Apuntes básicos de la asignatura:

SISTEMAS OPERATIVOS DISTRIBUIDOS

Tema 1.− Introducción

[Conceptos básicos de Sistemas Operativos] [Evolución Histórica de los SSOO] [Introducción a los SSDD] [Tendencias]

Capítulo 1.1.− Conceptos básicos de Sistemas Operativos

Qué es un SO?

Un sistema operativo es un programa que actua como intermediario entre el usuario de un ordenador y el hardware de esa máquina; por lo tanto, su misión principal es la de proporcionar un entorno donde los programas se puedan ejecutar de forma conveniente y eficiente.

Características de los Sistemas Operativos

- Sistemas Grandes
- Complejos:
- Asíncronos.
- Diferentes necesidades de los usuarios.
- HW muy diverso.
- Performance importante.
- Poco conocidos:
- Poco predecibles.
- Depuración complicada (bugs).
- Ciclo de vida largo.

Componentes (funciones)

- Gestor de procesos
- creación y borrado de procesos de sistema y usuario.
- suspensión y reanudación de programas.
- provisión de mecanismos para sincronizar.
- provisión de mecanismos para comunicar procesos.
- provisión de mecanismos para tratar bloqueos.
- Gestor de memoria principal
- cuenta de páginas de memoria en uso y usuario.
- decisión de procesos a cargar en memoria, con espacio libre.
- asignación de memoria dinámica, según necesidades.
- Gestor de memoria secundaria
- gestión de espacio libre.
- planificación de operaciones sobre el disco (concurrencia).
- gestión del espacio ocupado (contabilidad del mismo).
- Gestor del sistema de Entrada/Salida
- gestión del sistema de buffers.

- interfaz general con los dispositivos.
- control de los dispositivos.
- Gestor de ficheros:
- creación y borrado de ficheros.
- creación y borrado de directorios.
- manejo de ficheros y directorios.
- correspondencia de ficheros y directorios con espacio en memoria secundaria.
- volcados de ficheros.
- Mecanismos de protección.
- Mecanismos de comunicación en red.
- Intérprete de comandos.

Servicios de un Sistema Operativo

- De ayuda al usuario para la ejecución de programas.
- ejecución de programas (entorno).
- operaciones de Entrada/Salida.
- manipulación de ficheros.
- comunicaciones (memoria compartida, paso de mensajes).
- Para asignar y asegurar uso eficiente del Hardware.
- asignación de recursos.
- funciones de contabilidad (accounting).
- protección y seguridad.

Llamadas al Sistema (System Calls)

Son intefaces entre los procesos y el sistema operativo. Casi todos los lenguajes de alto nivel poseen una interfaz con UNX (standard POSIX: Portable Operating System Interface on Unix, C, Fortran, Ada, ...).

Ejemplos:

- Gestión de procesos: end, abort, load, execute, create process, terminate process, get process attribute, ...
- Manipulación de procesos: create file, delete file, open file, close file, write file, delete file...

Programas del Sistema

Constituyen un entorno de desarrollo y ejecución de programas que resulta cómodo al usuario. Permite facilidades de compilación, edición, accounting, configuración... que forma el interfaz gráfico (SHELL).

Estructura de un SO

- Estructura sin niveles (MS−DOS).
- Estructura de niveles (UNIX):
- Nivel 0: HW.
- Nivel 1: Microkernel.
- Nivel 2: Solo ve las funciones del nivel 1, y asi....
- ...
- NIvel n: programas de usuario.

Capítulo 1.2.− Evolución histórica de los Sistemas Operativos

Fase I

- Hardware caro, personas "baratas".
- Objetivo: optimizar el uso del HW.
- **Primeros sistemas**
- NO EXISTE SISTEMA OPERATIVO.
- Tras pequeñas mejoras: CARGAR PROGRAMA − CARGAR COMPILADOR − COMPILAR − CARGAR ENSAMBLADOR − ENSAMBLAR − CARGAR LINKADOR − LINKAR − EJECUTAR − DEPURAR. **DEVICE DRIVERS.**
- **Procesamiento por lotes**
- no existe depuración interactiva.
- existe operador.
- secuenciamiento automático de trabajos => comandos de control, tipo JCL (job control language).
- mapa de memoria: MONITOR(CARGADOR, SECUENCIADOR DE PROGRAMAS, INTERPRETE DE DIRECTIVAS DE JCL), PROGRAMAS DE USUARIO => SO = Monitor + Device Drivers.
- **Monitor que solapa la Entrada/Salida con el cálculo (comienzo de multiproceso)**
- Los periféricos son más lentos que la CPU.
- Se pasa de LECTORA DE TARJETAS − CPU − IMPRESORA a LECTORA DE TARJETAS − UNIDAD DE CINTA − CPU − UNIDAD DE CINTA − IMPRESORA. **SPOOL** (simultaneous peripheral operation on−line).
- **Multiprogramación**
- Mecanismos de protección de memoria y de reubicación de programas.
- Hay trabajos multiples => gestión de procesos.
- Hay concurrencia.

Fase II

- Hardware barato, tiempo humano caro.
- Objetivo: optimizar el tiempo de los usuarios.
- **Tiempo compartido**
- Se utiliza multiprogramación.
- Respuesta interactiva.
- Políticas de planificación.
- Sistemas de ficheros.
- Memoria virtual (swapping).
- Sincronización y coordinación de trabajos.
- Actualmente:
- **Sistemas Paralelos (fuertemente acoplados):** más de una CPU, multiprocesadores que comparten reloj, buses, y algunas veces memoria y periféricos.
- Procesamiento simétrico.
- Procesamiento asimétrico.
- Razones de inclusión de CPUs:
- una tarea por CPU.
- tolerancia a fallos.
- abaratamiento de sistemas.
- Performance:
- Tiempo de respuesta.
- Throughput = número de operaciones/unidad de tiempo.
- **Sistemas Distribuidos (debilmente acoplados)**: diferentes máquinas conectadas en red.
- CPUs comunicadas por medio de red.
- los procesos no han de ser iguales.
- compartición de recursos a distancia, ventaja añadida a los sistemas en paralelo.

- **Sistemas en Tiempo Real**
- Determinismo: tiempo de respuesta garantizado.
- Tipos:
- sistemas duros: cumplimiento rígido.
- sistemas blandos: restricciones flexibles.

Capítulo 1.3.− Introducción: Sistemas Distribuidos

Definición:

Un sistema distribuido es una colección de ordenadores independientes que aparecen ante el usuario como si se tratara de uno solo. Esto implica que:

- las máquinas pueden ser autónomas.
- los usuarios piensan en la aplicación Software como si estuviera integrada en su máquina.

Ejemplos:
1.− Red de estaciones de trabajo en un departamento universitario de ingeniería informática, con pool de procesadores

2.− Compañía bancaria con cientos de sucursales por todo el mundo.

3.− Correo electrónico. Tiene el problema de "name resolution": DNS.

Un sistema distribuido simple.

- Basado en una red local.
- Cumple los mismos requisitos que un sistema centralizado.
- UNIX
- UNIX 4BSD de Berkely, a finales de los 70. La arquitectura utilizada fue la mostrada aquí.
- La implementación **UNIX *COMERCIAL*** (no Linux) más utilizada es la de Sun Microsystems, basado en el BSD. Este sistema fue el que comenzó a utilizar el NFS (Network File System) para servicio de ficheros, el RPC (Remote Procedure Calling), y el NIS (Network Information Service).
- Otras han intentado evitar los problemas de UNIX, de diferentes maneras:

- Amoeba, Mach, Chorus como SSOO.
- Extensiones de UNIX: AFS (Andrew File System), Kerberos.

