

Sistemas Operativos

# Comunicación entre Procesos

# Comunicación entre Procesos

- Tuberías. Tipos
- PIPES. Llamadas asociadas
- Ejemplos de uso
- Creación de pipes desde el *shell*
- Más ejemplos

[Ste94]: capítulo 15



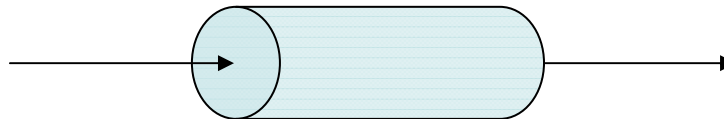
# Tuberías (1 de 2)

Dos tipos:

- UNAMED PIPES O PIPES
- NAMED PIPES, FIFOs O NAMED FIFOs

## CARACTERÍSTICAS COMUNES

- Unidireccionales (un proceso lee y el otro escribe)
- Implementación: i-nodo del que se utilizan los punteros directos  
=> capacidad limitada y dependiente de la implementación:  
mínimo 4096 bytes



# Tuberías (2 de 2)

## DIFERENCIAS

UNAMED	NAMED
comunican procesos emparentados	comunican cualquier proceso
No están representados en el Sistema de Ficheros mediante un nombre	Tienen nombre en el Sistema de Ficheros, como un fichero cualquiera
Se crean con <b>pipe( )</b>	Se crean con <b>mknod( )</b>
Muy utilizados, especialmente en el <i>shell</i> : who   sort   lp -oq	Muy poco utilizados

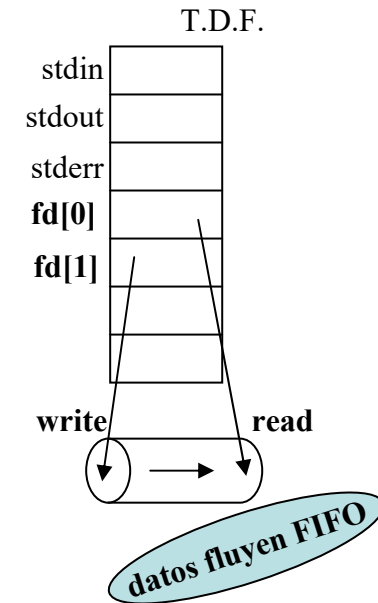
# PIPES: Llamadas asociadas (1)

- **Sintaxis:** `# include <unistd.h>`  
`int pipe( int fd[2] );`
- **Acción:** crea una pipe  
asigna i-nodo  
crea 2 entradas en TFA

en *fd* se almacenan los 2 primeros fd's libres:

- *fd*[0] abierto para lectura en pipe
- *fd*[1] abierto para escritura en pipe
- lo escrito en *fd*[1] se lee por *fd*[0]
- apuntan a las correspondientes entradas en TFA
- para manejar la pipe como si fuese un fichero normal (read, write)

- **Devuelve:** 0 si bien ó -1 si error



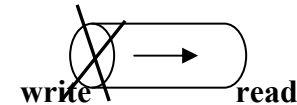
**Prob. clásico del productor-consumidor**

**Buffer en memoria (más rápido)**  
**16 KiB en hendrix**

# PIPES: Transmisión de datos

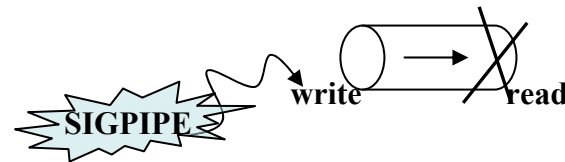
## read( )

- Se leen datos del extremo de lectura
- Si pipe vacía => bloquea
- Si pipe vacía y extremo de escritura cerrado  
=> no bloquea y devuelve 0 (final de datos)



## write( )

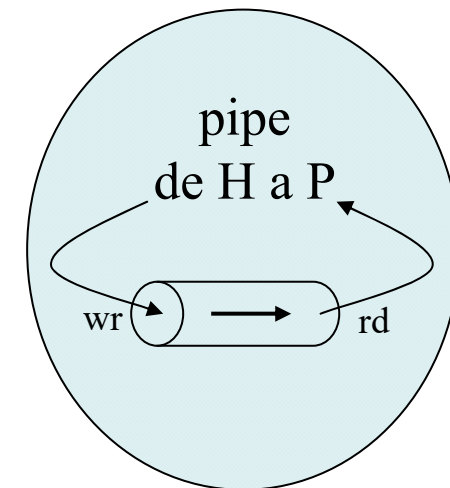
- Se escriben datos en el extremo de escritura
- Si pipe llena => bloquea
- Si no está abierto el extremo de lectura => señal SIGPIPE



