Sistemas Operativos

ITBA
Cuatrimestre 1 - 2012

Trabajo Practico 2: Multitasker

Mader Blanco, Conrado, 51270
Ramundo, Federico, 51596
Mehdi, Tomás, 51014

Introducción

En este trabajo se realizó un programa que conmuta diferentes procesos, asignándoles a cada uno un tiempo de ejecución. El multitasker no corre sobre ningún Sistema Operativo, se ubica en memoria utilizando el *bootloader* GRUB.

Objetivo

El objetivo del trabajo es la implementación de un Multitasker, cuyo objetivo es asignar tiempo de ejecución a diferentes procesos en memoria. El sistema deberá ser implementado para plataformas Intel de 32 bits, utilizando el procesador en modo protegido. El multitasker debe ser preemptivo, es decir, cualquier tarea puede ser desalojada del microprocesador. El encargado de administrar el CPU es el scheduler el cual tomará como base de tiempo la interrupción de hardware INT8 correspondiente al timer tick, para realizar la asignación de tiempo. Fue necesario realizar dos algoritmos distintos de scheduling, en donde al menos uno debí manejar prioridades. Además debían poder realizarse procesos en background (indicándolo con el signo & al final de la sentencia, al igual que linux) y fue necesaria la implementación de *yield, kill y top.*

Diseño y estructuras

Las estructuras diseñadas para llevar a cabo el sistema son:

task_t

- Nombre del proceso
- Estado (RUN, BLOCK, READY o FREE)
- *Process ID*
- *Parent Process ID*
- Prioridad (0 máxima, 4 mínima)
- ticks (utilizado para bloqueo de un proceso por tiempo)
- timeBlocks (utilizado para el calculo del porcentaje de uso del procesador)
- un flag *input*
- Información acerca de stack y heap (ss, ssize, heap)
- Puntero a estructura stackframe_t
- Puntero a estructura tty_t

stackframe_t

- Registros del microprocesador
- Dirección de retorno
- Argumentos del proceso (int argc, char** argv)

tty_t

- Numero de tty
- Contenido del video
- Color de letra
- Cursor coordenada X

- Cursor coordenada Y

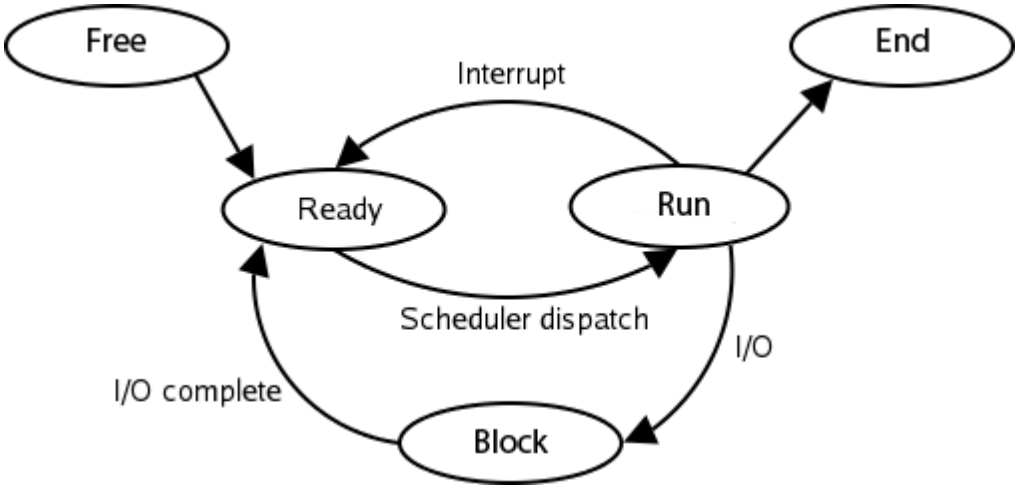
procTopInfo_t
- Nombre del proceso
- *Process ID*
- Porcentaje del uso del procesador
- Cantidad de páginas ocupadas por el proceso
- Estado del proceso

topInfo_t
- Vector procTopInfo_t
- Cantidad de procesos

head_t (utilizada en el Heap)
- Distancia al próximo y al anterior head.
- Flag que indica si esta libre o no

Los procesos pueden tener uno de los cuatro estados existentes a la vez. Estos estados son:

- RUN: El proceso esta ejecutándose en ese momento.
- READY: El proceso está listo para ejecutarse, esperando el control del procesador.
- BLOCK: El proceso se encuentra bloqueado, no puede ejecutar.
- FREE: No hay proceso.



