SISTEMAS OPERATIVOS II

TEMA 2

Conceptos de Sistemas Operativos: El caso Unix

Área de Arquitectura y Tecnología de Computadores

Escuela Universitaria Politécnica de Teruel

http://????.????.es/SOII/

Conceptos de S. O.: El caso Unix

- **Bibliografía:**
  - Silberschatz & Galvin: Sistemas Operativos
  - Stallings: Sistemas Operativos
  - Robbins: Unix programación práctica
    - Capítulos 1, 2 y 3
  - Ayuda de Unix: “man”

Conceptos de S. O.: El caso Unix

- **Objetivos**:
  - Recordar y afianzar algunos conceptos clave de los S. O. relacionados con procesos, planificación de la CPU, ficheros y gestión de memoria

Conceptos de S. O.: El caso Unix

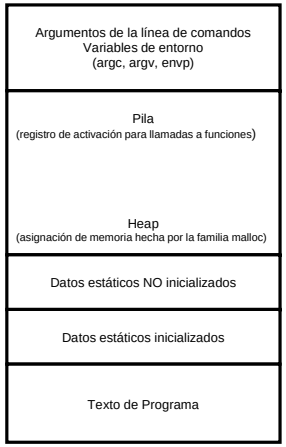
- Parte 1: **Procesos**
- Parte 2: **Planificación de la CPU**
- Parte 3: **Ficheros**
- Parte 4: **Gestión de memoria**

Conceptos de S. O.: El caso Unix

- Parte 1: **Procesos**
  - Estados de un proceso
  - El PCB
  - Modelado y operaciones
  - Llamadas al sistema
  - Procesos especiales
  - Procesos y threads

Procesos: Generalidades

- Estructura de un programa en la memoria principal



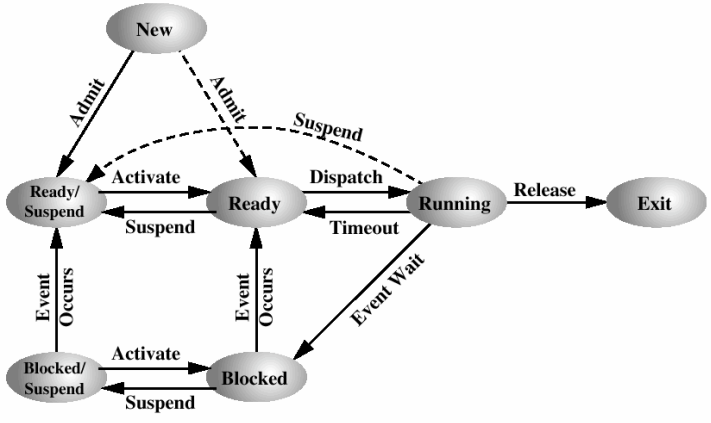
Procesos: Generalidades

- **El concepto de proceso**
  - Un proceso es un programa en ejecución
  - Un proceso es una instancia a un programa en ejecución

Un proceso es la entidad básica activa de un S. O.

Estados de un proceso

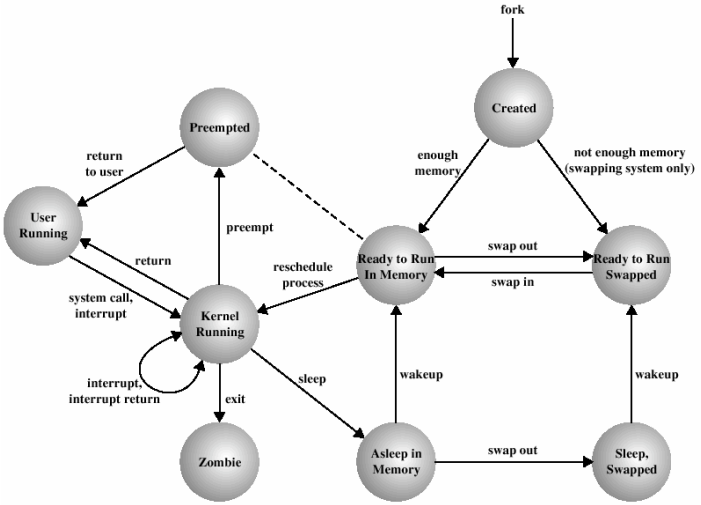
- **Modelo de proceso con dos estados “Suspendido”**



