Simple Operating System Documentation

Tong Wu - z3417604
Lewis Lee - z3415068

October 28, 2014

Contents

1 Assumptions & Known Issues

- Maximum processes name length is 32 characters long
- Error checking with processes is not robust
- Demand paging uses FIFO, not Second Chance algorithm. Not sure where to obtain the referenced bit for a frame.
- Swapping does not work with multiple processes running, we start getting cap faults at random places.

2 Clock Driver

2.1 Interface

The clock driver is implemented using the EPIT1 and EPIT2 timers and implements the following functions;

- **int** start_timer(**seL4_CPtr** interrupt_ep1, **seL4_CPtr** interrupt_ep2)
  This is called to initialize both EPIT1 and EPIT2 timers. It takes in two **seL4_CPtr**s and sets them as interrupt endpoints for EPIT1 and EPIT2 respectively. If the timers are already running (ie start_timer had been called before), then they will be stopped and restarted. **CLOCK_R_OK** will be returned on success, otherwise **CLOCK_R_FAIL** will be returned.
- **uint32_t** register_timer(**uint64_t** delay, **timer_callback_t** callback, **void** *data)
  This is called to set a timer which will expire after a given delay while calling a given callback function. If timers are not initialized, **CLOCK_R_UINT** will be returned, **CLOCK_R_FAIL** will be returned if the maximum number of timers is reached (this is currently 50, this can be changed in clock.h). **CLOCK_R_OK** will be returned on success.
- **int** remove_timer(**uint32_t** id)
  This is an internal function which when called, removes the timer from the priority queue.
- **int** EPIT1_interrupt(**void**)
  This function should normally be called when there is a timer interrupt for **EPIT1** (used for heartbeats). When called, it will update the current time and reset the underflow flag (so the timer will continue to interrupt). It then acknowledges the interrupt handler. If **EPIT1** is not initialized, **CLOCK_R_UINT** will be returned, otherwise **CLOCK_R_OK** will be returned.
- **int** EPIT2_interrupt(**void**)
  This function should normally be called when there is a timer interrupt for **EPIT2** (used for callback timers). When called, it will call the callback function for the current

timer in queue, then removes the timer from queue using **remove_timer()**. If **EPIT2** is not initialized, **CLOCK_R_UINT** will be returned, otherwise **CLOCK_R_OK** will be returned.

- **timestamp_t** time_stamp(**void**)
  This function prints out the current time after **start_timer()** was called. It checks the underflow flag in case it missed an interrupt and adds additional time if necessary. Then it adds the time elapsed since the last interrupt to the current time and returns the current time in microseconds (64-bit unsigned integer). If **EPIT1** is not initialized, **CLOCK_R_UINT** will be returned.

- **int** stop_timer(**void**)
  This function can be called to stop all current callback timers and to do a soft reset on both **EPIT1** and **EPIT2**. **CLOCK_R_UINT** will be returned if the timers are not initialized, otherwise **CLOCK_R_OK** will be returned.

2.2 Configurables

- **PRESCALAR_EPIT1**: Controls the prescale value for EPIT1, currently set to 6.
- **PRESCALAR_EPIT2**: Controls the prescale value for EPIT2, Currently set to 66.
- **TIMER_TICK_INTERVAL**: Controls the amount of timer ticks between each interrupt for EPIT1, currently set to 1100000.
- **MAX_TIMERS**: Controls the maximum amount of timers we can have registered at once, currently set to 50.

The current values for PRESCALAR_EPIT1 and TIMER_TICK_INTERVAL means that we get interrupts at 10Hz (100ms intervals) and due to the large TIMER_TICK_INTERVAL, we can get up to micro second accuracy since we can determine time passed between heart beats by reading the counter values directly.

2.3 Operation

In SOS, the clock driver is initialized on boot before the entering the main syscall loop. The EPIT1 interrupt is used for the 100ms heart beat and EPIT2 is used for timers (used to implement sleep).

