The Classical OS Model in Unix

A Lasting Achievement?

"Perhaps the most important achievement of Unix is to demonstrate that a powerful operating system for interactive use need not be expensive…it can run on hardware costing as little as $40,000."

The UNIX Time-Sharing System*
D. M. Ritchie and K. Thompson

DEC PDP-11/24

http://histoire.info.online.fr/pdp11.html

Elements of the Unix

1. rich model for IPC and I/O: "*everything is a file*"

   file descriptors: most/all interactions with the outside world are through system calls to read/write from *file descriptors*, with a unified set of syscalls for operating on open descriptors of different types.

2. simple and powerful primitives for creating and initializing child processes

   fork: easy to use, expensive to implement

   Command shell is an "application" (user mode)

3. general support for combining small simple programs to perform complex tasks

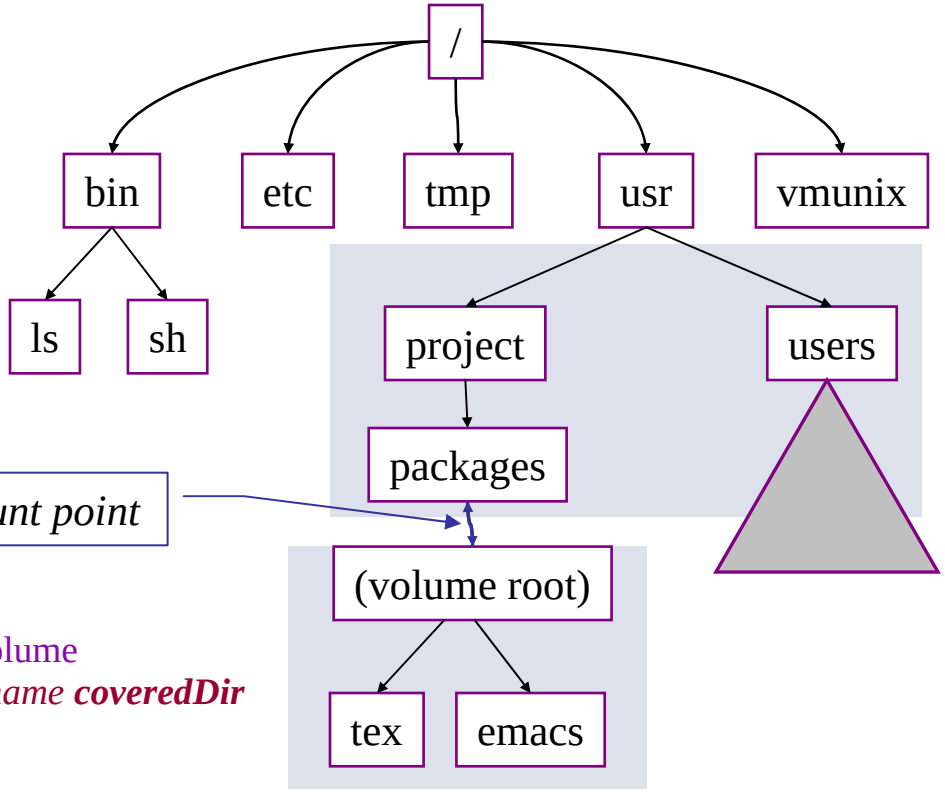
   standard I/O and pipelines

A Typical Unix File Tree

Each volume is a set of directories and files; a host's *file tree* is the set of directories and files visible to processes on a given host.

File trees are built by *grafting* volumes from different volumes or from network servers.

In Unix, the graft operation is the privileged *mount* system call, and each volume is a *filesystem*.



mount (coveredDir, volume)
coveredDir: directory pathname
volume: device specifier or network volume
volume root contents become visible at pathname ***coveredDir***

The Shell

The Unix command interpreters run as ordinary user processes with no special privilege.

- This was novel at the time Unix was created: other systems viewed the command interpreter as a trusted part of the OS.

Users may select from a range of interpreter programs available, or even write their own (to add to the confusion).

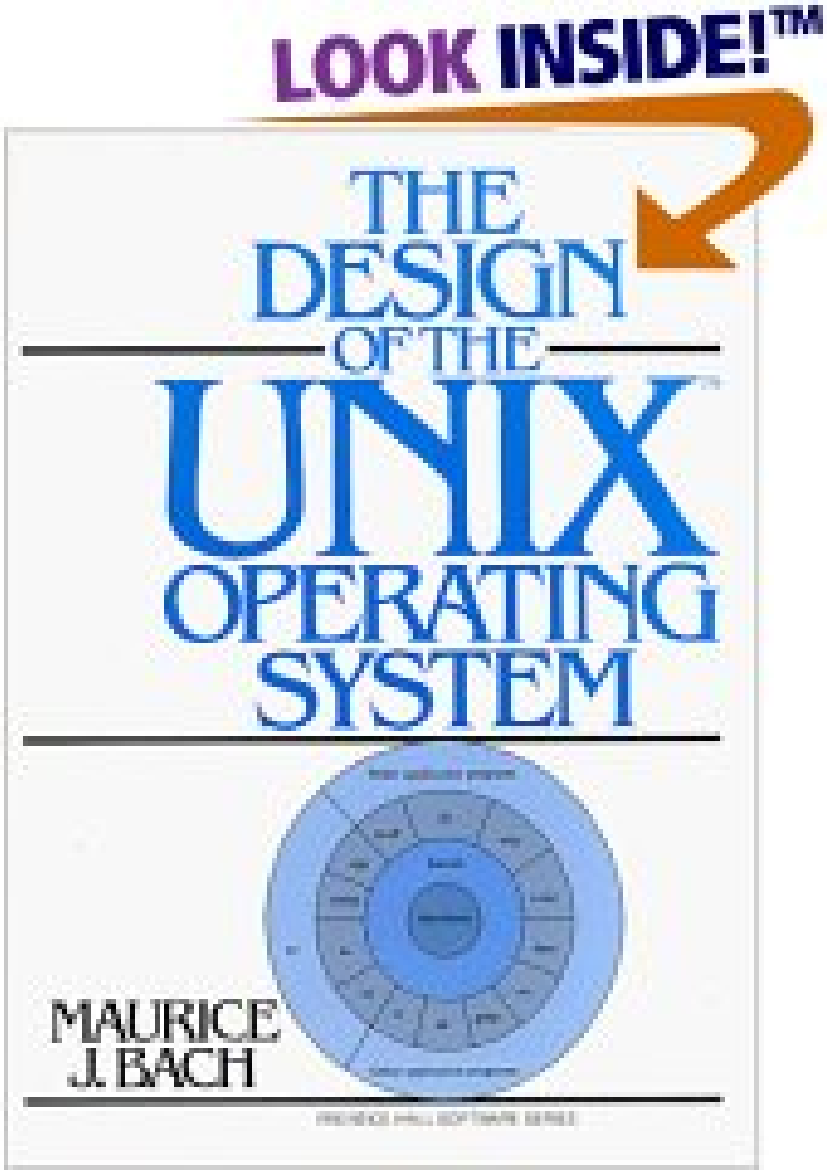
- *csh*, *sh*, *ksh*, *tcsh*, *bash*: choose your flavor...or use *perl*.

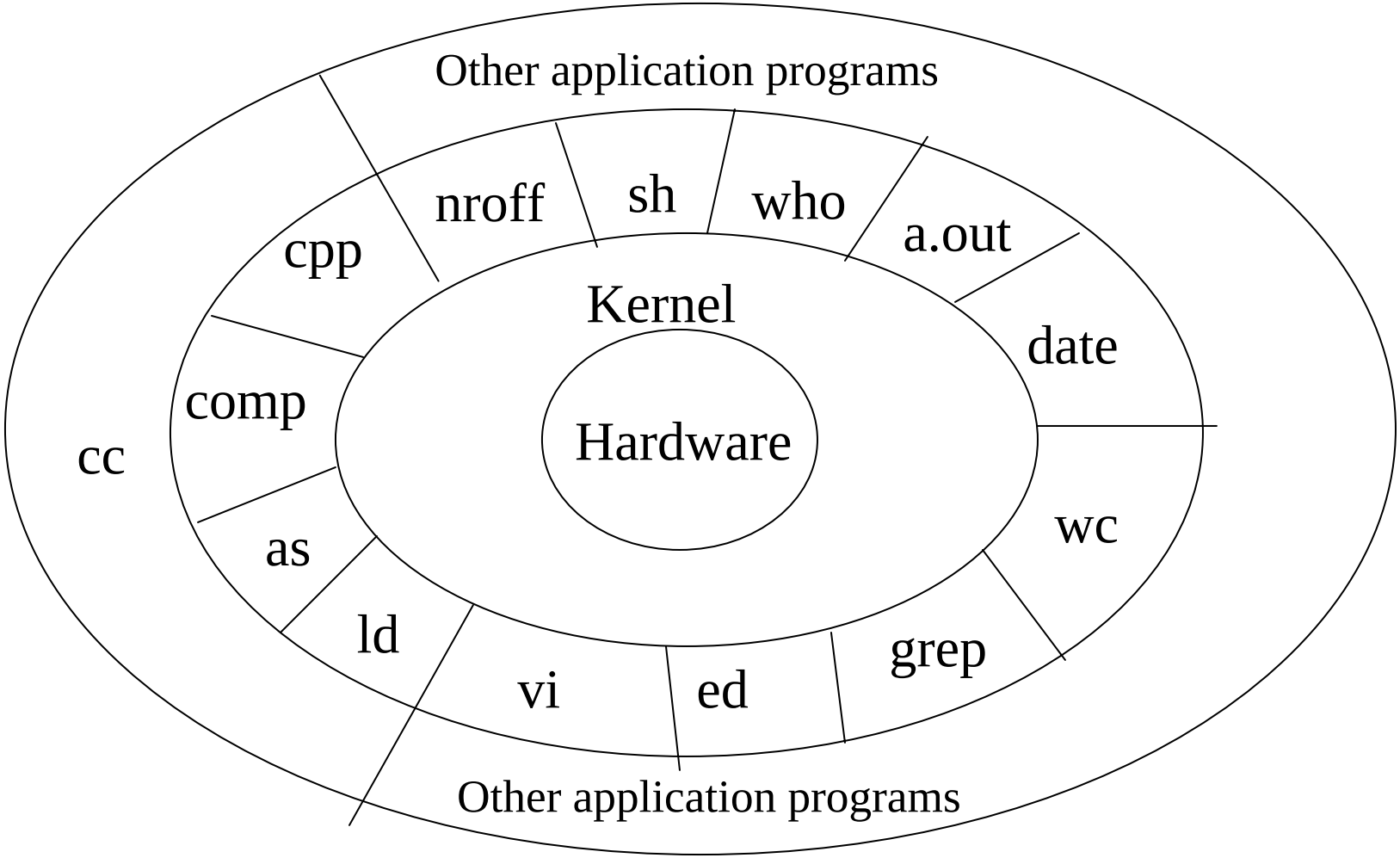
Shells use *fork/exec/exit/wait* to execute commands composed of program filenames, args, and I/O redirection symbols.

- Shells are general enough to run files of commands (*scripts*) for more complex tasks, e.g., by redirecting shell's *stdin*.
- Shell's behavior is guided by *environment variables*.

Using the shell

- Commands: *ls*, *cat*, and all that
- Current directory: *cd* and *pwd*
- Arguments: *echo*
- Signals: *ctrl-c*
- Job control, foreground, and background: &, *ctrl-z*, *bg*, *fg*
- Environment variables: *printenv* and *setenv*
- Most commands are programs: *which*, $PATH, and /bin
- Shells are commands: *sh*, *csh*, *ksh*, *tcsh*, *bash*
- Pipes and redirection: *ls* | *grep a*
- Files and I/O*: open*, *read*, *write*, *lseek*, *close*
- *stdin*, *stdout*, *stderr*
- Users and groups: *whoami*, *sudo*, *groups*


LOOK INSIDE!™
THE DESIGN OF THE UNIX OPERATING SYSTEM
MAURICE J. BACH


Other application programs
nroff
sh
who
a.out
cpp
Kernel
date
comp
Hardware
cc
wc
as
ld
grep
vi
ed
Other application programs

Questions about Processes

A process is an execution of a program within a private virtual address space (VAS).

1. What are the system calls to operate on processes?
2. How does the kernel maintain the state of a process?

   Processes are the “basic unit of resource grouping”.

