RTOS: Introducción y primeros pasos

Enfocado en Xinu: Un sistema operativo pequeño y elegante

Rafael Ignacio Zurita

Depto. Ingeniería de Computadoras

November 20, 2020

RTOS: Introducción y primeros pasos

- Definición de Sistema de Tiempo Real y RTOS
- ¿Por qué utilizar un RTOS?
- 3 maneras de desarrollar un sistema embedido
- Introducción a Xinu (características)
  - Servicios que proporciona al desarrollador (sleep)
- Gestión de Tareas en Xinu (planificador/scheduler RTOS)
- Xinu en ejemplos: Tareas
  - Tareas concurrentes
  - Reuso de código
  - Diseño de tiempo real: prioridades fijas
- Xinu en ejemplos: Sincronización y Comunicación de Tareas en Xinu
  - Sincronización: semáforos (ej. productor/consumidor)
  - Región crítica: solución con mutex
  - Comunicación: pasaje de mensajes con mailbox
  - Buffers y colas: ports
- Recursos

RTOS: Introducción y primeros pasos

- Definición de Sistema de Tiempo Real y RTOS
- ¿Por qué utilizar un RTOS?
- 3 maneras de desarrollar un sistema embedido
- Introducción a Xinu (características)
  - Servicios que proporciona al desarrollador (sleep)
- Gestión de Tareas en Xinu (planificador/scheduler RTOS)
- Xinu en ejemplos: Tareas
  - Tareas concurrentes
  - Reuso de código
  - Diseño de tiempo real: prioridades fijas
- Xinu en ejemplos: Sincronización y Comunicación de Tareas en Xinu
  - Sincronización: semáforos (ej. productor/consumidor)
  - Región crítica: solución con mutex
  - Comunicación: pasaje de mensajes con mailbox
  - Buffers y colas: ports
- Recursos

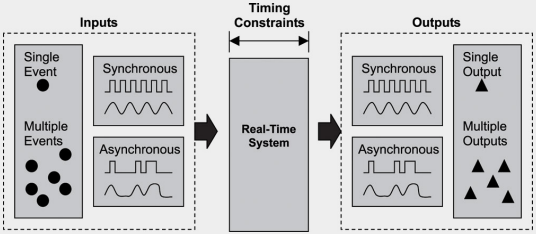
» Sistema de Tiempo Real (RTS)

RTS cuando la correctitud del sistema depende:

(1) del resultado lógico de la computación, y

(2) del tiempo que toma producir el resultado.

FALLO *Si las restricciones de tiempo no se cumplen, el sistema falló.*



- **Soft Real Time**: cuando se acepta que los requerimientos de tiempo límite para producir el resultado pueden no cumplirse en algunas ocasiones.
- **HARD Real Time**: cuando los requerimientos de tiempo límite de respuesta a un evento deben cumplirse siempre.

Manipulador (Ejemplo): `https://youtu.be/v26tDdINop4?t=46`

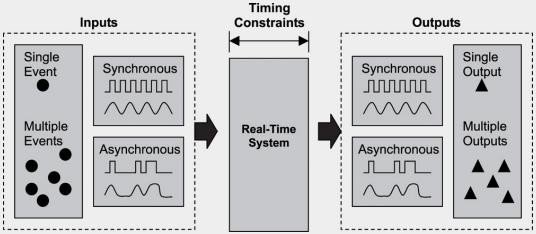
» Sistema de Tiempo Real (RTS)

RTS cuando la correctitud del sistema depende:

(1) del resultado lógico de la computación, y

(2) del tiempo que toma producir el resultado.

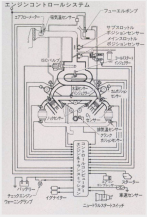
FALLO *Si una restricción de tiempo no se cumplió, el sistema falló (aunque se haya cumplido billones de veces anteriormente).*



Example (1) – Engine Management System

System Components
- control computer (ECU)
- many sensors
  - crank position sensor
  - air flow meter
  - intake temperature sensor
  - throttle sensor
- some actuators

Basic Functions of the Control System
- to calculate fuel injection volume and ignition timing, and to control the actuators in every rotation cycle



Example (2) – ABS

Function of ABS ABS = Anti-lock Breaking System
- The speed of the car and the rotational speed of the wheel are monitored, and a skid is detected.
- When a skid is detected, hydraulic pressure to the brake is reduced to stop the skid.
- The system is relatively simple, but is becoming more complex, recently.

Safety Requirement (Example) and Fail-Safe Design
- Continuous reduction of hydraulic pressure causes non-braking.
- If some fault is detected, ABS stops functioning. Then, the brake works though a skid cannot be avoided.

➔ *fail-safe design*

Example (3) – Airbag Control

Function of Airbag Control
- Airbag control system monitors various sensors including accelerometers and detects a collision.
- If a collision is detected, the ignition of a gas generator propellant is triggered to inflate a bag.

Real-Time Constraint
- The trigger must be within 10-20*msec.* after the collision.

Safety Requirements
- Fail-safe design cannot be applied.

! even harder than ABS

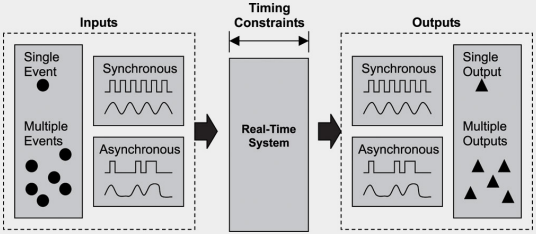
» Sistema de Tiempo Real (RTS)

RTS cuando la correctitud del sistema depende:

(1) del resultado lógico de la computación, y

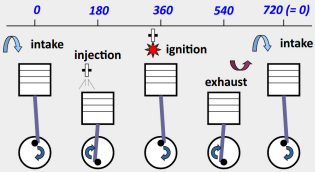
(2) del tiempo que toma producir el resultado.

