Design Overview of the FreeBSD Kernel

CIS 657
Organization of the Kernel

- **Machine-independent**
  - 86% of the kernel (80% in 4.4BSD)
  - C code

- **Machine-dependent**
  - 14% of kernel
  - Only 0.6% of kernel in assembler (2% in 4.4BSD)
What Code is Machine Independent?

- Basic kernel facilities: timer and clock handling, descriptor and process mgmt.
- Memory mgmt: paging & swapping
- Generic system interfaces: I/O, control, multiplexing performed on descriptors
- Filesystem
- Terminal-handling support
- Inter-Process communication (sockets)
- Network communication (protocols, routing)
- Linux compatibility
What Code is Machine Dependent?

- PCI and ISA bus support
- Virtual memory (hardware-specific)
- Assembly language (trap handling, context switching…)
- Linux compatibility
- “other” machine dependent
- Header files for specific hardware
Kernel Services

- Kernel runs in separate address space
- Kernel runs privileged instructions in system mode of processor
  - I/O
  - changing processor mode/state
  - stopping processor
- Boundary is crossed with system calls
- System calls perform complex actions in kernel space
- All system calls appear synchronous to applications (example: write)
Write Appears Synchronous

Process Timeline

write()

Kernel Timeline

copy buffer

buffer written to disk

process continues

Time
System Call Implementation

- Parameters of call, and system call #, pushed on process stack
- Process executes TRAP instruction
  - actual instruction is processor dependent
- CPU microcode changes privilege level to system mode and page tables
- executes system call indicated by call # (lookup table)
Arguments copied to kernel space and verified.

Kernel returns results in registers or copies them to user-level memory.

On error, return -1 and set global variable `errno`.

All kernel data structures stored in kernel address space... why?
Process Management

- Multitasking environment...what does this mean?
- A *process* is a thread of execution.
- Process *context*:
  - user level: address space, run-time environment
  - kernel level: scheduling parameters, resource controls, id information.
Process Management II

- Users can (all through system calls)
  - create processes
  - control processes’ execution
  - monitor execution status

- Each process gets a unique process identifier (pid)
  - user references pid in system call
  - kernel uses it for internal table lookup (directly?) and to report status to user
Kernel creates new process by copying context of an existing process

- `fork()` system call
- original process is the parent
- new process is the child

Except for pid, both user and kernel state is duplicated
Process States

- Parent process
- Child process
- Zombie process

Actions:
- fork()
- execve()
- wait()
- exit()
Process Management System Calls

- fork(): creates a new child process
  - “exact” copy of parent
    - file descriptors are the same
    - signal handling status
    - memory layout and copy of address space (but not shared memory)
  - Both process return from fork()
    - child receives 0
    - parent receives pid of child
Process Management
System Calls II

- `execve()`
  - an entire family of calls (`execl`, `execve`, etc.) we generically refer to as “exec”
  - overlays the current address space with the memory image of a program
  - keeps old context information
  - closely linked to `fork()`…the vast majority of `fork()` calls are almost immediately followed by `exec()` calls. How could we optimize this?
Process Management
System Calls III

- **exit()**
  - process terminates cleanly (a dirty termination could be the result of a bug)
  - returns an 8-bit status code to the parent (more requires use of IPC)

- **wait()**/**wait4()**
  - suspend the caller until a child exits
  - find out what the little bugger was up to
  - zombie processes have exited, but the parent hasn’t yet called wait()
Process Management
System Calls IV

- `nice()`
  - allows processes to influence the scheduling algorithm
  - can only decrease one’s own priority (hence the name: you can be nice to others)
  - this has only minor overall influence on the basic algorithm
Signals

- software interrupts
- generated by kernel, in response to a process’s actions, external events, or other processes’ system calls
- processes may specify handler functions that catch the signal
- a signal $X$ that has been raised is blocked while the handler for $X$ is running.
Signals II

- Processes may also tell the kernel to ignore (block) all occurrences of a signal.
- A process can also restore the default action of the kernel.
- Almost all uncaught signals cause process termination—might cause a core dump.
- Some signals cannot be caught or ignored (SIGKILL, SIGSTOP).
- All signals have the same priority (race conditions possible).
Process Groups

- Control access to terminals (job control)
- an entire pipeline is one process group:
  \texttt{cat myfile | grep foo | wc}
- allow signals to be delivered to a set of associated processes
- Process Group ID (pgid) inherited from parent
  - Can be changed by me, for me (or my descendants)
- Groups of process groups are called sessions
  - Primarily to provide daemons with isolated environments
Memory Management

- Each process has its own address space
- Three logical segments:
  - text -- program code (rx), size only changes on exec()
  - data -- the heap (rw), grow/shrink through system calls
  - stack -- stack for procedure calls (rw), grow/shrink automatically by the kernel
Memory Management II

