5.1 Processes and current execution

1. A process is a program in execution. Running a program multiple times will result in multiple processes.
2. Processes that run simultaneously are said to be concurrent.
3. A quick check shows that a system usually has significantly more processes than CPUs and cores. So how can all those processes be active at the same time since there are not enough processing units?
   a. The system gives each process a quanta of time on the CPU and switches between processes after each quanta.
   b. These time quants are impercepitbly small, so all processes seem to make progress simultaneously.

4. Problem : When multiple processes run concurrently and access the same shared resources (like a variable, file or memory location), problems can arise if they do not coordinate properly. Such a situation is called race condition.

5.2 Creating processes in unix

1. The UNIX way of creating a process is to duplicate the current process, by making a copy of it using the system call **fork**.
2. Take a look at the code below. It will print "after" twice. That's because after calling **fork**, there will be two processes : the original one (called parent) and the copy of it (called child). Both processes continue from the instruction after **fork**, and thus will both prin "after".

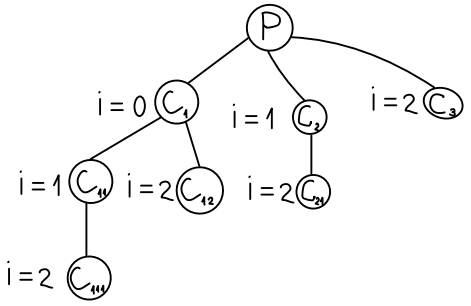
```
#include<stdio.h>
#include<unistd.h>

int main(int argc, char** argv) {
    printf("before\n");
    fork();
    printf("after\n");
    return 0;
}
```

3. Calling a fork in a loop can be dangerous if done wrongly, because the number of processes will grow exponentially. The code below will create 7 child processes :

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

int main(int argc, char** argv) {
    int i;
    for(i=0; i<3; i++) {
        fork();
    }
    return 0;
}
```



5.3 Distinguishing between the parent and the child process

1. There is a way to distinguish between the parent and child processes, by the return value of **fork**: it returns 0 in the child, and the PID of the child in the parent. Consider the code below, ignoring the wait system call which will be explained later. PID is the current processes ID and PPID is the parent's ID.

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

int main(int argc, char** argv) {
    int r;
    r = fork();
    printf("PID=%d PPID=%d R=%d\n", getpid(), getppid(), r);
    wait(0);
    return 0;
}
```

```
PID=1612 PPID=1436 R=1613
PID=1613 PPID=1612 R=0
```

2. This allows us to execute different code in the child, than in the parent. The exit call in the child ensures that the child is not going to continue with the parent code after finishing the if-statement. The parent never enters the if statement.

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

int main(int argc, char** argv) {
    int pid;
    pid = fork();
    if(pid == 0) {
        printf("Child-only code\n");
        exit(0);
    }
    printf("Parent-only code\n");
    wait(0);
    return 0;
}
```

3. This allows us to write concurrent programs, like a server that serves multiple requests simultaneously. A schematic sequential server code would look like the code below. If Google were implemented like this, any search you make would have to wait until all the other searches done by other people in the world before you, are completed. Definitely not a good user experience.

```
while(1) {
    req = get_request();
    res = process_request(req);
    send_response(res);
}
```

4. We can now write this code as follows. Every request is served by a separate process, and the main server (the parent process) can get a new request immediately and spawn another child process to serve it concurrently.

```
while(1) {
    req = get_request();
    pid = fork();
    if(pid == 0) {
        res = process_request(req);
        send_response(res);
        exit(0);
    }
}
```

5.4 Zombie processes

1. It is natural for a parent process to be interested in the execution status (exit code) of a child process it created. To get this state, the parent process uses either **wait** or **waitpid** system calls.
2. What if the child process ends before the parent calls wait? The system keeps the child process in a zombie state : it doesn't execute anything, but it appears in the list of processes.
3. A zombie process stops when the parent calls **wait** or **waitpid**.
4. Zombie processes are problematic because if they are not "waited" they will grow in number, ocuppying PIDs and bringing the system to a point where no other processes can be created because of lack of PIDs.
5. Let's go zombie watching. Run the program on the left in one console and the command on the right in another console.

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

int main(int argc, char** argv) {
    int pid;
    pid = fork();
    if(pid == 0) {
        sleep(10); exit(0);
    }
    sleep(15); wait(0); sleep(5);
    return 0;
}
```

```
while true; do clear; ps -f -u rares; sleep 1; done
```

6. If you replace "rares" with the username you are using to run the program, you will se for the first 5 seconds both parent and child running. For the next 5 seconds, the child will still be there, although it has finished. That is the zombie process. After the parent calls **wait**,the zombie will disappear, ad the parent process will run for 5 more seconds.
7. To avoid zombie processes, always call wait for each child process you create.
8. The system call **wait** waits for any child process to end. If you want to wait for a specific one, use **waitpid**.
9. The argument for **wait** is a pointer to **int**, where it will return the child process exit code. But since we do not care for that value here, we just pass a zero (NULL).

Examples:

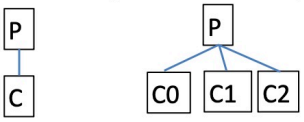
1. The correct way to create a child process ust involve calls to fork, exit and wait.
2. To create just one child process, you need to write something like the code on the left. To create multiple processes, you need to write something like the code on the right.

```
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);
}
```

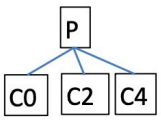
3. Problems

a. Draw the process hierarchy of the two examples above



b. Draw the process hierarchy of the code below

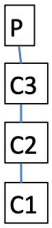
```
for(i=0; i<6; i++) {
    if(i % 2 == 0) {
        pid = fork();
        if(pid == 0) {
            exit(0);
        }
    }
}
```



c.

d. 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);
    }
    exit(0);
}
```



In acest caz, si copiii creeaza alti copii pentru ca :

- Apelul recursiv f(n-1) este facut doar in procesul copil (in ramura if(fork() == 0))
- Asadar fiecare copil devine parinte la randul lui, daca n > 0

e.

1. Write a program that creates the hierarchy of processes in the diagram below.

```
int main(int argc, char** argv) {
    int i, j;

    for(i=0; i<3; i++) {
        if(fork() == 0) {
            printf("%d %d\n", getppid(), getpid());
            for(j=0; j<3; j++) {
                if(fork() == 0) {
                    printf("%d %d\n", getppid(), getpid());
                    exit(0);
                }
            }
            for(j=0; j<3; j++) {
                wait(0);
            }
            exit(0);
        }
    }
    for(i=0; i<3; i++) {
        wait(0);
    }
    return 0;
}
```