Trabajo Práctico 2: Memoria

Sistemas Operativos

Maria Valentina Sancho
LU: 1214/ 23
Mail: vvalentinabelmonte@gmail.com

Primera parte: Asignación de memoria

1) Se ejecutó el simulador de Asignación de Memoria con la siguiente configuración:

python3 malloc.py -n 10 -H 0 -p BEST -s 0

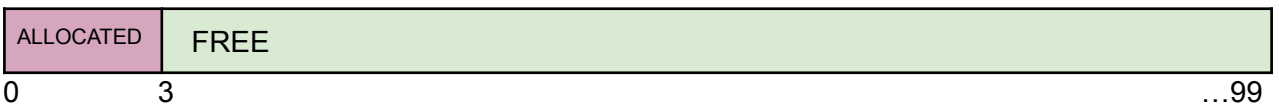
Genera 10 operaciones aleatorias con un header de tamaño 0, con la política BEST FIT.
Esta política encuentra el bloque más chico de memoria que alcance para satisfacer el pedido.
Por defecto el tamaño del heap es de 100 bytes.
El heap comienza en la dirección 1000 por default.
Cuando se pide un bloque de memoria de un tamaño x, si el alloc es exitoso, devuelve un puntero a un bloque de **ese tamaño exacto**. En este simulador no hay fragmentación interna.

En los gráficos se representa a el heap como un bloque de 100 bytes. Los índices ubicados debajo del heap indican el número de byte dentro del mismo.

Resultado de cada alloc()/ free() :

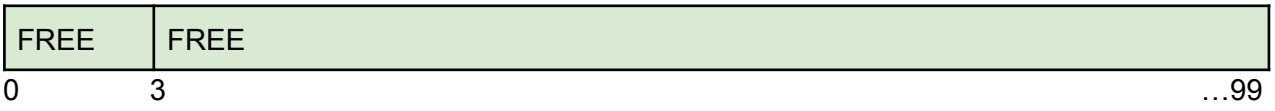
La primera operación es un alloc de 3 bytes. El espacio libre es de 100 bytes, así que el bloque se fragmenta en dos partes: una de 3 bytes y otra de 97 bytes.
Esto es debido al **Splitting:** el tamaño del pedido es menor al tamaño de la memoria disponible, entonces se divide el bloque en dos partes: la primer parte es del tamaño exacto al pedido y la segunda del tamaño del tamaño del bloque - tamaño del pedido. Se retorna la primer parte siempre.

El siguiente gráfico muestra el heap luego de realizar el alloc:



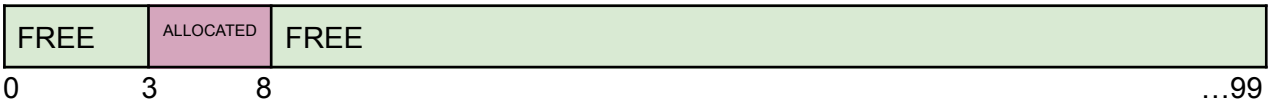
El bloque que va desde el comienzo hasta el byte número 3 (sin incluir) pasa a estar ALLOCATED.

Luego se hace un free de esa memoria, por lo que el heap quedaría así:

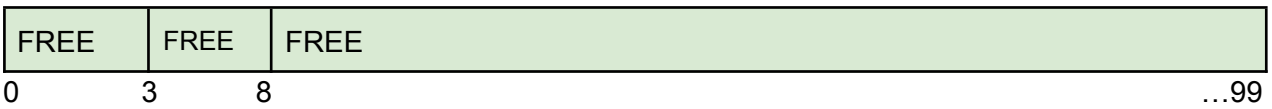


El coalescing está desactivado, por ende, la memoria queda fragmentada. No se mergean los bloques.

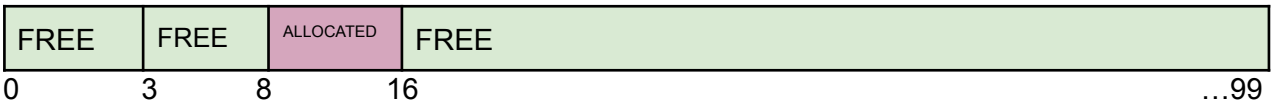
La siguiente operación es un alloc de 5 bytes. El primer bloque de 3 bytes no satisface el pedido. Se utiliza la siguiente porción más chica de memoria para satisfacer el pedido, es decir, se utiliza el bloque de 97 bytes, que se fragmenta en dos partes: uno de 5 bytes y uno de 92 bytes.



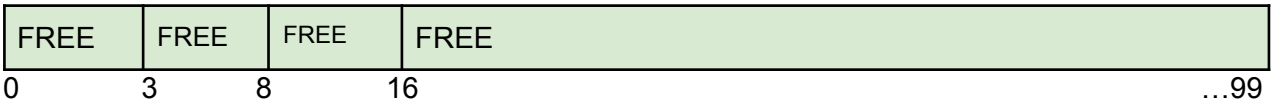
Luego se libera ese bloque, quedando tres bloques libres de memoria:



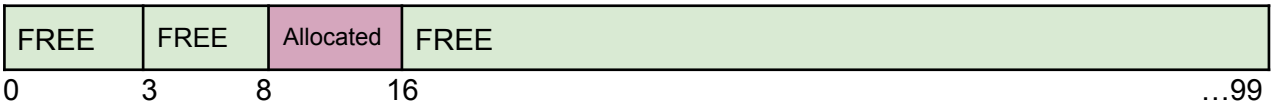
Luego se realiza un alloc de 8 bytes, y se busca el fragmento más chico que alcance para satisfacer el pedido. Se toma el bloque de 92 bytes y se lo particiona en dos bloques: uno de 8 bytes y otro de 84 bytes. Cuando se realiza el alloc y es exitoso se retorna el puntero al comienzo del bloque asignado.



