DOCUMENTACIÓN PROYECTO: XV6

Germán Gil Planes
(german.gilp@um.es – 49246858V)
Pedro Jiménez Gómez
(pedro.j.g@um.es – 21062899M)
Ampliación de Sistemas Operativos
Grupo 1 – Subgrupo 1.3 - PCEO
Profesor: Diego Sevilla Ruiz

ÍNDICE

BOLETÍN 1

Ejercicio 1.1. Añadir la nueva llamada al sistema: `int date(struct rtcdate *d)`

Tanto para este ejercicio como para el resto, hemos intentado seguir los pasos que nos indicaba cada boletín de prácticas. En este primer caso, queremos añadir la llamada al sistema date, que le pide el tiempo UTC actual al kernel y lo devolverá al programa usuario. Para ello, creamos en el directorio user un fichero llamado date.c, con el siguiente fragmento de código:

```
1. #include "types.h"
2. #include "user.h"
3. #include "date.h"
4.
5. int
6. main (int argc , char * argv [])
7. {
8.   struct rtcdate r;
9.   if (date (&r)) {
10.    printf (2, "date failed\n");
11.    exit(0) ;
12.  }
13.
14.   printf(1, "Fecha actual: %d/%d/%d\nHora: %d:%d:%d\n", r.day, r.month, r.year, r.hour, r.minute,
r.second);
15.
16.  exit(0) ;
17. }
18.
```

Luego, tenemos que añadir:

- `date\` en la definición de `UPROGS` (en el Makefile del directorio user/)

- un nuevo número de llamada al sistema, a date en syscall.h. En nuestro caso, hemos elegido el número 22 (`#define SYS_date 22`).

- `SYSCALL(date)` en usys.S.

- la llamada date() a user/user.h, que contiene la definición de llamadas al sistema para programas de usuario. Lo hacemos así: `extern int date(struct rtcdate *d)`.

Al haber hecho esto, los programas del sistema operativo pueden realizar esta *syscall.* Tras esto, debemos añadir la definición de la función en syscall.c, escribiendo `extern int sys_date(void)`, tras el resto de sentencias extern, y añadir `[SYS_date] sys_date` en el array de funciones.

Para acabar, ya solo nos queda implementar en sysproc.c la función `sys_date()`:

```
1. int
2. sys_date(void)
3. {
4.    struct rtcdate *d;
5.
6.    if(argptr(0, (void**)&d, sizeof(struct rtcdate)) < 0)
7.        return -1;
```

```
 8.
 9.    cmostime(d);
10.    return 0;
11. }
12.
```

Como se puede ver, comprobamos que `argptr()` no devuelva un número negativo, pues en ese caso ha habido un error, y se debe retornar -1.

Ejercicio 1.2. Implementar la llamada al sistema `dup2()` y modificar el *shell* para usarla.

Para hacer este apartado, nos hemos apoyado en la implementación de su "llamada hermana" `dup()`, y también hemos consultado cómo debe comportarse `dup2()` según el estándar POSIX, intentando seguir los pasos que completamos en el apartado anterior. Al igual que antes:

- escribimos `dup2test\` en la definición de `UPROGS` (en el Makefile del user/)

- asignamos el número 23 a esta llamada, en syscall.h, de esta forma: `#define SYS_dup2 23`.

- ponemos, junto al resto de *syscalls*, `SYSCALL(dup2)` en user/usys.S.

- añadimos `extern int dup2(int, int)` en user/user.h.

En syscall.c también hay que escribir `extern int dup2(int, int)` y `[SYS_dup2]  sys_dup2`, donde hemos puesto sus homólogos de `date()`, respectivamente.

