DOULOS

Delivering KnowHow www.doulos.com

Threads in Embedded Linux

Six Easy Pieces

Presenter: Loïc Domaigné
Senior Member Technical Staff
Doulos

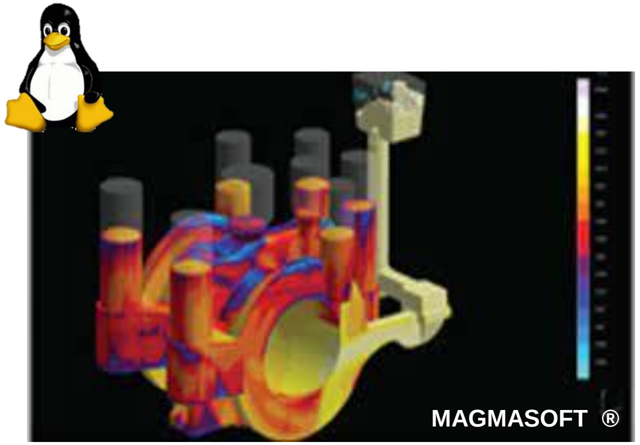
Six Easy Pieces

1. Getting Started
2. Thread Creation and Lifecycle
3. Thread Stack
4. Memory Access
5. Mutex and Condition Variable
6. Threads and Signals

DOULOS

Why Linux? Why Threads?

1996



Foundry simulation (Beowulf cluster)

Telecom

```
RWY 08                                          RWY 33
        50                                      45
        45   LOT3KM      17:34  188             EXIT
             LOT3241     17:28  179             EPMO
             SWR135Y     17:29  210             DOSIX SORIX NEPOX
 3VL         LOT632      17:24  193             GOGUS NIDRO LIMVI
        40   LOT6TK      17:25  189                         LOGDA
 RYR11ZT     TVP7509     17:23  164             EPWA
        35   LOT522      17:21  131             BIMPA SORIX NEPOX
             WZZ23NS     17:19  138             ASAVA LOGDA LIMVI
             LOT402      17:15  123             INFO
        30   MMOZX       17:15  114
             LOT75W      17:15  105
        25   QGA675R     17:11  89
             DLH5YF      17:11  78
        20   AFL2006     17:08  58
             BER119Y     17:04  48              REPLAN
             LOT4KM      17:03  R29             M    60   30   10
        15   ADR938      17:08  32              LND  0    0    0
             17 : 14 : 21                       SEP  0.0  0.0  0.0
        10   Change FLOW                        W    60   30   10
             Change RWY DIR                     LND  0    0    0
             Change RWY Option                  SEP  0.0  0.0  0.0
             Add SLOT
             Hide Miles
FLOW 6.5  17 06                                 FLOW 4.5
```

