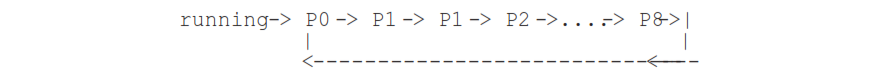
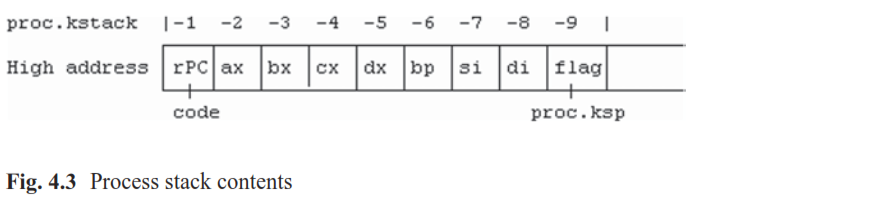
Un Sistema Multitarea Simple

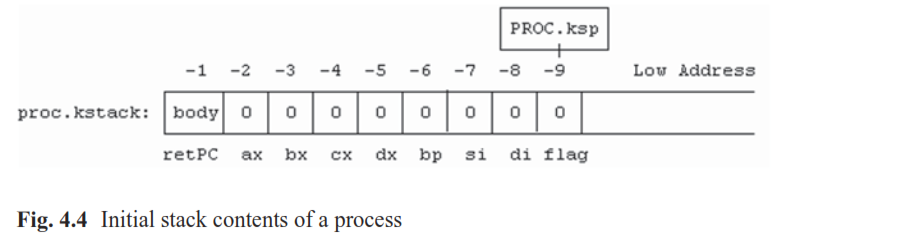
Con cambios de contexto, podemos crear un ambiente multitarea que contenga muchos procesos. En el siguiente programa, denotado por MTX4.1, definimos NPROC = 9 PROC estructuras. (Se justifica la suposición de 9 PROCs en el capítulo 5 cuando se extienden los procesos para tener imágenes de modo usuario). Cada PROC tiene un único número pid para identificación. Las PROCs son inicializadas como sigue



P0 es el proceso de ejecución inicial. Todas las PROCs forman una lista enlazada circular para scheduling de procesos simple. Cada uno de los PROCs, P1 a P8, es inicializado de tal manera que está listo para regresar a ejecución desde una función body( ). Dado que la inicialización de la pila de PROC es crucial, explicamos los pasos en detalle. Aunque los procesos nunca existieron antes, podríamos pretender que ellos no solamente existieron antes, sino que también se ejecutaron antes. La razón porque un PROC no está corriendo ahora es porque llamó a tswitch( ) para ceder la CPU antes. Si es así, el ksp de PROC debe apuntar a su área de pila conteniendo los registros de CPU guardados y una dirección de retorno, como se muestra en la figura 4.3, donde el índice –i significa SSIZE-i.

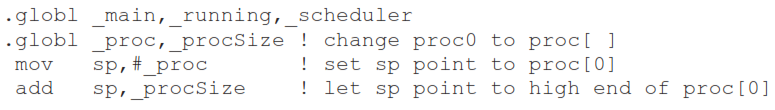


Dado que el PROC nunca realmente corrió antes, podríamos suponer que sus pilas estuvieron inicialmente vacías, así que la dirección de retorno, rPC, está hasta el fondo de la pila. ¿Qué debe ser el valor rPC? Podría apuntar a algún código ejecutable, por ejemplo, la dirección de entrada de una función body( ). ¿Qué hay acerca de los registros guardados? Dado que el PROC nunca corrió antes, los valores de los registros no importan, así que pueden todos ser inicializados a 0. De acuerdo con esto, podemos inicializar cada uno de los PROCs, P1 a P8, como se muestra en la figura 4.4.

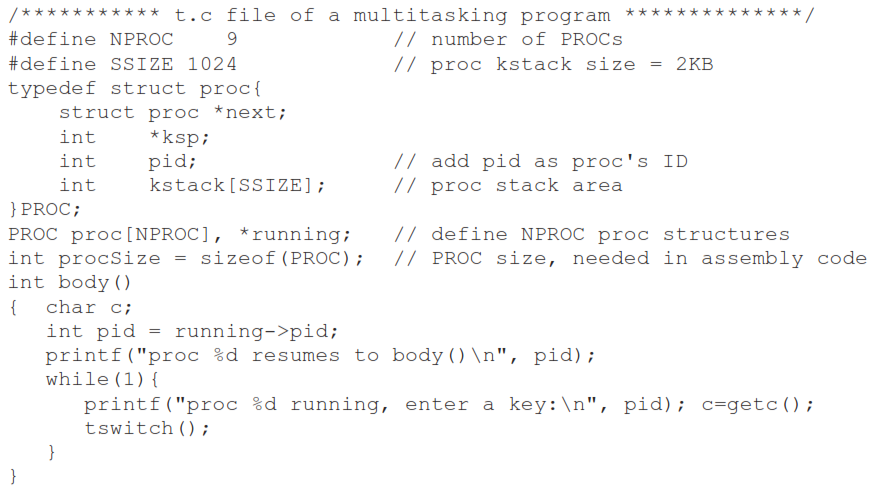


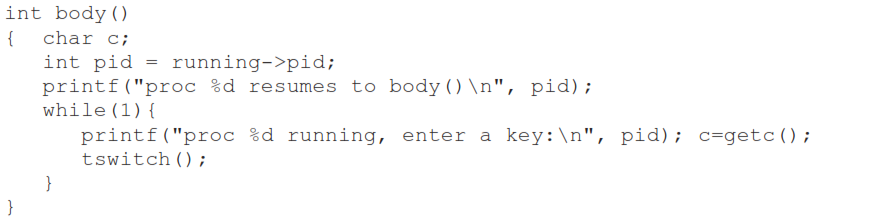
Con este setup, cuando un PROC se convierte en running, i.e. cuando running apunta a la estructura PROC, este ejecutaría la parte RESUME de tswitch( ), lo cual restaura los registros de CPU “guardados”, seguido por RET, causando que el proceso ejecute la función body( ).

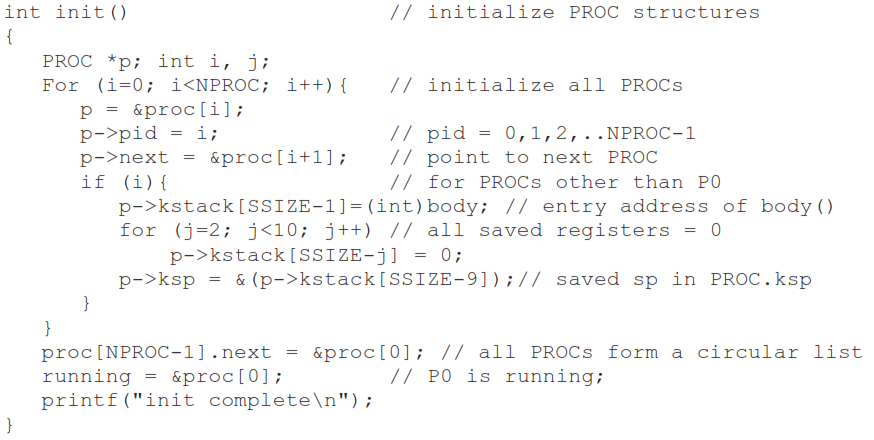
Después de la inicialización, P0 llama a tswitch( ) para switchear de proceso. En tswitch( ), P0 guarda los regsitros de la CPU en su propia pila, guarda el apuntador de pila en su PROC.ksp y llama a scheduler( ). La función scheduler( ) simplemente cambia running a running->next. Así P0 switchea a P1. P1 comienza ejecutando la parte RESUME de tswitch( ), causando que regrese a ejecutar la función body( ). Mientras está en body( ), el proceso corriendo imprime su pid y pide un carácter de entrada. Entonces llama a tswitch( ) para switchear al siguiente proceso, etc. Dado que los PROCs están en una lista circular, tomarán turnos para correr. El código de ensamblador es el mismo que antes, excepto por el apuntador de pila inicial, el cual es establecido al valor de kstack de proc[0] cuando la ejecución empieza, como en

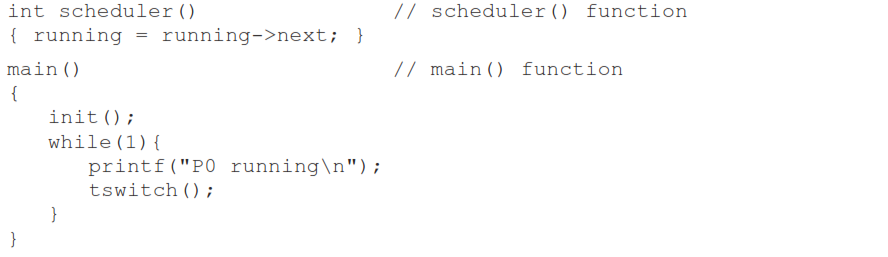


Lo siguiente lista el código C del programa multitarea.

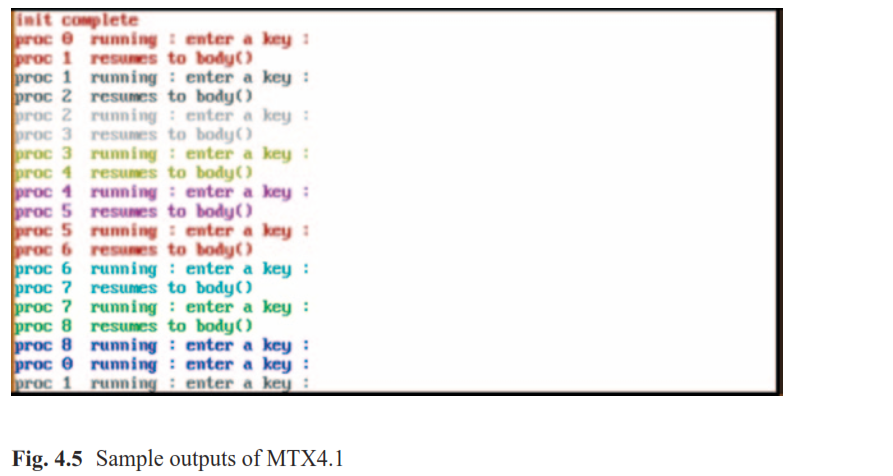








El lector podría compilar el programa MTX4.1 para generar una imagen bootable. Entonces bootear y correr el programa sobre ya sea una PC real o una PC virtual para observar su comportamiento de tiempo de ejecución. Mientras el sistema corre, cada tecla de entrada causa un switcheo de proceso. Usa el identificador de proceso para desplegar líneas en colores diferentes, solo por diversión. La figura 4.5 presenta las salidas muestra de ejecución de MTX4.1.

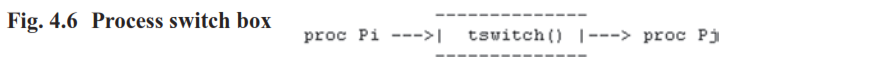


El lector podría modificar el código C de MTX4.1 para producir diferentes efectos. Por ejemplo, agregar sentencias printf( ) a la función scheduler( ) para identificar los PROCs durante un cambio de contexto. Después de la inicialización, en lugar de llamar a tswitch( ), permitir que P0 llame a la función body(), etc.

Note que ninguno de los procesos, P1 a P8, realmente llama a body(). Lo que hemos hecho es convencer a cada proceso de que él llamó a tswitch() para ceder la CPU justo antes de entrar al cuerpo de la función body(), y ahí es a donde regresará cada uno de ellos cuando empiecen su ejecución. Entonces, podemos establecer el ambiente de ejecución inicial de un proceso para controlar su curso de acciones. El proceso no tiene otra opción más que obedecer. Este es el poder (y el gusto) de la programación de sistemas.

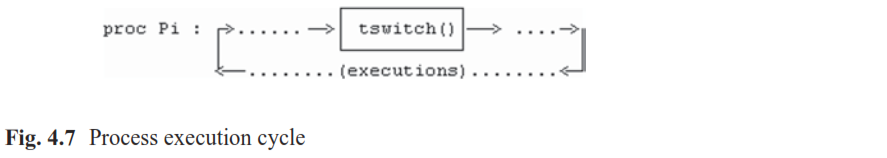
Note también que los procesos, P1 a P8, todos ejecutan el mismo código, digamos la misma función body(). Esto muestra la diferencia entre programas y procesos. Un programa es solo una pieza de código pasivo, la cual no tiene vida en sí misma. Los procesos son ejecuciones de programas, las cuales hacen que los programas estén vivos. Aun cuando están ejecutando el mismo programa cada proceso se ejecuta en su propio contexto. Por ejemplo, en la función body(), la variable local c está en el stack (arreglo stack) de cada estructura proc y el pid es el pid de la estructura proc, etc. En un ambiente multitarea, ya no es apropiado describir lo que un programa hace. Debemos mirar el comportamiento del programa desde el punto de vista de lo que un proceso está haciendo mientras se está ejecutando el programa. Similarmente, un kernel de sistema operativo es solo un programa (grande), el cual es ejecutado por un conjunto de procesos concurrentes. Por conveniencia, frecuentemente decimos lo que un kernel de sistema operativo está haciendo. Pero realmente debemos mirar a un kernel desde el punto de vista de los que los procesos están haciendo mientras están ejecutando el código de kernel. La principal ventaja de este punto de vista basado en procesos es que nos permite desarrollar un kernel de sistema operativo como un conjunto de procesos cooperativos desde cero.

La función de ensamblador tswitch() es la clave para entender los cambios de contexto y la multitarea. Una mejor forma de mirar a tswitch() no es como una pieza de código sino como una estación de remplazo donde los procesos cambian manos como se muestra en la figura 4.6.



En esta vista, tswitch() es una caja de switcheo a la cual entra un procesos Pi y, en general, sale otro procesos Pj. En tswitch(), el código de SAVE y de RESUME son complementarios en que cualesquiera elementos que deben ser guardados por SAVE, RESUME debe restaurar exactamente los mismos elementos. En principio, un proceso podría guardar su contexto de ejecución en cualquier parte, siempre y cuando éste pueda ser recuperado y restaurado cuando el proceso vuelva a ejecutarse otra vez. Algunos sistemas guardan el contexto de proceso en la estructura PROC. Dado que cada proceso ya tiene una pila, elegimos guardar el contexto del proceso en la pila del proceso y solamente guardamos el apuntador de pila en la estructura PROC.

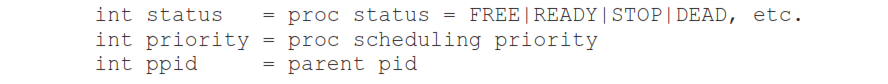
Otra forma de mirar la creación de procesos es como sigue. Cada proceso podría ser pensado como que está ejecutando un ciclo perpetuo, en el cual corre por un tiempo, entonces llama a tswitch() para ceder la CPU a otro proceso. Algún tiempo después, el proceso vuelve a tener acceso a la CPU y emerge de tswitch() continuar ejecutándose otra vez, etc. como se muestra en la figura 4.7.



