Server concorrenti Unix con stato condiviso

Come affrontare il problema: nozioni e approccio generali

ver. 1.1

Sistemi Operativi – LIET
Prof. Francesco Zanichelli
francesco.zanichelli@unipr.it

Cosa occorre sapere?

1) **Come funzionano le principali primitive Unix**

- Creazione dei **processi** (fork)
- Gestione affidabile dei **segnali**
  - Gestore di uno o più segnali: tipicamente deve modificare un flag (variabile globale) che verrà considerato per modificare l'esecuzione del processo quando viene riattivato dopo il segnale (normale **gestione asincrona** dei segnali)
  - Gestione ed eventuale blocco di segnali: i segnali vanno bloccati se il processo non vuole/può riceverli in una certa fase o se li attenderà con `sigsuspend` (**comportamento sincrono**: il processo non prosegue fino a che non ha ricevuto il segnale atteso)
- Comunicazione e sincronizzazione tra processi "parenti" mediante **pipe**
- Comunicazione tra processi anche remoti mediante **socket**

Segnali

Il gestore di uno o più segnali deve modificare un flag (variabile globale) che verrà considerato per modificare l'esecuzione del processo quando viene riattivato dopo il segnale (normale **gestione asincrona** dei segnali)

```
void handler(int signo) {
sig_received=1;
}

...
sigaction(…);

do {

  if (sig_received) {
       sig_received = 0;
       … // cambia/fa qualcosa perché è arrivato un segnale
       }

  …    // fa qualcosa in ogni caso
}
```

Segnali

L'uso di `sigsuspend` è indispensabile per un processo che intende sospendersi in attesa di uno specifico segnale (**gestione sincrona**): inizialmente va bloccato il segnale di interesse, per poi sbloccarlo e attenderlo atomicamente con la `sigsuspend` quando necessario

```
...
sigaction(…);
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask,SIGINT);
sigprocmask(SIG_BLOCK,&newmask, &oldmask);
/* Da qui il segnale SIGINT e' bloccato per il processo */
...

/* Sblocco di tutti i segnali pendenti
e attesa di un qualunque segnale */
sigsuspend(&zeromask);

sigprocmask(SIG_SETMASK,&oldmask, NULL);
/* sigsuspend rimette la maschera attiva prima
della sua chiamata: se necessario va rimessa
quella ancora precedente (oldmask) */
…
```

Pipe

- L'interazione di processi tramite pipe è normalmente una comunicazione sincronizzata tra **uno scrittore e un lettore**

Ps: write(piped[1],&mesg,sizeof(mesg));

Pl: read(piped[0],&mesg,sizeof(mesg));

// la read può completarsi solo dopo il completamento della write

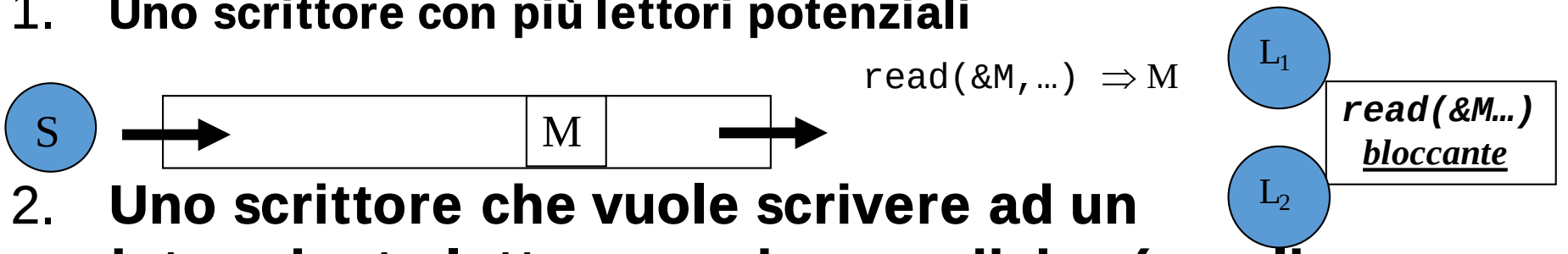
// read e write sono atomiche (per messaggi di dimensione < PIPE_BUF)

