Contents

Introduction

Our operating system is named **ROSA**

We tried to use as few threads as possible when designing our root server. Paging, system calls and clock maintenance all happen as part of the root server thread. This means that we have continuations **everywhere**. Continuation data is held in malloced structs that get passed around between functions and (hopefully!) free'd at some point.

We attempted to make our lives a little easier by creating an abstraction for chaining callbacks and their corresponding continuation data structs together - we called this a *Deferred* (see `sos/deferred.h`).

We had some success with deferreds, for example,

Extract from `sos/syscalls/process_create.c`

```
char *path                         = (char*)                msg->msg[1];
int size                           = (int)                  msg->msg[2];
uint32_t entryPoint                = (uint32_t)             msg->msg[3];
struct Elf32_Phdr *programHeaders = (struct Elf32_Phdr *) msg->msg[4];
uint16_t numProgramHeaders         = (uint16_t)             msg->msg[5];

Deferred deferred = deferred_create ();

ProcessCreateCallbackData processCreateCallbackData;
processCreateCallbackData = syscalls_PROCESS_CREATE_createCallbackData (path, size,
 entryPoint, programHeaders, numProgramHeaders, spaceID, threadID, deferred,
 L4_ThreadNo(sos_get_new_tid()));

deferred_enqueueCallback ( deferred,
                           pagetable_pinMemory,
                           pagetable_pinMemory_createCallbackData (spaceID,
 (L4_Word_t) path, size, deferred) );

deferred_enqueueCallback ( deferred,
                           pagetable_pinMemory,
                           pagetable_pinMemory_createCallbackData (spaceID,
 (L4_Word_t) programHeaders, sizeof (struct Elf32_Phdr) * numProgramHeaders,
 deferred) );

deferred_enqueueCallback ( deferred,
                           syscalls_PROCESS_CREATE_afterPinningMemory,
                           processCreateCallbackData);

deferred_enqueueCallback (deferred, syscalls_PROCESS_CREATE_cleanup,
 processCreateCallbackData);

deferred_fireNextCallback (deferred);
```

In this way deferreds allowed us to easily map out the top level structure of a procedure that may involve multiple callbacks.

Unfortunately we were not able to use deferreds everywhere; for example, many of the NFS library functions expected to be given callback functions with specific prototypes that did not match the prototypes of our deferred callback functions. This sometimes led to an unsightly mix of regular callbacks and deferreds which can make the code difficult to understand - the worst offender here is probably `sos/pagetable.c`.

We have created two pieces of documentation for ROSA. The first is this document, which contains details of our design (and known flaws) in a book-like form. The second source of documentation is our commented header files, that have been parsed using *doxygen* to create a nice looking reference manual. We hope you find both sources useful.

Thanks for running this course, and especially thanks to Aaron and Nick for all their help during consultations. We had great fun 🙂

-- John Garland and Rupert Shuttleworth, Very late at night, October 23, 2009

Frames

A *frame* is a contiguous section of physical memory. Our frames are all of size 4096 bytes[1].

We use a *frametable* to keep track of metadata for each frame, such as whether a frame has been allocated, whether a frame can be *evicted* (see **Pages**), the virtual address corresponding to a frame (if any), etc.

Our frametable is an *abstract data type* (ADT). The file `sos/frametable.h` supplies an interface to query the frametable, but gives no implementation details (other than publishing the size of each frame). The file `sos/frametable.c` implements the frametable interface but makes no assumptions about the rest of the operating system.

A practical benefit of using an ADT is is that it allowed us to easily change the implementation of our frametable without breaking the rest of the operating system. It also allowed us to control access to the frametable - everything has to go through an interface function and survive a serious of automatic assertions.

This makes everything a bit slower, but assertions could be disabled a compile-time and functions rewritten to be inline and more optimised if desired. When designing the frametable we didn't really focus on small optimisations, instead we concentrated on making the major frametable operations have constant worst-case time complexity.

