
Inter Process Communications

Most slides have been adapted from
«The Linux Programming interface: A Linux and UNIX® System Programming Handbook», Michael Kerrisk

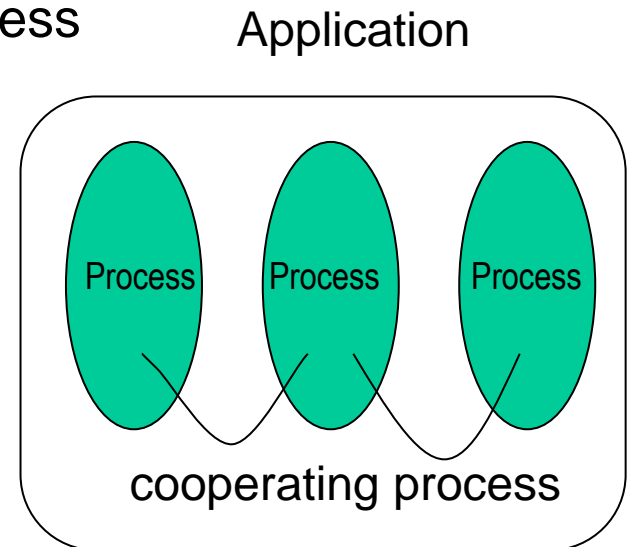
Sistemi di Calcolo 2
Riccardo Lazzeretti

Process Address Space

- A process can only access its address space
- Each process has its own address space
- Kernel can access everything

Cooperating Processes and the need for Interprocess Communication

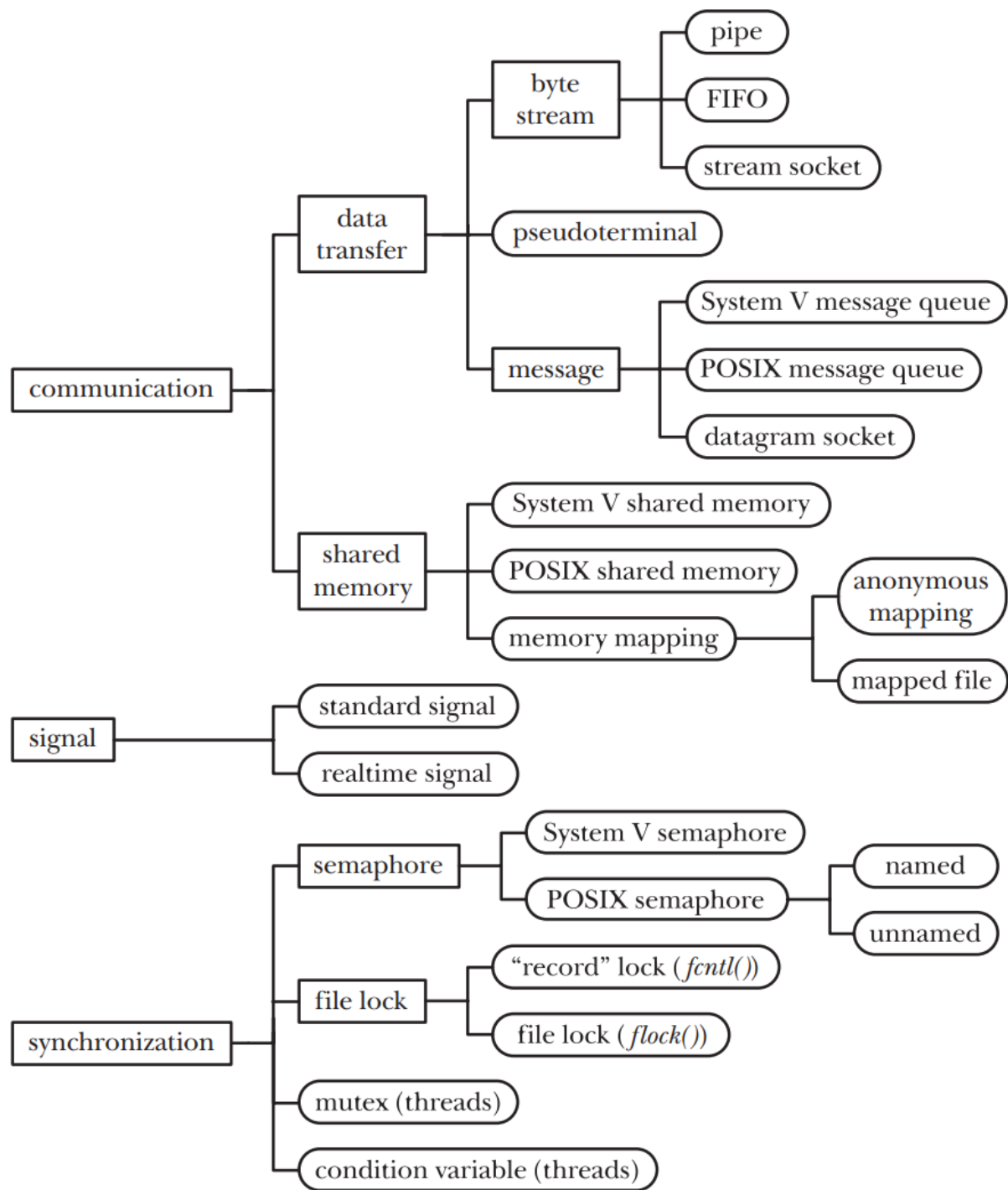
- Processes within a system may be **independent** or **cooperating**
 - Independent** process cannot affect or be affected by the execution of another process
 - Cooperating** process can affect or be affected by the execution of another process
- Reasons for process cooperation
 - Information **sharing**
 - Computation **speed-up**
 - Modularity** (application will be divided into modules/sub-tasks)
 - Convenience** (may be better to work with multiple processes)



The overall application is designed to consist of cooperating processes

Inter-process Communication (IPC)

Mechanism for processes to communicate and to synchronize their actions.