# Ejemplo de uso de pipes

```
#include "error.h"
main(argc,argv)
int argc;char *argv[];
{
    int fpipe[2], file, n; char buf[512];
    if(argc!=3) syserr("Numero de parametros");
    pipe(fpipe);
    switch(fork()) {
    case -1: syserr("fork");
    case 0: file=open(argv[1],0);
        while((n=read(file,buf,sizeof(buf)))!=0)
            write(fpipe[1],buf,n);
        printf("Fichero leido\n");
        break;
    default: close(fpipe[1]);
        file=creat(argv[2],0600);
        while((n=read(fpipe[0],buf,sizeof(buf)))!=0)
            write(file,buf,n);
        printf("Fichero copiado\n");
    }
    exit(0); }
```

para copiar un fichero perdiendo el tiempo

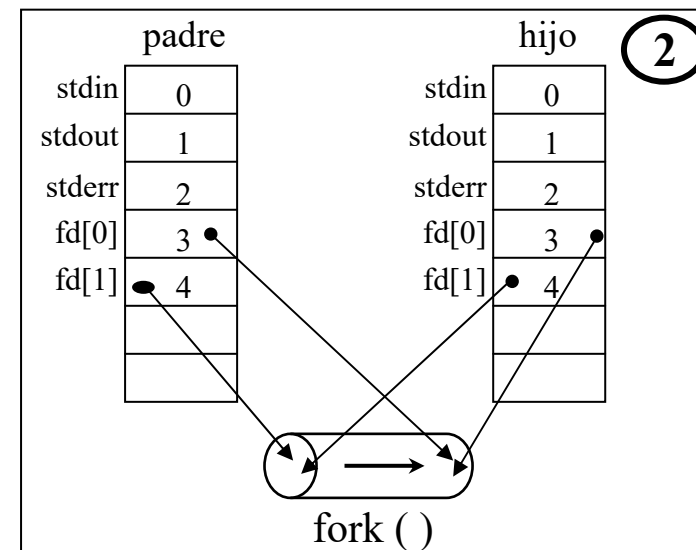
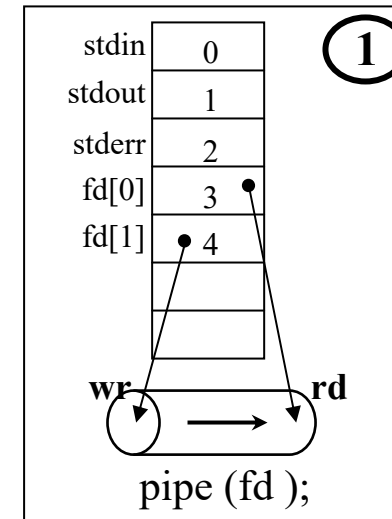


# Pipes y redirecciones (1 de 4)

```
#include "error.h"
main()
{  int fd[2];

    pipe( fd );
    switch( fork( ) ) {
    case -1: syserr( "fork" );
    case 0:  close( fd[0] );
            close( 1 );
            dup( fd[1] );
            close( fd[1] );
            execlp( "who", "who", 0 );
    default: close( fd[1] );
            close( 0 );
            dup( fd[0] );
            close( fd[0] );
            execlp( "sort", "sort", 0 );
    }
}
```

