Operativni sistemi 2

Vežbe 7
IPC primeri

Lista predmeta:
IR smer - 13e113os2@lists.etf.rs
SI Smer - 13s113os2@lists.etf.rs

Zadatak

Za operativni sistem Linux, napisati program na programskom jeziku C koji treba da iz N paralelnih procesa ispiše redom brojeve od 0 do M*N-1, tako da apsolutna razlika uzastopno ispisanih brojeva iz jednog procesa bude N. Dakle, jedan proces treba da ispiše brojeve 0, N, 2N,...; drugi proces ispisuje brojeve 1, N+1, 2N+1, itd. Za potrebe komunikacije i sinhronizacije između procesa dozvoljeno je koristiti isključivo semafore. Svi alocirani resursi moraju biti oslobođeni.

```
#include<stdio.h>
#include<sys/sem.h>
#include<stdlib.h>
#include <unistd.h>

#define N 5
#define M 5

int main(){
     int i, j, sem_id;
     sem_id = semget((key_t)123, N, 0666 | IPC_CREAT);
     struct sembuf sem;
     for(i=0;i<N;i++){
          if(fork()==0){
               for(j=0;j<M;j++){
                    //sem[i].wait();
                    sem.sem_num = i;
                    sem.sem_op = -1;
                    sem.sem_flg = SEM_UNDO;
                    semop(sem_id, &sem, 1);

                    printf("%d\n",i+j*N);
```

```
                    //sem[(i+1) % N].signal()
                    sem.sem_num = (i+1)%N;
                    sem.sem_op = 1;
                    sem.sem_flg = SEM_UNDO;
                    semop(sem_id, &sem, 1);
                }
                if (i==N-1) {
                    semctl(sem_id, 0, IPC_RMID);
                }
                exit(0);
            }
        }
        //sem[0].signal();
        sem.sem_num = 0;
        sem.sem_op = 1;
        sem.sem_flg = SEM_UNDO;
        semop(sem_id, &sem, 1);
}
```

Zadatak

Neki programer je rešavao sledeći problem. Potrebno je izračunati prvih N elemenata nekog reda koji je zadat sa: $a_0 = 5$ i $a_i = f(a_{i-1})$. Funkcija f u svom radu koristi samo prosleđeni parametar i lokalne promenljive. Rešenje koje je ponudio programer je sledeće:

```
void main(){
     int zeton = 0;
     int cevi[N+1][2];
     int i;
     int a[N];
     a[0] = 5;
     for (i=0;i<N;i++) pipe(cevi[i]);
     for (i=1;i<N;i++){
          if (fork() == 0){
               read(cevi[i][IN], &zeton, sizeof(int));
               a[i] = f(a[i-1]);
               printf("%d\n",a[i]);
               write(cevi[i+1][OUT], &zeton, sizeof(int));
               exit(0);
          }
     }
     write(cevi[1][OUT], &zeton, sizeof(int));
}
```

Dato rešenje ima više nedostataka koji ne utiču na ispravnost ispisa i kao takve ih ne treba uzimati u obzir pri obrazlaganju odgovora na sledeće pitanje. Da li će program ispisati korektne vrednosti (a_1, a_2,...,a_{N-1})? Odgovor kratko i precizno obrazložiti.

Odgovor:
Ne. Programer je ispravno sinhronizovao računanja, tako da se element a_i uvek računa tek pošto je izračunat element a_{i-1}, ali je previdio da se te vrednosti izračunavaju u različitim adresnim prostorima, te da će pri izračunavanju elementa a_i biti korišćena neka slučajna vrednost iz niza a umesto prethodno izračunate vrednosti a_{i-1}.

Napomena: Šta je potrebno izmeniti kako bi program ispisivao korektne vrednosti?

