



di.unito.it

DIPARTIMENTO  
DI INFORMATICA

DI INFORMATICA  
DIPARTIMENTO

di.unito.it

laboratorio di  
sistemi operativi

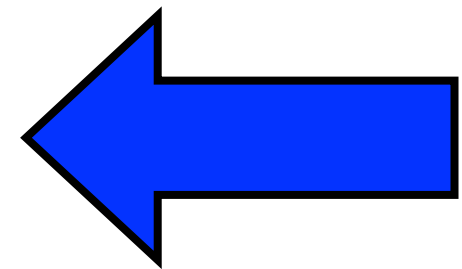
*memoria condivisa*

Marco Botta

Materiale preparato da Daniele Radicioni

# argomenti del laboratorio UNIX

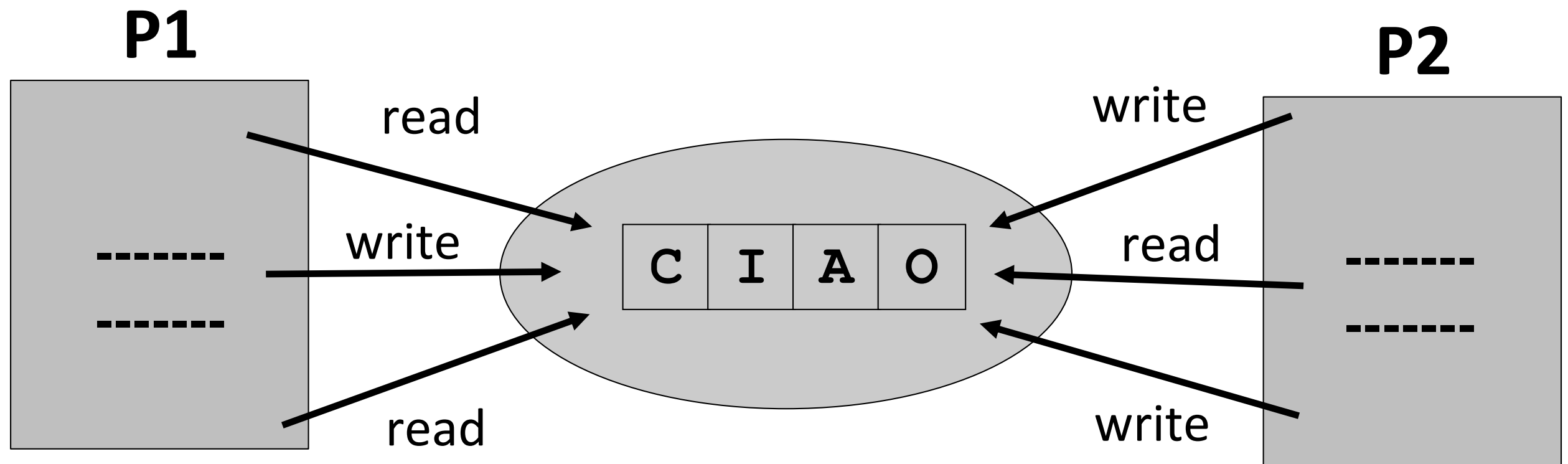
1. introduzione a UNIX;
2. integrazione C: operatori bitwise, precedenze, preprocessore, pacchettizzazione del codice, compilazione condizionale e utility make;
3. controllo dei processi;
4. segnali;
5. pipe e fifo;
6. code di messaggi;
7. semafori;
8. memoria condivisa;
9. introduzione alla programmazione bash.



- il materiale di queste lezioni è tratto:
  - dai lucidi del Prof. Gunetti degli anni scorsi;
  - Michael Kerrisk, *The Linux Programming interface - a Linux and UNIX® System Programming Handbook*, No Starch Press, San Francisco, CA, 2010;
  - W. Richard Stevens (Author), Stephen A. Rago, *Advanced Programming in the UNIX® Environment* (2nd Edition), Addison-Wesley, 2005;

# introduzione

- La memoria condivisa (shared memory, **SM**) consente a due o più processi di **condividere la stessa** regione (detta anche **segmento**) di memoria fisica.
- Un processo copia i dati all'interno della memoria **condivisa**; quei **dati** **divengono immediatamente disponibili** a tutti gli altri processi che condividono lo stesso segmento.
  - Si tratta di uno strumento che fornisce **una IPC veloce in confronto a tecniche come i pipe o le code di messaggi**, in cui il processo mittente copia i dati da un buffer dello spazio utente nella memoria, e in cui il ricevente effettua una copia nella direzione inversa.



