Tema 1
Arquitectura y procesos

Departamento de Informática
Universidad de Extremadura

Tema 1: Arquitectura y procesos

1. ¿Por qué una disciplina de diseño de sistemas operativos?
2. Historia de MINIX
3. La arquitectura del sistema operativo
4. Los procesos
5. El planificador

1.1 ¿Por qué una disciplina de diseño de sistemas operativos?

- Las **llamadas al sistema** son bien comprendidos en su uso y función
- Es común la opinión de que un sistema operativo es construido por un pequeño equipo de **programadores privilegiados**, únicos conocedores de técnicas y recursos que son la clave de la construcción del sistema
- El enorme **desequilibrio** que se percibe en la literatura clásica sobre sistemas operativos entre teoría y práctica parece apoyar esta impresión y, lo que es más, ha contribuido a extenderla.
- El propósito de DSO es mostrar que el proceso de diseño e implementación de un sistema operativo puede ser -y debe ser- objeto de un tratamiento *organizado y sistemático* como cualquier otra disciplina científica
- En DSO examinaremos cómo se implementan los conceptos que ya se conocen en teoría: las interrupciones de los dispositivos, el planificador de procesos y los manejadores de dispositivos.

1.1 ¿Por qué una disciplina de diseño de sistemas operativos?

- A pesar de todo, no existe una metodología de construcción de un sistema operativo.
- El diseño de cualquier sistema complejo como un sistema operativo tiene un componente muy importante de **arte e ingenio**.
- ¿Cómo se debe estudiar una asignatura como DSO? No resulta sencillo. Es posible que la mejor aproximación sea un **caso de estudio**.
- Para que resulte útil en la docencia, este sistema operativo debe tener unos criterios de diseño bien definidos y debe ser lo suficientemente **sencillo** para que permita ser comprendido en su totalidad por una sola persona.

1.1 ¿Por qué una disciplina de diseño de sistemas operativos?

- El sistema operativo **MINIX** es un sistema creado para ser enseñado. Del mismo modo que Niklaus Wirth diseñó el lenguaje Pascal para mostrar lo en su opinión debía ser un lenguaje de programación, **Andrew Tanenbaum** ha escrito MINIX para revelar sus ideas acerca de cómo debe ser el diseño y la implementación de un sistema operativo.
- MINIX es el modelo que usa la asignatura. Lo estudiaremos en la clase de teoría y lo **modificaremos** en la clase de prácticas.

Andrew Tanenbaum

1.2 Historia de MINIX

- Los sistemas de tiempo compartido nacen en el **MIT**, el Instituto Tecnológico de Massachussets, a principios de los 60, a través de los proyectos BASIC, CTSS (Computer Timeshared System) y finalmente **MULTICS** (MULTiplexed Information and Computing Service).

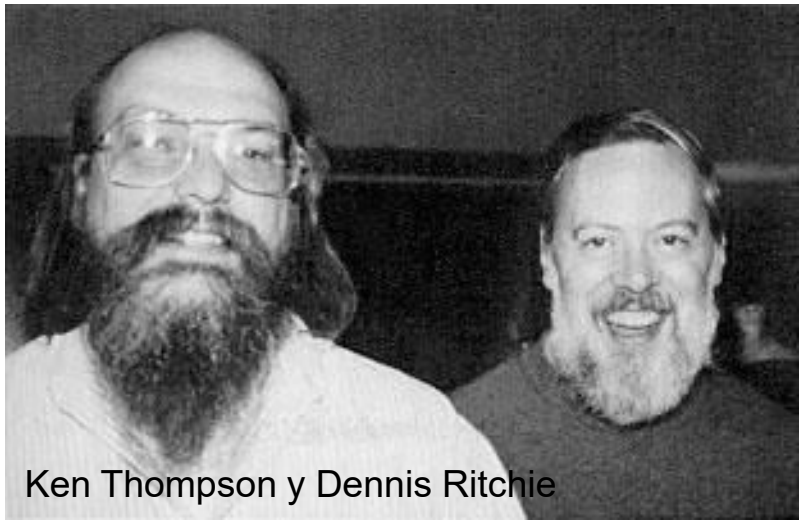
Digital PDP-7

- En 1969 **Ken Thompson**, un investigador de AT&T que había trabajado en el proyecto MULTICS, decidió escribir una **versión** de éste en miniatura para un computador Digital PDP-7.
- Su motivación: Necesitaba un sistema operativo adecuado para el juego Space Travel ([1]).

1.2 Historia de MINIX

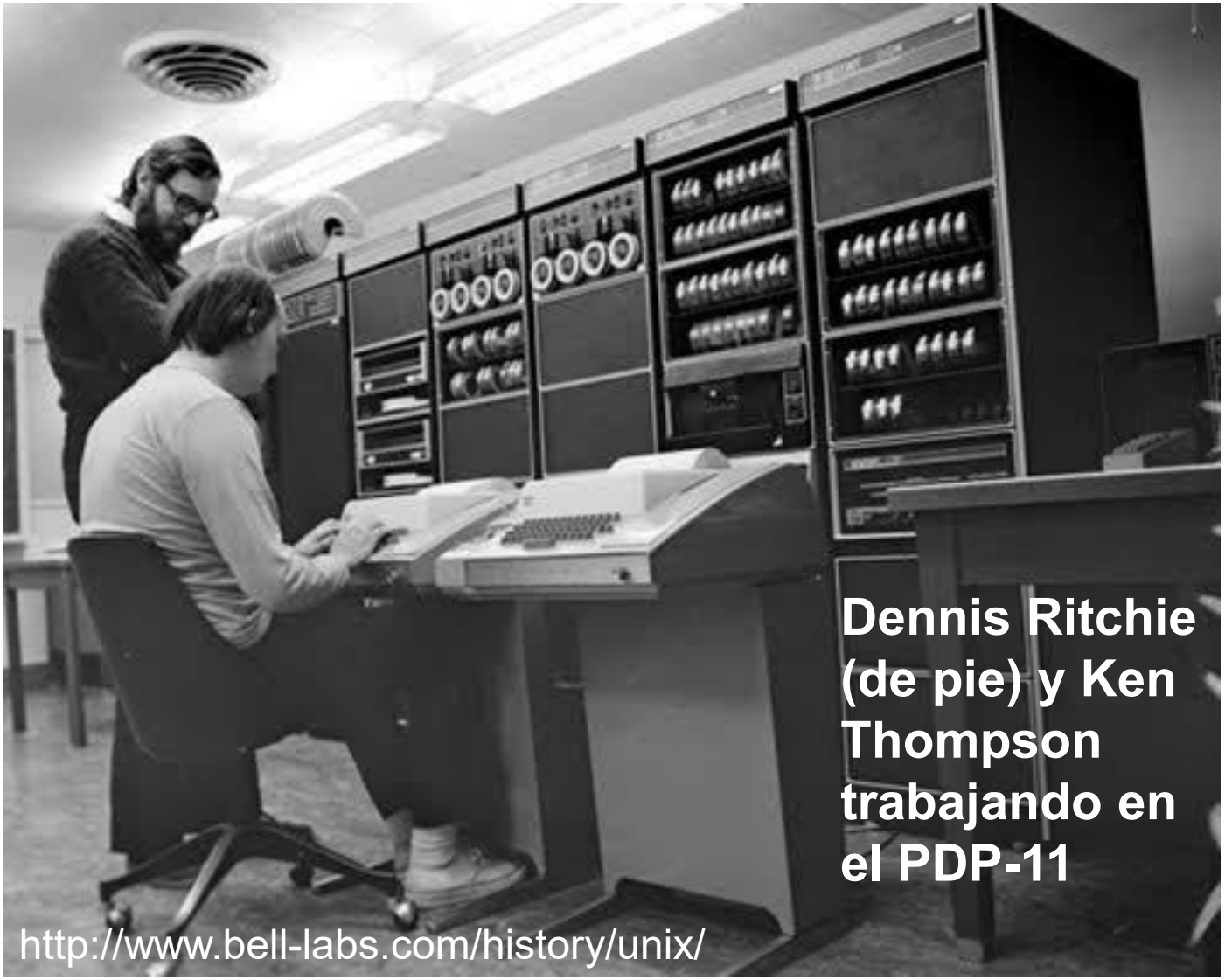
- *Brian Kernigham*, en tono jocoso, llamó al sistema de Ken Thompsom **UNICS** (UNiplexed Information and Computing Service) porque era monoproceso, frente al ineficiente pero funcional **MULTICS**, que era multiproceso.
- El éxito de UNICS para el PDP-7 llevó a que se proporcionase a Ken Thompson una máquina más moderna, el PDP-11, en la imagen

