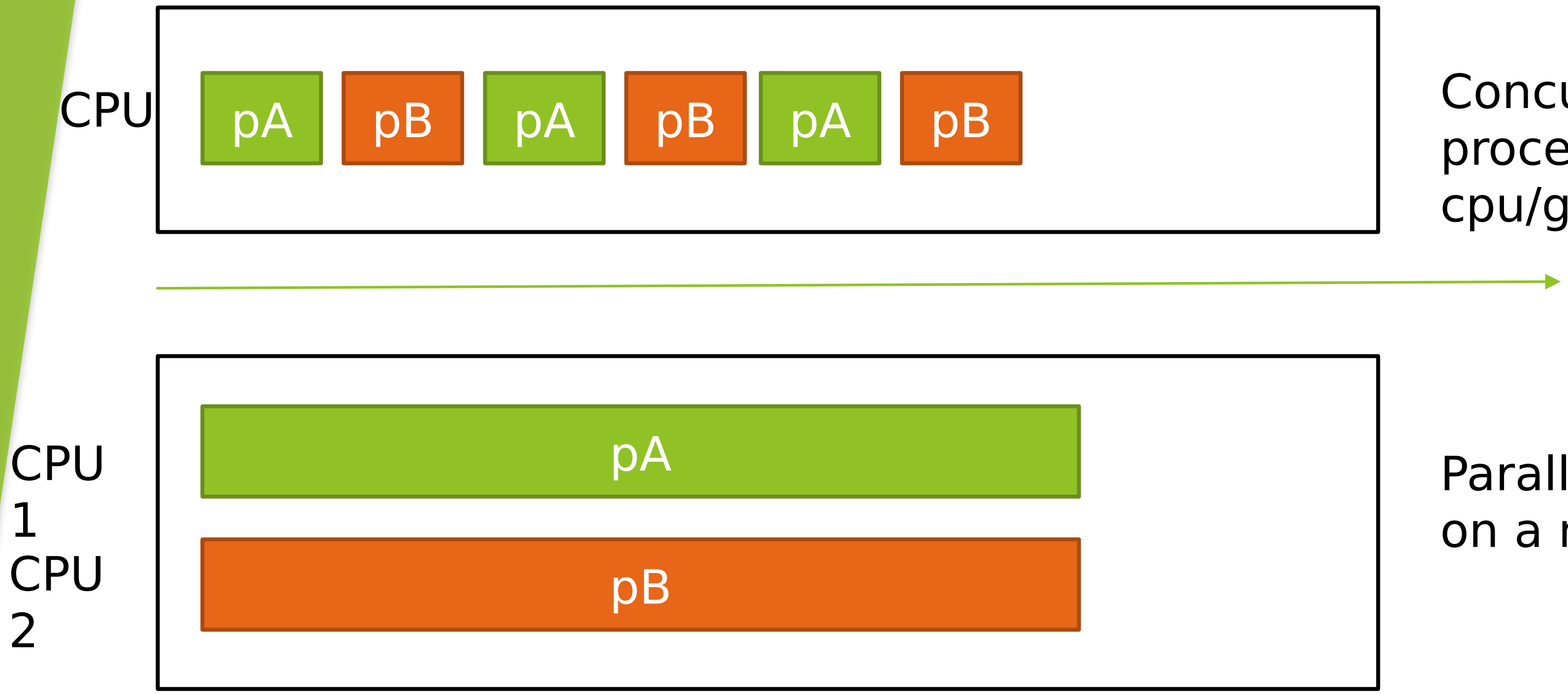
Operating Systems

Concurrency and Parallelism

Concurrency vs Parallelism



CPU

pA pB pA pB pA pB

CPU 1

pA

CPU 2

pB

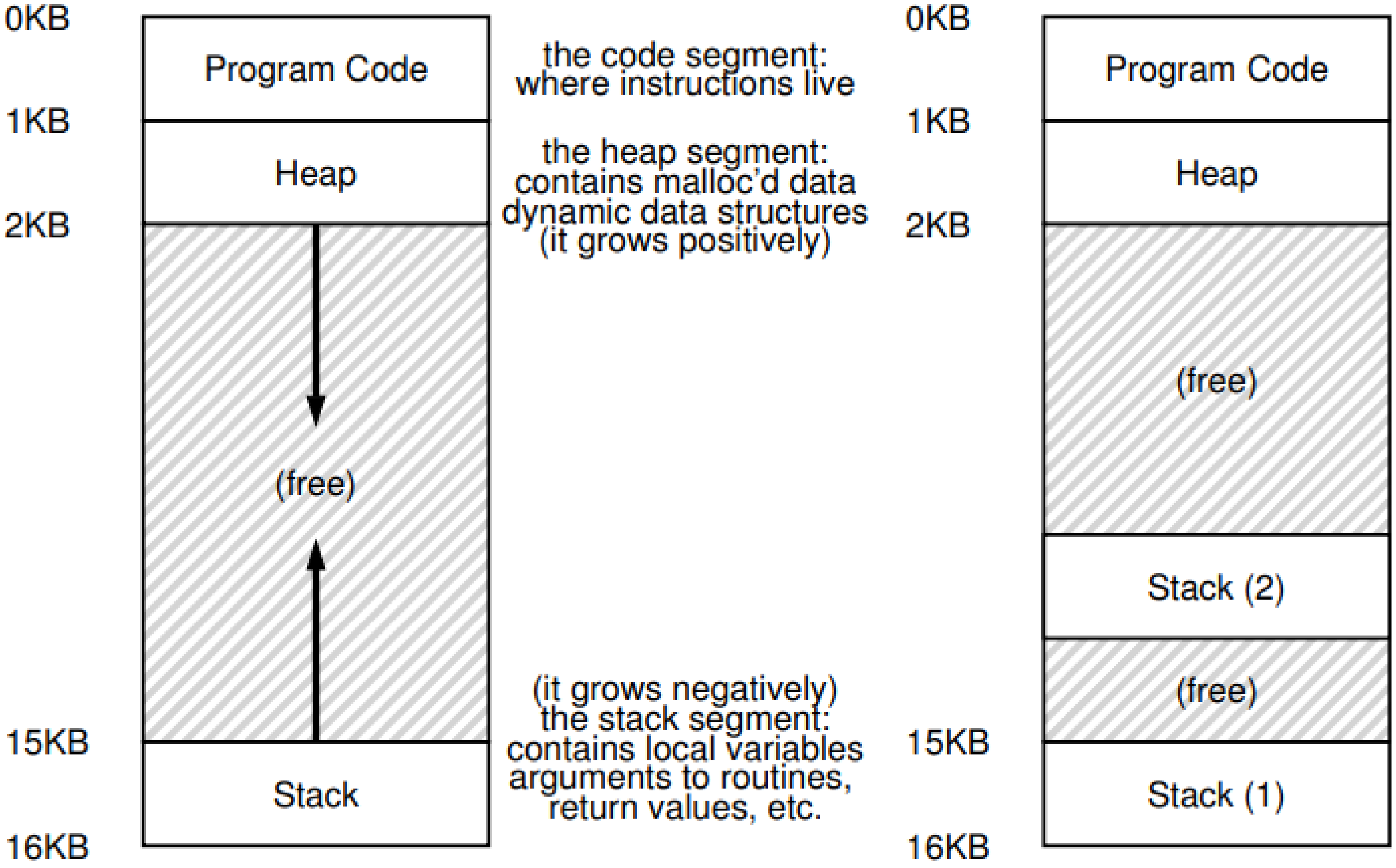
Concurrency: multiple processes/threads on a single cpu/gpu/soc

Parallelism: multiple processes/threads on a multiple cpu/gpu/soc

Concurrency

Process (syscall)	Thread (lib calls / pthread)
Heavy	Light
New virtual address space per process	New stack segment only
No implicit memory sharing (can still use mmap)	All other segments beside the stack are shared
Slow context switch	Fast context switch
Better separation/protection	No separation/protection between threads
Avoid app blocking on IO	Avoid app blocking on IO
Better protection against glitches/bugs/threats	If one thread fails it can block the full app