Air Traffic Control

DFS Deutsche Flugsicherung

Airline IT

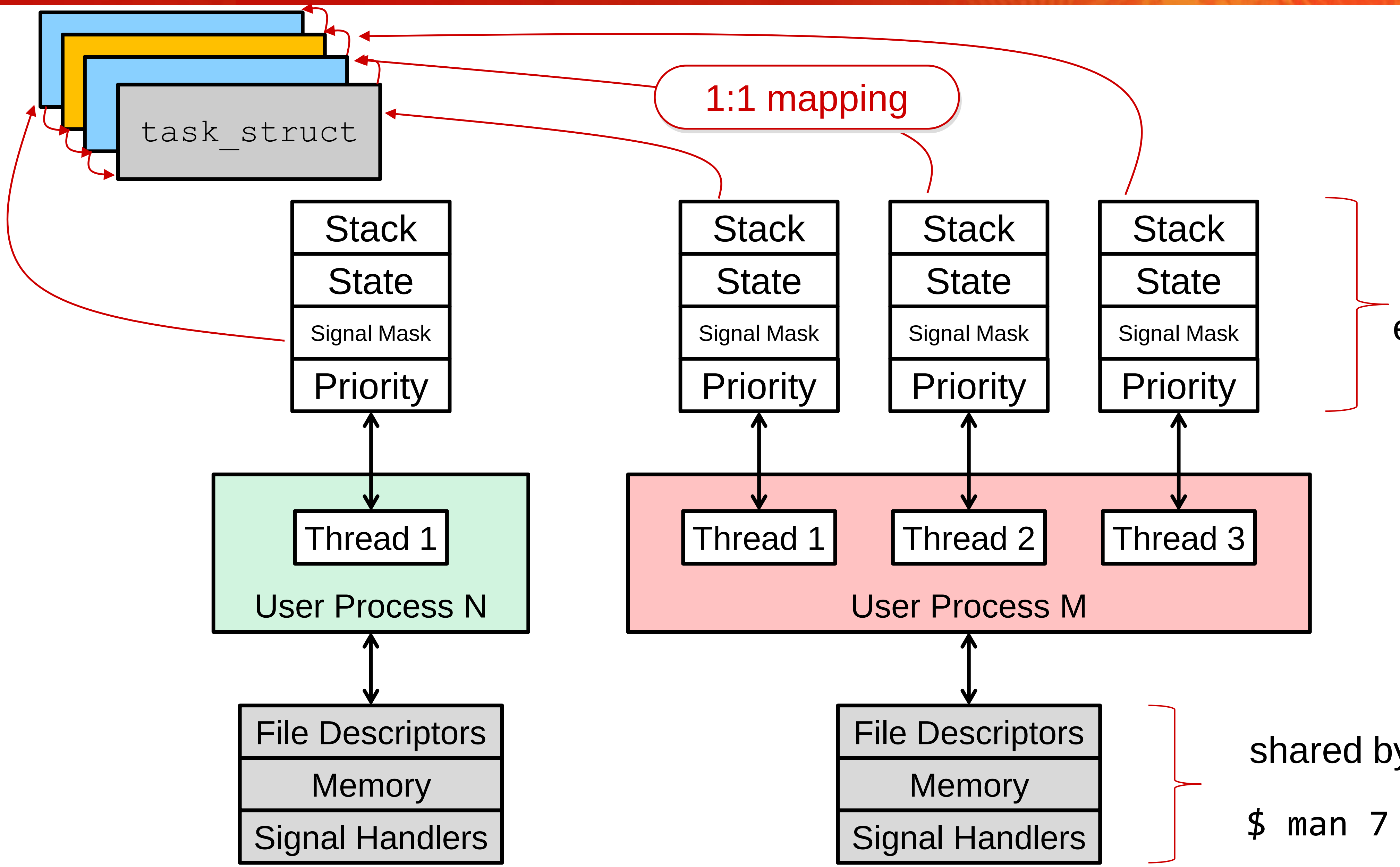
Medical

Automotive

2016

All product names, logos, and brands are property of their respective owner

Process and Threads



task_struct

1:1 mapping

Stack | State | Signal Mask | Priority

Thread 1

User Process N

File Descriptors | Memory | Signal Handlers

Stack | State | Signal Mask | Priority (×3)

Thread 1 | Thread 2 | Thread 3

User Process M

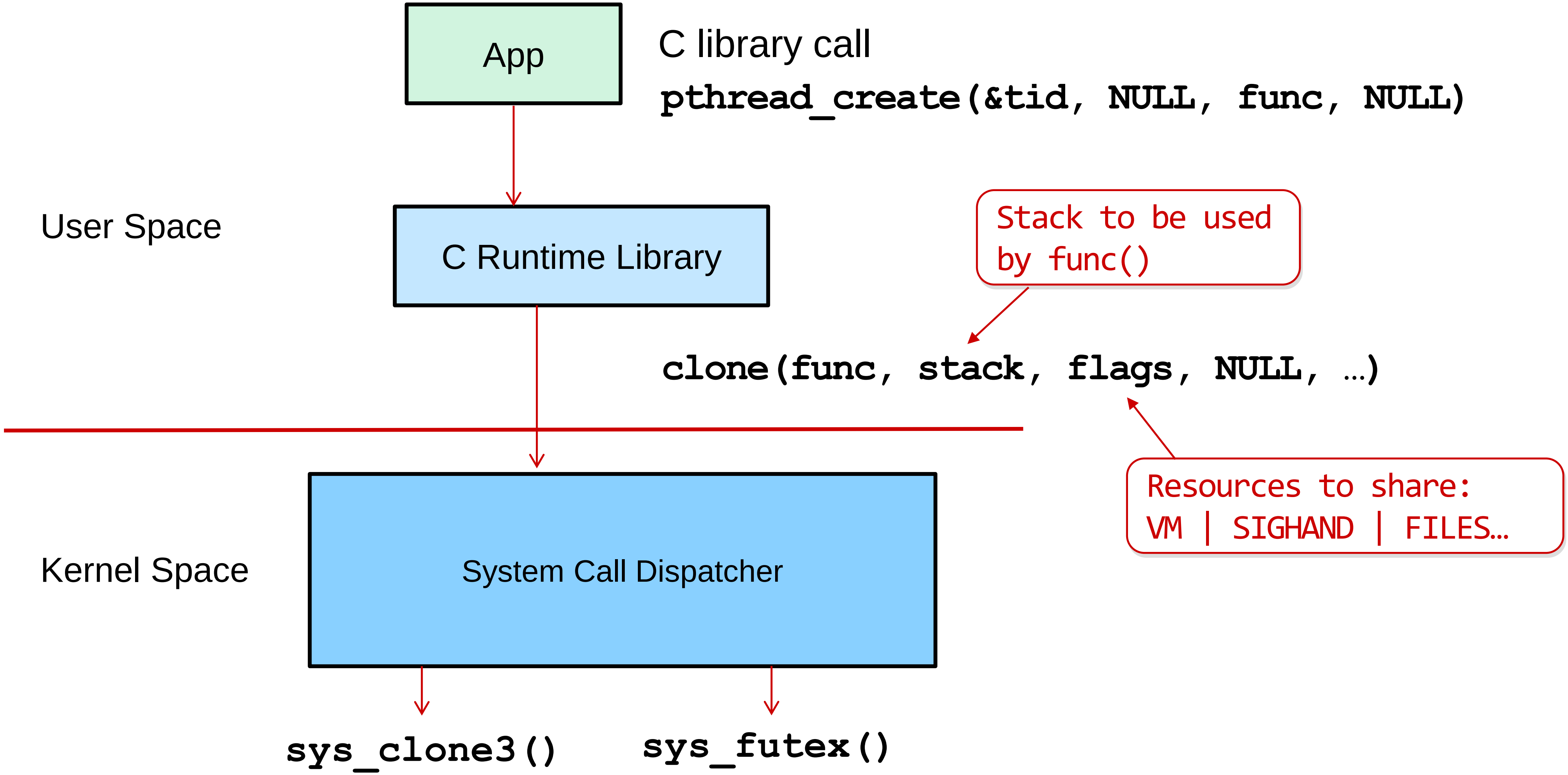
File Descriptors | Memory | Signal Handlers

