Operating System Security

Fall 2024

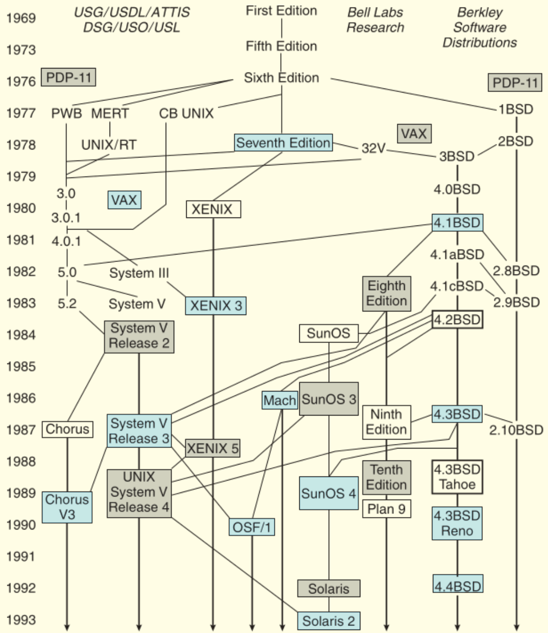
R. Sekar

References

- The UNIX Time-Sharing System
  - Dennis Ritchie and Ken Thompson ACM Symposium on Operation Systems Principles, 1974.
- 4.2BSD and 4.3BSD as Examples of the UNIX System
  - Quarterman, Silberschatz and Peterson
  - ACM Computing Surveys, 1985

History

- Started with one person, Ken Thompson in 1969 in Bell Labs
  - Dennis Ritchie joined soon after
  - V6 introduced multiprogramming, with the OS rewritten in C
- Built on the experience of Multics
  - Not just what to do, but also what *not* to do
    - UNIX $\neq$ Multics!
- No big committees, no grand vision, or elaborate plans
  - Just a system for the personal convenience of the authors
  - “by programmers for programmers”


1969
1973
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
USG/USDL/ATTIS
DSG/USO/USL
Bell Labs
Research
Berkley
Software
Distributions
First Edition
Fifth Edition
Sixth Edition
PDP-11
PDP-11
PWB
MERT
CB UNIX
1BSD
UNIX/RT
Seventh Edition
32V
VAX
2BSD
3BSD
3.0
VAX
4.0BSD
3.0.1
XENIX
4.1BSD
4.0.1
4.1aBSD
5.0
System III
2.8BSD
4.1cBSD
5.2
System V
XENIX 3
Eighth
Edition
2.9BSD
4.2BSD
System V
Release 2
SunOS
Mach
SunOS 3
Chorus
System V
Release 3
Ninth
Edition
4.3BSD
2.10BSD
XENIX 5
UNIX
System V
Release 4
Tenth
Edition
4.3BSD
Tahoe
Chorus
V3
SunOS 4
Plan 9
OSF/1
4.3BSD
Reno
Solaris
4.4BSD
Solaris 2

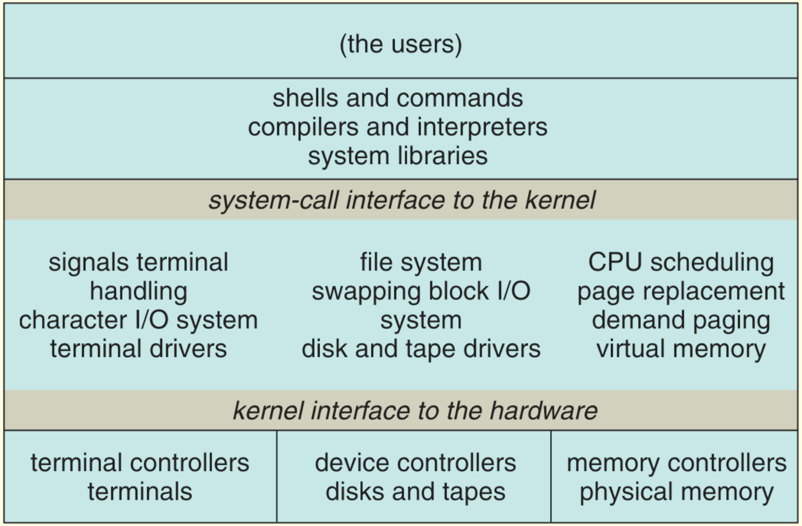
Hardware

- Initial versions developed on PDP-7
  - V6 ran on PDP-11
- BSD versions developed on VAX
  - it offered better hardware support to implement OS features
- Sun Microsystems used Motorola 68000 and then SPARC processor
- Intel became a viable candidate after 80286
  - More development started after 80386
- Linus Torvalds released Linux in 1991 (Intel 80386/80486)

