Sistemas operativos en tiempo real

# Unidad temática 2 Programación concurrente

# Práctica: Sincronización de procesos

## Parte 1

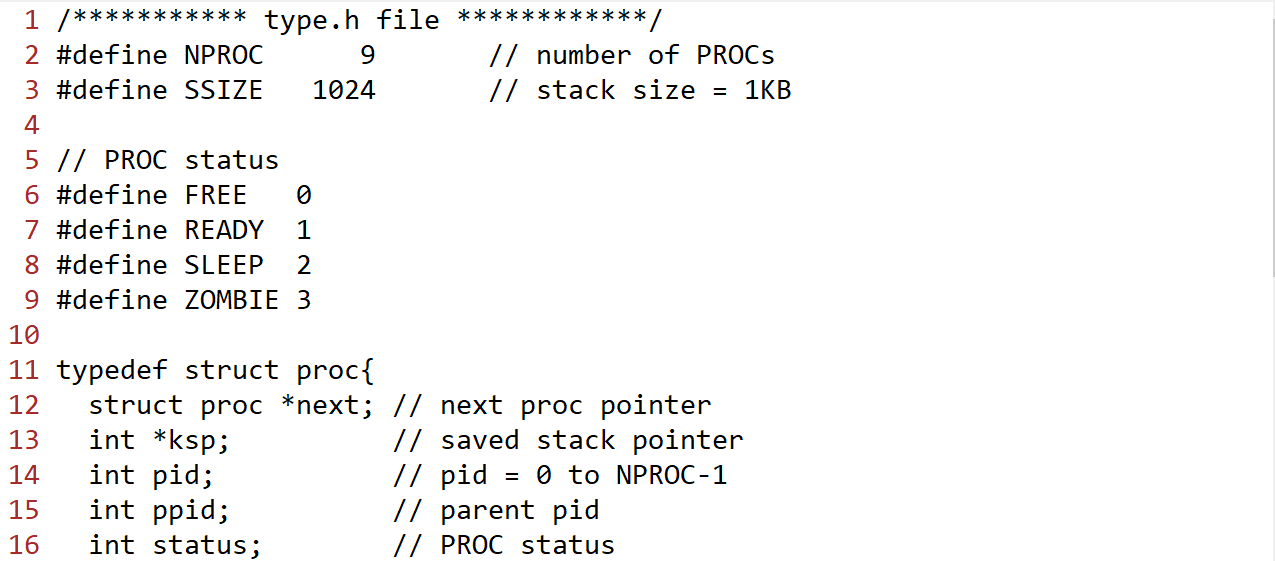
## Construcción de un sistema multitarea

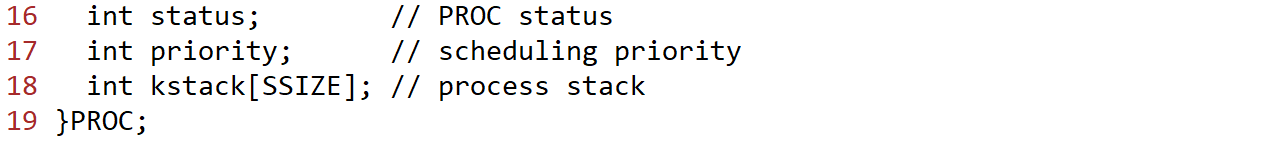
La totalidad de esta práctica está basada en el capítulo 3 del libro Systems Programming in Unix/Linux, del Prof. K. C. Wang, editorial Springer, 2018.

Para tener un mejor entendimiento de los sistemas multitarea y los procesos, en esta práctica se propone iniciar con un ejemplo de programación, el cual está diseñado para ilustrar los principios de la multitarea, cambios de contexto, y procesos. El programa implementa un ambiente multitarea, el cual simula las operaciones en modo kernel de un sistema operativo. El sistema multitarea, denotado por MT, consiste de los siguientes componentes.

### Archivo type.h

El archivo type.h define constantes del sistema y una estructura PROC simple para representar procesos.

