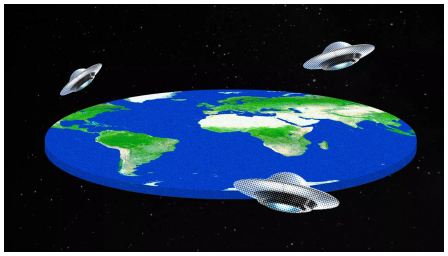
adamyi@, ericnguyen@

TrivialOS Design Doc



Summary
Trivial

Author: adamyi@, ericnguyen@
Status: Draft
Created: 2020-08-08
Self Link: adamy.io/tos

Adam Yi <{adamyi,z5231521}@cse.unsw.edu.au>

Eric Nguyen <{minhthiennhat.nguyen,z5137455}@unsw.edu.au>

Table of contents

Overview

TrivialOS is a trivial operating system implemented on top of seL4. It is designed by favoring simplicity and extensibility over performance. That being said, we have implemented the majority of the *better solutions*. We have also done the following advanced components: *mmap and munmap*, and *Clock driver loaded from the file system*.

Execution & syscall model

Main event loop

TrivialOS uses an event-based execution model. TrivialOS maintains a main event loop to wait for three different types of events:

- Syscalls from user process
- Hardware interrupts (ethernet MAC, watchdog, and timer)
- Process page faults and other faults

Except for interrupts, a coroutine[1] is created in the main event loop to facilitate blocking: it temporarily stores the coroutine to be resumed by a callback function and yields back to the main event loop.

Syscall interface

All user processes communicate with TrivialOS kernel with a pre-defined set of syscalls. A seL4 inter-process communication (IPC) endpoint is opened between the kernel and each application as the communication channel. Each syscall is associated with a fixed syscall number and takes in a few arguments. We pass the syscall number along with the arguments in seL4 message registers (MRs). The syscall return value is passed back to the user process via a single MR as well.

When a syscall argument contains a memory buffer (e.g., file name, I/O buffer), we pass the memory address and size to the kernel without passing the actual content. This way, none of the syscalls requires more than 4 MRs, the number of `seL4_FastMessageRegisters` under aarch64 architecture, enabling us to leverage seL4 IPC fastpath[2] for better performance.

For all syscalls that require the kernel to operate on the content of the process's virtual memory (e.g., read a file name from a memory pointer, write a file stat to a memory pointer), we just use the copy-in/copy-out method to copy the same content between process memory and kernel memory.

The `read` and `write` syscalls are two exceptions - we need to read/write a potentially significant number of pages to/from the user process's virtual memory, and double-buffering degrades I/O throughput. Instead, we directly map those memory frames to kernel address space and

[1] We use picoro for this. https://dotat.at/cgi/git/picoro.git
[2] https://docs.sel4.systems/Tutorials/ipc.html#fastpath

operate on those memory directly. We do not do this for other syscalls mainly for simplicity reasons.

Design considerations

Compared to a multi-threaded kernel, an event-based kernel offers simplicity while remaining adequately efficient with a single-core CPU. Specifically, we do not need much synchronization mechanisms since the kernel just runs under a single thread. This helps us with writing potentially bug-free code and avoiding race conditions.

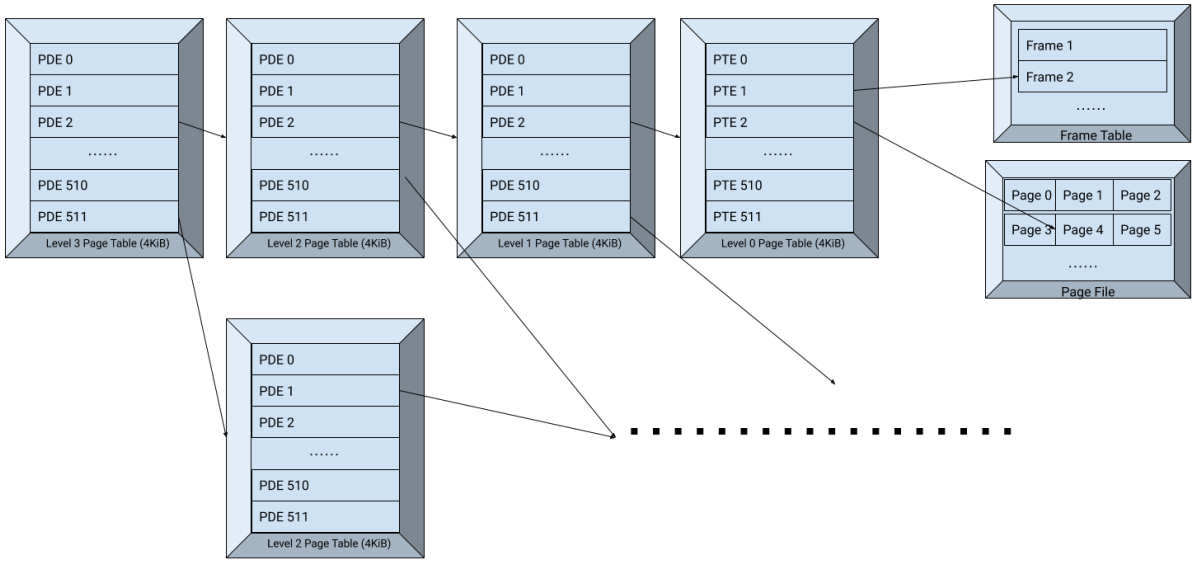
However, we had to ensure that every event handler is short and does not block the entire main event loop. This was a trade-off we had to make and we were happy with the drawbacks.

