RT Linux - Practica 4

Autores	Schattmann Tomas Emanuel, Tedesco Aldana, Mengoni Emiliano
Materia	Sistemas de tiempo real

Introducción - RT Linux

Para esta practica se nos ha solicitado trabajar con RTLinux el cual es un RTOS (Real Time Operating Systems) que permite la gestion de tareas de tiempo real y manejadores de interrupciones. Para lograr este cometido ejecuta linux como un hilo de menos prioridad que las tareas de tiempo real permitiendo asi que estas y los

manejadores de interrupciones no se vean retrasados por operaciones del sistema operativo.

Para realizar los ejercicios implementamos por medio del lenguaje C, tareas de tiempo real. Basandonos en el procesamiento mediante hilos, los cuales gestionamos con la libreria “*pthread.h*” (POSIX Threads).

Esta libreria es en realidad un modelo de ejecucion y de ejecucion paralela que existe independientemente del lenguaje de programacion que se este usando. Permite que un programa controle multiples tipos de flujos de trabajo que se superponen en el tiempo. Cada flujo de trabajo es llamado thread. La creacion y el control de los mismos se logra a travez de “*POSIX Threads API*”.

POSIX Threads API es una API definida por el estandar POSIX.1c declarado por IEEE (Institute of Electrical and Electronics Engineers) .

Pthreads define, en C, un conjunto de tipos, funciones y constantes que permite la creacion y gestion de hilos. Incorporado tambien por herramientas de mutexes, variables condicionales, sincronizacion entre threads usando read/right barreras y semaforos binarios.

Ejercicio 1

Para el ejercicio 1 implementamos la siguiente solución:

Función “main”

La función comienza declarando una variable entera `i` que se utilizará como contador en un bucle. Luego, se crea una variable del tipo `pthread_attr_t` llamada `attr` para almacenar los atributos del hilo. Se inicializan estos atributos con la función `pthread_attr_init`.

A continuación, se declara un arreglo de `pthread_t` llamado `threads` con un tamaño de `T` (número de hilos) para almacenar los identificadores de los hilos creados. Se inicia un bucle `for` que itera desde 0 hasta `T`, donde en cada iteración se crea un nuevo hilo utilizando la función `pthread_create`. Cada hilo se asociará con la función `tarea`.

Después de crear todos los hilos, la función `main` entra en otro bucle `for` que espera a que cada hilo termine su ejecución mediante la función `pthread_join`. Esta espera

asegura que el programa principal no finalice hasta que todos los hilos hayan completado sus tareas

```
int main(int argc, char *argv[])
{
    int i;

    // pthread
    pthread_attr_t attr;
    pthread_t threads[T];

    // inicializar atributos
    pthread_attr_init(&attr);

    for (i = 0; i < T; i++){
        pthread_create(&threads[i], &attr, tarea, NULL);
    }

    // join
    for (i=0; i<T; i++){
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

Función tarea

La función `tarea` es la rutina que ejecuta cada hilo creado en el programa. Su propósito principal es medir la latencia del sistema al realizar una operación periódica. En cada iteración de un bucle, el hilo registra el tiempo actual utilizando la función `dwalltime` antes de la suspensión, luego se suspende durante 10 milisegundos utilizando la función `usleep`, y finalmente, registra nuevamente el tiempo transcurrido después de la suspensión. La latencia se calcula restando el tiempo de inicio al tiempo de finalización y ajustando por el tiempo que debería haber transcurrido (10 milisegundos). Este cálculo se acumula a lo largo de múltiples iteraciones (guardadas en la constante ITERACIONES).

Una vez terminado la ejecución de tarea se hace un promedio de la latencia acumulada y se muestra en pantalla.