- **Cosa altro si può ottenere dalle pipe?**

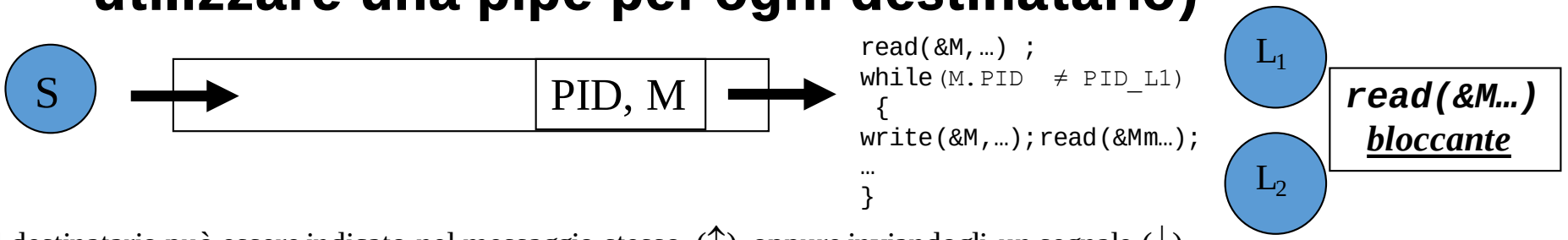
Usi "non ortodossi" delle pipe

Poiché la read è atomica, un lettore a caso riesce a leggere , l'altro si blocca

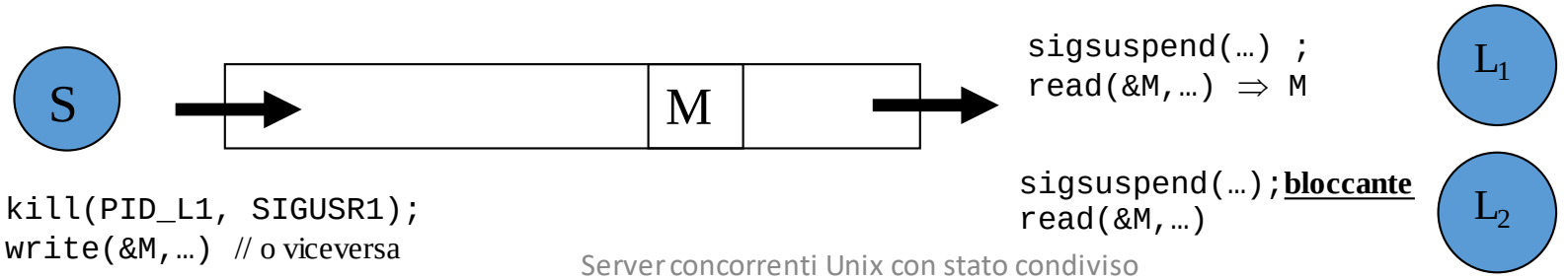
1. **Uno scrittore con più lettori potenziali**



2. **Uno scrittore che vuole scrivere ad un determinato lettore su pipe condivisa (meglio utilizzare una pipe per ogni destinatario)**



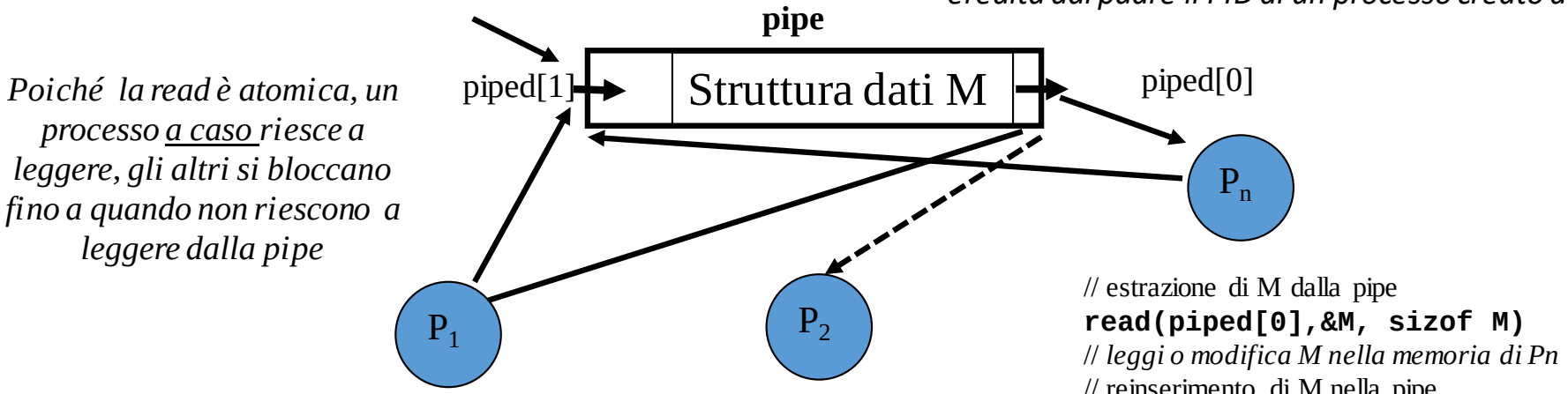
Il destinatario può essere indicato nel messaggio stesso (↑) oppure inviandogli un segnale (↓)



