Sisteme de Operare 1 - Curs 7

Curs tinut in 2012-2013 *de catre* lector dr. Sanda-Maria Dragos

Procese Unix

slide 2

Un concept cheie in orice sistem de operare este procesul.

Proces=un program in executie care foloseste un set de resurse ale sistemului (memorie, procesor, disc, interfata de retea, etc)

Un proces este un program secvential in executie, impreuna cu zona sa de date, stiva si numaratorul de instructiuni (program counter). Trebuie facuta inca de la inceput distinctia dintre proces si program. Un program este, in fond, un sir de instructiuni care trebuie executate de catre calculator.

Procesul include in plus fata de program informatiile de stare legate de executia programului respectiv (stiva, valorile registrilor CPU etc.). De asemenea, este important de subliniat faptul ca un program (ca aplicatie software) poate fi format din mai multe procese care sa ruleze sau nu in paralel.

Structura unui proces

slide 3

Un proces poate rula in mod *utilizator* (mai putin privilegiat; functii utilizator) sau in mod *nucleu* (privilegiat; functii nucleu).

Din punct de vedere structural, in memorie, un proces se reprezinta ca in Fig

slide 4

slide 5

pe langa structurile de date prezentate, mai apar si o serie de tabele intretinute de nucleul sistemului de operare:

Tabela proceselor

tabela cu cate o intrare pentru fiecare proces

Tabela paginilor de memorie virtuala

mentine pentru fiecare pagina de memorie informatii ca: procesul de care apartine pagina respectiva, drepturi, etc

Tabela descriptorilor de fisiere deschise din sistem si Tabela inodurilor

contin cate o intrare pentru fiecare descriptor, respectiv inod, de fisier deschis din sistem.

Imaginea unui proces in memorie este compusa din doua parti: contextul utilizator si contextul nucleu.

Contextul utilizator

Cuprinde:

» **Zona text:** - portiunea de instructiuni masina a procesului

» **Date initializate RO** - constante utilizate de catre proces

» **Date initializate RW** - acele variabile ale procesului care primesc valori initiale in faza de compilare

» **Date neinitializate** - zona ocupata de catre restul variabilelor, cu exceptia celor carora li se aloca spatiu pe stiva sau in zona heap

» **Stiva** - se foloseste pentru transferul parametrilor in cazul apelurilor de functii

» **Heap** - zona unde se aloca spatiu pentru variabilele dinamice

Contextul nucleu

Cuprinde:

Intrarea in tabela proceselor - contine urnatoarele informatii:

» starea procesului
» pointer la zona user si la Tabela regiunilor de memorie ale proceselor
» dimensiunea procesului in memorie
» PID - process ID
» PPID - parent process ID
» UID - user ID - identificatorul userului care a lansat procesul
» EUID - effective user ID - de obicei coincide cu UID cu exceptia cazurilor cand e setat bitul setuid (folosit la stabilirea drepturilor de acces ale procesului la diferite resurse) procesul ia UID-ul proprietarului fisierului executabil

slide 6

» GID - group ID - identificatorul grupului la care apartine utilizatorul
» EGID - effective group ID
» prioritatea procesului (numar intre 1 si 39)
» semnalele trimise procesului
» statistici de timp (ex: folosirea procesorului)
» statusul memoriei procesului (daca imaginea procesului se afla in memoria principala sau in memoria swap)
» pointer la urmatorul proces din coada de procese in starea READY
» descriptori de evenimente care au avut loc cat timp procesul a fost in starea Sleeping

Zona user - contine

» pointer la intrarea in tabela proceselor care corespunde acestei zone user
» UID, EUID, GID, EGID - folosite la determinarea drepturilor de acces ale procesului
» timpul cat acest proces a fost in mod de executie user si cat a fost in mod de executie nucleu
» vectorul de actiunilor de tratarea a semnalelor
» terminalul de control al procesului - cel de la care a fost lansat
» valorile returnate si posibile erori in urma efectuarii unor apeluri sistem
» directorul curent si directorul radacina
» zona variabilelor de mediu
» posibile restrictii impuse de nucleu procesului (ex: dim procesului in mem)

» tabela descriptorilor de fisiere (date despre toate fisierele deschise de proces)
» umask - masca drepturilor de acces pentru fisierele nou create de catre acest proces

slide 7

Tabela regiunilor de memorie ale proceselor - tabela de translatie adrese virtuale - adrese fizice de memorie

Stiva nucleu - memoria alocata pentru stiva folosita de nucleul sistemului de operare

Contextul registru (contextul hardware) - zona de memorie unde se salveaza continutul registrilor

Apelurile sistem de baza pentru gestionare proceselor

Orice proces trebuie creat de catre un alt proces. Procesul creator este numit proces parinte, iar procesul creat proces fiu. Exista o singura exceptie de la aceasta regula, si anume procesul **init**, care este procesul initial, creat la pornirea sistemului de operare si care este responsabil pentru crearea urmatoarelor procese. Interpretorul de comenzi, de exemplu, ruleaza si el in interiorul unui proces.

