AYUDANTÍA T2

Germán Leandro Contreras Sagredo
Ricardo Esteban Schilling Broussaingaray

IIC2333 [2019-1] - Sistemas Operativos y Redes

INTRODUCCIÓN

INTRODUCCIÓN

Los objetivos de esta ayudantía son:

1. Comprender el funcionamiento de la tarea.
2. Repasar la utilidad de **fork**, **exec** y **await**.
3. Presentar métodos para comunicar procesos independientes.
4. Repasar el uso de **pthreads**.

MAPREDUCE

MAPREDUCE

Para nuestra tarea, esperamos que usen una versión simplificada de `MapReduce`.

Podrán pensar cada proceso o thread como un **nodo**, con un trabajo particular, que puede ser de mapeo de input (**map**) o de agregación de múltiples mapeos (`reduce`).

El mapeo de input toma una fracción del input y lo procesa para poder ser agregado fácilmente en un nodo `reduce`.

FORK, EXEC, AWAIT

FORK

- Se usa para crear nuevos procesos que se ejecutan concurrentemente con el proceso padre o creador.
- Se ejecuta el mismo código desde la línea que se hizo **fork**.
- Es un proceso distinto.

```
void main() {
    int pid = fork();
    if (!pid) {
        printf("Nací!\n");
    } else {
        printf("Nuevo hijo %d!\n", pid);
    };
};
```

¿Cómo se si un proceso es padre o hijo?

Volvamos a ver el código anterior:

```
void main() {
    int pid = fork();
    if (!pid) {
        printf("Nací!\n");
    } else {
        printf("Nuevo hijo %d!\n", pid);
    };
};
```

En **pid** el padre recibe el **pid** del hijo, mientras que el hijo recibe un 0.

EXEC

Se usa para reemplazar todo el contenido de un proceso por otro programa.

Sirve principalmente si uno quiere ejecutar diferentes cosas después de haber hecho **fork**.

```
void main() {
    if (!fork()){
        printf("Naci!\n");
        char cmd[10] = "ls";
        char *args[2];
        args[0] = malloc(sizeof(char) * 10);
        strcpy(args[0], "-la");
        args[1] = NULL;
        execvp(cmd, args);
    };
};
```

En **pid** el padre recibe el **pid** del hijo, mientras que el hijo recibe un 0.

WAIT

Se usa para esperar la finalización de un proceso hijo.

Puede ser usado a modo de reasignar recursos escasos.

```
if ((pid = fork()) == 0) {
    sleep(2);
    exit(0);
} else do {
    if (((pid = waitpid(pid, &status, 1)) == 0) {
    // 0 si el hijo aun no termina.
        printf("Still running\n");
        sleep(1);
    } else {
        printf("Exit!\n");
    }
} while (pid == 0);
```

COMUNICACIÓN ENTRE PROCESOS

GETPPID

Se usa para obtener el pid del proceso padre.

Muy útil cuando queremos enviar señales a un proceso padre.

Tambien existe getpid() que nos entrega el **pid** del mismo proceso.

```
if ((pid = fork()) == 0) {
    printf("Hola soy %d y mi padre es %d", getpid(), getppid());
    exit(0);
} else {
    printf("Hola soy %d y mi hijo es %d", getpid(), pid);
    exit(0);
};
```

KILL

Se usa para enviar una señal a un proceso específico.

```
int main() {
    if (!fork()){
        sleep(1);
        kill(getppid(), SIGKILL);
        exit(0);
    } else {
        while(1){
            fprintf(stdout, "Correré para siempre >:)\n");
            usleep(100000);
        };
    };
};
```

Este código crea un proceso hijo, el cual le envía la señal SIGKILL para matarlo incondicionalmente.

FTOK

Una de las maneras de permitir comunicación entre procesos es usando memoria compartida.

Antes de pedir un segmento de memoria compartida, necesitamos generar una key única que nos dejará pedir memoria, lo cual se hace con la función `ftok(file, id)` donde `file` es un archivo (preferentemente vacío) e `id` es un id único que permite identificar el segmento de memoria.

SHMGET

Para asignar el segmento de memoria compartida, usamos la función `shmget(key, size, flags)`, donde `key` es la llave que obtenemos usando `ftok`, `size` es el tamaño de la memoria compartida que queremos y `flags` es un número que indica para que usaremos la memoria.