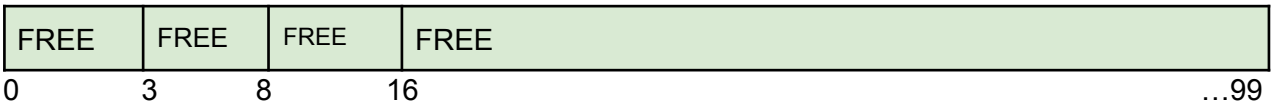
Se libera ese espacio:



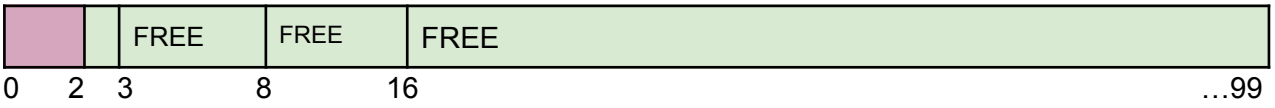
Se realiza un alloc de 8 bytes nuevamente. Se utiliza el bloque que se liberó en la operación anterior de 8 bytes, porque es el bloque más chico de memoria que satisface el pedido.



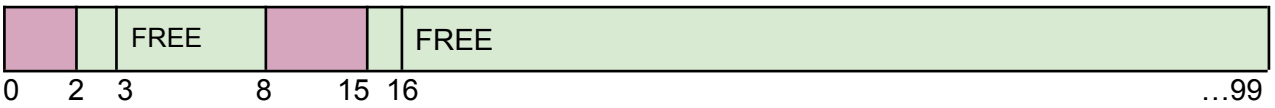
Y se vuelve a liberar:



Luego, se realiza un alloc de 2 bytes.
Como hay un espacio free en el primer bloque de 3 bytes, se aloja ahí. El primer bloque de 3 bytes es fragmentado en uno de 2 bytes y otro de 1 byte.
ptr[4] apunta a ese bloque asignado (el que está en color rosa).



Se realiza un alloc de 7 bytes. En el tercer bloque libre hay una porción de 8 bytes, que es suficiente. Como se sigue la política BEST FIT, se elige ese bloque.



La porción de 8 bytes fue fragmentada en dos: una de 7 bytes y otra de 1 byte.

b) La evolución de la **free list** durante la ejecución es la siguiente:

Recordemos que la free list es una lista que almacena información sobre los bloques de memoria disponibles.

La free list al principio se vé así:

[(addr: 1000, size = 100)]
El heap es un bloque no fragmentado de 100 bytes con la dirección base 1000.

Se realiza un alloc de 3 bytes:
[(addr: 1003, size = 97)]
Desde la address 1000 a la 1003 (sin incluir) la memoria está allocated.

Se realiza el free de ese espacio y queda la memoria fragmentada porque el coalescing está desactivado:
[(addr:1000, size = 3), (addr: 1003, size = 97)]

Se realiza un alloc de 5 bytes. Este bloque asignado comienza en la address 1003 y termina en la 1008, con un tamaño de 5 bytes.
[(addr:0, size = 1003), (addr: 1008, size = 92)]

Se libera el bloque:
[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1008, size = 92)]

Se realiza un alloc de 8 bytes. Ese nuevo bloque asignado comienza en la dirección 1008 y va hasta la 1016.
[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1016, size = 84)]

Se libera:
[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1008, size = 8),(addr:1016, size = 84)]

Se realiza un nuevo alloc de 8 bytes:
Se utiliza el bloque liberado en la anterior operación porque es el primer bloque de memoria más chico que satisface el pedido.

[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1016, size = 84)]

Se libera:
[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1008, size = 8),(addr:1016, size = 84)]

Se realiza un alloc de 2 bytes y se aprovecha el primer bloque fragmentado, porque es el bloque más chico que alcanza para satisfacer el pedido.
Se particiona el bloque de 3 bytes en uno de 2 bytes y otro de 1 byte. El bloque asignado comienza en la dirección 1000 y ocupa dos bytes.

[(addr:1002, size = 1), (addr: 1003, size = 5),(addr:1008, size = 8),(addr:1016, size= 84)]

Se realiza un alloc de 7 bytes, como hay un bloque de 8 bytes libre, se particiona en dos bloques: uno de 1 byte y otro de 7 bytes.

[(addr:1002, size = 1), (addr: 1003, size = 5),(addr:1015, size = 1),(addr:1016, size =84)]

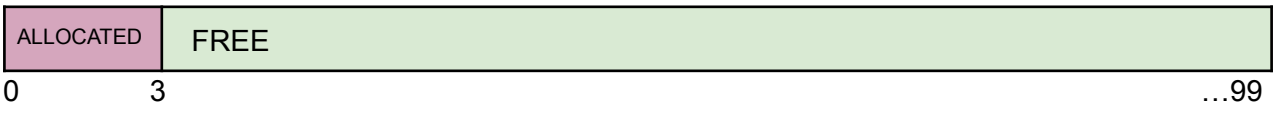
c) A lo largo del tiempo la free list gana un elemento cada vez que se realiza un alloc y no hay un fragmento exacto para satisfacer el pedido. Esto es porque debe particionarse el fragmento de memoria para quedarnos con una parte que sea del **tamaño exacto** al pedido, a raíz de esto la memoria es fragmentada quedando un bloque en la free list (el que tiene tamaño = tamaño del bloque - tamaño del pedido) y el otro que pasa a estar ALLOCATED, que es el del que se retorna el puntero.

