UNIX Tools for Concurrent Programming

1. The *process* (old)
2. The *thread* (relatively new)

The Process, I

Abstractly, a program in execution

Multiple simultaneous processes allow multiple activities; e.g., compile one source file while editing another

CPU is shared by **preemptive multitasking**—the OS **scheduler** interrupts one process to schedule another

Switching (at computer time scale) yields "apparent parallelism" (at human time scale)—*true* concurrency is possible only with multiple CPUs

The Process, II

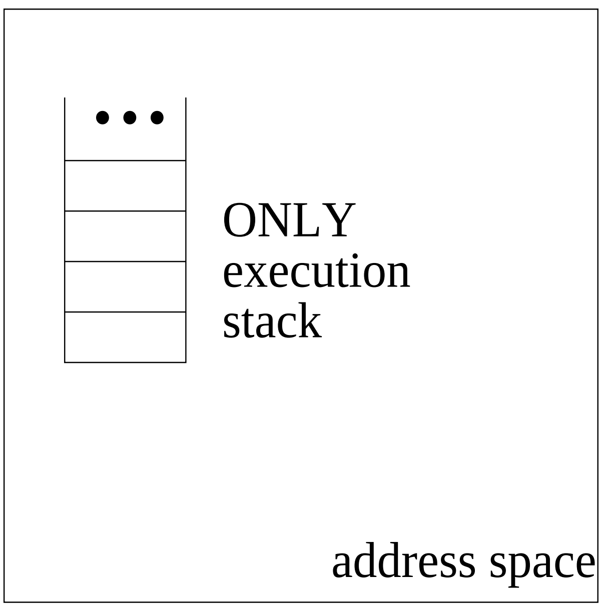
Each process kept separate from others by OS's intervention in addressing (enabled by specialized hardware)

Many-to-many relationship between process & program:

- 1 process can execute many programs during its lifetime
- And, of course, many processes can run the same program at the same time

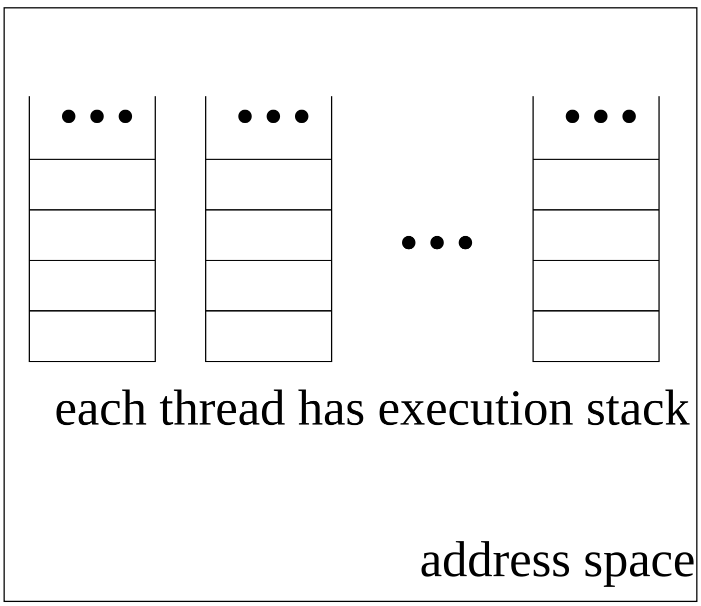
OSes now support concept of multiple **threads** within a process

Single-threaded Process



Process is the schedulable unit

Multiple Threads Within a Process



OS knows about threads

Thread is the schedulable unit

The Process, III

Process consists of:

- Process ID—unchanging, unique ID to distinguish it from all other processes
- Private *virtual address space*—set of memory locations process may access, and how it may access them (read-only, read/write, maybe some other possibilities)
- Instructions—from one or more executable files—placed into address space
- At least one **thread** of execution—unit of scheduling
- Set of “resources” (see below)

Process Resources

OS resources associated with process:

- Uid/gid, euid/egid, parent pid
- Current directory, current root
- Open files
- Signal state (which are being handled, which pending)
- Process group & current terminal/window (UNIX/Windows, respectively)
- Accounting info & scheduling parameters (OS priority, CPU time consumed)
- Address space mappings: indications of text, data, and stack segments

Thread Resources

Resources associated with thread:

- Registers (incl. program counter and stack pointer)
- Stack
- PSW (includes processor interrupt priority)

