1 Breve descripción.

Init es el padre de todos los procesos de usuario: si representásemos a todos estos en un árbol, entonces Init quedaría como el nodo raíz.

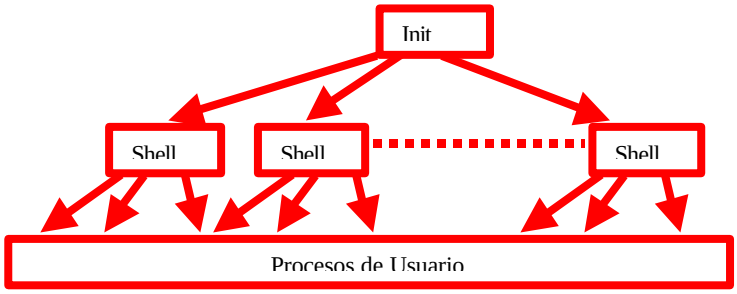
Básicamente, Init proporciona los servicios necesarios para que se conecten todas las terminales. De una forma muy general, los pasos que sigue son:

- *Configuración de la línea:* Primero, con el fin de hallar los parámetros físicos, se establece un diálogo con la terminal. Entre otros, se configura la paridad y la velocidad de transmisión de la línea.

- *Identificación del usuario:* Una vez concretados los parámetros anteriores, Init solicita la identificación del usuario.

- *Realizar el 'login':* Si la identificación es correcta, se pide la clave ('*password*') del usuario en cuestión.

- *Conexión de la terminal:* En el caso de que los datos del paso anterior sean correctos, se ejecuta el shell que el usuario haya configurado y se considera, a partir de aquí, que la terminal está conectada.



2 Antes que Init...

Tal y como se explica en la bibliografía de la asignatura, Minix se estructura en cuatro estratos: manejos de procesos, tareas de entrada/salida, procesos del servidor y procesos de usuario. Aunque todas estas capas se describen ulteriormente, para hacernos con una idea global, diremos que básicamente lo que sucede antes que Init son la carga del Kernel y la carga de los servicios útiles a los procesos del usuario, tales como el manejador de memoria y el sistema de archivos (servidores). El manejo de los procesos se hace principalmente en el Kernel, mientras que las interpretaciones de las llamadas al sistema toman lugar en los dos servidores mencionados. Además, debemos destacar que, una vez se hayan iniciado el manejador de memoria y el sistema de archivos, estos se bloquean así mismos y esperan a la ejecución del Init, es decir, esperan a que comiencen a ejecutarse los procesos de usuario y, por tanto, comiencen a recibir peticiones.

A continuación se describen brevemente los estratos en los que se compone Minix:

- *Manejo de procesos:* En esta capa, además de capturarse todas las interrupciones y trampas, se proporciona a los estratos superiores un modelo de procesos independientes que se comunican mediante el uso de mensajes.

- *Tareas de entrada/salida:* Este estrato contiene todos los manejadores de dispositivos (tareas). Debemos apuntar que a cada dispositivo sólo se le asigna, o asocia, una tarea.

- *Procesos del servidor:* En esta capa se efectúan todas las llamadas al sistema relacionadas con el manejo de la memoria y el sistema de archivos. Además, también podemos encontrar otros servidores, como el de red.

- *Procesos de usuario:* Init y todos los procesos de usuario.

3 Tareas que realiza Init.

En muchas de las tareas, referentes a las terminales, que realiza Init, se maneja el fichero *'/etc/ttytab'*. Éste, básicamente, contiene la información necesaria de todas las líneas (terminales) que pueden conectarse al sistema.

- *Inicializa y enlaza cada una de las terminales:* Para cada una de las terminales definidas en el fichero *'/etc/ttytab'* se intenta realizar su iniciación y, luego, si todo ha ido bien, se procede a su conexión y a la posterior autenticación del usuario. Todas estas acciones, en el código, quedan contempladas dentro de la función *'startup'*.

- *Mantiene un historial de las conexiones:* En los ficheros *'/etc/utmp'* y *'/usr/adm/wtmp'* , tanto de la ejecución actual como de las anteriores, se mantiene un historial de todos los procesos *'login/logout'* que se llevan a cabo, incluyendo el reinicio del sistema.

- *Gestiona la desconexión de las terminales:* Cada vez que muere un proceso asociado a una terminal o, lo que es lo mismo, cada vez que una línea queda libre, se vuelve a examinar el fichero *'/etc/ttytab'* para buscar nuevas conexiones.

- *Maneja los errores de conexión de las terminales:* En el caso de que una terminal intente conectarse infructuosamente un número determinado de veces, o en el caso de que en el intento de conexión se produzcan errores graves, entonces se deshabilita la terminal y, además, esta se ignora hasta que se solicite una nueva lectura del fichero *'/etc/ttytab'* mediante la señal *'SIGHUP'*.

4 Pasos de la ejecución de Init.

- *Ejecución del fichero '/etc/rc':* Podríamos considerar a este fichero como el equivalente del *'autoexec.bat'* que encontramos en los sistemas basados en DOS. Básicamente su misión es la de configurar los parámetros del sistema que, además de a los procesos de usuario, sirven de apoyo a la ejecución del Init. Entre estos encontramos: path del sistema, iniciación del teclado, establecimiento de la fecha y zona horaria, servicios de red, ... Además, se realizan otras tareas, tales como montar la partición *'/usr'*, comprobar que en la ejecución anterior se produjo un cierre correcto del sistema y, en lo concerniente a Init, preparar los ficheros *'/etc/utmp'* y *'/usr/adm/wtmp'* (se procura que este último no crezca indefinidamente).

