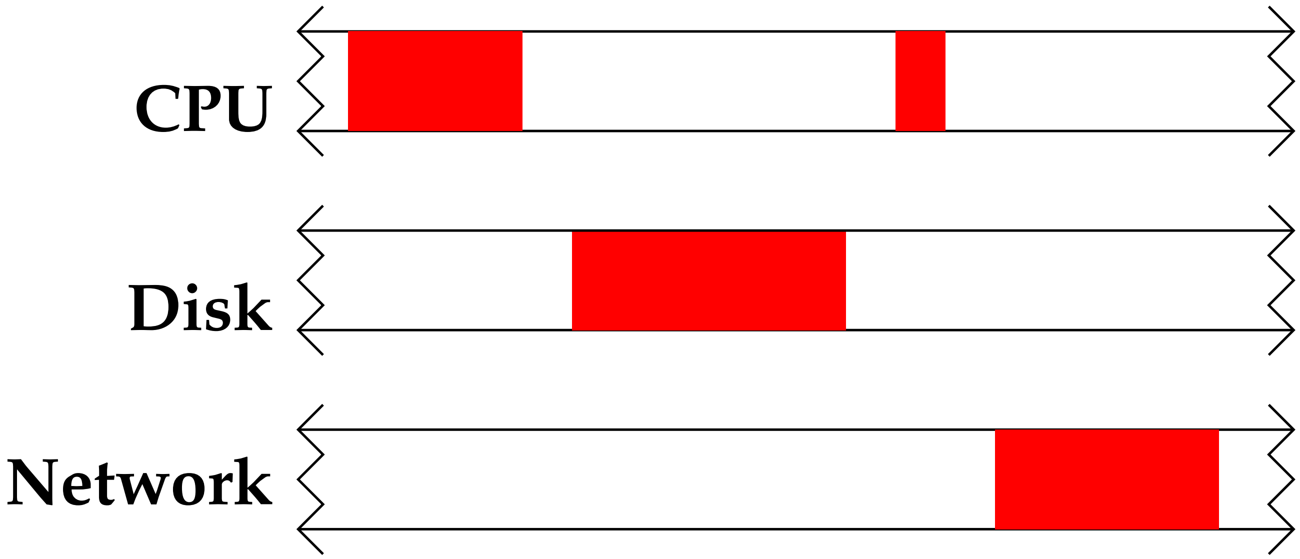
Abstractions

- **We said OS provides protection and abstraction**
- **What kind of abstractions?**
  - Process—address space, thread of control, user ID
  - File System
  - Pipe/Socket—local IPC, network communication
- **How to use interface?**

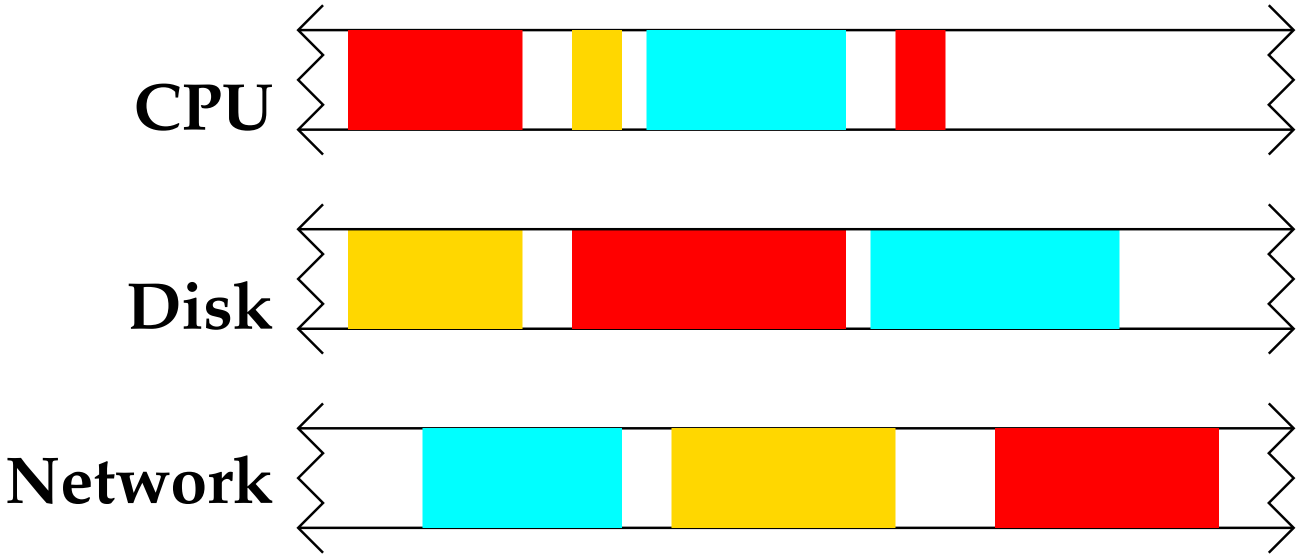
```
for (;;) {
  read_from_network ();
  parse_request ();
  read_from_disk ();
  write_to_network ();
}
```

