Índice

1. Boletín 6

En este boletín el objetivo es estudiar como se implementan las llamadas al sistema en XV6, en concreto vamos a realizar la implementación de dos nuevas llamadas. En particular van a ser **Date** y **Dup2**.

1.1. Ejercicio 1: Llamada al sistema Date

En primer lugar hemos modificado el archivo *syscall.h* para darle un nuevo número de llamada a **Date**. A continuación añadimos la llamada date en *usys.S*.

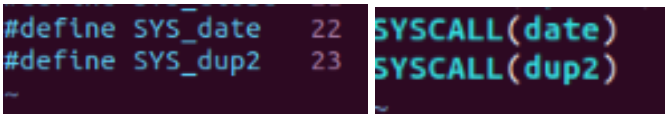

Figura 1: Definiciones de códigos y llamadas para date y dup2.

Luego añadimos la definición de la función **sys_date()** en *syscall.c* para poder posteriormente implementar la función **sys_date()** en *sysproc.c* donde recogemos el parámetro de la pila y hacemos uso de la función **cmostime()** para obtener la fecha a partir de dicho parámetro comprobando todos los errores. Por último, añadimos la llamada **date()** al fichero *user.h*.

```
int
sys_date(void)
{
        struct rtcdate * d;

        //obtener los parametros de la llamada (struct rtcdate*)
        if (argptr(0, (char**)&d, sizeof(struct rtcdate)) != 0)
                return -1;



        //Implementacion de la llamada con cmostime()
        cmostime(d);

        return 0;
}
```

Figura 2: Implementación de **date()**.

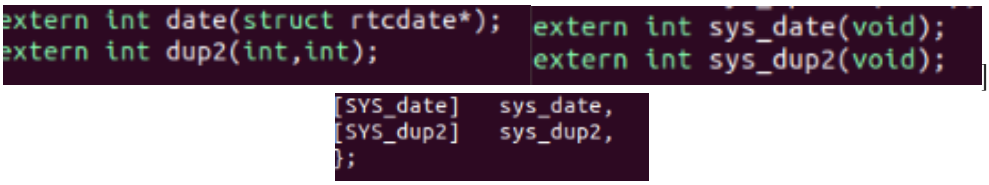

Figura 3: Declaraciones necesarias para la correcta implementación de date y dup2.

Para comprobar el correcto funcionamiento de la llamada al sistema será necesario añadir *_date* a la definición **UPROGS** del **Makefile**.

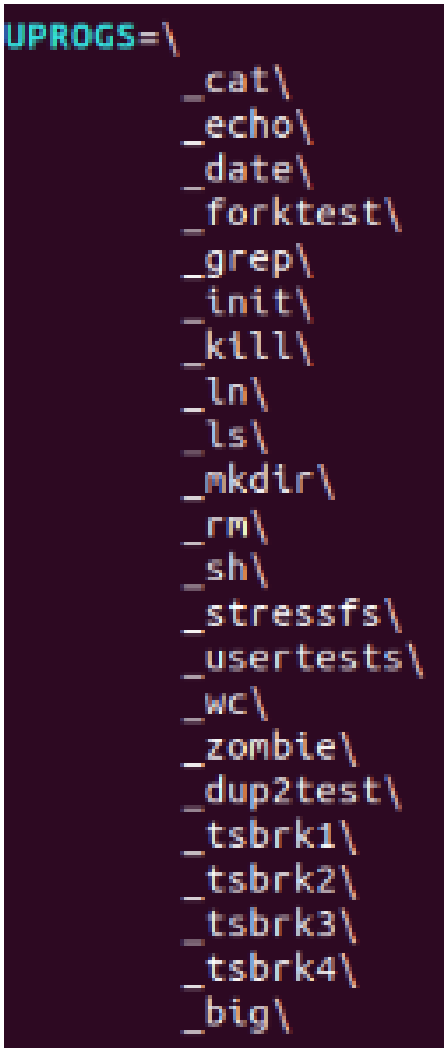

Figura 4: Sección UPROGS del archivo *Makefile*.