Esquema de transiciones entre estados de procesos

A su vez, el scheduler posee las siguientes variables globales:

- Un vector de task_t
- El índice de la tarea actual.
- La cantidad de tareas
- La tarea *idle* que se ejecuta cuando no hay procesos
- El vector de estados utilizado en uno de los algoritmos de scheduling

Implementación

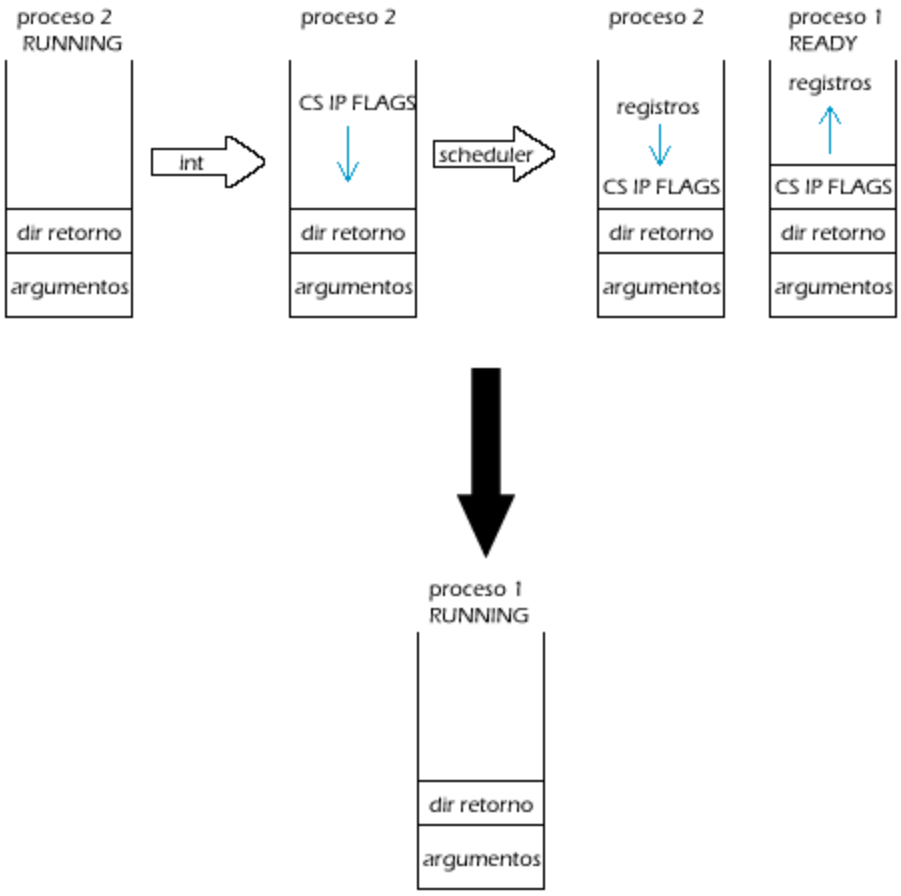
Nuestro sistema esta formado de la siguiente forma: El bootloader GRUB levanta a memoria cierta cantidad de páginas, de los cuales, las primeras 530 (4Kb cada una) corresponden al espacio de Kernel. El resto de las páginas, deshabilitadas inicialmente, pertenecen al espacio de usuario, en donde se habilitan las páginas en medida que se van necesitando. A continuación explicaremos en detalle el funcionamiento del sistema.

Cambio de contexto y tarea

Cuando se produce una interrupción de timer tick, se ejecuta el código del scheduler, que determina qué proceso debe correr a continuación. Esto se realiza de la siguiente forma:

1. Se guarda el stack actual, y se utiliza uno auxiliar perteneciente al Kernel para realizar las operaciones.
2. Se llama al código del scheduler, que determina el siguiente proceso a ejecutar.
3. Se carga el código del nuevo programa.

