1. Задание

Проект : thr1:

Реализация на взаимно изключване съгласно POSIX1.c или друга реализация (Solaris, Windows).

int mutex_init(mutex_t *mp, int type, void *arg)

int mutex_lock(mutex_t *mp)

int mutex_unlock(mutex_t *mp)

int mutex_trylock(mutex_t *mp)

int mutex_destroy(mutex_t *mp)

2. Постановка на задачата и предлагано решение.

2.1. Описание на задачата.

Въпреки че физическите и логическите ресурси могат да бъдат разделяни, обикновено във всеки момент от времето те са достъпни само за един процес. Такива ресурси се наричат критични. Ако няколко процеса искат да използват критичен ресурс, те трябва да съгласуват действията си във времето така, че в даден момент ресурсът да бъде използван само от един процес - останалите процеси трябва да чакат, докато се освободи ресурсът. Това е същността на понятието взаимно изключване на процеси.

Коректното решаване на проблема на взаимното изключване трябва да отговаря на следните критерии:

1. Само един процес може да използва ресурса в даден момент.

2. Ако няколко процеса едновременно желаят ресурса, той трябва да бъде предоставен на един от тях в крайно време.

3. Ако процес получи ресурс, той трябва да го освободи в крайно време.

Има множество различни методи за синхронизация, при които възникват различни проблеми, например мъртва хватка, безкрайно отлагане, блокиране на процеси и др.

Един от методите са семафорите., които са въведени от Дейкстра. Те представляват цели променливи, върху които се извършват операции P и V.

1. P(s): while s = 0 do skip; s = s – 1;

2. V(s): s = s + 1;

Извиквайки P(s), процесът се блокира при стойност на семафора s = 0, докато друг процес не го освободи чрез V(s).

Семафора има инициализация от вида: s = const.

Мутексите представляват вид семафори, които са двоични и приемат стойност само 1 и

0;

Решаването на проблема с критичните секции чрез мутекси става по следния начин:

P(s);

критична секция

V(s);

Конкретно в стандарта POSIX1.c са реализирани следния набор от функции за работа с мутекси:

int mutex_init(mutex_t *mp, int type, void *arg)

int mutex_lock(mutex_t *mp)

int mutex_unlock(mutex_t *mp)

int mutex_trylock(mutex_t *mp)

int mutex_destroy(mutex_t *mp)

Мутексите предотвратяват достъпа до общи данни едновременно от множество

процеси.

След успешно заключване на мутекса чрез *mutex_lock(),* всеки следващ процес, който се опита да го заключи, да бъде блокиран докато първият не освободи мутекса чрез *mutex_unlock(
).*

Множество потоци в един процес, или множество процеси могат да използват общи мутекси. Мутексите могат да синхронизират различни процеси, ако им е заделена памет и е споделена между процесите чрез изобразяване в памет(*memory-map*).

Мутексите трябва да се инициализират чрез *mutex_init().* Мутексът mp се инициализира в зависимост от типа сочен от *t
ype
*. След успешна инициализация мутексът е готов и отключен. Параметърът *arg*
не се използва понастоящем. Параметърът *type*
може да бъде:

USYNC_THREAD: Мутекса се използва за синхронизиране на потоци в един процес.

USYNC_PROCESS: Мутекса се използва за синхронизиране на потоци в различни процеси. За обекта трябва да бъде заделена обща памет, или чрез споделена памет(*shared memory*
), или чрез изобразен в памет файл(*memory-map*
).

Един мутекс не трябва да бъде инициализиран от няколко потока, а само от един.

Критична секция от кода се намира между извиквания на *mutex_lock()* и *mutex_unlock().*

Само един процес може да има достъп до тази секция. Процес, който извика *mutex_lock()* или получава достъп, или се блокира до освобождаване на мутекса. Не трябва да се допуска текущия собственик на мутекса, който го е заключил, да се опита да го заключи отново, тъй като това ще доведе до мъртва хватка.

