CSE 451 Section 4

Project 2 Design Considerations

Overview

- 4 major sections:
- File management
- File related system calls
  - open, close, read, write
- Process management
- Process related system calls
  - getpid, fork, exec, waitpid, _exit

File Management

- Need a per-process data structure to organize files – a **file table**
- Things to consider:
  - What data structure will you use?
  - What will data structure entries hold?
  - How will it be synchronized?
- Hint: open files are represented by unique integer called a **file descriptor**

File System Calls – open

- int open(const char **filename*, int *flags*)
- Takes in a filename of file to open
- Flags determine read/write permissions and create/truncate details – refer to man pages
- Returns a non-negative file descriptor on success, -1 on failure
- Note: ignore the optional mode

File System Calls – open

- File descriptors 0, 1, and 2 are reserved for stdin, stdout, and stderr respectively
- Attached to the console – named “:con”
- OS/161 provides the virtual file system (vfs). It is a layer of abstraction between the os and file system
    - You only need to interact through the vfs
    - Carefully read through the files in kern/vfs
    - Carefully read through **vnode** code – abstract representation of a file provided by OS/161

File System Calls – close

- `int close(int fd)`
- Takes in the file descriptor of the file to close.
- Things to consider:
  - Multiple processes may reference the same file.

File System Calls – read and write

- `int read(int *fd*, void **buf*, size_t *buflen*)`
- `int write(int *fd*, const void **buf*, size_t *nbytes*)`
- Read and write to the file given by file descriptor
- Depend on the use of `uio` and `iovec` structs to do the actually reading and writing
  - Look through loadelf.c to see how to use `uio` and `iovec`
  - `uio` structs represents a user or kernel space buffer
  - `iovec` structs are used for keeping track of I/O data in the kernel

Process Management

- Need a way to keep track of processes running on your machine
- Processes are identified by a unique integer called the process id (**pid**)
- Things to consider:
  - What data structure will you use?
  - What will data structure entries hold?
    - Hint: address space, file tables, etc.
  - How will pids be uniquely assigned?
  - How will it be synchronized?

Process System Calls – fork

- `pid_t fork(void)`
- Create a new process & thread, identical to the caller
- Child returns 0 and the parent returns the child's pid
- Things to consider:
  - How to copy/duplicate process related state
  - How to make child return 0 and behave exactly like the parent
    - Check out `mips_usermode()` and `enter_forked_process()`
  - When a process makes a system call, where how does it know where to return?
    - It saves a return address on the trapframe
    - Trapframe needs to be copied!

Process System Calls – exec

- `int execv(const char *program, char **args)`
- Replaces the currently executing program with a newly loaded program image
- program: name of program to be run
- args: array of 0-terminated strings. The array itself should be terminated by a NULL pointer

Process System Calls – exec

- `execv()` is quite similar to `runprogram()` in `syscall/runprogram.c`.
- Remember to test running the shell after exec works!
- Most difficult part is copying in user arguments correctly.
  - User passes in pointers to the arguments – need to `copyin` both the pointers and strings.
  - Then correctly format and `copyout` the arguments onto the process's stack.
    - Need to adjust pointers so they point to the copied strings
    - Remember to word align pointers!
  - Look at `vm/copyinout.c`

Process System Calls – exec

- Exec should set up the process' stack to look like this example of passing in 2 arguments "ls foo"

800	
799	∅
798	o
797	o
796	f
795	[padding]
794	∅
793	s
792	l
791	∅
790	∅
789	∅
788	∅ [null-terminate]
787	argv[1]
786	argv[1]
785	argv[1]
784	argv[1] = 796
783	argv[0]
782	argv[0]
781	argv[0]
780	argv[0] = 792 = stackptr

Process System Calls – waitpid

- `pid_t waitpid(pid_t pid, int *status, int options)`
- Wait for the process specified by *pid* to exit
- Returns pid of process waiting on
- Status: return parameter for exit status
- Closely tied to pid management and synchronization
- Things to consider:
  - How can you make a parent wait for a child? What happens if a child tries to wait for its parent?
  - You may need to add data to `stuct proc` to support this

Process System Calls – _exit

- `void _exit(int exitcode)`
- Causes the current thread to exit
- Also closely tied to pid management and synchronization
- Things to consider:
  - What are resources we need to free?
  - Do we always free all resources?
  - When do we free the process itself?
  - What about the exit code?
  - Don’t forget `kill_curthread()`

General Advice

- Remember to check if `kmalloc` fails!
- Read syscall man pages and pay careful attention to the many errors that can be thrown
- Errors should be handled gracefully – <u>do not</u> crash the OS
- You may need to increase your system's memory (again) in order for fork and exec to work

References

- Slides / Tutorial pages from Harvard:
  - http://www.eecs.harvard.edu/~margo/cs161/resources/sections/2013-MMM-ASST2.pdf
  - http://www.eecs.harvard.edu/~margo/cs161/resources/sections/2013-mxw-a2.pdf