CMPUT-379 Lab 2

Akemi Izuko, Armaan Katyal, Ellis McDougald, Han Yang, Patrick Zijlstra, Shasta Johnsen-Sollos, Steven Oufan Hai, Zhaoyu Li

Today’s lab

- Pipes
- Shared memory
  - Producer/consumer example
- Signals
- Using system calls in c programs
- Environment variables in Linux
- Review of assignment 1

Pipes

- Interprocess Communication (IPC)
- Half-duplex (unidirectional)
- Inherited by child processes

- For reference: Ch. 15 APUE.3e

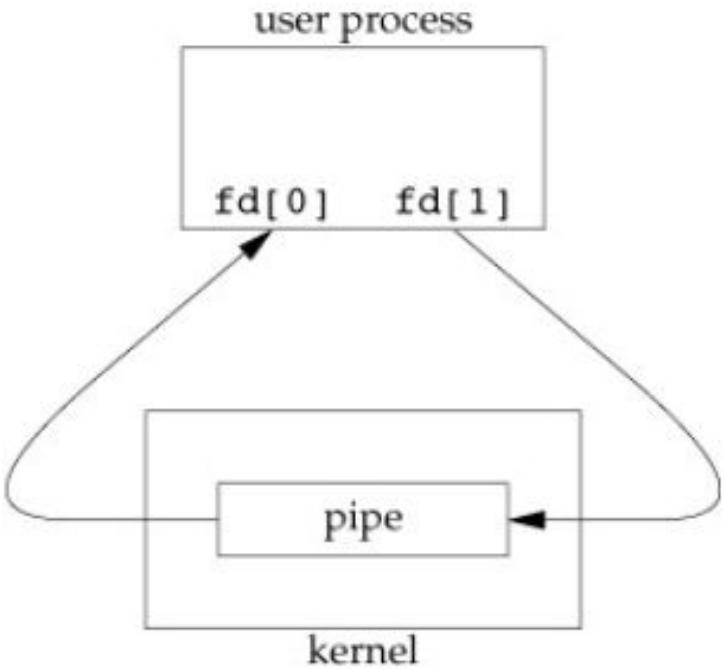
System call `pipe()`

- Creates a unidirectional data channel for interprocess communication (IPC)
- Example code - pipe.c

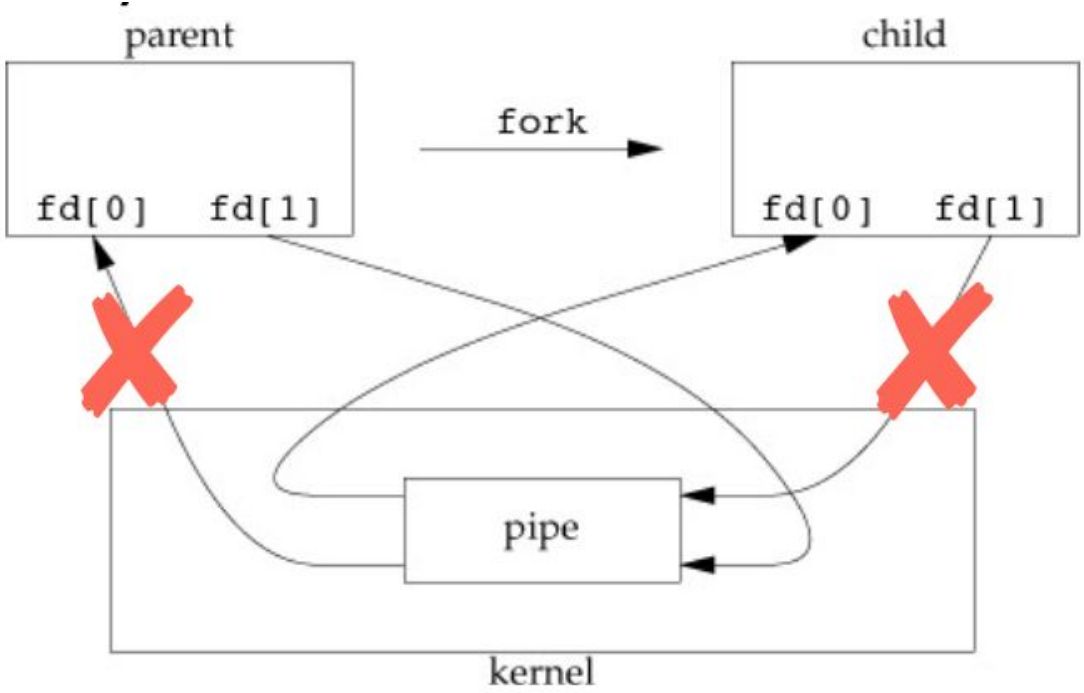
```
int pipe(int pipefd[2]);
```

- Parameter:
  - pipefd - gets populated with the file descriptors of the r/w ends
- Return Value:
  - -1 - error, sets errno
  - 0 - success