3. How is the process virtual address space laid out?

   What is the relationship between the program and the process?

4. How does the kernel create a new process?

   How to allocate physical memory for processes?

   How to create/initialize the virtual address space?

Process Internals

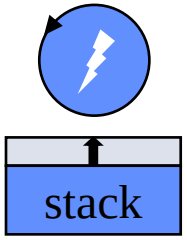
virtual address space

The address space is represented by *page table*, a set of translations to physical memory allocated from a kernel *memory manager*.

The kernel must initialize the process memory with the program image to run.

+

thread



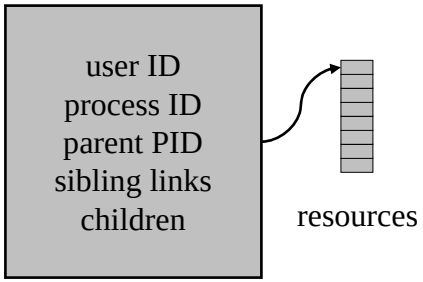
Each process has a thread bound to the VAS.

The thread has a saved user context as well as a system context.

The kernel can manipulate the user context to start the thread in user mode wherever it wants.

+

process descriptor (PCB)



Process state includes a file descriptor table, links to maintain the process tree, and a place to store the exit status.

Process Creation

Two ways to create a process

- Build a new empty process from scratch
- Copy an existing process and change it appropriately

Option 1: New process from scratch

- Steps

  Load specified code and data into memory;
  Create empty call stack

  Create and initialize PCB (make look like context-switch)

  Put process on ready list

- Advantages: No wasted work
- Disadvantages: Difficult to setup process correctly and to express all possible options

  Process permissions, where to write I/O, environment variables

  Example: WindowsNT has call with 10 arguments

Process Creation

Option 2: Clone existing process and change

- Example: Unix fork() and exec()
  - Fork(): Clones calling process
  - Exec(char *file): Overlays file image on calling process
- Fork()
  - Stop current process and save its state
  - Make copy of code, data, stack, and PCB
  - Add new PCB to ready list
  - Any changes needed to PCB?
- Exec(char *file)
  - Replace current data and code segments with those in specified file
- Advantages: Flexible, clean, simple
- Disadvantages: Wasteful to perform copy and then overwrite of memory

Process Creation in Unix

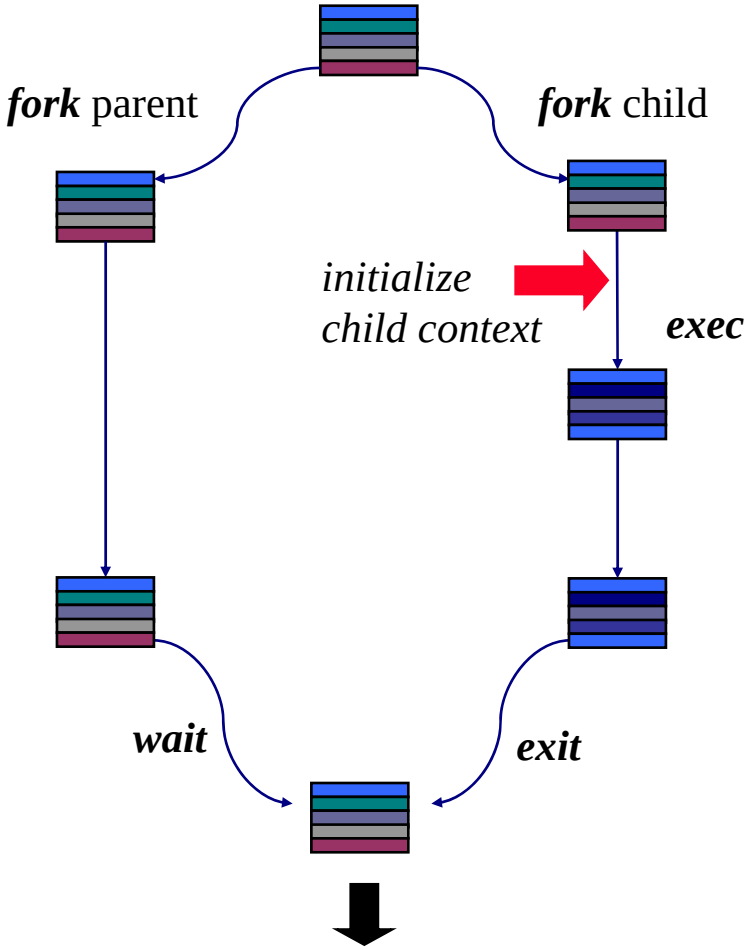
```
int pid;
int status = 0;

if (pid = fork()) {
        /* parent */
        …..
        pid = wait(&status);
} else {
        /* child */
        …..
        exit(status);
}
```

*The **fork** syscall returns twice: it returns a zero to the child and the child process ID (pid) to the parent.*

*Parent uses **wait** to sleep until the child exits; **wait** returns child pid and status.*

***Wait** variants allow wait on a specific child, or notification of stops and other signals.*

Unix *Fork/Exec/Exit/Wait* Example



int pid = fork();

Create a new process that is a clone of its parent.

exec*("program" *[, argvp, envp]*);

Overlay the calling process virtual memory with a new program, and transfer control to it.

exit(status);

Exit with status, destroying the process. Note: this is not the only way for a process to exit!

int pid = wait*(&status);

Wait for exit (or other status change) of a child, and "reap" its exit status. Note: child may have exited before parent calls wait!

How are Unix shells implemented?

```
while (1) {
    Char *cmd = getcmd();
    int retval = fork();
    if (retval == 0) {
            // This is the child process
            // Setup the child's process environment here
            // E.g., where is standard I/O, how to handle signals?
            exec(cmd);
            // exec does not return if it succeeds
            printf("ERROR: Could not execute %s\n", cmd);
            exit(1);
    } else {
            // This is the parent process; Wait for child to finish
            int pid = retval;
            wait(pid);
    }
}
```

The Concept of Fork

fork creates a child process that is a *clone* of the parent.

- Child has a (virtual) copy of the parent's virtual memory.
- Child is running the same program as the parent.
- Child *inherits* open file descriptors from the parent.

  (Parent and child file descriptors point to a common entry in the system open file table.)

- Child begins life with the same register values as parent.

The child process may execute a different program in its context with a separate *exec()* system call.

What's So Cool About *Fork*

1. *fork* is a simple primitive that allows process creation without troubling with what program to run, args, etc.

   Serves the purpose of "lightweight" processes (like threads?).

2. *fork* gives the parent program an opportunity to initialize the child process…*e.g.*, the open file descriptors.

   Unix syscalls for file descriptors operate on the current process.

   Parent program running in child process context may open/close I/O and IPC objects, and bind them to *stdin*, *stdout*, and *stderr*.

   Also may modify environment variables, arguments, etc.

3. Using the common *fork/exec* sequence, the parent (e.g., a command interpreter or shell) can transparently cause children to read/write from files, terminal windows, network connections, pipes, etc.

Unix File Descriptors

Unix processes name I/O and IPC objects by integers known as *file descriptors*.

- File descriptors 0, 1, and 2 are reserved by convention for *standard input*, *standard output*, and *standard error*.

  "Conforming" Unix programs read input from ***stdin***, write output to ***stdout***, and errors to ***stderr*** by default.

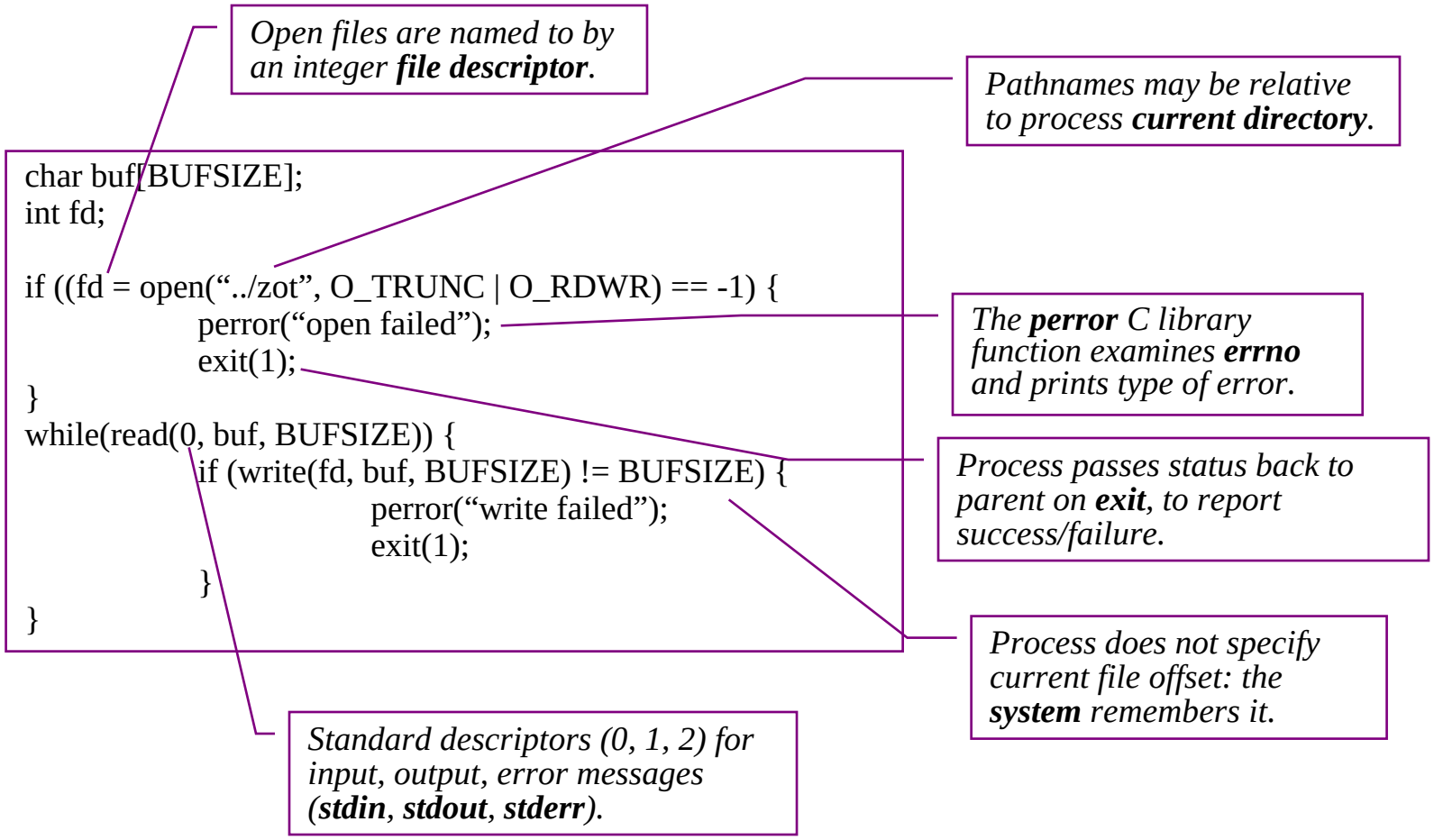
- Other descriptors are assigned by syscalls to open/create files, create pipes, or bind to devices or network sockets.

