DreamyOS

Glenn McGuire,
William Lonsdale

University of New South Wales

Overview. The purpose of this document is to outline and justify the design of COMP9242 seL4-based Operations System, Dreamy OS.

Table of Contents

1 Executive Summary

Dreamy OS was designed with simplicity in mind, rather than raw performance. This has made for a Simple Operating System, that is trivial to modify and improve - but still able to output considerable performance. This being said, the majority of the *better solutions* were implemented during development which made for a lean, extensible Operating System.

2 Third Party Content

Several third party libraries were used to aid in the development of Dreamy OS. These included a coroutine library, and a ring buffer library.

2.1 Picoro

Picoro is a coroutine library written by Tony Finch (http://dotat.at/cgi/git/picoro.git/). It provides the functionality to resume and yield from coroutines. The coroutines are also scheduled by a minimalistic scheduler. We used, and modified, Picoro to implement our multitasking model. The Picoro implementation can be found in the files `apps/sos/src/coro/coro/picoro.*`

2.2 Ringbuf

The ring buffer library by AndersKaloer (https://github.com/AndersKaloer/Ring-Buffer) was used to implement the input buffer for the serial device. The library can be found in `libs/libutils/src/ringbuf.c`

3 Multitasking Model

Dreamy OS multitasks using coroutines. The decision to use coroutines was for simplicity over performance. Using the Picoro library allowed our implementation to be relatively straight forward, and bug free, to help scale the emulated concurrency of the Operating System. Small modifications to the library were made when needed for basic operations, such as returning the current executing routine.

All coroutines are created in the main event loop. As a VM fault, or syscall is triggered, a coroutine is created and dispatched to process the request. Any time that a coroutine yields for asynchronous operations, it is returned back to the event loop, where Dreamy OS can wait for, and process more incoming requests. Coroutines that do this first register themselves with the asynchronous callback, to be resumed once the data they were waiting on is made available.

4 Testing

The following limits have been provided to assist in testing Dreamy OS. These can be modified to test the system under different conditions.

4.1 Pagefile Limit

To limit the number of pages the pagefile may contain, access `apps/sos/src/vm/pager.c` and modify the C definition:

```
PAGEFILE_MAX_PAGES
```

4.2 Frametable Limit

To enforce an artificial limit on the number of frames the system can manage, access `apps/sos/src/vm/frametable.c` and modify the C definition:

```
ARTIFICIAL_FRAME_LIMIT
```

4.3 Maximum number of Processes

To modify the maximum number of processes that the system supports, access `apps/sos/src/proc/proc.h`, and modify the C definition:

```
MAX_PROCS
```

5 Subsystems

5.1 Timer Driver

The timer driver `libs/llibclock/src/clock.c` is single threaded, and uses the General Purpose Timer (GPT) as an up-counter, configured with a prescale value of 66. This results in an uptick of the timer every microsecond, meaning the counter register can be used as an accurate timestamp of the OS boot duration.

5.1.1 Timestamps To support 64 bit timestamps, on the counter register overflow (which occurs approximately every 1.2 hours), an unused register in the GPT is incremented. This register is used as the upper 32 bits of the 64 bit timestamp. We do not support overflow of the upper 32 bits of the timestamp, however this is unnecessary as a 64 bit overflow occurs approximately every 584,554 years.

There is a race condition in constructing the 64 bit timestamp, given that the upper and lower 32 bits are read separately. Our timestamp function handles these edge cases and reports the timestamp accurately.

5.1.2 Tick-less design Our timer is designed tick-less, meaning the timer generates interrupts only when a registered event is required to occur. To achieve this, the compare register (the value to generate an interrupt on) is set to the timestamp value of the next upcoming event.

5.1.3 Data Structures to Handle Events Timer events in Dreamy OS were chosen to be stored in a heap-based priority queue. This provides an efficient look-up of the next upcoming event in O(1) time complexity and insertions of O(log n) complexity. This resulted in excellent performance during stress testing of the system, with very little overhead.

Although small, the overhead is enough to introduce a race condition whereby a 1 millisecond or less delayed event, could be processed and missed by the incrementing timer. To handle this case, upon event registration, we check if the event delay is less than 1 millisecond, and if so, execute the event immediately.