That is, allocating a frame and freeing a frame both have constant time complexity. We achieved this by keeping a stack of free frames - when frames are allocated they are popped from the top of the stack, and when frames are freed they are pushed on to the top of the stack.

We store the metadata for each frame in a struct and use an array of these structs to keep track of the metadata for all of our frames. The stack ordering is defined using a 'nextFrame' field inside these structs.

This frame metadata array is stored inside actual frames (and marks those frames used for itself as being allocated). Using frames to store our frame metadata array allows the array size to grow as the total number of frames grows, without needing to take up more of the root server's (fixed) heap memory.

Lastly, our frametable ADT also publishes an interface function to find the next frame for eviction (see **Pages**). Our implementation of this function is O(n) in the worst case, where n is the total number of frames. If we had time to improve the frametable, we would first focus on making our implementation of `frametable_findFrameToEvict()` faster.

Pages

Pages are contiguous sections of virtual memory. We used a page size of 4096 bytes, the same as our frame size[2].

Pagetables

We use a two-level pagetable for user processes. Given a 32-bit virtual address, we take the upper 10 bits as an index into an upper pagetable. This gives us a reference to a lower pagetable, and we take the next 10 bits of our virtual address as a index into that, which gives us a specific `union page`:

Extract from `sos/pagetable.h`

```
union page {
   physicalAddress_t physicalAddress;
   struct {
      unsigned :20; // must keep for frame number
      unsigned :9;  // can be used
      unsigned int isInFrametable :1;
      unsigned int isInSwapfile :1;
      unsigned int isValid :1;
   } x;
};
```

We use the lower three bits of each physical address for housekeeping:

- `isInFrametable == TRUE` means that the frame corresponding to the virtual address is currently in the frametable, in which case the physical address gives us the absolute physical address in RAM (ignoring the lower 12 offset bits)
- `isInSwapfile == TRUE` means that the frame corresponding to the virtual address has been evicted from the frametable and now lives in the swapfile. In this case, the physical address gives us an absolute offset into the swapfile, where we can find the evicted frame (again, ignoring the lower 12 offset bits)
- `isValid == TRUE` means that the pagetable entry has already been set up (e.g. by a previous pagefault) and doesn't merely contain junk data

Conveniently, our upper and lower pagetable arrays fit exactly inside frames, so we use frames to store them rather than chewing up the root server's heap memory. We allocate frames for the upper and lower pagetables on-demand, and mark the frames as unevictable to prevent them from being sent to the swapfile.

Demand paging

Should the frametable fill up, we attempt to make space by sending a frame to the swapfile; the frame is chosen using a second-chance eviction algorithm. This eviction could fail for a number of reasons:

- Firstly, all the frames in the frametable may be either pinned or marked as unevictable, in which case `frametable_findFrameToEvict()` in `sos/frametable.c` would return a `NULL` physical address and the user process who needed a frame would find themselves being deleted.
- A second chance eviction could also fail if the swapfile is full. In `sos/main.c` we specify a maximum swapfile size in frames (currently 4096 frames, which gives us a 16 megabyte swapfile). Similar to our frametable, we keep a stack of free swapfile *slots*. This stack lives in the root server's heap, and allows us to find free slots and release slots in constant time.

If a user process faults on a virtual address whose corresponding frame has been sent to the swapfile, we first have to find a new frame to back it (which may require an eviction), before bringing the old data in from the swapfile. Throughout our code we call this process an *admission* (see `sos/swapfile.c`)

To test demand paging, there is a comment block at the bottom of `sos/main.c` that allows one to limit the number of frames given to the frametable. In our Milestone 8 submission we limited this to 400 frames.

ELF loading on-demand

If a process faults on a region that lives in an ELF file then the necessary data is read in from the ELF-file. The actual regions are set up as part of a `process_create()` syscall (see **System Calls**).

Frames are automatically zeroed when we allocate and free them. If the ELF-file does not contain enough data to fill the frame corresponding to the faulting page, then the frame will appear to have been padded with zeroes.

Clock

Our clock runs as part of the root server thread.

