DOCUMENTACIÓN: XV6

Ampliación de sistemas operativos

Realizado por:

Carlos Cruzado Esteban - 49308284X - G2.1

Raúl Hernández Martínez - 24423648V - G2.1

Índice:

Boletín 2: Llamadas al sistema

Ejercicio 1 - date

Lo primero que debemos hacer para implementar la llamada al sistema de date es crear un fichero “date.c” con una parte del código ya proporcionada por el boletín y la estructura rtc (contenida en date.h), junto con un printf que debemos poner nosotros para que se imprima por la terminal la fecha actual y la hora actual. El fichero “date.c” finalmente queda asi:

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

int main(int argc, char *argv[]){
struct rtcdate r; // del "date.h"
if(date (&r)){
printf(2, "date failed\n");
exit (0) ;
}

printf(1, "Day:%d, Month:%d, Year:%d - Hour:%d, Min:%d, Sec:%d\n", r.day, r.month, r.year,
r.hour, r.minute, r.second);
exit (0) ;
}
```

Una vez hecho el “date.c” debemos modificar el Makefile situado en el directorio del usuario para que el fichero “date.c” pueda ser compilado y ejecutado, para ello en el Makefile habrá que agregar “date\” a la variable UPROGS, podemos observar que se ha introducido en la última línea del UPROGS, tal que así:

```
UPROGS=\
    cat\
    echo\
    forktest\
    grep\
    init\
    kill\
    ln\
    ls\
    mkdir\
    rm\
    sh\
    stressfs\
    usertests\
    wc\
    zombie\
    date\
```

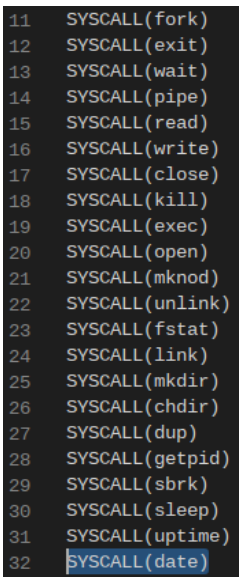
user/Makefile

Ahora nos toca definir la llamada al sistema de date, para ello debemos añadir un número a la llamada date en el fichero “syscall.h”, se quedara tal que así :

```
1  // System call numbers
2  #define SYS_fork    1
3  #define SYS_exit    2
4  #define SYS_wait    3
5  #define SYS_pipe    4
6  #define SYS_read    5
7  #define SYS_kill    6
8  #define SYS_exec    7
9  #define SYS_fstat   8
10 #define SYS_chdir   9
11 #define SYS_dup    10
12 #define SYS_getpid 11
13 #define SYS_sbrk   12
14 #define SYS_sleep  13
15 #define SYS_uptime 14
16 #define SYS_open   15
17 #define SYS_write  16
18 #define SYS_mknod  17
19 #define SYS_unlink 18
20 #define SYS_link   19
21 #define SYS_mkdir  20
22 #define SYS_close  21
23 #define SYS_date   22
```

syscall.h

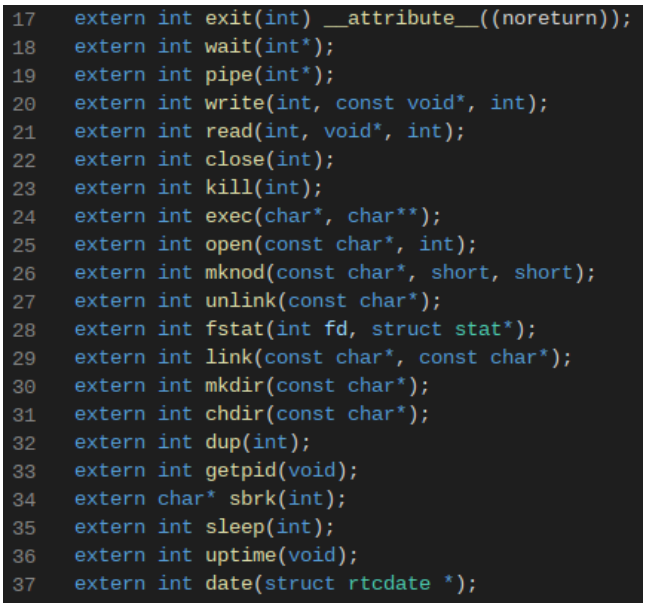
Después nos toca añadir al fichero usys.S situado en el directorio user, una SYSCALL(date), que nos sirve para que los programas de usuario invoquen llamadas al sistema desde el espacio del usuario. El fichero “usys.S” se queda tal que así:



```
11  SYSCALL(fork)
12  SYSCALL(exit)
13  SYSCALL(wait)
14  SYSCALL(pipe)
15  SYSCALL(read)
16  SYSCALL(write)
17  SYSCALL(close)
18  SYSCALL(kill)
19  SYSCALL(exec)
20  SYSCALL(open)
21  SYSCALL(mknod)
22  SYSCALL(unlink)
23  SYSCALL(fstat)
24  SYSCALL(link)
25  SYSCALL(mkdir)
26  SYSCALL(chdir)
27  SYSCALL(dup)
28  SYSCALL(getpid)
29  SYSCALL(sbrk)
30  SYSCALL(sleep)
31  SYSCALL(uptime)
32  SYSCALL(date)
```

usys.S

A continuación en “user.h” , donde se definen las librerías y las llamadas al sistema para el uso de los usuarios, debemos definir la llamada al sistema de date y con esto su definición se acaba. El fichero “user.h”, situado en el directorio user, se queda tal que así:



```
17  extern int exit(int) __attribute__((noreturn));
18  extern int wait(int*);
19  extern int pipe(int*);
20  extern int write(int, const void*, int);
21  extern int read(int, void*, int);
22  extern int close(int);
23  extern int kill(int);
24  extern int exec(char*, char**);
25  extern int open(const char*, int);
26  extern int mknod(const char*, short, short);
27  extern int unlink(const char*);
28  extern int fstat(int fd, struct stat*);
29  extern int link(const char*, const char*);
30  extern int mkdir(const char*);
31  extern int chdir(const char*);
32  extern int dup(int);
33  extern int getpid(void);
34  extern char* sbrk(int);
35  extern int sleep(int);
36  extern int uptime(void);
37  extern int date(struct rtcdate *);
```

user.h

Una vez definida la llamada al sistema debemos añadir por último su funcionalidad como llamada al sistema, para ello debemos añadir en el “syscall.c”, el cual es responsable de implementar el manejo de llamadas al sistema. La definición de la función sys_date(), a continuación se muestra el fichero modificado:

```
85  extern int sys_chdir(void);
86  extern int sys_close(void);
87  extern int sys_dup(void);
88  extern int sys_exec(void);
89  extern int sys_exit(void);
90  extern int sys_fork(void);
91  extern int sys_fstat(void);
92  extern int sys_getpid(void);
93  extern int sys_kill(void);
94  extern int sys_link(void);
95  extern int sys_mkdir(void);
96  extern int sys_mknod(void);
97  extern int sys_open(void);
98  extern int sys_pipe(void);
99  extern int sys_read(void);
100 extern int sys_sbrk(void);
101 extern int sys_sleep(void);
102 extern int sys_unlink(void);
103 extern int sys_wait(void);
104 extern int sys_write(void);
105 extern int sys_uptime(void);
106 extern int sys_date(void);
```

syscall.c

```
112 [SYS_fork]    sys_fork,
113 [SYS_exit]    sys_exit,
114 [SYS_wait]    sys_wait,
115 [SYS_pipe]    sys_pipe,
116 [SYS_read]    sys_read,
117 [SYS_kill]    sys_kill,
118 [SYS_exec]    sys_exec,
119 [SYS_fstat]   sys_fstat,
120 [SYS_chdir]   sys_chdir,
121 [SYS_dup]     sys_dup,
122 [SYS_getpid]  sys_getpid,
123 [SYS_sbrk]    sys_sbrk,
124 [SYS_sleep]   sys_sleep,
125 [SYS_uptime]  sys_uptime,
126 [SYS_open]    sys_open,
127 [SYS_write]   sys_write,
128 [SYS_mknod]   sys_mknod,
129 [SYS_unlink]  sys_unlink,
130 [SYS_link]    sys_link,
131 [SYS_mkdir]   sys_mkdir,
132 [SYS_close]   sys_close,
133 [SYS_date]    sys_date,
```

syscall.c

Y por último, en “sysproc.c” debemos dotar de funcionalidad a nuestra función sys_date(), la cual es la llamada al sistema de date. El fichero “sysproc.c” contiene las implementaciones de las llamadas al sistema. Para implementar la función deberemos recoger de la pila la estructura rtcdate *r, y con r podremos hacer la llamada cmostime como se indica en las diapositivas. A “sysproc.c” se le implementa la siguiente función tal que así:

```
int
sys_date(void)
{
// Recoger el parámetro struct rtcdate* de la primera posición de la pila.
struct rtcdate *r;

// Comprobar todos los errores y retornar -1 en caso de error.
if(argptr(0, (void ** ) &r, sizeof(struct rtcdate)) < 0){
return -1;
}

// Llamar a la función cmostime() con ese puntero para obtener la fecha.
cmostime(r);
return 0;
}
```

sysproc.c

Ejercicio 2 - dup2()

Para su implementación tuvimos que hacer recuerdo de cómo era el funcionamiento de dup(), para ello tuvimos que investigar y leer su funcionamiento y los parámetros que necesitaba para ser implementado. Tras la investigación nos quedó claro que dup2() permite duplicar un descriptor de archivo específico en un número de descriptor de archivo deseado, siendo su sintaxis así:

int dup2(int oldfd, int newfd)

Una vez tenemos claro cómo funciona dup2, debemos definir la llamada al sistema, eso previamente lo hemos visto en el ejercicio anterior, ya que solo deberemos modificar “syscall.h”, “usys.S”, “user.h”, a continuación muestro como quedaron los ficheros nombrados ya:

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_date   22
#define SYS_dup2   23
```