Usi "non ortodossi" delle pipe

3. La pipe come deposito di informazioni da condividere tra processi

Ad es. esercizio della metà dei processi che devono inviare un segnale ai processi dell'altra metà.
Occorre condividere la tabella dei PID, dato che un processo non eredità dal padre il PID di un processo creato dopo di lui.



- *E' un modo con cui realizzare una sorta di risorsa/variabile condivisa tra processi UNIX : in alternativa si può creare un processo dedicato gestore della risorsa a cui gli altri processi inviano richieste di lettura/modifica della risorsa*
- *Entrambi gli schemi sono applicabili ad un server concorrente su socket (vedi più avanti) che deve fornire un servizio basato su uno stato condiviso (ad es. operazioni sullo stesso contatore, conto, etc.) tra le diverse richieste dei clienti*

Uno schema generale di soluzione per gli esercizi UNIX

5 fasi principali con un preciso ordinamento:

1. **Controllo degli argomenti di invocazione**

2. **Selezione della politica di gestione dei segnali**
   1. per il processo padre (main) e/o per i figli che la ereditano

3. **Creazione pipe (ed eventuale inizializzazione del contenuto per utilizzarle / a come deposito di informazioni condivise tra i processi)**
   1. le pipe saranno così accessibili ai figli e ai discendenti