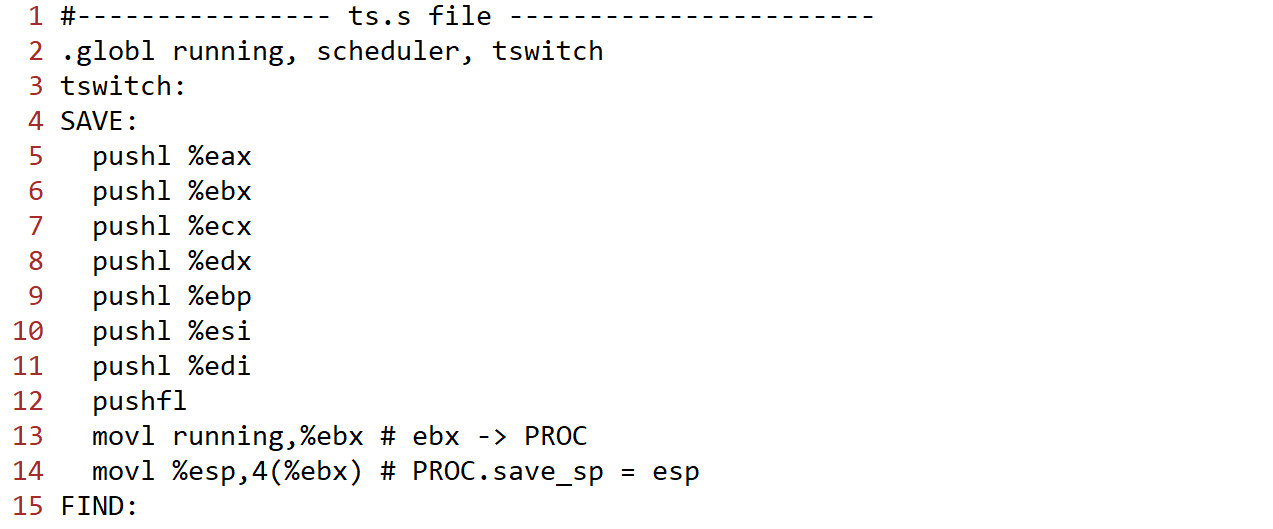


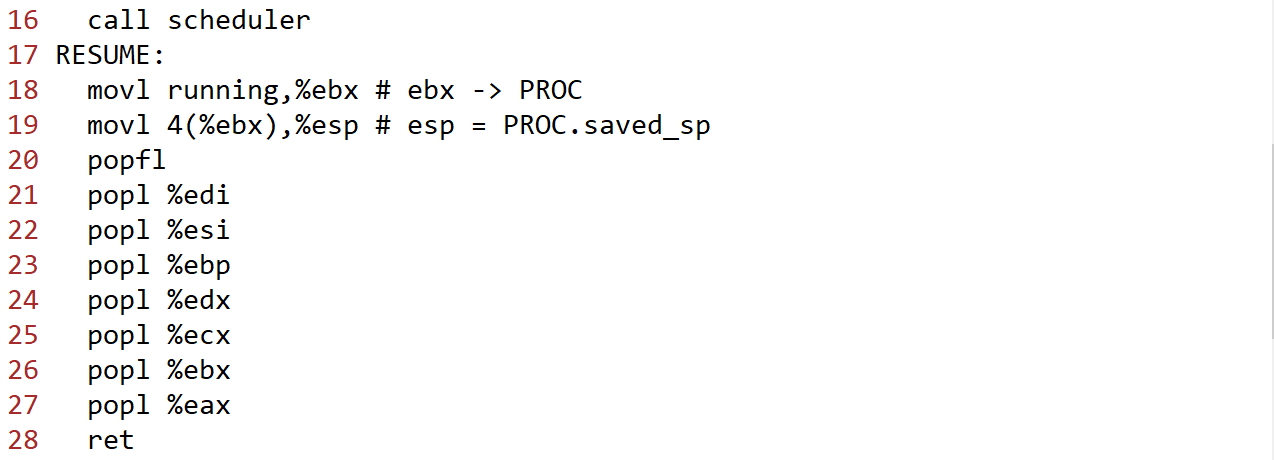


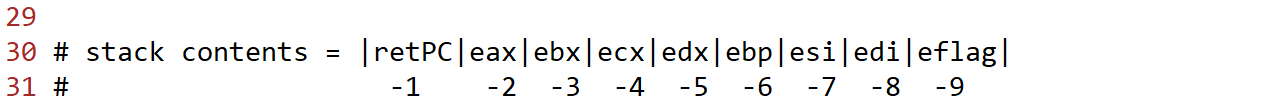
A medida que se expande el sistema MT, más tarde se agregarán más campos a la estructura PROC.

### El archivo ts.s

El archivo ts.s implementa cambios de contexto en código ensamblador para ser compilado usando GCC.

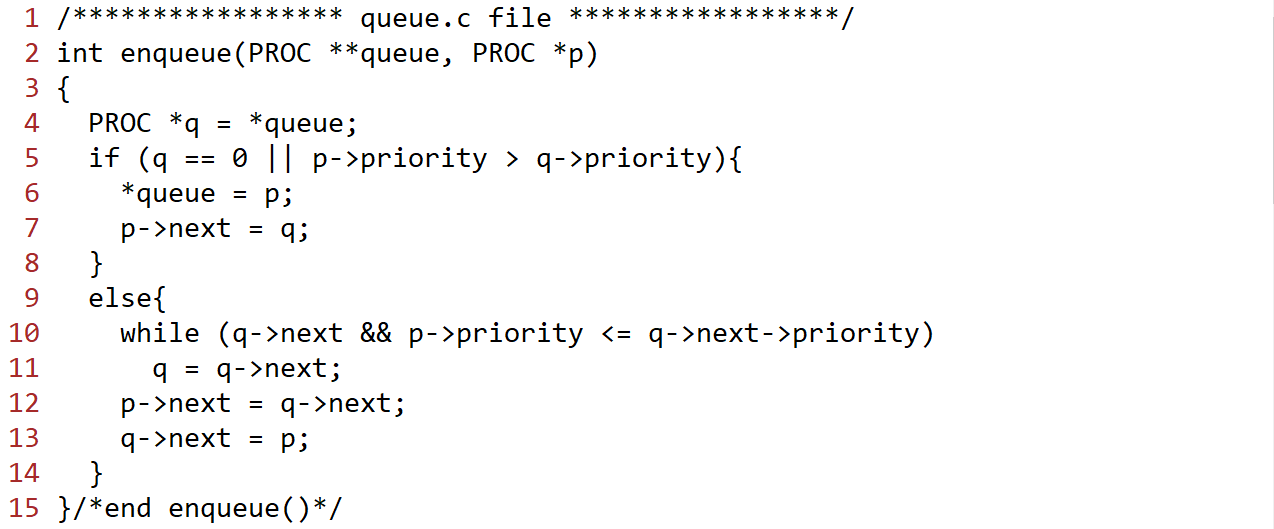


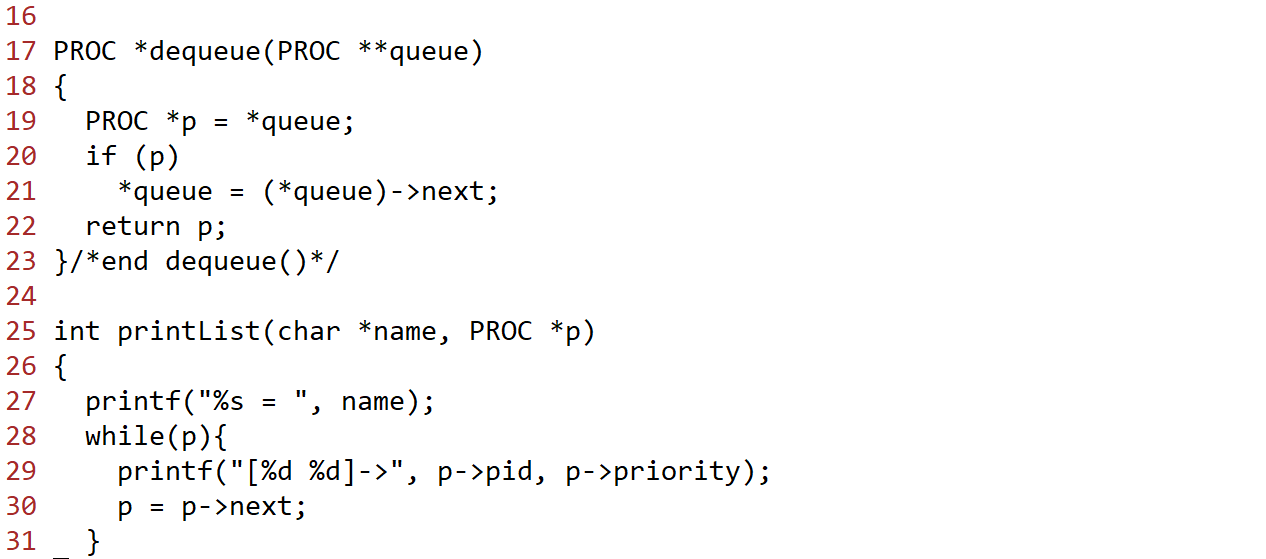




### El archivo queue.c

El archivo queue.c implementa funciones para operaciones con colas y con listas. La función enqueue() introduce una estructura PROC en una cola por prioridad. En una cola por prioridad, las estructuras PROC de la misma prioridad son ordenados de acuerdo a una política FIFO (First In First Out). La función dequeue() devuelve el primer elemento removido de una cola o una lista enlazada. La función printList() imprime los elementos de una lista enlazada.