specific to each thread

shared by all threads

```
$ man 7 pthreads
```

clone



App

C library call

```
pthread_create(&tid, NULL, func, NULL)
```

User Space

C Runtime Library

Stack to be used by func()

```
clone(func, stack, flags, NULL, …)
```

Resources to share:
VM | SIGHAND | FILES…

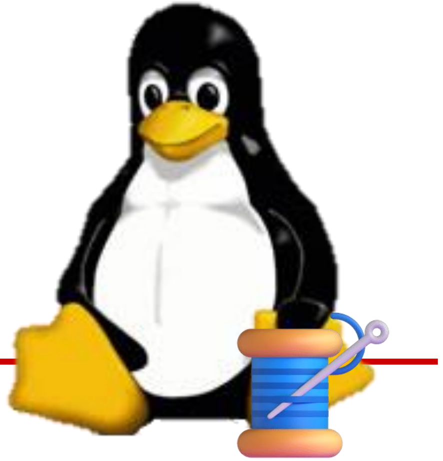
Kernel Space

System Call Dispatcher

```
sys_clone3()
sys_futex()
```

Piece #2

Thread Creation and Lifecycle

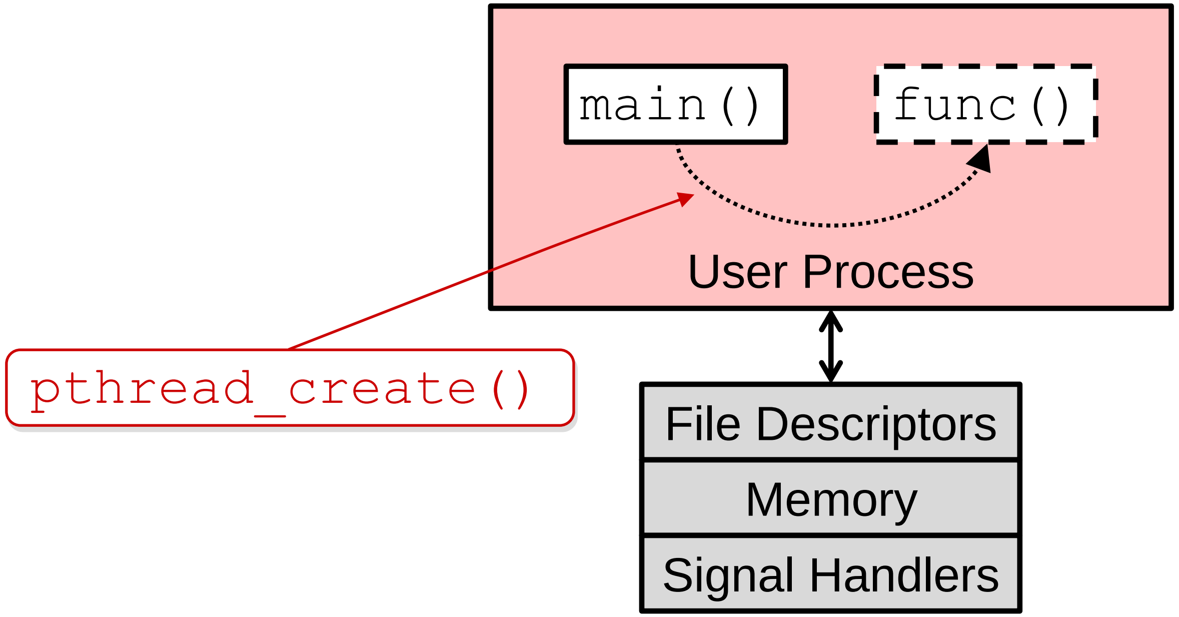


DOULOS

Starting a New Thread

```
#include <pthread.h>
int pthread_create(
    pthread_t *restrict thread,             // thread identifier
    const pthread_attr_t *restrict attr, // thread attribute (NULL=default)
    void *(*func)(void *),                  // start routine
    void *restrict arg                      // arg is passed to func()
    );
```

- **Return Value:**
  - 0 on success.
  - error number on failure.
- Compile/link flag: **`-pthread`**



Demo time!

```
// let's start our first thread

// how hard can that be???
```

https://github.com/Doulos/EOSS23

Your mileage may vary…

Aspect to watch out when coming from another OS/language:

- Error reporting?
- Is creation/start one or two separate states?
- Impact when the "main thread" terminates?
- Impact when a thread crashes (`SIGSEGV…`)?
- When are threads resources cleaned-up?
- …

While concepts are often similar, subtle differences in behaviours might catch us

Linux/POSIX behaviour (1)

Where	Action	Result	Comments
Thread	`pthread_create()`	New thread	Creation+start all in one Beware of race conditions
`main()`	`return`	Process ends	All threads die! `exit(main(argc,argv,envp))`
`func()`	`return`	Thread ends	Resources might be retained
Thread	`exit()`	Process ends	All threads die!

