SOS Design Documentation

1. Introduction

SOS (Simple Operating System), is a simple multitasking operating system designed to run on top of seL4 microkernel with MCS configuration on AArch64 processors. Every process is single-threaded and has their own address spaces. The design goal of this project is about simplicity, which is trivial to study, modify, and extend, while still being functional at the same time.

1.1 Supported Platforms

SOS is designed and tested only for AArch64 processors with 4 KiB (4096 bytes) paging size and little-endian configuration. Switching the target platform can be done by setting the `PLATFORM` CMake variable. The following platforms are officially supported:

- Hardkernel ODROID-C2 (`odroidc2`)
  Supported features: timer, serial output, networking
- QEMU (`qemu-system-aarch64`)

  Supported features: serial input and output

Other AArch64 platforms that are supported by seL4 should be supported by SOS, as long as they use PL011 for UART.

2. Execution Model

SOS multitasks by using the threading model, where all running user applications will have an internal SOS thread that will work on behalf of them during I/O operations. Simplicity is the main reason for the decision of using the threading model because it is easier to reason with. All threads in SOS, including user applications, run in the same priority. Therefore the thread scheduling is completely round-robin based.

2.1 Main Event Loop

Located in `src/main.c`, the SOS event loop is a single-threaded loop whose job is to wait on the main IPC endpoint, and either reply to the IPC request made to this endpoint or dispatch it to a different thread and continue waiting for the IPC endpoint for new IPC requests. The main event loop can distinguish the IPC requests by the means of labels and/or the first word in the IPC request. Sources of the IPC requests can also be distinguished by the use of the badging/minting mechanism provided by the seL4 microkernel. Various things that will be handled by the main event loop will be described later in this document.

The main event loop will have a pool of reply objects that will be used when receiving IPC requests. In accordance to the MCS spec, these reply objects will be reused after finishing handling the message. In the event where there are more outstanding IPC requests than the reply pool, the main event loop will dynamically allocate more reply objects from *cspace* and enlarge the pool.

3. SOS System Calls

Upon a user application startup, SOS will handle out an IPC endpoint capability to the user application, which then the user application can use to perform *SOS system call* (not to be confused with the actual seL4 system calls), that is privileged operations such as I/O, memory allocation, or management of other processes. This IPC endpoint capability is *minted* and badged from the main IPC endpoint, which means that the SOS system calls will be handled by the main event loop.

A user application can perform SOS system call by sending IPC requests in a predefined format to the IPC endpoint handed out upon their startups, and either wait or ignore the result (for example, by using `seL4_Call` or `seL4_Send`).

3.1 libsosapi

This library is provided in the source tree of SOS, which user applications can link to. This library provides several functions used to abstract away SOS system calls in a neat function call-like model. Additionally, it provides several constants such as file mode bits and capabilities for IPC endpoints. The function and constant definitions are provided in the `include/sos.h` in the `libsosapi` source tree.

Further, `libsosapi` can be linked together with musl libc. By calling `sosapi_init_syscall_table` upon user application startup, `libsosapi` will make certain functions in musl, such as `printf`, call the functions provided in `libsosapi` instead.

3.2 System Call Dispatching

All I/O SOS system calls (`OPEN`, `CLOSE`, `READ`, `WRITE`, `STAT`, `OPENDIR`, and `DIRREAD`) will be dispatched to the file management subsystem on behalf of the SOS thread that corresponds to the requesting user application.

Other SOS system calls, such as process management and memory or address space management (such as, but not limited to, `BRK`, `MMAP`, `PROC_NEW`, and so on), will be dispatched to their respective subsystems but will be run on the main thread. This is done to avoid race conditions and to simplify the logic.

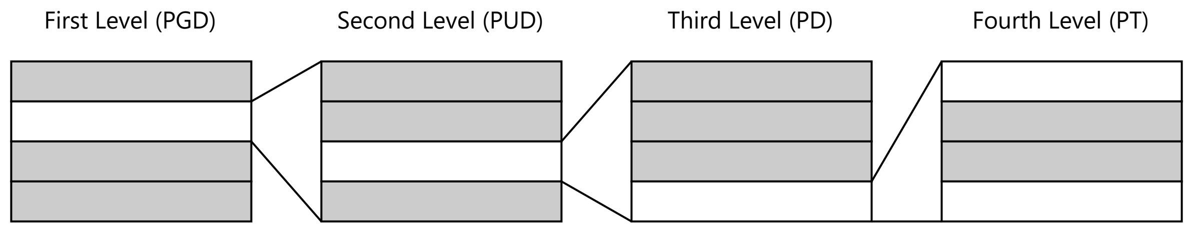
SOS system call for the timer (such as `USLEEP`) will be dispatched to the timer driver.

4. Memory Subsystem

SOS utilizes the four-level page tables, the standard for AArch64 in 4 KiB page size configuration. Each page table can store 512 entries. SOS assumes that the virtual address space is of 48 bits, which is the standard of ARMv8.0-A ISA. All threads in SOS share the same virtual address space. However, each process in SOS has their own virtual address spaces. The layout of the virtual address space for both SOS itself and user applications can be seen in `src/vmem_layout.h`.

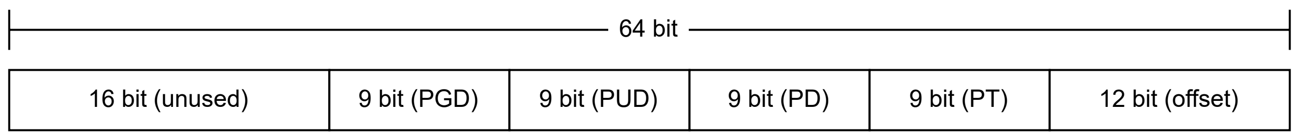
For every virtual address space managed by SOS, SOS keeps track of a *shadow page table*, which is a four-level page table similar in structure to the hardware/seL4's page table, but stores information regarding the underlying capabilities and untyped for every actual seL4's page table, in addition to the frame references to the next level depth of the page table.

4.1 The Four-Level Page Table



The four-level page table structure has the structure where each page table is of size 4 KiB and has exactly 512 eight-byte entries. Every page table, except for the last (fourth) level, stores the location of the next level of the page table. The last page table will store the location about the frame that is get mapped to the virtual address space. In order of the most significant bit, the order of the name of each level are:

1. Page Global Directory (PGD)
2. Page Upper Directory (PUD)
3. Page Directory (PD)
4. Page Table (PT)



Every time a virtual address is to be resolved, the virtual address will be decomposed into six parts, as can be seen in the diagram above. The first 16 bit part is unused as ARMv8.0-A only supports 48-bit address space. Then each of the 9-bit parts will be used to index to the respective page table. Finally, the last 12 bit part is used as the offset of the resolved frame.

4.2 Morecore

Internally, SOS needs to allocate some memory for a certain data structures, such as passing parameters between threads. To cope with this requirement, we use the musl's memory allocator algorithm (via `malloc` and `free`), and source the memory from a statically allocated region known as the Morecore. This implementation, as well as the declaration of the static variable that backs the underlying memory, is in the `sys/morecore.c`. The size of the morecore variable is limited to 1 MiB.

During the testing phase, we found some concurrency issues with both morecore and musl allocator. Since our design is threaded, and the memory allocator is basically used everywhere throughout various parts of SOS, we decided to move the musl allocator to a separate thread. This is done by renaming the `malloc`, `free`, `realloc`, and `__malloc0` symbols in the musl's source code, and creating a weak alias to those renamed symbols, so that other user applications linked to the musl in the SOS's source tree can still use those functions.