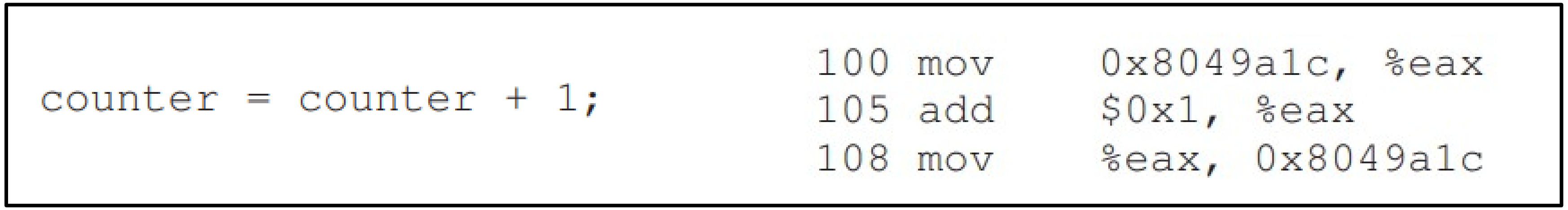
Single Threads vs Multiple Threads



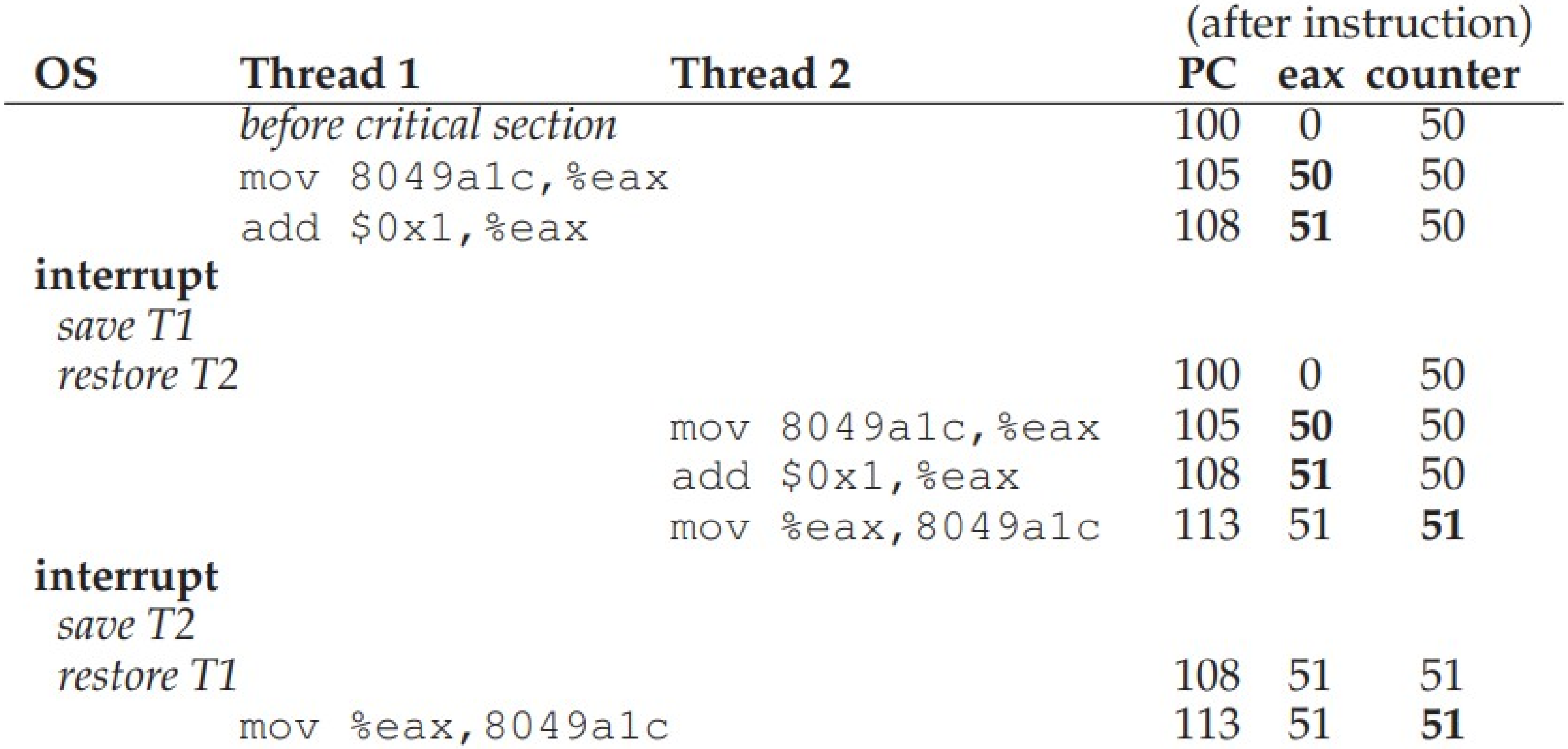
0KB
Program Code
the code segment: where instructions live
1KB
Heap
the heap segment: contains malloc'd data dynamic data structures (it grows positively)
2KB
(free)
15KB
Stack
(it grows negatively) the stack segment: contains local variables arguments to routines, return values, etc.
16KB
0KB
Program Code
1KB
Heap
2KB
(free)
Stack (2)
(free)
15KB
Stack (1)
16KB

Source: Textbook: https://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf

Critical Section



```
counter = counter + 1;          100 mov    0x8049a1c, %eax
                                105 add    $0x1, %eax
                                108 mov    %eax, 0x8049a1c
```



			(after instruction)		
OS	**Thread 1**	**Thread 2**	**PC**	**eax**	**counter**
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

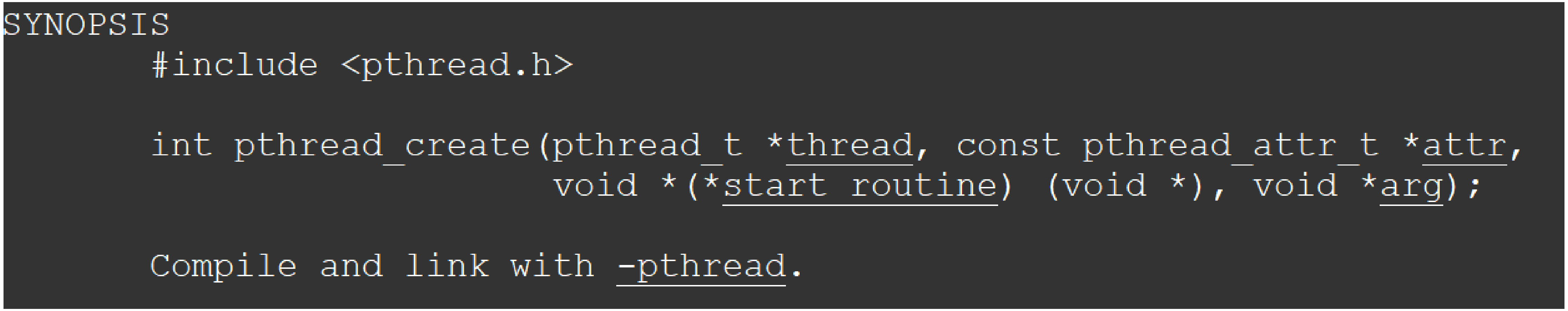
Source: Textbook: https://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf

LUT

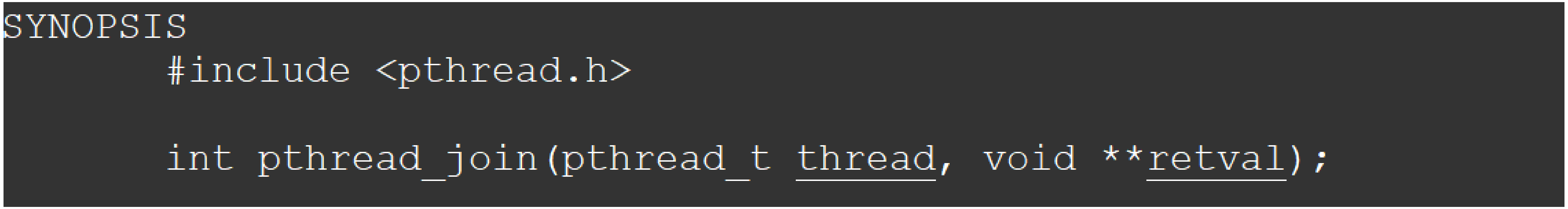
Name	Description
Critical section	A section of the code where a shared resource/variable is accessed
Race condition	Nondeterministic results; Result depends on the timing of the process/thread; Multiple threads access a critical section
Mutual exclusion	Guarantee that if one thread is executing a critical section, no other threads will interfere
Lock	

Threads API

Call/name	Description
pthread_create	Start a new thread in the calling process (man pthread_create)
pthread_join	Waits for a thread to terminate. (similar with the wait syscall used for processes)

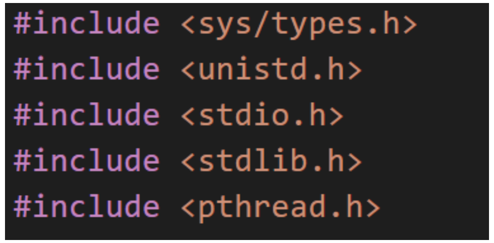


```
SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);

       Compile and link with -pthread.
```

