```
6.S081 2020 L18: Operating System Organization, Microkernels

Topic:
  What should a kernel do?
  What should its abstractions / system calls look like?

Answers depend on the application, and on programmer taste!
  There is no single best answer
  This topic is more about ideas and less about specific mechanisms

The traditional approach
  1) powerful abstractions, and
  2) a "monolithic" kernel implementation
  UNIX, Linux, xv6

The philosophy behind traditional kernels is powerful abstractions:
  portable interfaces
    files, not disk controller registers
    address spaces, not MMU access
  simple interfaces that hide complexity
    all I/O via FDs and read/write, not specialized for each device &c
    address spaces with transparent disk paging
  abstractions help the kernel manage and share resources
    process abstraction lets kernel be in charge of scheduling
    file/directory abstraction lets kernel be in charge of disk layout
  abstractions help the kernel enforce security
    file permissions
    processes with private address spaces
  lots of indirection
    e.g. FDs, virtual addresses, file names, PIDs
    helps kernel virtualize, hide, revoke, schedule, &c

Powerful abstractions have led to big "monolithic" kernels
  kernel is one big program, like xv6
  easy for kernel sub-systems to cooperate -- no irritating boundaries
    exec() and mmap() are part of both FS and VM system
    relatively easy to add sym links, COW fork, mmap, &c
  all kernel code runs with high privilege -- no internal security restrictions

What's wrong with traditional kernels?
  big => complex => buggy/insecure
  perhaps over-general and thus slow
    how much code executes to send one byte via a UNIX pipe?
    buffering, locks, sleep/wakeup, scheduler
  many design decisions are baked in, can't be changed, may be awkward
    maybe I want to wait for a process that's not my child
    maybe I want to change another process's address space
    maybe DB is better at laying out B-Tree files on disk than kernel FS
  hard to create kernel "extensions" that others can use
    new device drivers, file systems, &c

Microkernels -- a different approach
  big idea: move most O/S functionality to user-space service processes
  [diagram: h/w, kernel, services (FS disk VM TCP NIC display), apps]
  kernel can be small
    address spaces, threads, IPC (inter-process communication)
    IPC lets threads send each other messages
  1980s saw big burst of research on microkernel designs
    CMU's Mach perhaps the most influential
  used today in embedded systems, phone chips, car entertainment
  ideas (esp user-level servers and IPC) influential e.g. Windows and MacOS

Why the interest in microkernels?
  focused, elegant, clean slate
```

```
    small -> more security -- less code means fewer bugs to exploit
    small -> verifiable (see seL4)
    small -> easier to optimize
      you don't have to pay for features you don't use
    small -> avoid forcing design decisions on applications
    user-level -> may encourage modularity of O/S services
    user-level -> easier to extend / customize / replace user-level services
    user-level -> more robust -- restart individual user-level services
      most bugs are in drivers, get them out of the kernel!
    can run/emulate multiple O/Ses, like a VMM

  Microkernel challenges
    What's a minimum kernel API?
    Need simple primitives on which to build exec, fork, mmap, &c
    Need to build the rest of the O/S at user level
    How to get good performance, despite IPC and less integration?

  L4
    has evolved over time, many versions and re-implementations
    used commercially today, in phones and embedded controllers
    representative of the micro-kernel approach
    emphasis on minimality:
      7 system calls (Linux has 300+, xv6 has 21)
      13,000 lines of code

  L4 basic abstractions
    [diagram]
    address space ("task")
    thread
    IPC

  L4 system calls:
    create an address space
    create/destroy a thread in [another] address space
    send/recv message via IPC (addresses are thread IDs)
    map pages of your memory into another address space
      it must agree
      this happens via IPC -- one task can modify another task's page table
      used to create new tasks, share memory
    intercept another address space's page faults -- "pager"
      kernel delivers via IPC
    access device hardware (not a system call, happens directly)
    handle device interrupts
      kernel delivers via IPC

  Note L4 kernel is missing almost everything that Linux or even xv6 has
    file system, fork(), exec(), pipes, device drivers, network stack, &c
    If you want these, they have to be user-level code
      library or server process

  how does L4 thread switching work?