However, since those symbols in musl are now weak symbols, SOS can reimplement those four functions and the symbols defined in SOS will take precedence in the SOS binary. The new replacement functions will call those musl memory allocator functions internally, but does so in a delegate fashion.

4.3 Address Space Region Abstraction Model

Every virtual address space is abstracted in a region data structure. Each of these regions represents a valid virtual memory address range for that particular virtual address space, as well as their properties (attribute and permission). The data structures associated with the region model can be seen in `src/addrspace.h`.

4.3.1 Region

Each region is represented as the following structure:

```
struct addrspace {
    uintptr_t begin;
    uintptr_t end;
    struct addrspace_attr attr;
    seL4_CapRights_t perm;
};
```

Both the `begin` and `end` fields are divisible by the page size (4 KiB). They represent the range of the virtual address encompassed by the region in the form of `[begin, end)`. For example, if `begin` is of value `0x1000` and end is of value `0x2000`, then `0x1000` and `0x1FFF` would be a valid address for that region, whereas `0x2000` would not.

The `perm` field maps directly to the seL4's permission word used in `seL4_ARM_Page_Map`. Additionally, the attribute contains the information regarding the nature of the region (for example, whether it is a regular region or mmap-ed region).

4.3.2 Region Storage

Each of the regions is stored in a dynamic array and is maintained in sorted order, which means that a region's `end` field is always smaller than or equal to the next region's `begin` field. The whole dynamic array is in turn a representative of the possible virtual address for the associated virtual address space.

Maintaining regions in a sorted order means that searching for a particular address can be done quickly in $O(\log_2 n)$ by using binary search. The downside is that if the list is modified, we have to shift the regions next to the affected region, which would be a $O(n)$ operation in the worst case.

4.4 User Application's Memory

As mentioned before, each user application has a separate virtual address space. Initially, when the user application is first started, no frames are allocated for the user application except the IPC buffer, a single page of the stack, and the regions that contain code, global, and static data specified by the user application's ELF image. The user application's IPC buffer page is always pinned to memory for the lifetime of the user application.

When user application faults on a memory, SOS will check if the faulting address corresponds to one of the user application's region. If it is not the case, the offending user application will be terminated. Otherwise, SOS will either maps a new zero-initialized frame for the user or *page in* (read from page file) if the user's frame has already been allocated but paged. Should these operations fail (for example, because of I/O error or not enough memory left), the offending user application will be terminated as well.

In addition to the regions specified by user application's ELF executable image file, SOS also provides additional region:

- **Heap**
  Initially, user application has zero pages of heap. User can request to increase or decrease heap size by using the `BRK` syscall.
- **Stack**
  The initial stack region of the user application is set to 20 MiB in size (5120 pages). This initial limit is configurable, both statically in `src/vmem_layout.h` through `PROCESS_STACK_MIN_PAGES`, or

dynamically during application's runtime by calling the `GROW_STACK` SOS system call.

- **mmap**
  User applications can additionally request a new address space region separate from the heap by calling the `MMAP` SOS system call (and calling `MUNMAP` on the allocated region to remove it). This feature has similar semantics to the `mmap` system call with `MAP_ANONYMOUS` flag in POSIX systems.

Allocation of all those 3 additional regions will fail if the resulting region would clash into another region in the requesting user application's virtual address space. The virtual address space addresses in which those 3 regions will be allocated can be seen in `src/vmem_layout.h`.

Shrinking the heap or stack or unmapping the mmapped regions will free the backing any logical frame allocated within their respective region.

4.5 Frame Table Subsystem

The frame table subsystem is the repository of logical frames. This subsystem is implemented in `src/frame_table.*`. Other subsystems of SOS will ask the frame table subsystem for logical frames. Logical frames can be pinned to prevent their memory frames to be paged out. A logical frame may have three possibilities:

- Does not have any data.
- Data resides in memory.
- Data resides in the page file or backing file.

For the users of the frame table subsystem, a logical frame is represented as a 21-bit integer. This integer directly indexes to the internal structure of the frame table subsystem. Because of this 21-bit integer indexing, the maximum supported total combined memory size (RAM plus paging file) of SOS is 8 GiB ($2^{21} \times 4096$ bytes). The decision of choosing the 21-bit integer size is mostly limited by the shadow page table structure to keep it's size exactly one page. This value can be configured by setting the `FRAME_TABLE_BITS` in `src/grp01.h`. Do note that increasing this value beyond 21 will fire static asserts, and will require data structure change to the affected objects.

4.5.1 Logical Frame

Each logical frame by itself is a doubly linked list, stored in an array (the `frames` array). The aforementioned 21-bit logical frame reference indexes this array. If more logical frames are needed, an untyped will be allocated and mapped right at the end of the `frames` array in SOS's virtual address space to extend that array. This mapping bypasses the standard memory mapping subsystem, and the allocated untyped will be held for the lifetime of SOS.

A logical frame can be either in the `free` list or the `allocated` list. Allocating a new logical frame is a simple constant time operation by popping the front of the `free` list and putting it in the back of the `allocated` list, and the reverse applies for freeing a logical frame.

4.5.1.1 Structure

Each logical frame is a packed linked list node requiring only 2 words (16 bytes) for each node. The following table is the description of the fields:

Name	Size (bits)	Description
`back_idx`	21	If `backed` is true, index to either page file if `paged` is true or memory frames otherwise.
`prev`	21	Logical frame reference to the previous item in the linked list where this logical frame resides.
`next`	21	Logical frame reference to the next item in the linked list where this logical frame resides.
`list_id`	2	Indicates the linked list where this frame resides.
`pinned`	1	True if the logical frame is pinned, false otherwise.
`paged`	1	Meaningful only if `backed` is true. True if the logical frame is backed by a file or the page file, false otherwise.
`backed`	1	true if the frame has underlying backing storage, false otherwise.
`reqempty`	1	Applicable only if `backed` is false. If set to true, the next time a new memory frame is allocated for this physical frame, that memory frame will be zero-filled.
`usage`	1	Used as a counter for the second chance algorithm.
`file_backed`	1	True if this logical frame is backed by a file and is read only.
`file_pos`	1	Applicable only if `file_backed` is true. The 4 KiB offset of this logical frame to the backing file.
`file_backer`	40	If `file_backed` is true, refers to the pointer to the file handle.

4.5.2 Memory Frames

110011001			CPtr
CPtr	CPtr	CPtr	CPtr
CPtr	CPtr	CPtr	CPtr

Memory frames are bookkept in a series of memory pages. Each page has the format of a bitmap header indicating which slot of the frame capabilities page is unused (not allocated to any of the logical frames), and the array of the logical frame itself. A single page can store 992 seL4 frame capabilities. The number 992 is chosen because it is the maximum amount of data that can be stored in a 4 KiB page in such format. In a similar fashion to the `frame` array for the logical frame storage, these pages are allocated and mapped directly from untyped.

If a logical frame needs a new backing frame, a linear search operation will be performed on those frame pages. We could have used a linked list style here, however doing so would high memory overhead, and scanning for a bitmap is a pretty quick operation given that the `clz` instruction is used, allowing the use of a single instruction to scan for 64 entries.

4.5.3 Page File

Whenever an operation to retrieve the pointer to the data or the frame page capability upon a logical frame is performed (typically done by the memory mapping subsystem), a memory frame page capability will be allocated for that logical frame. In case of a successful allocation, the memory frame's usage bit will be set. On the other hand, if the allocation failed (for instance, because a memory frame limit is reached), the frame table subsystem will instead start looking for a frame to page out to the paging file.

