IPC – part 2
POSIX synchronization

Last modification date: 13.03.2019

UNIX System V IPC

	Message queues	Shared memory	Semaphores
Header file	**<sys/msg.h>**	**<sys/shm.h>**	**<sys/sem.h>**
Creation/opening	**msgget()**	**shmget()**	**semget()**
Control operations	**msgctl()**	**shmctl()**	**semctl()**
Communication	**msgsnd()** **msgrcv()**	**smat()** **shmdt()**	**semop()**

- Unix System V IPC objects have **kernel persistence** – they exist until the kernel is re-loaded or they are explicitly deleted.
- **Namespace**
  - IPC objects are global (one namespace, accessible to all processes)
  - A key of type **key_t** (positive integer) identifies an object in the system.. Suggested method of the key generation:

    key_t ftok(const char *pathname, int id);
  - After opening an IPC object is available to the process via IPC System V **object identifier**; the identifier is unique for each IPC object type.

Unix System V IPC objects – access rights

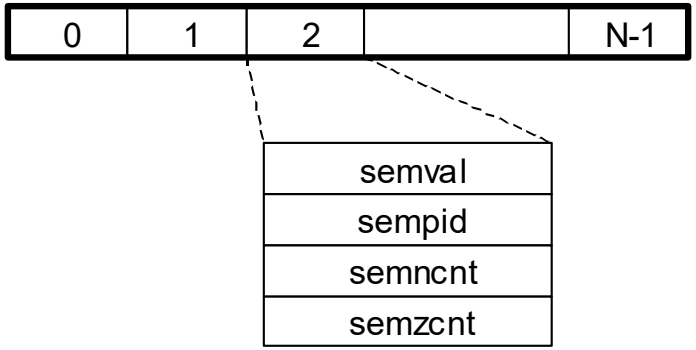
For each IPC object the kernel creates and maintains the following structure (see **<sys/ipc.h>, svipc(7)**) with access rights to the object:

```
struct ipc_perm {
  uid_t uid;   /* UID id of the owner */
  gid_t gid;   /* GID id of the owner group */
  uid_t cuid;  /* UID id of the object creator (user id) */
  gid_t cgid;  /* GID id of the object creator group */
  mode_t mode; /* access rights (RWXRWXRWX) */
  ulong_t seq; /* (SVR4) sequence number, incremented after
       each destruction of an object with given key */
  key_t key;   /* key */
};
```

Access rights are checked prior to **each operation** involving IPC object.

UNIX System V IPC – semaphores

- Each IPC semaphore entity is uniquely associated with an integer identifier and a system data structure **semid_ds** (see later).
- UNIX System V IPC semaphore entities contain **semaphore sets (arrays)**



- For each **semaphore** (element of semaphore set) the operating system maintains::

  semval - semaphore counter (**non-negative integer**)

  sempid - PID of a process, which performed semaphore operation most recently

  semncnt – the number of processes, that wait for **semval** to become **positive**

  semzcnt – the number of processes, that wait for **semval** to become **zero**

  semadj – a value that is to be added to semaphore after a process that used the semaphore terminates. **semadj** value changes after each semaphore operation done with **SEM_UNDO** flag on.

Creation of IPC semaphore sets

```
int semget(key_t key, size_t size, int oflag)
```

creates a semaphore set of given **size** or attaches process to an existing one.

oflag is a bit OR of access rights (RW-RW-RW-) and symbolic bits IPC_CREAT, IPC_EXCL

The function returns integer id of the semaphore set, which can be later used for semaphore operations.

Remarks

- **key==IPC_PRIVATE** works the same as for **shmget** and **msgget**, i.e. there is guarantee, that a new IPC entity is created with an internal key that is unique in the following sense: there is no combination of **pathname** and **id** in **ftok()** call which would created the same key.
- The values of the semaphores in a newly created set are indeterminate. Although Linux, like many other implementations, initializes the semaphore values to 0, a portable application cannot rely on this: it should explicitly initialize the semaphores to the desired values.

Creation and opening access to IPC objects

Results of **semget()** function calls depend on the parameter **oflag** as shown below. Success of the attempt depends also on access rights, obviously.

Parameter **oflag**	IPC object with given key does not exists	IPC object with given key already exists
No **IPC_CREAT** and no **IPC_EXCL** bit patterns	Error, **errno==ENOENT**	OK, the object with given key is subject to opening access operation
IPC_CREAT bit pattern set	OK, new object may be created	as above
(IPC_CREAT | IPC_EXCL) bit pattern set	OK, new object may be created	Error, **errno==EEXIST**

Modification of semaphore set properties

```
int semctl(int semid, int num_sem, int command,
           union semun arg);
```

The function returns **0** on success, and **–1** on failure (**errno** contains error code)

semid, num_sem - id of the semaphore object (set)

command - code of the command:

IPC_STAT - copies to **arg.buf** semaphore status

IPC_SET - sets access rights (**sem_perm.uid**, **sem_perm.gid**, **sem_perm.mode**) to values from **arg.buf**. The command available to a process with **EUID==0**, or **==sem_perm.cuid**, or **==sem_perm.uid**, where **sem_perm** is a system **semid_ds** structure associated with given **semid**.

IPC_RMID - **immediately** removes semaphore set; processes waiting for semaphore set operations are awoken.

GETVAL - returns semaphore counter (**semval**) to **arg.array**

GETALL - returns **all** **semval** counters of the set in **arg.array**

SETVAL - sets semaphore counter (**semval**) to **arg.val**

SETALL - sets **all** **semval** counters of the set to **arg,array**

GETCNCNT, **GETCZCNT**, **GETPID** – returns, respectively, fields **semncnt**, **semzcnt**, **sempid** of the system semid_ds. structure to **arg.val**

Semaphore set related data structures

Last parameter of **semctrl** call (i.e. **arg**) is a **union**, which a programmer **has to define** him/her-self, as follows

```
union semun{
  int val;  /* used with SETVAL */
  struct semid_ds *buf;  /* used with IPC_STAT, IPC_SET */
  unsigned short *array;  /* used with GETVAL,GETALL, SETALL */
} arg;
```

Location of **semid_ds** structure definition (see semctl(2))

contains at least three following fields:

struct ipc_perm **sem_perm**; /* access permition */

ushort_t **sem_nsems**; /* length of the semafor array (set) */

time_t **sem_otime**; /* last **semop** timestamp */

Example: semaphore creation and initialization

Example

Create a **single** new semaphore, setting its value to **10**

```
key_t key;
union semun arg;
key=ftok(name,getpid())
if (key == (key_t)-1) { /* error handling */ . . . }
arg.val=10;
semid=semget(key,1,IPC_CREAT | IPC_EXCL | 0600);
if (semid==-1) {
    if (errno==EEXIST){
          fprintf(stderr,"Semaphore already exists\n");
/* if access to the existing semaphore is needed:
          semid=semget(key,1,0);. . .      */
    } else {/* error handling */ . . . }
} else if (semctl(semid,0,SETVAL,arg)==-1){
      /* error handling */ . . .
}
```

IPC semaphore set operations

```
int semop(  int semid, struct sembuf *sops,
            size_t nsops);
```

The function executes a set of semaphore operations for semaphore set identified with **semid**. Upon success 0 is returned; -1 otherwise (**errno** contains code of error).

Each of **nsops** elements of a struct array pointed at by **sops** contains:

```
struct sembuf {
  unsigned short sem_num;  /* semaphore nr (0,..) */
  short sem_op;  /* operation code */
  short sem_flg;  /* flag==0 for default (blocking) behaviour.
        IPC_NOWAIT  – non-blocking behaviour,
        SEM_UNDO  – operation will be reversed by system upon process
                termination (system keeps semadj value = negative of the
                accumulated sum of  sem_op values to be undone) */
}
```

IPC semaphore set operations – cont.

- Operations
  - **sem_op < 0**
    - attempts to decrease semaphore counter by **|sem_op|**
    - blocks caller, when the semaphore counter **< |sem_op|**
  - **sem_op > 0**
    - increases the semaphore counter by **sem_op**
  - **sem_op == 0**
    - blocks the caller until semaphore counter **== 0**

- **Example**: standard semphore **wait** operation (taking semaphore)

```
struct  sembuf lock = {0, -1, SEM_UNDO };
if(semop(semid, &lock, 1)<0) { perror("lock"); ... }
```

- **Example**: standard semphore **post** operation (releasing semaphore)

```
struct    sembuf    unlock = {0, 1, 0 };
if(semop(semid, &unlock, 1)<0){ perror("unlock"); ... }
```

Example

```
. . .
union semun{int val; struct semid_ds. *buf, ushort_t *array; };
int s_init(char *name, int cnt) {/* creation of a single semaphore or
  attaching to existing one; always setting its value to cnt */
    int semid;
    key_t key;
    union semun arg;
    if( (key= ftok(name, 1))==(key_t -1)){ perror("key"); exit(1); }
    semid = semget(key, 1, IPC_CREAT | 0600);
    /* NOTE: we do not care if a new semaphore was created or we
     * use existing semaphore. Anyway its value is set to cnt
    */
    arg.val=cnt;
    if(semctl(semid,0,SETVAL,arg)<0){   perror("sem_init"); exit(1); }
    return semid;
}
void s_wait(int semid) {/* standard semaphore operation: wait */
    struct sembuf s;
    s.sem_num = 0; s.sem_op = -1; s.sem_flg = SEM_UNDO;
    if(semop(semid, &s, 1) < 0) { perror("sem_wait"); exit(1);}
}
void s_post(int semid) {{/* standard semaphore operation: post */
    struct sembuf s;
    s.sem_num = 0; s.sem_op = 1; s.sem_flg = SEM_UNDO;
    if(semop(semid, &s, 1) < 0) { perror("sem_post"); exit(1);}
}
```

Example – cont.

```
static int  flag;
void display(int semid, char c) {
    int i;
    if(flag) s_wait(s);
    for (i = 0; i < 5; i++) {/* begin of the critical section */
        printf("%c", c); fflush(stdout); sleep(1);
    }/* end of the critical section */
    if(flag) s_post(s);
}
int main(int argc, char *argv[]){
    int semid;
    if(argc>1) flag=1;
    semid = s_init("semprog",1); // semprog file has to exist !!
    if (fork() == 0) { // Child process
        for (;;) display(semid, '1');
        return(0);
    }
    for (;;) display(semid, '2');
    return(0);
}
```

The program called with no parameters => **flag==0** => no protection of the critical section.
Otherwis => **flag==1** => protection against concurrent **display**ing by a parent and a child.
What happens, when the program is run in two terminal windows? Why?

POSIX IPC

	Message queues	Shared memory	Semaphores
Header files	**<mqueue.h>**	**<sys/mman.h>**	**<semaphore.h>**
Creation/ opening access/removal	**mq_open(), mq_close(), mq_unlink()**	**shm_open(), shm_unlink()**	**sem_open() sem_close(), sem_unlink(), sem_init(), sem_destroy()**
Control operations	**mq_getattr(), mq_setattr()**	**ftruncate(), fstat()**	
Communication	**mq_send() mq_receive(), mq_notify()**	**mmap() munmap()**	**sem_wait(), sem_trywait(), sem_post(), sem_getvalue()**

- POSIX IPC object have mostly **kernel persistence** (with notable exception of the semaphore in memory, which is **process persistent**, i.e. it exists until it is explicitly destroyed (**sem_destroy()**) or its memory becomes unavailable).

POSIX SEM – namespace and id-space

- Named semaphores identify a semaphore instance by a string (**name** argument of **sem_open()** function call).
  - It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments.
  - The **name** argument shall conform to the construction rules for a pathname.
    - If **name** begins with the slash character, then processes calling **sem_open( )** with the same value of **name** shall refer to the same semaphore object, as long as that name has not been removed.
    - If **name** does not begin with the slash character, the effect is implementation-defined.
    - The interpretation of slash characters other than the leading slash character in *name* is implementation-defined.
- Linux requires name of the form **/somename** (not longer than {NAMELEN-4} bytes). Named semaphores are created in a virtual filesystem, normally mounted under **/dev/shm**, with names of the form **sem.somename**.
- Linux semaphore characteristics: see **sem_overview(7)**

Semaphore creation/opening

```
sem_t  *sem_open(const char *name, int oflag
      /* , mode_t mode, unsigned int value */);
```

The function returns a pointer to a semaphore structure; on failure it returns **(sem_t)(SEM_FAILED)**, setting error code in **errno** .

Parameters:

name - the semaphore name.

oflag - specifies access mode (**O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL**). If a new semaphore is created the two arguments are needed:

mode - access rights (**r** i **w** – as for files)

value - initial semaphore value

Remarks:

- Function call can be interrupted by signal delivery to the process calling **sem_open()**
- Maximum nr of POSIX semaphores: **{SEM_NSEM_MAX}** (>=256), max.semaphore value**:** **{SEM_VALUE_MAX}** (>=32767), see **<unistd.h>**

Closing access to semaphore/ removal

- When a process no longer needs access to a semaphore it should close access with:

```
int sem_close(sem_t *sem);
```

- A semaphore with given **name** can be removed with:

```
int sem_unlink(char *name);
```

Note: the function removes the name of the semaphore from the system immediately, but the semaphore is really destroyed when all processes, which opened access to this semaphore close the semaphore descriptor with **sem_close()** call.

Wait and post operations

- Blocking and non-blocking **wait** operations are invoked with:

```
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
```

- **post** operation is invoked with:

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

- Current value of the semaphore can be retrieved with:

```
int sem_getvalue(sem_t * sem, int *valp);
```

Unnamed semaphores

- Unnamed semaphore resides in a memory structure (**sem_t**) that is shared by cooperating threads or processes. The structure is initialized with given **value** with a call:

```
int sem_init(sem_t * sem, int shared,
      unsigned int value);
```

If **shared==0** – than the semaphore is shared by threads of one process, otherwise it is shared by processes.

- The following call destructs the semaphore associated with the given structure **sem** :

```
int sem_destroy(sem_t * sem);
```

Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using **sem_init(…)**.

- The semaphore operations (wait/post) are made with calls to **sem_wait()** and **sem_post()** - respectively (the same function which are used for named semaphores).

Other POSIX synchro. objects

- Mutex
- Condition variable
- Barrier
- Read-Write Lock

Other POSIX synchro. objects: mutex

- **Mutex** - a synchronization object used to allow multiple threads to serialize their access to shared data. The name derives from the capability it provides; namely, mutual-exclusion. **The thread that has locked a mutex becomes its owner and remains the owner until that same thread unlocks the mutex**.
- Basic mutex operations:
  - Locking access to **critical section**

```
int pthread_mutex_lock(pthread_mutex_t *mp);// blocking
int pthread_mutex_trylock(pthread_mutex_t *mp);// non-bl.
```

  - Unlocking access to critical section

```
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

Pattern of use:

```
lock
// critical section: code that should access a shared
// variable in exclusive fashion
unlock
```

Mutex creation and initialization

- Mutex creation and default initialization

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
```

- Initialization of a mutex

```
int pthread_mutex_init (
        pthread_mutex_t *mp,// ptr to mutex
        const pthread_mutexattr_t *mattr);// ptr to attributes
```

- Mutex destruction

```
int pthread_mutex_destroy( pthread_mutex_t *mutex);
```

Mutex attributes

- Initialization of an attributes structure with default values

```
int pthread_mutexattr_init(pthread_mutexattr_t *mattr);
```

For setting/getting individual attributes see `man pthread_mutexattr_destroy.`

Available attributes are implementation specific.

Linux mutex attributes (non-RT):

`pshared` mutex can be shared (or not) by processes

`type : NORMAL` mutex does not detect deadlock when locking locked

`ERRORCHECK` operations are checked for validity (e.g. locking locked etc.)

`RECURSIVE` multiple locking possible but require multiple unlocking

`PTHREAD_MUTEX_DEFAULT` - an implementation may map this mutex to one of the other mutex types.

- Destruction of attributes structure

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *mattr);
```

Robust mutex

- POSIX mutexes have robustness attribute (see **pthread_mutexattr_setrobust(3)**). It specifies the behavior of the mutex when the owning thread dies without unlocking the mutex.

- If a mutex is initialized with the PTHREAD_MUTEX_ROBUST attribute and its owner dies without unlocking it, any future attempts to call **pthread_mutex_lock()** on this mutex will succeed and return EOWNERDEAD, to indicate that the original owner of the locked mutex no longer exists and so the mutex is in inconsistent state. Usually after EOWNERDEAD is returned, the next owner should call **pthread_mutex_consistent()** on the acquired mutex first, to make it consistent again - before using it any further.

- If the next owner unlocks the mutex using **pthread_mutex_unlock()** before making it consistent, the mutex will be permanently unusable and any subsequent attempts to lock it using **pthread_mutex_lock()** fail with the error ENOTRECOVERABLE. The only permissible operation on such a mutex is **pthread_mutex_destroy( ).**

Condition variable

- **Condition variable (CV)** – A synchronization object which allows a thread to suspend execution, repeatedly, until some associated predicate becomes true. A thread whose execution is suspended on a condition variable is said to be blocked on the condition variable.
- Condition variable (CV) creation and default initialization

```
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
```

- Initialization of a CV:

```
int pthread_cond_init (
        pthread_cond_t *cond, // ptr to CV
        const pthread_condattr_t *mattr); // ptr to attributes
```

- CV destruction

```
int pthread_cond_destroy( pthread_cond_t *cond);
```

Condition variable

- Condition variable cooperates with a mutex.

```
int pthread_cond_wait ( pthread_cond_t *cv ,
                pthread_mutex_t *mutex ) ;
```

The function atomically:

- realese **mutex** and
- causes the calling thread to block on the condition variable **cond**.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

- These functions shall unblock threads blocked on a condition variable **cond**.

```
int pthread_cond_broadcast(pthread_cond_t * cond); /* all */
int pthread_cond_signal(pthread_cond_t * cond); /* >= 1 */
```

If no thread is actually found blocked, while signaling, the unblocking notification is permanently lost.

Example of CV + mutex use

```
pthread_cond_t cv=PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex;
volatile sig_atomic_t condition_is_false =1;
pthread_mutex_lock (&mutex ) ;
while ( condition_is_false ) {/* condition check */
        int ret = pthread_cond_wait (&cv , &mutex ) ;
        if ( ret ) { . . . . /* error */ }
. . . /* the main part of critical section code executes
          under mutex protection */
}
pthread_mutex_unlock (&mutex ) ;
```

```
. . .
pthread_mutex_lock (&mutex ) ;
condition_is_false =0; /* flag change under
                          mutex protection */
pthread_cond_signal (&cv ) ;/* signalling */
pthread_mutex_unlock (&mutex ) ;
. . .
```

Example: „Hello world” with CV

```
pthread_mutex_t prt_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t prt_cv = PTHREAD_COND_INITIALIZER;
int prt = 0;
void *hello_thread(void *arg){
        pthread_mutex_lock(&prt_lock);
        printf("hello ");
        prt = 1;
        pthread_cond_signal(&prt_cv);
        pthread_mutex_unlock(&prt_lock);
        return (NULL);
}
void *world_thread(void *arg){
        pthread_mutex_lock(&prt_lock);
        while (prt == 0)
                pthread_cond_wait(&prt_cv, &prt_lock);
        printf("world");
        pthread_mutex_unlock(&prt_lock);
        pthread_exit(0);
}
```

Example: „Hello world” with CV – cont.

```
int main(int argc, char *argv[]){
        int n;
        pthread_attr_t attr;
        pthread_t tid;
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
        if ((n = pthread_create(&tid, &attr, world_thread, NULL)) > 0){
                        fprintf(stderr, "pthread_create: %s\n",
                         strerror(n)); exit(1);
        }
        pthread_attr_destroy(&attr);
        if ((n = pthread_create(&tid, NULL, hello_thread, NULL)) > 0) {
                        fprintf(stderr, "pthread_create: %s\n",
                                        strerror(n)); exit(1);
        }
        if ((n = pthread_join(tid, NULL)) > 0) {
                fprintf(stderr, "pthread_join: %s\n", strerror(n));
                exit(1);
        }
        printf("\n");
        return (0);
}
```

Other POSIX synchronization objects

- **Barrier** - A synchronization object that allows multiple threads to synchronize at a particular point in their execution. See **man pthread_barrier_wait** . A barrier synchronization function

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

is blocking the calling thread until the required number of threads calls this function. Then one thread returns from **pthread_barrier_wait()** call with a non-zero valu (**PTHREAD_BARRIER_SERIAL_THREAD**) and remaining receive 0; Afterwards the barrier state is reset to that directly after initialization with **pthread_barrier_init()** function. **pthread_barrier_destroy()** function destroys the referenced barrier – see **man: thread_barrier_destroy(3P), thread_barrier_wait(3P)**

- **Multiple readers, single writer (read-write) locks** allow many threads to have simultaneous read-only access to data while allowing only one thread to have write access at any given time. They are typically used to protect data that is read-only more frequently than it is changed. Read-write locks can be used to synchronize threads in the current process and other processes if they are allocated in memory that is writable and shared among the cooperating processes and have been initialized for this behavior. See e.g. **man pthread_rwlock_rdlock, pthread_rwlock_wrlock, pthread_rwlock_unlock**

Effects of some system calls on IPC objects

Object type	fork()	exec()	_exit()
Sys.V sem	All **semadj** values in the child set to 0	All **semadj** values are moved to the new program	All **semadj** values are added to corresp. semaphore values
POSIX named sem	All opened semaphores are seen in child process	All open semaphores become closed	All open semaphores become closed
POSIX unnamed sem	Shared if in shared memory with sharing attribute	Disappear, unless they reside in still open shared memory and the semaphore inter-process sharing is on	Disappear, unless they reside in still open shared memory and the semaphore inter-process sharing is on
POSIX mutexes, cond. vars, rw locks	As above	As above	As above