On Threads, Processes and Co-Processes

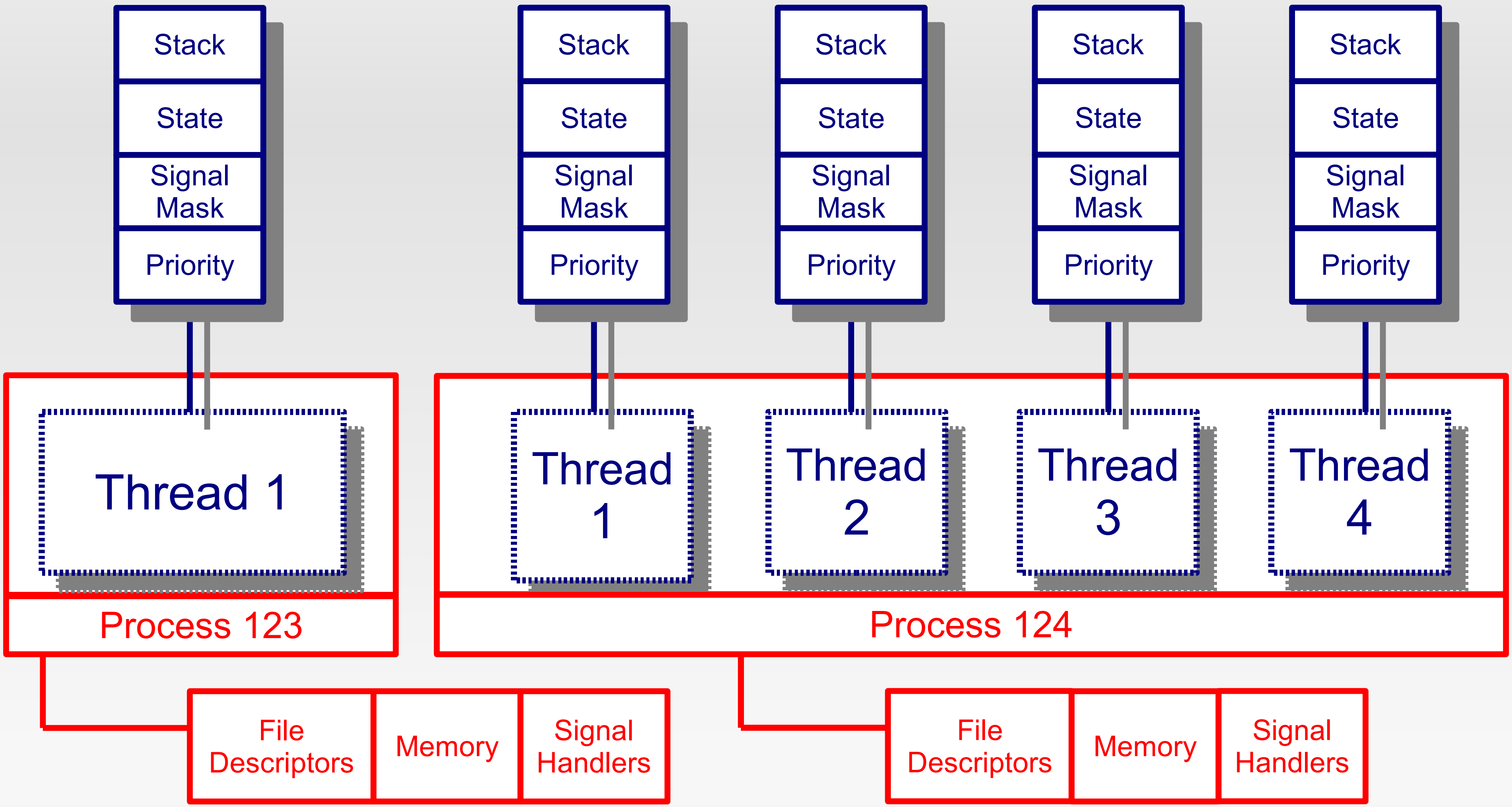
Gilad Ben-Yossef
Codefidence Ltd.

About Me

- **Chief Coffee Drinker** of Codefidence Ltd.
- Co-Author of **Building Embedded Linux System**, 2nd edition
- Israeli FOSS NPO **Hamakor** co-founder
- **August Penguin** co-founder
- git blame -- **FOSS**.*
  - Linux
  - Asterisk
  - cfgsh (RIP) ...

Codefidence
a name you can trust™

Linux Process and Threads



Codefidence
a name you can trust™

The Question

Should we use threads?

Given an application that calls for several tasks, should we implement each task as a thread in the same processes or in a separate processes?

What Do The Old Wise Men Say?

- **"If you think you need threads then your processes are too fat."**
  - *Rob Pike*, co-author of The Practice of Programming and The Unix Programming Environment.
- **“A Computer is a state machine. Threads are for people who can't program state machines."**
  - *Alan Cox*, Linux kernel programmer

