Today's lecture

- Interprocess communication:
  Pipes & FIFOs
- Memory-mapped files
- Klara Nahrstedt: Experiments with mobile technologies

Interprocess Communication

Interprocess Communication

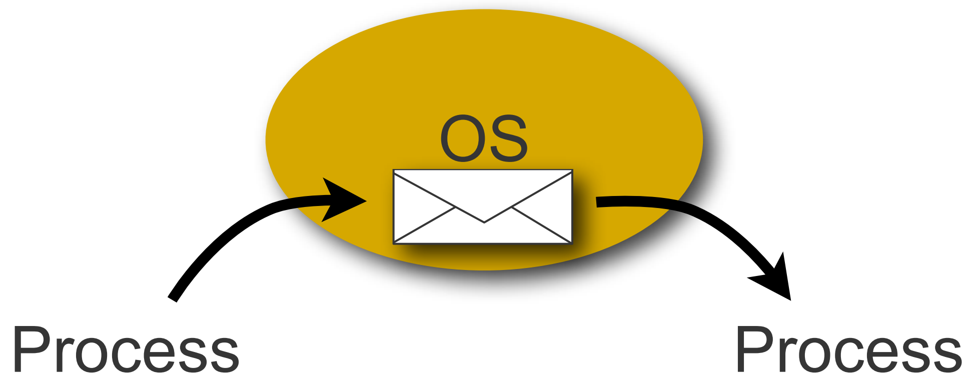
- What is IPC?
  - Mechanisms to transfer data between processes
- Why is it needed?
  - Not all important procedures can be easily built in a single process

Two kinds of IPC

Mind meld

Direct sharing of memory between processes

Intermediary



Message queues
Pipes, FIFOs
Files

UNIX Pipes

```
#include <unistd.h>
int pipe(int fildes[2]);
```

- Creates a message pipe
  - Anything can be written to the pipe, and read from the other end in the order it came in
  - OS enforces mutual exclusion: only one process at a time
  - Accessed by a file descriptor, like an ordinary file
  - Processes sharing the pipe must have same parent in common
- Returns a pair of file descriptors
  - `fildes[0]` is the output end of the pipe: you read from it
  - `fildes[1]` is the input end of the pipe: you write to it

UNIX Pipe Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
  int pfds[2];
  char buf[30];

  pipe(pfds);
```

```c
  if (!fork()) {
    printf(" CHILD: writing to pipe\n");
    write(pfds[1], "test", 5);
    printf(" CHILD: exiting\n");
    exit(0);
  } else {
    printf("PARENT: reading from pipe\n");
    read(pfds[0], buf, 5);
    printf("PARENT: read \"%s\"\n", buf);
    wait(NULL);
  }
  return 0;
}
```

UNIX Pipe Example: `ls | wc -l`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
  int pfds[2];

  pipe(pfds);
```

```
  if (!fork()) {
    close(1);        /* close stdout */
    dup(pfds[1]);    /* make stdout pfds[1] */
    close(pfds[0]);  /* don't need this */
    execlp("ls", "ls", NULL);
  } else {
    close(0);        /* close stdin */
    dup(pfds[0]);    /* make stdin pfds[0] */
    close(pfds[1]);  /* don't need this */
    execlp("wc", "wc", "-l", NULL);
  }
  return 0;
}
```

FIFOs

- A pipe disappears when no process has it open
- FIFOs = named pipes
  - Special pipes that persist even after all the processes have closed them
  - Actually implemented as a file and appears in filesystem!

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

int status;
...
status = mkfifo("/home/cnd/mod_done",
              S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH);
```

FIFO Example: Producer-Consumer

- Producer
  - Writes to fifo
- Consumer
  - Reads from fifo
  - Outputs data to file
- Fifo
  - Ensures atomicity of write

FIFO Example

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include "restart.h"

int main (int argc, char *argv[]) {
  int requestfd;

  if (argc != 2) { /* name of consumer fifo on the command line */
    fprintf(stderr, "Usage: %s fifoname > logfile\n", argv[0]);
    return 1;
  }
```

FIFO Example

```
/* create a named pipe to handle incoming requests */
if ((mkfifo(argv[1], S_IRWXU | S_IWGRP| S_IWOTH) == -1)
    && (errno != EEXIST))
{
  perror("Server failed to create a FIFO");
  return 1;
}

/* open a read/write communication endpoint to the pipe */
if ((requestfd = open(argv[1], O_RDWR)) == -1) {
  perror("Server failed to open its FIFO");
  return 1;
}
/* Write to pipe like you would to a file */
...
}
```

Demo!

