Implementación xv6

Luis Alberto Mejía Troya

DNI:01889311E

luisalberto.mejiat@um.es

Paula Marín Turpín

DNI:48841926T

p.marinturpin@um.es

Profesor: Diego Sevilla Ruíz

Grupo: PCEO

Fecha: 12-12-2022

Universidad de Murcia

Facultad de Informática

Índice general

Capítulo 1

Introducción

A lo largo de este documento se explicarán detalladamente las modificaciones necesarias para llevar a cabo todos los ejercicios propuestos en esta práctica.

Se adjuntan capturas de pantalla de las líneas de código implementadas en los respectivos ficheros para dejar reflejadas de forma más visual y accesible las modificaciones.

Toda la práctica ha sido realizada con la ayuda de las transparecias de la asignatura, así como apoyándonos en la documentación de xv6. Véase la bibliografía para más detalle.

Capítulo 2

Llamadas al sistema en xv6

Antes de pasar a explicar las implementaciones específicas de cada llamada al sistema, desglosaremos los pasos previos necesarios para llevar a cabo la creación de las mismas.

2.1. Pasos previos

En primer lugar, es necesario crear un programa de usuario que realice la llamada al sistema, con el fin de poder comprobar que funcionan correctamente. Este archivo había sido proporcionado con el material para la práctica, por lo que solo han sido necesarias las modificaciones pertinentes para las nuevas llamadas.

Tras asegurar que las llamadas al sistema están incluidas en ese directorio, será necesario incluir en el **Makefile** de dicho directorio, en la variable **UPROGS** `'comando\'`, sustituyendo 'comando' por el oportuno en cada caso. De este modo, el programa de usuario estará disponible en el *shell* de `xv6` y podremos realizar las comprobaciones oportunas.

A continuación, en **syscall.h** le otorgamos un nuevo número a cada llamada para identificarla, siguiendo el formato `'SYS_syscall'` sustituyendo 'syscall' por el nombre de cada llamada.

Una vez realizado esto, debemos añadir la llamada a **user/usys.S** al final del fichero de la forma `SYSCALL(nombre llamada)`. Este fichero tiene una macro SYSCALL que acepta llamadas al sistema. Se define una etiqueta siguiendo el formato `'SYS_name'` con 'name' el nombre de la llamada, para así poder hacer referencia a ellas desde los programas como si fueran funciones. La macro mueve el contenido de `'SYS_name'` al registro `%eax` y realiza la interrupción necesaria.

Ahora tenemos que añadir la definición de la función de la nueva llamada al sistema para los programas de usuario en el directorio **user/user.h**. Con esto ya estaría todo listo para que los programas del S.O. puedan llamar a la nueva llamada.

Finalmente, en **syscall.c** hay que añadir la definición de las nuevas funciones de las llamadas al sistema, y dependiendo de la llamada al sistema realizaremos su implementación en un fichero u otro.

2.2. Ejercicio 1: `date`

2.2.1. Descripción:

El objetivo principal del ejercicio es que se observen las diferentes piezas que componen el funcionamiento de una llamada al sistema. La llamada `date` obtendrá el tiempo UTC actual y lo devolverá al programa usuario. En el desarrollo se usará la función `cmostime()` (definida en `lapic.c`) para leer el reloj en tiempo real, y en **`date.h`** está contenida la definición de la estructura `rtcdate` que debe pasarse a `cmostime()` como un puntero.

2.2.2. Código `sys_date`:

En el siguiente fragmento de código podemos apreciar la implementación de la llamada `sys_date(void)`. En primer lugar, se usa una variable puntero a una estructura `rtcdate` que recibe el nombre r". Esta estructura tiene varios campos como `day, month, year, hours`. mediante la comprobación de `argptr` extraemos de la pila de usuario el valor de `rtcdate` que hemos pasado en la llamada al sistema `date`. Si ha habido algún error a la hora de extraer el dato, retornaremos el valor -1 que indica que se ha producido un error.

