Background: I/O Concurrency

Brad Karp

UCL Computer Science

CS GZ03 / M030

7th October, 2009

Outline

- "Worse Is Better" and Distributed Systems

- Problem: Naïve single-process server leaves system resources idle; I/O blocks
  - Goal: I/O concurrency
  - Goal: CPU concurrency

- Solutions
  - Multiple processes
  - One process, many threads
 - Event-driven I/O (not in today's lecture)

Review: How Do Servers Use Syscalls?

- Consider server_1() web server (in handout)

| R | | W | | W | C | time |

network syscalls

| R | | R | | time |

disk syscalls

| | | time |

application CPU
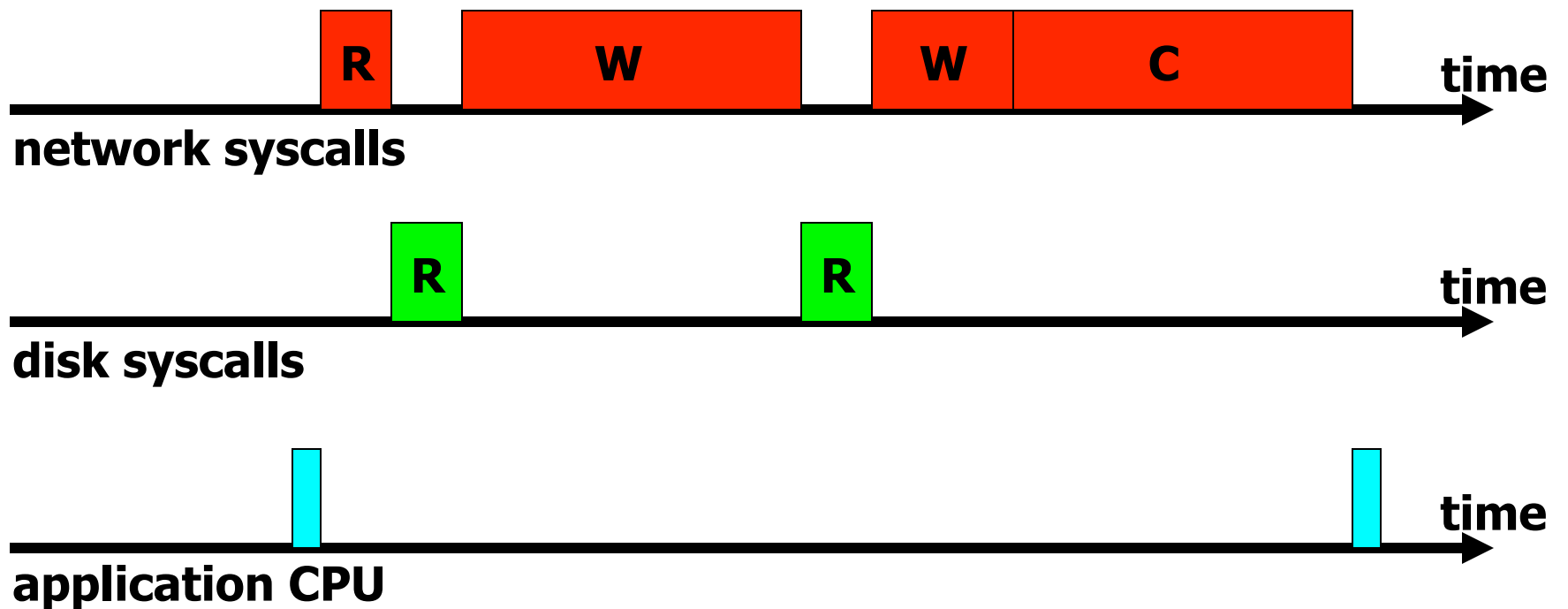
Review: How Do Servers Use Syscalls?