Area di memoria primaria condivisa da più processi.

# uso della MC

- Poiché l'utilizzo della **SM** non è mediato dal kernel, tipicamente è necessario predisporre qualche metodo di sincronizzazione
  - in questo modo i processi non accedono simultaneamente alla memoria condivisa, come nel caso in cui due processi eseguono aggiornamenti simultanei, o in cui un processo legge dei dati dalla **SM** mentre un altro li sta modificando.

# spazi di indirizzamento

- di norma ogni processo possiede uno *spazio di indirizzamento logico* separato dagli altri processi;
  - un segmento di memoria condivisa può essere invece letto e/o scritto da due o più processi, e permette quindi un rapido scambio di informazioni.
- quindi, ogni processo usa il segmento di *SM* come se fosse una *normale porzione del proprio spazio di indirizzamento logico*, che però è *fisicamente in comune a più processi*.

# uso della MC

- un figlio creato con una *fork()* eredita i segmenti di *SM* a disposizione del genitore. Quindi la *SM* fornisce uno strumento semplice per l'IPC fra genitore e figli.
  - Durante una *exec()*, tutti i segmenti attaccati sono staccati (*detached*). NB: staccati, *non distrutti*!
  - I segmenti di *SM* sono anche *automaticamente staccati al momento della terminazione* dei processi.



# uso della MC: il ciclo di operazioni

1. Chiamata ***shmget()*** per creare un nuovo segmento di ***SM*** o per ottenere l'identificatore di un segmento esistente (i.e., un segmento creato da un altro processo).
2. Uso di ***shmat()*** per attaccare il segmento di ***SM***; cioè, per rendere il segmento in questione parte della memoria virtuale del processo chiamante.
  - La memoria condivisa può essere trattata come qualsiasi altra porzione di memoria indirizzabile dall'interno del programma. Al fine di riferirsi alla memoria condivisa, il programma usa il valore ***addr*** restituito dalla chiamata ***shmat()***, che è un puntatore all'inizio del segmento di ***SM*** nello spazio di indirizzi virtuale del processo.

# uso della MC: il ciclo di operazioni

3. Chiamata *shmdt()* per staccare il segmento di *SM*.  
Dopo tale chiamata, il processo non può più fare riferimento alla *SM*.
  - Questo passo (è opzionale, e) occorre automaticamente alla terminazione del processo.
4. Chiamata *shmctl()* per cancellare il segmento di *SM*.
  - Il segmento sarà effettivamente distrutto solo dopo che tutti i processi correntemente attaccati lo avranno staccato. Un solo processo effettua la cancellazione.

# Creating or Opening a Shared Memory Segment

```
#include <sys/types.h> /* For portability */  
#include <sys/shm.h>
```

```
int shmget(key_t key, size_t size, int shmflg);
```

Returns shared memory segment identifier on success, or -1 on error

- L'argomento *key* è una chiave generata usando tipicamente il valore *IPC\_PRIVATE* o una key restituita da *ftok()*.

```
#include <sys/types.h> /* For portability */  
#include <sys/shm.h>
```

```
int shmget(key_t key, size_t size, int shmflg);
```

Returns shared memory segment identifier on  
success, or -1 on error

- Quando utilizziamo la *shmget()* per creare un nuovo segmento di *SM*, *size* specifica un intero positivo che indica la dimensione del segmento, espressa in bytes.
  - Il kernel alloca la *SM* in multipli della dimensione delle pagine di sistema, quindi in pratica *size* viene arrotondata al multiplo successivo della dimensione della pagina.
- Se stiamo usando *shmget()* per ottenere l'identificatore di un segmento esistente, *size* è un argomento privo di effetto sul segmento, ma deve essere minore o uguale alla dimensione del segmento.

```
#include <sys/types.h> /* For portability */
#include <sys/shm.h>
```

```
int shmget(key_t key, size_t size, int shmflg);
```

