When programming in **C** (or similar), you must include a number of files such as these:

```
#include<stdlib.h>
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<signal.h>
#include<string.h>
```

you may need more of these.

Every process has its own unique identifier assigned to it by the Operating System.

Processes form a tree, each process having a parent process. Processes started interactively from a terminal (window) have as parent the shell (click click click) process controlling the window.

```
pid =  getpid() ;        // This is my process id
ppid = getppid() ;       // This is my parent process id
```

pid and ppid are of type pid_t which looks like an int to me.

Forking a process

The system call fork() (click click click) creates a child process. Any process can create one or more child processes.

When fork() (click click click) is called, the process is duplicated into two identical copies, each copy having exactly the same values, the same files, etc. One copy is called the parent, the other is called the child process.

They differ in one thing: the return value of the call itself:

```
rv = fork() ;
```

This call will return in two places: in each copy of the forking process:

The parent will find the value of rv to be a positive integer equal to the process id of the other copy (i.e. the child).

The child will find rv to be equal to 0.

Note that the processes are two distinct copies and do not hold any shared variables.

A call to fork() is always followed by an if statement which distinguishes between the parent and the child:

```
int makechild( int t )
{
   pid_t pid ;
   if( (pid = fork()) == 0 ) {
      sleep( t ) ;
      fprintf( stderr , "\7\7Time is up\n" ) ;
      exit( 0 ) ;
   } else
      return pid ;
}
```

Where is the code of the child? The parent?

Make sure to understand why the if is needed and why one part of the code is ended by an exit() while the other is a return.

If you need more reading materials, try lupg.

If still more is needed you probably are tired and need to click here or go there.

A file as a lock

One can use the presence or absence of a file as an imaginary lock. When the file exists, the lock is on; otherwise, it is off.

```
int flock ;
int mode = S_IREAD | S_IWRITE ; // trust me
char lockfile[20] ;
sprintf( lockfile , "alertlock%d" , ppid ) ;
while( (flock = open( lockfile , O_CREAT | O_EXCL , mode
)) < 0 )
    sleep( 1 ) ;
    ... ... ... ... ...
close( flock ) ;
unlink( lockfile ) ;
```

The flags O_CREAT and O_EXCL make open() fail if the named file already exists.

It is necessary to release the lock in due time; this is accomplished by the close() and unlink() sequence.

Command–line arguments are received by a program in the form of arguments to main(). Two argumants are normal, the second (argv) being an array of pointers to strings and the first (argc) giving the length of the array.

The first element of the array (argv[0]) is the name of the program.

If you pass this invocation to the shell:

```
alert 3 5
```

argv will be an array of 3 strings equal to ”alert” , ”3” , ”5” respectively.

```
int main( int argc , char **argv )

  if( argc < 3 ) {
    printf( "Arguments?\n" ) ;
    exit( 0 ) ;
  }
  action = atoi( argv[1] ) ;
  t = atoi( argv[2] ) ;
```

You send a signal to a process through the grand–sounding system call kill(). There are many signals, each with its own official meaning. You can redefine the meaning of most signals by catching them; however, some signals cannot be caught.

```
kill( process , SIGUSR1 ) ;
kill( process , SIGBUS ) ;
kill( process , SIGTERM ) ;
```

A list of signals can be found click here.