We added two library functions to `libs/clock/include/clock.h` to handle interrupts, aptly named `clock_timestamp_interruptHandler()` and `clock_timer0_interruptHandler()`.

We modified `sos/main.c` to listen for `TIMESTAMP` and `TIMER0` interrupts. When the `syscall_loop()` receives these interrupts it calls the respective clock library interrupt handlers. These handlers do the obvious thing - the timestamp handler updates the current timestamp, and the timer0 handler checks the sleeping queue to see if someone should be woken up; if so, it wakes them up by sending a message to them.

When the clock is created (by `start_timer()`), it maps in the memory-mapped IO registers used by the timer cell, and also registers the root server thread to receive interrupts from the timer cell. The `TIMER0` timer is then started, with a refresh rate of 1 millisecond. This means that the minimum guaranteed sleep time is 1 millisecond. Processes can attempt to sleep for less than this, but we may not wake them up until an entire millisecond has passed.

When we initially wrote the clock we experienced stability issues when using faster refresh rates (the root server would stop receiving interrupts altogether), however it may be possible to go lower than 1 millisecond now that various patches have been applied to the microkernel we are using for COMP9242 (thanks to Bernard, Aleks and co!)

When `register_timer()` is called, we calculate the absolute wakeup time and insert the process into a queue of sleeping processes. The queue is ordered, with processes that are to be woken up earliest appearing at the front of the queue. This ordering means that insertions can be slow, in the worst case the time complexity would be `O(n)` to insert. However, keeping the queue ordered means that the `TIMER0` interrupt handler can be very fast as it doesn't have to check the entire queue when deciding whether it needs to wake processes up.

We felt our design was a good balance as interrupts usually happen much more frequently than `register_timer()` requests. However, if `register_timer()` requests began happening more frequently then `register_timer()` would need to be rewritten to use a balanced tree or another data structure which allowed fast insertions.

System Calls

Overview

All our system calls are done via message passing, using the `L4_Call()` and `L4_Reply()` helper functions.

To perform a system call, a user application first calls a function in *libsos* [3], such as `open()` or `read()`. Typically, this library function will perform some sanity checking in userspace, before proceeding to send one or more messages to the root server, waiting for a reply to each message before sending the next.

Eventually, the library function stops communicating with the root server and control returns to the user application, typically with with some status code.

From the perspective of a user application, all of our system calls appear to block until the desired task is completed. One exception to this is the `process_delete()` system call, which returns to the user application as soon as a process has been marked as a zombie, although that process may still be running.

We wrote our own helper functions, `quickCall()` and `quickReply()`, to simplify the message passing code[4]. These functions take in a recipient (like `L4_rootserver`), a system call tag (like `SYSCALL_PROCESS_CREATE_REQUEST`), the number of words to be attached to the message and sent in the message registers (like 3), and lastly the actual words, if there are any.

Typically a library function will attach 1-3 words to a syscall request message. If a potentially large amount of data needs to be shared, for example a buffer of characters to read or write to, the *libsos* library function attaches a pointer to the buffer. When the root server receives the message, it simulates a pagefault on any pages that the buffer lives in and marks the corresponding frames as *pinned*.

Pinning a frame ensures that the frame is not taken out of the frametable and placed in the swapfile while the system call is still in progress. (See **Pages**). When the system call is over, the frames are marked as unpinned.

We modified the given `libs/sos/include/sos.h` header slightly. Firstly, we moved most of the `#defines` into a separate file, `libs/sos/include/globals.h`, so that the root server could get access to them without needing to know about the functions prototyped in `libs/sos/include/sos.h`.

We also changed some function prototypes, namely, if `size_t` was used as a type, we changed this to `int`. This was done in order to perform some sanity checking in userspace - in particular, to check that we weren't given negative values, as `size_t` seemed to default to be unsigned.

Moving data between physical and virtual addresses

We wrote some helper functions in `sos/syscalls.c` that handle moving buffers of data between userspace and rootspace. Traditionally these sorts of functions might be named `copyIn()` and `copyOut()`, however we found those names to be dangerously meaningless so instead named them `syscalls_copyToRootspace()` and `syscalls_copyToUserspace()`.

