

Chat P07:

Que cuando hay mas de un cliente, se envien los mensajes, pero que no se quede pillado en el socket del otro.

El modo sincrónico no vale (no sabes quien va a escribir antes). Pero quieres interactuar con los otros clientes así que uno interactivo no va.

Aunque los threads suenan bien, cuando se hace una lectura, se bloquea el proceso, como si fuera concurrente.



Técnicas avanzadas de Programación en Redes


Programación en Red / Entornos Distribuidos

Curso 2024/2025
Universidad San Pablo-CEU
Escuela Politécnica Superior
Campus de Montepríncipe

1

El read(), al ser una llamada del sistema, se hacen toda una fiesta de i-nodos, sincronizarlo,... El que recibe el texto del fichero es el kernel, y despues meterlo en el buffer para pasarlo a la memoria del proceso.

El write() mas de lo mismo



Entrada/Salida avanzada

- » En una **operación de entrada**, como por ejemplo una **lectura** (`read()`), hay dos fases distintas a tener en cuenta:
 - **esperar** a que los **datos** estén listos, y
 - **copiar** los **datos** desde el kernel a la memoria del proceso
- » Igualmente, en una **operación de salida** (`write()`) hay también dos fases distintas a tener en cuenta:
 - **esperar** a que el **destino** de los datos, esté disponible para aceptarlos, y
 - **escribir** los **datos** de la memoria del proceso en la memoria del kernel

Tiempos negables de copia por las DMA de los ordenadores

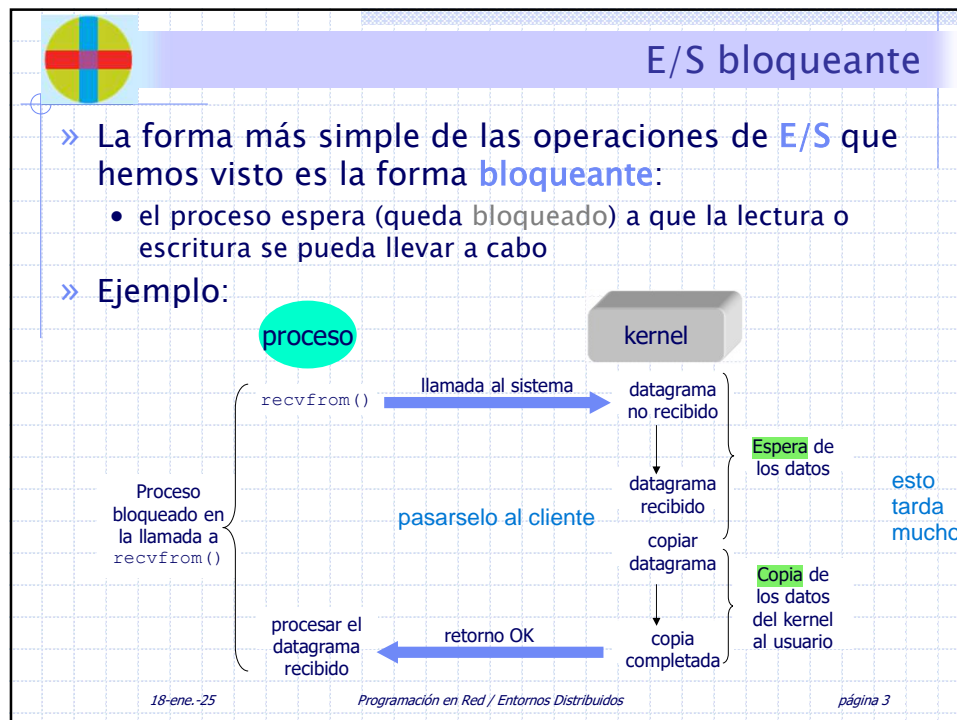
Lo que tarda mas es la espera, porque las copias tardan lo que tarda el reloj de la cpu (Ghz => ns)

La espera depende del disco duro (ms) y de la persona escribiendo y otros factores

En el sincrónico hay que atacar las esperas

La idea es intentar hacer otra tarea mientras se esta esperando a que el otro escriba

