Announcements

- Survey: tinyurl.com/cs241survey
- No discussion section Thursday
- But lecture **will happen** on Friday

- Correction from last time... pipe() gives you a pair of file descriptors:
    - fildes[0] is output end: you read from it
    - fildes[1] is input end: you write to it

Memory mapped files

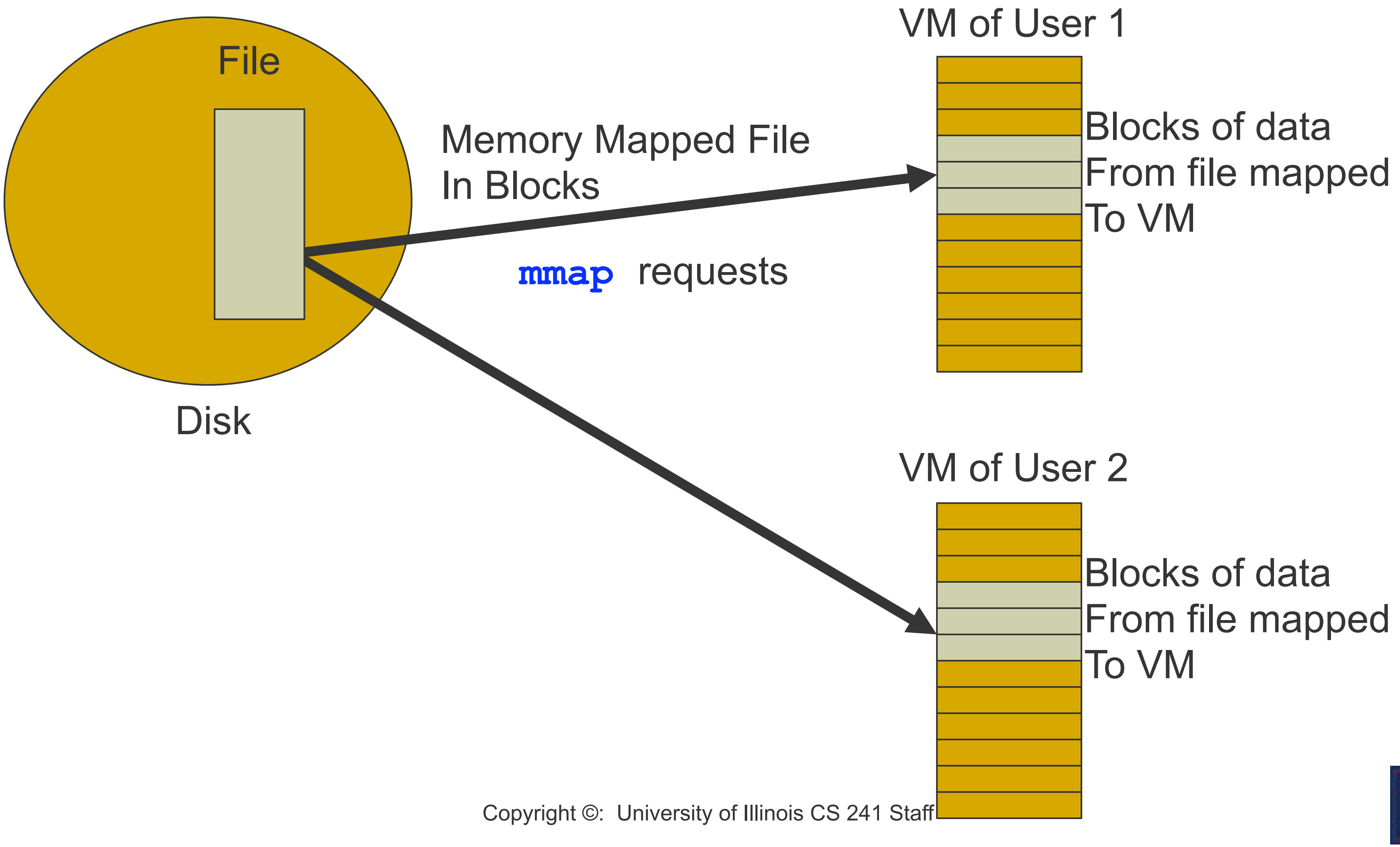
Two ways to access a file

- 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
- Alternative: Memory Mapping
  - Eliminate intermediary buffering
  - Significantly improve performance

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

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()
- Several processes can map the same file
  - Allows pages in memory to be shared -- saves memory space
- Dynamic loading
  - Map executable files and shared libraries into address space
  - Programs can load and unload executable code sections dynamically

Memory Mapped Files: Benefits

- Streamlining file access
  - Access a file mapped into a memory region via pointers
  - Same as accessing ordinary variables and objects
- Memory persistence
  - Enables processes to share memory sections that persist independently of the lifetime of a certain process

