Trabajo Práctico 2: Memoria

Lisandro Córdoba Lazzaro

3/27/23

Parte I
Asignación de memoria

1. Ejercicio 1

Primero aclaremos lo que hace el comando a ejecutar.

```
./malloc.py -n 10 -H 0 -p BEST -s 0
```

- '-n 10' indica que se generarán 10 operaciones aleatorias.
- '-H 0' indica que los headers tienen tamaño 0. Es decir, que cuando se reserva memoria con alloc, no se reserva espacio extra para header.
- '-p BEST' indica que se utilizará la política Best Fit para asignar memoria.
- '-s 0' setea la semilla 0 para la generación aleatoria.
- Por defecto, el tamaño máximo de asignación es 10, el orden de la free list es por address y el coalescing está desactivado.

A continuación, listo las operaciones generadas:

Orden	Operación
1	ptr[0] = Alloc(3)
2	free(ptr[0])
3	ptr[1] = Alloc(5)
4	free(ptr[1])
5	ptr[2] = Alloc(8)
6	free(ptr[2])
7	ptr[3] = Alloc(8)
8	free(ptr[3])
9	ptr[4] = Alloc(2)
10	ptr[5] = Alloc(7)

1.a. Resultado de cada alloc()/free()

Dado que el enunciado aclara que no se debe utilizar la opción '-c', respondo en base a lo que esperaría que hagan un malloc y un free. Además, analicé el código de malloc.py.

- Luego de cada ***alloc()*** se retorna el puntero a la dirección de memoria reservada junto con la cantidad de pasos que dio en la free list para encontrar el bloque asignado. Si no encontró memoria contigua suficiente para reservar, retorna -1.
- Luego de cada ***free()*** se retorna 0 si fue exitoso y -1 si hubo un error.

Como todos los free() son hechos a punteros válidos, devolverán 0.
A continuación, detallo el resultado de cada alloc():

Orden	Operación	Ptr	Count
1	Alloc(3)	1000	1
2	Alloc(5)	1003	2
3	Alloc(8)	1008	3
4	Alloc(8)	1008	4
5	Alloc(2)	1000	4
6	Alloc(7)	1008	4

Algunas cosas a notar:

- Los primeros 4 alloc() se asignan en el último bloque, pues es el más grande. Si bien cada alloc() hace su respectivo free() y, por lo tanto, los bloques se agregan a la free list, el tamaño de estos nunca es suficiente para las siguientes peticiones de memoria.
- El 5° alloc se vuelve a reservar en el comienzo del heap, pues en ese entonces hay un bloque libre de tamaño 3 y el pedido es únicamente de 2 bytes.
- Al seguir la política Best-fit, count siempre será igual a la cantidad de bloques en la free list, pues debe recorrerla toda para elegir la mejor opción.

1.b. Estado de la free list luego de cada operación

En primer lugar, veamos cómo se inicializa la free list.

Nodo	**Addr**	**Size**
0	1000	100

Ahora veamos cómo evoluciona con cada operación, teniendo en cuenta que, por defecto, el orden es por address ascendente.

Nodo	**Addr**	**Size**
0	1003	97

Cuadro 1: Free list luego de ptr[0] = Alloc(3)

Nodo	**Addr**	**Size**
0	1000	3
1	1003	97

Cuadro 2: Free list luego de free(ptr[0])

Nodo	**Addr**	**Size**
0	1000	3
1	1008	92

Cuadro 3: Free list luego de ptr[1] = Alloc(5)

Nodo	**Addr**	**Size**
0	1000	3
1	1003	5
2	1008	92

Cuadro 4: Free list luego de free(ptr[1])

Nodo	**Addr**	**Size**
0	1000	3
1	1003	5
2	1016	84

Cuadro 5: Free list luego de ptr[2] = Alloc(8)

Nodo	**Addr**	**Size**
0	1000	3
1	1003	5
2	1008	8
3	1016	84

Cuadro 6: Free list luego de free(ptr[2])

