Chuleta POSIX

Software para Sistemas Empotrados y Dispositivos Móviles

El propósito de esta chuleta es ayudarte a establecer un guion a seguir cuando realices el examen de POSIX en la asignatura. Deberás adaptar el contenido de tu código a lo que requiera el enunciado.

Esta guía no sustituye a la Guía de Referencia de POSIX que está en el Campus Virtual y seguramente no podrás usarla durante el examen.

ficheros a incluir

```
#include <pthread.h>      Hebras
#include <semaphore.h>    Semáforos
#include <signal.h>       Señales
#include <stdio.h>        Entrada y salida
#include <stdlib.h>       Biblioteca estándar
#include <sys/mman.h>     Para evitar paginación
#include <time.h>         Tiempo
#include <unistd.h>       Llamadas de sistema UNIX
```

compilación y enlazado (GCC)

para compilar: `gcc -o <fichero de salida> <fichero fuente> -lpthread -lrt -Wall -Werror`
para ejecutar: `sudo taskset -c 0 ./<ejecutable>`
contraseña admin: `alumno`

declaraciones globales

1. Define macros usando #define

Ten en cuenta que estas macros las evalúa el preprocesador de C antes de compilar tu código. Una vez evaluadas, se sustituyen donde las uses como si fuese la función «Buscar y Remplazar» de tu IDE favorito.

```
#define PRIORIDAD_CONTROL 23
#define PRIORIDAD_MONITORIZACION 22
#define PERIODO_MONITORIZACION_MS 2500
#define SIGTIMER_MONITOR (SIGRTMIN)
#define SIGALERTA (SIGRTMIN + 1)
```

También puedes poner los valores «a fuego» si lo ves más rápido, pero lo suyo es crearte tus macros para tener un punto donde ajustar fácilmente los valores.

Recuerda:

- Cuando te pidan asignar prioridades, **asigna mayor prioridad a tareas con un plazo menor**. Si te piden según periodo, usa el periodo en lugar del plazo
- Es muy recomendable poner nombres a tus señales. Puedes usar SIGRTMIN, SIGRTMIN + 1, SIGRTMIN + 2, etc. o, por el contrario, SIGRTMAX, SIGRTMAX - 1, SIGRTMAX - 2 para los números de tus señales.

2. Declara structs con las variables compartidas

Define structs con las variables, mútex, etc. que compartan las diferentes tareas:

```
struct termometro {
    int temperatura;
    pthread_mutex_t mutex;
}
```

Luego en tu main() declaras:
`struct termometro mi_termometro`
y das valores iniciales, según proceda.

3. Declara cualquier función auxiliar que pudieras necesitar

Por ejemplo, para sumar cierta cantidad de milisegundos a un objeto «timespec»:

```
void addtime(struct timespec *ts, long ms) {
    ts->tv_nsec += 1000000L * ms;
    ts->tv_sec += ts->tv_nsec / 1000000000L;
    ts->tv_nsec %= 1000000000L;
}
```

4. Crea las rutinas de cada hebra

```
void *mi_rutina(void *arg) {
    struct termometro = *(struct termometro*)arg;
    // lo que sea
    return NULL;
}
```

Si no usas el argumento, es un convenio escribir después de la instrucción «return»:
`(void)arg;`

qué hacer en tu main()

1. Inicializa variables compartidas.

```
struct termometro mi_termometro
```

Da valores iniciales, según proceda.

2. Bloquea la memoria para evitar paginación.

Esto evita la sobrecarga del sistema de paginación del SO

```
mlockall(MCL_CURRENT | MCL_FUTURE);
```

3. Establece una política de planificación.

A no ser que te digan lo contrario, puedes suponer Round Robin. Da a la hebra principal la mayor prioridad de todas.

```
struct sched_param schedparam;
schedparam.sched_priority = 50;
pthread_setschedparam(pthread_self(), SCHED_RR, &schedparam);
```

4. (Si usas señales) Bloquea todas las señales que vayas a usar

Así evitamos perderlas

```
sigset_t todas_las_seniales;
sigemptyset(&todas_las_seniales);
sigaddset(&todas_las_seniales, SIG1);
sigaddset(&todas_las_seniales, SIG2);
sigaddset(&todas_las_seniales, SIG3);
sigaddset(&todas_las_seniales, SIG4);
pthread_sigmask(SIG_BLOCK, &todas_las_seniales, NULL);
```

Haz esto ANTES de crear tus hebras para que puedan heredar esta configuración.

