Week 4:

IPC Mechanisms

Interrupts, signals, sockets, events

CSC 469/ CSC 2208

Fall 2018

University of Toronto, Department of Computer Science

Topics

- Interrupts
  - What is an interrupt?
  - How do operating systems handle interrupts?
    - FreeBSD example (Linux in tutorial)
- Signals
  - What is a signal?
  - How are they implemented and used?
- Messages through file descriptors
  - Pipes / fifos / sockets, notification of activity
- Generalizing the event notification mechanism

Interrupts

Defn: an event *external* to the currently executing process that causes a change in the normal flow of instruction execution; usually generated by hardware devices external to the CPU

- From *"Design and Implementation of the FreeBSD Operating System", Glossary*

- Key point is that interrupts are *asynchronous* with respect to current process
  - Typically indicate that some device needs service

Why Interrupts?

- People like connecting devices
  - Keyboard, mouse, screen, disk drives
  - Scanner, printer, sound card, camera, etc.
- These devices occasionally need CPU service
  - But we can't predict *when*
- External events typically occur on a macroscopic timescale
  - we want to keep the CPU busy between events

Need a way for CPU to find out when devices need attention

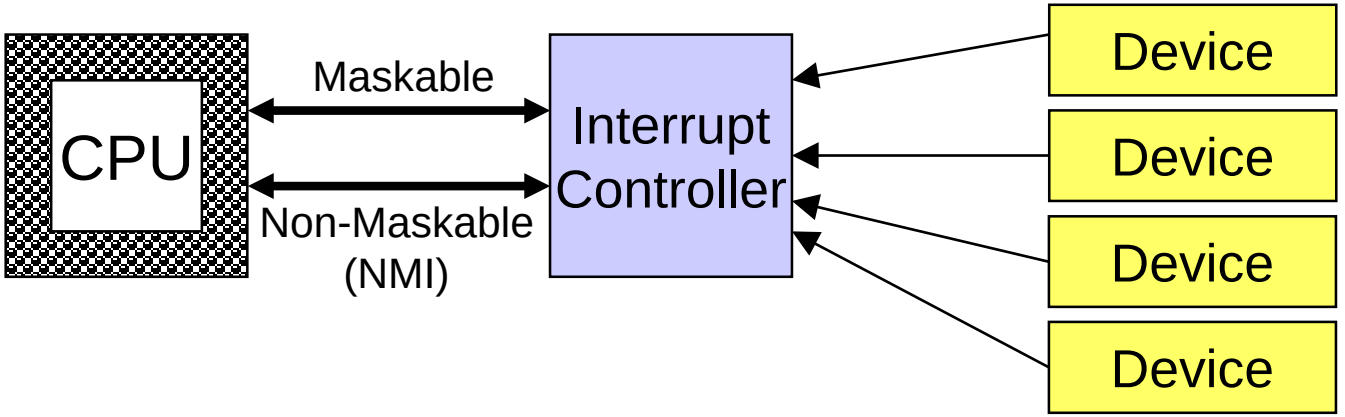
Possible Solution: Polling

- CPU periodically checks each device to see if it needs service
  - ✓ can be efficient if events arrive rapidly
  - ✗ takes CPU time even when no requests pending
  - ✗ overhead may be reduced at expense of response time

"Polling is like picking up your phone every few seconds to see if you have a call. …"

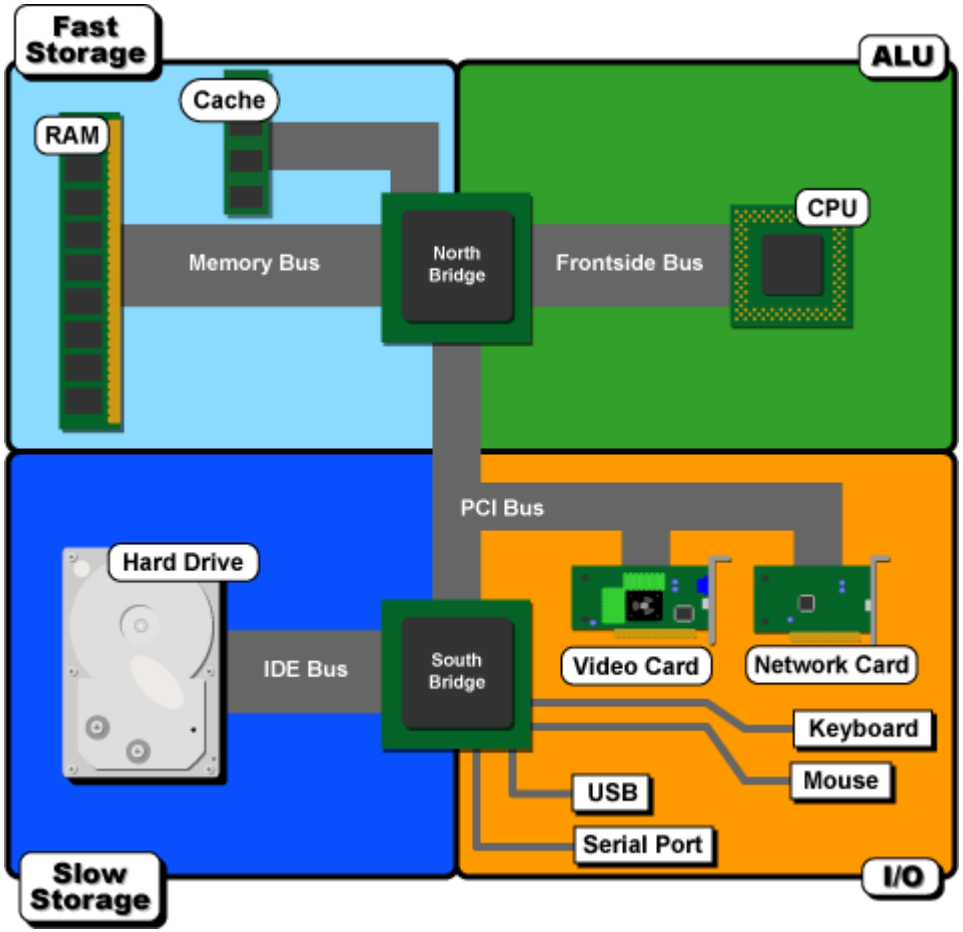
Alternative: Interrupts

- Give each device a wire (interrupt line) that it can use to signal the processor
  - When interrupt signaled, processor executes a routine called an *interrupt handler* to deal with the interrupt
  - No overhead when no requests pending



Intel 430HX Motherboard

- Programmable interrupt controller (PIC) part of the “Southbridge” chip
  - Commonly 8259A chip
  - 8 inputs, 1 output
  - Can be chained together
- Newer systems use “Advanced PIC” (APIC) for SMP support
  - Local APIC + I/O APIC
  - Principle is the same
- APIC controller translates IRQ # to an interrupt number, and signals CPU (e.g., INT 0eh)
- CPU handles interrupt



(image from The Ars Technica Motherboard Guide, Dec. 2005, Jon Hannibal Stokes)

Polling vs. Interrupts

"Polling is like picking up your phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring."

- Interrupts win if processor has other work to do *and* event response time is not critical

- Polling can be better if processor has to respond to an event ASAP
  - On multi-core systems, may use polling for some high-speed devices

Hardware Interrupt Handling

- Details are architecture dependent!
- Interrupt controller signals CPU that interrupt has occurred, passes interrupt number
  - Interrupts are assigned priorities to handle simultaneous interrupts
  - Lower priority interrupts may be disabled during service
- CPU senses (checks) *interrupt request line* after every instruction; if raised, then:
  - uses interrupt number to determine which handler to start
  - *interrupt vector* associates handlers with interrupts
- Basic program state saved (as for system call)
- CPU jumps to interrupt handler
- When interrupt done, program state reloaded and program resumes

Software Interrupt Handling

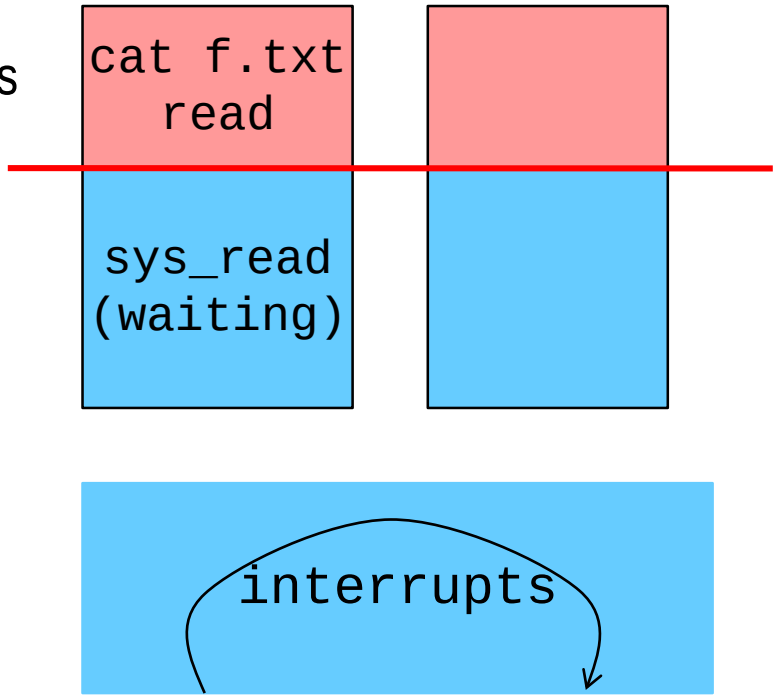
- Typically two parts to interrupt handling
  - The part that has to be done immediately
    - So that device can continue working
  - The part that should be deferred for later
    - So that we can respond to the device faster
    - Originally, so that we have a more convenient execution context
      - Hardware interrupts were processed in a very limited context, no separate stack, could not block

FreeBSD Unix Kernel Structure (5.2+)

User process

Top half of kernel

Bottom half of kernel



Preemptive scheduling.