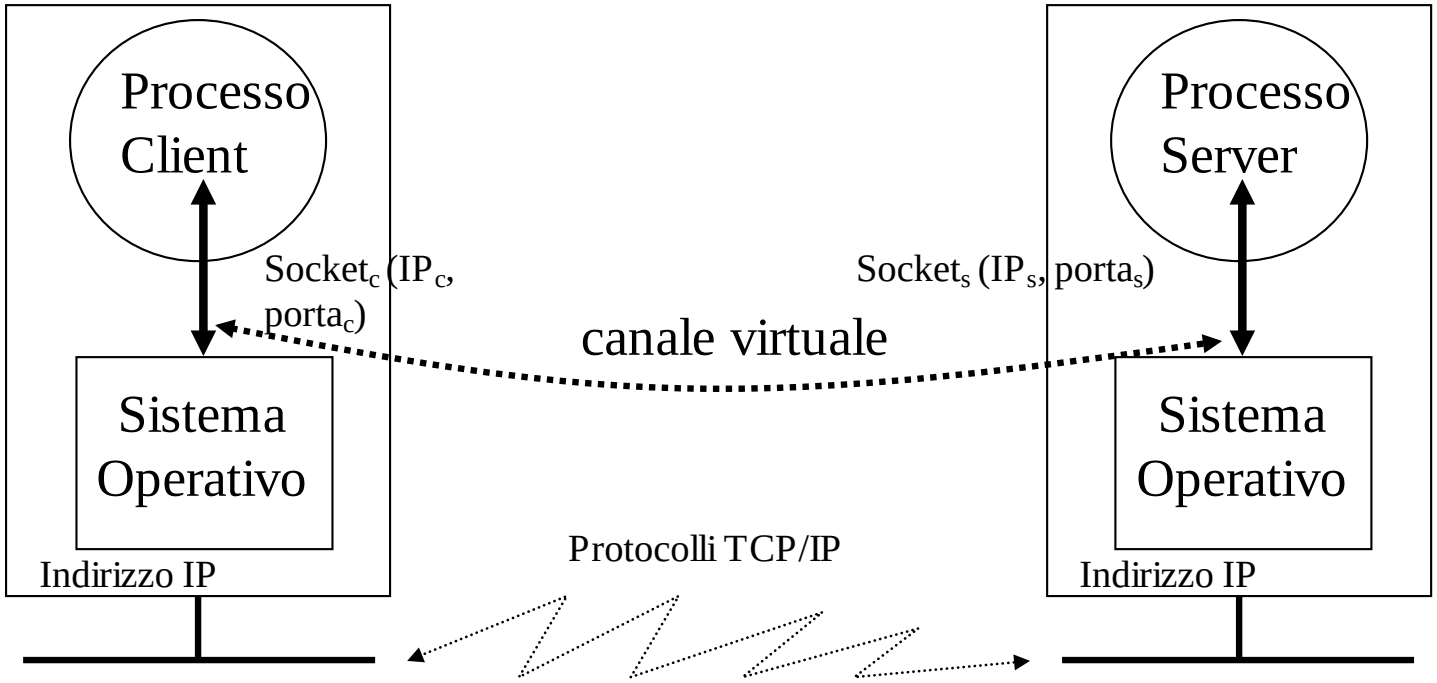
4. **Creazione dei processi figli**

5. **Esecuzione dei figli confinata in specifiche funzioni**
   - nel caso alcuni processi abbiano lo stesso comportamento si ottiene una maggiore compattezza del codice:

   $\Rightarrow$ se possibile le funzioni vanno scritte in modo parametrico rispetto ai propri argomenti e al PID del processo corrente

SOCK_STREAM nel dominio AF_INET

Prima di effettuare il trasferimento dati deve essere creata una connessione (protocollo TCP di TCP/IP)



Completata con successo la fase di creazione della connessione (socket "connessa") è sufficiente inoltrare i messaggi lungo la connessione perché raggiungano la destinazione

Creazione della connessione

Un **cliente** inizia una connessione sulla propria socket specificando l'indirizzo della socket del server:

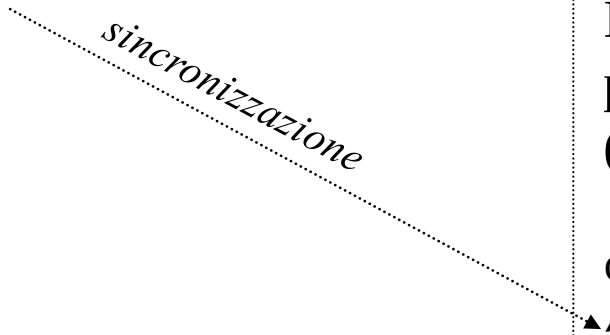
```
connect(int cli_sockfd, ...);
/* bloccante */
```

Il **server** dichiara al S.O. la sua disponibilità a ricevere connessioni sulla propria socket:

```
listen( int serv_sockfd,...) ;
/* non bloccante */
```

Il **server** attende richieste di connessioni sulla propria socket e riceve un **nuovo descrittore** (conn_sockfd) per ogni **nuova connessione**:

```
conn_sockfd=accept(int serv_sockfd,...);
/* bloccante */
```



Su socket connesse

```
write(cli_sockfd, …) ;
```

```
read(cli_sockfd, …) ;
```

```
read(conn_sockfd, …) ;
```

```
write(conn_sockfd, …) ;
```

Se la read ritorna 0 significa che la connessione è stata chiusa dal partner

Socket

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *serv_addr,
socklen_t addrlen);
```

