2 Concurență la nivel de threaduri Unix

2.1 Relația procese - thread-uri

La nivel conceptual, noțiunile de proces şi thread sunt relativ apropiate. Diferențele între ele ies în evidență atunci când se au în vedere aspectele de implementare ale acestora.

2.1.1 Definirea procesului

Ce este un proces? Un *proces* sau *task*, este un calcul care poate fi executat concurent (în paralel) cu alte calcule. El este o abstractizare a activității procesorului, fiind considerat ca un program în execuție. Existența unui proces este condiționată de existența a trei factori:

- o *procedură* - o *succesiune de instrucțiuni* dintr-un *set predefinit de instrucțiuni*, cu rolul de descriere a unui calcul - descrierea unui algoritm.
- un *procesor* - dispozitiv hardware/software ce recunoaşte şi poate executa setul predefinit de instrucțiuni, şi care este folosit, în acest caz, pentru a executa succesiunea de instrucțiuni specificată în procedură;
- un *mediu* - constituit din partea din resursele sistemului: o parte din memoria internă, un spațiu disc destinat unor fişiere, periferice magnetice, echipamente audio-video etc. - asupra căruia acționează procesorul în conformitate cu secvența de instrucțiuni din procedură.

2.1.2 Reprezentarea în memorie a unui proces

In ceea ce priveşte reprezentarea în memorie a unui proces, indiferent de platforma (sistemul de operare) pe care este operațional, se disting, în esență, următoarele zone:

- Contextul procesului
- Codul programului
- Zona datelor globale
- Zona heap
- Zona stivei

Contextul procesului conține informațiile de localizare în memoria internă şi informațiile de stare a execuției procesului:

- Legături exterioare cu platforma (sistemul de operare): numele procesului, directorul curent în structura de directori, variabilele de mediu etc.;
- Pointeri către începuturile zonelor de cod, date stivă şi heap şi, eventual, lungimile acestor zone;

- Starea curentă a execuției procesului: contorul de program (notat PC -program counter) ce indică în zona cod următoarea instrucțiune mașină de executat, pointerul spre vârful stivei (notat SP - stack pointer);
- Zone de salvare a regiştrilor generali, de stare a sistemului de întreruperi etc.

De exemplu, în [15] se descrie contextul procesului sub sistemul de operare DOS, iar în [13] contextul procesului sub sistemul de operare Unix.

Zona de cod conține instrucțiunile mașină care dirijează funcționarea procesului. De regulă, conținutul acestei zone este stabilit încă din faza de compilare. Programatorul descrie programul într-un limbaj de programare de nivel înalt. Textul sursă al programului este supus procesului de compilare care generează o secvență de instrucțiuni mașină echivalentă cu descrierea din program.

Conținutul acestei zone este folosit de procesor pentru a-şi încărca rând pe rând instrucțiunile de executat. Registrul PC indică, în fiecare moment, locul unde a ajuns execuția.

Zona datelor globale conține constantele şi variabilele vizibile de către toate instrucțiunile programului. Constantele şi o parte dintre variabile primesc valori încă din faza de compilare. Aceste valori inițiale sunt încărcate în locațiile de reprezentare din zona datelor globale în momentul încărcării programului în memorie.

Zona heap - cunoscută şi sub numele de zona variabilelor dinamice - găzduieşte spații de memorare a unor variabile a căror durată de viață este fixată de către programator. *Crearea* (operația *new*) unei astfel de variabile înseamnă rezervarea în heap a unui şir de octeți necesar reprezentării ei şi întoarcerea unui pointer / referințe spre începutul acestui şir. Prin intermediul referinței se poate utiliza în scriere şi/sau citire această variabilă până în momentul *distrugerii* ei (operație *destroy*, *dispose* etc.). Distrugerea înseamnă eliberarea şirului de octeți rezervat la creare pentru reprezentarea variabilei. In urma distrugerii, octeții eliberați sunt plasați în lista de spații libere a zonei heap.

In [13,47,2] sunt descrise mecanismele specifice de gestiune a zonei heap.

Zona stivă In momentul în care programul apelează o procedură sau o funcție, se depun în vârful stivei o serie de informații: parametrii transmişi de programul apelator către procedură sau funcție, adresa de revenire la programul apelator, spațiile de memorie necesare reprezentării variabilelor locale declarate şi utilizate în interiorul procedurii sau funcției etc. După ce procedura sau funcția îşi încheie activitatea, spațiul din vârful stivei ocupat la momentul apelului este eliberat. In cele mai multe cazuri, există o stivă unică pentru fiecare proces. Există însă platforme, DOS este un exemplu [14], care folosesc mai multe stive simultan: una rezervată numai pentru proces, alta (altele) pentru apelurile sistem. Conceptul de thread, pe care-l vom prezenta imediat, induce ca regulă generală existența mai multor spații de stivă.

