PRÁCTICA 3

Planificación de procesos

VERSIÓN BETA

Adaptación (P. P. López) para Windows de practica3.pdf (P. Carazo)

ÍNDICE

1 OBJETIVOS

Los objetivos de esta práctica son:

- Entender la política de planificación de procesos de MINIX
- Modificar la política de planificación de procesos de MINIX

2 ENUNCIADO

Lo que se pide en esta práctica es modificar la estrategia de planificación de procesos de MINIX 3 (transparencia 86 de procesos.ppt) atendiendo a los criterios siguientes:

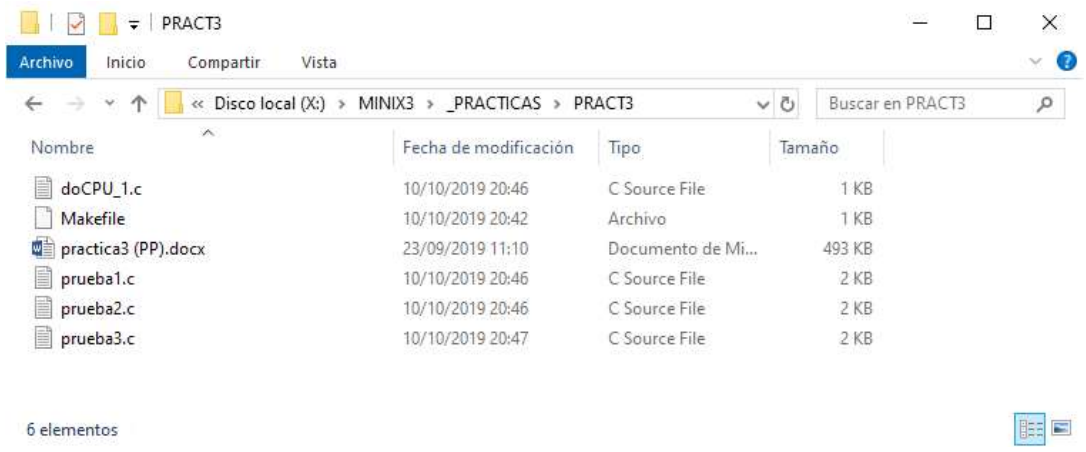
1. La política de planificación no debe modificar la prioridad de un proceso (sí podría modificarla algún comando externo al planificador)
2. En el momento de determinar qué proceso debe ejecutarse, se seleccionará uno de la lista más prioritaria no vacía (no cambia con respecto a MINIX 3)
3. Dentro de una lista, en lugar de aplicar turno circular, debe seleccionarse al proceso con mayor antigüedad en el sistema, considerando su edad como el tiempo transcurrido desde el momento de su creación por medio de la llamada al sistema **fork**.

Se pretende abordar la realización de la práctica en tres pasos: primero jugar un poco con la política de planificación actual sugiriendo hacer unos cambios y ver qué sucede; a continuación implementar ya una primera aproximación a la solución y acabar con una solución definitiva al problema planteado.

3 JUGANDO CON LA POLÍTICA DE PLANIFICACIÓN

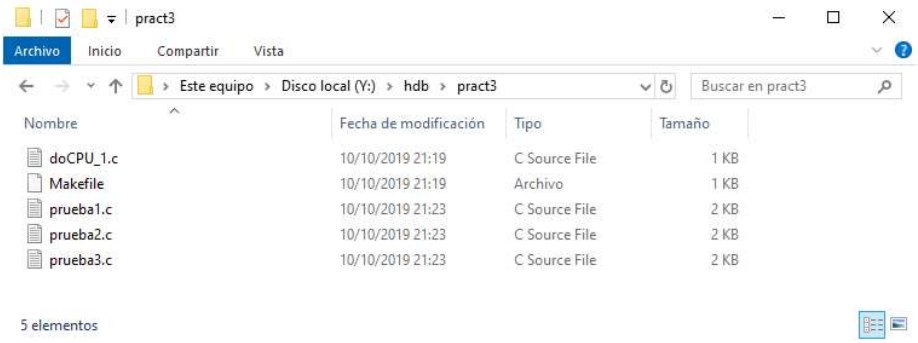
Se trata de realizar unos ejercicios básicos que nos permitan entender mejor algunos aspectos relacionados con la planificación de procesos en MINIX 3.

En esta práctica se utilizan tres programas de prueba: **prueba1.c**, **prueba2.c** y **prueba3.c**, disponibles en el directorio **X:\MINIX3_PRACTICAS\PRACT3** (descargar de Moodle).

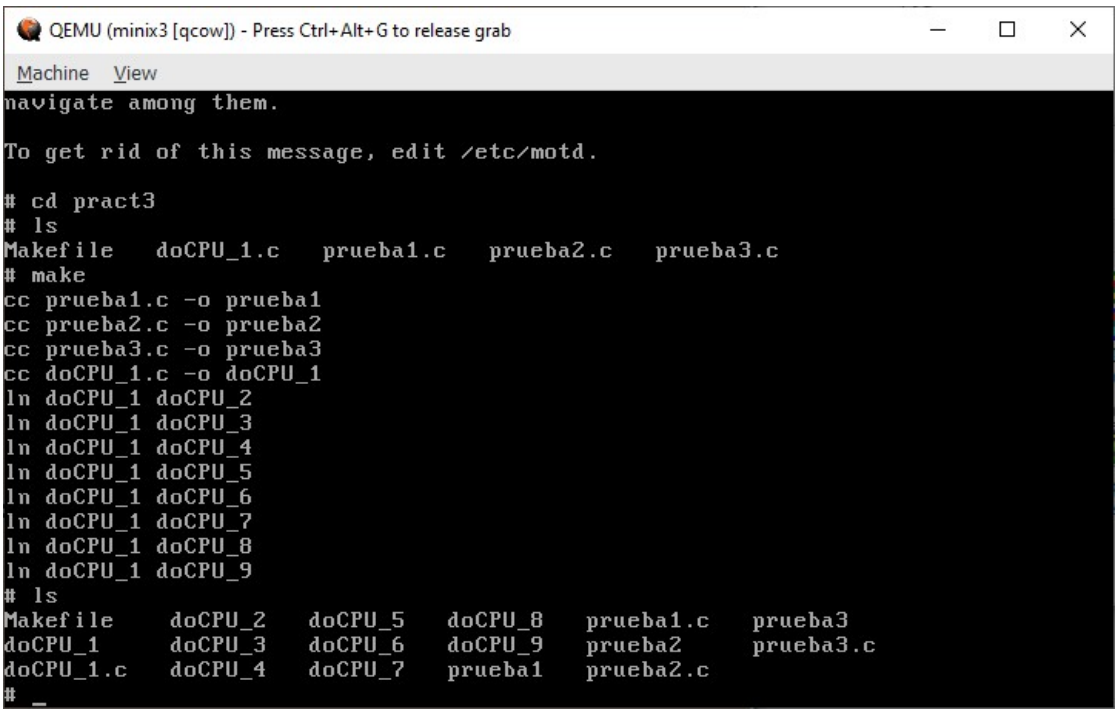


Para realizar esta parte de la práctica seguimos los siguientes pasos:

- Encender el equipo del laboratorio, entrar en Windows y preparar el software de prácticas **SO 2019**.
- Copiar a **Y:\hdb\pract3** los programas **prueba1.c**, **prueba2.c** y **prueba3.c**, junto con **doCPU_1.c** y **Makefile**, del directorio **X:\MINIX3_PRACTICAS\PRACT3**, para dejarlos accesibles a MINIX.



- Arrancar MINIX pinchando sobre el icono **X:\minix3.exe**.
- Entrar al sistema, crear una carpeta **/root/pract3** con el comando **mkdir /root/pract3**. Copiar en ella los tres ficheros [**mtools copy c0d1p0:/pract3/* /root/pract3**]. Comprobar que en **/root/pract3** están los 3 programas de prueba.



- Compilar todos los programas con el comando **make**.

Los tres programas **prueba1**, **prueba2** y **prueba3** son similares. Todos crean (mediante la llamada al sistema **fork**) procesos hijos que (mediante un **execl**) ejecutan el programa **doCPU_1**, posiblemente con otro nombre **doCPU_2**, **doCPU3**, … o **doCPU9**, quedando el proceso a la espera de la terminación de todos sus hijos mediante llamadas a **wait**.