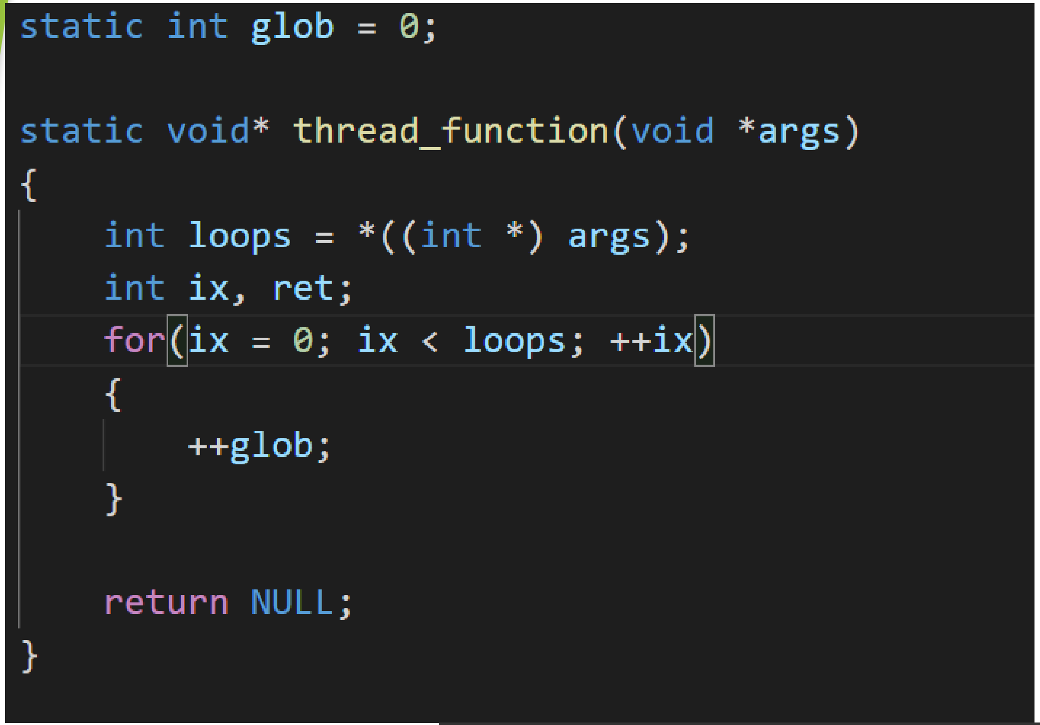


```
SYNOPSIS
       #include <pthread.h>

       int pthread_join(pthread_t thread, void **retval);
```

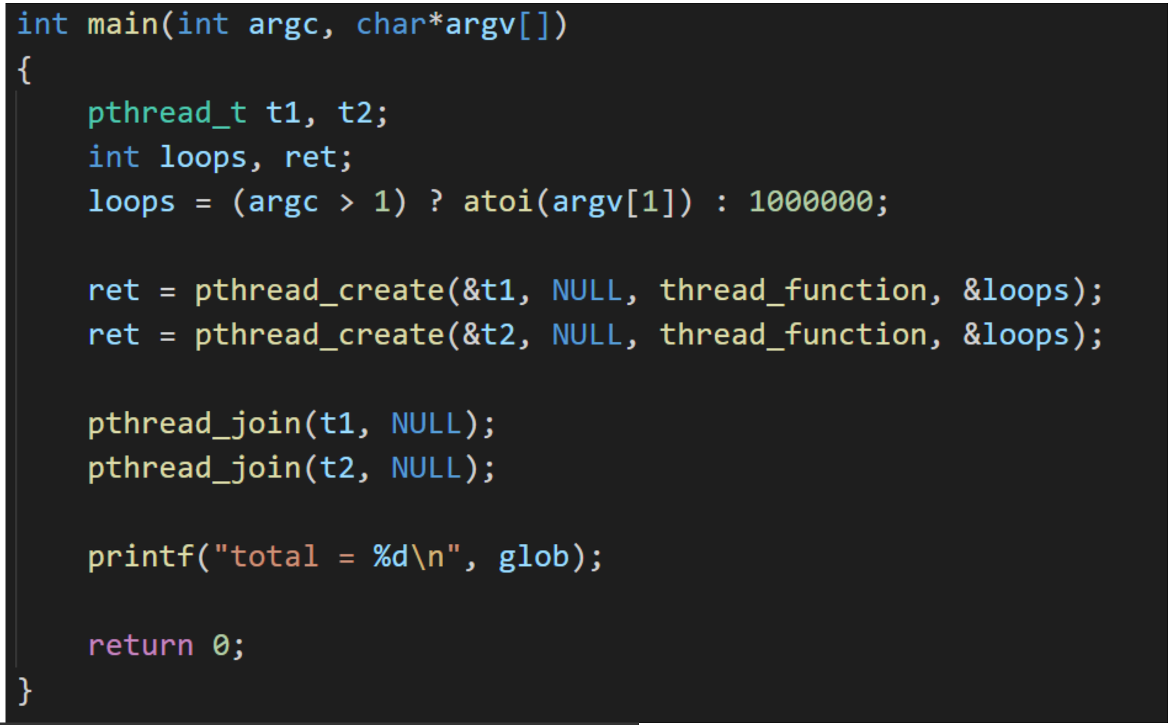
Threads API



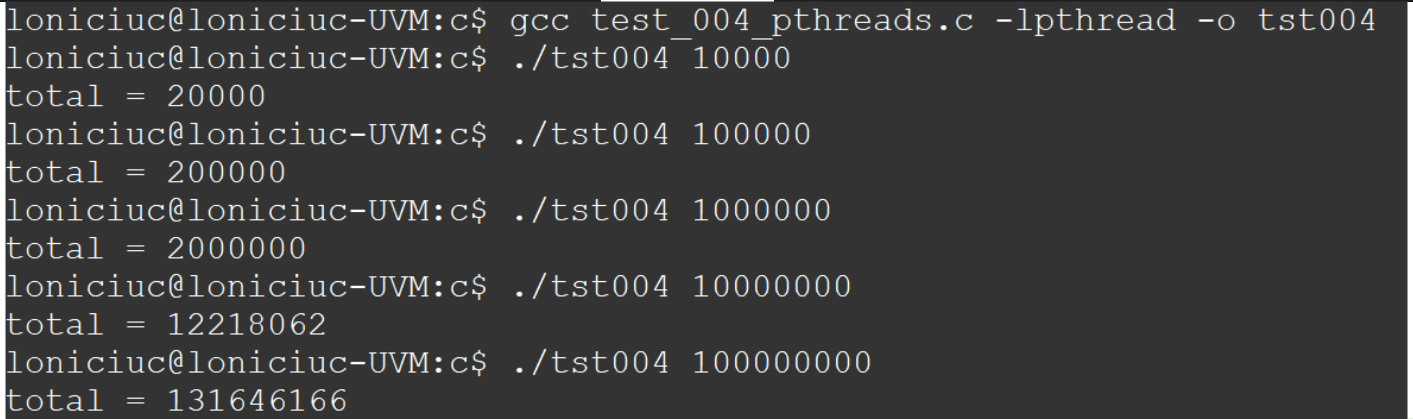
```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```



```c
static int glob = 0;

static void* thread_function(void *args)
{
    int loops = *((int *) args);
    int ix, ret;
    for(ix = 0; ix < loops; ++ix)
    {
        ++glob;
    }

    return NULL;
}
```



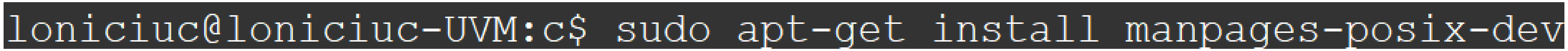
```c
int main(int argc, char*argv[])
{
    pthread_t t1, t2;
    int loops, ret;
    loops = (argc > 1) ? atoi(argv[1]) : 1000000;

    ret = pthread_create(&t1, NULL, thread_function, &loops);
    ret = pthread_create(&t2, NULL, thread_function, &loops);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("total = %d\n", glob);