Cada proceso posee su propio stack. El procesador cambia el stack en el que esta trabajando constantemente. Cuando se llama una interrupción de timer tick, se apilan los registros CS, EIP y EFLAGS, (dado que es una interrupción). El scheduler determina el nuevo proceso a ejecutar y se cambia el ESP hacia el stack de dicho proceso, de donde se desapilan CS, EIP y EFLAGS. Notar que estos registros apilados no son los mismos que apilo el proceso anterior, dado que se trata de un stack diferente. Cuando se le termine el tiempo de ejecución, el proceso volverá a apilar estos registros y se llama al scheduler nuevamente.



Cuando no hay programas para ser ejecutados se cuenta con uno llamado *idle* que mantiene ocupado el procesador hasta que hayan procesos por correr.

Algoritmos de scheduling

Algoritmo 1: Round Robin sin prioridad

El primer algoritmo es simple: itera por los procesos buscando alguno que este listo para ejecutar y le otorga el procesador a éste. En la siguiente interrupción continua iterando por los procesos, pero empezando por el último que se ejecutó, dado que de hacerlo de otra forma se ejecutarían siempre los que estén al principio de la estructura donde se almacenan los procesos.

Algoritmo 2:

Para este algoritmo se posee por cada proceso un numero de estado del 0 al 100. Cuando se desea obtener un nuevo proceso el scheduler aumenta el estado de cada proceso, despendiendo de su prioridad (a más prioridad, más rápido crece el estado). Cuando un estado llega o supera el numero 100, se le otorga el procesador y se vuelve a poner en 0.

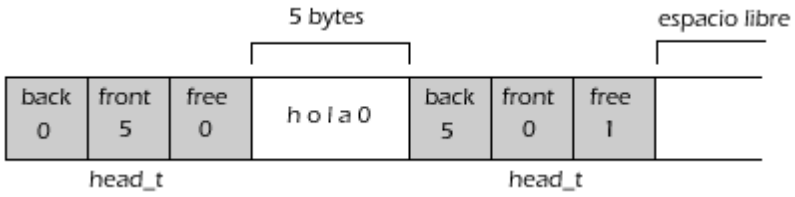
De esta forma, los procesos con más prioridad tendrán más seguido el control del procesador, dado que su estado crece más rápidamente.

Este algoritmo fue inspirado en el videojuego *Final Fantasy* en el que cada jugador posee una

barra de estado que crece según la velocidad del personaje, y que cuando se llena, tiene la posibilidad de actuar.

Heap

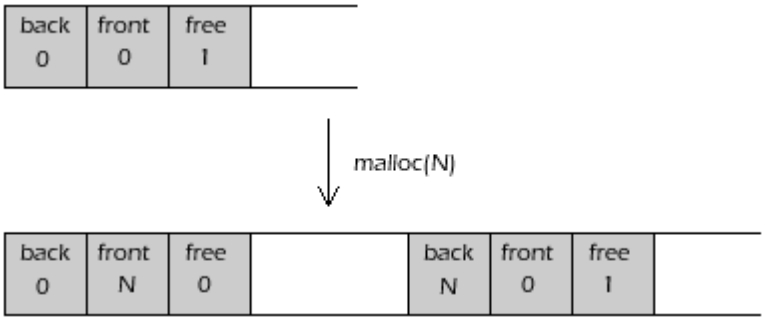
Para el manejo de memoria dinámica poseemos una estructura formada por una serie de headers, que contienen la información de la memoria reservada. Cada header cuenta con información acerca de la distancia en bytes hacia el siguiente header y hacia el anterior, y un flag que indica si esta libre o no (se utiliza a modo de baja lógica).



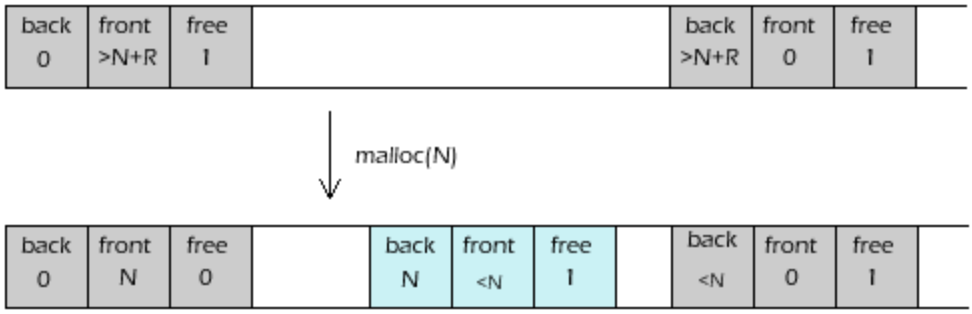
Reserva de memoria

El espacio que hay entre los headers corresponde a la zona de memoria reservada mediante el comando *malloc.* De esta forma, cuando se desea reservar N bytes de memoria, el algoritmo del heap va recorriendo los headers hasta que encuentra lugar donde localizar estos bytes. Pueden ocurrir las siguientes situaciones en donde se puede reservar la memoria.