Synchronous server



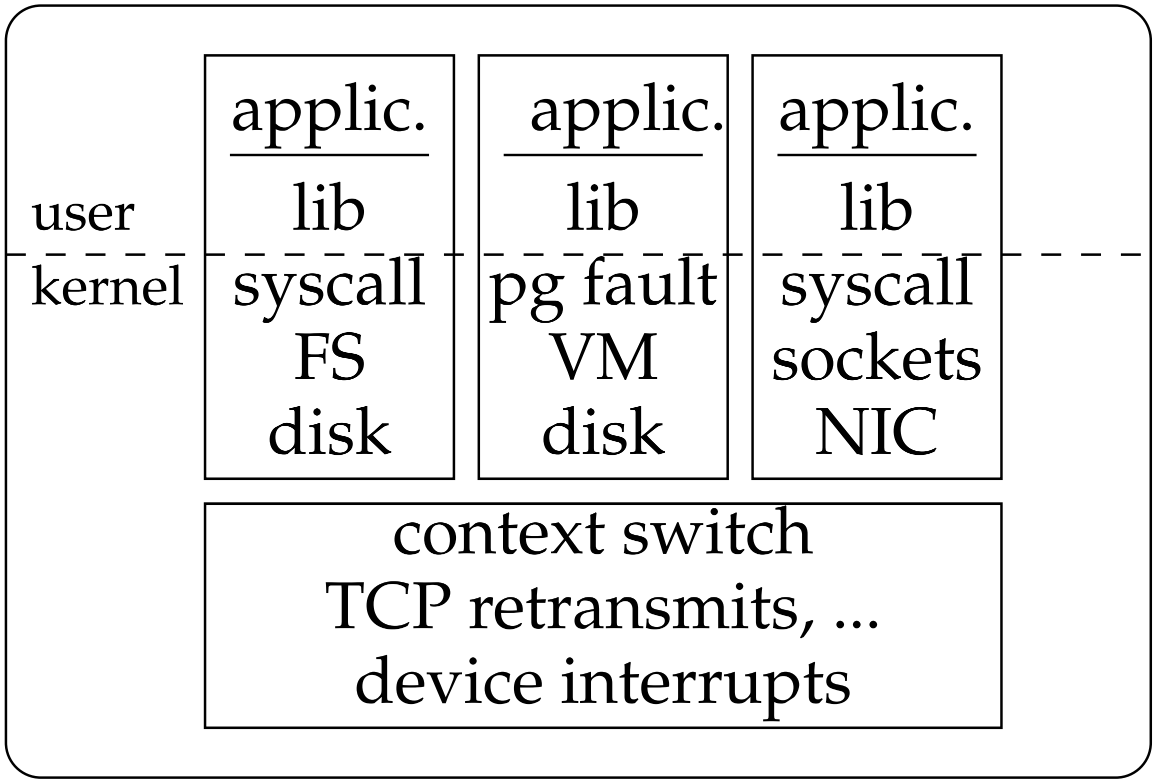
- OS handles the disk/network read/write
- Becomes more useful with multiple processes

