

Sistemas gobernados por tiempo (TDS) Implementación



Agenda.

- Demora Inicial
- Crear lista de tareas.
- Tareas multiciclo.
- Verificación Temporal.
- Balanceo de tiempo de ejecución.
- Sistemas multimodo
- Estrategia de datos compartidos
- Chequeos de las tareas.

Demora inicial de una tarea

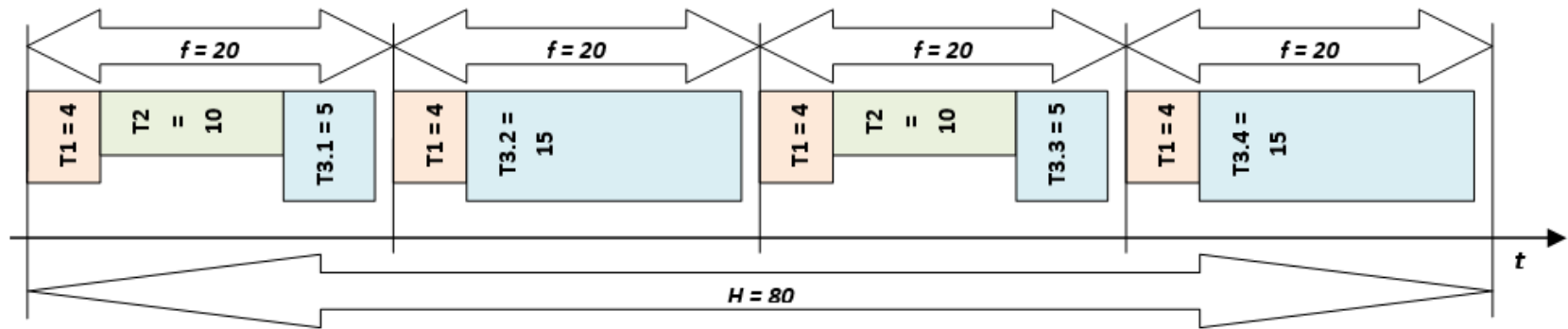
- Las tareas van a tener tres parámetros fundamentales que se van a utilizar para construir la lista de ticks del sistema:
 - El período de la tarea (T_i)
 - El peor tiempo de ejecución de la tarea ($WCET_i$)
 - La demora Inicial de la tarea ($Offset_i$)
- La demora inicial de cada una de las tareas juega un papel fundamental en la planificación, permite Modificar:
 - La precedencia de las tareas (que tarea va por delante de que otra).
 - El tiempo de respuesta a situaciones o requerimientos determinados.
 - El porcentaje de carga para cada tick de CPU

Demora Inicial. Planificación de tareas

| Tarea | T_i (ms) | C_i (ms) |
|-------|------------|------------|
| T1 | 20 | 4 |
| T2 | 40 | 10 |
| T3.1 | 80 | 5 |
| T3.2 | 80 | 15 |
| T3.3 | 80 | 5 |
| T3.4 | 80 | 15 |

- Dada esta lista de tareas. Determine para un $T_s = 20\text{ms}$:
 - ¿Qué tareas pueden compartir el mismo T_s ? ¿Cuáles no? ¿Por qué?
 - Proponga para cada tarea las demoras iniciales.

