Seminary 3

Processes

Very often processes get interrupted in the middle of what we usually consider a single (atomic) instruction, such as `n++, but this is not so! (Video example).Program works fine by itself… that is not enough anymore!`

<unistd.h>

Functions specific to processes	Functions for IPC
fork()	**pipe(f)**
exit(n)	**mkfifo(name, rights)**
wait(p)	**FILE *popen(c, "r|w")**
exec*(c, lc)	**pclose(FILE *)**
system(c)	**dup2(fo,fn)**

Fork

1. The correct way to create a child process must involve calls to fork, exit and wait. Remember good practice:

```
#include <errno.h>
 - - -
if (function(…) == SUCCES) {
            // do normal stuff }
else   { perror("Error is:"); //or printf("Error is:%s", strerror(errno);
            exit(1);}
```

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <unistd.h>
4    #include <sys/wait.h>
5    int main() {
6        int p, i;
7        p=fork();
8        if (p == -1) {perror("fork imposibil!"); exit(1);}
9        if (p == 0) {
10           for (i = 0; i < 10; i++)
11               printf("Fiu: i=%d pid=%d, ppid=%d\n", i, getpid(), getppid());
12           exit(0);
13       } else {
14           for (i = 0; i < 10; i++)
15               printf("Parinte: i=%d pid=%d ppid=%d\n", i, getpid(), getppid());
16           wait(0);
17       }
18       printf("Terminat; pid=%d ppid=%d\n", getpid(), getppid());
19   }
```

   a. What if we comment line 12, or 16 or both?
   b. How many new processes are created by this code?

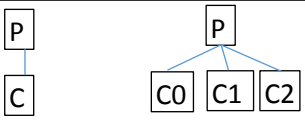
```
fork();     // +1 process, we have 2
if (fork()==0) {      // +1 process for each of the two, so +2, total is 4
   printf("Work\n");  // print Work by the child process created, so 2 times
   exit(0);           // exit 2 child proc, 2 remain; w/out exit, we'd have 4 now & 8 at the end
}
if (fork()==-1) {     // things get complicated, we need a chart or so; 2x2 processes
    perror("fork not working"); //printed only if error
    exit(1);          //  exits only on perror;
}                       // 4 processes if fork successfull
 wait(0); // stuck here to wait for the first child that will finish (any), and then continue
```

2. Problems

a. Draw the process hierarchy of the two examples

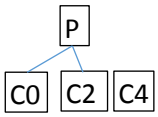
```
pid = fork();
if(pid == 0) {
    // do some stuff
    exit(0);
}
// do some other stuff
wait(0);
```

```
for(i=0; i<3; i++) {
    pid = fork();
    if(pid == 0) {
        // do some stuff
        exit(0);
    }
}
// do some other stuff
for(i=0; i<3; i++) {
    wait(0);
}
```



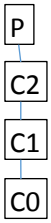
b. Draw the process hierarchy of the code below

```
for(i=0; i<6; i++) {
    if(i % 2 == 0) {
        pid = fork();
        if(pid == 0) {
            exit(0);
        }
    }
}
```



c. Draw the process hierarchy when calling `f(3)`, `f` being the function defined below.

```
void f(int n) {
    if(n > 0) {
        if(fork() == 0) {
            f(n-1);
        }
        wait(0);
    }
}
```



d. How many concurrent processes can we have simultaneously (maximum) :

```
for(i=0; i<3; i++) {
    pid = fork();
    if(pid == 0) {
        // do some stuff
        // no exit(0);
    }
}
// do some other stuff
for(i=0; i<3; i++) {
    wait(0);
}
```

2^3 processes including parent process

e. The same for (excluding parent process):

```
for(i=0; i<3; i++) {
    pid = fork();
    if(pid == 0) {
        // do some stuff
        exit(0);
    }
    wait(0);
}
// do some other stuff
for(i=0; i<3; i++) {
    wait(0);
```

```
while(1) {  //server side handling requests
    req = get_request();
    pid = fork();
    if(pid == 0) {
        res = process_request(req);
        send_response(res);
        exit(0);
    }
    wait(); // all is sequential now :-(
}
```

1 ???

Signal

```
# include <signal.h>

- - -
signal(SIGNAL, NEWHANDLERFUNCTION);
```

A process can assign a certain function to a signal using function `signal`. To generate a signal, we need to use function `kill`.

Example: Let's write a program that refuses to stop when it receives Ctrl-C

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

void f(int sgn) {
    printf("I refuse to stop!\n");
}

int main(int argc, char** argv) {
    signal(SIGINT, f);
    while(1);
    return 0;
}
```

1. Implement a program that upon receiving SIGINT (ctrl-C) asks the user if he/she is sure the program should stop, and if the answer is yes, stops, otherwise it continues.

