Operating Systems Design
5. Threads

Paul Krzyzanowski

pxk@cs.rutgers.edu
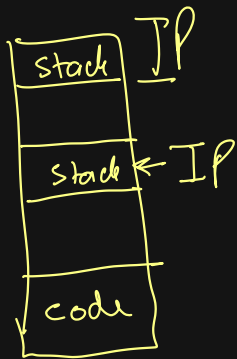
fork

Stack
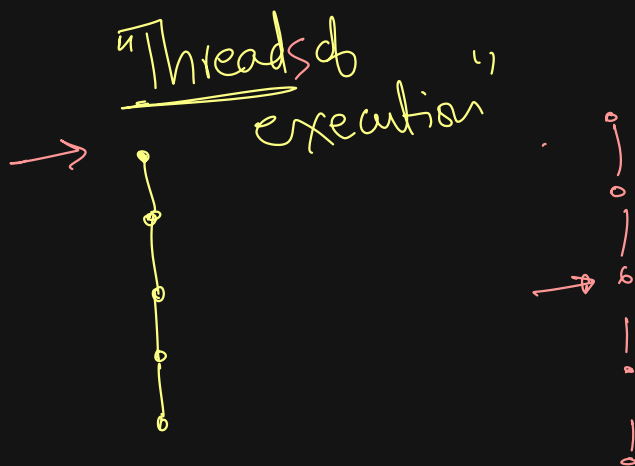
Heap

Text

COW

Copy on write

P

C

$x = 5$

$C \rightarrow \quad x = 15$

$P \rightarrow \quad P \rightsquigarrow 5$

Stack	IP
Stack	← IP
code	
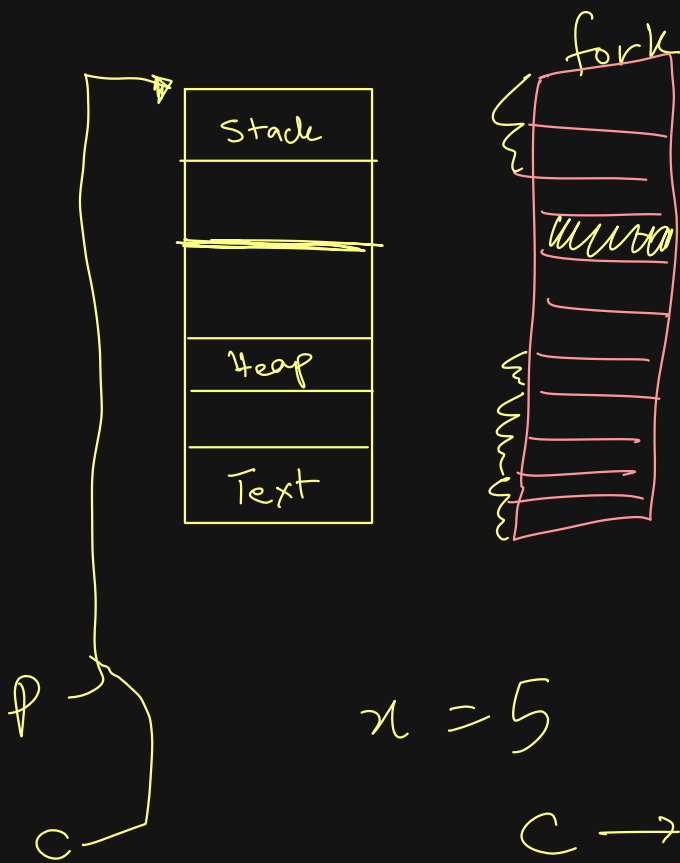
"lightweight processes"

"Threads of execution"

Thread of execution

Single sequence of instructions
- Pointed to by the program counter (PC)
- Executed by the processor

Conventional programming model & OS structure:
- Single threaded
- One process = one thread

