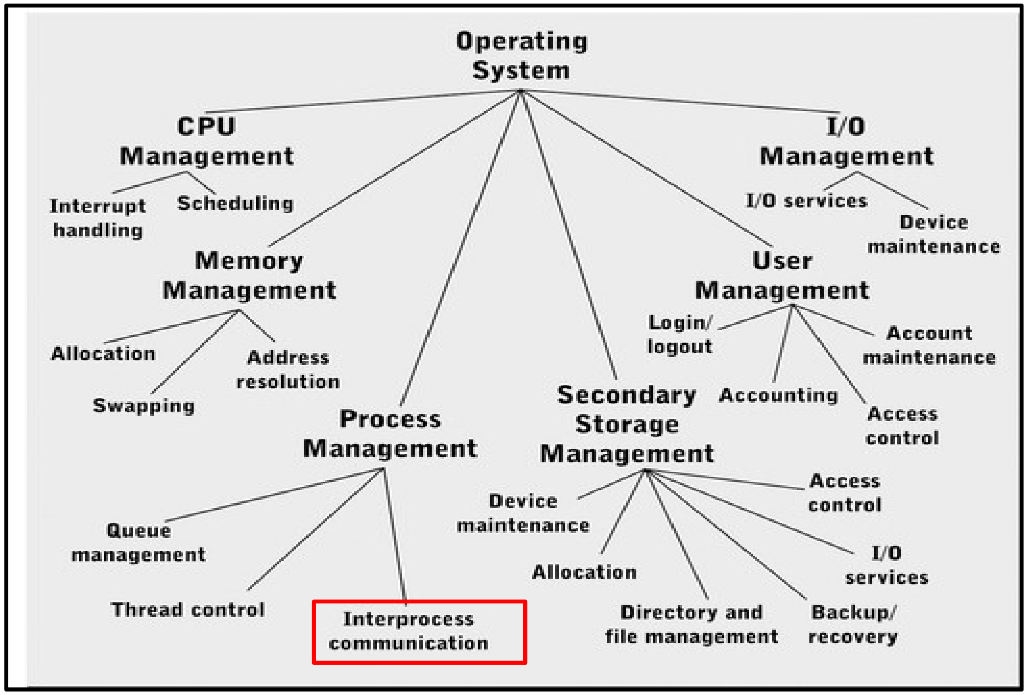
Operating Systems

Inter-Process Communication

ENCE 360

Outline

- Introduction
- Examples
  - Shared Memory
  - Files
  - Pipes
  - Signals



Pages 43-45, 733-734

MODERN OPERATING SYSTEMS (MOS)

By Andrew Tanenbaum

Interprocess Communication (IPC)

- Independent process cannot affect or be affected by execution of another process
- Cooperating process *can* affect or be affected by execution of another process

Examples?

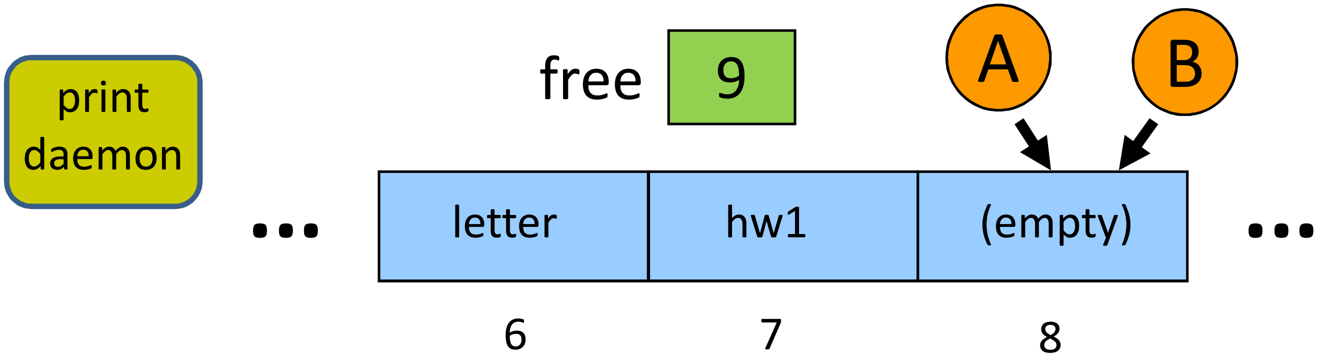
- Advantages of process cooperation:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Cooperating Processes - Examples

- Communication example – Unix shell

```
cat file.jpg | jpegtopnm | pnmscale 0.1 | ssh claypool@host.com "cat  > file.pnm"
```

- Sharing example – print spooler
  - Processes (A, B) enter file name in spooler queue
  - Printer daemon checks queue and prints



print daemon

free 9

A

B

… letter | hw1 | (empty) …

6 7 8

Interprocess Communication (IPC)

- Independent process cannot affect or be affected by execution of another process
- Cooperating process *can* affect or be affected by execution of another process

- Advantages of process cooperation:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

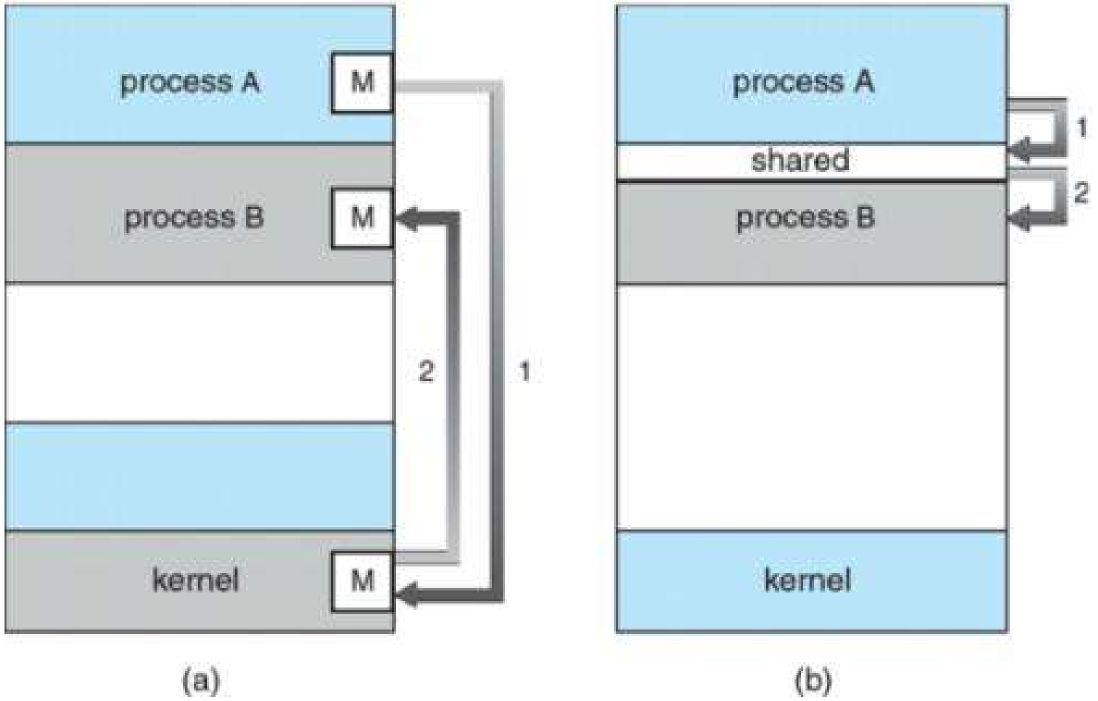
THE CRUX OF THE PROBLEM:
HOW TO EFFICIENTLY ENABLE PROCCESS
COMMUNICATION/COORDINATION?

