An introduction to Linux IPC

linux.conf.au 2013

Canberra, Australia
2013-01-30

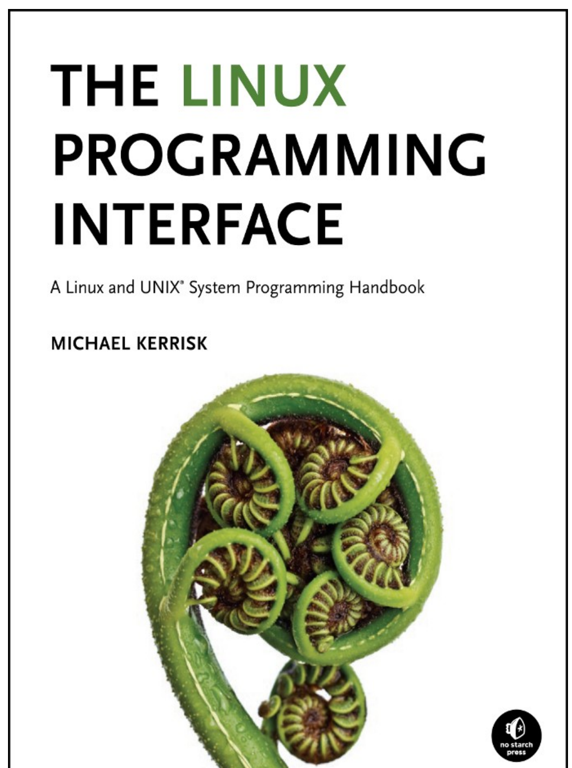
Michael Kerrisk © 2013
http://man7.org/
mtk@man7.org
http://lwn.net/
mtk@lwn.net

Goal

- Limited time!
- Get a flavor of main IPC methods

Me

- Programming on UNIX & Linux since 1987
- Linux *man-pages* maintainer
  - http://www.kernel.org/doc/man-pages/
  - Kernel + glibc API
- Author of:



Further info:
http://man7.org/tlpi/

You

- Can read a bit of C
- Have a passing familiarity with common syscalls
    - *fork()*, *open()*, *read()*, *write()*

There’s a lot of IPC

- Pipes
- FIFOs
- Pseudoterminals
- Sockets
  - Stream vs Datagram (vs Seq. packet)
  - UNIX vs Internet domain
- POSIX message queues
- POSIX shared memory
- POSIX semaphores
  - Named, Unnamed
- System V message queues
- System V shared memory
- System V semaphores
- Shared memory mappings
  - File vs Anonymous
- Cross-memory attach
  - proc_vm_readv() / proc_vm_writev()
- Signals
  - Standard, Realtime
- Eventfd
- Futexes
- Record locks
- File locks
- Mutexes
- Condition variables
- Barriers
- Read-write locks

It helps to classify

- Pipes
- FIFOs
- Pseudoterminals
- Sockets
  - Stream vs Datagram (vs Seq. packet)
  - UNIX vs Internet domain
- POSIX message queues
- POSIX shared memory
- POSIX semaphores
  - Named, Unnamed
- System V message queues
- System V shared memory
- System V semaphores
- Shared memory mappings
  - File vs Anonymous
- Cross-memory attach
  - proc_vm_readv() / proc_vm_writev()
- Signals
  - Standard, Realtime
- Eventfd
- Futexes
- Record locks
- File locks
- Mutexes
- Condition variables
- Barriers
- Read-write locks

It helps to classify

- Pipes
- FIFOs
- Pseudoterminals
- Sockets
  - Stream vs Datagram (vs Seq. packet)
  - UNIX vs Internet domain
- POSIX message queues
- POSIX shared memory
- POSIX semaphores
  - Named, Unnamed
- System V message queues
- System V shared memory
- System V semaphores

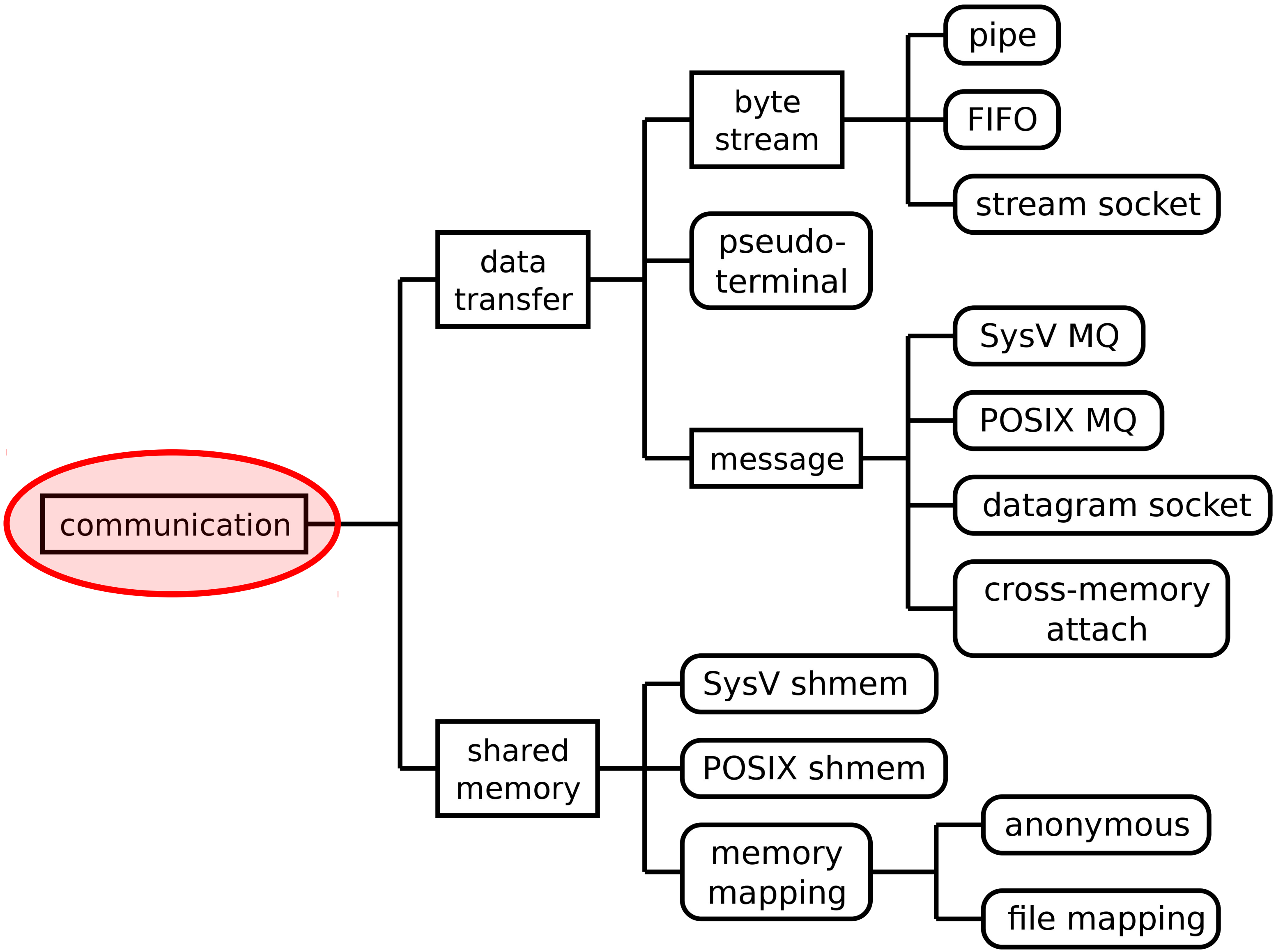
- Shared memory mappings
  - File vs Anonymous
- Cross-memory attach
  - proc_vm_readv() / proc_vm_writev()
- Signals
  - Standard, Realtime
- Eventfd
- Futexes
- Record locks
- File locks
- Mutexes
- Condition variables
- Barriers
- Read-write locks

Communication

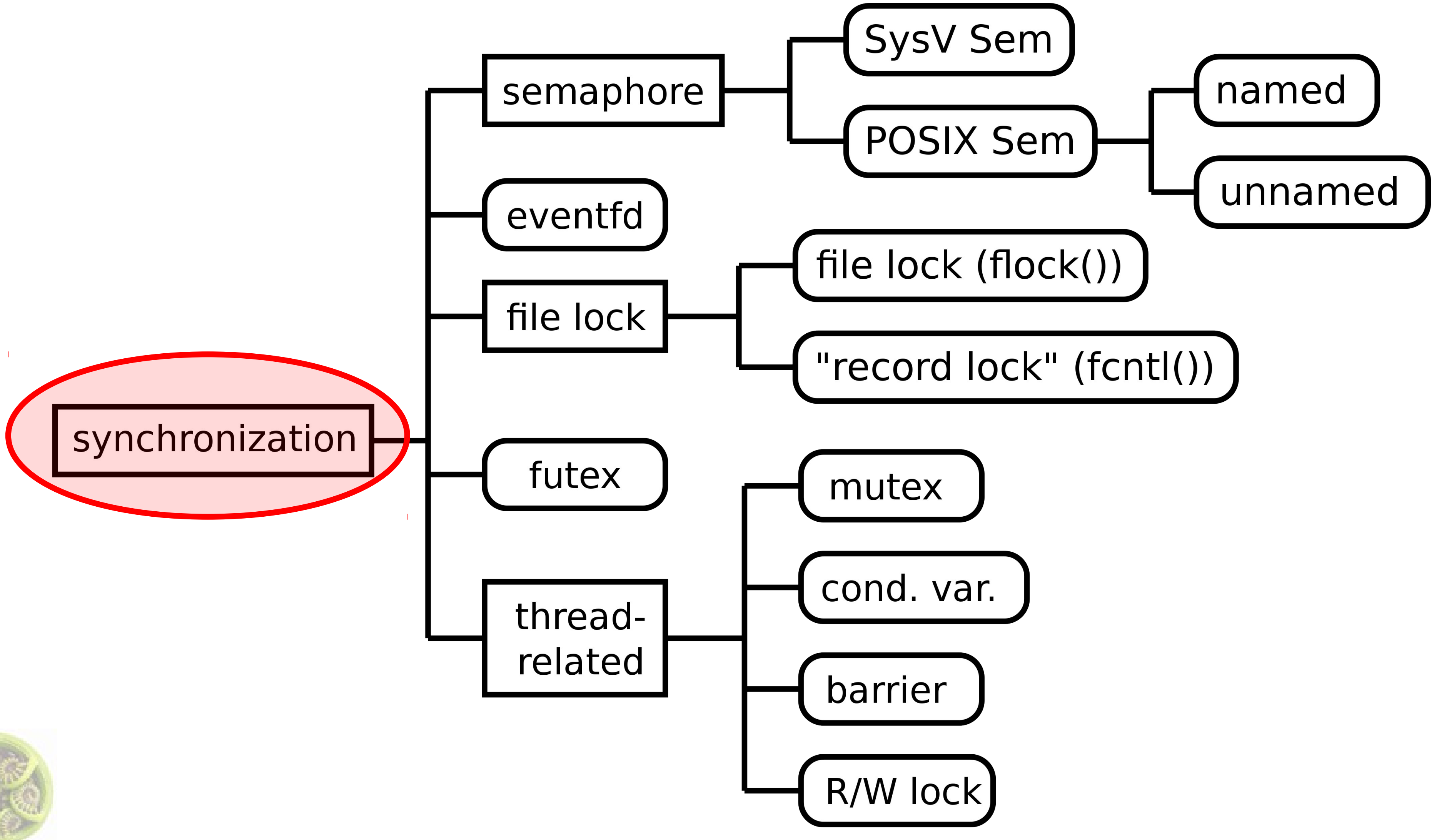
Signals

Synchronization

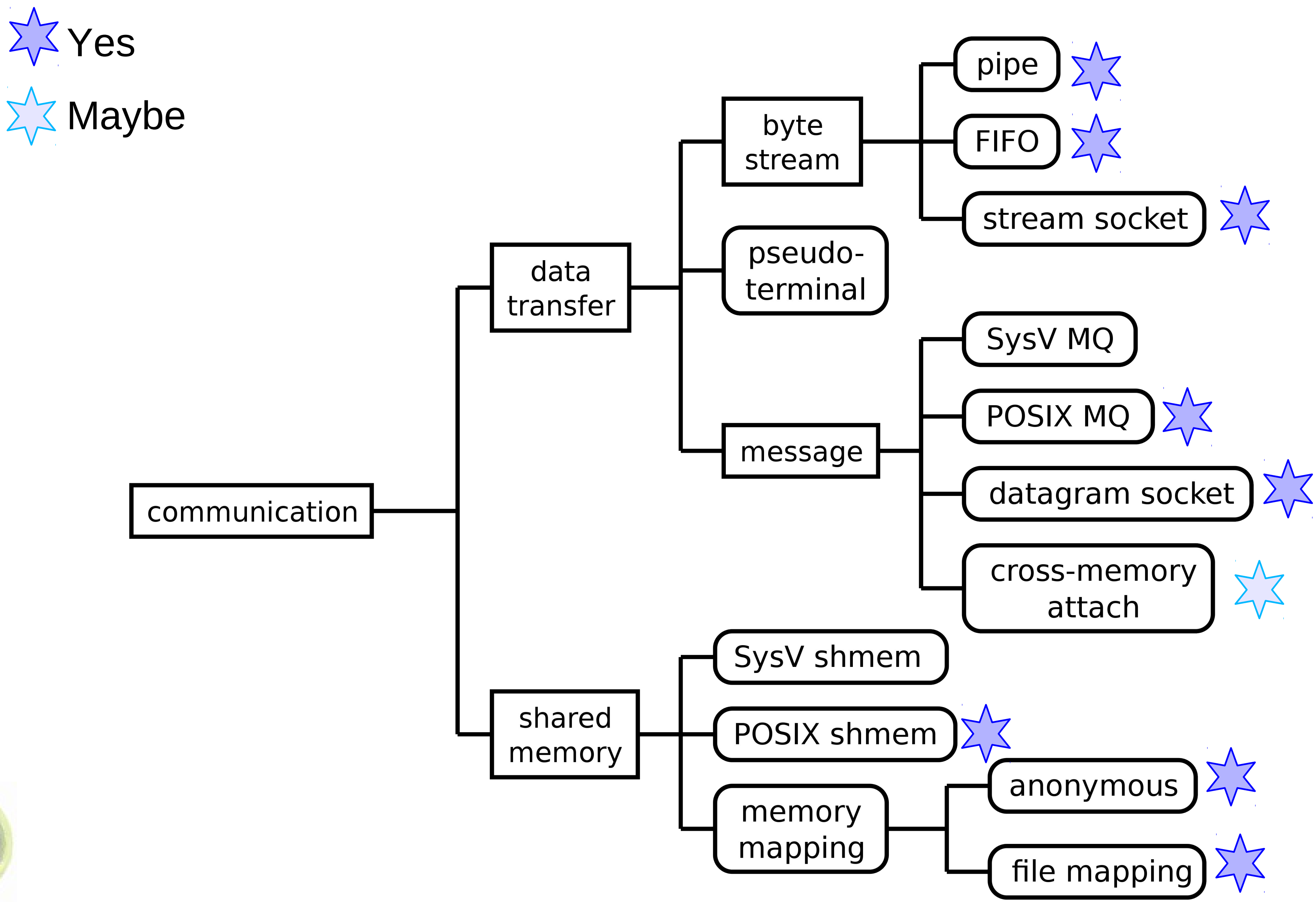
Communication



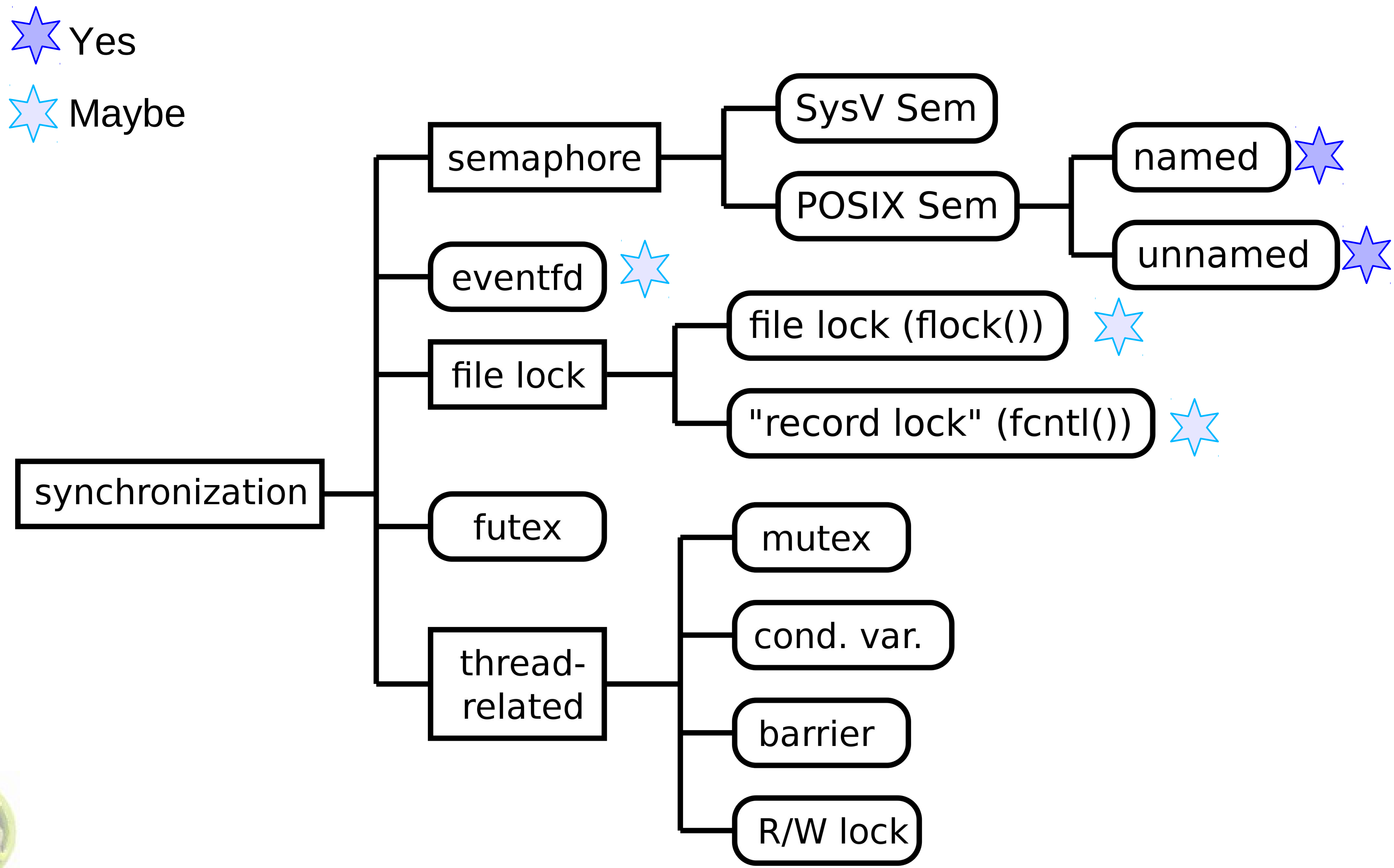
Synchronizatoin



What we'll cover



What we'll cover



What is not covered

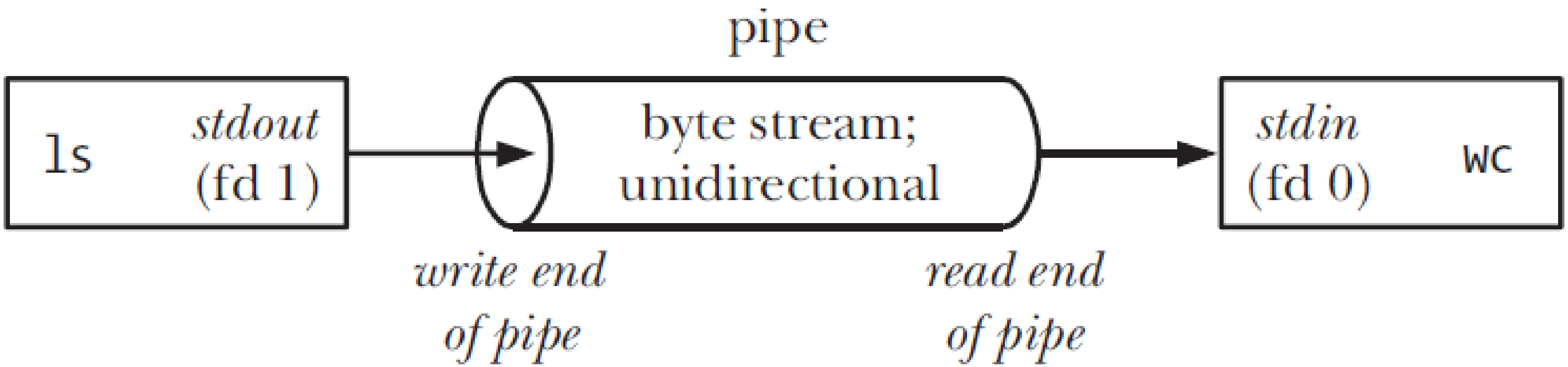
- Signals
  - Can be used for communication and sync, but poor for both
- System IPC
  - Similar in concept to POSIX IPC
  - But interface is terrible!
  - Use POSIX IPC instead
- Thread sync primitives
  - Mutexes, condition vars, barriers, R/W locks
  - Can use process shared, but rare (and nonportable)
- Futexes
  - Very low level
  - Used to implement POSIX sems, mutexes, condvars
- Pseudoterminals
  - Specialized use cases

Communication techniques

Pipes

Pipes

`ls | wc -l`



pipe
ls
stdout
(fd 1)
byte stream;
unidirectional
stdin
(fd 0)
wc
write end
of pipe
read end
of pipe

Pipes

- Pipe == byte stream buffer in kernel
  - Sequential (can’t *lseek()*)
  - Multiple readers/writers difficult
- Unidirectional
  - Write end + read end

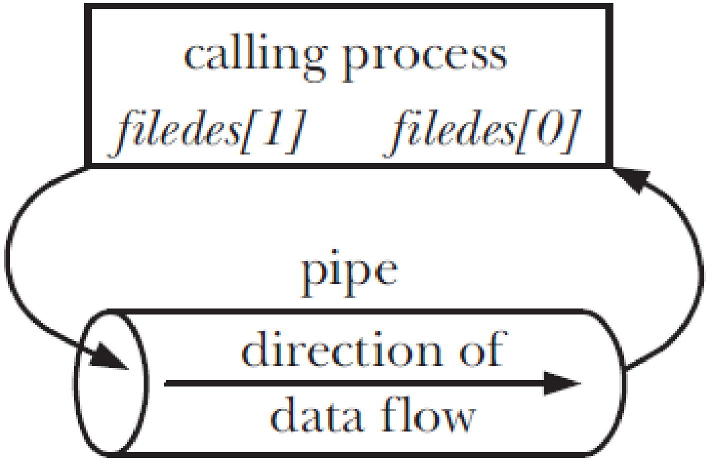
Creating and using pipe

- Created using *pipe()*:

```
int filedes[1];
pipe(filedes);

...

write(filedes[1], buf, count);
read(filedes[0], buf, count);
```



Sharing a pipe

- Pipes are anonymous
  - No name in file system
- How do two processes share a pipe?

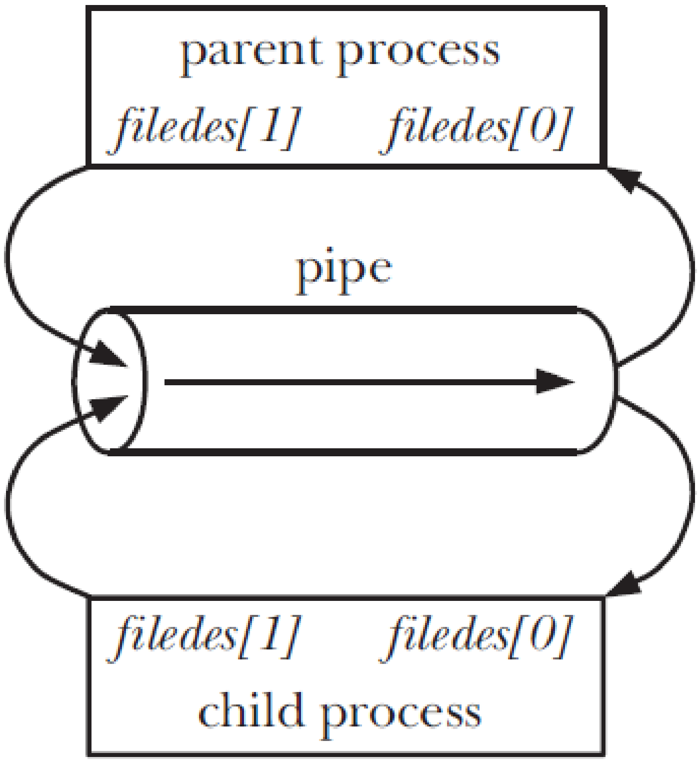
Sharing a pipe

```
int filedes[2];

pipe(filedes);

child_pid = fork();
```

fork() duplicates parent's file descriptors

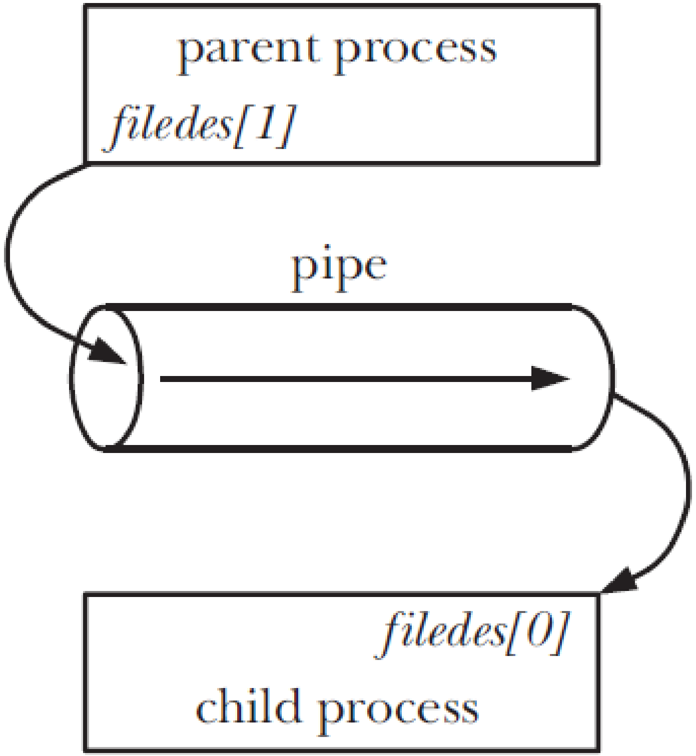


Sharing a pipe

```
int filedes[2];

pipe(filedes);

child_pid = fork();
if (child_pid == 0) {
    close(filedes[1]);
    /* Child now reads */
} else {
    close(filedes[0]);
    /* Parent now writes */
}
```



(error checking omitted!)

Closing unused file descriptors

- Parent and child must close unused descriptors
  - *Necessary for correct use of pipes!*
- *close()* write end
  - *read()* returns 0 (EOF)
- *close()* read end
  - *write()* fails with EPIPE error + SIGPIPE signal

```
// http://man7.org/tlpi/code/online/dist/pipes/simple_pipe.c.html
// Create pipe, create child, parent writes argv[1] to pipe, child reads
    pipe(pfd);                    /* Create the pipe */
    switch (fork()) {
    case 0:                       /* Child  - reads from pipe */
        close(pfd[1]);            /* Write end is unused */
        for (;;) {                /* Read data from pipe, echo on stdout */
            numRead = read(pfd[0], buf, BUF_SIZE);
            if (numRead <= 0) break;      /* End-of-file or error */
            write(STDOUT_FILENO, buf, numRead);
        }
        write(STDOUT_FILENO, "\n", 1);
        close(pfd[0]);
        ...
    default:                       /* Parent - writes to pipe */
        close(pfd[0]);             /* Read end is unused */
        write(pfd[1], argv[1], strlen(argv[1]));
        close(pfd[1]);             /* Child will see EOF */
        ...
    }
```

I/O on pipes

- *read()* blocks if pipe is empty
- *write()* blocks if pipe is full
- Writes <= `PIPE_BUF` guaranteed to be atomic
  - Multiple writers > `PIPE_BUF` may be interleaved
  - POSIX: `PIPE_BUF` at least 512B
  - Linux: `PIPE_BUF` is 4096B
- Can use *dup2()* to connect filters via a pipe
  - http://man7.org/tlpi/code/online/dist/pipes/pipe_ls_wc.c.html

Pipes have limited capacity

- Limited capacity
  - If pipe fills, *write()* blocks
  - Before Linux 2.6.11: 4096 bytes
  - Since Linux 2.6.11: 65,536 bytes
  - Apps should be designed not to care about capacity
    - But, Linux has *fcntl(fd, F_SETPIPE_SZ, size)*
      - (not portable)

FIFOs
(named pipes)

FIFO (named pipe)

- (Anonymous) pipes can only be used by related processes
- FIFOs == pipe with name in file system
- Creation:
  - *mkfifo(pathname, permissions)*
- Any process can open and use FIFO
- I/O is same as for pipes

Opening a FIFO

- *open(pathname, O_RDONLY)*
  - Open read end
- *open(pathname, O_WRONLY)*
  - Open write end
- *open()* locks until other end is opened
  - Opens are synchronized
  - *open(pathname, O_RDONLY | O_NONBLOCK)* can be useful

POSIX Message Queues

Highlights of POSIX MQs

- Message-oriented communication
  - Receiver reads messages one at a time
    - No partial or multiple message reads
  - Unlike pipes, multiple readers/writers can be useful
- Messages have priorities
  - Delivered in priority order
- Message notification feature

POSIX MQ API

- Queue management (analogous to files)
  - *mq_open()*: open/create MQ, set attributes
  - *mq_close()*: close MQ
  - *mq_unlink()*: remove MQ pathname
- I/O:
  - *mq_send()*: send message
  - *mq_receive()*: receive message
- Other:
  - *mq_setattr()*, *mq_getattr()*: set/get MQ attributes
  - *mq_notify()*: request notification of msg arrival

Opening a POSIX MQ

- *mqd = mq_open(name, flags [, mode, &attr]);*
- Open+create new MQ / open existing MQ
- *name* has form `/somename`
  - Visible in a pseudo-filesystem
- Returns *mqd_t*, a message queue descriptor
  - Used by rest of API

Opening a POSIX MQ

- *mqd = mq_open(name, flags [, mode, &attr]);*
- *flags* (analogous to *open()*):
  - `O_CREAT` – create MQ if it doesn't exist
  - `O_EXCL` – create MQ exclusively
  - `O_RDONLY`, `O_WRONLY`, `O_RDWR` – just like file open
  - `O_NONBLOCK` – non-blocking I/O
- *mode* sets permissions
- *&attr*: attributes for new MQ
  - `NULL` gives defaults

Opening a POSIX MQ

- Examples:

```
// Create new MQ, exclusive,
// for writing
mqd = mq_open("/mymq",
          O_CREAT| O_EXCL | O_WRONLY,
          0600, NULL);

// Open existing queue for reading
mqd = mq_open("/mymq", O_RDONLY);
```

Unlink a POSIX MQ

- *mq_unlink(name);*
- MQs are reference-counted
  - ==> MQ removed only after all users have closed

Nonblocking I/O on POSIX MQs

- Message ques have a limited capacity
  - Controlled by attributes
- By default:
  - *mq_receive()* blocks if no messages in queue
  - *mq_send()* blocks if queue is full
- `O_NONBLOCK`:
  - `EAGAIN` error instead of blocking
  - Useful for emptying queue without blocking

Sending a message

- *mq_send(mqd, msg_ptr, msg_len, msgprio);*
  - *mqd* – MQ descriptor
  - *msg_ptr* – pointer to bytes forming message
  - *msg_len* – size of message
  - *msgprio* – priority
    - non-negative integer
    - 0 is lowest priority

Sending a message

- *mq_send(mqd, msg_ptr, msg_len, msgprio);*
- Example:

```
mqd_t mqd;
mqd = mq_open("/mymq",
              O_CREAT | O_WRONLY,
              0600, NULL);
char *msg = "hello world";
mq_send(mqd, msg, strlen(msg), 0);
```

http://man7.org/tlpi/code/online/dist/pmsg/pmsg_send.c.html

Receiving a message

- *nb = mq_receive(mqd, msg_ptr, msg_len, &prio);*
  - *mqd* – MQ descriptor
  - *msg_ptr* – points to buffer that receives message
  - *msg_len* – size of buffer
  - *&prio* – receives priority
  - *nb* – returns size of message (bytes)

Receiving a message

- *nb = mq_receive(mqd, msg_ptr, msg_len, &prio);*
- Example:

```
const int BUF_SIZE = 1000;
char buf[BUF_SIZE];
unsigned int prio;
...
mqd = mq_open("/mymq", O_RDONLY);
nbytes = mq_receive(mqd, buf,
                    BUF_LEN, &prio);
```

http://man7.org/tlpi/code/online/dist/pmsg/pmsg_receive.c.html

POSIX MQ notifications

- *mq_notify(mqd, notification);*
- One process can register to receive notification
- Notified when new msg arrives on *empty* queue
  - & only if another process is not doing *mq_receive()*
- *notification* says how caller should be notified
  - Send me a signal
  - Start a new thread (see *mq_notify(3)* for example)
- One-shot; must re-enable
  - Do so before emptying queue!

POSIX MQ attributes

```
struct mq_attr {
  long mq_flags;   // MQ description flags
                   // 0 or O_NONBLOCK
                   // [mq_getattr(), mq_setattr()]
  long mq_maxmsg;  // Max. # of msgs on queue
                   // [mq_open(), mq_getattr()]
  long mq_msgsize; // Max. msg size (bytes)
                   // [mq_open(), mq_getattr()]
  long mq_curmsgs; // # of msgs currently in queue
                   // [mq_getattr()]
};
```

POSIX MQ details

- Per-process and system-wide limits govern resource usage
- Can mount filesystem to obtain info on MQs:

```
# mkdir /dev/mqueue
# mount -t mqueue none /dev/mqueue
# ls /dev/mqueue
mymq
# cat /dev/mqueue/mymq
QSIZE:129  NOTIFY:2  SIGNO:0  NOTIFY_PID:8260
```

- See *mq_overview(7)*

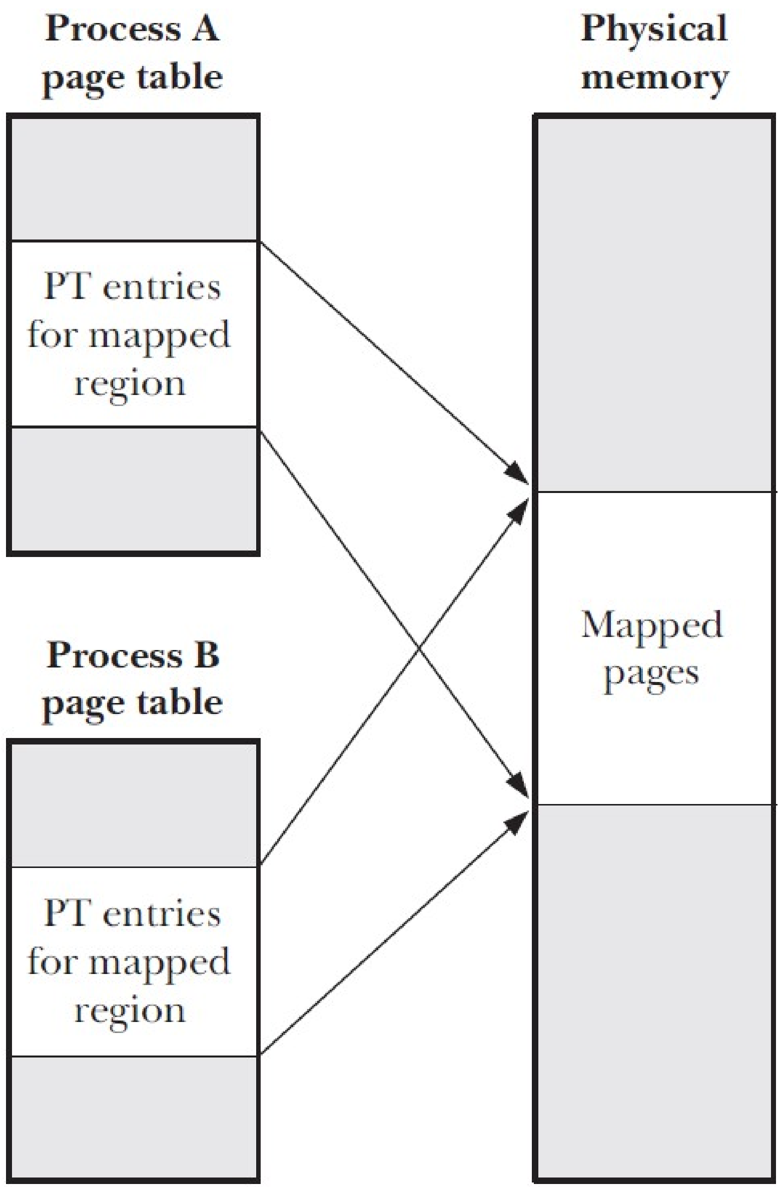
Shared memory

Shared memory

- Processes share same physical pages of memory
- Communication == copy data to memory
- Efficient; compare
  - Data transfer: user space ==> kernel ==> user space
  - Shared memory: single copy in user space
- But, need to synchronize access...

Shared memory

- Processes share physical pages of memory



Shared memory

- We'll cover three types:
  - Shared anonymous mappings
    - *related* processes
  - Shared file mappings
    - unrelated processes, backed by file in traditional filesystem
  - POSIX shared memory
    - unrelated processes, without use of traditional filesystem

mmap()

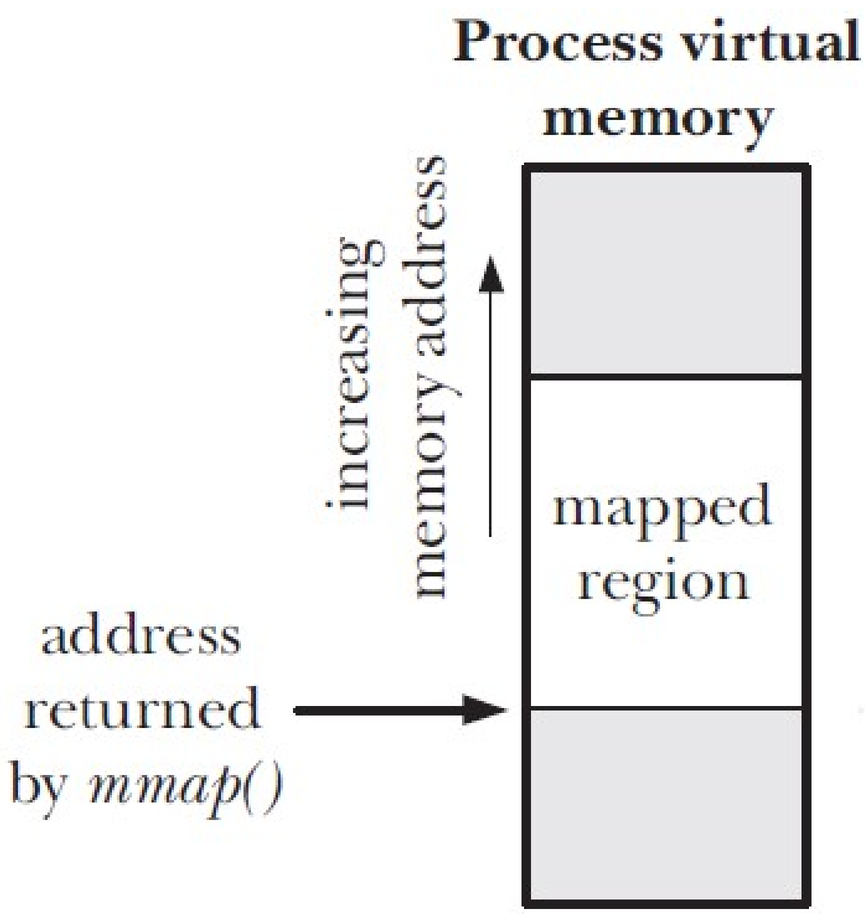
- Syscall used in all three shmem types
- Rather complex:
  - *void *mmap(void *daddr, size_t len, int prot,*
    int flags, int fd, off_t offset);

mmap()

- *addr = mmap(daddr, len, prot, flags, fd, offset);*
- *daddr* – choose where to place mapping;
  - Best to use `NULL`, to let kernel choose
- *len* – size of mapping
- *prot* – memory protections (read, write, exec)
- *flags* – control behavior of call
  - `MAP_SHARED`, `MAP_ANONYMOUS`
- *fd* – file descriptor for file mappings
- offset – starting offset for mapping from file
- *addr* – returns address used for mapping

Using shared memory

- *addr = mmap(daddr, len, prot, flags, fd, offset);*
- *addr* looks just like any C pointer
- But, changes to region seen by all process that map it



Shared anonymous mapping

Shared anonymous mapping

- Share memory between *related* processes
- *mmap() fd* and *offset* args unneeded

```
addr = mmap(NULL, length,
            PROT_READ | PROT_WRITE,
            MAP_SHARED | MAP_ANONYMOUS,
            -1, 0);
pid = fork();
```

- Allocates zero-initialized block of *length* bytes
- Parent and child share memory at *addr:length*
  - *http://man7.org/tlpi/code/online/dist/mmap/anon_mmap.c.html*

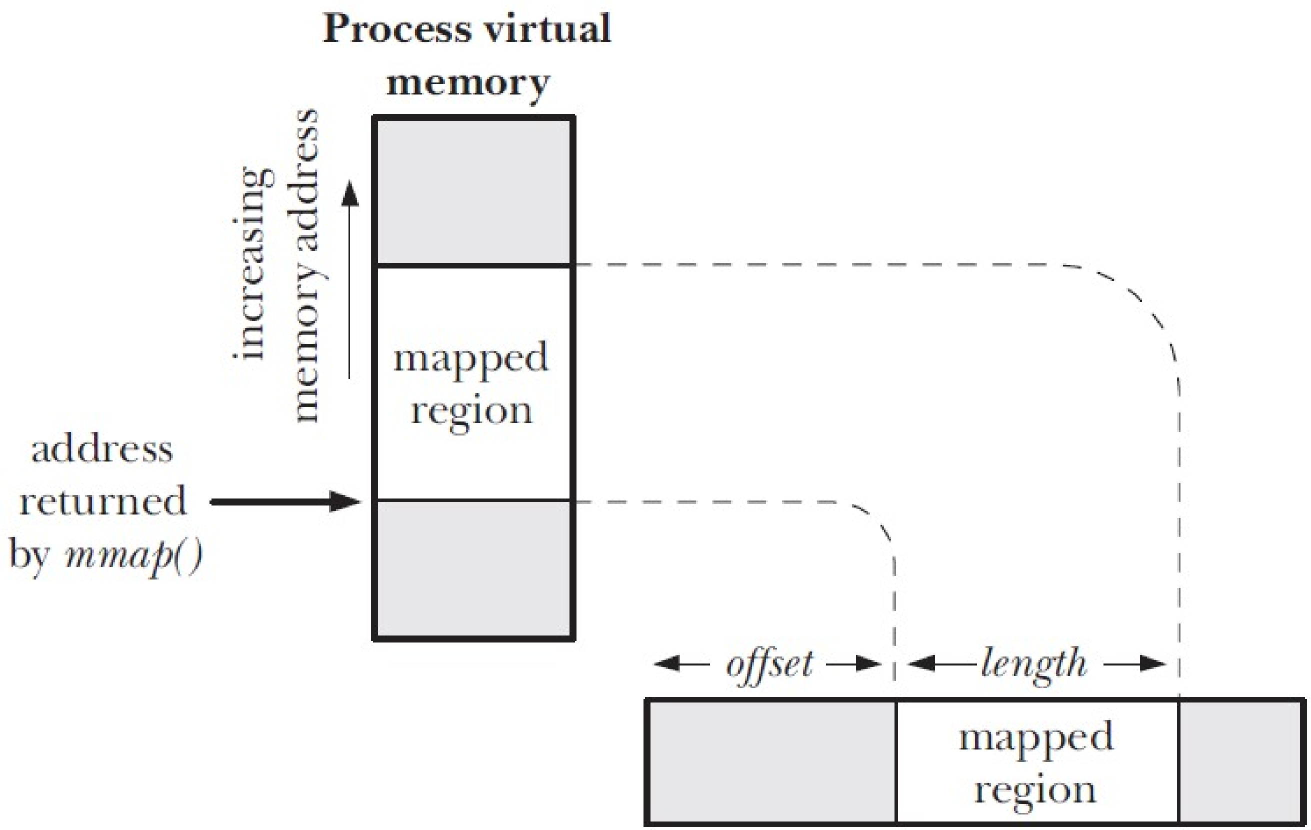
Shared anonymous mapping

```
addr = mmap(NULL, length,
            PROT_READ | PROT_WRITE,
            MAP_SHARED | MAP_ANONYMOUS,
            -1, 0);
pid = fork();
```

Shared file mapping

Shared file mapping

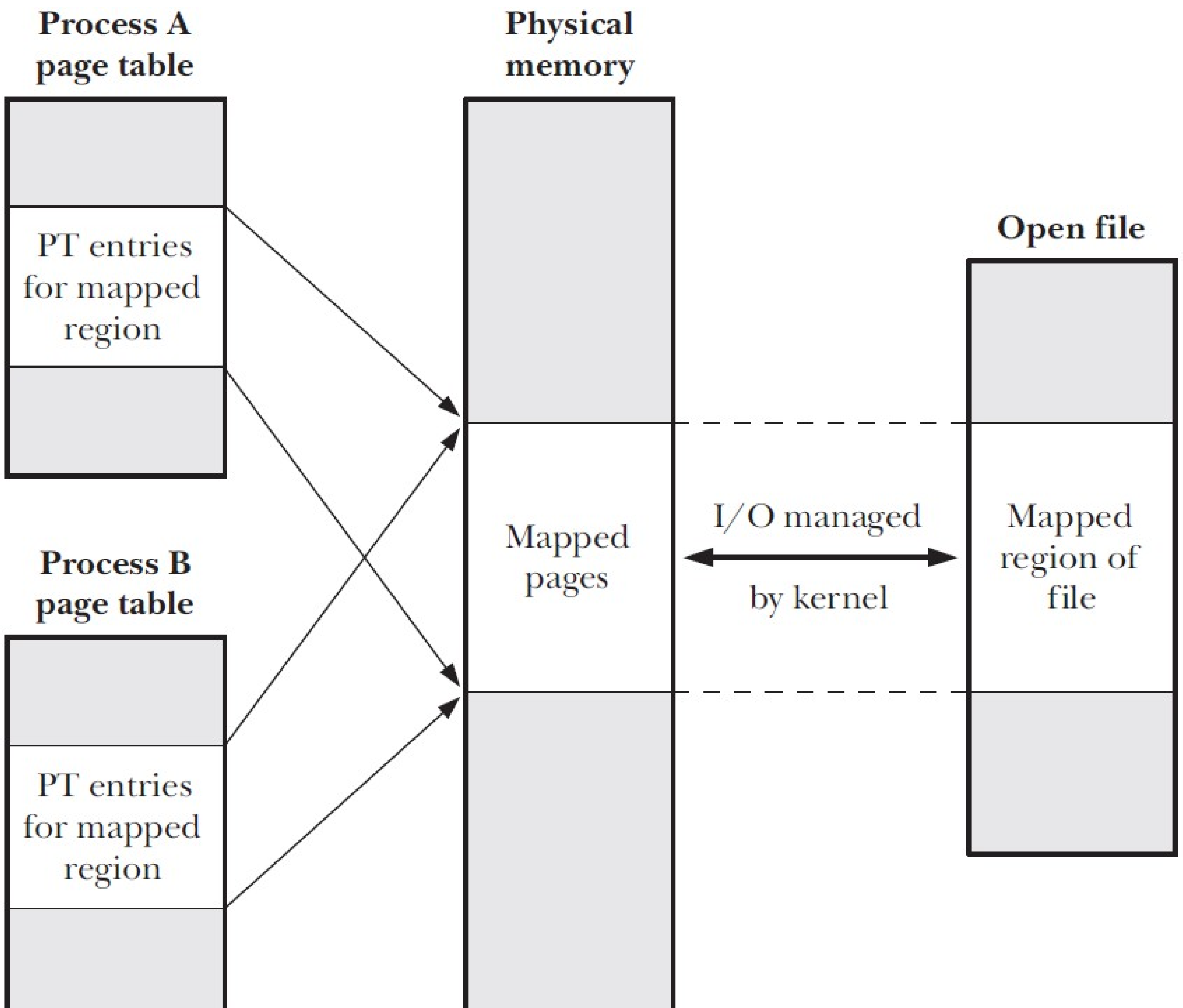
- Share memory between unrelated processes, backed by file
- *fd = open(...); addr = mmap(..., fd, offset);*



Shared file mapping

- *fd = open(...); addr = mmap(..., fd, offset);*
- Contents of memory initialized from file
- Updates to memory automatically carried through to file (“memory-mapped I/O”)
- All processes that map same region of file share same memory

Shared file mapping



Shared file mapping

```
fd = open(pathname, O_RDWR);

addr = mmap(NULL, length,
            PROT_READ | PROT_WRITE,
            MAP_SHARED,
            fd, 0);
...
close(fd);     /* No longer need 'fd' */
```

Updates are: visible to other process sharing mapping; and carried through to file

POSIX shared memory

POSIX shared memory

- Share memory between unrelated process, without creating file in (traditional) filesystem
  - Don’t need to create a file
  - Avoid file I/O overhead

POSIX SHM API

- Object management
  - *shm_open()*: open/create SHM object
  - *mmap()*: map SHM object
  - *shm_unlink()*: remove SHM object pathname
- Operations on SHM object via *fd* returned by *shm_open()*:
  - *fstat()*: retrieve info (size, ownership, permissions)
  - *ftruncate()*: change size
  - *fchown()*: *fchmod()*: change ownership, permissions

Opening a POSIX SHM object

- *fd = shm_open(name, flags, mode);*
- Open+create new / open existing SHM object
- *name* has form `/somename`
  - Can be seen in dedicated *tmpfs* at `/dev/shm`
- Returns *fd*, a file descriptor
  - Used by rest of API

Opening a POSIX SHM object

- *fd = shm_open(name, flags, mode);*
- *flags* (analogous to *open()*):
  - `O_CREAT` – create SHM if it doesn’t exist
  - `O_EXCL` – create SHM exclusively
  - `O_RDONLY`, `O_RDWR` – indicates type of access
  - `O_TRUNC` – truncate existing SHM object to zero length
- *mode* sets permissions
  - MBZ if `O_CREAT` not specified

Create and map new SHM object

- Create and map a new SHM object of *size* bytes:

```
fd = shm_open("/myshm",
              O_CREAT | O_EXCL | O_RDWR, 0600);

ftruncate(fd, size);    // Set size of object

addr = mmap(NULL, size,
            PROT_READ | PROT_WRITE,
            MAP_SHARED, fd, 0);
```

Map existing SHM object

- Map an existing SHM object of unknown size:

```
fd = shm_open("/myshm", O_RDWR, 0); // No O_CREAT

// Use object size as length for mmap()
struct stat sb;
fstat(fd, &sb);

addr = mmap(NULL, sb.st_size,
            PROT_READ | PROT_WRITE,
            MAP_SHARED, fd, 0);
```

http://man7.org/tlpi/code/online/dist/pshm/pshm_read.c.html

But...

- How to prevent two process updating shared memory at the same time?

Synchronization

Synchronization

- Synchronize access to a shared resource
  - Shared memory
    - Semaphores
  - File
    - File locks

POSIX semaphores

POSIX semaphores

- Integer maintained inside kernel
- Kernel blocks attempt to decrease value below zero
- Two fundamental operations:
  - *sem_post()*: increment by 1
  - *sem_wait()*: decrement by 1
    - May block

POSIX semaphores

- Semaphore represents a shared resource
- E.g., N shared identical resources ==> initial value of semaphore is N
- Common use: binary value
  - Single resource (e.g., shared memory)

Unnames and named semaphores

- Two types of POSIX semaphore:
  - Unnamed
    - Embedded in shared memory
  - Named
    - Independent, named objects

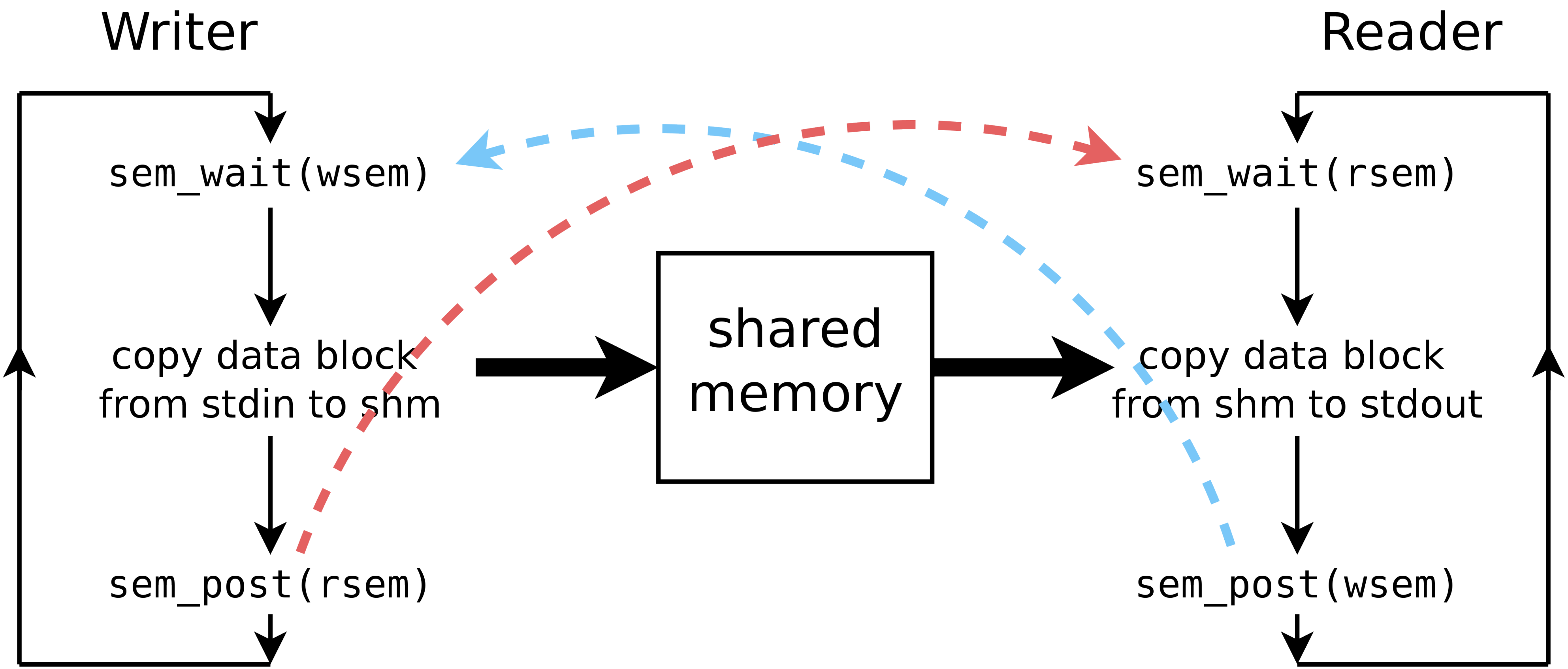
Unnamed semaphores API

- *sem_init(semp, pshared, value)*: initialize semaphore pointed to by *semp* to value
  - *sem_t *semp*
  - *pshared*: 0, thread sharing; != 0, process sharing
- *sem_post(semp)*: add 1 to value
- *sem_wait(semp)*: subtract 1 from value
- *sem_destroy(semp)*: free semaphore, release resources back to system
  - Must be no waiters!

Unnamed semaphores example

- Two processes, writer and reader
- Sending data through POSIX shared memory
- Two unnamed POSIX semaphores inside shm enforce alternating access to shm

Unnamed semaphores example



Header file

```
#define BUF_SIZE 1024

struct shmbuf {   // Buffer in shared memory
    sem_t wsem;          // Writer semaphore
    sem_t rsem;          // Reader semaphore
    int cnt;             // Number of bytes used in 'buf'
    char buf[BUF_SIZE]; // Data being transferred
}
```

Writer

```
fd = shm_open(SHM_PATH, O_CREAT|O_EXCL|O_RDWR, OBJ_PERMS);
ftruncate(fd, sizeof(struct shmbuf));
shmp = mmap(NULL, sizeof(struct shmbuf),
                PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

sem_init(&shmp->rsem, 1, 0);
sem_init(&shmp->wsem, 1, 1);      // Writer gets first turn

for (xfrs = 0, bytes = 0; ; xfrs++, bytes += shmp->cnt) {
    sem_wait(&shmp->wsem);        // Wait for our turn
    shmp->cnt = read(STDIN_FILENO, shmp->buf, BUF_SIZE);
    sem_post(&shmp->rsem);        // Give reader a turn

    if (shmp->cnt == 0)           // EOF on stdin?
        break;
}
sem_wait(&shmp->wsem);      // Wait for reader to finish

// Clean up
```

Reader

```
fd = shm_open(SHM_PATH, O_RDWR, 0);
shmp = mmap(NULL, sizeof(struct shmbuf),
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

for (xfrs = 0, bytes = 0; ; xfrs++) {
    sem_wait(&shmp->rsem);    // Wait for our turn */

    if (shmp->cnt == 0)       // Writer encountered EOF */
        break;
    bytes += shmp->cnt;

    write(STDOUT_FILENO, shmp->buf, shmp->cnt) != shmp->cnt);
    sem_post(&shmp->wsem);    // Give writer a turn */
}

sem_post(&shmp->wsem);   // Let writer know we're finished
```

Named semaphores API

- Object management
  - *sem_open()*: open/create semaphore
  - *sem_unlink()*: remove semaphore pathname

Opening a POSIX semaphore

- *semp = sem_open(name, flags [, mode, value]);*
- Open+create new / open existing semaphore
- *name* has form `/somename`
  - Can be seen in dedicated *tmpfs* at `/dev/shm`
- Returns *sem_t **, reference to semaphore
  - Used by rest of API

Opening a POSIX semaphore

- *semp = sem_open(name, flags [, mode, value]);*
- *flags* (analogous to *open()*):
  - `O_CREAT` – create SHM if it doesn't exist
  - `O_EXCL` – create SHM exclusively
- If creating new semaphore:
  - *mode* sets permissions
  - *value* initializes semaphore

Sockets

Sockets

- Big topic
- Just a high-level view
- Some notable features when running as IPC

Sockets

- “A socket is endpoint of communication...”
  - ... you need two of them
- Bidirectional
- Created via:
  - *fd = socket(domain, type, protocol);*

Socket domains

- Each socket exists in a *domain*
- Domain determines:
  - Method of identifying socket (“address”)
  - “Range” of communication
    - Processes on a single host
    - Across a network

Common socket domains

- UNIX domain (`AF_UNIX`)
  - Communication on single host
  - Address == file system pathname
- IPv4 domain (`AF_INET`)
  - Communication on IPv4 network
  - Address = IPv4 address (32 bit) + port number
- IPv6 domain (`AF_INET6`)
  - Communication on IPv6 network
  - Address = IPv6 address (128 bit) + port number

Socket type

- Determines semantics of communication
- Two main types available in all domains:
  - Stream (`SOCK_STREAM`)
  - Datagram (`SOCK_DGRAM`)
- UNIX domain (on Linux) also provides
  - Sequential packet (`SOCK_SEQPACKET`)

Stream sockets

- SOCK_STREAM
- Byte stream
- Connection-oriented
  - Like a two-party phone call
- Reliable == data arrives “intact” or not at all
- Intact:
  - In order
  - Unduplicated
- Internet domain: TCP protocol