- *Bucle principal (bucle infinito):* Una vez se hayan ejecutado todas las acciones del fichero anterior, Init entra en un bucle infinito. En este, principalmente, se gestionan los procesos que se comentaron en el punto 3 de este documento: *inicialización y enlace de cada una de las terminales*, *historial de las conexiones*, *desconexión de las terminales* y *manejo de los errores de conexión*. Además, dentro de este bucle se atiende a una serie de señales:

  - '*SIGHUP'*: Produce que los errores de todos los terminales se inicien a cero y que, posteriormente, buscando nuevas conexiones, se examine el fichero *'/etc/ttytab'*.

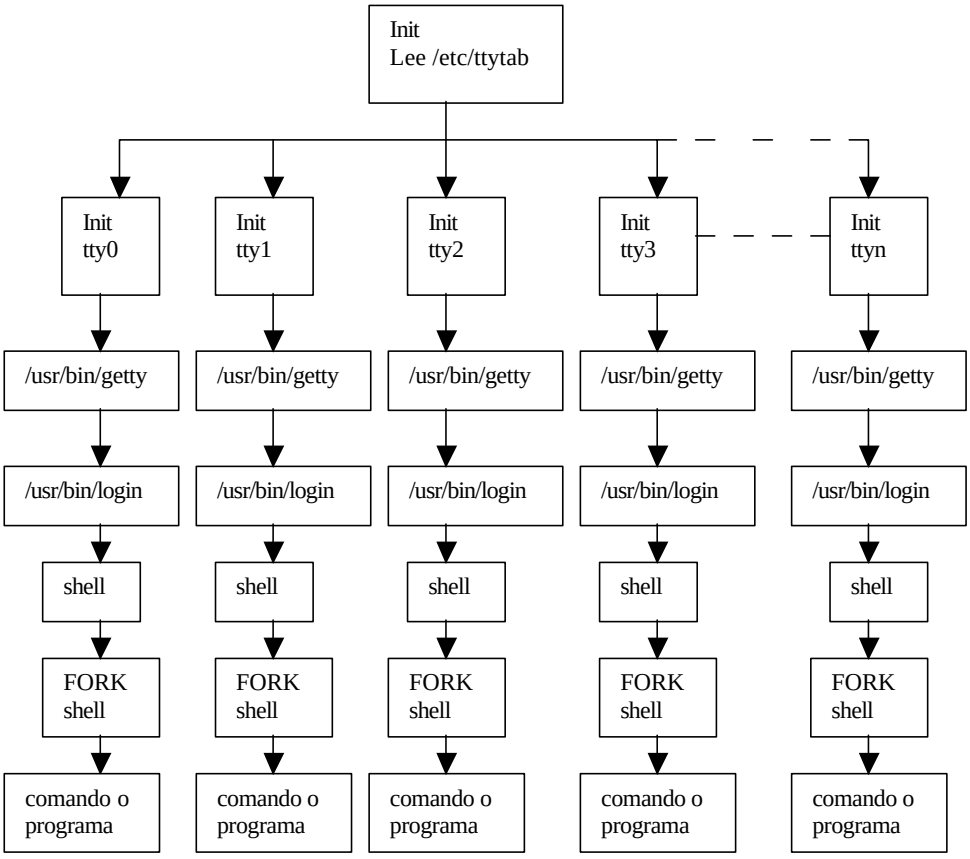
  - '*SIGABRT*': Se reinicia el sistema.

  Merece ser destacado que, una vez el cierre del sistema está cerca ('*SIGTERM'*), se desactiva un flag ('*spawn'*) que impide que nuevas terminales se conecten y que, por tanto, se prolongue dicho cierre.

- *Acciones de la función 'startup':* Esta función es llamada dentro del bucle anterior y, como ya se indicó en el punto 3 de este documento, *'startup'* es la responsable, entre comillas, de iniciar, conectar y asignar un proceso a una terminal. Antes que nada, como se describirá más adelante, en el fichero *'/etc/ttytab'* se asocia a cada terminal una serie de comandos, que son los que deben ser usados en la iniciación y conexión de dichas terminales. Para el primer paso, la iniciación, normalmente se usa el comando *'stty'*, el cual esencialmente lo que hace es configurar los parámetros físicos de la línea, tales como la paridad, la velociad de transmisión,... Luego, para el proceso de conexión se usa el comando *'getty';* éste pide la identificación del usuario, la cual, después, se usa para lanzar el programa *'login'*. Cabría destacar que el comando *'stty'* sólo se usa en caso de que la terminal sea desconocida como, por ejemplo, en una conexión remota en la que se usa módem.

- *Programa '/usr/bin/login':* Este programa, como se intuirá, solicita el *password* del usuario que se identificó previamente a través del comando *'getty'*. Si el resultado de la autenticación es satisfactorio, se lanza el shell indicado en el fichero *'/etc/passwd'*. Éste normalmente es *'/bin/sh'* o *'/usr/bin/ash'*.