How do processes share data?

How do processes communicate data?

How to avoid problems/issues when sharing data?

IPC Paradigms



a) Message passing

Why good? All sharing is explicit less chance for error

Why bad? Overhead. Data copying, cross protection domains

b) Shared Memory

Why good? Performance. Set up shared memory once, then access w/o crossing protection domains

Why bad? Can change without process knowing, error prone

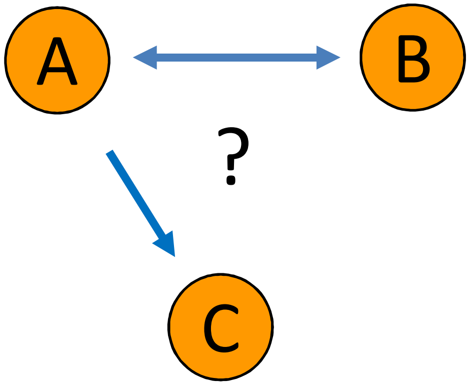
Outline

- Introduction (done)
- Examples
  - Shared Memory (next)
  - Files
  - Pipes
  - Signals

What Are Some IPC Mechanisms?

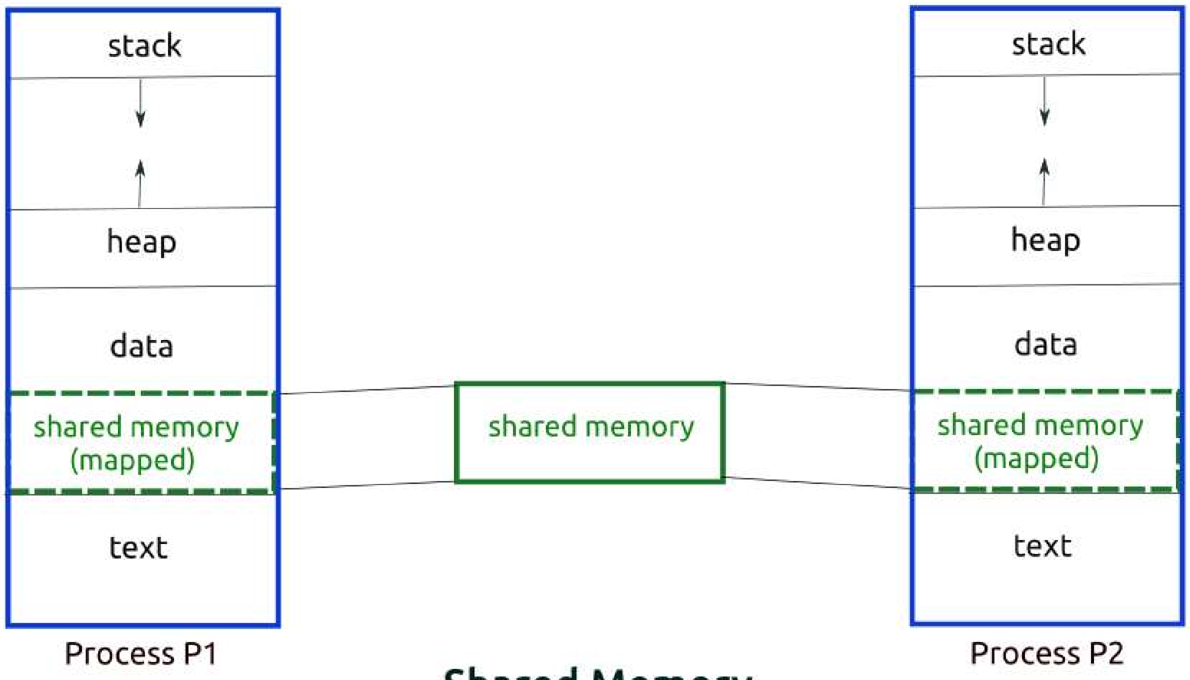
Some IPC Mechanisms

- Shared memory
  - Through shared variables
- File system
  - By reading and writing to file(s)
- Message passing
  - By passing data through pipe
  - Also: remote procedure call, sockets
- Signal
  - By indicating event occurred



