Organización del computador 2

Primer parcial

Martin Santesteban
43087400
397/20

Ejercicio 1: Convención C (64 bits)

Como la función es llamada desde main.c ,y se trata de una función definida en ASM, hay que tener en cuenta la convención C de 64 bits.
Esta define un contrato para poder realizar y recibir llamadas a código C de forma segura. Esta convención, al estar usando 64 bits, especifica que al llamar desde C una función en ASM el compilador llevará a cabo una serie de instrucciones para (leyendo los parámetros de la firma de la función en el llamado de C) mover la información de los parámetros a registros específicos. Por ejemplo, el primer parámetro de tipo entero irá a parar a el registro rdi, y el primer parámetro de tipo flotante a XMM0. En el caso que tengamos demasiados parámetros y se encuentren todos los registros en uso, usaremos la pila. Además, la convención asegura que el valor de retorno tiene que ser devuelto por la función en ASM en el registro rax, la pila debe estar alineada a 16 Bytes (en caso de usar funciones definidas en libx) y además, el valor de todos los registros no volátiles deben ser preservados.
La convención C divide en dos a los registros: los volátiles y los no volátiles. Los primeros no deben necesariamente preservar su valor al finalizar la ejecución, mientras que los no volátiles sí. Por esto, cuando se define la función se pushean (cuando se usan) todos los registros no volátiles al stack (prólogo), para luego poder hacer los pops necesarios para preservar los valores (epílogo). Además se tiene en cuenta el invariante de la pila: Cada push tiene su pop, de tal forma que el stack antes de llamar a la función es el mismo que el stack posterior a la ejecución.

En la función alternate_sum_5 el compilador le asigna a cada registro el valor de su parámetro correspondiente (rdi := x1, rsi:= x2, …). Luego, en el prólogo, se respeta la alineación del stack, pero no se pushean los valores de los registros no volátiles. Esto no genera ningún problema, mientras que durante la ejecución no se modifiquen sus valores. Sin embargo este NO es el caso, ya que en la línea 40 del archivo sum.asm se modifica el valor del registro no volátil r12 (para ser más específicos, se le modifica la parte baja r12d). Luego de esa línea, no hay ninguna instrucción que restaure su valor inicial, implicando que al terminar la ejecución, un registro no volátil no preservó su valor. Por ende, no se respeta el invariante de función especificado por la convención C. Esto puede generar errores graves al resumir el programa y viola el contrato.
La forma de solucionar este error sería agregando la instrucción push r12 en el prólogo, y agregando un pop r12 en el epílogo (podrían ser escritas en las líneas 26 y 45).

La segunda función alternate_sum_9 tiene un error parecido. A diferencia de la anterior, si tiene un prólogo completo. En las primeras líneas se pushean al stack todos los registros no volátiles, se usa luego r13 y al salir se lo restaura. Sin embargo, el orden en el que se hacen los pops no es el indicado: En el prólogo se pushea primero r14 y luego r15 y en el epílogo se popea primero r14 y luego r15. Esto implica que, al finalizar la ejecución, a r14 se le asigno el valor de r15 y viceversa! Esto viola, de nuevo, la convención C. La forma de solucionar el error es intercambiando las líneas

Ejercicio 2: Convención C (32 bits)

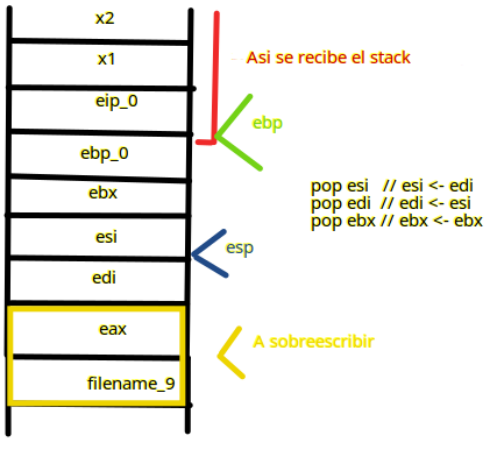
La convención C en 32 bits tiene un par de grandes distinciones en comparación con la de 64 bits. Hay solo 3 registros no volátiles (ebx, esi y edi), todos los parámetros de se pasan por la pila independientemente de su tipo(de derecha a izquierda), y la pila se mantiene alineada a 4 Bytes.

El invariante de pila se sigue manteniendo, todo push tiene su pop para que la pila vuelva al mismo estado previa a la ejecución luego de finalizar.

