



Tecnicatura Universitaria
en Programación

ARQUITECTURA Y SISTEMAS OPERATIVOS

Unidad Temática N°4:
Sistemas operativos

Material Teórico
1° Año – 2° Cuatrimestre



Índice

MULTIPROGRAMACION, MULTITAREA Y SISTEMAS OPERATIVOS EN TIEMPO REAL 3

Multiprogramación.....	3
Multiprogramación y sistemas embebidos.....	4
Sistemas Operativos en Tiempo Real (Real Time Operating System).....	7
Conceptos básicos	7

¿Qué es el kernel? 9

Caso particular de Linux.....	9
¿Cuál es la función del kernel?	9

Sistemas frente / fondo 10

Reentrada 11

Recursos 12

Multitarea y cambios de contexto 12

Tareas e interrupciones 14

Planificación Apropiativa vs. Cooperativa 15

Planificación Cooperativa 16

Multitarea Simple 18

Multitarea basada en prioridades 18

Estados de las tareas 18

Multitarea basada en eventos 20

Eventos y comunicación entre tareas 21

Inversión de Prioridades 21

Semáforos	21
Uso de los semáforos	21
Sistema operativo en tiempo real (SOTR)	22
Resumen	22
BIBLIOGRAFÍA	23

MULTIPROGRAMACION, MULTITAREA Y SISTEMAS OPERATIVOS EN TIEMPO REAL

Multiprogramación

Multiprogramación es un término que significa que muchos programas que no están relacionados unos con otros pueden residir en la memoria de una computadora y tomar turnos usando la unidad central de procesamiento, cualquiera que haya usado Windows®, Unix o Linux ha experimentado un entorno de multiprogramación porque estos sistemas pueden ejecutar un largo número de programas de usuario aparentemente de manera simultánea en una sola unidad central de procesamiento.

La sección que permite la multiprogramación en un sistema operativo se denomina **Kernel**, éste software consiste en un número de procedimientos que llevan a cabo funciones como crear tareas, decidir cuál tarea correrá en un momento dado, proveyendo un mecanismo para cambiar al procesador de una tarea a otra y provee formas primitivas de acceso controlado a recursos compartidos.

Un programa lo constituye una secuencia de instrucciones que cuando se ejecutan realizan alguna actividad. Las tareas a su vez la constituyen una serie de instrucciones (ejemplo, por ejecución de un programa o programas). Esta acción puede proporcionar una función de sistema o una función de aplicación, que un sistema operativo pueda correr concurrentemente.

La palabra concurrente, se refiere a la ejecución de varios procesos al mismo tiempo y con el mismo nivel de importancia. Este concepto importante, actualmente está resuelto con la tecnología denominada CPLDs y FPGAs.

A primera vista pareciera que tarea y programa son lo mismo. Sin embargo una tarea es un conjunto de un programa. Esto es mejor demostrado con un ejemplo. Considere un programa que implemente una cola FIFO (primera que entras es la primera que sale). Este programa podría ser compartido por otras tareas en donde parte de sus acciones requiera utilizar una cola FIFO.

Esto hace que una tarea aparezca como un módulo de programa el cual usa otro módulo de programa, sin embargo hay que recordar que las activaciones de módulos de programas en un entorno que no sea de sistema operativo no se podrán ejecutar concurrentemente. Por lo tanto el concepto de tarea está íntimamente atado a un entorno de sistema operativo. De hecho cuando una tarea es creada el sistema operativo crea un entorno de ejecución el cual está separado de los entornos de ejecución de otras tareas (por ejemplo, áreas de datos separadas, pila separada para

llamada de procedimientos). Muchas tareas pueden compartir el mismo código pero el código es ejecutado en un entorno de ejecución diferente, mientras cada tarea sepa que se está ejecutando en su propio procesador.

La diferencia entre una tarea y un programa puede ser vista en el siguiente ejemplo. Considere en un sistema de computadora que permita editar archivos ASCII. Generalmente en sistemas de computadora que permiten esto el código del editor está diseñado para ser re-entrante – eso es que múltiples hilos de ejecución puedan ejecutar el código al mismo tiempo. Código re-entrante a menudo tiene la misma estructura del código recursivo. El código recursivo permite a los procedimientos escritos en lenguajes tales como Pascal o C llamarse a sí mismos, esto implica que cada reencarnación del procedimiento tenga su propio juego de variables locales (cualquier variable global será manipulada por cada reencarnación. Esta propiedad es requerida por el código el cual está siendo usado por múltiples tareas, de no ser así una tarea corromperá la data que esté siendo manipulada por otra tarea. De vuelta al ejemplo planteado. En el caso del editor, cada usuario correrá una copia del editor. La tarea o la acción requerida por el usuario es la edición del archivo. Por lo tanto cada usuario está corriendo una tarea. Sin embargo cada tarea está ejecutando actualmente las mismas instrucciones, de hecho está ejecutando físicamente las mismas instrucciones (o programa). Ejemplo: existe físicamente una copia del editor en memoria. Sin embargo la data manejada por estas instrucciones es diferente para cada tarea, y las instrucciones en efecto están siendo ejecutadas en diferentes procesadores.

Multiprogramación y sistemas embebidos

La multiprogramación no se aplica solamente en sistemas de cómputo completos (PC, MainFrames, etc.). La primera computadora personal al principio de los años 80 tenía 40KB de ROM, 256 o 512KB de RAM, y opcionalmente un disco duro de 5 o 10 MB de capacidad. A mediados de los años 90 una computadora personal moderna tenía un poco más de ROM, 32 MB de RAM y un disco duro con 2 o 4GB de capacidad. Los discos flexibles “Floppy” con 360KB o 720KB de capacidad, los cuales fueron los medios standards para la distribución de paquetes de software y para respaldos, han sido reemplazados por unidades CD-ROM y dispositivos de almacenamiento magnético en cinta con capacidades por encima de 500MB.

Obviamente, la capacidad se ha doblado aproximadamente cada 2 años, y no hay indicador de que esta tendencia cambiará. Así que ¿Por qué preocuparse por los requerimientos de memoria?