Server waits for each resource in turn
Each resource largely idle
What if there are many clients?

network syscalls

disk syscalls

application CPU
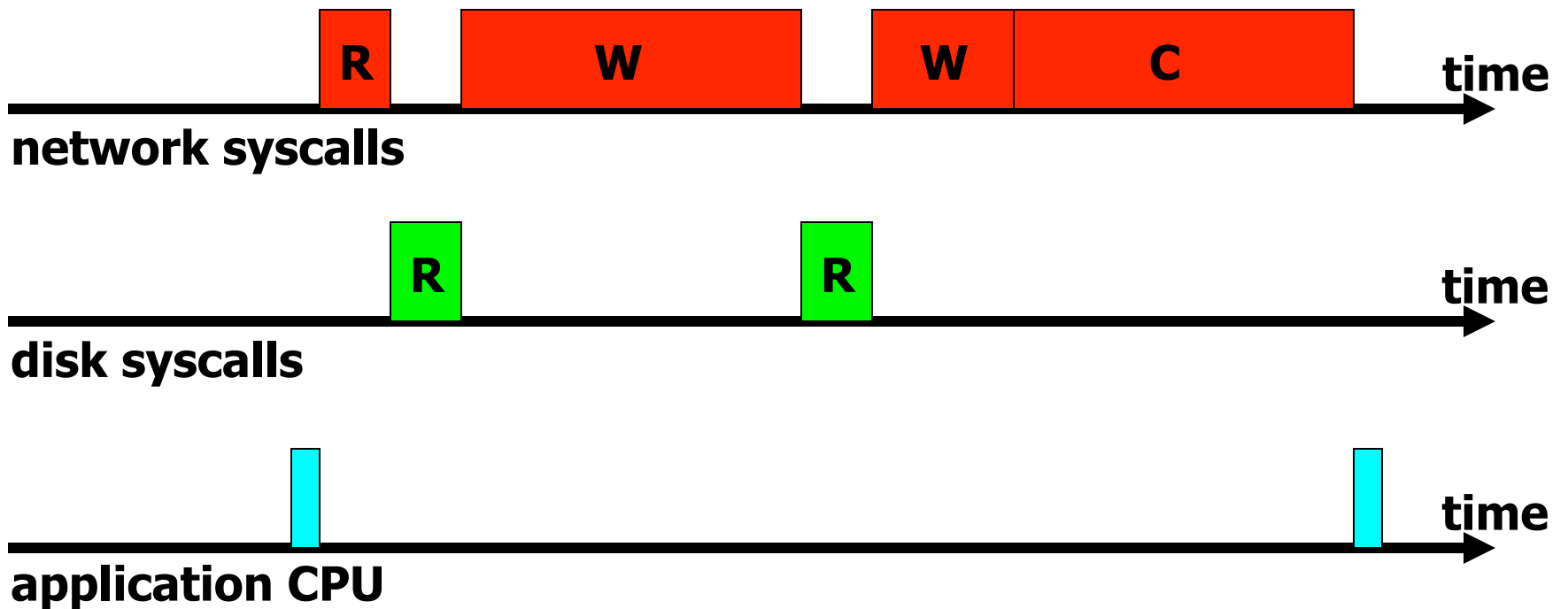
Performance and Concurrency

- Under heavy load, server_1():
  - Leaves resources idle
  - …and has a lot of work to do!
- Why?
  - Software poorly structured!
  - What would a better structure look like?

Solution: I/O Concurrency

- Can we overlap I/O with other useful work? Yes:
  - Web server: if files in disk cache, I/O wait spent mostly blocked on write to network
  - Networked file system client: could compile first part of file while fetching second part
- Performance benefits potentially huge
  - Say one client causes disk I/O, 10 ms
  - If other clients' requests in cache, could serve 100 other clients during that time!