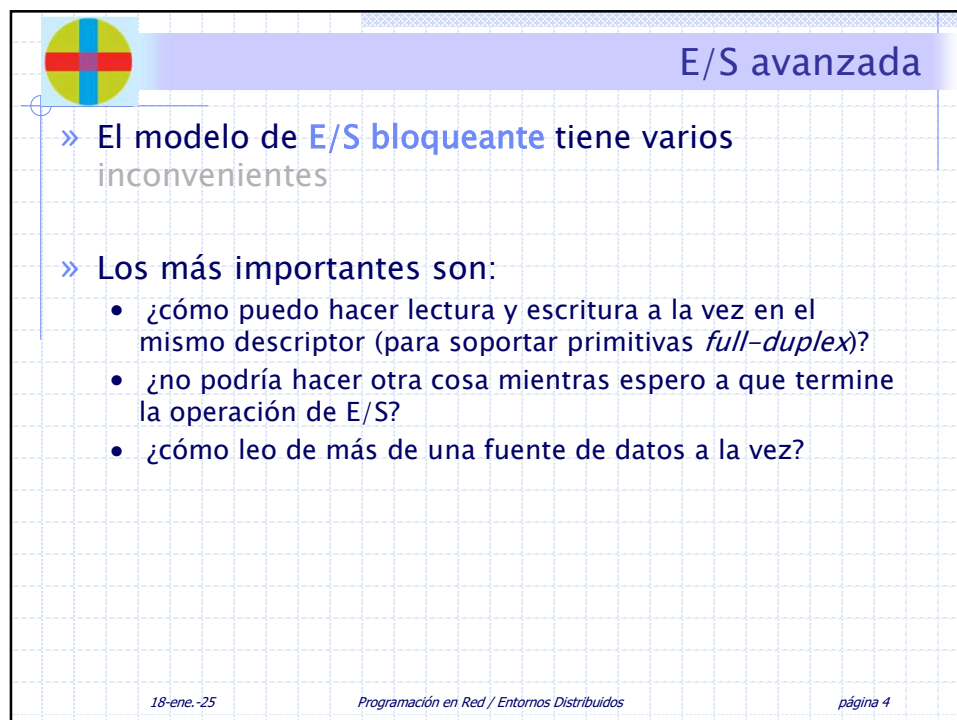
2



Mientras esto está pasando el proceso está dormido, que gasta menos, pero necesitamos el tiempo

esto tarda mucho

3



Como leo de dos sitios a la vez, para que me despierte al primero que llegue.

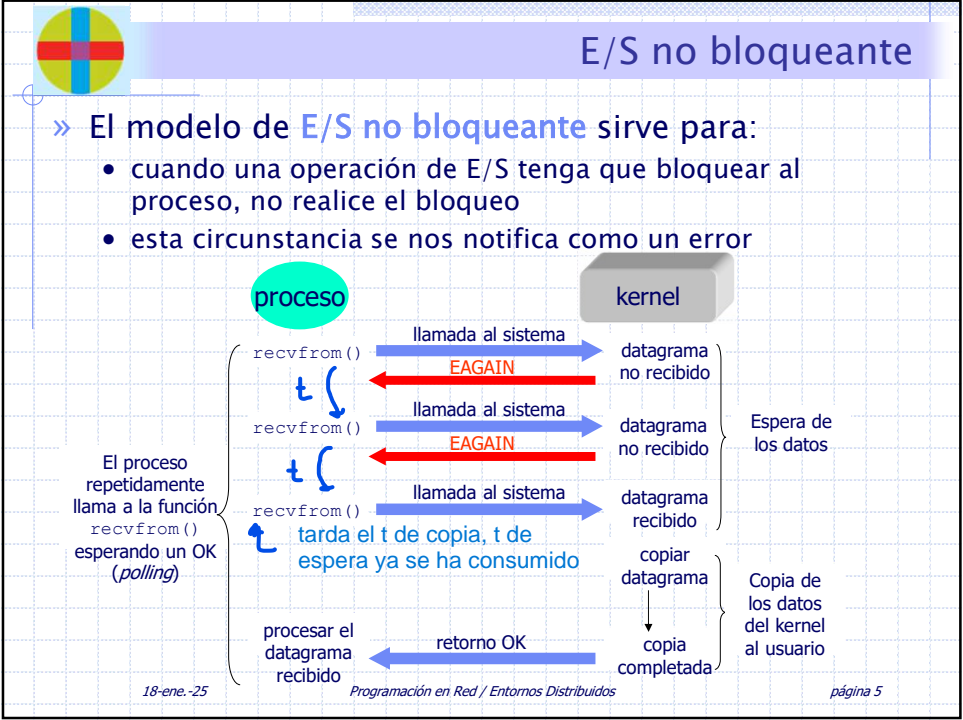
Si le haces un read() a uno y no da el mensaje todavía pues la has jodido y te bloqueas

4

Idealmente el tiempo ideal es el tiempo de espera, llenando del primer intento hasta el que pone algo uno de los clientes (y así no hacer `recvfrom()` innecesarios) pero no lo conocemos.