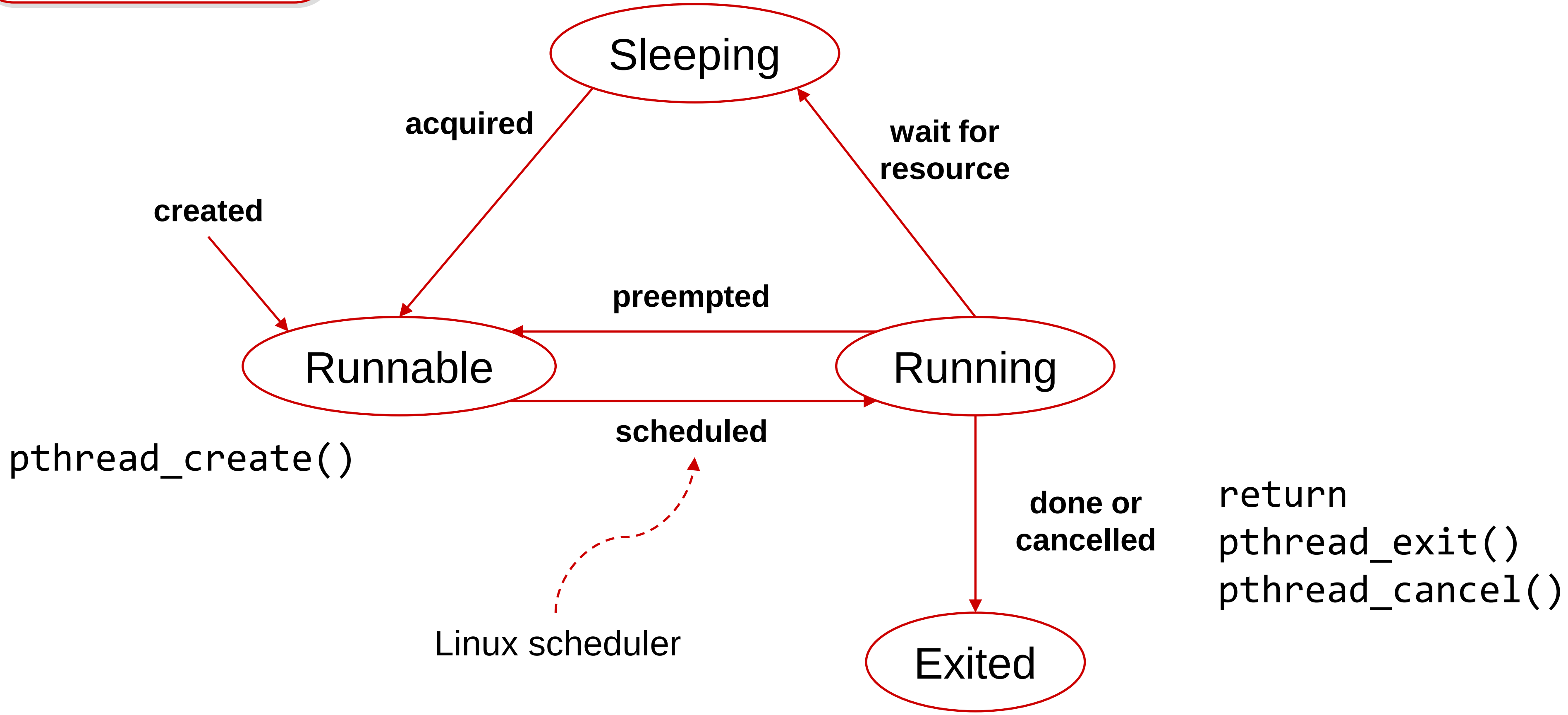
Linux/POSIX behaviour (2)

Where	Action	Result	Comments
Thread	HW Exception*	Process terminates	All threads die!
Thread	`pthread_exit()`	Thread terminates	Resources might be retained
Thread	`pthread_join()`	Wait until thread ends	Thread resources are recycled
Thread	`pthread_detach()`	Decouple a thread	Resources automatically reclaimed when thread ends

(*) `SIGBUS, SIGILL, SIGFPE, SIGSEGV`

Thread Life Cycle

Simplified view



Sleeping

acquired

wait for resource

created

preempted

Runnable

Running

scheduled

pthread_create()

done or cancelled

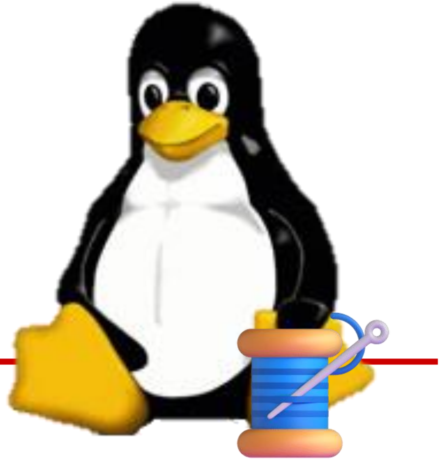
return
pthread_exit()
pthread_cancel()

Linux scheduler

Exited

Piece #3

Thread Stack



Thread Stack

- Starting a thread:

```
rc = pthread_create(&tid, NULL, start_routine, NULL);
```

- Threads require a separate stack:
  - Many "embedded OS" require to define the stack.
  - So does the `clone(2)` API.
  - Not seen yet?
- What about stack overflow?

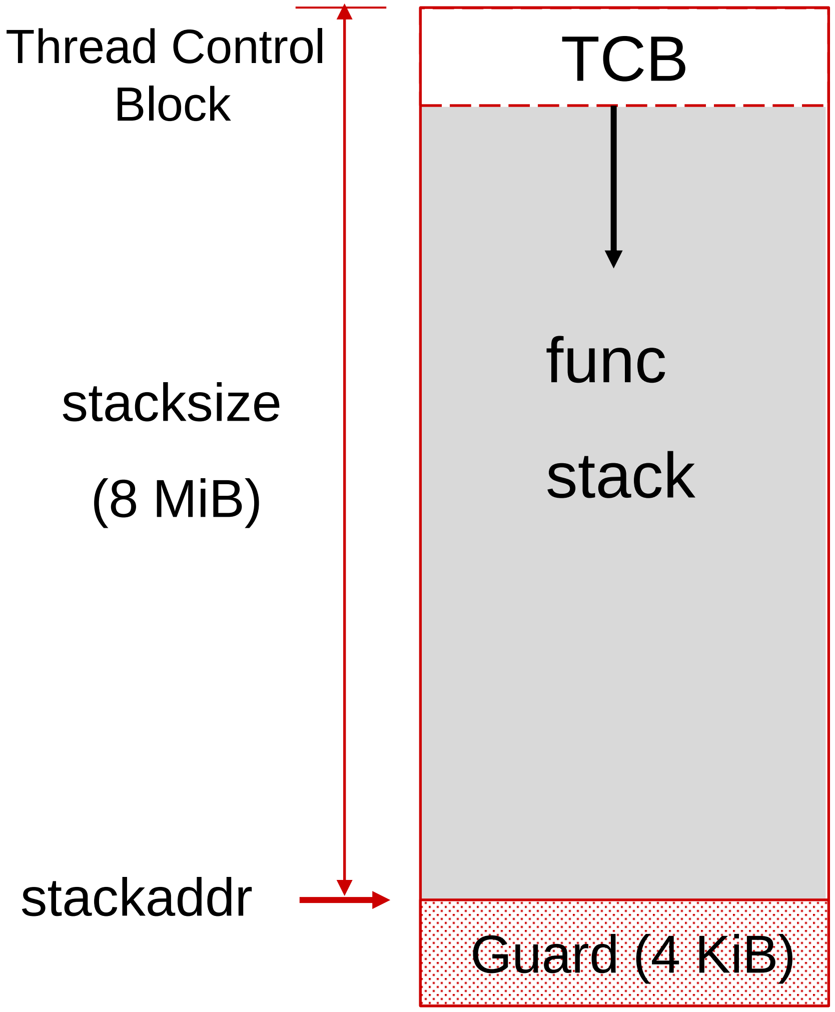
Demo time!

```
// let's overflow the stack

// for fun and teaching
```

