Proyecto de Programación: Threads a nivel de usuario

Este proyecto de programación consiste en implementar hilos a nivel de usuario (user-level threads) para simular las operaciones de hilos en Linux. El proyecto consiste de 5 partes (partes 0 a 4). La parte 0, presenta los conceptos de programación concurrente necesarios para la comprensión de las partes 1 a la 4. La parte 1 presenta el código base del proyecto para tener un punto de partida. El código base implementa un sistema multitarea (multitasking system), el cual soporta ejecuciones independientes de tareas dentro de un proceso Linux. Este es el mismo que el sistema multitasking MT presentado en el archivo 01\_Practica\_Sincronizacion\_y\_Administracion\_de\_Procesos.docx, pero adaptado al ambiente de hilos a nivel de usuario (user-level threads). La parte 2 es extender el código base para implementar soporte para la operación unión de tareas (task join). La parte 3 es extender el código base para soportar operaciones mutex. La parte 4 es implementar semáforos contadores para dar soporte a cooperación de tareas y demostrar el uso de los semáforos en el sistema multitarea.

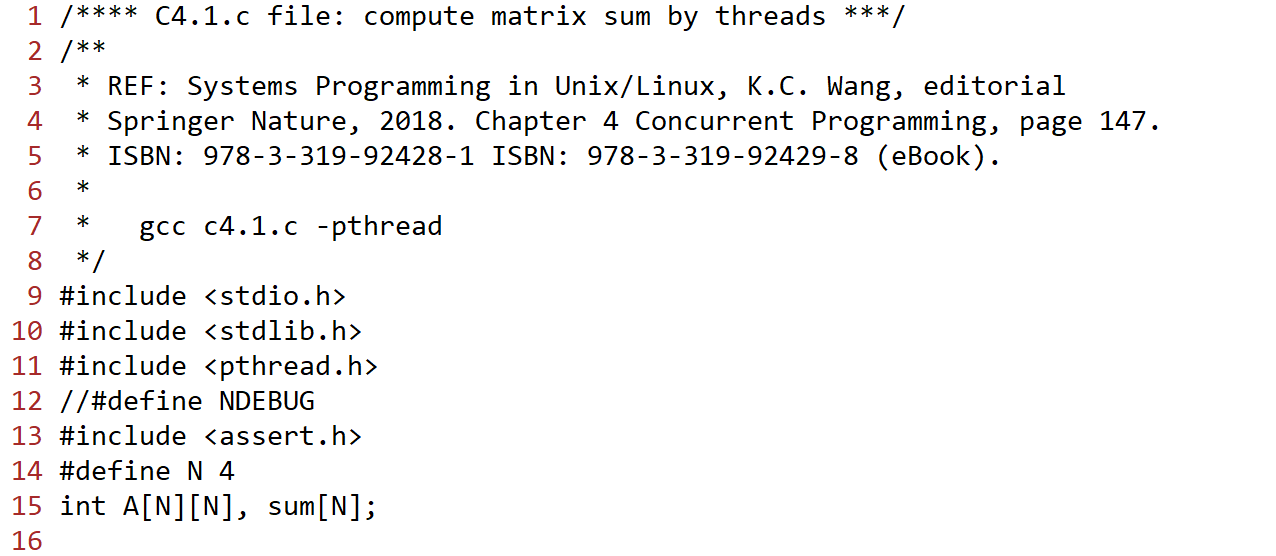
# Parte 0

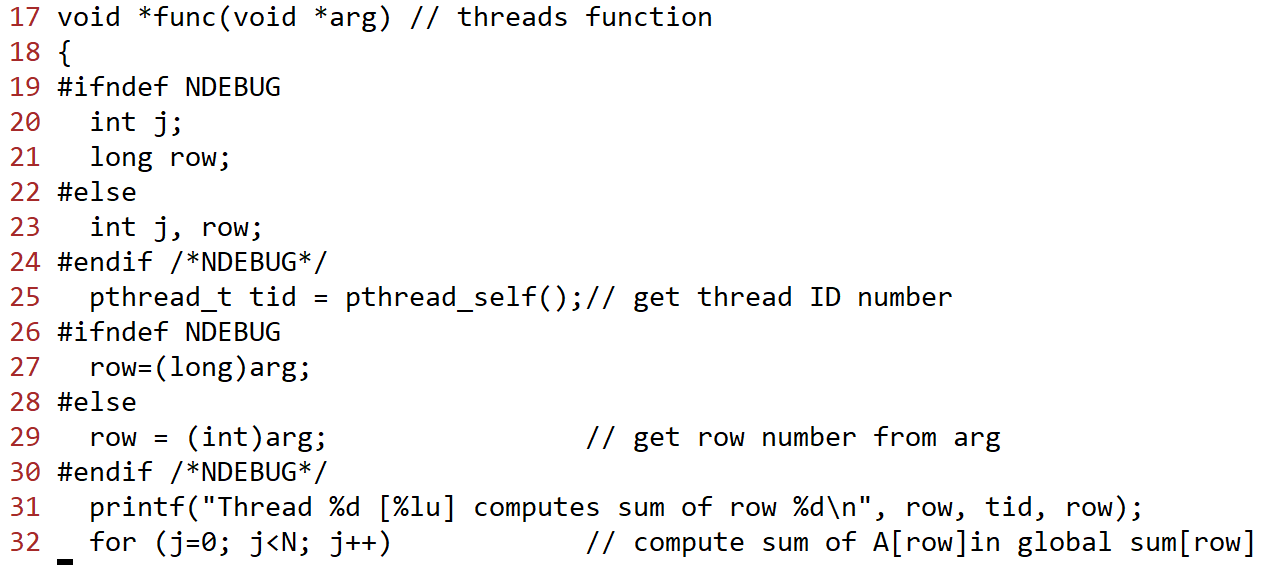
## Ejemplo de un programa que usa threads

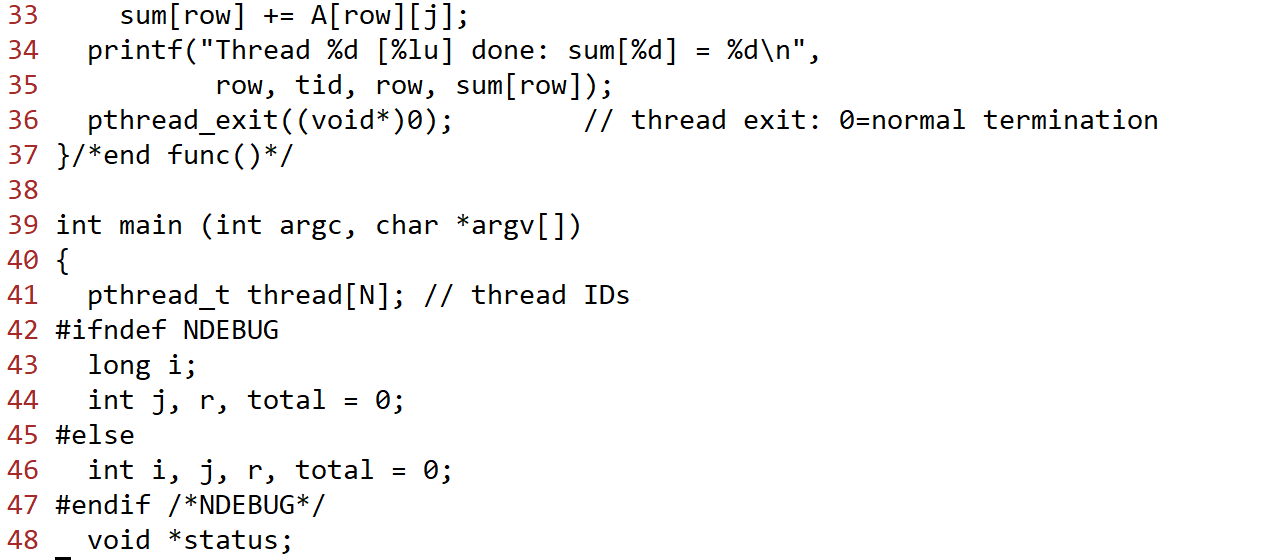
### Suma de los elementos de una matriz usando threads

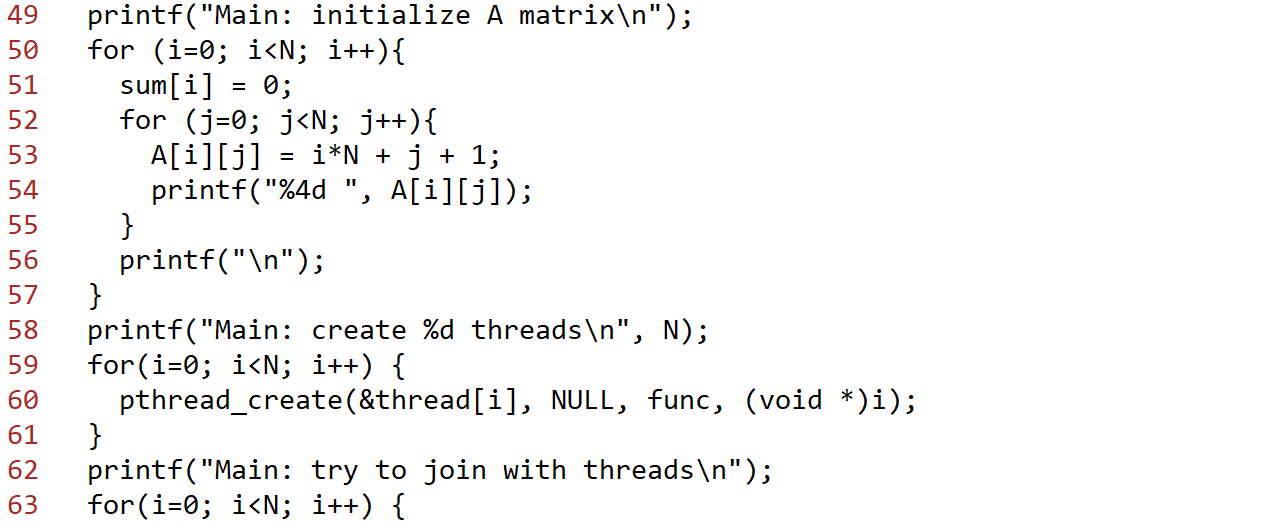
Suponga que queremos calcular la suma de todos los elementos de una matriz de %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
N\times N
\]
\end{document} enteros. Una forma de resolver el problema es con un algoritmo concurrente que usa hilos. En este ejemplo, el primer hilo genera una matriz de %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
N\times N
\]
\end{document} enteros. Entonces crea N hilos de trabajo, pasando como parámetro un número de fila único a cada hilo de trabajo, y espera a que todos los hilos de trabajo terminen. Cada hilo de trabajo calcula la suma parcial de una fila distinta de la matriz, y deposita la suma parcial en un elemento correspondiente de un arreglo global int sum[N]. Cuando todos los hilos de trabajo han terminado, el hilo principal continúa. El hilo principal entonces calcula la suma total sumando las sumas parciales generadas por los hilos de trabajo. A continuación se muestra el código C completo del programa c4.1.c. Sobre GNU Linux, el programa puede ser construido con













# Sincronización de hilos

Dado que los hilos se ejecutan en el mismo espacio de direcciones de un proceso, estos comparten todas las variables globales y estructuras de datos en el mismo espacio de direcciones. Cuando varios hilos tratan de modificar la misma variable compartida o estructura de datos, si el resultado depende del orden de ejecución de los hilos, a esto se le conoce como una **condición de carrera**. En los programas concurrentes, las condiciones de carrera se deben evitar (se debe hacer que no existan). De no hacer esto, los resultados podrían ser inconsistentes. Además de la operación unión, los hilos que se ejecutan concurrentemente frecuentemente necesitan cooperar unos con otros. Para prevenir condiciones de carrera, así como soportar cooperación de hilos, los hilos necesitan sincronización. En general, la sincronización se refiere a los mecanismos y reglas usadas para asegurar la integridad de los objetos dato compartidos y la coordinación de las entidades que se ejecutan concurrentemente. Esto puede ser aplicado ya sea a procesos en modo kernel o a hilos en modo usuario. En los párrafos siguientes, se discute el problema específico de sincronización de hilos en Pthreads.

## Candados mutex (Mutex Locks)

El tipo de herramienta de sincronización más simple es un lock, el cual le permite a una entidad de ejecución continuar solo si ésta posee el lock. En Pthreads, los locks son llamados **mutex**, los cual significa exclusión mutua (**Mutual Exclusion**). Las variables mutex se declaran con el tipo de dato **pthread\_mutex\_t**, y deben ser inicializados antes de ser usados. Hay dos formas de incializar una variable mutex.

1. Estáticamente como en



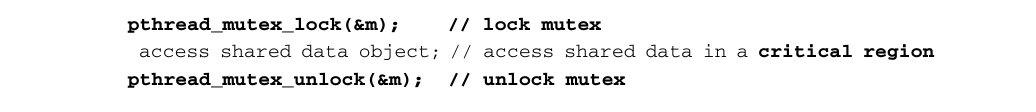
lo cual define una variable mutex m y la inicializa con atributos por defecto.

1. Dinámicamente con la función **pthread\_mutex\_init()**, la cual permite establecer los atributos del mutex con un parámetro attr, como en



Como es usual, el parámetro attr puede ser establecido a NULL para usar atributos por defecto.

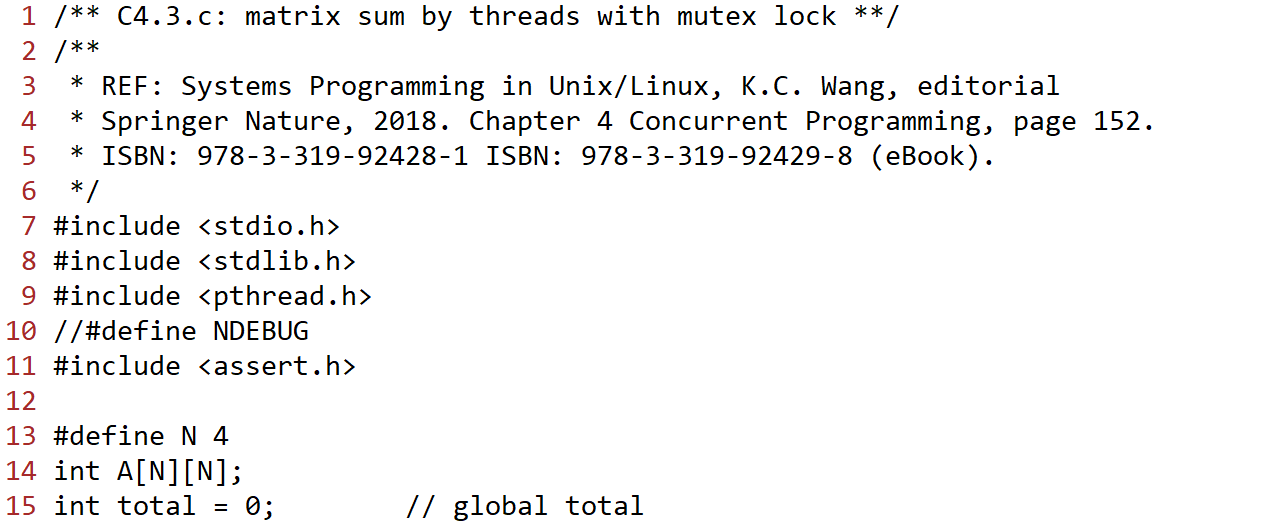
Después de la inicialización, las variables mutex pueden ser usadas por los hilos para proteger objetos dato compartidos. Un uso típico de un mutex es como sigue. Un hilo primero crea un mutex y lo inicializa una vez. Un mutex recién creado está en el estado no bloqueado y sin un dueño. Cada hilo trata de acceder al objeto dato compartido con

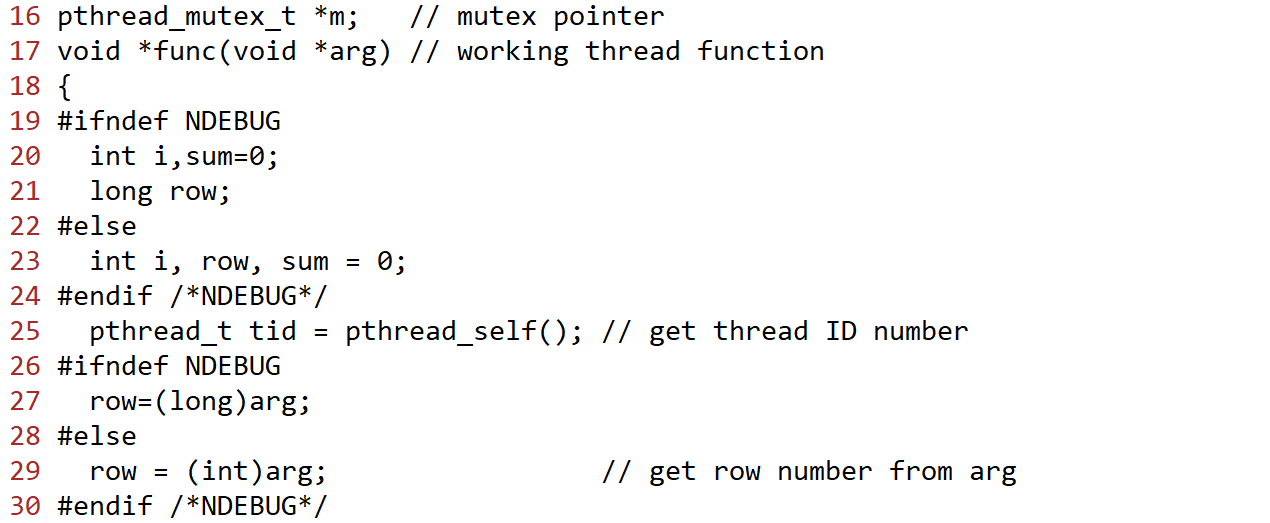


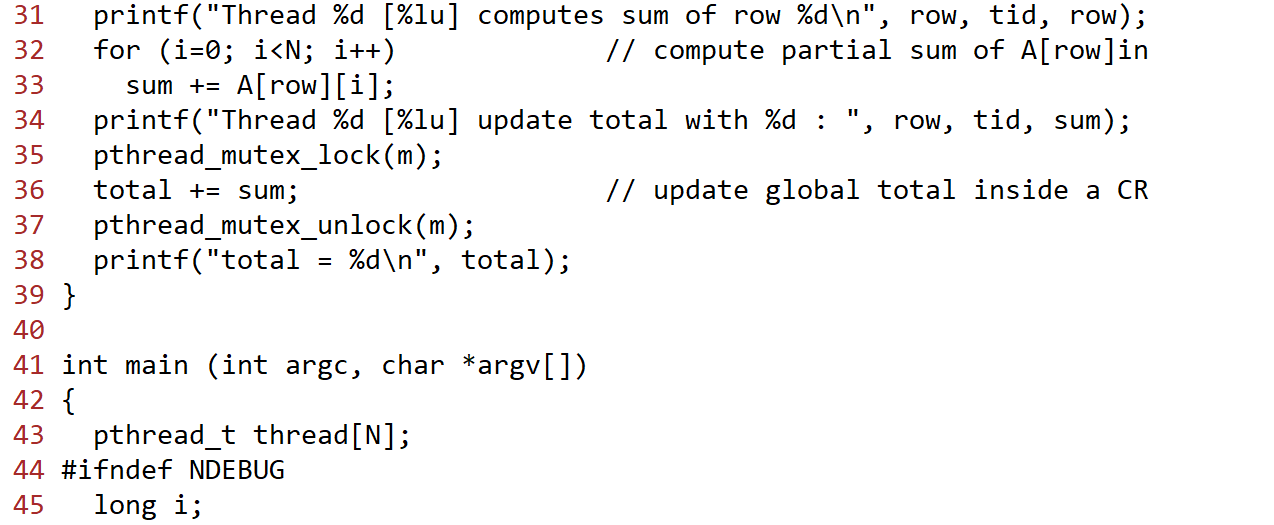
Cuando un hilo ejecuta pthread\_mutex\_lock(&m), si el mutex está desbloqueado, el hilo bloquea el mutex, se convierte en el dueño del mutex y continúa su ejecución. En caso contrario, el hilo se bloquea y espera en la cola de espera del mutex. Solamente el hilo que ha conseguido bloquear el mutex puede acceder al objeto dato compartido. Una secuencia de ejecuciones la cual puede solamente ser realizada por una entidad de ejecución a la vez es comúnmente conocida como una **Región Crítica** (CR). En Pthreads, los mutexes son usados como candados para proteger Regiones Críticas para asegurar que a lo más un hilo puede estar dentro de una CR en cualquier momento. Cuando el hilo termina de ocupar el objeto dato compartido, sale de la CR llamando a pthread\_mutex\_unlock(&m) para desbloquear el mutex. Un mutex bloqueado solo puede ser desbloqueado por el dueño actual. Cuando un hilo desbloquea un mutex, si no hay hilos bloqueados en la cola de espera del mutex, el hilo desbloquea el mutex y el mutex se queda sin dueño. En caso contrario, se desbloquea un hilo que estaba en la cola de espera del mutex, el proceso desbloqueado se convierte en el nuevo dueño del mutex, y el mutex permanece bloqueado. Cuando todos los hilos han terminado, el mutex puede ser destruido si fue alojado dinámicamente. Se demuestra la sincronización de hilos usando mutex lock con un ejemplo.

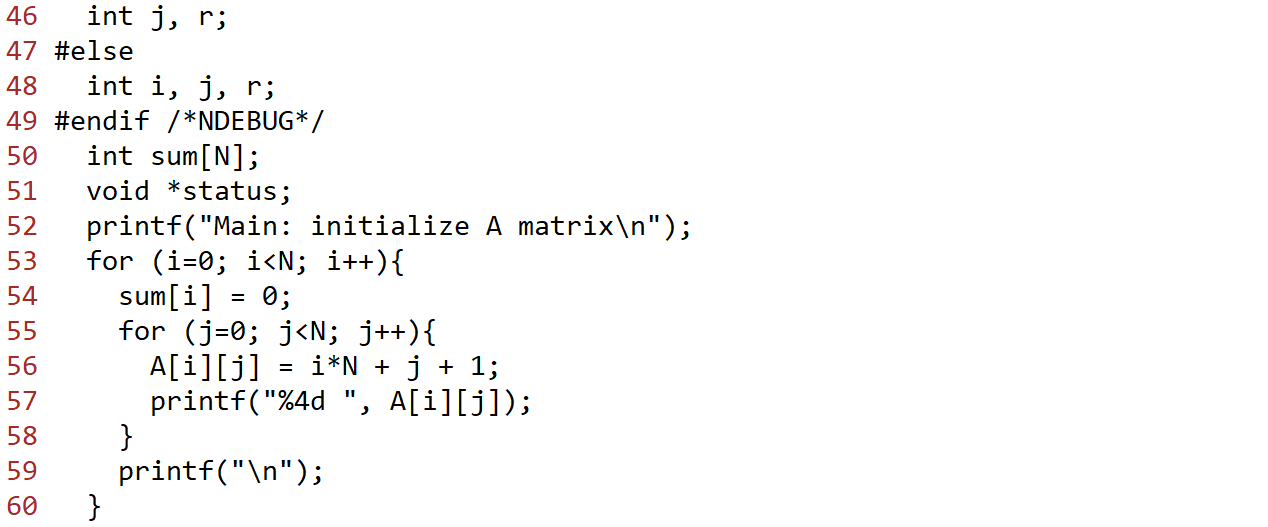
## Ejemplo: Uso de Mutex Lock

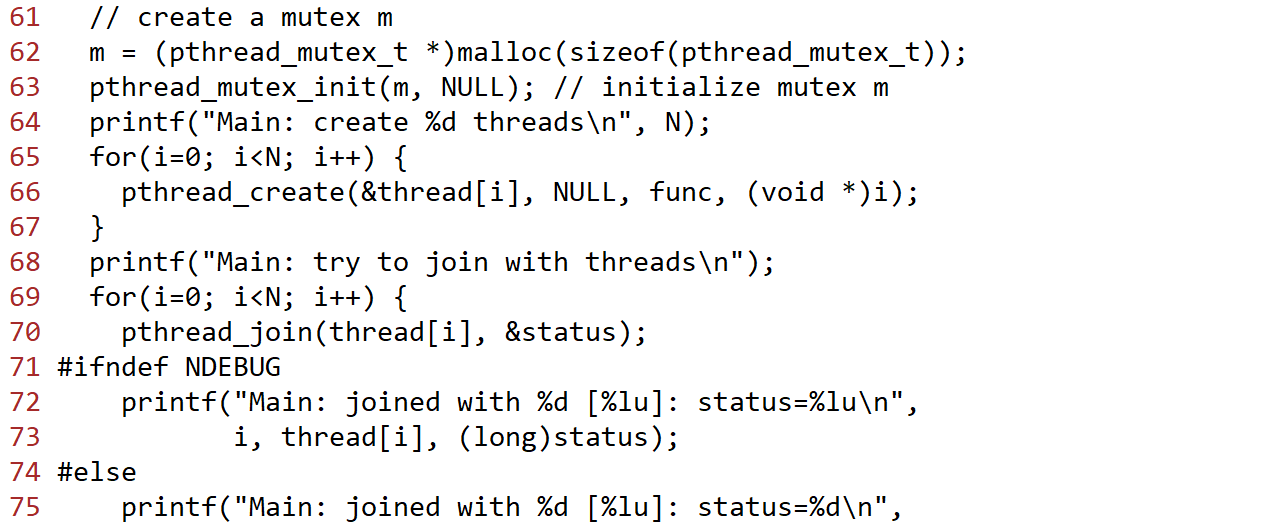
Este ejemplo es una versión modificada del ejemplo anterior. Como antes, usaremos N hilos de trabajo para calcular la suma de todos los elementos de una matriz de %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
N\times N
\]
\end{document} enteros. Cada hilo de trabajo calcula la suma parcial de una fila de la matriz. En lugar de depositar las sumas parciales en un arreglo global sum[ ], en este ejemplo, cada hilo de trabajo trata de actualizar una variable global, total, sumando su suma parcial a esta. Dado que todos los hilos de trabajo tratan de actualizar la misma variable global, estos hilos deben ser sincronizados para evitar condiciones de carrera. Esto se puede lograr con un mutex lock, el cual asegura que solamente un hilo de trabajo puede actualizar la variable total a la vez en una Región Crítica. El siguiente es el programa de ejemplo c4.3.c