Demora Inicial. Ejemplo



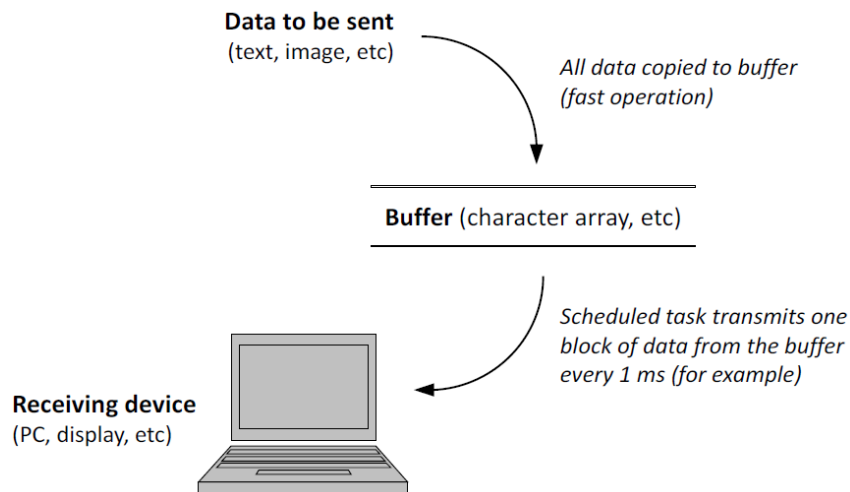
Creando la lista de tareas

- Una vez que tenemos las tareas definidas con su período y peor tiempo de ejecución (WCET), se arma la planificación como ya vimos.
 - Se verifican los tiempos de respuesta.
 - Y después? ***Hay que crear la lista de tareas***
1. Se ordenan las tareas por prioridad, si todas tienen (a priori) la misma prioridad se ejecutan primero las que tengan período más corto
 2. Se agregan las tareas al planificador
 3. En principio todas las tareas se elijen con ***demora inicial cero*** (offset cero).
 4. Se revisa la carga de CPU para cada tic del hiperperíodo (se elije una carga máxima de CPU, 80% por ejemplo.
 5. Sí no se logra se cambian las demoras iniciales.
 6. Se repite la prueba de planificabilidad
 7. ***Ya está la lista generada***

Tareas multicitelo

- En un sistema gobernado por tiempo es crucial que el WCET de cualquier tarea sea tan bajo como sea posible.
- Tareas con bajo WCET nos permiten planificaciones más simples y menos carga de CPU.
- Las tareas multicitelo son aquellas que involucran esperas, pero que ***no se desarrollarán como esperas activas.***
- Un ejemplo es la transmisión de un mensaje por línea serie con un baudrate bajo (9600-8-N-1).
- Un mensaje del tipo: "La temperatura es: 27.45 ° C" tardaría unos 27 ms en enviarse lo que sería inaceptable para la mayoría de los sistemas.
- Por lo que la solución al problema del mensaje serie es enviar un carácter en cada pasada por la tarea lo que baja el WCET a unos pocos µs.

