UNIVERSIDAD DE MURCIA

FACULTAD DE INFORMÁTICA

Ampliación de Sistemas Operativos

Profesor:

Antonio Gil Flores

Alumnos:

Martín Piñas Ayala
Mario Antelo Ribera

Curso 2017/2018 **Convocatoria:** Enero

ÍNDICE

Boletín 7: Llamadas al Sistema en xv6....4
Ejercicio 1:....5
Ejercicio 2:....7
Pregunta 1....7
Pregunta 2....8
Pregunta 3....8
Ejercicio 3:....9
Ejercicio 4:....11
Boletín 8: Reserva de páginas bajo demanda en xv6....13
Ejercicio 1:....14
Ejercicio 2....17
Valores negativos....17
Fallo de página por debajo de la pila....18
Comprobación modificaciones....19
Boletín 9: Ficheros grandes en xv6....20
Ejercicio 1....21

Boletín 7: Llamadas al Sistema en xv6

Ejercicio 1:

¿Cuál es la rutina de servicio asociada a la interrupción 64?(Nota: Utiliza el depurador para ver cómo se inicializa la IDT y que manejador se asocia al trap de las llamadas al sistema)

Interrupción: cuando un dispositivo genera una señal para indicar que necesita atención por parte del sistema operativo.

La Arquitectura x86 permite 256 interrupciones diferentes:

- 0-31: interrupciones de software (errores de división, acceso a direcciones de memoria no validas...)
- 32-63: interrupciones de hardware.
- **64: interrupción de llamada al sistema (syscall)**

Desde **main.c** se llama a la función “tvinit”:

```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

La función “tvinit” se encuentra implementada en el fichero **trap.c**. En dicha función se establecen las 256 entradas en la tabla IDT. Cada interrupción “i” es manejada por el código de la dirección de “vectores[i]”. Cada punto de entrada es diferente, porque en x86 no proporciona el número de trap al manejador de interrupción. La única manera de distinguir los 256 casos es con 256 manejadores diferentes.

```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```

Tvinit maneja T_SYSCALL, que es el trap de la llamada al sistema para el usuario.

EL kernel establece el privilegio de la llamada al sistema a DPL_USER, el cuál permite a un programa de usuario un trap con una instrucción “int”. XV& no permite que los procesos generen otras interrupciones con int.

Cada manejador establece un marco de interrupción, que luego llama a la función “trap”, la cual se encuentra implementada en el fichero **trap.c.**

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

Trap examina el número de interrupciones “tf →trapno” para decidir porqué se ha llamado y qué se debe hacer. Si el trap es T_SYSCALL, trap llama a la función “syscall”, que se encuentra implementada en el fichero **syscall.c.**

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

Ejercicio 2:

Pregunta 1.

¿En que punto del núcleo entra el kernel cuando se realiza una llamada al sistema?

Las llamadas al sistema se realizan con la función SYSCALL, que se encuentran definidas en el fichero **usys.S.** En este caso, se nos pide probar con el comando “ls”, el cuál, realiza la llamada al sistema a través de “open”.

```
void
ls(char *path)
{
  char buf[512], *p;
  int fd;
  struct dirent de;
  struct stat st;

  if((fd = open(path, 0)) < 0){
    printf(2, "ls: cannot open %s\n", path);
    return;
  }
```

Las llamadas al sistema se encuentran definidas en **usys.c.**

```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
```

Se llama al vector correspondiente, que se encuentra en **vector.S.** y es aquí cuando se produce la entrada en el núcleo del kernel.

Pregunta 2.

¿Qué llamada al sistema se está ejecutando?

La llamada al sistema que se está ejecutando es SYS_open, que se encuentra en **syscall.c.**

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
};
```

Pregunta 3.

¿Qué se ha guardado en pila kernel del proceso desde que se ejecutó int 64 hasta que se llama a la función trap() implementada en el fichero trap.c?

Los registros:

```
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal
```

Se encuentran en **trapasm.S.**

Ejercicio 3:

Añade una nueva llamada al sistema (int date(struct rtcdate* d)) a xv6.

Para añadir una nueva llamada al sistema en xv6 tenemos que modificar los ficheros **syscall.h, syscall.c, usys.S, user.h, sysproc.c** y **sysfile.c.**

- **syscall.h** Asignamos un valor entero que represente la nueva llamada al sistema.