Codefidence
a name you can trust™

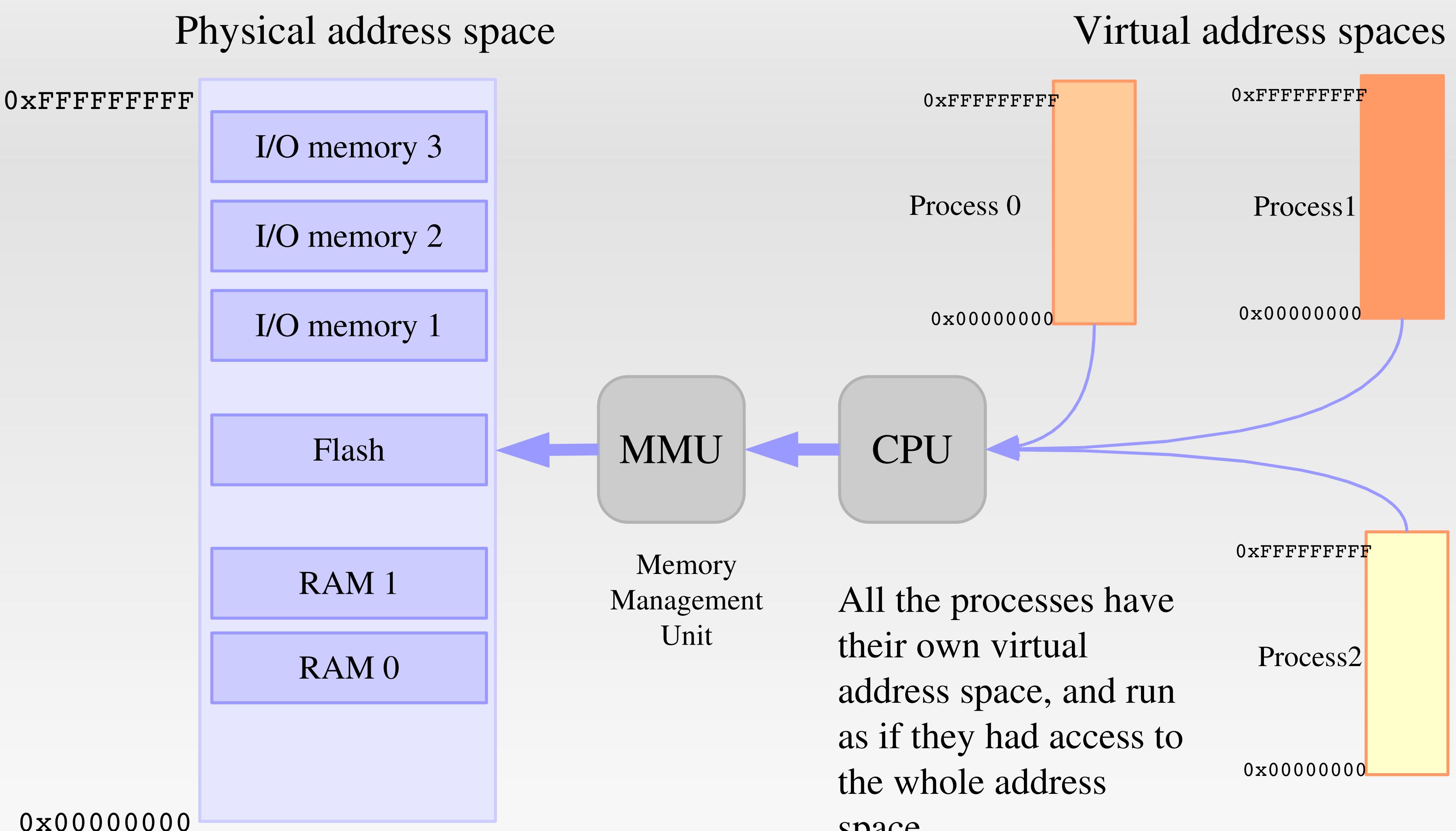
Spot the Difference (1)

- The Linux scheduler always operates at a thread granularity level.
- You can start a new process without loading a new program, just like with a thread
  - fork() vs. exec()
- Process can communicate with each other just like threads
  - Shared memory, mutexes, semaphores, message queues etc. work between processes too.

Spot the Difference (2)

- Process creation time is roughly double that of a thread.
  - ... but Linux process creation time is still very small.
  - ... but most embedded systems pre-create all tasks in advance anyway.
- Each process has it's own virtual memory address space.
- Threads (of the same process) share the virtual address space.

Physical and Virtual Memory



All the processes have their own virtual address space, and run as if they had access to the whole address space.

Page Tables

Address Space Number *

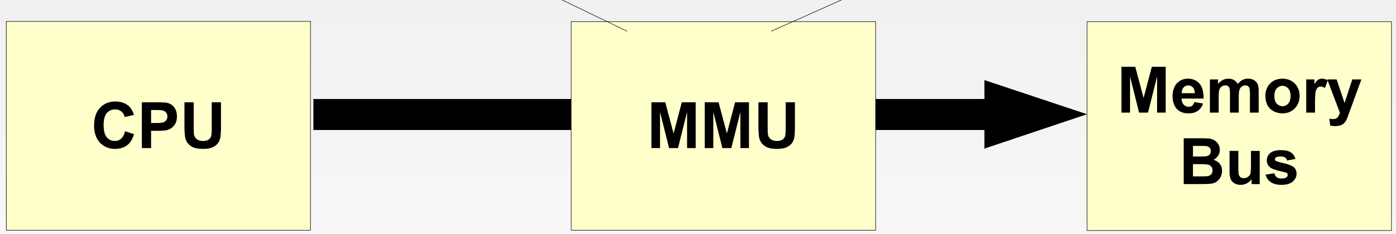
Virtual Page Address

Page Frame Address

Read Write Execute Cached

ASN	Virtual	Physical	Permission
12	0x8000	0x5340	RWXC
15	0x8000	0x3390	RX

Read Only Execute Non Cached



* Does not exists on all architectures.