Runs on user stack in user address space.

Runs until blocked or done.

Can block to await resource.

Runs on per-process kernel stack.

Scheduled by interrupts

Can block to await a resource

Runs on per-interrupt kernel stack.

Interrupt Context

- Modern FreeBSD kernel creates thread context for each device driver
  - Has own kernel execution stack
  - Can block if needed
- Handler needs to be kept fast and simple
  - Interrupt is disabled while running or blocked
  - Typically sets up work for second part of interrupt handling
  - Indicates that second part needs to execute, and re-enables interrupt

Software Interrupts

- The deferred parts of interrupt handling are sometimes referred to as “software interrupts”
  - In Linux, they are referred to as “bottom halves”
  - The terminology here is inconsistent and confusing
- What things can be deferred?
  - Networking
    - Time-critical work → copy packet off hardware, respond to hardware
    - Deferred work → process packet, pass to correct application
  - Timers
    - Time-critical → increment current time-of-day
    - Deferred → recalculate process priorities

Signals

- Software equivalent of hardware interrupts
- Allows process to respond to asynchronous external events (or synchronous internal events)
  - Process may specify its own *signal handlers* or may use OS *default action*
  - Defaults include IGNORE/TERM/CORE/STOP/CONT
    - Ignoring the signal
    - Terminating all threads in the process (with or without a *core dump*)
    - Stopping all threads in the process
    - Resuming all threads in the process
- Provide a simple form of inter-process communication (IPC)

Signal Terminology

- Posting – action taken when event occurs that process needs to be notified of (aka *signal generation*)
- Delivery – action taken when process recognizes arrival of event (aka *signal handling*)
- Catching – if user-level signal handler is invoked, process is said to *catch* the signal
- Pending – signals that have been posted, but not yet delivered

Basics

- Process structure has flags for possible signals and actions to take
- When signal is posted to process, signal pending flag is marked
- When process is next scheduled to run, pending signals are checked, appropriate action is taken
  - Delivery is *not* instantaneous!

Signal types

- `<sys/signal.h>` lists the signal types
- "man signal" gives some description of various signals
  - `SIGTERM, SIGABRT, SIGKILL`
  - `SIGSEGV, SIGBUS`
  - `SIGSTOP, SIGCONT`
  - `SIGCHLD`
  - `SIGPIPE`
  - `SIGUSR1, SIGUSR2`
- http://man7.org/linux/man-pages/man7/signal.7.html
- http://www.manpagez.com/man/3/Signal/

User-Level View

- Write a signal handler function
  - E.g. handle SIGINT (interrupt signal) ourselves

```
void sigint_handler(int sig) {
    close(tmp_file_fd);
    unlink(tmp_file_name);
    exit(EXIT_FAILURE);
}
```

- Install it:

```
struct sigaction new_action, old_action;
new_action.sa_handler = sigint_handler;
sigaction(SIGINT, &new_action, &old_action);
```

Other user-level actions

- Block signal delivery by *masking* signals
  - Similar in spirit to disabling interrupts
  - sigprocmask(int how, sigset_t *newset, sigset_t *oldset)
- Specify that signal handlers run on separate stack
  - sigaltstack(stack_t *signal_stack, stack_t *old_signal_stack)
- Retrieve list of pending signals
  - sigpending(sigset_t *signal_set)
- Block process until signal is posted
  - sigsuspend(setset_t *signal_mask)
- Send signal to process
  - kill(pid_t pid, int signal_number)

Complications

- Handler may execute at any time => unpredictable
  - Need to be careful of manipulating global state in signal handler
- Signal delivery may interrupt execution of signal handler!
  - Code should be *re-entrant*
  - Should block signals if this is not acceptable
- In some implementations (System V Unix, older Linux kernel, libc4,5), handler is reset to default action when it is dispatched
  - Can lead to ugly races… default is often terminate process
- Only one signal handler per signal per process
  - Can't use in library code
- In many implementations, no signal queuing

Kernel View

- Define fixed set of signals, identified numerically
  - E.g. #define SIGKILL 9 /* kill program */
  - Signal sets are bitvectors; each bit position gives the status of corresponding signal
- FreeBSD:
  - Process structure has field to mark pending signals
    - sigset_t p_siglist;
  - Thread structure field to mark pending signals for each thread
    - sigset_t td_siglist;
- Linux:
  - task_struct has field “struct sigpending pending”
    - List of signals and traditional sigset_t field

Signal Posting (FreeBSD)

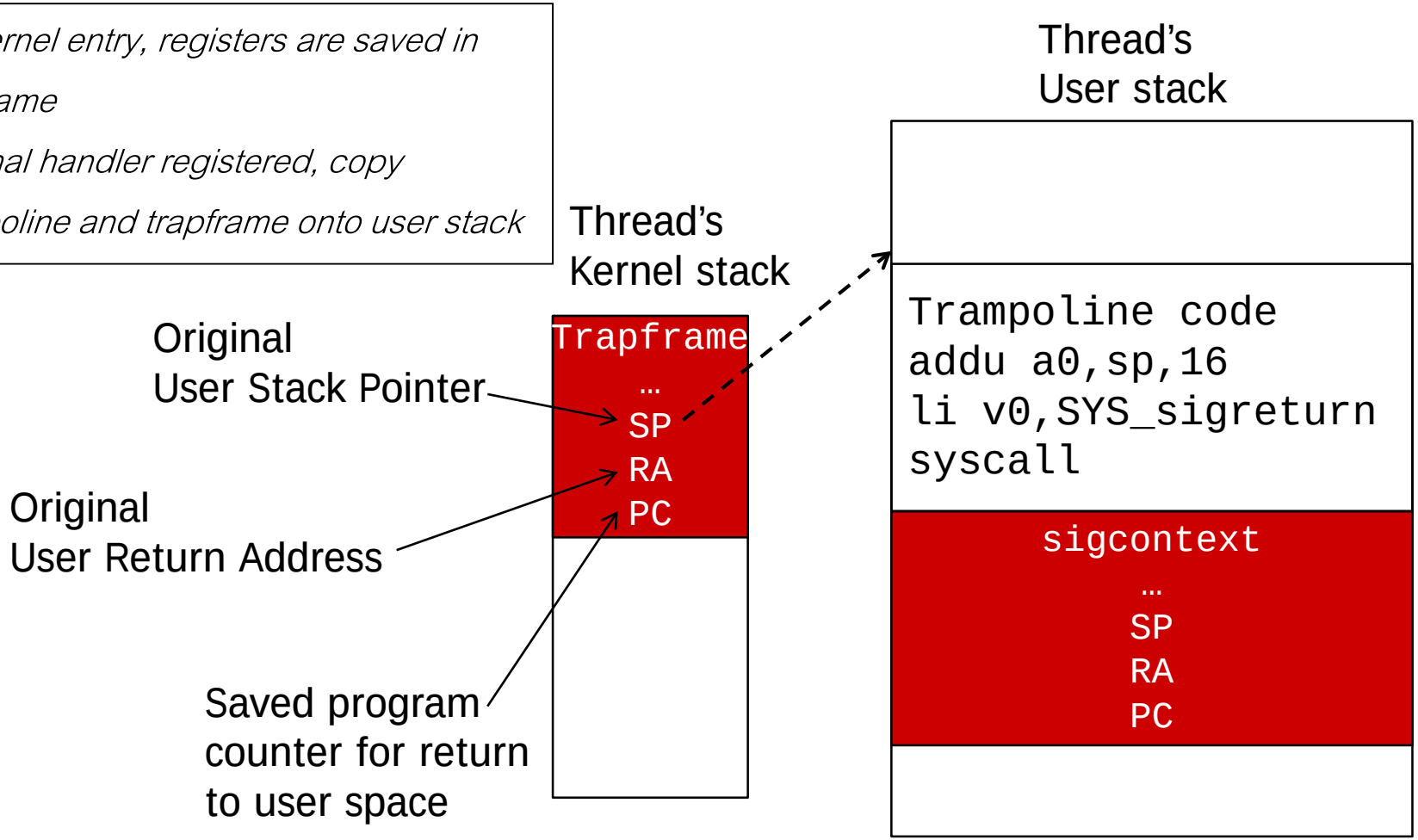
- Mark bit for specified signal in process' p_siglist, and set process to run
  - Process is woken up if in interruptible sleep
  - Many blocking system calls can be interrupted by signals! Should they complete?
- If process is multi-threaded, search for appropriate thread to post signal to
  - Synchronous signals (caused by thread's execution) are posted only to that thread
  - Other signals search thread list for first thread not masking signal and add to that thread's td_siglist
  - If all threads are masking signals, mark process p_siglist
- Some actions can be taken immediately
  - E.g., stopping the process (sometimes), continuing the process

Signal Delivery (FreeBSD)

- Thread checks pending signals (at least once) when in the kernel
  - Typically, just before leaving kernel
- If user-level handler exists, arranges for that handler to be invoked
  - 1. Saves signal state on user stack
  - 2. Sets up registers to begin executing user-mode signal handler *trampoline*
  - 3. Trampoline calls signal handler function
  - 4. When handler returns, trampoline makes sigreturn() system call
  - 5. OS cleans up stack
- Why do we need the trampoline?

