

# Sistemas Operativos

Departamento de Computación - FCEyN - UBA  
Primer cuatrimestre de 2019

Primer parcial - 30/4 - Primer cuatrimestre de 2019

1	2	3	4	Nota
B	R	B-	B	A

SERGIO ROMANO

**ACLARACIONES:** 1) Numere las hojas entregadas. Esta hoja se entrega y es la hoja cero. Complete en la primera hoja la cantidad total de hojas entregadas (sin contar el enunciado). 2) Realice cada ejercicio en hojas separadas y escriba nombre, apellido y L.U. en cada una. 3) Cada ejercicio se califica con Bien, Regular o Mal. La división de los ejercicios en incisos es meramente orientativa. Los ejercicios se califican globalmente. El parcial se aprueba con 2 ejercicios bien y a lo sumo 1 mal/incompleto. 4) El parcial NO es a libro abierto. 5) Justifique adecuadamente cada una de sus respuestas.

## Ejercicio 1.

Se tiene la estructura `Nodo` con dos componentes (`numero` y `siguiente`) que sirven para armar una lista de números. El valor por defecto de `numero` es 0 y el valor por defecto de `siguiente` es `null`;

```
struct Nodo {
    int numero;
    Nodo *siguiente;
};
```

Para conocer donde empieza la lista se tiene un puntero a una variable atómica global llamada `cabeza` (`atomic<Nodo*> cabeza`) que apunta al primer nodo.

Se desea implementar una función llamada `insertar_adelante`, que agregue un elemento al principio de la lista y pueda ser usado de manera concurrente por muchos procesos. Se propone el siguiente código:

```
void insertar_adelante(int numero_nuevo) {
    Nodo* nuevo = new Nodo();
    nuevo->numero = numero_nuevo;
    nuevo->siguiente = atomic_exchange(cabeza, nuevo);
}
```

*nuevo = \*cabeza*  
*cabeza = nuevo*

- Asumiendo que la función `insertar_adelante` tiene que ser atómica, explique qué problemas presenta esta solución si otros procesos intentan recorrer la lista en simultáneo.
- Proponga una solución utilizando variables atómicas (no se permite usar semáforos, ni mutex)

Tiene las siguientes funciones a disposición:

- `Nodo* atomic_load(atomic<Nodo*> objeto)`: Devuelve el valor actual del objeto atómico.
- `Nodo* atomic_exchange(atomic<Nodo*> objeto, Nodo* deseado)`: Reemplaza de manera atómica el valor del objeto atómico (el nodo apuntado por el objeto atómico) por el puntero deseado, y devuelve el valor que tenía el objeto atómico previamente.
- `bool atomic_compare_exchange(atomic<Nodo*> objeto, Nodo* esperado, Nodo* deseado)`: Compara de manera atómica el contenido del objeto atómico con el valor esperado. Si son el mismo, reemplaza el objeto atómico por el deseado. Si no son el mismo, carga el valor actual del objeto atómico en el puntero esperado. Devuelve el resultado de la comparación (true si `*objeto` era igual a `*esperado`, falso de lo contrario).

## Ejercicio 2.

- Nombre una ventaja y una desventaja del concepto de afinidad al procesador en Scheduling.
- En schedulers de multiples colas de prioridad, cada cola suele estar asignada a procesos de distinta prioridad y, a su vez, cada cola puede asignar un quantum distinto a los procesos en ella a la hora de ejecutarse. ¿Suele la relación entre quantum y prioridad ser directamente proporcional o inversamente proporcional? ¿Por qué? Mencione cómo esto podría influir en el tiempo de respuesta y en el throughput de procesos interactivos y de procesos intensivos en CPU.
- ¿Cuál es la diferencia entre un scheduler con desalojo y uno sin desalojo? Elija un scheduler de cada categoría y explique en detalle su funcionamiento.

*Si coligo en 7*  
*proceder*

## Ejercicio 3.

*Procesos en vez de threads.**colectiva  
escritura*

Se tiene un grafo conexo y no dirigido de 10 vértices, es decir, cada par de vértices está conectado por al menos un camino. Se desea crear la función `void pintar(int *colores)` que crea un thread por cada uno de los diez vértices y toma como parámetro una lista de colores para asignar a cada vértice según su número (`int colores[10]`).

El thread número cero será el encargado de pasar la lista de colores a los demás threads para que cada uno pueda llamar a la función `void pintame(int color)`. La comunicación entre dos threads se realizará mediante pipes y sólo se podrá realizar si son vecinos, es decir, si existe un eje entre ese par de vértices/threads (`bool sonVecinos(int numeroThreadA, int numeroThreadB)`).

Escribir el código de la función `pintar` que garantice que todos los threads reciban su color y puedan pintarse.

**Sugerencia:** Cada vértice/thread puede tener muchos vecinos, no es necesario que sólo uno de ellos le comunique su color. Aunque seguramente le alcance con enterarse por primera vez de la lista para comunicársela a sus propios vecinos.

## Ejercicio 4.

- a) Considere el arreglo de dos dimensiones A:

```
char A[][] = new char[100][100];
```

donde `A[0][0]` está en la dirección de memoria 200, en un sistema de memoria paginado, con páginas de tamaño 200B. Un proceso pequeño, que manipula la matriz, está localizado en la página 0 (con direcciones de la 0 a la 199). Así cada solicitud de instrucción se hará a partir de la página 0. Si se tienen tres marcos de página, cuántos fallos de página se generan utilizando los siguientes bucles de inicialización de la matriz. Utilice el esquema de reemplazo LRU, y asuma que el marco de página 1 contiene al proceso, y los otros dos están inicialmente vacíos. Recuerde que el tamaño de un char es de 1B.

- i. 

```
for (char j = 0; j < 100; j++)
  for (char i = 0; i < 100; i++)
    A[i][j] = 0;
```
- ii. 

```
for (char i = 0; i < 100; i++)
  for (char j = 0; j < 100; j++)
    A[i][j] = 0;
```

- b) Considere la siguiente secuencia de referencias a páginas:

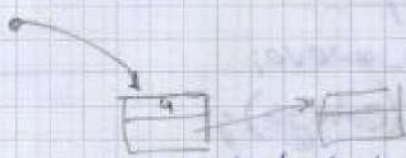
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

¿Cuántos fallos de página ocurrirían para los siguientes algoritmos de reemplazo? Asuma que se tiene tres marcos y que todos están inicialmente vacíos.

- LRU.
- FIFO.

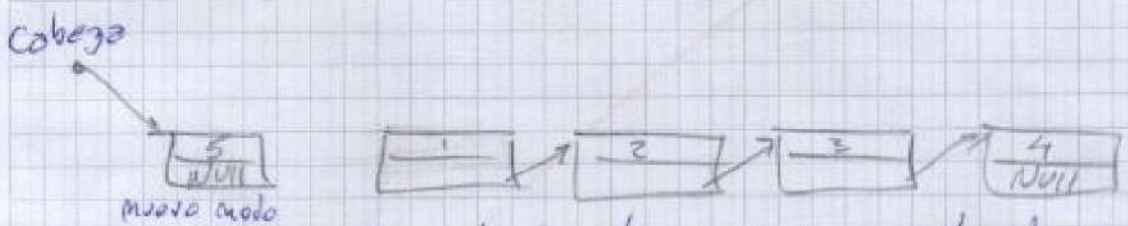
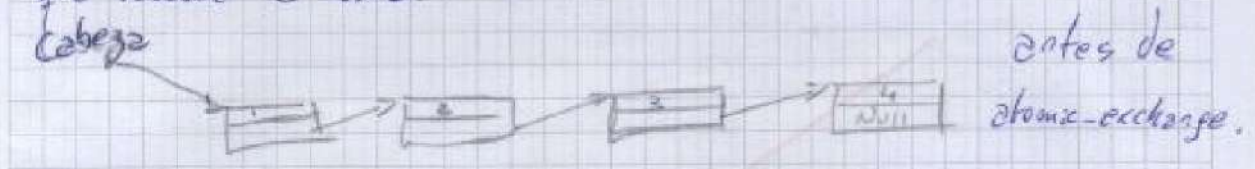


**Ej1** atomic global cabeza



```
void insertaradelante(int numero_nuevo) {
    Nodo* nuevo = New Nodo();
    nuevo->Numero = numero_nuevo;
    nuevo->siguiente = atomic-Exchange (cabeza, nuevo);
}
```

Imaginemos una situación en la que un proceso usa la función insertaradelante, hace el atomic-Exchange pero cuando está por asignar el resultado de ese exchange a nuevo->siguiente el scheduler lo de saca y pone a correr un proceso que recorre la lista.



Después de atomic-exchange antes de asignar nuevo->siguiente.

El proceso que recorre la lista la verá como con 1 solo elemento cuando debería tener 5. (o 4) ✓

CUALQUIER SERIALIZACIÓN POSIBLE (PERO NO ES SERIALIZACIÓN)

b) Solución usando variables atómicas.

void insertaradelante (int numero\_nuevo) {

  Nodo \* nuevo = new Nodo();

  nuevo->numero = numero\_nuevo;

  nuevo->sig = atomic\_load (cabeza);

  while (true) {

    if (atomic\_compare\_exchange (cabeza, nuevo->sig, nuevo)) {

      break;

    } Else {

      nuevo->sig = atomic\_load (cabeza);

NO  
HACE  
FALTA

  }

}

}

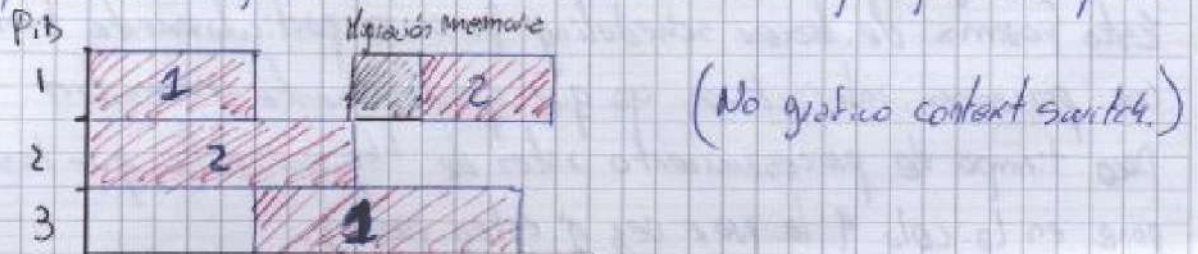


## Ejercicio 2

A-Concepto afinidad al procesador en scheduling.

El concepto surge de la situación de un sistema multiprocesador en que varios procesos corren simultáneamente.

Imaginemos un sistema con 2 procesadores (1 y 2) y 3 procesos A, B y C. El proceso A se está ejecutando en el procesador 1 y el B en el 2. Hagamos el sip diagrama de gantt:



Podemos ver que el proceso 1 estaba corriendo en el procesador 1 y después de ser desalojado volvió a correr en el procesador 2. Sin embargo como cada procesador tiene una parte de

memoria designada todo que ejecutarse una migración de memoria perdiendo tiempo de procesador. La migración copia toda la memoria del proceso 1 que está en la memoria del procesador 1 al procesador 2.

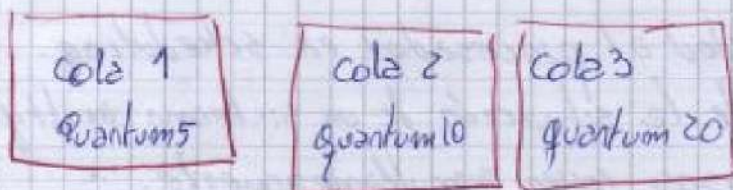
LA MEMORIA ES COMPARTIDA

NO. LA CACHE! (MEMORIAS DE LOS CPUS) NO SE COPIAN PERO SE PERDEN POSIBLES HITS

• la ventaja es que los procesadores pueden acceder simultáneamente a sus sectores de memoria sin tener que esperarse entre si. ~~SI~~ NO



6] La relación suele ser inversamente proporcional,  
ejemplo:



La cola 1 es la de mayor prioridad y tiene menor quantum si  
al momento de elegir el próximo proceso a ejecutar tiene algo  
se va a ejecutar ese.

Esta forma de hacer scheduling funciona particularmente bien  
con procesos interactivos ya que generalmente necesitan  
poco tiempo de procesamiento antes de bloquearse. (lo que los  
pone en la cola 1 la prox vez q' esten ready).

Al contrario penaliza a los procesos intensivos en CPU  
ya que si con el primer quantum de 5 no les alcanza para  
terminar los desaloja para ponerlos en la cola 2 y lo mismo  
de la cola 2 a la 3, teniendo que esperar a los procesos que  
si se bloquean con accesos a E/S.

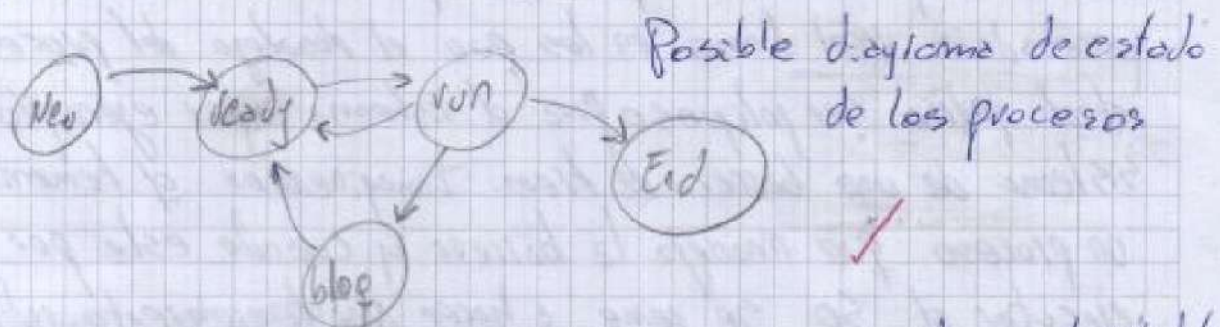
Y CÓMO SE RELACIONA  
CON TIEMPO DE RESPUESTA  
Y THROUGHPUT?

7] La diferencia entre un scheduler con y sin desalojo es que  
si hay desalojo el SO puede vetar el control del  
procesador por la fuerza por ejemplo si a un proceso  
se le acaba el quantum (cuando le interrumpe el clock). En cambio  
en uno sin desalojo una vez que el SO entrega el control del  
procesador a un proceso tiene q' esperar a q' se lo devuelva  
por ej terminando o con una system call. ✓



## Scheduler con desalojo:

Tenemos un proceso principal (scheduler) que va a elegir quien usa el procesador y por cuanto tiempo (quantum).



Cuando llega un nuevo proceso esta en estado ~~ready~~ y ~~se manda~~ el scheduler lo anota en su tabla de procesos listos para correr.

- Cuando el sched lo pone a correr este pasa al estado run hasta que se le acaba el quantum o el proceso hace una system call o se bloquea. ✓
- Del estado bloqueado pasa nuevamente al estado ready cuando se desbloquea (por ej: termino de esperar E/s). ✓
- Si esta corriendo el proceso puede terminar y entonces el so limpia su memoria y cualquier otra parte del sistema que tenga asignada. ✓

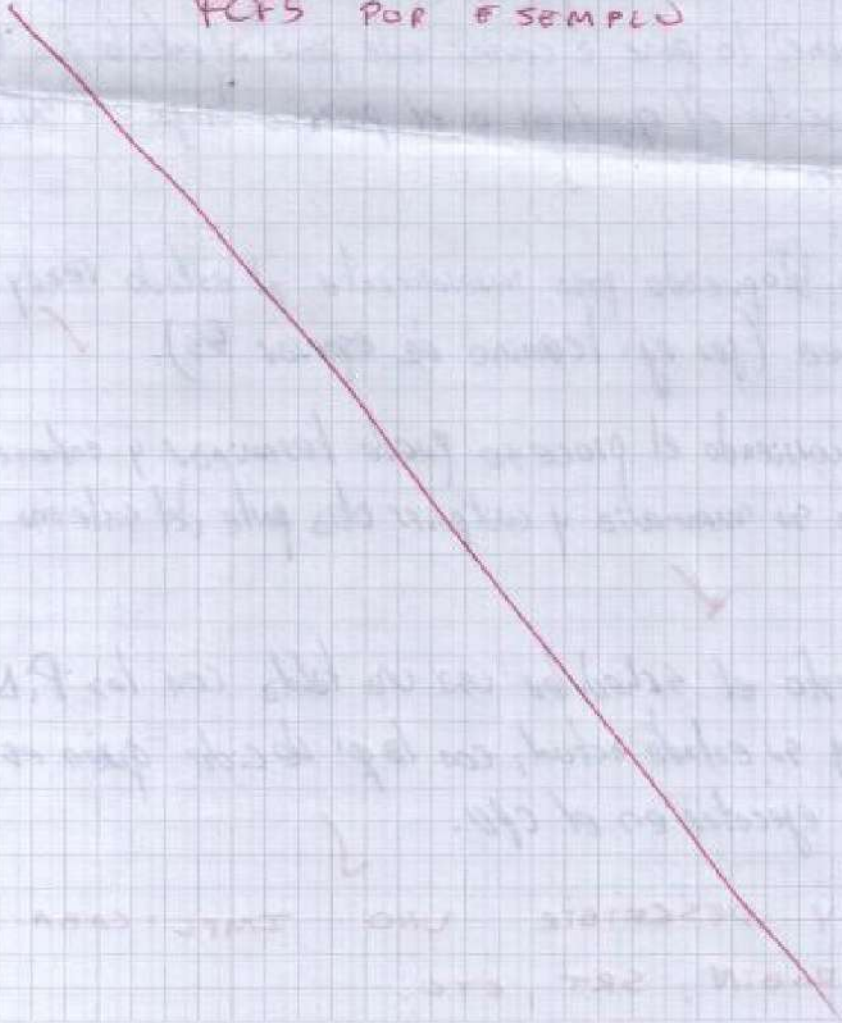
Para todo esto el scheduler usa una tabla con los PIDs de los procesos y su estado actual, con la que decide quien es el proximo a ejecutar en el CPU. ✓

ELEGIR Y DESCRIBIR UNO IMPLICABA  
ROUND ROBIN, SRT, ETC.



En los schedulers sin de retraso, el SO no puede retornar el control del CPU por la fuerza, esto presenta problemas si por ejemplo quiere hacer tareas de mantenimiento urgente o si tiene algun proceso con prioridad critica que no esta ejecutando. No son muy comunes hoy en dia y se usan sobretodo en procesadores real time en los que el retraso del proceso actual podria ser peligroso para el sistema, por ejemplo un sistema de una barrera de fren. Imaginemos q tenemos un proceso que maneja la barrera y cuando este por ejecutar el SO se pone a hacer mantenimiento, podria pasar que el SO tarde tanto q la barrera se baje y pese el tren.

IDEM, EXPLICAR  
FCFS POR EJEMPLO





**Ejercicio 3**

Grupo Conexo, 10 vertices

Void pintarme (int color)

bool son Vecinos (int numThread A, numThread B)

Void pintar (int \* colores) {

int pipes [10][2];

for (int a=0; a<10; a++) {

pipe [a] = pipe ();

}  
PIPE ( pipes [a] )

} creo 1 pipe por modo.

for (int i=0; i<10; i++) {

int pid = fork ();

if (pid == 0) {

if (i == 0) {

for (int j=1; j<10; j++) {

if (son Vecinos (i,j) {

pipe [j][1].write (colores);

}  
pintame (colores [i]);

}  
} Else {

int colores2[10] = pipe [i][0].read ();

for (int j=1; j<10; j++) {

if (son Vecinos (i,j) {

pipe [j][1].write (colores2);

}  
}

} uso un ciclo para forkear 10 veces  
y uso i como identificador de proceso  
y modo a pintar.

} Si soy proceso 0  
le comparto los  
colores a todos  
mis vecinos.

} Si no soy  
el proceso 0  
espero q alguien  
me comparta  
y luego comparto.

pintame (colores2 [i]);

}  
break;

} me pinto y salgo del  
for, para no crear mas  
procesos.



```

for (int b=0; b<10; b++) {
    pipe[b][0].close();
    pipe[b][1].close();
}
}

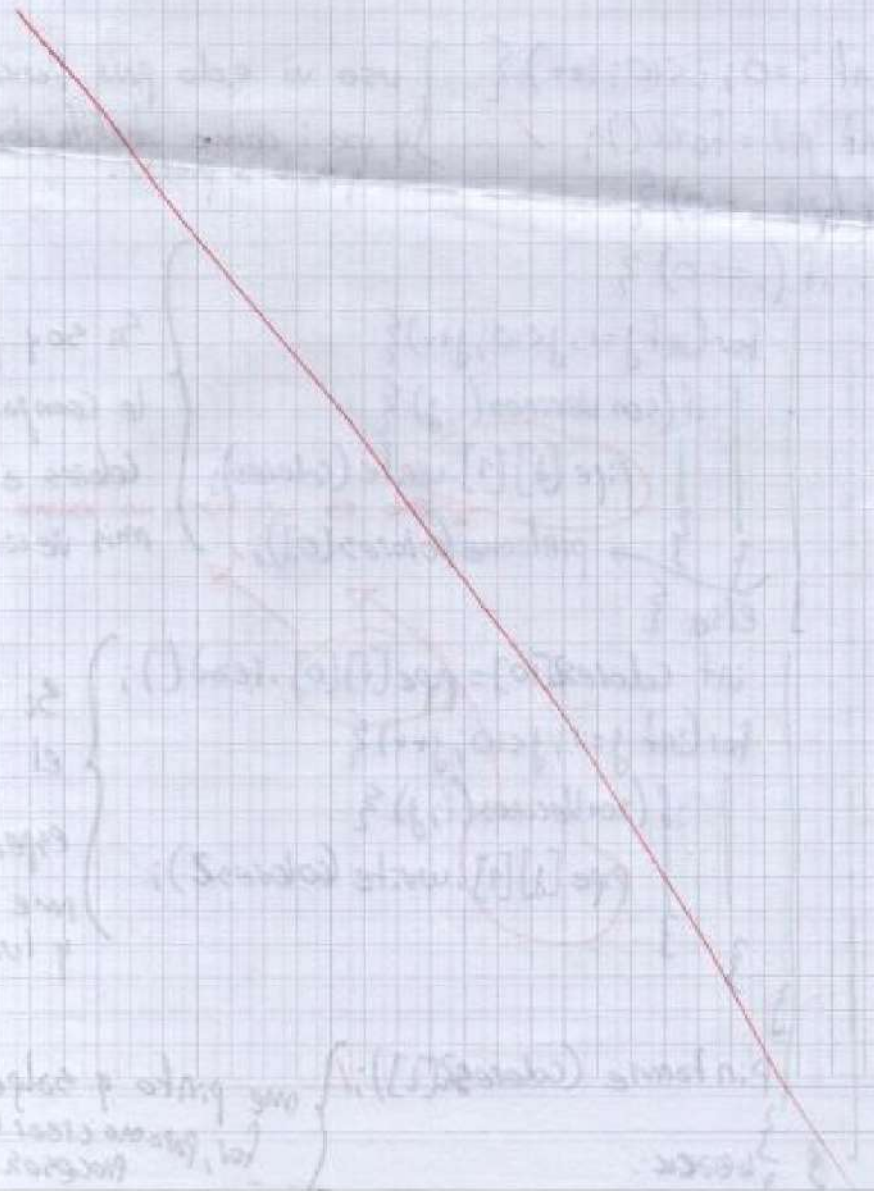
```

Cierro todos los pipes

X EL PADRE TAMBIEN DEBERIA CERRARLOS.

NO EXISTE OPERACIONES SCORE ENT

Podría esperar el padre a q' terminen los hijos pero como el enunciado no aclara nada no lo considere necesario. ✓





### Ejercicio 4

1200 páginas 200B. código en página 0.  
tres tercios de página.

a)  $A[0][0]$  dirección memoria 200

```
i) for(char j=0; j<100; j++)  
  [ for(char i=0; i<100; i++)  
    |   A[i][j]=0;
```

Como lo primer coordenado es la fila y la  
segunda es la columna y vemos en el  
práctico que en un 0 página entran 2 filas.

Tenemos un fallo de página cada 2 asignaciones de la matriz por  
que la estamos recorriendo por filas y la que vamos a acceder no  
la habíamos cargado hasta el momento. Cuando llegamos al final  
de las filas volvemos a la primera y pasamos a la segunda  
columna que ya la habíamos descargado.

En total  $\frac{10000 \text{ asignaciones}}{2} = 5000$  fallos de página.

```
ii) for(char i=0; i<100; i++)  
     for(char j=0; j<100; j++)  
       A[i][j]=0
```

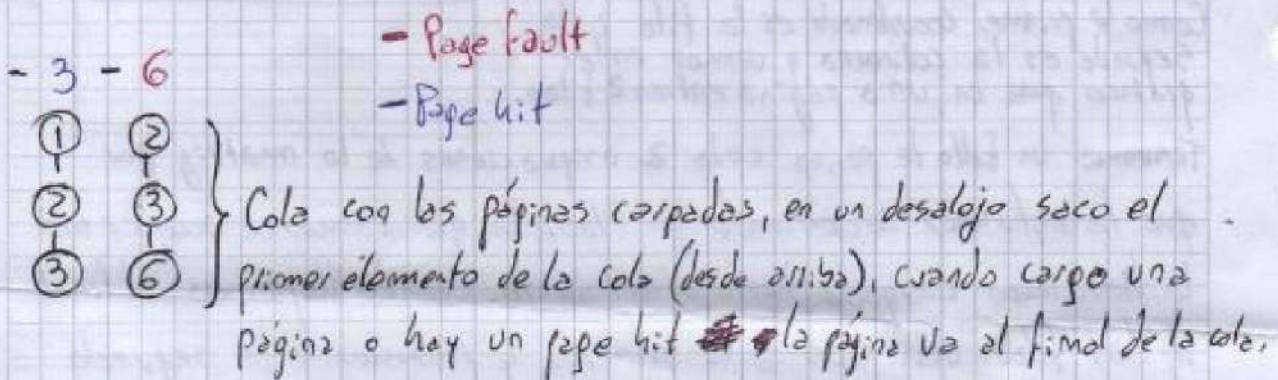
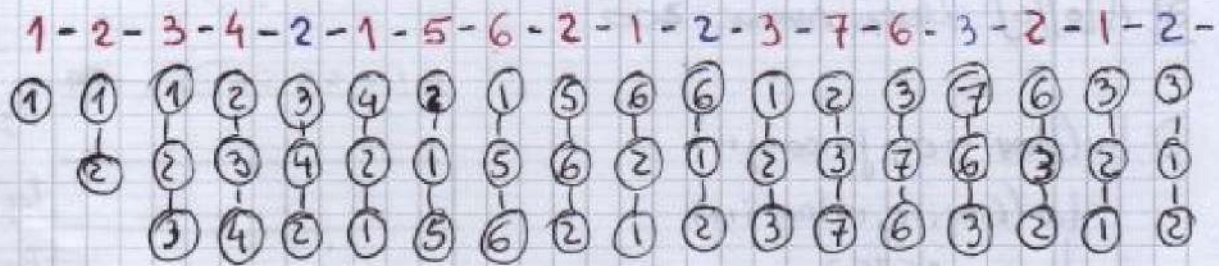
En este caso estamos recorriendo la matriz en el orden que  
está en memoria entonces tenemos 1 fallo de página cada  
200 asignaciones (la cont de chars es 1 página).

$\frac{10000 \text{ asign}}{200} = 50$  fallos de página.



6] Secuencia referencia páginas: (3 marcos de página)  
 1-2-3-4-2-1-5-6-2-1-2-3-7-6-3-2-1-2-3-6

Usando LRU (least recently used):



Con LRU hubo 15 fallos de página. ✓

Usando FIFO (first in first out):



- 3 - 6 Con FIFO hubo 16 fallos de página. ✓

