TRƯỜNG ĐẠI HỌC BÁCH KHOA – ĐẠI HỌC QUỐC GIA TP.HCM
TRUNG TÂM KỸ THUẬT ĐIỆN TOÁN
268 Lý Thường Kiệt, Phường 14, Quận 10, TP.Hồ Chí Minh
Điện thoại: 84-8-3864 7256 ext 5371 – Fax: 84-8-3865 8687
Website: **www.cce.hcmut.edu.vn**
E-mail: **dientoan@hcmut.edu.vn**

Lab 2 - Process

1. Theory:

1.1. Introduction:

A process is a program in execution. During execution, a process can change its state. The state of a process is defined in part by the current activity of that process. Figure 1.1 depicts how the process changes its state.

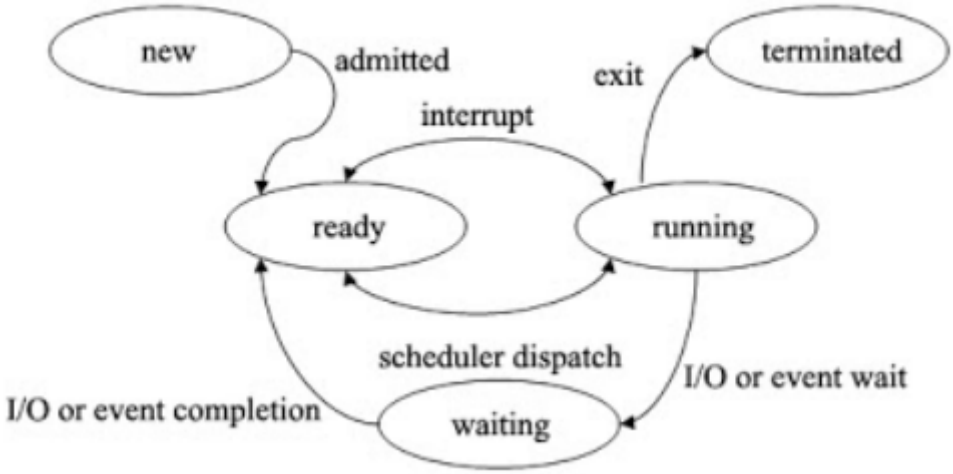


Fig 1.1: Diagram of process state.

To keep track of processes, the operating system maintains a process table (or list). Each entry in the process table corresponds to a particular process and contains fields with information that the kernel needs to know about the process. This entry is called a Process Control Block (PCB). Some of these fields on a typical Linux/Unix system PCB are:

- Machine state (registers, program counter, stack pointer)
- Parent process and a list of child processes
- Process state (ready, running, blocked)
- Event descriptor if the process is blocked
- Memory map (where the process is in memory)
- Open file descriptors
- Owner (user identifier). This determines access privileges & signaling privileges
- Scheduling parameters
- Signals that have not yet been handled
- Timers for accounting (time & resource utilization)
- Process group (multiple processes can belong to a common group)

1.2. Get information of a process

A process is identified by a unique number called the process ID (PID). Some operating systems (notably UNIX-derived systems) have a notion of a process group. A process group is just a way

to lump related processes together so that related processes can be signaled. Every process is a member of some process group.
A process can find its process ID with the getpid system call. It can find its process group number with the getpgrp system call, and it can find its parent‘s process ID with getppid. For example:

```
#include <cstdio>
#include <cstdlib>
int main(int argc, char **argv){
    printf("Process ID: %d\n", getpid());
    printf("Parent process ID: %d\n", getppid());
    printf("My group: %d\n", getpgrp());
    return 0;
}
```

With this id of process, we can get more information about the process by accessing the directory ***/proc/<pid>/***. For example, with process with PID is 295, we can:

- Extract the command line to execute this process:

```
root@DESKTOP-JT611DE:~# cat /proc/295/cmdline
./example
```

- Extract the current status of this process:

```
root@DESKTOP-JT611DE:~# cat /proc/295/status
Name:   example
State:  S
(sleeping)
Tgid:   295
Pid:
295
PPid:   8
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 3
Groups:
VmPeak: 0 kB
VmSize: 10536 kB
VmLck:  0 kB
VmHWM:  0 kB
VmRSS:  584 kB
VmData: 0 kB
VmStk:  0 kB
VmExe:  4 kB
VmLib:  0
kB
```

```
VmPTE:  0 kB
Threads:        1
SigQ:   0/0
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000000000000
SigCgt: 0000000000000000
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
CapBnd: 0000001fffffffff
Cpus_allowed:   ff
Cpus_allowed_list:      0-7
Mems_allowed:
1
Mems_allowed_list:      0
voluntary_ctxt_switches:
150
nonvoluntary_ctxt_switches:     545
```

1.3. Execution Priority of Processes and niceness Command:

There can be hundreds of processes running on an operating system but the computer hardware only has a limited number of CPU cores, so how can the operating system manage them?
The Linux Operating system denotes priorities of process by niceness value ranging from -20 (highest priority and need to be immediately executed) to +19 (lowest priority). Each running process has a niceness value assigned to it that determines how fast it is executed by the system. Higher priority processes are usually carried out earlier than low priority ones. To change the priority of a running process, we can use “renice” and “nice” commands:

- “nice” command is used to assign a niceness value for a new process before running it. (default niceness value of a process is 0).
- “renice” command is used to reassign a niceness value for a running process.

