MEMORIA XV6

ASIGNATURA: Ampliación de Sistemas Operativos

ALUMNOS: David Fernández Expósito y

Pablo Tadeo Romero Orlowska

DNI: 49444688R (David) y 48665752Y (Pablo)

GRUPO: PCEO

ÍNDICE DE CONTENIDOS

1. INTRODUCCIÓN

En esta memoria se resume el trabajo realizado para completar la práctica de xv6 de la asignatura Ampliación de Sistemas Operativos.

Para cada ejercicio se resumen los cambios realizados o el código que se tuvo que añadir, indicando también donde se hizo cada cosa. En todo caso es un resumen, y no se entra en detalle en aspectos que ya estén perfectamente claros en la especificación del ejercicio en el pdf del boletín de prácticas correspondiente.

2. BOLETÍN 2: SYSTEM CALLS

2.1 EJERCICIO 1

En este ejercicio se implementó una nueva llamada al Sistema, la llamada date(). Para ello, se siguieron los pasos indicados en el pdf y seguidos por el profesor en la clase de prácticas, ya que este ejercicio se implementó con su ayuda en clase. Además de los cambios necesarios para añadir cualquier llamada al sistema, los cuales vienen enumerados en el pdf (añadir número a la llamada en syscall.h, añadir la llamada a usys.S, añadir date() a user.h y añadir la definición de sys_date()), se tuvo que añadir una función sys_date() que implementa la llamada en sí, en sysproc.c. En esta función, se recoge un parámetro de tipo struct rtcdate* de la primera posición de la pila usando la función argptr. Ese parámetro se usa para llamar a la función cmostime() que es realmente con la que se obtiene lo buscado, la fecha.

Nuestra función sys_date de sysproc.c queda:

```
int
sys_date(void)
{
 //En el tope de la pila de usuario esta &r. esp apunta al tope de esta pila.

 struct rtcdate *r;
 if (argptr(0, (void**)&r, sizeof(struct rtcdate)) < 0)
        return -1;
 cmostime(r);
 return 0;

}
```

2.2 EJERCICIO 2

Este ejercicio pide implementar otra llamada al sistema, en este caso dup2(). Lo primero que se hizo fue añadir dup2test a UPROGS para poder ejecutar la prueba. Luego, se siguieron pasos similares a los seguidos para el ejercicio 1 y que se indican en el pdf, para poder añadir la llamada al sistema correctamente. Primero se asignó un nuevo número a la llamada dup2 en syscall.h (se asignó el número 23). También se tuvo que añadir la llamada dup2 en usys.S, añadir la llamada dup2(int, int) en user.h y añadir la definición de sys_dup2() en syscall.c. La función sys_dup2(), al igual que sys_dup(), se implementa en sysfile.c. En ella, hacemos lo que sabemos que debe hacer dup2. Primero se comprueba que old_fd es descriptor válido (argfd) y que el new_fd es válido para albergar un fichero (argint). Luego, en caso de que new_fd sea igual a old_fd y este último es válido, entonces dup2 no hace nada y se devuelve new_fd. Después, si esto último no ocurre, se comprueba si new_fd alberga un fichero abierto (argfd) (también se podría simplemente comprobar la posición correspondiente del array ofile). Si es así, hay que cerrarlo (fileclose). Por último, se duplica como en dup, usando filedup y actualizando la lista de ficheros abiertos del proceso adecuadamente. Se retorna new_fd.

La función sys_dup2 de sysfile.c queda:

```
int
sys_dup2(void)
{
  // 1.Comprobar que oldfd es un fichero valido (argfd)
  struct file *fold;
  struct file *fnew;
  int oldfd;
  int newfd;

  if(argfd(0,&oldfd,&fold) < 0)
        return -1;

  // 2.Comprobar que segundo fichero es valido para albergar un fichero (argint)
  if(argint(1, &newfd) < 0 )
         return -1;

  if (newfd < 0 || newfd >= NOFILE)
         return -1;

  // 3.Si oldfd es valido y es igual a newfd, dup2 no hace nada, y return newfd.
  if (newfd == oldfd)
          return newfd;
  // 4.Si newfd esta abierto, entonces cerrarlo(fileclose)
  if (argfd(1, 0, &fnew) == 0)
          fileclose(fnew);
```

```
  // 5.Duplicarlo parecido a dup
  filedup(fold);
  struct proc *curproc = myproc();
  curproc->ofile[newfd] = curproc->ofile[oldfd];
  // 6. return newfd
  return newfd;

}
```