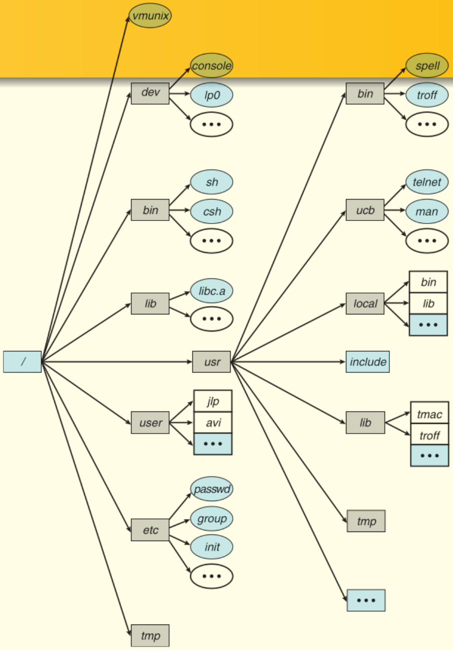
Key Features

- User level is a set of processes, the kernel does not place requirements on which processes should run etc.
- Command shell is one such process, and there can be many such command shells
- Programs can easily spawn other programs, orchestrate them to achieve higher level goals (e.g., make)
- Interconnect programs through pipes
  - enabled by restricting to one file type (sequence of bytes)
- Everything is a file
  - in particular, every device is a file
  - file permissions provide a universal access control mechanism

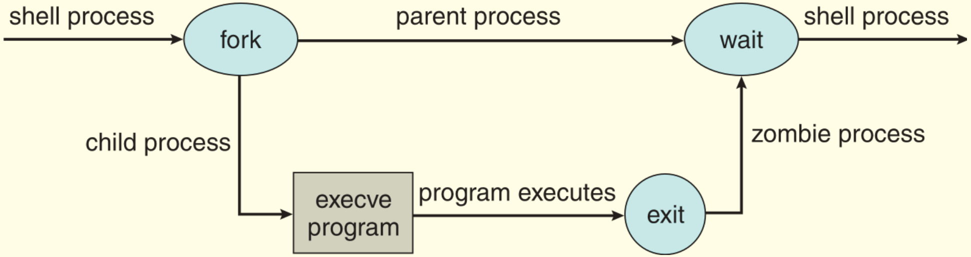
System Architecture/Organization



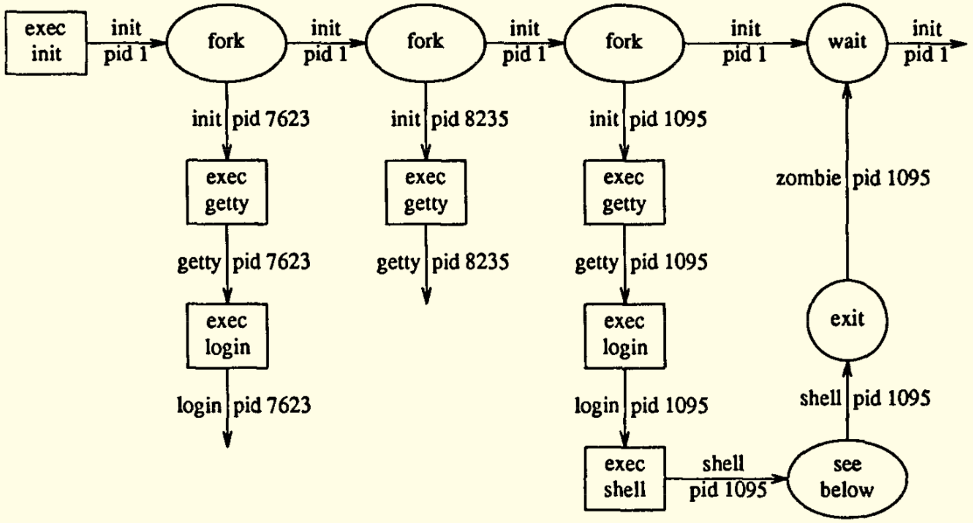
File System Organization



Process Lifecycle



Initial startup



Quotes

"The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided."

— Thompson [1978]

"Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

— Thompson [1978]

UNIX Philosophy [McIlroy et al. 1978]

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features."
- Expect the output of every program to become the input of another, as yet unknown, program. Do not clutter output with extraneous information. Avoid stringently columnar or binary input formats. Do not insist on interactive input.
- Design and build software, even operating systems, to be tried early, ideally within weeks. Do not hesitate to throw away the clumsy parts and rebuild them.
- Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you have finished with them.