Unix Process States

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

Estados de un proceso en Unix SVR4



Procesos: El PCB

- **El PCB: Bloque de Control de Proceos**
  "Process Control Block"
  - Identificación del proceso
  - Información de estado del procesador
  - Información de control del proceso

Procesos: El PCB

- **El PCB:** Identificación del proceso

  Se suele utilizar números para identificar los procesos en el sistema
  - Identificador del proceso
  - Identificador del proceso padre
  - Identificador del usuario

Procesos: El PCB

- **El PCB:** Información de estado del procesador
  - Contenido de los registros del procesador
    - Registros visibles al usuario
    - Registros de control y estado
      - PC: Progam Counter
      - Códigos de condición
        - Signo, acarreo, igualdad, overflow, etc.
      - PSW: Program Status Word (“La palabra de estado”)
        - Contiene información de estado
    - Punteros de pila

Procesos: El PCB

- **El PCB:** Información de control del proceso
  - Información de estado y planificación
    - Estado del proceso
    - Suceso esperado
    - Información de planificación
      - Prioridad
      - Punteros a las colas de planificación
  - Estructuración de los datos
  - Comunicación entre procesos
  - Privilegios del proceso

Procesos: El PCB

- **El PCB:** Información de control del proceso
  - Gestión de memoria
    - Registro base a la zona de memoria
    - Límite de la zona de memoria
    - Tablas de páginas o segmentos
  - Propiedades de los recursos y su utilización
    - Dispositivos de E/S asignados
    - Tablas de ficheros abiertos
  - Información contable
    - Tiempo en ejecución
    - Memoria ocupada

Procesos: Modelado del sistema

- **Sistemas multiprogramado:**
  - Cada proceso tiene su PCB
  - El sistema debe guardar y gestionar una tabla de PCBs
    - En Unix esta tabla es una matriz de tamaño fijo
  - Todos los procesos progresan en un intervalo de tiempo suficientemente largo
  - En cada instante sólo se ejecuta un proceso
  - La velocidad de ejecución de un proceso no es uniforme y no es reproducible
    - Los procesos deben ser programados sin suposiciones acerca de su comportamiento temporal

Operaciones con procesos

- **Creación y Finalización de procesos:**
  - Los procesos se pueden ejecutar concurrentemente, y deben ser creados y destruidos dinámicamente
  - El S. O. debe proporcionar mecanismos para la creación de procesos:
    - Creación de un proceso de la nada
    - Creación de un proceso por clonación
  - El S. O. debe proporcionar mecanismos para la finalización de los procesos

Operaciones con procesos

- **Creación de procesos:**
  - Creación de un proceso de la nada:
    - Cargar el código y los datos en memoria
    - Crear una pila de llamadas (vacía)
    - Crear e inicializar un PCB
    - Colocar el proceso en la cola “ready”
  - Creación de un proceso por clonación:
    - Detener el proceso en ejecución y guardar su estado
    - Hacer una copia del código, datos, pila y PCB
    - Añadir el nuevo PCB a la cola “ready”

Operaciones con procesos

- **Creación de procesos:**
  - Recursos:
    - Padre e hijos comparten todos los recursos
    - El hijo comparte un subconjunto de los del padre
    - Padre e hijo no comparten recursos
  - Ejecución:
    - Padre e hijos se ejecutan concurrentemente
    - El padre espera a que los hijos terminen