- UNICS estaba escrito en ensamblador y el trabajo de reescritura para el PDP-11 hizo pensar en que esta debía llevarse a cabo en un lenguaje de alto nivel.
- Por entonces, **Dennis Ritchie** había diseñado un lenguaje denominado **C**. Thompson y Ritchie reescribieron UNICS en C para el PDP-11 y decidieron que **UNIX** era un nombre más atractivo



Ken Thompson y Dennis Ritchie

1.2 Historia de MINIX



1.2 Historia de MINIX

- Un famoso artículo del año 1974 que describía UNIX atrajo la atención de las Universidades, que solicitaron de AT&T el código fuente de UNIX a fin de **estudiarlo y explicarlo en las aulas**
- Muy pronto, UNIX logró una amplia aceptación en la comunidad científica.
- AT&T se vio obligada a restringir su uso haciendo valer sus **derechos de propiedad**. A pesar de todo, el interés por UNIX siguió extendiéndose
- La Universidad de California, mediante un contrato con AT&T, portó la versión siete de UNIX, UNIX V7, a la arquitectura Digital VAX. Este UNIX fue bautizado como **UNIX BSD**.
- Las modificaciones de AT&T sobre UNIX V7 dieron lugar a otro sistema UNIX, **UNIX SYSTEM V**. Ambas versiones UNIX eran parcialmente incompatibles

1.2 Historia de MINIX

- La consecuencia del éxito comercial de UNIX fue la prohibición de que su código fuente no pudiera ser explicado y discutido en las aulas universitarias. El "saber hacer" en los sistemas operativos volvía a ser **monopolizado**, como en el pasado, por un reducido grupo de personas.
- Ante esta situación, un profesor de la Universidad de Vrije, en Amsterdam, **Andrew Tanenbaum**, se hizo con un ordenador personal **IBM PC** y decidió emular a Ken Thompson escribiendo un sistema operativo para la arquitectura PC.
- Este nuevo sistema operativo ofrecía la misma funcionalidad, es decir, las mismas llamadas al sistema que **UNIX V7** y era de un tamaño mucho más reducido, por lo que su autor lo bautizo como **MINIX** (Mini UNIX).
- El código de MINIX, como una vez lo fue UNIX, es público y ahora utilizado en las universidades en cursos de sistemas operativos.
- La versión 1.2 de MINIX fue publicada como un apéndice del libro “Operating Systems: Design and Implementation”. La versión 2.0 es conforme POSIX 1003.1a.

1.3 La arquitectura del sistema operativo

- ¿Cuál es la mejor arquitectura de sistema operativo?
- Este debate produjo un enconado encuentro en el grupo de noticias, comp.os.minix, entre Andy Tanenbaum y **Linus Torvalds**:

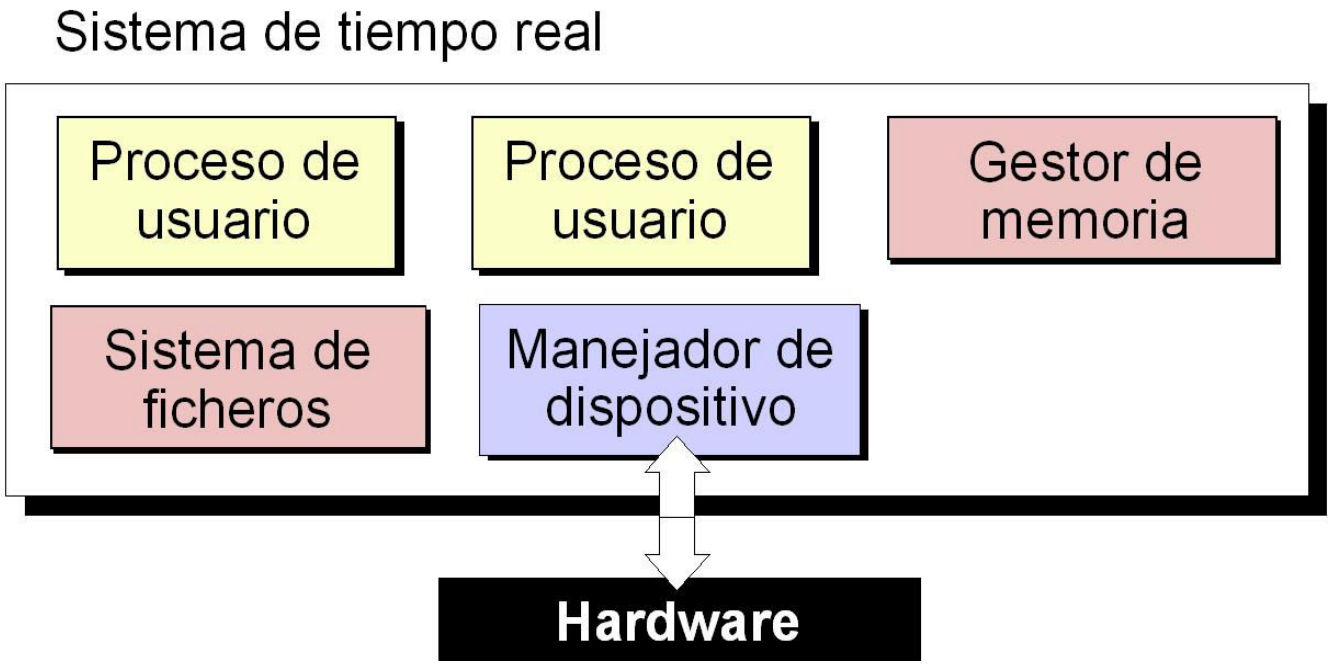
> From: ast@cs.vu.nl (Andy Tanenbaum)
> Newsgroups: comp.os.minix
> Subject: LINUX is obsolete
> Date: 29 Jan 92 12:12:50 GMT
> Organization: Fac. Wiskunde &h; Informatica, Vrije Universiteit, Amsterdam

- Las arquitecturas clásicas de sistema operativo son:
  - ✓La arquitectura plana
  - ✓La arquitectura monolítica: Linux y Windows
  - ✓La arquitectura microkernel: QNX y MINIX

1.3 La arquitectura del sistema operativo

Arquitectura plana

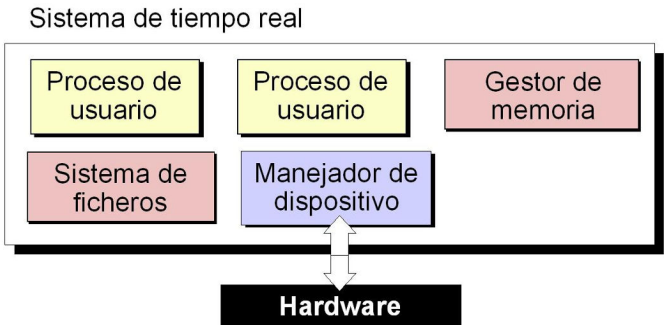
- Adoptada tradicionalmente por muchos sistemas de tiempo real.
- Integra todos los componentes de la aplicación en el mismo espacio de direccionamiento que el sistema operativo.



1.3 La arquitectura del sistema operativo

Arquitectura plana

- Suele emplearse en sistemas empotrados pequeños o procesadores sin facilidades de protección de memoria, como microcontroladores o procesadores digitales de señal (DSPs).

