GUÍA POSIX RELOJES

Se incluyen las dependencias:

```
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
#include <string.h>
#include <stdio.h>

#include <unistd.h>
#include <time.h>
#include <errno.h>
#include <sched.h>
#include <pthread.h>
#include <sys/mman.h>
```

Se definen los periodos de las tareas. Todos los periodos en ms se pasan a nsec.

Se definen las prioridades. Las tareas con menor plazo son las más prioritarias.

Se define el incremento o decremento.

```
#define PERIODO_TMP_SEC 1
#define PERIODO_TMP_NSEC 0
#define PRIO_TMP 28

#define PERIODO_CLI_SEC 4
#define PERIODO_CLI_NSEC 0
#define PRIO_CLI 26

#define PERIODO_MTR_SEC 4
#define PERIODO_MTR_NSEC 0
#define PRIO_MTR 24
```

Se crea un struct por cada mutex, con la variable compartida (contador, valor) y el mutex.

```
struct Data{
    pthread_mutex_t mutex;
    float valtemp;
    int aire;
    float gasto;
};
```

OPCIONAL: FASE DE SOPORTE

OPCIONAL: Se definen los CHKN y CHKE, para controlar los errores al llamar al sistema.

```
// #define CHKN(syscall) \
//     do { \
//         int err = syscall; \
//         if (err != 0) { \
//             fprintf(stderr, "%s: %d: SysCall Error: %s\n", \
//                     __FILE__, __LINE__, strerror(errno)); \
//             exit(EXIT_FAILURE); \
//         } \
//     } while (0)

// #define CHKE(syscall) \
//     do { \
//         int err = syscall; \
//         if (err != 0) { \
//             fprintf(stderr, "%s: %d: SysCall Error: %s\n", \
//                     __FILE__, __LINE__, strerror(err)); \
//             exit(EXIT_FAILURE); \
//         } \
//     } while (0)
```

OPCIONAL: Se define la función getTime, para obtener el tiempo actual y pasarlo a string. Esto se usa para mensajes en depuración.

```
// const char *get_time (char *buf){
//     time_t t = time(0); //Tiempo actual
//     char *f = ctime_r(&t, buf); //Convertir el tiempo en una cadena y
almacenarla en buf
//     f[strlen(f)-1] = '\0'; //Eliminar el salto de línea
//     return f; //Retornar la cadena
// }
```

Se define la función “espera_activa(time_t seg)”, que se usará siempre y cuando nos pidan que **esperemos activamente**.

```
void espera_activa(time_t seg){
    volatile time_t t = time(0) + seg;
    while (time(0)<t) { /*Esperar activamente*/}
}
```

Se define la función “addtime(struct timespec *tm, const struct timespec *val)”.

- Se suma a tm->tv_sec los val->tv_sec.
- Se suma a tm->tv_nsec los val->tv_nsec.
- Si los tm-> tv_nsec sobre pasan el 1000000000L de nsec (1 segundo):
    - A tm->tv_sec se le suma su división entre 1000000000L (tm->tv_nsec / 1000000000L)
    - tm->tv_nsec será igual al resultado de hacer el módulo entre 1000000000L. (tm->tv_nsec % 1000000000L)

```
- void addtime (struct timespec *tm, const struct timespec *val){
-     tm->tv_sec += val->tv_sec;
-     tm->tv_nsec += val->tv_nsec;
-     if(tm->tv_nsec > 1000000000L){
-         tm->tv_sec += (tm->tv_nsec / 1000000000L);
-         tm->tv_nsec = (tm->tv_nsec % 1000000000L);
-     }
- }
```

FASE DE LAS TAREAS

Se crean las tareas mediante métodos void:

- Se crea un periodo mediante “const struct timespec”, que es igual a {periodo_seg, periodo_nseg}
- Se crea un “struct timespec next”.
- Se asigna al puntero struct de Data el arg recibido por parámetro. “struct Data *data = arg;
- OPCIONAL: Se crea un parámetro para la política de planificación “struct sched_param”.
- OPCIONAL: Se crea un “const char *pol” y un “int policy”.
    - Posteriormente se obtiene la política de planificación mediante “pthread_getschedparam” recibiendo como parámetros (pthread_self(), &policy, &param).
    - Finalmente, se asigna a “pol” la política obtenida mediante condicionales:

```
- pol = (policy == SCHED_FIFO) ? "FF" : (policy == SCHED_RR) ? "RR" :
  "--"; //Obtener la política de planificación
```

- Se hace un “clock_gettime(CLOCK_MONOTONIC, &next)”.
- Se hace un bucle infinito “while(1)”:
    - Se hace un “clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &next, NULL)”
    - Se añade a next el tiempo del periodo. “addtime(&next, &periodo)”.
    - Se bloquea el mutex “pthread_mutex_unlock(&data->mutex)”;
        - Si la tarea tuviera iteraciones, el lock iría dentro del for, y dentro del for se haría los datos extra que cambian según el ejercicio.
            - Se haría el unlock del mutex “pthread_mutex_unlock(&data->mutex)”
            - Se haría ahora la espera activa, pasando por parámetro los segundos necesarios.
        - Si la tarea no tiene iteraciones, se hacen las operaciones extra correspondientes entre el lock y el unlock
    - Se desbloquea el mutex. “pthread_mutex_unlock(&data->mutexA);
- Retornas NULL.

```c
void *tareaTMP (void *arg){
    const struct timespec periodo = {PERIODO_TMP_SEC, PERIODO_TMP_NSEC};
    struct timespec next;
    struct Data *data = arg;
    float randTmp;
    // struct sched_param param; //Parámetros de planificación
    // const char *pol; //Política de planificación
    // int policy; //Política de planificación
    // pthread_getschedparam(pthread_self(), &policy, &param); //Obtener
la política de planificación y los parámetros
    // pol = (policy == SCHED_FIFO) ? "FF" : (policy == SCHED_RR) ? "RR"
: "--"; //Obtener la política de planificación
    // printf("# Tarea Sensor de Temperatura [%s:%d]\n", pol,
param.sched_priority); //Mensaje de depuración
    clock_gettime(CLOCK_MONOTONIC, &next);
    while(1){
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &next, NULL);
        addtime(&next, &periodo);
        pthread_mutex_lock(&data->mutex);

        randTmp = rand() % (M + 1 - N) + N;
        printf("La temperatura aleatoria es [%f]\n", randTmp);

        data->valtemp = (TMP_CONFORT-randTmp)*2.5;

        pthread_mutex_unlock(&data->mutex);
    }
    return NULL;
}
```

FASE PRINCIPAL

- OPCIONAL: Función usage para mostrar si hay error cuando introduces en consola un parámetro equivocado.

```c
- // void usage (const char *nm){
- //     fprintf(stderr, "usage: %s [-h] [-ff] [-rr] [-p1] [-p2]\n",
  nm); //Imprimir mensaje de uso
- //     exit(EXIT_FAILURE); //Salir con error
- // }
```

