Design Doc

David Fan & Tianyu Liu

Introduction

The design doc is organized into the following sections:
overview,
code reading answers,
coding style,
process management,
file data management,
system calls,
fork/execv,
waitpid/exit,
plan of action. The discussion for error handling and synchronization issues are included inside the descriptions for each section. The major issues we're facing right now are:
Testing and debugging: we're not sure how could we efficiently test that the system functions we wrote are correct. We're also concerned about debugging userland programs.
Benchmarking: It's not clear how could we efficiently benchmark our scheduler's performance.
Orphan processes: we're not sure if our implementation for collecting orphan processes will introduce unexpected overhead.
Synchronization: we're still not sure if our usage for locks and cvs, especially those in waitpid/exit will cause deadlock.

Overview

We plan to tackle the assignment by introducing/modifying three main APIs: process, file object and file table.

For process: we'll make the current interface for proc much more robust. Specifically, we'll add struct fields and functions to allow pid management, parent/child management, file management, exit_code status and timing for scheduler and synchronization. We'll also add helper functions to support the interface and more convenient destroy, copy and execution (for fork).

For file object: recognizing the need to keep track of opened files, offsets and other file-related metadata, we decide to add one more struct file_obj. File_obj will keep track of an opened file's vnode, mode, path, and offset.

For file table: we add one specific field, file_table, to the proc struct because we need to keep track of all opened files for each specific process. File_table is an array that associate each index with an opened file. It supports the following operations: create, destroy, add file_obj, and remove file_obj. This is also how file object and process fit with each other.

With the defined interface, our implementation for open, write, read, close, dup2 and lseek will be mainly manipulation on file_obj and file_table. Our implementation for chdir, getcwd, getpid will be using the refined proc interface. Waitpid and exit will mainly be working with the proc->exit_code field, handling child/parent pids, and synchronizing relevant locks and cvs. And execv and fork will be modeled on thread_fork and runprogram, while using the helper functions for process management.

For scheduling, we plan to start with a priority queue based on fastest running time, and explore a mechanism similar to completely fair scheduling if we have time.

Code Reading

1. 177, 'E', 'L', 'F'
2. UIO_USERISPACE is used when the memory blocks being referenced are executable, UIO_USERSPACE is used when they only contain data. UIO_SYSSPACE is used for kernel-specific data, since it uses memmove directly rather than copyin/copyout, which has to perform various checks.
3. The data is not being loaded into the uio struct, but rather into the address pointed to by vaddr, which the user process should already have allocated.
4. The number of references to the file needs to be decremented, or, if no longer used, the vnode needs to be reclaimed.
5. mips_usermode and asm_usermode are both responsible for switching the processor into usermode. mips_usermode in machine independent.
6. kern/vm/copyinout.c; common/libc/string/memmove.c. When copying data to/from userland, the operating system needs to check that the user process is allowed to access the relevant memory locations, and it needs to be able to store information about whether or not the copy failed.
7. userptr_t is defined to prevent accidentally mixing up user pointers and kernel pointers. const void * is used for kernel pointers.
8. #define EX_SYS 8
9. 4 bytes. In syscall.c:141, we advance 4 bytes to skip an instruction.
10. We should handle the cases when user faults, rather than just panic.
11. Additional arguments will be fetched from the user-level stack, starting at sp+16.
12. It provides a common entry point for all system calls. All user-side wrapper functions will just trigger exception according to SYS_CALL. We can then refer to a specific system call by a number in the v0 register.
13. Line 85 of syscalls-mips.S, which executes the instruction "syscall"
14. The first four arguments will be in register a0-3, and 64-bit arguments will be in aligned registers. arg0 of lseek() will be in a0, arg1 in registers a2 and a2 with register a1 skipped for alignment. The final argument will be found in the user level stack at sp+16. The 64-bit return value will be stored across registers v0 and v1.

Coding style

Pointers are “type *name”
Use underscores: int variable_name; int function_name(int foo) {} (first { stays on the same line)
Use “} else {“
Use “if (condition) {”
Declare all variables at top of function

Process Management

struct proc

```
struct proc {
        char *p_name;                    /* Name of this process */
        struct spinlock p_lock;          /* Lock for this structure */
        struct threadarray p_threads;    /* Threads in this process */

        struct addrspace *p_addrspace; /* virtual address space */
        struct vnode *p_cwd;             /* current working directory */

        struct filetable *p_filetable;    /* process’s filetable */

        pid_t pid;                        /* pid */
        pid_t parent_pid;                /* parent pid*/
```

```
        int exitcode;                          /* exit code used in _exit and waitpid */
        bool exited;                           /* indicate if we have already exited or not */

        struct cv *waitpid_cv;                 /* cv for watipid/exit logic */
};
```

We also declare a global variable proc_table that keeps tracks of all current processes.
We will modify and add functions as follow:

proc_table_bootstrap():
- will be called in thread_bootstrap()
- malloc proc_table[MAX_THREAD]
- start orphanage process

proc_create():
- get the next available pid and assign it to the new process
- assign the exite_status to NULL
- set the parent_pid and child_pids to 0
- set timer to 0
- add the new process to the global proc_table
- call file_table_create(see below) and attach it to proc

proc_destroy(*proc):
- remove proc from proc_table
- free memory

proc_run(data1, data2): helper function that gets forked by fork(). Responsible for running the new user process. Set tf->tf_v0 to 0; retval to new thread's pid. Return to usermode.

proc_get_exit_code(pid): return the exit_code for process pid, the functions needs to be synchronized using a lock

proc_set_exit_code(status): set curproc->exit_code to status, the functions needs to be synchronized using a lock

File data Management

struct file_obj

This keeps track of where the owning process is in the file.