Operaciones con procesos

- **Creación de procesos**
  - Espacio de direcciones:
    - El hijo es un duplicado del padre (Clonación)
    - El hijo tiene un programa cargado en él (Construcción de un proceso nuevo de la nada)
  - Caso Unix: **fork( )** y **exec( )**
    - **fork( )** crea un nuevo proceso por clonación
    - **exec( )** reemplaza el espacio de memoria de un proceso por un programa o fichero ejecutable

Operaciones con procesos

- **Finalización de procesos**
  - El S. O. debe proporcionar mecanismos para la finalización de los procesos
  - Proceso de finalización:
    - Se ejecuta la última instrucción
    - Se solicita al S. O. la liberación de todos los recursos asignados
    - El proceso puede devolver algún dato a su proceso padre
    - El S. O. libera los recursos del proceso
  - Caso Unix: **exit( )**

Procesos: Llamadas al sistema

- **ID de un proceso**
  - Unix identifica los procesos mediante un entero único denominado ID de proceso
  - getpid, getppid - get process identification

```
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

POSIX.1, Spec 1170

  - Unix asocia a cada proceso con un usuario particular conocido como propietario del proceso
  - getuid, geteuid - get user identity

```
#include <unistd.h>
#include <sys/types.h>
uid_t getuid(void);
uid_t geteuid(void);
```

POSIX.1, Spec 1170

Procesos: Llamadas al sistema

- **Creación de procesos: el fork de Unix**
  - Unix crea los procesos a través de la llamada al sistema fork; copiando la imagen en memoria que tiene el proceso padre
  - Los dos procesos continúan su ejecución en la instrucción que está después del fork
  - fork, vfork - create a child process

```
#include <unistd.h>
pid_t fork(void);
pid_t vfork(void);
```

POSIX.1, Spec 1170

    - Devuelve 0 al proceso hijo
    - Devuelve el ID del hijo al proceso padre

Procesos: Llamadas al sistema

- **Creación de procesos: el fork de Unix**
  - El proceso hijo hereda:
    - Las variables de entorno
    - El contexto
    - Disposición ante las señales
    - Parámetros de planificación
    - Tabla de descriptores de ficheros
      - El padre y los hijos comparten el cursor de todos los ficheros que fueron abiertos por el padre antes del fork

Procesos: Llamadas al sistema

- **Llamadas al sistema wait y waitpid**
  - Si un padre desea esperar hasta que el hijo termine, entonces debe ejecutar una llamada al sistema: wait o waitpid
  - wait, waitpid - wait for process termination

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
```

POSIX.1, Spec 1170

  - Devuelve –1 en caso de error y pone un valor en errno.
    - ECHILD: no existen hijos a los que esperar
    - EINTR: la llamada wait fue interrumpida por una señal
  - **¿Qué sucede con un proceso que ha finalizado y su padre no ha esperado su finalización?**

Procesos: Llamadas al sistema

- **Llamadas al sistema wait y waitpid**
  - POSIX especifica los siguientes macros para analizar el estado devuelto por el hijo:
    - WIFEXITED WEXITSTATUS
    - WIFSIGNALED WTERMSIG
    - WIFSTOPPED WSTOPSIG
    - Ejemplo:- if(WIFEXITED(status))
if(WIFSIGNALED(status))
  - **waitpid**
    - Proporciona un método más flexible para esperar la finalización de los hijos.
    - Tiene una forma que no bloquea (opción WNOHAN)
  - **Ejercicio**: Probar la relación entre wait, waitpid y exit

Procesos: Llamadas al sistema

- **Llamadas al sistema exec**
  - La llamada fork crea una copia del proceso que la ejecuta
  - La familia exec proporciona un método que permite sustituir el proceso que realiza la llamada con un módulo ejecutable nuevo
  - execl, execlp, execle, execv, execvp, execve - execute a file

```
#include <unistd.h>
extern char **environ;
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int  execle( const char *path, const char *arg , ..., char * const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
int  execve  (const  char  *filename, char *const argv [], char *const envp[]);
```