- OPCIONAL: Función get_args, que se complementa con usage, para asociar el parámetro recibido por consola con la política correcta.

```c
- // void get_args (int argc, const char *argv[], int *policy, int
  *prio1, int *prio2){
- //     int i; //Contador
- //     if (argc < 2){
```

```
-   //         usage(argv[0]); //Imprimir mensaje de uso
-   //     }else{
-   //         for(i = 1; i<argc; i++){
-   //             if (strcmp(argv[i], "-h") == 0){
-   //                 usage(argv[0]); //Imprimir mensaje de uso
-   //             }else if (strcmp(argv[i], "-ff") == 0){
-   //                 *policy = SCHED_FIFO; //Asignar la política de
    planificación FIFO
-   //             }else if (strcmp(argv[i], "-rr") == 0){
-   //                 *policy = SCHED_RR; //Asignar la política de
    planificación RR
-   //             }else if (strcmp(argv[i], "-p1") == 0){
-   //                 *prio1 = PRIORIDAD_A; //Asignar la prioridad de
    la tarea A
-   //                 *prio2 = PRIORIDAD_B; //Asignar la prioridad de
    la tarea B
-   //             }else if (strcmp(argv[i], "-p2") == 0){
-   //                 *prio1 = PRIORIDAD_B; //Asignar la prioridad de
    la tarea B
-   //                 *prio2 = PRIORIDAD_A; //Asignar la prioridad de
    la tarea A
-   //             }else{
-   //                 usage(argv[0]); //Imprimir mensaje de uso
-   //             }
-   //         }
-   //     }
-   // }
```

- Función “int main(int argc, const char *argv)”:
    - Se crea una variable del struct Data, que podemos llamarle “shared_data”, por ejemplo.
    - Se crea una variable “pthread_attr_t” attr.
    - Se crea una variable “struct sched_param” param.
    - Se crean t+1 variables de prioridad, siendo t el número de tareas.
        - Int prio0=1, el resto se le asigna la prioridad definida globalmente.
        - Se crean t variables de hebras. “pthread_t” t1, t2, etc.
    - “mlockall(MCL_CURRENT | MCL_FUTURE)”. Siempre igual
    - Se le asigna a prio0 la prioridad más alta +1. Ej: “prio0 = PRIO_A +1”
    - Se le asigna a param la prioridad 0, mediante “param.sched_priority”.
    - Se almacena en param la prioridad deseada. Ej: “pthread_setschedparam(pthread_self(), SCHED_FIFO, &param);
    - Se inicializa el struct
        - Se setean los valores a 0. “shared_data.valor=0”.
        - Se inicializan los mutex. “pthread_mutex_init(&shared_data.mutex, NULL)”.
    - Se inicializa el attr.
        - “pthread_attr_init(&attr)”
        - “pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED)”

        - “pthread_attr_setschedpolicy(&attr, SCHED_FIFO)”
    - Se crean las hebras
        - “param.sched_priority = prio”
        - “pthread_attr_setschedparam(&attr, &param)”
        - “pthread_create(&t, &attr, tarea, &shared_data)”
        - Este proceso se repite para todas las hebras.
    - Se destruye attr. “pthread_attr_destroy(&attr)”.
    - Se hace un join PARA CADA hebra. “pthread_join(t, NULL);
    - Se destruye el/los mutex. “pthread_mutex_destroy(&shared_data.mutex)
    - Se retorna 0.

```c
int main(int argc, const char *argv[]){
    struct Data shared_data;
    pthread_attr_t attr;
    struct sched_param param;
    int prio0=1, prio1=PRIO_TMP, prio2=PRIO_CLI, prio3=PRIO_MTR;
    pthread_t t1, t2, t3;



    mlockall(MCL_CURRENT | MCL_FUTURE);
    prio0 = PRIO_TMP + 1;
    param.sched_priority = prio0;
    pthread_setschedparam(pthread_self(), SCHED_FIFO, &param);

    srand(time(NULL));
    shared_data.gasto = 0.0;
    shared_data.valtemp = 0.0;
    pthread_mutex_init(&shared_data.mutex, NULL);

    pthread_attr_init(&attr);
    pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr, SCHED_FIFO);

    param.sched_priority = prio1;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&t1, &attr, tareaTMP, &shared_data);

    param.sched_priority = prio2;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&t2, &attr, tareaCLI, &shared_data);

    param.sched_priority = prio3;
    pthread_attr_setschedparam(&attr, &param);
    pthread_create(&t3, &attr, tareaMTR, &shared_data);

    pthread_attr_destroy(&attr);
```

```
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    pthread_mutex_destroy(&shared_data.mutex);

    return 0;
}
```