4.5.3.1 Page Replacement Policy

If the memory frames are full and more memory frames are needed, the frame table subsystem will traverse logical frames in the `allocated` list, and look for a logical frame that has a backing memory page storage and is not pinned. In accordance with the second chance page replacement policy, if such a frame is found, the `usage` bit will be check. If it is set to 1, it will be set to 0, otherwise, the underlying memory storage of the logical frame will be written to the page file and the backing frame will be freed.

The page replacement policy will look for the entire `allocated` list of the logical frame storage twice. If it cannot found any logical frame that matches the above criteria, or the page file is full, a failure status will be reported to the caller, otherwise the index of the page file will be returned.

4.5.3.2 Page File Sructure

The page file is structured as a heap of 4 KiB pages: the file consists purely of the memory frames that have been evicted, with no other metadata. The page file can be directly indexed from the `back_idx` field of the logical frame structure.

The information about the free space in the page file is stored as a bitmap in memory. As in the memory frame storage, this storage is allocated one 4 KiB page at a time from the untyped and mapped in the SOS' virtual address space. Unlike the memory frame storage however, this bitmap consists purely of bitmaps, that is a 4 KiB page of the bitmap can be used to represent 32768 pages in the page file.

During the page replacement policy execution, when a frame is about to be evicted, an empty slot in the bitmap will be consulted with the `clz` approach, which means that the first empty slot in the page file will be selected for writing the memory page content. In the same fashion as the memory frame storage, when the bitmap is full, a new memory page from untyped will be appended at the end of the bitmap array. Failure will be reported to the caller if the allocation of the page from untyped failed or I/O failure when writing to the page file.

4.6 Memory Mapping Subsystem

The task of this subsystem is to map a frame from the frame table to a virtual address space. This subsystem is closely tied to the shadow page table structure. This subsystem will blindly map a frame to the given virtual address space and the virtual address, without regard to whether the target address is valid (for example, falls within a region) or not: it is the responsibility of the caller of the mapping subsystem to check for the virtual address validity.

4.6.1 The Mapping Process

When this subsystem is being tasked for mapping a frame to a virtual address space, the following steps will be performed:

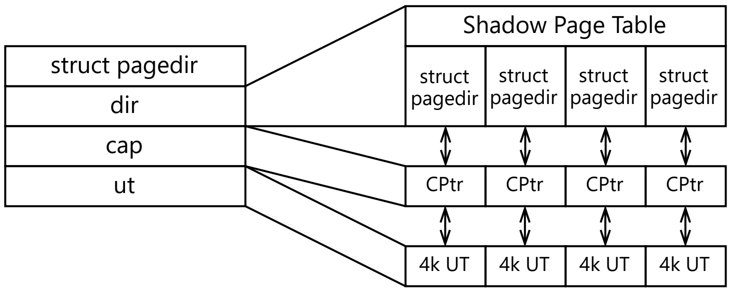
- Allocate the shadow page tables, as well as the associated capabilities frame and the untyped frame for all levels for the given virtual address.
- Traverse the shadow page table and see if there is already a frame on the shadow page table. If it is

the case, and the given frame reference to be mapped is not `NULL_FRAME`, report as failed to the caller. Else, get the physical frame of the frame reference and perform a remap.
The last scenario (remap) could happen in the case of a fault and the underlying frame resides in the page file instead of memory.
- Get the memory frame capability from the logical frame, make a copy of the capability, and map it to seL4 to the given virtual address space for the virtual address. In the process of doing this, the intermediary page tables will be mapped to seL4 for that virtual address space if required.

If any of the above processes fails, the partial allocation will be cleaned up, and failure will be reported to the caller.

4.7 Shadow Page Table Structure

The shadow page table structure follows the [four-level page table] structure, where each page table has 512 entries, and there are 4 levels of the page table. That is, every page table, except for the last level (PT) stores the location of the next level of the page table and the last level (PT) stores the information about the frame that is get mapped to the virtual address space.



Each entry of the shadow page table (shown in the diagram above), including the fourth level, has the following structure:

```
__attribute__((__packed__)) struct pagedir {
    frame_ref_t dir : FRAME_TABLE_BITS;
    frame_ref_t cap : FRAME_TABLE_BITS;
    frame_ref_t ut : FRAME_TABLE_BITS;
};
```

Where the `FRAME_TABLE_BITS` is equal to 21. Therefore, each of those entries is 8 byte in size. Such structure allows us to create a shadow page table structure that truly mimics the actual AArch64 page tables, where each page table is of one page in size (4 KiB), and each page table stores exactly 512 entries. However, two more frames are needed for each page table entry to store both the capability and the untyped.

For the first, second, and third level of the page table, each of those fields has these meanings:

- `dir`
  Contains a frame reference to the next shadow page table level.
- `cap`
  Contains a frame reference to a frame that stores the capabilities to the seL4 page table associated with the next level page table in `dir`. These page tables are the one that gets mapped to seL4.
- `ut`
  Contains a frame reference to a frame that stores 4 KiB untyped that back the capabilities of the mapped page table.

Whereas for the last level of the page table (PT), each of those fields has these meanings:

- `dir`
  Contains a frame reference to a frame that stores the frame reference to the frame table.
- `cap`
  Contains a frame reference to a frame that stores the copied capabilities of the frame reference from the frame table specified in `dir`. These capabilities represent the frames that get mapped to the virtual address space.
- `ut`
  Unused, because the mapped frame capabilities is backed by the frame table.

The shadow page table is stored in a regular frame from the frame table, therefore it is subject to be paged out, just like other user application's frames. However, the seL4 page tables are directly allocated from untypeds. Therefore, they will be remain mapped and allocated for the lifetime of the virtual address space.

5. Input Output (I/O) Subsystem

The primary method of doing I/O in SOS is by using files. A file can be either a stream (such as `console`), a block (such as other regular files, or the fake in-memory file), or a directory.

5.1 The File Abstraction

A file is like a class, where a new file object can be instantiated by using the `open` or `opendir` function, and be destroyed by using the `close` function. The *file handler* refers to such class. A *file handle* is the combination of an identification returned by the `open` or `opendir` function, and the file handler itself: the same file identification number might refer to a different file when paired with a different file file handler. A file might be a *singleton*, where multiple calls to `open` of its file handler will result in the same file handle.

A directory file will be obtained upon a successful operation to the `open` function. The following operations can be done for a stream or a block file:

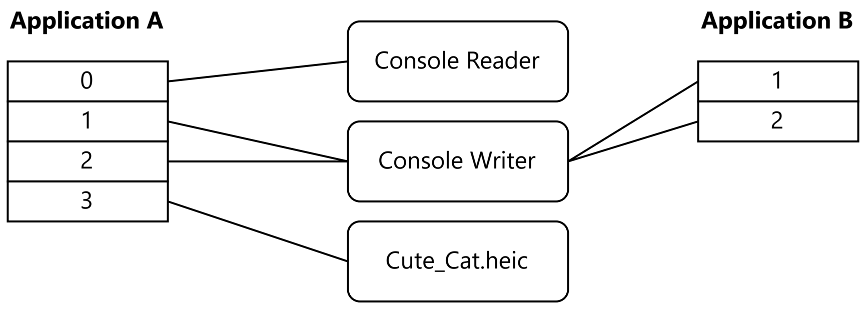
- **Read**
  Read a portion of the file to the SOS virtual address. The caller has to specify the size of the data to be read. For a block device, the position of where to start reading must be specified as well.
- **Write**
  Write from SOS virtual address to the file. The caller has to specify the size of the data to be written For a block device, the position of where to begin writing must be specified as well.
- **Stat**
  Get some information about the file, specifically size, permission, access time, and modification time.