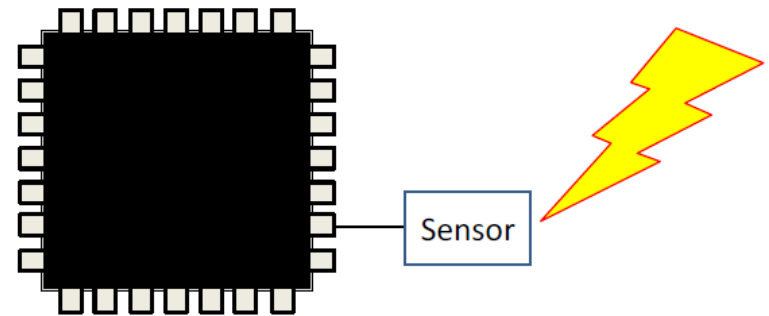
Tareas multicitelo



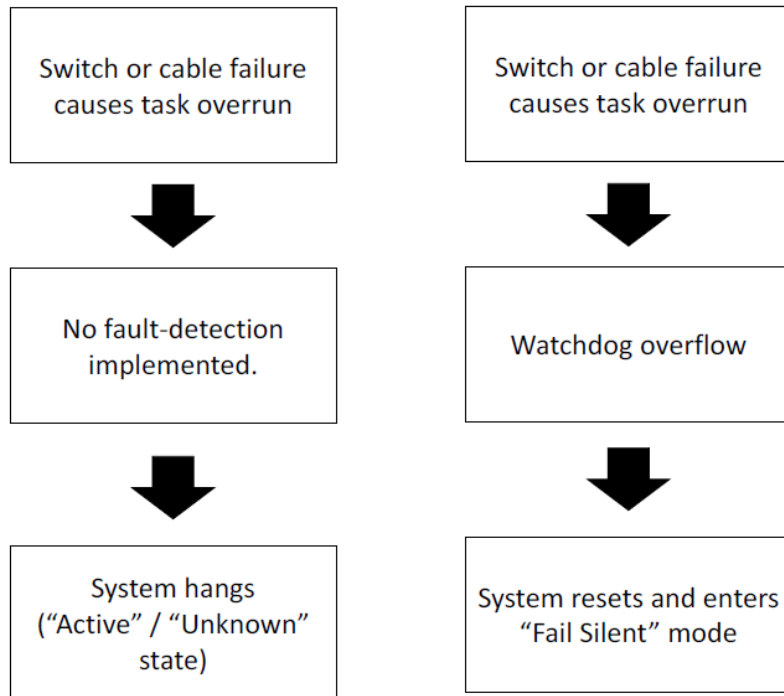
- Básicamente la estrategia es llenar un buffer con los datos a enviar y enviar un byte de ese buffer en todos los ciclos secundarios.
- La presunción es que los datos se van a generar de manera más lenta a la que los transmitimos

Chequeo de tiempos

- La principal diferencia entre un sistema gobernado por tiempo de otros sistemas **es la verificabilidad de los tiempos de ejecución de las tareas de manera individual o de la suma de todas ellas.** Por lo que es indispensable el chequeo de los tiempos del sistema.
- Un equipo puede fallar por un sensor que se rompa o no responda a su peor tiempo. Para ello se usan las estrategias de:
 - Watchdog
 - Chequeo de **tiempo de ejecución individual.**

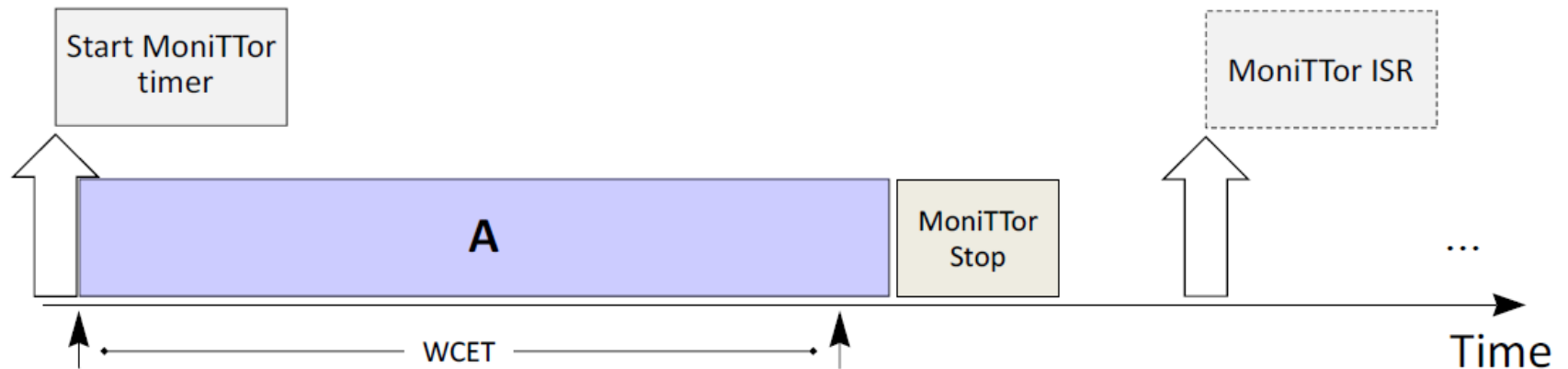


Temporizador Watchdog



- El Watchdog es un mecanismo de seguridad que consiste en un temporizador que irá continuamente decrementando un contador, inicialmente con un valor relativamente alto. Cuando este contador llegue a cero, se reiniciará el sistema, así que se debe diseñar una subrutina en el programa de manera que refresque o reinicie al perro guardián antes de que provoque el reset.
- Si el programa falla o se bloquea, al no actualizar el contador del perro guardián a su valor de inicio, éste llegará a decrementarse hasta cero y se reiniciará el sistema.

