Práctica 4 Exclusión Mútua

Sección crítica de código

Una sección crítica de código, también llamada una *región crítica*, es código que necesita ser tratado de forma indivisible. Una vez que la sección de código empieza a ejecutarse, esta no debe ser interrumpida. Para asegurar esto, típicamente las interrupciones son deshabilitadas antes de que el código crítico sea ejecutado y son habilitadas cuando termina la ejecución del código crítico.

Recurso

Un recurso es cualquier entidad usada por una tarea. Puede ser un dispositivo de I/O tal como una impresora, un teclado, un display, etc. o una variable, una estructura, un arreglo, etc.

Recurso compartido

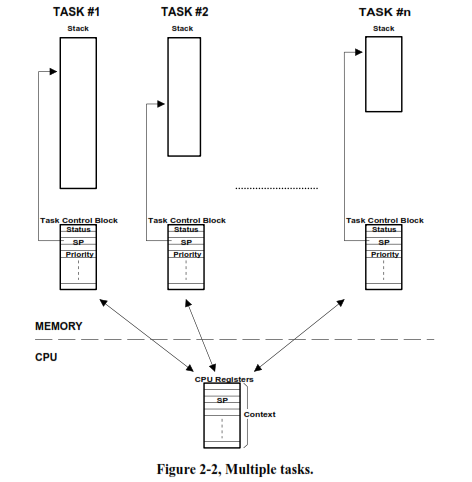
Un recurso compartido es un recurso que puede ser usado por más de una tarea. Cada tarea debe ganar acceso exclusivo al recurso compartido para prevenir corrupción de datos. Esto se conoce como *exclusión mutua*.

Multitasking

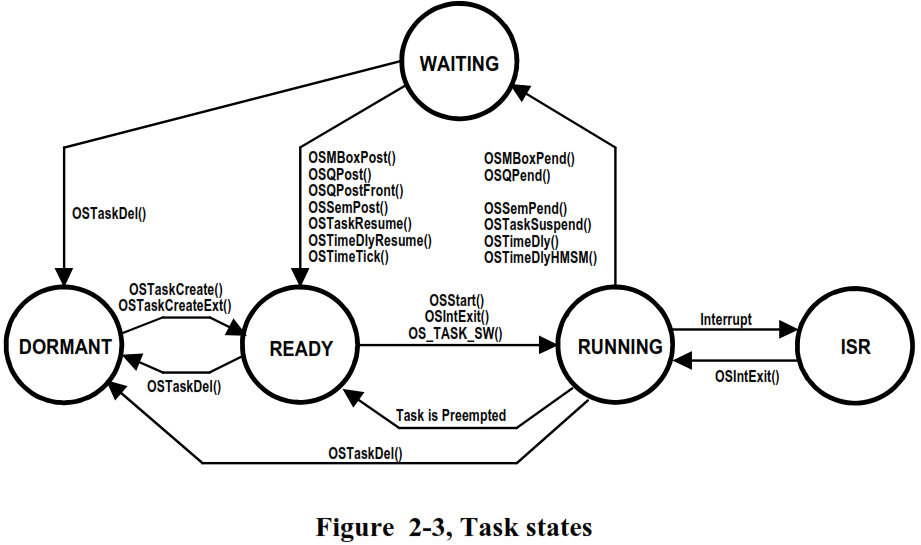
Multitasking o multitarea es el proceso de agendar el uso de la CPU (Central Processing Unit) y switchear el mismo entre varias tareas (scheduling and switching); el mutitasking maximiza la utilización de la CPU y también proporciona construcción modular de aplicaciones. Uno de los aspectos más importantes del multitasking es que permite al programador de aplicación manejar la complejidad inherente en las aplicaciones de tiempo real. Los programas de aplicación típicamente son más fáciles de diseñar y de mantener si se usa multitarea.

Tarea

Una tarea, también llamada un hilo, es un programa simple que “piensa” que la CPU solo es para él. El proceso de diseño para una aplicación de tiempo real involucra partir el trabajo a realizar en tareas las cuales son responsables de una parte del problema. A cada tarea se le asigna una prioridad, su propio conjunto de registros de CPU, y su propia área de pila (como se muestra en la siguiente figura).



Cada tarea típicamente es un ciclo infinito que puede estar en alguno de los cinco estados: Durmiendo (DORMANT), Listo (READY), Ejecución (RUNNING), Esperando un evento (WAITING FOR AN EVENT), on Interrumpido (INTERUPTED), (véase la Figura 2-3). El estado DORMANT corresponde a una tarea la cual está en memoria, pero no ha sido puesta disponible para el kernel multitarea. Una tarea está en estado READY cuando se puede ejecutar, pero su prioridad es menor que la tarea que se está ejecutando actualmente. Una tarea está en estado RUNNING cuando tiene el control de la CPU. Una tarea está en estado WAITING FOR AN EVENT cuando requiere la ocurrencia de un evento (puede estar esperando que se complete una operación de I/O, que un recurso compartido esté disponible, que ocurra un pulso de temporización, que expire un tiempo etc.) Finalmente, una tarea está INTERRUPTED cuando ha ocurrido una interrupción y la CPU está en proceso de dar servicio a la interrupción. La Figura 2-3 también muestra las funciones proporcionadas por %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document} para hacer que una tarea cambie de un estado a otro.



Context switch (o Task switch)

Cuando un kernel multitarea decide ejecutar una tarea diferente, simplemente guarda el contexto de la tarea (los registros de la CPU) en el área de almacenamiento de contexto de la tarea actual –su pila (véase la Figura 2-2). Una vez que esta operación es realizada, el contexto de la nueva tarea es restaurado de su área de almacenamiento y retoma la ejecución del código de la nueva tarea. Este proceso es llamado un cambio de contexto (context switch) o un cambio de tarea (tast switch). Los cambios de contexto le agregan una sobrecarga (overhead) a la aplicación. Mientras más registros tenga una CPU mayor es la sobrecarga. El tiempo para realizar un cambio de contexto depende de cuántos registros tengan que ser guardados y restaurados por la CPU. El desempeño de un kernel de tiempo real no debe ser juzgado por cuántos cambios de contexto por segundo es capaz de hacer el kernel.

Kernel

El kernel es la parte de un sistema multitarea responsable de administrar las tareas (esto es, de administrar el tiempo de la CPU) y la comunicación entre las tareas. El servicio fundamental proporcionado por el kernel es el cambio de contexto. El uso de un kernel de tiempo real generalmente simplificará el diseño de sistemas permitiendo que las aplicaciones sean divididas en múltiples tareas manejadas por el kernel. Un kernel agregará sobrecarga al sistema porque requiere ROM extra (espacio de código), RAM adicional para las estructuras de datos del kernel, pero principalmente, porque cada tarea requiere su propio espacio de pila el cual tiende a consumir RAM muy rápidamente. Un kernel también consumirá tiempo de CPU (típicamente entre 2 y 5%).

Sin embargo, un kernel puede permitir un mejor uso de la CPU proporcionando servicios indispensables tales como administración de semáforos, mailboxes, queues, retardos de tiempo, etc.