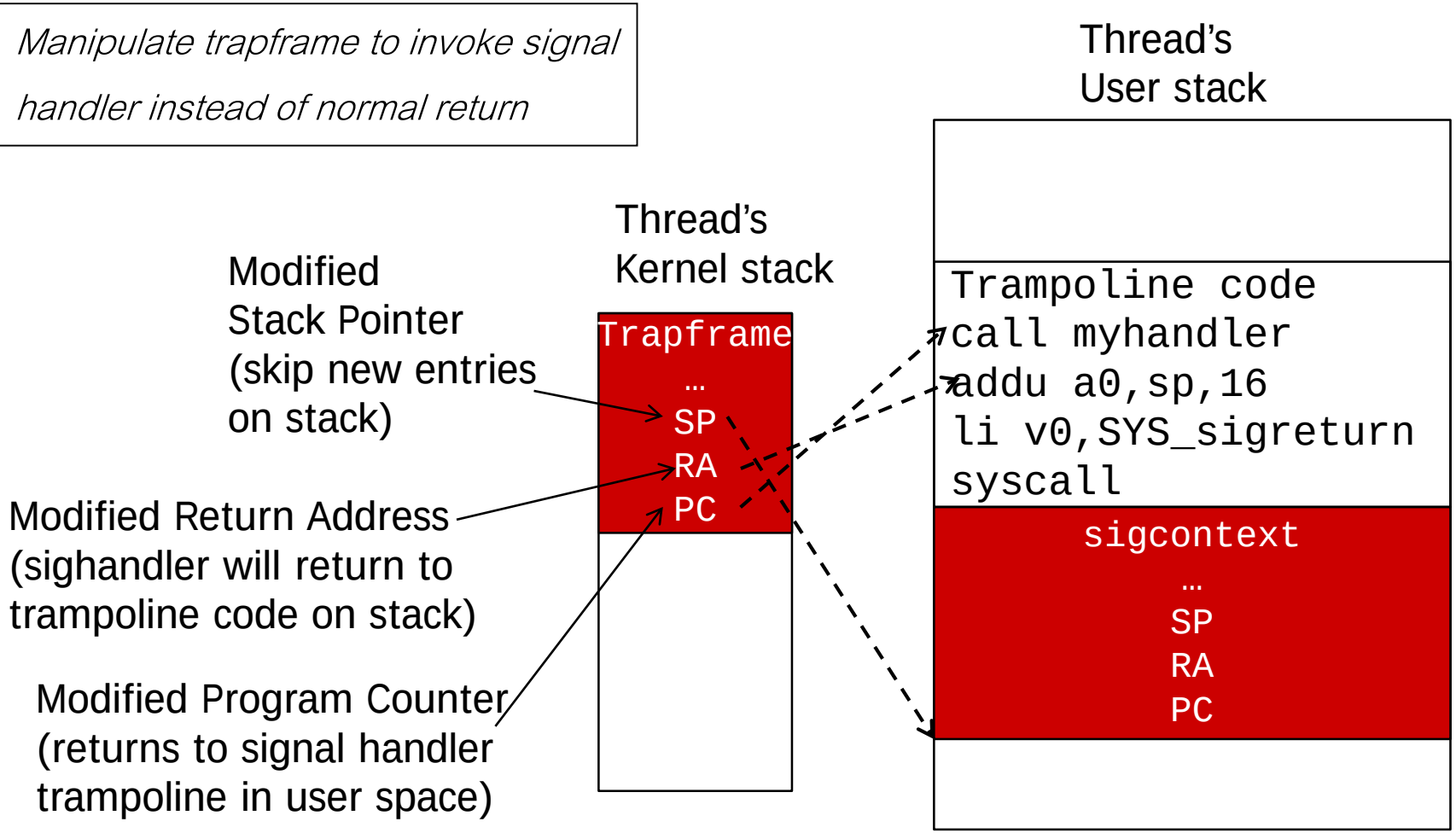
Signal Delivery Illustration

- *On kernel entry, registers are saved in trapframe*
- *If signal handler registered, copy trampoline and trapframe onto user stack*



Signal Delivery Illustration

- *Manipulate trapframe to invoke signal handler instead of normal return*



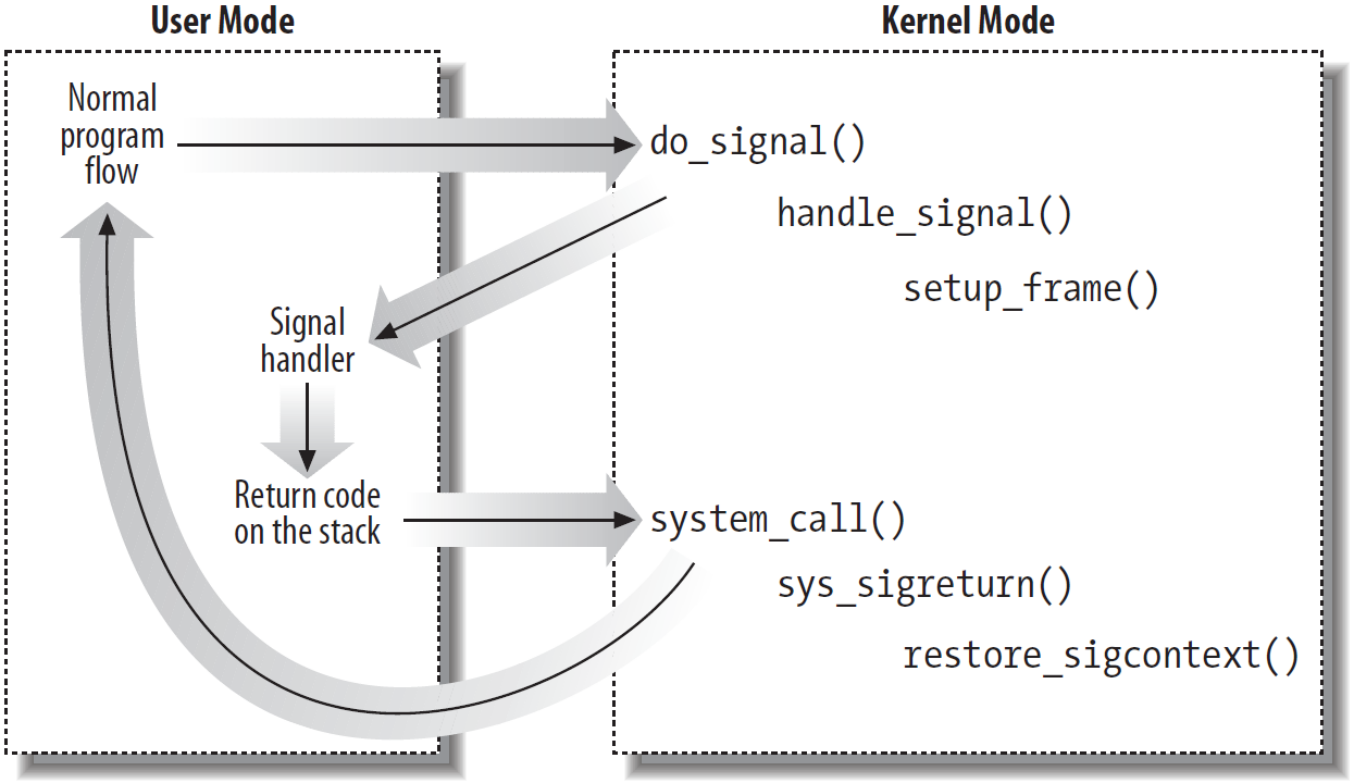
Execution Example

```
volatile int done = 0;

void sigint_handler(int sig){
   done = 1;
}

foo() {
   while(!done) {
        int x = rand();
        x = x % MAXVALUE;
        printf("%d",x);
   }
}
```

