1 – Planificación con Cheddar

Primero veamos cuáles son los parámetros que se le puede asignar a cada tarea:

[1] **Nombre de la tarea**

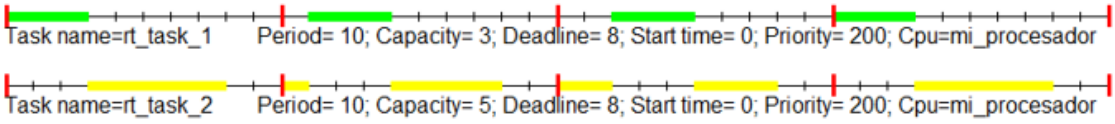
[2] **Periodo**: indica cada cuántos ticks debería repetirse la tarea, idealmente.

[3] **Capacidad**: es la duración de la tarea para cada ejecución de la misma.

- ❒ *Ej: si la capacidad es 2 y el periodo es 10, la tarea debe ejecutarse durante 2 ticks en cada rango de 10 ticks. Si es preemptive puede separarse.*

[4] **Deadline**: tick máximo donde puede finalizar la tarea, en cada periodo.

- ❒ *Ej: si el deadline es 8 y el periodo es 10, cada repetición siempre debe terminar 2 ticks antes del fin del periodo. Como mucho, la repetición podría comenzar en el tick "deadline – capacity" (3 en rt_task_2).*



[5] **Start Time**: delay inicial de la repetición de la tarea, en cada periodo.

[6] **Prioridad**: valor entre 1 y 255, donde un mayor número indica mayor prioridad. Se utiliza para determinar cuál tarea debe empezar primero o apropiar CPU.

[7] **CPU**: en el caso monocore, se asigna la misma a todas las tareas.

1.1 – Planificador

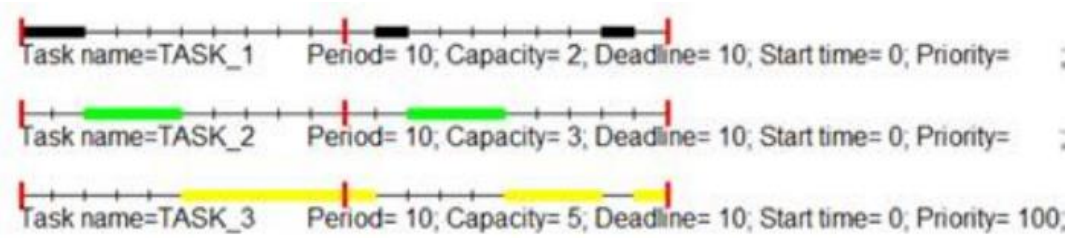
En la materia vimos el *scheduler* **POSIX 1003.1b/Highest Priority First**, que básicamente hace que las tareas de prioridad más alta se ejecuten primero. Un poco de contexto: **POSIX** es un conjunto de servicios y estándares para SO tipo Unix, para dar portabilidad a nivel de código; la versión 1003.1b define semáforos, señales, timers más precisos y scheduling con 3 políticas (SCHED_FIFO, SCHED_RR y SCHED_OTHER).

Las políticas se aplican entre las tareas de igual prioridad. En el caso SCHED_RR, cada tarea puede utilizar la CPU durante un quantum especificado, agregándose luego al final de la "cola de espera". Hay apropiación si arriba una tarea más prioritaria.

1.2 – Determinar prioridad

Resolvamos el ejercicio de parcial. Deberíamos ver si hay alguna tarea o tareas que siempre comienzan primero; en este caso, el orden es (1-2-3), (3-1-2). Deducimos:

- ❒ T1 siempre arranca antes que T2, entonces T1 "podría" tener mayor prioridad.
- ❒ T1 arranca antes que T3, pero luego no, entonces Prioridad(T1) = 100
- ❒ T2 arranca antes que T3, pero luego no, entonces Prioridad(T2) = 100



1.3 – Determinar políticas

Siguiendo con el mismo ejercicio, hay que determinar si se utiliza SCHED_FIFO o SCHED_RR en cada tarea. Por el orden, pareciera ser FIFO en general, pero se observa que las tareas 1 y 3 tienen su ejecución "particionada" en el segundo período.

Como la prioridad es igual en las 3 tareas, la apropiación de CPU se debe a una política Round Robin. En la tarea TASK_3 se observa que el máximo tiempo de uso continuo de CPU es 3 ticks. **Respuesta: la política es Round Robin con quantum 3.**

- Primer período: desde tick 0 hasta fin del tick 9. TASK_3 completa su quantum, pero sigue utilizando la CPU porque las otras ya completaron su ejecución.