IPC Using Shared Memory

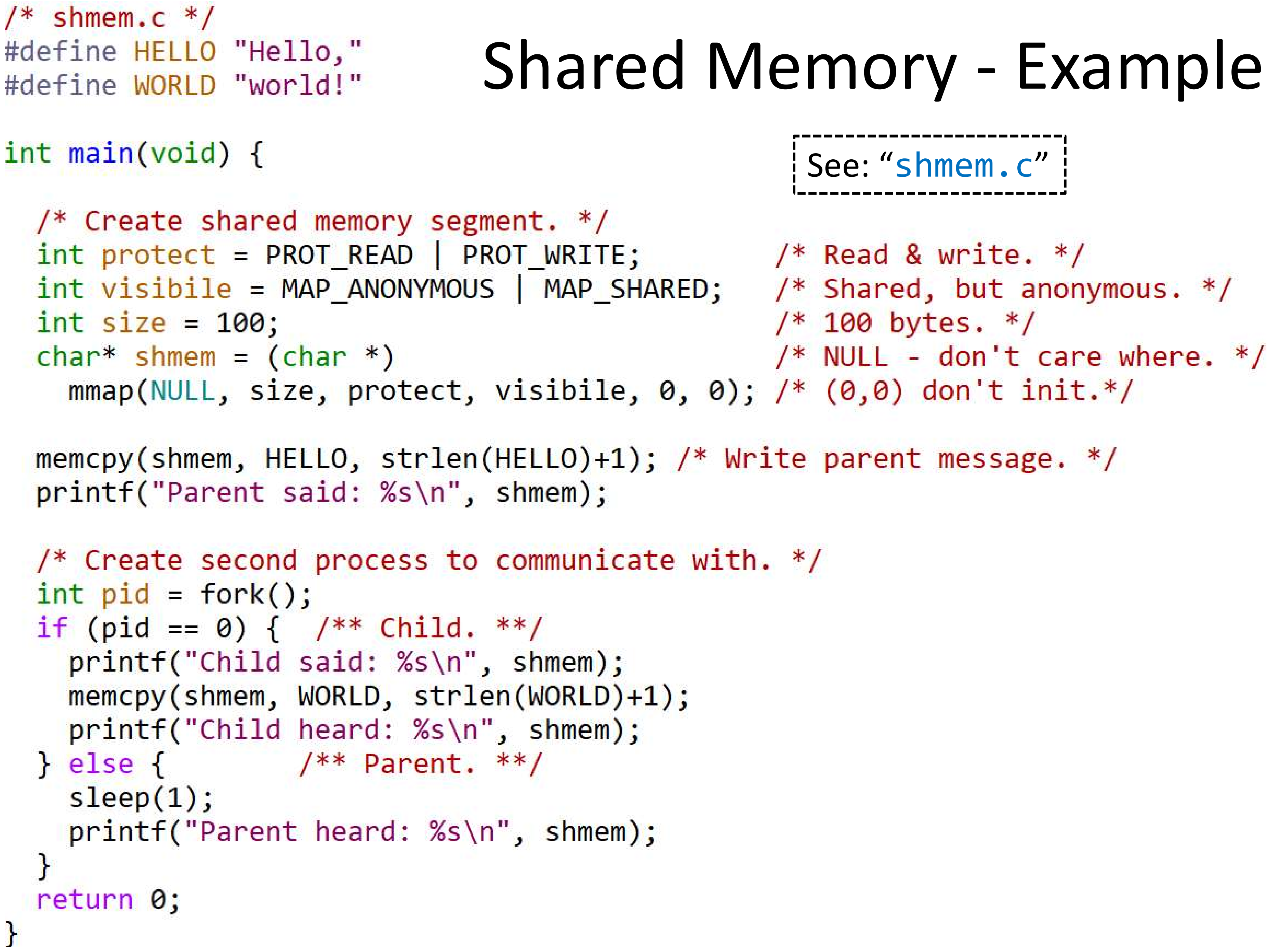
- System call to create shared memory segment
- Once created, access as “normal” memory



Shared Memory

Shared Memory - Example

See: “shmem.c”



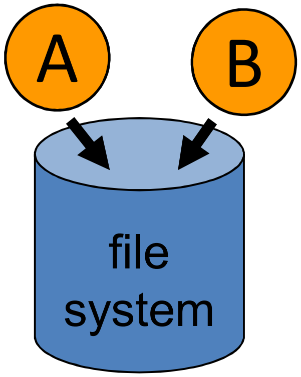
```c
/* shmem.c */
#define HELLO "Hello,"
#define WORLD "world!"

int main(void) {

  /* Create shared memory segment. */
  int protect = PROT_READ | PROT_WRITE;        /* Read & write. */
  int visibile = MAP_ANONYMOUS | MAP_SHARED;   /* Shared, but anonymous. */
  int size = 100;                              /* 100 bytes. */
  char* shmem = (char *)                       /* NULL - don't care where. */
    mmap(NULL, size, protect, visibile, 0, 0); /* (0,0) don't init.*/

  memcpy(shmem, HELLO, strlen(HELLO)+1); /* Write parent message. */
  printf("Parent said: %s\n", shmem);

  /* Create second process to communicate with. */
  int pid = fork();
  if (pid == 0) {  /** Child. **/
    printf("Child said: %s\n", shmem);
    memcpy(shmem, WORLD, strlen(WORLD)+1);
    printf("Child heard: %s\n", shmem);
  } else {        /** Parent. **/
    sleep(1);
    printf("Parent heard: %s\n", shmem);
  }
  return 0;
}
```

Outline

- Introduction (done)
- Examples
  - Shared Memory (done)
  - Files (next)
  - Pipes
  - Signals

IPC Using Files

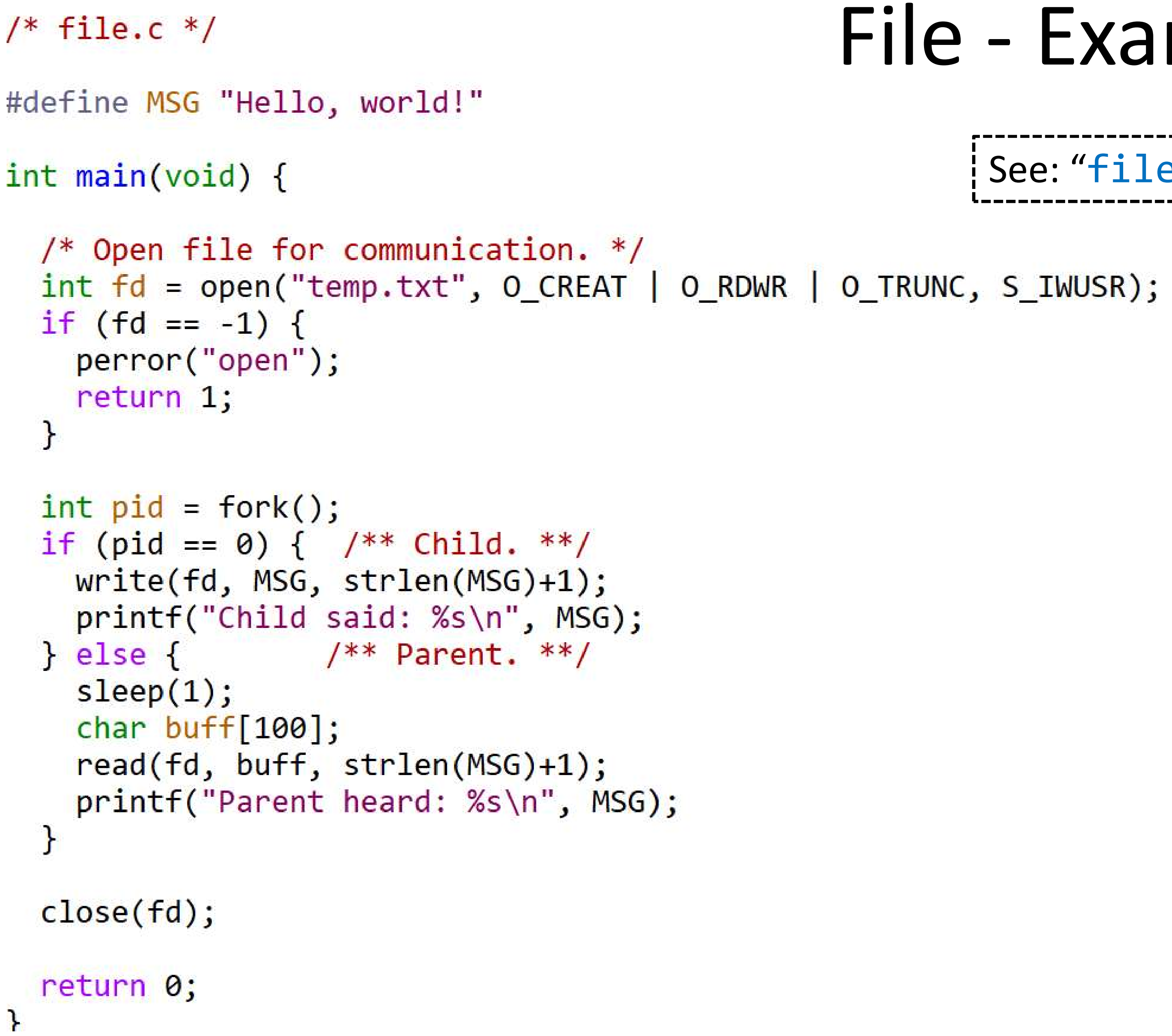
- Process writes to file, another reads from same file
- Note, if both writing, requires locking to share file safely
  - File – locks the whole file (e.g., `flock()`, `fcntl()`)
  - Record – locks portion of file (e.g., databases)