  pipe, socket, open, creat

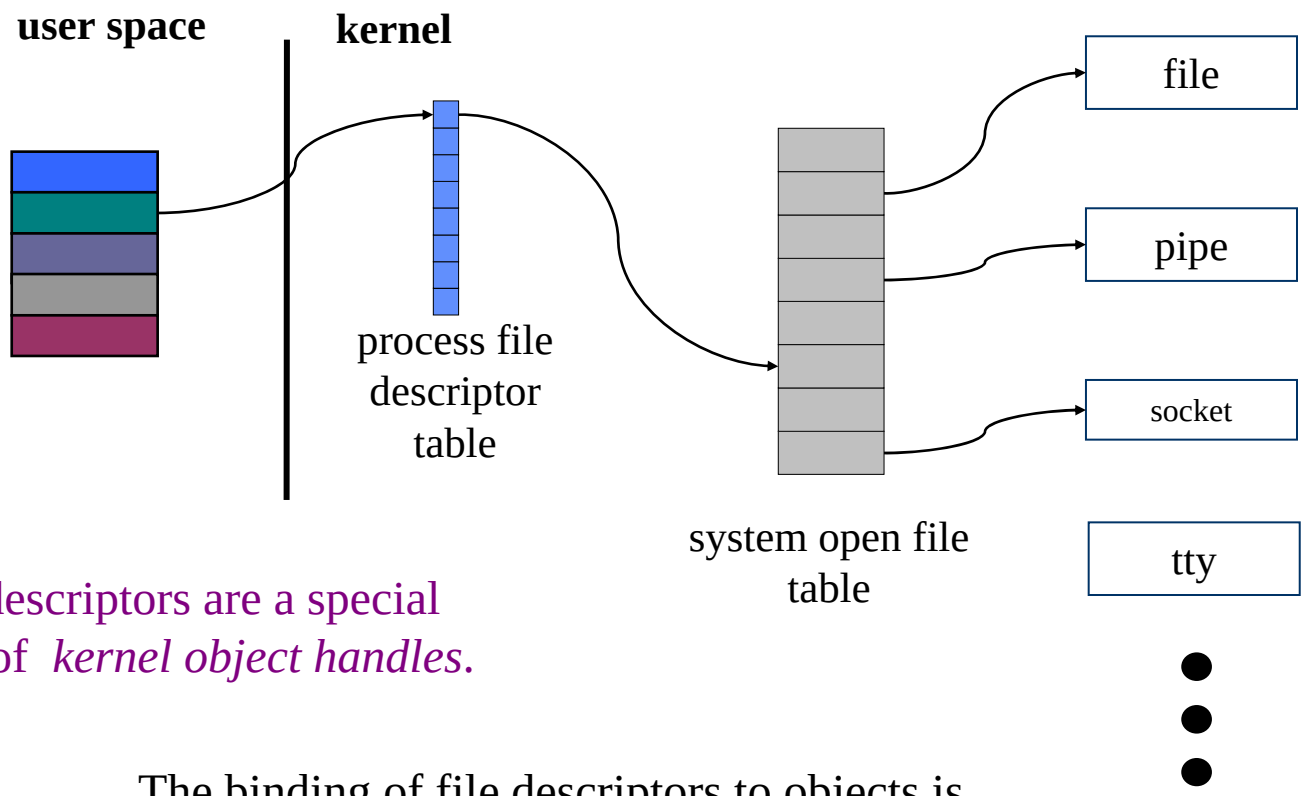
- A common set of syscalls operate on open file descriptors independent of their underlying types.

  read, write, dup, close

The Flavor of Unix: An Example



Unix File Descriptors Illustrated



File descriptors are a special case of *kernel object handles*.

The binding of file descriptors to objects is specific to each process, like the virtual translations in the virtual address space.

Disclaimer: this drawing is oversimplified.

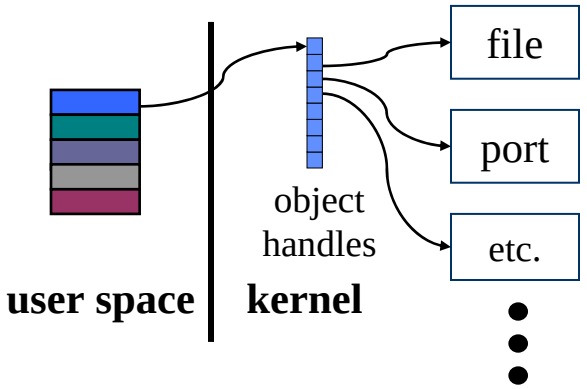
Kernel Object Handles

Instances of kernel abstractions may be viewed as "objects" named by protected *handles* held by processes.

- Handles are obtained by *create/open* calls, subject to security policies that grant specific rights for each handle.
- Any process with a handle for an object may operate on the object using operations (system calls).

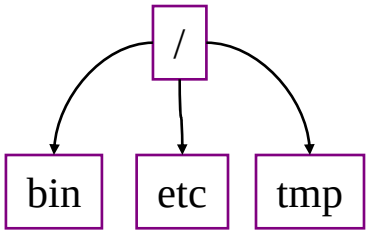
  Specific operations are defined by the object's type.
- The handle is an integer index to a kernel table.

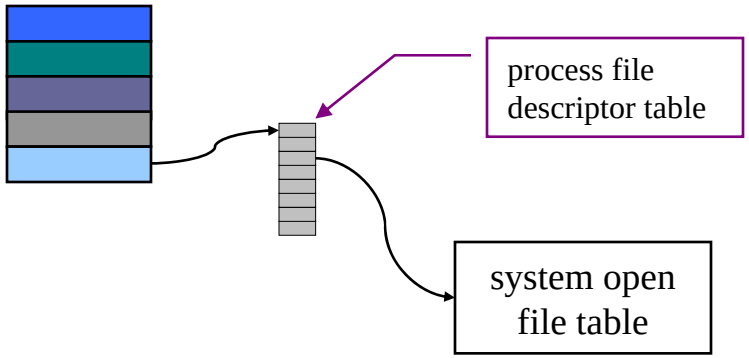
Microsoft NT object handles
Unix file descriptors



Unix File Syscalls

```
int fd;                    /* file descriptor */
fd = open("/bin/sh", O_RDONLY, 0);
fd = creat("/tmp/zot", 0777);
unlink("/tmp/zot");

char data[bufsize];
bytes = read(fd, data, count);
bytes = write(fd, data, count);
lseek(fd, 50, SEEK_SET);

mkdir("/tmp/dir", 0777);
rmdir("/tmp/dir");
```





Controlling Children

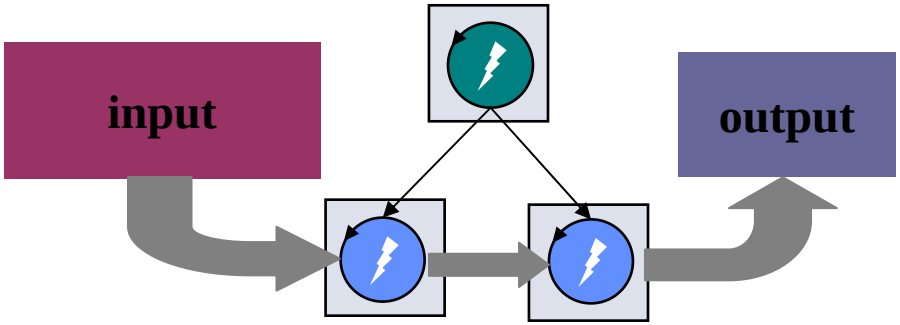
1. After a *fork*, the parent program has complete control over the behavior of its child.
2. The child inherits its execution environment from the parent...but the parent *program* can change it.
   - user ID (if superuser), global variables, etc.
   - sets bindings of file descriptors with *open*, *close*, *dup*
   - *pipe* sets up data channels between processes
   - *setuid* to change effective user identity
3. Parent program may cause the child to execute a different program, by calling *exec** in the child context.

Example: Pipes

Producer/Consumer Pipes

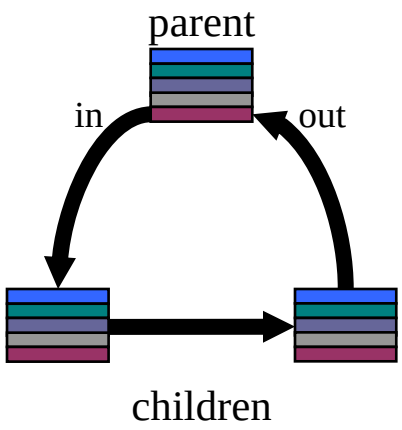
```
char inbuffer[1024];
char outbuffer[1024];

while (inbytes != 0) {
        inbytes = read(stdin, inbuffer, 1024);
        outbytes = process data from inbuffer to outbuffer;
        write(stdout, outbuffer, outbytes);
}
```

Pipes support a simple form of parallelism with built-in *flow control*.



e.g.: *sort <grades | grep Dan | mail varun*

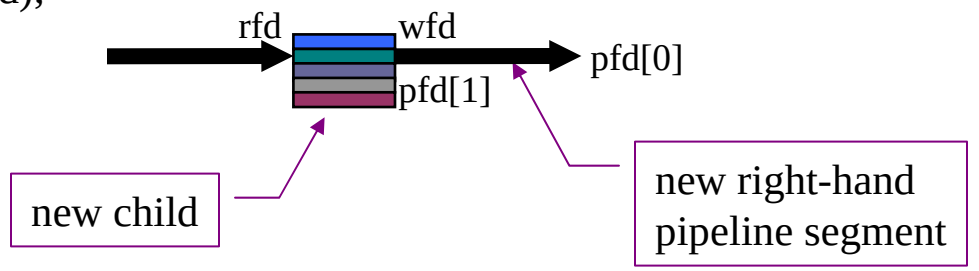
Setting Up Pipes



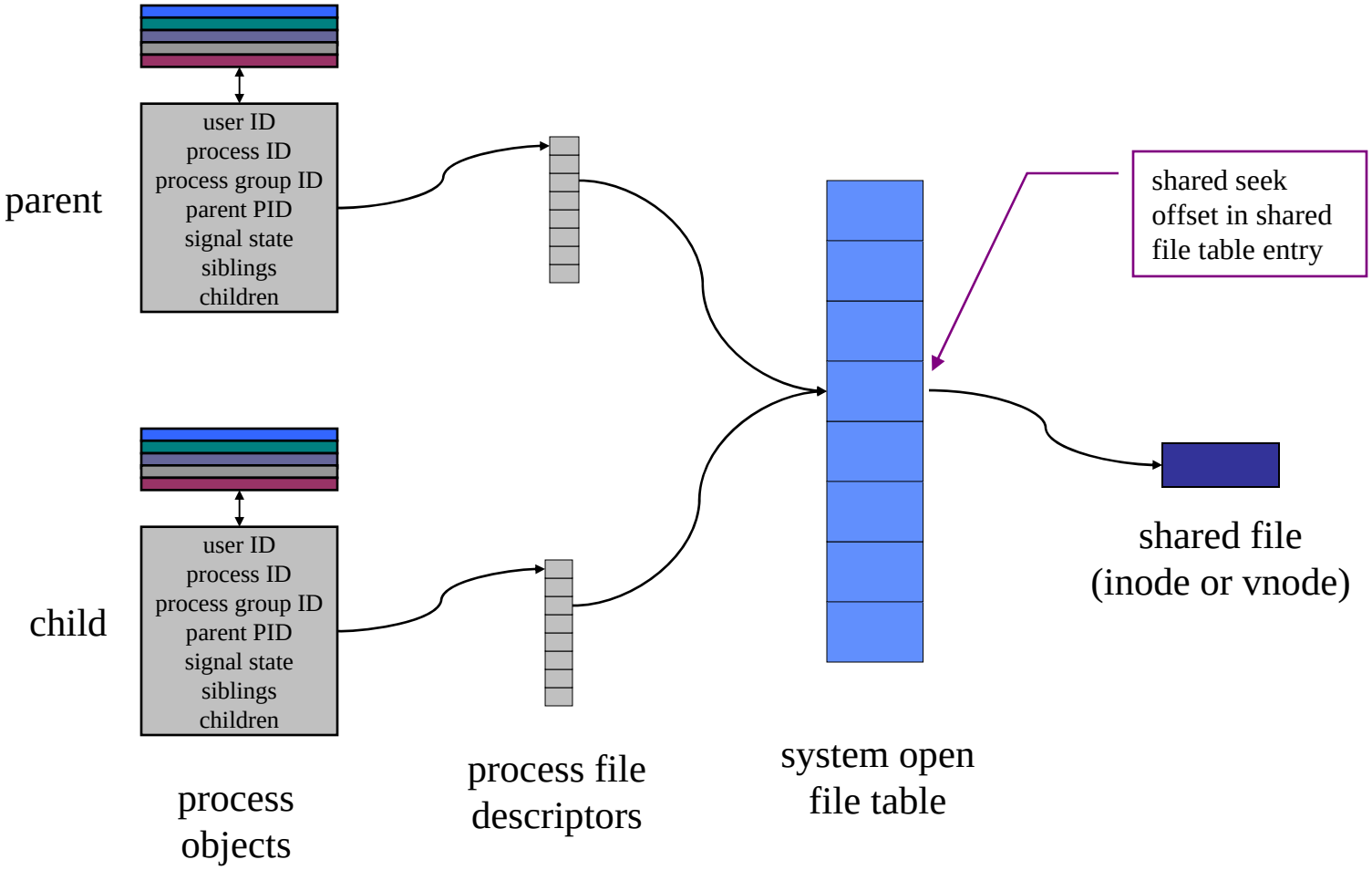

```
int pfd[2] = {0, 0};      /* pfd[0] is read, pfd[1] is write */
int in, out;              /* pipeline entrance and exit */

pipe(pfd);                /* create pipeline entrance */
out = pfd[0]
in = pfd[1];

/* loop to create a child and add it to the pipeline */
for (i = 1; i < procCount; i++) {
            out = setup_child(out);
}

/* pipeline is a producer/consumer bounded buffer */
write(in, ..., ...);
read(out,...,...);
```

Setting Up a Child in a Pipeline

```
int setup_child(int rfd) {
        int pfd[2] = {0, 0};        /* pfd[0] is read, pfd[1] is write */
        int i, wfd;

        pipe(pfd);                  /* create right-hand pipe */
        wfd = pfd[1];               /* this child's write side */
        if (fork()) {               /* parent */
                close(wfd); close(rfd);
        } else {                    /* child */
                close(pfd[0]);      /* close far end of right pipe */
                close(0); close(1);
                dup(rfd); dup(wfd);
                close(rfd); close(wfd);
                ...
        }
        return(pfd[0]);
}
```



Sharing Open File Instances



Unix as an Extensible System

"Complex software systems should be built incrementally from components."

