

Capitolul 7

Fire de execuție

O caracteristică esențială a unui sistem de operare de tip Unix/Linux este execuția concurrentă a mai multor procese, la fel ca și procesele, firele de execuție par a rula concurrent, responsabil de această “conurență” este nucleul, *kernel*-ul, sistemului de operare care întrerupe rularea *thread*-urilor la anumite intervale de timp pentru a oferi și altora șansa de a fi rulate.

Conceptual, *thread*-ul este o “subdiviziune”, el există doar în cadrul unui proces. La execuția unui program, sistemul de operare crează un nou proces și în cadrul procesului respectiv va crea un singur *thread* în care programul apelat se execută secvențial, opțional, din *thread*-ul inițial se pot crea *thread*-uri suplimentare, ansamblul acestor *thread*-uri reprezentând programul apelat, fiecare *thread* executând părți diferite ale programului la momente diferite de timp.

Dacă în cazul creării unui nou proces, procesul fiu execută programul procesului părinte dar cu resursele părintelui, memorie virtuală, descriptori de fișiere, ș.a., copiate procesul fiu putând modifica oricare resursă fără a afecta procesul părinte, și viceversa, la crearea unui *thread* nimic nu este copiat. Ambele *thread*-uri vor utiliza resursele sistem în comun, dacă un *thread* închide un descriptor de fișier, celelalte *thread*-uri nu vor mai putea efectua operații asupra acelu descriptor de fișier. De asemenea datorită faptului ca un proces, respectiv *thread*-urile asociate unui proces, poate executa doar un singur program la un moment dat, dacă un *thread* apelează una dintre funcțiile *exec* execuția celorlalte *thread*-uri este întreruptă.

Comutarea execuției între două procese implică un număr relativ mare de operații. Aceste operații sunt necesare pentru a se asigura faptul că nu apar interferențe între diferitelor resurse ale unor procese diferite, această condiție fiind necesară în cadrul unui sistem multiutilizator. În situația în care se dorește ca două sau mai multe procese să coopereze, pentru atingerea unui scop comun, acest obiectiv se poate realiza prin utilizarea mecanismelor de comunicare interproces. Totuși și în cazul utilizării acestor mecanisme sunt necesare operațiile de comunicare între procese care impun operații suplimentare. Dacă se dorește eliminarea acestor operații suplimentare atunci procesele pot fi înlocuite cu fire de execuție.

Firele de execuție mai sunt numite și “procese ușoare” (*light weight processes – LWP’s*). Un proces, în general, are mai multe părți componente cod, date, stivă, și de asemenea un anumit timp de execuție pe procesor. În cazul firelor de execuție vom avea mai multe procese care au alocate propriile lor intervale de timp de procesor dar folosesc în comun segmentele de cod, date, memoria și structurile de date.

În acest caz operațiile necesare pentru comutarea între două fire de execuție implică un efort semnificativ mai mic decât în situația unor procese. Deoarece firele de execuție au fost proiectate astfel încât să fie cooperative, nucleul sistemului de operare nu trebuie să ia măsuri suplimentare de protecție a resurselor acestora în cadrul unui proces. Din acest motiv firele de execuție au primit denumirea de “procese ușoare”. La polul opus se află procesele ca atare, care comparativ cu firele de execuție implică un efort de comutare mai mare și ca urmare se mai numesc și “procese grele” (*heavy weight processes – HWP’s*)

În general procesele pot fi implementate în două moduri:

- În spațiul nucleului – de obicei aceste procese sunt implementate în nucleul sistemului de operare, implementare ce se bazează pe controlul, în nucleu, a tabelilor de semnale asociate firului de execuție. În acest caz nucleul sistemului de operare este răspunzător de programarea execuției fiecărui fir, iar intervalul de timp alocat firului este scăzut din timpul alocat, global, procesului din care face parte firul. Dezavantajul acestei metode de control al timpului de execuție este dat de apariția unui efort suplimentar la comutarea utilizator-nucleu-utilizator. Avantajul major este dat de planificarea controlată a firelor de execuție (la expirarea cuantei de timp de procesor nucleul îi va întrerupe execuția și va transmite controlul firului următor), deci este exclusă blocarea procesului datorată unor operații de I/E, iar în cazul sistemelor de calcul cu mai multe procesoare performanța poate crește proporțional cu numărul de procesoare adăugate sistemului.
- În spațiul utilizatorului – în acest caz este evitat controlul nucleului asupra firului de execuție – pentru o astfel de implementare, numită și *multitasking cooperativ*, se definesc un set de rutine ce sunt responsabile de comutarea execuției între fire. Tipic un fir de execuție cedează în mod explicit procesorul, prin apelul explicit a unei rutine, trimiterii unui semnal. O problemă majoră ce poate apărea este acapararea de către un fir a întregului timp de procesor asociat unui proces (ex. dacă procesul execută o operație de I/E), o posibilă rezolvare este utilizarea unui semnal de timp care să determine comutarea între fire.

În paragrafele ce urmează discuția se va baza pe fire de execuție POSIX numite și *threads* (firele ce nu se conformează acestui standard se numesc *cthread*s).

În cadrul unui proces un *thread* este recunoscut pe baza unui identificator, iar la crearea fiecăr *thread* execută o funcție. Acesta este o funcție obișnuită ce conține codul ce trebuie rulat de *thread*, iar la ieșirea din funcție se termină și execuția *thread*-ului.

Apelul sistem de bază pentru crearea de noi *thread*-uri este *pthread_create*.

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *  
(*start_routine)(void *), void *arg);
```

Funcția *pthread_create* creează noi *thread*-uri și trebuie să i se transmită următoarele argumente:

- ⑩O referință către o variabilă de tip *pthread_t*, în care este salvat identificatorul *thread*-ului.
- ⑩O referință către un obiect de tip *pthread_attr_t*. Prin acest argument se controlează modul în care *thread*-ul interacționează cu restul programului. Dacă argumentul transmis este *NULL* atunci se vor folosi setările implicite.
- ⑩O referință către funcția ce urmează a fi executată de *thread*. Referința către funcție trebuie să fie de tipul: *void* (*) (void *)*.
- ⑩Argument către *thread*. Acesta trebuie să fie de tip *void **, și este transmis funcției *thread*-ului.

La apelul funcției *pthread_create* are loc crearea *thread*-ului, iar revenirea din funcție se face imediat. Rularea celor două *thread*-uri (părinte și fiu) are loc asincron, iar execuția unui program nu trebuie să se bazeze pe o anumită ordine de rulare a *thread*-urilor.

Un exemplu clasic de program este *Hello World*, de această dată realizat cu *thread*-uri:

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

```
//funcția ce va fi executată de thread-uri
```

```
void* print_message( void *ptr ){  
    char *message;  
    message = (char *) ptr;  
    printf("%s ", message);  
    return NULL;  
}
```

```
int main(){  
    pthread_t thread1, thread2;  
    char *message1 = "Hello";  
    char *message2 = "World\n";
```

```
// crearea primului thread – tipărește Hello  
if (pthread_create( &thread1, NULL, &print_message, (void*)message1)){  
    fprintf(stderr, "first thread error\n");  
    exit(1);  
}
```

```
// “adoarme” procesul părinte pentru o secundă.  
sleep(1);
```

```
// crearea celui de al doilea thread – tipărește World  
if(pthread_create( &thread2, NULL, &print_message, (void*)message2)){  
    fprintf(stderr, "second thread error\n");  
    exit(1);  
}
```

```
sleep(1);  
exit(0);  
}
```

Compilarea programului se realizează cu următoarea comandă:

```
$ gcc -o hello_thread hello_thread.cc -D_REENTRANT -lpthread
```

Opțiunea *lpthread* trebuie specificată la compilarea programului datorită faptului că suportul pentru *thread*-uri este adăugat sub forma unei biblioteci.

O problemă ce suportă două moduri de tratare, în cazul *thread*-urilor, este cea a terminării execuției și a valorii returnate de *thread*. O primă modalitate este cea prezentată în exemplul anterior în care funcția tratată de *thread* se încheie prin apelul *return*, iar valoarea transmisă este considerată ca fiind valoarea returnată de *thread*. O altă modalitate este prin intermediul apelului sistem *pthread_exit*.

```
#include <pthread.h>
```

```
void pthread_exit(void *retval);
```

Această funcție poate fi apelată direct din funcția *thread*-ului sau din altă funcție apelată direct sau indirect din funcția *thread*-ului. Valoarea primită ca și argument este valoarea ce va fi returnată de *thread*.

Tot din exemplul anterior se poate observa că transmiterea parametrilor către *thread* se realizează prin ultimul parametru al apelului *pthread_creat*. Dacă nu se dorește transmiterea unui parametru funcției *thread*-ului atunci argumentul respectiv primește valoarea *NULL*. Valoarea acestui parametru poate fi o referință către orice structură de date.

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

```
// definierea structurii de date ce va fi transmisă ca argument thread-ului
struct chars {
    char val;
    int count;
};
```

```
// funcția thread-ului
void *char_print(void *ptr){

    struct chars* c = (struct chars*)ptr;
    for(int i=0;i<c->count;i++) putchar(c->val);
    pthread_exit(0);
}
```

```
int main(){
    pthread_t thread1, thread2;
    struct chars thread1_arg,thread2_arg;
```

```
// stabilirea parametrilor pentru thread-uri
thread1_arg.val='a';
thread1_arg.count=100000;
```

```
thread2_arg.val='b';
thread2_arg.count=150000;
```

```
// creerea thread-urilor
pthread_create(&thread1,NULL,&char_print,&thread1_arg);
pthread_create(&thread2,NULL,&char_print,&thread2_arg);
```

```
// încheierea execuției
exit(0);
}
```

Exemplul de mai sus creează două *thread*-uri fiecare afișând câte un caracter de câte un anumit număr de ori, totodată este și un exemplu ce ilustrează modul în care se pot transmite funcției *thread*-ului argumente mai complexe. Totuși acest program ascunde o problemă de funcționare. Funcția *thread*-ului primește ca și argumente referințe către structuri de date definite ca și variabile locale în *thread*-ul principal al programului. Datorită faptului că nu există nici o regulă asupra ordinii de executare a *thread*-utilor este posibil ca *thread*-ul principal să își încheie execuția înaintea celor două *thread*-uri de afișare a caracterelor. O consecință este faptul că este dealocată memoria rezervată structurilor de date transmise ca și parametrii *thread*-urilor. O rezolvare a acestei probleme este apelul sistem *pthread_join*:

```
#include <pthread.h>
```

```
int pthread_join(pthread_t th, void **thread_return);
```

Acest apel sistem suspendă execuția *thread*-ului apelant până la încheierea execuției *thread*-ului identificat prin ID-ul *th*.

Pthread_join primește două argumente – primul este ID-ul *thread*-ului după care *thread*-ul apelant trebuie să aștepte să își încheie execuția, iar al doilea este o referință către o locație de memorie în care se va reține valoarea returnată de *thread*-ul *th*. Dacă valoarea returnată de *thread*-ul *th* nu interesează, atunci al doilea argument va fi *NULL*.

În consecință pentru ca exemplul anterior să fie funcțional trebuie completat cu următorul cod:

```
.
.
.
```

```

// crearea thread-urilor
pthread_create(&thread1,NULL,&char_print,&thread1_arg);
pthread_create(&thread2,NULL,&char_print,&thread2_arg);

// thread-ul părinte așteaptă încheierea execuției thread-urilor fiu.
pthread_join(thread1,NULL);
pthread_join(thread2,NULL);

// încheierea execuției
exit(0);
}

```

Observație: Trebuie avut grijă ca orice tip de date ce este transmis unui thread prin referință să nu fie dealocat, de thread-ul părinte sau de oricare alt thread.

Uneori este util de determinat care este thread-ul care execută un anumit cod. Pentru a determina ID-ul unui thread, avem la dispoziție apelul sistem `pthread_self`, care are ca și rezultat ID-ul thread-ului care îl apelează. De asemenea putem compara două thread-uri pe baza ID-urilor cu apelul `pthread_equal`.

```

#include <pthread.h>

pthread_t pthread_self(void);

int pthread_equal(pthread_t thread1, pthread_t thread2);

```

De exemplu este o eroare ca un thread să apeleze `pthread_join` pentru el însuși. Pentru a evita o astfel de eroare se poate utiliza următoarea secvență de cod:

```

if(!pthread_equal(pthread_self(), alt_thread))
    pthread_join(alt_thread, NULL);

```

Un alt aspect important în ceea ce privește crearea și utilizarea thread-urilor atributele thread-urilor ce pot fi transmise ca și argument funcției `pthread_create`. De fapt acest apel sistem primește ca și argument o referință către un obiect de tip `thread attribute`. Dacă referința este `NULL`, atunci se vor folosi atributele implicite pentru crearea thread-ului.

Dacă se dorește crearea unui thread pe baza unor atribute particulare este nevoie de crearea unui obiect de tip `thread attribute`. Relativ la acest tip de obiecte sunt definite mai multe apeluri sistem dintre care cele mai importante sunt: `pthread_attr_init` - inițializează obiectul primit ca și argument cu valorile implicite, și `pthread_attr_destroy` - dealocă resursele ocupate de obiectul primit ca și argument.

```

#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);

```

Pentru a defini un thread cu atribute particulare se aplică următorii pași:

1. Se creează un obiect de tipul `thread attribute` – se declară o variabilă de acest tip.
2. Obiectul creat este transmis ca și parametru funcției `pthread_attr_init` – obiectul este inițializat cu atributele implicite.
3. Se modifică atributele dorite.
4. Se creează un thread pe baza apelului `pthread_create` căruia i se transmite ca și argument nou creatul `thread attribute`.
5. Resursele ocupate de `thread attribute` trebuie eliberate – se folosește `pthread_attr_destroy`.

Observație: Un singur obiect de tip `thread attribute` poate fi utilizat pentru crearea mai multor thread-uri, după care prezența acestui nu mai este necesară și obiectul poate fi dealocat.

```

#include <pthread.h>

```

```

void* thread_function(void *arg){
    ....
}

int main(){
pthread_attr_t attr;
pthread_t thread;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
pthread_create(&thread,&attr,&thread_function,NULL);
pthread_attr_destroy(&attr);
....
exit(0);
}

```

Pentru majoritatea programelor un singur atribut al *thread*-ului este important, restul atributelor se utilizează doar în cazul programelor de timp real. Acest atribut este *detach state*. Conform acestui atribut un *thread* poate fi creat fie *joinable* (implicit), fie *detached*. În primul caz, *joinable*, starea *thread*-ului nu este preluată de către sistemul de operare automat și rămâne în sistem, ca și în cazul proceselor *zombie*, până când un alt *thread* apelează *pthread_join*. În cazul în care *thread*-ul creat are atributul *detached*, starea de ieșire a *thread*-ului este automat preluată de sistemul de operare. Dezavantajul în acest caz este dat de dispariția posibilității de sincronizare între *thread*-uri prin intermediul funcției *pthread_join*.

Observație: Chiar dacă un *thread* a fost creat inițial ca și *joinable* ulterior poate fi *detached* prin intermediul apelului sistem *pthread_detach*.

În condiții normale un *thread* își încheie execuția fie prin terminarea rulării funcției *thread*-ului returnând o valoare, fie prin apelul sistem *pthread_exit*. Mai există și o a treia posibilitate în care un *thread* solicită unui alt *thread* încheierea execuției. În acest caz este folosit apelul sistem *pthread_cancel*:

```

#include <pthread.h>

int pthread_cancel(pthread_t thread);

```

Deseori execuția normală a unui *thread* implică alocarea unor resurse, utilizarea lor și în final dealocarea lor. Dacă execuția unui *thread* este anulată în mijlocul unui astfel de proces, resursele alocate inițial se vor pierde.

În concluzie din acest punct de vedere un *thread* se poate afla în una din următoarele stări:

- ⓐ Asincron anulabil – își poate încheia execuția în orice moment.
- ⓑ Sincron anulabil – se poate solicita din exterior în cheiera execuției, dar nu în orice moment. Cererile de încheiere a execuției vor fi puse într-o coadă de așteptare și vor fi luate în considerare doar în anumite momente ale execuției.
- ⓒ Neanulabil – toate cererile de încheiere a execuției sunt ignorate.

Observație: Implicit toate *thread*-urile sunt sincron anulabile.

Acest mecanism de terminare a execuției unui *thread* pe baza unor cereri externe se bazează pe următoarele apeluri sistem:

```

#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);

int pthread_setcanceltype(int type, int *oldtype);

void pthread_testcancel(void);

```

Așa cum am s-a arătat anterior unui *thread* asincron anulabil i se poate solicita în orice moment să își încheie execuția, în timp ce un *thread* sincron anulabil își poate încheia execuția doar în anumite momente numite puncte de anulare.

Pentru ca un *thread* să fie asincron anulabil (implicit este sincron) se folosește apelul *pthread_setcanceltype* unde primul argument este *PTHREAD_CANCEL_ASYNCHRONOUS* (*PTHREAD_CANCEL_DEFFERD* pentru a reveni la cazul sincron), al doilea argument este o referință către o variabilă care conține vechea stare, dacă lipsește – *NULL*:

```

pthread_setcanceltype( PTHREAD_CANCEL_ASYNCHRONOUS, NULL);

```

Pentru a testa dacă un *thread* este anulabil într-un anumit punct se utilizează – *pthread_testcancel*. Utilizarea acestui apel sistem este valabilă în cazul *thread*-urilor asincrone.

Se poate indica din interiorul *thread* -ului secțiunile neanulabile prin apelul *pthread_setcancelstate*:

```
...  
  
int old_cancel_state;  
  
...  
  
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_cancel_state)  
  
...  
  
pthread_setcancelstate(old_cancel_state, NULL);  
  
...
```

Pe baza acestui apel se pot implementa secțiuni critice, care fie se execută în întregime, fie nu se execută de loc. Dacă un *thread* începe execuția unei astfel de secțiuni atunci trebuie să o continue până la sfârșitul secțiunii fără a fi anulat.

Implementarea *thread*-urilor POSIX este diferită de implementarea din multe alte sisteme de operare de tip UNIX. Diferența este dată de faptul că în Linux *thread*-urile sunt implementate ca și procese. Fiecare apel *pthread_create* creează un nou proces ce rulează *thread*-ul nou creat. Oricum acest proces este diferit de cel creat cu *fork*, el utilizează în comun spațiul de adrese și resursele cu procesul original.

O problemă ce se poate pune în cazul programelor implementate multithread este gestionarea semnalelor. Să presupunem că un astfel de program primește un semnal din exterior, se pune întrebarea care *thread* va apela funcția de tratare a semnalului (*handler*-ul)? În cazul sistemelor de operare de tip Linux problema este rezolvată chiar de implementarea acestora de tip proces – semnalul va fi tratat de *thread*-ul ce rulează programul principal. De asemenea, în interiorul unui program multithread este posibil, ca un *thread* să poată trimite un semnal unui alt *thread*, pentru aceasta se folosește apelul sistem *pthread_kill*.