2.3 EJERCICIO 3

En este ejercicio se pide modificar todo lo necesario para que las llamadas al sistema exit() y wait() pasen a ser exit(int status) y wait(int* status). Además de los pasos que se indican en el pdf, como ejecutar las líneas mostradas en el Shell, adaptar user.h, y adaptar sys_exit() y sys_wait() para que acepten argumentos (con argint y argptr), cabe comentar ciertos aspectos de la implementación.

Primero, es importante notar que hay que añadir un campo status a struct proc (en proc.h). Con esto hacemos posible lo indicado en el punto 5, que cuando se llame a wait se reciba el estado del hijo adecuadamente. Y además de añadir los parámetros a las llamadas sys_wait y sys_exit de sysproc.c, hay que hacer cambios también en las funciones exit y wait de proc.c. Además de añadir status como parámetro, hay que hacer que en exit se ponga el valor de status en el campo status del proceso, mientras que en wait habrá que coger ese valor del proceso que recojamos y meterlo en la variable por referencia que se pasa para poder devolver el status como queremos. Importante notar que hay que comprobar que el puntero pasado en wait no es 0, pues esto puede provocar comportamientos no deseados.

Luego, para que se puedan usar los macros, además de definirlos en user.h como se pide, hay que tener en cuenta: cambiar todas las llamadas a exit y wait para que sean con 0 como parámetro (comandos que se indican en el pdf), menos las del trap.c, que tienen que devolver en exit el valor del trap + 1 (tf->trapno + 1), y en sysproc.c hay que tener en cuenta que, cuando se llama a exit en sys_exit, el valor que hay que pasar no es el status y ya está, hay que desplazarlo hacia la izquierda, se debe hacer exit(status << 8). Todo esto para que funcionen correctamente los macros. También, en sh.c, donde decidamos imprimir “Output code: N” (nosotros lo hacemos la final del main), debemos tener en cuenta los macros a la hora de imprimir.

La función main de sh.c quedó:

```
int
main(void)
{
...
    int status;
```

```
    wait(&status);
    if (WIFEXITED(status)){
      printf(1, "Output code: %d\n", WEXITSTATUS(status));
    }
    if (WIFSIGNALED(status)){
      printf(1, "Output code (failure): %d\n", WEXITTRAP(status));
    }
  }
  exit(0);
}
```

Las funciones exit() y wait() quedaron:

```
void
exit(int status)
{
...
  curproc->cwd = 0;

  curproc->status = status;

  acquire(&ptable.lock);
...
}

int
wait(int *status)
{
...
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
       if (status != 0){
          *status = p->status;
       }
       p->status = 0;
        kfree(p->kstack);
...
}
```

Las funciones sys_wait y sys_exit quedaron:

```
int
sys_exit(void)
{
  int status;
```

