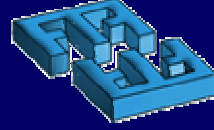
Facultad de Infórmatica



Diseño de Sistemas Operativos

Init.c

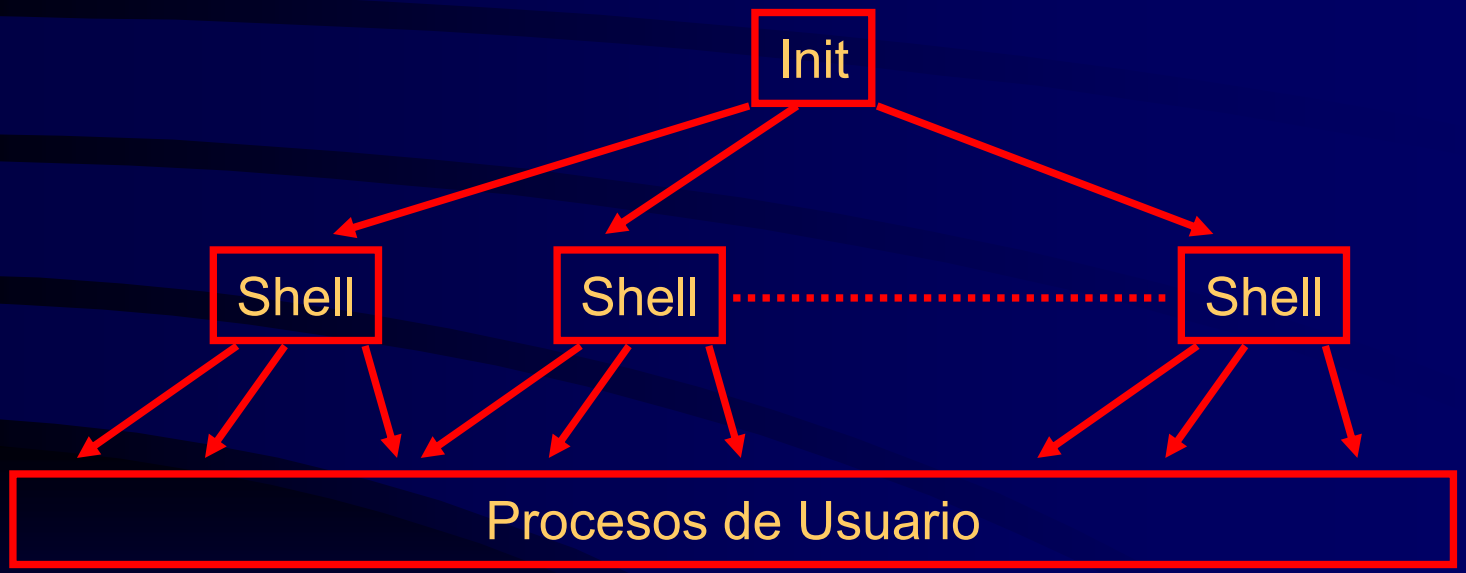
Autores: Francisco A. Ortega Muñoz
Noemí Guerra Sosa

Fecha: Abril 2002

Breve Descripción

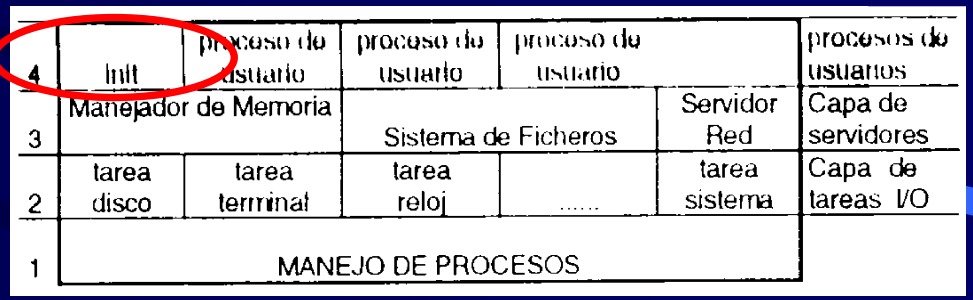
- Init es el padre de todos los procesos de usuario
- Para cada terminal crea un proceso hijo
  - Configuración de la línea.
  - Identificación del usuario.
  - Realizar el login.
  - Conexión de la terminal.

