Chapter 3

Multicore and Multiprocessor Systems: Part II

Pthread coordination mechanisms

Mutex and conditional variables

The Pthread standard supports four types of synchronization and coordination facilities:

1. `pthread_join()` we have seen this function above

Pthread coordination mechanisms

Mutex and conditional variables

The Pthread standard supports four types of synchronization and coordination facilities:

1. `pthread_join()` we have seen this function above
2. Mutex variable functions for handling mutexes as defined above

Pthread coordination mechanisms

Mutex and conditional variables

The Pthread standard supports four types of synchronization and coordination facilities:

1. `pthread_join()` we have seen this function above
2. Mutex variable functions for handling mutexes as defined above
3. Conditional Variable functions treat a conditional variable that can be used to indicate a certain event in which the threads are interested. Conditional variable may be used to implement semaphore like structures and triggers for special more complex situation that require the threads to act in a certain way.

Pthread coordination mechanisms

Mutex and conditional variables

The Pthread standard supports four types of synchronization and coordination facilities:

1. `pthread_join()` we have seen this function above
2. Mutex variable functions for handling mutexes as defined above
3. Conditional Variable functions treat a conditional variable that can be used to indicate a certain event in which the threads are interested. Conditional variable may be used to implement semaphore like structures and triggers for special more complex situation that require the threads to act in a certain way.
4. `pthread_once()` can be used to make sure that certain initializations are performed by one and only one thread when called by multiple ones.

Pthread coordination mechanisms

Mutex variables

Dynamic initialization:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
```

Static/Macro initialization:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Pthread coordination mechanisms

Mutex variables

Dynamic initialization:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
```

Static/Macro initialization:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- `mutex` is the mutex variable to be initialized

Pthread coordination mechanisms

Mutex variables

Dynamic initialization:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
```

Static/Macro initialization:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- `mutex` is the mutex variable to be initialized
- `attr` can be used to adapt the mutex properties, as for the pthreads `NULL` gives the default attributes,

Pthread coordination mechanisms

Mutex variables

Dynamic initialization:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
```

Static/Macro initialization:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- `mutex` is the mutex variable to be initialized
- `attr` can be used to adapt the mutex properties, as for the pthreads `NULL` gives the default attributes,
- `restrict` is a C99-standard keyword limiting the pointer aliasing features and guiding compilers and aiding in the caching optimization.