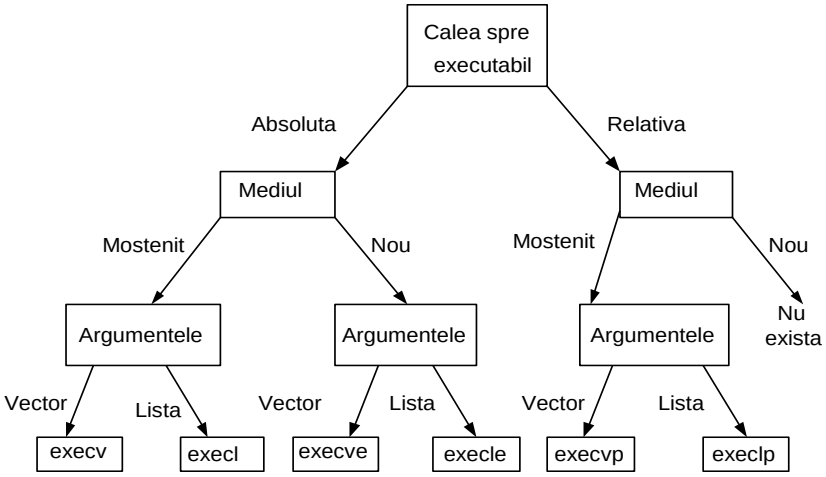
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

void f(int sgn) {
    char s[32];
    printf("Are you sure you want me to stop [y/N]? ");
    scanf("%s", s);
    if(strcmp(s, "y") == 0) {
      exit(0);
    }
}

int main(int argc, char** argv) {
    signal(SIGINT, f);
    while(1);
    return 0;
}
```

2. Solve the server problem with concurrent zombies (will be discussed at the lecture)

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

void f(int sgn) {
    wait(0);
}

int main(int argc, char** argv) {
    signal(SIGCHLD, f);
    while(1) {
        req = get_request();
        pid = fork();
        if(pid == 0) {
            res = process_request(req);
            send_response(res);
            exit(0);
        }
    }
    return 0;
}
```

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

int main(int argc, char** argv) {
    signal(SIGCHLD, SIG_IGN);
    while(1) {
        req = get_request();
        pid = fork();
        if(pid == 0) {
            res = process_request(req);
            send_response(res);
            exit(0);
        }
    }
    return 0;
}
```

Exec functions



Example: Write a program to execute `ls -l`

```
#include <stdio.h>
#include <unistd.h>
int main() {
    char* argv[3];
    argv[0] = "/bin/ls";
    argv[1] = "-l";
    argv[2] = NULL;
    execv("/bin/ls", argv);
    //execl("/bin/ls", "/bin/ls", "-l", NULL);
    // execlp("ls", "ls", "-l", NULL);
    // system("ls -l *.c");
}
```

1. Write a C program that measures the duration of another programs execution given as command line argument along with its own arguments (i.e. `./measure grep "/an1/gr911/" /etc/passwd`)

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

int main(int argc, char** argv) {
    char** a;
    int i;
    struct timeval start, finish;
    float duration;

    if(argc < 2) {
        fprintf(stderr, "No command provided.\n");
        exit(1);
    }

    a = (char**)malloc(argc*sizeof(char*));
    for(i=1; i<argc; i++) {
        a[i-1] = argv[i];
    }
    a[argc-1] = NULL;

    gettimeofday(&start, NULL);
    if(fork() == 0) {
        if(execvp(a[0], a) < 0) {
            fprintf(stderr, "Command execution failed.\n");
            exit(1);
       }
    }
```

```
    wait(0);
    gettimeofday(&finish, NULL);

    duration = (
      (finish.tv_sec - start.tv_sec)*1000.0f +  // convert seconds difference to milliseconds
      (finish.tv_usec - start.tv_usec)/1000.0f  // convert microseconds diff to milliseconds
    )/1000.0f;                                  // convert duration from milliseconds to seconds

    printf("Duration: %f seconds\n", duration);

    return 0;
}
```

In a similar way we can launch any program, for example a c executable named **par** and two arguments:

```
if (fork() == 0) {
            execl("./par", "./par", argv[i], argv[i+1], NULL);
        }
```

Funny code when all goes wrong

1. What does this do in 5 characters:
$0&$0
Crashes system because: $0 is bash , & runs in background

```
$ echo $0
Bash
```

2. Fork bomb, fork bomb, you're my fork bomb, you can crash the system when I though nothing is wrong...:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    while(1) {
      fork();
    }
    return 0;
}
```

3. Shell script short and crazy
Here's an example in which a shell script is told to run two instances of $0 — $0 is a shell variable returning the name of the script itself — and pipe the output of one through the other, which results in exponentially replicating processes.

```
#!/bin/sh
./$0|./$0&
A simpler way is to just run ./$0& twice:

#!/bin/sh
./$0&
./$0&
```

4. Bash BOMB
In Bash, a fork bomb can be performed by declaring and calling a multiple-recursive function:

```
bomb(){
  bomb | bomb &
}
bomb
```

Additionally, one of the most famous and commonly cited examples of a fork bomb is this dense one-line Bash command:

```
:(){ :|:& };:
```

This command is an obfuscated version of the above. The trick here is that : is used as a function name, which is possible because the colon is not a reserved character.