UNIX provided the setting for numerous tools that have been critical in software:

- make, awk, sed, lex, yacc, find, SCCS, ...

Processes

Processes

- Process is a program in execution
  - Each process has a pid, owner, group and other attributes
- Address space
  - Processes have separate *virtual* address spaces
    - But do share address space with kernel
    - Typically, upper half of address space used by the kernel
- Memory isolation provides the basis of security
  - A process cannot access the memory of other processes
  - Access to kernel memory controlled by page permissions
  - Read/write/execute permissions set at the granularity of a page

Virtual memory

- Each process has a logically separate address space
  - Virtual address translated to physical address on each memory access
  - Hardware needs to provide support for fast translation
- Swap space: disk space for backing up pages that don't fit into physical memory
- Memory organized into pages (typically, 4KB each)
  - Reduces fragmentation and improves efficiency
- Page fault: Processor exception when a page in not in physical memory
  - OS handles the exception *transparently* to bring this page into memory
- Page replacement algorithms
  - Mostly, approximations of LRU (least recently used)
- Page fault servicing overhead and thrashing

Virtual Memory Allocation

- May be OS initiated ...
  - e.g., when a process is loaded, or when its stack grows
- ... or be requested by a process
  - older mechanism: `brk` and `sbrk` syscalls
  - more powerful: mmap syscall
    - Allows control over size, placement, page permissions, etc.
  - `mprotect` syscall allows permission changes on existing memory.
- Don't confuse between syscalls and user-level programs
  - Shell commands are user-level processes that make their own set of syscalls

Process Control Block (PCB)

- Contains all process state that the OS needs to manage
- Most of the information is the *user* structure
  - register values
    - Process suspension and resumption uses this info
  - uid, gid, current directory, ...
  - open file table
    - file descriptors are indices into this table

`fork` System Call

- Copies user structure
  - So, child inherits uid/gid, open file descriptors, memory, ...
  - So, child is an exact copy of parent, except for return value of syscall (zero for child, pid of child for parent)
- Copying memory is expensive
  - `vfork` avoids needless copying of memory if the next action is `execve`
  - Code memory is shared, does not need copy
  - Modern systems use copy-on-write for data pages
    - So, only page tables need to be copied, not page.
    - `vfork` avoids this as well
- Linux uses `clone`, a generalization that allows fine control
  - whether to create a thread or a process
  - whether to copy page tables, stack, data, ...

execve System Call

- Used to replace calling process with a new program
- All of the memory is overwritten
  - New executable is loaded
  - data pages are reinitialized as specified in the executable
  - command-line argument and environment variables passed in through the new process stack
- But file descriptors are still inherited
  - Unless explicitly closed before or after execve
  - Some fd's have a close-on-exec flag

`exit` and `wait` System calls

- Processes return a status code when they exit
  - Provided as an argument to the `exit` syscall
- Parent receives this status when it `wait`s on a child
- Child cannot fully exit until parent collects this code
  - It is called a “zombie” in this state.
  - Parent should ensure that it collects the status, or else there is a resource leak
- `init` process waits for orphaned children
  - processes whose parent already terminated

Process Scheduling

- Processor time is split across running processes
  - Scheduler is responsible for stopping one process and giving a turn to the next process
  - But most of the time, a running process gets blocked on I/O, avoiding preemption
- Variants of round-robin scheduling
  - Favors I/O bound computation (likely interactive)
  - Different variants of the algorithm based on process behavior
- Process priority allows external control of these choices
  - UNIX lower numeric values for higher priority
  - Only root processes can have negative priority

Process ownership and `setuid`/`setgid` syscalls

- Each process has a *owner*, a *group* owner
  - Represented by `uid` and `gid`
- A root process can change its uid and gid
- Processes can also have *supplementary groups*
  - Roughly speaking, access to a resource is granted if the uid or one of the gids are permitted
- Setuid permission bit allows uid change on `execve`
  - Process assumes the uid of the executable file's owner
  - Perhaps the only feature of original UNIX that was patented
  - Allows normal users to access features that require privilege e.g., `sudo`

Effective, real and saved userids

- Effective: all access checks use this id
- Real: The “real” user, i.e., the user that logged in
    - After executing a setuid executable, real user remains the same, but effective user becomes root.
- Saved: under certain conditions, effective uid is saved in “saved uid”
- A process is allowed to switch between its real, saved and effective userids.
    - Root processes need to be careful in relinquishing privileges: unless all three uids are set to nonzero, the process can reacquire root privilege.
- For fine grained control over different ids, use `seteuid`, `setreuid` and `setresuid` instead of `setuid`.