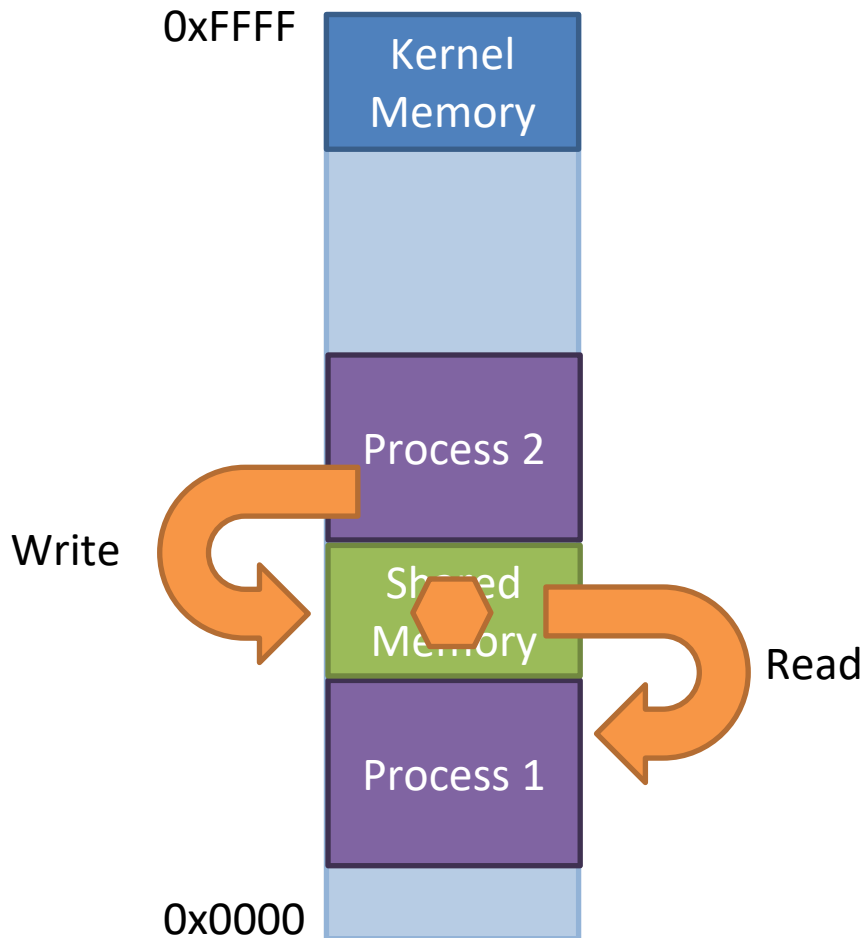


IPC Mechanisms

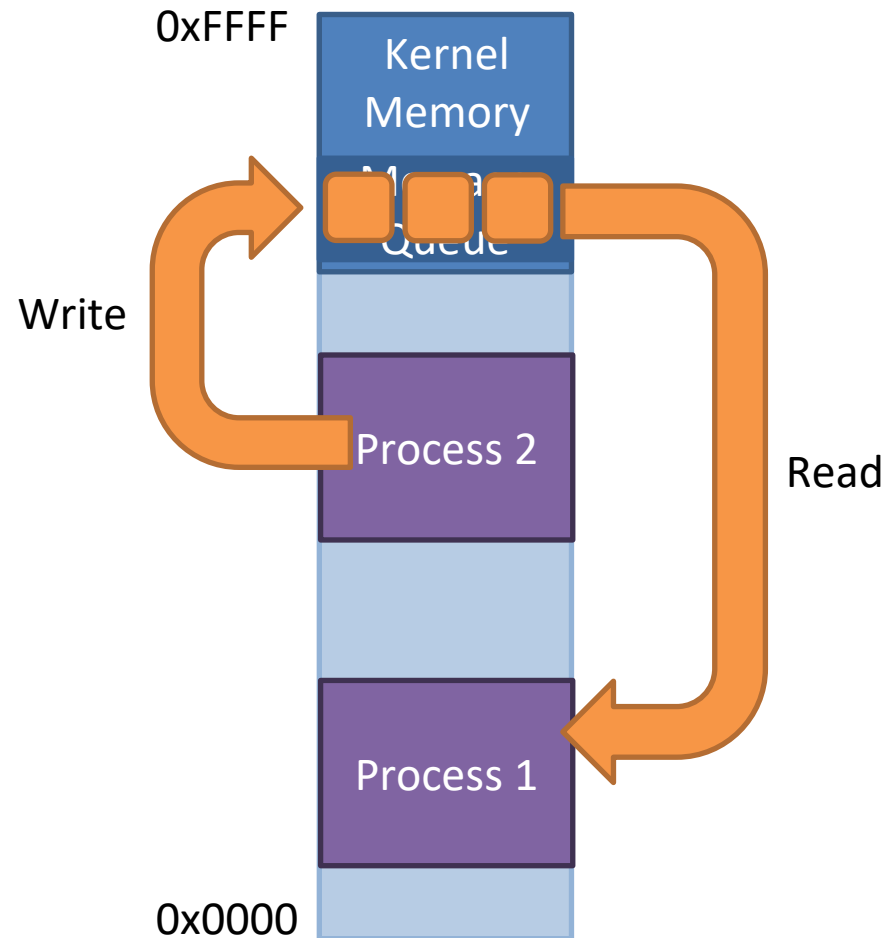
- Cooperating processes require a facility/mechanism for inter-process communication (IPC)
- There are two basic IPC models provided by most systems:
 - 1) Shared memory model
processes use a shared memory to exchange data
 - 2) Message passing model
processes send messages to each other through the kernel

