SOS Documentation

Group 8

6th August 2021

1 Execution Model

1.1 Structure of SOS system

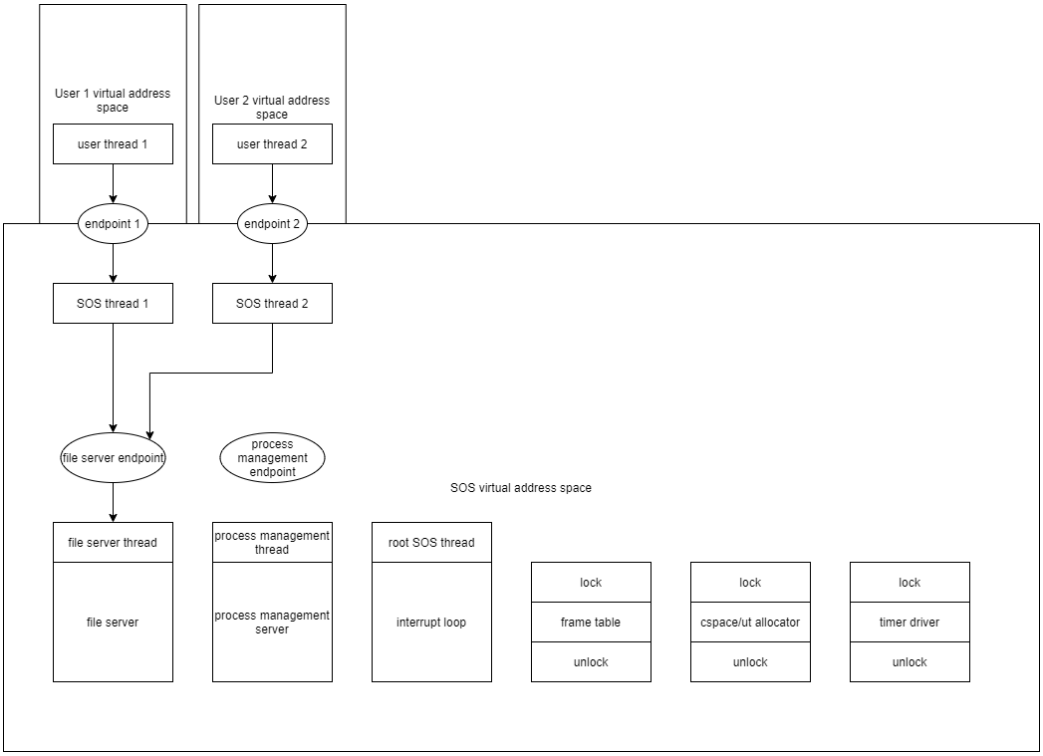

Figure 1: SOS system overview

We use a threaded model of execution with most of the services of SOS encapsulated in servers which look like event loops. There are three main threads of execution which are spawned off from the main SOS:root thread when the system is initialized. The main SOS:root primarily deals with the interrupt_loop which waits on a notification object created specifically for dealing with the interrupts and once it receives a signal on that notification object it handles the IRQ corresponding to it and then starts waiting again for any potential interrupts. The two other main threads of control/SOS threads that exists in the system are the **File server thread** and the **Process management server**, both of them are created inside main.c.

1. The file server thread when initialized starts waiting on its server loop on the specialized endpoint named file_server_ep for the request from the user to deal with any of the file management/updation of the file

interfaces already in place, it creates a new reply object to reply back to the specific user/SOS thread which has called in and then stores in the correct reply object to reply back to the user inside the call context and goes back in to the server loop by creating a new reply object if it is not dealing with any interrupts/IRQs.

2. Another major thread that we have in the system is the process management thread which is used for the global management of the user processes in the system and is again an abstraction of the SOS service broken into a separate thread of execution. The process management thread again waits on an ipc endpoint on which the processes can call in and it again creates a new reply object to reply back to a specific process and saves the reply object correctly inside correct index of the process management struct for the particular process of which the process id is the index into the process management struct where its reply object is stored in. Whenever a new process is exec'd a request into the process management loop is made which spawns of a new kernel thread corresponding to the new user thread that is going to be created which has its own separate private endpoint to communicate back into the system and its associated SOS thread. Both the newly spawned off kernel thread/SOS thread and the user process associated with it are stored inside teh process management structure at the correct index which is new processes PID.

1.2 Locks for the shared objects:

We have two types of locks in our system, and the lock objects corresponding to which are declared in the special file used for managing the synchronization in the system named as "**sync.h**" and the lock objects are created using the alloc_retype and the ut_allocator library provided by seL4 inside main.c, we use seL4_wait and seL4_Signal function calls on the lock objects to provide the intrinsic mechanism for the synchronization on the lock objects and the two lock objects in the system are namely corresponding to:

1. Global Cspace and the ut allocator, named as **SOS_CU_LOCK()** which does nothing but waits on the cspace_ut_lock object implemented as seL4_Wait(cspace_ut_lock, NULL) and has a corresponding **SOS_CU_UNLOCK()** which signals the cspace_ut_lock as seL4_Signal(cspace_ut_lock).

