

Applicazioni concorrenti

- Attraverso la syscall `fork` è possibile eseguire un programma in più processi distinti
- Tali processi, come abbiamo visto in più occasioni, non possono condividere informazioni in quanto i dati sono assolutamente privati ad ogni processo
- Possono al più scambiarsi informazioni in due occasioni:
 - al momento dell'avvio, in quanto il processo “figlio” ottiene una copia esatta dei dati che il processo “padre” aveva al momento della `fork()`. In questo modo, ad esempio, un processo figlio può accedere alla command line o a dati calcolati dal padre in precedenza
 - al momento della terminazione, quando il processo “padre” attraverso la `wait()` ottiene due byte relativi alla causa di terminazione, segnale o exit value
- ... poco per progettare applicativi concorrenti!

Inter process communication

- Un applicativo concorrente deve tipicamente coordinare le attività dei vari processi,
- eseguendo sincronizzazioni,
- trasferendo dati da un processo all'altro,
- fino a richiedere l'esecuzione di codice in processi “remoti”, ovvero RPC

- per questo esistono delle syscall opportune

- l'intervento del Sistema Operativo è necessario per condividere dati (file), eventi (segnali) e organizzare il time slicing (sospensione su attesa I/O o segnale)

Sistemi di IPC

- La realizzazione di applicazioni multiprocesso (es. client/server) in ambiente unix risulta particolarmente comoda per la disponibilità di primitive per la comunicazione fra processi IPC per:
 - ‘trasmettere’ informazioni:
 - pipe
 - fifo
 - socket
 - sincronizzazioni e segnalazioni:
 - signal
 - kill
 - pause

pipe

- Una pipe rappresenta un canale di comunicazione unidirezionale fra due processi che:
- si presenta come una coppia di file, uno aperto in sola lettura e l'altro aperto in sola scrittura
- ogni byte/carattere scritto sul file in sola scrittura viene reso disponibile in lettura sull'altro file descriptor
- non corrisponde ad alcun file 'reale' nel filesystem
- ha, salvo richiesta contraria, un comportamento "bloccante"

comportamento bloccante

- il processo che tenta di leggere dal file descriptor associato al lato di lettura quando non vi sono caratteri disponibili, viene messo in “attesa” dal Sistema Operativo
- lo stesso avviene al processo che tenta di scrivere sul file descriptor associato al lato di scrittura quando non vi è spazio per ulteriori caratteri nel buffer, ovvero quando il ritmo di prelievo di byte dal lato di lettura è più lento di quello con cui vengono prodotti
- Il Sistema Operativo, quindi, sincronizza in modo implicito i due processi che si scambiano dati, regolando la velocità del processo più “rapido” diminuendo le time slices a sua disposizione

pipe

- La primitiva della LIBC per la creazione di una pipe è:

```
int pipe(int fd[2]);
```

- L'argomento richiesto da pipe() è un puntatore ad intero che punti ad un area di memoria in grado di contenere almeno due interi
- Nel primo di questi due interi, fd[0], verrà scritto il file descriptor corrispondente al file in sola lettura, semplicemente “lato di lettura”
- Nel secondo, fd[1], verrà memorizzato il file descriptor corrispondente al file in sola scrittura, o “lato di scrittura”

Uso di pipe

- La pipe rappresenta un semplice canale di comunicazione fra processi, usato ad esempio dalla shell per le pipeline sulla command line (cat file | grep “ciao” | less)
- Deve essere creato dal processo “padre” prima della fork(), in modo da poter trasmettere i descrittori in “eredità” al processo “figlio”
- Uno dei due processi presenti dopo la fork() assumerà il ruolo di produttore, usando solo il lato di scrittura
- L'altro processo userà solo il lato di lettura ed assumerà il ruolo di consumatore

```
/* interfaccia per fork() */
#include <sys/types.h>
#include <unistd.h>
/* interfaccia per printf() */
#include <stdio.h>
/* errno */
#include <errno.h>

/* prototipo della funzione 'main' */
int main(void);

static int pipe_fd[2];
static pid_t child;
#define BSIZE 30
static char child_buffer[BSIZE];
static char father_buffer[BSIZE]="Prova di messaggio";
static int blen;

/* main definition */
int main() {
```

```

if (pipe(pipe_fd)) {
    /* Error */
    printf("Internal error: %d\n", errno);
    exit(-1);
}
switch (child = fork()) {
    case 0: /* Child */
        if ((blen=read(pipe_fh[0],child_buffer,BSIZE-1))>0) {
            child_buffer[blen]='\0';
            printf("Received:%s\n",child_buffer);
            exit(0);
        }
        exit(-3);
    case -1: /* Error */
        printf("Internal error: %d\n",errno);break;
    default:
        /* Father */
        write(pipe_fh[1],father_buffer,BSIZE);
        printf("Sent\n");
        close(pipe_fh[1]);
        break;
}