Memory mapped files

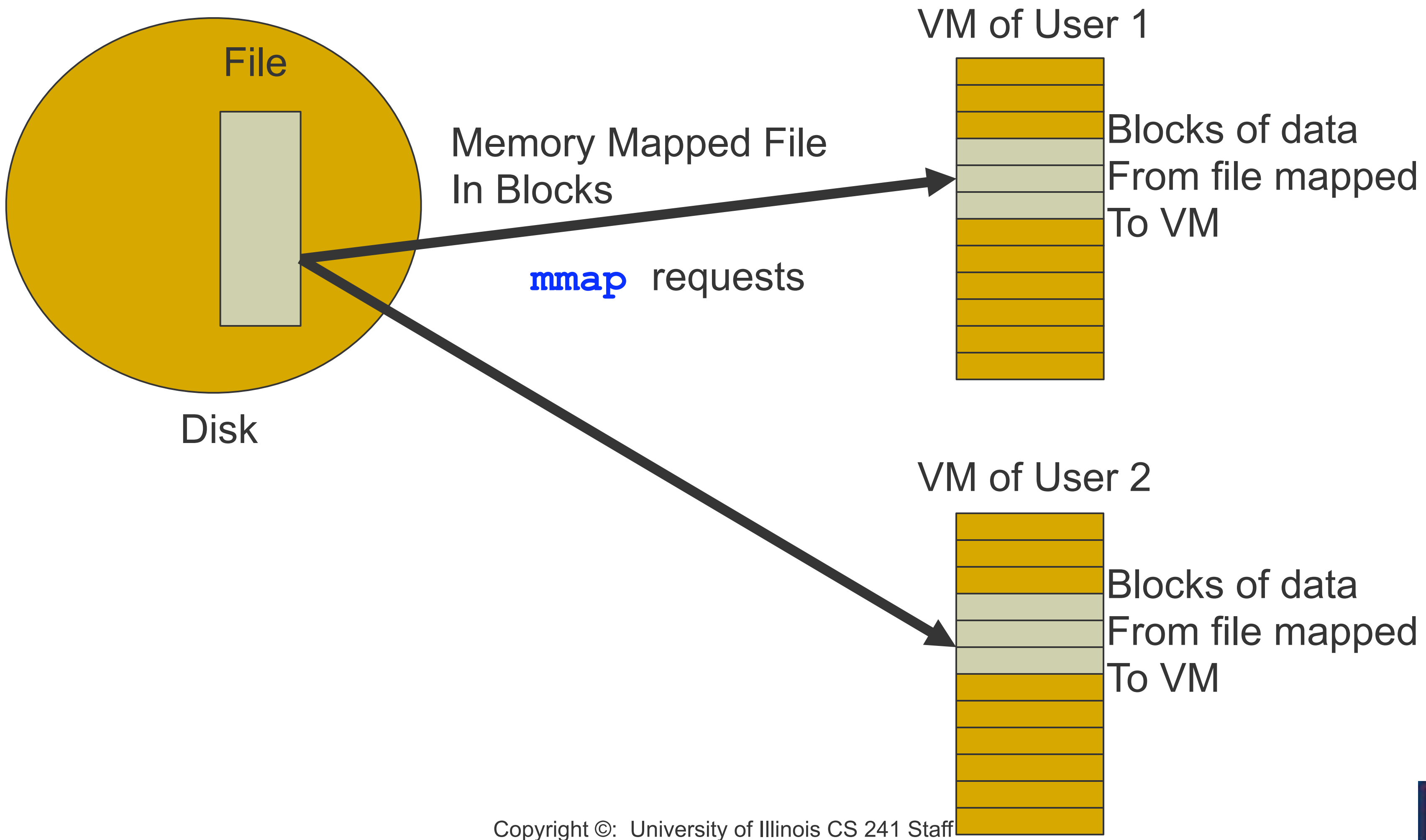
File Access

- 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