While we did not implement a microservice-like kernel where different subsystems are split to different servers (processes) and they all use IPC to communicate with each other, we did refactor clock driver out to a separate process, but this later proves to be more troublesome than we thought under our single-threaded model. See Clock driver section for more discussion on this.

Memory management

Page table

TrivialOS follows a similar design for our page table as the aarch64 hardware page table where we use a four-level structure.



It is both simpler and faster to mirror the underlying architecture data structure. With a multi-layer page table, we have decent performance in entry look-up. Contrary to the seL4-internal representation of aarch64 page table, we are re-using the same internal data structure for different levels of page directories. This makes our code more modular and

potentially easy to port to another architecture that uses a similar page table structure with different number of layers (e.g., x86).

Each page table is 4KiB in size and contains 512 entries. Each entry in the page level is 8 bytes long and of 2 types:

- **Page Directory Entry (PDE)**, used by level 1-3 page table

- **Page Table Entry (PTE)**, used by level 0 page table

Each entry uses the `Frame` field to store the frame reference of the page in memory. However, for PTE, when the page is paged out, we no longer need (or have) the frame reference, instead we repurpose the field to store the index of that frame in the page file.

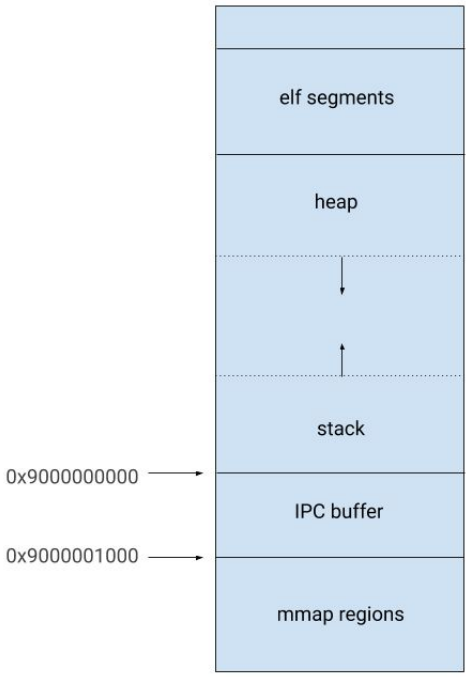
Since TrivialOS's kernel malloc uses a fixed-size morecore area, we want to avoid using mallocs as much as possible in the kernel. For this reason, we purposely designed each page table to nicely fit in a frame with 4KiB size so that we can directly allocate a page from the untyped memory.

A side effect is that we also need to keep track of the capability and untyped memory used by the page table so that we can free them properly when the process quits, but we do not have extra space left for them in the page table after storing 512 entries. Instead, page tables "steal" space from its entries via the 16-bit `Reserved` field. With 2 bytes in each entry, we use the first 4 entries to store the capability pointer and the next 4 entries to store the untyped memory pointer. We do not use the `Reserved` field of the rest of the entries, but this could allow further bookkeeping that might be added in the future for the page tables.

The PTE also has a `Type` field to determine how to handle the mapped memory. There are currently 5 types of page:

- **`IN_MEM`**: this page maps to a frame in memory
- **`PAGING_OUT`**: this page is being paged out by the kernel, potentially blocked due to NFS operations
- **`PAGED_OUT`**: this memory lives in page file
- **`DEVICE`**: this page is mapped to a device (e.g., timer)
- **`SHARED_VM`**: not yet implemented due to time constraints

Address space



Address space is the representation of a process's virtual memory. Pages that are consecutive in memory are grouped into a single region, which is defined and used for specific purposes by the process (e.g., a code region to store and execute code instructions). Each process has its own address space to maintain its memory regions. The address space structure contains a doubly-linked list of memory region structure, as well as a pointer to the stack and heap regions for faster look-up for these regions.

Each memory region has its own range and permissions (readable, writable and/or executable). This provides a level of abstraction between the process address space and the page table, so that we can create and map these pages on demand without preallocating all the pages. In addition, the region data structure contains pointers to the next and previous regions of the process for the doubly-linked list, and the list is sorted by the address ranges of the regions, from lowest to highest in memory.

```
typedef struct region {
      seL4_CapRights_t rights;
      seL4_ARM_VMAttributes attrs;
      vaddr_t vbase;
      struct region *prev;
      struct region *next;
      size_t memsize;
      bool mmaped;
} region_t;
```

Stack

The process stack starts at a defined address specified in `vmem_layout.h` along with other predefined ranges for other regions. The stack grows up, meaning, the kernel will extend the stack region by mapping extra free pages that are above the stack top when the process accesses it. The stack cannot be further extended if there are regions created near or on top of the stack. To allow the stack to grow more, we can potentially move the stack bottom to a higher address in the process virtual address space.

Heap

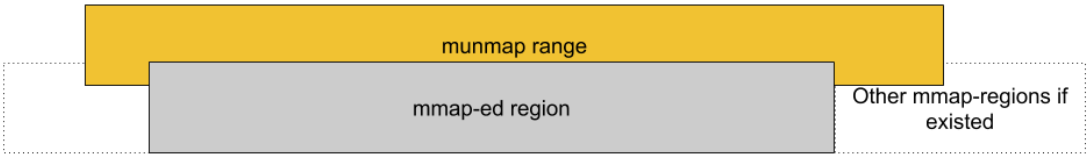
The heap region is located after the last segment of the process's ELFsegments defined in the ELF binary. In a normal ELF binary, the ELF segments are defined in the lower address space. Therefore placing our heap after those segments could potentially give the process more memory for the heap to grow (towards the higher address space). The heap is a dynamic memory region, which can change its size via the call to `sys_brk`. Different to Linux's heap management, we decide to enforce page alignment of our heap for simplicity, i.e. the heap size must be changed to a 4KiB page aligned address. It is the process's responsibility to manage the memory in the heap sensibly, by allocating and freeing memory properly.

