Documento ayuda DMUTEX

Índice

1. Compilar el código de apoyo

Ejecutar make en el directorio "raíz" de la práctica (el que tiene los directorios Controlador y Proceso), y te creará los ejecutables "proceso" (que es el código que hay que desarrollar) y "controlador" (un programa que sirve de ayuda para poner en ejecución varias instancias del programa “proceso”). Después de compilar exitosamente el contenido del directorio debe ser el siguiente:

```
$ ls -l
total 108
-rw-r--r-- 1 ptoharia ptoharia    96 feb 10 20:08 Changelog.md
-rwxrwxr-x 1 ptoharia ptoharia 39184 jun  1 08:44 controlador
drwxrwxr-x 2 ptoharia ptoharia  4096 jun  1 08:44 Controlador
drwxrwxr-x 2 ptoharia ptoharia  4096 may 23 14:12 Ejemplos
-rw-r--r-- 1 ptoharia ptoharia   904 feb 10 20:09 Makefile
-rw-r--r-- 1 ptoharia ptoharia   334 feb 10 20:08 Makefile.common
-rw-r--r-- 1 ptoharia ptoharia   172 feb 10 20:08 MakeVars
-rwxrwxr-x 1 ptoharia ptoharia 40784 jun  1 08:48 proceso
drwxrwxr-x 2 ptoharia ptoharia  4096 jun  1 08:48 Proceso
```

2. Ejecutar un único proceso.

```
$ ./proceso A
A: 1111
```

3. Abrir un puerto UDP

El número de puerto que se ha imprimido (1111) no es correcto, así que el siguiente paso es crear un socket y pedir al sistema operativo que nos asigne un puerto libre (para no "chocar" con otros alumnos si ejecutáis en triqui).Hay que editar el código Proceso/main.c añadiendo:

- El código necesario para crear un socket UDP
- Hacer un bind (indicando como puerto el 0 para el SO nos asigne un puerto disponible)
- Consultar el puerto asignado con "getsockname"
- Guardar el puerto en formato host (usando ntohs)
- Imprimir el puerto

4. Abrir un puerto UDP

Una vez hecho el paso anterior la ejecución de un proceso debería imprimir un puerto válido:

```
$ ./proceso A
A: 38770
```

Se puede comprobar que el puerto imprimido es el correcto usando netstat:

```
$ netstat -ulp | grep proceso

 no se mostrarán, necesita ser superusuario para verlos todos.)
udp        0      0 0.0.0.0:38770           0.0.0.0:*
492655/./proceso
```

Donde, resaltado en verde se puede ver que coincide el número de puerto

5. Leer lista de procesos

Se debe leer la lista de procesos con el formato de NOMBRE: PUERTO, con una línea por cada proceso. La lista terminará cuando llegue la línea “START”. Será necesario crear y rellenar una estructura de datos para almacenar esta lista de procesos y sus puertos.

Una ejecución de ejemplo de un par de proceso ejecutados en dos terminales distintas podría ser la siguiente (en verde se ha marcado la entrada estandar y en rojo la salida estándar):

TERMINAL 1:

```
./proceso A
A: 34567      <--- Abre el puerto y lo imprime
A: 34567      <---- Empieza a leer la lista de procesos
B: 23456
START
```

TERMINAL 2:

```
./proceso B
B: 23456
A: 34567
B: 23456
START
```

Podemos probar que está funcionando bien imprimiendo la estructura de datos mediante la salida de error (fprintf(stderr, “... “,). Este código para imprimir la lista dependerá de la estructura de datos que se haya elegido.