POSIX.1, Spec 1170

Procesos: Llamadas al sistema

- **Llamadas al sistema exec**
  - execl: los argumentos se pasan en una lista
  - execv: los argumentos se pasan en un array
  - Atributos conservados después de realizar una llamada exec:

ID del proceso:	getpid();
ID del padre:	getppid();
ID del grupo:	getpgid();
Membresía de sesión:	getsid();
ID real del usuario:	getuid();
ID real del grupo:	getgid();
Directorio de trabajo en uso:	getcwd();
Tiempo que resta en la señal alarma:	alarm();
Máscara del modo de creación del fichero:	umask();
Máscara de señal del proceso:	sigprocmask();
Señales pendientes:	sigpending();
Tiempo utilizado hasta el momento:	times();

Procesos: Llamadas al sistema

- **Finalización de procesos en Unix**
  - Cuando finaliza un proceso, el S. O. recupera los recursos asignados al proceso
  - Tareas realizadas en la finalización de un proceso:
    - Cancelar temporizadores y señales pendientes
    - Liberar la memoria y otros recursos asignados
    - Cerrar los ficheros abiertos
  - El S. O. notifica al padre, si ejecuta wait(-), la “muerte” de sus hijos
  - Cuando un proceso termina, sus hijos huérfanos son adoptados por el proceso init (ID ≅ 1)
  - Un proceso se convierte en “zombie” si su padre no espera su finalización
    - El proceso init espera a los hijos huérfanos para liberarse de los “zombies”

Procesos: Llamadas al sistema

- **Finalización de procesos en Unix**
  - _exit - terminate the current process

```
#include <unistd.h>
void _exit(int status);
```

POSIX.1, Spec 1170

  - exit - cause normal program termination

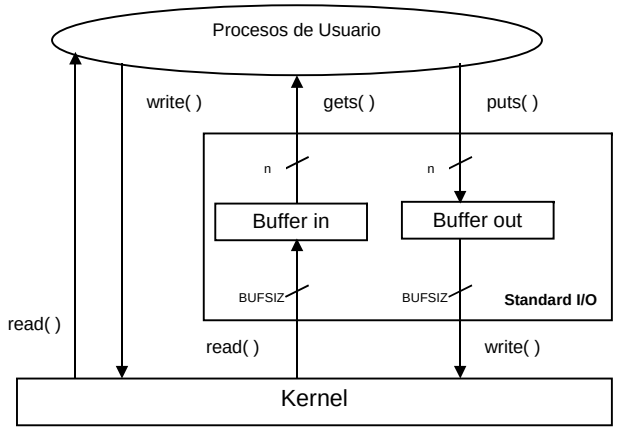
```
#include <stdlib.h>
void exit(int status);
```

POSIX.1, Spec 1170

  - Si un proceso termina anormalmente, tal vez se produzca un vaciado de la memoria (core dump), además no se llamará a los manejadores de finalización instalados por el usuario
  - !!! atexit !!!

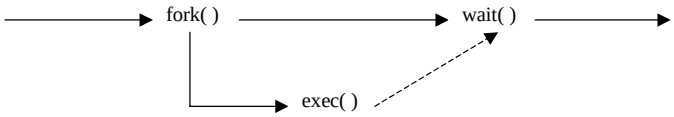
El sistema de ficheros Unix*

- Llamadas al sistema vs Funciones estándar de E/S



Procesos especiales

- **El intérprete de comandos: “el shell”**



  - Notas:
    - Con el carácter de interrupción “Ctr + c”, se puede finalizar cualquier comando ejecutado desde el shell
    - Los shell interpretan una línea que termina con un “&”, como un comando que debe ser ejecutado por un proceso asíncrono (en plano secundario)