Una computadora personal es un sistema abierto que puede ser extendido tanto en términos de memoria y periféricos. Por un corto tiempo, una computadora personal

puede estar al día con los desarrollos tecnológicos añadiendo memoria y periféricos hasta que últimamente dejen de ser actualizados. De todas formas una computadora personal puede vivir por décadas; pero su tiempo de vida real a menudo está determinado por la creciente demanda de memoria de los sistemas operativos y aplicaciones más que por la vida útil de su hardware. Así que para extender la vida útil de una computadora personal lo más posible y aún reducir los costos, su configuración debe ser debidamente planificada.

Para un sistema embebido, en contraste, los requerimientos de memoria son conocidos por adelantado; así que los costos pueden ser reducidos usando solo la memoria que se necesite. A diferencia de las computadoras personales, donde la memoria ROM es solo usada para el sistema de carga, el tamaño de la ROM juega un rol mayor en los requerimientos de memoria de sistemas embebidos, y eso es porque la ROM es usada como memoria de programa. Para las ROM, varios tipos de memoria están disponibles, y sus precios difieren de manera dramática EEPROM es la mas costosa, seguida por RAM, EPROM estáticas, RAM dinámica, discos duros, discos flexibles, CD-ROMs, y cintas magnéticas. La solución mas económica es combinar discos duros (el cual provee no volatilidad) y RAM dinámica (que provee tiempos de acceso rápidos).

Generalmente, la tecnología de memoria usada para una aplicación embebida está determinada por la misma aplicación. Por ejemplo: Para una impresora láser, la RAM será dinámica, y la memoria de programa será EEPROM, EPROM o RAM cargada desde un disco duro. Para un teléfono móvil, EEPROM y RAM estática serían preferiblemente usadas.

Una tecnología que es particularmente interesante para sistemas embebidos es la memoria dentro del chip. Comparativamente muchas ROM dentro del chip han estado disponibles por años, pero su falta de flexibilidad limitó su uso a sistemas producidos en grandes cantidades. La próxima generación de microcontroladores fueron EPROMs dentro de un chip, los cuales eran aceptables incluso para cantidades más pequeñas. Los microcontroladores recientes proveen de memoria EEPROM y RAM estática. La serie de Motorola 68HC9xx, por ejemplo, ofrece EEPROM dentro del chip de 32KB a 100KB y un RAM estática de 1KB a 4KB.

Con el regreso del microprocesador Z80, otra solución interesante se ha vuelto interesante. Aunque tiene dos décadas de antigüedad, este chip parece sobrepasar a sus sucesores. La estructura es tan simple que puede ser integrada en FPGAs (Field Programmable Logic Arrays – Matrices Lógicas Programables). Con ésta técnica microcontroladores enteros pueden ser diseñados para que quepan en un chip,

proveyendo exactamente las funciones requeridas por una aplicación. Como muchos otros microcontroladores el Z80 provee un total de espacio de memoria de 64KB.

Aunque el tamaño de memoria dentro de los chips probablemente crecerá en el futuro, las capacidades disponibles hoy sugieren que un sistema operativo para un sistema embebido debería ser menor a 32KB, dejando suficiente espacio para la aplicación.

Independientemente de los desarrollos tecnológicos aplicados a microcontroladores hoy en día como periféricos integrados u otras tecnologías en memoria, por ejemplo, la tecnología FLASH.

El concepto de multiprogramación nació desde la observación que las computadoras gastaban mucho de su tiempo esperando a los dispositivos periféricos ya sea para tomar o guardar data. Al principio de la computación los dispositivos periféricos eran bastante lentos (cintas de papel, cintas magnéticas y teletipos) unos 10 caracteres por segundo era una velocidad muy común. A pesar de que los dispositivos periféricos actualmente han incrementado su velocidad considerablemente también lo han hecho las computadoras, por lo tanto incluso hoy la diferencia de velocidad relativa entre periféricos es casi la misma y por tanto el mismo problema sigue siendo de relevancia. Pero ¿Cuál es problema en esperar a que una acción de Entrada/Salida se lleve a cabo?; la respuesta: Una es simplemente que no se está aprovechando lo mejor del hardware de computación. En una situación de un único usuario no es de tanta importancia (aunque los sistemas multiprogramación son muy populares en esta situación debido a otros beneficios)

A finales de los años 60 y los principios de los años 70 las computadoras que hoy en día tendrían un modesto poder en comparación con los estándares actuales, eran muy costosas.

Por lo tanto era de mucha importancia sacar el máximo partido de aquellas maquinas; esperar por una operación de Entrada/Salida era tiempo improductivo. A partir de entonces empezó a aparecer sistemas los cuales podían manejar el trabajo de alguien mientras otra operación estaba esperando una Entrada/Salida en otro recurso que no estaba disponible cuando se hizo la petición del mismo. El software que permitió la ordenada planificación de los trabajos es el entorno conocido como sistema operativo multiprogramación. Tal software también provee abstracción y control de acceso. Colectivamente la provisión de estos dos aspectos es lo que se deberá llamar un sistema operativo. Cabe destacar que MSDOS ® no provee un soporte adecuado para multiprogramación.

Sistemas Operativos en Tiempo Real (Real Time Operating System)

Conceptos básicos

Una tarea es una secuencia de instrucciones, algunas veces hechas repetidamente, para realizar una acción (ejemplo, leer un teclado, mostrar un mensaje en una pantalla LCD, hacer parpadear a un LED o generar una forma de onda). En otras palabras, es usualmente un programa dentro de uno más grande. Cuando se corre en un procesador relativamente simple (ejemplo, Z80, 68HC11, PIC), una tarea puede tener todos los recursos del sistema para sí sin importar cuantas tareas se usen en la aplicación.

Los recursos disponible son la memoria RAM,EEPROM, contadores, conversores ADC y DAC, puertos I/O, etc...

Una interrupción es un evento interno o externo en el hardware que causa que la ejecución del programa sea suspendida. Las interrupciones deben estar habilitadas para que puedan ocurrir. Cuando esto ocurre, el procesador se dirige a una **Rutina de Servicio de Interrupción** (*ISR - Interrupt Service Routine*), la cual corre en su totalidad. Entonces la ejecución del programa retoma desde el punto en que fue interrumpido. Debido a su capacidad para suspender la ejecución de programas, se dice que las interrupciones corren al frente (**foreground**) y que el resto del programa corre al fondo (**background**).