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

  status = status << 8;
  exit(status);
  return 0;  // not reached
}

int
sys_wait(void)
{
  int *status;
  if (argptr(0, (void**)&status, sizeof(int)) < 0)
          return -1;
  return wait(status);
}
```

3. BOLETÍN 3: LAZY PAGE ALLOC

3.1 EJERCICIO 1

Este ejercicio consiste en implementar de forma básica la reserva de páginas bajo demanda. Lo hicimos de forma conjunta con el profesor en clase y los pasos a seguir están especificados en el pdf. Lo primero que hicimos fue eliminar (o comentar la línea) donde se llama a growproc() en sys_sbrk() y aumentar el tamaño del proceso de forma "manual" de forma adecuada. Si solo hiciésemos esto, saltaría un error, por lo que debemos modificar el código en trap.c para que responda a un fallo de página en el espacio de usuario mapeando una nueva página en la dirección que generó el fallo. Para ello, todo lo que hay que añadir se puede poner en la sección default del switch antes de llamar a cprintf (lo que hará que se impriman algunos errores pese a ser solucionados). Sin embargo, nosotros optamos por añadir un case T_PGFLT y tratar este error aquí. Lo que hicimos fue llamar a kalloc (para reservar una nueva página física), y luego a mappages (mapear esa página en la dirección virtual que ha dado fallo (rcr2())) y a lcr3 (para invalidar el TLB). Cabe mencionar que para poder usar mappages sin problema hay que borrar la declaración de función estática en vm.c.

3.2 EJERCICIO 2

Este ejercicio consiste en tener en cuenta todos los casos donde la implementación anterior no es correcta (falta tener en cuenta ciertas situaciones o casos que no son correctamente tratados). En primer lugar, en caso de pasar un argumento negativo a sbrk, lo que debemos hacer es llamar a growproc (growproc ya se encargará de hacer lo necesario cuando n es negativo. Hacemos esto pues el caso negativo es más complicado

y delegamos directamente en growproc para simplificar). También debemos comprobar que el tamaño que crecemos no es demasiado, ni que el tamaño que decrecemos es demasiado. Es decir, que el tamaño no pasará de KERNBASE ni bajará a la pila.

En segundo lugar, debemos controlar que no se acceda a la página guarda. Para ello, creamos un campo más en la struct proc para almacenar la dirección base de la página guarda (la inicializamos en exec() cuando se reservan las páginas guarda y la de la pila). Una vez hecho esto, simplemente se comprueba (en trap.c) si se está intentando acceder a la página guarda. Si es así, se pone killed = 1.

En tercer lugar, para hacer que fork funcione correctamente, hay que modificar el código de la función copyuvm, en vm.c, para que funcione cuando hay direcciones virtuales sin memoria reservada para ellas. Simplemente hay que comentar o eliminar las llamadas a panic en 2 sentencias if y sustituirlas por continue.

Por último, hay que asegurarse de que el uso por parte del kernel de páginas de usuario que todavía no han sido reservadas funciona correctamente. En nuestro caso, como todo lo que añadimos o modificamos en trap.c lo hacemos un nuevo caso del switch, no nos afecta en nada el if que hay en el default (pues ponemos un break al final del case) y todo funciona correctamente. Por ejemplo, tsbrk3 nos funciona sin problema. Lo único que tuvimos que tener en cuenta fue añadir la comprobación de si el proceso era el 0, pues por el break no se haría si no se añadiese.

La función sys_sbrk de sysproc.c queda:

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

  addr = myproc()->sz;

  if(argint(0, &n) < 0)
  {
    return -1;
  }
  if (n < 0){
    //Compruebo que no se decremente demasiado
    //como la pila tiene codigo + datos de tamaño, y la guarda es eso justamente,
    //lo usamos como tamaño de la pila.
    if (myproc()->sz + n < 2*myproc()->guarda + 1){
            return -1;
    }
    if (growproc(n) < 0){
            return -1;
```

```
    }
  } else{
    //Compruebo que no nos pasamos.
    if (myproc()->sz + n >= KERNBASE){
           return -1;
    }
    //añadimos lo de abajo por ejercicio1
    myproc()->sz += n;
    //if(growproc(n) < 0)
    //return -1;

  }
  return addr;
}
```

La función copyuvm de vm.c queda:

```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      continue;
      //panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      continue;
      //panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
...
}
```

El case correspondiente en trap.c queda:

```
case T_PGFLT:
    {
     if (myproc() == 0){
         cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
              tf->trapno, cpuid(), tf->eip, rcr2());
         panic("trap");
        break;
     }
     if (rcr2() > myproc()->sz){
         myproc()->killed = 1;
         cprintf("pid %d: out of range\n", myproc()->pid);
         break;
```

```
    }
    if((tf->err & 0x01) == 1){
        uint direccion = PGROUNDDOWN(rcr2());
        myproc()->killed = 1;
        if (myproc()->guarda == direccion){
           cprintf("pid %d: no puedes acceder a la pagina guarda\n", myproc()->pid);
        }else{
          cprintf("pid %d: fallo de permisos", myproc()->pid);
        }
        break;
    }
    //Necesitamos un lcr3, que invalida el TLB
    //Reservar una nueva pagina fisica (kalloc)
    //Mapear esa pagina en la direccion virtual que ha dado fallo (en rcr2())(con
mappages)
    char * mem = kalloc();
    if (mem == 0){
       myproc()->killed = 1;
       cprintf("pid %d: out of memory\n", myproc()->pid);
    } else {
       memset(mem, 0, PGSIZE);
       //Hemos tenido que quitar el static de mappages en vm.c para que no falle al
compilar y se pueda llamar.
       if (mappages(myproc()->pgdir, (char*)PGROUNDDOWN(rcr2()), PGSIZE, V2P(mem), PTE_W
| PTE_U) < 0){
          myproc()->killed = 1;
          cprintf("pid %d: mappages failed\n", myproc()->pid);
       }
       lcr3(V2P(myproc()->pgdir));
    }

    break;
  }
```