Userid Vs Usernames

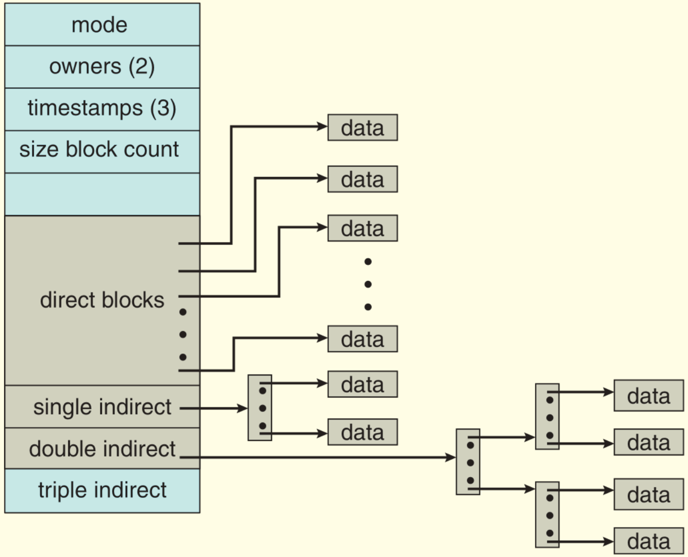
- The file `/etc/passwd` maps userids to usernames
  - Kernel does not know or care about usernames
- Kernel does not interpret userid's either
  - With the sole exception of uid 0 (root, superuser)
- Similarly, group names are specified in `/etc/group`: kernel only cares about gids.
  - Note: gid 0 has no special meaning
- Note: `/etc/passwd` is a "public database," readable by by any user on the system.
  - Encrypted passwords are in `/etc/shadow` that can be read only by the root.

File System

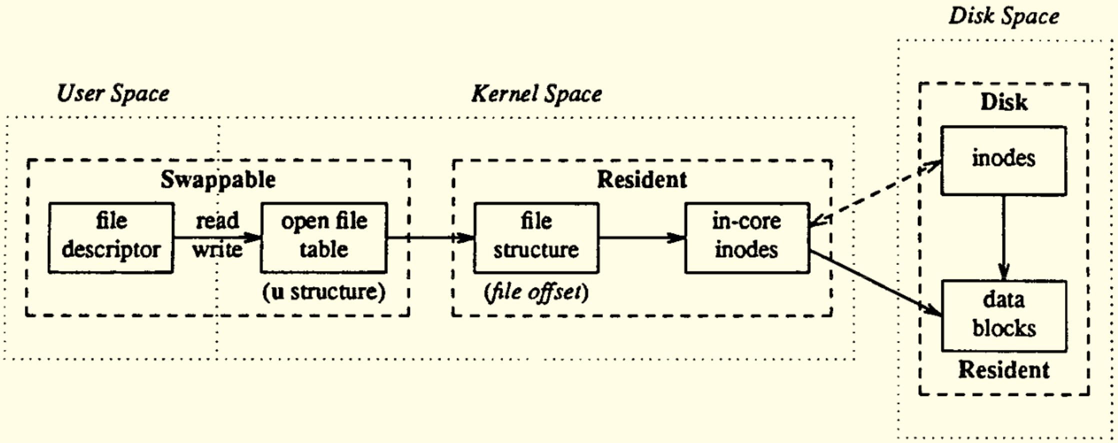
File

- A file is simply a sequence of bytes.
  - The OS does not impose any structure
  - A departure from previous OSes that supported many file types, e.g., text, binary, records, etc.
- Directory: a special file that contains information about the location of files.
- Path: a sequence of directory names followed by a file name
  - Tells the OS how to find the file.
  - Relative path: interpreted relative to current directory of process
  - Absolute path: interpreted relative to the root directory of the system

File implementation: Inodes



Process-related file system data



Key system calls

- open: returns a file descriptor for reading/writing the file
- read, write
- mmap
- lseek
- chmod
- chown

Link vs File

- (Hard) Link is like a file name: it points to the actual file
  - There can be multiple (hard) links
    - No such thing as “the” name of a file
- Symlink: a special file; content interpreted as name of another file/dir
  - Hard links cannot span file systems, but symlinks can
- Lookups and caching
  - Path-to-file translation involves dereferencing many links
  - Caching links is essential for performance
- Cycle prevention during lookup
  - Hard links to directories are not permitted
  - Symlinks to directories are permitted, so OS limits number of symlinks traversed: prevents infinite loops