Si hemos podido extraer el valor, realizaremos una llamada a `cmostime()` con el argumento que hemos extraído de la pila. Finalmente, devolveremos el valor 0 puesto que la llamada al sistema se ha realizado correctamente.

```
int
sys_date(void)
{
  struct rtcdate *r;
  // Comprobacion de parametros de entrada
  if ((argptr(INITIAL, (void**)&r, sizeof(struct rtcdate)))<0)
        return -1;
  // Llamada
  cmostime(r);
  return 0;
}
```

2.2.3. Código `user/date.c`:

Para la implementación del código de la llamada al sistema `date` realizamos un procedimiento similar al que tenemos en una maquina actual Linux. Es decir, si introducimos el comando `date` en la *shell*, nos mostrará el día seguido de la fecha y de la hora.

Hemos utilizado unas constantes que hemos almacenado en un array para saber el día en el que nos encontramos dada una fecha.

Esto hace que el código quede un poco más extenso pero realmente el orden d complejidad de este código sigue siendo de `O(1)` sin contar la llamada al sistema.

```
int main(int argc, char *argv[])
{
        char *month[12] = {"ene", "feb", "mar", "abr", "may", "jun", "jul", "ago", "sep
            ", "oct", "nov", "dic"};
        char *day[7] = {"dom", "lun", "mar", "mier", "jue", "vie", "sab"};
        // CONSTANTES PARA ANNOS NORMALES
        int r_m[12] = {0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5};
        //CONSTANTES PARA ANNOS BISIESTOS
        int b_m[12] = {6, 2, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5};
        //CONSTANTES DE SIGLO
        int siglo[4] = {6, 4, 2, 0};
        struct rtcdate r;

        int ks; //INDICE DE LA CONSTANTE DE SIGLO
        int d, m,a;     // VARIABLES PARA CALCULAR EL INDICE DEL ARRAY DAY: dia,
            constante de mes y los dos ultimos digitos del anho
        int calculate_day;
        if (date(&r))
        {
                printf(2,"date failed\n");
                exit();
        }

        ks = r.year%400;
        if (ks<100) ks = 0;
        else if (ks < 200) ks = 1;
        else if (ks<300) ks = 2;
        else ks = 3;
        d = r.day;
        if ((r.year % 4) == 0) m = b_m[r.month -1];
        else m = r_m[r.month -1];
        a = r.year % 100;
        calculate_day = (d + m + a + (a/4) + siglo[ks]) % 7;
        printf(1,"%s %d %s %d %d:%d:%d\n",day[calculate_day],r.day,month[r.month-1], r.
            year, r.hour, r.minute, r.second);
        exit();
}
```

Si introducimos el comando `date` en la terminal de sistema operativo `xv6` tenemos lo siguiente

```
$ date
vie 2 dic 2022 12:31:5
```

2.3. Ejercicio 2: dup2

2.3.1. Descripción

Para la realización de esta llamada al sistema, seguimos un procedimiento similar al de la llamada al sistema `date`.

El funcionamiento de `int dup2(int oldfd, int newfd)` se basa en convertir el descriptor `newfd` en una copia del `oldfd`. En el caso de que `newfd` estuviera abierto, se cierra antes de ser reusado.

En caso de que `oldfd` no sea un descriptor de fichero válido, la llamada al sistema termina y `newfd` no se cierra. Si `oldfd` es un descriptor de fichero válido, y `newfd` tiene el mismo valor que `oldfd`, `dup2` no hace nada y retorna `newfd`.

2.3.2. Código `sysfile.c`

En primer lugar, comprobamos que `oldfd` no esté cerrado, haciendo uso de `argfd`, con la que recuperamos el primer elemento de la pila (que será el propio `oldfd`) y se mete en el mismo. Al mismo tiempo, con esta función, comprobamos que el descriptor de fichero recuperado es correcto y que el fichero asociado a él también está abierto.

Seguidamente, usamos `argint` para recuperar el siguiente elemento de la pila de usuario, `newfd`. En este caso usamos esta función en lugar de `argfd` pues no es necesario que el descriptor esté abierto.