2. Lock for the Frame table accessed globally by all the processes and is created as **SOS_FT_LOCK()** and waits on frame_table_lock as **SOS_FT_LOCK()**

seL4_Wait(frame_table_lock, NULL) and has a corresponding **SOS_FT_UNLOCK()** which signals the frame_table_lock as seL4_Signal(frame_table_lock).

All other synchronization requirements in the system are dealt with implicitly by breaking and delegating the individual responsibility of to separate servers in the system wherein each processes calls are dealt with on separate endpoint objects which makes the system implicitly synchronized and each of the server also provides an synchronized access to all of its management data structures Eg. process manage server provides synchronized access to process management data structures.

2 System call dispatching

2.1 General System call layout

The System call interface to the userspace and user processes are provided using the intrinsic IPC facility provided by seL4 wherein we make the use of the specialized function calls **seL4_Call**, **seL4_Recv** and **seL4_Send** to instantiate the communication between the userspace and the services provided by our **Simple Operating System(SOS)**.

2.2 IPC Protocol

The general IPC protocol that our system maintians to initate a system call from the usersapce is to pass on the payload(i.e the function arguments, userspace pointers) using the Message registers/IPC buffer facility provided by **seL4**. And the common behavior that all system call handling code matches is to pass in the **SYSCALL NUMBER** in the seL4 message register(MR 0) and then the rest of the arguments are passed in from the message registers **1..N**. All userspace pointers are passed in as seL4Word pointers and are copied in the kernel using the frame iterator(talked about in the section below) in a secure way. And the general convention followed by all the System calls is to receive back any errors in a consistent manner using the Message register 0 wherein **SUCCESS** represents that the users request for the particular System call has been dealt with consistency and the user is responded back with the results that are desired using the value reeved back from the message registers 1 onwards depending on the system call semantics, and **FAILURE** represents that system call has not been dealt with due to an error either in the system or due to the arguments passed by the user, of which the user is notified with correctly by passing back a desired value with

regards to te system call error. All the user level library calls(libsosapi) are redirected back correctly into the specific user level system call(which uses the **seL4** IPC mechanism to communicate back into SOS) using the correct syscall number and the mapping back into the correct syscall using the number is done by the function sosapi_init_syscall_table inside vsyscall.c.

2.3 System call numbers:

The system call number table for our Syscalls is as follows

SYSCALL NUMBER	SYSCALL NAME
__NR_openat	sos_sys_open
__NR_close	sos_sys_close
__NR_read	sos_sys_read
__NR_write	sos_sys_write
SOS_SERIAL_CONSOLE_OUT	sos_write
__NR_getdents64	sos_getdirent
__NR_fstat	sos_stat
__NR_execve	sos_process_create
__NR_kill	sos_process_delete
__NR_getpid	sos_my_id
__NR_io_getevents	sos_process_status
__NR_wait4	sos_process_wait
__NR_nanosleep	sos_sys_usleep
__NR_getitimer	sos_sys_time_stamp

2.4 SOS sycalls

We have the following syscalls corresponding to each of the user operations that the sos thread corresponding to that user needs to handle:

void sos_serial_console_out(seL4_CPtr reply, struct serial * serial, process_data_t * process);
void sos_nanosleep(seL4_CPtr reply);
void sos_getitimer(seL4_CPtr reply);
void sos_openat(seL4_CPtr reply, process_data_t * process);
void sos_close(seL4_CPtr reply, process_data_t * process);
void sos_read(seL4_CPtr reply, process_data_t * process);
void sos_write(seL4_CPtr reply, process_data_t * process);
void sos_brk(seL4_CPtr reply, seL4_Word heap_bottom, seL4_Word

*** heap_top);**
void sos_stat(seL4_CPtr reply, process_data_t * process);
void sos_getdents(seL4_CPtr reply, process_data_t* process);
void sos_getpid(seL4_CPtr reply);
void sos_exec(seL4_CPtr reply, process_data_t * process);
void sos_waitpid(seL4_CPtr reply);
void sos_process_status(seL4_CPtr reply, process_data_t * process);
void sos_kill(seL4_CPtr reply);