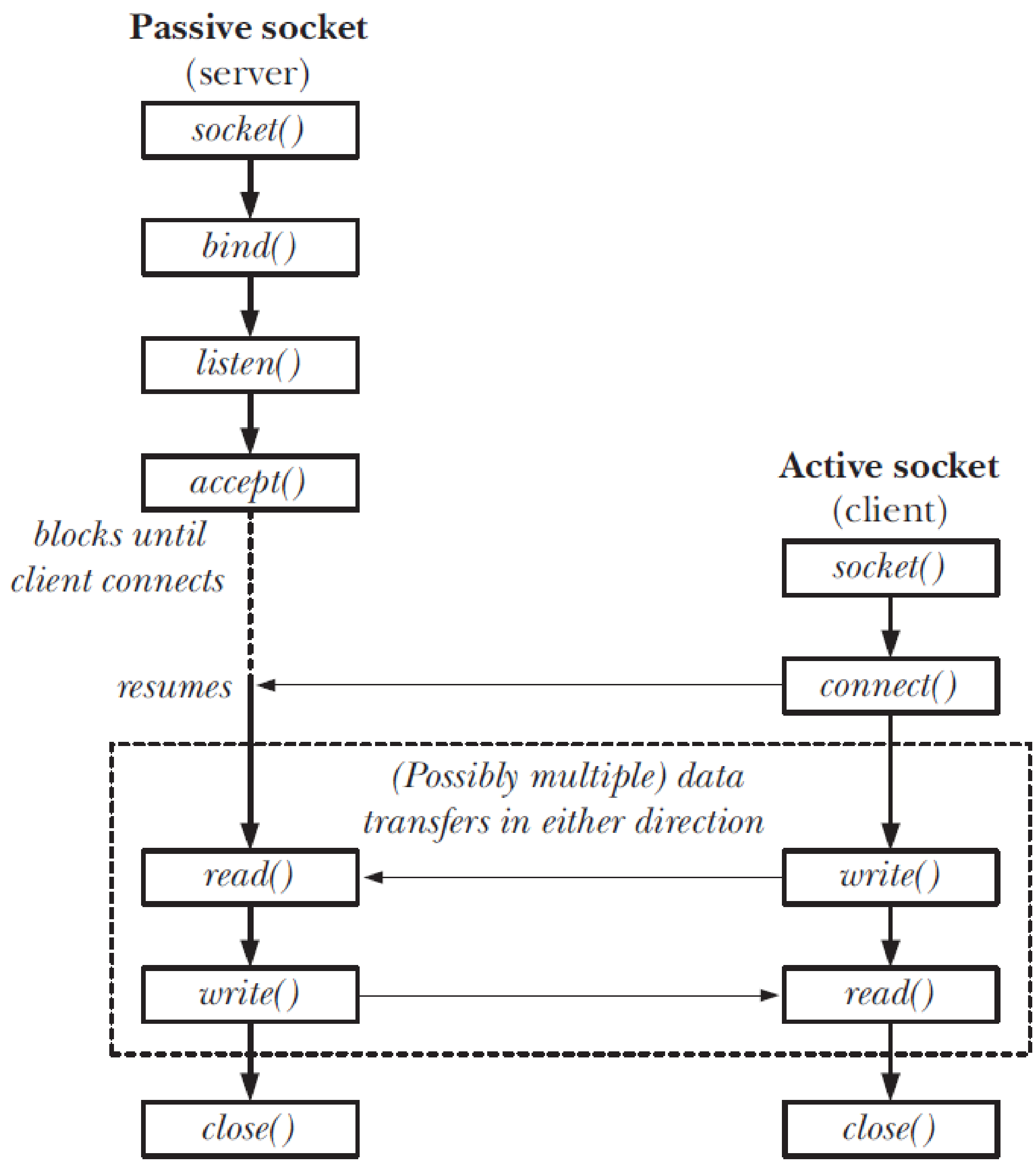
Datagram sockets

- `SOCK_DGRAM`
- Message-oriented
- Connection-less
  - Like a postal system
- Unreliable; messages may arrive:
  - Duplicated
  - Out of order
  - Not at all
- Internet domain: UDP protocol

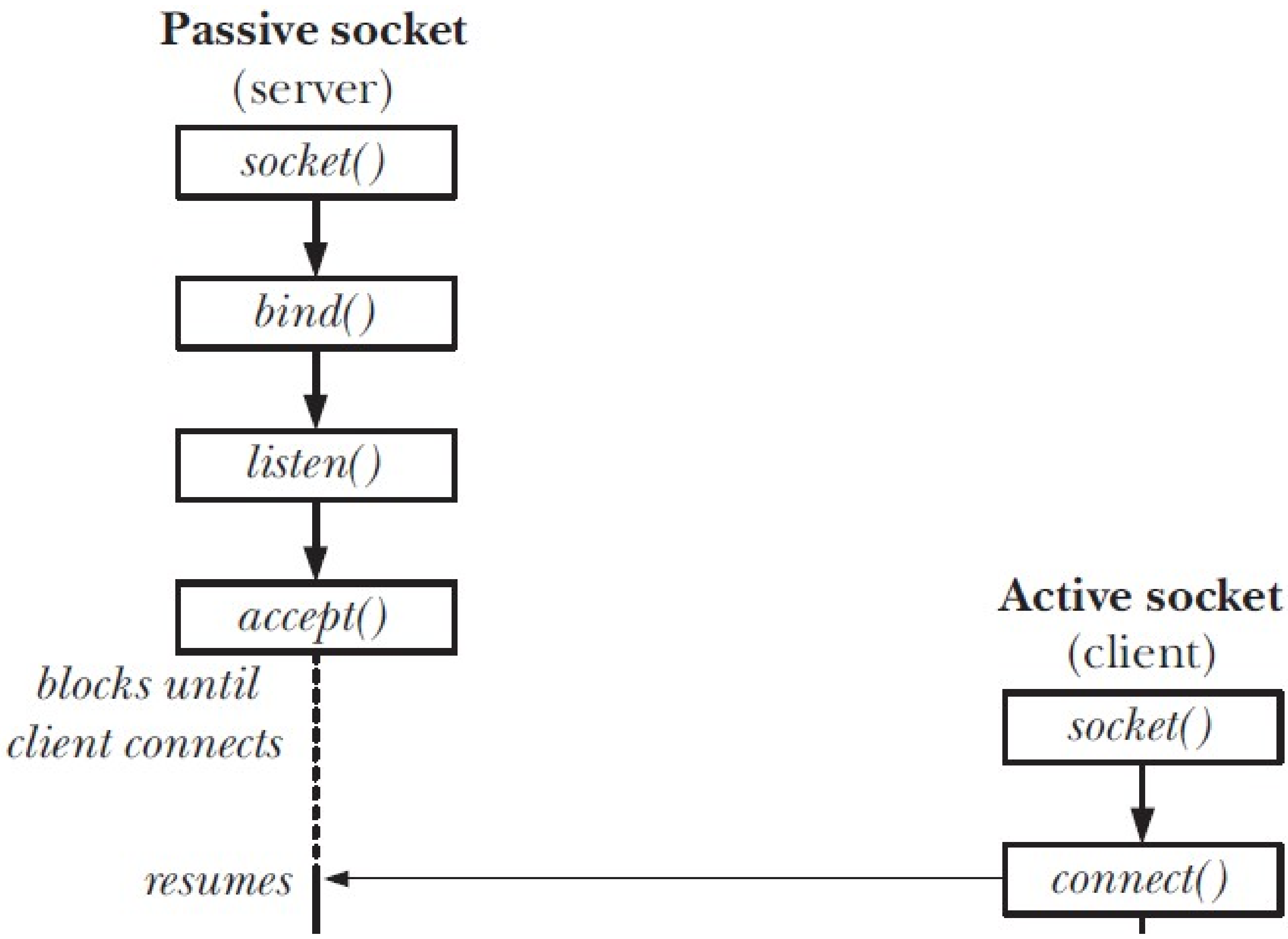
Sequential packet sockets

- `SOCK_SEQPACKET`
- Midway between stream and datagram sockets
  - Message-oriented
  - Connection-oriented
  - Reliable
- UNIX domain
  - In INET domain, only with SCTP protocol

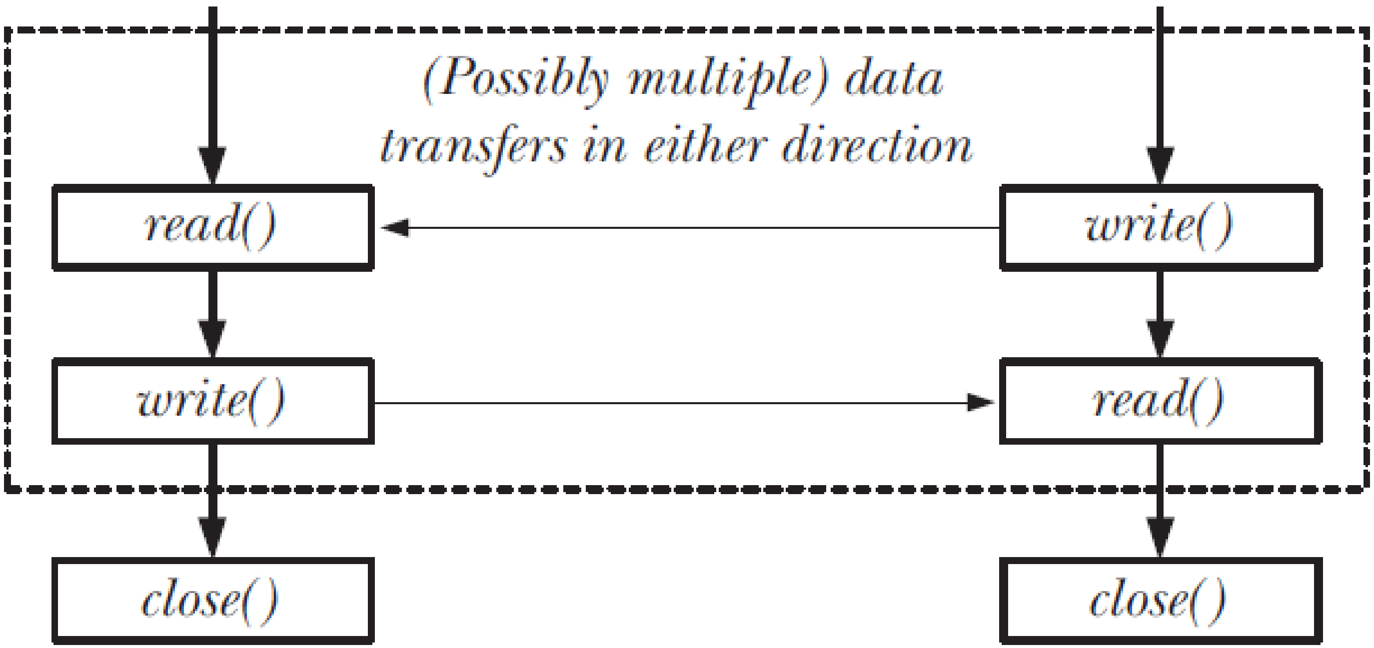
Stream sockets API



Stream sockets API



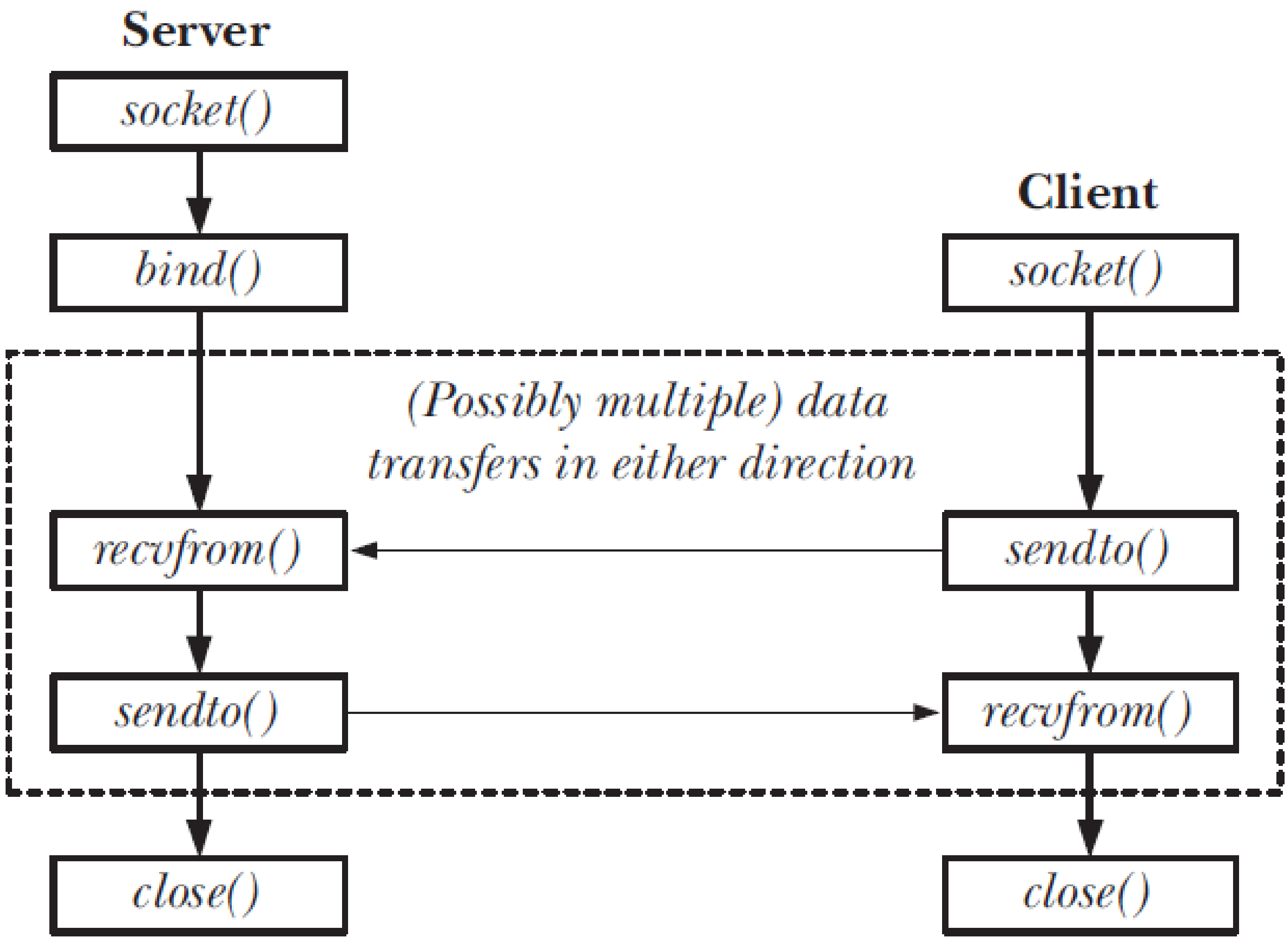
Stream sockets API



Stream sockets API

- *socket(SOCK_STREAM)* – create a socket
- Passive socket:
  - *bind()* – assign address to socket
  - *listen()* – specify size of incoming connection queue
  - *accept()* – accept connection off incoming queue
- Active socket:
  - *connect()* – connect to passive socket
- I/O:
  - *write()*, *read()*, *close()*
  - *send()*, *recv()* – socket specific flags

Datagram sockets API



Datagram sockets API

- *socket(SOCK_DGRAM)* – create socket
- *bind()* – assign address to socket
- *sendto()* – send datagram to an address
- *recvfrom()* – receive datagram and address of sender
- *close()*

Sockets: noteworthy points

- Bidirectional communication
- UNIX domain datagram sockets **are** reliable
- UNIX domain sockets can pass file descriptors
- Internet domain sockets are only method for network communication
- UDP sockets allow broadcast / multicast of datagrams
- *socketpair()*
  - UNIX domain
  - Bidirectional pipe

Other criteria affecting choice of an IPC mechanism

Criteria for selecting an IPC mechanism

- The obvious
  - Consistency with application design
  - Functionality
- Let’s look at some other criteria

IPC IDs and handles

- Each IPC object has:
  - ID – the method used to identify an object
  - Handle – the reference used in a process to access an open object

IPC IDs and handles

Facility type	**Name used to identify object**	**Handle used to refer to object in programs**
Pipe FIFO	no name pathname	file descriptor file descriptor
UNIX domain socket Internet domain socket	pathname IP address + port number	file descriptor file descriptor
System V message queue System V semaphore System V shared memory	System V IPC key System V IPC key System V IPC key	System V IPC identifier System V IPC identifier System V IPC identifier
POSIX message queue POSIX named semaphore POSIX unnamed semaphore POSIX shared memory	POSIX IPC pathname POSIX IPC pathname no name POSIX IPC pathname	*mqd_t* (message queue descriptor) *sem_t* * (semaphore pointer) *sem_t* * (semaphore pointer) file descriptor
Anonymous mapping Memory-mapped file	no name pathname	none file descriptor
flock() lock *fcntl()* lock	pathname pathname	file descriptor file descriptor

File descriptor handles

- Some handles are file descriptors
- File descriptors can be multiplexed via *poll()* / *select()* /*epoll*
  - Sockets, pipes, FIFOs
  - On Linux, POSIX MQ descriptors are file descriptors
  - One good reason to avoid System V message queues

IPC access permissions

- How is access to IPC controlled?
- Possibilities
  - UID/GID + permissions mask
  - Related processes (via *fork()*)
  - Other
    - e.g., Internet domain: application-determined

IPC access permissions

Facility type	**Accessibility**
Pipe	only by related processes
FIFO	permissions mask
UNIX domain socket	permissions mask
Internet domain socket	by any process
System V message queue	permissions mask
System V semaphore	permissions mask
System V shared memory	permissions mask
POSIX message queue	permissions mask
POSIX named semaphore	permissions mask
POSIX unnamed semaphore	permissions of underlying memory
POSIX shared memory	permissions mask
Anonymous mapping	only by related processes
Memory-mapped file	permissions mask
flock() file lock	*open()* of file
fcntl() file lock	*open()* of file

IPC object persistence

- What is the lifetime of an IPC object?
  - **Process**: only as long as held open by at least one process
  - **Kernel**: until next reboot
    - State persists even if no connected process
  - **Filesystem**: persists across reboot
    - Memory mapped file

IPC object persistence

Facility type	Persistence
Pipe	process
FIFO	process
UNIX domain socket	process
Internet domain socket	process
System V message queue	kernel
System V semaphore	kernel
System V shared memory	kernel
POSIX message queue	kernel
POSIX named semaphore	kernel
POSIX unnamed semaphore	depends
POSIX shared memory	kernel
Anonymous mapping	process
Memory-mapped file	file system
flock() file lock	process
fcntl() file lock	process

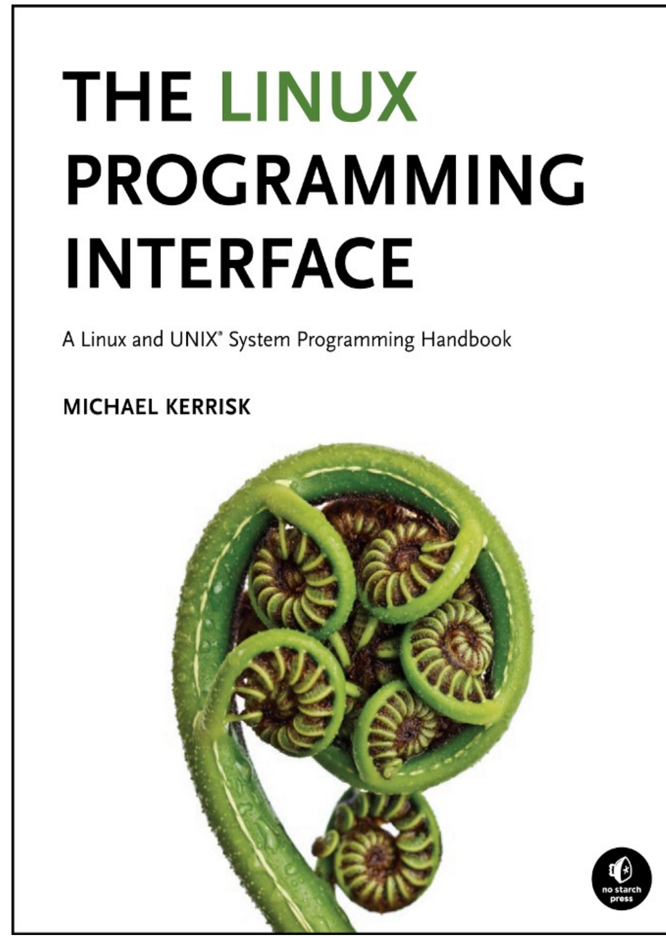
Thanks! And Questions

(slides up soon at http://man7.org/conf/)

Michael Kerrisk
mtk@man7.org
http://man7.org/tlpi

LWN.net
mtk@lwn.net
http://lwn.net/

Linux *man-pages* project
mtk.manpages@gmail.com
http://www.kernel.org/doc/man-pages/



(No Starch Press, 2010)

Mamaku (Black Tree Fern) image (c) Rob Suisted
naturespic.com