Syntax for these commands:

```
$ nice -n <niceness_value> [other arguments] <command to start a process>
$ renice -n <niceness_value> [other arguments] -p <process_id>
```

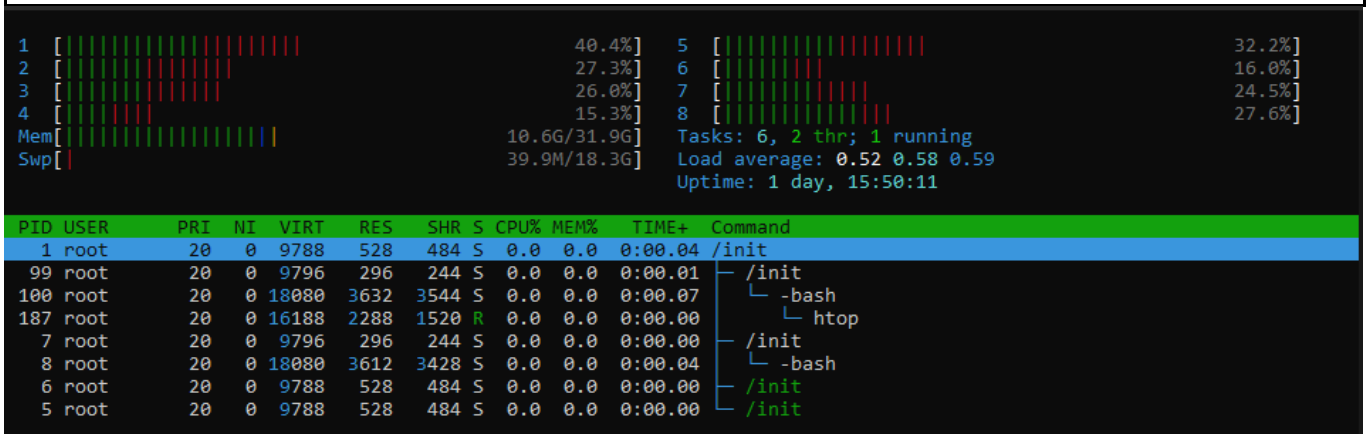
For examples,

```
$ nice -n 15 sleep 30 # assume that ID of this "sleep" process is 1333
$ renice -n -10 -p 1333
```

To check priority of running processes, you can use the htop command, which is described in the Appendix section. There is a noteworthy stuff here that besides niceness value, Linux kernel also

has priority values to denote for the execution priorities of processes. For example, running htop command as follows:

```
$ htop
```



The columns named ***"PRI"*** and ***"NI"*** are respectively priority values and niceness values of processes. These priority values are actual priorities of processes and used by the kernel module to schedule tasks. On the other hand, the *niceness* value is only used for user-space.

1.4. Creating a Process

This lab will introduce three common functions to create a new process

1.4.1. system()

```
#include <stdlib.h>
int system(const char *command);
```

- *Description:*
  - **system()** executes a command specified in command by calling ***"/bin/sh -c command"***, and returns after the command has been completed.
  - The return value of system() is one of the following:
    - If *command* is NULL, then a nonzero value if a shell is available, or 0 if no shell is available.
    - If a child process could not be created, or its status could not be retrieved, the return value is -1 and *errno* is set to indicate the error.
    - If a shell could not be executed in the child process, then the return value is as though the child shell terminated by calling _exit(2) with the status 127.
    - If all system calls succeed, then the return value is the termination status of the child shell used to execute *commands*. (The termination status of a shell is the termination status of the last command it executes.)
- Examples: A C++ program that list all files and directories in the current directory.
  - Run in Linux OS:

```
#include <bits/stdc++.h>
using namespace std;
int main ()
```

```
{
    system("ls");
    return 0;
}
```

○ Run in Windows OS:

```
#include <bits/stdc++.h>
using namespace std;
int main ()
{
    system("dir");
    return 0;
}
```

- Drawbacks:
  - ○ It's a very expensive and resource heavy function call
  - ○ It is not portable (OS dependent). (e.g. the command lines in Windows OS cannot be executed in Linux OS).

1.4.2. fork()

```
#include <unistd.h>
pid_t fork(void);
```

- *Description:*
  - ○ **fork()** creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process. The child process and the parent process run in separate memory spaces. At the time of **fork**() both memory spaces have the same content.
  - ○ Return values: On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and *errno* is set to indicate the error.
- Examples:

```
#include <cstdio>
#include <sys/types.h>
#include <unistd.h>
void forkexample() {
    pid_t child_pid = fork();
    if (child_pid  == 0) {
        printf("Hello from Child \n");
        sleep(50);
    } else if (child_pid == -1){
        perror ( " fork " ) ; /print a system-defined error
    } else {
```

```
        printf("Hello from Parent!, pid: %d\n", getpid());
        sleep(10);
    }
}
int main()
{
    forkexample();
    return 0;
}
```

Question: *Execute the above program 3-4 times, observe the outputs, and explain the phenomenon.*

1.4.3. exec family

The exec family of functions replaces the current running process with a new process. It can be used to run a C/C++ program by using another C/C++ program. It comes under the header file **unistd.h**. Particularly, the available exec functions along with their function parameters are given below:

- **`int execl(const char *path, const char *arg, …, NULL);`**
  - *Description:* In execl() system function takes the path of the executable binary filem or a shell script as the first and second argument. Then, the arguments that you want to pass to the executable followed by **NULL**. Then execl() system function runs the command and prints the output. If any error occurs, then execl() returns -1. Otherwise, it returns nothing.
  - Example: Run command "ls -lh /home"

```
#include <unistd.h>
int main(void) {
  char *binaryPath = "/bin/ls";
  char *arg1 = "-lh";
  char *arg2 = "/home";
  execl(binaryPath, binaryPath, arg1, arg2, NULL);
  return 0;
}
```

```
root@DESKTOP-JT611DE:~# ls -lh /home
total     0       drwxr-xr-x 1 root root 4.0K Apr 16 13:50 picarib
root@DESKTOP-JT611DE:~# ./test
total     0       drwxr-xr-x 1 root root 4.0K Apr 16 13:50 picarib
```

- **`int execlp(const char *file, const char *arg, …, NULL );`**
  - *Description:* execl() does not use the **PATH** environment variable. So, the full path of the executable file is required to run it with execl(). execlp() uses the PATH environment variable. So, if an executable file or command is available in the PATH, then the command or the filename is enough to run it, the full path is not needed.