- Ejemplo: cajero automático.
- Ejemplo: sistema Pandora.

Importante: un sistema se ha de considerar distribuido si es capaz de actuar como si fuera un sistema uniprocesador, sin conocimiento directo del usuario, y con transparencia.

Ventajas de Sistemas Distribuidos con respecto a Sistemas Centralizados:

1.− Económico

2.− Prestaciones en modo absoluto

3.− Algunas aplicaciones HAN de ser distribuidas

4.− Trabajo cooperativo.

5.− Mayor confianza del sistema.

6.− Escalabilidad.

Ventajas de Sistemas Distribuidos con respecto a Sistemas Aislados:

1.− Compartición de datos.

2.− Compartición de dispositivos.

3.− Comunicación entre usuarios.

4.− Flexibilidad.

Desventajas de los Sistemas Distribuidos:

1.− Diseño de aplicaciones Software.

2.− Red de comunicación.

3.− Seguridad.

Capítulo 1.4.− Tendencias

Objetos Distribuídos:

- OMG CORBA.
- Microsoft DCOM.

Remote Method Invocation:

- Java RMI.

Agentes móviles:

- Voyager con CORBA.

Tema 2.− Sistemas de Multiprocesado

[Threads y Procesos] [Gestión y Planificación de Tareas]

Capítulo 2.1.− Threads y Procesos

Concepto de proceso

- Definición:
- En procesado por lotes: jobs (trabajos).

- En sistemas de tiempo compartido: tasks (tareas).
- En general: actividades.
- Un proceso es un programa en ejecución, un flujo de ejecución en un ESTADO particular del proceso (estado: todo aquello que puede afectar o ser afectado por el proceso).
- No existe concurrencia dentro del proceso.
- Estados de ejecución

- **PCB** (process control block): representación que un SO tiene de un proceso. Contiene muchas piezas de información asociadas con un determinado proceso:
- Estado del proceso.
- Contador de programa: dirección de la próxima instrucción a ejecutarse.
- Registros de CPU: acumuladores, registros de índice, punteros a pila, registros de propósito general...
- Información de scheduling de CPU: prioridad de proceso, punteros a colas de scheduling...
- Información de gestión de memoria: registros base y límite...
- Información de contabilidad.
- Información de estado E/S: lista de dispositivos.

Ejecución concurrente de procesos

- **Dispatcher**: parte más interna del SO encargada de lanzar la ejecución de los procesos.
- **Scheduler**: asigna prioridades a procesos y selecciona procesos ejecutables para entrar en las colas de procesos
- **Cambios de contexto:** mecanismo del dispatcher para salvar y recuperar el contexto (estado) de un proceso cuando se produce una conmutación de procesos en la CPU.
- contador de programa.
- contenidos de registros de estado.
- contenidos de registros de propósito general.
- contenidos de registro de punto flotante.
- ¿memoria?. Es mucha información =>
- no se salva la memoria (PC. Mac).
- optmiización del mecanismo de transferencia de memoria a disco.
- protección y gestión de memoria (UNIX).

Creación y terminación de procesos

- La relación es PADRE−HIJO.
- Posibilidades de ejecución PADRE−HIJO:
- ejecución concurrente P−H.
- el padre espera a que los hijos hayan terminado.
- Compartición del espacio de direcciones entre P−H:
- el hijo es un duplicado del padre: fork().
- el hijo cargaen memoria un nuevo programa: exec().
- fork():
- se para y salva el proceso actual del padre.
- hace una copia del código, datos, pila y PCB.
- se añade un nuevo PCB a la lista de procesos preparados.
- exec():
- se para y salva el proceso actual.
- cargar código y datos en memoria.
- crear una pila vacía.
- crear e inicializar un PCB.
- poner el nuevo proceso en la cola de procesos preparados.

- exit(): autoterminación (término de ejecución de una forma normal):
- abort(): el padre aborta la ejecución de un hijo (siempre se puede).
- no tiene sentido prolongar la ejecución de un hijo.
- el hijo consume más recursos de los que se le pueden proporcionar.
- el padre quiere ejecutar un exit().

Concepto de Thread

Existen muchas ocasiones en las que resulta útil que los recursos fueran compartidos y accedidos de modo concurrente.

Son procesos, unidades lógicas de planificación en la CPU, que consisten en:

- contador de programa.
- conjunto de registros.
- espacio para la pila.

Los threads creados por un proceso comparten:

- sección de código.
- sección de datos.
- recursos del SO (ficheros abiertos, señales...)

Capítulo 2.2.− Gestión y Planificación de Tareas

[Multithreading] [Modelos de diseño] [Arquitectura Solaris 2.x]

Beneficios de Mutithreading

- Mejora la respuesta de las aplicaciones
- Utilización eficiente de multiprocesadores
- Mejora de la estructura del programa: módulos independientes.
- Utilización de menos recursos del sistema que mediante la utilización de procesos (cambio de contexto).
- Combinación de threads y RPC
- Mejora de prestaciones (performance):

TIEMPO DE CREACIÓN DE THREADS	**Microsegundos**	**Ratio**
Unbound threads	52	–
Bound threads.	350	6.7
fork()	1700	32.7

TIEMPO DE SINCRONIZACIÓN MEDIANTE SEMÁFOROS	**Microsegundos**	**Ratio**
Unbound threads.	66	–
Bound threads.	390	5.9
Entre procesos.	200	3

Problemas Potenciales

- Complejidad.

- Condiciones de carrera.
- Deadlocks.
- Inversión de prioridades
- Código no reentrante

Modelos de Implementación Multithread

- Modelo Boss−Worker
- Un thread funciona como jefe, y asigna tareas al resto para que las ejecuten.
- Cuando el thread empleado termina, puede mandar una notificación al jefe informándole del resultado final, y de que está preparado para otra tarea, o es el propio jefe el que realiza un "polling" sobre los empleados.
- Variación: el modelo de colas: el jefe pone sus trabajos en una cola accesible por el resto de los threads, y estos los van capturando y ejecutando. Ejemplo: pool de secretarias, donde el jefe va poniendo en una cesta los documentos que han de ser pasados a máquina.
- Modelo Work Crew
- Un conjunto de threads trabaja en paralelo sobre una tarea.
- La tarea se divide horizontalmente en piezas, y cada thread ejecuta una de ellas.
- Ejemplo: gente limpiando un edificio.

- Modelo Pipelining
- La tarea se divide VERTICALMENTE en pasos, que deben ser ejecutados secuencialmente.
- Cada thread ha de ejecutar un paso, y avisar al siguiente thread para que realice el siguiente.
- Ejemplo: fábrica de ensamblaje de automóviles.

Arquitectura Multithread Solaris 2.x: definiciones

- Thread: secuencia de instrucciones ejecutada en el contexto de un proceso.
- Single−threaded: restricción de acceso a un solo thread.
- Multithreaded: permiso de acceso simultaneo de dos o más threads.
- Threads de usuario (user−level threads): threads gestionados por bibliotecas de threads en el espacio de usuario (en contraposición a espacio de kernel).
- Procesos ligeros (lightweight process): threads en el kernel que ejecutan código kernel y llamadas al sistema (LWPs).
- Bound threads: threads enlazados permanentemente a LWPs.
- Unbound threads: threads que se enlanzan y desligan del pool del LWP.

Threads de Usuario

Los threads comparten las instrucciones de código, y la mayoría de los datos, así que un cambio por parte de un thread de los datos compartidos puede ser visto por el resto de los threads del proceso. Cuando un thread necesita comunicarse con otros threads del mismo proceso, el sistema operativo no tiene por qué estar involucrado.

Los threads de usuario se manejan en el espacio de usuario, para así evitar retrasos por cambios de contexto del kernel. Una aplicación así puede tener miles de threads sin consumir recursos kernel.

