Signals

Within the UNIX virtual computer (process) model, signals serve a role analogous to interrupts in a physical computer. They (can) cause execution of the user-mode program to be interrupted and transfer to a pre-established handler, just like vectored hardware interrupts. They can also be masked and unmasked.

Like hardware interrupts, signals can be said to be **synchronous** or **asynchronous**, if, respectively, the cause of the signal can or can not be associated with a particular instruction or operation that the process is currently executing. An example of an asynchronous signal would be an interactive user delivering SIGINT to the process by pressing Control-C on the terminal. An example of a synchronous signal is a SIGSEGV delivered to the process when it attempts an illegal memory access.

As we shall see in a later unit, signals may be viewed as taking place in two stages: **sending** (aka **generating** or **posting**) a signal, and then **delivery**, i.e. the action in the recipient process which results. Signals may be sent explicitly by one user process to one or more processes (including itself) via the `kill` system call. The kernel may also send different signals to a process as a result of errors or events, e.g (by no means a complete list):

- Illegal memory access: SIGSEGV or SIGBUS (sync)
- Illegal instruction opcode: SIGILL (sync)
- Hangup on controlling terminal session: SIGHUP (async)
- Keyboard interrupt from controlling terminal: SIGINT (async)
- Asynchronous I/O notification: SIGIO (async)
- Death of a child: SIGCHLD (async)

kill system call

A process can send any other process, including itself, any signal by using the `kill` system call:

```
kill(pid,sig)
```

The user id of the killer must match the recipient, i.e. you can't willy-nilly kill other folks' processes. However, the super-user (uid==0) may send any process a signal. Special values can be used for `pid` to do nifty things:

- pid==0: Send to all processes in the process group (to be covered in another unit)
- pid== -1: Kill all processes with user id equal to killer.
- pid== -n: (n!=1) Kill all processes in process group n.

A 0 value for `sig` doesn't actually send the signal, but checks whether it would have been possible to do so. It can be used to probe the existence of a process.

Pre-defined UNIX signal numbers and behaviors

UNIX defines dozens of signals, some system-dependent, which represent different conditions. Symbolic names, such as `SIGHUP`, are found by #includ'ing `<sys/signal.h>`. The assignment of signal numbers is kernel-specific. Although certain signal numbers such as 1, 2, 3, 9 and 15 are unlikely ever to change, good programmatic practice is to use the symbolic names at all times. Here are signals defined under Linux kernel 3.16, x86 architecture:

Sig#	Name	Def Act	Desc
1	SIGHUP	Terminate	Hangup on controlling tty
2	SIGINT	Terminate	Keyboard interrupt
3	SIGQUIT	Coredump	Keyboard abort
4	SIGILL	Coredump	Illegal Instruction
5	SIGTRAP	Coredump	Tracing/breakpoint trap
6	SIGABRT	Coredump	Abort (raised by abort(3))
7	SIGBUS	Coredump	Bus Error (bad memory access)
8	SIGFPE	Coredump	Floating Point/Math Exception
9	SIGKILL	Terminate	Non-blockable kill
10	SIGUSR1	Terminate	User definable
11	SIGSEGV	Coredump	Segmentation Violation (non-exist addr)
12	SIGUSR2	Terminate	User definable
13	SIGPIPE	Terminate	Write to a broken pipe
14	SIGALRM	Terminate	Alarm clock (see alarm(2))
15	SIGTERM	Terminate	Normal terminate request
16	SIGSTKFLT	Terminate	Co-processor stack fault (UNUSED)
17	SIGCHLD	Ignore	Child exit/status change
18	SIGCONT	Resume	Continue stopped process
19	SIGSTOP	Stop	Stop process (unblockable)
20	SIGTSTP	Stop	Keyboard stop
21	SIGTTIN	Stop	Stopped waiting for tty input
22	SIGTTOU	Stop	Stopped waiting for tty output
23	SIGURG	Ignore	Urgent data on socket
24	SIGXCPU	Coredump	Exceeded CPU time limit
25	SIGXFSZ	Coredump	Exceeded file size limit
26	SIGVTALRM	Terminate	Virtual alarm clock (see setitimer(2))
27	SIGPROF	Terminate	Profiling timer
28	SIGWINCH	Ignore	Tty window size change
29	SIGIO	Terminate	Signal-driven I/O (aka SIGPOLL)
30	SIGPWR	Ignore	Power going down
31	SIGSYS	Coredump	Bad system call #