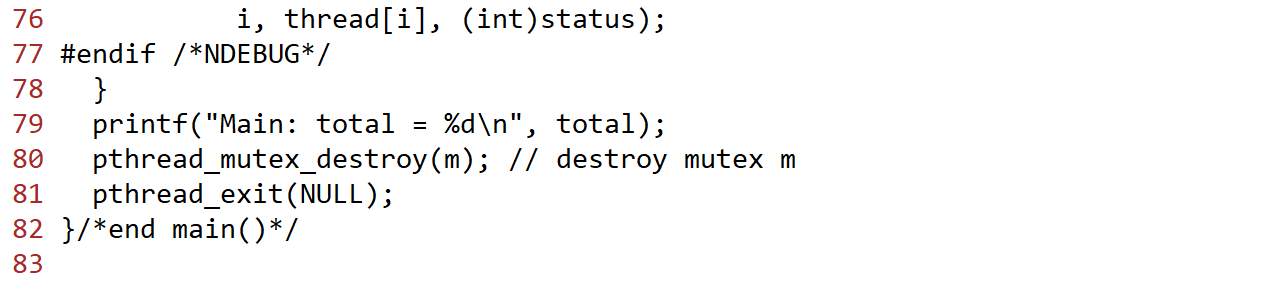




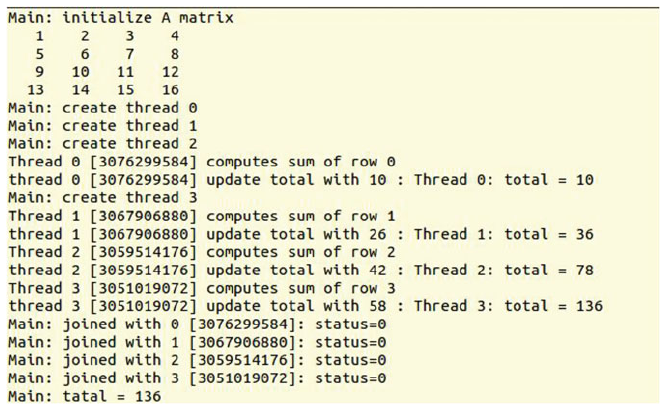






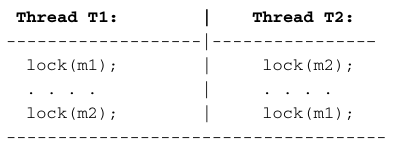


La siguiente figura muestra una salida de ejemplo del programa anterior

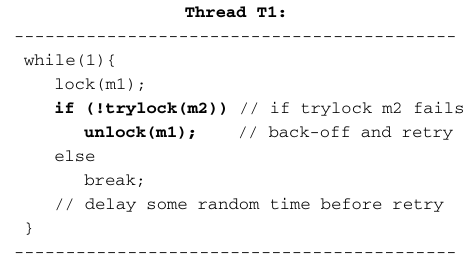


# Prevención de Deadlock

Los mutex usan el protocolo locking. Si un hilo no puede adquirir un mutex lock, el hilo se bloquea, esperando a que el mutex sea desbloqueado para poder continuar. En cualquier protocolo locking, el uso descuidado de los locks puede conducir a problemas. El problema más conocido es el de deadlock. **Deadlock** es una condición, en la cual varias entidades de ejecución mutuamente esperan que alguna otra de ellas haga algo, y de esa forma, ninguna de ellas puede continuar. Para ilustrar esto, suponga que el hilo T1 ha adquirido un mutex lock m1 y trata de bloquear otro mutex m2. Otro hilo T2 ha adquirido el mutex lock m2 y trata de bloquear el mutex m1, como se muestra en el siguiente diagrama.



En este caso, T1 y T2 esperarán mutuamente uno por el otro para siempre, así que ellos están en un deadlock debido a las peticiones de bloqueo cruzadas. Al igual que sucede con las condiciones de carrera, los deadlocks no deben existir en los programas concurrentes. Hay algunas formas de tratar con los posibles deadlocks, las cuales incluyen **prevención de deadlocks,** detección de deadlocks y recuperación. En los sistemas reales, la única forma práctica es la prevención de deadlocks, la cual trata de prevenir que ocurran los deadlocks cuando se están diseñando algoritmos paralelos. Una forma simple de prevenir los deadlocks es ordenar los mutex y asegurarse de que cada hilo pide los mutex lock en una sola dirección, de tal manera que no hay ciclos en las secuencias de petición de mutex locks. Sin embargo, podría no ser posible diseñar cada algoritmo paralelo con solamente peticiones de locking unidireccionales. En tales casos, la función de locking condicional, **pthread\_mutex\_trylock()**, puede ser usada para prevenir deadlocks. La función trylock() regresa inmediatamente con un error si el mutex ya está bloqueado (locked). En ese caso, el hilo que está llamando podría retroceder liberando algunos de los locks que hasta ese momento ya tenga adquiridos, permitiendo a otros hilos continuar. En el ejemplo de bloqueo cruzado de arriba, podríamos rediseñar uno de los hilos, por ejemplo, T1, como sigue, lo cual usa bloqueo (locking) condicional y retrocede para prevenir el deadlock.



## Variables de condición

Los mutex son usados solamente como locks, lo cual asegura que los hilos acceden a objetos dato compartidos de forma exclusiva en Regiones Críticas. Las variables de condición proporcionan medios para cooperación de hilos. Las variables de condición siempre se usan junto con los mutex lock. Esto no es una sorpresa porque la exclusión mutua es la base de todos los mecanismos de sincronización. En Pthreads, las variables de condición son declaradas con el tipo de dato **pthread\_cond\_t**, y deben ser inicializadas antes de usarlas. Como los mutex, las variables de condición también pueden ser inicializadas en una de dos formas.

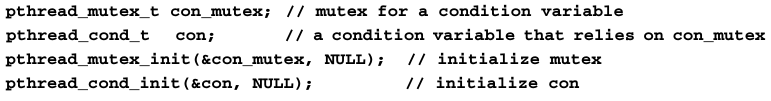
1. Estáticamente, cuando es declarada, como en



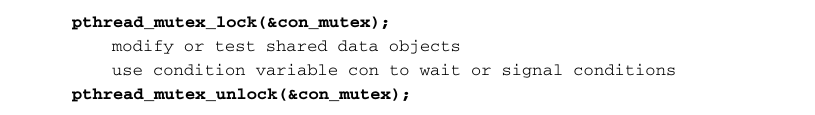
Lo cual define una variable de condición, con, y la inicializa con atributos por defecto.

1. Dinámicamente, con la función **pthread\_cond\_init()**, la cual permite establecer una variable de condición con un parámetro attr. Por simplicidad, suele usarse un parámetro attr NULL para atributos por defecto.

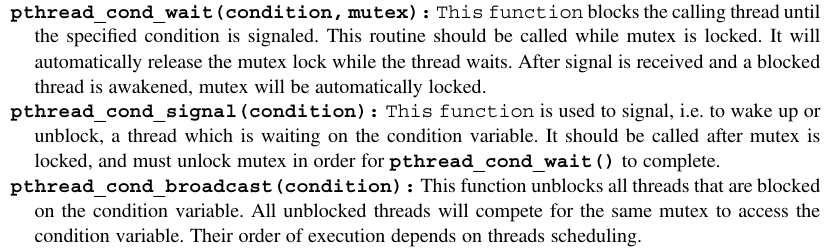
Los siguientes segmentos de código muestran cómo definir una variable de condición con un mutex lock asociado.



Cuando se usa una variable de condición, un hilo debe adquirir el mutex lock asociado primero. Entonces realiza operaciones dentro de la región crítica del mutex lock y libera el mutex lock, como en



Dentro de la CR del mutex lock, los hilos podrían usar variables de condición para cooperar con otro a través de las siguientes funciones.



Demostramos la cooperación de hilos usando variables de condición con un ejemplo.

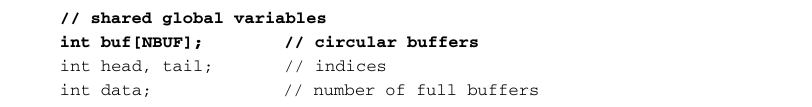
# El problema productor-consumidor

## Ejemplo: Productor-consumidor

En este ejemplo, implementaremos una versión simplificada del **problema productor-consumidor**, el cual también es conocido como el problema de **buffer acotado**, usando hilos y variables de condición. El problema productor-consumidor es usualmente definido con procesos como entidades de ejecución, las cuales pueden ser considerados como hilos en el contexto actual. El problema es definido como sigue.

Un conjunto de procesos productores y consumidores comparten un número finito de buffers. Cada buffer contiene un elemento único en un momento dado. Inicialmente, todos los buffers están vacíos. Cuando un productor pone un elemento en un buffer vacío, el buffer se convierte en un buffer lleno. Cuando un consumidor obtiene un elemento de un buffer lleno, el buffer se convierte en un buffer vacío, etc. Un productor debe esperar si no hay buffers vacíos. Similarmente, un consumidor debe esperar si no hay buffers llenos. Además, a los procesos que están esperando se les debe permitir continuar cuando el evento por el que esperan ocurre.

En el programa de ejemplo, supondremos que cada buffer contiene un valor entero. Las variables globales compartidas son definidas como

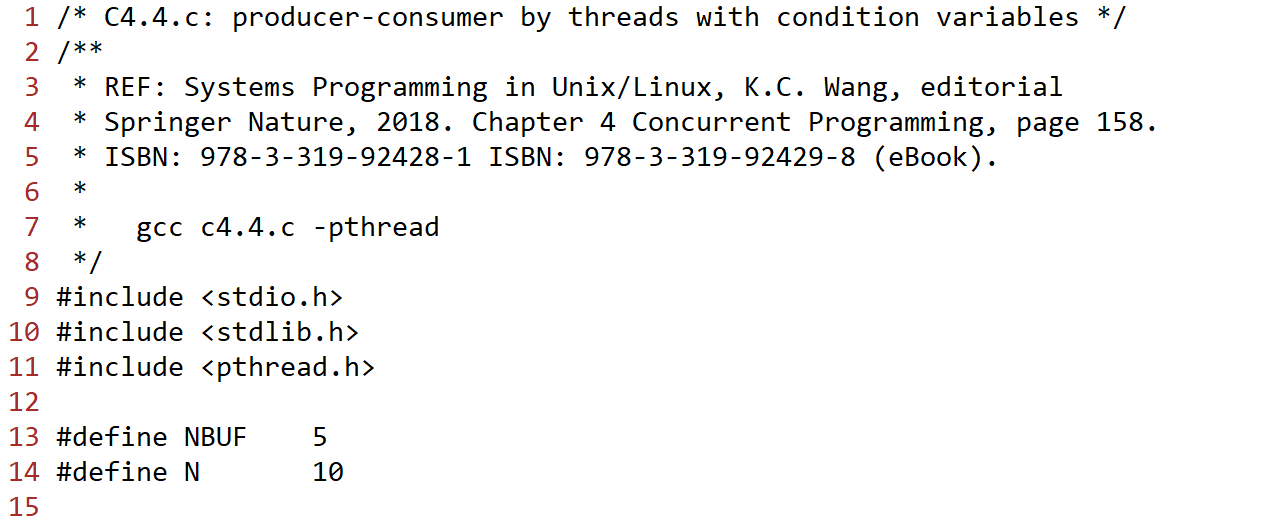


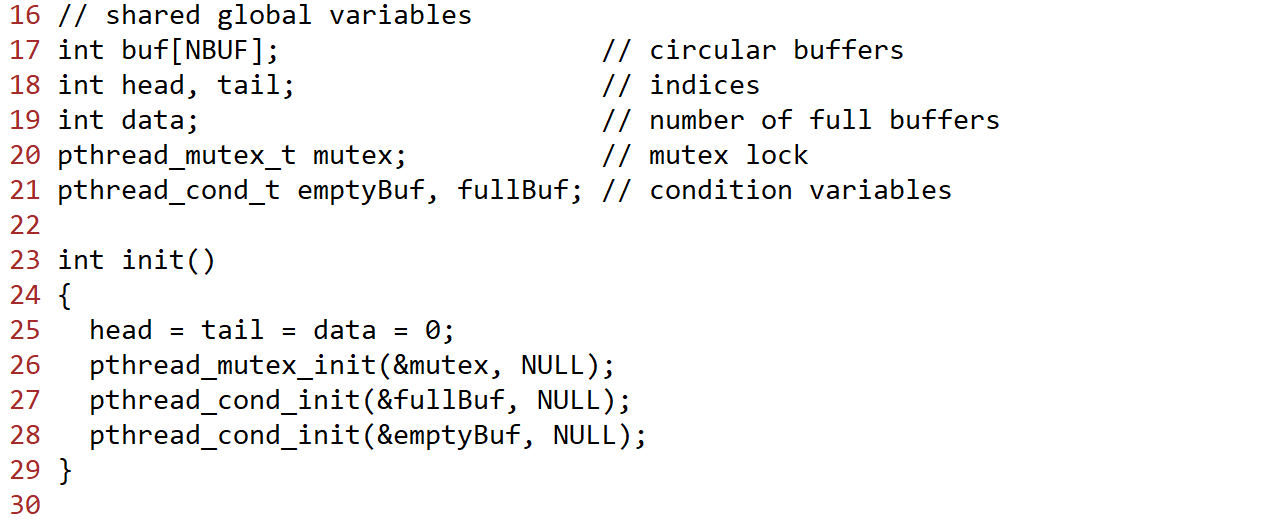
Los buffers son usados como un conjunto de buffers circulares. La variable índice **head** es para poner un elemento en un buffer vacío, y la variable índice **tail** es para tomar un elemento de un buffer lleno. La variable data es el número de buffers llenos. Para soportar cooperación entre productor y consumidor, definimos un mutex y dos variables de condición.

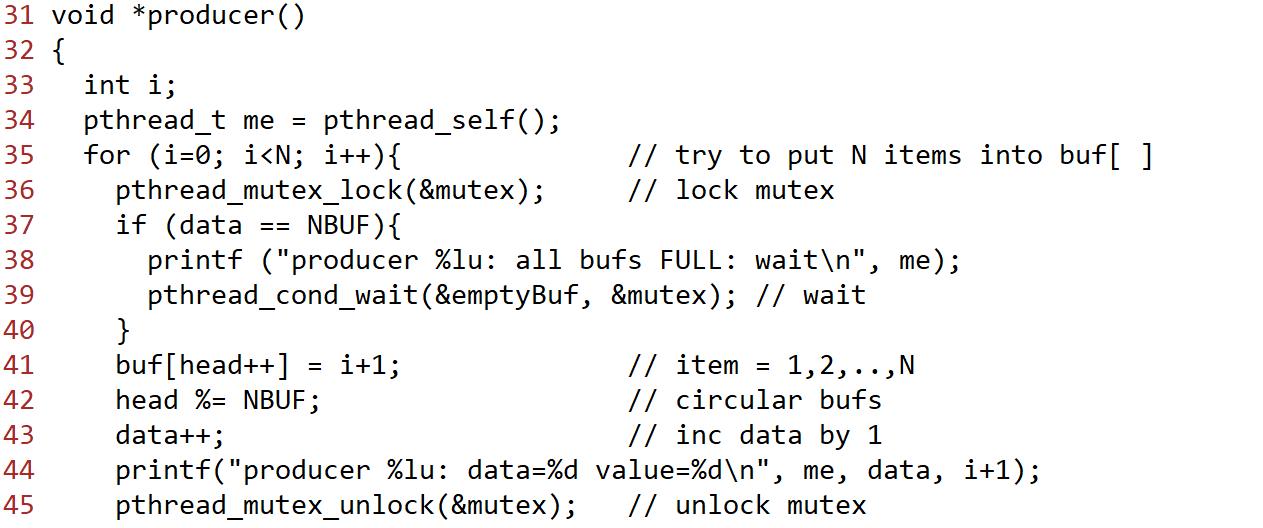


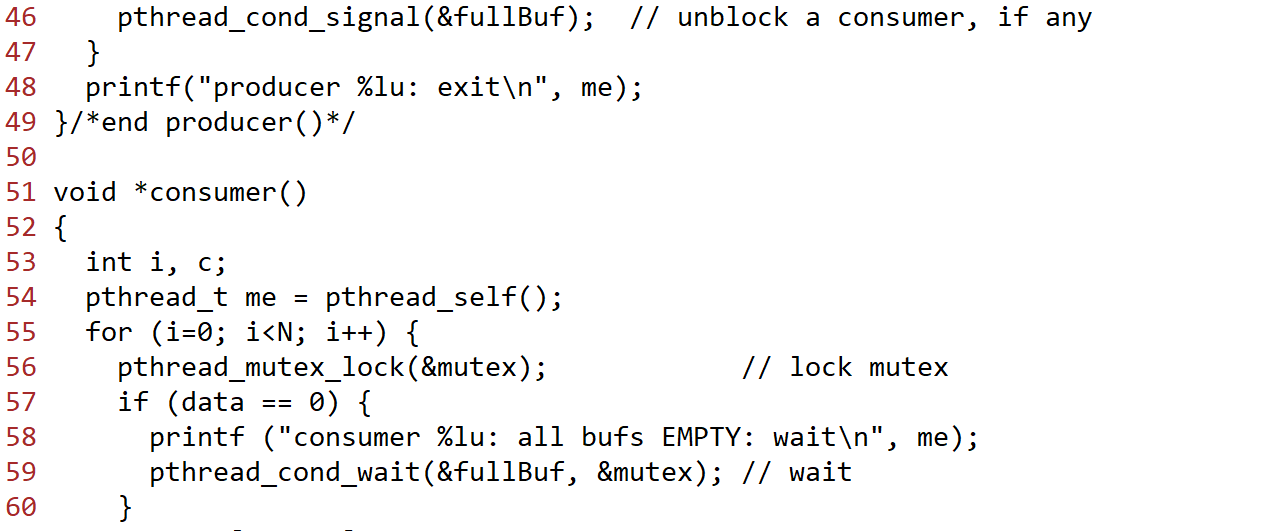
donde emptyBuf representa la condición de que hay algunos buffers vacíos. Y fullBuf representa la condición de que hay algunos buffers llenos. Cuando un productor encuentra que no hay buffers vacíos, éste espera por la variable de condición emptyBuf, la cual es señalada siempre que un consumidor ha consumido un buffer lleno. Similarmente, cuando un consumidor encuentra que no hay buffers llenos, éste espera por la variable de condición fullBuf, la cual es señalada siempre que un productor pone un elemento en un buffer vacío.

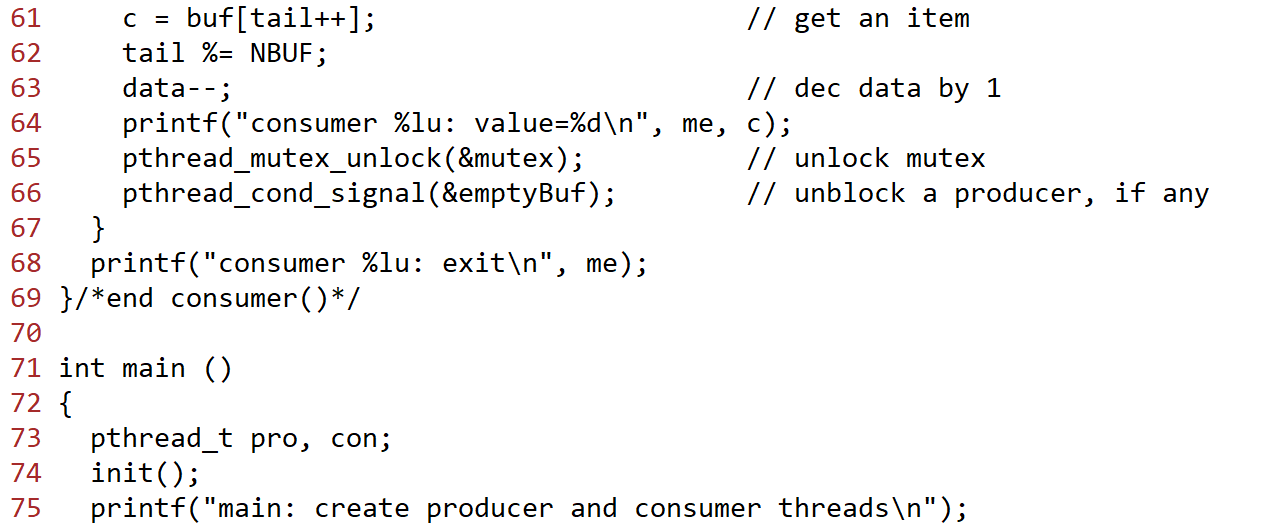
El programa comienza con un hilo principal, el cual inicializa las variables de control del buffer y las variables de condición. Después de la inicialización, el hilo principal crea un hilo productor y un hilo consumidor y espera a que esos hilos terminen utilizando la operación join. El tamaño del buffer es establecido a NBUF=5 pero el productor trata de colocar N=10 elementos en el área de buffer, lo cual causaría que el productor tenga que esperar si en un momento dado todos los buffers están llenos. Similarmente, el consumidor trata de obtener N=10 elementos de los buffers, lo cual causaría que el consumidor tenga que esperar si en un momento dado todos los buffers están vacíos. En cualquier caso, un hilo que está esperando recibirá una señal del otro hilo cuando las condiciones esperadas se cumplan. Entonces, los dos hilos cooperan uno con otro a través de las variables de condición. A continuación, se muestra el código del ejemplo c4.4.c, el cual implementa una versión simplificada del problema productor-consumidor con solamente un productor y un consumidor.

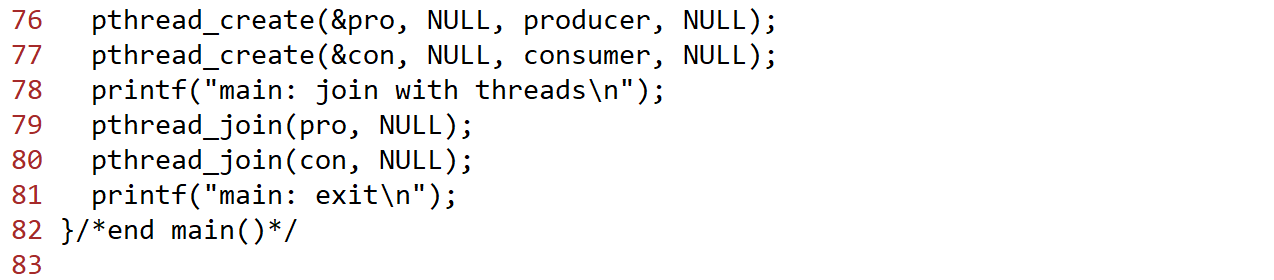




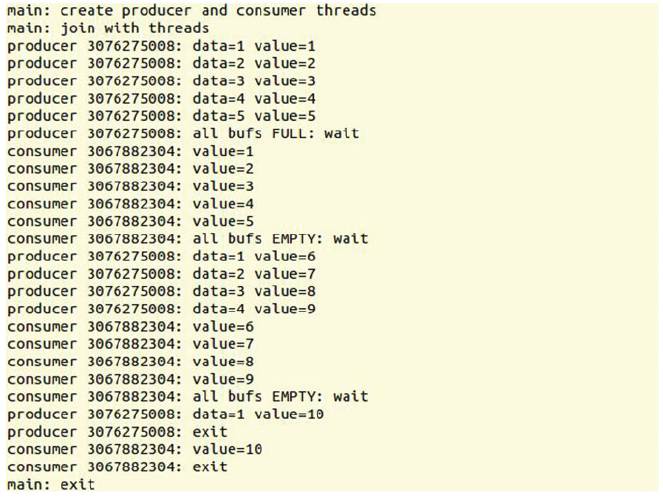






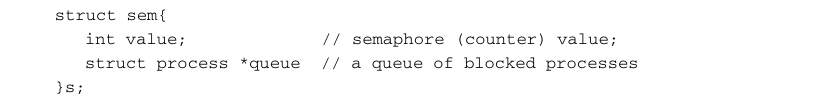


La figura siguiente muestra un ejemplo de la salida del ejemplo programa productor-consumidor.