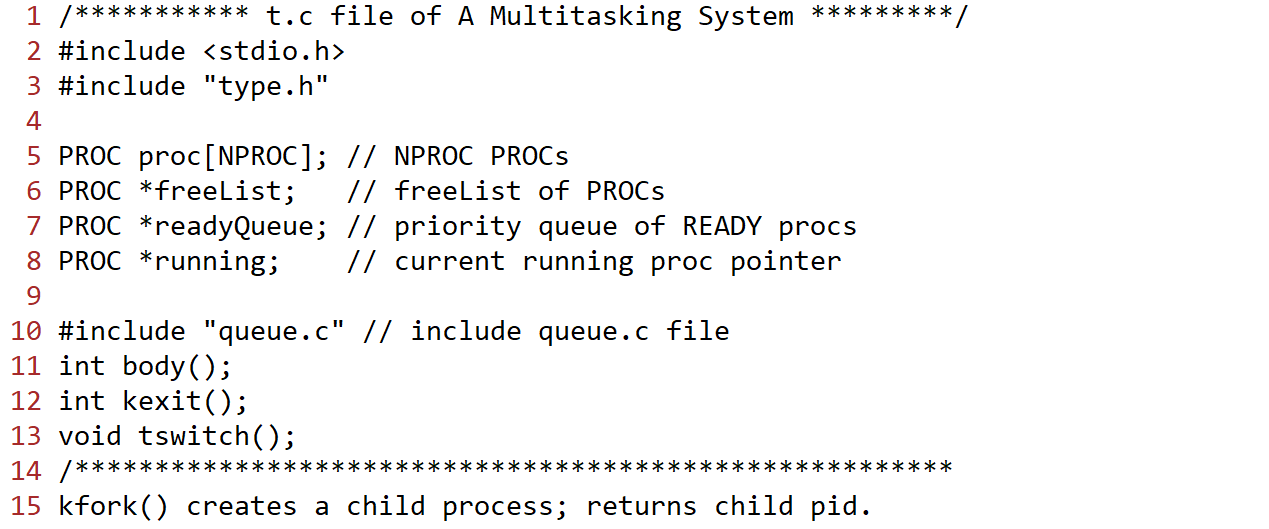


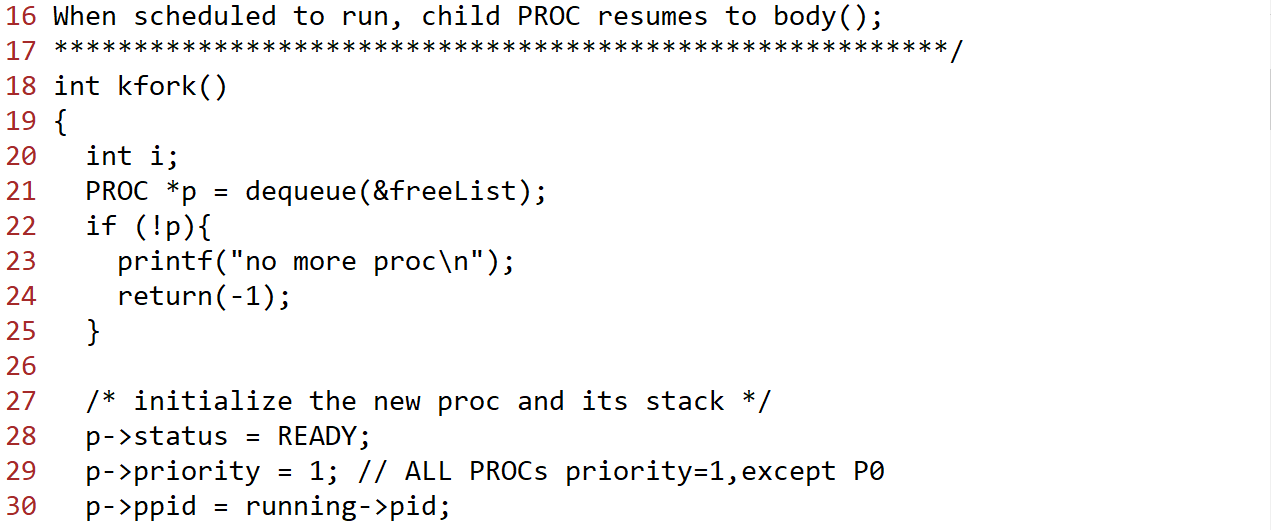


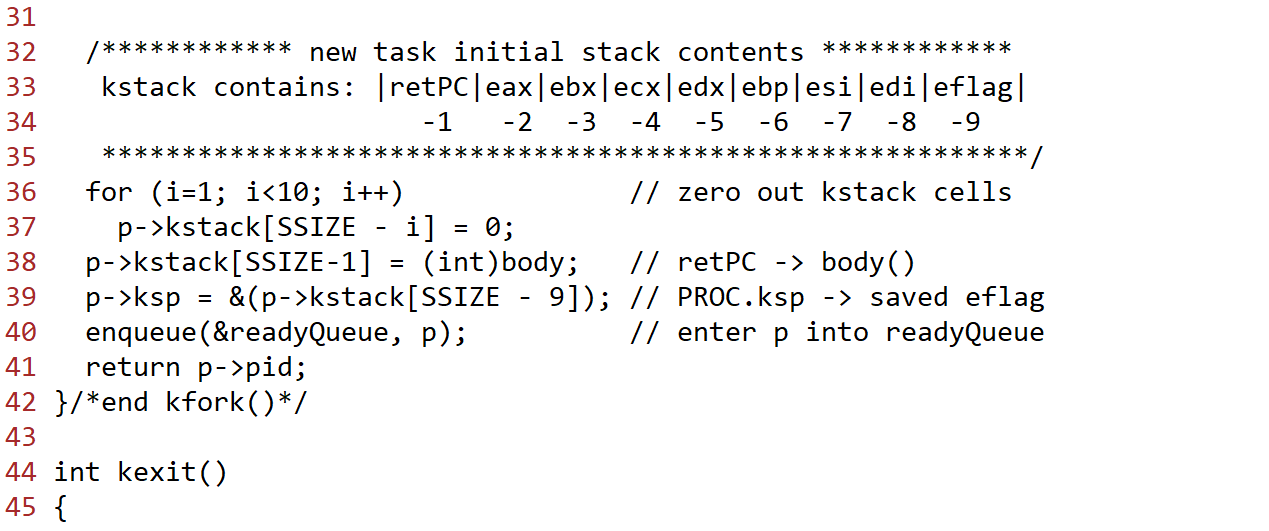


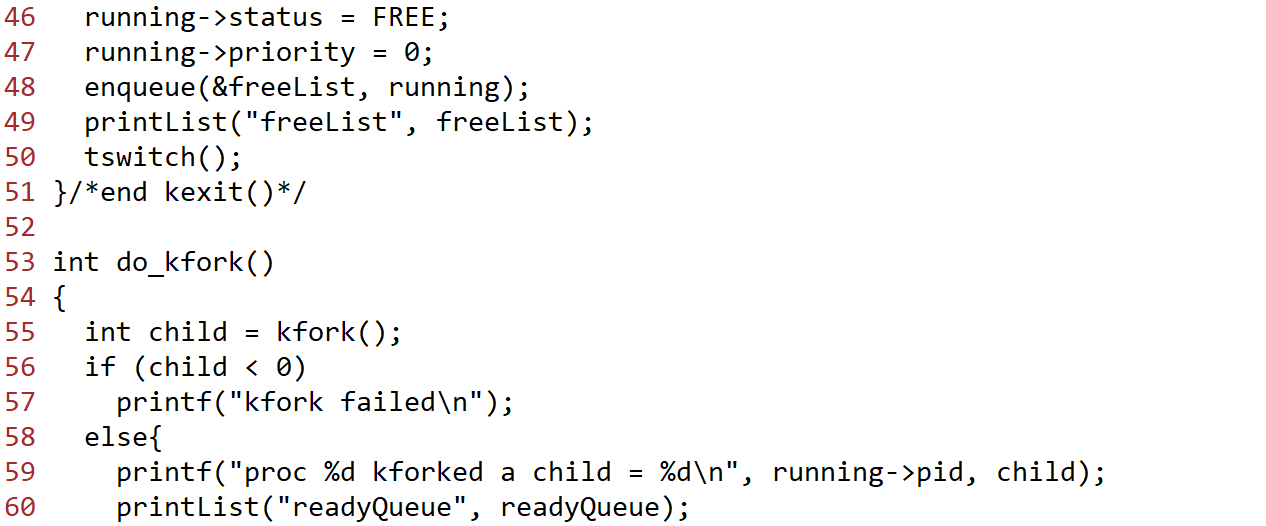