3 Frame Table

3.1 Interface

- **int** frametable_init(**void**)
  Initialises the frame table
- **int** frame_alloc(**seL4_Word** *vaddr)
  Maps a page to the vaddr given, returns a paddr
- **int** frame_free(**seL4_Word** vaddr)
  Frees the paddr given
- **int** frametable_add_page_cap(**seL4_CPtr** cap, **seL4_Word** sos_vaddr)
  Add a page cap to the corresponding frame of frametable
- **int** frametable_map_app_page(**pid_t** pid, **seL4_CPtr** cap, **seL4_Word** sos_vaddr, **seL4_Word** app_vaddr)
  Maps an app allocated page
- **seL4_CPtr** frame_get_cap(**seL4_Word** vaddr)
  Returns the page cap associated with a frame

3.2 Structure

The frametable is simply an contiguous array of frames. The frametable itself resides at a fixed offset past DMA region.
Each frame contains the following data;

- **seL4_CPtr** cptr
- **seL4_CPtr** page_cptr
- **seL4_Word** app_vaddr
- **pid_t** pid
  The process that owns the frame
- **int** next_swap_frame
  The next frame in the swap queue for demand paging
- **int** prev_swap_frame
  The previous frame in the swap queue for demand paging
- **int** swap_send_count
- **int** spare

3.3 Operation

Our frametable keeps track of allocated frames (physical memory which has been retyped and mapped). It maps one to one in a region of virtual memory past the DMA region. It keeps track of which addrspace it belongs to (and related information) as well as caps for freeing and mapping memory.

4 Page Table/Address Space

4.1 Interface

- **struct addrspace** *addrspace_create(**void**)
  Create a new address space with first page level initialized
- **int** sos_map_page_table(**struct addrspace** *as, **seL4_Word** vaddr)
  Maps a second level table into the given page directory (1st level)
- **int** sos_map_page(**pid_t** pid, **seL4_Word** vaddr, **seL4_ARM_VMAttributes** attr, **int** swappable)
  Map a physical address to a page
- **int** sos_define_region(**struct addrspace** *as, **seL4_CapRights** permissions, **int** start, **int** end)
  Defines a usable region for the application
- **int** check_region(**struct addrspace** *as, **seL4_Word** vaddr, **seL4_CapRights** *permissions)
  Check whether a region is valid and extract its permissions
- **seL4_CPtr** app_vaddr_frame_cap(**struct addrspace** *as, **seL4_Word** app_vaddr)
  Return the frame cap for the corresponding application virtual address
- **seL4_Word** get_sos_vaddr(**addrspace** *as, **seL4_Word** app_vaddr)
  Given an address space and application address, returns the corresponding sos virtual address
- **void** set_cur_as(**struct addrspace** *as)
  Set the cur_as (a global variable identifying the address space of the last process that sent a syscall)
- **uint32_t** expand_virtual_heap(**uint32_t** newbrk)
  Expands the virtual heap for the application, used by sys_brk
- **void** addrspace_destroy(**struct addrspace** *as)
  Destroy the given address space
- **int** sos_map_elf_page(**struct addrspace** *as, **seL4_Word** kvpage, **seL4_Word** vpage)
  sed for mapping elf pages, as kdst needs to be mapped.

4.2 Structure

4.3 Operation

Our two level pagetable keeps track of the relevant memory information related to a process. This includes things such as whether a page is in file or memory and where in memory it is located. There is also another table which keeps track of pagetable details (paddr and caps) so that items can be freed correctly. Each pagetable also keeps a region list in which applications can access memory when a page fault is triggered. The key function of addrspace is to map pages between our OS and applications.

5 System Calls

5.1 Overview

In the application side, the system call functions determines the SOS system call number and sends data through IPC Message Registers and blocks until a reply comes from SOS. For sending and receiving large amounts of data, the libsos sends a pointer to an empty array, which will get translated, page by page by SOS, and filled. When a reply is received, libsos will retrieve data from the message registers and for large data, SOS would have filled it in already.
When SOS receives a system call, it retrieves the system call number from message register 0 and uses it to index an array of function pointers pointing to the corresponding hand system call function. When SOS is finished handling the system call, it replies and the application will resume.

5.2 Interface

5.2.1 Application System Calls

- **fildes_t** sos_sys_open(**const char** *path, **int** flags)
  Opens console if path is "console" otherwise, tries to open the given path from the file system. flags can be **O_RDONLY**, **O_WRONLY** or **O_RDWR**
- **int** sos_sys_close(**fildes_t** file)
  Closes the file given by the file handle
- **int** sos_sys_read(**fildes_t** file, **char** *buf, **size_t** nbyte)
  Reads the file given by the file handle, using the function pointer to the correct read function (eg. tty or nfs) defined when the file was opened. tries to read nbytes but can return with less than that amount.
- **int** sos_sys_write(**fildes_t** file, **const char** *buf, **size_t** nbyte)
  Write to the file given by the file handle, using the function pointer to the correct write function defined when the file was opened.
- **int** sos_getdirent(**int** pos, **char** *name, **size_t** nbyte)
  Get directory entry given by pos from nfs.
- **int** sos_stat(**const char** *path, **sos_stat_t** *buf)
  Returns information about file "path".
- **pid_t** sos_process_create(**const char** *path)
  Creates a new process running the executable "path".
- **int** sos_process_delete(**pid_t** pid)
  Deletes the process given by "pid".
- **pid_t** sos_my_id(**void**)
  Returns the pid of the current process.

