# Mechanizmy POSIX IPC

| | Kolejki komunikatów | Pamięć wspólna | Semafory |
|---|---|---|---|
| Plik nagłówkowy | **<mqueue.h>** | **<sys/mman.h>** | **<semaphore.h>** |
| Tworzenie/ otwieranie/usuwanie | **mq_open(), mq_close(), mq_unlink()** | **shm_open(), shm_unlink()** | **sem_open() sem_close(), sem_unlink(), sem_init(), sem_destroy()** |
| Operacje sterujące | **mq_getattr(), mq_setattr()** | **ftruncate(), fstat()** | |
| Operacje komunikacji | **mq_send() mq_receive(), mq_notify()** | **mmap() munmap()** | **sem_wait(), sem_trywait(), sem_post(), sem_getvalue()** |

- Trwałość obiektów POSIX IPC to tzw. **trwałość jądra** za wyjątkiem **semafora w pamięci**, który ma **trwałość procesu** (*proces persistence*) – obiekt istnieje tak długo jak jest dostępna procesom pamięć w której przebywa, chyba że zostanie wcześniej jawnie zniszczony (**sem_destroy()** )

# Semafory nazwane POSIX – przestrzenie nazw i identyfikatorów

- Nazwane semafory POSIX są związane z nazwą (argument **name** wywołania funkcji **sem_open()**).
  - POSIX nie wymaga, by nazwa była widoczna w systemie plików czy była dostępna dla funkcji systemowych korzystających z nazw ścieżkowych.
  - Parametr **name** musi spełniać wymagania nazwy ścieżkowej (*pathname*).
    - Jeśli **name** rozpoczyna znak ‚/', to każdy proces wywołujący **sem_open( )** z taką nazwą wskazuje na ten sam semafor – póki nie zostanie usunięty z systemu,
    - Jeśli **name** nie rozpoczyna znak ‚/' – konsekwencje zależą od implementacji.
    - Konsekwencje wielokrotnego wystąpienia w nazwie znaku ‚/' zależą od implementacji.
- Linux wymaga nazw postaci **/somename** (nie dłuższych niż {NAMELEN-4} bajtów). Semafory nazwane przechowywane są w wirtualnym systemie plików, normalnie montowanym pod **/dev/shm**, przy czym nazwy mają postać **sem.somename**.
- Dokumentacja Linux semaforów POSIX: **sem_overview(7)**

# Semafory nazwane – tworzenie, otwieranie

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

Funkcja zwraca wskaźnik na strukturę semafora; w przypadku błędu zwraca **(sem_t)(SEM_FAILED)**, po ustawieniu kodu błędu w **errno** .

Parametry:

**name** - nazwa semafora.

**oflag** - określa tryb dostępu (**O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL**). Jeżeli tworzony jest nowy semafor wymagane są dwa dodatkowe parametry wywołania:

**mode** - prawa dostępu (**r** i **w** – jak dla plików)

**value** - początkowa wartość semafora

**Uwagi**:

- Wywołanie **sem_open()** jest przerywane przez asynchroniczną obsługę sygnału w procesie wywołującym **sem_open()** .
- Maks. liczba semaforów: **{SEM_NSEM_MAX}** (>=256), maks. Wartość semafora**: {SEM_VALUE_MAX}** (>=32767), patrz **<unistd.h>**

# Odłączenie/usunięcie semafora

- Jeśli proces nie potrzebuje dostępu do semafora powinien go zamknąć:

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

- Semafor o danej nazwie (**name**) można zaznaczyć do usunięcia: :

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

**Uwaga:** Funkcja usuwa natychmiast jedynie nazwę semafora z systemu, ale semafor jest naprawdę usunięty, gdy zostanie odłączony (przez wywołanie **sem_close** ()) przez wszystkie procesy, które go wcześniej dołączyły.

