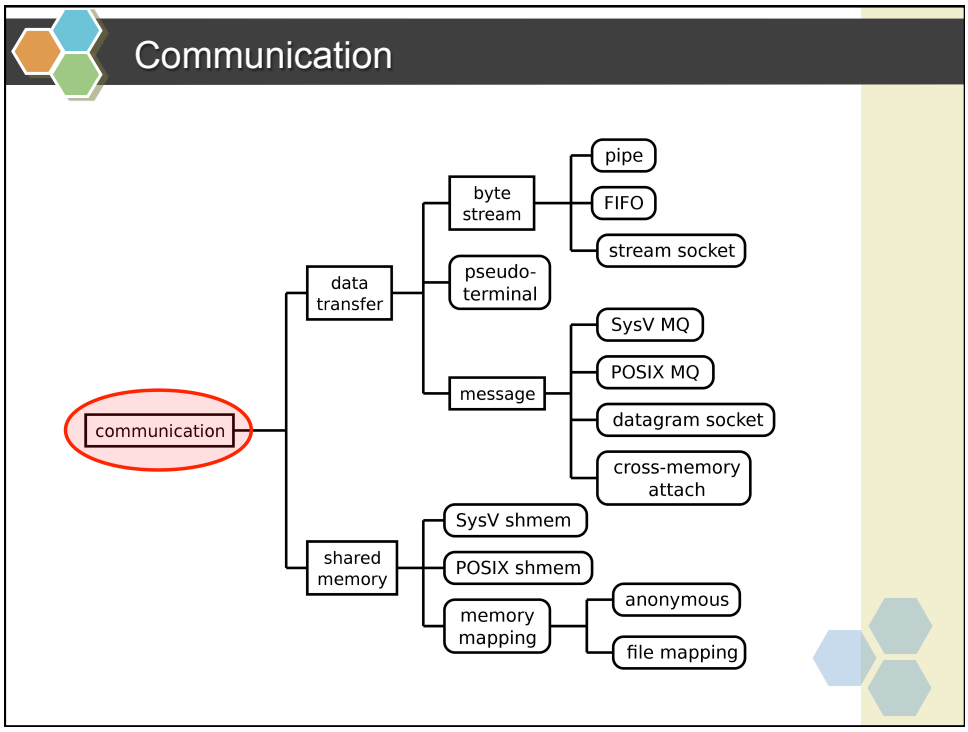
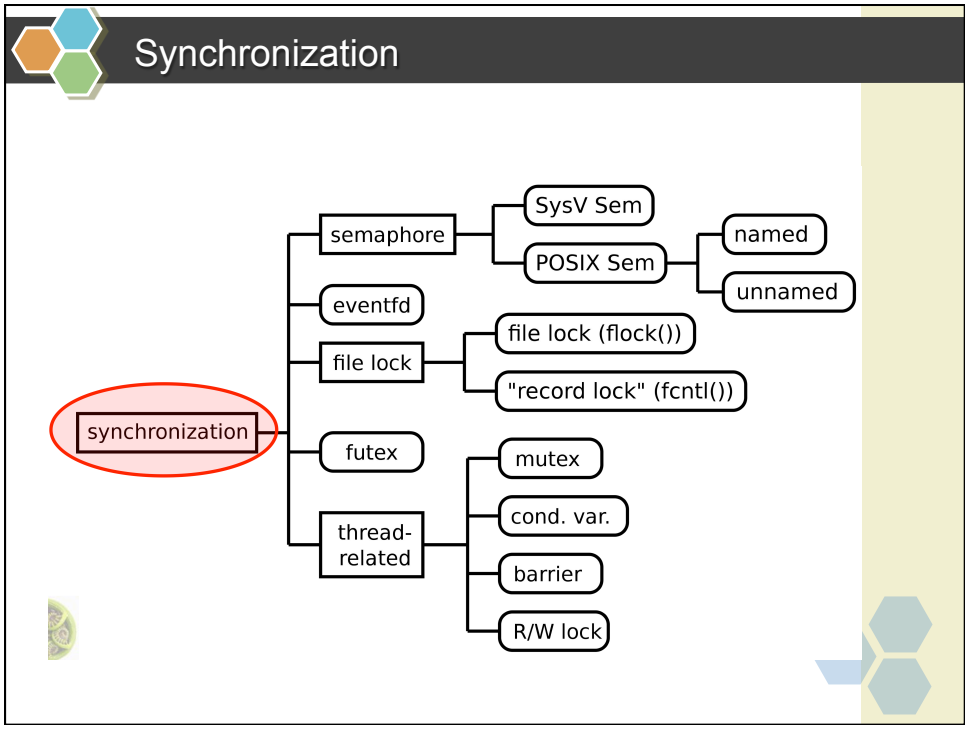
Unix Programming

Inter-Process Communication

A lot of IPC

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


Communication
communication
data transfer
byte stream
pipe
FIFO
stream socket
pseudo-terminal
message
SysV MQ
POSIX MQ
datagram socket
cross-memory attach
shared memory
SysV shmem
POSIX shmem
memory mapping
anonymous
file mapping

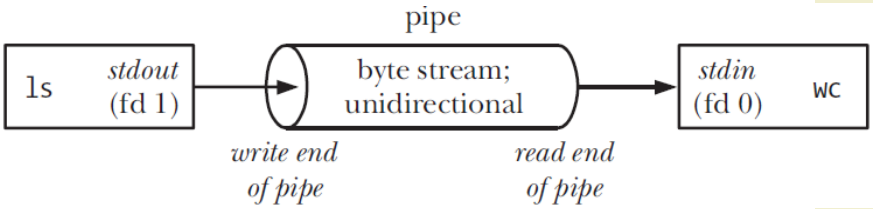

Synchronization
synchronization
semaphore
SysV Sem
POSIX Sem
named
unnamed
eventfd
file lock
file lock (flock())
"record lock" (fcntl())
futex
thread-related
mutex
cond. var.
barrier
R/W lock

Outline

- **Communication Techniques**
  - **Pipes**
  - FIFO (named pipes)
  - POSIX Message Queues
  - Shared Memory
  - Shared Anonymous Mapping
  - Shared File Mapping
  - POSIX Shared Memory
- **Synchronization**
  - POSIX Semaphores

Pipes

- **ls | wc -l**



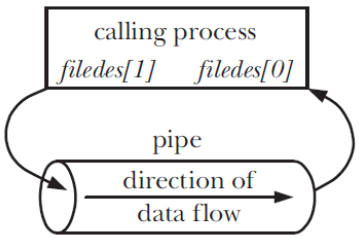
Pipes

- 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[2];
pipe(filedes);
...
write(filedes[1], buf, count);
read(filedes[0], buf, count);
```



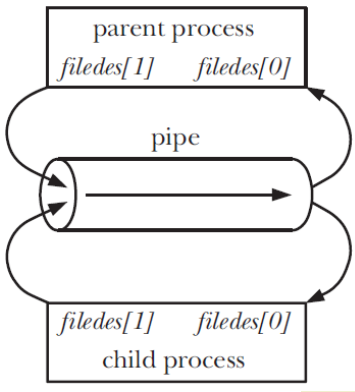
Sharing a pipe

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

Sharing a pipe

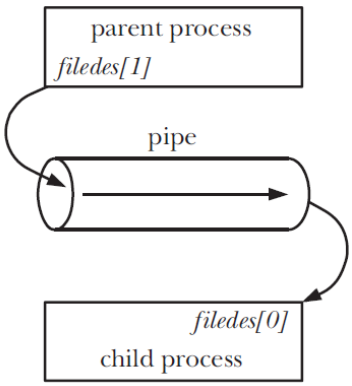
```
int filedes[2];
pipe(filedes);
child_pid = fork();
```

- 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);
switch (fork()) {
case 0:
   close(pfd[1]);
   for (;;) {
/* Create the pipe */
/* Child  - reads from pipe */
/* Write end is unused */
/* 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)

Outline

- **Communication Techniques**
  - Pipes
  - **FIFO (named pipes)**
  - POSIX Message Queues
  - Shared Memory
  - Shared Anonymous Mapping
  - Shared File Mapping
  - POSIX Shared Memory
- **Synchronization**
  - POSIX Semaphores

FIFO (named pipe)

- 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

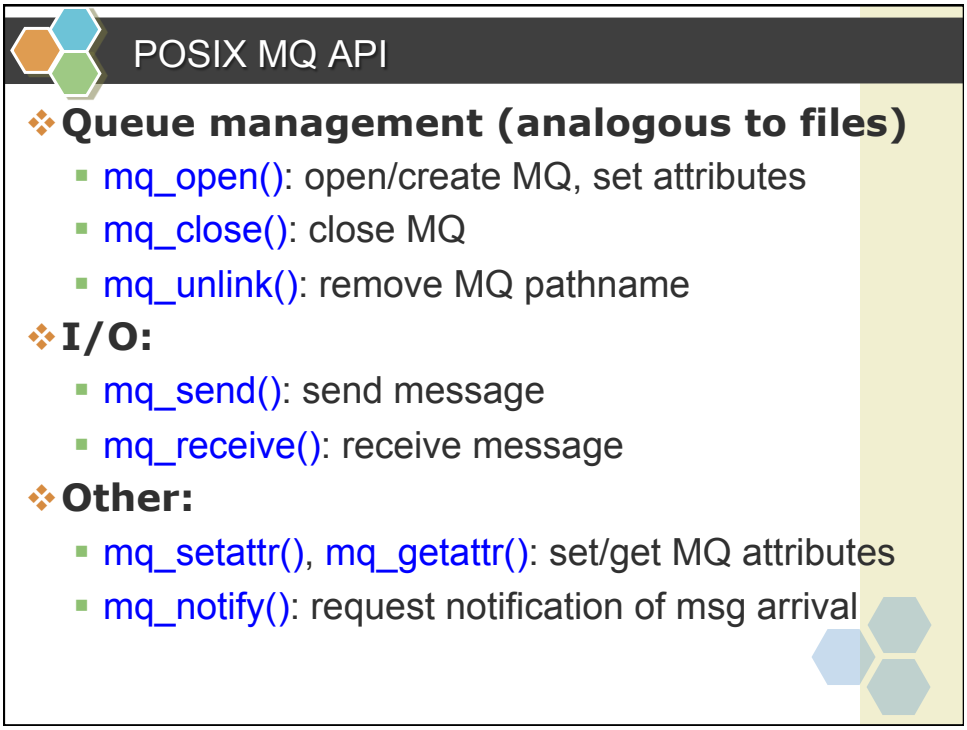
- open(pathname, O_RDONLY)
  - Open read end
- open(pathname, O_WRONLY)
  - Open write end
- open() locks until other end is opened
  - Opens are synchronized
  - open(pathname, O_RDONLY | O_NONBLOCK) can be useful

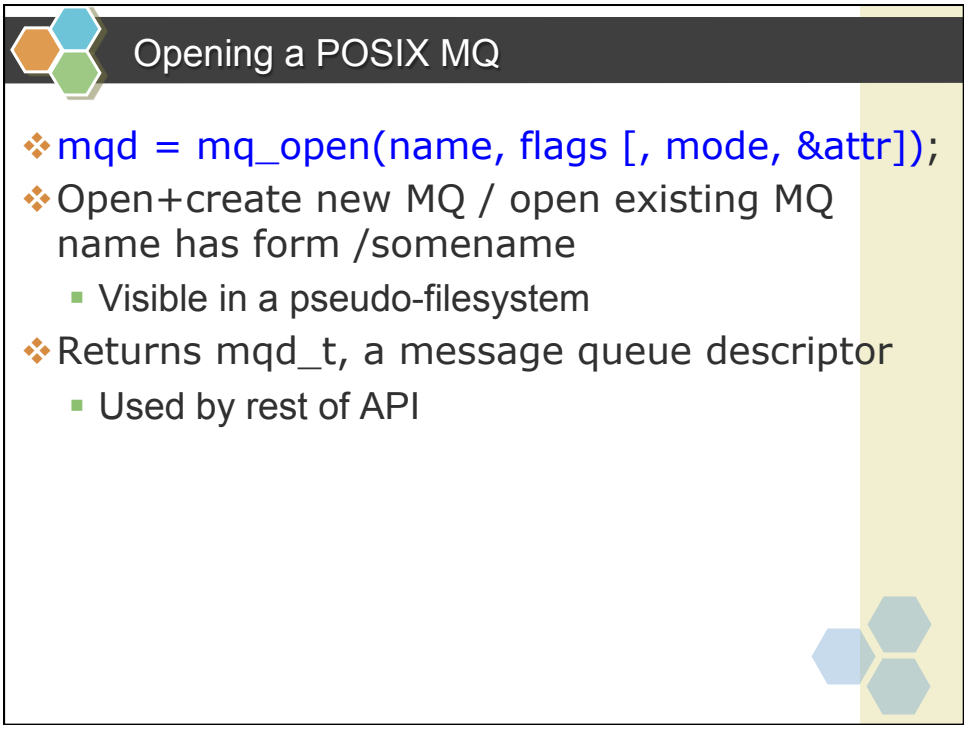
Outline

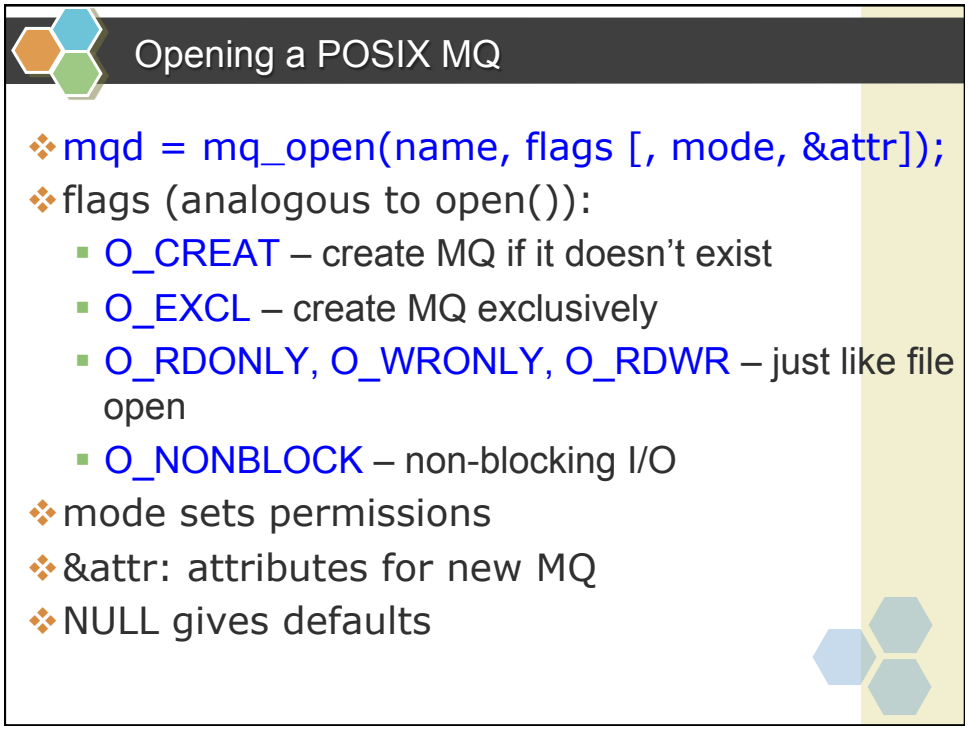
- **Communication Techniques**
  - Pipes
  - FIFO (named pipes)
  - **POSIX Message Queues**
  - Shared Memory
  - Shared Anonymous Mapping
  - Shared File Mapping
  - POSIX Shared Memory
- **Synchronization**
  - POSIX Semaphores

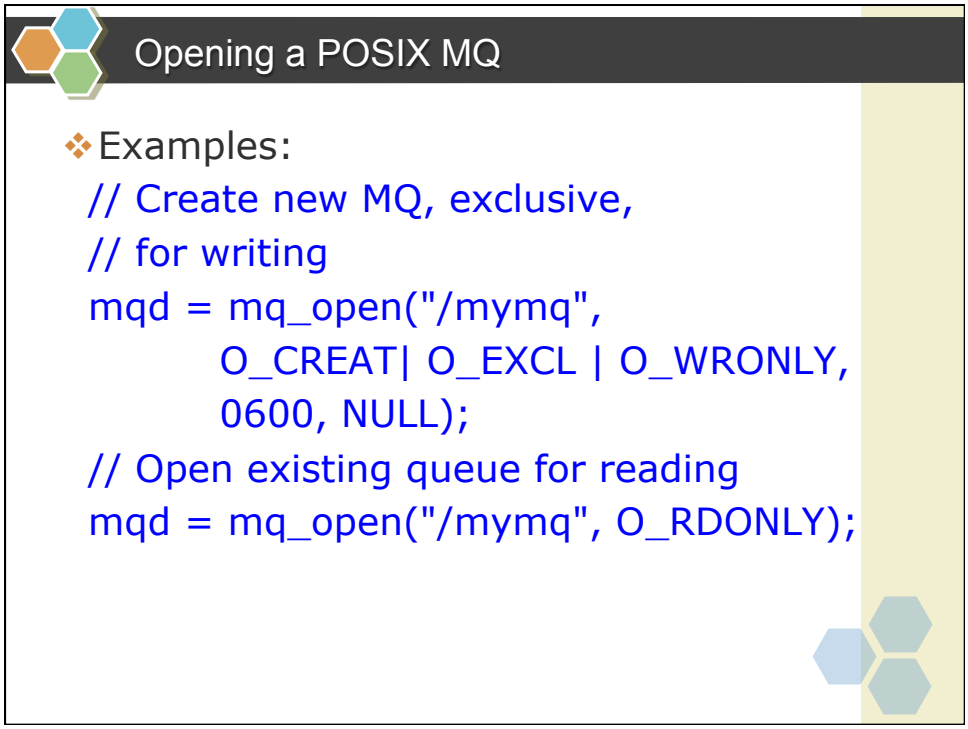
Highlights of POSIX MQs

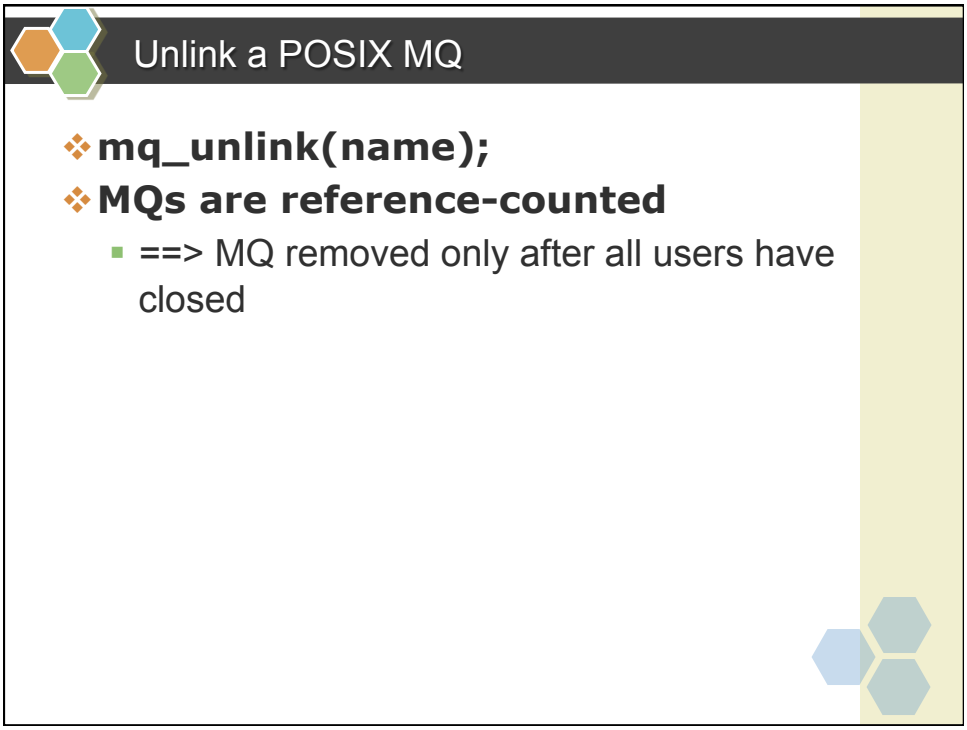
- **Message-oriented communication**
  - Receiver reads messages one at a time
    - No partial or multiple message reads
  - Unlike pipes, multiple readers/writers can be useful
- **Messages have priorities**
  - Delivered in priority order
- **Message notification feature**

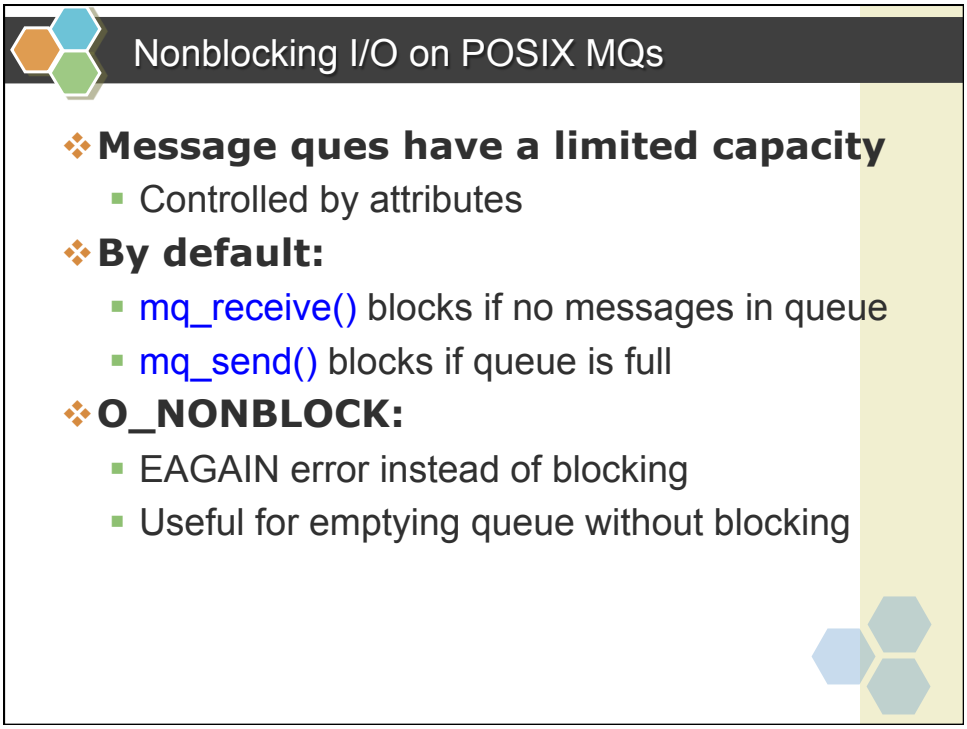

POSIX MQ API
Queue management (analogous to files)
mq_open(): open/create MQ, set attributes
mq_close(): close MQ
mq_unlink(): remove MQ pathname
I/O:
mq_send(): send message
mq_receive(): receive message
Other:
mq_setattr(), mq_getattr(): set/get MQ attributes
mq_notify(): request notification of msg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

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);
```

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
```

Outline

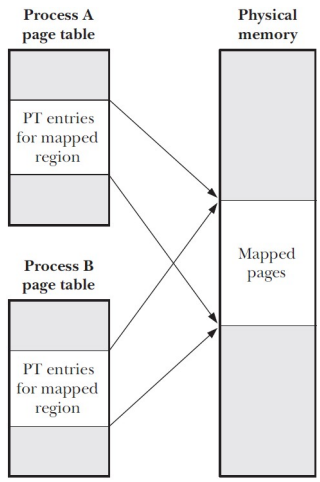
- **Communication Techniques**
  - Pipes
  - FIFO (named pipes)
  - POSIX Message Queues
  - **Shared Memory**
  - Shared Anonymous Mapping
  - Shared File Mapping
  - POSIX Shared Memory
- **Synchronization**
  - POSIX Semaphores

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:

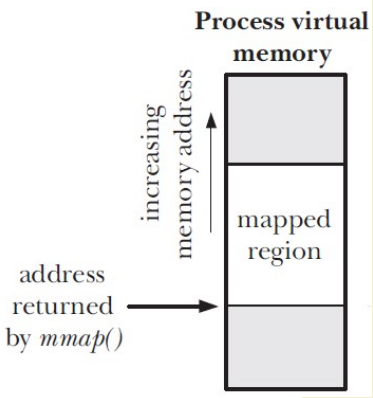
```
void *mmap(void *daddr, size_t len, int prot, int
flags, int fd, off_t offset);
```

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



Outline

- **Communication Techniques**
  - Pipes
  - FIFO (named pipes)
  - POSIX Message Queues
  - Shared Memory
  - Shared Anonymous Mapping
  - Shared File Mapping
  - POSIX Shared Memory
- **Synchronization**
  - POSIX Semaphores

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

Shared anonymous mapping

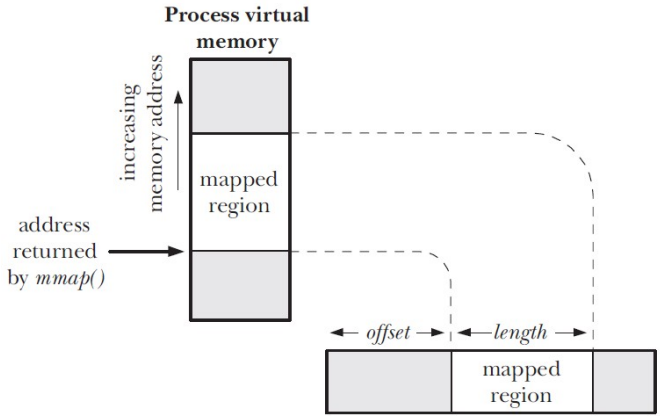
```
addr = mmap(NULL, length,
          PROT_READ | PROT_WRITE,
          MAP_SHARED |MAP_ANONYMOUS,
          -1, 0);
pid = fork();
```

Outline

- **Communication Techniques**
  - Pipes
  - FIFO (named pipes)
  - POSIX Message Queues
  - Shared Memory
  - Shared Anonymous Mapping
  - **Shared File Mapping**
  - POSIX Shared Memory
- **Synchronization**
  - POSIX Semaphores

Shared file mapping

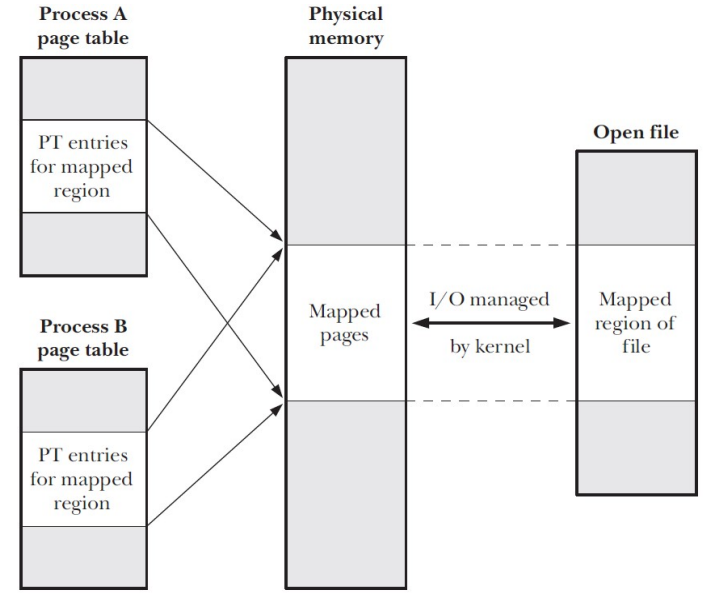
- Share memory between unrelated processes, backed by file
- fd = open(...); addr = mmap(..., fd, offset);



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**

Outline

- **Communication Techniques**
  - Pipes
  - FIFO (named pipes)
  - POSIX Message Queues
  - Shared Memory
  - Shared Anonymous Mapping
  - Shared File Mapping
  - POSIX Shared Memory
- **Synchronization**
  - POSIX Semaphores

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);
```

Posix Shared memory - shmget

- shmget function is used to create a new shared memory segment or to locate an existing one based on a key

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int flag);
```

  - Returns: shared memory ID if OK, -1 on error
  - key is of type key_t or IPC_PRIVATE.
    - key is the numeric key to be assigned to the returned shared memory segment.
    - The IPC_PRIVATE key is used in a parent/child relationship.
  - size parameter is the size of the shared memory segment in bytes.
    - If a new segment is being created, we must specify its size.
    - If we are referencing an existing segment , we can specify size as 0.
  - Flag parameter specifies the mode (creating, reading, writing): IPC_CREAT | S_IRUSR | S_IWUSR for reading and writing. Note that, IPC_PRIVATE guarantees that the process creates a new IPC structure.

Posix Shared memory - shmat

- Once a shared memory segment has been created, a process attaches it to its address space by calling shmat.

```
#include <sys/shm.h>
void *shmat(int shmid, const void *addr, int flag);
```

  - shmid: shared memory ID i.e., the return value of shmget
  - addr: address in the calling process at which the segment is attached depends on the addr argument. Set to NULL, so the segment is attached at the first available address selected by the kernel.
  - flag : If the SHM_RDONLY bit is specified in flag, the segment is attached read-only. Otherwise (0, or NULL), the segment is attached read/write.
  - Returned value: the address at which the segment is attached, or -1 on error occur.
- kernel maintains a structure for each shared segment (e.g., number of current attaches, pid of creator, size of segment in bytes, last attach time, etc.)

Posix Shared Memory

- When we're done with a shared memory segment, we call shmdt to detach it.
- shmdt does not remove the identifier and its associated data structure from the system.

```
#include <sys/shm.h>
int shmdt(void *addr);
```

  - addr: is the value that was returned by a previous call to shmat. Returned value: 0 if OK, -1 on error
- The identifier remains in existence until some process specifically removes it by calling shmctl with a command of IPC_RMID.

```
#include <sys/shm.h>
int shmctl(int shmid, IPC_RMID, NULL);
```

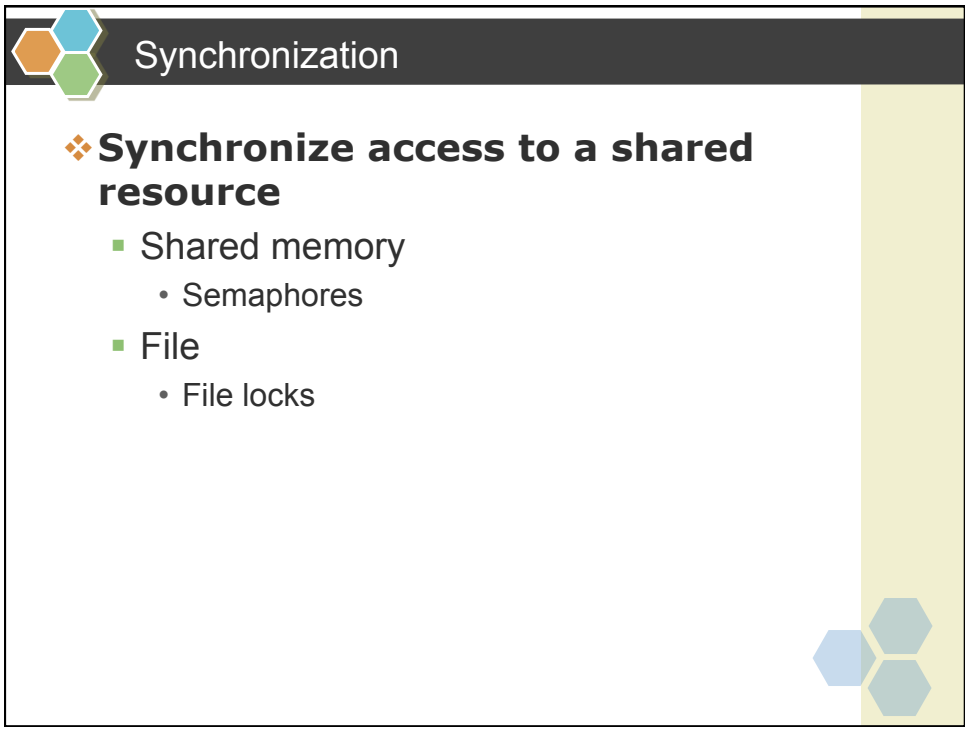
-

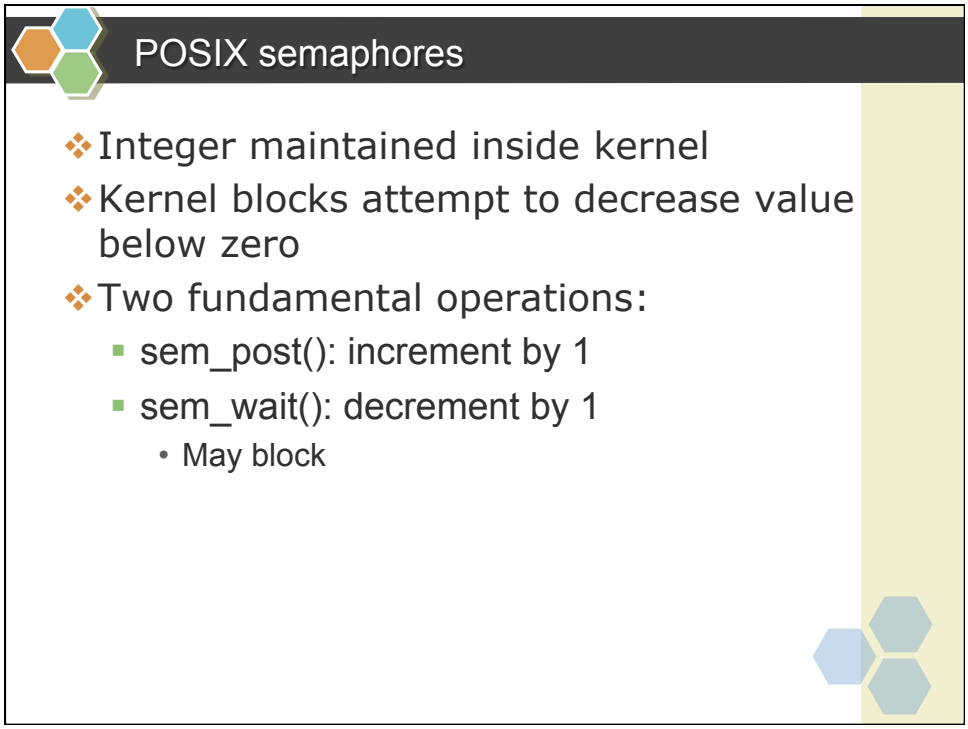
But …

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

Outline

- **Communication Techniques**
  - Pipes
  - FIFO (named pipes)
  - POSIX Message Queues
  - Shared Memory
  - Shared Anonymous Mapping
  - Shared File Mapping
  - POSIX Shared Memory
- **Synchronization**
  - **POSIX Semaphores**


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

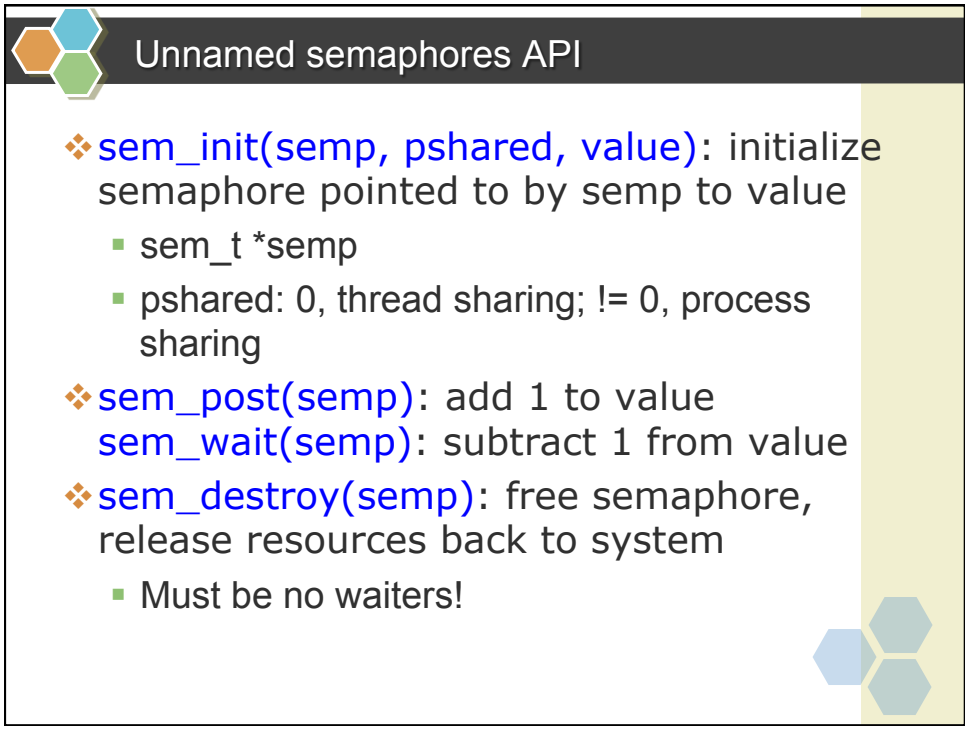

POSIX semaphores
Integer maintained inside kernel
Kernel blocks attempt to decrease value below zero
Two fundamental operations:
sem_post(): increment by 1
sem_wait(): decrement by 1
May block

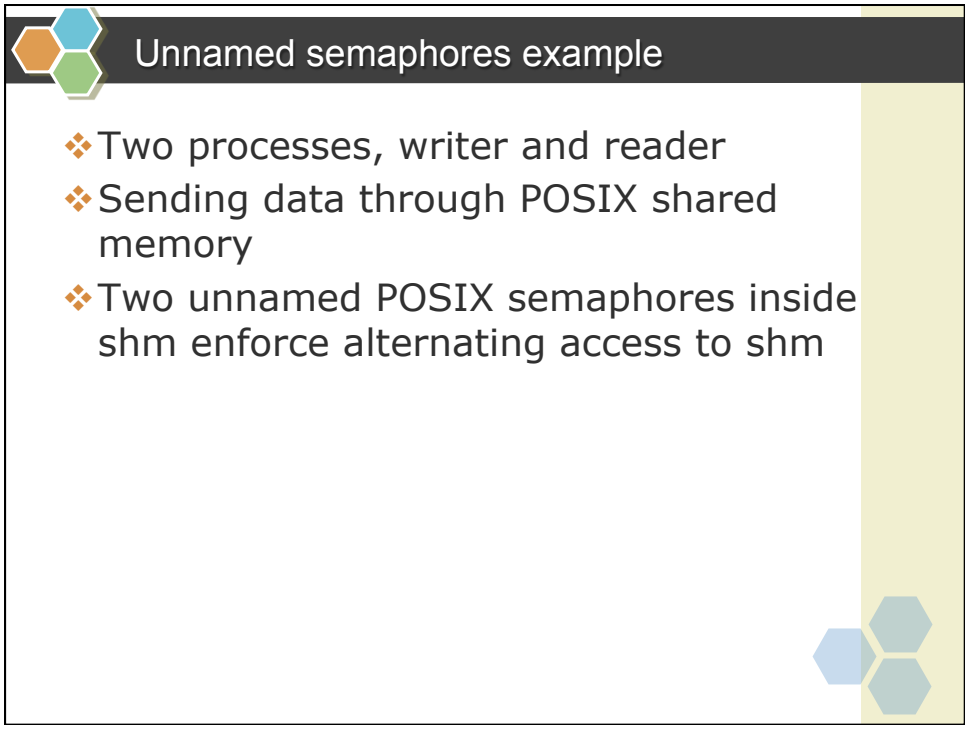
POSIX semaphores

- **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)

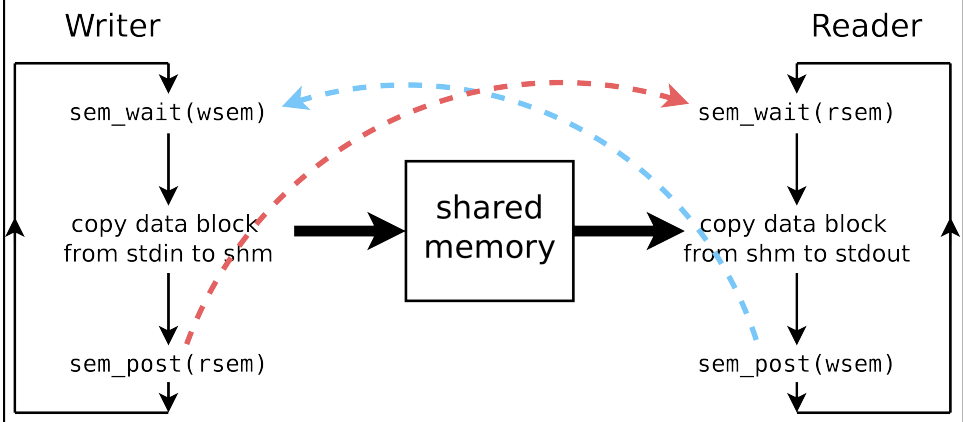
Unnamed and named semaphores

- **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
```

Reader

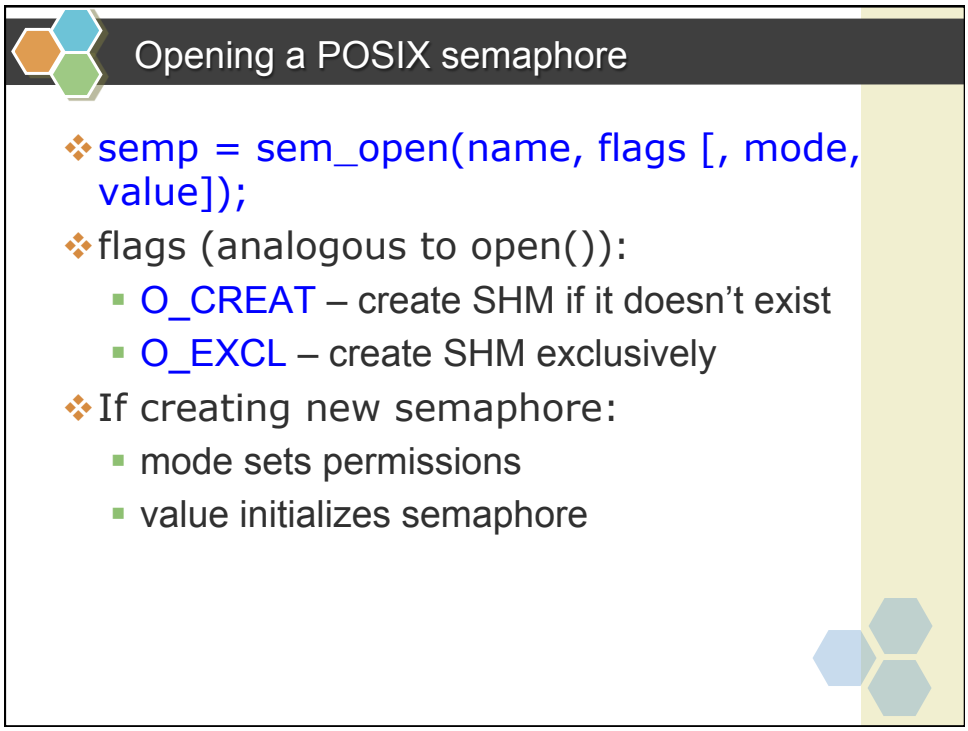
```
fd = shm_open(SHM_PATH, O_RDWR, 0);
shmp = mmap(NULL, sizeof(struct shmbuf),
          PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
for (xfrs = 0, bytes = 0; ; xfrs++) {
    sem_wait(&shmp->rsem);    // Wait for our turn */
    if (shmp->cnt == 0)       // Writer encountered EOF */
        break;
    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