FALLO *Si una restricción de tiempo no se cumplió, el sistema falló (aunque se haya cumplido billones de veces anteriormente).*



Timing Behavior of Engine Management System

- When rotation speed is 6000*rpm*, one cycle is 20*msec.*
- Timing precision of the ignition is 10*µsec.* order.

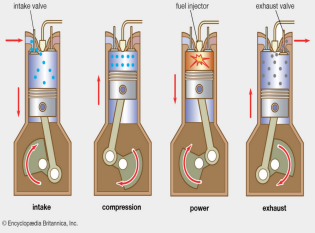


Required Real-Time Property (Example)

- The calculation of the fuel injection volume *must be finished* before the injection timing.
- The calculation of the ignition timing *must be finished* before the ignition timing.
- Calculating too early has no additional value.

Safety Requirement (Example)

- Missing an ignition *must not happen*, because inflammable gas is emitted outside of the engine and can lead to a fire (because catalyst burns).
- If the ignition plug of a cylinder is broken, fuel *must not be injected* to the cylinder.
- *The engine management system monitors the ignition plug and stops the injection if the plug is broken.*

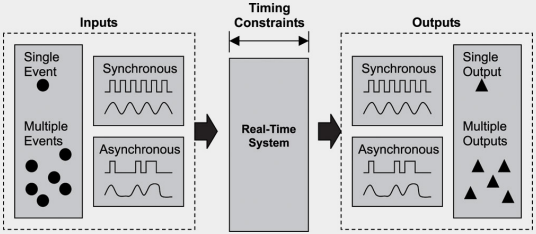


» Sistema de Tiempo Real (RTS)

RTS cuando la correctitud del sistema depende:

(1) del resultado lógico de la computación, y

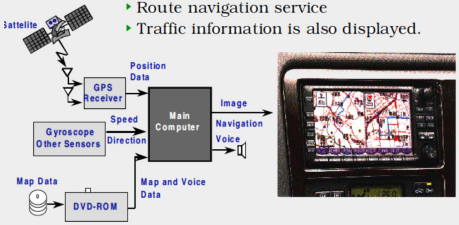
(2) del tiempo que toma producir el resultado.



El siguiente ejemplo: **¿Es un sistema de tiempo real?**

Example (4) – Car Navigation System

- The current position of the car obtained from GPS, gyroscope, and others is displayed with the map.
  - Route navigation service
  - Traffic information is also displayed.



Car Navigation System

- largest and most complicated software in a car
- large computing power required
- moderate reliability, real-time property still required

» Sistema Operativo de Tiempo Real

RTOS incluyen las siguientes características:

(1) Cambio de contexto veloz.

(2) Multitarea con comunicación entre procesos (entre tareas).

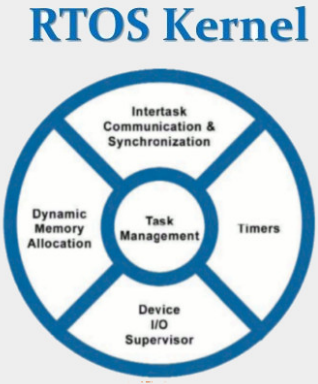
(3) Capacidad para responder a las interrupciones externas rapidamente.

(4) Tamaño reducido.

(5) Planificador apropiativo (preemptive scheduling).

(6) Otros.

William Stallings. Operating Systems



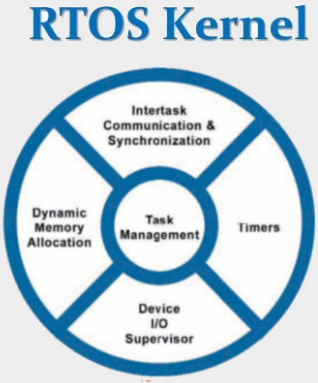
» Sistema Operativo de Tiempo Real

RTOS Definido por su planificador (scheduler).
El kernel de un RTOS tiene 3 funciones:

(1) El kernel se encarga de las rutinas de atención de interrupciones.

(2) El kernel provee un planificador que determina la secuencia en que las tareas son ejecutadas.

(3) El kernel provee mecanismos de comunicación entre procesos (entre tareas).



Alan Clements. Microprocessor System Design: 68000 hardware, software and interfacing

» Sistema Operativo de Tiempo Real

RTOS tiene que soportar algún modo de compartir recursos y los requerimientos de tiempo de las tareas.

* Sus funciones son:

(1) Planificador de tareas.

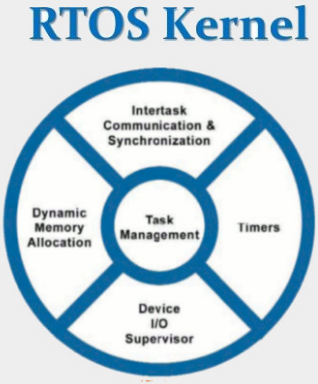
(2) Manejo de interrupciones.

(3) Administración de memoria.

(4) Compartir código entre tareas.

(5) Compartir dispositivos entre tareas.

(6) Sincronización y Comunicación entre procesos.