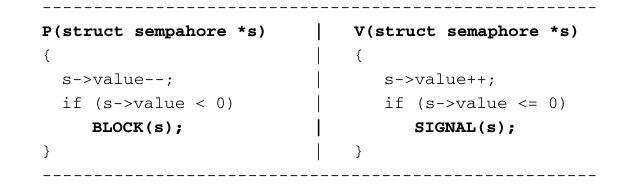


## Semáforos

Los semáforos son mecanismos generales para sincronización de procesos. Un semáforo contador es una estructura de datos

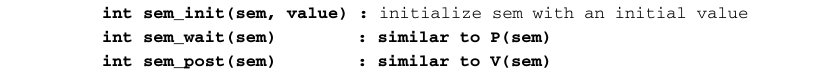


Antes de poder usarlo, un semáforo debe ser inicializado con un valor inicial y una cola de espera vacía. Independientemente de la plataforma hardware, esto es, si sobre un sistema con una cpu única, o sistemas de multiprocesamiento, la implementación de bajo nivel de los semáforos garantiza que cada semáforo puede solamente ser operado por una entidad de ejecución a la vez y las operaciones sobre los semáforos son **atómicas** (indivisibles) o **primitivas** desde el punto de vista de las entidades de ejecución. Nos concentraremos en las operaciones de alto nivel sobre los semáforos y su uso como mecanismo de sincronización de procesos. Las operaciones más importantes y mejor conocidas de los semáforos son P y V, las cuales están definidas como



donde BLOCK(s) bloquea al proceso que llama y lo coloca en la cola de espera del semáforo, y SIGNAL(s) desbloquea un proceso de los que están en la cola de espera del semáforo.

Los semáforos no son parte del estándar Pthreads. Sin embargo, la mayoría de las implementaciones de Pthread, ahora soportan semáforos de POSIX 1003.1b. Los semáforos POSIX incluyen las siguientes funciones



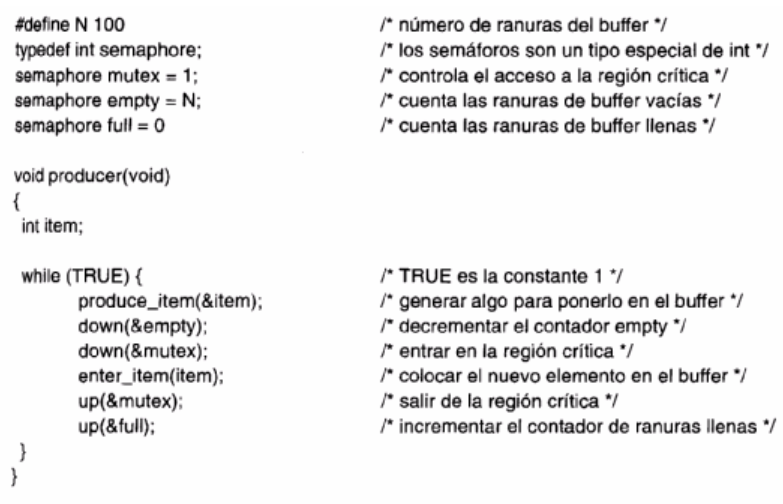
La principal diferencia entre los semáforos y las variables de condición es que el semáforo incluye un contador, manipula el contador, prueba o testea el valor del contador para tomar decisiones, etc. todo en la Región Crítica de operaciones atómicas o primitivas, mientras que en las variables de condición se requiere un mutex lock específico para forzar la Región Crítica. En Pthreads, los mutex son estrictamente para bloquear (locking) y las variables de condición son para cooperación de hilos. En contraste, los semáforos contadores con valor inicial 1 pueden ser usados como locks. Los semáforos con otros valores iniciales pueden ser usados para cooperación. Por lo tanto, los semáforos son más generales y flexibles que las variables de condición. El siguiente ejemplo ilustra las ventajas de los semáforos sobre las variables de condición.

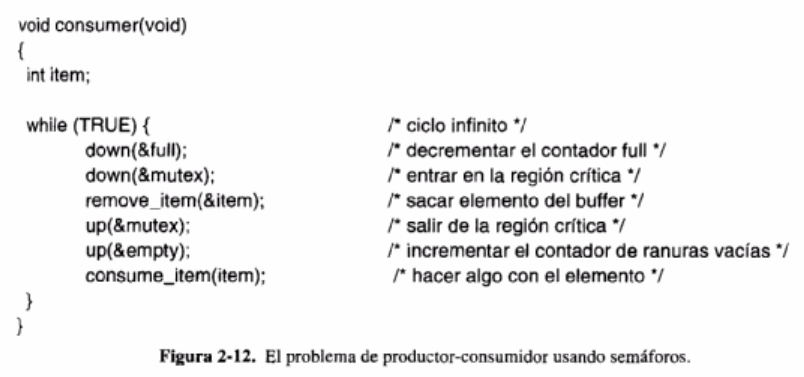
## Ejemplo: Solución del problema productor-consumidor usando semáforos

El problema productor-consumidor puede ser resuelto más eficientemente usando semáforos. En este ejemplo, **empty=NBUF** y **full=0** son semáforos para que productores y consumidores cooperen uno con otro, y **mutex=1** es un semáforo lock para que los procesos accedan a los buffers compartidos uno a la vez en una Región Crítica. Para un ejemplo de solución del problema productor-consumidor con semáforos, puede consultarse el libro de Andrew Tanenbaum, Sistemas Operativos, Diseño e Implementación.

### Resolución del problema de productor-consumidor usando semáforos

Los semáforos resuelven el problema de la señal de despertar perdida, como se muestra en la siguiente figura. Para que los semáforos funcionen correctamente es indispensable que se implementen de modo que sus operaciones sean indivisibles. El método normal consiste en implementar las operaciones UP y DOWN (sem\_post() y sem\_wait()) como llamadas al sistema, para que el sistema operativo inhabilite brevemente todas las interrupciones mientras prueba el semáforo, lo actualiza y pone el proceso a dormir, si es necesario. Todas estas acciones requieren sólo unas cuantas instrucciones, así que la inhabilitación de las interrupciones no tiene consecuencias adversas. Si se están usando múltiples CPU, cada semáforo debe estar protegido con una variable de candado, usando la instrucción TSL (una instrucción de ensamblador de los microprocesadores intel) para asegurarse que sólo una CPU a la vez examine el semáforo. Cerciórese de entender que el empleo de TSL para evitar que varias CPU accedan al semáforo al mismo tiempo es muy diferente de la espera activa del productor o el consumidor cuando esperan que el otro vacíe o llene el buffer. La operación del semáforo sólo toma unos cuantos microsegundos, mientras que, si se usa espera activa, el productor o el consumidor podrían tardar un tiempo arbitrariamente largo.





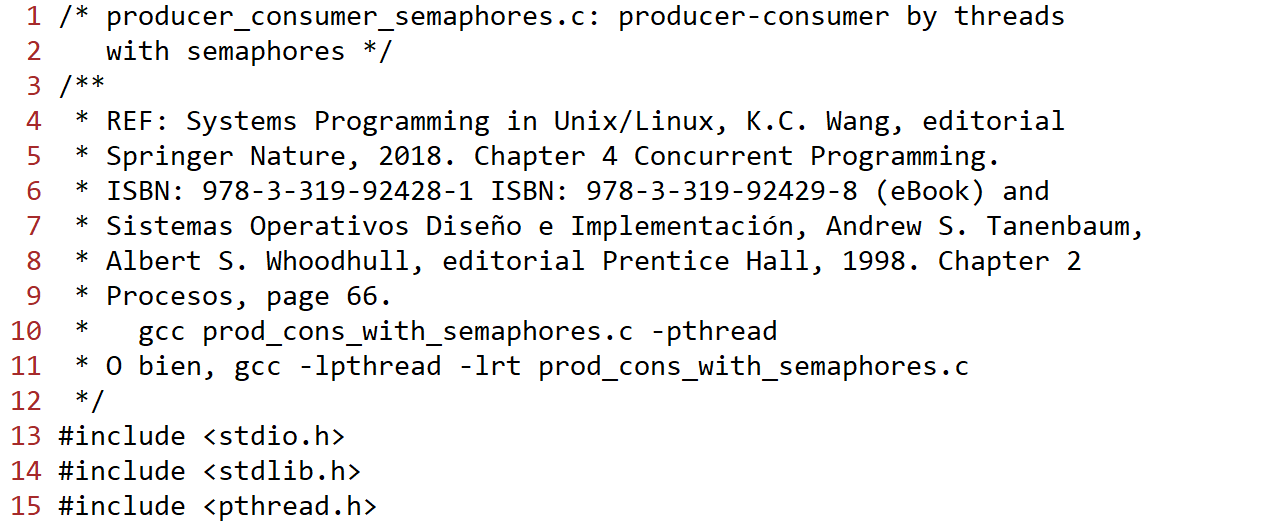
Esta solución usa tres semáforos: uno llamado full para contar el número de ranuras que están llenas, uno llamado empty para contar el número de ranuras que están vacías, y otro llamado mutex para asegurarse de que el productor y el consumidor no accedan al buffer al mismo tiempo. Full inicialmente vale 0 (cero), empty inicialmente es igual al número de ranuras del buffer y mutex inicialmente 1. Los semáforos a los que se asigna 1 como valor inicial y son utilizados por dos o más procesos para asegurar que sólo uno de ellos pueda entrar en su región crítica al mismo tiempo se denominan **semáforos binarios**. Si cada proceso ejecuta DOWN (sem\_wait()) justo antes de entrar en su región crítica, y UP (sem\_post()) justo después de salir de ella, la exclusión mutua está garantizada.

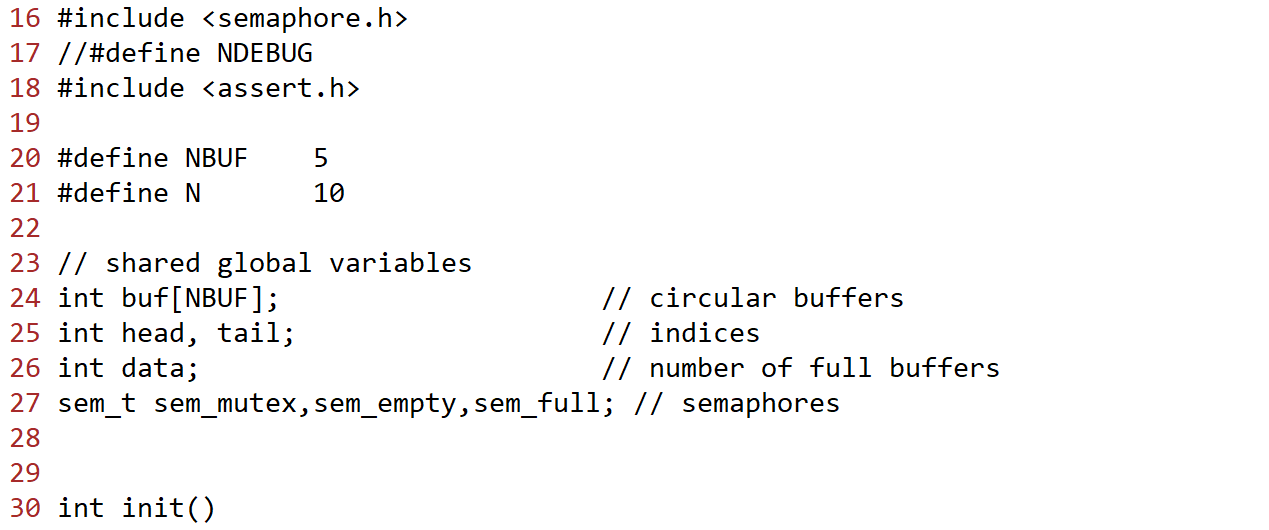
Los semáforos son una buena primitiva de comunicación entre procesos. En un sistema que usa semáforos, la forma natural de ocultar las interrupciones es tener un semáforo, inicialmente puesto en 0, asociado a cada dispositivo de E/S. Inmediatamente después de iniciar un dispositivo de E/S, el proceso que lo administra ejecuta DOWN (sem\_wait()) con el semáforo correspondiente, bloqueándose así de inmediato. Cuando llega la interrupción, el manejador de interrupciones ejecuta UP (sem\_post()) con el semáforo correspondiente, haciendo que el proceso en cuestión quede otra vez listo para ejecutarse.

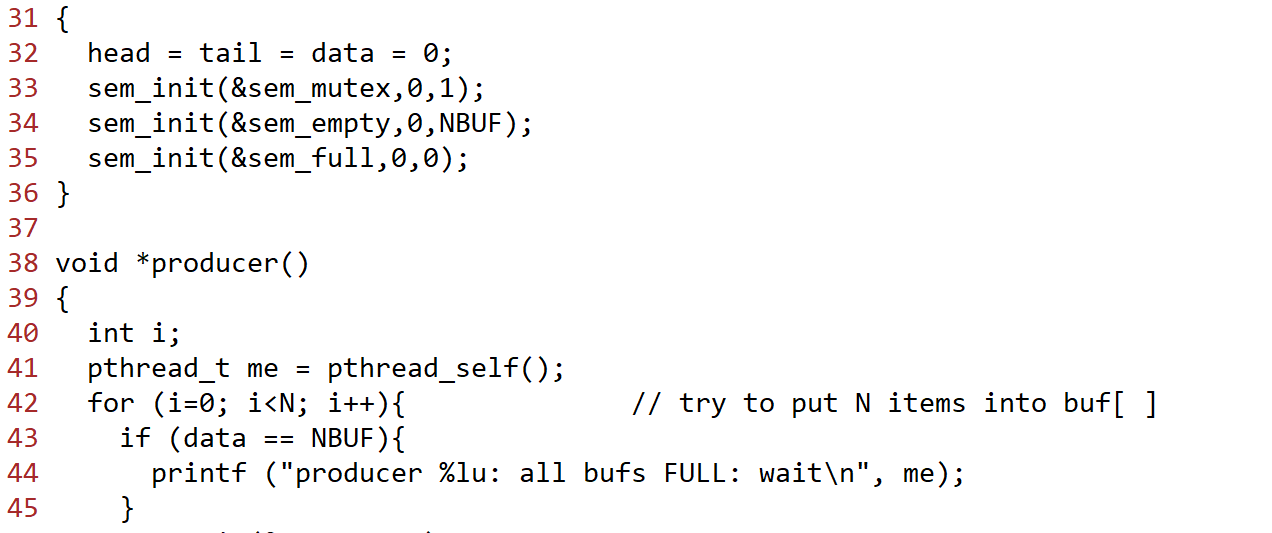
En el ejemplo de la figura 2-12, realmente usamos los semáforos de dos formas distintas. Esta diferencia es lo bastante importante como para hacerla explícita. El semáforo mutex se usa para exclusión mútua; está diseñado para garantizar que solo un proceso a la vez estará leyendo o escribiendo el buffer y las variables asociadas a él. Esta exclusión mutua es necesaria para evitar el caos.

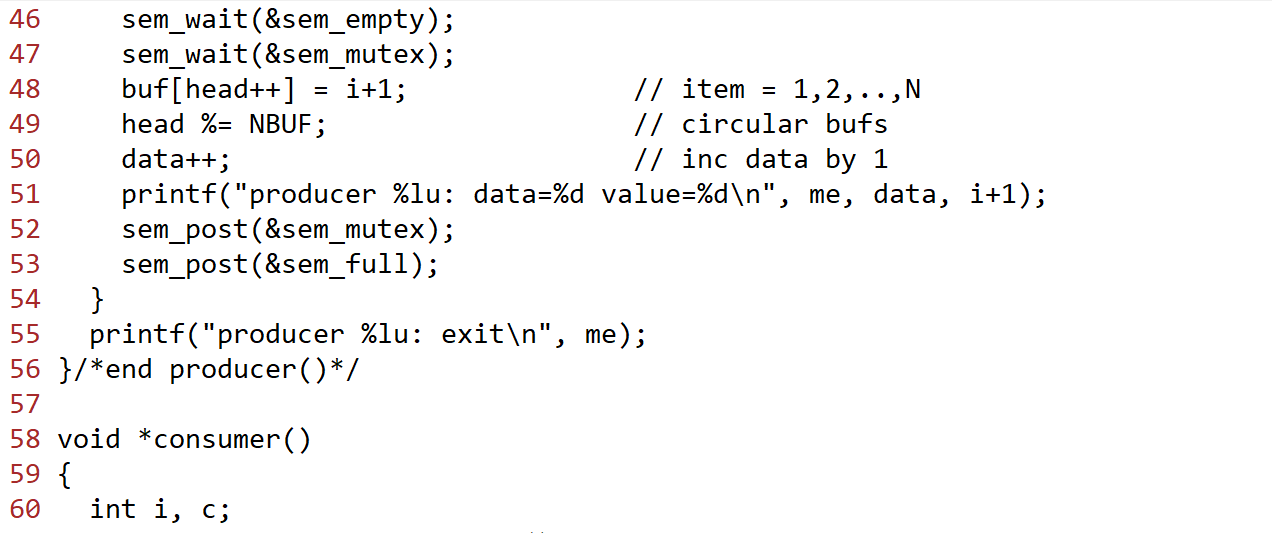
El otro uso de los semáforos es la **sincronización**. Los semáforos full y empty se necesitan para garantizar que ciertas secuencias de sucesos ocurran o no ocurran. En este caso, los semáforos aseguran que el productor dejará de ejecutarse cuando el buffer esté lleno y que el consumidor dejará de ejecutarse cuando el buffer esté vacío. Este uso es diferente de la exclusión mutua.

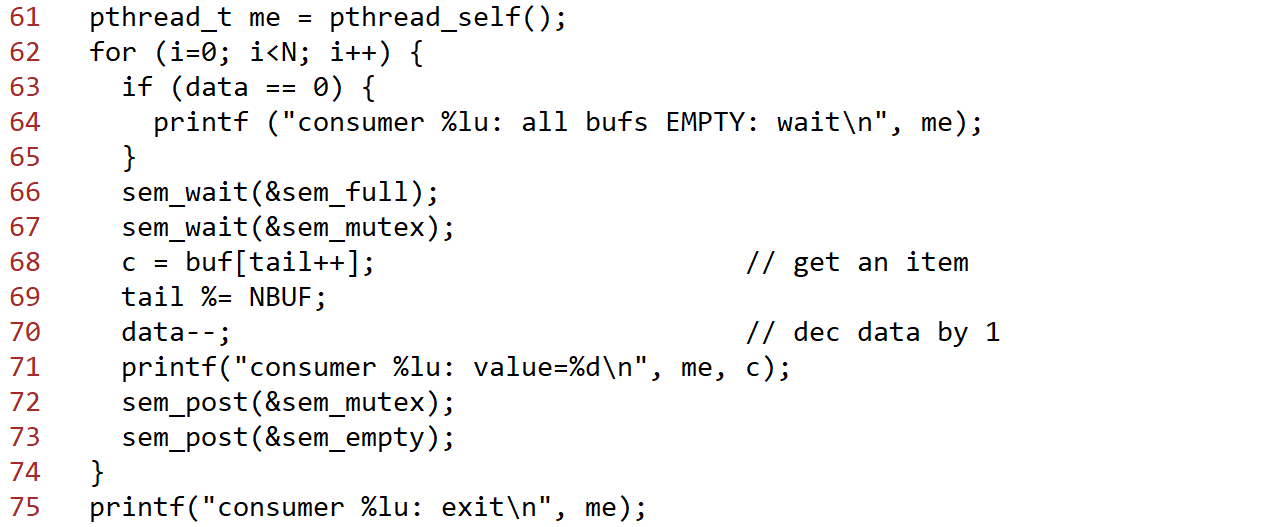
A continuación, se incluye el código de un programa que soluciona el problema de buffer acotado o productor consumidor.















# Barreras

La operación join de hilos permite a un hilo (usualmente el hilo principal) esperar por la terminación de otros hilos. Después de que los hilos por los que se está esperando han terminado, el hilo principal podría crear nuevos hilos para continuar ejecutando las siguientes partes de un programa paralelo. Esto requiere el trabajo extra de crear nuevos hilos. Hay situaciones en las cuales sería mejor mantener los hilos vivos, pero requerir que no avancen hasta que todos ellos han alcanzado un punto de sincronización prescrito. En Pthreads, el mecanismo es la **barrera**, junto con un conjunto de funciones para barreras. Primero, el hilo principal crea un objeto barrera



y hace la llamada



Para inicializar el objeto barrera, con el número de hilos que se van a sincronizar en la barrera. Entonces el hilo principal crea hilos de trabajo para realizar tareas. Los hilos de trabajo usan



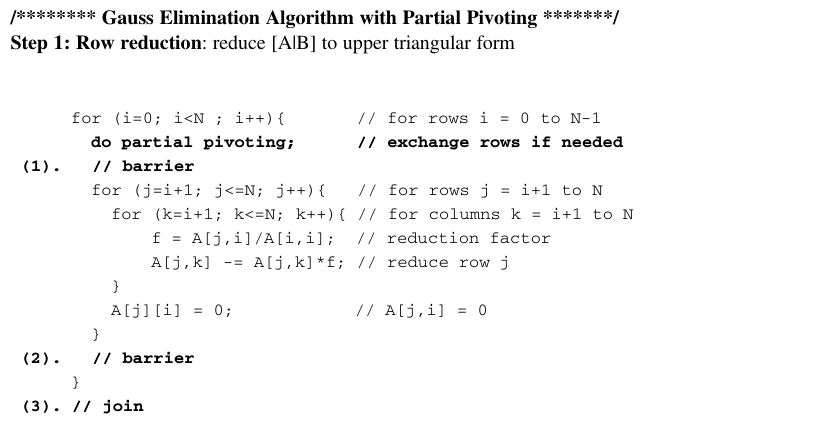
para esperar en la barrera hasta que el número de hilos especificado han alcanzado la barrera. Cuando el último hilo llega a la barrera, todos los hilos continúan o retoman su ejecución otra vez. En este caso, una barrera actúa como un punto encuentro (rendezvous), en lugar de un cementerio (graveyard), de hilos. A continuación, se ilustra el uso de barreras con un ejemplo.

## Resolver un sistema de ecuaciones lineales usando hilos concurrentes

Demostramos la aplicación de hilos concurrentes, reunión (join) de hilos, y operaciones de barrera con un ejemplo.

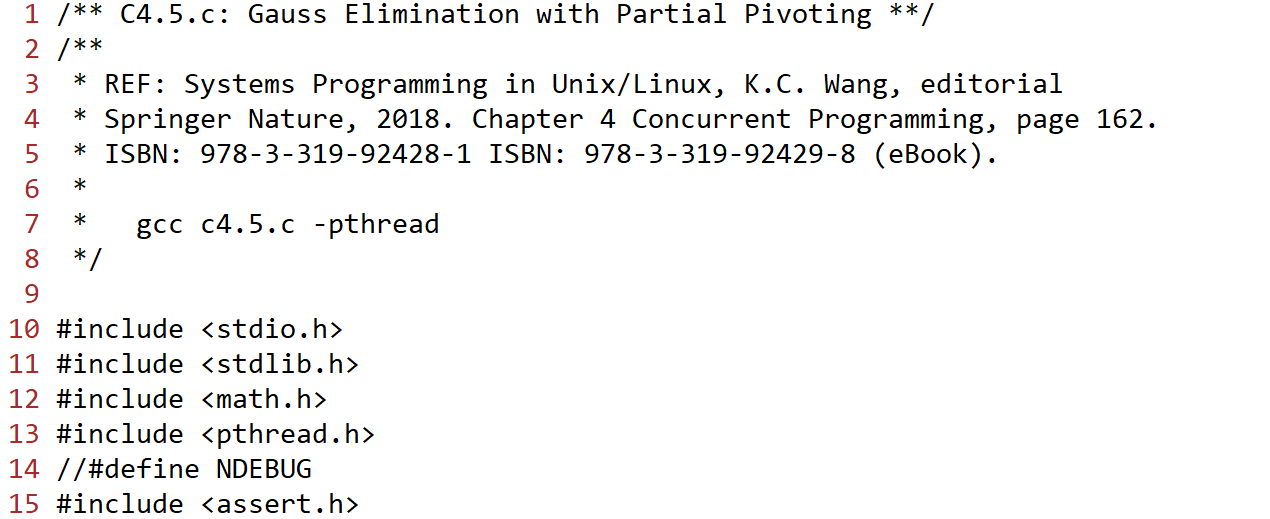
### Ejemplo:

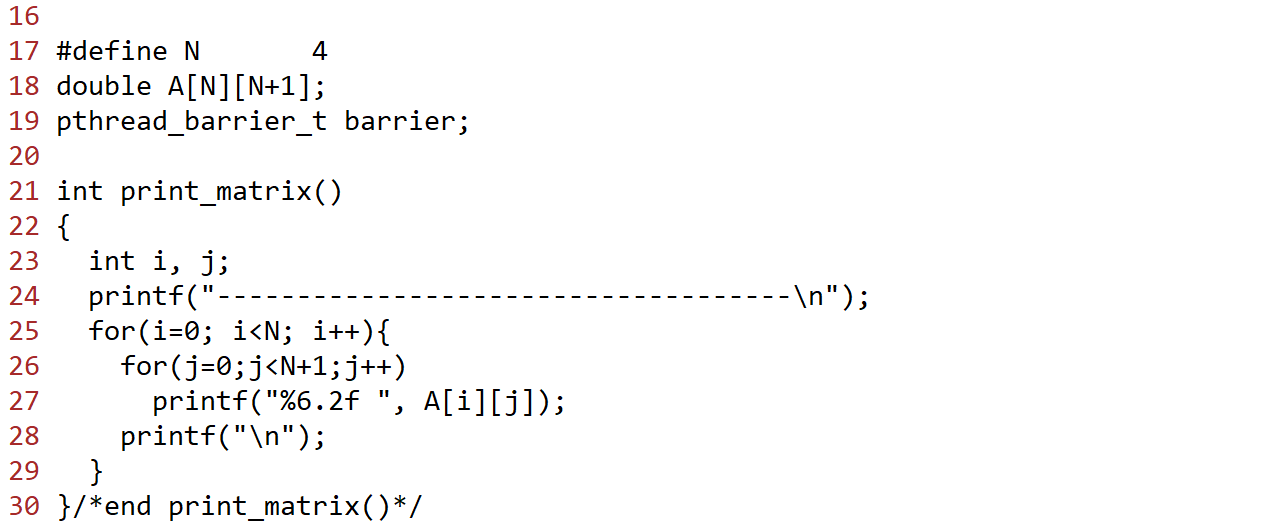
El ejemplo es resolver un sistema de ecuaciones lineales utilizando hilos concurrentes. Supongamos que %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
AX=B
\]
\end{document} es un sistema de ecuaciones lineales, donde %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
A
\]
\end{document} es una matriz de %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
N\times N
\]
\end{document} números reales, %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
X
\]
\end{document} es un vector columna de %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
N
\]
\end{document} incógnitas, y %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
B
\]
\end{document} es un vector columna de constantes. El problema es calcular el vector solución %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
X
\]
\end{document}. El algoritmo mejor conocido para resolver sistemas de ecuaciones lineales es la eliminación de Gauss. El algoritmo consiste de dos pasos principales; reducción de filas, el cual reduce la matriz combinada %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\left[\left.A\right|B\right]
\]
\end{document} a una forma triangular superior, seguido por una sustitución en reversa, con lo cual se calculan las componentes del vector solución %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
X
\]
\end{document}. En los pasos de reducción de filas, el **pivoteo parcial** es un esquema que asegura que el elemento principal de la fila usado para reducir otras filas tiene el valor absoluto máximo. El pivoteo parcial ayuda a mejorar la exactitud de los cálculos numéricos. Los siguientes fragmentos de código muestran un algoritmo de eliminación de Gauss con pivoteo parcial.

