Laborator 4 – Procese

Ce este un proces ?

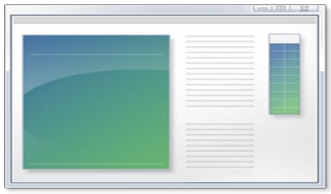
Un proces este o instanta a unui program. Acesta este modul prin care sistemul de operare abstractizea executia unui program. Dintr-un program se pot creea mai multe procese care sunt insa independente logic.

Diferenta dintre un program si un proces

Un **program** este o entitate pasiva ce descrie modul in care ar trebui sa se execute un proces. Acesta se afla salvat pe disc sub forma unui fisier. Astfel de fisiere sunt referite ca find fisiere **executabile.** Fisierele executabile exista in mai multe formate. Astfel :

- Formatul folosit in Linux poarta denumirea de **ELF (Executable and Linking Format)**
- Formatul folosit in Windows poarta denumirea de **PE (Portable Executable)**

ELF



PE

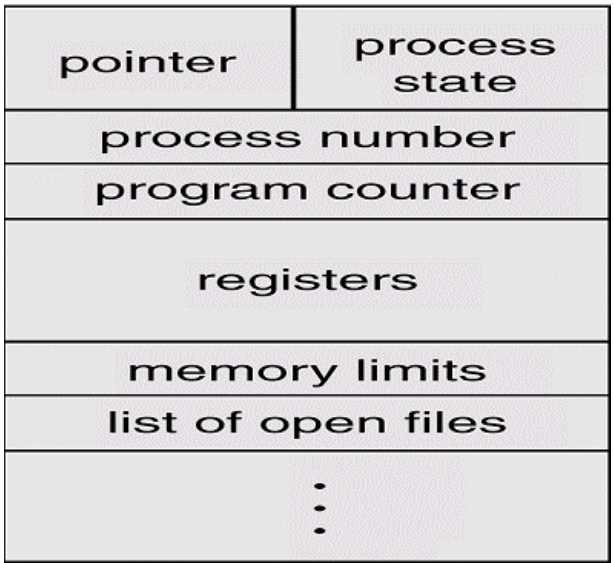
ATENTIE ! Fisierele executabile sunt, din punct de vedere al sistemului, fisiere de tip **REGULAR**

Un **proces** este o entitate activa ce reprezinta imaginea in memoria principala (RAM) a unui program. Pentru orice cerere a utilizatorul, sistemul de operare lanseaza un proces pentru a o satisface.

De gestiunea proceselor se ocupa **kernel-ul** sistemului de operare. Pentru a putea realiza acest lucru , acesta are nevoie sa tina intern o structura pentru fiecare proces existent numita **PCB (Process Control Block).**

Printre cele mai importante informatii continute de PCB regasim:

- Identificatorul unic al procesului
- Spatiul de adresa
- Valoarea registrilor generali: **PC (Program Counter) SP (Stack Pointer)**
- Tabela de descriptori deschisi
- Directorul de lucru
- Identificatorul propietarului
- Starea procesului



PID-ul (Process Identification) reprezinta identificatorul unic al unui proces din sistem, aceasta este cea mai importanta caracteristica a unui proces si este folosit in majoritatea API-urilor POSIX de lucru cu procese.

In Linux procesele sunt organizate sub forma ierarhica pe principiul tata – fiu si sunt abstractizate in sistemul virtual de fisiere din "**/proc"** unde pentru fiecare pid exista un director aferent. Primul proces lansat de **kernel** dupa ce a fost incarcat in memorie de catre **bootloader** este **init.** Acesta are pid-ul 1 si este lansat dupa imaginea din **/sbin/init.** Procesul init este tatal majoritatea proceselor de sistem, ce asigura buna functionare a calculatorului. Un proces poate avea oricati fi dar doar un singur tata ! Pentru a afisa ierarhia de procese ce sunt active un moment dat in sistem se foloseste comanda:

$ pstree

O alta caracteristica importanta a unui proces este id-ul propietarului. Acesta poate sa fie diferit de proprietarul fisierului executabil din care s-a creat procesul. Id-ul este al utilizatorului ce a lansat procesul (acest lucru implica bineinteles sa aiba drepturi de executie asupra fisierului). De exemplu, majoritatea fisierelor de configurare a sistemului din directorul **/etc** pot fi modificate doare de procese-editor-text ce il au drept propietar pe utilizatorul privilegiat **root.**

Pentru a afla informatii despre procesele ce ruleasa in sistem se foloseste comanda:

$ ps

Lansat fara nici un argument, aceasta va afisa procesele ce au fost lansate de la terminalul curent. Pentru a afisa informatii despre toate procesele din sistem se utilizeaza cu argumentele:

$ ps -e

Comanda are o multitudine de alte optiuni ce pot sa fie consultate in pagina de manual (**man ps**)

Printre ele mai precizam doar

$ ps -p pid1,pid2,.....,pidn

Ce afiseaza informatii despre procesele cu pid-urile date ca argument in lista separata prin virgula

$ ps -C cmd1,cm2,...,cmdn

Ce afiseaza informatii despre procesele ce au avut ca imagine fisierele executabile date ca argument in lista separata prin virgula.

$ ps -u user1,user2,...usern

Ce afiseaza informatii despre procesele ce au fost lansate de utilizatorii cu numele date ca argument in lista separata prin virgula.

Comanda:

$ top

Afiseaza in mod dinamic (in timp real) informatii complete despre procesele ce ruleaza in sistem.

Un proces, pe toata durata executiei sale se poate afla in mai mult stari. Starea sa curenta este tinuta in **PCB.** Starile in care se poate afla un proces in Linux sunt :

1. **NEW –** procesul doar ce a fost lansat
2. **READY –** procesului i-au fost alocate toate resursele necesare de catre sistemul de operare si asteapta timp procesor
3. **RUNNING –** instructiunile procesului sunt rulate pe procesor
4. **SLEEPING** – procesul este adormit asteptand un eveniment extern pentru a fi trezit
5. **SUSPENDED –** procesul este suspendat si asteapta primirea unui semnal din partea sistemul de operare pentru a continua executia.
6. **ZOMBIE –** procesul este terminat si asteapta ca parintele sa ia act de acest lucru.

Un proces poate sa fie in una din starile 2-5 de mai multe ori pe perioada executiei.

