
Lecture 7: Other IPC Mechanisms

CSC 469H1F / CSC 2208H1F

Fall 2007

Angela Demke Brown



Topics

- Messages through file descriptors
 - Pipes / fifos / sockets
 - Receiving notification of activity
- Generalizing the event notification mechanism
 - Kqueue
- System V IPC
 - Message queues, semaphores and shared memory
- Relation to multiprocessor operating systems

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
- 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

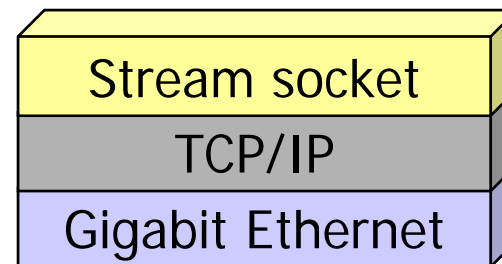
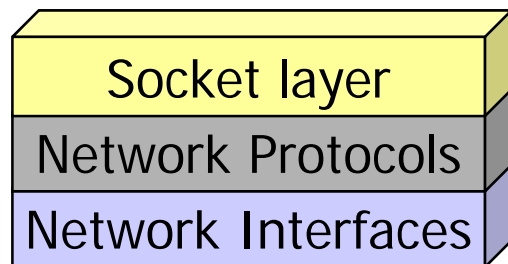
- Fifo - named pipe
 - ✓ can be opened with `open()` syscall as needed (not just prior to fork)
 - ✓ Can be used by processes not directly related to each other
 - ✗ Unidirectional
 - ✗ No message boundaries
- Socket - IPC object supporting various semantics
 - Introduced with 4.2BSD Unix in 1983
 -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

Socket IPC

- A socket is a communication endpoint
- Data is sent from a source socket to a destination socket
- 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

Implementation (BSD)

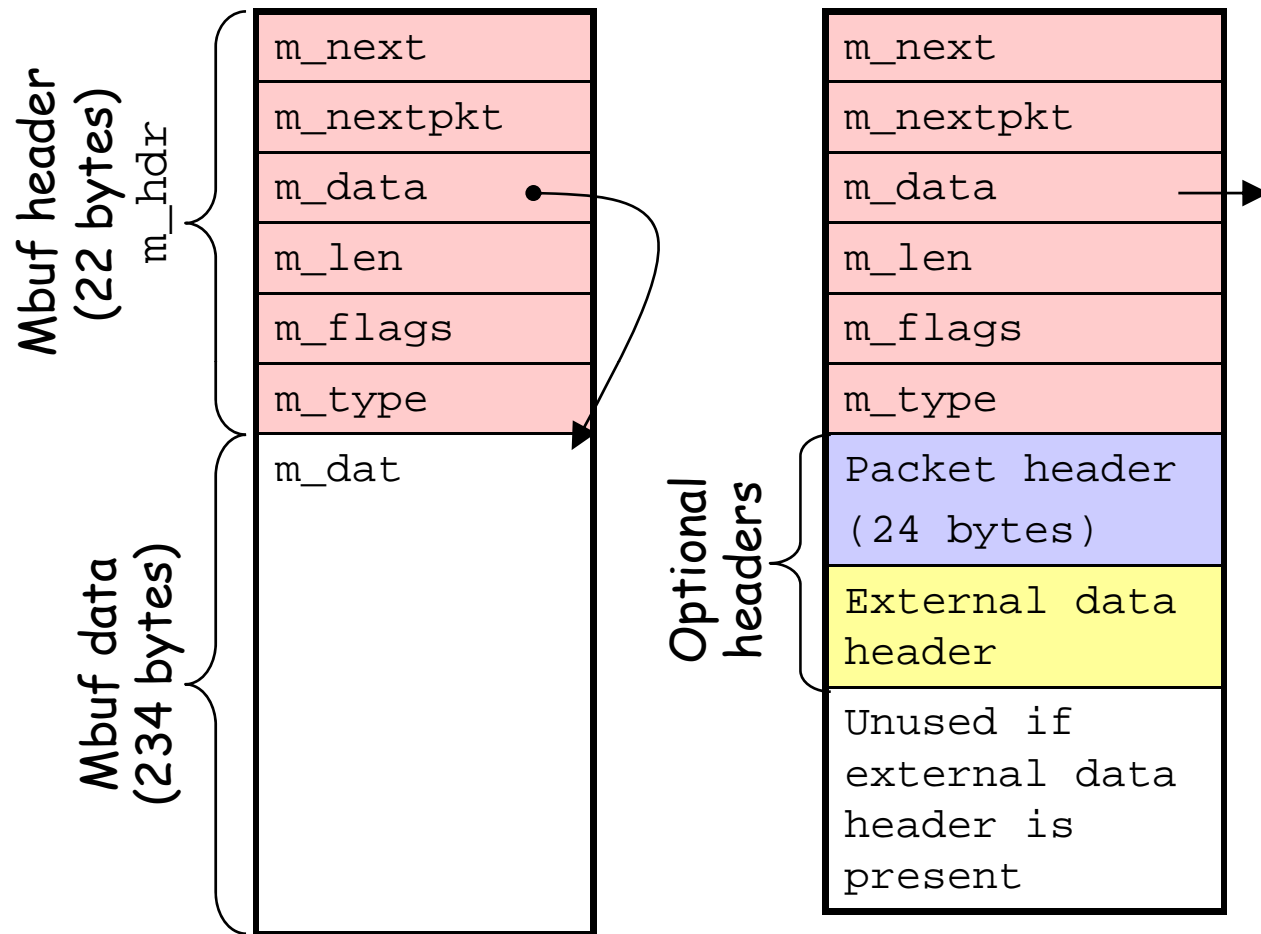
- Socket facility layered on top of networking
- Key issues are
 - Memory management
 - Connection setup
 - Data transfer
 - Connection teardown



Socket Memory Management

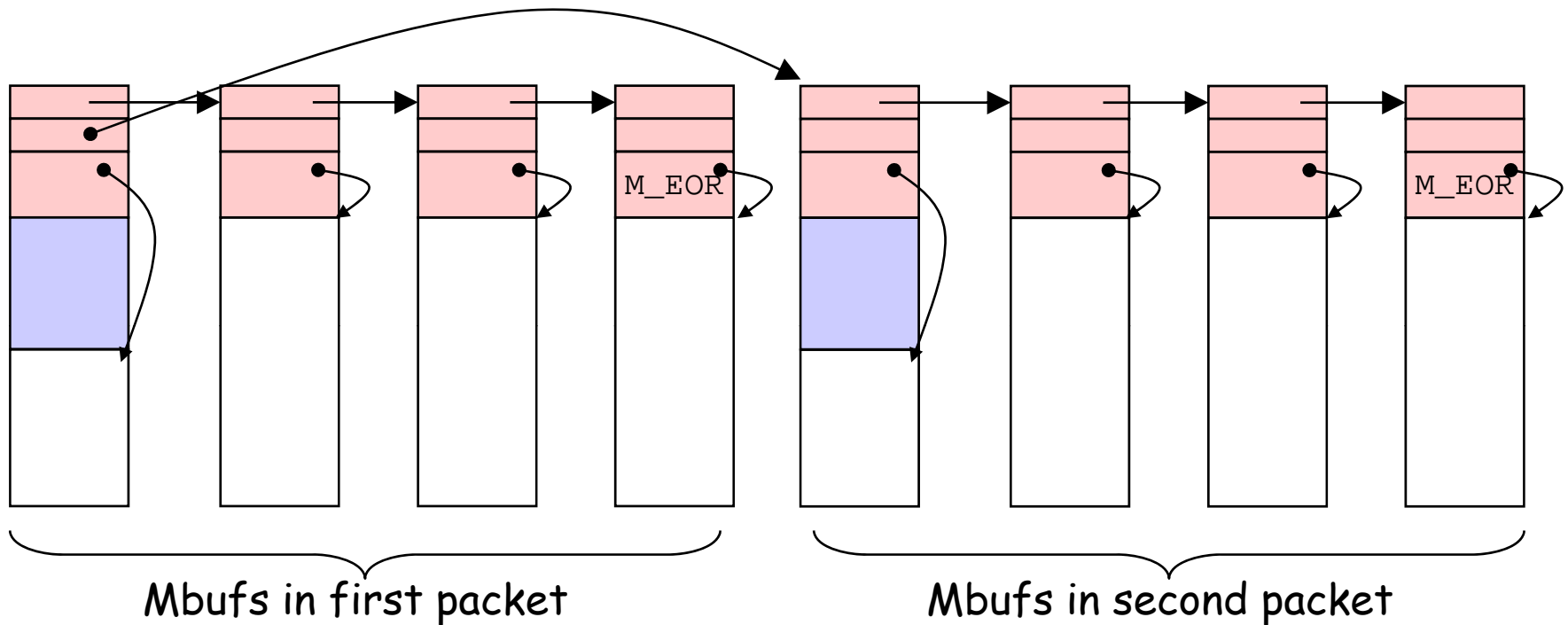
- Main data structure, *mbuf*, used by sockets and network layer
- Requirements: efficient support for
 - Allocation and reclamation
 - Adding and removing headers
 - Dividing data stream into packets for transmission, and combining received packets
 - Moving data between different queues
- Mbufs are allocated from 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 msgs

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)

Connection Setup/Tearardown

- Establishing a connection
 - Client/server model (client and server are user-level apps)
 - Server listen()s on a 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
- Closing a connection
 - If reliable data delivery, socket has to (try to) transmit queued data before closing

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)

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
- 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
- Up to 3 scans over the set is required
 - 1) Kernel scans descriptors to mark activity
 - If no activity, process may be put to sleep until at least 1 descriptor has some
 - 2) On wakeup, kernel scans all descriptors again to mark
 - 3) Application level scans info returned by kernel
- Really specific to descriptors

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 (several thousand)
 - Flexible
 - Simple interface
 - Expand information conveyed
 - Reliable
 - "level-triggered" not "edge triggered"

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 {
    uintptr_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 whether 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* data structure used by kernel, corresponding to 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
 - 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

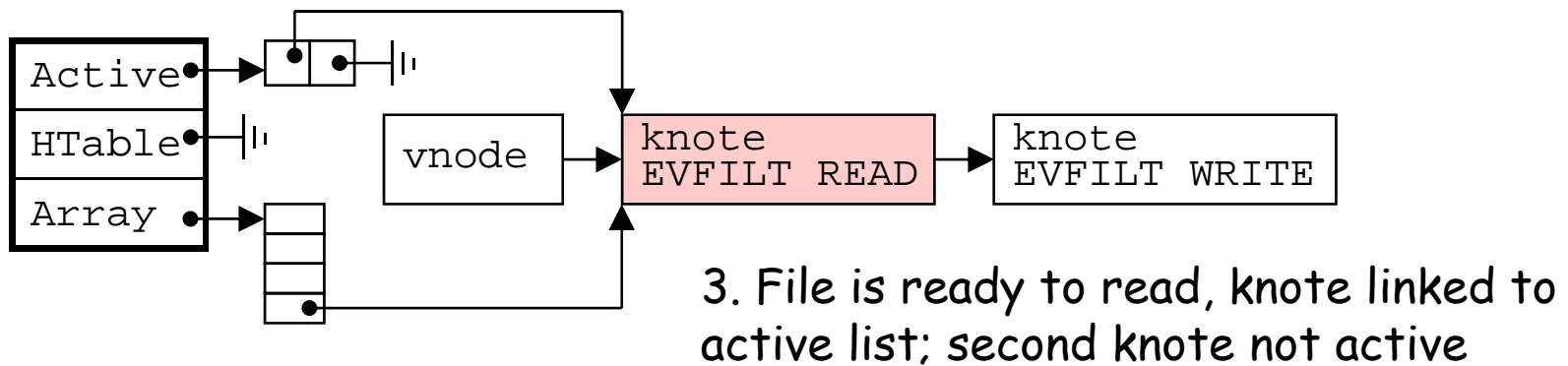
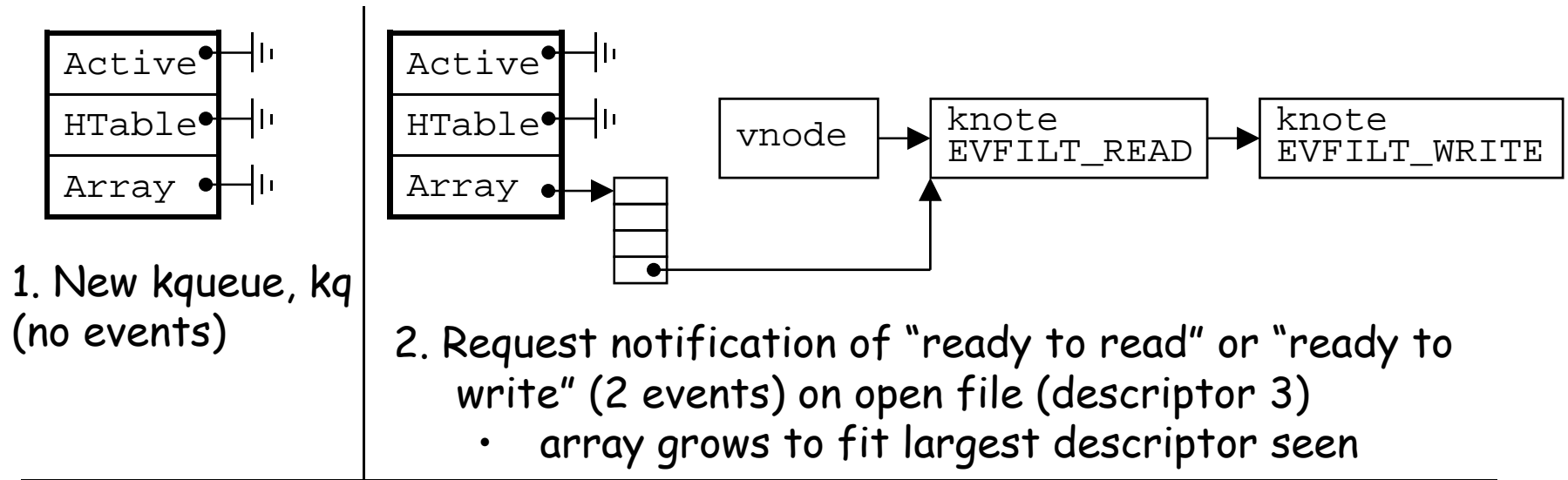
Registering Events

- 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
 - May also record values in knote's fflag or data fields
- 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



Solutions in Other Systems

- General solution separates *expression of interest* in an event from *notification* of that event
 - 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 (not integrated with asynchronous I/O, signals, or other types of events)
 - Newest proposals (2006) are for a unified *kevent* API similar to FreeBSD's `kqueue`
 - Solaris - `/dev/poll`
 - Added in Solaris 7 (1999)
 - Open pseudo-device, add/remove descriptors of interest by writing to `/dev/poll`
 - Retrieve events with `ioctl(DP_POLL)` on `/dev/poll`
 - Mac OSX - `kqueue()/kevent()` from FreeBSD
 - Added in 10.3.9 (2005), reportedly quite buggy, but better in 10.4.X

Other Local IPC

- Descriptor based IPC can be heavyweight for local (same machine) communication
- OS typically provides specialized 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 characteristics of object and use object 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)
- Types of Local IPC
 - General message queues
 - Send/receive messages with application-specified types
 - Similar to socket IPC without network protocols
 - Semaphores
 - Kernel support for process synchronization
 - Shared memory
 - Built on virtual memory mapping system
- 2 major API's for each: System V and Posix

Multiprocessor OS

- Internally, OS is much like a collection of communicating processes
- Uses similar mechanisms to coordinate activity and manage the hardware
 - For multiprocessor and distributed operating systems, explicit messages may be preferable to shared memory
 - We'll start looking at this next time...