```
#include <sys/socket.h>

int listen(int s, int backlog);
```

`backlog` specifica la dimensione massima della coda delle richieste di connessione pendenti (non ancora accettate)

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

`s` è il descrittore della socket di controllo, in `addr` (se non è NULL) viene memorizzato l'indirizzo del cliente che si è connesso. **Un valore di uscita positivo identifica il descrittore della socket connessa che va utilizzata per comunicare con il cliente che si è connesso.**

Server **NON** concorrente **(… non una grande idea…)**

- Un server su SOCK_STREAM che dopo l'accept gestisce (read/write) direttamente la connessione con il nuovo cliente
- Il server **serializza (serve una dopo l'altra) le richieste dei clienti** che si connettono, ed eventuali ritardi di un client (ad es. dell'utente al terminale) o tempi elevati di elaborazione fanno attendere i clienti in coda per il proprio turno di servizio

```
do
 { /* Attesa di una connessione */
 if((msgsock= accept(sock,(struct sockaddr *)&client,(socklen_t *)&len))<0) {
        perror("accept"); exit(-1); }
 else  {
        printf("Serving connection from %s, port %d\n",
            inet_ntoa(client.sin_addr), ntohs(client.sin_port));

        myservice(msgsock); /* Servizio specifico del server attraverso
                        la server connessa */
        close(msgsock); /* La socket connessa può essere rimossa */
                }
    }
while(1);
```

Server concorrente

- Normalmente un **server su SOCK_STREAM (cfr. i servizi TCP, ad es. ftp) crea un nuovo server figlio dedicato a gestire una nuova connessione da un cliente** mentre il server padre continua ad attendere nuove connessioni
- In questa soluzione i clienti in coda vengono serviti al più presto da un servitore dedicato, ed eventuali ritardi di un client non hanno effetto sul servizio agli altri

```
do
 { /* Attesa di una connessione */
 if((msgsock= accept(sock,(struct sockaddr *)&client,(socklen_t *)&len))<0) {
        perror("accept"); exit(-1); }
 else  {
    if(fork()==0) {
       /* Server figlio  */
        printf("Serving connection from %s, port %d\n",
            inet_ntoa(client.sin_addr), ntohs(client.sin_port));