syscall.h

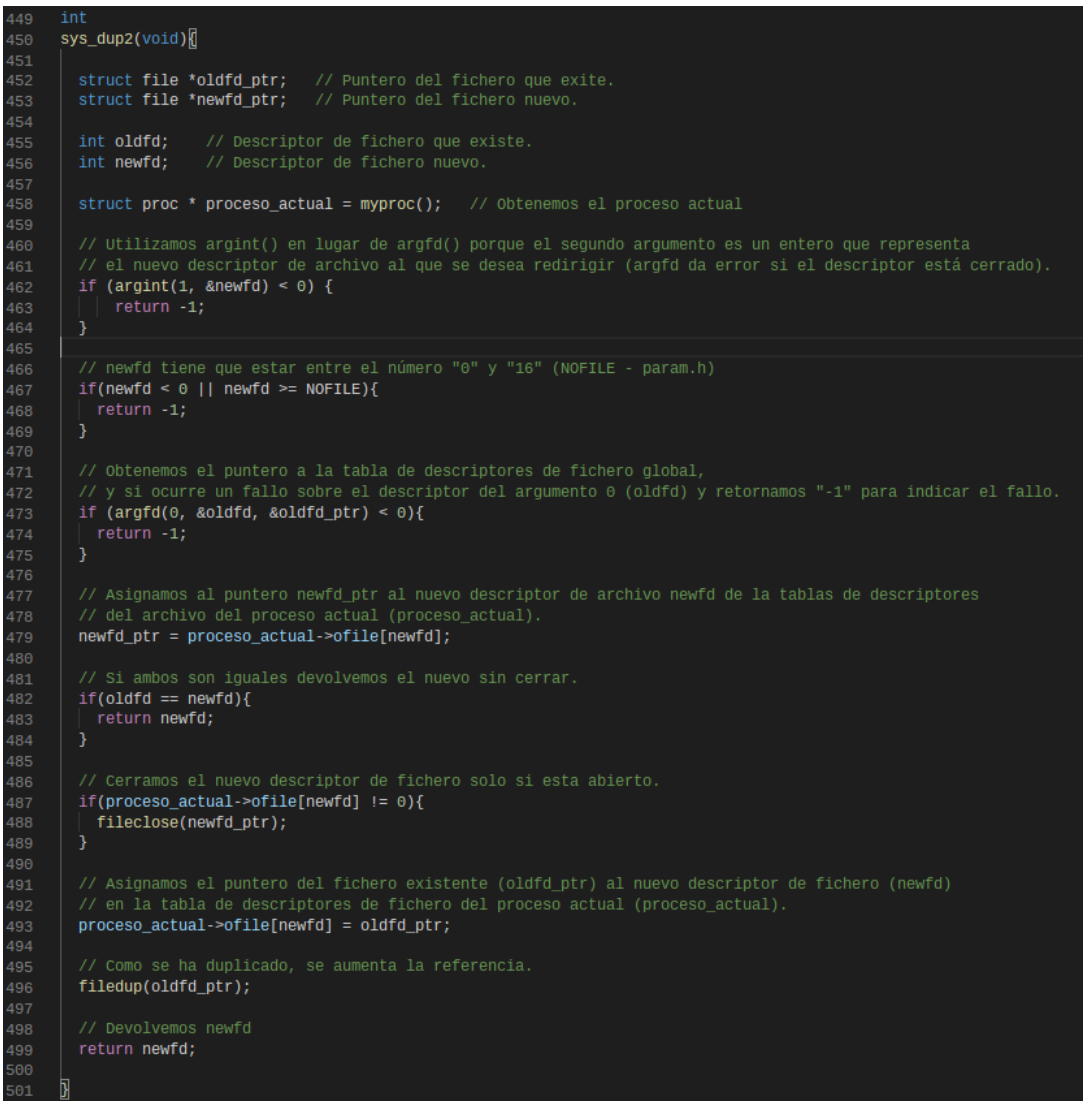
```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(date)
SYSCALL(dup2)
```

usys.S

```
17  extern int exit(int) __attribute__((noreturn));
18  extern int wait(int*);
19  extern int pipe(int*);
20  extern int write(int, const void*, int);
21  extern int read(int, void*, int);
22  extern int close(int);
23  extern int kill(int);
24  extern int exec(char*, char**);
25  extern int open(const char*, int);
26  extern int mknod(const char*, short, short);
27  extern int unlink(const char*);
28  extern int fstat(int fd, struct stat*);
29  extern int link(const char*, const char*);
30  extern int mkdir(const char*);
31  extern int chdir(const char*);
32  extern int dup(int);
33  extern int getpid(void);
34  extern char* sbrk(int);
35  extern int sleep(int);
36  extern int uptime(void);
37  extern int date(struct rtcdate *);
38  extern int dup2(int, int);
```

user.h

Una vez definida ya dup2() como llamada al sistema habrá que dotarla de funcionalidad, para ello debemos añadirle la funcionalidad en el fichero “sysfile.c”, en este fichero se hacen las llamadas a sistemas que tienen que ver con los ficheros. En “sysfile.c” queda dup2() definida así:



```
449  int
450  sys_dup2(void){
451
452    struct file *oldfd_ptr;   // Puntero del fichero que exite.
453    struct file *newfd_ptr;   // Puntero del fichero nuevo.
454
455    int oldfd;    // Descriptor de fichero que existe.
456    int newfd;    // Descriptor de fichero nuevo.
457
458    struct proc * proceso_actual = myproc();   // Obtenemos el proceso actual
459
460    // Utilizamos argint() en lugar de argfd() porque el segundo argumento es un entero que representa
461    // el nuevo descriptor de archivo al que se desea redirigir (argfd da error si el descriptor está cerrado).
462    if (argint(1, &newfd) < 0) {
463        return -1;
464    }
465
466    // newfd tiene que estar entre el número "0" y "16" (NOFILE - param.h)
467    if(newfd < 0 || newfd >= NOFILE){
468      return -1;
469    }
470
471    // Obtenemos el puntero a la tabla de descriptores de fichero global,
472    // y si ocurre un fallo sobre el descriptor del argumento 0 (oldfd) y retornamos "-1" para indicar el fallo.
473    if (argfd(0, &oldfd, &oldfd_ptr) < 0){
474      return -1;
475    }
476
477    // Asignamos al puntero newfd_ptr al nuevo descriptor de archivo newfd de la tablas de descriptores
478    // del archivo del proceso actual (proceso_actual).
479    newfd_ptr = proceso_actual->ofile[newfd];
480
481    // Si ambos son iguales devolvemos el nuevo sin cerrar.
482    if(oldfd == newfd){
483      return newfd;
484    }
485
486    // Cerramos el nuevo descriptor de fichero solo si esta abierto.
487    if(proceso_actual->ofile[newfd] != 0){
488      fileclose(newfd_ptr);
489    }
490
491    // Asignamos el puntero del fichero existente (oldfd_ptr) al nuevo descriptor de fichero (newfd)
492    // en la tabla de descriptores de fichero del proceso actual (proceso_actual).
493    proceso_actual->ofile[newfd] = oldfd_ptr;
494
495    // Como se ha duplicado, se aumenta la referencia.
496    filedup(oldfd_ptr);
497
498    // Devolvemos newfd
499    return newfd;
500
501  }
```