Arbol de procesos



Antes que init ...

- Manejo de procesos
- Tareas de E/S
- Procesos Servidores
- Procesos de usuario



- Manejador de Memoria (MM).
- Sistema de Ficheros (FS).
- Servidor de Red.

Tareas que realiza init

- Inicializa y enlaza cada una de las terminales.
- Mantiene un historial de las conexiones.
- Gestiona la desconexión de las terminales.
- Maneja los errores de conexión de las terminales.

Pasos de la ejecución de init

- Ejecución del fichero /etc/rc:
  - Path del sistema.
  - Inicialización del teclado.
  - Establecimiento de la fecha y de la zona horaria.
  - Preparación de los ficheros /etc/utmp y /etc/wtmp.
  - Se monta la partición /usr.
  - Comprobación del cierre correcto en la sesión anterior.
  - Inicialización de los servicios de red.
  - Se borran ficheros temporales.

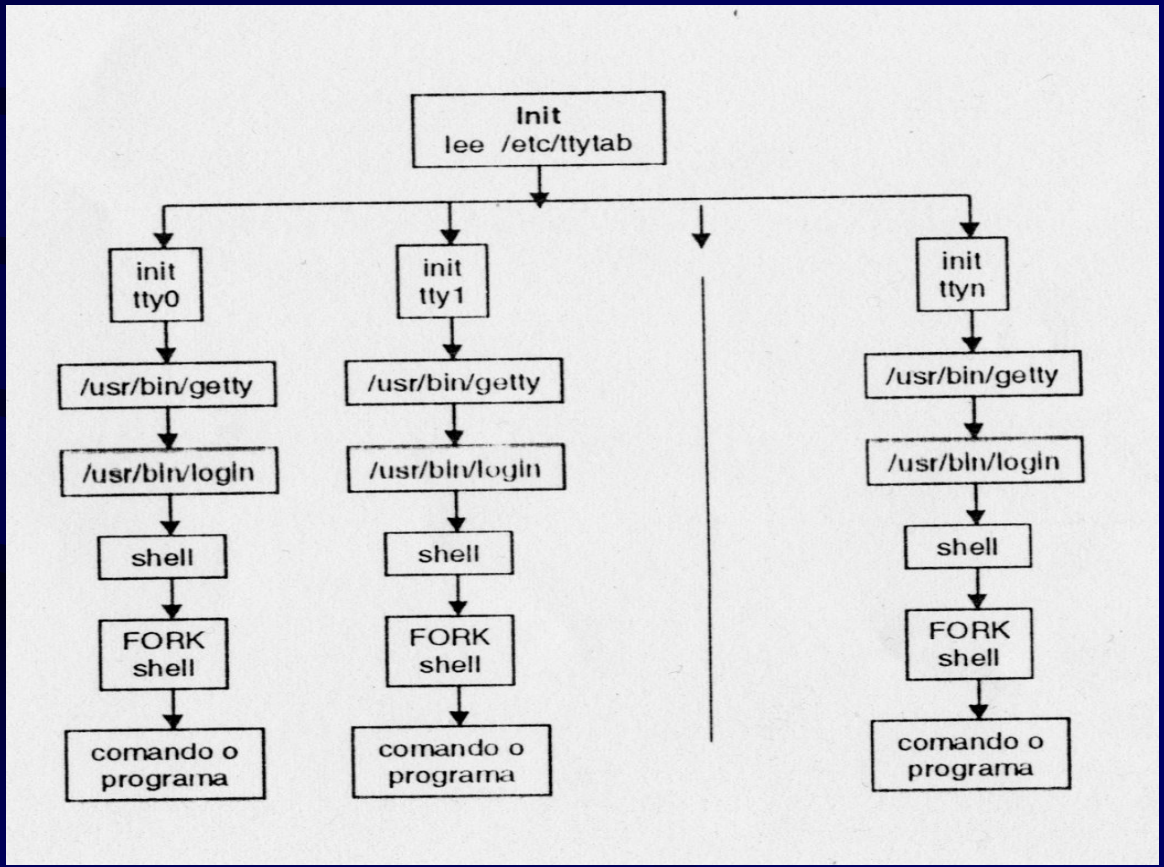
Pasos de la ejecución de init

- Bucle principal:
  - Inicialización de cada una de las terminales del fichero /etc/ttytab (startup)
  - Si un proceso hijo (terminal) finaliza se reexamina el fichero /etc/ttytab para buscar nuevas conexiones.