A directory file will be obtained upon a successful operation to the `opendir` function. The following operations can be done for a directory file:

- **Get directory entry**
  Get the file name inside the directory according to the given index.

For any subsystem in SOS, opening a file is a two-step process. First, they have to find the correct file handler associated with the file name. This facility is provided as a `find_handler` function in the `src/fileman.h`. A file handle can be opened afterward after a file handler has been obtained.

5.2 File SOS System Call



If a user application wants to perform an I/O operation, it has to do so via SOS system calls. There is an SOS system call for each of the possible operations that can be performed upon a file. User applications cannot see the actual file handle directly: they will have a file identifier unique to themselves. Internally in SOS, an actual file handle will be resolved based on the combination of the user application's badge and the user application's file handle.

Since I/O operations are expected to take a long time and potentially blocking, any I/O SOS system calls made by user applications, including open and close, will be run in an SOS thread dedicated to that user application.

5.2.1 Memory Management

Since user applications have different virtual address space compared to SOS, I/O operations cannot be directly performed upon it. Therefore, SOS will first map user's page specified by the pointer and length passed to the SOS system call to SOS's own virtual address space, in a dedicated "scratch" area range (specified in `src/vmem_layout.h`. The rule for mapping user application's page is different, depending on whether it is an in operation (SOS read from user application's pages) or out operation (SOS write to user application's pages).

- **In operation**
  SOS will first map all user pages matching the arguments passed to the SOS system call. If part of the user application's virtual address is not valid (for example, outside of any user application's region), or no logical frame has been allocated yet, `EFAULT` will be sent back to the requestor, and no partial I/O operation will be performed. The reasoning of doing the such is that if the requesting user application does not have a logical frame there, the memory should have not been initialized anyway.
  In operation is typically used when a user application does the `WRITE` SOS system call or as a means for SOS to read a passed in user application pointers such as file names for `OPEN` and `STAT` SOS system calls.
- **Out operation**
  SOS will map one user application page at a time doing the I/O operation. If an I/O operation has reached the end of the specified page, the current user application's page will be unmapped, and the next page will be mapped. If there is no logical frame allocated, SOS will allocate a logical frame to the requestor's virtual address space associated with that current virtual address as well. This decision was made because the user might want to initialize their memory using data from a file instead. However, if during this process SOS encounters an invalid virtual address (for example, outside of any user application's region), I/O operation will be stopped and `EFAULT` will be sent back to the requestor. The partial I/O will have been performed by then.
  Out operation is typically used when SOS needs to handle out data to user application's provided pointers, such as the `READ` and `STAT` SOS system calls.

5.3 Available File Handlers

SOS supports various kind of files. This section mentions all of the supported file handlers available in SOS.

5.3.1 UDP Serial

This file handler exposes a singleton stream file that can be read or written into. This file handler requires networking support, as the input and output of this stream file will be sent to the UDP console. This is the default file handler for the `console` file if SOS is compiled for the `odroidc2` platform. This file handler sends and receive data directly by utilizing picoTCP library.

Multiple writers can be opened from this file handler, however only a single reader can be opened instead: attempts to open more readers will fail unless the first reader is closed.

5.3.2 PL011 Serial

This file handler is the same as the above UDP Serial file handler, with the exception that the input and output will be done via the PrimeCell PL011 UART I/O. This is the default file handler for the `console` file if SOS is compiled for any platforms other than the `odroidc2` platform.

5.3.3 Fake File

If enabled, this file handler exposes a special block file named `fake`. This file behaves like any standard block device, where the user can read and write from any starting position. Set the CMake `SosUseFakePageFile` to other than 0 to enable this file handler. When enabled, this file handler will get its backing store directly from untyped, for the size specified in the `SosFakePageFileSize`. The underlying data is mapped in SOS's virtual address space in the `SOS_FAKE_FS` virtual address (specified in `src/vmem_layout.h`).

This file handler can be used to test the page file implementation.

5.3.4 NFS

This file handler is the default file handler for any other file names not specified explicitly by the other handlers. This file handler provides support for I/O operations on the NFS server specified in `SosNFSDir` and `SosGateway` CMake configuration and exposes the files in the NFS server as block files.

The majority of the NFS logic is handled by the LIBNFS library. The file handle identification returned by this file handler will be a pointer to the internal LIBNFS handler library. As this file handler is meant to be run on a separate thread, this file handler will block upon reading or writing. However, as LIBNFS only provides asynchronous callback interfaces, we will pass along an seL4 notification object to the callback function and wait upon it. Once the callback finishes executing, it will signal this notification and the file handler operation will continue. This decision was made because in doing so prevents us from allocating a separate state object. Instead, as the originating function was waiting, its stack would be still valid, therefore the parameter for the callback can be stored in the stack.

For compatibility with the supplied `sosh` code from Data61, this file handler will create the file upon opening, even if the `O_CREAT` flag was not specified in the `mode` argument.

5.3.5 CPIO

This file handler is the default file handler for any other file names not specified explicitly by the other handlers if the CMake configuration `SosUseLocalFS` is set to other than 0. This file handler exposes all files in the boot image's CPIO archive as read-only block files.

6. Networking Subsystem

The network subsystem only supports the meson hardware, therefore it is only enabled when SOS is configured for the `odroidc2` platform. It utilizes the meson ethernet and the meson watchdog timer to perform its task. To enable truly concurrent access, even when the main thread is blocked, all the networking subsystem, including LIBNFS and picoTCP, is run on a separate SOS thread. The interrupts from both the meson ethernet and meson watchdog timer are handled on this thread as well.

Upon startup and network initialization, the following steps are performed on the main thread:

- Setup seL4 to redirect IRQs for both the meson ethernet and the meson watchdog timer to the notification. This notification will be eventually bound to the endpoint that belongs to the network thread, and also badged so that the network thread can identify if the endpoint it is waiting for was sourced from this notification.
- Initialize meson ethernet.
- Initialize picoTCP.
- Request DHCP and wait until get a DHCP reply.
- Initialize meson watchdog with an interval of 0.5 milliseconds.
- Initialize LIBNFS and mount the server's NFS directory specified in the CMake configuration.
- Bind the notification from the first step to the network thread.

In the similar fashion as the main event loop, the network thread will wait upon its endpoint capabilities. Since the IRQ notification was bound to the network thread, any interrupt that happens from either the meson ethernet or the meson watchdog will wake up the network thread.

6.1 Delegation

To avoid a race condition, especially in LIBNFS, all `async` requests from other SOS subsystems will be delegated to the network thread as well. As a result, the networking subsystem exposes functions for LIBNFS `async` operations that will be used by the file abstraction. Invoking any of these functions will send an IPC request to the network thread's endpoint, thus waking it up and forward the actual `async` operation to LIBNFS. These functions are defined in `src/network.h`.

LIBNFS's `async` operation will eventually finish or fail and in turn, will invoke the supplied callback. Since the LIBNFS callback is invoked by the `nfs_service`, which in turn invoked as a result of an IRQ request, the callback will be invoked by the network thread as well.

7. Demand Paging

SOS implements demand paging, which means that memory frames are allocated on demand upon a virtual memory fault. This demand paging approach also make it possible to implement paging file to extend the available memory, and also file backed memory.

7.1 Virtual Memory Fault Handler

The core of the demand paging system is the fault handler. This handler handles all virtual memory faults on all threads managed by seL4, except the main thread itself, as this fault handler runs on the main thread. Faults are handled slightly differently depending on the source, whether it is originating from user applications or SOS's internal thread.

