Sistemas Operativos

2º Curso

Dobles Grados en Ingeniería Informática y Matemáticas y ADE

Tema 1:

Estructuras de SOs

Estructuras de SOs

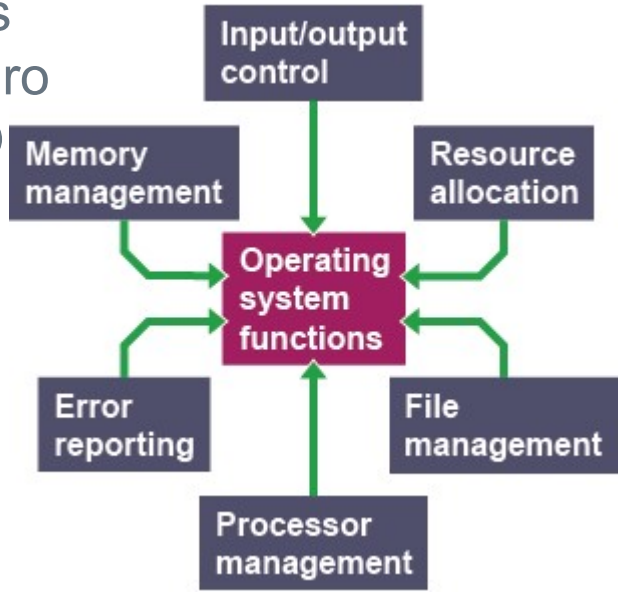
- Tipos de arquitecturas SOs:
  - Monolítica
  - Capas
  - Máquinas virtuales:
    - Soporte de Linux a la virtualización
- Sistemas operativos de propósito específico:
  - Sistemas de tiempo compartido
  - Sistemas de tiempo-real
  - Sistemas distribuidos y paralelos