### El archivo t.c

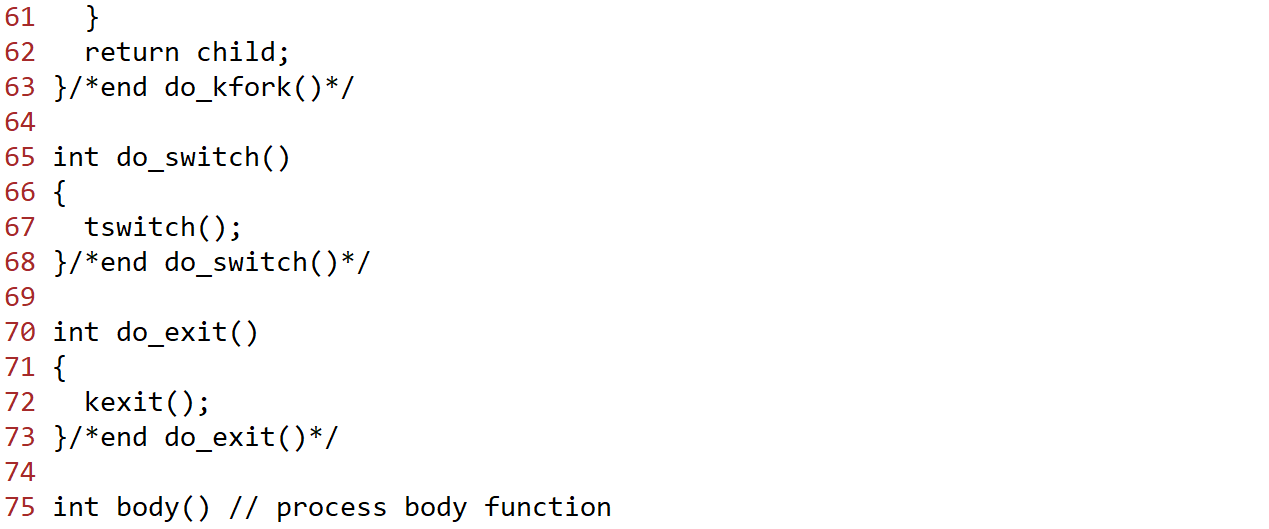
El archivo t.c define las estructuras de datos del sistema MT, el código de inicialización del sistema y las funciones de administración de procesos.