Cuando se libera un fragmento ALLOCATED la free list gana un nuevo elemento.
Cuando hay particiones que son útiles al realizar un alloc, en particular si existe una partición que sea del mismo tamaño que el alloc que se realiza, la free list no crece porque no se fragmenta ese bloque, sino que pierde un elemento.
Al estar el coalescing desactivado, una vez que un bloque es fragmentado, cada partición representa un potencial nuevo elemento de la free list.

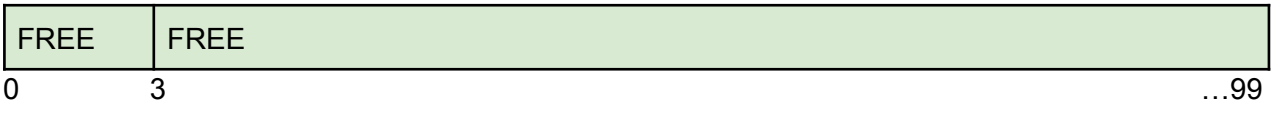
2) Análisis del caso con las políticas Worst Fit y First Fit:

Con la política de WORST FIRST:

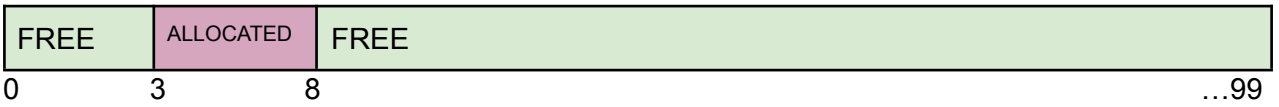
Se realiza un alloc de 3 bytes, se elige la porción más grande de memoria. El bloque de 100 bytes es fragmentado en dos: uno de 3 bytes y otro de 97.



Luego se hace un free de esa memoria:

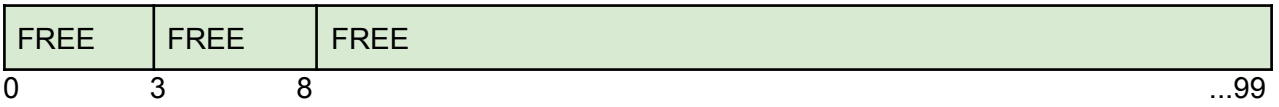


Como antes, la memoria queda fragmentada.
Se piden 5 bytes. Se fragmenta el bloque de 97 bytes (el más grande) en dos: uno de 5 bytes y otro de 92 bytes.

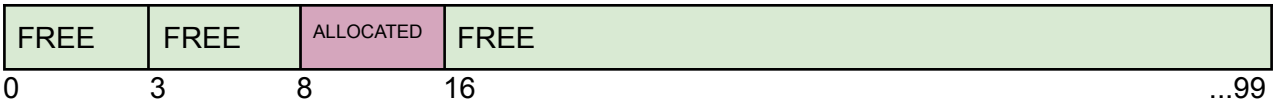


Hasta este punto, el resultado no es diferente al que tuvimos con BEST FIT.
Es importante notar que con la política BEST FIT el bloque más grande es utilizado para la última operación porque el de 3 bytes no es suficiente, con la política WORST FIT aunque fuera suficiente se elegiría el bloque más grande igual. Los resultados son los mismos pero no por la misma causa.

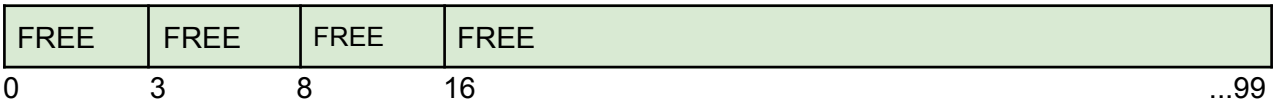
Luego se libera el bloque:



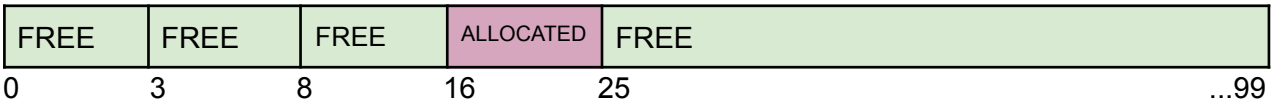
Se piden 8 bytes de memoria, utilizamos el bloque de 92 bytes, que es el más grande. Este se particiona en dos bloques de 8 bytes y 84 bytes respectivamente.
Ahora ptr[2] apunta a ese bloque.



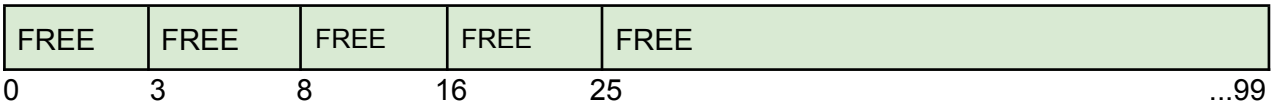
Se libera ese espacio de memoria:



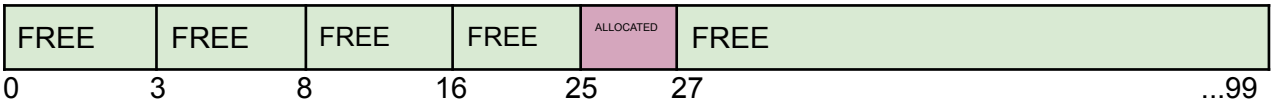
Se realiza un alloc de 8 bytes nuevamente. Como la política es WORST FIT, no se reutiliza el bloque de 8 bytes que fue liberado recientemente. Se utiliza el siguiente fragmento más grande, que es el de 84 bytes.
Ese bloque es particionado en uno de 8 bytes y otro de 76 bytes.
ptr[3] apunta a ese bloque de memoria.



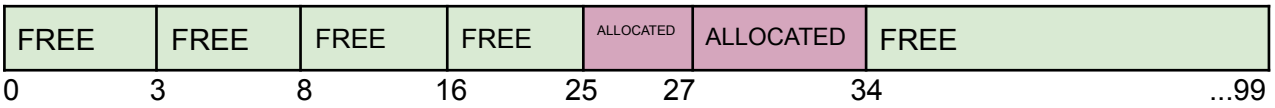
Y se libera esa memoria:



Luego, se realiza un alloc de 2 bytes. Nuevamente, por la política de WORST FIT, utilizamos el bloque más grande que es de 76 bytes, y queda fragmentado en uno de 2 bytes y otro de 74 bytes.



Se realiza un alloc de 7 bytes. Se elige el bloque de 74 bytes y queda fragmentado en dos bloques: uno de 7 bytes y otro de 67 bytes.



La política Worst Fit en este caso (con el coalescing desactivado) favorece la fragmentación externa de la memoria. El problema es que, como hay que elegir el bloque más grande que cumpla con el pedido, cada alloc genera que la memoria se fragmente, lo que resulta en muchos pequeños fragmentos de memoria libres que no son mergeados nunca porque el coalescing no está activado. Si hubiera un pedido grande de memoria no se podría satisfacer por esto mismo. El mayor espacio de memoria es aquel que nunca fue asignado, porque no está fragmentado.

La Free List utilizando esta política va evolucionando de la siguiente forma:

Inicio: [(addr: 1000, size = 100)]

Se realiza un alloc de 3 bytes:
[(addr: 1003, size = 97)]

Se realiza el free de ese espacio:
[(addr:1000, size = 3), (addr: 1003, size = 97)]

Se realiza un alloc de 5 bytes:
[(addr:1000, size = 3), (addr: 1008, size = 92)]

Se realiza un free de ese espacio:
[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1008, size = 92)]

Se realiza un alloc de 8 bytes, que va de la address 1008 a la 1016.

[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1016, size = 84)]

Se libera:
[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1008 ,size= 8),(addr:1016, size = 84)]

Se realiza un alloc de 8 bytes. Se utiliza el bloque de 84 bytes y es fragmentado en uno de 8 y otro de 76 bytes.
El bloque asignado va desde la address 1016 a la 1024.

[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1008 ,size= 8),(addr:1024, size = 76)]

Se libera:
[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1008 ,size= 8),(addr:1016, size = 8), (addr:1024, size = 76)]

Se realiza un alloc de 2 bytes. Se fragmenta el bloque de 76 bytes (por ser el más grande) en uno de 2 bytes y otro de 74.
El bloque asignado comienza en la dirección 1024 y va hasta la 1026.
[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1008 ,size= 8),(addr:1016, size = 8), (addr:1026, size = 74)]

Se realiza un alloc de 7 bytes. Se elige el bloque más grande que es de 74 bytes y es fragmentado en uno de 7 bytes y otro de 67.

[(addr:1000, size = 3), (addr: 1003, size = 5),(addr:1008 ,size= 8),(addr:1016, size = 8), (addr:1033, size = 67)]

La free list aumenta cada vez que se realiza un alloc porque no se reutilizan los espacios libres fragmentados. Siempre se fragmenta el mayor espacio para satisfacer el pedido de memoria.

Análisis con política FIRST FIT:

En este caso particular, el resultado es el mismo que utilizando la política BEST FIT. Si vemos el avance de la free list y el uso de la memoria con BEST FIT, justo para este caso el bloque de memoria más chico lo suficientemente grande para cumplir con el pedido (best fit) coincide con el primer bloque de memoria lo suficientemente grande para satisfacer el pedido (first fit).

Esto es porque hay pocas fragmentaciones: no hay muchas opciones para elegir.

Como se elige en cada paso el mismo bloque, en cada paso la fragmentación es la misma y por ende el bloque de memoria quedará asignado de la misma forma en ambas políticas. La free list será igual.

3) Se ejecutó el simulador con el siguiente comando:

python3 malloc.py -n 1000 -H 0 -p BEST -s 0 -c

Nota: Es bastante útil redireccionar la salida estándar a un archivo para poder ver mejor el resultado, ya que el número de operaciones es muy grande.

Por ejemplo: python3 malloc.py -n 1000 -H 0 -p BEST -s 0 -c > resultado.txt

El comando -c detalla la ejecución del simulador. Utilicé el hecho de que los allocs exitosos retornan la dirección base del bloque que se asignó mientras que los que fallan retornan -1 para saber la cantidad de pedidos de memoria que fallaron.