Link-related syscalls

- rename
- link, unlink
- mkdir and rmdir

Interprocess Communication

Pipes and Socket pairs

- Pipe is a unidirectional communication channel
  - `pipe` syscall returns a pair of fd's: what is written into fd[1] can be read from fd[0]
- `socketpair` is similar, but allows bidirectional communication
  - What is written into fd[0] can be read from fd[1] *and vice-versa*

How does I/O redirection work

- Key point: application transparency
  - Application needs no special code to support redirection
- Approach: by default, all applications read from fd 0, write to fd 1, and display errors on fd 2.
- To redirect input from a file, simply "rename" the fd to 0!
  - uses `dup2(orig_fd, new_fd)` syscall
- To create a pipeline `cat | wc`
  - create a pipe with endpoints fd[0], fd[1]
  - fork: note that the child inherits these fds
  - parent: `close(fd[0]); dup2(fd[1], 1); execve("/bin/cat", ...)`
  - child: `close(fd[1]); dup2(fd[0], 0); execve("/bin/wc", ...)`

Socket Communication

- Sockets may be used by *servers* or *clients*
- Server's steps:
  - `create` $\longrightarrow$ `bind` $\longrightarrow$ `listen` $\longrightarrow$ `accept` $\longrightarrow$ `read/write`
  - create, bind and listen operate on the same socket
  - accept returns a new socket
- Client's steps:
  - `create` $\longrightarrow$ `connect` $\longrightarrow$ `read/write`
  - no new fds are created
- Datagram sockets use `sendto` and `recvfrom`
- Connected sockets can use `read/write` or `send/recv`.

Server Design

- Forking servers
  - Parent sets up server socket
  - accepts each connection
  - forks a child to handle the request
- Single-threaded I/O multiplexing servers
  - Simultaneously wait for input from any of the clients using `select` or `epoll`
  - Process that client
  - Must ensure that no processing step takes too long
- Multi-threaded servers
  - But concurrency is tricky, and programs can be buggy

Signals

Signals Vs Exceptions

- Signal is a exception-related control-flow mechanism
- Modelled after hardware interrupts
  1. Suspend current processing
  2. Serice interrupt in a *signal handler*
  3. Return to original code, resume execution
- Contrast with exception handling in programming languages
  1. Abandon execution of code that caused the error
  2. Unwind call stack until you reach a handler for the exception
  3. Execute the (recovery) code in this handler, continue execution
- Resumption approach is a good match for many signals (e.g., SIGALRM, SIGPIPE), but not all (e.g., SIGSEGV).

Key Signals

- SIGKILL, SIGTERM, SIGINT, SIGQUIT, SIGSTOP, SIGSTP, SIGCONT: Process control (stop, interrupt, resume or kill process)
- SIGSEGV, SIGBUS: illegal memory access
- SIGFPE, SIGILL, SIGABRT: other low-level errors
- SIGCHLD: child exited
- SIGALRM: timer interrupt
- SIGPIPE, SIGHUP, SIGTTIN, SIGTTOU, SIGIO, SIGPOLL, SIGURG: input/output related condition
- SIGSYS: bad (or denied) system call
- SIGUSR1, SIGUSR2: user-defined; unused by OS.

Signal generation, delivery and handling

- Synchronous signals: signal caused by program execution
- Asynchronous signals: generated due to external events
  - Any process can send signal to another process (or itself) using `kill` syscall.
    - Sender should have permission (belong to same process group)
  - Some signals are generated from the keyboard
- Handlers: SIG_IGN (ignore), SIG_DFL (default handler) and user-defined handlers
  - Handlers installed using `signal` or `sigaction` syscalls
  - Signal handlers use a stack that is logically distinct from the program stack
  - Normally, this stack is at the top of program stack; change using `sigaltstack`
- Signals can be blocked (i.e., delayed)
  - But there is no queue: at most one instance of signal can be pending

Signal Handler Issues

- Need care because handlers execute concurrently with the program
- Not safe to call arbitrary library functions
  - Call only async-signal-safe functions
- Nested handler invocations are possible
- Handlers can block signals to reduce (but not eliminate) the critical section problem
- Solution
  - Keep it simple: set some global flag, let the program handle the problem when it is ready.
  - Follow best practices

Recovering from serious errors

- If you take exceptional care, programs can recover from serious errors through signal-handling.
- Prepare for recovery by setting up a recovery context using `setjmp`.
  - You cannot use C++ exceptions inside signal handlers
- Set up a signal handler for SEGV, SYS, etc.
  - Use `longjmp` to jump to the recovery context
- Make sure you don’t leak resources