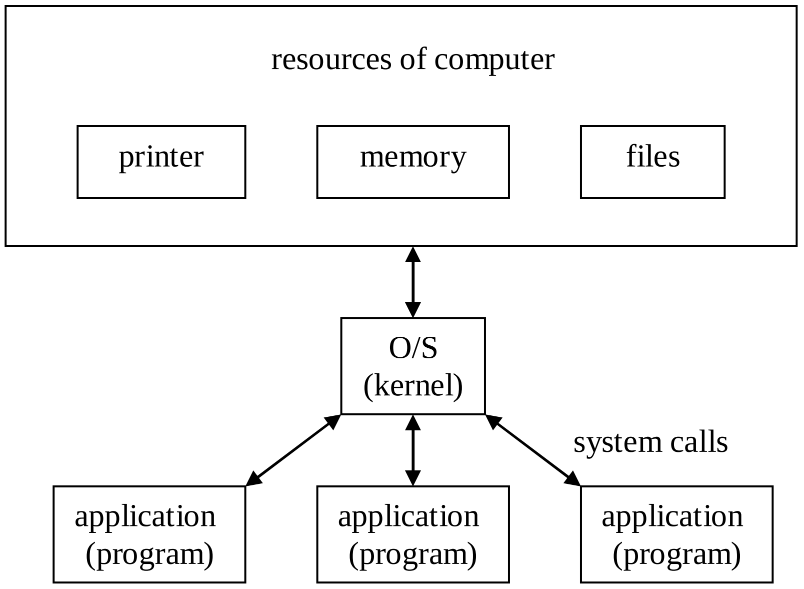
ECE 222 System Programming Concepts
Lecture notes – Signals System Calls

Recall from our earlier discussion that an operating system (O/S) has two jobs: it manages the resources of the computer (peripherals, files, memory, etc.) and it manages all other programs running on the computer.



One of the most important aspects of managing programs is supporting communication between them. Communication between running programs is called interprocess communication (IPC).

Since the kernel is itself a program, there are a set of system calls designed primarily to facilitate communication between the kernel and an application. These system calls are often called signals.

What sorts of messages would a kernel have for an application? Mostly, they are messages informing a program that it needs to terminate. Signals can be generated by three different means:

1. user – pressing CTRL-C (user wants process to terminate)
2. kernel – process did something wrong, e.g. divide by zero or access bad memory (kernel wants process to terminate)
3. other process – sends a signal using the kill() system call function (parent process tells child process to terminate)

Synchronous signal events happen predictably, meaning they always occur at the same point in program execution. An example is a divide by zero event. Asynchronous signal events cannot be predicted. An example is the user pressing CTRL-C.

We have three choices as to how our process will handle each signal:

1. accept the default signal action (usually death)

```
signal(SIGINT,       SIG_DFL);
       which signal  what to do
```

2. ignore the signal

```
signal(SIGINT, SIG_IGN);
```

3. install a custom signal handling function

```
signal(SIGINT, ourfunction);
               name of a function in our program
```

There are 30-40 different signals handled by a typical unix/linux kernel:

```
man 7 signal        [shows the list of signal events]
```

Each signal has a unique integer code. Where are all these codes defined? In order to use signals, we must

```
#include <signal.h>
```

Looking inside there, we see the line

```
#include <bits/signum.h>
```

The codes are in that file. All the references to SIGINT, etc., are #defines.

Each signal also has a default "signal handler". A signal handler is a function, inside the kernel, that is executed whenever the signal is received by the process. For example, the default signal handler function for SIGINT is code to exit() the process. We can provide our own signal handler function:

[see sigdemo1.c example]