    return 0;
}
```



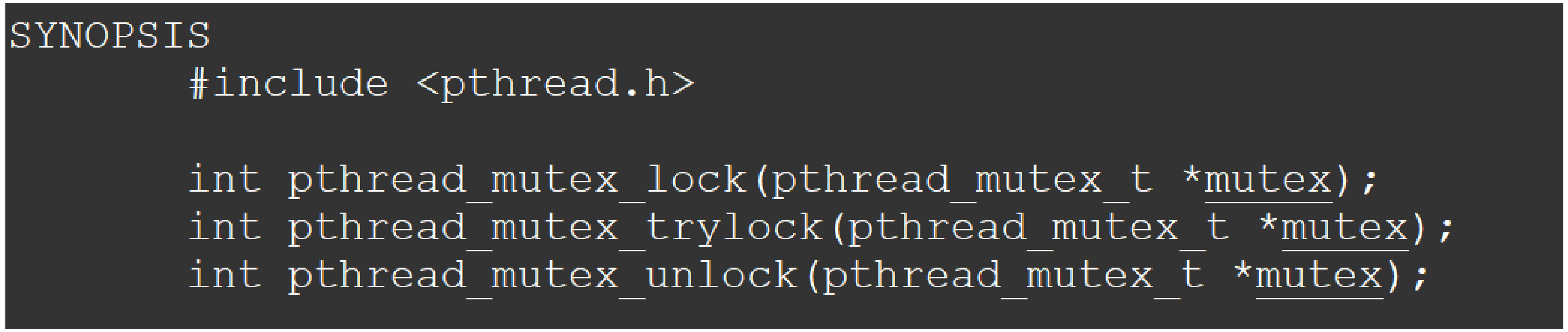
```
loniciuc@loniciuc-UVM:c$ gcc test_004_pthreads.c -lpthread -o tst004
loniciuc@loniciuc-UVM:c$ ./tst004 10000
total = 20000
loniciuc@loniciuc-UVM:c$ ./tst004 100000
total = 200000
loniciuc@loniciuc-UVM:c$ ./tst004 1000000
total = 2000000
loniciuc@loniciuc-UVM:c$ ./tst004 10000000
total = 12218062
loniciuc@loniciuc-UVM:c$ ./tst004 100000000
total = 131646166
```

Threads API

Call/name	Description
pthread_create	Start a new thread in the calling process (man pthread_create)
pthread_join	Join threads (similar with the wait syscall used for processes)
pthread_mutex_lock	Lock a mutex
pthread_mutex_trylock	If the mytex object is locked already, the call shall return immediately
pthread_mutex_unlock	Unclock a mutex



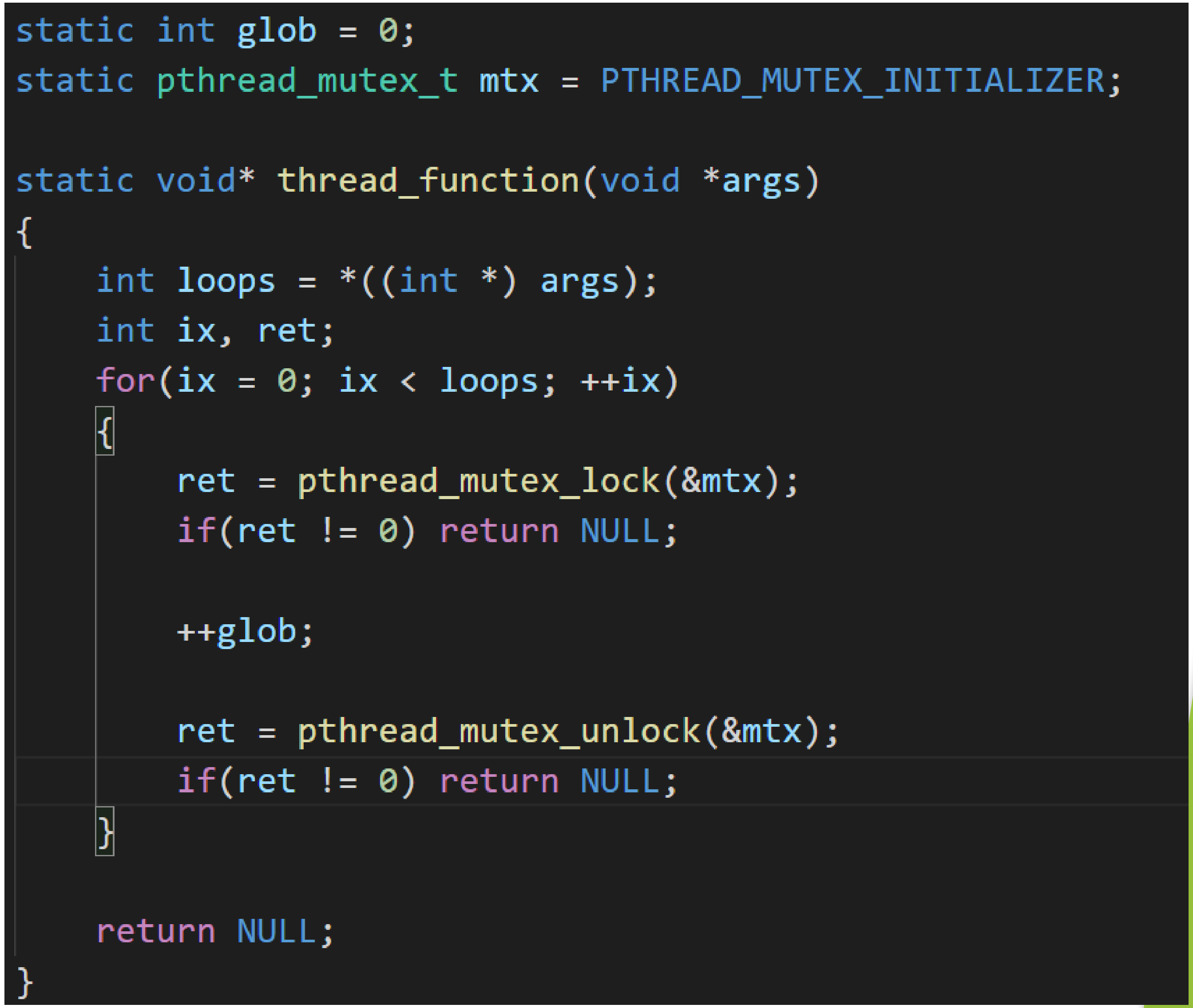
```
loniciuc@loniciuc-UVM:c$ sudo apt-get install manpages-posix-dev
```



```
SYNOPSIS
       #include <pthread.h>

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

Atomicity and Locks

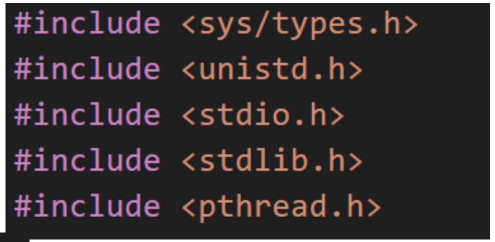
- For the code/app to be deterministic we need the critical section to be:
  - executed in one instruction (atomically)
    - OR
  - Create a process/thread lock (other thread can’t execute the critical section until the current thread releases the lock)



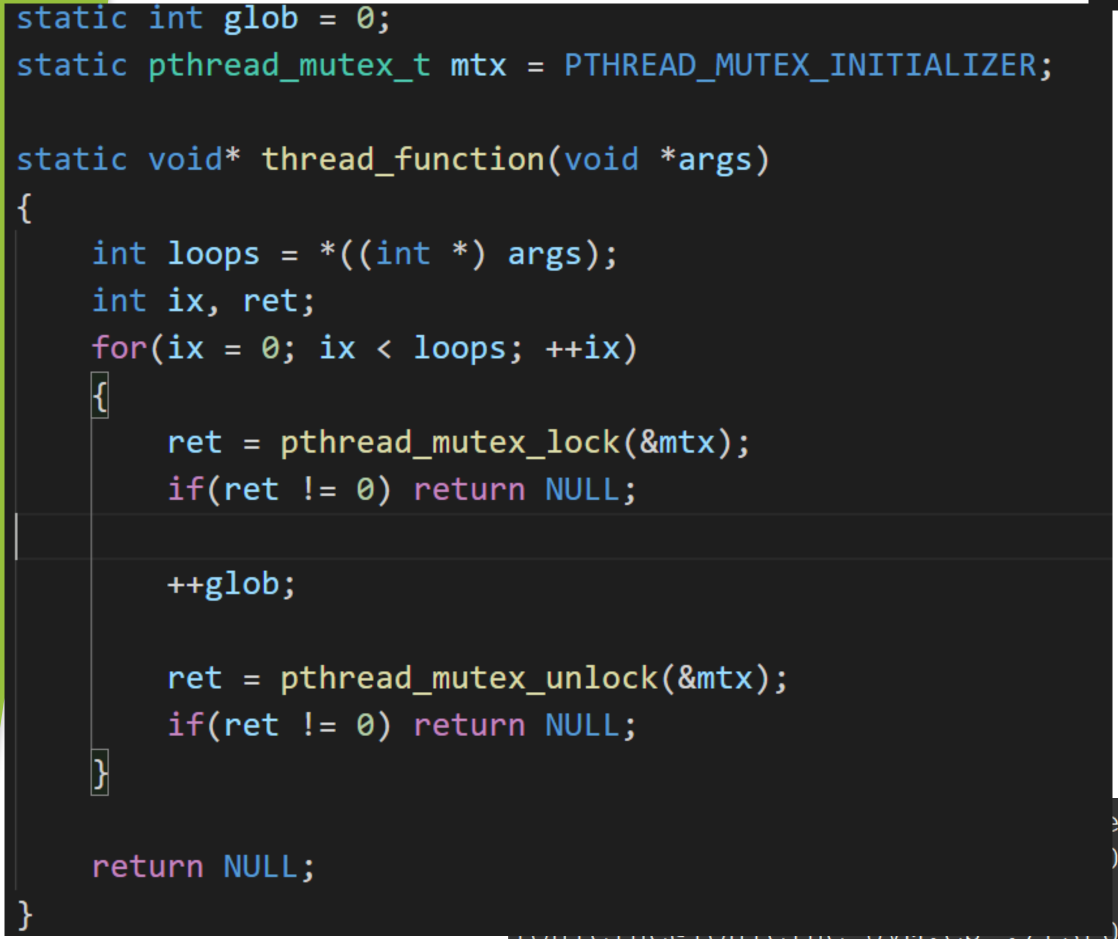
```c
static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void* thread_function(void *args)
{
    int loops = *((int *) args);
    int ix, ret;
    for(ix = 0; ix < loops; ++ix)
    {
        ret = pthread_mutex_lock(&mtx);
        if(ret != 0) return NULL;

        ++glob;

        ret = pthread_mutex_unlock(&mtx);
        if(ret != 0) return NULL;
    }

    return NULL;
}
```