Translation Look-aside Buffers

- The MMU caches the content of page tables in a CPU local cache called the TLB.
- TLBs can be managed in hardware automatically (x86, Sparc) or by software (Mips, PowerPC)
- Making changes to the page tables might require a TLB cache flush, if the architecture does not support TLB ASN (Alpha, Intel Nehalem, AMD SVM)

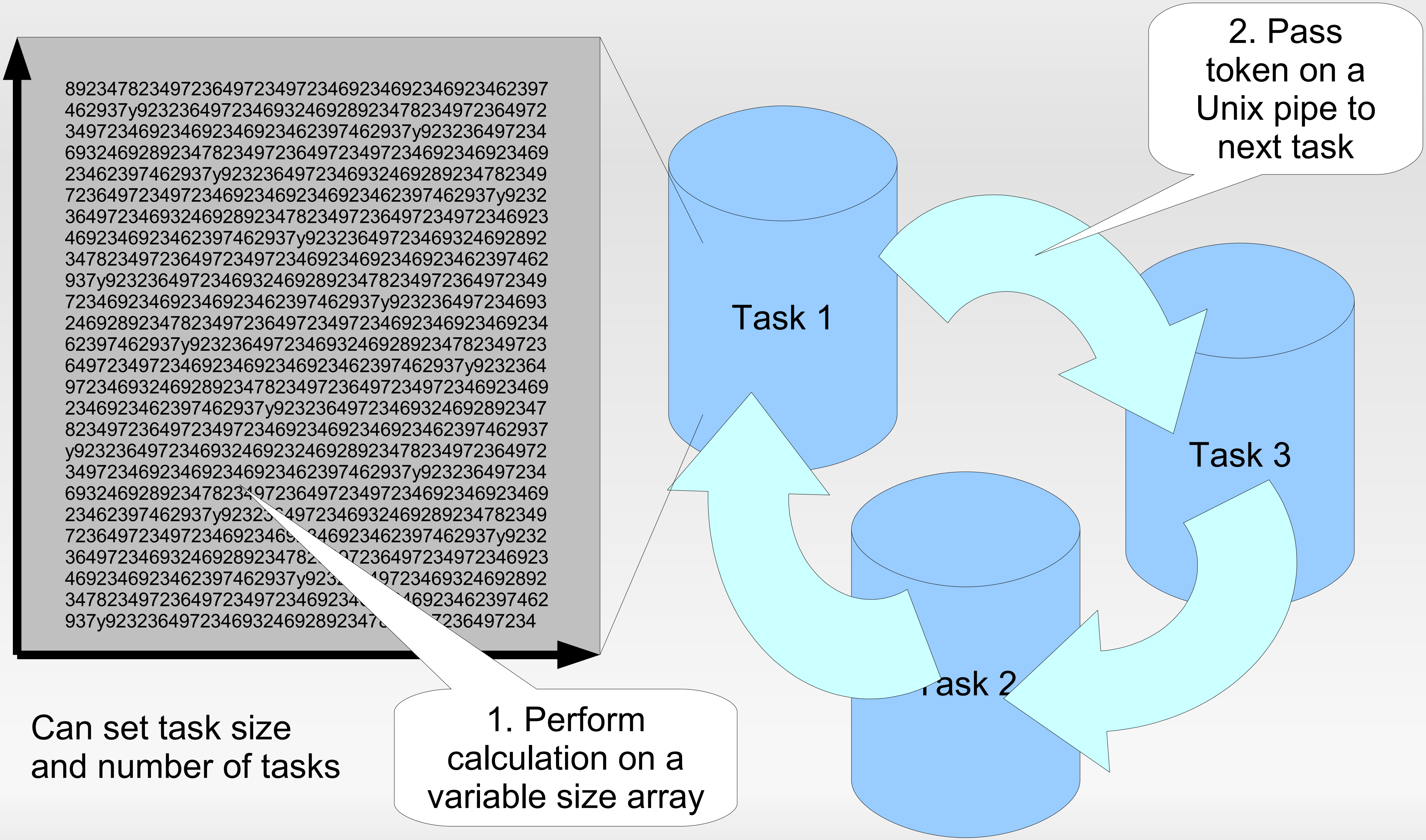
Cache Indexes

- Data and Instruction caches may use either virtual or physical address as the key to the cache.
- Physically indexed caches don't care about different address spaces.
- Virtually indexed caches cannot keep more then one alias to the same physical address
  - Or need to carefully manage aliasing using tagging.

LMBench

- LMBench is a suite of simple, portable benchmarks by Larry McVoy and Carl Staelin
- lat_ctx measures context switching time.
- Original lat_ctx supported only measurement of inter process context switches.
- I have extended it to measure also inter thread content switches.
- All bugs are mine, not Larry and Carl :-)

How lat_ctx works (1)



Can set task size and number of tasks

How lat_ctx works (2)

- Both the data and the instruction cache get polluted by some amount before the token is passed on.
  - The data cache gets polluted by approximately the process ``size''.
  - The instruction cache gets polluted by a constant amount, approximately 2.7 thousand instructions.
  - The benchmark measures only the context switch time, not including the overhead of doing the work.
    - A warm up run with hot caches is used as a reference.