Zadatak

Implementirati funkciju `void mergeSort(int n, int m, int niz[])` koja sortira niz od *n* elemenata u rastućem redosledu na sledeći način. Funkcija uzima po *m* elemenata niza i sortira ih u konkurentnim procesima pomoću algoritma datog funkcijom `int sort(int n, int a[])`, a zatim tako sortirane podnizove spaja (*engl.* merge sort). Na raspolaganju su sledeće funkcije za rad sa imenovanim cevovodima (*engl.* Named Pipes):

```
// Upisuje elemente niza a, duzine n u cevovod pipeName
void sendToPipe(char* pipeName, int n, int a[]);
// Čita niz elemenata a iz cevovoda pipeName i njihov broj broj smešta u *n
void readFromPipe(char* pipeName, int* n, int a[]);
// Vraća ime sledećeg cevovoda koji može da se koristi
char* getNextPipeName();
// Obezbeđuje da funkcija getNextPipeName vraća imena cevovoda od početka
void resetPipeNames();
```

```
int mergeSort(int n, int m, int niz[]){
  // Creating child processes
  int numOfChildren = 0;
  for (int i = 0; i < n; i += m){
      numOfChildren++;
      char* childPipeName = getNextPipeName();
      int len = (i + m < n) ? m : n - i;
      int pid = fork();
      if (pid == 0){        //child
        sort(len, niz + i);
        sendToPipe(childPipeName, len, niz + i);
        exit(0);
      }
  }

  // Collecting sorted subarrays
  int** b = new int [numOfChildren];
  int* cntrs = new int [numOfChildren];
  int* sizes = new int [numOfChildren];

  resetPipeNames();
```

```
for(int c = 0; c < numOfChildren; ++c) {
   b[c] = new int[m];
   cntrs[c] = 0;
   readFromPipe(getNextPipeName(), &sizes[c], b[c]);
 }
 // Merging
 int cntr = 0;
 while (cntr < n) {
   int arrayInd = -1, min = -1;
   for (int i = 0; i < numOfChildren; ++i)
     if (cntrs[i] < sizes[i] && (arrayInd == -1 || min > b[i][cntrs[i]])){
       min = b[i][cntrs[i]];
       arrayInd = i;
     }
     niz[cntr++] = min;
     cntrs[arrayInd]++;
 }
```

```
  // Release temporary space
  for (int i = 0; i < numOfChildren; i++)
    delete b[i];
  delete b;
  delete cntrs;
  delete sizes;
  return 0;
}
```

Zadatak – Januar 2012

Na jeziku C/C++, koristeći mehanizam prosleđivanja poruka operativnog sistema Linux, dati rešenje problema filozofa koji večeraju (*dining philosophers*), pri čemu je data funkcija `philosopher` koja kao argument prima jedinstveni broj (identifikator) filozofa. Svaki filozof pri slanju zahteva identifikuje se ovim brojem. Takođe, navedena je struktura poruka koje se razmenjuju, kao i značenje vrednosti svakog polja.