Scheduler

El scheduler, también conocido como *dispatcher*, es la parte del kernel que es responsable de determinar cuál tarea se ejecutará a continuación. La mayoría de los kernel de tiempo real son basados en prioridades. A cada tarea se le asigna una prioridad de acuerdo a su importancia. La prioridad para cada tarea depende de cada aplicación (es “aplicación específica”). En un kernel basado en prioridad, el control de la CPU siempre le será dado a la tarea con prioridad más alta que esté en el estado listo para correr (READY). Cuándo la tarea con prioridad más alta obtiene la CPU, sin embargo, está determinado por el tipo de kernel usado. Hay dos tipos de kernels basados en prioridad: no expulsivos y expulsivos (non-preemptive and preemptive).

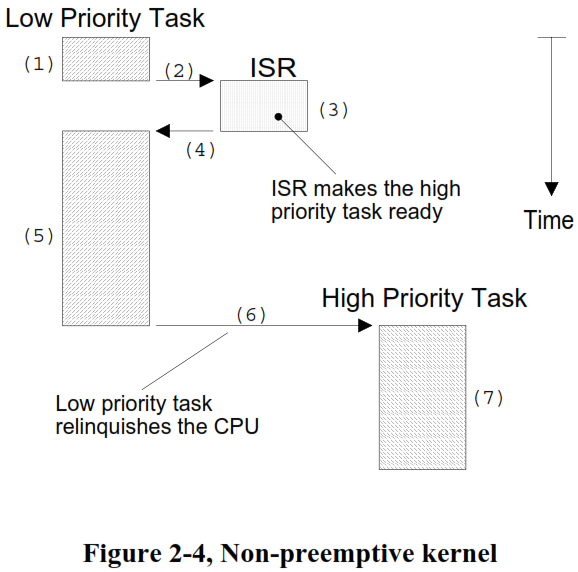
Kernel no expulsivo

Los kernels no expulsivos requieren que cada tarea haga algo para ceder explícitamente el control de la CPU. Para mantener la ilusión de concurrencia, este proceso debe ser hecho frecuentemente. El scheduling no expulsivo también es llamado multitareas coperativas (cooperative multitasking); las tareas cooperan unas con otras para compartir la CPU. En estos kernels los eventos asíncronos son manejados por rutinas de servicio de interrupción (ISRs). Una ISR puede poner a una tarea de prioridad más alta en el estado listo para correr, pero la ISR siempre regresará a la tarea interrumpida. La nueva tarea de más alta prioridad obtendrá el control de la CPU solamente cuando la tarea actual ceda la CPU.

Una de las ventajas de un kernel no expulsivo es que la latencia de interrupción es típicamente baja. En el nivel de tarea, los kernels no expulsivos pueden también usar funciones no reentrantes. Las funciones no reentrantes pueden ser usadas por cada tarea sin temor de corrupción por otra tarea. Esto es debido a que cada tarea puede correr hasta completarse antes de renunciar a la CPU. Sin embargo, las funciones no reentrantes no deben ceder el control de la CPU.

La respuesta en el nivel de tarea cuando se usa un kernel no expulsivo depende de la tarea que tenga el tiempo para completarse más largo.

Otra ventaja de los kernels no expulsivos es la menor necesidad de vigilar datos compartidos a través del uso de semáforos. Cada tarea es dueña de la CPU y usted no tiene que preocuparse de que una tarea sea expulsada. Esta no es una regla absoluta, y en algunos casos, se deben usar semáforos. Aun podría requerirse el uso de semáforos de exclusión mutua para dispositivos de I/O compartidos; por ejemplo, una tarea podría requerir acceso exclusivo a una impresora.



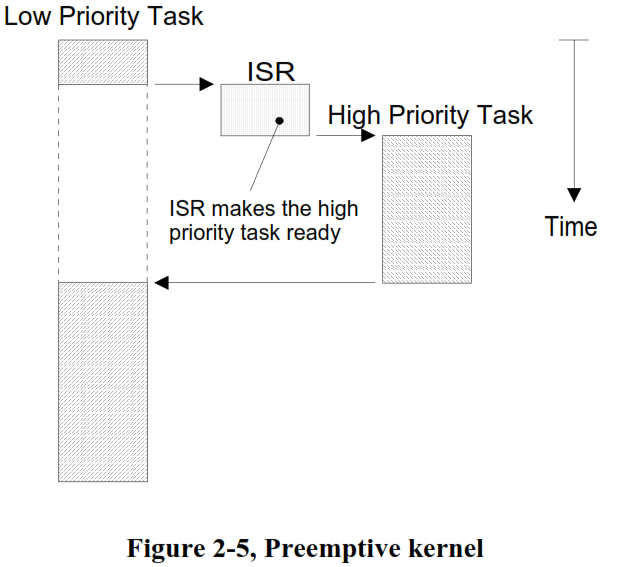
En la Figura 2-4 se muestra el perfil de ejecución de un kernel no expulsivo. Una tarea se está ejecutando F2-4(1) pero se presenta una interrupción. Si las interrupciones están habilitadas, la CPU vectoriza (i.e., salta) a la ISR F2-4(2). Las ISR maneja el evento F2-4(3) y pone una tarea de más alta prioridad en el estado READY. Una vez que se completa la ISR se ejecuta una instrucción *Return From Interrupt* y la CPU regresa a la tarea interrumpida F2-4(4). El código de la tarea retoma su ejecución en la instrucción que sigue después de la instrucción que se estaba ejecutando cuando se presentó la interrupción F2-4(5). Cuando el código de la tarea se completa, esta llama a un servicio proporcionado por el kernel para ceder la CPU en favor de otra tarea F2-4(6). Entonces la nueva tarea de prioridad más alta se ejecuta para manejar el evento señalado por la ISR F2-4(7).

El inconveniente más importante de un kernel no expulsivo es su responsividad. Una tarea de más alta prioridad que ha sido puesta en READY podría tener que esperar un tiempo largo para correr, porque la tarea actual cederá la CPU cuando esté lista para hacerlo. El tiempo de respuesta a nivel de tarea en kernels no expulsivos es no determinístico; usted realmente nunca sabe cuándo la tarea de prioridad más alta obtendrá el control de la CPU. Depende de la aplicación ceder el control de la CPU.

En resumen, un kernel no expulsivo le permite a cada tarea correr hasta que voluntariamente cede el control de la CPU. Una interrupción expulsará una tarea. Una vez completada la ISR, la ISR regresará a la tarea interrumpida. La respuesta a nivel tarea es no determinística.

Kernel expulsivo

Se usa un kernel expulsivo cuando la responsividad es importante. Debido a esto, %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document} y la mayoría de los kernels de tiempo real comerciales son expulsivos. En estos kernels el control de la CPU siempre se le da a la tarea de más alta prioridad lista para correr. Cuando una tarea hace que otra tarea de prioridad más alta pase al estado de lista para correr, la tarea actual es expulsada (suspendida) e inmediatamente se le da el control de la CPU a la tarea de más alta prioridad. Si una ISR pone a una tarea de prioridad más alta en el estado READY, cuando la ISR se completa, la tarea interrumpida es suspendida y la nueva tarea de prioridad más alta es retomada. Esto se ilustra en la Figura 2-5.