sysfile.c

Como hemos dicho el funcionamiento de dup() es similar al de dup2(), pero este último podemos especificar el valor del nuevo descriptor del fichero (newfd). Para conseguir este la funcionalidad hemos seguido estos pasos:

1. Obtener mediante la función argint() el nuevo descriptor de archivo al que se desea duplicar el descriptor existente. Además obtenemos el puntero a la tabla de descriptores del fichero global con la función argfd(). Además argfd() devolverá error cuando el newfd está cerrado, lo cual es algo que no queremos que pase.
2. Comprobamos que newfd es válido.
3. Deberemos comprobar que el descriptor de ficheros de oldfd y newfd es igual para devolver newfd sin cerrarlo.
4. En caso de que newfd y oldfd no sean iguales se cierra newfd.
5. Y por último, se llama a la función filedup() como en dup().

Ejercicio 3 - exit() and wait()

Para realizar este ejercicio tuvimos que modificar los exit() que había por defecto ya en el proyecto y ponerlos a exit(0), y los wait() se modifican a wait(NULL), esto se hace con los dos siguientes comandos:

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

A continuación debemos adaptar el fichero "user.h" del directorio user para aceptar los argumentos para el exit() y el wait(), quedando así el fichero:

```
17  extern int exit(int) __attribute__((noreturn));
18  extern int wait(int*);
19  extern int pipe(int*);
```

user.h

Debemos también modificar las funciones de implementación de las llamadas dentro del núcleo en "sysproc.c", quedando asi:

```
21  int
22  sys_exit(void)
23  {
24    int estado;
25
26    if(argint(0, &estado) < 0){
27      panic("exit");
28    }
29
30    // Desplazamos el estado 8 bytes para cumplir con los macros que se proporcionan en las diapositivas.
31    // Si no se añade da "failure" en el test.
32    exit(estado << 8);
33    return 0;  // not reached
34  }
35
36  int
37  sys_wait(void)
38  {
39    int * estado;
40
41    if(argptr(0, (void **)&estado, sizeof(int)) < 0){
42      panic("wait");
43    }
44
45    return wait(estado);
46  }
```

sysproc.c

Observamos que en el sys_exit() se usa la función argint(0, &estado) para obtener el estado de salida del proceso a través del primer argumento pasado a la función. Si hay un error al

obtener el estado, se llama a la función panic, que se usa para manejar situaciones de error críticas en el sistema operativo. Luego, la función exit(estado << 8) se llama para finalizar el proceso con el estado proporcionado desplazando hacia la izquierda 8 bits el estado, esto se hace para poder cumplir con el estándar de posix de los macros que se proporcionan en el boletín.

Siendo más minuciosos, en el sys_wait() se usa un puntero estado para almacenar el estado de un proceso hijo al finalizar. Usa la funcion argptr(0, (void **)&estado, sizeof(int)) para obtener el puntero al estado de salida desde el primer argumento pasado a la función. Si hay un error se llama a la función panic. Luego, la función wait(estado) se llama para esperar a que un proceso hijo finalice y almacenar su estado de salida en la ubicación de memoria proporcionada por estado. Finalmente, la función devuelve el resultado de wait(estado), que generalmente es el identificador del proceso hijo que ha finalizado.

Ahora nos falta almacenar qué wait() pueda recoger el mismo valor que exit() en cuanto a su estado, para ello debemos hacer los siguiente:

1. Añadir un atributo exit_status a la estructura proc en “proc.h”

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int estado_exit;             // Hara de estado de salida del proceso
  int pagina_de_guarda;        // Para manejar el caso de fallos en la página inválida debajo de la pila
  enum proc_prio priority;     //TO-DO prioridad del proceso
};
```

proc.h

2. Modificar las funciones exit() y wait() ubicadas en “proc.c” tal que, exit() guarda el valor del estado de salida y seguidamente lo almacenará en exit_status y wait() recogerá dicho valor.

```
// Guardamos en el proceso actual el estado (en la variable que hemos creado "estado_exit")
curproc->estado_exit = estado;
```

función exit(int estado) de ”proc.c”

```
// Recogemos el valor del estado del proceso.
if(estado != 0){
  * estado = p->estado_exit;
}
```

función wait(int * estado) de “proc.c”

Ahora nos toca modificar “sh.c” para que muestre por pantalla el código de finalización de los procesos:

```
/*Modificad la llamada al sistema wait() del shell (sh.c) en su función main()
  para que cada vez que ejecute un programa produzca una salida:
    Output code : N*/

wait(&estado);

// Si el proceso ha terminado muestra el codigo del proceso
if(WIFEXITED(estado)){
  // Definicion que se ha puesto en "user/user.h"
  printf(1, "Codigo de salida: %d\n", WEXITSTATUS(estado));
}
// Si el proceso ha terminado mediante una señal se muestra el "trap" del proceso para saber el motivo
else if(WIFSIGNALED(estado)){
  printf(1, "Trap correspondiente: %d\n", WEXITTRAP(estado));
}
```

función main() de “sh.c”

Hay que destacar que aún no hemos acabado, pues el número del trap puede ser 0, con lo que habrá que aumentar su valor en el fichero “trap.c”.

```
// Force process exit if it has been killed and is in user space.
// (If it is still executing in the kernel, let it keep running
// until it gets to the regular system call return.)

// Sumamos +1 al trap
if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
  exit(tf->trapno + 1);

// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();

// Check if the process has been killed since we yielded

// Sumamos +1 al trap
if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
  exit(tf->trapno + 1);
```

final del fichero “trap.c”

Se realiza así para poder cumplir con los estándares de POSIX al añadir las macros. El problema de este ejercicio surge a la hora de cuando acaba un proceso con trap 0 , el sistema lo puede confundir con un éxito 0, y esto lo solucionamos incrementando a 1.

Boletín 3: El sistema de memoria de xv6

Ejercicio 1 - Implementación de reserva diferida

Tenemos que reemplazar la reserva directa utilizando la llamada a “sbrk()” del fichero “sysproc.c” por un enfoque de reserva diferida, donde las páginas no se reservan de inmediato sino hasta que la aplicación las necesite. Esta estrategia es más eficiente que la anterior, ya que evita la reserva anticipada de páginas en la propia llamada a “sbrk()”.

Eliminamos la reserva de memoría pero aumentamos el tamaño del proceso.

```
79  // La nueva función debe incrementar el tamaño del proceso (proc->sz) y devolver
80  // el tamaño antiguo pero no debe reservar memoria
81
82  if(n >= 0){
83    myproc()->sz = myproc()->sz + n;
84  }
```

```
91  /*if(growproc(n) < 0)
92    return -1;*/
```

“sys_sbrk” de sysproc.c

A continuación, tecleamos “echo hola”, y podemos ver que el mensaje que sale proviene del manejador de traps, definido en trap.c. Ha capturado un fallo de página que el propio kernel no sabe manejar.

```
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ echo hola
pid 3 sh: trap 14 err 6 on cpu 1 eip 0xef2 addr 0x4004--kill proc
Trap correspondiente: 14
$ _
```

Para poder responder al fallo de página en el espacio de usuario mapeando una nueva página física en la dirección que generó el fallo y regresar después al espacio de usuario

para que el proceso continúe, hemos añadido un nuevo “case” (T_PGFLT) en la función trap para que detecte el fallo de página.

Usamos “PGROUNDDOWN( )” para redondear la dirección virtual a límite de página. Obtenemos una página en memoria sacada de la memoría física disponible con “kalloc( )”. Y por último mapeamos las nuevas páginas con “mappages( )”. En el siguiente ejercicio se muestra una imagen con el código teniendo en cuenta la corrección de las situaciones mostradas en el ejercicio 2.

Ejercicio 2 - Corrección de situaciones del ejercicio 1

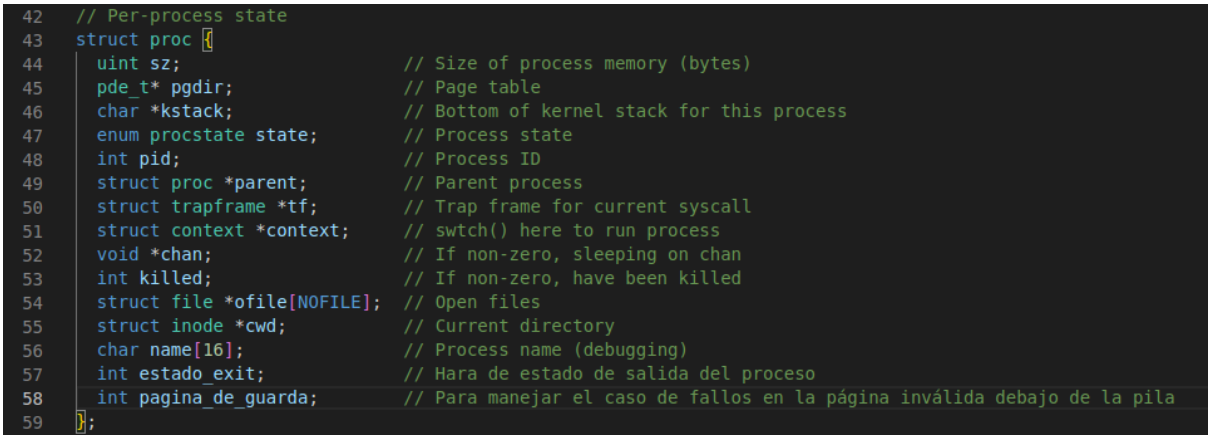
En el boletín comenta que la implementación realizada en el ejercicio 1 no es totalmente correcta ya que no contempla varias situaciones.

Una de ellas es el caso de un argumento negativo al llamarse a “sbrk( )”. Para ello, tenemos que permitir que la llamada “growproc( )” se pueda ejecutar cuando el número de bytes (“n”) sea inferior a cero.