Mmap regions

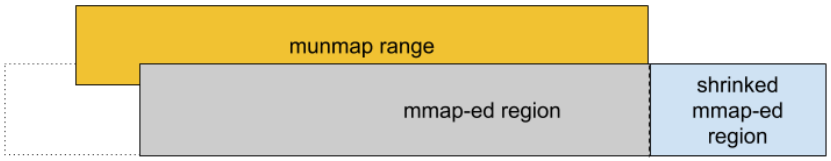
TrivialOS supports `mmap` regions. The process can create a new memory region with specific permissions via the call to `mmap`. The caller can optionally provide a specific address to `mmap`. If not provided, `mmap` uses the space after the last region in address space if there is enough space. `mmap` checks whether it is possible to allocate the whole contiguous memory region, i.e. not colliding with existing regions and returns the start address upon success, or `NULL` if it fails.

TrivialOS provides the feature to destroy the `mmap`-ed regions via the `munmap` syscall. Given an address and the memory size, `munmap` looks up the `mmap`-ed regions that this address lives in and decides how to free or decompose the memory range.

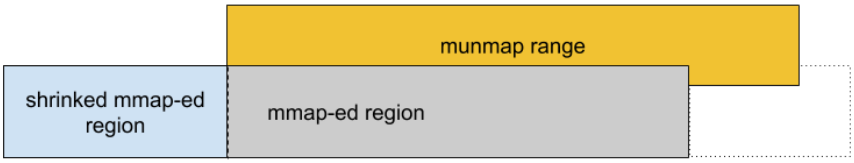
If the provided range is not entirely covered with `mmap`-ed regions, or if it collides with other not `mmap`-ed regions then `munmap` fails and returns to the process. If the range covers one or more contiguous `mmap`-ed regions, then `munmap` goes through each region and tries to break the regions up and destroy/unmap part or all of the pages specified in the range as such:



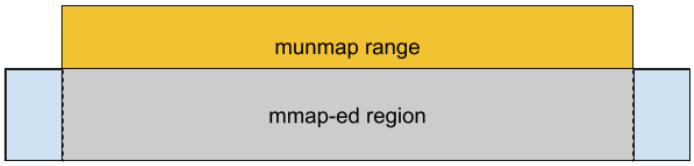
- If the range covers an `mmap`-ed region completely, then `munmap` simply unmaps and destroys this region from memory.



- If the range falls into the situation above, munmap then shrinks the `mmap`-ed region so that it now starts from the end of the `munmap` range.



- Similarly for the above case, `munmap` shrinks the region so that it now ends at the start of the `munmap` range.



- In the case that the range is within an `mmap`-ed region, `munmap` breaks the `mmap`-ed region up into two smaller regions, the first one has the same start address of the initial region but only ends at the start of the `munmap` range, while the second one starts from the end of the `munmap` range and ends at the end address of the initial region.

Page fault

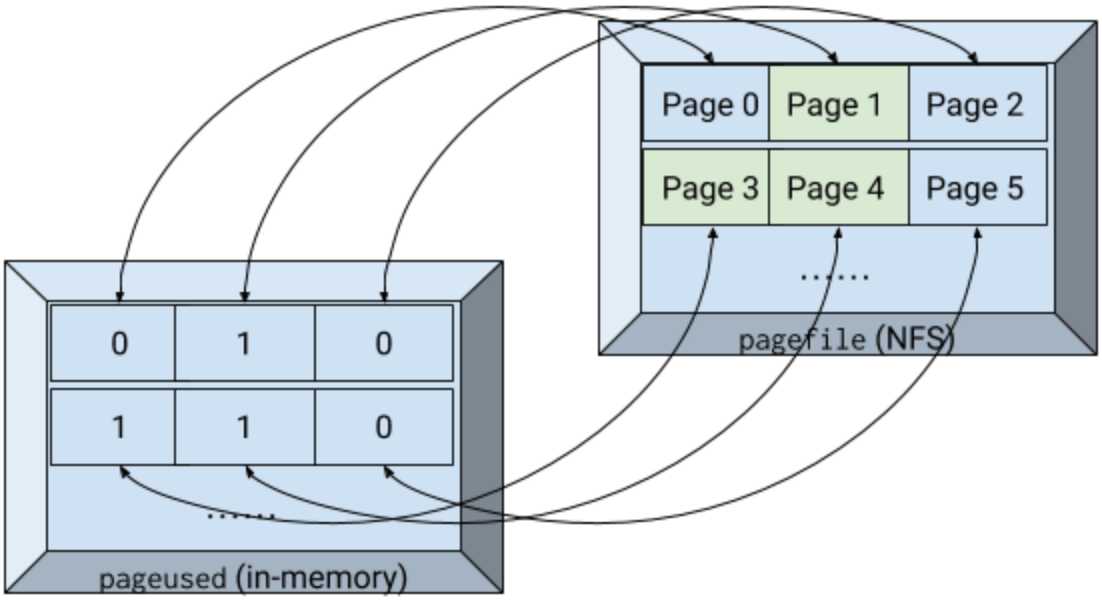
When a process accesses a memory address that does not have a TLB mapping, it gets a page fault received and handled by the kernel in the main event loop. The fault can be categorized in the following cases:

- Kill the process if it is a page fault originated from a permission issue.
- If the address does not belong to any memory region then the kernel checks whether this is a valid stack address, i.e. if this comes from a request to extend the stack. If the address is above the stack top, and there are available pages in between the address and the top of the stack to be used then the kernel extends the stack for the process by that amount of pages. Otherwise, it is not a valid address, and the process gets segmentation fault and is killed by the kernel.
- If the address is within a valid memory region, then the kernel looks up the PTE associated with that address. If no entry is found, then the kernel allocates a new frame for this page and maps it in memory.
- If there is a PTE in the page table, the kernel checks for the PTE type. If the page containing the fault address is in memory, the kernel simply remaps this page to a frame in memory.
- If the page has been paged out, then the kernel tries to page it back in memory, which might need to yield to load it from the page file. See demand paging for more information on this.
- However if the page is currently being paged out, then the kernel is blocked until the page has finished paging out, and resumes the fault handling to page it back in.

After handling the fault, if the kernel successfully maps the page associated with the fault address the kernel will return back to the main event loop and let the process continue execution, otherwise the kernel terminates the process (potentially due to not enough memory for mapping).

Aside from page fault, If the process generates a different kind of fault (e.g., invalid instruction), the kernel just kills the process.

Demand paging



For demand paging, TrivialOS uses a file in the root directory called `pagefile`. This file only contains the pages that are paged out, and has no metadata to manage the entries and access for itself. TrivialOS keeps an in-memory bitmap (called `pageused` in the code) to store the bits associated with the indices in the pagefile to denote whether that page is currently in use in the page file or free. When the kernel finds a free entry in the `pagefile`, it updates the bit in the bitmap to true, and writes the content of the evicting frame to this location. On the other hand, if the kernel loads a page from the pagefile back to memory, it then unsets the bit in the bitmap to be false, indicating that the entry is free to be used in the `pagefile`. We favors storing the bitmap in memory over storing it in the `pagefile` itself is because (1) it is a simpler design, and we do not need to waste an extra page in the pagefile, (2) this avoids executing extra I/O operations read bitmaps from the `pagefile` just to find a free entry to page out to and also to update it back in the `pagefile`, which helps improve performance.

To find a frame to evict, we implemented the second-chance page-replacement algorithm. Each frame has a reference bit, which is set to true if the frame is in use (have a mapping to a virtual page). When there is no free frame to map a virtual page, we then choose to evict a frame and write it to the `pagefile` on disk, and use this frame to map the new page. To choose a frame, we loop through all allocated frames in the frame table and find the first frame that is not in use

(i.e. has a reference bit set to false). If a frame is in use, we set the reference bit to false and move on to the next. To assist with this, we added a `ref` field in the frame table entry data structure.

TrivialOS also supports pinning frames in memory, to prevent paging it out. We introduced a `pin` field to mark a frame pinned in memory, and the second-change policy will ignore this frame while trying to find a frame to evict. In the case that all the frames are pinned, the kernel cannot evict a frame to map the new virtual page, it then returns and terminates the process with the page fault due to not enough memory for mapping.

For paging out, the kernel finds a frame to evict. It first sets this frame to be pinned in memory. This prevents a race condition where we are trying to page this frame out but another process tries to page it in, which could corrupt the memory in the pagefile. The kernel then sets the type of the PTE of the virtual page mapped to this frame to be `PAGING_OUT`, to avoid other processes mapping this page while paging out the frame associated. Then the kernel looks for a free entry in the `pagefile` to write the page to disk. Once done, the kernel then sets the above mentioned PTE type to `PAGED_OUT`, and stores the pagefile index in the frame field of this PTE (since it is already paged out, there is no need to maintain the frame reference for this PTE), and also sets the bit for the `pagefile` entry to be true in the bitmap `pageused`.

For paging in, the kernel first needs a free frame to store the content read from the pagefile, either to an available frame or use an evicted frame. It then pins this frame in memory for the similar reason in paging out. It gets the `pagefile` index from the frame field of the PTE of the page that requests paging in. The kernel then reads the 4 KiB content of this entry in the `pagefile` and stores this in the frame found initially, unpins this frame from memory, unsets the bit for this `pagefile` entry in the bitmap `pageused` to be false and returns to the fault handling process. From here, the kernel then sets the PTE of this page to be `IN_MEM`, and updates the frame reference in the PTE, and allows the process causing the page fault to continue executing.