3.3 EJERCICIO 3

En este ejercicio se nos pide que se asigne el mismo número de páginas a la pila que número de páginas de código+datos. Para ello, en exec() (en exec.c), donde se asignan las dos páginas de la pila y la guarda (el mismo sitio donde se inicializa el campo guarda de struct proc), hay que hacer allocvm(pgdir, sz, sz+suma) (suma = sz+PGSIZE) en lugar de allocvm(pgdir, sz, sz + 2*PGSIZE). Esto lo hacemos así porque, hasta ese momento, sz vale exactamente código + datos, por lo que simplemente hay que duplicar sz (asignar el número de páginas que sea sz a la pila). Nótese que se asigna uno más, para la guarda.

El código queda:

```
int
exec(char *path, char **argv){
  ...
  sz = PGROUNDUP(sz);
  curproc->guarda = sz;
  int suma = sz + PGSIZE;
  //Para tener mismo numero paginas en pila que datos mas codigo, sz es ese tamano hasta
este momento.
  if((sz = allocuvm(pgdir, sz, sz + suma)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - suma));
  sp = sz;
  ...
}
```

3.4 EJERCICIO 4

En este ejercicio se pide implementar una llamada al sistema freemem para conocer cuanta memoria libre queda. Para ello, añadimos a la estructura de memoria que se usa en kalloc.c un contador que se inicializa a 0 en kinit1, y que en cada llamada a kfree y kalloc se va actualizando como corresponda (en kalloc se resta 1 y en kfree se suma 1) (notar que freerange llama a kfree por lo que todo se inicializa bien). Luego, añadimos una función freemem que, tomando el cerrojo de la memoria correctamente, simplemente devuelve el valor del contador. En sysproc.c, es donde implementamos una función sys_freemem que en función del type devuelve una cosa u otra, pero que en el fondo simplemente llama a freemem de kalloc.c. Para que esto funcione, debemos añadir en defs.h esta nueva función, freemem, de kalloc.c.

Además de esto, hay que hacer las modificaciones usuales a la hora de añadir una llamada al sistema, como se hizo con los primeros ejercicios del boletín anterior.

Las funciones kalloc.c y kfree.c de kalloc.c quedan:

```
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
```

```
  r->next = kmem.freelist;
  kmem.freelist = r;
  kmem.freemem++;
  if(kmem.use_lock)
    release(&kmem.lock);
}
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r){
    kmem.freelist = r->next;
    kmem.freemem--;
  }
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

La función freemem de kalloc.c:

```
int
freemem(void)
{
 if (kmem.use_lock)
   acquire(&kmem.lock);
 int free = kmem.freemem;
 if (kmem.use_lock)
   release(&kmem.lock);
 return free;
}
```

La función sys_freemem de sysproc.c:

```
int
sys_freemem()
{
  int type;
  if (argint(0, &type) < 0){
          return -1;
  }
```