```
68  int
69  sys_sbrk(void)
70  {
71    int addr;
72    int n; // -> número de bytes
73
74    if(argint(0, &n) < 0)
75      return -1;
76
77    addr = myproc()->sz;
78
79    // La nueva función debe incrementar el tamaño del proceso (proc->sz) y devolver
80    // el tamaño antiguo pero no debe reservar memoria
81
82    if(n >= 0){
83      myproc()->sz = myproc()->sz + n;
84    }
85    // No se debe llamar a growproc() en caso de que el proceso crezca
86    else{
87      if(growproc(n) < 0){
88        return -1;
89      }
90    }
91    /*if(growproc(n) < 0)
92      return -1;*/
93
94    return addr;
95  }
```

“sys_sbrk” de sysproc.c

Otra situación es manejar el caso de fallo en la página inválida debajo de la pila. Tenemos que crear en la estructura del proceso una página de guarda para poder almacenar la dirección.



```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int estado_exit;             // Hara de estado de salida del proceso
  int pagina_de_guarda;        // Para manejar el caso de fallos en la página inválida debajo de la pila
};
```

proc.h

Para la situación de verificar que fork() y exit()/wait() funciona en el caso de que haya direcciones virtuales sin memoria reservada para ellas, hemos tenido que tocar también dentro del archivo “vm.c”, es decir, hemos tenido que eliminar los panic que contiene el método “copyuvm( )” para que se permita la reserva de páginas bajo demanda.

El motivo es porque, puede pasar que el proceso no tenga mapeadas entradas de la tabla pero si estén reservadas.

```
319  pde_t*
320  copyuvm(pde_t *pgdir, uint sz)
321  {
322    pde_t *d;
323    pte_t *pte;
324    uint pa, i, flags;
325    char *mem;
326
327    if((d = setupkvm()) == 0)
328      return 0;
329    // Se recorre la tabla de páginas de padre
330    for(i = 0; i < sz; i += PGSIZE){
331
332      // Comentamos el código del panic para que no salte, ya que el proceso
333      // no tendra mapeadas entradas de la tabla pero estaran reservadas.
334
335      // [si comentamos también los "if" da error, por lo que comentamos solo los panic y ponemos un "continue"]
336
337      if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
338        //panic("copyuvm: pte should exist");
339        continue;
340      if(!(*pte & PTE_P))
341        //panic("copyuvm: page not present");
342        continue;
343      pa = PTE_ADDR(*pte);
344      flags = PTE_FLAGS(*pte);
345      if((mem = kalloc()) == 0)
346        goto bad;
347      memmove(mem, (char*)P2V(pa), PGSIZE);
348      if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
349        kfree(mem);
350        goto bad;
351      }
352    }
353    return d;
354
355  bad:
356    freevm(d, 1);
357    return 0;
358  }
```

“copyuvm” de vm.c

Por último, para la situación de asegurarse de que funciona el uso por parte del kernel de páginas de usuario que todavía no han sido reservadas hemos implementado en “trap.c” la siguiente condición, ya que la dirección de fallo no puede ser mayor que el tamaño del proceso.

```
// Extraemos la dirección de fallo
uint direccion_fallo = rcr2();

// La direccion de fallo no tiene que sobrepasar el tamaño que ya incrementamos en la llamada a sbrk()
if(direccion_fallo >= myproc()->sz){
  myproc()->killed = 1;
  break;
}
```

trap.c

El código de trap.c completo con la corrección de las situaciones es el siguiente:

```
case T_PGFLT:
  cprintf("Entra al case T_PGFLT\n");

  // Extraemos la dirección de fallo
  uint direccion_fallo = rcr2();

  // La direccion de fallo no tiene que sobrepasar el tamaño que ya incrementamos en la llamada a sbrk()
  if(direccion_fallo >= myproc()->sz){
    myproc()->killed = 1;
    break;
  }

  // Usa PGROUNDDOWN(va) para redondear la dirección virtual a límite de página
  // Hay que comprobar de que el fallo de página no es por la página de guarda.
  if (PGROUNDDOWN(rcr2()) == myproc()->pagina_de_guarda){
    myproc()->killed = 1;
    break;
  }

  // Redondeamos la dirección virtual al límite de página
  uint pagina_fallo = PGROUNDDOWN(direccion_fallo);

  // Asignamos una nueva página física desde la memoria libre
  char* p = kalloc();

  if(p == 0){
    // Comprobamos que tengamos memoria libre disponible, y sino salimos del proceso.
    exit(-1);
  }

  // Limpiamos la página asignada
  memset(p, 0, PGSIZE);

  // Mapeamos las nuevas páginas del proceso
  // "V2p" nos traduce la dirección virtual a física
  if(mappages(myproc()->pgdir, (char*)pagina_fallo, PGSIZE, V2P(p), PTE_W|PTE_U) < 0){
    // Si falla el mapeo, liberar la página asignada, marcar el proceso como terminado y salir del ciclo
    kfree(p);
    myproc()->killed = 1;
    break;
  }

  break;
```

trap.c

Boletín 4: Planificación de procesos y procesos de prioridad alta

Ejercicio 1 - Mecanismo de primero mayor prioridad y despues los demas

Para poder realizar este ejercicio primero deberemos crear un enumerado con HI_PRIO y NORM_PRIO, para ello creamos el fichero “priority.h” para declarar el enumerado:

```
1  #ifndef PRIORITY_H
2  #define PRIORITY_H
3
4  enum proc_prio { NORM_PRIO, HI_PRIO };
5
6  #endif
```

“priority.h”

Una vez creado este fichero deberemos añadir una variable de tipo enumerado a la estructura del proceso, que se sitúa en el fichero “proc.h”

```
42  // Per-process state
43  struct proc {
44    uint sz;                     // Size of process memory (bytes)
45    pde_t* pgdir;                // Page table
46    char *kstack;                // Bottom of kernel stack for this process
47    enum procstate state;        // Process state
48    int pid;                     // Process ID
49    struct proc *parent;         // Parent process
50    struct trapframe *tf;        // Trap frame for current syscall
51    struct context *context;     // swtch() here to run process
52    void *chan;                  // If non-zero, sleeping on chan
53    int killed;                  // If non-zero, have been killed
54    struct file *ofile[NOFILE];  // Open files
55    struct inode *cwd;           // Current directory
56    char name[16];               // Process name (debugging)
57    int estado_exit;             // Hara de estado de salida del proceso
58    int pagina_de_guarda;        // Para manejar el caso de fallos en la página inválida debajo de la pila
59    enum proc_prio priority;     //TO-DO prioridad del proceso
60  };
```

“proc.h”

Una vez declarado, sabemos que los procesos serán creados con una prioridad normal (NORM_PRIO) para ello deberemos ir a la función allocproc() “proc.c” y modificarla:

```
89  found:
90    p->state = EMBRYO;
91    p->pid = nextpid++;
92    p->priority=NORM_PRIO;    //TO-DO: al crear un proceso le asignamos una prioridad normal
```

función allocproc() del fichero “proc.c”

Ahora también debemos implementar la herencia de la prioridad del padre para que también el hijo tenga la misma prioridad, eso se hace en el mismo fichero mencionado anteriormente, pero en la función fork(), y debemos modificarla:

```
201    np->sz = curproc->sz;
202    np->parent = curproc;
203    np->priority = curproc->priority;   //TO-DO:el proceso np hereda la prioridad del proceso padre
204    *np->tf = *curproc->tf;
```

función fork() del fichero “proc.c”

Y por último debemos modificar la función del scheduler() para poder hacer que se ejecuten antes procesos con prioridad HI_PRIO que NORMAL_PRIO, para ello deberemos buscar en la tabla de procesos (ptable) los procesos con HI_PRIO y ejecutarlos antes que los procesos con NORMAL_PRIO. Sabemos que la equidad entre procesos de misma prioridad no se cumple, imaginemos que entran todo el rato procesos con prioridad HI_PRIO, en nuestro caso se ejecutarán los procesos de HI_PRIO uno a uno como también sería en el caso de NORMAL_PRIO, pues esto en parte es ineficiente pues habrá procesos que necesiten menos tiempo para ejecutarse que el primero en un caso hipotético. Con lo cual, resolver los procesos en nuestro caso nos llevará más tiempo. Para implementar la equidad entre procesos de misma prioridad pensé en hacer ejecutar estos procesos unos segundos cada uno asi hasta que se termine los procesos de esa prioridad, pero al implementarlo me daban bugs que no supe arreglarlos es por ello que no está plasmado en el código, pero si se pensó.

```
339  void
340  scheduler(void)
341  {
342    struct proc *p;
343    struct cpu *c = mycpu();
344    struct proc *aux;
345    struct proc *proc_temp;
346    c->proc = 0;
347
348    for(;;){
349      // Enable interrupts on this processor.
350      sti();
351
352      // Loop over process table looking for process to run.
353      acquire(&ptable.lock);
354
355      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
356        if(p->state != RUNNABLE)
357          continue;
358        // Switch to chosen process.  It is the process's job
359        // to release ptable.lock and then reacquire it
360        // before jumping back to us.
361
362        //guardo el proceso en una variable auxiliar
363        aux=p;
364
365        //Busco otro proceso (proc_temp) por si hay alguno con una prioridad superior al aux
366        for(proc_temp =ptable.proc; proc_temp < &ptable.proc[NPROC]; proc_temp++){
367          if(proc_temp->state != RUNNABLE)
368          continue;
369
370          if(proc_temp->priority==HI_PRIO && aux->priority==NORM_PRIO){
371            //aux coge el procesador con mas prioridad, para asi asegurarnos que el de mas prioridad se ejecuta antes
372            aux=proc_temp;
373            break;
374          }
375
376        }
377
378        //guardo en p el proceso mas alto que se ha encontrado
379        p=aux;
380
381        c->proc = p;
382        switchuvm(p);
383        p->state = RUNNING;
384
385        swtch(&(c->scheduler), p->context);
386        switchkvm();
387
388        // Process is done running for now.
389        // It should have changed its p->state before coming back.
390        c->proc = 0;
391      }
392      release(&ptable.lock);
393
394    }
395  }
```

función scheduler() del fichero “proc.c”

Ejercicio 2 - getprio() y setprio()

Para implementar estas dos nuevas llamadas al sistema debemos definirlas previamente como hemos visto anteriormente, habrá que modificar “syscall.h” , “usys.S” y “user.h” , muestro a continuación como se quedan los ficheros modificados:

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_date   22
#define SYS_dup2   23
#define SYS_getprio 24
#define SYS_setprio 25
```

