Sistemas Operacionais
Roteiro de Laboratório 3

1 Objetivos

O objetivo deste laboratório é ter um contato prático com dois mecanismos de comunicação entre processos: *pipes* e memória compartilhada.

2 *Pipes*

Um *pipe* é uma forma de comunicação entre dois processos. *Pipes* foram um dos primeiros mecanismos de IPC (*inter process communication*) criados para o UNIX. Em geral, usar um *pipe* é a forma mais simples para processos trocarem informações. Diferentes SOs proveem diferentes tipos de *pipes*. Neste laboratório vamos estudar o tipo mais simples: *pipes* anônimos (*unnamed pipes*).

Pipes anônimos permitem que dois processos se comuniquem conforme o modelo padrão de produtor-consumidor: o produtor escreve em uma ponta do *pipe* (o *write-end*) e o consumidor lê da outra ponta (o *read-end*). Dessa forma, *pipes* anônimos são unidirecionais, permitindo somente a comunicação em um sentido. Se a comunicação em ambos os sentidos for necessária, é preciso empregar dois *pipes*, com cada *pipe* enviando dados em uma direção.

Em sistemas UNIX, *pipes* são construídos usando a função abaixo.

```
pipe(int fd[])
```

Esta função cria um *pipe* que é acessado através do vetor de descritores de arquivos `fd[]`: `fd[0]` é o lado de leitura e `fd[1]` é lado de escrita. Sistemas UNIX tratam *pipes* como um tipo especial de arquivo. Desta forma, *pipes* podem ser acessados através das *system calls* `read()` e `write()`.

Um *pipe* anônimo não pode ser acessado de fora do processo que o criou. Normalmente, um processo pai cria um *pipe* e o utiliza para se comunicar com um processo filho que vai ser criado via `fork()`. Lembre-se que um processo filho herda os arquivos abertos do pai. Como o *pipe* é um tipo especial de arquivo, o filho recebe uma cópia do *pipe* criado.

O programa abaixo ilustra o uso de *pipes*.

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

#define READ_END 0
#define WRITE_END 1

int main(void) {
    // Create the pipe.
    int fd[2];
    pipe(fd);

    pid_t pid = fork();
    if (pid > 0) { // Parent process.
```

```
        char *write_msg = "Greetings!";
        // Close the unused end of the pipe.
        close(fd[READ_END]);
        // Write to the pipe.
        write(fd[WRITE_END], write_msg, 10);
        // Close the write end of the pipe.
        close(fd[WRITE_END]);
    } else { // Child process.
        char read_msg[25];
        // Close the unused end of the pipe.
        close(fd[WRITE_END]);
        // Read from the pipe.
        read(fd[READ_END], read_msg, 25);
        // Close the read end of the pipe.
        close(fd[READ_END]);
        printf("[CHILD]: Parent says '%s'\n", read_msg);
    }

    return 0;
}
```

É interessante notar que no programa acima, tanto o processo pai quanto o filho começam fechando o lado do *pipe* que eles não vão utilizar. Embora isso não seja estritamente necessário, é um passo importante para garantir que um processo que esteja lendo do *pipe* receba a indicação de EOF quando o escritor fechar o outro lado do *pipe*.

3 *Shared Memory*

Conforme discutido nas aulas teóricas, o uso de memória compartilhada permite que dois ou mais processos se comuniquem. Vamos usar o modelo de produtor-consumidor como exemplo. O programa abaixo é o produtor, que cria a área de memória compartilhada e escreve alguns bytes nela.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/mman.h>

int main() {
    const char *name = "OS"; // Name of shared memory area.
    const int SIZE = 4096; // Size (in bytes) of area.
    int shm_fd; // Shared memory file descriptor.
    void *ptr; // Pointer to the shared memory area.
    const char *message0= "Hello";
    const char *message1= " World!";

    // Create the shared memory segment.
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

