Master en Computación

Programación Concurrente

Bloque II: Programación concurrente en POSIX

Tema 1. Introducción al estándar POSIX

1.1 Estándar POSIX

***P**ortable **O**perating **S**ystem **I**nterface*

- Basado en el sistema operativo UNI**X**

Estándar desarrollado conjuntamente por la *Computer Society* del IEEE y *The Open Group*

- también denominado *The Single UNIX Specification*
- la denominación oficial es IEEE Std. 1003, e ISO/IEC-9945

Accesible en Internet (sólo hace falta registrarse):

`http://www.unix.org/online.html`

También puede obtenerse información sobre las distintas funciones POSIX en las páginas de manual ("man pages") de Linux

Objetivos del POSIX

"*This standard defines a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level*"

(Introducción del standard POSIX)

El estándar define:

- **la interfaz del sistema operativo: conjunto de funciones, tipos y constantes (en lenguaje C) agrupadas en ficheros de cabeceras**
- **intérprete de comandos: redirección, *pipes*, etc.**
- **programas de utilidad: `vi`, `c99`, `ls`, `more`, etc.**

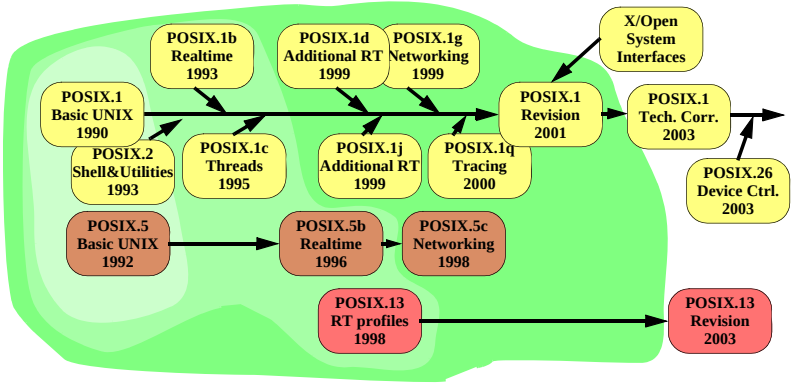
Pretende la portabilidad:

- **de las aplicaciones a nivel de código fuente**
- **de los programadores**

Categorías de estándares POSIX

- **Estándares Base: interfaz, *shell* y utilidades**
  - ***POSIX.1***
- **Perfiles o subconjuntos para diferentes entornos de aplicación**
  - **POSIX.13: Tiempo real**
- **Interfaces en diferentes lenguajes de programación ("*bindings*")**
  - **POSIX.5: *Bindings* en Ada**
  - **POSIX.9: *Bindings* en Fortran 77**

Evolución de los estándares POSIX



1.2 POSIX de tiempo real

Motivación:

- **Gran diversidad de sistemas de tiempo real:**
  - **Kernels de tiempo real (VRTX, VxWorks, etc.)**
  - **Ejecutivos Ada**
  - **Para sistemas grandes: VMS, OS9, sistemas privados**
  - **UNIX de tiempo real**
- **Era necesario un estándar para conseguir la portabilidad**
- **Son necesarios subconjuntos de los servicios del OS:**
  - **sistemas empotrados pequeños**
  - **controladores industriales de tiempo real**
  - **sistemas empotrados grandes**
  - **sistemas convencionales grandes de tiempo real**

Grupo de trabajo de tiempo real

***Objetivo*: "Desarrollar estándares que sean la mínima extensión sintáctica y semántica a los estándares POSIX para soportar la portabilidad de aplicaciones con requerimientos de tiempo real."**

1.3 Perfiles de entornos de aplicación

Los perfiles definen un subconjunto de los servicios POSIX que es obligatorio para un determinado ámbito de aplicación. Se definen 4 perfiles:

Sistema de Tiempo Real Mínimo

- **sistema empotrado pequeño, sin MMU, sin disco, sin terminal**
- **modelo: el "tostador"**

Controlador de Tiempo Real

- **controlador de propósito especial, sin MMU, pero con un disco con un sistema de ficheros simplificado**
- **modelo: robot industrial**

Perfiles de entornos de aplicación (cont.)

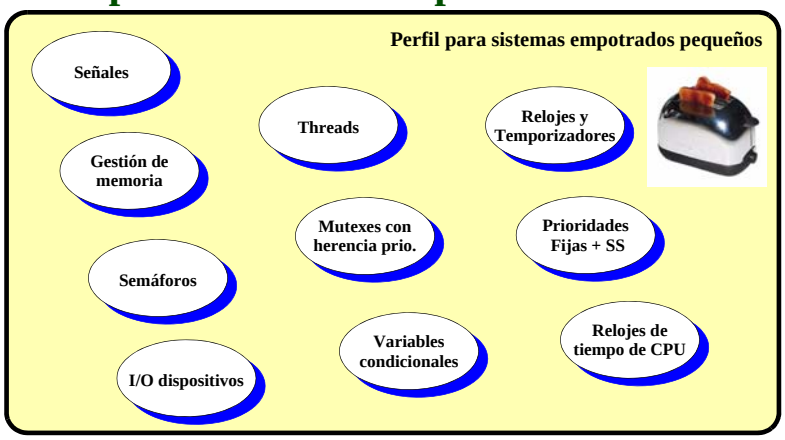
Sistema de Tiempo Real Dedicado

- **sistema empotrado grande sin disco, pero con MMU, quizás con un sistema de memoria secundaria en memoria flash**
- **el software es complejo y requiere protección de memoria y comunicaciones**
- **modelo: avión, célula de un sistema de telefonía móvil**