Multi-threaded model
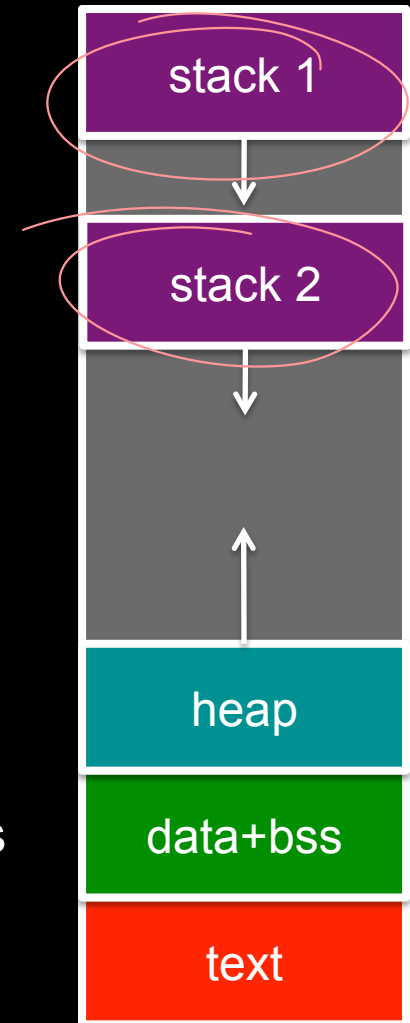
A thread is a subset of a process:
– A process contains one or more kernel threads

Share memory and open files
– BUT:
   separate program counter, registers, and stack
– Shared memory includes the heap and global/ static data
– No memory protection among the threads

Preemptive multitasking:
– Operating system preempts & schedules threads

stack 1
stack 2
heap
data+bss
text

Sharing

Threads share:

- Text segment (instructions)
- Data segment (static and global data)
- BSS segment (uninitialized data)
- Open file descriptors
- Signals
- Current working directory
- User and group IDs

Threads do not share:

- Thread ID
- Saved registers, stack pointer, instruction pointer
- Stack (local variables, temporary variables, return addresses)
- Signal mask
- Priority (scheduling information)
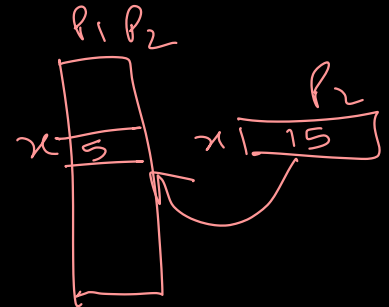
Why is this good?

Threads are more efficient
- Much less overhead to create: no need to create new copy of memory space, file descriptors, etc.

Sharing memory is easy (automatic)
- No need to figure out inter-process communication mechanisms