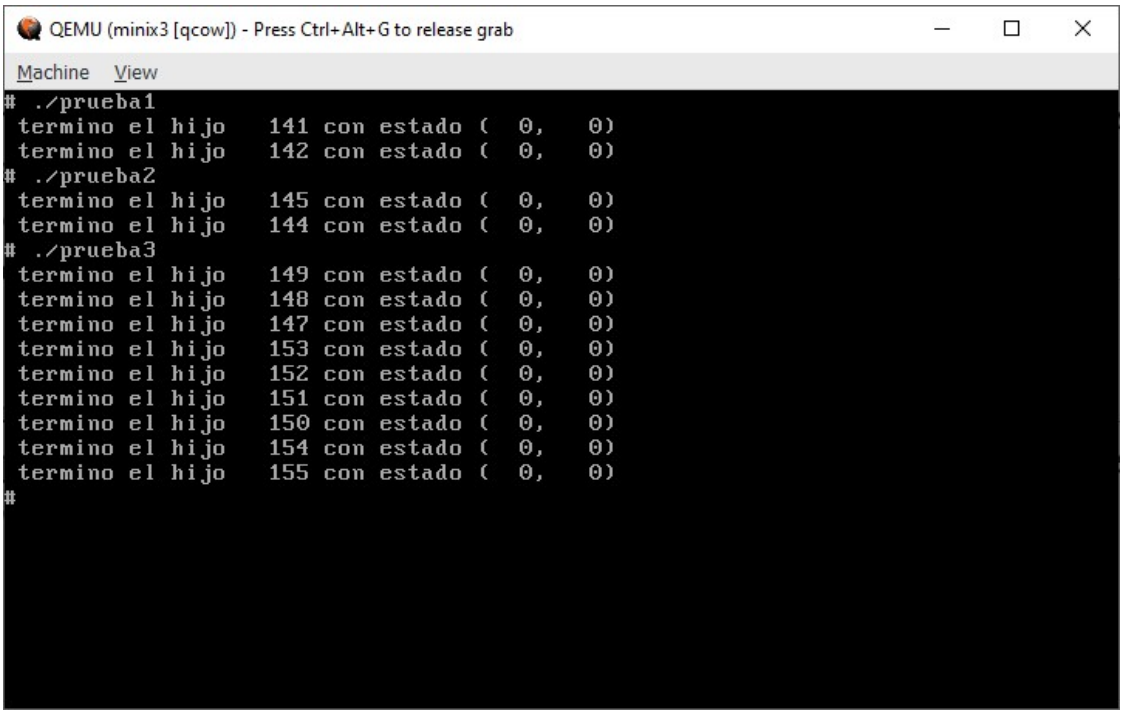
En el caso de **prueba1** y **prueba2** sólo se crean dos procesos hijos que ejecutan **doCPU_1** y **doCPU_2** respectivamente. La diferencia entre **prueba1** y **prueba2** es simplemente que en **prueba2** el segundo hijo espera un segundo gracias a la llamada **sleep(1)** antes de transformarse mediante un **execl** en **doCPU_1**.

Por su parte **prueba3** es similar a **prueba1**, salvo que se crean nueve procesos hijos en vez de dos, que son **doCPU_1**, **doCPU_2**, … , **doCPU_9**.

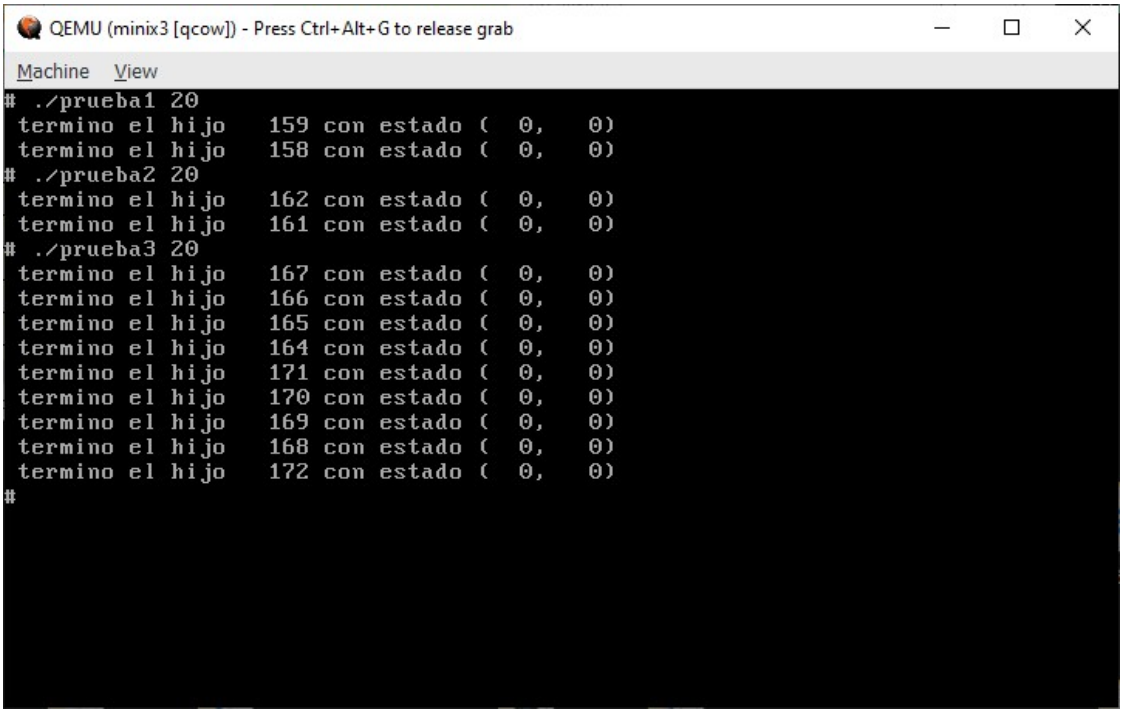
El programa **doCPU_1** que ejecutan todos los procesos hijos es simplemente un programa que hace un uso continuado de la CPU durante un cierto número de iteraciones. Por defecto el número de iteraciones es de 10, pero puede modificarse indicando el número de iteraciones que queremos como parámetro de **prueba1**, **prueba2** o **prueba3** (e incluso de **doCPU_1**). Este tipo de programas con largas ráfagas de CPU se denominan intensos en CPU (*CPU bound*).

De esta manera los programas **prueba1**, **prueba2** y **prueba3** nos van a permitir observar cómo el planificador va asignando rodajas de CPU a los procesos hijos que se ejecutan en el sistema.

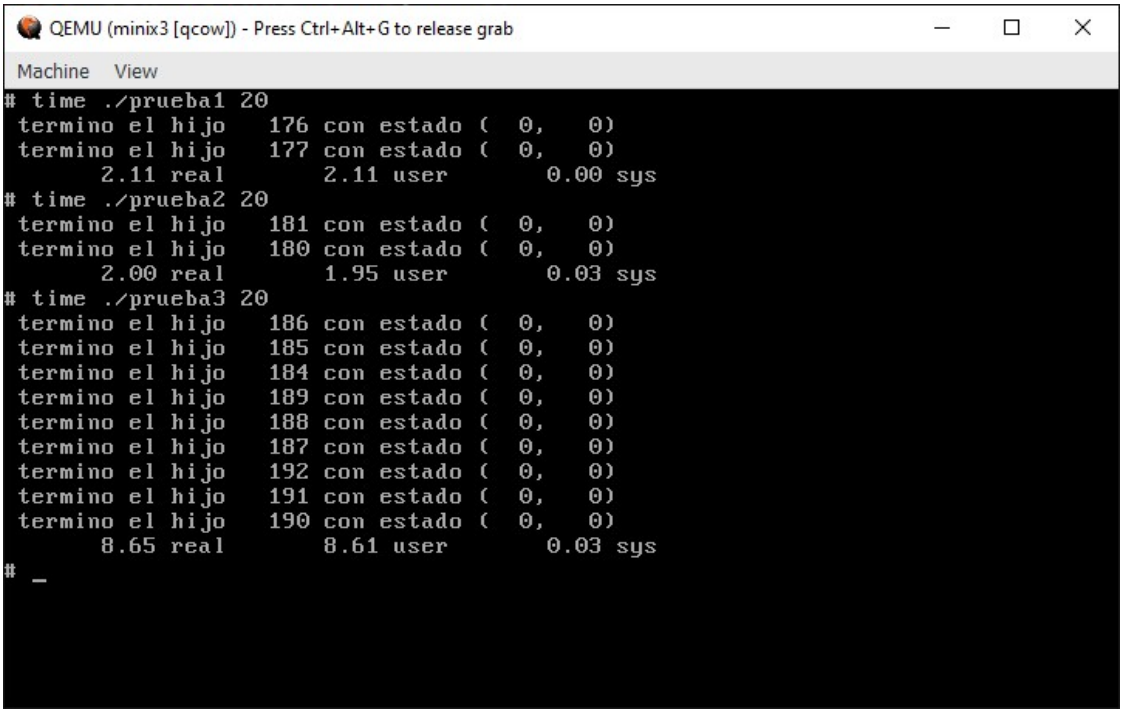
ejecución con el número de iteraciones por defecto (10)



