PRÁCTICA 4

SHELL CON CONTROL DE TAREAS

EL TERMINAL

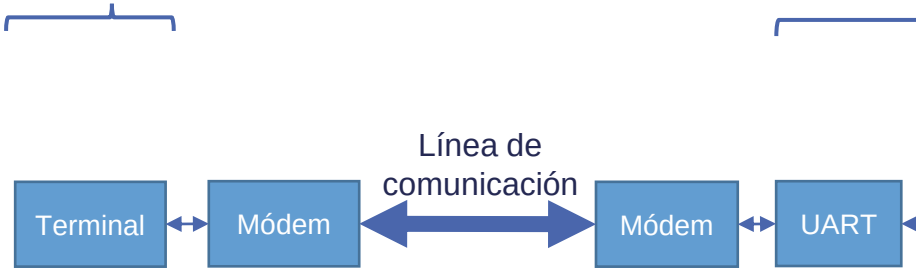
- **Historia**
  - 1869: *stock ticker* → precursor del teletipo
    - Máquina de escribir conectada por cable a una impresora
    - Propósito: distribuir precios de acciones a larga distancia en tiempo real
  - Teletipo (TTY): comienzos del siglo XX
    - Basado en ASCII
    - Conectados por todo el mundo:
      - Red Telex: red conmutada similar a la telefónica
    - Usados para comunicación de información:
      - Interna de gobiernos e industria
      - Militar
      - Pronóstico del tiempo
      - Prensa
      - Policía

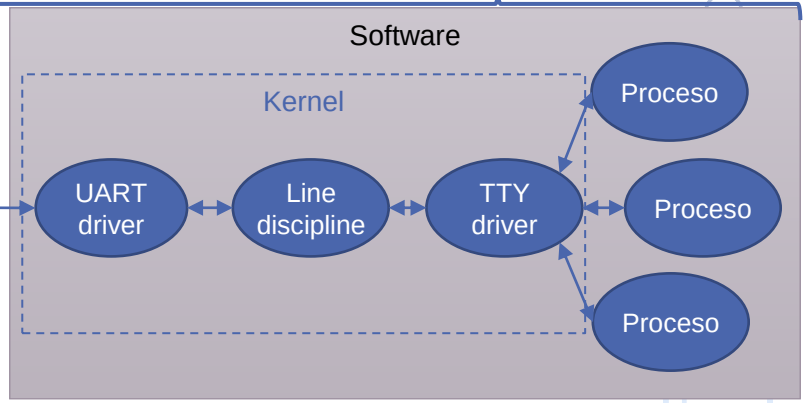
Teletipos en la WWII. *Fuente*: Wikipedia

EL TERMINAL

- **Historia**
  - Con la aparición de los computadores:
    - En la 3ª generación (1965-1971) empieza a introducirse la interacción con usuarios en tiempo real
    - Primero se utilizan teletipos y luego terminales con pantalla y teclado
      - *UART*: Universal Asynchronous Receiver-Transmitter. Se utiliza para la comunicación serie entre el terminal y el computador
      - *Line discipline*: interpreta ciertos caracteres (borrar, imprimir, ^C, ^Z,…)
      - *TTY driver*: manda señales a procesos, mantiene el que está *fg*,…