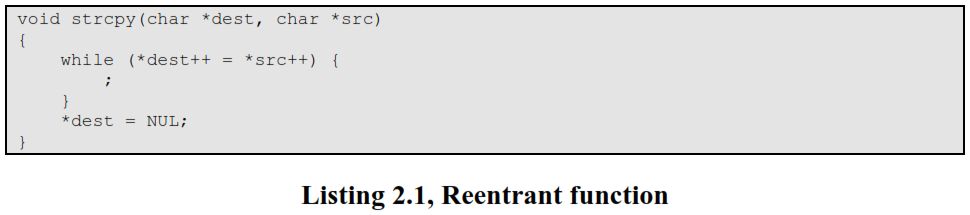
Con un kernel expulsivo, la ejecución de la tarea de más alta prioridad es determinística; usted puede determinar cuándo la tarea de más alta prioridad obtendrá el control de la CPU. Usando un kernel expulsivo, el tiempo de respuesta a nivel de tarea se minimiza.

El código de aplicación que usa un kernel expulsivo no debe usar funciones funciones no-reentrantes a menos que se asegure acceso exclusivo a esas funciones a través del uso de semáforos de exclusión mutua, porque tanto una tarea de baja prioridad como una tarea de alta prioridad pueden hacer uso de una función común. La corrupción de datos podría ocurrir si la tarea de más alta prioridad expulsa una tarea de prioridad más baja que está haciendo uso de la función.

En resumen, un kernel expulsivo siempre ejecuta la tarea de prioridad más alta que está lista para correr. Una interrupción expulsará una tarea. Cuando se completa la ISR, el kernel retomará la ejecución de la tarea con prioridad más alta lista para correr (no la de la tarea interrumpida). La respuesta a nivel de tarea es óptima y determinística. %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document} es un kernel expulsivo.

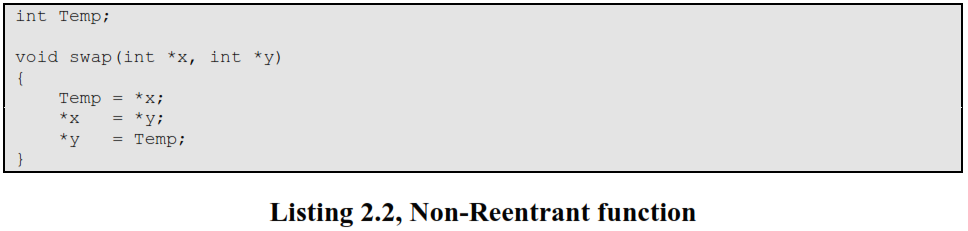
Reentrancia

Una *función reentrante* es una función que puede ser usada por más de una tarea sin temor de corrupción de datos. Una función reentrante puede ser interrumpida en cualquier momento y retomada en un momento posterior sin pérdida de datos. Las funciones reentrantes usan variables locales (i.e., registros de CPU o variables sobre la pila) o protegen los datos cuando usan variables globales. En el listado 2.1 se muestra un ejemplo de una función reentrante.

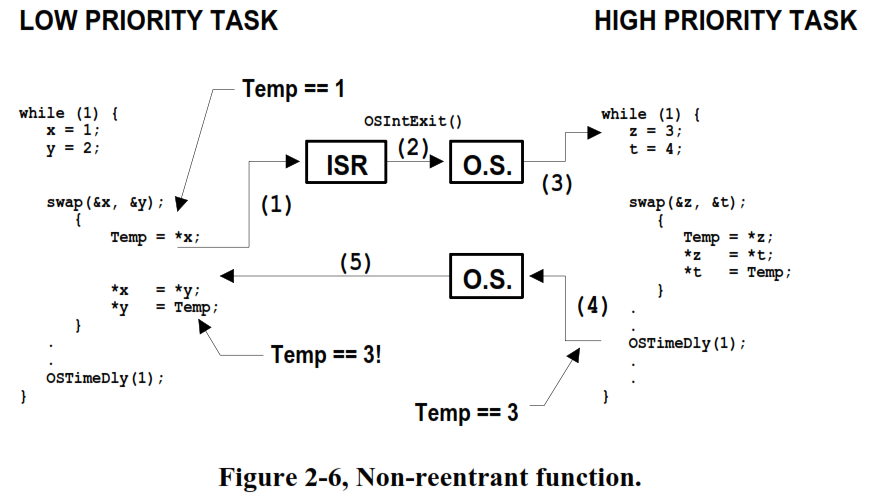


Debido a que en la pila se colocan copias de los argumentos para strcpy(), strcpy() puede ser invocada por varias tareas sin temor a que las tareas corrompan los apuntadores de las otras tareas.

Un ejemplo de una función no reentrante se muestra en el listado 2.2. swap() es una función simple que intercambia los contenidos de sus dos argumentos. En los siguientes comentarios, se asume que usted está usando un kernel expulsivo, que las interrupciones están habilitadas y que Temp está declarada como un entero global:



El programador intentó hacer swap() utilizable por cualquier tarea. La Figura 2-6 muestra lo que podría pasar si una tarea de baja prioridad es interrumpida mientras swap() está ejecutándose F2-6(1). Note que en este punto Temp contiene 1. La ISR pone a la tarea de más alta prioridad en el estado (READY) lista para correr, y entonces, al terminar la ISR F2-6(2), el kernel (suponiendo que se está usando %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document}) es invocado para switchear hacia esta tarea F2-6(3). La tarea de prioridad alta establece Temp a 3 e intercambia los contenidos de sus variables correctamente (esto es, z es 4 y t es 3). La tarea de prioridad alta eventualmente cede el control a la tarea de prioridad baja F2-6(4) llamando a un servicio del kernel para retardarse a símisma un tick de reloj (descrito posteriormente). Entonces se retoma la tarea de prioridad baja F2-6(5). Note que, en este punto, ¡Temp aun está establecido a 3! Cuando la tarea de prioridad baja retoma su ejecución, establece y a 3 y no a 1.



Note que este es un ejemplo simple y es obvio cómo hacer que el código sea reentrante. Sin embargo, otras situaciones no son tan fáciles de resolver. Un error causado por una función no-reentrante podría no mostrarse en su aplicación durante la fase de prueba; típicamente ocurrirá una vez que el producto ha sido liberado. Si usted no tiene experiencia en multitasking, deberá ser cuidadoso cuando use funciones no-reentrantes.

Se puede hacer que la función swap() sea reentrante usando una de las siguientes técnicas:

a)Declarar Temp local a swap.

b)Deshabilitar las interrupciones antes de la operación y habilitarlas después.

c)Usar un semáforo (descrito posteriormente).