- **int** sos_process_status(**sos_process_t** *processes, **unsigned** max)
  Returns the status of currently running processes. Max specifies the maximum number of process statuses to return.

- **pid_t** sos_process_wait(**pid_t** pid)
  Waits for the process given by pid to finish before resuming. If pid is -1 then wait for anything to finish before resuming.

- **int64_t** sos_sys_time_stamp(**void**)
  Returns the current time since boot in microseconds.

- **void** sos_sys_usleep(**int** msec)
  Waits for "msec" milliseconds before resuming.resuming

- **long** sys_brk(**va_list** ap)
  Increases the heap region to a new brk point determined by "ap".

5.2.2 Internal Functions

All handle syscall functions in SOS have void return type and take one argument, "mypid" (PID of the caller application extracted from the incoming badge). Functions are called by indexing an array of function pointers to the handle syscall function corresponding to the syscall number.
Currently implemented syscall numbers are;

1. SOS_SYS_OPEN
2. SOS_SYS_CLOSE
3. SOS_SYS_READ
4. SOS_SYS_WRITE
5. SOS_SYS_GETDIRENT
6. SOS_SYS_STAT
7. SOS_SYS_PROCESS_CREATE
8. SOS_SYS_PROCESS_DELETE
9. SOS_SYS_MYID
10. SOS_SYS_PROCESS_STATUS
11. SOS_SYS_PROCESS_WAIT
12. SOS_SYS_TIMESTAMP
13. SOS_SYS_USLEEP

14. SOS_SYS_BRK
15. SOS_TTY_OPEN
16. SOS_TTY_CLOSE
17. SOS_TTY_WRITE
18. SOS_TTY_READ
19. SOS_SYS_CREATE

5.3 Coroutines

We use coroutines sparingly and it was only first introduced to handle demand paging. Prior to that all our system calls relies purely on blocking and replying to the application through the method provided by seL4. For coroutines, we are using an open source library called "picoro". The reason we chose picoro is because it provides us with a minimal number of functions we need to use coroutines effectively and its code base is extremely small (less than 100 lines), thus easy to understand. Picoro provides the following functions;

- **coro** coroutine(**void** *fun(**void** *arg))
  Create a new coroutine that executes the given function when resumed for the first time
- **int** resumable(**coro** c)
  Returns whether the coroutine is resumable
- **void** *resume(**coro** c, **void** *arg)
  Resumes the given coroutine
- **void** *yield(**void** *arg)
  Yield from the coroutine we are currently in

Picoro has been added as a library, "libpicoro" to our code and some slight modifications were made to the picoro code in order for correct operation with SOS. Each coroutine has a stack of 16KB, used to store local variables when we switch coroutines. Each process has its own coroutine, created during each system call.

5.4 Operation

Below is a diagram of how our system calls generally operate. Blue lines indicate flow within SOS, red lines within an application and black lines indicate IPC. The large rounded rectangles are coroutines and everything inside belongs to that coroutine. Smaller rounded rectangles indicate a coroutine operation.