Communication models

Shared Memory

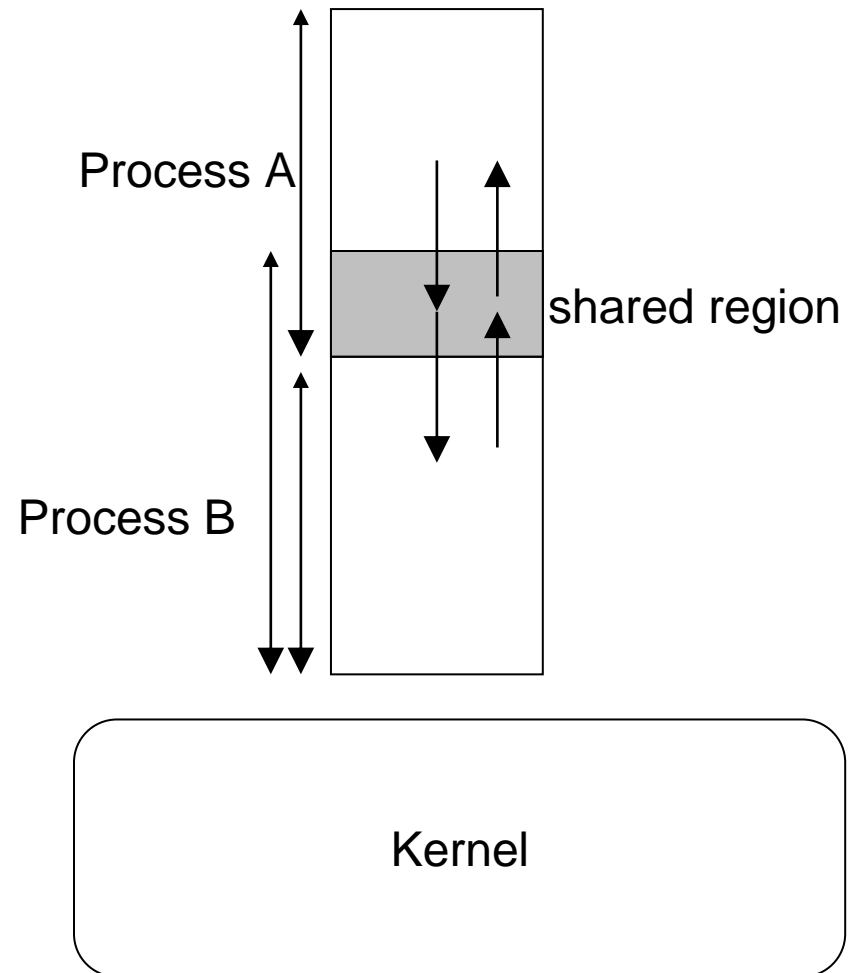


Message Passing



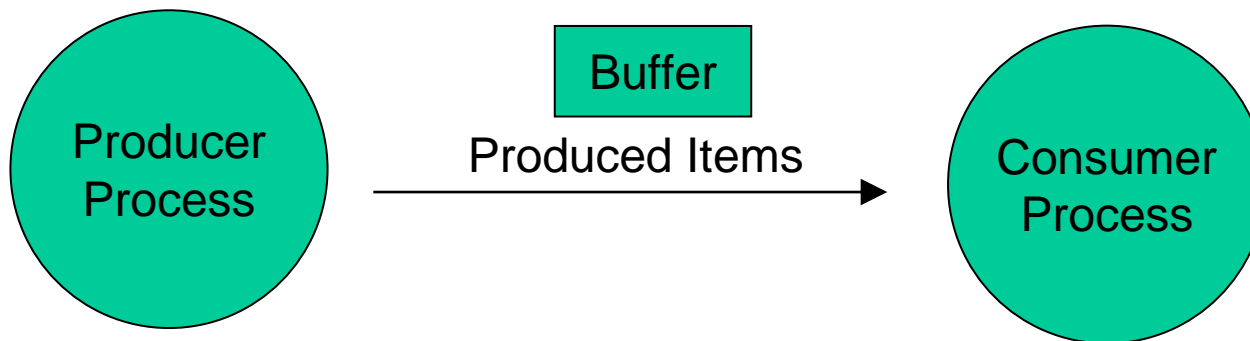
Shared Memory IPC Mechanism

- A **region of shared memory** is established between (among) two or more processes.
 - via the help of the operating system kernel (i.e. **system calls**).
- Processes can **read** and **write** shared memory region (segment) directly as ordinary memory access (**pointer** access)
 - During this time, kernel is not involved.
 - Hence it is fast