Pipe - talk to self



Pipe - talk to child process



Pipe - redirect child's output to parent

```
int main (int argc, char *argv[]) {
    char buf[MAXBUF];
    int n, status, fd[2];
    pid_t pid;
    if (pipe(fd) < 0) perror("pipe error!");
    if ((pid = fork()) < 0) perror("fork error!");
    if (pid == 0) {
        close(fd[0]);                // child won't read
        dup2(fd[1], STDOUT_FILENO); // stdout = fd[1]
        close(fd[1]);                // stdout is still open
        if (execl("/usr/bin/w", "w", (char *) 0) < 0) perror("execl error!"); //
don't use this
    } else {
        close(fd[1]);                // parent won't write
        while ((n= read(fd[0], buf, MAXBUF)) > 0)
            write(STDOUT_FILENO, buf, n);
        close(fd[0]);
        wait(&status);}
```

$ w | wc -w

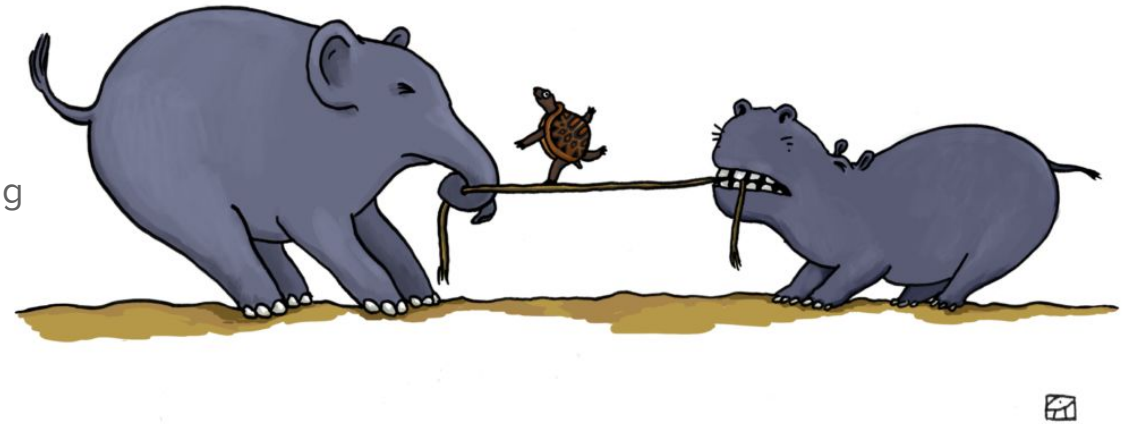
```
int main (int argc, char *argv[]) {
    int fd[2]; pid_t pid;

    if (pipe(fd) < 0) perror("pipe error!");
    if ((pid = fork()) < 0) perror("fork error!");
    if (pid == 0) {
        close(fd[1]);                   // child won't write
        dup2(fd[0], STDIN_FILENO);      // stdin = fd[0]
        close(fd[0]);                   // stdin is still open
        if (execl("/usr/bin/wc", "wc", "-w", (char *) 0) < 0)
            perror("execl error!");
    } else {
        close(fd[0]);                   // parent won't read
        dup2(fd[1], STDOUT_FILENO);      // stdout = fd[1]
        close(fd[1]);                   // stdout is still open
        if (execl("/usr/bin/w", "w", (char *) 0) < 0)
            perror("execl error!");
    }
    return 0;
}
```

Interprocess Communication (IPC)

- A process is **Independent** if it does not share data
- A process is **Cooperating** if it can affect or be affected by the other processes executing in the system