6. Cómo debería ser la ejecución de 2 procesos sin usar “controlador”

Abrir 2 terminales y hacer estos pasos:

1º Ejecutar en una terminal

```
./proceso P
P: 14124  <-- Este proceso imprime el puerto que usará
```

2º Ejecutar en otra terminal

```
./proceso Q
Q: 12345 <-- Este otro proceso hace lo mismo
```

3º Pasarle la lista de procesos al proceso P ahora que sabemos qué puertos usa cada proceso

```
./proceso P    <--- Esto estaba ya de antes
P: 14124  <--- Esto también estaba ya de antes
P: 14124  <--- Le voy copiando la lista de procesos, primero P
Q: 12345  <--- Le voy copiando la lista de procesos, segundo Q
START  <-- Le indico que ha terminado la lista
```

4º Pasarle la misma lista exacta de procesos a Q

```
./proceso Q
Q: 12345 <-- Este otro proceso hace lo mismo
P: 14124  <--- Le voy copiando la lista de procesos, primero P
Q: 12345  <--- Le voy copiando la lista de procesos, segundo Q
START  <-- Le indico que ha terminado la lista
```

7. Ejecutar usando “controlador”

Poner en ejecución los procesos "a mano" para cada prueba puede volverse tedioso enseguida, ya que hay que copiar los nombres y puerdos escritos por cada “proceso” en cada terminal y pasarles a todos la lista de procesos.

Por eso os damos el programa "controlador" que se encarga de poner en marcha los procesos y pasarles la lista de forma “automática”. Además, agrupa las salidas de los procesos y presenta una traza más "compacta".

El controlador usa un fichero de "configuración" donde se representan las acciones que se van a ir pasando a cada uno de los procesos. Algunos ejemplos de esos ficheros es lo que tenéis en el directorio Ejemplos. Por ejemplo, para ejecutar el primer ejemplo habría que hacer:

```
./controlador Ejemplos/01.ord
```

Además os proporcionamos los resultados que tienen que salir en los ficheros .result, para que podáis comparar lo que devolvéis con respecto a lo que deberíais.

El corrector de triqui ejecuta otras pruebas que podéis ver en la traza que devuelve el corrector e incluso copiar y probar la prueba que haya fallado para analizar dónde está el problema

IMPORTANTE: el controlador usa la salida estándar de los procesos, por lo que si immpirmis mensajes de “debug” por la salida estándar el controlador intentará interpretarlos

como traza y dará errores. Para depurar con mensajes debéis imprimir los mensajes por la salida de error. De esta forma el controlador no los tendrá en cuenta ni tampoco el corrector de triqui.

8. Leer acciones y procesarlas

Leer una línea e identificar qué acción hay que tratar. En el enunciado de la práctica están especificadas las acciones y la traza que deben generar.

9. Definir y componer mensajes

Para crear los mensajes hay básicamente 2 opciones:

1. Enviar estructuras donde se deje tamaño suficiente para el máximo número de procesos y nombres de sección crítica. (nunca se deben poner punteros dentro de la estructura porque enviaríamos el puntero, no el contenido, y el puntero no le serviría al proceso receptor). Podría ser algo así en pseudo-código:

```
struct MiMensaje
{
    int tipoMensaje;
    char nombreSeccionCritica[TAM_MAXIMO];
    int LC[NUM_MAX_PROCESOS];
}

struct MiMensaje mensaje;
mensaje.tipo = LOCK;
strcpy(mensaje.nombreSeccionCritica, variableconelnombre);
memcpy(&mensaje.LC, LC, sizeof(int)*num_procesos);
sentd(sd, &mensaje, sizeof(MiMensaje), .... );
```

2. Reservar memoria dinámica para el mensaje ir copiando a ese espacio reservado los distintos campos. Podría ser algo similar al siguiente pseudo-código:

```
// Reservar espacio para el contenido de un mensaje, por ejemplo:
// un entero para el tipo de mensaje,
// el nombre de la sección (más el carácter nulo de fin de string),
// y tantos enteros como procesos para el vector de relojes
int tam = sizeof(int) +
          sizeof(char) * (strlen(nombreSeccionCritica)+1) +
          num_procesos * sizeof(int);

void *mensaje = malloc(tam);

// Copiar los campos con memcpy para vectores y strcpy para los strings.
((int *) mensaje)[0] = MSG;
strcpy(mensaje + sizeof(int), nombreSeccionCritica);
memcpy(mensaje + sizeof(int) + sizeof(char)*(strlen(argv[1]+1), LC,
sizeof(int)*num_procesos);
sendto(socket, mensaje, tam, .... );
```