lat_ctx Accuracy

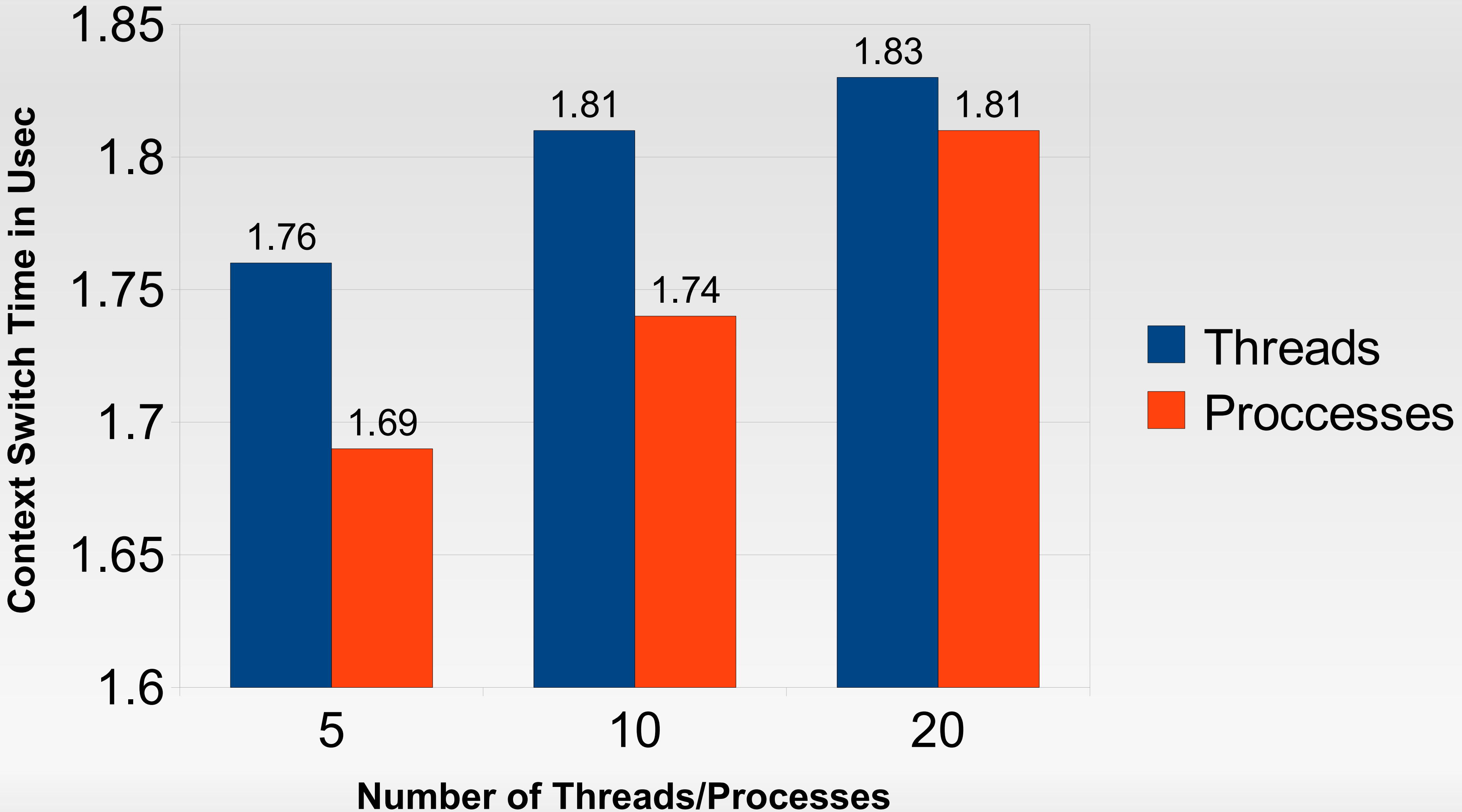
- The numbers produced by this benchmark are somewhat inaccurate;
  - **They vary by about 10 to 15% from run to run.**
- The possible reasons for the inaccuracies are detailed in LMBench documentation
  - They aren't really sure either.

Context Switch Costs

- Using the modified lat_ctx we can measure the difference between the context switch times of threads and processes.
- Two systems used:
  - **Intel x86 Core2 Duo 2Ghz**
    - Dual Core
    - Hardware TLB
  - **Freescale PowerPC MPC8568 MDS**
    - Single core
    - Software TLB

