

Operative Systems: Laboratory 5

Final Review

01-Processes-ExecCmds: A fin de ejecutar todos los comandos de la lista, debemos crear un proceso mediante el sys-call fork(), que ejecutará el comando execvp(). Una vez ejecutado este comando, el proceso padre esperará a los hijos y finalizará.

```
int main(int argc, char *argv[])
{
    int num_cmds = 2;
    char* cmds[][10] = {{ "ps", NULL }, { "ls", "-la", NULL }};

    for (int i=0; i<num_cmds; i++) {
        execvp(cmds[i][0], cmds[i]);
    }
    while (wait(NULL) > 0);
    return 0;
}
```

```
int main(int argc, char *argv[])
{
    int num_cmds = 2;
    char* cmds[][10] = {{ "ps", NULL }, { "ls", "-la", NULL }};

    for (int i=0; i<num_cmds; i++) {
        int pid = fork();
        if (pid == 0) {
            execvp(cmds[i][0], cmds[i]);
        }
        pid = wait(NULL);
    }
    return 0;
}
```

02-Processes-Sequential: Para cambiar el tipo de ejecución de concurrente a secuencial, debemos crear la variable PID, a la que asignaremos el valor de un fork(). El proceso hijo creado ejecutará el código del condicional if. Una vez ejecutado, el proceso padre esperará a los hijos y finalizará.

```
int main(int argc, char *argv[])
{
    for (int i=0; i<10; i++) {
        if(fork() == 0) {
            usleep(10000+rand()%10000);
            printf("I'm process %d\n",i);
            _exit(0);
        }
    }
    while (wait(NULL) > 0);
    return 0;
}
```

```
int main(int argc, char *argv[])
{
    int pid;
    for (int i=0; i<10; i++) {
        pid = fork();
        if(pid == 0) {
            usleep(10000+rand()%10000);
            printf("I'm process %d\n",i);
            _exit(0);
        }
        pid = wait(NULL);
    }
    return 0;
}
```

03-Threads-NoThreads: El objetivo de este ejercicio es conseguir el mismo resultado que en código original, pero sin crear threads. En el código original, se crean hilos que ejecutarán la función fsleep usando "&i" como parámetro, por lo tanto solo tenemos que sustituir las funciones pthread_create() i pthread_join() por la función fsleep(&i).

```
int main(int argc, char *argv[])
{
    pthread_t tids[NTHREADS];
    srand(time(NULL));

    for(int i=0; i<NTHREADS; i++) {
        printf("Creating thread %d\n", i);
        pthread_create(&tids[i], NULL, fsleep, (void*) &i);
        pthread_join(tids[i], NULL);
    }
}
```

```
int main(int argc, char *argv[])
{
    pthread_t tids[NTHREADS];
    srand(time(NULL));

    for(int i=0; i<NTHREADS; i++) {
        printf("Creating thread %d\n", i);
        fsleep(&i);
    }
}
```

04-Threads-Join: En este caso, la modificación se ha hecho en el paso de argumentos en la función `pthread_create`, anteriormente se pasaba la dirección de memoria de `i`, que iba cambiando en el bucle `for`. En este caso pasamos solo el valor de `i`.

```
void* fsleep(void* arg) {
    usleep(1);
    int id = (long) arg;
    printf("fsleep thread function ID: %d\n", id);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tids[NTHREADS];
    srand(time(NULL));

    for(int i=0; i<NTHREADS; i++) {
        printf("Creating thread %d\n", i);
        pthread_create(&tids[i], NULL, fsleep, (void*) (long) i);
    }

    for(int i=0; i<NTHREADS; i++) pthread_join(tids[i], NULL);
}
```

05-Synch-Sum:

En este caso, hemos creado un array de locks, este array es del mismo tamaño que el array `sum`, y cada vez que se accede a la posición `n` de nuestro array `sum`, se bloquea el lock de la posición `n` del array de locks.

```
pthread_mutex_t locks[NSUMS]; // Creamos un array de locks, para hacer el lock de cada posición.

void* fsum(void * arg) {
    pthread_mutex_lock(&locks[i%NSUMS]); // Hacemos lock de esa posición determinada del array.
    . . . .
    pthread_mutex_unlock(&locks[i%NSUMS]); // Ya podemos desbloquear la posición del array.
    return NULL;
}

int main(int argc, char *argv[])
{
    . . . .

    // Inicializamos el array de locks.
    for(int i=0; i<NSUMS; i++) { sums[i] = 0; pthread_mutex_init(&locks[i], NULL); }

    . . . .
}
```

06-Comm-Pipe:

```
#include <stdio.h>          /* printf, sprintf */
#include <stdlib.h>
#include <string.h>
#include <unistd.h>         /* fork, _exit */
#include <sys/wait.h>       /* wait */
#include <sys/types.h>

int main(int argc, char *argv[]){
    int fd[2]; int num; // Declaramos el Pipe y la variable donde iremos extrayendo el número de nuestro
    PIPE.
    pipe(fd); // Creamos el PIPE.

    for (int i=0; i<10; i++) {
        write(fd[1], &i, sizeof(i)); // Escribimos el ID en el Pipe.
        if(fork() == 0) {
            close(fd[1]); // En cada hijo, cerramos el canal de escritura.
            usleep(10000+rand()%10000);
            read(fd[0], &num, sizeof(num)); // Leemos el ID del Pipe.
            printf("I'm process %d\n", num);
            _exit(0);
        }
    }
    close(fd[0]); // En el padre, cerramos el canal de lectura.

    while (wait(NULL) > 0);
    return 0;
}
```

07-Comm-FileSem

En este caso, se usa la misma idea que en el 05, pero no se crea un array de locks, se crea un array de semáforos.

```
for (int i=0; i<NSUMS; i++){ // Inicializamos el array de semáforos.
    snprintf(nameBuffer, sizeof(nameBuffer), "mutex%d", i);
    mutexList[i] = sem_open(nameBuffer, O_CREAT, 0600, 1);
}
```

```
for(int i=0;i<500;i++) {
    snprintf(nameBuffer, sizeof(nameBuffer), "mutex%d", i);
    sem_t* mutex = sem_open(nameBuffer, O_CREAT, 0600, 1);
    sem_wait(mutex);
    . . .
    sem_post(mutex);
}
```

Antes de entrar en la zona con posible Race Condition, seleccionamos el semáforo de esa posición del array, y hacemos su lock.