```
void* tarea(void* params){
    int i;
    double act, ant, acum = 0.0;

    // se ejecutan periócamente en 1000 oportunidades
    for(i=0; i<ITERACIONES; i++){
        ant = dwalltime();
        usleep(10000);        // dormir por 10000 us = 10 ms
        act = dwalltime();

        // Medir lo que tarda en retomar menos la centesima
        // de segundo que debía transcurrir (0.01 s = 10 ms)
  acum += (act - ant - 0.01);
    }

    // imprimir resultados
    double prom = acum / ITERACIONES;
    printf("Latencia prom: %f s = %f us\n", prom, prom * 1000000);
}
```

Función dwalltime

La función `dwalltime` se encarga de medir el tiempo transcurrido con alta precisión, hasta microsegundos. Su implementación utiliza la librería `sys/time.h` y la estructura `struct timeval` para obtener la hora actual del sistema, tanto en segundos como en microsegundos. El tiempo obtenido se combina y retorna como un valor de tipo `double` que representa el tiempo transcurrido desde un momento de referencia (normalmente, el inicio de la ejecución del programa).

```
double dwalltime()
{
  double sec;
  struct timeval tv;

  gettimeofday(&tv,NULL);
  sec = tv.tv_sec + tv.tv_usec/1000000.0;
    return sec;
}
```

Ejercicio 2

Para la resolución del ejercicio dos desarrollamos dos programas:

Programa 1

Este código en C implementa un programa que lee datos desde un archivo y los envía a otro programa a través de un pipe. Cada línea del archivo contiene dos valores separados por tabulaciones: el tiempo en nanosegundos que se debe esperar antes de enviar el siguiente dato, y la temperatura en grados Celsius que se debe enviar. El programa utiliza pipes para comunicarse con otro programa y envía la temperatura a través de la tubería después de esperar el tiempo especificado.

Definición de constante

Se define la ubicación donde crearemos la pipe

```
#define MYFIFO "/tmp/myfifo"
```

Función main

El bloque principal de la función `main` inicia abriendo el archivo "fichero.in" en modo lectura (`"r"`). Luego, se crea un archivo FIFO (pipe) en la ubicación especificada en la constante “MYFIFO” con permisos de lectura y escritura para todos los usuarios (0666). A continuación, se entra en un bucle que se ejecutará mientras no se alcance el final del archivo. Dentro de este bucle, se lee cada línea del archivo, que contiene dos valores separados por tabulaciones: el tiempo en nanosegundos que se debe esperar

antes de enviar el siguiente dato y la temperatura en grados Celsius que se debe enviar.

Después de la lectura, el programa utiliza la función `nanosleep` para esperar el tiempo especificado antes de continuar. Posteriormente, se abre el archivo FIFO en modo escritura (`O_WRONLY`) y se escribe la temperatura a través del pipe. Finalmente, se cierra el descriptor de archivo asociado al pipe.

Este proceso de lectura, espera y envío se repite hasta que se alcanza el final del archivo. Una vez que se completa la lectura del archivo, se cierra el archivo "fichero.in". Sin embargo, hay un bucle infinito al final del programa (`while (1)`) que mantiene la ejecución del programa de manera indefinida.

```
int main(int argc, char *argv[])
{
  // variables del programa
  int tiempo, temp, status, fd;
  char* msg;
  FILE *arch;

  // abrir archivo
  arch = fopen ( "fichero.in", "r" );

  if (arch == NULL) {
    fputs("Error al abrir el archivo", stderr);
    exit(1);
  }

  // generar pipe
  status = mkfifo(MYFIFO, 0666);
  // if (status != 0) printf("ERROR: mkfifo returned %d\n", status);


  // repetir mientras no se acabe el archivo...
  while (!feof(arch)){
    // leer linea: tiempo \t temperatura
    // leer tiempo en ns que debo esperar para entregar un dato
    // leer la temperatura (°C) que debo entregar
    fscanf(arch, "%d\t%d\n", &tiempo, &temp);

    // esperar que se cumpla el plazo
    nanosleep((const struct timespec[]){{0, tiempo}}, NULL);

    // abrir pipe para escritura
    // printf("Abriendo pipe, esperando al otro programa...\n");
    fd = open(MYFIFO, O_WRONLY);
```

```
    // escribir temperatura por pipe
    // printf("Temp: %d, cerrando pipe...\n", temp);
    write(fd, &temp, sizeof(int));
    close(fd);
  }