- Example:

```
#include <unistd.h>

int main(void) {
  char *programName = "ls";
  char *arg1 = "-lh";
  char *arg2 = "/home";

  execlp(programName, programName, arg1, arg2, NULL);
  return 0;
}
```

- **`int execv(const char *path, char *const argv[]);`**
  - *Description:* In execl() function, the parameters of the executable file is passed to the function as different arguments. With execv(), you can pass all the parameters in a NULL terminated array **argv**. The first element of the array should be the path of the executable file. Otherwise, execv() function works just as execl() function.
  - Example:

```
#include <unistd.h>

int main(void) {
  char *binaryPath = "/bin/ls";
  char *args[] = {binaryPath, "-lh", "/home", NULL};
  execv(binaryPath, args);
  return 0;
}
```

- **`int execvp(const char *file, char *const argv[]);`**
  - *Description:* Works the same way as execv() system function. But, the PATH environment variable is used. So, the full path of the executable file is not required just as in execlp().

```
#include <unistd.h>
int main(void) {
  char *programName = "ls";
  char *args[] = {programName, "-lh", "/home", NULL};
  execvp(programName, args);
  return 0;
}
```

- **`int execle(const char *path, const char *arg, …, NULL, char * const envp[] );`**

- *Description:* Works just like execl() but you can provide your own environment variables along with it. The environment variables are passed as an array **envp**. The last element of the **envp** array should be NULL. All the other elements contain the key-value pairs as string.
- Example:

```
#include <unistd.h>

int main(void) {
  char *binaryPath = "/bin/bash";
  char *arg1 = "-c";
  char *arg2 = "echo "Visit $HOSTNAME:$PORT from your browser."";
  char *const env[] = {"HOSTNAME=www.hpcc.hcmut.edu.vn.com", "PORT=80",
NULL};

  execle(binaryPath, binaryPath,arg1, arg2, NULL, env);

  return 0;
}
```

- **`int execve(const char *file, char *const argv[], char *const envp[]);`**
- *Description:* Just like execle() you can provide your own environment variables along with execve(). You can also pass arguments as arrays as you did in execv().
- Example:

```
#include <unistd.h>
int main(void) {
  char *binaryPath = "/bin/bash";
  char *const args[] = {binaryPath, "-c", "echo "Visit $HOSTNAME:$PORT
    from your browser."", NULL};
  char *const env[] = {"HOSTNAME=www.hpcc.hcmut.edu.vn.com", "PORT=80",
NULL};
   execve(binaryPath, args, env);
   return 0;
}
```

1.5. Signal/Signal handling

Signals are the interrupts delivered to a process by the operating system which can terminate a program prematurely. You can generate interrupts by pressing Ctrl+C on a UNIX, LINUX, Mac OS X or Windows system. There are signals which can not be caught by the program but there is a following list of signals which you can catch in your program and can take appropriate actions based on the signal. You can use signals by including the header <csignal> (file signal.h).

Signal	Operation
SIGINT	produces a receipt for an active signal
SIGTERM	sends a termination request to a program
SIGBUS	bus error which indicates access to a invalid address
SIGILL	detect illegal command
SIGALRM	This signal is used by the alarm() function and indicates the expiration of the timer
SIGABRT	abnormally terminate a program
SIGSTOP	This signal is used to stop a process. It cannot be blocked, handled and ignored.
SIGSEGV	Invalid access to storage
SIGFPE	Overflow operations or mathematically incorrect operations like divide by zero

To handle the above signals, we can use the following functions:

- **`int signal () (int signum, void (*func)(int))`**
  - *Description:* The ***signal()*** will call the ***func*** function if the process receives a signal ***signum***. The ***signal()*** returns a pointer to function ***func*** if successful or it returns an error to *errno* (SIG_ERR) or *-1* otherwise.
  - Example:

```
#include<cstdio>
#include<csignal>
#include<unistd.h>
void sig_handler(int signum){
  //Return type of the handler function should be void
  printf("\nInside handler function\n");
}
int main(){
  signal(SIGINT,sig_handler); // Register signal handler
  for(int i=1;;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

```
root@DESKTOP-JT611DE:~# ./example
1 : Inside main function
2 : Inside main function
^C
Inside handler function
3 : Inside main function
```

- **`int raise(int signum)`**
  - *Description:* A process also can explicitly send signals to itself or to another process. **raise()** and **kill()** function can be used for sending signals. Both functions are declared in <**signal.h**> header file. The **raise()** function is used for sending signal *signum* to the calling process (itself). It returns zero if successful and a nonzero value if it fails.
  - Example:

```
#include<cstdio>
#include<csignal>
void sig_handler(int signum){
  printf("Inside handler function\n");
}
int main(){
  signal(SIGUSR1,sig_handler); // Register signal handler
  printf("Inside main function\n");
  raise(SIGUSR1);
  printf("Inside main function\n");
  return 0;
}
```

```
root@DESKTOP-JT611DE:~# ./example
Inside main function
Inside handler function
Inside main function
```

- **`int kill(pid_t pid, int signum)`**
  - *Description:* The kill function used for sending a signal *signum* to a process or process group specified by *pid*.
  - Example: Re-implement the previous example with ***kill()*** (replace raise() function with kill() function)

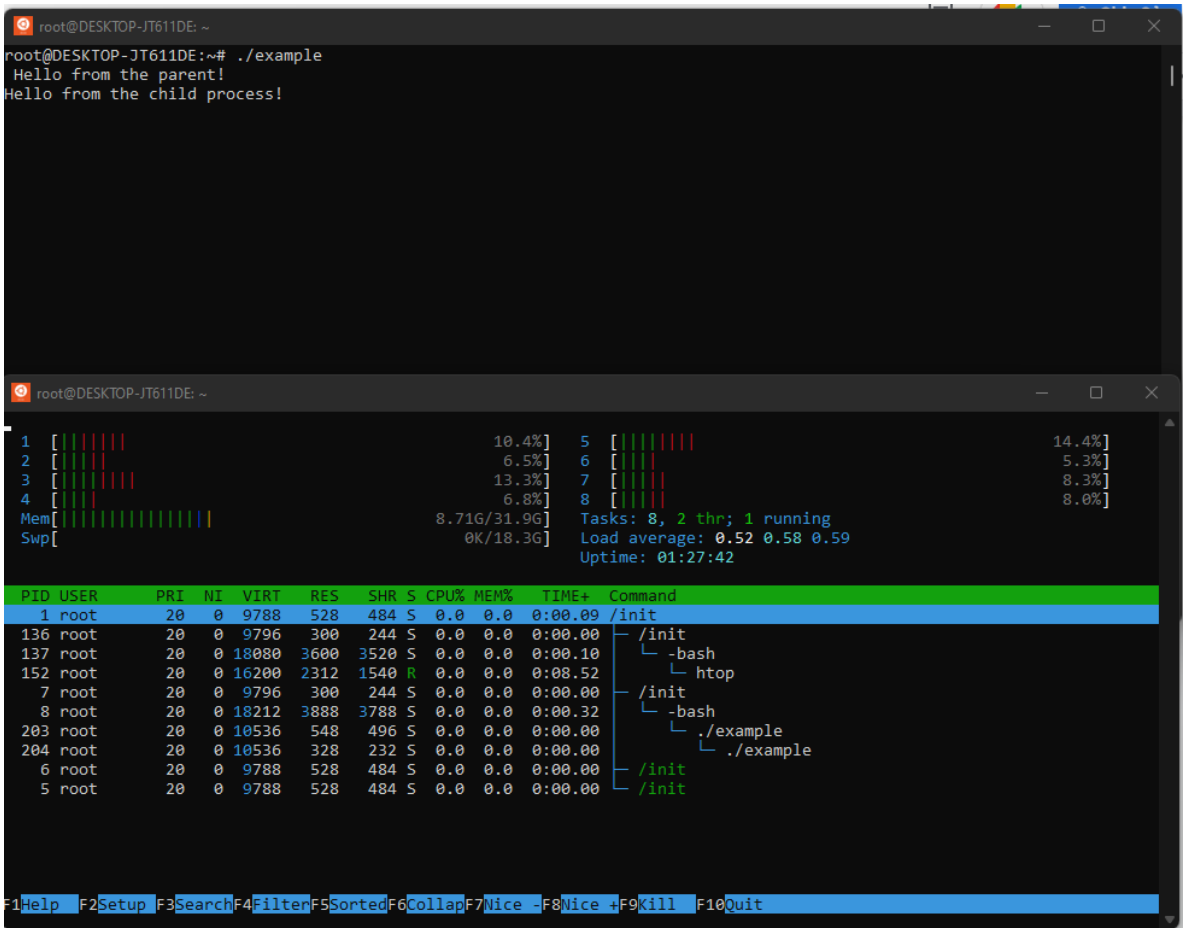
```
#include<cstdio>
```

```
#include<unistd.h>
#include<csignal>
void sig_handler(int signum){
  printf("Inside handler function\n");
}
int main(){
  pid_t pid;
  signal(SIGUSR1,sig_handler); // Register signal handler
  printf("Inside main function\n");
  pid=getpid();      //Process ID of itself
  kill(pid,SIGUSR1);        // Send SIGUSR1 to itself
  printf("Inside main function\n");
  return 0;
}
```

1.6. Multi-processing

Basically, we can use ***fork()*** function to implement a multi-processing schema in C/C++. For example, you can run the following code in a terminal, and run htop command in another terminal to monitor the relationship between parent and child processes. (htop was described in the Appendix section).

```
#include <cstdio>
#include <unistd.h>
#include <cstdlib>
int main(int argc, char ** argv)
{
    pid_t pid = fork();
    if (pid == 0) { // we are in the child process
        printf("Hello from the child process!\n");
        sleep(60);
    } else if (pid == -1) {
        perror ( " fork " ) ;
    } else {
        // now in the parent process
       printf(" Hello from the parent!\n");
       sleep(600);
    }
    return 0;
}
```



In the above code, the parent process and its childs are independent and do not have any further information about each other. In the case that we need to finish the parent process after the child process finishes, we can use **wait()** or **waitpid()** function. For example,

```
#include <cstdio>
#include <unistd.h>
#include <cstdlib>
#include <sys/wait.h>

int main(int argc, char ** argv)
{
    pid_t pid = fork();
    if (pid == 0) { // we are in the child process
        printf("Hello from the child process!\n");
        exit (0); // terminates the child process
    } else {
        // now in the parent process
        waitpid(pid, NULL, 0); // we can use wait(NULL);
        printf("The child process has exited. Hello from the parent!\n");
    }
```

