Barret Rhoden, Kevin Klues, Andrew Waterman, Ron Minnich, David Zhu, Eric Brewer

UC Berkeley and Google

High Level View

- Research OS, made for single nodes
  - Large-scale SMP / many-core architectures
  - Scheduling decisions made by the cluster manager
    - Enforced by the OS
- Support for high-performance, parallel apps
  - Transparent access to physical resources
  - M:N threading model (sort of)
  - Performance isolation / minimize interference
  - Mix of low latency and batch workloads

Low Latency vs Batch Workloads

- Live service jobs (low latency):
  - Minimize latency, especially tail latency
  - Predictable, efficient performance
  - Guaranteed resources for peak workload
- Batch jobs:
  - Low priority
  - Fill in the peak/average gap
  - No guarantee for resources

Minimize Tail Latency

Akaros offers:

- Spatially allocated, dedicated cores to processes
- User-level thread schedulers running at high frequency (or any frequency)
- Low frequency resource reallocation, driven by cluster managers
- Control over IRQ routing: no unexpected interrupts on dedicated cores

Provisioning vs. Allocation

Provisioning:

- Guaranteed future access to resources
- Used for low-latency services
  - Amount based on peak load
  - Amount used at any time may be less

Allocation:

- The actual granting of the resource (dynamic)
- When provisioned, uninterruptible, irrevocable
- Without, can be revoked at any time
  - Used for batch jobs

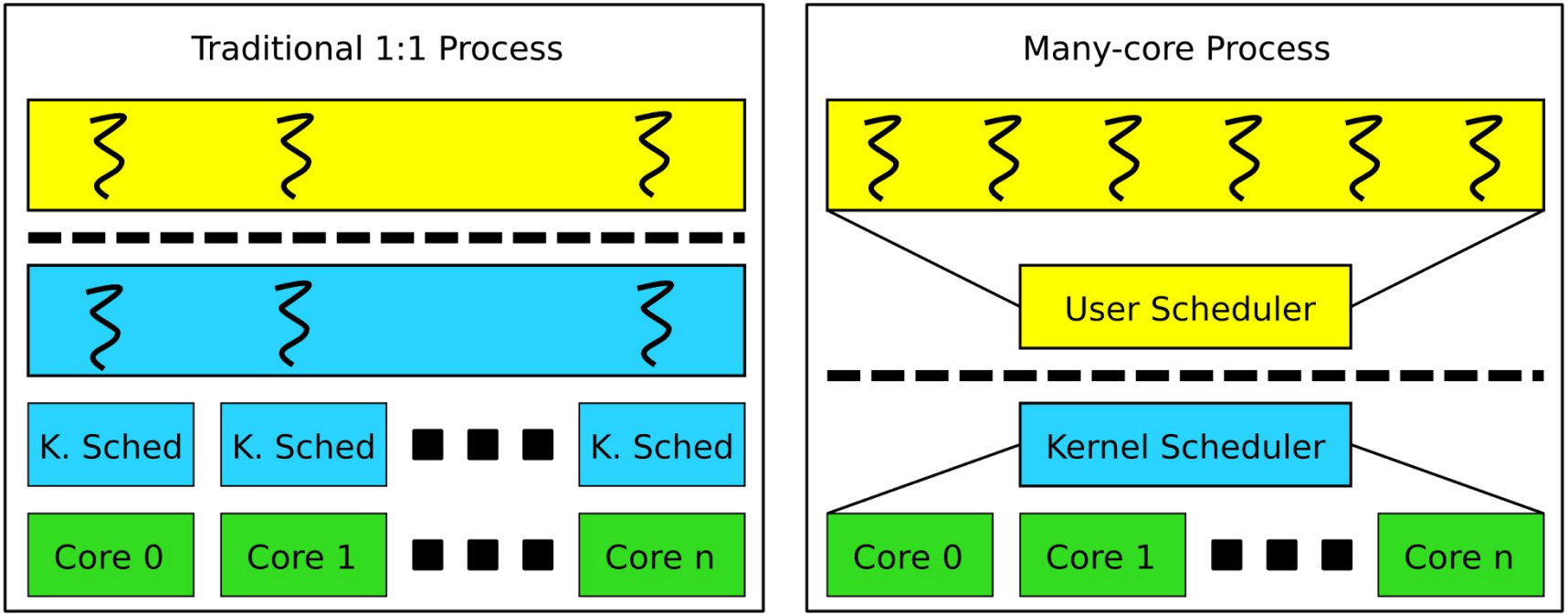
Transparent Resources

- Expose info about the underlying system
- Provide interfaces to control guaranteed, allocated resources
- Virtual resources for naming, not for deception
  - Processes use virtual memory and paging
  - Can view their page tables
  - Physical memory is pinned - no swapping