Chequeo de tiempo de ejecución



Tiempos de ejecución. Exceso de tiempo en una tarea

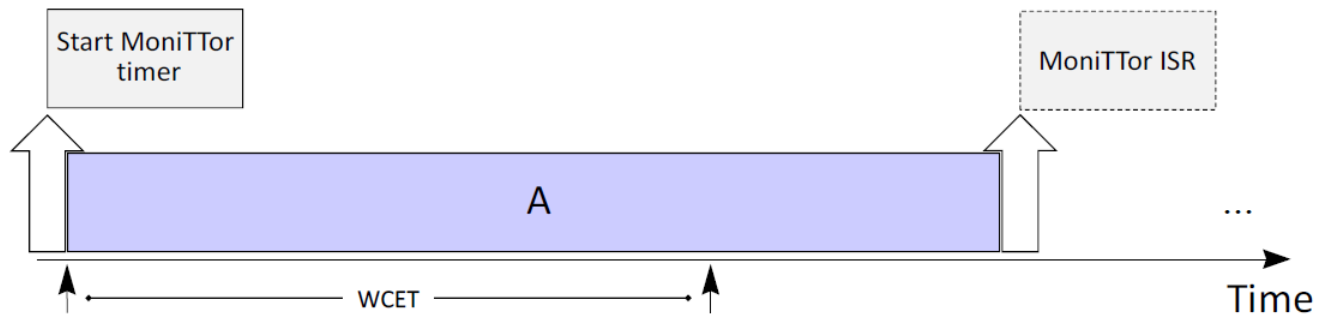
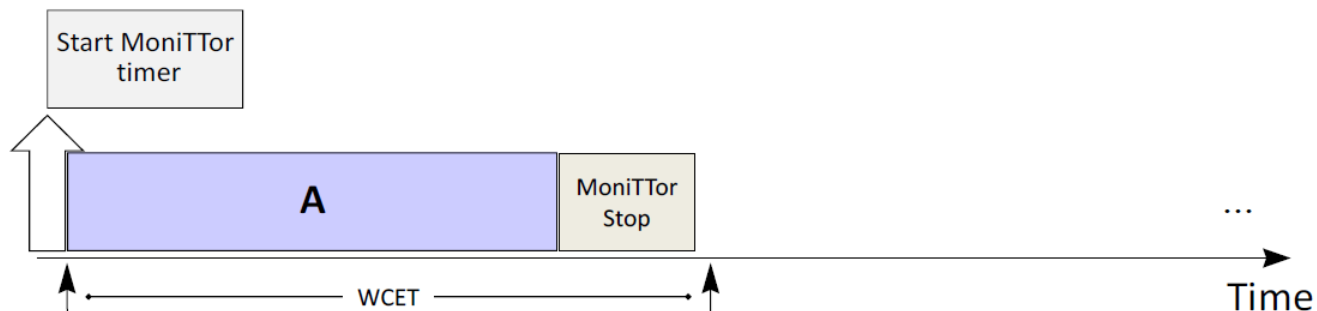
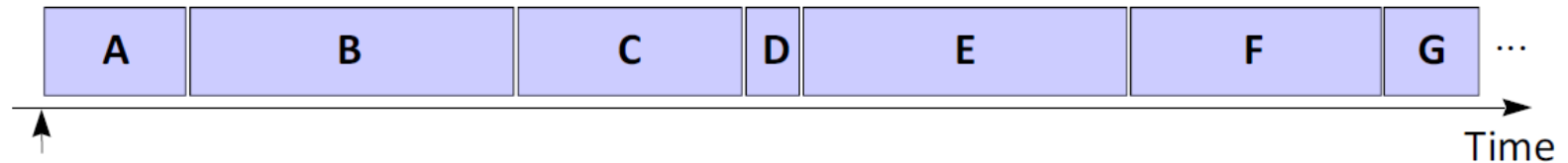


Figure 79: The operation of the MoniTtor when a task overruns.



Balanceo de tiempo de ejecución



- En esta lista de tareas que se ejecutan en un mismo tick secundario si la tarea A o B (por poner un ejemplo) tienen mucha diferencia entre su BCET y WCET todas las tareas siguientes van a correrse en el tiempo. Esto se lo conoce como ***jitter de tareas***.
- Si se quiere maximizar la previsibilidad temporal es necesario “ecualizar” el BCET y el WCET.