Функцията *mutex_trylock()* е почти същата като *mutex_lock()*, но ако мутекса е вече заключен, процеса не се блокира, а фунцкията веднага приключва с резултат EBUSY и процесът продължава.

mutex_unlock() се извиква от текущия собственик на мутекса за да го отключи. Мутекса трябва да е в заключено състояние и да не е с друг собственик. Всички блокирани процеси се продължават, и някой от тях заключва мутекса. Останалите се блокират отново.

mutex_destroy() унищожава мутекса и той става неинициализиран.

2.2. Предлагано решение

Мутексите са реализирани чрез използване на вътрешен брояч, който следи блокираните процеси. При заключване на мутекса, този брояч се увеличава с единица. Ако брояча е нула, първото извикване на *mutex_lock()* го заключва. Следващите извиквания увеличават брояча и се блокират. Извикването на *mutex_unlock()* нулира брояча и деблокира процесите.

За блокиране и освобождаване на процесите са използвани семафорни операции(semget(), semop(), semctl()) реализирани в ядрото на операционната система.

3. Описание на алгоритмите

За работа с мутексите са реализирани пет функции съгласно стандарта POSIX:

int mutex_init(mutex_t *mp, int type, void *arg)

int mutex_lock(mutex_t *mp)

int mutex_unlock(mutex_t *mp)

int mutex_trylock(mutex_t *mp)

int mutex_destroy(mutex_t *mp)

Мутекса представлява следната структура:

typedef struct _mutex {

unsigned int semid;

int count;

unsigned int semops;

} mutex_t;

където *semid* е идентификатор на семафора използван за блокиращите операции, *count* е броя на процесите подали заявка за мутекса,
semops
е брояч на операциите извършени върху семафора.

За създаване на мутекс се дефинира променлива от типа *mutex_t*. Ако мутекса се използва за синхронизация между процеси, то тази променлива трябва да бъде в споделена памет между процесите.

След това се извиква функцията *mutex_init(),* която проверява валидността на параметрите и ако те са валидни създава семафора чрез
semget()
и инициализира
semid
. В противен случай се връща съобщение за грешка.

При извикване на *mutex_lock()* се изпълняват следните стъпки:

1. Проверява се стойността на count и се увеличава с 1.

2. Ако старата стойност е била 0, се излиза от функцията, след което мутекса вече е заключен.

3. Ако стойността е била различна от нула, се изпълнява операция за намаляване на семафора с 1 чрез *semop()* Тъй като първоначално той е със стойност 0, тя става -1 и това води до блокиране на процеса. Това продължава докато семафора не бъде увеличен с необходимата стойност и процеса се разблокира.

4. Преминава се към стъпка 1.

Тези стъпки се изпълняват за всички процеси извикващи *mutex_lock()*. При това стойността на семафора намалява с 1 за всеки процес. Блокирането продължава докато стойността не се увеличи така че да стане равна или по-голяма от 0, или докато се унищожи семафора.

Увеличаването на семафора става във функцията *mutex_unlock(),* която се извиква от текущия притежател на мутекса за да го освободи.

Тя се изпълнява на следните стъпки:

1. Стойността на count се присвоява на *n*.

2. Отново се проверява стойността на *count*.

3. Ако стойностите от т.1 и т.2 съвпадат се нулира *count* и се продължава изпълнението. Ако не съвпадат се преминава към т.1

4. Проверява се дали *n* е по-голяма от 1 (дали има чакащи процеси).

5. Ако е по-голяма от 1 се изпълнява операция над семафора чрез *semop()* за увеличаването му с
n
- 1. Това води до разблокиране на всички чакащи процеси.

След разблокирането на процесите те започват отново да изпълняват цикъла в *mutex_lock( )*.
Един от тях заключва мутекса, излиза от цикъла и започва да се изпълнява, а останалите се блокират отново.

Функцията *mutex_trylock()* има действие подобно на *mutex_lock()*, но е по-опростена. Проверява се стойността на
count
и ако тя е нула, се увеличава с 1, при което мутекса се заключва. Ако не е нула, се връща резултат, че мутекса е зает. Ефекта е, че функцията не блокира процеса, ако мутекса е заключен.