Las asignaciones grandes a lo largo del tiempo fallan. Lo que ocurre es que, utilizando la política BEST y con coalescing desactivado, no se pueden satisfacer esos pedidos de memoria, porque la memoria ya está fragmentada en su totalidad por los pedidos anteriores. Hay bloques que están free, pero como el coalescing está desactivado esos bloques no se vuelven a unir y no forman bloques que satisfagan pedidos grandes de memoria. **Hay fragmentación externa.**

De 1000 operaciones, 591 son allocs y hay 177 que fallan. Comienza a fallar desde el alloc número 53 intermitentemente (ya que depende del tamaño del pedido, y de qué bloques se liberan durante la ejecución), porque la memoria en ese punto de la ejecución está fragmentada en partes que no pueden satisfacer los pedidos. El alloc número 53 en particular es un pedido de 5 bytes, no es grande, pero el bloque más grande en la free list es de 4 bytes.
La free list en ese punto cuenta con 18 elementos, y la mayoría de bloques son de 1 byte.

Se ejecuta nuevamente el simulador con el siguiente comando, que activa el coalescing:

python3 malloc.py -n 1000 -H 0 -p BEST -s 0 -C -c

Cuando el coalescing está activado el resultado cambia: aunque siguen habiendo asignaciones que no pudieron ser realizadas, a medida que se liberan bloques y estos bloques son vecinos, se mergean y esto permite que nuevos pedidos de memoria grandes puedan realizarse. Solo va a depender de que tanto espacio se libera y que tanto espacio libre tiene alrededor un bloque.

La diferencia entre los dos casos es que sin coalescing las asignaciones grandes no pueden realizarse (en particular, si se quiere realizar una asignación de un tamaño **mayor a todos los pedidos anteriores**), y con coalescing la situación depende de los bloques liberados a lo largo de las operaciones, pero es posible poder asignar nuevos bloques. Con coalescing la fragmentación externa se reduce a medida de que se liberan bloques vecinos.

De las 1000 operaciones, 591 son allocs y 29 fallaron. Hay una mejora significativa de la fragmentación externa debido al coalescing y por eso esta configuración tuvo menos allocs que fallaron. El primer alloc que falla es el número 93, en ese punto de la ejecución la freelist tiene 3 elementos y el bloque más grande tiene 5 bytes, mientras que el pedido es de 7 bytes.

La free list en este caso tiene menos elementos que con el coalescing desactivado, pero tiene bloques de mayor tamaño que los que están en la free list en la ejecución sin coalescing.

Segunda parte: Reemplazo de páginas

1) Se ejecutó el simulador de Reemplazo de Páginas con diferentes políticas.

Se utilizó la siguiente configuración:

python3 paging-policy.py -s 0 -n 10

Este fue el resultado de la operación:
Access: 8 Hit/Miss? State of Memory?
Access: 7 Hit/Miss? State of Memory?
Access: 4 Hit/Miss? State of Memory?
Access: 2 Hit/Miss? State of Memory?
Access: 5 Hit/Miss? State of Memory?
Access: 4 Hit/Miss? State of Memory?
Access: 7 Hit/Miss? State of Memory?
Access: 3 Hit/Miss? State of Memory?
Access: 4 Hit/Miss? State of Memory?
Access: 5 Hit/Miss? State of Memory?

Genera 10 referencias aleatorias.
Por default la política de reemplazo de páginas es FIFO (First In First Out).
Por default la caché tiene 3 marcos de página.
La caché al principio de la ejecución está vacía.

PÁGINA PEDIDA	FRAME 1	FRAME 2	FRAME 3	LISTA DE PÁGINAS A DESALOJAR
8	8			8
7	8	7		8 7
4	8	7	4	8 7 4
2	2	7	4	7 4 2
5	2	5	4	4 2 5
4	2	5	4	4 2 5
7	2	5	7	2 5 7
3	3	5	7	5 7 3
4	3	4	7	7 3 4

5	3	4	5	3 4 5

Los pedidos de página en **rojo** indican que hubo un **MISS**. Al pedir la página se ocasionó un PAGE FAULT y se trajo esa página a un frame.
Los pedidos de página en **verde** representan un **HIT**. La página ya estaba en memoria al solicitarla.
Cuando se reemplaza una página por otra, la nueva página pasa a estar en **el frame que ocupaba la que fue desalojada** . Se quita la página reemplazada de la lista de páginas a desalojar y se agrega al final de la lista la página traída.
Cada vez que una página es traída del disco se coloca al final de la lista de páginas a desalojar.

El page fault rate, que es igual al número de MISSES sobre el número de referencias totales es de: 9/10.

Se ejecutó el simulador con la política LRU (Last Recently Used). Esta fue la configuración utilizada:

python3 paging-policy.py -s 0 -n 10 -p LRU

Y el resultado fue el siguiente:

PÁGINA PEDIDA	FRAME 1	FRAME 2	FRAME 3	LISTA DE PÁGINAS A DESALOJAR
8	8			8
7	8	7		8 7
4	8	7	4	8 7 4
2	2	7	4	7 4 2
5	2	5	4	4 2 5
4	2	5	4	2 5 4
7	7	5	4	5 4 7
3	7	3	4	4 7 3
4	7	3	4	7 3 4
5	5	3	4	3 4 5

Cuando una página es accedida o pedida pasa al final de la lista de páginas a desalojar.
Esta política tuvo un HIT de ventaja sobre la política FIFO.

El page fault rate es de 8/10.

La política de reemplazo de páginas óptimo consta de reemplazar la página que se va a usar más lejos en el futuro.
Se ejecuta el simulador con la siguiente configuración:

python3 paging-policy.py -s 0 -n 10 -p OPT

Que resulta en:

PÁGINA PEDIDA	FRAME 1	FRAME 2	FRAME 3	LISTA DE PÁGINAS A DESALOJAR
8	8			8
7	8	7		8 7
4	8	7	4	8 7 4
2	2	7	4	2 7 4
5	5	7	4	5 7 4
4	5	7	4	5 4 7
7	5	7	4	7 5 4
3	5	3	4	3 5 4
4	5	3	4	3 4 5
5	5	3	4	3 4 5

El page fault rate es de: 6/10.

Aclaración: En el simulador por default el algoritmo de ordenación de la lista de páginas a reemplazar es ADDRSORT. Cada vez que se debe reemplazar una página utilizando la política ÓPTIMA, se calcula dinámicamente la que se usará más lejos en el futuro en esa lista y esa es la elegida. No se sigue el orden de la lista.
Yo elegí mostrar la lista de otra forma en el cuadro para que se entienda mejor durante la ejecución cual es la próxima página a reemplazar. Cada vez que se realiza una operación la lista se actualiza dinámicamente y se coloca en orden de uso más lejano a menos lejano en el futuro. Se elige siempre reemplazar la página que está primera en esa lista.

Ambas formas van a elegir en cada paso la misma página a reemplazar y la lista de páginas tiene los mismos elementos, solo están ordenadas diferente.

Se ejecutó el simulador nuevamente con la siguiente configuración:

python3 paging-policy.py -s 1 -n 10

Bajo las mismas condiciones de antes: La caché tiene 3 frames y la política de reemplazo de páginas es FIFO.

Este fue el resultado:
Access: 1 Hit/Miss? State of Memory?
Access: 8 Hit/Miss? State of Memory?
Access: 7 Hit/Miss? State of Memory?
Access: 2 Hit/Miss? State of Memory?
Access: 4 Hit/Miss? State of Memory?
Access: 4 Hit/Miss? State of Memory?
Access: 6 Hit/Miss? State of Memory?
Access: 7 Hit/Miss? State of Memory?
Access: 0 Hit/Miss? State of Memory?
Access: 0 Hit/Miss? State of Memory?

PÁGINA PEDIDA	FRAME1	FRAME2	FRAME3	LISTA DE PÁGINAS A DESALOJAR
1	1			1
8	1	8		1 8
7	1	8	7	1 8 7
2	2	8	7	8 7 2
4	2	4	7	7 2 4
4	2	4	7	7 2 4
6	2	4	6	2 4 6
7	7	4	6	4 6 7
0	7	0	6	6 7 0
0	7	0	6	6 7 0

De 10 accesos, 2 generaron HIT y el resto son MISSES.
El page fault rate es de 8/10.

Con la política LRU el resultado es el siguiente:

PÁGINA PEDIDA	FRAME1	FRAME2	FRAME3	LISTA DE PÁGINAS A DESALOJAR
1	1			1
8	1	8		1 8
7	1	8	7	1 8 7

2	2	8	7	8 7 2
4	2	4	7	7 2 4
4	2	4	7	7 2 4
6	2	4	6	2 4 6
7	7	4	6	4 6 7
0	7	0	6	6 7 0
0	7	0	6	6 7 0

No hay diferencias con el resultado utilizando la política FIFO. Esto es porque la lista de páginas a desalojar ordenada por último acceso y por la antigüedad que fue traída a memoria es la misma.

Con la política óptima este es el resultado:

PÁGINA PEDIDA	FRAME1	FRAME2	FRAME3	LISTA DE PÁGINAS A DESALOJAR
1	1			1
8	1	8		1 8
7	1	8	7	1 8 7
2	2	8	7	8 2 7
4	2	4	7	2 7 4
4	2	4	7	2 4 7
6	6	4	7	4 6 7
7	6	4	7	4 6 7
0	6	0	7	6 7 0
0	6	0	7	6 7 0

Los misses se generan sólo cuando la página nunca fue cargada en la memoria. Una vez que fue cargada, siempre que se accede es un HIT.
La page fault rate con esta política es de 7/10.

Vale la misma aclaración que en el caso de simulación anterior acerca del orden de la lista de páginas a desalojar.

Se ejecuta el simulador con la siguiente configuración:

python3 paging-policy.py -s 2 -n 10

Teniendo el siguiente resultado con la política FIFO:

Access: 9 Hit/Miss? State of Memory?
Access: 9 Hit/Miss? State of Memory?
Access: 0 Hit/Miss? State of Memory?
Access: 0 Hit/Miss? State of Memory?
Access: 8 Hit/Miss? State of Memory?
Access: 7 Hit/Miss? State of Memory?
Access: 6 Hit/Miss? State of Memory?
Access: 3 Hit/Miss? State of Memory?
Access: 6 Hit/Miss? State of Memory?
Access: 6 Hit/Miss? State of Memory?

PÁGINA PEDIDA	FRAME1	FRAME2	FRAME3	LISTA DE PÁGINAS A DESALOJAR
9	9			9
9	9			9
0	9	0		9 0
0	9	0		9 0
8	9	0	8	9 0 8
7	7	0	8	0 8 7
6	7	6	8	8 7 6
3	7	6	3	7 6 3
6	7	6	3	7 6 3
6	7	6	3	7 6 3

En esta generación aleatoria, los MISSES ocurren solo cuando la página no está en memoria por primera vez. Los pedidos/accesos a página ocurren poco tiempo después de haber traído la página a memoria, por ende no hubo suficientes operaciones para desalojar la página luego de haber sido pedida, y esto ocasiona un HIT.