Características:

- Baratos de crear, pues son "pedazos" de memoria virtual en el espacio de usuario, en tiempo de ejecución.

- Rápidos de sincronizar, pues se realiza a nivel de aplicación, no de kernel.
- Fácilmente gestionables por la biblioteca de threads (libthread).

Generalmente, los threads son muy ligeros. Si es necesario ligarlos a recursos de ejecución, se convierten en recursos kernel.

Procesos ligeros

La biblioteca de threads utiliza threads inherentes de control: LWP, soportados por el kernel. Se pueden ver como CPU VIRTUAL que ejecuta código o llamadas al sistema.

Generan un puente entre el nivel de usuario y el nivel kernel.

Cada proceso contiene uno o más LWPs, cada uno de los cuales ejecuta uno o más threads de usuario. La creación de un thread implica, generalmente, la creación de algún contexto de usuario, pero no la creación de un LWP.

No existe una relación one−on−one entre threads de usuario y LWPs, pudiendo los threads de usuario migrar de un LWP a otro. Es la biblioteca de threads, con la ayuda del programador y del sistema operativo, quien se asegura de que el número de LWPs disponible sea el adecuado para los threads de usuario activos en ese momento.

Cuando un thread de usuario se bloquea (sincronización), su LWP se marcha a otro thread en ejecución.

El sistema operativo decide que LWP se ejecuta en qué procesador y cuándo. No ha de preocuparse de cuántos threads de usuario
existen ni nada por el estilo. Son los LWPs los que se planifican en los recursos CPU a partir de sus prioridades.

Unbound Threads

Los threads que se planifican dentro de un pool de LWP se denominan "threads ligados". Generalmente se desea esta situación para que puedan flotar libremente entre diferentes LWPs.
Los LWPs se asignan a threads en ejecución => el LWP asume sus instrucciones. Cuando el thread se bloquea, u otro tiene más prioridad, el estado se almacena y el LWP va a otro thread (todo gestionado por la biblioteca de threads).

Bound Threads

Para qué ligar permanentemente un thread a un LWP:

- Thread planificado globalmente.
- Se da al thread una pila alternativa.
- Se da al thread una alarma o timer único.

A veces, tener más threads que LWPs, caso que puede ocurrir con threads desligados, es una desventaja.

Clases de Planificación (scheduling classes)

El kernel Solaris tiene tres clases de planificación:

- RealTime (RT). Tiene la más alta prioridad.
- System: esta no puede ser aplicada a ningún proceso de usuario.
- Timeshare (TS): por defecto.

La clase de planificación la mantiene cada LWP por separado. Cuando un proceso es creado, el LWP inicial hereda la clase de planificación y la prioridad del proceso padre.

Cuando se crean nuevos LWPs para threads desligados, también heredan esa clase y prioridad. De hecho, todos los threads desligados de un mismo proceso tienen la misma clase de planificación y prioridad.

Los threads ligados tienen la clase de planificación y la prioridad de sus LWPs.

Planificación Timeshare:

Distribuye el recurso en modo Round−Robin, dependiendo de la prioridad.

La prioridad de despacho de los LWPs "time−shared" se calcula a partir del ratio de uso instantaneo de la CPU, y de la prioridad relativa de los procesos con respecto al planificador timeshare (a un LWP que haya recibido una gran cantidad de tiempo de procesamiento se le asigna una prioridad más baja).

Planificación Realtime:

Se puede aplicar a todo un proceso, o a un LWP o más dentro de un proceso, lo cual requiere privilegio de superusuario.

El planificador siempre despacha el LWP relatime con más alta prioridad. Este retiene control del procesador hasta que es "preempted", se suspende, o su prioridad realtime se cambia.

Los LWPs realtime tienen prioridad absoluta.

Planificación de LWP y Binding de threads:

La biblioteca de threads ajusta automáticamente el número de LWPs en el pool para ejecutar threads desligados. Así:

- se evita que el programa se bloquee por falta de LWPs no bloqueados.
- se consigue un uso eficiente de LWPs.

Cuando todos los LWPs en un proceso están bloqueados, el sistema operativo lanza una señal SIGWAITING al proceso, gestionada por la biblioteca de threads, de forma que se cree un nuevo LWP y se asigne a ejecución.

Al contrario, cuando existe un "overbooking" de LWPs, debido a que la biblioteca de threads asigna a cada LWP una edad, los no utilizados serán eliminados en un tiempo medio de cinco minutos.

Categorías funcionales en la implementación de Threads

- Rutinas generales de threads (fork, create, exit...)
- Rutinas de acceso a atributos (attr_destroy, attr_getstacksize...).
- Rutinas Mútex (mutexa_init, mutex_destroy...).
- Rutinas de Variables de condición (cond_init, cond_destroy).
- Rutinas de datos específicos de thread (getspecific, setspecific...).

- Rutinas de Cancelación de threads (cancel, setcancelstate...).
- Rutinas de Planificación y Prioridad de Threads (getschedparam, yield...).

POSIX

El grupo POSIX 1003.1c (anteriormente POSIX 1003.4a) está trabajando en un estándar para programación multithread.
No existen diferencias fundamentales entre los threads Solaris que aquí se han expuesto y el estándar POSIX (Portable Operating System Interface for UniX), también conocidos como pthreads.

Tema 3.− Sistemas Multiprocesadores

[Conceptos HW] [Conceptos SW] [Combinación HW−SW]

Capítulo 3.1.− Conceptos HW

- Existen diferentes formas de organización.
- Existen varias clasificaciones; una de las más citadas taxonomías: Flynn (1972):

- Dos características:
- Número de "streams" de instrucciones.
- Número de "streams" de datos.
- ***Clasificación:***
- **SISD (Single instruction, Single Data)**: ordenadores monoprocesador.
- **SIMD (Single instruction, Multiple Data):** array de procesadores con una unidad de instrucción.
- **MISD (Multiple Instruction, Single Data)**: No existen ordenadores así.
- **MIMD (Multiple Instruction, Multiple Data):** grupo de ordenadores independientes cada uno con su propio contador de programas, programa y datos. Todos los sistemas distribuidos son MIMD => la clasificación no es muy útil.

    - Nueva clasificación:
    - Conceptos:

- **Bus**: hay una red, bus, cable, etc, que conecta todas las máquinas (televisión por cable).
- **Switch**: no tienen un backbone, sino cables individuales de máquina a máquina. Los mensajes se mueven por los cables, con decisiones de "switching" .

- **Tightly coupled** (fuertemente acoplado): el retraso de mensajes entre un ordenador a otro es pequeño, y le ratio de datos es alto (circuito integrado con dos chips).
- **Loosely coupled** (débilmente acoplado).
- Los sistemas fuertemente acoplados suelen trabajar en la resolución de un sólo problema, mientras que los débiles resuelven problemas a veces independientes.

3.1.1.− Multiprocesadores "Bus−based"

- CPUs en un bus común más un módulo de memoria.
- El bus puede tener:

- 32/64 líneas de dirección.
- 32/64 líneas de datos.
- 32 líneas de control.

- **Coherencia en memoria:** si la CPU A escribe en dirección X y la CPU B lee X => obtiene el mismo valor.
- **Memoria Caché:** para evitar sobrecarga, la caché almacena las palabras accedidas más recientemente (**HIT RATE** máximo: la caché suele tener entre 64 K y 1 M).
- **Problema: s**e presenta un problema de incoherencia: A y B tienen X en su caché. A escribe en X, B lee de X en su caché: **valor antiguo**!!!
- Solución: **Write−through caché:** cuando se escribe en caché, se escribe en memoria: las "read misses" y los "writes" causan tráfico en la red.
- Además, todos los cachés monitorizan el bus para que, cuando exista un "write" sobre una dirección de memoria presente en la caché, o elimina la entrada del caché, o la actualiza: **SNOOPY CACHÉ.**
- SNOOPY + WRITE−THROUGH = COHERENCIA.