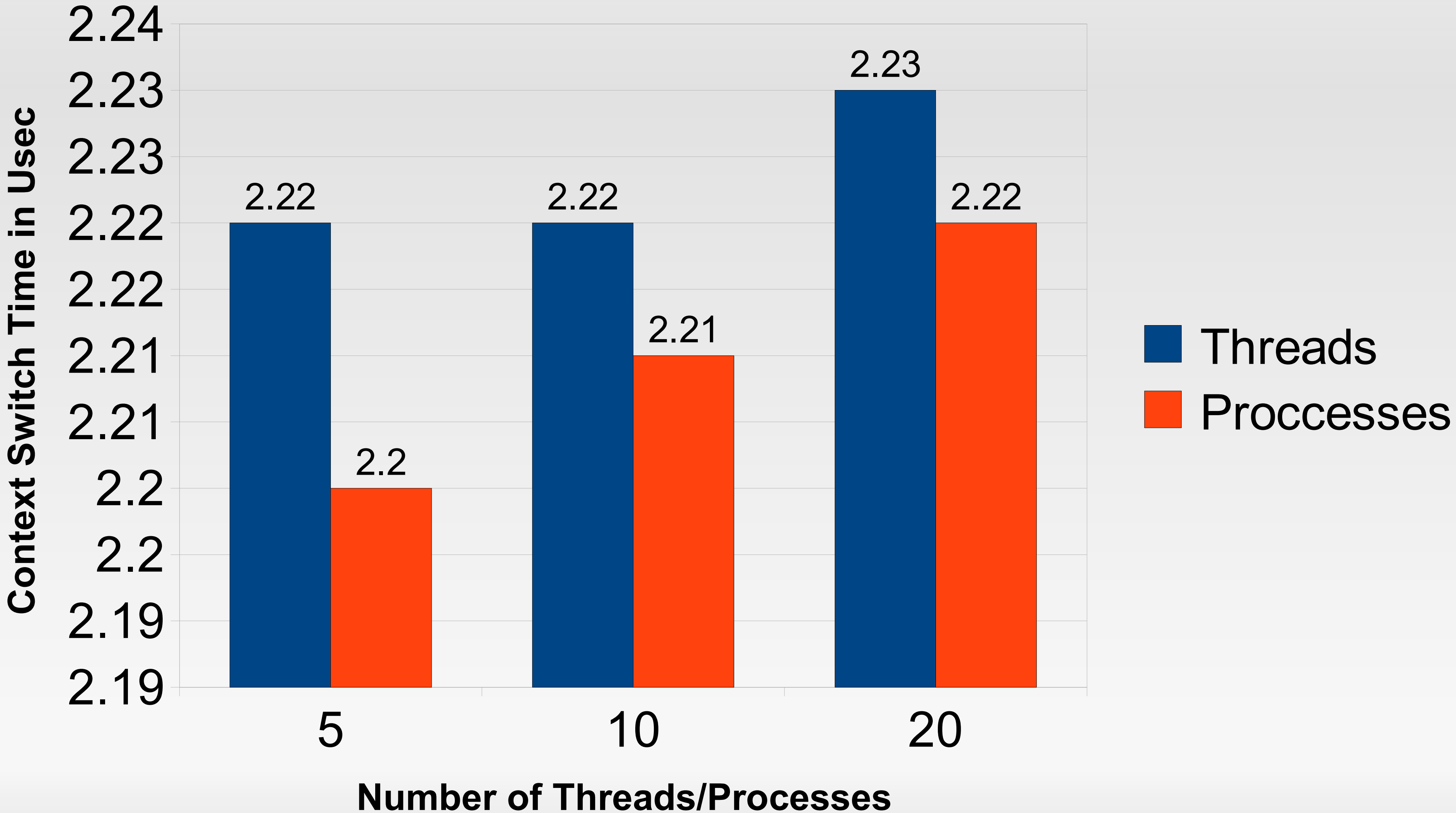
Codefidence
a name you can trust™

X86 Core2 Duo 2Ghz 0k data



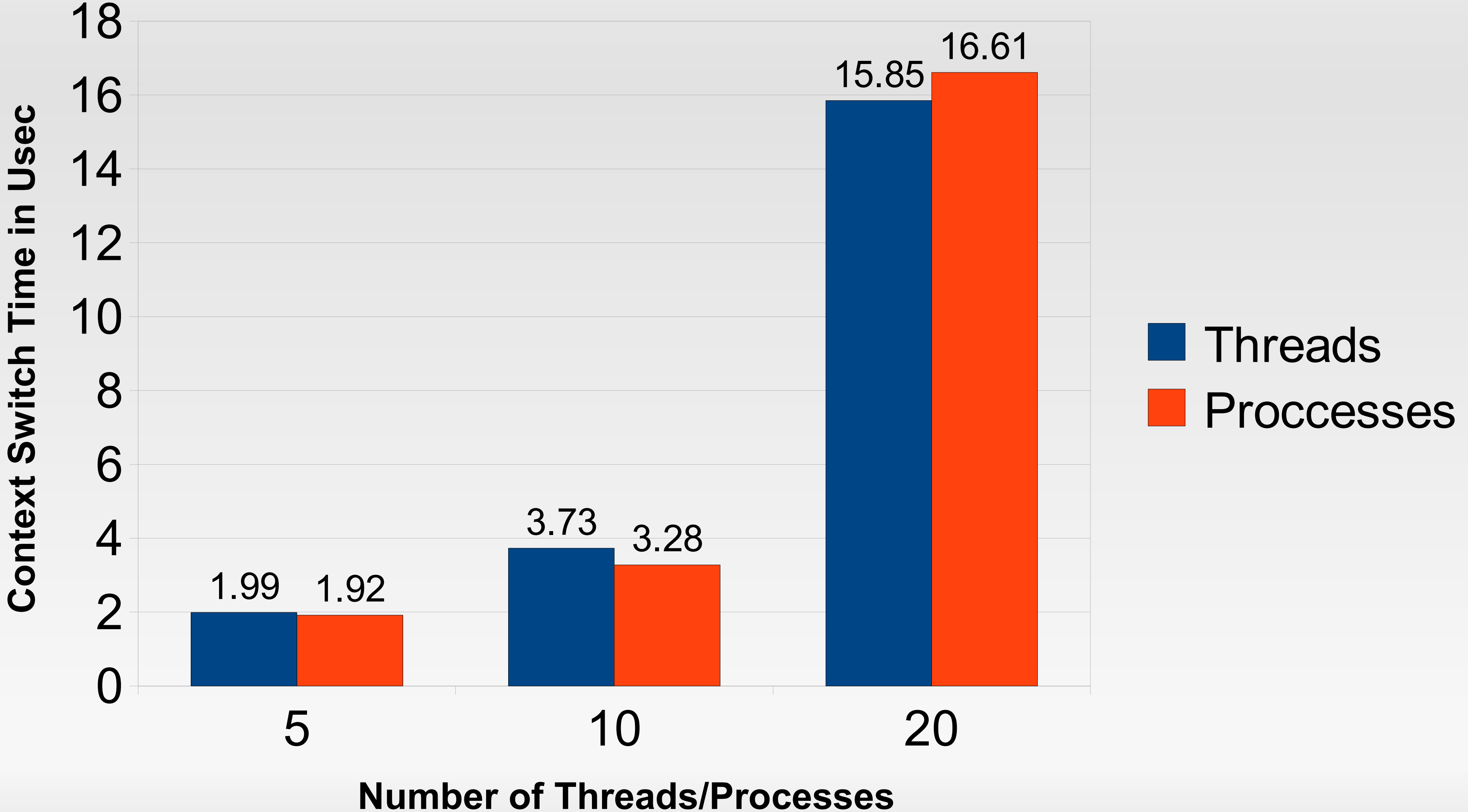
Codefidence
a name you can trust™

X86 Core2 Duo 2Ghz 16k data



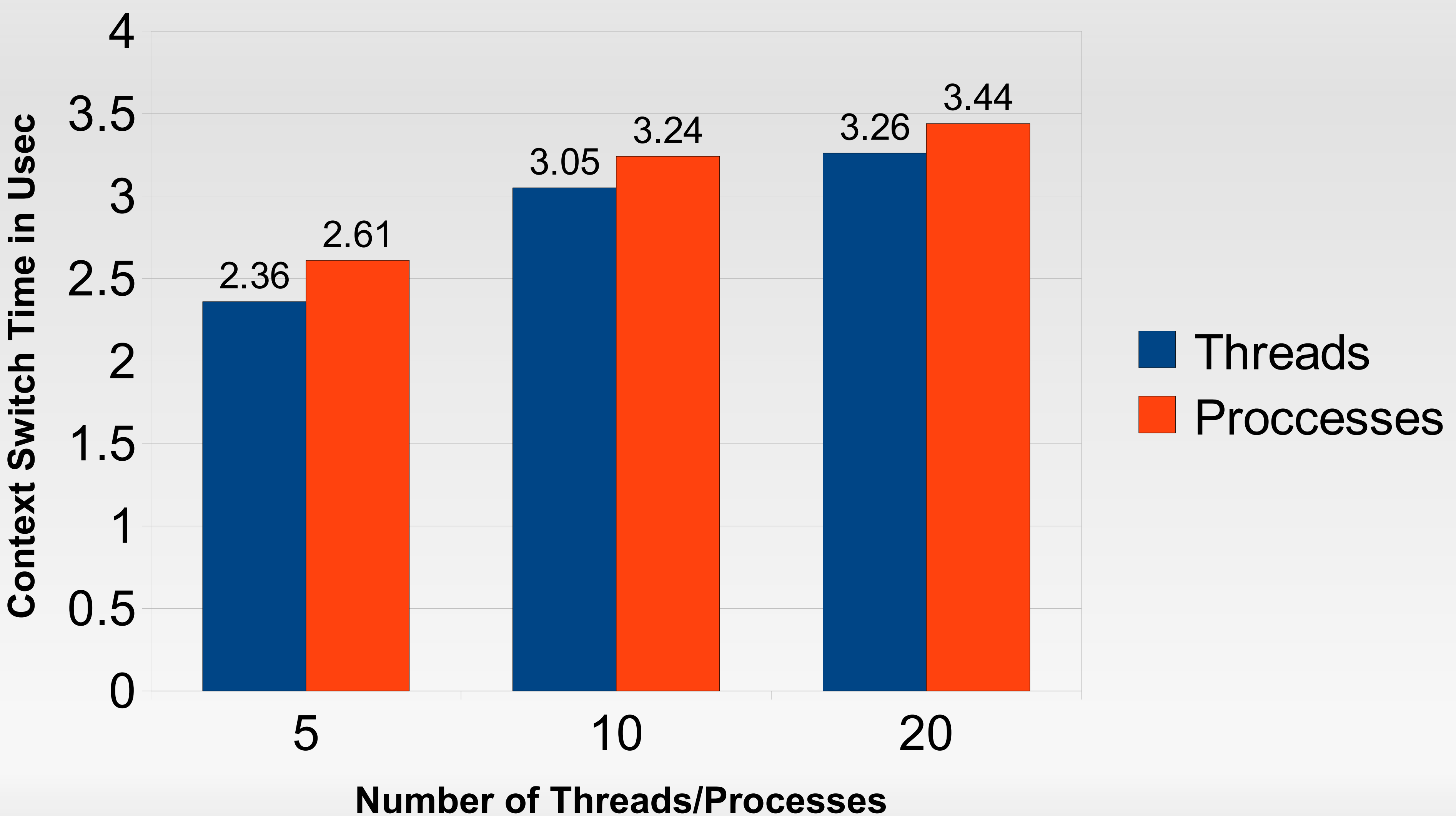
Codefidence
a name you can trust™

X86 Core2 Duo 2Ghz 128k data



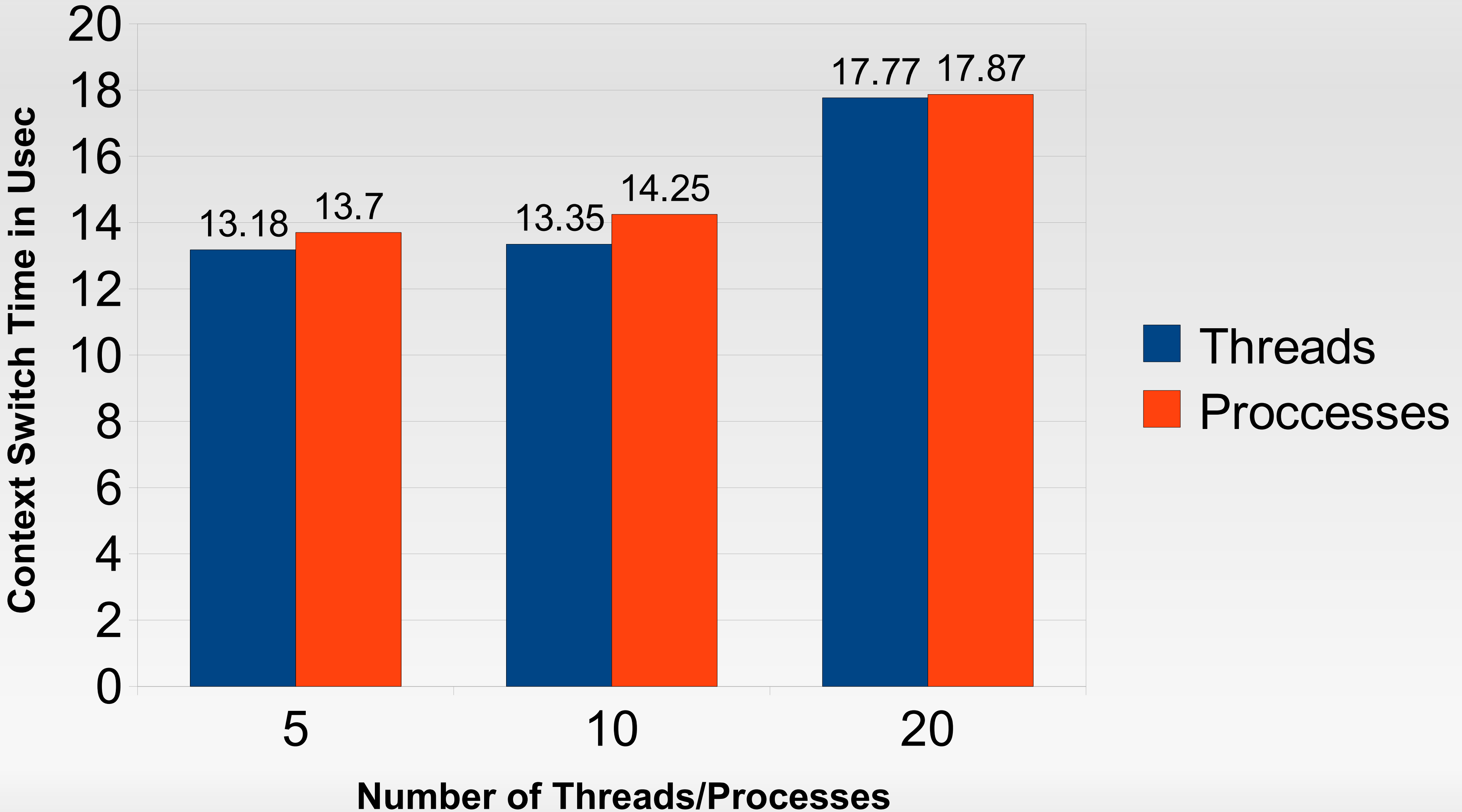
odefidence
a name you can trust™

PPC MPC8568 MDS 0k data

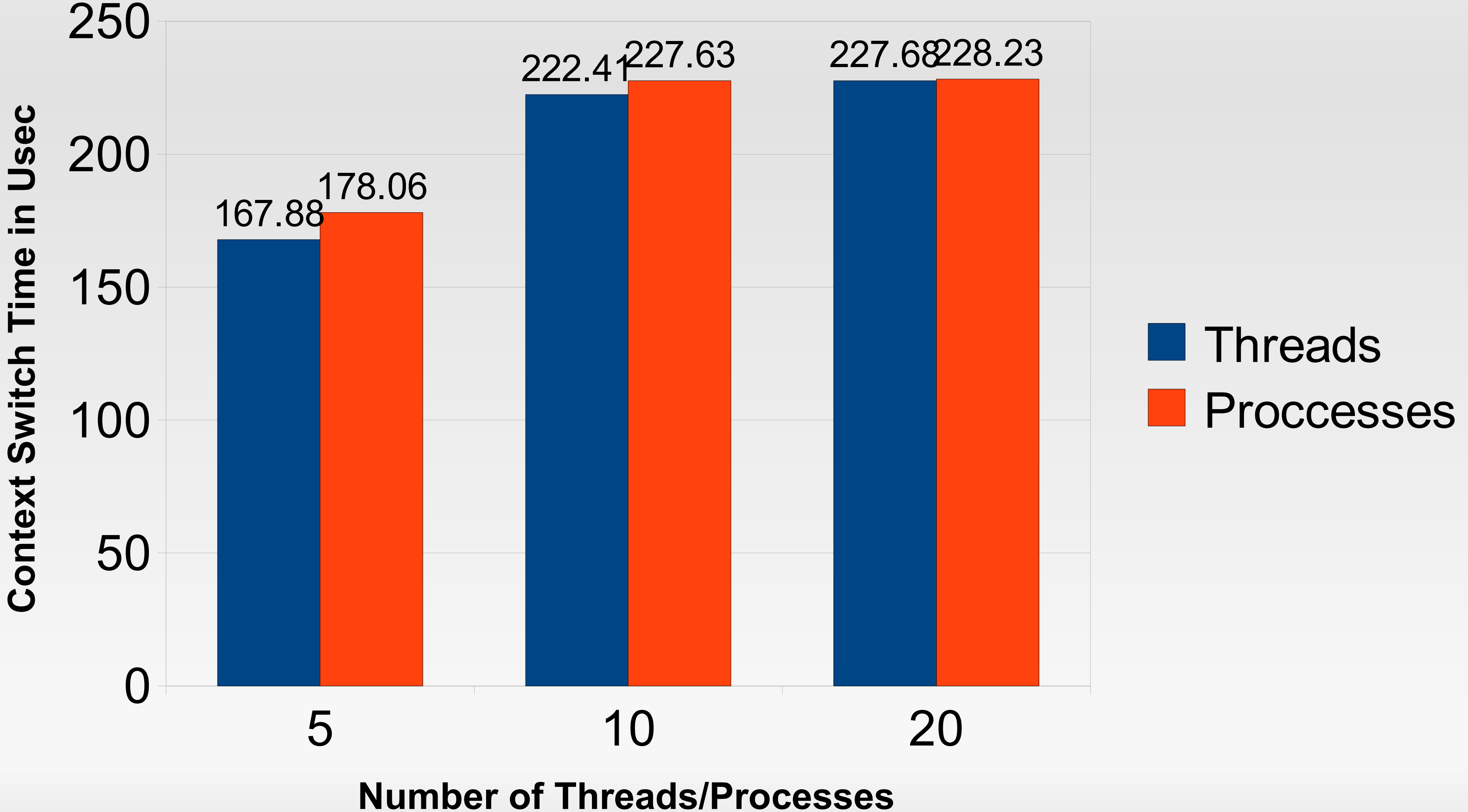


Codefidence
a name you can trust™

PPC MPC8568 MDS 16k data



PPC MPC8568 MDS 128k data



Conclusions

- Context switch times change between threads and processes.
- It is not a priori obvious that threads are better.
- The difference is quite small.
- The results vary between architectures and platforms.
- Why do we **really** use threads?

Codefidence
a name you can trust™

Why People Use Threads?

It's the API, Silly.

POSIX thread API offers simple mental model.

API Complexity: Task Creation

- **fork()**
  - Zero parameters.
  - Set everything yourself after process creation.
  - New process begins with a virtual copy of parent at same location
  - Copy on write semantics require shared memory setup
- **pthread_create()**
  - Can specify most attributes for new thread during create
  - Can specify function for new thread to start with
  - Easy to grasp address space sharing