Crear un proceso que corra por primera vez es lo mismo que hacer que un proceso vuelva a ejecutarse otra vez. Podemos hacer un corte en el ciclo de ejecución de un proceso e inyectar una condición adecuada para que éste vuelva a ejecutarse. El lugar natural para hacer tal corte es cuando el proceso no está corriendo, i.e. inmediatamente después de que ha ejecutado la parte SAVE en tswitch() y cedido la CPU a otro proceso. E el punto de corte, podemos fabricar un marco de pila como si este hubiese sido guardado por el proceso mismo. Cuando el proceso vuelve a correr otra vez, éste obedecerá el marco de pila y regresará a donde sea que queramos que él esté.

**4.4 Dynamic Process Creation and Process Scheduling**

A continuación, modificaremos el programa multitarea como la versión inicial del kernel MTX. Mientras continuamos el desarrollo del kernel MTX, demostraremos cada paso con un sistema muestra completo. Para facilidad de identificación, etiquetaremos los sistemas muestra como MTXx.y, donde x es el capítulo en el cual el kernel MTX es desarrollado, e y es el número de versión. Para comenzar, extenderemos MTX4.1 para soportar creación de procesos dinámica y planificación de procesos por prioridad (estática). Para hacer esto, necesitamos modificar la estructura PROC agregándole más campos. Por conveniencia, (principalmente para facilidad de referencia en código ensamblador), suponemos que ksp es la segunda entrada y que la pila de kernel del proceso está siempre en el extremo alto de la estructura PROC. **Cualesquiera campos de PROC nuevos deben ser agregados entre las entradas ksp y kstack**. La CPU en el modo real de 16 bits hace push y hace pop sobre la pila en unidades de 2 bytes. Cualquier campo agregado también debe estar en unidades de 2 bytes, así que podemos acceder al área de stack (pila) como un arreglo de enteros en C. Para el kernel MTX4.2, los campos de PROC agregados son



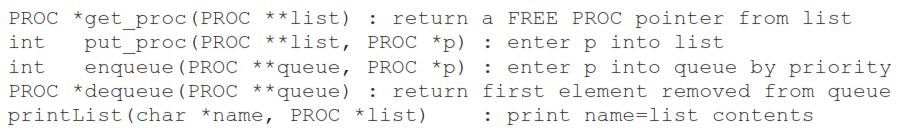
Además, el kernel MTX4.2 también contiene las estructuras de datos freeList y readyQueue, donde

* freeList = una lista simplemente enlazada que contiene todas las PROCs que están libres. Inicialmente, todas las PROCs están en la freeList. Cuando se esté creando un nuevo proceso trataremos de alojarar una FREE PROC de freeList. Cuando un proceso termine, su estructura PROC eventualmente su estuctura PROC será regresada a la freeList para que sea reutilizada.
* readyQueue = una cola de prioridad que contendrá las PROCs que están listas para correr. En la readyQueue, las PROCs de la misma prioridad son ordenadas first-in-first-out (FIFO).

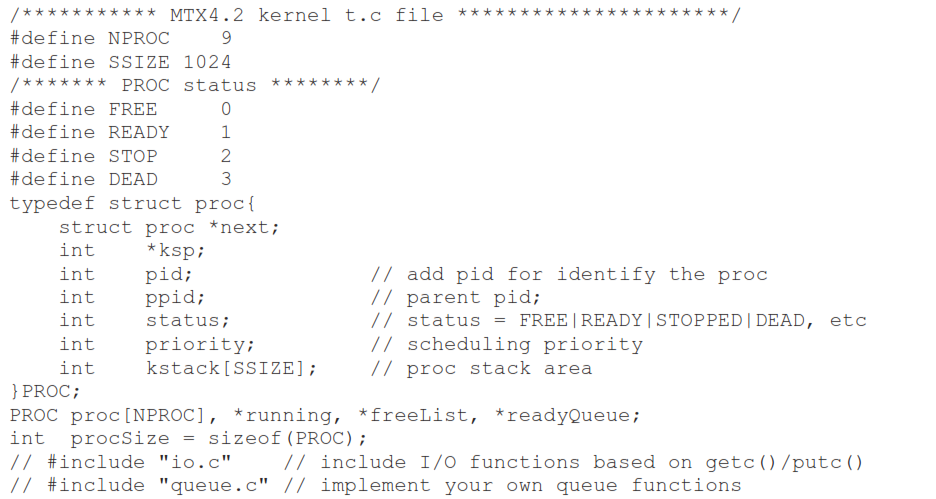
Cuando empieza el kernel MTX4.2 empieza en ts.s, el apuntador de pila es inicializado al extremo alto de proc[0].kstack. Entonces llama a main(), la cual llama a init() para inicializar las estructuras de datos como sigue.

* Inicializar cada PROC asignando pid = índice del PROC, status = FREE y priority = 0.
* Introducir todas las PROCs en freeList.
* Inicializar readyQueue = 0.
* Crear P0 como el primer proceso que corre, i.e. alojar proc[0] y hacer que la variable running apunte a ella.

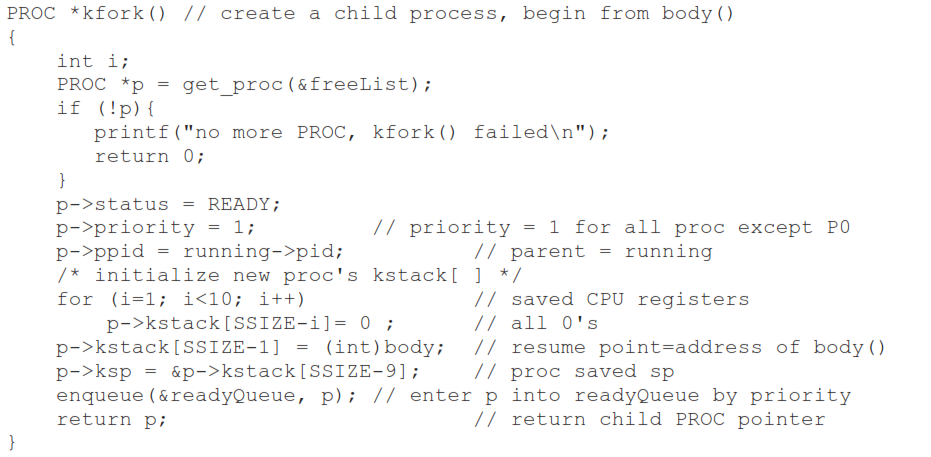
Cuando init() termina, el sistema está corriendo P0. P0 llama a kfork() para crear un proceso hijo P1 y lo introduce en readyQueue. Cuando se crea un proceso el proceso que llama es el padre y el proceso creado es el hijo. Pero como veremos en breve, la relación de procesos padre-hijo no es permanente. El padre de un proceso podría cambiar si su padre original termina primero. Cada nuevo proceso que se crea tiene priority = 1 y comienza su ejecución desde la misma función body(). Después de crear P1, P0 entra en un ciclo while(1), en el cual llama a tswitch() siempre que readyQueue no está vacía. Dado que P1 ya está en readyQueue, P0 switchea para correr P1. En este punto, la planificación de procesos es muy simple. Entre los procesos, P0 tiene la prioridad más baja 0. Todos los otros procesos tienen prioridad 1. Esto implica que P0 es siempre la última PROC en readyQueue. Dado que las PROCs de la misma prioridad son ordenadas de acuerdo con FIFO, las otras PROCs tomarán turnos para correr. P0 correrá otra vez sí y solo si todas las otras PROCs no son ejecutables. La planificación de procesos con prioridades ajustables se discutirá más tarde en la sección 4.4. En la función body(), el proceso que está corriendo imprime su pid y pide un carácter de entrada, donde ‘s’ es para switchear de proceso y ‘f’ es para crear un proceso hijo. En el kernel MTX, las funciones para manipulación de la cola y la lista son:

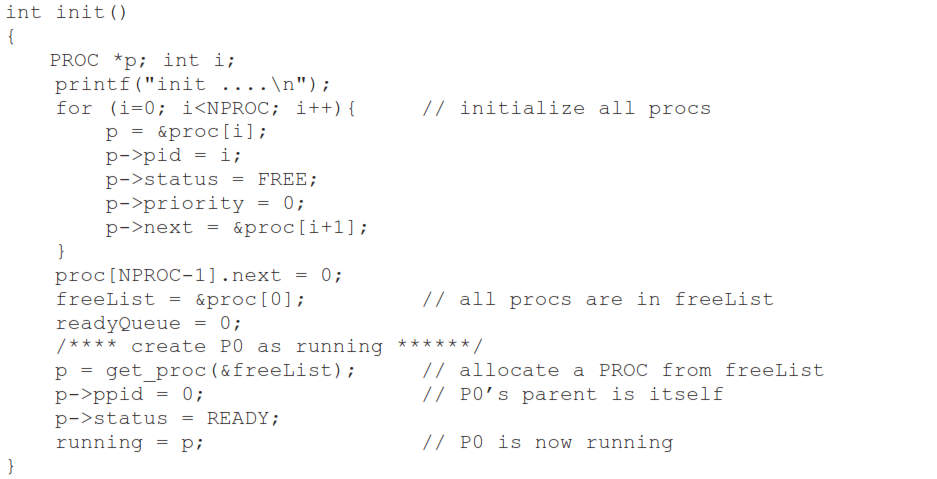


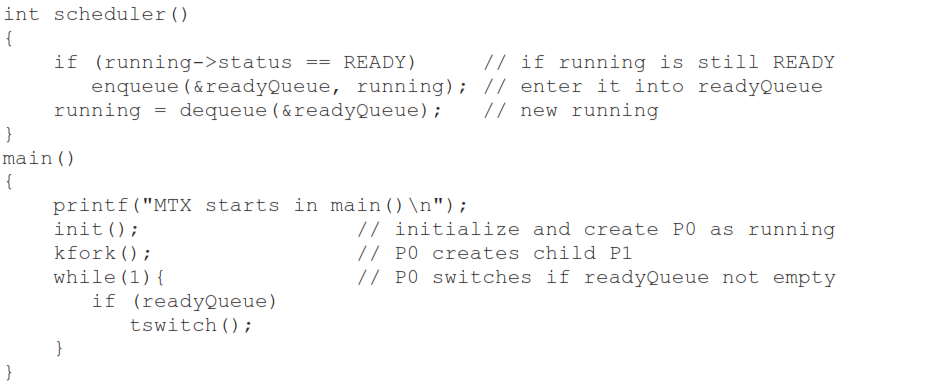
Dado que estas funciones serán usadas en todos los kernels MTX, las hemos precompilado en archivos .o y las hemos puesto en una biblioteca de enlazado, mtxlib, para enlazarlas. Como un ejercicio de programación, se le exhorta al lector a implementar esas funciones en un archivo queue.c e incluirlo en el código C. Durante el enlazado, el enlazador usará las funciones definidas por el lector en lugar de las de la bilioteca mtxlib. Esto se deja como un ejercicio. Lo siguiente es el listado de código C del kernel MTX4.2.

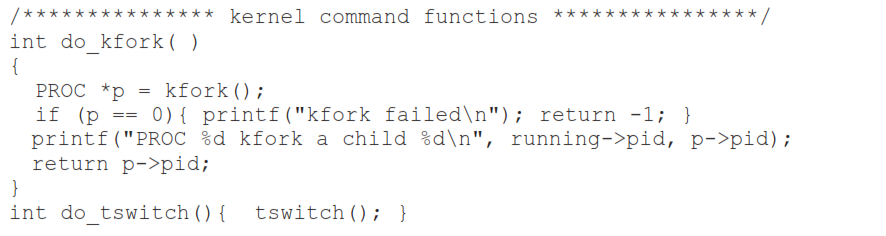






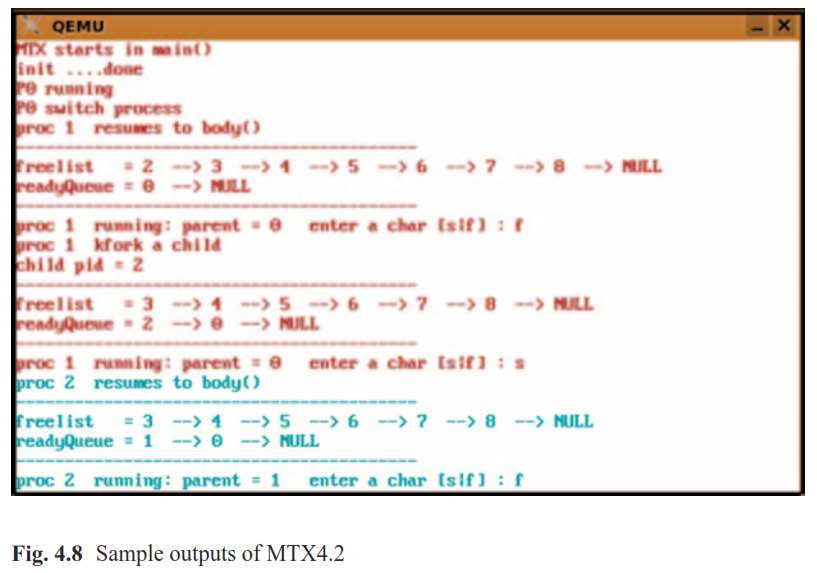






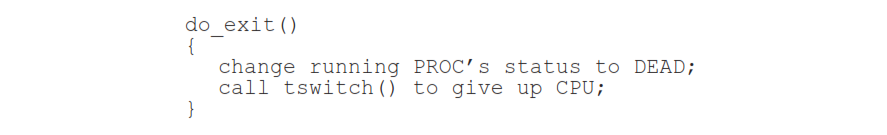
4.5 MTX4.2: Demostración de Creación de Procesos y Switcheo de Procesos

MTX4.2 demuestra creación dinámica de procesos y planificación de procesos por prioridad estática. La figura 4.8 muestra la salida de ejecución del kernel MTX4.2.