# Operacje czekaj i sygnalizuj

- Blokujące i nieblokujące wykonanie operacji czekaj **wait** :

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

- Wykonanie operacji sygnalizuj:

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

- Aktualną wartość semafora można uzyskać przez wywołanie :

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

# Nienazwane semafory POSIX

- Nienazwane semafory są przechowywane w strukturze **sem_t** , dostęp do tej struktury musi być zorganizowany przez współpracujące wątki czy procesy. Inicjacja struktury z nadaniem wartości początkowej **value** :

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

Jeśli **shared==0** – to semafor jest współdzielony przez wątki jednego procesu; w przeciwnym przypadku jest współużytkowany przez procesy..

- Usunięcie semafora przechowywanego w strukturze danych wskazywanej przez **sem** :

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

Wykorzystywanie usuniętego semafora ma nieokreślone skutki – aż do ponownej inicjalizacji przez **sem_init(…)** .

- Operacje semaforowe (wait/post) wykonywane są przez te same funkcje (**sem_wait()** and **sem_post()** ) , które są używane dla semaforów nazwanych.

# Inne obiekty synchronizacji POSIX

- **Muteks/zamek** (*mutex*)
- **Zmienna warunku** (*Condition variable)*
- **Bariera** (*barrier*)
- **Zamek czytelników-pisarza/odczytu-zapisu** *(Read-Write Lock)*

# Inne obiekty synchronizacji POSIX: muteks

- **Muteks** (zamek). Obiekt synchronizacji, który umożliwia wykluczenie dostępu do sekcji krytycznej. Wątek, który zajął muteks staje się jego czasowym właścicielem. Tylko wątek-właściciel może zwolnić muteks – przez co inny wątek może muteks zająć.
- Podstawowe operacje:
  - Zajęcje muteksu (zablokowanie dostępu do sekcji krytycznej dla innych)

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

  - Zwolnienie muteksu (odblokowanie dostępu do sekcji krytycznej)

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

**Sposób użycia muteksu:**

```
lock
// sekcja krytyczna: kod, który powinien mieć wyłączny dostęp
// do współdzielonych danych
unlock
```

# Tworzenie i inicjacja muteksu

- Tworzenie muteksu o domyślnych atrybutach

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
```

- Nadawanie muteksów początkowych atrybutów

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

- Niszczenie muteksu

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

# Atrybuty muteksu

- Domyślna inicjacja struktury atrybutów muteksu

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

Informacja o odczycie/modyfikacji atrybutów: `man pthread_mutexattr_destroy`
Liczba atrybutów zależy od implementacji.

Niektóre atrybuty muteksów w systemie Linux (non-RT):

`pshared` muteks może (albo nie może) być współdzielony przez procesy
`type : NORMAL` muteks nie wykrywa blokady (deadlock) kiedy wątek próbuje zająć zajęty muteks
`ERRORCHECK` muteks sprawdzający poprawność użycia
`RECURSIVE` możliwe jest wielokrotne zajmowanie tego samego muteksu przez jeden wątek, ale wymaga to wielokrotnego odblokowowywania – by muteks stał się wolny

- Niszczenie (unieważnianie) struktury atrybutów

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

# Robust mutex

- Muteksy POSIX mogą mieć atrybut **robustness** (patrz **pthread_mutexattr_setrobust(3)**). Atrybut ten określa zachowanie funkcji obsługujących muteks, gdy zakończy się wątek, który nie zwolnił zajętego przez siebie muteksu.
- Jeśli muteks został inicjowany przez atrybut PTHREAD_MUTEX_ROBUST, a później wątek, który zajął ten muteks (stając się jego przejściowym „właścicielem”) zakończył się bez zwolnienia tego muteksu, to wszystkie następne próby wykonania funkcji **pthread_mutex_lock()** dla tego muteksu zawiodą, zwracając kod EOWNERDEAD, by zwrócić uwagę na to że wątek jest w stanie niespójnym (zajęty muteks ma nieistniejącego właściciela). Zwykle w tej sytuacji kandydat na właściciela powinien wywołać funkcję **pthread_mutex_consistent()** na muteksie, by naprawić jego stan – przed próbą wykonania jakiejkolwiek innej operacji.
- Jeśli kandydat na następnego właściciela spróbuje jednak zwolnić muteks przy pomocy wywołania funkcji **pthread_mutex_unlock()** - zanim naprawi jego stan – muteks stanie się trwale nieużyteczny („popsuty”). Wszystkie następne próby jego zajęcia przy pomocy wywołania **pthread_mutex_lock()** zawiodą z kodem błędu ENOTRECOVERABLE. Jedyna dozwolona (mogąca się wykonać poprawnie) operacja, to wywołanie dla popsutego muteksu funkcji **pthread_mutex_destroy( ).**

# Zmienna warunku

- **Zmienna warunku** – Obiekt synchronizacji, który pozwala wątkowi zawiesić wielokrotnie wykonanie, dopóki warunek związany ze zmienną stanie się prawdziwy. Wątek zawieszony w ten sposób nazywany jest zablokowanym przez zmienną warunku.”
- Tworzenie zmiennej warunku (CV) z domyślną inicjalizacją:

```
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
```

- Inicjalizacja CV:

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

- Niszczenie CV

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

# Zmienna warunku

- Zmienna warunku zawsze współpracuje z muteksem.

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

Wywołanie pthread_cond_wait() nierozdzielnie (atomowo):

- zwalnia **mutex** oraz
- rozpoczyna blokowanie wątku na zmiennej warunku poniższych.

Po pomyślnym powrocie z funkcji pthread_cond_wait() muteks jest ponownie zajęty przez wątek wywołujący.

- Wywołanie poniższych funkcji powoduje odblokowanie wątków zablokowanych na zmiennej warunku **cond**

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

Jeżeli aktualnie nie ma wątków zablokowanych na zmiennej warunku – powiadomienie o odblokowaniu nie powoduje żadnych skutków (teraz i w przyszłości).

# Schemat użycia cv+mutex

```
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 ) {/* sprawdzenie warunku */
        int ret = pthread_cond_wait (&cv , &mutex ) ;
        if ( ret ) { . . . . /* error */ }
}
. . .    /* główna część kodu sekcji krytycznej,
             wykonywana pod ochroną mutex-u */
pthread_mutex_unlock (&mutex ) ;
```

```
. . .
pthread_mutex_lock (&mutex ) ;
condition_is_false =0; /* zmiana warunku
                          pod ochroną mutex-u */
pthread_cond_signal (&cv );/* sygnalizacja */
pthread_mutex_unlock (&mutex ) ;
. . .
```

# Przykład: „Hello world” z 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);
}
```

# Przykład: „Hello world” z CV – c.d.

```
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);
}
```

# Inne obiekty synchronizacji POSIX

- **Bariera** – Obiekt synchronizacji, który pozwala zablokować pewną liczbę wątków w funkcji **pthread_barier_wait** () . Funkcja

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

przestaje blokować, gdy określona liczba wątków dotrze do blokady. Jeden z oczekujących wątków otrzymuje z funkcji wartość niezerową (**PTHREAD_BARRIER_SERIAL_THREAD**), a pozostałe: 0, po czym bariera zostaje ustawiona w stan początkowy (taki jak bezpośrednio po wywołaniu funkcji inicjacji: **pthread_barrier_init()**). Funkcja **pthread_barrier_destroy()** niszczy barierę. Patrz **man: thread_barrier_destroy(3P), thread_barrier_wait(3P)**

- **Zamek czytelników-pisarza/odczytu-zapisu (Multiple readers, single writer locks)** umożliwia wielu wątkom na jednoczesny dostęp do współdzielonej danej w trybie odczytu oraz wyłączny dostęp jednemu wątkowi w trybie zapisu. Wątki mogą należeć do jednego procesu, bądź różnych procesów. Ważne jest, by struktura reprezentująca zamek była dla wszystkich współpracujących wątków dostępna w trybie R/W. Patrz: **man pthread_rwlock_rdlock, pthread_rwlock_wrlock, pthread_rwlock_unlock**