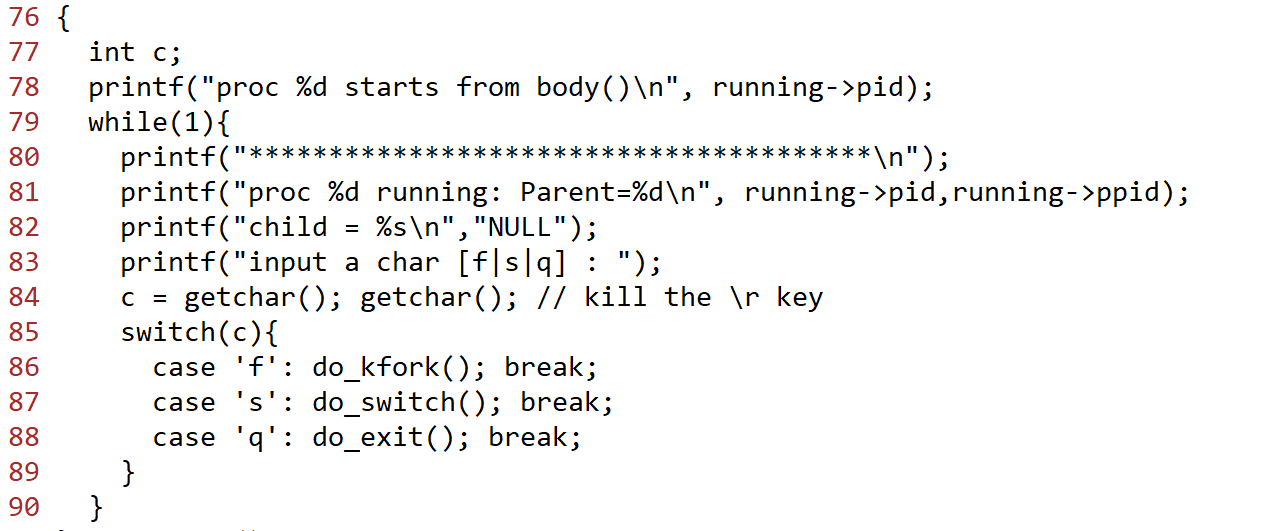


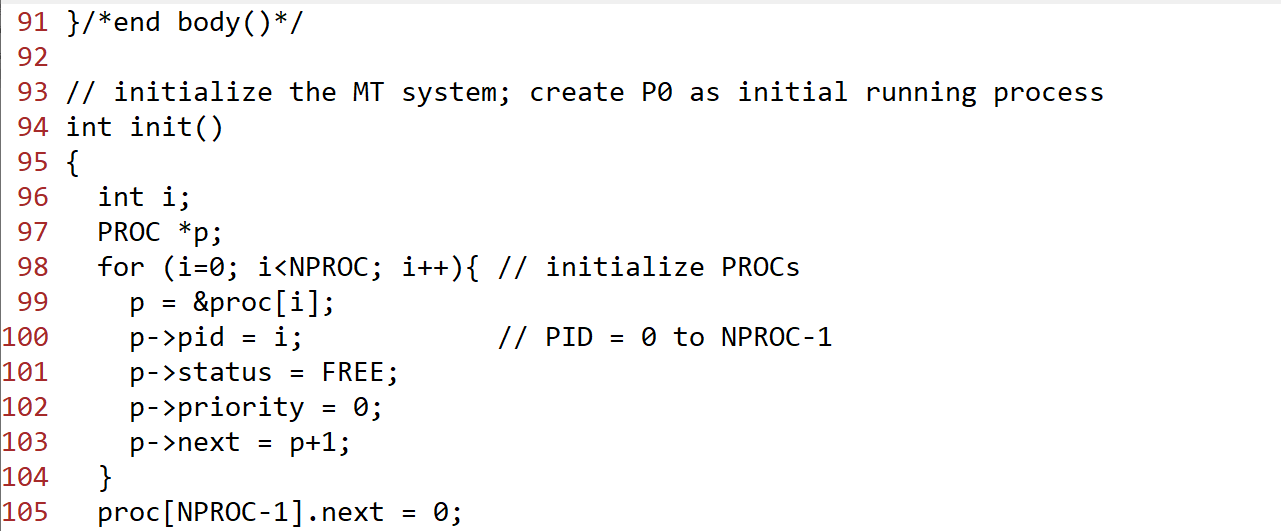


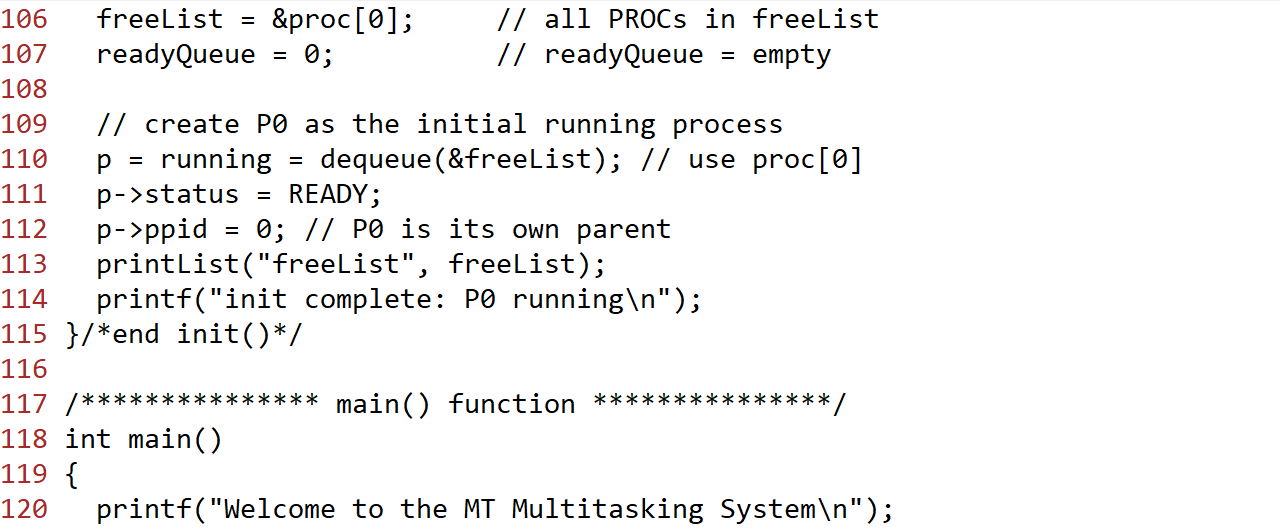


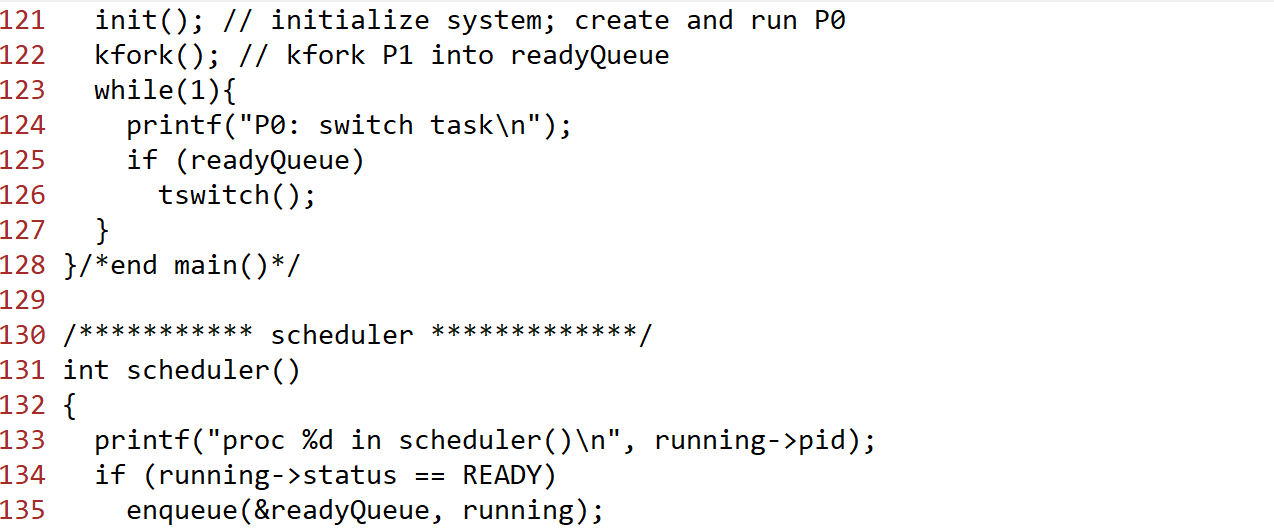


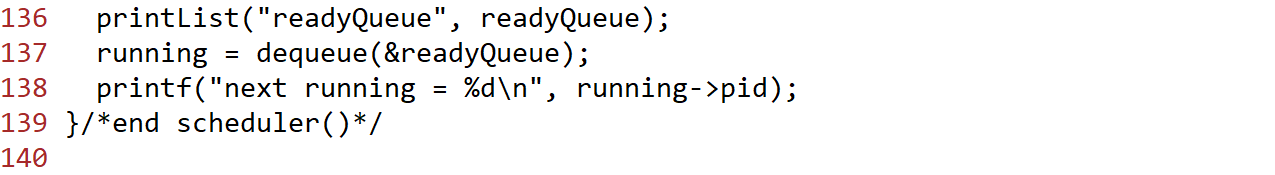












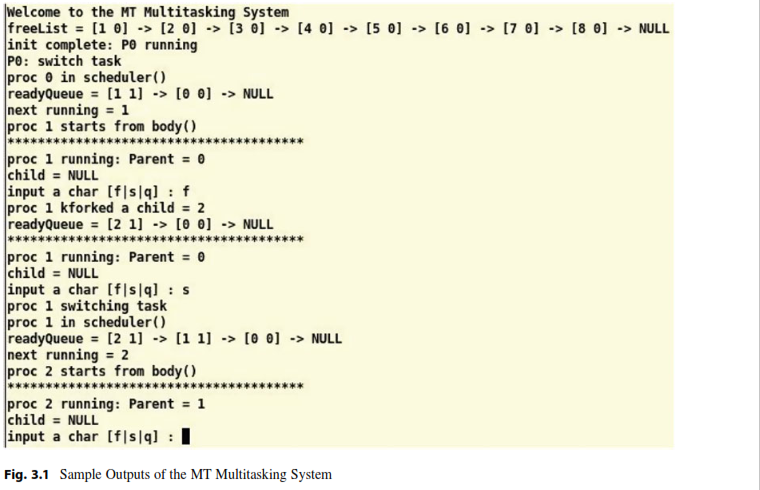
La explicación del funcionamiento del sistema multitarea MT puede ser consultada en la página 108 del libro Systems Programming in Unix/Linux.

### Ejecución del sistema multitarea MT

En un sistema GNU Linux el sistema se puede construir con el comando



Con este comando se compila y se enlaza el sistema MT. El resultado del comando anterior es un archivo a.out. Para ejecutar el archivo a.out debemos ejecutar el comando ./a.out; la siguiente figura muestra algunas líneas de salida de muestra de una ejecución del sistema multitarea MT. La figura muestra la inicialización del sistema, el proceso inicial P0, el cual crea P1 y hace un cambio de contexto para poner a P1 en ejecución. P1 hace kfork para crear su proceso hijo P2 y hace cambio de contexto para poner a P2 en ejecución. Mientras un proceso está en ejecución, el usuario del sistema puede introducir alguno de los comandos: f, s, o q para probar la ejecución del sistema.



