Microkernels

Required reading: Improving IPC by kernel design

Overview

This lecture looks at the microkernel organization. In a microkernel, services that a monolithic kernel implements in the kernel are running as user-level programs. For example, the file system, UNIX process management, pager, and network protocols each run in a separate user-level address space. The microkernel itself supports only the services that are necessary to allow system services to run well in user space; a typical microkernel has at least support for creating address spaces, threads, and inter process communication.

The potential advantages of a microkernel are simplicity of the kernel (small), isolation of operating system components (each runs in its own user-level address space), and flexibility (we can have a file server and a database server). One potential disadvantage is performance loss, because what in a monolithich kernel requires a single system call may require in a microkernel multiple system calls and context switches.

One way in how microkernels differ from each other is the exact kernel API they implement. For example, Mach (a system developed at CMU, which influenced a number of commercial operating systems) has the following system calls: processes (create, terminate, suspend, resume, priority, assign, info, threads), threads (fork, exit, join, detach, yield, self), ports and messages (a port is a unidirectionally communication channel with a message queue and supporting primitives to send, destroy, etc), and regions/memory objects (allocate, deallocate, map, copy, inherit, read, write).

Some microkernels are more "microkernel" than others. For example, some microkernels implement the pager in user space but the basic virtual memory abstractions in the kernel (e.g, Mach); others, are more extreme, and implement most of the virtual memory in user space (L4). Yet others are less extreme: many servers run in their own address space, but in kernel mode (Chorus).

All microkernels support multiple threads per address space. xv6 and Unix

until recently didn't; why? Because, in Unix system services are typically implemented in the kernel, and those are the primary programs that need multiple threads to handle events concurrently (waiting for disk and processing new I/O requests). In microkernels, these services are implemented in user-level address spaces and so they need a mechanism to deal with handling operations concurrently. (Of course, one can argue if fork efficient enough, there is no need to have threads.)

L3/L4

L3 is a predecessor to L4. L3 provides data persistence, DOS emulation, and ELAN runtime system. L4 is a reimplementation of L3, but without the data persistence. L4KA is a project at sourceforge.net, and you can download the code for the latest incarnation of L4 from there.

L4 is a "second-generation" microkernel, with 7 calls: IPC (of which there are several types), id_nearest (find a thread with an ID close the given ID), fpage_unmap (unmap pages, mapping is done as a side-effect of IPC), thread_switch (hand processor to specified thread), lthread_ex_regs (manipulate thread registers), thread_schedule (set scheduling policies), task_new (create a new address space with some default number of threads). These calls provide address spaces, tasks, threads, interprocess communication, and unique identifiers. An address space is a set of mappings. Multiple threads may share mappings, a thread may grants mappings to another thread (through IPC). Task is the set of threads sharing an address space.

A thread is the execution abstraction; it belongs to an address space, a UID, a register set, a page fault handler, and an exception handler. A UID of a thread is its task number plus the number of the thread within that task.

IPC passes data by value or by reference to another address space. It also provide for sequence coordination. It is used for communication between client and servers, to pass interrupts to a user-level exception handler, to pass page faults to an external pager. In L4, device drivers are implemented has a user-level processes with the device mapped into their address space. Linux runs as a user-level process.

L4 provides quite a scala of messages types: inline-by-value, strings, and

virtual memory mappings. The send and receive descriptor specify how many, if any.

In addition, there is a system call for timeouts and controling thread scheduling.

L3/L4 paper discussion

- This paper is about performance. What is a microsecond? Is 100 usec bad? Is 5 usec so much better we care? How many instructions does 50-Mhz x86 execute in 100 usec? What can we compute with that number of instructions? How many disk operations in that time? How many interrupts can we take? (The livelock paper, which we cover in a few lectures, mentions 5,000 network pkts per second, and each packet generates two interrrupts.)
- In performance calculations, what is the appropriate/better metric? Microseconds or cycles?
- Goal: improve IPC performance by a factor 10 by careful kernel design that is fully aware of the hardware it is running on. Principle: performance rules! Optimize for the common case. Because in L3 interrupts are propagated to user-level using IPC, the system may have to be able to support many IPCs per second (as many as the device can generate interrupts).
- IPC consists of transfering control and transfering data. The minimal cost for transfering control is 127 cycles, plus 45 cycles for TLB misses (see table 3). What are the x86 instructions to enter and leave the kernel? (int, iret) Why do they consume so much time? (Flush pipeline) Do modern processors perform these operations more efficient? Worse now. Faster processors optimized for straight-line code; Traps/Exceptions flush deeper pipeline, cache misses cost more cycles.
- What are the 5 TLB misses: 1) B's thread control block; loading %cr3 flushes TLB, so 2) kernel text causes miss; iret, accesses both 3) stack and 4+5) user text - two pages B's user code looks at message
- Interface:
    - call (threadID, send-message, receive-message, timeout);
    - reply_and_receive (reply-message, receive-message, timeout);
- Optimizations:
    - New system call: reply_and_receive. Effect: 2 system calls per RPC.

- Complex messages: direct string, indirect strings, and memory objects.
- Direct transfer by temporary mapping through a communication window. The communication window is mapped in B address space and in A's kernel address space; why is this better than just mapping a page shared between A and B's address space? 1) Multi-level security, it makes it hard to reason about information flow; 2) Receiver can't check message legality (might change after check); 3) When server has many clients, could run out of virtual address space Requires shared memory region to be established ahead of time; 4) Not application friendly, since data may already be at another address, i.e. applications would have to copy anyway--possibly more copies.
- Why not use the following approach: map the region copy-on-write (or read-only) in A's address space after send and read-only in B's address space? Now B may have to copy data or cannot receive data in its final destination.
- On the x86 implemented by coping B's PDE into A's address space. Why two PDEs? (Maximum message size is 4 Meg, so guaranteed to work if the message starts in the bottom for 4 Mbyte of an 8 Mbyte mapped region.) Why not just copy PTEs? Would be much more expensive
- What does it mean for the TLB to be "window clean"? Why do we care? Means TLB contains no mappings within communication window. We care because mapping is cheap (copy PDE), but invalidation not; x86 only lets you invalidate one page at a time, or whole TLB Does TLB invalidation of communication window turn out to be a problem? Not usually, because have to load %cr3 during IPC anyway
- Thread control block registers, links to various double-linked lists, pgdir, uid, etc.. Lower part of thread UID contains TCB number. Can also dededuce TCB address from stack by taking SP AND bitmask (the SP comes out of the TSS when just switching to kernel).
- Kernel stack is on same page as tcb. why? 1) Minimizes TLB misses (since accessing kernel stack will bring in tcb); 2) Allows very efficient access to tcb -- just mask off lower 12 bits of %esp; 3) With VM, can use lower 32-bits of thread id to indicate which tcb; using

one page per tcb means no need to check if thread is swapped out (Can simply not map that tcb if shouldn't access it).

- Invariant on queues: queues always hold in-memory TCBs.
- Wakeup queue: set of 8 unordered wakeup lists (wakup time mod 8), and smart representation of time so that 32-bit integers can be used in the common case (base + offset in msec; bump base and recompute all offsets ~4 hours. maximum timeout is ~24 days, 2^31 msec).
- What is the problem addressed by lazy scheduling? Conventional approach to scheduling:

```
A sends message to B:
  Move A from ready queue to waiting queue
  Move B from waiting queue to ready queue
This requires 58 cycles, including 4 TLB misses.
  One each for head of ready and waiting queues
  One each for previous queue element during the
```

- Lazy scheduling:

```
Ready queue must contain all ready threads except
  Might contain other threads that aren't actuall
Each wakeup queue contains all threads waiting in
  Again, might contain other threads, too
  Scheduler removes inappropriate queue entries w
  queue
```

- Why does this help performance? Only three situations in which thread gives up CPU but stays ready: send syscall (as opposed to call), preemption, and hardware interrupts. So very often can IPC into thread while not putting it on ready list.
- Direct process switch. This section just says you should use kernel threads instead of continuations.
- Short messages via registers.
- Avoiding unnecessary copies. Basically can send and receive messages w. same vector. Makes forwarding efficient, which is important for Clans/Chiefs model.
- Segment register optimization. Loading segments registers is slow, have to access GDT, etc. But common case is that users don't change their segment registers. Observation: it is faster to check that

segment descriptor than load it. So just check that segment registers are okay. Only need to load if user code changed them.

- Registers for paramater passing where ever possible: systems calls and IPC.
- Minimizing TLB misses. Try to cram as many things as possible onto same page: IPC kernel code, GDT, IDT, TSS, all on same page. Actually maybe can't fit whole tables but put the important parts of tables on the same page (maybe beginning of TSS, IDT, or GDT only?)
- Coding tricks: short offsets, avoid jumps, avoid checks, pack often-used data on same cache lines, lazily save/restore CPU state like debug and FPU registers. Much of the kernel is written in assembly!
- What are the results? figure 7 and 8 look good.
- Is fast IPC enough to get good overall system performance? This paper doesn't make a statement either way; we have to read their 1997 paper to find find the answer to that question.
- Is the principle of optimizing for performance right? In general, it is wrong to optimize for performance; other things matter more. Is IPC the one exception? Maybe, perhaps not. Was Liedtke fighting a losing battle against CPU makers? Should fast IPC time be a hardware, or just an OS issue?