gcc test_003_pthreads.c -lpthreads

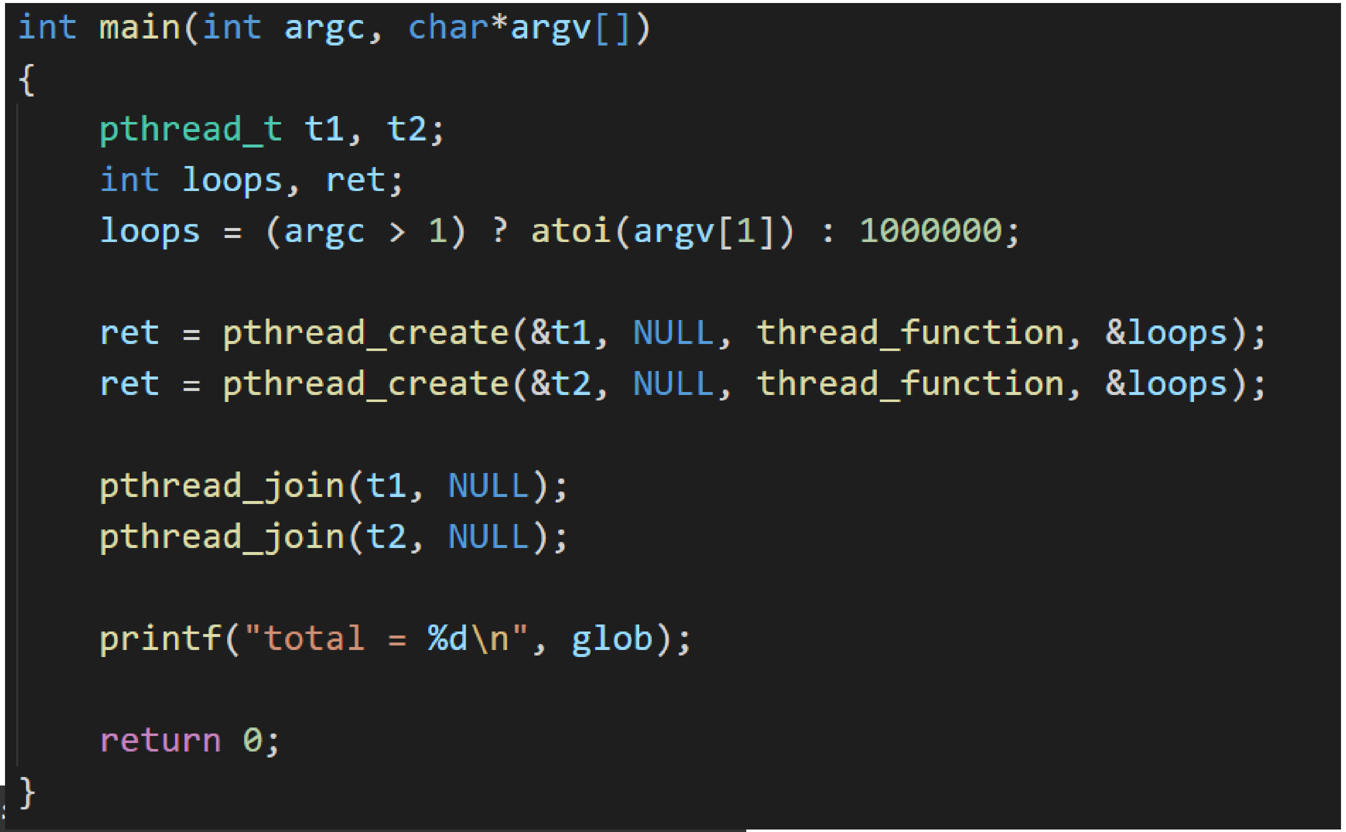
Locks API



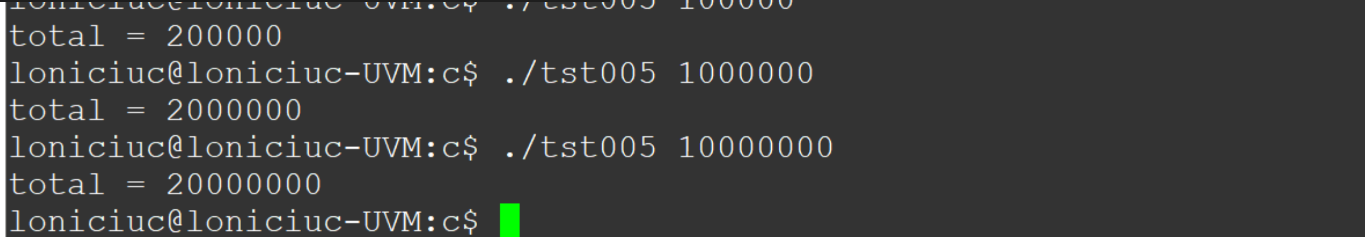
```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```



```
static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void* thread_function(void *args)
{
    int loops = *((int *) args);
    int ix, ret;
    for(ix = 0; ix < loops; ++ix)
    {
        ret = pthread_mutex_lock(&mtx);
        if(ret != 0) return NULL;

        ++glob;

        ret = pthread_mutex_unlock(&mtx);
        if(ret != 0) return NULL;
    }

    return NULL;
}
```



```
int main(int argc, char*argv[])
{
    pthread_t t1, t2;
    int loops, ret;
    loops = (argc > 1) ? atoi(argv[1]) : 1000000;

    ret = pthread_create(&t1, NULL, thread_function, &loops);
    ret = pthread_create(&t2, NULL, thread_function, &loops);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("total = %d\n", glob);

    return 0;
}
```



```
005 10000

loniciuc@loniciuc-UVM:c$ ./tst005 100000
total = 200000
loniciuc@loniciuc-UVM:c$ ./tst005 1000000
total = 2000000
loniciuc@loniciuc-UVM:c$ ./tst005 10000000
total = 20000000
loniciuc@loniciuc-UVM:c$
```

Hardware Support for Building Locks: Test-And-Set

Description	int TestAndSet(int *old_ptr, int new) {
	    int original = *old_ptr; // fetch original value at old_ptr
	    *old_ptr = new; // store new into old_ptr
	    return original; // return the original value
	}

Sample Usage	typedef struct __lock_t {
	    int flag;
	} lock_t;
	
	void init(lock_t *lock) {
	    // 0: lock is available, 1: lock is held
	    lock->flag = 0;
	}
	
	void lock(lock_t *lock) {
	    while (TestAndSet(&lock->flag, 1) == 1)
	        ; // spin-wait (do nothing)
	}
	
	void unlock(lock_t *lock) {
	    lock->flag = 0;
	}

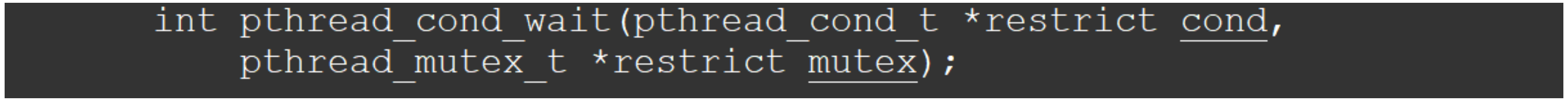
HW Support for Building Locks: Compare-And-Change

Description	int CompareAndExchange(int *ptr, int expected, int new) {
	    int original = *ptr;
	    if (original == expected)
	        *ptr = new;
	    return original;
	}

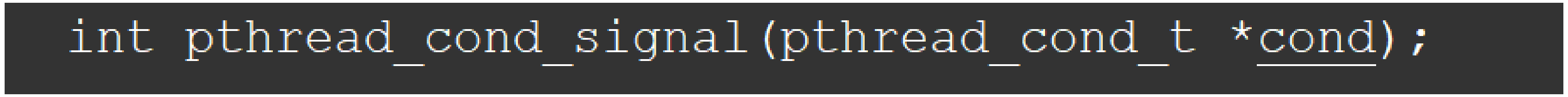
Sample Usage	typedef struct __lock_t {
	    int flag;
	} lock_t;
	
	void init(lock_t *lock) {
	    // 0: lock is available, 1: lock is held
	    lock->flag = 0;
	}
	
	void lock(lock_t *lock) {
	    while (CompareAndSwap(&lock->flag, 0, 1))
	        ; // spin-wait (do nothing)
	}
	
	void unlock(lock_t *lock) {
	    lock->flag = 0;
	}

Threads API – Add cond calls

Call/name	Description
pthread_create	Start a new thread in the calling process
pthread_join	Join threads (similar with the wait syscall used for processes)
pthread_mutex_lock	Lock a mutex
pthread_mutex_trylock	If the mutex object is locked already, the call shall return immediately
pthread_mutex_unlock	Unlock a mutex
pthread_cond_wait	Put the calling thread to sleep. Wait for another thread to wake/signal. Release the lock!!! Will lock back when awake.
pthread_cond_signal	Wake up/signal at least one of the threads that are waiting

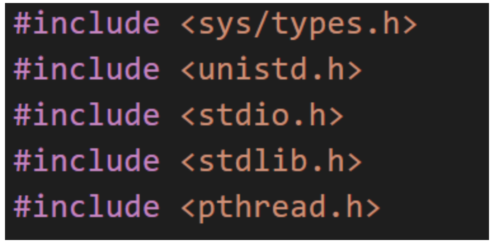