```
    return 0;
}
```

1.7. Address Space

(Virtual) Address space of a Unix process is divided into segments. The standard segments are code segment, data segment, BSS (block started by symbol), and stack segment (may have heap segment).

- The code segment (a.k.a text segment) consists of the code - the actual executable program. The code of all the functions we write in the program resides in this segment. The addresses of the functions will give us an idea where the code segment is. If we have a function func() and let p be the address of func() (p = &func;). We know that p will point within the code segment.
- The data segment consists of the initialized global variables of a program. The Operating system needs to know what values are used to initialize the global variables. The initialized variables are kept in the data segment. To get the address of the data segment we declare a global variable and then print out its address. This address must be inside the data segment.
- The BSS consists of the uninitialized global variables of a process. To get an address which occurs inside the BSS, we declare an uninitialized global variable, then print its address.
- The stack segment contains the local variables, return addresses, etc.
- A process may also include a heap,which is memory that is dynamically allocated during process run time.

Under Linux, a process can execute in two modes - user mode and kernel mode. A process usually executes in user mode, but can switch to kernel mode by making system calls. When a process makes a system call, the kernel takes control and does the requested service on behalf of the process. The process is said to be running in kernel mode during this time. When a process is running in user mode, it is said to be “in userland” and when it is running in kernel mode it is said to be “in kernel space”. We will first have a look at how the process segments are dealt with in userland and then take a look at the book keeping on process segments done in kernel space.

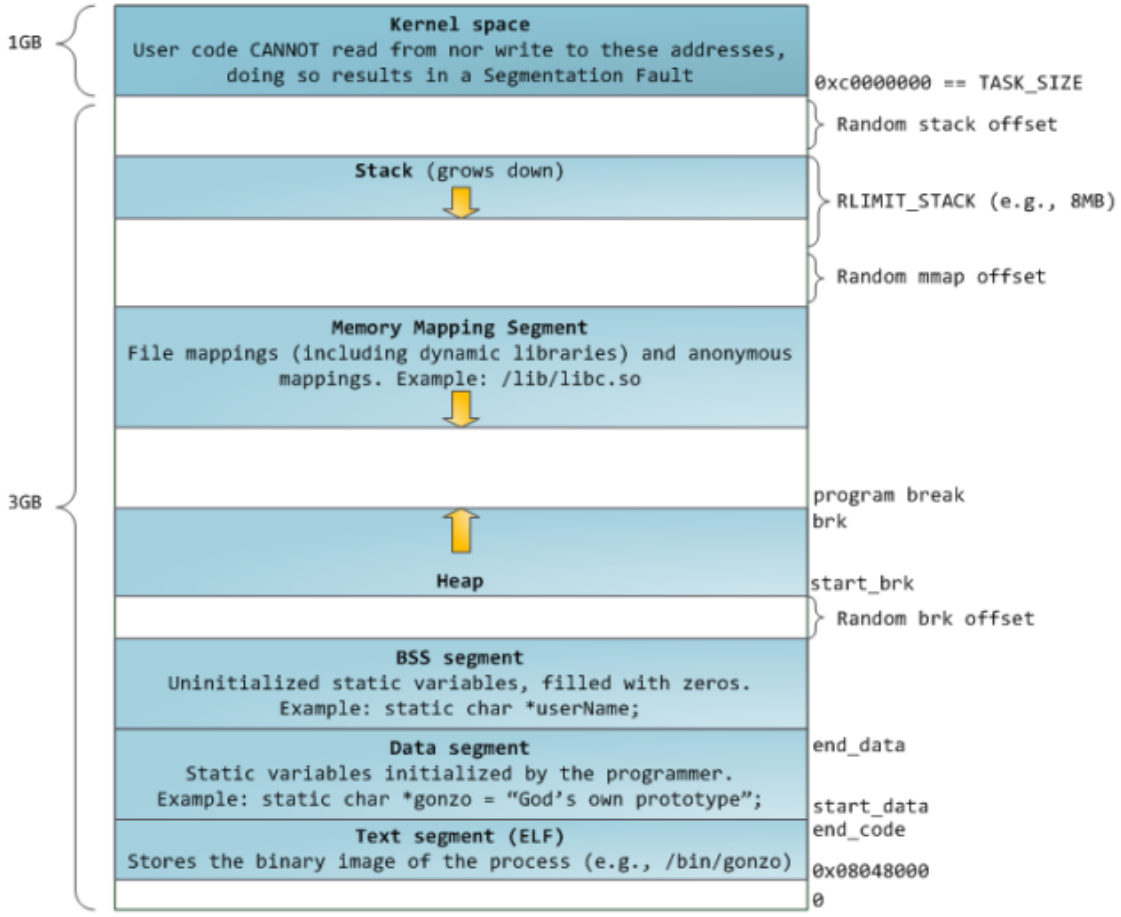
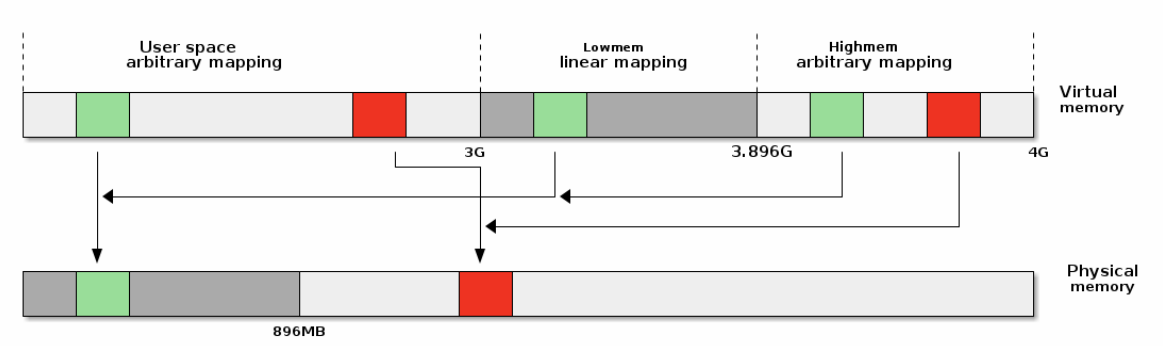


Fig 1.2: Layout of memory segments with process.

In **Figure 1.2** blue regions represent virtual addresses that are mapped to physical memory, whereas white regions are unmapped. The distinct bands in the address space correspond to memory segments like the heap, stack, and so on.



Question: Run the following example many times and give the discussion about the segments of a process. And answer the following questions:

- Which variable belongs to the BSS/Data/Text/Stack segment?
- Which offset do the variables belong to?

```
#include <iostream>
#include <unistd.h>
int glo_init_data = 9;
int glo_noninit_data;
void print_func (){
    int local_data = 9 ;
    printf( "Process ID = %d\n", getpid());
    printf( "Addresses of the process : \n");
    printf( "  1 . glo_init_data = %p\n", &glo_init_data);
    printf( "  2 . glo_noninit_data = %p\n", &glo_noninit_data);
    printf( "  3 . print_func() = %p\n", &print_func);
    printf( "  4 . local_data = %p\n", &local_data);
}
int main( int argc , char **argv ) {
    print_func();
    return 0;
}
```

```
root@DESKTOP-JT611DE:~# ./example
Process ID = 295
Addresses of the process :
1. glo_init_data = 0x7f8c04601010
2. glo_noninit_data = 0x7f8c04601018
3. print_func ( ) = 0x7f8c045fe1c9
4. local_data = 0x7ffff6c7c054
```

Question: if you run the following program multiple times, you will see that sometimes the address of *dynamic_data* (heap segment) is bigger than the address of *local_data* (stack segment). So why is this?