1	2								
		1	2	3					
					1	2	3	1	2

- Segundo período: desde tick 10 hasta fin de tick 19. TASK_3 utiliza la CPU hasta completar quantum, luego pasa a TASK_1, que solo le restaba 1 quantum.

	3							1	
		1	2	3					
3					1	2	3		1

2 – Pthreads

Se presentan las funciones POSIX para el manejo de hilos y semáforos.

2.1 – Creación de hilos

Se utiliza la función *pthread_create()*, que crea un hilo y comienza su ejecución en el proceso que lo invoca. Si la operación fue exitosa retorna 0, sino un código de error.

```
int pthread_create(
    pthread_t* thread,
    pthread_attr_t* attr,
    void* (*tarea)(void*),
    void* arg
);
```

Requiere 4 parámetros.

- thread: referencia al ID del hilo creado.
- attr: atributos del hilo a crear o NULL.
- tarea: nombre de la función que el hilo ejecutará al crearse, **retorno tipo void***.
- arg: parámetro pasado a la función tarea, puede ser de cualquier tipo o NULL.

2.2 – Espera de hilos

Se utiliza la función *pthread_join()* para esperar hasta la finalización de un hilo especificado. Si el hilo ya finalizó, retorna el control de manera inmediata.

```
int pthread_join(
    pthread_t thread,
    void** retval
);
```

Requiere 2 parámetros:

- thread: ID del hilo a esperar que finalice.
- retval: referencia al valor que devuelve el hilo al finalizar. Se puede utilizar NULL.

2.3 – Finalización de hilos

Se puede utilizar la función *pthread_exit()*, desde un hilo, para finalizar su ejecución. Tiene un único parámetro *void* retval*, para asignar un valor de retorno, que corresponde al obtenido en la función pthread_join() a dicho hilo. Otra manera es escribir la sentencia *return retval* al final de la tarea, cumpliendo la misma función.

2.4 – Asignación de prioridades

La función *pthread_attr_init()* inicializa una variable tipo *pthread_attr_t* con los valores por defecto. Si se desea especificar una prioridad, antes de crear el hilo, se debe obtener y actualizar la estructura sched_param de los atributos. Consultar Anexo II.

2.5 – Manejo de semáforos

En POSIX, se utiliza la biblioteca **<semaphore.h>**, que define el tipo **sem_t** e incluye varias funciones, entre ellas la creación de un semáforo mediante *sem_init()*.

```
int sem_init(
    sem_t *sem,
    int pshared,
    unsigned int value
);
```

Requiere 3 parámetros:

- sem: puntero al semáforo a crear.
- pshared: compartido en fork(), dejar en 0.
- value: valor inicial del semáforo.

Para ser utilizado por varios hilos, debe ser una variable compartida. Luego para hacer un **P** se invoca **sem_wait(&sem)**, y para hacer un **V** es **sem_post(&sem)**.

3 – FreeRTOS

3.1 – Manejo de semáforos

En FreeRTOS, se dispone de una biblioteca más amplia de funciones para el manejo de semáforos: **<semphr.h>**, que además define el tipo **SemaphoreHandle_t**.

- SemaphoreHandle_t **xSemaphoreCreateBinary**(): crea semáforo binario.
- SemaphoreHandle_t **xSemaphoreCreateMutex**(): crea mutex (ídem anterior).
- SemaphoreHandle_t **xSemaphoreCreateCounting**(maxCount, initValue).
    - maxCount limita el valor interno que puede alcanzar el semáforo.
- int **xSemaphoreTake**(semHandle, ticksToWait): retorna pdTRUE si se logró hacer **P** al semáforo antes de expirar los ticks especificados, sino pdFALSE.
- int **xSemaphoreGive(**semHandle): retorna pdTRUE si se logró hacer **V** al semáforo porque no excede el máximo valor interno, sino pdFALSE.

3.2 – Creación de tareas

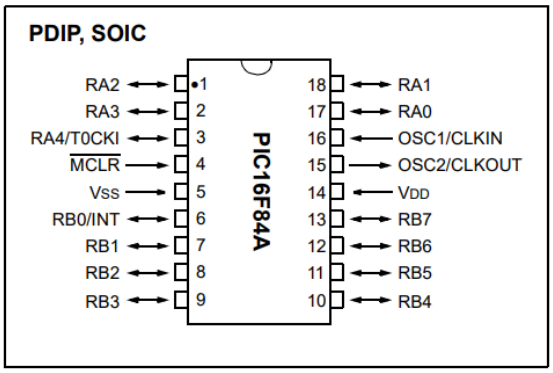
Es suficiente con utilizar la función *xTaskCreate()*, que crea e inicia una tarea.

```
xTaskCreate(
    TaskFunction_t pvTaskCode,
    char* pcName,
    configSTACK_DEPTH_TYPE usStackDepth,
    void *pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t *pxCreatedTask
);
```

Se requieren 6 parámetros: el nombre de la función a ejecutar **(tipo void)**, descripción, tamaño de stack (ej: 128), argumentos de función a ejecutar, prioridad (ej: 1) y un manejador opcional (NULL).

