A2 Design Considerations

CS161, Spring 2014

http://goo.gl/lzxych

Agenda

1. processes
2. file descriptors
3. fork
4. waitpid & exit
5. exec
6. scheduler
7. suggestions for testing
8. lessons learned from Kenny

Design Document

- These notes contain lots of open-ended questions to get you think about your design.
- Be sure to address these questions in your design document.

Processes

- a process is an address space & threads
- our recommendation: only implement single-threaded processes
  - feel free to implement multi-threaded processes...

Processes

- What per-process state might you want?
  -
- What per-thread state might you want?
  -
- How is the **first** process created? When?

Processes

- What per-process state might you want?
  - pid, address space, fd table, cwd, ...
- What per-thread state might you want?
  - execution state, user stack pointer, …
- How is the **first** process created? When?

Processes - Thread States

- What are the possible thread states?
- TIP: Draw a state transition diagram on how a thread can go from one state to the next.

pid

- process ids
- can pids be recycled?
- how do you allocate & free them?
- how do you prevent multiple processes from having the same pid?
- is there a limit to pids?
- for waitpid(pid): given the pid, how do you get to the process with that pid?

bootstrapping

- where in code should things be bootstrapped?
  - grep for “bootstrap”
- what kinds of things might need bootstrapping?
  - e.g. pid allocation data structures
  - depends on your implementation

file descriptors

- open(), close(), read(), write(), lseek(), dup2(), chdir(), getcwd()
  - see man pages in the man/ directory for a full specification of these system calls
- a **file descriptor** represents an index into a table of file descriptors (file descriptor table)
- All processes have the standard fds open by default:
  - stdin (0), stdout (1), stderr(2)
  - How do you initialize these file descriptors?
  - What kernel objects do these correspond to?

a note about VFS

- virtual file system
  - abstraction between OS and file-like systems
- API that file systems implement
  - allow you to treat all file systems in the same way by working with vnodes, vfs_*, VOP_* operations
  - VOP_*, vfs_* operations will delegate to the actual implementation for that file system (think object-oriented programming)
  - example: vfs_open() -> sfs_open() for sfs objects
- you will be working at the level of vnodes for ASST2

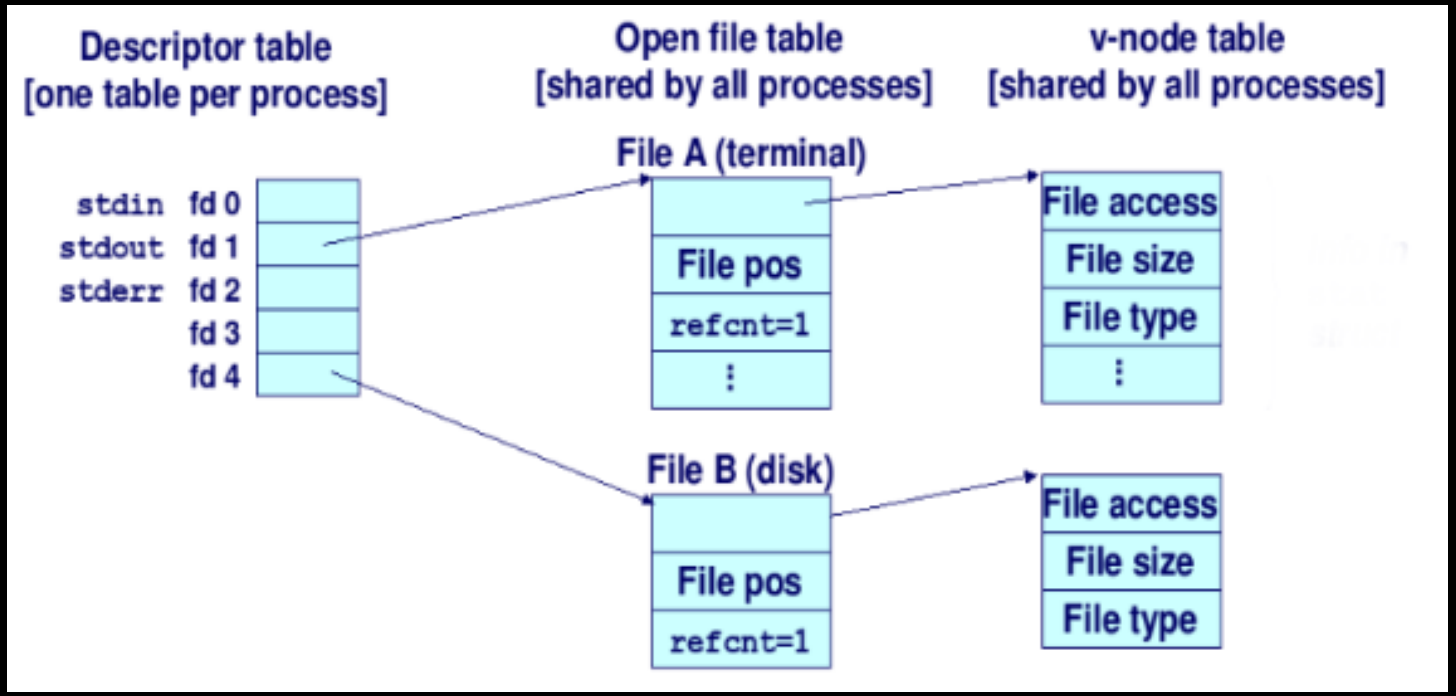
file descriptors - file descriptor table