Este es el resultado con la política Least Recently Used:

PÁGINA PEDIDA	FRAME1	FRAME2	FRAME3	LISTA DE PÁGINAS A DESALOJAR
9	9			9
9	9			9
0	9	0		9 0
0	9	0		9 0
8	9	0	8	9 0 8
7	7	0	8	0 8 7
6	7	6	8	8 7 6
3	7	6	3	7 6 3
6	7	6	3	7 3 6
6	7	6	3	7 3 6

El resultado es el mismo que en el resultado utilizando la política FIFO. Lo único que cambia es el orden de la lista de páginas a desalojar. Esto es debido a que la mayoría de páginas pedidas luego no son accedidas frecuentemente, por ende, con la política LRU la primer página a desalojar tiende a ser la que entró primero porque fue la última vez donde fue accedida.

Para ambas políticas la page fault rate es de 6/10.
Con la política óptima este es el resultado:

PÁGINA PEDIDA	FRAME1	FRAME2	FRAME3	LISTA DE PÁGINAS A DESALOJAR
9	9			9
9	9			9
0	9	0		9 0
0	9	0		9 0
8	9	0	8	9 0 8
7	7	0	8	0 8 7
6	7	6	8	8 7 6
3	7	6	3	7 3 6
6	7	6	3	7 3 6
6	7	6	3	7 3 6

El page fault rate para esta política se mantiene: es de 6/10.
Para las 3 políticas el resultado es el mismo: se generan la mayor cantidad de HITS posibles para esta secuencia. Cada vez que una página es cargada en memoria, si es accedida, es un HIT. Los únicos MISSES ocurrieron al cargar la página.

2)
Se generó la siguiente secuencia de acceso a páginas:

0,1,2,3,4,5,0,1,2,3,4,5

Considerando que hay 5 frames, se espera que tenga una alta tasa de page faults: al llegar al quinto acceso, se comienza la secuencia de nuevo, y se va reemplazando la página que será accedida en el paso siguiente.

Se ejecutó el simulador con la siguiente configuración:

python3 paging-policy.py -a 0,1,2,3,4,5,0,1,2,3,4,5 -p LRU -C 5

La configuración define la lista de páginas pedidas, con la política LRU y con 5 marcos de página.

Este fue el resultado de la ejecución:

PÁGINAS PEDIDAS	FRAME 0	FRAME 1	FRAME 2	FRAME 3	FRAME 4	LISTA DE PÁGINAS A DESALOJAR
0	0					0
1	0	1				0 1
2	0	1	2			0 1 2
3	0	1	2	3		0 1 2 3
4	0	1	2	3	4	0 1 2 3 4
5	5	1	2	3	4	1 2 3 4 5
0	5	0	2	3	4	2 3 4 5 0
1	5	0	1	3	4	3 4 5 0 1
2	5	0	1	2	4	4 5 0 1 2
3	5	0	1	2	3	5 0 1 2 3
4	4	0	1	2	3	0 1 2 3 4
5	4	5	1	2	3	1 2 3 4 5

Todos son misses. Una vez que se reemplaza una página de la caché, se realiza el pedido de esa página.

La misma configuración tiene el peor rendimiento posible para la política FIFO:

python3 paging-policy.py -a 0,1,2,3,4,5,0,1,2,3,4,5 -C 5

Y tienen la misma traza de ejecución:

PÁGINAS PEDIDAS	FRAME 0	FRAME 1	FRAME 2	FRAME 3	FRAME 4	LISTA DE PÁGINAS A DESALOJAR
0	0					0
1	0	1				0 1
2	0	1	2			0 1 2
3	0	1	2	3		0 1 2 3
4	0	1	2	3	4	0 1 2 3 4
5	5	1	2	3	4	1 2 3 4 5
0	5	0	2	3	4	2 3 4 5 0
1	5	0	1	3	4	3 4 5 0 1
2	5	0	1	2	4	4 5 0 1 2
3	5	0	1	2	3	5 0 1 2 3
4	4	0	1	2	3	0 1 2 3 4
5	4	5	1	2	3	1 2 3 4 5

Con ambas políticas la traza de ejecución es la misma porque el acceso a una página solo ocurre cuando esta es reemplazada. Por eso, coincide la página menos recientemente usada con la primera que fue cargada a memoria.
La page fault rate para ambas políticas es de 12/12. La peor posible.

La ejecución con la política óptima tiene el siguiente resultado:

PÁGINAS PEDIDAS	FRAME 0	FRAME 1	FRAME 2	FRAME 3	FRAME 4	LISTA DE PÁGINAS A DESALOJAR
0	0					0
1	0	1				1 0

2	0	1	2			2 1 0
3	0	1	2	3		3 2 1 0
4	0	1	2	3	4	4 3 2 1 0
5	0	1	2	3	5	5 3 2 1 0
0	0	1	2	3	5	0 5 3 2 1
1	0	1	2	3	5	0 1 5 3 2
2	0	1	2	3	5	0 1 2 5 3
3	0	1	2	3	5	0 1 2 3 5
4	4	1	2	3	5	1 2 3 4 5
5	4	1	2	3	5	1 2 3 4 5

El óptimo tiene un page fault rate de 7/11.
Mejora considerablemente porque detecta qué páginas son las próximas a utilizar.

Podemos hallar una cantidad de frames tal que con la misma secuencia y política, acercarnos al resultado con OPT.