        close(sock); /* Non interessa la socket di controllo */
        myservice(msgsock); /* Servizio specifico del server attraverso
                    la server connessa */
        close(msgsock); /* La socket connessa può essere rimossa */
        exit(0);
        }
    else /* Server padre */
        close(msgsock); /* Non interessa la socket connessa : si
                    ritorna in accept */
        }
    }
while(1);
```

Server concorrente

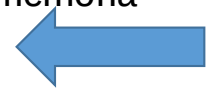
Un server concorrente è estremamente utile nei casi in cui il servizio erogato a ciascun cliente può essere erogato contemporaneamente ai client connessi, ovvero non è necessario alcun coordinamento o sincronizzazione tra i server figli :

- download/upload file
- servizi il cui risultato **è** funzione unicamente dell'input del client, ad es. risultato prodotto da funzioni dei dati ricevuti dal client

E' diverso il caso in cui sia necessario coordinare o sincronizzare l'attività tra i server figli, **basandosi su uno stato comune condiviso**, come nel caso di:

- gestione di una risorsa condivisa (ad es. contatore, conto bancario, bacheca messaggi, etc.)
- servizi il cui risultato **non è** funzione unicamente dell'input del client, ad es. risultato prodotto da funzioni dei dati ricevuti dai clienti anche in precedenza

Per realizzare un server concorrente con questo requisito non è possibile memorizzare la risorsa nella memoria di un processo (server padre e/o figli) in quanto i **processi UNIX tradizionali non condividono la memoria** e ogni modifica alla memoria di un processo è visibile solo ai processi figli che esso genera



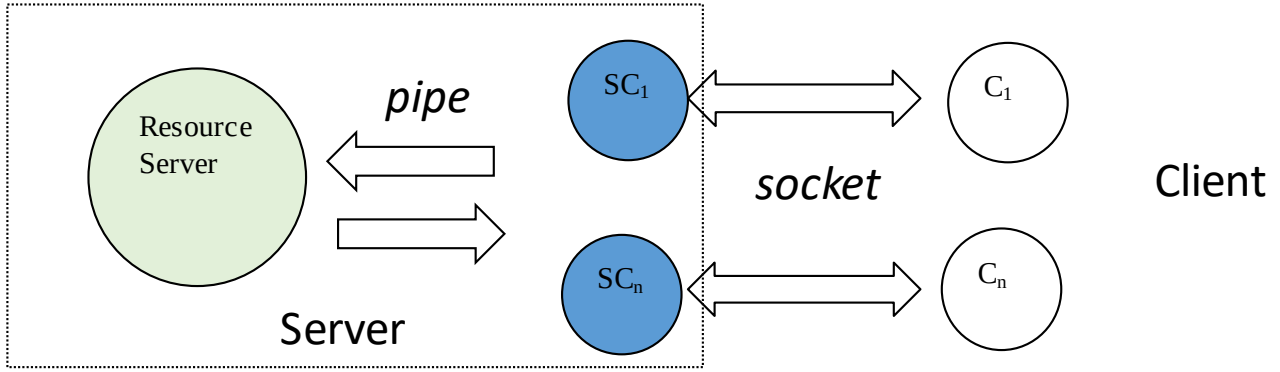
Sono quindi possibili sostanzialmente due approcci

1. **incapsulare la risorsa in processo gestore lato server (un Resource server)** con il quale i server figli interagiscono inviando richieste e ricevendo risposte: è la soluzione tipica per i sistemi a scambio di messaggi

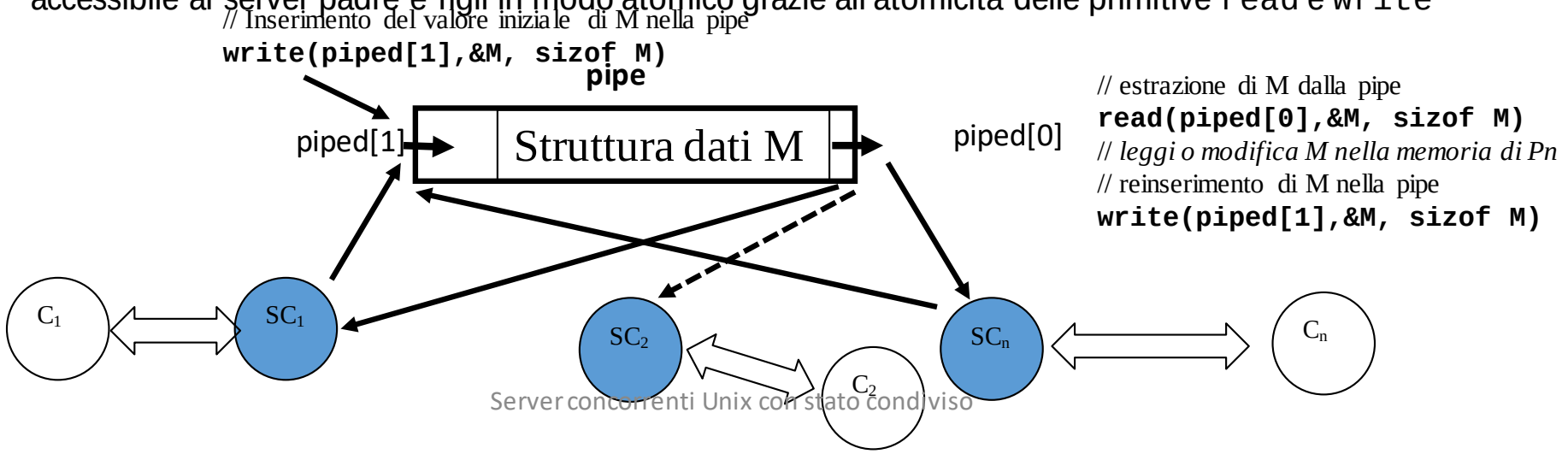
2. **simulare una "struttura dati condivisa" tramite un messaggio che viene inserito e prelevato in / da una pipe** accessibile ai server padre e figli in modo atomico grazie all'atomicità delle primitive `read` e `write`

Server concorrente

1. incapsulare la risorsa in un processo gestore lato server (Resource Server) con il quale i server figli interagiscono inviando richieste e ricevendo risposte : è la soluzione tipica del modello di interazione a scambio di messaggi

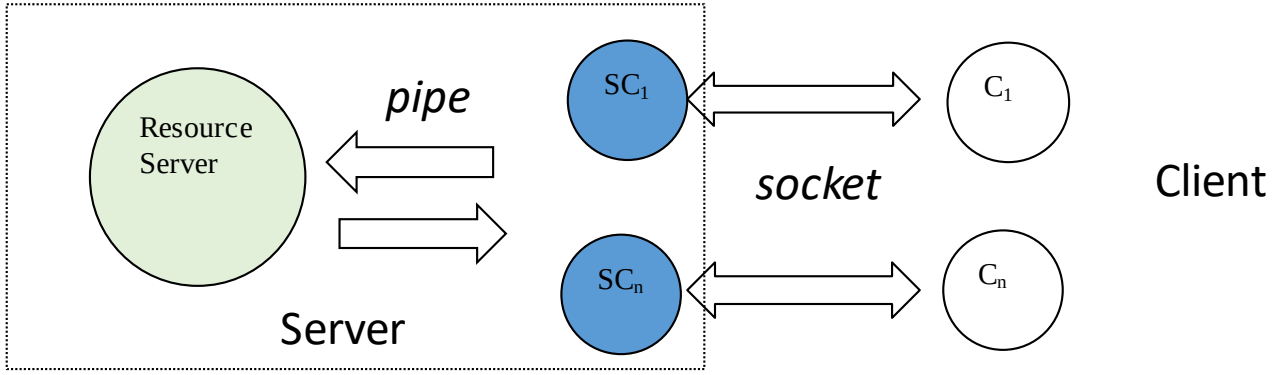


2. simulare una "struttura dati condivisa" tramite un messaggio che viene inserito e prelevato in/da una pipe accessibile ai server padre e figli in modo atomico grazie all'atomicità delle primitive `read` e `write`



1. Esempio di server concorrente con stato condiviso : utilizzo di un processo gestore

- Incapsulare la risorsa in un processo gestore lato server (Resource Server) con il quale i server figli interagiscono inviando richieste e ricevendo risposte : è la soluzione tipica del modello di interazione a scambio di messaggi



Come può il Resource Server rispondere al server figlio SC che gli ha fatto una richiesta? Si possono usare le FIFO oppure una sola pipe per le risposte inviando un segnale al processo che deve ricevere la risposta (cfr. prova Unix "gestore risorsa")

- Esempio Pstore: nell'esercizio risolto distribuito a fine corso viene creato un processo persistente (creato prima del ciclo del server concorrente) per gestire un vettore di memoria modificato sulla base delle richieste dei clienti.
  ⇒ **Pstore è un Resource Server**

- **MOLTO IMPORTANTE**
  - **Se le modifiche venissero fatte nella propria memoria da parte dei processi figli del server concorrente, non sarebbero visibili all'esterno di ciascuno di quei processi, dato che <u>di norma la memoria di ciascun processo UNIX è privata e non viene condivisa con altri processi</u>**

2. Esempio di server concorrente con stato condiviso : utilizzo di una pipe

- Simulare una "struttura dati condivisa" tramite un messaggio che viene inserito e prelevato in/da una pipe accessibile ai server padre e figli in modo atomico grazie alla indivisibilità delle primitive `read` e `write`

- **Esempio contatore : il server riceve un intero dal client e lo aggiunge al valore corrente del contatore (memorizzato come messaggio nella pipe)**

Altri esempi: esercizi biblioteca e global max

```
int contatore=0
…
write(piped[0],&contatore, sizeof(int));   // inizializza il valore del contatore memorizzato nella pipe

// figli del server concorrente
if ((pid=fork())==0) {
     read(msgsock,buffer,sizeof(buffer));  // riceve messaggio tramite socket dal client
     sscanf(buffer,"%d",&value));          // estrae un intero dal messaggio
     read(piped[0],&cont,sizeof(int));     // recupera dalla pipe il valore attuale del contatore,
                                           // altri processi che tentano la stessa read vengono bloccati, la pipe è vuota!
     cont = cont + value;                  // aggiorna il valore del contatore in memoria (è una modifica solo locale!!!)
     write(piped[1],&cont,sizeof(int));    // riscrive il valore aggiornato nella pipe a beneficio degli
                                           // altri processi che possono (uno per volta) leggere dalla pipe
     sprintf(buffer2,"%d",cont);          // prepara la risposta testuale per il client
     write(msgsock,buffer2,sizeof(buffer2));  // invia la risposta al client tramite socket
     }
```