Threads and DragonFly BSD

Conduits for program execution

- Concurrency

  A property that allows several vessels of execution to be run without a predefined order.

- Parallelism

  A property that allows vessels of execution to be run simultaneously.

Conduits for program execution

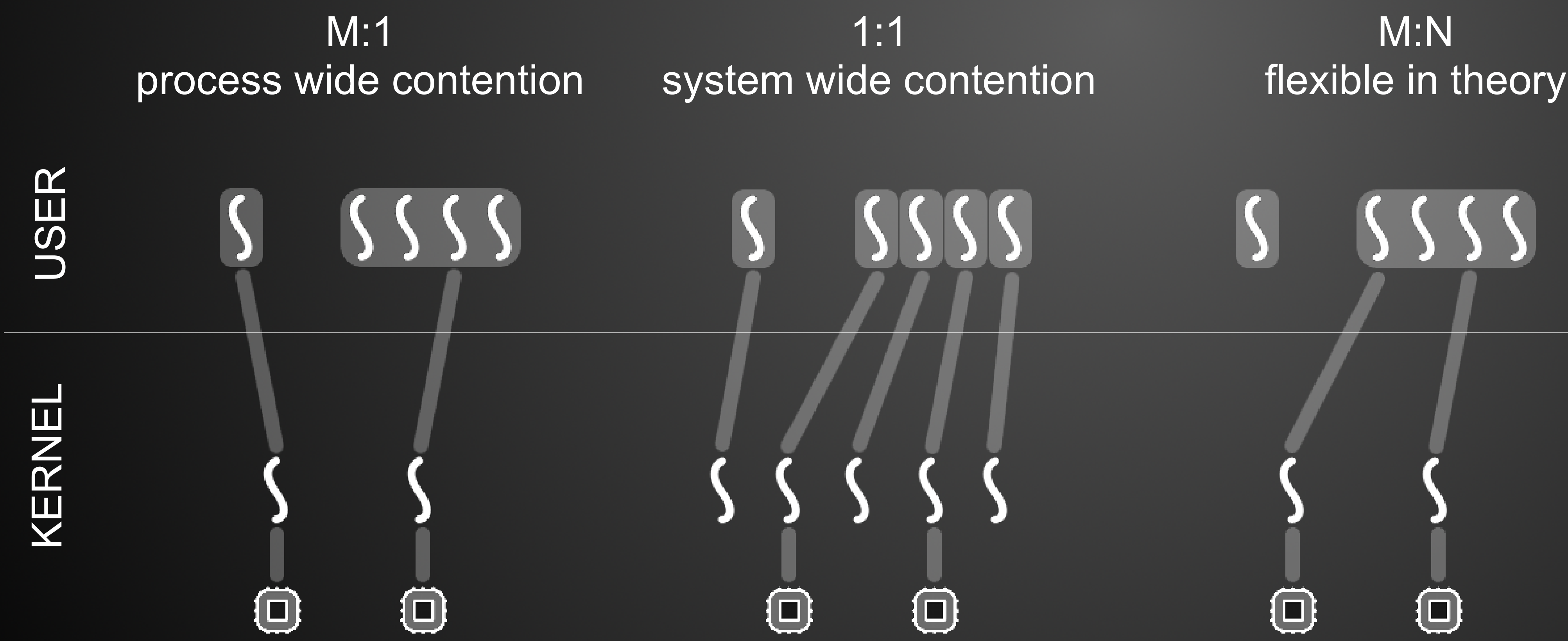
	Process	Thread
data	PID & parent PID signal state tracing information timers	thread state machine state user & kernel state scheduling statistics
structs	process group id user credentials VM management file descriptors resource accounting process statistics syscall() vectors signal actions thread list	

Conduits for program execution

Kernel Thread	User Thread
Provided by the kernel	Provided by a system library
has a kernel-stack	has a user-stack
scheduled by the kernel	scheduled by the user
	views kernel threads as execution contexts

Conduits for program execution

Contention Scope of Threading Models



Hypothesis

Thread performance in DragonFly could potentially be Improved using an M:N threading model.