A new file_obj should be created whenever open is called successfully. If a process has "foo.txt" open, a subsequent call to open("foo.txt",...) should result in a new file_obj being created.

The file_obj's should be global across the entire process. E.g. the file descriptor 3 should map to the same file_obj regardless of what thread is using it. If a process opens a file, and then forks, both descendants should be referencing the same file_obj, i.e. the side effects of read (incrementing position) should be reflected for both processes.

```
struct file_obj{
        struct vnode *file_node;      /* the vnode of the file */
        struct lock *file_lock;     /* lock of file, used for syncing file access */
        off_t pos;                  /* offset of the file, used in lseek */
```

```
        int file_refcount;          /* offset of the file, used in lseek */
        int file_mode;              /* file mode, initialized by open */
};

struct filetable
{
        struct file_obj *filetable_files[OPEN_MAX]; /* a table of file_objects */
        struct lock *filetable_lock;                /* used to sync filetable updates */
};
```

Functions for file_table and file_obj

We need the following helper functions for efficient file_table management:

file_table_create():

- malloc file_table object
- initialize STDIN, STDOUT and STDERR
- return file_table object

file_table_destroy(file_obj file _table):** destroy the file_table and free memory

file_table_add(file_obj file _table, file_obj* file):** Add file to file_table, and return the associated file descriptor

file_table_remove(file_obj file_table, file_obj* file):** Remove file to file_table, return 0

file_table_get(file_obj file_table,int file_descriptor)**: Return file_obj associated with file_descriptor in file_table

System Calls

open

initialized with path, flags, mode, and pos to 0; create a new file_obj struct with that information. Pass in the information to vfs_open. Use the result to update file_obj struct. Get a new file descriptor number (next_fd) and increment next_fd. Return the file descriptor number. Delete and free the file_obj if any error occurs. Calling vfs_open needs to be synchronized?

read

checks:

buf is not NULL
file descriptor actually exists (return EBADF if not)
file_obj has read mode set

Use the filetable and file descriptor number to find the associated file_obj. Use the file_obj’s data, esp. position, to create a uio object. This uio object holds information about where the data should be read to, how many bytes to read, and where to start reading. Call VOP_READ with the uio object and vnode associated with file_obj to perform the actual read.

VOP_READ needs to be synced, in case 2 processes read on the same file

write
Basically the same as read(), but we change VOP_READ to VOP_WRITE.

lseek

checks:

whence is one of SEEK_SET, SEEK_CUR, SEEK_END (fail: return EINVAL)
seek position >= 0 (fail: return EINVAL)
file_obj exists (fail: EBADF)
file_obj is seekable (fail: ESPIPE)

Load the file_obj using filetable and then update the offset value based on the argument. SEEK_CUR: add pos to offset, SEEK_SET: set the offset to pos, SEEK_END: keep reading until we can't read anymore.

close

checks:

file_obj exists
[EIO handled by vfs_close?]

Lookup the file_obj struct, pass its vnode to vfs_close, set the location in the process's filetable to NULL. This should not affect the underlying file_obj struct, only the owning process's filetable (processes should still have access to the file .

dup2

checks:

file descriptor actually exists
(EMFILE and ENFILE should never occur due to dynamic file_table size - we will run out of memory before either error occurs)

look up the file_obj of *newfd* in filetable and create a clone new_file_clone of the object. We then index into filetable[*oldfd*] and replace the object with new_file_clone.

chdir
Call vfs_chdir and return check error. vfs_chdir handles all error code (?)

getcwd
Create a uio object that has uio_iov[0] point to buf, uio_resid to buflen, call vfs_getcwd, passing the uio. vfs_getcwd handles all error (?)

getpid
Return the process->pid of the current process

kill_curthread
Should exit the faulting thread by calling thread_exit, rather than just panic.

Fork/Execv

fork

- Call proc_create to create a new process
- Initialize the new process based on curproc's info. Helper functions include: filetable_copy and as_copy. Note the filetable_copy only creates a shallow copy with incremented refcount
- Add current pid as parent_pid for the new process
- Set retval to the new pid
- Call thread_fork with the new process and use run_forked_proc as wrapper
- Return 0

functions for proc

We keep track of all the processes in an array, proc_table for easier management of a process's pid, parent/child relationship, exitstatus, etc.

enter_forked_process(struct trapframe *tf):
Called in run_forked_proc. This function will set up the trapframe for user's program and warp to user mode using mips_usermode. Specifically, it will set v0, a3 and advance the argument by 4.

run_forked_proc(void *tf, unsigned long junk):
This function is directly called in thread_fork for interface compatibility. In the function, we just simply copies over the trapframe on to kernel stack and hand all the job to enter_forked_process.