https://github.com/Doulos/EOSS23

Thread Stack Layout (glibc 2.35)



```
0x7f7216400000
0x7f72163ff640
```

```
rc = pthread_create(&tid, NULL, func, NULL);
printf("tid = %p", tid);
…
tid = 0x7f72163ff640

[  963.372358] ex2[2949]: segfault at 7f7215bfffc8
ip  00007f721645a95b sp  00007f7215bfffa0 error 6
in libc.so.6
```

```
0x7f7215c00000    8192K rw---    [ anon ]
0x7f7215bff000       4K -----    [ anon ]
```

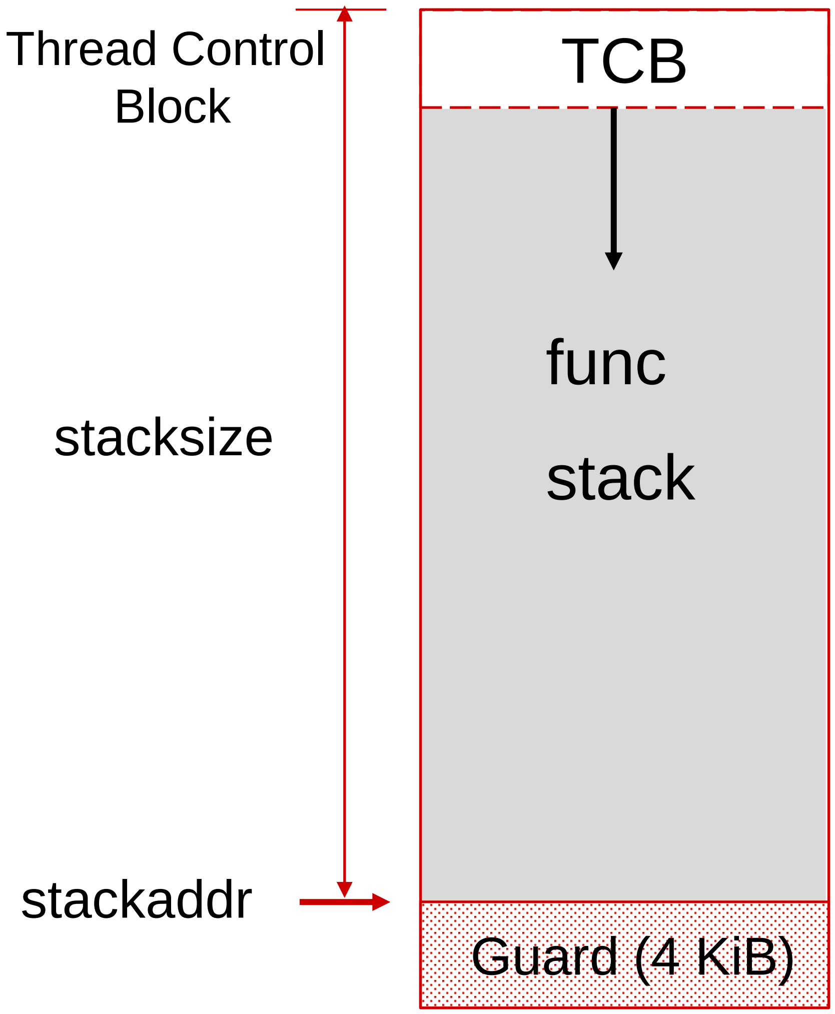
Thread Stack – Quick Facts

- Only Virtual Memory range is mapped:
  - Memory is committed using page-faults mechanism.
  - For Real-time App: use `mlockall()`.

- C-library may use different default stack size:
  - Glibc : 8 MiB (v2.35), Musl : 384 KiB (v1.2.4).
  - Solution: control stack size.

- "Thread Control Block" stored on the thread's stack:
  - Stack mapping retained, even if thread has terminated.
  - Solution: `pthread_join()` or `pthread_detach()`.

Controlling the Stack



```
pthread_attr_getdetachstate()
pthread_attr_setdetachstate()
pthread_detach()
```

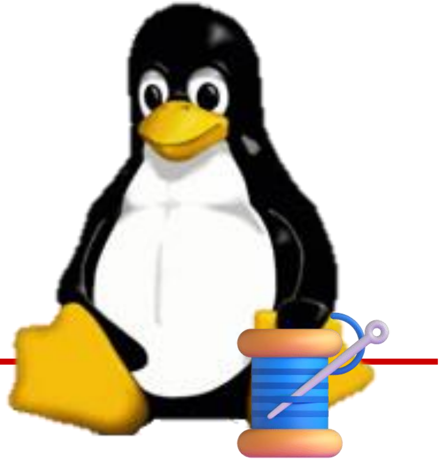
```
pthread_attr_getstacksize()
pthread_attr_setstacksize()
```

```
pthread_attr_getstack()
pthread_attr_setstack()
```

```
pthread_attr_getguardsize()
pthread_attr_setguardsize()
```

Piece #4

Memory Access



Accessing Memory

- Any thread can access any (valid) memory address.
  - local variable passed to the thread's start routine,
  - global variables,
  - any variable or location, if the address is known.