In figura 2.4 sunt reprezentate două procese active simultan într-un sistem de calcul.

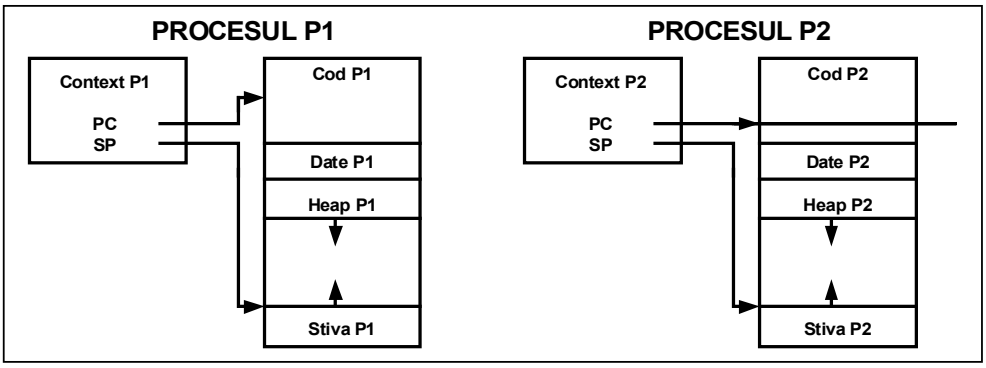

Figura 2.1 Două procese într-un sistem de calcul

2.1.3 Definiția threadului

Conceptul de *thread*, sau *fir de execuție*, a apărut în ultimii 10-15 ani. Proiectanții şi programatorii au "simțit nevoia" să-şi definească entități de calcul independente, dar în cadrul aceluiaşi proces. Astfel, un *thread* se defineşte ca o entitate de execuție din interiorul unui proces, compusă dintr-un context şi o secvență de instrucțiuni de executat.

Deşi noțiunea de thread va fi prezentată pe larg în capitolele următoare, punctăm aici câteva caracteristici de bază ale acestor entități:

- Thread-urile sunt folosite pentru a crea programe formate din unități de procesare concurentă.
- Entitatea thread execută o secvență dată de instrucțiuni, încapsulate în funcția thread-ului.
- Execuția unui thread poate fi întreruptă pentru a permite procesorului să dea controlul unui alt thread.
- Thread-urile sunt tratate independent, fie de procesul însuşi, fie de nucleul sistemului de operare. Componenta sistem (proces sau nucleu) care gestionează thread-urile depinde de modul de implementare a acestora.
- Operațiile de lucru cu thread-uri sunt furnizate cu ajutorul unor librării de programe (C, C++) sau cu ajutorul unor apeluri sistem (în cazul sistemelor de operare: Windows NT, Sun Solaris).

Esența conceptuală a threadului este aceea că execută o procedură sau o funcție, în cadrul aceluiaşi proces, concurent cu alte thread-uri. Contextul şi zonele de date ale procesului sunt utilizate în comun de către toate thread-urile lui.
Esența de reprezentare în memorie a unui thread este faptul că singurul spațiu de memorie ocupat exclusiv este spațiul de stivă. In plus, fiecare thread îşi întreține propriul context, cu elemente comune contextului procesului părinte al threadului.

In figura 2.5 sunt reprezentate trei thread-uri în cadrul aceluiaşi proces.

Există cazuri când se preferă folosirea proceselor în locul thread-urilor. De exemplu, când este nevoie ca entitățile de execuție să aibă identificatori diferiți sau să-şi

gestioneze independent anumite atribute ale fişierelor (directorul curent, numărul maxim de fişiere deschise) [96].

Un program multi-thread poate să obțină o performanță îmbunătățită prin execuția concurentă şi/sau paralelă a thread-urilor. Execuția concurentă a thread-urilor (sau pe scurt, *concurență*) înseamnă că mai multe thread-uri sunt în *progres*, în acelaşi timp. Execuția paralelă a thread-urilor (sau pe scurt, *paralelism*) apare când mai multe thread-uri se execută *simultan* pe mai multe procesoare.

