Final 7/12/2023 Sistemas Operativos

1) Describir que hace el programa, cuantos padres e hijos tiene, etc. Se ejecuta con: ./a.out ./a.out ./a.out ./a.out ./a.out ./a.out ./a.out (7 ./a.out)

```
int main(int argc, char **argv) {
    char buff[L] = {'\0'};
    if (argc < 2){
        return 0;
    }
    int rc = fork();
    if (rc < 0){
        return -1;
    }
    if(rc == 0){
        close(1);
        open(argv[0], 0);
        read(1, buff, L);
        close(0);
        open(argv[0], O-TRUNCATE| O-WRONLY);
        write(0, buff, L);
    }else{
        execvp(argv[1], argv+1)
    }
}
```

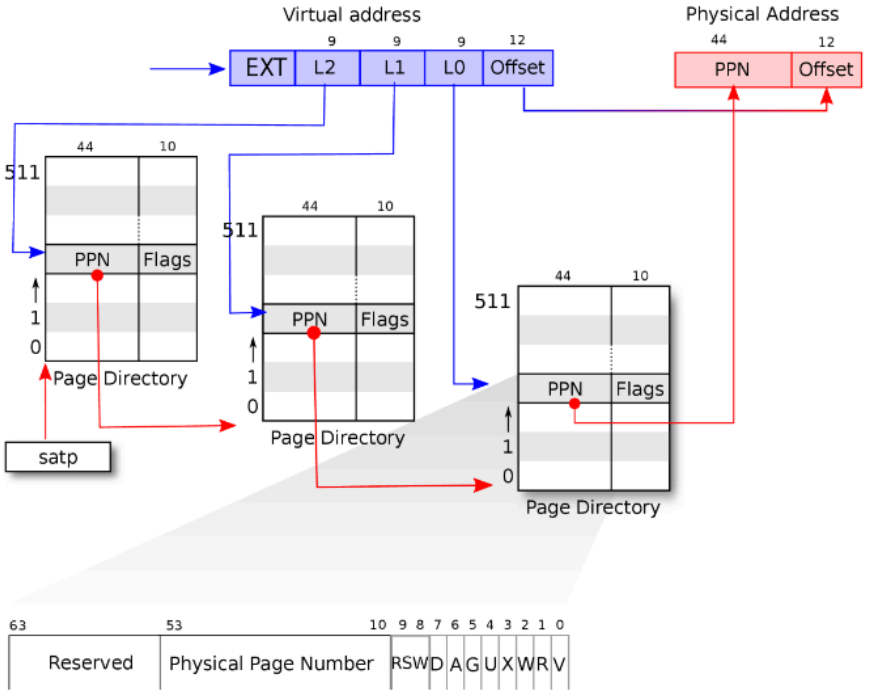
2) Tenemos un esquema RISC-V, de 3 Niveles osea (9,9,9,12) -> (44,12) con paginas de 4KiB (no copie las tablas xd)

a) Traducir de virtual a fisica direcciones de memoria(No las copie xd)

b) Traducir de fisica a todas las virtuales (Tampoco las copie xd)

{correccion, encontre los ejercicios, son iguales a los del parcial 1 de 2022}

Tenemos un esquema de paginación RISC-V con páginas de 4 KiB de 3 niveles con formato 9,9,9,12 -> 44,12 como muestra la figura.



Bits de control

V: *válido*

R: se puede leer, *readable*

W: se puede escribir, *writable*

X: se puede ejecutar, *executable*

Supongamos que tenemos el registro de paginación apuntando al marco físico `satp=0x00000000FE0`.

0x00000000FE0	0x00000000FEA	0x000000AD0BE
`0x1FF: 0x00000000000, ----`	`0x1FF: 0x00000000000, ----`	`0x1FF: 0x00000000000, ----`
⋮	⋮	⋮
`0x004: 0x00000000000, ----`	`0x004: 0x00000000000, ----`	`0x004: 0x00000000000, ----`
`0x003: 0x00000000000, ----`	`0x003: 0x00000000000, ----`	`0x003: 0x00000D1AB10, XWR-`
`0x002: 0x00000000FEA, XWRV`	`0x002: 0x000000AD0BE, XWRV`	`0x002: 0x00000DECADA, -WRV`
`0x001: 0x00000000FEA, XWRV`	`0x001: 0x000000AD0BE, XWRV`	`0x001: 0x000CAFECAFE, ----`
`0x000: 0x00000000FEA, XWRV`	`0x000: 0x000000AD0BE, XWRV`	`0x000: 0x0000000ABAD, X--V`

a) Traducir de **virtual a física** las direcciones:

`0x0000`

`0x1000`

`0x2000`

`0x3000`

---

b) Traducir la direccion fisica 0xDECADA980 a TODAS LAS VIRTUALES que la apuntan.

3) Codigo assembler risc-v que es la suma prefijo en el array.
El array a esta en ELF.bss que empieza en 0x2FC0, y termina en 0X38008 EXCLUSIVO.
El arreglo tiene 9 elementos que ocupan 8 bytes cada uno

Consigna: Usar el esquema del ej2, escribir la traza de memoria fisica completa incluyendo instrucciones y datos

El siguiente código de máquina y su desensamblado RISC-V **computa la suma prefijo en el mismo arreglo** (*in-place prefix sum*). El arreglo `a` está en el segmento ELF `.bss` y empieza en `0x2FC0` y termina en `0x3008` **exclusive**. Como sus elementos son `unsigned long`, cada uno ocupa 8 bytes y por lo tanto tiene 9 elementos.

```
0000000000000634 <main>:
 634: 0613                  li    a2,0x3008   # <__BSS_END__> &a[9]
 636: b206                  li    a5,0x2FC8   # <a+0x8> &a[1]
 638: 6398                  ld    a4,0(a5)    # a4 = a[i]
 63a: ff87b683              ld    a3,-8(a5)   # a3 = a[i-1]
 63e: 9736                  add   a4,a4,a3
 640: e398                  sd    a4,0(a5)    # a[i] = a4
 642: 07a1                  addi  a5,a5,8     # "i++"
 644: fec79ae3              bne   a5,a2,0x638 # <main+0x10>, "i<9"
 648: 8082                  ret
```

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++;
    }                    |    }
```

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

	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 write	read	read write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

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		