University of New South Wales

COMP9242 - Advanced Operating Systems

SOS - Spaghetti Operating System

Design Documentation

William Chan - z5109880
Patrick Song - z5059167
October 25, 2018

Contents

1 System Timer

The timer driver is implemented as a single threaded driver and uses Timer F to manage every timer request. The timer ticks every 10ms and executes every request that has been scheduled from the current tick to 9.999ms into the future. It is approximately accurate up to 2ms.

We use a priority queue implemented as a linked list so that the interrupt handler can get the next request in O(1) time. However, insertion is done in O(n) time, thus the more requests done in the same 10ms time frame, the slower the interrupt handling becomes.

A tickless timer was decided against due to the timers we tried to use not working as we thought they did, and producing unexpected and strange results.

2 Execution Model

SOS uses coroutines to handle multitasking. The picoro[1] library was used to help manage our coroutines. We decided to use picoro since it was a simple implementation which we could easily add to our operating system without any major hassle. A few extra functions were added to the library to further simplify its use in managing coroutines.

Each coroutine is created through either a VM fault or a system call, the syscall loop receives the request for the fault and dispatches a coroutine for it. When a coroutine yields, it returns to the syscall loop to ready itself to dispatch another request.

3 Frame Table

The frame table is allocated at the boot time of SOS. It uses the sel4 untyped memory allocator to remap the untyped memory into useable seL4 page objects. The frametable is then mapped into the SOS address space at `SOS_FRAME_TABLE` defined in `projects/aos/sos/src/vm/vmem_layout.h`. The size of the frame table is determined by the amount of untyped memory in the system that is found by sel4, multiplied by the size of a frame table entry. The size of the frame table is limited to account for other objects that have been allocated by the untyped manager before the frame table was allocated.

The frame table entry stores the following data:

The data in the frame table entry stored is related to 3 factors. First, the frame table entry attempts to store any data that allows mapping and page memory to be free'd and restored back to the type manager. Secondly, data is stored to

Table 1: Frame Table Entry

Type	**Name**	**Description**
seL4_CPtr	cap	Capability for the frame mapping to SOS virtual address
seL4_CPtr	user_cap	Capability for the frame mapping to a threads virtual address
ut_t*	ut	A pointer to the untyped memory information so that it can later be retyped to an untyped with ut_free
seL4_Word	next_free_page	The next free page if the frame has no cap
pid_t	pid	The process id this frame belongs to (Note: this value is not necessarily set, default: -1) it is used for a sanity check in the page table
seL4_Word	user_vaddr	The user vaddr this frame is mapped to
bool	ref_bit	Reference bit to support 2nd chance clock replacement
bool	important	If important is set, the frame cannot be paged out

facilitate demand paging. Thirdly, it stores useful information to ensure sanity throughout the operating system.

To create a frametable entry and therefore get a frame mapping, frame_alloc is called and a page number and virtual address where it is mapped is given back. Only cap, ut are set by default and it is up to the caller to set the rest of the frame table as needed. This is as in some circumstances, particularly in the kernel space, the user fields are not needed. However, leaving the data in the frame table entry ensures a simpler system design, when it comes to free'ing and unmapping frames.

Frametable lookup and alloc'ing is O(1) (when frametable is not full, i.e. no clock replacement).

A high, low watermarking system, was decided against. This is because SOS only frees user mappings once a process is destroyed, this means that frame free'ing is essentially already completed in batch.

4 Pagetable structure

4.1 Pagetable

The pagetable contains two pointers. One to a 4 level pagetable structure and the other to a list of seL4_page_object_frames. The pagetable and a PGD is