One Process
May Be Better Than You Think

- OS provides I/O concurrency to application transparently when it can, e.g.,

  – Filesystem does read-ahead into disk buffer cache; write-behind from disk buffer cache

  – Networking code copies arriving packets into application's kernel socket buffer; copies app's data into kernel socket buffer on write()

I/O Concurrency with Multiple Processes

- Idea: start new UNIX process for each client connection/request
- Master process assigns new connections to child processes
- Now plenty of work to keep system busy!
  - One process blocks in syscall, others can process arriving requests
- Structure of software still simple
  - See server_2() in webserver.c
  - fork() after accept()
  - Otherwise, software structure unchanged!

Multiple Processes: More Benefits

- Isolation
  - Bug while processing one client's request leaves other clients/requests unaffected
  - Processes do interact, but OS arbitrates (e.g., "lock the disk request queue")
- CPU concurrency for "free"
  - If more than one CPU in box, each process may run on one CPU

CPU Concurrency

- Single machine may have multiple CPUs, one shared memory
  - Symmetric Multiprocessor (SMP) PCs
  - Intel Core Duo
- I/O concurrency tools often help with CPU concurrency
  - But way more work for OS designer!
- Generally, CPU concurrency way less important than I/O concurrency
  - Factor of 2X, not 100X
  - Very hard to program to get good scaling
 - Easier to buy 2 machines (see future lectures!)

Problems with Multiple Processes

- fork() may be expensive
  - Memory for new address space
  - 300 us minimum on modern PC running UNIX
- Processes fairly isolated by default
  - Memory not shared
  - How do you build web cache on server visible to all processes?
  - How do you simply keep statistics?