These are all hardware resources

Generally, OS abstract resources are assoc. w/ process, shared by all threads

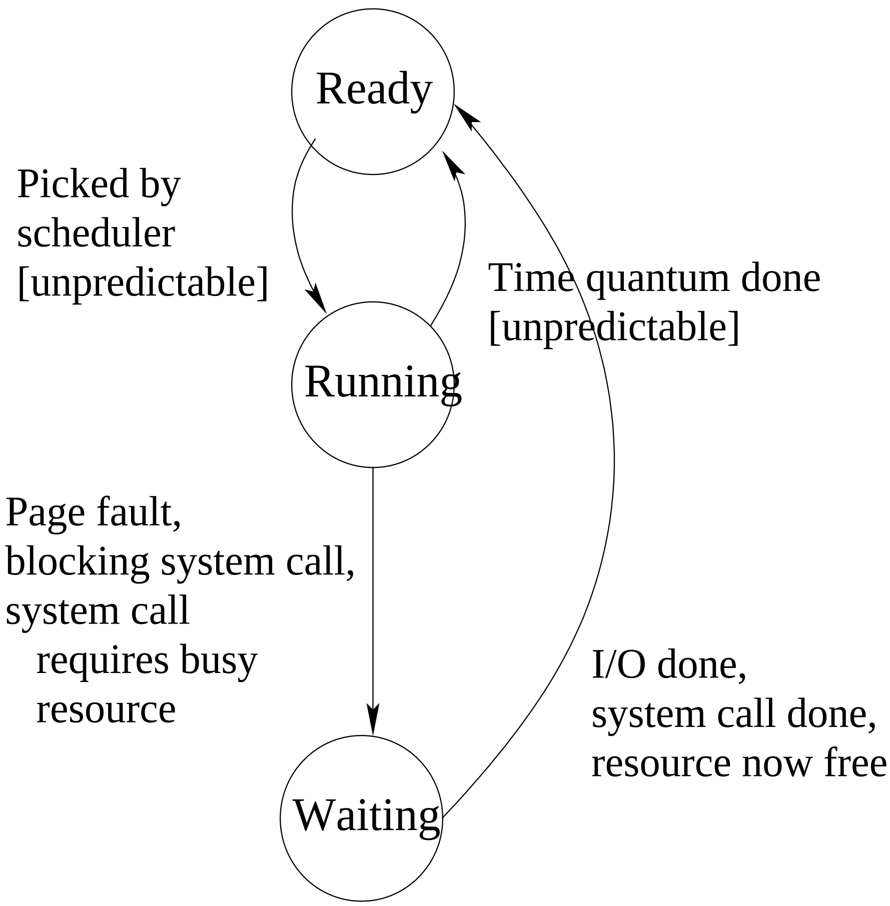
Thread States

Three major states:

1. Running (only one at a time on uniprocessor)
2. Ready
3. Waiting (for some resource)

"Waiting" is an abstraction—there are MANY wait states, one for each resource type

State Transitions



Process Creation, I

`fork()` makes near-exact copy of running process

`exec()` makes existing process run specified executable file from the beginning

`exec()` is a **loader**

Use of `fork()` and `exec()`:

```
pid_t child;
if ((child = fork()) < 0) {
   // ERROR:
   // errno indicates which error
} else if (child != 0) {
   // PARENT:
   // fork returns child’s pid to parent
   // parent knows child pid, but not its own
} else {
   // CHILD:
   // fork returns 0 to child
   // child typically calls exec soon
}
```

Process Creation, II

First few processes are specially created during OS initialization

Process 0, `swapper`, is the scheduler

Process 1, `/sbin/init`, processes `/etc/rc` files

Process 2, `pagedaemon`, plays role in virtual memory

`swapper` and `pagedaemon` are **kernel processes**—no executable file, not created by `fork`

Process Creation, III

Process creation example:

1. First, `init` forks shell, which interprets `/etc/rc` then dies; `init` waits for child death
2. Then, for each terminal line listed in `/etc/ttys`, `/sbin/init forks`

   Child `execs` a `/usr/libexec/getty` process, which offers “login:” prompt and waits for user input
3. When login name is given, `/usr/libexec/getty forks` a child, which `execs /usr/bin/login`, with login name passed as argument

   Parent `waits` for child exit then resumes listening to terminal line

Process Creation, IV

4. `/usr/bin/login` prompts for & reads password and compares to entry in `/etc/passwd` (or `/etc/master.passwd` or remote file)

   `/etc/passwd` contains: uid, encrypted passwd, name of user's shell

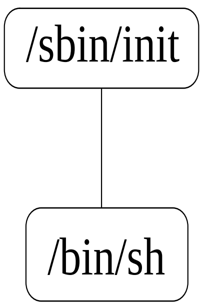
5. Assuming password matches, `/usr/bin/login` `execs` user's shell

6. Shell forks a process for each command (or command group)

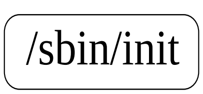
7. When shell exits, `getty` regains control and prints "login:"

Process Genealogy, I

Initially, `/sbin/init` is the only process

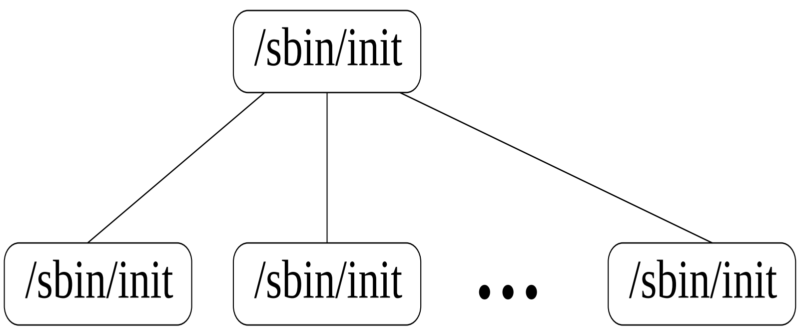


`init` forks and the child execs a shell (`/bin/sh`) which executes machine's boot scripts in directory `/etc/rc`



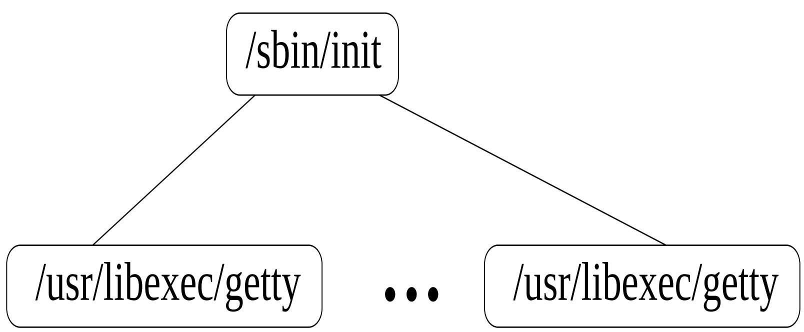
After the shell finishes, once again only `init` exists