To make distinguishing between user's fault and SOS' internal thread fault possible, the main event loop's endpoint is badged appropriately, and this badged endpoint is then attached to the respective thread by using the `seL4_TCB_SetSchedParams` facility of seL4.

7.1.1 User Application Fault

Upon a user application's virtual memory fault, the fault handler will first obtain some information about the fault itself, including whether it is a read or write fault, and the fault address. Next, the fault handler will proceed to find the virtual address space region for the offending user application. The permission of the region is then checked, and if it does not match with the access type, the offending user application will be terminated. The offending user application will be terminated as well if no matching region was found for the faulting address.

The fault handler will invoke the memory mapping subsystem to map logical frames to the user application's virtual address space. First, the fault handler will try to remap an existing logical frame, in case a logical frame was mapped to the virtual address space before but got evicted because of page out. If there was no frame allocated, the fault handler will then allocate a new logical frame from the frame table, and map it to the user's memory.

If the remapping operation fails (for example, because of I/O error) or a new frame could not be allocated (either because of I/O error or memory full), the offending user application will be terminated.

7.1.2 Internal SOS Thread Fault

Since all of SOS's memory resides either in the stack (which is always preallocated), static section, global section, or Morecore, demand paging is generally not needed here. However, sometimes SOS needs a file-backed memory, which is used extensively in the ELF loader for loading the user application's image. Therefore, upon a virtual memory fault originating from the internal SOS thread, the fault handler will only attempt to remap the virtual memory of the offending virtual address. If this remapping process failed for any reason (for instance, I/O error, unmapped logical frame in the faulting address space, or memory full), SOS will either panic or call the continuation point set by the offending thread.

A thread can set a continuation point beforehand by using the `setjmp` function from the musl libc and storing the buffer in the designated area in the thread's TLS, so that in the event of a virtual memory fault, SOS will resume the thread to that continuation point and set the value to 1. In case of an unrecoverable virtual memory fault, and the SOS thread has set a continuation beforehand, the fault handler will perform the following operations:

- Allocate an emergency stack with a size of 128 bytes from the offending thread's stack base.
- Set the stack pointer of the offending thread to the newly allocated stack, and set the program counter to the `thread_wrap` function. This is performed by using the `seL4_TCB_WriteRegisters` feature of seL4.
- Resume the execution of the offending thread.

Upon resumption, the offending thread will execute the `thread_wrap` function under the offending thread's context. The `thread_wrap` function will then do a `longjmp` to the stored `setjmp` buffer in the TLS before and set the return value to 1.

8. Process Management

A *process* in an instance of a running user application. SOS supports the management of multiple single-threaded processes running at the same time. SOS also supports the full lifecycle of a process, from its creation to deletion.

8.1 The Process Control Block

The state of each process managed by SOS will be stored in a static array known as the *process table*. Each element in this array is also known as the *process control block* (PCB). Looking up the correct process control block is a constant time operation: the process table is indexed directly from the process ID. to obtain the process control block. The size of the process table is controlled by the `SosMaxPID` CMake configuration, which in turn dictates the maximum amount of running process in SOS. The default configuration of this variable is 128, which means that 127 user processes can run at the same time (the zeroth entry of the process table is reserved for the SOS itself).

8.1.1 PCB Structure

The main process control block structure has the following fields:

Name	Description
`active`	If true, indicates if there is a running process associated with this TCB.
`state_flag`	Can contain any combination of the following flags: `PROC_STATE_CONSTRUCTING` or `PROC_STATE_PENDING_KILL`. These flags are related to the process destruction.
`tcb_ut`	Contains the untyped that backs the TCB capability of the process.
`tcb`	Contains the TCB capability associated with the thread of the process.
`vspace_ut`	Contains the untyped that backs the PGD (virtual address space) capability of the process.
`vspace`	Contains the PGD (virtual address space) capability of the process.
`ipc_buffer_frame`	Logical frame reference to the IPC buffer for the thread associated with this process.
`ipc_buffer_mapped_cap`	A copied capability of the memory page capability of the `ipc_buffer_frame` that get's mapped to this process' virtual address space.
`sched_context_ut`	Contains the untyped that backs the scheduling context capability of this process' `tcb`.
`sched_context`	Contains the scheduling context capability of this process' `tcb`.
`fault_ep`	Contains the endpoint capability for this process' `tcb`. This endpoint capability is minted and badged from the main event loop end point, so that the main event loop and the fault handler can determine the originating process of the fault.
`cspace`	Contains a data structure of the one-level cspace of the process.
`as`	Contains list of valid regions for this process' virtual address space.
`bgthrd`	A reference to the internal SOS thread that handles I/O operations on behalf of this process.

Name	Description
`file_size`	Stores the ELF image file size of the executable in bytes.
`command`	Stores a partial name (first 31 characters) of the process' image file.
`start_msec`	The time when the process started relative to the uptime of SOS in microseconds.
`waitee_reply`	If this process is doing `WAITPID`, this stores the reply object capability so that SOS can inform this process when the target process is destroyed.
`wait_target`	Stores the ID of the process that this process is waiting for.
`waitee_list`	Stores the list of processes that are waiting on this process. This field is implemented as a bitfield of the size of `SosMaxPID` for compact representation.
`loader_state`	Stores information about the ELF loader when constructing this process.

Additionally, the I/O subsystem also stores the file table of this process, which in turn stores all file handles opened by this process. Similar to the main process table, the file table is indexed directly by the process ID, and individual file handles are indexed directly by the unique file handle number of the process. The mapping subsystem also keeps track of the shadow page table for each SOS process. Finally, the timer subsystem keeps track of the reply objects of every process that is still "actively" doing the USLEEP SOS system call.

All the fields contained in the TCB structure combined with the bookkeeping of the I/O subsystem, the mapping subsystem, and the timer subsystem will have enough information to keep track all resources used by the process, which in turn can be used as the information for the process destruction without leaking any resources.

8.2 Process Creation

Process creation will be initialized by SOS during startup to run the first process. Afterward, any process can create another process. Process creation requires a compatible ELF executable image to reside in the file. The procedure of a process starting another process is as follows:

- The main event loop will invoke the process manager, which will in turn find an unused process control block (`active` is false) in the process table. If there are multiple possible process control block to put the new process in, the selection will be done in accordance to the process ID allocation policy.
- The process manager will then proceed to bootstrap the process control block, in which the virtual address space, TCB, IPC buffer, scheduling context, SOS thread for the process, and the file table is created. To make SOS simple and prevent the race condition, these creation is still performed on the main thread. Those operations are also relatively quick.
- If all of the above processes succeeded, the target TCB will have its `PROC_STATE_CONSTRUCTING` flag set and the SOS thread associated with the *creating* process will be invoked to load the ELF file from the file system.
- The ELF loader, now running in the SOS thread of the creating process, sets up a scratch virtual address space with the size of the target ELF file according to the `stat` of the file, and proceed to set up file-backed logical frames and map those logical frames to the scratch space.
- ELF loader will then set up a continuation point, and then invoke the libelf with the base ELF address

pointing at the base address of the scratch address space initialized in the previous step. The ELF file will be lazy-loaded as the libelf faults while parsing the given ELF file from the memory. If the libelf's fault failed, such as because of a corrupted ELF file or not enough memory, the fault handler will revert this execution to the continuation point, and failure will be reported to the creating process.
- After the ELF loading process succeeded, the aforementioned scratch space will be unmapped, and the PCB flag will be checked for the `PROC_STATE_PENDING_KILL`. If no such flag exists, the `PROC_STATE_CONSTRUCTING` flag is then unset, and the process' thread will be started. Otherwise, the newly constructed process will be killed.

