CMPUT 379 Lab

ETLC E1003: Tuesday, 5:00 – 7:50 PM.

Tianyu Zhang, Peiran Yao

CAB 311: Thursday, 2:00 – 4:50 PM.

Max Ellis, Aidan Bush

Last Week...

- Got familiar with the programming environment
- Basic steps for Unix system programming
- System calls used in Assignment 1

execve Review

```
int pid;
char *argv[]={"ls", "-al", NULL};
char *env[]={NULL};
if ((pid = fork()) ==-1)
    perror("fork error");
else if (pid == 0)
    if(execve("/bin/ls", argv, env) == -1)
        perror("execve");
```

Today’s Lab

- Handling processes
- Sending and handling signals
- Get familiar with Interprocess Communication (IPC)
- Information Sharing
- More Pipes

Process Control

- Each process gets its own section of memory
- Copy values of all variables in parent process when **fork()**
- Processes are running parallel
  - hard to tell which process produce which line of code first

Process Control - variable

```
hello world (pid:21719)
Parent of 21720 (pid:21719), shared_int = 0
Pid 21719, shared_int = -1
Child (pid:21720), shared_int = 0
Pid 21720, shared_int = 1
```

```c
// process-control/p1.c
int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int shared_int=0;
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        _exit(1);
    } else if (rc == 0) { // child (new process)
        printf("Child (pid:%d), shared_int = %d\n", (int) getpid(), shared_int);
        ++shared_int;
    } else { // parent goes down this path (original process)
        printf("Parent of %d (pid:%d), shared_int = %d\n", rc, (int) getpid(), shared_int);
        --shared_int;
    }
    printf("Pid %d, shared_int = %d\n", (int) getpid(), shared_int);
    return 0;
}
```

Process Control - order

```c
// process-control/p2.c

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

int main(int argc, char *argv[]) {
    for (int i = 0; i < 5; i++) {
        if (fork() == 0) {
            printf("Child %d with pid %d\n", i, getpid());
            exit(0);
        }
    }
    printf("hello\n");
    return 0;
}
```

```
Child 0 with pid 23513
hello
Child 1 with pid 23514
Child 2 with pid 23515
Child 3 with pid 23516
Child 4 with pid 23517
```

```
Child 0 with pid 23519
Child 1 with pid 23520
hello
Child 2 with pid 23521
Child 3 with pid 23522
Child 4 with pid 23523
```

```
Child 0 with pid 23710
Child 1 with pid 23711
hello
Child 3 with pid 23713
Child 2 with pid 23712
Child 4 with pid 23714
```

Process Control - fork and execve

```
printf("hello world (pid:%d)\n", (int) getpid());
int rc = fork();
if (rc < 0) { // fork failed; exit
    fprintf(stderr, "fork failed\n"); exit(1);
} else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[4]; char *env[] = {NULL};
    myargs[0] = strdup("/usr/bin/wc");   // program: "wc" (word count)
    myargs[1] = strdup("-w");   // argument: print the word counts
    myargs[2] = strdup("p3.c"); // argument: file to count
    myargs[3] = NULL;           // marks end of array
    if (execve(myargs[0], myargs, env) < 0)  // runs word count
        perror("execve");
    printf("this shouldn't print out\n");
} else { // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc, (int) getpid());
} // process-control/p3.c
```