3.1.2.− Multiprocesadores "Switched"

- **Crossbar switch:**

- cada switch se puede abrir o cerrar.
- problema: si 2 CPUs intentan acceder a la misma memoria simultaneamente, una tiene que esperar.
- n CPUs, n memorias: n2 crosspoints => prohibitivo.

- **Omega switching network**

- 4 switches 2x2 (2 entradas, 2 salidas). Cada CPU puede acceder a cada memoria.
- El "switching setting" se realiza en tiempo de NANOSEGUNDOS.
- n CPUs, n memorias: log2n pasos de switching, cada una con n/2 switches, para un total de (nlog2n)/2 switches.
- **Problema:** retardo por tener que realizar tantos "stages" (pasos).

- **NUMA (Non Uniform Memory Access)**

- Se piensa en sistemas jerárquicos: cada CPU tiene asociada una memoria => acceso rápido.
- Si esa CPU necesita acceder a otra memoria, puede, aunque su acceso sea más lento.
- **Ventaja:** menores tiempos promedio de acceso que los sistemas Omega.
- **Inconveniente:** la localización de datos y programas es crítico.

- Conclusión: Los grandes sistemas multiprocesadores son posibles, pero caros.

3.1.3.− Multicomputadores "Bus−based"

- LAN (10 − 100 Mbps).
- Bus backplane: 300 Mbps.

3.1.4.− Multicomputadores "Switched"

- **Grid (rejilla)**
- Basado en circuitos integrados.
- Mejor para problemas 2−D (teoría de grafos, visión).

- **Hipercubo: cubo n−dimensional.**
- En un hipercubo n−dimensional, cada CPU tiene n conexiones a otras CPUs.
- La complejidad del cableado se incrementa en O(lg2n).
- El camino más largo crece en O(log2n) con el tamaño (grid: sqrt(n)).
- 1024 CPUs han funcionado durante años. Actualmente se llega a 16.384 CPUs.

Capítulo 3.2.− Conceptos SW

- **Débilmente acoplados:** máquinas más o menos independientes que pueden interaccionar entre sí. En caso de caída, alguna funcionalidad puede caer, pero no todas.
- **Fuertemente acoplados:** una sóla acción en paralelo.

Capítulo 3.3.− Combinaciones HW−SW

3.3.1.− SW débilmente acoplado con HW débilmente acoplado

- La combinación más típica: LAN: **NETWORK OPERATING SYSTEM**.
- Cada usuario tiene una Workstation para su uso exclusivo.
- Puede o no tener disco duro, pero sí Sistema Operativo.
- A veces conexión remota: rlogin machine, rcp m1:file1 m2:file2...pero es un esquema primitivo => CLIENTE/SERVIDOR (sistema de ficheros global, con servidores de fichero, necesitando la concepción de CLIENTE).
- Ya se tiende a la nueva concepción de objetos distribuidos, y movilidad de la funcionalidad.
- Diferentes usuarios pueden tener diferentes visiones del mismo sistema operativo: servidor con dos directorios.
- Ante diferentes SSOO: mismo protocolo de mensajes.

3.3.2.− SW fuertemente acoplado con HW débilmente acoplado

- Se trata de crear la ilusión de un monoprocesador virtual (**single−system image**).
- Cómo se consigue:

- Mecanismo de comunicación entre procesos.
- Esquemas de protección (no basta con bits de protección UNIX): Access Control Lists).
- Misma gestión de procesos (no nos vale con la idea del NOS).
- Mismo "look" del sistema de ficheros.
- Mismo modo de "System Calls" (mismo interfaz): Kernels Idénticos.
- Swapping independiente.

3.3.3.− SW fuertemente acoplado con HW fuertemente acoplado

- Por ejemplo: máquinas de Bases de Datos dedicadas. Multiprocesadores operados como sistemas de compartición de tiempo UNIX.
- **Clave:**
- Una sola cola de ejecución: lista de todos los procesos del sistema que están desbloqueados y listos para ejecutarse.
- El scheduler ha de estar en una región crítica (no puede haber dos CPUs cogiendo el mismo proceso).

3.3.4.− Comparación

CARACTERÍSTICA	**NOS**	**DOS**	**MP OS**
Monoprocesador virtual	Sí (con reservas)	Sí	Sí

Mismo SO/CPU	No	No (apetecible)	Sí
Copias de SO	N	N	1
Modo de comunicación	Ficheros compartidos	Mensajes	Memoria compartida
Protocolos de red "comunes"	Sí	Sí	No.
Single run queue.	No	No	Sí.
Compartición de ficheros con semántica bien definida	Normalmente no.	Sí	Sí

Nota: Algunas de las imágenes utilizadas en este tema están recogidas del curso "Commnunication Technology" del MIT.

Tema 4.− Introducción a los Sistemas Distribuidos

[Base histórica] [Modelos de SSDD] [Requisitos y características de un SD]

Capítulo 4.1.− Base histórica

- El comienzo potencial de la utilización de sistemas distribuidos es a principios de los 70. Tras el emergimiento de los
- minicomputadores; la utilización de estos como estaciones de trabajo gráficas uniusuario fue una gran éxito, sobre todo en el desarrollo de SW. Este éxito condujo a pensar en la posibilidad de conseguir los mismos resultados desde un punto de vista multiusuario.
- Problema: la tecnología necesaria para transformar aquel sistema uniusuario en otro coopoerativo no existía en aquellos momentos.
- Uno de los primeros éxitos en desarrollo fue gracias al Xerox Palo Alto Research Center (Xerox PARC), en 1971−1980:
- Servidores de impresión y de ficheros.
- Primera red local de alta velocidad (Ethernet).
- Sistemas distribuidos experimentales.
- Primera estación de trabajo: Alto:
- monitor monocromo de alta resolución.
- 128 Kb de memoria principal.
- Disco duro de 2.5 Mb.
- CPU microprogramada a la velocidad de 1 instrucción/2−6 microsegundos.
- Xerox DFS (Distributed File System).
- En primera época, también:
- UNIX
- ARPANet (padre de la Internet actual).
- SmallTalk (primer concepto de programación orientada a objetos).
- Después:
- Berkeley UNIX.
- Ethernet (ya comentada)
- Sun NFS
- AFS (Andrew File System)
- Mach
- Amoeba
- Chorus

Capítulo 4.2.− Modelos de sistema distribuído

4.2.1.− Estaciones y servidores

- Es el más común en la actualidad.
- A cada usuario se le asigna una estación.
- Las estaciones:
- ejecutan las aplicaciones.
- dan soporte a la interfaz de usuario (GUI).
- acceden a los servicios compartidos, mediante SW de comunicaciones.
- Los servidores dan acceso a:
- información compartida (servicio de ficheros)
- dispositivos HW compartidos (impresoras, scanners, ...)
- funciones del Sistema Operativo (autentificación, ...)
- Ejemplos:
- XDS y Cedar, del Xerox PARC.
- V−system de la Universidad de Stanford.
- Argus, del MIT.

4.2.2.− Banco de procesadores

- Los procesadores del pool tienen un CPU, suficiente memoria, pero no tienen ni discos ni terminales.
- Los usuarios acceden al sistema desde terminales−X.
- Gestor de recursos: controla el acceso a los procesadores del pool (PP).
- el usuario especifica sus requerimientos (CPU, memoria, programa).
- el Gestor de Recursos le asigna un procesador como ordenador personal.
- Ventajas con respecto a Estaciones y Servidores:
- mejor utilización de recursos: muchos puntos de entrada con pocas CPUs.
- mayor flexibilidad: se pueden dedicar procesadores para expandir servicios.
- compatibilidad con el SW preexistente.
- utilización de procesadores heterogeneos.
- Incoveniente: mala respuesta interactiva: terminales conectados a una linea serie.
- Ejemplo: CDCS, basado en el anillo de Cambridge.