```

popen()

- Nel caso in cui uno dei due processi sia realizzato mediante l'invocazione di un eseguibile esistente (per il quale potremmo usare la `exec`), è possibile realizzare la pipe semplicemente chiamando la funzione

```
FILE *fp = popen(const char *command, const char  
                *mode)
```

```

#include <stdio.h>
#include <errno.h>
int main(void);
static FILE* tochild;
#define BSIZE 30
static char father_buffer[BSIZE]="Prova di messaggio";
/* main definition */
int main() {
    if ((tochild=popen("rev","w"))==NULL) {
        /* Error */
        printf("Internal error: %d\n", errno);
        exit(-1);
    }
    fprintf(tochild,"%s\n",father_buffer);
    exit(0);
}

```

fifo

- Il concetto di pipe può essere esteso fornendo alla pipe un nome nel filesystem, di fatto estendendo a qualsiasi processo (anche non discendente) la possibilità di accedervi.
- La creazione di un file di questo tipo è resa possibile dalla primitiva:

```
int mkfifo(const char *filename, mode_t mode);
```

- che crea un file particolare nella directory corrente denominato filename; mode può essere utilizzato per determinare i permessi di accesso necessari.
- Prima di potervi accedere, la fifo deve essere aperta da entrambe gli utilizzatori, uno in lettura ed uno in scrittura

```
/* interfaccia per mkfifo() e read() */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
/* interfaccia per printf() */
#include <stdio.h>
/* errno */
#include <errno.h>

#define FIFO_FILENAME "named_pipe"

/* prototipo della funzione 'main' */
int main(void);

static int pipe_fd,out_fd;
static char ch;

/* main definition */
int main() {
```

```

if (mkfifo(FIFO_FILENAME,00600)) {
    /* Error */
    printf("Pipe create error: %d\n", errno);
    exit(-1);
}
if ((pipe_fd=open(FIFO_FILENAME,O_RDWR))==-1) {
    /* Error */
    printf("Pipe open error: %d\n", errno);
    exit(-1);
}
/* close filedes 0,2 not used */
close(0);
close(2);
/* out to stdout */
out_fd=1;
for (;;) {
    /* wait until receive '!' */
    switch(read(pipe_fd,&ch,1)>0) {
        case 1:
            if (ch=='!') {
                close(pipe_fd);
                unlink(FIFO_FILENAME);
                exit(0);
            }
            write(out_fd,&ch,1);

```

```
        break;
    case 0:
        /* should never happen */
        write(out_fd,"0 read\n",7);
        break;
    default:
        /* error */
        write(out_fd,"Bad pipe\n",9);
        close(pipe_fd);
        unlink(FIFO_FILENAME);
        exit(-1);
    }
}
}
```

Particolarità e limiti della fifo

- E' possibile utilizzare una fifo come canale di comunicazione fra più processi produttori ed un lettore (server)
- Non è assicurata l'atomicità dell'operazione di scrittura se non per buffer di dimensione limitata; se due processi scrivono contemporaneamente in una fifo, il ricevitore potrebbe leggere i dati mescolati
- Due operazioni di scrittura potrebbero originare una sola lettura del messaggio complessivo; se è richiesta la separazione dei messaggi, questa deve essere curata dal programmatore
- Se due processi si mettono in 'ascolto' sulla stessa pipe, non è possibile determinare quale dei due riceverà il messaggio

socket

- Il socket rappresenta forse lo strumento di comunicazione più noto e potente di unix; un socket instaura un canale di comunicazione bidirezionale fra processi che possono risiedere addirittura su macchine diverse collegate fra loro.
- Per accedere a risorse distribuite, il meccanismo socket necessita di una regola per individuare gli estremi del canale di comunicazione, il *namespace*; la selezione del namespace determina il protocollo (PF_) ed il formato degli indirizzi (AF_)
- I namespace più utilizzati e normalmente reperibili in tutte le macchine unix sono:
- PF_INET (canali fra macchine diverse su protocollo internet)
- PF_FILE o PF_UNIX (canali locali alla macchina)