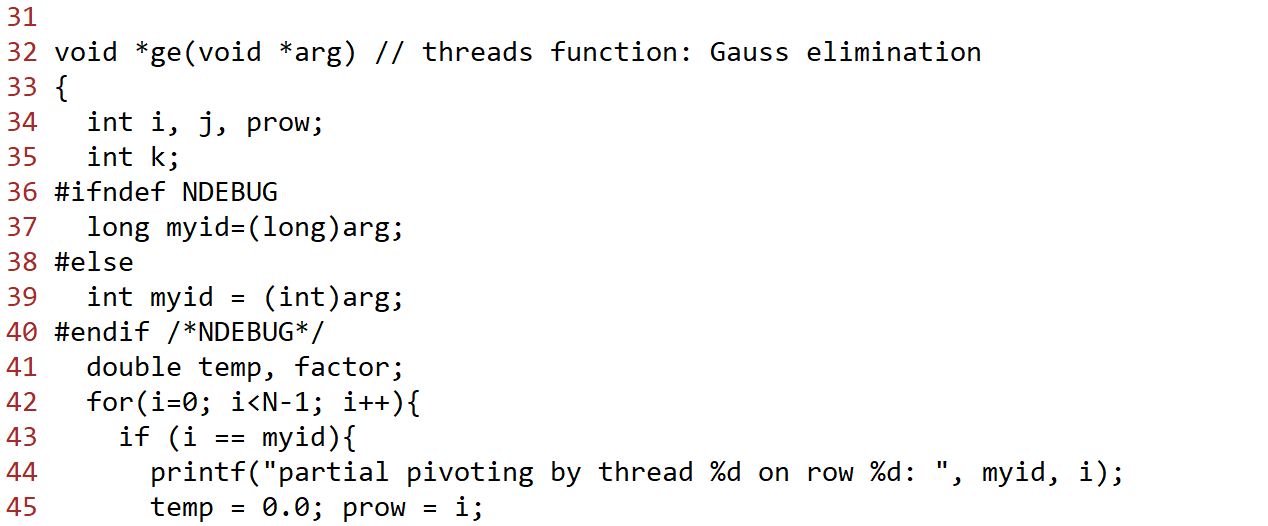


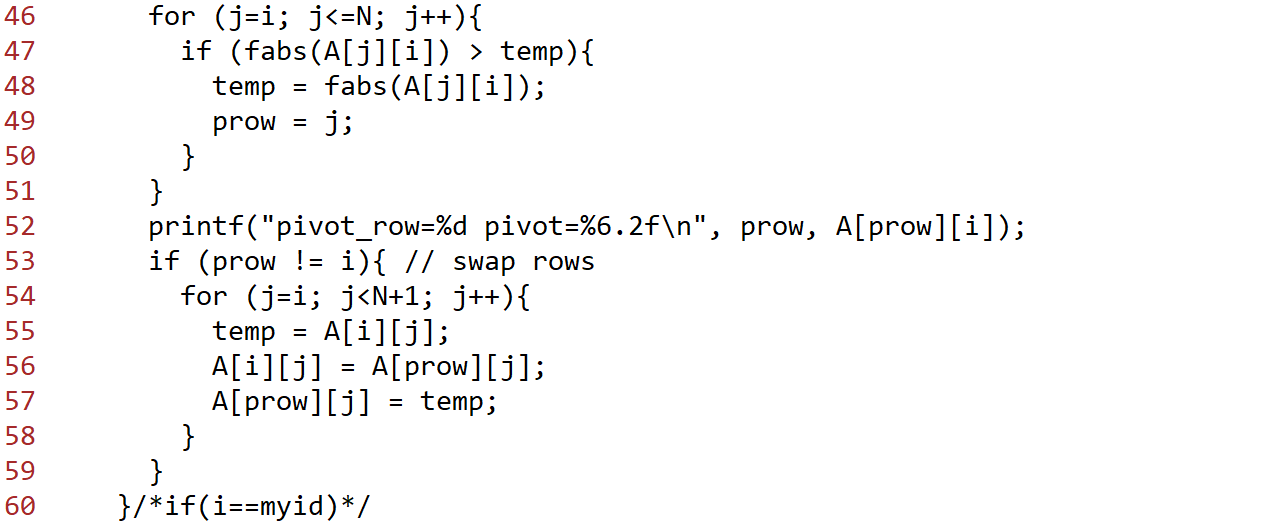


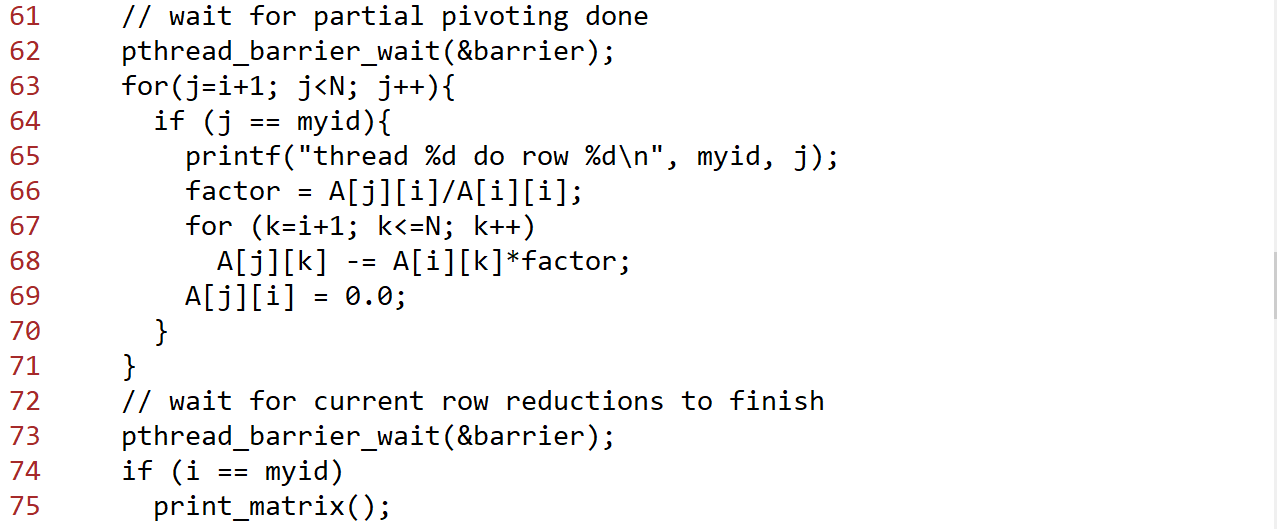
El algoritmo de eliminación de Gauss puede ser paralelizado como sigue. El algoritmo comienza con un hilo principal, el cual crea %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
N
\]
\end{document} hilos de trabajo para ejecutar la función **ge(thread\_id)** y espera a que todos los hilos de trabajo se reúnan (operación join). La función de hilos ge() implementa el paso de reducción de fila del algoritmo de eliminación de Gauss. En la función ge(), por cada iteración de row = 0 a %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
N-2
\]
\end{document}, el hilo con thread\_ID=i hace el pivoteo parcial sobre una fila i correspondiente. Todos los otros hilos esperan en una barrera (1) [indicado en el fragmento de código arriba] hasta que el pivoteo parcial está completo. Entonces cada hilo de trabajo hace reducción de filas sobre una única fila igual a su número de ID. Dado que todos los hilos de trabajo deben haber terminado las reducciones de fila actual antes de que la iteración sobre la siguiente fila pueda empezar, por lo tanto, ellos esperan en otra barrera (2) [indicado en el fragmento de código arriba]. Después de reducir la matriz %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
\left[\left.A\right|B\right]
\]
\end{document} a la forma triangular superior, el paso final es calcular las soluciones %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
X[N-i]
\]
\end{document}, para %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
i\,=\,1
\]
\end{document} hasta %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
N
\]
\end{document} en ese orden, lo cual es inherentemente secuencial. El hilo principal debe esperar hasta que todos los hilos de trabajo han terminado antes de comenzar la sustitución en reversa. Esto se consigue con operaciones join en el hilo principal. Las siguientes imágenes muestran el código completo del programa de ejemplo c4.5.c.

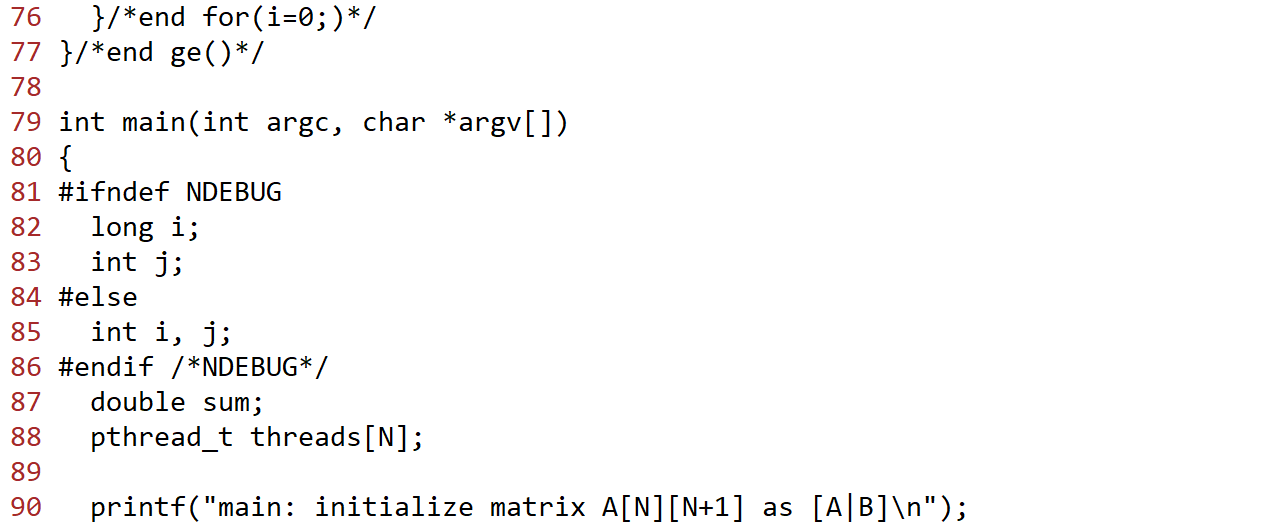


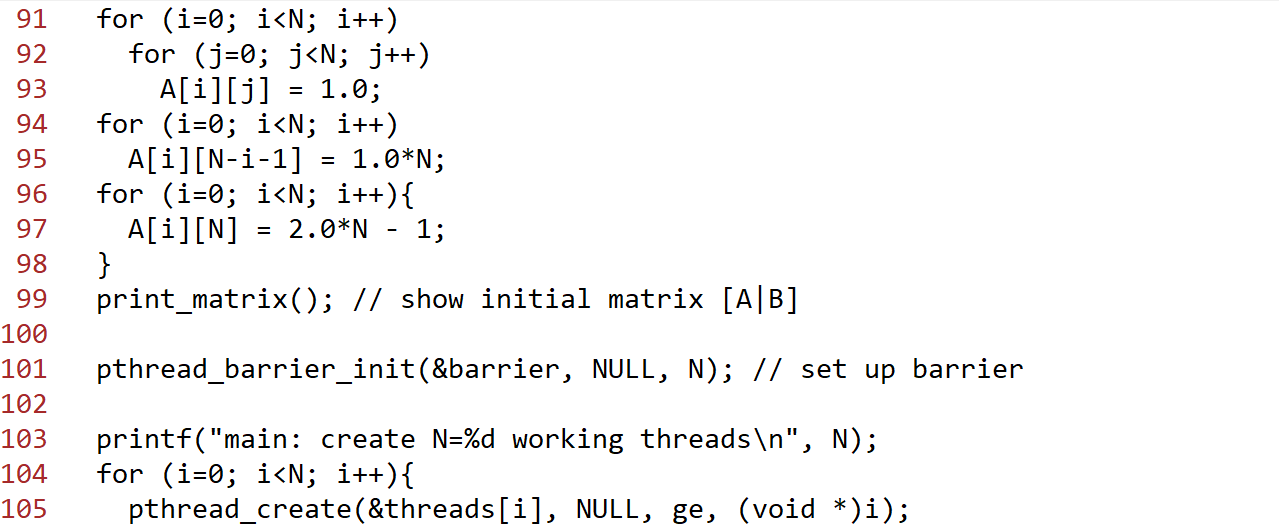


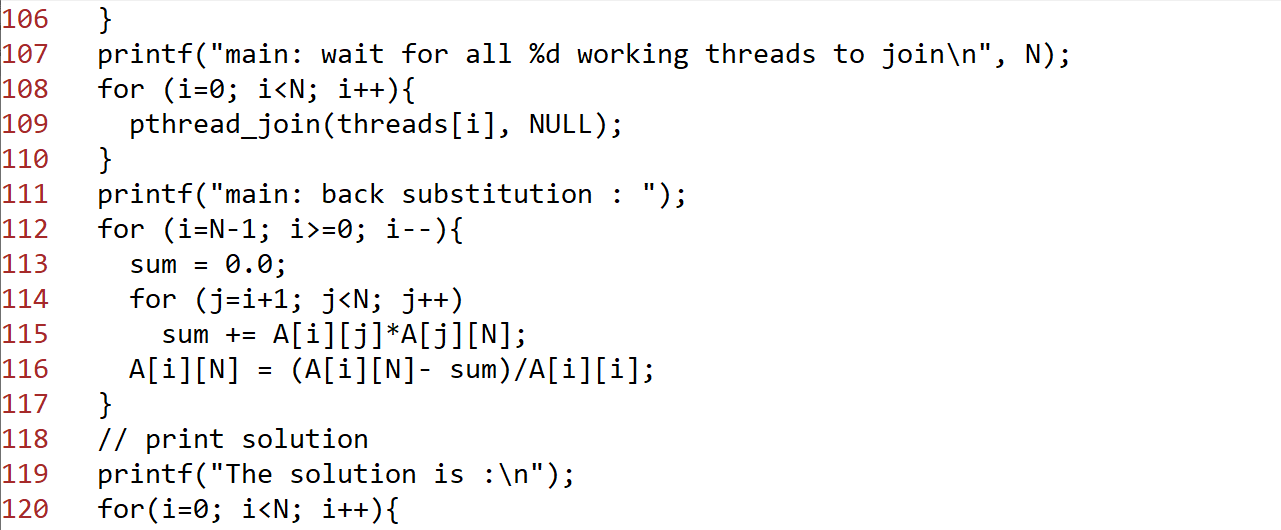






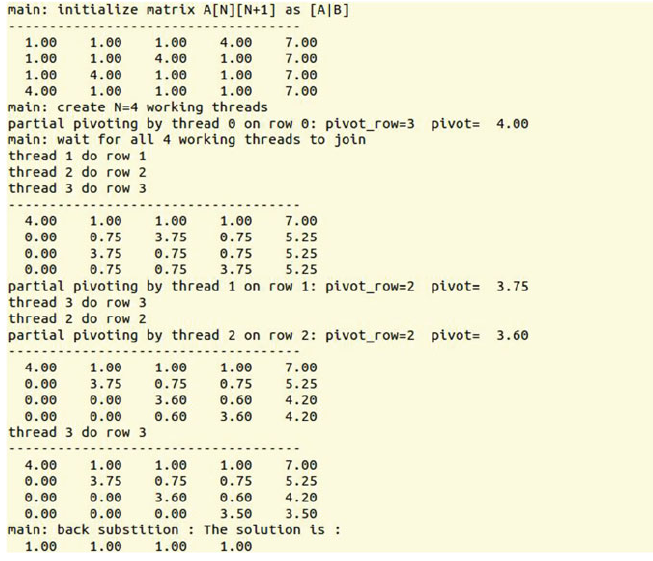








La siguiente figura muestra la salida del programa ejemplo c4.5.c, en esa ejecución se resuelve un sistema de ecuaciones con %FontSize=11
%TeXFontSize=11
\documentclass{article}
\pagestyle{empty}
\begin{document}
\[
N=4
\]
\end{document} incógnitas por el método de eliminación de Gauss con pivoteo parcial.

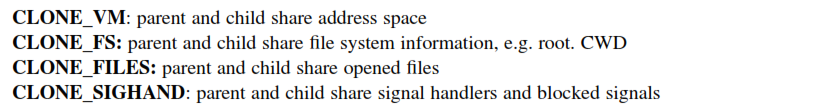


## Hilos en Linux

A diferencia de lo que sucede en otros sistemas operativos, Linux no distingue procesos de hilos. Para el kernel Linux, un hilo es meramente un proceso que comparte ciertos recursos con otros procesos. En Linux, ambos procesos e hilos son creados con la llamada al sistema clone(), la cual tiene el prototipo



Como se puede ver, clone() es más como una función de creación de hilos. Crea un proceso hijo para ejecutar una función fn(arg) con un child\_stack. El campo flags especifica los recursos a ser compartidos por el padre y el hijo, el cual incluye

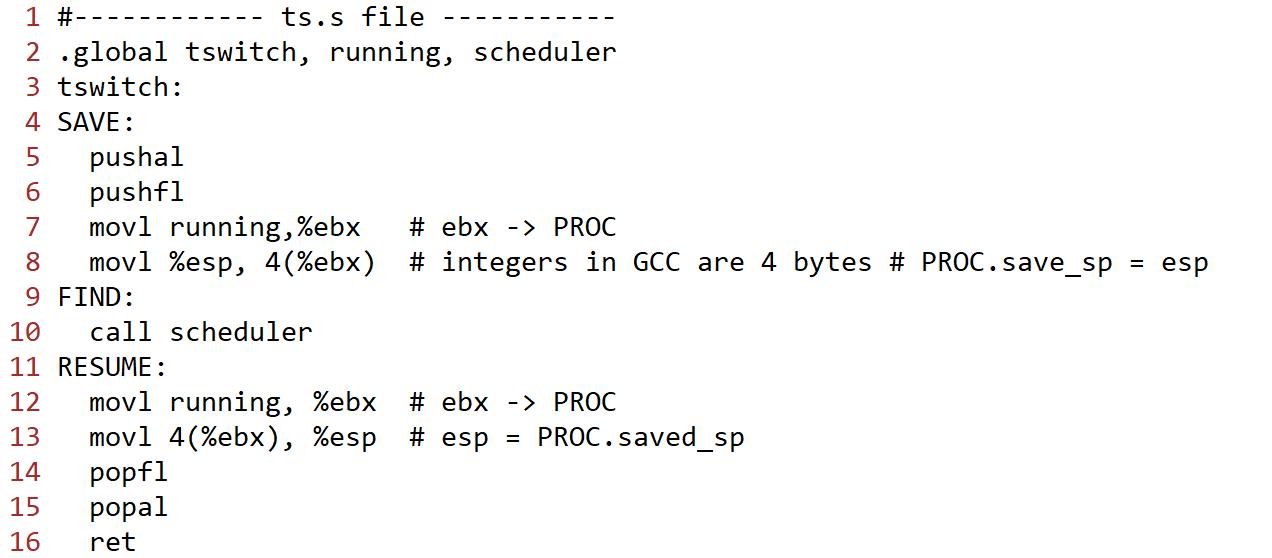


Si se especifica alguna de las banderas, ambos procesos comparten exactamente el mismo recurso, no una copia separada del recurso. Si una bandera no está especificada, el proceso hijo usualmente obtiene una copia separada del recurso. En este caso, los cambios hechos a un recurso por un proceso no afectan al recurso del otro proceso. El kernel Linux retiene fork() como una llamada al sistema pero podría ser implementada como un wrapper de biblioteca que llama a clone() pasando las banderas apropiadas. Un usuario promedio no tiene que preocuparse acerca de tales detalles. Es suficiente con decir que Linux tiene una forma eficiente de soportar hilos. Además, la mayoría de los kernels Linux actuales soportan multiprocesamiento simétrico (SMP). En tales sistemas Linux, los procesos (hilos) son planificados (scheduled) para correr sobre multiprocesadores en paralelo.

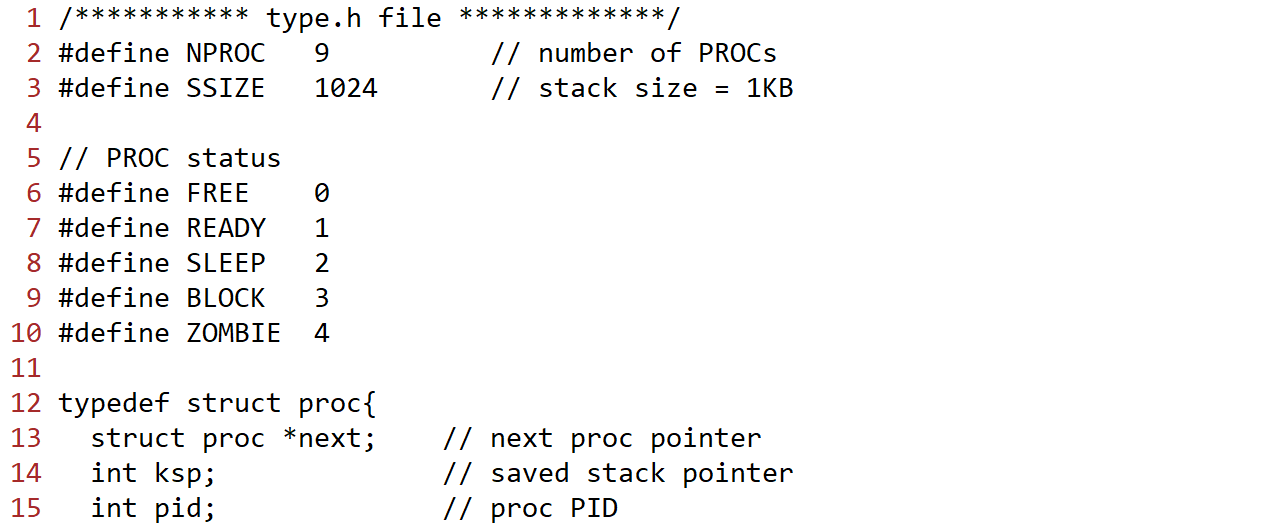
# Parte 1 Código base del proyecto

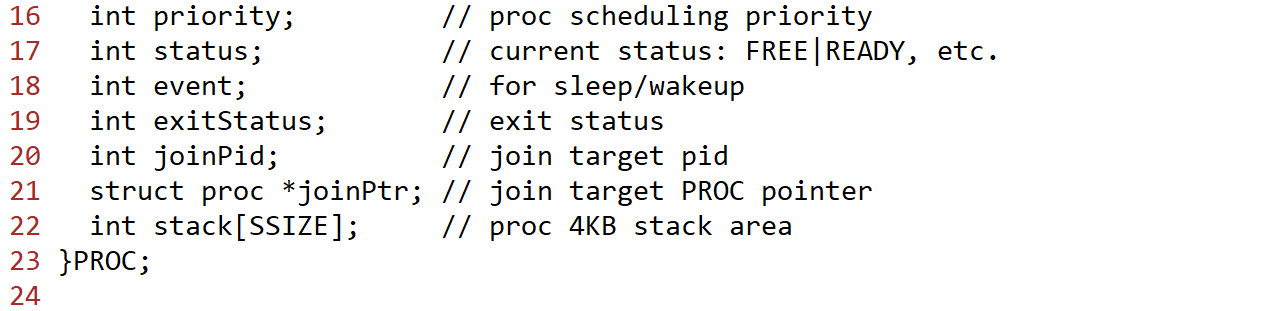
Código base: a continuación, se muestra el código base del proyecto de programación, el cual será explicado en secciones posteriores.

Archivo **ts.s** en código ensamblador GCC de 32 bits.

