6th Seminar

IPCs

There are several types of Inter Process Communication mechanisms:

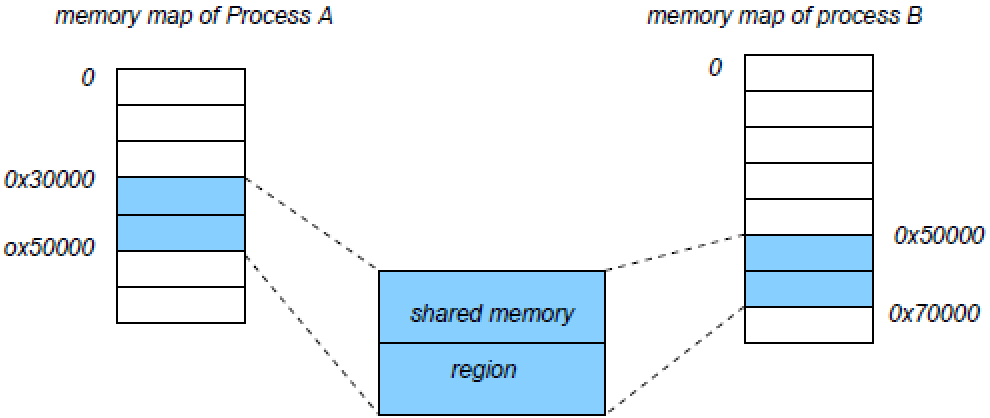
- Message queues
- Shared memory
- semaphores

IPC structures are identified by an integer identifier (non negative)

IPCs are created with the use of a key of `key_t` type and OS transforms the key into IPC identifier

Shared Memory

- A shared memory region is a portion of physical memory that is shared by multiple processes. In this region, structures can be set up by processes and others may read/write on them. Synchronization, if required, is achieved with the help of synchronization mechanisms (semaphores for example).



Shared Memory

- create a shared memory segment
  `int shmget(key_t key, size_t size, int shmflg)`
  returns the identifier of the shared memory segment associated with the value of the argument `key`.
  the returned size of the segment is equal to size rounded up to a multiple of PAGE SIZE.
  `shmflg` helps designate the access rights for the segment.
  If `shmflg` specifies both IPC_CREAT and IPC_EXCL and a shared memory segment already exists for key, then `shmget()` fails with errno set to EEXIST.

Shared Memory

- Attaching to a shared memory segment
  `void *shmat(int shmid, const void *shmaddr, int shmflg)`
  attaches the shared memory segment identified by `shmid` to the address space of the calling process.
  If `shmaddr` is NULL, the OS chooses a suitable (unused) address at which to attach the segment (**frequent choice**). Otherwise, `shmaddr` must be a page-aligned address at which the attachment occurs.
- Detaching from a shared memory segment
  `int shmdt(const void *shmaddr)`
  detaches the shared memory segment located at the address specified by `shmaddr` from the address space of the calling process.

Shared Memory

`int shmctl(int shmid, int cmd, struct shmid_ds *buf)`

performs the control operation specified by `cmd` on the shared memory segment whose identifier is given in shmid. The `buf` argument is a pointer to a `shmid_ds` structure:

```
struct shmid_ds {
  struct ipc_perm shm_perm ; /* Ownership and permissions */
  size_t shm_segsz ; /* Size of segment ( bytes ) */
  time_t shm_atime ; /* Last attach time */
  time_t shm_dtime ; /* Last detach time */
  time_t shm_ctime ; /* Last change time */
  pid_t shm_cpid ; /* PID of creator */
  pid_t shm_lpid ; /* PID of last shmat (2) / shmdt (2) */
  shmatt_t shm_nattch ; /* No . of current attaches */
  ...
};
```

Shared Memory

- some possible values for `cmd`:

- `IPC_STAT`: copy information from the kernel data structure associated with shmid into the `shmid_ds` structure pointed to by `buf`.
- `IPC_SET`: write the value of some member of the `shmid_ds` structure pointed to by `buf` to the kernel data structure associated with this shared memory segment, updating also its `shm_ctime` member.
- `IPC_RMID`: mark the segment to be destroyed. The segment will be destroyed after the last process detaches it (i.e.,`shm_nattch` is zero).

```c
// writer.c
int main()
{
	// ftok to generate unique key
	key_t key = ftok("shmfile",65);
	// shmget returns an identifier in shmid
	int shmid = shmget(key,1024,0666|IPC_CREAT);
	// shmat to attach to shared memory
	char *str = (char*) shmat(shmid,(void*)0,0);
	printf("Write Data : ");
	gets(str);
	printf("Data written in memory: %s\n",str);
	//detach from shared memory
	shmdt(str);
	return 0;
}
```

```c
// reader.c
int main()
{
        // ftok to generate unique key
        key_t key = ftok("shmfile",65);
        // shmget returns an identifier in shmid
        int shmid = shmget(key,1024,0666|IPC_CREAT);
        // shmat to attach to shared memory
        char *str = (char*) shmat(shmid,(void*)0,0);
        printf("Data read from memory: %s\n",str);
        //detach from shared memory
        shmdt(str);
        // destroy the shared memory
        shmctl(shmid,IPC_RMID,NULL);
        return 0;
}
```

Semaphores

- Semaphores were proposed by Dijkstra to manage concurrent processes by using a simple integer value, which is known as a semaphore.

- is simply a variable which is non-negative and shared between threads.
- is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

- Semaphores are of two types:

- Binary Semaphore – also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes/threads.
- Counting Semaphore – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Semaphores

- There are two operations which can be used to access and change the value of the semaphore variable:

- P - is also called wait, sleep or down operation
- V - is also called signal, wake-up or up operation

- Both operations are atomic and semaphore is always initialized to value 1
- A critical section is surrounded by both operations to implement process synchronization.

```
P(semaphore s){
   while (s==0);
   s=s-1;
}
```

```
V(semaphore s){
   s=s+1;
}
```

```
P(s);
  // critical section
V(s);
```

Semaphores vs Mutexes

- There is a similarity between binary semaphore and mutex. But they are not the same. The purpose of mutex and semaphore are different. Due to similarity in their implementation a mutex would be referred as binary semaphore.
- A mutex is locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).
- A semaphore is a signaling mechanism.

Semaphores

POSIX semaphores (two forms): named semaphores and unnamed semaphores.

- A named semaphore is identified by a name of the form /somename; a null-terminated string of up to 251 characters consisting of an **initial slash**, followed by one or more characters, none of which are slashes. Two processes can operate on the same named semaphore by passing the same name to `sem_open`.
- The `sem_open` function creates a new named semaphore or opens an existing named semaphore. After the semaphore has been opened, it can be operated on using `sem_post` and `sem_wait`. When a process has finished using the semaphore, it can use `sem_close` to close the semaphore. When all processes have finished using the semaphore, it can be removed from the system using `sem_unlink`.

Semaphores

- An unnamed (memory-based) semaphore does not have a name. Instead the semaphore is placed in a region of memory that is shared between multiple threads (a thread-shared semaphore) or processes (a process-shared semaphore). A thread-shared semaphore is placed in an area of memory shared between the threads of a process, for example, a global variable. A process-shared semaphore must be placed in a shared memory region (e.g., a System V shared memory segment created using `shmget`, or a POSIX shared memory object built created using `shm_open`).
- Before use, an unnamed semaphore must be initialized using `sem_init`. It can then be operated on using `sem_post` and `sem_wait`. When is no longer required, and before the memory in which it is located is deallocated, the semaphore should be destroyed using `sem_destroy`.

