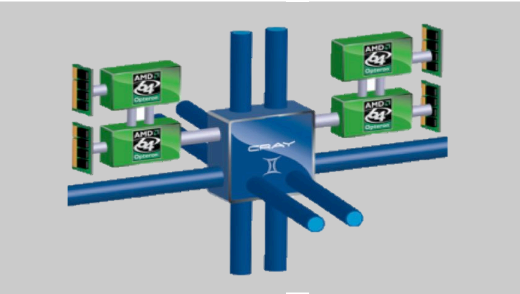
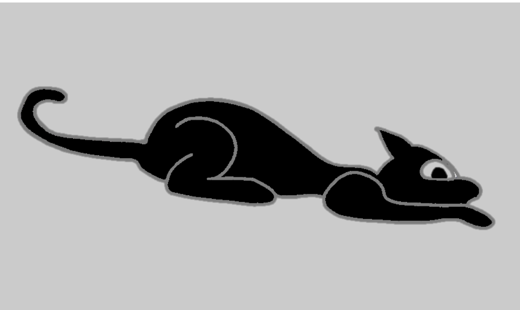
SAND2013-9592P





Kitten Lightweight Kernel Overview

October 28, 2013

Kevin Pedretti
Scalable System Software
Sandia National Laboratories
ktpedre@sandia.gov

Sandia National Laboratories

Exceptional service in the national interest

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

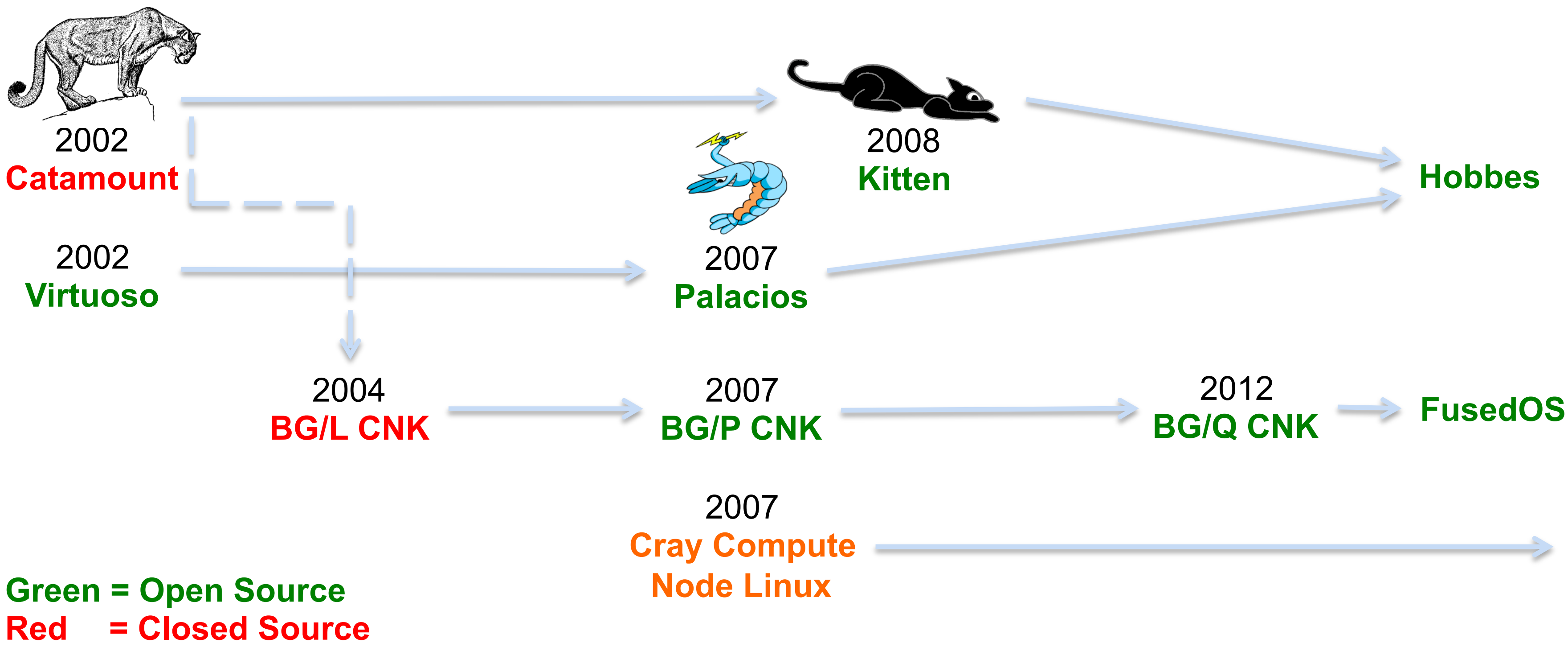
Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