1. Another process sends SIGINT
   a. SIGINT flag is marked
2. Timer interrupt fires
   a. State saved in trapframe on kernel stack
   b. Interrupt handler runs
   c. Thread checks signal state before return to userspace
3. Setup to invoke user sighandler
   a. Trampoline and trapframe copied to user stack
   b. Registers in trapframe set to start execution in handler on return from interrupt
4. Handler runs
   a. Return from handler goes to trampoline code
5. Trampoline does sigreturn syscall
   a. Restores original trapframe using sigcontext saved on user stack
6. Return from syscall resumes execution in userspace where original interrupt occurred
7. while condition is now false and loop exits

Linux

- Similarly complex: need to juggle stacks when switching between user and kernel mode



Src: Understanding the Linux Kernel

Real-Time Signals (POSIX.4)

- Ordinary signals carry no information other than signal number
- Real-time signals can include a value as well
- POSIX defines SIGRTMIN and SIGRTMAX for range of real-time signals
  - All ignored by default – have no predefined meaning
- Linux queues real-time signals so they won't be lost or merged
  - Uses the "list" field of the sigpending struct

Using Signals

- Used to implement timers
  - E.g. send SIGALRM after N seconds
- Used in some programming language interpreters to implement language-defined exceptions
  - E.g. JamVM, SableVM (open source Java VMs) implement NULL pointer checks by catching the SIGSEGV that the access causes, and then handling it according to the Java specification
- Simple "X has occurred" communication between processes
  - E.g. parent forks child and wants to know when child has completed initialization before continuing, child sends signal to parent, or parent wants to tell all children to stop after a certain amount of time has elapsed
- Portability can be a concern as different systems have different signal behavior
  - E.g. Linux implements signal queues so multiple signals of the same time can be recorded, but FreeBSD just has the bit marking so repeated signals can be lost

Topics

- Interrupts
  - What is an interrupt?
  - How do operating systems handle interrupts?
    - FreeBSD example (Linux in tutorial)
- Signals
  - What Is a signal?
  - How are they implemented and used?
- Messages through file descriptors
  - Pipes / fifos / sockets, notification of activity
- Generalizing the event notification mechanism

Unix Communication

- Signals are best used for notification of exceptional events
  - Not a general communication strategy
- Unix I/O model is simply a sequence of bytes (aka *byte stream*)
  - I/O streams are referenced with *descriptors*
    - Unsigned integers returned by open() or socket() calls
  - May have multiple types of streams with different characteristics
    - E.g. a file is a linear array of bytes with a name
  - Buffered vs. unbuffered I/O
- Original Unix provided only 1 IPC mechanism – pipe()
  - Pipe – unnamed, unidirectional linear array of bytes
  - Parent creates, fork'd children inherit descriptors

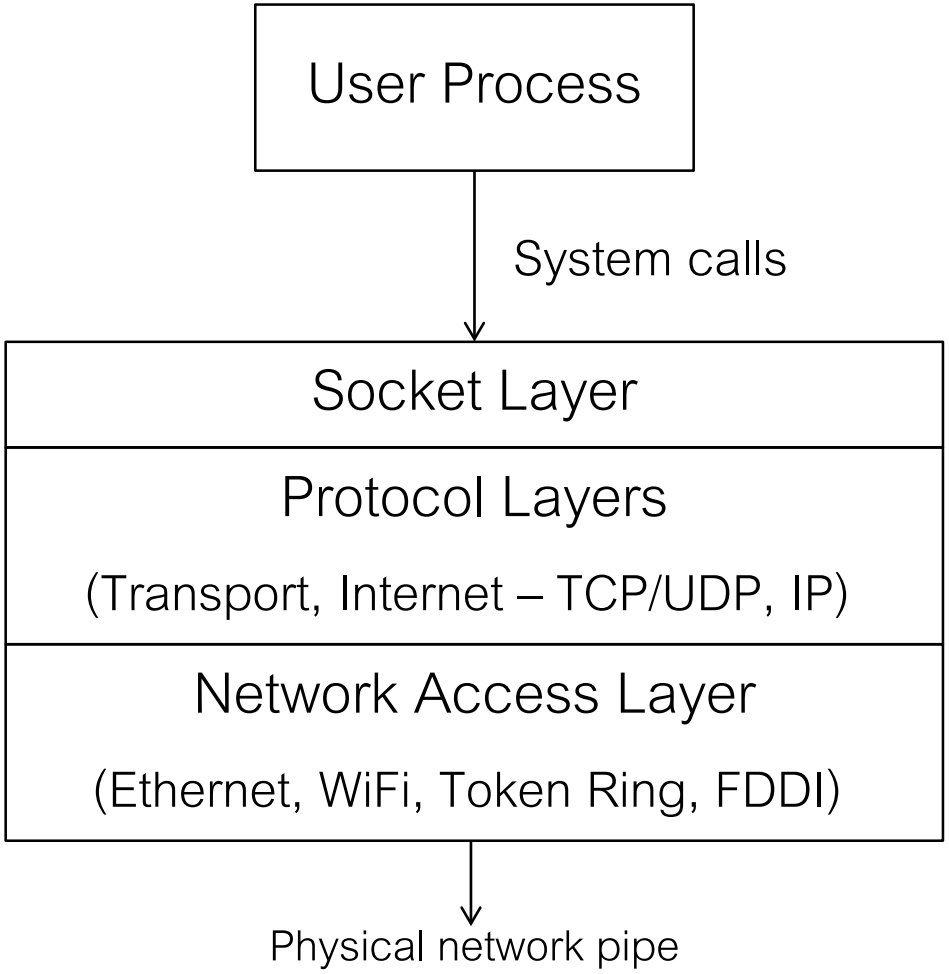
Enhancements to Unix IPC

- Fifo – named pipe
  - ✓ can be opened with open() syscall as needed (not just prior to fork)
  - ✓ Can be used by processes not directly related to each other
  - ✗ Unidirectional
  - ✗ No message boundaries
- Socket - IPC object
  - Introduced with 4.2BSD Unix in 1983
  - Widespread use today

Socket IPC

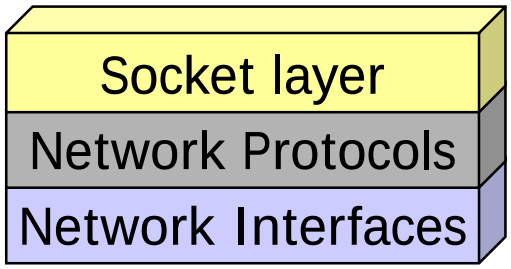
- A socket is a essentially a communication endpoint
- Data is sent from a source socket to a destination socket
- Socket supports various semantics
  - Style of communication, type of protocol, etc.
  - Units of data transmission? Data loss during normal operation? Connection-based?
- Communicating processes each create a socket and connect them together
  - Can now read and write the socket descriptors as for regular file or pipe
  - Can also use special socket communication calls (send, recv, sendto, recvfrom, sendmsg, recvmsg)
- Can be used for local (same machine) or remote communication
  - Characteristics set by choosing *domain* and *type* for socket

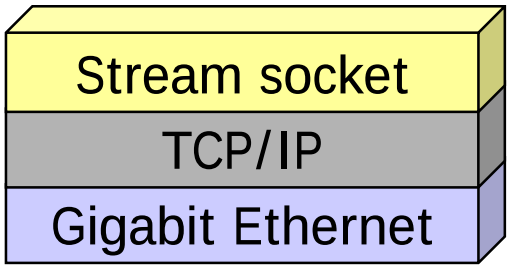
Network Organization



Implementation (BSD)