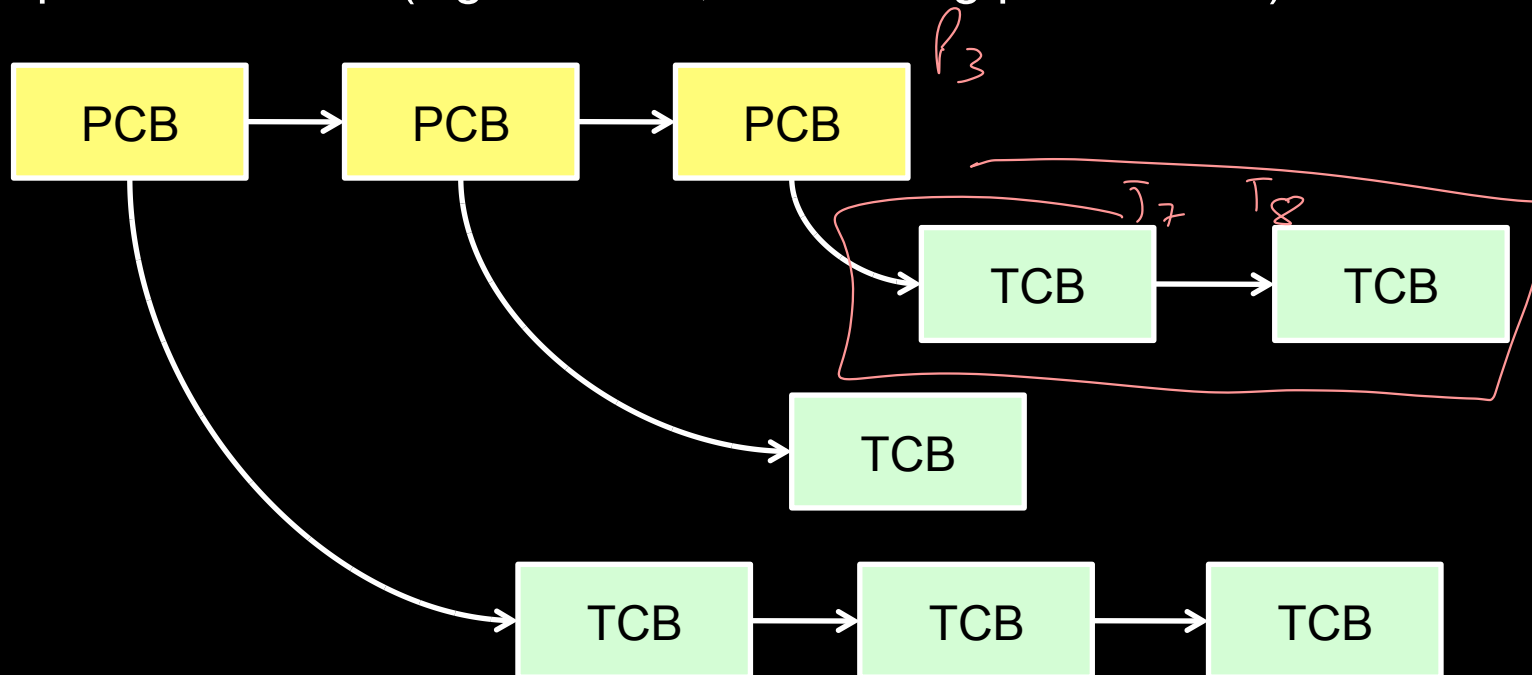Take advantage of multiple CPUs – just like processes
- Program scales with increasing # of CPUs
- Take advantage of multiple cores

Implementation

Process info (Process Control Block) contains one or more Thread Control Blocks (TCB):

- Thread ID
- Saved registers
- Other per-thread info (signal mask, scheduling parameters)

Scheduling

A threaded-aware operating system scheduler schedules *threads*, not *processes*

- A process is just a container for one or more threads

Scheduler has to realize:

- Context switch among threads of different processes is more expensive:
    - Flush cache memory (or have memory with process tags)
    - Flush virtual memory TLB (or have tagged TLB)
    - Replace page table pointer in memory management unit
- Scheduling threads onto a different CPU is more expensive
    - The CPU's cache may have memory used by the thread cached
    - CPU affinity

Process vs. Thread context switch

[A thread switch within the same process is not a context switch]

```
linux/arch/i386/kernel/process.c:

/* Re-load page tables for a new address space */
{
    unsigned long new_cr3 = next->tss.cr3;
    if (new_cr3 !=3D prev->tss.cr3)
        asm volatile("movl %0,%%cr3": :"r" (new_cr3));
}
```

Programming patterns

Design Patterns

(1) Single task thread

- Do a specific job and then release the thread

(2) Worker threads

- Specific task for each worker thread
- Dispatch task to the thread that handles it

(3) Thread pools

- Create a pool of threads *a priori*
- Use an existing thread to perform a task; wait if no threads available
- Common model for servers

Kernel-level threads vs. User-level threads

Kernel-level

- Threads supported by operating system
- OS handles scheduling, creation, synchronization

User-level

- Library with code for creation, termination, scheduling
- Kernel sees one execution context: one process
- May or may not be preemptive

pthreads .