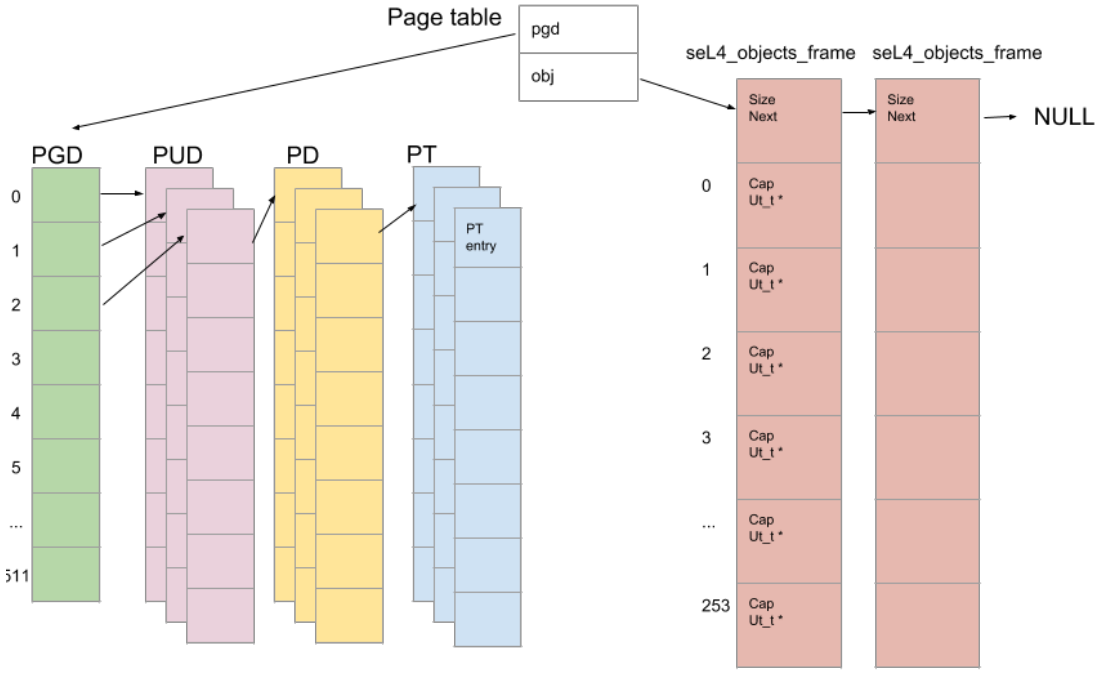

Figure 1: Page table Structure

created when a process is created. Once any data is mapped in the address space, usually from a vm fault. The last 48 bits are used map the address into the page table. The first 9 of the last 48 bits form an index to PGD, the next 9 to PUD, the next 9 to PD, the next 9 to PT. The last 12 bits are the offset to the page. The reason for this page indexing is so each paging structure is the size of one 4K page exactly. The PT finally contains a page table entry, which holds information to access the frametable so that page structures can be free'd or paged out in the future.

4.2 Pagetable Entries

To ensure the page table was compact and page aligned, the entry is stored as a seL4_Word which is 8 bytes in size. The first 8 bits in the page table entry are dedicated for storing bit information and the rest of the bits store a entry number which indexes either the frame table or page file depending on which bits are set. The bit information can be found in `vm/pagetable.h` and in the table below.

The above limits the size of the overall frametable to have a maximum of 2^{56}

entries. With a page size of 4K that is still a huge amount of a data and certainly has no effects on the O-Droid C2 which this operating system is designed on.

Table 2: Page Table Entry

Bits	Name	Description
0	P_INVALID	Mark the page as invalid (this is set by default)
1	P_PAGEFILE	If this bit is set, the index refers to an index in the pagefile otherwise it refers to an index in the frametable
2-8	Unused	Currently unused
9-64	Index	The index number to the pagefile or frametable

4.3 seL4_page_object_frame

Table 3: seL4_page_object struct

Type	Name	Description
ut_t*	ut	the ut pointer used to create a seL4 paging structure
seL4_CPtr	cap	the cap used to create a seL4 paging structure

Table 4: seL4_page_object_frame struct

Type	Name	Description
uint64_t	size	entries used in the current frame
seL4_page_object_frame *	next	Pointer to the next frame in the list
seL4_page_object[253]	page_objects	Array of a ut_t and cap pair used by seL4

The seL4_page_object_frame store any extra page mapping capabilities and ut pointers used for a process so they can be free'd when the process is destroyed.

As it is implemented as a list of frames, the pages need to store the next frame. It also uses a size pointer so it knows at which index to store the next available object and when to allocate a new frame.

Due to the above, to ensure it is page aligned and uses as much data as possible in the page the page_objects array ends up being of size 253.

4.4 Region Abstraction

A region abstraction similar to OS161 is used for paging as well. This manages the permissions for regions and where vm faults are allowed and properly

handled.

If a region does not exist or have proper permissions the system will panic and dump the registers and die.

5 Demand-Paging

The frametable uses a clock replacement policy to page out frames and reuse that frame on demand, the replaced page's raw data is written to a pagefile and its pagefile index is written to the pagetable. Once a fault triggers for the file the lookup will indicate it is in the pagefile at a specified index. At that point the file is read back into an available frame and that pagefile index is added back into a list of available indexes.