```
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

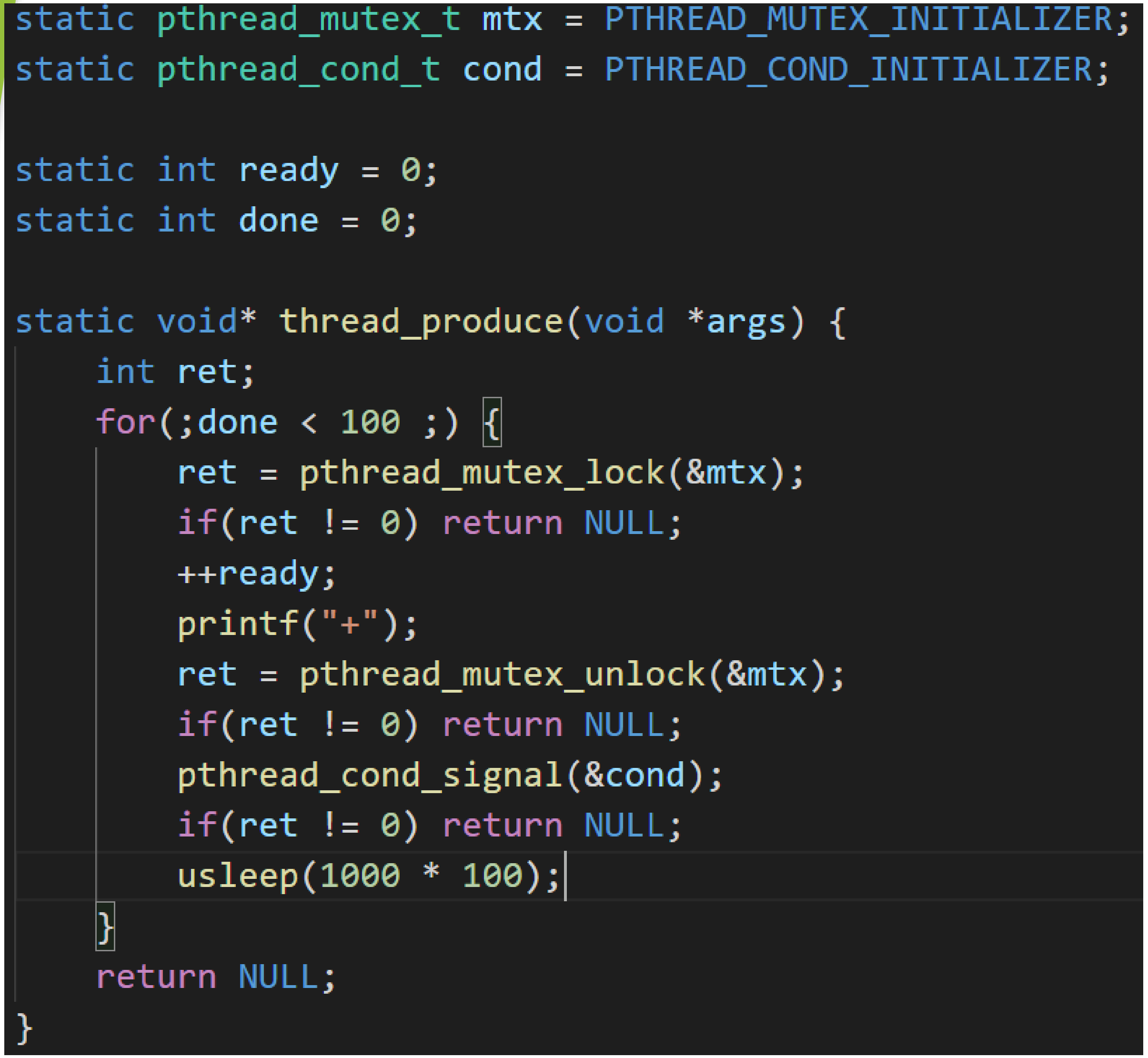


```
int pthread_cond_signal(pthread_cond_t *cond);
```

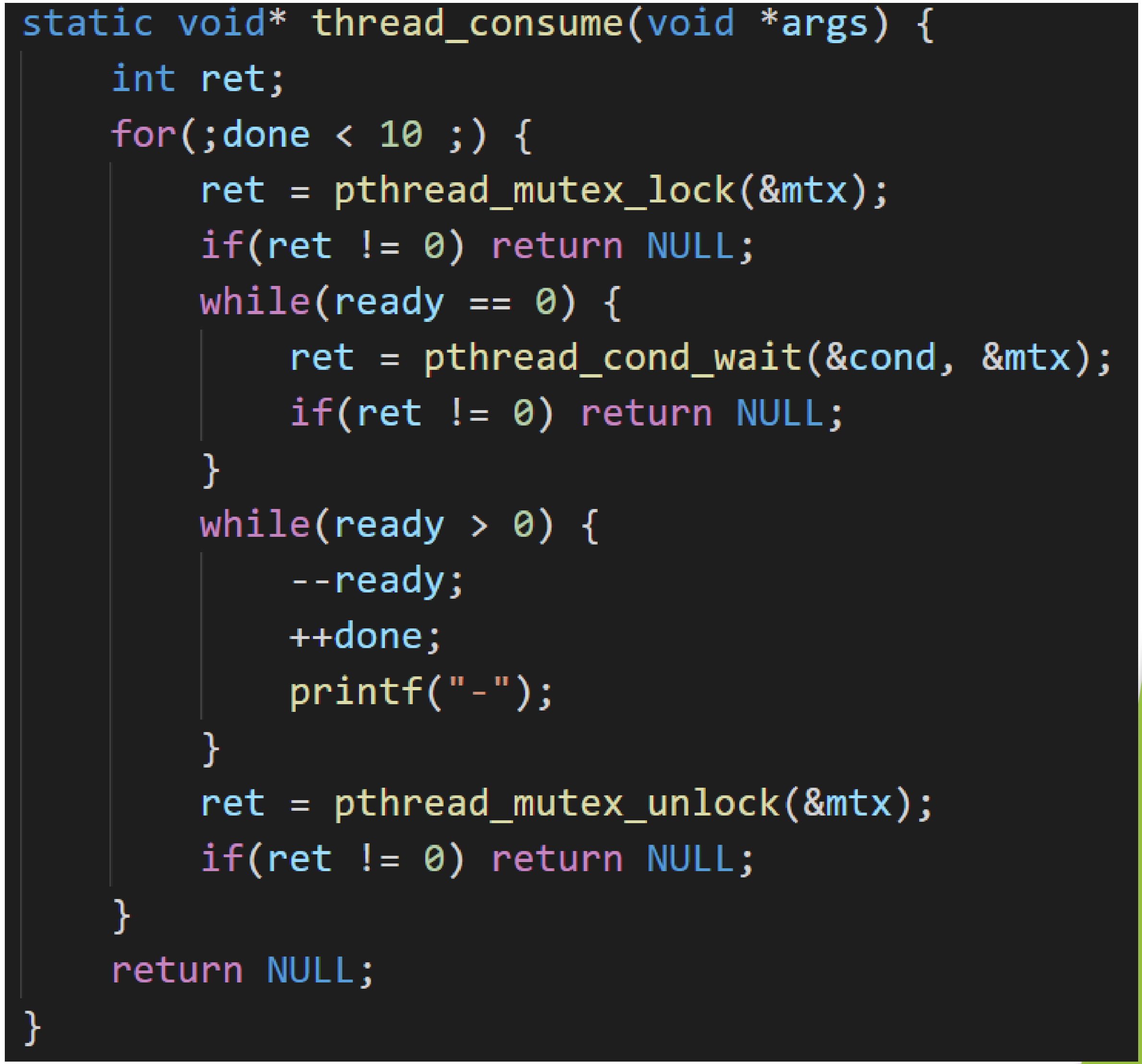
Cond API



```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

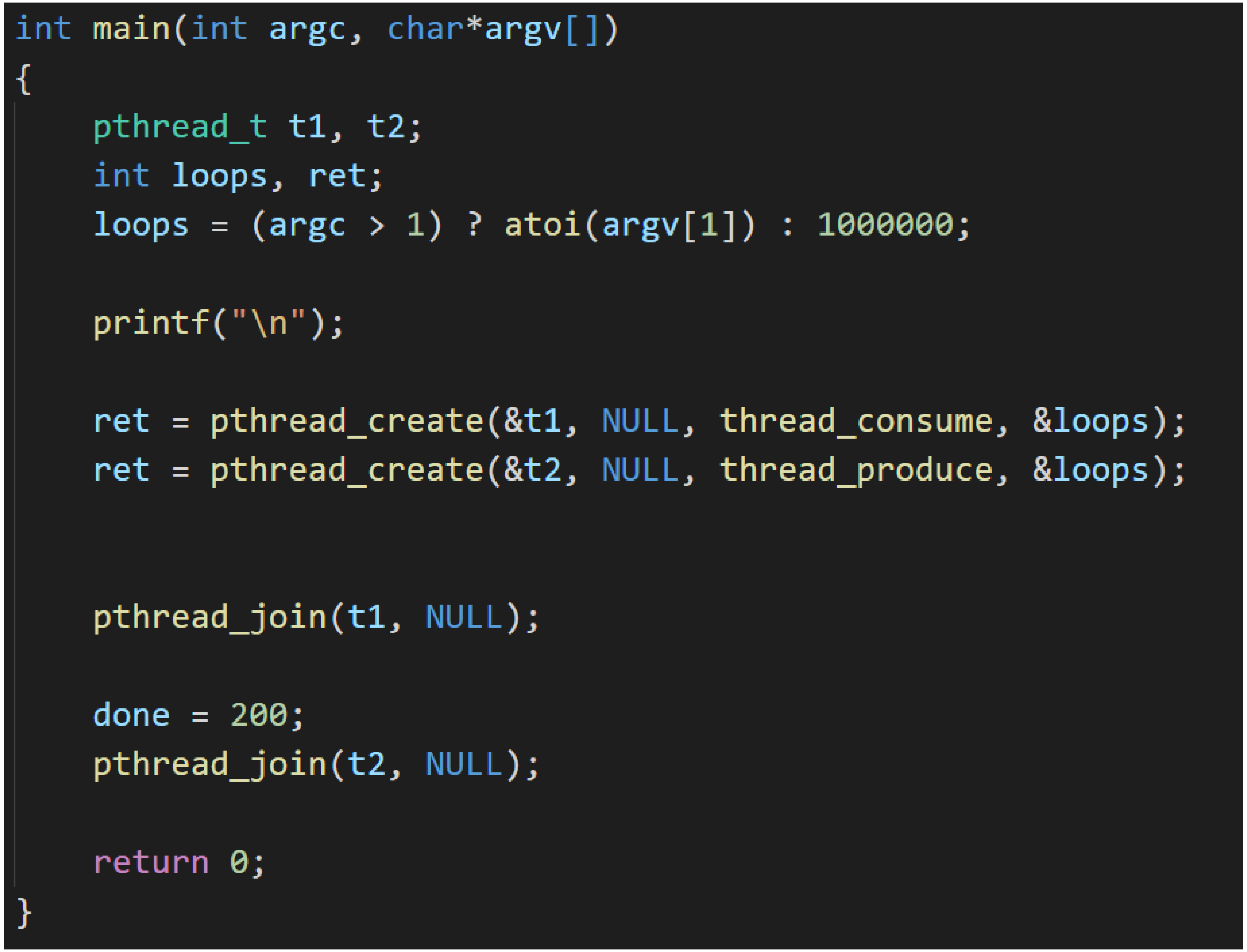