API Complexity: Shared Memory

- Threads share all the memory.
- You can easily setup a segment of shared memory for processes using shm_create()
  - Share only what you need!
- BUT... each process may map shared segment at different virtual address.
  - Pointers to shared memory cannot be shared!
  - A simple linked list becomes complex.

Codefidence a name you can trust™

API Complexity: File Handles

- In Unix, Everything is a File.
- Threads share file descriptors.
- Processes do not.
  - Although Unix Domain Socket can be used to pass file descriptors between processes.
  - System V semaphores undo values not shared as well, unlike threads.
- Can make life more complicated.

Processes API Advantages

- **Processes**
  - PID visible in the system.
  - Can set process name via program_invocation_name.
  - Easy to identify a processes in the system.
- **Threads**
  - No way to name task in a unique name.
  - Kernel thread id not related to internal thread handle.
  - Difficult to ID a thread in the system.

Codefidence a name you can trust™

The CoProc Library

- CoProc is a proof of concept library that provides an API implementing share-as-you-need semantics for tasks
  - Wrapper around Linux clone() and friends.
- Co Processes offer a golden path between traditional threads and processes.
- Check out the code at: http://github.com/gby/coproc

Codefidence a name you can trust™

CoProc Highlights

- A managed shared memory segment, guaranteed to be mapped at the same virtual address
- Coproc PID and name visible to system
- Decide to share file descriptors or not at coproc creation time.
- Set attributes at coproc creation time.
  - Detached/joinable, address space size, core size, max CPU time, stack size, scheduling policy, priority, and CPU affinity supported.

CoProc API Overview

- `int coproc_init(size_t shm_max_size);`
- `pid_t coproc_create(char * coproc_name, struct coproc_attributes * attrib, int flags, int (* start_routine)(void *), void * arg);`
- `int coproc_exit(void);`
- `void * coproc_alloc(size_t size);`
- `void coproc_free(void * ptr);`
- `int coproc_join(pid_t pid, int * status);`

Codefidence a name you can trust™

CoProc Attributes

```
struct coproc_attributes {

  rlim_t address_space_size;   /* The maximum size of the process
                                   address space in bytes */
  rlim_t core_file_size;      /* Maximum size of core file */
  rlim_t cpu_time;            /* CPU time limit in seconds */
  rlim_t stack_size;          /* The maximum size of the process
                                  stack, in bytes */
  int    scheduling_policy;    /* Scheduling policy. */
