Module 4: Processes

What is a Process?

A ***process*** is a program that is running on a computer, also known as **a running program**. A ***program*** alone is just a file containing a list of instructions for a computer. It is only when these instructions are being executed on a computer, that we can say that the program is running and becomes a process.

To write a program, we are probably using a text editor (like vim) that creates a file with the instructions in some predefined computer language (like C). The instructions in this file are later translated to instructions in machine language by a compiler (like gcc). A successful compilation generates another file containing the translated instructions. To run this compiled file in the computer, the operating system must load its instructions in the computer's memory and make them ready to take control of the computer. It is only then that we can consider it to be a process, and it will remain a process until the operating system terminates it.

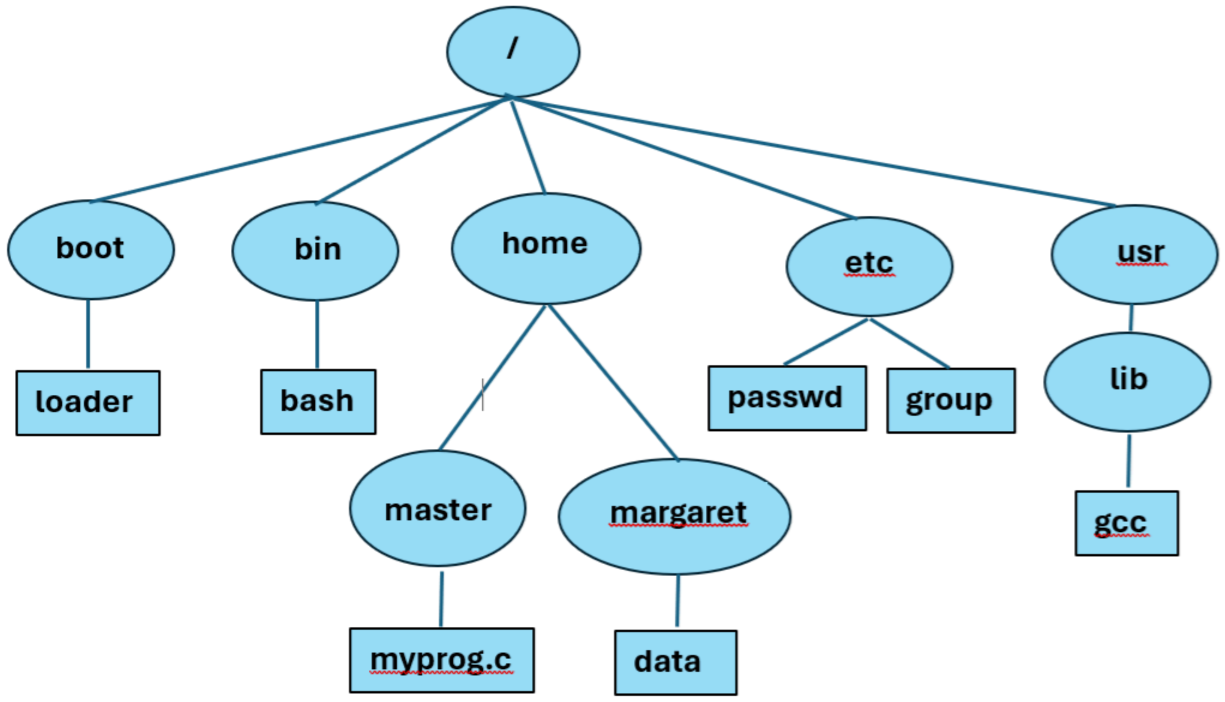
Notice that we are using other programs and processes to generate our program. The text editor and the compiler are collections of many files with instructions that help us write code and create executable files, respectively. To perform these activities, the files of these programs need to run on the computer. When they are running, they also become processes. Any program running on a computer is a process.

The operating system has at its core a set of instructions known as the ***kernel***. The kernel is the actual manager of the computer and handles all its operations. Among other things, it decides everything about the processes: when can they run, how to run and for how long. It also allocates resources for these processes: CPU to run, memory, files and peripherals. The kernel itself is not a process.

While working, the operating system is constantly switching between two modes of operation, the kernel mode and the user mode. In kernel mode, the kernel is in control of all operations. It can perform any activity in the computer without any restriction. When there are processes to run, the kernel selects one of them, loads it in memory, assigns resources to it and switches to user mode, to let the process take control of the computer, but in a restricted fashion. The process can only perform activities over the resources that it has control. If the process were to require more resources, like more memory, or access to a device, or even terminate, it must request the kernel for these actions to be performed. When it does, the kernel regains control of the computer in kernel mode and decides to grant or not these resources, to let the process continue in user mode again, or to grant access to the CPU to another process in user mode again.

Processes and the File System

The Linux file system is a hierarchical tree of directories, links, and files. The base of this tree is the ***root directory***, symbolized by the slash symbol (/). Every directory, link, and file in the file system is a descendant of the root directory. A previous unit in this course explored files and how to access them. A directory is a repository of filenames, each one associated with a reference to a file, this association is called a ***link***. The link could contain a reference to a physical file in the system. This is known as a ***hard link***. Alternatively, it could contain a reference to another filename in what is known as a ***soft link***, or more commonly, ***a symbolic link***. The existence of symbolic links allows for the creation of multiple filenames for a single physical file, and the ability to make the same file visible in multiple directories. Among the entries a directory may have, we can find other directories. This is what creates the structure of the tree hierarchy. Every directory will contain at least two other directories: dot (**.**) and double-dot (**..**), representing itself and its parent directory, respectively. The following figure shows a sample tree structure of a file system. The ellipsis are directories while the squares are files.:



When a process starts, it is assigned a ***current working directory*** that is inherited from its parent process, unless it is a shell process which gets its assignment from the ***home directory*** of the user who creates the shell. The home directory is part of the information stored for every user in the password file for the system. Processes use their current working directory to discover the location of files within the file hierarchy.

There are two ways to indicate the placement of files in the file system. The ***absolute pathname*** describes the position of a file starting at the root. For example, in the figure above, the absolute pathname for the file ***myprog.c*** is ***/home/master/myprog.c*** . Notice that an absolute pathname always begins with the slash (*/*) for the root directory. That is its give away. In an absolute pathname we write the name of every directory in the path from root to the required file, each name separated by slashes. The other way to indicate file placement is with the ***relative pathname***, and this is where the current working directory is needed. A relative pathname for a process is based on its current working directory. For example, if the user named ***master*** opens a shell, and its home directory is ***/home/master***, then this will also be its current working directory. Now, under these circumstances, all that the shell needs to do to refer to the file ***myprog.c*** is to name it, because it is already placed in its current working directory. There is no need to add any directory name.If on the other hand, the shell wants to refer to the file named ***data*** under the ***margaret*** directory, it will have to indicate the path to this file from the current working directory. This will go something like this: ***../margaret/data*** . Notice this time that the relative pathname does not begin with the slash for the root directory, instead it begins with the double-dot (*..*), indicating that from the the ***master*** directory (the current working directory), the search must go up to its parent directory (***home***), and from there it should go down to the directory named ***margaret*** to find the file ***data***.

While in the shell, the user may change its current working directory with the command ***cd***. This command takes a pathname as a parameter (absolute or relative) which becomes the new current working directory. Used without parameters, ***cd*** moves the current working directory backs to the home directory. If a C program wants to retrieve the current working directory, it may use the ***getcwd()*** system call that returns it as string (*char* *). The parameters for this function call are a string with enough space to receive the pathname and the size of this string. If that string is not long enough, the function call will fail. To change this current working directory, the C program may use the ***chdir()*** system call that takes the new pathname as a parameter and returns the integer 0 for success or -1 in error. The prototypes for both system calls are as follows:

```
char * getcwd (char *cwdbuf, size_t size )

 int chdir ( const char *pathname )
```

To access a file, it is not enough to know the placement of a file in a file system, we must also be aware of the file permissions granted to the file. File permissions are divided in three groups: permissions for the owner of a file, permissions for the group to which the file belongs, and permissions for anyone else. Each one of these groups may have the following permissions: permission to ***read*** the file, permission to ***write*** on the file and permission to ***execute*** the file (used if the file is a compiled program or a script). The same permissions are granted for directories, but with slight different meanings. Reading a directory means to be able to see its contents, writing in a directory means to be able to modify its contents, but the execution permission is actually a search permission. It allows for the search of a particular entity inside the directory.

We can observe the permissions on every file or directory if we use the command ***ls -la***. The following figure shows samples of this command applied to the directories ***master*** and ***margaret*** in the file system depicted before, by user named ***master***:

```
master $ls -la
drwx------   21 master   master    4096 Jun  5 22:14 .
drwx-xr--x    3 root     root        20 Apr  2 16:40 ..
-rw-r--r-     1 master   master    1167 Jul 26 12:36 myprog.c
master $ls -la ../margaret
drwx------    3 margaret margaret  4096 May  6 11:10 .
drwx-xr--x    3 root     root        20 Apr  2 16:40 ..
-rw-r--r-     1 margaret margaret 10423 Jun  6 17:14 data
master $
```

The results presented by this command show the permissions on the left side. The first character indicates that kind of file being observed (***-*** for a regular file, ***d*** for a directory). This is followed by three sets of three characters, for user, group and other users. For each of these sets there is a character for the read permission (***r***), the write permission (**w**) or the execute/search permission(***x***). If any of these permissions is not granted, then a dash (-) is being displayed instead. For example, the ***myprog.c*** and ***data*** files both allow read and write permissions for the owners (***master*** and ***margaret***, respectively) , and read only permissions for their group and everyone else.

After the permissions, the results of the command also show the number of links the file has, the owner of the file, the group to which it belongs, the size in bytes, the last date and time that was modified and the file name and extension. With the permissions above, if the master user wants to read the data file in the margaret directory, it may do so because read permissions have been granted to everyone. However, if it tries to modify it, it cannot, because it is neither the owner, and it does not belong to the same group as the file, and the permissions for everyone do not include the write permission.

Owners and privileged users may change the permissions of a file with the ***chmod*** shell command. The most basic structure of this command is as follows:

```
chmod mode files
```

This command will change the permissions of the files mentioned in the files parameter as indicated by the mode. There are two ways to indicate the mode, explicitly and implicitly. The explicit mode mentions the users affected (***u*** for user, ***g*** for group, ***o*** for other, or ***a*** for all) followed by an operation symbol (**+** to add, **-** to remove, or **=** to set and remove unspecified) followed by the previously mentioned symbols for the permissions (***r*** to read, ***w*** to write, and ***x*** to execute/search). For example if the ***margaret*** user is going to allow everyone to read and write in her data file, not just the owner, it may use any of the following alternatives:

```
chmod g+w,o+w data
chmod g=rw,o=rw data
chmod =rw data
```

The implicit mode can only be used to assign all permissions at once. It assigns values to the various permissions: 4 to read, 2 to write and 1 to executer/search. Each one of these values is used at face value if we are given permissions to other users, but they are multiplied by 10 if we are giving permissions to the group, or by 100 if the permissions are granted to the owner. All these values must be added and the final number is used as the mode. For example if the margaret user is going to grant read and write permissions to everyone we need to add 4+2 for everyone with 40+20 for the group and 400+200 for the owner, giving a total of 666 that can be used as the mode.