- So, .. Socket mechanism layered on top of networking



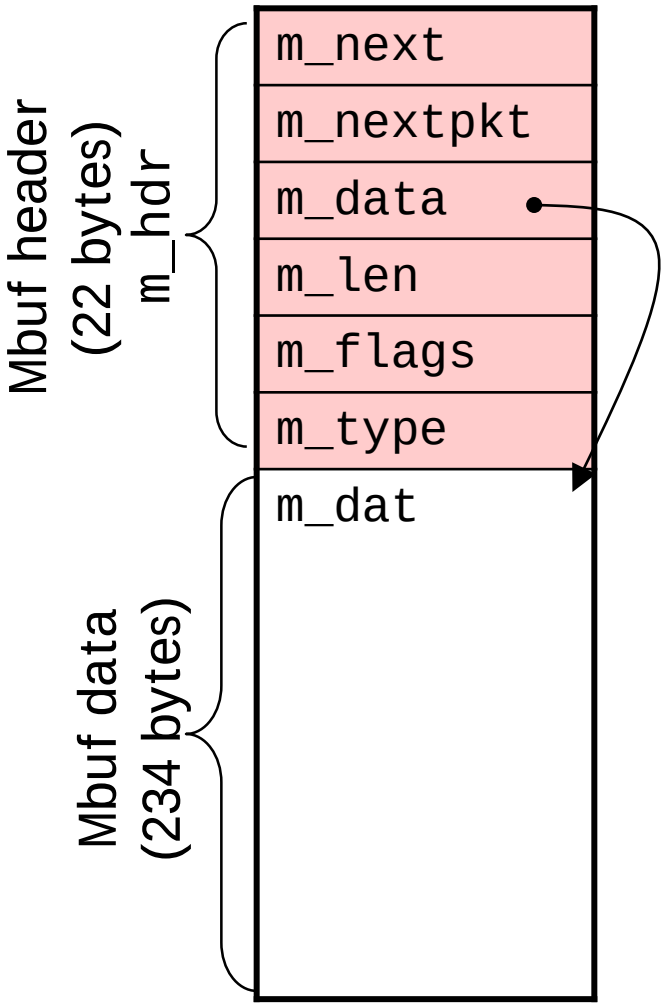


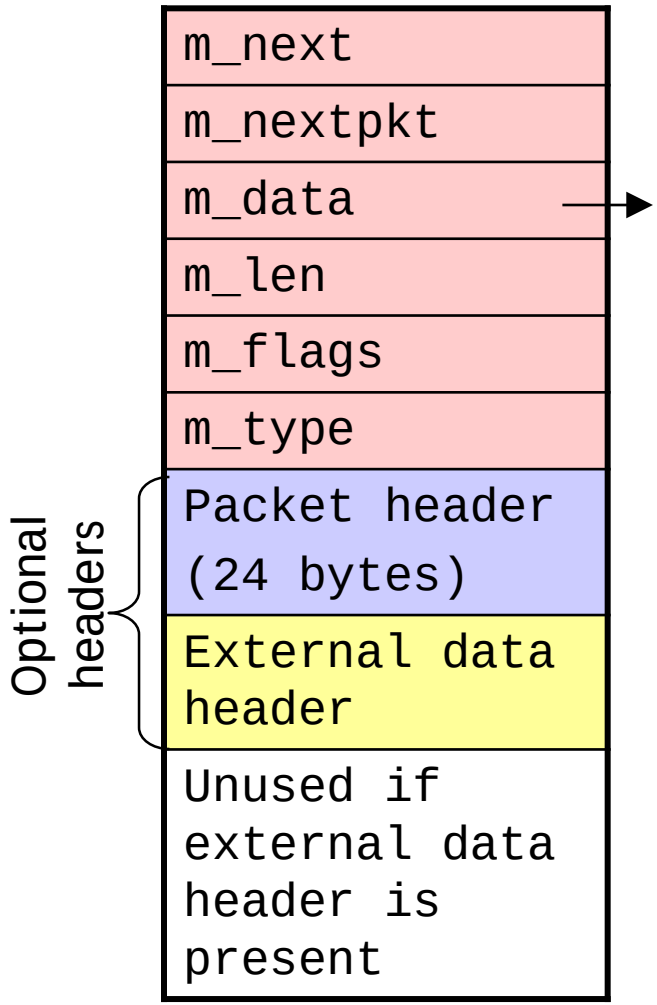
- Key issues are
  - 1. Memory management
  - 2. Connection setup
  - 3. Connection teardown
  - 4. Data transfer

1. Socket Memory Management

- Memory Management Requirements:
  - Allocating and reclaiming memory fast and efficiently
  - Handling small and large packets and copying them efficiently
  - Adding and removing headers efficiently
  - Dividing data stream into packets for transmission, and combining received packets
    - Packet sequence (streams), packet boundaries (datagrams)
  - Moving data between different queues
- Main data structure (BSD): *mbuf*, used by sockets and network layer
- Mbufs allocation: pool of fixed-size blocks of memory (256 bytes each)
  - Makes allocation/reclamation fast
    - no external fragmentation, can allocate any block
    - no coalescing is needed when freeing
  - Multiple blocks can be chained together for larger messages

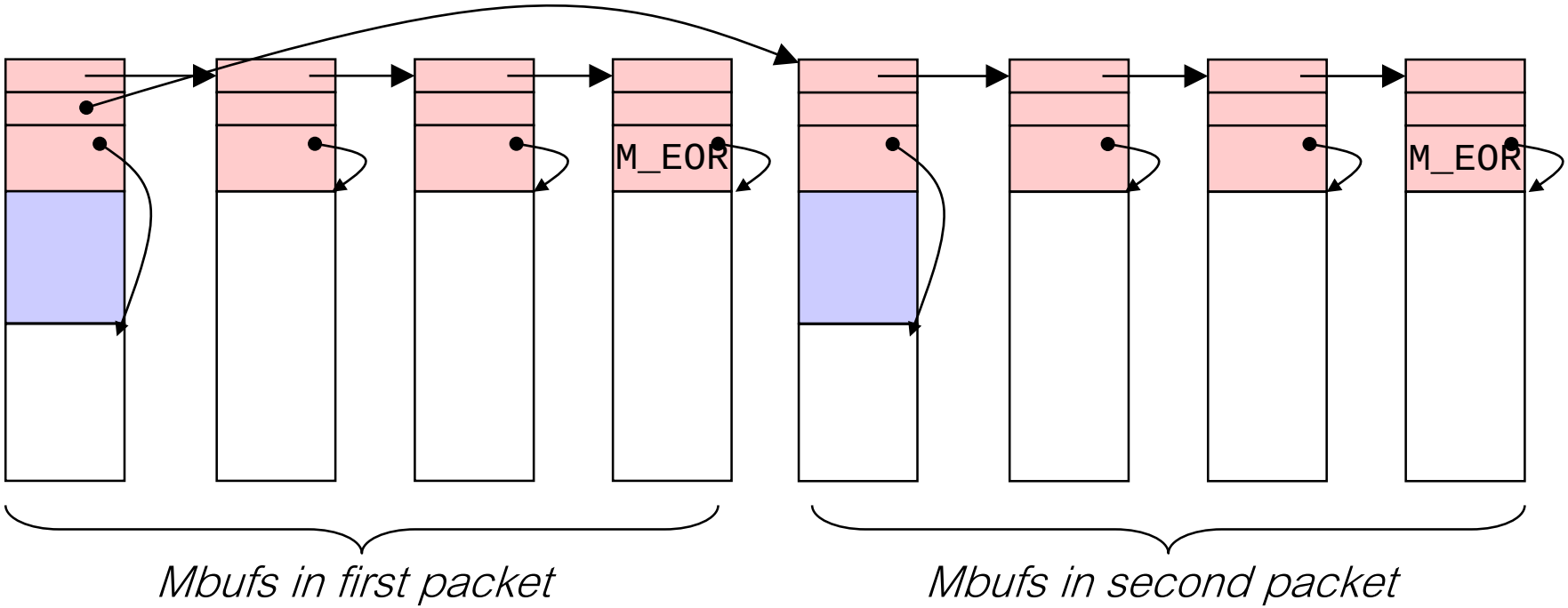
Mbuf structure





- Pointer to data and length of data in header make adding/removing headers easy
- m_next field chains mbufs together to hold arbitrary amount of data (last in chain has M_EOR in m_flags)
- m_nextpkt field links chains of mbufs into objects
- Data may also be external to mbuf structure
  - Why?

Multi-packet Message example



- Entire message can be moved between queues by adjusting pointer to first mbuf
- Socket data structure has two queues of mbufs (send & recv)

Linux equivalent

- Struct `sk_buff` – contains all the packet information
- `sk_buff` elements organized as a doubly-linked list
  - Can move `sk_buff-s` efficiently between lists
- Queue: `struct sk_buff_head` head and tail pointer to `sk_buff` elements.
- Struct `sock` contains protocol-specific info about a socket
  - Includes a receive and send queue
  - Kernel-level internal representation of sockets
- Struct `socket` is the user-level abstraction for sockets

2. Connection Setup

- Client/server model (both are user-level applications)
  - Server listen()s on socket for incoming connection requests
    - Tells protocol layer socket will accept connections
    - Initializes socket data structure with lists for incoming connections
  - Client connect()s its socket to the server socket
    - Asks protocol layer to initiate connection
    - On server side, protocol layer tells socket layer about incoming connection request
    - Connection is placed on destination socket's queue of unaccepted requests
  - Server accept()s connections
    - New descriptor is allocated and returned to server