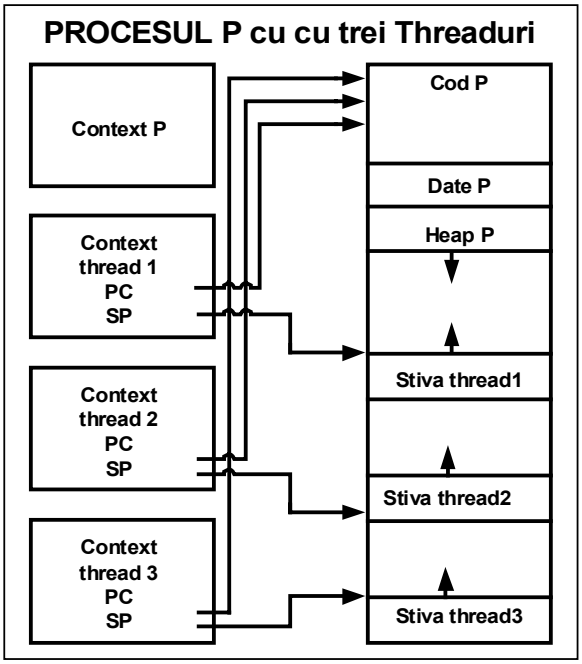

Figura 2.2 Trei thread-uri într-un proces

2.2 Thread-uri pe platforme Unix: Posix şi Solaris

2.2.1 Caracteristici şi comparații Posix şi Solaris

Thread-urile Posix sunt răspândite pe toate platformele Unix, inclusiv Linux, SCO, AIX, Xenix, Solaris, etc. In particular, unele dintre aceste platforme au şi implementări proprii, cu caracteristici mai mult sau mai puțin apropiate de Posix. Dintre acestea, implementarea proprie platformei Solaris este cea mai elaborată şi mai răspândită, motiv pentru care o vom trata în această secțiune, în paralel cu implementarea Posix. Cele două modele au multe similarități la nivel sintactic dar au modalități de implementare diferite

2.2.1.1 Similarități şi facilități specifice

Intre thread-urile Posix şi Solaris există un grad Posix mare de similaritate, atât la nivel *sintactic*, cât şi *funcțional*. Pentru operațiile principale cu entitățile thread există apeluri diferite pentru Posix şi Solaris. Funcțiile thread-urilor Posix au prefixul `pthread_` iar cele Solaris `thr_`. Astfel, o conversie a unui program simplu cu thread-uri de pe una din cele două platforme pe cealaltă, se realizează doar la nivel sintactic, modificând numele funcției şi a unor parametri.
Inainte de a detalia principalele operații cu thread-uri Posix şi Solaris, punctăm câteva diferențe Intre aceste platforme.

Facilități Posix care nu sunt prezente pe Solaris:
- portabilitate totală pe platforme ce suportă Posix
- obiecte purtătoare de atribute
- conceptul de abandon (cancellation)
- politici de securitate

Facilități Solaris care nu sunt prezente pe Posix:
- blocări *reader/writer*
- posibilitatea de a crea thread-uri daemon
- suspendarea şi continuarea unui thread
- setarea nivelului de concurență
- (crearea de noi lwp-uri)

2.2.2 Operații asupra thread-urilor: creare, terminare

Inainte de a descrie operațiile cu thread-uri, trebuie să precizăm că pentru thread-urile Posix trebuie folosit fişierul header `<pthread.h>`, iar pentru Solaris headerele `<sched.h>` şi `<thread.h>`. Compilările în cele două variante se fac cu opțiunile: `-lpthread` (pentru a indica faptul că se foloseşte biblioteca `libpthread.so.0`, în cazul thread-urilor Posix), respectiv opțiunea `-lthread`, care link-editează programul curent cu biblioteca `libthread.so`, în cazul thread-urilor Solaris.

2.2.2.1 Crearea unui thread

Posix API:

```
int pthread_create(pthread_t *tid, pthread_attr_t *attr,
      void *(*func)(void*), void *arg);
```

Solaris API:

```
int    thr_create(void    *stkaddr,    size_t    stksize,    void
*(*func)(void*),
      void *arg, long flags, thread_t *tid);
```

Prin aceste funcții se creează un thread în procesul curent şi se depune în pointerul `tid` descriptorul threadului. Funcțiile întorc valoarea 0 la succes şi o valoare nenulă (cod de eroare), în caz de eşec.