- *Ejecución en el shell:* Por último, para lanzar comandos en el shell se hace uso de las funciones *'fork'* y *'exec'*; se crea un hijo del mencionado shell y se ejecuta en este último, mediante la rutina *'exec'*, el nuevo comando.



5 Señales esperadas por Init.

Init acepta varias señales, las cuales son enviadas con el identificador de proceso puesto a 1. A continuación se describen:

5.1 'SIGHUP'.

Cuando se recibe una señal '*hangup'*, Init se *olvida* de todos los errores y reexamina, en busca de nuevas conexiones, el fichero *'/etc/ttytab'*. Init sólo recorre *'/etc/ttytab'* cada vez que se le indica que puede habilitar una nueva terminal. Esto último pasa cuando una línea queda libre, o bien, cuando recibe la señal *'SIGHUP'*. Debemos apuntar que los errores sólo se resetean para este último caso.

5.2 'SIGTERM'.

'SIGTERM' es enviada cuando el cierre del sistema está cerca, ya sea porque se quiera apagar la máquina, o porque se quiera reiniciar. Init, cuando recibe esta señal, como medida para no prolongar el cierre del sistema, deja de conectar nuevas terminales.

5.3 'SIGABORT'.

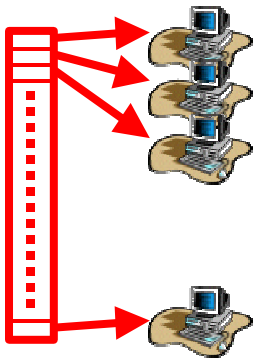
Esta señal es enviada por el Kernel, y provoca que el Init ejecute el comando *'shutdown'*.

6 Estructuras de datos.

6.1 Estructura 'slots'.

Asocia una terminal con un proceso.

```
Struct slotent{
int errct;
pid_t pid;
}
PIDSLOTS = 32;
Struct slotent slots[PIDSLOTS];
```



La estructura *'slotent'* contiene para cada terminal el *pid* del proceso asociado a esa línea (normalmente el del proceso que ejecuta el comando *'getty'*) y un número de errores. La gestión de estos últimos es simple:

- Si al intentar conectar la terminal sucede un error grave, como por ejemplo que no exista el dispositivo o el driver asociado a ella, entonces se deshabilita directamente la línea. Si, por el contrario, el error no es de estas características, entonces se incrementa en una unidad el contador de errores.

- Si el contador de errores supera cierto umbral, se deshabilita la línea.

- Cuando se recibe la señal *'SIGHUP'* se resetea el contador de errores de todas las terminales; se les da una nueva oportunidad.

Lo único que se pretende con todos estos pasos, es que una terminal que no se pueda iniciar de forma satisfactoria, no esté continuamente *molestando*.

6.2 Estructura 'ttyent'.

Para cada terminal se define: el nombre, el tipo y los comandos que la inician y la configuran.

```
Struct ttyent{
   char *ty_name; /* nombre de la terminal */
   char *ty_type; /* tipo de la terminal */
   char **ty_getty; /* comando para conectar la terminal */
   char **ty_init; /* comando para iniciar la terminal */
}
Struct ttyent *ttyp;
```

El comando que normalmente se usa para iniciar una terminal es el *'stty'*, mientras que para realizar la conexión, de nuevo, *normalmente* se usa *'getty'*. Las operaciones que realizan estas ya se comentaron en el apartado 3 del punto 4.

6.3 Ejemplo del fichero '/etc/ttytab'.

Cada entrada del fichero *'/etc/ttytab'* contiene exactamente los mismos campos (columnas en el fichero) que la estructura *'ttyent'*. A continuación se muestra un ejemplo de dicho archivo.

```
#ttytab - terminals
#
#Device  Type      Program    Init
console  minix     getty
ttyc1    minix     getty
ttyc2    minix     getty
ttyc3    minix     getty
tty00    unknown
tty01    unknown
ttyp1    network
ttyp2    network
ttyp3    network
```

7 Funciones de Init.

7.1 Funciones que manejan las señales.

7.1.1 Función 'onhup'.

Esta es la función asociada a la señal *'SIGHUP'*. Lo único que hace es activar dos flags:

- *'gothup'*: Indica que se deben *olvidar* los errores de todas las terminales, y que se debe reexaminar el fichero *'/etc/ttytab'* para buscar nuevas conexiones.

- *'spawn'*: Cuando este flag se inicia a 1, se establece que pueden seguir realizándose nuevas conexiones.

7.1.2 Función 'onterm'.

Esta función está asociada a la señal *'SIGTERM'*, y tan sólo contiene una línea. En esta se desactiva el flag *'spawn'*, es decir, se indica que no pueden seguir realizándose nuevas conexiones.

7.1.3 Función 'onabrt'.

La función *'onabrt'* está asociada a la señal *'SIGABRT'*, y su misión es la de llamar al comando *'shutdown'* cuando se haga el saludo de los tres dedos ('*Ctrl-Alt-Del'*).