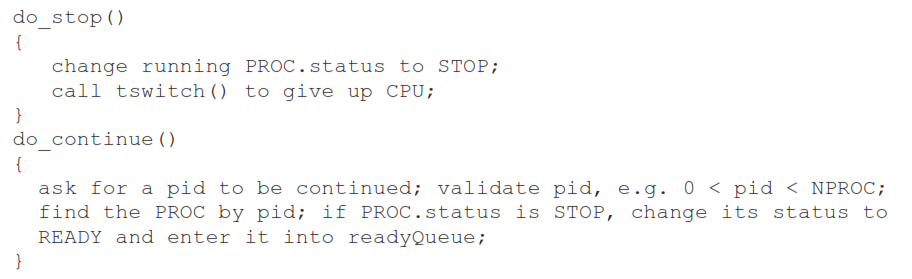


4.6 Detener y continuar un proceso

En la función body(), cada comando invoca a una función do\_command(), la cual llama a una función de kernel correspondiente. El lector podría probar las funciones de kernel y observar su ejecuciones en acción. Como un ejercicio, el lector podría modificar el kernel MTX4.2 para hacer lo siguiente. En la función body(), agregar un comando ‘q’, el cual permita al proceso que está corriendo llamar a do\_exit() para convertirse en DEAD. El algoritmo de do\_exit() es

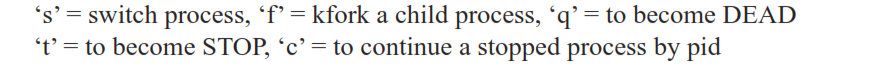


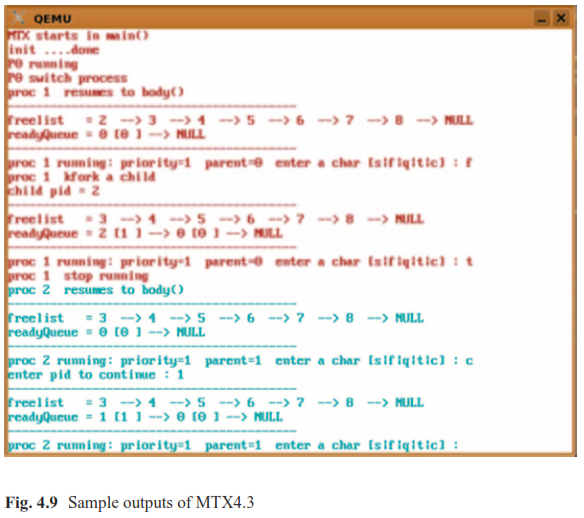
Similarmente, agregar un comando ‘t’, el cual detenga el actual proceso en ejecución, y un comando ‘c’, el cual permita continuar a un proceso detenido. Los algoritmos de stop y continue son



4.7 MTX4.3 Demostración de las operaciones Stop/Continue

La figura 4.9 presenta salidas muestra de MTX4.3, las cuales demuestran las operaciones stop/continue. En la figura, el proceso corriendo despliega su pid, pide un carácter comando y ejecuta el comando, donde

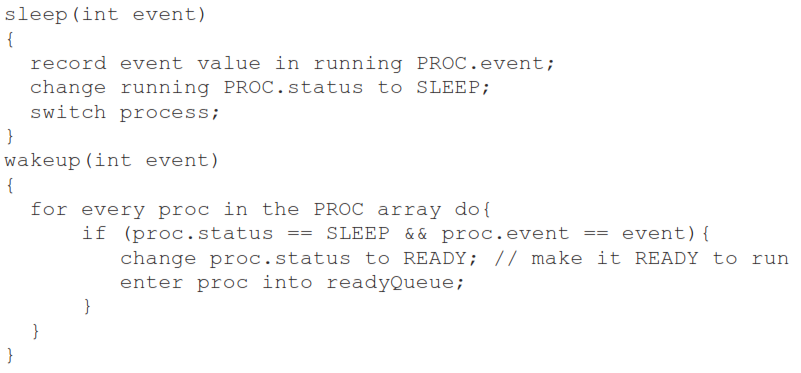




Cuando un proceso llega a los estados STOP o DEAD, cede la CPU llamando a tswitch(). Dado que el status del proceso no es READY, no será introducido en readyQueue, lo cual hace que el proceso no sea ejecutable. Un proceso detenido se convierte en ejecutable otra vez cuando otro proceso le permite continuar. Como un ejercicio, el lector podría modificar stop a una operación stop [pid], la cual detenga un proceso identificado por pid. Si no se especifica pid, detiene al mismo proceso que se está ejecutando. Similarmente, el lector podría modificar la operación continue para permitir a un proceso DEAD se convierta en ejecutable otra vez. Esto debe convencer al lector de que si conocemos el principio de las operaciones de procesos en un kernel de sistema operativo, podríamos hacer cualquier cosa con ellos, inclusive realizar milagros como la resurrección del muerto.

4.8 Operaciones Sleep y Wakeup

La operación stop simplemente hace que un proceso deje de ejecutarse, y la operación continue hace que un proceso detenido esté listo para correr otra vez. Podemos extender stop a una operación sleep(event), la cual detiene un proceso que está corriendo para que espere un evento, y extender la operación continue a una operación wakeup(event), la cual despierte procesos que estén durmiendo cuando el evento por el que esperan ocurra. sleep() y wakeup() son el mecanismo básico de sincronización de procesos en el kernel Unix. Suponga que a cada PROC se le ha agregado un campo event. Los algoritmos de sleep() y wakeup() son



4.9 Uso de Sleep y Wakeup

En un kernel de sistema operativo, sleep y wakeup se usan como sigue.

1 Cuando un proceso debe esperar por algo, por ejemplo, un recurso, identificado por un valor de evento, el proceso llama a sleep(event) para ir a dormir, esperando a que el evento ocurra. En sleep(), registra el valor del evento en la estructura PROC, cambia su status a SLEEP y cede la CPU. Un proceso que está durmiendo no es ejecutable dado que no está en la cola readyQueue. Se volverá ejecutable otra vez cuando el evento esperado ocurra, en el momento en que otro proceso (o un manejador de interrupción) llama a wakeup(event) para despertarlo.

2 Un evento es cualquier valor sobre el que un proceso podría dormir, siempre algún otro proceso hará una llamada a wakeup sobre el valor del evento. Depende del diseñador del sistema asociar cada recurso con un valor de evento único. En Unix, los valores de evento usualmente son direcciones de variable (global) en el kernel Unix, las cuales son únicas, de tal manera que los procesos puedan dormir sobre valores de eventos distintos. wakeup(event) solo despierta a aquellos procesos que están durmiendo sobre el valor de evento especificado. Por ejemplo, cuando un proceso espera por la terminación de un proceso hijo, usualmente duerme sobre su propia dirección PROC, la cual es única y también es conocida para sus hijos. Cuando un proceso termina, hace la llamada wakeup(&parentPROC) para despertar a su padre.

3 Muchos procesos pueden dormir sobre el mismo evento, lo cual es natural porque muchos procesos podrían necesitar el mismo recurso que actualmente está ocupado o no disponible.

4 Cuando un evento ocurre, alguien (un proceso o un manejador de interrupción) llamará a wakeup(event), lo cual despierta a TODOS los procesos que están durmiendo sobre ese evento. Si ningún proceso está durmiendo sobre ese evento wakeup() no tiene efecto, i.e. no hace nada. Cuando un proceso despertado corre, debe tratar de obtener el recurso otra vez dado que el recurso podría ya haber sido obtenido por otro proceso.

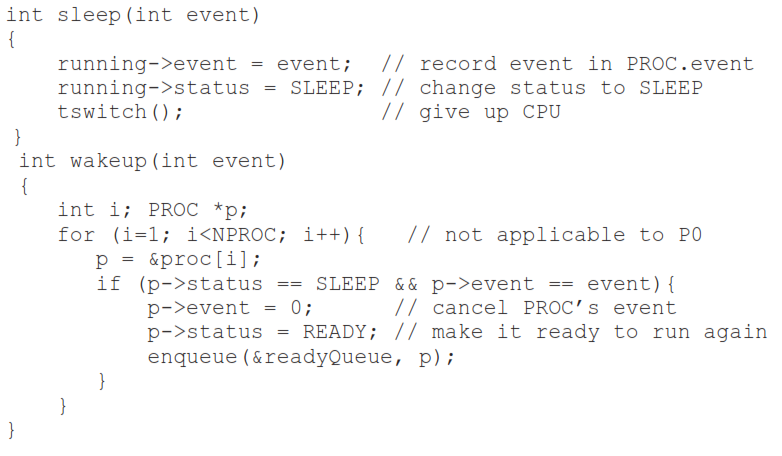
5 Dado que un evento es solo un valor, no tiene una ubicación de memoria para registrar la ocurrencia del evento. Para que no se pierda una llamada a wakeup, un proceso debe ir a dormir ANTES de que el evento esperado ocurra. En un sistema uniprocesador, esto siempre se puede conseguir. En un sistema multiprocesador, el orden sleep\_first\_wakeup\_after no puede ser garantizado porque los procesos podrían correr en paralelo en tiempo-real. Así que sleep/wakeup funciona solamente para sistemas uniprocesador. **Para sistemas multiprocesador, necesitamos otros tipos de herramientas de sincronización de procesos, por ejemplo, semáforos, los cuales serán discutidos más tarde en el capítulo 6**.

6 El kernel Unix asigna una prioridad fija a un proceso cuando este va a dormir. La prioridad asignada en la importancia del recurso por el que el proceso está esperando. Esta es la prioridad de planificación del proceso cuando este despierta. Si un proceso despertado tiene una prioridad más alta que el proceso que está corriendo actualmente, el cambio de proceso no sucede inmediatamente. Este es pospuesto hasta que el proceso que está corriendo actualmente va a salir del modo kernel para regresar al modo usuario. Las razones para esto también serán discutidas más tarde.

7 La prioridad asignada clasifica a un proceso que está durmiendo ya sea como un SOUND sleeper o como un LIGHT sleeper. Un sound sleeper solamente puede ser despertado por su evento esperado. Un light sleeper puede ser despertado por otros medios, los cuales podrían no ser el evento por el que está esperando. Por ejemplo, cuando un proceso espera por I/O de disco, duerme con una prioridad altay no debe ser despertado por señales. Si espera por una tecla de entrada de una terminal, lo cual podría no llegar por un largo tiempo, duerme con una prioridad baja y puede ser despertado por señales. En el kernel Linux, un proceso que duerme (task) es ya sea INTERRUPTABLE o UNINTERRUPTABLE, lo cual es lo mismo que un light sleeper o un sound sleeper en Unix.

4.10 Implementación de Sleep y Wakeup

Para el kernel MTX, podríamos implementar sleep() y wakeup() como sigue.



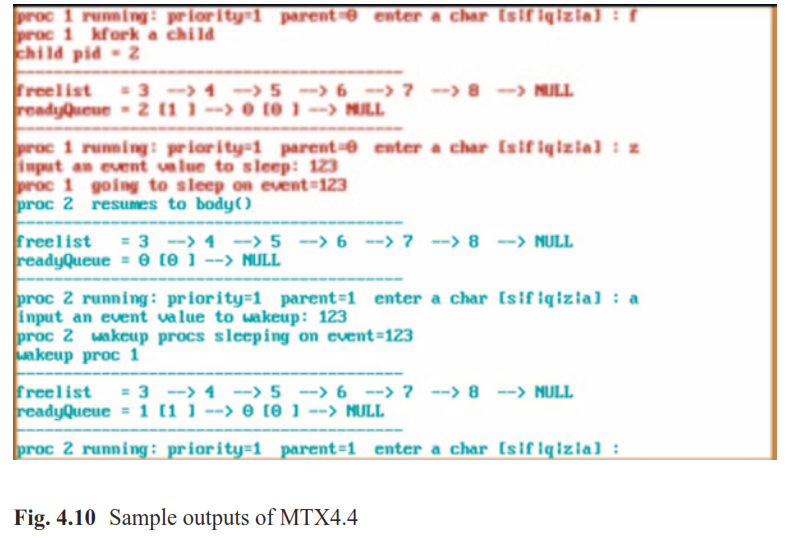
Estrictamente hablando, la implementación de arriba tiene serios inconvenientes pero funciona para el kernel MTX actual porque

* Por ahora, MTX es un sistema uniprocesador, en el cual solamente un proceso puede correr en cualquier instante de tiempo. Un proceso corre hasta que está listo para switchear, por ejemplo, cuando ve un comando ‘s’ o ‘t’. Un cambio de procesos ocurre solamente después de que un proceso ha terminado una operación, nunca en medio de una operación. En otras palabras, cada proceso corre solo sin interferencia laguna de otro proceso.
* Hasta ahora, el kernel MTX no tiene interrupciones. Cuando un proceso corre, no puede ser interferido desde algún manejador de interrupción.

**Los detalles de cómo implementar sleep/wakeup y otros mecanismos de sincronización de procesos apropiadamente serán cubiertos más tarde en el capítulo 6**.

4.11 MTX4.4 Demostración de las operaciones Sleep/Wakeup

Para demostrar las operaciones sleep y wakeup, agregamos los comandos ‘z’ y ‘a’ a la función body(), donde ‘z’ es para que un proceso valla a dormir sobre un valor de evento y ‘a’ despierta procesos, si hay alguno, que esté durmiendo sobre un valor de evento. La figura 4.10 presenta salidas muestra de la ejecución de MTX4.4.



5.1 Imagen de Ejecución de Proceso

La imagen de ejecución de un proceso consiste de tres segmentos lógicos; Código, Datos y Pila, como se muestra en la figura 5.1.



En teoría, todos los segmentos pueden ser independientes, cada uno en un área de memoria diferente, siempre y cuando los segmentos sean apuntados por los registros de segmentos de la CPU. En la práctica, algunos de los segmentos podrían coincidir. Por ejemplo, en el modelo de memoria de un solo segmento, todos los segmentos son el mismo. Durante la ejecución, los registros CS, DS y SS de la CPU los tres apuntan al mismo segmento de la imagen de ejecución. En el modelo de memoria espacio I&D separado, CS apunta al segmento de código, pero DS y SS apuntan al segmento de datos y de pila combinados. Por simplicidad, asumiremos primero el modelo de memoria de un solo segmento. Cada imagen de proceso tiene solamente un segmento y el tamaño del segmento es 64 KB. Las imágenes de proceso con espacios I&D separados y tamaños variable serán considerados más tarde en el capítulo 7 cuando discutimos administración de memoria.

5.2 Modos de kernel y de usuario

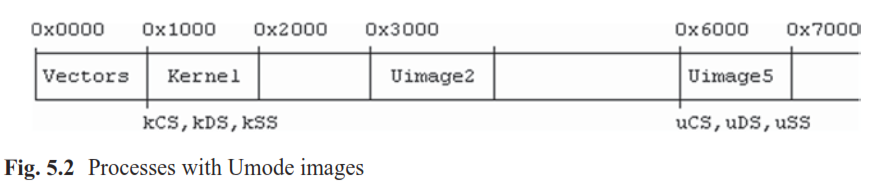
De ahora en adelante supondremos que un proceso podría ejecutarse en dos modos diferentes; modo kernel y modo usuario, denotados por Kmode y Umode por brevedad. Mientras están en Kmode, todos los procesos comparten el mismo código y datos del kernel del sistema operativo, pero cada proceso tiene su propia pila de modo kernel (en la estructura PROC), la cual contiene el contexto de ejecución del proceso mientras está en Kmode. Cuando un proceso cede la CPU, guarda su contexto dinámico (registros de la CPU) en kstack. Cuando un proceso se convierte en ejecutable otra vez, restaura el contexto guardado desde kstack. En Umode, las imágenes de proceso son en general todas diferentes. Cada proceso tiene un área de memoria de Umode diferente que contiene el código, datos y pila de modo de usuario, denotados por Ucode, Udata y Ustack, del proceso. Por facilidad de la discusión, comenzamos con las siguientes suposiciones.