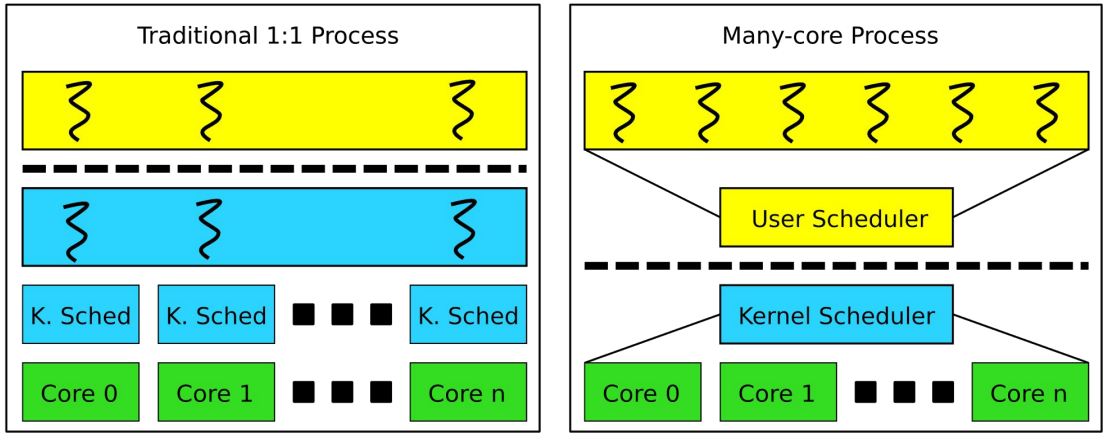
Classic Threading Models

- 1:1 - One kernel thread/task/process per user thread (Unix, Mesa/Cedar)
  - Heavy-weight threads, decisions made by kernel
- M:1 - Many user threads per kernel thread (Green threads, Capriccio)
  - If one thread blocks, the entire process stalls
- M:N - (Solaris's Light Weight Processes, Scheduler Activations, Psyche)
  - Akaros is M threads : N **cores**

Many-Core Process (MCP)



MCP



- Treat parallel processes as a single entity
  - Gang scheduled, no kernel thread per “pthread”/core
  - Single address space
- The process is aware of its state
  - Number of cores, which ones are running, etc
- Allows 2-Level scheduling (2LS), spinlocks, etc

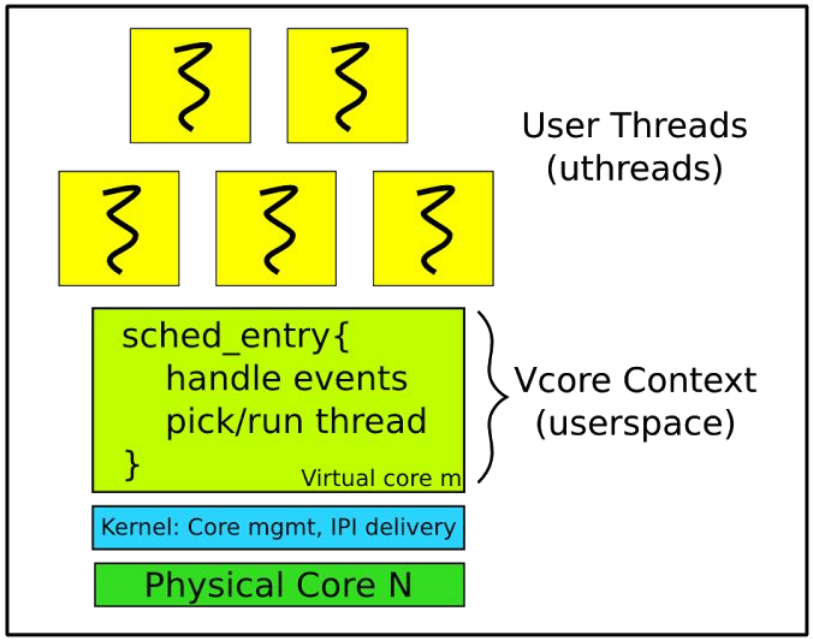
Cores != Threads

- Cores are for parallelism
- Threads are for concurrency (blocking I/O)
- Blocking (syscall, page fault) doesn’t mean the process loses the core
- Kernel threads are not part of the interface
- Notified of and can handle changing numbers of cores
- Process has full control over upcalls/events