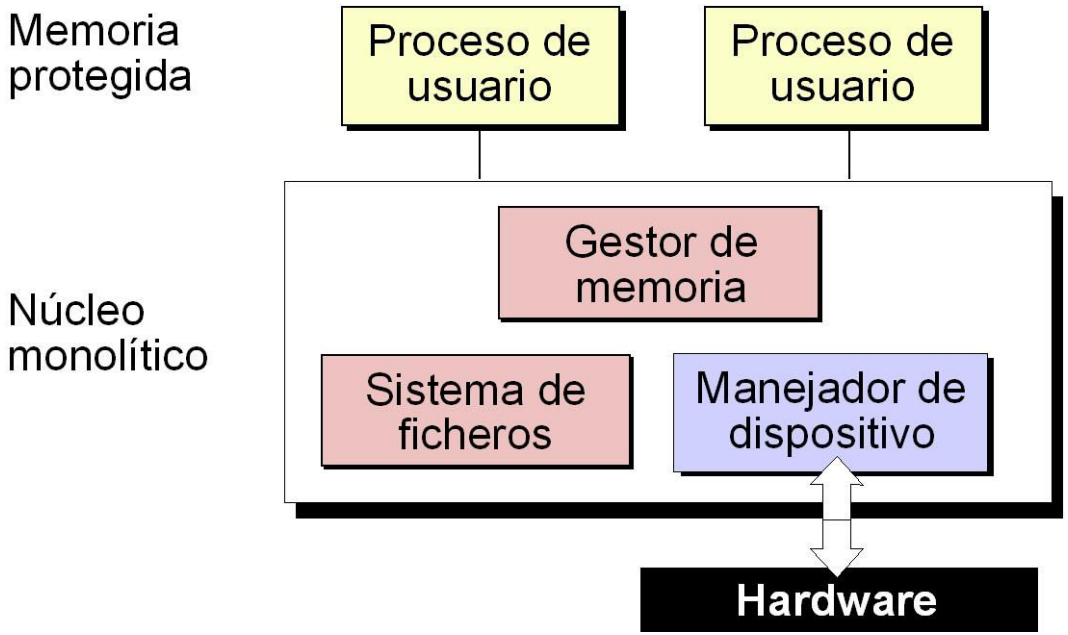


- La fiabilidad de todo el conjunto depende de cada nuevo componente. La carencia de protección conlleva el que si un proceso falla, puede escribir sobre código o datos del núcleo, de modo que todo el sistema se corrompe. Además, en cuanto el sistema crece, la probabilidad del fallo se multiplica.
- Cada fallo es difícil de localizar. Incluso las mejores herramientas pueden ser incapaces de aislarlo y es preciso que una persona conozca bien todo el sistema completo.
- Las limitaciones de la arquitectura plana, por lo tanto, son patentes.

1.3 La arquitectura del sistema operativo

Arquitectura monolítica

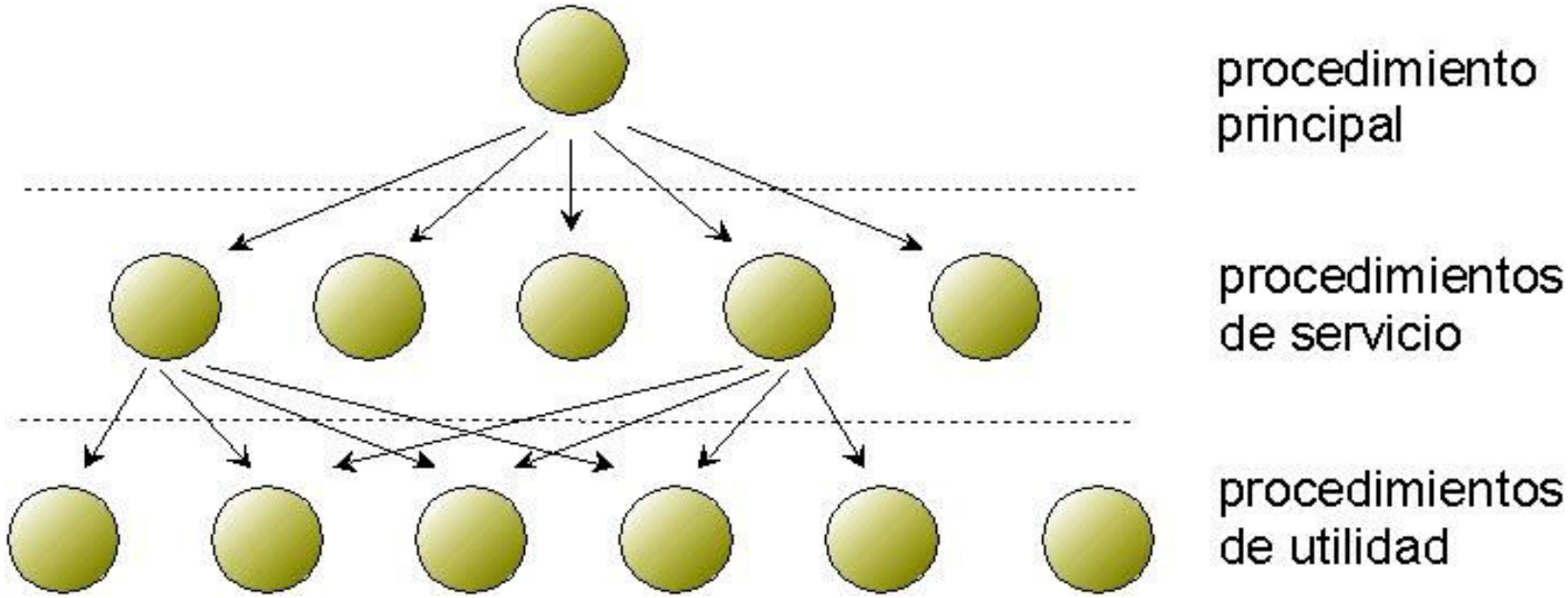
- Es un intento de paliar los problemas de la arquitectura plana
- Su aportación estriba en que los procesos de usuario ejecutan en espacios de direccionamiento diferentes al del sistema operativo



1.3 La arquitectura del sistema operativo

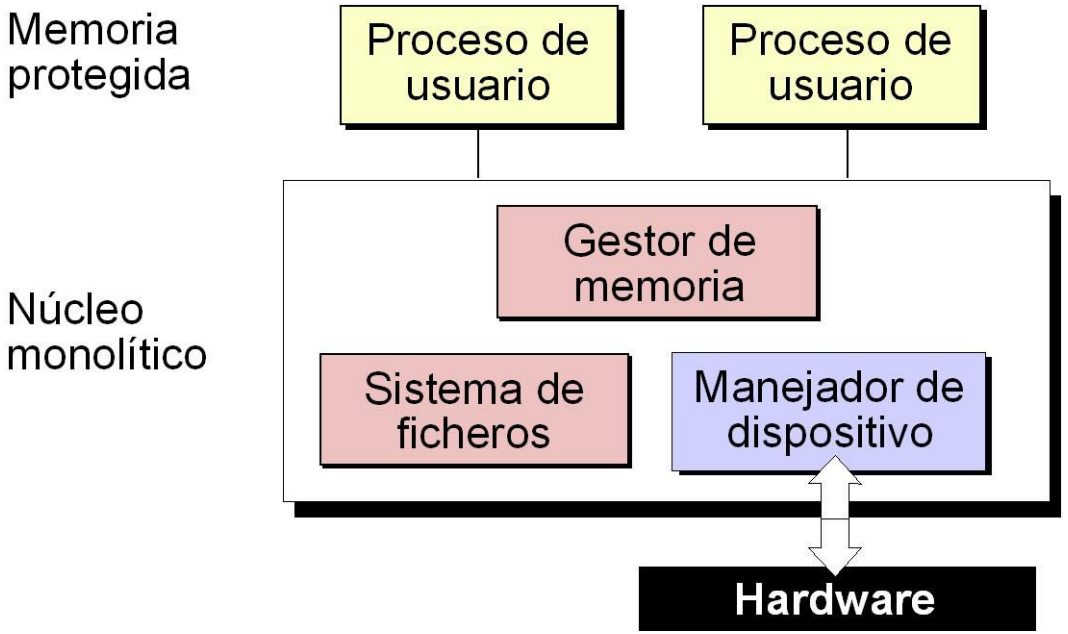
Arquitectura monolítica

- Las implementaciones de UNIX han respondido tradicionalmente este diseño
- El sistema operativo es una colección de procedimientos compilados en un único programa objeto donde cada procedimiento es visible a todos los demás.



1.3 La arquitectura del sistema operativo

Arquitectura monolítica



http://www.taringa.net/post/humor/7968750/Bill-Gates---Pantallazo-azul-presentacion-en-vivo.html

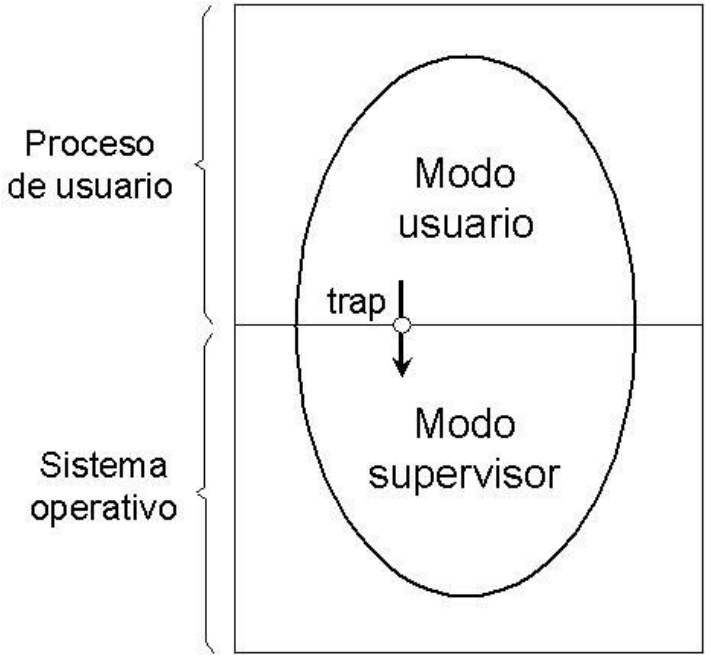
- Hemos aislado al sistema de los errores de los procesos de usuario, pero nuevos **dispositivos** aparecen en el mercado continuamente y es preciso escribir **manejadores** para soportarlos. De nuevo el sistema crece y la probabilidad del **fallo** aumenta.