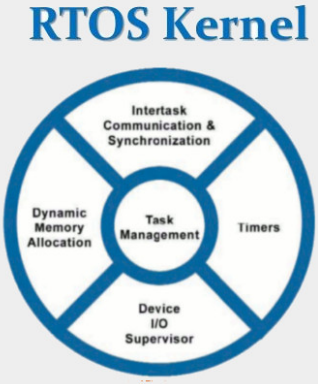


Stuart Bennet. Real-Time Computer Control: An Introduction

» Sistema Operativo de Tiempo Real

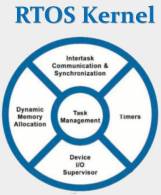
RTOS *Un RTOS es un conjunto de herramientas que posibilita la confección de un sistema: posiblemente embebido, posiblemente de tiempo real, multitarea, posiblemente predecible, y posiblemente determinista. (by pse2020)*

- Un RTOS provee:
  1. Planificador de tareas apropiativo.
  2. Administración de memoria.
  3. Administración de la E/S.
  4. Comunicación entre procesos y datos compartidos.
- Un RTOS debe:
  1. Ser predecible.
  2. Determinista.
- Un RTOS suele:
  1. Estar preparado para microcontroladores (o microprocesadores).
  2. Venir en modo código fuente.
  3. La aplicación embebida de tiempo real se desarrolla mezclada con el código fuente RTOS.
  4. Ser predecible asignado prioridades fijas en tiempo de compilación.



» Sistema Operativo de Tiempo Real

Dos definiciones extras (Richar Barry y olvidado):



1. A Real Time Operating System is an operating system that is optimised for use in embedded/real time applications. Their primary objective is to ensure a timely and deterministic response to events. An event can be external, like a limit switch being hit, or internal like a character being received.

   Using a real time operating system allows a software application to be written as a set of independent tasks. Each task is assigned a priority and it is the responsibility of the Real Time Operating System to ensure that the task with the highest priority that is able to run is the task that is running.

2. An RTOS is an operating system whose internal processes are guaranteed to be compliant with (hard or soft) realtime requirements. The fundamental qualities of a n RTOS are: Predictabe, and Deterministic.

IMPORTANTE An RTOS is not a magic wand, your system will not be "realtime" just because you are using an RTOS, what matters is your system design. The RTOS itself is just a toolbox that offers you the required tools for creating a realtime system, you can use the tools correctly or in the wrong way.

RTOS: Introducción y primeros pasos

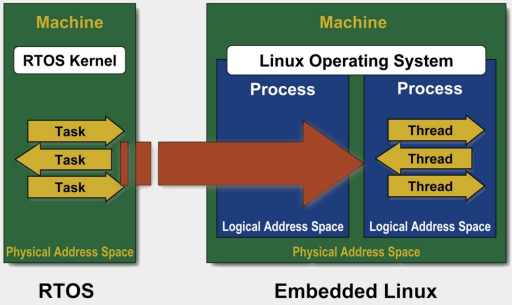
- Definición de Sistema de Tiempo Real y RTOS
- ¿Por qué utilizar un RTOS?
- 3 maneras de desarrollar un sistema embedido
- Introducción a Xinu (características)
  - Servicios que proporciona al desarrollador (sleep)
- Gestión de Tareas en Xinu (planificador/scheduler RTOS)
- Xinu en ejemplos: Tareas
  - Tareas concurrentes
  - Reuso de código
  - Diseño de tiempo real: prioridades fijas
- Xinu en ejemplos: Sincronización y Comunicación de Tareas en Xinu
  - Sincronización: semáforos (ej. productor/consumidor)
  - Región crítica: solución con mutex
  - Comunicación: pasaje de mensajes con mailbox
  - Buffers y colas: ports
- Recursos

» ¿Por qué utilizar un RTOS?

Resumen *Porque un RTOS es una excelente herramienta para el desarrollo de un sistema embebido, posiblemente RT.*

Modelo

1. El problema se divide en tareas independientes.
2. Si existen tareas relacionadas, un RTOS provee comunicación y sincronización entre tareas.
3. Las regiones críticas pueden ser resueltas con componentes del RTOS.
4. El sistema puede ser diseñado con tareas actividadas mediante eventos (activadas por el scheduler).
5. Las tareas se planifican de manera predecible en base a prioridades.
6. Reuso de código para diferentes tareas.



RTOS: Introducción y primeros pasos

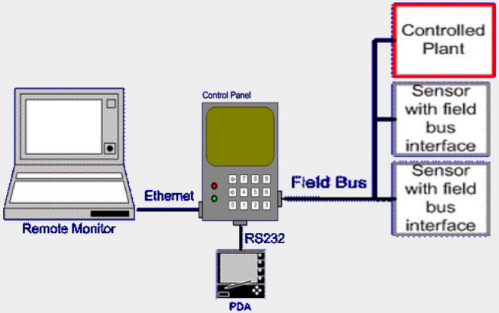
- Definición de Sistema de Tiempo Real y RTOS
- ¿Por qué utilizar un RTOS?
- **3 maneras de desarrollar un sistema embedido**
- Introducción a Xinu (características)
  - Servicios que proporciona al desarrollador (sleep)
- Gestión de Tareas en Xinu (planificador/scheduler RTOS)
- Xinu en ejemplos: Tareas
  - Tareas concurrentes
  - Reuso de código
  - Diseño de tiempo real: prioridades fijas
- Xinu en ejemplos: Sincronización y Comunicación de Tareas en Xinu
  - Sincronización: semáforos (ej. productor/consumidor)
  - Región crítica: solución con mutex
  - Comunicación: pasaje de mensajes con mailbox
  - Buffers y colas: ports
- Recursos

» 3 maneras de desarrollar un sistema embebido

Ejemplo *Un sistema embebido que controla una planta quimica.*

Presenta una interfaz al usuario con LCD, teclado y leds.