The pagefile contains raw data and can therefore support maximum 2^{64} bytes as it uses libnfs positional read which can only supports offsets of a 64bit number. However the pagefile can fail if a lot of data has been written and then free'd as the free list could be large and could use up heap.

5.1 Clock Replacement

Once the frametable is full,a clock index loops around every frame index and attempts to find a frame where the reference bit is set to 0, if it is not 0 and not important it will set the reference bit to 0 and unmap the user_vaddr. Once a page is found it is handed off to the pager to write out and set the pagetable entry correctly.

The user_vaddr must be unmapped to allow the address to fault back in so the operating system knows when to set the reference bit back to 1, indicating that the bit has been touched again and not to replace that frame.

5.2 Concurrency Limitations

Paging in and out uses a kernel buffer. This means that there can be collisions with copying in and out data as reading from a file is asynchronous. In the future a lock could be implemented over the kernel buffer to prevent this from occurring.

6 System Calls

6.1 Jump Table

Each system call is handled by the syscall loop in `projects/aos/sos/src/main.c` which creates a coroutine for each request. The syscall handler then checks if the syscall is valid and executes the syscall via a jump table. Each entry in the jump table only takes in the current process as a parameter since most system calls only require information from the process that requested a system call. Searching the jump table takes O(1) time and makes the code a lot more readable. This is good since system calls happen on a very frequent basis, i.e. when mass reads/writes are occurring.

6.2 Syscall Transmission Conventions

Each syscall returns the number of words that were sent through the IPC buffer back to the syscall handler. If larger buffers are required, that information gets sent to the shared buffer and then the client reads that information out of the shared buffer and vice versa. Data that is usually stored on the heap is accessed through this shared buffer.

6.3 Shared Buffer

The shared buffer is a per-process buffer which facilitates the sharing of resources. A large region of memory (around 32MB) was allocated to store this buffer. The region is partitioned such that each process gets 1MB of memory to transfer data to and from the client and the server. Each process has a hard cap of 1MB so that no process can overwrite one another. Since we currently only support 32 processes, each buffer is accessed by using the pid as an offset. There is a hard cap of 1MB since this amount of memory is large enough that if very large buffers were to be read, the syscall overhead would not be as taxing.

6.4 Blocking Framework

A new coroutine is created each time a system call is requested. If a process were to block, the current coroutine is saved by a structure (depending on what initiated the block) and then returns back to the syscall loop to wait for a new request to occur. When the process wants to unblock, it resumes from where it originally yielded. This ensures the system is lively and does not block the entire system over menial duties (such as waiting for I/O).

7 File System

7.1 File Table

The File Table is implemented as an array of pointers to open files. It currently only supports 16 open files. The file table is a per-process structure which holds

Table 5: Open File Structure

Type	Name	Description
struct vnode *	vn	Pointer to the Vnode
uint64_t	offset	Current offset of the file
int	flags	The access mode of the file
size_t	refcnt	Amount of references to this file

the currently open files of the process. By default, STDOUT and STDERR are open on the console on file descriptors 1, and 2, respectively. STDIN must be opened explicitly on the console before any input can be read.

Lookup is done in O(1) time since the file table is just an array. This makes reading and writing operations much quicker and helps reduce the system call overhead. Insertion is done in O(n) time, since a lowest-available policy is used to assign file descriptors.

7.2 Virtual File System

We use a VFS for easier access to multiple file systems, i.e. devices and NFS. Through our VFS we can use a standard interface to access devices and NFS with the same functions.

7.3 Devices

Devices are stored as a linked list of devices. Although look-up and insertion is done in O(n) time, we only have one device, i.e. the console. Thus our lookup and insertion is effectively done in O(1) time.

The console enforces a single reader policy by checking if there is currently a reader assigned to the console. If there is, the current process fails to read, otherwise the current process is allowed to read from console.

Reading data from the console is achieved by waiting until a read syscall is requested and only reading when that happens. Once the read is finished, the console no longer stores any data which is currently being read. The console blocks until it reaches a newline or hits the max buffer size of 8KB.

7.4 Network File System

All operations are taken care of by libnfs. The async functions are used in tandem with our blocking framework (Section 6.4), to create a synchronous file system.

8 Process Management

8.1 PID

Table 6: Process ID

PID	Description
0	Kernel process (cannot be killed)
1	Init process (cannot be killed)
2-31	Regular processes

Process ID is assigned from 1 onwards, after reaching the max processes i.e. 32, it will attempt to assign pid from 0 again.

