Universidad Nacional De Asunción

FACULTAD POLITÉCNICA

Lenguajes de Programación III – Primer Examen Parcial – 07/04/2017

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 – 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.

```
#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;
}
```

<u>Tema 3 – 20 p –</u> El problema de baño unisex:

En un local muy concurrido, un único baño es utilizado en forma unisex. Durante el uso del baño en todo momento se cumplen estas condiciones:

1- *Hombres y mujeres no pueden encontrarse dentro del baño al mismo tiempo.*
2- *Más de 3 personas no pueden encontrarse en el baño al mismo tiempo.*

Escriba un programa en C que resuelva el problema del baño unisex utilizando semáforos, mutexes y condiciones de variables proveídos por la librería pthreads. Considere las ilustraciones a continuación como referencia:

```
semaphore empty = 1;            /* controls access to the bathroom */
semaphore male_mutex = 1;       /* mutex for male_counter */
semaphore male_multiplex = 3;   /* limits # of men in the bathroom */
int male_counter;               /* # of men in bathroom or waiting */
semaphore female_mutex = 1;     /* mutex for female_counter */
semaphore female_multiplex = 3; /* limits # of women in the bathroom */
int female_counter;             /* # of women in bathroom or waiting */

male()
{
    down(&male_mutex);
    male_counter++;
    if (male_counter == 1) {
        down(&empty);           /* make this a male bathroom or wait */
    }
    up(&male_mutex);

    down(&male_multiplex);      /* limit # of people in the bathroom */
    use_bathroom();
    up(&male_multiplex);        /* let the next one in */

    down(&male_mutex)
    male_counter--;
    if (male_counter == 0) {
        up(&empty);             /* may become a female bathroom now */
    }
    up(&male_mutex);
}
```

```c
#include <pthread.h>

int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;

void initialize_flag ()
{
  /* Initialize the mutex and condition variable.  */
  pthread_mutex_init (&thread_flag_mutex, NULL);
  pthread_cond_init (&thread_flag_cv, NULL);
  /* Initialize the flag value.  */
  thread_flag = 0;
}

/* Calls do_work repeatedly while the thread flag is set; blocks if
   the flag is clear.  */

void* thread_function (void* thread_arg)
{
  /* Loop infinitely.  */
  while (1) {
    /* Lock the mutex before accessing the flag value.  */
    pthread_mutex_lock (&thread_flag_mutex);
    while (!thread_flag)
      /* The flag is clear.  Wait for a signal on the condition
         variable, indicating that the flag value has changed.  When the
         signal arrives and this thread unblocks, loop and check the
         flag again.  */
      pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
    /* When we've gotten here, we know the flag must be set.  Unlock
       the mutex.  */
    pthread_mutex_unlock (&thread_flag_mutex);
    /* Do some work.  */
    do_work ();
  }
  return NULL;
}

/* Sets the value of the thread flag to FLAG_VALUE.  */

void set_thread_flag (int flag_value)
{
  /* Lock the mutex before accessing the flag value.  */
  pthread_mutex_lock (&thread_flag_mutex);
  /* Set the flag value, and then signal in case thread_function is
     blocked, waiting for the flag to become set.  However,
     thread_function can't actually check the flag until the mutex is
     unlocked.  */
  thread_flag = flag_value;
  pthread_cond_signal (&thread_flag_cv);
  /* Unlock the mutex.  */
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

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

/* Prints x's to stderr.  The parameter is unused.  Does not return.  */

void* print_xs (void* unused)
{
  while (1)
    fputc ('x', stderr);
  return NULL;
}

/* The main program.  */

int main ()
{
  pthread_t thread_id;
  /* Create a new thread.  The new thread will run the print_xs
     function.  */
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  /* Print o's continuously to stderr.  */
  while (1)
    fputc ('o', stderr);
  return 0;
}
```