Tambien una interfaz web para verificar el estado de la planta remotamente.



System context

» 3 maneras de desarrollar un sistema embebido

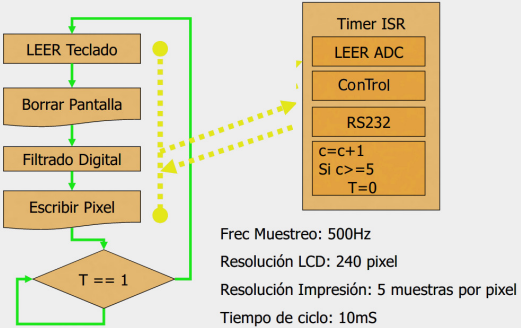
Ejemplo *Un sistema embebido que controla una planta (o experimento químico). Presenta una interfaz al usuario con LCD, teclado y leds. Tambien una interfaz web para control remoto.*

Loop+Timers

1. Las tareas de interfaz fisica se realizan en un bucle infinito en main.
2. Dos tareas controladas por software timers: una que lee los sensores y controla la planta; y otra que recibe o envía por la interfaz web.

RTOS

1. El probema se divide en tareas independientes.
2. Cada tarea se ejecuta sólo si sucede un evento que les interese.
3. Cada tarea tiene una prioridad.

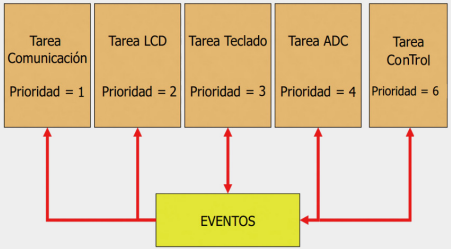


Frec Muestreo: 500Hz

Resolución LCD: 240 pixel

Resolución Impresión: 5 muestras por pixel

Tiempo de ciclo: 10mS



» 3 maneras de desarrollar un sistema embebido

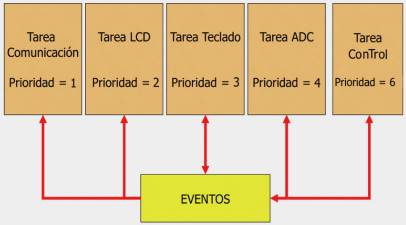
Ejemplo *Un sistema embebido que controla una planta (o experimento químico). Presenta una interfaz al usuario con LCD, teclado y leds. Tambien una interfaz web para control remoto.*

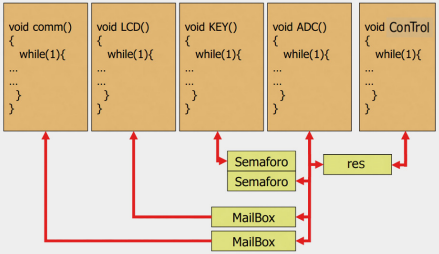
Diseño

1. El probema se divide en tareas independientes.
2. Cada tarea se ejecuta sólo si sucede un evento (o más) de interés;
3. o regularmente para, por ejemplo, adquirir señales.
4. Cada tarea tiene una prioridad.

RTOS

1. Las tareas Control, LCD y Comunicaciones duermen esperando un evento.
2. Las tareas KEY y ADC adquieren y procesan señales.
3. Las tareas KEY y ADC generan eventos si se debe *actuar*.
4. Las tareas son ejecutadas concurrentemente por el RTOS.
5. Los eventos son semáforos y mailboxes provistos por el RTOS.
6. El planificador del RTOS despierta las tareas dormidas cuando el evento esperado está listo.
7. El planificador RTOS ejecuta las tareas en base a la prioridad.
8. El planificador RTOS permite compartir recursos entre tareas de manera segura.





RTOS: Introducción y primeros pasos

- Definición de Sistema de Tiempo Real y RTOS
- ¿Por qué utilizar un RTOS?
- 3 maneras de desarrollar un sistema embedido
- **Introducción a Xinu (características)**
  - Servicios que proporciona al desarrollador (sleep)
- Gestión de Tareas en Xinu (planificador/scheduler RTOS)
- Xinu en ejemplos: Tareas
  - Tareas concurrentes
  - Reuso de código
  - Diseño de tiempo real: prioridades fijas
- Xinu en ejemplos: Sincronización y Comunicación de Tareas en Xinu
  - Sincronización: semáforos (ej. productor/consumidor)
  - Región crítica: solución con mutex
  - Comunicación: pasaje de mensajes con mailbox
  - Buffers y colas: ports
- Recursos

» Xinu

Xinu Es un **sistema operativo** pequeño y elegante (fácil de comprender), desarrollado originalmente por Douglas Comer en la Universidad de Purdue, a fines de los 70.

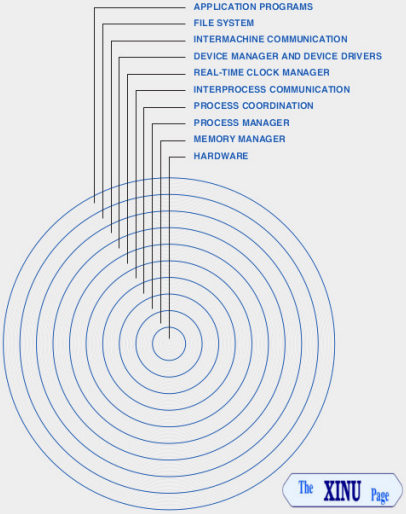
Versiones Actualmente existen versiones para *x86 (PC), ARM, MIPS, AVR, y maquinas virtuales*.

USO *Académico, en investigación y en la industria*.

Como el tamaño del código fuente es pequeño Xinu es adecuado en sistemas embebidos.