- Threads are faster than processes in context switches
- No need to dive into kernel for scheduling
- Flexible contention scopes
- Pluggable schedulers through libraries linked at runtime

Hypothesis

Kernel support for user-mode threading could be done using a variant of 'unstable threads'. [Inohara et al]

- Kernel creates and terminates kernel-threads
- Shared memory communication areas
- Asynchronous user-thread scheduler
- Event notifier threads carrying information

Attempts at M:N Threading

-- SORT OF SUCCESSFUL --

Tru64	David Butenhof implemented a solid M:N system using a shared memory communication area for upcalls called “mxn”. Unfortunately it is closed source and phased out by HP-UX.

Attempts at M:N Threading

-- NOT AS SUCCESSFUL --

AIX	Used a proprietary M:N system for a long time but due to high customer demand it now defaults to 1:1
Solaris	Used M:N through SA (Scheduler Activations) for many years but bureaucracy forced a switch to 1:1
Linux	NGPT was about to offer M:N through SA but Ulrich Drepper and Ingo Molnar wrote the 1:1 NPTL and included it in glibc.
NetBSD	Nathan Williams implemented SA, but it was never “finished”
FreeBSD	Implemented a very sophisticated M:N system called Kernel Scheduled Entities, but it was never “finished”
Windows	Singularity only works with type-checked (.NET) programs
OS X	Never tried (publicly)

Notable Attempts at Pure User-Mode Threading

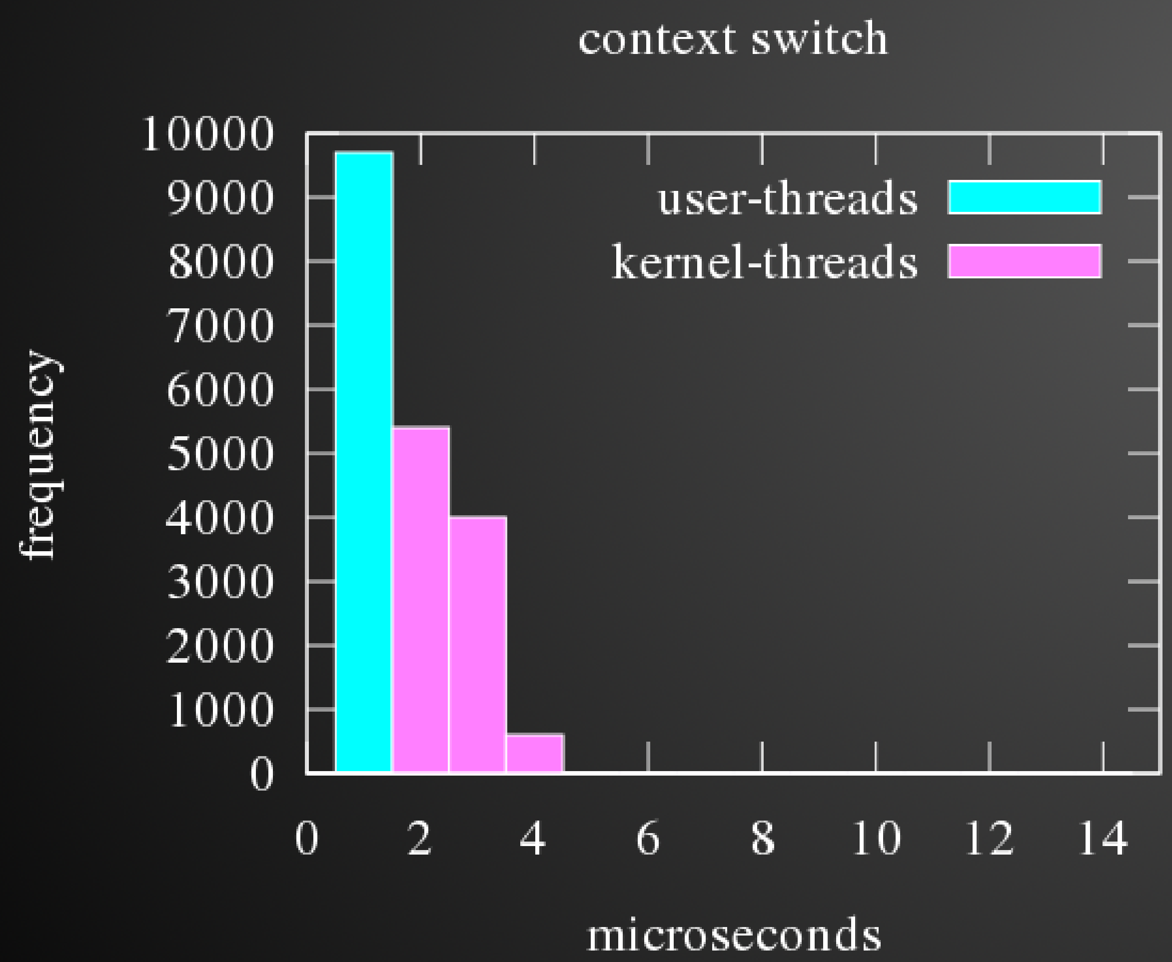
Erlang	A programming language which offers extremely cheap M:1 threads. Utilizes statistics to migrate them across CPU's and uses message passing for synchronization.
Pros:	Language support makes synchronization easy for the programmer.
	Facilitates use of concurrency for problem solving
Cons:	Message passing is bottleneck on SMP systems.
	Performs poorly on file I/O
	Co-operative thread can block the CPU scheduler
	Can't do real-time
	Not all problems are best solved by opening a million TCP sockets