7.2 Función ‘*startup*’.

Esta función, como ya se comentó en el apartado 3 del punto 4, tiene la misión de iniciar y conectar una nueva terminal. Además, en ella también se desarrolla el control de errores (descrito en el punto 6.1) y el registro de los procesos ‘*logins*’ en los ficheros *‘/etc/utmp’* y *‘/usr/adm/wtmp’*.

7.3 Función ‘execute’.

Esta función ejecuta el comando que se le pasa por parámetro. La diferencia que introduce, respecto a otras de su estilo, es que la ejecución la intenta realizar en cuatro rutas diferentes; hasta que la operación tenga éxito. Las rutas en las que prueba son: *‘/sbin’, ‘/bin’, ‘/usr/sbin’* y *‘/usr/bin’*.

7.4 Función ‘*wtmp*’.

Esta función es la encargada de registrar en los ficheros *‘/etc/utmp’* y *‘/usr/adm/wtmp’* todos los procesos *‘login/logout’* que se realicen, incluyendo el reinicio del sistema.

7.5 Funciones para escribir los errores.

7.5.1 Función ‘tell’.

La función *‘tell’* escribe una ristra de caracteres en un fichero.

7.5.2 Función ‘report’.

Esta función escribe un mensaje de error en un fichero. Primero se escribe la palabra *‘init:’*, depués una etiqueta del error y, por último, se añade una pequeña descripción del mismo.

8 Preguntas

8.1 ¿En qué momento se ejecuta Init?

Después de que se cargue el Kernel y los servicios útiles a los procesos de usuario, es decir, el manejador de memoria y el sistema de archivos. (Mirar punto 2)

8.2 ¿Qué tareas realiza Init?

Inicializa y enlaza cada una de las terminales, mantiene un historial de las conexiones, gestiona la desconexión de las terminales y maneja los errores de conexión. (Mirar punto 3)

8.3 ¿Por qué Init se queda residente en memoria?

Puesto que Init es el padre de todos los procesos de usuario y, además, el encargado de gestionar la conexión y desconexión de las terminales, se hace imprescindible que éste siempre se encuentre en ejecución. Sin él, los usuarios ni siquiera podrían identificarse y realizar un *login*.