4 – Microcontroladores

Se usó el PIC16F84A. La cabecera del MCU es **<htc.h>**. Si se utiliza la función *__delay_ms()*, es necesario definir la macro *_XTAL_FREQ = 1.000.000*. Por otra parte, las interrupciones globales se habilitan seteando el bit GIE = 1.



4.1 – Puertos I/O

Se utilizan los registros TRISA y TRISB para indicar si los pines se utilizarán para escritura (0) o lectura (1) [*al revés que en ATmega 328P*], y los registros PORTA y PORTB para leer o escribir bits en los mismos. Resulta conveniente usar solo el puerto B, ya que tiene 8 pines completos.

A diferencia de *CDyM*, se asigna el valor directamente al bit TRISAn, TRISBn, RAn o RBn, donde "n" es un número. Por ejemplo: *TRISB2* asigna RB2 como lectura.

4.2 – Timer 0

Es el único Timer presente en este microcontrolador. Posee un contador interno de 8 bits, y produce interrupción de Overflow al exceder de 0xFFh. Existen 2 modos de funcionamiento: **timer (reloj interno, T0CS = 0)** y counter (externo, T0CS = 1). En el primer modo, se incrementa el valor del registro TMR0 cada 4 μs (ciclo de instrucción).

También posee un *prescaler*, con divisores de potencias de 2, elegibles mediante los bits PS2, PS1 y PS0. Ejemplo: *PS2:0 = 111 = 7* asigna 1/256. Luego, se debe escribir un 0 en el bit PSA para asignar el prescaler elegido al Timer (escribir 1 para WDT).

La interrupción está habilitada si el bit T0IE = 1 (además de GIE = 1), y luego en la rutina de interrupción (función *void interrupt nombreFun*) es necesario limpiar el flag T0IF para la próxima interrupción, y reasignar valor inicial de TMR si corresponde.

Ejemplo: si deseamos interrumpir cada 250 ms, usando prescaler de 256 se debe calcular el valor inicial de TMR0. Resulta $4\ \mu s * 256 * Incrementos = 250\ ms$*, donde "incrementos" es 256 – ValorIni(TMR0). Para este caso, se asigna TMR0 = 12.*

4.3 – ADC

Ejercicio 3 de la Práctica 1. Se usó el PIC 16877, que además posee un puerto C y un puerto D de 8 bits. Se trabaja con los siguientes bits/registros:

- PCFG3:0 = 1110: *utiliza AN0 (RA0) como entrada analógica y VREF = VDD.*
- ADFM: *justificar los bits hacia la derecha (0) o izquierda (1).*
- ADON: *se escribe un 1 para habilitar ADC.*
- GO: *se escribe un 1 para iniciar conversión. Valdrá 0 cuando finalice.*
- ADRESH: *contiene la parte alta del resultado de la conversión.*
- ADRESL: *contiene la parte baja del resultado de la conversión.*

En caso de optar por interrupciones, se evita el polling *while(GO)* mediante:

- GIE = 1: *habilita las interrupciones globales (incluye a Timer 0).*
- PEIE = 1: *habilita las interrupciones de periféricos.*
- ADIE = 1: *habilita la interrupción de conversión completada.*
- ADIF = 0*: limpia el flag de interrupción realizada para que suceda la siguiente.*
- GO = 1: *produce el inicio de una nueva conversión. Debe activarse cada vez.*

4.4 – Display de 8 segmentos

Se encuentran manejados por Latches, que poseen 8 entradas de datos (para recibir el número a mostrar entre 00h y FFh), y una entrada de reloj, de modo que los datos de entrada son transferidos hacia el display de salida cuando se recibe **un flanco de subida**.