Semaphores

```
static int sem_init (sem_t *sem, int pshared, unsigned value)
```

Initialize an unnamed semaphore.

```
static int sem_post (sem_t *sem)
```

Unlock a semaphore.

```
static int sem_wait (sem_t *sem)
```

Lock a semaphore.

```
int sem_timedwait (sem_t *sem, const struct timespec *abstime)
```

Similar to `sem_wait' but wait only until abstime.

```
static int  sem_trywait (sem_t *sem)
```

Test whether sem is posted.

```
static int  sem_getvalue (sem_t *sem, int *sval)
```

Get current value of sem and store it in sval.

Semaphores

`static sem_t * sem_open (const char *name, int oflag,...)`

Open a named semaphore name with open flags oflag.

`static int sem_close (sem_t *sem)`

Close descriptor for named semaphore sem.

`static int sem_unlink (const char *name)`

Remove named semaphore name.

`static int sem_destroy (sem_t *sem)`

destroy an unnamed semaphore

```
//POSIX Semaphore test example using shared memory
# define SEGMENTSIZE sizeof(sem_t)   // sem_t means semaphore type
# define SEGMENTPERM 0666
int main (int argc ,char ** argv )
{
   sem_t *sp;
   int retval, id, err ;
   if ( argc <= 1) {
       printf("Need shmem id. \n");
      exit (1) ;
   }
   sscanf( argv [1], "%d", &id);    // Get id from command line
   printf(" Allocated %d\n", id);
   sp = (sem_t *) shmat (id ,( void *) 0, 0);   // Attach the segment
   if (( int ) sp == -1) {
       perror (" Attachment .");
      exit (2) ;
   }
```

```
    retval = sem_init (sp ,1 ,1) ; // Initialize the semaphore
    if ( retval != 0) {
        perror(" Couldn't initialize .");
        exit (3) ;
    }
    retval = sem_trywait (sp);
    printf("Did trywait . Returned %d >\n", retval ); getchar ();
    retval = sem_trywait (sp);
    printf("Did trywait . Returned %d >\n", retval ); getchar ();
    retval = sem_trywait (sp);
    printf("Did trywait . Returned %d >\n", retval ); getchar ();
    err = shmdt (( void *) sp); //  Remove segment
    if (err == -1)
        perror (" Detachment .");
    return 0;
}
```

