Sistemas Operativos
PC2

Julio 23, 2021

Angélica Sánchez, 201710405
Alejandro Otero, 201810548
Jonathan Prieto, 201710179

---

1. Implementar en C el Contiguous Memory Allocator del libro de Silberschatz.

   Your program must respond to four different requests:

   (a) Request for a contiguous block of memory.

   (b) Release of a contiguous block of memory.

   (c) Compact unused holes of memory into one single block

   (d) Report the regions of free and allocated memory

Rpta:

(a) En la Figura1, Figura2, Figura3 y Figura4 se pueden apreciar snippets de nuestra implementación y, además se adjuntará el código fuente en el repositorio junto a este documento. Asimismo, se siguieron los pasos propuestos del libro.

En la Figura5 y Figura6 se pueden apreciar las pruebas realizadas que muestran el funcionamiento de las funciones `request, release y status`. Además, en la Figura7 se observan los outputs de errores al momento de insertar datos erróneos en la línea de comando, tales como:

- Tamaño solicitado por el método `request` fuera del rango máximo de la memoria.
- Proceso no encontrado en memoria por el método `release`.
- Fit type no identificado por el método `request`.
- Espacio insuficiente para asignar un proceso en la memoria por el método `request`.
- Espacio muy pequeño (0 o menor que 0) para asignar un proceso en la memoria por el método `request`.
- Comando no identificado.

2. Explicar su implementaciónn de forma clara y directa. Definir una secuencia de prueba y usarla para mostrar el funcionamiento correcto de su programa.

Rpta:

(a) Nuestro programa consiste en un parser (correspondiente al método `options`, el cual lee la línea de comando ingresada, la parsea y envía los argumentos al método respectivo para realizar las acciones correspondientes a este. Luego, utilizamos un struct y dos variables globales que se usan en todos los métodos: `struct memory`, `contiguos_memory` y `blocks` (correspondiente a la cantidad de bloques existentes en la memoria).

En el caso del `struct memory`, este representa los bloques de memoria y posee tres atributos:

- `pid` que se usa para identificar el id del proceso o si es que ese espacio de memoria está libre.
- `top` para guardar el límite superior del bloque de memoria.
- `bottom` el cual indica el límite inferior del bloque de memoria.

Con respecto a `contiguos_memory`, es un array de punteros, con un tamaño constante de 100, de tipo `struct memory` que almacena los bloques de memoria, y finalmente, con respecto a `blocks`, es un entero que almacena la cantidad de bloques que hay en este arreglo, puesto que `MEM_SIZE` se utiliza como una constante para el tamaño del arreglo, asumiendo que la cantidad de particiones no va a superar ese valor.

La implementación está dividida en cinco métodos principales:

1. `void allocate_memory` el cual se encarga de generar un bloque de memoria de un tamaño insertado en consola, siendo este almacenado en nuestro arreglo `contiguos_memory` con un id de U, que hace referencia a que es un espacio libre (unused) en la memoria.
2. `void request` este método se encarga de asignar la memoria con un `malloc` dependiendo del tipo de fit: first, best y worst. Esta función hace una llamada a unos de los tres métodos correspondientes al identificador que se indicó en el input, para así colocar el proceso en el espacio en memoria que le corresponda. En estos métodos se hace la lógica de asignar el proceso en un bloque ya existente o de realizar una partición para luego asignar el proceso, dentro de la memoria.
   - `void first_fit` se encarga de buscar el primer bloque en memoria donde el proceso pueda ser asignado, ya sea este del mismo tamaño que el proceso, o en caso este sea mayor, se realiza una partición.

- `void best_fit` se encarga de buscar un bloque en memoria con una diferencia en tamaño mínima con respecto al tamaño del proceso. Este puede ser del mismo tamaño que el proceso, o en caso este sea mayor, se realiza una partición dejando un espacio libre en memoria de tamaño mínimo.
- `void worst_fit` se encarga de buscar un bloque en memoria con una diferencia en tamaño máxima con respecto al tamaño del proceso. Este puede ser del mismo tamaño que el proceso, o en caso este sea mayor, se realiza una partición dejando un espacio libre en memoria de tamaño elevado.

3. `void release` se encarga de buscar el proceso correspondiente, realizar un free a la memoria y modificar el `pid` de ese bloque de memoria para que vuelva a ser U, indicando que está libre y puede ser ocupado por otro proceso. En caso existan huecos adyacentes al bloque que acabar de ser liberado, estos espacios libres se çombinan.[en] uno solo sin la necesidad de llamar a `compact`, actualizando el arreglo `contiguos_memory` y la cantidad de bloques final.
4. `void compact` este método busca todos los huecos en la memoria y los junta en un solo bloque, de modo que la memoria libre este nuevamente junta, sin partición alguna. Para este caso se suele hacer un resize del arreglo para que en vez de contar, por ejemplo, 3 bloques de memoria libre sea 1 solo bloque grande. Hay que tener en cuenta que esta unión se da entre bloques que no necesariamente son contiguos entre sí.
5. `void status_report` esta función imprime el estado actual de la memoria. Para ello, verificamos que, en caso el pid del bloque en memoria sea distinto de P, se indica que es un espacio libre y poder imprimirlo mostrando el estado de este bloque. Pero, en caso el pid empiece con un P, significa que ese bloque de memoria está siendo ocupado por un proceso y al imprimirlo se muestra asimismo el estado del bloque correspondiente.

Se han implementado funciones auxiliares con la finalidad de hacer más práctica y sencilla nuestra implementación, además de que nuestros métodos no sean tan extensos y nuestro código sea más limpio. Estas son las siguientes:

- `copy_mem` encargado de copiar y actualizar el arreglo `contiguos_memory` luego de realizada una partición dentro del método `request` o al realizar una llamada al método `release`.
- `set_mem_values` encargado de asignar un espacio en memoria para una nueva variable de tipo `struct memory` y asignar los valores de esta.

- `update_mem` encargado de realizar una partición en la posición correcta, llamada dentro del método `request`.
- `combine_blocks` encargado de verificar si es necesario realizar una unión de bloques libres contiguos luego de realizar una llamada al método `release`, en caso existan. Si así lo fuera, realiza la unión de los bloques respectivos.
- `set_combination` encargado de asignar los nuevos valores al bloque de memoria resultante de la unión de varios bloques libres y de liberar la memoria.

La prueba mostrada en la Figura5 ejecuta los siguientes comandos:

1. Size de memoria = ./allocator 1000
2. RQ P0 400 F
3. RQ P2 200 F
4. RQ P3 100 F
5. RL P2
6. RQ P4 100 W
7. RQ P5 100 B

La prueba mostrada en la Figura6 ejecuta los siguientes comandos:

1. Size de memoria = ./allocator 5000
2. RQ P0 1200 F
3. RQ P1 2000 W
4. RQ P2 300 B
5. RQ P3 1050 F
6. RL P2
7. RQ P4 50 W

Referencias

[1] Silberchatz, A. Operating System Concepts. Chapter 9. 2018. 10th edition.

```c
void request(char process[2], int size, char typefit[1]){
    printf("Request for process: %s of size: %d with a fit of type: %s\n", process, size, typefit);
    int status;
    switch(typefit[0]){
        case 'F':
            status = first_fit(process,size);
            break;
        case 'B':
            status = best_fit(process,size);
            break;
        case 'W':
            status = worst_fit(process,size);
            break;
        default:
            printf("ERROR: Fit type not found.\n");
            return;
    }
    if(status != 1) printf("ERROR: Not enough space for allocate the process\n");
}
```

Figura 1: Método de request.

```c
void release(char process[2]){
    printf("Realeasing process: %s\n",process);
    struct memory* temp;
    for(int i = 0; i < blocks; i++){
        temp = contiguous_memory[i];
        if(temp->pid[0] == 'P'){
            if(strcmp(temp->pid,process)==0){
                struct memory *new_unused_mem = set_mem_values("U ",temp->bottom,temp->top);
                //combine_blocks(new_unused_mem);
                release_mem(new_unused_mem);
                break;
            }
        }
    }
}
```

Figura 2: Método de release.

```c
void status_report(){
    printf("Enters Status Report\n");
    struct memory* temp;
    for(int i = 0; i < blocks; i++){
        temp = contiguous_memory[i];
        if(temp->pid[0] == 'P'){
            printf("Addresses   [%d:%d] Process %s\n", temp->bottom, temp->top, temp->pid);
        } else{
            printf("Addresses   [%d:%d] Unused\n", temp->bottom, temp->top);
        }
    }
}
```

Figura 3: Método de status.

```c
void allocate_memory(int size){
    printf("The size of the memory will be: %d\n", size);
    struct memory *block = malloc(sizeof(struct memory));
    block->bottom = 0;
    block->top = size;
    block->pid[0] = 'U';
    contiguous_memory[0] = block;
    blocks = 1;
}
```

Figura 4: Método para inicializar la memoria.

```
./allocator 1000
The size of the memory will be: 1000
allocator> RQ P0 400 F
Request for process: P0 of size: 400 with a fit of type: F
allocator> RQ P2 200 F
Request for process: P2 of size: 200 with a fit of type: F
allocator> RQ P3 100 F
Request for process: P3 of size: 100 with a fit of type: F
allocator> STAT
Enters Status Report
Addresses   [0:400] Process P0
Addresses   [401:601] Process P2
Addresses   [602:702] Process P3
Addresses   [703:1000] Unused
allocator> RL P2
Realeasing process: P2
allocator> STAT
Enters Status Report
Addresses   [0:400] Process P0
Addresses   [401:601] Unused
Addresses   [602:702] Process P3
Addresses   [703:1000] Unused
allocator> RQ P4 100 W
Request for process: P4 of size: 100 with a fit of type: W
allocator> STAT
Enters Status Report
Addresses   [0:400] Process P0
Addresses   [401:601] Unused
Addresses   [602:702] Process P3
Addresses   [703:803] Process P4
Addresses   [804:1000] Unused
allocator> RQ P5 100 B
Request for process: P5 of size: 100 with a fit of type: B
allocator> STAT
Enters Status Report
Addresses   [0:400] Process P0
Addresses   [401:601] Unused
Addresses   [602:702] Process P3
Addresses   [703:803] Process P4
Addresses   [804:904] Process P5
Addresses   [905:1000] Unused
allocator>
```

Figura 5: Prueba 1 de funcionalidad de request (incluyendo los 3 métodos de fit), release y status report.

```
./allocator 5000
The size of the memory will be: 5000
allocator> RQ P0 1200 F
Request for process: P0 of size: 1200 with a fit of type: F
allocator> STAT
Enters Status Report
Addresses   [0:1200] Process P0
Addresses   [1201:5000] Unused
allocator> RQ P1 2000 W
Request for process: P1 of size: 2000 with a fit of type: W
allocator> STAT
Enters Status Report
Addresses   [0:1200] Process P0
Addresses   [1201:3201] Process P1
Addresses   [3202:5000] Unused
allocator> RQ P2 300 B
Request for process: P2 of size: 300 with a fit of type: B
allocator> STAT
Enters Status Report
Addresses   [0:1200] Process P0
Addresses   [1201:3201] Process P1
Addresses   [3202:3502] Process P2
Addresses   [3503:5000] Unused
allocator> RQ P3 1050 F
Request for process: P3 of size: 1050 with a fit of type: F
allocator> STAT
Enters Status Report
Addresses   [0:1200] Process P0
Addresses   [1201:3201] Process P1
Addresses   [3202:3502] Process P2
Addresses   [3503:4553] Process P3
Addresses   [4554:5000] Unused
allocator> RL P2
Realeasing process: P2
allocator> STAT
Enters Status Report
Addresses   [0:1200] Process P0
Addresses   [1201:3201] Process P1
Addresses   [3202:3502] Unused
Addresses   [3503:4553] Process P3
Addresses   [4554:5000] Unused
allocator> RQ P4 50 W
Request for process: P4 of size: 50 with a fit of type: W
allocator> STAT
Enters Status Report
Addresses   [0:1200] Process P0
Addresses   [1201:3201] Process P1
Addresses   [3202:3502] Unused
Addresses   [3503:4553] Process P3
Addresses   [4554:4604] Process P4
Addresses   [4605:5000] Unused
```