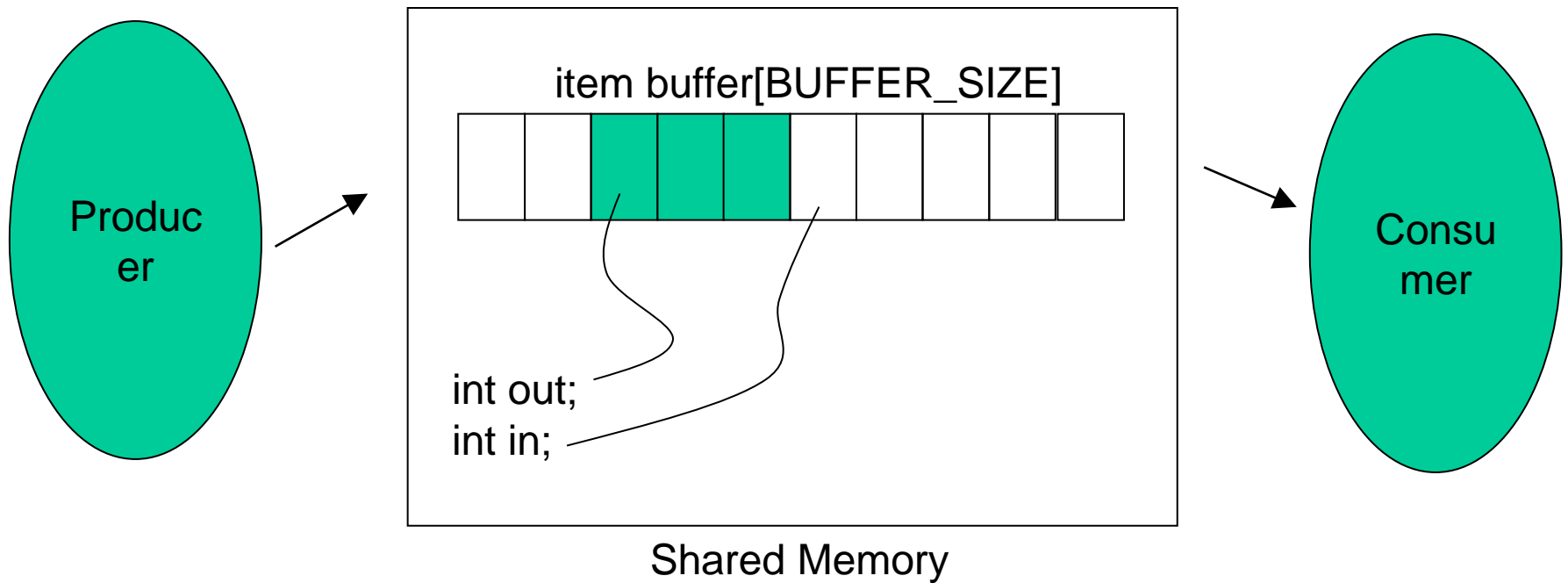
Shared Memory IPC Mechanism

- To illustrate use of an IPC mechanism, a general model problem, we use the **producer-consumer problem with bounded buffer**



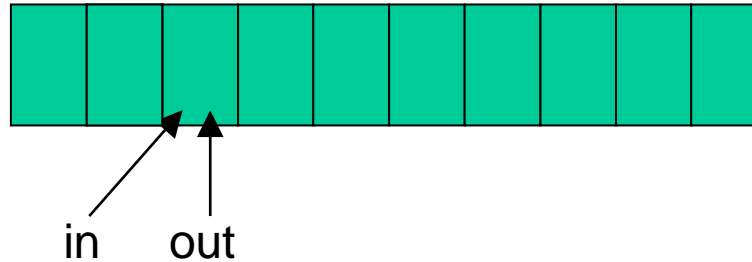
We can solve this problem via shared memory IPC mechanism

Buffer State in Shared Memory



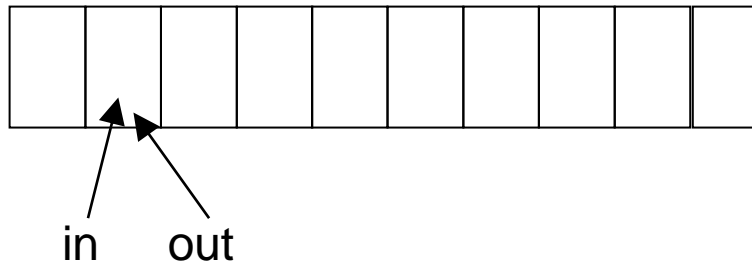
Buffer State in Shared Memory

Buffer Full



`in == out; sem_empty.val == 0; sem_filled.val == BUFFER_SIZE`

Buffer Empty



`in == out; sem_empty.val == BUFFER_SIZE; sem_filled.val == 0`

Bounded-Buffer - Producer and Consumer Code

```
while (true) {  
    /* Produce an item */  
    sem_wait(sem_empty); %named semaphores  
    sem_wait(sem_cs);  
    writeToBuffer (ITEM);  
    in = (in+1) % BUFFER_size;  
    sem_signal(sem_cs);  
    sem_signal(sem_filled);  
}
```

Producer

Consumer

```
while (true) {  
    sem_wait(sem_filled);  
    sem_wait(sem_cs);  
    readFromBuffer (ITEM);  
    out = (out +1) % BUFFER_size;  
    sem_signal(sem_cs);  
    sem_signal(sem_empty);  
    /* Consume the item */  
}
```

Buffer (an array)
in, out integer variable

Shared Memory

Posix Shared Memory API

- **shm_open()** – create and/or open a shared memory page
 - Returns a file descriptor for the shared page
 - We can create a private shared memory
 - Only child processes can access the SHM too
- **ltruncate()** or **ftruncate()** – limit the size of the shared memory page
- **mmap()** – map the memory page into the processes address space
 - Now you can read/write the page using a pointer
- **close()** – close a file descriptor
- **shm_unlink()** – remove a shared page
 - Processes with open references may still access the page

```
/* Program to write some data in shared memory */
```

```
int main() {
```

```
    const int SIZE = 4096;
```

```
    const char * name = "MY_PAGE";
```

```
    const char * msg = "Hello World!";
```

```
    int shm_fd;
```

```
    char * ptr;
```

```
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

```
    ftruncate(shm_fd, SIZE);
```

```
    ptr = (char *) mmap(0, SIZE, PROT_WRITE,  
                        MAP_SHARED, shm_fd, 0);
```

```
    sprintf(ptr, "%s", msg);
```

```
    close(shm_fd);
```

```
    return 0;
```

```
}
```

```
/* Program to read some data from shared memory */
```

```
int main() {
```

```
    const int SIZE = 4096;
```

```
    const char * name = "MY_PAGE";
```

```
    int shm_fd;
```

```
    char * ptr;
```

```
    shm_fd = shm_open(name, O_RDONLY, 0666);
```

```
    ptr = (char *) mmap(0, SIZE, PROT_READ,  
        MAP_SHARED, shm_fd, 0);
```

```
    printf("%s\n", ptr);
```

```
    close(shm_fd);
```

```
    shm_unlink(shm_fd);
```

```
    return 0;
```

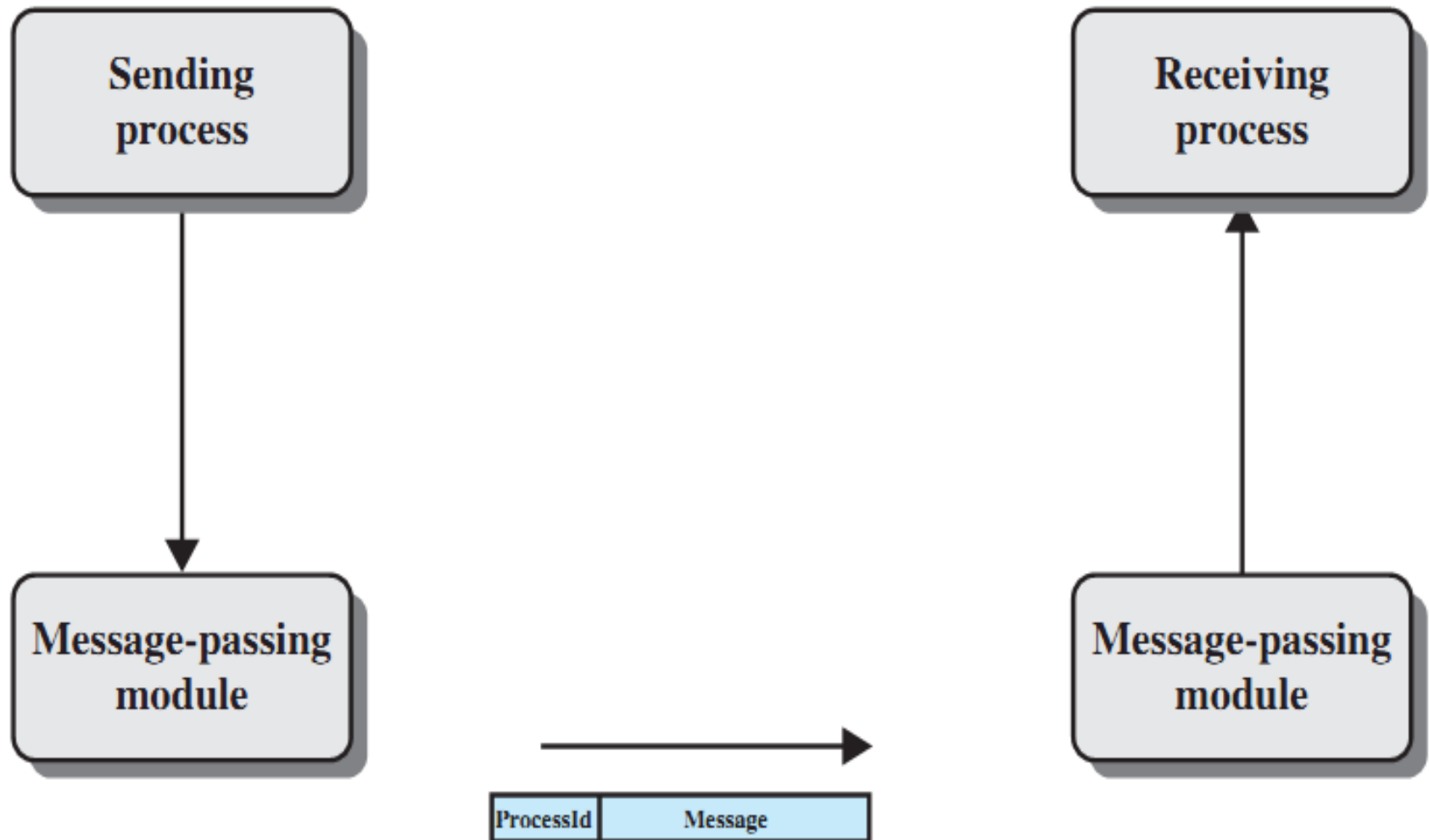
```
}
```

Message Passing IPC Mechanisms

- Message system – processes communicate with each other without resorting to shared variables.
- We have at least two primitives:
 - **send**(*destination, message*) or **send**(*message*)
 - **receive**(*source, message*) or **receive**(*message*)
- Message size is fixed or variable.

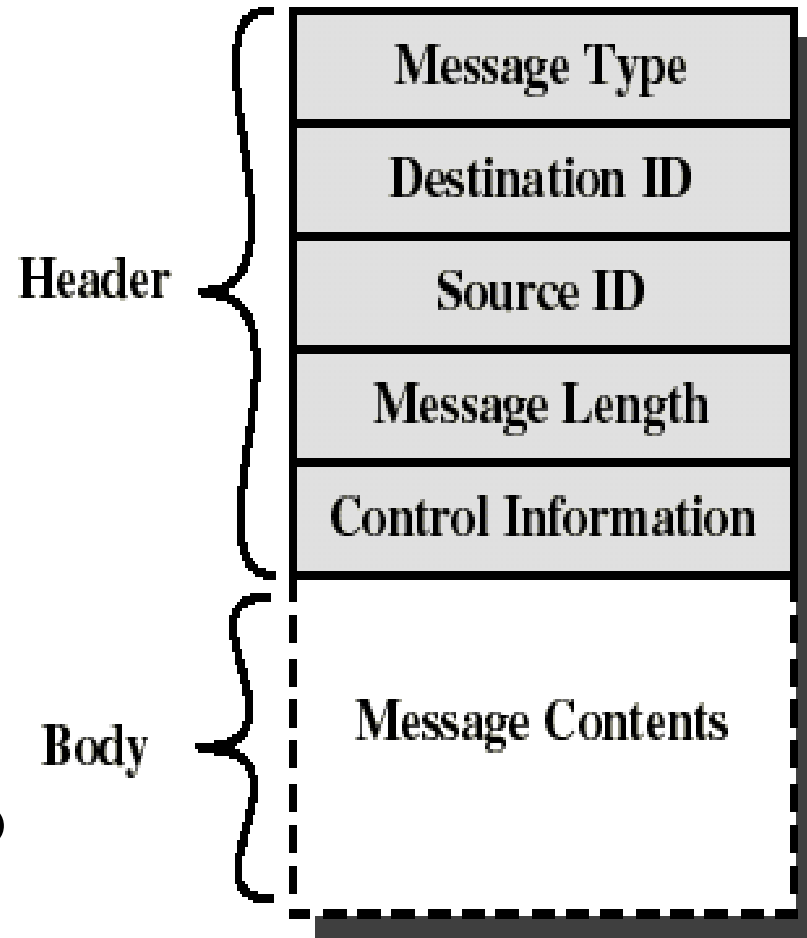


Basic Message-passing Primitives



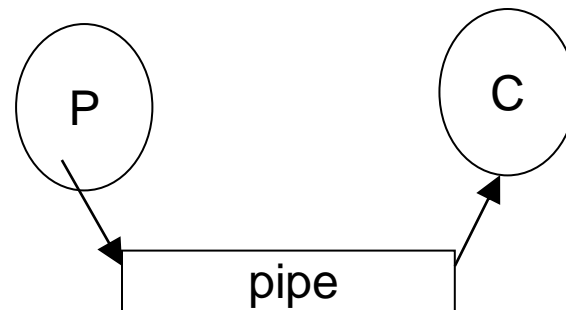
Message format

- Consists of header and body of message.
- In Unix: no ID, only message type.
- Control info:
 - what to do if run out of buffer space.
 - sequence numbers.
 - priority.
- **Queuing discipline: usually FIFO but can also include priorities.**



Message Passing: pipes

- Piped and Named-Pipes (FIFOs)
- In Unix/Linux:
 - A **pipe** enables one-way communication between a parent and child
 - It is easy to use.
 - When process terminates, pipe is removed automatically
 - **pipe()** system call



Concetti di base sulle PIPE (1/2)

- permettono a più processi di comunicare come se stessero accedendo a dei file sequenziali
- il termine “pipe” significa tubo in Inglese, e la comunicazione avviene in modo unidirezionale
- una volta lette, le informazioni spariscono dalla PIPE e non possono più ripresentarsi
 - a meno che non vengano riscritte all'altra estremità del tubo
- a livello di OS, le PIPE non sono altro che buffer di dimensione più o meno grande (solitamente 4096 byte)
 - quindi è possibile che un processo venga bloccato se tenta di scrivere su una PIPE piena

Concetti di base sulle PIPE (2/2)

- i processi che usano una PIPE devono essere “relazionati” come conseguenza di operazioni `fork()`
- le named PIPE (FIFO) permettono la comunicazione anche tra processi non relazionati
- in sistemi UNIX l’uso delle PIPE avviene attraverso la nozione di descrittore