Process Genealogy, II



`init` then forks one child for each terminal line

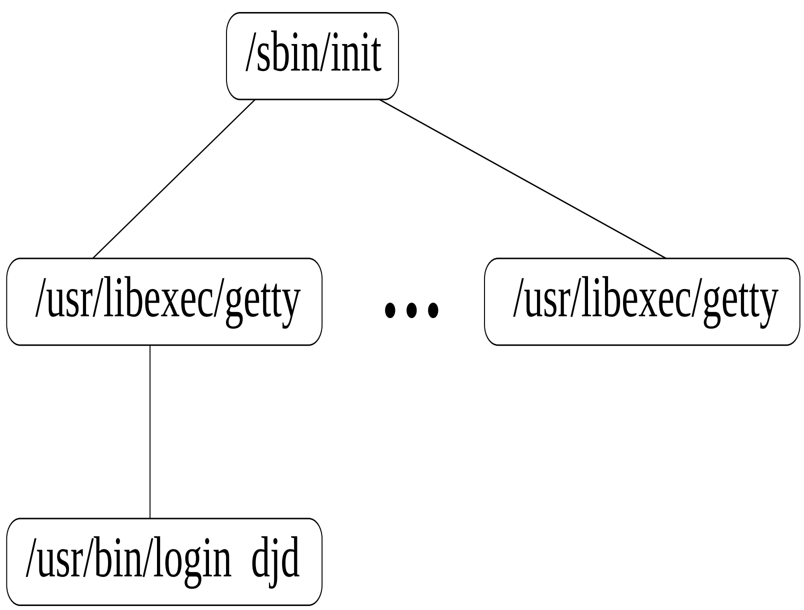
Process Genealogy, III



Each child then execs `/usr/libexec/getty`

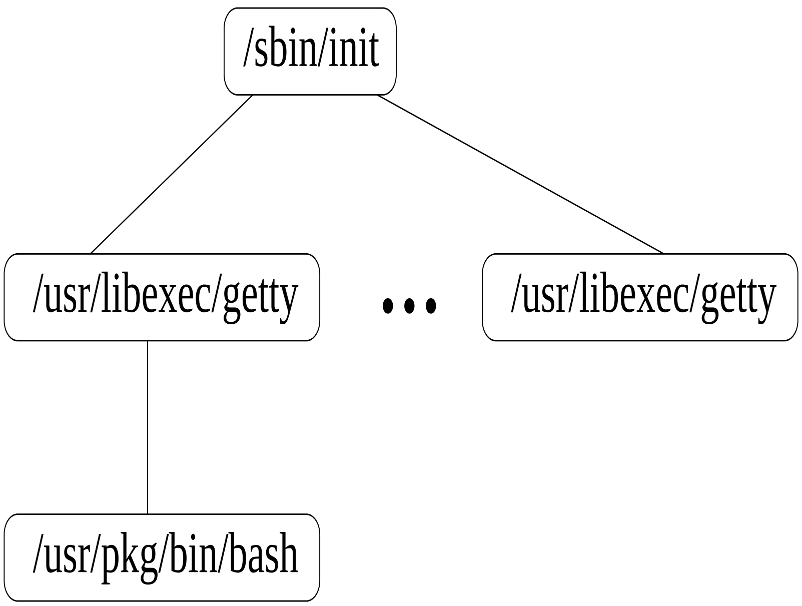
`getty` prints the familiar "`login:`" prompt

Process Genealogy, IV



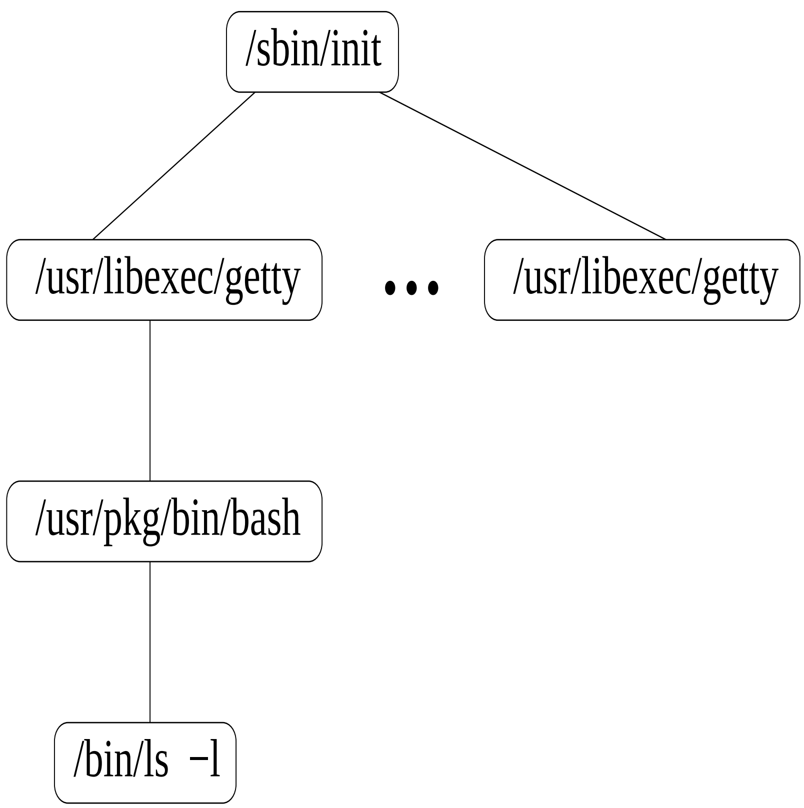
If user djd types his login name in response to the prompt, that line's `getty` will fork & child will exec `/usr/bin/login` with command line argument of `djd`

Process Genealogy, V



`login` prompts for a password; if the password is correct, `login` execs the user's preferred shell

Process Genealogy, VI



Shell then forks & execs for each program user runs

Process Creation, V

Q: Why have separate `fork()` and `exec()`?

A1: All-inclusive "spawn" call would have huge number of (complex) arguments, to cover (tiny fraction of) all possible initial states for new process. (Nevertheless, Windows does it this way.)