  // cerrar archivo
  fclose(arch);

  while(1);

  return 0;
}
```

Programa 2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// Para manejo de hilos
#include <pthread.h>
#include <sched.h>    // para cambiar prioridad
#define T     2
#define PRIORIDAD_ALTA  10
#define PRIORIDAD_BAJA  5

// Semáforo
#include <semaphore.h>

//nanosleep
#include <time.h>

//  pipes
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#define MYFIFO "/tmp/myfifo"


// Prototipos de tareas
void* tarea1(void*);
void* tarea2(void*);

// Variables globales
sem_t semaforo;
```

```
int temp;

int main(int argc, char *argv[])
{
  // variables del programa
  pthread_attr_t attr;
  pthread_t threads[T];
  struct sched_param param;
  int i;

  // inicializar atributos de hilos
  pthread_attr_init(&attr);

  // inicializar semaforo en cero
  sem_init(&semaforo, 0, 0);

  // crear hilo de mayor prioridad
  pthread_attr_getschedparam(&attr, &param);
  param.sched_priority = PRIORIDAD_ALTA;
  pthread_attr_setschedparam(&attr, &param);
  pthread_create(&threads[0], &attr, tarea1, NULL);

  // crear hilo de menor prioridad
  pthread_attr_getschedparam(&attr, &param);
  param.sched_priority = PRIORIDAD_BAJA;
  pthread_attr_setschedparam(&attr, &param);
  pthread_create(&threads[1], &attr, tarea2, NULL);

  // espero a que finalicen
  for (i=0; i<T; i++) pthread_join(threads[i], NULL);

  return 0;
}

void* tarea1(void* args){
  int fd, valor, nBytesLeidos;

  while(1){
    // abrir pipe para escritura
    // printf("Abriendo pipe para lectura...\n");
    fd = open(MYFIFO, O_RDONLY);

    // leer pipe
    nBytesLeidos = read(fd, &valor, sizeof(int));
    // printf("%d bytes leidos, valor %d\n", nBytesLeidos, valor);

    // cerrar pipe
    close(fd);

    // si se leyeron datos...
    if (nBytesLeidos > 0){
```

```
  // controlar que no exceda 90°C
      if (valor > 90)
          printf("Atencion: la temperatura excede los 90°C \n");
      else {
          temp = valor;
          sem_post(&semaforo);
  }
    }
  }
}


void* tarea2(void* args){
  int actual, valores[3];
  int i, recibidos = 0, suma = 0;

  while(1){
    // esperar por nuevo dato
    sem_wait(&semaforo);
    actual = temp;

    // leerlo y almacenarlo
    if (recibidos < 3) recibidos++;

    valores[2] = valores[1];
    valores[1] = valores[0];
    valores[0] = actual;

    // imprimir promedio
    for (i=0; i<recibidos; i++) suma += valores[i];
    printf("Promedio: %f\n", (double) suma / recibidos);

    // reiniciar suma
    suma = 0;
  }
}
```

Implementa un programa con dos hilos de ejecución que realizan tareas específicas y se comunican a través de un semáforo y un pipe.

La función `main` inicializa atributos y variables, crea dos hilos con diferentes prioridades (`tarea1` con prioridad alta y `tarea2` con prioridad baja), y espera a que ambos hilos finalicen.

La función `tarea1` es la ejecutada por el hilo de mayor prioridad. Periódicamente lee un valor desde una tubería (pipe) y verifica si la temperatura excede los 90°C. Si no excede, actualiza la variable `temp` y señaliza al semáforo `semaforo`.

La función `tarea2` es la ejecutada por el hilo de menor prioridad. Espera a que `tarea1` actualice la variable `temp`, luego almacena los últimos tres valores de temperatura, calcula el promedio y lo imprime. Este proceso se repite indefinidamente en un bucle infinito.

En resumen, el programa utiliza dos hilos con diferentes prioridades para monitorear la temperatura y calcular el promedio de los últimos tres valores. La comunicación entre los hilos se realiza mediante un semáforo y un pipe.