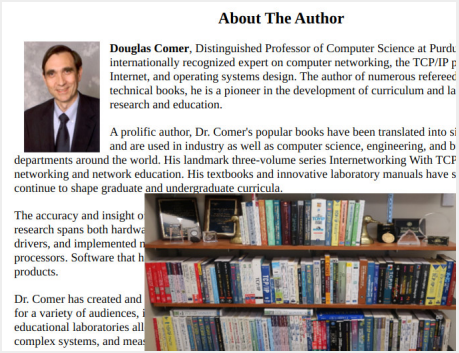
web Mas información:

1. https://xinu.cs.purdue.edu/
2. http://se.fi.uncoma.edu.ar/xinu-avr/
3. *Douglas Comer, Operating System Design - The Xinu Approach, Second Edition CRC Press, 2015. ISBN 9781498712439*



» Douglas Comer

- Ha escrito una decena de libros sobre Sistemas Operativos, Arquitectura de Computadoras y Redes.
- Considerado el padre académico de internet:
  - Sus 3 vólumenes sobre redes TCPI/IP en los 80 son considerados la autoridad de referencia de los protocolos de Internet.
  - Jugaron un rol clave en popularizar esos protocolos, haciéndolos entendibles a la generación de ingenieros y técnicos que se formaron previo a internet.
  - https://www.cs.purdue.edu/homes/comer/
  - comer@purdue.edu



About The Author

Douglas Comer, Distinguished Professor of Computer Science at Purdu
internationally recognized expert on computer networking, the TCP/IP p
Internet, and operating systems design. The author of numerous refereed
technical books, he is a pioneer in the development of curriculum and la
research and education.

A prolific author, Dr. Comer's popular books have been translated into si
and are used in industry as well as computer science, engineering, and b
departments around the world. His landmark three-volume series Internetworking With TCF
networking and network education. His textbooks and innovative laboratory manuals have s
continue to shape graduate and undergraduate curricula.

The accuracy and insight o
research spans both hardwa
drivers, and implemented n
processors. Software that h
products.

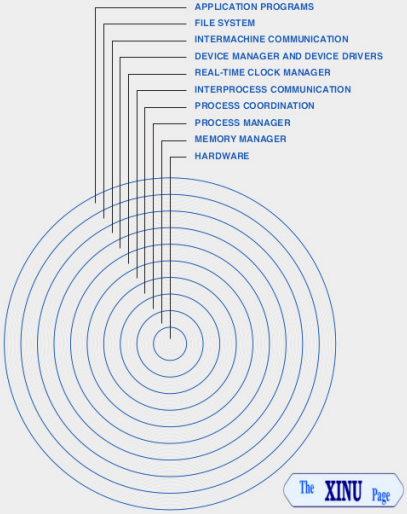
Dr. Comer has created and
for a variety of audiences,
educational laboratories all
complex systems, and meas

» Xinu

Xinu Es un **sistema operativo** pequeño y elegante (fácil de comprender), desarrollado originalmente por Douglas Comer en la Universidad de Purdue, a fines de los 70.

Características Xinu provee:

1. Creación dinámica de procesos (threads).
2. Asignación dinámica de memoria.
3. Administración del real-time clock.
4. Coordinación y sincronización de procesos.
5. Un planificador apropiativo multi-tarea (preemptive multi-tasking).
6. Sistemas de archivos locales y remotos.
7. Un shell (si es necesario).
8. Funciones de E/S (API) independientes del dispositivo.
9. Una pila de protocolos TCP/IP.



» Xinu

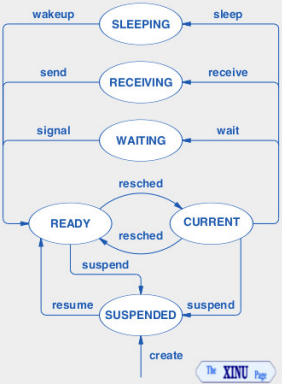
OS style Los sistemas operativos multitarea pueden ser:

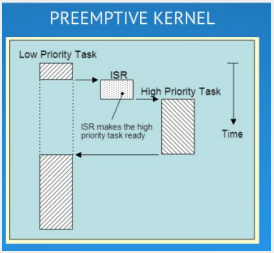
- de Tiempo Compartido
- de Tiempo-Real

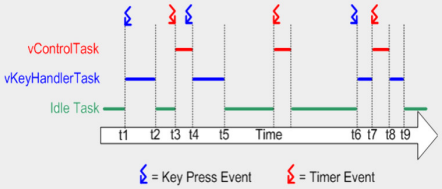
Xinu RTOS Xinu utiliza fixed-priority, y coloca en ejecución al proceso de mas alta prioridad, lo que lo categoriza como RTOS.

Planificador Su planificador (scheduler) puede ser apropiativo (preemptive) o cooperativo.

Concurrente El desarrollo de la aplicación se debe realizar teniendo en cuenta detalles de la programación concurrente.