Between `fork` and `exec`, often initialize/restrict child; e.g.,

- Set up file descriptors (e.g., to establish parent-child pipe)
- Set uid, gid, process group
- Change signal handlers
- Set process resource limits

Process Creation, VI

A2: Sometimes important to have child start with exactly the same state as parent, for parallelism—can be virtually impossible to achieve with spawn semantics

A3: With copy-on-write, `fork` followed by `exec` no longer so expensive

Process Creation, VII

Inherited by child process:

- Real, effective, and saved UIDs and GIDs
- Environment
- Stack
- Memory
- Open file descriptors (file pointers are shared between parent and child)
- Close-on-exec flags for all fd's
- Signal handling settings
- Scheduler class, "nice" value
- Process group ID, session ID
- Current working directory
- Current root directory
- Default file mode creation mask (umask)
- Resource limits
- Controlling terminal
- Attached shared memory segments

Process Creation, VIII

Unique to child process:

- Process ID
- Different parent
- Own copy of file descriptors and directory streams
- Process, text, data, and other memory locks are NOT inherited
- Process times, in the tms struct
- Resource utilizations are set to 0
- Pending alarms are cleared
- Pending signals initialized to the empty set
- Timers created by timer_create not inherited
- Asynchronous input or output operations not inherited

Exec, I

`exec` was once UNIX's *only* loader system call

Now refers to `family` of similar system calls

E.g., in NetBSD: `execl`, `execlp`, `execle`, `exect`, `execv`, `execvp`

They differ only in how arguments are passed in

Exec, II

Maintained across the exec:

- Process ID and parent process ID
- Real user & group IP
- Supplementary group IDs
- Session ID
- Controlling terminal
- Time left until alarm
- cwd & curroot
- umask
- File locks
- Signal mask
- Pending signals
- Resource limits
- tms values
- Every open file for which `close-on-exec` flag is 0 (0 is default)

Exit

Difference between `_exit(2)` and `exit(3)` is that `exit(3)` ...

1. Flushes & closes open streams
2. Calls all `atexit(3)` functions
3. Unlinks temp files

... before calling `_exit(2)`

`_exit(2)` takes "exit status" argument

Parent `wait(2)`s to collect "termination status"

Termination status is combination of exit status PLUS indication of how process ended (normally or by signal)

Wait

If parent terminates before child: `init` is made child's parent

If child terminates before parent: it becomes **zombie**—OS saves its exit code so that parent can later `wait` for it

Typical Use of Fork/Exec/Wait

```
pid_t child;
int result;

if ((child = fork()) < 0) {
    /* error fork-ing */
} else if (child != 0) {
    (void) waitpid(child, &result, 0);
    printf("child returns %d\n", (result >> 8));
} else {
    if (execv(...) < 0) {
        /* error execv-ing */
    }

    /*
     * at this point, this program
     * is no longer running in the
     * child process -- the exec'ed
     * program is running
     */
}
```

Illustration: Initial Situation

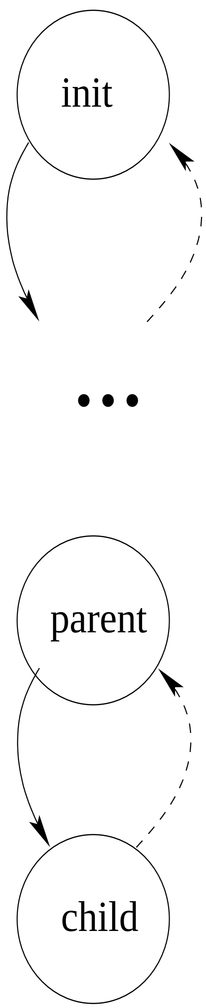
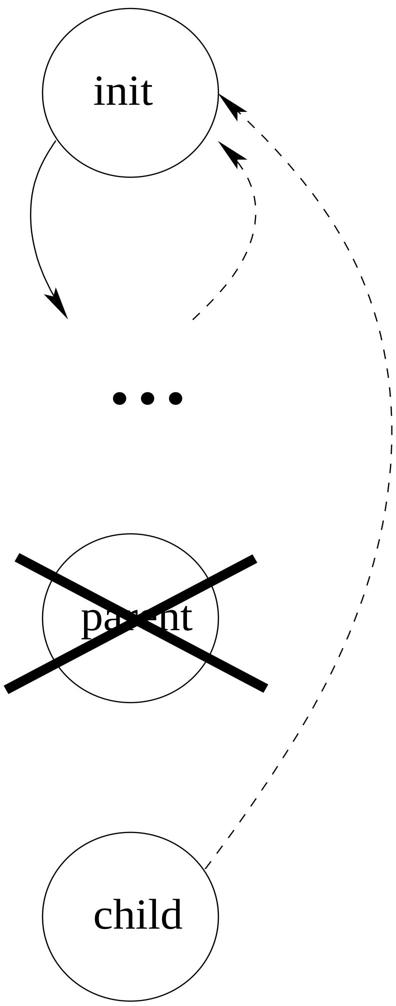
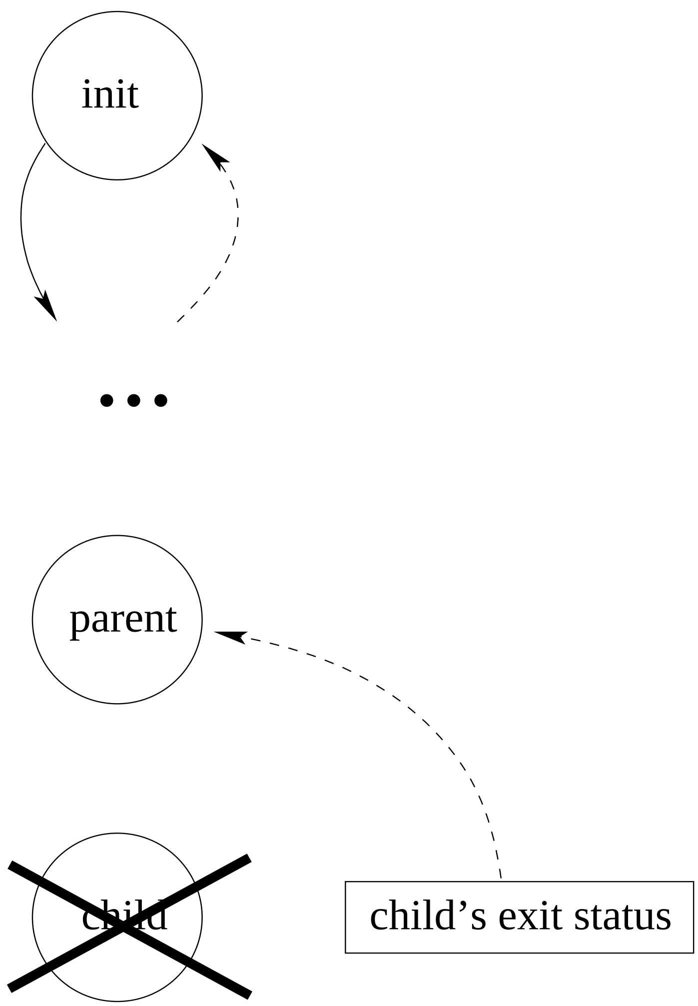