1. Tipos de arquitecturas

Cómo organizar los componentes de un SO

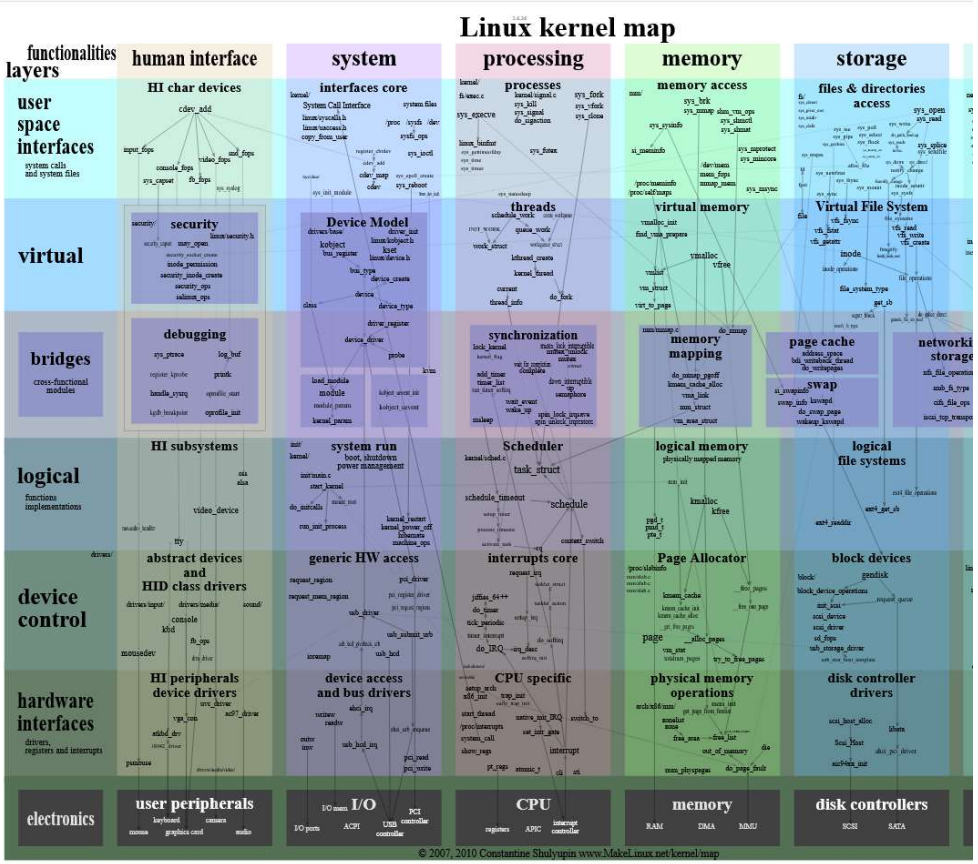
Servicios y componentes del SO

- Algunos de los servicios suministrados por el sistema operativo (SO):
  - Ejecutar diferentes actividades
  - Acceso a información permanente
  - Acceso a los dispositivos
  - Suministrar memoria para ejecutarlos
  - El uso de los recursos debe ser seguro
  - Interfaz de acceso a servicios del SO
  - . . .
- Estos servicios se suelen integrar en componentes con interrelaciones entre ellos.



Estructura del SO

- Los SOs constan de muchos componentes entre los cuales existen muchas y complejas relaciones.
- La estructura del SO, o arquitectura, define cuales son dichos componentes y como se inter-relacionan.
- Ejemplo de la Figura: núcleo (*kernel*) de Linux.



Fuente: https://makelinux.github.io/kernel/map/

La gran cuestión

- Las preguntas importantes desde el punto de vista arquitectónico son:
  - ¿Cómo se organiza todos estos elementos?
  - ¿Qué procesos hay para ejecutar el sistema?
  - ¿Cómo cooperan dichos procesos?

¿Cómo construir un sistema complejo que sea eficiente, fiable y adaptable?

Modelo de procesos

- La estructura de SO multiprogramado propuesta en el tema de repaso (Tema 0) no contempla la existencia de procesos ejecutándose el modo kernel.
- El problema que presenta es que si no hay procesos ejecutándose en modo kernel (de sistema), éste no se ejecuta, provocando poca responsividad especialmente en el tratamiento de dispositivos.
- En Unix, el problema se abordó introduciendo cierto número de procesos, **demonios**, que ejecutan en modo kernel labores de sistema. En Linux, como veremos en el Tema 2, se han sustituido por **hilos kernel**.

Características de una buena arquitectura

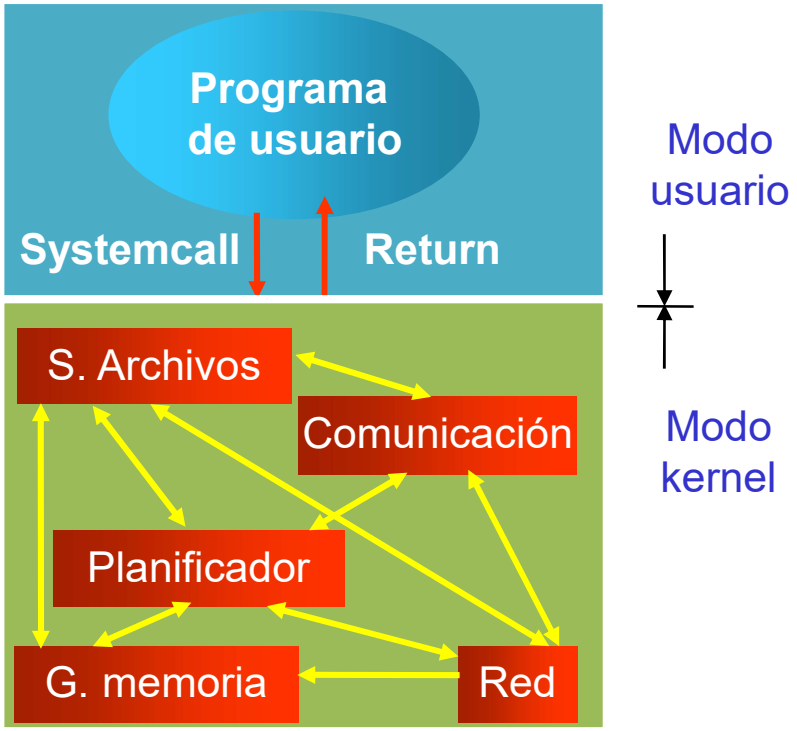
- **Eficiente**: el SO debe ser lo más eficiente posible para no degradar el funcionamiento global del sistema.
- **Fiable**: Dos aspectos a cosiderar:
  - **Robusto** – El SO debe responder de forma predecible a condiciones de error.
  - El SO debe protegerse activamente a si mismo y a los usuarios frente a acciones accidentales o malintencionadas.
- **Adaptable**: Deberíamos poder añadir/modificar funcionalidad del SO de forma flexible.