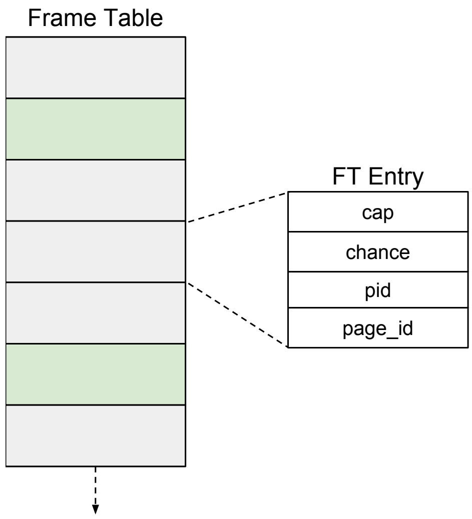
To handle the case where multiple events are triggered within the time to process a single event, our timer looks forward one millisecond and processes all events that were to trigger within that next time slice. Without this, these events would be missed.

5.2 Better Solutions

- ✓ Timestamp accuracy to sub-millisecond without increasing the tick rate of the timer.
- ✓ A tickless timer where interrupts are only generated when threads need waking (or due to the finite size of the timer).

5.3 Frametable

```
apps/sos/src/vm/frametable.*
```



The frametable is allocated by stealing memory from the UT pool at startup. The virtual address for which, like all physical addresses in Dreamy OS, are mapped to a 1:1 window of addresses representing physical memory. This window sits between the addresses of 0x20000000 and 0x60000000, for a total size of 1GB.

5.3.1 Data Structure to Hold Frames

The frametable is an array of frame entries. The table is sized using the amount of remaining UT memory provided to Dreamy OS by seL4. Frame lookup is done in O(1) complexity through indexed access. Frame allocation is also O(1) time complexity regarding the frame table. Converting between OS virtual addresses and indexes into the frametable can be done in O(1).

A design decision was made to reserve 20% of the remaining pool of UT memory for untyped memory, such that there was enough to support Dreamy OS retyping memory into other resources needed by processes. This left the frametable to manage 80% of the remaining memory for use as frames by both the OS and it's user applications. This proportion was decided on after significant testing to ensure the maximum performance of the OS.

5.3.2 Frame Entry

The frame entries, throughout development, have always remained compact to ensure a minimal memory footprint. Pre-paging, the frame entry only contained a capability to the retyped memory. However to support paging, this struct grew to accommodate the required meta-data.

The frame table entry includes the following attributes:

```
typedef struct {
    seL4_CPtr cap;
    enum chance_type chance;
    seL4_Word pid;
    seL4_Word page_id;
} frame_entry;
```

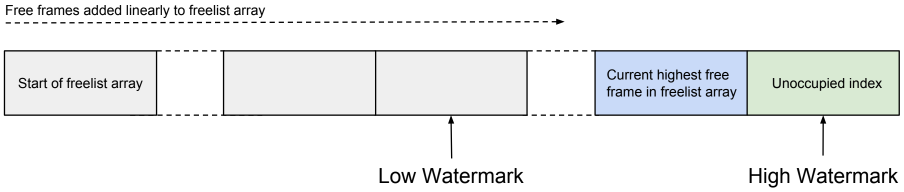
Where *cap* represents the capability for the frame allocation that was performed using *ut_alloc()*, this is saved such that the frame may be properly released.

The enum *chance* represents the state of the frame in reference to the second-chance page-replacement algorithm. The two main states, *FIRST_CHANCE* and *SECOND_CHANCE* specify whether a frame is eligible to be evicted. An additional state, *PINNED*, marks the frame as ineligible for replacement. This is used to ensure frames related to OS meta-data management are always resident, and to synchronize state preventing race conditions.

The attribute *pid* represents the PID of the process which has this frame mapped into their address space. This is used within the paging logic to notify the process that a part of their address space is to be evicted, so that they may fault and remap it later.

Similarly, the attribute *page_id* represents the virtual address which uses this frame as physical memory. This is used in conjunction with *pid* to unmap pages from a processes address space.

5.3.3 High and Low Frame Watermarking


Free frames added linearly to freelist array
Start of freelist array
Current highest free frame in freelist array
Unoccupied index
Low Watermark
High Watermark

For a performance enhancement, Dreamy OS free's frames according to a high watermark of 64. As frames are free'd, they are initially stored in a stack of recently released frames. This allows for quick reuse by another requesting process. When the number of frames in the stack reaches the high watermark, all frames between the high and low watermark and umapped, and their memory released back to the Untyped pool.

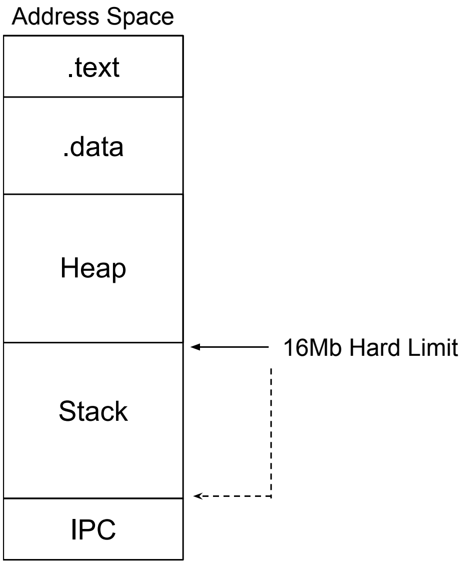
This design decision provided a dramatic performance enhancement in managing memory within the operating system, as the overhead of de-allocations is reduced by releasing memory in bulk.

5.3.4 Better Solutions

- ✓ Designs that can easily be extended to support a second-chance clock replacement policy later in the semester.
- ✓ The frame table is not allocated using malloc.
- ✓ Frame table entries are compact.
- ✓ Designs that avoid deleting and retyping a frame object for every frame free and frame alloc request.

5.4 Address Space

`apps/sos/src/vm/addrspace.*`



An Address Space is our abstraction to represent the virtual address space of a process. Inside an address space is the hardware page table, a per process Dreamy OS page table, and a list of memory regions present in the process.

```
typedef struct {
    region *region_list;
    region *region_stack;
    region *region_heap;

    /* Hardware page table */
    seL4_Word vspace_addr;
    seL4_ARM_PageDirectory vspace;

    /* SOS supplied page table */
    page_directory *directory;
} addrspace;
```

All regions in an address space are stored in a linked list. This decision was made for simplicity rather than performance. However, on average only a low amount of regions would be present per process, therefore traversing the linked list would introduce a very minimal amount of overhead.

As a time / space complexity trade-off, we decided to include a pointer to the heap and stack within the address space for faster lookup, given the frequency of memory accesses in these regions.

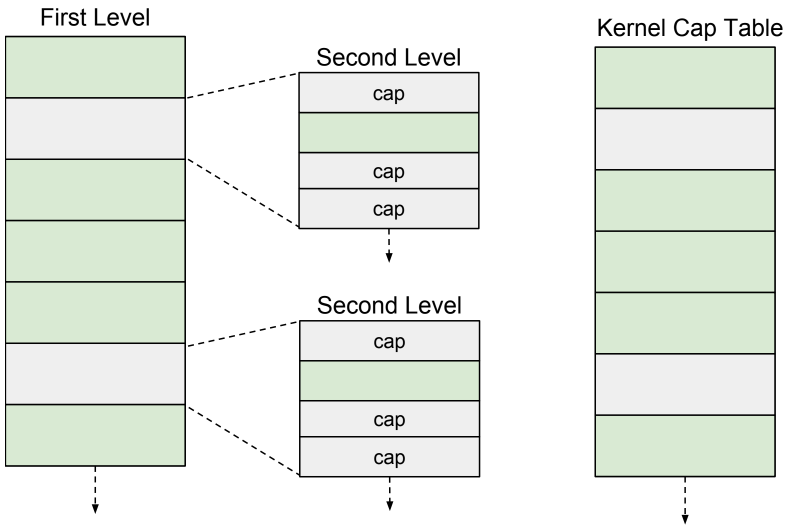
A pointer to the OS page table for this process is included as the attribute *directory*, within the address space structure. The components required to manage the hardware page table, *vspace_ addr* and *vspace*, are also included, as they are needed for seL4 management and process destruction.