4.2.3.− Miniordenadores integrados

- Basado en máquinas multiusuario.
- El usuario se conecta a una máquina específica.
- Enfoques históricos:
- acceso remoto mediante copias: de esta forma no se mezclan los espacios de nombramiento locales, pero no es distribuido.
- sistemas de ficheros contiguos:
- superdirectorio virtual: "/../", esquema de nombramiento global.
- no es distribuido: el nombre de los ficheros depende de su ubicación (máquina A, B, C????)
- ejemplo: NETIX, Newcastle Connection.
- **Locus (UCLA)**:
- cada ordenador mantiene su autonomía (conjunto completo de SW estándar, aplicaciones y servicios propios).
- esquema de nombramiento global: acceso independiente de la ubicación, posibilidad de migración de ficheros entre máquinas.

4.2.4.− Modelos híbridos

- **AMOEBA**, de la Universidad Libre de Amsterdam.
- Consiste en: sistema de estaciones y servidores más un pool de procesadores.
- Funcionalidad mixta:
- estaciones para las aplicaciones interactivas.
- procesadores variados.
- servidores especializados.
- Características:
- Kernel pequeño: planificación y paso de mensajes.
- el SO corre como procesos de usuario.
- servicio de gateway a WAN.
- gestión del pool mediante un servidor de carga y otro de procesos.
- Ventajas:
- recursos de procesamiento ajustados a las necesidades del usuario.
- ejecución concurrente.
- acceso a través de terminales (precio).

Capítulo 4.3.− Requisitos y características de un SOD.

4.3.1.− Características

4.3.1.1.− Compartición de recursos (resource sharing)

- Definición de **recurso**: rango de "cosas" que pueden ser compartidas en un sistema distribuido, de forma útil. Desde componentes HW (disqueteras, impresoras) a entidades SW (ficheros, ventanas, bases de datos).
- Dispositivos HW: se comparten para reducir costes.
- SW: compartición de herramientas de desarrollo, ficheros, facilidad de actualización.
- Groupware.
- **Gestor de recursos:** módulo SW que gestiona un conjunto de recursos de un tipo particular. Para cada conjunto de recursos existe un número de políticas diferentes, pero tambien características comunes.
- **Modelos de gestión de recursos:**
- Modelo **Cliente/Servidor**
- UNIX.
- Necesidad de centralización de gestión.
- Necesidad de diferenciar entre servidor y servicio.
- Inconveniente: algunas aplicaciones pueden requerir una cooperación más directa entre clientes.
- Modelo **orientado a objetos**
- Cada recurso es visto como un objeto, unívocamente identificado, y móvil.
- Ventajas: simplicidad, flexibilidad. Los objetos pueden actuar como usuarios de recursos y como gestores de recursos.
- Se necesita un gestor de objetos: colección de procedimientos y datos que caracterizan una clase de objetos.
- Ejemplos: Argus, Amoeba, Mach.

4.3.1.2.− Apertura (openness)

- Definición: permite a un SOD expandirse en múltiples direcciones.
- Un sistema puede ser abierto o cerrado con respecto a HW o SW:
- La apertura se consigue especificando y documentando las interfaces fundamentales del sistema y poniéndolos disponibles a los desarrolladores, para, finalmente, estandarizar esas interfaces:
- UNIX: los recursos del SO se acceden mediante System Calls, totalmente documentadas y disponibles

a los desarrolladores. La apertura de UNIX está inherentemente asociada a sus System Calls.

4.3.1.3.− Concurrencia (concurrency)

- Varios procesos existen al mismo tiempo en una máquina: ejecución concurrente.
- Con un solo procesador, esto implica planificación de CPU.

4.3.1.4.− Escalabilidad (scalability)

- Definición: capacidad de un sistema de operar de manera efectiva independientemente de su tamaño.
- La existencia de crecimientos en el sistema no implican modificaciones de la arquitectura básica del mismo.
- Distribución de cargas. Se trata de evitar las siguientes centralizaciones:

- componentes centralizados.
- tablas centralizadas.
- algoritmos centralizados. Características:
- ninguna máquina tiene información completa del estado del sistema.
- las máquinas toman decisiones basadas en información local.
- el fallo de una máquina no implica la terminación en fallo del algoritmo.
- no se asume la existencia de un reloj global.

- Técnicas empleadas:
- **replicación de datos.**
- **caching.**
- **duplicación o complemento de tareas entre servidores.**

4.3.1.5.− Tolerancia a Fallos (fault tolerance)

- Un fallo HW o SW puede producir inconsistencias o pérdidas de servicio.
- Dos ideas: **redundancia del HW, programas de recuperación de fallos.**
- Redundancia: arquitecturas de servidores en HOT STANDBY: muy caro (cada vez menos).
- Redireccionamiento de servidores (yahoo).
- Roll−back: transacciones atómicas.
- Sistemas de Alta disponibilidad (HA).

4.3.1.6.−Transparencia (transparency)

- Definición: abstracción que permite al usuario o al desarrollador desconocer los diferentes componentes del sistema distribuído.
- La separación de componentes en inherente en un sistema distribuído => acceso a recursos es un problema.
- **Tipos de transparencia** (ANSA Reference Manual, ISO's Reference Model for Open Distributed Processing, [ISO RM ODP]:

- *t. de acceso*: acceso a objetos locales o remotos de la misma manera.
- *t. de lugar*: acceso a objetos sin conocer dónde están.
- *t. de concurrencia:* varios procesos pueden operar concurrentemente usando objetos de información compartidos sin estorbarse.
- *t. de replicación:* diferentes réplicas de un mismo objeto de información sin enterarse a cuál se accede, ni diferencias entre ellos.
- *t. de fallo:* aislamiento de fallos, de forma que las aplicaciones puedan completar sus tareas.

- *t. de migración:* permite mover los objetos de información sin afectar a las aplicaciones.
- *t. de rendimiento:* redistribución de cargas en el sistema sin modificación en las aplicaciones.
- *t. de escalabilidad:* permite asumir cambios de tamaño del sistema y aplicaciones sin modificar la estructura del sistema ni los algoritmos de los programas.

    - Las más importantes: t. de acceso y de lugar: **TRANSPARENCIA DE RED.**

4.3.2.− Requisitos de diseño

Los componentes y arquitecturas del sistema utilizadas para cumplir con las características mencionadas arriba necesitan a su vez cumplir con unos requisitos:

4.3.2.1.− Metas en el diseño:

- **Prestaciones**
- **Fiabilidad**
- **Escalabilidad**
- **Consistencia**
- **Seguridad**

4.3.2.2.− Requisitos básicos de diseño:

- **Nombrado:** los nombres asignados a recursos u objetos han de tener un significado global e independiente de la localización. Además, ha de existir un sistema de interpretación que pueda trasladar nombres a recursos (ejemplo: internet(host + port)).
- **Comunicación**: doble optimización: HW y SW (protocolos, algoritmos). Sincronización en la comunicación.
- **Estructura del SW**: en sistemas centralizados, el diseño de SW es monolítico. En SSDD, los programas pueden acceder a diferentes servicios, cada uno de los cuales provee su propio interfaz de operación.