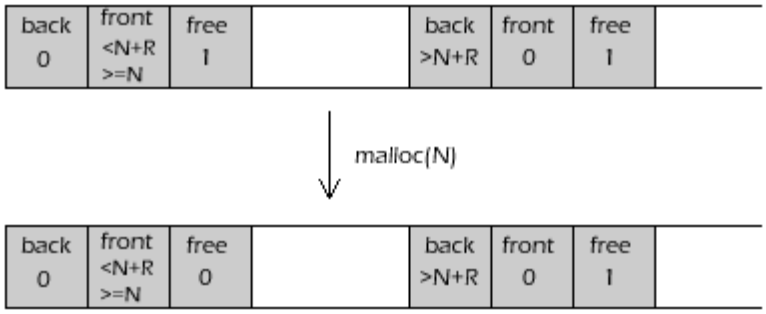
- Se encuentra un header cuya distancia al próximo es 0, lo quiere decir que es el último, por lo que lo agrega al final, y crea un header nuevo al fondo de los N bytes reservados.



- Se encuentra un header con el flag free prendido, y la distancia al siguiente es mayor que los bytes que se desean reservar, (más un pequeño offset R, dado que debe haber lugar además para un nuevo header al final, y un pequeño espacio para que entre una variable entre el nuevo header y el siguiente). En este caso se devuelve esta zona de memoria, se sobrescribe el header y se crea un nuevo header al final con lugar para una nueva variable.



- Se encuentra un header con el flag free prendido, y la distancia al siguiente es mayor que los bytes que se desean reservar, (pero no alcanza para crear un nuevo header). En este caso se devuelve esta zona de memoria y se sobreescribe el header.

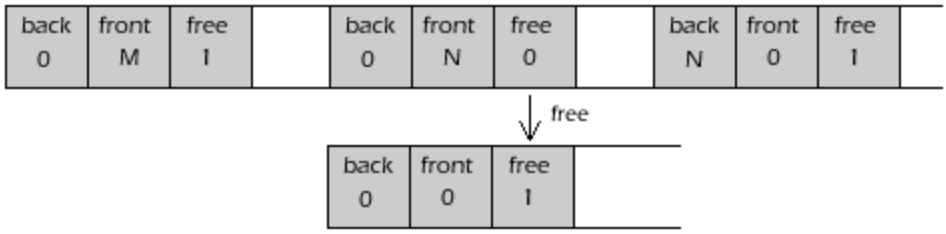


- Si por otro lado, encuentra un header ocupado, o uno libre pero en donde no posee lugar suficiente para los N bytes que se quieren reservar, se sigue hasta el siguiente header.

Liberación de memoria

Cuando una zona de memoria es liberada lo primero que se hace es darla una baja lógica activando el flag *free* del header. Lo que se realiza a continuación es controlar que las zonas de memoria aledañas estén liberadas también. Si lo están, se unen dichas zonas actualizando los headers como corresponde.

Al realizar el liberado de esta forma es imposible que hayan dos o más zonas vecinas libres, por lo que cuando se desea liberar una, sólo hay que controlar por las que están pegadas a esta, y no más allá.



Manejo de páginas de heap y stack

Cada proceso tiene a disposición una cantidad máxima de páginas a utilizar, es decir, un bloque de páginas (en nuestro caso, de diez). Inicialmente se asigna la última página del bloque para el stack y la primera para el heap. Se decidió distribuirlas de esta forma dado la forma en que crecen dichas estructuras.
Si el stack o el heap crece de manera que el tamaño actual le queda pequeño, el sistema le asignara una nueva página de las que tiene a disposición. Si no quedan más páginas disponibles en el bloque y se pide una más, el sistema mata al proceso.