Napisati implementaciju centralnog procesa koji na početku nad funkcijom `philosopher` kreira potreban broj filozofa predstavljenih procesima, a zatim u vidu konobara (*waiter*) komunicira sa filozofima i obezbeđuje njihovu sinhronizaciju. Nije potrebno proveravati uspešnost izvršavanja operacije nad sandučićima (*message queue*).

```
#define MESSAGE_Q_KEY 1

struct requestMsg {
 long mtype;  // tip poruke – identifikator filozofa
 char msg[1]; // operacija - vrednost: 1 - request forks, 2 - release forks
};
void philosopher(int id) {
 int requestMsgQueueId = msgget(MESSAGE_Q_KEY, IPC_CREAT | 0666);
 int responseMsgQueueId = msgget(MESSAGE_Q_KEY + 1, IPC_CREAT | 0666);
 size_t len = sizeof(char);

 while (1) {
  //request forks
  struct requestMsg msg;
  msg.mtype = id;
  msg.msg[0] = (char) 1;
  msgsnd(requestMsgQueueId, &msg, len, 0);
  msgrcv(responseMsgQueueId, &msg, len, id, 0);

  //eat
  sleep(1);

  //release forks
  msg.mtype = id;
  msg.msg[0] = (char) 2;
  msgsnd(requestMsgQueueId, &msg, len, 0);

  //think
  sleep(1);
 }
}
```

```
void acquireForksForPhilosopher(int *forks[N], int id, int msgQueueId) {
 forks[id] = 0;
 forks[(id + 1) % N] = 0;
 struct requestMsg msg_buf;
 msg_buf.mtype = id;
 msg_buf.msg[0] = 1;
 msgsnd(msgQueueId, &msg_buf, sizeof(char), 0);
}

int main() {
 int requestMsgQueueId = msgget(MESSAGE_Q_KEY, IPC_CREAT | 0666);
 int responseMsgQueueId = msgget(MESSAGE_Q_KEY + 1, IPC_CREAT | 0666);
 size_t len = sizeof(char);

 //philosophers
 int id;
 for (id = 1; id <= N; id++) { // rezervisana vrednost za mtype=0
  if (fork() == 0) {
   philosopher(id);
  }
 }
```

```
//waiter
 int *forks[N], *requests[N];
 for (id = 0; id < N; id++) {
  forks[id] = 1;
  requests[id] = 0;
 }

 struct requestMsg msg_buf;
 while (1) {
  int r = msgrcv(requestMsgQueueId, &msg_buf, len, 0, 0);
  id = (int) msg_buf.mtype - 1;

  if (msg_buf.msg[0] == 1) { //request forks
   if (forks[id] && forks[(id + 1) % N]) {
    acquireForksForPhilosopher(forks, id + 1, responseMsgQueueId);
   } else {
    requests[id] = 1;
   }
  } else { //Release forks
   forks[id] = 1;
   forks[(id + 1) % N] = 1;
```

```
 // check neighbors
    int leftNeighbour = id ? id - 1 : N - 1;
    int rightNeighbour = (id + 1) % N;
    if (requests[rightNeighbour] && forks[(rightNeighbour + 1) %
N]) {
     requests[rightNeighbour] = 0;
     acquireForksForPhilosopher(forks, rightNeighbour,
       responseMsgQueueId);
    }
    if (requests[leftNeighbour] && forks[leftNeighbour]) {
     requests[leftNeighbour] = 0;
     acquireForksForPhilosopher(forks, leftNeighbour,
       responseMsgQueueId);
    }
   }
  }
}
```

Zadatak

Na jeziku C/C++, koristeći mehanizam deljene memorije i semafora kod operativnog sistema Linux, napisati program koji pravi procese koji međusobno komuniciraju preko razmene poruka (nije dozvoljeno koristiti mehanizam prosleđivanja poruka putem poštanskog sandučeta kod operativnog sistema Linux). Broj procesa koje je potrebno pokrenuti je dat kao konstanta `N`. Vrednost ključa za dohvatanje odgovarajućeg segmenata deljene memorije je data kao konstanta `keyMem`. Vrednost ključa za dohvatanje odgovarajućeg semafora je data kao konstanta `keySem`. Svaki kreirani proces ima jedinstveni identifikator procesa:

```
typedef unsigned Id;
```

Programi treba da komuniciraju putem poruka koji su tipa:

```
struct Msg {
Id receiver;
...
};
```

gde je polje `receiver` identifikator procesa koji treba da primi poruku. Broj poruka koji svaki kreirani proces treba da napravi i pošalje je dat kao konstanta `NUM`.

Data je funkcija za pravljenje poruke:

```
void getMsg(struct Msg *msg, Id id);
```

gde parametar `msg` pokazuje na memorijsku lokaciju gde poruku treba smestiti, dok je parameter `id` identifikator procesa koji pravi poruku.

