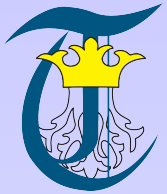

Universitatea *Transilvania* Brașov
Facultatea de Inginerie Electrică și Știința Calculatoarelor
Departamentul de Electronică și Calculatoare

Sisteme de operare

– curs 5 –

2012.03.27

ş.l. dr. ing. Kertész Csaba-Zoltán

2.3. Probleme IPC clasice

- probleme elementare ale sistemelor multiprogramate cu comunicație între procese
- mai multe probleme propuse și rezolvate de-a lungul timpului
- ori de câte ori se propune o nouă primitivă de sincronizare se testează funcționarea pe aceste probleme

2.3.1. Problema cinei filozofilor

- propus de Dijkstra în 1965
- 5 filozofi sunt așezați la o masă rotundă
- fiecare filozof are în față o farfurie cu spaghetti
- pentru a mânca spaghetti un filozof are nevoie de două furculițe
- între două farfurii există o furculiță (5 în total)
- viața unui filozof constă din perioade în care gândește și perioade când mănâncă

Problema cinei filozofilor

- când un filozof devine flămând încearcă să ia furculițele din stânga și din dreapta, dacă reușește atunci va mânca un anumit timp după care pune furculițele la locul lor

```
#define N 5

void philosopher(int i)
{
   while (TRUE) {
      think();
      take_fork(i); //block until fork present
      take_fork((i+1)%N);
      eat();
      put_fork(i);
      put_fork((i+1)%N);
   }
}
```

Problema cinei filozofilor

- probleme cu această soluție:
  - dacă toți ridică furculița din stânga simultan:

deadlock

- putem modifica programul astfel încât după preluarea furculiței din stânga, să se verifice dacă furculița din dreapta este disponibilă, iar dacă nu atunci se pune înapoi cea din stânga
  - dacă toți ridică furculița din stânga simultan, vor vedea furculița din dreapta indisponibil, vor pune înapoi furculița din stânga și se reia din început

starvation

Rezolvarea problemei cinei filozofilor

- putem introduce un timp aleator de așteptare între ridicarea furculițelor, dar nu putem baza pe evoluția aleatoare a programului
- o soluție deterministă fără deadlock și fără starvation este obținută prin protejarea celor cinci instrucțiuni de după think cu un semafor binar:
  - înainte de a lua furculițele filozoful execută **down**, iar după ce pune înapoi furculițele execută **up**
  - performanța nesatisfăcătoare (un singur filozof mănâncă la un moment de timp)

Implementarea cinei filozofilor

```
#define N 5
#define LEFT ((i-1)%N)
#define RIGHT ((i+1)%N)
#define THINKING 0
#define HUNGRY 1
#define EATING 2

typedef int semaphore;
int state[N];
semaphore mutex=1;
semaphore s[N];

void philosopher(int i)
{
   while (TRUE) {
      think();
      take_forks(i);
      eat();;
      put_forks(i);
   }
}
```

```
void take_forks(int i)
{
   down(&mutex);
   state[i] = HUNGRY;
   test(i);
   up(&mutex);
   down(&s[i]);
}
void put_forks(int i)
{
   down(&mutex);
   state[i] = THINKING;
   test(LEFT);
   test(RIGHT);
   up(&mutex);
}
void test(int i)
{
   if (state[i] == HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING) {
      state[i] = EATING;
      up(&s[i]);
   }
}
```

Problema scriitor/cititor

- propus de Courtois în 1971
- modelează accesul la o bază de date
- avem o bază de date mare (de ex. sistemul de rezervare a biletelor de avion) și mai multe procese care doresc să scrie sau să citească în/din baza de date
- se acceptă posibilitate ca mai multe procese să citească deodată, dar dacă un proces accesează pentru scriere nici un alt proces nu poate accesa nici pentru scriere, nici pentru citire