When creating a file (with the ***open()*** system call) or a directory (with ***mkdir***) we can specify their permissions with a parameter, however, these permissions are modified by the ***process file mode creation mask***, also known as the ***umask***. Every process inherits a umask from its parent process. This contains a set of permissions that will be eliminated in every file the process creates. A typical umask contains the value 022, representing the write permission for group and other users. This eliminates this permissions from all new files and directories the process creates.

Process Identification

Processes can be recognized by their Process Identification Number, commonly known as the ***PID***. This is a unique positive integer number that is assigned to the process by the kernel. The system call ***getpid()*** in C returns the PID associated with the calling process. Every process is also a child of another process. For example, when a user requests to run a program from a shell, the process generated by that program becomes a child of the shell. A process may obtain the PID of its parent process from the system call ***getppid()***.

For greater control, related processes belong to a group with a group id. Usually, a process inherits its group and group id from its parent process. However, the parent process may decide to assign a different group to its children. This is what usually happens with commands in shells. For example, if a shell creates a pipe of commands, like ***ls | sort***, two processes are created, one for the ***ls*** command and another for the ***sort*** command, both with different PIDs. Because these processes are related by working in tandem, the shell makes them both part of a new group and designates the first process (***ls***) as the group leader. Its own PID also becomes the group id.

In a process, a system call ***getpgrp()*** is used to obtain its own group id. To identify the group id for any other process, one may use the system call ***getpgid(pid),*** where the ***pid*** parameter contains the PID of the requested process.

A ***session*** is a set of process groups, all of which are assigned to the same session ID. The session ID is the PID of the process that created the session, known as the session leader. For example, when a shell starts running, it creates a session made out of a single group with a single process, the shell itself. This shell process is the session leader. Every command that runs later on the shell will belong to a group that may be different from the shell group, but they all inherit the same session ID, the session ID of the session leader. The system call ***getsid(pid)*** returns the session id of a process with the parameter ***pid***. When this system call is given a ***pid*** of zero, it returns the session id of the calling process.

The figure below shows a program named ***identify,c*** that uses these system calls on the left side and the output it produces when running on the right side.

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
   pid_t myPID = getpid();
   pid_t myParentPID = getppid();
   gid_t myGID = getpgrp();
   gid_t myParentGID = getpgid(myParentPID);
   pid_t mySID = getsid(0);
   pid_t myParentSID = getsid(myParentPID);

   printf("I am process %d\n", myPID);
   printf("I belong to the group %d\n", myGID);
   printf("My parent is process %d\n", myParentPID);
   printf("My parent belongs to the group %d\n", myParentGID);
   printf("My session ID is %d\n",mySID);
   printf("My parent session ID is also %d\n",myParentSID);
   sleep(5);

   return 0;
}
```

```
master $identify
I am process 40833
I belong to the group 40833
My parent is process 40228
My parent belongs to the group 40228
My session ID is 40228
My parent session ID is also 40228
master $identify & ps
[1] 40838
I am process 40838
I belong to the group 40838
My parent is process 40228
My parent belongs to the group 40228
My session ID is 40228
My parent session ID is also 40228
   PID TTY          TIME CMD
 40228 pts/0    00:00:00 bash
 40838 pts/0    00:00:00 identify
 40839 pts/0    00:00:00 ps
master $
```

The output above also shows how the command ***ps*** can be used to display the ID of processes currently running by the shell. Notice that the process called ***identify*** is made asleep in the background so the ***ps*** command can find it and display it. To view all processes in the system the ***ps -A*** command can be used.

A session also has a ***controlling terminal*** that is determined when the session is created. The terminal is where user interaction happens, where the user may enter commands through the standard input and see results from the standard output (if not redirected). A session may have control of just one terminal and a terminal may be controlled just by one session leader. Every process in the session is associated to the terminal, but only one process group is considered to be on the ***foreground***. This means that this process group is the only one that is able to read messages from the standard input, namely to receive commands from the terminal. All other groups belonging to the same session are considered to be on the ***background***.

The system call ***setpgid(pid, pgid)*** is used to change the group id of the process with the parameter ***pid*** to a value of ***pgid***. This system call has the following characteristics:

- If ***pid*** is zero, the change happens to the calling process.
- If ***pgid*** is zero, the group id of the process with the parameter ***pid*** is changed to ***pid*** and becomes that group leader.
- The parameter ***pid*** may only refer to the calling process itself or one of its children and it cannot be a session leader.
- The calling process, as well as both processes on the parameters (***pid*** and ***pgid***) must belong to the same session.

Similarly, a session id for a process can also be changed with the system call ***setsid()***. A process other than the session leader can make this call that creates a separate session with a single group. The PID of this calling process becomes the group id and session id of the new group and the new session. However, this new session will have no controlling terminal.

Every process is also running at the bequest of a user. All users are assigned a ***user ID***, or ***UID***. A process may request the identity of its user with the ***getuid()*** system call.The user must have the right to run the process either because it is the owner, or because it belongs to a group with the right to run that process or because that process is available for everyone to run.

On occasions, the owner of a program may grant its right to run it as a process to another user which cannot. To do that, the program must set its ***set-user-id*** and/or ***set-group-id*** bits. The following figure shows how this can be done and the effect it causes in the program. Notice in the figure how the privileges for the file **identify** are changed

```
master $ls -l identify
-rwxr-xr-x. 1 master master 17824 Jun 14 21:46 identify
master $chmod u+s identify
master $chmod g+s identify
master $ls -l identify
```