- In many situations, we want a **sequentially consistent ordering**.
- Needs synchronization! (execution order + memory visibility).

Time

Thread A	Thread B
`x = 42;`	
	`printf("x=%d\n", x)`

We expect: x=42

Memory Visibility

```
int x,y,z,t; // global variables
```

Time

Thread A	Thread B
`x = 6;`	
`y = 7;`	
`pthread_create(B);`	`//create ThrB`
`t = x + y;`	sees x,y
`pthread_join(B);`	`z = x * y;`
	`t += 8;`
	`return NULL;`
sees z,t	
`assert(z==42);`	
`assert(t==21);`	

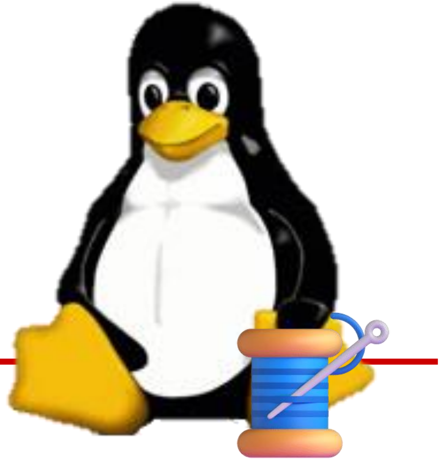
value set by a thread prior to pthread_create() can be seen by the new thread.

No such guarantee here!! (depends on timing, and system architecture!)

Likewise, value set by a thread can be seen by the thread joining it.

Piece #5

Mutex and Condition Variable



Mutex

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INIALIZER;
```

Time

Thread A	Thread B
`x = 6; y = 7;`	
`pthread_mutex_lock(&mtx);`	
`t = x + y;`	Blocks until A unlocks
	`pthread_mutex_lock(&mtx)`
`pthread_mutex_unlock(&mtx);`	
	`t += 8;`
After unlock: t will be seen in any thread that locks the same mutex	`pthread_mutex_unlock(&mtx)`

Mutex Quick Facts

- Mutex = binary semaphore conceptually, but is:
  - faster: syscall only when contended.
  - owned by a thread.
  - For RT-App: prio inheritance/ceiling protocol possible.

- Thread that locks should also unlock.
  - Behaviour in error situation (relock, not owner…)?
  - Depends on the mutex type: NORMAL, ERRORCHECK, RECURSIVE.
  - Example: NORMAL mutex causes a deadlock if relocked by the same thread.

- Mutex doesn't synchronize which thread locks first.

Condition Variables (Incorrect use of)

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INIALIZER;
pthread_cond_t cv   = PTHREAD_COND_INITIALIZER;
```

Time

Thread A	Thread B
`x = 6; y = 7;`	`pthread_mutex_lock(&mtx);`
`pthread_mutex_lock(&mtx);`	
	unlock mtx, blocks until A signals cv
	`pthread_cond_wait(&cv, &mtx);`
`t = x + y;`	mtx is locked
`pthread_cond_signal(&cv);`	
`pthread_mutex_unlock(&mtx);`	`t += 8;`
	`pthread_mutex_unlock(&mtx)`

Where is the "condition" ?

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INIALIZER;
pthread_cond_t cv   = PTHREAD_COND_INITIALIZER;
```

Time

Thread A	Thread B
`x = 6; y = 7;`	`pthread_mutex_lock(&mtx);`
`pthread_mutex_lock(&mtx);`	`while (t==0)`
	`{`
	`    pthread_cond_wait(&cv, &mtx);`
`t = x + y;`	`}`
`pthread_cond_signal(&cv);`	
`pthread_mutex_unlock(&mtx);`	`t += 8;`
	`pthread_mutex_unlock(&mtx)`

wait until t is set

signal change on t

Condition Variable – Quick Facts

- Usage pattern:

notifier

```
pthread_mutex_lock(&mtx);
// change condition
...
// signal or broadcast
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mtx);
```

waiter