Execuția noului thread este descrisă de funcția al cărei nume este precizat prin parametrul `func`. Această funcție are un singur argument de tip pointer, transmis prin argumentul `arg`.

Varianta Posix prevede atributele threadului prin argumentul `attr`. Asupra acestuia vom reveni Intr-o secțiune ulterioară. Pe moment vom folosi pentru `attr` valoarea NULL, prin aceasta indicând stabilirea de atribute implicite de către sistem.

`stkaddr` şi `stksize` indică adresa şi lungimea stivei threadului. Valorile NULL şi 0 pentru aceşti parametri cer sistemului să fixeze valori implicite pentru adresa şi dimensiunea stivei.

Parametrul `flags` -reprezintă o combinație de constante legate prin '|', cu valori specifice thread-urilor Solaris:

- THR_SUSPENDED – determină crearea threadului în starea suspendat pentru a permite modificarea unor atribute de planificare înainte de execuția funcției ataşată threadului. Pentru a începe execuția threadului se apelează `thr_continue`.
- THR_BOUND –leagă threadul de un nou lwp creat în acest scop. Threadul va fi planificat doar pe acest lwp.
- THR_DETACHED – creează un nou thread în starea detached. Resursele unui thread detached sunt eliberate imediat la terminarea threadului. Pentru acest thread nu se poate apela `thr_join` şi nici nu se poate obține codul de ieşire.
- THR_INCR_CONC sau, echivalent THR_NEW_lwp – incrementează nivelul de concurență prin adăugarea unui lwp la colecția inițială, indiferent de celelalte flag-uri.
- THR_DAEMON – creează un nou thread cu statut de daemon (de exemplu, pentru a trata evenimente asincrone de I/O). Procesul de bază se termină când ultimul thread non-daemon îşi încheie execuția.
- Dacă sunt precizate atât THR_BOUND cât şi THR_INCR_CONC sunt create două lwp-uri odată cu crearea threadului.

2.2.2.2 Terminarea unui thread

In mod obişnuit, terminarea unui thread are loc atunci când se termină funcția care descrie threadul. Din corpul acestei funcții se poate comanda terminarea threadului prin apelurile funcțiilor `pthread_exit`, respectiv `thr_exit`, cu prototipurile:

Posix API:
```
int pthread_exit(int *status);
```

Solaris API:
```
int thr_exit(int *status);
```

Pointerul `status` se foloseşte atunci când se doreşte ca în urma execuției threadul să întoarcă nişte date rezultat spre procesul părinte al threadului. După cum vom vedea imediat, acesta obține datele printr-un apel de tip `join`. In cele mai multe cazuri, `status` are valoarea NULL.

Un apel `pthread_exit/thr_exit` termină procesul, numai dacă threadul apelant este ultimul thread non-daemon din proces.

La terminarea unui thread, acesta este pus într-o listă *deathrow*. Din motive de performanță, resursele asociate unui thread nu sunt eliberate imediat ce acesta îşi încheie execuția. Un thread special, numit *reaper* parcurge periodic lista deathrow şi dealocă resursele thread-urilor terminate [30].

Posix permite comandarea terminării - abandonului - unui thread de către un alt thread. Trebuie remarcat că, în cazul utilizării abandonului, pot să apară probleme dacă execuția threadului este întreruptă în interiorul unei secțiuni critice. De asemenea, threadul nu va fi abandonat înainte de a dealoca resurse precum segmente de memorie partajată sau descriptori de fişiere.

Pentru a depăsi problemele de mai sus, interfața *cancellation* permite abandonarea execuției threadului, doar în anumite puncte. Aceste puncte, numite *puncte de abandon*, pot fi stabilite prin apeluri specifice.

Abandonarea threadului se poate realiza în 3 moduri:
i) asincron, *ii)* în diferite puncte, în timpul execuției, conform specificațiilor de implementare a acestei facilități sau *iii)* în puncte discrete specificate de aplicație.

Pentru efectuarea abandonului se apelează:

```
int pthread_cancel(pthread_t tid);
```

cu identificatorul threadului ca argument.

Cererea de abandon este tratată în funcție de starea threadului. Această stare se poate seta folosind apelurile: `pthread_setstate()` şi `pthread_setcanceltype()`.

Funcția `pthread_setcancelstate` stabileşte punctul respectiv, din threadul curent, ca punct de abandon, pentru valoarea PTHREAD_CANCEL_ENABLE şi interzice starea de abandon, pentru valoarea PTHREAD_CANCEL_DISABLE.