- independently developed
- replaceable, interchangeable, adaptable

The power of *fork/exec/exit/wait* makes Unix highly flexible/extensible...at the application level.

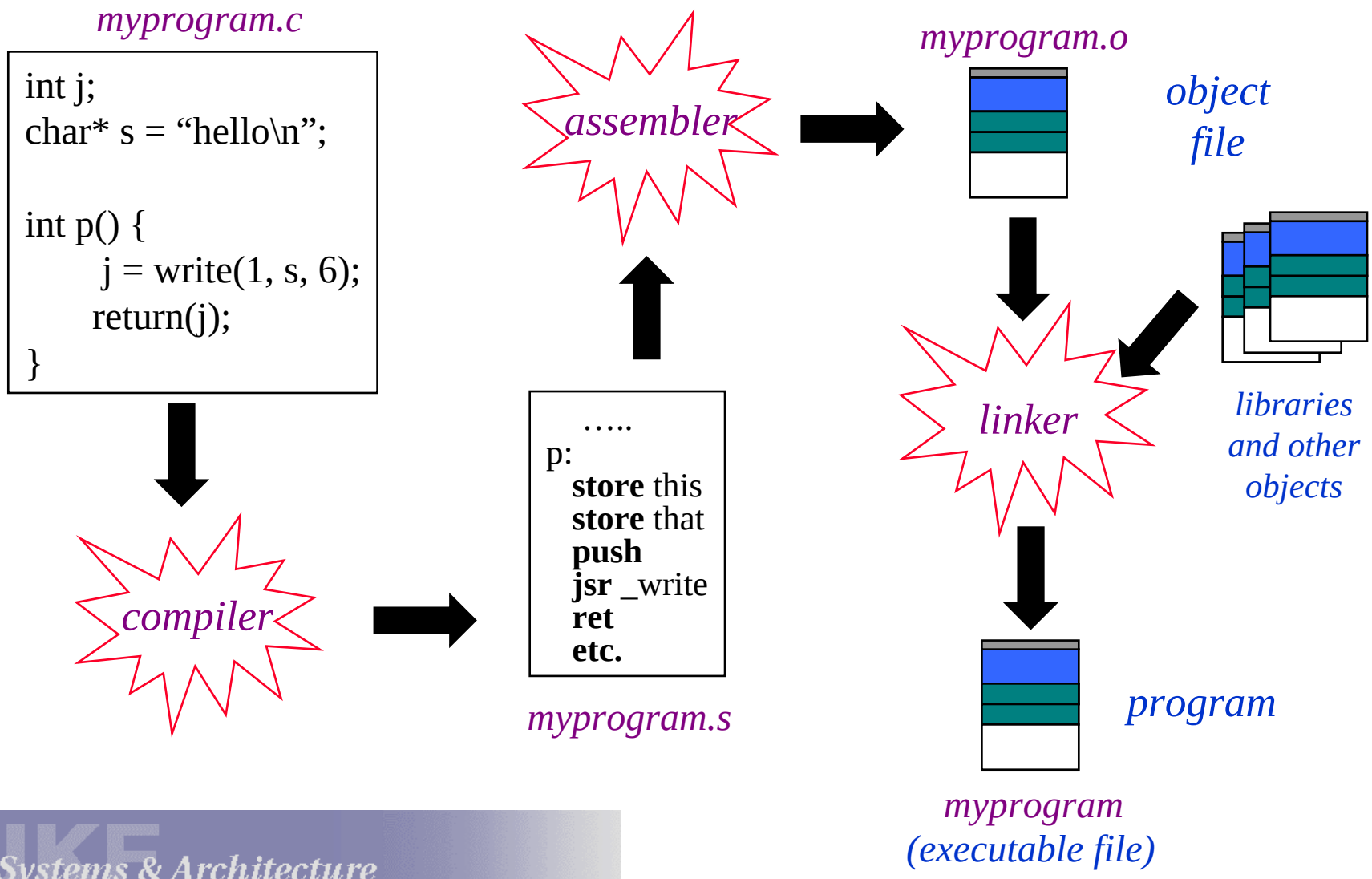
- write small, general programs and string them together

  general stream model of communication

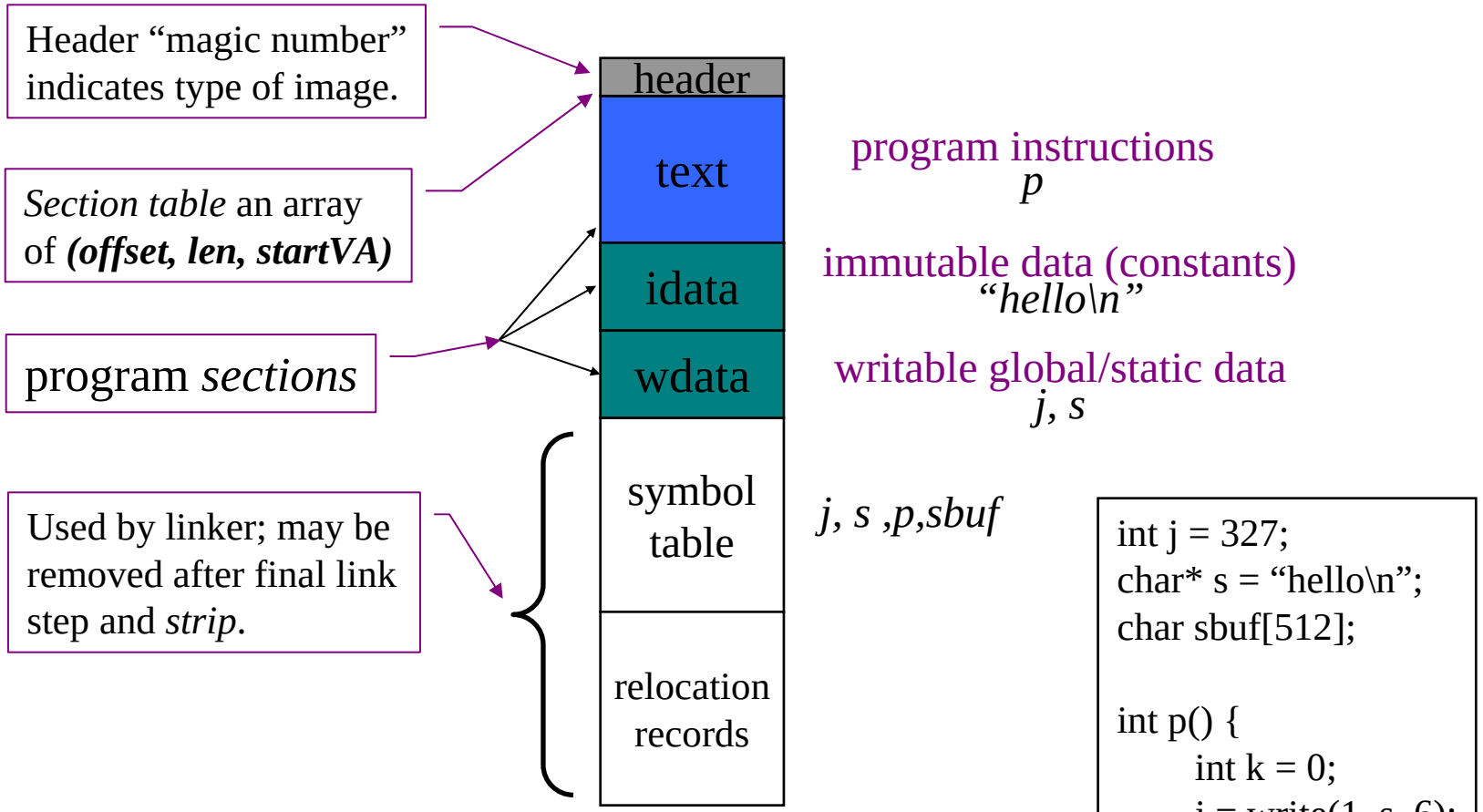
- this is one reason Unix has survived

These system calls are also powerful enough to implement powerful command interpreters (*shell*).

The Birth of a Program



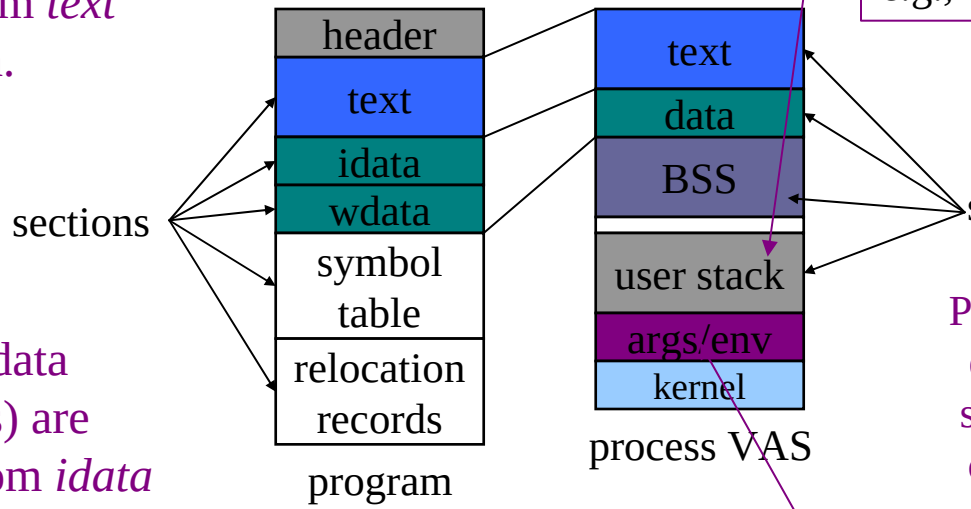
What's in an Object File or Executable?




```
int j = 327;
char* s = "hello\n";
char sbuf[512];

int p() {
        int k = 0;
        j = write(1, s, 6);
       return(j);
}
```

The Program and the Process VAS

Process *text* segment is initialized directly from program *text* section.



BSS
"Block Started by Symbol"
(uninitialized global data)
e.g., heap and *sbuf* go here.

Process data segment(s) are initialized from *idata* and *wdata* sections.