User-level threads

Advantages

- Low-cost: user level operations that do not require switching to the kernel
- Scheduling algorithms can be replaced easily & custom to app
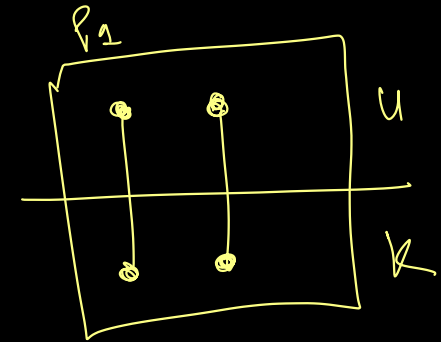- Greater portability

Disadvantages

- If a thread is blocked, all threads for the process are blocked
  - Every system call needs an asynchronous counterpart
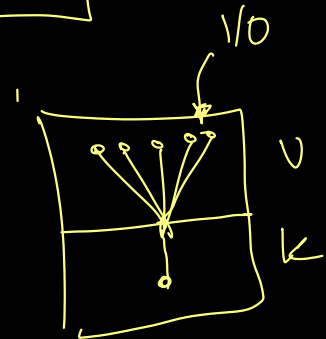- Cannot take advantage of multiprocessing

You can have both

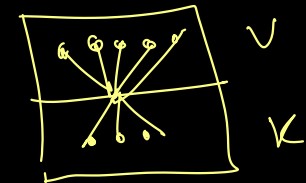User-level thread library on top of multiple kernel threads

1:1 – pure kernel threads only
      (1 user thread = 1 kernel thread

N:1 – pure user threads only
      (N user threads on 1 kernel thread/process)

N:M – hybrid threading
      (N user threads on M kernel threads)

pthreads: POSIX Threads

- POSIX.1c, Threads extensions
  (IEEE Std 1003.1c-1995)

- Defines API for managing threads

- Linux: native POSIX Thread Library (as of 2.6 kernel)

- Also on Solaris, Mac OS X, NetBSD, FreeBSD

- API library on top of Win32

Using POSIX Threads

Create a thread *(metadata)* *(function pointer)*

```
pthread_t t;
pthread_create(&t, NULL, func, arg)
```

- Create new thread *t*
- Start executing function *func(arg)*

Join two threads: *(wait)*

```
void *ret_val;
pthread_join(t, &ret_val);
```

- Wait for thread *t* to terminate (via *return* or *pthread_exit*)

No parent/child relationship!

- Any one thread may wait (join) on another thread

Linux *clone()* system call

- Clone a process, like *fork,* but:
  - Specify function that the child will run (with argument)
    - Child terminates when the function returns
  - Specify location of the stack for the child
  - Specify what's shared:
    - Share memory (otherwise memory writes use new memory)
    - Share open file descriptor table
    - Share the same parent
    - Share root directory, current directory, and permissions mask
    - Share namespace (mount points creating a directory hierarchy)
    - Share signals
    - *And more…*