Funcția `pthread_setcanceltype` poate seta starea threadului la valoarea PTHREAD_CANCEL_DEFERRED sau PTHREAD_CANCEL_ASYNCHRONOUS. Pentru prima valoare, întreruperea se poate produce doar în puncte de abandon, iar pentru cea de a doua valoare, întreruperea se produce imediat.

In rutina threadului, punctele de abandon se pot specifica explicit cu `pthread_testcancel()`.
Funcțiile de aşteptare precum `pthread_join`, `pthread_cond_wait` sau `pthread_cond_timedwait` determină, de asemenea, puncte implicite de

abandon. Variabilele mutex şi funcțiile ataşate acestora nu generează puncte de abandon.

Marcarea pentru ştergere a unui thread se face prin apelul:

```
int pthread_detach(pthread_t tid);
```

Această funcție marchează pentru ştergere structurile interne ale threadului. In acelaşi timp, apelul informează nucleul că după terminare threadul `tid` va fi radiat, iar memoria ocupată de el eliberată. Aceste acțiuni vor fi efectuate abia după ce threadul `tid` îşi termină în mod normal activitatea.

2.2.2.3 Aşteptarea terminării unui thread

Cea mai simplă metodă de sincronizare a thread-urilor este aşteptarea terminării unui thread. Ea se realizează prin apeluri de tip `join`:

Posix API: `int pthread_join(pthread_t tid, void **status);`

Solaris API: `int thr_join(thread_t tid, thread_t *realid, void **status);`

Funcția `pthread_join` suspendă execuția threadului apelant până când threadul cu descriptorul `tid` îşi încheie execuția.

In cazul apelului `thr_join`, dacă `tid` este diferit de NULL, se aşteaptă terminarea threadului `tid`. In caz contrar, se aşteaptă terminarea oricărui thread, descriptorul threadului terminat fiind întors în parametrul `realid`.

Dublul pointer `status` primeşte ca valoare pointerul `status` transmis ca argument al apelului `pthread_exit`, din interiorul threadului. In acest fel, threadul terminat poate transmite apelatorului `join` o serie de date.

Thread-urile pentru care se apelează funcțiile `join`, nu pot fi create în starea *detached*. Resursele unui thread *joinable*, care şi-a încheiat execuția, nu se dealocă decât când pentru acest thread este apelată funcția `join`. Pentru un thread, se poate apela `join` o singură dată.

2.2.2.4 Un prim exemplu

Fără a intra deocamdată în prea multe amănunte, pregătim prezentarea programului `ptAtribLung.c`, (programul 4.1), parametrizat de două constante (deci în patru variante posibile), şi în care introducem utilizarea thread-urilor sub Unix.

```
//#define ATRIBLUNG 1
//#define MUTEX 1
#include <stdio.h>
#include <pthread.h>
```

```
typedef struct { char *s;  int nr;  int pas;  int sec;}argument;
int p = 0;
pthread_mutex_t mutp = PTHREAD_MUTEX_INITIALIZER;

void f (argument * a) {
   int i, x;
   for (i = 0; i < a->nr; i++) {
   #ifdef MUTEX
   pthread_mutex_lock (&mutp);
   #endif
      x = p;
      if (a->sec > 0)
         sleep (random () % a->sec);
      printf ("\n%s i=%d pas=%d", a->s, i, a->pas);
      x += a->pas;
   #ifdef ATRIBLUNG
   p = x;
   #else
      p += a->pas;
   #endif
   #ifdef MUTEX
   pthread_mutex_unlock (&mutp);
   #endif
   }
}

main () {
   argument x = { "x:", 20, -1, 2 },
   argument y = { "y:", 10,  2, 3 };
   pthread_t th1, th2;

   pthread_create ((pthread_t *) & th1, NULL, (void *) f,
                   (void *) &x);
   pthread_create ((pthread_t *) & th2, NULL, (void *) f,
                   (void *) &y);
   pthread_join (th1, NULL);
   pthread_join (th2, NULL);
   printf ("\np: %d\n", p);
}
```

Programul 2.1 Sursa `primPThread.c`

Este vorba de crearea şi lansarea în execuție (folosind funcția `pthread_create`) a două thread-uri, ambele având aceeaşi acțiune, descrisă de funcția `f`, doar cu doi parametri diferiți. După ce se aşteaptă terminarea activității lor (folosind funcția `pthread_join`), se tipăreşte valoarea variabilei globale `p`. Rezultatele pentru cele patru variante sunt prezentate în tabelul următor. Pentru compilare, trebuie să fie specificată pe lângă sursă şi biblioteca `libpthread.so.0`. (sau, pe scurt, se specifică numele `pthread`, la opțiunea `-l`).