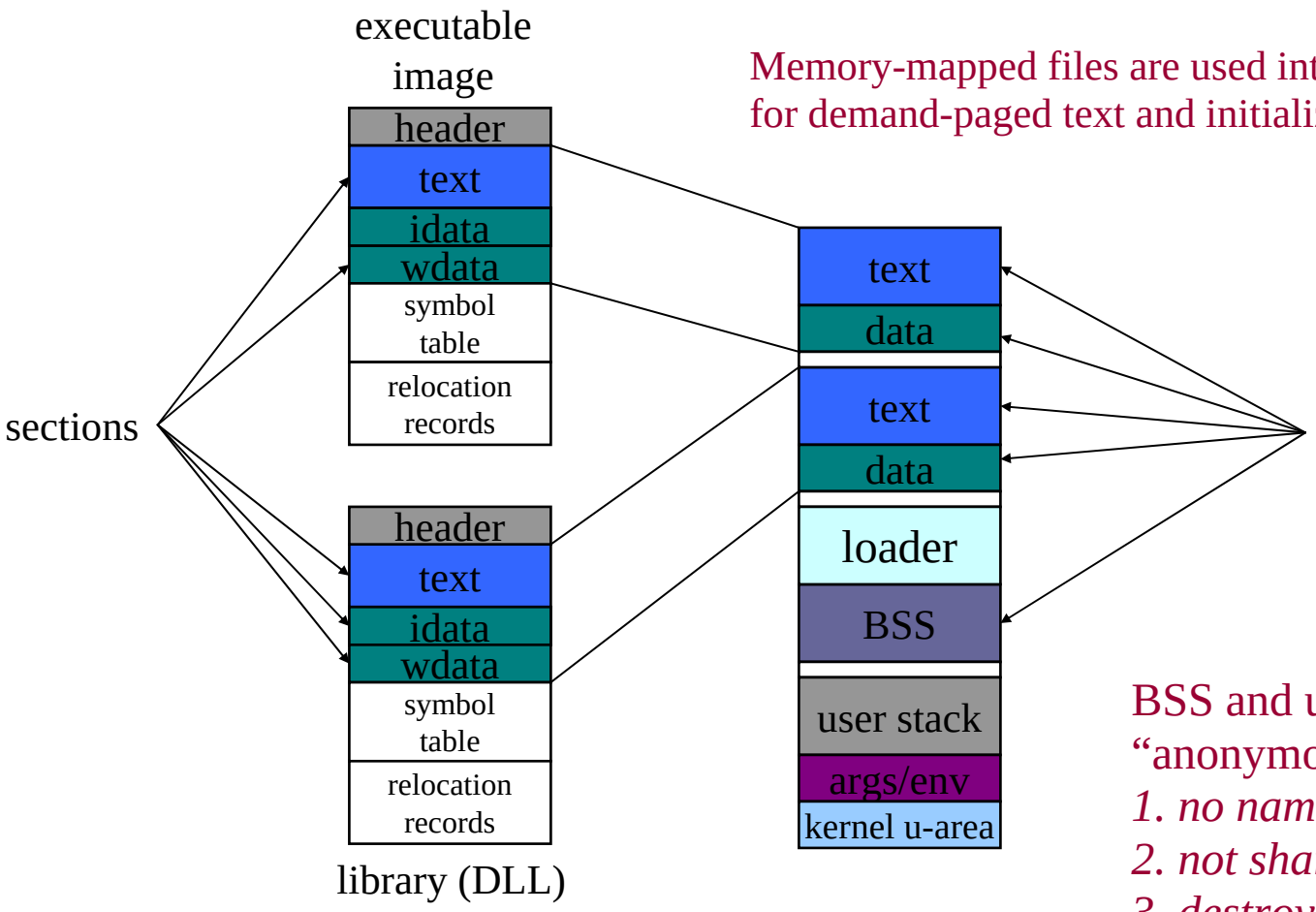
Process BSS segment may be expanded at runtime with a system call (e.g., Unix sbrk) called by the heap manager routines.

Text and *idata* segments may be write-protected.

Process stack and BSS (e.g., heap) segment(s) are zero-filled.

Args/env strings copied in by kernel when the process is created.

Using Region Mapping to Build a VAS



Memory-mapped files are used internally
for demand-paged text and initialized static data.

BSS and user stack are
"anonymous" segments.

1. no name outside the process
2. not sharable
3. destroyed on process exit

Unix Signals 101

Signals notify processes of internal or external events.

- the Unix software equivalent of interrupts/exceptions
- only way to do something to a process “from the outside”
- Unix systems define a small set of signal types

Examples of signal generation:

- keyboard ***ctrl-c*** and ***ctrl-z*** signal the *foreground process*
- synchronous fault notifications, syscall errors
- asynchronous notifications from other processes via ***kill***
- IPC events (SIGPIPE, SIGCHLD)
- alarm notifications

signal == “upcall”

Process Handling of Signals

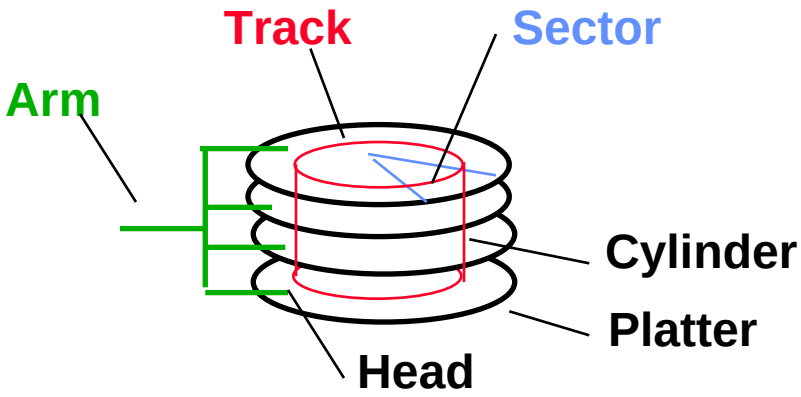
1. Each signal type has a system-defined default action.
   - abort and dump core (SIGSEGV, SIGBUS, etc.)
   - ignore, stop, exit, continue
2. A process may choose to *block* (inhibit) or *ignore* some signal types.
3. The process may choose to *catch* some signal types by specifying a (user mode) *handler* procedure.
   - specify alternate signal stack for handler to run on
   - system passes interrupted context to handler
   - handler may munge and/or return to interrupted context

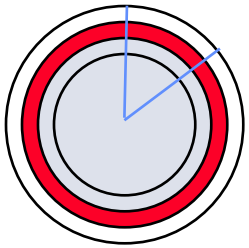
Sharing Disks

How should the OS mediate/virtualize/share the disk(s) among multiple users or programs?

- Safely
- Fairly
- Securely
- Efficiently
- Effectively
- Robustly

Rotational Media [2002]





Access time = seek time + rotational delay + transfer time

seek time = 5-15 milliseconds to move the disk arm and settle on a cylinder
rotational delay = 8 milliseconds for full rotation at 7200 RPM: average delay = 4 ms
transfer time = 1 millisecond for an 8KB block at 8 MB/s

Bandwidth utilization is less than 50% for any noncontiguous access at a block grain.

Unix Philosophy

Rule of Modularity: Write simple parts connected by clean interfaces.

Rule of Composition: Design programs to be connected to other programs.

Rule of Separation: Separate policy from mechanism; separate interfaces from engines.

Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.

Rule of Transparency: Design for visibility to make inspection and debugging easier.

Rule of Repair: When you must fail, fail noisily and as soon as possible

Rule of Extensibility: Design for the future, because it will be here sooner than you think.

Rule of Robustness: Robustness is the child of transparency and simplicity.

[Eric Raymond]

Unix Philosophy: Simplicity

Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.

Rule of Clarity: Clarity is better than cleverness.

Rule of Simplicity: Design for simplicity; add complexity only where you must.

Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.

Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.

Rule of Optimization: Prototype before polishing. Get it working before you optimize it.

[Eric Raymond]

Unix Philosophy: Interfaces

Rule of Least Surprise: In interface design, always do the least surprising thing.

Rule of Silence: When a program has nothing surprising to say, it should say nothing.

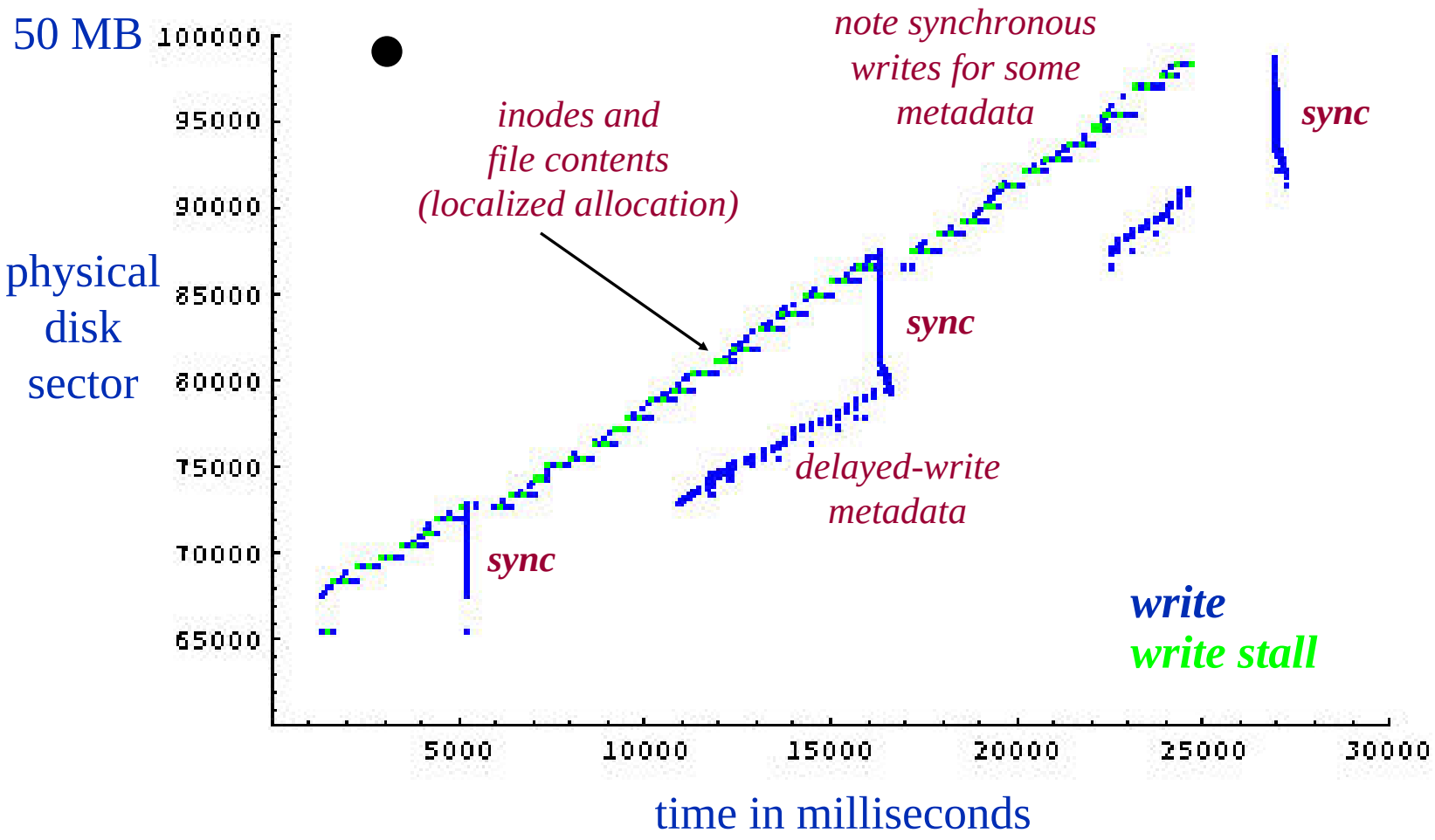
Rule of Diversity: Distrust all claims for “one true way”.

[Eric Raymond]

Worse is Better?