    - Capas en un sistema centralizado.
    - En un SD, los servicios deben ser fácilmente añadidos y escalables, por lo que el Kernel será la única parte rígida, restringiéndose a:
    - alojamiento y protección de memoria.
    - creación de procesos y planificador.
    - comunicación entre procesos (IPC).
    - manejo de dispositivos periféricos.
        - ♦ Categorías principales de SW en un SD. De áquí se distinguen tres categorías:
        - ♦ **Kernel (OS Kernel Services)**. Ya comentado anteriormente. Se ha llegado al desarrollo de microkernels: el conjunto mínimo de servicios y recursos a partir de los cuales el resto de servicios se pueden construir. De esta forma se consigue el máximo nivel de APERTURA.
        - ♦ **Servicios abiertos (Open Services)**: facilidades añadidas. Generalmente se ofrece el conjunto de servicios típicos de un sistema UNIX (servicio de ficheros, correo electrónico...).
        - ♦ **Soporte para programación distribuida (distributed programming support):** por ejemplo, RPC.
        - ♦ **Alojamiento de la carga de trabajo:**
        - ♦ **Modelo de pool de procesadores.**
        - ♦ **Aprovechamiento de "idle workstations"**.
        - ♦ **Multiprocesadores de memoria compartida.** Cada procesador ejecuta lo que va pudiendo.
        - ♦ **Mantenimiento de la consistencia:**
        - ♦ **c. de actualización**: problema que aparece cuando varios procesos acceden y actualizan datos constantemente. Es necesaria la atomicidad de operaciones, mediante mecanismos de

exclusión mutua.

- **c. de replicación**: la réplica de datos ha de efectuarse óptimamente.
- **c. de caché.**
- **c. de fallos:** en un sistema centralizado, si falla, todo falla. En uno distribuido, existirán diferentes MODOS DE FALLOS (failure modes), donde si el sistema en conjunto falla, en cada máquina su proceso se encontrará en un estado determinado: transacciones atómicas.
- **c. de reloj:** debido al ancho de banda de las redes, la sincronización entre máquinas no es obvia. Afortunadamente, este requerimiento en SSDD no suele ser absoluto, sino relativo a orden.
- **c. de interfaz de usuario.**

4.3.2.3.− Requerimientos del usuario:

- **Funcionalidad:** qué debería hacer el sistema por los usuarios.
  - ◊ **Requerimiento mínimo:** la funcionalidad provista por un sistema distribuído no debería de ser menos que el obtenido por un sistema aislado.
  - ◊ **Requerimiento máximo:** mejora sobre los servicios provistos por cualquier ordenador personal por sí sólo, a través de:
  - ◊ compartición a través de la red, y especialización de recursos (p.e. servidor de impresión).
  - ◊ aplicación de las ventajas de la distribución al nivel de APIs (application programming interface), para que puedan ser implementadas.
    - · Opciones de migración de computación centralizada multiusuario a computación distribuída:
    - · **Adaptar Sistemas Operativos ya existentes:** adición de servicios, tipo NFS.
    - · **Migrar a un SO totalmente nuevo**: el SW anterior es inútil.
    - · **Emulación:** migrar a un SO nuevo, pero con características de emulación de uno o más SSOO ya existentes. Ejemplos: Mach, Chorus. **Problema: prestaciones.**
    - · **Reconfiguración:** necesidad del sistema para adaptación a cambios sin causar discapacidad a la provisión de servicio existente.
      - Diseño de un sistema distribuído en relación a la escala temporal:
      - **Cambios a corto plazo:**
        - una máquina, red, componente... estropeada se reemplaza por otra.
        - la carga de trabajo se ha de reequilibrar, pasándo procesos a máquinas libres.
        - la comunicación de red se minimiza.
        - **Cambios a medio y largo plazo:**
          - ◊ Asignación de nuevos roles a máquinas.
          - ◊ Adición de nuevas máquinas, o nuevos tipos.

        Calidad de servicio: temas de prestaciones, confianza y seguridad.

Tema 5.− Diferentes Sistemas Distribuidos

[Introducción][El Kernel][Procesos y Threads][Dominios de Protección][Comunicación e Invocación][Memoria Virtual][Sistemas Distribuidos de Tiempo Real][Un caso práctico: Sun Spring][Ejercicios]

5.1.− Introducción

- ◊ La base de construcción de un SD es su **Microkernel.**
- ◊ En este capítulo el concepto de SOD es purista (UNIX no puede gestionar procesos en red de forma transparente, por ejemplo).
- ◊ Un kernel será al fin y al cabo un **gestor de recursos.**
- ◊ Qué requerimos de un kernel de un SD:
- ◊ **Encapsulamiento**: conjunto de operaciones que satisfagan las necesidades de sus clientes. Los detalles de gestión de memoria han de permanecer ocultos.
- ◊ **Procesamiento concurrente:** acceso concurrente a recursos por parte de los usuarios.
- ◊ **Protección.**
- ◊ **Invocaciones:** acceso a un recurso encapsulado por parte del cliente, ya sea mediante una System Call, o una RPC, por ejemplo.
- ◊ Tareas relativas a invocaciones:

♦ *Name resolution:* el servidor o kernel que gestiona un recurso ha de ser localizado a partir del identificador del recurso.
♦ *Comunicación:* los parámetros de entrada de la invocación, así como el resultado de salida, han de poder ser transmitidos a través de la red de comunicaciones.
♦ *Planificación*:

5.2.− El Kernel

5.2.1.− Protección en un Kernel

- ◊ ¿Qué es un **kernel**?: es un programa cuyo código se ejecuta con total privilegio de acceso sobre los recursos físicos en el host: gestión de memoria, registros de proceso, etc.
- ◊ **Protección**: el kernel puede preparar espacios de direcciones separados, para evitar que un proceso pueda acceder a datos restringidos de otro. Un proceso no puede acceder a memoria fuera de su espacio de direcciones.
- ◊ El kernel tiene un espacio de direcciones propio.
- ◊ El kernel ejecuta sus procesos en modo **supervisor**, y se las arregla para que cualquier otro proceso de usuario tenga un modo no privilegiado (**unprivileged**).
- ◊ El kernel gestiona la invocación de recursos mediante sus **system call traps**: al invocar este tipo de llamadas, una instrucción máquina, llamada **TRAP**, pone el procesador en modo de supervisión y cambia al espacio de direcciones del kernel, y es

forzado a utilizar un "**handler**" provisto por el kernel, de forma que ningún otro usuario pueda conseguir control del HW.

5.2.2.− Kernels monolíticos y microkernels

- ◊ Un SD tiene como una de sus ventajas principales la capacidad de adición de nuevos servicios sin prejuicio de los ya existentes.
- ◊ Se puede obtener un grado de apertura en sistemas tradicionales como UNIX, añadiéndoles ciertas características.
- ◊ Por ejemplo: OSF DCE (Distributed Computing Environment of the Open Software Foundation).− es una arquitectura SW para computación distribuida, diseñada para funcionar en diferentes kernels, como UNIX, VMS, OS/2, etc. Incluye estándars para RPC, servicios de name binding, servicios de sincronización de tiempo, servicios de seguridad y servicios de threads.
- ◊ Pero, ¿qué ocurre con la apertura de servicios del propio kernel?
- ◊ Tipos de kernel:

♦ **Kernels monolíticos:**
♦ Como el de UNIX.
♦ Es *masivo*: realiza todas las funciones básicas de un SO.
♦ Ocupara más o menos 1 Mb de código fuente y datos.
♦ *No diferenciado*: no está codificado de forma modular.
♦ *Intratable*: el rediseño de SW es complicado.
♦ Un kernel monolítico puede estar diseñado para funcionar como SOD (Sprite), pero tendría problemas ante ajustes.
♦ **Microkernels**:
♦ El kernel solo provee las **abstracciones básicas** (procesos, memoria, IPC).
♦ El resto de los servicios son provistos por servidores que se **cargan dinámicamente** en aquellas máquinas que necesitan ofrecer esos servicios.
♦ Generalmente, estos servicios serán ejecutados como procesos del usuario; cuando se necesite un acceso directo a HW, existirán *system calls* especiales que permitan manejar interrupciones del sistema. Como contrapartida, el microkernel del Chorus permite que los servicios se carguen dinámicamente, o en el espacio de direcciones del kernel, o en el del espacio de usuario.