Outline

- Background and Motivation
- Kitten Overview
  - Basic architecture
  - Physical memory management
  - Kernel memory
  - Address spaces
  - Tasks / Threads
  - System calls
  - Networking + I/O
  - Getting started
- Discussion

Lightweight Kernel Timeline

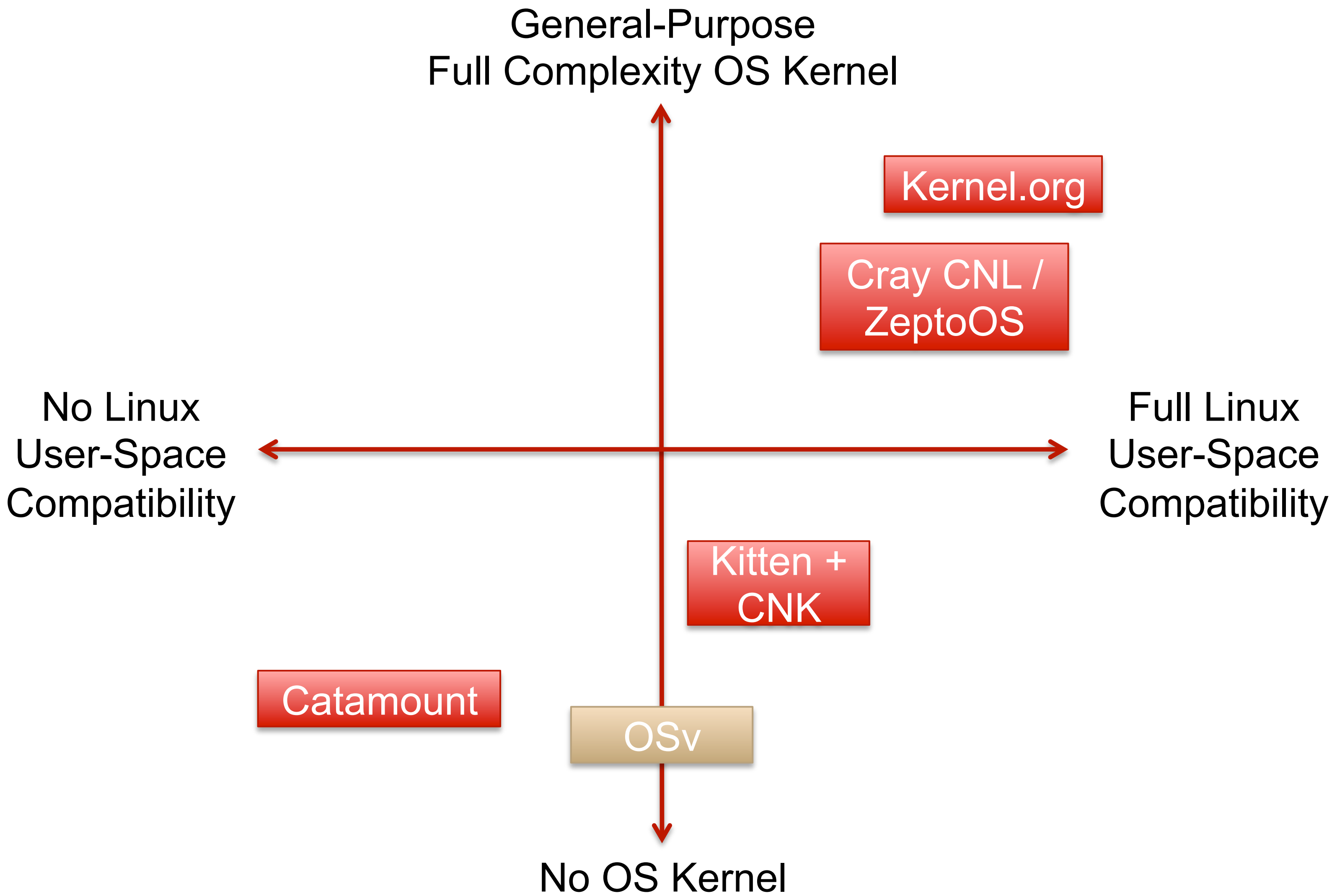
Sandia National Laboratories



- Kitten and CNK similar in concept
  - Both support Linux API subset and ABI compatibility
  - Kitten targets x86 (ARM underway), CNK targets PowerPC only
  - Kitten leverages Linux source code, CNK uses no Linux source code