Si ponemos un tiempo muy bajo entre cada `recvfrom()` se hacen demasiadas llamadas, pero si es muy alto el tiempo, se puede acumular en el buffer los mensajes

Para "saber" el tiempo, se usa estadística (específicamente variables aleatorias)



5

si el datagrama no ha llegado, da error de que no ha llegado, tendras que volver a repetir la llamada

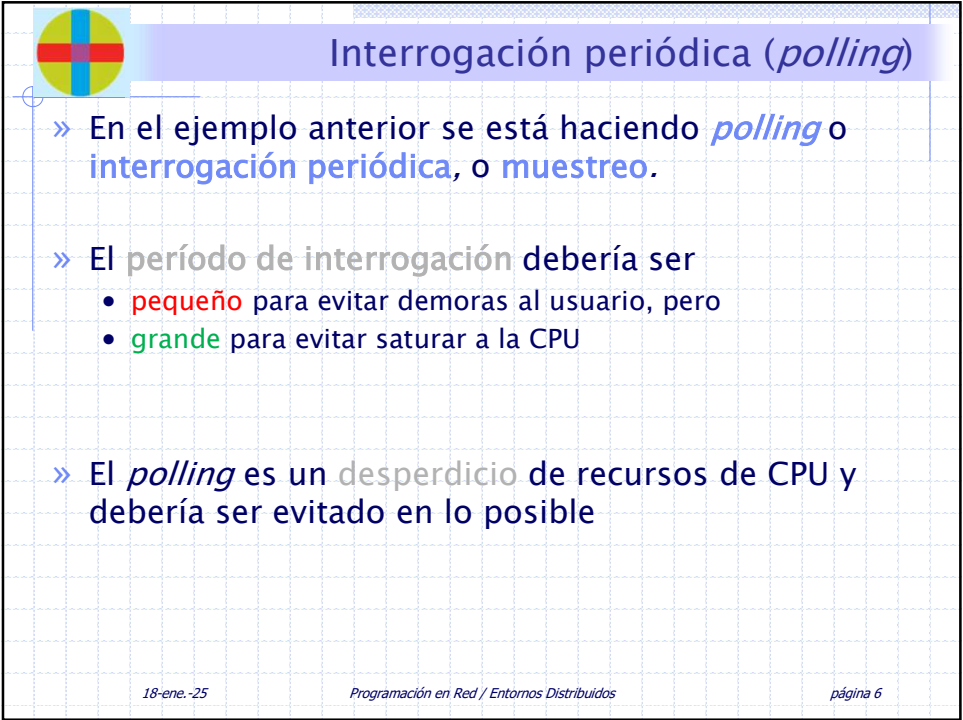
Mientras no haya escrito en el buffer, va haciendo `recvfrom()` hasta que uno de los dos ya ha puesto algo

Los discos se suelen hacer o con una VA normal o una Poisson

Cuando es red hay mas de una variable aleatoria porque hay muchos factores.

Pero haciendo esto la vas a cagar.

Al no saber calcular los t de espera, usar asincronismo



6

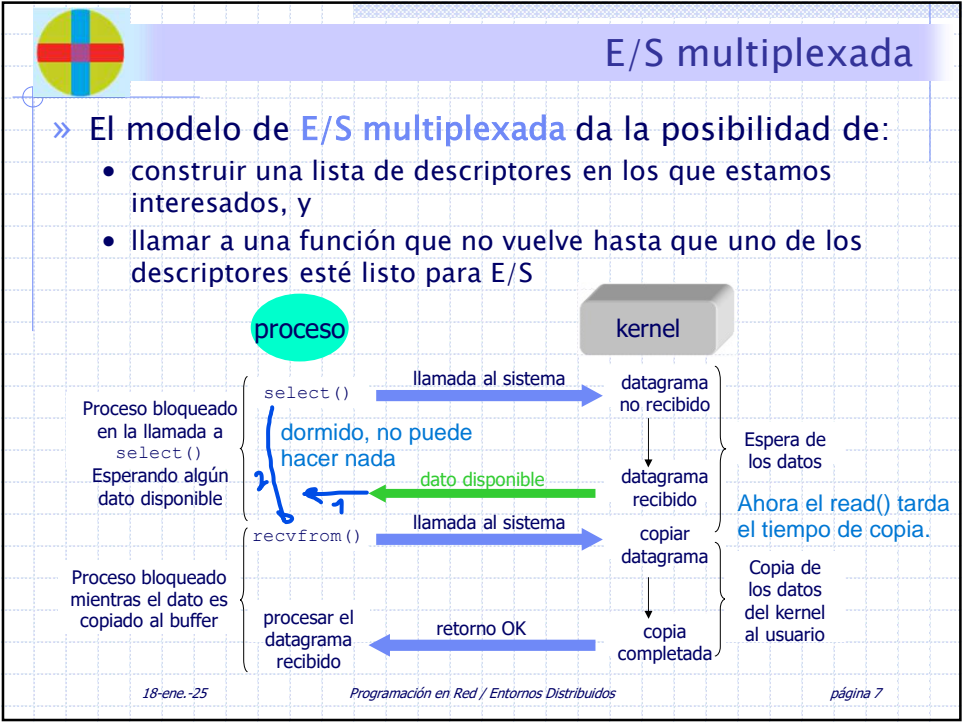
Submuestreo: muy lento tomando las muestras.

Sobremuestreo: demasiado rapido tomando muestras, gastando recursos.

El sobremuestreo pone mucho estres en la CPU, haciendo que un proceso que no gastaba mucho, ahora pone la CPU al 100%

En vez de leer constantemente a todos, esperas a que `select()` te diga quien ha escrito algo

El `select()` en el momento en el que uno de los clientes ha escrito, cuando ha llegado el paquete el kernel avisa y dice quien lo ha enviado, y cuando se hace el `read()` no se bloquea porque ya hay datos (solo un `read()` si quieres hacer mas reads hay que hacer mas `select()`)



No se puede hacer nada útil mientras estas dormido (`select()` duerme el proceso hasta que se recibe un mensaje)

No es una solución general, pero si se quiere que haga otra cosa, mejor que lo haga otro proceso

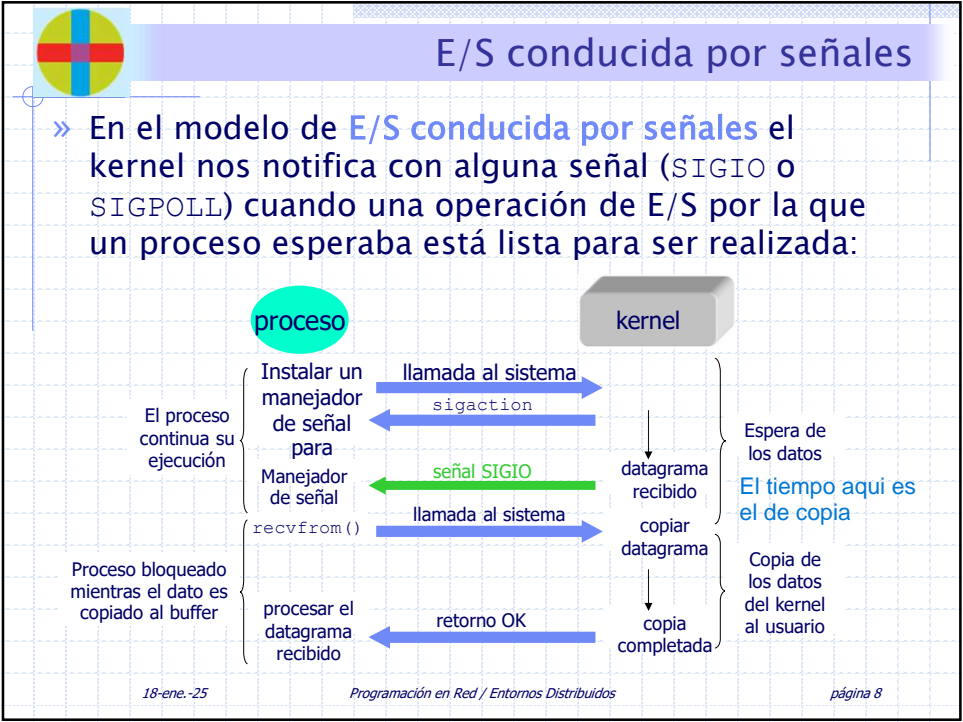
No es una señal es una llamada al sistema

7

El kernel te avisa diciendo que hay E/S aqui, y ya hace el `read()`, cuando hace signal es que los datos no estan disponibles

Mientras no recibe una signal E/S, puede hacer cualquier otra cosa.

Los problemas bienen al interrumpir en el t de copia, que cuando hay muchos clientes simultaneos puede ser mucho lio de manejo de signals




Al interrumpir el proceso, le cortas el flujo linal del programa. Pierdes el control, porque al final te van a avisar cuando esta ocupado, y tendra que retomarse despues (mucho lio)

Y si recibe otra signal mientras esta haciendo la tarea de una signal anterior, asi que lo que pasa es que acaba ignorando signals y se pierden datagramas.

O puede pasar que lo hagas reentrante, pero es muy dificil y se pueden quedar mensajes a medias.

Con muchas signals a la vez, manejarlas es muuy jodido

8




E/S conducida por señales

- » Independientemente de cómo manipulemos la señal, este sistema ofrece varias **ventajas**:
 - La ventaja fundamental de la E/S conducida por señales es que el proceso no permanece bloqueado mientras espera a que los datos estén listos
- » El programa puede continuar ejecutándose después de la petición...
 - y esperar a ser notificado por el manipulador de señales
 - o bien, que sea el propio manejador de señales el que realice la operación

18-ene.-25 Programación en Red / Entornos Distribuidos página 9

9



E/S asíncrona

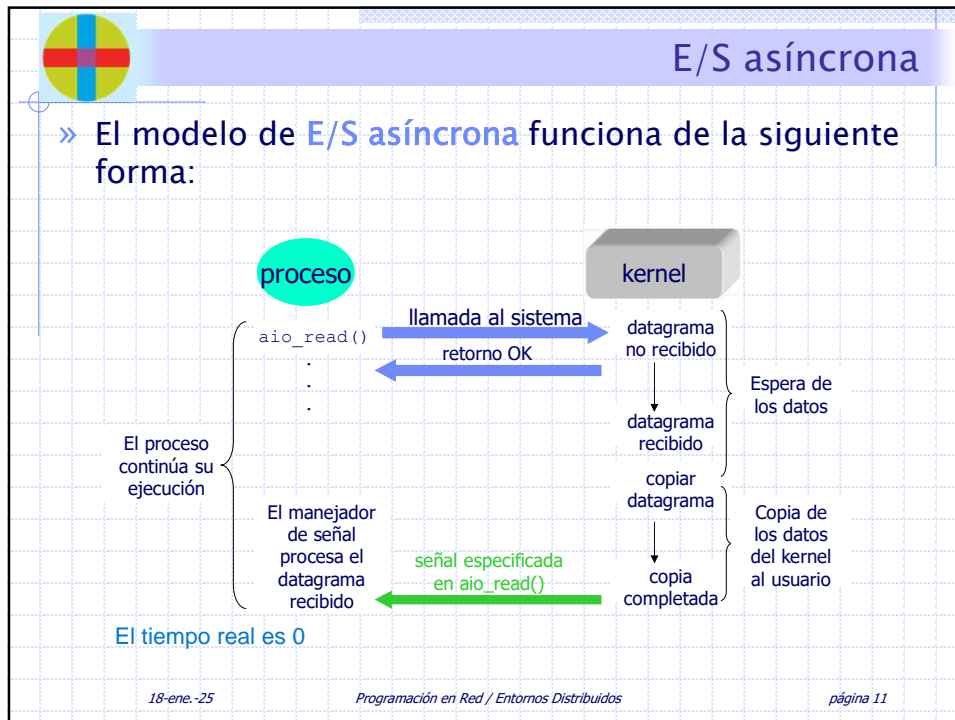
- » El estándar POSIX.1 introdujo el modelo de **E/S asíncrona**
 - Las funciones del modelo asíncrono definidas por el estándar comienzan por los sufijos `aio_` ó `lio_`
 - En este esquema le indicamos al kernel el **comienzo** de una operación y éste nos notificará cuando se **complete** la operación entera, incluyendo el copiado de datos desde el kernel a la memoria del proceso
- » La principal diferencia entre este modelo y la E/S conducida por señales es la siguiente:
 - en este último el kernel nos avisa cuando una operación de E/S puede ser **iniciada**
 - en el modelo asíncrono el kernel nos indica cuando la operación está **completada**