```c
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static int ready = 0;
static int done = 0;

static void* thread_produce(void *args) {
    int ret;
    for(;done < 100 ;) {
        ret = pthread_mutex_lock(&mtx);
        if(ret != 0) return NULL;
        ++ready;
        printf("+");
        ret = pthread_mutex_unlock(&mtx);
        if(ret != 0) return NULL;
        pthread_cond_signal(&cond);
        if(ret != 0) return NULL;
        usleep(1000 * 100);
    }
    return NULL;
}
```



```c
static void* thread_consume(void *args) {
    int ret;
    for(;done < 10 ;) {
        ret = pthread_mutex_lock(&mtx);
        if(ret != 0) return NULL;
        while(ready == 0) {
            ret = pthread_cond_wait(&cond, &mtx);
            if(ret != 0) return NULL;
        }
        while(ready > 0) {
            --ready;
            ++done;
            printf("-");
        }
        ret = pthread_mutex_unlock(&mtx);
        if(ret != 0) return NULL;
    }
    return NULL;
}
```

Cond API



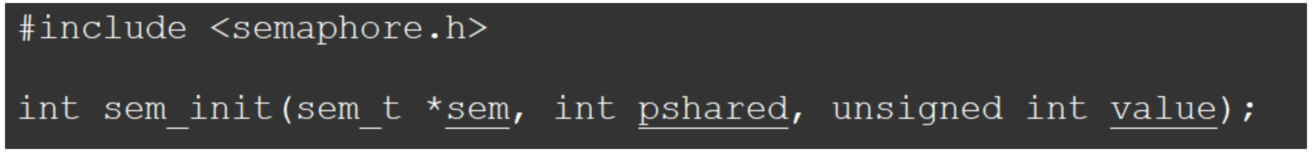
```
int main(int argc, char*argv[])
{
    pthread_t t1, t2;
    int loops, ret;
    loops = (argc > 1) ? atoi(argv[1]) : 1000000;

    printf("\n");

    ret = pthread_create(&t1, NULL, thread_consume, &loops);
    ret = pthread_create(&t2, NULL, thread_produce, &loops);


    pthread_join(t1, NULL);

    done = 200;
    pthread_join(t2, NULL);

    return 0;
}
```

gcc test_003_pthreads.c -lpthreads

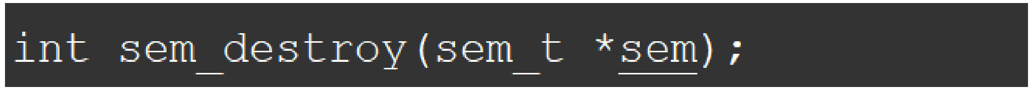
Threads API – Add Semaphores

Call/name	Description
pthread_create	Start a new thread in the calling process
pthread_join	Join threads (similar with the wait syscall used for processes)
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unclock a mutex
pthread_cond_wait	Put the calling thread to sleep. Wait for another thread to wake/signal. Release the lock!!!
pthread_cond_signal	Wake up/signal at least one of the threads that are waiting
sem_init	Initializes a semaphore
sem_destroy	Destroys the semaphore
sem_wait	If sem value >0 the function decrements the val and returns immediately If sem value <=0 the call blocks and waits
sem_post	Increments the sem value. If sem value becomes >0, awake another process/thread blocked in sem_wait

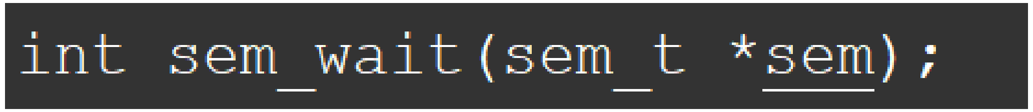
Semaphores API Prototypes



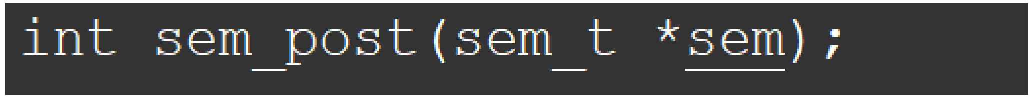
```
#include <semaphore.h>

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



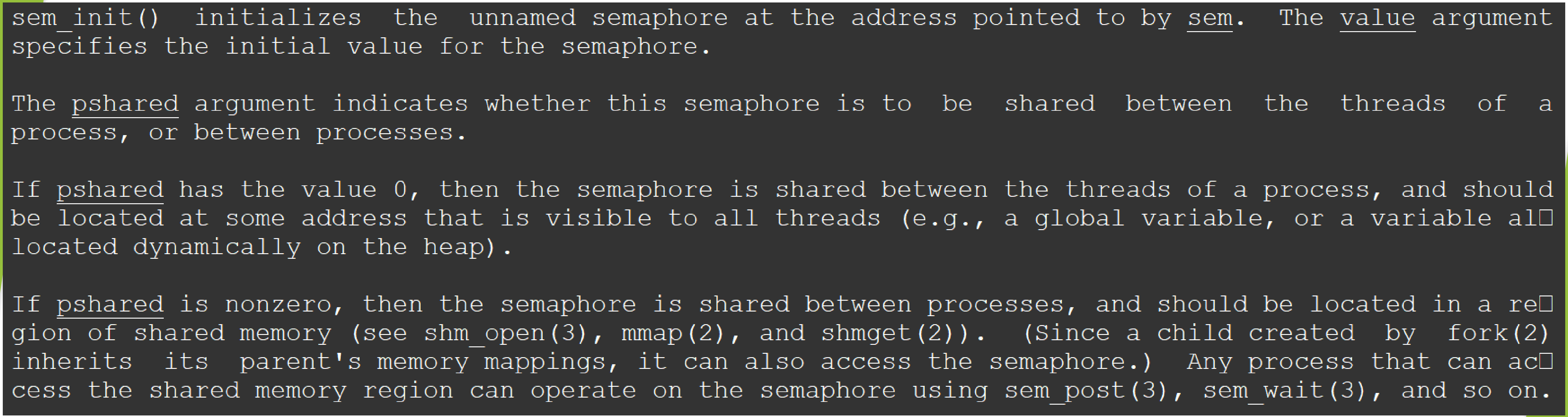
```
int sem_destroy(sem_t *sem);
```



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



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



sem_init() initializes the unnamed semaphore at the address pointed to by sem. The value argument specifies the initial value for the semaphore.