9 Código.

```
/* 'init' es el padre de todos los procesos de usuario.
 * Cuando Minix se inicia, este es el proceso número 2, y tiene a 1 su pid.
 * Ejecuta el fichero de shell '/etc/rc', y luego lee el archivo '/etc/ttytab'
 * para determinar qué terminales necesitan un proceso 'login'.
 *
 * Si los ficheros '/usr/adm/wtmp' y '/etc/utmp' existen y son escribibles,
 * 'init' mantiene en ellos el recuento de todos los procesos 'login'. Enviar
 * una señal 1 (SIGHUP) al 'init' provoca un reescaneo de '/etc/ttytab' y, si es
 * necesario, el inicio de nuevos procesos de shell. Por otro lado, 'init' no
 * mata los procesos 'login' de las líneas que se han desactivado; esto se hace
 * manualmente.La señal 15 (SIGTERM) hace que el 'init' pare de lanzar nuevos
 * procesos, puesto que esta implica que el cierre del sistema está cerca. */

#include <sys/types.h>       /* Ficheros cabecera */
#include <sys/wait.h>
#include <sys/stat.h>
#include <ttyent.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include <utmp.h>

/* Orden a ejecutar cuando se produce el saludo de los tres dedos */
char *REBOOT_CMD[] =    { "shutdown", "now", "CTRL-ALT-DEL", NULL };

/* Entrada para marcar el inicio del sistema */
struct ttyent TT_REBOOT = { "console", "-", REBOOT_CMD, NULL };

char PATH_UTMP[] = "/etc/utmp";     /* 'logins' en curso */
char PATH_WTMP[] = "/usr/adm/wtmp";/* Historial de login/logout */

/* Define el número máximo de terminales que pueden estar encendidas */
#define PIDSLOTS  32

/* Estructura usada para hacer referencia a las terminales conectadas */
struct slotent {
  int errct;      /* Contador de errores */
  pid_t pid;      /* 'pid' del proceso de 'login' para esta línea 'tty' */
};

#define ERRCT_DISABLE   10  /* Deshabilitar si se producen tantos errores */
#define NO_PID    0       /* Valor del 'pid' que indica que no hay procesos */

/* tabla de las terminales conectadas */
struct slotent slots[PIDSLOTS];

int gothup = 0; /* bandera que indica que se ha recibido la señal 1 (SIGHUP)*/
int gotabrt = 0; /* bandera que indica que se ha recibido la señal 6 (SIGABRT)*/
/* bandera que indica si se pueden seguir enviando procesos*/
int spawn = 1;

/* Prototipos de las funciones de este módulo */
void tell(int fd, char *s);
```

```
void report(int fd, char *label);
void wtmp(int type, int linenr, char *line, pid_t pid);
void startup(int linenr, struct ttyent *ttyp);
int execute(char **cmd);
void onhup(int sig);
void onterm(int sig);
void onabrt(int sig);

int main(void)
{
  pid_t pid;                  /* 'pid' del proceso hijo */
  int fd;                     /* generalmente útil */
  int linenr;                 /* variable de bucle */
  int check;                  /* chequea si un proceso debe ser lanzado */
  struct slotent *slotp;      /* puntero a 'slots[]' */
  struct ttyent *ttyp;        /* entrada del fichero 'ttytab' */
  struct sigaction sa;
  struct stat stb;

  if (fstat(0, &stb) < 0) {
      /* Abrimos la entrada, salida y el error estándar */
      /* Sirve únicamente para capturar los tres descriptores de fichero;
       * todavía no se van a usar */
      (void) open("/dev/null", O_RDONLY);
      (void) open("/dev/log", O_WRONLY);
      dup(1);
  }

  /* En el campo 'sa_mask' se especifican las acciones que deberían bloquearse
   * durante la ejecución del manejador de señal. A continuación esta variable
   * se inicia al conjunto vacío */
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = 0;

  /* Se prepara para recibir cualquiera de las tres siguientes señales: */

  /* 1) Señal 'onhup' (inicio): indica que debe revisarse el fichero
   * '/etc/ttytab' para habilitar nuevas líneas de terminal */
     sa.sa_handler = onhup;
     sigaction(SIGHUP, &sa, NULL);

  /* 2) Señal 'onterm' (terminación): indica que se está intentando cerrar el
   * sistema y que, por tanto, no se deben seguir lanzando procesos 'login'. */
  sa.sa_handler = onterm;
  sigaction(SIGTERM, &sa, NULL);

  /* 3) Señal 'onabrt' (abortar): señal enviada por el kernel cuando se pulsa
   * CTRL-ALT-DEL (cae el sistema) */
  sa.sa_handler = onabrt;
  sigaction(SIGABRT, &sa, NULL);

  /* Ejecución del fichero '/etc/rc' */
  if ((pid = fork()) != 0) {
      /* Proceso padre: este sólo espera a que el hijo ejecute el fichero
       * '/etc/rc' o, en su defecto, se reinicie el sistema */
      while (wait(NULL) != pid) {
            if (gotabrt) reboot(RBT_HALT);
      }
  } else {
      /* Proceso hijo: lanza un shell y ejecuta el fichero '/etc/rc' */
```

```
      /* Prepara el comando que va a ejecutar: 'sh /etc/rc' */
      static char *rc_command[] = { "sh", "/etc/rc", NULL, NULL };

#if __minix_vmd
      /* Minix-vmd: Coge las opciones de 'boot' del entorno de 'boot'. */
      rc_command[2] = getenv("bootopts");
#else
      /* Minix: Entrada desde la consola. */
      close(0);
      (void) open("/dev/console", O_RDONLY);
#endif

      execute(rc_command); /* Se manda a ejecutar el fichero '/etc/rc' */
      report(2, "sh /etc/rc"); /* Se informa al usuario de los errores */
      _exit(1);   /* Si la ejecución ha sido correcta, no se debería llegar
                   * aquí */
  }

  /* Si existe, borra el fichero '/etc/utmp'; es decir, de la ejecución anterior
   * del sistema se borra la historia de los procesos 'login' */
  if ((fd = open(PATH_UTMP, O_WRONLY | O_TRUNC)) >= 0) close(fd);