Notable Attempts at Pure User-Mode Threading

Capriccio	A Ptherad library written at Berkeley. Achieves massive scaling by using Edgar Toernig's co-routine library, and co-operative scheduling.
Pros:	Easily juggles hundreds of thousands of user-threads Very very low context switching overhead
Cons:	Never implemented support for SMP systems. Performs poorly on file I/O Programs need to be “optimized” for co-operative scheduling.

Development

Thread ⟺ Thread Interaction



User threads were consistently faster by a few microseconds in every synthetic benchmark.

Development

Kernel ⟺ User Interaction

System calls take a few hundred nanoseconds

Diving into the kernel is slower than...
not diving into the kernel.

Development

Kernel ⟺ User Interaction

Thread ⟺ Thread Interaction

Problems

CPU bound workloads did not perform enough context switches to take advantage of user-threads

Many workloads exhibited significant delays that overshadowed the advantages of user-mode context switches.

Simple tasks that could be solved in the kernel followed complicated code paths.

Development

Handling Input / Output

"Upcall" to the user-thread scheduler, in true M:N style

Problem: All upcall mechanisms require many switches between kernel and user mode, which defeats the point of M:N.

Make all I/O non-blocking and asynchronous by using kqueue

Problem: It performs poorly during low concurrency or high cache misses. This is because of the many syscalls required of the mechanism.

Use shared memory FIFO TX/RX queues

Problem: It performs poorly during bursting I/O because the kernel needs to be kicked back on when there is a new entry on the FIFO.

Development

Interacting with the MMU

My computer's 2.6Ghz Core 2 Duo processor:

- Needs 2500 cycles to process a TCP packet.
- Needs 14 cycles for an L3 cache lookup. (0.5% performance hit)
- Needs 470 cycles after a basic cache miss. (19% performance hit)
- Needs 1040 cycles after an `invlpg` instruction. (41% performance hit)
- Has 119 documented bugs

mmap() & munmap() operations needed for a shared memory mechanism can be expensive and lead to "OS X" like performance penalties.

Ineffective decisions in schedulers result in a loss of cache-affinity.