1.2. Ejercicio 2: Llamada al sistema Dup2

Para la realización de este ejercicio hemos seguido los mismos pasos al ejercicio anterior, con la diferencia de que la implementación de la función **sys_dup2()** está en el archivo *sysfile.c*. En cuanto a la funcionalidad, hemos seguido las indicaciones sobre el comportamiento de dicha llamada al sistema según el estándar POSIX. En concreto hemos comprobado en primer lugar los errores antes de duplicar la entrada; comprobando que estuviera en el rango de la tabla de descriptores de ficheros. Además, comprobamos si ambos descriptores de ficheros son iguales, en cuyo caso devolvemos el nuevo descriptor de ficheros. En otro caso, comprobaremos si está abierto el fichero destino lo cerramos silenciosamente antes de continuar, y posteriormente duplicamos la entrada en la nueva posición y devolvemos dicha entrada.

```
int
sys_dup2(void)
{
  struct file *f;
  int fd0, fd1;


  if(argfd(0, &fd0, &f) < 0
    || argint(1,&fd1) < 0 || fd1 < 0 || fd1 >= NOFILE)
    return -1;
 // If the newfd isnt in the reange of the open files per process, return -1


  // If oldfd and newfd are the same, dup2 does nothing and return the newfd
  if(fd1 == fd0)
    return fd1;

  // Silently close the newfd if it was previosly open
  if(myproc()->ofile[fd1]){
    fileclose(myproc()->ofile[fd1]); // Atomic close
  }

  // After every error is checked, duplicate.
  myproc()->ofile[fd1] = filedup(f);

  return fd1;

}
```

Figura 5: Implementación de **dup2()** siguiendo el estándar POSIX.

2. Boletín 7

En este boletín vamos a realizar alguna optimización sobre el S.O XV6, como puede ser la reserva bajo demanda de páginas en la memoria *heap(lazy allocation)*. Esta mejora tiene como objetivo mitigar el problema de tener en memoria arrays dispersos de gran tamaño, ya que estos reservan mucha memoria que quizas nunca utilicen.

2.1. Ejercicio 1 y 2: Reserva de páginas bajo demanda

A la hora de implementar esta característica, en *sysproc.c* hemos diferenciado dos casos, ya que no se debe llamar a **growproc()** si queremos que el proceso aumente de tamaño en la función **sys_sbrk()**, sino que comprobaremos que el proceso no se pase del tamaño máximo de la memoria del usuario en ejecución y aumentaremos directamente su tamaño sin reservar páginas; para que así, al ocurrir un fallo de página, dicha página se reserve bajo demanda. En el caso de que queramos decrementar el tamaño del proceso reutilizaremos la implementación de **growproc()** para liberar las páginas y actualizar el TLB con la función **switchuvm()**.

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

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

  addr = myproc()->sz;

  if(n < 0)
  {                     // On decrease of process size,
    if(growproc(n) < 0) // use growproc() implementation.
      return -1;
  }
  else
  {
    if (myproc()->sz + n >= KERNBASE) // Important: check that the increase is inside
      return -1;                      // the process space.
    myproc()->sz += n;
  }

  return addr;
}
```

Figura 6: Implementación de **sys_sbrk()**.

```
int
growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();

  sz = curproc->sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```

Figura 7: Implemetación de **growproc()** reutilizada a la hora de decrementar el tamaño del proceso.

A continuación, será necesario modificar el codigo en *trap.c* para que sea posible responder a un fallo de página en el espacio de usuario mapeando una nueva página física en la dirección que generó el fallo **rcr2()** mediante una nueva función definida en *vm.c* llamada **allocatepg()** en la que reservamos una nueva página física con **kalloc()** y mapeamos la dirección virtual a esa nueva página física con la función **mappages()** y actualizamos el TLB con **lcr3(V2P(proc->pgdir))**. Además, comprobaremos fallos como que la dirección que dió el fallo esté por debajo del inicio de la pila ó que sobrepase la última dirección asignada dinámicamente al proceso ya que si se da este caso, debemos matarlo.

```
/** Allocate a page and map the virtual address to that physical page.
 * Return -1 on error or 0 if the allocation and map was successful.
**/
int
allocatepg(struct proc * proc, uint va) {
  char* mem;
  uint a = PGROUNDDOWN(va);
  // Reserva una nueva pagina fisica
  mem = kalloc();
  if (mem == 0) {
    cprintf("allocatepg.kalloc out of memory\n");
    return -1;
  }

  memset(mem, 0, PGSIZE);

  // Map virtual address to the previously reserved physical page.
  if (mappages(proc->pgdir, (void*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0) {
    cprintf("allocatepg.mappages out of memory to map pages\n");
    kfree(mem); // On error, free the physical page
    return -1;
  }
  // TLB stores the recent translations of virtual memory to physical memory
  // New page allocation => refresh TLB
  lcr3(V2P(proc->pgdir));
  return 0;
}
```

Figura 8: Implementación de función auxiliar **allocatepg()** situada en *vm.c*.