RTOS: Introducción y primeros pasos

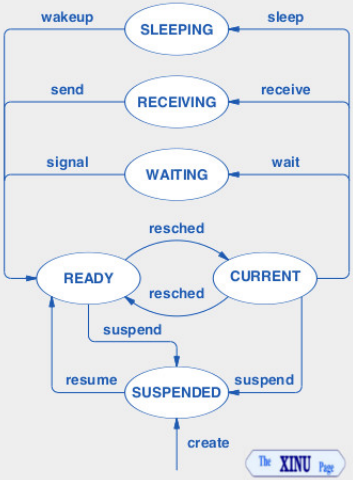
- Definición de Sistema de Tiempo Real y RTOS
- ¿Por qué utilizar un RTOS?
- 3 maneras de desarrollar un sistema embedido
- Introducción a Xinu (características)
  - Servicios que proporciona al desarrollador (sleep)
- **Gestión de Tareas en Xinu (planificador/scheduler RTOS)**
- Xinu en ejemplos: Tareas
  - Tareas concurrentes
  - Reuso de código
  - Diseño de tiempo real: prioridades fijas
- Xinu en ejemplos: Sincronización y Comunicación de Tareas en Xinu
  - Sincronización: semáforos (ej. productor/consumidor)
  - Región crítica: solución con mutex
  - Comunicación: pasaje de mensajes con mailbox
  - Buffers y colas: ports
- Recursos

» Tareas en Xinu

modelo **Xinu sigue el modelo de threads.**

tareas
* Cada tarea (proceso o thread) tiene su propia pila y contexto, pero comparte el segmento de código y segmento de datos.
* Una tarea es una función en C. Se crea con la llamada al sistema **create()**, el cual recibe como argumento el nombre de la función.
* Se pueden crear muchas tareas a partir de una misma función (posiblemente con diferentes argumentos).

API
* **create(args)** y **resume(pid)** crean una tarea y la colocan en estado de *LISTO*.
* **sleep(n)** y **sleepms(n)** colocan la tarea en estado *DURMIENDO*.
* **kill(d)** permite matar un proceso. Por lo que un proceso puede finalizar con **kill(**getpid()**)**.
* **suspend(pid)** suspende un proceso.



» Xinu: administrador de memoria

cor de memoria

En Xinu existe una API para requerir memoria en tiempo de ejecución.

* Utilizado principalmente por create y kill, para asignar pila a un nuevo proceso (thread).
* Se puede solicitar un bloque de memoria en ejecución, pero no es recomendado en embebidos y RT.

API

```
getstk(n);
freestk(b,s);

getmem(n);
freemem(b,s);
```

» Xinu: dispositivo de E/S console

CONSOLE La consola en xinu-avr es la terminal del usuario, y está controlada por el driver uart.

- Xinu ofrece algunas funciones similares al estandar de C, que utilizan la consola (serial/uart en avr) de manera predeterminada.
- Internamente Xinu utiliza una capa de software de E/S independiente del dispositvo (con funciones open, read, write, seek, putc, close, etc).

API de algunas funciones de biblioteca de Xinu

```
/* son equivalentes */
putchar(ch)
putc(CONSOLE,(ch))

/* son equivalentes */
getchar(ch)
getc(CONSOLE,(ch))

printf(args);

/* todas la anteriores emiten la salida a través del serial uart en avr */
```

RTOS: Introducción y primeros pasos

- Definición de Sistema de Tiempo Real y RTOS
- ¿Por qué utilizar un RTOS?
- 3 maneras de desarrollar un sistema embedido
- Introducción a Xinu (características)
  - Servicios que proporciona al desarrollador (sleep)
- Gestión de Tareas en Xinu (planificador/scheduler RTOS)
- **Xinu en ejemplos: Tareas**
  - Tareas concurrentes
  - Reuso de código
  - Diseño de tiempo real: prioridades fijas
- Xinu en ejemplos: Sincronización y Comunicación de Tareas en Xinu
  - Sincronización: semáforos (ej. productor/consumidor)
  - Región crítica: solución con mutex
  - Comunicación: pasaje de mensajes con mailbox
  - Buffers y colas: ports
- Recursos

» Xinu: crear y ejecutar 3 procesos concurrentes

Secuencia

1. Al iniciar Xinu existen 2 procesos en ejecución: main() y null().
2. Cada proceso tiene su propia pila y contexto.
3. main() entonces crea 3 nuevos procesos y finaliza.
4. send_a(), led() y led_placa() son procesos con loop infinitos.
5. Los 4 procesos se ejecutan indefinidamente de manera concurrente.
6. Video demostrativo

```
/*------------------------------------------------------------------------
 * main  --  example of creating 3 concurrent processes in Xinu
 *------------------------------------------------------------------------
 */
void  main(void)
{
      resume( create(send_a, 128, 20, "process 1", 0) );
      resume( create(led, 128, 20, "process 2", 0) );
      resume( create(led_placa, 128, 20, "process 3", 0) );
}

/*------------------------------------------------------------------------
 * process 1: repeatedly emit 'A' on the console without terminating
 *------------------------------------------------------------------------
 */
void  send_a(void)
{
      while (1) {                         /* infinite loop in process 1 */
            putc(CONSOLE, 'A');
      }
}

/*------------------------------------------------------------------------
 * process 2: -- parpadear les pb4 (pin 12) cada 400 ms infinitamente
 *------------------------------------------------------------------------
 */
void  led(void)
{
      while (1) {                         /* infinite loop in process 2 */
            gpio_arduino_write(12, 1);
            sleepms(400);
            gpio_arduino_write(12, 0);
            sleepms(400);
      }
}

/*------------------------------------------------------------------------
 * process 3: parpadear led de la placa cada 1 segundo infinitamente
 *------------------------------------------------------------------------
 */
void  led_placa(void)
{
      while (1) {                         /* infinite loop in process 3 */
            gpio_arduino_write(13, 1);
            sleep(1);
            gpio_arduino_write(13, 0);
            sleep(1);
      }
}
```

» Xinu: ejemplo 0
Crear y ejecutar una tarea

Descripción

1. **create()** crea el proceso *ex_0*
   - El código fuente de este proceso esta implementado en **example0()**.
   - Recibe 3 argumentos: nargs (la cantidad de argumentos), y 2 argumentos en args.