  /* Registra en el fichero '/usr/adm/wtmp' (historia global de los
   * 'login/logout') el inicio del sistema */
  wtmp(BOOT_TIME, 0, NULL, 0);

  /* Bucle principal:
   * - Comprueba si alguna terminal se ha desconectado. Si es así, se prepara
   *    para realizar una nueva conexión (se comprueban los 'slots' de todas las
   *    terminales).
   * - Si los procesos de 'login' ya han empezado, espera a que uno termine, o
   *   a que se reciba alguna señal para continuar ('SIGHUP').
   * - Cuando se recibe la señal 'SIGTERM', es decir, cuando el cierre del
   *   sistema está cerca, se indica, mediante la variable 'spawn', que ya no se
   *   pueden seguir enviando procesos de 'login'.
   * - La información sobre las terminales que se van a conectar, se recoge,
   *   usando las funciones 'gettyent' y 'endttyent', del fichero 'ttytab'. */

  check = 1; /* esta variable se usa para indicar que se pueden seguir
conectando terminales */
  while (1) {
      while ((pid = waitpid(-1, NULL, check ? WNOHANG : 0)) > 0) {
            /* Comprueba qué línea  terminó su ejecución */
            for (linenr = 0; linenr < PIDSLOTS; linenr++) {
                  slotp = &slots[linenr];
                  if (slotp->pid == pid) {
                        /* Graba el fin del proceso en el historial */
                        wtmp(DEAD_PROCESS, linenr, NULL, pid);
                        slotp->pid = NO_PID; /*Ya no hay proceso en ese 'tty'*/
                        check = 1; /*Permite lanzar un nuevo proceso en algún
                                    * 'tty' */
                  }
            }
      }



      /* Si se recibe la señal 'SIGHUP' (una nueva línea ha sido abierta), se
       * resetea el contador de errores de todas las terminales */
      if (gothup) {
```

```
            gothup = 0;
            for (linenr = 0; linenr < PIDSLOTS; linenr++) {
                  slots[linenr].errct = 0;
            }
            check = 1; /* Se pueden conectar nuevas terminales */
      }

      /* Reinicio al recibir la señal 6 (SIGABRT)*/
      if (gotabrt) {
            gotabrt = 0;
            startup(0, &TT_REBOOT);
      }

      /* Se lanzan nuevos procesos sólo si el sistema no se está cerrando
       * ('spawn') y si, además, se pueden conectar nuevas terminales ('check')
      */
      if (spawn && check) {
            /* Comprueba qué líneas pueden activar un proceso 'login' */
            for (linenr = 0; linenr < PIDSLOTS; linenr++) {
                 /* Recorre la lista de posibles terminales*/
                  slotp = &slots[linenr];
                  /* Lee una entrada del '/etc/ttytab' */
                  if ((ttyp = getttyent()) == NULL) break;
                  if (ttyp->ty_getty != NULL  /*Existe el proceso 'getty' */
                        && ttyp->ty_getty[0] != NULL
                        && slotp->pid == NO_PID /*No hay un proceso en ejecución
                                                 * en esa terminal*/
                        && slotp->errct < ERRCT_DISABLE)
                  {
                        startup(linenr, ttyp); /* Se inicia la conexión con la
                                                * terminal */
                  }
            }
            endttyent();
      }
      check = 0;  /* Mientras no se cierre alguna de las terminales abiertas, o
                   * bien, hasta que no se produzca una nueva recepción de la
                   * señal 'SIGHUP', no se permite lanzar nuevos procesos */
   }
}

/* ---------------- FUNCIONES QUE ATIENDEN A LAS SEÑALES -------------- */

/* Los tres procedimientos que vienen a continuación se ejecutan para modificar
 * variables de estado del 'init' */

void onhup(int sig)
/* Se llama a esta función cuando se recibe la señal 'SIGHUP'; cuando se van a
 * habilitar nuevas líneas de terminal */
{
  gothup = 1; /* Habilitar nuevas líneas de terminal */
  spawn = 1; /* Indica que se pueden seguir conectando terminales */
}

void onterm(int sig)
/* Se llama a esta función cuando el sistema se está intentando cerrar */
{
  spawn = 0; /* Indica que no se pueden seguir conectando terminales */
}

void onabrt(int sig)
```

```
/* Se llama a esta función cuando se recibe la señal CTRL-ALT-DEL (cae el
 * sistema) */
{
  static int count;

   /* La primera vez que se entra a esta función simplemente se indica que el
    * sistema debe ser reiniciado. Si se entra por segunda vez, es decir, en
    *caso de que no se trate la señal fuera, se reinicia el sistema desde aquí
   */
  if (++count == 2) reboot(RBT_HALT);
  gotabrt = 1;
}

/* ---------- FUNCION 'startup'  (inicia una terminal) ------------------- */

void startup(int linenr, struct ttyent *ttyp)
/* Esta función inicia y lanza un proceso para la línea (términal) indicada */
{


  struct slotent *slotp; /* puntero a una entrada de la tabla de terminales */
  pid_t pid;             /* nuevo 'pid' */
  int err[2];           /* 'pipe' para la comunicación de los errores */
  char line[32];        /* nombre de la 'tty' */
  int status;

  /* Obtiene de la tabla de 'ttys', según la entrada 'liner', el terminal a
   * tratar */

   slotp = &slots[linenr];

  /* A continuación se lanza un 'pipe'. Este retorna dos descriptores, uno de
   * lectura (err[0]) y otro de escritura (err[1]). Estos, luego, se usarán para
