Universidad Nacional De Asunción

FACULTAD POLITÉCNICA

Lenguajes de Programación III – Primer Examen Parcial – 14/04/2017

Tema 1 – 40 p – Explique breve y concisamente:

1- En qué consisten los Procesos Pesados y Livianos, cuál es la diferencia y cuál es el objetivo de los mismos.
2- El objetivo de las funciones fork y exec, y en qué casos deberían ser usadas en forma conjunta.
3- Las formas de comunicación interproceso. Compare las mismas.
4- A que se refiere el valor de *niceness* de un proceso. Proporcione y explique un ejemplo de invocación para los comandos nice y renice de Linux.
5- A que se refiere una sección crítica.
6- Que entiende por mutex y por semáforos. Indique en qué casos conviene aplicar cada uno.
7- Explique la diferencia entre enlace dinámico y enlace estático. Indique en qué casos conviene aplicar cada uno.
8- Que entiende por Sockets Locales. Indique qué consideraciones deben tenerse respecto a los mismos.

Tema 2 – 20 p

a- (10 p.) El siguiente programa implementa un proceso que instancia un socket local (cuyo nombre es recibido como argumento) y espera por conexiones para leer mensajes de texto. Si el mensaje de texto es "quit" el programa cierra el socket y termina. Escriba un programa cliente que envíe mensajes al programa ilustrado.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Read text from the socket and print it out.  Continue until the
   socket closes.  Return nonzero if the client sent a "quit"
   message, zero otherwise.  */

int server (int client_socket)
{
  while (1) {
    int length;
    char* text;

    /* First, read the length of the text message from the socket.  If
       read returns zero, the client closed the connection.  */
    if (read (client_socket, &length, sizeof (length)) == 0)
      return 0;
    /* Allocate a buffer to hold the text.  */
    text = (char*) malloc (length);
    /* Read the text itself, and print it.  */

    read (client_socket, text, length);
    printf ("%s\n", text);
    /* Free the buffer.  */
    free (text);
    /* If the client sent the message "quit," we're all done.  */
    if (!strcmp (text, "quit"))
      return 1;
  }
}

int main (int argc, char* const argv[])
{
const char* const socket_name = argv[1];
  int socket_fd;
  struct sockaddr_un name;
  int client_sent_quit_message;

  /* Create the socket.  */
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  /* Indicate that this is a server.  */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  bind (socket_fd, &name, SUN_LEN (&name));
  /* Listen for connections.  */
  listen (socket_fd, 5);

  /* Repeatedly accept connections, spinning off one server() to deal
     with each client.  Continue until a client sends a "quit" message.  */
  do {
    struct sockaddr_un client_name;
    socklen_t client_name_len;
    int client_socket_fd;

    /* Accept a connection.  */
    client_socket_fd = accept (socket_fd, &client_name, &client_name_len);
    /* Handle the connection.  */
    client_sent_quit_message = server (client_socket_fd);
    /* Close our end of the connection.  */
    close (client_socket_fd);
  }
  while (!client_sent_quit_message);

  /* Remove the socket file.  */
  close (socket_fd);
  unlink (socket_name);

  return 0;
}
```

b- (10 p.) Reimplemente ambos programas con FIFOs.

<u>Tema 3 – 20 p –</u> La cena de los filósofos:

Un programa debe realizar una simulación de la cena de 5 filósofos, los cuales concurren todos juntos a cenar a un restaurante, sentándose todos ellos en la misma mesa que posee 5 platos de spaghetti y 5 tenedores, uno de cada lado del plato. Cada filosofo puede comer cuando obtiene los 2 tenedores correspondientes a su plato. Cada filosofo intenta primero obtener el tenedor de su izquierda y luego obtener el tenedor de su derecha. Si consigue ambos tenedores puede servirse 1 vez y comer. Cuando no puede conseguir alguno de los tenedores, libera aquel que ya posea y aprovecha el caso para reflexionar un momento y luego volver a intentar servirse de nuevo. Todos los filósofos ingieren spaghetti hasta que la cantidad total de porciones se agota.

Complete o replantee el programa propuesto de manera a que pueda conseguirse la siguiente salida.

```
Initial number of meals = 20.
All philosophers are sitting at the table.

philosopher 3: I am going to eat!
philosopher 2: I am going to eat!
philosopher 2: left=1
Philosopher 2: I got the left one!
philosopher 1: I am going to eat!
philosopher 1: left=1
Philosopher 1: I got the left one!
philosopher 3: left=1
Philosopher 3: I got the left one!
philosopher 3: right=1
philosopher 2: right=0
philosopher 1: right=0
Philosopher 1: I cannot get the right one!

Philosopher 3: I got two chopsticks!
philosopher 3: I am eating!

philosopher 0: I am going to eat!
philosopher 0: left=1
Philosopher 0: I got the left one!
philosopher 0: right=1

Philosopher 0: I got two chopsticks!
philosopher 0: I am eating!
Philosopher 2: I cannot get the right one!

philosopher 4: I am going to eat!
philosopher 4: left=0
Philosopher 4: I cannot even get the left
chopstick!

..........

Philosopher 4 has finished the dinner and is
leaving!
Philosopher 3 has finished the dinner and is
leaving!
Philosopher 1 has finished the dinner and is
leaving!
Philosopher 2 has finished the dinner and is
leaving!
Philosopher 0 has finished the dinner and is
leaving!

Philosopher 0 ate 5 meals.
Philosopher 1 ate 2 meals.
Philosopher 2 ate 6 meals.
Philosopher 3 ate 3 meals.
Philosopher 4 ate 4 meals.

main(): The philosophers have left. I am going to
exit!
```

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

#define thread_num 5
#define max_meals 20

/* initial status of chopstick */
/* 1 - the chopstick is available */
/* 0 - the chopstick is not available */
int chopstick[thread_num]={1,1,1,1,1};
/* mutex locks for each chopstick */
pthread_mutex_t m[thread_num];
/* philosopher ID */
int p[thread_num]={0,1,2,3,4};
/* number of meals consumed by each philosopher
*/
int numMeals[thread_num]={0,0,0,0,0};
/* counter for the number of meals eaten */
int mealCount = 0;
/* prototype */
void *philosopher(void *);

int main() {
    pthread_t tid[thread_num];
    void *status;
    int i,j;

    srand( (long)time(NULL) );

    /* create 5 threads representing
       5 dinning philosopher */
    for (i = 0; i < thread_num; i++) {
      if(pthread_create( tid + i, 0,
      philosopher, p + i ) != 0) {
        perror("pthread_create() failure.");
        exit(1);
       }
    }

    /* wait for the join of 5 threads */
    for (i = 0; i < thread_num; i++) {
        if(!pthread_join(tid[i], &status )==0) {
            perror("thr_join() failure.");
            exit(1);
        }
    }

    printf("Initial number of meals = %d.\n",
       max_meals);

    printf("All philosophers are sitting at the
          table.\n\n");

    for(i=0; i<thread_num; i++)
        printf("Philosopher %d ate %d
          meals.\n", i, numMeals[i]);

    printf("\nmain(): The philosophers have
       left. I am going to exit!\n\n");

    return (0);
}

void *philosopher(void  *arg) {.....}
```