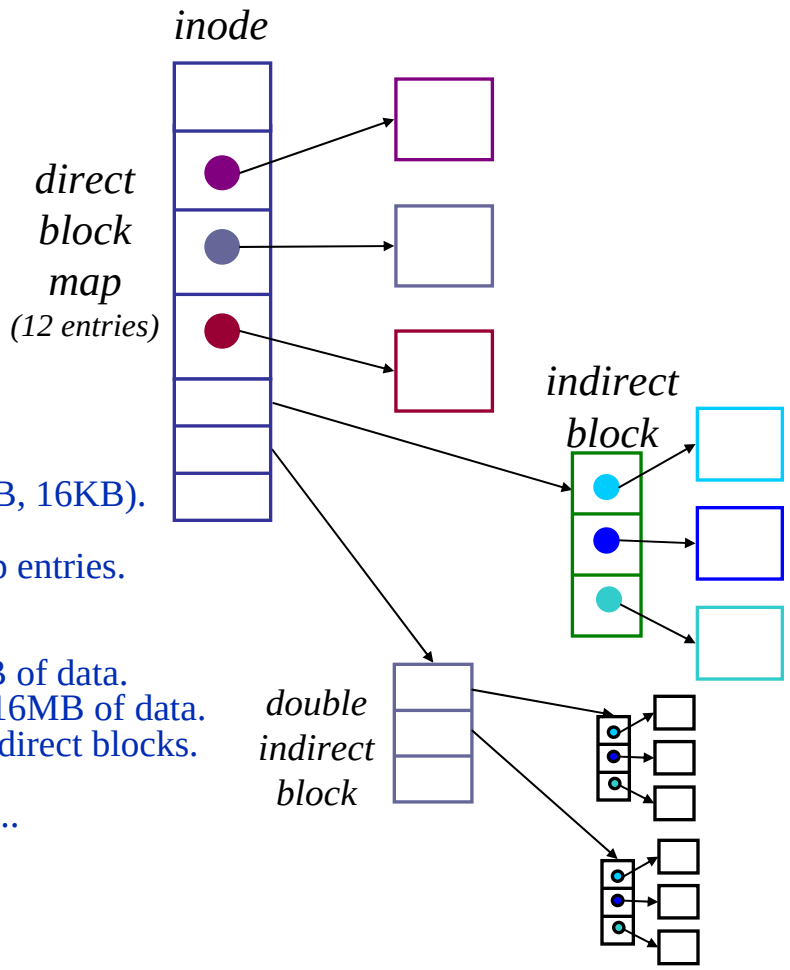
The following slides are "extra" slides that were not explicitly discussed in class. Based on what we did talk about, they should not be too mysterious, but don't worry about them for the midterm exam. We will plan to come back to them later.

Small-File Create Storm



Representing Large Files

UFS Inodes represent large files using a hierarchical block map, similar to a hierarchical page table.



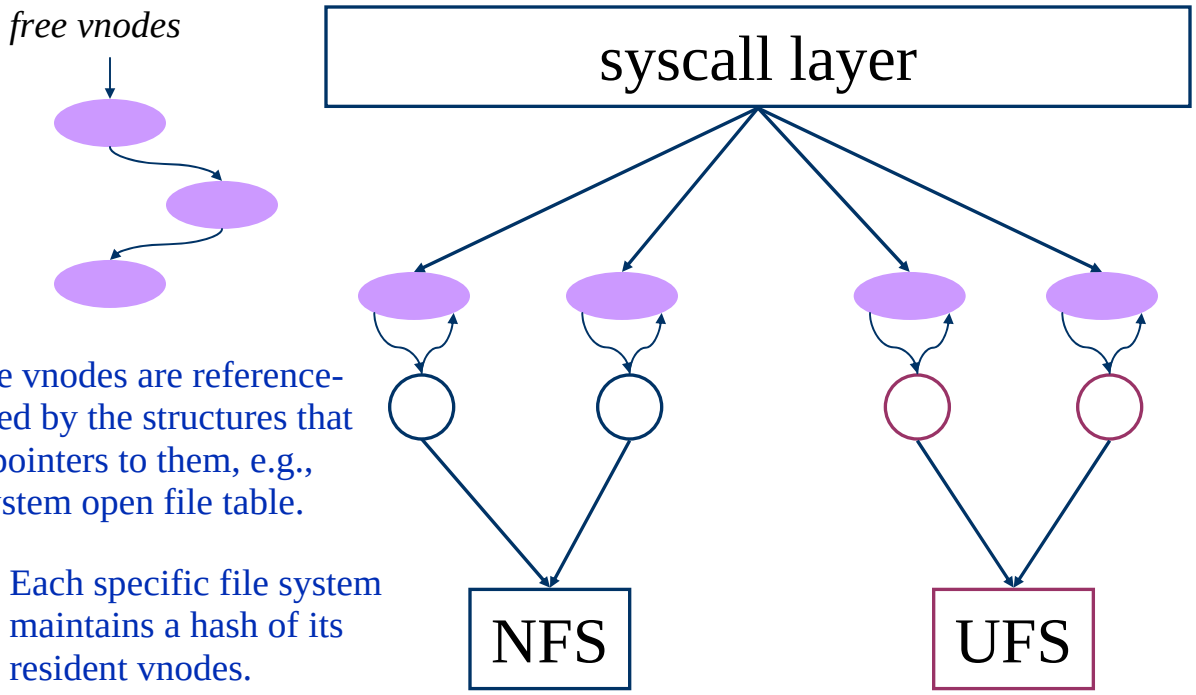
Each file system *block* is a clump of sectors (4KB, 8KB, 16KB).
Inodes are 128 bytes, packed into blocks.
Each inode has 68 bytes of attributes and 15 block map entries.

suppose block size = 8KB
12 direct block map entries in the inode can map 96KB of data.
One indirect block (referenced by the inode) can map 16MB of data.
One double indirect block pointer in inode maps 2K indirect blocks.

maximum file size is 96KB + 16MB + (2K*16MB) + ...

Vnodes

In the VFS framework, every file or directory in active use is represented by a *vnode* object in kernel memory.



Active vnodes are reference-counted by the structures that hold pointers to them, e.g., the system open file table.

Each specific file system maintains a hash of its resident vnodes.

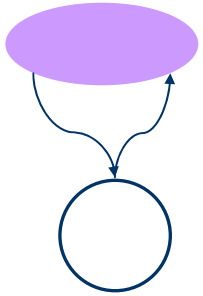
Each vnode has a standard *file attributes* struct.

Generic vnode points at filesystem-specific struct (e.g., *inode*, *rnode*), seen only by the filesystem.

Vnode operations are macros that vector to filesystem-specific procedures.

Vnode Operations and Attributes

vnode attributes (*vattr*)
type (VREG, VDIR, VLNK, etc.)
mode (9+ bits of permissions)
nlink (hard link count)
owner user ID
owner group ID
filesystem ID
unique file ID
file size (bytes and blocks)
access time
modify time
generation number



generic operations
vop_getattr (vattr)
vop_setattr (vattr)
vhold()
vholdrele()

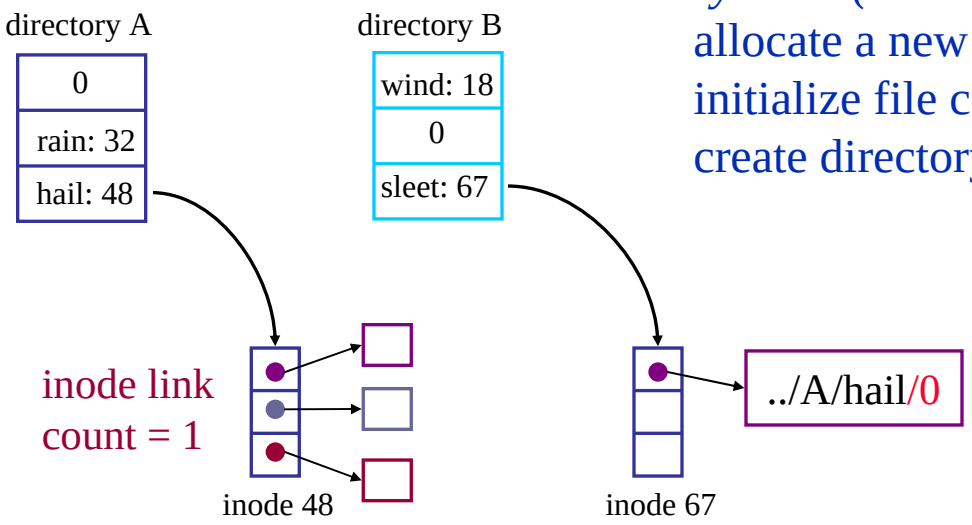
directories only
vop_lookup (OUT vpp, name)
vop_create (OUT vpp, name, vattr)
vop_remove (vp, name)
vop_link (vp, name)
vop_rename (vp, name, tdvp, tvp, name)
vop_mkdir (OUT vpp, name, vattr)
vop_rmdir (vp, name)
vop_symlink (OUT vpp, name, vattr, contents)
vop_readdir (uio, cookie)
vop_readlink (uio)

files only
vop_getpages (page**, count, offset)
vop_putpages (page**, count, sync, offset)
vop_fsync ()

Unix Symbolic (Soft) Links

Unix files may also be named by *symbolic (soft) links*.

- A soft link is a file containing a pathname of some other file.



symlink system call
symlink (existing name, new name)
allocate a new file (inode) with type symlink
initialize file contents with *existing name*
create directory entry for new file with *new name*

The target of the link may be removed at any time, leaving a dangling reference.

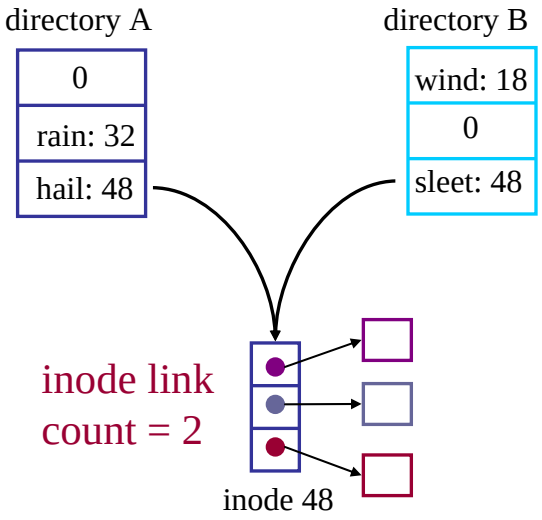
How should the kernel handle recursive soft links?

Unix File Naming (Hard Links)

A Unix file may have multiple names.

Each directory entry naming the file is called a *hard link*.

Each inode contains a *reference count* showing how many hard links name it.



link system call
link (existing name, new name)
create a new name for an existing file
increment inode link count

unlink system call ("remove")
unlink(name)
destroy directory entry
decrement inode link count
if count = 0 and file is not in active use
free blocks (recursively) and on-disk inode

Process Blocking with Sleep/Wakeup

A Unix process executing in kernel mode may block by calling the internal *sleep()* routine.

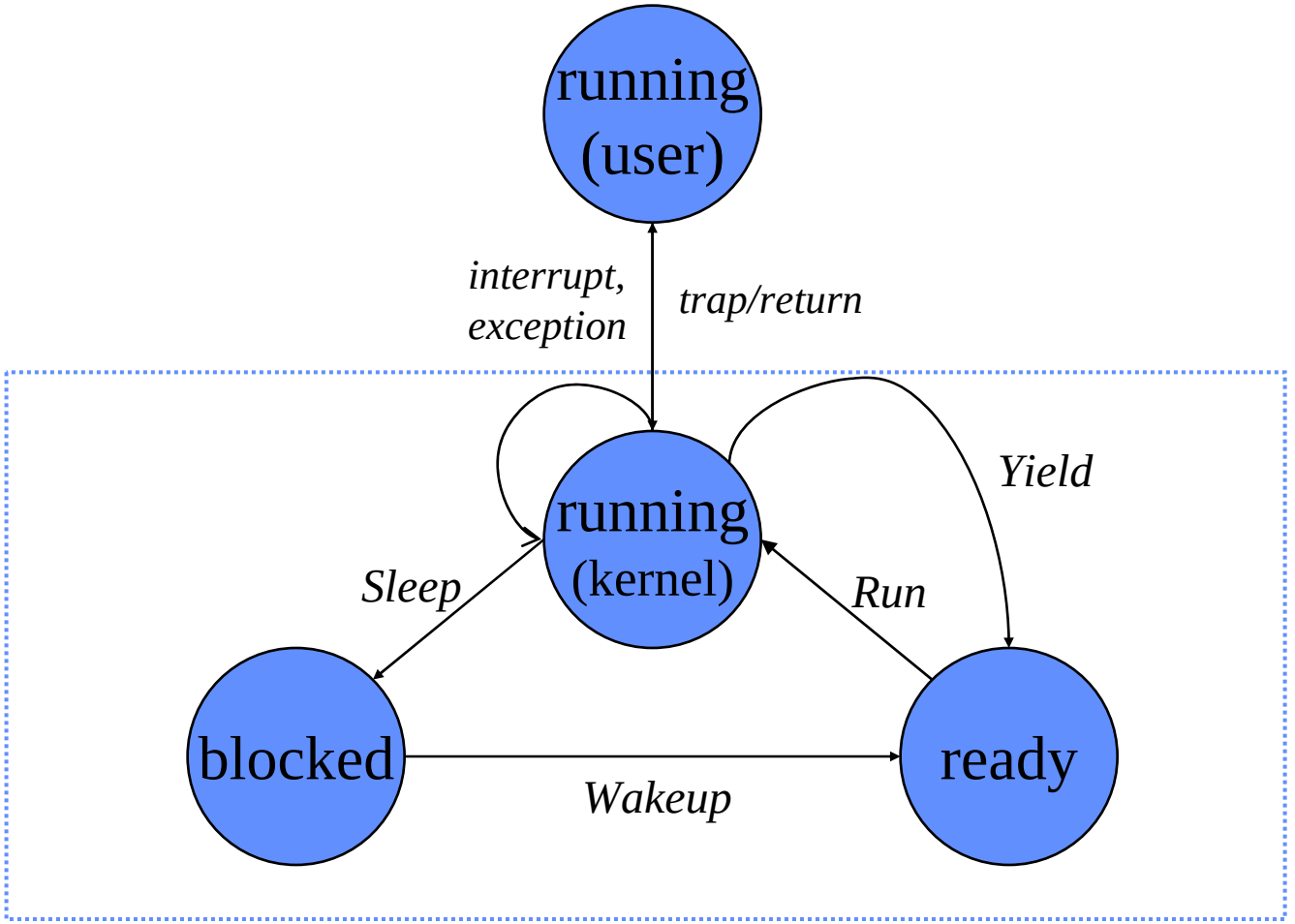
- wait for a specific event, represented by an address
- kernel suspends execution, switches to another ready process
- *wait** is the first example we've seen

  also: external input, I/O completion, elapsed time, etc.