Unix domain socket

- L'utilizzo dei socket per IPC locale è molto simile a quello di una pipe o fifo, ma con alcune importanti eccezioni:
 - la bidirezionalità del canale di comunicazione
 - la possibilità di selezionare un sistema *continuo* di comunicazione SOCK_STREAM o un sistema *slottizzato* SOCK_DGRAM
- La creazione di un socket locale è realizzabile con l'invocazione della primitiva:

```
int socketpair (int namespace, int style, int  
protocol, int fh[2]);
```

- con i valori:
 - namespace = PF_UNIX (sinonimo di PF_FILE)
 - style = SOCK_STREAM o SOCK_DGRAM
 - protocol = 0

Socket come pipe

- Un socket creato con `socketpair` rappresenta forti analogie con una pipe e può essere utilizzato per sostituirla, semplicemente invocando *socketpair* al posto di *pipe*
- Esattamente come una pipe non ha alcuna traccia nel filesystem e, di conseguenza, la comunicazione è ristretta a processi con legame di *parentela*.
- E' possibile anche creare socket con un nome nel filesystem, come accade per una fifo in rapporto ad una pipe. In questo caso la comunicazione può avvenire anche fra processi indipendenti.

```

/*****
 * $Id: es45.c,v 1.1 2002/03/04 08:45:54 valealex Exp valealex $
 * socket come pipe
 * *****/

/* interfaccia per fork() */
#include <sys/types.h>
#include <unistd.h>
/* interfaccia per printf() */
#include <stdio.h>
/* errno */
#include <errno.h>
/* socket */
#include <sys/socket.h>

/* prototipo della funzione 'main' */
int main(void);

static int pipe_fh[2];
static pid_t child;
#define BSIZE 30
static char child_buffer[BSIZE];
static char father_buffer[BSIZE]="Prova di messaggio";
static int blen;

```

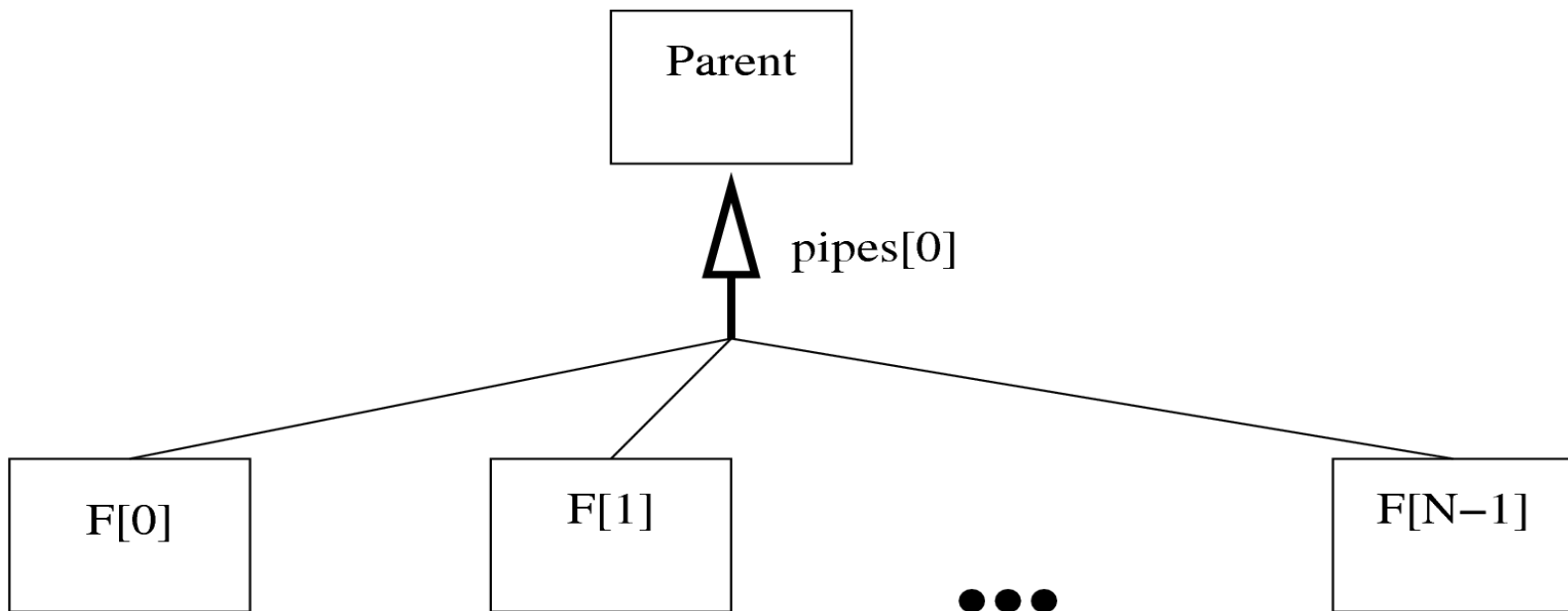
```

int main() {
    if (socketpair(PF_UNIX, SOCK_STREAM, 0, pipe_fh)) {
        /* Error */
        printf("Internal error: %d\n", errno);
        exit(-1);
    }
    switch (child = fork()) {
        case 0: /* Child */
            if ((blen=read(pipe_fh[0], child_buffer, BSIZE-1))>0) {
                child_buffer[blen]='\0';
                printf("Received:%s\n", child_buffer);
                exit(0);
            }
            exit(-3);
        case -1: /* Error */
            printf("Internal error: %d\n", errno);
            break;
        default:
            /* Father */
            write(pipe_fh[1], father_buffer, BSIZE);
            printf("Sent\n");
            close(pipe_fh[1]);
            break;
    }
    exit(0);
}

