Sisteme de operare

Laborator 8

Semestrul I 2025-2026

Laborator 8

- IPC System V
  - cozi de mesaje
- POSIX threads
  - mutex-uri

System V IPC

- cozi de mesaje, semafoare, memorie partajata
- toate aceste mecanisme folosesc un set comun de functii
- in kernel, toate structurile de date corespunzatoare sunt identificate printr-un ID (intreg nenegativ)
- pentru a folosi aceste mecanisme IPC e necesar sa stim acest ID
  - de ex: ca sa trimitem/primim mesaje intr-o/dintr-o coada de mesaje avem nevoie de ID-ul cozii
- in schimb, apelurile de creare a acestor mecanisme IPC necesita o cheie (o valoare de tip *key_t*) care va fi convertita de kernel intr-un ID

IPC rendez-vous

- pt. ca doua procese sa comunice prin mecanisme System V IPC trebuie sa poata identifica aceeasi structura IPC din kernel
- in acest sens, exista 3 posibilitati

1) IPC_PRIVATE
   - un proces (*server*) creeaza o noua structura IPC folosind o cheie de tip IPC_PRIVATE
   - stocheaza ID-ul returnat de kernel undeva accesibil altui proces (*client*) care participa la IPC

ex: in fisier sau partajat prin *fork*

2) serverul si clientul specifica o cheie intr-un header comun

Pb: daca exista in kernel o alta structura care foloseste aceeasi cheie, creearea noii structuri IPC esueaza

IPC rendez-vous (cont.)

3) serverul si clientul folosesc o cale comuna catre un fisier si un ID de proiect
   - le convertesc intr-o cheie cu ajutorul functiei *ftok*

- pentru a accesa o structura IPC existenta se foloseste cheia cu care a fost creata

- creare structura IPC noua
  - pt a evita folosirea unei structuri existente, apelurile de creeare a mecanismelor IPC folosesc IPC_CREAT | IPC_EXCL

- permisiuni structura IPC din kernel:
  - UID/GID proprietar
  - UID/GID creator
  - mod de acces
  - nr de secventa
  - cheie

Structuri IPC, pros & cons

- system-wide, fara reference count
  - ex: coada de mesaje cu mesaje in ea nu este stearsa din sistem cand procesul a terminat executia
  - e nevoie de apeluri sistem explicite (sau comenzi shell) pentru a o sterge

  Ex: *$ ipcs*

  $ ipcrm –{m|s|q} <id>
  - prin comparatie, *pipe*-urile dispar odata cu procesele care le utilizeaza
    - FIFO: ramane numele de fisier, dar datele sunt sterse
- necunoscute sistemului de fisiere
  - e nevoie de comenzi si apeluri sistem speciale pentru a lucra cu ele
- DAR, sunt reliable, au flow control, orientate pe inregistrari, pot fi procesate si in alta ordine decat FIFO

Cozi de mesaje

- create cu *msgget*

int msgget(key_t key, int flag);

ex: flag IPC_CREAT |IPC_EXCL | 0644

- atributele lor (stocate intr-o structura *msqid_ds*) pot fi manipulate cu *msgctl*

*int msgctl(int msgid, int cmd, struct msgid_ds *buf);*

Valori *cmd*: IPC_STAT, citeste structura *msgid_ds* asociata cozii

IPC_SET, seteaza UID/GID, *mode* si dimensiunea cozii (dimensiunea cozii poate fi crescuta doar de *root*)

IPC_RMID, sterge imediat coada din sistem (si mesajele din ea; alte procese care folosesc ulterior coada primesc EIDRM)

Obs: pt SET si RMID e nevoie fie de drepturi de root fie ca

UID creator = UID proprietar

Cozi de mesaje (cont.)

