Parte 1: Asignación de memoria

1 - Ejecutar el simulador con estas opciones: *-n 10 -H 0 -p BEST -s 0* **para generar unas pocas operaciones aleatorias. Sin utilizar la opción** *-c***, analizar:**

a) **¿Cuál es el resultado de cada alloc()/free()?**

Al ejecutar el programa desde una terminal por medio del comando *"python3 malloc.py -n 10 -H 0 -p BEST -s 0"* se obtuvo el resultado presentado en el **apéndice 1**. En el mismo se puede observar que ejecutar el programa con el comando mencionado hace que se simule una secuencia de 10 solicitudes aleatorias de reserva y liberación de memoria para un sistema con una free list de 100 unidades de memoria libre contiguas que inicia en la posición 1000 siguiendo la política de asignación 'Best fit' y sin realizar coalescencia 'coalescing'.

El resultado consiste de una breve lista de parámetros que describen el sistema que se está simulando, seguida de una serie de impresiones de dos tipos, de las cuales una corresponde a pedidos de reserva de memoria y el otro tipo de impresión corresponde a solicitudes de liberación de memoria que debe haber sido solicitada previamente.

Las solicitudes de memoria son representadas de la siguiente manera en el resultado del programa:

ptr[x] = Alloc(y) returned ?
List?

Con este tipo de salida se indica que ocurrió una solicitud de reserva de 'y' unidades de memoria que son referenciadas por un puntero que es guardado en la variable ptr[x], dicho puntero apunta al inicio de la sección de memoria reservada, que en el caso de la primera solicitud de reserva seria la dirección 1000 (donde inicia la free list), en cambio para otras solicitudes de reserva que no son la primera el puntero apunta al inicio de una sección de la free list que cumpla con la política 'Best fit'.
Por otra parte al resolver esta solicitud se modifica la free list haciendo que, para el caso de la primera solicitud de reserva la free list pase a iniciar en la posición '1000 + y' tras asignar las posiciones 1000, 1001, …, (1000 + y - 1) para la solicitud de reserva, y para el caso de una solicitud de memoria que no es la primera, se modifica el nodo de la free list del cual se obtendrá la memoria solicitada haciendo que inicie en la posición 'y' unidades de memoria por delante de donde iniciaba anteriormente.
Por su parte las solicitudes de liberación de memoria son representadas de la siguiente manera:

Free(ptr[x])
returned ?
List?

Con esto se indica que ocurrió una solicitud de liberación de memoria que esta siendo referenciada por el puntero guardado en la variable ptr[x]. Este tipo de solicitudes hace que se agregue un nodo nuevo a la free list debido a que no se está realizando coalescencia 'coalescing', el nodo nuevo representa la memoria que era ocupada por la solicitud ptr[x] que inicia en la posición de memoria guardada en ptr[x] y tiene longitud igual a la cantidad de unidades de memoria solicitadas con Alloc(y).

b) **¿Cuál es el estado de la free list luego de cada operación?**

Teniendo en cuenta la explicación presentada en el inciso anterior podemos interpretar la información resultante de la ejecución del programa presentada en el apéndice 1 como:

- Después de Alloc(3).
  Ptr[0] = 1000 y free list = [(1003, 97)]
- Después de Free(ptr[0])
  Ptr[0] = puntero inválido y free list = [(1000, 3)(1003, 97)]
- Después de ptr[1] = Alloc(5)
  ptr[1] = 1003 y free list = [(1000, 3), (1008, 97)]
- Después de Free(ptr[1])
  ptr[1] = puntero inválido y free list = [(1000, 3), (1003, 5), (1008, 92)]
- Después de ptr[2] = Alloc(8)
  ptr[2] = 1008 y free list = [(1000, 3), (1003, 5), (1016, 84)]
- Después de Free(ptr[2])
  ptr[2] = puntero inválido y free list = [(1000, 3), (1003, 5), (1008, 8), (1016, 84)]
- Después de ptr[3] = Alloc(8)
  ptr[3] = 1008 y free list = [(1000, 3), (1003, 5), (1016, 84)]
- Después de Free(ptr[3])
  ptr[3] = puntero inválido y free list = [(1000, 3), (1003, 5), (1008, 8), (1016, 84)]
- Después de ptr[4] = Alloc(2)
  ptr[4] = 1000 y free list = [(1002, 1), (1003, 5), (1008, 8), (1016, 84)]
- Después de ptr[5] = Alloc(7)
  ptr[5] = 1008 y free list = [(1002, 1), (1003, 5), (1015, 1), (1016, 84)]

c) ¿Qué podés observar de la free list a lo largo del tiempo?

Va aumentando la cantidad de nodos a medida que se hacen más solicitudes de reserva lo que causa un aumento en la fragmentación externa, porque no está activada la coalescencia ‘coalescing’, esto puede provocar que al ocurrir una solicitud de reserva haya suficientes unidades de memoria libres para satisfacer el pedido, pero debido a que se encuentra fraccionada en pedazos de menor tamaño al solicitado no se pueda cumplir con la solicitud.

2 - Repetir el análisis anterior para las políticas de WORST FIT y FIRST FIT. ¿Qué cambia?

En los anexos 2 y 3 están los comandos ejecutados y los resultados completos obtenidos para ejecuciones aleatorias de cada política solicitada

Worst fit (Los resultados completos están en el anexo 2)

Free list = [(1000,100)]

- Después de ptr[0] = Alloc(3)
  Free list = [(1003,100)]
- Después de Free(ptr[0])
  Free list = [(1000,3), (1003,97)]
- Después de ptr[1] = Alloc(5)
  Free list = [(1000,3), (1008,97)]
- Después de Free(ptr[1])
  Free list = [(1000,3), (1003,5), (1008,92)]
- Después de ptr[2] = Alloc(8)
  Free list = [(1000,3), (1003,5), (1016,84)]
- Después de Free(ptr[2])

Free list = [(1000,3), (1003,5), (1008,8), (1016,84)]
- Después de ptr[3] = Alloc(8)
  Free list = [(1000,3), (1003,5), (1008,8), (1024,84)]
- Después de Free(ptr[3])
  Free list = [(1000, 3), (1003, 5), (1008, 8), (1016, 8), (1024, 76)]
- Después de ptr[4] = Alloc(2)
  Free list = [(1000, 3), (1003, 5), (1008, 8), (1016, 8), (1026, 74)]
- Después de ptr[5] = Alloc(7)
  Free list = [(1000, 3), (1003, 5), (1008, 8), (1016, 8), (1033, 67)]

Se puede observar como la free list va aumentando la cantidad de nodos a medida que se hacen más solicitudes de reserva al igual que en el caso de la política Best fit, solo que con la política Worst fit el incremento de fragmentación es mayor que con la política anterior, ya que debido a la política siempre se elige el fragmento de memoria mas grande de la free list para obtener la memoria a reservar, esto provoca que al liberar esa memoria se agregue un nuevo nodo a la free list que no se va a volver a usar hasta que ese nodo sea el mas grande y haya una solicitud de memoria del tamaño adecuado para entrar en ese fragmento.

First fit (Los resultados completos están en el anexo 3)

Free list = [(1000, 100)]
- ptr[0] = Alloc(3)
  Free list = [(1003, 97)]
- Free(ptr[0])
  Free list = [(1000, 3), (1003, 97)]
- ptr[1] = Alloc(5)
  Free list = [(1000, 3), (1008, 92)]
- Free(ptr[1])
  Free list = [(1000, 3), (1003, 5), (1008, 92)]
- ptr[2] = Alloc(8)
  Free list = [(1000, 3), (1003, 5), (1016, 84)]
- Free(ptr[2])
  Free list = [(1000, 3), (1003, 5), (1008, 8), (1016, 84)]
- ptr[3] = Alloc(8)
  Free list = [(1000, 3), (1003, 5), (1016, 84)]
- Free(ptr[3])
  Free list = [(1000, 3), (1003, 5), (1008, 8), (1016, 84)]
- ptr[4] = Alloc(2)
  Free list = [(1002, 1), (1003, 5), (1008, 8), (1016, 84)]
- ptr[5] = Alloc(7)
  Free list = [(1002, 1), (1003, 5), (1015, 1), (1016, 84)]

Con la política First fit para el ejemplo aleatorio generado se obtuvo el mismo resultado que para el caso con política Best fit, pero mas allá de estos casos particulares y el resultado que generaron, las dos políticas producen resultados muy similares.

3. Incrementar la cantidad de asignaciones aleatorias (digamos -n 1000). ¿Qué sucede con las asignaciones más grandes a lo largo del tiempo? Ejecutar con y sin coalescing (o sea, con y sin la opción -C) y analizar las diferencias en los resultados.

Al incrementar la cantidad de solicitudes de reserva de memoria lo que ocurre es que aumenta el nivel de fragmentación externa de manera relativa a la cantidad de asignaciones que se hacen, al no estar activado el mecanismo de coalescencia la memoria va dividiéndose en fragmentos cada vez mas chicos y el tamaño máximo de solicitudes de reserva de memoria que se puede satisfacer se vuelve cada vez menor

hasta que eventualmente puede llegar a ser 1 y no se van a poder satisfacer solicitudes de reserva mayores a 1 unidad de memoria.

Sin embargo al ejecutar el mismo comando pero con la opción de coalescencia activada ocurre algo completamente distinto, la fragmentación externa se ve reducida en gran medida, esto es así debido a que este mecanismo permite fusionar fragmentos de memoria libre adyacentes, lo cual permite que la fragmentación externa se mantenga siempre en niveles muy bajos. Esto permite que nunca una solicitud de reserva de memoria no se pueda satisfacer debido a fragmentación externa, aun así hay casos en los que no se pueden satisfacer ciertas solicitudes pero son casos en los que la memoria esta tan llena que no hay suficiente espacio libre.

Como medida de comparación entre los resultados de fragmentación entre ambos casos, podemos ver la cantidad máxima de elementos de la free list para cada caso, para el caso sin coalescencia activada la free list alcanzó un máximo de 51 fragmentos de memoria, en cambio en el caso de la coalescencia activada la free list solo se alcanzó un máximo de 8 fragmentos de memoria. Esto no solo hace que activar la coalescencia permita satisfacer muchas mas solicitudes de reserva de memoria sino que también permite mejorar la velocidad de dichas solicitudes porque la mmu no tiene que recorrer una free list mucho menor.

Parte 2: Reemplazo de páginas

1 - Generar accesos aleatorios con los siguientes argumentos: -s 0 -n 10, -s 1 -n 10, y -s 2 -n 10. Probar con las políticas FIFO, LRU y OPT. Sin utilizar la opción -c, calcular si cada acceso resulta en hit o miss.

Al ejecutar el programa desde una terminal por medio del comando *“python3 paging-policy.py -s 0 -n 10 -c”* se obtuvo el resultado presentado en el **apéndice 4**. En el mismo se puede observar que ejecutar el programa con el comando mencionado hace que se simule una secuencia de 10 accesos a memoria, con un cache de 3 páginas siguiendo una política FIFO.

Los resultados obtenidos se presentan en el apéndice 4, los cuales consisten en una breve serie de parámetros que describen al sistema que se esta simulando, seguidos de una secuencia de números de página que representan los accesos a memoria generados aleatoriamente.

Resultados de ejecutar: *python3 paging-policy.py -s 0 -n 10*

Páginas pedidas	Acceso	Marcos de página	Páginas a desalojar
Access: 8	MISS	[8]	-
Access: 7	MISS	[8, 7]	-
Access: 4	MISS	[8, 7, 4]	-
Access: 2	MISS	[7, 4, 2]	8
Access: 5	MISS	[4, 2, 5]	7
Access: 4	HIT	[4, 2, 5]	-
Access: 7	MISS	[2, 5, 7]	4
Access: 3	MISS	[5, 7, 3]	2
Access: 4	MISS	[7, 3, 4]	5
Access: 5	MISS	[3, 4, 5]	7