3. Connection Teardown

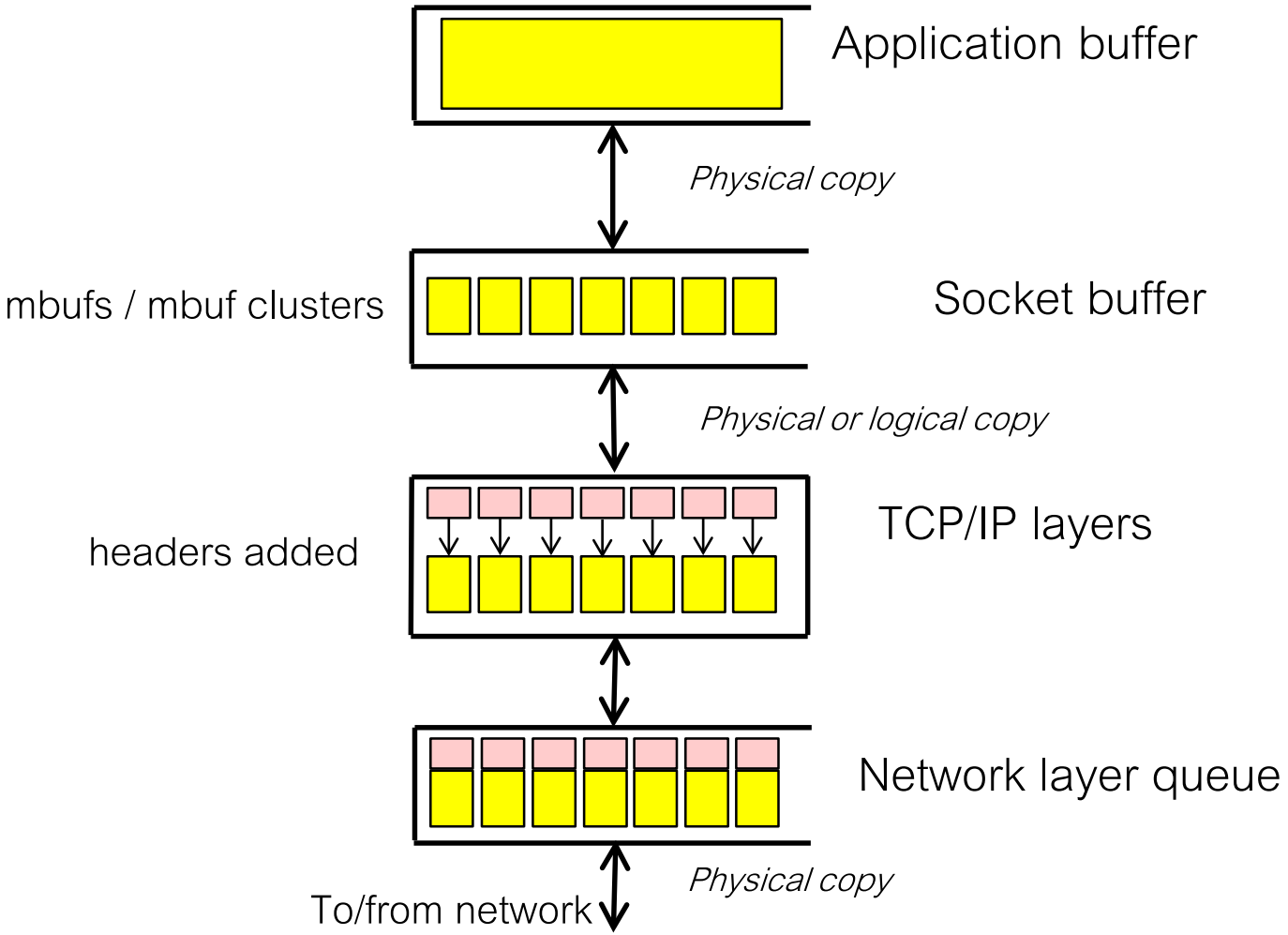
- If reliable data delivery, socket has to (try to) transmit queued data before closing
- Why is it important?
  - Open connections take up client and server resources
  - A server can only support so many connections
  - Attackers could leverage this otherwise (modern security measures do exist though)
- Two ways to teardown
  - Elegant handshake: Use FIN bit in TCP flags

  - Rough: Use Reset bit in TCP Flags

4. Data Transfer

- Mostly copying data from sender address space into mbufs, and from mbufs to receiver address space
  - Network protocol layer handles transfer
- For local communication, optimizations are possible
  - Unmap page(s) containing data from sender address space, re-map them into receiver address space
- Can do part of this for remote communication (map from user into kernel address space, instead of copying)

Data transfer



Using Descriptor-based IPC

- read() and write() are blocking operations
- Often useful to know if you will need to wait before starting
  - Might have other useful work to do in between
  - Might have a large number of active descriptors
    - Don't want to block reading one that has no data while others are ready to go
- Need notification of activity on descriptor
  - Traditional Unix provides poll() and select()

Poll / select

- Give kernel a list of file descriptors of interest, and events on those descriptors
- Need support from the device driver's poll() method:
  - poll_wait() on wait queues, return bitmask => ops doable without blocking
- Poll and select have equivalent function, but different interfaces
  - Poll passes an array of *pollfd* structures, each identifying one file descriptor
  - Select passes 3 bitmasks, one each for read, write and exceptions
    - Setting bit 3 in *readfds* means we want to know if fd 3 is readable
  - Select passes less data but has fixed maximum number of descriptors
- Kernel fills in data and passes back to user level

Problems

- Ok if set of descriptors is small, but inefficient for large sets
- Application must pass an entire list of descriptors, for every call
- Must perform 2 mem copies across user/kernel – ~95% unnecessary
- Up to 3 scans over the set are required
  1) Kernel walks the list to record select info (events that should be tracked)
  2) On wakeup, kernel scans all descriptors again to mark them. If no activity, process may be put to sleep until at least 1 descriptor has some
  3) Application level scans info returned by kernel to see which were marked

  => Duplicated work!
- Really specific to descriptors, no AIO, signals, file or process events

FreeBSD Solution: Kqueue

- See “Kqueue: A generic and scalable event notification facility”, Jonathan Lemon, Usenix Technical 2001
- Goals:
  - Efficient and scalable to large number of descriptors (1000s)
  - Flexible
  - Simple interface
  - Expand information conveyed
  - Reliable
  - Design decision: “level-triggered” not “edge triggered”
  - Correctness

Kqueue API

- Two system calls:
  - kqueue() creates a new event queue where application can register events of interest and retrieve events
    - Returns descriptor for new queue

```
int kqueue(void)
```

  - kevent() used to register new events and retrieve existing ones
    - Returns number of entries placed in *eventlist*

```
int kevent(int kq,
    const struct kevent *changelist, int nchanges,
    struct kevent *eventlist, int nevents,
    const struct timespec *timeout)
```

Specifying Events

```
struct kevent {
  uintpt_t ident; // event identifier
  short filter;   // event filter
  u_short flags;  // action flags for kq
  u_int fflags;   // filter flags
  intptr_t data;  // filter data value
  void *udata;    // application data unused by kernel
}
```

- Filter identifies kernel function to execute when there is activity from event source
  - Determines if event needs to be returned to application or not
  - Interpretation of ident (and other) field depends on type of filter in use
    - E.g. for EVFILT_READ/EVFILT_WRITE filters:
      - ident is descriptor number
      - data is number of bytes ready to read, or available space to write

Implementation

- *knote* structure (used by kernel) <==> kevent structure (specified by user)
- kqueue data structure created by kqueue() system call
  - Entered in process open file table, referenced by descriptor returned to user level
  - Two-fold purpose:
    - Queue containing notes ready to be delivered to app
    - Keep track of knotes corresponding to kevents that the app registered interest in
  - Provides:
    - List for knotes ready for delivery
    - Hash table to look up (or keep track of) knotes if “ident” field is *not* a descriptor
    - Linear array of linked lists indexed by descriptor to look up (or keep track of) knotes whose “ident” field *is* a descriptor
      - Corresponds to open file table

Registering Events

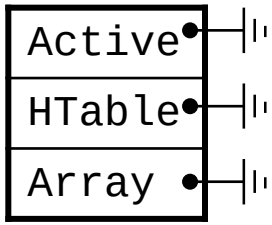
- Application calls kevent() with list of event structs in changelist. For each event:
  - <ident, filter> pair is used to look for existing knote attached to specified kq
    - If not found, a new one may be allocated
  - knote is initialized with data from kevent struct
  - Filter attach routine is called to attach knote to event source
  - If ident is a descriptor, new knote is linked to array; otherwise knote is linked to hash table
  - Changelist may also specify events to remove
    - Same lookup, followed by deletion of knote
- Only after processing changes is kq scanned for active events to pass back to application