Функцията *mutex_destroy()* унищожава семафора чрез *semctl().* При това се разблокират всички блокирани процеси.

4. Описание на програмите

4.1. thrdemo.cpp

Програмата демонстрира използването на мутекси за синхронизация между различни потоци в един процес.

Използва се един мутекс, който разрешава на всеки поток последователен достъп до кода във функцията *change_global_data()*. В нея се използва обща променлива, която се увеличава с 1 при всяко извикване на функцията.

Тялото на функцията има вида:

```
mutex_lock(&Global_mutex);

Global_data++;

sleep(1);
```

mutex_unlock(&Global_mutex);

В него се заключва мутекса, променят се данните, симулира се извършването на някаква друга работа, и мутекса се отключва.

В тялото на основната програма се инициализира мутекса и в цикъл се създават отделните потоци. След като се изчака приключването на всеки от тях, мутекса се унищожава.

Този вид синхронизация успешно защитава данните, но също така възпрепятства паралелното изпълнение.

4.2. procdemo.cpp

Програмата демонстрира използването на мутекси за синхронизация между потоци в различни процеси.

За целта се използва структура, в която се намират мутекса и общите данни.

Един от процесите я инициализира и я записва във файл, който после се изобразява в паметта на всички процеси.

В последствие други независими процеси, могат да стартират програмата (едновременно или не), и да споделят данните.

Ако програмата се извика с параметър ‘1’, тя нямалява общата променлива, а ако се

извика с ‘0’ я увеличава. За целта има две функции: *add_interprocess_data()* създава потоци, които увеличават;
subtract_interprocess_data()
, създава потоци, които намаляват. Телата на двете функции са аналогични на тази в програмата
thrdemo.cpp.

В тялото на основната програма се създава гореописаната структура, след което се създава файл със същия размер. След това, в зависимост от параметъра, се извиква една от горните функции. Тя изобразява файла в паметта. По този начин се осигурява комуникацята между процесите. Функцията създава определен брой потоци и изчаква тяхното приключване.

Файла и мутекса се създават само ако програмата се извика с параметър ‘0’, затова е необходимо да се извика поне един процес с параметър ‘0’.

За демонстрация, програмата се извиква със следния команден ред:

“procdemo 0 & procdemo 1”

5. Използвана литература

1. Николов Л., Системно програмиране. Сиела, София, 2001

2. Николов Л., Операционни системи. Сиела, София, 2001

3. UNIX man pages

6. Листинг на програмите

6.1. mutexes.h

```
#include

#include

#include

#include

#include

#include

#include

#define USYNC_THREAD  0   //За синхр. на потоци в 1 процес

#define USYNC_PROCESS 1   //За синхр. между процеси

//Структура на мутекса:
```

```
//Използва се семафор за реализиране на блокирането

typedef struct _mutex {

unsigned int semid;    //Идентификатор на семафора

int count;             //Брой блокирани потоци/процеси

unsigned int semops;   //Брояч за семафорни операции

} mutex_t;

//Използва се във функциите на мутекса за извличане на стойност и увеличаването и

int fetch_and_add(int *n, int i)

{

int t;

t = atomic_read((atomic_t *) n);

atomic_add(i, (atomic_t *) n);
```

```
return t;

}

int compare_and_swap(int *n, int *old, int newv)

{

int t;

t = atomic_read((atomic_t *) n);

if (t == *old) {

atomic_set((atomic_t *) n, newv);

return 1;

}

else {
```

```
*old = t;

return 0;

}

}

/* Инициализиране на мутекса сочен от mp

Мутекса става инициализиран и отключен.

arg не се използва */

int mutex_init(mutex_t *mp, int type, void *arg)

{

int semid;

if ((mp == NULL) || (type != USYNC_PROCESS) || (type != USYNC_THREAD))

return EINVAL;
```

```
//Инициализиране на структурата

memset(mp, 0, sizeof *mp);

//Създаване на семафора

if ((semid = semget(IPC_PRIVATE, 1, IPC_CREAT|S_IRUSR|S_IWUSR))
return errno;

}

mp->semid = semid;

return 0;

}
```