Procesos especiales

- **Los procesos demonios: “los deamons”**
  - Un demonio es un proceso en plano secundario que normalmente se ejecuta por tiempo indefinido
    - Ejemplos:
      - Correo electrónico
      - FTP
      - Solicitudes de impresión
      - Estadísticas, etc.
  - En Unix hay muchos procesos demonios que realizan tareas rutinarias
    - **pagout,** maneja la paginación para la administración de la memoria
    - **in.rlogin,** maneja las solicitudes remotas de acceso al sistema

Procesos y threads

- **Threads**

  http://es.wikipedia.org/wiki/Hilo_de_ejecuci%C3%B3n
  - Un proceso está definido por los recursos que usa y por el estado de su ejecución:
    - podría ser útil que el acceso a los recursos compartidos fuese de manera concurrente
    - pensar en fork( ) ejecutándose en el mismo espacio de direcciones con un nuevo PC
  - **Def.-** Un **thread** (o proceso ligero) es la unidad básica de utilización de la CPU
  - **Def.-** Una **tarea** es un conjunto de threads que comparte código y datos

Procesos y threads

- **Partes de un thread**
  - Parte **privada**:
    - Contador de programa: PC
    - Conjunto de registros
    - Espacio de pila
  - Parte **compartida**:
    - La sección de código
    - La sección de datos
    - Recursos del S. O.

Procesos y threads

- **Threads**
  - ¿Los S. O. soporta el concepto de thread?
    - Si
  - Mecanismos de los S. O. para facilitar el uso de threads:
    - A nivel de núcleo (kernel)
      - La planificación puede NO ser equitativa
    - A nivel de usuario (librerías)
      - No interviene el kernel
      - Los cambios de contexto son más rápidos

Procesos y threads

- Un proceso tradicional (o proceso pesado) es igual a una tarea con un único thread
- **Ventajas de los threads frente a los procesos**
  - La creación de un thread es menos costosa
  - La gestión de memoria es más sencilla
  - Los cambios de contexto entre threads son menos costosos ya que comparte código y datos
    - Sólo se cambian los valores de los registros del thread

Procesos y threads

- **Parecidos entre procesos y threads**:
  - Estados (los mismos)
  - Los threads comparten la CPU (sólo uno activo)
  - Un thread se ejecuta secuencialmente
  - Puede crear threads hijos
  - Pueden bloquearse en llamadas al sistema del tipo wait( )

Procesos y threads

- **Diferencias entre proceso y threads**:
  - Los threads no son independientes, trabajan sobre los mismos recursos, espacios de direcciones, etc.
  - No existe protección entre threads
  - En una tarea con varios threads, si uno está bloqueado, se puede ejecutar otro thread de la misma tarea
  - La cooperación de varios threads en un trabajo confiere una mayor velocidad y mejora las prestaciones
  - Las aplicaciones que requieren un buffer compartido pueden beneficiarse de la utilización de threads

Threads en Solaris

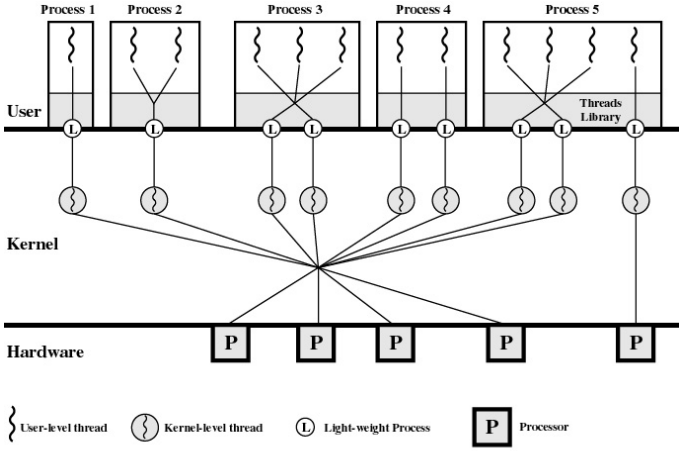


Figure 4.15 Solaris Multithreaded Architecture Example