Returns shared memory segment identifier on  
success, or -1 on error

- L'argomento *shmflg* svolge le stesse operazioni comuni alle altre chiamate IPC get, [specificando i permessi](#) da associare al nuovo segmento, o da verificare su un segmento esistente. Inoltre, zero o più fra i seguenti flag possono essere concatenati in OR (|):
  - [\*IPC\\_CREAT\*](#). Se non esiste un segmento con la *key* specificata, crea un nuovo segmento.
  - [\*IPC\\_EXCL\*](#). Se è stato specificato *IPC\_CREAT*, e un segmento con la *key* specificata esiste già, fallisce con errore *EEXIST*.

# area condivisa e tipi

- La *shmget()* funziona in modo simile alla *malloc*, con la differenza che l'area allocata è accessibile a più processi. Si può quindi prelevare spazio facendo riferimento a diversi tipi di dati:

- Tipi di dati fondamentali:

```
shmget ( ..., sizeof(int), ... );
```

- Array:

```
shmget ( ..., sizeof(char) *N, ... );
```

- Strutture:

```
shmget ( ..., sizeof(struct libro), ... );
```

- Array di tipi derivati:

```
shmget ( ..., sizeof(struct dato) *N, ... );
```

# Using Shared Memory

```
#include <sys/types.h> /* For portability */  
#include <sys/shm.h>
```

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Returns address at which shared memory is attached on  
success, or (void \*) -1 on error

- La system call ***shmat()*** attacca un'area di memoria identificata da ***shmid*** allo spazio di indirizzamento del processo.



```
#include <sys/types.h> /* For portability */
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);

    Returns address at which shared memory is attached on
    success, or (void *) -1 on error
```

- L'argomento *shmaddr* e l'impostazione del **SHM\_RND** bit nella bit-mask *shmflg* controllano il modo in cui il segmento è attaccato:
  - Se *shmaddr* è **NULL**, allora il segmento è attaccato all'indirizzo appropriato dal kernel. Questo è il modo migliore per attaccare un segmento.
- Specificare un valore non-**NULL** per *shmaddr* non è raccomandabile:
  - Infatti riduce la **portabilità** di un'applicazione. Un indirizzo valido in una implementazione UNIX può non essere valido in un'altra.
  - Un tentativo di attaccare un segmento di **SM** ad un particolare indirizzo fallirà se quell'indirizzo è già utilizzato.



```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

## valore di ritorno di *shmat()*

- *shmat()* restituisce l'indirizzo al quale il segmento di *SM* è attaccato.
  - Questo valore può essere trattato *come un normale puntatore C*; il segmento può essere trattato come qualsiasi altra parte della memoria virtuale del processo.
- Tipicamente, *assegnamo il valore di ritorno di shmat()* ad un puntatore a qualche struttura definita nel programma, al fine di imporre quella struttura sul segmento:

```
struct shmseg *shmp;
```

```
shmp = (struct shmseg *) shmat(shmid, NULL, 0);
```

```
if (shmp == (void *) -1)  
    errExit("shmat");
```

```
#include <sys/types.h> /* For portability */  
#include <sys/shm.h>
```

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Returns address at which shared memory is attached on  
success, or (void \*) -1 on error

- Per attaccare un segmento di **SM** per un accesso read-only, specifichiamo il flag **SHM\_RDONLY** nel *shmflg*.
  - Tentativi di aggiornare i contenuti di un segmento disponibile solo in lettura produrranno un segmentation fault (segnale **SIGSEGV**).
- Se **SHM\_RDONLY** non è specificato, la memoria può essere sia letta sia modificata.