/* Използва се в началото на критична секция за защитата и от едновременен достъп от множество потоци/процеси. Процеса или заключва мутекса и получава достъп, или се блокира до отключването му. */

```
int mutex_lock(mutex_t *mp)
```

```
{

struct sembuf sb;

for(;;) {

//Опит да се заключи мутекса и изход ако е успешно

if (fetch_and_add(&(mp->count), 1) == 0)

break;

//Ако мутекса е заключен, процеса се блокира със семафорна операция

sb.sem_flg = 0; sb.sem_num = 0; sb.sem_op = -1;

if (semop(mp->semid, &sb, 1)

return errno;

}

return 0;
```

```
}

/* Същото като mutex_lock, но не блокира ако мутекса е заключен */

int mutex_trylock(mutex_t *mp)

{

int x = 0;

if (mp->count != 0)     //Проверка дали мутекса е блокиран

return EBUSY;

//Ако не е заключен, опит да се заключи

else if (compare_and_swap(&(mp->count), &x, 1) == 0)

return EBUSY;

else {

return 0;
```

```
}

}

/* Освобождаване на мутекса в края на критичната секция.

Всички чакащи процеси/потоцо се освобождават и един от тях получава мутекса и го заключва */

int mutex_unlock(mutex_t *mp)

{

struct sembuf sb;

int n;

n = mp->count;

//Извличане на броя блокирани процеси и установяването му на 0

while(!compare_and_swap(&(mp->count), &n, 0));
```

```
//Увеличаване на семафора с броя на блокирание процеси

// така че всички да бъдат деблокирани

if (n > 1) {

mp->semops++;

sb.sem_flg = 0; sb.sem_num = 0; sb.sem_op = n - 1;

if (semop(mp->semid, &sb, 1)

return errno;

}

return 0;

}

/* Унищожаване на мутекса сочен от mp. Мутекса става неинициализиран */

int mutex_destroy(mutex_t *mp)
```

```
{



if (semctl(mp->semid, 0, IPC_RMID, 0)

return errno;



return 0;



}
```

6.2. thrdemo.cpp

```
/* Програмата демонстрира синхронизация между потоци в един процес с използване на мутекси.



Тя стартира NUM_THREADS на брой потоци, които опитват да увеличат стойността на глобална променлива



*/



#include



#include



#include
```

```
#include "mutexes.h"

#define NUM_THREADS 5

mutex_t Global_mutex;    //Мутексът за синхронизация

int Global_data = 0;     //Глобалните данни

//Функцията, която се изпълнява с потоците

void * change_global_data(void *arg)

{

//Заключване на мутекса и получаване на изкл. достъп до данните

//Всички останали потоци се блокират докато мутекса се освободи

mutex_lock(&Global_mutex);

Global_data++;           //Работа с данните

sleep(1);                //Симулиране на друга работа
```

```
printf("%d is global datan",Global_data);

//Освобождаване на мутекса

mutex_unlock(&Global_mutex);

return NULL;

}

int main(void)

{

int i;

pthread_t thr[NUM_THREADS];

//Инициализиране на мутекса

mutex_init(&Global_mutex, 1, NULL);
```

```
//Създаване на потоците


for (i = 0; i

if (pthread_create(&(thr[i]), NULL, change_global_data, NULL))


printf("error creating thread");


}


//Изчакване на потоците да приключат


for (i = 0; i

pthread_join(thr[i], NULL);


mutex_destroy(&Global_mutex);


printf("exiting");


return 0;


}
```

6.3. procdemo.cpp

/* Програмата демонстрира синхронизация между потоци в различни процеси с използване на мутекси

Два или повече екземпляра трябва да бъдат стартирани

Всеки процес стартира няколко потока, които записват в глобални данни

В зависимост от аргументите на програмата, потоците добавят или изваждат от данните.