   * escribir y leer los errores que se pudieran producir durante la iniciación
   * de la terminal o el lanzamiento del proceso 'login' */

  /* Comprueba si se puede crear el 'pipe' */
  if (pipe(err) < 0) err[0] = err[1] = -1;

  if ((pid = fork()) == -1 ) {  /*Error en el fork*/
      report(2, "fork()");
      sleep(10);
      return;
  }

  if (pid == 0) {
      /* Proceso hijo (1): éste se encarga de configurar la terminal e iniciar
       * el proceso de conexión ('login')*/

      close(err[0]);  /*Lo cierra porque el hijo no lee errores, los escribe*/

      /* La línea que viene a continuación simplemente añade al descriptor de
       * fichero 'err[1]' (donde se escriben los errores de la iniciación) el
       * flag 'CLOSE ON EXEC'. Este flag provoca que el fichero 'err[1]' se
       * cierre una vez que el hijo haya ejecutado algún proceso */
      fcntl(err[1], F_SETFD, fcntl(err[1], F_GETFD) | FD_CLOEXEC);



      /* Inicio de la sesión. Después de este comando, todos los procesos
       * ejecutados en la terminal pertenecerán al mismo grupo de procesos. */
```

```
setsid();

/* Construye, para su apertura, el nombre del 'tty' */
strcpy(line, "/dev/");
strncat(line, ttyp->ty_name, sizeof(line) - 6);

/* Abre la terminal como la entrada y la salida estándar */
close(0);
close(1);
if (open(line, O_RDWR) < 0 || dup(0) < 0) {
      /* Si el proceso hijo no pudo abrir la terminal, escribe el error en
       * el 'pipe' */
      write(err[1], &errno, sizeof(errno));
      _exit(1);
}


if (ttyp->ty_init != NULL && ttyp->ty_init[0] != NULL) {
      /* Ejecuta el comando que inicia la línea de la terminal ('stty').
       * Básicamente, las funciones que realiza 'stty' son comprobar la
       * paridad y la velocidad de transmisión de la terminal */

      /* FORK: Un nuevo 'hijo' ejecuta el comando 'stty' y el padre
       * ('hijo' anterior) espera a que acabe */
      if ((pid = fork()) == -1) {  /*Error en el fork*/
            report(2, "fork()");
            errno= 0;
            write(err[1], &errno, sizeof(errno));
            _exit(1);
      }
      /* Proceso hijo (2): ejecución del comando 'stty' */
      if (pid == 0) {
            alarm(10); /* Se dejan 10 segundos para que se inicie la
                        * terminal */
            execute(ttyp->ty_init);     /* Llamada a 'stty' */
            report(2, ttyp->ty_init[0]);/* Error en la ejecución */
            _exit(1);
      }

      /* Proceso padre (2): espera a que se inicie la terminal y, luego,
       * comprueba el resultado de tal operación */
      while (waitpid(pid, &status, 0) != pid) {}
      if (status != 0) { /* Examina la condición de fin del 'hijo (2)' */
            /* Hubo un error. Se informa de la situación */
            tell(2, "init: ");
            tell(2, ttyp->ty_name);
            tell(2, ": ");
            tell(2, ttyp->ty_init[0]);
            tell(2, ": bad exit status\n");
            errno = 0;
            write(err[1], &errno, sizeof(errno));
            _exit(1);
      }
}




/* Redirecciona la salida estándar */
dup2(0, 2);
```

```
    /* Ejecuta el proceso 'getty'. Este comando pide el nombre del usuario y
     * llama al proceso 'login' */
    execute(ttyp->ty_getty);

    /* Aquí no se debería llegar. Hubo un error en la ejecución del 'getty' */
    fcntl(2, F_SETFL, fcntl(2, F_GETFL) | O_NONBLOCK);
    if (linenr != 0) report(2, ttyp->ty_getty[0]);
    write(err[1], &errno, sizeof(errno));
    _exit(1);
}


/* Proceso padre (1): comprueba si se ha producido algún error en la
 * iniciación de la terminal o en la ejecución del comando 'getty'. Esta
 * comprobación se hace mediante la lectura del fichero que se creo con
 * 'pipe', fichero que actualizaba el proceso hijo (1) cuando intentaba
 * realizar estas tareas */

if (ttyp != &TT_REBOOT) /* No es un reinicio*/
    slotp->pid = pid;   /* PID del proceso (su hijo) que se ejecuta en esa
                         * línea */

close(err[1]);  /* Cierra ese descriptor del 'pipe' porque el 'padre'
                 * no escribe errores, los lee */
if (read(err[0], &errno, sizeof(errno)) != 0) {

    /* Se produjeron errores en la iniciación de la terminal o en el
     * lanzamiento del comando 'getty' */

    switch (errno) {
    case ENOENT:
    case ENODEV:
    case ENXIO:
    /* Dispositivo inexistente, no existe el driver,... */
          slotp->errct = ERRCT_DISABLE; /*No podrá iniciarse este 'tty' */
          close(err[0]);
          return;
    case 0:
          /* El error ya fue indicado */
          break;
    default:
          /* Otro error en la línea */
          report(2, ttyp->ty_name);
    }
    close(err[0]);

    if (++slotp->errct >= ERRCT_DISABLE) {
       /* Se alcanzó el número máximo de errores permitido; se informa de la
        * situación al usuario */
          tell(2, "init: ");
          tell(2, ttyp->ty_name);
          tell(2, ": excessive errors, shutting down\n");
    } else {
          sleep(5);
    }
    return;
}
close(err[0]);
/* Si no es un reinicio, almacena la historia referente al proceso que ejecuta
 * el 'login' */
```

```
  if (ttyp != &TT_REBOOT) wtmp(LOGIN_PROCESS, linenr, ttyp->ty_name, pid);

  slotp->errct = 0;  /* Inicia a cero el contador de errores de la terminal,
                      * puesto que se pudo realizar la conexión */
}


/* ----------------- FUNCION PARA EJECUTAR LOS COMANDOS ----------------- */