Arquitecturas de SOs

- El punto de partida para alcanzar estas características se ha partido tradicionalmente de la **arquitectura software**, o estructura con la cual construimos el software (sus componentes y sus interacciones).
- Debemos distinguir entre:
  - **Arquitectura de diseño**: cómo se diseña 'sobre el papel'.
  - **Arquitectura de ejecución**: cómo se construye(n) el(los) programa(s) del SO y cómo se ejecutan.
- Ambas arquitecturas pueden diferir para un mismo sistema. Por ejemplo, la arquitectura de ejecución de muchos SOs actuales es esencialmente monolítica (un único programa ejecutable) si bien su arquitectura de diseño puede ser de capas: Linux y Windows.

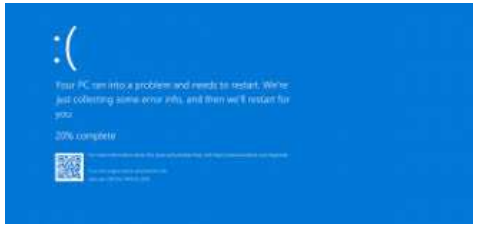
Estructura monolítica

- Todos los componentes del sistema se ejecutan en modo kernel. El SO es un único programa–ejecutable). Las relaciones entre ellos son complejas.
- El modelo de obtención de servicios es la llamada a *procedimiento 'protegido'* que tiene el menor coste.
- ¿Cuál es el coste de obtener un servicio?



Monolíticos: caracteristicas

- Son **difíciles de comprender**, ya que son un único programa (decenas de MB), por tanto, difíciles de modificar y mantener.
  - Ej.: Versión 4.5 del kernel de Linux: 33 LoC [1]
- **No confinamiento de errores**: un fallo de algún módulo puede provocar la "caída" del sistema.
  - Ej.: *Pantalla azul* de Windows o *panic()* en Linux.



- Por esto, los diseñadores han buscado mejores formas de estructurar un SO para simplificar su diseño, construcción, depuración, ampliación y mejora de sus funciones.