syscall.h

```
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)
SYSCALL(date)
SYSCALL(dup2)
SYSCALL(getprio)
SYSCALL(setprio)
```

usys.S

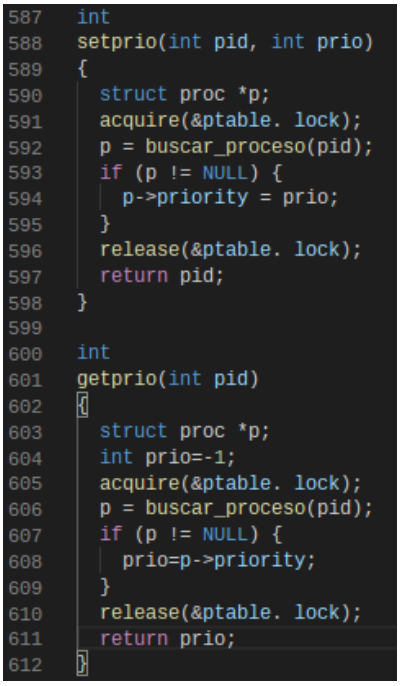
```
15  // Las funciones de llamada al sistema (en user/user.h) se tienen que adaptar
16  // para aceptar los argumentos
17  extern int exit(int) __attribute__((noreturn));
18  extern int wait(int*);
19  extern int pipe(int*);
20  extern int write(int, const void*, int);
21  extern int read(int, void*, int);
22  extern int close(int);
23  extern int kill(int);
24  extern int exec(char*, char**);
25  extern int open(const char*, int);
26  extern int mknod(const char*, short, short);
27  extern int unlink(const char*);
28  extern int fstat(int fd, struct stat*);
29  extern int link(const char*, const char*);
30  extern int mkdir(const char*);
31  extern int chdir(const char*);
32  extern int dup(int);
33  extern int getpid(void);
34  extern char* sbrk(int);
35  extern int sleep(int);
36  extern int uptime(void);
37  extern int date(struct rtcdate *);
38  extern int dup2(int, int);
39  extern enum proc_prio getprio (int prio);
40  extern int setprio (int pid, enum proc_prio);
```

user.h

Ahora en “proc.h” declaro la función setprio() y getprio() para poder implementar la funcionalidad de la función en “proc.c”, cuyo archivo es esencial para la administración de procesos en xv6 y proporciona la base para la ejecución y el control de los procesos en el sistema operativo. A continuación muestro su implementación:

```
37  //TO-DO:Declaracion de las funciones prioridad
38  int setprio(int pid, int prio);
39  int getprio(int pid);
```

“proc.h”



```
587  int
588  setprio(int pid, int prio)
589  {
590    struct proc *p;
591    acquire(&ptable. lock);
592    p = buscar_proceso(pid);
593    if (p != NULL) {
594      p->priority = prio;
595    }
596    release(&ptable. lock);
597    return pid;
598  }
599
600  int
601  getprio(int pid)
602  {
603    struct proc *p;
604    int prio=-1;
605    acquire(&ptable. lock);
606    p = buscar_proceso(pid);
607    if (p != NULL) {
608      prio=p->priority;
609    }
610    release(&ptable. lock);
611    return prio;
612  }
```

“proc.c”

Para la implementación de int setprio() primero tenemos que saber que tenemos acceso exclusivo a la tabla de procesos una vez lo sabemos llamó a una función buscar_proceso() (que hice para reutilizar tanto en getprio() como en setprio()), nos sirve para buscar un determinado proceso, después de obtener el proceso que buscábamos le asignamos la prioridad que queremos y después liberamos la tabla de procesos devolviendo asi el pid del proceso modificado.

Para la implementación de int getprio() primero tenemos que saber que tenemos acceso exclusivo a la tabla de procesos una vez lo sabemos llamó a una función buscar_proceso(), buscando así el procesos específico que buscamos, seguidamente le asigno a mi variable pero la prioridad del proceso para así devolverla, pero antes habrá que desbloquear la tabla de procesos.

A continuación añado el código de la función buscar_proceso():

```
575  struct proc*
576  buscar_proceso(int pid)
577  {
578    struct proc *p;
579    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
580      if(p->pid == pid){
581        return p;
582      }
583    }
584    return NULL;
585  }
```

función buscar_proceso() en “proc.c”

Y por último ya falta implementar la funcionalidad de las llamadas al sistema, siendo más específicos sys_getprio() y sys_setprio(), se implementa en el fichero “sysproc.c”.

```
146  int
147  sys_getprio(void)
148  {
149    int pid;
150    if(argint(0,&pid)<0)
151      return -1;
152    return getprio(pid);
153  }
154
155  int
156  sys_setprio(void)
157  {
158    int pid;
159    int prio;
160    if(argint(0,&pid)<0 || argint(1,&prio)<0)
161      return -1;
162    return setprio(pid,prio);
163  }
```

“sysproc.c”

Estas llamadas al sistema implementan las funciones argint() para asegurarse que pueden obtener los valores de pid y prio desde el espacio de memoria del proceso para realizar la llamada al sistema. Y una vez obtiene estos valores termina realizando una llamada a la función setprio o getprio (dependiendo de la función que estemos viendo) implementadas ya anteriormente.