who | sort



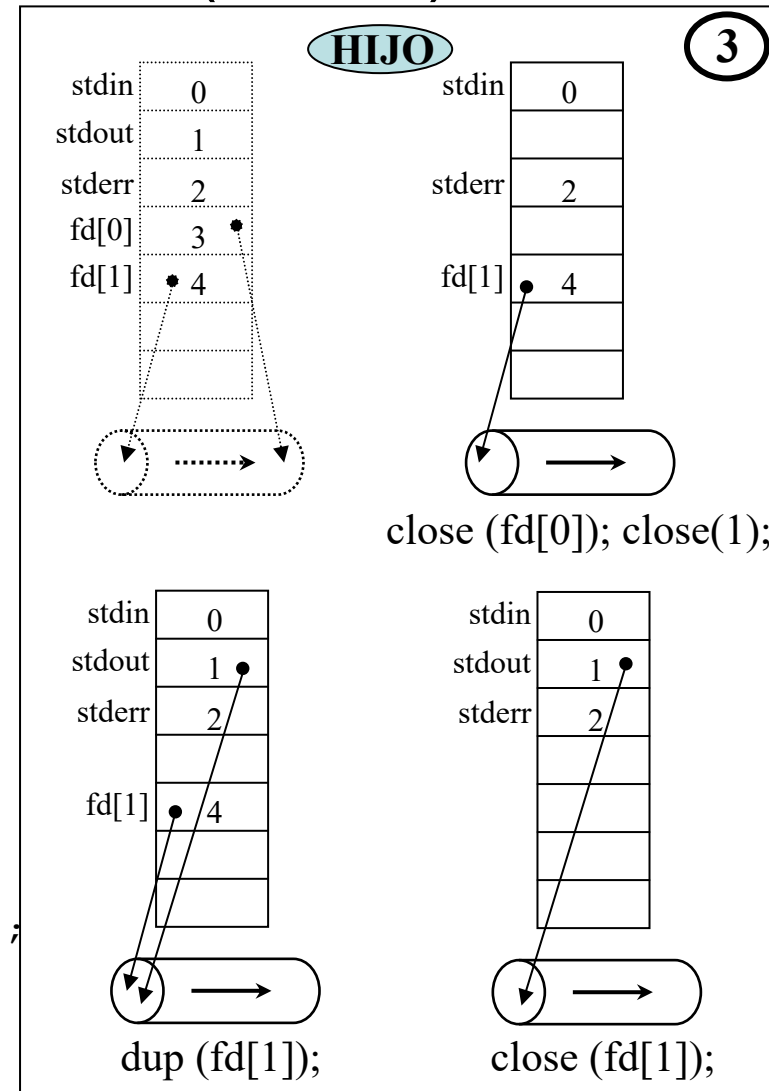
# Pipes y redirecciones (2 de 4)

```
#include "error.h"
main()
{  int fd[2];

    pipe( fd );
    switch( fork() ) {
    case -1: syserr( "fork" );
    case 0: close( fd[0] );
            close( 1 );
            dup( fd[1] );
            close( fd[1] );
            execlp( "who", "who", 0 );
    default: close( fd[1] );
            close( 0 );
            dup( fd[0] );
            close( fd[0] );
            execlp( "sort", "sort", 0 );
    }
}
```

who | sort

3



# Pipes y redirecciones (3 de 4)

```
#include "error.h"
```

```
main()
```

```
{  int fd[2];
```

```
    pipe( fd );
```

```
    switch( fork() ) {
```

```
    case -1: syserr( "fork" );
```

```
    case 0:  close( fd[0] );
```

```
            close( 1 );
```

```
            dup( fd[1] );
```

```
            close( fd[1] );
```

```
            execlp( "who", "who", 0 );
```

```
    default: close( fd[1] );
```

```
            close( 0 );
```

```
            dup( fd[0] );
```

```
            close( fd[0] );
```

```
            execlp( "sort", "sort", 0 );
```

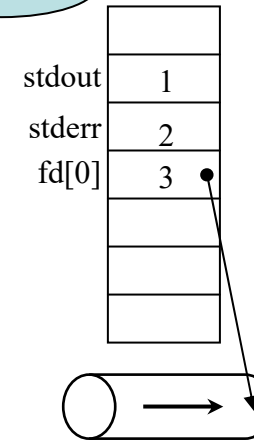
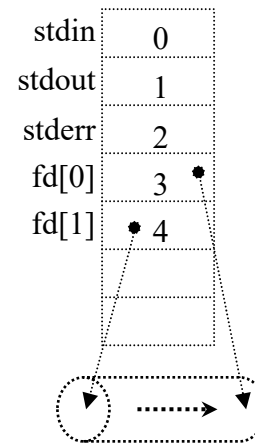
```
    }
```

```
}
```