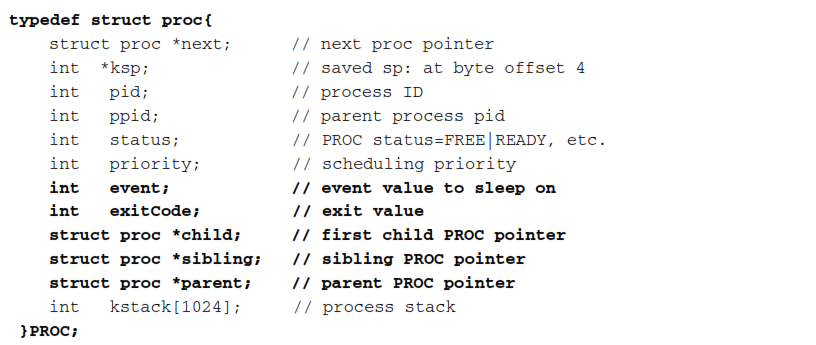
## Parte 2

## Sincronización de procesos

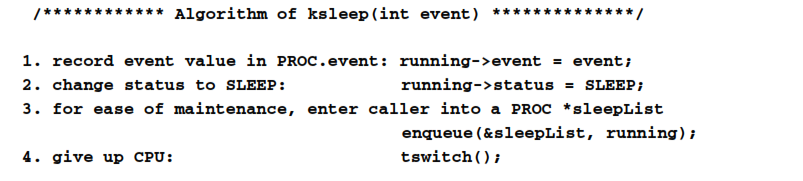
Un sistema operativo contiene muchos procesos concurrentes, los cuales pueden interactuar unos con otros. La sincronización de procesos res refiere a las reglas y los mecanismos usados para controlar y coordinar las interacciones de los procesos para asegurar que se ejecuten apropiadamente. Las herramientas más simples para sincronización de procesos son las operaciones sleep y wakeup.

### La operación sleep

Siempre que un proceso necesita algo, VyG. un área de memoria para uso exclusivo, un carácter de entrada desde stdin, etc. que actualmente no está disponible, el proceso se va a dormir sobre un valor de evento, el cual representa la razón para dormir. Para implementar la operación sleep, podemos agregar un campo event a la estructura PROC e implementar una función ksleep(int event), la cual le permite a un proceso ir a dormir. A continuación, supondremos que la estructura PROC ha sido modificada para incluir los campos mostrados en negritas.