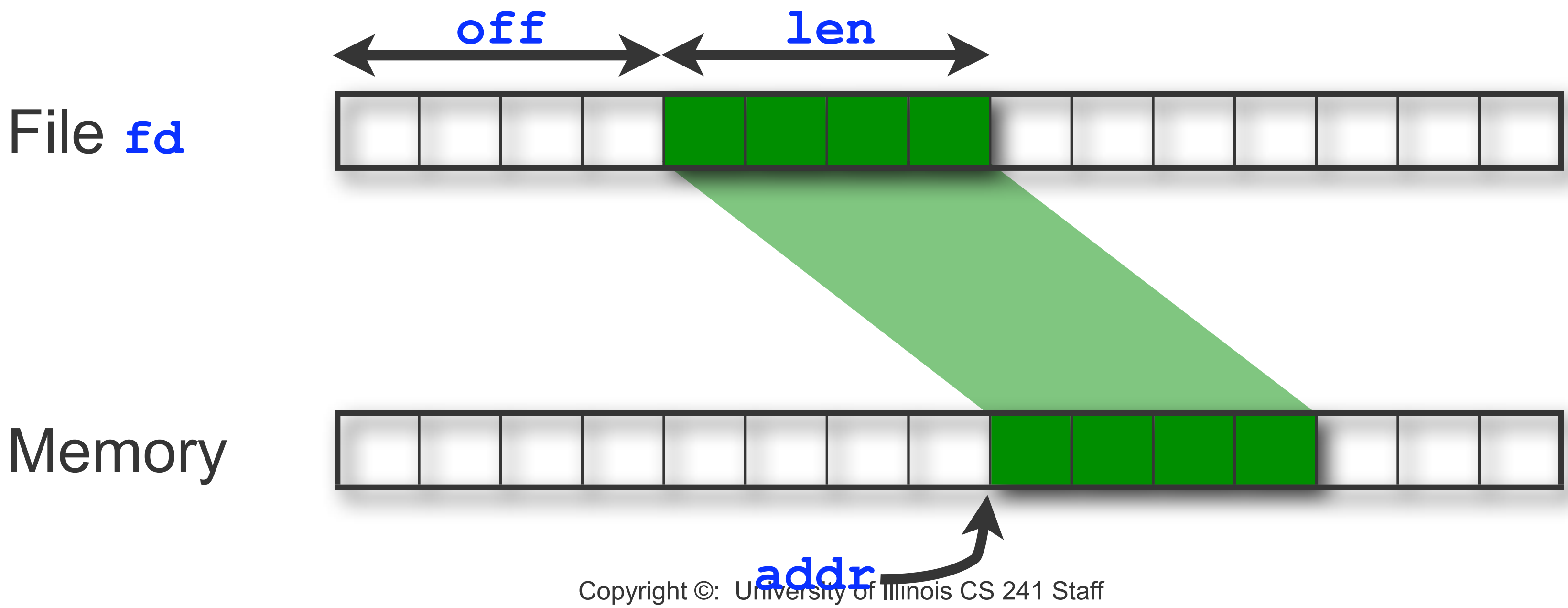
POSIX Memory Mapping

```
#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);
```



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`

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 int

```
#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;
   }
```