*/

#include

#include

#include

#include

#include

#include

```
#include


#include


#include "mutexes.h"


/* INTERPROCESS_FILE се използва за споделяне на данни и синхронизация между различни процеси. Един порцес инициализира мутекса и го записва във файл, който по-късно се изобразява в памет от другите процеси */


#define INTERPROCESS_FILE "ipcsharedfile"


#define NUM_ADDTHREADS 6


#define NUM_SUBTRACTTHREADS 5


#define INCREMENT '0'


#define DECREMENT '1'


typedef struct {                // Струкура която се изобразява.


mutex_t Interprocess_mutex;   // Съдържа общите данни и мутекса.
```

```
int Interprocess_data;

} buffer_t;

buffer_t * buffer;

int zeroed[sizeof(buffer_t)];

int ipc_fd, i = 0;

pthread_t thr_add[NUM_ADDTHREADS], thr_sub[NUM_SUBTRACTTHREADS];

void * add_interprocess_data(void * arg), *subtract_interprocess_data(void * arg);

void create_shared_memory(), test_argv(char argv1[]);

int main(int argc,char * argv[])

{

test_argv(argv[1]);

switch (*argv[1]) {
```

```
//Стартиране на потоци, които увеличават данните.

case INCREMENT:

//Създаване на файла

create_shared_memory();

ipc_fd = open(INTERPROCESS_FILE, O_RDWR);

//Изобразяване на файла с данните

buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),

PROT_READ|PROT_WRITE, MAP_SHARED, ipc_fd, 0);

buffer->Interprocess_data = 0;

if (mutex_init(&buffer->Interprocess_mutex, USYNC_PROCESS, NULL)) {

printf("mutex_init error n");
```

```
exit(1);

}

// Стартиране на потоци

for (i = 0; i

pthread_create(&thr_add[i], NULL, add_interprocess_data, NULL);

//Изчакване на потоците да приключат

for (i = 0; i

pthread_join(thr_add[i], NULL);

break;

// Стартиране на потоци, които намаляват данните.

case DECREMENT:

//Изчакване да се създаде файла

while((ipc_fd = open(INTERPROCESS_FILE, O_RDWR)) == -1)
```

```
sleep(1);

//Изобразяване на файла

buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),

PROT_READ|PROT_WRITE, MAP_SHARED, ipc_fd, 0);

// Стартиране на потоци

for (i = 0; i

pthread_create(&thr_sub[i], NULL, subtract_interprocess_data, argv[1]);

for (i = 0; i

pthread_join(thr_sub[i], NULL);

break;

} /* end switch */

return 0;
```

```
} /* end main */

void *add_interprocess_data(void * arg)

{

//Заключване на мутекса и изключителен достъп до данните

mutex_lock(&buffer->Interprocess_mutex);

buffer->Interprocess_data++;

sleep(1);     //Симулиране на работа

printf("%d is add-interprocess datan", buffer->Interprocess_data);

mutex_unlock(&buffer->Interprocess_mutex);     //Освобождаване на мутекса

return NULL;

}

void *subtract_interprocess_data(void * arg)
```

```
{

mutex_lock(&buffer->Interprocess_mutex);

buffer->Interprocess_data--;

sleep(1);

printf("%d is subtract-interprocess datan", buffer->Interprocess_data);

mutex_unlock(&buffer->Interprocess_mutex);

return NULL;

}

void create_shared_memory()

{

unsigned int i;
```

```
ipc_fd = creat(INTERPROCESS_FILE, O_CREAT|O_RDWR );


for (i = 0; i

zeroed[i] = 0;


write(ipc_fd, &zeroed[i],2);


}


close(ipc_fd);


chmod(INTERPROCESS_FILE, S_IRWXU|S_IRWXG|S_IRWXO);


}


void test_argv(char argv1[])


{


if (argv1 == NULL) {


printf("use 0 as arg1 for initial processn
```

```
or use 1 as arg1 for the second processn");



exit(0);
```