El algoritmo de ksleep() es

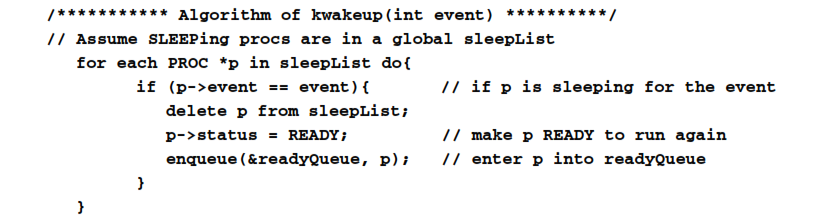


Dado que el proceso que está empezando a dormir no está en la cola ReadyQueue, esté no será ejecutable hasta que sea despertado por otro proceso. Así que después de ponerse a símismo a dormir, el proceso llama a tswitch() para ceder la CPU.

### La operación wakeup()

Varios procesos podrían dormir sobre el mismo evento, lo cual es natural dado que todos ellos podrían necesitar el mismo recurso, VyG. una impresora, la cual podría estar ocupada actualmente. En ese caso, todos esos procesos se irían a dormir sobre el mismo valor de evento. Cuando un evento por el cual se está esperando ocurre, otra entidad de ejecución, la cual podría ser un proceso o un manejador de interrupción, realizará la llamada kwakeup(event), la cual despierta a todos los procesos que están durmiendo sobre el valor event. Si ningún proceso está durmiendo sobre el evento, kwakeup() no tiene efecto, esto es, no hace nada.

El algoritmo de kwakeup() es



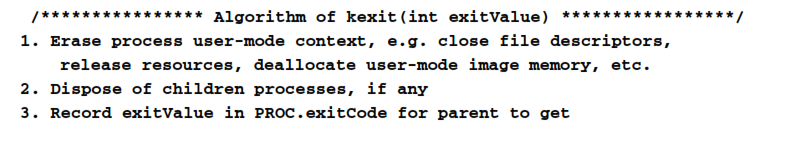
Se debe notar que un proceso despertado podría no correr inmediatamente. Solamente es puesto en la cola readyQueue, a esperar su turno para correr. Cuando un proceso despertado corre, este debe tratar de obtener el recurso otra vez si estaba tratando de obtener un recurso antes de irse a dormir. Esto es debido a que el recurso podría ya no estar disponible para el momento en que el proceso logre correr. Las funciones ksleep() y kwakeup() son para sincronización de procesos en general, pero también son usadas en el caso especial de sincronizar procesos que son padre e hijo, lo cual es nuestro siguiente tópico.

### Terminación de proceso

Un proceso podría terminar de dos posibles formas:

* Terminación normal: el proceso llama a exit(value), lo cual en su turno hace la llamada al sistema \_exit(value) para ejecutar kexit(value) en el kernel del sistema operativo, el cual es el caso que estamos contemplando aquí.
* Terminación anormal: el proceso termina anormalmente debido a una señal.

En cualquier caso, cuando un proceso termina, eventualmente llama a kexit() en el kernel del sistema operativo. El algoritmo general de kexit() es como sigue:





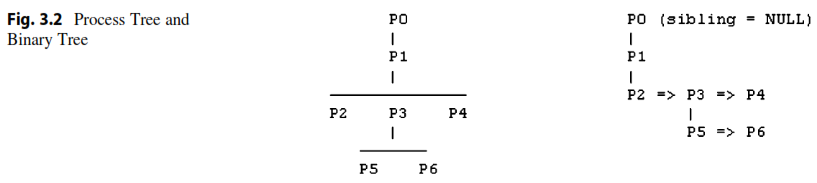
Todos los procesos en el el sistema MT corren en el modo kernel simulado de un sistema operativo. Como tal, no tienen un contexto de modo de usuario. Así que comenzamos nuestra discusión en el paso 2 de kexit(). En algunos sistemas operativos, el ambiente de ejecución de un proceso podría depender del de su padre. Por ejemplo, el área de memoria del hijo podría estar dentro de la del padre, así que el proceso padre no puede morir a menos que todos sus hijos hayan muerto. En Unix/Linux, los procesos tienen apenas una relación vaga de padre-hijo pero sus ambientes de ejecución son todos independientes. Entonces, en Unix/Linux un proceso puede morir en cualquier momento. Si un proceso con hijos muere antes que sus hijos, todos los procesos hijo no tendrían padre, esto es, se convierten en huérfanos. Entonces, la pregunta es: ¿qué hacer con esos huérfanos? Inmediatamente se hace claro que debe haber un proceso que no debe morir si aun existen otros procesos. En todos los sistemas tipo Unix, el proceso P1, el cual es también conocido como el proceso INIT, es elegido para jugar este rol. Cuando un proceso muere, envía todos sus hijos huérfanos, vivos o muertos, a P1, esto es, esos procesos se convierten en hijos de P1. Siguiendo ese modelo, también designaremos a P1 en el sistema MT como el proceso INIT. Entonces, P1 no debe morir si aún existen otros procesos. El problema restante es cómo implementar el paso 2 de kexit() eficientemente. Para que un proceso que está muriendo, se desentienda de los hijos huérfanos, el proceso debe ser capaz de determinar si él tiene algún hijo, y si su hijo tiene hijos, encontrar a todos los hijos (todos los niños) rápidamente. Si el número de procesos es pequeño, VyG solo unos cuantos como en el sistema MT, ambas preguntas pueden ser contestadas efectivamente buscando todas las estructuras PROC. Por ejemplo, para determinar si un proceso tiene algún hijo, simplemente buscar en todas las estructuras PROC por alguna que no tenga estado FREE y que su ppid coincida con el pid del proceso. Si el número de procesos es grande, VyG en el orden de cientos e incluso miles, este esquema de búsqueda simple sería demasiado lento para ser aceptable. Por esta razón, la mayoría de los kernel de sistema opetrativo grandes rastrean las relaciones de los procesos manteniendo un árbol de familia de procesos.

### Árbol de familia de procesos

Típicamente, el árbol de familia de procesos es implementado como un árbol binario por un par de apuntadores, child y sibling en cada estructura PROC, como en



donde child apunta al primer hijo de un proceso y sibling apunta a una lista de otros hijos del mismo padre. Por conveniencia, cada estructura PROC también usa un apuntador parent para apuntar a su padre. Como un ejemplo, el árbol de procesos mostrado en el lado izquierdo de la Figura 3.2 puede ser implementado como el árbol binario mostrado en el lado derecho de la misma Figura, en el cual, cada enlace vertical es un apuntador child y cada enlace horizontal es un apuntador sibling. Por claridad no se muestran los apunatdores parent y os apuntadores null.