La primera función a analizar tiene como entrada 5 parámetros de tipo entero, de 32 bits. Por convención, sabemos que el compilador los pusheo a la pila y para realizar la suma alternada se las accede calculando el desplazamiento con respecto a ebp. Luego se ejecutará la función save_result. Para esto, el programador cumple su parte del contrato: deja en la pila los parámetros que usará la función al estar definida en C. Primero se pushea eax y luego filename_5 para que el compilador tome como primer parámetro al último elemento del stack y como segundo al anteultimo (filename := [rsp] ; result:= [rsp + 0x4]).

Luego se llama a la función y se espera que al terminar la ejecución, la pila se encuentre en el mismo estado que antes. La siguiente instrucción desplaza el esp dos posiciones arriba. Esto se hace porque el compilador no se encarga de hacer los pops para los pushes que hizo el programador, sino que se le devuelve el stack como lo dejó. Se desplaza el esp para sobrescribir los parámetros que se dejaron anteriormente. Luego se le restaura el valor a eax para devolverlo y se restaura el stack frame. El gran error, está en que se modificó el valor de ebx, cuando ebx es un registro no volátil. VIolar la convención puede generar errores mas adelante en el codigo. Si ejecutamos alternate_sum_5 muchas veces, ebx cambiará su valor al finalizar cada ejecución, acumulando basura. Este es un error que se soluciona pusheando ebx al principio del código, y restaurando su valor al final con un pop.

La segunda función sigue los mismos pasos que la primera, pero con mas parámetros. Sin embargo, tiene un prologo muy completo, donde pushea los tres registros no volátiles. Se limpia eax, se hacen las sumas y restas pertinentes y luego se ejecuta de la misma forma que en alternate_sum_5 a save_result. Notar que esta vez, si se preserva el valor de ebx al pushearlo al principio de la ejecución y popearlo al finalizar. SIn embargo, no se puede decir lo mismo con respecto a los dos registros no volátiles restantes: El orden de los pops es erróneo con respecto al de los pushes. Para cuando se finaliza la ejecución, edi y esi intercambiaron sus valores, cuando esto no debería suceder ya que son registros no volátiles. Esto se resuelve intercambiando en código en las líneas 84 y 85.



ESTADO DEL STACK EN LA LINEA 79

Ejercicio 3: Conocimiento en system programing

Tanto en 32 como 64 bits, la unidad de direccionamiento es de un byte. Esto significa que cada dirección de memoria, independientemente de la cantidad de bits que usemos para direccionarla, representa 8 bits en memoria física. Cuando el procesador se encuentra en modo real (luego de bootear) su espacio de direccionamiento se ve reducido drásticamente, debido a que, por compatibilidad, se utiliza un registro de segmento de 16 bits, junto con un offset del mismo tamaño. Esto implica que se puede representar el conjunto de direcciones [0, 2^{20}-1] (1 MB de memoria). El modo real manipula la memoria empleando segmentación de una forma rudimentaria, sin estructuras como la GDT. Usa un registro de segmento de 16 bits, se lo multiplica por 4 y se le suma un offset para direccionar el MB.

Luego de pasar al modo protegido mediante una serie de instrucciones, se pueden usar direcciones de 32 bits, dejando que se mapeen hasta 2^{32} Bytes (4GB de memoria). Sin embargo la forma de manipular la memoria es mucho más compleja. El modo protegido introduce niveles de privilegio para el uso de paginación y segmentación, además de estructuras nuevas como los directorios de pagina.
Activando el modo de 64 bits, el espacio direccionable es [0, 2^{64} - 1], logrando direccionar hasta 2^{64} Bytes.

Al decidir que cada dirección de memoria se mapea con 8 bits en memoria, también se tuvo que definir cómo se guarda la información. Se definieron contratos que hacen referencia a la endianness y alineación de los datos en memoria.
La endianness define cómo se guarda un dato con respecto al orden de sus bytes. El x86 usa little endian, haciendo referencia a que el byte menos significativo se guarda primero en memoria. Si el dato D tiene 4 Bytes de tamaño, entonces el byte menos significativo se almacenará en la posición de memoria i mientras que el byte más significativo se encontrará en i + 4.

Otro factor importante para guardar datos en memoria es el alineamiento. Por convención, todos los datos en memoria están alineados con respecto a su tamaño. Si el dato D atómico es tal que sizeof(D) = n bytes, entonces D comienza en una posición de memoria múltiplo de n. En el caso que D no sea un dato atómico sino una estructura, por contrato sabemos que estará alineado con respecto al tamaño de su atributo más grande, y además cada atributo estará alineado con respecto a su propio tamaño(como dijimos antes).