Código balanceado

```
void Task_A(void)
{
    /* Task_A has a known WCET of A milliseconds */
    /* Task_A is balanced */

    // Read inputs (KNOWN AND FIXED DURATION)

    // Perform calculations (KNOWN AND FIXED DURATION)

    /* Starting at t = A1 ms
       (measured from the start of the task),
       for a period of A2 ms, generate outputs */

    /* Task_A completes after (A1+A2) milliseconds */
}
```

Código desbalanceado

```
void PROCESS_DATA_Update(void)
{
    if (Data_set_1_ready_G)
    {
        Process_Data_Set_1(); // WCET = 0.6 ms
    }

    if (Data_set_2_ready_G)
    {
        Process_Data_Set_2(); // WCET = 0.2 ms
    }
}
```

Código desbalanceado (2)

```
void PROCESS_DATA_Update(void)
{
    if (Data_set_1_ready_G)
    {
        Process_Data_Set_1(); // WCET = 0.6 ms
    }
    else
    {
        Delay_microseconds(600);
    }

    if (Data_set_2_ready_G)
    {
        Process_Data_Set_2(); // WCET = 0.2 ms
    }
    else
    {
        Delay_microseconds(200);
    }
}
```

- El balanceo del código se da en este caso generando **demoras en los ciclos else de cada if.**
- También se puede ver en este ejemplo, **lo fundamental de tener bien caracterizado el WCET de cada parte de la tarea**

Demora de Sándwich

```
while(1)
{
    // Delay value in microseconds (1 second)
    SANDWICH_DELAY_T3_Start(1000000);

    HEARTBEAT_Update();

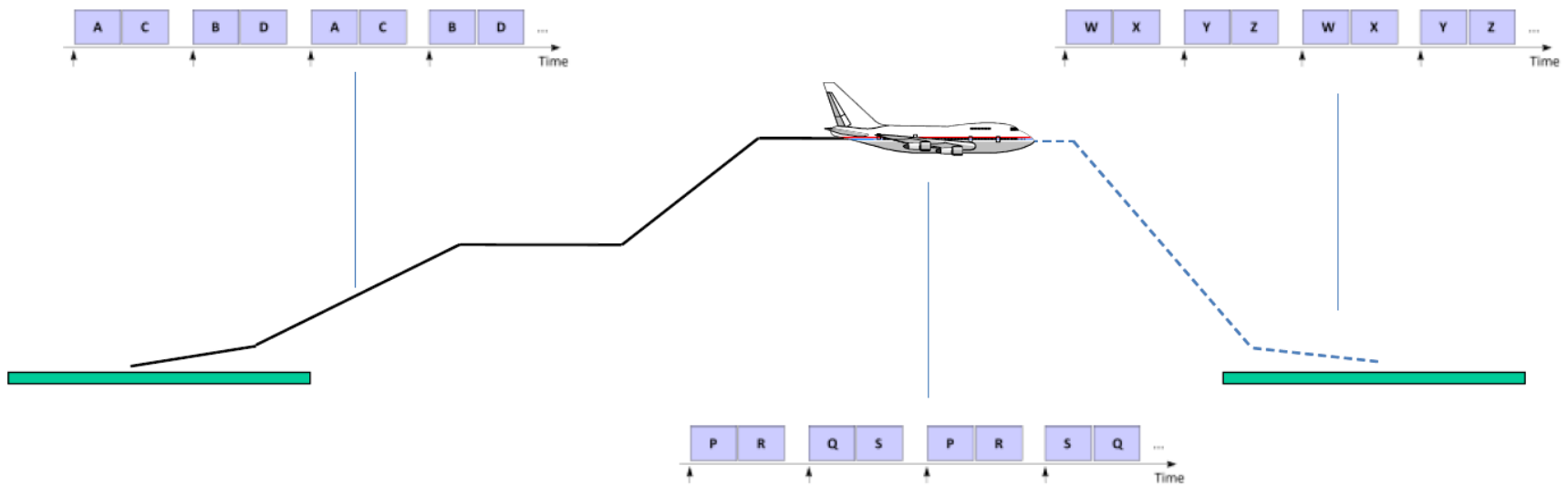
    SANDWICH_DELAY_T3_Wait();
}
```