Para verificar el correcto funcionamiento de **fork()** y **exit()**/**wait()** en el caso de que hayan direcciones virtuales sin memoria reservada para ellas, nos hemos fijado en el momento en el que se realiza la copia de la tabla de paginas del proceso padre al hijo, en la que evitaremos los panic en caso en el que estas páginas estén sin reservar y continuaremos copiando en la siguiente iteración, sustituyendo **panic** por **continue**.

```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      //panic("copyuvm: pte should exist");
      continue;
    if(!(*pte & PTE_P))
      continue; // Si no esta presente la pagina, continuamos con la siguiente
      // iteración en el caso de que haya direcciones virtuales sin memoria reservada
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
      kfree(mem);
      goto bad;
    }
  }
  return d;

bad:
  freevm(d);
  return 0;
}
```

Figura 9: Implementación de **copyuvm()** tras modificaciones.

```
  case T_PGFLT:   // Page fault
   /*  (tf->err & PTE_P = 0) => Fault caused by non-present page
   * if the fault is in the user space, allocate page
   */
   if ((tf->err & PTE_P) == 0 && rcr2() < myproc()->sz)
   {
     if (allocatepg(myproc(), rcr2()) < 0)
     {
         cprintf("Not enough memory\n");
         myproc()->killed = 1;
     }

     break;

   }
   /* It is possible to add an attribute to struct proc
   * pointing to the address of the stack; this attribute would be initialized in exec.
   * However, using this alternative we get rid of unnecessary overhead.
   */
   else if(rcr2() < myproc()->tf->esp)
   {
     //cprintf("Stack overflow\n");
     cprintf("Access violation at: 0x%x\n",rcr2());
     myproc()->killed = 1;
     break;
   }
   else if(rcr2() >= myproc()->sz)
   {
       cprintf("Access violation at: 0x%x\n",rcr2());
       myproc()->killed = 1;
       break;

   }

   //Keep going through default

 //PAGEBREAK: 13
 default:
   if(myproc() == 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, cpuid(), tf->eip, rcr2());
     panic("trap");
   }
   // In user space, assume process misbehaved.
   cprintf("pid %d %s: trap %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;

 }
```

Figura 10: Implementación del tratamiento de fallo de página.

3. Boletín 8

En este boletín la mejora realiza consistirá en incrementar el tamaño máximo permitido para un fichero en XV6 ya que el tamaño está limitado a 71680 bytes mientras que tras la implementación de un bloque doblemente indirecto, un fichero podrá estar constituido por cerca de 8.5 megabytes.

Antes de empezar a hacer los ejercicios hemos aumentado el número de bloques libres en el sistema de ficheros en el archivo *param.h* a 20000, pues los 2000 iniciales eran insuficientes para estos y añadimos el *flag* **QEMUEXTRA=-snapshot** en el Makefile justo antes de QEMUOPTS, acelerando así la creación de ficheros grandes en el emulador.

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

Figura 11: Aumento del número de bloques en el sistema de ficheros.

3.1. Ejercicio 1: Soporte de ficheros grandes

Para empezar hemos cambiado la constante donde almacenamos el número de bloques directos, disminuyendolo en 1 para poder así sustituirlo por un bloque doblemente indirecto. Además hemos modificado la constante MAXFILE, aumentandolo en el máximo número de sectores, y en la estructura de un nodo-i tanto en disco como en memoria hemos cambiado la definición del tamaño del atributo addrs ya que hemos modificado anteriormente NDIRECT con el objetivo de obtener mayor legibilidad.