Nota:

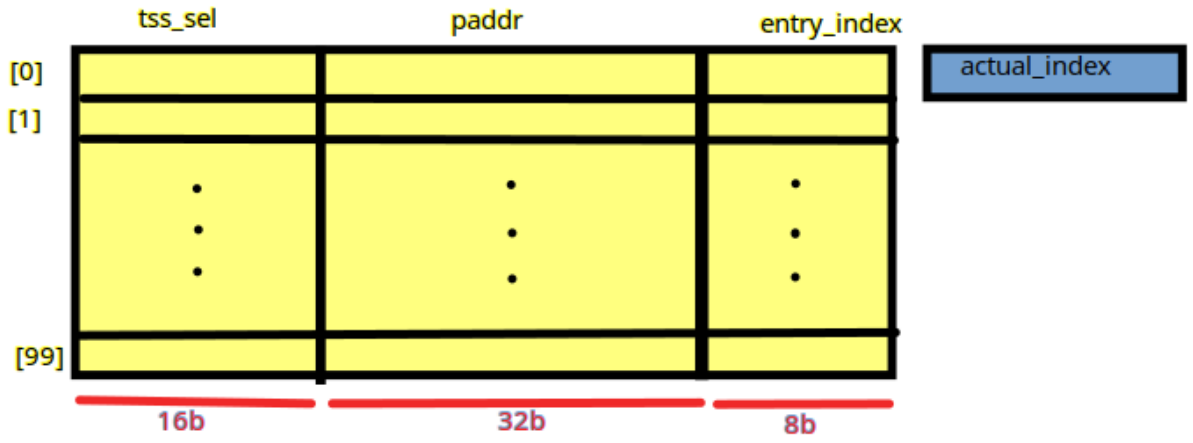
Aunque se pueda direccionar a byte, las funciones push y pop, independientemente del tamaño del parámetro, pushea 32 o 64 bits dependiendo del modo en el que se encuentre el procesador.

Ejercicio 4: Mecanismos de system programming

Para determinar cuál es la página menos usada, necesitaremos una estructura nueva. Esta actuará como tabla, donde cada entrada corresponde a una página de usuario en particular. Cada entrada estará compuesta por un selector de tarea, la dirección física de la página, y un índice que representará su número de acceso. La estructura además de contar con un arreglo de entradas, también tendrá un contador que hará referencia a el siguiente acceso. Por lo tanto, cuando la tabla está vacía, el índice actual será 0 y la siguiente modificación a una entrada tendrá como entry_index al 0.
La estructura debe poder cumplir con lo siguiente:

- Al crear una página nueva, debe crearse una entrada nueva. A esta se le asignará el índice actual y se incrementa.
- Al sacar una página (unmap) debe eliminarse su entrada de la estructura.
- Al acceder a una página, debe encontrarse su entrada y actualizar el valor por el índice actual e incrementar el índice actual.
- Si se accede a la misma pagina dos o mas veces seguidas, se le actualiza el index_entry solo la primera vez.

Cada entrada tendrá un tamaño de 44 bits (|selector de tarea| + |direccion fisica| + |indice|) y en total ocupa 551B (al sumar el índice actual) de memoria del kernel.



Donde cada tarea tiene 10 indices consecutivos, una entrada por pagina.

Esta es una estructura que deberá poder ser manipulada solo a nivel de privilegio 0.

Cada vez que se lleve a cabo un acceso a memoria/ mapeamos una página para una tarea contamos con la información suficiente como para poder determinar un intervalo de 10 entradas posible, ya que podemos dividirla en 10 secciones: Una sección de 10 entradas por tarea.

Con cada referencia a una página, el hardware se encarga de setear el bit de acceso A del descriptor de la página referenciada en la tabla de página, luego es tarea del software limpiar este bit. Cuando se haga la limpieza de bits de acceso en las tablas de página, podemos iterar entre los descriptores de las mismas y actualizar las entradas en nuestra estructura mediante un llamado a una función auxiliar(notar que además hay que iterar entre las entradas del directorio). Por lo pronto, cuanto más seguido se limpie el bit de acceso en los descriptores, mejor será la estimación de accesos. Este proceso sería muy lento en el caso de tener demasiadas páginas y tablas de página por tarea, que no es nuestro caso.

Notar lo siguiente: Si una tarea accede muchas veces a una sola página, el bit de acceso se setea una sola vez, por lo tanto en nuestra estructura se actualiza su índice de acceso una sola vez.

Para saber cual es la página menos usada, debemos iterar entre todas las entradas de la estructura y encontrar aquella cuyo índice de entrada sea el menor, ya que esa será la que hace más tiempo se actualizó.