El sistema de reasignación de páginas funciona de manera distinta en el heap que en el stack, dado que en el caso del stack no es tan fácil saber cuanto se va a necesitar con cada llamada a función o con cada variable declarada.

- Heap: Si la reserva de memoria solicitada es mayor que el tamaño que se posee, se solicitará una nueva página
- Stack: Cuando el tiempo de procesamiento del proceso actual termina, el scheduler controla que el tamaño del stack no haya superado el 80% de su tamaño total. Si esto sucede, pide otra página.

El sistema de reasignación de páginas del stack es un poco vulnerable, dado que si s el stack crece muy deprisa, podría llegar a pisar una página no habilitada sin que al scheduler le de tiempo a darle una nueva. Una forma de evitar que esto suceda sería manejar anillos de prioridades, en donde el usuario este en el anillo 3, y el kernel en el 0, donde todas las páginas están habilitadas. Cuando el usuario pisase una página no habilitada, el código de *page fault*, del lado del kernel seria el encargado de asignársela, si es posible.

Terminales

Nuestro sistema cuenta con 8 terminales. Para cambiar el foco de las terminales se presionan las teclas de función (F1 lleva a la terminal 1). Cuando se cambia de terminal, el sistema guarda la información sobre la terminal actual (textos impresos, color de letra, etc.) y carga la de la nueva terminal.

Los procesos poseen una terminal, que usan como salida standard. A su vez, solo la terminal que esta en foco recibe entrada del teclado, por lo que sólo los procesos que estén en dicha terminal podrán recibir información del teclado. Cuando un proceso se crea, se debe especificar a qué terminal corresponde.

Procesos en background

Para realizar procesos en background nos vimos obligados a realizar procesos padres y procesos hijos. De esta forma, cuando se desea correr un proceso en background sobre otro (es decir, que se ejecuten los dos a la vez) este lo crea como otro proceso, teniéndolo como hijo. Para llevar esto a cabo agregamos a la estructura *task* una variable correspondiente al *process ID* de su padre. Por lo tanto, cuando se mata un proceso, la función encargada de eliminarlo controla si algún otro proceso es hijo de este, lo mata también, y repite con los hijos de este último.

Cuando un proceso genera un hijo, éste hereda de su padre su prioridad y su terminal. Los procesos, al momento de su creación cuentan con parámetros variables, que son manejados por dos parámetros: un entero que indica la cantidad y un vector de strings con los parámetros en sí. El primero de ellos corresponde al nombre de dicho proceso.

Bloqueo por entrada

Los procesos que piden entrada del teclado quedan bloqueados hasta recibir dicha entrada. Es decir, cuando se llama a la función de sistema *getchar* el proceso queda bloqueado hasta que esta función retorna. Se decidió implementar para evitar un *busy waiting* en la entrada del teclado, sobretodo al momento de tener varias terminales, con varios procesos que pueden estar consumiendo del teclado. Para llevarlo a cabo se utiliza un flag en la estructura del proceso *task_t.*

Contamos, además, con una función que lee de entrada sin bloquearse. Fue necesario implementar este método dado que existen comandos que necesitan este comportamiento (el comando *top* que lee la letra “q” para salir).

Poseemos un único buffer circular de teclado para las 8 terminales, que solo devuelve información a los procesos pertenecientes a la terminal que tiene el foco en ese momento.

Comando top

El comando top informa cuanto procesador consume cada proceso, en forma de porcentaje, la cantidad de páginas que ocupa cada uno, el estado de dicho proceso y su *processID*. Para poder calcular este porcentaje, se cuenta con una variable en la estructura de un proceso llamada timeBlocks. Esta variable se aumenta cada vez que el proceso obtiene el control del procesador. Cuando se llama al comando top, este calcula el porcentaje dividiendo los timeBlocks del proceso en cuestión entre la suma de todos los timeBlocks, y lo multiplica por 100.

Cuando un usuario ejecuta desde la shell el comando top, se genera una *Systemcall* que genera una estructura de tipo topInfo_t. El kernel recibe un puntero esta estructura y la rellena con la información pertinente. A continuación, la función llamada por el usuario imprime en pantalla la información obtenida. Se decidió realizar esto de esta forma para no mezclar *User Space* de *Kernel Space*. Esta secuencia de pasos se repite indefinidamente de manera que se mantenga actualizada la información que se va mostrando, hasta que el usuario decide salir del proceso.