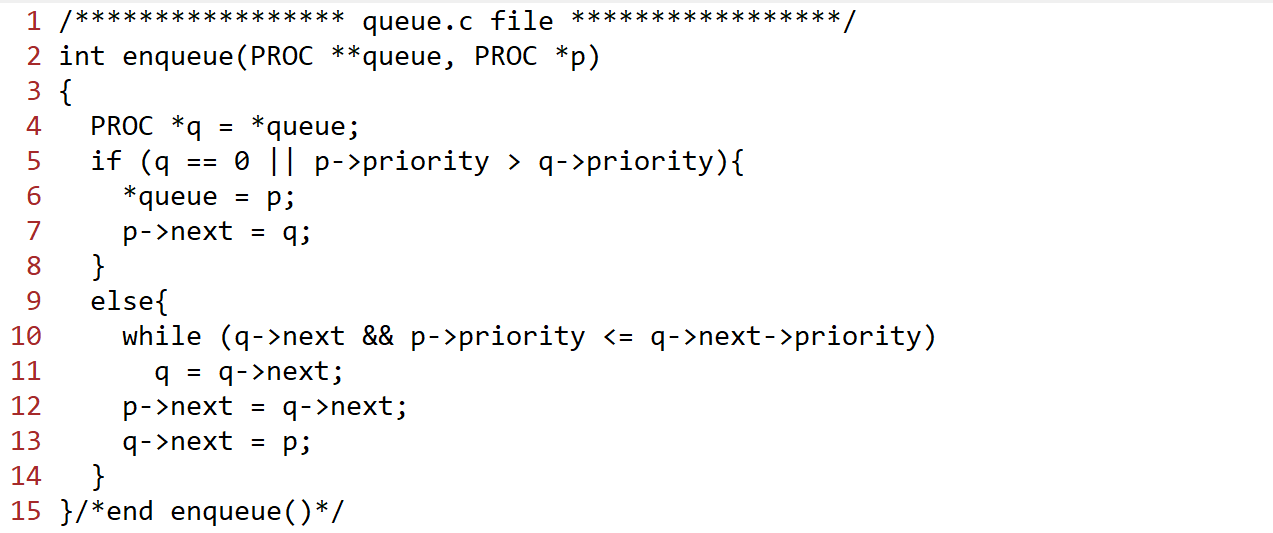


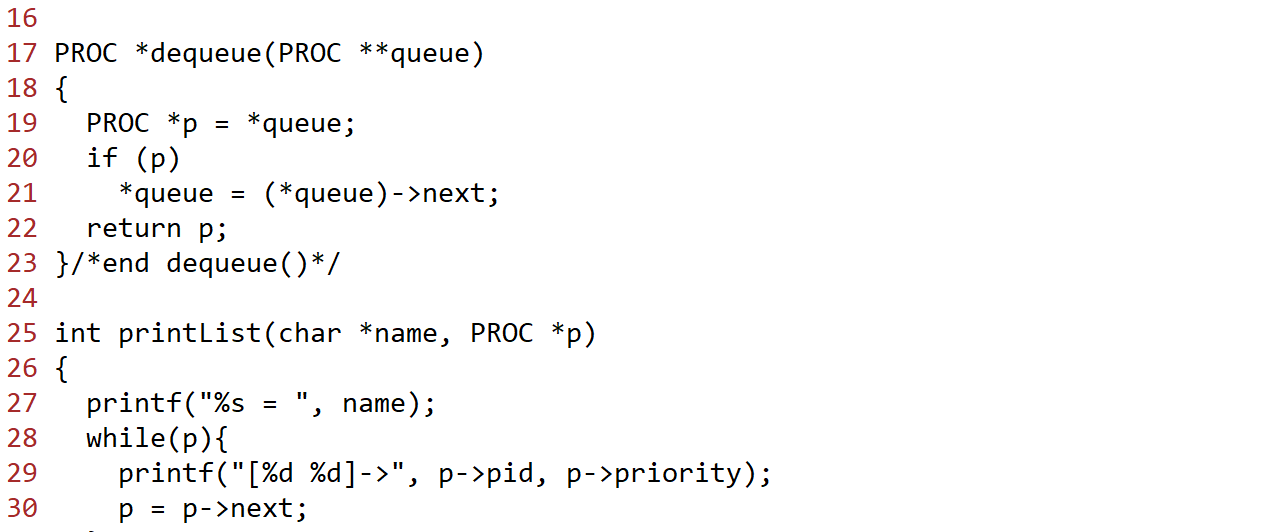
Archivo **type.h**: Este archivo define las constantes del sistema y la estructura PROC





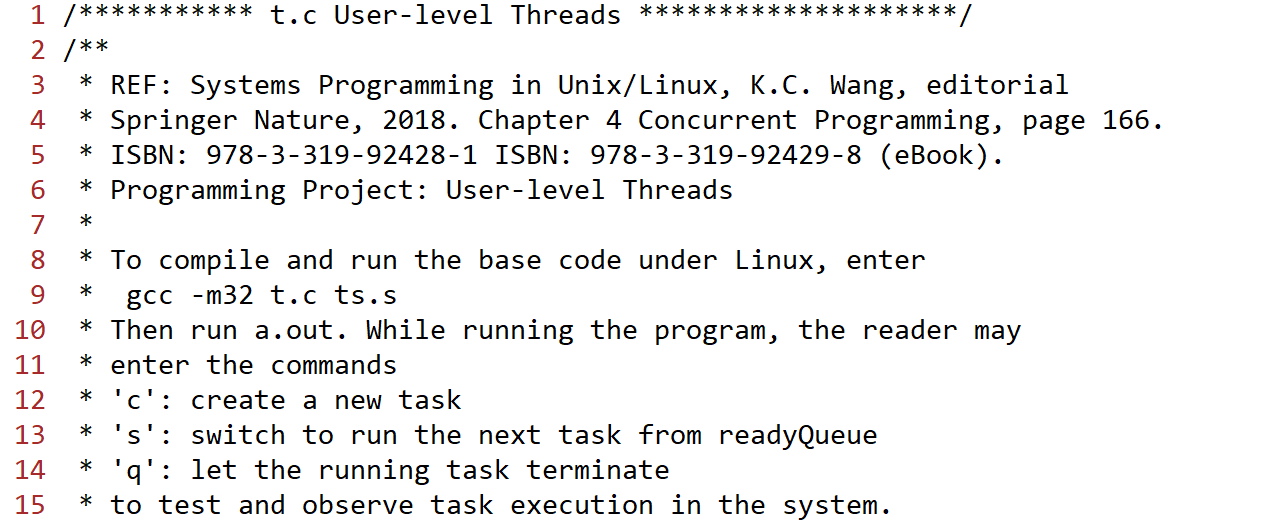
Archivo **queue.c**: el cual implementa las funciones de operación de colas. (el archivo queue.c de este proyecto es el mismo que en el archivo 01\_Practica\_Sincronizacion\_y\_Administracion\_de\_Procesos.docx)

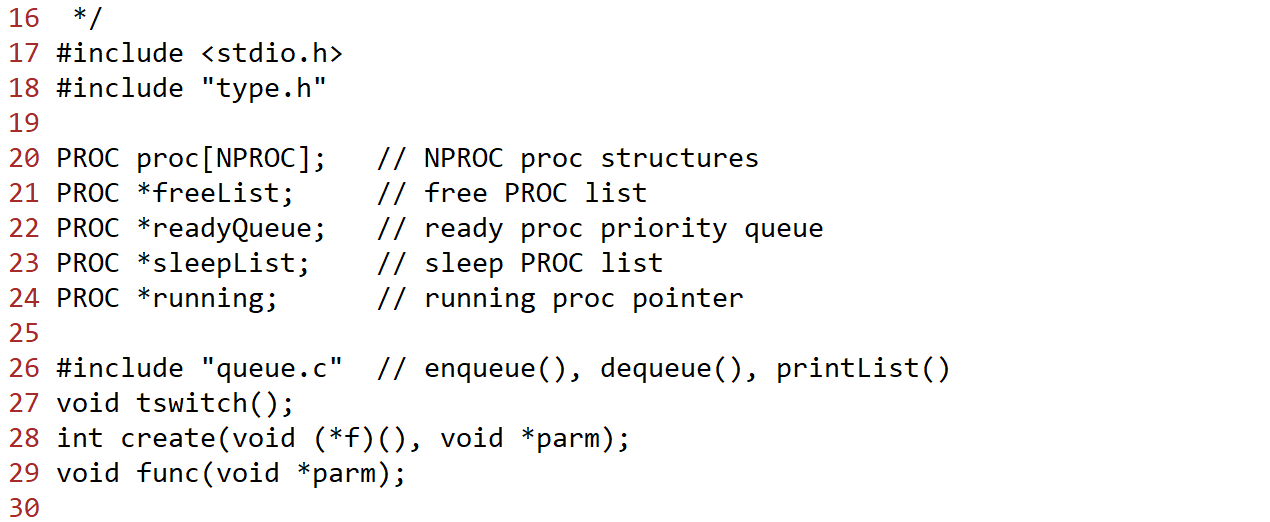


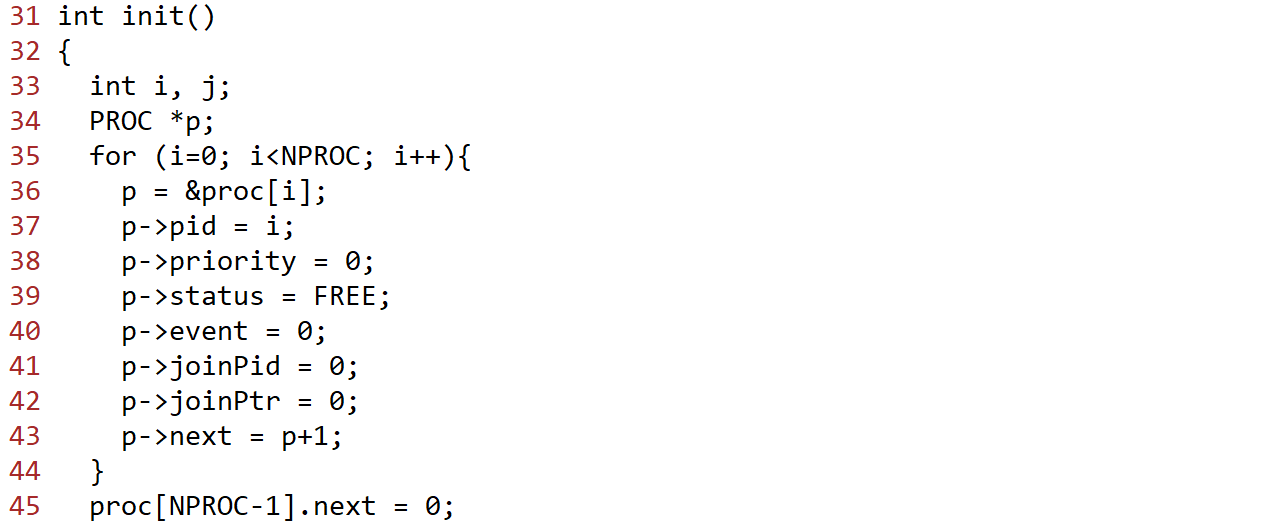


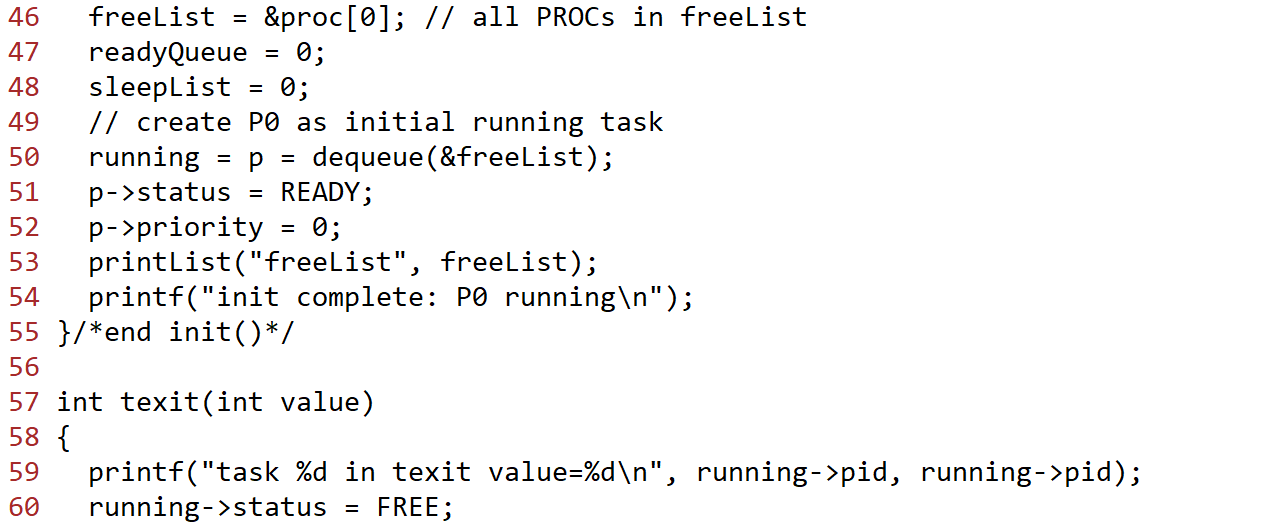


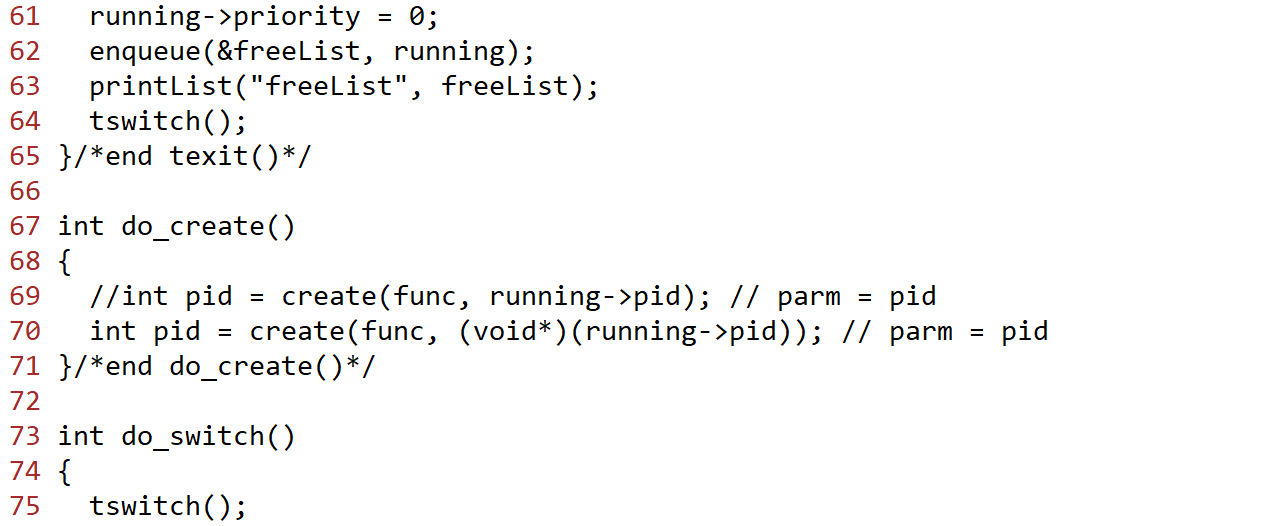
Archivo **t.c**: es el archivo principal del programa

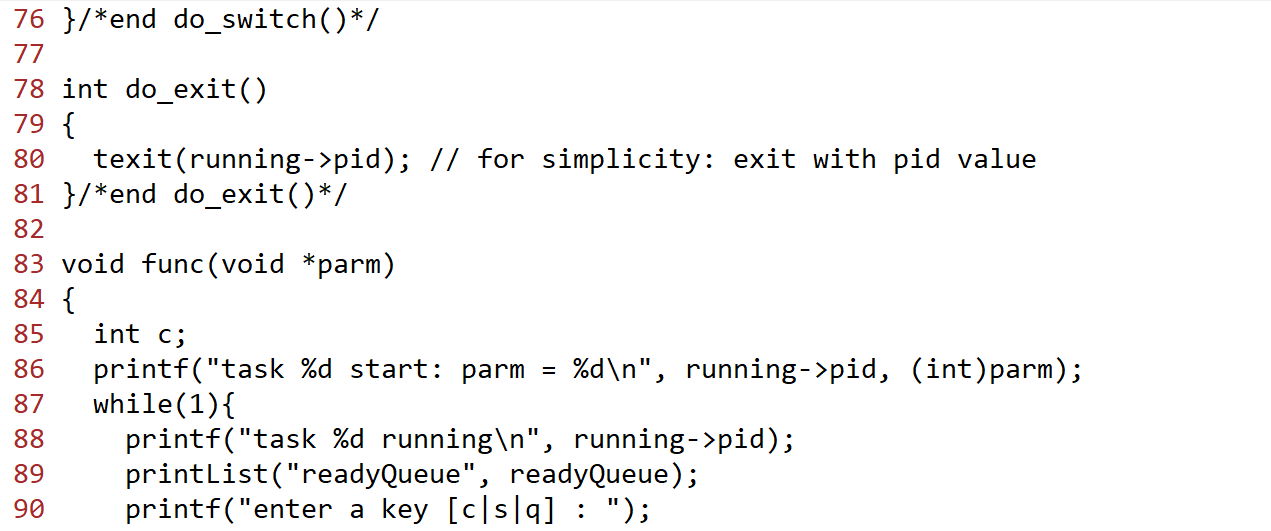


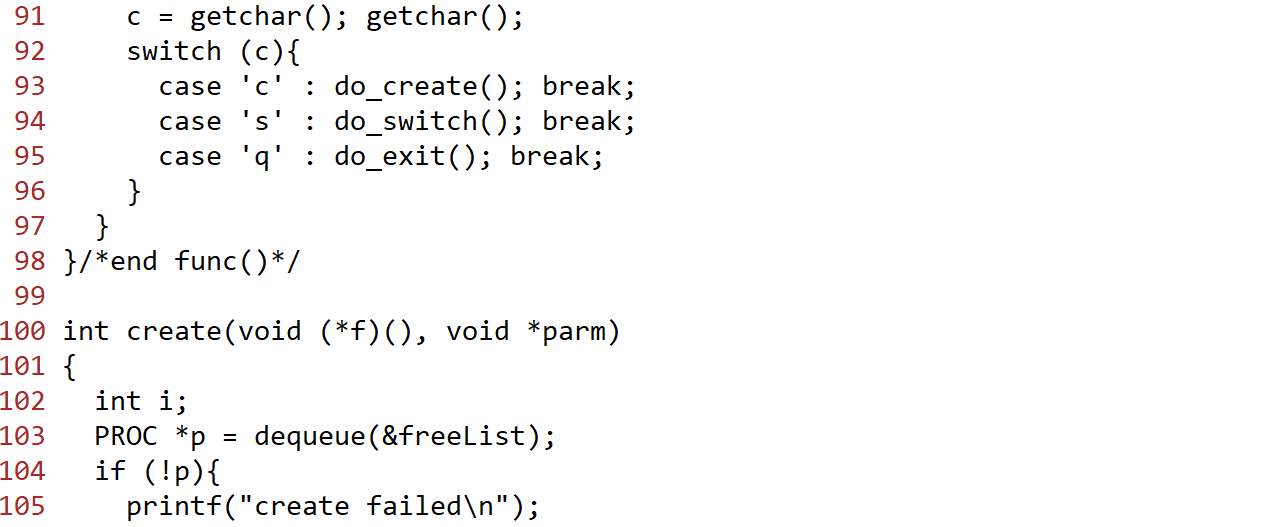


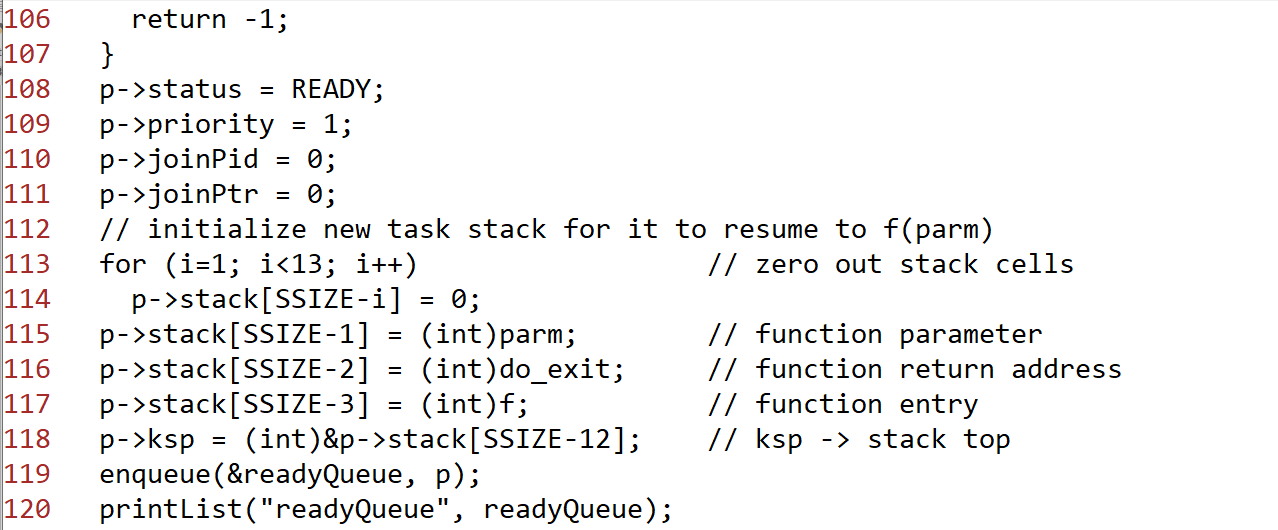


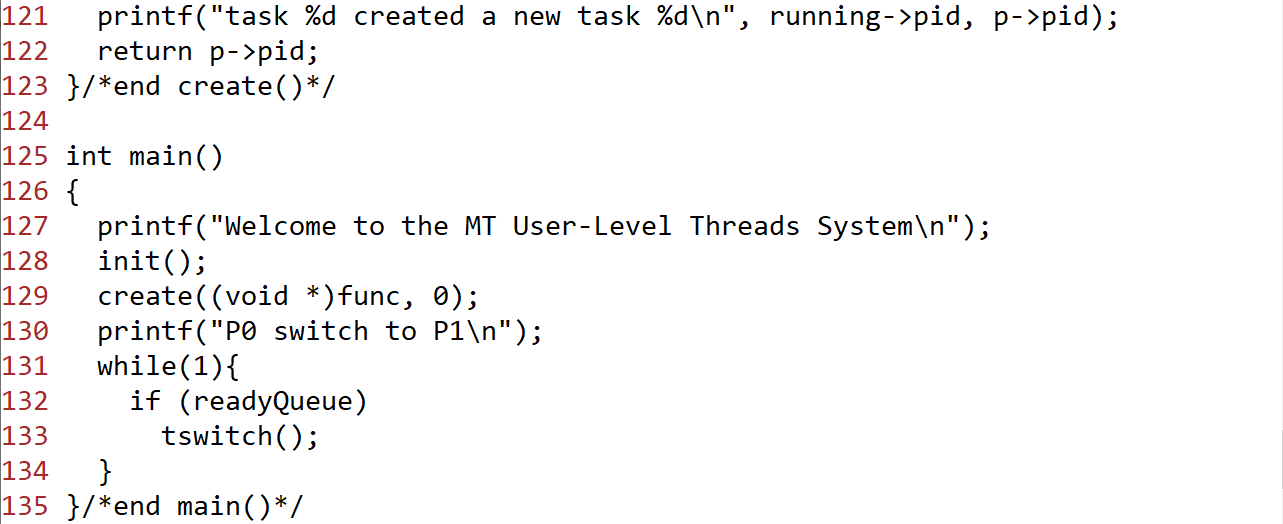


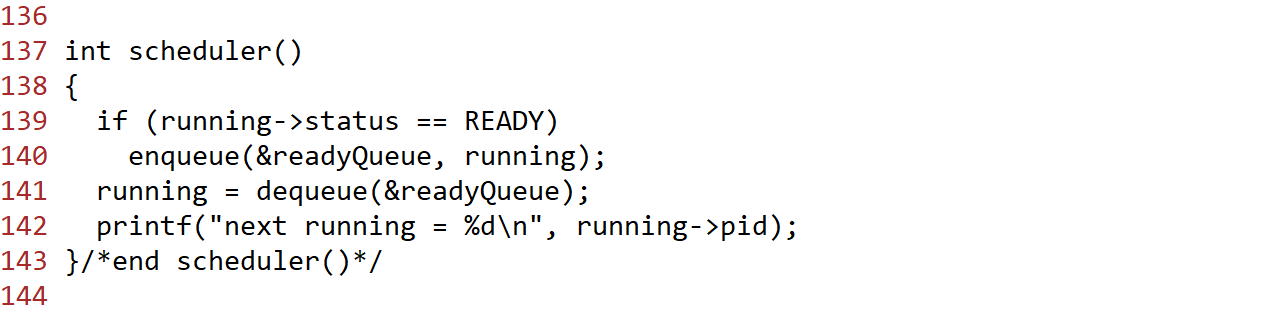




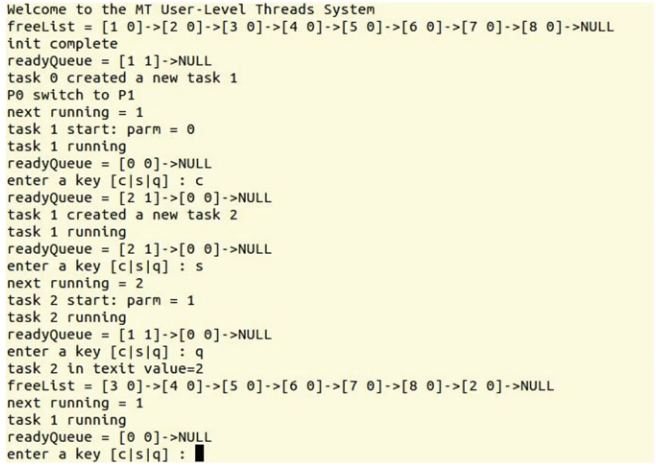






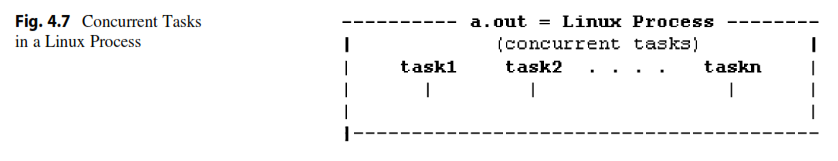


A continuación, se incluye una salida muestra del código base de este proyecto:



# Hilos a nivel de usuario (User-Level Threads)

El programa del código base completo corre como un proceso Linux en modo usuario. Dentro del proceso Linux están las entidades de ejecución independientes, las cuales son equivalentes a hilos convencionales. Que no deben ser confundidos con procesos o hilos Linux, nos referiremos a esas entidades de ejecución como tareas. En el sistema, las tareas corren concurrentemente usando multitasking, La Figura 4.7 muestra el modelo de tareas concurrentes dentro de un proceso Linux.