Illustration: Parent Dies Before Child



Orphaned process becomes child of `init`

Illustration: Child Dies Before Parent



Child exists "undead" so that it can return exist status when parent calls `wait`

Process IDs

Each process has 6 or more IDs:

- Real user & group IDs

  Who you really are; don't change

- Effective user & group IDs

  Determine file access; initially equal to "real" IDs; changed by exec-ing a program whose file is setuid or setgid

- Supplementary group IDs

  Up to 16 additional group IDs; specified in `/etc/group`

- Saved set-user-ID & set-group-ID

Setuid Concept, I

Consider “ping” program in `/sbin/ping`

Ping sends ICMP packets over “raw socket”—requires superuser privilege

Result of “`ls -l /sbin/ping`”

```
-r-sr-xr-x  1 root  wheel  27461 Dec  3  2004 /sbin/ping*
```

“`r-s`” means (1) executable and (2) setuid bit is set

Setuid Concept, II

Because setuid bit is set, ordinary user can exec `/sbin/ping` and *process will have superuser privilege* until it execs a non-setuid executable

During this time, effective uid is root but real uid remains unchanged

When process execs another program, effective uid becomes same as real uid

Multi-Process Communication & Coordination

Q: What do you mean by "communication" and "coordination"?

A: A concurrent multi-process program must include *communication* because separate processes have distinct address spaces

Coordination, a trait common to both multi-process and single-process-multi-threaded programs, is the mechanism of deciding which process executes when

I.e., "you go, OK now you go, ..."

UNIX Inter-Process Communication/Coordination Mechanisms

1. Pipe
2. FIFO, or *named pipe*
3. Socket
4. Message
5. Waiting on multiple descriptors
6. Signal
7. Shared memory

Pipe, I

`int pipe(fildes[2])`

`fildes[1]` is writable

`fildes[0]` is readable

Data written into `fildes[1]` can be read from `fildes[0]`

Pipe, II

Useful only among processes related as ancestor & descendant

- Child closes `fildes[0]`
- Parent closes `fildes[1]`

Result: child can write to parent

Named Pipe

FIFO, aka “named pipe”

`int mkfifo(char *name, mode_t mode)`

Unidirectional data flow like with a pipe: one process writes, another reads

But: has a name in file system name space—two *unrelated* processes can use FIFO, unlike pipe

Another unique feature: writes of up to `PIPE_BUF` bytes are *atomic*

Socket

`int socket(int domain, int type, int protocol)`

Bidirectional data flow

Meant for network communication

"Domain" argument selects network stack; e.g., TCP/IP

Message

Descriptors

Open pipe, FIFO, socket, file, device — each represented by **descriptor** (aka *file descriptor*

UNIX philosophy: as much as possible, represent every data source/sink with descriptor

Waiting on Multiple Descriptors

Common to wait on (typically: read from) multiple descriptors at once

E.g., server has sockets open with N clients, waits for first client input

Problem #1: `read(2)` takes only 1 descriptor argument

Problem #2: `read(2)` blocks when there is no input

Non-blocking I/O

Possible to make *descriptor* non-blocking

I.e., `read(2)` will NOT block if there is no input

(Nor will ANY other normally-blocking system call)

Q: How to make descriptor non-blocking?

A: There are 2 ways

Non-blocking Open

Provide `O_NONBLOCK` flag to `open(2)`:

```
int open(char *name, (O_NONBLOCK | ...), mode)
```


Non-blocking Fcntl

`fcntl(2)` is system call that allows manipulation of properties of descriptor

Provide `O_NONBLOCK` flag to `fcntl(2)`:

```
int fcntl(int fd, F_SETFL, (O_NONBLOCK | ...))
```

`F_GETFL` "command" argument GETS flags, `F_SETFL` "command" argument SETS flags

Can change state of descriptor during program using `fcntl`

Select

The other way to do non-blocking I/O ...

`select(2)` takes a SET of fds as argument

Actually, 3 sets: “read set,” “write set,” and “exception set”

`select(2)` also takes timeout argument—can wait for specified time or forever

UNIX File Descriptors

Descriptors ...