8.2.1 Process ID Allocation Policy

Recall that the process ID of SOS indexes directly to the SOS's process table. Therefore, each process must have a *PID* (process ID). If there are multiple empty PCBs to put the newly created process in, a circular indexing scheme will be employed. The process manager will keep track of a variable which contains the last allocated process ID. When a new process ID is requested, this variable will be incremented, and the TCB associated with that variable will be checked whether it is in use or not. If it is not in use, then the content of the variable will be used as the new process ID. Otherwise, the variable will be incremented again and the process is repeated.

8.3 Process Destruction

The process destruction is performed by closing all the file handles, unmapping all the mapped frames associated with the user's virtual address space, unregistering the timer callback and reusing the free object associated with the `USLEEP` SOS system call, and freeing all the resources stored in the PCB. Process destruction can be initialized by another process (by using the `KILL` SOS system call), by an unhandled exception such as when a process fault outside its designated address space region, or by a process killing itself (such as exit).

When the process destruction is initialized by another process via the SOS system call, the SOS system call will return immediately. To check whether the destruction process has finished, the process can use the `WAITPID` SOS system call to wait for the target process. In case the process was already destroyed by the time the killer process call `WAITPID`, then the target process ID will be very likely (but not impossible) become invalid due to the process ID allocation policy.

However, a process might be destroyed while they are being constructed in the ELF loader phase or performing I/O, therefore a special consideration needs to be taken in such cases, as destroying the target process too early will make the I/O subsystem of SOS references to an invalid memory area, which will crash SOS itself.

Process destruction will always be performed in the main event loop to avoid race conditions.

8.3.1 Destruction on Being-Constructed Processes

The following procedure will happen if a process is getting destroyed while being constructed:

- The process destruction procedure will see that the `PROC_STATE_CONSTRUCTING` flag of the target PCB is set. Process destruction will then set the `PROC_STATE_PENDING_KILL` on the flag and abort the operation.
- The ELF loader will eventually finish. As mentioned in the process creation step, the ELF loader will then check the PCB flag for the `PROC_STATE_PENDING_KILL`. If the flag is there, the process destruction procedure will be invoked again on that process.

8.3.2 Destruction on Processes With Active I/O Operation

When a process is having an active I/O, especially read and write operation, then the file handler, which is running in SOS' address space, will be still doing the I/O operation and copying the result to or reading the data from the mapped user process. Therefore, destroying this process right away will disturb the operation and ultimately crashes the SOS itself. Therefore, a special procedure is needed to handle this case. The following condition will happen if a process is doing the I/O SOS system call:

- Upon carrying on the I/O SOS system call, before handling over the I/O operation to the background thread and invoking the file handler there, the I/O subsystem will store to its internal data that the process is actively doing I/O operation.
- After the I/O operation is finished, the I/O subsystem will then set the active I/O flag of the process to off.

When the process destruction procedure is invoked, it will first try to destroy the file table associated with the to-be destroyed process. Since the file table is owned by the I/O subsystem, the procedure has to invoke the file table destruction procedure of the I/O subsystem, which in turn will consult the I/O active flag of the process. If the I/O flag was set to off, the destruction process will then proceed as usual. However, if it was set to on, the following procedure will be performed:

- The file table destroyer of the I/O subsystem will set the to-be destroyed process' `pending_destroy` flag, and cancel the destruction. This will inform the process manager that the destruction is pending, and the `PROC_STATE_PENDING_KILL` flag will be set to the TCB.
- Upon I/O operation completion by the file handler, the I/O subsystem will check if the `pending_destroy` flag is set. If it is the case, the process destroyer is then invoked again, and the process destruction proceeds as usual.

8.4 Process Waiting

SOS provides a facility for a process to wait for the termination of another process. When a process invokes the SOS system call `WAITPID` on a valid target, the process will be blocked until the target terminates. Additionally, a process can pass a -1 value to the target PID to be waited, in which case SOS will wake up the process if any other process is terminated. Passing an invalid PID will cause the process waiting SOS system call to return an error immediately.

If multiple processes are waiting on a single PID, then all the waiters will be signaled upon the termination of the process that was being waited. However, if multiple processes are waiting on -1, then the termination of one process will only wake up one of the waiters in a *FIFO* (first in first out) fashion.

Because of the behaviour mentioned before, the waiter list for a specific PID will be stored as a bitfield in the target process' PCB in the `waitee_list` field. Each bit corresponds directly to the PID of the waiting process, as a result, the size of this field is proportional to the configured maximum number of processes supported by SOS.

On the other hand, due to the FIFO behaviour of the waiting upon the -1 PID, the list of -1 waiters are implemented as a doubly linked list stored in a static array in a similar fashion as the logical frame storage.

9. Timer Subsystem

The timer subsystem in SOS provides user application's capability to sleep for a certain amount of time, with the accuracy of 1 millisecond. This subsystem also allows SOS to tell the passage of the time. This subsystem utilizes the meson timer hardware and is using the meson timer driver provided by Data61.

9.1 Timestamp

The timestamp is defined as the passage of the time in microseconds since the timer was initialized. This subsystem will utilize meson timer E for this function. Since timer E has a very large 64 bit counter, the timer E will be initialized once for the SOS's lifetime. If any of the SOS's subsystem requires a timestamp, which includes handling the user application's request for the `TIMESTAMP` SOS system call, this subsystem will simply return the value of the timer E from the DMA.

9.2 Sleep

Only the meson timer A is used for implementing the sleep operation. The timer A can be configured to fire an interrupt after a certain passage of time. Since timer A only has a 16 bit counter, the maximum amount of delay time that can be waited is 65535 milliseconds if the timer was configured to use millisecond precision, which is always the case with this timer subsystem. This decision allows us to make the code simpler while at the same time not making the timer spam the SOS with periodic 65-millisecond interval, at the expense of the maximum timer accuracy is limited to 1 millisecond.

The timer A is utilized in a tickless fashion, which means that the timer is not set to fire periodically at a fixed time interval. Instead, the timer will only be set to fire interrupt for the delay required by the subsystem's client.

The sleep queue is implemented as a sorted singly linked list, ordered ascending by the target timestamp of each queued event. The target timestamp is the value of the timestamp in which the event should be fired, which is equivalent to the then-current timestamp when the event was enqueued plus the required delay.

If the timer A's interrupt fires, the head of the queue will be checked against the current timestamp. If the current timestamp is larger to or equal than the head or is smaller than the head but under 1 millisecond difference, then the event will be fired. Otherwise, timer A's delay will be set to the difference of the timestamp up to a maximum amount of 65535 milliseconds. This approach allows the timer subsystem to enqueue events that require delays longer than what is supported by the meson timer A.

10. SOS System Call References

List of SOS system call number is available on the `include/sossysnr.h` file in the `libsosapi`, with the prefix of `SOS_SYSCALL_*`. The following list describes all supported SOS system calls, their required parameters, as well as their return arguments. All arguments and return values are stored in the IPC buffer message, and are of type 64 bit signed integers. All SOS system call requests must be sent to the capability pointer "1". The following conventions apply for the SOS system call:

- When sending an SOS system call request, 0th word is used as the system call number, and the rest of the words are used for the arguments to the system call. See the list below for the arguments.
- Upon receiving a reply from SOS system call, the 0th word is used as the return value of the SOS system call for those SOS system calls that have a return values. Some SOS system calls may return a structure instead, which will begin at the same base address as the message buffer of the IPC buffer. User application can distinguish between the two by checking for the returned message length, where a structure reply will always have the word length of more than 1. Consult the list below for the meaning of the return value.