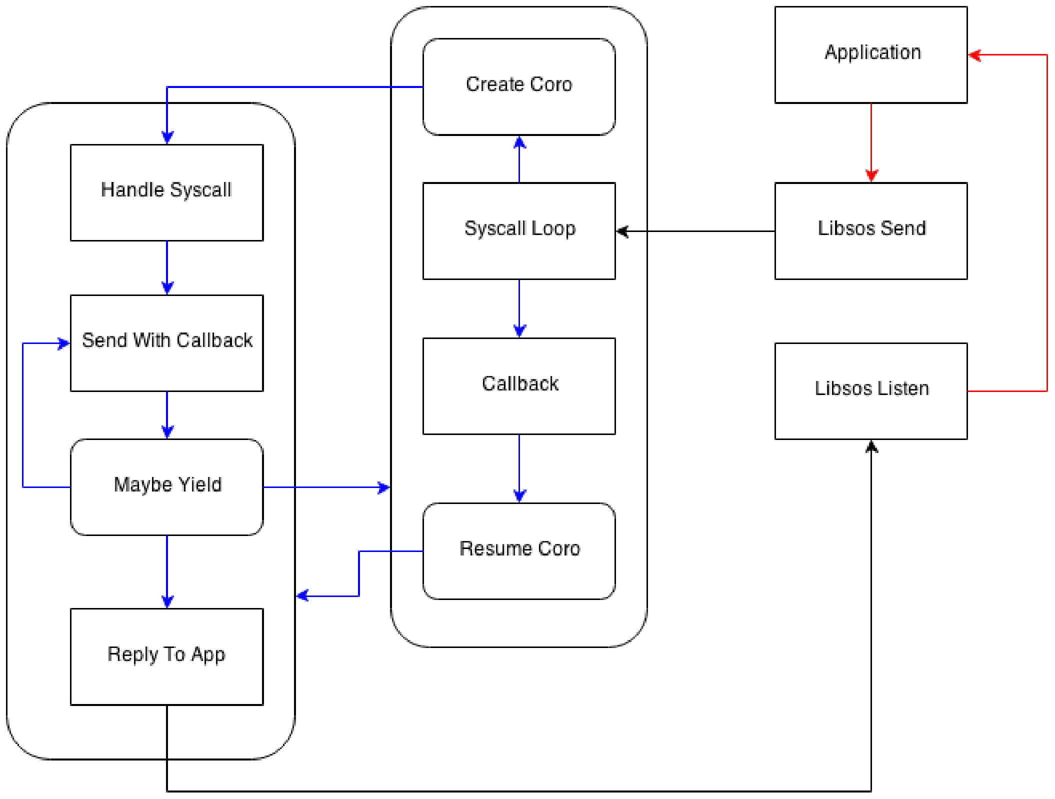

Figure 1: System Calls Operation

Stepping through;

1. Application does a system call
2. Libsos sends relevant information to SOS via IPC
3. SOS receives the message and determines the PID of the caller via its badge.
4. SOS then creates and executes a new coroutine assigned to the caller process.
5. If the system call involves being able to receive callbacks before continuing (eg. reading from nfs), we yield back to the main syscall loop. This way we avoid blocking the kernel.
6. When the callback comes through, we resume back to the point we left off in the handle syscall relating the PID of the original caller.

7. When no more operations need to be done, and the last call back comes through, we reply to the application

8. Libsos, depending on the system call, retrieve information received and return back to the main application.

6 File System

6.1 Interface

Handle syscall functions;

- **void** syscall_sys_create(**pid_t** mypid)
  Sends NFS request to create a file with Read/Write permissions and yields to main.
- **void** syscall_sys_open(**pid_t** mypid)
  Calls nfs_lookup with path and yields to main.
- **void** syscall_sys_close(**pid_t** mypid)
  Removes the file descriptor from the file table and yields to main.
- **void** sys_read_routine(**struct read_write_data** *data)
  Sends NFS requests one at a time and yields back to main. When this resumes, it repeats the process until no more reads are needed.
- **void** syscall_sys_read(**pid_t** mypid)
  Gets information from message registers and checks permissions with file descriptor table. If the permissions for the file is write only, fail and reply back to caller. Otherwise, call sys_read_routine.
- **void** sys_write_routine(**struct read_write_data** *data)
  Sends NFS requests up to MAX_IN_FLIGHT (currently 4), then yields to the main syscall loop. When this next resumes, it will repeat the process until it has no more to send.
- **void** syscall_sys_write(**pid_t** mypid)
  Gets information from message registers and checks permissions with the file descriptor table. If the permissions for the file is read only, fail and reply back to the caller. Otherwise, call sys_write_routine.
- **void** syscall_sys_getdirent(**pid_t** mypid)
  Sets up tokens, calls nfs_readdir and yields back to main.
- **void** syscall_sys_stat(**pid_t** mypid)
  Calls nfs_lookup with the path and yields back to main.

Callbacks;

- **void** nfs_lookup_reply(**uintptr_t** token, **enum nfs_stat** status, **fhandle_t** *fh, **fattr_t** *fattr)
  Callback function for the sys_stat syscall. Replies to original caller.
- **void** nfs_create_reply(**uintptr_t** token, **enum nfs_stat** status, **fhandle_t** *fh, **fattr_t** *fattr)
  Callback function for the sys_create syscall. Replies to original caller.

- **void** nfs_open_reply(**uintptr_t** token, **enum nfs_stat** status, **fhandle_t** *fh, **fattr_t** *fattr)
  Callback for the sys_open syscall. Replies to original caller.

- **void** nfs_readdir_reply(**uintptr_t** token, **enum nfs_stat** status, **int** num_files, **char** *file_names[], **nfscookie_t** nfscookie)
  Callback for the sys_getdirent syscall. Replies to original caller.