```
#include <sys/types.h> /* For portability */
#include <sys/shm.h>
```

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Returns address at which shared memory is attached on  
success, or (void \*) -1 on error

Value	Description
<i>SHM_RDONLY</i>	Attach segment read-only
<i>SHM_REMAP</i>	Replace any existing mapping at shmaddr

# detaching segments

- Quando un processo non accede più a un segmento di **SM**, può chiamare la system call ***shmdt()*** per staccare il segmento dal proprio spazio di indirizzi virtuale.
- L'argomento ***shmaddr*** identifica il segmento da staccare. Dovrebbe essere un valore restituito da una precedente chiamata ***shmat()***.

# Detaching segments

```
#include <sys/types.h> /* For portability */  
#include <sys/shm.h>
```

```
int shmdt(const void *shmaddr);
```

Returns 0 on success, or -1 on error

- La *shmdt()* sgancia l'area di memoria condivisa dallo spazio degli indirizzi del processo chiamante;
  - *shmaddr* è l'indirizzo del punto di accesso, restituito dalla *shmat()*;
- NB: sganciamento  $\neq$  cancellazione.
  - La *cancellazione* è eseguita per mezzo della *shmctl()*, con il comando *IPC\_RMID*.

```

00 int main(int argc, char** argv) {
01     int shmid_1, shmid_2, return_val;
02     char *stringa_1, *stringa_2;
03     char msg[] = "ciao a tutti!";
04
05     shmid_1 = shmget(MYKEY, sizeof(char)*SHMSZ, IPC_CREAT|0666);
06     stringa_1 = (char *)shmat(shmid_1, NULL, 0);
07
08     snprintf(stringa_1, sizeof(msg), "%s", msg);
09     return_val = shmdt(stringa_1);
10
11     // un altro processo che si attacchi alla stessa area
12     // di memoria condivisa potrà leggerne il contenuto
13     shmid_2 = shmget(MYKEY, sizeof(char)*SHMSZ, 0);
14
15     stringa_2 = (char *)shmat(shmid_2, NULL, 0);
16     printf("%s\n", stringa_2); // leggo il contenuto della SM
17
18     shmctl(shmid_2, IPC_RMID, 0);
19     exit(EXIT_SUCCESS);
20 }

```

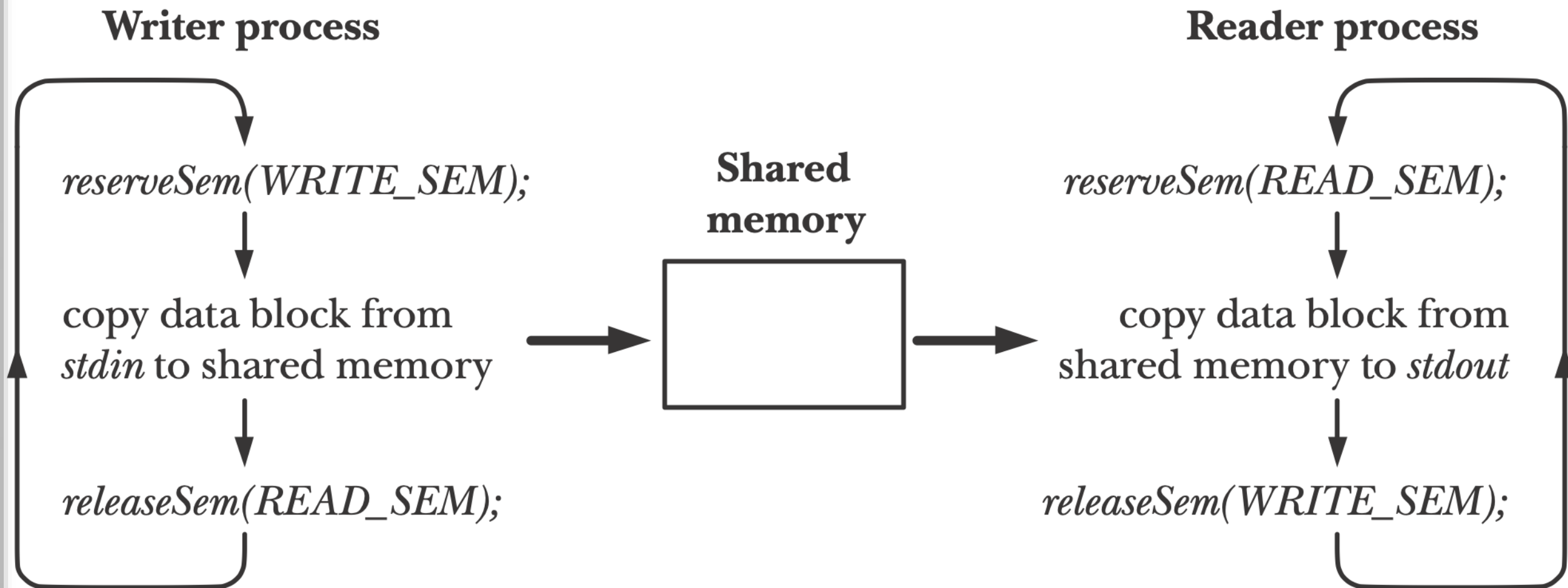
# Esempio: *lettore* e *scrittore*

- L'applicazione consiste di due programmi: il *lettore* e lo *scrittore*.
  - Lo *scrittore* legge blocchi di dati da standard input e li scrive su un segmento di **SM**.
  - Il *lettore* legge blocchi di dati dal segmento di memoria condivisa allo standard output.

# Esempio: lettore e scrittore

- I due programmi utilizzano un paio di semafori che implementano il **binary semaphore protocol** (costituito dalle funzioni *initSemAvailable()*, *initSemInUse()*, *reserveSem()*, and *releaseSem()*) per assicurare che:
  - un solo processo alla volta acceda al segmento di **SM**; e che
  - i processi si alternino nell'accesso al segmento (cioè lo scrittore scriva alcuni dati, quindi il lettore li legga, quindi lo scrittore ne scriva altri, etc.).





- lo scrittore inizializza i due semafori in modo da essere il primo dei due processi ad accedere al segmento di SM;
- cioè, il semaforo dello scrittore è inizialmente disponibile, e il semaforo del lettore è inizialmente in uso.

# Shared Memory Control Operations