5 – Dwalltime

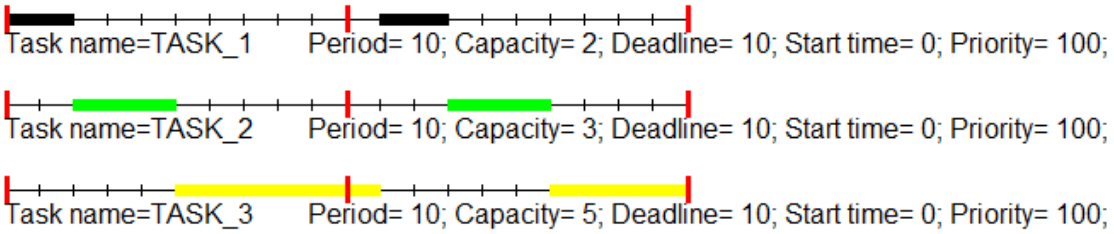
Se utiliza la biblioteca *<sys/time.h>*, que ofrece la función *gettimeofday()*. Dicha función requiere una *struct timeval* por referencia y un timezone (lo dejamos nulo). La estructura timeval posee dos campos: *tv_sec* y *tv_usec*, donde el 2° es menor a 1M.

6 – Pipes

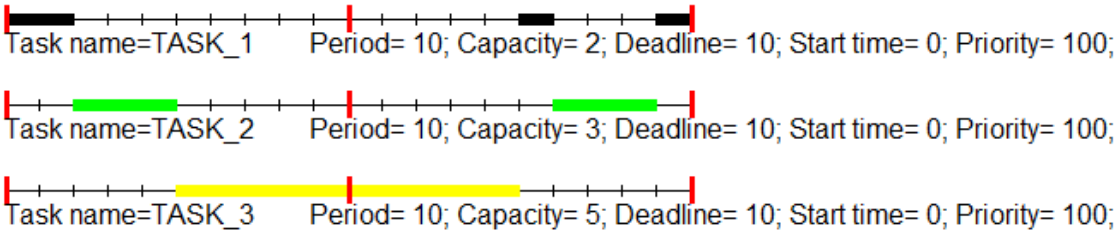
Se crean mediante *mkfifo(nom, 0666)*. Luego *fd = open(nom, tipo)*, donde el tipo puede ser WR_ONLY o RD_ONLY. Para cerrar, *close(fd)*. Las lecturas y escrituras se realizan con *read()* y *write()*, ambas reciben fd, puntero a variable y *sizeof(variable)*.

Anexo I – Resultados según políticas

Si cambiamos la política de TASK_1 a SCHED_FIFO, entonces no se aplica el uso de quantum para la misma, deja que finalice completamente su ejecución.



Cambiar la política de TASK_2 a SCHED_FIFO no produce ningún cambio, ya que la duración (capacidad) coincidía con el quantum de SCHED_RR. Por otra parte, cambiar la política de TASK_3 a SCHED_FIFO, dejando el resto como SCHED_RR, produce una larga ejecución de TASK_3, como se puede observar a continuación:



De esta forma, las respuestas posibles al ejercicio del parcial son 2:

- Respuesta 1:
  - Tarea 1 con prioridad 100 y política Round Robin.
  - Tarea 2 con prioridad 100 y política Round Robin.
  - Tarea 3 con política Round Robin.
  - Quantum 3

- Respuesta 2:
  - Tarea 1 con prioridad 100 y política Round Robin.
  - Tarea 2 con prioridad 100 y política FIFO.
  - Tarea 3 con política Round Robin.
  - Quantum 3.

Anexo II – Prioridades en Pthreads

Se presenta un programa en C que crea dos hilos de diferentes prioridades:

```c
// Biblioteca Pthreads
#include <pthread.h>

// Prototipos de función
void* tarea1(void* arg);
void* tarea2(void* arg);


int main(int argc, char *argv[])
{
  // variables del programa
  pthread_attr_t attr;
  pthread_t threads[2];
  struct sched_param param;
  int i;

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

  // crear hilo de mayor prioridad
  pthread_attr_getschedparam(&attr, &param);
  param.sched_priority = 200;
  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 = 100;
  pthread_attr_setschedparam(&attr, &param);

  pthread_create(&threads[1], &attr, tarea2, NULL);

  // esperar a que finalicen
  for (i=0; i<2; i++) pthread_join(threads[i], NULL);

  return 0;
}
```

Anexo III – Secuencias en FreeRTOS

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

#define CANT_TAREAS 3
#define PRIMERA_TAREA 3
#define VECES(num) (num == 3 ? 3 : 1)
#define SIGUIENTE(num) (num == 1 ? 2 : num == 2 ? 3 : 1)