```
#define SYS_date    22
```

- **syscall.c** Se declara el método que se va a usar para la nueva llamada al sistema.

```
extern int sys_date(void);
```

```
[SYS_date]    sys_date,
```

- **usys.S** Declaramos macros para que el trap reconozca que es una llamada al sistema.

```
SYSCALL(date)
```

- **user.h** Declaración de la función para que desde el modo usuario pueda ser accesible.

```
int date(struct rtcdate*);
```

- **sysproc.c** Implementación de la funcionalidad de la llamada al sistema “date”.

```
//Boletin 7. Llamada al sistema date
int
sys_date(void)
{
  struct rtcdate *date;
  /*
   La funcion argptr obtiene el argumento de la llamada al sistema
   y comprueba que este argumento es un puntero del espacio de
   usuario valido.
  */
  int ok = argptr(0, (void*)&date, sizeof(struct rtcdate *));
  if (ok < 0)
    return -1;

  cmostime(date);
  return 0;
}
```

- Para finalizar, agregamos a **Makefile** el nuevo comando:

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _date\
```

Programa de prueba

```c
#include "types.h"
#include "user.h"
#include "date.h"

int main (int argc, char* argv [])
{
    struct rtcdate r;

    if (date(&r)){
        printf (2, "date failed\n");
        exit();
    }
    printf(1, "Date: %d:%d:%d    %d/%d/%d\n", r.hour, r.minute, r.second, r.day, r.month, r.year);
    exit();

}
```

Después de probar la nueva llamada al sistema que hemos implementado, podemos cerciorar que funciona y muestra la fecha tal como la hemos programado. Un ejemplo de la salida mostrada:

```
$ date
Date: 15:46:17    19/12/2017
```

Ejercicio 4:

Implementa la llamada al sistema dup2() y modifica el shell para usarla.

Para añadir la llamada al sistema de dup2, realizamos los mismos pasos que para la llamada “date”, excepto que la implementación del “dup2” se realiza en **sysfile.c.**

- **syscall.h**

```
#define SYS_dup2    23
```

- **syscall.c**

```
extern int sys_dup2(void);
```

- **usys.S**

```
SYSCALL(dup2)
```

- **user.h**

```
int dup2(int, int);
```

- **sysfile.c**

Para no liar demasiado la memoria con el código del dup2, el cual está en el archivo **sysfile.c** pasamos a describir las condiciones o pasos que seguimos:

1. Comprobamos si existe el primer descriptor pasado por parámetro.
2. Se comprueba que "newfd" es un indice de tabla de archivos válido.
3. Comprobamos que el nuevo descriptor es válido.
4. Si el nuevo descriptor "newfd" no tiene asociado ninguna estructura de tipo file es que "newfd" esta vacío, por lo que podemos duplicar de forma segura el descriptor en el nuevo.
5. Si el descriptor de fichero nuevo ya existe, se recupera el puntero a memoria de ese descriptor.
6. Comprobamos si el puntero de "fd1" es igual a "fd2", por lo que, en caso de ser cierto, no habría que hacer el duplicado, tan solo devolver el numero del descriptor.
7. Decrementamos contador ref de "fd2".
8. Realizamos el duplicado del descriptor de fichero.
9. Incrementamos el valor del contador ref de “fd1”.