- Used by pthreads
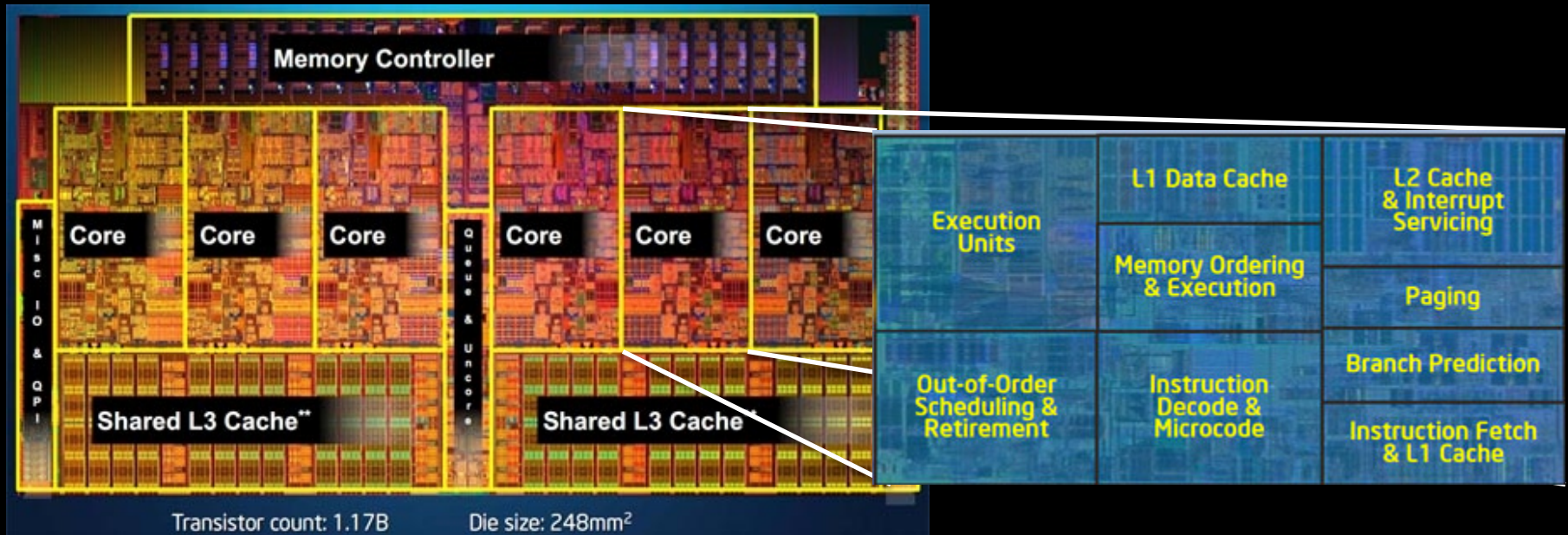
Threading in hardware