```
#include <iostream>
void print_func (){
    int local_data = 9 ;
    printf( "  1 . local_data = %p\n", &local_data); // in stack
}
int main( int argc , char **argv ) {
    int *dynamic_data = new int;
    print_func();
    printf( "  2 . dynamic_data = %p\n", &dynamic_data); // in heap
    return 0;
}
```

(Short answer: Conceptually, the heap address should be lower than the stack address, however, in modern systems, it will become more complicated. Each operating system will have its own mechanisms to allocate for dynamic variables and therefore the above memory layout for heap address may be wrong in some cases.)

2. Exercises:

1. A practice with multi-processing programming:

- Firstly, downloading two text files from the url:

https://drive.google.com/file/d/1fgJqOeWbJC4ghMKHkuxfIP6dh2F911-E/view?usp=sharing
These file contains the 100000 ratings of 943 users for 1682 movies in the following format:
userID <tab> movieID <tab> rating <tab> timeStamp
userID <tab> movieID <tab> rating <tab> timeStamp
…

- Secondly, you should write a program that spawns two child processes, and each of them will read a file and compute the average ratings of movies in the file and write the answer into a result file. So you will have two result files for two input text files.
- Finally, the parent process will wait until its children finish their tasks and it will read the result files to compute the final results and write them to a text file named **"final_output.txt".**

2. Copy the following program to a file named ***"make-zombie.c"*** and do the following steps:

```
#include <cstdlib>
#include <sys/types.h>
#include <unistd.h>
int main () {
    pid_t child_pid;
    /* Create a child process. */
    child_pid = fork();
    if (child_pid > 0) {
        /* This is the parent process. Sleep for a minute. */
        sleep (120);
    }
    else {
       /* This is the child process. Exit immediately. */
       exit (0);
    }
    return 0;
}
```

- Run the above program, while the program is still running, list the processes on the system by invoking the following command in another window.

```
ps -e -o pid,ppid,stat,cmd
```

- This command lists the process ID, parent process ID, process status, and process command line. Observe that, in addition to the parent make-zombie process, there is another make-zombie process listed. It's the child process; note that its parent process ID is the process ID of the main make-zombie process. The child process is marked as <defunct>, and its status code is Z, for zombie.

```
32125   26660 S+   ./make-zombie
32126   32125 Z+   [make-zombie] <defunct>
```

- Answer the following questions:
  - What happens when the main make-zombie program ends when the parent process exits?
  - A few zombie processes are fine, but what happens if you have so many zombie processes? Although a zombie process only consumes a very small amount of resource, an enormous number of zombie processes can overflow the process pool of the OS and new processes can not be launched. What can you do to avoid creating the zombie process?

3. The relationship between processes could be represented by a tree. When a process uses fork system call to create another process then the new process is a child of this process. This process is the parent of the new process. For example, if process A uses two fork system calls to create two new processes B and C,then we could display their relationship by a tree in Figure 1.4. B is a child process of A. A is the parent of both B and C.

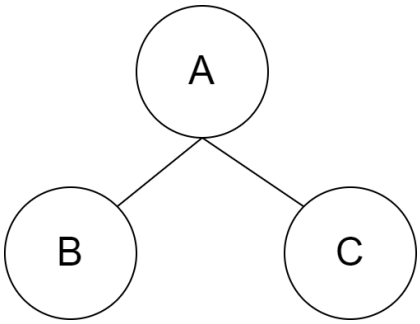


Fig 1.4: Creating processing with fork().

a) Write a program that uses fork system calls to create processes whose relationship is similar to the one shown in Figure 1.4.
(Note: You can use the sleep() function to keep the processes from terminating immediately and use ***htop*** command to check the relationship between these processes.)
Expected outputs:

- Outputs of the program:

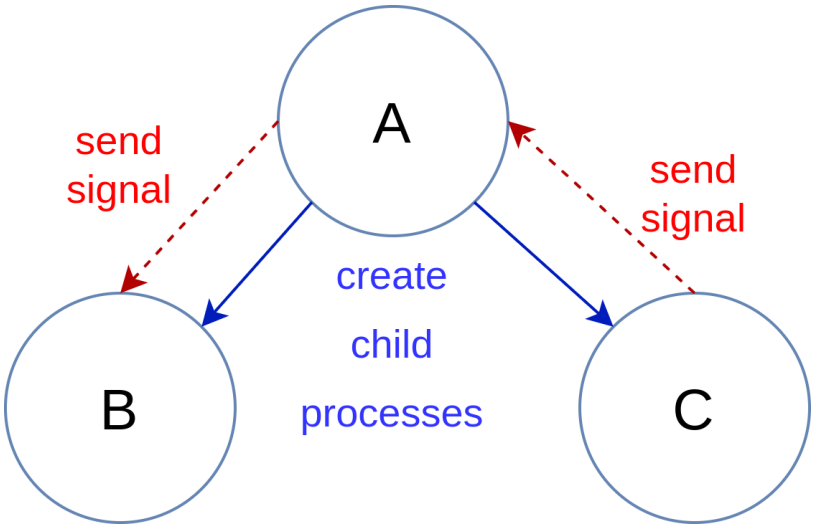
```
This is Process A with PID: 32890
This is Process B with PID: 32891
This is Process C with PID: 32892
```

- htop in another terminal window to monitor the relationship tree.