La prioridad de una tarea sugiere la importancia de la tarea relativamente de otras tareas. Esta puede ser fija o variable, única o compartida con otras tareas.

Un **cambio de tarea** ocurre cuando una tarea es suspendida y otra tarea comienza o continúa su ejecución. También es llamado **cambio de contexto**, esto se debe a que el contexto de la tarea (generalmente el contenido completo de la pila y el valor de los registros) es almacenado para ser utilizado cuando se reanude la ejecución de la tarea.

Una tarea debe voluntariamente liberar el control del procesador antes de que otra tarea pueda correr. Depende del programador estructurar la tarea para que esto ocurra. Si una tarea que está en ejecución falla en cooperar, entonces la aplicación también fallará.

El cambio de contextos tanto apropiativo como cooperativo es manejado por el kernel.

El software que conforma el kernel administra el cambio de tareas (también llamado **planificación**) y la comunicación entre tareas. Un **kernel** generalmente asegura que la tarea elegible con la prioridad más alta es la tarea que se está

ejecutando (en la planificación apropiativa) o la próxima a ejecutarse (planificación cooperativa). Los **kernels** son escritos para ser lo más **pequeño y rápido** posible, esto es para garantizar alto rendimiento sobre el programa de aplicación.

Un **retardo** es una cantidad de tiempo (a menudo especificado en milisegundos) en la cual la ejecución de una tarea puede ser suspendida. Mientras está suspendida una tarea debería usar el menor recurso de procesador como sea posible para maximizar el desempeño de toda la aplicación, la cual es común que incluya otras tareas que no están suspendidas concurrentemente. Una vez que el retardo ha concluido la tarea reanuda su ejecución. El programador especifica cuánto tiempo durará el retardo y que tan a menudo sucede.

Un **evento** es la ocurrencia de algo (por ejemplo, una tecla fue presionada, ocurrió un error, o una respuesta que se esperaba nunca sucedió) que una tarea puede esperar.

También, casi cualquier parte de un programa puede señalar la ocurrencia de un evento, por tanto dejando saber a los demás que el evento ocurrió.

Comunicación entre tareas son formas ordenadas de pasar información de una tarea a otra siguiendo conceptos de programación bien definidos. **Semáforos, mensajes, cola de mensajes y bandera de eventos** puede ser usados para pasar información de una manera u otra entre tareas, en algunos caso a la rutina de servicio de interrupción.

El **tiempo fuera**, es una cantidad de tiempo (usualmente en milisegundos) que una tarea puede esperar por un evento. El tiempo fuera es opcional, una tarea puede espera por un evento de manera indefinida. Si una tarea precisa un tiempo fuera cuando está esperando un evento y dicho evento no ocurre, decimos que el tiempo fuera ha ocurrido y un tratamiento especial es invocado.

El **estado** de una tarea está actualmente haciendo. Las tareas cambian de un estado a otro a través de reglas bien definidas. Los estados comunes para las tareas son **lista / elegible, corriendo, demorada, en espera, detenida / suspendida, destruida**.

El **temporizador** es otra pieza del software que mantiene la pista del tiempo transcurrido y/o del tiempo real para las demoras, tiempos fuera y otros servicios relacionados con el temporizador. El temporizador es tan preciso como el reloj que incorpore el sistema.

Se dice que un sistema está **ocioso** cuando no hay tareas que correr.

El **sistema operativo** contiene al kernel, el temporizador, y el software restante (llamado **servicios**) para manejar tareas y eventos (por ejemplo, creación de tareas, señalización de eventos, habilitación de temporizadores, entre otros). **Los sistemas operativos en tiempo real** son los elegidos al momento cuando ciertas operaciones son críticas y deben completarse correctamente dentro de una cierta cantidad de tiempo.

¿Qué es el kernel?

Caso particular de Linux

El kernel de Linux® es el elemento principal de los sistemas operativos (SO) Linux, y es la interfaz fundamental entre el hardware de una computadora y sus procesos. Los comunica entre sí y gestiona los recursos de la manera más eficiente posible.

Se llama kernel porque se encuentra dentro del sistema operativo, al igual que las semillas de las frutas con cáscara dura, y controla todas las funciones principales del hardware, ya sea un teléfono, una computadora portátil, un servidor o cualquier otro tipo de equipo.

¿Cuál es la función del kernel?

El kernel cumple cuatro tareas:

Gestión de la memoria: supervisa cuánta memoria se utiliza para almacenar qué tipo de elementos, así como el lugar en que los guarda.

Gestión de los procesos: determina qué procesos pueden usar la unidad central de procesamiento (CPU), cuándo y durante cuánto tiempo.

Controladores de dispositivos: actúa como mediador o intérprete entre el hardware y los procesos.

Seguridad y llamadas al sistema: recibe solicitudes de servicio por parte de los procesos.

Si se implementa de **forma correcta**, es **invisible para el usuario** y funciona en su propio mundo pequeño, conocido como el espacio del kernel, donde asigna la memoria y supervisa el lugar en el que se almacena cada elemento. Lo que está a la vista del usuario, como los exploradores y los archivos, se conoce como el espacio del usuario. Estas aplicaciones interactúan con el kernel a través de una interfaz de llamadas al sistema (SCI).

Piénselo de esta manera. El kernel es el asistente personal atareado de un poderoso ejecutivo (el hardware). Debe transmitir los mensajes y las solicitudes (los

procesos) de los empleados y el público (los usuarios) al ejecutivo, recordar qué elementos se almacenan y dónde (la memoria) y determinar quién puede acceder al ejecutivo en un momento dado y durante cuánto tiempo.

Sistemas frente / fondo

La estructura de programa más simple es una de bucle principal (algunas veces llamada Súper bucle), las llamadas a funciones se hace de manera secuencial. Esto es porque la ejecución de un programa puede cambiar desde el bucle principal a la rutina de servicio de interrupciones y regresar al bucle principal, se dice que el bucle principal corre en el fondo, donde el servicio a las interrupciones corre en el frente. Ésta es la clase de programación con la que se encuentran muchos principiantes mientras aprenden a programar simples sistemas.