Esta llamada se implementa en el fichero sysfile.c, donde está también implementado `dup()`. El código que añadimos es:

```
 1. int
 2. sys_dup2(void)
 3. {
 4.   struct file *oldf, *newf;
 5.   int oldfd, newfd;
 6.
 7.   if((argfd(0, &oldfd, &oldf) < 0 ))
 8.     return -1;
 9.
10.   if (argint(1, &newfd) < 0)
11.     return -1;
12.
13.   if (newfd < 0 || newfd > NOFILE)
14.     return -1;
15.
16.   if (oldfd == newfd)
17.     return newfd;
18.
19.   if ((newf=myproc()->ofile[newfd]) != 0)
20.     fileclose(newf);
21.
22.   myproc()->ofile[newfd] = myproc()->ofile[oldfd];
23.   filedup(oldf);
24.
25.   return newfd;
26. }
27.
```

Para implementarla, nos hemos inspirado en `sys_dup()`, que devuelve el descriptor de fichero más pequeño disponible, con `fdalloc()`. Según el estándar POSIX, `dup2()` funciona de manera similar, pero utiliza el descriptor de fichero *newfd* en vez de encontrar el más bajo. En esta función,

lo primero que hacemos es comprobar si los descriptores son válidos. Si no lo son, se trata de un error, y hay que devolver -1. Después, mira si ambos descriptores coinciden, pues en ese caso, se devuelve el nuevo descriptor. Si no son iguales, intenta cerrar *newfd* si está abierto, *oldfd* se asocia al descriptor *newfd* del proceso actual, se duplica *oldf*, y se devuelve *newfd*.

Ejercicio 1.3. Modifica las llamadas `exit()` y `wait()` para que sigan la signatura de las funciones de POSIX, es decir: `int wait(int *status)` y `void exit (int status).`

Análogamente al apartado anterior, hemos estado mirando ambas llamadas según el estándar POSIX, para completar correctamente el ejercicio. Al principio, hay que cambiar todas las llamadas `exit()` y `wait()` por `exit(0)` y `wait(NULL)` en los programas de usuario, respectivamente. Lo resolvemos ejecutando desde la terminal:

```
$ sed -i -e 's/\bexit()/exit(0)/g' user /*.c
$ sed -i -e 's/\bwait()/wait(NULL)/g' user /*.c
```

Hemos añadido a la `struct proc` un nuevo campo, un número entero, `status`, para almacenar el estado de salida del proceso.

Seguidamente hay que adaptar las funciones de llamada al sistema en user/user.h para aceptar los argumentos. Las 2 líneas modificadas quedarían así:

```
extern int exit(int) __attribute__((noreturn));
extern int wait(int *);
```

También hemos tocado defs.h, para poder devolver en un paso posterior el número de trap:

- Pasamos de `void` exit(`void`) a void exit(`int`)
- Pasamos de `int` wait(`void`) a `int` wait(`int *`)

Además, en sysproc.c, el código de las funciones quedaría de la siguiente forma:

```
 1. int
 2. sys_exit(void)
 3. {
 4.   int status;
 5.
 6.   if(argint(0, &status) < 0)
 7.     return -1;
 8.   exit(status << 8);     //De los 32 bits que devolvemos, los ultimos 16 bits son: los 8 primeros para el status y los 8 ultimos para el trap.
 9.   return 0;  // not reached
10. }
11.
12. int
13. sys_wait(void)
14. {
15.   int *status;
16.
17.   if(argptr(0, (void**)&status, sizeof(int)) < 0)
18.     return -1;
19.   return wait(status);
20. }
21.
```

De la parte del `exit`, cabe destacar que llamamos a `exit()` con el status desplazado 8 parámetros porque, de los 32 bits que devolvemos, los 8 "penúltimos" para el status y los 8 últimos para el *trap*. Por su lado, `wait` necesita un puntero, por lo que llama a `argptr()` y devuelve el resultado de `wait(status)`.