Another process or interrupt handler may call *wakeup (event address)* to break the sleep.

- search sleep hash queues for processes waiting on *event*
- processes marked *runnable*, placed on internal run queue

Process States and Transitions

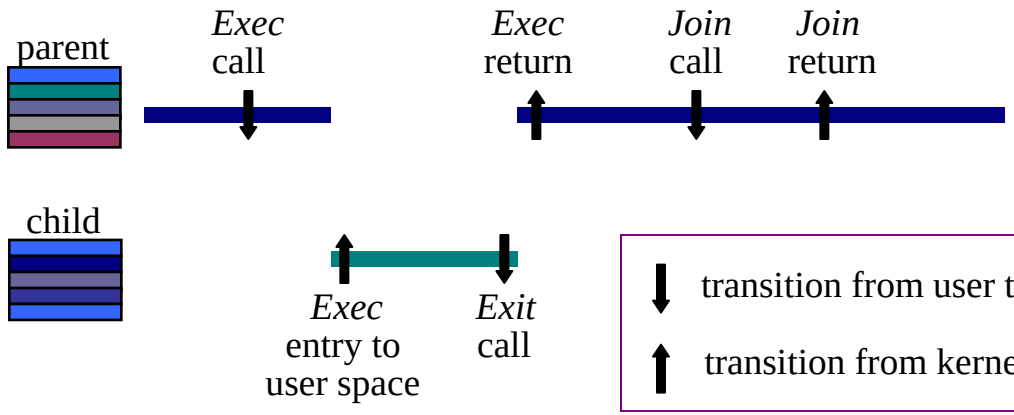


Mode Changes for Exec/Exit

Syscall traps and “returns” are not always paired.

- *Exec* “returns” (to child) from a trap that “never happened”
- *Exit* system call trap never returns
- system may switch processes between trap and return

In contrast, interrupts and returns are strictly paired.

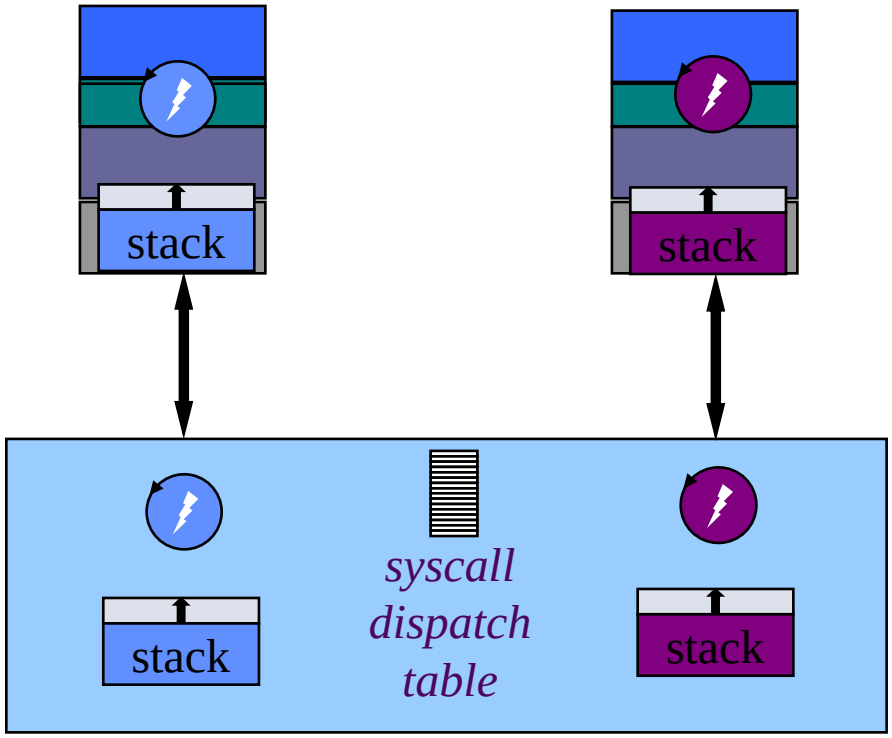


***Exec** enters the child by doctoring up a saved user context to “return” through.*

Kernel Stacks and Trap/Fault Handling

Processes execute user code on a ***user stack*** in the user portion of the process virtual address space.

Each process has a second ***kernel stack*** in kernel space (the kernel portion of the address space).

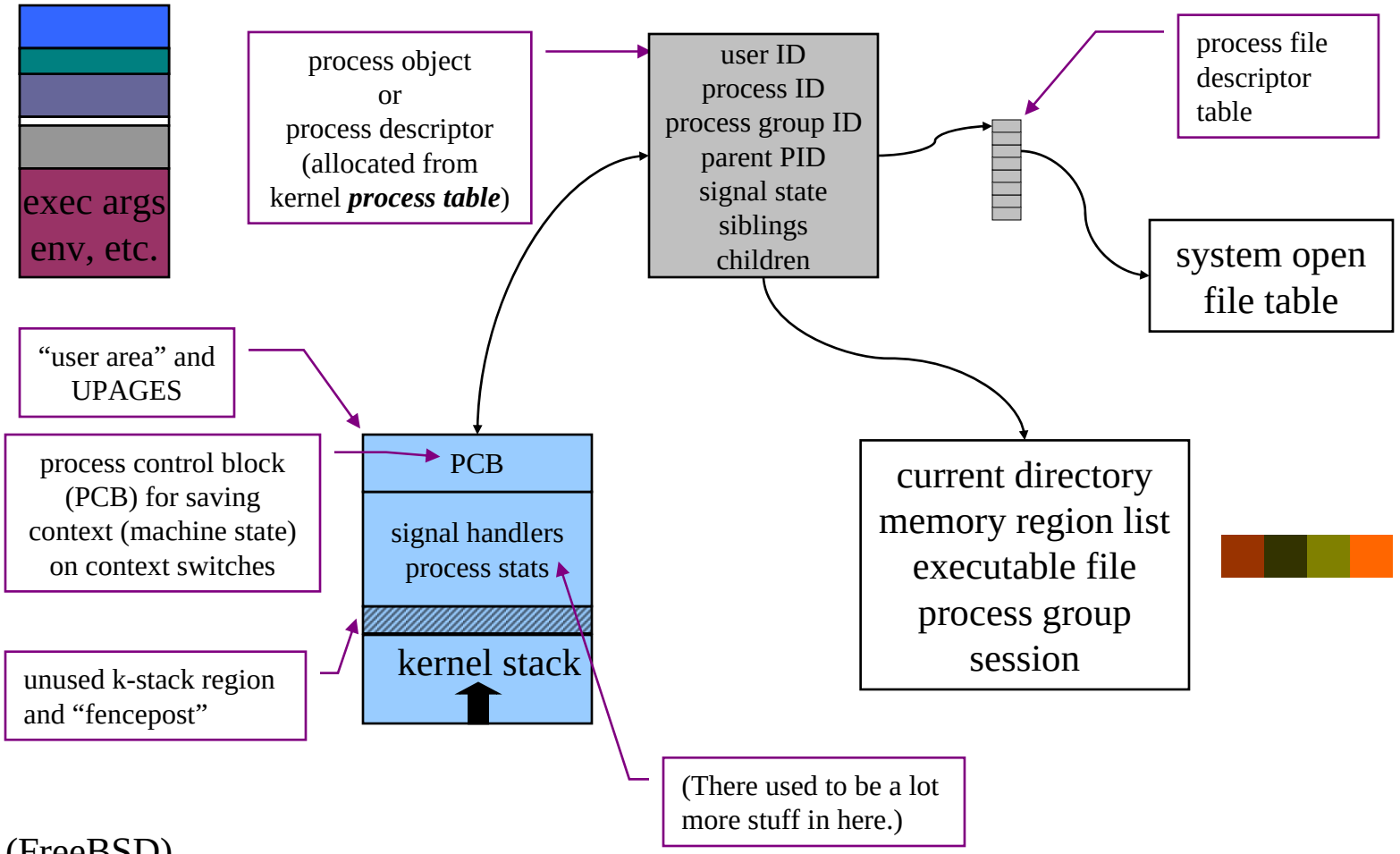


System calls and faults run in kernel mode on the process kernel stack.

System calls run in the process space, so *copyin* and *copyout* can access user memory.

The syscall trap handler makes an indirect call through the *system call dispatch* table to the handler for the specific system call.

Internal View of a Unix Process



(FreeBSD)

Implementation of Fork

1. Clone kernel state associated with the parent process.
    - Allocate new process descriptor and initialize it.
        - Copy key fields from parent process struct.
    - Increment reference counts on shared objects (e.g., VM regions).
2. Clone kernel stack, including kernel/user context.
    - Copy portions of stack, but just what we need.
        - How to address both user spaces at once (read from parent, write to child)?
    - Save parent context (registers) on child stack.
        - Are pointers in context and kernel stack valid in the child?
3. Mark child ready and (eventually) run it.
    - Enter child process by “returning” from *fork* syscall trap.

Safe Handling of Syscall Args/Results

1. Decode and validate by-value arguments.

   Process (stub) leaves arguments in registers or on the stack.

2. Validate by-reference (pointer) IN arguments.

   Validate user pointers and copy data into kernel memory with a special safe copy routine, e.g., *copyin()*.

3. Validate by-reference (pointer) OUT arguments.

   Copy OUT results into user memory with special safe copy routine, e.g., *copyout()*.

4. Set up registers with return value(s); return to user space.

   Stub may check to see if syscall failed, possibly raising a user program exception or storing the result in a variable.