```

Uso di pipe da più processi

- E' possibile condividere un lato della pipe (tipicamente quello in scrittura) fra più processi, realizzando una struttura di comunicazione simile alla seguente:

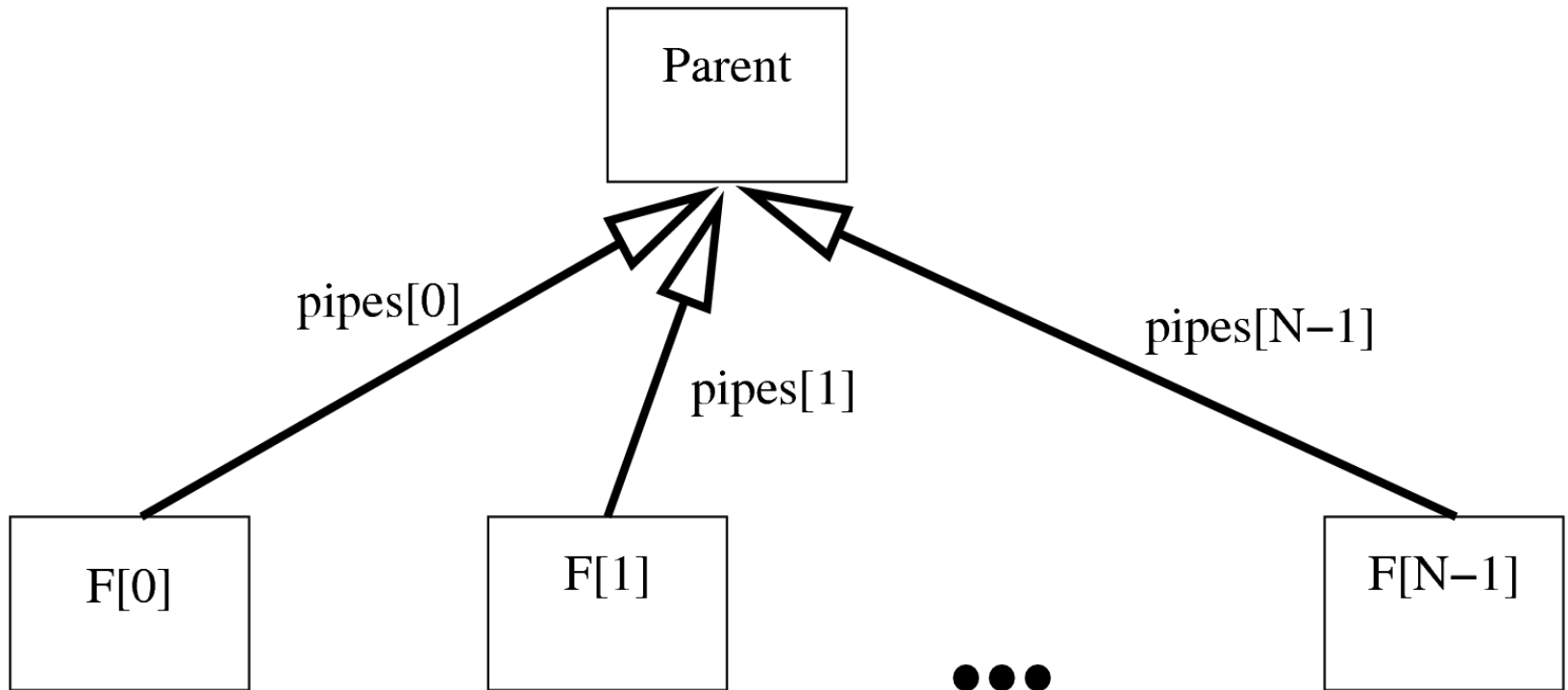


1 pipe N processi

- Usando una sola pipe, il consumatore deve usare qualche informazione nei dati scambiati per sapere quale processo li ha generati (es. struttura con pid o indice)
- Non è possibile decidere l'ordine di ricezione, il primo produttore che scrive sarà l'autore della prima informazione ricevuta