The `syscalls_copyToRootspace()` function is used whenever the root server needs a contiguous version of a userspace buffer that might span across several non-contiguous frames. For example, the `open()` syscall takes in a path as an argument, and the path could span several frames.

The `syscalls_copyToUserspace()` function is used whenever the root server has some buffer sitting in its private heap that it wants to copy back to userspace. For example, the `stat()` syscall expects the root server to write to a userspace struct buffer. In the root server, we create a new private struct and write to that, then copy it to userspace when we are done.

Notes on specific system calls

open (path,mode)

If the console is being opened, and the open was successful, then we save its local file descriptor (see **Virtual File System**) as a global variable called `stdin_fd`. If `sos_read` is called, it will use this local file descriptor.

close (localFileDescriptor)

If the console is being closed, and the close was successful, then we update `stdin_fd` to contain `INVALID_FILE`.

read (localFileDescriptor, buf, nbyte) & write (localFileDescriptor, buf, nbyte)

Both of these functions split the virtual buffer up on page boundaries before sending messages to the root server. Therefore, these functions may end up sending several messages before the system call is complete, depending on how many pages the buffer spanned.

This means that processes have to pay a higher cost (in the form of more messages) as they increase buffer sizes.

sleep (milliseconds)

In the root server, we impose a minimum bound on sleeping time (see **Clock**).

process_create (path)

We read the start of the file in to userspace, in order to extract information from the ELF program headers. We then package this information up in userspace structs, and send pointers to these structs to the root server.

The root server then checks whether there is an available address space[5] and if there is, sets up the necessary regions and starts the process running. The actual ELF file segment data is loaded on-demand as part of a pagefault. (See **Pages**)

Virtual File System

Devices

We designed a file system abstraction that allowed us to treat the console and network file system (NFS) files in similar ways, the primary difference being in the `open()` syscall.

Virtual files are backed by an abstract data type called a *Device*. Anything that implements the interface specified in `sos/device.h` can claim to be a device.

In particular,

Extract from `sos/device.h`

```
struct device {
   char *name;
   DeviceData                data;
   DeviceInitCallback        init;
   DeviceOpenCallback        open;
   DeviceCloseCallback       close;
   DeviceWriteCallback       write;
   DeviceWriteFlushCallback  writeFlush;
   DeviceReadCallback        read;
   DeviceSeekCallback        seek;
};
```

A device is something that provides implementations of the above functions. Devices also carry with them a *name* and some private *data*, which can contain anything the device wants.

For example, the *console device* specified in `sos/devices/console.c` uses its private device data to keep track of which serial device is being used. In theory this would allow multiple consoles to exist using different serial ports, however we never tested this. Thinking about it now, it may have been cleaner to use the global file data for each file to specify which serial device should be used (more on that below.)

Our only other device, the *NFS device* (specified in `sos/devices/filesystem.c`), doesn't use the device data field at all.

Local file table

We keep a local file table for every process, as well as a global file table shared across all processes. For each local file, the local file table keeps track of a file *mode* (read, write, or both), a reference to a *global file* in the global file table, and optionally some *local file data*.

Extract from `sos/pagetable.h`

```
struct localFile {
   fildes_t globalFileDescriptor;
   L4_Word_t mode;
   int isActive;
   LocalFileData localFileData;
};
```

The local file data for a file is defined by the device that backs the file, and can contain anything the device wants. Our console device does not use the local file data at all, however our NFS device uses the local file data for each file to store an offset into that file. This allows each user process to have different offsets into a shared file.

Global file table

Our global file table stores the following information for each global (shared) file,

Extract from `sos/vfs.c`

```
struct globalFile {
   Device device;
   GlobalFileData globalFileData;
   char *name;
   int numReaders;
   int numWriters;
   int isActive;
   int numReferences;
};
```

In particular, each global file is given a reference to the device that should be used to operate on the file, the name of the file, some housekeeping data so that we know when we can remove a global file from our global file table, and optionally some *global file data*.