- Palacios and Xen are both hypervisors
  - Palacios designed to be embeddable in a host OS, Kitten or Linux
  - Palacios is designed for HPC, low overhead, predictable performance
  - Palacios targets x86, Xen targets x86 + other archs

HPC OS Kernel Design Space



Motivation

- Catamount worked well, wanted LWK option to go forward
  - Less cognitive load to modify and extend compared to Linux
  - Lower bar to entry for HPC specific changes
  - Point of comparison against CNL
- To add HPC-specific OS-level functionality to Lightweight Linux
  - Must comprehend large Linux code base, complex interactions
  - Must keep forward porting changes, or get them into Linux (high bar)
  - Must work around issues not relevant to MPP-style HPC (e.g., memory pinning, swapping large page fragmentation, OOM killer)
- To add HPC-specific OS-level functionality to LWK
  - Must comprehend smaller codebase compared to Linux
  - Must convince smaller, HPC-oriented dev community (low bar)
  - No need to work around issues that should not exist for MPP-style HPC

Overall Design Goals for Kitten

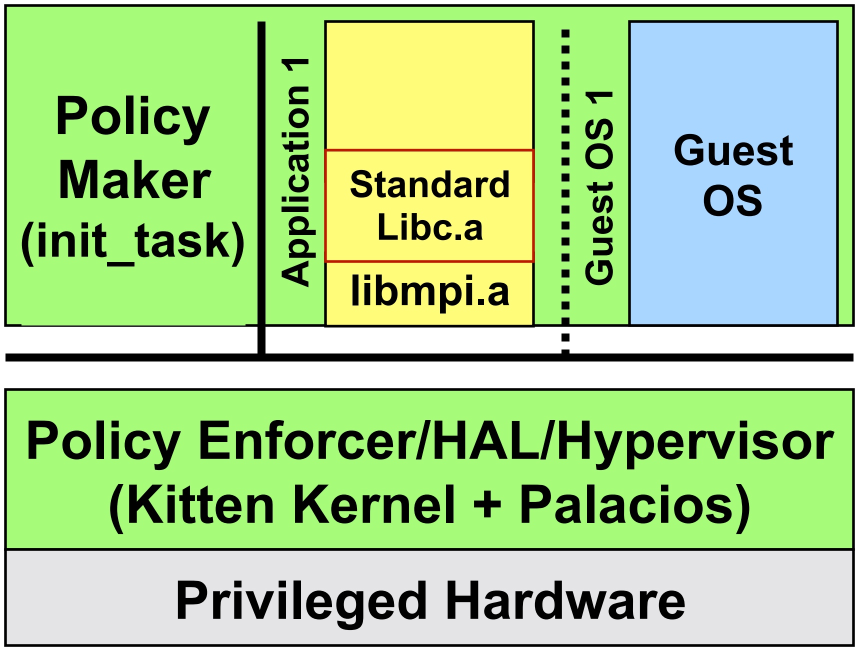
- **Support DOE's scientific computing application workloads running on extreme-scale, distributed-memory supercomputers with a tightly-coupled interconnect**

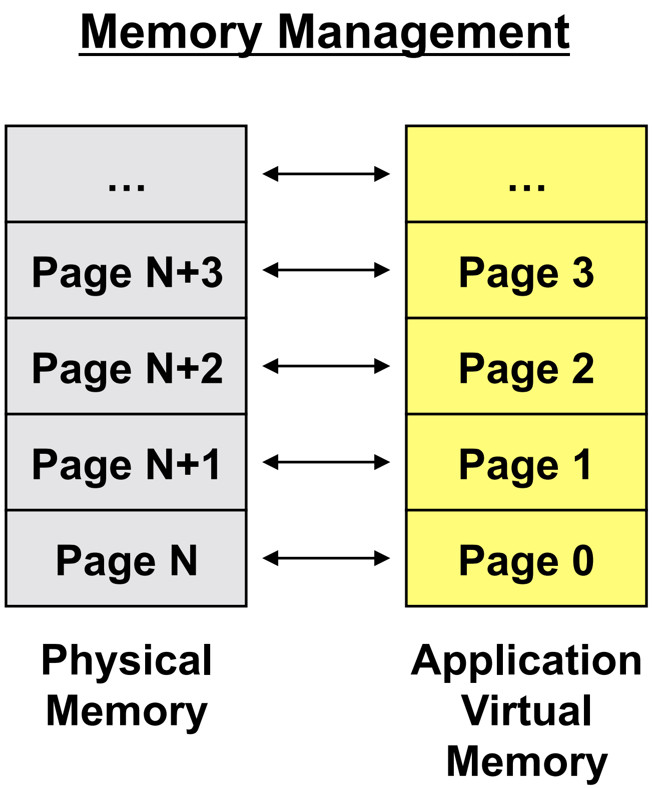
- Provide partial Linux API and ABI compatibility (fit in better)
- Add hypervisor capability for full OS support (LWK escape hatch)
- Maintain key characteristics of Catamount
- Build a good platform for HPC OS R&D