1.3 La arquitectura del sistema operativo

Arquitectura monolítica

- El programa de usuario lleva a cabo las llamadas al sistema mediante instrucciones de interrupción software o *trap*.
- ¿Por qué es necesaria una instrucción de interrupción software y no una instrucción de salto? Debido a que esta instrucción de interrupción software conmuta el procesador de *modo usuario* a *modo supervisor*.

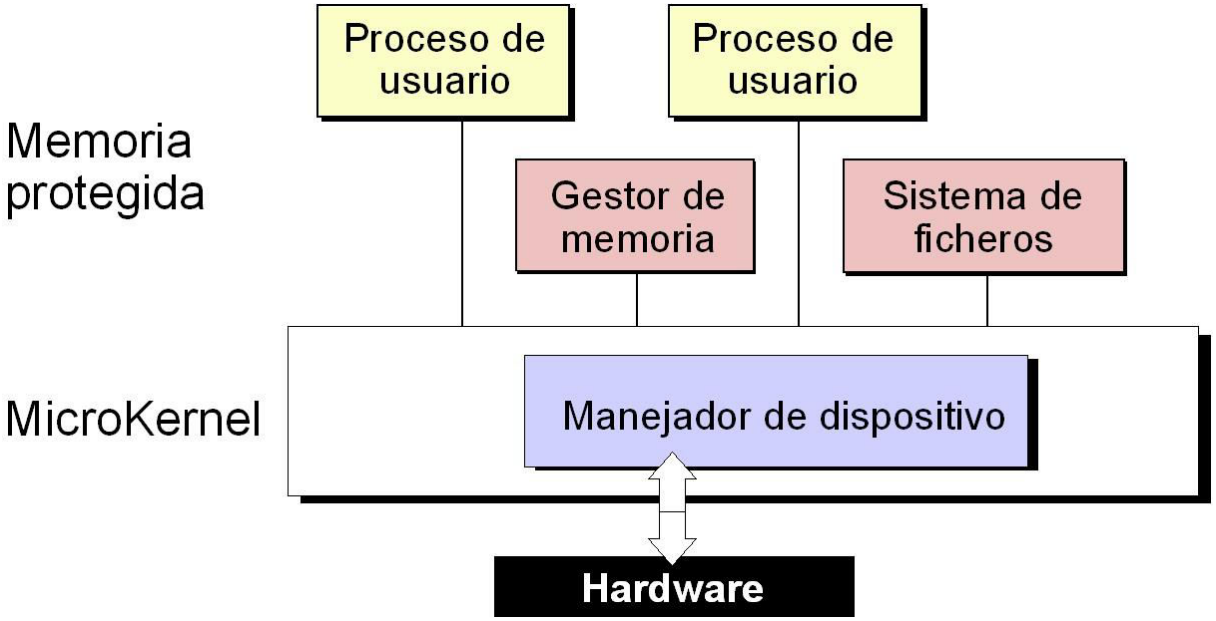
- Sólo en modo supervisor se permite al procesador ejecutar **instrucciones privilegiadas**
  1. Accesos a las posiciones de memoria asignadas a los adaptadores de dispositivo
  2. Copia de datos entre espacios de direccionamiento diferentes.



1.3 La arquitectura del sistema operativo

Arquitectura micronúcleo: MINIX

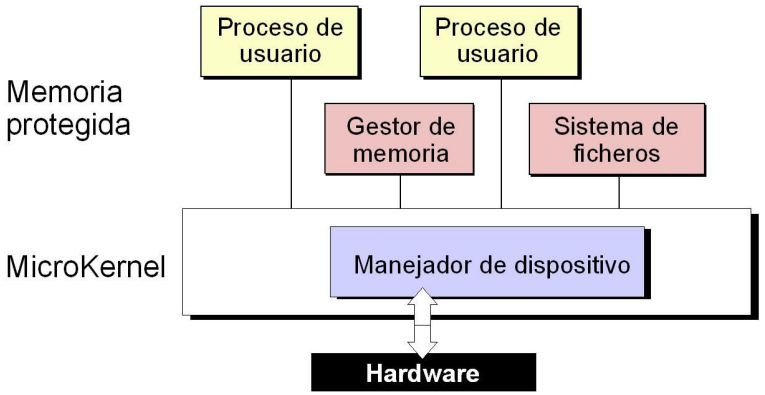
- Una tendencia en los sistemas operativos modernos es disponer de un núcleo lo más pequeño posible, procurando que algunas de sus **funciones** -cuantas más mejor- sean **desplazadas fuera** del mismo y sean implementadas como programas de usuario.
- Este nuevo núcleo reducido se viene denominando ***microkernel***.



1.3 La arquitectura del sistema operativo

Arquitectura micronúcleo: MINIX

- MINIX hace uso del paradigma **cliente-servidor**
- El sistema de ficheros y el gestor de memoria se programan como **servidores**

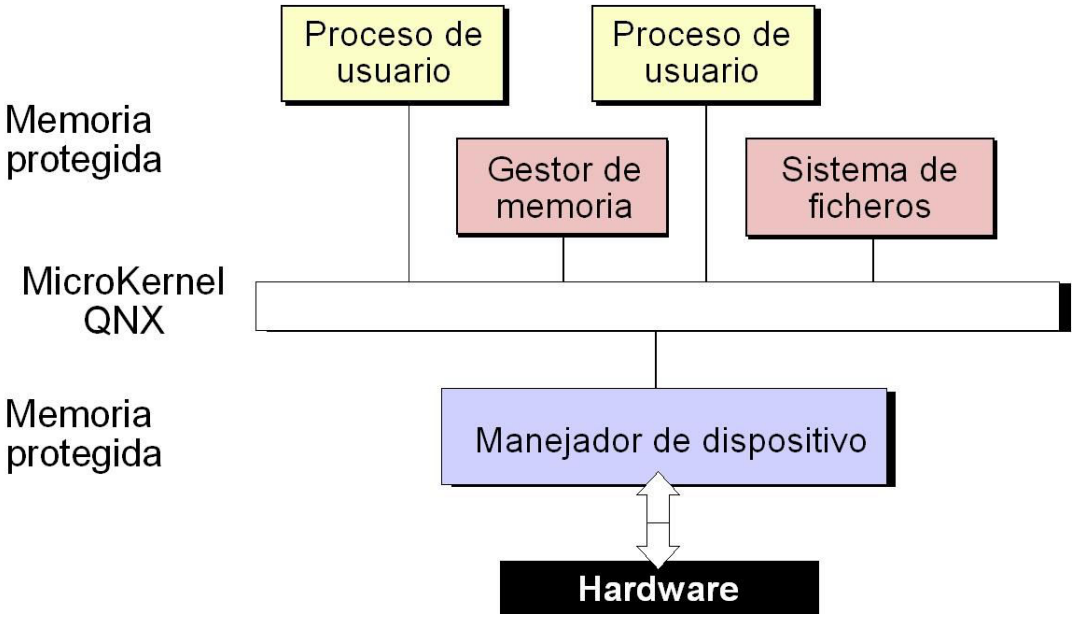


- Los manejadores de los dispositivos también toman la forma de servidores; no obstante, se mantienen integrados en el núcleo para facilitar su acceso a los dispositivos físicos.
- La labor del **microkernel** es soportar el **paso de mensajes** entre clientes y servidores
- Una de las ventajas de un sistema microkernel es que las partes se dividen por fronteras bien definidas, por lo que resultan más comprensibles y manejables.
- Una segunda ventaja es su idoneidad para construir sistemas distribuidos, donde clientes y servidores ejecutan en máquinas diferentes y se intercambian los mensajes a través de mecanismos de red.