[1] Disponible (17/7/2020) en https://en.wikipedia.org/wiki/Linux_kernel

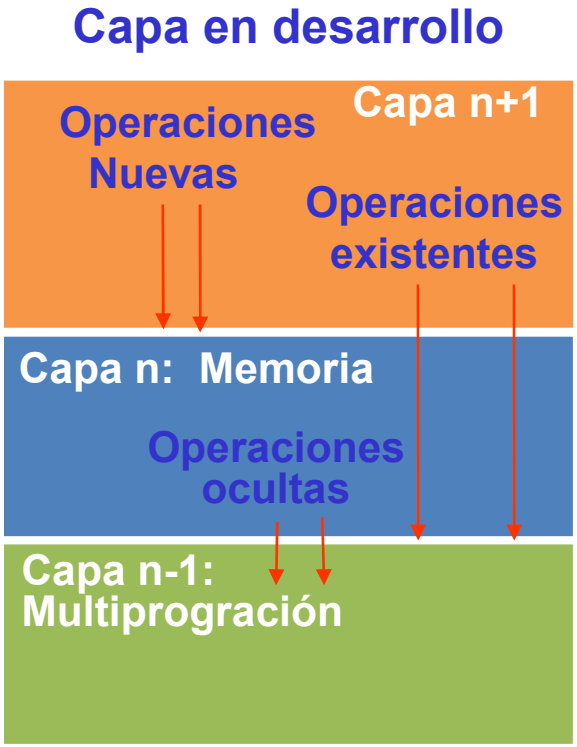
Monolíticos: Linux

- Los kernels actuales de Linux son de este tipo (al igual que la mayoría de sistemas operativos de producción) por razones de **eficiencia**.
- Esto no quiere decir que el código fuente del sistema no esta estructurado (ver directorio */usr/src/linux/*[1]).
- Indica que el programa ejecutable correspondiente en complejo, en componentes e interacciones entre los mismos (recordar mapa del kernel en http://www.makelinux.net/kernel_map/).

[1] Por defecto, no están instalados. En Ubuntu se instalan con `apt-get install linux-source`.

Arquitectura de capas

- Implementado como una serie de capas; cada una de ellas es una máquina más abstracta para la capa superior.
- Por modularidad, las capas se seleccionan para que cada una utilice sólo funciones de las capas inferiores.



Capas: ventajas/desventajas

- Los sistemas de capas son jerárquicos pero los reales son más complejos, p. ej.,
  - El sistema de archivos podría ser un proceso en la capa de memoria virtual.
  - La capa de memoria virtual podría usar archivos como almacén de apoyo de E/S.
- Existe sobrecarga de comunicaciones entre las distintas capas.
- A menudo, los sistemas se modelan con esta estructura (arq. diseño) pero no se construyen así (arq. ejecución).

Virtualización

"Virtualización es una tecnología que combina o divide recursos de computación para presentar uno o varios entornos de operación utilizando metodologías como particionamiento o agregación ya sea hardware o software, simulación de máquinas completa o parcial, emulación, tiempo compartido, y otras"

[Susanta Nanda y Tzi-cker Chiueh, "A Survey on Virtualization Technologies", RPE Report, SUNY at Stony Brook, New York. Feb. 2005.]

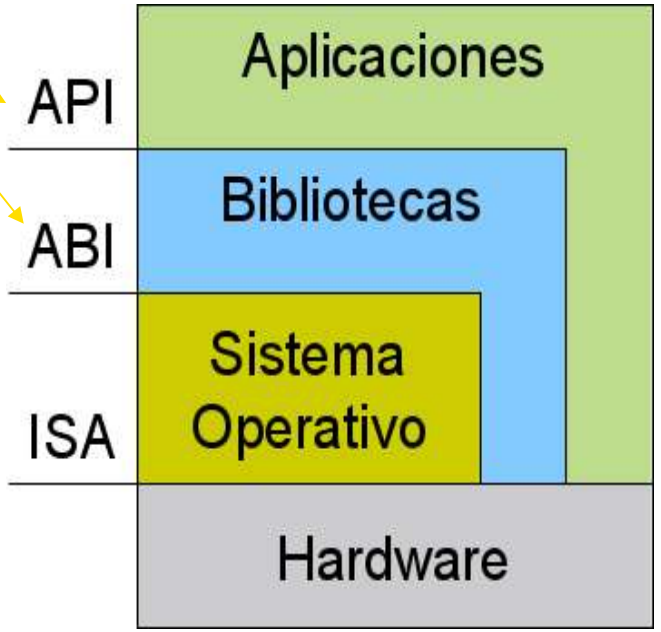
Máquina virtual

- Denominamos **Máquina Virtual** (MV) al software que ‘simula’ una computadora y que puede ejecutar programas como si fuese una computadores ‘real’.
- Podemos verlo como un duplicado eficiente y aislado de una máquina real.
- Los procesos que ejecuta una MV están limitados por los recursos y abstracciones que proporciona. Además, estos procesos están aislados del resto de procesos en otras MVs.