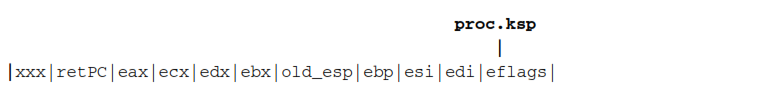


Dado que todas las tareas se ejecutan dentro de un proceso Linux, el kernel Linux no sabe de su existencia. Para el kernel Linux, el sistema MT completo es un solo proceso Linux, pero subdividimos el tiempo de ejecución de la CPU para correr diferentes tareas dentro de un solo proceso Linux. Por esta razón, las tareas son conocidas como **hilos a nivel de usuario** (**user-level threads**). El propósito de este proyecto de programación es crear un ambiente multitasking dentro de un proceso Linux para soportar hilos a nivel de usuario.

# Explicación del código base

1. **Task PROC Structure** Una diferencia principal entre procesos e hilos es que los primeros obedecen una relación padre-hijo, pero los hilos no. En el modelo de hilos, todos los hilos son iguales. Las relaciones entre los hilos son peer-to-peer en lugar de padre a hijo. Así que no se necesita el PID de una tarea padre y el árbol de la familia de tareas. Por lo tanto, se eliminan estos campos de la estructura PROC. Los nuevos campos joinPid y joinPtr son para implementar la operación unión de hilos, la cual será comentada posteriormente.
2. **Init():** Cuando el sistema comienza, main() llama a init() para inicializar el sistema. Init inicializa las estructuras PROC y las introduce a freeList. También inicializa readyQueue a empty. Entonces usa proc[0] para crear P0 como la tarea de ejecución inicial. P0 tiene la prioridad más baja 0. Todas las otras tareas tienen prioridad 1, de tal manera que tomarán turnos para correr desde readyQueue.
3. **P0** llama a create() para crear una tarea P1 para ejecutar la función task1(parm) con un parámetro parm=0, y la introduce en la readyQueue. Entonces P0 llama a switch() para cambiar la ejecución a la tarea P1.
4. **tswitch():** La función tswitch() implementa cambio de contexto de tarea. En lugar de usar pushl/popl sobre registros individuales de la CPU, utiliza las instrucciones PUSHAL y POPAL. tswitch() como una caja de cambio de tarea, a donde una tarea va y, en general, sale otra tarea. Consiste de tres partes separadas, las cuales se explican más detalladamente ahora.

(4).1. parte SAVE de tswitch(): Cuando una tarea llama a tswitch(), guarda la dirección de retorno sobre su propia pila y entra tswitch() en código ensamblador. En tswitch(), la parte SAVE guarda los registros de la CPU en la pila de la tarea que está llamando, guarda el apuntador de pila en proc.ksp. La CPU Intel x86 en el modo de 32 bits tiene muchos registros, pero solamente los registros eax, ebx, ecx, edx, esp, ebp, esi, edi y eflags son visibles para los procesos en modo usuario. Así que solo necesitamos guardar y recuperar esos registros cuando un proceso (un proceso Linux) se está ejecutando. El siguiente diagrama muestra el contenido de la pila y el apuntador de pila guardado de la tarea que está llamando después de ejecutar la parte SAVE de tswitch(), donde xxx denota el contenido de la pila antes de llamar a tswitch().

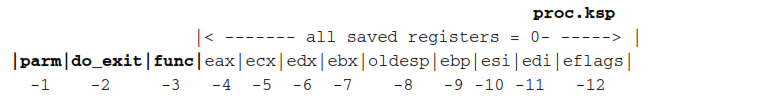


En la PCs Intel x86 en el modo de 32 bits, cada registro de CPU es de 4 bytes y las operaciones de pila son siempre en unidades de 4 bytes. Entonces, podemos definir la pila de PROC como un arreglo de enteros.

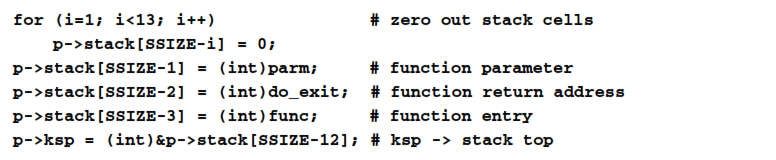
(4).2. **scheduler():** Después de ejecutar la parte SAVE de tswitch(), la tarea llama a scheduler() para elegir la siguiente tarea a ejecutarse. En scheduler(), si la tarea que está llamando aún tiene estado READY (aún puede correr), la tarea que está llamando llama a enqueue() para ponerse a símismo en la readyQueue por prioridad. En caso contrario, la tarea no estará en readyQueue, lo cual la hace no ejecutable. Entonces llama a dequeue(), lo cual devuelve la primera PROC lista removida de readyQueue como la siguiente tarea a ejecutarse.

(4).3. **Parte RESUME de tswitch():** Cuando la ejecución regresa de scheduler(), running podría haber cambiado para apuntar a la estructura PROC de una tarea diferente. La parte RESUME de tswitch() establece el apuntador de pila de la CPU al valor del apuntador de pila guardado de la tarea que se está ejecutando actualmente. Entonces extrae de la pila (ejecuta pop) los registros guardados, seguido por RET, causando que la tarea que está en ejecución actualmente regrese a donde llamó a tswitch() antes.

(5). **create():** La función create(func,parm) crea una nueva tarea y la introduce en la readyQueue. La nueva tarea comenzará su ejecución desde la función llamada **func()** con **parm** como parámetro. Aunque la nueva tarea nunca existió antes, podríamos pretender no solo que existió antes, sino que también corrió antes. La razón por la cual no está corriendo ahora es porque antes llamó a tswitch() para ceder la CPU. Si es así, su pila debe contener un frame guardado por la parte SAVE de tswitch() y su ksp guardado debe apuntar a la cima de la pila. Además, cuando una nueva tarea comienza la ejecución desde func(), la cual es escrita como func(void \*parm), esta debe tener una dirección de retorno y un parámetro parm sobre la pila. Cuando func() termina, la tarea “regresará a donde func() fue llamada antes”. Por eso, en create(), inicializamos la pila de la nueva tarea como sigue.



Donde el índice –i significa SSIZE-i. Estos son hechos por el siguiente segmento de código en create().



Cuando una nueva tarea comienza a correr, ésta ejecuta la parte RESUME de tswitch(), causando que regrese a la dirección de entrada de func(). Cuando la ejecución entra a func(), la cima de la pila contiene un apuntador a do\_exit() y un parámetro parm, como si ésta fue llamada como func(parm) desde la dirección de entrada de do\_exit(). En la práctica, la función de tarea raramente regresa. Si lo hace, regresaría a do\_texit(), lo cual causa que la tarea termine.

(5).**func():** Para propósitos de demostración, todas las tareas son creadas para ejecutar la misma función, func(parm). Por simplicidad, el parámetro de la función es establecido al pid de la tarea creadora, pero el lector podría pasar a la función cualquier parámetro, por ejemplo, un apuntador a estructura. Mientras se está ejecutando en func(), una tarea pide un comando

char = [c | s | q],

donde

‘c’ : crea una nueva tarea para ejecutar func(parm), con parm=pid de quien llama

‘s’ : switchea tarea;

‘q’ : termina y regresa la estructura PROC como FREE a la freeList

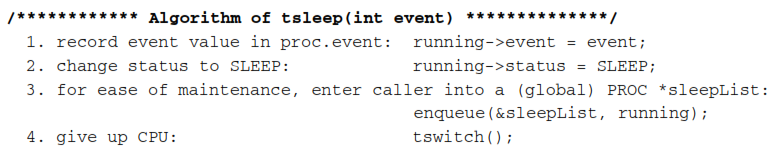
(6). **La tarea Idle P0:** P0 es especial en que tiene la prioridad más baja entre todas las tareas. Después de la inicialización del sistema, P0 crea P1 y switchea para correr P1. P0 correrá otra vez si y solo si no hay más tareas ejecutables. En ese caso, P0 simplemente se cicla. El usuario podría teclear Control-C para terminar el proceso (el proceso Linux). Se debe notar que la pila de P0 es realmente la pila del modo usuario del proceso Linux.

# Implementación de la operación join de Thread

## PARTE 2: Extender el código base para dar soporte a la operación join de tarea.

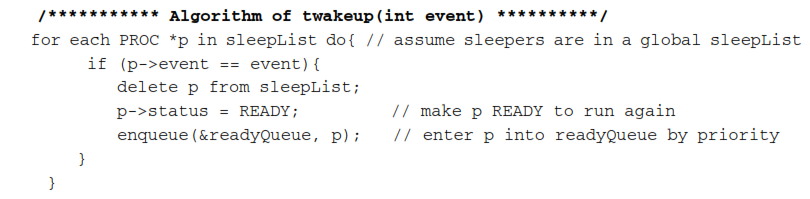
Las extensiones que se deben hacer aparecen listadas a continuación.

1. **tsleep(int event):** Siempre que una tarea debe esperar por algo que no está disponible, ésta llama a tsleep() para ir a dormir sobre un valor de evento, el cual representa la razón del sueño. El algoritmo de tsleep() es

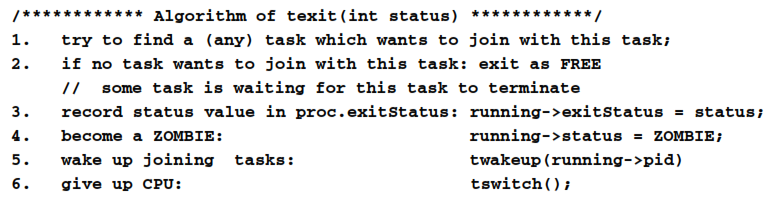


Una tarea que está durmiendo no es ejecutable hasta que es despertada por otra tarea. Así que en tsleep(), la tarea llama a tswitch() para ceder la CPU.

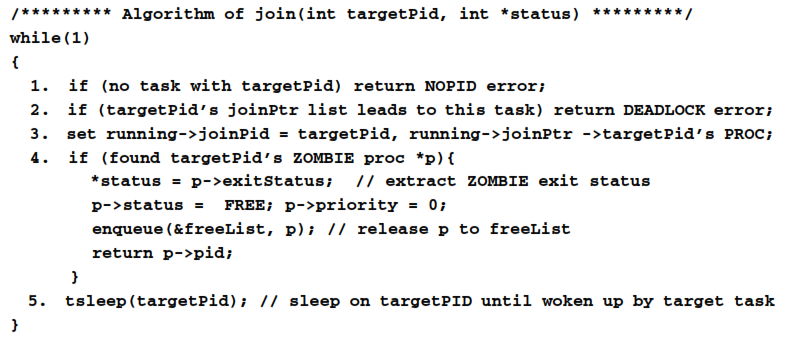
(2). **int twakeup(int event):** twakeup() despierta TODAS las tareas que están durmiendo sobre el valor de evento especificado. Si ninguna tarea está durmiendo sobre el evento, ésta no hace nada. El algoritmo de twakeup() es



(3). **texit(int status):** es texit() modificada en el código base para que las tareas terminen. En el modelo de proceso, cada proceso tiene un padre único (el cual podría ser el proceso INIT P1), el cual siempre espera a que el proceso termine. Así que un proceso que está terminando puede ser convertido a ZOMBIE y despertar a su padre, el cual dispondrá del hijo ZOMBIE. Si un proceso que está terminando tiene hijos, debe dirigir sus hijos ya sea al proceso INIT o a un proceso subreaper antes de convertirse en un ZOMBIE. En el modelo de hilos, los hilos son todos iguales. Un hilo no tiene un padre o hijos. Cuando un hilo termina, si no hay otros hilos esperando a que éste termine, el hilo debe salir como FREE. Si hay algún hilo esperando a que éste termine, el hilo debe convertirse en un ZOMBIE, despertar a tales hilos, permitiendo a uno de ellos encontrar y disponer del ZOMBIE. En el modelo de hilos, el algoritmo de terminación es

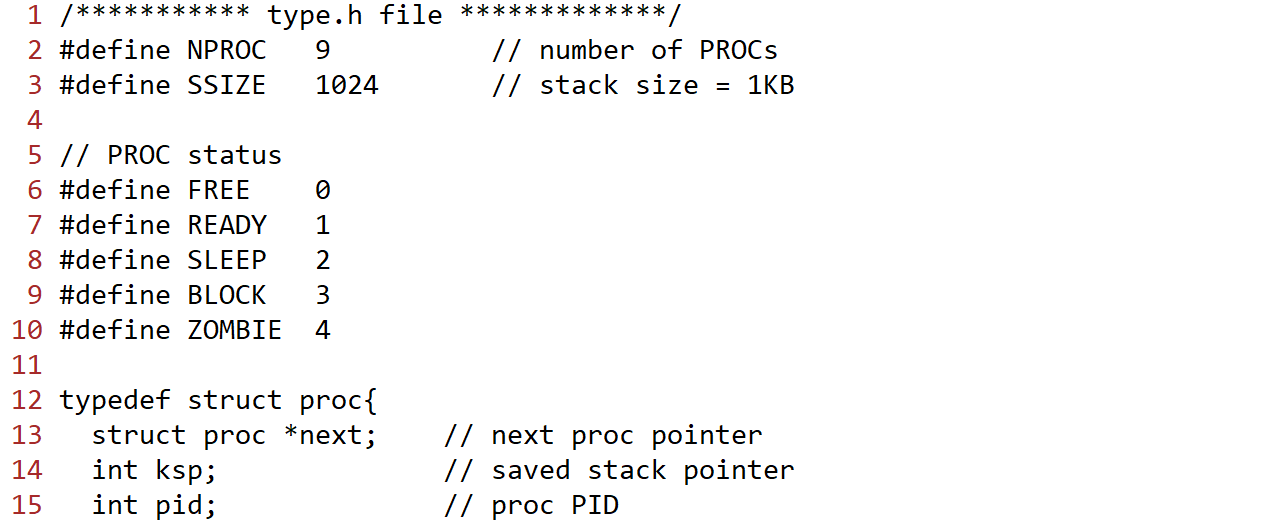


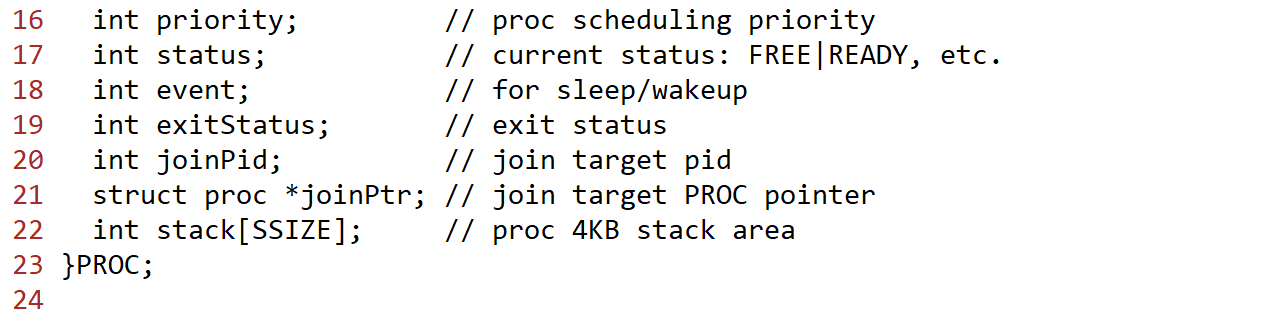
(4). **join(int targetPid,int \*status):** la operación join espera que un hilo con targetPid termine. En el modelo de proceso, cada proceso solamente puede ser esperado por su padre, que es único, y un proceso nunca espera que su propio padre termine. Así que la secuencia de espera es siempre una cadena unidireccional. En el modelo de hilos, a un hilo se le permite unirse (hacer la operación join) con cualquier otro hilo. Esto podría causar dos problemas. Primero, cuando un hilo trata de unirse con un hilo objetivo, el hilo objetivo podría ya no existir. En este caso, el hilo que está tratando de unirse debe regresar con un error. Si el hilo objetivo existe, pero no ha terminado aún, el hilo que trata de unirse debe ir a dormir sobre el PID del hilo objetivo, esperando a que el hilo objetivo termine. Segundo, dado que los hilos no obedecen la relación padre-hijo, una secuencia de operaciones join sobre hilos podría conducir a un ciclo de espera deadlock. Para prevenir los deadlocks, cada hilo que trate de unirse debe registrar la identidad del hilo objetivo en su estructura PROC para permitir que otros hilos que traten de unirse revisen si la petición de unión conducirá a un deadlock. Cuando el hilo que se tiene como objetivo para haer join, termina, puede revisar si hay algunos hilos esperando a que él termine. Si es así, éste se convertiría en un ZOMBIE y despertaría a todos esos hilos que están tratando de unirse. Una vez despertando, los hilos que se están uniendo deben ejecutar la operación join otra vez dado que el ZOMBIE podría estar ya liberado por otro hilo que estaba haciendo la operación join. Por eso, el algoritmo de la operación join de hilos es como sigue.

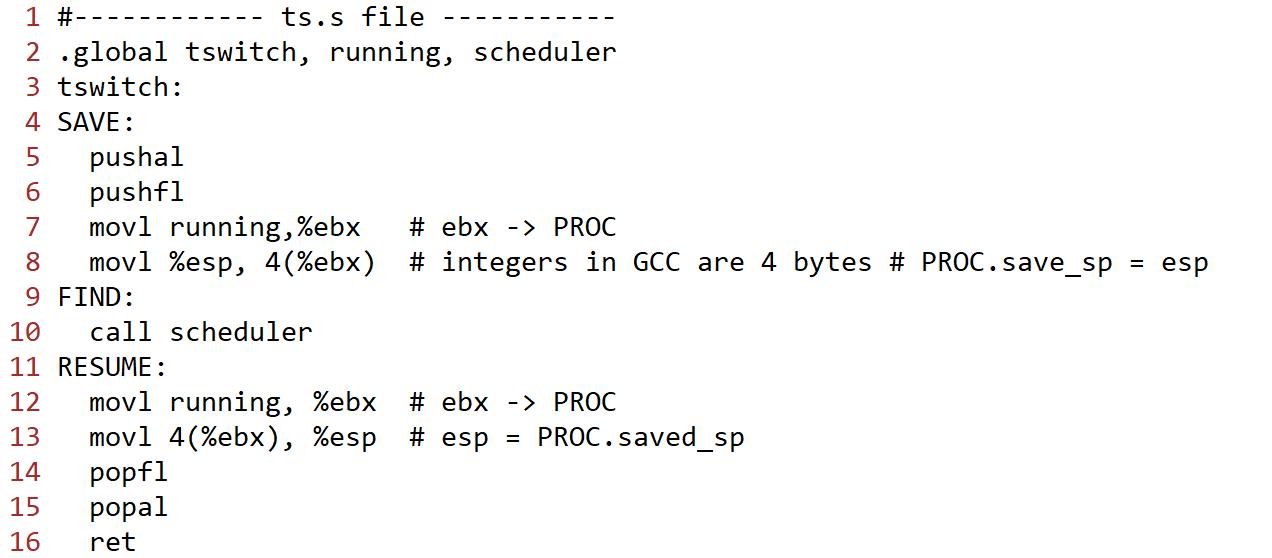


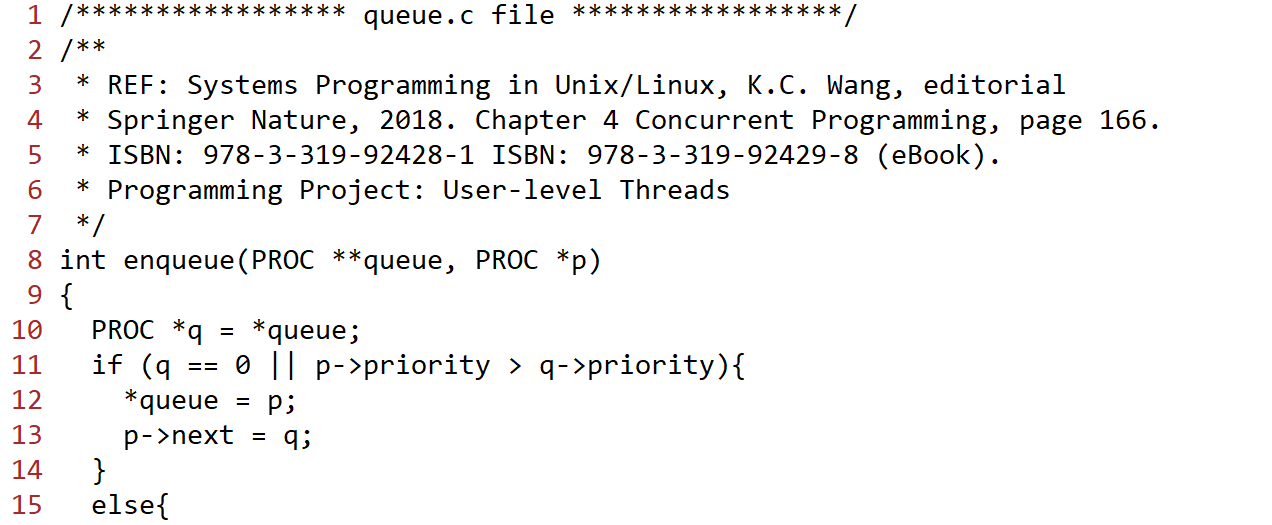
El paso 2 del algoritmo de join() previene el deadlock debido a ya sea peticiones de unión cruzadas o circulares, lo cual se puede hacer revisando los targetPid al recorrer los apuntadores de la lista de apuntadores joinPtr. Cada joinPtr de las tareas que están tratando de unirse apunta al PROC objetivo con el que intentan unirse. Si la lista apunta de regreso a la tarea corriendo actualmente, se tendría una lista de espera circular, la cual debe ser rechazada debido a un deadlock potencial.

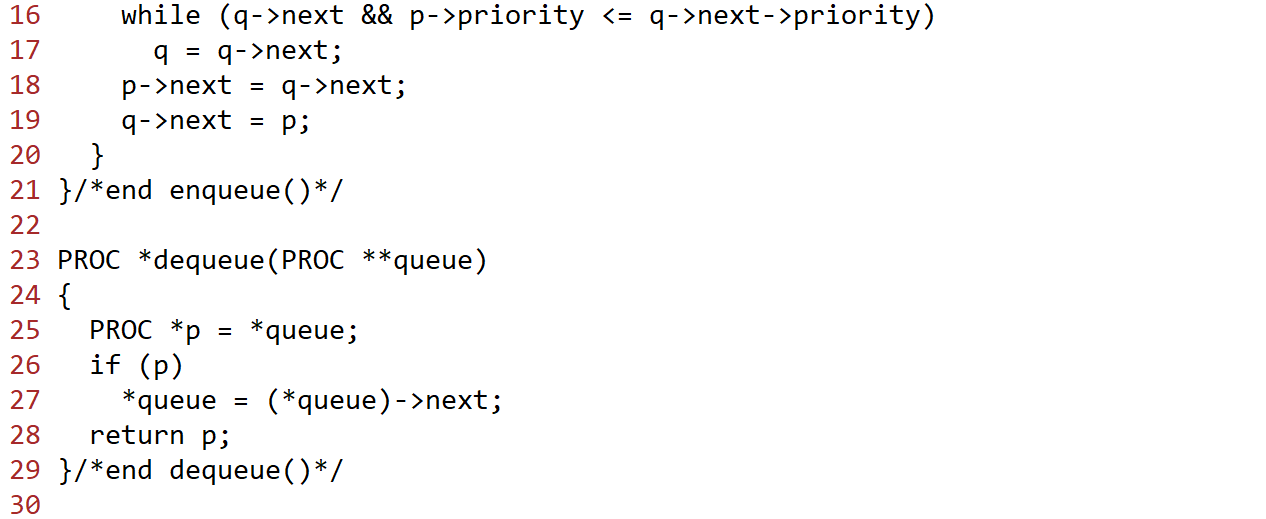
**PARTE 2 del Proyecto de Programación:** En la parte 2 del proyecto de programación se debe implementar las funciones tsleep(), twakeup(), texit() y join(). Entonces probar el sistema resultante con el siguiente programa, el cual usa creación de tareas y operaciones join del proyecto de programación.

