Universidad Nacional De Asunción

FACULTAD POLITÉCNICA

Lenguajes de Programación III – Primer Examen Parcial – 14/04/2019

Tema 1 – 20 p – Explique breve y concisamente:

1- En qué consisten los Procesos Pesados y Livianos, así como cuál es la diferencia y el objetivo de los mismos.
2- El objetivo de las funciones fork y exec, y en qué casos cree que deberían ser usadas en conjunto y en qué casos por separado.
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 – Servidor Multi Proceso Pesado

El siguiente programa implementa un proceso servidor que instancia un socket local (cuyo nombre es recibido como argumento) y espera por conexiones para leer mensajes de texto y registrarlos por salida estándar. Si el mensaje de texto es "quit" el programa cierra el socket y termina.

Modifique el programa anterior, transformándolo en uno similar que sea del tipo de "multi proceso pesado", a fin de aprovechar eficientemente un nuevo sistema hw multiprocesador disponible y además, dotar al desarrollo de una mayor robustez respecto a una implementación multi hilo. En la nueva implementación, ya no es necesario que el programa principal termine al recibir el mensaje "quit".

```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;
}
```

Tema 3 – 20 p – Simulador Multi Hilo

Desarrolle en C/C++ un programa simulador multihilo basado en la librería **pthread** que se denomine **ticket_seller** que implemente un simulador de venta de tickets por varios vendedores.
El mismo debe recibir como parámetros:

1. El número de hilos vendedores a lanzar,
2. Un valor de tiempo de máximo en segundos entre ventas (las ventas deben realizarse por cada hilo en forma consecutiva, esperando aleatoriamente un instante entre ventas de entre 1 segundo y el tiempo máximo).
3. El número de tickets disponibles para la venta.

La salida del programa debe ser como sigue:

```
$>ticket_seller 4 5 35
Seller #1 sold one (34 left)
Seller #0 sold one (33 left)
...
Seller #1 sold one (2 left)
Seller #3 sold one (1 left)
Seller #2 sold one (0 left)
Seller #3 noticed all tickets sold! (I sold 5 myself)
Seller #2 noticed all tickets sold! (I sold 7 myself)
Seller #1 noticed all tickets sold! (I sold 15 myself)
Seller #0 noticed all tickets sold! (I sold 8 myself)
All done!
```

Tema 4 – 10 p extras – Trabajo Práctico para el 1er Parcial.