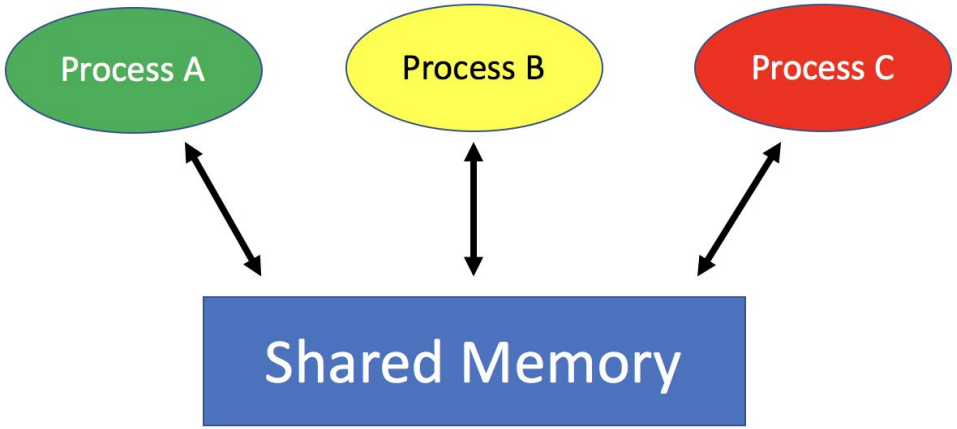
Information Sharing

- Several applications may be interested in the same information
- We must provide an environment to allow concurrent access
- Cooperating processes require an IPC mechanism to allow for data exchanges
- Two models
  - Shared Memory
  - Message Passing
- Several ways
  - mmap (shared memory)
  - System V message passing
  - Send signals
  - Pipes
  - Sockets



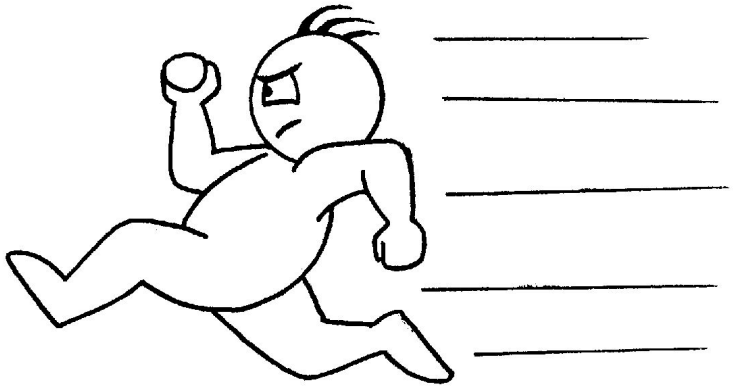
Shared Memory

- A shared region of memory is established
- Processes can exchange information by reading and writing to this region



Shared Memory

- Shared memory is typically faster than message passing due to less system calls and kernel intervention
- System calls are only required to establish shared memory regions
- Once established, all accesses are treated as routine memory accesses



Functions for shared memory

```c
#include <sys/mman.h>
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */

int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

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

```
LINK WITH -lrt!!!!!!!! (real-time library)
```

Shared Memory Example - producer.c

```c
int main() {
    const int SIZE = 4096;                                  /* size (B) of shared memory object */
    const char* name = "/PC";                               /* name of the shared memory object */

    const char* message_0 = "Hello";
    const char* message_1 = "World!";

    int shm_fd;                                             /* shared memory file descriptor */
    void* ptr;                                              /* pointer to shared memory object */

    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);        /* create the shared memory object */
    ftruncate(shm_fd, SIZE);                                /* configure size of the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0); /* memory map the shared memory object */

    sprintf(ptr, "%s", message_0);                          /* write to the shared memory object */
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);

    return 0;
}
```

Shared Memory Example - consumer.c

```c
int main() {
    const int SIZE = 4096;                  /* size (B) of shared memory object */
    const char* name = "/PC";               /* name of the shared memory object */

    int shm_fd;                             /* shared memory file descriptor */
    void* ptr;                              /* pointer to shared memory object */

    shm_fd = shm_open(name, O_RDONLY, 0666); /* open the shared memory object */

    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0); /* memory map the shared memory object */

    printf("%s", (char*)ptr);               /* read from the shared memory object */
    shm_unlink(name);                       /* remove the shared memory object */

    return 0;
}
```

Shared Memory Concurrency

- We do not want two processes to access the shared memory at the same time
- Semaphores are a signaling mechanism that act similar to a lock

Shared Memory Concurrency

Good Case

```
 Thread 1              Thread 2
----------------------------------------
1. Load to Reg. (0)
2. Incr. Reg. (1)     doing something
3. Store Reg. (1)          else


                      1. Load to Reg. (1)
  doing something     2. Incr. Reg. (2)
      else            3. Store Reg. (2)

Result:  counter = 2, as expected!
```

Shared Memory Concurrency

```
 Thread 1                Thread 2
-------------------------------------
1. Load to Reg. (0)
                        1. Load to Reg. (0)
2. Incr. Reg. (1)
                        2. Incr. Reg. (1)
3. Store Reg. (1)
                        3. Store Reg. (1)

Result:  counter = 1
```

Shared Memory Concurrency

Solutions:

- Using **signals** to coordinates different processes
- FYI: Using **Semaphore** to restrict the maximum number of processes accessing the shared memory at the same time
- FYI: Using **inter-process mutex** to lock shared memory when a process is accessing
    - Mutex usually use for multithreading
    - To use mutex in multiprocessing, the mutex need to be allocated in shared memory

Signal

- Software interrupt
- Use **signal()** or **sigaction()** to install signal handlers
- **int kill(pid_t pid, int sig)** sends a signal

- For reference: Ch. 10 APUE.3e

Signal Handling with **signal()**

```
#include <signal.h>

sighandler_t signal(int signum, sighandler_t handler);

// Sets the disposition of signal signum to handler
```

- Parameters:
  - signum: Specifies the signal
  - handler: A function that takes a single int and returns nothing
- Return Value:
  - On success: The previous value of the signal handler
  - On failure: SIG_ERR

Portability Problem: Behavior across UNIX versions may vary

Signal handling with **signal()**

```c
int  i;
void quit(int signum) {
    fprintf(stderr, "\nInterrupt (code= %d, i= %d)\n", signum, i);
}
int main () {
    if(signal(SIGQUIT, quit) == SIG_ERR)
        perror("can't catch SIGQUIT");
    for (i= 0; 1; i++) {
        usleep(1000);
        if (i % 100 == 0) putc('.', stderr);
    }
    return(0);
 } // signal/signal2.c
```

```
$ ./signal2
........^\
Interrupt (code= 3, i=
752)
.......^\
Interrupt (code= 3, i=
1416)
........^\
Interrupt (code= 3, i=
2336)
..............^C
```

Signal Handling with **sigaction()**

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);

// Changes the action taken by a process on a receipt of a
specific signal
```

- Parameters:
  - **signum**: Specifies the signal - Cannot be **SIGKILL** or **SIGSTOP**
  - **act**: If non-NULL, this is the new action for signal **signum**
  - **oldact**: If non-NULL, the previous action is saved in **oldact**
- Return Value:
  - 0 - success
  - -1 - error

sigaction struct

```
struct sigaction {
    void (*sa_handler)(int);      // signal handler function
                                  // or SIG_IGN (ignore)
                                  // or SIG_DFL (default)
    sigset_t sa_mask;             // additional signals to block
    int sa_flags;                 // signal options
    void (*sa_handler)(int, siginfo_t *, void *) // alternate handler
}
```

siginfo struct

- Set SA_SIGINFO to use alternate handler to get information about signal

```
struct siginfo{
    int si_signo;          // signal number
    int si_errno;          // errno value
    int si_code;           // more specific signal category
    pid_t si_pid;          // sending process id
    uid_t si_uid;          // sending process user id
    void *si_addr;         // address that caused signal
    ...
}
```

Signal handling with **sigaction()**

```
$ ./signal1
^CCaught signal!
^CCaught signal!
^CCaught signal!
```

```c
// signal/signal1.c
void signal_callback_handler(int signum) {
    printf("Caught signal!\n");
}

int main() {
    struct sigaction sa;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);          // clears
    sa.sa_handler = signal_callback_handler;
    sigaction(SIGINT, &sa, NULL);
    // sigaction(SIGTSTP, &sa, NULL);
    while (1) {}
}
```

The sigset_t data type is an array of booleans representing a set of signals


```c
sigemptyset(&s);
```

Using system calls

- For this assignment you have to use system calls to implement your shell. Continuing from last week, here are a few more:
  - `open()`
  - `close()`
  - `write()`
  - `read()`
  - `dup2()`
  - `kill()`
  - `nice()`
  - `mmap()`
  - `munmap()`

- `Discussed earlier:`
  - `pipe()`
  - `signal()`
  - `sigaction()`

System call open( )

- Open a file

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

#include <fcntl.h>

int open(const char *path, int oflags, mode_t mode);
```

- Parameter:
  - oflags - O_RDONLY, O_WRONLY,O_RDWR, O_APPEND, O_CREAT, etc.
  - mode   - S_IRUSR, S_IWUSR, S_IXUSR, etc.
- Return Value:
  - -1   - error
  - Others - file descriptor

System call `close()`

- Close a file