As with the local file data, the global file data is device-specific. Our NFS device uses the global file data for each file to keep track of the filehandle cookie that the NFS library gives us when we call `nfs_lookup` on a file. Our console device uses the global file data to keep track of the number of readers, so that it can enforce that the console is only opened for reading once.

Extract from `sos/devices/console.c`

```
struct globalFileData {
   int numReaders;
};
```

In an alternate design, the console device could have simply used our existing global file housekeeping data directly, as it was already keeping track of the number of readers. Or as yet another alternative, we could have removed the `numReaders` and `numWriters` fields from our housekeeping data and only kept track of the number of references there. In the end we decided to keep our current design, partly because the duplication allowed us to perform some extra sanity checking (we love our assertions!), but mainly because we ran out of time to rewrite it.

VFS layer

The file `sos/vfs.c` implements an abstraction layer that allows the root server to operate on virtual files without needing to know what device is being used to back them. The actual device being used to back a file is only ever explicitly specified when a file is opened.

Extract from `sos/syscalls/open.c`

```
if (strcmp(syscalls_OPEN_continuation->pathContiguous, "console") == 0) {
      device = console;
   } else {
      device = filesystem;
```

```
    }
}
```

Notes on specific devices

Console device

The console keeps a read and write buffer in its device data. The read buffer is one frame (4096 bytes), the write buffer is 200 bytes[6].

When the console is initialised, it registers a handler to receive input from the serial device, and will begin recording input when a user types, even if a read syscall is not currently active. If new user input is received while the read buffer is full, the new input will be dropped. When the read buffer is full, or a newline character is read, the console marks the read buffer as being ready to be flushed.

Flushing the read buffer can be an inefficient process. If a user application has performed a `read()` syscall on the console, but their user buffer is not big enough to contain the amount of characters ready to be flushed, then the user buffer will be filled from the read buffer and any extra characters in the read buffer will be shifted down to the start of the read buffer.

If one were aiming to improve the console implementation, the read flush code could be rewritten to avoid the shifting (perhaps use a circular buffer). We opted not to do this; we were already having so many problems with libserial[7] that we wanted to keep everything we wrote as simple as possible.

The write buffer is flushed whenever our write buffer is full, or when we have seen a newline character. Since we noticed `printf()` was calling `sos_write()` for every character, we also modified `libs/c/src/sys-sos/sys_init.c` to automatically create 256 byte line buffers for `stdin` and `stdout` whenever a user process is created.

One final quirk of our console device is that we ended up placing locks all over the place, primarily to protect access to the read and write buffers. It wasn't clear to us why we had to do this; we assumed the serial callbacks would happen as part of the root server thread and that the locks were unnecessary, but the behaviour seemed incorrect without them.

NFS device

Reminder: the NFS device lives in `sos/devices/filesystem.c`

If a file is being opened and the file doesn't exist, the NFS device will always create it (regardless of whether write permissions are part of the file mode). This was because the comment in `libs/sos/include/sos.h` only stated that "*A new file should be created if 'path' does not already exist*", and made no mention of which file mode was used.

Reading (`filesystem_read()`) and writing (`filesystem_write()`) at the device level operate on physical addresses, and can read and write an unbounded amount of data across contiguous physical addresses. We limit the NFS device to sending or receiving 1100 bytes at a time[8], and physical buffers larger than this are transparently broken up inside the NFS device into multiple `nfs_read()` or `nfs_write()` requests.

In terms of system calls, we break `read()` and `write()` buffers up on page boundaries in `libsos` and translate the page addresses to frame addresses before writing to a file (See **System Calls**).

Every 100 milliseconds, the `init_thread` in `sos/main.c` sends a message to the `syscall_loop` telling it to call `nfs_timeout`; this ensures that unacknowledged packets are resent across the network.

User Processes

In our current design, user threads are explicitly tied to address spaces. An address space has exactly one user thread, and there are a maximum of 16 address spaces[9].