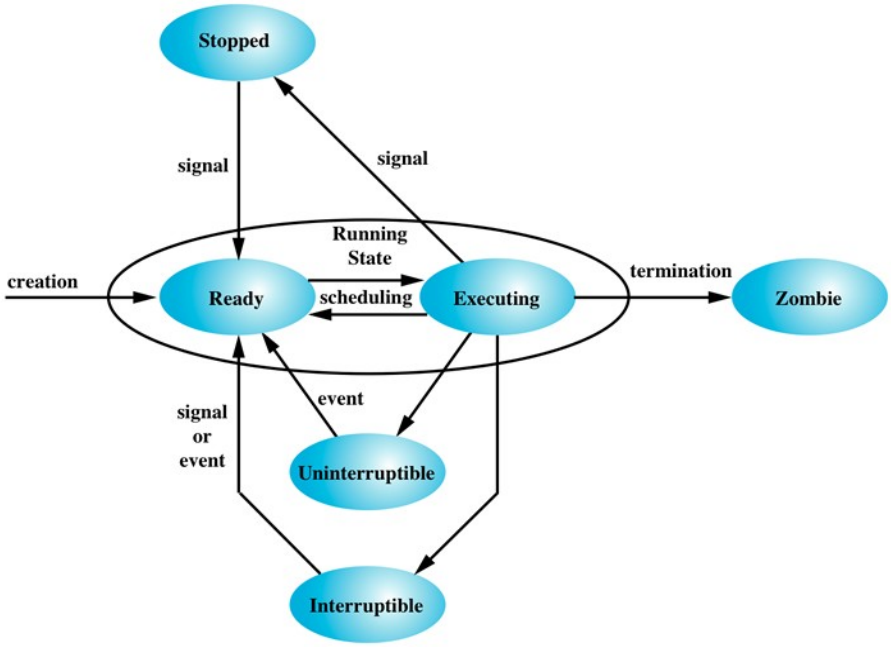


Figure 4.18 Linux Process/Thread Model

In timp ce este in starea **RUNNING** un proces poate executa instructiuni utilizator si atunci spunem ca este in **user space** sau instructiuni sistem privilegiate si atunci spunem ca este in **kernel space.**

Rularea proceselor in background

Un proces poate sa fie rulat in doua moduri

- **foreground** – atunci cand acapareaza terminalul curent, poate sa citeasca input utilizator. Prompt-ul de la shell ne revine doar atunci cand procesul s-a terminat
- **background –** prompt-ul de la shell ne revine imediat, procesul isi continua executia insa nu mai poate primi input de la tastatura.

Pentru a lansa un proces in background se pune “&” dupa comanda astfel:

$ nume_comanda &

Pentru a pune un proces care ruleaza in foreground in background se utilizeaza combinatia de taste (CTRL + Z). Acesta insa trece procesul in starea **SUSPENDED** si este nevoie sa il repunem in functiune manual prin comenzile:

$ fg

Ce reintoarce procesul in starea **RUNNING** in foreground sau:

$ bg

Ce reintoarce procesul in starea **RUNNING** dar in background.

In cazul in care sunt mai multe procese aflate in background ce sunt in starea **SUSPENDED,** se introduce comanda:

$ jobs

Ce va returna procesele aflate in background la terminalul curent alaturi de un numar de identificare (NU **PID-ul**). Pentru a trece un proces specific in starea **RUNNING** se introduce comanda:

$ fg numar_identificare

sau

$ bg numar_identificare

Memorie Virtuala

Sistemele de operare moderne ofera o facilitate foarte importanta ce se numeste **multitasking.** Asta inseamna ca sistemul poate tine in memoria principala mai multe programe ce pot sa fie executate pe procesor intr-un mod echitabil prin cea ce se numeste **context switching.** Aceasta abordare aduce insa o problema !

Spatiul de adresare a doua procese poate sa se suprapuna si sa cauzeze o alterare a integritatii datelor. Astfel, s-a introdus un mecanism ce se numeste **memorie virtuala.** Prin acest mecanism spatiul de adresare a unui proces este virtualizat, adresele sale sunt defapt adrese virtuale ce sunt mapate la adrese fizice de catre o componenta din procesor ce poarte numele **MMU (Memory Management Unit).**

Un proces are impresia ca detine intreg spatiul din memoria principala (RAM) deoarece spatiul fizic din RAM este suplimentat de un spatiu pe disc ce poarta denumirea de **zona de swap**. Pentru o descriere detaliata a mecanismului de memorie virtuala accesati link-ul :

https://en.wikipedia.org/wiki/Virtual_memory

Memoria virtuala a unui proces se imparte in mai mult segmente. Acestea sunt:

1. **Zona de text** – contine instructiunile cod masina a procesului
2. **Zona de date initializate** – contine variabilele globale si statice initializate ce sunt citite din executabil la momentul incarcarii programului
3. **Zona de date neinitializate** – contine variabile globale si statice neinitializate, acestea sunt

definite in executabil doar la nivel de simbol si sunt initializate cu valori aleatoare la run-time