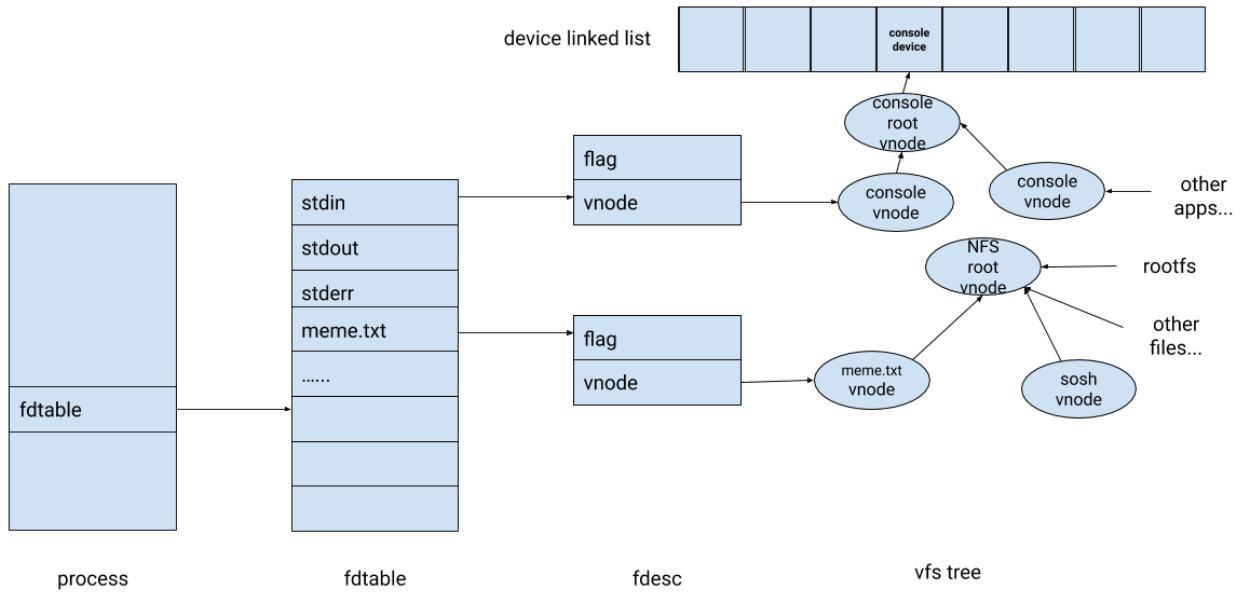
TrivialOS by default uses a 32 MiB file for the pagefile to store the memory content of evicted frames. We could also support up to 4 GiB pagefile (by changing the `#define` for `PAGEFILE_PAGES` to indicate the maximum number of entries the `pagefile` stores), since we only store bitmaps in memory which requires to store 1 MiB for the pagefile entries in memory, and the frame field in PTE is 20-bit long which is big enough to reference the pagefile index value.

If testing demand paging, please set your physical frame limit higher than 128 to ensure the kernel functions well, because we have a separate clock driver process and its memory is pinned.

I/O subsystem

Virtual File System (VFS)

TrivialOS employs a unified VFS abstraction, through which access to files and I/O devices are provided.



With VFS, TrivialOS can potentially support different file systems under the same interface. Each file system has a root vnode for the entire file system and a child vnode for each file. One can also create a hierarchy by having a directory vnode. Each vnode supports some or all of the following operations. The behavior is defined via function pointers so they can be easily changed and extended, even at runtime.

```
int (*vop_open)(vnode_t *object, char *pathname, int flags_from_open,
vnode_t **ret, coro_t me);
int (*vop_read)(vnode_t *file, uio_t *uio, process_t *proc, coro_t me);
int (*vop_write)(vnode_t *file, uio_t *uio, coro_t me);
int (*vop_pread)(vnode_t *file, uio_t *uio, coro_t me);
int (*vop_pwrite)(vnode_t *file, uio_t *uio, coro_t me);
int (*vop_close)(vnode_t *vnode, coro_t me);
int (*vop_stat)(vnode_t *vnode, char *pathname, sos_stat_t *stat, coro_t
me);
int (*vop_get_dirent)(vnode_t *vnode, int pos, char *name, size_t nbyte,
coro_t me);
```

TrivialOS has a `rootfs` file system, denoting the file system we use for `/` (root directory). The kernel calls `register_rootfs` function when booting up to register this. Each I/O device is also treated as a file system and its root vnode is registered with a given name through `register_device` function. For simplicity, devices are stored in a linked list, requiring `O(n)` lookup time to find the device with name. It is not our immediate concern to optimize this because users usually only have a small amount of devices. Optimization can be trivially done with either a HashMap or a Trie. But inter alia we only have a single device (console) now, and optimizing it actually makes it slower despite having a better time complexity.

`vfs_open` and `vfs_stat` calls `vfs_lookup` to find the required root vnode for the file system. `vfs_lookup` is implemented to first search device linked list and then fall back to `rootfs` if the required file system is not a device.

Network File System (NFS)

TrivialOS's `rootfs` is registered to be NFS. TrivialOS's NFS implementation uses a flattened design - there are no directories, and every file lives under root, i.e. every file vnode is a direct child of the root vnode.

TrivialOS delegates actual operations to libnfs[3]'s asynchronous NFSv3 functions, while providing a synchronous vnode operation interface. Before calling libnfs functions, we store the current syscall coroutine to local kernel stack and yield. NFS callback function is called from watchdog IRQ handlers, which then resumes the blocked coroutine.

As mentioned earlier, `read` and `write` syscalls map the user space buffer to kernel space (in fact, every physical frame is already mapped to TrivialOS address space by the frame table). Page by page, TrivialOS ensures the buffer page is paged in, pins the frame, and calls libnfs's `nfs_read_async/nfs_pread_async/nfs_write_async/nfs_pwrite_async` for that page. While calling libnfs functions with a contiguous kernel buffer that covers all required pages (instead of page by page) could provide a higher throughput, that requires we have enough physical frames to map the entire buffer in at the same time. With the page-by-page solution, it is simple and requires less physical memory.

As mentioned above, for other syscalls other than `read` and `write`, we just copy-in/copy-out user space string/struct because their performance is not critical.