Note! Windows and Linux do not lock by default

File - Example

See: “file.c”

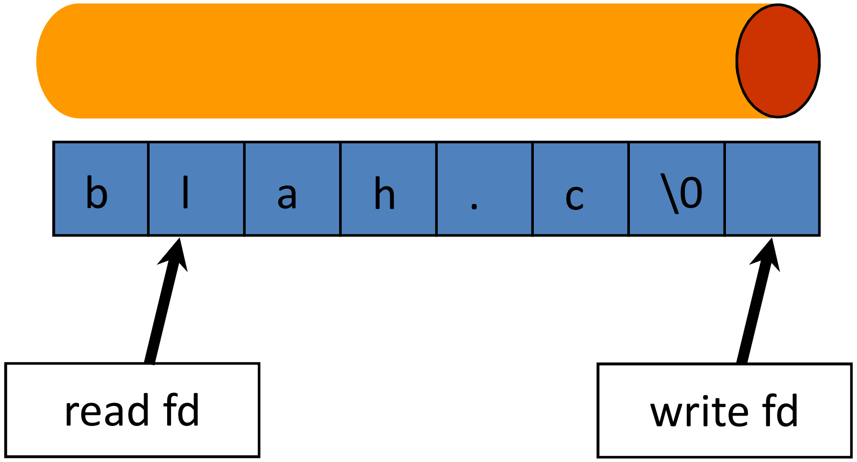


```c
/* file.c */

#define MSG "Hello, world!"

int main(void) {

  /* Open file for communication. */
  int fd = open("temp.txt", O_CREAT | O_RDWR | O_TRUNC, S_IWUSR);
  if (fd == -1) {
    perror("open");
    return 1;
  }

  int pid = fork();
  if (pid == 0) {  /** Child. **/
    write(fd, MSG, strlen(MSG)+1);
    printf("Child said: %s\n", MSG);
  } else {         /** Parent. **/
    sleep(1);
    char buff[100];
    read(fd, buff, strlen(MSG)+1);
    printf("Parent heard: %s\n", MSG);
  }

  close(fd);

  return 0;
}
```

Outline

- Introduction (done)
- Examples
  - Shared Memory (done)
  - Files (done)
  - Pipes (next)
  - Signals

IPC Using Pipes



b	l	a	h	.	c	\0	

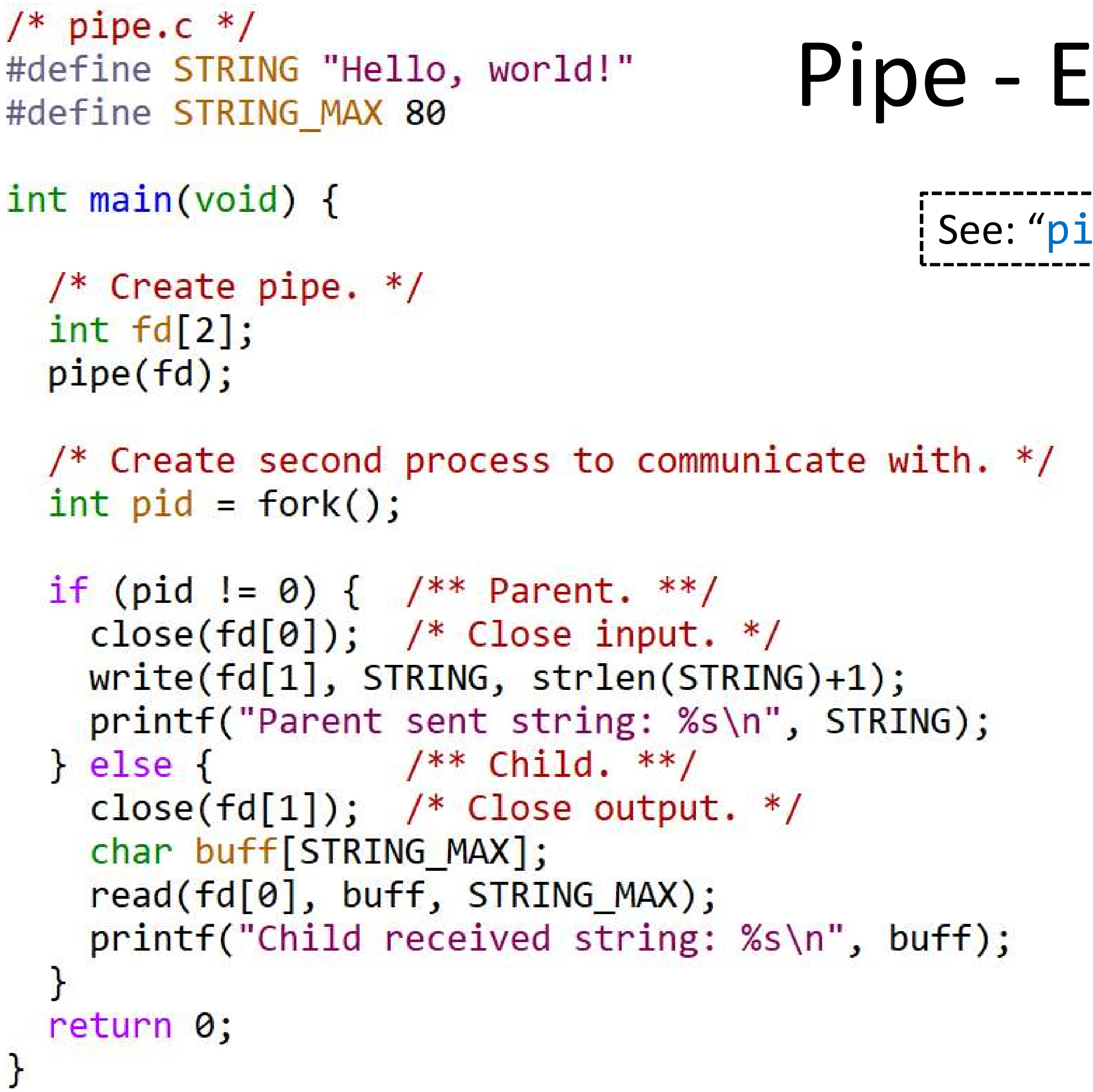
read fd

write fd

- A bounded buffer, provided by OS
  - Shared buffer
  - Block writes to full pipe
  - Block reads to empty pipe
- System calls to create/destroy
  - e.g., `pipe()`
- System calls to read/write
  - e.g., `read()`, `write()`