```
#include <sys/types.h> /* For portability */
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

Returns 0 on success, or -1 on error
```

- La system call **shmctl()** esegue un insieme di operazioni di controllo sul segmento di **SM** identificato da **shmid**.
- L'argomento **cmd** specifica l'operazione da eseguire.
- L'argomento **buf** è richiesto dalle operazioni **IPC\_STAT** e **IPC\_SET**, e dovrebbe essere **NULL** per le altre.

```
#include <sys/types.h> /* For portability */
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shm_id_ds *buf);

Returns 0 on success, or -1 on error
```

- **IPC\_RMID**. Marca il segmento di **SM** e l'associata struttura **shm\_id\_ds** per la cancellazione. Se nessun processo ha il segmento attaccato, la cancellazione è immediata; diversamente, il segmento è rimosso dopo che tutti i processi lo hanno staccato.
    - In alcune applicazioni possiamo assicurarci che un segmento sia rimosso al momento della terminazione dell'applicazione marcandolo per la cancellazione immediatamente dopo che tutti i processi lo hanno attaccato al proprio spazio di indirizzi con la **shmat()**.
- Questa operazione è analoga all'**unlinking** dei file una volta che li abbiamo aperti (vedere **man 2 unlink**!).

```
#include <sys/types.h> /* For portability */
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shm_id_ds *buf);

Returns 0 on success, or -1 on error
```

- **IPC\_STAT**. Copia la struttura *shm\_id\_ds* associata a questo segment nel buffer puntato da *buf*.
- **IPC\_SET**. Aggiorna i membri della struttura *shm\_id\_ds* associata a questo segmento con il buffer puntato da *buf*.

# *shmid\_ds* data structure

```
struct shmid_ds {  
    struct ipc_perm shm_perm; /*Ownership e permissions */  
    size_t shm_segsz; /* Size of segment in bytes */  
    time_t shm_atime; /* Time of last shmat() */  
    time_t shm_dtime; /* Time of last shmdt() */  
    time_t shm_ctime; /* Time of last change */  
    pid_t shm_cpid; /* PID of last creator */  
    pid_t shm_lpid; /* PID of last shmat() / shmdt() */  
    shmatt_t shm_nattch; /* Number of currently attached  
                           processes */  
}
```