Semaphores

Intre A şi B sunt n linii prin care trec m trenuri, m > n
In gara A intră simultan maximum m trenuri care vor să ajungă în gara B. De la A spre B există simultan n linii, m > n. Fiecare tren intră în A la un interval aleator. Dacă are linie liberă între A şi B, o ocupă şi pleacă către B, durata de timp a trecerii este una aleatoare. Să se simuleze aceste treceri. Soluţiile, una folosind variabile condiţionale, cealaltă folosind semafoare, sunt prezentate în tabelul următor.

```c
#define N 5
#define M 13
#define SLEEP 4
pthread_mutex_t mutcond;
pthread_cond_t cond;
int linie[N], tren[M], inA[M+1],
dinB[M+1], liniilibere;
pthread_t tid[M];
time_t start;
void t2s(int *t, int l, char *r) {
        int i;
        char n[10];
        sprintf(r, "[");
        for ( i = 0; i < l; i++) {
        sprintf(n,"%d, ",t[i]);
        strcat(r, n);}
        i = strlen(r) - 1;
        if(r[i] == ' ') r[i - 1] = 0;
        strcat(r, "]");
}
```

```c
#define N 5
#define M 13
#define SLEEP 4
sem_t sem; // Asteapta / semnaleaza
eliberarea uneia din cele N linii
sem_t sem, mut; // Asigura acces
exclusiv la tabelele globale
int linie[N], tren[M], inA[M+1],
dinB[M+1];
pthread_t tid[M];
time_t start;
void t2s(int *t, int l, char *r) {
        int i;
        char n[10];
        sprintf(r, "[");
        for( i = 0; i < l; i++) {
        sprintf(n,"%d, ",t[i]);
        strcat(r, n);
        }
```

```
void prinT(char *s, int t) {
        int i;
        char a[200],l[200],b[200];
        for(i=0; inA[i] != -1; i++);
        t2s(inA, i, a);
        t2s(linie, N, l);
        for(i=0; dinB[i] != -1; i++);
        t2s(dinB, i, b);
        printf("%s
%d\tA:%s\tLines:%s\tB:%s\ttime:
%ld\n",s,t,a,l,b,time(NULL)-start);
}

//rutina unui thread
void* trece(void* tren) {
        int i, t, l;
        t = *(int*)tren;
        sleep(1 + rand() % SLEEP); //
Modificati timpii de stationare
```

```
        i = strlen(r) - 1;
        if( r[i] == ' ') r[i - 1] = 0;
        strcat(r, "]");
}
void prinT(char *s, int t) {
        int i;
        char a[200],l[200],b[200];
        for(i = 0; inA[i] != -1; i++);
        t2s(inA, i, a);
        t2s(linie, N, l);
        for(i = 0; dinB[i] != -1;
i++);
        t2s(dinB, i, b);
        printf("%s
%d\tA:%s\tLines:%s\tB:%s\ttime:
%ld\n",s,t,a,l,b,time(NULL)-start);
}
```

```
        pthread_mutex_lock(&mutcond);
        for(i=0;inA[i]!=-1;i++);
        inA[i]=t;
        prinT("EnterA", t);
        for(;liniilibere==0;)
pthread_cond_wait(&cond,&mutcond);
        for(l=0;linie[l]!=-1;l++);
        linie[l]=t;
        liniilibere--;
        for(i=0;inA[i]!=t;i++);
        for(;i<M;inA[i]=inA[i+1],i++);
        prinT(" A => B", t);
        pthread_mutex_unlock(&mutcond);

        sleep(1 + rand() % SLEEP);
```

```
//rutina unui thread
void* trece(void* tren) {
    int i, t, l;
    t = *(int*)tren;
    sleep(1+rand()%SLEEP); //Inainte de
==> A
    sem_wait(&mut);
    for( i = 0; inA[i] != -1; i++);
    inA[i] = t;
    prinT("EnterA", t);
    sem_post(&mut);
    sem_wait(&sem); // In A ocupa linia
    sem_wait(&mut);
    for(l = 0; linie[l] != -1; l++);
    linie[l] = t;
```

```c
        pthread_mutex_lock(&mutcond);
        linie[l] = -1;
        liniilibere++;
        for(i=0; dinB[i] != -1; i++);
        dinB[i] = t;
        prinT("  OutB", t);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutcond);
}

//main
int main(int argc, char* argv[]) {
        int i;
        start = time(NULL);
        pthread_mutex_init(&mutcond,NULL
);
        pthread_cond_init(&cond, NULL);
        liniilibere = N;
```

```c
        for( i = 0; inA[i] != t; i++);
        for( ; i < M; inA[i] = inA[i +
1], i++);
        prinT(" A => B", t);
        sem_post(&mut);

        sleep(1+rand()%SLEEP); //
Trece trenul  A ==> B
        sem_wait(&mut);
        linie[l] = -1;
        for(i=0; dinB[i] != -1; i++);
        dinB[i] = t;
        prinT("  OutB", t);
        sem_post(&mut);

        sem_post(&sem); //In B
elibereaza linia
}
```

```
        for(i=0;i<N;linie[i]=-1,i++);
        for(i=0;i<M;tren[i]=i,i++);
        for (i=0;i<M+1;inA[i]=-
1,dinB[i]= -1, i++);

        // ce credeti despre ultimul
parametru &i?
        for(i=0;i<M;i++)
pthread_create(&tid[i],NULL,trece,
&tren[i]);
        for(i=0;i<M;i++)
pthread_join(tid[i], NULL);

        pthread_mutex_destroy(&mutcond
);
        pthread_cond_destroy(&cond);
        return 0;
}
```

```
int main(int argc, char* argv[]) {
        int i;
        start = time(NULL);
        sem_init(&sem, 0, N);
        sem_init(&mut, 0, 1);
        for(i=0;i<N;linie[i]=-1,i++);
        for(i=0;i<M;tren[i]=i,i++);
        for(i=0;i<M+1;inA[i]=-
1,dinB[i] = -1, i++);
        // ce credeti despre ultimul
parametru &i in loc de &tren[i]?
        for(i=0; i < M; i++)
pthread_create(&tid[i],NULL,trece,
&tren[i]);
        for(i=0;i<M;i++)
pthread_join(tid[i], NULL);
        sem_destroy(&sem);
        sem_destroy(&mut);
        return 0;
}
```