Handling orphan child process: Specifically, we propose a mechanism to deal with child and parent's exit problem. Since waitpid has to be called to fully clean up a process, a child process will run into a case when its parent exits without calling waitpid. In this case, the child process never gets cleaned up. To deal with the problem, we create a master 'orphanage' process where all abandoned children got attach to if their parent exits first. The orphanage process will simply call waitpid for all it's children to make sure they all exits cleanly. (Similar to Linux's approach)
* In our final implementation, we decide to adopt an easier approach: if a child exit without a parent waiting, it will just destroy itself.

execv

checks:

- Do sanity checks:
  - The arguments are not NULL(EFAULT)
  - The program actually exist (EINVAL)
  - The pointer addresses are actually valid
- Copy argc/argv/(env) into kernel memory:
  - First, we copy the file name into kernel memory.
  - Iterate through the user's pointer once to figure out nargs, check each argument is valid
  - Keep track of the length for the argv arguments in order to save malloc space
  - Malloc kernel space to keep the strings
  - Copy strings over using copyinstr, based on the lengths we recorded
- Open the program file
- Create new address space for the process
- load executable
- Initialize stack
- Copy argc/argv to the proper user address:
  - First, we decrease stackptr by nargs+1 for the argv pointers, record the argument base
  - Iterate through our cloned strings in kernel, decrease stackptr by strlen and copy each string on stack

    - Assign the new string's start to the correct index of argument base
    - NULL terminate the argv pointer
- Free argc/argv/(env) in kernel memory.
- Run executable with nargs and user mode argv(enter_new_process).

waitpid/_exit

_exit will not terminate the process until its parent has called waitpid. It will then proceed to handle cleanup.

_exit

- Acquire the global proc_table_lock. We'll have to modify proc_table.
- Orphan all the children by setting their parent_pid to INVALID_PID
    - if we run into any child that has already exited, just clean them by by calling proc_destroy
- Check the parents of curproc:
    - If our parent has already exited, we just simply call proc_destroy and release proc_table_lock
    - If our parent has not exited, don't destroy ourselves. Instead, we set our exitstatus, exited, and use our cv and signal our parent. Our parent should handle cleaning up for us.

waitpid

check:

- pid must be valid (ESRCH)
- must not wait on anything that's not a child (ECHILD)
- flag must be valid(EINVAL)
- returncode can't be NULL(EFAULT)

implementation:

- lock proc_table_lock and look up pid
- Error if we found anything other than a child
- Check if child has already exited
    - If so, immediately release the lock and return
    - If not, wait on child's condition variable using proc_table_lock
- After child wake up the parent proc, get the exitcode, copy it out to retval and destroy child proc

Scheduling

Priority queue based on fastest running time. Using a timer to keep track of how long each process takes from running to interrupt/block. Keep two priority queues, each sorted by increasing running time. Only run process in one queue and use another queue for scheduling, so that we're guaranteed that each process could run at least once.

Alternative method (Shortest Job First variant; sort of like Completely Fair Scheduler):
For each thread, keep track of (1) amount of time that the thread has been running, i.e. **total active time**; (2) **total time** since the thread started, i.e. total active time + total inactive time. Threads that are not blocking (state = READY) are sorted by total active time / total time in **ascending** order on the thread queue. All threads are forced to yield after their allowed time slice has elapsed (e.g. 10ms). The result should be that threads get to run for as close to

A thread that spends a lot of time waiting (e.g. a shell waiting for user input that later runs a compiler) may end up getting too high a priority when it cashes in on its "debt," starving other threads. A brand new thread will get a disproportionate amount of time initially as other threads' active time / total time approaches the lower "fair value," but this may be desirable for giving new thread an initial burst of time.

Plan of Action

[brackets denote stretch goals]

	David Fan	Tianyu Liu
Sat, Feb 21	Setup and test environment	Setup and test environment
Sun, Feb 22	Travelling to Quora, [open, read, write]	Implement helper functions for file_table
Mon, Feb 23	At Quora	Implement helper functions for file_table
Tue, Feb 24	open, read, write, [lseek, close]	Implement helper function for process management
Wed, Feb 25	lseek, close, dup2, [fork]	Implement helper function for process management, fork
Thu, Feb 26	Physics and CS181 psets, fork	fork
Fri, Feb 27	At Microsoft	waitpid and exit
Sat, Feb 28	Travelling from Microsoft, [execv]	waitpid and exit
Sun, Mar 1	scheduling	execv
Mon, Mar 2	scheduling and debug	execv debug
Tue, Mar 3	Debug Debug Debug Debug	Debug Debug Debug Debug
Wed, Mar 4	Debug Birthday :) Debug	Debug Debug Debug Debug
Thu, Mar 5	Debug Debug Debug Debug	Debug Debug Debug Debug
Fri, Mar 6	Final check and submit, Par-tay!	Final check and submit