Exclusión Mutua

La forma más fácil de hacer comunicación entre tareas es a través de estructuras de datos compartidas. Esto es especialmente fácil cuando todas las tareas existen en un solo espacio de direcciones. Las tareas pueden entonces, referenciar variables globales, apuntadores, buffers, listas enlazadas, buffers circulares, etc. Mientras compartir datos simplifica el intercambio de información, usted debe asegurar que cada tarea tiene acceso exclusivo al dato para evitar la contención y corrupción del dato. Los métodos más comunes para obtener acceso exclusivo a recursos compartidos son:

a)Deshabilitar interrupciones

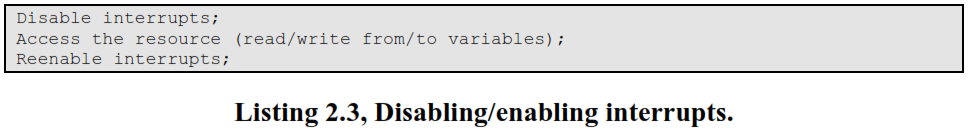
b)Test-And-Set

c)Deshabilitar scheduling

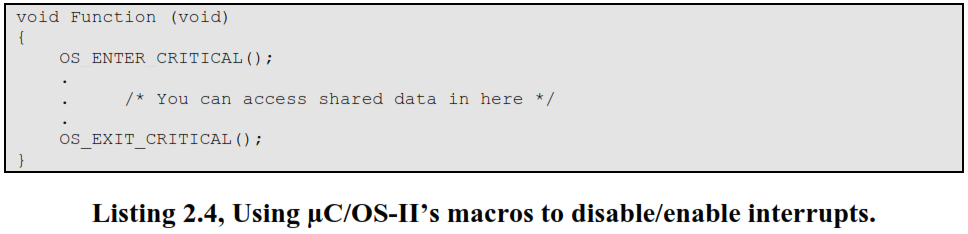
d)Usar semáforos

Exclusión Mutua, Deshabilitando y habilitando interrupciones

La forma más fácil y rápida de ganar acceso exclusivo a un recurso compartido es deshabilitando y habilitando interrupciones como se muestra en el pseudocódigo del listado 2.3.



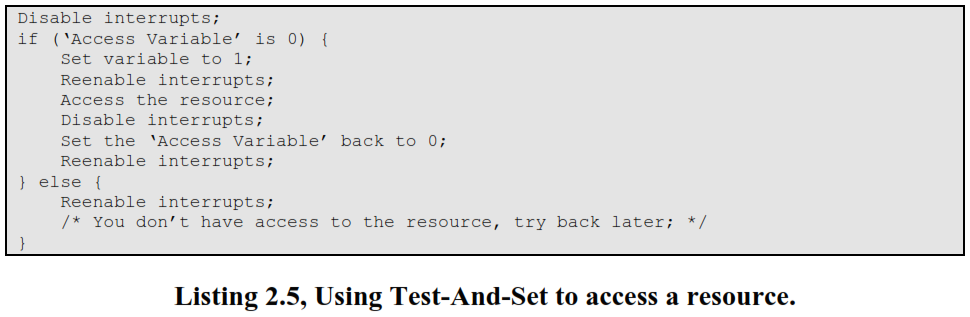
%FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document} usa esta técnica (como lo hace la mayoría, si no es que todos los kernels) para acceder a variables y estructuras de datos internas. De hecho, %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document} proporciona dos macros para permitirle a usted deshabilitar y entonces habilitar interrupciones desde su código C: OS\_ENTER\_CRITICAL() y OS\_EXIT\_CRITICAL(), respectivamente (véase OS\_CPU.H, OS\_ENTER\_CRITICAL() y OS\_EXIT\_CRITICAL() ). Estas macros se deben usar pareadas como se muestra en el listado 2.4



Usted debe ser cuidadoso, sin embargo, de no deshabilitar las interrupciones durante demasiado tiempo, porque esto afecta la respuesta de su sistema a las interrupciones. Esto es conocido como *latencia de interrupción.* Usted debe considerar este método cuando está cambiando o copiando solo unas cuantas variables. También, esta es la única forma en que una tarea puede compartir variables o estructuras de datos con una ISR. En todos los casos, usted debe mantener las interrupciones deshabilitadas por tan poco tiempo como sea posible.

Exclusión Mutua, Test-And-Set

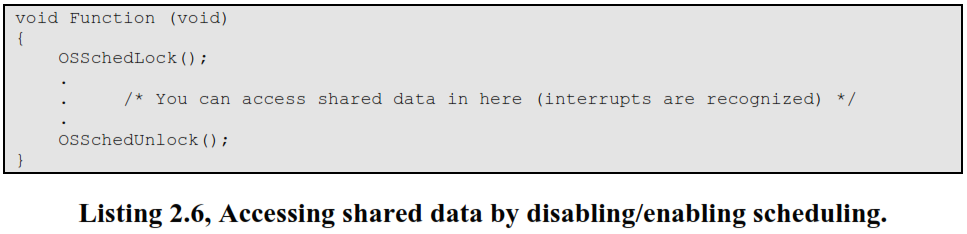
Si usted no está usando un kernel, dos funciones podrían acordar que, para acceder a un recurso, primero deben revisar una variable global, y si la variable es 0 la función tiene acceso al recurso. Para prevenir a otra función de acceder al recurso, sin embargo, la primera función que obtiene el recurso simplemente establece la variable a 1. Esto es comúnmente llamado una operación Test-And-Set (o TAS). La operación TAS debe ser realizada indivisiblemente (por el procesador), o usted debe deshabilitar las interrupciones cuando está haciendo el TAS sobre la variable como se muestra en el listado 2.5.



Algunos procesadores realmente implementan una operación TAS en hardware (por ejemplo, la familia de procesadores 68000 tienen la instrucción TAS).

Exclusión Mutua, Deshabilitando y habilitando el scheduler

Si su tarea no está compartiendo variables o estructuras de datos con una ISR entonces usted puede deshabilitar/habilitar el scheduling (véase sección 3.06, Locking and Unlocking the Scheduler de [Labrosse, Jean J]) como se muestra en el listado 2.6 (usando %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document} como un ejemplo). En este caso, dos o más tareas pueden compartir datos sin la posibilidad de contención. Usted debe notar que mientras el scheduler está bloqueado las interrupciones están habilitadas y, si una interrupción ocurre mientras se está ejecutando la sección crítica, la ISR será ejecutada inmediatamente. Al final de la ISR, el kernel siempre regresará a la tarea interrumpida aun si una tarea de prioridad más alta ha sido puesta en el estado de lista para correr por la ISR. El scheduler será invocado cuando la función OSSchedUnlock() sea llamada para ver si hay una tarea de prioridad más alta que ha sido puesta en el estado lista para correr por la ISR. Si hay una tarea que esté en el estado lista para correr sucederá un cambio de contexto. Aunque este método trabaja bien, usted debe evitar deshabilitar el scheduler porque esto va en contra del propósito de tener un kernel en primer lugar. En lugar de esto se debe elegir el siguiente método.



Exclusión Mutua, Semáforos

El semáforo fue inventado por Edgser Dijkstra a mediados de los 1960s. Un semáforo es un mecanismo protocolo ofrecido por la mayoría de los kernels multitasking. Los semáforos son usados para:

a)control de acceso a un recurso compartido (exclución mutua);

b)señalar la ocurrencia de un evento;

c)permitir a dos tareas sincronizar sus actividades.

Un semáforo es una llave que su código adquiere para continuar su ejecución. Si el semáforo ya está en uso, la tarea que hizo la petición es suspendida hasta que el semáforo es liberado por su dueño actual. En otras palabras, la tarea que hace la petición dice: “dame la llave. Si alguien más la está usando, estoy dispuesto a esperar por ella”.

Hay dos tipos de semáforos: semáforos *binarios* y semáforos *contadores*. Como su nombre lo implica, un semáforo binario solo puede tomar dos valores: 0 o 1. Un semáforo contador permite valores entre 0 y 255, 65535 o 4294967295, dependiendo de si el mecanismo semáforo es implementado usando 8, 16, o 32 bits, respectivamente. El tamaño real depende del kernel utilizado. Junto con el valor del semáforo, el kernel también necesita llevar el registro de las tareas que están esperando por la disponibilidad del semáforo.