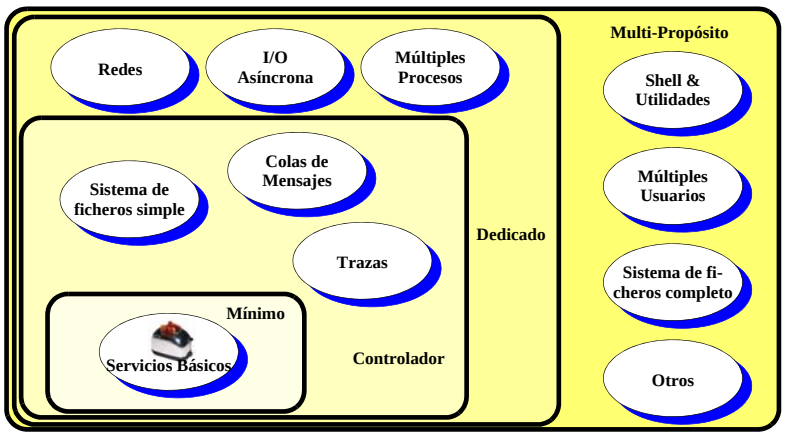
Sistema de Tiempo Real Multi-Propósito

- **sistema de tiempo real grande, con todas las facilidades**
- **modelo: sistema de control de tráfico aéreo, sistema de telemetría para un fórmula 1**

Principales servicios del perfil mínimo



Resumen de los perfiles



1.4 Generalidades sobre la interfaz POSIX

Conjunto de funciones, tipos y constantes (en lenguaje C)

- **constituyen la interfaz que el SO presenta a las aplicaciones**

Agrupadas en ficheros de cabeceras:

- **`<string.h>`: operaciones con strings (estándar C)**
- **`<time.h>`: tiempo, relojes, temporizadores**
- **`<pthread.h>`: todo lo relacionado con los threads**
- **`<signal.h>`: gestión de señales**
- **`<sched.h>`: planificación**
- **...**

Incluye parte de la librería estándar C

- **en ese caso el estándar POSIX se remite al estándar *ISO C 1999***

Identificadores POSIX

Siguen dos convenios diferentes:

- **identificadores heredados directamente de los sistemas UNIX originales o del lenguaje C**
  - **nombres cortos (a menudo una palabra) más o menos crípticos**
    - **ej.: `kill()`, `malloc()`, `time()`, `wctomb()`, `strcmp()`, ...**
- **identificadores introducidos durante el desarrollo del estándar**
  - **funciones: `servicio_acción()`**
    - **ej.: `timer_settime()`, `pthread_create()`**
  - **o también: `servicio_objeto_acción()`**
    - **ej.: `pthread_condattr_init()`, `pthread_key_delete()`**
  - **tipos de datos: finalizan en "`_t`". Ej.: `pthread_t`**
  - **constantes: en mayúsculas y comenzando por el nombre del servicio. Ej.: `TIMER_ABSTIME`, `PTHREAD_PRIO_INHERIT`**

Códigos de error

Las funciones POSIX informan de la ocurrencia de un error mediante un código numérico

El estándar define un conjunto de constantes que identifican los diferentes errores que pueden producirse (`<errno.h>`)

- **`EACCES`: permiso denegado**
- **`EAGAIN`: recurso no disponible, reintentar la operación**
- **`EFAULT`: dirección incorrecta**
- **`EINVAL`: argumento inválido**
- **...**

Detección de errores

Existen dos formas de que una función comunique que se ha producido un error

- **Las funciones "antiguas" (anteriores al estándar de threads) retornan el valor -1 cuando se produce un error**
  - **el código de error se puede obtener consultando la variable global `errno`**
- **Las funciones "modernas" retornan 0 cuando no ha habido error**
  - **y retornan el código numérico correspondiente al error en el caso de que lo haya habido**

Código de detección de errores

Es fundamental comprobar que no se ha producido un error en cada llamada a una función POSIX

- **de otra manera el error pasaría inicialmente inadvertido**
- **pudiendo producir un error posterior de muy difícil diagnosis**

El problema es que el chequeo de errores dificulta la comprensión del código:

```
if (pthread_create(&th1, NULL, body, NULL) != 0) {
  printf("error de creación del thread\n");
  exit(1);
}
if (timer_create(CLOCK_REALTIME, &event, &timer_id) == -1) {
  perror("error de creación del timer\n");
  exit(1);
}
```


Para mejorar la legibilidad del código utilizaremos las macros definidas en `misc/error_checks.h` (MaRTE OS)

- **`CHK`: para funciones que retornan 0 o el código de error**

```
#define CHK(p) { int ret;                                          \
                 if ((ret = p)) {                                  \
                   printe ("Error:"#p":%s\n", strerror(ret));      \
                   exit (-1);                                      \
                 }                                                 \
               }
```

- **`CHKE`: para funciones que retornan -1 en caso de error**

```
#define CHKE(p) { if ((p)==-1) {                                   \
                    perror (#p);                                   \
                    exit (-1);                                     \
                  }                                                \
                }
```

- **Además existen las macros `CHK_INFO` y `CHKE_INFO`**
  - **iguales que las anteriores pero que sólo informan del error, sin finalizar la aplicación**

Ejemplo del uso de las macros CHK y CHKE:

```
if (pthread_create(&th1, NULL, body, NULL) != 0) {
  printf("error de creación del thread\n");
  exit(1);
}
if (timer_create(CLOCK_REALTIME, &event, &timer_id) == -1) {
  perror("error de creación del timer\n");
  exit(1);
}
```

usando las macros es mucho más sencillo y fácil de leer:

```
CHK( pthread_create(&th1, NULL, body, NULL) );
CHKE( timer_create(CLOCK_REALTIME, &event, &timer_id) );
```