1.3 La arquitectura del sistema operativo

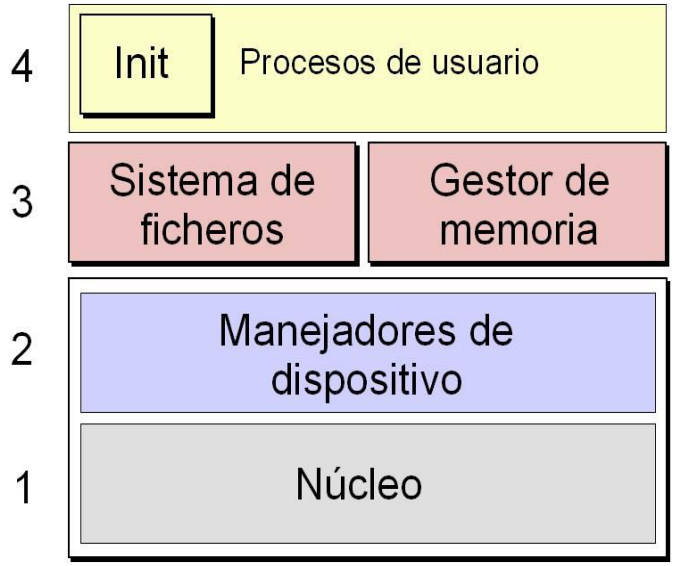
Arquitectura micronúcleo: QNX

- QNX, como MINIX, construye los manejadores de dispositivo como procesos, pero con la ventaja adicional de que los desplaza fuera del núcleo en espacios de direccionamiento propios. Este es el diseño más fiable de todos por que es el que proporciona mayor protección
- La versión 3.0 de MINIX incorpora esta mejora



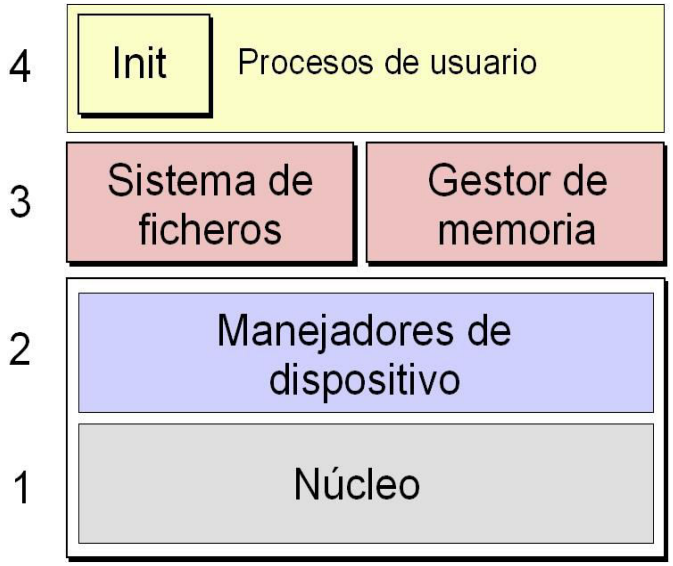
1.4 Los procesos

A diferencia de un sistema monolítico como Linux, MINIX se estructura en cuatro niveles



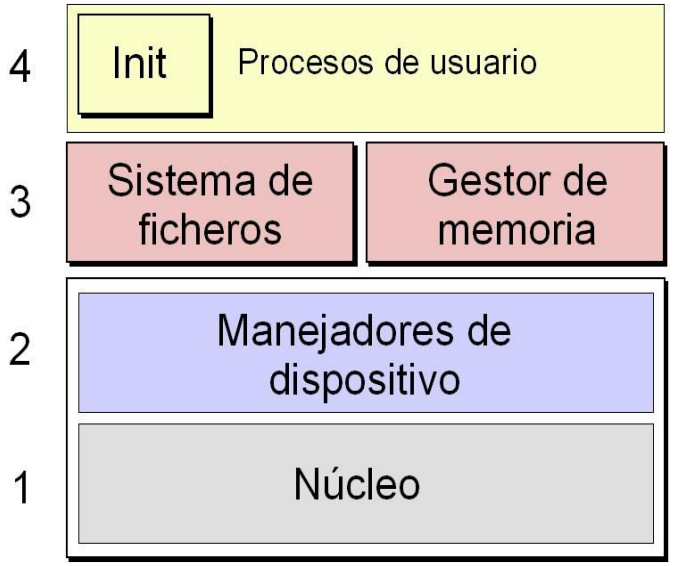
- El **nivel uno**, "el núcleo“. Tiene dos funciones fundamentales.
  - ✓ La primera es proporcionar a los niveles superiores el *concepto de proceso*. Se encarga de gestionar *interrupciones* y *traps* salvando el estado del proceso en ejecución en su descriptor.
  - ✓ La segunda es soportar el mecanismo de comunicación entre procesos a través de mensajes.
- Sólo una mínima parte del nivel uno está escrita en ensamblador. El resto y los demás niveles están escritos en *lenguaje C*.

1.4 Los procesos



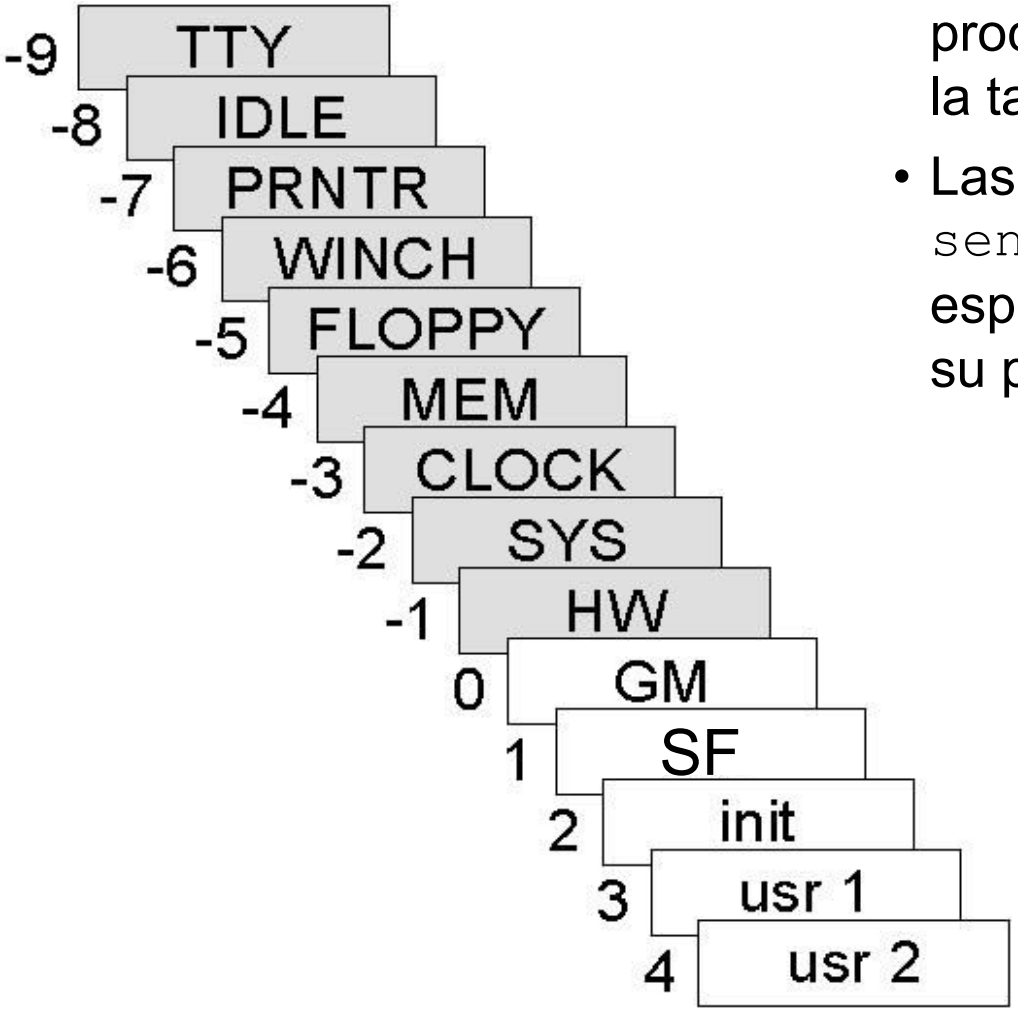
- El **nivel dos** contiene los manejadores de los dispositivos de entrada-salida (teclado, disco, etc) o *tareas*.
- El código del nivel uno y las tareas se compilan conjuntamente en MINIX en un único *espacio de direccionamiento*.
- A pesar de compartir un mismo espacio de direccionamiento, las tareas se comportan como procesos independientes y se comunican mediante el mecanismo de paso mensajes del sistema, igual que lo hacen los servidores y los procesos de usuario.

1.4 Los procesos

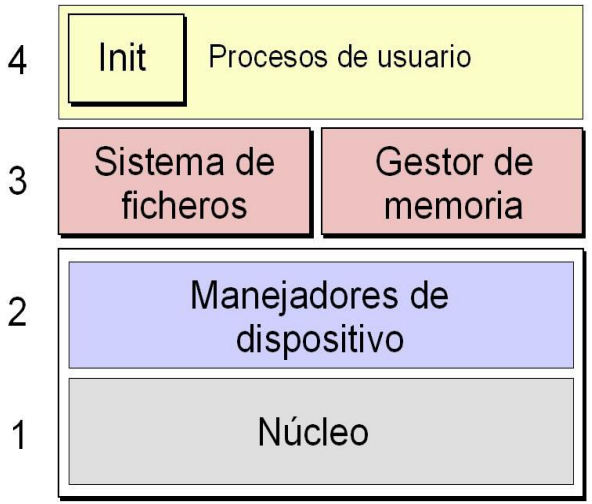


- El nivel tres contiene tres procesos denominados servidores. Son el gestor de memoria, que sirve las llamadas al sistema como *fork*, *exec* y *brk*, el sistema de ficheros, que sirve llamadas al sistema como *open*, *read*, etc, y, finalmente el servidor de red, que implementa la pila TCP/IP.
- El sistema de ficheros de MINIX ha sido diseñado como un servidor de ficheros, de manera que es sencillo portarlo a una máquina diferente como servidor remoto.
- Lo mismo ocurre con el gestor de memoria.
- El nivel cuatro alberga los procesos de usuario, entre ellos *init*.

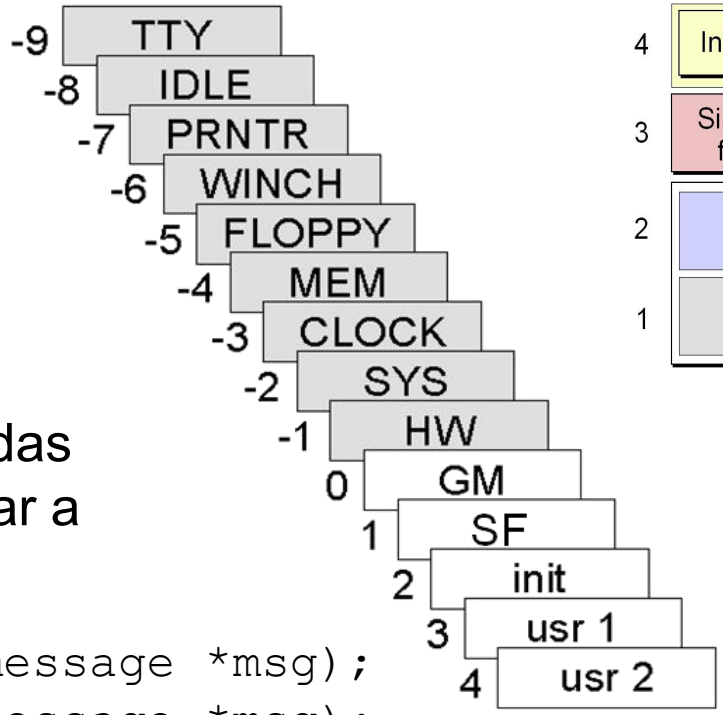
1.4 Los procesos

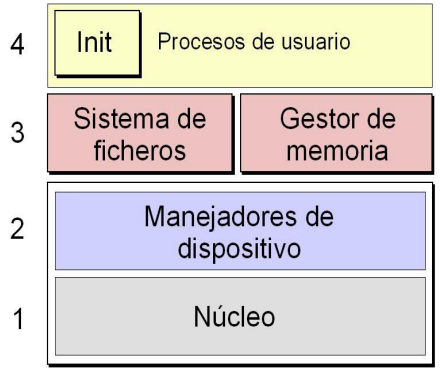


- Cada tarea o proceso en un sistema MINIX tiene asociado un *número de proceso*. El número de proceso es el índice del proceso en la tabla de descriptores de proceso.
- Las primitivas de paso de mensajes `send`, `receive` y `sendrec` especifican números de proceso en su parámetros fuente/destino



1.4 Los procesos



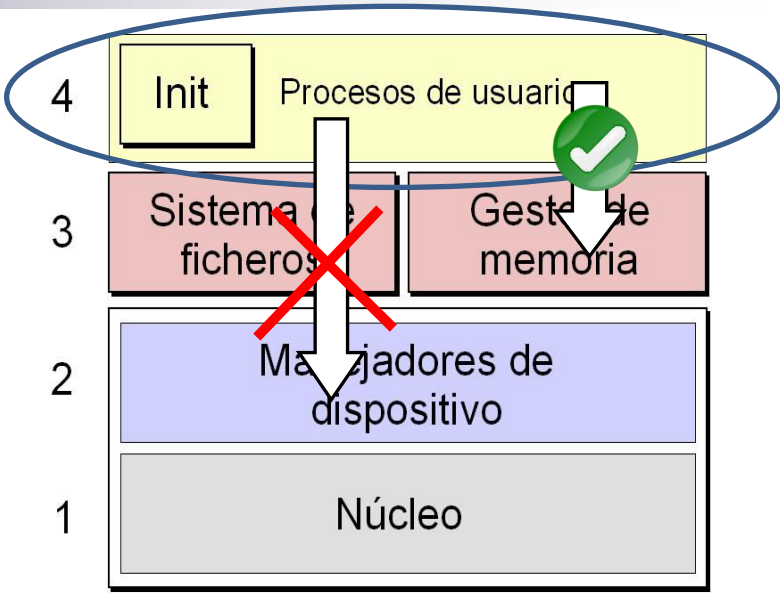


- El núcleo de MINIX realmente soporta únicamente tres llamadas al sistema, encargadas de llevar a cabo el paso de mensajes:

```
int send   (int dst, message *msg);
int receive(int src, message *msg);
int sendrec(int dst, message *msg);
```

- A los procesos de usuario sólo les está permitido invocar `sendrec`. El autor de MINIX considera que el uso de `send` y `receive` en los programas de usuario les hace difícil de entender y mantener, dando lugar a lo que denomina **programación *spaghetti***.
- Tanenbaum considera tan nocivo su uso en comunicación de procesos como la sentencia goto en programación estructurada.

1.4 Los procesos



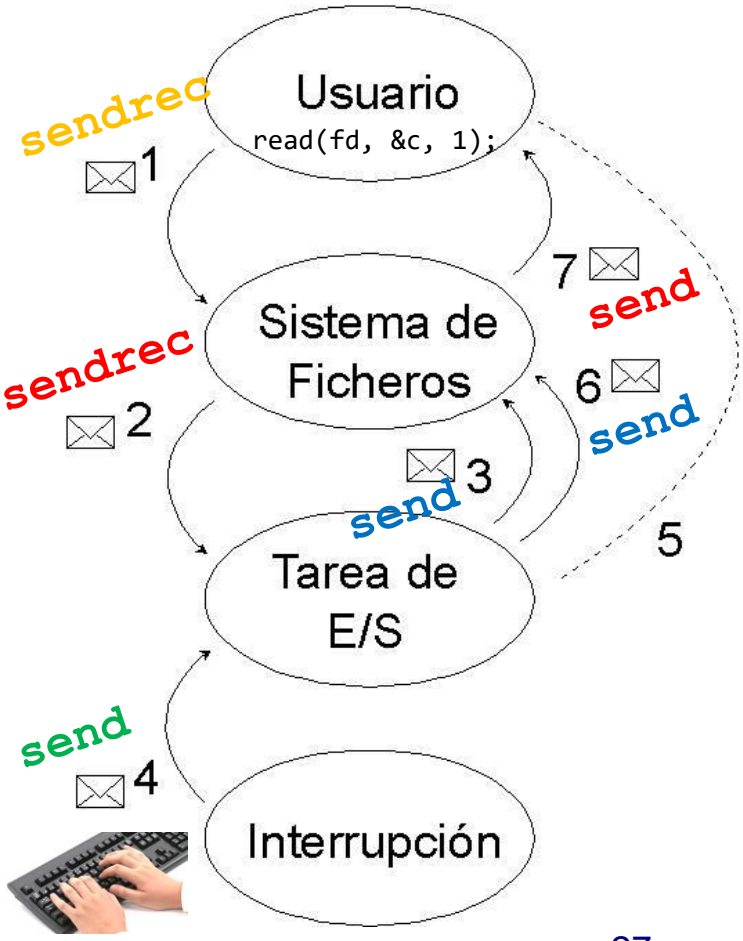
- Un proceso de usuario sólo puede usar `sendrec` para solicitar los servicios del gestor de memoria o el sistema de ficheros y ...
- ... no puede comunicarse directamente con las tareas de entrada-salida.

```
int send    (int dst, message *msg);
int receive (int src, message *msg);
int sendrec (int dst, message *msg);
```

1.4 Los procesos

Mensajes involucrados en la función POSIX **`read`**: lectura de un carácter

1. El PU invoca la función `read`, y esta `sendrec`, que envía un mensaje (1) al SF y bloquea al PU
2. El SF comprueba que los parámetros, permisos, etc. son correctos y también invoca `sendrec`, que envía un mensaje (2) a la tarea del terminal y bloquea al SF
3. Si **el carácter no está disponible** en el buffer de la tarea del terminal, esta envía un mensaje al SF (3) para desbloquearlo para atender a otros procesos. Note que el PU continúa bloqueado en `sendrec`.
4. Eventualmente, el dispositivo proporciona el carácter e interrumpe. La rutina de interrupción envía un mensaje (4) a la tarea.
5. La tarea recoge el carácter de la controladora y lo copia (5) a la dirección &c en el PU.
6. La tarea envía un mensaje (6) al sistema de ficheros comunicándole que puede despertar al PU
7. El SF envía un mensaje (7) al PU con el retorno de `read`. `Sendrec` retorna y `read` retorna.

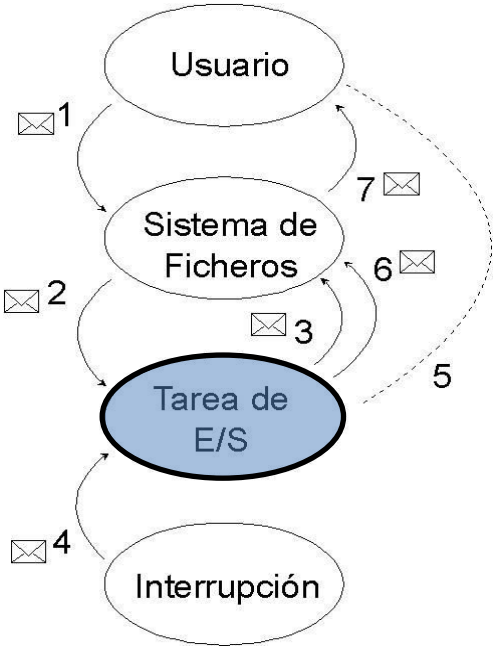


1.4 Los procesos

```
cualquierTarea()
{
  mensaje mens;
  int     r, emisor;

  inicializacion();
  while(TRUE) {
    receive(ANY, &mens);
    emisor = mens.fuente;
    switch(mens.tipo) {
      case READ:     r = do_read();      break;
      case WRITE:    r = do_write();     break;
      case INTERRUP: r = do_interrup();  break;
      case OTRO:     r = do_otro();      break;
      default:       r = ERROR;
    }
    mens.tipo = TASK_REPLAY;
    mens.REP_STATUS = r;
    send(emisor, &mens);
  }
}
```

El manejador de dispositivo o **tarea** es un procesos de pleno derecho, con su contador de programa, descriptor de proceso, pila, etc.



1.4 Los procesos

```
cualquierTarea()
{
  mensaje mens;
  int     r, emisor;

  inicializacion();

  while(TRUE) {
    receive(ANY, &mens);

    emisor = mens.fuente;
    switch(mens.tipo) {
      case READ:     r = do_read();      break;
      case WRITE:    r = do_write();     break;
      case INTERRUP: r = do_interrup();  break;
      case OTRO:     r = do_otro();      break;
      default:       r = ERROR;
    }

    mens.tipo = TASK_REPLAY;
    mens.REP_STATUS = r;
    send(emisor, &mens);
  }
}
```

Inicialización

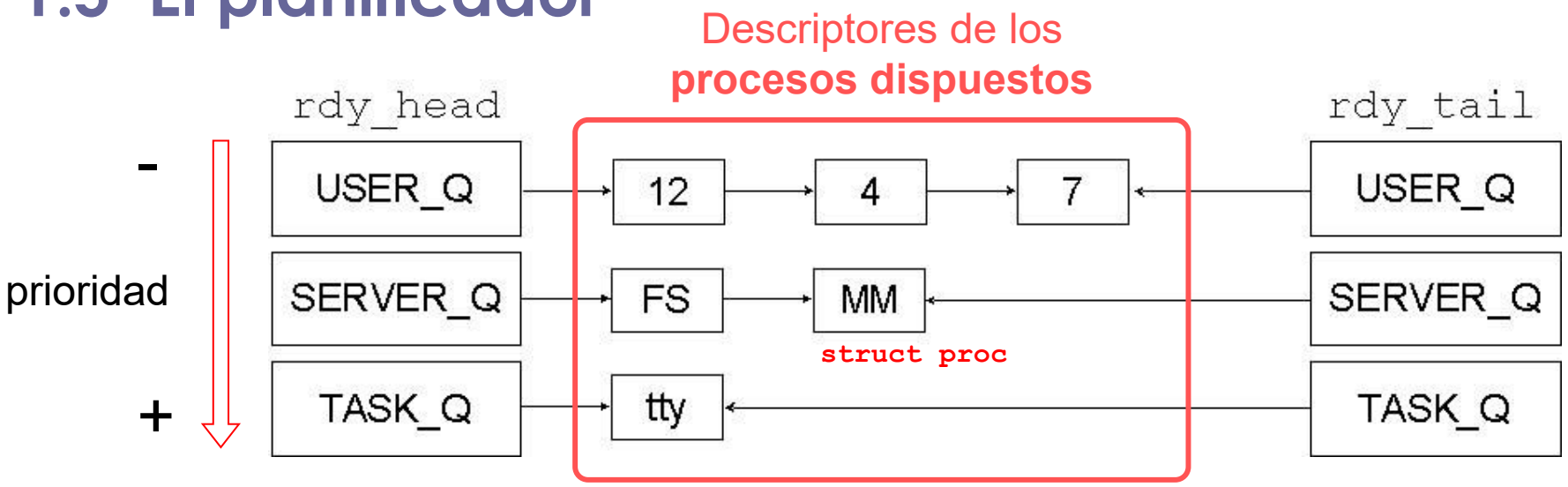
Bucle de servicio

El manejador de dispositivo es un **servidor**

Un servidor consta de inicialización y bucle de servicio

Cada iteración del bucle de servicio consta de recepción, despacho y réplica

1.5 El planificador




```
struct proc *rdy_head[NQ];
struct proc *rdy_tail[NQ];
```

- En un instante dado, un número determinado de procesos están **dispuestos** para ejecutar
- El resto están suspendidos en el envío o recepción de un mensaje
- En un sistema con un solo procesador, de todos los procesos dispuestos para ejecutar sólo uno puede hacerlo. El **planificador** es un algoritmo que decide cuál de ellos

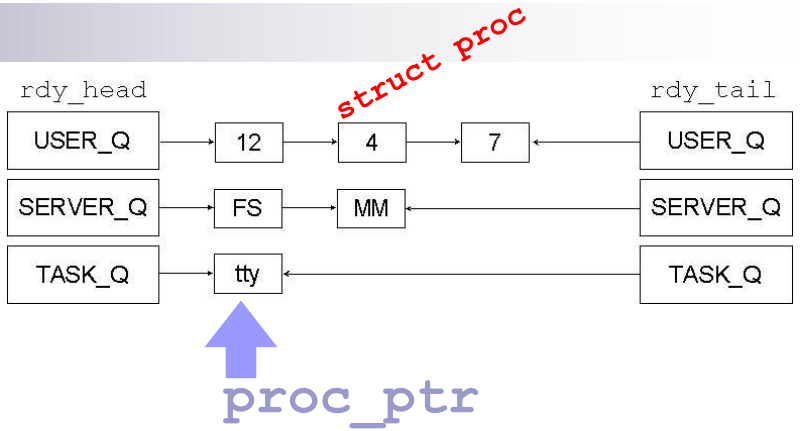
1.5 El planificador

- Cuando ningún proceso está dispuesto, es decir, no hay ningún proceso en las colas, se elige la tarea ociosa, **IDLE**.
- IDLE se define en el núcleo (fichero `mpx88.x`) a diferencia del resto de las tareas, que se definen en ficheros propios en C, como *tty.c*.

```
_idle_task:      ! executed when there is no work
    jmp _idle_task
```

- Aunque simple, IDLE se comporta como el resto de las tareas, ya que tiene su propia pila y su propio código y dispone de su propio descriptor de proceso.
- IDLE no invoca a ninguna rutina, de modo que su **pila** es más pequeña que el resto. Tiene tan sólo 20 octetos, el tamaño necesario y suficiente para soportar la interrupción que reactive el sistema.