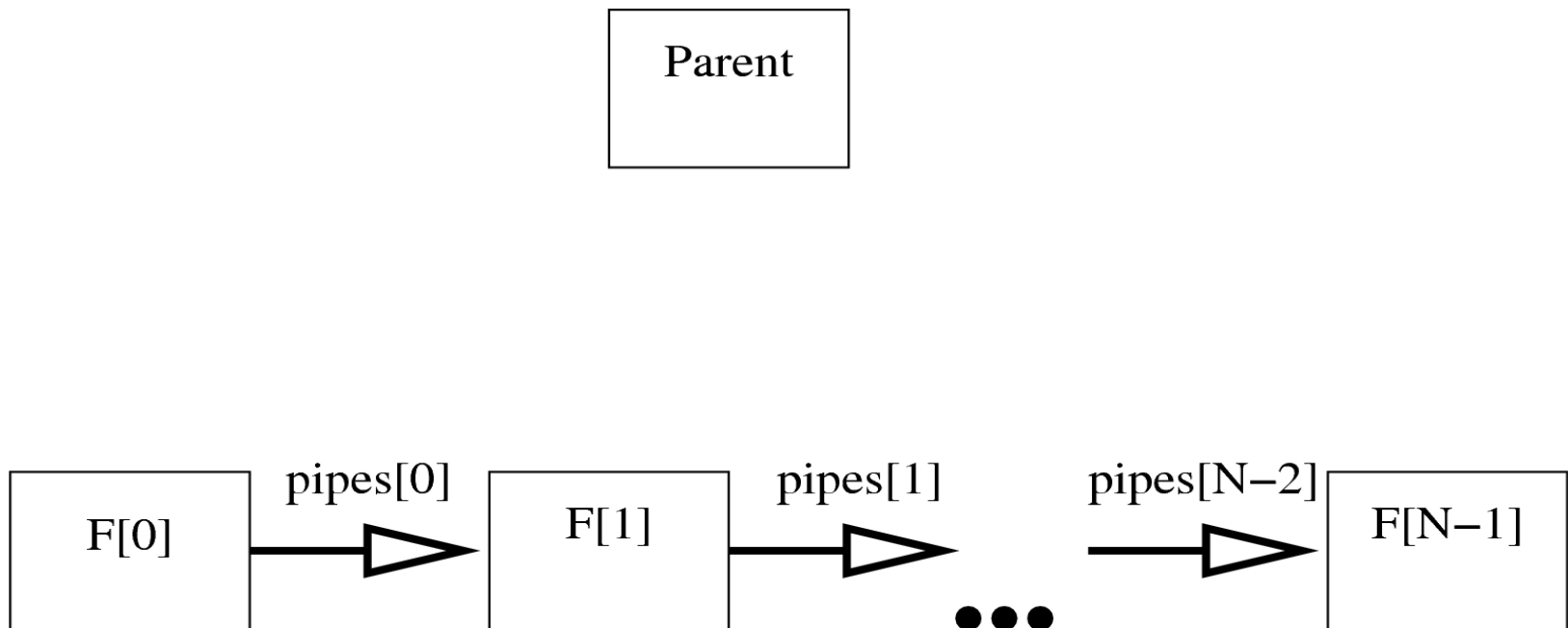
N pipe N processi

- Usando N pipe distinte, il consumatore può decidere da chi prelevare l'informazione, senza pid o indice nei dati



pipe in catena

- Se il “padre” crea una pipe per ogni relazione d'ordine o di comunicazione ed i figli le usano opportunamente, si ottiene una sincronizzazione “senza l'intervento del processo padre”:



catena chiusa

- E' possibile avere una comunicazione ciclica con eventuale rimozione dei processi terminati:

