TP2 - SO

Santiago Ruben Cremón

October 2025

1 Parte 1: Asignación de Memoria

Se pide en el enunciado que ejecutemos el programa `malloc.py` con las opciones: `-n 10 -H 0 -p BEST -s 0`

Es decir se realizaran diez operaciones aleatorias en un heap de 100bytes. Este proceso se lleva a cabo con una estrategia *best-fit* por lo que siempre se intentará localizar la memoria en el espacio más pequeño posible que cumpla con los requerimientos del usuario. Notemos que *coalesce* no se encontrará activado por lo que, a lo largo de la ejecución, el espacio asignado al heap comenzará a fragmentarse (externa). En caso que la memoria no alcance para realizar las localizaciones de memoria pedidas por el usuario, podrían producirse errores a la hora de realizarlas, siendo un comportamineto esperable (mencionado en OSTEP) el de devolver **NULL** cuando esto sucede en la función que se utiliza para implementar el comportamiento de **malloc**.

1.1 Best fit

En la siguiente tabla podemos ver el resultado de cada *alloc()/free()* y, a su vez, la evolución de la *Free list* a lo largo de la ejecución.

Num Operación	Operación	Resultado	Free List
1	ptr[0] = Alloc(3)	1000	[addr:1003, len:97]
2	Free(ptr[0])	0	[addr:1000, len:3][addr:1003, len:97]
3	ptr[1] = Alloc(5)	1003	[addr:1000, len:3][addr:1008,len:92]
4	Free(ptr[1])	0	[addr:1000, len:3] [addr:1003, len:5] [addr:1008, len:92]
5	ptr[2] = Alloc(8)	1008	[addr:1000, len:3] [addr:1003, len:5] [addr:1016, len:84]
6	Free(ptr[2])	0	[addr:1000, len:3] [addr:1003, len:5] [addr:1008, len:8] [addr:1016, len:84]
7	ptr[3] = Alloc(8)	1008	[addr:1000, len:3] [addr:1003, len:5] [addr:1016, len:84]
8	Free(ptr[3])	0	[addr:1000, len:3] [addr:1003, len:5] [addr:1008, len:8] [addr:1016, len:84]
9	ptr[4] = Alloc(2)	1000	[addr:1002, len:1] [addr:1003, len:5] [addr:1008, len:8] [addr:1016, len:84]
10	ptr[5] = Alloc(7)	1008	[addr:1002, len:1] [addr:1003, len:5] [addr:1015, len:1] [addr:1016, len:84]

Table 1: **Best Fit**

En este caso, al llevar una estrategia Best Fit, podemos notar que la Free List aumentará la cantidad de nodos almacenados en las primeras tres asignaciones localizando siempre desde la última dirección (que a su vez es la de mayor dimensión). Luego de liberar `ptr[2]`, y debido a que las siguientes asignaciones no superan los ocho bytes, las asignaciones de memoria se comenzarán a realizar sobre los fragmentos de memoria libres establecidos anteriormente. Notemos que, si bien las últimas dos asignaciones de memoria se realizan sobre espacios ligeramente mas grandes que el tamaño solicitado, el algoritmo utilizado determina que, luego de recorrer toda la lista, los dos mejores espacios son los que comienzan respectivamente en la dirección 1000 y 1008. Luego se procede a realizar el *split* (evitando fragmentación interna) y dejando en la free list dos direcciones de un byte de memoria.

1.2 Worst fit

En el caso de **Worst Fit** lo primero que podemos notar es que a lo largo de la ejecución siempre se procederá a dividir la última y más grande porción de memoria. De esta manera le Free List tendrá más entradas y, en particular, siguiendo este esquema, se fomentará la condición de **framentación externa** a medida que se agreguen más operaciones.

Num Operación	Operación	Resultado	Free List
1	ptr[0] = Alloc(3)	1000	[addr:1003, len:97]
2	Free(ptr[0])	0	[addr:1000, len:3][addr:1003, len:97]
3	ptr[1] = Alloc(5)	1003	[addr:1000, len:3][addr:1008,len:92]
4	Free(ptr[1])	0	[addr:1000, len:3] [addr:1003, len:5] [addr:1008, len:92]
5	ptr[2] = Alloc(8)	1008	[addr:1000, len:3] [addr:1003, len:5] [addr:1016, len:84]
6	Free(ptr[2])	0	[addr:1000, len:3] [addr:1003, len:5] [addr:1008, len:8] [addr:1016, len:84]
7	ptr[3] = Alloc(8)	1016	[addr:1000,len:3] [addr:1003,len:5] [addr:1008,len:8] [addr:1024,len:76]
8	Free(ptr[3])	0	[addr:1000, len:3] [addr:1003, len:5] [addr:1008, len:8] [addr:1016, len:8] [addr:1024, len:76]
9	ptr[4] = Alloc(2)	1024	[addr:1000, len:3] [addr:1003, len:5] [addr:1008, len:8] [addr:1016, len:8] [addr:1026, len:74]
10	ptr[5] = Alloc(7)	1026	[addr:1000, len:3] [addr:1003, len:5] [addr:1008, len:8] [addr:1016, len:8] [addr:1033, len:67]