Concurrency with Threads

- Similar to multiple processes
- Difference: one address space
  - All threads share same process' memory
  - One stack per thread, inside process
- Seems simple: single-process structure!
- Programmer needs to use locks
- One thread can corrupt another (i.e., no cross-request isolation)
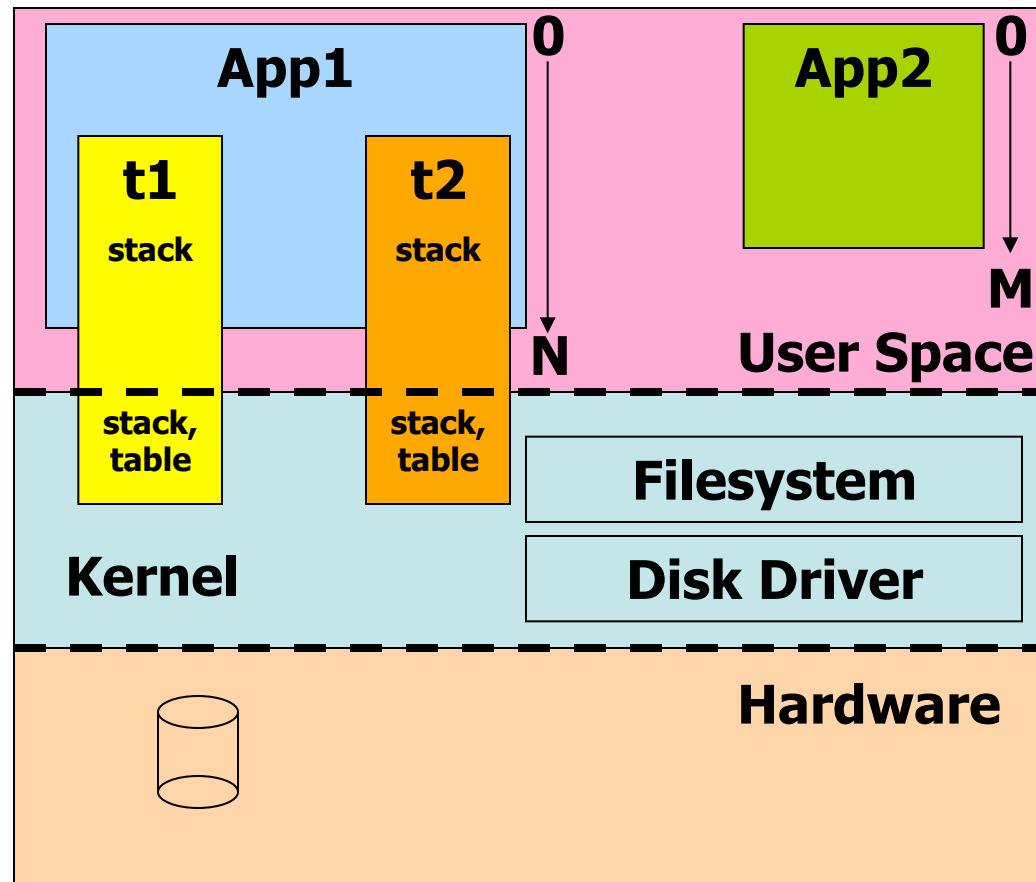
Concurrency with Threads
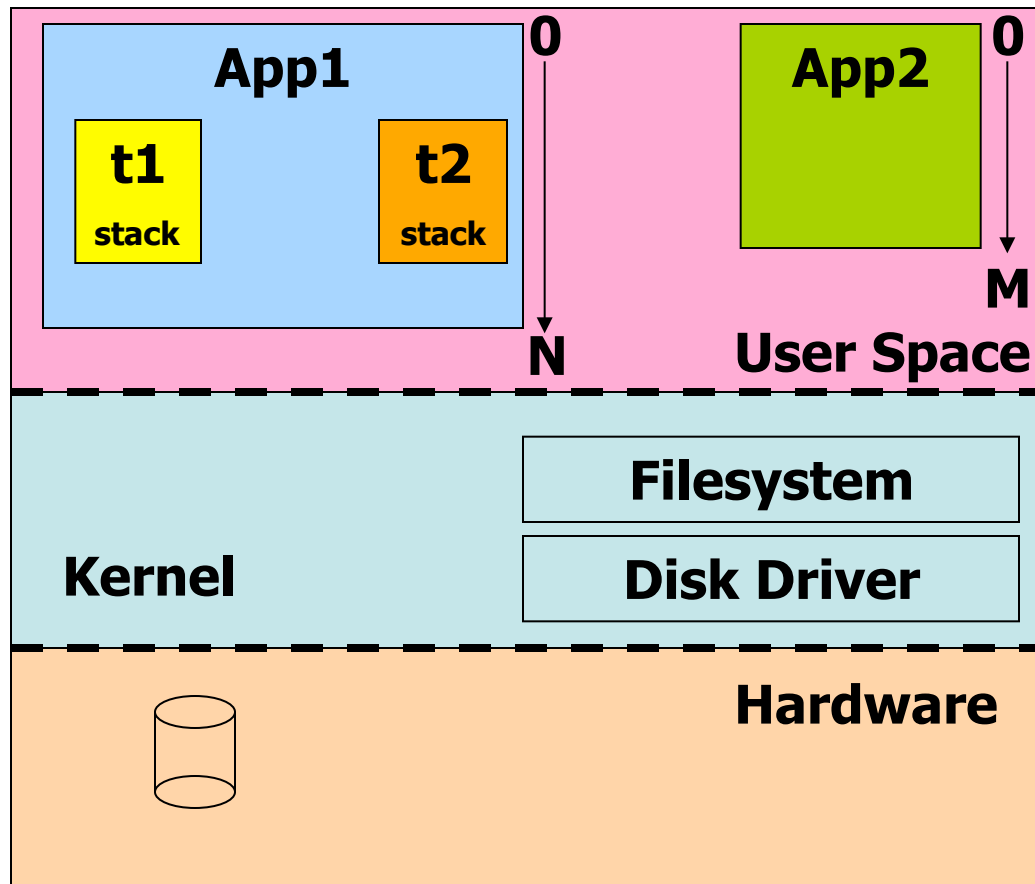
Threads: Low-Level Details Are Hard!

- Suppose thread calls read() (or other blocking syscall)

  – Does whole process block until I/O done?

  – If so, no I/O concurrency!