2.5 Concurrent syscall management

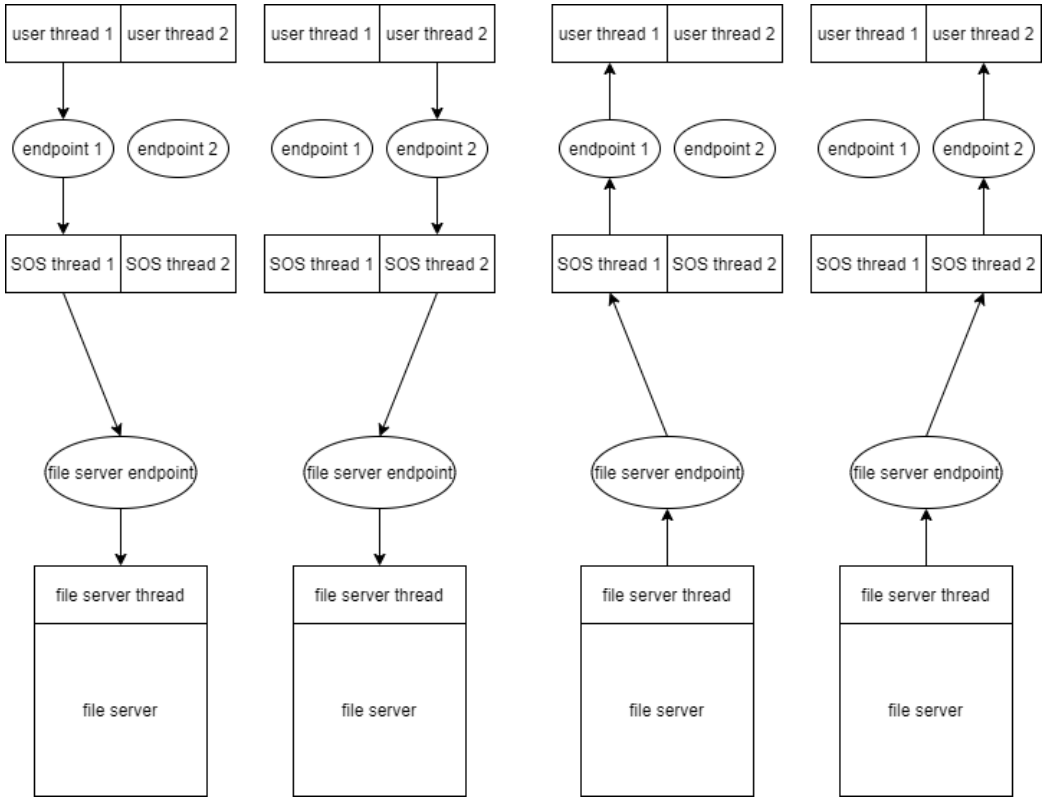

Figure 2: File server handling concurrent IO

The system is also capable of managing concurrent system calls as following from the execution model, there is a kernel thread corresponding to each user thread/process and there is a separate endpoint on which the communication corresponding to the user thread and the sos thread is being initiated and this is what we use to manage the concurrent system call requests and to make

sure that the system is free of any race conditions of any sort we add locks corresponding to any global/shared objects between each of the threads.
So, following off from the diagram added above whenever a user threads make in a request for any file management specific details the system call corresponding to that goes to its corresponding SOS thread which calls into the file server that is waiting inside the server loop on its specific endpoint, when ti receives a request it saves the reply object corresponding to that process and then goes back into the loop once again waiting for new requests to come by creating a new reply object specific that process and in this way we deal with the concurrent requests on the file server as all the nfs calls later as asynchronous which does'nt cause any races.

3 Device Drivers

3.1 Timer Device

The two main purposes of the timer device is track the time elapsed since booting and to provide the ability for timeout callbacks to be registered.

Our timer stores timeouts as a linked list structure sorted in ascending order of when each timeout will occur. We choose this data structure as the cost of removing timeouts from the queue in the IRQ handler is minimized. The nodes in the linked list are malloc'd from the heap, but instead of freeing the nodes in the IRQ handler, we add them to a free list to be freed (or reused) in subsequent calls to register new timeouts. This is again done in order to minimize the work done in the IRQ handler.

Since there are multiple system threads that may simultaneously access the timer device, we use a notification object in order to provide exclusive access to critical sections of the device code. Since the callback functions in timeouts may call into the timer device themselves (to register another callback for example) our implementation will safely release the lock to the timer device before executing callbacks, and re-acquire the lock afterwards.

When invoked, our IRQ handler will call as many callbacks as it can and then program the timer device to create another interrupt when the next timeout occurs. Our code will choose the best timer frequency depending on how far in the future the earliest timeout occurs, so it is a tickless timer to the extent that is permitted by its limited size. The countdown timer is also reprogrammed when new timeouts are registered and when a callback is

removed.

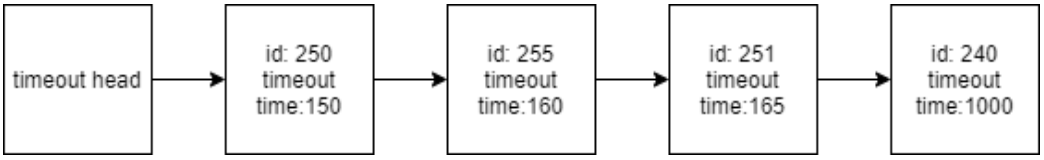

Figure 3: ID allocation example

Our scheme for allocating ids is relatively simple. Ids are allocated in ascending order, and we have a strategy for wraparound when the maximum assignable id is given. For example if the size of the id variable is 1 byte and the last id assigned to a timeout was 255, then our timer will find the minimum assigned id by iterating through the list of currently registered callbacks and then start assigning ids in descending order from the id before the minimum assigned id. In the situation depicted above, the minimum assigned id is 240, so the next timeout registered would get an id of 239, and allocation would continue downward.

A similar thing happens when the value reaches 0 again, the maximum assigned value is found and id assignment continues upward from there. Given the size of the id field in timeouts is 32 bits, this 'turnaround' operation will almost never need to be invoked in normal usage, and id assignment will occur in amortized constant time. There are some cases where this will fail, but such cases require extremely large timeout values and registering callbacks at a high frequency for a long period of time, situations unlikely to occur during normal usage of SOS.

4 I/O Subsystem

The **Simple Operating System(SOS)**, also provides an abstraction of the file systems and the corresponding read, write, open, close operations on it to the userspace using the UNIX semantics wherein the corresponding calls on an arbitrarily provided path goes back to the sos system call management loop, which in turn creates a generalized file interface for the particular being opened and store a pointer to it in the the file allocation table for its processes struct and then the specific file interface function call makes a call to the server loop with setting the correct call_context makes a call to the file server loop waiting on the specific endpoint for the user input to appear which in turn spawns of the request to the correct asynchronous nfs call using

the libnfs interface which makes a call to the correct callback corresponding to each of the calls that the user makes in.

The SOS in general supports the serial and the console operation semantics using the libserial interface and the generalized files are initialized and managed using the Network file system(**NFS**) which are both dealt with asynchronously and corresponding to each request we destroy the reply object after replying back correctly inside the corresponding callback for the libnfs asynchronous call.

Currently we make an assumption that there can only be 1024 files open for a particular process at any time, which is represented by FD_MAX inside process.h for the process file descriptor table represented by file_interface fd_table[FD_MAX] inside the process_data struct.

4.1 Generalized File Interface

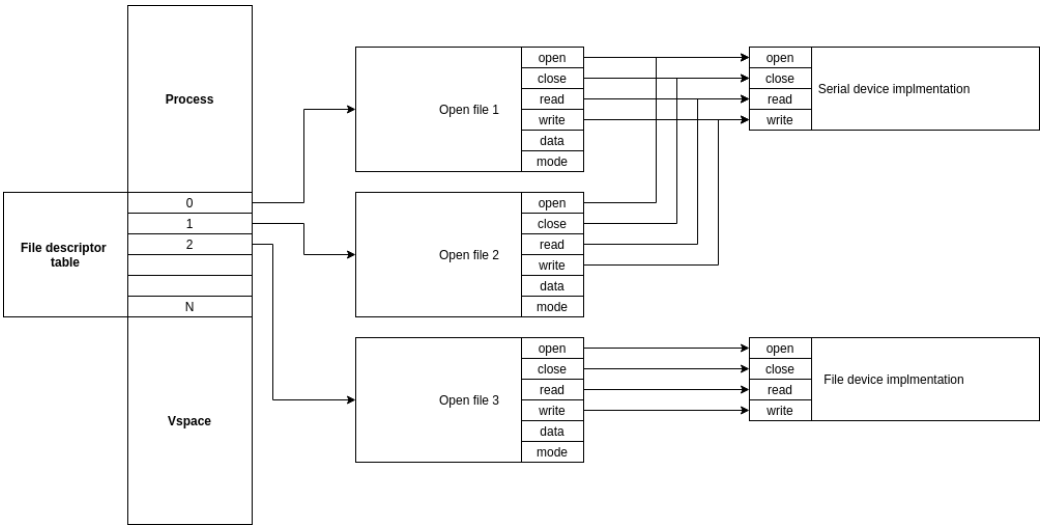

Figure 4: File interface overview

We use this generalized file interface inside **SOS** for any of the files that are created,

{

int (*open)(struct fi_inner * interface, const char * filename, sos_thread_t * thread);

int (*read)(struct fi_inner * interface, char * buffer, int len, bool * stop);
int (*write)(struct fi_inner * interface, const char * buffer, int len, bool * stop);
int (*close)(struct fi_inner * interface);
int mode;
void * data;
}

We store a reference to each of the file interfaces whenever a file is opened inside the processes vspace(as shown in the figure above) using a file descriptor table,where in the index value inside the processes vspace represents the file descriptor allocated to the that particular file.

For the purpose of initializing an entry of any particular type of device we compare the path name given to the SOS for creating a file and if it is "console' then we call our specialized serial interface initialization function called as make_serial_device(interface) which initializes a new serial interface over which we can call its interface specific open, read, write and close and other file operation semantics and if the path for the file being created is anything other than "console" then we make a call to the make_file_device(interface) to initialize the file interface. To handle the read and write permissions for the file when we are creating a file interface itself we store the 'mode' with which the file was created in the file interface struct corresponding to that file itself and whenever we are reading and writing we make sure that the the readable files are not been written and not changed up.

4.2 File descriptor allocation

For the purposes of the file descriptor allocation we use a bitmap structure inside util.c representing the which of the indexes inside the fd_table for the process are currently storing the reference to an open file. We find the first free bit inside our bitmap represented by bitfield_fd[BITFIELD_MAX], where BITFIELD_MAX is (1024/8) as we need each entry inside it will represent 8 bits for the 1024 files that can potentially be open at any time. We make an assumption that the first 3 file descriptors will always be allocated for STDIN, STOUT AND STDERR and any other file being opened will only be given a file descriptor from 3 onwards. The bitmap structure helps us to find the first free bit/ first free id that can be allocated any process quite quickly in comparison to looping through the entire array of the File interfaces and

therefore considering this aspect itself we have tried to incorporate a bitmap for each process to help us with the file descriptor allocation. Whenever we close the file the bit value corresponding to its file descriptor is reset to 0.

4.3 Serial Interface

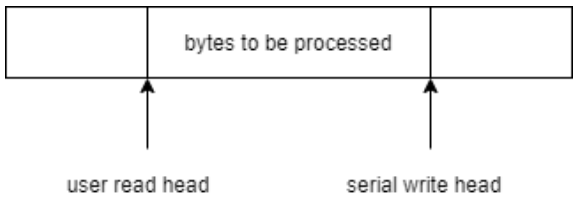

Figure 5: Console circular buffer

The serial console is a many-writer, single-reader device. Therefore when a process opens the console device for reading, it will block if there is already another device reading the console. Processes opening the console just for writing will be able to open it without blocking. While no one has the console opened the console for reading, any bytes coming in through the console handler are dropped.

When the console is opened for reading, the serial handler will place received bytes into the circular buffer. When a user tries to read a number of bytes from the console, they are read out of this buffer. In this way, a user will be able to read all the bytes typed in the console after they opened it for reading. If there are no newlines in the buffer, or if the the number of bytes in the buffer is smaller than requested by the user, the SOS thread processing the user's request will block until more input is received from the console. The circular buffer makes no attempt to prevent the write head moving past the read head. If the user is unable to process input to the console fast enough, some bytes will be dropped.

4.4 File server

For the purpose of handling the file management system calls we separately create file server thread initially when the system is initialized which is waiting on a separate endpoint inside a server loop waiting for a request from

the user. When we initialize the system we also a create a separate notification object called load_ntfn which is used to notify the process management thread which waits on the load_ntfn until the libnfs is mounted and is signalled back when inside network.c the nfs_mount_async signals on the same notification object as soon as it is mounted in. The file server thread is always waiting on a separate endpoint for any user requests which are given to it from the syscall management specific SOS thread or by directly calling into the file interface specific functions which directly makes a call into the server loop. The specific system call primarily makes a call to the specific file handling function defined in file.h which sets up the call context for the server loop and when the server loop acquires a request on its endpoint then then based on the type set in the call_context it makes in a call to the specific nfs library asynchronous call with the correct parameters and after it correctly deals with the correctly dealing with file management with regards to the call made inside the libnfs directory then it makes in a call to the callback functions corresponding to the asynchronous calls which checks in the errors received if any by the libnfs function and then reply back the reply using the reply object passed in as the call context and then the reply object is freed as the sever loop creates a new reply object corresponding to each call by saving the previous reply object each time inside the call context structure unless it is handling a notification. The callback specific information which is created is sent back to the fie interface specific functions which receives back the data from the callback and then set the data of the file interface passed in with the data returned from the specific callback and this how we persist the information for the file interface which is passed in, and to persist the information for the libnfs file which is opened in we use the structure open_file_t which stores a pointer to the nfsfh which is stored in the structure whenever a new nfs file is created and its cache position is updated with writes to get a better write performance for the overall system.

5 Virtual Memory

5.1 Page Table Structure

Our page page is structured very similarly to the hardware page table structures on the ARMv8-A architecture. There are 4 levels of tables, each containing 512 entries. Similar to the ARMv8-A architecture, 48 bit virtual addresses are broken up into series of bits and then used to index into the page table structures. The last 12 bits are used as the offset into the physical frame backing the virtual memory.

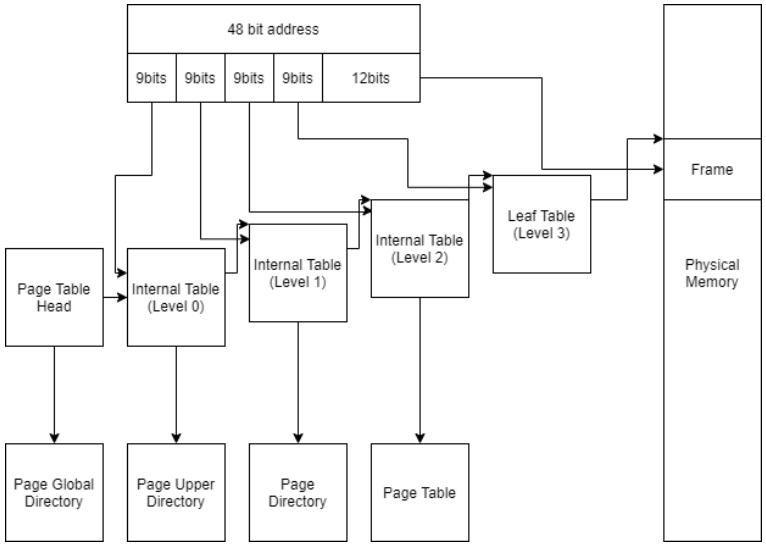

Figure 6: Page table structure

Each entry in the page table is also responsible for keeping track of the seL4 memory management structures. As shown in the diagram, the head of the page table maintains references to the Page Global Directory and each entry in level 0 maintains a reference to a Page Upper Directory etc.

Entries in internal and leaf table all have this structure

```
{
    seL4_CPtr cap : 39;
    frame_ref_t frame : 19;
    bool valid : 1;
    bool init_pte: 1;
    bool readonly: 1;
    bool never_execute: 1;
    bool data_in_elf: 1;
    unsigned char ut_bit : 1;
}
```

Some fields serve slightly different roles in internal and leaf entries

In internal entries, the 'frame' field is a reference to the memory backing the table that this entry points to. In leaf entries, the 'frame' field is a reference to the memory backing the virtual memory in consideration.

In both types of entries, the valid field indicates whether there is memory currently backing the entry. For internal entries this value will always be true after memory has been allocated to back the entry as page table frames don't get evicted in our implementation. For external entries however valid entries may become invalid when the physical frames they reference are allocated to other entries.

The init_pte field is used to ensure that read/write/execute permissions for frames can only be set when they are initially loaded.

The readonly and never_execute fields are only for leaf entries and specify permissions for a particular frame.

The data_in_elf field impacts how the 'cap' field is interpreted in leaf entries. If data_in_elf is true, this indicates that the contents of the frame backing this entry should be identical to a corresponding location in the elf file.

The cap field serves a variety of purposes. In internal entries, it is a seL4_CPtr to a seL4-managed memory structure. In external entries, its interpretation is dependent on whether data_in_elf is true. If data_in_elf is false and cap is non-zero, then it is an index in the pagefile to the memory backing the current entry. If data_in_elf is true, then the cap field is an offset into the elf file for this program from which the data should be loaded in.

5.2 Modifications to the Frame Table Structure

We add the following fields to the frame entry table structure:

```
{
    bool no_evict : 1;
    bool used : 1;
    bool dirty : 1;
    bool unused : 1;
    seL4_ARM_Page user_page_cap : 36;
    frame_ref_t frame_and_entry_index : 28;
}
```

The no_evict field indicates whether a frame can be evicted by the page replacement algorithm. Frames backing internal page table entries and ipc buffers are always not evictable. A frame on which IO operations are being done is also marked as not evictable to avoid race conditions. It should be noted that since IO is done frame-by-frame, this will not lead to the situation

where all frames could get marked as not evictable when processing a large buffer.

The used and dirty bits are used to implement demand paging.

The no_evict, used, dirty and user_page_cap fields could be stored inside the page table entries, but we reason that it is preferable to increase the size of frame table entries which increase in number linearly with the size of physical memory being managed, rather than in page table entries which increase with the number of processes and the amount of memory that the process has touched. Also, in this way, we only have two maintain two capabilities to physical frames managed by the frame table. One for SOS, and another for the user process which currently has access to the frame.

The 'frame_and_entry_index' field is a reference to the page table entry that currently owns the frame. It is used by the page replacement algorithm to invalidate page table entries when the frames backing the memory they reference are paged out and allocated to someone else. As depicted above, this field is 28 bits long. The first 19 bits is a reference to a frame in the frame table, and the last 9 bits index to the entry of that frame. It would be possible to simply store the memory address to the page table entry, but this scheme saves some space in frame table entries.

5.3 Data Transfer between User Processes and SOS

The SOS read and write syscalls accept the virtual address of the user buffer and the number of bytes to process. The request is then processed frame by frame. SOS consults the page table to find the SOS virtual address corresponding to the frame and marks the frame as no evict. The request for IO is then forwarded on to the SOS file server using the SOS virtual address for the frame. When the file server has completed its operation the frame is again marked as evictable. This process enforces the virtual memory layout outlined below, so buffers referencing invalid memory result in the syscall failing and the IO not being completed. Read and write permissions are also enforced as the permissions for mapped memory are checked before allowing IO.

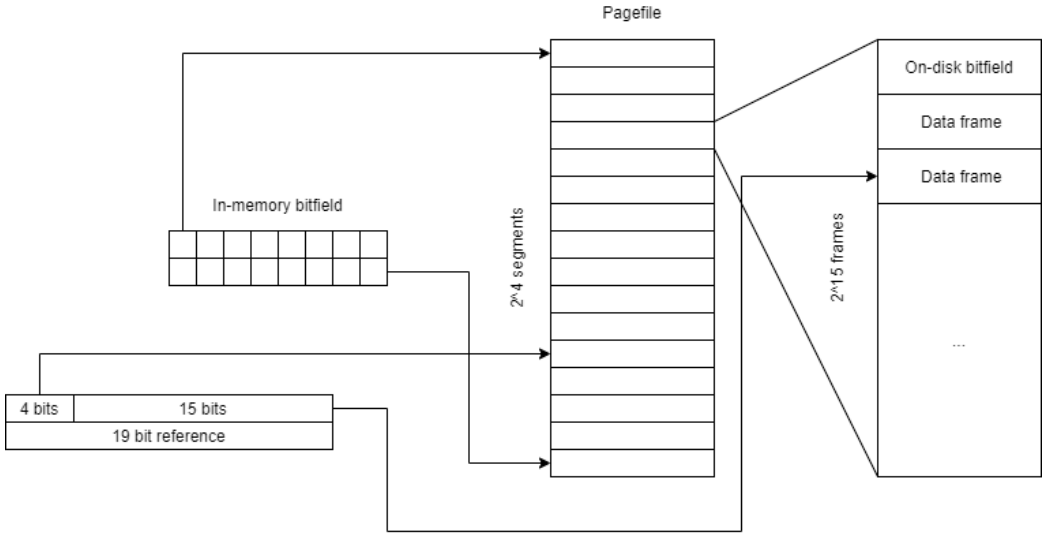

Figure 7: Page file layout

5.4 Demand Paging

Our pagefile management system is capable managing pagefiles up to sizes of just over 2GB, as indexes into our pagefiles 19 bit references to 4096-byte pages. The first 4 bits of these indexes are used to index into a 2 byte bitfield which is stored in memory. This in-memory bitfield tracks whether each of the 16 segments that we break up our pagefile into are full or not. The Last 15 bits are used to index into the first page of each segment, which itself is a bitfield used to track whether each page in the segment is in use or not. So we have a two-level free list, where the first level is stored in memory and the second level is stored on disk.

We implement the second-chance clock replacement algorithm as required by the project specification. From our research, the ARMv8-A architecture does not provide hardware page used bits so we check whether a page has been used between separate rounds of the replacement algorithm by un-mapping the page and marking it and waiting for a VM fault on the page before un-marking it. We also implement checking pages for dirtiness in a similar way: all pages are initially mapped in as readonly, and are marked as dirty if we get a write fault on them.

We try to limit writes back to the pagefile as much as possible. In particular, we never write back pages which are just a copy of what is in the elf file, and we never write back pages which are clean and already have an entry in the

pagefile.

5.5 Memory Layout and VM Fault Handling

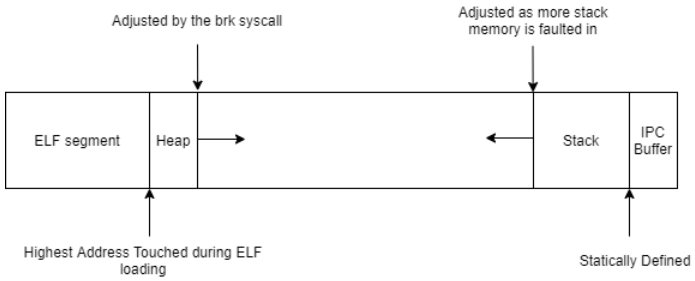

Figure 8: Process memory layout

Our process memory layout is depicted above. It is quite simple and makes a couple of assumptions. Our memory layout has four segments.

The ELF segment contains all the code and data loaded from the ELF file. The ELF loader keeps track of the highest address used by the ELF file, and after loading is complete, all memory from the bottom of the virtual address space to this highest address is considered to be the ELF segment. After loading is complete, any attempts to create new mappings in this segment will be considered illegal and will result in the process being killed.

The bottom of the heap is the top of the ELF segment. The top of the heap is dynamically changed by the brk syscall. Attempts to change to top of the heap so that it intersects with other segments will result in a failed syscall.

The top of the stack is hardcoded to be at a particular address and it grows down toward the heap as usual. The IPC buffer is fixed in size and sits above the stack.

Attempts to map in memory above the top of the heap and more than a couple of pages below the bottom of the stack will be considered illegal and result in the process being killed.

An assumption made by our memory layout is that the largest address touched during ELF loading is sufficiently below the hardcoded top of the stack so as to allow space for the stack and heap.

6 Process Management

Our SOS implementation supports up to 512 processes. These processes are managed by a static array. We synchronize access to this static array by using a process management server (PM server). When SOS threads desire to manipulate the state of SOS processes, they perform an IPC call to the process management server's endpoint.

6.1 Process ID Allocation

Process ids are simply the index of a given process in the process management static array. Note that the process management thread and the file server thread don't have entries in this array, so syscalls invokes by users cannot wait or kill these system processes.

6.2 Process Creation

The thread_create syscall calls into the PM server in order to create a new thread. When creating a new thread, the PM server first allocates resources for and starts up a new SOS thread. This is done similarly to the starter code, with some additions to cleanup failed attempts at starting a new thread. The newly created thread then begins executing and allocates resources for a user thread which executes the desired app. When thread_create is called, the resources for an old SOS thread may be reused as described below. Newly created SOS threads get added to the first free entry in the process management array and uses its index in the array as its process ID.