Table 2: **Worst Fit**

1.3 First Fit

Por último, al ser la misma semilla y, por lo tanto, tener las mismas operaciones en todas las tablas. La ejecución con una estrategia de **First Fit** se comportará de igual manera que aquella con la estretegia **Best Fit**. Es necesario resaltar que este suceso es accidental al input dado en la ejecución del programa ya que las asignaciones y liberaciones de memoria estarán dadas de tal manera que ambas estrategias realizarán las mismas elecciones de fragmentos de memoria a la hora de localizar. De todas maneras, al evitar tener que recorrer toda la lista, el método First Fit será menos costoso en cantidad de operaciones.

Num Operación	Operación	Resultado	Free List
1	ptr[0] = Alloc(3)	1000	[addr:1003, len:97]
2	Free(ptr[0])	0	[addr:1000, len:3][addr:1003, len:97]
3	ptr[1] = Alloc(5)	1003	[addr:1000, len:3][addr:1008,len:92]
4	Free(ptr[1])	0	[addr:1000, len:3] [addr:1003, len:5] [addr:1008, len:92]
5	ptr[2] = Alloc(8)	1008	[addr:1000, len:3] [addr:1003, len:5] [addr:1016, len:84]
6	Free(ptr[2])	0	[addr:1000, len:3] [addr:1003, len:5] [addr:1008, len:8] [addr:1016, len:84]
7	ptr[3] = Alloc(8)	1008	[addr:1000,len:3] [addr:1003,len:5] [addr:1016,len:84]
8	Free(ptr[3])	0	[addr:1000, len:3] [addr:1003, len:5] [addr:1008, len:8] [addr:1016, len:84]
9	ptr[4] = Alloc(2)	1000	[addr:1002, len:1] [addr:1003, len:5] [addr:1008, len:8] [addr:1016, len:84]
10	ptr[5] = Alloc(7)	1008	[addr:1000, len:3] [addr:1003, len:5] [addr:1015, len:1] [addr:1016, len:8]

Table 3: **First Fit**

1.4 Asignaciones aleatorias

En el caso de aumentar la cantidad de operaciones conservando el tamaño del *heap* y una estrategia **best fit**, el comportamiento cambiará según se encuentre habilitada la opción de *coalesce*.

Luego de almacenar los resultados de las ejecuciones con y sin la opción de *coalesce* en los archivos `p1_3_c.txt` y `p1_3_sc.txt` respectivamente, podemos observar que, en el caso de no haber diseñado un *allocator* que unifique espacios de memoria libres contiguos, aumentarán en gran medida los errores de asignación de memoria. Vemos que en `p1_3_sc.txt` la cantidad de resultados de *Alloc()* cuyo valor es `-1` (valor utilizado para referenciar un error y símil a NULL como es estipulado en la implementación de OSTEP) aumenta de 29 a 178 con la misma cantidad de operaciones. A su vez, es necesario mencionar que estos errores comenzarán a suceder luego de menos operaciones de asignación, ya que en el caso de *no-coalesce* el heap comenzará a sufrir de fragmentación externa comparativamente más rápido y carecerá de la herramienta para poder solucionar este problema.

2 Parte 2: Reemplazo de Páginas

2.1 Ejercicio 1

2.1.1 Seed 0

Las siguientes tres tablas corresponden a los resultados de cada acceso y al estado de los marcos de página del programa con las opciones `-s 0 -n 10` y las estrategias FIFO, LRU y OPT.

Acceso	Resultado	Páginas a Desalojas
8	miss	8
7	miss	8 7
4	miss	8 7 4
2	miss	7 4 2
5	miss	4 2 5
4	hit	4 2 5
7	miss	2 5 7
3	miss	5 7 3
4	miss	7 3 4
5	miss	3 4 5

Table 4: Seed 0 FIFO
Final Stats - hits: 1, miss: 9, hitrate: 10.0

Acceso	Resultado	Páginas a Desalojar
8	miss	8
7	miss	8 7
4	miss	8 7 4
2	miss	7 4 2
5	miss	4 2 5
4	hit	2 5 4
7	miss	5 4 7
3	miss	4 7 3
4	hit	7 3 4
5	miss	3 4 5

Table 5: Seed 0 LRU
Final Stats - hits: 2, misses: 8; hitrate: 20.0

Acceso	Resultado	Páginas a Desalojar
8	miss	8
7	miss	8 7
4	miss	8 7 4
2	miss	2 7 4
5	miss	5 7 4
4	hit	5 7 4
7	hit	7 5 4
3	miss	3 5 4
4	hit	3 4 5
5	hit	3 4 5

Table 6: Seed 0 OPT
Final Stats - hits: 4, misses: 6; hitrate: 40.0

Notar que el orden de el cache en los ultimos tres accesos podría ser diferente dependiendo de cuales sean los próximos accesos (de los cuáles no poseemos información).