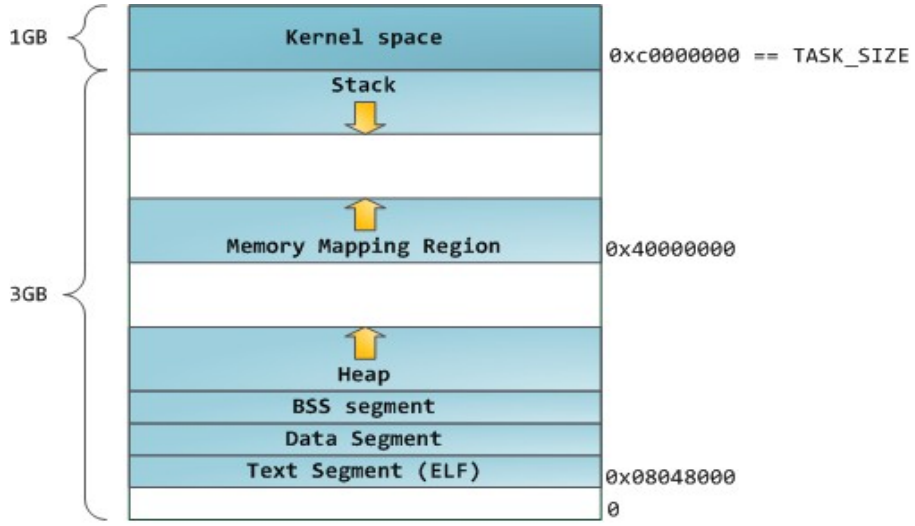
4. **Stiva –** reprezinta o zona de memorie a carei dimensiuni este definita dinamic la runtime, aici sunt retinute variabilele locale si argumentele functilor, pentru fiecare cadru-apel al unei functii se creeaza un asa numit **stack frame.**
5. **Heap-ul –** reprezinta o zona de memorie unde se pot aloca variabile dinamic. Varful zonei de heap poarte denumirea de **breakpoint** sau **program break**

Heap-ul si stiva cresc una catre cealalta, heap-ul creste de jos in sus iar stiva de sus in jos. La adresele cele mai mari se afla stocate argumentele in **linia de comanda** si **variabilele de mediu.** O parte din memoria unui proces este rezervata pentru kernel space.

In codul de mai jos avem fiecare variabila din C in ce zona va fi translatata.

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

char globBuf[65536]; /* Zona de date neinitializata */
int primes[] = {2,3,5,7}; /* Zona de date initializata */

static int square ( int x ) /* Se aloca in frame-ul pentru functia square() */
{
  int result;

  result = x * x;
  return result;
}

static void doCalc( int val ) /* Se aloca in frame-ul pentru functia doCalc() */
{
  printf("The square of %d is %d\n",val, square(val));

  if ( val < 1000 )
  {
     int t; /* Se aloca in frame-ul pentru functia doCalc() */

     t - val*val*val;
     printf("The cube of %d is %d\n",val,t);
  }
}

int main(int argc,char* argv[])
{
   static int key = 9973; /* Zona de date initializata */
   static char mbuf[10240000]; /* Zona de date neinitializata */
   char* p;  /* Se aloca in frame-ul pentru main() */