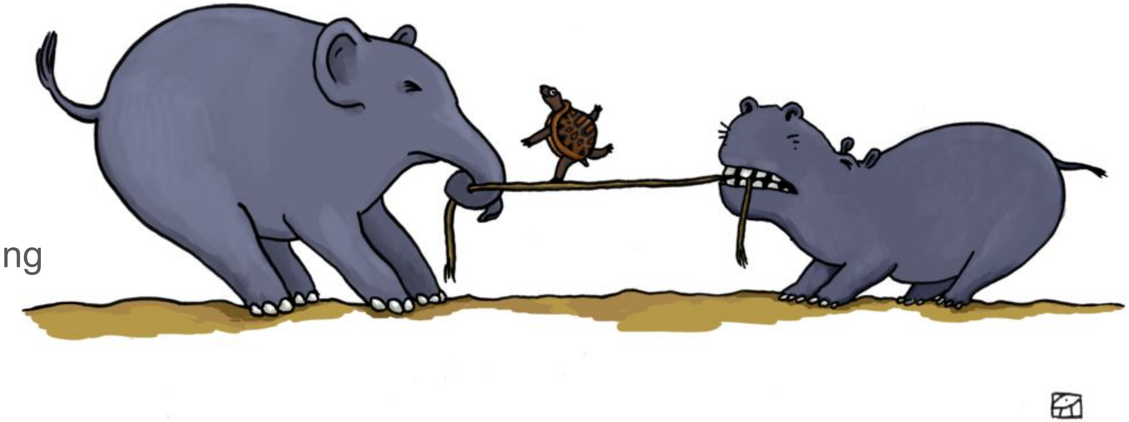
```
hello world (pid:8293)
hello, I am child
(pid:8294)
     141 p3.c
hello, I am parent of
8294 (wc:8294) (pid:8293)
```

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

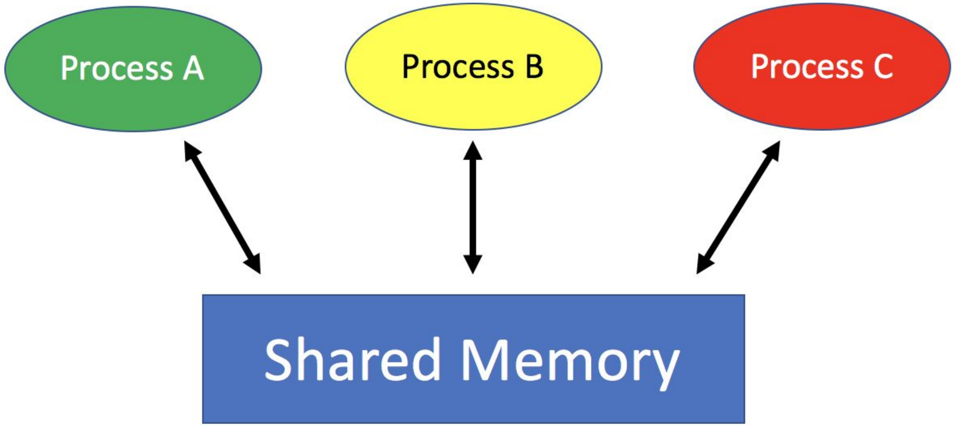
Information Sharing

- 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



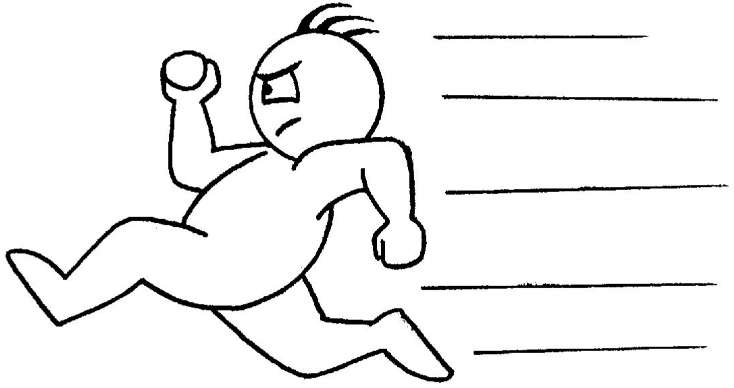
Shared Memory

- 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

```
#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);
```

```
#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!!!!!!!!
```

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

```
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

Semaphores

```
int    sem_close(sem_t *sem);
int    sem_destroy(sem_t *sem);
int    sem_getvalue(sem_t *sem, int *sval);
int    sem_init(sem_t *sem, int pshared, unsigned int value);
sem_t *sem_open(const char *name, int oflag, ...);
int    sem_post(sem_t *sem);
int    sem_trywait(sem_t *sem);
int    sem_unlink(const char *name);
int    sem_wait(sem_t *sem);
```

Signal

- Use **kill()** to send a signal
- Use **signal()** or **sigaction()** to install signal handlers

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

<u>Portability Problem: Behavior across UNIX versions may vary</u>

Signal handling with **signal()**

