**lab 6 procese**

**6.2.1 conceptul de procese**

**proces = entitate ce rep un program ce ruleaza**

🡺 contine 🡪 codul programului

🡪 date, descriptori de fisiere, thread-uri etc.

**PID** = process identifier --> identificator **unic** pentru fiecare proces **🡪 folosit pentru interactionarea cu procesul**

exceptand procesul cu PID=0, **fiecare proces are un parinte** 🡺 este generata implicit o structura de arbore pt procesele din sistem

🡪 daca un proces parinte isi incheie executia in timp ce unii din copiii sai inca ruleaza, copiii devin orfani

🡪 **copiii orfani sunt adoptati in mod automat de procesul cu PID=1**

🡪 daca un proces copil isi incheie executia in timp ce parintele lui inca se executa (**si nu asteapta terminarea copilului respectiv**) procesul respectiv (**copilul**) **e pus in starea zombie** **--> proces terminat dar pentru care inca se pastreaza informatii --> acestea vor fi sterse ( implicit si procesul zombie)** **atunci cand parintele se termina sau cere sistemului de operare info despre copilul terminat**

**6.2.2 crearea de procese si obtinerea identificatorilor**

**#include <sys/types.h>**

**#include <unistd.h>**

**pid\_t fork(void);**

**pid\_t getpid(void);**

**pid\_t getppid(void);**

**6.2.2.1 fork 🡪 fork();**

**fork** = da nastere unui proces nou 🡪 **se creaza un proces copil identic cu procesul parinte** ( **dpdv al codului si resurselor**) ce continua executia imediat dupa apelul de sistem

🡪singura diferenta este valoarea returnata de fork():

🡪 **in procesul copil se va returna 0 iar in procesul parinte se va returna PID-ul copilului**

🡪 **eroare ( nu se poate crea procesul fiu ) : -1;**

**6.2.2.2 getpid & getppid**

**--> utilizate pt obtinerea pid-ului propriu respectiv parinte ; tipul returnat este pid\_t**

**6.2.2.3 exemplu**

**int main(void){**

**pid\_t pid;**

**if(pid==-1){**

**perror(" nu se poate crea un proces fiu");**

**return -1;**

**}**

**else if(pid==0){**

**// cod executat doar de copil**

**printf(" [child] my pid is %d. MY PARENT's pid is %d", getpid(), getppid());**

**}**

**else{**

**//acest cod e executat doar de parinte**

**printf(" [parent] my pid is %d . i created a child with pid %d" ,getpid(),pid);**

**wait(NULL);**

**}**

**return 0;**

**}**

// explicatii

--> pid\_t --> intreg cu semn (int)

--> dupa apelul fork(), se executa in paralel atat procesul parinte cat si procesul copil/

--> daca apelul de sistem = 0 ==> apelul de sistem a reusit --> ne aflam in procesul copil

> 0 ==> ne aflam in procesul parinte;

in procesul copil ==> getpid() = pid propriu

==> getppid() = pid parinte

in procesul parinte ==> getpid() = pid propriu (adica al parintelui)

pid-ul returnat de fork() = pid-ul copilului

// afisare:

**[PARENT] My PID is 14982. I created a child with PID 14983.**

**[CHILD] My PID is 14983. My parent's PID is 14982.**

**6.2.3 terminarea proceselor si asteptarea terminarii copiilor**

**#include <sys/types.h>**

**#include <sys/wait.h>**

**#include <unistd.h>**

**#include <stdlib.h>**

**void exit(inst status);**

**pid\_t wait(int \*wstatus);**

**pid\_t waitpid(pid\_t pid, int \*wstatus, int options);**

**6.2.3.1 exit**

🡪 **inchide procesul curent** 🡪 param = nr ce indica starea programului de iesire (0,255);

* Return: 0=executie cu succes, altfel eroare

**6.2.3.2 wait si waitpid**

scopul unui proces copil: parintele creeaza copii pentru a le delega unele sarcini si de obicei asteapta terminarea lor

**waitpid** --> param: **pid** = pid-ul copilului pentru care se asteapta terminarea ,

**wstatus** = salvare stare cu care procesul copil s-a terminat/NULL daca nu avem nev de stre;

**options** = 0 astept mereu terminarea copilului ( sau alta valoare pentru a indica conditii in care nu se asteapta terminarea copilului)