   p = malloc(1024); /* Indica catre o zona de memorie din HEAP !!! */

   doCalc(key);
```

```
    exit(EXIT_SUCCESS);
}
```



Terminarea unui proces

Un proces se poate termina in doua moduri:

- **anormal –** atunci cand executa instructiuni ilegale ce duce la uciderea sa de catre sistem (ex : efectueaza o impartire la 0, dereferentiaza un pointer NULL etc)
- **normal** – atunci cand executa apelul sistem **_exit().**

Orice proces are un cod de retur din care se pot trage concluzii referitoate la modul sau de terminare In mod obisnuit, valoare de retur 0 indica faptul ca procesul s-a terminat corect iar orice alta valoare indica faptul ca procesul s-a terminat cu esec. Este insa mai bine sa folosim constantele **EXIT_SUCCESS** sau **EXIT_FAILURE** pentru a avea o implementare independenta de platforma. Pentru a examina codul de retur intors de ultima comanda lansata din terminal se lanseaza comanda:

$ echo $?

ATENTIE ! O a doua lansare consecutiva a acestei comenzi va intoarce intotdeauna 0 deoarece “echo” devine imaginea pentru ultimul proces lansat.

Dintr-un program C se poate iesi apeland direct apelul sistem **_exit()** declarat in header-ul **unistd.h** dar este mai bine sa folosim functia de biblioteca **exit()** declarata in header-ul **stdlib.h** ce foloseste intern acest apel sistem. Dintr-un proces se mai poate iesi atunci cand se executa o instruciune de **return** in interiorul functiei **main().**

Codul de retur al procesului va fi cel transmis ca argument functiei **exit()** sau instructiunii de **return.** In lipsa oricareia din aceste metode, procesul va intoarce o valoarea aleatoare cu exceptia situatie cand se foloseste flag-ul **-std=c99** la compilare, caz in care procesul va intoarce 0 intotdeauna.

Se pot instala si handle-uri ce vor fi lansate atunci cand procesul isi termina executia normal prin metodele enumerate mai sus.

Un handle nou poate sa fie instalat prin apelul functiei:

int atexit(void (* func)(void));

definita in header-ul **stdlib.h,** ce primeste ca argument un pointer catre functia ce o dorim. Va intoarce 0 in caz de succes si o valoare diferita de 0 in caz de esec.

Exercitii

1. Scrieti un program demonstrativ ce nu iese din **main()** prin niciuna din metodele enumerate mai sus. Compilati fisierul sursa de doua ori, o data normal si o data cu flag-ul **-std=c99** si observati codul de retur in cele doua situatii.

2. Scrieti un program demonstrativ ce citeste un numar intreg de la tastatura si face **exit()** cu succes in caz ca numarul este mai mare ca 0 si **exit()** cu esec in cazul in care numarul nu este mai mare ca 0

3. Adaugati la programul de la exercitiul precedent doua handle-uri ce se vor executa la iesirea din **main().** Unul din ele va afisa mesajul “Procesul s-a terminat !” iar celalalt va afisa valoarea curenta a variabilei globale **errno.**

Dupa cum am mentionat mai sus, o caracteristica importanta a fiecarui proces este **PID-ul**, un identificator intreg unic. Acesta poate sa fie obtinut prin apelul la functia:

pid_t getpid()

ce intoarce **PID-ul** procesului apelant. Pentru a obtine pid-ul procesului parinte se face apel la functia:

pid_t getppid()

cele doua functii se termina mereu cu succes si intorc un pid corect. Ele sunt declarate in header-ul **unistd.h**.

!!! pid_t este un **typedef** dupa tipul de date **int**. Astfel, il putem gasi in header-ul **unistd.h** ca find declarat ca : **typedef int pid_t.** Se folosesc mult **typedef-urile** in API-uri deoarece dau o claritate mai mare codului **!!!**

Exercitiu

Scrieti un program care afiseaza pid-ul procesului curent si pid-ul procesului parinte.

Crearea de noi procese

Pentru a crea un nou proces in UNIX se foloseste apelul sistem :

pid_t fork()

declarat in header-ul **unistd.h**. Acesta realizeaza un nou proces, copie fidela a procesului ce a

realizat apelul. Sistemul ii aloca un nou **PID** si o parte din datele si resursele procesului apelant. Apelul returneaza pid-ul fiului in contextul procesului tata si 0 in contextul procesului fiu. Daca apelul esueaza atunci va intoarce -1.