ejecución con número de iteraciones igual a 20



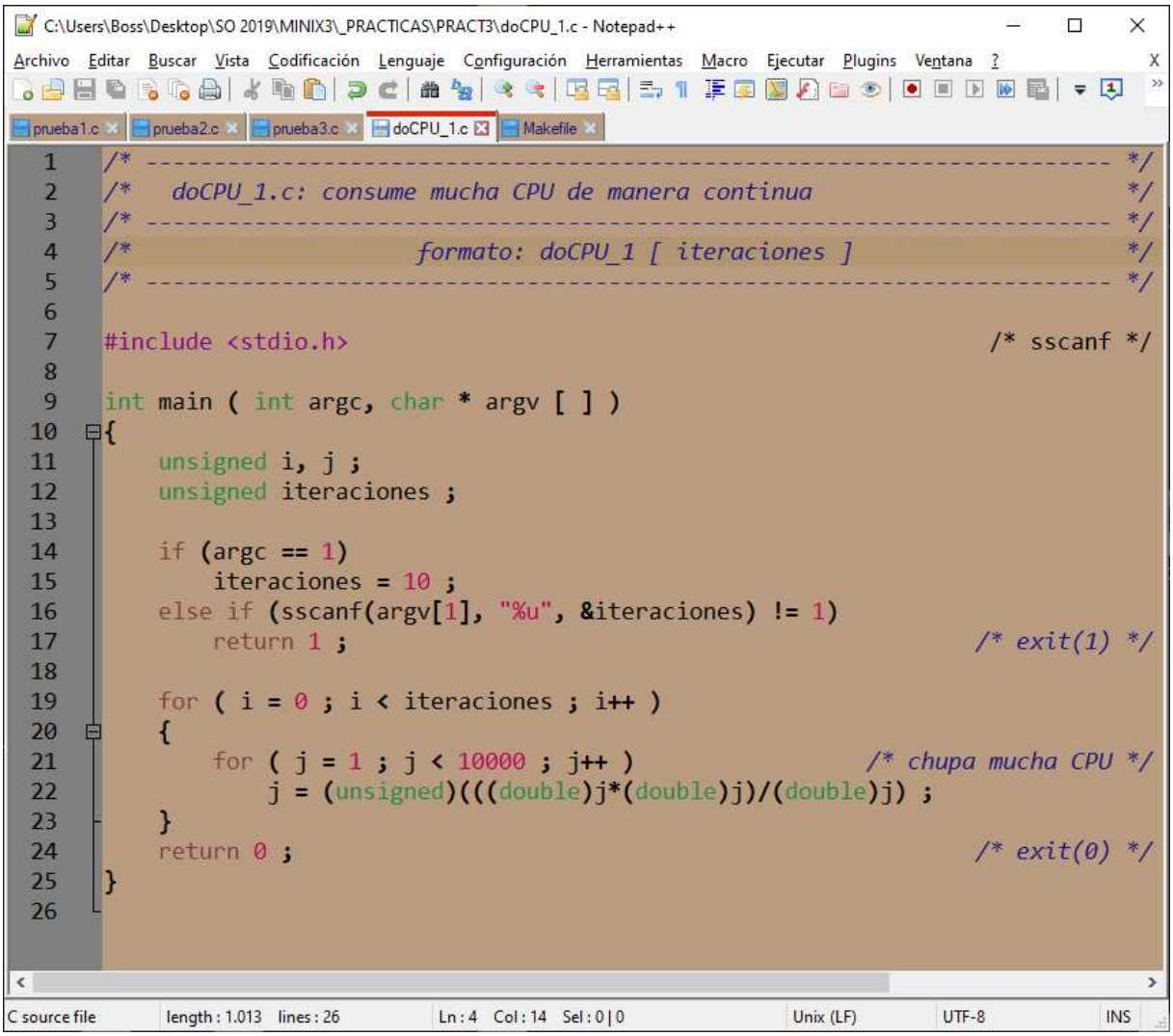
tiempos de ejecución con número de iteraciones igual a 20



Vemos que en nuestro caso con 20 iteraciones el programa tarda en terminar aproximadamente 1 segundo multiplicado por el número de procesos hijos creados. Es decir: 2 segundos para el caso de **prueba1** y **prueba2**, y 9 segundos para el caso de **prueba3**.

- En vuestro equipo/máquina virtual Minix ejecutar con **time** los programas ./**prueba1 20**, ./**prueba2 20** y ./**prueba3 20**. Indicar en la pantalla siguiente los tiempos que os salen (por ejemplo mostrando un pantallazo) y estimar cuánto tiempo tardan los programas en terminar, en función del número de procesos (a mí se salía 1 segundo por proceso hijo creado).

- Veamos los fuentes de todos los programas: **doCPU_1.c**, **prueba1.c**, **prueba2.c** y **prueba3.c**.



C:\Users\Boss\Desktop\SO 2019\MINIX3_PRACTICAS\PRACT3\prueba1.c - Notepad++

Archivo Editar Buscar Vista Codificación Lenguaje Configuración Herramientas Macro Ejecutar Plugins Ventana ?

prueba1.c | prueba2.c | prueba3.c | doCPU_1.c | Makefile

```c
/* ------------------------------------------------------------------------ */
/*   prueba1.c: Crea dos procesos hijos                                     */
/* ------------------------------------------------------------------------ */
/*                      formato: prueba1 [ iteraciones ]                    */
/* ------------------------------------------------------------------------ */

#include <unistd.h>                             /* fork, execl, _HIGH, _LOW */
#include <sys/wait.h>                                              /* wait */
#include <stdio.h>                                                /* printf */
#ifndef _HIGH
#define _HIGH(i) (int)(i >> 8)
#define _LOW(i) (int)(i & 0x000000FF)
#endif

char str10 [ 3 ] = "10" ;

int main ( int argc, char * argv [ ] )
{
    char * strIteraciones = (char * )&str10 ;                /* por defecto */
    int estado, pid ;

    if (argc > 1) strIteraciones = argv[1] ;

    if (fork() == 0)
        if (execl("doCPU_1", "doCPU_1", strIteraciones, NULL)) return 13 ;

    if (fork() == 0)
        if (execl("doCPU_2", "doCPU_2", strIteraciones, NULL)) return 13 ;

    pid = wait(&estado) ;
    printf(" termino el hijo %5i con estado (%3i, %3i)\n",
           pid, _HIGH(estado), _LOW(estado)) ;

    pid = wait(&estado) ;
    printf(" termino el hijo %5i con estado (%3i, %3i)\n",
           pid, _HIGH(estado), _LOW(estado)) ;

    return(0) ;
}
```

C source file | length : 1.409 lines : 40 | Ln : 17 Col : 39 Sel : 0 | 0 | Unix (LF) | UTF-8 | INS

C:\Users\Boss\Desktop\SO 2019\MINIX3_PRACTICAS\PRACT3\prueba2.c - Notepad++

Archivo Editar Buscar Vista Codificación Lenguaje Configuración Herramientas Macro Ejecutar Plugins Ventana ?

prueba1.c prueba2.c prueba3.c doCPU_1.c Makefile

```c
/* ------------------------------------------------------------------------ */
/*   prueba2.c: Crea dos procesos hijos (el segundo proceso con sleep(1))   */
/* ------------------------------------------------------------------------ */
/*                     formato: prueba2 [ iteraciones ]                     */
/* ------------------------------------------------------------------------ */

#include <unistd.h>                             /* fork, execl, _HIGH, _LOW */
#include <sys/wait.h>                                              /* wait */
#include <stdio.h>                                                /* printf */
#ifndef _HIGH
#define _HIGH(i) (int)(i >> 8)
#define _LOW(i) (int)(i & 0x000000FF)
#endif

char str10 [ 3 ] = "10" ;

int main ( int argc, char * argv [ ] )
{
    char * strIteraciones = (char * )&str10 ;                /* por defecto */
    int estado, pid ;

    if (argc > 1) strIteraciones = argv[1] ;

    if (fork() == 0)
    {
        sleep(1) ; /* zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz */
        if (execl("doCPU_1", "doCPU_1", strIteraciones, NULL)) return 13 ;
    }

    if (fork() == 0)
        if (execl("doCPU_2", "doCPU_2", strIteraciones, NULL)) return 13 ;

    pid = wait(&estado) ;
    printf(" termino el hijo %5i con estado (%3i, %3i)\n",
           pid, _HIGH(estado), _LOW(estado)) ;

    pid = wait(&estado) ;
    printf(" termino el hijo %5i con estado (%3i, %3i)\n",
           pid, _HIGH(estado), _LOW(estado)) ;

    return(0) ;
}
```