Tabelul din figura 4.6 ilustrează funcționarea programului în cele patru variante. In această secțiune ne vor interesa doar primele două coloane.

In continuare, o să explicăm, pe rând, elementele introduse de către acest program, făcând astfel primii paşi în lucrul cu thread-uri. Pentru o mai bună înțelegere, este bine ca din program să se "rețină" din sursă numai liniile ce rămân în urma parametrizării.

	ATRIBLUNG	MUTEX	ATRIBLUNG MUTEX
x: i=0 pas=-1	x: i=0 pas=-1	x: i=0 pas=-1	x: i=0 pas=-1
y: i=0 pas=2	y: i=0 pas=2	y: i=0 pas=2	y: i=0 pas=2
x: i=1 pas=-1	x: i=1 pas=-1	x: i=1 pas=-1	x: i=1 pas=-1
y: i=1 pas=2	y: i=1 pas=2	y: i=1 pas=2	y: i=1 pas=2
x: i=2 pas=-1	x: i=2 pas=-1	x: i=2 pas=-1	x: i=2 pas=-1
y: i=2 pas=2	x: i=3 pas=-1	y: i=2 pas=2	y: i=2 pas=2
x: i=3 pas=-1	x: i=4 pas=-1	x: i=3 pas=-1	x: i=3 pas=-1
y: i=3 pas=2	y: i=2 pas=2	y: i=3 pas=2	y: i=3 pas=2
x: i=4 pas=-1	x: i=5 pas=-1	x: i=4 pas=-1	x: i=4 pas=-1
y: i=4 pas=2	x: i=6 pas=-1	y: i=4 pas=2	y: i=4 pas=2
x: i=5 pas=-1	y: i=3 pas=2	x: i=5 pas=-1	x: i=5 pas=-1
x: i=6 pas=-1	x: i=7 pas=-1	y: i=5 pas=2	y: i=5 pas=2
y: i=5 pas=2	x: i=8 pas=-1	x: i=6 pas=-1	x: i=6 pas=-1
x: i=7 pas=-1	y: i=4 pas=2	y: i=6 pas=2	y: i=6 pas=2
x: i=8 pas=-1	x: i=9 pas=-1	x: i=7 pas=-1	x: i=7 pas=-1
x: i=9 pas=-1	x: i=10 pas=-1	y: i=7 pas=2	y: i=7 pas=2
x: i=10 pas=-1	x: i=11 pas=-1	x: i=8 pas=-1	x: i=8 pas=-1
x: i=11 pas=-1	x: i=12 pas=-1	y: i=8 pas=2	y: i=8 pas=2
x: i=12 pas=-1	y: i=5 pas=2	x: i=9 pas=-1	x: i=9 pas=-1
x: i=13 pas=-1	x: i=13 pas=-1	y: i=9 pas=2	y: i=9 pas=2
x: i=14 pas=-1	x: i=14 pas=-1	x: i=10 pas=-1	x: i=10 pas=-1
y: i=6 pas=2	x: i=15 pas=-1	x: i=11 pas=-1	x: i=11 pas=-1
y: i=7 pas=2	y: i=6 pas=2	x: i=12 pas=-1	x: i=12 pas=-1
x: i=15 pas=-1	x: i=16 pas=-1	x: i=13 pas=-1	x: i=13 pas=-1
x: i=16 pas=-1	x: i=17 pas=-1	x: i=14 pas=-1	x: i=14 pas=-1
x: i=17 pas=-1	x: i=18 pas=-1	x: i=15 pas=-1	x: i=15 pas=-1
x: i=18 pas=-1	x: i=19 pas=-1	x: i=16 pas=-1	x: i=16 pas=-1
y: i=8 pas=2	y: i=7 pas=2	x: i=17 pas=-1	x: i=17 pas=-1
x: i=19 pas=-1	y: i=8 pas=2	x: i=18 pas=-1	x: i=18 pas=-1
y: i=9 pas=2	y: i=9 pas=2	x: i=19 pas=-1	x: i=19 pas=-1
p: 0	p: 20	p: 0	p: 0

Figura 2.3 Comportări ale programului 4.1

Vom începe cu cazul în care nici una dintre constante nu este definită. In spiritul celor de mai sus, variabila `mutp` nu este practic utilizată, vom reveni asupra ei în secțiunile următoare. De asemenea, se vede că variabila `p` este modificată direct cu parametrul de intrare, fără a mai folosi ca şi intermediar variabila `x`.

