Interprocess Communication

Interprocess Communciation

- What is IPC?
  - Mechanisms to transfer data between processes
- Why is it needed?
  - Not all important procedures can be easily built in a single process

Interprocess Communication

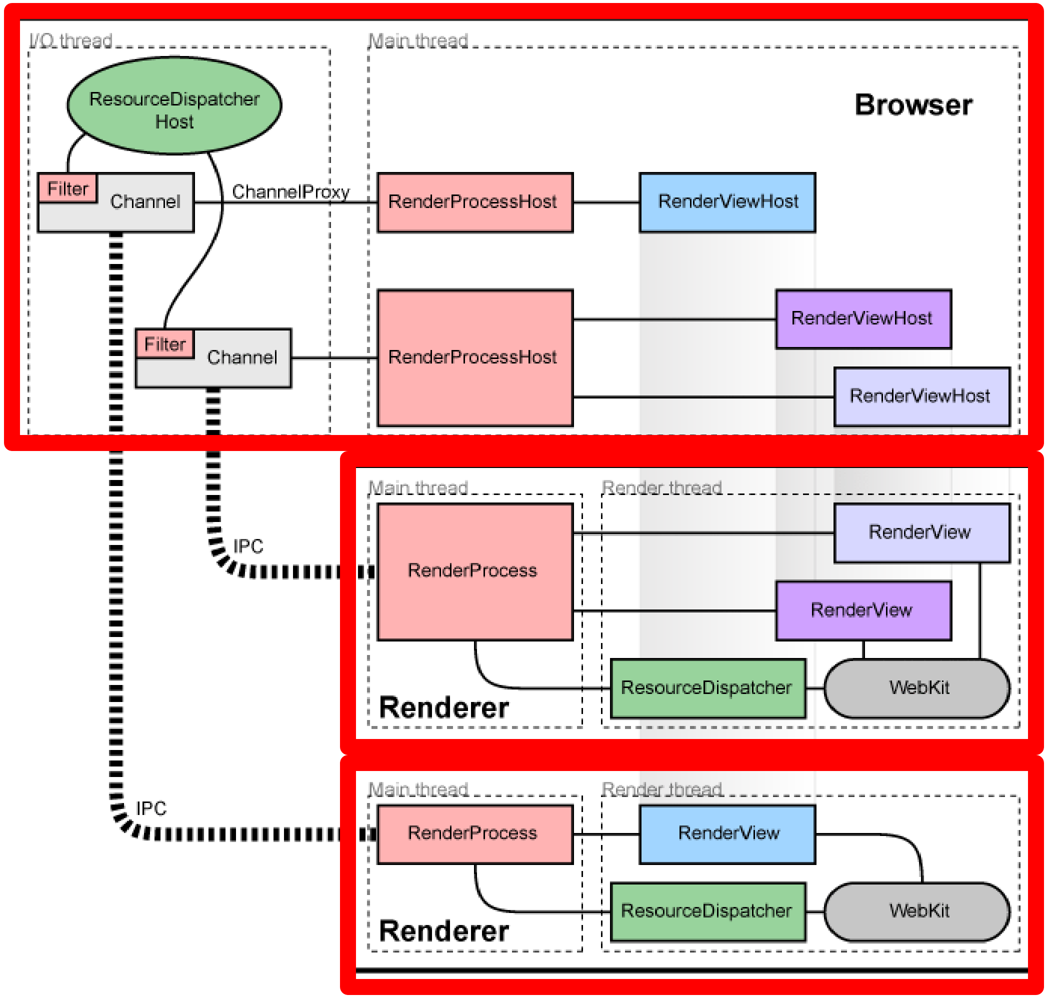
- Cooperating processes
  - Can affect or be affected by other processes, including sharing data
    - Just like cooperating threads!
  - Benefits
    - Information sharing
    - Computation speedup
    - Modularity
    - Convenience

Interprocess Communication

- Can you think of a common use of IPC?

- Can you think of any large applications that use IPC?

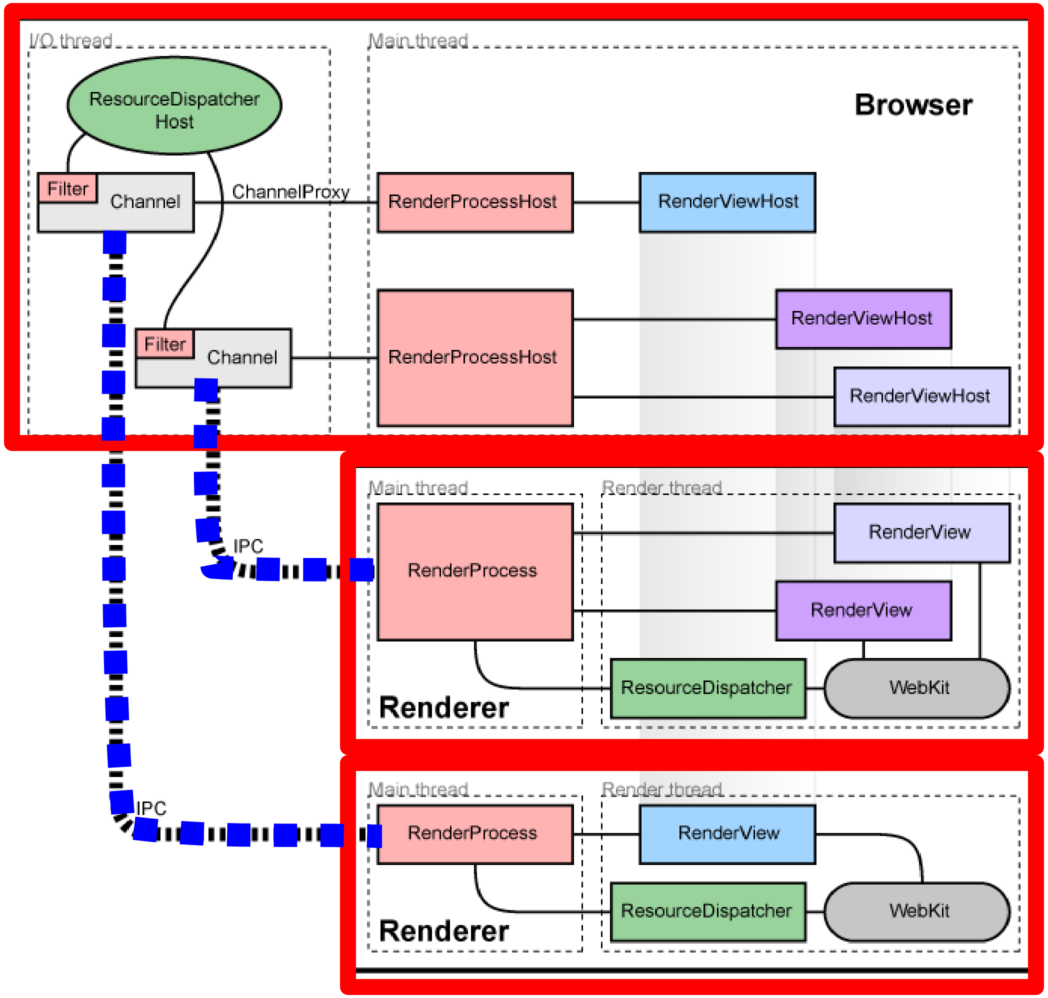
Google Chrome architecture (figure borrowed from Google)



- Separate processes for browser tabs to protect the overall application from bugs and glitches in the rendering engine
- Restricted access from each rendering engine process to others and to the rest of the system

https://sites.google.com/a/chromium.org/dev/developers/design-documents/multi-process-architecture

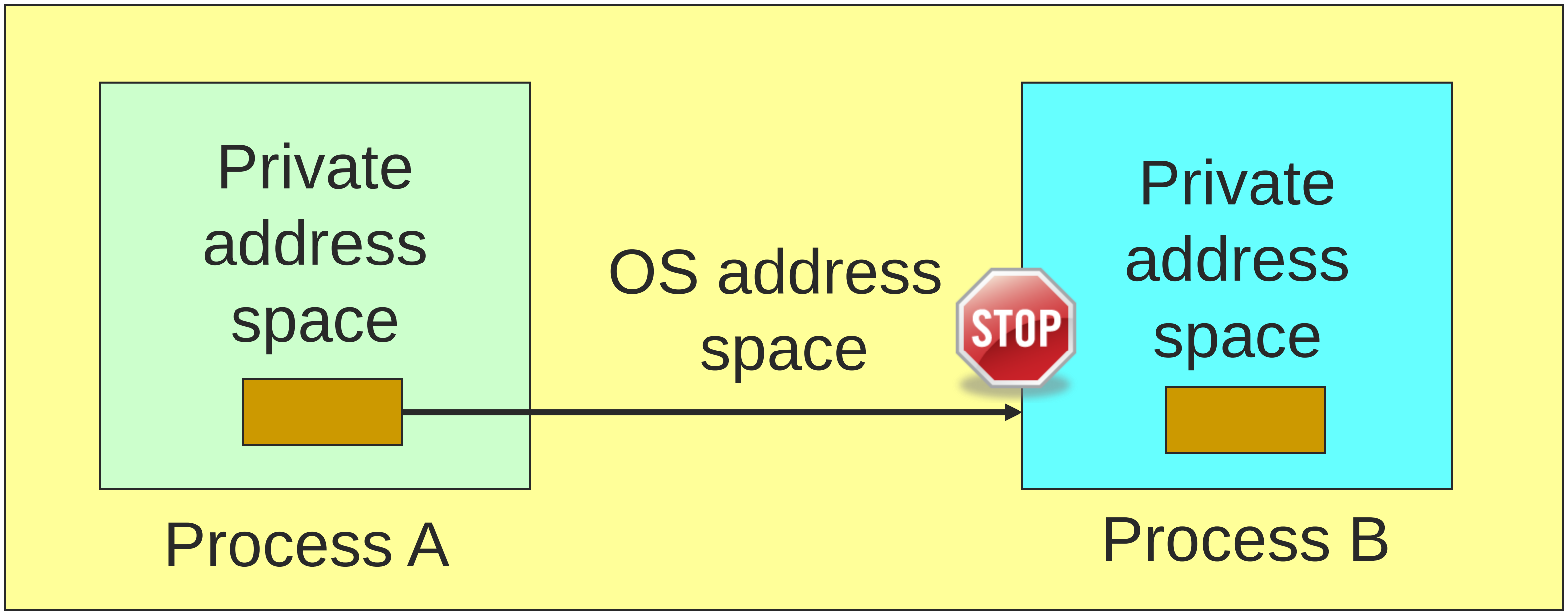
Google Chrome architecture (figure borrowed from Google)



- A named pipe is allocated for each renderer process for communication with the browser process
- Pipes are used in asynchronous mode to ensure that neither end is blocked waiting for the other

https://sites.google.com/a/chromium.org/dev/developers/design-documents/multi-process-architecture

IPC Communications Model



- Each process has a private address space
- No process can write to another process's space
- How can we get data from process A to process B?

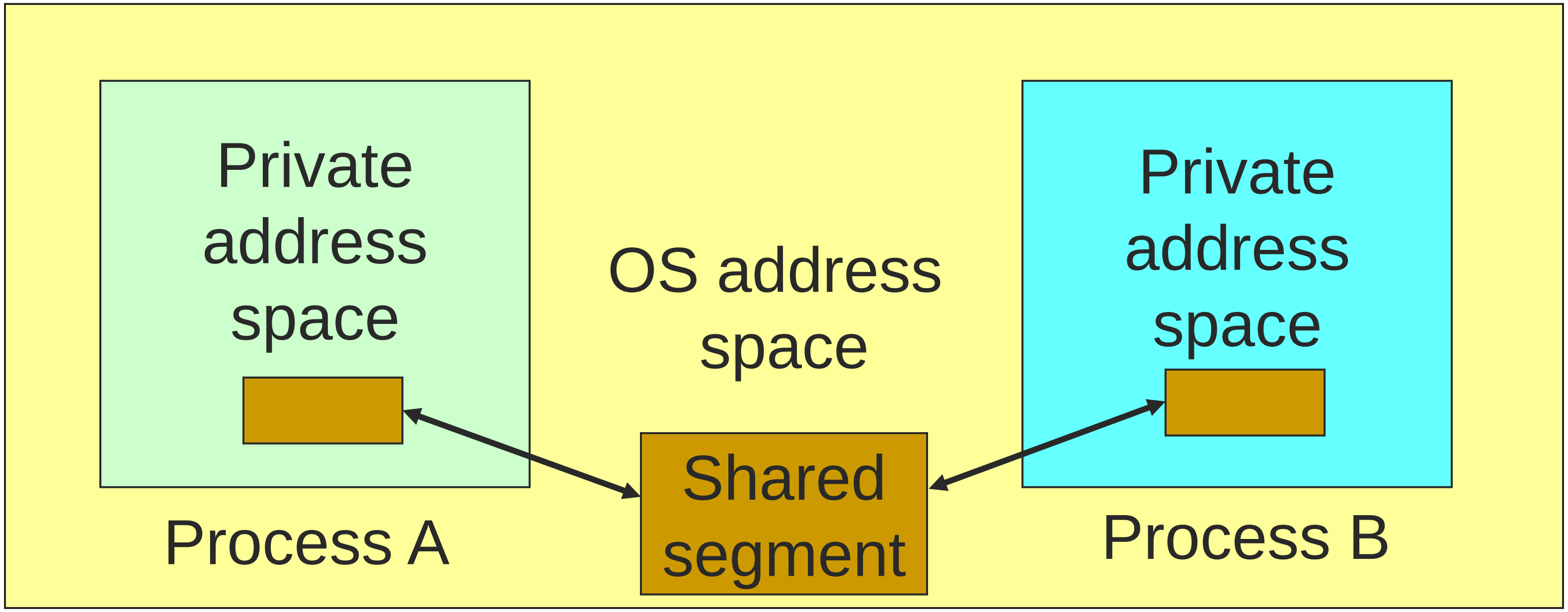
IPC Solutions

- Two options
  - Support some form of shared address space
    - Shared memory, memory mapped files
  - Use OS mechanisms to transport data from one address space to another
    - Pipes, FIFOs
    - Messages, signals

Shared Memory

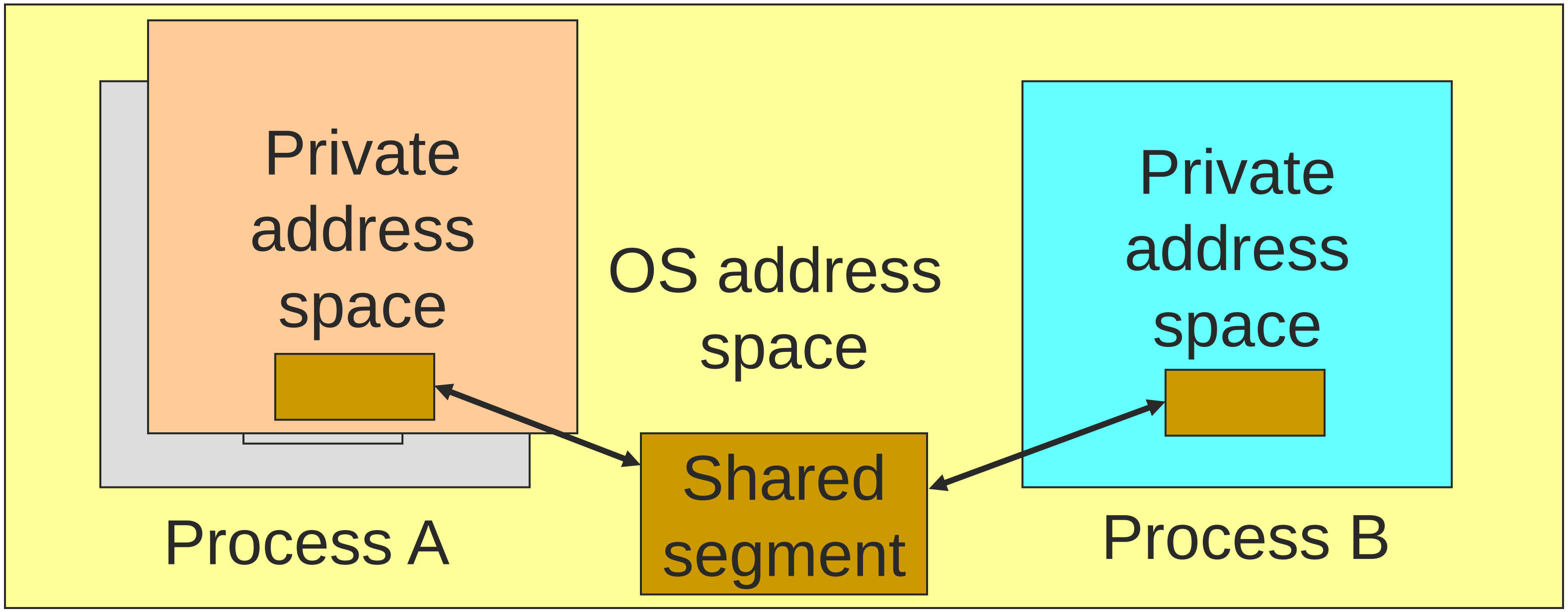
- Processes share the same segment of memory directly
  - Memory is mapped into the address space of each sharing process
  - Memory is persistent beyond the lifetime of the creating or modifying processes (until deleted)
- Mutual exclusion **must** be provided by processes using the shared memory

Shared Memory



- Processes request the segment
- OS maintains the segment
- Processes can attach/detach the segment

Shared Memory



- Can mark segment for deletion on last detach

POSIX Shared Memory

```
#include <sys/types.h>
#include <sys/shm.h>
```

- Create identifier (“key”) for a shared memory segment

```
key_t ftok(const char *pathname, int proj_id);
k = ftok("/my/file", 0xaa);
```

- Create shared memory segment

```
int shmget(key_t key, size_t size, int shmflg);
id = shmget(key, size, 0644 | IPC_CREAT);
```

- Access to shared memory requires an attach

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
shared_memory = (char *) shmat(id, (void *) 0, 0);
```

POSIX Shared Memory

- Write to the shared memory using normal system calls

```
sprintf(shared_memory, "Writing to shared
   memory");
```

- Detach the shared memory from its address space

```
int shmdt(const void *shmaddr);
shmdt(shared_memory);
```

Shared Memory example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  /* a 1K shared memory segment */

int main(int argc, char *argv[]) {
    key_t key;
    int shmid;
    char *data;
    int mode;
```

Shared Memory example

```
/* make the key: */
if ((key = ftok("shmdemo.c", 'R')) == -1) {
    perror("ftok");
    exit(1);
}
/* connect to (and possibly create) the segment: */
if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
    perror("shmget");
    exit(1);
}
/* attach to the segment to get a pointer to it: */
data = shmat(shmid, (void *)0, 0);
if (data == (char *)(-1)) {
    perror("shmat");
    exit(1);
}
```

Shared Memory example

```
/* read or modify the segment, based on the command line: */
if (argc == 2) {
    printf("writing to segment: \"%s\"\n", argv[1]);
    strncpy(data, argv[1], SHM_SIZE);
} else
    printf("segment contains: \"%s\"\n", data);
```

```
/* detach from the segment: */
if (shmdt(data) == -1) {
    perror("shmdt");
    exit(1);
}
```

```
    return 0;
}
```

Run demo

Memory Mapped Files

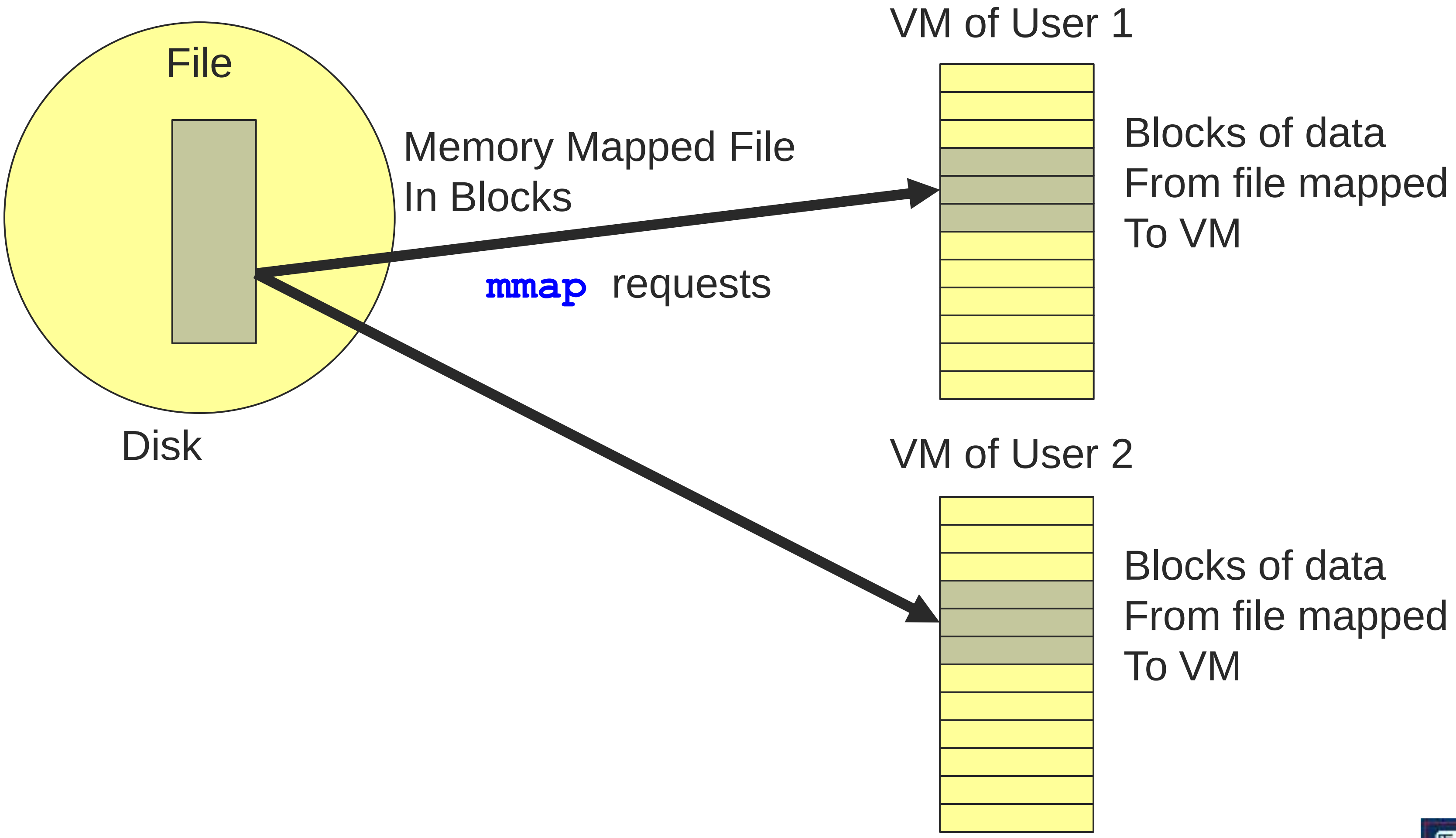
- Memory-mapped file I/O
  - Map a disk block to a page in memory
  - Allows file I/O to be treated as routine memory access
- Use
  - File is initially read using demand paging
  - When needed, a page-sized portion of the file is read from the file system into a physical page of memory
  - Subsequent reads/writes to/from that page are treated as ordinary memory accesses

OK, we haven’t really talked about memory yet, so bear with us …

Memory Mapped Files

- Traditional File I/O
  - Calls to file I/O functions (e.g., **`read()`** and **`write()`**)
    - First copy data to a kernel's intermediary buffer
    - Then transfer data to the physical file or the process
  - Intermediary buffering is slow and expensive
- Memory Mapping
  - Eliminate intermediary buffering
  - Significantly improve performance

Memory Mapped Files



Memory Mapped Files: Benefits

- Treats file I/O like memory access rather than **`read()`**, **`write()`** system calls
  - Simplifies file access; e.g., no need to **`fseek()`**
- Streamlining file access
  - Access a file mapped into a memory region via pointers
  - Same as accessing ordinary variables and objects
- Dynamic loading
  - Map executable files and shared libraries into address space
  - Programs can load and unload executable code sections dynamically

Memory Mapped Files: Benefits

- Several processes can map the same file
  - Allows pages in memory to be shared -- saves memory space
- Memory persistence
  - Enables processes to share memory sections that persist independently of the lifetime of a certain process

Enables IPC!

POSIX Memory Mapping

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot,
           int flags, int fd, off_t off);
```

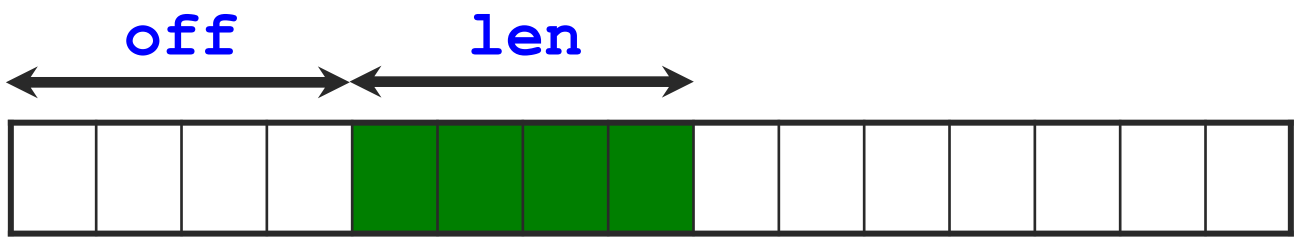
- Memory map a file
  - Establish mapping from the address space of the process to the object represented by the file descriptor
- Parameters:
  - `addr`: the starting memory address into which to map the file
  - `len`: the length of the data to map into memory
  - `prot`: the kind of access to the memory mapped region
  - `flags`: flags that can be set for the system call
  - `fd`: file descriptor
  - `off`: the offset in the file to start mapping from

POSIX Memory Mapping

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot,
           int flags, int fd, off_t off);
```

- Memory map a file
  - Establish mapping from the address space of the process to the object represented by the file descriptor

File **fd**



POSIX Memory Mapping

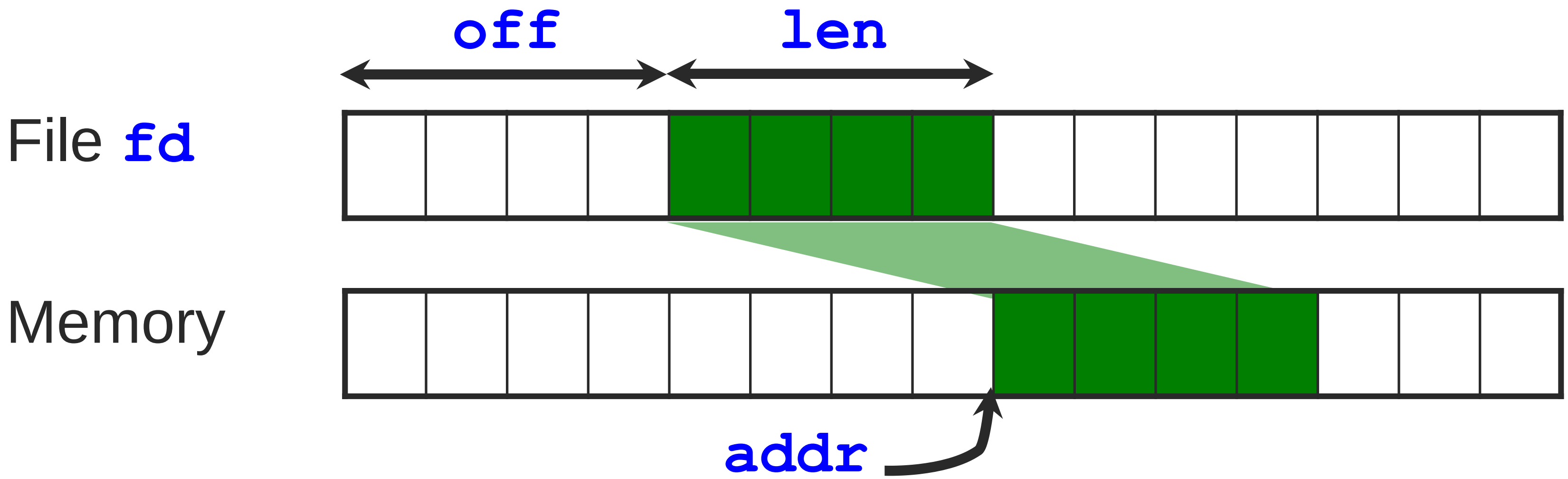
```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot,
                int flags, int fd, off_t off);
```