8.2 ELF Loading

The initial process uses a cpio image to always ensure that an initial process can be started. Afterwards the elf loader uses the file system and attempts to read an executeable file (the entire file, as we must get __vsyscalls section). If the file is not executable or doesn't exist it returns with an error and the process is marked for deletion.

ELF loader uses libelf to read the file information and segments. For each each segment it maps it into the kernel address space at SOS_ELF_VMEM then remaps that information to the process address space. It then deletes the mapping in the kernel address space.

8.3 Process Structure

Processes store the following information

Table 7: Proc Struct

Type	Name	Description
char*	name	Process name
pid_t	pid	Process ID
unsigned int	stime	Process start time
struct addrspace *	as	pointer to address space struct which contains the page table
ut_t*	tcb_ut	ut pointer to TCB
seL4_CPtr	tcb	capability for TCB
ut_t*	vspace_ut	ut pointer to vspace
seL4_CPtr	vspace	capability for vspace
ut_t*	ipc_buffer_ut	ut pointer to IPC Buffer
seL4_CPtr	ipc_buffer	capability for IPC Buffer
ut_t*	stack_ut	ut pointer to stack
seL4_CPtr	stack	capability for stack
struct proc_node *	wait_list	list of process to wake up
struct proc_node *	child_list	list of children processes
struct filetable *	fdt	Open file table
coro	wake_co	coroutine to a sleeping process
bool	protected_proc	If true the process cannot be killed
void *	shared_buf	pointer to shared buffer

Table 8: Process States

State	Description
RUNNING	Normal state, this process is awake and can use asynchronous operations
WAITING	Process is blocked waiting for another process to end
ZOMBIE	Process is marked for deletion

The process struct stores information on each state to facilitate file system operations and also process operations such as kill and wait. On process init it creates the required sections, maps in the shared buffer and attempts to connect stderr and stdout.

8.4 FS Concurrency

As each process has its own buffer used for reading and writing there are no concurrency issues. Async FS requests are all stopped when a process is marked for deletion, no more coroutines are allowed to be made and any current requests are waited on to be finished. At deletion the process closes every file that is in the openfile table.

8.5 Parent/Child Hierarchy

The operating system has a ”weak” parent child hierarchy. What is meant by ”weak” is that there is a hierarchy, but it is not followed by every operation necessarily.

A process that spawns another process is a parent, and the new process is the child. If the parent is killed the child is added as a child to the init process, i.e. process 1.

A process can wait on any process (not necessarily a child, hence ”weak”) if it is given a PID. However, when it is given -1, it will only wait on its children (not all other processes).

8.6 Delete and Wait

Any process can delete any other process, however protected processes will always fail to be marked for death. Delete will mark the process for death and turn it into a zombie and is blocked, as mentioned earlier this stops async requests from being made and suspends the thread. The zombie'd process is added to the end of reap_list.

At each kernel syscall loop, the kernel will run reap(). Reap attempts to delete the process at the head of the reap list and free objects and frames within the process and close any files as long as there are no async requests to be waited on.

Delete and wait will both fail on process that no longer exist or are in the zombie state.

9 Bonus: mmap/munmap

As kernel has its own process structure, an address space was defined for the kernel to manage memory mappings. mmap has its own address space defined in `vmemlayout.h` that is 60.25 GiB in space which is more than sufficient to malloc for the O-Droid. mmap will start from the bottom address (MMAP_BOT) then

seek at the address to find if it is available to map, if it is, it attempts to define a region, if this fails it is because there is an overlapping region at some point. mmap will lazily try add the length of the to be mapped data to see if the next region is available. It will fail if it reaches MMAP_TOP address in `vmemlayout.h`.

If available space is found it will use frame alloc to allocated the required frames and then map it into the mmap address space given. It also sets the frame table entries as important to prevent kernel heap from being paged out.

mmunmap will unmap the mmap address space and free the frames used by mmap, then delete the region structure so that it is free for use.

An issue that was run into was that muslibc uses the same file for both the kernel and user malloc's user mallocs should not be mapped into the kernel address space. To fix this the libsosapi version of mmap returns 0, and if this is the case then it uses the standard malloc code and simply lets it fault in. In the kernel all errors return -1, and will still return the correct value from malloc if mmap fails.

A References

[1] https://www.chiark.greenend.org.uk/ucgi/~fanf/git/picoro.git/