1 El kernel MTX corre en el segmento 0x1000.

2 El sistema tiene 9 estructuras PROC, P0 a P8. P0 siempre corre en Kmode.

3 Solamente P1 a P8 podrían correr en Umode, cada uno tiene una imagen Umode de 64 KB distinta en el segmento (pid+1)\*0x1000, por ejemplo, P1 en 0x2000, P2 en 0x3000, P8 en 0x9000, etc.

La asignación de memoria de segmento fija de imágenes de proceso es solo por limpieza de discusión. Esto será removido más tarde cuando implementemos administración de memoria.



A continuación, suponga que el proceso P5 está corriendo en Umode ahora, P2 está en READY pero no está corriendo. La figura 5.2 muestra el mapa de memoria actual del sistema MTX.

Dado que P5 está corriendo en Umode, los registros CS, DS, SS de la CPU todos deben apuntar a la imagen Umode de P5 en 0x6000. El Ustack de P5 está en la región alta de Uimage5. Cuando P5 entra en Kmode, ejecutará la imagen de kernel en el segmento 0x1000. Los registros CS, DS, SS de la CPU deben ser cambiados para que apunten a 0x1000, como se representa con kCS, kDS, kSS en la figura 5.2. Para que P5 corra en el modo kernel, el apuntador de pila también debe ser cambiado para que apunte al kstack de P5. Naturalmente, P5 debe guardar sus registros de segmento de Umode, uCS, uDS, uSS, y usp si intenta regresar a Uimage5 más tarde.

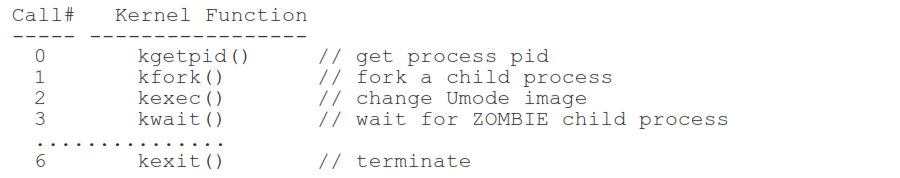
Suponga que, mientras está en Kmode, P5 switchea a P2, el cual podría regresar a su imagen de Umode en 0x3000. Si es así, P2 debe cambiar los registros de segmento de la CPU a sus propios uCS, uDS, uSS, los cuales todos deben apuntar a 0x3000. Cuando P2 corre en Umode, su ustack está en la región alta de Uimage2. Similarmente, el lector podría deducir que sucedería si P2 entra en Kmode para switchear a otro proceso, etc.

5.3 Transición entre los modos de Usuario y de Kernel

En un sistema operativo, un proceso migra entre Umode y Kmode muchas veces durante su tiempo de vida. Aunque cada proceso comienza en Kmode, supondremos que un proceso ya se está ejecutando en Umode. Esto suena como otro problema de la gallina y el huevo, pero podemos manejarlo fácilmente. Un proceso en Umode entrará en Kmode si uno de los siguientes eventos ocurre:

* Excepciones : también llamadas trampas (traps), tales como instrucción ilegal, dirección inválida, etc.
* Interrupciones : interrupciones de timer, completación de I/O de dispositivo, etc.
* Llamadas al sistema : INT n (o insgtrucciones equivalentes en otras CPUs).

Las excepciones y las interrupciones serán cubiertas en capítulos posteriores. Aquí, solamente consideramos llamadas al sistema. Llamada al sistema o syscall por brevedad, es un mecanismo el cual permite a un proceso en Umode entrar al Kmode para ejecutar funciones de kernel. Las syscalls no son llamadas a funciones ordinarias porque involucran que la CPU opere en modos diferentes (si la CPU puede hacerlo) y ejecute instrucciones en espacios de direcciones diferentes. Sin embargo, una vez que el enlazado es establecido, las syscalls pueden ser usadas como si fueran llamadas a funciones ordinarias. Suponga que hay N funciones de kernel, cada una corresponde a un número de llamada n=0, 1, . . . , N-1, por ejemplo,



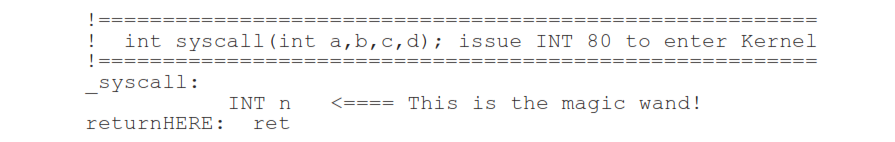
donde el prefijo k de lso nombres de función enfatizan que son funciones en el kernel del SO. Un proceso en Umode podría usar



para entrar al Kmode a ejecutar la función de kernel correspondiente, pasando los parámetros a la función de kernel según como se necesiten. Cuando la función de kernel termina, el proceso regresa a Umode con los resultados deseados y u valor de retorno. Para la mayoría de las syscalls, un valor de retorno 0 significa éxito y -1 significa una falla.

5.3.1 Llamadas al sistema

Suponiendo 4 parámetros, la implementación de syscall() se muestra abajo.



En la CPU Intel x86, syscall es implementada con la instrucción INT n, donde n es un valor byte (0--255). Aunque podríamos usar diferentes INT n para implementar diferentes syscalls, es suficiente con usar solamente un número: INT 80, dado que el parámetro a representa el número de syscall. La elección de INT 80 es completamente arbitraria. Podríamos elegir algún otro número, por ejemplo 0x80, siempre que no sea usado como IRQ del hardware de interrupción o por el BIOS. Cuando la CPU ejecuta INT 80, hace lo siguiente.

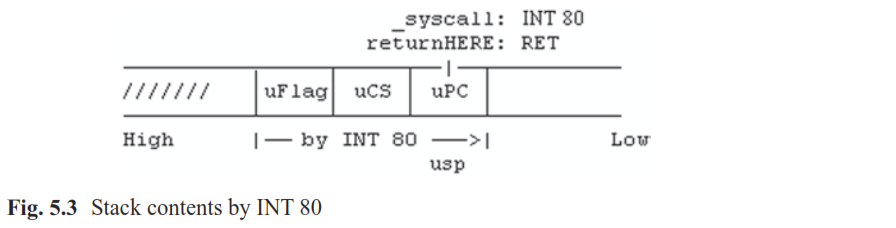
PUSH : guarda en pila el registro de banderas, limpia los T-bit e I-bit del registro de banderas; guarda en pila uCS y uPC.

LOAD : cargar (PC,CS) con los contenidos de (4\*80,4\*80+2)=(\_int80h,KCS).

HANDLER : continua la ejecución de los valores cargados (PC,CS)=(\_int80h,KCS).

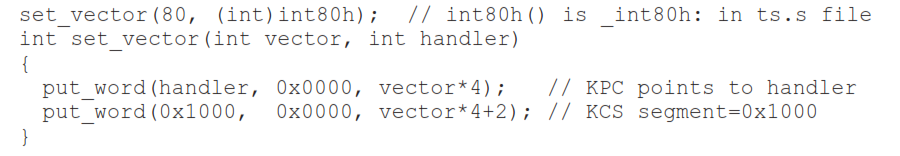
Dado que esas operaciones son cruciales para entender el procesamiento de llamadas al sistema y de interrupciones en capítulos posteriores, las expliacremos en más detalle.

PUSH: INT n causa que la CPU guarde en pila los actuales uFlag, uCS, uPC. En la mayoría de otras máquinas, una instrucción especial como INT n causa que la CPU entre a un Kmode separado y switchee la pila a la pila de Kmode automáticamente. La CPU Intel x86 en el modo real de 16 bits no tiene un Kmode separado o un apuntador de pila de Kmode separado. Después de ejecutar INT 80, solamente switchea el punto de ejecución de (uPC,uCS) a (kPC,kCS), lo cual cambia el segmento de código de UCode a KCode. Todos los otros segmentos (DS,SS,ES) y registros de CPU son aún aquellos en el Umode. Entonces, cuando la CPU entra en la función manejadora int80h(), la pila aún es la ustack del proceso en el segmento uSS. Esta pila contiene a uFlag, uCS, uPC en la cima, donde uPC apunta a la dirección de returnHERE en \_syscall, como se muestra en la figura 5.3.



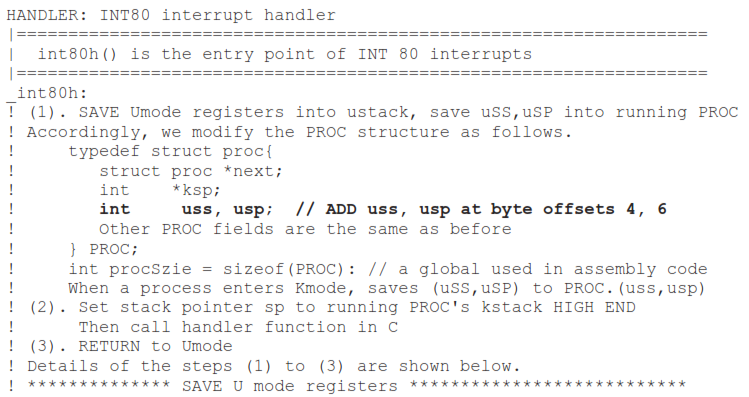
Correspondiente a INT n, la instrucción IRET extrae de la pila actual tres elementos y los coloca en los registros PC, CS y de banderas de la CPU, en ese orden. Esta instrucción es usada por manejadores de interrupción para regresar al punto de interrupción original.

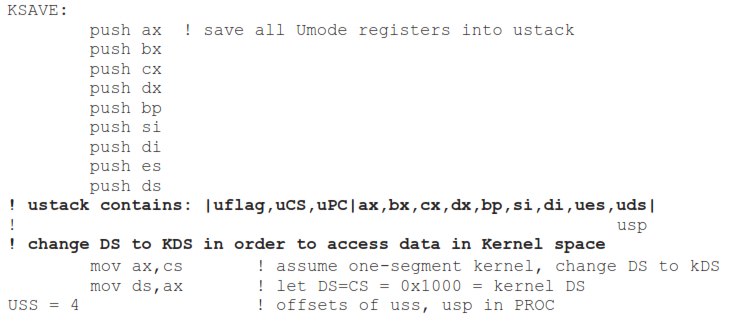
LOAD: Para la CPU x86 en modo real de 16 bits, el área de memoria física de 1 KB más baja está dedicada a 256 vectores de interrupción. Cada área vector de interrupción contiene un par de (PC,CS), el cual apunta al punto de entrada de un manejador de interrupción. Después de guardar uFlag, uCS y uPC del punto interrumpido, la CPU apaga los bits T (trace) e I (Interrupt Mask) en el registro de banderas para deshabilitar trace trap y enmascarar las interrupciones. Entonces carga (PC,CS) con los contenidos del vector de interrupción 80 como el nuevo punto de ejecución. El área del vector de interrupción 80 debe ser inicializada antes de ejecutar INT 80, como se muestra abajo.

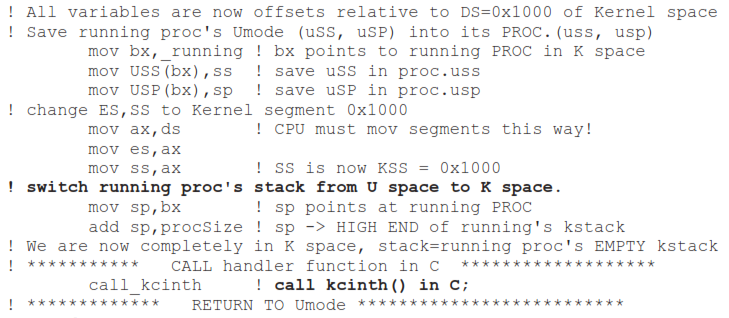


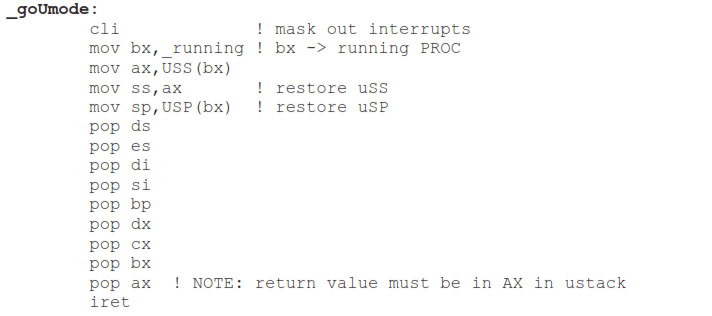
5.3.2 Manejador de Interrupción de Llamada al Sistema

Después de cargar el contenido del vector 80 en los registros (PC,CS), la CPU ejecuta int80h() en el segmento de código del kernel MTX. int80h() es el punto de entrada del manejador de interrupción INT 80 en código ensamblador, el cual se muestra abajo.





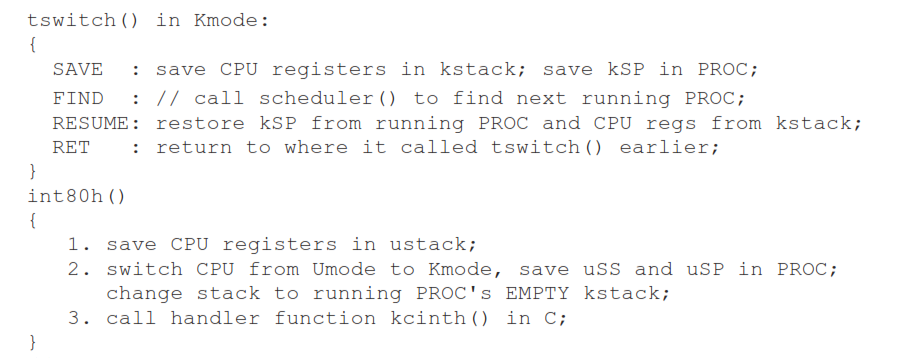


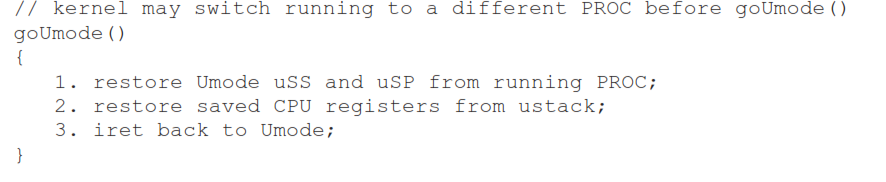


La función manejadora en C, kcinth(), realmente maneja la syscall. Cuando la función manejadora en C termina, el proceso regresa a ejecutar el código de ensamblador \_goUmode. Este primero restaura la pila de Umode del par PROC.(uss,usp) que está en ejecución. Entonces restaura los registros Umode guardados, seguido por IRET, causando que el proceso regrese al punto interrumpido en Umode. Dado que \_goUmode es un símbolo global, un proceso en Kmode podría llamar goUmode() directamente para retornar al Umode.