Pthread coordination mechanisms

Mutex variables

Dynamic initialization:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
```

Static/Macro initialization:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- `mutex` is the mutex variable to be initialized
- `attr` can be used to adapt the mutex properties, as for the pthreads `NULL` gives the default attributes,
- `restrict` is a C99-standard keyword limiting the pointer aliasing features and guiding compilers and aiding in the caching optimization.
- initialization may fail if the system has insufficient memory (error code `ENOMEM`) or other resources (`EAGAIN`)

Pthread coordination mechanisms

Mutex variables

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

- If `mutex` is unlocked the function returns with the mutex in locked state,

Pthread coordination mechanisms

Mutex variables

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

- If `mutex` is unlocked the function returns with the mutex in locked state,
- If `mutex` is already locked the execution is blocked until the lock is released and it can proceed as above,

Pthread coordination mechanisms

Mutex variables

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

- If `mutex` is unlocked the function returns with the mutex in locked state,
- If `mutex` is already locked the execution is blocked until the lock is released and it can proceed as above,
- Four types of mutexes are defined:
  - `PTHREAD_MUTEX_NORMAL`
  - `PTHREAD_MUTEX_ERRORCHECK`
  - `PTHREAD_MUTEX_RECURSIVE`
  - `PTHREAD_MUTEX_DEFAULT`

  All of them show different behavior when locked mutexes should again be locked by the same thread or a thread tries to unlock a previously unlocked mutex and similar unintended situations. This especially regards error handling and deadlock detection.

Pthread coordination mechanisms

Mutex variables

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

- The function is equivalent to `pthread_mutex_lock()`, except that it returns immediately in any case.

Pthread coordination mechanisms

Mutex variables

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

- The function is equivalent to `pthread_mutex_lock()`, except that it returns immediately in any case.
- Success or failure are determined from the return value.

Pthread coordination mechanisms

Mutex variables

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

- The function is equivalent to `pthread_mutex_lock()`, except that it returns immediately in any case.
- Success or failure are determined from the return value.
- If the mutex type is `PTHREAD_MUTEX_RECURSIVE` the lock count is increased by one and the function returns success.

Pthread coordination mechanisms

Mutex variables

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

- the function releases the lock

Pthread coordination mechanisms

Mutex variables

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

- the function releases the lock
- what exactly “release” means, depends on the properties of the mutex variable

Pthread coordination mechanisms

Mutex variables

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

- the function releases the lock
- what exactly “release” means, depends on the properties of the mutex variable
- e.g., for type `PTHREAD_MUTEX_RECURSIVE` mutexes it means that the counter is decreased by one and they become available once it reaches zero

Pthread coordination mechanisms

Mutex variables

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

- the function releases the lock
- what exactly “release” means, depends on the properties of the mutex variable
- e.g., for type `PTHREAD_MUTEX_RECURSIVE` mutexes it means that the counter is decreased by one and they become available once it reaches zero
- if the mutex becomes available, i.e., unlocked by the function call and there are blocked threads waiting for it, the threading policy decides which thread acquires `mutex` next.

Pthread coordination mechanisms

Mutex variables

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

- destroys the mutex referenced by `mutex`

Pthread coordination mechanisms

Mutex variables

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

- destroys the mutex referenced by `mutex`
- the destroyed mutex then becomes uninitialized

Pthread coordination mechanisms

Mutex variables

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

- destroys the mutex referenced by `mutex`
- the destroyed mutex then becomes uninitialized
- `pthread_mutex_init()` can be used to initialize the same mutex variable again

Pthread coordination mechanisms

Mutex variables

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

- destroys the mutex referenced by `mutex`
- the destroyed mutex then becomes uninitialized
- `pthread_mutex_init()` can be used to initialize the same mutex variable again
- if `mutex` is locked or referenced `pthread_mutex_destroy()` fails with error code `EBUSY`

Pthread coordination mechanisms

Avoiding mutex triggered deadlocks

Example (A deadlock situation when locking multiple mutexes)

Problem:

- Consider two mutex variables `ma` and `mb`, as well as two threads T1 and T2.

Pthread coordination mechanisms

Avoiding mutex triggered deadlocks

Example (A deadlock situation when locking multiple mutexes)

Problem:

- Consider two mutex variables `ma` and `mb`, as well as two threads T1 and T2.
- T1 locks `ma` first and then `mb`,

Pthread coordination mechanisms

Avoiding mutex triggered deadlocks

Example (A deadlock situation when locking multiple mutexes)

Problem:

- Consider two mutex variables `ma` and `mb`, as well as two threads T1 and T2.
- T1 locks `ma` first and then `mb`,
- T2 locks `mb` first and then `ma`,

Pthread coordination mechanisms

Avoiding mutex triggered deadlocks

Example (A deadlock situation when locking multiple mutexes)

Problem:

- Consider two mutex variables `ma` and `mb`, as well as two threads T1 and T2.
- T1 locks `ma` first and then `mb`,
- T2 locks `mb` first and then `ma`,
- In case T1 is interrupted by the scheduler after locking `ma`, but before locking `mb` and in the meantime T2 succeeds in locking it, then the classical deadlock occurs.

Pthread coordination mechanisms

Avoiding mutex triggered deadlocks

Example (A deadlock situation when locking multiple mutexes)

Locking hierarchy solution:
The basic idea here is that all threads need to lock the critical mutexes in the same order. This can easily be guaranteed by hierarchically ordering the mutexes.

Pthread coordination mechanisms

Avoiding mutex triggered deadlocks

Example (A deadlock situation when locking multiple mutexes)

Back off strategy solution:
When we want to keep the differing locking orders, we may use `pthread_mutex_trylock()` with a back off strategy.

Pthread coordination mechanisms

Avoiding mutex triggered deadlocks

Example (A deadlock situation when locking multiple mutexes)

Back off strategy solution:
When we want to keep the differing locking orders, we may use `pthread_mutex_trylock()` with a back off strategy.

- Locking is tried in the desired order,

Pthread coordination mechanisms

Avoiding mutex triggered deadlocks

Example (A deadlock situation when locking multiple mutexes)

Back off strategy solution:
When we want to keep the differing locking orders, we may use `pthread_mutex_trylock()` with a back off strategy.

- Locking is tried in the desired order,
- when a trylock fails, the thread unlocks all previously locked mutexes (it backs off of the protected resources),

Pthread coordination mechanisms

Avoiding mutex triggered deadlocks

Example (A deadlock situation when locking multiple mutexes)

Back off strategy solution:
When we want to keep the differing locking orders, we may use `pthread_mutex_trylock()` with a back off strategy.

- Locking is tried in the desired order,
- when a trylock fails, the thread unlocks all previously locked mutexes (it backs off of the protected resources),
- after the back off it starts over from the first one.

Pthread coordination mechanisms

Conditional variables

Dynamic initialization:

```
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
```

Static/Macro initialization:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- `cond` the condition to be initialized

Pthread coordination mechanisms

Conditional variables

Dynamic initialization:

```
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
```

Static/Macro initialization:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- `cond` the condition to be initialized
- `attr` can be used to adapt the condition properties, as for the pthreads `NULL` gives the default attributes,

Pthread coordination mechanisms

Conditional variables

Dynamic initialization:

```
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
```

Static/Macro initialization:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- `cond` the condition to be initialized
- `attr` can be used to adapt the condition properties, as for the pthreads `NULL` gives the default attributes,
- `restrict` is a C99-standard keyword limiting the pointer aliasing features and guiding compilers and aiding in the caching optimization,

Pthread coordination mechanisms

Conditional variables

Dynamic initialization:

```
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
```

Static/Macro initialization:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- `cond` the condition to be initialized
- `attr` can be used to adapt the condition properties, as for the pthreads `NULL` gives the default attributes,
- `restrict` is a C99-standard keyword limiting the pointer aliasing features and guiding compilers and aiding in the caching optimization,
- every condition variable is associated to a mutex.

Pthread coordination mechanisms

Conditional variables

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

- destroys the condition variable referenced by `cond`

Pthread coordination mechanisms

Conditional variables

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

- destroys the condition variable referenced by `cond`
- the destroyed condition then becomes uninitialized

Pthread coordination mechanisms

Conditional variables

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

- destroys the condition variable referenced by `cond`
- the destroyed condition then becomes uninitialized
- `pthread_cond_init()` can reinitialize the same condition variable

Pthread coordination mechanisms

Conditional variables

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

- destroys the condition variable referenced by `cond`
- the destroyed condition then becomes uninitialized
- `pthread_cond_init()` can reinitialize the same condition variable
- if `cond` is blocking threads when destroyed the standard does not specify the behavior of `pthread_cond_destroy()`.

Pthread coordination mechanisms

Conditional variables

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

- assumes that `mutex` was locked before by the calling thread,

Pthread coordination mechanisms

Conditional variables

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

- assumes that `mutex` was locked before by the calling thread,
- results in the thread getting blocked and at the same time (atomically) releasing `mutex`

Pthread coordination mechanisms

Conditional variables

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

- assumes that `mutex` was locked before by the calling thread,
- results in the thread getting blocked and at the same time (atomically) releasing `mutex`
- another thread may evaluate this to wake up the now blocked thread (see `pthread_cond_signal()`)

Pthread coordination mechanisms

Conditional variables

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

- assumes that `mutex` was locked before by the calling thread,
- results in the thread getting blocked and at the same time (atomically) releasing `mutex`
- another thread may evaluate this to wake up the now blocked thread (see `pthread_cond_signal()`)
- upon waking up the thread automatically tries to gain access to `mutex` again,

Pthread coordination mechanisms

Conditional variables

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

- assumes that `mutex` was locked before by the calling thread,
- results in the thread getting blocked and at the same time (atomically) releasing `mutex`
- another thread may evaluate this to wake up the now blocked thread (see `pthread_cond_signal()`)
- upon waking up the thread automatically tries to gain access to `mutex` again,
- if it succeeds it should test the condition again to check whether another thread changed it in the meantime.

Pthread coordination mechanisms

Conditional variables

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

- if no thread is blocked on the condition variable `cond` there is no effect,
- otherwise, one of the waiting threads is woken up and proceeds as described above.

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

- wakes up all threads blocking on `cond`,
- all of them try to acquire the associated mutex,
- only one of them can succeed,
- the others get blocked on the mutex now.

Pthread coordination mechanisms

Conditional variables

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime);
```