Resultados de ejecutar: *python3 paging-policy.py -s 1 -n 10*

Páginas pedidas	Acceso	Marcos de página	Páginas a desalojar
Access: 1	MISS	[1]	-
Access: 8	MISS	[1, 8]	-
Access: 7	MISS	[1, 8, 7]	-
Access: 2	MISS	[8, 7, 2]	1
Access: 4	MISS	[7, 2, 4]	8
Access: 4	HIT	[7, 2, 4]	-
Access: 6	MISS	[2, 4, 6]	7
Access: 7	MISS	[4, 6, 7]	2
Access: 0	MISS	[6, 7, 0]	4
Access: 0	HIT	[6, 7, 0]	-

Resultados de ejecutar: *python3 paging-policy.py -s 2 -n 10*

Páginas pedidas	Acceso	Marcos de página	Páginas a desalojar
Access: 9	MISS	[9]	-
Access: 9	HIT	[9]	-
Access: 0	MISS	[9, 0]	-
Access: 0	HIT	[9, 0]	-
Access: 8	MISS	[9, 0, 8]	-
Access: 7	MISS	[0, 8, 7]	9
Access: 6	MISS	[8, 7, 6]	0
Access: 3	MISS	[7, 6, 3]	8
Access: 6	HIT	[7, 6, 3]	-
Access: 6	HIT	[7, 6, 3]	-

Resultados de ejecutar: *python3 paging-policy.py -s 0 -n 10 –policy=OPT*

Páginas pedidas	*Acceso*	*Marcos de página*	*Páginas a desalojar*
Access: 8	*MISS*	*[8]*	-
Access: 7	MISS	[8, 7]	-
Access: 4	MISS	[8, 7, 4]	-
Access: 2	MISS	[7, 4, 2]	8
Access: 5	MISS	[4, 2, 5]	7
Access: 4	HIT	[2, 5, 4]	-
Access: 7	MISS	[5, 4, 7]	2
Access: 3	MISS	[4, 7, 3]	5
Access: 4	HIT	[7, 3, 4]	-
Access: 5	MISS	[3, 4, 5]	7

Resultados de ejecutar: *python3 paging-policy.py -s 1 -n 10 –policy=OPT*

Páginas pedidas	*Acceso*	*Marcos de página*	*Páginas a desalojar*
Access: 1	*MISS*	*[1]*	-
Access: 8	MISS	[1, 8]	-
Access: 7	MISS	[1, 8, 7]	-
Access: 2	MISS	[8, 7, 2]	1
Access: 4	MISS	[7, 2, 4]	8
Access: 4	HIT	[7, 2, 4]	-

Access: 6	MISS	[2, 4, 6]	7
Access: 7	MISS	[4, 6, 7]	2
Access: 0	MISS	[6, 7, 0]	4
Access: 0	HIT	[6, 7, 0]	-

Resultados de ejecutar: python3 paging-policy.py -s 2 -n 10 --policy=OPT

Páginas pedidas	*Acceso*	*Marcos de página*	*Páginas a desalojar*
Access: 9	*MISS*	*[9]*	*- [Hits:0 Misses:1]*
Access: 9	HIT	[9]	- [Hits:1 Misses:1]
Access: 0	MISS	[9, 0]	- [Hits:1 Misses:2]
Access: 0	HIT	[9, 0]	- [Hits:2 Misses:2]
Access: 8	MISS	[9, 0, 8]	- [Hits:2 Misses:3]
Access: 7	MISS	[0, 8, 7]	9 [Hits:2 Misses:4]
Access: 6	MISS	[8, 7, 6]	0 [Hits:2 Misses:5]
Access: 3	MISS	[7, 6, 3]	8 [Hits:2 Misses:6]
Access: 6	HIT	[7, 3, 6]	- [Hits:3 Misses:6]
Access: 6	HIT	[7, 3, 6]	- [Hits:4 Misses:6]

2 - Considerando 5 marcos de página, generar secuencias de referencias a páginas que tengan el peor rendimiento posible para FIFO y LRU (es decir, que provoquen la mayor cantidad de misses posible). Para la peor secuencia hallada, ¿cuántos marcos de página podrían mejorar el rendimiento y acercarse a OPT?