```
#include <unistd.h>

int close(int fildes);
```

- Parameter:
  - fildes - The file descriptor to be closed
- Return Value:
  - -1 - error
  - 0 - success

System call `write()`

- Write to a file descriptor
- Example code - write.c

```
#include <unistd.h>

ssize_t write(int fd, const void* buf, size_t count);
```

- Parameter:
  - fd - The file descriptor to be written to
  - buf - The buffer to write to the fd
  - count - The number of bytes to write to the fd
- Return Value:
  - -1 - error
  - Number of bytes written - success

System call `read()`

- Read from a file descriptor
- Example code - read.c

```
#include <unistd.h>

ssize_t read(int fd, const void* buf, size_t count);
```

- Parameter:
  - fd - The file descriptor to be read from
  - buf - The buffer to read the contents in fd into
  - count - The number of bytes to read from the fd
- Return Value:
  - -1 - error
  - Number of bytes read - success (0 on EOF)

System call dup2( )

- Duplicates one file descriptor, making them aliases, and then deleting the old file descriptor
- Example code - dup2.c

```
#include <unistd.h>

int dup2(int fildes, int fildes2);
```

- Parameter:
  - fildes - source file descriptor
  - fildes2 - target file descriptor
- Return Value:
  - <0   - error
  - Others - second file descriptor

System call `kill()`

- Send signal to a process
- Example code - kill.c

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

#include <signal.h>

int kill(pid_t pid, int sig);
```

- Parameter:
  - pid - target process
  - sig - signal want to send
- Return Value:
  - 0 - success
  - -1 - error

System call `nice()`

- Change process priority (higher nice value ➜ lower priority)

```
#include <unistd.h>

int nice(int incr);
```

- Parameter:
  - incr - amount to add to current nice value
- Return Value:
  - new nice value - success
  - -1 - error

System call `mmap()`

- Map files into memory

```
#include <sys/mman.h>

int mmap(void *addr, size_t len, int prot, int flag, int fd, off_t
off);
```

- Parameter:
  - addr - start address of mapping
  - len - length of mapping
  - prot - desired memory protection of mapping
  - flag - behaviour when updating mapping
  - fd - file to get content from
  - off - offset to start from within file
- Return Value:
  - start address of mapped region - success
  - MAP_FAILED - error

System call `munmap()`

- Unmap files from memory

```
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

- Parameter:
  - addr - start address of mapping
  - len - length of mapping
- Return Value:
  - 0 - success
  - -1 - error

Environment Variables in Linux

- Named strings that are accessible by all applications (“ultra global” variables)
- Typically used for defining environment specific configurations

  ```
  NAME=value
  ```

- Common env variables that are set automatically:
  - PATH: list of paths to search for executable files
  - HOME: home directory
  - etc.

- For reference: Section 7.9 of APUE.3e

Modifying Environment Variables

- Places the provided string of form `name=value` into the environment list
- Always overwrites if variable exists: if name already exists, replaces with new value

```
#include <stdlib.h>

int putenv(char *str);
```

- Parameter:
  - str - string to place into environment list
- Return Value:
  - 0 - success
  - !0 - error

Modifying Environment Variables

- Sets the value for an environment variable
- Can overwrite existing variables if desired

```
#include <stdlib.h>

int setenv(const char *name, const char *value, int rewrite);
```

- Parameter:
  - name - name of environment variable
  - value - value to set
  - rewrite - whether to overwrite an existing definition
- Return Value:
  - 0 - success
  - -1 - error

Assignment 1

- Build an interactive command line shell (dragonshell)
- Gain experience with system calls for process management and IPC
- Remember: for core features, only use system call functions from Section 2 of the man pages
- Familiarize yourself with the rubric on eclass

- Due date:
  - **October 1, 2025 at 5:59 pm**

Assignment 1 - high level approach

Make it easy for yourself! Do one step at a time, don't try to do pieces end-to-end

1. Read line of input into array of strings
2. Construct command struct
3. Validate command struct for inconsistency
4. Run command struct in one of 3 cases: (builtin, pipe, background + redirect)

```
typedef struct Cmd {
  char* cmd;
  char** args;
  int read_from; // fd for read
  int write_to; // fd for write
  bool is_background;
  bool is_builtin;
  struct Cmd* pipe_to;
} Cmd;
```

Quiz

Write a program that takes in a single argument, then prints out the unique files in that command sorted. You must use pipe(2) twice!

Similar to running the following in bash... where $1 is the first argument in argv

```
cat $1  | sort | uniq
```