Pipe - Example

See: “pipe.c”



```
/* pipe.c */
#define STRING "Hello, world!"
#define STRING_MAX 80

int main(void) {

  /* Create pipe. */
  int fd[2];
  pipe(fd);

  /* Create second process to communicate with. */
  int pid = fork();

  if (pid != 0) {  /** Parent. **/
    close(fd[0]);  /* Close input. */
    write(fd[1], STRING, strlen(STRING)+1);
    printf("Parent sent string: %s\n", STRING);
  } else {         /** Child. **/
    close(fd[1]);  /* Close output. */
    char buff[STRING_MAX];
    read(fd[0], buff, STRING_MAX);
    printf("Child received string: %s\n", buff);
  }
  return 0;
}
```

Named versus Unnamed Pipes

- Unnamed pipe

```
int pid[2];
pipe(pid);

write(pid[1], buffer, strlen(buffer)+1);
read(pid[0], buffer, BUFSIZE);
```

Persistent (after processes exit)
Can be shared by any process)

- Named pipe

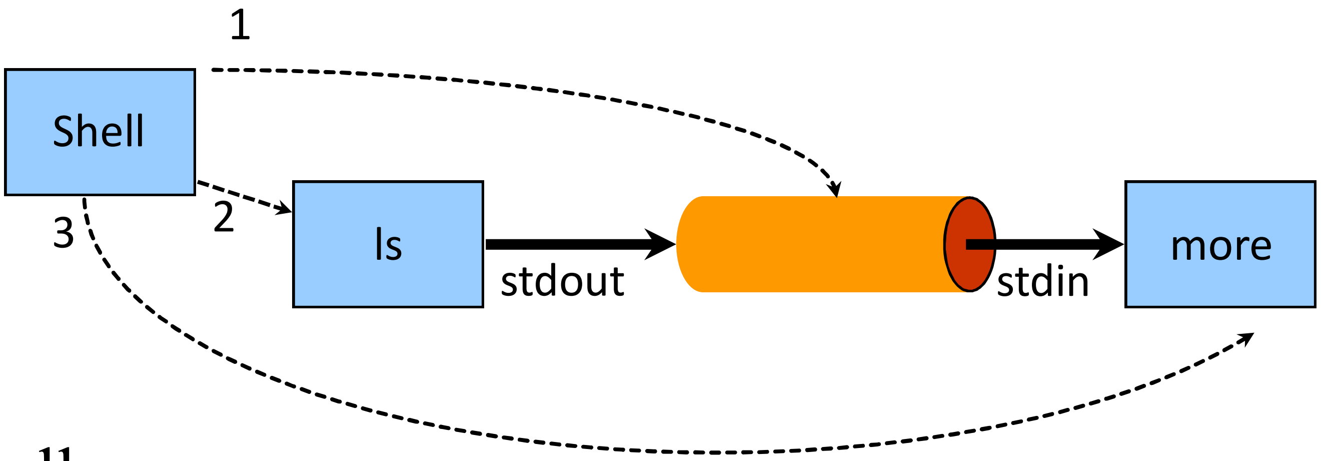
```
int pid0, pid1;
mknod("named_pipe_filename", S_IFIFO | 0666, 0);
pid1 = open("named_pipe_filename", O_WRONLY);
pid0 = open("named_pipe_filename", O_RDONLY);

write(pid1, buffer, strlen(buffer)+1);
read(pid0, buffer, BUFSIZE);
```

Can be treated like FIFO file

The Shell Using a Pipe

- One process writes, 2nd process reads

```
% ls | more
```



Shell:

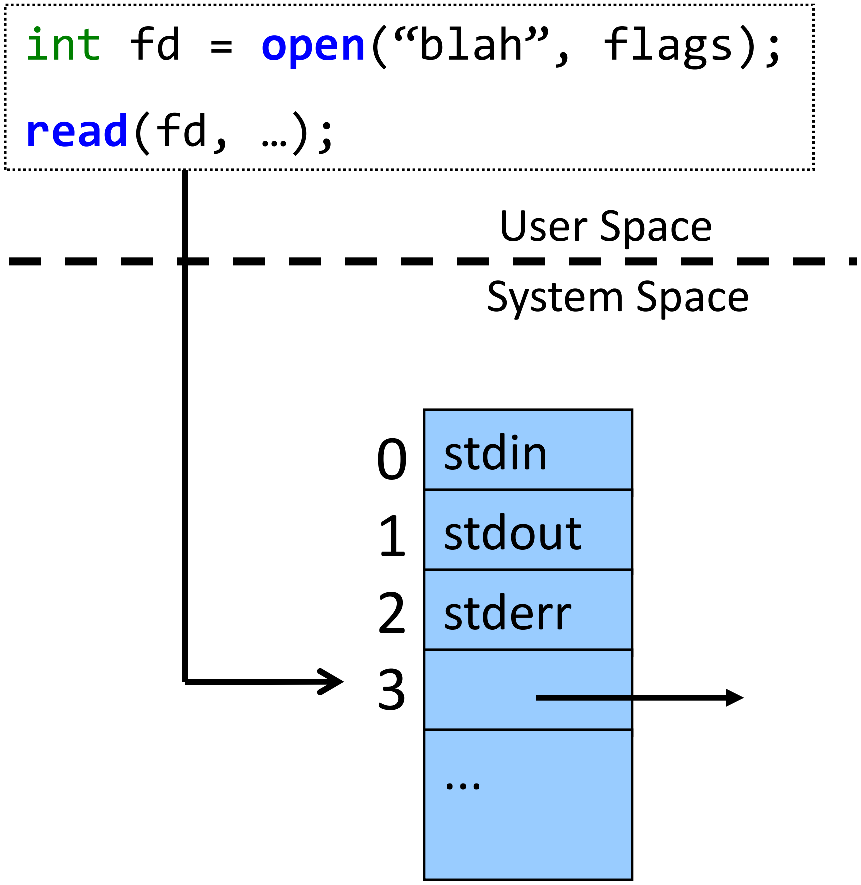
1 Create unnamed pipe

2 Create process for `ls`, setting `stdout` to write side

3 Create process for `more`, setting `stdin` to read side

Ok, but how to “set” `stdout` and `stdin`?