- equivalent to `pthread_cond_wait()` except that it only blocks for the period specified by `abstime`,

Pthread coordination mechanisms

Conditional variables

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime);
```

- equivalent to `pthread_cond_wait()` except that it only blocks for the period specified by `abstime`,
- if the thread did not get signaled or broadcast before `abstime` expires it returns with error code `ETIMEDOUT`.

Pthread coordination mechanisms

A counting semaphore for Pthreads

Semaphores are not available in the POSIX Threads standard.

Pthread coordination mechanisms

A counting semaphore for Pthreads

Semaphores are not available in the POSIX Threads standard.

However, they can be created using the existing mechanisms of mutexes and conditions.

Pthread coordination mechanisms

A counting semaphore for Pthreads

Semaphores are not available in the POSIX Threads standard.

However, they can be created using the existing mechanisms of mutexes and conditions.

A counting semaphore should be a data type that acts like a counter with non-negative values and for which two operations are defined:

1. A signal operation increments the counter and wakes up a task blocked on the semaphore if one exists.
2. A wait operation simply decrements the counter if it is positive. If it was zero already the thread is blocking on the semaphore.

Pthread coordination mechanisms

A counting semaphore for Pthreads: Data type, Init and Cleanup

- data structure for the semaphore:

```
typedef struct _sema_t{
  int count;
  pthread_mutex_t m;
  pthread_cond_t c;
} sema_t;
```

- the initialization

```
void InitSema(sema_t *ps){
  pthread_mutex_init(&ps->m,NULL);
  pthread_cond_init(&ps->c,NULL);
}
```

- and the cleanup

```
void CleanupSema(void *arg){
  pthread_mutex_unlock((pthread_mutex_t *) arg);
}
```

---

source: [Rauber/Rünger'10]

Pthread coordination mechanisms

A counting semaphore for Pthreads: Operation realization

```
void ReleaseSema(sema_t *ps){ // signal operation
  pthread_mutex_lock(&ps->m) ;
  pthread_cleanup_push(CleanupSema,&ps->m);
  {
    ps->count++;
    pthread_cond_signal(&ps->c) ;
  }
  pthread_cleanup_pop ( 1 ) ;
}