```
-rwsr-sr-x. 1 master master 17824 Jun 14 21:46 identify
```

If a user other than **master**, its owner, may run **identify** as a process, it will run as if it were the owner. To do that, the kernel uses what is called the ***effective user ID***, or ***EUID*** and its associated ***effective group ID***, or ***EGID*** instead of its own uid and gid. A process may identify its EUID and EGID with the ***geteuid()*** and ***getegid()*** system calls, respectively. To make multiple uses of this program, the EUID and EGID are stored as ***set-user-ID***, or ***SUID***, and the ***set-group-ID***, or ***SGID***.

We invite you to read more about credentials for process in the credentials page of the man pages.

The *init* Process

We mentioned that every process is a child of another process, for this to be possible, there must be a process that was the first. This process is known as the ***init*** process and it is created by the kernel when the operating system is at booting stage. This process is the first one to appear and it is the last one to go. It remains active from start-up to system shutdown. It cannot be destroyed, not even by a superuser, because it keeps control of other important processes that allow the correct running of the operating system. In particular it is in charge of taking care of "orphaned" processes as we will later see. Every process in the system is a descendant of init. The process ID for the ***init*** process is the number 1 and it has the highest privileges, as if it were run by the superuser.

Process Creation

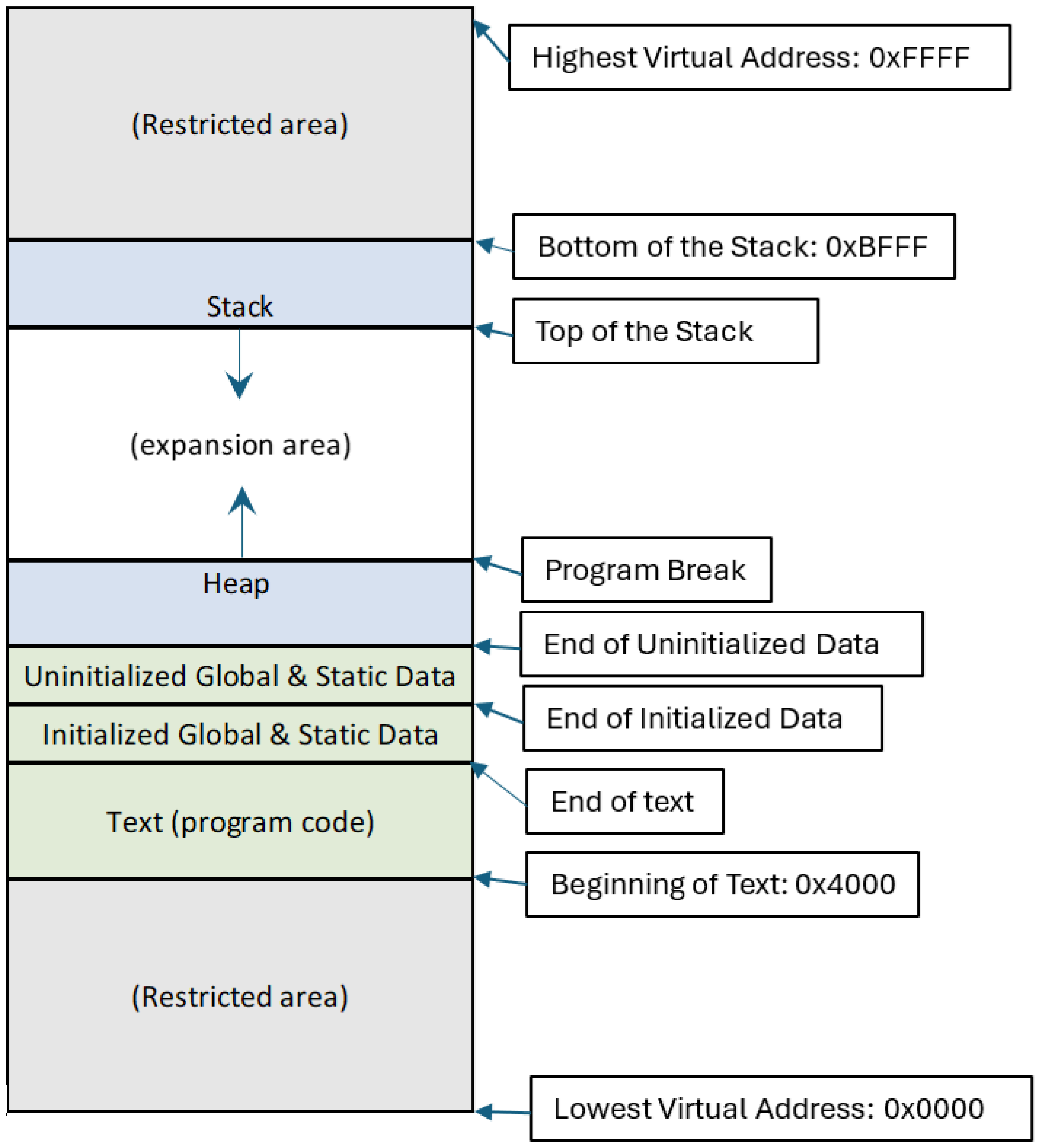
When a program is to be run, the operating system allocates various portions of memory (RAM) to the process. They constitute the ***virtual memory address space*** for the process. The amount of memory to be assigned varies from machine to machine and from program to program, and it solely depends on what the operating system decides. For this discussion, let's assume that a process is assigned 64Kb of memory. Its address space begins with memory at position zero (0x0000 in hexadecimal) and its last address is one position before 64Kb (address 0xFFFF). Certain portions at the beginning and at the end of this address space are reserved by the operating system for process management and to store kernel references that will be used by the process. For this reason, let's also assume that this leaves us with a remaining address space for the process beginning at 16Kb (address 0x4000) and ending one position before 48Kb (address 0xBFFF).

The remaining address space is further divided in ***segments***. Each segment is assigned a different function and contains specific information for the process. On this first segment, known as the ***text***, the operating system loads all the instructions of the compiled file (the program code). These are followed by all global and static data that are initialized by the program, and after them all global and static data that is not initialized. In our example, the text will begin at 16Kb (address 0x4000) and will continue until all code and global and static data is placed. The actual addresses where the program code, the initialized data, and then uninitialized data end, are all stored in CPU registers to inform the running of the process.

A second segment, the ***stack,*** begins at the other end of the address space (address 0xBFFF in our example), and it actually grows backwards towards the beginning of the address space. The stack is used to keep track of the functions the process calls. A CPU register (labeled as %rsp) holds the value of the memory address at the end of the stack. This address is known as the ***top of the stack***. Every time a function is called, a new ***stack frame*** is added after the top of the stack. The stack frame contains information relevant to the function call and includes values for the parameters passed to the function call as well as places for the values of all the variables that are local to the function. Once the stack frame is added, the top of the stack is updated to point to the end of this new frame. When the function finishes calculations, its return value is stored in a CPU register to be sent to the calling function, the memory allocated to the stack frame is released, and the top of the stack is moved back to point at the end of the previous function call.

The third and last assigned segment is the **heap**. This usually lies right after the memory assigned for the uninitialized data and grows in opposite direction to the stack. The heap provides memory for all variables that are dynamically allocated when the process runs. The end of the heap is known as the **program break**, and the kernel may change its location to make it grow or shrink, based on the need and requests from the process.

The following figure shows an example of the placement of these segments in the virtual memory address space for a process:


(Restricted area)
Highest Virtual Address: 0xFFFF
Bottom of the Stack: 0xBFFF
Stack
Top of the Stack
(expansion area)
Heap
Program Break
Uninitialized Global & Static Data
End of Uninitialized Data
Initialized Global & Static Data
End of Initialized Data
Text (program code)
End of text
Beginning of Text: 0x4000
(Restricted area)
Lowest Virtual Address: 0x0000

It is important to realize that this diagram shows memory from the point of view of the process, and likely, it will not represent the actual memory arrangement of hardware that the kernel has at its disposal. For example, the kernel will not grant access to the physical address 0x0000 to any process. This is reserved for the operating system. Also, the process may assume that all its segments are placed continuously in memory, when in fact the kernel usually makes allocations in non-contiguous areas. However, the process does not need to worry about any misalignment, because a big part of the kernel's job is to maintain the illusion of a unified memory space for every process.

Once these segments are allocated, the operating system makes some other initializations. Every process is associated with three standard files: standard input, standard output, and standard error. The standard input file is the place from where the process will get external input, usually the keyboard. The standard output is the place from where all textual output from the process will be written. This is generally the screen or terminal from where the process is launched. Finally, the standard error is the place where all errors encountered by the process while running are to be reported. Most of the time this is also the same screen or terminal used for standard output, but there are other options, like sending it to a file in disk. The operating system will treat each one of these standard files like any other file in the system and will set file descriptors for each one of them. If there is redirection of these files, the operating system will modify the appropriate file descriptor to point to the requested device or file.

Finally, the operating system must initialize a CPU register with the address of the first executable sentence from the program. This register is known as the **Instruction Pointer** or **IP** (also known as the **program counter** or **PC**).

As an example of how the kernel assign a program to various segments in the process' memory space, consider the following C program that creates a small table of compound interest:

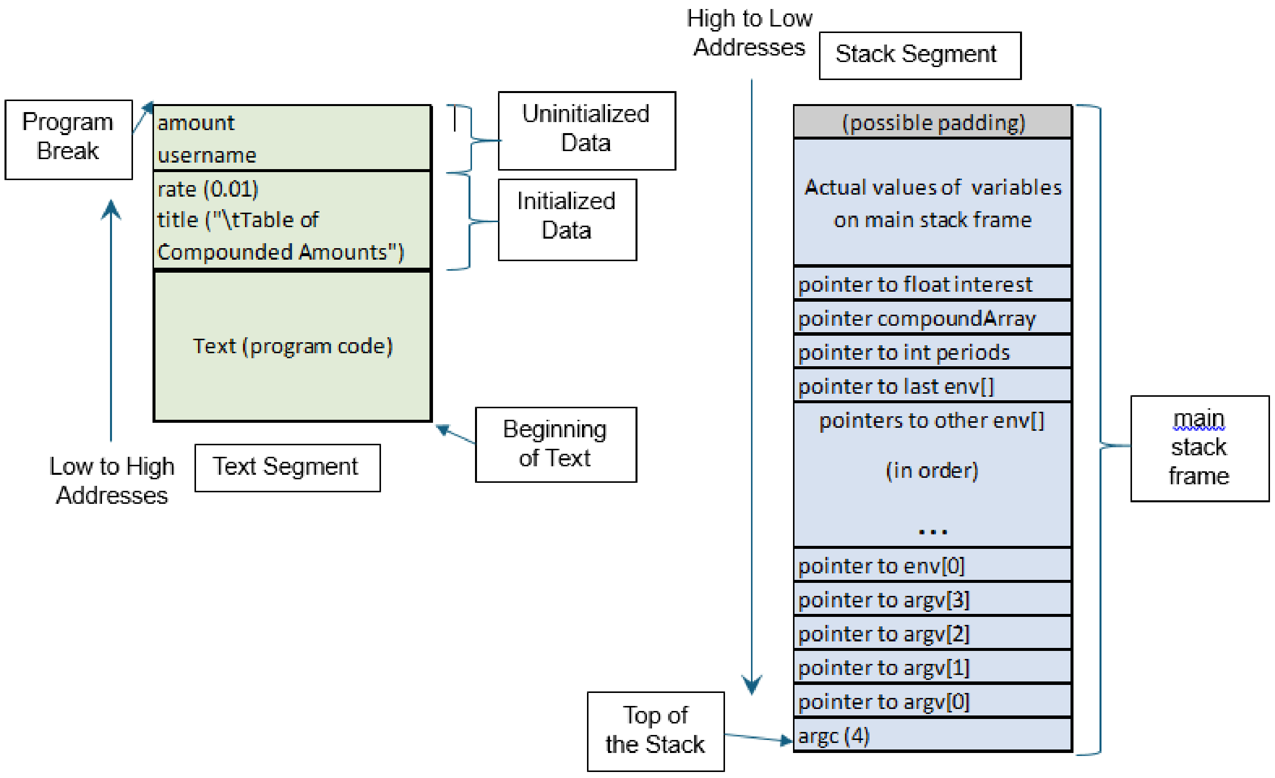
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Program: compound.c
// Version: 1.0
// Author: Guillermo Tonsmann Ph.D.

char *username;                                // Name of the User <Uninitialized data>
char *title = "\tTable of Compounded Amounts";   // Title to be Displayed <Initialized Data>

float compound(float *array, float a, float r, int p) {
   // Function to evaluate compounded interest over a number of periods
   // Arguments:  <on Stack>
   //   array: Pointer to an array of p+1 floats to be filled with compounded amounts
   //   a    : initial amount to be compounded.
   //   r    : interest rate as a fraction
   //   p    : number of periods to be compounded (== size_of_array-1)

   *array = a;    // First position of array contains the initial amount to be compounded

   // Evaluating the compound interest at each period and placing it in the array
   for (int i=1; i<=p ; i++) {
      *(array+i) = *(array+i-1)*(1.+r);
   }
   return (*(array+p)-(*array));          // <Value returned in a register>
}

void display(float *array, float p) {
   // Function to display a table of compounded amounts
   // Arguments:  <on Stack>
   //   array: Pointer to an array of p+1 compounded amounts
   //   p    : number of periods to be compounded (== size_of_array-1)

   printf("%s\n",title);                    // <from Initialized Data>
   printf("\t\tCreated by:%s\n",username);    // <from Uninitialized Data after initialization>


   printf("%15s %15s\n","Amount"," After Period:");    // Table headings
   // Printing each line of the table
   for (int i=0; i<=p ; i++) {
      printf("%15.2f %8d\n",*(array+i),i);
   }
   return;                              // Nothing to return
}

int main(int argc, char *argv[], char* environ[])
{
   // Main program, gets data from argv, generates an array for compound values and
   // ask to be populated and displayed

   // Static Variables
   static float amount;        // Initial amount to be compounded. No default. <Uninitialized Data>
   static float rate = 0.01;   // Interest rate as a fraction. Default: 1% as 0.01
                            // < Initialized Data>

   // Local variables <on Stack>
   int periods = 5;            // Number of periods to be compounded. Default: 5 periods
   float *compoundArray;       // Array of compounded amounts (to be created)
   float interest = 0.0;       // Accumulated interest over all periods

   username = getenv("USERNAME");   // Getting a variable from the Environment Array <On Stack>
                              // Saving it into <Uninitialized Data>

   // Reading and validating parameters from argv <on Stack>
   switch (argc) {
      case 4: periods = atoi(argv[3]);
      case 3: rate = atof(argv[2])/100.0;
      case 2: amount = atof(argv[1]); break;
          default:
            printf("Invalid number of parameters\n");
            printf("Usage: compound amount rate periods\n");
            exit(1);
   }

   // Validating periods
   if (periods < 1) {
            printf("Periods should be integer with value greater than 1 \n");
            exit(1);          // program returns with errors
```

```
    }

    // Creation of a dynamic array for the list of compounded amounts <on Heap>
    compoundArray = (float *) malloc( (periods+1) * sizeof(float) );
    if (compoundArray == NULL) {
            printf("Memory Allocation error.\n");
            exit(1);        // program returns with errors
    }

    // Call to evaluate compounded amounts. Total interest earned is returned.
    interest = compound(compoundArray, amount, rate, periods);
    // Call to display a table of compounded amounts.
    display(compoundArray, periods);
    // Additional information added to the table.
    printf("\tAccumulated Interests:%.2f\n",interest);

    return 0;               // program returns successfully
}
```

