Estructura y funcionalidades del OS a implementar

Mg. Ing. Gonzalo E. Sanchez
Esp. Ing. Hanes N. Sciarrone
MSE - 2023

Implementación de Sistemas Operativos (I)

Estructura y funcionalidades del OS

- Generalidades
- Lineamientos
- Detalles de implementación

Generalidades

RTOS en sistemas embebidos

- Se denomina Real Time OS (RTOS) al OS pensado para atender a aplicaciones de tiempo real.
- En un OS de propósito general los tiempos entre atenciones de interrupciones y cambios de contexto es variable.
- Existen aplicaciones donde el tiempo debe ser determinado.
- Ejemplo: La estabilidad de un lazo de control digital depende directamente del tiempo de muestreo.

RTOS en sistemas embebidos

- Un RTOS es determinista.
- Esto significa que está especificado el tiempo que requiere un cambio de contexto entre tareas, atención de ISR, etc.
- Puede ser lento, pero es conocido.
- Puede no ser un valor definido, pero existe una ventana de tiempo dada.

RTOS en sistemas embebidos

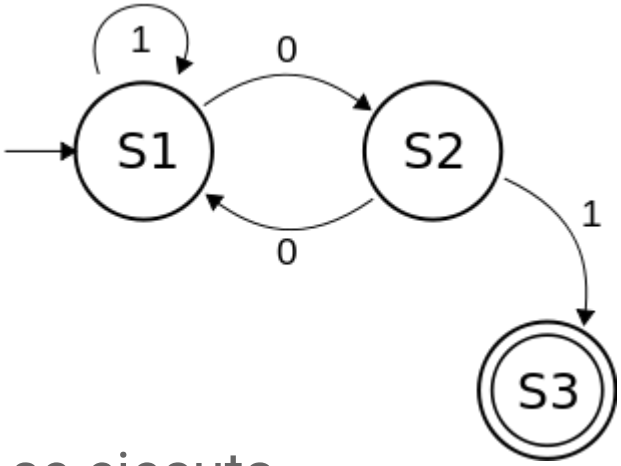
- El RTOS tiene herramientas y recursos para asegurar la respuesta del sistema, con una tolerancia determinada.
- Según los requerimientos y las tolerancias permitidas podemos dividir dos grandes grupos:
  - Soft Real Time.
  - Hard Real Time.

RTOS en sistemas embebidos

- Los tiempos que deben cumplirse se denominan dead-lines.
- En un sistema Hard Real Time, no cumplir un dead-line implica una falla de sistema.
- En un sistema Soft Real Time, se tolera no cumplir algún dead-line, pero implica que el sistema se degrade.