Recomendamos que en `flags` se use el valor `IPC_CREAT | SHM_W | SHM_R` el cual sirve para crear el segmento y darnos permisos para leerlo y modificarlo.

La función retorna un identificador para la memoria compartida.

Esta función toma el identificador obtenido usando `shmget` y adjunta la memoria a la memoria del mismo proceso, permitiendo accederla como uno accede a un puntero común y corriente.

Para llamar a la función, debemos hacer

```
shmat(identificador, NULL, 0);
```

La cual retorna un puntero a `void`, `(void*)`

SHMCTL

Esta función nos permite modificar las opciones de la memoria compartida. Para esta tarea usaremos la opción `IPC_RMID`, que nos permite eliminar el segmento luego que todos los procesos que lo usan mueren o lo sueltan.

Para llamar a la función, debemos hacer

```
shmctl(identificador, IPC_RMID, NULL);
```

Cuidado con esto, pues al usar esta función la memoria es marcada para ser eliminada y ya no es posible adjuntar la memoria usando `shmap`.

MEMORIA COMPARTIDA

```
# Proceso 1
int main() {
    key_t key = ftok("./memoria", 2019);
    long shmid = shmget(key, 2048, IPC_CREAT | SHM_W | SHM_R);
    char *str1 = (char*)shmat(shmid, NULL, 0);
    strcpy(str1, "Hola!\n");
    shmctl(shmid, IPC_RMID, NULL);
    // Esto debe pasar despues que el proceso 2 llama a shmat
};

# Proceso 2
int main() {
    key_t key = ftok("./memoria", 2019); // Igual al Proceso 1
    long shmid = shmget(key, 2048, IPC_CREAT | SHM_W | SHM_R);
    char *str1 = (char*)shmat(shmid, NULL, 0);
    printf("Data written in memory: %s\n", str1);
    // Data written in memory: Hola!
};
```

PTHREADS

CREAR UN THREAD

En C, un pthread nos permite correr una función concurrentemente, mientras compartimos recursos como memoria y tiempo en la CPU.

Veamos un código que crea threads para realizar una función simple.

```c
void *thread_funct(void *args){
    printf("Hola soy un simple thread\n");
    return NULL;
};

int main(){
    pthread_t my_thread;
    printf("Creando thread\n");
    pthread_create(&my_thread, NULL, thread_funct, NULL);
    printf("Thread creado\n");
    sleep(1);
    // Retornar en el main mata a todos los threads
    return 0;
};
```

En nuestra tarea, habrá momentos en el que algunos threads no hagan nada, esperando que otro thread termine su tarea antes de poder realizar la propia.

Para estos casos disponemos de funciones como `pthread_join` que sirve para esperar la terminación de un thread específico.

```
void *thread_funct(void *args){
    printf("Hola soy un simple thread\n");
    sleep(1);
    return NULL;
};
int main(){
    pthread_t my_thread;
    printf("Creando thread\n");
    pthread_create(&my_thread, NULL, thread_funct, NULL);
    printf("Thread creado, esperando\n");
    pthread_join(my_thread, NULL);
    printf("Adiós\n");
    return 0;
};
```

VALORES DE RETORNO

Es posible capturar el valor de retorno de un thread, usando la función `pthread_exit` y `pthread_join`.

```
void *thread_funct(void *args){
    char* string = malloc(sizeof(char) * 20);
    strcpy(string, "Hola soy un string\n");
    pthread_exit(string);
};
int main(){
    pthread_t my_thread;
    void* return_val;
    printf("Creando thread\n");
    pthread_create(&my_thread, NULL, thread_funct, NULL);
    printf("Thread creado, esperando\n");
    pthread_join(my_thread, &return_val);
    printf("Mensaje del thread: %s", (char *)return_val);
    printf("Adiós\n");
    return 0;
};
```

HEADER FILES

HEADERS

Comunes:

- stdio.h
- stdlib.h
- string.h
- unistd.h

Threads:

- pthread.h
  Para compilar: gcc -o <ejecutable> <codigo> -lpthread
- sys/types.h

HEADERS

Procesos:

- wait.h
- errno.h (Para errores)
- signal.h

Memoria compartida:

- sys/ipc.h
- sys/shm.h

FIN