- Memory map a file
  - Establish a mapping between the address space of the process to the memory object represented by the file descriptor
- Return value: pointer to mapped region
  - On success, implementation-defined function of **`addr`** and **`flags`**.
  - On failure, sets errno and returns **`MAP_FAILED`**

POSIX Memory Mapping

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot,
           int flags, int fd, off_t off);
```

- Memory map a file
  - Establish a mapping between the address space of the process to the memory object represented by the file descriptor



`mmap` options

- Protection Flags
  - `PROT_READ` Data can be read
  - `PROT_WRITE` Data can be written
  - `PROT_EXEC` Data can be executed
  - `PROT_NONE` Data cannot be accessed
- Flags
  - `MAP_SHARED` Changes are shared.
  - `MAP_PRIVATE` Changes are private.
  - `MAP_FIXED` Interpret `addr` exactly

mmap Example

- Map first 4kb of file and read an integer

```
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/types.h>
int main(int argc, char *argv[]) {
    int fd;
    void *pregion;
    if (fd = open(argv[1], O_RDONLY) <0) {
        perror("failed on open");
        return -1;
    }

    write(fd,"\0",1);  // make sure at least 1 page is mapped
```

mmap Example

```
pregion = mmap(NULL, 4096,  PROT_READ,
                MAP_SHARED, fd, 0);
if (pregion == MAP_FAILED) {
  perror("mmap failed")
  return -1;
}
close(fd);        /* close the physical file */
/* access mapped memory; read the first int in
 * the mapped file */
int val = *((int*) pregion);
}
```

munmap

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
```

- Remove a mapping
- Return value
  - 0 on success
  - -1 on error, sets `errno`
- Parameters:
  - `addr`: returned from `mmap()`
  - `len`: same as the `len` passed to `mmap()`

msync

```
#include <sys/mman.h>
int msync(void *addr, size_t len, int flags);
```

- Write all modified data to permanent storage locations
- Return value
  - 0 on success
  - -1 on error, sets **errno**
- Parameters:
  - **addr**: returned from **mmap()**
  - **len**: same as the **len** passed to **mmap()**
  - **flags**:
    - **MS_ASYNC =** Perform asynchronous writes
    - **MS_SYNC** = Perform synchronous writes
    - **MS_INVALIDATE** = Invalidate cached data

Example 2: Shared memory using **mmap**

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>

int main(int argc, char** argv) {
  int    fd;
  char * shared_mem;
  fd = open(argv[1], O_RDWR | O_CREAT);
  write(fd,"\0",1); // make sure at least 1 page is mapped
  shared_mem = mmap(NULL, 10, PROT_READ | PROT_WRITE,
                    MAP_SHARED, fd, 0);
  close(fd);
```

Example 2: Shared memory using `mmap`

Reader

```
if (!strcmp(argv[2], "read")) {
  while (1) {
    printf("%s\n", shared_mem);
    sleep(1);
  }
}
```

Writer

```
else {    while (1)
 scanf("%s\n", shared_mem);   }
}
```

Run demo

Recall POSIX Shared Mem...

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

- Create shared memory segment