- Otra forma de “ecualizar” la duración de las tareas es con las demoras tipo sándwich.
- Estas demoras funcionan de la siguiente forma:
 - La función de inicio de la demora pone a correr un timer con la duración total del tiempo que debe tardar la tarea.
 - Se ejecuta la función de la tarea.
 - La función sándwich, genera la diferencia entre el tiempo que tardó en ejecutarse la tarea y el tiempo total.

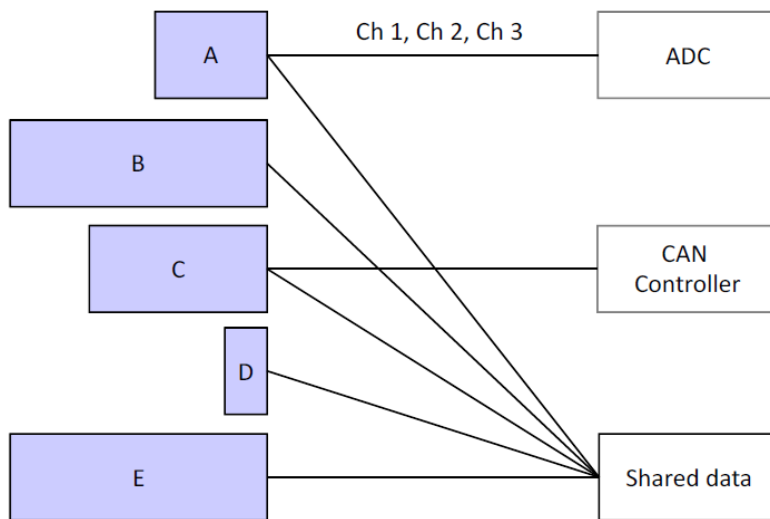
Sistemas multimodo

- Hasta ahora los sistemas gobernados por tiempo, para nosotros, manejan dos posibles modos de sistema:
 - **Normal.** Donde se está ejecutando la lista de tareas periódicamente y se respetan todas las condiciones del sistema.
 - **Falla.** El sistema violó los tiempos de funcionamiento y se pasa a un modo de **apagado ordenado** para indicar que no se cumplen las condiciones de funcionamiento. A este modo se va a llegar a través del **watchdog o una tarea excedida en su WCET**
- Existe otra posibilidad que es que un mismo sistema tenga varias listas de tareas (validadas y planificadas correctamente) y en función de condiciones del sistema **se cambie la lista de tareas en ejecución.**
- Normalmente este cambio va a requerir un **reset del sistema** para que el nuevo modo entre en funcionamiento.
- En particular para el LPC1769 existen los registros del RTC en NVRAM que se pueden utilizar como mecanismo para pasar parámetros en los cambios de modo

Sistemas multimodo



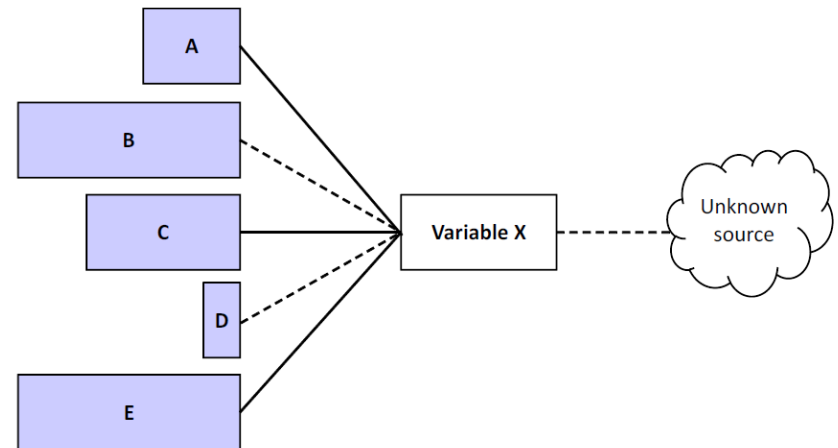
Estrategias de datos compartidos



- En los sistemas gobernados por tiempo los conflictos por compartir datos son muy raros ya que todas las tareas se ejecutan hasta completarse.
- Por lo que es recomendable que una única tarea maneje un único periférico y esta escriba los datos compartidos con otras tareas.