  int    scheduling_param;    /* Scheduling priority or
                                 nice level */
  cpu_set_t cpu_affinity_mask;   /* The CPU mask of the co-proc */
};
```

Codefidence
a name you can trust™

CoProc Simple Usage Example

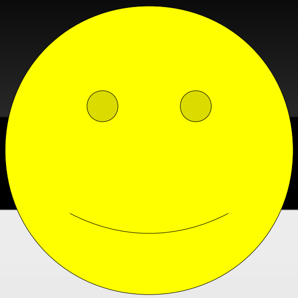
```
int pid, ret;
char * test_mem;
struct coproc_attributes = { ... };

coproc_init(1024 * 1024);

test_mem = coproc_alloc(1024);

if(!test_mem) abort();

pid = coproc_create("test_coproc", &test_coproc_attr, \
  COPROC_SHARE_FS ,test_coproc_func, test_mem);

if(pid < 0) abort();

coproc_join(pid, &ret);

coproc_free(test_mem);
```

Codefidence a name you can trust™

Credits

- The author wishes to acknowledge the contribution of the following people:
  - Larry McVoy and Carl Staelin for LMBench.
  - Joel Issacason, for an eye opening paper about a different approach to the same issue
  - Rusty Russel, for libantithreads, yet another approach to the same issue.
  - Free Electrons, for Virt/Phy slide.
  - Sergio Leone, Clint Eastwood, Lee Van Cleef, and Eli Wallach for the movie :-)

Thank You For Listening!



Questions?

- Codefidece Ltd.: http://codefidence.com
- Community site: http://tuxology.net
- Email: gilad@codefidence.com
- Phone: +972-52-8260388
- Twitter: @giladby
- SIP: gilad@pbx.codefidence.com
- Skype: gilad_codefidence

codefidence
a name you can trust™