int execute(char **cmd)
{
  /* Esta función ejecuta el comando pasado por parámetro (‘cmd’). Para realizar
   * esta operación prueba en las rutas siguientes:’/sbin’ , ‘/bin’, ‘/usr/sbin’
   * y ‘/usr/bin’ */

  static char *nullenv[] = { NULL };
  char command[128];
  char *path[] = { "/sbin", "/bin", "/usr/sbin", "/usr/bin" };
  int i;

  if (cmd[0][0] == '/') {
      /* Si el comando comienza por ‘/’ , se considera que proporciona el Path
       * completo */
      return execve(cmd[0], cmd, nullenv);
  }

  /* En cada iteración del bucle se intenta ejecutar el comando con una de las
   * cuatro rutas diferentes; así hasta que se acierte */
  for (i = 0; i < 4; i++) {
      if (strlen(path[i]) + 1 + strlen(cmd[0]) + 1 > sizeof(command)) {
            errno= ENAMETOOLONG; /* Nombre muy grande */
            return -1;
      }
      /*Construye cada comando según la lista ‘path’*/
      strcpy(command, path[i]);
      strcat(command, "/");
      strcat(command, cmd[0]);
      execve(command, cmd, nullenv);
      /* A continuación se comprueba si el error que se produjo  no es debido a
       * que no existe el fichero. En ese caso se ‘rompe’ el bucle y se retorna
       * ‘-1’. Este último es el único error que se permite. Otros, debidos por
       * ejemplo a que no existe memoria o a que el binario no es compatible, no
       * son tolerados.  */

      if (errno != ENOENT) break;
  }
  return -1;  /*Ejecución imposible*/
}
```

```
/* -------- FUNCION PARA REGISTRAR LOS ‘LOGIN’ Y LOS ‘LOGOUT’ ------------ */
```

```
void wtmp(type, linenr, line, pid)
/* Este procedimiento actualiza los ficheros '/etc/utmp' y '/user/adr/wtmp';
 * ficheros en los que se almacenan todos los procesos 'login/logout' que se han
 * producido durante la ejecución actual del sistema, y durante la historia
 * global del mismo */

int type;               /* tipo de entrada */
int linenr;             /* número de línea en 'ttytab' */
char *line;             /* nombre del 'tty' (sólo util en login) */
pid_t pid;              /* pid del proceso */
{

  struct utmp utmp;
  int fd;

  /* Borra el contenido de la estructura 'utmp'. */
  memset((void *) &utmp, 0, sizeof(utmp));

 /* Se completan los campos de la estructura que se va a escribir en el fichero
  * 'utmp'. */
  switch (type) {
  case BOOT_TIME:
      /* Hace un 'log' especial del reboot. */
      strcpy(utmp.ut_name, "reboot");
      strcpy(utmp.ut_line, "~");
      break;

  case LOGIN_PROCESS:
      /* Un nuevo 'login': se almacena el nombre de la línea para luego copiarlo
       * en el fichero */
      strncpy(utmp.ut_line, line, sizeof(utmp.ut_line));
      break;

  case DEAD_PROCESS:
      /* Un logout. Usa la entrada actual del 'utmp' para asegurarse de que
       * de verdad se trata de un 'logout' y no de un 'getty' o un 'login'
       * fallido */
      if ((fd = open(PATH_UTMP, O_RDONLY)) < 0) {
            if (errno != ENOENT) report(2, PATH_UTMP);
            return;
      }
      if (lseek(fd, (off_t) (linenr+1) * sizeof(utmp), SEEK_SET) == -1
            || read(fd, &utmp, sizeof(utmp)) == -1
      ) {
            report(2, PATH_UTMP);
            close(fd);
            return;
      }
      close(fd);
      if (utmp.ut_type != USER_PROCESS) return;
      /* Si se llega hasta aquí quiere decir que se trata de un 'logout' y no de
       * un 'getty' o de un login fallido */
      strncpy(utmp.ut_name, "", sizeof(utmp.ut_name));
      break;
  }



  /* Escribe los datos que faltan para completar la nueva entrada del fichero
   * 'utmp' */
```

```
  utmp.ut_pid = pid;
  utmp.ut_type = type;
  utmp.ut_time = time((time_t *) 0);

  /* Si se trata de un 'login' o de un 'logout' se escribe la nueva estructura
   * en el fichero '/etc/utmp' */
  switch (type) {
  case LOGIN_PROCESS:
  case DEAD_PROCESS:
      if ((fd = open(PATH_UTMP, O_WRONLY)) < 0 /* Escribe la nueva entrada */
            || lseek(fd, (off_t) (linenr+1) * sizeof(utmp), SEEK_SET) == -1
            || write(fd, &utmp, sizeof(utmp)) == -1
      ) {
            if (errno != ENOENT) report(2, PATH_UTMP);
      }
      if (fd != -1) close(fd);
      break;
  }


  /* Si se trata de un 'logout' o de la iniciación del sistema, los datos se
   * escriben en el fichero '/user/adr/wtmp' */
  switch (type) {
  case BOOT_TIME:
  case DEAD_PROCESS:
      /* Escribe una nueva entrada en el fichero 'wtmp' */
      if ((fd = open(PATH_WTMP, O_WRONLY | O_APPEND)) < 0
              || write(fd, &utmp, sizeof(utmp)) == -1
      ) {
            if (errno != ENOENT) report(2, PATH_WTMP);
      }
      if (fd != -1) close(fd);
      break;
  }
}


/* -------------- FUNCIONES PARA ESCRIBIR LOS ERRORES -------------------- */

void tell(fd, s)
/* Este procedimiento escribe en el fichero con descriptor 'fd' la ristra de
 * caracteres 's' */
int fd;
char *s;
{
      write(fd, s, strlen(s));
}
```

```
void report(fd, label)
/* Este procedimiento escribe en el fichero con descriptor 'fd' el error con
```

```
 * etiqueta ‘label’, acompañado, además, de una pequeña descripción */
int fd;
char *label;
{
      int err = errno;

      tell(fd, "init: ");
      tell(fd, label);
      tell(fd, ": ");
      tell(fd, strerror(err));
      tell(fd, "\n");
      errno= err;
}
```

Indice

Ampliación de Sistemas Operativos

Init.c

Sergio Sosa de la Fe
Ignacio José López Rodríguez