5.4.1 Regions Regions represent a contiguous area of virtual memory. In addition to a start and end address, a region has a set of access permission, used to restrict modes of access. A common example of this is read only access. Each process is created with a set of existing regions from the provided ELF Binary. Dreamy OS additionally provides a region for IPC communication, and a stack and heap.

5.4.1.1 Dynamic Stack Initially, a processes stack starts at address 0x90000000 and has a size of 0 bytes. As the process faults on memory locations, the stack is dynamically expanded to accommodate the memory use patterns of the process. The stack grows upwards by moving the starting address lower in the address space. In addition to collision checking (*as_ region_ collision_ check*), the stack is limited in its expansion by the constant *RLIMIT_ STACK_ SZ*, which is by default 16MB. The default permissions on the stack are read and write.

5.4.1.2 Dynamic Heap The heap is initialized starting a page after the last defined section from the ELF Binary. This was chosen in order to maximum the amount of memory the heap could grow in the address space. Similar to Linux, the initial size of the heap is 0 bytes. The *brk* syscall can be used to set the end address of the region. Although extensions are restricted by collision checks with other regions, the heap has no internal limit, meaning it can be extending as much as possible, until it collides with another region in the address space.

5.4.2 Page Table `apps/sos/src/vm/vm.*`



To represent the state of the virtual memory of a process, we use a two level page table to store the capabilities for pages mapped into a processes address space. For a 32 bit virtual address, the highest 20 bits are used as the page id. This means that every page in Dreamy OS is 4096 bytes. To index into the two level page table, the top 10 bits of the 20 bit value is used to index into the top most level, and the remaining bottom 10 bits are used to index into the second level.

Using a two level page table provides the benefit that, the second level of the page table need not to be created if no mapping exist. Therefore on startup, the page table is quite compact, and only grows to accommodate the program as it needs to.

```
typedef struct {
    seL4_CPtr page;
} page_table_entry;
```

Each page table entry only contains 4 bytes, and holds the capability id for the currently mapped in frame. Because of this compact design, the top and second levels of our page table are exactly 4096 bytes, and therefore can be managed trivially by our frame table. Therefore memory for the page table is self contained within Dreamy OS.

To support paging, which is described in a later section, the leftmost bit of *page* cap value is used to indicate whether a page is resident in memory (0), or currently swapped out to disk (1).

5.4.3 Kernel Cap Table During the mapping of pages into the virtual address space of processes, all kernel capabilities which are created for the purpose of mapping into the hardware page table, are stored in a per-process buffer *kernel_page_table_caps*. Since our pagetable was configured sepeartely to the hardware pagetable, we needed to provide a seperate memeory region to store these caps. We are able to store this buffer as several contiguous frames. The array is large enough such that page ids can be used for O(1) index accesss into the array to story and remove capabilities.

5.4.4 Better Solutions

- ✓ As much heap and stack as the address space can provide
- ✓ Designs that probe page table efficiently - minimal control flow checks in the critical path, and minimal levels traversed.
- ✓ Designs that don't use SOS's malloc to allocate SOS's page tables (to avoid hitting the fixed size of the memory pool).
- ✓ Designs that have a clear SOS-internal abstraction for tracking ranges of virtual memory for applications.
- ✓ Solutions that minimize the size of page table entries (e.g. only contain the equivalent of a PTE and a cptr).
- ✓ Enforcing read-only permissions as specified in the ELF file or API calls.

5.5 VM Fault

A process can trigger a VM fault for many reasons; Dreamy OS handles several cases. The execution path for this logic starts in the *vm_fault* function in `apps/sos/src/vm/vm.c`

5.5.1 Permissions Fault Regions of memory are mapped into an address space with certain permissions. For example, the code segment has read only access. Trying to write to these memory addresses would trigger a permissions fault.

Upon triggering a permissions fault, a process is terminated for behaving erroneously.

5.5.2 Translation Fault The majority of faults are translation faults, accesses to memory that is not currently mapped into the address space. As Dreamy OS supports paging, translation faults can be handled in two distinct ways.

5.5.2.1 Page in on fault The page the process is trying to access might have been paged out to disk; This can be check by testing the left most bit of the cap in the page table entry. If the page has been pushed to disk, the pager retrieves the memory and maps it into the processes address space. Permissions checking is also enforced, to ensure that incorrect access is not granted.