- Who owns the fd table abstraction?
- What happens to a fd table when a process forks or exits?
- What happens to the fd table and seek position when:
  - you open() on the same file twice, and seek on one of them?
  - you seek on a file descriptor returned by dup2()?
  - you call fork, and the child seeks on a file descriptor?

file descriptors - file descriptor table

- How should you assign new file descriptors?
- Should there be a limit on file descriptors?
- Each file descriptor has an associated file offset. What happens if you open the same file twice?
- How should you convert file descriptors to more meaningful information? How are open files represented in the kernel?
  - vnode, part of the VFS (virtual file system layer)

file descriptors - file descriptor table



file descriptors - details

- lseek()
  - what if you seek beyond end of file? is that legal?
  - VOP_TRYSEEK
  - how do you handle 64-bit offset values?
- read()/write()
  - look at struct uio
- close()
  - a kernel object might be referenced by multiple fds
  - what happens if we close() a file descriptor returned by dup2()? when should the underlying kernel object actually be freed
  - TIP: use reference counting

exit() & waitpid()

- What process state can be cleaned up on exit()? on waitpid()?
- When does a process and thread *actually* get destroyed?
  - A thread/process can’t destroy itself...

exit() & waitpid()

- Who can wait for our exit status?
  - the parent
- How does the parent collect the exit status?
- waitpid() should fail it the pid doesn’t exist, or if the pid is not a child of the current process.
  - how do you enforce this?
  - how do you maintain the process hierarchy?

exit() & waitpid() - synchronization

- How does the parent wait for the child to finish?
- How does the child notify the parent that it has exited?
- What are the scenarios to consider?

exit() & waitpid() - synchronization

- How does the parent wait for the child to finish?
- How does the child notify the parent that it has exited?
- What are the scenarios to consider?
  - parent exits, then child exits
    - who will clean up the children?
  - parent waits, then child exits
  - child exits, then parent waits
  - child exits, then parent exits without waiting
    - who will clean up the children?

fork()

- create a new process and thread
- what needs to be copied?
- what needs to be different?

fork()

- create a new process and thread
- what needs to be copied?
  - process stuff: address space, file descriptors, cwd
  - thread stuff: execution state, stack pointer,
- what needs to be different?
  - pid, return values
  - how do you change the return value in child/parent?
    - trapframe!

fork() - synchronization

- How do you setup the process hierarchy?
- How does the child return to user land?
- How does the parent wait for the child to finish initializing?
  - why? parent process needs to return the child pid in fork()
- How does the child wait for the parent to finish initializing?
  - why? setting up process hierarchy

fork() - error handling

- need to allocate lots of things for the child:
  - process struct, pid, address space, file descriptors, new thread, etc.
- what if an error occurs?
  - need to cleanup/free all the resources that we allocated, and return error to userland
- what if an error occurs in child after thread_fork()?
  - how does the parent cleanup now?

execv

- replace a process’s address space and execution state with new binary
  - kind of like runprogram()
  - does NOT return on success
- need to be careful on how to handle and setup arguments
- execv is the one that sets up argc and argv for main()

execv

- What does execv do?