Generalmente solo hay tres operaciones que pueden ser realizadas sobre un semáforo: INITIALIZE (también llamada CREATE), WAIT (también llamada PEND), y SIGNAL (también llamada POST).

El valor inicial del semáforo debe ser proporcionado cuando el semáforo es inicializado. La lista de tareas que están esperando siempre está inicialmente vacía.

Una tarea que desea adquirir el semáforo realizará una operación WAIT. Si el semáforo está disponible (el valor del semáforo es mayor que 0), el valor del semáforo es decrementado y la tarea continúa su ejecución. Si el valor del semáforo es 0, la tarea que está realizando un WAIT sobre el semáforo es colocada en una lista de espera. La mayoría de los kernels permiten especificar un tiempo límite (timeout); si el semáforo no está disponible dentro de una cierta cantidad de tiempo, la tarea que está haciendo la petición es puesta en el estado lista para correr y se regresa un código de error a la tarea que hizo la llamada (indicando que ocurrió un tiemout).

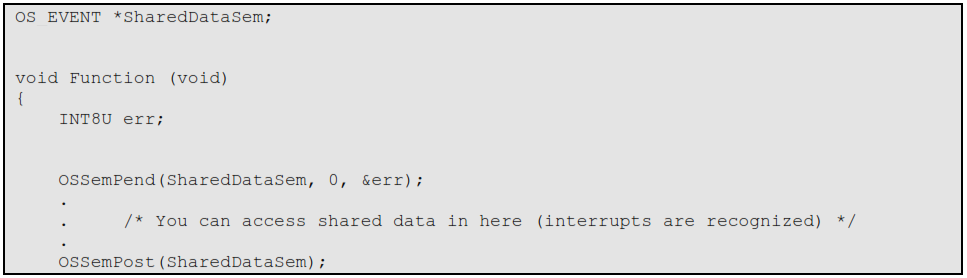
Una tarea libera un semáforo realizando una operación SIGNAL. Si ninguna tarea está esperando el semáforo, simplemente se incrementa el valor del semáforo. Si alguna tarea está esperando el semáforo, sin embargo, se pone una de las tareas en el estado lista para correr y el valor del semáforo no se incrementa; la llave se le da a una de las tareas que estaba esperando por ella. Dependiendo del kernel, la tarea que recibirá el semáforo será:

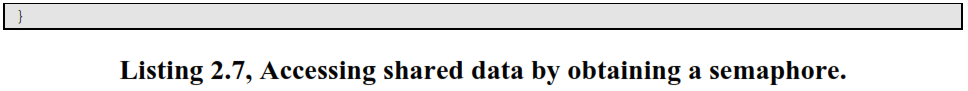
a)la tarea de más alta prioridad que estaba esperando el semáforo, o

b)la primera tarea que pidió el semáforo (First In First Out, o FIFO).

Algunos kernels le permiten a usted elegir algún método a través de una opción cuando el semáforo es inicializado. %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document} solamente soporta el primer método. Si la tarea puesta a READY tiene una prioridad más alta que la tarea actual (la tarea que está liberando el semáforo), ocurrirá un cambio de contexto (con un kernel expulsivo) y la tarea con prioridad más alta retomará su ejecución; la tarea actual será suspendida hasta que ella nuevamente se convierta en la tarea de más alta prioridad con estado lista para correr (READY).

El listado 2.7 muestra cómo se puede compartir datos usando un semáforo (usando %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document}). Cualquier tarea que necesita acceder al dato compartido llamará a OSSemPend() y cuando la tarea ha terminado con el dato, la tarea llama a OSSemPost(). Se debe notar que el semáforo es un objeto que necesita ser inicializado antes de que sea usado y para exclusión mutua, un semáforo es inicializado al valor 1. Utilizar un semáforo para acceder a datos compartidos no afecta la latencia de interrupción y, si una ISR o la tarea actual pone a una tarea de mayor prioridad en el estado ready-to-run mientras está accediendo a los datos entonces, esta tarea de prioridad más alta se ejecutará inmediatamente.

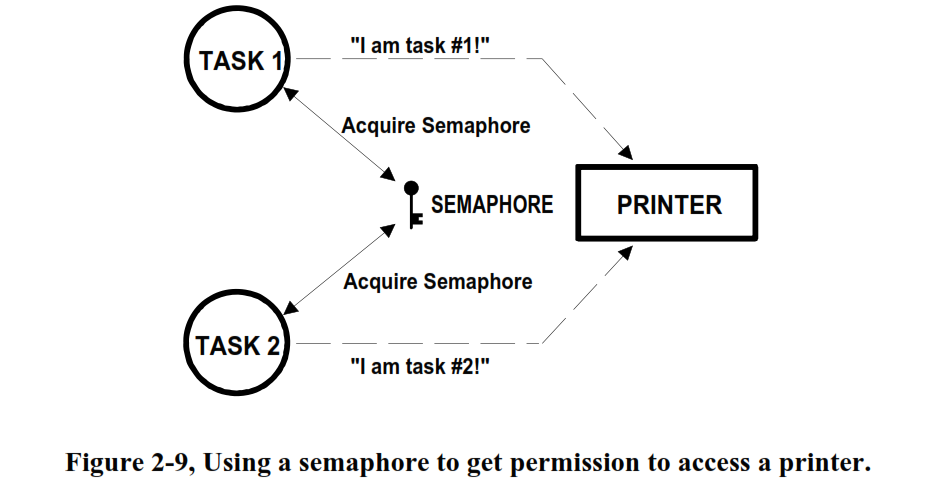




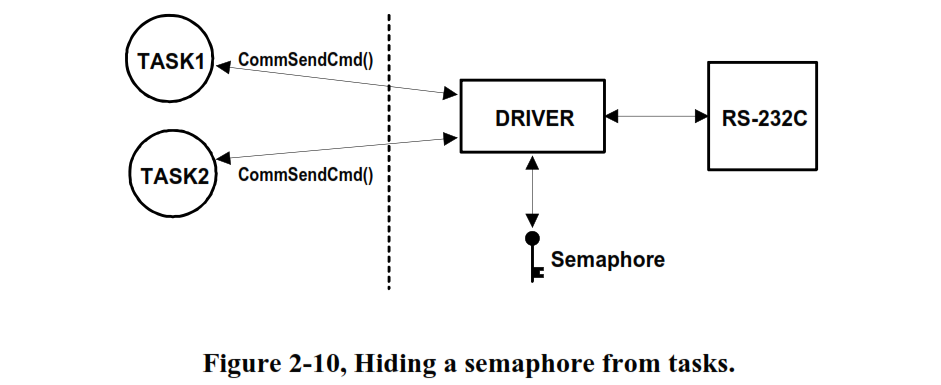
Los semáforos son especialmente útiles cuando las tareas están compartiendo dispositivos de I/O. Imagine lo que pasaría si se les permitiera a dos tareas enviar caracteres a una impresora al mismo tiempo. La impresora contendría datos intercalados de cada tarea. Por ejemplo, si la tarea #1 tratara de imprimir “I am task #1!” y la tarea #2 tratara de imprimir “I am task #2” entonces la impresión podría verse como sigue:

I Ia amm t tasask k#1 #!2!