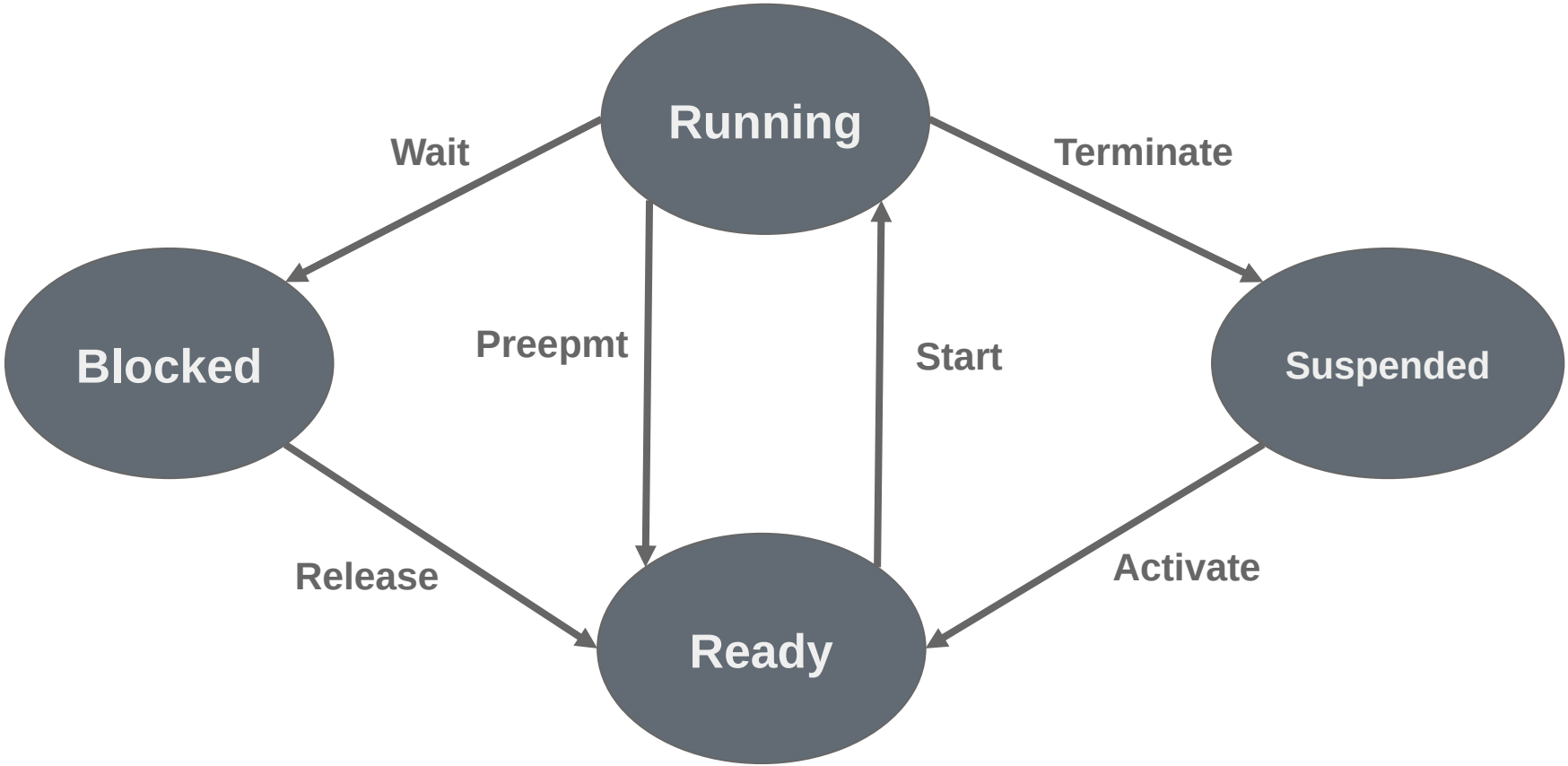
Tareas y scheduling

- Las tareas tienen estados:
  - Running
  - Waiting
  - Ready
  - Suspended



- Esto define una máquina de estados que se ejecuta como parte del Kernel de RTOS.