Comandos soportados en la shell

- help
  Informa al usuario los posibles comandos que pueden ser ejecutados.

- echo [string]
  Imprime en pantalla los caracteres escritos a continuación del comando.
  Ejemplo: echo HOLA MUNDO

- color [color]
  Cambia el color de la letra de escritura. Los colores soportados son:
  white, cyan, red, blue, magenta, green, orange, brown, yellow, gray, pink, light gray, light blue, light green, light cyan.

- time
  Imprime la hora actual del sistema.

- who

Imprime los autores del trabajo en pantalla.

- keyboard [layout]
  Cambia la disposición del teclado. Hay dos disponibles: español (ESP) e inglés (ENG) Ejemplo: keyboard ESP // Cambia el teclado a la distribución española.
  (La disposición actual del teclado se encuentra siempre visible en la esquina inferior izquierda de la pantalla).

- kill [processID]
  Mata al proceso con el ID indicado y a sus hijos. Imprime en pantalla el resultado de la operación.

- top
  Indica los procesos que se están corriendo en ese instante, y el porcentaje del uso de la CPU que ocupan. Para salir del comando top se presiona la letra “q”.

- malloc [bytes]
  Reserva N bytes en el heap local. Si el parámetro no es un número, se indica el error.

- free [dir]
  Libera memoria. Si el parámetro no es un número, se indica el error.

- lostquote
  Cita una frase célebre al azar de algún personaje de la serie televisiva LOST.

- mario
  Dibuja en ASCII art la cara del personaje de videojuegos Mario Bros.

- mastersword
  Dibuja en ASCII art la espada maestra del famoso videojuego “The Legend of Zelda”.

- multiply3
  Imprime la tabla del 3 infinitamente.

Memoria de comandos

La shell guarda en un buffer los últimos comandos utilizados, de manera que si el usuario presiona las flechas de arriba y abajo, es posible acceder a dichos comandos sin necesidad de volverlos a escribir, igual que en las shells convencionales.

Problemas encontrados

Paginación de heap y stack

Como primer acercamiento a la reserva de páginas para los procesos ocupábamos 2 contiguas, una de heap y una de stack. Esto funcionaba correctamente pero nos comportaba un problema al momento que necesitábamos agrandar alguna de estas estructuras. Como solución a este problema se decidió manejar las páginas de los procesos como fue explicado anteriormente.

Bloqueo de procesos

Algunos procesos deben bloquearse a la espera de una entrada. En nuestro caso a la espera de una entrada por teclado. La primera idea fue manejar esta espera con un *busy waiting*. Esta implementación hacia que un procesos que no debería utilizar el procesador lo utilizara varias veces para ejecutar un ciclo sin sentido, lo que provocaba que el sistema resulte. Para optimizar dicha cuestión realizamos una llamada a yield dentro de el ciclo del bloqueo, lo que resultaba una solución más elegante, pero no la óptima. Finalmente llegamos a la conclusión que lo mejor sería que los procesos se bloqueen y no se les ceda el procesador mientras estén bloqueados.

Procesos padres e hijos

En un principio no teníamos ninguna forma obvia para pasar el nombre de la función que ejecutaban a los hijos de un proceso, dado que todos los registros del procesador ya se estaban utilizando por la interrupción 0x80 para crear dicho proceso. Logramos solucionar este problema emulando a los sistemas reales: dado que los procesos reciben parámetros variables en su creación, se decidió pasar el nombre de dicha función como primer parámetro.

Debugging

La mayor complicación de este trabajo práctico puede ser el hecho que resulta incomodísimo y muchas veces imposible la búsqueda de errores en el código. El hecho de no poder contar ni con el *debugging* que puede lograrse con la función *printf* en algunas ocasiones provoca que te den ganas de darte de golpes contra la pared. Afortunadamente logramos salir adelante.

Conclusiones

Este trabajo nos resultó a los tres muy interesante. El hecho de continuar el trabajo de arquitectura para las computadoras y mejorarlo hasta el punto de lograr que sea multiproceso nos hizo apreciar mucho el labor que hacen las grandes compañías para lograr que un sistema operativo funcione de manera eficiente para atender cientos de tareas a la vez. Cada linea de código debe ser pensada cuidadosamente, ya que un error puede llevar a una falla en el sistema.