6.824 2006 Lecture 2: I/O Concurrency

Recall timeline
  [draw this time-line]
  Time-lines for CPU, disk, network
  How can we use the system's resources more efficiently?

What we want is *I/O concurrency*
  Ability to overlap I/O wait with other useful work.
  In web server case, I/O wait mostly for net transfer to client.
  Could be disk I/O: compile 1st part of file while fetching 2nd part.
  Could be user interaction: emacs GC while waiting for you to type.

Performance benefits of I/O concurrency can be huge
  Suppose we're waiting for disk for client one, 10 milliseconds
  We can probably server 100 other clients from cache during that time!

Typical ways to get concurrency.
  This is about s/w structure.
  There are any number of potential structures.
  [list these quickly]
  0. (One process)
  1. Multiple processes
  2. One process, many threads
  3. Event-driven
  Depends on O/S facilities and type of application.
    Degree of interaction among different sub-tasks.

One process can be better than you think!
  O/S provides I/O concurrency transparently when it can
  O/S does read-ahead into cache, write-behind from buffer
    works for disk and network connections

I/O Concurrency with multiple processes
  Start a new UNIX process for each client connection / request
  Master processes hands out connections.
  Now plenty of work available to keep system busy
  Still simple:
    look at server_2() in handout.
    fork() after accept()
    Preserves original s/w structure.
  Isolated: bug for one client does not crash the whole server
    Most interaction hidden by O/S. E.g. lock the disk queue.
  If > 1 CPU, CPU concurrency as a side effect

We may also want *CPU concurrency*
  Make use of multiple CPUs on shared memory machine.
  Often I/O concurrency tools can be used to get CPU concurrency.
  Of course O/S designer had to work a lot harder...
  CPU concurrency much less important than I/O concurrency: 2x, not
100x
    In general, very hard to program to get good scaling.
    Usually easier to buy two separate computers, which we *will* talk
about.

Multiple process problems
  Cost of starting a new process (fork()) may be high.

```
    New address space &c. 300 microseconds *min* on my computer.
  Processes are fairly isolated by default
    E.g. they do not share memory
    What if you want a web cache? Must be shared among processes.
    Or even just keep statistics?

Concurrency with threads
  Looks a bit like multiple processes
  But thread_fork() leaves address space alone
  So all threads share memory
  One stack per thread, inside process
  [picture: thread boxes inside process boxes]
  Seems simple -- still preserves single-process structure.
  Potentially easier to have e.g. shared web cache
    But programmer needs to know about some kind of locking.
  Also easier for one thread to corrupt another

There are some low-level but very important details that are hard to
get right.
  What happens when a thread calls read()? Or some other blocking
system call?
  Does the whole process block until disk I/O has finished?
  If you don't get this right, you don't get I/O concurrency.

Kernel-supported threads
  O/S kernel knows about each thread
  It knows a thread was just blocked, e.g. in disk read wait
    Can schedule another thread
  [picture: thread boxes dip down into the kernel]
  What does kernel need for this?
    Per-thread kernel stack.
    Per-thread tables (e.g. saved registers).
  Semantics:
    per-process resources: addr space, file descriptors
    per-thread resources: user stack, kernel stack, kernel state
  Kernel can schedule one thread per CPU
  This sounds like just what we want for our server
  BUT kernel threads are usually expensive, just like processes
    Kernel has to help create each thread
    Kernel has to help with each context switch?
      So it knows which thread took a fault...
    lock/unlock must go through kernel, but bad for them to be slow
  Many O/S do not provide kernel-supported threads, not portable

User-level threads
  Implemented purely inside program, kernel does not know
  User scheduler for threads inside the program
    In addition to kernel process scheduler
  [picture]
  User-level scheduler must:
    Know when a thread is making a blocking system call.
    Don't actually block, but switch to another thread.
    Know when I/O has completed so it can wake up original thread.
  Answer:
    thread library has fake read(), write(), accept(), &c system calls
    library knows how to *start* syscall operations without waiting
    library marks threads as waiting, switches to a runnable thread
```

```
    kernel notifies library of I/O completion and other events
      library marks waiting thread runnable
  read(){
    tell kernel to start read;
    mark thread as waiting for read;
    sched();
  }
  sched(){
    ask kernel for I/O completion events
      mark threads runnable
    find a runnable thread;
    restore registers and return;
  }
  Events we would like from kernel:
    new network connection
    data arrived on socket
    disk read completed
    client/socket ready to receive new data
  Like a miniature O/S inside the process

Problem: user-level threads need significant kernel support
  1. non-blocking system calls
  2. uniform event delivery mechanism

Typical O/S provides only partial support for event notification
   yes: new TCP connections, arriving TCP/pipe/tty data
   no: file-system operation completion

Similarly, not all system calls operations can be started w/o waiting
   yes: connect(), socket read(), write()
   no: open(), stat()
   maybe: disk read()

Why are non-blocking system calls hard in general?
  Typical system call implementation, inside the kernel:
  [sys_read.c]
  Can we just return to user program instead of wait_for_disk?
    No: how will kernel know where to continue?
    ie. should it run userspace code or continue in the kernel syscall?
  Big problem: keeping state for multi-step operations.

Options:
  Live with only partial support for user-level threads
  New operating system with totally different syscall interface.
    One system call per non-blocking sub-operation.
    So kernel doesn't need to keep state across multiple steps.
    e.g. lookup_one_path_component()
  Microkernel: no system calls, only messages to servers.
    and non-blocking communication
  Helper processes that block for you (Flash paper next week)

Threads are hard to program
  The point is to share data structures in one address space
  Thread *model* involves CPU concurrency even on a single CPU
    so programmer may need to use locks
    even if only goal was to overlap I/O wait
  But *events* usually occur one at a time
```

```
    could do CPU processing sequentially, overlap only the I/O waiting

Event-driven programming
  Suggested by user threads implementation
  Organize the s/w around arrival of events
  Write s/w in state-machine style
    When this event occurs, execute this function
  Library support to register interest in events
  The point: this preserves the serial natures of the events
    Programmer sees events/functions occuring one at a time
```