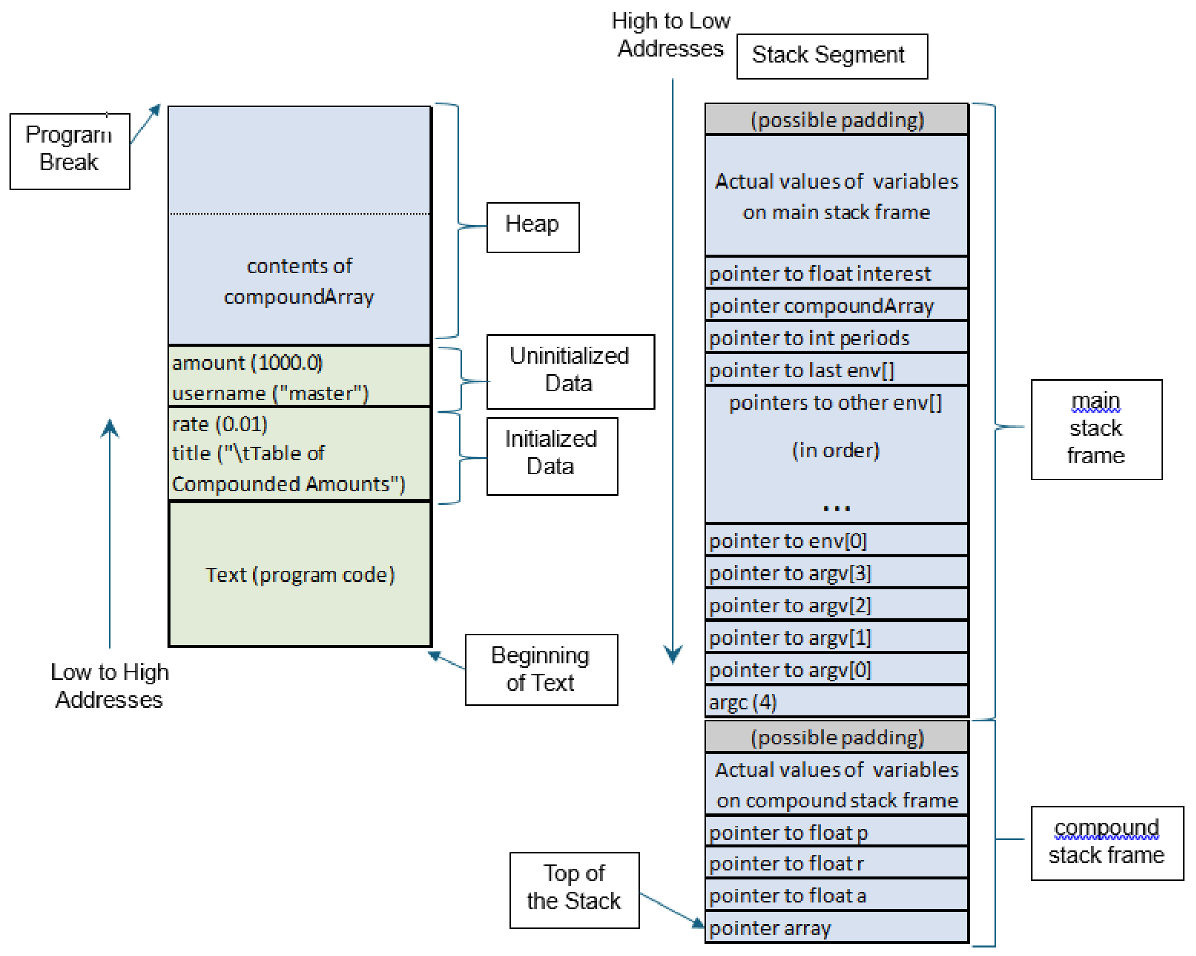
After compilation, the program can be executed with parameters 1000, 2, and 7, corresponding to an amount of $1000.00 to be compounded at 2% per period during 7 periods. This produces the outcome shown in Figure 5.3:

```
master $compound 1000 2 7
        Table of Compounded Amounts
                Created by:master
        Amount   After Period:
        1000.00         0
        1020.00         1
        1040.40         2
        1061.21         3
        1082.43         4
        1104.08         5
        1126.16         6
        1148.69         7
        Accumulated Interests:148.69
master $
```

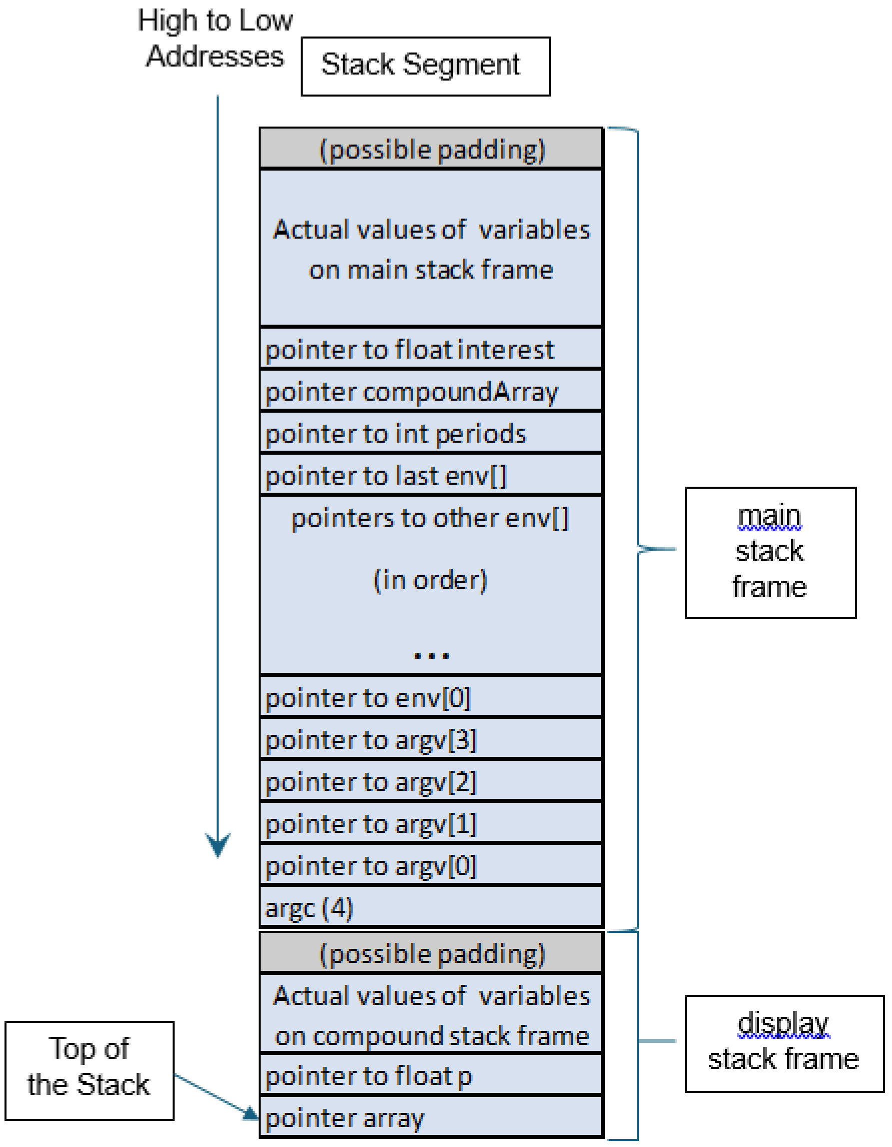
When the operating system receives the request to run the program for compound interest, it will initially load the various segments as shown in Figure 5.4. Notice that the values of global variable ***title*** and the static variable ***rate*** are placed in the initialized data section of the text segment, while some memory space is left for the global variable ***username*** and the static variable ***amount*** that will be filled with values when the program runs. The first stack frame to be loaded on the stack segment contains information for the running of the first function in any C program, ***main***. The stack frame is filled in the opposite direction than the text segment. At the top of the stack, we found the value of ***argc***, that is the number of arguments passed by the shell to the program (4 in this case). This is followed in order by the arguments from ***argv*** and the environment variables from ***env***, also received from the shell. Given that these values may have different sizes, rather than placing them in that position, they are actually located at the end of the stack frame, and only pointers to their actual location are stored at the beginning of the stack. The stack frame also includes memory to store the contents of the local variables from the main function. The stack frame may contain some additional memory assigned as padding; because they should be aligned at specific word sizes (16 or 32 bytes depending on the architecture). Since no dynamic allocation was performed at this stage, the heap may not have any actual memory associated with it. Nevertheless, a program break will be already set, probably at the end of the text segment.



By the time the program is running and calls the compound function, the segments may appear as described by Figure 5.5. By then, the main function has requested memory for the contents of the compoundArray. The operating system will grant this request on the heap, by increasing the program break to a size that provides enough space. However, the operating system may decide to provide more than the requested space, based on its own policies. The compound function will add a new stack frame to the stack segment. This stack frame will contain enough memory to hold all the parameters for the function and its local variables. Some padding memory may also be added. The top of the stack is moved accordingly to the end of this stack frame. When the function ends running, the value returned by the function will be placed on a CPU register for the calling function to take and the allocated memory is released by moving the top of the stack back to its previous value at the end of the stack frame for the main function.



When the program calls the display function, a new stack frame is added on top of the stack segment. This new frame contains memory to hold all its parameters, its local variables, and padding memory if required. Once again, the top of the stack is moved to the end of this stack frame. This function uses the file descriptor for the standard output to display the program's results. When it finishes it does not return any value. The top of the stack is moved back to the end of the stack frame for the main function.



Process Lifecycle

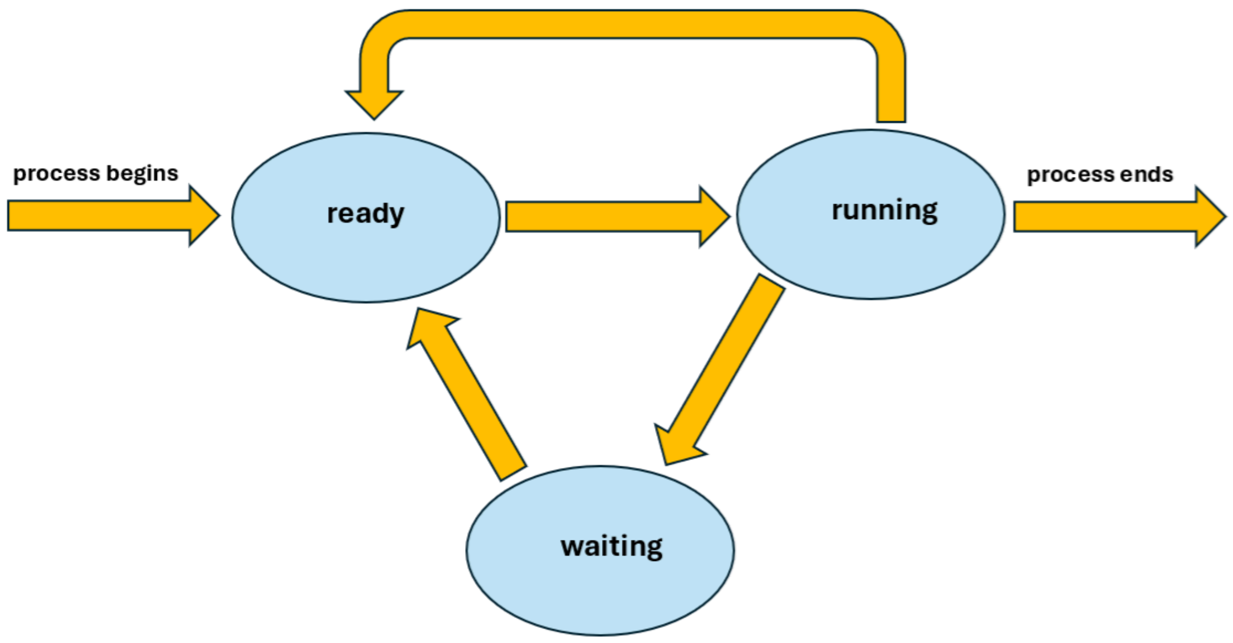
Once a process is loaded in memory we say that the process is in a ***ready state***. It is ready for its instructions to be run by the CPU. At any given time there could be many processes in ready state trying to access the CPU. All these processes constitute a ***ready queue*** from which the kernel decides which process to run next. If the kernel decides to run a process from this queue, it updates the ***CPU context***. This is the set of CPU registers that contain pointers to the locations where the process segments for that process are loaded. These include the values for program break, base of the stack, bottom of the stack, and instruction pointer. The kernel updates the CPU context with the appropriate values for the process to be run. The kernel then switches to user mode and allows the process to take charge of the computer to perform its instructions. This is a second state of the process known as the ***running state***. The process will remain in this stage until one of the following events happens:

1. The process completes all its instructions and requests the kernel to terminate it.
2. The process does not complete all its instructions, but it is terminated by the kernel due to an irrecoverable error while running.
3. The process does not complete all its instructions, but it is terminated by the kernel because a predetermined amount of time has passed, and the kernel wants to give access to the CPU to other processes in the ready queue.
4. The process requires some resource (memory, reading of a file, or access to a peripheral, for example), and it is making a request to the kernel for that resource.