Esta fue la configuración utilizada:

python3 paging-policy.py -a 0,1,2,3,4,5,0,1,2,3,4,5 -C 6

Y ejecutando el simulador con la política FIFO se obtiene el siguiente resultado:

PÁGINAS PEDIDAS	FRAME 0	FRAME 1	FRAME 2	FRAME 3	FRAME 4	FRAME 5	LISTA DE PÁGINAS A DESALOJAR
0	0						0
1	0	1					0 1
2	0	1	2				0 1 2
3	0	1	2	3			0 1 2 3
4	0	1	2	3	4		0 1 2 3 4
5	0	1	2	3	4	5	0 1 2 3 4 5
0	0	1	2	3	4	5	0 1 2 3 4 5

1	0	1	2	3	4	5	0 1 2 3 4 5
2	0	1	2	3	4	5	0 1 2 3 4 5
3	0	1	2	3	4	5	0 1 2 3 4 5
4	0	1	2	3	4	5	0 1 2 3 4 5
5	0	1	2	3	4	5	0 1 2 3 4 5

El page fault rate es de 6/12.

Con 6 marcos de página, la configuración hallada tiene mejor rendimiento que la óptima con 5 marcos de página.

Para la política LRU, se ejecuta el simulador con la siguiente configuración:

python3 paging-policy.py -a 0,1,2,3,4,5,0,1,2,3,4,5 -p LRU -C 6

PÁGINAS PEDIDAS	FRAME 0	FRAME 1	FRAME 2	FRAME 3	FRAME 4	FRAME 5	LISTA DE PÁGINAS A DESALOJAR
0	0						0
1	0	1					0 1
2	0	1	2				0 1 2
3	0	1	2	3			0 1 2 3
4	0	1	2	3	4		0 1 2 3 4
5	0	1	2	3	4	5	0 1 2 3 4 5
0	0	1	2	3	4	5	1 2 3 4 5 0
1	0	1	2	3	4	5	2 3 4 5 0 1
2	0	1	2	3	4	5	3 4 5 0 1 2
3	0	1	2	3	4	5	4 5 0 1 2 3
4	0	1	2	3	4	5	5 0 1 2 3 4
5	0	1	2	3	4	5	0 1 2 3 4 5

La page fault rate es la misma. Con 6 frames utilizando la política FIFO y LRU la page fault rate es de 6/10, mientras que la política OPT con 5 frames tiene una page fault rate de 7/10.

3) Se generó la siguiente traza de pedidos/accesos a páginas:

0,1,2,3,0,1,1,2,4,0,1

La localidad es temporal: las páginas más recientemente utilizadas son reusadas a corto plazo.

El simulador se ejecuta con la siguiente configuración:

python3 paging-policy.py -a 0,1,2,3,0,1,1,2,4,0,1 -p LRU -C 4

Y la traza de ejecución es la siguiente:

PÁGINAS PEDIDAS	FRAME 0	FRAME 1	FRAME 2	FRAME 3	LISTA DE PÁGINAS A DESALOJAR
0	0				0
1	0	1			0 1
2	0	1	2		0 1 2
3	0	1	2	3	0 1 2 3
0	0	1	2	3	1 2 3 0
1	0	1	2	3	2 3 0 1
1	0	1	2	3	2 3 0 1
2	0	1	2	3	3 0 1 2
4	0	1	2	4	0 1 2 4
0	0	1	2	4	1 2 4 0
1	0	1	2	4	2 4 0 1

La política LRU favorece los accesos cuando siguen el patrón de localidad temporal porque las páginas reemplazadas son aquellas que no fueron recientemente utilizadas.
La page fault rate es de 5/11.

Con la política Second Best la ejecución tiene la siguiente traza:

PÁGINAS PEDIDAS	FRAME 0	FRAME 1	FRAME 2	FRAME 3	LISTA DE PÁGINAS A DESALOJAR
0	0				0
1	0	1			0 1

2	0	1	2		0 1 2
3	0	1	2	3	0 1 2 3
0	0	1	2	3	0 1 2 3
1	0	1	2	3	0 1 2 3
1	0	1	2	3	0 1 2 3
2	0	1	2	3	0 1 2 3
4	0	1	2	4	1 2 3 0 2 3 0 1 3 0 1 2 0 1 2 4
0	0	1	2	4	0 1 2 4
1	0	1	2	4	0 1 2 4

Cuando se busca un página para desalojar, se mira el bit de referencia de la primer página en la lista (en este caso, las páginas referenciadas están en naranja). Si ese bit es 0, se desaloja esa página y si es 1, se limpia el bit, se pasa la página al final de la lista y se continúa con el siguiente elemento de la lista.

Cuando se pide la página 4, se empieza a recorrer la lista:
El primer elemento es 0 y tiene el bit de referencia encendido, entonces pasa al final de la lista y se limpia el bit.
Se sigue con el 2, que también tiene el bit de referencia encendido: pasa al final de la lista y se limpia el bit.
El siguiente elemento de la lista es la página 3, que como no tiene el bit de accedido en 1, es reemplazada.
Se trae la página 4 y se agrega al final de la lista.

La page fault rate es de 5/11.

Esta política también favorece los accesos que siguen el patrón de localidad temporal porque hace que sean desalojadas las páginas que no fueron recientemente accedidas.
Si tuviéramos que todas las páginas fueron accedidas y hay que elegir alguna para desalojar, lo que va a pasar es que se van a limpiar los bits de acceso a cada página y como se agrega cada elemento al final de la lista, se obtiene la misma lista pero con todos los bits de acceso en 0, y se desalojará el primer elemento.