Pasos de la ejecución de init

- Acciones de startup:
  - Configurar los parámetros de la terminal (stty):
    - Velocidad de la línea
    - Paridad
  - Ejecutar el Getty:
    - Se pide la identificación del usuario
    - Se lanza el login
  - Ejecutar /usr/bin/login:
    - Si el login es correcto, lanza el shell indicado en /etc/passwd
  - Ejecución en el shell:
    - Fork
    - Exec

Pasos de la ejecución de init



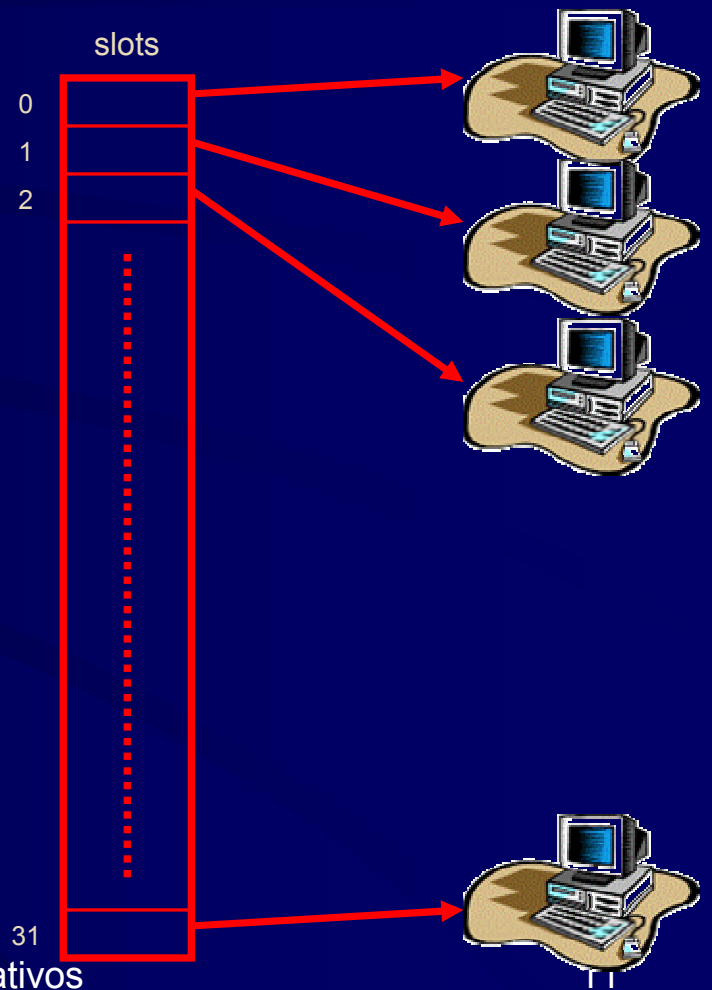
Señales esperadas por init

- SIGHUP
  - Se olvidan todos los errores.
  - Lee de nuevo el fichero /etc/ttytab.
  - Para cada terminal de /etc/ttytab que no tenga un proceso asociado, y que no tenga el número máximo de errores permitidos, se crea un proceso.
- SIGTERM
  - Provoca que init deje de generar procesos.
  - Normalmente se usa cuando se resetea o se apaga el sistema.
- SIGABRT
  - La envía el driver del teclado.
  - Provoca que se de la orden de apagar el sistema.