```
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

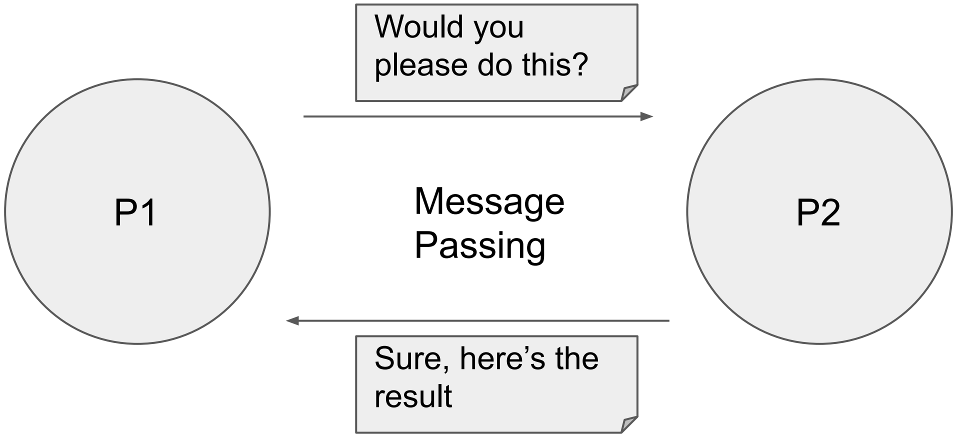
Signal handling with **sigaction()**

```
// 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);
    sa.sa_handler = signal_callback_handler;
    sigaction(SIGINT, &sa, NULL);
    // sigaction(SIGTSTP, &sa, NULL);
    while (1) {}
}
```

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

Message Passing

- Allows processes to send messages between each other
- Can have messages formatted into nice to use containers
- Handles ordering of messages



Message Passing functions

Generate unique key: **ftok**

Get message queue identifier: **msgget**

Send message: **msgsnd**

Receive message: **msgrcv**

Message queue controls: **msgctl**

Message Passing Example - writer.c

```c
struct mesg_buffer { // structure for message queue
    long mesg_type;
    char mesg_text[100];
} message;

int main() {
    key_t key = ftok("progfile", 65); // generates a unique key
    int msgid = msgget(key, 0666 | IPC_CREAT); // creates a message queue
    message.mesg_type = 1; // specify a message type
    printf("Write Data: ");
    scanf("%s", message.mesg_text); // read text from stdin to the message body
    msgsnd(msgid, &message, sizeof(message), 0); // sends the message
    printf("Data sent is: %s \n", message.mesg_text);
    return 0;
} // message-passing/writer.c
```

Message Passing Example - reader.c

```
struct mesg_buffer { // structure for message queue
    long mesg_type;
    char mesg_text[100];
} message;

int main() {
    key_t key = ftok("progfile", 65); // generates a unique key
    int msgid = msgget(key, 0666 | IPC_CREAT); // gets the message queue with the unique key
    msgrcv(msgid, &message, sizeof(message), 1, 0); // receives the message
    printf("Data received is: %s \n", message.mesg_text);
    msgctl(msgid, IPC_RMID, NULL); // destroys the message queue
    return 0;
} // message-passing/reader.c
```

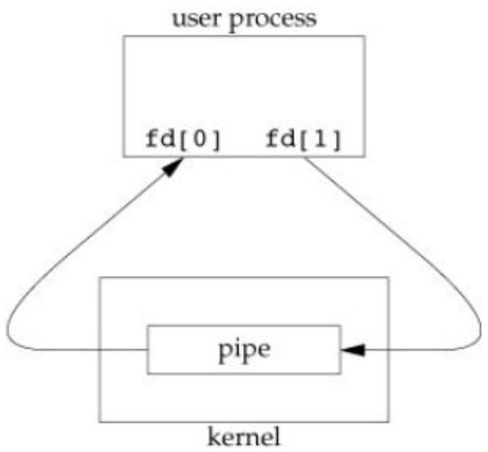
Pipe

```
#include <unistd.h>

int pipe(int pipefd[2]);

// creates a unidirectional data channel for interprocess
communication
```

- Parameter:
    - Pipefd - two file descriptors, read/write ends of the pipe
- Return Value:
    - -1 - error
    - 0 - success