Life for an MCP

- No unexpected interrupts
- Long time quanta
- Shared memory pages with the kernel
  - Procinfo (read-only), procdata (read-write)
- Have a set of virtual cores (vcores)
  - Pinned to physical cores when running
  - Can see the vcoremap
  - Each vcore has an “interrupt handling” context
- Schedule your own threads

Vcore Context



- Analogous to interrupt context in OSes
- Handles events and schedules threads
- Has its own stack and per-vcore storage
- Event driven
- IPIs / software IRQs disabled

Asynchronous Syscall Interface

- The struct syscall is the contract with the kernel
- The kernel may use threads and block internally, but userspace doesn’t know or care
- User threads (uthreads) that issue syscalls that blocked in the kernel hand off to the 2LS
- Userspace / 2LS can poll or request an event
- Can process syscalls on remote cores

What about Page Faults?

- Kernel will handle any soft faults (no blocking)
- Unhandled faults are reflected to userspace
- Faults in “vcore context” kill the process
- Pin critical code/data
- Uthreads that PF on file-backed mmaps are serviced by the 2LS via a syscall

Kernel Scheduling

- Different types of cores (can be dynamic)
- MCPs run on Coarse-Grained (CG) cores
  - No timer IRQs or per-core scheduler
  - Will run in kernel mode for IPIs for start-up/tear-down
- SCPs (single core processes), daemons, etc, run on Low-Latency (LL) cores
  - Management tasks, high frequency timer tick
  - Scheduler runs on an LL core (Core 0)

Kernel Perspective

- Monolithic kernel
- Can run the kernel anywhere; choose to run most of the kernel on a subset of cores
- Userspace determines where syscalls run
  - Locally, via sysenter/syscall traps into the kernel
  - Remotely, via shared memory rings (requires server)
- Designed to handle tricky circumstances
  - e.g. syscall completion event sent during preemption recovery of a lock-holder, while yielding spare cores