18-ene.-25 Programación en Red / Entornos Distribuidos página 10

10

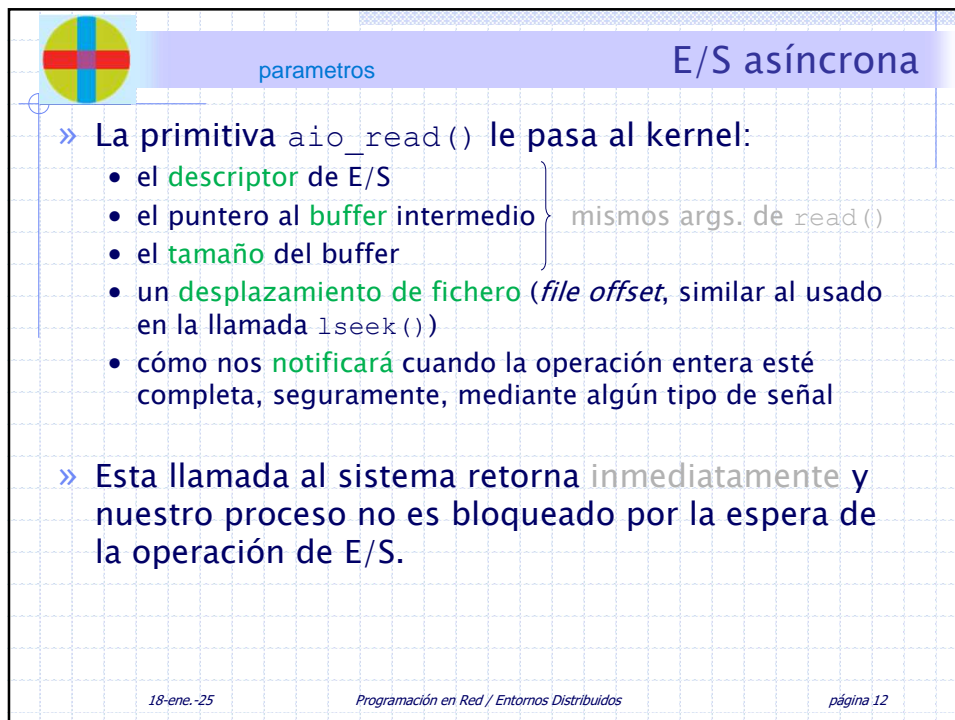
llamas al sistema y le dices que quieres hacer un `read()` de un cliente y le dices donde quieres que lo pongas, tardas 0 porque el tiempo de copia ya esta hecho.

Pero aqui la signal se envia cuando ya esta todo hecho, mientras tanto el proceso puede hacer otra cosa.



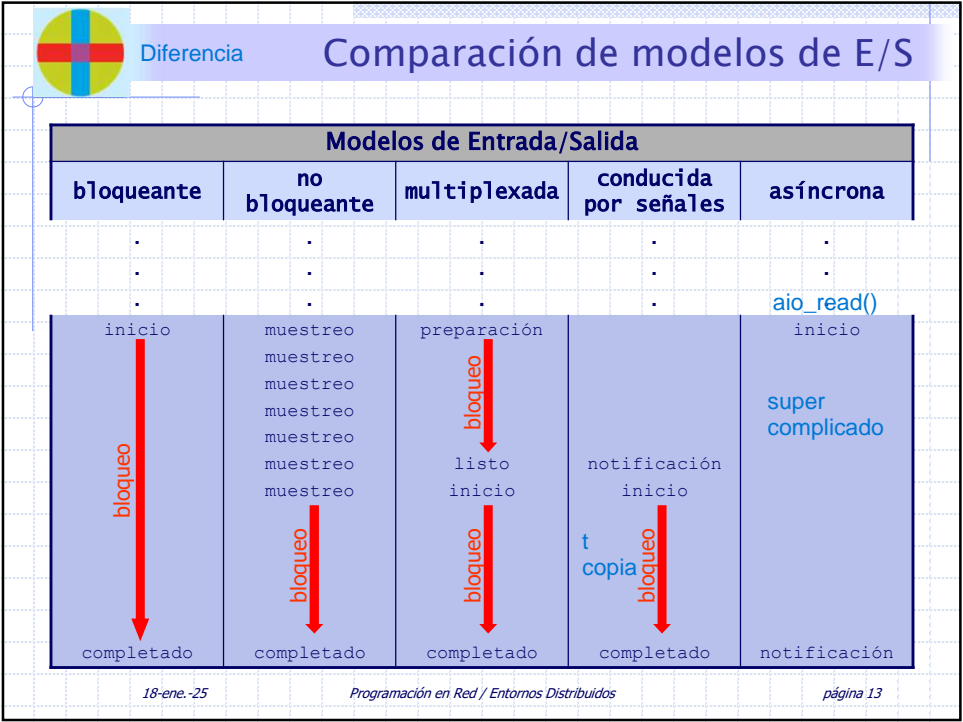
Se puede desordenar, pero se puede poner cosas como num de seq a los paquetes por ejemplo.

11

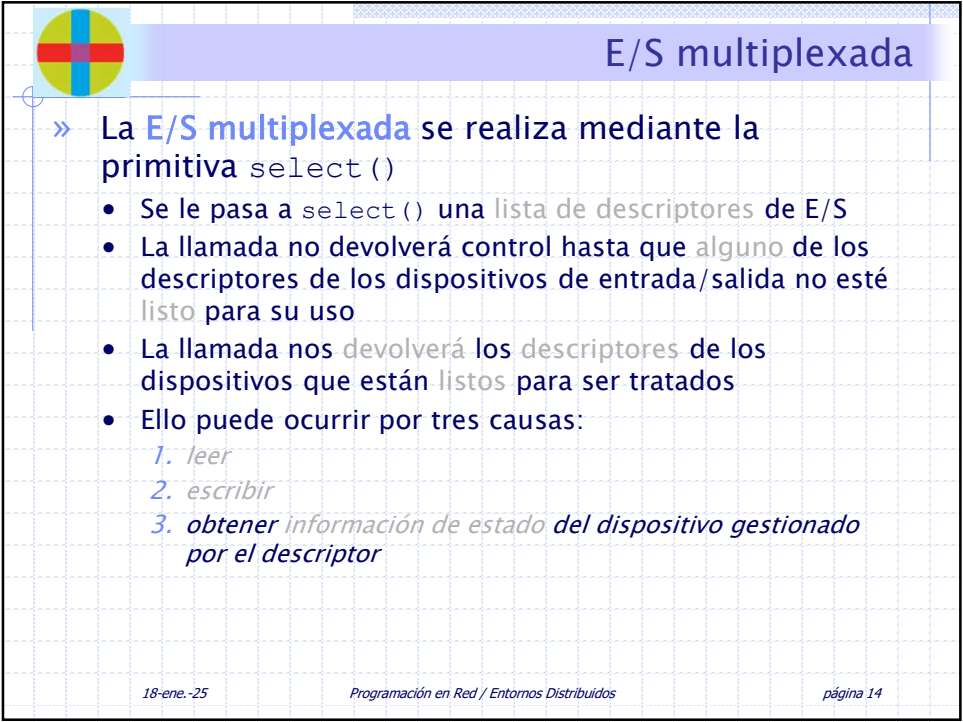


12


La multiplexada es la mas adecuada para la P07



13



14



La llamada `select()`

» A continuación se presenta la **firma** de `select()`


- Elementos válidos en los listas:
 - › Objetos de tipo fichero (los devueltos por `open()`)
 - › Descriptores de fichero (los devueltos por `os.open()` o similares)
 - › Sockets (los devueltos por `socket.socket()`)

```
import select
ready_rlist, ready_wlist, ready_xlist =
    select.select(rlist, wlist, xlist[, timeout])
```

- Funcionamiento
 - › En Unix funciona con todo tipo de descriptores de fichero
 - › En Windows esta llamada solamente funciona con sockets ya que la implementa la librería Winsock y no el sistema de E/S
 - › Es la llamada de muestreo más multiplataforma

18-ene.-25 Programación en Red / Entornos Distribuidos página 15

15



Parámetros de `select()`

» Parámetros de entrada de `select()`:

- **Conjuntos de descriptores:** listas de descriptores de E/S que contienen los dispositivos...
 - › `rlist`: desde los que el proceso desea **leer** datos
 - › `wlist`: en los que el proceso desea **escribir** datos
 - › `xlist`: desde los que el proceso desea estar informado de **cambios excepcionales** en los mismos
 - › Si no se va a usar alguna lista (lo normal es usar sólo `rlist`) puede pasarse como parámetro la lista vacía `[]`
- **Tiempo `timeout`:** límite superior del tiempo de espera de respuesta; hay **tres posibilidades**:
 - › *Sin tiempo de espera* (muestreo: retorno inmediato)


```
Timeout = 0
```
 - › *Con tiempo de espera* en segundos definido por el usuario


```
Timeout = 4.5
```


 (punto flotante: fracciones de segundo)
 - › *Bloqueo* (interrumpible mediante la recepción de una señal)


```
Timeout = None
```

 (valor por defecto)

18-ene.-25 Programación en Red / Entornos Distribuidos página 16

16




Valor de retorno de `select()`

» Valor de retorno de `select()`:

- Devuelve tres listas con los *descriptores listos* para cada uno de los criterios.
- Cada una de las listas es un subconjunto de las pasadas como parámetros
- En Windows no se garantiza que se soporte la operación con más de una lista
- Si se produce un **error** (ejemplo: la recepción de una señal) `select()` lanza una excepción y fija `errno` con el valor apropiado.
 - › En este caso los valores de retorno son indefinidos y **no se debe confiar en ellos**.
 - › Posibles errores:
 - EBADF – descriptor de fichero inválido en uno de los conjuntos.
 - EINTR – capturada una señal no bloqueante.
 - EINVAL – `nfds` es menor que cero.
 - ENOMEM – no se ha podido pedir memoria para buffer interno.