- Hyper-Threading (HT) vs. Multi-core vs. Multi-processor

- One core = One CPU

- Hyper-Threading
  - One physical core *appears* to have multiple processors
    - Looks like multiple CPUs to the OS
  - Multiple threads run but compete for execution unit
  - Events in the pipeline switch between the streams
  - Threads do not have to belong to the same process
    - But the processors share the same cache
    - Performance can degrade if two threads compete for the cache
  - Works well with instruction streams that have large memory latencies
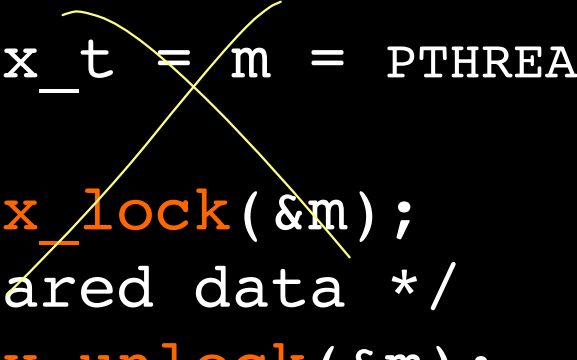
Multi-core architecture
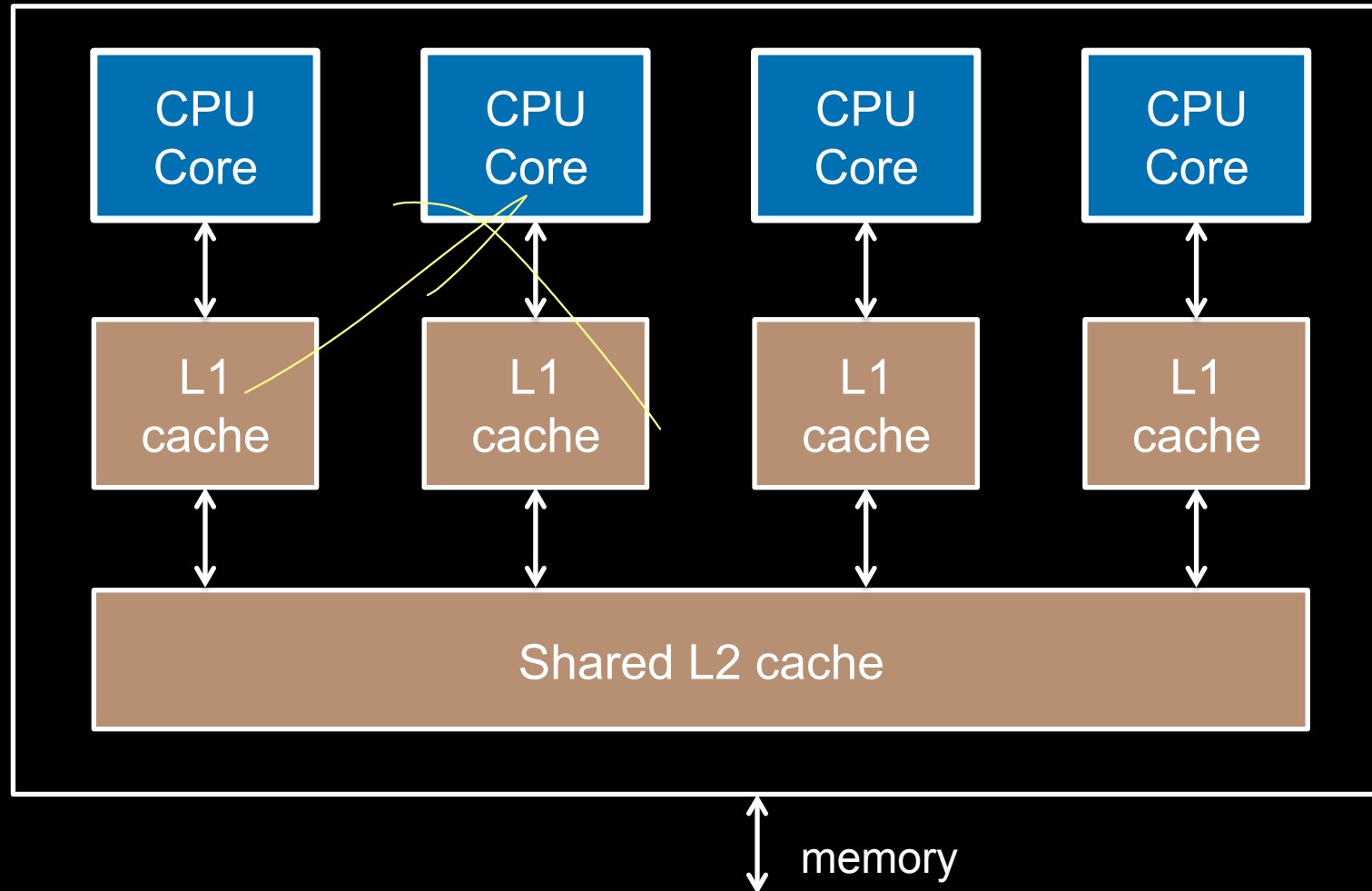
Example CPU

- Intel® Core™ i7-980X Processor Extreme Edition (Sandy Bridge) 3.3 GHz up to 3.6 GHz (Turbo)

- 6 cores; 12 threads

- Per-Core caches:
  - 64 KB L1 cache (32 KB data; 32 KB instruction)
  - 256 KB L2 cache

- 12 MB L3 cache

Memory Controller

Misc IO & QPI

Core Core Core Queue & Uncore Core Core Core

Shared L3 Cache**   Shared L3 Cache**

Transistor count: 1.17B   Die size: 248mm²

Execution Units | L1 Data Cache | L2 Cache & Interrupt Servicing

Memory Ordering & Execution | Paging

Out-of-Order Scheduling & Retirement | Instruction Decode & Microcode | Branch Prediction

Instruction Fetch & L1 Cache

Stepping on each other

- Threads share the same data

- Mutual exclusion is critical

- Allow a thread be the only one to grab a critical section
  - Others who want it go to sleep

```
pthread_mutex_t = m = PTHREAD_MUTEX_INITIALIZER;
    ...
pthread_mutex_lock(&m);
/* modify shared data */
pthread_mutex_unlock(&m);
```

The End