C source file | length : 1.499 lines : 43 | Ln : 26 Col : 14 Sel : 0 | 0 | Unix (LF) | UTF-8 | INS

C:\Users\Boss\Desktop\SO 2019\MINIX3_PRACTICAS\PRACT3\prueba3.c - Notepad++

Archivo Editar Buscar Vista Codificación Lenguaje Configuración Herramientas Macro Ejecutar Plugins Ventana ?

prueba1.c prueba2.c prueba3.c doCPU_1.c Makefile

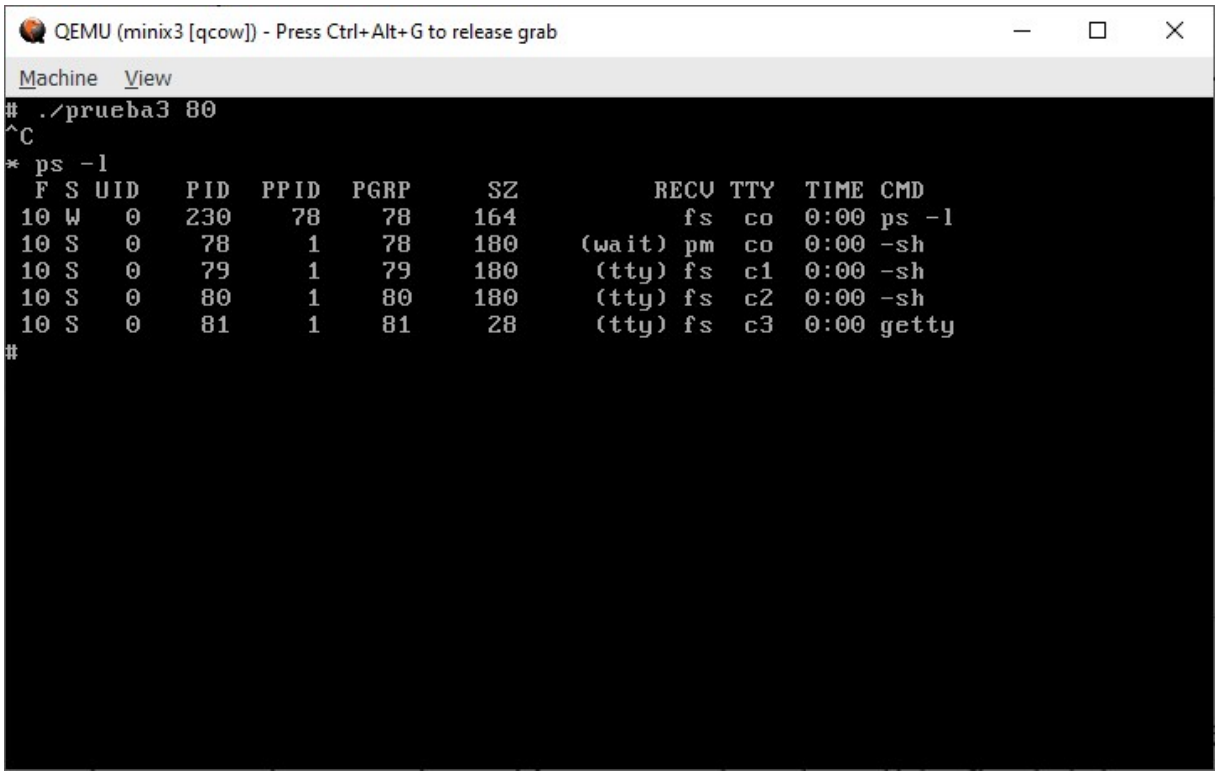
```c
/* ------------------------------------------------------------------- */
/*   prueba1.c: Crea 9 procesos hijos                                  */
/* ------------------------------------------------------------------- */
/*                      formato: prueba3 [ iteraciones ]               */
/* ------------------------------------------------------------------- */

#include <unistd.h>                              /* fork, execl, _HIGH, _LOW */
#include <sys/wait.h>                                          /* wait */
#include <stdio.h>                                            /* printf */
#ifndef _HIGH
#define _HIGH(i) (int)(i >> 8)
#define _LOW(i) (int)(i & 0x000000FF)
#endif

#define CREAR_PROCESO(fichero)                                          \
    if (fork() == 0)                                                    \
        if (execl(fichero, fichero, strIteraciones, NULL)) return 13 ;  \

#define ESPERAR_PROCESO()                                               \
    pid = wait(&estado) ;                                               \
    printf(" termino el hijo %5i con estado (%3i, %3i)\n",              \
           pid, _HIGH(estado), _LOW(estado)) ;                          \

char str10 [ 3 ] = "10" ;

int main ( int argc, char * argv [ ] )
{
    char * strIteraciones = (char * )&str10 ;              /* por defecto */
    int estado, pid, i ;

    if (argc > 1) strIteraciones = argv[1] ;

    CREAR_PROCESO("doCPU_1") ;
    CREAR_PROCESO("doCPU_2") ;
    CREAR_PROCESO("doCPU_3") ;
    CREAR_PROCESO("doCPU_4") ;
    CREAR_PROCESO("doCPU_5") ;
    CREAR_PROCESO("doCPU_6") ;
    CREAR_PROCESO("doCPU_7") ;
    CREAR_PROCESO("doCPU_8") ;
    CREAR_PROCESO("doCPU_9") ;

    ESPERAR_PROCESO() ;
    ESPERAR_PROCESO() ;
    ESPERAR_PROCESO() ;
    ESPERAR_PROCESO() ;
    ESPERAR_PROCESO() ;
    ESPERAR_PROCESO() ;
    ESPERAR_PROCESO() ;
    ESPERAR_PROCESO() ;
    ESPERAR_PROCESO() ;

    return(0) ;
}
```

C source file | length : 2.004 lines : 55 | Ln : 26 Col : 30 Sel : 0 | 0 | Unix (LF) | UTF-8 | INS

- Entrar en Minix también desde la consola 1 (Alt + F2) y la consola 2 (Alt + F3).

- Desde la consola 0 (Alt + F1) ejecutar el programa **./prueba3 80** (en mi caso tardará unos 9*(80/20) = 36 segundos). Mientras se ejecuta, introducir en la consola 1 el comando **ps -l** y en la consola 2 el comando **top**. Explicar en el siguiente recuadro qué es lo que está sucediendo (además de acompañar los pantallazos del ps –l y del top).

Si en una posterior ejecución no se quiere esperar a que termine el comando **./prueba3 80** puede interrumpirse su ejecución con Ctrl-C, la salida final del programa sería:

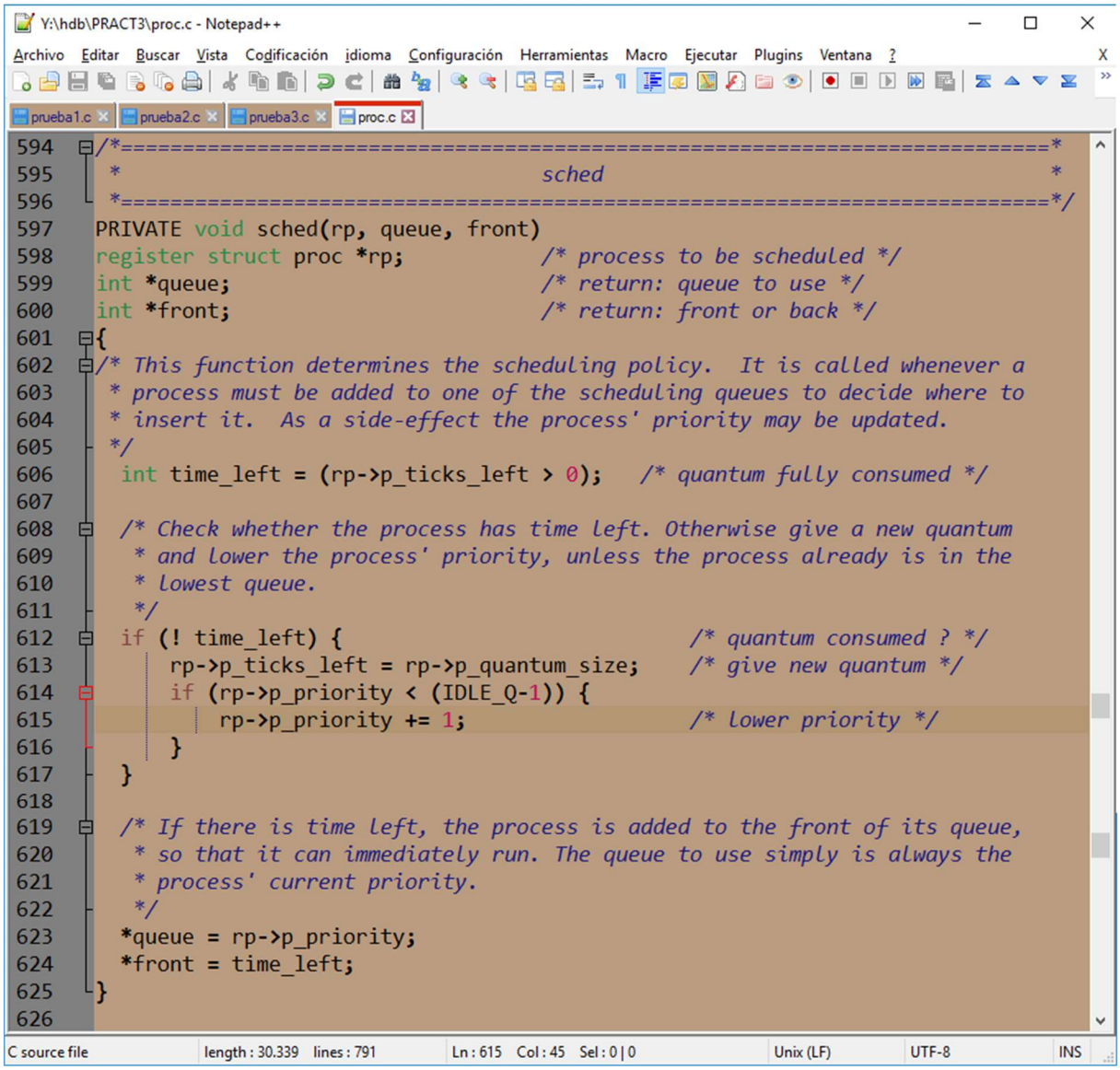


El prompt * indica que ha sucedido algo extraño en la ejecución del comando anterior, como así ha sido. Para recuperar el prompt normal basta con ejecutar sin problemas cualquier comando. Un comando que podemos utilizar en estos casos y que nunca da problemas es el comando **true**.

- Ahora vamos a realizar en el fichero **/usr/src/kernel/proc.c** algunos cambios que tienen que ver con modificaciones en la política de planificación, con el fin de observar el efecto que producen. Concretamente lo que haremos será incrementar la prioridad lógica de los procesos, lo que significa en Minix reducir el valor numérico de la prioridad. Comenzamos situándonos en el directorio **/usr/src/kernel** y hacemos una copia del fichero **proc.c** con el nombre **proc.c.org** con el fin de no perder la versión original ya que vamos a hacer modificaciones.

- Editar **proc.c** y localizar la función **sched** que tiene como encabezamiento

PRIVATE void **sched** (**rp**, **queue**, **front**)




```
/*===========================================================================*
 *                                sched                                      *
 *===========================================================================*/
PRIVATE void sched(rp, queue, front)
register struct proc *rp;           /* process to be scheduled */
int *queue;                         /* return: queue to use */
int *front;                         /* return: front or back */
{
/* This function determines the scheduling policy.  It is called whenever a
 * process must be added to one of the scheduling queues to decide where to
 * insert it.  As a side-effect the process' priority may be updated.
 */
  int time_left = (rp->p_ticks_left > 0);   /* quantum fully consumed */

  /* Check whether the process has time left. Otherwise give a new quantum
   * and lower the process' priority, unless the process already is in the
   * lowest queue.
   */
  if (! time_left) {                                /* quantum consumed ? */
      rp->p_ticks_left = rp->p_quantum_size;        /* give new quantum */
      if (rp->p_priority < (IDLE_Q-1)) {
          rp->p_priority += 1;                      /* lower priority */
      }
  }

  /* If there is time left, the process is added to the front of its queue,
   * so that it can immediately run. The queue to use simply is always the
   * process' current priority.
   */
  *queue = rp->p_priority;
  *front = time_left;
}
```

- Sustituir la instrucción:

```
if (rp->p_priority < (IDLE_Q-1)) {
    rp->p_priority += 1;
}
```

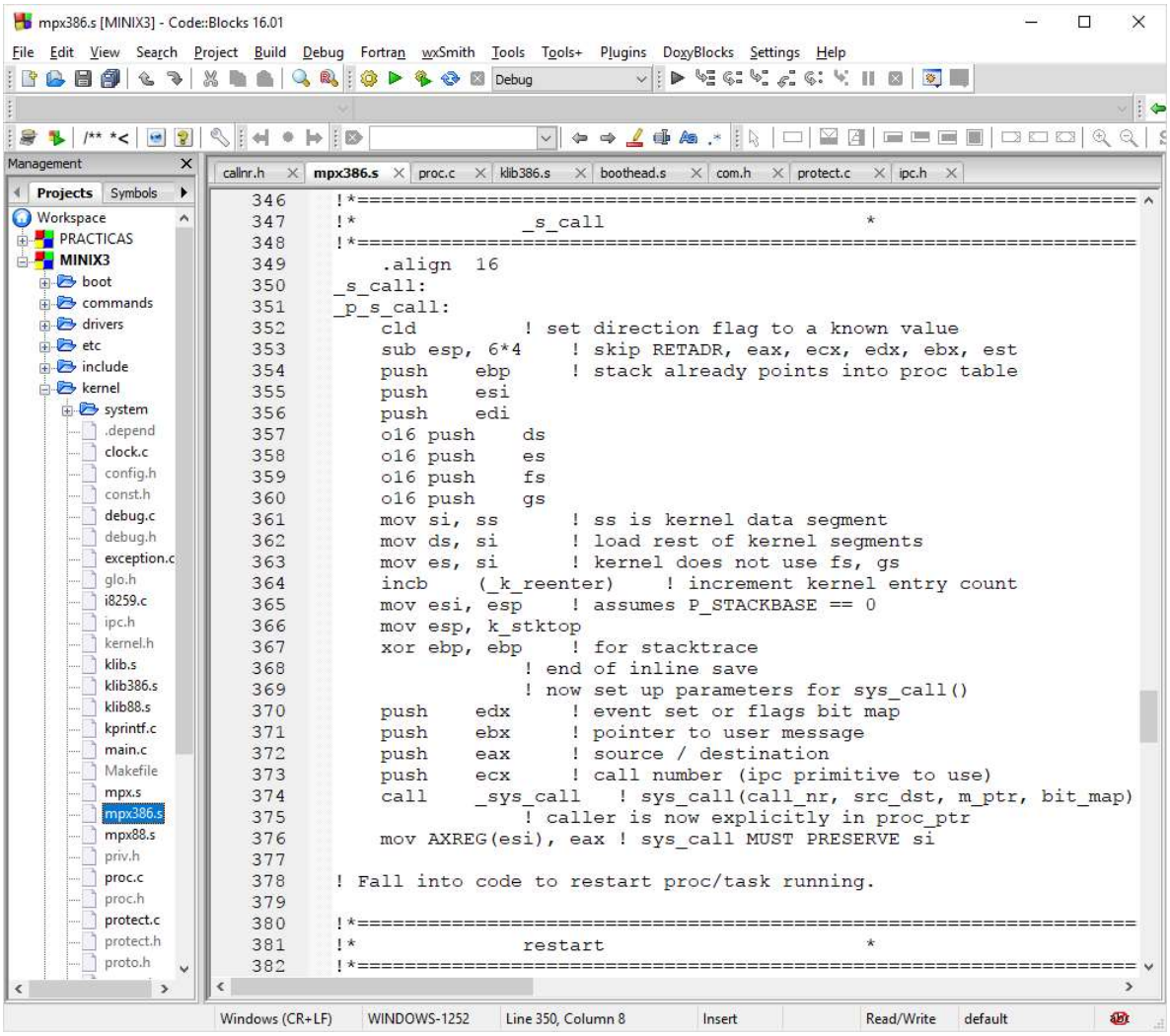
por:

```
switch (plot_sched) {
case 0:
    if (rp->p_priority < (IDLE_Q-1)) {
        rp->p_priority += 1;
    }
    break ;
case 1:
    if (rp->p_priority > 0)
        rp->p_priority -= 1;
    break ;
default: ;
}
```

Para que la variable global **plot_sched** esté definida en el núcleo de Minix es necesario copiar en /usr/src/kernel el fichero de cabeceras **plotear.h**. Dicho fichero debe incluirse al principio de **proc.c** con la directiva del preprocesador **#include "plotear.h"**.