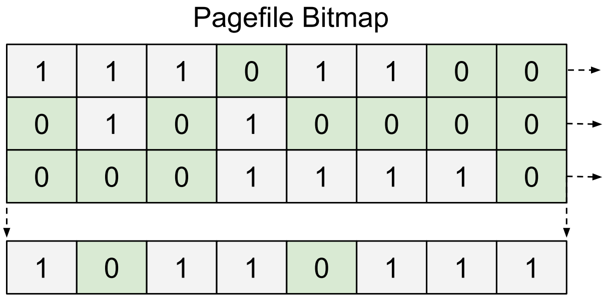
5.5.2.2 Map memory on fault Dreamy OS supports on demand memory mapping. Upon receiving this fault, stack expansion is checked to see if it can support this out of region memory address. If yes, then the stack is expanded to accommodate this address. Otherwise, the memory address is associated with a region and the permissions of that region checked with the access permissions of the fault. If the checks have successfully passed, the memory is created and mapped into the process.

5.6 Paging

`apps/sos/src/vm/pager.*`

Demand Paging was designed for simplicity, not efficiency; As such, this resulted in several trade offs on our design.

5.6.1 Pagefile Metadata



Pagefile Bitmap

1	1	1	0	1	1	0	0
0	1	0	1	0	0	0	0
0	0	0	1	1	1	1	0
1	0	1	1	0	1	1	1

Whilst inefficient, the entire pagefile is represented in memory by a bit array. A 0 in an entry represents the page at that position in the file as free, whilst a 1 specifies a page is currently occupying this position in the pagefile. Though not being the best solution, our decision to use a bit array meant that the memory footprint is as minimal, for example, a 2 Gigabyte pagefile size is managed by a 65 Kilobyte bit array.

Insertions (marking a page as free) is O(1), but finding a free page in the array is an O(N) linear search. Given more time, this would be improved, but it was chosen for simplicity.

5.6.1.1 PTE Size As mentioned in the page table section, our page table entries are compact, and this allows us to manage our page table efficiently through frames. Because we wanted to keep this property, we decided to integrate paging meta-data into the page table entry. The left most bit of the cap value is used to represent if this page is currently paged to disk. If the value is 0, then the memory is resident and the cap value corresponds to a seL4 mapped frame capability. If the value is 1, then the memory has been paged to disk, and the cap value corresponds to the index into the pagefile where the page is stored.

Because we reserve the left most bit for meta-data, this restricts our maximum pagefile size to 2 GB. Because this is the maximum pagefile size that was required by the specification, we decided this would be an acceptable tradeoff.

5.6.2 Paging Race Conditions With multiple processes, several race conditions exist with demand paging. In particular, faults may be triggered on memory that has not yet completely paged due to the asynchronous behaviour of NFS operations. Therefore, to solve this and other race conditions, we implemented a lock around the pager, such that only 1 operation can be done at a time.

As paging operations are triggered, they go through the *page_gate_open* function, which forces the coroutine to yield if a paging operation is already taking place. The coroutine is placed on a waiting queue to be resumed later. When a paging operation has been completed, it exits through *page_gate_close* where it checks to see if there are any waiting coroutines in the queue, and resumes them if so.

This behaviour is not always desired, for example, a *page_in* that also triggers a *page_out* would prefer the *page_out* return immediately instead of resuming other waiting coroutines, therefore we provide avenues to exempt certain coroutines from passing through the locking gate.

5.6.3 Better Solutions

- ✓ Solutions that do not increase the page table entry size when implementing demand paging.
- ✓ In theory, support large page files (e.g. 2-4 GB)
- ✗ Avoid paging out read-only pages that are already in the page file.
- ✗ Solutions that avoid keeping the entire free list of free page-file space in memory.

5.7 Syscalls

`apps/sos/src/syscall/`

System calls are handled initially by a separate coroutine, where the syscall number is checked. Given a valid syscall, the corresponding handler is dispatched via a jump table to complete the main work related to the system call. The use of the jump table reduces look-up time to O(1). Since syscalls are made frequently, it was important to us to reduce the code path before the syscall code could be run.