Además de ambas comprobaciones, es necesario que el `fd` sea válido, es decir, que no sea menor que 0 y no supere el número máximo de `fd` simultáneos: `NOFILE`, constante asignada al valor 16 y definida en `param.h`

Tras todas as comprobaciones, se accede a la tabla de ficheros, a la entrada correspondiente a `newfd`, para cerrar, si es necesario, el fichero asociado. Finalmente, asociamos a esta entrada la estructura `f`, es decir, duplicamos `oldfd` en `newfd`.

```
int
sys_dup2(void)
{
  struct file *f;
  int fd_old;
  int fd_new;
  // Extraemos el fichero y su descriptor actual
  if (argfd(0, &fd_old, &f) < 0) return -1;
  // Extraemos el descriptor nuevo
  if (argint(1, &fd_new) < 0) return -1;
  // Comprobamos que el descriptor sea correcto
  if(fd_new < 0 || fd_new > NOFILE) return -1;
  // Si son iguales, devolvemos directamente
  if (fd_new == fd_old) return fd_new;
  // Si esta abierto el fichero, lo cerramos
  if(myproc()->ofile[fd_new] != 0)
        fileclose(myproc()->ofile[fd_new]);
  // Le asignamos el fichero
  myproc()->ofile[fd_new] = f;
  // Llamamos a dup para la clonacion
  filedup(f);
  // Devolvemos el nuevo descriptor
  return fd_new;
}
```

2.4. Ejercicio 3: `exit` y `wait`

2.4.1. Descripción de `exit` y `wait`:

En este ejercicio debemos modificar ambas llamadas al sistema, proporcionadas por `xv6`, para que cumplan con la signatura del estándar `POSIX` en `xv6`.

El funcionamiento de `void exit(int status)` es bastante sencillo, pues causa la terminación normal del proceso y devuelve el byte menos significativo de `status` al proceso padre.

La función `pid_t wait(int *status)` se usa para esperar cambios de estado en un hijo del proceso que realiza la llamada, tales como si ha terminado, ha sido detenido o reanudado por una señal.

2.4.2. Código **sysproc.c**:

En primer lugar, tenemos que intercambiar todas las llamadas `exit()` por `exit(0)` y `wait()` por `wait(NULL)`. Seguidamente, debemos adaptar las declaraciones de dichas llamadas al sistema para que puedan aceptar el argumento entero o un puntero a entero, respectivamente. Estas declaraciones se encuentran en **user/user.h**.

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

A continuación, debemos adaptar las llamadas `sys_wait(void)` y `sys_exit(void)` del fichero **sysproc.c**. Primero, tenemos que extraer de la pila de usuario el valor del estado. Como en exit usamos un valor entero, usaremos la función `argint`. En wait tenemos un puntero a entero, y por lo tanto usaremos la función `argptr`.

Finalmente, en `sys_exit` llamamos a la función `exit` con el valor extraído de la pila y devolvemos 0 (pues tiene que devolver un valor entero pero su significado real es `nonreturn`). En `sys_wait` por otro lado, devolvemos el valor de retorno de `wait` si le pasamos como parámetro el valor extraído de la pila.

Como comentario adicional, destacar el desplazamiento aplicado en la llamada a `exit`. Esta modificación sirve como mecanismo para diferenciar cuando un proceso acaba por causas normales o cuando lo hace por alcanzar un trap.

```
int
sys_exit(void)
{
  int status;
  // Extraemos el estado de la pila de usuario
  if(argint(0, &status) < 0) return -1;

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

int
sys_wait(void)
{
  int* status;
  // Extraemos el estado de la pila de usuario
  if (argptr(INITIAL, (void**)&status, sizeof(int)) < 0)
    return -1;

  return wait(status);
}
```

2.4.3. Código **proc.c**:

exit:

En este caso solo ha sido necesario añadir la línea `curproc->status = status`, donde guardamos dentro de un parámetro de la estructura del proceso, el estado de salida del hijo. Es necesario hacerlo antes de que el padre despierte para evitar confusiones en las esperas entre padre e hijo.