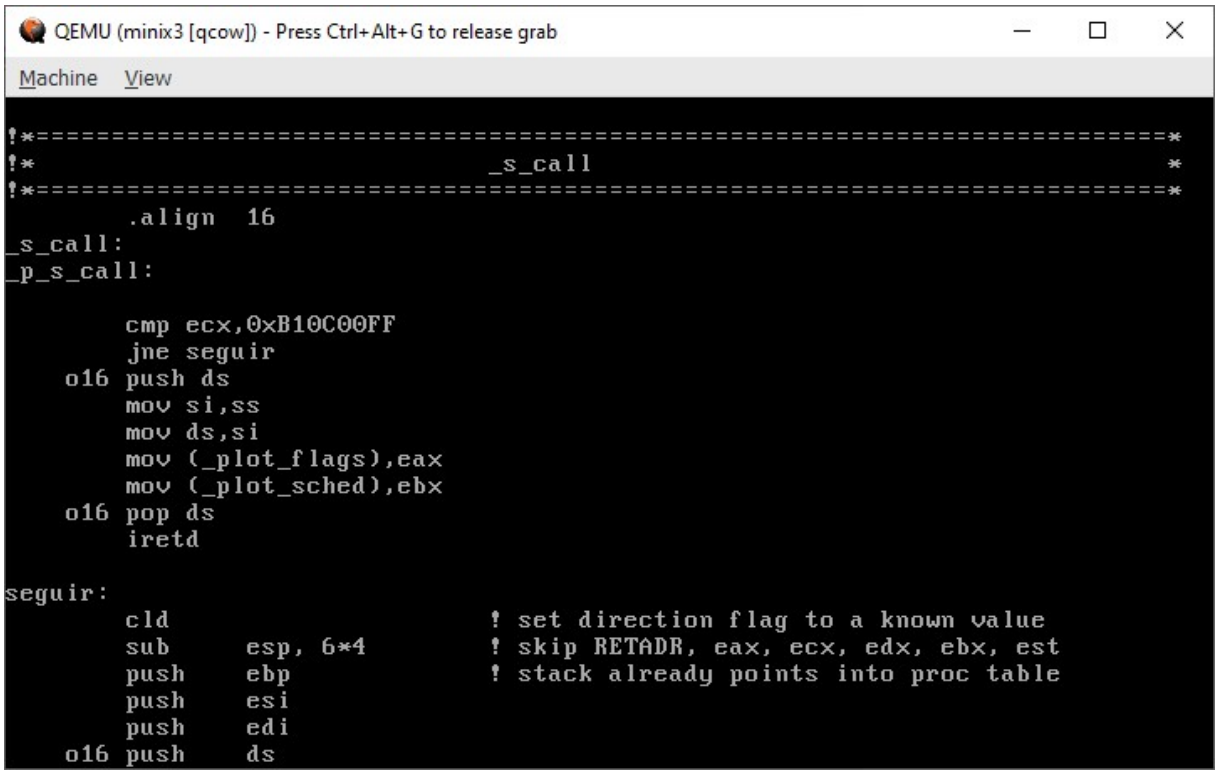
- Salvar el fichero **proc.c** una vez modificado.

Para permitir que los programas de usuario modifiquen la variable **plot_sched** es necesario modificar **mpx386.s** a partir de su línea 352 (punto de entrada de todas las llamadas al sistema):



- Incluir las siguientes líneas en **mpx386.s** línea 352 (comando **Ctrl +] 352** de mined):

```
_s_call:
_p_s_call:
    cmp ecx,0xB10C00FF          ! ¿ plot ?
    jne seguir
o16 push ds
    mov si,ss
    mov ds,si
    mov (_plot_flags),eax       ! plot_flags = eax ;
    mov (_plot_sched),ebx       ! plot_sched = ebx ;
o16 pop ds
    iretd                       ! retorno de interrupción/excepcion
seguir:
    ...
```

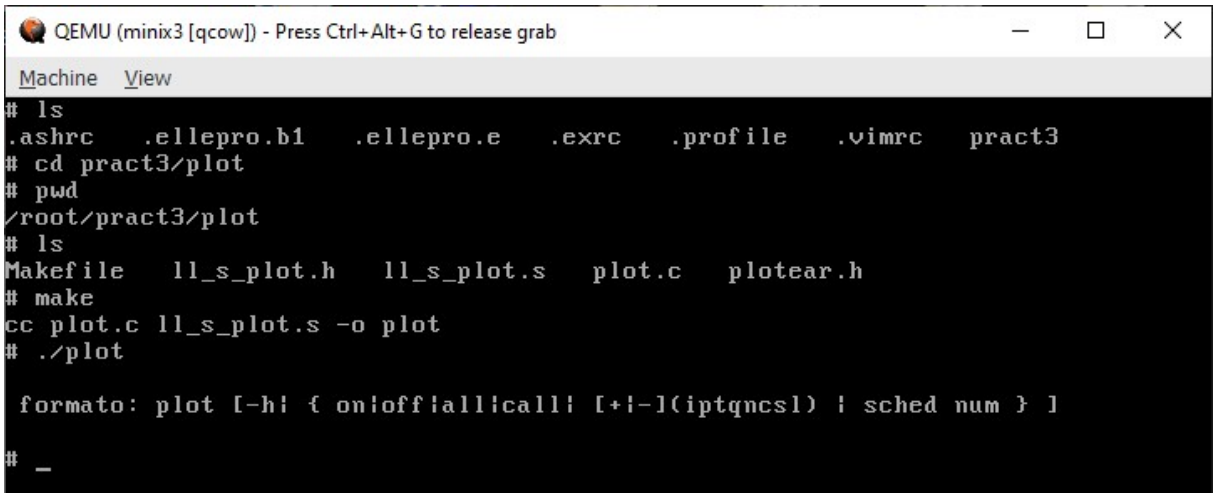


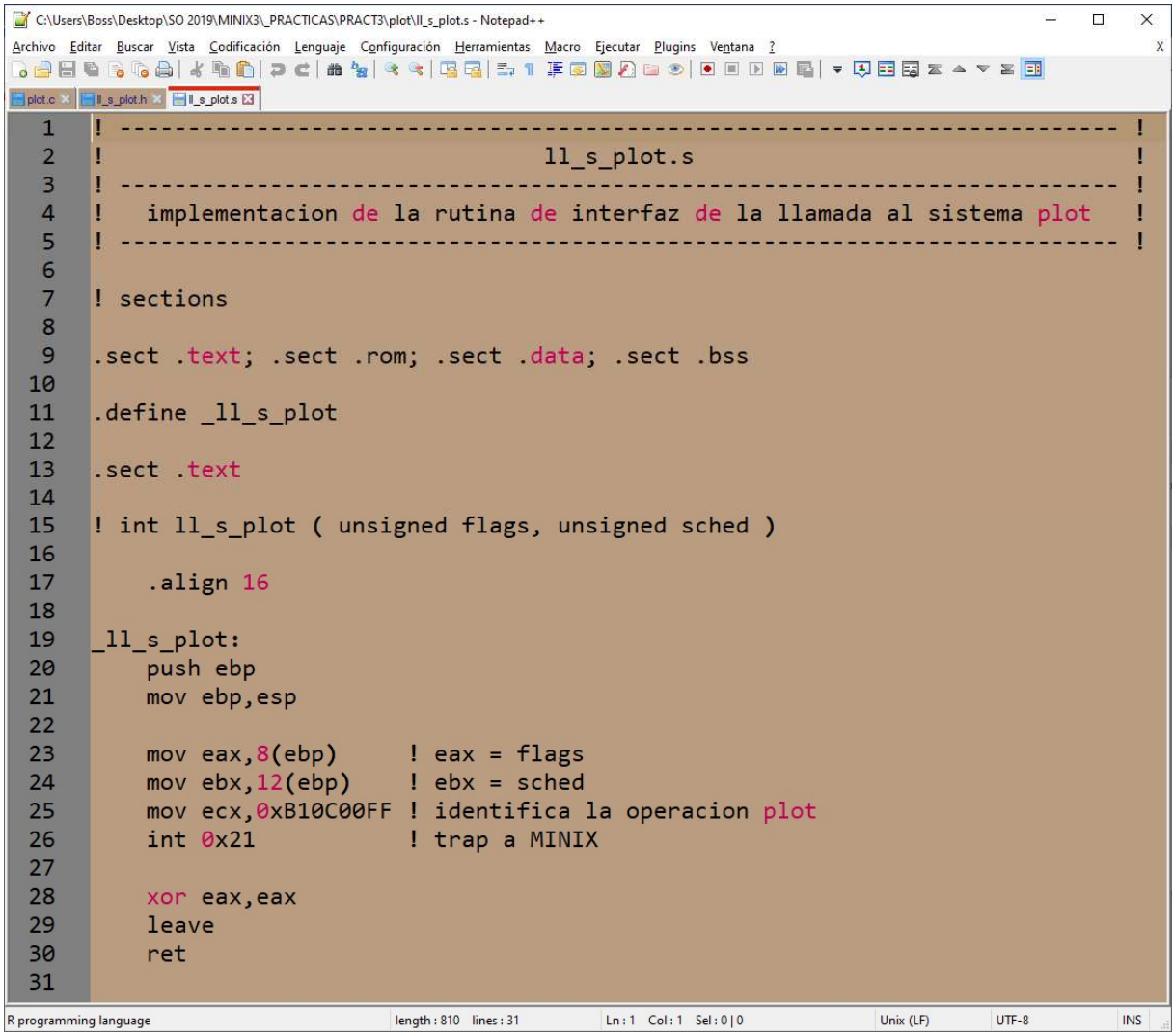
- Salvar el fichero mpx386.s una vez modificado y cambiar el directorio de trabajo a **/usr/src/tools.**