Din conținutul programului se observă că funcția `f` primeşte ca argument o variabilă incluzând în structura ei: numele variabilei, numărul de iterații al funcției `f` şi pasul de incrementare al variabilei `p`. Se vede că cele două thread-uri primesc şi cedează relativ aleator controlul. Variabila `x` indică 20 iterații cu pasul –1, iar `y` 10 iterații cu pasul 2.

In absența constantei `ATRIBLUNG`, incrementarea lui `p` se face printr-o singură instrucțiune, `p+=a->pas`. Este extrem de puțin probabil ca cele două thread-uri să-şi treacă controlul de la unul la altul exact în timpul execuției acestei instrucțiuni. In consecință, variabila `p` rămâne în final cu valoarea 0.

Prezența constantei `ATRIBLUNG` are menirea să "încurce" lucrurile. Se vede că incrementarea variabilei `p` se face prin intermediul variabilei locale `x`. Astfel, după ce reține în `x` valoarea lui `p`, threadul stă în aşteptare un număr aleator de secunde şi abia după aceea creşte `x` cu valoarea `a->pas` şi apoi atribuie lui `p` noua valoare. In

mod natural, în timpul aşteptării celălalt thread devine activ şi îşi citeşte şi el aceeaşi valoare pentru `p` şi prelucrarea continuă la fel. Bineînțeles, aceste incrementări întrețesute provoacă o mărire globală incorectă a lui `p`, motiv pentru care `p` final rămâne cu valoarea 20. (De fapt, acest scenariu concret de aşteptări are drept consecință faptul că efectul global coincide cu efectul celui de-al doilea thread.)

2.2.3 Instrumente standard de sincronizare

Instrumentele (obiectele) de sincronizare specifice thread-urilor sunt, aşa cum am arătat în 2.5: variabilele mutex, variabilele condiționale, semafoarele şi blocările cititor/scriitor (reader/writer). Fiecare variabilă de sincronizare are asociată o coadă de thread-uri care aşteaptă - *sunt blocate*- la variabila respectivă.

Cu ajutorul unor primitive ce verifică dacă variabilele de sincronizare sunt disponibile, fiecare thread blocat va fi "trezit" la un moment dat, va fi şters din coada de aşteptare şi controlul lor va fi cedat componentei de planificare. Trebuie remacat faptul că thread-urile sunt repornite într-o ordine arbitrară, neexistând nici o relație între ordinea în care au fost blocate şi elimnarea lor din coada de aşteptare.

O observație importantă! In situația în care un obiect de sincronizare este blocat de un thread, iar acest thread îşi încheie fără a debloca obiectul, acesta - obiectul de sincronizare - va rămâne blocat! Este deci posibil ca thread-urile blocate la obiectul respectiv vor intra în impas.

In continuare descriem primitivele de lucru cu aceste obiecte (variabile, entități) de sincronizare pe platforme Unix, atât sub Posix, cât şi sub Solaris.

2.2.3.1 Operații cu variabile mutex

Inițializarea unei variabile mutex se poate face static sau dinamic, astfel:
Posix, inițializare statică:

```
pthread_mutex_t numeVariabilaMutex = PTHREAD_MUTEX_INITIALIZER;
```

Posix, inițializare dinamică:

```
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t
*mutexattr);
```

Solaris, inițializare statică:

```
mutex_t numeVariabilaMutex = 0;
```

Solaris, inițializare dinamică:

```
int mutex_init(mutex_t *mutex, int type, void *arg);
```

Deci, inițializarea statică presupune atribuirea unei valori standard variabilei mutex. Inițializarea dinamică se face apelând o funcție de tip `init`, având ca prim argument un pointer la variabila mutex.

Apelul `pthread_mutex_init` inițializează variabila mutex cu atribute specificate prin parametrul `mutexattr`. Semnificația acestei variabile o vom

prezenta într-o secțiune ulterioară. Pe moment acest argument are valoarea NULL, ceea ce semnifică fixarea de atribute implicite.