Barriers

- When multiple threads are working together, it might be required that threads wait for each other at a certain event or point in the program before proceeding ahead.
- Such operations can be implemented by adding a barrier in the thread. A barrier is a point where the thread is going to wait for other threads and will proceed further only when predefined number of threads reach the same barrier.
- A barrier is a variable of the type `pthread_barrier_t`.

```
int pthread_barrier_init(pthread_barrier_t *restrict
barrier,const pthread_barrierattr_t *restrict attr, unsigned
count);
```

- `barrier`: The variable used for the barrier
- `attr`: Attributes for the barrier, which can be set to NULL to use default attributes
- `count`: Number of threads which must wait call pthread_barrier_wait on this barrier before the threads can proceed further.

Barriers

- Once a barrier in initialized, a thread can be made to wait on the barrier for other threads using the function
  `int pthread_barrier_wait(pthread_barrier_t *barrier);`
  where the barrier is the same variable initialized using pthread_barrier_init.
- A thread will keep waiting till the "`count`" number of threads passed during init do not call the `wait` function on the same barrier.
- Once `pthread_barrier_wait` has been called by “`count`” threads, the constant PTHREAD_BARRIER_SERIAL_THREAD is returned to one unspecified thread and 0 is returned to each of the remaining threads. The barrier is then reset to the state it had as a result of the most recent `pthread_barrier_init`() function that referenced it..

Barriers

- A barrier can be destroyed using the function
  `int pthread_barrier_destroy(pthread_barrier_t *barrier);`
  The barrier should be destroyed only when no thread is executing a wait on the barrier

```
pthread_barrier_t   barrier; // the barrier synchronization object
void * thread1 (void *not_used){
       time_t  now;
       char    buf [27];
       time (&now);
       printf ("thread1 starting at %s", ctime_r (&now, buf));
       // do the computation
       // let's just do a sleep here...
       sleep (20);
       pthread_barrier_wait (&barrier);
       // after this point, all three threads have completed.
       time (&now);
       printf ("barrier in thread1() done at %s", ctime_r (&now,
buf));
}
```

```
void * thread2 (void *not_used){
        time_t  now;
        char   buf [27];

        time (&now);
        printf ("thread2 starting at %s", ctime_r (&now, buf));

        // do the computation
        // let's just do a sleep here...
        sleep (40);
        pthread_barrier_wait (&barrier);
        // after this point, all three threads have completed.
        time (&now);
        printf ("barrier in thread2() done at %s", ctime_r (&now,
buf));
}
```

```
int main (){
        time_t  now;
        char buf [27];
        // create a barrier object with a count of 3
        pthread_barrier_init (&barrier, NULL, 3);
        // start up two threads, thread1 and thread2
        pthread_create(NULL, NULL, thread1, NULL);
        pthread_create(NULL, NULL, thread2, NULL);
        // at this point, thread1 and thread2 are running
        // now wait for completion
        time(&now);
        printf("main waiting for barrier at %s", ctime_r (&now, buf));
        pthread_barrier_wait(&barrier);
        // after this point, all three threads have completed.
        time(&now);
        printf("barrier in main () done at %s", ctime_r (&now, buf));
}
```