Inlocuirea imaginii unui proces

Familia de functii **exec** va executa un nou program, inlocuind imaginea procesului curent, cu cea dintr-un fisier (executabil).

int execl(const char *path, const char *arg, ...);

int execv(const char *path, char *const argv[]);

int execlp(const char *file, const char *arg, ...);

Exemplu de utilizare a acestor functii:

execl("/bin/ls", "ls", "-la", NULL);

char *const argvec[] = {"ls", "-la", NULL};
execv("/bin/ls", argvec);

execlp("ls", "ls", "-la", NULL);

Primul argument este numele programului. Ultimul argument al listei de parametri trebuie sa fie NULL, indiferent daca lista este sub forma de vector sau sub formă de argumente variabile.

Exercitii

1. Folositi apelul sistem **fork()** pentru a duplica procesul curent apoi faceti un apel la functia la **sleep(int seconds)** ce pune in starea **SLEEPING** procesul apelant pentru **seconds** secunde. Lansati procesul in backgroun dupa care dati comanda **ps.** Ce observati ?

2. Scrieti un program ce va face un apel la **fork()**. Procesul tata va afisa pid-ul sau si pid-ul fiului pe care tocmai l-a creat iar procesul fiu va afisa pid-ul sau si pid-ul tatalui. Ce observati ?

3. Scrieti un program ce va prelucra un vector de numere prin intermediul a doua procese. Procesul tata va afisa suma numerelor din prima jumatate, iar procesul fiu va afisa suma numerelor din a doua jumatate.

4. Scrieti un program care deschide pentru scriere un fisier reprezentat printr-un descriptor apoi face **fork().** Procesul tata va scrie in fisier sirul de caractere “abcdefghij” iar procesul fiu va scrie (prin acelasi descriptor) sirul de caractere “klmnoprtsqwxyz”. Dati **cat** pe continutul fisierul obtinut. Ce observati ?

5. Scrieti un program care va lansa comanda **firefox** cu argumentul **http://fmi.unibuc.ro/ro/**

6. Scrieti un program care va face **fork()** iar procesul fiu va lansa comanda **gnome-screensaver-command** cu optiunea **-l.**

Procese zombie si sincronizare tata-fiu.

De cele mai multe ori procesele tata si fiu nu se termina in acelasi timp intre ele existand un comportament asincron. Astfel, distingem urmatoarele scenarii:

- **Procesele se termina in acelasi timp –** lucru rar intalnit
- **Procesul tata se termina inaintea procesului fiu –** caz in care procesul fiu devine **“orfan”** si este adoptat de un proces a carui imagine a fost comanda **/sbin/init.**
- **Procesul fiu se termina inaintea procesului tata –** caz in care procesul fiu trece in starea **ZOMBIE** asteptand ca tatal sa ia act de terminarea sa. Daca tatal se termina fara sa ia act atunci noul tata, **init,** va lua act de terminarea sa si zombie-ul va iesi din tabela de procese a sistemului.

Procesul tata poate executa urmatorul apel pentru a lua la cunostinta de terminarea proceselor fiu:

#include<sys/types.h>

#include<sys/wait.h>

pid_t wait(int* stare)

daca nu exista nici un proces fiu atunci apelul va returna -1 si va seta **errno**. Daca exista procese fiu atunci apelul va bloca procesul curent pana cand unul dintre acestea se va termina si va intoarce pid-ului primului proces returnat si va stoca starea sa in adresa indicata de pointerul transmis ca parametru.

Pentru a astepta dupa un proces cu un anumit pid se va folosi functia:

pid_t waitpid(pid_t pid, int* stare, int optiuni)

Pentru o descriere detaliata a acestor functii si modul in care se poate exploata starea acesati link-urile:

http://linux.die.net/man/2/wait

http://linux.die.net/man/2/waitpid

Precum si cartea “Dezvoltarea aplicatiilor in C/C++ sub sistemul de operare UNIX”, Andrei Baranga, paginile 82-84.

Exercitii

1. Scrieti un program care demonstreaza ca noul tata al unui proces orfan devine un proces lansat dupa comanda **/sbin/init.**

2. Scrieti un program ce lanseaza procese pana cand **fork()** returneaza -1. Dupa care face un **sleep(20).** Ce s-a intamplat ?