```
    // Configure the size of the shared memory segment.
    ftruncate(shm_fd,SIZE);
    // Map the shared memory segment to process address space.
    ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    // Now write to the shared memory region.
    // Note: we must increment the value of ptr after each write.
    sprintf(ptr, "%s", message0);
    ptr += 5;
    sprintf(ptr, "%s", message1);
    ptr += 7;

    return 0;
}
```

O programa consumidor abaixo se conecta à área compartilhada e lê os dados que foram guardados lá.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/mman.h>

int main() {
    const char *name = "OS"; // Name of shared memory area.
    const int SIZE = 4096; // Size (in bytes) of area.
    int shm_fd; // Shared memory file descriptor.
    void *ptr; // Pointer to the shared memory area.

    // Open the shared memory segment.
    shm_fd = shm_open(name, O_RDONLY, 0666);
    // Map the shared memory segment to process address space.
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    // Read from the shared memory region.
    printf("[CONSUMER]: received '%s'\n", ptr);
    // Remove the shared memory segment.
    shm_unlink(name);

    return 0;
}
```

Exemplo de compilação e execução.

```
$ gcc -o producer0 producer0.c -lrt
$ gcc -o consumer0 consumer0.c -lrt
$ ./producer0
$ ./consumer0
[CONSUMER]: received 'Hello World!'
```

É interessante notar que a comunicação é completamente assíncrona. O processo produtor termina antes do consumidor realizar a leitura. A área de memória compartilhada só deixa de existir quando for desalocada com `shm_unlink`.

4 Exercícios de programação

IMPORTANTE: os exercícios desta seção NÃO devem ser enviados para o AVA. Veja a seção Tarefa, a seguir.

1. Faça o *download* no AVA dos arquivos de exemplos. Compile-os e execute-os como indicado acima e discutido pelo professor no início da aula.

2. Desenvolva um programa que usa *pipes* aonde o processo pai envia uma *string* para o processo filho, e a seguir o filho inverte a caixa de cada caractere da mensagem e envia de volta para o pai. Por exemplo, se o pai enviar `"Hi, there."`, o filho deve retornar `"hI, THERE."`. Serão necessário dois *pipes*, um para enviar a mensagem do pai para o filho e outro para a mensagem de retorno no outro sentido.

5 Tarefa

Crie um modelo de produtor-consumidor utilizando memória compartilhada, aonde o produtor envia uma sequência de números primos para o consumidor, que os exibe na tela. O processo consumidor lê um número natural do teclado e envia esse número para o produtor. O número recebido pelo produtor indica a quantidade de primos que deve ser enviada para o consumidor. Se o usuário digitar 0, ambos processos terminam. Você deve projetar a área de memória compartilhada de forma que os processos consigam cooperar entre si e trocar as informações necessárias. *Obs: Você pode estabelecer um limite máximo de primos gerados.*

Exemplo de compilação e execução. (Ambos os processos devem ser executados simultaneamente em dois terminais separados.)

```
$ gcc -o producer1 producer1.c -lrt
$ ./producer1
[PRODUCER]: Ready to send primes.
[PRODUCER]: Sending 2 new primes.
[PRODUCER]: Sending 6 new primes.
[PRODUCER]: Finished.

$ gcc -o consumer1 consumer1.c -lrt
$ ./consumer1
[CLIENT]: Amount of primes to request: 2
[CLIENT]: Got the following primes: 2 3
[CLIENT]: Amount of primes to request: 6
[CLIENT]: Got the following primes: 5 7 11 13 17 19
[CLIENT]: Amount of primes to request: 0
[CLIENT]: Finished.
```

Envie a sua implementação pelo AVA na tarefa correspondente. O prazo (inadiável) para submissão é dia 26/09/2017 (terça-feira) às 23:55. Essa tarefa vale um *bit* (1 – fez corretamente dentro do prazo, 0 – caso contrário) na nota dos exercícios de laboratório.