In all these cases, the process stops operations and the kernel takes control in kernel mode. In cases 1 and 2, where the process will no longer be active, it is terminated by the kernel, and all the resources that were allocated to that process are released at the kernel's disposal. In case 3, when the allotted time for running has expired, the kernel places the process back into the ready queue, to wait for a new time slot. In case 4, where the process requires a resource, if the kernel can satisfy that request quickly, it will grant it, and let the process resume operations again in user mode. If on the other hand, the process must wait for some resource in a queue, the kernel changes the state of the process to a ***waiting state*** until it can get hold of the resource. When this happens, the kernel moves the process back to the ready queue to wait for a new time slot, just like in case 3.

In both cases, 3 and 4, before the process changes state, all registers in the CPU context are stored. These stored registers will be used later to restore execution of the program when the process is sent back to the running state. Therefore, every time that the kernel changes the process to be running in the CPU, it saves CPU context of the old process and copies the recorded CPU context of the new one (if it exists) on the actual CPU registers. This activity is called ***context switching***,

The figure below shows the flow of a process along its possible states:



Process Termination

There are two ways in which a process is terminated. In the first one, the process itself requests its termination to the kernel by calling the system call ***_exit(status)***. The underscore is part of the name of the system call, and the *status* is an integer value, usually between 0 and 128, known as the exit code. Processes that end successfully usually return with an exit code of zero. The process may use a value different than zero for the exit code to indicate some possible error that was encountered while running. Users and processes expecting the conclusion of that process may review this value to determine further actions. Within a shell, the exit code of the latest terminated process is stored in the environment variable ***$?***. The second way in which a process may be terminated is by using signals.

Signals

Signals are messages sent to processes to notify them of certain events. These could be sent between processes, but the most common signals are sent by the kernel. The kernel will send a signal to a process if there is a hardware issue, like an attempt to divide by zero, or to access a non-existent address in memory. The kernel will also send signals to let processes know of any condition that affects them, like the reading of a file being completed, or a terminal being closed; more importantly, if the allotted CPU time for a process has concluded and needs to be terminated or sent to a waiting state. The kernel may also notify the process with a signal if the user pressed specific combinations of keys, like Ctrl-C or Ctrl-\, that have a special meaning for processes as we will see below.

Signals are codified with integer numbers greater than 1. The kernel uses the first 31 signals for specific well known messages. They are known as the ***standard signals***. Each of these standard signals has a name that begins with the prefix ***SIG*** followed by three or four other letters. All of them are defined in the ***<signal.h>*** library that must be included when a process deals with signals.

When an event ***generates*** a signal, it is ***delivered*** to the process, if the process is currently running in the CPU, it receives the signal immediately. If the process is waiting or in the ready queue, the signal will be delivered as soon as the process goes back to the running state. In the time between the generation and the actual delivery of the signal we said that the signal is ***pending***. Once the signal is delivered, it normally elicits a ***default action*** from the process. These default actions could be one of following five:

1. ***Ignore*** the signal, basically do nothing.
2. ***Terminate*** the process. This is an abnormal termination or an abnormally ending (***abend***), because the process did not terminate with ***exit()***.
3. Terminate with a ***Core Dump***. This is similar to terminate, but the virtual memory of the process is dumped on a file/. This can later be used for debugging if needed.
4. ***Stop*** execution of the process. The process is sent to the ready queue to wait until it is allowed to be run again with the following default action.
5. ***Resume*** execution of the process. A previously stopped process can become ready to run if this is the default action for the signal.

The previous list shows the second way in which a process may be terminated: when it receives a signal with a default action of termination. This signal is usually sent by the kernel, but other processes may also send them if authorized to do so. The following signals may terminate a process:

- **SIGTERM (number 15)** is the basic signal to request termination of a process.
- **SIGINT (number 2)** is called the "program interrupt". It is similar to SIGTERM, and it is usually generated by pressing the ***Ctrl-C*** buttons in the keyword.
- **SIGQUIT (number 3)** is also similar to SIGTERM, but this time is generated by pressing the ***Ctrl-\*** buttons in the keyword.The default action on this signal is actually "core dump", so a file with the virtual memory is generated for debugging..
- **SIGKILL (number 9)** is the "sure kill" signal and it is designed to obtain unconditional termination of a process. This signal will terminate the process "no-matter-what", and the process cannot ignore, block, or handle the signal, as we will see one can do with other signals, next.

The top left cell in the figure below presents the program ***run_forever.c*** that all it does is to consume CPU cycles. It has a loop that runs forever doing nothing. We can use this program to review the four signals above. We see in the bottom left cell that when the program runs in the foreground, we can terminate it by pressing either ***Ctrl-C*** or ***Ctrl-\***. In the first case the keystrokes makes the kernel send a SIGINT signal. When the program receives it, the process terminates. The second case is similar, but the signal sent is SIGQUIT, and its default action is to terminate, but creating a core dump. In the middle cell we see the same process running, but this time we use the ampersand (&) to send it to the background. To terminate this process, we need to use the system call ***kill()***. Despite its name, the main purpose of this system call is to send signals to processes. We can use the shell command ***KILL*** that generates this system call. The format of this command is ***KILL -# pid***, where the hash is replaced by the signal number and the pid parameter is replaced by the pid of a process that is currently running. In the middle cell panel the SIGTERM (number 15) is sent to the process for termination. A similar scenario is depicted in the right cell, but this time the signal used is SIGKILL (number 9) that terminates the process unconditionally.

```
#include <stdbool.h>
int main(int argc, char *argv[])
{
   while (true) { }
}
```

```
master $run_forever
^C
master $run_forever
^\Quit (core dumped)
master $
```

```
master $run_forever &
[1] 4823
master $ps
   PID TTY        TIME CMD
  2605 pts/0    00:00:00 bash
  4823 pts/0    00:00:03 run_forever
  4828 pts/0    00:00:00 ps
master $kill -15 4823
[1]+  Terminated                 run_forever
master $ps
   PID TTY        TIME CMD
  2605 pts/0    00:00:00 bash
  4837 pts/0    00:00:00 ps
master $
```

```
master $run_forever &
[1] 4854
master $ps
   PID TTY        TIME CMD
  2605 pts/0    00:00:00 bash
  4854 pts/0    00:00:03 run_forever
  4859 pts/0    00:00:00 ps
master $kill -9 4854
[1]+  Killed                     run_forever
master $ps
   PID TTY        TIME CMD
  2605 pts/0    00:00:00 bash
  4870 pts/0    00:00:00 ps
master $
```

Notice that when a shell process has background processes, it assigns them a ***job control number*** or ***JobID***. In the figure above, that number is 1 surrounded by square brackets [1]. Use that number to refer to this process. We can review all jobIDs with the bash command ***jobs***. The ***fg*** and ***bg*** commands can be used to move jobs to/from foreground or background respectively. These jobs also receive signals with the kill command. The following figure shows the run_forever.c program run three times on the shell and being handled by the these various job control commands:

```
master $run_forever &
[1] 3300
master $run_forever &
[2] 3394
master $run_forever &
[3] 3417
master $ps
    PID TTY          TIME CMD
   2647 pts/0    00:00:00 bash
   3300 pts/0    00:00:04 run_forever
   3394 pts/0    00:00:02 run_forever
   3417 pts/0    00:00:01 run_forever
   3399 pts/0    00:00:00 ps
```

Running ***run_forever*** in the background three times

`master $jobs`
`[1]   Running                  run_forever &`
`[2]-  Running                  run_forever &`
`[3]+  Running                  run_forever &`
	Using command ***jobs*** to display them

`master $fg %2`
`run_forever`
`^Z`
`[2]+  Stopped                  run_forever`
`master $jobs`
`[1]   Running                  run_forever &`
`[2]+  Stopped                  run_forever`
`[3]-  Running                  run_forever &`
	Sending job [2] to the foreground.
	Stopping it with ***Ctrl-Z***.

`master $bg %2`
`[2]+ run_forever &`
`master $jobs`
`[1]   Running                  run_forever &`
`[2]-  Running                  run_forever &`
`[3]+  Running                  run_forever &`
	Sending job [2] to the background again

`master $kill -STOP %3`
 	
`[3]+  Stopped                  run_forever`
`master $jobs`
`[1]   Running                  run_forever &`
`[2]-  Running                  run_forever &`
`[3]+  Stopped                  run_forever`
	Stopping job [3] with the SIGSTOP signal

`master $kill -CONT %3`
`master $jobs`
`[1]   Running                  run_forever &`
`[2]-  Running                  run_forever &`
`[3]+  Running                  run_forever &`
	Resuming job [3] with the SIGCONT signal

`master $kill -TERM %3`
`[3]+  Terminated               run_forever`
	Terminating job [3] with the SIGTERM signal

`master $kill -INT %2`
`[2]+  Interrupt                run_forever`
	Terminating job [2] with the SIGINT signal

`master $kill -KILL %1`
`[1]+  Killed                   run_forever`
`master $jobs`
`master $`
	Terminating job [1] with the SIGKILL signal

In general, a process may decide not to take the default action requested by a signal, instead it may allow for the signal to be ignored, or it may decide to do something else when it receives it. A process changing its default action under a signal is said to be ***changing its signal disposition***. This can be done if the process invokes the ***signal()*** system call from the ***<signal.h>*** library that has the following prototype:

```
void (signal ( int sig, void (*handler) (int) )  )
```

The parameters for the ***signal()*** system call are the signal number to which the process wants to change its disposition and a pointer to a ***handler function***. The process must have access to the handler function and the handler function must receive an integer and return void. The handler function will contain the actions the process wants to be executed when the signal arrives. A common use of the handler function is to tidy up before the process is terminated, by returning resources, like memory previously requested with malloc, or close files, before calling the ***exit()*** system call itself.