Ahora tocamos la implementación de las funciones en el núcleo para que se cumpla lo que dice el paso 5: cuando un proceso llame a `wait(estado)` y reciba el estado de su hijo, este sea exactamente el que el programa que terminó especificó en su llamada a `exit(estado)`. Nos vamos ahora a proc.c:

```
 1. void
 2. exit(int status)
 3. {
 4.   struct proc *curproc = myproc();
 5.   struct proc *p;
 6.   int fd;
 7.
 8.   if(curproc == initproc)
 9.     panic("init exiting");
10.
11.   curproc->status = status;
12.
13.   // Close all open files.
14.   for(fd = 0; fd < NOFILE; fd++){
15. (…)
16.
17. int
18. wait(int *status)
19. {
20.   struct proc *p;
21.   int havekids, pid;
22.   struct proc *curproc = myproc();
23.
24.   acquire(&ptable.lock);
25.   for(;;){
26.     // Scan through table looking for exited children.
27.     havekids = 0;
28.     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
29.       if(p->parent != curproc)
30.         continue;
31.       havekids = 1;
32.       if(p->state == ZOMBIE){
33.
34.         if(status != NULL)
35.           *status = p->status;
36.         p->status=0;
37.
38.         // Found one.
39.         pid = p->pid;
40.       (…)
```

Aquí, para exit solamente hemos añadido una línea, que asigna al proceso actual *status* como estado de salida. Esto hay que hacerlo antes de despertar al proceso padre (antes de la invocación a `wakeup()`). Para arreglar `wait()`, solamente debemos hacer una pequeña comprobación si el estado es un estado *zombie*. Para ello, miramos que el puntero *status* no sea nulo. En ese caso, accedemos y guardamos el estado. Posteriormente, pase lo que pase, lo ponemos a 0.

Como el paso 6 nos pide que modifiquemos la llamada al sistema `wait()` del *shell* para que cada vez que ejecute un programa produzca una salida, abrimos sh.c y quedaría, dentro de su `main()`:

```
// Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      // Chdir must be called by the parent, not the child.
      buf[strlen(buf)-1] = 0;  // chop \n
      if(chdir(buf+3) < 0)
        printf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0)
      runcmd(parsecmd(buf));

    wait(&status);
    if(WIFEXITED(status))
      printf(1, "Output code: %d\n", WEXITSTATUS(status));

    if(WIFSIGNALED(status))
      printf(1, "Output code: %d\n", WEXITTRAP(status));

```

Cabe destacar que estamos haciendo uso de unos macros que debemos definir en user/user.h:

```
#define WIFEXITED(status)   (((status) & 0x7f) == 0)
#define WEXITSTATUS(status) (((status) & 0xff00) >> 8)
#define WIFSIGNALED(status) (((status) & 0x7f) != 0)
#define WEXITTRAP(status)   (((status) & 0x7f) - 1)
```

Por último, también hemos modificado todos los `exit()` dentro de trap.c, para que devuelvan el número de *trap*. Este pequeño cambio consistía en cambiar cada línea `exit()` por `exit(tf->trapno + 1)`.

BOLETÍN 2

Ejercicio 2. Implementar la reserva diferida en xv6, contemplando las siguientes situaciones:

- El caso de un argumento negativo al llamarse a `sbrk()`

- Manejar el caso de fallos en la página inválida debajo de la pila

- Verificar que `fork()` y `exit()`/`wait()` funciona en el caso de que haya direcciones virtuales sin memoria reservada para ellas

- Asegurarse de que funciona el uso por parte del kernel de páginas de usuario que todavía no han sido reservadas (p.e., si un programa pasa una dirección de la zona de usuario todavía no reservada a `read()`)

Como para hacer el ejercicio 2 de este boletín hay que modificar lo hecho en el primer apartado, hemos decidido tratarlo como un solo ejercicio, y comentar los cambios finales sobre los ficheros de xv6.

Comenzamos con sysproc.c, en el que debemos eliminar la reserva de páginas de la *syscall* `sbrk()`, implementada en la función `sys_sbrk()`. Tras completar los cambios, el código nos quedaría tal que así:

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

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

  addr = myproc()->sz;
```

```
11.   if(n < 0)
12.   {
13.    if(myproc()->sz + n < PGROUNDUP(myproc()->tf->esp) || deallocuvm(myproc()->pgdir, addr, addr+n)==0
)
14.       return -1;
15.   }
16.   if(myproc()->sz + n >= KERNBASE)
17.     return -1;
18.
19.   myproc()->sz += n;
20.   lcr3(V2P(myproc()->pgdir));
21.   /*
22.   if(growproc(n) < 0)
23.     return -1;
24.   */
25.   return addr;
26. }
27.
```

Esta función debe devolver el tamaño antiguo tras modificar el tamaño del proceso, pero no debe reservar memoria. Además, como dice el enunciado, no se debe llamar a `growproc()` en caso de que el proceso crezca. Por eso, comentamos la parte en la que llama a `growproc()`, y modificamos la memoria del proceso manualmente, habiendo comprobado previamente que no entremos en el espacio de direcciones del Kernel cuando incrementamos. Sin embargo, si *n* es un número negativo, hay que desmapear y liberar los bloques que ya no se necesitan. En este caso, no podemos permitir que se libere por debajo de la zona de usuario, llegando a la pila o los datos del programa. Por esta razón lo comprobamos. Siempre que haya un error devolvemos -1.

A continuación, en trap.c hay que capturar el trap número 14, `T_PGFLT`, definido en traps.h. Para ello añadimos un nuevo case en el `switch(tf→trapno)`, en el que se debe mapear una nueva página física en la dirección del fallo y regresar al espacio usuario, para que el proceso continúe. Notificamos el fallo, reservamos una página física y la vaciamos con `memset()`. También tenemos que comprobar que los accesos a memoria sean válidos. Una vez ya se cumplan estas condiciones, ya podemos reservar memoria siempre que quede espacio disponible, es decir, que el valor devuelto por `kalloc()` no sea 0.

Como indica el boletín, para utilizar `mappages()` debemos eliminar su declaración estática en vm.c, e incluir la cabecera de la función en defs.h() (`int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)`). Cuando llamemos a esta función, si nos devuelve un error, liberamos la memoria reservada con `kfree()` y continuamos.

```
1.   case T_PGFLT:         //page fault
2.     /*
3.     cprintf("pid %d %s: trap %d err %d on cpu %d "
4.             "eip 0x%x addr 0x%x--kill proc\n",
5.             myproc()->pid, myproc()->name, tf->trapno,
6.             tf->err, cpuid(), tf->eip, rcr2());
7.     */
8.
9.     uint dir_fallo = rcr2(); // 0x4004
10.    uint pag_fallo = PGROUNDDOWN(dir_fallo); // 0X4000
```

```
11.
12.     if(tf->err & PTE_P || dir_fallo > myproc()->sz)
13.     {
14.       cprintf("Page fault: Unreserved memory access attempt\n");
15.       myproc()->killed = 1;
16.     }
17.     else
18.     {
19.       char* pagina_nueva_fisica = kalloc();
20.       if(pagina_nueva_fisica == 0)
21.       {
22.         cprintf("allocuvm out of memory\n");
23.         myproc()->killed = 1;
24.       }
25.       memset(pagina_nueva_fisica, 0, PGSIZE);
26.       if(mappages(myproc()->pgdir, (void*)pag_fallo, PGSIZE, V2P(pagina_nueva_fisica), PTE_W|PTE_U)
< 0)
27.       {
28.         cprintf("allocuvm out of memory (2)\n");
29.         kfree(pagina_nueva_fisica);
30.         myproc()->killed = 1;
31.       }
32.     }
33.     break;
34.
```

Por último, en la función `copyvm()` de vm.c hemos insertado unos `continue` en vez de llamar a `panic()`, ya que debemos dejar de tratarlos como errores. Tras todos los cambios, no nos importa si está o no la página: no se ha necesitado si no está presente, y no tendrá información al copiar.