- **void** nfs_read_reply(**uintptr_t** token, **enum nfs_stat** status, **fattr_t** *fattr, **int** count, **void** *data)
  Callback for the sys_read syscall. If not the last callback (ie, there are still things left to be read), then resume the coroutine corresponding to the pid of the original caller. Otherwise, reply to the original caller.

- **void** nfs_write_reply(**uintptr_t** token, **enum nfs_stat** status, **fattr_t** *fattr, **int** count)
  Callback for the sys_write syscall. If not the last callback (ie, the entire string has not yet been written), then resume the coroutine corresponding to the pid of the original caller. Otherwise, reply to the original caller.

6.2 Operation

Here is a diagram of the write operation. Stepping through;

Figure 2: File System Write Operation

1. Application does a system call, passing through the address of the data array it wants to write via IPC

2. SOS receives the message and determines the PID of the caller via its badge.

3. SOS then creates and executes a new coroutine assigned to the caller process.

4. SOS copies 1024 bytes of data at a time from the address given by the application (translated into sos virtual address), into a temporary write buffer. The nfs_write is called and the pid along with other information is passed through the token. After we send the maximum amount of NFS packets inflight at a time, we yield.

5. When the last callback comes through, we resume back to handle syscall and repeat step 4.

6. When no more operations need to be done, and the last call back comes through, we reply to the application with the total number of bytes written.

The read operation is similar, however instead of copying data from the application in handle syscall, we copy read data from events to the application.

7 Demand Paging

7.1 Interface

- **int** frame_swap_in(**int** file_index, **seL4_Word** sos_vaddr)
  Swap in the frame that is in the swap file at position pointed to by file index

- **int** frame_swap_out(**seL4_Word** *vaddr)
  Swap out the given frame, appends to swap file

- **void** swap_file_create(**void**)
  Creates the swap file on NFS

- **void** swap_write_reply(**uintptr_t** token, **enum nfs_stat** status, **fattr_t** *fattr, **int** count)
  Callback function for swapping out

- **void** swap_read_reply(**uintptr_t** token, **enum nfs_stat** status, **fattr_t** *fattr, **int** count, **void** *data)
  Callback function for swapping in

- **void** swap_create_reply(**uintptr_t** token, **enum nfs_stat** status, **fhandle_t** *fh, **fattr_t** *fattr)
  Callback function for creating the swap file

7.2 Operation

- When a frame is allocated via vm fault, it is added to the frame queue. This is done by two variables in each frame which point to the next frame and previous frame in the queue.

- For swapping out, we use a stack of indexes, as the order in which they return when freed is inconsequential. These indexes are stored in the page table entry of pages that are swapped out. When a page is in file, it has a infile bit set. On vm_faults this bit is checked first and the frame swapped in if in file.

- Swapping means we needed a method of blocking, as the nfs operation must complete before we can continue on in the program (to keep memory safe). We used coroutines to carry this out, which simply yield back to the main loop after an nfs call, and return to the function on the nfs callback.

8 Processes

8.1 Interface

- **uint32_t** init_procs(**void**)
  Resets proccess array to default
- **pid_t** create_process(**char** *app_name, **uint32_t** priority, **seL4_CPtr** fault_ep)
  Finds the elf file given be "app_name" that is contained within the cpio archive and creates a new process using that elf file.
- **uint32_t** destroy_process(**pid_t** pid)
  Given a PID, destroys its address space, file descriptor table and frees any caps. If there were processes waiting for the process to be killed to finish, then resume them.
- **uint32_t** wait_for_process(**pid_t** mypid, **pid_t** pid, **seL4_CPtr** my_reply_cap)
  Adds mypid to target pid's waiting list, which will resume mypid when pid is destroyed. If target pid is -1, then add mypid to the global wait list, which will resume mypid when any process is destroyed.
- **struct addrspace** *get_proc_addr_space(**pid_t** pid)
  Gets the pointer to the address space of pid.
- **fdtable** get_proc_fd_table(**pid_t** pid)
  Gets the file descriptor table of pid
- **cspace_t** *get_proc_cspace(**pid_t** pid)
  Gets the cspace of this process.
- **coro** get_proc_coro(**pid_t** pid)
  Gets the coroutine of this process.
- **seL4_CPtr** get_proc_ipc_reply_cap(**pid_t** pid)
  Gets the reply cap of the process.
- **uint32_t** get_proc_stime(**pid_t** pid)
  Gets the time since boot that the process was started at.
- **uint32_t** get_proc_size(**pid_t** pid)
  Gets the size of the process in pages.
- **char** * get_proc_name(**pid_t** pid)
  Gets the name of the process.
- **char** get_proc_taken(**pid_t** pid)
  Returns whether or not the pid number is in use.
- **coro** create_coro(**pid_t** pid, **void** *fun(**void** *arg))
  Creates a new coroutine unique to the process.