```
26660 picarib    20   0 23072  6240  4276 S  0.0  0.0  0:00.28          ├─ bash
32890 picarib    20   0  6044  1996  1824 S  0.0  0.0  0:00.00          │  └─ ./ex3
32891 picarib    20   0  6044   196     0 S  0.0  0.0  0:00.00          │     ├─ ./ex3
32892 picarib    20   0  6044   196     0 S  0.0  0.0  0:00.00          │     └─ ./ex3
```

b) After process A creates B and C, it will sleep 10 second and exit (keep process B and C still live with sleep() function). Normally, the parent process will independently terminate and does not care about the child process status. In this exercise, before the process A exits, it will kill all its children.

4. Similar to the above exercise, after process A creates B and C, process A will wait for its children. Process B will sleep 60 seconds and Process C will sleep 10 seconds. After Process C wakes up, it will send a signal to process A to notify that it will terminate. The process A after receiving the signal from C, it will send a signal to process B and exit. The process B after receiving the signal from A, will terminate. The signal propagation schema is as follows:

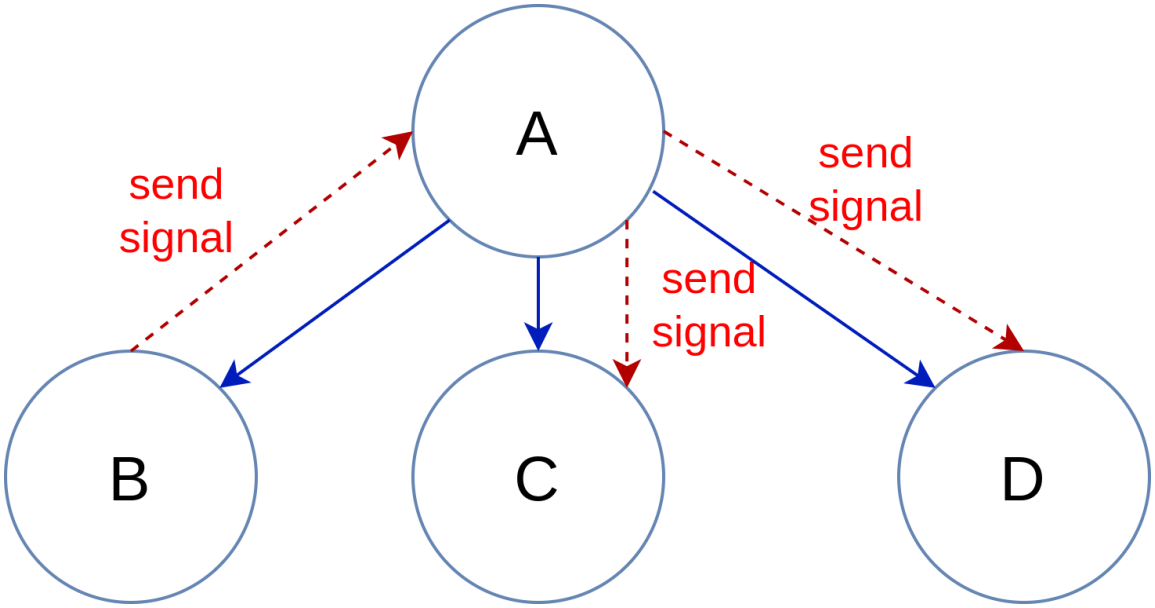


The code sample of the above workflow as follows:

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
#include<stdlib.h>
#include <sys/wait.h>
pid_t pid_C;
pid_t pid_B;
void sig_par(int signo){
    if (signo == SIGUSR1){
        printf("PROCESS A RECEIVE SIGNAL FROM C AND SEND THE TERMINAL
SIGNAL TO B...\n");
        sleep(1);
```

```
        kill(pid_B, SIGUSR1);
        exit(signo);
    }
}
void sig_child(int signo){
    if(signo == SIGUSR1){
        printf("PROCESS B RECEIVE SIGNAL FROM A...\n");
        exit(signo);
    }
}
int main(void){
    pid_t ppid;
    ppid = getpid();
    pid_B = fork();
    if( pid_B== 0){
        signal(SIGUSR1, sig_child);
        sleep(10);
    }
    pid_C = fork();
    if( pid_C == 0){
        printf("PROCESS C SEND TERMINAL SIGNAL TO A...\n");
        sleep(1);
        kill(ppid, SIGUSR1);
        exit(0);
    }
    signal(SIGUSR1, sig_par);
    wait(NULL);
    return 0;
}
```

Now, you should write a program to implement the following signal propagation schema.



5. Write a program that uses fork system calls to create processes whose relationship is similar to the one shown in Figure 1.5.
Note: If a process has multiple children then its children must be created from left to right. For example, process A must create B first then create C and finally D. You can use sleep() function and htop command to monitor the relationship between processes.

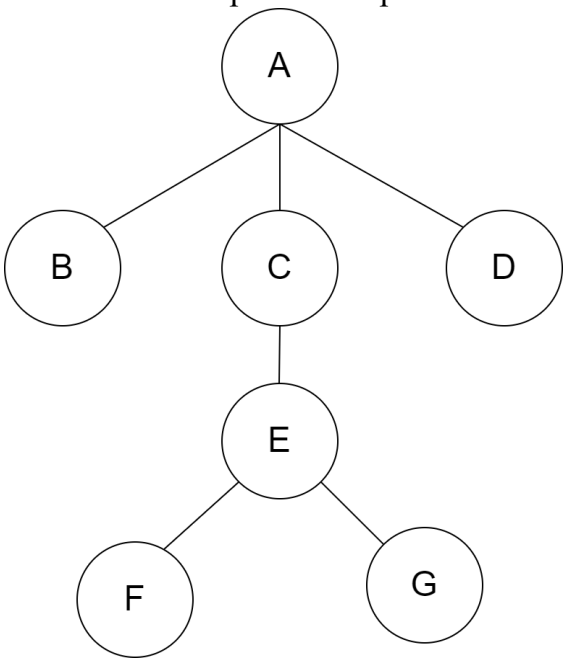


Fig 1.5: Creating processing with fork().

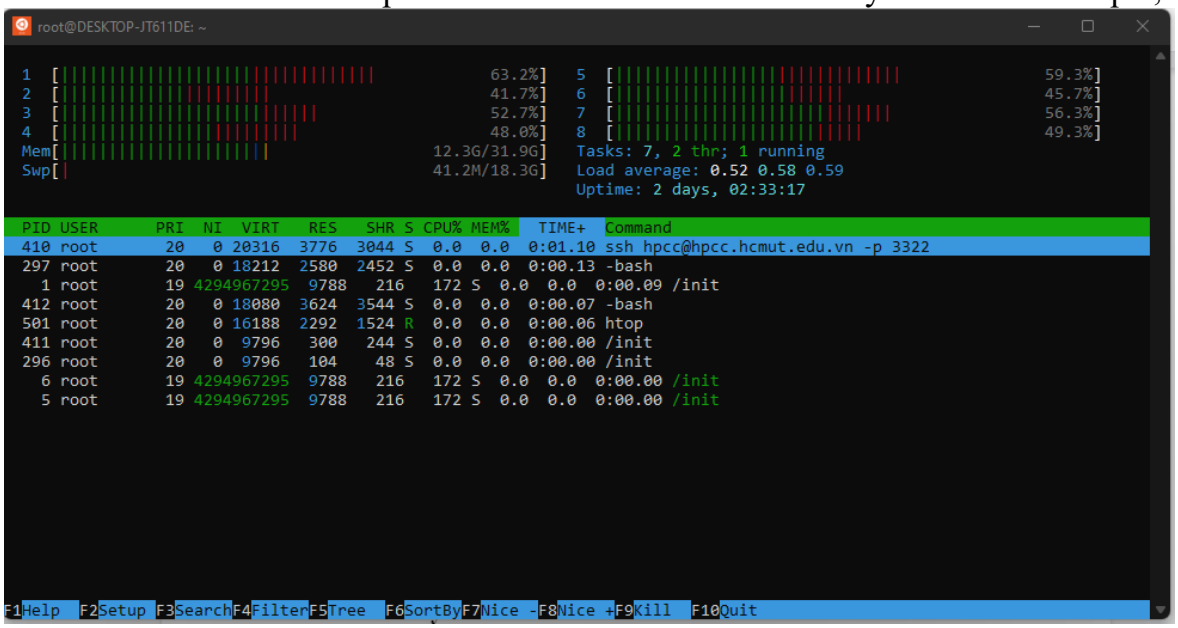
Expected Output:

```
Process B 15409...
Process A 15408...
Process C 15410...
Process D 15411...
Process E 15412...
Process F 15413...
Process G 15414...
```