Concurrent server



- Overlapping operations makes for higher resource utilization

Traditional OS structure



- **One large piece of software in supervisor mode**
  - Offers convenient, portable high-level programming model
  - Easy for kernel subsystems to cooperate (FS, disk driver, buffer cache all just communicate through procedure calls)

Example: OpenBSD/x86 System call

- **Application:** `read (fd, buf, len);`
- **C Library:**
  - Ensure fd, buf, len on stack
  - Put 0x3 (syscall no. SYS_read) in designated register
  - Execute INT instruction (e.g., `int $0x80`)
- **INT instruction**
  - Sets privileged mode bit
  - Sets SP to kernel stack
  - Saves a few user registers on stack (e.g., user IP, SP)

Finishing the system call

- **Kernel trap handler**
  - Fix up any remaining state (e.g., segmentation regs)
  - Copy arguments in from user stack
  - Transfer control to `sys_read ()` (ordinary C function)
- `sys_mkdir ()`
  - Calls FS → buffer cache
  - Copies data out to application, returns
- **Back in trap handler**
  - `iret` instruction restores regs, returns to user mode

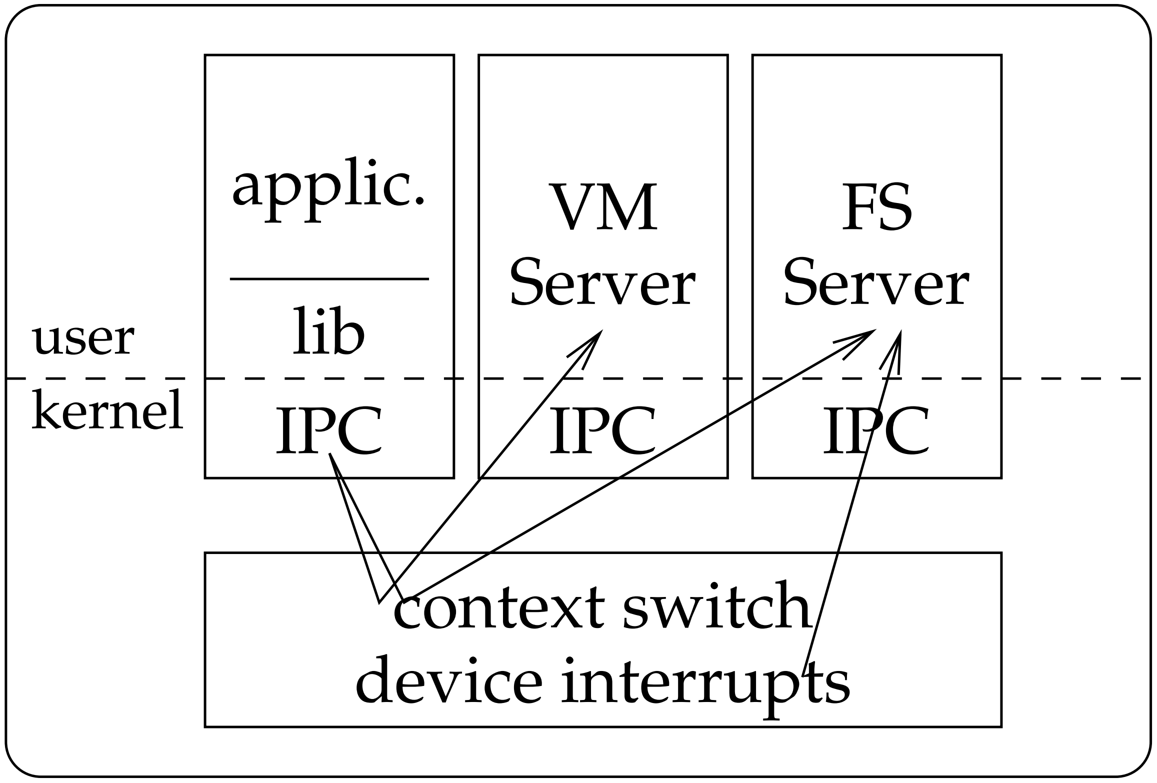
What if read missed in buffer cache?