The provided libnfs has a bug that if the connection is idle for a while, picotcp receives a `POLLHUP` and libnfs tries to reconnect but always fails and the NFS connection dies. This is due to libnfs's auto-reconnect mechanism using `dup2` syscall, which is not available in picotcp[4]'s socket descriptor. We fixed this by removing the `dup2` logic in libnfs reconnect routine.

Console Device

UDP serial port is implemented as a multi-writer-single-reader I/O device and registered in the aforementioned manner. If no process is reading the console, user input is stored in a buffer. We use a circular buffer (called `rollingarray_t` in code) and a queue of newline character indexes to optimize its time and memory usage. If there is no newline in the queue, `console_read` blocks and yields, waiting for the serial port's read handler to resume it.

File Descriptor

As shown in the VFS diagram above, a process keeps a file descriptor table. This is implemented as an array of pointers. We store a vnode pointer and a permission flag (`O_RDONLY/O_WRONLY/O_RDWR`) in each file descriptor. We reject `read` operations on `O_WRONLY` file descriptors and `write` operations on `O_RDONLY` file descriptors. File permission is checked in

[3] https://github.com/sahlberg/libnfs
[4] https://github.com/tass-belgium/picotcp

`fdesc_open` for all file systems, instead of in underlying VFS, because we need to bypass readable permission checks to load an execute-only ELF file (in fact, we had to comment out permission checking code in libnfs because we could not find a way to load exec-only files). Opening files without the corresponding file permissions will be denied.

File descriptor table has size `OPEN_MAX` (set to 128), and entries are added to the lowest available fd. This is done with a linear scan with `O(n)` time complexity. This does not matter too much because `OPEN_MAX` is small. We could potentially optimize it with a free-fd priority queue but it would take more memory and we think it is not worth it since `open` is not a critical code path - programs do not spend much time in `open`, but in `read` and `write` instead. When a program first starts, the `console` device is automatically opened and set to fd 1 (`stdout`) and 2 (`stderr`). The first file (or device) that the process opens will have fd 0, and the second one has fd 3.

Process management

Process Control Block (PCB)

Apart from the aforementioned `fdtable`, there are a few other attributes we store in a PCB.

```
struct process {
    pid_t pid;                  /* process id */
    unsigned stime;             /* starting timestamp */
    char command[N_NAME];       /* process filename */
    char state;                 /* process running state (PROC_FREE,
PROC_RUNNING, PROC_BLOCKED, PROC_TO_BE_KILLED, PROC_CREATING) */

    ut_t *tcb_ut;               /* untyped memory for TCB */
    seL4_CPtr tcb;              /* cptr for TCB */
    ut_t *vspace_ut;            /* untyped memory for vspace */
    seL4_CPtr vspace;           /* cptr for vspace */
    ut_t *sched_context_ut;     /* untyped memory for scheduling context
*/
    seL4_CPtr sched_context;    /* cptr for scheduling context */
    seL4_CPtr kernel_ep;        /* cptr for badged ep in kernel cspace
*/

    cspace_t cspace;            /* process cspace */
    addrspace_t *addrspace;     /* address space */
    fdtable_t fdt;              /* file descriptor table */

    runqueue_t *exit_blocked;   /* coroutines to resume upon this
process quitting */
    void (*kill_hook)(void *data); /* hook function to run before killing
this process */
```

```
    void *kill_hook_data;          /* data to pass into kill_hook function
*/

    coro_t paging_coro;            /* if blocked by nfs paging, the paging
coroutine */
};
```

ELF loading

TrivialOS loads all ELFs (including clock driver and first sosh) from NFS. It first reads in the first page to check the ELF headers, then loads each of the segments. While loading in, region permission is enforced. We also implemented `elf_find_vsyscall` to find the vsyscall address without loading everything to kernel memory.

Process ID (PID) management

We have the following configuration for PID management.

```
/* max num of simultaneously-running processes
 * this can't be too large since we'll run out of sos stack space for
coroutine. Increase stack space if you want to increase this. */
#define MAX_PROCS 20
/* max possible PID value.
   once we reach max, we start filling from 1 again (with a simple
non-collision hashtable) */
#define MAX_PID 65535
```

PID is allocated on a circular basis (see `rollingid_t`, this is also used in our clock driver to assign timeout IDs). It maintains an in-memory bitmap for whether a specific ID is free or not and scans the bitmap in a circular way to find the next available ID (roll over to 1 once it reaches `MAX_PID` 65535).

In addition to the circular ID allocation, we also need a way to ensure that we keep the old process around for a while after it is killed in case another process calls `process_wait` on it, not knowing it is already killed. To solve this, we use an el-cheapo hash table, where collision is not allowed. We have `MAX_PROCS` (20) buckets. Bucket `i` holds processes whose `pid % MAX_PROCS = i`. Each bucket stores the current process, and the last process in this bucket that gets killed (this is implemented as two simple arrays). Every time we get a new PID from our circular bitmap, we check if the bucket is free - if it is not, we just skip this PID and get another one.

Combining these two together, our PID assignment algorithm allows `O(1)` lookup and `O(n)` insertion.