Procesi treba da šalju poruke preko bafera u deljenoj memoriji u koji može da se smesti `SIZE` poruka. Svaki proces prvo proverava da li ima mesta u baferu i ako ima, stavlja jednu poruku. Zatim proces treba da proveri da li se na početku bafera nalaze poruke za njega i ako se nalaze, da ih pročita. Taj posao svaki proces treba da ponavlja dok ne pošalje sve poruke. Proces treba da završi svoje izvršavanje kad primi sve poruke i kad oslobodi sve sistemske resurse. Nije potrebno proveravati uspešnost sistemskih poziva za deljenu memoriju i semafore. Ako je neki deo koda indentičan kao kod dat na vežbama, nije ga potrebno pisati, već samo navesti odakle je preuzet.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <unistd.h>
#define N ...
#define keyMem ...
#define keySem ...
#define SIZE ...
#define NUM ...
#define COUNT_MSG (NUM * N)

union semun {
        int val; /* Value for SETVAL */
        struct semid_ds *buf; /* Buffer for IPC_STAT, IPC_SET */
        unsigned short *array; /* Array for GETALL, SETALL */
        struct seminfo *__buf; /* Buffer for IPC_INFO (Linux-specific) */
};

typedef unsigned Id;
struct Msg {
        Id receiver;
        . . .
};

void getMsg(struct Msg *msg, Id id);

struct Buffer {
        int head, tail, count;
        struct Msg data[SIZE];
        int sent_msg;
};
```

```
static void init_sem(int sem_id, int value)
{
      union semun arg;
      arg.val = value;
      semctl(sem_id, 0, SETVAL, arg);
}

static void sem_wait(int sem_id)
{
      struct sembuf sem_b;
      sem_b.sem_num = 0;
      sem_b.sem_op = -1;
      sem_b.sem_flg = SEM_UNDO;
      semop(sem_id, &sem_b, 1);
}

static void sem_signal(int sem_id)
{
      struct sembuf sem_b;
      sem_b.sem_num = 0;
      sem_b.sem_op = 1;
      sem_b.sem_flg = SEM_UNDO;
      semop(sem_id, &sem_b, 1);
}
```

```
static void inc(int *pvalue)
{
	*pvalue = (*pvalue + 1) % SIZE;
}

static void read_msg(struct Buffer *buffer, Id process_id, int
sem_id)
{
	struct Msg msg;
	int head;
	sem_wait(sem_id);
	whiIe (1) {
		head = buffer->head;
		if (buffer->count > 0 && buffer->data[head].receiver
== process_id) {
			msg = buffer->data[head];
			inc(&buffer->head);
			buffer->count--;
			buffer->sent_msg--;
			// Save msg somewhere
		} else break;
	}
	sem_signal(sem_id);
}
```

```
int main()
{
     int shm_buffer_size;
     Id process_id = 0;
     shm_buffer_size = sizeof(struct Buffer);
     int shmid = shmget(keyMem, shm_buffer_size, IPC_CREAT
| S_IRUSR | S_IWUSR);
     struct Buffer *buffer = (struct Buffer*)shmat(shmid,
0, 0);
     int sem_id = semget(keySem, 0, 066 | IPC_CREAT);
     buffer->head = 0;
     buffer->tail = 0;
     buffer->count = 0;
     buffer->sent_msg = COUNT_MSG;
     init_sem(sem_id, 0);
     for (int i = 1; i < N; i++)
     {
          if (fork() != 0){
               process_id = i;
               break;
          }
     }
```

```
    int i = 0;
    while (i < NUM) {
        struct Msg msg;
        sem_wait(sem_id);
        if (buffer->count < SIZE) {
            getMsg(&msg, process_id);
            buffer->data[buffer->tail] = msg;
            buffer->count++;
            inc(&buffer->tail);
            i++;
        }
        sem_signal(sem_id);
        read_msg(buffer, process_id, sem_id);
    }
    while (buffer->sent_msg > 0) {
    read_msg(buffer, process_id, sem_id);
    }
    shmdt(buffer);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```