The left cell in the following figure shows the program ***catching_signals.c*** that changes the signal disposition for the SIGINT, SIGQUIT and SIGTERM signals using the ***signal()*** system call. SIGINT and SIGQUIT change their disposition to the handler function named *myHandler* that prints the number and the description of the signal received. The description is obtained with the ***strsignal()*** system call.On the other hand, SIGTERM has its disposition changed to ignore, using the SIG_IGN parameter instead of the handler function. The right cell of the figure below shows the performance of the catching_signals process when the ***KILL*** command is used with the various terminating signals. Notice that the SIGKILL signal cannot be ignored or handled. As indicated before. Attempts to do so will not compile.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <signal.h>
#include <string.h>
#include <ctype.h>

// Simple handler to display the message received
static void myHandler(int sig) {
   printf("Signal Received: %d ==> %s\n", sig, strsignal(sig));
   return;
}

int main(int argc, char *argv[])
{
   // Changing the disposition of SIGINT, SIGQUIT and SIGTERM
   if (signal(SIGINT, myHandler) == SIG_ERR) {
          printf("Could not initialize handler for SIGINT\n");
          exit (1);
   }
   if (signal(SIGQUIT, myHandler) == SIG_ERR) {
          printf("Could not initialize handler for SIGQUIT\n");
          exit (1);
```

```
master $catching_signals &
[1] 5119
master $kill -2 5119
Signal Received: 2 ⇒ Interrupt
master $kill -3 5119
Signal Received: 3 ⇒ Quit
master $kill -15 5119
master $ps
   PID TTY          TIME CMD
  2605 pts/0    00:00:00 bash
  5119 pts/0    00:01:38 catching_signals
  5138 pts/0    00:00:00 ps
master $kill -9 5119
[1]+  Killed                  catching_signals
master $ps
   PID TTY          TIME CMD
  2605 pts/0    00:00:00 bash
  5147 pts/0    00:00:00 ps
master $
```

```
   }
   if (signal(SIGTERM, SIG_IGN) == SIG_ERR) {
         printf("Could not initialize handler for SIGTERM\n");
         exit (1);
   }
//    SIGKILL cannot be added. It does not compile
   while (true) {    }
}
```

Although by the description of the ***signal()*** system call, it appears to be returning void, it is in fact returning a pointer to the previous disposition the signal had. This is handy when we want to preserve the previous handling conditions to be restored at a later point in time. The left cell of the following figure shows the program ***swapping_handlers.c*** that alternates between the *myHandler* handler function we previously saw, and the default handling for the SIGINT signal (***Ctrl-C***). When the program invokes the ***signal()*** system call to set the disposition to myHandler for the SIGINT signal, it returns the default handling of this signal that is stored in a pointer named *backhandler*, with the same signature as the myHandler function. The *backhandler* function is later restored with another call to ***signal()***. The right cell of the figure below shows that while myHandler is in use, the process cannot be terminated with ***Ctrl-C***, but this behavior stops when the *backhandler* is in charge.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <signal.h>
#include <string.h>

// Simple handler to display the message received
static void myHandler(int sig) {
   printf("Receiving requests: %d => %s\n", sig, strsignal(sig));
   return;
}

// generic pointer to a handler function with a int parameter
void (*back_handler)(int);

int main(int argc, char *argv[])
{
   while (true) {          // Loop oscillates between two states
      // State 1: myHandler is catching SIGINT
      back_handler = signal(SIGINT, myHandler);
      if (back_handler == SIG_ERR) {
         printf("Could not initialize handler for SIGINT\n");
              exit (1);
      }
       printf("myHandler is active ....\n");
```

```
master $swapping_handlers
myHandler is active ….
^CReceiving Requests: 2 ⇒ Interrupt
            Default Handler is active ….
myHandler is active ….
^CReceiving Requests: 2 ⇒ Interrupt
            Default Handler is active ….
myHandler is active ….
^CReceiving Requests: 2 ⇒ Interrupt
            Default Handler is active ….
^C
master $
```

```
        sleep(3);

     // State 2: The default action (terminate) is catching SIGINT
     back_handler = signal(SIGINT, back_handler);
     if (back_handler == SIG_ERR) {
        printf("Could not initialize handler for SIGINT\n");
            exit (1);
     }
        printf("\tDefault Handler is active ....\n");
        sleep(3);
   }
}
```

A process may also block the delivery of a signal. This will do nothing to the signal, but it may allow the process to complete some activity that must be done uninterrupted by the disposition of the signal. To block a signal, the process keeps track of a ***blocked set***, this a set of signals with data type ***sigset_t*** that collects the numbers of all signals to be blocked by the process. A signal will remain blocked until it is released, at which point is delivered.

The blocked set must be initialized as empty with the ***sigemptyset()*** system call. To which we must add all the signals we wish to block, one by one, with the ***sigaddset()*** system call. We will also need the ***sigprocmask()*** system call to manage the blocked set. The prototypes for these system calls are:

```
int sigemptyset ( sigset_t *set )

int sigaddset ( sigset_t *set, int sig )

int sigprocmask (int how, const sigset_t *set, sigset_t *oldset )
```

The top cell of the following figure contains the program ***blocking_signals.c*** with three sets of signals: *blockedSet* for all signals we wish to block, *pendingSet* for all pending signals, and *defaultSet* for all signals that are blocked by default for the process. *blockedSet* is initialized as empty with the **sigemptyset(**) system call, then the ***SIGINT*** and ***SIGQUIT*** signals are added to this set with the ***sigaddset()*** system call.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <signal.h>
#include <string.h>
```

```
// Simple handler to display acknowledge receipt of signal and exit
static void myHandler(int sig) {
   printf("Blocking is no longer active .... Signals delivered.\n");
   exit(0);
}

int main(int argc, char *argv[])
{
   sigset_t blockedSet;        // Set of signals to be blocked
   sigset_t pendingSet;        // Set of signals that are pending
   sigset_t defaultSet;        // Set of signals from start of process

   // Initializing Blocked set of signals
   sigemptyset(&blockedSet);                    // empty set to start
   sigaddset(&blockedSet, SIGINT);              // adding SIGINT
   sigaddset(&blockedSet, SIGQUIT);             // adding SIGQUIT

   // Changing the disposition of SIGINT and SIGQUIT
   if (signal(SIGINT, myHandler) == SIG_ERR) {
          printf("Could not initialize handler for SIGINT\n");
          exit (1);
   }
   if (signal(SIGQUIT, myHandler) == SIG_ERR) {
          printf("Could not initialize handler for SIGQUIT\n");
          exit (1);
   }

   while (true) {           // Loop oscillates between two states
      // State 1: Blocking SIGINT AND SIGQUIT
      if (sigprocmask(SIG_BLOCK, &blockedSet, &defaultSet) == -1) {
         printf("Error trying to block signals SIGINT and SIGQUIT\n");
             exit (1);
      }
          printf("Blocking of SIGINT(2) and SIGQUIT(3) is active ....\n");
          sleep(5);
          if (sigpending(&pendingSet) == -1) {
         printf("Error trying to retrieve pending signals\n");
             exit (1);
      }
      if (sigismember(&pendingSet, SIGINT)) printf("\nSIGINT is pending\n");
      if (sigismember(&pendingSet, SIGQUIT)) printf("\nSIGQUIT is pending\n");

      // State 2: Going back to default
      if (sigprocmask(SIG_SETMASK, &defaultSet, NULL) == -1) {
         printf("Could not restore original set of Blocks\n");
             exit (1);
      }
          printf("\tBlocking is NOT active ....\n");
          sleep(5);
   }
}
```

```
master $blocking_signals
Blocking of SIGINT(2) and SIGQUIT(3) is active ....\n
^C^C^\^\SIGINT is pending
SIGQUIT is pending
myHandler is active ….
            Blocking is NO longer active …. Signals delivered.
master $
```

The ***sigprocmask()*** system call is used to enable and disable a blocking set. It takes a first parameter (named how in the prototype) to indicate the action to be taken. This parameter could be one of the following values:

- ***SIG_BLOCK*** to add a set of signals to be blocked
- ***SIG_UNBLOCK*** to remove a set of signals from the blocked set
- ***SIG_SETMASK*** to replace an entire set of blocked signals with another set

In state 1 of the loop inside the program, we first use ***sigprogmask()*** to add the signals in the *blockedSet* to the set of signals currently blocked. The old set of signals is returned in the *defaultSet* set of signals. In our case, this *defaulSet* has no signals blocked. In state 2 of the loop, ***sigprogmask()*** is used to replace the set of blocked signals with the *defaultSet* that was saved with the ***SIG_SETMASK*** parameter. Alternatively, it could just unblock the same signals with the ***SIG_UNBLOCK*** parameter. The general effect of this loop is that initially it blocks the SIGINT and SIGQUIT signals, as can be seen in the bottom cell of the picture. Once the old set of blocked signals (*defaultSet*) is restored in state 2 of the loop, SIGINT and SIGQUIT are no longer blocked and they are delivered. The program uses a handler function that displays the signal that is received first, and stops the program naturally with ***exit(0)***.

Before entering into state 2, the program reports which signals were pending. To do so, it uses the ***sigpending()*** and ***sigismember()*** system calls that have the following prototypes:

```
int sigpending ( sigset_t *set )

 int sigismember ( const sigset_t *set, int sig )
```

The ***sigpending()*** system call retrieves a set of signals that are pending. As it can be seen in the bottom cell of the previous figure, the user pressed ***Ctrl-C*** and ***Ctrl-\*** twice each, these produce the SIGINT and SIGQUIT signals that are part of the pending signals set retrieved by the ***sigpending()*** system call (*pendingSet*). To verify that a signal is part of a set we use the ***sigismember()*** system call. In our example, it is separately checking if the SIGINT and SIGQUIT signals are members of the *pendingSet* set of signals, and reporting on it. Notice here that the pending set only reports once per every different signal received. In other words, it does not matter how many times a signal is sent, the pending signal set will only contain one acknowledgment per each kind of signal.

It is important to emphasize that the ***SIGKILL*** signal cannot be blocked, for the same reason that it cannot be ignored or handled. This signal guarantees termination of a process, and, therefore the process cannot do anything about it. On the other hand, it should only be used as a last resort, because if a process uses a handler to have a clean exit, without resources left open or engaged, when the ***SIGKILL*** signal is received, the process will not be able to use the handler at all.

Execution Times

The time a process takes to run on a computer depends on many factors. To begin with, it depends on the speed of the CPU, the registers, the memory and the secondary storage. It also depends on the kind and the number of instructions in the process, how long the process waits in the ready queue, how many times the process is interrupted by the kernel, and how long it waits to go back to the ready queue.With so many factors to consider, there are many possible time measurements that can be made. From all these possibilities, the most important to consider are the following three:

- ***Real Time*** for a process is the time from when the kernel begins to load it in memory until it is terminated. This corresponds to our classical idea of time keeping.
- ***User Time*** for a process is the time that it spends while running in the CPU in user mode. It is smaller than the real time because it only considers the time when the process is in running state, performing instructions. If a process is interrupted many times, the user time adds up all the times the process was executing statements before these interruptions.
- ***System Time*** for a process is the time the kernel takes performing tasks on behalf of the process. This includes the loading time, the time used by the kernel to handle all the interruptions for the process, and the times to do context switching for the process. It does not include any waiting time from the process.

The command ***time*** used in a shell provides measurements of these three times. This command is prefixed to the normal program call to be made. For example, the following table shows the programs compound.c and identify being timed with the time command:

```
master $time compound 1000 2 7
          Table of Compounded Amounts
                    Created by:master
          Amount   After Period:
          1000.00           0
          1020.00           1
```

```
master $time identify
I am process 40833
I belong to the group 40833
My parent is process 40228
My parent belongs to the group 40228
My session ID is 40228
My parent session ID is also 40228
```