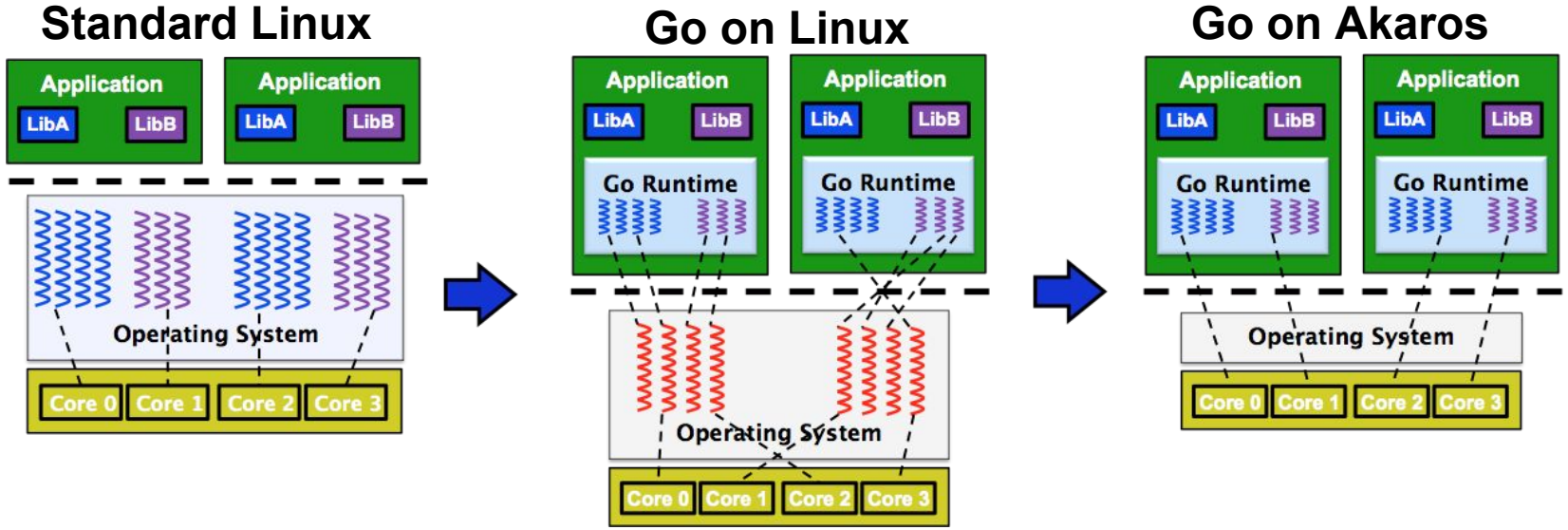
Akaros Programming Environment

- GCC toolchain, x86 and RISCV, 32/64 bit
- Glibc ported
- Some POSIX support (basic pthread apps)
- Plan 9 namespaces and network stack
- Ideal environment for Go!
- Custom extensions for Akaros (parlib)
- Barebones system (many things broken)

Plan 9 Stack

- Replacing our VFS with Plan 9 namespaces
  - Used Coccinelle to transform for Akaros
  - Ron and I can port a Plan 9 NIC driver in an hour
- Still have glibc, it just uses Plan 9 devices
- Work in progress to build mmap() for Plan 9
- Currently, we have an uneasy mix of VFS (with an in-memory FS) and Plan 9
- Plan 9's networking stack needs work

Go on Akaros



- User-level scheduling and high concurrency: ideal for Go
- High performance Go apps run directly inside an MCP
- **Passes 92% of the Go tests**
  - 1962 pass, 36 fail, 112 skipped, 2110 total

Early Evaluations / Microbenchmarks

- Intel Xeon E5-2670, 2.6GHz
- Sandy Bridge
- 16 Cores, 32 hyperthreads
- 256 GB RAM
- Linux 3.11, Ubuntu
- Akaros commit 0b940e7e

Thread Context Switch Latency

- Thread context-switch latency
- Pthread program:

```
pthread_thread() {
    for num_loops
        pthread_yield();
}
```

Thread Context Switch Latency

Values in nsec	**Linux Pthreads with TLS**	**Linux Uthreads with TLS**	**Linux Uthreads without TLS**	**Akaros Uthreads with TLS**	**Akaros Uthreads without TLS**
1 Thread	254	474	251	340	174
2 Threads	465	477	251	340	172
100 Threads	660	515	268	366	194
1000 Threads	812	583	291	408	221

- Thread local storage (TLS) hurts
- Uthread (2LS) scheduler is slow

Akaros User Context Switch Latency

Times in nsec	With TLS	No TLS	No Locking in Scheduler	No Locking, No asserts	Switch_to (bypass 2LS decision)
2 threads	340	172	95	88	55
100 threads	366	194	113	105	

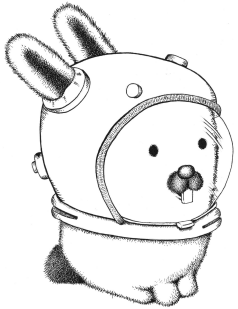
- TLS, dumb scheduler, untuned
- Akaros's user threading library (uthread.c) allows individual threads to have TLS or not
- All context switches drop into vcore context

Isolation, Interference, and Noise

- Fixed Time Quantum benchmark
  - Sottile and Minnich, *Analysis of Microbenchmarks for Performance Tuning of Clusters*, Cluster 2004
  - github.com/rminnich/ftq
- Perform work in a constant time interval
  - FTQ parameter: *frequency* of samples (e.g. 10KHz)
- FFT the result to detect periodic interference

Summary

- Akaros: research OS for high perf / parallel apps
- Provision and allocate ‘bare-metal’ resources
- Process model: cores != threads
- Go, Plan 9, and Glibc
- More info:
  - github.com/brho/akaros.git
  - http://akaros.cs.berkeley.edu/
- The giraffe’s name is Nanwan



AKAROS