2.1.2 Seed 1

Las siguientes tres tablas corresponden con la ejecución del programa con las estrategias anteriormente mencionadas en la sección anterior. En este caso las opciones de ejecución utilizadas son `-s 1 -n 10` para cada una de las estrategias.

Acceso	Resultado	Páginas a Desalojar
1	miss	1
8	miss	1 8
7	miss	1 8 7
2	miss	8 7 2
4	miss	7 2 4
4	hit	7 2 4
6	miss	2 4 6
7	miss	4 6 7
0	miss	6 7 0
0	hit	6 7 0

Table 7: Seed 1 FIFO
Final Stats - hits: 2, misses: 8; hitrate: 20.0

Acceso	Resultado	Páginas a Desalojar
1	miss	1
8	miss	1 8
7	miss	1 8 7
2	miss	8 7 2
4	miss	7 2 4
4	hit	7 2 4
6	miss	2 4 6
7	miss	4 6 7
0	miss	6 7 0
0	hit	6 7 0

Table 8: Seed 1 LRU
Final Stats - hits: 2, misses: 8; hitrate: 20.0

Acceso	Resultado	Páginas a desalojar
1	miss	1
8	miss	1 8
7	miss	1 8 7
2	miss	8 2 7
4	miss	2 7 4
4	hit	2 4 7
6	miss	4 6 7
7	hit	4 6 7
0	miss	6 7 0
0	hit	6 7 0

Table 9: Seed 1 OPT
Final Stats - hits: 3, misses: 7; hitrate: 30.0

2.1.3 Seed 2

Por último, podemos ver el resultado de repetir los pasos anteriores con las opciones de ejecucion del programa `-s 2 -n 10`.

Acceso	Resultado	Páginas a Desalojar
9	miss	9
9	hit	9
0	miss	9 0
0	hit	9 0
8	miss	9 0 8
7	miss	0 8 7
6	miss	8 7 6
3	miss	7 6 3
6	hit	7 6 3
6	hit	7 6 3

Table 10: Seed 2 FIFO
Final Stats - hits: 4, misses: 6; hitrate: 40.0

Acceso	Resultado	Páginas a Desalojar
9	miss	9
9	hit	9
0	miss	9 0
0	hit	9 0
8	miss	9 0 8
7	miss	0 8 7
6	miss	8 7 6
3	miss	7 6 3
6	hit	7 3 6
6	hit	7 3 6

Table 11: Seed 2 LRU
Final Stats - hits: 4, misses: 6; hitrate: 40.0

Acceso	Resultado	Páginas a Desalojar
9	miss	9
9	hit	9
0	miss	9 0
0	hit	9 0
8	miss	9 0 8
7	miss	0 8 7
6	miss	8 7 6
3	miss	7 3 6
6	hit	7 3 6
6	hit	7 3 6

Table 12: Seed 2 OPT
Final Stats - hits: 4, misses: 6; hitrate: 40.0

2.2 Ejercicio 2

Procedemos a generar una secuencia de accesos que tengan el peor rendimiento para las estrategias FIFO y LRU respectivamente. Luego compararemos el *hitrate* con aquel de la ejecución de estrategia OPT para poder así evaluar si un aumento del tamaño del **cache** es capaz de acercar el rendimiento de los dos primeros a este último.

Los resultados para FIFO y LRU con sus respectivos resultados en OPT y la re-ejecución con un mayor cache se encuentran almacenados en los archivos de la carpeta `p2/ej2`.

2.2.1 Peor caso para FIFO

Al siempre desalojar del cache a la primera página en ser referenciada, es fácil incurrir en una secuencia de accesos que pueda generar un *worst case scenario* en el cual todos los accesos generan *cache misses*. En este caso, y teniendo en cuenta cierto principio de localidad, podemos pensar en un bucle de accesos a seis espacios de memoria distintos en un orden establecido (como podemos ver en la primera seccion de `ej2_fifo.txt`). Al sólo tener disponible cinco marcos de página cada espacio será guardado en la cache por una cantidad de tiempo insuficiente, ya que cuando sea necesario el próximo acceso a ese mismo espacio de memoria, esta ya habrá sido desalojada y, por lo tanto, será necesario recuperarla desde el disco nuevamente. Esto provocará que, efectivamente, el *hitrate* en el cache sea **0%**.

Podemos ver que este valor se aleja de aquel *hitrate* de **55.56%** obtenido con una estrategia OPT.