- `sys_read ()`
  - Invokes FS → Disk driver → sleep → switch
- **switch**
  - Saves all state of current process
  - Finds a process to run (or jumps to idle loop)
  - Switch address spaces, continue process
- **Later, disk interrupt signals I/O completion**
  - Set flag saying want to reschedule mkdir proc. again
  - Preempt current proc—make it call switch
  - Switches address space, our sleep returns

Drawbacks of traditional kernels

- **All privileged, room for many bugs**
  - Developing new OS abstractions painful (crash→reboot)
  - Bad OS code destabilizes whole system
    Hard to convince people to run your OS extension
- **Limits flexibility**
  - Want multiple threads per process?
  - Want single thread crossing into a different address space?
  - Want control disk layout of files for performance?
  - Don't like the kernel's TCP implementation?

Alternative: μkernels



- **Move complex abstractions to server processes**
  - Kernel mostly handles IPC
  - Also grants hardware access to privileged servers

CISC vs. RISC

- **CISC = complex instruction set computing**
  - One instruction may do many things (e.g., strcpy)
  - One instruction can take many cycles
  - Often variable-length instructions, special-purpose registers
    e.g., stack push & pop inst specific to sp register
- **RISC = reduced instruction set computing**
  - Fixed-length instructions, many general purpose registers
  - Hardwired control, exposed pipeline

RISC philosophy

- **Simpler instructions → faster implementation**
  - w/o stalls, retire one instruction every cycle
  - more instructions, but faster so isn't usually a problem?
- **Don't do in hardware what can be left to software**
- **Optimize for the common case**
  - Don't waste silicon for uncommon operations like system calls
  - Use transistors to make ADD fast instead!
- **How to to decide what's common?**
  - Industry standard benchmarks like SPEC

Example: MIPS

- **31 general purpose registers (29 usable)**
  - Fixed-length instructions include 5 bits for each register
  - Any instruction can operate on any register
  - By convention divide into caller & callee saved registers
- **Load/store architecture**
  - ALU operations only on registers (e.g. R1 $\leftarrow$ R2 + R3)
- **Exposed pipeline – delay slots**
  - Branch delay slot – instruction *after* branch always executed (Delay slot instruction must be idempotent–why?)
  - Load delay slot – can’t use register right after loading it

MIPS Traps & Exceptions

- **Most traps/system calls vector to the same location**
  - Software demultiplexes different types of exception
- **Software-managed TLB has optimized fault handler**
  - Hardware vectors directly to fault handler
  - Two registers reserved for use by TLB miss handler
  - Handler + most of kernel run in unmapped memory,
    but page tables mapped, so handler is allowed to fault
- **No atomic read-modify-write memory operations**
  - Requires trap to the kernel

Example: SPARC

- **Wanted even more registers**
  - Since ALU operations only work on registers
  - Plus loads/stores are slow
  - But can't fit more bits for register number in instructions
- **New mechanism: Register windows**
  - Divide registers into 7 global, 8 in, 8 out, 8 local
  - On procedure call, rotate windows (in $\leftarrow$ out, new in+local)
  - Trap on window overflow/underflow (kernel saves values to stack)
- **Also conditional branch delay slots**
  - Slot instruction executed only if branch taken

CISC revisited

- **Do we care about RISC now?**
  - Still in use for niche markets (Mac, 64-bit apps., . . .)
  - But most machines are now x86/Pentium
- **Still, Pentium shares many properties with RISC**
  - Since PentiumPro, internally translates code to RISC-like instructions
  - Deep pipelines (Pentium IV particularly)
- **Legacy instruction sets make some operations even more expensive than on RISCs**
  - Traps to kernel (`int`) many cycles (manipulates the stack)
  - Trap handler code expensive (e.g., must load segment registers)
  - Context switch requires TLB flush