5. (Si usas mútex) Inicializa tus mútex

Para impedir la inversión de prioridad, crea un objeto atributos para fijar los atributos de tus mútex.

```
pthread_mutexattr_t mutexattr;
pthread_mutexattr_init(&mutexattr);
```

Según te pida el ejercicio, especificas el protocolo a usar: herencia de prioridad (PTHREAD_PRIO_INHERIT) o techos de prioridad (PTHREAD_PRIO_PROTECT). Si usas techos de prioridad, calcula el techo de cada recurso y asígnalo (recuerda, la prioridad de la hebra más prioritaria que lo vaya a usar):

```
pthread_mutexattr_setprotocol(&mutexattr, PTHREAD_PRIO_PROTECT);
pthread_mutexattr_setprioceiling(&mutexattr, 30);
```

Finalmente, crea cada mutex:

```
pthread_mutex_init(&termometro.mutex, &mutexattr);
```

Consejo: puedes usar el mismo objeto atributos para varios mútex, siempre que asignes cada configuración a su mútex respectivo. A partir de este punto puedes ir modificando los atributos y creando los mútex adicionales que se necesiten.

6. Crea tus hebras

Para aplicar las políticas y prioridades a tus hebras, debes crear un objeto atributos:

```
pthread_t th_monitor, th_control;
pthread_attr_t attr;
```

Especifica explícitamente qué política usar:

```
pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_RR);
```

Y ve fijando las prioridades y creando las hebras, repitiendo las siguientes líneas para cada hebra.

```
myparam.sched_priority = PRIORIDAD_MONITORIZACION;
pthread_attr_setschedparam(&attr, &myparam);
pthread_create(&th_monitor, &attr, sensor_temp,
(void*)&mi_termometro);
```

Una vez creadas todas las hebras, haz que la hebra principal espere las que acabas de crear:

```
pthread_join(th_monitor, NULL)
pthread_join(th_control, NULL)
...
```

liberar recursos

Práctica necesaria para usar correctamente la memoria.

```
pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

pthread_mutex_destroy(pthread_mutex_t *mutex);

pthread_attr_destroy(pthread_attr_t *attr);
```

qué NO se pide en el examen

- **Funciones CHKE, CHKN** para comprobar si las llamadas al sistema devuelven algún error. (En el mundo real habría que hacerlo.)
- **Leer argumentos de línea de comandos** (funciones usage() y get_args())

qué podría QUIZÁS pedirse en el examen

- **Semáforos y variables condición** (no se vieron en la práctica pero sí en clases teóricas)

Si alguien habla con el profesor y consigue resolver la duda, que me lo haga saber para actualizar este guion

qué hacer en la rutina de cada hebra

Si te piden relojes de tiempo real (RTR)

```
void *monitorizacion(void *arg) {
    struct termometro *termometro = (struct termometro*)arg;
    struct timespec siguiente;
    clock_gettime(CLOCK_MONOTONIC, &siguiente);
    while (1) {
        // lo que sea
        addtime(&ts_siguiente, PERIODO_MONITORIZACION_MS);
        clock_nanosleep(CLOCK_MONOTONIC,
            TIMER_ABSTIME, &ts_siguiente, NULL);
    }
    return NULL;
}
```

Para recibir señales

Debes crear una máscara específica para cada hebra, además de una máscara «global» en el main() para bloquear todas las señales que uses y evitar perderlas.

```
sigset_t mis_seniales;
sigemptyset(&mis_seniales);
sigaddset(&mis_seniales, <una señal que esta hebra debe esperar>);
sigaddset(&mis_seniales, <otra señal que esta hebra debe
esperar>);
...
int info;
while (1) {
    sigwait(&mis_seniales, &info);
    // hacer lo que sea
}
return NULL;
```

Si te piden usar timers

```
struct termometro *termometro = (struct termometro*)arg;
struct itimerspec its;
// Periodo en segundos y nanosegundos
its.it_interval.tv_sec = 1;
its.it_interval.tv_nsec = 0;

// Retraso de la primera señal tras timer_settime
// Mínimo 0,000000001 para que se envíe
its.it_value.tv_sec = 0;
its.it_value.tv_nsec = 1;

struct sigevent sigev;
sigev.sigev_notify = SIGEV_SIGNAL;
sigev.sigev_signo = <señal que se enviará periódicamente>;

timer_t timer;
timer_create(CLOCK_MONOTONIC, &sigev, &timer);
timer_settime(timer, 0, &its, NULL);

// Crea tu máscara
sigset_t mis_seniales;
sigemptyset(&mis_seniales);
sigaddset(&mis_seniales, <señal que se enviará periódicamente>);

int info;
while (1)
{
    sigwait(&mis_seniales, &info);
    // hacer algo
}
// Liberar recursos (buena práctica)
timer_delete(timer);
return NULL;
```