SemaphoreHandle_t sem[CANT_TAREAS];

void TareaGenerica(void* arg){
  int *param, num, id, sig, impresiones = 0;
  param = (int*) arg, id = *param, num = id+1, sig = SIGUIENTE(num);

  if (num == PRIMERA_TAREA)
    for (int i=0; i<VECES(num); i++) xSemaphoreGive(sem[id]);

  while(true){
    if (xSemaphoreTake(sem[id], (TickType_t) 15) == pdTRUE){
      Serial.print("tarea "); Serial.println(num);
      impresiones++;

      if (impresiones == VECES(num)) {
        for (int i=0; i<VECES(sig); i++) xSemaphoreGive(sem[sig-1]);
        impresiones = 0;
      }
    }

    vTaskDelay(100 / portTICK_PERIOD_MS);
  }
}

void setup() {
  int* ids = (int*) malloc(sizeof(int) * CANT_TAREAS);
  Serial.begin(9600);

  for (int i=0; i<CANT_TAREAS; i++){
    sem[i] = xSemaphoreCreateCounting(VECES(i+1), 0);
    ids[i] = i;
  }

  xTaskCreate(TareaGenerica, "tarea 1", 128, &ids[0], 1, NULL);
  xTaskCreate(TareaGenerica, "tarea 2", 128, &ids[1], 1, NULL);
  xTaskCreate(TareaGenerica, "tarea 3", 128, &ids[2], 1, NULL);
}
```

Anexo IV – Mutex en FreeRTOS

Se propone desarrollar tres tareas que operen sobre una misma variable global.

- Tarea 1: mayor prioridad, cada 100 ms incrementa en uno la variable global.
- Tarea 2: menor prioridad, cada 50 ms imprimir valor e incrementarla en dos.
- Tarea 3: exactamente igual a la anterior, misma prioridad y período.

Se obtuvo el siguiente resultado: **1, 3, 5, 7, 10, 12, 14, 16…**

En el instante t=0, se ejecuta primero Tarea 1, quedando la variable en 1. Luego se ejecutan las Tareas 2 y 3, que imprimen 1 y 3, dejándola en valor 5.

En el instante t=50 ms, se ejecutan nuevamente las Tareas 2 y 3, que imprimen los valores 5 y 7, dejándola en 9. Para t=100 ms, se ejecuta Tarea 1, quedando en valor 10, y luego las Tareas 2 y 3, imprimiendo 10 y 12, dejándola en valor 14. Verificado.

```
void setup() {
  Serial.begin(9600);
  mutex = xSemaphoreCreateMutex();

  xTaskCreate(tarea1, "tarea 1", 128, NULL, 200, NULL);
  xTaskCreate(tarea2, "tarea 2", 128, NULL, 100, NULL);
  xTaskCreate(tarea2, "tarea 3", 128, NULL, 100, NULL);
}

void tarea1(void* arg){

  while(1){
    if (xSemaphoreTake(mutex, 15) == pdTRUE){
      varCompartida++;
      xSemaphoreGive(mutex);

      vTaskDelay(100 / portTICK_PERIOD_MS);
    }
  }
}


void tarea2(void* arg){

  while(1){
    Serial.println(varCompartida);

    while(xSemaphoreTake(mutex, 15) == pdFALSE);
    varCompartida += 2;
    xSemaphoreGive(mutex);

    vTaskDelay(50 / portTICK_PERIOD_MS);
  }
}
```

```
// Bibliotecas
#include "Arduino_FreeRTOS.h"
#include <semphr.h>

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

// Variable compartida
int varCompartida = 0;

// Variable mutex
SemaphoreHandle_t mutex;
```

Anexo V – Timers en PIC

```
#include <htc.h>
#define _XTAL_FREQ 1000000

static void RELOJ_Init();

int main(){

    TRISA |= (1<<0)|(1<<1);          // Entradas
    TRISB &= ~((1<<4)|(1<<5));       // Salidas

    RB4 = 1; RB5 = 1;

    while (RA0 == 1 && RA1 == 1);    // Esperar por pulsador

    RELOJ_Init();
    while(1);
    return 0;
}

static void RELOJ_Init(){
    T0CS = 0;

    PS2 = 1; PS1 = 1; PS0 = 1;
    PSA = 0;

    T0IE = 1;           // Interrupción de Timer 0 Overflow
    GIE = 1;            // Interrupciones generales

    // Establecer valor de inicio para interrumpir a los 250 ms (aprox)
    // 256 - 244 = 12, ya que 4 us * 256 * 244 us = 249 856 us
    TMR0 = 12;
}

void interrupt manejador(void){

    RB4 = RB4 ? 0 : 1;
    RB5 = RB5 ? 0 : 1;

    T0IF = 0;
    TMR0 = 12;
}
```