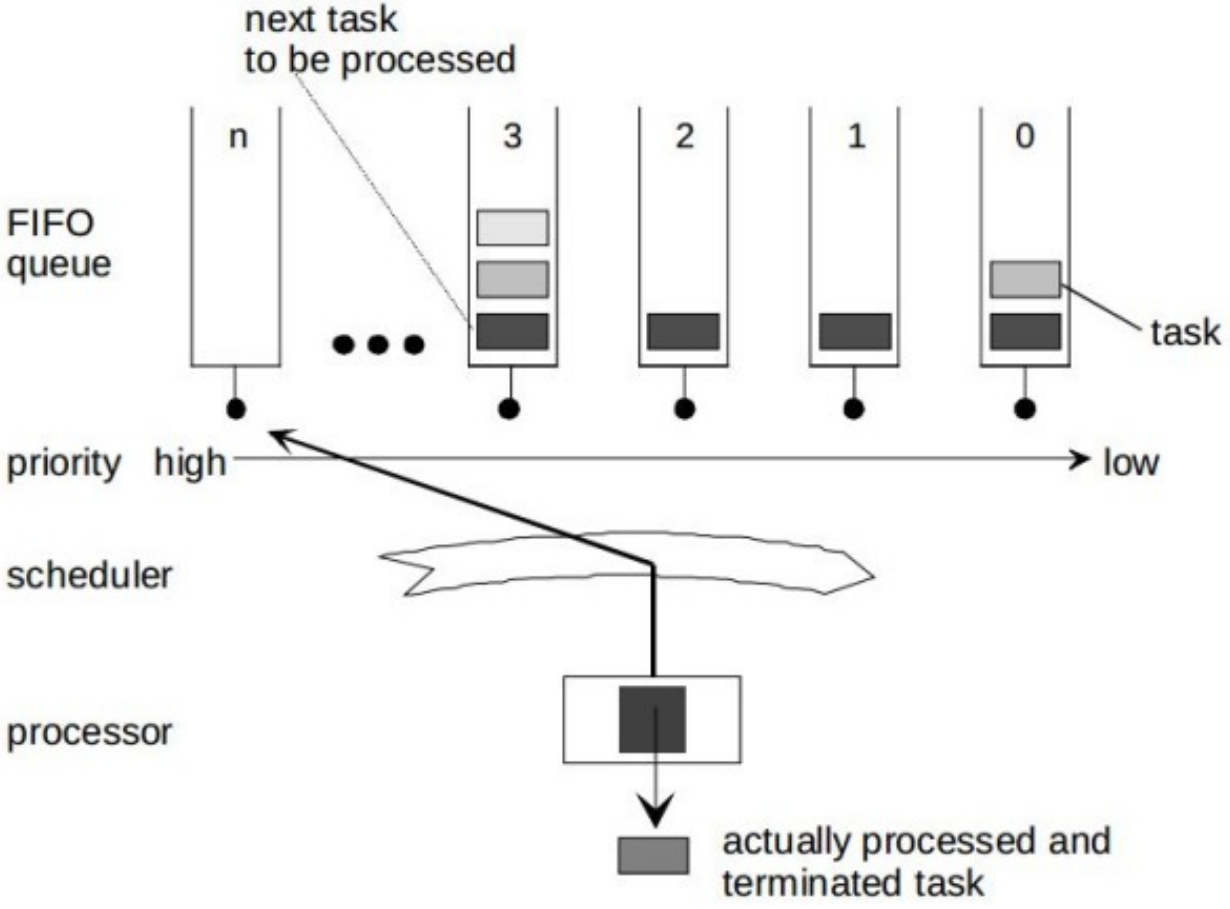
Tareas y scheduling



Tareas y scheduling

- El estado **suspended** hace que la tarea tenga un comportamiento similar a **blocked**.
- La diferencia es que no existe un timeout o evento para pasar a ready.
- No se ingresa al scheduler, pero se mantiene la memoria asociada (stack).
- Solamente podrá pasar a **ready** cuando explícitamente se active la misma con una llamada de API correspondiente.

Tareas y scheduling



Lineamientos

Lineamientos

- El OS a implementar seguirá los lineamientos descritos en el documento “Requerimientos de implementación para Sistema Operativo”
- Es un documento guía para guiar al desarrollador en su implementación.
- Se tiene libertad para todo aspecto que no esté listado en ese documento.
- Deben cumplirse los requerimientos mínimos.

Recomendaciones

- Es preferible tener el scheduler encapsulado en una función.
- Eso dará libertad de cambiar la política de scheduling sin tocar otras partes del kernel.
- Siempre tener en cuenta la escalabilidad.
- Si la implementación en este estadio es rebuscada, seguramente necesita de un Refactoring.
- Comentar, comentar, comentar.
- Opcional: Doxygen.

Recomendaciones

- Tener una estructura que define campos de control tanto para las tareas y para el OS da flexibilidad y escalabilidad.
- Mantener siempre una política de encapsulamiento.
- No es buena práctica dejar a la vista del usuario funciones y variables propias del kernel.
- Hacer uso extensivo de la palabra reservada **static**.
- Hacer uso extensivo del atributo del compilador **WEAK**.

Recomendaciones

```
/* Ejemplo estructura minima para tarea */
struct _tarea  {
    uint32_t stack[STACK_SIZE/4];
    uint32_t stack_pointer;
    void *entry_point;
    estadoTarea estado;                          //definido como enum
    uint8_t prioridad;
uint8_t id;
    uint32_t ticks_bloqueada;
    char nombre[TASK_NAME_SIZE];
}
```

Recomendaciones

```
/* Ejemplo estructura minima para control de OS */
struct _osControl  {
    estadoOS estado_sistema;  //definido como enum
    bool schedulerIRQ;        //scheduling al volver de IRQ
    tarea *tarea_actual;
    tarea *tarea_siguiente;
    int32_t error;                //ultimo error generado
}
```

Detalles para la implementación

Detalles para la implementación

- La palabra reservada **static** no solamente sirve para crear variables locales con persistencia.
- Se pueden definir funciones **static**.
- Este tipo de funciones solo son visibles dentro del archivo donde están definidas.
- Muy útil para hacer ocultamiento de información.
- El lenguaje C no está orientado a objetos, pero se puede obtener algo similar.

Lineamientos

```
/* Archivo: funcion.c
   La funcion foo() solo es accesible dentro del mismo
archivo (funcion.c) */
static uint32_t foo(uint32_t a, uint32_t b)  {
       /*
        -- codigo de la funcion --
       */
       return bar;
       }
```

Detalles para la implementación

- Otra herramienta útil es el mecanismo de atributos definido para compiladores GNU.
- Permite al desarrollador dar más información al compilador sobre funciones específicas.
- Tiene utilidad para checkeo de errores entre otras cosas.
- Utilizaremos el atributo **weak**.

Detalles para la implementación

- El atributo **weak** permite una definición “débil” de una función.
- Quiere decir que esta función puede ser redefinida en otro lugar del código.
- La redefinición no debe tener el atributo weak.
- Esto es recomendable para funciones que pueden o no ser redefinidas por el usuario del OS.
- Ejemplos: hooks de sistema.

Lineamientos

```
/* La funcion foo() esta definida como weak. Quiere decir
      que puede ser redefinida en otro archivo, pero sin
utilizar el atributo weak */

__attribute__((weak)) void foo(void)  {
   while(1);
}
```

Estructura y funcionalidades

HANDS ON

1. Estructura de control
2. Estructura de tareas
3. Implementar estados READY y RUNNING

Gracias.