Laboratorul 6

Fire de execuție

1 Crearea firelor de execuție

Laboratoarele anterioare au discutat despre crearea unor procese noi cu ajutorul funcțiilor de tip `fork(2)` sau `execve(2)`. Procesele nou-create erau o copie fidelă a celui inițial, dar resuresele erau complet separate, fiind necesare diferite mecanisme de comunicație inter-proces pentru a colabora.

Pentru ușurarea comunicării și sporirea performanței, se pot folosi fire de execuție separate ale aceluiași proces. Acestea au avantajul că împart toate resursele și orice modificare făcută în spațiul procesului de un fir este instantaneu vizibilă tuturor celorlalte fără a apela la un mecanism exterior.

Dezavantajele apar o dată cu nevoia de a scrie și/sau citi concomitent aceeași zonă de memorie. În acest caz, concurența și drepturile de acces asupra resursei trebuie dictate de terți, îngreunând astfel procesul de execuție și de multe ori introducând defecte subtile în program.

Firele de execuție, denumite thread în literatura de specialitate, sunt implementate în mediile POSIX prin variabile de tip `pthread_t`. Pentru a crea un thread nou se folosește funcția `pthread_create(3)`

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

care inițializează pointer-ul `thread` cu noul fir de execuție lansat prin apelarea funcției `start_routine` cu argumentele oferite de `arg`. **Atenție**, spre deosebire de procesele create cu `fork(2)`, un thread nou pornește execuția de la o funcție dată.

La inițializare, se pot particulariza anumite detalii privind noul thread (ex. dimensiunea stivei, chestiuni de securitate etc.). Pe parcursul laboratorului vom folosi atributele implicit setate de sistemul de operare, semnalând aceasta prin folosirea valorii `NULL` pentru `attr`.

Un apel tipic pentru lansarea unui nou fir de execuție este

```
pthread_t thr;
if (pthread_create(&thr, NULL, hello, "world!")) {
        perror(NULL);
        return errno;
}
```

unde funcția `hello` trebuie să respecte prototipul `start_routine` de mai devreme

```
void *
hello(void *v)
{
        char *who = (char *)v;
        printf("Hello, %s!", who);
        return NULL;
}
```

Pentru a aștepta finalizarea execuției unui thread se folosește `pthread_join(3)`

```
int pthread_join(pthread_t thread, void **value_ptr);
```

care, diferit de `wait(2)`, așteaptă explicit firul de execuție din variabila `thread`. Dacă `value_ptr` nu este `NULL`, atunci `pthread_join` va pune la adresa indicată rezultatul funcției `start_routine`. Un apel tipic este

```
if (pthread_join(thr, &result)) {
        perror(NULL);
        return errno;
}
```

În mediile POSIX, funcționalitatea thread se găsește într-o bibliotecă separată numită `libpthread`. Astfel, la compilare este nevoie să specificăm explicit această legătură

```
$ cc hello.c -o hello -pthread
```

2 Sarcini de laborator

1. Scrieți un program care primește un șir de caractere la intrare, ale cărui caractere le copiază în ordine inversă și le salvează într-un șir separat. Operația de inversare va avea loc într-un thread separat. Rezultatul va fi obținut cu ajutorul funcției `pthread_join`. Exemplu

   ```
   $ ./strrev hello
   olleh
   ```

2. Scrieți un program care să calculeze produsul a două matrice date (de dimensiuni compatibile) unde fiecare element al matricei rezultate este calculat de către un thread distinct.

Reamintim că date

$$\boldsymbol{A} = (a_{ik})_{\substack{1 \le i \le m \\ 1 \le k \le p}} \in \mathbb{R}^{m \times p},$$

$$\boldsymbol{B} = (b_{kj})_{\substack{1 \le k \le p \\ 1 \le j \le n}} \in \mathbb{R}^{p \times n}$$

și

$$\boldsymbol{C} = (c_{ij})_{\substack{1 \le i \le m \\ 1 \le j \le n}} \in \mathbb{R}^{m \times n},$$

cu $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$, avem că pentru orice $i \in \{1, \ldots, m\}$ şi orice $j \in \{1, \ldots, n\}$,

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}.$$