Process state

Each process is associated with one of the following states:

- **`PROC_FREE`**, this PCB is not in use (either process quitted or was never created)
- **`PROC_RUNNING`**, this process is running
- **`PROC_BLOCKED`**, this process is in the middle of a syscall or a page fault (we set this at beginning of syscall/fault handler and revert it back when they finish)
- **`PROC_TO_BE_KILLED`**, this process is in the middle of a syscall or page fault but will be killed after it finishes
- **`PROC_CREATING`**, this process is initializing

Killing process

`process_delete` syscall is used to kill a process. If the target process is the caller (process quitting itself), we can directly kill it. If the target process is a different process, we check its state. If it is `PROC_RUNNING`, we can directly kill it. If it is `PROC_BLOCKED`, it is blocked in a coroutine and we cannot immediately kill it because we cannot free its address space (e.g. NFS might be writing to its buffer). Instead, we set it to `PROC_TO_BE_KILLED`. Before the blocked syscall/fault handler returns, it checks its process state: kill the process if it is `PROC_TO_BE_KILLED` or otherwise reset it back to `PROC_RUNNING`.

In other words, we are waiting for the blocked coroutine to finish before killing it. This works great for killing unblocked processes and processes blocked by NFS, but it does not work well for killing processes blocked by serial input, `usleep`, or `process_wait` syscall. For serial input, we need to wait till the user enters a new line; for `usleep` we need to wait till the timeout is triggered; for `process_wait` we need to wait till another process gets killed and that might not happen in a while.

To solve this, we added a function pointer called kill hook. Kill hook function, if set, is called when we kill a blocked process after setting it to `PROC_TO_BE_KILLED`. For example, for usleep, before yielding, kill hook is set to a function that removes the timeout and directly resumes coroutine, and the hook is removed after the coroutine resumes. Similar kill hooks are added to `process_wait` and `console_read`. For `nfs_read` and fault handler (which also blocks on NFS for paging), we do not set and cannot set kill hooks because (1) the NFS operation operates on user space memory and (2) these coroutines are usually resumed shortly without long waiting.

By design, we made `process_delete` a blocking syscall: it only returns after the target process is successfully killed (with our kill hook design, this happens almost instantly). This is purely a design choice. Another possible design is to directly return after setting the target process state to `PROC_TO_BE_KILLED`.

Waiting for process

As shown in the PCB struct definition above, we have an `exit_blocked` run queue (linked list of coroutines) for each process that is called upon the process being killed. In addition, we also have a global run queue that is called when any process is killed.

When `process_wait` syscall is called, it checks the target process bucket. If the target process is currently running, we set kill hook, add current coroutine to target process's `exit_blocked` run queue, and yield. If the target process is the last killed process in the bucket, we can directly return, knowing the process was killed before `process_wait` is called. If the target process is not found in either of them, we just return an error (either the pid is wrong or it quited a long time ago).

Clock driver

TrivialOS has a tickless clock driver running in a separate process. It uses timer A for timeouts and timer E for timestamp. Internally it uses a min-heap as a priority queue for upcoming user timers (sorted according to expiry timestamp). When a timer IRQ is triggered, it checks from the beginning of the priority queue whether this user timer is up and calls the corresponding callback. It then sets timer A to the next expiry timestamp. We have average time complexity `O(1)` to insert a new user timer and `O(log n)` to remove the next user timer from the heap.

TrivialOS's clock driver also runs in a separate process, loaded from a separate ELF binary from NFS. This process could be killed by the user but is automatically re-spawned. An uncached memory page for the SoC's timer device is mapped to its address space. A separate seL4 endpoint and reply object is created in its cspace.

This turns out to be tricky with our kernel being single-threaded and event-based. The kernel blocks on the clock driver and the clock driver can also block on the kernel. It is easy for the entire kernel to hang. There are two solutions to this: (1) make the clock driver entirely asynchronous; (2) clock driver never blocks waiting for the kernel

We went forward with (2), because it is, to the best of our understanding, not very feasible to make it asynchronous. We cannot yield the coroutine waiting for the clock driver to be able to receive. seL4 NBSend intentionally does not provide a boolean response, while Call blocks our only kernel thread. We could potentially use NBSend and implement a TCP-like protocol, with Acks, but it becomes hard to know when to retransmit since we do not have the timer working. We could potentially rely on the watchdog timer, but it sounds way too complex and also introduces huge delays to the latency-critical clock driver.

We ended up not calling any syscall from the clock driver and pin the entire clock driver program in memory (~20 frames) to avoid page faults from the clock driver. Apart from the additional timer endpoint, we also implemented 2 new syscalls that can only be called from the clock driver: `timer_callback` and `timer_ack` to be handled in the main event loop.

Better solutions for milestones

Milestone 1: A timer driver

✓ Implementing a tickless timer:

- We have implemented this in our clock driver code by implementing a priority queue. See clock driver for more information on this.

Milestone 2: System call interface

✓ Having a clear framework for handling all blocking within SOS in a consistent way:

- By using coroutine, we ensure to have all blocking mechanisms in place (e.g., make all required nfs calls synchronous)

Milestone 3: A virtual memory manager

✓ Providing as much heap and stack as the address space can provide

- By placing our heap after the last elf segments of the process, it can grow down to higher address space as much as it could, until it either reaches the stack or other mmap regions
- For the stack, by placing at a higher address space, it can then grow up in the opposite direction to the heap, as much as possible until it reaches the heap or other mmap regions. We could also potentially move our stack to an even higher address space, providing more space for the stack
- The above steps would ensure the heap and stack can grow at the best possible way given there is room between them.

✓ Probing page table efficiently - minimal control flow checks in the critical path, and minimal levels traversed

- We are mirroring the underlying page table structure of aarch64 so it does not need too much overhead to convert between two different data structures.
- We have written our code to be clean and efficient with bitwise calculation optimizations.
- We are optimizing for the common cases - not for the errors. For example, in the fault handler, we just map valid pages in without checking permission. If there is a permission issue, we get a second page fault where we kill the process.