execv

- What does execv do?
  - open binary file
  - create new address space
  - load executable there
  - copy arguments from old address space into new address space
  - define the stack
  - enter usermode

execv - arguments

- How do you copy in arguments from old address space?
  - copyin() - why do we need this?
  - user level pointers are dangerous!
- How do you copy out arguments to new address space?
  - copyout()
- How should the arguments be laid out in the new address space?
  - this is the hard part.
- Where should the stack pointer be after arguments have been copied into the new address space?

execv - arguments

Old address space:

```
char *prog = “ls”;
// argv must be
// NULL-terminated
char *argv[3];
argv[0] = “ls”;
argv[1] = “foo”;
argv[2] = NULL;
execv(prog, argv);
```

What does the new address space look like?

execv - arguments

Old address space:

```
char *prog = “ls”;
// argv must be
// NULL-terminated
char *argv[3];
argv[0] = “ls”;
argv[1] = “foo”;
argv[2] = NULL;
execv(prog, argv);
```

new address space:

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

execv - arguments

- Where is the new stack pointer?
- Why do argv[0] and argv[1] each occupy 4 bytes?
- Why is the box at 800 empty?
- Why is there nothing at 795?
- Why are 788-791 all 0?

new address space:

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

execv - arguments

- Where is the new stack pointer?
  - at the base of argv
- Why do argv[0] and argv[1] each occupy 4 bytes?
  - because they are pointers, and sizeof(pointer) == 4
- Why is the box at 800 empty?
  - 0x80000000 is the start of kernel memory
- Why is there nothing at 795?
  - pointers must be 4-byte aligned
- Why are 788-791 all 0?
  - argv must be NULL terminated in the new address space

new address space:

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

execv - synchronization

- can multiple processes call execv at the same time?

execv - synchronization

- can multiple processes call execv at the same time?
  - is this feasible? what will determine if this is possible?
    - memory constraints? other resource constraints?

execv - error handling

- What if you free your old address space, and then an error occurs when copying over to new address space?

scheduler

- do something better than round-robin
  - convince us why it’s good
- make it easily swappable and tunable
  - use #ifdef’s
- what parameters can be used to tune your scheduler?
- what bookkeeping information do you need?
- how do you collect these statistics?
- when/how should you migrate a thread between processors?

scheduler - key metrics

- cpu utilization: % time CPU is running threads
- cpu throughput: # jobs per time
- turnaround time: {end time} - {start time}
- response time: total time jobs spend on ready queue
- waiting time: total time jobs spend on wait/sleep queue

scheduler - goals

- efficiency - maximize cpu utilization & throughput
- latency - minimize response time
- fairness - distribute resources equitably

scheduler - strategies

- Random (it’s actually pretty good)
- MLFQ (multilevel feedback queue)
- Lottery
- First-come, first serve
- Shortest remaining time first
- others

suggestions for testing

- kernel level tests (from the kernel menu)
  - feel free to add more kernel-level tests/commands to test kernel data structures/invariants (e.g. pid allocation)
- userland tests
  - get runprogram() to work (“p” in kernel menu) first ⇒ can test userland programs without implementing fork, waitpid, exec
  - after fork, waitpid, exec are implemented, you can now test more thoroughly using the userland shell (“s” in kernel menu)
  - no gdb at user level :(
- feel free to add more simple userland programs to test system calls
  - e.g. opening a file, reading, writing, fork, wait, exec

lessons learned from Kenny

- synchronization and error handling are the hardest part
  - THINK VERY CAREFULLY about synchronization before starting to code
  - coding without a synchronization plan ⇒ eventually require massive rewrite
- use goto pattern for error handling/resource cleanup
  - little memory (4MB) ⇒ failures are the norm
  - expect lots of bugs to occur from improper error handling and resource cleanup
- check your invariants with KASSERT
  - fail quickly ⇒ faster debugging sessions
- review your partner's commits!
  - you and your partner should agree on a coding style and be consistent
  - use "git show COMMIT" or "git diff COMMIT1 COMMIT2"
  - reviewing your partner's commits ⇒ less likely to be caught off guard by changes you didn't know about
- drink some coffee
  - coding when you're sleepy ⇒ more errors/harder to debug/need rewrite