- Are small integers
- Form an *intra-process* name space: `open` (and `creat` and `socket`) are operations that **bind** an extra-process name (path name) to intra-process name
- Conventional meanings:

  0 = stdin
  1 = stdout
  2 = stderr

Descriptor Table

Descriptor is index into **per-process descriptor table**

Descriptor table entry:

1. Open mode (read-only, etc.)
2. "Special behavior" flags: no-delay, async, close-on-exec, append, etc.
3. Pointer to **global open file table**

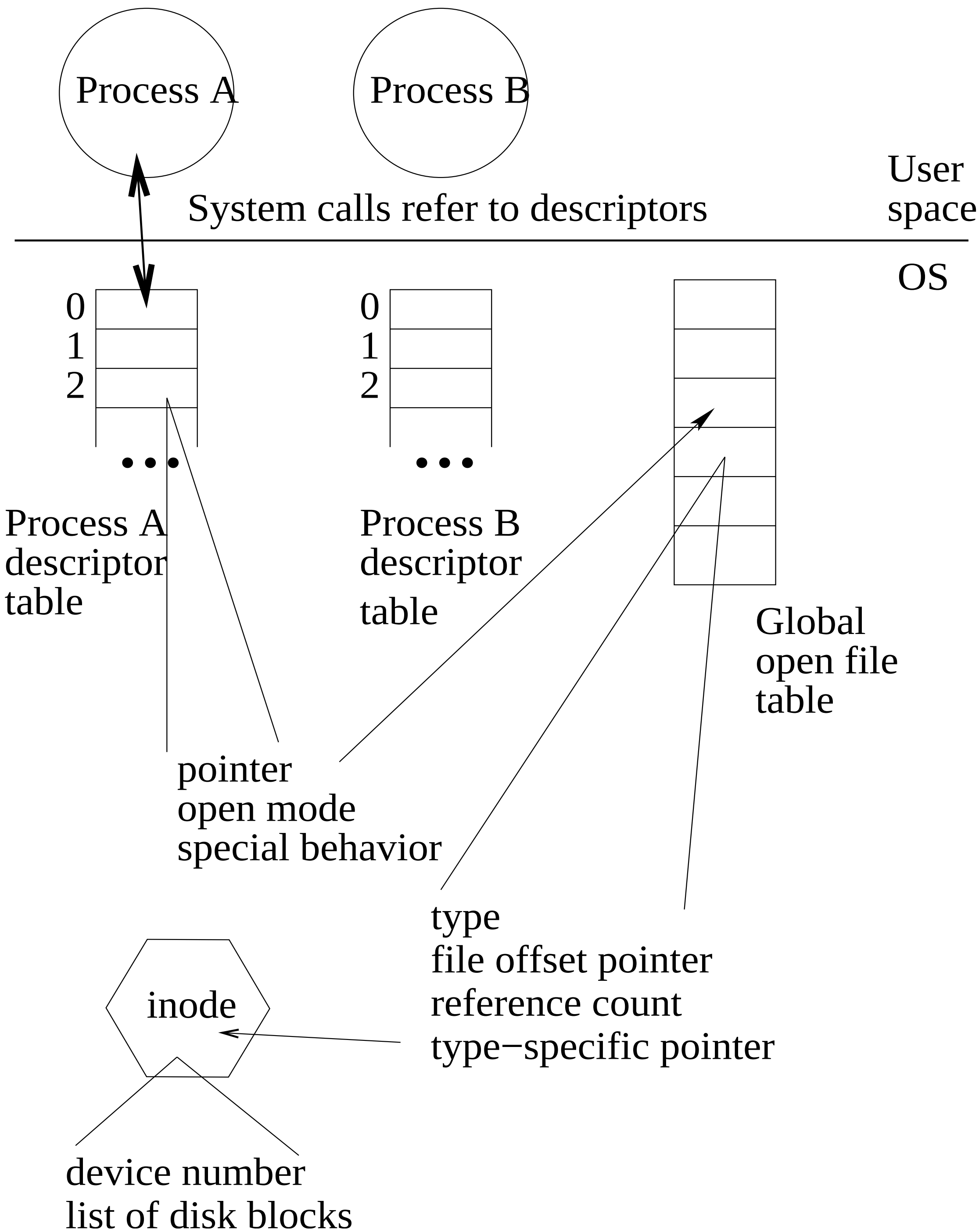
Global File Table Entry

1. **File offset pointer**
2. Reference count
3. Type of object (e.g., file, socket, etc.)
4. Pointer to object-specific record; e.g., inode, rnode, sock

GOFT entry created on every `open`

GOFT entry destroyed when reference count goes to 0

Illustration



File Offset Pointer

Pointer into file, indicates where next `read` or `write` will start

Set to 0 by `open`

Implicitly advanced by `read`, `write`

Explicitly changed with `lseek`

System Call Examples

Signature of `read`—no way to say where in file to read from

```
ssize_t read(int fd, void *buf, size_t count)
```

Signature of `lseek`—2nd and 3rd arguments say where in file to move file offset pointer

```
off_t lseek(int filedes, off_t offset, int whence)
```

- `whence=SEEK_SET` ⇒ go to byte number `offset`
- `whence=SEEK_CUR` ⇒ go to `offset` bytes beyond current pointer
- `whence=SEEK_END` ⇒ go to `offset` bytes beyond end of file

Reference Count

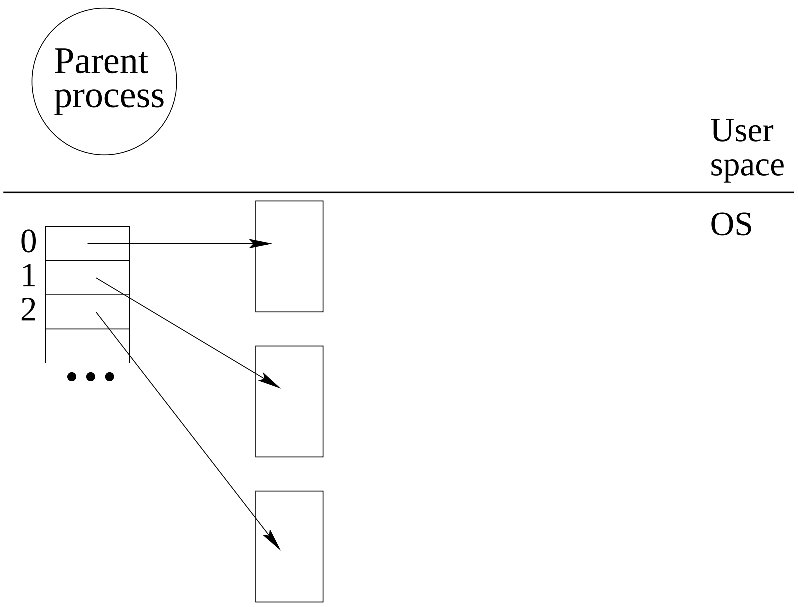
Incremented by `fork`, `dup`

Decremented by `close`, `_exit(2)` (implicit close), and `exec` (iff descriptor marked `close-on-exec`)