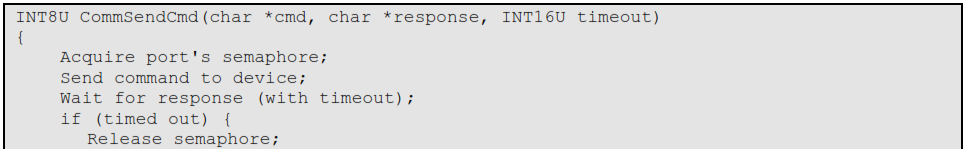
En este caso, podemos usar un semáforo e inicializarlo a 1 (i.e., un semáforo binario). La regla es simple: para acceder a la impresora cada tarea primero debe obtener el semáforo del recurso. La Figura 2-9 muestra las tareas compitiendo por un semáforo para ganar acceso exclusivo a la impresora. Note que el semáforo es representado simbólicamente por una llave que indica que cada tarea debe obtener esta llave para usar la impresora.

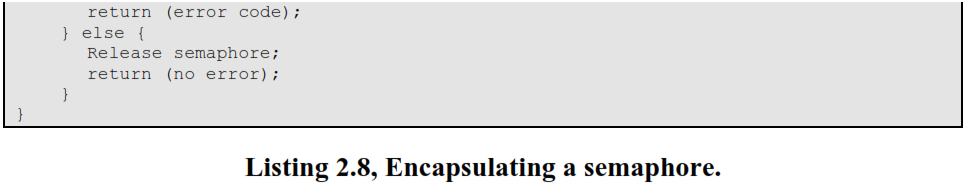


El ejemplo de arriba implica que cada tarea debe saber acerca de la existencia del semáforo para acceder al recurso. Hay situaciones en las que es mejor encapsular el semáforo. Entonces cada tarea no sabría que en realidad está adquiriendo un semáforo cuando está accediendo al recurso. Por ejemplo, un puerto RS-232C es usado por múltiples tareas para enviar comandos y recibir respuestas de un dispositivo conectado en el otro extremo del puerto RS-232C. En la Figura 2-10 se muestra un diagrama de flujo.



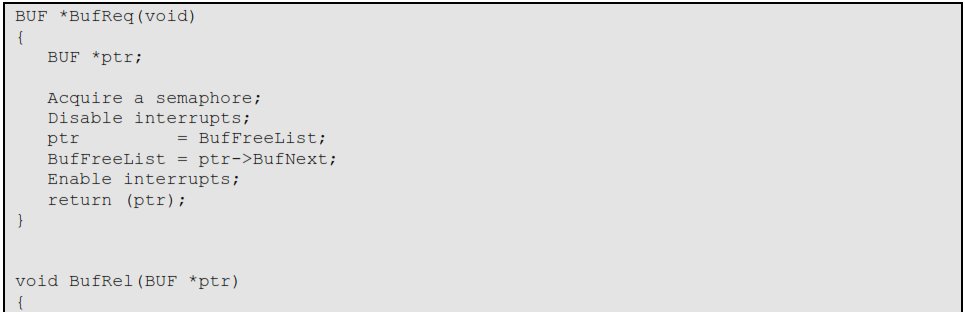
La función CommSendCmd() es llamada con tres argumentos: la cadena ASCII conteniendo el comando, un apuntador a la cadena de respuesta del dispositivo, y finalmente, un plazo de vencimiento (timeout) en caso de que el dispositivo no responda dentro de una cierta cantidad de tiempo. El pseudocódigo para esta función es:

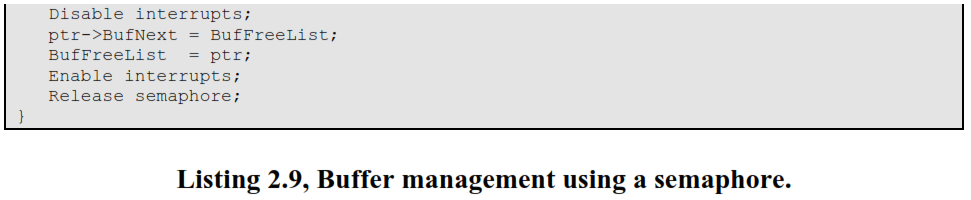




Cada tarea que necesita enviar un comando al dispositivo tiene que llamar a esta función. Se asume que el semáforo está inicializado a 1 (i.e., disponible) por la rutina de inicialización del driver de comunicación. La primera tarea que llame a CommSendCmd() adquirirá el semáforo y entonces procederá a enviar el comando y a esperar una respuesta. Si otra tarea intenta enviar un comando mientras el puerto está ocupado, esta segunda tarea será suspendida hasta que el semáforo sea liberado. La segunda tarea aparenta simplemente haber hecho una llamada a una función normal que no regresará hasta que la función haya realizado su trabajo. Cuando el semáforo es liberado por la primera tarea, la segunda tarea adquirirá el semáforo y entonces se le permitirá usar el puerto RS-232C.

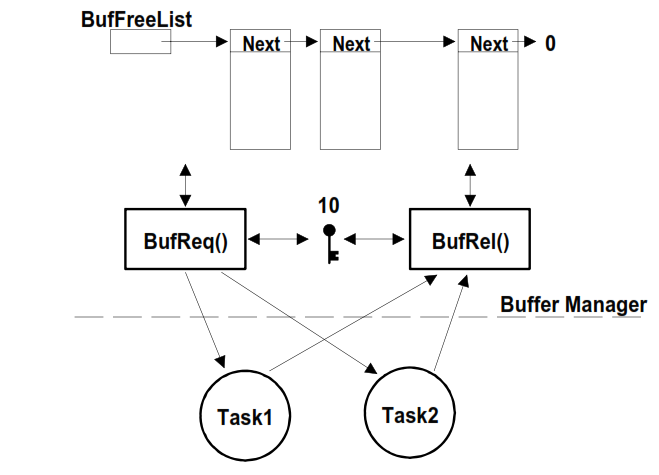
Cuando un recurso puede ser usado por más de una tarea al mismo tiempo se usa un semáforo contador. Por ejemplo, se usa un semáforo contador en la administración de un buffer pool como se muestra en la Figura 2-11. Supongamos que el almacén inicialmente contiene 10 buffers. Una tarea obtendría un buffer del administrador de buffer llamando a BufReq(). Cuando ya no necesita el buffer, la tarea devolvería el buffer al administrador de buffer llamando a BufRel(). El pseudocódigo para estas funciones se muestra en el listado 2.9.





El administrador de buffer satisfará las primeras 10 peticiones de buffer (dado que hay 10 llaves). Cuando ya se hayan usado todos los semáforos, una tarea que pida un buffer sería suspendida hasta que un semáforo esté disponible. Las interrupciones están deshabilitadas para ganar acceso exclusivo a la lista enlazada (esta operación es muy rápida). Cuando alguna tarea haya terminado de trabajar con el buffer que adquirió, esta tarea llama a BufRel() para devolver el buffer al administrador de buffers; el buffer es insertado en la lista enlazada antes de que el semáforo sea liberado. Encapsulando la interface al administrador de buffer en BufReq() y BufRel(), la parte del código que llama a estas funciones (por ejemplo una tarea) no necesita conocer los detalles de implementación reales.