```
        1040.40          2
        1061.21          3
        1082.43          4
        1104.08          5
        1126.16          6
        1148.69          7
        Accumulated Interests:148.69

real    0m0.002s
user    0m0.001s
sys     0m0.001s
master $
```

```

real    0m5.003s
user    0m0.002s
sys     0m0.001s
master $
```

If a process is short and it is not often interrupted, the real time may be similar to the addition of the user and system times. Notice that the difference of 5 seconds in these figures on the identify program is due to sleeping time forced on the process. This waiting time is not added neither to the user time, nor to the system time.

Multiprogramming

Multiprogramming is the ability of an operating system to hold more than one process running in memory at the same time. When we open a text editor to write a program, and at the same time we open a Web Browser to review a website, while listening to music from our favorite streaming service, all done in the same computer, we are using multiprogramming. Each one of these activities may generate one or more processes that are trying to run on the computer. From the point of view of the user, all these processes may appear to be running at the same time, but all of them require access to a CPU to perform their instructions. In the simplest case with a single CPU in the computer, all processes will take turns accessing it. The operating system is the one deciding which process accesses the CPU at any given time and for how long. Therefore, the decisions that it makes will affect the performance of each one of the applications running in the system. This is one of the main responsibilities for the operating system, as a CPU scheduler.

Forking

In Linux, multiprogramming is achieved by a technique called ***forking***. This requires that a process “clones” itself using the ***fork()*** system call with no parameters. When a process makes this call, the kernel creates a copy of the current process that will run independently and concurrently with the original process and all other processes already in the system. The newly created process (called the child process) receives copies of all segments from the original process (called the parent process), as well as copies of the file descriptors. After the call to fork, both processes may begin execution of the next sentence in the program. If no allowances are made, both processes may execute the same sentences when they take control of the CPU. However, the call to fork returns different values for parent and child processes. The parent process receives the process ID from the children just created, however the children process receives a value of zero. This fact can be used to request each process to perform different instructions. The following figure presents the ***simplefork.c*** program that shows a process that forks. The value this system call returns is used in a switch statement to assign different instructions to parent and child processes. Notice that if the call to fork returns a -1 if the call fails. The fork system call is declared in the ***<unistd.h>*** header.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
   pid_t myPID;
   pid_t forkResult;    // status of the fork

   switch (forkResult = fork()) {
   case -1:     // If the call to fork did not work
      printf("There is a problem with fork.");
      exit(1);

   case 0:      // If the forked succeeds the child receives a zero from fork
      // Child Zone
      myPID = getpid();
      printf("\tI am the child process.\n" \
           "\tThe fork gave me a %d, but my PID is %d\n",
           forkResult, myPID);
      break;
   default:     // If the forked succeeds the parent receives the pid of the child
      // Parent Zone
      // Following sleep sentence can be uncommented to let child execute first
      //  sleep(1);
      myPID = getpid();
      printf("I am the parent process.\n" \
          "The fork gave me my child PID (%d), but my own PID is %d\n",
           forkResult, getpid());
   }
   // Parent and children have access to this Zone
   if (forkResult == 0) printf("\t");
   printf("Bye from %d.\n\n",myPID);

   exit(0);
}
```

When the program runs, it produces a result similar to:

```
master $simplefork
I am the parent process.
The fork gave me my child PID (8288), but my own PID is 8287
Bye from 8287.

        I am the child process.
        The fork gave me a 0, but my PID is 8288
        Bye from 8288.
master $
```

The messages show that the parent process kernel is run immediately after the fork before the child process. This is the current way that Linux schedules processes after a fork. Earlier versions of Linux may have scheduled the child process before the parent process, but there is no consensus if either arrangement is more advantageous than the other. Programmers should not rely on either outcome, but use other techniques if they require an specific order of execution. In this short program, one can delay the execution of the parent process with a sleep command, to allow access to the CPU to the children process. If the comment prefix is removed in line 25 from the previous program, a the process may produce something like the following:

```
master $simplefork
        I am the child process.
        The fork gave me a 0, but my PID is 8395
        Bye from 8395.

I am the parent process.
The fork gave me my child PID (8395), but my own PID is 8394
Bye from 8394.
master $
```

As previously indicated, copies of the file descriptors from a parent process are given to the child process. Because they are copies, if any of the processes modifies the attributes of the files they reference, like offset placement or access mode, all these changes will be visible and relevant to both processes. If the programmer wants to keep different attributes, they should create file descriptors after fork. Alternatively, they could have duplicates of the file descriptors before fork and then each process will update one of them after fork to suit its particular needs.

Even though a child process should receive copies of segments after fork, this is actually done only when the children require a change in any of them. This strategy is called “copy-on-write” and it avoids doing copies of resources that may not be needed. Before that, both parent process and children process just keep references to the same segments. If any of the processes requires to update the stack or heap, or maybe load a different set of instructions for the text, this is when the kernel will generate the copy with the appropriate change. This is particularly useful in cases where a child is created to run a different program than the one in the parent process, as we will see later.

A typical application of forking is to have a main program that generates tasks, each one to be handled by a different child process. The parent process waits for these processes to complete. The following figure shows the basic structure of such an application in a program called ***forkcycle.c***.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>

int main(int argc, char *argv[])
{
   pid_t myPID;
   pid_t childOutPID;   // PID of returning child
   int nForks;          // Number of requested forks
   int childrenExiting;  // Number of children finishing its tasks
   int time=0;          // Sleeping time for a child process
   int i;               // Counter

   if (argc != 2 || ((nForks = atoi(argv[1]))< 1)) {
      printf("Invalid use. Should be:\n");
      printf("cyclefork #_of_forks\n");
      printf("#_of_forks should be a integer > 0\n");
      exit(1);
   }

   setbuf(stdout, NULL);         // Disabling buffering for stdout

   for (i = 0; i < nForks; i++) {   // Creating a child with decreasing wait time for each loop
      switch (fork()) {
      case -1:     // If the call to fork did not work
         printf("There is a problem with fork.");
         exit(1);

      case 0:                   // Child prints, sleeps and exits
         myPID = getpid();
         printf("Process %d created. Waiting %d seconds.\n", myPID, nForks-i);
         sleep(nForks-i);
         _exit(EXIT_SUCCESS);

      default:                  // Parent continues creating processes
         break;
      }
   }

   childrenExiting = 0;
   while (childrenExiting < nForks) {
      childOutPID = wait(NULL);
```

```
        if (childOutPID < 1) {
            printf("Unexpected Error.");
            exit(1);
        }
        childrenExiting++;
        printf("Children # %d (Pid=%d) exit acknowledged\n",childrenExiting, childOutPID);
    }
    printf("All children are back.\n");
    printf("Bye!\n");
    exit(EXIT_SUCCESS);
}
```

This program expects an integer parameter to indicate the number of children to be created. A loop creates these children. Each one could be performing a different task, but in this program, they just pause for a different number of seconds with the sleep system call and then exit. The parent process waits for them with the wait system call in a loop. The wait system call blocks the execution of any other command from the process until it receives the process ID of a returning child or -1 if there is any error. The parent process increments the count of returning children after any new one is received. The loop ends when all children have returned. Notice that only the parent process exits with a call to exit(status), where the status indicates success or a termination with an error. All children return using the _exit(status) system call. The difference between these two commands to exit is that the exit(status) performs some cleaning before terminating the process (like clearing stream buffers and calling exit handlers) before it calls _exit(status) itself. If children processes where to call exit(status) they may compromise the actions of other children processes and the parent process. In your applications, it is advisable always to let the children processes end with _exit(status) and the parent process with the exit(status) system call. A sample run of this program can be seen in the following figure.

```
master $forkcycle
Process 3651 created. Waiting 2 seconds.
Process 3647 created. Waiting 6 seconds.
Process 3648 created. Waiting 5 seconds.
Process 3649 created. Waiting 4 seconds.
Process 3650 created. Waiting 3 seconds.
Process 3652 created. Waiting 1 seconds.
Process 3646 created. Waiting 7 seconds.
Children # 1 (Pid=3652) exit acknowledged
Children # 2 (Pid=3651) exit acknowledged
Children # 3 (Pid=3650) exit acknowledged
Children # 4 (Pid=3649) exit acknowledged
Children # 5 (Pid=3648) exit acknowledged
Children # 6 (Pid=3647) exit acknowledged
Children # 7 (Pid=3646) exit acknowledged
All children are back.
Bye!
master $
```

The exec() system call

Sometimes it is necessary for a child process to forgo the sets of instructions it inherited from its parent process and open a complete new program.This can be done with the ***exec()*** system call. Once this call is made, the kernel will reload the virtual memory address space of the child process with a new program and update the registers to start the process. However, many other elements inherited from the parent process will remain in place. The process will keep its PID, as well as its parent ID, process group IDs, and session ID. It also keeps all the environment variables inherited from the parent process and the file descriptors with all its settings.The process will execute the new code and when it ends, it will exit and never come back to the parent process. There will be no notification of completion or return to the parent process.

In reality, the ***exec()*** system call does not exist. This is just the generic name for a set of system calls, all beginning with the ***exec*** prefix. The most basic system call is ***execve()***. All other variants are convenient ways to prepare the call to ***execve()*** with different parameters. Basically, all these system calls require three main parameters: 1.The location and name of the program to be executed, 2. The set of all the strings containing the arguments for the program to be executed, just like the ***argv*** parameter that would be created for that program, and 3.The environment variables that are passed to the program to be executed.The variants of this system call prepare these parameters in different ways. It is easier to remember all of them if we relate their names with the parameters they request as described below.

The first decision a programmer has to make on selecting an ***exec()*** system call is to decide if the arguments for the program to be executed are going to be supplied together arranged inside an array (just like ***argv[ ]***) or if they are going to be supplied separately in an explicit list. In the first case we must select a system call starting with the ***execv*** prefix, for a vector (another name for an array). In the second case we must select a system call starting with the ***execl*** prefix, for a list. These system calls have the following prototypes:

```
execv ( const *pathname, char *const argv[ ] )

execl ( const *pathname, char *arg1, char *arg2, … , (char *) NULL)
```

Both require a pointer to the pathname and filename of the program to be executed as the first parameter. The pathname can be absolute or relative. After that, ***execv*** requires an array with the parameters, like ***argv[ ]***, while ***execl*** needs all these parameters described individually. To indicate that the list ended, the last argument for ***execl*** should be NULL, type casted as a pointer to a char. NULL must also be the last argument in the ***argv[ ]*** array. The NULL parameter must be included in either presentation, even if the program to be executed does not require parameters, at least to indicate this fact.

Let us consider the following example to see how to call ***execv()***. It uses the program shown below called ***weather_report.c*** that reports weather conditions using the parameters it receives from ***argv*** and environment variables named "**SKY_COND**" and "**USERNAME**". You may also see that when this program is called on its own, without any parameter, it produces no meaningful report. The only information it contains is its PID and the value of the environment variable "**USERNAME**", generally available for the shell.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
// weather_report
// This program receives weather reports with argv
//  and environment variables
// Author: Guillermo Tonsmann Ph.D.

int main(int argc, char *argv[])
{
   pid_t myPID = getpid();       // getting local PID
   char *sky_condition;          // repository of environment variable
   char *my_username;            // $USERNAME

   printf("I am process %d running (%s)\n", myPID, argv[0]);

   // Using the argv array
   if (argc < 2) {
      printf("Process %d received no weather report\n",myPID);
   }
   else {
      printf("Process %d received prognosis of weather: %s\n", myPID, argv[1]);
   }

   // Using the environment variable
   if ((sky_condition = getenv("SKY_COND")) == NULL)  sky_condition = "Not Available";
   if ((my_username = getenv("USERNAME")) == NULL)  my_username = "Not Available";
   printf("Process %d - Sky condition: %s\n",myPID,sky_condition);
   printf("Process %d - $weather report courtesy of: %s\n\n",myPID,my_username);
   return 0;
}
```

```
master $weather_report
I am process 3388 running (weather_report)
Process 3388 received no weather report
Process 3388 - Sky condition: Not Available
Process 3388 - $weather report courtesy of: master

master $
```