✓ Not using SOS's malloc to allocate SOS's page tables

- We have achieved this by fitting our page tables within 4 KiB memory and use the untyped memory instead. See page table for more information on this.

✓ Clear SOS-internal abstractions for tracking ranges of virtual memory for applications

- We have achieved this abstraction level with our address space and memory region data structures. See address space for more information on this.

✓ Minimizing the size of page table entries (e.g. only contain the equivalent of a PTE and a cptr)

- We managed to fit our page table entry in a single `seL4_Word` (which is 8 bytes long). This is the smallest size for our page table entry we can make, without affecting the page table. See page table for more information on this.

✓ Enforcing read-only permissions as specified in the ELF file or API calls

- We have achieved this with our permission setting on the memory region. See address space for more information on this.

✓ Deferring actual mapping of pages until they are used (mapping on the fault rather than always mapping on the initial request)

- For our memory region, we only created the region struct to store the ranges for this region and not allocating memory for it. On page fault, the kernel then allocates memory for this region. See address space for more information on this.

Milestone 4: File system

✓ No virtual memory management (mapping) in the system call path

- We have achieved this for our `read` and `write` syscalls (See syscall interface and NFS for more information on this). However if the page we are accessing in the syscall call path is paged out then we have to page it in and map it in memory.

✓ Support for multiple outstanding requests to the NFS server, i.e. attempts to overlap I/O to hide latency and increase throughput

- We have achieved this with our execution event model. If there is a read (or other syscalls interacting with the NFS), the process then yields waiting for the operation to finish, which then resumes the process. When the process is yielded, execution flow can be resumed for other processes that might also be blocked and yielded, hence providing more throughput as no processes wait on a single process blocked by the syscall.

✓ Only pinning in main memory the pages associated with current active I/O

- With our memory management design, we support pinning pages in memory, which is called by the syscalls performing I/O operations on the current page of the buffer. As discussed in the read and write syscalls, we only read/write page by page, so we simply need to pin that 1 page in memory instead of the whole buffer requested from the syscall. See NFS for more information on this.

✓ Avoiding double buffering

- We have achieved this by mapping the frames directly in our kernel space for our read and write syscalls. We did not implement this feature for other syscalls for simplicity reasons. See syscall interface for more information on this.

Milestone 5: Demand paging

✓ Not increasing the page table entry size when implementing demand paging

- We have achieved this by making our frame reference field in the PTE to be 20 bits long, which we use to store the pagefile index (which can be a large value up to 2^{20}) and not need to modify the PTE in the page table. See page table for more information on this.

✕ Avoiding paging out read-only pages that are already in the page file

- We did not implement this due to time constraints. But since there are some extra bits free in the PTE, we could potentially use them to mark if this page is read-only and already paged out in the pagefile, then we would not need to page it out again. However we then need to modify our bitmaps to mark the entry is from a read-only page which can potentially be used again, and decide to avoid using this entry in the pagefield as much as possible (maybe similar second-chance policy for demand paging).

✓ In theory, supporting large page files (e.g. 2-4 GiB)

- We have achieved this by only using 20 bits in the PTE to reference the pagefile index, which can go up to 2^{20}. Also we have our bitmaps in memory to handle the entries in the pagefile which is relatively small (for a 4GiB pagefile it is 1MiB large). See demand paging for more information on this.

✕ Avoiding keeping the entire free list of free page-file space in memory

- We did not implement this due to design choice. If we did not have our bitmap in memory then we had to place it in the `pagefile` itself. This would hinder our performance since we needed to perform extra I/O operations to access and also update this bitmap, along with paging in and out of the `pagefile`. Also, bitmap comes with a small memory overhead, so there is no issue having it in memory. See demand paging for more information on this.

Milestone 6: Process management & loading

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

- We have achieved this by keeping the last process killed in the one of the buckets mentioned in PID management, so that the process that waits on the killed process can check and simply return without actually waiting. See PID management and waiting for process for more information on this.

✕ Lazy load the executable or data from the file

- We did not implement this feature due to time constraints. However, our current loading process for an ELF file is modular, which we could potentially refactor to support this feature. We could also have a flag in our memory region to say we need to load this region later when we reference it.

✕ Mapping read-only ELF segments read-only, and not paging them, but re-reading them from the ELF file

- We have already mapped the segments in memory as read-only but we did not implement this feature to avoid paging them out and paging them in from the ELF file due to time constraints. But we could implement this by utilizing the free bits in the PTE, to mark this is a read-only page from the ELF (not from an mmap region). So when there is a need to page out, we can skip this page directly. And for paging in, we can simply load from the ELF file instead of paging in from the `pagefile`.

Built-in programs

For your convenience, we provide the following userland applications together with TrivialOS:

- **`sosh`**, a trivial shell program, modified from the provided sosh
- **`original_sosh`**, the unmodified original sosh in starter code
- **`tty_test`**, a simple Hello World
- **`mem_test`**, test stack, heap (`malloc` via `brk`), heap (`malloc` via `mmap`), and direct `mmap` syscall
- **`rdonly_test`**, test writing to read-only `.code` segment
- **`nx_test`**, test jumping to executable and non-executable `mmap`-ed regions separately
- **`tick`**, print out Hello World every 1 second
- **`sleep10`**, sleeps for 10 seconds and die
- **`forkbomb`**, starts 2 new `forkbomb` processes (keep trying if failed) then sleeps for 3 seconds and quit