Filters and Activity

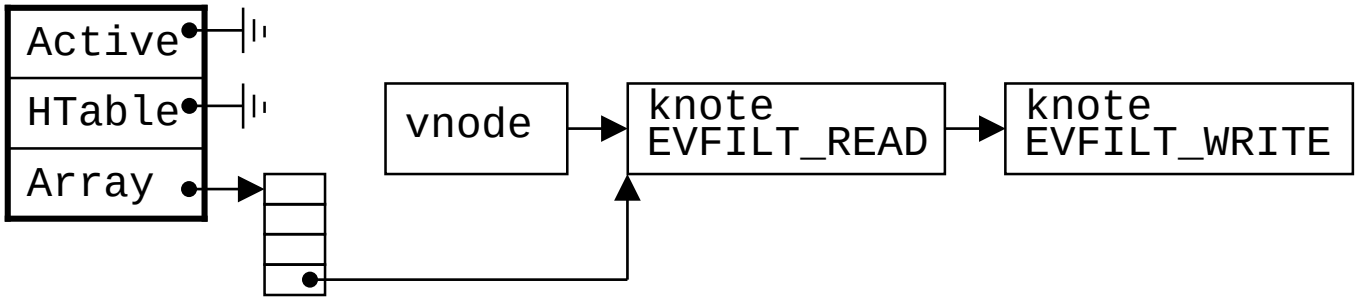
- A filter identifies 3 functions
  - *attach* → code responsible for attaching knote to event source (or object which receives events being monitored)
  - *detach* → code responsible for removing knote from event source
  - *filter* → code executed whenever there is activity at the event source
    - Decides whether activity represents an event that should be reported to application; returns TRUE or FALSE

- Activity at a source (socket data structure, open file object, etc.) causes scan of attached knotes
  - *Filter* function is called for each note
    - If it returns true, knote is linked to kqueue's active list

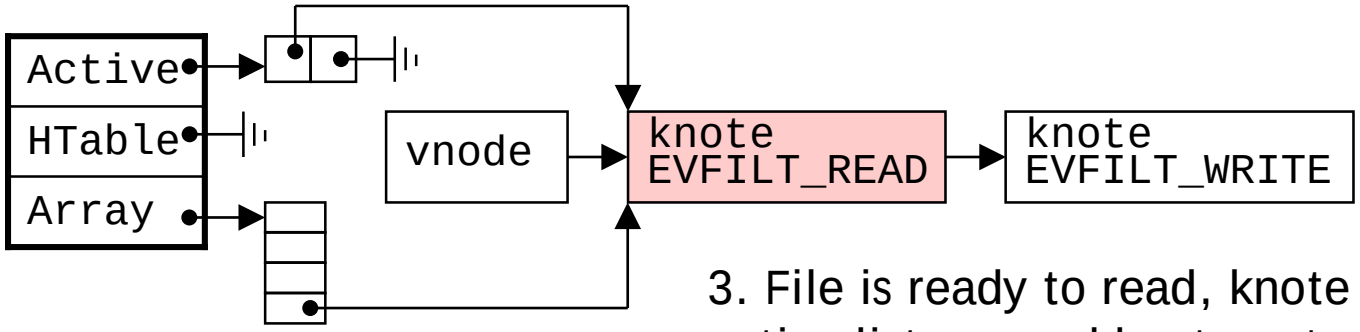
Example



1. New kqueue, kq (no events)

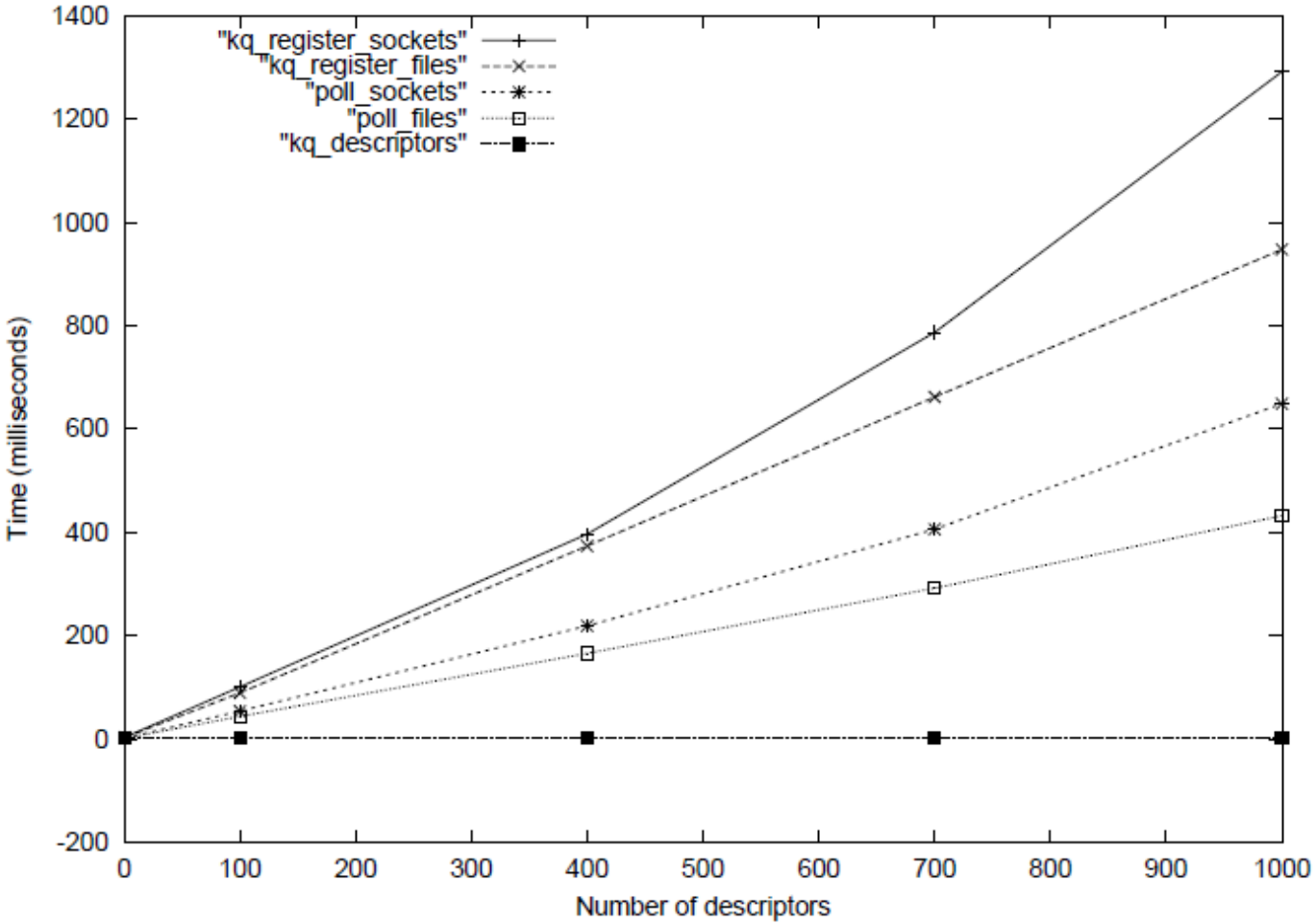


2. Request notification of “ready to read” or “ready to write” (2 events) on open file (descriptor 3)
   - array grows to fit largest descriptor seen

