Sistemas Operativos I 2023

Trabajo Práctico 2 bis

Objetivos

- Analizar diferencias entre **procesos y threads**
- **API de C de llamadas al sistema** para crear y finalizar procesos y threads

Referencias

[1] Tanenbaum, Bos – Modern Operating Systems - Prentice Hall; 4 edition (March 10, 2014) - ISBN-10: 013359162X
[2] Douglas Comer - Operating System Design - The Xinu Approach. CRC Press, 2015. ISBN : 9781498712439
[3] Silberschatz, Galvin, Gagne - Operating Systems Concepts - John Wiley & Sons; 10 edition (2018) – ISBN 978-1-119-32091-3

Software y Hardware

Este trabajo práctico se realiza en su totalidad en UNIX/Linux.

Ejercicio 1. *Programa para calcular primos en PC UNIX/Linux.*

a. ¿Qué significa el acrónimo gcc?. ¿Desde cuándo está el proyecto?. ¿Cuál es su objetivo?
b. ¿Qué es POSIX?
c. Ejecutar el programa a continuación en una PC UNIX o GNU/Linux.

```
/* primos entre el número 1 y MAX */

#include <stdio.h>

#define MAX 500000

int total = 0;

int encontrar_primos(int from, int to)
{
        int i, n, primo;
        printf( "\n   " );

        for ( i = from ; i <= to ; i++ ) {
                primo = 1;
                n = 2;

                while ( n <= i / 2 && primo ) {
                        if ( i % n == 0 )
                                primo = 0;

                        n++;
                }

                if ( primo )
                        printf( "%d \n", i );
        }

        total++;
        return 0;
}

int main()
{
        encontrar_primos(1, MAX);
        printf("Total : %i \n", total);

        return 0;
}
```

Ejercicio 2. *Sistema concurrente/paralelo compuesto de **varios procesos** en PC UNIX/Linux.*

a. Ejecute el programa debajo en un sistema GNU/Linux.
b. ¿Cuáles son funciones de la biblioteca de C qué son simplemente útiles de manera general y cuales son funciones de la biblioteca de C que realizan llamadas al sistema?
c. Complete el programa para encontrar los mismos primos que el programa en el Ejercicio 1. Pero esta vez la mitad de los primos los encuentra el primer proceso hijo y la otra mitad el segundo proceso hijo.

```
/* Padre crea dos hijos */

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

int total = 0;

#define MAX 500000

int  main(void)
{
     int  pid;
     int  pid2;
     int wstatus;

     pid = fork();
     if (pid == 0) {
        printf("Proceso hijo porque pid es %i\n", pid);

     } else {

        printf("Proceso padre porque pid es %i \n", pid);

        pid2 = fork();
        if (pid2 == 0) {
                printf("Proceso hijo porque pid es %i \n", pid2);
        } else
                printf("Proceso padre porque pid es %i \n", pid2);

     }

     /* esperamos a los hijos */
     waitpid(pid, &wstatus, 0);
     waitpid(pid2, &wstatus, 0);
     printf("Total : %i \n", total);

     return 0;
}
```

Ejercicio 3. *Sistema concurrente/paralelo compuesto de **varios threads en un mismo proceso** en PC UNIX/Linux.*

a. Ejecute el programa debajo en un sistema GNU/Linux.
b. Complete el programa para encontrar los mismos primos que el programa en el Ejercicio 1. Pero esta vez la mitad de los primos los encuentra el primer thread y la otra mitad el segundo thread.

```
/* Ejemplo de dos threads en Linux. Compilar con: gcc -o p p.c -lpthread */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define MAX 500000

int total = 0;

void *thread(void * i)
{
        int n = *((int *) i);

        if (n == 0) {
                printf("Este es el thread nro: %i \n", n);
        } else {
                printf("Este es el thread nro: %i \n", n);
        }
}


int main()
{
        int i0=0;
        int i1=1;
        pthread_t tid[2];

        /* Create independent threads each of which will execute function */
        pthread_create( &tid[0], NULL, thread, (void *) &i0);
        pthread_create( &tid[1], NULL, thread, (void *) &i1);

        /* Wait till threads are complete before main continues.  */
        for (int i = 0; i < 2; i++)
                pthread_join(tid[i], NULL);

        printf("Total : %i \n", total);

        return 0;
}
```

Ejercicio 4. Responda

a. ¿Porqué **total** es diferente en los 3 programas que encuentran primos?
b. Utilice el commando **time** para medir los tiempos de los 3 programas que encuentran primos (El del Ejercicio 1, 2 y 3).
c. Observe que con procesos (y también con threads), uno de los procesos (y también uno de los threads) termina bastante antes de ejecutarse (puede utilizar el comando htop para observar los procesos en ejecución) que el otro proceso (y el otro thread). Balancee empíricamente los dos procesos (y los dos threads) para que el tiempo final de ejecución mejore aún más con respecto al programa del Ejercicio 1.
d. ¿Cuál versión fue la que tomó menos tiempo?. ¿Existe gran diferencia de ganancia entre hacerlo con procesos que con threads?. ¿Para qué tipo de programas puede que la ganancia sea mejor con threads?