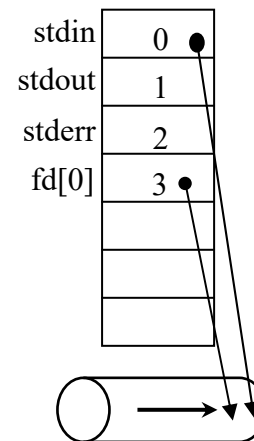
who | sort

PADRE

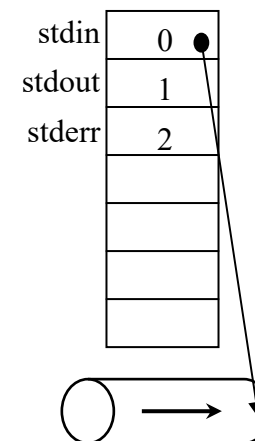
4



close (fd[1]); close(0);



dup (fd[0]);



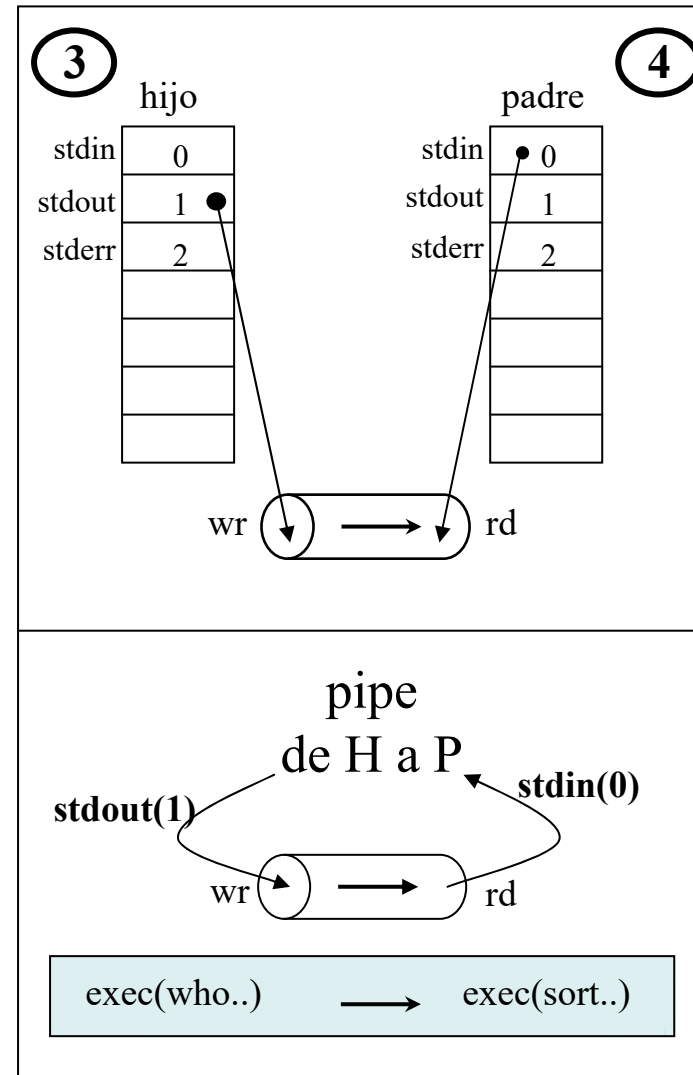
close (fd[0]);

# Pipes y redirecciones (4 de 4)

```
#include "error.h"
main()
{  int fd[2];

    pipe( fd );
    switch( fork() ) {
    case -1: syserr( "fork" );
    case 0:  close( fd[0] );
            close( 1 );
            dup( fd[1] );
            close( fd[1] );
            execlp( "who", "who", 0 );
    default: close( fd[1] );
            close( 0 );
            dup( fd[0] );
            close( fd[0] );
            execlp( "sort", "sort", 0 );
    }
}
```