Con un árbol de procesos, es mucho más fácil encontrar los hijos de un proceso. Primero, hay que seguir el apuntador child para encontrar la primera estructura PROC del primer hijo del proceso. Después hay que seguir los apuntadores sibling para recorrer las estructuras PROC sibling. Para enviar todos los hijos a P1, simplemente hay que desenganchar los hijos y anexarlos a la lista de hijos del proceso P1 (y cambiar su ppid y apuntador parent también).

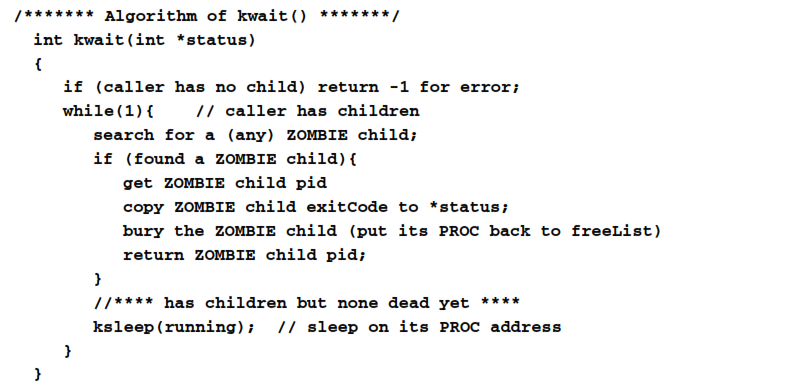
Cada estructura PROC tiene un campo exitCode, el cual es el valueExit del proceso cuando este termina. Después de registrar el exitValue en PROC.exitCode, el proceso cambia su estado (status) a ZOMBIE pero no libera la estructura PROC. Entonces el proceso llama a kwakeup(event) para despertar a su padre, donde event debe ser el mismo valor único usado por ambos el proceso padre y el proceso hijo, VyG la dirección de memoria de la estructura PROC del proceso padre o el pid del padre. Éste también despierta a P1 si le ha enviado algunos huérfanos a P1. El acto final del proceso que está muriendo es llamar a tswitch() por última vez. Después de eso, el proceso está esencialmente muerto pero aún tiene un cuerpo muerto en la forma de una estructura PROC ZOMBIE, la cual será desaparecida (establecida a FREE) por el proceso padre a través de la operación wait.

### Wait para pa terminación del proceso hijo

En algún momento, un proceso podría llamar a la función de kernel



Por un proceso hijo ZOMBIE. Si tiene éxito, el pid devuelto es el pid de un hijo ZOMBIE y el estatus (\*status) contiene el exitCode del hijo ZOMBIE. Además, kawait() tambié libera la estructura PROC del hijo ZOMBIE y la manda de regreso a la lista freeList para su reutilización. El algoritmo de kwait() es



En el algoritmo de kwait(), la función devuelve -1 para señalar un error si el proceso no tiene hijos. En caso contrario, busca un hijo ZOMBIE. Si encuentra un hijo ZOMBIE, colecta el pid y el exitCode del hijo ZOMBIE, libera y envía la estructura PROC del hijo ZOMBIE a freeList y devuelve el pid del hijo ZOMBIE. En caso contrario, se va a dormir sobre su propia dirección de memoria de PROC, esperando a que algún hijo termine. Dado que cada dirección de memoria de estructura PROC es un valor único es cual es también conocido por todos los procesos hijos, un padre puede esperar sobre su propia dirección de PROC para que un hijo lo despierte más tarde. Correspondientemente, cuando un proceso termina, éste debe llamar a



para despertar al padre. En lugar de la dirección de memoria de la estructura PROC del padre, se puede verificar que es posible usar el pid del padre. En el algoritmo de kwait(), cuando el proceso se despierta, encontrará un hijo muerto cuando ejecuta el ciclo while otra vez. Note que cada llamada a kwait() maneja solamente un hijo ZOMBIE, si hay alguno. Si un proceso tiene muchos hijos, podría tener que llamar a kwait() varias veces para desentenderse de todos los hijos muertos. Alternativamente, un proceso podría terminar primero sin esperar por algún hijo muerto. Cuando u proceso muere, todos sus hijos se convierten en hijos de P1. En un sistema real, P1 ejecuta un ciclo infinito, en el cual repetidamente espera hijos muertos, incluyendo huérfanos adoptados. Por lo tanto en un sistema tipo Unix, el proceso INIT P1 hace varias cosas:

-Es el ancestro de todos los procesos, excepto de P0. En particular, es el abuelo de todos los procesos de usuario dado que todos los procesos login son hijos de P1.

-Es la cabeza de todo un orfanato, dado que todos los huérfanos son enviados a su casa y le llaman papá.

-Es el administrador de una morgue, dado que se la pasa buscando para desintegrar sus cuerpos muertos.

Así que, en un sistema tipo Unix, si el proceso INIT P1 muere o se queda pegado (congelado), el sistema dejaría de funcionar porque ningún usuario puede hacer login otra vez y el sistema pronto se llenará de cuerpos derruidos.

## Administración de procesos en el sistema multitarea MT

El siguiente es el ejercicio correspondiente a la parte 2 de esta práctica, éste consiste en hacer un refinamiento del sistema MT base presentado en la parte 1 de esta práctica, para implementar las funciones de administración de procesos para el sistema MT. Específicamente,

1. Implementar el árbol de familia de procesos como un árbol binario.
2. Implementar las funciones ksleep() y kwakeup() para sincronización de procesos.
3. Implementar las funciones kexit() y kwait() para administración de procesos.
4. Agregar un comando ‘w’ para probar y demostrar la operación wait.

La Figura 3.3 incluye algunas salidas de muestra de la ejecución del sistema MT modificado. Lo que se muestra en la figura es que P1 tuvo un hijo P2, el cual tuvo sus propios hijos P3 y P4. Entonces, P2 ejecuta el comando ‘q’ para terminar. La figura muestra que P2 se convierte en un ZOMBIE y envía todos sus hijos huérfanos a P1, cuya lista de hijos ahora incluye a todos los huérfanos de P2. Cuando P1 ejecuta el comando ‘w’, encuentra el hijo ZOMBIE P2 y lo devuelve a la lista freeList. Alternativamente, se puede cambiar el orden de hijo-exit y padre-wait para verificar que el orden no importa.

0