Outline

- Background and Motivation
- Kitten Overview
  - Basic architecture
  - Physical memory management
  - Kernel memory
  - Address spaces
  - Tasks / Threads
  - System calls
  - Networking + I/O
  - Getting started
- Discussion

Kitten Basic Architecture





- POSIX-like environment
- Inverted resource management
- Low noise OS noise/jitter
- Straight-forward network stack (e.g., no pinning)
- Less to go wrong, easier to harden

Kitten Kernel Implementation

- Monolithic, C code, GNU toolchain, Kbuild configuration
- Supports x86-64 architecture only, porting to ARM
  - Boots on standard PC architecture, Cray XT, and in virtual machines
  - Boots identically to Linux (Kitten bzImage and init_task)
- Repurposes basic functionality from Linux
  - Hardware bootstrap
  - Basic OS kernel primitives (lists, locks, wait queues, etc.)
  - Directory structure similar to Linux, arch dependent/independent dirs
- Custom address space management and task management
  - User-level API for managing physical memory, building virtual address spaces
  - User-level API for creating tasks, which run in virtual address spaces

Physical Memory Management

- Region based physical memory management
- Broadly separated into two partitions
  - Kmem (Kernel Memory)
  - Umem (User Memory)
- Kmem pool is fixed at boot time, doesn't grow
  - Size configurable using kmem_size boot parameter, 64 MB by default
  - Kernel uses kmem API to allocate kmem
- Umem pool managed by user-space
  - PCT uses pmem syscall API to allocate physical memory
  - PCT uses aspace syscall API to bind physical memory to address spaces

Pmem Region Data Structure

(include/lwk/pmem.h and kernel/mm/pmem.c)

```
/**
 * Defines a physical memory region.
 */
struct pmem_region {
        paddr_t         start;              /* region occupies: [start, end) */
        paddr_t         end;

        bool            type_is_set;        /* type field is set? */
        pmem_type_t     type;               /* physical memory type */

        bool            numa_node_is_set;   /* numa_node field is set? */
        numa_node_t     numa_node;          /* locality group region is in */

        bool            allocated_is_set;   /* allocated field set? */
        bool            allocated;          /* region is allocated? */

        bool            name_is_set;        /* name field is set? */
        char            name[32];           /* human-readable name of region */
};
```

Pmem Core API

(include/lwk/pmem.h and kernel/mm/pmem.c)

```
/* Add a region of physical memory to the pmem pool */
int pmem_add(const struct pmem_region *rgn);

/* Update a region of physical memory's meta-data */
int pmem_update(const struct pmem_region *update);

/* Find a region of physical memory meeting given criteria */
int pmem_query(const struct pmem_region *query,
               struct pmem_region *result);

/* Atomically query and mark result as allocated */
int pmem_alloc(size_t size, size_t alignment,
               const struct pmem_region *constraint,
               struct pmem_region *result);
```

Example Pmem Layout after Boot

- VMware guest configured for 4 GB memory:

```
Physical Memory Map:
    [0000000000000000, 0x0000000000083000) BOOTMEM    numa_node=0  (Bootstrap allocs)
    [0x0000000000083000, 0x000000000009f000) KMEM       numa_node=0
    [0x000000000009f000, 0x0000000000100000) BOOTMEM    numa_node=0  (BIOS reserved)
    [0x0000000000100000, 0x0000000000200000) KMEM       numa_node=0
    [0x0000000000200000, 0x0000000000413000) BOOTMEM    numa_node=0
    [0x0000000000413000, 0x0000000004000000) KMEM       numa_node=0
    [0x0000000004000000, 0x0000000004119000) INITRD     numa_node=0
    [0x0000000004119000, 0x0000000006162000) INIT_TASK  numa_node=0
    [0x0000000006162000, 0x00000000bfee0000) UMEM       numa_node=0
    [0x00000000bfee0000, 0x00000000bff00000) BOOTMEM    numa_node=0  (ACPI stuff)
    [0x00000000bff00000, 0x00000000c0000000) UMEM       numa_node=0
    [0x00000000c0000000, 0x0000000100000000) BOOTMEM    numa_node=0  (GPU, APIC, ...)
    [0x0000000100000000, 0x0000000140000000) UMEM       numa_node=0

Total User-Level Managed Memory: 4192722944 bytes
```