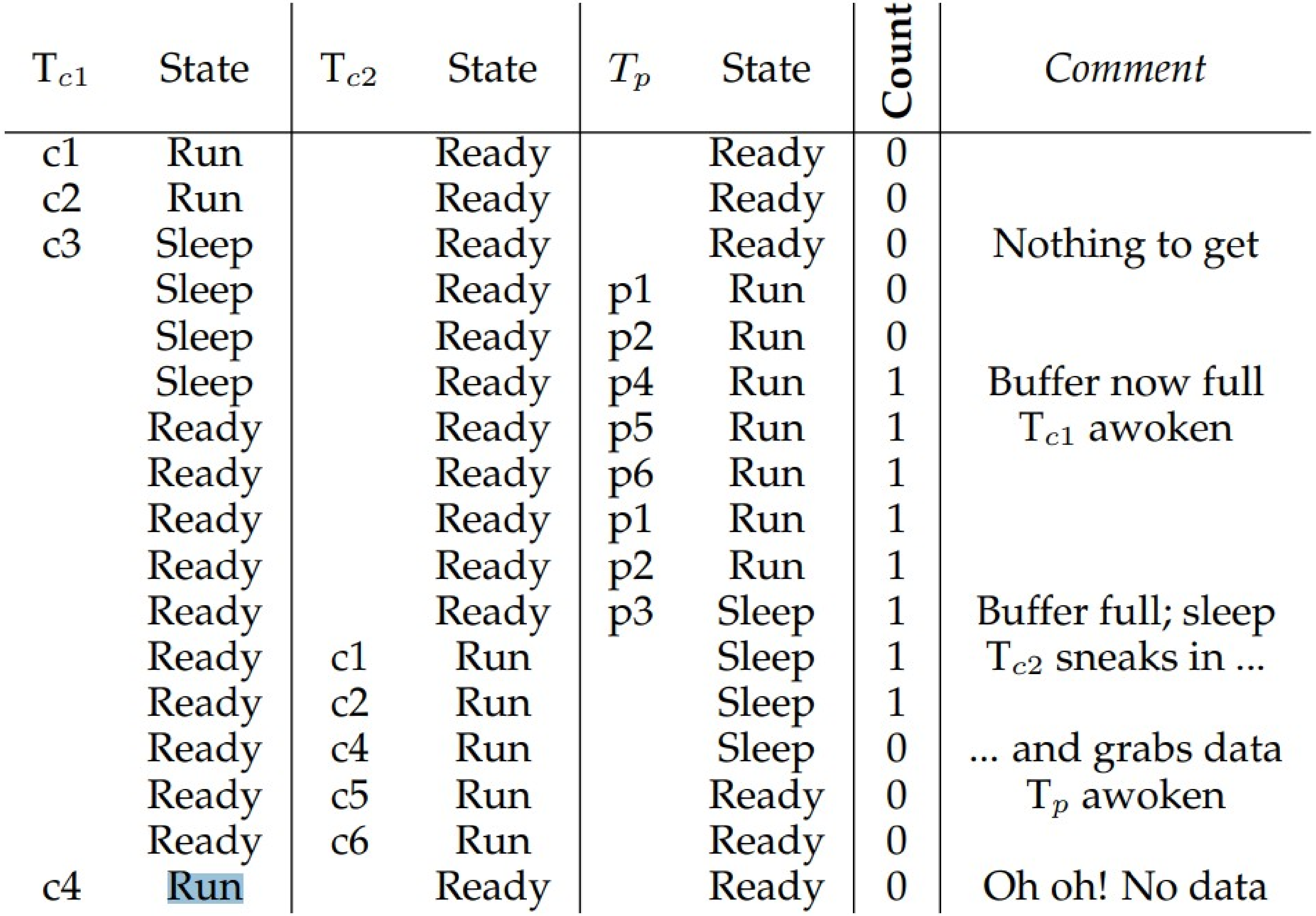
The pshared argument indicates whether this semaphore is to be shared between the threads of a process, or between processes.

If pshared has the value 0, then the semaphore is shared between the threads of a process, and should be located at some address that is visible to all threads (e.g., a global variable, or a variable allocated dynamically on the heap).

If pshared is nonzero, then the semaphore is shared between processes, and should be located in a region of shared memory (see shm_open(3), mmap(2), and shmget(2)). (Since a child created by fork(2) inherits its parent's memory mappings, it can also access the semaphore.) Any process that can access the shared memory region can operate on the semaphore using sem_post(3), sem_wait(3), and so on.

Source: linux man page

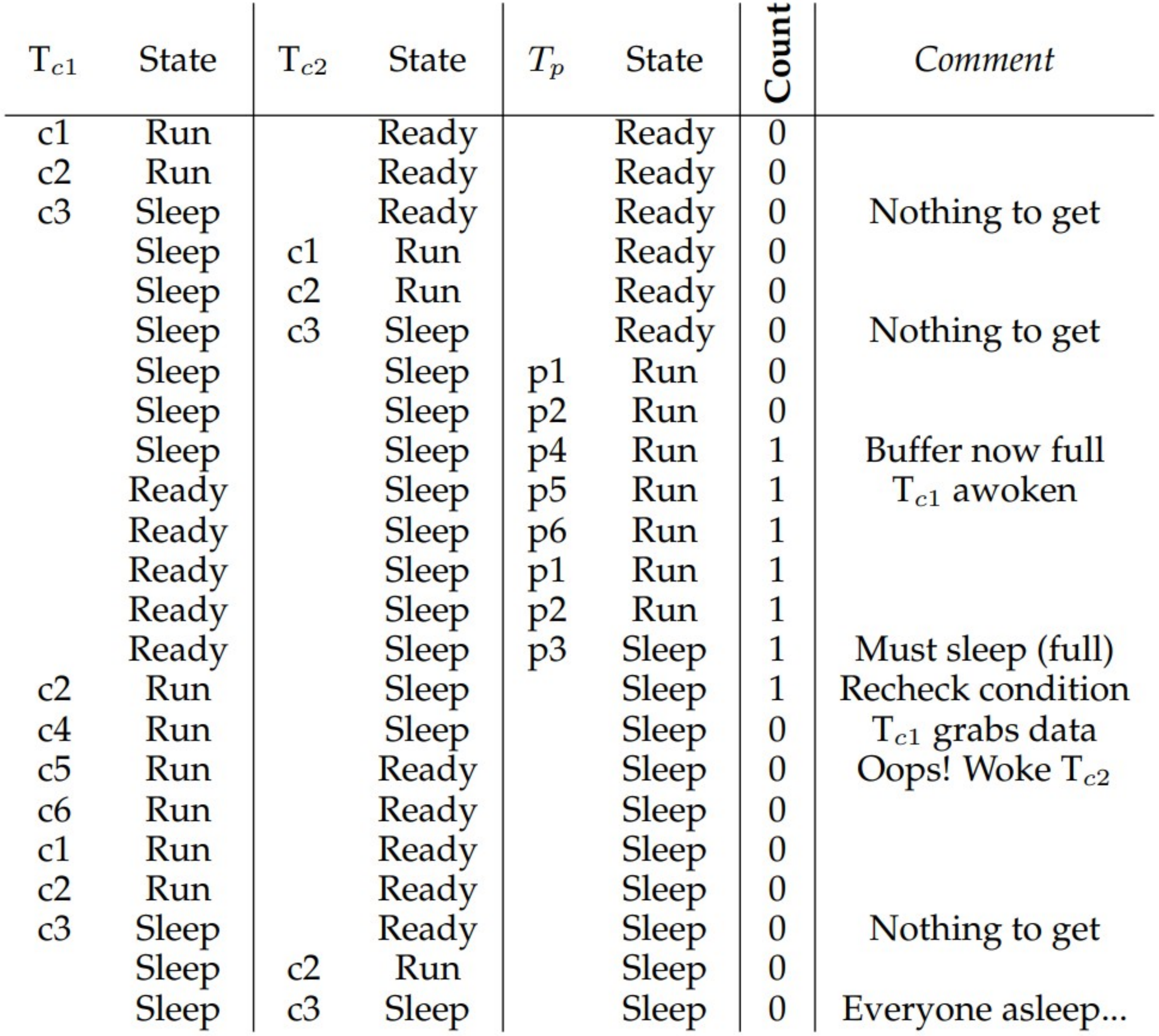
Cond API (if vs while check issue)



T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Run	0	
	Sleep		Ready	p2	Run	0	
	Sleep		Ready	p4	Run	1	Buffer now full
	Ready		Ready	p5	Run	1	T_{c1} awoken
	Ready		Ready	p6	Run	1	
	Ready		Ready	p1	Run	1	
	Ready		Ready	p2	Run	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Run		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Run		Sleep	1	
	Ready	c4	Run		Sleep	0	... and grabs data
	Ready	c5	Run		Ready	0	T_p awoken
	Ready	c6	Run		Ready	0	
c4	Run		Ready		Ready	0	Oh oh! No data

https://pages.cs.*wisc*.edu/~remzi/OSTEP/threads-cv.pdf

Cond API (same cond for consumer and producer issue)



T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Run		Ready	0	
	Sleep	c2	Run		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Run	0	
	Sleep		Sleep	p2	Run	0	
	Sleep		Sleep	p4	Run	1	Buffer now full
	Ready		Sleep	p5	Run	1	T_{c1} awoken
	Ready		Sleep	p6	Run	1	
	Ready		Sleep	p1	Run	1	
	Ready		Sleep	p2	Run	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Run		Sleep		Sleep	1	Recheck condition
c4	Run		Sleep		Sleep	0	T_{c1} grabs data
c5	Run		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Run		Ready		Sleep	0	
c1	Run		Ready		Sleep	0	
c2	Run		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Run		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

https://pages.cs.*wisc*.edu/~remzi/OSTEP/threads-cv.pdf