- Two solutions:

  – Kernel-supported threads

  – User-supported threads

Kernel-Supported Threads

- OS kernel aware of each thread
  - Knows if thread blocks, e.g., disk read wait
  - Can schedule another thread
- Kernel requirements:
  - Per-thread kernel stack
  - Per-thread tables (e.g., saved registers)
- Semantics:
  - Per-process: address space, file descriptors
  - Per-thread: user stack, kernel stack, kernel state

Kernel-Supported Threads

Kernel Threads: Trade-Offs

- Kernel can schedule one thread per CPU
  - Fits our goals well: both CPU and I/O concurrency
- But kernel threads expensive, like processes:
  - Kernel must help create each thread
  - Kernel must help with thread context switch!
    - Which thread took a page fault?
  - Lock/unlock must invoke kernel, but heavily used
- Kernel threads not portable; not offered by many OSes

User-Level Threads

- Purely inside user process; kernel oblivious
- Scheduler within user process for process' own threads
  - In addition to kernel's process scheduler
- User-level scheduler must
  - Know when thread makes blocking syscall
  - Not block process; switch to another thread
  - Know when I/O done, to wake up original thread

User-Level Thread Implementation
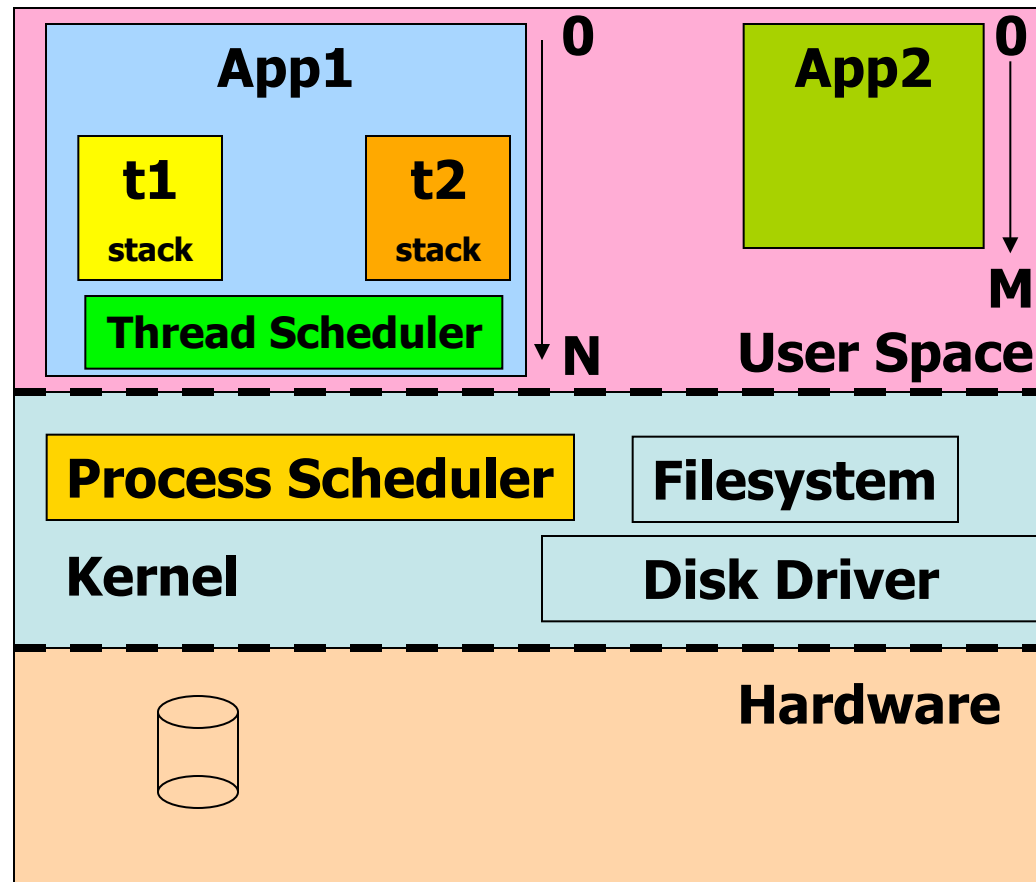
User-Level Threads: Details