Los semáforos son frecuentemente sobreutilizados. El uso de un semáforo para acceder a una simple variable compartida es un overkilling en la mayoría de las situaciones. El overhead involucrado en adquirir y liberar el semáforo puede consumir tiempo valioso. Usted puede hacer el trabajo con la misma eficiencia deshabilitando y habilitando interrupciones. Supongamos que dos tareas están compartiendo una variable entera de 32 bits. La primera tarea incrementa la variable mientras que la otra tarea la limpia. Si usted considera cuánto le toma a un procesador realizar cualquiera de las operaciones, usted se dará cuenta de que no necesita un semáforo para ganar acceso exclusivo a la variable. Cada tarea simplemente necesita deshabilitar las interrupciones antes de realizar su operación sobre la variable y habilitar las interrupciones cuando la operación esté completa. Sin embargo, se debe usar un semáforo si la variable es una variable de punto flotante y el microprocesador no soporta punto flotante en hardware. En este caso, el tiempo para procesar la variable de punto flotante podría afectar la latencia de interrupción si usted ha deshabilitado las interrupciones.





# Semáforos de exclusión mutua

Los semáforos de exclusión mutua (mutexes) son usados por las tareas para ganar acceso exclusivo a recursos. Los mutexes son semáforos binarios que tienen características adicionales más allá del mecanismo de semáforos normales proporcionados por %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document}.

Se usa un mutex en el código de aplicación para reducir el problema de inversión de prioridad (véase Sección 2.6 de [Labrosse, Jean J]). Ocurre una inversión de prioridad cuando una tarea de baja prioridad es dueña de un recurso que una tarea de prioridad alta necesita utilizar. Para reducir la inversión de prioridad, el kernel puede incrementar la prioridad de la tarea de prioridad más baja a la prioridad de la tarea de prioridad más alta hasta que la tarea de prioridad más baja ha terminado de usar el recurso.

Para implementar mutexes, un kernel de tiempo real necesita proporcionar la habilidad de soportar que varias tareas tengan la misma prioridad. Desafortunadamente, %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document} no permite que múltiples tareas tengan la misma prioridad. Sin embargo, hay una forma de resolver este problema. Qué tal si el mutex reserva una prioridad justo por arriba de la prioridad de la tarea de prioridad más alta que necesita acceder a él para permitir que la prioridad de la tarea de prioridad más baja sea elevada hasta este nivel de prioridad.

Usaremos un ejemplo para ilustrar como trabajan los mutexes de %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document}. El listado 8.1 muestra tres tareas que posdrían necesitar acceder a un recurso común. Para acceder al recurso, cada tarea debe hacer la operación pend sobre el mutex ResourceMutex. La tarea #1 tiene la prioridad más alta (10), la tarea #2 tiene una prioridad media (15), y la tarea #3, la prioridad más baja (20). Se reserva una prioridad no utilizada justo por arriba de la prioridad de tarea más alta (i.e., prioridad 9) como prioridad de herencia de herencia de prioridad (priority inheritance priority (PIP)).

Listado 8.1 Ejemplo de uso de un Mutex

/\*\* test\_mutex\_semaphore.c \*/

#include "includes.h"

OS\_EVENT \*ResourceMutex;

OS\_STK TaskPrio10Stk[1000];

OS\_STK TaskPrio15Stk[1000];

OS\_STK TaskPrio20Stk[1000];

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FUNCTION PROTOTYPES

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

void TaskPrio10 (void \*pdata);

void TaskPrio15 (void \*pdata);

void TaskPrio20 (void \*pdata);

void main()

{

INT8U err;

OSInit();

/\*Application Initialization\*/

OSMutexCreate(9, &err);

OSTaskCreate(TaskPrio10, (void\*)0, &TaskPrio10Stk[999], 10);

OSTaskCreate(TaskPrio15, (void\*)0, &TaskPrio15Stk[999], 15);

OSTaskCreate(TaskPrio20, (void\*)0, &TaskPrio20Stk[999], 20);

OSStart();

}/\*end main()\*/

void TaskPrio10 (void \*pdata)

{

INT8U err;

pdata = pdata;

while (1) {

/\*Application code\*/

OSMutexPend(ResourceMutex, 0, &err);

/\*Access common resource\*/

OSMutexPost(ResourceMutex);

/\*Application code\*/

}

}/\*end TaskPrio10()\*/

void TaskPrio15 (void \*pdata)

{

INT8U err;

pdata = pdata;

while (1) {

/\*Application code\*/

OSMutexPend(ResourceMutex, 0, &err);

/\*Access common resource\*/

OSMutexPost(ResourceMutex);

/\*Application code\*/

}

}/\*end TaskPrio15()\*/

void TaskPrio20 (void \*pdata)

{

INT8U err;

pdata = pdata;

while (1) {

/\*Application code\*/

OSMutexPend(ResourceMutex, 0, &err);

/\*Access common resource\*/

OSMutexPost(ResourceMutex);

/\*Application code\*/

}

}/\*end TaskPrio20()\*/

Listado 8.1 Ejemplo de uso de un Mutex (%FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document})

El código del listado 8.1 se puede construir utilizando el archivo make siguiente:

########## Path for uCOS-II core source files ############

UCOS\_SRC=/software/uCOS-II/source

########## Path for uCOS-II WIN32 port source files ############

UCOS\_PORT\_SRC=/software/uCOS-II/Ports/80x86/Linux/GCC/src

########## Name of Example source file ###########

EXAMPLE= test\_mutex\_semaphore

all:

gcc -D\_\_LINUX\_\_ -I$(UCOS\_SRC) -I$(UCOS\_PORT\_SRC) $(EXAMPLE).c $(UCOS\_SRC)/uCOS\_II.c $(UCOS\_PORT\_SRC)/pc.c $(UCOS\_PORT\_SRC)/os\_cpu\_c.c -o $(EXAMPLE)

clean:

rm -vf \*.o $(EXAMPLE)

rm -vf \*.bak

rm -vf \*~

Archivo makefile.lin

En el directorio de trabajo (digamos MutexSemaphore0) debemos tener los siguientes archivos:

.

|\_\_\_makefile.lin

|\_\_\_test\_mutex\_semaphore.c

|\_\_\_os\_cfg.h

La forma de utilizar este archivo make es tecleando (teniendo como directorio de trabajo el directorio donde se encuentra el código fuente y el archivo makefile.lin, el comando:

$ make –f makefile.lin

La salida deberá ser algo parecido a lo que se indica a continuación:

gcc -D\_\_LINUX\_\_ -I/software/uCOS-II/source

-I/software/uCOS-II/Ports/80x86/Linux/GCC/src test\_mutex\_semaphore.c /software/uCOS-II/source/uCOS\_II.c

/software/uCOS-II/Ports/80x86/Linux/GCC/src/pc.c

/software/uCOS-II/Ports/80x86/Linux/GCC/src/os\_cpu\_c.c

-o test\_mutex\_semaphore

Con ello se deberá tener la creación del archivo ejecutable test\_mutex\_semaphore

Cuando se ejecuta, este programa no imprime texto a la pantalla, únicamente nos sirve para probar que se tiene todos los archivos necesarios para la construcción de ejemplos para el port de uC/OS-II para Linux.

