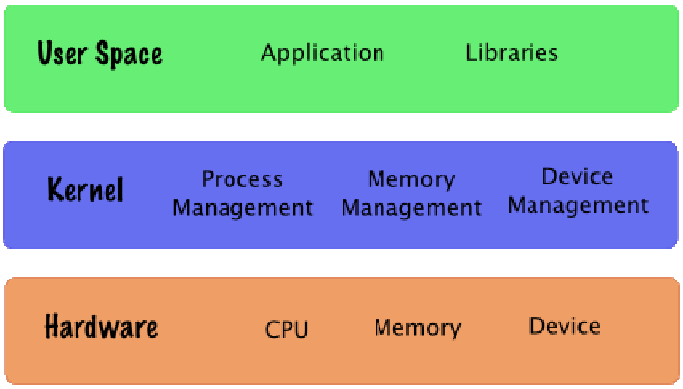
SIGNAL HANDLING

The Partition between Kernel and User Space

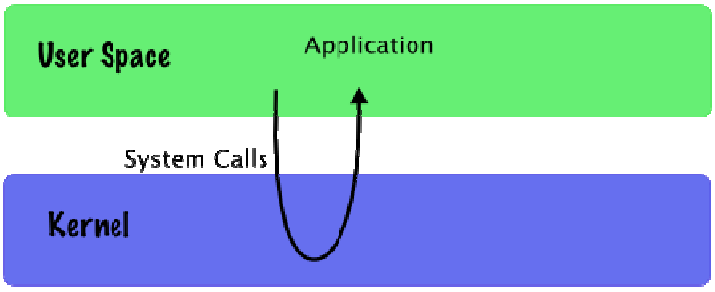
In the UNIX operating system, applications do not have direct access to the computer hardware (say a hard-drive). Applications have to request hardware access from a third-party that mediates all access to computer resources, the Kernel.



System Calls

The only way for an application in User Space to *explicitly* trigger a switch to Kernel Mode is to issue a *system call*.

Each system call provides a basic operation such as opening a file, getting the current time, creating a new process, or reading a character. System calls essentially are *synchronous calls to the operating system*.

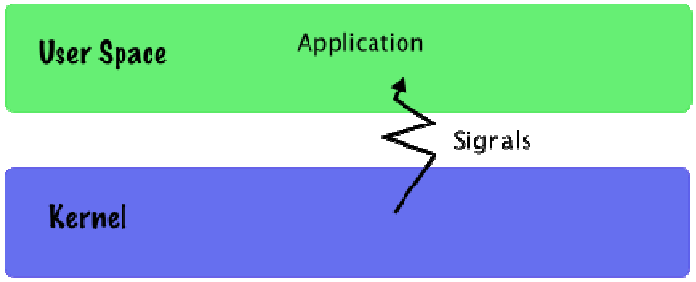


Signals

Signals provide a mechanism for allowing 'events' external to the program (typically interrupts) being passed to the program.

Signals offer another way to transition between Kernel and User Space. While system call are *synchronous* calls originating *from User Space*, signals are *asynchronous* messages coming *from Kernel space*. Signals are always *delivered* by the Kernel but they can be *initiated* by:

- **other processes** on the system (using the `kill` command/system call)
- **the process itself** : when a program executes an illegal instruction, such as dividing a number by zero, the hardware detects it and a signal is sent to the faulty program.
- **the Kernel** : when a program sets a system alarm, the Kernel sends a signal to the process every time a timer expires (e.g. every 10 seconds).



The overall dynamic for signal delivery is quite simple:

1. When a process receives a signal that is not ignored, the program immediately interrupts its current execution flow.
2. Control is then transferred to a dedicated *signal handler*, a custom one defined by the process or the system default.
3. Once the signal handler completes, the program resumes where it was originally interrupted.

Syntax :

```
#include <signal.h>

        :

signal(SIGNAL_NAME, signal_handler_function) ;
```

Example :

```
signal (SIGINT, sig_handler) ;
```

Example :

```
// SIGNAL FOR DIVISION BY ZERO ERROR
#include<stdio.h>
#include<signal.h>
```

```
void fp_err ( int );

main( )
{
  int X=5, Y=0, Z ;
 signal ( SIGFPE , fp_err ) ;
 Z = X / Y ;
}

void fp_err ( int signo )
{
 printf( "signal cought … … SIGFPE \n") ;
 exit (0) ;
}
```

When x / y is evaluated a division by 0 occurs and function fp_err is called and a message displayed.

Sending Signals

To send a signal use **kill (or raise)** as follows :-

```
val = kill ( pid, sig ) ;
```

or

```
val = raise ( sig ) ;
```

pid is the process number of the process we wish to send the signal to, and sig is the signal we wish to send. We can send a signal to ourselves with raise (or with kill). If pid is 0 then the signal will be sent to all processes in the senders group - including the sender.