- demand paged virtual memory
- swapping of entire process context when necessary
- with 4.4BSD, the VM subsystem was redesigned
  - from: small, expensive memory; fast disk
  - to: large, cheap memory; slow disk; multiprocessors; shared memory; machine independence
Memory Management III: Copying vs. Memory Mapping

- 4.4BSD has `mmap()`, so why copy into the kernel? Discuss:
  - alignment
  - copy-on-write overhead
  - address translation cache effects

- Result: `mmap()` is used to access large files and to share data between processes without copying, but not for passing system call parameters.
Memory Management in the Kernel

- Kernel needs to allocate memory to service system calls—short term need
- Kernel has limited stack; can’t just allocate memory there as we would with a user process
- Kernel has its own memory allocator (like malloc()/free() for user programs)
- Requires extremely careful programming
  … why?
Zone Allocation

- Some large, persistent allocations don’t work well with kernel malloc
  - E.g., structure tracking information about process
- Solution: zone allocation
  - Useful for a collection of items of identical size
  - zalloc(), zfree()
I/O

- Powerful fundamental model: sequence, or stream, of bytes, with either random or sequential access
  - No access methods, no control blocks, etc.
  - User-level libraries can build structure on this, but the kernel only sees sequences of bytes
  - “Amish” files: no bells, no whistles, no shiny objects
  - Simplicity begets efficiency
Descriptors and I/O

- Unix processes don’t reference I/O streams directly; they use descriptors
  - unsigned integers
  - obtained from open(), pipe(), and socket() syscalls; inherited from parent process; or received via socket IPC
  - read(), write() transfer data to/from descriptors
  - close() deallocates a descriptor
Three Things Descriptors Can Reference

- **Files:**
  - linear array of bytes,
  - at least one name
  - exists until all names are deleted, and no open descriptors
  - I/O devices look like files
  - open() system call

- **Sockets**
  - transient object used for interprocess communication: only exists when a process holds descriptor for it
  - generic communication endpoints
  - heavily used for networking
Three Things II

- **Pipes:**
  - linear array of bytes
  - used only as I/O stream
  - unidirectional
  - accessed through pair of descriptors
    - one for writing
    - one for reading
  - pipe() system call

- **FIFO**
  - special kind of pipe, appears in file space
  - one process uses open() for reading
  - one process uses open() for writing
Descriptor Information

- Kernel keeps a *descriptor table* for each process
- Map from descriptor (index) to information about the object
- On process exit, kernel reclaims open descriptors; might then delete object

- Kernel keeps a *file offset* associated with each descriptor
  - updated on each `read()`, `write()`, or `lseek()`
  - can’t `lseek()` on a pipe or socket
Descriptor Management

- Three standard descriptors:
  - 0: standard input
  - 1: standard output
  - 2: standard error
  - inherited via `fork()` and persist across `exec()`
  - start out associated with terminal for a login session
  - I/O redirection changes this
I/O Redirection

myprog | grep foo > foofile

- **Pipe (“|”)**
  - create pipe with pipe() call
  - fork() two new processes
  - close() stdout of first process
  - dup() write end of pipe onto stdout

- **Output (“>”)**
  - close() stdin of second process
  - dup() read end of pipe onto stdin
  - close() stdout
  - open() output file for writing
  - dup() file descriptor onto stdout
Descriptor Management Review

- `open()`, `pipe()`, `socket()` calls allocate descriptors
  - allocate lowest available descriptor
- `dup()`, `dup2()` clone descriptors
  - `dup()` picks lowest available
- `close()` deallocates and makes the table slot available
Devices

- Appear in the file space (except networks)
  - filenames (typically /dev/…)
  - can be accessed with regular file syscalls

- Two kinds of devices:
  - structured (block)
  - unstructured (character)

- Device drivers sit “below” some of the system calls (read, write, ioctl)
Special Device System Calls

- **mknod()**
  - creates device special files (those things that live in /dev) to be associated with device drivers

- **ioctl()**
  - the “kitchen sink” I/O call. Everything that doesn’t map onto standard calls
  - some devices have most of their interface here
Sockets

- Remember that network devices don’t have device special files
- They use sockets: generic communications endpoints
- Can be mapped to multiple protocols (e.g., IPX, TCP/IP, etc.)
- Create the endpoints, then connect.
Filesystems & Filestores

- All the usual stuff (tree-structured, directories, links, protection…)
- Split the filesystem into two parts:
  - naming, locking, protection (common to all filesystems)
  - layout of storage on the physical medium (the filestore)
  - Berkeley FFS, Sprite LFS, VM-based