- **void** set_proc_ipc_reply_cap(**pid_t** pid, **seL4_CPtr** cap)
  Sets the reply cap of the process.

8.2 Structure

The process struct;

- **seL4_Word** tcb_addr
  TCB cap, stored for destroy purpose
- **seL4_TCB** tcb_cap
  TCB cap, stored for destroy purpose
- **struct addrspace** *as
  The address space for this process, including the page table and region list
- **fdtable** fd_table
  File descriptor table for this process, used for file systems
- **seL4_Word** ipc_buffer_addr
  IPC buffer cap, stored for destroy purpose
- **seL4_CPtr** ipc_buffer_cap
  IPC buffer cap, stored for destroy purpose
- **cspace_t** *croot
  The croot for this process
- **seL4_CPtr** ipc_reply_cap
  The reply cap for this process
- **pid_t** wait_list[MAX_PROC]
  List of processes waiting for this process to finish
- **uint32_t** num_waits
  Num of other processes waiting on this process to finish
- **coro** proc_coro
  The process coroutine, used for handle syscalls while not blocking kernel
- **uint32_t** stime
  Time after boot at which the process was started
- **uint32_t** size
  Size of the application in pages
- **char** name[N_NANME]
  Name of the application, currently maximum 32 characters

- **uint32_t** name_length
  Length of the application name
- **char** taken
  When taken is not 0 then pid is in use.

There is an array "proc_array" that contains a bunch of these structures and the index represents the PID.

8.3 Operation

- The processes infrastructure is initialized in main before the first process is created. This ensures that all the processes in the process list is set to their default values.
- When create is called, it assigns the process the lowest available pid number and a badge is created based on a offset of the pid number. An address space including page table is created for this process and regions for heap and stack are defined. Then its cspace is defined and an IPC buffer is created, the fault endpoint is minted to the process and a TCB object is created. Now, we either load the elf file from nfs or if it is the first process, load from cpio. Then we map in the stack, ipc buffer and start the process.
- When destroy is called we reply to all processes waiting on the current process and also all processes in the global wait list. Then we destroy the file descriptor table, unmap the ipc buffer cap. The free the ipc buffer using ut_free. Then we free the ipc_reply_cap, and destroy the cspace. Now we free the TCB object and destroy the address space. Finally, make the pid available for a new process again.

9 ELF Loading

9.1 Interface

- **int** elf_load_from_nfs(**struct addrspace** * as, **const char** *name, **pid_t** pid)
  Loads elf file given by "name" into the address space of process given by "pid".
- **void** elf_lookup_reply(**uintptr_t** token, **enum nfs_stat** status, **fhandle_t** *fh, **fattr_t** *fattr)
  Callback for nfs_lookup
- **void** elf_read_reply(**uintptr_t** token, **enum nfs_stat** status, **fattr_t** *fattr, **int** count, **void** *data)
  Callback for nfs_read
- **void** elf_segment_read_reply(**uintptr_t** token, **enum nfs_stat** status, **fattr_t** *fattr, **int** count, **void** *data)
  Callback for nfs_read, used in load_segment_into_vspace2
- **static int** load_segment_into_vspace2(**struct elf_nfs_data** *nfs_data, **char** *src, **unsigned long** segment_size, **unsigned long** file_size, **unsigned long** dst, **unsigned long** permissions)
  Loads the elf file obtained from nfs to address space
- **int** elf_load2(**struct elf_nfs_data** *nfs_data, **char** *elf_file)
  Loads the elf file obtained from nfs to address space, calls load_segment_into_vspace2

9.2 Operation

- Elf loading from the NFS server was done by initially reading in the elf file header. From here, the points in the file which had to be read were done so by nfs, and then copied into the corresponding frame allocated. This was done one packet at a time. Due to the nfs operation we had to use coroutines, as with demand paging.

10 Old File System Benchmarks

We benchmarked the filesystem for milestone 5 which had a different implementation than what we currently have (The old system was difficult to adapt to demand paging, the new system is slower with writes at about 1Mb/s, because we are only sending 4 packets at a time with 1024 bytes per packet).
All benchmarking was done without the ethernet dongle and our mean speed at 30 packets in flight, 1284 bytes per packet is 4Mb/s for write and almost 17Mb/s for read. I had to artificially slow down reading, any more and we would get spurious interrupts slowing things down again.