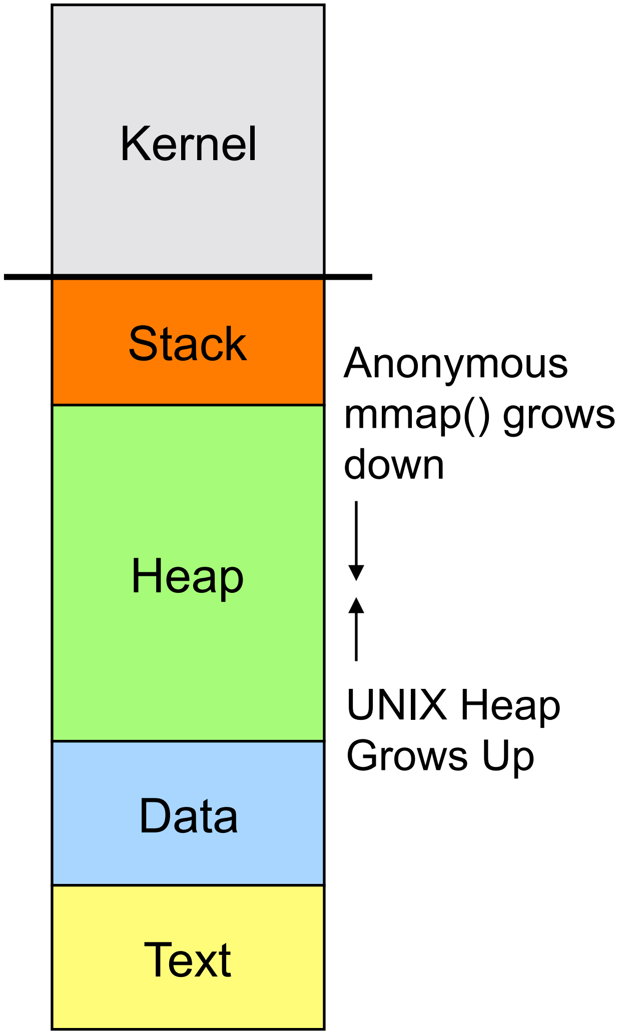
Kmem Management API

(include/lwk/kmem.h and kernel/mm/kmem.c)

- All Kmem managed by buddy allocator (kernel/mm/buddy.c)
- Two ways to allocate:
  - malloc() style — give me some memory
  - Page-based — give me a contiguous set of pages

```
/* malloc-style, implementation tracks block size internally */
extern void *kmem_alloc(size_t size);
extern void kmem_free( const void *addr);

/* page-based, caller must remember order of the block */
extern void *kmem_get_pages(unsigned long order);
extern void kmem_free_pages(const void *addr,
                            unsigned long order);
```

LWK Virtual Memory Regions



- User address space divided into virtual memory regions:
  - Text
  - Data
  - Heap
  - Stack
- Each region is mapped to a contiguous region of physical memory
  - Straightforward to use large pages
  - PCT in user-space sets up the mapping
- All virtual<->physical mapping occurs before application starts
  - No demand paging
  - No memory oversubscription

Aspace Management

Sandia National Laboratories

- Every execution context must execute in the context of a virtual address space, represented by an **aspace structure**
- After bootstrap, all address spaces have the kernel mapped into them above PAGE_OFFSET (matches Linux design)
  - Avoids context switch to enter kernel
  - Enables kernel threads to run without context switch
- Address space consists of non-overlapping virtual memory regions, each mapped to physical memory or hardware

- Currently no support for handing page faults
- In future may allow dynamic binding of virtual memory region to a physical memory pool for NUMA first-touch support