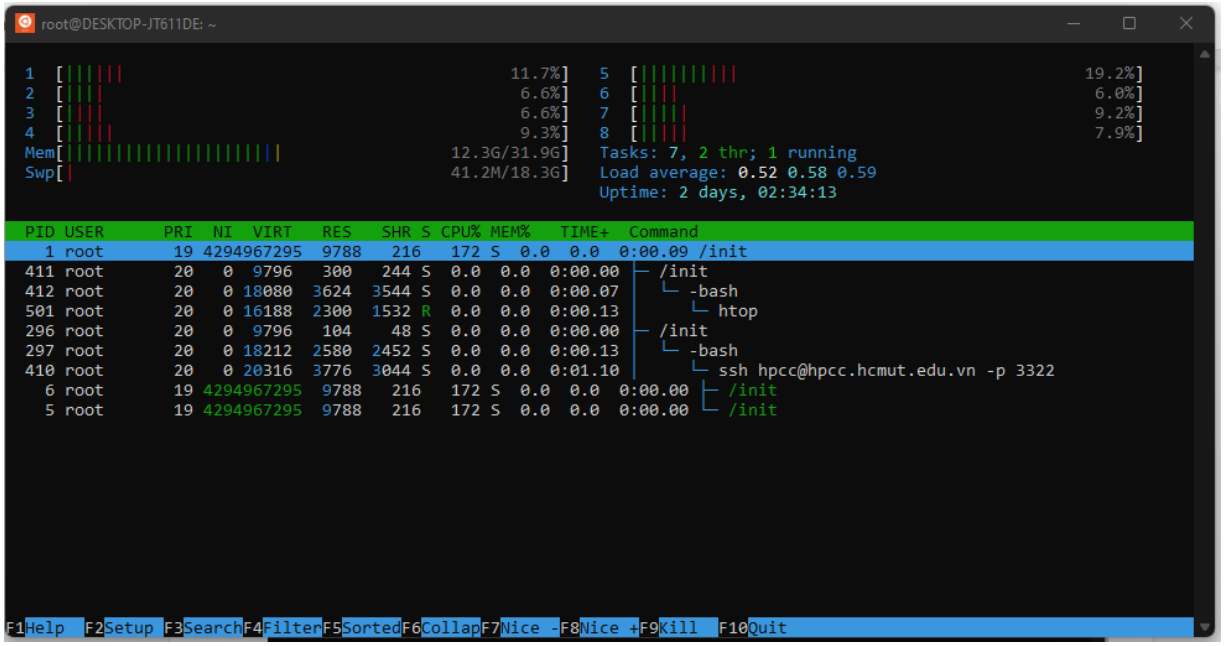
```
15408 picarib  20   0  2772   920   832 S  0.0  0.0  0:00.00          └─ ./ex4
15409 picarib  20   0  2772    88     0 S  0.0  0.0  0:00.00             ├─ ./ex4
15410 picarib  20   0  2772    88     0 S  0.0  0.0  0:00.00             ├─ ./ex4
15412 picarib  20   0  2772    96     0 S  0.0  0.0  0:00.00             │  └─ ./ex4
15413 picarib  20   0  2772    96     0 S  0.0  0.0  0:00.00             │     ├─ ./ex4
15414 picarib  20   0  2772    96     0 S  0.0  0.0  0:00.00             │     └─ ./ex4
15411 picarib  20   0  2772    88     0 S  0.0  0.0  0:00.00             └─ ./ex4
```

3. Appendix: htop command

htop is a common tool to monitor processes and resources of Linux systems. For example,



You can press F5 to sort the process list in a tree format. As you can see in the below example, the process with PID 411 is the father of the process with PID 412,... And the process with PID 1 is the root of the process tree.


root@DESKTOP-JT611DE: ~
1 [|| 11.7%]
2 [| 6.6%]
3 [| 6.6%]
4 [|| 9.3%]
5 [|| 19.2%]
6 [| 6.0%]
7 [|| 9.2%]
8 [|| 7.9%]
Mem[|| 12.3G/31.9G]
Swp[| 41.2M/18.3G]
Tasks: 7, 2 thr; 1 running
Load average: 0.52 0.58 0.59
Uptime: 2 days, 02:34:13
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
1 root 19 4294967295 9788 216 172 S 0.0 0.0 0:00.09 /init
411 root 20 0 9796 300 244 S 0.0 0.0 0:00.00 /init
412 root 20 0 18080 3624 3544 S 0.0 0.0 0:00.07 -bash
501 root 20 0 16188 2300 1532 R 0.0 0.0 0:00.13 htop
296 root 20 0 9796 104 48 S 0.0 0.0 0:00.00 /init
297 root 20 0 18212 2580 2452 S 0.0 0.0 0:00.13 -bash
410 root 20 0 20316 3776 3044 S 0.0 0.0 0:01.10 ssh hpcc@hpcc.hcmut.edu.vn -p 3322
6 root 19 4294967295 9788 216 172 S 0.0 0.0 0:00.00 /init
5 root 19 4294967295 9788 216 172 S 0.0 0.0 0:00.00 /init
F1Help F2Setup F3Search F4Filter F5Sorted F6Collap F7Nice - F8Nice + F9Kill F10Quit