Hemos tenido que añadir el parámetro `status` en la estructura `proc`, que se encarga de almacenar el estado del proceso.

```
void
exit(int status)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;
  ...
  acquire(&ptable.lock);

  // Asignamos el estado pasado como parametro a nuestro proceso
  curproc->status = status;

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);
  ...
}
```

wait:

Para el caso de `wait`, debemos tener en cuenta que el padre debe ser capaz de recuperar el estado de salida de sus hijos, por lo que el padre recorre toda la tabla de procesos y comprobando que él es el padre de todos. Una vez realizada esa comprobación, comprueba si el proceso está zombie y, en caso afirmativo, realiza una serie de acciones descritas en el código, entre las que hemos añadido la línea que asigna al puntero `status` pasado como parámetro el estado del proceso hijo.

```
int
wait(int *status)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        ...
        // Recuperamos el estado de salida del hijo
        *status = p->status;
        release(&ptable.lock);
        return pid;
      }
    }
    ....
  }
}
```

2.4.4. Código `trap.c`

Por último, debemos modificar el código de este fichero para que el parámetro que recibe el `exit` sea el número de interrupción. De este modo será posible saber qué tipo de error ha sido el que ha producido la llamada a la función. Sin embargo, para que esta llamada sea totalmente correcta y funcione respecto a la macro `WEXITTRAP`, también es necesario sumar 1 al valor del trap, para evitar el caso de que sea 0.

Es necesario hacer la modificación en todas las llamadas a `exit` en la función `trap`.

```
exit(0x01 + tf->trapno);
```

Capítulo 3

Reserva de páginas bajo demanda

3.1. Ejercicio 1 y 2:

Dado que el ejercicio 2 solo se trata de una modificación y corrección del ejercicio 1, hemos creído conveniente realizar una explicación continuada y conjunta de ambos como uno solo.

3.1.1. Descripción:

La reserva de páginas bajo demanda, o *lazy page allocation*, consiste en mapear en memoria principal únicamente las páginas que sean necesarias. Mapear, por otro lado, consiste en si queremos usar una página de memoria virtual, será necesario alocarla en la memoria física.

3.1.2. Código `sysproc.c`:

```
int
sys_sbrk(void)
{
  int addr;
  int n;
  // Extraemos de la pila el valor de n
  if(argint(0, &n) < 0)
    return -1;
  // Guardamos el tamanno actual del proceso
  addr = myproc()->sz;
  // Si n es negativa, desallocamos los bytes correspondientes a su valor
  if (n<0)
  {
    if(growproc(n) < 0)
      return -1;
  }
  // En otro caso aumentamos el tamanno del proceso
  else
  {
    myproc()->sz += n;
  }
  // Devolvemos el tamanno antiguo del proceso
  return addr;
}
```

La primera parte del ejercicio consiste en modificar la función `sys_sbrk()` para eliminar la reserva de páginas. En su lugar, si el entero `n` es mayor o igual a 0, incrementamos el tamaño del proceso en tantas unidades como indique `n`, y si es negativo, llamamos a `growproc()` para que desaloque los bytes de memoria correspondientes a ese valor.

3.1.3. Código `trap.c`:

Si compilamos hasta ahora, todo parece estar en orden, pero al ejecutar un simple 'echo hola' obtenemos el siguiente mensaje en la terminal:

```
$ echo hola
pid 3 sh : trap 14 err 6 on cpu 0 eip 0 x12f1 addr 0 x4004 - - kill proc
```

Esto sucede porque, al producirse el fallo de página, el sistema no sabe cómo reaccionar y lanza por pantalla el mensaje del trap por defecto. Para evitar que esto suceda, creamos una nueva entrada dentro de la estructura *switch* que corresponda con el error `T_PGFLT` o trap 14, que es el que obtenemos en este caso.

Si observamos el código del mensaje que se está imprimiendo por pantalla, vemos que la dirección donde se produce el error se obtiene con la función `rcr2()`, por lo que la usaremos para realizar las siguientes comprobaciones antes de pasar a resolver el error:

- Comprobamos si la dirección pertenece al KERNEL, es decir, si el proceso está intentando acceder a código protegido al que no tiene permiso de acceso.
- Comprobamos si la dirección se encuentra fuera del rango de direcciones del proceso, es decir, comprobamos si se sale del tamaño del proceso actual.
- Comprobamos si la dirección coincide con la página de guarda, a la que el usuario no tiene acceso.