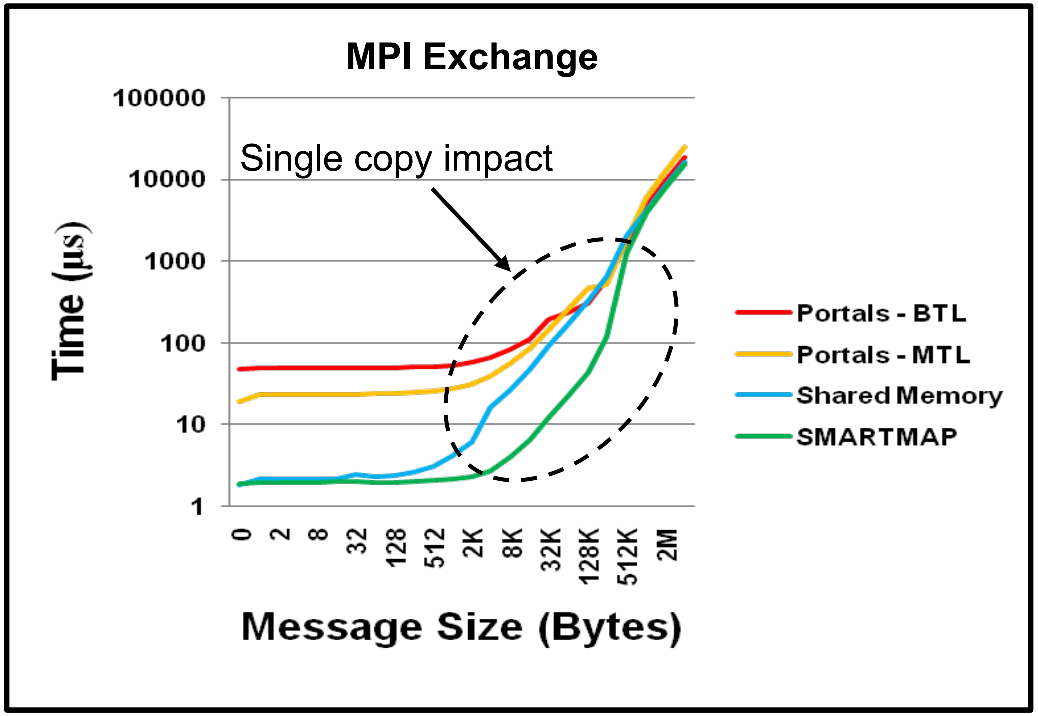
Aspace Core API

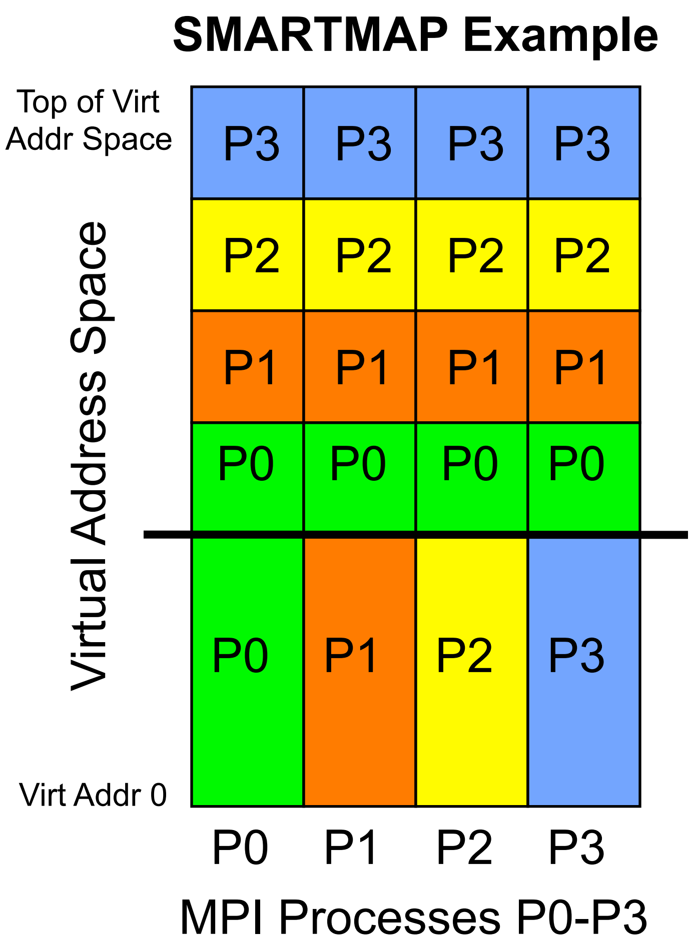
(include/lwk/aspace.h and kernel/mm/aspace.c)

```
/* Create a new aspace, possibly with a specific ID */
int aspace_create(id_t id_request, const char * name,
                  id_t *id);

/* Create a virtual memory region */
int aspace_add_region(id_t id, vaddr_t start, size_t extent,
                      vmflags_t flags, vmpagesize_t pagesz,
                      const char * name);

/* Map physical memory to a virtual memory region */
int aspace_map_pmem(id_t id, paddr_t pmem,
                    vaddr_t start, size_t extent);

/* Map one aspace into another at a given virtual address */
int aspace_smartmap(id_t src, id_t dst,
                    vaddr_t start, size_t extent);
```