## Construcción del driver scull

scull: Simple character utility for loading localities

Descargue el archivo MutexSemaphore\_v00.tar del repositorio

https://github.com/sotrteacher/dirtywork/

desde el directorio

https://github.com/sotrteacher/dirtywork/PRACTICA\_4\_Exclusion\_Mutua/

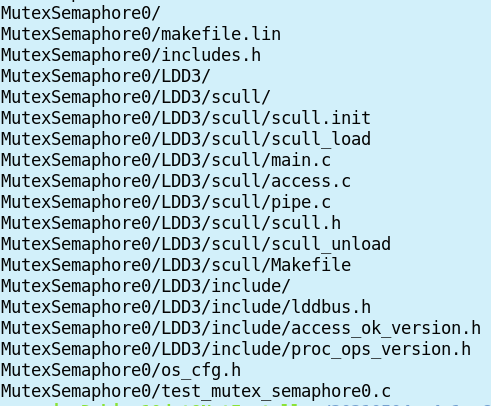
Coloque el archivo MutexSemaphore\_v00.tar en la distribución de %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document} utilizada en la práctica 3 Creación de Procesos, en el directorio

/software/uCOS-II/Ports/80x86/Linux/GCC/

$ cd /software/uCOS-II/Ports/80x86/Linux/GCC/

Extraer el contenido del archivo tar con el comando

$ tar xvf MutexSemaphore\_v00.tar



Ingresar al directorio MutexSemaphore0/

$ cd MutexSemaphore0/

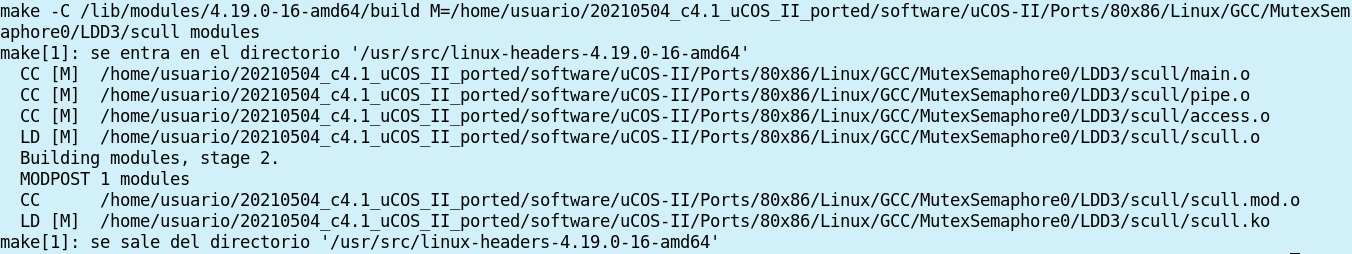
Después, ingrese al directorio LDD3/scull/

$ cd LDD3/scull/

Para construir el driver scull, solo ejecute el comando

$ make

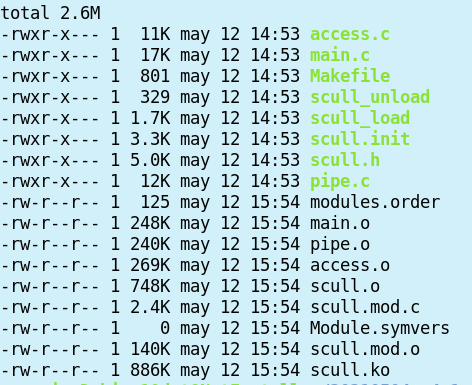
La salida debe ser similar a la siguiente:



Si no ocurrió algún error, la salida de

$ ls –hgotr

debe ser algo como lo siguiente:



$ sudo insmod skull.ko

## Construcción del programa MutexSemaphore versión 1

Programa que escribe al dispositivo /dev/scull sin utilizar semáforos de exclusión mutua.

Descargue el archivo MutexSemaphore\_v01.tar del repositorio

https://github.com/sotrteacher/dirtywork/

desde el directorio

https://github.com/sotrteacher/dirtywork/PRACTICA\_4\_Exclusion\_Mutua/

Coloque el archivo MutexSemaphore\_v01.tar en la distribución de %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document} utilizada en la práctica 3 Creación de Procesos, en el directorio

/software/uCOS-II/Ports/80x86/Linux/GCC/

$ cd /software/uCOS-II/Ports/80x86/Linux/GCC/

Extraer el contenido del archivo tar con el comando

$ tar xvf MutexSemaphore\_v01.tar



Ingresar al directorio MutexSemaphore1/

$ cd MutexSemaphore1/

Construya y cargue el driver scull como se indicó antes. Después, desde el directorio MutexSemaphore1/, construya el programa MutexSemaphore1 con el comando

$ make –f makefile.lin

Ejecute el programa MutexSemaphore

$ ./MutexSemaphore

y después observe los caracteres que quedaron almacenados en el dispositivo /dev/scull con el comando

$ cat /dev/scull

$ lsmod | grep scull (Comprobar si el driver/módulo está cargado en kernel linux )

$ sudo rmmod scull

## Construcción del programa MutexSemaphore versión 2

Programa de ejemplo que usa semáforos de exclusión mutua

Programa que escribe al dispositivo /dev/scull sin utilizar semáforos de exclusión mutua.

Descargue el archivo MutexSemaphore\_v02.tar del repositorio

https://github.com/sotrteacher/dirtywork/

desde el directorio

https://github.com/sotrteacher/dirtywork/PRACTICA\_4\_Exclusion\_Mutua/

Coloque el archivo MutexSemaphore\_v02.tar en la distribución de %FontSize=12
%TeXFontSize=12
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\mu\mbox{C/OS-II}
\]
\end{document} utilizada en la práctica 3 Creación de Procesos, en el directorio

/software/uCOS-II/Ports/80x86/Linux/GCC/

$ cd /software/uCOS-II/Ports/80x86/Linux/GCC/

Extraer el contenido del archivo tar con el comando

$ tar xvf MutexSemaphore\_v02.tar



Ingresar al directorio MutexSemaphore2/

$ cd MutexSemaphore2/

Construya y cargue el driver scull como se indicó antes. Después, desde el directorio MutexSemaphore1/, construya el programa MutexSemaphore1 con el comando

$ make –f makefile.lin

Ejecute el programa MutexSemaphore

$ ./MutexSemaphore

y después observe los caracteres que quedaron almacenados en el dispositivo /dev/scull con el comando

$ cat /dev/scull

Escriba sus conclusiones en cuanto a las salidas de los caracteres que almacenan en el dispositivo /dev/scull las versiones 1 y 2 de los programas MutexSemaphore.

REFERENCIAS

[Corbet] Corbet, J., & Rubini, A., & Kroah-Hartman, G. (2005). Linux Device Drivers (3/a edición). O’Reilly Media.

[Labrosse, Jean J] Labrose, J.J. (2002). Micro C/OS-II The Real Time Kernel (2/a edición). CRC Press.

Algunos enlaces para descargar los archivos pdf de los textos mencionados en las referencias.

FLORES JUAREZ DAVID ALEJANDRO

15:16

<https://bootlin.com/doc/books/ldd3.pdf>

Tú

15:16

Corbet, J., & Rubini, A., & Kroah-Hartman, G. (2005). Linux Device Drivers (3/a edición). O’Reilly Media.

Gabriel Cabrera Sánchez

15:17

<https://bootlin.com/doc/books/ldd3.pdf>

2021-05-12

Carlos Esquivel Dávila

14:39

<https://lwn.net/Kernel/LDD3/> El libro :)