- ◊ Papel del microkernel:
- ◊ Entre el HW y los denominados subsistemas.
- ◊ Puede haber más de un interfaz de llamadas al sistema, es decir, más de un "Sistema Operativo". Por ejemplo, las implementaciones de Unix y OS/2 por encima de Mach.
- ◊ Comparación:
- ◊ *Ventajas del microkernel:*
- ◊ Apertura.

◊ Modularidad inherente (para un kernel monolítico, se puede forzar mediante técnicas de ingeniería de SW, como en MULTICS, o de orientación a objetos, como en Choices).
◊ Más fácil de programar, libre de bugs.
◊ Pocas personas son necesarias para la implementación.
◊ Más fácil de mantener.
◊ *Ventajas del kernel monolítico:*
◊ Eficiencia en invocación de operaciones: el cambio de espacio de direcciones es caro, y en un kernel monolítico, la mayoría de las operaciones se pueden realizar desde el kernel.

5.2.3.− Arquitectura de un microkernel

◊ Diseñado para ser **portable** => lenguaje de alto nivel (C, C++, Java).
◊ **Diseño en capas**: acceso a recursos HW en capa más profunda (manipulación del procesador, gestión de memoria, registros, gestión de interrupciones y traps, etc).
◊ Componentes principales:
◊ **Gestor de procesos** (process manager): creación y manipulación de procesos. Generalmente existirá algún subsistema por encima de mejora de la utilidad.
◊ **Gestor de threads** (thread manager): creación, sincronización y planificación de threads. La política de planificación se puede dejar a módulos superiores.
◊ **Gestor de comunicaciones** (communication manager): comunicación entre threads de diferentes procesos locales, o incluso en procesos remotos. A veces, es necesario un soporte adicional superior.
◊ **Gestor de memoria** (memory manager): gestión de recursos de memoria física, caché, etc.
◊ **Supervisor**: "dispatcher" de interrupciones, traps y excepciones.

5.3.− Procesos y Threads

◊ **Mach y Chorus** mejoran la copia del espacio de direcciones, mediante el método **copy−on−write:** cuando se crea un proceso hijo, no se duplica el espacio de direcciones, sino que el hijo apunta al del padre, hasta que el hijo intenta modificarlo.

5.4.− Dominios de Protección

◊ Un dominio de protección (protection domain) es un entorno de protección compartido por una colección o conjunto de procesos: es un conjunto de pares {recurso, derechos}. Por ejemplo, el dominio de protección de un proceso en UNIX se determina con

los identificadores de usuario y de grupo.
- ◊ Es una abstracción. Existen dos implementaciones básicas:

- ♦ **Capacidades** (capabilities): conjunto de capacidades por proceso.
- ♦ **Listas de control de acceso** (access control list): cada recurso guarda una lista, dando los dominios que tienen acceso al recurso, y las operaciones permitidas.

5.5.− Comunicación e Invocación

- ◊ A tratar en el tema: Remote Procedure Call.

5.6.− Memoria Virtual

- ◊ Diferencia con respecto a la memoria virtual de un SO convencional: el interfaz de almacenamiento de back−up es un **servidor** en vez de un disco local.
- ◊ La memoria virtual permite incrementar el número de procesos simultaneos, mediante swapping en cambios de contexto.
- ◊ **External pager** (gestor de memoria externo): la información de paginación se almacena en el servidor, por lo que el paginador habrá de ser externo al lugar donde el proceso se crea. El servidor mantiene **objetos de memoria (memory objects)**, regiones del espacio de direcciones del servidor que serán "*mapeadas*" al proceso remoto (por ejemplo, los datos de un fichero). Después del *mapeo*, existirá un paso de mensajes entre el kernel creador del proceso y el servidor externo para el tratamiento de la paginación.

5.7.− SSDD de Tiempo Real

5.7.1.− Introducción

- ◊ La corrección de su ejecución no sólo depende de la secuencia lógica de instrucciones ejecutadas, sino de cuándo son ejecutadas.
- ◊ Cuando aparece un estímulo, el sistema debe responder de una cierta forma antes de un tiempo determinado; el hecho de que responda correctamente, tras el tiempo necesario, se considera como un fallo del sistema.
- ◊ Ejemplos: lector de Compact−Disc, mando y control, control aéreo...
- ◊ **Tipos de estímulos:**

- ♦ **Periódicos**: ocurren regularmente cada delta segundos.
- ♦ **Aperiódicos:** estímulos recurrentes, pero no regulares.
- ♦ **Esporádicos:** inesperados.

- ◊ **Tipos de sistemas de tiempo real:**

- ♦ **Duros:** restricciones inflexibles.
- ♦ **Blandos**: restricciones flexibles.

5.7.2.− Factores de diseño

- ◊ **Sincronización de relojes.**
- ◊ **Orientación a eventos vs. Orientación a tiempo**

♦ **Orientación a eventos**: cuando un evento ocurre, se detecta mediante algun sensor, y se lanza una interrupción a la CPU.
- ◊ Problema: puede fallar si existe una gran carga (muchas alarmas a la vez).
- ◊ Solución: correlación de alarmas.

♦ **Orientación a tiempo**: existe una interrupción de reloj cada delta segundos.
- ◊ Problema: elección de delta.
- ◊ **Predicción**: es necesario saber desde la fase de diseño cómo se tiene que comportar *EXACTAMENTE* el sistema en cada situación: análisis estadístico del comportamiento.
- ◊ **Tolerancia a fallos.**
- ◊ **Soporte de lenguaje:**
- ◊ el tiempo máximo de ejecución de cada tarea ha de poder ser computada en tiempo de compilación => **no se permiten bucles while** (sólo for con parámetros constantes); no se permite recursividad tampoco.
- ◊ esto nos lleva a la necesidad de lenguajes de programación específicos: **Ada**, por ejemplo.
- ◊ una característica concreta: variable **clock**, para poder escribir: *every (25 msec) { ... }*

5.7.3.− Planificación en Tiempo Real

- ◊ Los sistemas en tiempo real se programan generalmente como una colección de tareas cortas (procesos o threads), cada una con una función bien definida y un tiempo de ejecución.
- ◊ Problema: **orden de ejecución** => se necesitan unos algoritmos de scheduling, caracterizados por:

♦ Tiempo real **DURO** contra tiempo real **BLANDO**.
♦ Planificación "preemptive" contra "nonpreemptive".
♦ Dinámica contra estática: una planificación dinámica, ante la llegada de una nueva tarea, ve si ha de ejecutarla automáticamente o continuar ejecutando la tarea actual. Una planificación dinámica no preemptiva tan sólo guardaría información de que otro proceso esta en estado de ejecución.
♦ Centralizada contra descentralizada: en una planificación descentralizada, cada procesador decide lo que hay que hacer con sus procesos.

Tema 6.− Comunicación en Sistemas Distribuidos

[Modelo Cliente/Servidor]

6.1.− Modelo Cliente/Servidor

[Características][Tipos de sistemas C/S][Pesadez C/S][Arquitectura 2, 3−tier]["Star−Trek"][Arquitectura]

6.1.1.− Características de un sistema Cliente/Servidor

- ◊ Cómo podemos diferenciar entre un sistema C/S y un sistema sencillamente conectado en red. Todo es muy parecido a las características de los Sistemas Distribuidos, aunque especializándonos un poco.