5.3.3. Entrar y Salir del Modo Kernel

El código ensmablador \_int80h y \_goUmode son claves para entender las transiciones entre Umode y Kmode. Lógicamente, int80h() y goUmode() son similares a las operaciones SAVE y RESUME de tswitch() en Kmode. A continuación, se comparan sus similitudes y diferencias.





La principal diferencia es 2 en int80h(), lo cual cambia el ambiente de ejecución de la CPU de Umode a Kmode y guarda el uSS y uSP del Umode en el PROC que está corriendo. En la mayoría de las CPUs diseñadas para operaciones en Kmode y Umode, el switcheo es automático. Cuando una de esas CPUs ejecuta la instrucción syscall (o acepta una interrupción), automáticamente switchea al Kmode y guarda el contexto de interrupción en la pila del Kmode. Dado que la CPU x86 en el modo real no tiene esta capacidad, tenemos que forzarla a hacer el switcheo manualmente, lo cual es al mismo tiempo una maldición y una bendición. Es una maldición porque tenemos que hacer el trabajo extra. Es una bendición porque le permite al lector entender mejor cómo la CPU switchea el modo y ambiente de ejecución. \_int80h y \_goUmode() son el código de entrada y de salida del manejador de syscall. Dado que la syscall es solo un tipo especial de interrupción, el mismo código de entrada y de salida pueden también ser usados para manejar otros tipos de interrupciones y excepciones. Mostraremos esto más tarde en los capítulos 8 y 9 cuando discutimos procesamiento de interrupciones y excepciones.

5.3.4 Función manejadora de llamada al sistema en C

La función kcinth() es la función manejadora de syscall en C. Los parámetros usados en syscall(a,b,c,d) están en el ustack del proceso, la cual contiene las entradas como se muestran en la figura 5.4.



El segmento de pila del Umode es guardado en PROC.uss, y usp es guardado en PROC.usp. Usando las funciones de copia inter-segmento, get\_word()/put\_word(), podemos acceder al ustack del proceso. Por ejemplo, podemos obtener el número de syscall (en índice 13) con



Similarmente, podemos obtener los parámetros de la syscall desde el ustack. Basándose en el número de llamada a, podemos enrutar a una función de kernel correspondiente, pasándole los parámetros b, c, d a esa función según como se requiera. La función de kernel realmente maneja la syscall. Como un ejemplo, supongamos que la syscall número 0 es para obtener el pid del proceso. La llamada es enrutada a



Lo cual simplemente devuelve el pid del proceso que está corriendo. Le la misma forma, podemos escribir al ustack de un proceso para cambiar sus contenidos. En particular, podemos cambiar el uax guardado (en índice 8) como el valor de retorno a Umode usando put\_word(r,running->uss,running->usp+2\*8);

5.3.5 Regreso al Modo de Usuario

Cada función de kernel devuelve un valor de regreso al Umode, excepto kexit(), la cual nunca regresa, y kexex(), la cual regresa a una imagen diferente si la operación tiene éxito. EL valor de retorno es llevado en el registro AX de la CPU. Dado que goUmode() saca el uax de la pila ustack, debemos corregir el uax guardado con el valor de retorno deseado antes de ejecutar goUmode(). Esto puede ser hecho en la función manejadora en C kcinth() antes de que regrese a ejecutar goUmode().

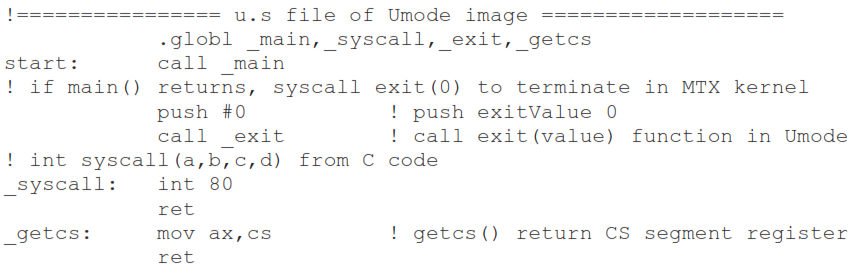
El párrafo anterior describe el flujo de control de las syscalls. Muestra la traza de ejecución de un proceso cuando éste hace una syscall para entrar al kernel, ejecuta funciones de kernel y regresa al Umode con un valor de retorno. Las syscalls son el mecanismo principal en el soporte de operaciones de modo kernel y de modo usuario. La implementación del mecanismo de syscall es uno de los pasos más importantes en el desarrollo de un kernel de sistema operativo. En lo que sigue, mostraremos cómo implementar syscalls en MTX. Esto debe permitirle al lector entender mejor las operaciones internas de un kernel de sistema operativo.

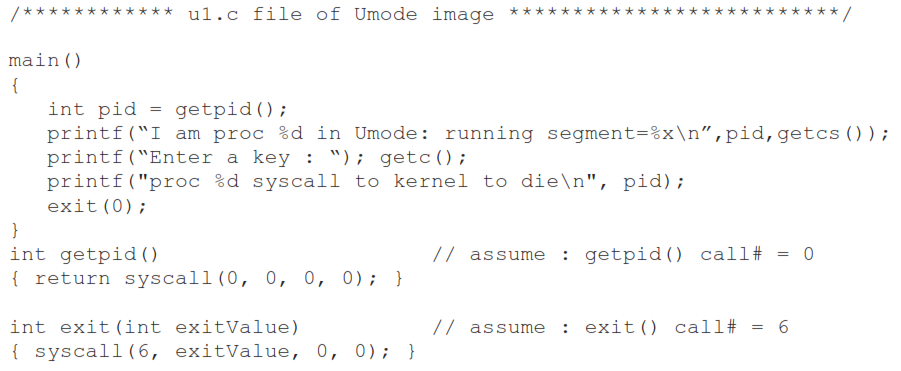
5.4 Programas de Modo Usuario

Para poder lanzar syscalls, necesitamos un archivo imagen de Umode, lo cual es un binario ejecutable para ser ejecutado por un proceso en Umode.

5.4.1 Desarrollar Programas de Modo Usuario

El siguiente listado muestra un programa de Umode simple. Como es usual, consiste de un archivo u.s en lenguaje ensamblador BCC y de un archivo u1.c en C.

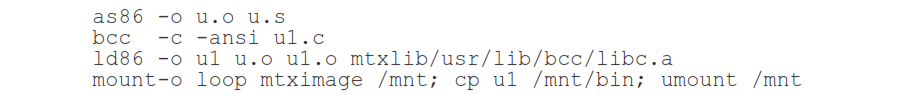




El archivo u.s es necesario porque solamente podemos lanzar syscalls usando INT 80 en ensamblador. Este también sirve como punto de entrada de programas de Umode. Como pronto se verá, cuando un programa de Umode comienza su ejecución, los registros de segmento de la CPU están ya establecidos por el código de kernel y tiene ya un ustack. Así que al entrar, u.s simplemente llama a main() en C.

Un programa del Umode puede solamente hacer computaciones generales. Cualquier operación no disponible en el Umode debe ser hecha a través de syscalls al kernel del SO. Por ejemplo, getpid() y exit() son syscalls. Esto es porque un proceso solamente puede obtener su pid desde el espacio de kernel y debe terminar en el kernel. Cada syscall tiene una función interface, la cual lanza una syscall real al kernel. Por conveniencia, las funciones interface de syscalls son implementadas en un único archivo, ucode.c, el cual es compartido por todos los programas del Umode. Usualmente, las funciones interfaces de syscalls son precompiladas como parte de las bibliotecas de enlazado del sistema. Los programas del Umode podrían llamar funciones interface de syscall, tales como getpid(), exit(), etc. como llamdas a función ordinarias.

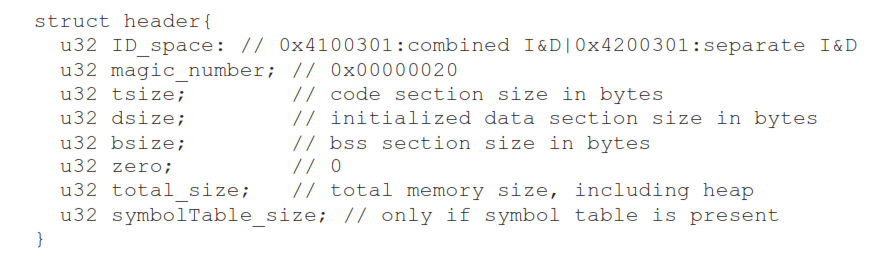
Hasta ahora, el kernel MTX aun no tiene sus propios drivers de dispositivo. Todas las operaciones de I/O en MTX están basadas en BIOS. Las funciones de I/O, tales como getc(), putc(), gets(), printf(), son código objeto precompilado en una biblioteca mtxlib, la cual es usada por ambos el kernel MTX y los programas del Umode. Por lo tanto, getc() y putc() en Umode son también llamadas al BIOS. Estrictamente hablando, los programas de Umode no deben ser capaces de hacer I/O directamente. I/O básica de Umode, por ejemplo, getc() y putc(), también deben ser syscalls. Esto se deja como un ejercicio. Como antes, use BCC para generar un binario ejecutable (con header, lo cual es necesario para el loader), como en



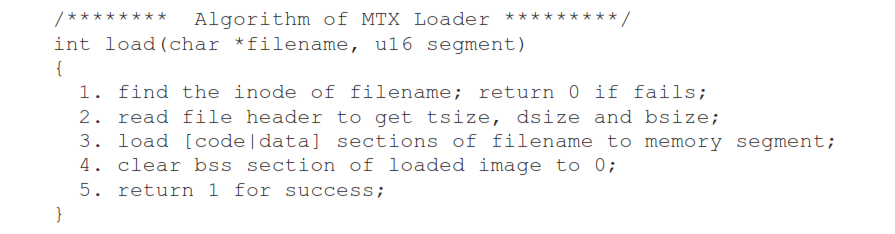
La última línea de comando copia u1 a /bin/u1 en una imagen de sistema MTX bootable.

5.4.2 Cargador de Programa (Program Loader)

Los archivos a.out generados por BCC tienen un header de 32 bits que contiene 8 valores long.

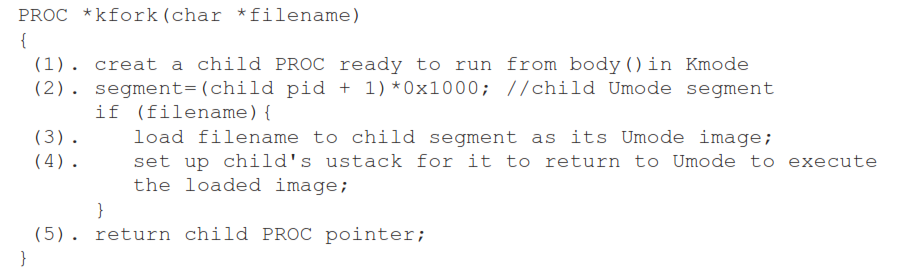


Un loader es un programa el cual carga un archivo binario ejecutable en la memoria para ejecución. En un kernel de SO real con soporte de sistema de archivos, el loader típicamente usa la función open() interna del kernel para abrir el archivo imagen para lectura. Entonces usa la función read() interna del kernel para cargar el archivo imagen en la memoria. Mostraremos esto más tarde cuando agreguemos soporte de sistema de archivos al kernel MTX. Mientras tanto, modificaremos el booter de MTX para convertirlo en un loader. En este caso, el loader es casi idéntico al booter, excepto que no es un programa standalone sino una función que se puede llamar en el kernel. El algoritmo del loader es



5.5 Crear Proceso con imagen de Modo Usuario

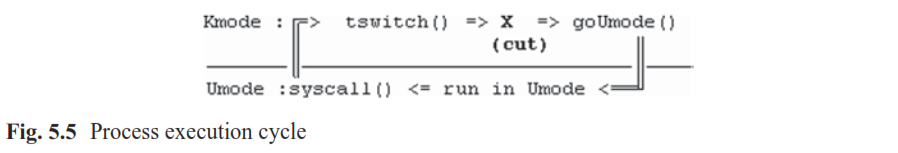
En el kernel MTX, modificamos kfork() a kfork(char \*filename), la cual crea un nuevo proceso y carga filename como su imagen de Umode. Lo siguiente muestra el algoritmo de kfork modificado.



En el kfork(), el paso (1) es el mismo que antes. El kernel MTX tiene 9 PROCs pero solamente P1 a P8 podrían correr en el Umode. El segmento de Umode de cada proceso es asignado estáticamente en el paso (2) como (pid+1)\*0x1000, así que cada proceso corre en un segmento de Umode único. En el paso (3), llama a load(“/bin/u1”,segment), la cual carga /bin/u1 en el segmento de Umode del nuevo proceso y limpia la sección bss en la imagen de Umode a 0. El paso (4) es el más crucial, así que lo explicaremos en detalle.

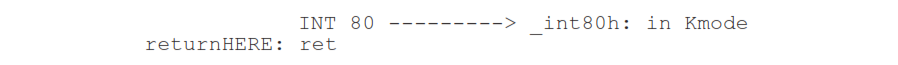
5.6 Inicializar la Pila de Modo de Usuario del Proceso

Nuestro objetivo aquí es ajustar el ustack de un nuevo proceso para que este regrese a Umode para ejecutar la imagen cargada. Para hacer esto, otra vez pensamos de cada proceso como que está corriendo en un ciclo perpetuo, como se muestra en la figura 5.5.