18-ene.-25 Programación en Red / Entornos Distribuidos página 17

17




Descriptores listos para su uso

» ¿Cuándo está listo un descriptor?

- Un descriptor en el conjunto `rlist` está listo cuando es seguro que una lectura realizada sobre dicho descriptor **no** se bloqueará
- Un descriptor en el conjunto `wlist` está listo cuando es seguro que una escritura realizada sobre dicho descriptor **no** se bloqueará
- Un descriptor en el conjunto `xlist` está listo cuando:
 - › se reciben datos **fuera de banda** en una conexión de red
 - › en **determinadas condiciones** que pueden darse en una pseudo terminal
 - › Su uso queda como ejercicio para los más aventurados

18-ene.-25 Programación en Red / Entornos Distribuidos página 18

18



E/S multiplexada (II)


- » SVR3 proporciona la llamada `poll()` sólo para trabajar con identificadores de *streams*
 - Mecanismos que proporcionan conexiones *full-duplex* entre un proceso y un manejador de dispositivo de entrada/salida (*i/o device driver*)
- » SVR4 la amplía para cualquier tipo de dispositivo

```
import select
objeto_poll = select.poll()
```

- » Implementada en casi todas las versiones de Unix y algunos otros sistemas operativos (Windows)
 - Más eficiente [$O(n^{\circ} \text{descriptores registrados})$] que la llamada `select()` [$O(n^{\circ} \text{de descriptor más alto})$]

18-ene.-25
Programación en Red / Entornos Distribuidos
página 19

19




Interfaz de objetos `poll()` (I)

- » `obj_poll.register(fd[, eventmask])`
 - Registra el **descriptor de fichero u objeto fichero** `fd`
 - El parámetro opcional `eventmask` es la máscara de bits con el tipo de **eventos** sobre los que se muestrea a `fd`
 - › Constantes a usar en `eventmask`:
 - `POLLIN` datos pendientes de lectura
 - `POLLPRI` datos urgentes pendientes de lectura
 - `POLLOUT` la escritura no se bloqueará
 - `POLLERR` ha habido algún error
 - `POLLHUP` comunicación terminada (hung up)
 - `POLLNVAL` petición inválida (descriptor no abierto)
 - › Valor por defecto: `POLLIN | POLLPRI | POLLOUT`
- » `obj_poll.modify(fd, eventmask)`
 - Modifica un descriptor ya registrado con la nueva máscara
- » `obj_poll.unregister(fd)`
 - Elimina el descriptor `fd` del registro

18-ene.-25
Programación en Red / Entornos Distribuidos
página 20

20




Interfaz de objetos `poll()` (II)

- » `[(fd, evento)] = obj_poll.poll([timeout])`
 - Muestra el conjunto de descriptors registrados
 - El parámetro `timeout` (es opcional) especifica un tiempo de espera
 - › Si se pasa un valor, debe ser un entero en milisegundos
 - › Si no se pasa, es cero, negativo, o `None`, la llamada se bloquea hasta que se produzca algún evento
 - Devuelve una lista de tuplas: `[(fd, evento)]`
 - › `fd` es el descriptor afectado
 - › `evento` es la máscara de eventos para los que `fd` está listo
 - Una lista vacía indica que ha vencido el `timeout` sin que ningún descriptor esté listo

18-ene.-25 Programación en Red / Entornos Distribuidos página 21

21




Otras llamadas del módulo `select`

```
import select
objeto_epoll = select.epoll([sizehint=-1])
objeto_kqueue = select.kqueue()
objeto_kevent = select.kevent(ident, filter=KQ_FILTER_READ,
                               flags=KQ_EV_ADD, fflags=0, data=0, udata=0)
```


- » `epoll()`
 - Sólo soportada en Linux (versión del kernel $\geq 2.5.44$, liberada en octubre de 2002)
- » `kqueue()` y `kevent()`
 - Sólo soportadas en BSD
- » Cada una de ellas devuelve un objeto diferente, similar al objeto `poll`, con una interfaz específica
 - Su uso queda como ejercicio para los más aventurados

18-ene.-25 Programación en Red / Entornos Distribuidos página 22

22



¿Preguntas?



18-ene.-25

Programación en Red / Entornos Distribuidos

página 23