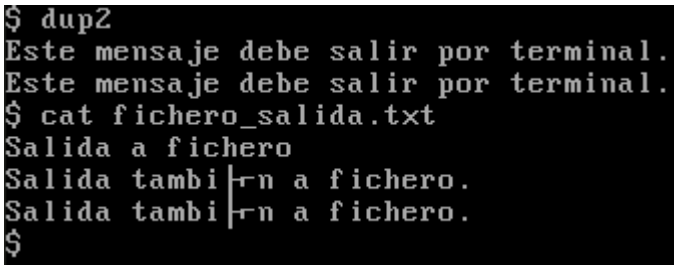
- **Makefile**

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _date\
    _dup2\
```

Comprobación dup2

Usando el programa de prueba **dup2test.c** al cuál le asignamos el nombre de **dup2.c** por comodidad a la hora de hacer las pruebas. Podemos comprobar que la salida es la esperada, tuvimos que cambiar las condiciones para que capture el error del dup2, es decir, que imprima el mensaje.

En la siguiente imagen mostramos el resultado esperado de la prueba.



Boletín 8: Reserva de páginas bajo demanda en xv6

Ejercicio 1:

Implementa esta característica de reserva diferida en xv6

Para la resolución del ejercicio necesitamos acceder a **sysproc.c, trap.c y vm.c.**

1. **sysproc.c**

Tenemos que eliminar la reserva de páginas de la llamada al sistema sbrk, la cuál esta implementada a través de la función “sys_sbrk()”. Para ello borramos la parte del método “growproc()”. A continuación aumentaremos el tamaño del proceso (myproc()→sz) de forma manual y devolveremos el tamaño antiguo, por lo que la reserva de memoria se hará para un tamaño menor del que le hemos puesto al proceso, por lo que se producirá un fallo de página cada vez que el método que ejecutamos necesite memoria física.

```
//Boletín 8. Reserva de páginas.
int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;

/* Incrementamos el tamaño del proceso(myproc->sz)
   Devolvemos el tamaño antiguo, pero no reservamos memoria
*/

  addr = myproc()->sz;
  myproc()->sz += n;

 /*
  Borramos la llamada a growproc()

  if(growproc(n) < 0)
    return -1;
 */
  return addr;
}
```

2. **Comprobación del fallo**

```
$ echo hola
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x1303 addr 0x4004--kill proc
```

3. **Controlar el fallo de pagina**

Como el kernel de xv6 no sabe cómo manejar un fallo de página (trap 14 o T_PGFLT) tendremos que añadir en el fichero **trap.c**, que es el que controla los fallos en el sistema, un caso especifico para dicho fallo. En dicho caso, lo que vamos a hacer es asignar páginas físicas al proceso solamente cuando éste las solicite, y esto se va a producir justamente en los fallos de página. Para ello vamos a hacer uso de:

- Kalloc → obtenemos un puntero a una página física libre.
- Memset → inicializamos la página libre obtenida a 0 (evitando residuos que pudieran haber)
- Mappages → mapeamos el proceso que ha fallado a la página/s

```
case T_PGFLT:
  if(proc == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",tf->trapno, cpunum(), tf->eip, rcr2());
      panic("trap");
  }

  //Redondear la direccion virtual a limite de pagina
  //rcr2 () es la dirección que causa el error de página
  addr=PGROUNDDOWN(rcr2());

  //Busqueda pagina libre
  char *mem = kalloc();

  //Comprobar si falla (matar al proceso)
  if(mem == 0){
    cprintf("Error kalloc\n");
    proc->killed = 1;
  }else{
  //Inicializacion de la pagina
  memset(mem, 0, PGSIZE);

  //Mapeo
  if(mappages(proc->pgdir, (char*)addr, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("Error mappages\n");
      kfree(mem);
      proc->killed = 1;
  }
 }
 break;
```

Nota: tenemos que borrar la declaración de función estática en el fichero **vm.c** y declarar mappages() en el fichero **trap.c**

```c
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"

// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uint vectors[];  // in vectors.S: array of 256 entry pointer
struct spinlock tickslock;
uint ticks;

extern int mappages(pde_t *, void *, uint, uint, int);
```

4. **Comprobación de que funciona.**

Ejecutamos el mismo comando anterior.

Ejercicio 2.

Modifica el código del primer ejercicio para que contemple las dos primeras situaciones descritas a continuación.

Valores negativos

Vamos a modificar la implementación de los siguientes métodos:

- sys_sbrk() en sysproc.
- deallocuvm_Nuevo() en vm.c

en el método sys_sbrk si reducimos el tamaño del proceso(Valores negativos), entonces vamos a liberar las paginas con respecto al tamaño reducido. El código queda de la siguiente manera:

```
int
sys_sbrk(void)
{
  int addr;
  int n;

  //cogemos el arg pasado a la función
  if(argint(0, &n) < 0)
    return -1;

  addr = myproc()->sz; // valor actual del tamaño del proceso
  myproc()->sz+=n; // aumentamos el tamaño del proceso

  //Controlamos la liberación de memoria
  if(n<0){
    deallocuvm_Nuevo(myproc()->pgdir, addr, myproc()->sz);
    switchuvm(myproc());
   }

  return addr;
}
```

Como podemos ver en el codigo cuando se recibe valores negativos entonces llamamos al método deallocuvm_Nuevo el cual libera las paginas que tienen memoria reservadas y las que no.

El codigo de deallocuvm_Nuevo es el siguiente:

```c
//Mejora de liberación de paginas cuando el valor pasado a sbrk es negativo
void deallocuvm_Nuevo(pde_t* pgdir, uint oldsz, uint newsz)
{
  pte_t *pte;
  uint a, pa;

  a = PGROUNDUP(newsz);
  for(; a  < oldsz; a += PGSIZE){
    pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte)
      a += (NPTENTRIES - 1) * PGSIZE;
    else if((*pte & PTE_P) != 0){
      pa = PTE_ADDR(*pte);
      if(pa == 0) // si la direccion solo ponemos la direccion de memoria
       fisica a 0
        *pte = 0;
      else{
        char *v = P2V(pa);
        kfree(v);
        *pte = 0;
      }
    }
  }
}
```

Con estas modificaciones liberamos la memoria del proceso al reducir su tamaño.

Fallo de página por debajo de la pila

Vamos a controlar el fallo de pagina que se produce al intentar acceder a la dirección de memoria de una pagina de guarda, esto ocurre cuando crece la pila.

Todo fallo de pagina lleva asociado un código de error descrito a continuación:

```
 31              4              0
+---+--  --+---+---+---+---+---+---+
|   Reserved   | I | R | U | W | P |
+---+--  --+---+---+---+---+---+---+
```

	Length	Name	Description
P	1 bit	Present	When set, the page fault was caused by a page-protection violation. When not set, it was caused by a non-present page.
W	1 bit	Write	When set, the page fault was caused by a page write. When not set, it was caused by a page read.
U	1 bit	User	When set, the page fault was caused while CPL = 3. This does not necessarily mean that the page fault was a privilege violation.
R	1 bit	Reserved write	When set, the page fault was caused by writing a 1 in a reserved field.
I	1 bit	Instruction Fetch	When set, the page fault was caused by an instruction fetch.

para controlar el fallo de pagina en una pagina de guarda vamos a tener una condición de que si el código de error que ha generado el fallo es 7 entonces se mata el proceso.

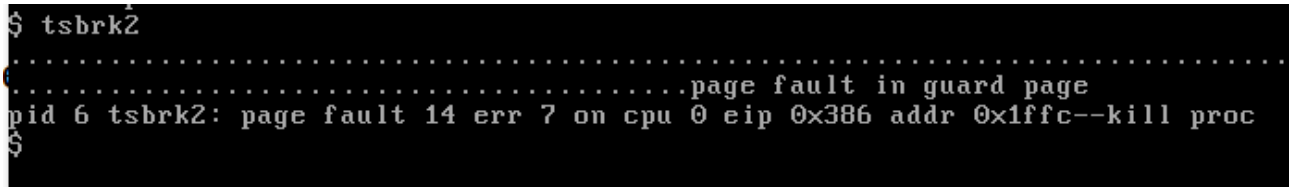
```c
if(tf->err == 7){
  cprintf("page fault in guard page\n");
  cprintf("pid %d %s: page fault %d err %d on cpu %d "
          "eip 0x%x addr 0x%x--kill proc\n",
          myproc()->pid, myproc()->name, tf->trapno,
          tf->err, cpuid(), tf->eip, rcr2());
  myproc()->killed = 1;
}else{
```

Comprobación modificaciones

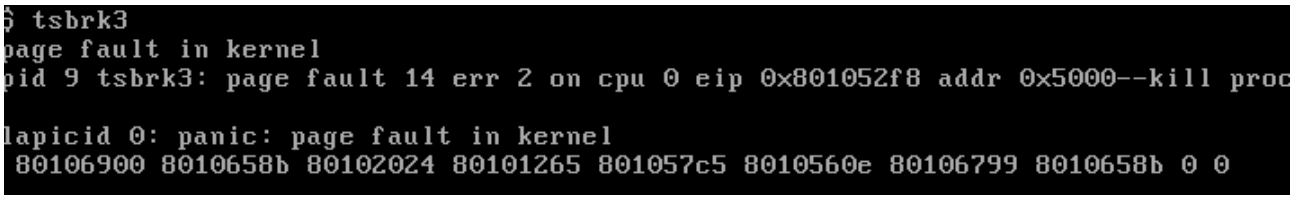
Se ha procedido a añadir al fichero Makefile la referencia a 3 ficheros de prueba: tsbrk1, tsbrk2 y tsbrk3.

El primero comprueba el funcionamiento correcto del crecimiento y decrecimiento del tamaño del proceso.

La segunda prueba desborda la pila y crea un fallo de página en la página de guarda.



La tercera prueba comprueba que el fallo de pagina ocurre en modo kernel, lo cual ocasiona que se mate el sistema.



Boletín 9: Ficheros grandes en xv6

Ejercicio 1

Modifica bmap() para que implemente un bloque doblemente indirecto. No se puede cambiar el tamaño del nodo-i en disco.

Para la resolución del ejercicio necesitamos acceder a varios archivos: **Makefile, param.h, big.c, fs.c y fs.h.**

- **Makefile**