Línea de comunicación

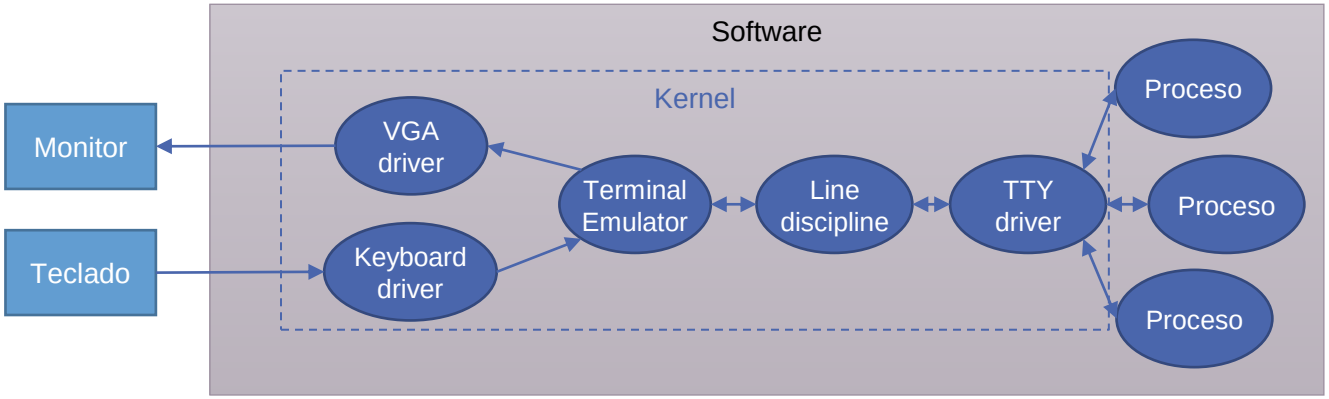




Fuente: http://www.linusakesson.net/programming/tty/index.php

EL TERMINAL

- **En Linux:**
  - Ya no existe el terminal físico → Ahora se **simula** (búfer para *frames* y máquina de estados)
  - Line discipline y TTY driver se mantienen. La UART ya no tiene sentido (aunque se pueden ver los baudios: `stty -a`)
  - **TTY device**: Terminal emulator + Line discipline + TTY driver
    - /dev/tty# → Ctrl+Alt+F# para cambiar entre tty's

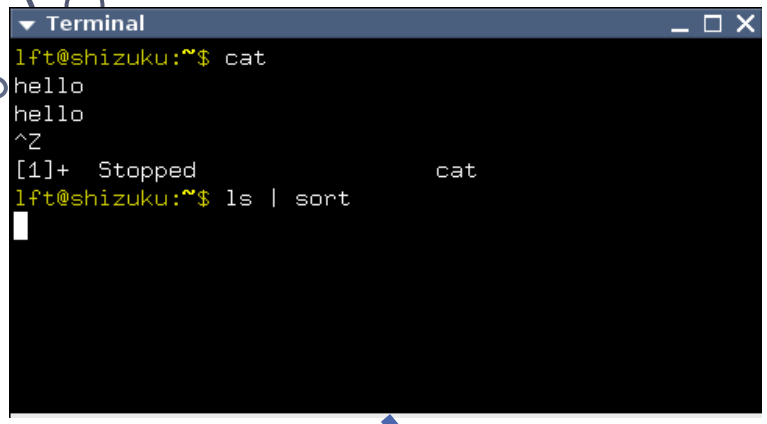


  - **Pseudo-terminal (PTY)**: Terminal llevado al espacio de usuario
    - /dev/pts/# → Lo que se abre en una ventana de terminal (xterm)
  - Comando `tty`: para conocer el *TTY/PTY* del terminal actual