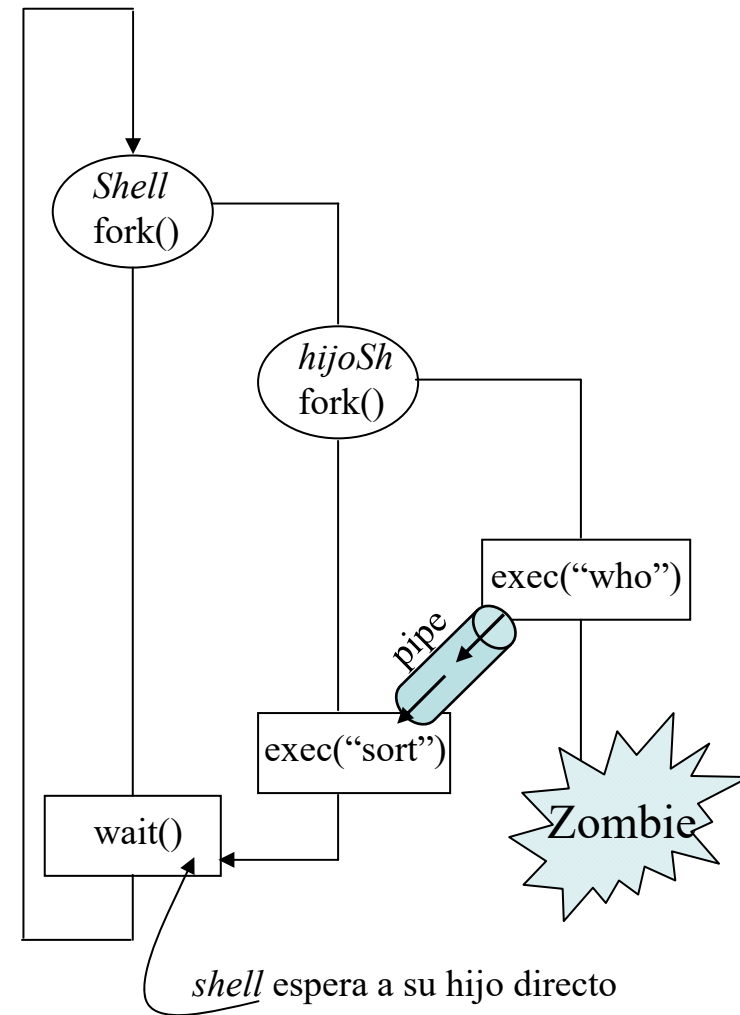
who | sort



# Creación de pipes desde el *shell*

```
for( ; ; ){
    lectura de comandos...;
    parsing...;
    switch( fork() ) {
        case -1: ...
        case 0: pipe( fd );
                switch( fork() ) {
                    case -1: ...
                    case 0: redirecciones;
                           exec ( "who" );
                    default: redirecciones;
                           exec ( "sort" );
                }
        default: wait( estado );
    }
}
```

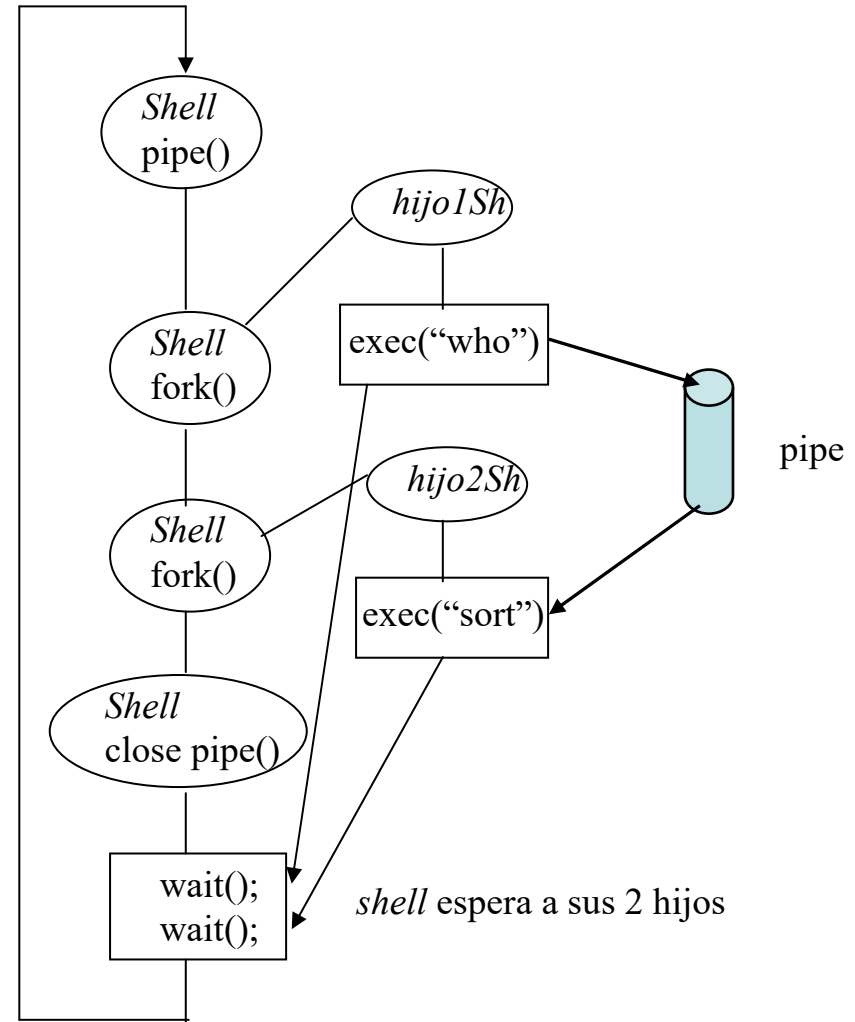
**who | sort**



# Creación de pipes desde el *shell*

```
for( ; ; ){
    lectura de comandos...;
    parsing...;
    pipe( fd );
    switch( fork() ) {
        case -1: ...
        case 0: redirecciones;
                exec("who");
        default: switch( fork() ) {
            case -1: ...
            case 0: redirecciones;
                    exec ( "sort" );}}
    close fd[0] fd[1];
    wait( estado ); /* 1er hijo */
    wait( estado ); /* 2º hijo */
}
```

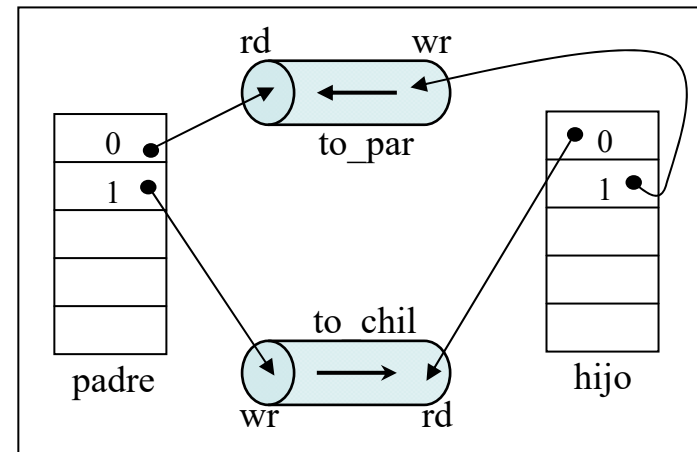
who | sort



# Comunicación bidireccional con 2 pipes

```
#include <string.h>
char string[] = "hello world";
main()
{
    int count, i, to_par[2], to_child[2];
    char buf[256];
    pipe(to_par); pipe(to_child);
    if( fork() == 0 ){
        close(0); dup(to_chil[0]);
        close(1); dup(to_par[1]);
        close(to_par[1]); close(to_chil[0]);
        close(to_par[0]); close(to_chil[1]);
        for( ; ; ){
            if (( count = read(0, buf, sizeof(buf))) == 0)
                exit(1);
            write(1, buf, count);
        }
    }

    close(1); dup(to_chil[1]);
    close(0); dup(to_par[0]);
    close(to_chil[1]); close(to_par[0]);
    close(to_chil[0]); close(to_par[1]);
    for( i = 0 ; i < 15 ; i++ ){
        write(1, buf, count);
        read(0, buf, sizeof(buf));
    }
}
```



## Otro ejemplo

```
/* primer.c */

main(){
    int  id, fd[2];
    id = fork();
    pipe(fd);
    switch ( id ) {
        case -1: exit(1);
        case 0:  close(0);
                 dup(fd[0]);
                 close(fd[0]);
                 close(fd[1]);
                 execlp("segun", "segun", 0);
                 exit(1);
        default: close(1);
                 dup(fd[1]);
                 close(fd[0]);
                 close(fd[1]);
                 execlp("segun", "segun", 0);
                 exit(1);
    }
    exit(0);
}
```

```
/* segun.c */

main()
{  char c;

    while ( read(0, &c, 1 )!= 0 )
        write(2, &c, 1);
    write(1, &c, 1);
    exit(0);
}
```

No funciona