```
QEMUEXTRA = -snapshot
```

Añadiendo QEMUEXTRA= -snapshot conseguiremos un aumento de la velocidad de qemu cuando se crean ficheros grandes en xv6.

- **Param.h**

```
#define FSSIZE       20000  // size of file system in blocks
```

Aumentamos el tamaño de bloques libres a 20,000 pues de inicio comienza con 1,000 bloques.

- **Big.c**

```
	_big\
```

Añadimos el archivo big para que se encuentre en UPROGS dentro de **Makefile.**

- **Fs.h**

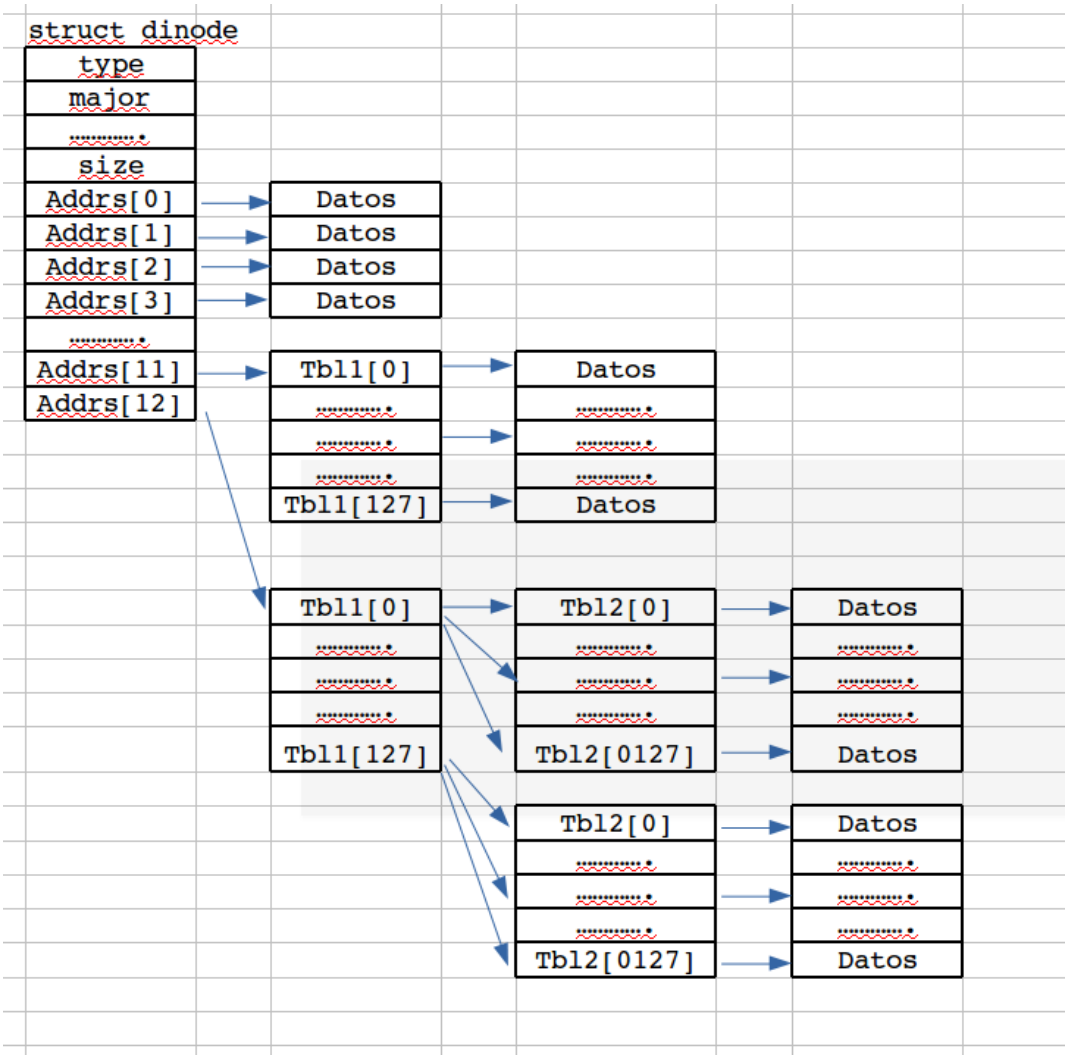
```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define NDOUBLE_INDIRECT (NINDIRECT * NINDIRECT)
#define MAXFILE (NDIRECT + NINDIRECT + NDOUBLE_INDIRECT)
```

En este archivo vamos a modificar las contantes utilizadas para indicar la cantidad de bloques en un fichero de disco, es decir, se modificar la cantidad de bloques que se podrán almacenar en una estructura inode.

```
// on-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+2];   // Data block addresses
  //sumamos 2 para los bloques simplemente indirecto y el doblemento indirecto
};
```

Todo archivo tiene un array de 13 posiciones para almacenar los bloques de datos, estas posiciones están repartidas de la siguiente manera:

- 11 bloques directos de datos. Addrs[0] al Addrs[10]
- 1 bloque simplemente indirectos Addrs[11]
- 1 bloque doblemente indirectos Addrs[12]



- **Fs.c**