Estructuras de datos

- Estructura slotent
  - Asocia una terminal con un proceso

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



Estructuras de datos

- Estructura ttyent
  - Define para cada terminal: el nombre, el tipo y los comandos para configurar e iniciar una terminal

```
Struct ttyent{
    char *ty_name;
    char *ty_type;
    char **ty_getty;
    char **ty_init;
}
Struct ttyent *ttyp;
```

Estructuras de datos

- Fichero /etc/ttytab

```
#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
```

Llamadas al sistema

- Procesos
  - fork
  - wait
  - waitpid
  - exit
  - execve
  - setsid
- Señales
  - sigaction
  - alarm
- Ficheros
  - open
  - close
  - read
  - write
  - lseek
  - dup
  - fstat
  - pipe
  - fcntl

Código fuente

```
#include <sys/types.h>
#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>

char *REBOOT_CMD[] =   { "shutdown", "now", "CTRL-ALT-DEL", NULL };
struct ttyent TT_REBOOT = { "console", "-", REBOOT_CMD, NULL };
char PATH_UTMP[] = "/etc/utmp";
char PATH_WTMP[] = "/usr/adm/wtmp";
```

Init.c

- Se crea la estructura slots y se definen constantes relacionadas con esta: nº máximo de terminales y nº máximo de errores.

```
#define PIDSLOTS        32
struct slotent {
  int errct;
  pid_t pid;
};
#define ERRCT_DISABLE  10
#define NO_PID 0
struct slotent slots[PIDSLOTS];

int gothup = 0;
int gotabrt = 0;
int spawn = 1;
```

Init.c

- Se inicializan gothup, gotabrt y spawn.
  - gothup: Ha llegado la señal SIGHUP
  - gotabrt: Ha llegado la señal SIGABORT
  - spawn: Se pueden seguir conectando nuevas terminales

```
#define PIDSLOTS        32
struct slotent {
  int errct;
  pid_t pid;
};
#define ERRCT_DISABLE  10
#define NO_PID 0
struct slotent slots[PIDSLOTS];

int gothup = 0;
int gotabrt = 0;
int spawn = 1;
```

Init.c

- Declaración de prototipos de las funciones de este módulo.
- Declaración de la función main.
- Inicialización de variables:
  - check: Se debe reexaminar el fichero ttytab

```
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;
  int fd;
  int linenr;
  int check;
```

Init.c

- Se capturan los descriptores de la entrada(0), la salida(1) y el error(2) estándar.

```
struct slotent *slotp;
struct ttyent *ttyp;
struct sigaction sa;
struct stat stb;

if (fstat(0, &stb) < 0) {
        (void) open("/dev/null", O_RDONLY);
        (void) open("/dev/log", O_WRONLY);
        dup(1);
 }

sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
```

Init.c

- Se asocian funciones a las señales.

```
sa.sa_handler = onhup;
sigaction(SIGHUP, &sa, NULL);
sa.sa_handler = onterm;
sigaction(SIGTERM, &sa, NULL);
sa.sa_handler = onabrt;
sigaction(SIGABRT, &sa, NULL);
```

Init.c

- Se crea un proceso hijo para ejecutar el fichero /etc/rc. El padre espera a que este acabe.

```
if ((pid = fork()) != 0) {
    while (wait(NULL) != pid) {
        if (gotabrt) reboot(RBT_HALT);
    }
} else {
    static char *rc_command[] = { "sh", "/etc/rc", NULL, NULL };
    #if __minix_vmd
        rc_command[2] = getenv("bootopts");
    #else
        close(0);
        (void) open("/dev/console", O_RDONLY);
    #endif

    execute(rc_command);
    report(2, "sh /etc/rc");
    _exit(1);  /* No debería llegar aquí */
}
```

Init.c

- Se borra el fichero /etc/utmp.
- Se registra el inicio del sistema en el fichero /usr/adm/wtmp.

```
if ((fd = open(PATH_UTMP, O_WRONLY | O_TRUNC)) >= 0) close(fd);
wtmp(BOOT_TIME, 0, NULL, 0);
```

Init.c

Se inicia el bucle principal.

Espera a que el proceso asociado a una terminal muera.