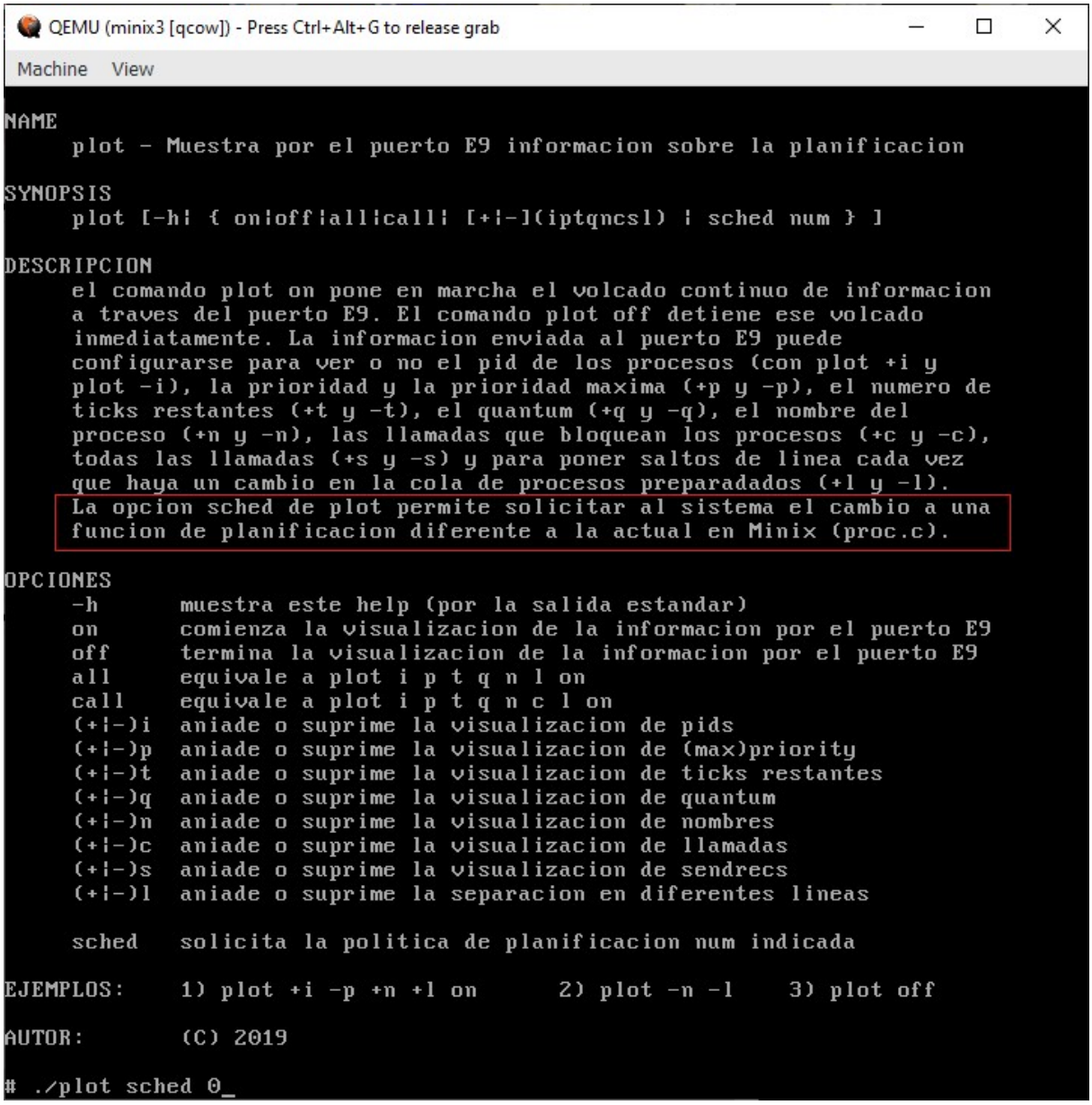
- Recompilar el sistema tecleando: **make install**

- Reiniciar MINIX (**halt** + exit) con la nueva imagen. ¿Qué ocurre?

- Entrar en Minix como root, ir al directorio /root/pract3/plot y compilar el comando plot.c con **make** o con **cc plot.c ll_s_plot.s –o plot**, donde **ll_s_plot.s** es la implementación de la rutina de interfaz de la llamada al sistema plot.







- Ejecutar el comando: **./plot sched 0** ¿Qué ocurre?

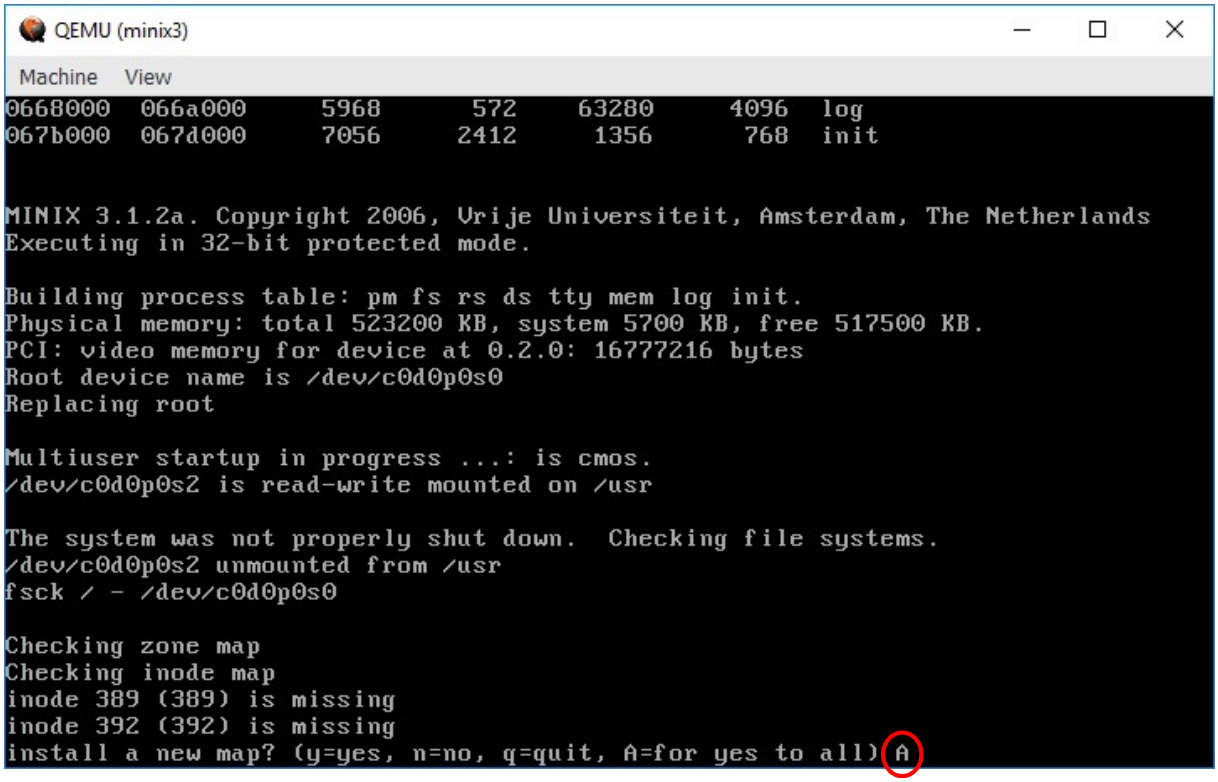
- Ejecutar el comando: **./plot sched 1** ¿Qué ocurre?

- ¿Cuál puede ser la explicación de este comportamiento? Consultar con el profesor.

- Para constatar un poco lo que pasa y, de paso, comprobar lo dificultoso que puede ser hacer un seguimiento de lo que está pasando, vamos a intentar informar con cierta perioricidad de lo que está haciendo la función "sched". La idea es utilizar un contador declarado como "**PUBLIC int totVeces = 0 ;**" justo antes de la declaración del cuerpo de la función y, cada vez que se eejcute incrementar dicha variable e imprimir cada 10000 veces el valor de "totVeces" así como el p_nr del proceso que entra en "sched" y su prioridad.