Every syscall receives the calling process as a parameter, and returns the number of words in the reply message back to the user process. As every syscall receives and sends data back differently, setting and receiving from the message buffer is done within each system call handler.

5.7.1 Syscall Argument Conventions Copying data from and into userland processes is done through sending over the processes virtual address. Dreamy OS will translate this virtual address into an OS virtual address, handling memory mapping, permissions and paging issues during the translation.

Buffers larger than 4096 bytes are handled through progressive translations in the syscalls that require handling large regions of memory. On a page by page basis, these addresses are translated and processed by the system call. For the read and write system calls, the addresses are translated and manipulated directly, without any double buffering. For data related to file names, for example, are copied into a local buffer, because the entire data needs to be contiguous in order to be processed.

5.7.2 Framework for blocking Every time a syscall is made, and received by Dreamy OS; a new co-routine is spawned. Any time a blocking operation can occur, the coroutine is saved by the asynchronous handler, to be resumed when the data is available for processing. Every syscall then yields back to the event loop to process further events. When the corresponding interrupt occurs, the coroutine for the syscall is resumed. This consistent behaviour provides a clear framework for handling blocking operations such that the OS is never stuck waiting and is always able to process incoming requests.

5.7.3 One syscall for buffer transfers Passing virtual addresses as syscall arguments means that the OS has to do more work in order to access and manipulate the memory provided by the user. However, this has the added benefit that only a single syscall is needed to handle large buffers.

5.7.4 Write Implementation Decision As memory address translation is done progressively, we made the decision to perform writes (and reads) progressively as memory is translated. This means that for a large buffer that extends into an invalid region, the memory that is valid is processed and written into, but once an invalid address is translated, the system call returns error to the user process. Therefore part of the requested data will be in the memory location, but it will not be complete, which is acceptable given we notify the user that the system call did not complete successfully.

5.7.5 Directory List Implementation Decision

For completeness, we decided to include the console device as a listing in the results returned from the *sos_getdirent* syscall.

5.7.6 Better Solutions

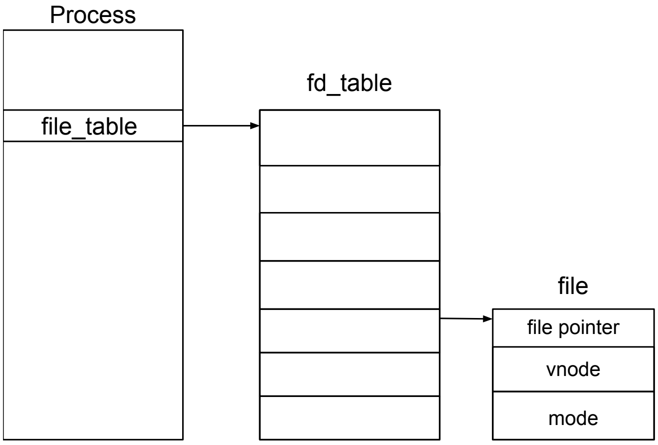
- ✓ Have a clear framework for handling all blocking within SOS in a consistent way.
- ✓ Only make one system call (between sos and user-level) for large buffers.
- ✓ Be able to handle user buffers that are larger than the physical memory size.

5.8 Virtual File System

`apps/sos/src/vfs/`

Access to files and devices are managed by the same virtual interface, the VFS. This design allowed us to treat devices and files as the same, and provides methods to easily mount other file systems and devices.

5.8.1 File Table `apps/sos/src/vfs/file.*`



Per process, there is a file table which represents the files currently opened by the process. By default processes can support a maximum of 16 open files. Inside each slot in the file table is a pointer to a file struct.

```
typedef struct {
    off_t fp;   /* file pointer, current location in the file
    */
    vnode *vn; /* Vnode attached to this file */
    int mode;   /* Mode of access */
} file;
```

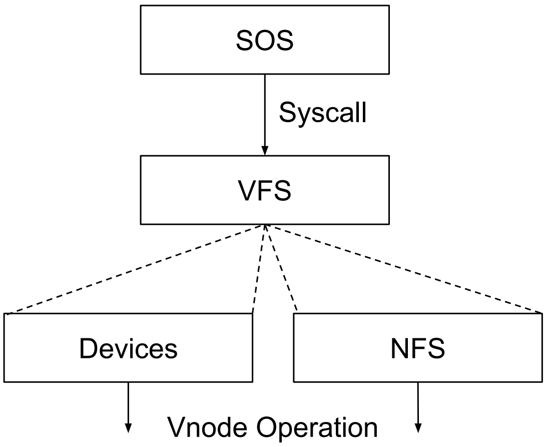
Each file has associated with it, a file pointer that represents the current offset in the file; The mode the file is currently opened with (for example, read only). And finally a pointer to a vnode, which provides the implementation abstraction.