Figura 6: Prueba 2 de funcionalidad de request (incluyendo los 3 métodos de fit), release y status report.

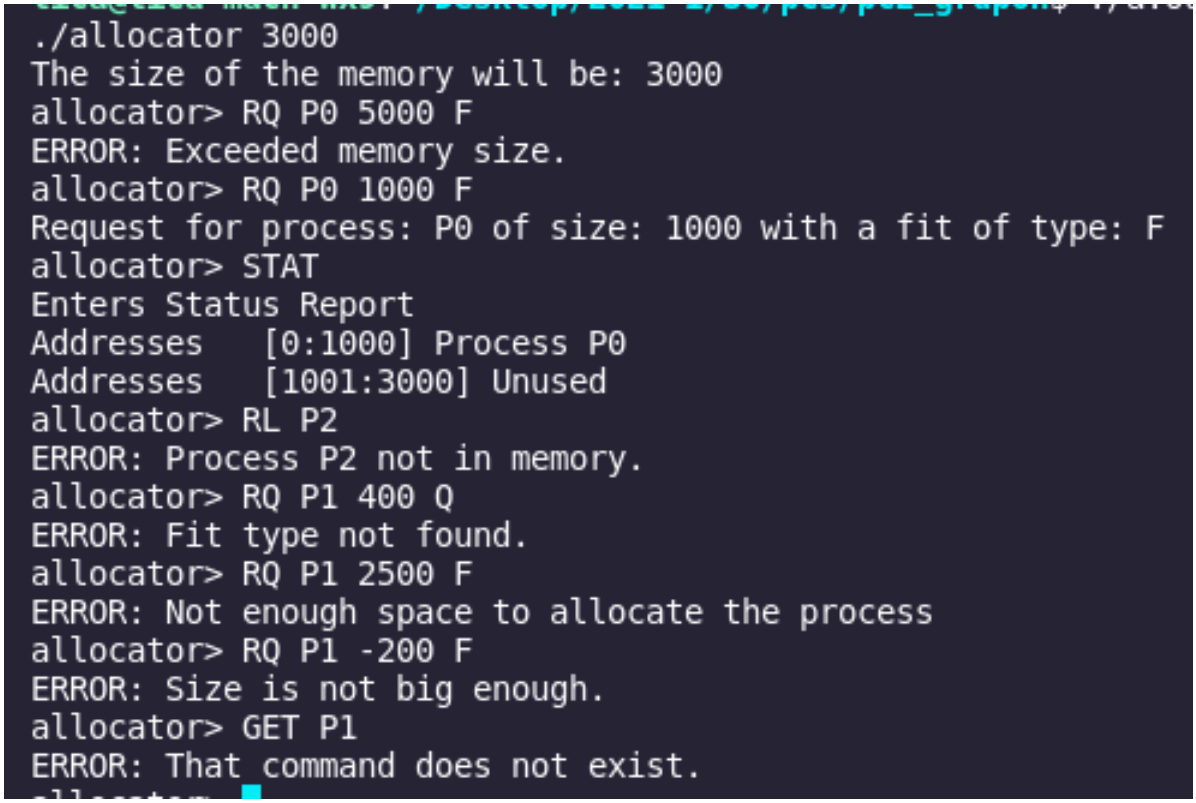

Figura 7: Prueba de errores.