To learn more about signals on your Linux platform, **man 7 signal**

Pending signals and signal masks

The kernel maintains two bitfields concerning signals for each process. Conceptually, these are arrays of boolean (1-bit) values. There is an element of this array for each valid signal number (signal 0 is never a valid signal, so this array conceptually starts at index 1). The *pending signals* array is set when a particular signal is posted to that process. Note that since this is a boolean array, not an integer array, the kernel does not "count" *how many* pending signals there are of a given signal number, just the presence or absence of such.

The other such (conceptual) array represents the *signal mask*, or the list of signal numbers which are currently blocked. When a signal is posted to a process but that signal is currently masked (blocked), the signal is not forgotten. The corresponding bit in the pending signals field is set. The signal is not delivered when it is blocked, but at the instant that the signal number is un-blocked, that signal will be delivered if pending. We will discuss the mechanisms for changing the signal mask shortly.

Signal delivery and disposition

It is indeterminate when the signal is delivered to the process, other than to say that it is some time after the signal has been sent. In the case of signals resulting from process faults, such as bad memory accesses, the signal is delivered immediately and the process does not continue to execute the faulting instruction. Blocked signals are not delivered at all while the signal is still blocked, and then are delivered at some indeterminate time after the signal becomes unblocked.

Upon signal delivery, the pending signal bit for that signal number is cleared. There are several basic outcomes or "dispositions". As we will see shortly, the process can establish the desired outcomes on a per-signal-number basis.

• **Terminate the process.** In some cases (determined by signal number, see table above), a **coredump** file is generated, which is a file containing the contents of memory. This file is named `core` and is created in the current working directory of the process, as if the process itself had opened this file for creation/writing and wrote the contents of memory to it. If the process does not have permission to create this file, it is not created. A resource limit (see the `ulimit` command or the `setrlimit` system call) can be set to limit the maximum size of the core file. The core dump file contains the contents of all relevant memory sections ("core" is an old term for main memory) at the time of termination, as well as the registers. This allows a debugger (such as gdb or adb) to reconstruct the call stack and the contents of all local and global variables, and determine where in the code the offense occured. The format of the core dump file is architecture-specific.

• **Ignore** the signal entirely. Certain signals (SIGKILL and SIGSTOP) can't be ignored.

• **Stop** the process. This is used in conjunction with **job control**, which allows a running

process or processes, attached to a user's terminal session, to be frozen, unfrozen, detached or re-attached to that terminal. Stopping the process is not the same as terminating it. Stop is the default action for certain signal numbers (see table above) which the kernel has assigned for job control.
• **Resume** the process, if stopped. This is the default action for `SIGCONT`, which is the only signal which can produce this action.
• **Handle** the signal with a specified user-level function call as detailed below.

Establishing signal disposition

Classically, the `signal` system call provides the means for controlling signal disposition, although it can not block or unblock signals.

```
void (*signal(int signum, void (*handler)(int)))(int);
```

This rather complicated prototype indicates that, for the signal `signum`, the handler will be set to the function `handler`. `signal` will return the previous value of the handler, which can be stashed away if it is desired to restore it later. On failure, `signal` returns the value `SIG_ERR`.

In addition to a bona-fide signal handler function address, `handler` can be:
• SIG_IGN: Ignore the signal or
• SIG_DFL: Reset to the default handling for this signal, as given in the table above.