Datos compartidos

- La estrategia más simple para compartir datos en un sistema gobernado por tiempo es a través de ***variable globales***.
- ¿Qué pasa si una tarea se descontrola y altera los datos de estas variables?

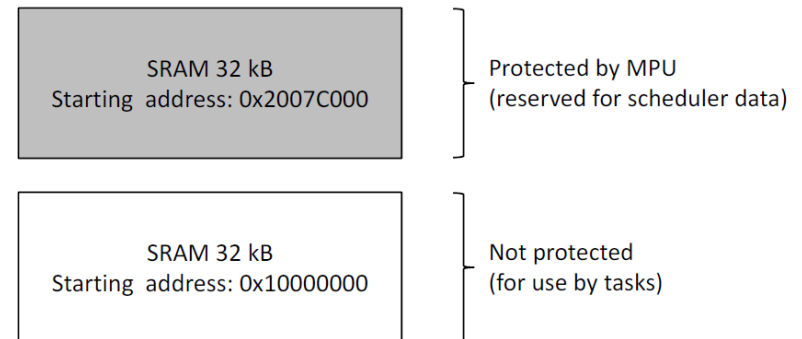


Datos compartidos

- Para reducir la posibilidad de que una tarea desbocada “rompa” los datos de otras es recomendable:
 - No proveer acceso directo a la variable compartida, sino a través de funciones como Leer() y Escribir()
 - Usar el esquema “Una tarea escribe N tareas leen”
 - En caso de ser posible tener una segunda copia de los datos que es la compartida con las otras tareas.
 - Tener una “copia invertida” de la variable global en memoria y chequearla antes de usarla

Datos del planificador

- El planificador también va a tener sus datos en memoria.
- Si la “tarea desbocada” corrompe estos datos, se corrompe todo el sistema.
- El planificador puede usar la estrategia de “la copia invertida” y en caso de falla llevar el sistema a falla o bien usar la **MPU del procesador**.
- La MPU (memory protection unit) es un periférico del core que permite cambiar los permisos de acceso a diferentes zonas de memoria del procesador.



Chequeos en las tareas

```
uint32_t Task_With_Contract(const uint32_t TASK_ID)
{
    // Check pre conditions (incl. shared data)


    ...

    // On-going checks

    ...

    // Check post conditions

    return Task_status;
}
```



Timing conditions
(WCET, BCET)

[Checked by scheduler]

Manos a la obra

- Con esta presentación se provee el código https://gitlab.frba.utn.edu.ar/jalarcon/led_planificador_tds_td2.git que implementa un pequeño sistema TDS de ejemplo con dos leds y un pulsador.
- Para este sistema se pide:
 - ☐ ¿Cuántas Tareas hay?
 - ☐ Mida los tiempos de cada tarea y actualice los valores de WCET en función de los valores medidos.
 - ☐ Verifique si el sistema es planificable o no.
 - ☐ Estime la carga de CPU en casos de que sea planificable
 - ☐ Determine los modos de sistema.
 - ☐ Determine que mecanismos de protección temporal se usan.
 - ☐ Determine que tareas están balanceadas y cuales no. Proponga una mejora.
 - ☐ Determine cómo se comparten los datos entre tareas. ¿Qué variables se usan? ¿Hay barreras para evitar errores? Proponga mejoras.



Bibliografía.

- Sistemas Operativos. Diseño e Implementación. Andrew S. Tanenbaum.
- Sistemas de Tiempo Real y Lenguajes de Programación. Alan Burns, Andy Wellings. Tercera Edición. Addison Wesley.