Podemos inicializar la estructura ,junto con las tareas, con todas las entradas en 0 menos por el entry_index: Este será inicializado en -1, para que al asignar nuevas entradas, iterando, simplemente se deba hacer el desplazamiento hasta la sección de la tarea actual e iterar hasta encontrar una entrada libre (con el índice de entrada en -1).

Para optimizar el tiempo de acceso a la estructura, podemos hacer lo siguiente: Cuando se inicializa una tarea nueva, se le suma una entrada al arreglo de tareas en el scheduler. Esto se puede ver bien en la función sched_add_Task del archivo sched.c . Cada vez que se le asigna una posición en este arreglo, podemos mapear sus 10 entradas para sus páginas en nuestra estructura. Si a la tarea nueva se le asigna la posición i en sched_tasks, entonces desde el índice i * 10 hasta i * 10 + 9 de la estructura se le asignará en el campo .tss_sel el selector de la tarea nueva (notar que solo cambiaría el tss_sel, el resto de los campos se encuentran igual que al inicializar de tal forma que todavía se puede diferenciar una entrada vacía). Así, en vez de iterar entre 100 entradas, podemos iterar entre 10 al actualizar una página de alguna tarea. Si se accede a una página, solo tengo que actualizar una de 10 entradas posibles de la sección de mi tarea actual en la estructura. *

Notar que para el funcionamiento de la estructura, requerimos del arreglo de tareas del scheduler, y además deberemos poder iterar entre todas las entradas de las tablas de página al limpiar los bits de acceso en los descriptores. Por lo pronto, se usarán todas las estructuras de paginación junto con sched_tasks de nuestro kernel.

<u>* Nota:</u>

Sin embargo, usar un índice para determinar cuando se accedieron las páginas presenta un problema: El índice puede superar su valor máximo representable y comenzar a asignarle valores erróneos a las últimas páginas accedidas.

Una posible solución es limpiar la tabla entera cuando el indice pasa su valor maximo, sin embargo al tener solo 8 bits como índice esta será una medida que se lleve a cabo muy frecuentemente. Teniendo esto en cuenta, el tamaño del contador del índice debería ser lo suficientemente grande como para que la limpieza de la estructura entera se lleve a cabo la

menor cantidad de veces posible. Una alternativa (más complicada) es el uso del reloj interno del procesador: A partir del Pentium se comenzó a implementar el registro de 64 bits TSC, que cuenta la cantidad de ciclos del CPU desde el último booteo. Podemos acceder al valor de TSC con la instrucción RDTSC, que retorna el valor del registro en edx:eax. Si a cada entrada se le asignará el valor del TSC, la estructura tendrá un tamaño mucho más grande (100 * (16b + 20b + 64b) = 10000b = 1250B), pero al tener un valor tan grande como contador, se tendrá que limpiar la estructura a los 2**64 ciclos de CPU. Por lo tanto, si un CPU hace (aproximadamente) 4 * 10**9 ciclos por segundo, cada 2**64 / (4 * 10**9) = 4611686018s se deberá limpiar una tabla de 100 entradas, lo cual parecería bastante eficiente. Sin embargo, reescribir 64 bits constantemente también tiene un costo muy elevado. Debido a esto, hay que encontrar un balance entre el tamaño del contador y los tiempos de escritura.

Para implementar la función pedir_pagina_menos_usada, debemos iterar en la estructura y encontrar la entrada cuyo índice de entrada sea el menor y devolver sus datos por referencia mediante los parámetros. Abajo adjunto pseudocódigo que intenta representar la idea:

```
typedef struct lru_entry_t {
    uint16_t tss_sel;
    paddr_t  direccion_fisica;
    uint8_t  entry_index;
}lru_entry_t;


typedef struct lru_table_t {
    lru_entry_t estructura[100];
    uint8_t  actual_index;
}lru_table_t;

static lru_table_t lru_table = {0};

void init_lru_table(){
  for(int i = 0; i < 100; i++){
    lru_table[i].entry_index = -1 ;
  }

}
```

```
void pedir_pagina_menos_usada(uint16_t *tss_sel , paddr_t *pagina_menos_usada)
        int min  = 0
        for(int i = 0 ,  i < 100, i++)
                if (  lru_table[i].entry_index > (-1) &   //chequeo que no sea una entrada vacía
                    lru_table[min].entry_index > lru_table[i].entry_index){
                        min = i
        *tss_sel = lru_table[min].tss_sel
        *pagina_menos_usada = lru_table[min].paddr
}
```