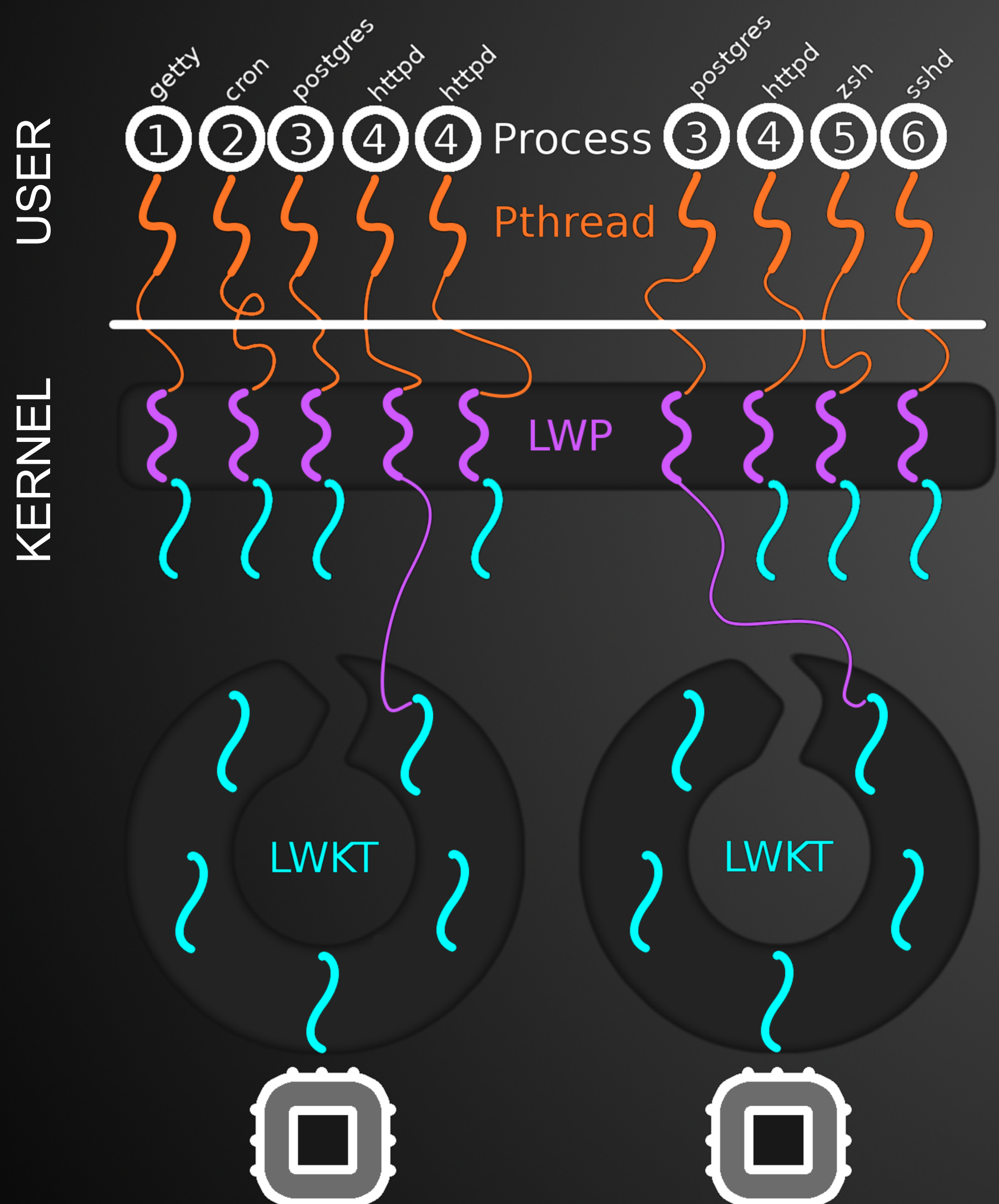
Development

Fine!!

We'll stick with 1:1

- Easiest to implement and maintain
- Easiest to debug
- Tried, tested, and proven
- Works now