Example :

```
// SIGNAL FOR GENERATING 'SIGINT' INTERRUPT THRU kill( )
#include<stdio.h>
#include<signal.h>
void fp_kill ( int );

main( )
{
  int i=1, pid ;
  pid = fork( ) ;
  signal ( SIGINT , fp_kill ) ;
  if ( pid = = 0 )
  {
    while ( i < 200 )
    {
```

```
            if ( i = = 100 )
            {
              kill ( pid, SIGINT );        /*  or use,   raise (SIGINT) ;   */
            }
              printf( "\n %d i … Hello … ");
              i ++ ;
          }
      }
}

void fp_kill ( )
{
  printf( "signal cought … … SIGINT \n") ;
  exit (0) ;
}
```

Some Associated System Calls

alarm : sets an alarm clock. The SIGALRM signal is used to tell the process the timer has elapsed. For example:-

```
unsigned int left, secs;

left = alarm(secs);
```

Note the process carries on immediately after the alarm with normal processing, until SIGALRM is received or until the alarm is turned off. You can only have one alarm outstanding. If you call alarm twice, the second call supersedes the first. The return value, left, gives how much time was remaining of the previous call. Note that alarm(0) will turn the alarm off.

pause : goes with alarm. Pause causes the process to wait until a signal arrives.

sleep : causes a process to sleep for a specified time. e.g. **sleep(10);** sleeps for 10 seconds.

wait : wait for a child to terminate. Eg. **retval = wait(&status);**

waits for a child to terminate. *retval* will contain the process id of the child (or -1 if wait failed) and status (an int) contains the exit status of the child, or is ignored if you use a null pointer (e.g. retval = wait((int *)0)) .

waitpid is similar but waits for a particular child.

COMMON SIGNALS

Most UNIX systems define about 30 signals. The most commons are described in the table below.

Name	No.	Default Action	Semantics
SIGHUP	1	Terminate	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Terminate	Interrupt from keyboard. Usually terminate the process. Can be triggered by `Ctrl-C`
SIGQUIT	3	Core dump	Quit from keyboard. Usually causes the process to terminate and dump core. Cab be triggered by `Ctrl-\`
SIGILL	4	Core dump	The process has executed an illegal hardware instruction.
SIGTRAP	5	Core dump	Trace/breakpoint trap. Hardware fault.
SIGABRT	6	Core dump	Abort signal from abort(3)
SIGFPE	8	Core dump	Floating point exception such as dividing by zero or a floating point overflow.
SIGKILL	9	Terminate	Sure way to terminate (kill) a process. Cannot be caught or ignored.
SIGSEGV	11	Core dump	The process attempted to access an invalid memory reference.
SIGPIPE	13	Terminate	Broken pipe: Sent to a process writing to a pipe or a socket with no reader (most likely the reader has terminated).
SIGALRM	14	Terminate	Timer signal from alarm(2)
SIGTERM	15	Terminate	Termination signal. The `kill` command send this signal by default, when no explicit signal type is provided.
SIGSTKFLT	16	Terminate	Stack fault.
SIGCHLD	17	Ignore	Child stopped or terminated
SIGCONT	18	Continue	Signal sent to process to make it continue.
SIGSTOP	19	Stop	Sure way to stop a process: cannot be caught or ignored. Used for non interactive job-control while `SIGSTP` is the interactive stop signal.
SIGTSTP	20	Stop	Interactive signal used to suspend process execution. Usually generated by typing `Ctrl-Z` in a terminal.
SIGTTIN	21	Stop	A background process attempt to read from its controlling terminal (tty input).

Name	No.	Default Action	Semantics
SIGTTOU	22	Stop	A background process attempt to write to its controlling terminal (tty output).
SIGIO	29	Terminate	Asynchronous I/O now event.
SIGBUS	10	Core dump	Bus error (bad memory access)
SIGPOLL		Terminate	Signals an event on a pollable device.
SIGPROF	27	Terminate	Expiration of a profiling timer set with `setitimer`.
SIGSYS	12	Core dump	Invalid system call. The Kernel interpreted a processor instruction as a system call, but its argument is invalid.
SIGURG	23	Ignore	Urgent condition on socket (e.g. out-of-band data).
SIGVTALRM	26	Terminate	Expiration of a virtual interval timer set with `setitimer`.
SIGXCPU	24	Core dump	CPU soft time limit exceeded (Resource limits).
SIGXFSZ	25	Core dump	File soft size limit exceeded (Resource limits).
SIGWINCH	28	Ignore	Informs a process of a change in associated terminal window size.
SIGPWR	30		Power failure restart.
SYGSYS	31		Bad system call.

Open Signals :

```
#include <stdio.h>
#include<signal.h>
int pid;
main()
{
        void abc(),def();
        pid=fork();
        if(pid==0)
        {
                signal(SIGUSR2,abc);
                sleep(1);
                printf("Hello Paapa\n");
                kill(getppid(), SIGUSR1);
                sleep(5);
        }
        else
        {
                signal(SIGUSR1,def);
```

```
                sleep(5);
        }
}

void abc()
{
        sleep(1);
        printf("Bye Papa\n");
        exit(0);
}

void def()
{
        sleep(1);
        printf("Hello baby\n");
        kill(pid,SIGUSR2);
}
```