Open, write, read, close, and other operations are all contained inside a vnode as vnode operations. This abstracts away the concrete implementation and allows processes to treat all open files the same.

5.8.2 Vnode `apps/sos/src/vfs/vfs.*`

```
typedef struct _vnode {
    void *vn_data; /* Implementation specific data */
    const vnode_ops *vn_ops; /* Operations on a vnode */
    seL4_Word readcount; /* Number of read references on this
    node */
    seL4_Word writecount; /* Number of read references on this
     node */
} vnode;
```

5.8.3 Name Resolution



Our VFS supports a single namespace, and thus names must be resolved consistently. Namespaces can be mounted onto the VFS and included into the name resolution path. Similar to Linux, Dreamy OS has the concept of mount points, which represent a concrete implementation of a file system.

```
/*
 * Linked list of mount points on the VFS
 * Each mount point is a VFS with vop_lookup defined.
 */
typedef struct mnt {
    vnode *node;
    struct mnt *next;
} mount;
```

Mount points are stored as a linked list. While not advantageous for efficiency, it does provide an easy name resolution strategy. We implemented a first come first serve strategy where by the first mount point to resolve the name successfully is where the file that is chosen. Mount points are appended to the end of the list, so priority is given to namepsaces at the start of the list. For namespaces that has special reserved names, like devices for instance, this is mounted first to ensure that resolution of these names is not confused with files on the file system. This does however restrict the naming of files on any file systems mounted after another that has a file of the same name, however with the given specification we felt it was an acceptable trade off.

Each mount point is a vnode with the list and lookup functions defined, it's at this stage that control is passed over to the concrete implementations.

5.8.4 Devices `apps/sos/src/vfs/device.*`

The device namespace has a linked list of known devices. Devices can be added to this list by registering a device with a name, and a corresponding vnode. A linked list is not ideal for name resolution inside of the device namespace, it was chosen for simplicity over efficiency.

5.8.4.1 Serial Device `apps/sos/src/dev/sos_serial.*`

Serial is a device registered in the device namespace under the name "console". This device enforces a single reader policy; upon opening the device vnode, the reader reference count is checked to ensure that the device is not currently in use. If so, the open fails.

Receiving data from the serial is done through an asynchronous handler which is called every time a byte is available. If a process has requested data from the serial device, this character is written directly into the memory address for the user process. Otherwise, if data arrives and no process has requested any, it is cached in an input ring buffer, to be read from the next time a process requests data from the console. Similar to system calls, if a process requests data that has not yet arrived, the coroutine will be saved and yiled back to the event loop to process more events. It will be resumed (and the process replied to) when data arrives.

5.8.5 NFS `apps/sos/src/fs/sos_nfs.*`

The Network File System is the second mounted namespace in the VFS. All operations are delegated to the NFS library. The method for handling blocking requests is consistent to the rest of the Operating System.

5.9 Better Solutions

Before asynchronous IO operations, only the pages being access by the VFS are pinned. Upon return they are unpinned.

✓ Solutions that support multiple concurrent outstanding requests to the NFS server, i.e. attempts to overlap I/O to hide latency and increase throughput.

✓ Better solutions only pin in main memory the pages associated with currently active I/O. Not necessarily every page in a large buffer.

✓ Avoid double buffering.

✗ Perform no virtual memory management (mapping) in the system call path (for better performance).