En la figura 5.5, un proceso en Umode hace una syscall para entrar a Kmode, se ejecuta en Kmode hasta que llama a tswitch() par cedr la CPU y está en la readyQueue hasta que corre otra vez. Entonces ejecuta goUmode para regresar al Umode y repite el ciclo. Podemos hacer un corte en el ciclo de ejecución del proceso e inyectar una condición para que regrese al Umode. El punto de corte es etiquetado con la letra X, justo antes de que el proceso ejecute goUmode(). Para crear una condición adecuada para que un proceso valla a goUmode(), podríamos hacer la pregunta: ¿Cómo un proceso llegó al punto de corte? La secuencia de eventos debe ser como sigue:

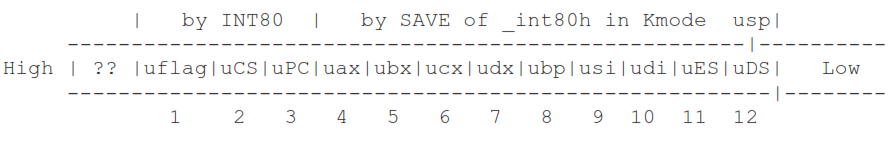
4 El proceso hizo un INT 80 en Umode a través de



Cuando el proceso entra primero a Kmode, su ustack contiene

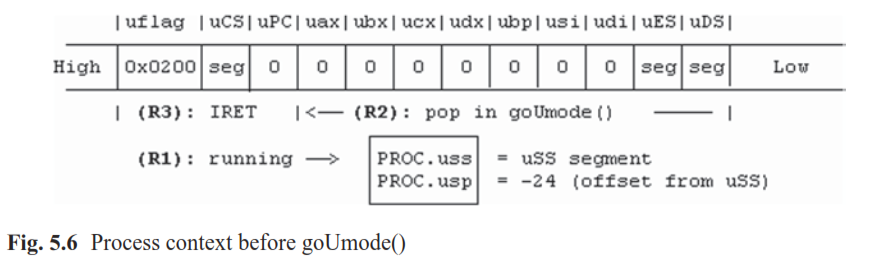


donde el uPC guardado apunta a returnHERE: en la sección de código de Umode. Entonces ejecuta \_int80h: para guardar los registros de la CPU a su ustack, lo cual se convierte en



Entonces guarda el segmento Umode en su PROC.uss y el apuntador de pila en PROC.usp. Nuestra tarea aquí es crear un ustack para el proceso recientemente creado para goUmode(). Llenaremos los contenidos de ustack como si esto hubiese sido hecho por el proceso mismo cuando entró al Kmode. Después de cargar la imagen de Umode a un segmento, el ustack está en el extremo alto del segmento. Aunque el proceso nunca existió antes, pretendemos que si.

El proceso ejecutó INT 80 desde la dirección virtual 0 en Umode y ahí es a donde regresará cuando ejecute goUmode(). Además, antes de ejecutar INT 80 su ustack estaba vacía, los registros de segmento de la CPU CS, DS, ES, SS todos apuntaban a su segmento Umode y todos los otros registros eran 0´s, excepto uflag, el cual debe permitir interrupciones mientras está en Umode. Por lo tanto, el contexto de proceso guardado debe ser como el mostrado en la figura 5.6.



En la figura 5.6, uflag=0x0200 (I-bit=1para permitir interrupciones), uCS=uDs=uEs=Umode segment, uPC=0 (para la dirección virtual 0) y todos los otros registros “guardados” son 0. En la PROC del proceso, establecemos PROC.(uss,usp)=(Umode segment,-24). El lector podría asombrarse de ¿por qué -24? En la ustack, el uDS guardado es la 12/o entrada desde la izquierda. Su offset es -2\*12=-24 desde el extremo alto de ustack. En binario en 16 bits, -24 es 1111111111101000 (en forma de complemento a 2) o 0xFFE8 en hexadecimal, lo cual es exactamente la dirección offset del uDS guardado en ustack. En otras palabras, una dirección virtual en un segmento puede ser expresada como un offset positivo desde el extremo bajo del segmento o como un offset negativo desde el extremo alto (64 KB) del segmento.

5.7 Ejecución de la Imagen de Modo Usuario

Con los contenidos de ustack como se muestran en la figura 5.6, cuando la PROC se convierte en running y ejecuta \_goUmode, haría los pasos (R1) a (R3) mostrados en la figura 5.6.

(R1): Restaurar los registros (SS,SP) de la CPU desde (PROC.SS,PROC.SP). La pila es ahora el ustack de proceso como se muestra en la figura.

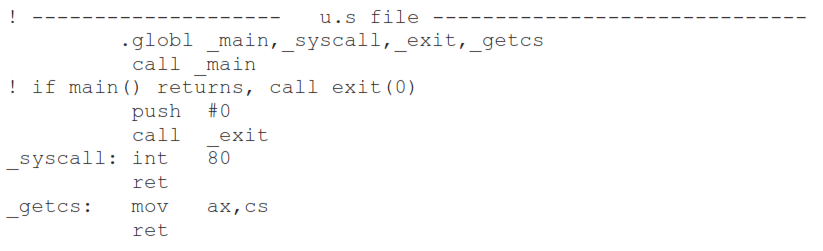
(R2): Pop “saved” registers into CPU, lo cual establece DS y ES al segmento Umode.

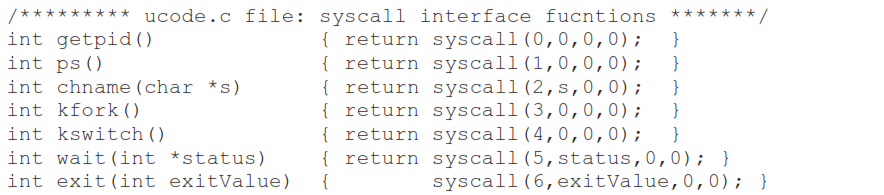
(R3): Ejecutar IRET, lo cual extrae con pop de la pila los tres elementos restantes de ustack en los registros de baderas, CS, PC de la CPU, causando que la CPU ejecute desde (CS,PC)=(seg,0), lo cual es el principio del programa imagen Umode.

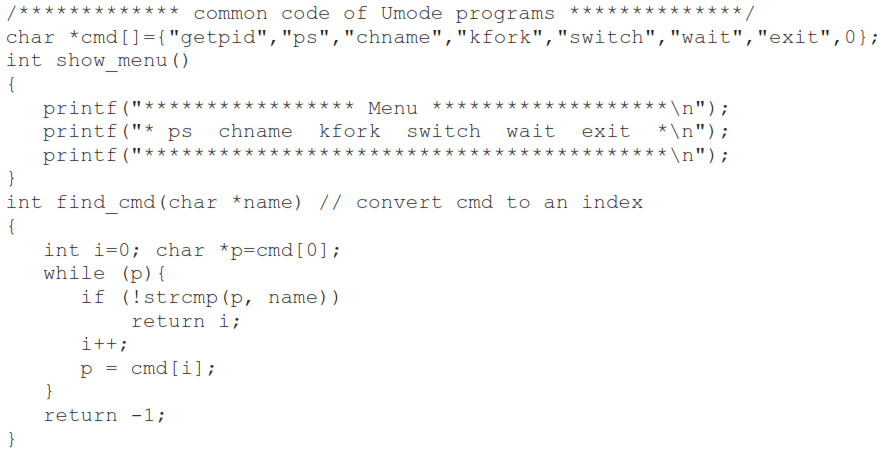
Cuando la ejecución de la imagen Umode comienza, el ustack está lógicamente vacío. Tan pronto como la ejecución empieza, el contenido de ustack cambiará. Mientras el proceso continúa en ejecución, los marcos de ustack crecerán y decrecerán como se describe en el capítulo 2. Note que cuando el control entra por primera vez en la imagen Umode, el registro bp de la CPU es inicialmente 0. Recordemos que bp es el apuntador de marco de pila en llamadas a función. Esta es la razón por la que la lista enlazada de marco de pila de Umode termina con un 0.

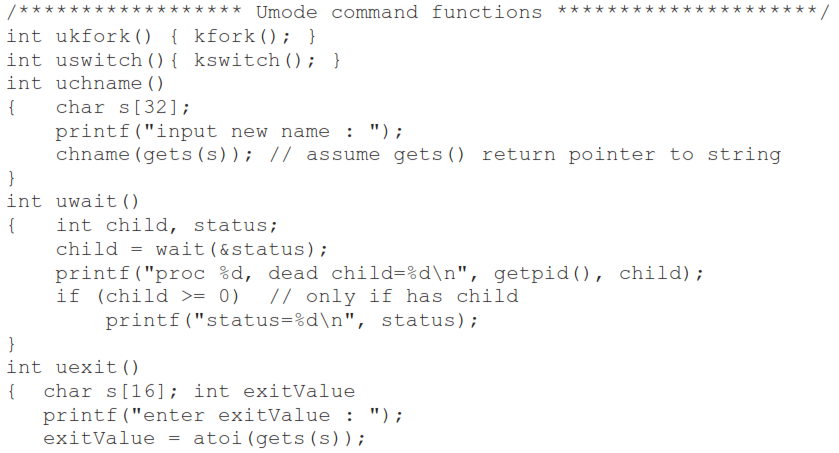
En la función body(), agregar un comando ‘u’, el cual llama a goUmode() para permitir al proceso que está corriendo regresar al Umode. Alternativamente, podríamos modificar kfork() para establecer el punto de resume de cada proceso nuevo creado a goUmode(). En ese caso, un proceso regresaría al Umode inmediatamente cuando comienza a correr. Lo siguiente enlista los cambios al código del sistema MTX5.0.

5.8 MTX5.0: Demostración de Llamadas al Sistema Simples



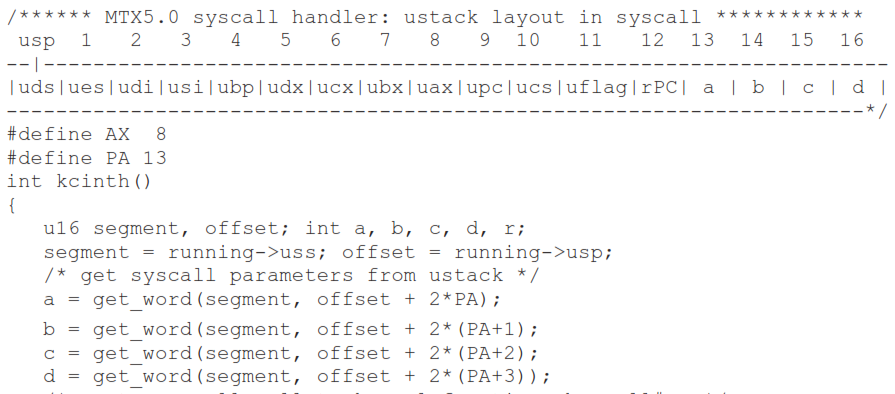


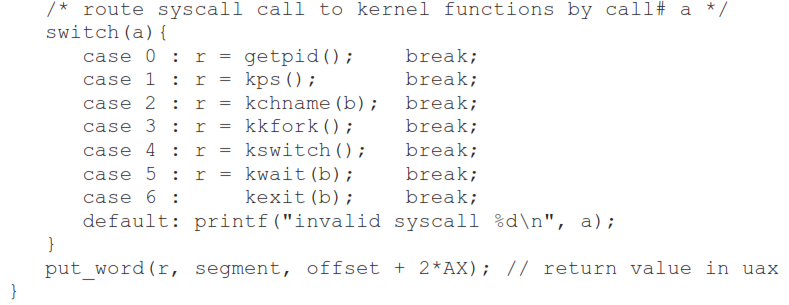






Cada syscall es enrutada a una función de kernel correspondiente por la función manejadora de syscall, kcinth() en el kernel, la cual se muestra abajo.





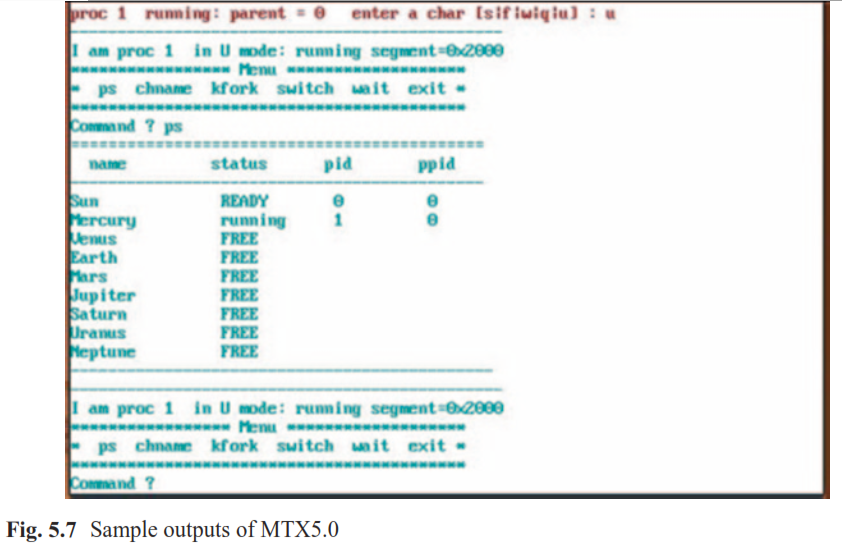
Si el número de syscalls es pequeño, es suficiente usar una tabla switch para enrutar las syscalls. Cuando el número de syscalls es grande, es mejor usar una tabla branch que contenga apuntadores a función del kernel. Lo siguiente ilustra la técnica de tabla de apuntadores a función.

Pag. 148 del libro (\approx al inicio) K. C. Wang, Design and Implementation of the MTX Operating System, Springer International Publishing, 2015.

5.8.1 Validación de Parámetros de Llamadas al Sistema

Además de ilustrar la diferencia entre espacios de direcciones de modo usuario y de modo kernel, también se intenta que la syscall chname ponga de manifiesto otro punto importante, y es que el kernel debe validar los parámetros de la syscall. En la syscall chname(char \*newname), el parámetro newname es una dirección virtual en el espacio de usuario. ¿Qué tal si es una dirección inválida? En el MTX de modo real, esto no puede suceder dado que un offset en un segmento siempre es una dirección virtual válida. En sistemas con hardware de protección de memoria, el kernel podría generar una excepción si trata de acceder a una dirección inválida. Además, la syscall chname también tiene un parámetro de valor implícito, la longitud de la cadena newname. En el kernel MTX5.0, el campo name de cada PROC tiene solamente espacio para 32 caracteres. ¿Qué tal si el usuario trata de pasar una cadena newname de más de 32 caracteres? Si el kernel simplemente acepta la cadena completa, podría desbordar el campo name de PROC y escribir a algunas otras áreas en el espacio de kernel, haciendo que el kernel choque (crash). Similarmente, en algunas syscalls el kernel podría escribir información al espacio de usuario. Si el parámetro dirección de la syscall es inválido, el kernel podría generar un error de protección o escribir a la imagen de proceso equivocada. Por estas razones, el kernel debe validar todos los parámetros de la syscall antes de procesar la syscall.

MTX5.0 demuestra las transiciones de Umode a Kmode y syscalls simples. La figura 5.7 presenta la salida muestra de ejecutar MTX5.0 sobre QEMU. El lector podría correr el sistema para poner a prueba otros comandos de syscall.