```
id = shmget(key, size, 0644 | IPC_CREAT);
```

```
void *shmat(int shmid, const void
   *shmaddr, int shmflg);
```

- Access to shared memory requires an attach

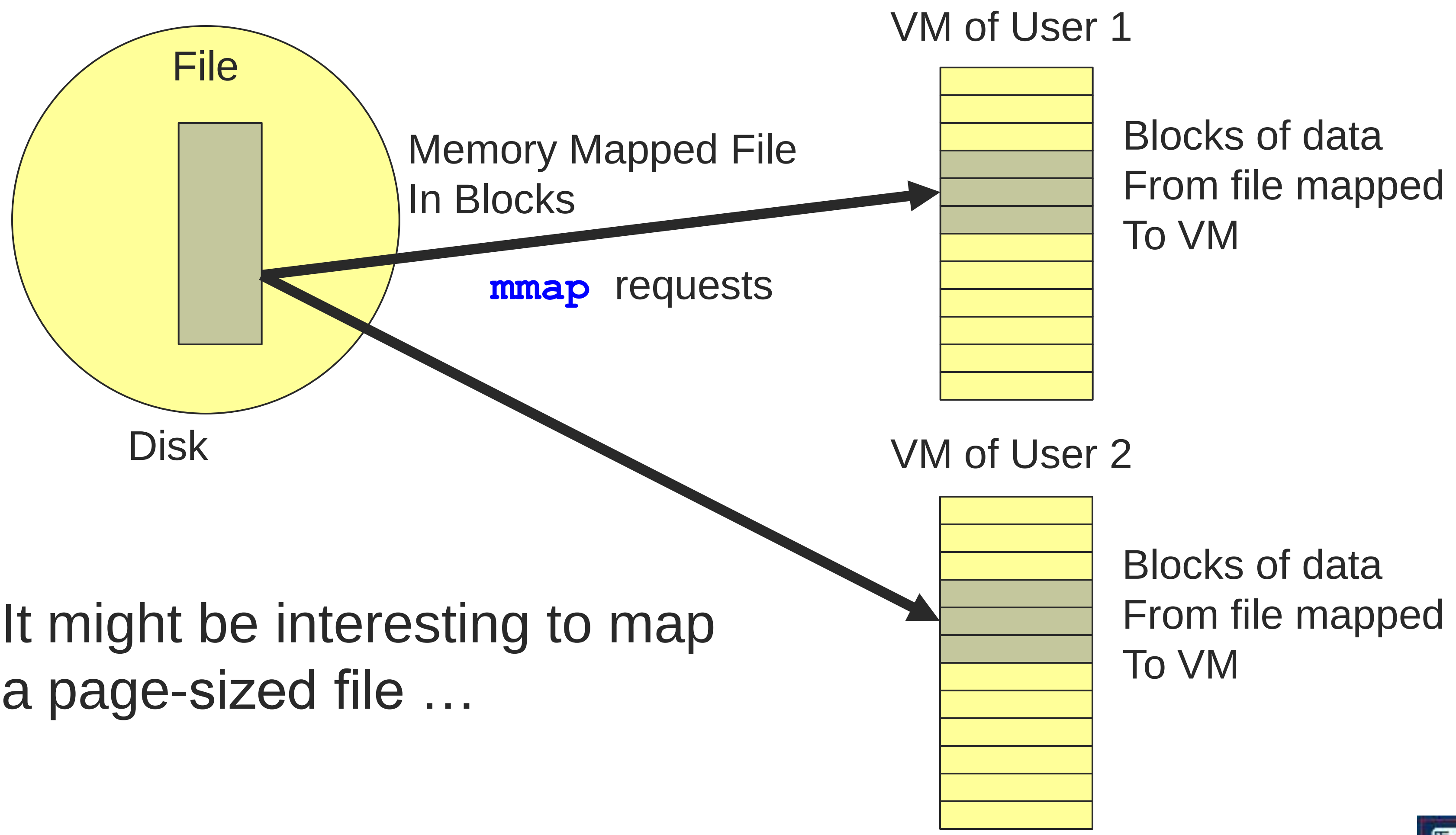
```
shared_memory = (char *) shmat(id, (void
   *) 0, 0);
```

How do `mmap` and POSIX shared memory compare?

- Persistence
  - `shm` memory kept in memory
    - Remains available until system is shut down
  - `mmap` backed by a file
    - Persists even after programs quit or machine reboots

34

Memory mapped files and virtual memory



It might be interesting to map a page-sized file …

Memory mapped files and virtual memory

```
#include <unistd.h>
long sysconf(int name);
```

- Determine the current value of a configurable system variable
- Return value
  - 0 on success
  - -1 on error, sets `errno`
- Parameters:
  - `name`: the system variable to be queried
    - `_SC_PAGESIZE`

sysconf: Creating page-sized memory mapped segments

```
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/mman.h>
main(void) {
    size_t bytesWritten = 0;
    int fd;
    int PageSize;
    const char text = "This is a test";
```

Example

```
if ((PageSize = sysconf(_SC_PAGE_SIZE)) < 0) {
    perror("sysconf() Error=");
    return -1;
}
fd = open("/tmp/mmsyncTest", (O_CREAT | O_TRUNC |
         O_RDWR), (S_IRWXU | S_IRWXG | S_IRWXO));
if (fd < 0) {
    perror("open() error");
    return fd;
}
off_t lastoffset = lseek(fd, PageSize, SEEK_SET);
bytesWritten = write(fd, "x", 1 );
if (bytesWritten != 1 ) {
    perror("write error. ");
    return -1;
}
```

More Examples

```
/* mmap the file. */
void *address;
int len;
off_t my_offset = 0;
len = PageSize;

/* Map one page */
address = mmap(NULL, len, PROT_WRITE, MAP_SHARED, fd,
               my_offset);

if (address == MAP_FAILED) {
    perror("mmap error.");
    return -1;
}
```

More Examples

```
    /* Move some data into the file using memory map. */
    (void) strcpy((char*) address, text);

    /* use msync to write changes to disk. */
    if (msync(address, PageSize , MS_SYNC) < 0 ) {
        perror("msync failed with error:");
        return -1;
    } else
        (void) printf("%s","msync completed successfully.");

    close(fd);
    unlink("/tmp/msyncTest");
}
```

Run demo

Illegal Memory Access

- Use signals!
  - **SIGSEGV** signal allows you to catch references to memory that have the wrong protection mode
- Coming soon... signals!