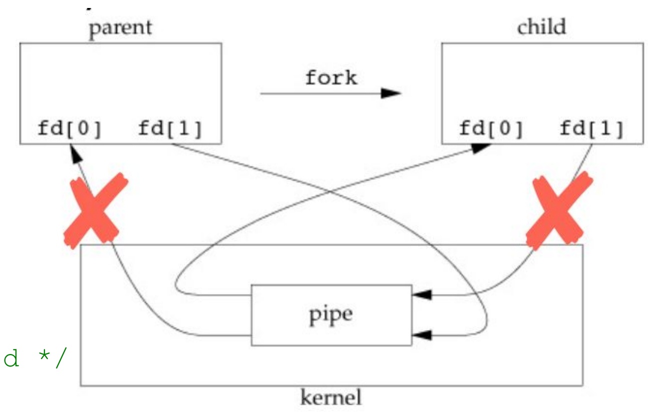
Pipe - talk to self

```c
#define MSG_SIZE 5
enum PipeSel {rd_pipe = 0, wt_pipe = 1};
char* first = "msg1"; char* second = "msg2";
int main(void) {
    int fd[2];
    char line[MSG_SIZE];
    if (pipe(fd) < 0)                    /* create a pipe */
        perror("pipe error");

    write(fd[wt_pipe], first, MSG_SIZE); /* write to the write end */
    write(fd[wt_pipe], second, MSG_SIZE); /* write to the write end */

    read(fd[rd_pipe], line, MSG_SIZE); /* read from the read end*/
    printf("%s\n", line);              /* print the first message */
    read(fd[rd_pipe], line, MSG_SIZE); /* read from the read end */
    printf("%s\n", line);              /* print the second message */
    return 0;
} /* ordinary-pipe/pipe1.c */
```



Pipe - talk to child process

```
#define MAXLINE 128
int main(void) {
    int n; pid_t pid;
    int fd[2];
    char line[MAXLINE];
    if (pipe(fd) < 0)            /* create a pipe before forking a child */
        perror("pipe error");
    if ((pid = fork()) < 0) {    /* fork a child */
        perror("fork error");
    } else if (pid > 0) {        /* parent continues */
        close(fd[0]);            /* close the unused end of the pipe */
        write(fd[1], "hello world!", 13);
    } else {                     /* child continues */
        close(fd[1]);            /* close the unused end of the pipe */
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);    }
    _exit(0);
} /* ordinary-pipe/pipe2.c */
```



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!");
    } 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);}
    return 0;     } /* ordinary-pipe/pipe3.c */
```

Pipe - Word Count of 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;
}
```

Pipe - **popen / pclose**

```c
#define LINESIZE 20

int main (int argc, char *argv[]) {
    char buf[LINESIZE];
    FILE *fp;

    fp = popen("ls -l", "r");

    while (fgets(buf, LINESIZE, fp) != NULL)
      printf("%s\n", buf);
    pclose(fp);
    return 0;
}
```

Pipe - redirect command output to a file

```
int main (int argc, char *argv[]) {
    int ffd;
    if((ffd = open("a.txt", O_CREAT | O_RDONLY, 0644)) < 0)
        perror("open failed!");

    dup2(ffd, STDIN_FILENO);    // stdin = ffd
    close(ffd);                 // stdin is still open
    if (execl("/usr/bin/wc", "wc", "-w", (char *) 0) < 0)
        perror("execl error!");

    return 0;
}
```

Practice 1

Write two programs that implement the Producer Consumer problem, for both **pipes** and **message passing**. The producer will read from a file and send to the consumer who will write to stdout.

Hint: Start from the example code in the slides.

Practice 2

Write a program **primes.c** that is supposed to find all primes between a and b (inclusive with maximum b of 40,000,000) **using n processes**, and write the result to a local output **out.txt** with **sorted order** (parent wait until all children finished and then sort). You can safely assume the number of primes < 2,500,000.

Measure the speedup for n=1, 2, 3, 4, 5, 6, 7, 8 on the lab computers for a = 1 and b = 40,000,000 and discuss what the speedup results mean.

Hint:

> Use **fork()** to create processes and use **wait()** in the parent.
>
> Use **pipe()** or any other IPC methods to send the results back the the main process.