    current user-level thread can yield for 3 reasons:
      IPC system call waits
      timer interrupt
      yield() system call
    L4 kernel saves user thread registers,
      picks a RUNNABLE thread to run,
      restores user registers,
      switches page table,
      jumps to user space
    no surprises here

  how do L4 external pagers work?
    every task has a pager task
    1. page fault
```

```
    2. kernel suspends thread
    3. kernel sends fault info in IPC to pager
    4. pager picks one its own pages
    5. pager sends virtual page address in IPC reply to faulting thread
    6. kernel intercepts IPS, maps in target, resumes target

  what can you use an L4 pager for?
    allocating memory -- "sigma0" allocates on fault for early tasks
    copy-on-write fork
      coupled with a system call that revokes access
    mmap of file

  problem: IPC performance
    Microkernel programs do lots of IPC!
    Was expensive in early systems
      multiple kernel crossings, TLB misses, context switches, &c
    Cost of IPC caused many to dismiss microkernels
    L4 designers put huge effort into IPC performance

  Here's a slow IPC design
    patterned on UNIX pipes
    [diagram, message queue in kernel]
    send(id, msg)
      append msg to queue in kernel, return
    recv(&id, &data)
      if msg waiting in queue, remove, return
      otherwise sleep()
    called "asynchronous" and "buffered"
    now the usual request-response pattern (RPC) involves:
      [diagram: 2nd message queue for replies]
      4 system calls (user->kernel->user)
        send() -> recv()
        recv() <- send
        each may disturb CPU's caches (TLB, data, instruction)
      four message copies (two for request, two for reply)
      two context switches, two general-purpose schedulings

  L4's fast IPC
    "Improving IPC by Kernel Design," Jochen Liedtke, 1993
    * synchronous
      [diagram]
      send() waits for target thread's recv()
      common case: target is already waiting in recv()
      send() jumps into target's user space, as if returning from recv()
        no real context switch, no scheduler loop
    * unbuffered
      no queue in kernel
      since synchronous, kernel can copy directly between user buffers
    * small messages in registers
      kernel send() path does not disturb many of the registers
        e.g., no context switch
      no copying required for small messages
        since send() jumps into target's user space, along with registers
    * huge messages as virtual memory grants
      again, no copy required, though kernel send() code must change page table
    * combined call() and sendrecv() system calls
      [diagram]
      IPC almost always used as request-response RPC
      thus wasteful to use separate send() and recv() system calls
      client: call(): send a message, wait for response
      server: sendrecv(): reply to one request, wait for the next one
      2x reduction in user/kernel crossings
    * careful layout of kernel code to minimize cache footprint
    result: 20x reduction in IPC cost
```

```
How to build a full operating system on a microkernel?
  Remember the idea was to move most features into user-level servers.
    File system, device drivers, network stack, process control, &c
  For embedded systems this can be fairly simple.
  What about services for general-purpose use, e.g. workstations, web servers?
  Really need compatibility for existing applications.
    E.g. the system needs mimic something like UNIX.
  Re-implement UNIX kernel services as lots of user-level services?
  Or: run existing Linux kernel as a process on top of the microkernel.
    An "O/S server".
    Perhaps not elegant, but pragmatic.
    Part of a path to adoption:
      Users might start by just running Linux apps.
      Then gradually exploit possibilites of underlying microkernel.

Which brings us to today's paper:
  "The Performance of micro-Kernel-Based Systems",
  by Hartig et al, 1997

basic picture
  [diagram]
  L4 kernel
  Linux kernel server
  one L4 task per Linux process
  IPC for system calls

What does it mean to run a Linux kernel at user-level?
  The Linux kernel is just a program!
  The authors modified Linux in a number of ways,
    replacing hardware access with L4 system calls or IPC.
  Process creation, configuring user page tables, memory allocation,
    system call handling, interrupt handling.

L4/Linux's use of threads
  Each Linux process has one or more L4 threads for its user code
  Linux server has just one L4 thread (plus L4 threads waiting for interrupts)
    At rest it is waiting for IPCs with system calls
  Linux server switches its own L4 thread among kernel threads for its processes
    When e.g. file system code sleep()s waiting for disk read
    Or pipe read() sleep()s waiting for someone to write the pipe
    Much as xv6 switches among kernel threads.