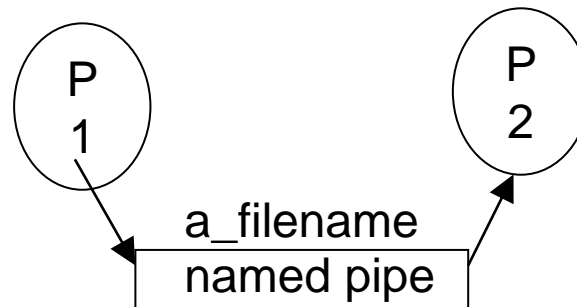
IPC methods: Pipes



- File-like abstraction for sending data between processes
 - Can be read or written to, just like a file
 - Permissions controlled by the creating process
- Two types of pipes
 - Named pipe: any process can attach as long as it knows the name
 - Typically used for long lived IPC
 - Unnamed/anonymous pipe: only exists between a parent and its children
- Full or half-duplex
 - Can one or both ends of the pipe be read?
 - Can one or both ends of the pipe be written?

IPC methods: named-pipes (FIFOs)

- A **named-pipe** is called FIFO.
- It has a name
- When processes terminate, it is not removed automatically
- No need for parent-child relationship
- bidirectional
- Any two process can create and use named pipes.



PIPE nei Sistemi UNIX

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

Descrizione invoca la creazione di una PIPE

Argomenti fd: puntatore ad un buffer di due interi

- fd[0] : descrittore di lettura dalla PIPE
- fd[1]: descrittore di scrittura sulla PIPE

Restituzione -1 in caso di fallimento, 0 altrimenti

fd[0] è un canale aperto in lettura che consente ad un processo di leggere dati da una PIPE

fd[1] è un canale aperto in scrittura che consente ad un processo di immettere dati sulla PIPE

fd[0] e fd[1] possono essere usati come normali descrittori di file tramite le chiamate read() e write()

Letture e scrittura

- Un processo che provi a leggere da una pipe vuota rimane bloccato finché non ci sono dati disponibili
- Un processo che provi a scrivere su una pipe piena rimane bloccato finché un altro processo non ha letto (e rimosso) abbastanza dati da permettere la scrittura
- Una pipe ha una dimensione massima equivalente a 16 pagine di memoria in linux
 - 65,536 byte in un sistema con una page size di 4096 byte
- E' possibile rendere la lettura e la scrittura non bloccanti (non in questo corso)
- E' possibile cambiare la dimensione della pipe

Attenzione alle scritture!!!

