (libsos/sos.h) can be opened at the same time by any process.

1.1 Bugs

This section describes some unresolved bugs within SOS or libraries that SOS depends on.

Bug	Description
Too many parallel reads	If too many parallel reads are done simultaneously, the lwip stack runs of out memory: `Assertion "mem_free: mem->used" failed at line 284 in libs/lwip/core/mem.c`
No cleanup in `remove_implementor` (vfileserver.c)	When an implementor is removed from the virtual file server, all files having this implementor associated should by invalidated in some way, such that subsequent read/write calls will fail. Alternatively another implementor that is able to handle the files could be associated to them.
Correct stderr/stdin handling	The virtual file server should correctly handle stderr and stdin. stderr should be a valid file descriptor for any process (the corresponding file is automatically opened), whereas stdin should only be valid for the process having opened the file ÒconsoleÓ in read mode.
Virtual file server memory leak	The virtual file server has to free the file descriptors table for a process after it has terminated. It should also close any files that remained open for the process. The task server would have to notify the virtual file server in its cleanup thread. As the reference to the file descriptors table is stored in the task local storage of the corresponding process, the taskserver would have to introduce some "zombie" mechanism instead of directly destroying the process.
Too large bss section	When the bss section of an ELF binary is too large, SOS fails correctly running the code of this binary.
IPC/name server problem	Spurious error with registering/querying the name server: Sometimes incomplete IPC messages were fabricated that contained only part of the name (string) to register/query. Apparently, the problem was not the actual send operation of the IPC but the L4 message was already corrupted when loaded into the message registers, even though the structure

Bug	Description
Wrong server configuaration	When the server (192.168.0.1) has no running inet daemon or nfs-server, SOS fails to start and prints no appropriate error message.
Frame locking/unlocking not done	Problems occur whenever the pager server swaps out frames mapped into multiple address spaces. In such a case the frame can get freed but still mapped into server address spaces. Thus if the frame is assigned to another process the processes data may be overwritten by the invalid mapping within the server address space. Even though there is code to handle this case tests have proven it to not work properly.

Chapter IV: Testing

The correctness of all implemented features is, as good as possible during the short time available for testing, verified using two different methods. Firstly there is verification functionality built into SOS and secondly there are some explicit test programs to test SOS. The following sections describe those two approaches.

1 Built-In Tests

This is done through various verification methods executed during the boot process and using assert statements during the code execution. Using lots of assertions to ensure that assumptions about the code are correct helped to prevent many bugs. Built-In Tests are

q) *sosroot_verification_nameserver (sos/verification.c)*: This Verification is run after the name server has been started. It checks that the name server is working correctly. Namely one can insert a new entry, look it up and remove it afterwards.

r) *sosroot_verification_localstorage (sos/verification.c)*: This verification is run after the task server has been started. It verifies the availability of process local storage.

s) *sosroot_verification_syncserver (sos/verification.c)*: This verification is run after the synchronisation server has been started. It verifies the functionality of the two synchronisation primitives used within SOS. The event and the mutex structures.

t) *hat_test (sos_pagerserver/hat.c)*: This verification is run after the SOS page table is initialised. It verifies that inserting, removing and querying items from the page table properly work.

u) *assertions (sos_timerserver/timerserver.c)*: This verification asserts that the structs within the timer server used to represent the timer driver's registers are correctly aligned.

v) *sos_test_atomic_assignments (libs/sos/include/test.c)*: This verification asserts that the atomic assignment functionality (which is written in Assembler) works properly.

w) *sos_test_hashtable (libs/sos/include/test.c)*: This verification asserts that the iterator that is implemented for the hash table works properly.

2 Test Programs

Secondly there are some explicit test programs written to verify special features, test that features work, that they work together or just that the system works correctly under stress situations. The tests are:

2.1 test_all

The program test_all is an approach to do pseudo-automatic testing. It consists of a set of other programs to execute and the expected return codes. It then executes all test programs (one after each other) and collects the programs return codes. It then prints a summary of all executions comparing the expected results with the actual results. This test gave the ability to verify that the system still work as expected after having changes and bugfixes. Additionally this test program is another opportunity to do system stress tests. Instead of starting the tests sequential the test would have the possibility to start all processes at once and thus enable some more system stress testing.

2.2 test_parallel_reads

This test verifies the system to work correctly under the load of lots of parallel large reads. It revealed a bug within page mapping algorithm.

2.3 test_threads

Kernel level threads make it easier to write tests for parallel functionality since only one process (application) has to be written and started instead of many. This test verifies that kernel level threads can be started and executed as designed within the application programming interface. This test revealed a deadlock problem between the timer client server and the timer driver server.

2.4 test_parallel_timestamps

In response to the deadlock problem between the timer servers, this test was introduced. It stresses the timer servers heavily by issueing a lot of parallel time stamp requests from multiple threads.

2.5 test_large_read and test_large_alloc

It was feared that terminating a process that is in the middle of a read syscall could be hazardous. These two tests were introduced to test that. `test_large_read` just reads from a large file of the NFS share. At the same time `test_large_alloc` has to be started. It allocates a large array of memory and reads from it. Since for debugging purposes a freshly allocated frame is always initialised with all zeroes, `test_large_alloc` asserts, that the data it reads from the large array is all zero. In combination, it can be tested whether terminating a process blocking in an active system call leads to corruption of memory.

2.6 test_swapserver

This test verifies that the pager server correctly swaps out and swaps in frames. It allocates a chunk of memory which is bigger than the available amount of physical memory. This buffer gets filled with some reconstructible data. It is later verified that still contains the correct data. Through writing to the buffer and reading from the buffer, the pager server is forced to swap frames dynamically.

3 Conclusion

With this approach to test SOS it was possible to find and eliminate many bugs. A lot more have been prevented through the consisent use of assert statements and the built-in verification functions. Even though a lot of bugs have been discovered, the test cases cover only a small part of the overall system. Therefore many more tests would be needed to verify the system to work correctly in special situations or under heavy load.