Address space IDs are the same as process IDs, and the IDs are reused when processes (and hence address spaces) are deleted. User threads can not exist outside of an address space, and user threads can not migrate between address spaces.

Much of the code for dealing with user processes lives within `sos/pagetable.c` alongside our regular address space code. We even went so far as to say that we can *run an address space* - see `pagetable_runAddressSpace()` in `sos/pagetable.c`. This function runs the thread which corresponds to a specific address space.

Each process is given a fixed heap region and a fixed stack region:

Extract from `sos/libsos.h`

```
#define STACK_TOP     0xD0000000
#define STACK_BOTTOM  0xB0000000

#define HEAP_TOP      0xA0000000
#define HEAP_BASE     0x80000000
```

These virtual addresses are in hexadecimal - `0x10000000` = 2^{28} bytes = 256 megabytes. So we allow a 512 megabyte stack and a 512 megabyte heap for each process.

Process creation

There are two ways that user processes can be created. The first is via the `process_create()` syscall, which loads a process from an ELF file. The second is by calling `sos_task_new()` in `sos/libsos.c`; this loads processes directly from the boot image. User applications do not have access (either directly or indirectly) to `sos_task_new()`.

Although we have migrated normal user applications to be loaded from ELF files, we kept `sos_task_new()` in order to load a very specific startup application called the `soshloader`[10] directly from the bootimage. The `soshloader`'s job is to load up `sosh` through the normal `process_create()` system call pathway, and them prompty die.

Extract from `userapps/soshloader/soshloader.c`

```
int main (int argc, char *argv[]) {
   pid_t pid = process_create ("sosh");
   assert (pid != PROCESS_CREATE_FAILURE);

   return EXIT_SUCCESS;
}
```

This allowed us to place `sosh` in an ELF file and treat it just like any other user application, and gave us the ability to easily run `sosh` within `sosh` within `sosh` within `sosh`, etc. One fun thing to do is to run 7 or 8 `sosh`'s in a chain and then try killing one of the ones in the middle of the chain - this causes the entire first half of the chain to die in a domino fashion.

For more about ELF loading, see **System Calls**.

Process deletion

Processes can be deleted for a number of reasons. Perhaps `process_delete` was called, perhaps an assertion failed within a process, perhaps a process faulted on an invalid region (like a `NULL` pointer), perhaps a process simply reached the end of its main function and had nothing left to do.

Whatever the reason, if a process needs to be deleted, we first mark it as a *zombie* - which is a process that should be dead but is still alive. Then, every 100ms[11] we check whether there are any processes who are marked as zombies. If we find a zombie process and notice that it isn't in the middle of a system call, then we delete it (destroy the thread, destroy the address space). Otherwise we do nothing and in another 100ms time the zombie process will be checked again.

For this to work, we have to do some extra housekeeping at the start and end of every syscall - namely, marking a process as being in the middle of a syscall or not. This is handled by the `pagetable_setAddressSpaceIsInKernel()` function. One reason for not wanting to delete a process that is in the middle of a syscall is that, if the syscall was using a callback (e.g. `nfs_read`), then it would be unfortunate if the user read buffer were to be deleted before the callback was fired.

When a zombie process is finally deleted, a call is made to `syscalls_PROCESS_WAIT_checkQueue()` which wakes up any relevant waiting processes.

Locks

We created a *Lock* ADT that enables a lock to be acquired and released, the code lives in `sos/lock.c`.

Locks are used in a few places:

- Our console uses a lock to protect access to its read and write buffers, but we aren't sure whether this is really necessary (see **Virtual File System**)
- `sos/libsos.c` uses a lock when creating tasks with `sos_task_new()`; since the tasks are created at root server initialisation time, we allow the root server to block here, waiting for each address space to be created. It might seem odd that creating an address space could block; this is because an address space requires frames to store the upper and lower pagetables, and acquiring a new frame could require the eviction of an existing frame to the swapfile, which requires continuations (see: **Pages**).
    - In practice an eviction at initialisation time should never happen (unless we were operating with a terribly small amount of frames), and `sos_task_new` is only ever called for one, tiny program anyway (see: **User Processes**). But just to be safe we use continuations and a lock.
- `sos/swapfile.c` uses a lock when opening the swapfile - this happens at root server initialisation, and we allow the root server to block here, waiting for the swap file to be created by the NFS library.