`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);
  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);
  }
}
```

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?

- mmap named using filesystem
  - more flexible, convenient naming
  - filesystem permissions
- mmap persists even after programs quit or machine reboots

Signals and Timers

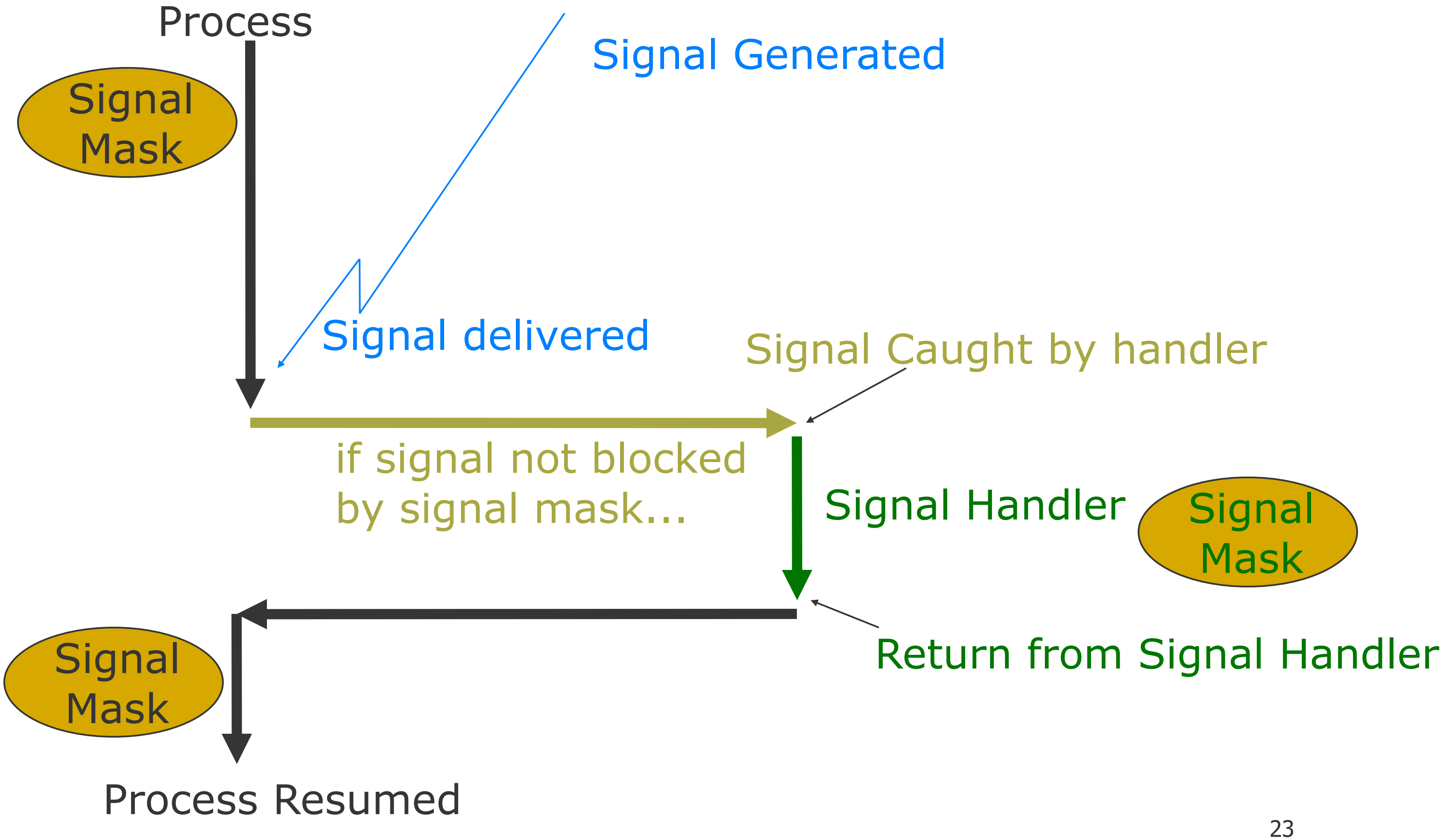
Introduction to Signals

- Signal: notification to a process of an event.
- Enables asynchronous events, e.g.,
  - **Email message arrives** on my machine – mailing agent (user) process should retrieve it
  - **Invalid memory access happens** – OS should inform scheduler to remove process from the processor
  - **Alarm clock goes off** – process which sets the alarm should catch it
- Synchronous event example?

Basic Signal Concepts

- Signal is *generated* when the event that causes it occurs.
- Signal is *delivered* when a process receives it.
- The *lifetime* of a signal is the interval between its generation and delivery.
- Signal that is generated but not delivered is *pending*.
- Process *catches* signal if it executes a *signal handler* when the signal is delivered.
- Alternatively, a process can *ignore* a signal when it is delivered, that is, take no action.
- Process can temporarily prevent signal from being delivered by *blocking* it.
- *Signal Mask* contains the set of signals currently blocked.

How Signals Work



Process

Signal Mask

Signal Generated

Signal delivered

Signal Caught by handler

if signal not blocked by signal mask...

Signal Handler

Signal Mask

Return from Signal Handler

Signal Mask

Process Resumed

Examples of POSIX Required Signals

Signal	Description	Default action
SIGABRT	process abort	implementation dependent
SIGALRM	**alarm clock**	**abnormal termination**
SIGBUS	access undefined part of memory object	implementation dependent
SIGCHLD	child terminated, stopped or continued	ignore
SIGILL	invalid hardware instruction	implementation dependent
SIGINT	**interactive attention signal (usually ctrl-C)**	**abnormal termination**
SIGKILL	**terminated (cannot be caught or ignored)**	**abnormal termination**

Examples of POSIX Required Signals

Signal	Description	Default action
SIGSEGV	**Invalid memory reference**	**implementation dependent**
SIGSTOP	Execution stopped	stop
SIGTERM	**termination**	**Abnormal termination**
SIGTSTP	Terminal stop	stop
SIGTTIN	Background process attempting read	stop
SIGTTOU	Background process attempting write	stop
SIGURG	**High bandwidth data available on socket**	**ignore**
SIGUSR1	**User-defined signal 1**	**abnormal termination**

Generating Signals

- Signal has a symbolic name starting with SIG
- Signal names are defined in `signal.h`
- Users can generate signals (e.g., SIGUSR1)
- OS generates signals when certain errors occur (e.g., SIGSEGV – invalid memory reference)
- Specific calls generate signals such as alarm (e.g., SIGALRM)