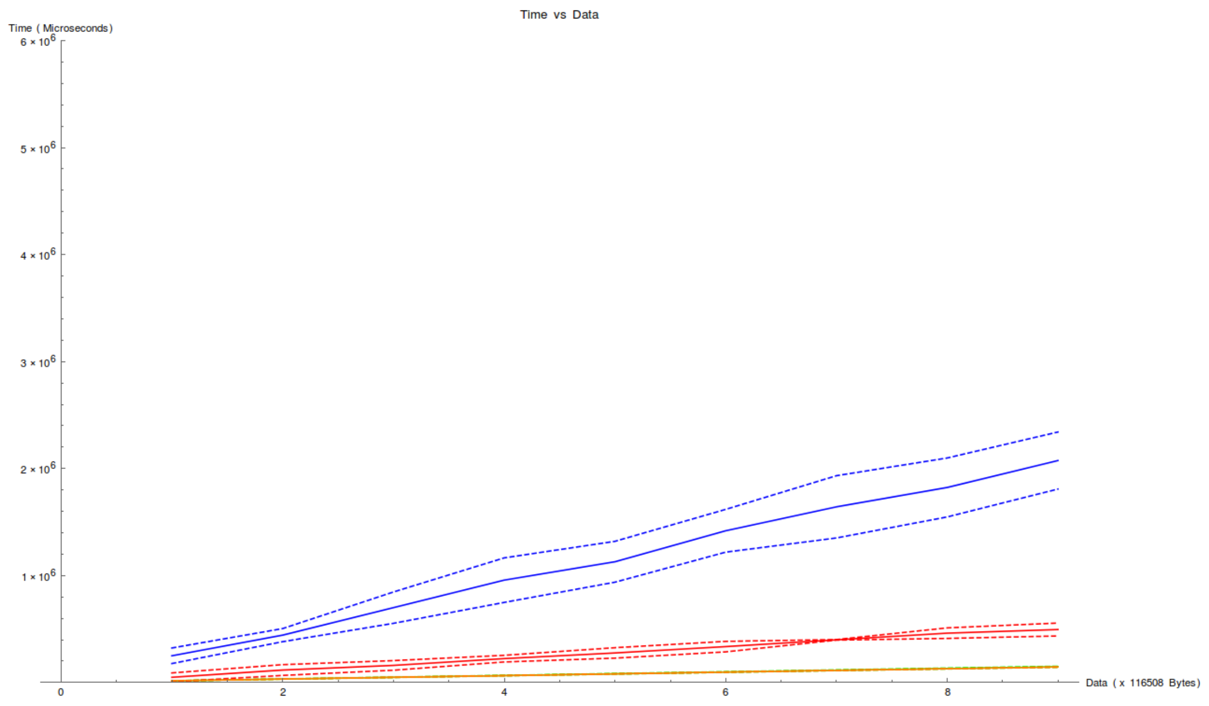

Figure 3: The Blue line is write, and the red line is read. The green and orange lines are overhead for write and read respectively.