void AcquireSema(sema_t *ps){ // wait operation
  pthread_mutex_lock(&ps->mutex);
  pthread_cleanup_push(CleanupSema,&ps->m);
  {
    while(ps->count==0)
      pthread_cond_wait(&ps->c,&ps->m) ;
    ps->count--;
  }
  pthread_cleanup_pop(1);
}
```

---

source: [Rauber/Rünger'10]

Pthread coordination mechanisms

A typical application example for semaphores

Example (Producer/Consumer queue buffer protection)

- A buffer of fixed size n is shared by
- a producer thread generating entries and storing them in the buffer if it is not full,
- a consumer thread removing entries from the same buffer for further processing unless it is empty.

For the realization of the protected access two semaphores are required:

1. Number of entries occupied (initialized by 0),
2. Number of free entries (initialized by n).

The Mechanism works for an arbitrary number of producers and consumers. (Details will be worked out on exercise sheet 2.)

Task Pools

Coordination models for the cooperation of threads

1. Master/Slave model:
    - A master thread is controlling the execution of the program,
    - the slave threads are executing the work.

Task Pools

Coordination models for the cooperation of threads

1. Master/Slave model:
   - A master thread is controlling the execution of the program,
   - the slave threads are executing the work.
2. Client/Server model:
   - Client threads produce requests,
   - Server threads execute the corresponding work.

Task Pools

Coordination models for the cooperation of threads

1. Master/Slave model:
    - A master thread is controlling the execution of the program,
    - the slave threads are executing the work.
2. Client/Server model:
    - Client threads produce requests,
    - Server threads execute the corresponding work.
3. Pipeline model:
    - Every thread (except for the first and last in line) produces output that serves as input for another thread,
    - after a startup phase (filling the pipeline) the parallel execution is achieved.

Task Pools

Coordination models for the cooperation of threads

1. Master/Slave model:
   - A master thread is controlling the execution of the program,
   - the slave threads are executing the work.
2. Client/Server model:
   - Client threads produce requests,
   - Server threads execute the corresponding work.
3. Pipeline model:
   - Every thread (except for the first and last in line) produces output that serves as input for another thread,
   - after a startup phase (filling the pipeline) the parallel execution is achieved.
4. Worker model:
   - equally privileged workers organize their workload,
   - an important variant is the task pool treated as detailed example next.

Task Pools

Basic idea of the task pool

Idea:

Creation of a parallel threaded program that can dynamically schedule tasks on the available processors.

Task Pools

Basic idea of the task pool

Idea:

Creation of a parallel threaded program that can dynamically schedule tasks on the available processors.

Key ingredients in the approach are:

- usage of a fixed number of threads
- organization of the pending tasks in a task pool,
- threads fetch the tasks from the pool and execute them leading to a dynamic assignment of the work load.

Task Pools

Basic idea of the task pool

Idea:

Creation of a parallel threaded program that can dynamically schedule tasks on the available processors.

Key ingredients in the approach are:

- usage of a fixed number of threads
- organization of the pending tasks in a task pool,
- threads fetch the tasks from the pool and execute them leading to a dynamic assignment of the work load.

Main advantages

- automatic dynamic load balancing among the threads
- comparably small overhead for the administration of threads

Task Pools

Implementation of a basic task pool: Data structures

- data strucutre for one task:

```
typedef struct _work_t{
    void (*routine) (void*); //worker function to call
    void* arg ;
    struct _work_t *next;
} work_t ;
```

- data structure for the task pool:

```
typedef struct _tpool_t{
    int num_threads ; // number of threads
    int max_size, curr_size; // max./cur. number of tasks in pool
    pthread_t *threads; //array of threads
    work_t *head , *tail; // start/end of the task queue
    pthread_mutex_t lock; //access control for the task pool
    pthread_cond_t not_empty ; // tasks are available
    pthread_cond_t not_full ; // tasks may be added
} tpool_t ;
```

---

source: [Rauber/Rünger'10]

Task Pools

Implementation of a basic task pool: Initialization