1.5 El planificador



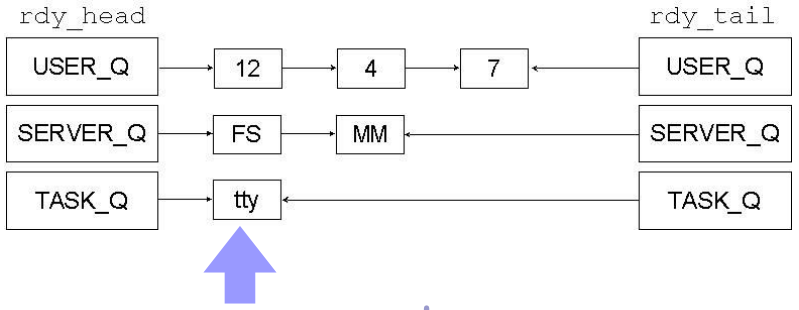

```
void pick_proc()
{
  struct proc *rp;

  if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
    proc_ptr = rp;
    return;
  }
  if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
    proc_ptr = rp;
    return;
  }
  if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
    proc_ptr = rp;
    bill_ptr = rp;
    return;
  }
  bill_ptr = proc_ptr
           = proc_addr(IDLE);
  return;
}
```

El método **pick_proc** del planificador elige el proceso al que se otorga el procesador

proc_ptr es una variable global del núcleo que apunta al proceso en ejecución

1.5 El planificador




```
void pick_proc()
{
  struct proc *rp;
  if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
    proc_ptr = rp;
    return;
  }
  if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
    proc_ptr = rp;
    return;
  }
  if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
    proc_ptr = rp;
    bill_ptr = rp;
    return;
  }
  bill_ptr = proc_ptr
           = proc_addr(IDLE);
}
```

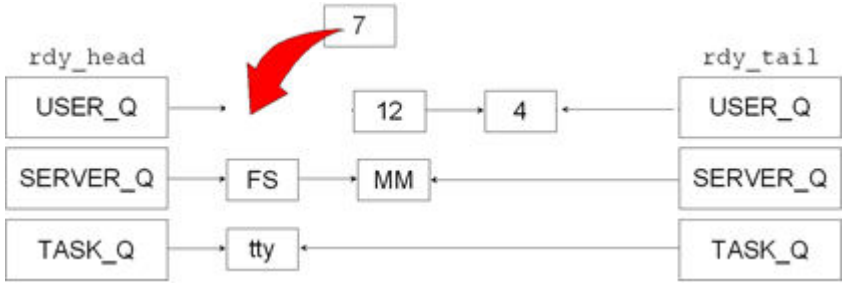
bill_ptr es una variable global del núcleo que apunta al proceso en facturación

Sólo se factura el uso del procesador a los procesos de usuario y a IDLE

1.5 El planificador

```
void ready(struct proc *rp)
{
  if (istaskp(rp)) {
    if (rdy_head[TASK_Q] != NIL_PROC)
      rdy_tail[TASK_Q]->p_nextready = rp;
    else {
      proc_ptr =
      rdy_head[TASK_Q] = rp;
    }
    rdy_tail[TASK_Q] = rp;
    rp->p_nextready = NIL_PROC;
    return;
  }
  if (!isuserp(rp)) {
    if (rdy_head[SERVER_Q] != NIL_PROC)
      rdy_tail[SERVER_Q]->p_nextready = rp;
    else
      rdy_head[SERVER_Q] = rp;
    rdy_tail[SERVER_Q] = rp;
    rp->p_nextready = NIL_PROC;
    return;
  }
  if (rdy_head[USER_Q] == NIL_PROC)
     rdy_tail[USER_Q] = rp;
  rp->p_nextready = rdy_head[USER_Q];
  rdy_head[USER_Q] = rp;
  return;
}
```

El proceso 7 ha terminado una comunicación y queda dispuesto. Para ello se invoca a `ready`



El método **`ready`** del planificador introduce un proceso en la cola

Los procesos de usuario entran por la cabeza

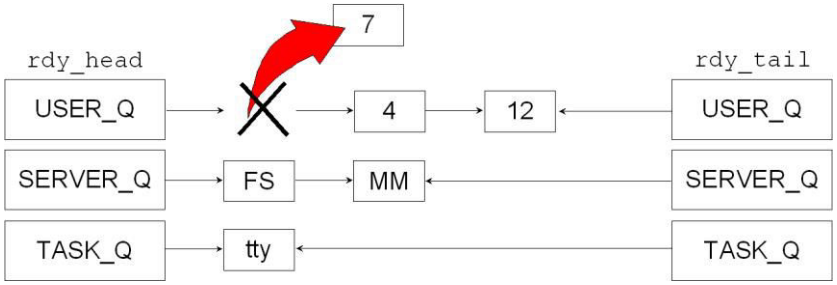
Servidores y tareas por la cola

```
void unready(struct proc *rp)
{
  struct proc  *xp;
  struct proc **qtail;

  if (istaskp(rp)) {
    xp = rdy_head[TASK_Q];
    if(xp == rp) {
      rdy_head[TASK_Q] = xp->p_nextready;
      pick_proc();
      return;
    }
    qtail = &rdy_tail[TASK_Q];
  }

  else if(!isuserp(rp)) {
    xp = rdy_head[SERVER_Q];
    if (xp == rp) {
      rdy_head[SERVER_Q] = xp->p_nextready;
      pick_proc();
      return;
    }
    qtail = &rdy_tail[SERVER_Q];

  } else  {
    xp = rdy_head[USER_Q];
    if (xp == rp) {
      rdy_head[USER_Q] = xp->p_nextready;
      pick_proc();
      return;
    }
    qtail = &rdy_tail[USER_Q];
  }
  /* Search body of queue.  A process can be made unready even if it is
   * not running by being sent a signal that kills it. */
  while (xp->p_nextready != rp)
    xp = xp->p_nextready;
  xp->p_nextready = xp->p_nextready->p_nextready;
  if (*qtail == rp) *qtail = xp;
}
```

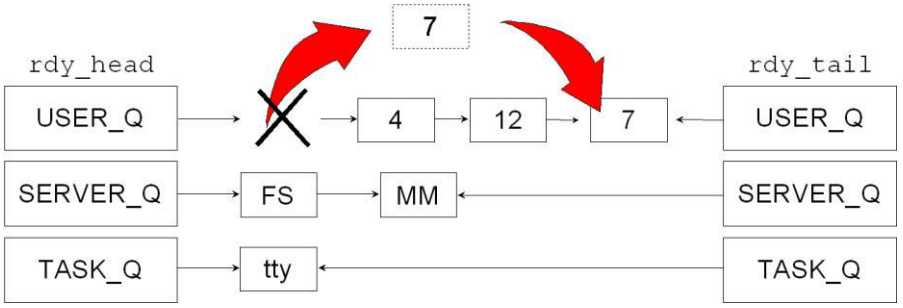


El proceso 7 queda bloqueado en una comunicación. Para ello el núcleo invoca `unready`

El método **`unready`** del planificador extrae un proceso de la cola

1.5 El planificador

El método **sched** pasa el proceso de usuario activo al último lugar de la cola




```
void sched()
{
  rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
  rdy_tail[USER_Q] = rdy_head[USER_Q];
  rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
  rdy_tail[USER_Q]->p_nextready = NIL_PROC;
  pick_proc();
}
```

- `Sched` es invocado en la tarea del reloj cuando el proceso de usuario en curso ha completado su quantum (Round-Robin).
- Sólo se aplica a la cola USER_Q: Las tareas y los servidores una vez que reciben un mensaje lo despachan rápidamente y se bloquean (Su planificación no es round-robin, sino FIFO). Se confía en que operan correctamente y no acaparan el procesador.

Referencias

[1] http://sites.fas.harvard.edu/~lib215/reference/history/spacetravel.html