TERMINAL, SESIÓN Y SHELL

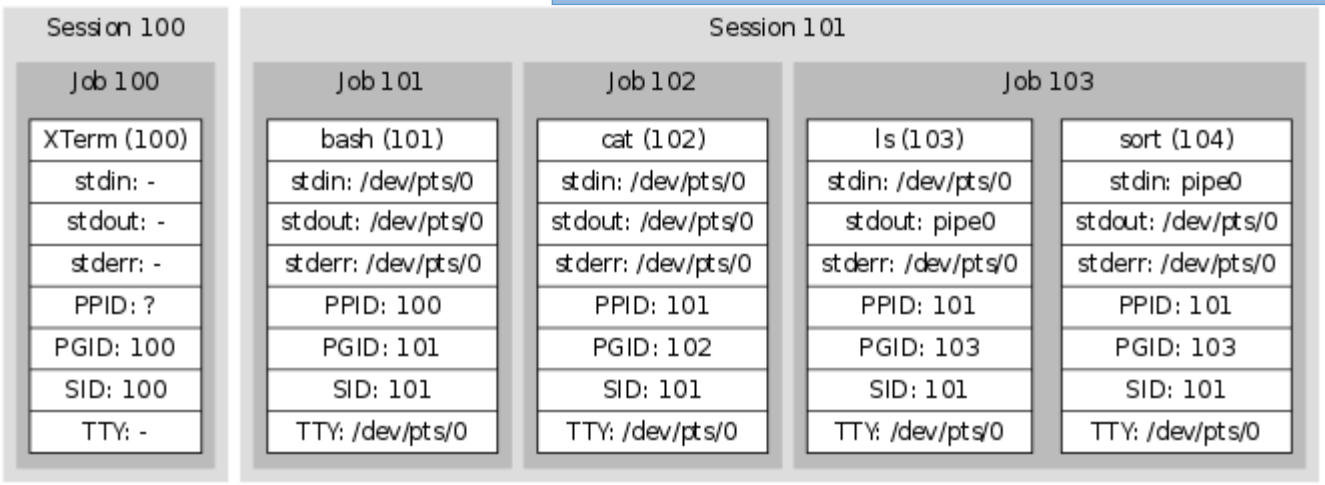
- Cuando se abre un terminal:
  - El demonio de login (*logind*) nos pedirá las credenciales
    - Se crea un identificador de sesión (SID)
  - Si son correctas se inicia un *shell* (e.g., `bash`)
    - El proceso *shell* es el líder de la sesión (PID == SID)
    - Todos sus hijos tendrán el mismo SID
    - Si el terminal se cierra, se manda SIGHUP al líder, que mandará SIGHUP a los hijos (véase comando `nohup`)
  - Cuando se teclea un comando en el *shell* se crea un proceso para ejecutarlo (`fork` y `exec`)
    - El comando puede ser una combinación de comandos (e.g., separados por *pipe*)
    - O el comando puede hacer `fork` y crear hijos
    - **Control de tareas:** Para facilitar el control de estos grupos de procesos se les asigna un *Process Group ID (PGID)*
      - ***Job o tarea**:* conjunto de procesos con el mismo PGID
      - Ejemplo: cuando el usuario pulsa ^Z se envía SIGTSTP al grupo/tarea/job

TERMINAL, SESIÓN Y SHELL



Estructuras del kernel

- TTY Driver (/dev/pts/0):
  - Size: 45x13
  - Controlling process group: (101)
  - Foreground process group: (103)
  - UART configuration (ignored, since this is an xterm): Baud rate, parity, word length and much more.
  - Line discipline configuration: cooked/raw mode, linefeed correction, meaning of interrupt characters etc.
  - Line discipline state: edit buffer (currently empty), cursor position within buffer etc.
- Pipe0:
  - Readable end (connected to PID 104 as file descriptor 0)
  - Writable end (connected to PID 103 as file descriptor 1)
  - Buffer