The following figure shows the ***fork_execv.c*** program that forks to create three child processes. Each of them prepares the parameters to run the weather_report program with ***execv()*** system calls, including the creation of different ***child_argv[ ]*** arrays and different values being set for the "**SKY_COND**" environment variable for each child. A sample output of the program executing shows the scrambled output from the parent and children processes, because the kernel may run them in any order it can. We can follow the output for each process by following their stated PIDs, and we can see that each call to ***weather_report*** produces different outputs.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#define NFORKS 3

// fork_execv
// This program illustrates how to fork and
// how to make the children run another program with execv
// Author: Guillermo Tonsmann Ph.D.
// Version 1.0

int main(int argc, char *argv[])
{
   pid_t myPID;                                    // Current PID
   char *prog_to_call =  "weather_report" ;               // Program to be called

   char *weather[NFORKS+1] = { "No Rain", "Hot and Humid", "Thunderstorm", "Snowing" };
           // array of weather to send to processes with argv

   char *sky_conditions[NFORKS+1] = {"Clear", "Scattered", "Cloudy", "OverCast"};
           // array of sky conditions to send the processes with environment variables

   setbuf(stdout, NULL);         // Disabling buffering for stdout

   // Adding the environment variable "SKY_COND"
   if (setenv("SKY_COND", sky_conditions[0],0) == -1) {
     printf("Could not add environment variable\n");
     exit(1);
   }

   // Printing Parent information
   myPID = getpid();
   printf("Parent Process %d created %d children\n", myPID, NFORKS);
   printf("Parent Process %d - Prognosis of weather: %s\n", myPID, weather[0]);
   printf("Parent Process %d - Sky condition: %s\n", myPID, getenv("SKY_COND"));


   for (int i = 1; i <= NFORKS; i++) {   // Creating children with different calls to weather_report
      switch (fork()) {
      case -1:    // If the call to fork did not work
```

```
            printf("There is a problem with fork.");
            exit(1);

        case 0:
            myPID = getpid();                                    // Getting PID

            // Child prepares array of argv for the program to be called
            char *child_argv [] = { prog_to_call, weather[i], NULL };    // argv to send to exec

            // Changing environment variable "SKY_COND"
            if (setenv("SKY_COND", sky_conditions[i],1) == -1) {
               printf("Process %d could not change environment variable\n", myPID);
            }
           // Printing process Information before call to execv
           printf("Process %d created\n", myPID);
           printf("Process %d Calling exec (%s)\n", myPID, prog_to_call);

          // Call to exec variant execv with a vector array for argv
          execv(prog_to_call, child_argv);

            // Child will never return

        default:                   // Parent continues creating processes
            break;
        }
    }
    printf("Parent Process %d exiting.\n\n", myPID);
    exit(EXIT_SUCCESS);
}
```

```
master $fork_execv
Parent Process 3412 created 3 children
Parent Process 3412 - Prognosis of weather: No Rain
Parent Process 3412 - Sky condition: Clear
Parent Process 3412 exiting.

Process 3415 created
Process 3415 Calling exec (weather_report)
Process 3414 created
Process 3414 Calling exec (weather_report)
I am process 3415 running (weather_report)
Process 3415 received prognosis of weather: Snowing
Process 3415 - Sky condition: OverCast
Process 3415 - $weather report courtesy of: master

I am process 3414 running (weather_report)
Process 3414 received prognosis of weather: Thunderstorm
Process 3414 - Sky condition: Cloudy
Process 3414 - $weather report courtesy of: master
Process 3413 created
Process 3413 Calling exec (weather_report)
I am process 3413 running (weather_report)
Process 3413 received prognosis of weather: Hot and Humid
Process 3413 - Sky condition: Scattered
Process 3413 - $weather report courtesy of: master

master $
```

If instead of calling ***execv()***, we were to replace it with a call to ***execl()***, we will not have to create the ***child_argv[ ]*** array, just use its contents directly in the call, as follows:

```
execl(prog_to_call, prog_to_call, weather[i], (char *) NULL)
```

An additional set of ***exec*** system calls requires an array of environment variables as an added parameter. to include in addition to the ones the process gets from its parent process. These system calls are similar to ***execv*** and ***execl***, but include an added letter "***e***" in their names: ***execve*** and ***execle***. The following figure shows how this array could have been created and submitted in an ***execve*** system call from fork_execv.c.

```
char for_env[100] = "SKY_COND=";            // Auxiliary string
strcat(for_env,sky_conditions[i]);                  // Modify SKY_COND value
char *child_env[] = { for_env, "WIND=Strong", NULL };       // Modify env for exec

// Call to exec variant execve with a vector array for argv and vector array for environment
execve(prog_to_call, child_argv, child_env);

// Alternative call to exec variant execle with a list of char for argv and vector array for environment
execle(prog_to_call, prog_to_call, weather[i], (char *) NULL, child_env);
```

Finally, a last set of ***exec*** system calls allows the use of just a filename instead of a pathname for their first parameter. The filename will have to be found inside the directories declared in the **PATH** environment variable. Once again , these system calls are similar to ***execv*** and ***execl***, but include an added letter "***p***" in their names: ***execvp*** and ***execlp***. Nothing else changes, but the meaning of the first parameter.

Of orphans, zombies and daemons

Parent processes usually wait for their children to end execution, and when they do, the parent process requests their removal from the table of running processes managed by the kernel. Consequently, all resources allocated to the children processes are released, to be used by other processes, and their entry in the table of running processes is removed.

Children processes whose parents terminate before themselves are considered ***orphans***. Because no process may be left without a parent, the ***init*** process “grandfathers” all orphaned children. The ***init*** process becomes the parent ID of these processes. The orphaned processes are then “waited” by the ***init*** process, and when they finish, ***init*** removes them from the table of running processes managed by the kernel and deallocates its resources.This is the reason why the ***init*** process is also known as the ***reaper process***.

If a child process terminates before the parent process issues the ***wait()*** system call, the kernel will still release its resources, but a record of the child process will remain in the table of running processes. This allows the parent a chance to issue the ***wait()*** system call at a later stage and receive a reply from the terminated process. Because in practice the child process no longer exists, but there is still a record of it in the table of running processes, this process becomes what is known as a ***zombie*** process. Zombie processes cannot be killed by normal means, and the only way to eliminate them is to issue a call to ***wait()***. If zombie processes are not removed from the table of running processes they may remain there indefinitely. Zombie processes in which the parent terminates without issuing the ***wait()*** system call become orphan processes, and therefore are waited by the init process that removes them from the table of running processes.

Other important processes in the operation of Linux systems are ***daemons***. Daemons are special background processes, unconnected to a terminal, that provide special services for the system. Because these services may be needed at any point of the system operation, they are usually running from the system start-up through shut down. Examples of daemons are the processes that control networked printers, the ones that allow secure remote login to the system, and the ones that handle requests to web page servers.

Process Resource Limits

Resources in a computer are limited. For example, there is just a certain amount of memory in RAM, a certain storage capacity in files, and a certain amount of time to execute programs. All these limits can be set, so that all users can fairly share these resources. To do so, the following two system calls from the ***<sys/resource.h>*** library can be used:

```
int getrlimit ( int resource, struct rlimit *rlim )

int setrlimit ( int resource, const struct rlimit *rlim )
```

The ***getrlimit()*** system call retrieves the limits of the *resource* parameter into a *rlimit* structure, while the ***setrlimit()*** system call sets the limits for the *resource* parameter from the *rlimit* structure.

The various system resources are codified with a number that is also identified by a name. Some of the most useful resource names are shown below:

Resource name	**Description**
RLIMIT_CPU	CPU time in seconds
RLIMIT_AS	Process virtual memory size in bytes
RLIMIT_STACK	Size of the stack segment in bytes
RLIMIT_DATA	Process data segment in bytes
RLIMIT_NOFILE	Maximum number of file descriptors plus 1
RLIMIT_FSIZE	File size in bytes
RLIMIT_NPROC	Number of processes open for the real user ID

The *rlimit* structure has the following declaration:

```
struct rlimit {
   rlim_t  rlim_cur;    // Soft limit (actual current limit)
   rlim_t  rlim_max;  // Hard limit (maximum ceiling for
rlim_cur
}
```

We use this structure, because every user may have a hard limit for a resource over which it cannot go. That is the *rlim_max*. However, a user may be using less than that limit, therefore the current limit is specified by *rlim_cur*. Privileged users, like the superuser or root, may modify these limits as they see fit, as long as the current limit is smaller than or equal to the hard limit. Other users may lower the current limit, or raise it up to the hard limit, but cannot go over it.

The following figure shows the program ***show_limits.c*** that displays the soft and hard limits of some resources. This program uses a *printRlimit* function taken from "*The Linux Programming Interface*" by Michael Kerrisk (recommended textbook for this course). This function uses the ***getrlimit()*** system call to obtain the limits , but before printing them, it checks if they are too large, and if so it prints "infinite" as the value. Similarly, it also checks if the specific Linux implementation has a limit that is beyond the number allowed to be stored in the rlim_t data type. If that is the case, the *rlim_cur* may hold a predefined *RLIM_SAVED_CUR* value, and equivalently, the *rlim_max* may hold a predefined *RLIM_SAVED_MAX* value. The detection of these values will make the function to print "unrepresentable" instead. The bottom cell shows the program running for a privileged user.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>

// Function printRLimit taken from Listing 36-2 Page 758 from
// The Linux Programming Interface by Michael Kerrisk
int printRlimit(const char *, int);

int main(int argc, char *argv[])
{
   printRlimit("CPU Time limits (seconds):", RLIMIT_CPU);
   printRlimit("Process Virtual Memory limits (bytes):", RLIMIT_AS);
   printRlimit("Stack Segment limits (bytes):", RLIMIT_STACK);
   printRlimit("Process Data Segment limits (bytes):", RLIMIT_DATA);
   printRlimit("Number of File Descriptors (+1):", RLIMIT_NOFILE);
   printRlimit("File Size (bytes):", RLIMIT_FSIZE);
   printRlimit("Number of Processes allowed to open:", RLIMIT_NPROC);
}

/* Print 'msg' followed by limits for 'resource' */
int printRlimit(const char *msg, int resource)
{
   struct rlimit rlim;

   if (getrlimit(resource, &rlim) == -1)
      return -1;

   printf("%s soft=", msg);
   if (rlim.rlim_cur == RLIM_INFINITY)
      printf("infinite");
#ifdef RLIM_SAVED_CUR          /* Not defined on some implementations */
   else if (rlim.rlim_cur == RLIM_SAVED_CUR)
      printf("unrepresentable");
#endif
   else
      printf("%lld", (long long) rlim.rlim_cur);

   printf("; hard=");
   if (rlim.rlim_max == RLIM_INFINITY)
      printf("infinite\n");
#ifdef RLIM_SAVED_MAX          /* Not defined on some implementations */
   else if (rlim.rlim_max == RLIM_SAVED_MAX)
      printf("unrepresentable");
#endif
   else
      printf("%lld\n", (long long) rlim.rlim_max);

   return 0;
}
```

```
master $show_limits
CPU Time limits (seconds): soft=infinite; hard=infinite
Process Virtual Memory limits (bytes): soft=infinite; hard=infinite
Stack Segment limits (bytes): soft=8388606; hard=infinite
Process Data Segment limits (bytes): soft=infinite; hard=infinite
Number of File Descriptors (+1): soft=1024; hard=524288
File Size (bytes): soft=infinite; hard=infinite
Number of Processes allowed to open: soft=5377; hard=5377
master $
```