```
pthread_mutex_lock(&mtx);
while (! wanted_condition ) {
    pthread_cond_wait(&cv, &mtx);
}
// do work, change condition
...
pthread_mutex_unlock(&mtx);
```

- `signal` = wakes up one waiter, `broadcast` = all waiters.
- no waiter = notification is lost. (not a problem, wait is skipped).
- use a `while` loop! a strict `if` statement might fail us.

Thread Synchronization

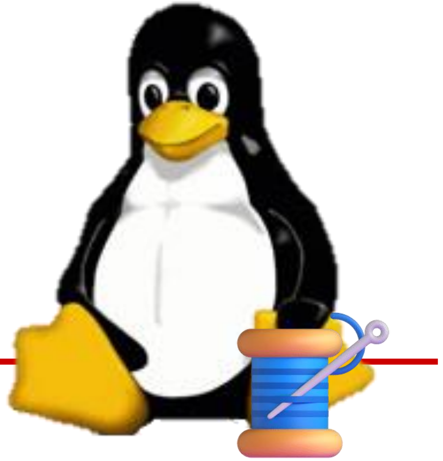
Thread of execution + Memory

Objects	Main operation	Typical Usage
Mutex	`lock*, unlock`	Mutual exclusion / Access shared data ("one thread at a time").
Condition Variables	`wait*, signal, broadcast`	Wait for some condition to become true
Barrier	`wait`	Blocks until N threads reaches the barrier
Read-Write Locks	`lock*, unlock`	Like mutex, but reader blocks only if writer holds the locks.
Spinlocks	`lock, unlock`	Like mutex, but spins.

* For mutex, condition variable, read-write lock: timed lock/wait exists.

Piece #6

Threads and Signals

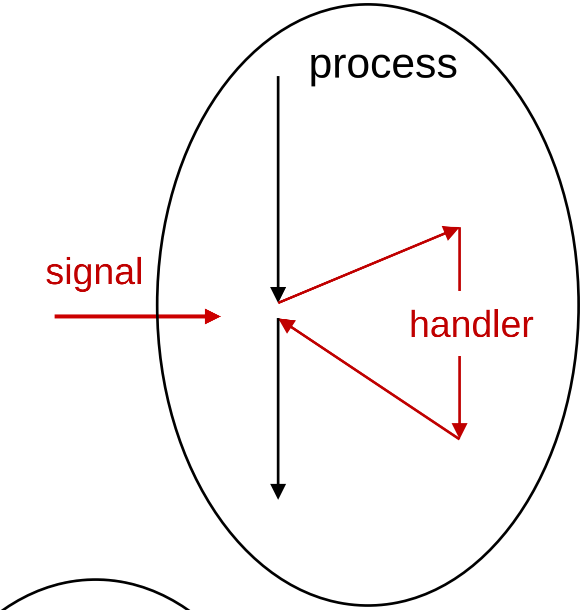


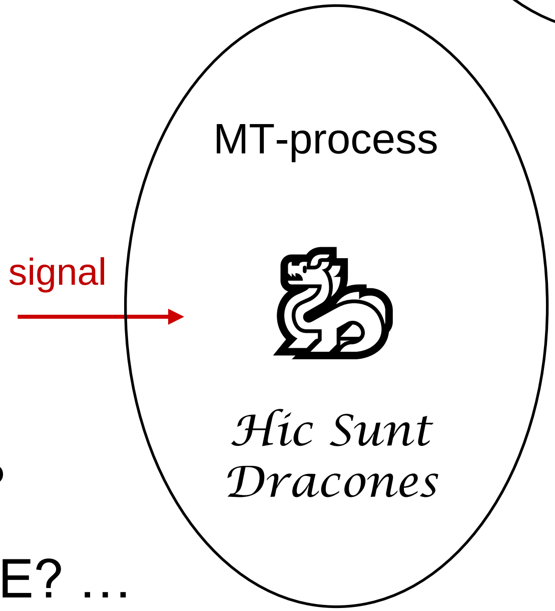
Thread and Signal

- Single threaded process:
  - Signal is delivered to process.
  - Interrupt Execution flows when unblocked.
  - Runs the signal handler asynchronously.

- Thread makes asynchronous code synchronous.
- Somewhat antinomic to signal.

- Multi-threaded process:
  - Signal is still delivered to the process.
  - Which thread is interrupted?
  - Can we block delivery in some thread?
  - Signal semantic for SIGSTOP? SIGFPE? …





Demo time!

```
// Let's meet some dragons

// compiling with -pthread
// breaks my single threaded code!

// PS: won't happen on Linux ;)
```

https://github.com/Doulos/EOSS23

Handle Signal with Threads

block signals in all threads

```
sigset_t mask; // signals to handle

int main()
{
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    pthread_sigmask(SIG_BLOCK, &mask, NULL);

    // start thread incl. sighandler_thread
    ...

}
```

sigmask is inherited

wait synchronously

```
void *sighandler_thread(void *ign)
{
    int caught;
    while (1) {
        sigwait(&mask, &caught);
        // handle signal caught
        ...
    }
}
```

Signals – Quick Facts

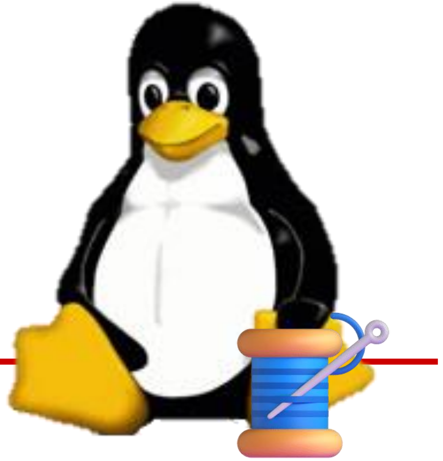
- Previous pattern allows to:
  - use any (mt-safe) functions in signal handler thread.
  - remove limitation regular process signal handler.

- Signal and process:
  - `SIGKILL` terminates process (= all threads).
  - `SIGSTOP` = stop all threads, `SIGCONT` = restart.
  - HW exception always delivered to the "faulty thread".
  - Other signals delivered to arbitrary thread, unless blocked.

- Use:
  - `pthread_kill()` to send a signal to a specific thread.
  - `pthread_sigmask()` to modify the per-thread signal mask.

Piece #7

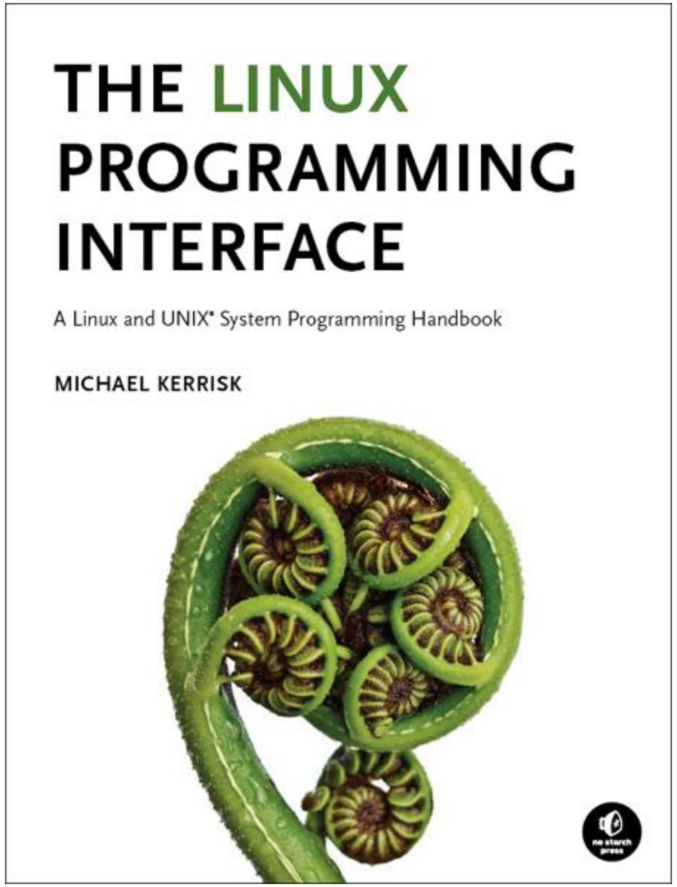
Going Further



DOULOS

References and Further Readings