Implementarea scriitor/cititor

```
typedef int semaphore;
semaphore mutex=1;
semaphore db=1;
int rc=0;

void reader(void)
{
   while (TRUE) {
      down(&mutex);
      rc = rc+1;
      if (rc == 1) down(&db);
      up(&mutex);
      read_database();
      down(&mutex);
      rc = rc-1;
      if (rc==0) up(&db);
      up(&mutex);
      use_data();
   }
}
```

```
void writer(void)
{
   while (TRUE) {
      think_update();
      down(&db);
      write_database();
      up(&db);
   }
}
```

Problema sleeping barber

- într-o frizerie există 1 frizer, 1 scaun pentru frizer și n scaune pentru clienți care așteaptă
- când nu sunt clienți care așteaptă frizerul stă pe scaunul lui și doarme
- când apare un client, aceasta va trebuie să trezească frizerul
- dacă mai apare un client în timp ce frizerul tunde un client, aceasta va trebui să aștepte pe un scaun (dacă este liber unul) sa va părăsi frizeria

Implementarea sleeping barber

```
#define CHAIRS 5

typedef int semaphore;
semaphore mutex=1;
semaphore customers=0;
semaphore barbers=0;
int waiting=0;

void barber(void)
{
   while (TRUE) {
      down(&customers);
      down(&mutex);
      waiting = waiting-1;
      up(&barbers)
      up(&mutex);
      cut_hair();
   }
}
```

```
void customer(void)
{
   down(&mutex);
   if (waiting < CHAIRS) {
      waiting = waiting + 1;
      up(&customers);
      up(&mutex);
      down(&barbers);
      get_haircut();
   }
   else {
      up(&mutex);
   }
}
```

2.4. Thread-uri

- un thread (fir de execuție) este o unitate de execuție de bază a unui CPU și constă din PC, un set de registre și un spațiu pentru stivă
  - se mai numește și lightweight process (LWP)
- secțiunea de cod, cea de date și resursele SO (fișiere deschise, semnale) vor fi partajate între thread-urile care aparțin unui proces
- partajarea resurselor determină timpi de comutare mult mai mici decât a proceselor

Modele de threaduri

- există mai multe modele de threaduri în funcție cum sunt mapate între user space și kernel space
- threadurile la user level sunt tratate de procesul cărora le aparțin fără intervenția kernelului
- threadurile la kernel level sunt tratate de către kernel similar cu procesele
- există relații între threaduri user level și kernel level:
  - multe la unu, unu la unu, multe la multe

Thread-uri user level

- unele sisteme folosesc user-level threads: acest tip de thread-uri sunt implementate folosind biblioteci multithreading
  - implementarea se face fără a se folosi apeluri de sistem deci la comutarea contextului între threaduri nu este necesar intervenția SO
  - comutarea se realizează independent de SO deci foarte rapid
  - un thread user-level poate bloca tot procesul
  - siguranța între threduri este lăsat pe seama programatorului

Thread-uri kernel level

- toate SO moderne oferă suport pentru threaduri kernel level
- toate threadurile sunt tratate la nivelul planificatorului de procese
- există siguranță ridicată între threaduri
- impune un overhead la crearea și schimbarea între threaduri
- Linux: pthreads (POSIX threads): bibliotecă de funcții standard pentru manipularea threadurilor

Sincronizarea threadurilor

- și în cazul thread-urilor pot apare condiții de concurență care se pot rezolva ca în cazul proceselor
- pentru pthreads există extensii care permit folosirea semafoarelor și a mutecșilor
- există suport și pentru blocarea accesului la zonele de memorie partajate

Procese vs. thread-uri

- procesele pot opera independent: există protecție între procese
- fiecare proces are propriul PC, SP și spațiu de adrese: utilizează multe resurse

- thread-urile nu sunt complet independente: nu există protecție între thread-uri
- partajează resursele: utilizează mai puține resurse