File Descriptors



```
int fd = open(“blah”, flags);
read(fd, …);
```

0	stdin
1	stdout
2	stderr
3	
	…

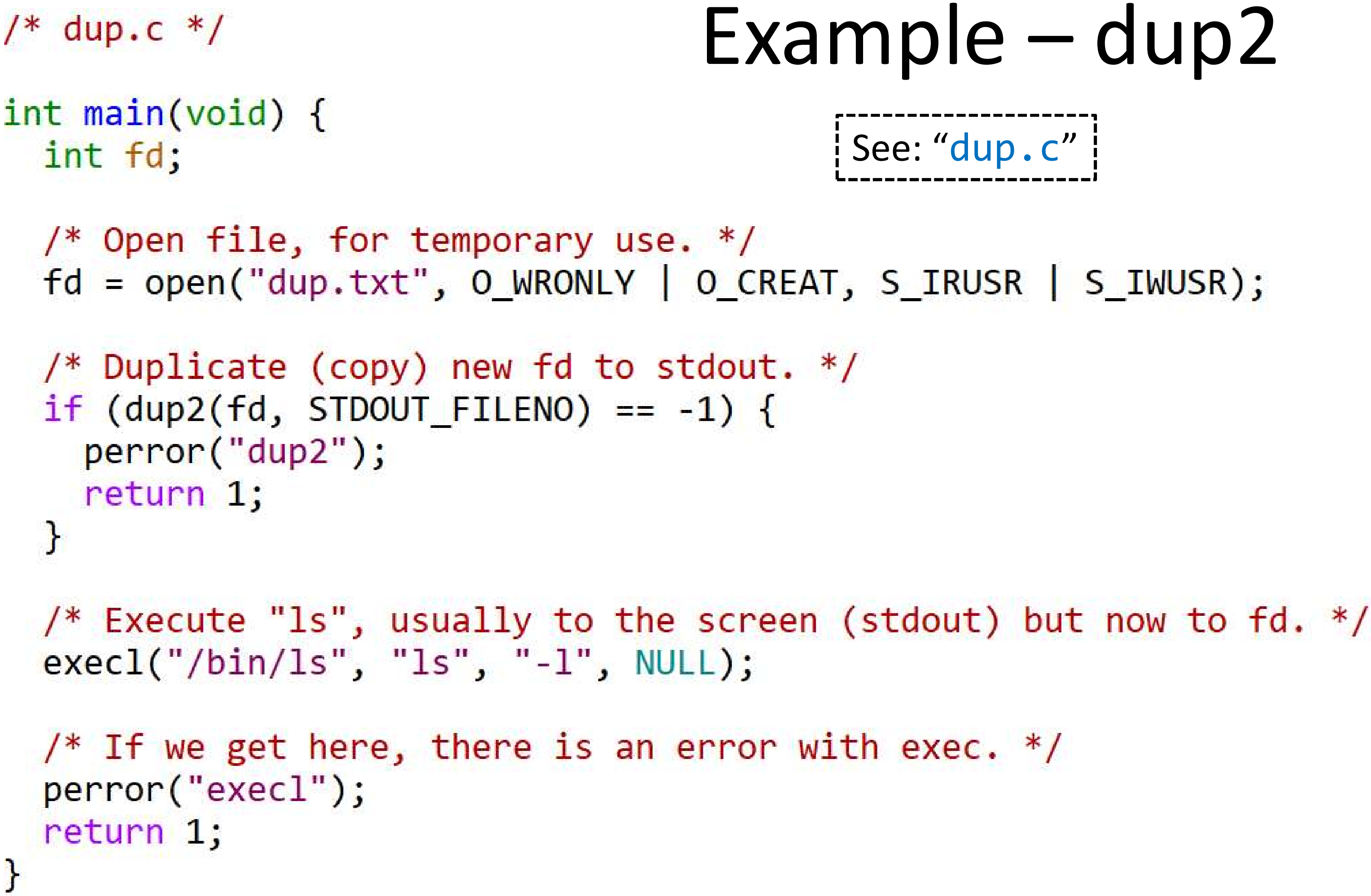
(index)

(Per process)

- 0-2 standard for each process
- Used for files, pipes, sockets …
- Can be changed
  - – Openend
  - – Closed
  - – Copied (dup2())

Example – dup2

See: “dup.c”

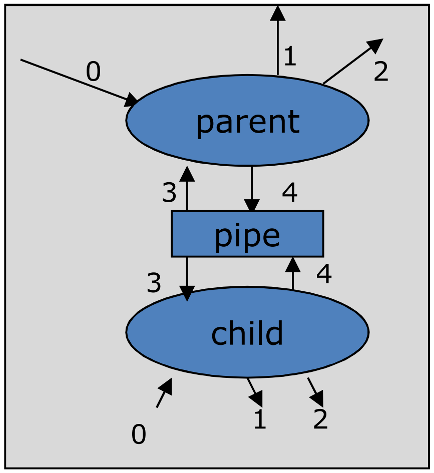


```
/* dup.c */

int main(void) {
  int fd;

  /* Open file, for temporary use. */
  fd = open("dup.txt", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);

  /* Duplicate (copy) new fd to stdout. */
  if (dup2(fd, STDOUT_FILENO) == -1) {
    perror("dup2");
    return 1;
  }

  /* Execute "ls", usually to the screen (stdout) but now to fd. */
  execl("/bin/ls", "ls", "-l", NULL);

  /* If we get here, there is an error with exec. */
  perror("execl");
  return 1;
}
```

Example – dup2 w/pipe

```
/* dup-2.c */

void main(void) {

  int fd[2];
  pipe(fd);

  if (fork() == 0) { /** Child. **/

    /* copy pipe out to stdout */
    dup2(fd[1], STDOUT_FILENO);
    execl("/bin/ls","ls","-s","-1",NULL);

  } else {           /** Parent. **/

    /* copy pipe in to stdin */
    dup2(fd[0], STDIN_FILENO);
    execl("/usr/bin/sort","sort","-n",NULL);
  }
}
```

before



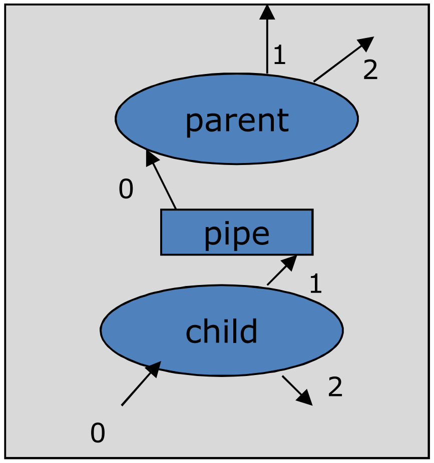
0 stdin
1 stdout
2 stderror
3 pipe read
4 pipe write