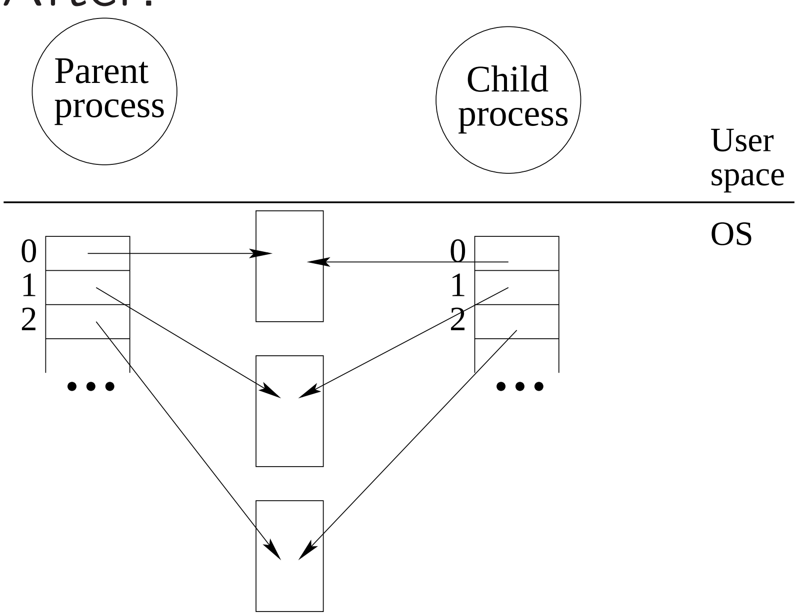
(Note difference between system call `_exit(2)` and library call `exit(3)`)

Fork

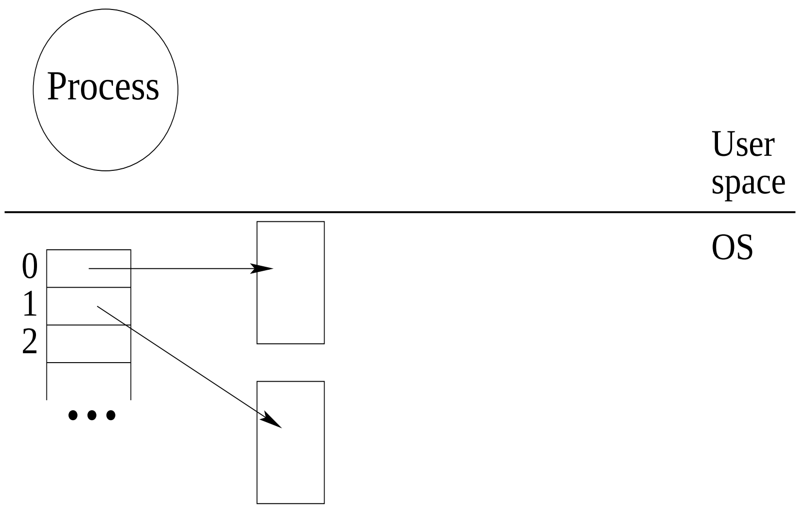
Before `fork(2)`:

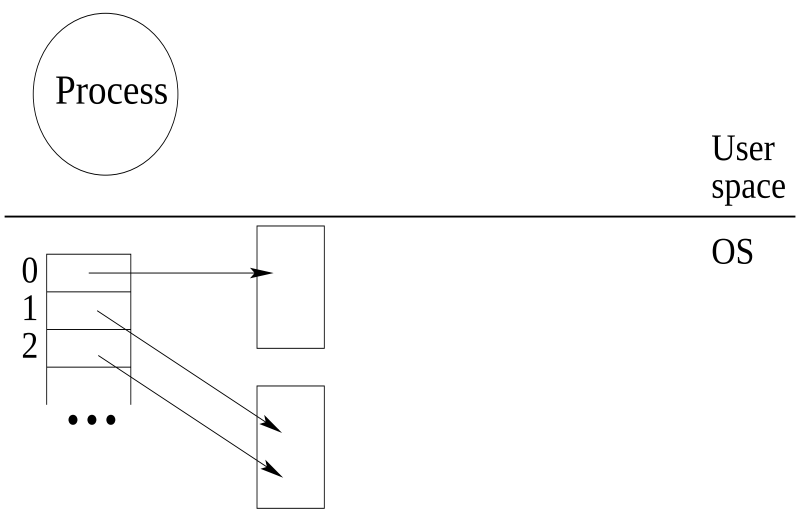


After:



Dup





Sharing a GOFT Entry

Possible for different processes to share same GOFT entry

How: `fork`

Possible for SAME process to have multiple descriptors point to same GOFT entry

How: `dup`

Object-Specific Record

Open file in local file system represented by `struct inode`, which includes

- Device number
- List of blocks

Open file in remote file system represented by `struct rnode`, which includes

- “File handle”
- Cached attributes
- Cached attributes lifetime

Open socket (for Internet communication) represented by `struct sock`

Descriptor vs. GOFT Entry

Multiple descriptors pointing to same *global file table entry* are created ...

- within single process: `dup`
- across separate processes: `fork`

Doing 2nd `open` of same file in same process creates new descriptor AND 2nd file table entry

Descriptors returned by `creat`, `open`, `dup`, `pipe`, `socket`, `socketpair`

Descriptors manipulated by `fcntl`

Why Two Similar Tables?