These are numbers chosen so that they can't possibly be the valid address of a function.

The signal handler

A signal handler is an ordinary function. When a signal is delivered and the disposition of that signal number is to invoke a handler, it appears as if the handler function had been called from the instruction within the program that was executing when the signal was delivered. This is analogous to an interrupt service routine in responding to a hardware interrupt.

The signal handler function is called with a single integer parameter, the signal number that is being handled (using the `sigaction` interface, additional parameters can be passed. This is beyond the scope of this unit). The same handler function can be specified for multiple signals and this parameter can be used to disambiguate.

In most versions of UNIX, including Linux, the default behavior is that when signal *n* is delivered to a process and handled, *n* is temporarily added to the blocked signals mask upon entry to the handler, at the same time that the pending signal bit for *n* is cleared. This prevents the signal handler from being called re-entrantly. Upon return from the handler, the original blocked signal mask is restored. If the signal *n* was received while the handler's was executing, then the handler will be immediately re-invoked (but this is not re-entrant).

The signal handler function has a return type of void. When the signal handler function returns, execution resumes at the point of interruption. For certain synchronous signals, such as SIGILL, this may result in the same invalid operation being repeated endlessly, or other undefined results. Or, the signal handler may have taken action to correct the problem. The signal handler can also exit the program, possibly taking cleanup or recovery action. Yet a third commonly seen approach is the long jump:

setjmp/longjmp

Consider a command-line mathematical program. The main loop executes commands through a function `do_command`. Eventually, one of these commands leads, via a long chain of function calls, to:

```
long_compute()
{
        /* a very long calculation */
}
```

At some point during `long_compute()`, the user gets impatient and hits ˆC. We wouldn't want to abort the entire program and potentially lose unsaved data. We'd like to return to the command prompt. We would also like to avoid cluttering the code with frequent checks of some sort of global "abort" flag. Thankfully, C has a user-level mechanism known as "setjmp/longjmp". Another name for this is **non-local goto**.

```
#include <setjmp.h>

jmp_buf int_jb;              /*typedef'd in setjmp.h, really a char[] */

void int_handler(int sn)
{
        longjmp(int_jb,1); /* No need for & since it is an array */
}

main()
{
 char cmdbuf[BUFSIZ];
        (void)signal(SIGINT,int_handler);
        for(;;)
        {
                printf("Command> ");
                if (!fgets(cmdbuf,sizeof cmdbuf,stdin)) break;
                if (sigsetjmp(int_jb,1)!=0)
                {
                        printf("\nInterrupted\n");
                        continue;
                }
                do_command(cmdbuf);
```

```
        }
}
```

`setjmp`, from the program's standpoint, appears to be a function that is called once, and returns one or more times. `setjmp` stores context information, such as the program counter, stack pointer and register contents, into the `jmp_buf`, which is simply a `typedef` for an array of a sufficient size to hold this information. The program counter location stored is the address of the instruction following the call to setjmp. setjmp returns 0 when it is called (for the first time.)

A call to `longjmp` uses the context information stored in the supplied `jmp_buf` to return control to the setjmp point. The second argument will be the perceived return value from setjmp. It can not be 0, and if 0 is supplied the argument will silently be changed to 1. `longjmp` restores all of the saved register values, except the one used to indicate the return value from a function, which it sets to the supplied value.

The jump buffer must be visible both to setjmp and to longjmp, which means it must be a global variable, not a local variable. Because longjmp rolls back the stack frame to its state when setjmp was called, longjmp must be called from a function descended from the one in which setjmp was called. Otherwise, the stack frame may have already been overwritten by other function calls and sheer terror and chaos will reign.

In effect, longjmp provides the ability to `goto` across function boundaries. As is the case with `goto`, `setjmp/longjmp` is not an evil thing. Frequently it is the cleanest way to bail out of deep code.

Note that setjmp/longjmp are entirely a user-level mechanism. They are not system calls.

When a signal handler takes a longjmp, the restoration of the blocked signals mask does not take place. By using `sigsetjmp` instead, the signal mask is stored in the jump buffer, and is restored during longjmp. Without this, the code above would only work the first time the SIGINT was received. After that, SIGINT would continue to be masked.

Masking/Blocking signals

We will now explore how to manipulate the process's blocked signals mask. The block/unblock mechanism is most useful in protecting **critical regions** of code, where data structures are in an inconsistent state. For example, consider a routine to transfer a balance between accounts:

```
transfer_balance(a,b,n)
struct account *a,*b;
{
        a->balance-=n;
        b->balance+=n;
}
```

Every 24 hours, we want to prepare a summary report:

```
main()
```

```
{
          /*...*/
          signal(SIGALRM,alarm_handler);
          set_alarm();
          /*...*/
}

alarm_handler()
{
 struct account *p;
          signal(SIGALARM,alarm_handler);
          for (p=first_account;p;p=p->next)
                    printf("Account %s   Balance %d0,a->name,a->balance);
          set_alarm();
}

set_alarm()
{
          alarm(24*3600);
}
```

This uses the `alarm` system call to set an alarm which, at the specified number of seconds in the future, will cause delivery of SIGALRM to the process.

Consider what happens when the alarm signal arrives between the two statements of `transfer_balance`. The data structure is in an inconsistent state and the summary report is incorrect. We will examine these issues of concurrency and synchronization in greater detail in Unit 7. For now, the best fix is to block the alarm signal during the critical region:

```
transfer_balance(a,b,n)
struct account *a,*b;
{
 sigset_t set;
          sigemptyset(&set);
          sigaddset(&set,SIGALRM);
          sigprocmask(SIG_BLOCK,&set,NULL);
          a->balance-=n;
          b->balance+=n;
          sigprocmask(SIG_UNBLOCK,&set,NULL);
}
```

Each process has a bit field which represents which signals are blocked. The `sigprocmask` system call can be used to set bits in that mask, remove bits, set the entire mask at once, or query the current value of the mask. It uses a data type `sigset_t` to abstract the issue of how many bits are in that vector, and what data type has enough bits to hold it. For example, on the Linux kernel, where there are 64 signal numbers and when running on an architecture with 32 bit ints, it is defined in the system header files as:

```
typedef struct {
        unsigned long sig[_NSIG_WORDS];                  //this works out to [2]
} sigset_t;
```

If the SIGALRM signal arrives while it is blocked, it will stay in the pending signal set until the second `sigprocmask` system call removes the mask. Shortly thereafter the signal will be delivered and the handler will be invoked.

The sigaction interface

In the early years of UNIX, there were some inconsistencies in how the signal(2) system call functioned, in particular some UNIX versions reset the signal disposition of the signal in question upon entry to the handler. Because of the potential confusion, a new system call was introduced among all UNIX variants in the 1990s. The `sigaction` system call uses a more complicated interface, but can accomplish a lot more in a single call.

```
int sigaction(int sig,const struct sigaction *act, struct sigaction *oldact)

struct sigaction {

         void     (*sa_handler)(int);
         sigset_t sa_mask;
         int      sa_flags;
         /* Some more stuff...pretty complicated, read man page */
};
```

As with `signal`, `sa_handler` can be set to SIG_DFL to reset to default action, or SIG_IGN to ignore the signal, or the address of the signal handler function. `sa_flags` is a bitwise flags word that specifies a potpourri of options, including:

• SA_RESETHAND: Restore signal action to default when entering handler, aka unreliable signal semantics.

• SA_NODEFER: Do not automatically block signal while being handled. Aka SA_NOMASK

• SA_SIGINFO: Call the signal handler with 3 arguments. The first argument is the signal number. The second is a pointer to a `siginfo_t` structure which contains a lot of useful (and potentially architecture-specific) information about the cause of the signal. The third argument is a pointer to a `ucontext_t` structure which represents the context at the point in the program at which the signal was received. This argument is primarily for use in multi-threaded programs.

• SA_RESTART: Increases the likelihood that system calls interrupted by this signal and handled will restart when the handler returns. See discussion below.

Make sure to set sa_flags=0 if you don't want any flags. Otherwise the field may contain random garbage that will be mis-interpreted as various SA_XXX flags!

`sa_mask` describes which signals to block upon entry to the handler, in addition to the signal being handled itself, which is blocked unless SA_NODEFER has been specified. Be sure to set this correctly with `sigemptyset`, `sigaddset`, etc. otherwise extraneous signals may be masked during your handler's invocation.

Real-time signals

Signals were originally conceived as a way to terminate or control processes, hence the rather severe name `kill` for the system call to send a signal. Signals have been used as a primitive inter-process communications (IPC) mechanism, e.g. a daemon process which responds to a SIGHUP and re-reads its configuration files. However, the fact that traditional signals don't queue and carry no additional data makes them of limited use for IPC. In a later unit, we will see better ways for processes to send messages to each other.

On some more modern versions of UNIX, **real-time** signals are supported. These are numbered `SIGRTMIN` to `SIGRTMAX` and have no pre-defined meanings. Under the current Linux kernel, signals 32 through 63 are real-time signals. However signals 32-34 are used by the threads library, and should thus be avoided by other code. Therefore SIGRTMIN is defined as 35. Your mileage may vary, so always use the macros and avoid hard-coded signal numbers.

Unlike standard signals, real-time signals queue. If a process is sent, e.g. 5 instances of signal #35 while that signal is being blocked, upon un-blocking the signal, it will be delivered to the process 5 separate times. Furthermore, the real time signals are delivered in the order in which they were sent.

The `sigqueue` system call is similar to `kill`, but allows the sender of the signal to attach a small, opaque chunk of data:

```
int sigqueue(pid_t pid, int sig, union sigval value)

union sigval {
          int sival_int;
          void *sival_ptr;
}
```

The union will be the larger of the int size or the pointer size for that machine. The contents are passed along to the signal handler by the kernel and are opaque (the kernel does not interpret the union value). `sigqueue` could also be used to send a traditional signal (signal number < 32), but such a signal still won't queue. The opaque data will be delivered, but if multiple instances of the traditional (not real-time) signal occur, newer instances will overwrite the earlier data.

Interrupted System Calls

Certain system calls are designated as "long" calls. Generally, these are calls that may block for an indefinite period of time, for example, reading from the terminal or network connection. When an asynchronous signal arrives while the process is blocked in a "long" system call (of course a synchronous signal can not arrive during this point since the process is not executing instructions), and that signal is handled with a signal handler function, the system call is said to have been interrupted. Assuming the signal handler returns (it doesn't longjmp out or exit the process), the system call will return a failure with errno set to EINTR (Interrupted System Call).

Often, this is exactly what one wants. However, if it is necessary to make the signal handling perfectly transparent, the system call must be "restarted" when the signal handler returns and the process must go back to sleep until the system call finishes. To specify that behavior, use the SA_RESTART flag with `sigaction`.

System calls which support being restarted must be specifically coded to do so. When a system call is restarted after a signal, execution has gone from kernel mode (the interrupted system call), to the user-mode signal handler, and then back in to kernel mode, restoring the interrupted context of the system call (e.g. how many bytes had already been written).

The Linux kernel is somewhat inconsistent about restarting system calls. Certain system calls always restart, regardless of the SA_RESTART preference. Certain calls never restart and thus always give EINTR when interrupted, and certain calls pay attention to the SA_RESTART preference. Note that the SA_RESTART preference flag is off by default for all signals.

SIGCHLD

The `SIGCHLD` signal is somewhat special. By default, whenever a process terminates, a `SIGCHLD` signal is sent to its parent. The default disposition (SIG_DFL) of `SIGCHLD` is to ignore the signal, but this is not quite the same thing as explicitly ignoring the signal with `sigaction` or `signal`. The distinction has to do with compatability with different varieties of UNIX, some of which (including Linux) maintain that when `SIGCHLD` is explicitly ignored (disposition is set to SIG_IGN), it tells the kernel that the parent has no interest in the child, and thus a zombie will not be created.

Signal interactions with fork and exec

During a successful `exec` system call:
• The kernel clears the set of pending signals. The set of blocked signals remains the same.
• Any signals which were set to go to a signal handler are changed to `SIG_IGN`, because in the new executable, the address of the signal handler(s) would obviously be invalid.
• If the SIGCHLD signal was being explicitly ignored, its disposition is reset to the default (which is to implicitly ignore it).

After a `fork`, the set of pending signals for the child process is initialized to be empty. The parent is not affected. The signal disposition in the child and set of blocked signals is exactly the same as the parent.

Pipes

The UNIX **pipe** provides the fundamental means of interprocess communication, in accordance with the UNIX philosophy of providing small, flexible tools that can be combined to build up larger solutions.

A pipe is a uni-directional FIFO. It is created with the `pipe` system call:

```
main()
{
 int fds[2];
          if (pipe(fds)<0)
          {
                    perror("can’t create pipe");
                    return -1;
          }
```

Two new file descriptor numbers are allocated and returned via the two-element array `fds`. `fds[0]` is the read side of the pipe, `fds[1]` is the write side.

Although a pipe is accessed like a file, using file descriptors, there is no path in the filesystem which refers to the pipe. This means that processes which communicate via a pipe must be descended from a common process which created the pipe and passed along the file descriptors via the I/O redirection methods covered in Unit 3. This is a common application when spawning a pipeline of commands from a UNIX shell, in which all of the command processes are descended from the same shell process. In subsequent units, we will explore other pipe-like mechanisms that allow arbitrary processes to communicate, including processes on different host systems.

Pipe properties

• The pipe is uni-directional. An attempt to read from the write side, or write to the read side, will be greeted with EBADF. On some systems (e.g. Solaris), the pipe is bidirectional, however, this behavior should not be relied upon.
• The pipe is a FIFO. The order of the bytes written to the write side of the pipe will be strictly preserved when read from the read side. Data will never be lost, modified, duplicated or transferred out of sequence. However, if multiple processes are attempting to read from the same pipe, the distribution of data to each will be unpredictable.
• Message boundaries are not preserved by the pipe. If one process performs, e.g., 4 small writes to the pipe, and later a read is performed, all of the data will be returned as one large chunk. In order to perform record-oriented, as opposed to stream-oriented I/O, some sort of application-level record marking protocol needs to be employed (e.g. records delimited with newlines).