File Descriptor Table (FDT) after fork parent

0 stdin
1 stdout
2 stderror
3 pipe read
4 pipe write

FDT after fork child

after



0 pipe read
1 stdout
2 stderror
3 pipe read
4 pipe write

FDT after dup2 parent

0 stdin
1 pipe write
2 stderror
3 pipe read
4 pipe write

FDT after dup2 child

0 pipe read
1 stdout
2 stderror

0 stdin
1 pipe write
2 stderror

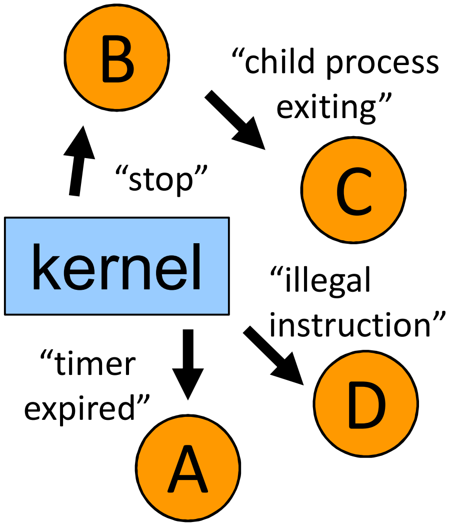
FDTs after execl

Outline

- Introduction (done)
- Examples
  - Shared Memory (done)
  - Files (done)
  - Pipes (done)
  - Signals (next)

IPC using Signals

- Signal corresponds to an event
  - Raised (or "sent") by one process (or hardware)
  - Handled by another
  - E.g., ctrl-c → sends signal (SIGINT) to process
- Originate from various sources
  - Hardware. e.g., divide by zero
  - Operating System. e.g., file size limit exceeded
  - User (shell)
    - Keyboard. e.g., ctrl-Z (SIGTSTP), ctrl-C (SIGINT)
    - Kill command
  - Other processes. e.g., child
- Handling varies by processes
  - default – most terminate process
  - catch – catch and do appropriate action
  - ignore – do not take any action, but do not terminate



Generating & Handling Signals

Generate

- `kill()`- send signal to specified process
  - `kill(int pid, int sig);`
  - signal: 0-31
  - pid == 0 → goes to all user's processes
- `alarm()`- send SIGALRM to itself after specified time
- `raise()`- send signal to itself
  - `kill(getpid(), sig);`

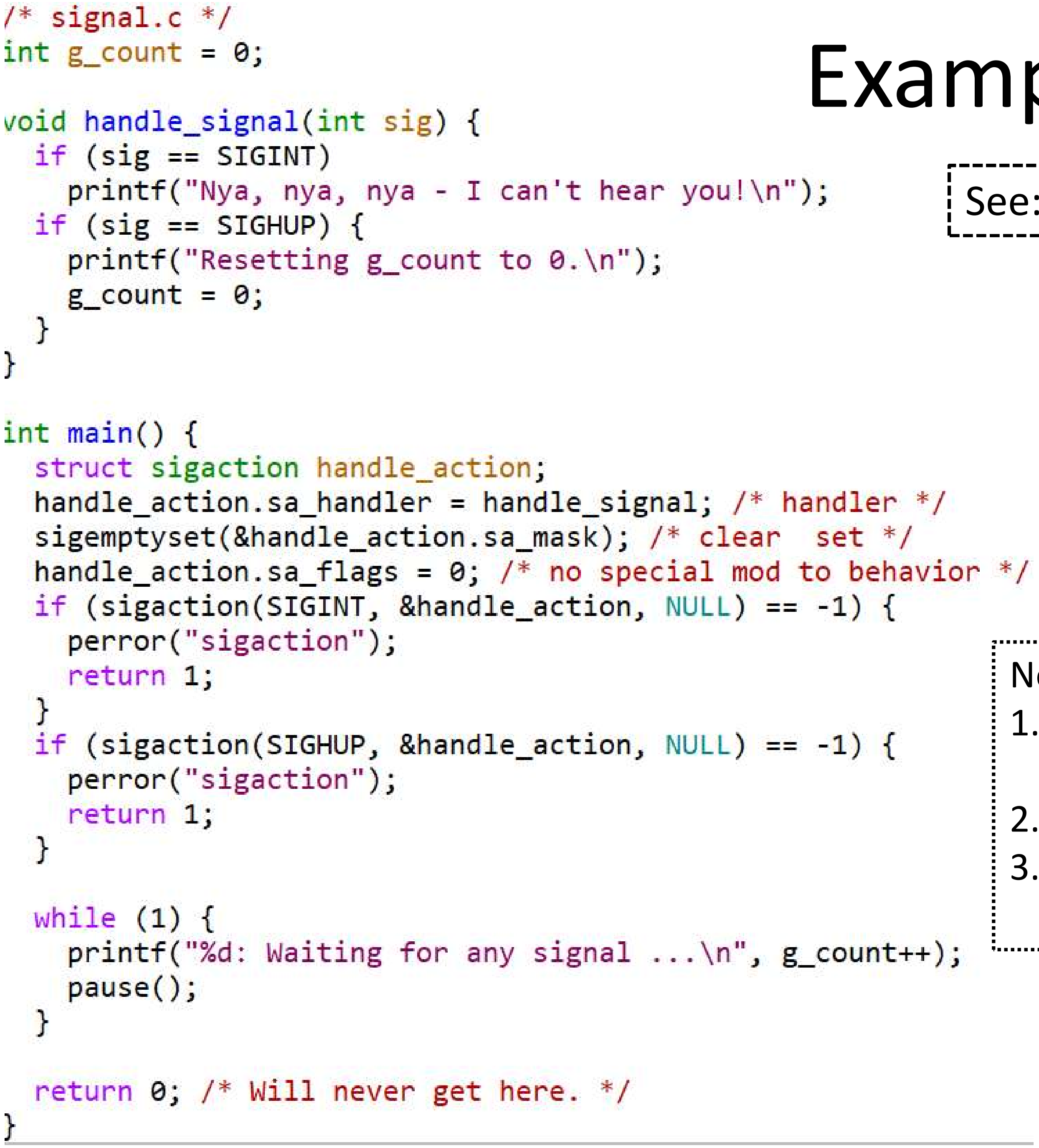
Handle

`sigaction()` - change behaviour for when signal arrives

See: "man 7 signal"

Example - Signal

See: “signal.c”