- Compilar y arrancar con las nuevas modificaciones. Escribir lo relevante de lo que aparece en pantalla e intentar explicar qué sucede.

- Ahora vamos a eliminar el cambio anterior que alteraba la prioridad de los procesos y en su lugar vamos a elegir el siguiente proceso a ejecutar sin tener en cuenta si terminó o no su rodaja de tiempo de CPU. Para ello, arrancar MINIX con la imagen original. Si el sistema de ficheros da error al arrancar, seleccionar en el menú del comando **fsck** la reparación automática (**A**) para todos los errores que nos aparezcan.



- Editar de nuevo el fichero **proc.c** y borrar la instrucción que hemos tocado en las pruebas anteriores:

```
if (rp->p_priority > 0)
    rp->p_priority -= 1;
```

- Cambiar la línea "*** front = time_left ;**" por la sentencia "*** front = TRUE ;** ".
- Grabar la modificación y cambiar el directorio de trabajo al directorio **/usr/src/tools.**
- Recompilar el sistema tecleando: **make install**
- Reiniciar MINIX con la nueva imagen.

- Situarse en el directorio **/root/pract3d** y volver a ejecutar el programa de prueba **prueba1.c** (**a.out**). ¿Qué diferencia se detecta respecto de la ejecución anterior de este mismo programa?

- ¿Cuál puede ser la explicación de este comportamiento?

- Ahora echar un vistazo al programa **prueba2.c** para entender su comportamiento: básicamente el primer proceso hijo se duerme durante un segundo antes de pasar a hacer un uso intesivo de la CPU.

- ¿Se consigue cumplir con el objetivo del enunciado de esta práctica?

- Al salir del sistema con **halt** es posible que se quede colgado y no aparezca el monitor del bootloader.

4 PRIMERA APROXIMÁCIÓN

En MINIX, cuando se crea un proceso, se busca un hueco libre en la tabla de descriptores de proceso para gestionar el proceso recién creado. Esta búsqueda de un descriptor libre se hace en orden estrictamente creciente. En esta primera aproximación se propone utilizar el índice que ocupa el proceso en dicha tabla ya que parece correlacionarse con el orden de creación de los procesos, en definitiva, con la antigüedad de los mismos.

El índice que ocupa un proceso en la tabla de descriptores puede ***derivarse*** del campo **p_nr** (process number) del descriptor de cada proceso, cuya definición puede consultarse en **/usr/src/kernel/proc.h**. Hay que tener cuidado, especialmente en el siguiente apartado, ya que este campo toma valores negativos para las tareas y de cero en adelante para los procesos.

Si consultamos con más detalle el fichero **proc.h**, veremos que la tabla de descriptores de proceso está declarada como:

```
EXTERN struct proc proc [NR_TASKS + NR_PROCS] ; /*process table*/
```

En la versión de MINIX que utilizamos, NR_TASKS es igual a 4 y NR_PROCS es igual a 100. En definitiva, mientras que **p_nr** variará de [-4 a 99], los índices para acceder adecuadamente a la tabla de procesos deben variar de [0 a 103] tal y como se refleja en la figura siguiente:

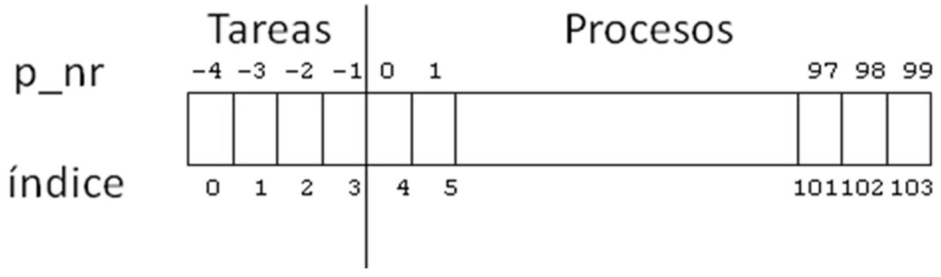


Figura 1: Relación entre el campo **p_nr** y el índice en la tabla de procesos

En definitiva, para tener realmente el índice de un proceso en la tabla de procesos a partir del campo **p_nr**, tendría que escribirse: **p_nr + NR_TASKS**. En cualquier caso, en esta primera

parte nos basta con utilizar el campo **p_nr** tal cual ya que refleja, de forma aproximada la idea de antigüedad.

Seleccionar el proceso más antiguo a través del campo **p_nr** tiene una pega. Cuando el número de procesos que se han ido creando (y posiblemente eliminando) hace que hayamos consumido todas las entradas de la tabla de procesos, se empieza a buscar de nuevo un descriptor libre desde el principio de la tabla, por lo que el procedimiento planteado funciona correctamente mientras no se creen en la instalación un número de procesos mayor que el número de entradas de la tabla de descriptores de procesos.

La ventaja de esta aproximación es que nos basta con modificar sólo el fichero **proc.c** (**partir del que tenemos modificado** para garantizar que **sched** no cambia la prioridad de un proceso). A la hora de hacer efectiva la planificación, bastará con modificar en **proc.c** la función **enqueue**. Si echamos un vistazo a esta función, veremos que desde ella se llama a **sched**, la cual nos devuelve la cola **q** donde insertar el proceso. Nuestra modificación debe hacer esta inserción en base al campo "**p_nr**" para tener en cuenta la antigüedad del proceso.

- Modificar el fichero **proc.c** siguiendo las ideas sugeridas.

- Tras conseguir compilar el sistema correctamente, arrancar con la nueva imagen y probar su funcionamiento ejecutando la **prueba2**. El resultado debería ser el esperado según lo marcado por el enunciado. ¿Es así?

- Si ejecutamos una prueba en la que se cree un número más elevado de procesos, es muy probable que encontremos algún comportamiento discrepante con el comportamiento esperado. Para ello, puede utilizarse en repetidas ocasiones la **prueba3**.

- Indicar cuál es el comportamiento extraño que puede darse al ejecutar **prueba3** y en qué momento se produce:

- Comprobar la corrección de esta parte con el profesor.

5 UNA SOLUCIÓN MÁS ADECUADA

Una alternativa algo mejor a la anteriormente expuesta (aunque con algún defecto menor) puede ser registrar explícitamente la antigüedad de cada proceso en el momento de ser creado. Para ello debe crearse un campo, para cada posible proceso, en el cual registrar el número de orden con el que fue creado. La función "**enqueue**" debe entonces ordenar los procesos, dentro de cada cola de prioridad, atendiendo a dicho orden de creación. Esta solución requiere modificar, además de **proc.c**, otros tres ficheros:

- **proc.h** para hacer la declaración compartida del campo de antigüedad de cada proceso creado. La modificación natural sería añadir este campo en la estructura "**proc**" que contiene todos los campos de un descriptor de proceso. Sin embargo, tal y como puede

consultarse en el propio código, el acceso a los campos de un descriptor se hace con desplazamientos y, si cambiamos esta estructura, hay que tocar, posiblemente, dichos desplazamientos. Una forma de evitar este problema es declararse un array del mismo tamaño que la tabla de procesos y guardar ahí el campo de antigüedad.

- **system/do_fork.c** para que cuando se cree un proceso se registre correctamente su antigüedad.

- **main.c** para inicializar correctamente los campos de antigüedad de los procesos cargados en la imagen de arranque del sistema así como para anular la antigüedad de los procesos todavía no creados.

En esta parte, cuando realicemos búsquedas en el array auxiliar donde guardamos la antigüedad de cada proceso, debemos tener cuidado de acceder correctamente a partir del campo **p_nr** de cada proceso tal y como quedó reflejado en la **Figura 1**.

- Entrar al sistema y, antes de modificar los ficheros **proc.h**, **system/do_fork.c**, y **main.c**, copiarlos como **procOrg.h**, **system/do_forkOrg.c** y **mainOrg.c**.

- Hacer las modificaciones oportunas en **proc.h**, **proc.c**, **system/do_fork.c** y **main.c** para la realización de este apartado.

- Con esta alternativa deben seguir funcionando correctamente las pruebas 1 y 2. El programa prueba3.c debe funcionar siempre, incluso tras un número muy elevado de intentos.

- Comprobar la corrección de esta parte con el profesor.

6 PUNTUACIÓN

El apartado 3 “Jugando con la política de planificación” supone **0,4** puntos, el apartado 4 “Primera aproximación” aporta otros **0,6** puntos y el último apartado “Una solución más adecuada” aporta los **0,5** puntos restantes. Los apartados deben irse completando de forma secuencial.

La puntuación total de esta práctica, por lo tanto, es de **1,5** puntos.