  But an L4/Linux kernel thread switch has
    no relation to user process switching
  Instead, L4 separately switches among runnable L4 threads that
    implement the Linux processes
  So Linux kernel server can be running a kernel thread for process P1,
    while L4 is running process P2 on another core

Why not use L4 threads to implement Linux server's kernel threads?
  Because that would cause pain without any benefit.
  Would introduce parallelism inside Linux.
    But Linux 2.0 did not have SMP support -- e.g. no spinlocks.
  And their hardware had only one core, so could be no parallel speedup anyway.

Drawback: L4 is in charge of scheduling user threads
  So L4/Linux couldn't enforce Linux's notions of priority &c

L4/Linux server maps all user memory into its address space
  (really, it allocates lots of memory, then gives its own memory to user processes)
  uses this for copyin()/copyout(), to dereference user pointers from sys calls
  this keeps system call IPCs small -- data address, not the data itself
  Linux server also uses its memory access for fork() and exec()

Example: how does fork() work?
  process P1 calls fork() (P1 is really an L4 task)
```

```
    P1's libc library turns fork() into an IPC to L4/Linux server
    L4/Linux asks L4 to create a new task and thread -- P2
    L4/Linux allocates memory pages (as many as P1 has)
    L4/Linux uses IPC to tell L4 to map pages into P2
    L4/Linux copies data from P1's pages to P2's pages
    L4/Linux sends special IPC to P2 with SP and PC to cause it to run
    L4/Linux sends reply to P1 via IPC

  L4/Linux server acts as the pager for user processes
    so L4 turns process page faults into IPC to Linux server
    for e.g. copy-on-write fork, lazy allocation, memory mapped files

  Drawback: L4 doesn't allow direct control over page tables
    so Linux server could not switch its page table to include user virt addresses
    until recently Linux used this trick to gain performance (no page table switch),
      and for convenience in dereferencing syscall arguments

  L4/Linux server uses Linux device drivers unchanged!
    since L4 allows it direct access to device registers
    except interrupts arrive via L4 IPC

  How to evaluate?
    What are some questions that the paper might answer?
    It's not really about whether microkernels are a good idea.
    It's main goal is to show they have good performance.

  What kind of performance do we care about?
    Is IPC fast?
      -> microbenchmark
    Is there some other performance obstacle?
      -> whole-system benchmarks

  IPC microbenchmarks
    Table 2
    getpid() is one system call on native Linux
      and two L4 system calls (IPC send, IPC recv) on L4/Linux
    nice result: takes only somewhat more than 2x as long on L4/Linux
      and FAR faster than Mach+LinuxServer

  What do we think the impact of syscalls taking 2x as long might be?
    Disaster?
    Hardly noticeable?

  Whole-system benchmark: AIM
    AIM forks a bunch of processes
    Each randomly uses the disk, allocates memory, uses pipe, computes, &c
      To do a fixed amount of total work
    Figure 8 x-axis shows [some function of] number of concurrent AIM processes
      y-axis shows time for all processes to complete
    Only the slope really matters
      slope is time per unit of work, so lower is better
      Native Linux is best, but L4Linux is only a little slower
      Mach+Linux is noticeably less efficient
    Conclusions:
      2x IPC time doesn't seem to make much overall difference
      L4+Linux is only somewhat slower than Linux
      L4+Linux is significantly faster than Mach+Linux

  These results are not by themselves an argument for using L4
    But they are an argument against rejecting L4 due to performance worries

  What's the current situation?
    Microkernels are sometimes used for embedded computing
      Microcontrollers, Apple "enclave" processor
      Running custom software
```

```
  Microkernels, as such, never caught on for general computing
    No compelling story for why one should switch from Linux &c
  Many ideas from microkernel research have been adopted into modern UNIXes
    Mach spurred adoption of sophisticated virtual memory support
    Virtual machines are partially a response to the O/S server idea
    Loadable kernel modules are a response to need to extensibility
    Client/server e.g. DNS server, window server
    MacOS has microkernel-style IPC

References:
The Fiasco.OC Microkernel -- a current L4 descendent
  https://l4re.org/doc/
fast IPC in L4
  https://cs.nyu.edu/~mwalfish/classes/15fa/ref/liedtke93improving.pdf
later evolution of L4
  https://ts.data61.csiro.au/publications/nicta_full_text/8988.pdf
```