3. Scrieti un program care face **fork()** de 5 ori dupa care asteapta dupa terminarea tuturor proceselor fiu si afiseaza pid-urile acestora in ordinea in care s-au terminat.

4. Scrieti un program care face **fork()** dupa care fiul face un apel la **sleep(5).** Tatal va face un apel la wait dar nu va astepta ca procesul sa devina zombie.

5. Scrieti un program care citeste un vector de numere intregi de la tastatura dupa care lanseaza doi fi. Primul fiu va calcula suma numerelor din prima jumatate a vectorului iar al doilea va calcula suma numerelor din a doua jumatate. Tatal va face media aritmetica a celor doua sume.

6. Scrieti un program care verifica daca un numar apare sau nu intr-un vector (numarul si vectorul sunt dati ca argumente in linia de comanda) printr-o strategie de tip divide et impera: se imparte vectorul in doua, apoi se initiaza cate un proces fiu care cauta numarul in fiecare jumatate, in aceeasi maniera; cele doua procese se desfasoara in paralel.

Fiecare subproces returneaza 0 = negasit, 1 = gasit. Fiecare proces nu se termina pana nu i se termina toti fii si in final returneaza suma valorilor returnate de ei. Procesul initial trebuie in plus sa afiseze un mesaj de tip "gasit"/"negasit".

TEME !!!

1. Scrieti un program care numara aparitiile unui sir de caractere ca subcuvant in alt sir de caractere (cele doua siruri sunt date ca argumente in linia de comanda). De fiecare data cand se verifica daca primul sir apare ca subcuvant incepand de pe o pozitie, verificarea se va face de catre un proces fiu (obtinut cu fork) iar procesul tata nu asteapta ca acesta sa se termine pentru a initia o cautare incepand de la o alta pozitie – astfel verificarile au loc in paralel. Fiecare proces fiu returneaza 0 = nu s-a verificat (nu apare ca subcuvant de la pozitia respectiva), 1 = s-a verificat. Dupa initierea tuturor cautarilor, procesul tata asteapta sa i se termine toti fii si aduna codurile lor de retur - acesta valoarea se afisaza (este numarul de aparitii ca subcuvant).

2. Scrieti un program care primeste ca argument in linia de comanda un fisier sursa C si il compileaza cu comanda "**gcc -std=c99 -o executabil**" , asteapta pana cand compilarea se termina, verifica daca s-a terminat cu succes, dupa care lanseaza executabilul obtinut si asteapta ca acesta sa isi termine executia dupa care afiseaza codul sau de retur.

3. Scrieti un program care sa sorteze prin interclasare un sir de caractere in maniera descrisa mai jos

Sortarea prin interclasare presupune impartirea sirului in doua jumatati, sortarea fiecarei parti prin aceeasi metoda (deci recursiv), apoi interclasarea celor doua parti (care sunt acum sortate).

Programul va lucra astfel: se imparte sirul in doua, se genereaza doua procese fiu (fork) care sorteaza cele doua jumatati in paralel, tatal ii asteapta sa se termine (wait sau waitpid), apoi interclaseaza jumatatile.

4. Scrieti un program care primeste ca argument in linia de comanda numele unui director si un nume de fisier. Acesta cauta daca exista fisierul cu acest nume in ierarhia de directoare ce incepe din directorul dat ca argument. Pentru fiecare director care apare in arborescenta se va lansa un nou proces ce cauta in arborescenta din directorul respectiv. La final, se va afisa numarul de aparitii al fisierului cu acest nume.

5. Utilitarul **wget** este folosit pentru a obtine o resursa din internet. Acesta salveaza intr-un fisier denumit defaul **index.html** continutul unei pagini. Scrieti un program care lanseaza **wget** cu argumentul **http://fmi.unibuc.ro/ro/,** asteapta ca procesul sa se termine, dupa care lanseaza doua

procese care sa parseze fisierul index.hmtl astfel obtinut. Cele doua procese vor afisa link-urile din pagina, adica acele stringuri ce incep cu atributul “href=”, primul proces va cauta in prima jumatate a fisierului, iar al doilea in a doua jumatate. Procesul tata va astepta ca cele doua procese sa se termine dupa care va afisa un mesaj de tipul “ S-a terminat ”.

Pentru mai multe detalii despre o pagina html, consultati pagina:

https://en.wikipedia.org/wiki/HTML