Test ELF Applications

We provide a number of test applications in the `userapps` folder. Notable among these is `userapps/hotfuzz/hotfuzz.c` which performs some rather pretty unit testing on `libs/sos/src/sos.c`.

We also attempted to move a number of applications out of `userapps/sosh/sosh.c` and into their own ELF files. However, we ran into problems with applications that required command line arguments, as there is currently no way to send an application command line arguments in our operating system. This meant that some "applications" were forced to stay as hardcoded functions within `sosh`.

Other notes

We should have been more consistent with the use of `physicalAddress_t` and `virtualAddress_t` - sometimes `L4_Word_t` is used instead. These were late additions.

We never tested what happens if a system call is given an invalid pointer that is not a NULL pointer. For example a pointer that doesn't live in a valid virtual region.

Similarly, we never tested what happens if a system call is given a valid pointer to data that lives in the swapfile, but the frametable can't find a frame to evict to make room for it. For example, if the frametable is full and all its frames are marked as unevictable.

For both of these cases, the pagefault simulation for pinning the buffer should kill the process (as with a normal pagefault), but the root server may continue to process the system call regardless of whether the pinning was successful. The behaviour would be undefined, and might actually kill the root server.

There may be some stray tab characters stuck in the source code that will break the indentation (we know of at least one). We tried to use only spaces for indentation (3 spaces per tab.)

When creating a process, we check that the ELF file is valid, but that is just a simple check for some magic characters at the start of the file. If we are given an invalid ELF file (but one that nevertheless contains the right magic characters), then we will attempt to load the file.

In particular, we never tested what happens if an exception occurs (like an invalid instruction being processed). The process should probably be killed, but at the moment an exception will actually cause the entire root server to die after dumping the frametable metadata to the screen.

For a file mode, sometimes we use `L4_Word_t` instead of using the proper type `fmode_t` specified in `libs/sos/include/globals.h`.

Throughout development we had various problems with `L4_DefaultMemory` (which is cached). We opted to run under `L4_UncachedMemory` the vast majority of the time. We ended up switching back to using the cached `L4_DefaultMemory` in the end, but perhaps with a superfluous number of `L4_CacheFlushAll()` calls scattered throughout our code. If we could reduce the number of cache flushes then our root server would probably operate faster. 🙂

To turn debugging comments on, edit `sos/rosa.h` - change `VERBOSE` to some large positive number like 100.

Footnotes

The cover image of the neon mudflap girl was taken from http://blogs.epi.es/lostiemposcambian/files/2006/11/mud-flap-girl.jpg

1. See `sos/frametable.h: #define FRAME_SIZE 4096` (1)
2. See: `sos/pagetable.h: #define PAGE_SIZE FRAME_SIZE` (2)
3. For more about libsos functions, see `libs/sos/src/sos.c` (3)
4. For more about `quickCall()` and `quickReply()`, see: `libs/sos/include/globals.h` (4)

5. See `sos/pagetable.h: #define MAX_ADDRESS_SPACES 16` (5)
6. See: `sos/devices/console.c` (6)
7. See: *more libserial problems and a possible solution*, https://cgi.cse.unsw.edu.au/~forums/support/viewtopic.php?t=10998 (7)
8. See: `sos/devices/filesystem.c: #define FILESYSTEM_MAX_BUFFER_SIZE 1100` (8)
9. See: `sos/pagetable.h: #define MAX_ADDRESS_SPACES 16` (9)
10. For more about the `soshloader`, see: `userapps/soshloader` (10)
11. Zombies are checked alongside `nfs_timeout` requests, see `sos/syscalls/nfs_timeout.c` (11)