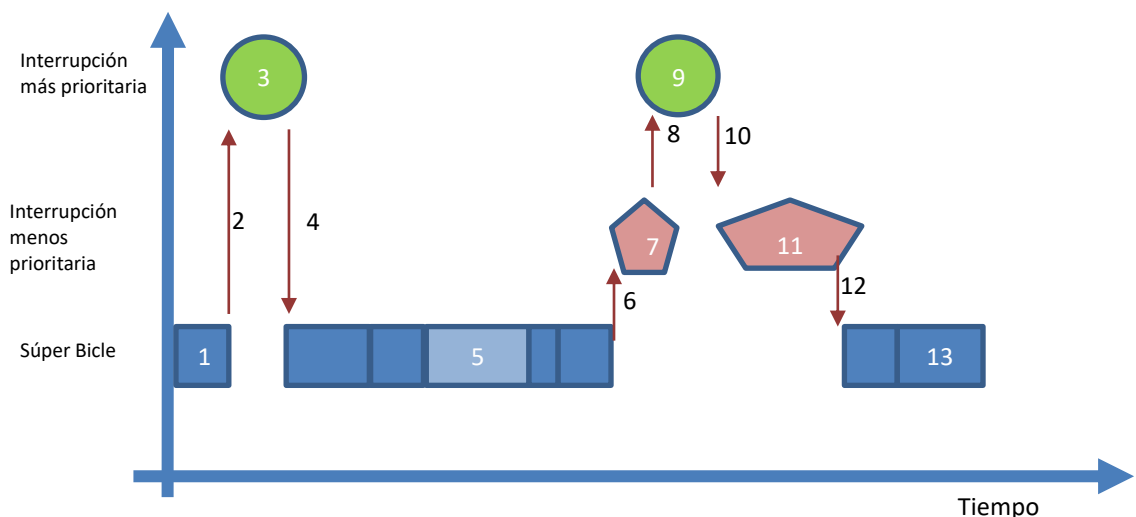


Figura 1 – Diagrama témpora de las tareas - : Elaboración propia

En la figura 1 se observa un grupo de funciones repetidas una y otra vez [1, 5, 13] en el bucle principal. Las interrupciones pueden ocurrir en cualquier momento, inclusive en distintos niveles. Cuando una interrupción ocurre (Interrupción de alta prioridad en [2] y [8], interrupción de baja prioridad en [6]), el procesamiento en la función es detenida hasta que finalice la interrupción, donde el programa retorna al bucle principal o a la rutina de servicio de interrupción interrumpida previamente. El bucle principal es ejecutado de una manera estrictamente serial, todo con la misma prioridad, sin cambios en la forma en que la función deba ejecutarse. Las rutinas de servicio de interrupción deben usarse para responder rápidamente a eventos externos, y pueden ser priorizadas si se dispone de múltiples niveles de interrupción.

Los sistemas frente / fondo son relativamente simples desde el punto de la programación, siempre y cuando haya poca interacción entre las funciones y el bucle principal, y entre éste y las rutinas de servicio de interrupción.

Pero tiene muchas desventajas: La temporización de los bucles se ve afectada por cualquier cambio en el bucle mismo y/o el código en las rutinas de servicio de interrupción. Inclusive, la respuesta del sistema a las entradas es pobre porque la información hecha disponible por una rutina de servicio de interrupción para una función en el bucle no puede ser procesada por la función hasta que sea su turno de ejecutarse. Ésta naturaleza secuencial rígida de la ejecución del programa en el súper bucle brinda muy poca flexibilidad al programador, y complica operaciones que son críticas de tiempo. Las máquinas de estado pueden ser usadas para solventar este problema parcialmente. A medida que la aplicación crece, la temporización del bucle se vuelve impredecible, y una variedad de factores de complicación van apareciendo.

Reentrada

Un factor es la reentrada. Una función reentrante puede ser usada simultáneamente en una o más partes de la aplicación sin corromper la data. Si la función no está escrita para ser reentrante, llamadas simultáneas pueden corromper la data interna de la función, con resultados impredecibles en la aplicación. Por ejemplo, si una aplicación tiene una función `printf()` y ésta es llamada tanto por el código del bucle principal (el fondo) e inclusive desde dentro de la rutina de servicio de interrupción (el frente), allí hay una excelente probabilidad que cada vez en un momento la salida resultante de una llamada a:

`printf` ("Estamos en el bucle principal.\n");

desde el interior del bucle principal y una llamada a:

`printf` ("Ahora estamos en la rutina de servicio de interrupción.\n");

Desde el interior de la rutina de servicio de interrupción, el resultado de la función sería:

`Estamos en el bu`
`Ahora estamos en la rutina de servicio de interrupción.`

Esto es claramente un error, el cual ha pasado desde la primera instancia de `printf()` (llamada desde el bucle principal) apenas llegó a imprimir los primeros 16 caracteres ("Estamos en el bu") de su argumento de cadena antes de ser interrumpida. La rutina de servicio de interrupción también incluía una llamada a `printf()`, la cual reiniciaba sus variables locales y tenía éxito en imprimir la cadena entera de 54 caracteres ("Ahora estamos ... interrupción."). Después de la rutina de servicio de interrupción el `printf()` del bucle principal continúa su ejecución desde el punto donde fue interrumpido, pero sus variables internas reflejaron haber escrito exitosamente hasta el final de un

argumento de cadena, y aparentemente ninguna otra salida era necesaria, así que sencillamente retornó y el bucle principal continuó su ejecución.

Nota: Llamar a funciones no reentrantes como si lo fueran raras veces arrojan un comportamiento tan benigno como el mostrado en el ejemplo.

Varias técnicas pueden ser aplicadas para evitar el problema de una función printf() no reentrante. Una de ellas es deshabilitar las interrupciones antes de llamar a una función no reentrante y habilitarlas al finalizar la función. Otra es reescribir printf() para que use solo variables locales (mantenida dentro de la pila de la función). La pila juega un importante papel en las funciones reentrantes.