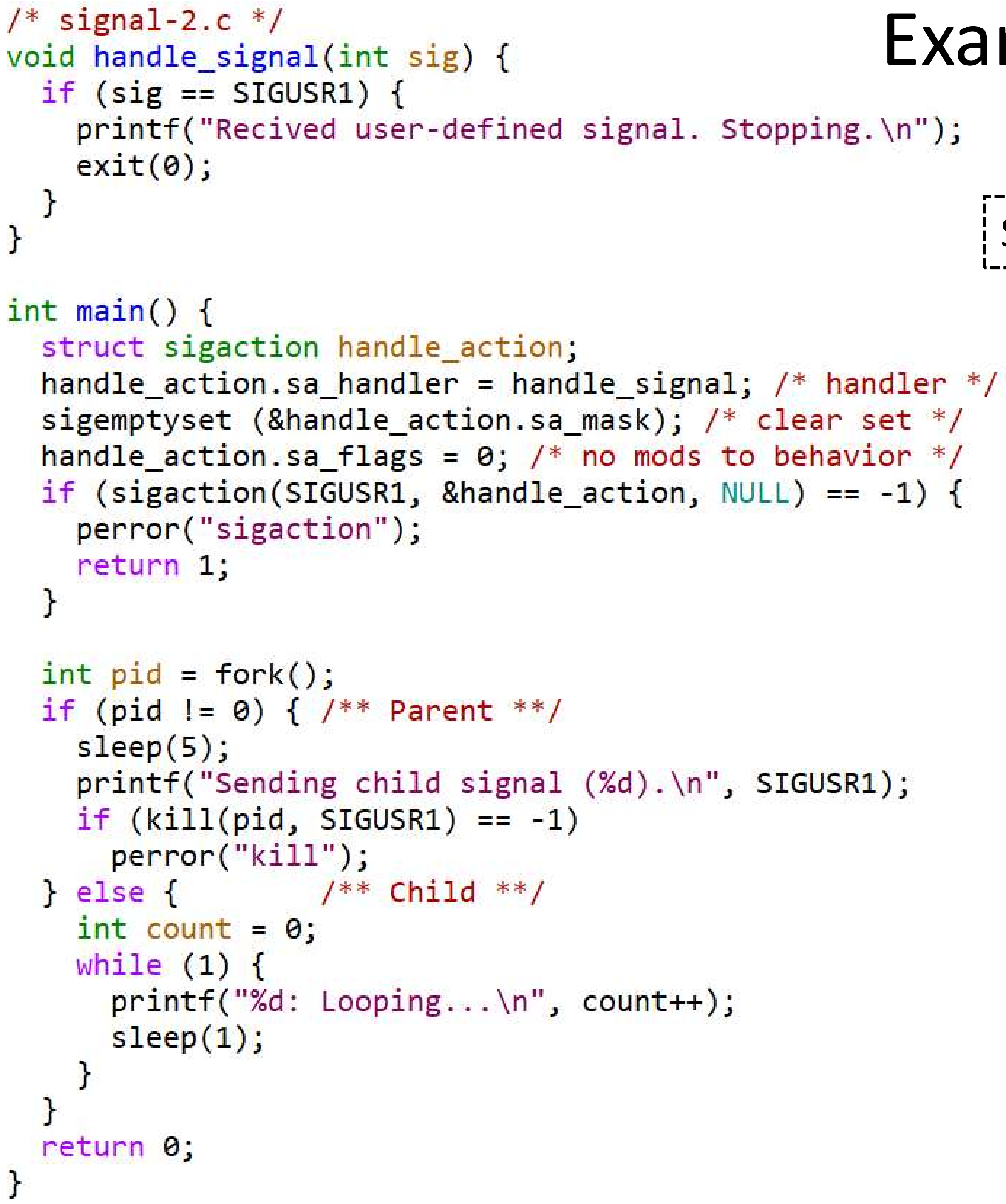
```
/* signal.c */
int g_count = 0;

void handle_signal(int sig) {
  if (sig == SIGINT)
    printf("Nya, nya, nya - I can't hear you!\n");
  if (sig == SIGHUP) {
    printf("Resetting g_count to 0.\n");
    g_count = 0;
  }
}

int main() {
  struct sigaction handle_action;
  handle_action.sa_handler = handle_signal; /* handler */
  sigemptyset(&handle_action.sa_mask); /* clear  set */
  handle_action.sa_flags = 0; /* no special mod to behavior */
  if (sigaction(SIGINT, &handle_action, NULL) == -1) {
    perror("sigaction");
    return 1;
  }
  if (sigaction(SIGHUP, &handle_action, NULL) == -1) {
    perror("sigaction");
    return 1;
  }

  while (1) {
    printf("%d: Waiting for any signal ...\n", g_count++);
    pause();
  }

  return 0; /* Will never get here. */
}
```

Note, *handling* is like interrupt

1. Store state/location where process was (stack)
2. Move to handler
3. When handler done, return to previous location

Example – Signal-2

See: “signal-2.c”



```c
/* signal-2.c */
void handle_signal(int sig) {
  if (sig == SIGUSR1) {
    printf("Recived user-defined signal. Stopping.\n");
    exit(0);
  }
}

int main() {
  struct sigaction handle_action;
  handle_action.sa_handler = handle_signal; /* handler */
  sigemptyset (&handle_action.sa_mask); /* clear set */
  handle_action.sa_flags = 0; /* no mods to behavior */
  if (sigaction(SIGUSR1, &handle_action, NULL) == -1) {
    perror("sigaction");
    return 1;
  }

  int pid = fork();
  if (pid != 0) { /** Parent **/
    sleep(5);
    printf("Sending child signal (%d).\n", SIGUSR1);
    if (kill(pid, SIGUSR1) == -1)
      perror("kill");
  } else {         /** Child **/
    int count = 0;
    while (1) {
      printf("%d: Looping...\n", count++);
      sleep(1);
    }
  }
  return 0;
}
```

Defined Signals

SIGABRT	Process abort signal.
SIGALRM	Alarm clock.
SIGFPE	Erroneous arithmetic operation.
SIGHUP	Hangup.
SIGILL	Illegal instruction.
SIGINT	Terminal interrupt signal.
SIGKILL	Kill (cannot be caught or ignored).
SIGPIPE	Write on pipe no one to read it.
SIGQUIT	Terminal quit signal.
SIGSEGV	Invalid memory reference.
SIGTERM	Termination signal.
SIGUSR1	User-defined signal 1.
SIGUSR2	User-defined signal 2.
SIGCHLD	Child process terminated
SIGCONT	Continue executing, if stopped.
SIGSTOP	Stop (cannot be ignored).
SIGTSTP	Terminal stop signal.
SIGTTIN	Background attempt read.
SIGTTOU	Background attempting write.
SIGBUS	Bus error.
SIGPOLL	Pollable event.
SIGPROF	Profiling timer expired.
SIGSYS	Bad system call.
SIGTRAP	Trace/breakpoint trap.
SIGURG	High bandwidth data at socket.
SIGVTALRM	Virtual timer expired.
SIGXCPU	CPU time limit exceeded.
SIGXFSZ	File size limit exceeded.

See man pages for details

Outline

- Introduction (done)
- Examples (done)
  - Shared Memory (done)
  - Files (done)
  - Pipes (done)
  - Signals (done)