- Se registra en los ficheros /usr/adm/wtmp y /etc/utmp.
- Se libera la entrada correspondiente en slots.
- Se debe reexaminar el fichero /etc/ttytab (check).

```
check = 1;
while (1) {
   while ((pid = waitpid(-1, NULL, check ? WNOHANG : 0)) > 0) {
      for (linenr = 0; linenr < PIDSLOTS; linenr++) {
         slotp = &slots[linenr];
         if (slotp->pid == pid) {
            wtmp(DEAD_PROCESS, linenr, NULL, pid);
            slotp->pid = NO_PID;
            check = 1;
         }
      }
   }
```

Init.c

- En caso de haber recibido la señal SIGHUP:
  - Se resetean los errores de las terminales.
  - Se debe reexaminar el fichero /etc/ttytab (check).

```
if (gothup) {
        gothup = 0;
        for (linenr = 0; linenr < PIDSLOTS; linenr++)
           slots[linenr].errct = 0;
        check = 1;
    }
```

Init.c

- Se reexamina el fichero /etc/ttytab
  - Se lee una entrada del fichero /etc/ttytab
  - Se comprueba que la terminal no supera el nº máximo de errores
  - Se conecta la terminal (startup)

```
    if (gotabrt) {
        gotabrt = 0;
        startup(0, &TT_REBOOT);
    }

    if (spawn && check) {
        for (linenr = 0; linenr < PIDSLOTS; linenr++) {
            slotp = &slots[linenr];
            if ((ttyp = getttyent()) == NULL) break;
            if (ttyp->ty_getty != NULL && ttyp->ty_getty[0] != NULL
&& slotp->pid == NO_PID && slotp->errct < ERRCT_DISABLE)
                startup(linenr, ttyp);
        }
        endttyent();
    }
        check = 0;
 }
}
```

Init.c

- Funciones para el manejo de las señales:

```
void onhup(int sig)
{
  gothup = 1;
  spawn = 1;
}

void onterm(int sig)
{
  spawn = 0;
}

void onabrt(int sig)
{
  static int count;
  if (++count == 2) reboot(RBT_HALT);
  gotabrt = 1;
}
```

Init.c

- Función startup. Se crea un proceso hijo.
- Se crea un pipe (err) para la comunicación de errores del hijo al padre

```
void startup(int linenr, struct ttyent *ttyp)
{
  struct slotent *slotp;
  pid_t pid;
  int err[2];
  char line[32];
  int status;

  slotp = &slots[linenr];

  if (pipe(err) < 0) err[0] = err[1] = -1;

  if ((pid = fork()) == -1 ) {
     report(2, "fork()");
     sleep(10);
     return;
  }
```

Init.c

- Se crea una sesión

```
if (pid == 0) {
   close(err[0]);
   fcntl(err[1], F_SETFD, fcntl(err[1], F_GETFD) | FD_CLOEXEC);

   setsid();

   strcpy(line, "/dev/");
   strncat(line, ttyp->ty_name, sizeof(line) - 6);

   close(0);
   close(1);
   if (open(line, O_RDWR) < 0 || dup(0) < 0) {
      write(err[1], &errno, sizeof(errno));
      _exit(1);
   }
```

Init.c

- Se abren la entrada y salida estándar de la terminal

```
if (pid == 0) {
   close(err[0]);
   fcntl(err[1], F_SETFD, fcntl(err[1], F_GETFD) | FD_CLOEXEC);

   setsid();

   strcpy(line, "/dev/");
   strncat(line, ttyp->ty_name, sizeof(line) - 6);

   close(0);
   close(1);
   if (open(line, O_RDWR) < 0 || dup(0) < 0) {
      write(err[1], &errno, sizeof(errno));
      _exit(1);
   }
```

Init.c

- El hijo crea una nuevo proceso.
- Este último inicializa la terminal.
- La terminal debe iniciarse antes de 10 segundos

```
if (ttyp->ty_init != NULL && ttyp->ty_init[0] != NULL) {

   if ((pid = fork()) == -1) {  /*Error en el fork*/
      report(2, "fork()");
      errno= 0;
      write(err[1], &errno, sizeof(errno));
      _exit(1);
   }

   if (pid == 0) {
      alarm(10);
      execute(ttyp->ty_init);
      report(2, ttyp->ty_init[0]);
      _exit(1);
   }
```

Init.c

- El proceso hijo inicial espera a que se inicie la terminal
- Se ejecuta el comando getty: Conexión de la terminal

```
    while (waitpid(pid, &status, 0) != pid) {}
    if (status != 0) {
        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);
    }
}

dup2(0, 2);

execute(ttyp->ty_getty);
```