Una vez realizadas estas comprobaciones, estamos seguros de que es un fallo de página normal y podemos pasar a reservar la página física necesaria para paliar el error. Tras reservarla, por precaución, la llenamos de ceros para eliminar cualquier dato residual.

Finalmente, realizamos el mapeo usando la función `mappages`. Al usar esta función, debemos acceder al fichero **`vm.c`** y eliminar de su definición el parámetro `static`.

```
int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
```

La función `mappages` recibe los siguientes parámetros:

- La tabla de páginas del proceso.
- Dirección virtual a mapear, es decir, la que ha producido el fallo.
- Tamaño de la página
- Dirección devuelta por `kalloc()`. Como la función la devuelve como dirección virtual, es necesario aplicarle `V2P` para pasarla a dirección física.
- Permisos de lectura, escritura y acceso para el usuario.

```
// trap 14
case T_PGFLT:
   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());
  char *mem;
  // Redondeamos la direccion
  uint page = PGROUNDDOWN(rcr2());

  // Si estamos en el  KERNEL -> ERROR
  if(page >= KERNBASE){
    cprintf("kernel addresses cannot be accessed\n");
    myproc()->killed = 1;
    break;
  }
  // Si excedemos el tamanno del proceso -> ERROR
  if(rcr2() >= myproc()->sz){
    cprintf("Unreserved memory access\n");
    myproc()->killed = 1;
    break;
  }
  // Si coincidimos con la pagina de guarda -> ERROR
  if(page == myproc()->page_guarda){
    cprintf("Page guarda cannot be accessed\n");
    myproc()->killed = 1;
    break;
  }
  // Reservamos la pagina fisica
  mem = kalloc();
  // Si devuelve 0 no ha sido posible devolver un marco fisico
  if(mem == 0){
    cprintf("kallocuvm out of memory\n");
    myproc()->killed = 1;
    break;
  }
  // En caso de exito, limpiamos la pagina
  memset(mem, 0, PGSIZE);
  //Mapeamos
  if(mappages(myproc()->pgdir, (char*)page, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
    cprintf("kallocuvm out of memory (2)\n");
    kfree(mem);
    myproc()->killed = 1;
    break;
  }
  break;
```

3.1.4. Últimas comprobaciones:

Finalmente, para completar la implementación, es necesario verificar que las llamadas a `fork()`, `exit()` y `wait()` se ejecutan sin problemas en el caso de que haya direcciones virtuales sin memoria reservada para ellas.

En el caso de las dos últimas llamadas no tenemos ningún problema, pues el *lazy page allocator* no les afecta. Sin embargo, en `fork()` debemos realizar algunas modificaciones.

En primer lugar, en **`proc.h`** modificamos la estructura `proc` para añadir el campo que representa la página de guarda.

```
struct proc {
  ...
  int status;                  // Process status (int)
  uint page_guarda;            // Process page guarda
};
```

A continuación, en **proc.c**, modificamos la función `fork()` para que el hijo obtenga la página de guarda del padre, pues ambos deben tener la misma estructura.

```
int
fork(void)
{
  ...
  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;
  // Copiamos el nuevo parametro en el hijo para que tenga la misma estructura que el
      padre
  np->page_guarda = curproc->page_guarda;
  ...
}
```

Finalmente, modificamos la función `copyuvm()` en el fichero **vm.c**, que se encarga de copiar la tabla de páginas del padre al hijo. Ahora, en lugar de lanzar un *panic* cuando no hay una página mapeada, ponemos un *continue*, pues la página no se mapeará hasta que no se produzca el fallo.

```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i, flags;
  char *mem;
  ....
  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))
      //panic("copyuvm: page not present");
      continue;
    ....
}
```

Capítulo 4

Procesos e hilos con prioridad alta

4.1. Ejercicio 1: Prioridad de los procesos

4.1.1. Descripción:

El objetivo de este ejercicio es conseguir que haya procesos con mayor prioridad que otros, es decir, que al ejecutarse tengan preferencia respecto al resto de procesos. Es importante tener en cuenta que `xv6` usa un mecanismo *Round-Robin*, pues eso influirá en cómo y cuando se ejecutan los procesos.