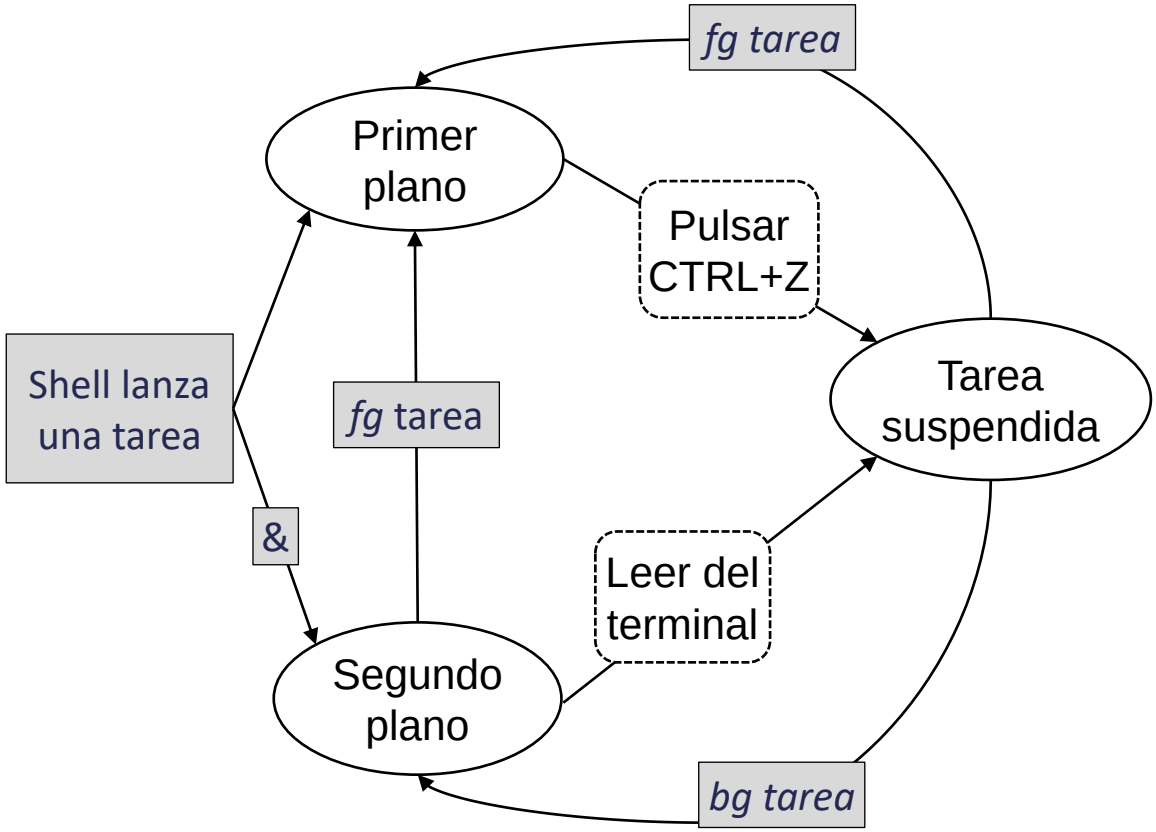
CONTROL DEL TERMINAL

- El **terminal** se asocia a un grupo de procesos (*job*)
- El *shell* **controla** qué tarea (*job*) accede al terminal en cada momento (`tcsetpgrp`)
- El grupo de procesos con el terminal es la tarea en **primer plano** (*fg*)
- Las demás que se ejecutan sin terminal se conocen como tareas en **segundo plano** (*bg*)
- El *shell* es la tarea en primer plano mientras lee los comandos
- El *shell* debe identificar a cada tarea (`setpgid`)

CONTROL DE TAREAS

- El *shell* debe:
  - Mantener una **lista de tareas** (*job list*)
    - Corriendo en segundo plano: pueden ser varias
    - Suspendidas (*stopped*): pueden ser varias
      - Puede ser la de primer plano (^Z)
      - Pueden suspenderse las de segundo plano (`kill -STOP`)
  - Controlar el estado de las tareas
    - Sistema de señales: permite controlar los cambios de estado
      - SIGCHLD: notifica al *shell* si un hijo se suspende, continúa o termina
        - Instalar manejador (`signal(SIGCHLD,manejador)`)
    - Comandos internos:
      - `fg` y `bg`: permiten cambiar de plano las tareas
      - `jobs`: permite listar las tareas en segundo plano