- Una scrittura di $n \leq \text{PIPE_BUF}$ byte è atomica
 - $\text{PIPE_BUF} = 4096$ byte
 - Si blocca se non ci sono n byte disponibili
- Una scrittura di $n > \text{PIPE_BUF}$ byte si potrebbe inframezzare con altre `write()`
 - Il processo rimane bloccato finchè non sono stati scritti tutti gli n byte
- Nel caso di `write` non bloccanti
 - Se $n \leq \text{PIPE_BUF}$ ma non ci sono n byte disponibili `write()` fallisce e setta `errno`
 - Se $n > \text{PIPE_BUF}$ potrebbe risultare in una scrittura parziale (ma non in un errore)

Avvertenze

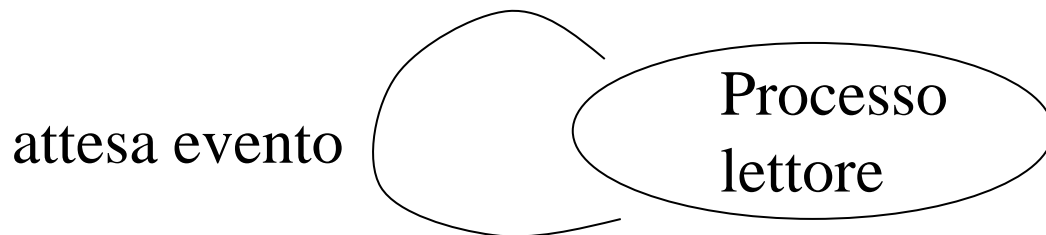
- le PIPE non sono dispositivi fisici, ma logici, pertanto viene spontaneo chiedersi come un processo sia in grado di vedere la fine di un “file” su una PIPE
- per convenzione, ciò avviene quando tutti i processi scrittori che condividevano il descrittore `fd[1]` lo hanno chiuso
- in questo caso la chiamata `read()` effettuata da un lettore restituisce zero come notifica dell'evento che tutti gli scrittori hanno terminato il loro lavoro
- allo stesso modo, un processo scrittore che tenti di scrivere sul descrittore `fd[1]` quando tutte le copie del descrittore `fd[0]` siano state chiuse (non ci sono lettori sulla PIPE), riceve il segnale `SIGPIPE`, altrimenti detto «Broken pipe»

PIPE e deadlock

Per fare in modo che tutto funzioni correttamente e non si verifichino situazioni di deadlock, è necessario che tutti i processi chiudano i descrittori di PIPE che non gli servono, usando una normale `close()`

Si noti che ogni processo lettore che erediti la coppia `(fd[0],fd[1])` deve chiudere la propria copia di `fd[1]` prima di mettersi a leggere da `fd[0]` dichiarando così di non essere uno scrittore

Se così non facesse, l'evento "tutti gli scrittori hanno terminato" non potrebbe mai avvenire se il lettore è impegnato a leggere, e si potrebbe avere un deadlock



Un esempio: trasferimento stringhe tramite PIPE

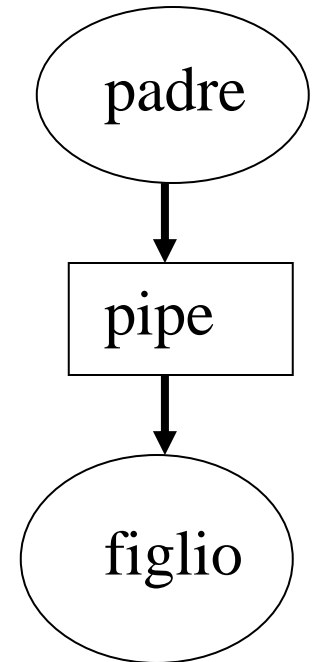
```
#include <stdio.h>
#define Errore_(x) { puts(x); exit(1); }

