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**