DIAGRAMA DE CONTROL DE TAREAS



TERMINAL Y SEÑALES

- Señales generadas por el *TTY driver*:
  - Provocadas tras el *parsing* del *Line discipline*:
    - SIGINT: carácter INTR (^C). *Interrupt* desde terminal
    - SIGQUIT: carácter QUIT (^\). Como ^C pero con *core dump*
    - SIGTSTP: carácter STOP (^Z). Suspender desde terminal
  - Provocadas por procesos:
    - SIGTTIN: si un proceso de un *job* en segundo plano intenta leer del TTY, el TTY driver le manda esta señal a todo el *job*
      - Acción por defecto: suspensión (*stopped*)
    - SIGTTOU: si un proceso de un *job* en segundo plano intenta escribir en el TTY, el TTY driver manda esta señal a todo el *job*
      - Acción por defecto: suspensión (*stopped*)
      - Se puede desactivar (`stty -tostop`)

LLAMADAS AL SISTEMA

- A usar por el *shell* (se proporcionan *wrappers* para un uso más sencillo)
  - `setpgid(pid, pgid):`

```
#define new_process_group(pid)  setpgid (pid, pid)
```

    - Asigna un id de grupo (pgid) a un proceso
    - Se usa su propio pid para un nuevo pgid
    - Uso: siempre que creemos una tarea nueva
  - `tcsetpgrp(fd, pgid):`

```
#define set_terminal(pid)  tcsetpgrp(STDIN_FILENO,pid)
```

    - Asigna el terminal a un id de grupo
    - El terminal se identifica como un file descriptor
    - Uso:
      - Siempre que pasemos una tarea a *fg*
      - Siempre que una tarea *fg* termine o se suspenda

LLAMADAS AL SISTEMA

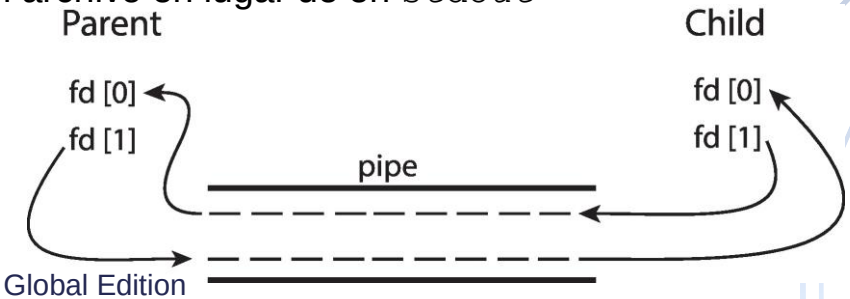
- A usar por el *shell* (se proporcionan *wrappers* para un uso más sencillo)
  - `sigprocmask(how,set,oldset)`:

```
#define block_SIGCHLD()      block_signal(SIGCHLD, 1)
#define unblock_SIGCHLD()    block_signal(SIGCHLD, 0)
void block_signal(int signal, int block)
{
    sigset_t block_sigchld;
    sigemptyset(&block_sigchld);
    sigaddset(&block_sigchld,signal);
    if(block) {
        sigprocmask(SIG_BLOCK, &block_sigchld, NULL);
    } else {
        sigprocmask(SIG_UNBLOCK, &block_sigchld, NULL);
    }
}
```

    - Enmascara señales
    - Uso: para proteger la modificación de la lista de *jobs*

LLAMADAS AL SISTEMA

- A usar por el *shell* (para las ampliaciones del *shell* básico)
  - `fileno(FILE *stream);`
    - Devuelve el número de descriptor de fichero correspondiente al *stream*
    - Uso: para pasar los parámetros de `dup2` y `pipe`
  - `dup2(int oldfd, int newfd);`
    - Hace que la entrada `newfd` de la tabla de ficheros del proceso apunte al fichero `oldfd`
    - Uso: para implementar la redirección y el pipe
      - Ej. `ls -la > listado.txt` Si `newfd` es `fileno(stdout)` y `oldfd` apunta a `listado.txt`, tras hacer `dup2` podemos hacer `fork` y `exec` de `ls` y el hijo heredará la tabla de descriptores de fichero vertiendo el resultado en el archivo en lugar de en `stdout`
  - `pipe(int pipefd[2]);`



Fuente: Silberschatz, Global Edition