Example: Mechanics of an Alpha Syscall Trap

1. *Machine* saves return address and switches to kernel stack.

   save user SP, global pointer(GP), PC on kernel stack

   set kernel mode and transfer to a syscall trap handler (*entSys*)

2. *Trap handler* saves software state, and dispatches.

   save some/all registers/arguments on process kernel stack

   vector to syscall routine through *sysent[v0: dispatchcode]*

3. Trap handler returns to user mode.

   when syscall routine returns, restore user register state

   execute privileged return-from-syscall instruction (*retsys*)

   machine restores SP, GP, PC and sets user mode

   emerges at user instruction following the *callsys*

Questions About System Call Handling

1. Why do we need special *copyin* and *copyout* routines?

   validate user addresses before using them

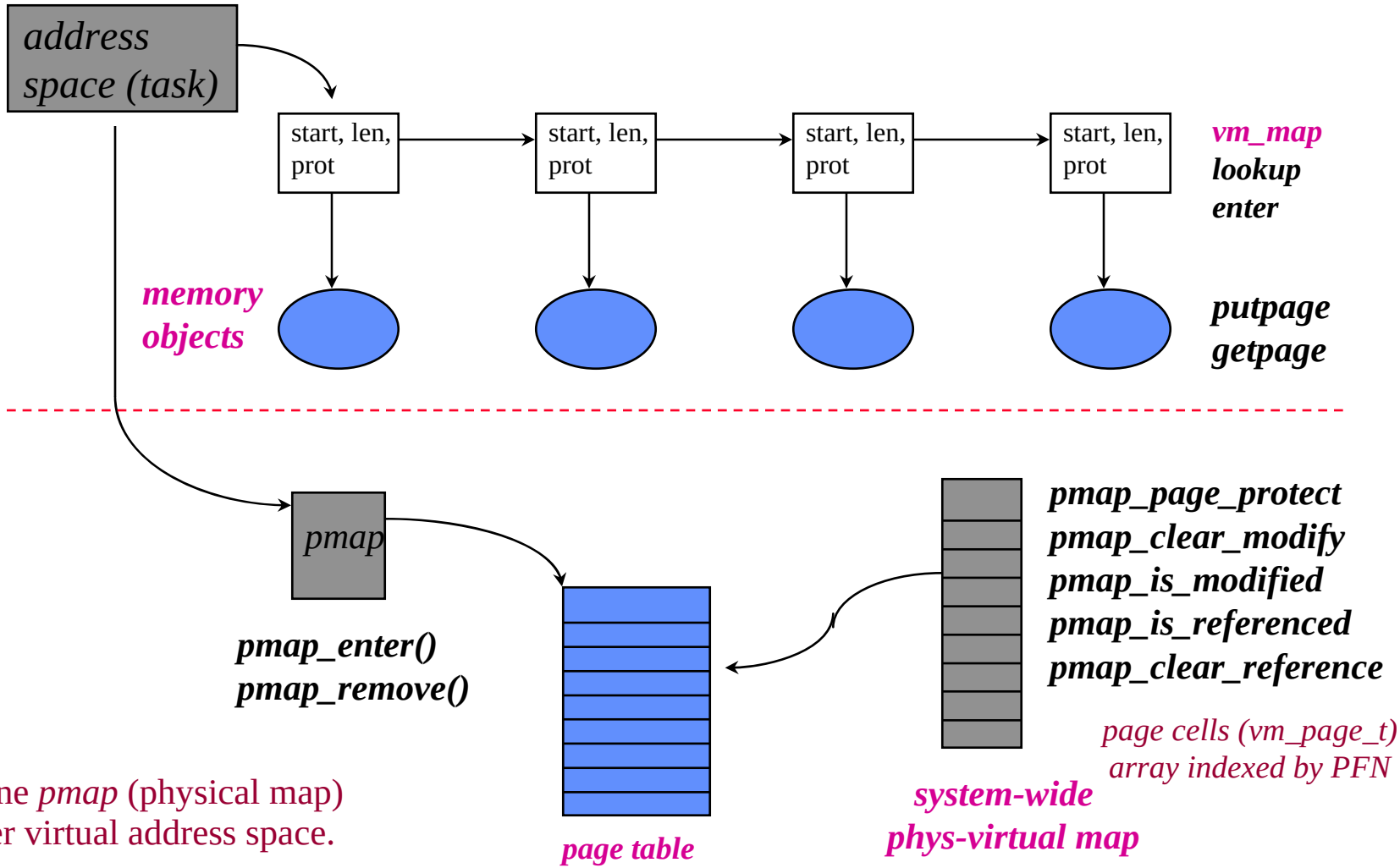
2. What would happen if the kernel did not save all registers?
3. Where should per-process kernel global variables reside?

   syscall arguments (consider size) and error code

4. What if the *kernel* executes a **callsys** instruction? What if user code executes a **retsys** instruction?
5. How to pass references to kernel objects as arguments or results to/from system calls?

   pointers? **No**: use integer *object handles* or *descriptors* (also sometimes called *capabilities).*

VM Internals: Mach/BSD Example



Performance-Driven OS Design

1. Design *implementations* to reduce costs of primitives to their architecturally imposed costs.

   Identify basic architectural operations in a primitive, and streamline away any fat surrounding them.

   "Deliver the hardware."

2. Design system *structures* to minimize the architecturally imposed costs of the basic primitives.

   If you can't make it cheap, don't use it as much.

3. Microbenchmarking is central to this process.

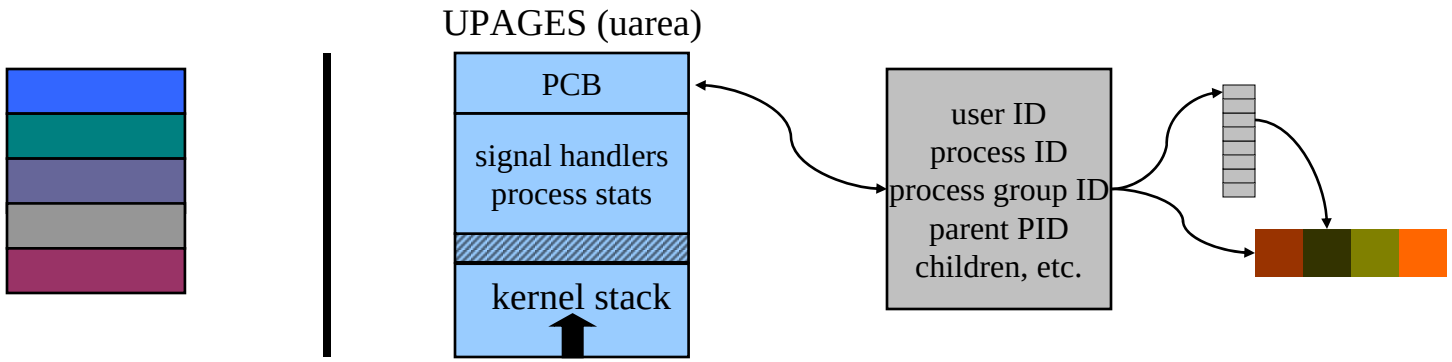
   [lmbench] is about microbenchmaking methodology.

Costs of Process Creation by *Fork*

- one syscall trap, two return-from-trap, one process switch.
- allocate UPAGES, copy/zero-fill UPAGES through alias
- initialize page table, copy one stack page (at least)

  fits in L2/L3? Depends on size of process?
- cache actions on UPAGES? for context switch?

  (consider virtually indexed writeback caches and TLB)



Costs of Exec

1. Deallocate process pages and translation table.

   TLB invalidate

2. Allocate/zero-fill and *copyin/copyout* the arguments and base stack frame.

3. Read executable file header and reset page translations.

   map executable file sections on VAS segments

4. Handle any dynamic linking.

   jump tables or load-time symbol resolution

Pathname Traversal

When a pathname is passed as an argument to a system call, the syscall layer must "convert it to a vnode".

Pathname traversal is a sequence of ***vop_lookup*** calls to descend the tree to the named file or directory.

```
open("/tmp/zot")
vp = get vnode for / (rootdir)
vp->vop_lookup(&cvp, "tmp");
vp = cvp;
vp->vop_lookup(&cvp, "zot");
```

Issues:

1. crossing mount points
2. obtaining root vnode (or current dir)
3. finding resident vnodes in memory
4. caching name->vnode translations
5. symbolic (soft) links
6. disk implementation of directories
7. locking/referencing to handle races with name create and delete operations

Delivering Signals

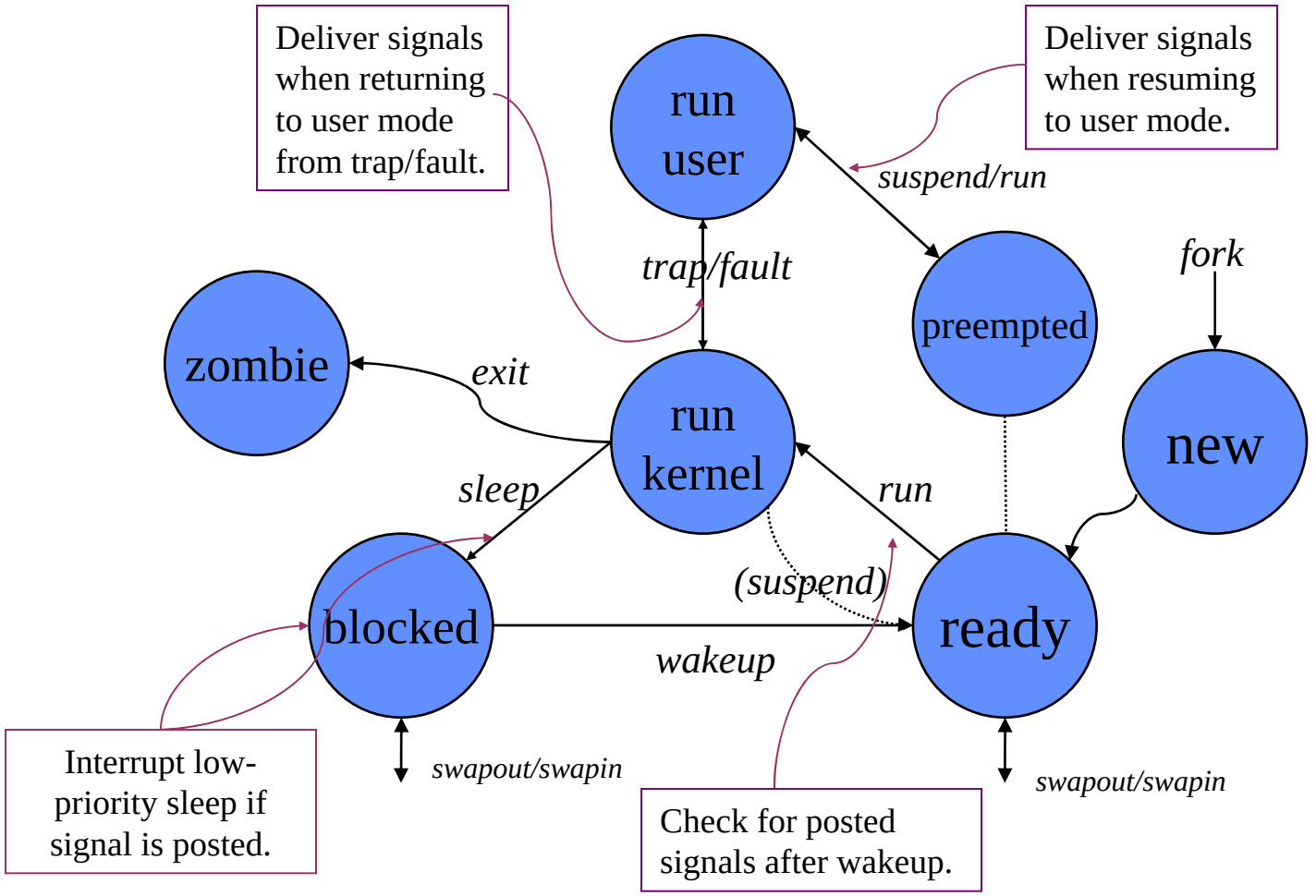
1. Signal delivery code always runs in the process context.
2. All processes have a *trampoline* instruction sequence installed in user-accessible memory.
3. Kernel delivers a signal by doctoring user context state to enter user mode in the trampoline sequence.

   First copies the trampoline stack frame out to the signal stack.

4. Trampoline sequence invokes the signal handler.
5. If the handler returns, trampoline returns control to kernel via *sigreturn* system call.

   Handler gets a *sigcontext* (machine state) as an arg; handler may modify the context before returning from the signal.

When to Deliver Signals?



Filesystems

Each file volume (*filesystem*) has a *type*, determined by its disk layout or the network protocol used to access it.

- *ufs (ffs)*, *lfs*, *nfs*, *rfs*, *cdfs*, etc.
- Filesystems are administered independently.

Modern systems also include “logical” pseudo-filesystems in the naming tree, accessible through the file syscalls.

- *procfs*: the */proc* filesystem allows access to process internals.
- *mfs*: the *memory file system* is a memory-based scratch store.

Processes access filesystems through common system calls.

Limitations of the Unix Process Model

The pure Unix model has several shortcomings/limitations:

- Any setup for a new process must be done in its context.
- Separated *Fork/Exec* is slow and/or complex to implement.

A more flexible process abstraction would expand the ability of a process to manage another externally.

> This is a hallmark of systems that support multiple operating system "personalities" (e.g., NT) and "microkernel" systems (e.g., Mach).

Pipes are limited to transferring linear byte streams between a pair of processes with a common ancestor.

> Richer IPC models are needed for complex software systems built as collections of separate programs.