Round-Robin es un algoritmo de planificación de procesos, cuyo funcionamiento consiste en asignar a cada proceso una porción de tiempo equitativa y ordenada, tratando a todos los procesos con la misma prioridad. Cuando ese tiempo se acaba, la CPU pasa al siguiente proceso. Debemos conseguir que esto siga cumpliéndose pero dentro de cada nivel de prioridad.

4.1.2. Código `proc.h`:

En primer lugar, debemos declarar en **`proc.h`** un enumerado que defina los dos tipos de prioridad que tenemos: alta y normal.

```
enum proc_prio {HI_PRIO, NORM_PRIO};
```

Del mismo modo, para poder modificar la prioridad de un proceso, necesitamos añadir a la estructura del mismo un campo que indique con que prioridad estamos trabajando.

```
struct proc {
  ...
  int status;                  // Process status (int)
  uint page_guarda;            // Process page guarda
  enum proc_prio prio;         // Prioridad del proceso
};
```

4.1.3. Código `proc.c`:

Una vez declaradas las variables necesarias para llevar a cabo el ejercicio, procedemos con las modificaciones. En primer lugar, asignamos una prioridad por defecto (prioridad normal) en la creación del proceso, en la función `allocproc()`.

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  ....

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  // Asignamos al proceso la prioridad por defecto
  p->prio = NORM_PRIO;
  ...
  return p;
}
```

A continuación, es necesario modificar el proceso `fork()` para que el hijo herede la prioridad del padre cuando se cree, pues deben tener la misma estructura.

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  ...

  // Copiamos el nuevo parametro en el hijo para que tenga la misma estructura que el
      padre
  np->page_guarda = curproc->page_guarda;
  // Copiamos la prioridad del padre en el hijo
  np->prio = curproc->prio;
  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  ...
  return pid;
}
```

Finalmente, la parte crucial del ejercicio es la modificación de la función `scheduler()`, en la que se encuentra el bucle de ejecución de los procesos. Dentro del bucle infinito, añadimos un bucle `for` justo encima del que ya hay, y recorremos la tabla de procesos en busca de los que tienen mayor prioridad y están listos para ser ejecutados (*RUNNABLE*).

Además, para controlar que los procesos de prioridad normal solo se ejecuten cuando no haya procesos de prioridad alta, hemos incluido una variable entera que actúa como booleana: `prio`. Esta variable se inicializa a 0 al inicio del bucle, y en cuanto entra un proceso de prioridad alta, cambia su valor a 1. Así, asegurándonos de que los procesos de prioridad normal solo se ejecuten si el valor de dicha variable es 0, estaríamos consiguiendo el funcionamiento correcto esperado.

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  // Variable 'booleana' que controla la prioridad
  int prio;
  c->proc = 0;
  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    // inicializamos la variable a 0 -> prioridad normal
    prio = 0;
    // Recorremos la tabla buscanco los procesos de prioridad alta
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      // si no es ejecutable o no tiene prioridad alta, lo saltamos
      if(p->state != RUNNABLE || p->prio != HI_PRIO)
        continue;
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      // Se ejecutan procesos de prioridad alta y se cambia el valor de prio
      prio = 1;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();
      // Process is done running for now.
      // It should have changed its p->state before coming back.

      c->proc = 0;
    }
    // La prioridad normal no se ejecuta si hay procesos de prioridad alta pendientes
    if (prio == 0){
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
          continue;
        // Switch to chosen process.  It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        switchkvm();