**wait** 🡪 nu se specifica un copil anume**, ci se asteapta terminarea ORICARUI copil**

**WEXITSTATUS** --> **extragem starea cu care s-a terminat procesul**

**6.2.3.3 exemplu**

**#include <sys/types.h>**

**#include<sys/wait.h>**

**#include <unistd.h>**

**int main(void){**

**pid\_t pid1=-1,pid2=-1,pid3=-1;**

**int x=5,y=3,z=0; //aici PARINTELE initializeaza variabilele --> variabilele suunt mostenite de copii**

**int status1=0,status2=0;**

**// fiecare copil trebuie sa astepte un timp aleator inainte de a efectua calculele --> se foloseste usleep**

**pid1=fork();**

**if(pid1==0){**

**//1st child**

**srand(getpid()); // initializare generator numere random cu ceva unic...pid**

**usleep(random()%500000); // pun procesul sa astepte o perioada aleatoare de timp**

**z=x+y;**

**exit(z); // transmitere rezultat la parinte**

**}**

**pid2=fork();**

**if(pid2==0){**

**//2nd child'**

**srand(getpid());**

**usleep(random()%500000);**

**z=x-y;**

**exit(z);**

**}**

**pid3=fork();**

**if(pid3==0){**

**//3rd child**

**srand(getpid());**

**usleep(random()%5000000);**

**z=x\*y;**

**exit(z);**

**}**

**waitpid(pid2,&status1,0); // procesul parinte asteapta dupa primul parinte --> procesul parinte este blocat pana ce al doilea copil isi termina executia**

**// chiar daca un alt copil isi termina executia mai repede , parintele nu continua pana ce nu primeste rezultatul de la al doilea**

**printf(" subtraction result = %d" ,WEXITSTATUS(status1));**

**wait(&status1);**

**wait(&status2);**

**// primire rezultat de la oricare din copiii 1/3 --> rezultatul poate veni in orice ordine**

**printf("The result of the other two operations are %d and %d.\n", WEXITSTATUS(status1), WEXITSTATUS(status2));**

**}**

// explicatii

--> generare de numere random --> srand(time(NULL)) initializat o singura data la inceputul programului a.i sa genereze nr diferite

-- OBS: nu putem folosi aceasta abordare pentru ca cei 3 copii ar mosteni aceeasi stare a generatorului si ar genera aceleasi timp de asteptare

-- OBS2: cand lucram cu numere random in procesele ce au copii, trebuie ca fiecare proces copil sa isi initializeze generatorul de numere aleatoare si nu putem folosi

time(NULL) ( timp curent in secunde) la initializare pentru ca sunt sance ca cele 3 procese sa apeleze in aceeasi secunda si sa initializeze aceeasi valoare -->

--> e necesara initializarea cu o valoare unica --> de exemplu PID-ul acestuia

**6.2.4. lansarea in executie a programelor**

**#include <unistd.h>**

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

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

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

**int execvp(const char\* file, char \*const argv[]);**

**6.2.4.1 Familia de functii exec**

**exec** **🡺 rolul de a inlocui imaginea procesului curent cu alt program ( fisier executabil ) a carui cale e transmisa ca argument**

**🡺 dupa executia noului program, NU se revine in codul programului apelant.**

* **nu se creeaza un proces nou, ci se inlocuieste cel curent** 🡺 **anumite resurse(fd de exemplu) se mostenesc**

**execl** == primeste un nr variabil de argumente

🡪 **path** = **calea spre executabil**

🡪 **restul sunt argumente pe care executabilul le primeste in linia de comanda (inclusiv numele sau ca arg 0)**

🡪 **ultimul arg trebuie sa fie NULL** -- fct variadice in C (cu nr variabil de arg) nu stiu cate argumente au primit si trebuie sa isi dea seama unde sa se opreasca

🡪 **exemplu**:

**execl("/bin/ls","ls" ,"-l" argv[1],NULL);**

**perror("execl failed"); // nu e nev sa verificam valoarea returnata ; in caz de eroare se executa pur si simplu urmatoarea linie**

🡪 cale spre executabilul ls (/bin/ls); numele comenzii (ls); argumente (-l) ; santinela NULL

**🡪 spre deosebire de alte apeluri de sistem, la execl NU e nevoie sa verificam valoarea retunata**

🡺 in caz de **succes** , **programul primit ca parametru se executa fara sa revina la acestea**

🡺 in caz de **eroare** , **se executa pur si simplu urmatoarea linie din program**

🡪 **obs**: daca in loc de calea completa in path i-am fi scris numele, functia execl nu l-ar fi gasit; pentru asta folosim execlp

**execlp** **== executabilul e cautat in vb de mediu path**

🡪 exemplu:

**execlp("ls","ls","-l",argv[1],NULL);**

**perror(" ... " );**

**execv,execvp** --> similare cu execl si execlp doar ca primesc vectorul cu argumente; **ultimul elem al vectorlui e NULL;**

1 ...

2 char \*arguments[4];

3 arguments[0] = "ls";

4 arguments[1] = "-l";

5 arguments[2] = argv[1];

6 arguments[3] = NULL;

7 execv("/bin/ls", arguments);

8 perror("execv failed");

9 ...

**6.2.4,2 exemplu redirectare**

int main(int argc, char\*\*argv){

pid\_t pid;

int fd,status;

if(argc!=2){

..//eroare

}

pid=fork();

if(pid==-1){

perror("nu se poate crea copilul proces");

return -1;

}

else if(pid==0){

//aici ajunge procesul fiu

fd = open(argv[1],O\_CREAT | O\_TRUNC | O\_WRONLY, 0644); // deschidere fisier DOAR IN procesul copil

// obs: chiar daca argumentul pt fisier a fost primit de parinte, copilul in acest moment e o clona a parintelui, deci va avea aceleasi valori ale argumentelor

if(fd<0){

perror("could not open result file");

exit(1);

}

dup2(fd,1); // redirectare stdout --> acum fd va fi folosit ca STANDARDOUTPUT

execlp("ls","ls","/",NULL);

perror(" execlp failed");

exit(2);

}else{

//aici ajunge procesul parinte //obs--> mesajele in parinte sunt afisate pe ECRAN pt ca redirecarea s-a facut doar in procesul copil

wait(&status); // parintele asteapta terminarea procesului copil

if(WEXITSTATUS(status)==0){

printf(" command completed successfuly!");

}

else{

printf("command failed");

}

}

return 0;

}

// explicatii

--> dup2(fd1,fd2)--> duplica primul descriptor de fisier primit ca prim parametru, utilizant descriptorul din al doilea parametru

-- se foloseste fisierul deschis pe post de iestire standart -- tot ce returneaza ls se va afisa in fisier, nu la iesire

**6.3.1 intrebari recapitulative**

**1. ce returneaza fork ?**

in caz de esec: -1 --> nu se poate crea proces fiu

in caz de succes 0 --> a fost creat procesul copil --> codul va fi executat de copil

pt valori mai mari > 0 --> cod executat doar de parinte

r: un pid -- identificator de proces

**2. cum poate un proces sa isi afle PID-ul propriu? Dar PID-ul p ̆arintelui?**

-- pid propriu :: getpid();

-- pid parinte :: getppid();

**3. Cum se transmite starea de terminare a unui proces copil p ̆arintelui acelui copil?**

prin: exit(stare) din copilul

in parinte este interceptata cu : wait(&status) sau waitp(pid,&status,0);

**4. Ce se ˆıntˆampl ̆a dac ̆a un proces copil ˆıs, i ˆıncheie execut, ia ˆınainte ca p ̆arintele s ̆a apeleze wait?**

**Dar invers?**

-- daca un proces copil isi termina executia inainte ca parintele a apeleze wait , procesul copil devine zombie

== adica el e terminat, dar inca se pastreaza informatiile despre el care vor fi sterge ( implicit si procesul zombie) cand parintele se termina SAU cand so cere informatiile despre copilul terminat

-- daca un parinte se termina inaintea unui copil, procesul copil devine ofran si este adoptat de procesul cu PID-ul 1

**5. De ce execut, ia liniei imediat urm ̆atoare de dup ̆a o funct, ie din familia exec semnific ̆a o eroare?**

-- in caz de succes , programul se executa fara a mai reveni din acesta ;

deci nu putem ajunge pe urmatoarea linie doar in caz de eroare

**6. Care este diferent, a dintre primul s, i al doilea argument al funct, iei execl?**

-- primul argument da path-ul unde se afla executabilul , al doilea argument reprezinta numele comenzii

**7. Dac ̆a dorim s ̆a execut ̆am comanda cat f ̆ar ̆a s ̆a s, tim exact unde se afl ̆a programul respectiv**

**ˆın sistemul de fis, iere, folosim funct, ia execv sau execvp? Justificati;**

folosim execvp pentru ca executabilul este cautat in variabila de mediu PATH si e suficient ca primul argument sa fie "cat"