2. Para compilar esta aplicación junto con Xinu:

```
cd xinu-avr/
cd compile/
make clean
make
make flash
```

3. Los printf están configurados para utilizar el dispositivo CONSOLE en Xinu.

4. En xinu-avr, el dispositivo CONSOLE es el USART.

5. Para observar el output: *cutecom*

```
/* Basic Xinu app for quick start */

#include <xinu.h>


process example0(int nargs, char *args[])
{
        printf("  nargs = %d\n", nargs);
        printf("  args[0]: %d\n", args[0]);
        printf("  args[1]: %s\n", args[1]);

        return OK;
}

process main(void)
{
        /* We create() a new process. Arguments:
         *         process code: example0()
         *         stack size: 128 bytes
         *         priority: 50
         *         name: "ex0"
         *         arguments: 2
         *         first argument: 3 (int)
         *         second argument: "hello world"
         */

        resume(create(example0, 128, 50, "ex_0", 2, 3, "hello world"));

        /* Wait for example0 to exit */

        receive();
        printf("\n %s process has completed.\n", proctab[currpid].prname);

        return OK;
}
```

» Xinu: ejemplo 1
Utilizar putc con el dispositivo CONSOLE

Descripción

1. **main()** hace uso de la función de biblioteca **putc()**

2. Para compilar esta aplicación junto con Xinu:

```
cd xinu-avr/
cd compile/
make clean
make
make flash
```

3. Las funciones vistas hasta el momento:

```
create() : system call para crear un proceso y dejarlo en estado SUSPENDIDO
resume(): función de Xinu para colocar el proceso en estado de LISTO
nargs y args[]: argumentos de entrada de un proceso
putc(): función de biblioteca para enviar un char a un dispositivo
printf(): función de biblioteca para enviar un output a CONSOLE
```

4. En xinu-avr, el dispositivo CONSOLE es el USART.

5. Para observar el output: *cutecom*

```
/* ex1.c - main */

#include <xinu.h>

/*------------------------------------------------------------------------
 * main  --  write "hi" on the console
 *------------------------------------------------------------------------
 */
void    main(void)
{
        putc(CONSOLE, 'h'); putc(CONSOLE, 'i');
        putc(CONSOLE, '\r'); putc(CONSOLE, '\n');
}
```

» Xinu: ejemplo 2
Tareas concurrentes

Descripción

1. **main()** crea dos procesos: sndA y sndB.
   - Cada proceso tiene un bucle infinito.
   - Cada proceso envía de manera independiente una letra indefinidamente.
2. Para observar el output: *cutecom*
3. Observación: el codigo fuente de un proceso es de tipo void o process.

```
/* ex2.c - main, sndA, sndB */

#include <xinu.h>

void    sndA(void), sndB(void);

/*------------------------------------------------------------------------
 * main  --  example of creating processes in Xinu
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndA, 128, 20, "process 1", 0) );
        resume( create(sndB, 128, 20, "process 2", 0) );
}

/*------------------------------------------------------------------------
 * sndA  --  repeatedly emit 'A' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndA(void)
{
        while( 1 )
                putc(CONSOLE, 'A');
}

/*------------------------------------------------------------------------
 * sndB  --  repeatedly emit 'B' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndB(void)
{
        while( 1 )
                putc(CONSOLE, 'B');
}
```

» Xinu: ejemplo 3
Reuso de código

Descripción

1. **main()** crea y ejecuta dos procesos independientes que utilizan el mismo código (pero con diferentes argumentos).
   - Cada proceso recibe un argumento (letra A o B).
   - Cada proceso envía de manera independiente su letra indefinidamente.
2. Para observar el output: *cutecom*
3. Observación: el codigo fuente de un proceso es de tipo void o process.

```
/* ex3.c - main, sndch */

#include <xinu.h>

void    sndch(int nargs, char *args[]);

/*------------------------------------------------------------------------
 * main  --  example of 2 processes executing the same code concurrently
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndch, 128, 20, "send A", 1, 'A') );
        resume( create(sndch, 128, 20, "send B", 1, 'B') );
}

/*------------------------------------------------------------------------
 * sndch  --  output a character on a serial device indefinitely
 *------------------------------------------------------------------------
 */
void    sndch(int nargs, char *args[])

{
        char ch = args[0]; /* character to emit continuously */

        while ( 1 )
                putc(CONSOLE, ch);
}
```

RTOS: Introducción y primeros pasos

- Definición de Sistema de Tiempo Real y RTOS
- ¿Por qué utilizar un RTOS?
- 3 maneras de desarrollar un sistema embedido
- Introducción a Xinu (características)
  - Servicios que proporciona al desarrollador (sleep)
- Gestión de Tareas en Xinu (planificador/scheduler RTOS)
- Xinu en ejemplos: Tareas
  - Tareas concurrentes
  - Reuso de código
  - Diseño de tiempo real: prioridades fijas
- **Xinu en ejemplos: Sincronización y Comunicación de Tareas en Xinu**
  - Sincronización: semáforos (ej. productor/consumidor)
  - Región crítica: solución con mutex
  - Comunicación: pasaje de mensajes con mailbox
  - Buffers y colas: ports
- Recursos

» Xinu: ejemplo 4
Productor/Consumidor NO SINCRONIZADOS!

Descripción

1. **main()** crea dos tareas, un productor y un consumidor.
   - La tarea productora genera 2000 valores de n.
   - La tarea consumidora muestra por CONSOLA el valor de n producido.