Recursos

Un recurso es cualquier cosa dentro de un programa que puede ser usado por otras partes del programa. Un recurso puede ser un registro, una variable o una estructura de datos, o puede ser algo físico como una pantalla LCD o un zumbador. Un recurso compartido es un recurso que puede ser usado por más de una parte de un programa. Si dos partes separadas de un programa se están conteniendo el mismo recurso, se necesitará manejar la situación a través de la exclusión mutua. Cualquier parte de un programa que quiera usar el recurso debe obtener acceso exclusivo a él para evitar corromperlo.

Multitarea y cambios de contexto

Muchas ventajas pueden ser alcanzadas dividiendo una aplicación frente / fondo con múltiples e independientes tareas, para poder hacer multitarea, de tal manera que todas las tareas parezcan correr concurrentemente, algún mecanismo debe existir para pasar el control del procesador y sus recursos de una tarea a otra. Éste es el trabajo del planificador, parte del kernel que (entre otras labores) suspende una tarea para permitirle la ejecución a otra cuando ciertas condiciones se cumplen. Hace esto **almacenando el contador de programa** para una tarea y restaurándolo para otra. Mientras más rápido sea el planificador mejor el desempeño de toda la aplicación, ya que el tiempo ocupado mientras se cambia de tarea es tiempo gastado sin tareas corriendo.

Un cambio de contexto debe parecer transparente para la misma tarea. La “visión del mundo” de la tarea antes del cambio de contexto que la suspende y después del cambio de contexto que la continua debe ser el mismo. De esta manera la tarea “A” puede ser interrumpida en cualquier momento para permitir al planificador correr una tarea de mayor prioridad, la tarea “B”. Una vez finalizada la tarea “B”, la tarea “A” continúa su ejecución.

El único efecto del cambio de contexto para la tarea “A” es que fue suspendida por un largo lapso de tiempo potencial como resultado del cambio de contexto mismo. Aunque las tareas que realizan operaciones críticas en tiempo deben prevenir los cambios de contexto durante esos períodos críticos.

Desde el punto de vista de una tarea, un cambio de contexto puede salirse de los lineamientos normales, en el sentido de que el cambio de contexto puede ser forzado por razones externas a la tarea, o puede ser intencional debido al deseo del programador de suspender temporalmente la tarea para realizar otros procesos.

La mayoría de los procesadores soportan pilas de propósito general y tienen múltiples registros. Con sólo restaurar el contador de programa no bastará para garantizar la continuidad de ejecución de una tarea. Esto es debido a que la pila y los valores de los registros de la tarea al momento de realizar el cambio de contexto.

Un cambio de contexto almacena todo el contexto de la tarea (por ejemplo, contador de programa, registros, contenido de la pila). La mayoría de los procesadores requieren que la memoria sea asignada para poder llevar a cabo el cambio de contexto.

Tareas e interrupciones

Tal cual es el caso con los sistemas frente / fondo, los sistemas multitarea a menudo hacen uso extensivo de las interrupciones. Las tareas deben ser protegidas de los efectos de las interrupciones, las rutinas de servicio de interrupción deben ser lo más rápidas posible, y las interrupciones debe estar habilitadas la mayor parte del tiempo. Las interrupciones y las tareas coexisten simultáneamente – una interrupción puede ocurrir a la mitad de una tarea.

Se debe minimizar la deshabilitación de las interrupciones mientras corren las tareas, aún las interrupciones tendrán que ser controladas para evitar conflictos entre tareas e interrupciones cuando ambas accedan a recursos compartidos.

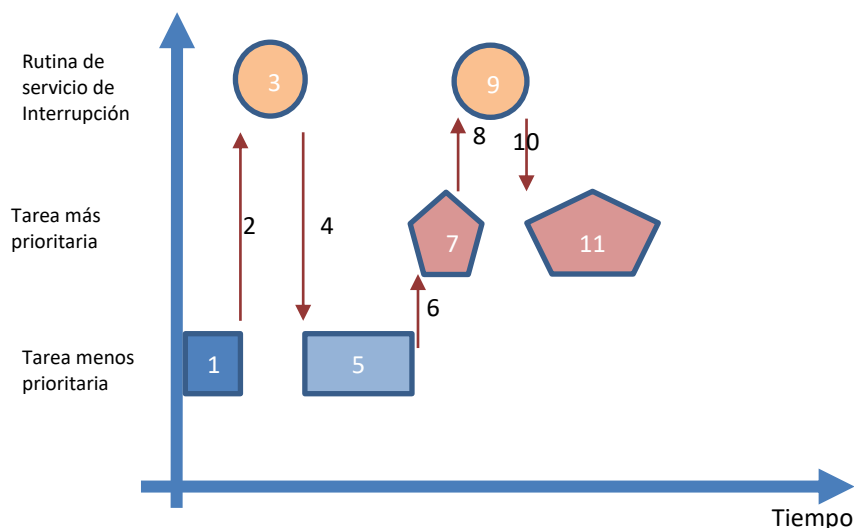


Figura 2 Las interrupciones pueden ocurrir mientras las tareas corren. – Elaboración propia

En la figura 2 una tarea de baja prioridad está corriendo [1] cuando ocurre una interrupción [2]. En éste ejemplo las interrupciones están siempre habilitadas. La interrupción [3] corre hasta completarse [4], donde la tarea de baja prioridad [5] continúa su ejecución. Un cambio de contexto ocurre [6] y la tarea de alta prioridad [7] comienza su ejecución. El cambio de contexto es manejado por el planificador (el cual no se muestra en la gráfica). La tarea de alta prioridad también es interrumpida [8-10] antes de continuar [11].

Latencia de interrupción se define como la **máxima** cantidad de **tiempo** en que están **deshabilitadas las interrupciones**, más el tiempo que toma ejecutar la primera instrucción de la rutina de servicio de interrupción. En otras palabras, es el

retardo en el peor de los casos entre la ocurrencia de una interrupción y cuando la rutina de servicio de interrupción correspondiente comienza a ejecutarse.