```
  if (type < 0 || type > 1){
          return -1;
  }

  if (type == 1){
          return freemem()*PGSIZE;
  }
  return freemem();
}
```

4. BOLETÍN 4: BIG FILES

4.1 EJERCICIO 1

Hay que modificar bmap() (fichero fs.c) para que se implemente un bloque doblemente indirecto. Para ello, sacrificamos un bloque directo. Por tanto, hay que modificar el valor de NDIRECT a 11 en fs.h. Esto hace que haya que modificar el tamaño de los arrays addrs en inode y en dinode (y deben coincidir ambos tamaños).

En cuanto a modificaciones en bmap(), tras tratar el bloque indirecto hay que tratar el doblemente indirecto. Hay que comprobar si esta asignado y crearlo si no lo está. Luego hay que ver si el BSI dentro del BDI se ha creado o no, y crearlo o reservarlo con balloc si es que no. El numero de BSI dentro del BDI es bn/NINDIRECT (bn es el número de bloque, pero se le han ido haciendo restas para ir viendo si era directo, o si estaba dentro de uno indirecto o doblemente indirecto) y la posición dentro del BSI es bn%NINDIRECT. Por ultimo, hay que ver la entrada del BSI, y si no esta asignada, reservarla con balloc. Recordar que hay que hacer brelse cada vez que se haga bread.

La función bmap queda:

```
static uint
bmap(struct inode *ip, uint bn)
{
...
  bn -= NINDIRECT;
  // tratar el BDI
  if(bn < NINDIRECT*NINDIRECT)
  {
    //Comprobar si el BDI esta asignado
    //ver el numero del BSI dentro del BDI
    //ver si ese BSI se ha creado
    //calculoae que entrada del BSI tienes que coger, y si no esta asignada,
    //asignarla con balloc y retomarla
    if((addr = ip->addrs[NDIRECT+ 1]) == 0)
      ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
```

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

    int entrada = bn/NINDIRECT;
    int despl = bn%NINDIRECT;

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

    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[despl]) == 0)
    {
      a[despl] = addr = balloc(ip->dev);
      log_write(bp);
    }
    brelse(bp);
    return addr;

  }
...
}
```

4.2 EJERCICIO 2

Este ejercicio pide implementar el borrado de ficheros con bloques doblemente indirectos. Se hace todo en la función itrunc() de fs.c. Simplemente hay que ver, tras eliminar el BSI, si hay BDI. En caso afirmativo, se lee el BDI (bread) y, para cada BSI, se lee y se van borrando los bloques (bfree) y al final se borra el BSI en sí. Cuando se ha hecho esto con todos los BSI dentro del BDI, solo queda borrar el BDI en sí. Hay que recordar hacer brelse por cada bloque que se haga bread.

La función itrunc queda:

```
static void
itrunc(struct inode *ip)
{
  ...
  //si existe el BDI
  if(ip->addrs[NDIRECT+1])
  {
    //lees BDI
```

```
    bp = bread(ip->dev, ip->addrs[NDIRECT + 1]);
    a = (uint*) bp->data;
    for(j = 0; j < NINDIRECT; j++)
    {
      if (a[j]){
        bp2 = bread(ip->dev,a[j]);
        a2 = (uint*)bp2->data;
        for (h = 0; h < NINDIRECT; h++){
          if(a2[h] != 0){
            bfree(ip->dev, a2[h]);
           a2[h] = 0;
          }
        }
       brelse(bp2);
       bfree(ip->dev, a[j]);
        a[j] = 0;
      }

    }
   brelse(bp);
   bfree(ip->dev, ip->addrs[NDIRECT + 1]);
   ip-> addrs[NDIRECT+1] = 0;

  }

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

5. CONCLUSIONES

Como conclusión, estamos satisfechos con el resultado final de la práctica pues hemos podido realizar todos los ejercicios de forma, en un principio, satisfactoria. Hemos hecho todos los test y pruebas que tenemos a disponibles y todas funcionan correctamente.

Sin embargo, durante el transcurso de la práctica nos fuimos encontrando con ciertas dificultades para completar algunos ejercicios. En una gran parte de los casos, dedicándole tiempo a comprender mejor lo que se pedía y a entender mejor cómo funciona xv6, pudimos solventar la dificultad. No obstante, hubo ocasiones en las que tuvimos que consultar a nuestro profesor de prácticas, Diego Sevilla. Con sus aclaraciones pudimos seguir adelante con el desarrollo de la práctica.

En resumen, esta práctica, a pesar de no ser sencilla, nos ha resultado interesante y nos ha ayudado a comprender mejor y poner en práctica conceptos vistos en la teoría de la asignatura.