- mesajele se scriu/citesc in/din coada cu *msgsnd*/*msgrcv*
- apelurile folosesc o structura definita de utilizator

```
struct msg {
        long mtype;             // tip mesaj, valoare pozitiva
        char mtext[BUFSIZE];    // date mesaj
}
```

*int msgsnd(int msgid, const void *ptr, size_t nbytes, int flag);*

*int msgrcv(int msgid, void *ptr, size_t nbytes, long type, int flag);*

- pt. type = 0 la *msgrcv* se citeste primul mesaj din coada (politica FIFO), altfel se citeste primul mesaj din coada cu acel tip

POSIX threads

- creare

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

  - porneste un nou thread in proces apeland *start_routine* cu *arg* ca parametru
  - *attr* setat cu *pthread_attr_init*, NULL indica atributele implicite

- terminare
  - normal, prin iesirea din *start_routine*
  - cu *pthread_exit*
  - cu *pthread_cancel*
  - *exit* din orice thread al procesului
  - iesire din *main*

POSIX threads (cont.)

- tipuri de thread-uri (atribut)
  - detached
    - la terminarea thread-ului resursele sale sunt eliberate fara sa fie nevoie ca alt thread sa apeleze *pthread_join*, INDIFERENT de cauza terminarii sale
    - se face cu *pthread_detach*

      int pthread_detach(pthread_t thread);
  - joinable
    - eliberarea resurselor thread-ului se face doar cand un alt thread cheama *pthread_join*
- pthread_exit

  *void pthread_exit(void *retval);*
  - termina thread-ul apelant
  - paseaza *retval* catre *pthread_join*
  - resursele partajate ale procesului NU sunt eliberate !

POSIX threads (cont.)

- pthread_join

  *int pthread_join(pthread_t thread, void **retval);*

  - asteapta terminarea thread-ului *thread*
  - daca *thread* a terminat deja, apelul e neblocant
  - in *retval* se regaseste starea de terminare a thread-ului *thread*
  - daca *thread* a fost anulat, **retval* = PTHREAD_CANCELED
  - *thread* trebuie sa fie joinable

POSIX threads (cont.)

- pthread_cancel

  int pthread_cancel(pthread_t thread);

  - trimite o cerere de anularea a executiei lui *thread*
  - daca posibilitatea de anulare e dezactivata, se asteapta pana la activarea ei
  - altfel, anularea poate fi
    - *asincrona* – *thread* poate fi terminat imediat (dar nu e garantat)
    - *intarziata* – pana cand *thread* ajunge intr-un *cancellation point*
  - anularea determina apelul handlerelor de clean-up + destructori de date specifice *thread* + terminare *thread* (pthread_exit)

  Obs: *pthread_cancel* doar trimite o cerere de anulare intr-o coada, nu asteapta anularea efectiva si consecintele ei

Ex: producator-consumator POSIX

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

#define ITEMS           8
#define BUFFER_SIZE     4
int buffer[BUFFER_SIZE];

void *producer(void*);
void *consumer(void*);

int spaces, items, tail;
pthread_cond_t space, item;
pthread_mutex_t buffer_mutex;

int main()
{
        pthread_t producer_thread, consumer_thread;
        void *thread_return;
        int result;

        spaces = BUFFER_SIZE;
        items = 0;
        tail = 0;

        pthread_mutex_init(&buffer_mutex, NULL);
        pthread_cond_init(&space, NULL);
        pthread_cond_init(&item, NULL);

        if(pthread_create(&producer_thread, NULL, producer, NULL) ||
           pthread_create(&consumer_thread, NULL, consumer, NULL))
                exit(1);

        if(pthread_join(producer_thread, &thread_return))
                exit(1);
        else
                printf("producer returns with %d\n", (int)thread_return);

        if(pthread_join(consumer_thread, &thread_return))
                exit(1);
        else
                printf("consumer returns with %d\n", (int)thread_return);
        exit(0);
}

```

POSIX Mutex Locks

- crearea si initializarea lock-ului mutex

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- obtinerea si eliberarea lock-ului

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```