Parametrul `type` din apelul `mutex_init` indică domeniul de vizibilitate al variabilei mutex. Dacă are valoarea USYNC_PROCESS, atunci ea poate fi accesată din mai multe procese. Dacă are valoarea USYNC_THREAD, atunci variabila este accesibilă doar din procesul curent. Argumentul `arg` este rezervat pentru dezvoltări ulterioare, deci singura valoare permisă este NULL.

Distrugerea unei variabile mutex înseamnă eliminarea acesteia şi eliberarea resurselor ocupate de ea. In prealabil, variabila mutex trebuie să fie deblocată. Apelurile de distrugere sunt:
Posix:
```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```
Solaris:
```
int mutex_destroy(mutex_t *mutex);
```

Blocarea unei variabile mutex:
Posix:
```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```
Solaris:
```
int mutex_lock(mutex_t *mutex);
int mutex_trylock(mutex_t *mutex);
```

In apelurile `lock`, dacă variabila `mutex` nu este blocată de alt thread, atunci ea va deveni proprietatea threadului apelant şi funcția returnează imediat. Dacă este deja blocată de un alt thread, atunci funcția intră în aşteptare până când variabila `mutex` va fi eliberată.

In apelurile `trylock`, funcțiile returnează imediat, indiferent dacă variabila `mutex` este sau nu blocată de alt thread. Dacă variabila este liberă, atunci ea va deveni proprietatea threadului. Dacă este blocată de un alt thread (sau de threadul curent), funcția returnează imediat cu codul de eroare EBUSY.

Deblocarea unei variabile `mutex` se realizează prin:
Posix:
```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```
Solaris:
```
int mutex_unlock(mutex_t *mutex);
```

Se presupune că variabila `mutex` a fost blocată de către threadul care apelează funcția de deblocare.

Este momentul să analizăm rezultatele din coloanele 3 şi 4 ale tabelului 4.6. In ambele variante accesul la variabila `p` este exclusiv, fiind protejat de variabila `mutp`. Se observă că `p` final are valoarea corectă. De asemenea, indiferent de faptul că `ATRIBLUNG` este sau nu definită (ceea ce diferențiază cele două cazuri), se observă o mare regularitate în succesiunea la control a thread-urilor.

Intrebarea naturală care se pune este “ce se întamplă dacă mutex-ul este deja blocat de threadul curent?”. Răspunsul diferă de la platformă la platformă. Programul 4.2

prezintă o situație bizară, cititorul poate să-l testeze, dar să nu-l utilizeze în aplicații!:)

```
#include <synch.h>
#include <thread.h>
mutex_t mut;
thread_t t;
void* f(void* a) {
   mutex_lock(&mut);
   printf("lin 1\n");
   mutex_lock(&mut); // 1
   printf("lin 2\n");
   mutex_unlock(&mut);
   mutex_lock(&mut);
}
main() {
   mutex_init(&mut,USYNC_THREAD,NULL);
   thr_create(NULL,0,f,NULL,THR_NEW_lwp,&t);
   thr_join(t,NULL,NULL);
}
```

Programul 2.2 Un (contra)exemplu: sursa `dublaBlocareMutex.c`

Punctăm ca observație faptul că, pe Solaris, execuția acestui program produce impas în punctul `// 1`.

2.2.3.2 Operații cu variabile condiționale

Orice variabilă condițională aşteaptă un anumit eveniment. Ea are asociată o variabilă mutex şi un predicat. Predicatul conține condiția care trebuie să fie îndeplinită pentru a apărea evenimentul, iar variabila mutex asociată are rolul de a proteja acest predicat. Scenariul de aşteptare a evenimentului pentru care există variabila condițională este:

```
Blochează variabila mutex asociată
Câttimp (predicatul este fals)
   Aşteaptă la variabila condițională
Execută eventuale acțiuni
Deblochează variabila mutex
```

Este de remarcat faptul că pe durata aşteptării la variabila condițională, sistemul eliberează variabila mutex asociată. In momentul în care se semnalizează îndeplinirea condiției, înainte de ieşirea threadului din aşteptare, i se asociază din nou variabila mutex. Prin aceasta se permit în fapt două lucruri: (1) să se aştepte la condiție, (2) să se actualizeze predicatul şi să se semnalizeze apariția evenimentului.

Scenariul de semnalare - *notificare* - a apariției evenimentului este:

```
Blochează variabila mutex asociată
Fixează predicatul la true
Semnalizează apariția evenimentului
      la variabila condițională
      pentru a trezi thread-urile ce aşteaptă
Deblochează variabila mutex.
```

Inițializarea unei variabile conditionale se poate face static sau dinamic, astfel: