EJ 4) Atomicidad linea a linea a[] compartido, i y j locales.

a) Mostrar escenario de ejecucion donde el arreglo termine a ={1,0,1,0, …., 1,0}

b) Agregando semaforos con condicionales sobre i, j o sin ellos obtener un resultado final deterministico como el anterior.

```
                    PRE: a[n] = {2,2,2, .....,2}

P0: i = 0                |P1:j=0;
    while (i<N){         |    while(j<){
        a[i]=0;          |        a[j] = 1;
        i++;             |        j++;
    }                    |    }
```

a)

P0	P1

a: i = 0;
b: while(i<N){
c: a[i] = 0;
d: i++
e: }
	A: j = 0;
	B: while(j<N){
	C: a[j] = 1;
	D: j++
	E: }

Escenario (N=4)	valor final de a[]
abcde 2(ABCDE) 2(abcde) 2(ABCDE) abcde	a = {1, 0, 1, 0}

abcd
a = 0 i = 1
ABCDE
ABCDE
a = 1 1 i = 2
abcde
abcde
a = 1 0 0 i = 3
ABCDE
ABCDE
a = 1 0 1 1 i = 4
abcde

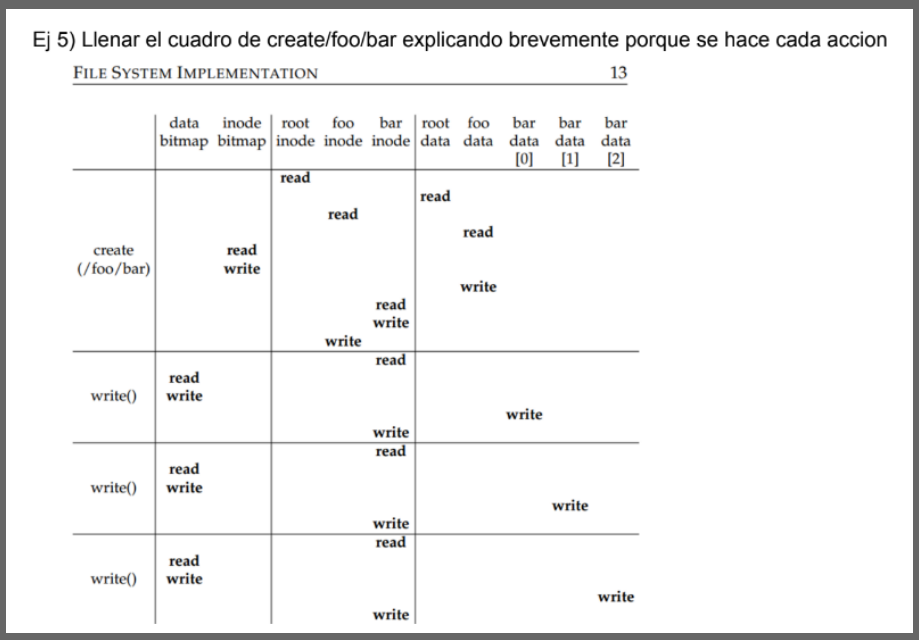
a = 1 0 1 0

b)

P0	P1

a: i = 0;
b: while(i<N){
if(i mod 2 != 0){
a[i] = 0;
}
d: i++
e: }
	A: j = 0;
	B: while(j<N){
	if(j mod 2 == 0){
	a[j] = 1;
	}
	D: j++
	E: }

Ej 5) Llenar el cuadro de create/foo/bar explicando brevemente porque se hace cada accion



FILE SYSTEM IMPLEMENTATION 13

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
			read			read				
				read			read			
create (/foo/bar)		read write								
							write			
					read write					
				write						
					read					
write()	read write							write		
					write					
					read					
write()	read write								write	
					write					
					read					
write()	read write									write
					write					

1) Leer root inode y root data buscando la entrada de foo
2) Leer foo inode y foo data buscando la entrada de bar
3) Leer el *inode bitmap* para ver si existe.
4) Si no existe:
    a) Escribir el *inode bitmap* para marcar el inodo como asignado.
    b) Agregar/linkear la entrada de bar en el directorio padre foo.
    c) Leer y escribir el inodo del archivo nuevo.
    d) Actualizar el inodo del directorio padre.
5) Si existe:
    a) Actualizar el inodo de bar con los nuevos cambios.

Leyendo un archivo:

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**			**read**				
				read						
							read			
					read					
read()								**read**		
					read					
read()									**read**	
					read					
read()										**read**
					write					

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

- Se comienza leyendo el inodo de la raíz (siempre).
- Se busca en la data de la raíz la entrada de foo. Se guarda el inodo encontrado.
- Se lee el inodo de foo para checar su metadata.
- Se ve la data de foo buscando al entrada de bar. Se guarda el inodo encontrado.
- Para efectuar la lectura siempre es (read):
    - Primero se consulta el inodo del archivo
    - Se lee el bloque
    - Finalmente se actualiza el offset y el último acceso al inodo con un write para este último.

Escribiendo un archivo:

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read	read	read write	read	read	write		
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

Figure 40.4: **File Creation Timeline (Time Increasing Downward)**

- Debemos abrir el archivo como arriba, luego efectuar un write.
- Para efectuar la escritura siempre es (write):
    - Lee el inodo del archivo por permisos
    - Se lee la data bitmap (Para marcar un nuevo bloque en uso)
    - Se escribe la data bitmap (agregando un bloque)
    - Se escribe el inodo (actualizando con el nuevo bloque)
    - Finalmente se escribe el bloque agregado.

Ej 6) En un filesystem de tipo UNIX con 12 bloques Directos, 1 bloque indirecto, 1 bloque doble indirecto y un bloque triple indirecto. Cada bloque es de 4 KiB y los indices de bloques son de 32 bits.
Dos posiblidades para formatear el disco:

a) Tabla de inodos de 1024 entradas
b) Tabla de inodos de 2^20 entradas

Completar la siguiente tabla con KiB, MiB, GiB o TiB segun corresponda, suponiendo que cada entrada de la inode table ocupa 128 bytes y que la data region esta al maximo posible usando todos los indices.

Parte del formato de disco/ Opcion	1024 i-nodos	2^20 i-nodos
i-bmap		
d-bmap		
inode table		
data region		

Parte del formato de disco/Opcion	1024 i-nodos	2^20 i-nodos
i-bmap	128 bytes ≡ 0.00012 MiB	131072 bytes ≡ 0.125 MiB
d-bmap	512 MiB	512 MiB
inode table	128 KiB	128 MiB
data region	16 TiB	16 TiB

1024 i-nodos:

- i-bmap = 1024 bits / 8 = 128 bytes →
- d-bmap = 2^32 bits / 8 = 512MiB
- inode table = 128 bytes * 1024 = 128 KiB
- data región = 2^32 * 4 KiB = 2^32 * 2^12 = 16 TiB

2^20 i-nodos:

- i-bmap = 2^20 bits / 8 = 131072 bytes
- d-bmap = 2^32 bits / 8 = 512MiB
- inode table = 128 bytes * 2^20 = 2^7 * 2^20 = 128 MiB
- data región = 2^32 * 4 KiB = 2^32 * 2^12 = 16 TiB