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

// 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+1+1];   // Data block addresses
};
```

Figura 12: Estructura de nodo-i en disco.

Para realizar la implementación del bloque doblemente indirecto, hemos modificado la función **bmap()**, seguiremos un proceso análogo al realizado para el bloque simplemente indirecto. En primer lugar, comprobamos que el número de bloque no esté fuera del rango. Una vez hecho esto, cargaremos el bloque doblemente indirecto en memoria, en caso de que no tenga memoria asignada se la reservaremos. Posteriormente, accederemos a la entrada del bloque simplemente indirecto mediante el cociente del número de bloque, cargaremos el bloque accedido ó lo reservaremos y escribiremos transaccionalmente en el bloque doblemente indirecto la dirección del bloque simplemente indirecto con la función **log_write()** y liberamos el bloque para el uso de este en otros hilos mediante **brelse()**. Para acceder al bloque de datos correspondiente utilizaremos el módulo y seguiremos los mismos pasos anteriormente citados.

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1+1];
};
```

Figura 13: Estructura de nodo-i en memoria.

```
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  if (bn < NDIRECT)
  {
    if ((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;
  //en addrs[NDIRECT] esta el bsi
  if (bn < NINDIRECT)
  {
    // Load indirect block, allocating if necessary.
    if ((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);

    bp = bread(ip->dev, addr); //BSI
    a = (uint *)bp->data;

    if ((addr = a[bn]) == 0)
    {
      a[bn] = addr = balloc(ip->dev); //Modifica BSI
      begin_op();
      log_write(bp);                  //Programamos la escritura
      end_op();
    }
    brelse(bp);
    return addr;
  }

  bn -= NINDIRECT;

  if (bn < NINDIRECT * NINDIRECT)
  {
    // Load BDI; si necesario asignar memoria
    if ((addr = ip->addrs[NDIRECT + 1]) == 0)
      ip->addrs[NDIRECT + 1] = addr = balloc(ip->dev);

    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;

    // entrada del BSI
    uint indexlvl1 = bn / NINDIRECT;

    if ((addr = a[indexlvl1]) == 0)
    {
      a[indexlvl1] = addr = balloc(ip->dev);
      begin_op();
      log_write(bp);
      end_op();
    }
    brelse(bp);

    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;

    // entrada del bloque directo
    uint indexlvl2 = bn % NINDIRECT;

    if ((addr = a[indexlvl2]) == 0)
    {
      a[indexlvl2] = addr = balloc(ip->dev);
      begin_op();
      log_write(bp);
      end_op();
    }
    brelse(bp);

    return addr;
  }

  panic("bmap: out of range");
}
```

Figura 14: Implementación de **bmap()** tras el soporte de bloques doblemente indirectos.

3.2. Ejercicio 2: Borrado de ficheros con bloques doblemente indirectos

A la hora de realizar el borrado de los ficheros en la función **itrunc()**, hemos creado una función auxiliar llamada **truncbsi()** en la que recogemos la funcionalidad de eliminar un bloque simplemente indirecto y sus bloques de datos. Para empezar, tendremos que comprobar si existe el bloque doblemente indirecto para asi, leerlo y para cada bloque simplemente indirecto del bloque doblemente indirecto, eliminar dicho bloque y sus datos con la función auxiliar anteriormente descrita. Para terminar, liberando el bloque doblemente indirecto.

```
static void
itrunc(struct inode *ip)
{
  int i;
  struct buf * bp;
  uint * a;

  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }

  //eliminar BSI y bloques de datos
        truncbsi(&ip->addrs[NDIRECT],ip->dev);


  //Borrar si tiene el BDI, todos los BSI
  if(ip->addrs[NDIRECT+1])
  {

    // Leemos BDI
    bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
        a = (uint *)bp->data;

    // Para cada BSI del BDI, eliminar BSI y bloques de datos
    for(int j = 0; j< NINDIRECT; j++)
    {
      truncbsi(&a[j],ip->dev);
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT+1]);
    ip->addrs[NDIRECT+1] = 0;
  }


        ip->size = 0;
  iupdate(ip);
}
```

```
void truncbsi(uint *p, uint dev)
{
  int j;
  struct buf *bp;
  uint *a;
  if (*p)
  {
    bp = bread(dev, *p);
    a = (uint *)bp->data;
    for (j = 0; j < NINDIRECT; j++)
    {
      if (a[j])
        bfree(dev, a[j]);
    }
    brelse(bp);     //liberar el bloque para que lo puedan usar otros procesos
    bfree(dev, *p); //liberar en el disco el bloque que tenia el BSI
    *p = 0;
  }


}
```

Figura 15: Implementación de **itrunc()** tras la adición del bloque doblemente indirecto y función auxiliar utilizada para mejorara la legibilidad del código.