The address of the IPC buffer in the user application's virtual address space can be retrieved from the 8th word from the stack top (for example, offset `0xfc0` from the topmost stack page). The structure of this IPC buffer follows the standard seL4 convention, for example, the first word is the tag (which is unused here in SOS), and the message starts from the 2nd word.

- `READ`
  SOS system call number: 0
  This SOS system call will read at most the number of bytes given by the 3rd argument from the file handle supplied at the 1st argument to the buffer given at the 2nd argument. Depending on the underlying file handler and the conditions, the amount of read data might be less than specified, or it might block until additional data is available.
  Arguments:

  1. The unique user application's file handle returned by the `OPEN` SOS system call.
  2. Pointer to a buffer that will receive the data.
  3. Maximum size of the data to be read in bytes.

  Returns: Number of bytes read if successful, negative `errno` otherwise.

- `WRITE`
  SOS system call number: 1
  This SOS system call will write at most the number of bytes given by the 3rd argument to the file handle supplied at the 1st argument from the buffer given at the 2nd argument. Depending on the underlying file handler and the conditions, the amount of written data might be less than specified, or it might block until the underlying file is ready to accept more data.
  Arguments:

  1. The unique user application's file handle returned by the `OPEN` SOS system call.
  2. Pointer to a buffer that contains the data to be written.
  3. Maximum size of the data to be read in bytes.

  Returns: Number of bytes read if successful, negative `errno` otherwise.

- `OPEN`
  SOS system call number: 2
  This SOS system call will open the file associated with the file name given at the 1st argument on behalf of the calling user application. The exact behavior depends on the underlying file handler that will be associated with the file name.
  Arguments:

  1. Pointer to a file name, which is a buffer that contains a NULL terminated string.
  2. Length of the string pointed by the 1st argument.
  3. File mode to be used. This mode follows the `fcntl` semantics. For example, the The `O_RDONLY`, `O_WRONLY`, or `O_RDWR` flags can be used to limit the data direction that will be supported by the opened file handle.

  Returns: An integer that represents the file handle if successful, negative `errno` otherwise. Note that zero is a valid file handle.

- `CLOSE`
  SOS system call number: 3
  This SOS system call will close the file handle returned by the `OPEN` or `OPENDIR` SOS system call. After this operation, user application cannot operate on the file handle anymore.
  Arguments:

  1. The unique user application's file handle returned by the `OPEN` or `OPENDIR` SOS system call.

Returns: None.

- `STAT`
  SOS system call number: 4
  This SOS system call will retrieve the information of the file name given in the 1st argument.
  Arguments:

  1. Pointer to a file name, which is a buffer that contains a NULL terminated string.
  2. Length of the string pointed by the 1st argument.

  Returns: The `sos_stat_t` structure describing the stat-ed file if successful, negative `errno` otherwise.

- `MMAP`
  SOS system call number: 9
  This SOS system call allows a user application to create a new address space region. The user application can use this newly created address space as an additional heap area.
  Arguments:

  1. Reserved. Must be 0.
  2. The desired length of the region in bytes. This length will be rounded up to the nearest multiply of 4 KiB.
  3. The protection mode flag. This argument follows the `prot` semantic of `mmap` system call in POSIX's `fcntl`. Only combinations of `PROT_WRITE` and `PROT_READ` is supported, and at least one of those flags must be specified.
  4. The flag for the region. This argument follows the `flag` semantic of `mmap` system call in POSIX's `fcntl`. However, only `MAP_ANON` is supported here, and has to be specified. Specifying any other value will result in failure.
  5. Reserved. Can be filled with any value.
  6. Reserved. Can be filled with any value.

  Returns: Base address of the newly created region if successful, negative `errno` otherwise.

- `MUNMAP`
  SOS system call number: 11
  Invalidates a subset of the region returned by `MMAP` SOS system call pointed by the 1st argument with the length of 2nd argument. The region represented by both the 1st argument and 2nd argument must be a valid subset of the calling user application's mmap region. Upon a successful operation, the memory region specified cannot be accessed anymore.
  Arguments:

  1. A pointer to the base address of the region to be unmapped. This argument must be divisible by 4 KiB.
  2. Length of the region to be invalidated in bytes. This argument must be divisible by 4 KiB.

  Returns: 1 if successful, negative `errno` otherwise.

- `BRK`
  SOS system call number: 12
  Set the break (heap) limit to the target specified in the 1st argument.
  Arguments:

  1. A pointer to the new break limit, or 0 to get the current break limit.

  Returns: The new break limit (or current break limit if argument was 0) if successful, negative `errno` otherwise.

- `USLEEP`
  SOS system call number: 35
  Blocks the calling user application for the amount of microseconds specified in the 1st argument. In this version of SOS, accuracy is limited to 1 millisecond (1000 microseconds).
  Arguments:

  1. Time to wait in microseconds.

  Returns: 1 If the sleep has been performed successfully, negative `errno` otherwise (This could happen for example, if SOS is exhausted of memory used to store the sleeping queue).

- `MY_ID`
  SOS system call number: 39
  Use this SOS system call to get process ID. This SOS system call never fails.
  Arguments: None.
  Returns: Process ID of the caller.

- `PROC_NEW`
  SOS system call number: 57
  This SOS system call allows a user application to start another user application.
  Arguments:

  1. Pointer to a file name which contains the ELF executable image to be started. The file name is a buffer that contains a NULL terminated string.
  2. Length of the string pointed by the 1st argument.

  Returns: Process ID of the newly started user application if successful, negative `errno` otherwise.

- `PROC_DEL`
  SOS system call number: 62
  This SOS system call allows a user application to stop another user application. After a successful invocation of this SOS system call, the target process might not be immediately stopped, as they might have been busy performing I/O operations. The process destruction section provides more details.
  Arguments:

  1. Process ID of the target process to be stopped.

  Returns: 1 if successful, negative `errno` otherwise.

- `TIMESTAMP`
  SOS system call number: 201
  Retrieves the duration of SOS uptime in microseconds since SOS startup.
  Arguments: None.
  Returns: The duration of SOS uptime in microseconds since SOS startup.

- `WAITPID`
  SOS system call number: 247
  This SOS system call allows a user application to block until another user application matching the criteria is stopped. The process waiting section provides more details about the waiting behavior.
  Arguments:

  1. Process ID to be waited for deletion, or specify -1 to wait for any process deletion.

  Returns: Process ID of the stopped user application if successful, negative `errno` otherwise.

- `LIST_PROC`
  SOS system call number: 391
  This SOS system call allows a user application to view list of all user applications and their PID currently running in SOS, alongside some basic info upon them.
  Arguments:
  1. Pointer to an array of `sos_process_t` structure.
  2. Size of the buffer in the 1st argument, in the number of elements of `sos_process_t`.

  Returns: Number of `sos_process_t` structures filled in the supplied buffer.
- `GROW_STACK`
  SOS system call number: 1001
  Set the limit of the process stack by the number of pages specified in the 1st argument. Note that the stack can be shrunk as well, by setting a smaller value to the 1st argument.

  Arguments:
  1. The desired size of the stack in pages.

  Returns: New number of pages available for the user application's stack.
- `OPENDIR`
  SOS system call number: 1781
  This SOS system call will open the specified file name given at the 1st argument on behalf of the calling user application as a directory.
  Arguments:
  1. Pointer to a file name, which is a buffer that contains a NULL terminated string.
  2. Length of the string pointed by the 1st argument.