Virtualización: tipos

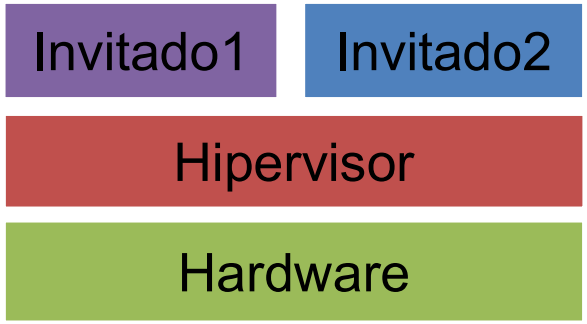
▷ Podemos clasificar las MVs:

- Máquina virtual de procesos:
  I. Nivel biblioteca: *Wine*
  II. Nivel Lenguaje programación:
  JVM, *Microsoft .NET CLI*
  III. Nivel SO: *Jail*, *Containers*
- Máquina virtual de sistemas
  I. Emulación:
  Traducir ISA1 →ISA2
  II. Virtualización:
  1. Hipervisor nativo (Tipo 1)
  2. Hipervisor anfitrión (Tipo 2)



Tipos de hipervisores

- **Hipervisor Tipo 1 (*bare metal*):**



  - Ejemplos:
    - Xen (https://xenproject.org/)
    - VMware ESX (https://www.vmware.com/es/products/esxi-and-esx.html)

- **Hipervisor Tipo 2 (*hosted*):**

  - Ejemplos:
    - VMWare Workstation (https://www.vmware.com/es/products/workstation-pro.html)
    - KVM (https://www.linux-kvm.org/)

Hipervisores: implementación

- Los SOs están construidos para ejecutarse en modo kernel. Ahora es el hipervisor el que se ejecuta en modo kernel ¿cómo se solventa? = ¿cómo manejo las instrucciones privilegiadas?
- Según la implementación podemos separarlos en:
  - *Traducción binaria* de código binario "crítico" del SO invitado (no de las aplicaciones).
  - *Paravirtualización*: El código del SO invitado es recompilado para amoldarlo a la API anfitriona, eliminando la necesidad de que la MV atrape las instrucciones privilegiadas.
  - *Virtualización asistida por hardware*: trata de resolver problemas del hardware relativos a la gestión de trampas. Ej. VT-x de Intel o SVM de AMD.

2. Soporte de Linux a la virtualización

Describiremos los elementos de los kernels actuales de Linux destinados a soportar virtualización

Soporte de Linux a la virtualización

> Linux soporta una virtualización ligera basada en dos mecanismos:

- **Espacios de nombres** (*namespaces*) – permite que se vean propiedades globales del sistema desde diferentes aspectos o "vistas".
- **Grupos de control** (*cgroups*)- permite la gestión particionada de recursos asignables a tareas o grupos de tareas.

> Además, el kernel da soporte a MV como UML, XEN, KVM, VirtualBox, VMWare, Wine, etc.

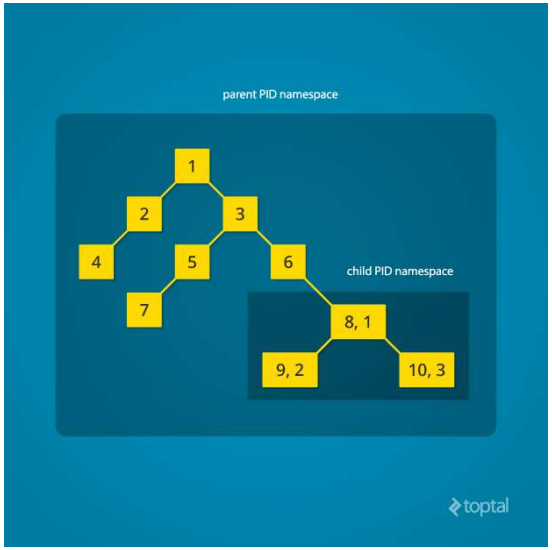
Namespaces

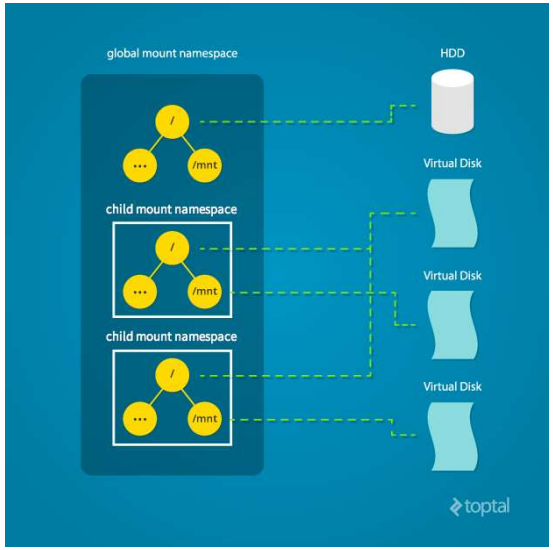
▷ Los espacios de nombre actualmente incluidos son:

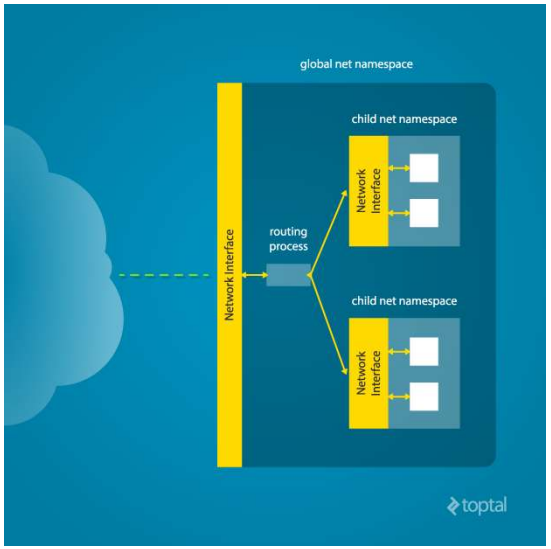
- ***System V IPC*** – mecanismos de comunicación entre procesos.
- ***mounts*** - sistemas de archivos montados.
- ***UTS*** *(Unix Timesharing System)* – información del sistema (nombre, versión, tipo arquitectura, etc.)
- ***pid*** – espacio de nombres de identificadores de procesos
- ***network*** – conjunto de dispositivos de red.
- ***userid*** – permite limitar los recursos por usuario.

▷ Otros Sos que soportan una virtualización similar son: *Zones* de Solaris o *Jails* de FreeBSD.

Namespaces: ejemplos







Fuente: https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces

Namespaces: usos

- Posibles usos de los *namespaces*:
  - Servidores Privados Virtuales (VPS) como por ejemplo Linux *Containers* (lxc.sourceforge.net)
  - *Application checkpoint* and *restart* (ACS)
  - En clúster:
    - Sustitución de NFS
    - Re-construcción de */proc*

Grupos de control

- Controlan los recursos asignados a una tarea/grupo de tareas.
- Diferentes subsistemas (controladores de recursos):
  - ***cpu***: utilizado por el planificador para suministrar el acceso de las tareas de un cgroup a la CPU.
  - ***cpuset***: asigna CPUs individuales y memoria en multicores.
  - ***devices***: permite/deniega el acceso a un dispositivo.
  - ***memory***: limita el uso de memoria a tareas de un cgroup, y genera informes automáticos.
  - ***blkio***: establece los límites de accesos de E/S desde/hacia dispositivos de bloques (discos, USB,…)
  - **ns**: subsistemas de espacios de nombres.