6.3 Blocking and Waiting for Access to Devices

Sleeping and waiting for device access is also handled by the PM server. For example, a process desiring to sleep will make an IPC call to the PM server, specifying how long it wishes to sleep for.

Within the process management loop, a process desiring to sleep will be suspended and a timeout will be registered that will call back into PM server at the appropriate time to wake the thread back up.

A thread requesting access to a device will pass the id of the device which it wants to access (currently there is only a console device however). If the device is already in use, the process is put into a queue and suspended.

Queues to access devices and sleeping is handled inside the PM server so that killing a process can be easily implemented, as discussed in the next section.

6.4 Killing a Process

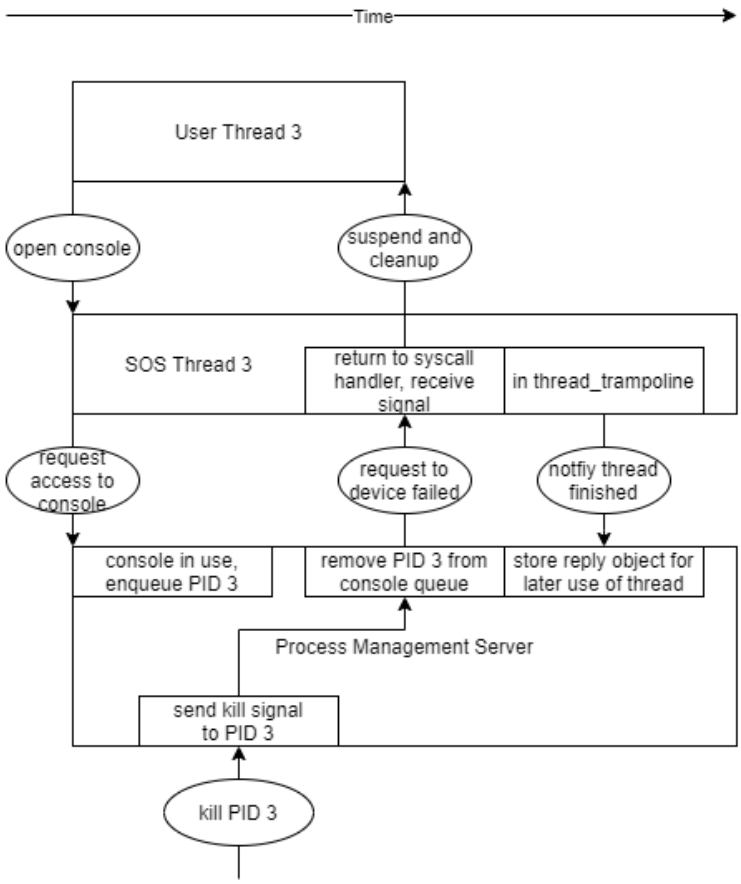

Figure 9: Process kill example

Each SOS 'syscall handler' thread, that is, the threads paired with user-level processes, have a notification object bound to their TCB. The PM server is given a minted capability to this notification object which it used to signal to a thread that it is being killed. The advantage of this design is that it is

relatively easy for the SOS syscall handler thread to cleanup appropriate user level thread resources itself before calling into the PM server to have itself cleaned up. The only problem with this approach is that SOS threads which are blocked for an arbitrary amount of time (sleeping, waiting for access to a device) will not immediately respond to the kill signal, as the signal will only be received when seL4_Recv is called in the SOS thread's syscall handler loop. This blocking for device access and sleeping are both handled inside the process management loop. After signalling the process being killed, the PM server will check to see whether the process being killed is sleeping or waiting for a device. If it is, then the process will be prematurely woken up and will respond to the kill signal when it again calls seL4_Recv after responding to the syscall that it was processing. The running time of other syscalls (even ones that block, such as IO calls) are assumed to be sufficiently short so that no special action needs to be taken. The process will complete whatever system call it is performing, call seL4_Recv, and receive the signal that it is being killed.

When a kill occurs, a user process's syscall handling SOS thread cleans up all the resources associated with the user thread, but the PM server doesn't actually clean the resources associated with the SOS thread. We've modified the thread_trampoline code in threads.c so that when a SOS thread returns to the trampoline, it calls back into the PM server. When this happens, the PM server marks the thread as free and stores the reply object to the thread. When a user calls thread_create, this old SOS thread will be reused if it exited sufficiently long ago (currently 5 seconds) to avoid causing race conditions. To reuse the thread, the PM server simply responds to the saved reply object the address of the buffer containing the name of the app to be started. If this thread is killed it will again return to thread_trampoline and call into the PM server, indicating it is ready for reuse.

The primary advantage of this approach is that an explicit bitfield keeping track of free virtual addresses for new SOS stacks and IPC buffers doesn't need to be maintained, and SOS doesn't have to keep reclaiming and allocating the same resources. It is true that if session of SOS usage uses x number of threads then resources for all those x threads will remain allocated until the system is rebooted, but it is also true that if x threads were used in the past, that many could be used again.