- Apps linked against thread library
- Library contains "fake" read(), write(), accept(), &c. syscalls
- Library can start non-blocking syscall operations
- Library marks threads as waiting, switches to runnable thread
- Kernel notifies library of I/O completion and other events; library marks waiting thread runnable

User-Level Threads: read() Example

```
read() {
    tell kernel to start read;
    mark thread waiting for read;
    sched();
}
sched() {
    ask kernel for I/O completion events;
    mark corresponding threads runnable;
    find runnable thread;
    restore registers and return;
}
```

User-Level Threads: Event Notification

- Events thread library needs from kernel:
  - new network connection
  - data arrived on socket
  - disk read completed
  - socket ready for further write()s
- Resembles miniature OS inside process!
- Problem: user-level threads demand significant kernel support:
  - non-blocking system calls
  - uniform event delivery mechanism

Event Notification in Typical OSes

- Usually, event notification only partly supported; e.g., in UNIX:
  - new TCP connections, arriving TCP/pipe/tty data: YES
  - filesystem operation completion: NO
- Similarly, not all syscalls can be started without waiting, e.g., in UNIX:
  - connect(), read() from socket, write(): YES
  - open(), stat(): NO
  - read() from disk: SOMETIMES

Non-blocking System Calls: Hard to Implement

- Typical syscall implementation, inside the kernel, e.g., for read() (sys_read.c):

```
sys_read(fd, user_buffer, n) {
    // read the file's i-node from disk
    struct inode *I = alloc_inode();
    start_disk(…, i);
    wait_for_disk(i);
    // the i-node tells us where the data are; read it.
    struct buf *b = alloc_buf(i->…);
    start_disk(…, b);
    wait_for_disk(b);
    copy_to_user(b, user_buffer);
}
```

Non-blocking System Calls: Hard to Implement

- Typical syscall implementation, inside the kernel,

> **Why not just return to user program instead of calling wait_for_disk()?**
>
> **How will kernel know where to continue?**
>
> **In user space? In kernel?**

```
    wait_for_disk(i);
    // the i-node tells us where the data are; read it.
    struct buf *b = alloc_buf(i->…);
    start_disk(…, b);
    wait_for_disk(b);
    copy_to_user(b, user_buffer);
}
```

Non-blocking System Calls: Hard to Implement

- Typical syscall implementation, inside the kernel,

Why not just return to user program instead of calling wait_for_disk()?

How will kernel know where to continue?

In user space? In kernel?

wait_for_disk(i);

Problem: Keeping state for complex, multi-step operations

start_disk(n, b);

wait_for_disk(b);

copy_to_user(b, user_buffer);

}

User-Threads: Implementation Choices

- Live with only partial support for user-level threads

- New operating system with totally different syscall interface
  - One syscall per non-blocking "sub-operation"
  - Kernel doesn't need to keep state across multiple steps
  - e.g., lookup_one_path_component()

- Microkernel: no system calls, just messages to servers, with non-blocking communication

Threads: Programming Difficulty

- Sharing of data structures in one address space
- Even on single CPU, thread model necessitates CPU concurrency
  - Locks often needed for mutual exclusion on data structures
  - May only have wanted to overlap I/O wait!
- Events usually occur one-at-a-time
  - Can we do CPU sequentially, and overlap only wait for I/O?
  - Yes: event-driven programming

Event-Driven Programming

- Foreshadowed by user-level threads implementation
  - Organize software around event arrival
- Write software in state-machine style
  - "When event X occurs, execute this function."
- Library support for registering interest in events (e.g., data available to read())
- Desirable properties:
  - Serial nature of events preserved
  - Programmer sees only one event/function at a time