SMARTMAP Intra-node Optimization Eliminates Unnecessary Memory Copies

- **Basic Idea: Each process on a node maps the memory of all other processes on the same node into its virtual address space**
- **Enables single copy process to process message passing (vs. multiple copies in traditional approaches)**





Task Management

- Every context of execution represented by a task
  - Each task is associated with an aspace
  - Threads implemented as multiple tasks associated with the same aspace
- Each task represented by a kernel-level task_struct
  - Contiguous block of memory including TCB and kernel stack (on x86)
  - Includes the task's permissions (uid/gid), fdtable, signal table, etc.
- Each CPU maintains its own task queue
  - Runnable tasks schedule round-robin
  - Blocked tasks are idle until they are woken up

Task Core API

(include/lwk/task.h and kernel/task.c)

```
/* Specifies the initial conditions to use when spawning a new task */
typedef struct {
    id_t        task_id;
    char        task_name[32];

    id_t        user_id;        // User ID the task executes as
    id_t        group_id;       // Group ID the task executes as
    id_t        aspace_id;      // Address space the task executes in
    id_t        cpu_id;         // CPU ID the task starts executing on

    vaddr_t     stack_ptr;      // Ignored for kernel tasks
    vaddr_t     entry_point;    // Instruction address to start executing at

    int         use_args;       // If true, pass args to entry_point()
    uintptr_t   arg[4];         // Args to pass to entry_point()
} start_state_t;

/* Spawn a new task with the requested start_state */
int task_create(const start_state_t *start_state, id_t *task_id);

int task_switch_cpus(id_t cpu_id);  /* allow task to migrate itself */
```

Thread Support

- Kitten user-applications link with standard GNU C library (Glibc) and other system libraries installed on the Linux build host
- Functionality added to Kitten to support Glibc NPTL POSIX threads implementation
  - Futex() system call (fast user-level locking)
  - Basic support for signals
  - Match Linux implementation of thread local storage
  - Support for multiple threads per CPU core, preemptively scheduled
- Kitten supports runtimes that work on top of POSIX threads
  - GOMP OpenMP implementation
  - Qthreads
  - Probably others with a little effort

Task Migration Optimization

Operating System	Round-trip Task Migration Time (task on core A migrates to core B, then back to A)
Linux 2.6.35.7	4435 ns
Kitten 1.3	2630 ns

Core-switching performance between two cores in the same Intel X5570 2.93 GHz processor. Kitten achieves a speedup of 1.7 compared to Linux, due to simpler implementation.

System Calls

- Kitten syscall calling conventions identical to Linux
- Syscall linkage defined in include/arch/unistd.h
- Syscall implementations
  - Linux syscall implementations kernel/linux_syscalls/
  - LWK specific syscalls kernel/lwk_syscalls
- General approach is to implement a Linux syscall when we find it is needed, only implement as much as is needed
- Current Linux syscall list, some are –ENOSYS stubs:
  - brk, clock_gettime, clone, close, dup2, dup, exit, exit_group, fcntl, fork, fstat, futex, getcpu, getdents64, getdents, getgid, getgroups, getpid, getrlimit, getrusage, gettid, gettimeofday, getuid, ioctl, kill, lseek, madvise, mkdir, mknod, mmap, mprotect, mremap, munmap, nanosleep, open, pipe, poll, read, readlink, readv, rmdir, rt_sigaction, rt_sigpending, rt_sigprocmask, sched_getaffinity, sched_yield, sethostname, set_robust_list, set_tid_address, settimeofday, stat, time, uname, unlink, wait4, write, writev