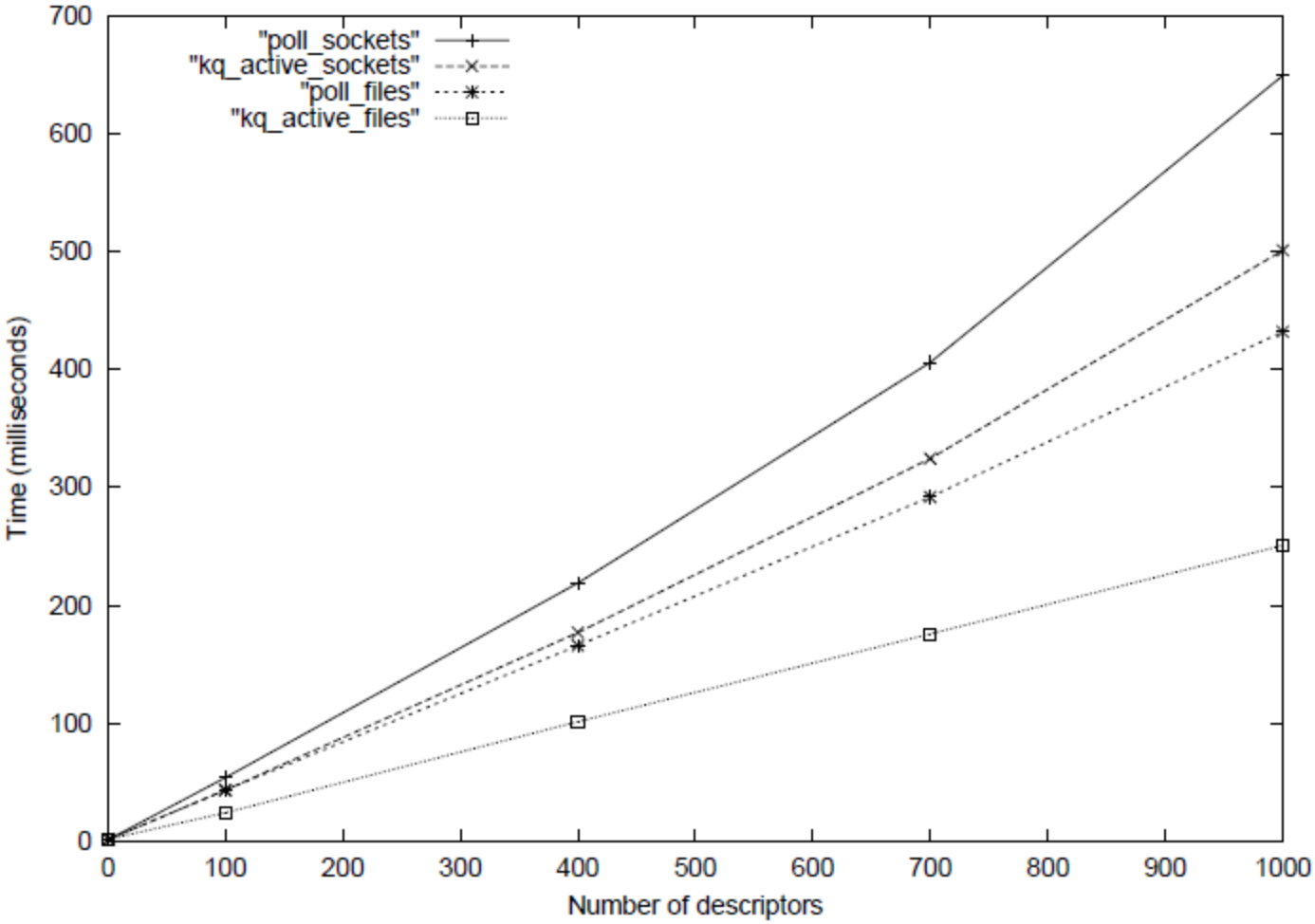


3. File is ready to read, knote linked to active list; second knote not active

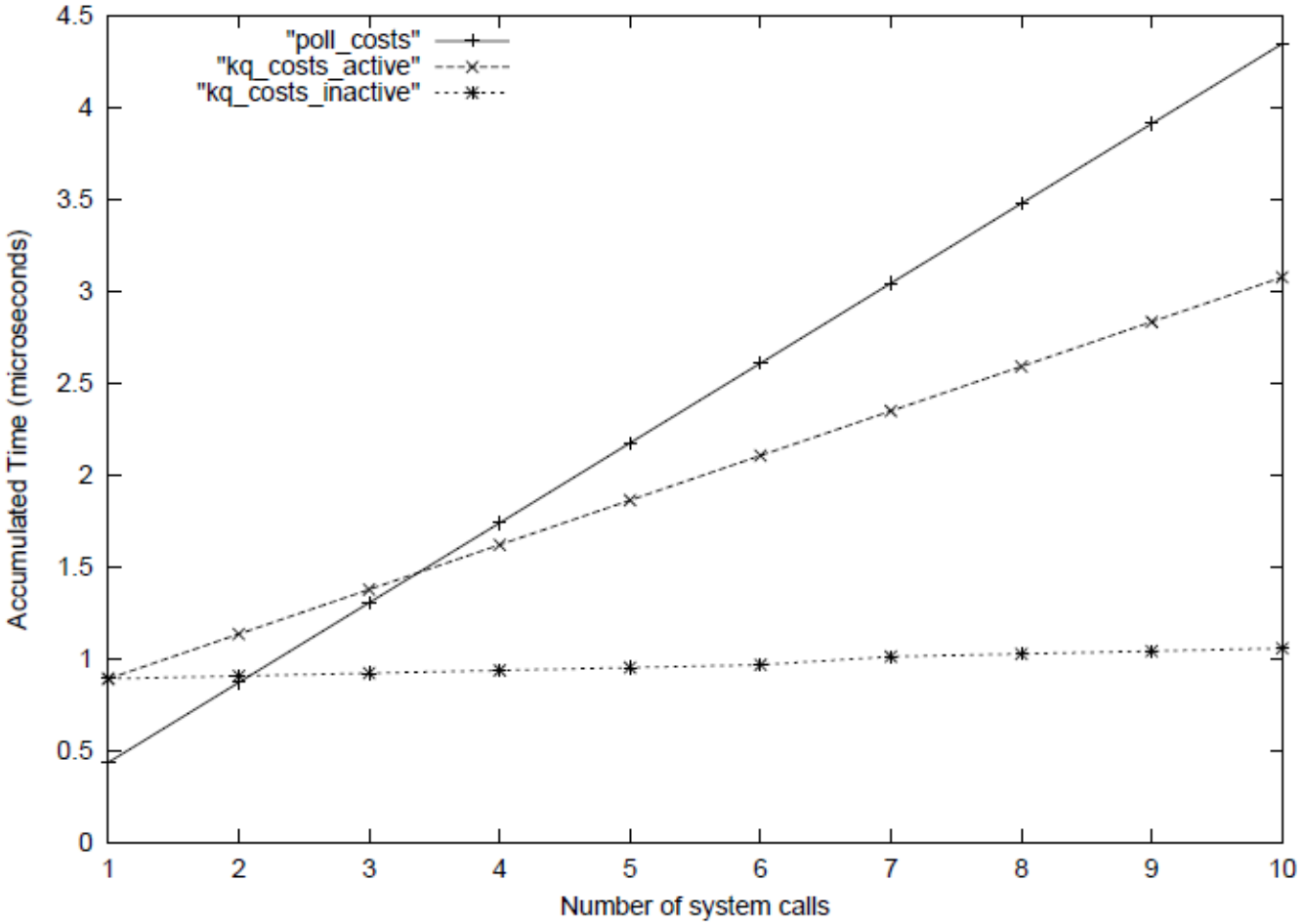
Performance



Performance



Performance



Solutions in Other Systems

- General solution separates *expression of interest* in an event from *notification* of that event

- Solaris 7 (1999) - /dev/poll
  - Open pseudo-device, add/remove descriptors of interest by writing to /dev/poll
  - Retrieve events with ioctl(DP_POLL) on /dev/poll
- Solaris 10 (2004) – unified Event Completion Framework
  - Applications use ports to register and see events on the objects of interest

- Mac OSX – kqueue()/kevent() from FreeBSD
  - Added in 10.3.9 (2005), reportedly quite buggy
  - Better by 10.4.X

Linux Solution

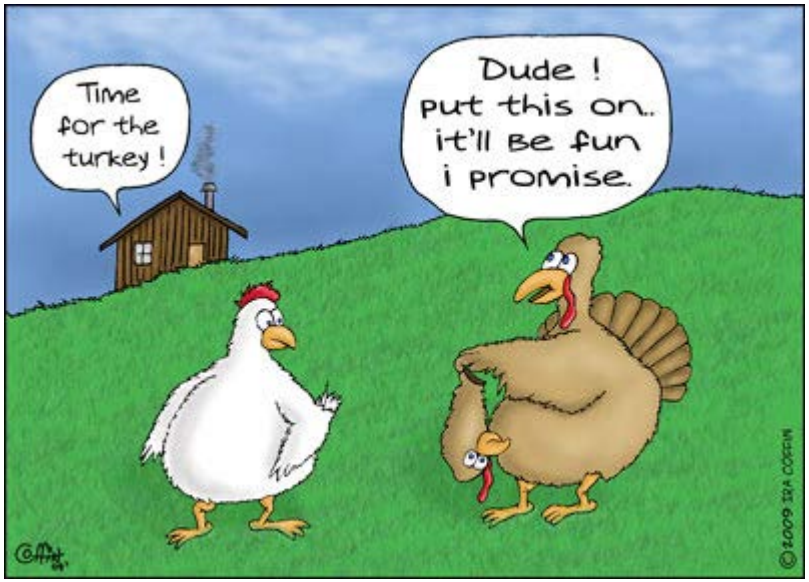
- Linux – epoll()
  - Original patch for 2.4 kernels, now included in 2.6 kernel
  - Supports edge-triggered or level-triggered
  - Coalesces multiple events
  - Register interest with epoll_ctl(), check for events with epoll_wait()
  - Only for events on descriptors
    - Originally, not integrated with asynchronous I/O, signals, or other types of events
- 2006 brought competing proposals for unified event notification mechanism
  - *kevent* API similar to FreeBSD's kqueue and *eventfs*
    - Controversial, eventually died in 2007
  - Descriptor interface for other types of events, use with epoll
    - signalfd(), timerfd(), eventfd() available since 2.6.22
- Libraries! ... see libkqueue

Other Local IPC

- Descriptor based IPC can be heavyweight for local (same machine) communication
- OS typically provides other mechanisms for local use
- Basics:
  - syscall to initialize a kernel object using a *name* or *key* known to all processes that will communicate
  - Other system calls control properties and use of object for IPC
  - Kernel maintains global list of objects referenced by *id* returned to user-level, rather than per-process lists
    - any process can refer to object, but must meet access control restriction to use it
- Examples: message queues, semaphores, shared memory
  - 2 major APIs for each: System V (original) and POSIX (newer)

Next Week…

- No class on Monday – happy Thanksgiving!



- Wednesday: Tutorial (I'm out of town)
- Friday: Lecture