```
 1. pde_t*
 2. copyuvm(pde_t *pgdir, uint sz)
 3. {
 4.   pde_t *d;
 5.   pte_t *pte;
 6.   uint pa, i, flags;
 7.   char *mem;
 8.
 9.   if((d = setupkvm()) == 0)
10.     return 0;
11.   for(i = 0; i < sz; i += PGSIZE){
12.     if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
13.       continue; //panic("copyuvm: pte should exist");
14.     if(!(*pte & PTE_P))
15.       continue; //panic("copyuvm: page not present");
16.     pa = PTE_ADDR(*pte);
17.     flags = PTE_FLAGS(*pte);
18.     if((mem = kalloc()) == 0)
19.       goto bad;
20.     memmove(mem, (char*)P2V(pa), PGSIZE);
21.     if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
22.       kfree(mem);
23.       goto bad;
24.     }
25.   }
26.   return d;
27.
28. bad:
29.   freevm(d, 1);
30.   return 0;
31. }
32.
```

BOLETÍN 3

Ejercicio 3.1. Añade el mecanismo de dotar de mayor prioridad a uno o varios procesos a xv6.

Hemos empezado creando un tipo enumerado para la prioridad de los procesos. En proc.h insertamos esta línea:

```
1. enum proc_prio { HI_PRIO=0, NORM_PRIO=1, ERR_PRIO=2 };
```

Aunque el enunciado nos dice que el enumerado tenga 2 posibles valores, creemos que es conveniente añadir un tercer valor para devolver un posible error a la hora de implementar `get_prio()`.

También debemos modificar la estructura del proceso para añadirle una variable de este tipo que hemos creado para guardar la prioridad del proceso. Así, tras los cambios debidos a este ejercicio y al 1.3, la `struct proc` quedará de esta forma:

```
 1. // Per-process state
 2. struct proc {
 3.   uint sz;                     // Size of process memory (bytes)
 4.   pde_t* pgdir;                // Page table
 5.   char *kstack;                // Bottom of kernel stack for this process
 6.   enum procstate state;        // Process state
 7.   int pid;                     // Process ID
 8.   struct proc *parent;         // Parent process
 9.   struct trapframe *tf;        // Trap frame for current syscall
10.   struct context *context;     // swtch() here to run process
11.   void *chan;                  // If non-zero, sleeping on chan
12.   int killed;                  // If non-zero, have been killed
13.   struct file *ofile[NOFILE];  // Open files
14.   struct inode *cwd;           // Current directory
15.   char name[16];               // Process name (debugging)
16.   int status;                  // Estado del proceso
17.   enum proc_prio priority;     // Prioridad del proceso
18. };
19.
```

En el tercer paso nos pide que a cada nuevo proceso le asignemos prioridad normal. Para ello, en la función `allocproc()`, de proc.c, debemos asignar `NORM_PRIO` a la prioridad del proceso en la etiqueta `found`, antes del `release()`. La línea que hay que añadir es `p->priority = NORM_PRIO`. También tenemos que tocar el `fork()`, para que el proceso hijo herede la prioridad del proceso padre, poniendo `np->priority = curproc->priority`.

Nos queda ahora modificar el `scheduler()`. Inicialmente, la implementación recorría la tabla de procesos e iba comprobando si algún proceso estaba listo (`p->state == runnable`). Ahora le hemos añadido un nuevo nivel de prioridad, por lo que ningún proceso de prioridad normal puede ejcutarse antes que uno de alta prioridad, siempre que ambos estén listos a la vez. Esto lo hemos comprobado mediante otro bucle dentro: El primer bucle recoge un proceso listo, *best*. independientemente de su prioridad. Entonces, si se trata de uno de prioridad alta, debe ejecutarse inmediatamente; pero, si es de prioridad normal, volvemos a recorrer la tabla de procesos para buscar un proceso listo de prioridad alta, y se le asigna a *best*. Solamente si no se verifica esto último se ejecutará *best* el proceso listo que hemos elegido al principio. El código del `scheduler()` queda:

```
1. void
2. scheduler(void)
3. {
4.   struct proc *p;
5.   struct cpu *c = mycpu();
6.   c->proc = 0;
```