2. El sistema falla en su objetivo.
   La E/S es miles de veces mas lenta que la CPU.

3. Para observar el output: *cutecom*

4. Observación: el codigo fuente de un proceso es de tipo void o process.

```
#include <xinu.h>

void    produce(void), consume(void);

int32   n = 0;          /* external variables are shared by all processes */

/*------------------------------------------------------------------------
 *  main  --  example of unsynchronized producer and consumer processes
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(consume, 256, 20, "cons", 0) );
        resume( create(produce, 256, 20, "prod", 0) );
}

/*------------------------------------------------------------------------
 * produce  --  increment n 2000 times and exit
 *------------------------------------------------------------------------
 */
void    produce(void)
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ )
                n++;
}

/*------------------------------------------------------------------------
 * consume  --  print n 2000 times and exit
 *------------------------------------------------------------------------
 */
void    consume(void)
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ )
                printf("The value of n is %d \n", n);
}
```

» Xinu: ejemplo 4
Sincronización entre tareas (Productor/Consumidor)

Descripción

1. **main()** crea dos tareas, un productor y un consumidor.
   - La tarea productora genera 2000 valores de n.
   - La tarea consumidora muestra por CONSOLA el valor de n producido.
2. Solución clásica al problema productor/consumidor, utilizando dos semáforos.
   Las tareas están ahora sincronizadas.
3. ***Xinu provee semáforos para sincronización***
4. Para observar el output: *cutecom*

```
#include <xinu.h>

void    prod2(int nargs, char *args[]);
void    cons2(int nargs, char *args[]);

int32   n = 0;                  /* n assigned an initial value of zero  */

/* main  --  producer and consumer processes synchronized with semaphores */
void    main(void)
{
        sid32   produced, consumed;

        consumed = semcreate(0);
        produced = semcreate(1);
        resume( create(cons2, 256, 20, "cons", 2, consumed, produced) );
        resume( create(prod2, 256, 20, "prod", 2, consumed, produced) );
}

/* prod2  --  increment n 2000 times, waiting for it to be consumed */
void    prod2(int nargs, char *args[])
{
        sid32 consumed = args[0];
        sid32 produced = args[1];
        int32   i;

        for( i=1 ; i<=2000 ; i++ ) {
                wait(consumed);
                n++;
                signal(produced);
        }
}

/* cons2  --  print n 2000 times, waiting for it to be produced */
void    cons2(int nargs, char *args[])
{
        sid32 consumed = args[0];
        sid32 produced = args[1];
        int32   i;

        for( i=1 ; i<=2000 ; i++ ) {
                wait(produced);
                printf("n is %d \n", n);
                signal(consumed);
        }
}
```

» Xinu: Semáforos y mutex. Región crítica. Ejemplo de uso.

Descripción

1. Un buffer circular es una excelente herramienta de comunicación entre dos procesos.
2. Una tarea produce elementos.
3. Una tarea remueve elementos y los procesa. Si no hay elementos en el buffer la tarea consumidora bloquea.
4. Los semáforos se utilizan para controlar los límites del buffer.
5. El mutex se utiliza para proteger la región crítica.
6. ***En Xinu, un mutex puede ser implementado utilizando un semáforo binario correctamente utilizado.***

   Se debe prestar especial atención a esta implementación.

```
/*
 * Las siguiente funciones pertenecen a una hipotética biblioteca de
 * gestión de un buffer circular
 */

/* Note that this simple queue is meant to be used with semaphores--
 * on its own, it doesn't know if it's empty or full!  Both cases
 * have head == tail.  One semaphore is enough to keep track of this.
 * Two semaphores are needed for control of both producer and consumer.
 */
#define N 10;     /* number of slots in buffer */
int buf[N];       /* shared buffer */
int head, tail;   /* for buffer used as queue */
sid32 produced, consumed, mutex;  /* two semaphores and one mutex */

void buffer_init(void)
{
  consumed = semcreate(N);   /* scount = number of empty spots in buffer */
  produced = semcreate(0);   /* scount = number of filled spots in buffer */
  mutex = semcreate(1);
  head = tail = 0;
}
```

```
void buffer_put(int n)
{

      wait(consumed);

      /* región crítica */
      wait(mutex);
      buf[head++] = n;          /* enqueue(n) */
      if (head >= N)            /* ... */
            head = 0;           /* ... */
      signal(mutex);

      signal(produced);

}

int buffer_get()
{

      wait(produced);

      /* region crítica */
      wait(mutex);
      n = buf[tail++];          /* n = dequeue() */
      if (tail >= N)            /* ... */
            tail = 0;           /* ... */
      signal(mutex);

      signal(consumed);

      return n;
}
```

RTOS: Introducción y primeros pasos

- Definición de Sistema de Tiempo Real y RTOS
- ¿Por qué utilizar un RTOS?
- 3 maneras de desarrollar un sistema embedido
- Introducción a Xinu (características)
  - Servicios que proporciona al desarrollador (sleep)
- Gestión de Tareas en Xinu (planificador/scheduler RTOS)
- Xinu en ejemplos: Tareas
  - Tareas concurrentes
  - Reuso de código
  - Diseño de tiempo real: prioridades fijas
- Xinu en ejemplos: Sincronización y Comunicación de Tareas en Xinu
  - Sincronización: semáforos (ej. productor/consumidor)
  - Región crítica: solución con mutex
  - Comunicación: pasaje de mensajes con mailbox
  - Buffers y colas: ports
- Recursos

» Recursos

WEB
1. https://xinu.cs.purdue.edu/
2. http://se.fi.uncoma.edu.ar/xinu-avr/

LIBRO
1. *Douglas Comer, Operating System Design - The Xinu Approach, Second Edition CRC Press, 2015. ISBN 9781498712439*