Creare proceselor Unix. Apelul sistem *fork()*

Din perspectiva programatorului, sistemul de operare UNIX pune la dispozitie un mecanism elegant si simplu pentru crearea si utilizarea proceselor.

slide 8

```
pid_t fork()
```

EFECT: se copiaza imaginea procesului intr-o zona de memorie libera, aceasta copie fiind noul proces creat, in prima faza identic cu procesul initial. Cele doua procese isi continua executia in paralel dupa apelul lui fork.

Duplicarea procesului parinte nu se face in tottalitate. Partea **pura** ramane in memorie intr-un singur exemplar, iar duplicarea se face doar pe partea **impura**.

In sistem vor exista din acest moment doua procese independente, (desi identice), cu zone de date si stiva distincte. Orice **modificare** facuta, prin urmare, asupra unei variabile din procesul fiu, va ramane **invizibila** procesului parinte si invers.

slide 9

Procesul fiu va mosteni de la parinte toti **descriptorii de fisier** deschisi de catre acesta, asa ca orice prelucrari ulterioare in fisiere vor fi efectuate in punctul in care le-a lasat parintele.

Deoarece codul parintelui si codul fiului sunt identice si pentru ca aceste procese vor rula in continuare in paralel, trebuie facuta clar **distinctia**, in interiorul programului, intre actiunile ce vor fi executate de fiu si cele ale parintelui. Cu alte cuvinte, este nevoie de o metoda care sa indice care este portiunea de cod a parintelui si care a fiului.

Acest lucru se poate face simplu, folosind valoarea returnata de functia fork( ). Ea returneaza:

» -1, daca operatia nu s-a putut efectua (eroare)
» 0, in codul fiului
» pid, in codul parintelui, unde pid este identificatorul de proces al fiului nou-creat.

Prin urmare, o posibila schema de apelare a functiei fork( ) ar fi:

```
if (fork() == 0)
        { - - - instructiuni ale
                procesului fiu - - - }
else
        { - - - instructiuni ale
                procesului parinte - - - }
```

slide 10

```
    ...
if( ( pid=fork() ) < 0)
      {
        perror("Eroare");
        exit(1);
      }
if(pid==0)
      {
        /* codul fiului */
        ...
        exit(0)
      }
/* codul parintelui */
    ...
wait(&status)
```

```
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
...
pid_t childpid;
...
childpid=fork();
switch(childpid)
{
   case -1:
      fprintf(stderr,"ERROR: %s\n", sys_errlist[errno]);
      exit(1);
      break;
   case 0:
      /* Child's code goes here */
      break;
   default:
      /* Parent's code goes here */
      break;
}
```

Executia unui program extern; apelurile sistem *exec*()*

Functia fork( ) creeaza un proces identic cu procesul parinte. Pentru a crea un **nou proces care sa ruleze un program diferit de cel al parintelui**, aceasta functie se va folosi impreuna cu unul din apelurile sistem de tipul exec( ): execl( ), execlp( ), execv( ), execvp( ), execle( ), execve( ).

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Toate aceste functii primesc ca parametru un nume de fisier care reprezinta un program executabil si realizeaza lansarea in executie a programului. Programul va fi lansat atfel incat se va suprascrie codul, datele si stiva procesului care apeleaza exec( ), astfel incat, imediat dupa acest apel **programul initial nu va mai exista in memorie**. Procesul va ramane, insa, identificat prin acelasi numar (PID) si va mosteni toate eventualele redirectari facute in prealabil asupra descriptorilor de fisiere (de exemplu intrarea si iesirea standard). De asemenea, el va pastra relatia parinte-fiu cu procesul care a apelat fork( ).

slide 11

Singura situatie in care procesul apelant revine din apelul functiei exec( ) este acela in care operatia nu a putut fi efectuata, caz in care functia returneaza un cod de eroare (-1).

```
//EXEMPLE
...
main (int argc, char * argv[]){
    printf("\nTest execv\n");
    if(execv("/bin/ls", argv) < 0) printf("eroare la exec");
    printf("Dupa exec\n");
}

execl("/bin/ls", "/bin/ls", argv[1], 0);
```

Apelurile sistem *exit()*, *wait()* si *waitpid()*

```
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int flags)
```

Functia wait( ) este folosita pentru asteptarea terminarii fiului si preluarea valorii returnate de acesta. Parametrul status este folosit pentru evaluarea valorii returnate, folosind cateva macro-uri definite special (vezi paginile de manual corespunzatoare functiilor wait( ) si waitpid( )). Functia waitpid( ) este asemanatoare cu wait( ), dar asteapta terminarea unui anumit proces dat, in vreme ce wait( ) asteapta terminarea oricarui fiu al procesului curent. Este obligatoriu ca starea proceselor sa fie preluata dupa terminarea acestora, astfel ca functiile din aceasta categorie nu sunt optionale.

slide 12

Alte functii pentru lucrul cu procese

```
int system(const char *cmd)
```

Lanseaza in executie un program de pe disc, folosind in acest scop un apel fork( ), urmat de exec( ), impreuna cu waitpid( ) in parinte.

```
pid_t vfork()
```

Creeaza un nou proces, la fel ca fork( ), dar nu copiaza in intregime spatiul de adrese al parintelui in fiu. Este folosit in conjunctie cu exec( ), si are avantajul ca nu se mai consuma timpul necesar operatiilor de copiere care oricum ar fi inutile daca imediat dupa aceea se apeleaza exec( ) (oricum, procesul fiu va fi supascris cu programul luat de pe disc).

slide 13

slide 14

Starile unui proces Unix si tranzitia intre stari

Orice proces este executat secvential, iar mai multe procese pot sa ruleze in paralel (intre ele). De cele mai multe ori, executia in paralel se realizeaza alocand pe rand procesorul cate unui proces. Desi la un moment dat se executa un singur proces, in decurs de o secunda, de exemplu, pot fi executate portiuni din mai multe procese. Din aceasta schema rezulta ca un proces se poate gasi, la un moment dat, in una din urmatoarele trei stari [Tanenbaum]:

- » in executie
- » pregatit pentru executie
- » blocat

slide 15

slide 16

Procesul se gaseste **in executie** atunci cand procesorul ii executa instructiunile. **Pregatit de executie** este un proces care, desi ar fi gata sa isi continue executia, este lasat in asteptare din cauza ca un alt proces este in executie la momentul respectiv. De asemenea, un proces poate fi **blocat** din doua motive: el isi suspenda executia in mod voit sau procesul efectueaza o operatie in afara procesorului, mare consumatoare de timp (cum e cazul operatiilor de intrare-iesire - acestea sunt mai lente si intre timp procesorul ar putea executa parti din alte procese). Un proces Unix se poate afla intr-una din urmatoarele 9 stari:

slide 17

slide 18

1. Created

procesul este creat prin intermediul unui apel sistem fork

2. ReadyMemory

procesul ocupa loc in memoria RAM si este gata pentru a fi executat; nucleul decide momentul intrarii lui in executie.

3. ReadySwapped

*procesul este gata de executie, dar ocupa spatiu in memoria secundara (zona swap pe disc); pentru a intra in ciclul executiei trebuie sa intre mai intain in starea **ReadyMemory***

4. RunKernel

procesului ii sunt executate instructiuni in mod nucleu (ex: apeluri sistem, handler de intreruperi)

5. RunUser

sunt executate instructiunile programului

6. Preemted

aproape identica cu ReadyMemory. Dupa executia unui apel sistem procesul trebuie trecut in starea RunUser sau in starea ReadyMemory. Daca in starea ReadyMemory exista un proces mai prioritar, procesul curent este trecut in starea Preemted iar procesul mai prioritar trece in starea RunUser, altfel procesul curent trece in starea RunUser. Un proces aflat in starea

Preemted se muta in starea RunUser cand este cel mai prioritar.

7. SleepMemory

procesul se afla in memorie dar nu poate fi executat pana se produce un eveniment care sa-l scoata din aceasta stare

8. SleepSwapped

procesul este evacuat pe disc, el nu poate fi executat pana se produce un eveniment care sa-l scoata din acasta stare

9. Zombie

Procesul nu mai exista insa informarea faptului ca s-a terminat, trimisa spre procesul parinte, nu a fost inca preluta de acesta; ***nu a fost inca scos din tabela proceselor***