```
tpool_t *tpool_init(int num_threads , int max_size){
  int i;
  tpool_t *tpl;

  tpl=(tpool_t *) malloc (sizeof(tpool_t));
  tpl->num_threads=num_threads ;
  tpl->max_size=max_size ;
  tpl->cur_size=0;
  tpl->head=tpl->tail=NULL;

  pthread_mutex_init(&tpl->lock, NULL);
  pthread_cond_init(&tpl->not_empty, NULL);
  pthread_cond_init(&tpl->not_full, NULL);
  tpl->threads=(pthread_t *) malloc(num_threads *sizeof(pthread_t));
  for(i=0; i<num_threads; i++)
    pthread_create(tpl->threads+i, NULL, tpool_thread, (void *)tpl) ;
  return tpl;
}
```

---

source: [Rauber/Rünger'10]

Task Pools

Implementation of a basic task pool: Worker Threads

```
void *tpool_thread(void *vtpl){
  tpool_t *tpl=(tpool_t *) vtpl;
  work_t *wl ;

  for ( ; ; ) {
    pthread_mutex_lock(&tpl->lock);
    while(tpl->cur_size==0)
      pthread_cond_wait(&tpl->not_empty , &tpl->lock);
    wl=tpl->head; tpl->cur_size--;
    if(tpl->cur_size==0)
      tpl->head=tpl->tail=NULL;
    else tpl->head = wl->next;
    if (tpl->cur_size==tpl->max_size-1) // pool full
      pthread_cond_signal(&tpl->not_full);
    pthread_mutex_unlock(&tpl->lock);
    (*(wl->routine)) (wl->arg);
    free(wl);
  }
}
```

---

source: [Rauber/Rünger'10]

Task Pools

Implementation of a basic task pool: Task insertion

```
void tpool_insert(tpool_t *tpl, void(*f) (void*), void *arg){
  work_t *wl ;

  pthread_mutex_lock(&tpl->lock);
  while(tpl->cur_size==tpl->max_size)
    pthread_cond_wait(&tpl->not_full, &tpl->lock);
  wl=(work_t *) malloc(sizeof(work_t));
  wl->routine=f; wl->arg=arg; wl->next=NULL ;
  if( tpl->cur_size==0){
    tpl->head=tpl->tail=wl;
    pthread_cond_signal(&tpl->not_empty);
  }
  else{
    tpl->tail->next=wl; tpl->tail=wl;
  }
  tpl->cur_size++;
  pthread_mutex_unlock(&tpl->lock);
}
```

---

source: [Rauber/Rünger'10]

Shared Memory Blocks

General shared memory blocks

In contrast to Threads, different processes do not share their address space. Therefore, different ways to communicate in multiprocessing applications are necessary.

One possible way are shared memory objects. Unix-like operating systems provide at least one of:

- old System V Release 4 (SVR4) Shared Memory[2]
- new POSIX Shared Memory[3].

Both techniques implement shared memory objects, like common memory, semaphores and message queues, which are accessible from different applications with different address spaces.

---

[2] *System V Interface Definition, AT&T Unix System Laboratories, 1991*

[3] *IEEE Std 1003.1-2001 Portable Operating System Interface System Interfaces*

Shared Memory Blocks

POSIX Shared Memory

Common Memory Locations

- are used to share data between applications,
- are managed by the kernel and not by the application,
- each location is represented as a file in `/dev/shm/`,
- handled like normal files,
- created using `shm_open` and mapped to the memory using `mmap`,
- exist as long as no application deletes them, even when the creating program exits they stay available,
- seed manpage `man 7 shm_overview`.

Shared Memory Blocks

POSIX Shared Memory

POSIX Semaphores

- counting semaphores available form different address spaces,
- correspond to `pthread_mutex_*` in threaded applications,
- represented as a file in `/dev/shm/sem.*`,
- see manpage `man 7 sem_overview`.

Message Queues

- generalized Signal concept which can transfer a small payload (2 to 4 KiB),
- correspond to `pthread_cond_*` in threaded applications,
- can be represented as file in `/dev/mqueue`,
- see manpage `man 7 mq_overview`.