Flow Control

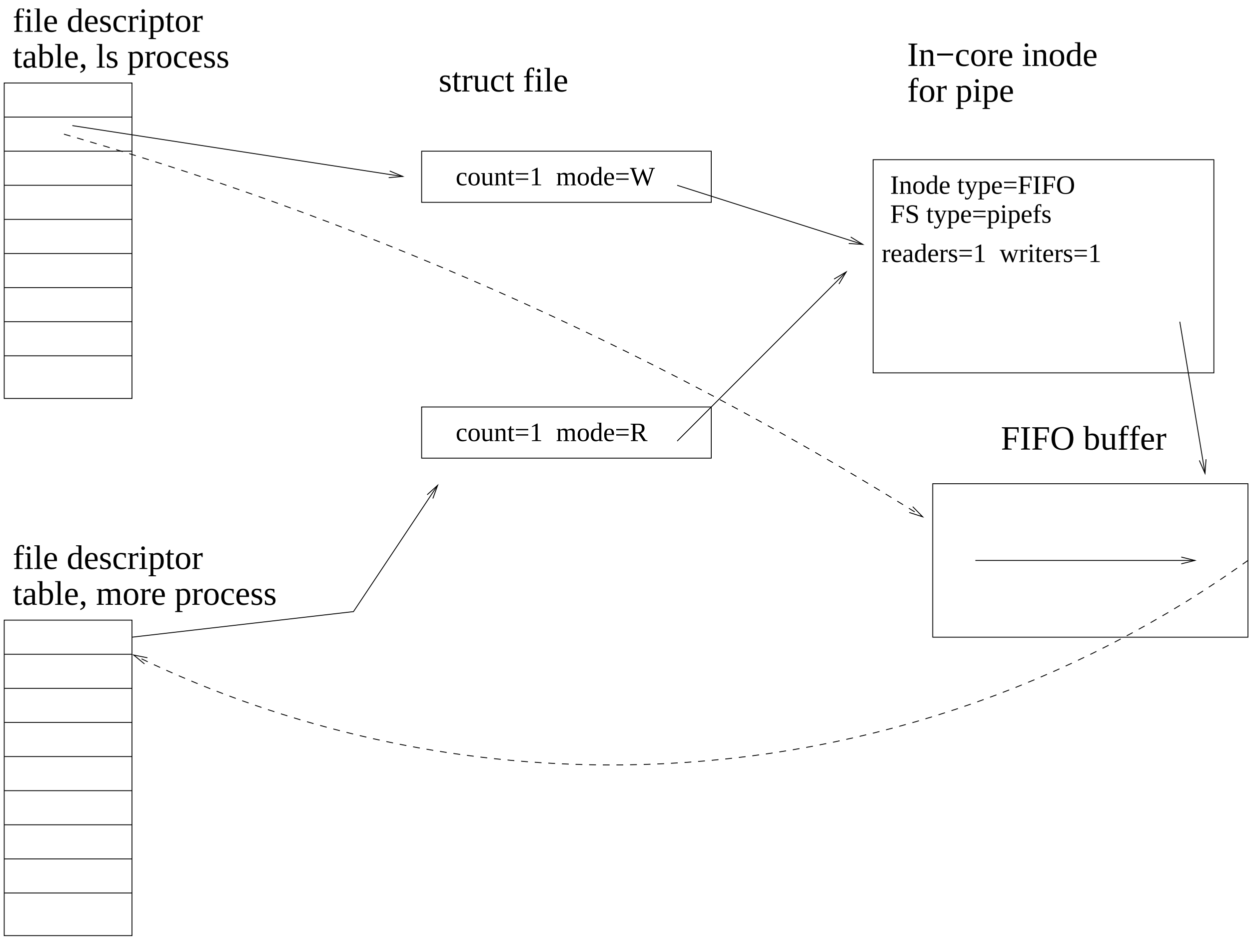
• When reading from a pipe, the `read` system call will block until there are some data available. It will then return any and all available data (up to limit specified by the caller as the third argument to read), and will not wait until an entire buffer full of data are available. [We are ignoring non-blocking I/O options for now.]
• Each pipe is a FIFO of a specific capacity, which is system-dependent and possibly controllable by the system administrator. A typical FIFO size is 16K. If there is insufficient room in the FIFO to handle the entire write request, the write will block until there is room. This provides the "back pressure" against the writing process to create "flow control" and throttle a fast writer piping to a slow reader. (Again, we are ignoring non-blocking I/O options at this time. Non-blocking behavior can be specified for pipes using the `fcntl` system call. This is beyond the scope of this unit.)

Atomicity

• If the write request size is 4K or less, the write will be handled **atomically**, and, in the event that there are multiple processes writing to the same pipe, these atomic writes will be preserved and not interleaved. However, if the write request size exceeds 4K, the write will be broken up into smaller chunks of 4K. In either case, the write blocks until it has completed entirely (i.e. a write system call of 65,536 bytes will not return until all 64K have made it into the pipe). writes to a pipe never return a "short" aka partial write.