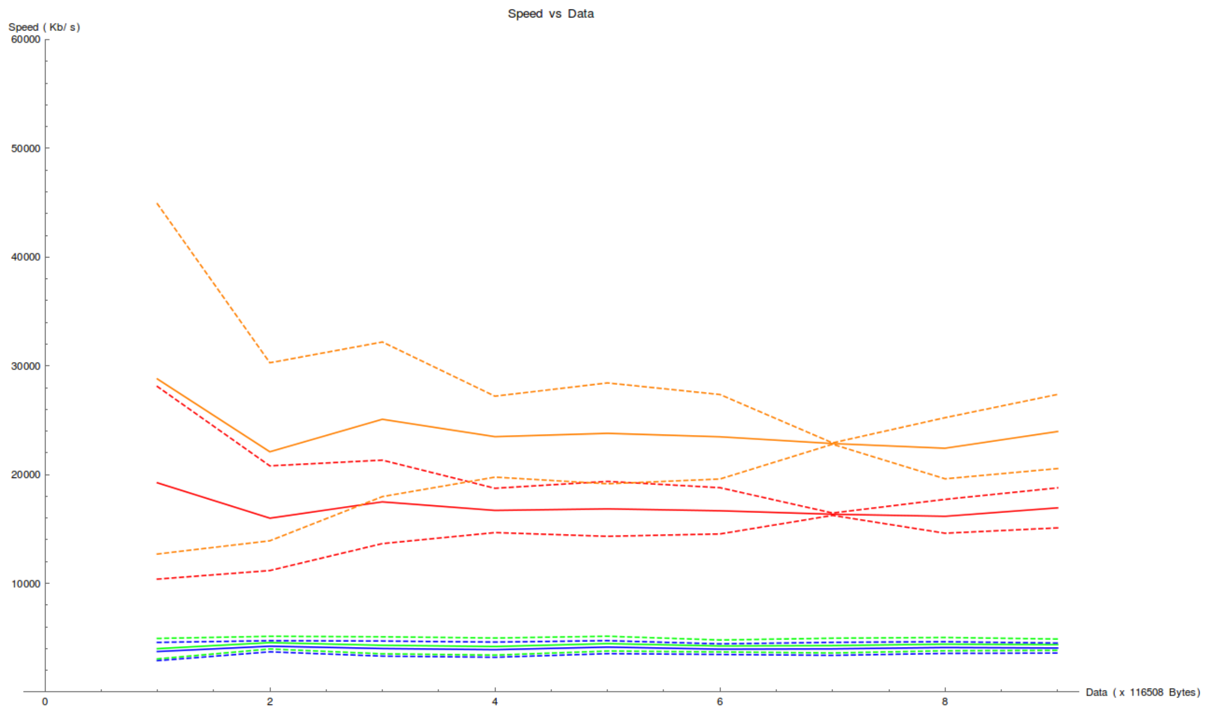

Figure 4: The Blue line is write, and the red line is read. The green and orange lines are "ideal" speeds (minus overhead) for write and read respectively.

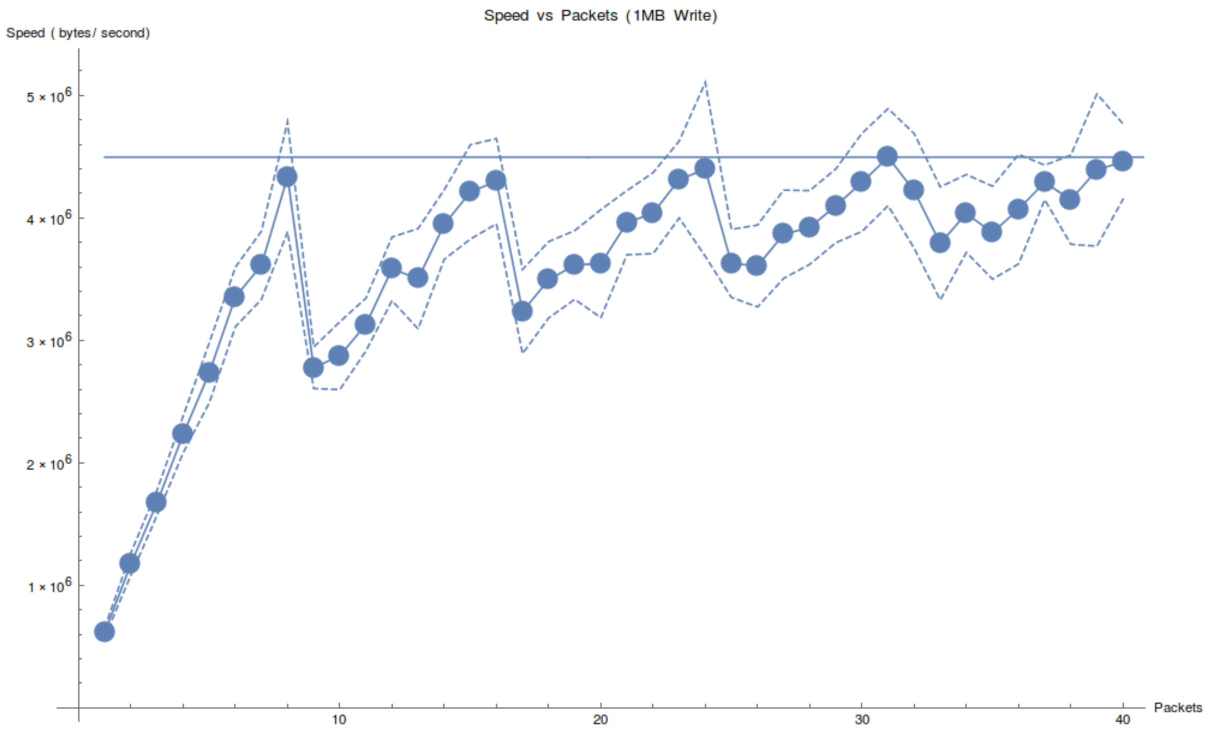

Figure 5: The is a graph of speed of writing a 1MB file vs maximum number of packets inflight, interestingly, it follows a zig zag pattern, it seems that the optimal is 30 packets in flight