Grupos de control: uso

- Varias formas de uso:
  - Seudo-sistema de archivos
  - cgroupsfs
  - herramientas *libcgroup*
  - demonio *engine rules*
- Ejemplos, podemos:
  - Ver los grupos con `cat /proc/cgroups`
  - Ajustarlos a través de `/sys/fs/cgroup/`
  - `Algunas ordenes se construyen sobre cgroups: cpuset`

3. SOs de propósito específico

Qué características especiales tienen algunos SOs
Trabajo autónomo: cualquier libro de fundamentos de SOs

Sistemas de tiempo compartido

- Soportan el uso “interactivo” del sistema:
  - Cada usuario tiene la ilusión de disponer de la máquina completa.
  - Tratan de optimizar el tiempo de respuesta.
  - Basados en asignar fracciones de tiempo - se reparte el tiempo de CPU de forma equitativa entre los procesos.
- Permiten la participación activa de los usuarios en la edición, depuración de programas, y ejecución de los procesos.
- Implementación: ¿cómo se haría? . . .a continuación

Tiempo compartido: implementación

- Implementar tiempo compartido requiere modificar la RSI de Reloj para:
  - Contabilizar el tick actual de reloj al proceso en ejecución
  - Cuando el proceso haya consumido su tiempo (`n.º ticks consumidos = tiempo asignado`), lo marcamos como preparado y planificamos.
- Para verlo en Linux es mejor usar una versión antigua (2.4.1) en el archivo *kernel/timer.c*, función `do_timer.c()`.

http://lxr.linux.no/#linux-old+v2.4.31/kernel/timer.c#L595

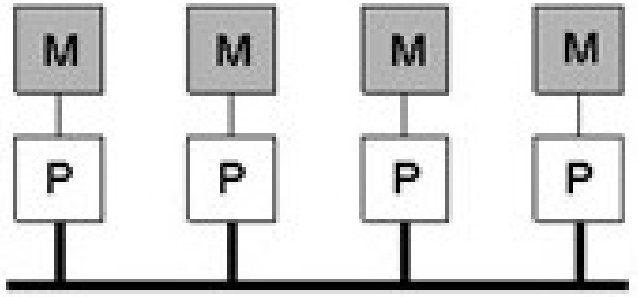
Sistemas de tiempo-real

- Un *sistema de tiempo-real* (RTS) es un sistema informático que reacciona con el entorno (responde a eventos físicos) a los que debe responder en un plazo determinado.
- Las tareas ejecutadas tienen asociado **tiempo límite** (*deadline*) en el que deben ejecutarse. Si no se satisface el *deadline* los resultados pueden ser inútiles e incluso catastróficos.
- Un RTS necesita el soporte de un **SO de tiempo-real** (RTOS), que debe garantizar la planificación de todas las tareas de forman que cumplan sus plazos.
- Prácticamente todos los SOs actuales incorporan alguno de los requisitos de tiempo-real.

Sistemas distribuidos

- Un **sistema distribuido** es un sistema multicomputador donde las diferentes CPUs no comparten memoria, ni reloj.
- El objetivo básico es compartir recursos (hard o soft).
- También permite un aumento de la fiabilidad del sistema: si una parte falla, el resto puede seguir con la ejecución.

P = procesador
M = memoria
— = Red comm.



Sistemas paralelos

- SOs para **sistemas multiprocesadores** – sistemas de computador con varios procesadores que comparten memoria y un reloj.
- Sustentan aplicaciones paralelas cuyo objetivo es obtener aumento de velocidad computacional.
- **Multiprocesamiento Simétrico** (*SMP*) - todos los procesadores pueden ejecutar tanto código del SO como de las aplicaciones.

P = procesador
M = memoria
—— = bus