Init.c

- El padre comprueba si se produjeron errores en la inicialización o en la conexión

```
    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);
}

if (ttyp != &TT_REBOOT)
    slotp->pid = pid;

close(err[1]);

if (read(err[0], &errno, sizeof(errno)) != 0) {
    switch (errno) {
    case ENOENT:
    case ENODEV:
    case ENXIO:
```

Init.c

- Se gestionan los errores
  - Errores graves: Se deshabilita directamente la terminal
  - Errores simples: Se incrementa en una unidad el nº de errores
- Si el nº de errores supera un umbral se deshabilita la terminal

```
        slotp->errct = ERRCT_DISABLE;
        close(err[0]);
        return;
    case 0:
        break;
    default:
        report(2, ttyp->ty_name);
    }
    close(err[0]);

    if (++slotp->errct >= ERRCT_DISABLE) {
        tell(2, "init: ");
        tell(2, ttyp->ty_name);
        tell(2, ": excessive errors, shutting down\n");
    } else {
        sleep(5);
    }
```

Init.c

- Si no se produjeron errores se registra el inicio de la conexión en /etc/utmp

```
        return;
    }

    close(err[0]);

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

    slotp->errct = 0;
}
```

Init.c

- Función execute.
  - Prueba a ejecutar un comando en cuatro rutas diferentes.
  - Si el comando empieza por '/' se considera que el path ya está completo

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

  if (cmd[0][0] == '/') {
    return execve(cmd[0], cmd, nullenv);
  }
```

Init.c

- Se prueba la ejecución del comando en cada una de las rutas.
  - Si se produce un error, y este no hace referencia a que no existe el fichero, se prueba el siguiente path.

```
for (i = 0; i < 4; i++) {
        if (strlen(path[i]) + 1 + strlen(cmd[0]) + 1 >
              sizeof(command)) {
                 errno= ENAMETOOLONG;
                 return -1;
        }
        strcpy(command, path[i]);
        strcat(command, "/");
        strcat(command, cmd[0]);
        execve(command, cmd, nullenv);

      if (errno != ENOENT) break;
  }
  return -1;
}
```

Init.c

- Función wtmp. Actualiza los ficheros wtmp y utmp.
- Se escribe en la estructura utmp el nombre de la línea.

```
void wtmp(type, linenr, line, pid)
int type;
int linenr;
char *line;
pid_t pid;                          {

  struct utmp utmp;
  int fd;

  memset((void *) &utmp, 0, sizeof(utmp));

 switch (type) {
  case BOOT_TIME:
        strcpy(utmp.ut_name, "reboot");
        strcpy(utmp.ut_line, "~");
        break;
  case LOGIN_PROCESS:
        strncpy(utmp.ut_line, line, sizeof(utmp.ut_line));
        break;
```

Init.c

```
case DEAD_PROCESS:
        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;
        strncpy(utmp.ut_name, "", sizeof(utmp.ut_name));
        break;
  }
```

Init.c

- Se completan el resto de los campos de utmp (hora, pid y tipo de proceso).
- El inicio y la muerte de los procesos asociados a una terminal se registran en el fichero utmp.

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

switch (type) {
  case LOGIN_PROCESS:
  case DEAD_PROCESS:
      if ((fd = open(PATH_UTMP, O_WRONLY)) < 0
       || 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;
}
```

Init.c

- El inicio del sistema y la muerte de los procesos asociados a una terminal se registran en el fichero 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;
    }
}
```

Init.c

- Funciones tell y report. Muestran los mensajes de error.

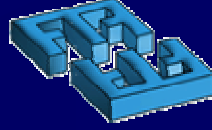
```
void tell(fd, s)
int fd;
char *s;
{
  write(fd, s, strlen(s));
}

void report(fd, label)
int fd;
char *label;
{
   int err = errno;
   tell(fd, "init: ");
   tell(fd, label);
   tell(fd, ": ");
   tell(fd, strerror(err));
   tell(fd, "\n");
   errno= err;
}
```

Preguntas

¿En qué momento se ejecuta Init?

¿Qué tareas realiza Init?

¿Por qué Init se queda residente en memoria?



Fin de la Presentación

Init.c

```
struct sigaction {
  __sighandler_t sa_handler;  //nombre de la función
  sigset_t sa_mask;    //señales a ser bloqueadas durante el manejo
  int sa_flags;         // flags especiales
}
```