5.10 Processes

`apps/sos/src/proc/proc.*`

Dreamy OS processes are designed to be as small as possible, to reduce complexity and increase the number of concurrent processes that the OS can support.

```
typedef struct \_proc {
    seL4_Word tcb_addr;             /* Physical address of the TCB */
    seL4_TCB tcb_cap;               /* TCB Capability */
    seL4_CPtr ipc_buffer_cap;       /* IPC buffer cap */
    cspace_t *croot;                /* cspace root pointer */

    addrspace *p_addrspace;         /* Process address space */
    fdtable *file_table;            /* File table */
    list_t *children;                /* Linked list of children
     */

    pid_t waiting_on;                /* Pid of the child proc
    is waiting on */
    coro waiting_coro;               /* Coroutine to resume
    when the wait is satisfied */

    pid_t ppid;                      /* Parent pid */
    pid_t pid;                       /* Pid of process */
    char *proc_name;                 /* Process name */
    int64_t stime;                   /* Process start time */
    proc_states p_state;             /* Enum of the process
    state */

    size_t blocked_ref;              /* Number of blocks on
    this process */
    bool kill_flag;                  /* Flag to specify if this
     process has received a kill signal */
    bool protected;                  /* Flag to specify if the
    process can be killed */
} proc;
```

Every process has associated with it the meta-data of the thread control block, the IPC buffer capability and the cspace pointer. These are required in order to delete the resources on process deletion.

Each process also has an address space, a file table, and list of children. The list of children being the process id's of the kids.

Waiting is supported by the *waiting_on* and *waiting_coro* attributes.

Each proc also has a pid, the pid of its parent, and other information needed for process listing.

The process also has a state associated with it. This can be *CREATED* for a process that has been created but not yet started. *RUNNING* for a running process and *ZOMBIE* for an exiting but not yet reclaimed process.

5.10.1 Pid Allocation

PID allocation is done using a ring-array, such that allocations are made starting from index 0 upwards to *MAX_PID*, resuming from the last allocation on each new allocation. After reaching max pid, the allocator starts searching again from the start of the array. An allocated pid is one who's index into the process array corresponds to a non-NULL process pointer.

This design choice was made, as it allowed for O(1) look-up, and O(n) insert. This was a desirable trade-off as it was deemed that look-ups are made more often than insertions during normal OS operation - and thus for performance was the better option.

5.10.2 Process Delete When a process is deleted, all resources associated with the running process are destroyed permanently. These includes resources like the file table and the address space. But because the process has yet to be claimed by its parent, information regarding the state of the process, and the pid, is kept until it is waited on by the parent.

5.10.2.1 Voluntary A process can voluntarily exit, either through calling the *sos_process_ delete* system call on itself, or by using the exit() syscall.

5.10.2.2 Involuntary A process can be involuntarily terminated by another process through the *sos_process_ delete*. It was a design decision to implement the delete operation similar to the Linux implementation of kill. Upon deleting a process, it is checked to see if it is currently in the blocked state. A process can become blocked when waiting for Dreamy OS to complete it's syscall or vm_fault. Instead of destroying a process while it is blocked, and handle outstanding asynchronous requests, the process is marked for deletion and will be destroyed by the OS as soon as the process is unblocked. While this does provide the benefit of protecting against outstanding async requests, it does prevent Dreamy OS from killing processes that are waiting for other processes to exit. To make this design decision, we decided to emulate the Linux implementation.

5.10.3 Re-parenting When a process is destroyed, all children, alive or dead, are re-parenting the init process. The init process is a special SOS process where orphaned children are re-parented. Inside the SOS event loop, zombie children of the init process are reclaimed so that their resources may be used by other processes.

5.10.4 Process Wait A process can only wait on its immediate children. This is enforced by checking the request pid with those stored in the children linked list. Upon calling wait, the coroutine is saved to the process in the waiting_coro attribute. The requested child to wait on is stored in the waiting_on attribute.

When a process is deleted, it checks to see if its parent is waiting for that processes exit. If so, SOS will resume the waiting coroutine, and unblock the parent.

Processes that have exited before a parent has called wait on them are marked as zombies and still occupy space in the process array. Only when a parent process has reclaimed its child can it's meta-data be destroyed and the pid reused. Not all process meta-data is needed during this stage, file tables, address spaces, and other run time memory are permanently destroyed.

5.11 Better Solutions

- ✓ A sound strategy for handling waiting on a process that exited quickly (before the call to wait)