```
 7.
 8.   for(;;){
 9.     // Enable interrupts on this processor.
10.     sti();
11.
12.     struct proc *best = 0;
13.
14.     // Loop over process table looking for process to run.
15.     acquire(&ptable.lock);
16.     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
17.       if(p->state != RUNNABLE)
18.         continue;
19.       best = p;
20.
21.       if(best->priority != HI_PRIO)
22.       {
23.         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
24.           if(p->state == RUNNABLE && p->priority == HI_PRIO)
25.           {
26.             best = p;
27.             break;
28.           }
29.         }
30.       }
31.       p = best;
32.
33.       // Switch to chosen process.  It is the process's job
34.       // to release ptable.lock and then reacquire it
35.       // before jumping back to us.
36.       c->proc = p;
37.       switchuvm(p);
38.       p->state = RUNNING;
39.
40.       swtch(&(c->scheduler), p->context);
41.       switchkvm();
42.
43.       // Process is done running for now.
44.       // It should have changed its p->state before coming back.
45.       c->proc = 0;
46.     }
47.     release(&ptable.lock);
48.
49.   }
50. }
51.
```

Ejercicio 3.2. Añade a xv6 dos nuevas llamadas al sistema: `enum proc_prio getprio(int pid)` **y** `int setprio(int pid , enum proc_prio).`

Para añadir las 2 llamadas al sistema, hemos tomado como referencia la práctica del primer boletín, que tuvimos que añadir y modificar varias *syscalls*. Mostramos los cambios en algunos ficheros para realizar esta implementación:

- en usys.S, añadimos estas 2 líneas:

```
SYSCALL(getprio)
SYSCALL(setprio)
```

- en syscall.h asignamos un número a cada una de ellas:

```
#define SYS_getprio 24
#define SYS_setprio 25
```

- en user/user.h, hay que declarar las 2 funciones. También debemos definir el enumerado.

```
enum proc_prio { HI_PRIO=0, NORM_PRIO=1, ERR_PRIO=2 };
(...)
```

```
extern enum proc_prio getprio(int);
extern int setprio(int, enum proc_prio);
```

- en syscall.c, añadimos estas líneas:

```
extern int sys_getprio(void);
extern int sys_setprio(void);
(…)
[SYS_getprio] sys_getprio,
[SYS_setprio] sys_setprio,
```

Recordemos que estas 2 últimas líneas se insertaban dentro de la definición *syscall[]*.

- también en defs.h hemos tenido que añadirlas, junto al resto de funciones de proc.c:

```
enum proc_prio  getprio(int);
int setprio(int, enum proc_prio);
```

Junto a todo lo anterior, en sysproc.c realizamos las llamadas a las respectivas funciones.

```
int
sys_getprio(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;
  return getprio(pid);
}

int
sys_setprio(void)
{
  int pid, priority;

  if(argint(0, &pid) < 0 || argint(1, &priority) < 0)
    return -1;
  return setprio(pid, priority);
}

```

Para acabar, ya solo nos queda mostrar las funciones de las *syscalls*. Sendas implementaciones tienen estructura similar al resto de funciones implementadas aquí, como por ejemplo, `kill()`. En `getprio()` tan solo pedimos que se cierre el *lock* de la tabla de procesos, miramos si existe algún fichero con el entero *pid* que recibimos como parámetro, y devolvemos su prioridad. Si no existe ningún proceso con dicho identificador se devuelve `ERR_PRIO`, que indica que se ha producido un error. Por su parte, para implementar `setprio()` comprobamos si existe en la tabla de procesos y, si se cumple, le asignamos la prioridad que recibimos como parámetro, ejecutándose todo esto de forma atómica. Devuelve 0 sí se ha podido asignar perfectamente, y -1 si no existe el proceso.

```
enum proc_prio
getprio(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid == pid)
    {
      release(&ptable.lock);
```

```
12.       return p->priority;
13.     }
14.   }
15.   release(&ptable.lock);
16.   return ERR_PRIO;
17. }
18.
19. int
20. setprio(int pid, enum proc_prio priority)
21. {
22.   struct proc *p;
23.
24.   acquire(&ptable.lock);
25.   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
26.   {
27.     if(p->pid == pid)
28.     {
29.       p->priority = priority;
30.       release(&ptable.lock);
31.       return 0;
32.     }
33.   }
34.   release(&ptable.lock);
35.   return -1;
36. }
37.
```

Bibliografía

Información interesante relativa al ejercicio 2 (sbrk):

- https://www.cs.virginia.edu/~cr4bd/4414/S2019/paging-and-protection.html