  Returns: An integer that represents the file handle if successful, negative `errno` otherwise. Note that zero is a valid file handle.
- `DIRREAD`
  SOS system call number: 1782
  Get the file name from a directory with the given file handle specified at the 1st argument from the index specified at the 2nd argument.
  Arguments:
  1. The unique user application's file handle returned by the `OPENDIR` SOS system call.
  2. The index of the directory entry that the user application wishes to retrieve.
  3. Pointer to a buffer that will receive the file name.
  4. Size of the buffer pointed at the 3rd argument in bytes.

  Returns: The length of the requested file name if successful, negative `errno` otherwise.

NOTE: negative `errno` return semantics has the same semantic as POSIX's `errno` values, but with the value multiplied by -1 instead.

10.1 Structure Definitions

These following structures are used for interfacing with SOS system calls.

`sos_stat_t`

```
typedef struct {
    st_type_t st_type;    /* file type */
    fmode_t   st_fmode;   /* access mode */
    unsigned  st_size;    /* file size in bytes */
    long      st_ctime;   /* Unix file creation time (ms) */
    long      st_atime;   /* Unix file last access (open) time (ms) */
} sos_stat_t;
```

`sos_process_t`

```
typedef struct {
    pid_t     pid;
    unsigned  size;             /* in pages */
    unsigned  stime;            /* start time in msec since booting */
    char      command[N_NAME]; /* Name of exectuable */
} sos_process_t;
```

10.2 Miscellaneous

The libsosapi and libsel4runtime library are provided in the source tree. User applications can link to these libraries to abstract away seL4 system calls as well as SOS system calls in a neat and easy function call style model. Furthermore, libsosapi can be linked together with musl libc to provide a POSIX style programming model (very loosely) to user applications (to activate this function, the `sosapi_init_syscall_table` function of libsosapi needs to be called upon startup).

11. Configurations

Various SOS parameters can be configured by setting up certain CMake variables. The following table lists the supported CMake variables.

Name	Description	Default Value
`SosNFSDir`	The mount point used by the NFS file handler.	`/export/odroid`
`SosGateway`	The NFS server for the NFS file handler.	`192.168.168.1`
`SosFrameLimit`	Limits the memory frames (not to be confused with logical frames) to be allocated by the frame table. Unlimited if set to `0`.	`0`
`SosUseFakePageFile`	If set to other than 0, enables the fake file system and points the page file to it.	`0`
`SosFakePageFileSize`	Control the size of the fake file system in bytes. Has no effect if fake file system is disabled.	`10485760`
`SosPageFileName`	Set the name of the page file if using the NFS file handler.	`pf`

Name	Description	Default Value
`SosUseLocalFS`	If set to other than 0, sets the default file handler to CPIO instead of NFS.	`0`
`SosMaxPID`	Controls the maximum amount of running user applications. One process value, the PID 0, is reserved for SOS itself.	`128`
`SosExtraThreads`	Controls the maximum amount of additional internal SOS threads (that is, in addition to the background threads for each user application).	`8`
`SosIntThreadsStackPages`	Controls the number of stack pages for each internal SOS threads. Note that some threads will not honor this configuration.	`2`
`SosMaxFileName`	Controls the maximum length of file names. The size specified here includes the NULL terminator.	`4096`
`SosDisableCPIO`	A CMake boolean value. If set to true, the CPIO archive in the boot image will contain the kernel, ELF loader, and SOS only (no user applications).	`false`

12. Summary

12.1 General Shortcomings

This project is far from perfect because of the limited development time poured into this project. The following are a non-exhaustive list of general shortcomings of this project:

- Free memory frames are stored in such a manner that looking it up requires a linear search.
- The entire network subsystem will stop working if the LIBNFS connection to the NFS server timed out, requiring a reboot to restore the networking functionality.
- Timer accuracy will be reduced if the system is thrashing.
- Many important memory pages, including stack pages for internal SOS threads and other arrays such as frame table, are allocated directly from untyped, which are not subject to paging and prone to early exhaustion when the amount of the actual physical memory is small.

12.1.1 Future Work

This project has a lot of room for improvement. A non-exhaustive list of short term goal would be:

- Use a doubly linked list data structure for the memory frame list, to enable constant time search for free memory frames.
- The queue for the timer driver could be put in a static array instead of using `malloc`.
- Decouple the ethernet driver from the networking system.
- Provide a way to automatically reconnect the LIBNFS when the connection to the NFS server is lost.
- Run the page file management in a separate thread.
- Utilize the frame table for allocating dynamic SOS memory pages.

12.2 Better Solutions

The following lists summarize which of the "better solutions" part from each of the milestone specifications.

1. **Milestone 1**
   - A *tickless timer* where interrupts are only generated when threads need waking (or due to the finite size of the timer).
     Already implemented in the timer subsystem.
2. **Milestone 2**
   - Have a clear framework for handling all blocking within SOS in a consistent way.
     The file abstraction in the I/O subsystem implements this framework.
3. **Milestone 3**
   - As much heap and stack as the address space can provide.
     SOS is capable of supporting the entire 48-bit virtual address space for each process with arbitrary region size.
   - Designs that probe page table efficiently - minimal control flow checks in the critical path, and minimal levels traversed.
     The page table is probed in using the typical indexing scheme of a four-level page table.
   - Designs that don't use SOS's malloc to allocate SOS's page tables (to avoid hitting the fixed size of the memory pool).
     The page tables are allocated from untyped.
   - Designs that have a clear SOS-internal abstraction for tracking ranges of virtual memory for applications.
     The region abstraction model is used to implement this abstraction.
   - Designs that defer the actual mapping of pages until they are used (mapping on the fault rather than always mapping on the initial request).
     The implementation of this part is described in the user memory section.
4. **Milestone 4**
   - Solutions that support multiple concurrent outstanding requests to the NFS server, i.e. attempts to overlap I/O to hide latency and increase throughput.
     Each user application that performs I/O operation will have a dedicated SOS thread that will do I/O in SOS on behalf of the respective user applications. When multiple user applications are doing I/O simultaneously, SOS will execute those operations on different threads, and as such, will have multiple outstanding requests to LIBNFS.
   - Better solutions only *pin* in main memory the pages associated with currently active I/O. Not necessarily every page in a large buffer.
     SOS does not pin any memory pages for the buffer I/O operation. When the file handler faults, SOS will remap the frame associated with the I/O buffer (which is also typically shared with the user application).
   - Avoid double buffering.
     The file SOS system call section describes that user application's frames that corresponds to the I/O operation is mapped into SOS's virtual address space directly, with no copying involved.
5. **Milestone 5**
   - Solutions that do not increase the page table entry size when implementing demand paging.
     The shadow page table does not store anything related to the paging file, only to another page table or a logical frame index. The paging information is stored in the frame table instead.
   - In theory, support large page files (e.g. 2-4 GB)

A total of 8 GiB space is shared between physical memory and paging file, therefore up to almost 8 GiB of page file is supported if the physical memory frame limit is set to a small number. The frame table section describes this limitation.

6. **Milestone 6**
    - Lazy load the executable or data from the file.
      The ELF loader for the process creation will mark a certain SOS virtual memory region to be backed by the ELF file, in which the ELF loader will fault on that region and lazy load the ELF file.

12.3 Advanced Parts

For the advanced parts, only anonymous `MMAP` and `MUNMAP` is implemented in this project. This implementation allows musl memory allocator to allocate larger `malloc`s. The detail of this feature is described in the User Application's Memory section.