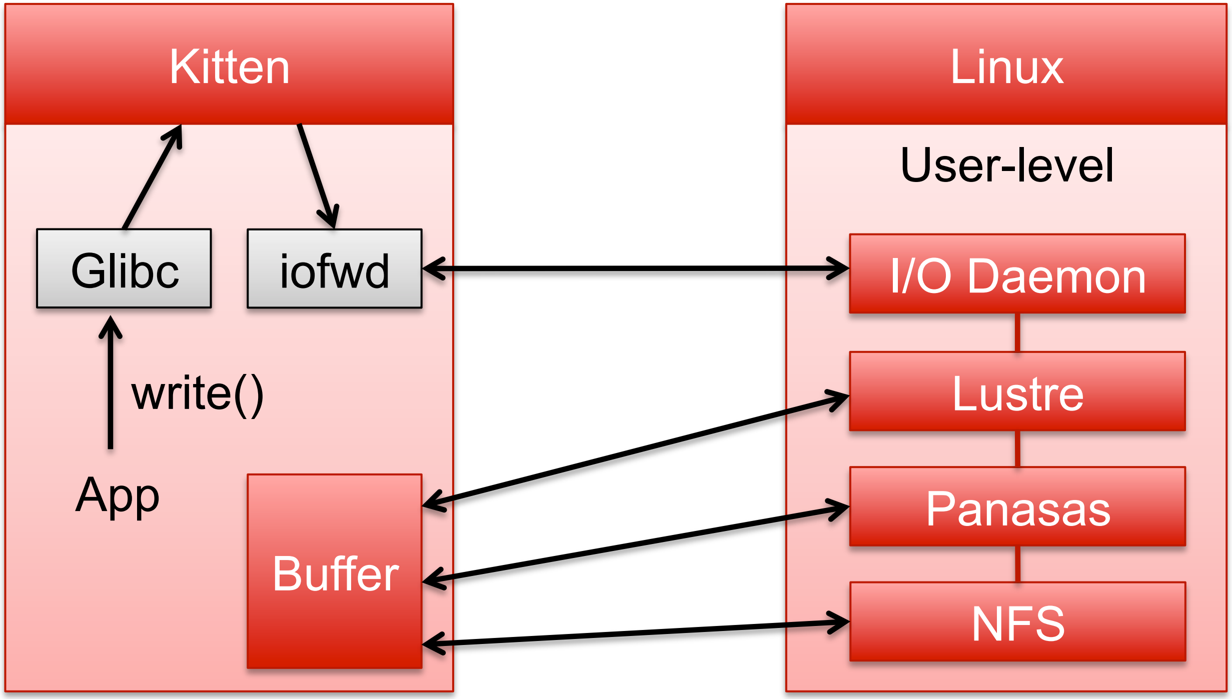
Kitten Networking

- Ported OFA Infiniband stack to Kitten a couple years ago
  - Implemented Linux compatibility layer to support OFA stack mostly unmodified
  - Turned out to be a lot of work
  - Difficult to make work on new IB clusters different than ours

- Recently started focusing on Portals4
  - Target Portals4 as lowest-level communication API
  - For development purposes, create implementations over Ethernet and (possibly) Infiniband
  - Portals4 reference implementation currently running in VMware virtual machine over VMware's virtual e1000 Ethernet device
    - Enables Kitten virtual cluster development environment

Kitten I/O Forwarding

- Prototype implementation developed over summer
- Influenced by IOFSL, wanted to use SMARTMAP and Portals
- Supports local files for drivers, forwards all else off node
- Kitten reflects off-node I/O calls to user-space
  - Avoids need for custom Glibc port
  - Only control reflected, no extra buffer copies



Other Bits

- Platform independent subsystems, rely on arch code to implement
  - ELF loader (mostly in user-space liblwk)
  - PCI enumeration, reads/writes to config space
  - Driver infrastructure
  - Interrupt registration and dispatch
  - Cross-calls
  - Timekeeping and timers
  - Console subsystem
  - KGDB support
- Job launch tool in progress
  - Similar to yod, aprun, mpirun, etc… Linux tool for launching Kitten apps
  - Uses Portals4 for all communication
  - Implements PMI over Portals4
  - I/O forwarding layer over Portals4

Getting Started

hg clone https://code.google.com/p/kitten

make menuconfig (chose all defaults)

make isoimage

- Then boot the isoimage wherever you'd like
- You should see a bunch of boostrap messages detailing the hardware detected
- Once boostrap is done, the "hello world" init task will be started
- You can replace the "hello world" init task with an ELF executable of your choosing (e.g., an OpenMP application)
- All binaries must be statically linked
- By default, init_task limited to 64 MB. To increase, either edit kernel/init_task.c to increase defaults or use kernel command line options:
  - init_heap_size=1073741824 init_stack_size=4194304

Backup

Sandia Lightweight Kernel Targets

Sandia National Laboratories

- Massively-parallel, distributed-memory machine with a tightly-coupled network
- Scientific and engineering modeling and simulation applications
- Enable fast message passing and execution
- Small memory footprint
- Deterministic performance
- Emphasize efficiency over functionality
- Maximize performance delivered to application

Reasons for a Specialized Approach

- Maximize available compute node resources
  - Maximize CPU cycles delivered to application
    - Minimize time taken away from application process
    - No daemons
    - No paging
    - Deterministic performance
  - Maximize memory given to application
    - Minimize amount of memory used for message passing
    - Static kernel size
  - Maximize memory bandwidth
    - Use large pages to avoid TLB misses, speed TLB miss handling
  - Maximize network resources
    - Physically contiguous memory layout
    - Simple address translation and validation, no pinning
- Increase reliability
  - Relatively small amount of source code
  - Reduced complexity
  - Support for small number of devices