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
      }
    }
    release(&ptable.lock);
  }
}
```

4.2. Ejercicio 2: Dos nuevas llamadas al sistema

4.2.1. Descripción:

En este ejercicio, se nos pide añadir las llamadas al sistema `enum proc_prio getprio(int pid)` y `int setprio(int pid , enum proc_prio)`. El funcionamiento de ambas es bastante intuitivo, pues la primera se encargará de devolver un objeto del tipo enumerado, informando de la prioridad del proceso que le pasemos como parámetro, y la segunda hace el proceso contrario, asigna al proceso pasado como parámetro la prioridad pasada como segundo parámetro. Vamos a proceder de forma similar a la seguida en el capítulo 2.

4.2.2. Código **sysproc.c**:

En primer lugar, debemos añadir las declaraciones de ambas llamadas al fichero **user/user.h**, junto al resto de llamadas.

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

Seguidamente, en el fichero **sysproc.c**, añadimos las llamadas `sys_getprio(void)` y `sys_setprio(void)`. En ambas, el primer paso es obtener de la pila el PID del proceso sobre el que queremos trabajar. A continuación, en `sys_getprio`, si el argumento se ha obtenido correctamente, llamamos a la función `getprio(pid)`. En el caso de `sys_setprio` será necesario además, obtener la prioridad que queremos asignar al proceso antes de poder llamar a `setprio(pid, prio)`.

```
int
sys_getprio(void)
{
  int pid;
  if (argint(INITIAL, &pid) < 0)
    return -1;
  return getprio(pid);
}

int
sys_setprio(void)
{
  int pid;
  int prio;

  if (argint(INITIAL, &pid) < 0)
    return -1;
  if (argint(1, &prio) < 0)
    return -1;
  setprio(pid, prio);
  return 0;
}
```

4.2.3. Código `proc.c`:

getprio:

Esta función devuelve un valor de tipo `enum proc_prio`, y recibe como parámetro el PID del proceso del que queremos obtener la prioridad. El cuerpo de la función es bastante sencillo: recorremos la tabla de procesos buscando el que coincida con el PID que nos han dado, y cuando lo encontremos devolvemos esa prioridad. Si no lo hemos encontrado, devolvemos -1.

```
enum proc_prio
getprio (int pid)
{
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->pid == pid)
        return p->prio;
  }
  return -1;
}
```

setprio:

En este caso, la función devuelve un entero y recibe como parámetro el PID del proceso cuya prioridad se quiere modificar y el nuevo valor de prioridad que le vamos a asignar. Como en la función anterior, el cuerpo es muy sencillo: recorremos la tabla en busca del proceso que coincida con el PID y, cuando lo encontramos, le asignamos la prioridad pasada como parámetro y devolvemos la prioridad del proceso tras el cambio. Si no hemos encontrado el proceso, devolvemos -1.

Cabe destacar, que pese a que la prioridad no es un entero sino un enumerado, al devolverla no se produce ninguna incompatibilidad con el hecho de que el tipo de retorno del proceso sea entero, pues cada elemento de un enumerado tiene asociado un valor entero. Si no se especifica, como es nuestro caso, cada componente del enumerado tiene asociado el entero que corresponde a su posición en el enumerado, como si se tratara de un *array*, comenzando por la posición 0. Por tanto, cuando la prioridad establecida sea alta, devolverá un 0, y cuando sea normal devolverá un 1.

```
int
setprio(int pid , enum proc_prio prio)
{
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->pid == pid){
        p->prio = prio;
        return p->prio;
      }
  }
  return -1;
}
```

Capítulo 5

Conclusiones y valoraciones personales

El desarrollo de esta práctica, aunque al principio se presentaba largo y tedioso, ha resultado gratificante y nos ha enseñado a apreciar mucho más el valor de los procesadores de hoy en día, que ya implementan todas las modificaciones que nosotros hemos implementado, entre otras muchas más.

Mediante la realización de los boletines, hemos ido descubriendo los entresijos de las llamadas a sistema, un funcionamiento bastante delicado y con muchos puntos que tratar con detenimiento, pues un pequeño descuido puede suponer un gran error. Sin embargo, esta necesidad de estar atentos a cada paso, nos ha ayudado a detectar errores con más facilidad y a saber cómo resolverlos en caso de que aparecieran.

Por otro lado, los ejercicios nos han servido sin lugar a duda para interiorizar los contenidos vistos en teoría, muy complejos y abstractos en los libros, y más familiares y claros tras la realización de esta práctica.