vamos a modificar el método bmaps del fichero fs.c para que trate

```
//doblemente indirecto
bn -= NINDIRECT;
// double indirect
if(bn < NDOUBLE_INDIRECT){
    //cargamos los bloques doblemente indirectos, asignamos memoria si es
     necesario
    if((addr = ip->addrs[NDIRECT + 1]) == 0)
        ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);

    // seleccionamos el primer nivel los bloques doblemente indirectos
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;

    //calculamos la entrada  y el indice del primer nivel
    ind_n1 = bn / NINDIRECT;
    ind_n2 = bn % NINDIRECT;

    //cargamos el segundo nivel y asignamos memoria si es necesario
    if((addr = a[ind_n1]) == 0){
        a[ind_n1] = addr = balloc(ip->dev);
        log_write(bp);
    }
    brelse(bp);

    // seleccionamos el segundo nivel los bloques doblemente indirectos
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;

    // si el indice del segundo nivel esta vacio asignamos memoria
    if((addr = a[ind_n2]) == 0){
        a[ind_n2] = addr = balloc(ip->dev);
        log_write(bp);
    }

    brelse(bp);
    return addr;
}
```

comprobamos las modificaciones realizadas ejecutando el programa de prueba big y nos sale que xv6 permite escribir ficheros de 16523 vectores.

```
init: starting sh
$ big
......................
......................
.....
wrote 16523 sectors
done; ok
$ _
```