Planificación Apropiativa vs. Cooperativa

Existen dos tipos de planificadores: el apropiativo y el cooperativo. Un planificador apropiativo puede causar a que la tarea actual (por ejemplo, la tarea que está corriendo actualmente) sea suspendida para reanudar la ejecución de otra. La apropiación ocurre cuando una tarea de mayor prioridad que la que está corriendo actualmente se vuelve elegible para correr. Como esto puede ocurrir en cualquier momento, la apropiación requiere el uso de interrupciones y el manejo de la pila para garantizar un correcto cambio de contexto. Deshabilitando la apropiación temporalmente los programadores pueden prevenir disruptions en sus programas durante secciones críticas de código.

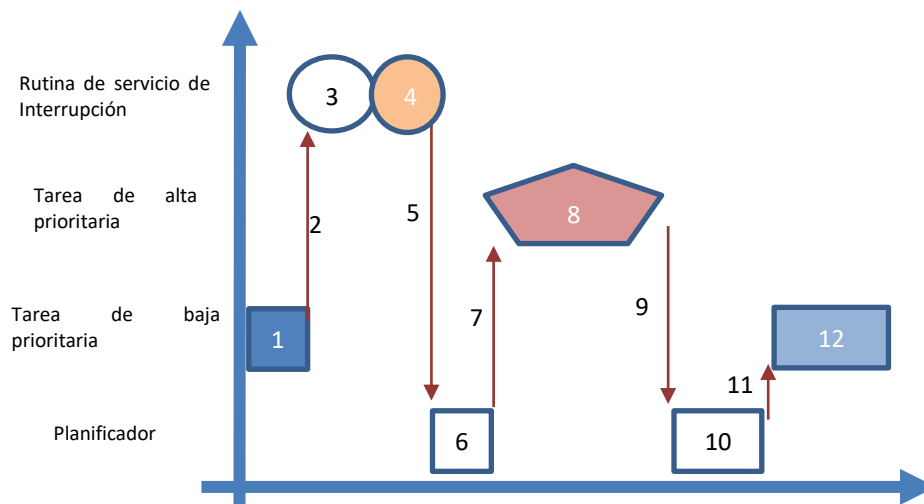


Figura 3 - Planificación Apropiativa – Elaboración propia

La figura 3 ilustra la manera de trabajo de un planificador apropiativo. Una tarea de baja prioridad [1] está ejecutándose cuando ocurre un evento externo [2] que activa una interrupción. El contexto de la tarea y alguna otra información para el planificador es almacenado [3] en la rutina de servicio de interrupción, y se atiende el servicio [4]. En este ejemplo la tarea de prioridad más alta está esperando por éste evento en particular y debe ejecutarse tan pronto como sea posible después de ocurrido el evento. Cuando termina el servicio de interrupción [5], procede al planificador [6], el cual comienza la ejecución de [7] la tarea más prioritaria [8]. Cuando finaliza el control vuelve al planificador [9, 10], el cuál restaura el contexto de la tarea de baja prioridad y permite que continúe su ejecución desde donde fue interrumpida [11, 12].

La planificación apropiativa hace uso intensivo de la pila. El planificador mantiene una pila separada para cada tarea de tal manera que cuando una tarea

continúa su ejecución después de un cambio de contexto, todos los valores de la pila que son únicos de la tarea están correctamente ubicados. Normalmente esto podrían ser direcciones que retornan una subrutina, parámetros y variables locales (para un lenguaje como el C). El planificador también debe salvar el contexto de una tarea suspendida en la pila, debido a que puede ser conveniente hacerlo.

Los planificadores apropiativos son por lo general algo complejo por la gran cantidad de elementos que deben ser direccionados para proveer un cambio de contexto apropiado en cualquier momento. Esto es especialmente cierto con relación al manejo de las interrupciones. También, como puede verse en la figura 3 hay un cierto retraso desde que una interrupción ocurre hasta que la correspondiente rutina de servicio pueda correr. Esta, más la latencia de interrupción, es el tiempo de respuesta de interrupción ($t_4 - t_2$). El tiempo existente entre la culminación de la rutina de servicio de interrupción y la continuación de la ejecución de una tarea se denomina tiempo de recuperación y es mostrado como ($t_7 - t_5$).

El tiempo de respuesta a los eventos se muestra en ($t_7 - t_2$).

Planificación Cooperativa

Un planificador cooperativo tiende a ser más simple que su contraparte apropiativa, ya que todas las tareas deben cooperar para que ocurra el cambio de contexto, el planificador es menos dependiente de las interrupciones y pueden ser más pequeños y potencialmente rápidos. Inclusive, el programador sabe exactamente cuando ocurren los cambios de contexto y puede proteger secciones críticas del código con no colocar una llamada de cambio de contexto en esa sección del programa. Con su relativa simplicidad y control sobre los cambios de contexto, los planificadores cooperativos poseen ciertas ventajas.

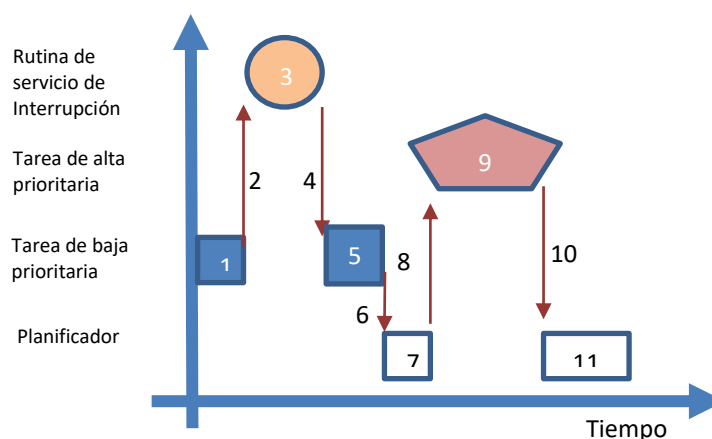


Figura 4 Planificación Cooperativa – Elaboración propia

La figura 4 ilustra la manera de trabajo de un planificador cooperativo. Como en el ejemplo anterior, la tarea de alta prioridad correrá después que el evento manejado por interrupción ocurra. El evento ocurre mientras la tarea de baja prioridad se está ejecutando [1, 5]. La rutina de servicio es ejecutada [2 – 4] y el planificador es informado del evento, pero no habrá cambio de contexto hasta que la tarea de baja prioridad explícitamente lo permita [6]. Una vez que el planificador tiene oportunidad de correr [7], comienza y corre la tarea de alta prioridad hasta su culminación [8, 10]. El planificador [11] comenzará cualquier tarea elegible que posea la mayor prioridad.

En comparación con la planificación apropiativa, la planificación cooperativa tiene la ventaja de poseer tiempos de respuesta de interrupción y tiempo de recuperación más cortos, aunado a su simplicidad. Sin embargo, la respuesta de la aplicación es peor porque una tarea de alta prioridad no puede correr hasta que una de baja prioridad haya liberado el control del procesador a través de un cambio de contexto.

Multitarea Simple

La forma más simple de multitarea involucra “compartir” el procesador equitativamente entre dos o más tareas. Cada tarea corre, en su turno, por algún período de tiempo. Las tareas ciclan (método conocido como Round-Robin), o se ejecutan una después de la otra.

Esto tiene una utilidad limitada, y sufre de los problemas de la arquitectura de Súper Bucle. Eso es porque todas las tareas tienen un acceso igualitario al procesador y su secuencia de ejecución tiende a ser fija.

Multitarea basada en prioridades

Añadir prioridades a las tareas cambia la situación dramáticamente. Porque al asignar prioridades a las tareas se puede garantizar que en cualquier instante, el procesador está corriendo la tarea más importante en el sistema.

Las prioridades pueden ser **estáticas** o **dinámicas**. Las estáticas son prioridades asignadas a las tareas en tiempo de compilación y no cambian mientras corre la aplicación.

Con prioridades dinámicas una tarea puede cambiar su prioridad en tiempo de ejecución. Sería aparente que si la tarea más prioritaria se le permitiera correr continuamente, entonces el sistema ya no sería multitarea. ¿Cómo múltiples tareas con diferentes prioridades pueden coexistir en un sistema multitarea? La respuesta yace en como las tareas realmente se comportan - ¡No siempre están corriendo! En cambio, lo que cierta tarea esté haciendo en un momento en particular depende de su **estado** y en otros factores, tales como los **eventos**.

Estados de las tareas

Un sistema operativo en tiempo real (SOTR) mantiene a cada tarea con un número de **estados** de tareas. La figura 5 ilustra los diferentes estados de una tarea en la mayoría de los SOTR, y las transiciones permitidas entre estados. **Corriendo** es uno de los muchos exclusivos estados de una tarea. Una tarea también puede ser **elegible** para correr, puede estar **demorada**, puede ser **detenida** o incluso **destruida** / **no inicializada**, y puede estar esperando a que ocurra un evento. Estas están explícitas abajo.

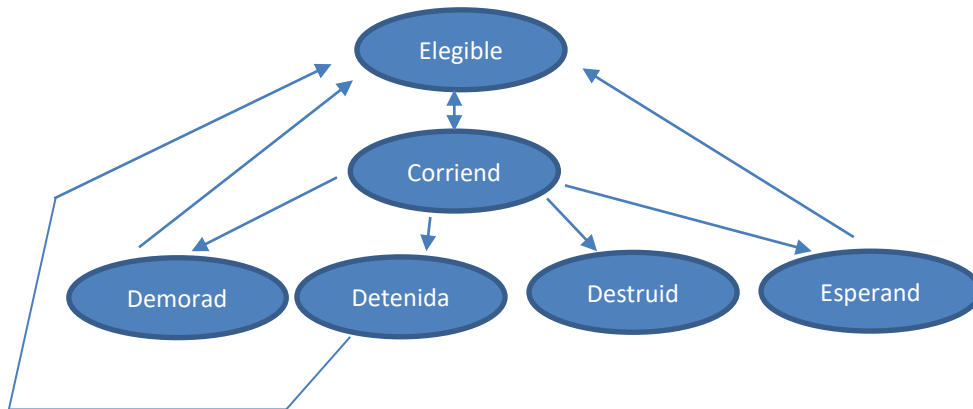


Figura 5 Estados de las tareas en un SOTR común. – Diseño propio

Antes de que una tarea sea creada, se encuentra en el estado de no inicialización. Vuelve a ese **estado** cuando y si es destruida. No hay mucho que se pueda hacer con una tarea destruida, se podría crear otra en su lugar, o crear nuevamente la misma tarea.

Las **transiciones** desde el estado destruido al estado detenido ocurren cuando se crea una tarea a través de una llamada al **servicio** del SOTR encargado de crear las tareas.

Una tarea elegible es aquella que está lista para correr, pero no puede porque no es la tarea con la mayor prioridad. Seguirá en este estado hasta que el planificador determine que ella es la tarea elegible con más prioridad, y se ejecute. Las tareas detenidas, demoradas o en espera pueden convertirse en elegibles a través de las llamadas correspondientes a los servicios del sistema.

Una tarea que esté corriendo retornará al estado elegible después de un simple cambio de contexto. Sin embargo, puede cambiar a un estado diferente ya sea porque la tarea llamó a un servicio del SOTR que destruya, detenga, demore o ponga en espera a la tarea, o por que la tarea sea forzada a uno de estos estados a través de una llamada a un servicio de SOTR desde algún otro lugar en la aplicación.

Una tarea demorada es una tarea que previamente estaba corriendo pero es suspendida y está esperando a que un contador de retardo expire. Una vez que ha expirado, el temporizador del SOTR hace elegible a la tarea nuevamente.

Una tarea detenida es una que previamente estaba corriendo, fue suspendida indefinidamente. No correrá al menos que sea reiniciada través del servicio del sistema que inicializa las tareas.

Una tarea en espera está suspendida y seguirá así hasta que el evento que espera ocurra.

Es típico en una aplicación multitarea tener sus distintas tareas con muchos estados diferentes en un momento dado. Las tareas periódicas tienden a estar demoradas en un momento en particular. Las tareas de baja prioridad pueden estar en el estado elegible pero no podrán correr porque una tarea de mayor prioridad está ejecutándose. Algunas tareas tienden a esperar un evento, pueden estar destruidas o detenidas, pero depende del planificador administrar las todas garantizar que cada tarea corra cuando le toque. El planificador y otras partes del RTOS aseguran la transición exitosa de las tareas de un estado a otro, y el planificador es el corazón de la aplicación multitarea basada en prioridades.

Un SOTR trata a las tareas con un estado en particular de la misma manera, y por tanto mejora el desempeño de la aplicación. Por ejemplo, no se deberá gastar ningún ciclo de procesador en una tarea que está detenida o destruida. Después de todo, están "sentadas allí" y se quedarán así de manera indefinida, o hasta que el programa las haga elegibles.

Multitarea basada en eventos

Una tarea demorada está esperando a que su temporizador de retardo expire; la expiración de un temporizador de retardo es un ejemplo de un evento, y estos pueden causar que una tarea cambie de estado; por lo tanto los eventos se usan para controlar la ejecución de la tarea. Ejemplos de eventos pueden ser:

- Una interrupción,
- Aparición de un error,
- Un temporizador expirando,
- Una interrupción periódica,
- Un recurso siendo liberado,
- Un pin de E/S cambiando de estado.
- Una tecla presionada de un teclado,
- Un carácter siendo recibido o transmitido vía RS-232,
- Información que se pasa de una parte de la aplicación a otra, etc.

Resumiendo, un evento puede ser cualquier acción que ocurre tanto interna como externa al procesador. Se asocia un evento al resto de la aplicación (primeramente tareas, pero también a la rutina de servicio de interrupción, y código de fondo) a través de los servicios de eventos del SOTR.

Eventos y comunicación entre tareas

Un SOTR soporta muchas maneras de comunicarse con las tareas. En la multitarea basada en eventos, para que una tarea reaccione a un evento, éste debe establecer algún tipo de comunicación con la tarea. Las tareas también pueden desear comunicarse unas con otras. Semáforos, mensajes, cola de mensajes, se usan entre la comunicación entre tareas.

Inversión de Prioridades

Ocurren cuando una tarea de alta prioridad está esperando un recurso controlado por una tarea de baja prioridad. La tarea de alta prioridad debe esperar hasta que la tarea de baja prioridad libere el recurso. Como resultado la prioridad de la tarea más prioritaria es efectivamente reducida a aquella de baja prioridad.

Semáforos

Un semáforo es una variable especial protegida (o tipo abstracto de datos) que constituye el método clásico para restringir o permitir el acceso a recursos compartidos (por ejemplo, un recurso de almacenamiento) en un entorno de multiprocesamiento. Fueron inventados por Edsger Dijkstra y se usaron por primera vez en el sistema operativo THEOS.

Uso de los semáforos

Los semáforos se emplean para permitir el acceso a diferentes partes de programas (llamados secciones críticas) donde se manipulan variables o recursos que deben ser accedidos de forma especial. Según el valor con que son inicializados se permiten a más o menos procesos utilizar el recurso de forma simultánea.

Un tipo simple de semáforo es el binario, que puede tomar solamente los valores 0 y 1. Se inicializan en 1 y son usados cuando sólo un proceso puede acceder a un recurso a la vez. Son esencialmente lo mismo que los mutex (mutual exclusion) o exclusión mutua.

Cuando el recurso está disponible, un proceso accede y decrementa el valor del semáforo con la operación P. El valor queda entonces en 0, lo que hace que si otro proceso intenta decrementarlo tenga que esperar. Cuando el proceso que decrementó el semáforo realiza una operación V, algún proceso que estaba esperando puede despertar y seguir ejecutando.

Para hacer que dos procesos se ejecuten en una secuencia predeterminada puede usarse un semáforo inicializado en 0. El proceso que debe ejecutar primero en la

secuencia realiza la operación V sobre el semáforo antes del código que debe ser ejecutado después del otro proceso. Éste ejecuta la operación P. Si el segundo proceso en la secuencia es programado para ejecutar antes que el otro, al hacer P dormirá hasta que el primer proceso de la secuencia pase por su operación V. Este modo de uso se denomina señalación (signaling), y se usa para que un proceso o hilo de ejecución le haga saber a otro que algo ha sucedido.

Sistema operativo en tiempo real (SOTR)

Resumen

Estos sistemas se caracterizan por tener el tiempo como un parámetro clave. Por ejemplo, en los sistemas de control de procesos industriales, las computadoras en tiempo real tienen que recolectar datos acerca del proceso de producción y utilizarlos para controlar las máquinas en la fábrica. A menudo hay tiempos de entrega estrictos que se deben cumplir. Por ejemplo, si un auto se desplaza sobre una línea de ensamblaje, deben llevarse a cabo ciertas acciones en determinados instantes. Si un robot soldador realiza su trabajo de soldadura antes o después de tiempo, el auto se arruinará. Si la acción debe ocurrir sin excepción en cierto momento (o dentro de cierto rango), tenemos un sistema en tiempo real duro. Muchos de estos sistemas se encuentran en el control de procesos industriales, en aeronáutica, en la milicia y en áreas de aplicación similares. Estos sistemas deben proveer garantías absolutas de que cierta acción ocurrirá en un instante determinado.

Otro tipo de sistema en tiempo real es el **sistema en tiempo real suave**, en el cual es aceptable que muy ocasionalmente se pueda fallar a un tiempo predeterminado. Los sistemas de audio digital o de multimedia están en esta categoría. Los teléfonos digitales también son ejemplos de sistema en tiempo real suave.

BIBLIOGRAFÍA

Tanenbaum Andrew S. (2011). Sistemas Operativos Modernos. 3er edición. Amsterdam, Holanda - ISBN: 978-607-442-046-3.

World Wide Web. Disponible en:

www.w3c.es

Pumkin Incorporated. Disponible en:

www.pumpkunspace.com

Salvo SA. Disponible en:

www.pmsalvo.com

uC/OS. Disponible en:

www.fci.ucm.es



Atribución-No Comercial-Sin Derivadas

Se permite descargar esta obra y compartirla, siempre y cuando no sea modificado y/o alterado su contenido, ni se comercialice. Referenciarlo de la siguiente manera:

Universidad Tecnológica Nacional Facultad Regional Córdoba (S/D). Material para la Tecnicatura Universitaria en Programación, modalidad virtual, Córdoba, Argentina.