Q: Why have BOTH per-process descriptor & global file table entry? Can't they be one?

Answer:

1. Separate processes can share file pointer $\Rightarrow$ file pointer cannot be per-process
2. Separate programs can use same descriptor number for different purposes (e.g., different files, different open-modes for same file) $\Rightarrow$ descriptor table cannot be global

Synchrony

All above mechanisms are ways to transmit data

All above mechanisms are **synchronous**—reader makes specific system call then waits (if it chooses) until something is written

Synchrony provides the "you go, OK now you go, ..." coordination among readers/writers

Signal is **asynchronous** mechanism—receiver never knows when signal is coming

Signal

Signal indicated by a small integer—the only "data" received

To send signal:

```
int kill(int pid, int signum)
```

To receive signal, must register function with OS

OS will interrupt program (at any point) and run *signal handler* function

Some Terminology & Facts

Signal is **generated** somehow

Eventually it is **delivered** to target process

Between generation & delivery, signal is **pending**

Delivery can occur only when target is running

Signal number has corresponding symbolic name (beginning with "`SIG`") listed in `/usr/include/sys/signal.h`

E.g., `SIGFPE` ("floating point exception") is signal 8

How to Generate Signal

`kill(2)` system call

`kill(1)` program ... just calls `kill(2)` of course

Hardware exception (e.g., `SIGSEGV`, `SIGBUS`, `SIGFPE`, `SIGILL`)

OS condition (e.g., `SIGURG`, `SIGPIPE`)

Shell translates certain keys into `kill(2)`: `SIGINT`, `SIGQUIT`

Process can signal itself with `raise(3)`

Process can `SIGALRM` itself with `alarm(3)`/`setitimer(2)`

Signal Handling

Signal can be:

- Blocked—keep pending until signal unblocked
- Ignored—delivered & immediately dropped
- Handled—delivered & handled

To block: `sigprocmask(2)`

To ignore or handle: `sigaction(2)`

Sigprocmask, I

Recall: in computer science, a **mask** is a sequence of bits where each bit specifies some unique action/information

Signal set (“`sigset_t`”) is mask for all signals

There are several functions to manipulate signal sets

After calling manipulation functions, call `sigprocmask`:

```
int sigprocmask(int how,
                sigset_t new,   // really a pointer
                sigset_t *old)
```

Sigprocmask, II

“`how`” argument may be:

- `SIG_BLOCK`—add specified signals to those being blocked
- `SIG_UNBLOCK`—subtract specified signals from those being blocked
- `SIG_SETMASK`—specified signals are exactly those to be blocked

Some signals cannot be blocked (`SIGKILL`, `SIGSTOP`)

Sigaction, I

```
int sigaction(int signal,
              struct sigaction new,    // really a pointer
              struct sigaction *old)
```

Sigaction, II

`struct sigaction`:

```
void      (*sa_handler)(int);
sigset_t  sa_mask;
int       sa_flags;
void      (*sa_sigaction) (int, siginfo_t *, void *);
```

`void (*sa_handler)(int)` — function, `SIG_IGN` (ignore), or `SIG_DFL` (default)

`sigset_t sa_mask` — signals to block during execution of function

`int sa_flags` — various options

`void (*sa_sigaction)` — please ignore

Signal Delivery

Usually signal delivered *asynchronously*

Sometimes want *synchronous* delivery —
i.e., wait for signal

There are 3 ways:

1. `pause(3)` — don't use this!
2. `sigwait(2)`
3. `sigsuspend(2)`

The Problem with pause(3)

Signal can be "lost:"

1. Unblock signal
2. Signal delivered
3. `pause`(3) called — will never return

Programmer did not want #2 to happen between #1 and #3 but it did

Need **atomic** unblock-and-pause

Atomic Unblock+Pause

`sigwait` — wait for signal in set

`sigsuspend` — wait for signal NOT in set

`sigwait(sigset_t set, int *signal)`:

- Unblocks all signals in set
- Returns when one of them is delivered
- Out parameter indicates which

`sigsuspend(sigset_t mask)`:

- Save and replace blocked signal mask with argument mask
- Wait until some unblocked signal occurs
- Restore previous signal mask
- Return

Signals & Concurrency, I

Signals raise some of same issues as threads

Reason: signal handler is preemptively scheduled

For instance:

- Function is partially complete
- Signal is delivered & handled
- Handler calls same function, which runs to completion
- Function resumes and runs to completion

"Function" may be user-written or library

Signals & Concurrency, II

Only certain library functions can be called by signal handler!

These are “signal safe”

Prohibited: any function that accesses (reads or writes) static data

Prohibited: `malloc`/`free`, `fprintf`

Signals & Concurrency, III

Another point of contention: `errno`

Every system call potentially writes it

Example:

1. Program makes system call
2. As result, `errno`=12
3. Handler makes system call
4. As result, `errno`=3
5. Program examines `errno`, sees 3 instead of 12

Solution: handler should save & restore `errno`

Reentrancy

Code that is “signal safe” is **reentrant**

Reentrant code can be safely “re-entered”

This scenario is OK:

1. Function is partially complete

2. Signal is delivered & handled

3. Handler calls same function, which runs to completion

4. Function resumes and runs to completion

Handler “re-entered” the function in step #3

How to Write A Reentrant Function

1. Function only reads, never writes — not practical!

2. Function writes only activation-specific variables

In other words: function writes only local variables (allocated on stack), not (A) globals or (B) static locals

How to Spot A Non-Reentrant Function

In general, must understand function *implementation* to determine if it is reentrant

But some non-reentrant functions can be spotted from interface spec

Giveaway: returns pointer to static/global

Example: `asctime(3)`

There is now also `asctime_r(3)` — caller must supply buffer to accept return value

Others: `localtime(3)`, `gmtime(3)`, `ctime(3)`, `strtok(3)`, `readdir(3)`