```
$ man 7 pthreads
```

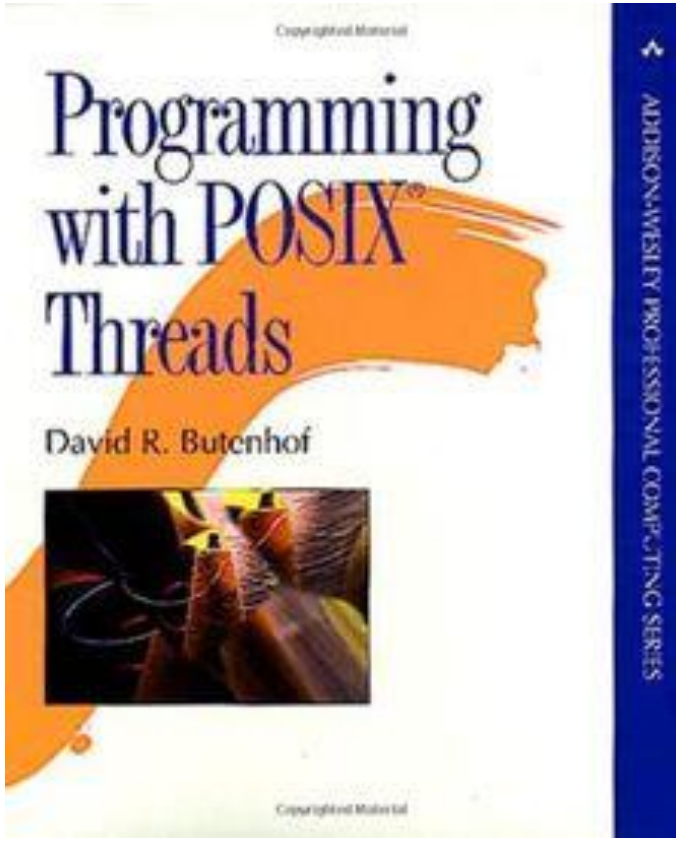


THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX® System Programming Handbook

MICHAEL KERRISK

https://man7.org/tlpi/



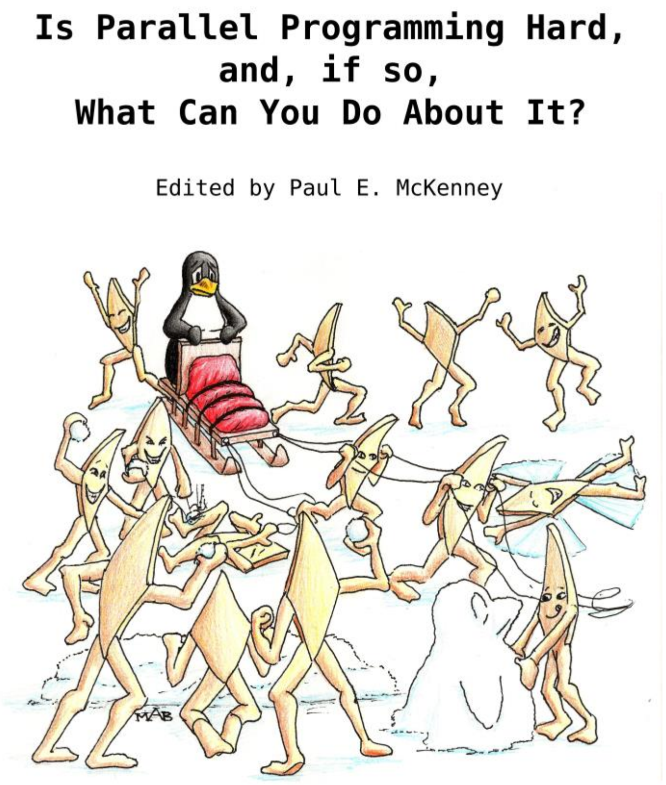
Copyrighted Material

Programming with POSIX® Threads

David R. Butenhof

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Copyrighted Material



Is Parallel Programming Hard,
and, if so,
What Can You Do About It?

Edited by Paul E. McKenney

https://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

DOULOS

Delivering KnowHow www.doulos.com

SoC Design & Verification	» SystemVerilog » UVM » Formal » SystemC » TLM-2.0
FPGA & Hardware Design	» VHDL » Verilog » SystemVerilog » Tcl » AMD-Xilinx » Intel FPGA
Embedded Software	» Emb C/C++ » Emb Linux » Yocto » RTOS » Security » Arm
AI & Deep Learning	AI & DEEP LEARNING

SystemVerilog™ OUT OF THE BOX

UVM

SYSTEMC™

yocto PROJECT PARTICIPANT

free RTOS

SECURE EMBEDDED

arm APPROVED TRAINING PARTNER