♦ **Servicio**
- ◊ el cliente es un consumidor de servicios.
- ◊ el servidor es un proveedor de servicios.
- ◊ el modelo C/S nos da una separación limpia de funcionalidad basada en la idea de SERVICIO.

♦ **Recursos compartidos**
♦ **Protocolos asimétricos**
- ◊ relación de muchos a uno.
- ◊ los clientes inician el diálogo, pidiendo un servicio.
- ◊ los servidores esperan de forma pasiva.

♦ **Transparencia de localización**
♦ **"Mix−and−match"**
- ◊ mezcla de plataformas, SSOO, etc.

♦ **Comunicación basada en el paso de mensajes.**
♦ **Encapsulamiento de servicios**
- ◊ petición de servicio = caja negra. El servidor es el especialista, no el cliente, por lo que no ha de tener conocimiento alguno en el **CÓMO**, sino en el **QUÉ**.

♦ **Escalabilidad**
♦ *Horizontal*: adición/eliminación de clientes, sin impacto.
♦ *Vertical*: migración a servidores más grandes, o a multiservidores.
♦ **Integridad**

6.1.2.− Tipos de Sistemas Cliente/Servidor

♦ **Servidores de ficheros**
- ◊ compartición de fichero (documentos, imágenes, BLOBS, ...).

♦ **Servidores de BBDD**
- ◊ el servidor es quien procesa la consulta y devuelve los datos => el Cliente es ligero (Quest, Paradox), no como el servidor de fichero.

♦ **Servidores de transacciones**
- ◊ el cliente invoca "procedimientos remotos" que residen en el servidor con una base de datos SQL. Esos PRs ejecutan un conjunto de queries SQL => por red sólo se pasa un mensaje de petición y otro de repuesta => On Line Transaction Processing: OLTP.
- ◊ se suele utilizar para sistemas críticos (1.3 segundos de tiempo medio de respuesta).
- ◊ requieren: integridad, seguridad.
- ◊ *Tipos:*

◊ TP Lite: basado en procedimientos almacenados.
◊ TP Heavy: TP Monitors.
◊ **Servidores Groupware**
◊ Gestión de información semi−estructurada (texto, imágenes, correo, bulletin boards, flujo de trabajo).
◊ Ej: Lotus Notes.
◊ **Servidores de Objetos**

· La aplicación C/S se escribe como un conjunto de objetos comunicantes.
· Los objetos cliente se comunican con los objetos servidores mediante un ORB (Object Request Broker):
· el cliente invoca un método sobre un objeto remoto.
· el ORB localiza la instancia de esa clase.
· el ORB invoca el método pedido.
· el ORB devuelve los resultados al objeto cliente.

- Los objetos servidores han de proveer soporte de Concurrencia y Compartición.
- Ejemplos de OMG CORBA:
- IBM's SOM 3.0.
- Sun's NEO.
- HP's ORBPlus.
- Iona's ORBIX.
- Visigenic's Visibroker.
  - ♦ Aparte: Microsoft's DCOM.
- **Servidores Web**
  - ♦ Idea básica: clientes universales, portables y pequeños que hablan con servidores pesados: un servidor Web devuelve documentos cuando un cliente los pide.
  - ♦ Comunicación C/S mediante un sistema parecido al RPC: protocolo HTTP.
  - ♦ Primer paso hacia la red objetual (Object Web).

6.1.3.− "Pesadez" Cliente/Servidor

- ♦ Un modelo Cliente/Servidor puede distribuir la carga entre el C y el S:

- Cliente Pesado: más tradicional: servidores de fichero (ejercicio).
- Servidor Pesado: groupware, Web, transacciones.
  - ♦ Los objetos distribuídos

dependerán.
- Las dos ideas pueden cooperar: groupware "all−in−one": servidores de ficheros, transacciones y objetos.
- (transparencia 6.4)

6.1.4.− Arquitectura 2, 3−tier

- Cuáles son las unidades funcionales básicas en una aplicación:
- Interfaz de usuario.
- Lógica.
- Datos compartidos.
- Arquitectura 2−Tier
- la lógica está "enterrada" dentro del GUI del cliente, o de la BBDD del servidor, o en ambos.
- Ejemplos: servidores de ficheros, s. de BBDD con procedimientos almacenados.
- Arquitectura 3−Tier
- la lógica (y/o sus procesos) viven en la capa intermedia.
- Ventajas:
- escalabilidad.
- robustez.
- flexibilidad.
- pueden integrar datos de diferentes fuentes.
- Ejemplos: TP Monitors, s. de objetos distribuídos, s. Web.

6.1.5.− Visión "Star−Trek"

- El futuro está ya aquí (tabla comparativa).
- Qué necesitamos para conseguirlo:

- **Alto grado de procesamiento de transacciones:**
- transacciones anidadas entre múltiples servidores.
- transacciones seguras.
- **Agentes "desarraigados":** Java, agentes móviles, scripts multiplataforma (Javascript, Perl)
- **Entidades inteligentes de autogestión:**

- No podemos tener un administrador por cada máquina de 60.000 pts.

- **Middleware inteligente:** idea de monoprocesador virtual.
  - Estas son necesidades de alto nivel, y no todas las apps. pueden o tienen que conseguirlo.

6.1.6.− Arquitectura

- ***Bloque Cliente***
- tiene GUI/OOUI.
- puede acceder a servicios distribuidos.
- generalmente, el C se encarga de lo local, dejando los temas distr. al middleware.
- ejecuta un componente del DSM (Dist. System Management): desde un agente PC, al sistema de atención al cliente.
- ***Bloque Servidor***
- generalmente se ejecuta por encima de algún SW servidor (p.e. NT 4.0 Server).
- Next Generation (¿actual?):
- Servidores BBDD SQL.
- TP Monitors.
- S. Groupware.
- S. de Objetos.
- S. de Web.
- Tb. ejecuta un componente DSM (desde agente PC al back−end del sistema gestor).
- ***Middleware***
- se ejecuta en ambas partes.
- es el sistema nervioso de la infraestructura C/S.
- Se divide en:
- pila de transporte
- NOS
- "Service−specific" Middleware
- Tb. tiene un componente DSM.

6.1.6.1.− Pila de Transporte

- **Stack sandwich**
- provee los "ganchos" de unión de protocolos de diferentes vendedores en un SO.
- Los nuevos SSOO han de incoporar muchos protocolos, redirectores y APIs => interfaces BIEN DEFINIDOS, que escondan esos sistemas propietarios.
- **Driver lógico de red**
- provee un único interfaz para todos los adaptadores de red. Es un interfaz entre el adaptador de red y las pilas de transporte.
- lo último que quieren los fabricantes de pilas de transporte es tener que escribir un interfaz para cada posible pila.
- Ejemplos: Microsoft/3Com's NDIS, Novell's ODI.
- **APIs independientes de transporte**
- permiten a los desarrolladores "enchufar" sus programas en un único interfaz (que soporta protocolos múltiples).
- Ejemplos:

- Sockets.
- TLI (transport layer interface) utilizado en Netware, y algo en Unix.
- CPI−C: API punto−a−punto. En SNA y TCP/IP.
- Name Pipes: por encima de NetBIOS, IPX/SPX y TCP/IP.

- **Emparejador de protocolos** (protocol matchmaker)
- Permite que applicaciones escritas para un transporte específico (SNA, por ejemplo) se pueda ejecutar en otros (TCP/IP, IPX/SPX).
- Ejemplo: LotusNotes para NetBIOS => SNA sin cambiar una línea de código.

- Herramienta: IBM's AnyNet.