Light Weight Kernel Threads

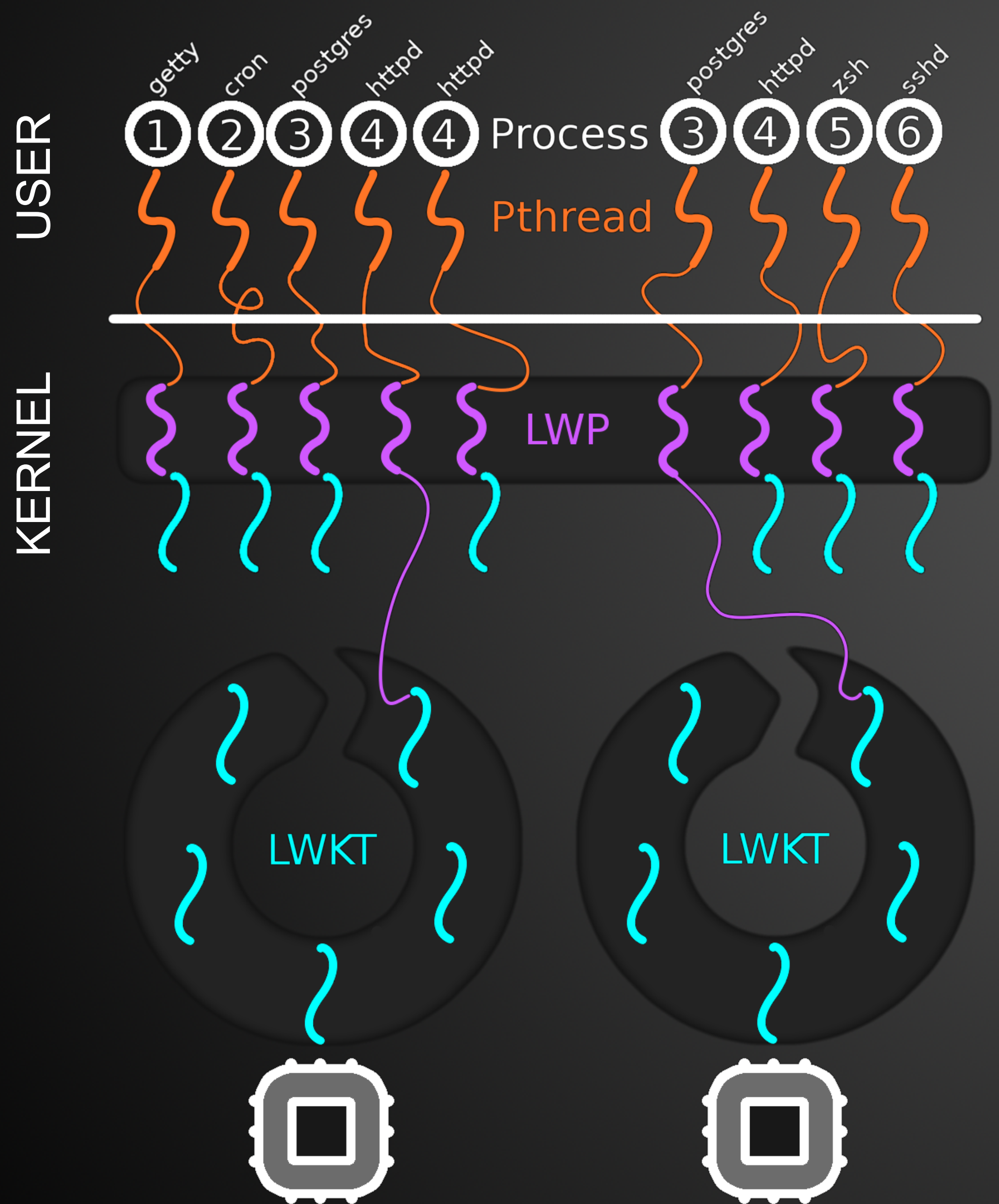


Pthread with user-mode stack, and struct containing thread attributes, id, and more

LWP only contains scheduling statistics, signal handler data, and some pointers between user-mode and kernel-mode.

Bound by proc struct which contains PID, VM space, file descriptors, and vnode

Light Weight Kernel Threads



LWKT's are scheduled In a round-robin manner, are bound to CPU's, and can have priorities

There could be several user-mode schedulers, each of which assigns an LWP to a LWKT

Simplifying Synchronization

LWKTs can communicate using messages

Generally require only a short critical section on same CPU

Use IPI messages to notify threads on other CPU's

Are very light-weight
Do not track memory mappings / pointers like Mach

Lockless Synchronization

Network stack is almost MP-safe

One TCP, UDP, ifnet, and netisr thread per CPU

Is nearly lock-free, with the exception of access from user-threads (which could be further tuned in the future).

Signs point toward excellent performance characteristics, but we have a few inter-process communication bugs to swat.

DragonFly - more than just threads.

HAMMER	we all use it (all 20 of us)
vkernel	DragonFly kernel can run as a user-mode process. Excellent for deveopment.
mistakes	survives USB flash-stick unplugging :-)
nimble	small team can make quick changes

Thank You For Listening!

For more information:
http://www.dragonflybsd.org