Para esta secuencia anteriormente mencionada tan sólo será necesario agregar un marco de página extra a los cinco preexistentes. De esta manera, todas las páginas accedidas en la secuencia podrán ser alojadas y accedidas directamente desde el cache provocando un aumento del *hitrate* a **66.67%**, un valor hasta superior al obtenido por OPT teniendo en cuenta que el porcentaje de *misses* restante se debe a aquellos que son mandatorios al alojar una página por primera vez en memoria cache.

2.2.2 Peor caso para LRU

En este caso podemos usar exactamente la misma secuencia de accesos que aquella utilizada para el peor caso de FIFO en el punto anterior. Notemos que aquí el problema radica en que la página que se considera *LRU* será a su vez la primera en haber sido accedida en la secuencia y así sucesivamente. Por este mismo motivo la traza de accesos es idéntica a la generada al ejecutar la secuencia con la estrategia FIFO en el punto anterior, lo cual provoca que el *hitrate* sea, a su vez, de **55.56%**.

Como estamos utilizando exactamente la misma secuencia de accesos que el caso anterior, podemos hacer uso de el mismo resultado con OPT. Luego si proponemos una expansión a seis marcos de página con LRU notaremos que la mejora, también, será idéntica a la anterior.

2.3 Ejercicio 3

Utilizaremos una traza generada con los valores del 1 a 6 para representar dos localidades distintas. En un principio se realizaran accesos entre los tres primeros, para luego pasar a los tres restantes y, por último, realizar accesos aleatorios entre todos. La idea es analizar que sucede cuando dos localidades distintas (que podrían representar rudimentariamente a dos programas) son cargadas en memoria y luego se tiene que alternar los accesos entre ellas.

Podemos ver que sucede con los accesos utilizando una política LRU con la siguiente tabla:

Acceso	**Resultado**	**Cache**
1	miss	1
2	miss	1 2
3	miss	1 2 3
2	hit	1 3 2
3	hit	1 2 3
1	hit	2 3 1
1	hit	2 3 1
1	hit	2 3 1
3	hit	2 1 3
4	miss	1 3 4
6	miss	3 4 6
5	miss	4 6 5
4	hit	6 5 4
4	hit	6 5 4
5	hit	6 4 5
6	hit	4 5 6
6	hit	4 5 6
4	hit	5 6 4
2	miss	6 4 2
1	miss	4 2 1
4	hit	2 1 4
3	miss	1 4 3
3	hit	1 4 3
5	miss	4 3 5

Table 13: LRU localidades

Final Stats - hits: 14, misses: 10, hitrate: 58.33

Acceso	Resultado	Cache
1	miss	1
2	miss	1 2
3	miss	1 2 3
2	hit	1 2 3
3	hit	1 2 3
1	hit	1 2 3
1	hit	1 2 3
1	hit	1 2 3
3	hit	1 2 3
4	miss	1 2 4
6	miss	2 4 6
5	miss	4 6 5
4	hit	4 6 5
4	hit	4 6 5
5	hit	4 6 5
6	hit	4 6 5
6	hit	4 6 5
4	hit	4 6 5
2	miss	4 5 2
1	miss	4 5 1
4	miss	4 5 1
3	miss	4 5 3
3	hit	4 5 3
5	hit	4 5 3

Table 14: OPT localidades

Final Stats - hits: 15, misses: 9, hitrate: 62,50

Acceso	Resultado	Cache
1	miss	1
2	miss	1 2
3	miss	1 2 3
2	hit	1 2s 3
3	hit	1 2s 3s
1	hit	1s 2s 3s
1	hit	1s 2s 3s
1	hit	1s 2s 3s
3	hit	1s 2s 3s
4	miss	2 3 4
6	miss	3 4 6
5	miss	4 6 5
4	hit	4s 6 5
4	hit	4s 6 5
5	hit	4s 6 5s
6	hit	4s 6s 5s
6	hit	4s 6s 5s
4	hit	4s 6s 5s
2	miss	6 5 2
1	miss	5 2 1
4	miss	2 1 4
3	miss	1 4 3
3	hit	1 4 3
5	miss	4 3 5

Table 15: SC localidades
Final Stats: hits: 13, misses: 14, hitrate: 54,17

Podemos ver que en este caso **LRU** es ligeramente superior en *hitrate* a una política **SC**. Esto se debe a que, al realizar accesos entre la misma secuencia de páginas, se termina perdiendo el beneficio de la segunda oportunidad al terminar todas marcadas para no ser desalojadas inmediatamente (notado con una *s* en este caso). De esta manera, la política **SC** se ve reducida en práctica a una política **FIFO** poco eficiente en tiempo ya que al estar todos los frames marcados para *second chance* el algoritmo pasará uno por uno desmarcando la flag correspondiente y volverá para desalojar a la primera página que había encontrado en la lista.