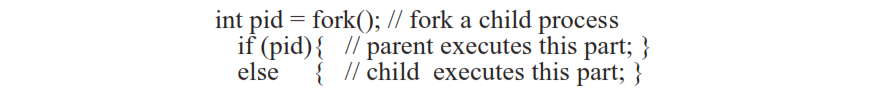


5.9 fork-exec en Unix/Linux

En Unix/Linux, la llamada al sistema

int pid = fork();

crea un proceso hijo con una imagen de Umode idéntica a la del padre. Si tiene éxito, fork() devuelve el pid del proceso hijo. En caso contrario, devuelve -1. Cuando el proceso hijo corre, regresa a su propia imagen de Umode y el pide devuelto es 0. Esto es la base el código C en los programas de modo usuario



El código usa el pid devuelto para para distinguir entre los procesos padre e hijo. Regresando de fork(), el proceso hijo usualmente usa la llamada al sistema

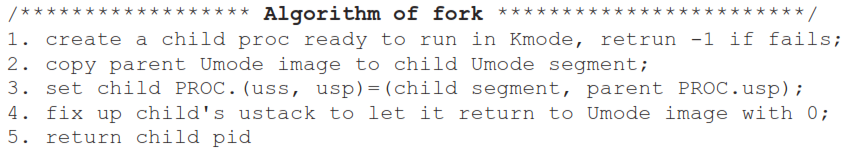


para cambiar la imagen de ejecución por la de un programa diferente y regresar al Umode a ejecutar la nueva imagen. Si tiene éxito, exec() justamente remplaza la imagen de Umode con una nueva imagen. Aun es el mismo proceso pero con una imagen de Umode diferente. Esto permite que un proceso ejecute diferentes programas. En general, exec toma más parámetros que un solo nombre de archivo (filename). Los parámetros extra, conocidos como parámetros de línea de comandos, son pasados a la nueva imagen cuando la ejecución empieza. Para empezar, consideraremos exec con solamente un filename primero, y consideraremos los parámetros de línea de comandos después.

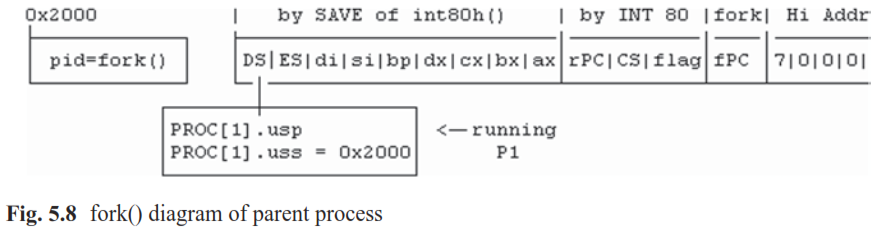
Fork y exec podrían ser llamados el pan y la mantequilla de Unix porque casi cualquier operación en Unix depende de fork-exec. Por ejemplo, cuando un usuario introduce un comando, el proceso sh con fork tiene un proceso hijo y espera a que el hijo termine. El proceso hijo usa exec para cambiar su imagen por la del archivo comando y ejecuta el comando. Cuando el proceso hijo termina, despierta al padre sh, el cual despliega el prompt para esperar por otro comando, etc. mientras Unix usa el paradigma fork-exec, el cual crea un proceso para ejecutar un programa diferente en dos pasos, hay esquemas alternativos. En el sistema operativo MVS [IBM MVS], el sistema llama a créate(filename) crea un proceso hijo para ejecutar filename en un paso, y attach(filename) permite a un proceso ejecutar un nuevo archivo sin destruir la imagen de ejecución original. En las secciones siguientes, implementaremos fork y exec exactamente como están implementadas en Unix.

5.9.1 Implementación de fork en MTX

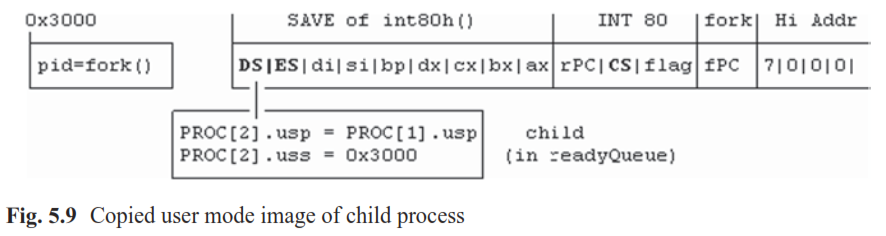
Primero presentamos el algoritmo de fork. Entonces explicamos los pasos con un ejemplo específico.



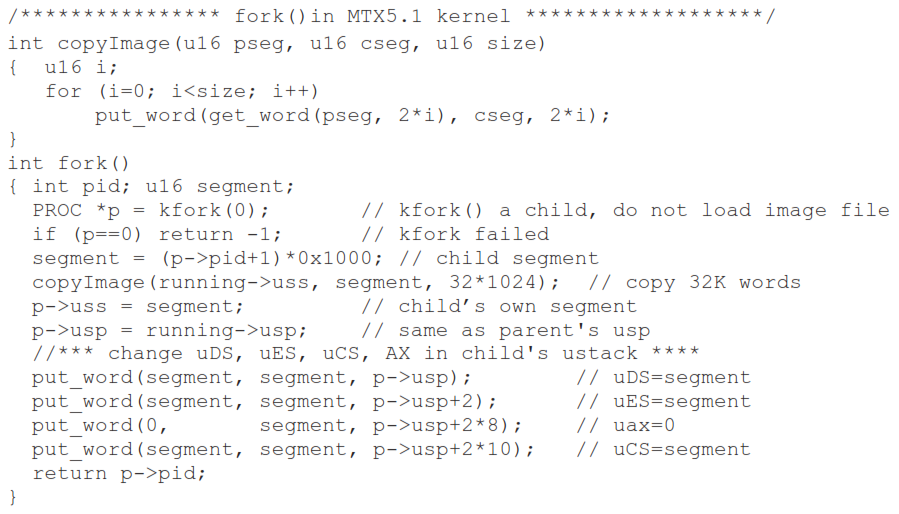
Suponga que el proceso P1 llama a pid=fork() desde Umode, como se muestra en la figura 5.8.



En la figura 5.8, en el lado izquierdo se muestran las secciones de Código y de Datos y el lado derecho muestra los cambios en su ustack. Cuando P1 ejecuta fork(), llama a syscall(7,0,0,0) en ensamblador. El fPC guardado en su ustack apunta al punto de retorno en fork. En \_syscall, ejecuta INT 80 para entrar al kernel MTX. En int80h(), guarda los registros de la CPU en ustack y guarda uSS=0x2000 y uSP en PROC.(uss,usp). Entonces, a llama kfork(0) para crear un proceso hijo P2 en el segmento 0x3000 pero no carga archivo de imagen de Umode alguno. En lugar de eso, copia la imagen de Umode de P1 completa al segmento del hijo. Esto hace la imagen de Umode del hijo idéntica a la del padre. La figura 5.9 muestra la imagen copiada del proceso hijo.



Dado que la imagen de Umode de PROC[2] está en el segmento 0x3000, su uss guardado debe ser establecido a 0x3000. Dado que usp es un offset relativo al segmento de PROC, el usp guardado de PROC[2] debe ser el mismo que el del padre. Sin embargo, si le permitimos al hijo regresar al Umode como está, éste iría a goUmode con el contenido de ustack como se muestra en la figura 5.9, causando que regresara al segmento 0x2000 dado que los registros DS, ES, CS copiados en su ustack son todos 0x2000. Esto enviaría a P2 a ejecutar en el segmento de P1 también, lo cual no sería bueno dado que los procesos interferirán uno con otro. Lo que debemos hacer es enviar a P2 de regreso a su propio segmento. Así que debemos arreglar la ustack de P2 antes de permitirle ejecutar goUmode. Para permitir al proceso hijo regresar a su propio segmento, debemos cambiar los DS, ES y CS copiados al segmento de hijo 0x3000. Todas las otras entradas en la imagen copiada, tales como rPC y fCS, no necesitan cambio alguno dado que son desplazamientos (offsets) relativos a un segmento. Para hacer que el hijo devuelva pid=0, simplemente cambiamos su valor de uax guardado a 0. Con esas modificaciones, el hijo regresará a una imagen de Umode idéntica a la del padre pero en su propio segmento 0x3000. Debido a que los valores de rPC y fPC copiados, el hijo regresará al mismo lugar que lo hace el padre, i.e. a la sentencia pid=fork(); como si él hubiese llamado a fork() antes, excepto que el pid retornado es 0. El código de fork() en el kernel MTX5.1 se muestra abajo.

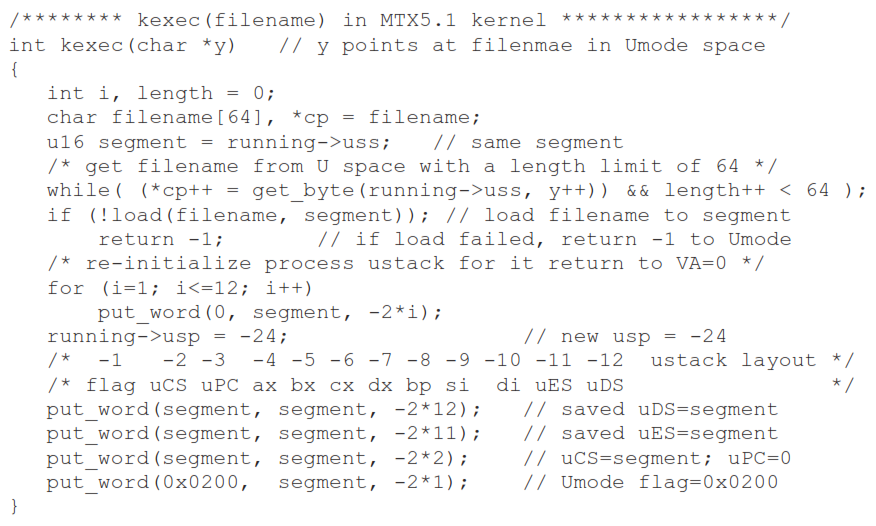


5.9.2 Implementación de exec en MTX

La implementación de exec también es muy simple. El algoritmo de exec es

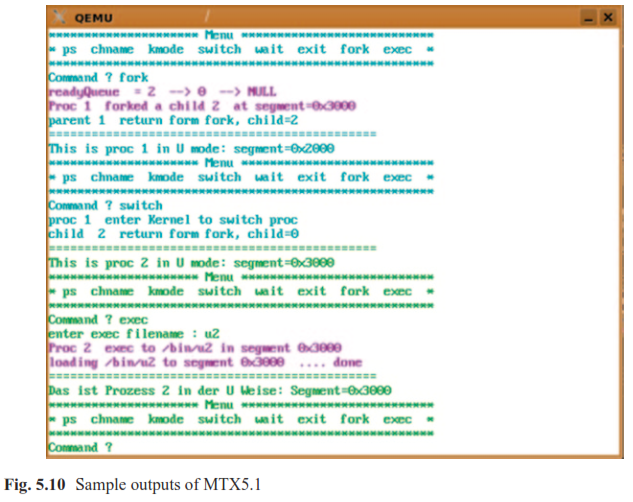


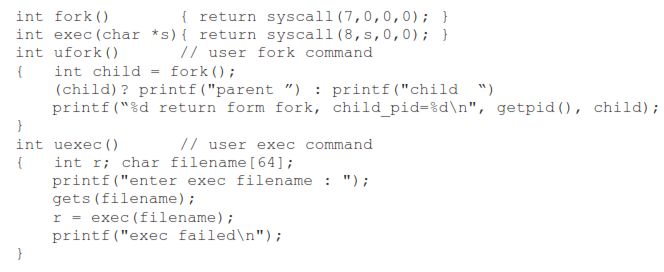
Note que exec() devueleve -1 si la operación falla. Cuando tiene éxito no devuelve valor alguno. De hecho, cuando tiene éxito, nunca regresa. El código C de kexec() se muestra abajo.



5.9.3 MTX5.1: Demostración de fork-exec en MTX

Para poner a prueba fork y exec, agregamos los comandos de usuario fork, exec y las correspondientes funciones interface de syscall a MTX5.1.

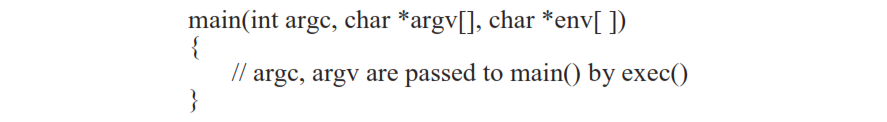




Para demostrar exec, necesitamos una imagen de Umode diferente. El archivo de imagen u2 es el mismo que u1 excepto que se despliega en alemán. El lector podría crear otros archivos de imagen de Umode para diferentes lenguajes, solo por diversión. La figura 5.10 presenta la salida muestra de la ejecución de MTX5.1.

5.10 Parámetros de Línea de Comandos

En Unix/Linux, cuando un usuario introduce una línea de comandos = “cmd a1 a2 . . . an”, sh llama a fork para crear un proceso hijo para ejecutar el programa cmd, lo cual puede ser escrito como



Cuando entra a main, argc=n+1, argv apunta a un arreglo de apuntadores a cadena terminados en null, cada uno de esos apuntadores apunta a una cadena parámetro, como se muestra en la figura 5.11.



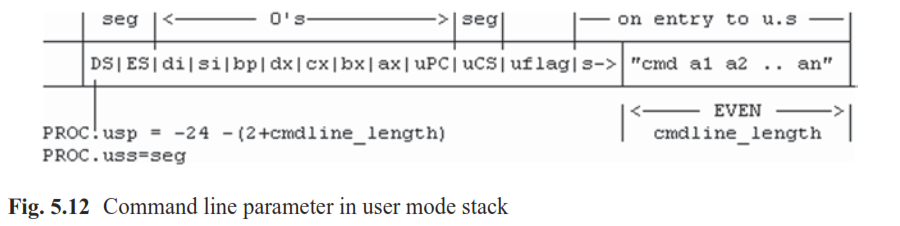
EL parámetro env apunta a un arreglo terminado en null que contiene apuntadores a cadenas de ambiente similar a argv. Por convención, argv[0] es el nombre del programa y argv[1] a argv[argc-1] son los parámetros de línea de comando para el programa. En Unix/Linux, los parámetros de la línea de comando son ensamblados en Umode antes de llamar a kexec en el kernel a través de la syscall



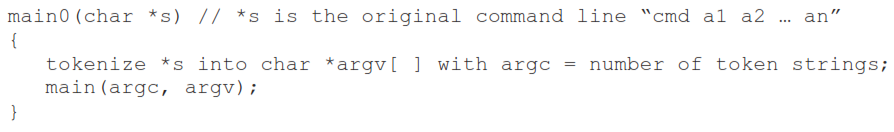
El kernel Unix pasa argv y env a la nueva imagen. Debido a las limitaciones de espacio en el kernel MTX, implementaremos parámetros de línea de comando de una forma diferente, pero con los mismos resultados finales. En MTX, la syscall exec toma la línea de comando completa como parámetro, i.e.



Modificamos kexec en el kernel usando el primer token , cmd, como el nombre de archivo (filename). Despues de cargar el filename, establecemos el ustack de la nueva imagen de Umode como se muestra en la figura 5.12.