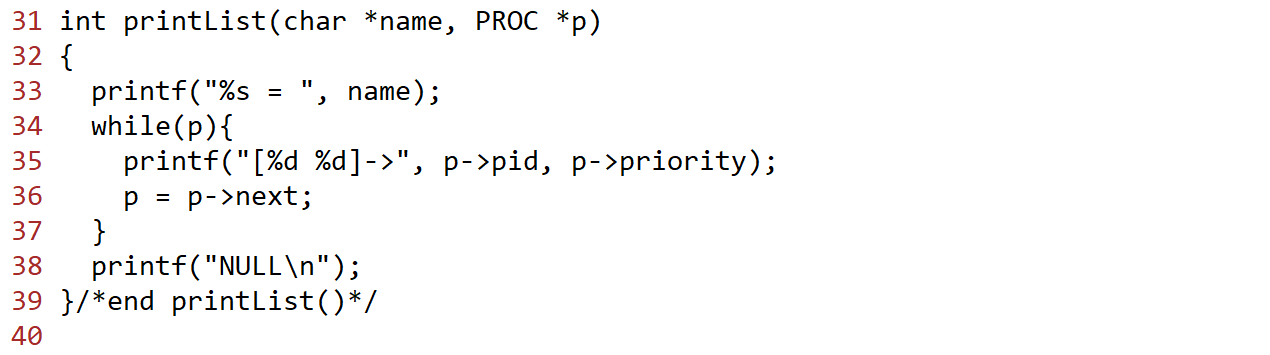




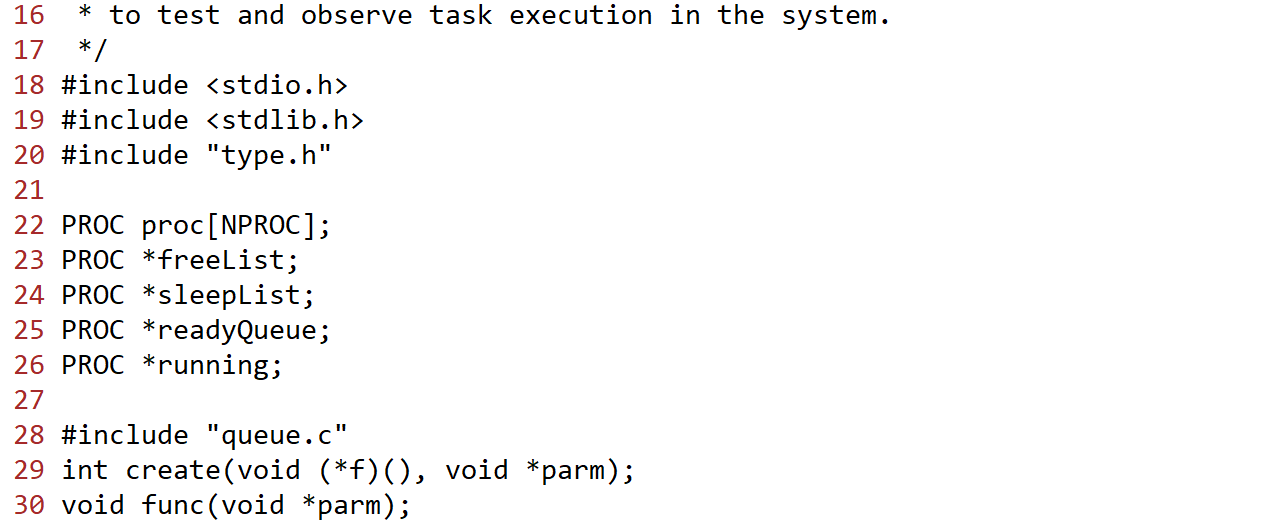


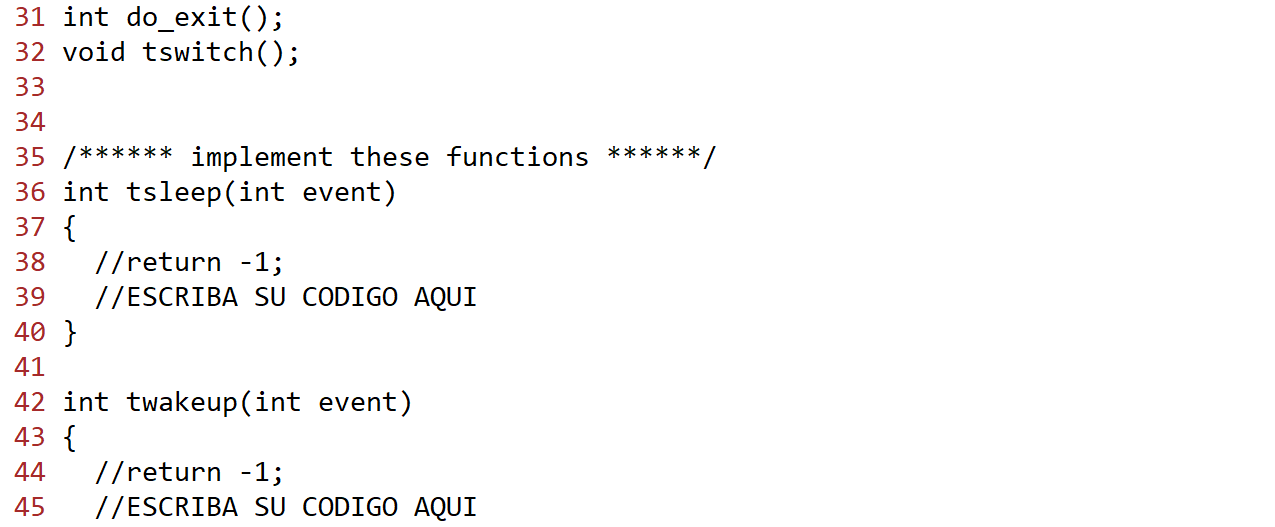


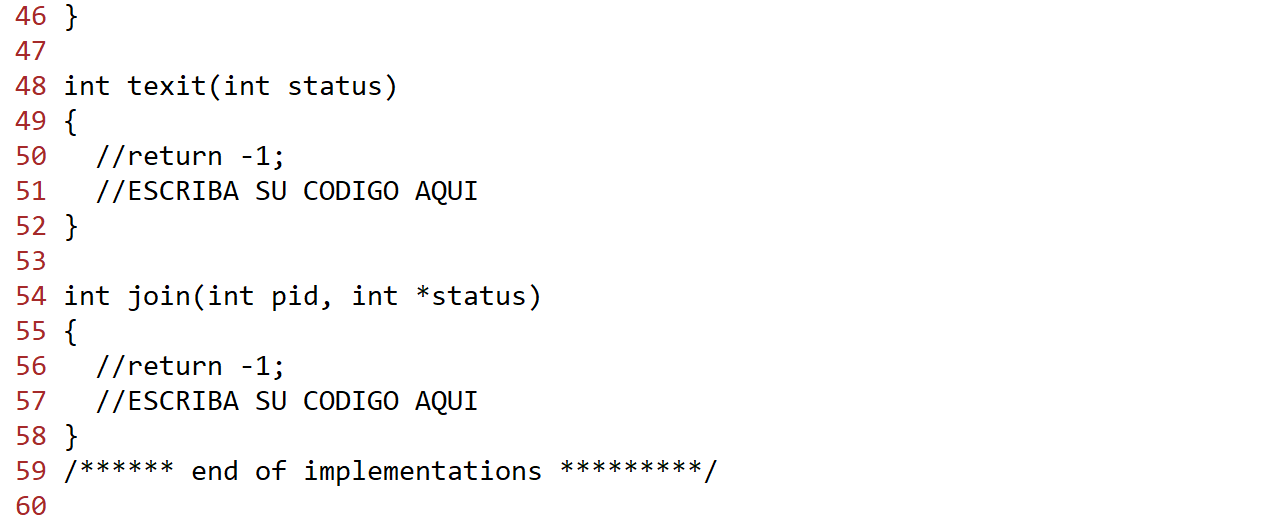


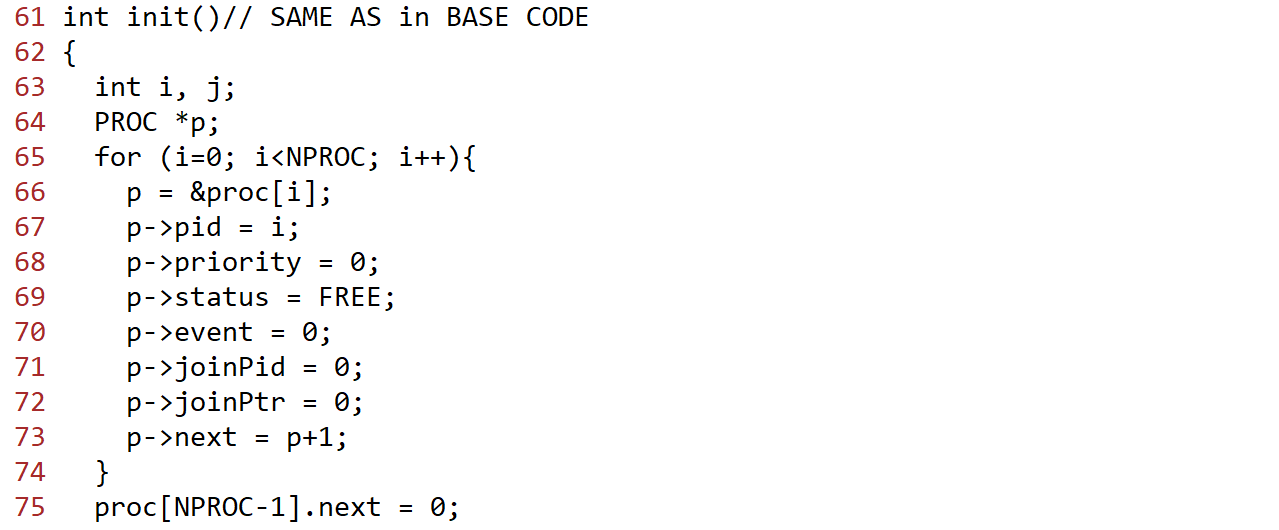


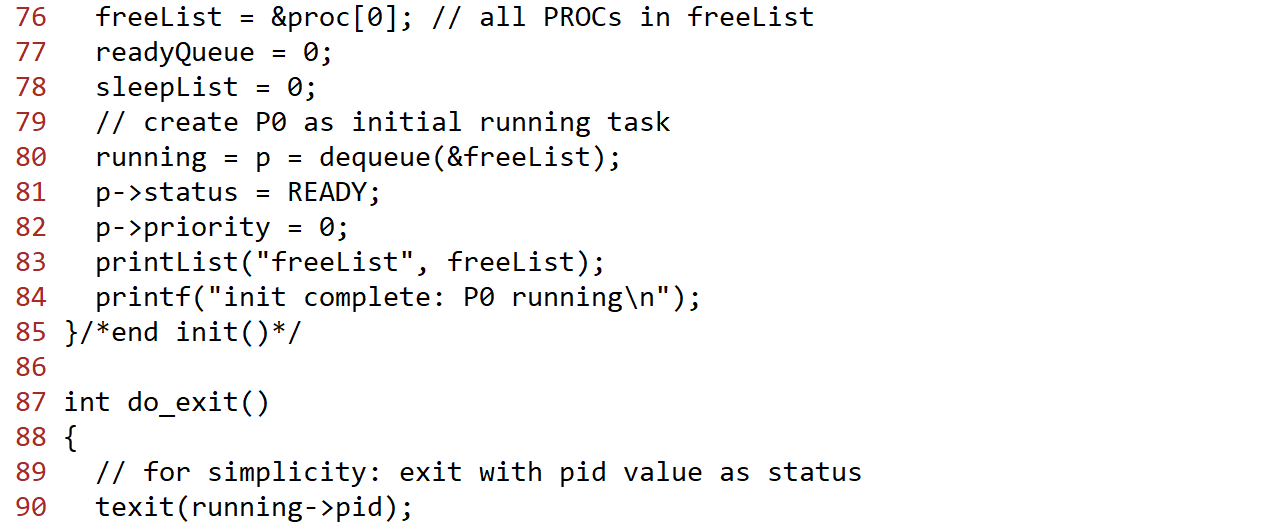


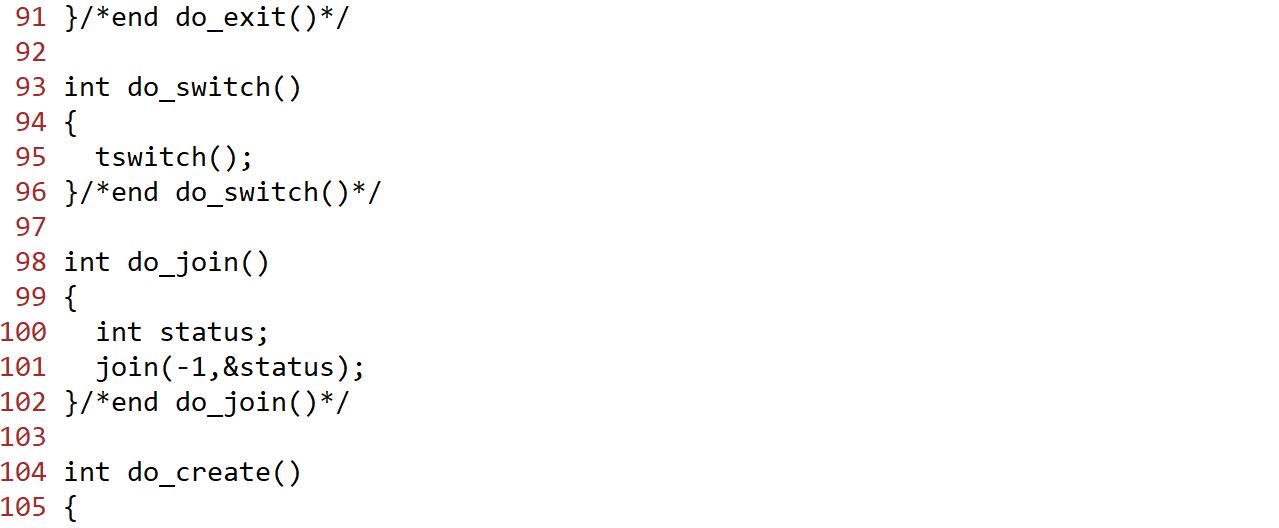


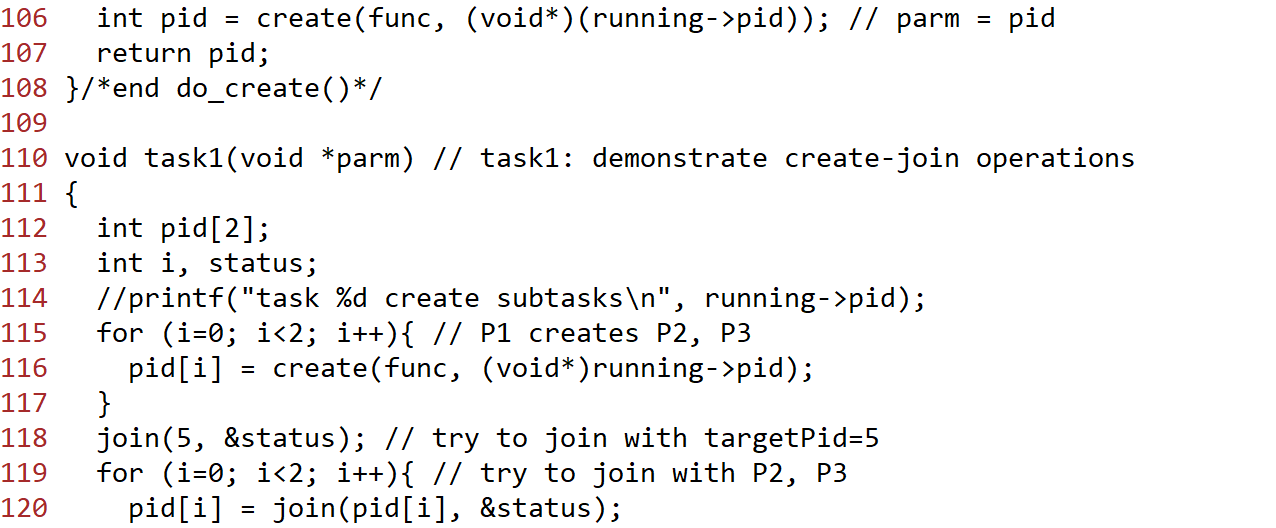


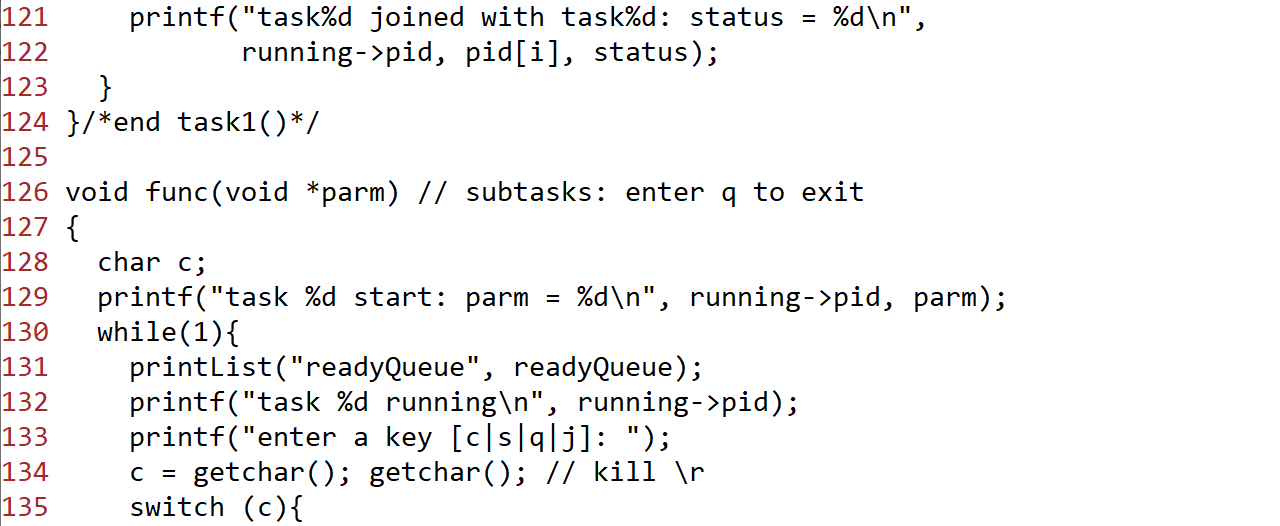


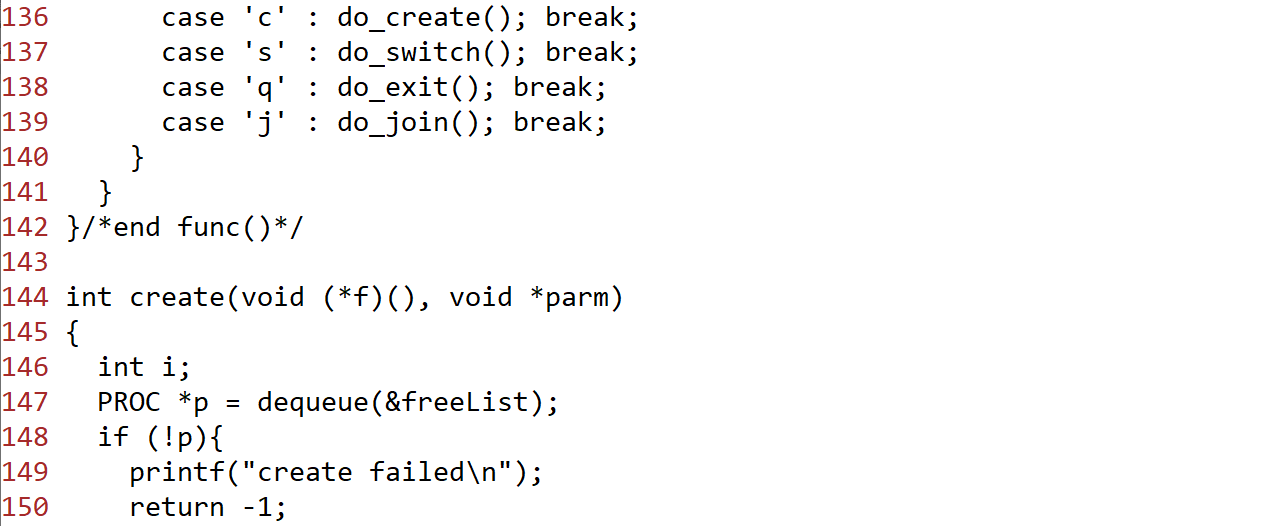


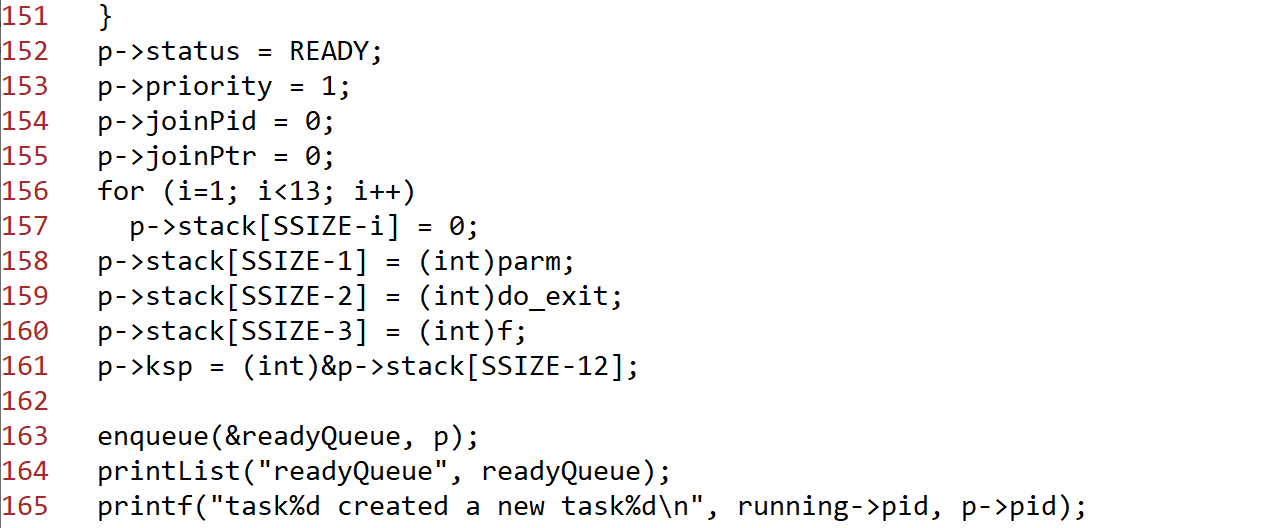


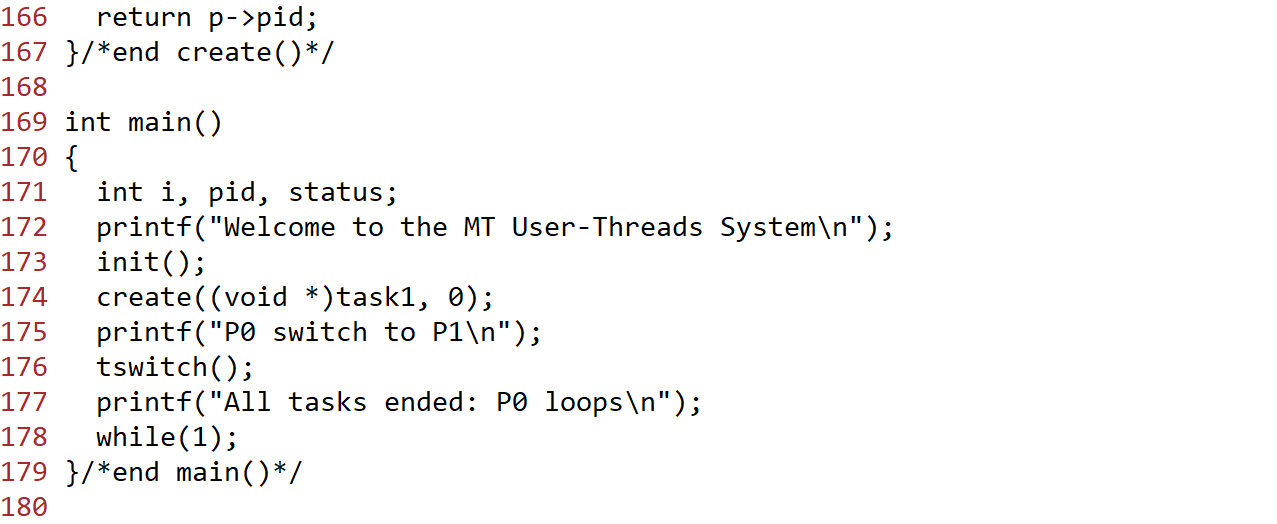


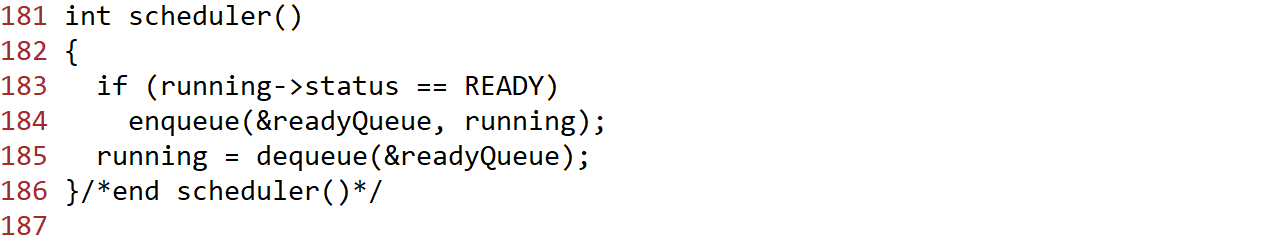


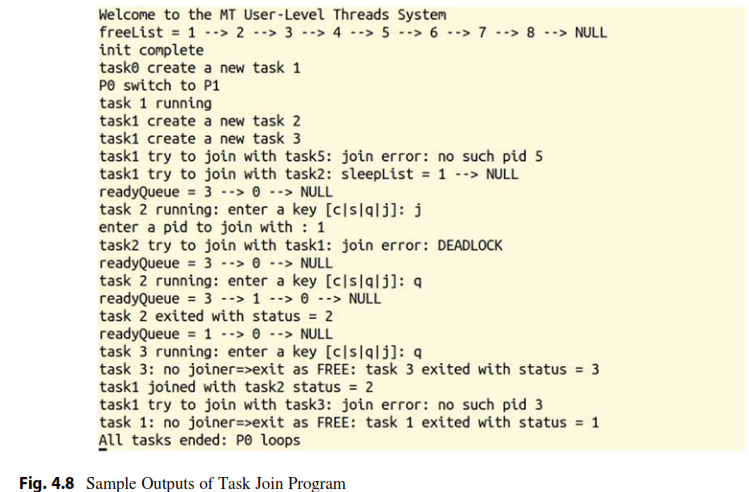


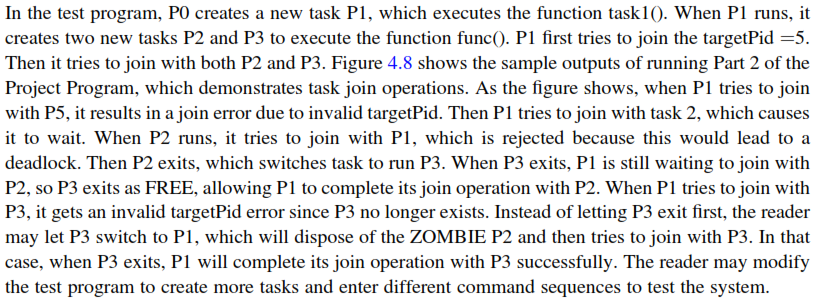










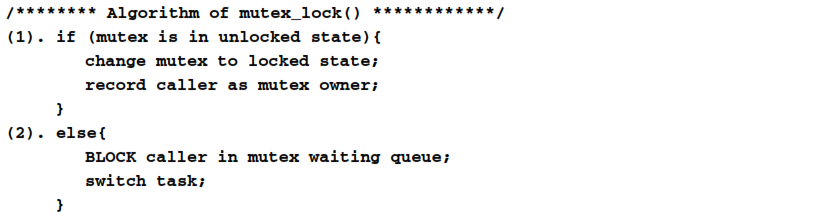


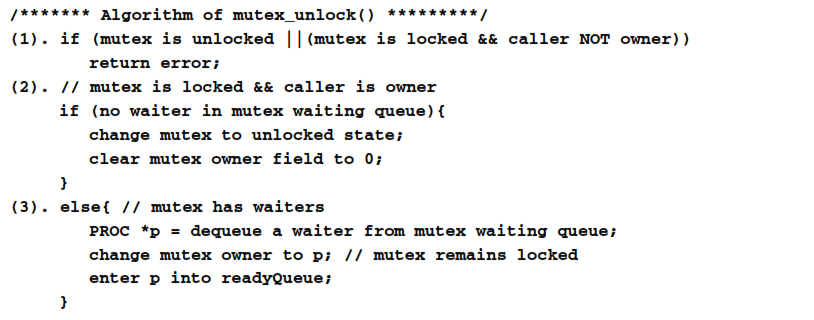
# Implementación de operaciones Mutex

## PARTE 3 del Proyecto de Programación:

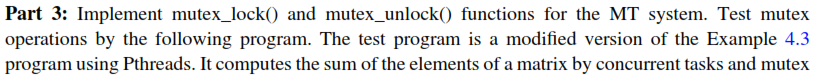
Extender la parte 2 del proyecto para dar soporte a operaciones mutex. Las extensiones se describen a continuación.

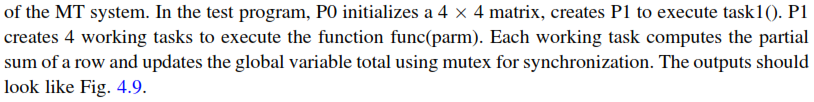


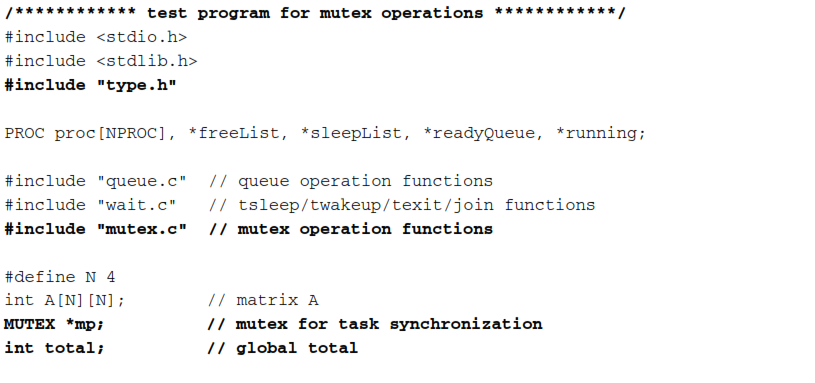


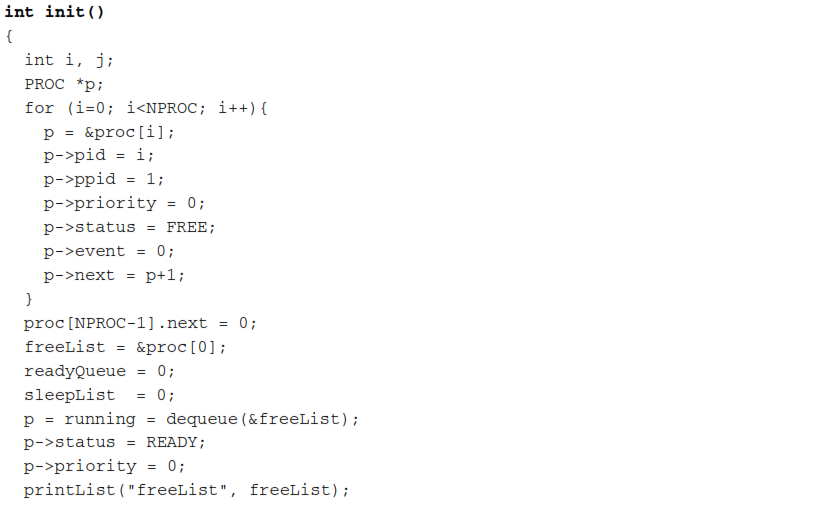


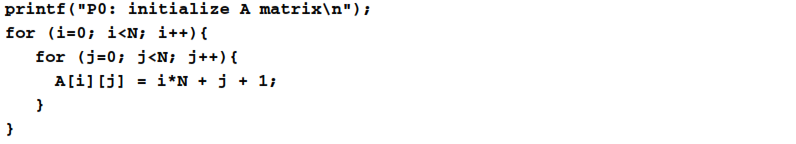
Poner a prueba el proyecto con Mutex usando Programas Concurrentes

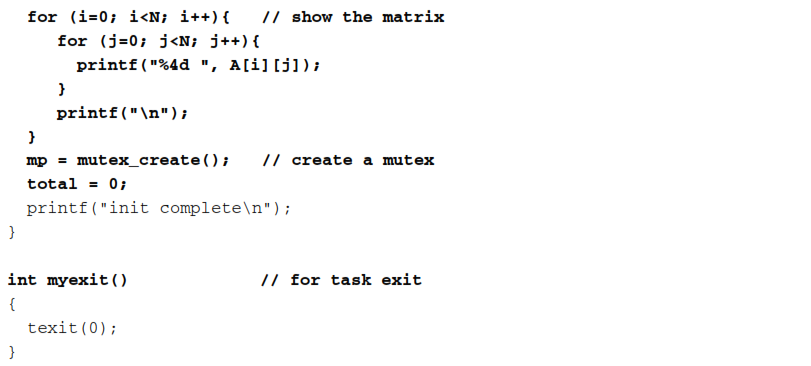


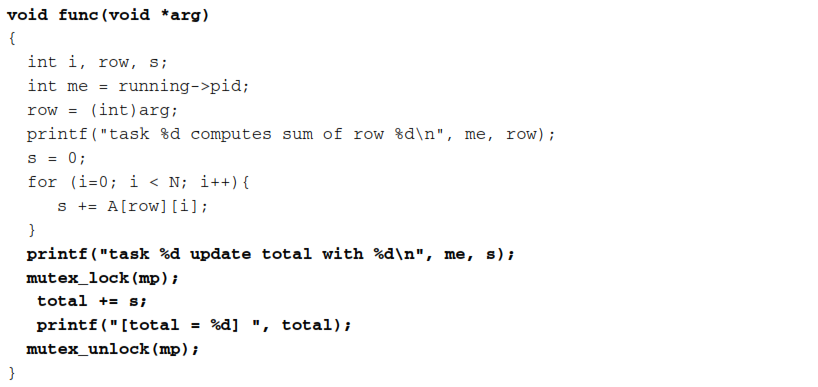


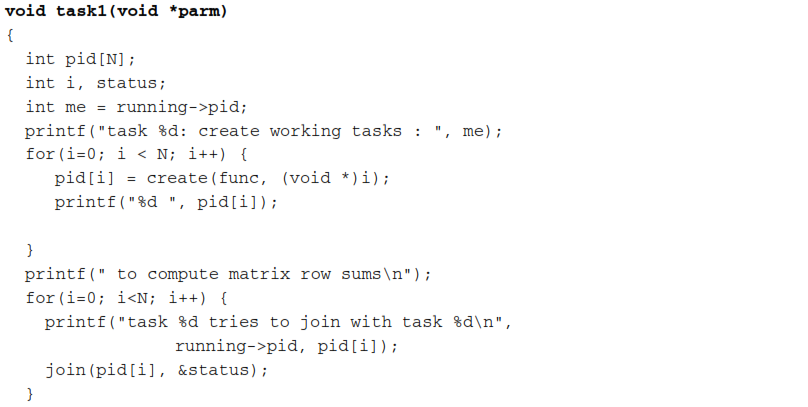


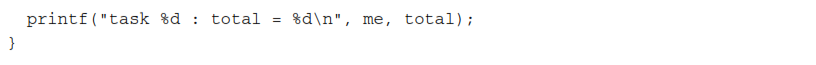


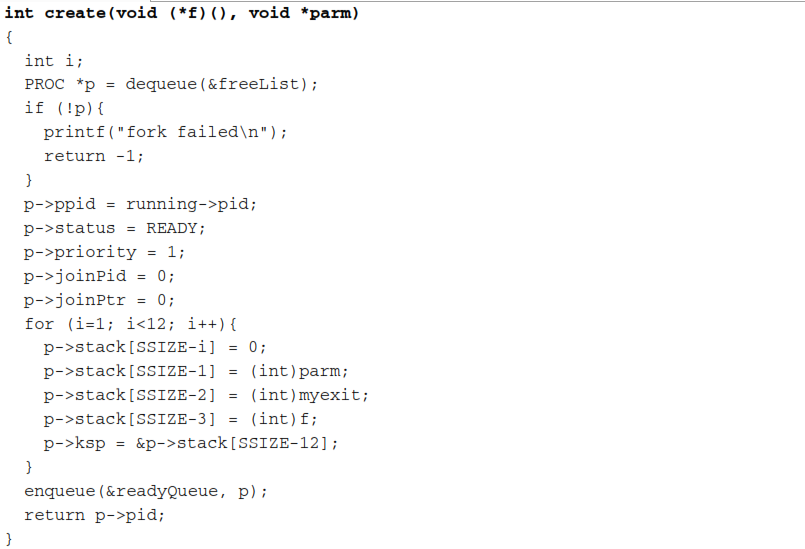


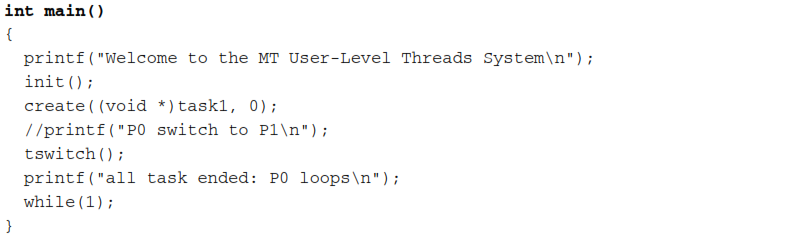


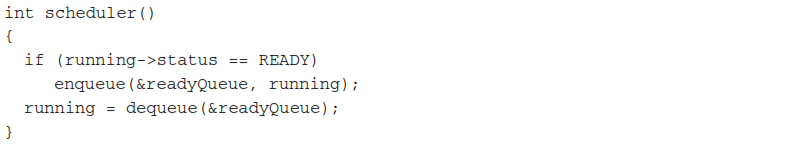








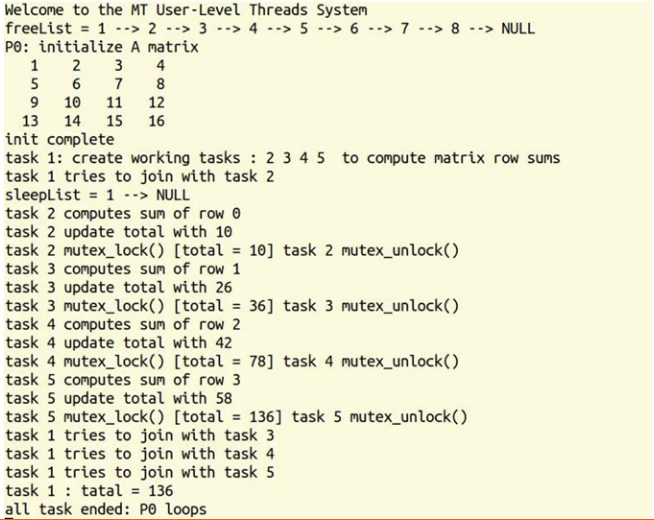




Archivo t.c para la PARTE 3 del proyecto de programación concurrente

Threads a Nivel de Usuario





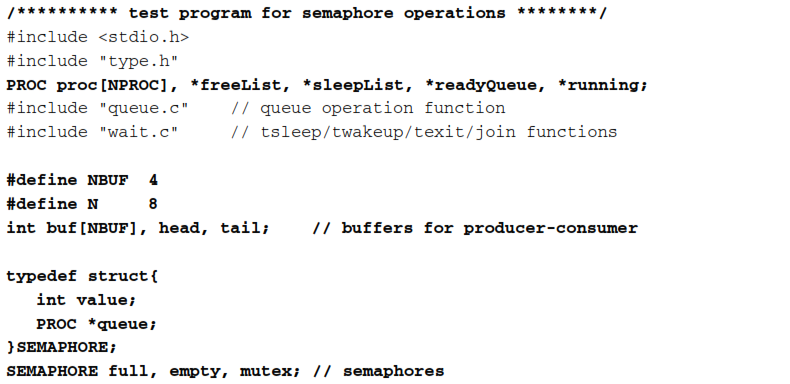


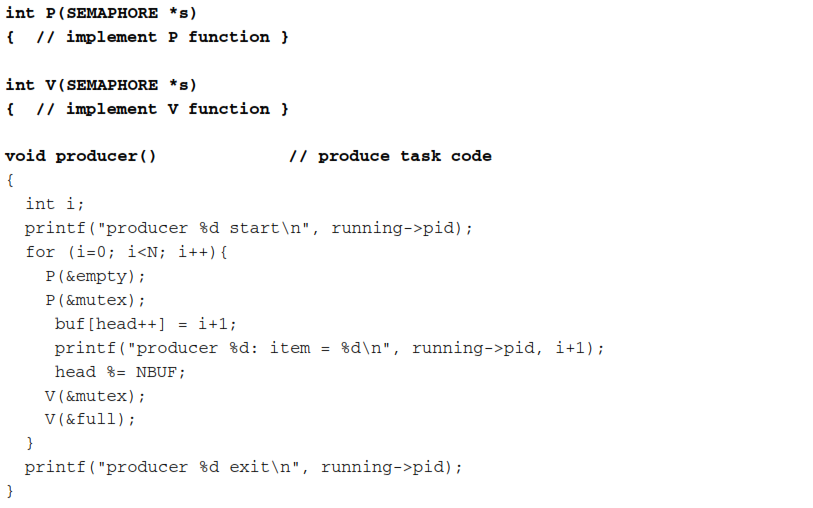
# Implementación de semáforos

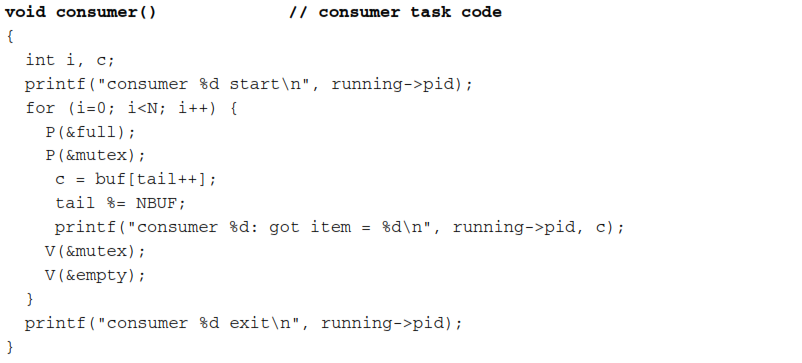
## PARTE 4 del proyecto de programación:

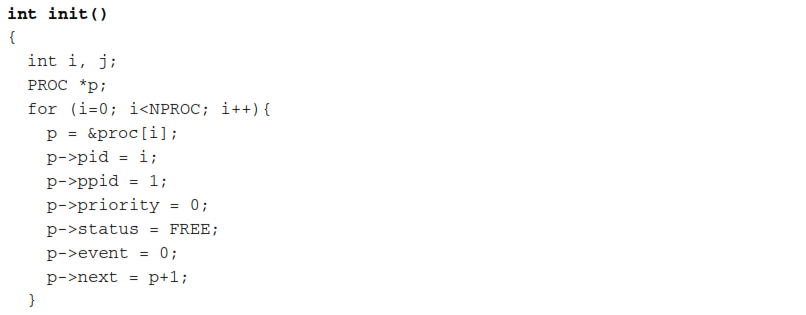
Implementar semáforos contadores como se describió en la sección Semáforos (páginas 13, 14 y 15 de este documento). Demostrar el funcionamiento de los semáforos implementados con el siguiente programa de prueba, el cual es el problema productor-consumidor del ejemplo producer\_consumer\_semaphores.c (página 16 de este documento) adaptado al sistema MT.

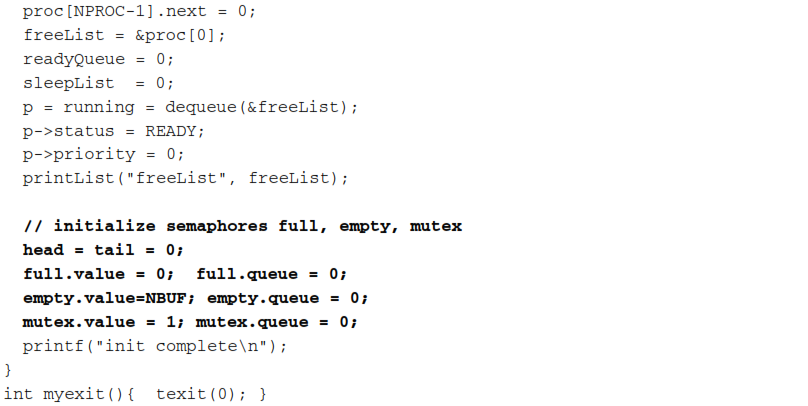
## Problema Productor-Consumidor usando Semáforos

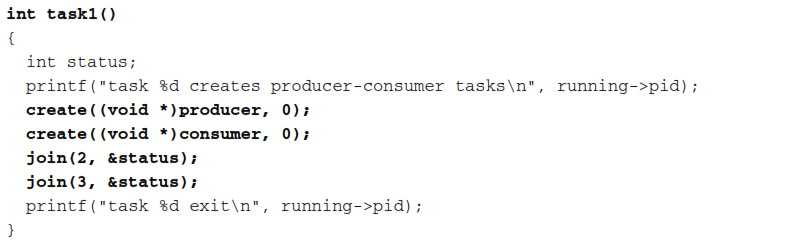


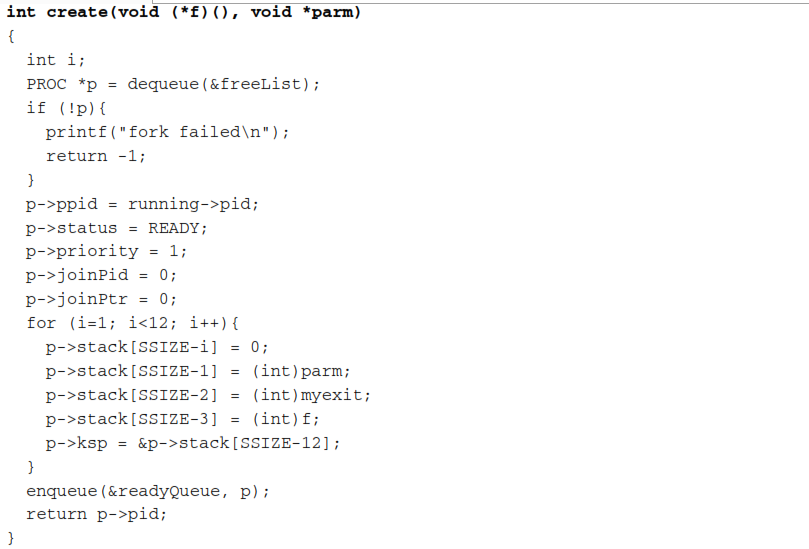


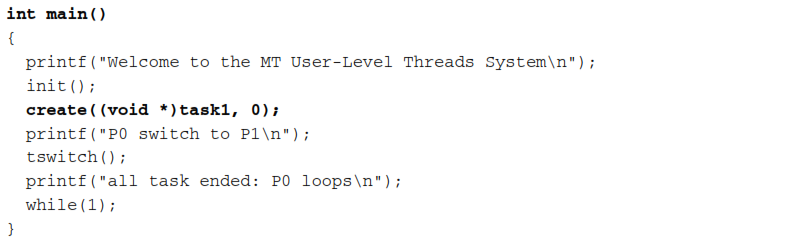


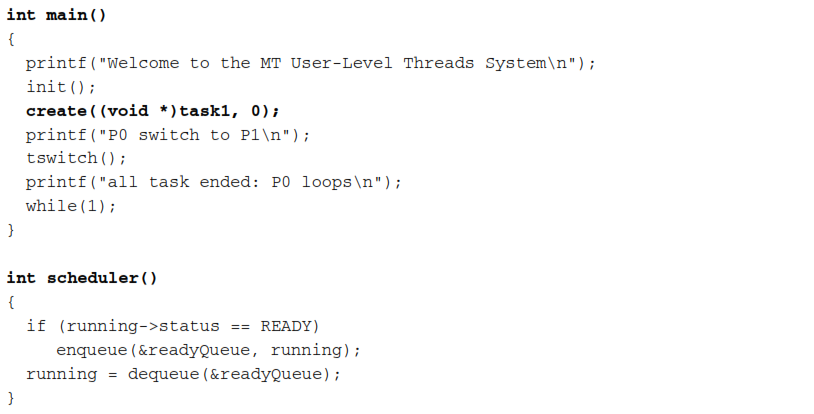








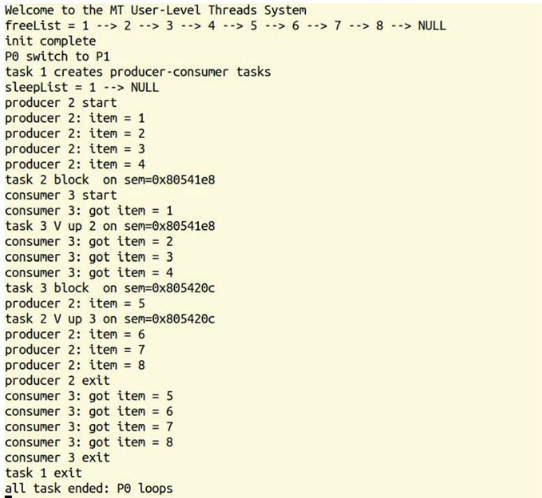




Archivo t.c de la parte 4 del proyecto de programación concurrente

Threads a Nivel de Usuario







# Resumen

En el proyecto de programación descrito en este documento se explican los principios de hilos (threads) y de procesos. Se presentan las operaciones de threads usando la biblioteca Pthreads a través de ejemplos. Éstas incluyen funciones de administración de threads, las herramientas de sincronización de threads, a saber, mutex, join, variables de condición y barreras. Se introduce programación concurrente usando threads a través de ejemplos detallados. Entre los que se incluyen, cálculos con matrices, solución de sistemas de ecuaciones lineales usando threads concurrentes. Se explica el problema de deadlock y se muestra cómo prevenir deadlocks en programas concurrentes. Se contemplan los semáforos y se muestran sus ventajas sobre las variables de condición. También se explica una forma de soportar threads en Linux. El proyecto de programación es implementar threads a nivel de usuario. Se incluye un sistema base que se debe usar como punto de partida para desarrollar el proyecto. El sistema base soporta creación dinámica, ejecución y terminación de tareas concurrentes, las cuales son equivalentes a threads ejecutándose en el mismo espacio de direcciones de un proceso. El proyecto tiene como objetivos que el lector implemente la operación join de threads, mutex y semáforos para sincronización de threads, y que demuestre su uso en programas concurrentes. Se espera que el proyecto de programación le permita al lector tener una comprensión más profunda de los principios y técnicas de multitareas (multitasking), sincronización de threads y programación concurrente.