Connection Control

• When the write side of the pipe closes (there are no more open file descriptors in any processes referring to it), this generates an EOF condition. Once any pending data have been read out of the FIFO, all subsequent reads will return 0.
• When the read side of the pipe closes, this condition is known as a **broken pipe**. Any pending data in the FIFO are discarded. An attempt to write to a broken pipe will result in the failure of the write system call with EPIPE. A signal (SIGPIPE) will also be delivered to the writer process. The default action of SIGPIPE is to terminate the process. This ensures that a program which is poorly written and does not error check the `write` system call doesn't continue to try to push data into the broken pipe forever.
• In order to preserve this behavior with respect to EOF and SIGPIPE/EPIPE, it is important to close any extraneous file descriptors that might be hanging on to the read or write side of a pipe, especially during dups and forks.
• An interesting "feature" that occurs with broken pipes is that if the pipe is partially filled and then the reader side is closed, this will **not** generate an error on `close` on the write side. This would appear to be a design flaw, in that data loss would be undetectable from the writing process's view. However, the presumption is that a parent process invoked both the reader and the writer and connected them with a pipeline. The abnormal exit of the reader process would be detected by the parent (through the `wait` system call) which would then report the failure of the pipeline as a whole.
• The figure below illustrates a writer process (ls) connected to a reader process (more) via a pipe.



Named Pipes

Pipes created with the `pipe` system call do not have any associated pathname in the filesystem namespace. This means that the only access is through the file descriptors; it is not possible to use `open` to gain access. Thus, the only way two processes can communicate using a pipe is if they share a common ancestor.

A **named pipe** is identical to a regular pipe, but it is accessed through a node in the filesystem having type `S_IFIFO`. A named pipe is sometimes called a FIFO. To create this special node one can use the `mknod` system call:

```
if (mknod("/tmp/namedpipe",S_IFIFO|0600,0)<0)
{
        perror("unable to create named pipe /tmp/namedpipe");
}

$ ./a.out
$ ls -l /tmp/namedpipe
```

```
prw-------    1 hak      root            0 Sep 25 23:12 /tmp/namedpipe
```

In the example above, the leading "p" in the ls output identifies the created inode as a named pipe (IFIFO). Note that the permissions mode is bitwise-OR'd.

The FIFO inode can also be created from the command prompt using the `mknod` or `mkfifo` command.

Semantics of named pipes are slightly different from anonymous pipes. Since the named pipe is opened with `open`, it can be opened read-only, write-only or read/write (i.e. `O_RDONLY`, `O_WRONLY`, `O_RDWR`). There can be more than one opening of the pipe too (i.e. multiple `struct file` in the kernel). When a named pipe is opened read-only, the open system call blocks until there is at least one instance of the same pipe being opened for writing (or read/write). Likewise, a write-only open blocks waiting for readers. A read/write open always succeeds immediately.

`read` from a named pipe returns whatever data are currently in the FIFO, or blocks if there are no data, just like anonymous pipes. If there are no writers, read returns 0 (after returning any pending data). But unlike an anonymous pipe, this EOF-like condition is not permanent. If another writer appears, future reads may return additional data. In the meantime, all reads would return 0. Because this condition results in a "spin loop," it is better practice to close the read side of the named pipe upon an EOF condition, and then re-open it. The open will then block until another writer comes along.

Likewise, a write to a named pipe with no readers results in either a `SIGPIPE` or `EPIPE`, just like anonymous pipes. Furthermore, the same kind of non-blocking options can be applied.

Named pipes can be used as a means of inter-process communication among unrelated processes on the same local system. On some systems, named pipes are synonymous with "UNIX-domain sockets", which will be covered in a later unit.