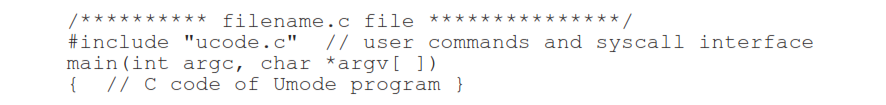


Primero, complementamos (si es necesario) la línea de comando con un byte extra para hacer la longitud total par. Después, ponemos la cadena completa en el extremo alto de ustack y hacemos que s apunte a la cadena en ustack. Entonces creamos un marco de pila de interrupción de syscall para que el proceso valla a goUmode, como se muestra en el lado izquierdo de la figura 5.12. Cuando la ejecución comienza en el Umode, la cima de ustack contiene s, el cual apunta a la cadena de comando en ustack (Umode?). De acuerdo con esto, modificamos u.s para llamar a main0(), la cual “parsea” la línea de comando en tokens cadena y entonces llama a main(int argc, char \*argv[]). El algoritmo de main0() se muestra aquí:

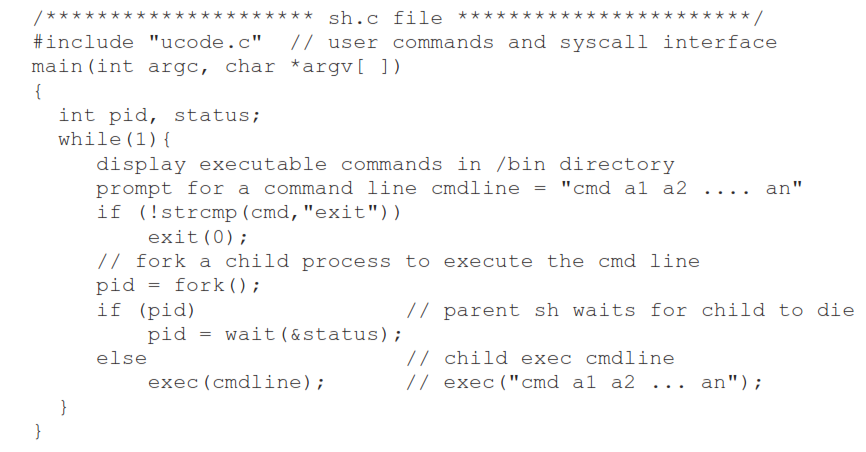


5.11 Sh Simple para Ejecución de Comando

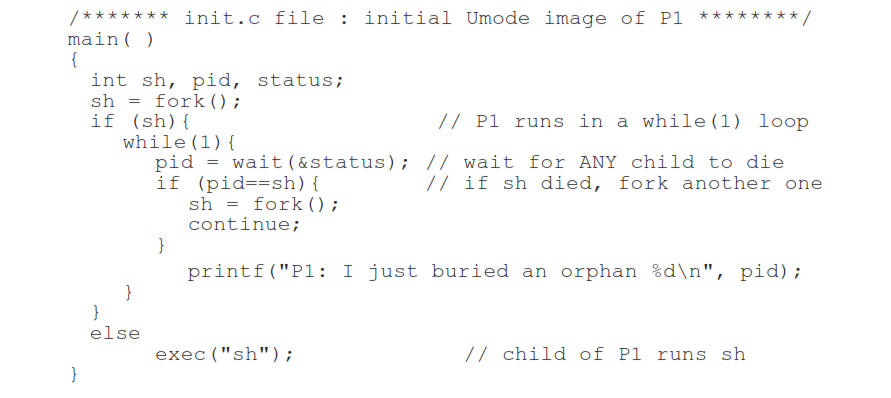
Con fork y exec, podemos estandarizar la ejecución de comandos de usuario con un programa sh simple. Primero, precompilamos main0.c como crt0.o y lo ponemos en la biblioteca de enlazado mtxlib como el código de inicialización de todos los programas de Umode de MTX. Entonces escribimos programas de Umode en C como



Entonces implementamos un sh rudimentario para ejecución de comandos como sigue.

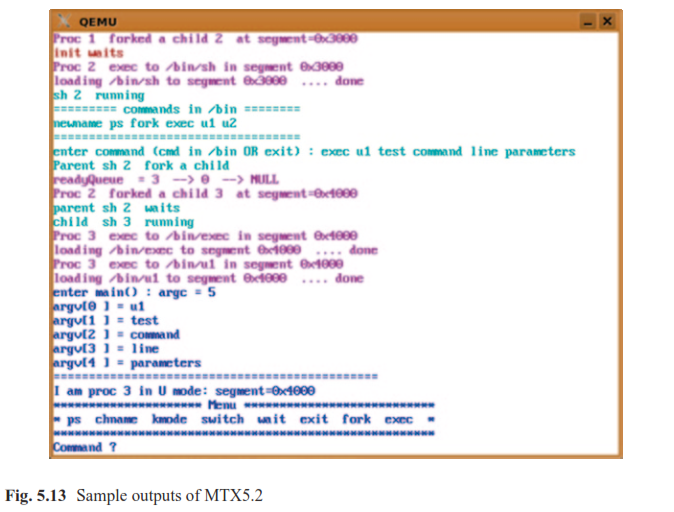


Entonces compilamos todos los programas del Umode como ejecutables binarios en el directorio /bin y corremos sh cuando comienza el sistema MTX. Esto se puede mejorar cambiando la imagen de Umode de P1 a un archivo init.c.



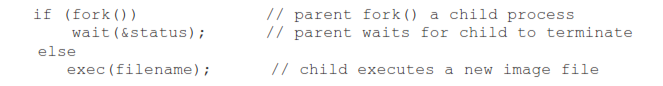
El sistema MTX5.2 demuestra un sh simple para ejecución de comandos. En MTX5.2, P1 es el proceso INIT, el cual ejecuta el archivo /bin/init. Este usa fork para crear un proceso hijo P2 y espera por algún hijo ZOMBIE. P2 hace la syscall exec para convertirse en el proceso sh. El proceso sh corre en un ciclo while(1), en el cual despliega un menú y pide una línea de comando para ejecutar. Todos los comandos en el menú son programas de modo usuario. Cuando el usuario introduce una línea de comando de la forma cmd lista-de-parámetros, sh hace fork para que un proceso hijo ejecute la línea de comando y espera a que el proceso hijo termine. El proceso hijo usa exec para ejecutar el archivo cmd, pasándole la lista-de-parámetros al programa. Cuando el proceso hijo termina, éste despierta al proceso sh, el cual despliega un prompt para esperar por otra línea de comando. Cuando el proceso sh mismo termina, el proceso INIT P1 bifurca (forks) otro proceso sh, etc.

Eventualmente, expandiremos init.c para permitir que P1 bifurque varios procesos login sobre diferentes terminales para que los usuarios se registren. Cuando un usuario hace login (se registra), el proceso login se convierte en proceso de usuario y ejecuta sh. Esto haría el ambiente de ejecución de MTX idéntico al de Unix/Linux. La figura 5.13 presenta salidas muestra de MTX5.2.



5.12 vfork

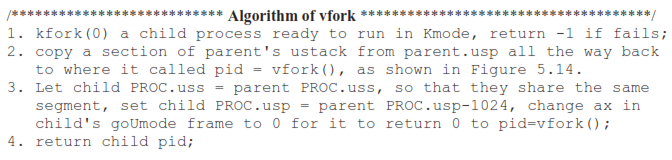
En un sistema tipo Unix los comportamientos usuales de los procesos padre e hijo son como sigue.



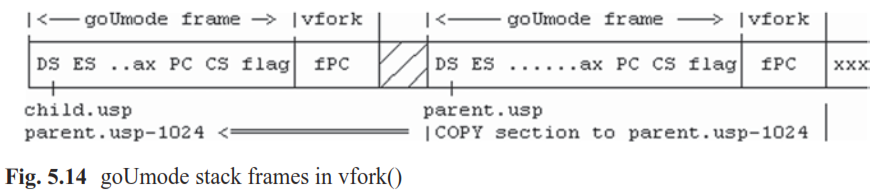
Después de crear un hijo, el padre espera a que el hijo termine. Cuando el hijo corre, el cambia la imagen de Umode a un nuevo archivo. En este caso, copiar la imagen en fork sería un deperdicio dado que el proceso hijo abandona la imagen copiada inmediatamente. Por esta razón, la mayoría de los sistemas Unix soportan una vfork, la cual, es similar a fork pero no copia la imagen del padre. En lugar de eso, el proceso hijo es creado para compartir la misma imagen con el padre. Cuando el hijo hace exec, él solamente se desengancha a símismo de la imagen compartida sin destruirla. Si todo proceso hijo se comportara de esta forma, el esquema funcionaría bien. Pero ¿qué tal si los usuarios no obedecen esta regla y permiten al hijo modificar la imagen compartida? Esto alteraría la imagen compartida, causando problemas a ambos procesos. Para prevenir esto, el sistema debe apoyarse en protección de memoria. En sistemas con hardware de protección de memoria, la imagen compartida puede ser marcada como de solo lectura así que un proceso puede solamente ejecutar la imagen, pero no modificarla. Si algún proceso necesita modificar la imagen compartida, se deben utilizar imágenes separadas. Hasta este punto, MTX corre sobre máquinas Intel x86 en modo real, el cual no tiene mecanismo de protección de memoria. A pesar de ello, podemos implementar vfork también, siempre que el proceso hijo solamente ejecute exec sin modificar la imagen compartida.

5.12.1 Implementación de vfork en MTX

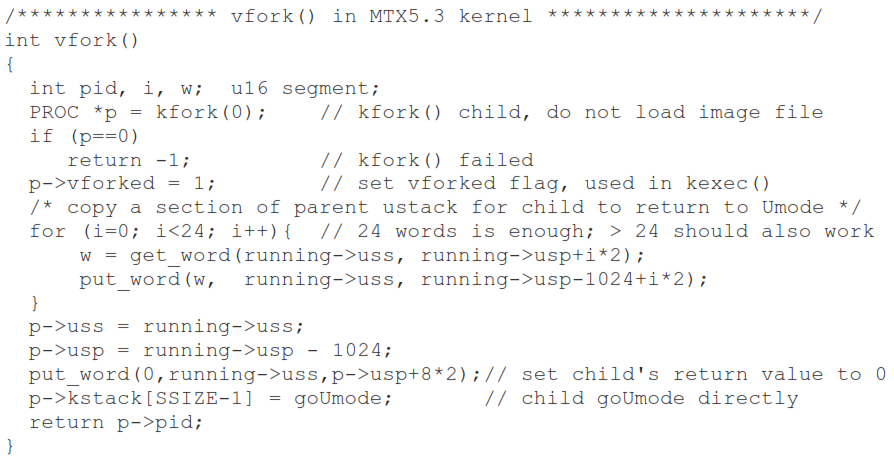
El algoritmo de vfork es como sigue.



La figura 5.14 ilustra el paso 2 del algoritmo de vfork.

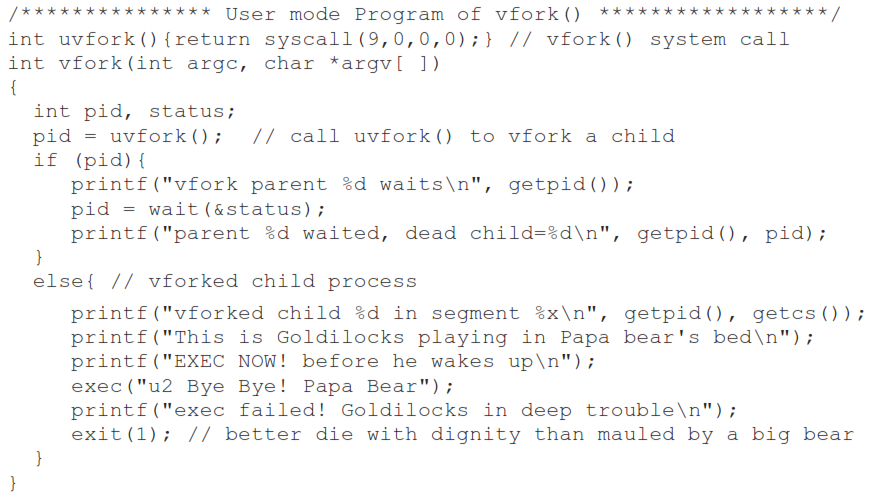


En la figura 5.14, el lado derecho muestra el marco de pila del proceso padre en una llamada al sistema vfork(). El marco de pila es copiado a un área baja en el ustack del padre como el marco de pila de Umode del proceso hijo bifurcado con vfork (vforked), como se muestra en el lado izquierdo de la figura. Cuando el padre regresa, ejecuta goUmode() con el marco de pila de parent.usp, y el valor devuelto es el pid del hijo. Cuando el hijo corre, ejecuta goUmode() con el marco de pila de child.usp. Los marcos de pila son idénticos, excepto por los valores de retorno. Por lo tanto, el hijo también regresa a pid=vfork() con un 0. Cuando el hijo regresa al Umode se ejecuta en la misma imagen que el padre. Entonces realiza una syscall exec para cambiar la imagen. Cuando el hijo entra a kexec, el segmento de umode aún es el del padre. El hijo puede obtener la línea de comando y entonces hacer exec a su propio segmento. Para dar soporte a vfork, kexec solamente necesita una ligera modificación. Si el que llama es un proceso bifurcado con vfork, en lugar del segmento actual de quien llama, esta carga la nueva imagen al segmento por defecto de quien llama (por pid), en consecuencia, desenganchándolo de la imagen del padre. Lo siguiente muestra el código de vfork en el kernel MTX5.3.



5.12.2 MTX5.3: Demostración de vfork en MTX

MTX5.3 demuestra vfork en MTX. Para hacer esto, agregamos una interface de syscall uvfork y un programa comando vfork de modo usuario.



Cuando se esté ejecutando el comando vfork este mostrará que cada proceso hijo bifurcado con vfork comienza su ejecución en el mismo segmento del padre, como Risitos de Oro jugando en la cama de Papá oso. Si el proceso hijo realiza exec a su propio segmento antes de que el padre corra o mientras el padre está durmiendo en wait(), el Papá oso nunca sabría que Risitos de Oro jugó en su cama antes. La figura 5.15 presenta la salida muestra de correr el comando vfork en MTX5.3.



5.13 Hilos (Threads)

5.13.1 Principio de Hilos

POR AHORA OMITO ESTA SECCION (5.13), PERO ES IMPORTANTE REVISARLA LO ANTES POSIBLE. 2020.09.22.

Pag. 160 del libro K. C. Wang, Design and Implementation of the MTX Operating System, Springer International Publishing, 2015.

5.14 Implementación de Hilos en MTX

POR AHORA OMITO ESTA SECCION (5.14), PERO ES IMPORTANTE REVISARLA LO ANTES POSIBLE. 2020.09.22.

Pag. 163 del libro K. C. Wang, Design and Implementation of the MTX Operating System, Springer International Publishing, 2015.