3 - Generar una traza de pedidos con alguna localidad. Explicar cómo se generó dicha traza y cómo se comporta LRU en ese caso. Hacer el análisis también para SECOND CHANCE (no incluido en el simulador).

Apéndice 1

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

```
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute False
ptr[0] = Alloc(3) returned ?
List?
Free(ptr[0])
returned ?
List?
ptr[1] = Alloc(5) returned ?
List?
Free(ptr[1])
returned ?
```

List?
ptr[2] = Alloc(8) returned ?
List?
Free(ptr[2])
returned ?
List?
ptr[3] = Alloc(8) returned ?
List?
Free(ptr[3])
returned ?
List?
ptr[4] = Alloc(2) returned ?
List?
ptr[5] = Alloc(7) returned ?
List?

Apéndice 2

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

seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy WORST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute False
ptr[0] = Alloc(3) returned ?
List?
Free(ptr[0])
returned ?
List?
ptr[1] = Alloc(5) returned ?
List?
Free(ptr[1])
returned ?
List?
ptr[2] = Alloc(8) returned ?
List?
Free(ptr[2])
returned ?
List?
ptr[3] = Alloc(8) returned ?
List?
Free(ptr[3])
returned ?
List?
ptr[4] = Alloc(2) returned ?
List?
ptr[5] = Alloc(7) returned ?
List?

Apéndice 3

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

seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy FIRST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute False
ptr[0] = Alloc(3) returned ?
List?
Free(ptr[0])
returned ?
List?
ptr[1] = Alloc(5) returned ?
List?
Free(ptr[1])
returned ?
List?
ptr[2] = Alloc(8) returned ?
List?
Free(ptr[2])
returned ?
List?
ptr[3] = Alloc(8) returned ?
List?
Free(ptr[3])
returned ?
List?
ptr[4] = Alloc(2) returned ?
List?
ptr[5] = Alloc(7) returned ?
List?

Apéndice 4

ARG addresses -1
ARG addressfile
ARG numaddrs 10
ARG policy FIFO
ARG clockbits 2
ARG cachesize 3
ARG maxpage 10
ARG seed 0
ARG notrace False
Assuming a replacement policy of FIFO, and a cache of size 3 pages,
figure out whether each of the following page references hit or miss
in the page cache.
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?

Apéndice 5

ARG addresses -1
ARG addressfile
ARG numaddrs 10
ARG policy FIFO
ARG clockbits 2
ARG cachesize 3
ARG maxpage 10
ARG seed 1
ARG notrace False
Assuming a replacement policy of FIFO, and a cache of size 3 pages,
figure out whether each of the following page references hit or miss
in the page cache.
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?

Apéndice 6

ARG addresses -1
ARG addressfile
ARG numaddrs 10
ARG policy FIFO
ARG clockbits 2
ARG cachesize 3
ARG maxpage 10
ARG seed 2
ARG notrace False
Assuming a replacement policy of FIFO, and a cache of size 3 pages,
figure out whether each of the following page references hit or miss
in the page cache.
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?

Apéndice 7

ARG addresses -1
ARG addressfile
ARG numaddrs 10
ARG policy FIFO
ARG clockbits 2
ARG cachesize 3
ARG maxpage 10
ARG seed 0
ARG notrace False
Assuming a replacement policy of LRU, and a cache of size 3 pages,
figure out whether each of the following page references hit or miss
in the page cache.
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?

Apéndice 8

ARG addresses -1
ARG addressfile
ARG numaddrs 10
ARG policy FIFO
ARG clockbits 2
ARG cachesize 3
ARG maxpage 10
ARG seed 1
ARG notrace False
Assuming a replacement policy of LRU, and a cache of size 3 pages,
figure out whether each of the following page references hit or miss
in the page cache.
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?

Apéndice 9

ARG addresses -1
ARG addressfile
ARG numaddrs 10

ARG policy FIFO
ARG clockbits 2
ARG cachesize 3
ARG maxpage 10
ARG seed 2
ARG notrace False
Assuming a replacement policy of LRU, and a cache of size 3 pages,
figure out whether each of the following page references hit or miss
in the page cache.
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?