Nodo	Addr	Size
0	1000	3
1	1003	5
2	1016	84

Cuadro 7: Free list luego de ptr[3] = Alloc(8)

Nodo	Addr	Size
0	1000	3
1	1003	5
2	1008	8
3	1016	84

Cuadro 8: Free list luego de free(ptr[3])

Nodo	Addr	Size
0	1002	1
1	1003	5
2	1008	8
3	1016	84

Cuadro 9: Free list luego de ptr[4] = Alloc(2)

Nodo	Addr	Size
0	1002	1
1	1003	5
2	1015	1
3	1016	84

Cuadro 10: Free list luego de ptr[5] = Alloc(7)

1.c. La free list a lo largo del tiempo

A medida que las operaciones ocurren, la free list queda cada vez más fragmentada. Esto se debe a que no está activado el coalescing, por lo que en cada free() queda un bloque independiente de memoria.
De esta forma, se permite que haya muchos bloques libres chicos contiguos, en vez de pocos bloques grandes. Veamos un ejemplo de esto:
En el cuadro 2 podemos ver que los 100 MB del heap quedaron libres, pero divididos en 2 bloques. Por esto, en el cuadro 3 el alloc(5) debe fragmentar el segundo bloque de memoria, pues el primero quedó aislado como un bloque de tamaño 3.

2. Ejercicio 2

2.a. Worst Fit

Esta política aumenta aún más la fragmentación sin coalescing, pues todos los alloc() reservan memoria del bloque más grande (en este caso siempre el último).
Para ejemplificar, podemos pensar en el Cuadro 7: con Best Fit es uno de los pocos casos donde no se fragmenta la memoria, pues se reserva un bloque cuyo tamaño es exactamente el pedido. Pero al usar Worst Fit se asignará el bloque más grande, dejando fragmentaciones en cada paso.

2.b. First Fit

Como se puede ver en los cuadros de Best Fit, para todos los alloc() vale: dentro de los bloques de la free list cuyos tamaños son mayores o iguales a la memoria solicitada, el de menor tamaño es también el primero.
Por esta razón, la política First Fit será igual a Best Fit para estas operaciones generadas.
Lo único que será diferente es la variable Count retornada por los alloc(). En esta política no es necesario iterar por toda la lista para asegurarse de que el bloque encontrado sea el Best/Worst; simplemente, una vez que se encuentra un bloque libre, se asigna.

3. Ejercicio 3

Al aumentar la cantidad de asignaciones aleatorias se evidencia más claramente la diferencia entre usar o no coalescing.
Al tenerlo desactivado, llega un punto en que la memoria está tan fragmentada que puede estar la mayor parte del heap libre, pero dividido en bloques pequeños. Podemos notar que la última operación es un alloc(9) y no logra ser reservado por más que esté casi todo el heap vacío.
En cambio, al usar coalescing no hay ningún alloc() que falle. Cada vez que hay dos bloques libres contiguos, estos se unen formando uno de mayor tamaño y permitiendo que lleguen reservas más grandes.

Parte II
Algoritmo de reemplazo de páginas

4. Ejercicio 1

En esta sección vamos a analizar 3 secuencias distintas de 10 accesos de páginas, con el fin de comparar las políticas FIFO, LRU y ÓPTIMO.
Cabe aclarar que, por defecto, el cache size (la cantidad de páginas disponibles en memoria) es 3.

4.a. Primera semilla (0)

Ejecutamos:

```
./paging-policy.py -s 0 -n 10
```

Y la secuencia de accesos obtenida es:

```
[8, 7, 4, 2, 5, 4, 7, 3, 4, 5]
```

4.a.1. Similitud entre FIFO y LRU

En un primer análisis general, sabemos que tanto con FIFO como con LRU, si una página es ingresada al caché y se la quiere volver a acceder luego de 3 reemplazos, será un page fault. Esto se debe a que al tercer reemplazo ya será su turno de ser desalojada, pues tenemos cache size = 3.
En la secuencia obtenida podemos observar que solo las páginas 4, 5 y 7 son accedidas más de una vez. Sin embargo, la única que pasa menos de 3 reemplazos hasta ser accedida es la 4, por lo que será la única que dé HIT.
Todas las demás páginas serán misses.

4.a.2. Diferencia entre FIFO y LRU

La única diferencia que veremos será en el 3° acceso a 4.

- FIFO: al acceder a 4 por 2° vez obtenemos un hit, pero no se le renueva la vida en caché. Por lo que en el siguiente paso ya es su turno de ser desalojada, y el 3° acceso a 4 es un miss.
- LRU: al acceder a 4 por 2° vez se la envía al final de la cola de desalojo. Por esto, en el 3° acceso a 4 tendremos un hit, pues no ocurrieron 3 pasos y sigue en caché.

4.a.3. Reemplazo óptimo

Si siguiéramos la política de desalojar la página que pasará más tiempo sin ser accedida, lograríamos hit en todos los casos donde se accede una página que anteriormente ya fue accedida. Es decir, cuando volvemos a acceder a 4, 5 y 7 ya las tenemos en caché.
Esto se debe a que tenemos cache size = 3 y, por el orden de los accesos, siempre podemos preservar una página que luego será nuevamente accedida.

4.a.4. Conclusión

- FIFO: 9 page faults y 1 hit.
- LRU: 8 page faults y 2 hits.
- OPT: 6 page faults y 4 hits.

4.b. Segunda semilla (1)

Ejecutamos:

```
./paging-policy.py -s 1 -n 10
```

Y la secuencia de accesos obtenida es:

```
[1, 8, 7, 2, 4, 4, 6, 7, 0, 0]
```

4.b.1. Igualdad entre FIFO y LRU

Las páginas 4 y 0 vuelven a ser accedidas inmediatamente después de ingresar al caché. Gracias a esto, el 2° acceso a cada página son hits tanto con FIFO como con LRU.
Si bien la página 7 también es accedida dos veces, el 2° acceso ocurre luego de más de 2 reemplazos, por lo que no logramos un hit.
Las demás páginas son accedidas únicamente 1 vez y son misses en ambas políticas.
De esta forma, ambas políticas se comportan igual con esta secuencia de accesos.

4.b.2. Reemplazo óptimo

En un algoritmo de reemplazo óptimo lograríamos hit en los mismos casos que FIFO y LRU, pero también en el 2° acceso a 7. Veamos por qué.
Supongamos que luego del 1° acceso a 7 la reservamos en caché y manejamos los próximos accesos a páginas únicamente con los 2 lugares restantes.

```
[..., 2, 4, 4, 6, ...]
```

Podemos acceder a 2, luego reemplazarla por 4, el siguiente acceso a 4 es hit y finalmente reemplazamos 4 por 6.
Claramente, con un solo lugar en el caché fue suficiente, por lo que 7 puede seguir cacheada sin perjudicar accesos a otras páginas.

4.b.3. Conclusión

- FIFO: 8 page faults y 2 hits.
- LRU: 8 page faults y 2 hits.
- OPT: 7 page faults y 3 hits.

4.c. Tercera semilla (2)

Ejecutamos:

```
./paging-policy.py -s 2 -n 10
```

Y la secuencia de accesos obtenida es:

```
[9, 9, 0, 0, 8, 7, 6, 3, 6, 6]
```

4.c.1. Igualdad entre FIFO, LRU y OPT

Por las mismas razones desarrolladas anteriormente, FIFO y LRU se comportarán igual en esta semilla. Las páginas 9, 0 y 6 logran hits en sus siguientes accesos luego de ser accedidas por primera vez.
OPT no puede lograr un mejor rendimiento que este, pues no hay más páginas que se accedan nuevamente.

4.c.2. Conclusión

- FIFO: 6 page faults y 4 hits.
- LRU: 6 page faults y 4 hits.
- OPT: 6 page faults y 4 hits.

5. Ejercicio 2

Buscamos generar una secuencia de accesos donde tanto FIFO como LRU sean ineficientes con 5 marcos de pagina. Luego analizaremos cuantos marcos habria que agregar para que ambos algoritmos se comporten similar a OPT.

5.a. Secuencia de accesos propuesta

2 rafagas de accesos de las paginas 1 a 6 en orden.

```
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

Tanto con FIFO como con LRU tendremos 12 misses. Cuando una página quiera ser accedida por 2° vez no estará en cache, pues pasaron más de 4 reemplazos e inevitablemente ya pasó su turno de ser desalojada.

Ejecutamos:

```
./paging-policy.py -a 1,2,3,4,5,6,1,2,3,4,5,6 -C 5 -c -p FIFO
./paging-policy.py -a 1,2,3,4,5,6,1,2,3,4,5,6 -C 5 -c -p LRU
```

Y comprobamos que efectivamente obtenemos 12 misses para FIFO y LRU respectivamente.

5.b. Comparación con OPT

Es imposible evitar los misses de la 1° ronda, pero bajo un reemplazo óptimo podríamos obtener 5 hits en la 2°. Una forma de lograr esto sería mantener en caché las páginas 1, 2, 3 y 4 durante la 1° ronda, dejando el lugar restante del caché para acceder a 5 y 6. En la 2° ronda obtenemos hits en las páginas 1, 2, 3, 4 y 6.

Pag Pedidas	Frames	Pags a desalojar
1	1 - - - -	1
2	1 2 - - -	1, 2
3	1 2 3 - -	1, 2, 3
4	1 2 3 4 -	1, 2, 3, 4
5	1 2 3 4 5	1, 2, 3, 4, 5
6	1 2 3 4 6	6, 1, 2, 3, 4
1	1 2 3 4 6	1, 2, 3, 4, 6
2	1 2 3 4 6	1, 2, 3, 4, 6
3	1 2 3 4 6	1, 2, 3, 4, 6
4	1 2 3 4 6	1, 2, 3, 4, 6
5	5 2 3 4 6	5, 2, 3, 4, 6
6	5 2 3 4 6	5, 2, 3, 4, 6

Óptimo

Pag Pedidas	Frames	Pags a desalojar
1	1 - - - -	1
2	1 2 - - -	1, 2
3	1 2 3 - -	1, 2, 3
4	1 2 3 4 -	1, 2, 3, 4
5	1 2 3 4 5	1, 2, 3, 4, 5
6	6 2 3 4 5	2, 3, 4, 5, 6
1	6 1 3 4 5	3, 4, 5, 6, 1
2	6 1 2 4 5	4, 5, 6, 1, 2
3	6 1 2 3 5	5, 6, 1, 2, 3
4	6 1 2 3 4	6, 1, 2, 3, 4
5	5 1 2 3 4	1, 2, 3, 4, 5
6	5 6 2 3 4	2, 3, 4, 5, 6

LRU y FIFO

5.c. Mejora de FIFO y LRU agregando marcos de pagina

Con solo agregar 1 marco de pagina llegamos al rendimiento máximo tanto para LRU como FIFO. La 1° ronda seguiran siendo 6 misses inevitables, pero la 2° ronda es una secuencia de 6 hits, pues ninguna pagina es desalojada.

Pag Pedidas	Frames	Pags a desalojar
1	1 - - - - -	1
2	1 2 - - - -	1, 2
3	1 2 3 - - -	1, 2, 3
4	1 2 3 4 - -	1, 2, 3, 4
5	1 2 3 4 5 -	1, 2, 3, 4, 5
6	1 2 3 4 5 6	1, 2, 3, 4, 5, 6
1	1 2 3 4 5 6	1, 2, 3, 4, 5, 6
2	1 2 3 4 5 6	1, 2, 3, 4, 5, 6
3	1 2 3 4 5 6	1, 2, 3, 4, 5, 6
4	1 2 3 4 5 6	1, 2, 3, 4, 5, 6
5	1 2 3 4 5 6	1, 2, 3, 4, 5, 6
6	1 2 3 4 5 6	1, 2, 3, 4, 5, 6

Óptimo

Pag Pedidas	Frames	Pags a desalojar
1	1 - - - - -	1
2	1 2 - - - -	1, 2
3	1 2 3 - - -	1, 2, 3
4	1 2 3 4 - -	1, 2, 3, 4
5	1 2 3 4 5 -	1, 2, 3, 4, 5
6	1 2 3 4 5 6	1, 2, 3, 4, 5, 6
1	1 2 3 4 5 6	2, 3, 4, 5, 6, 1
2	1 2 3 4 5 6	3, 4, 5, 6, 1, 2
3	1 2 3 4 5 6	4, 5, 6, 1, 2, 3
4	1 2 3 4 5 6	5, 6, 1, 2, 3, 4
5	1 2 3 4 5 6	6, 1, 2, 3, 4, 5
6	1 2 3 4 5 6	1, 2, 3, 4, 5, 6

LRU y FIFO

5.c.1. Conclusión

- 5 marcos de página.
  - FIFO con 5 marcos: 12 page faults y 0 hits.
  - LRU con 5 marcos: 12 page faults y 0 hits.
  - OPT con 5 marcos: 5 page faults y 7 hits.
- 6 marcos de página.
  - FIFO con 6 marcos: 6 page faults y 6 hits.
  - LRU con 6 marcos: 6 page faults y 6 hits.
  - OPT con 6 marcos: 6 page faults y 6 hits.

6. Ejercicio 3

Supongamos que tenemos cache size = 4, queremos ver cómo las políticas de LRU y Second Chance aprovechan la localidad de los accesos.

6.a. Secuencia de accesos propuesta

El conjunto de páginas 1, 2 y 3 son accedidas siempre juntas por localidad espacial, pero luego de acceder a las 3 se accede a una página externa a la localidad.

```
[1, 2, 3, 10, 3, 2, 1, 11, 1, 2, 3, 12, 1, 2, 3]
```

6.b. Análisis de la secuencia en LRU y Second Chance

Tanto LRU como Second Chance favorecen la localidad priorizando las páginas que son recurrentemente accedidas. Ambos algoritmos logran hits en las rafagas del conjunto 1,2,3 sin importar en que orden sean los accesos ni el acceso fuera de localidad luego de cada ráfaga
Es importante notar que esta con FIFO no veríamos estos beneficios: el acceso fuera de localidad provocaria que todos los accesos sean misses pues todas las paginas tendran que ser desalojadas por orden de llegada.

Pag Pedidas	Frames	Pags a desalojar
1	1 - - -	1
2	1 2 - -	1, 2
3	1 2 3 -	1, 2, 3
10	1 2 3 10	1, 2, 3, 10
3	1 2 3 10	1, 2, 10, 3
2	1 2 3 10	1, 10, 3, 2
1	1 2 3 10	10, 3, 2, 1
11	1 2 3 11	1, 2, 3, 11
1	1 2 3 11	2, 3, 11, 1
2	1 2 3 11	3, 11, 1, 2
3	1 2 3 11	11, 1, 2, 3
12	1 2 3 12	1, 2, 3, 12
1	1 2 3 12	2, 3, 12, 1
2	1 2 3 12	3, 12, 1, 2
3	1 2 3 12	12, 1, 2, 3

LRU (Least Recently Used)

Pag Pedidas	Frames	Pags a desalojar
1	1 - - -	1
2	1 2 - -	1, 2
3	1 2 3 -	1, 2, 3
10	1 2 3 10	1, 2, 3, 10
3	1 2 3 10	1, 2, 3*, 10
2	1 2 3 10	1, 2*, 3*, 10
1	1 2 3 10	1*, 2*, 3*, 10
11	1 2 3 11	1, 2, 3, 11
1	1 2 3 11	1*, 2, 3, 11
2	1 2 3 11	1*, 2*, 3, 11
3	1 2 3 11	1*, 2*, 3*, 11
12	1 2 3 12	1, 2, 3, 12
1	1 2 3 12	1*, 2, 3, 12
2	1 2 3 12	1*, 2*, 3, 12
3	1 2 3 12	1*, 2*, 3*, 12

Second Chance (Accedida marcada con *)