int main(int argc, char *argv[]) {
    char messaggio[30];    int  pid, status, fd[2];

    ret = pipe(fd); /* crea una PIPE */
    if ( ret == -1 )
        Errore_("Errore nella creazione pipe");

    pid = fork(); /* crea un processo figlio */
    if ( pid == -1 ) Errore_("Errore nella fork");

    if ( pid == 0 ) { /* processo figlio: lettore */
        close(fd[1]); /* il lettore chiude fd[1] */
        while( read(fd[0], messaggio, 30) > 0 )
            printf("letto messaggio: %s", messaggio);
        close(fd[0]);
        _exit();
    }
}
```



.....continua

```
/* processo padre: scrittore */
else {

    close(fd[0]);
    puts("digitare testo da trasferire (quit per
terminare):");

    do {
        fgets(messaggio, 30, stdin);
        write(fd[1], messaggio, 30);
        printf("scritto messaggio: %s", messaggio);
    } while( strcmp(messaggio, "quit\n") != 0 );

    close(fd[1]);
    wait(&status);
}

}
```

Named PIPE (FIFO) in Sistemi UNIX

```
int mkfifo(char *name, int mode)
```

Descrizione invoca la creazione di una FIFO

Argomenti 1) *name: nome della FIFO da creare
 2) mode: specifica i permessi di accesso alla FIFO

Restituzione -1 in caso di fallimento, 0 altrimenti

La rimozione di una FIFO dal file system avviene mediante la chiamata di sistema `unlink()`

Avvertenze

- normalmente, l'apertura di una FIFO è bloccante, nel senso che il processo che tenta di aprirla in lettura (scrittura) viene bloccato fino a quando un altro processo non la apre in scrittura (lettura)
- se si vuole inibire questo comportamento è possibile aggiungere il flag `O_NONBLOCK` al valore del parametro `mode` passato alla system call `open()` su di una FIFO
- ogni FIFO deve avere sia un lettore che uno scrittore: se un processo tenta di scrivere su una FIFO che non ha un lettore esso riceve il “segnale `SIGPIPE`” da parte del sistema operativo

Un esempio: client/server tramite FIFO

Processo server

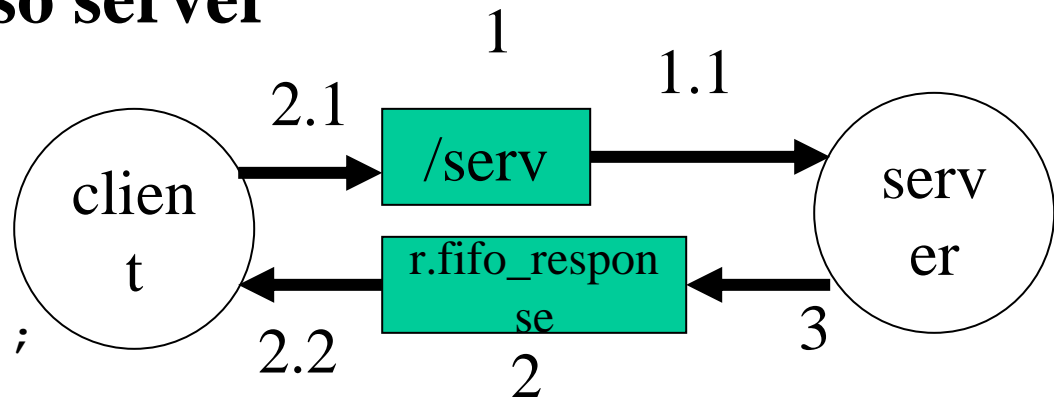
```
#include <stdio.h>
```

```
#include <fcntl.h>
```

```
typedef struct {  
    long type;  
    char fifo_response[20];  
} request;
```

```
int main(int argc, char *argv[]){  
    char *response = "fatto";  
    int pid, fd, fdc, ret;  
    request r;
```

```
    ret = mkfifo("/serv", 0666);  
    if ( ret == -1 ) {  
        printf("Errore nella chiamata mkfifo\n");  
        exit(1);  
    }
```

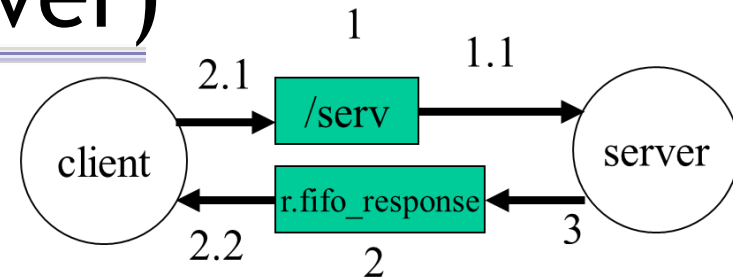


1

.....continua (Processo Server)

```
fd = open("/serv", O_RDONLY);
```

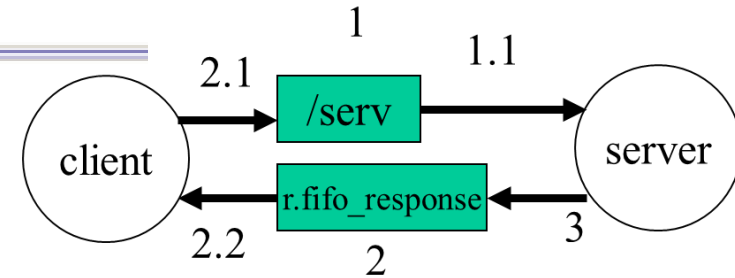
```
while(1) {  
    ret = read(fd, &r, sizeof(request));  
    if (ret != 0) {  
        printf("Richiesto un servizio (fifo di restituzione = %s)\n",  
               r.fifo_response);  
        /* switch sul tipo di servizio */  
        sleep(10); /* emulazione di ritardo per il servizio */  
        fdc = open(r.fifo_response, O_WRONLY);  
        write(fdc, response, 20);  
        close(fdc);  
        exit(0);  
    } /* end if (ret != 0) */  
}  
}
```



1.1

3

Processo client



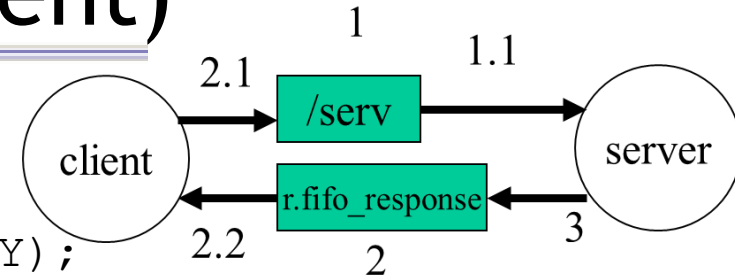
```
#include <stdio.h>
#include <fcntl.h>
typedef struct { long type; char fifo_response[20]; } request;

int main(int argc, char *argv[]) {
    int pid, fd, fdc, ret;    request r;    char response[20];

    printf("Selezionare un carattere alfabetico minuscolo: ");
    scanf("%s", r.fifo_response);
    if (r.fifo_response[0] > 'z' || r.fifo_response[0] < 'a' ) {
        printf("carattere selezionato non valido, ricominciare operazione\n");
        exit(1);
    }
    r.fifo_response[1] = '\0';
    ret = mkfifo(r.fifo_response, 0666);
    if ( ret == -1 ) {
        printf("\n servente sovraccarico - riprovare \n");
        exit(1);
    }
}
```

2

.....continua (Processo Client)



```
fd = open("/serv", O_WRONLY);
if ( fd == -1 ) {
    printf("\n servizio non disponibile \n");
    ret = unlink(r.fifo_response);
    exit(1);
}
```

```
write(fd, &r, sizeof(request));
close(fd);
```

2.1

2.2

```
fdc = open(r.fifo_response, O_RDONLY);
read(fdc, response, 20);
printf("risposta = %s\n", response);
```

```
close(fdc);
unlink(r.fifo_response);
```

```
}
```

Confronto tra pipe e FIFO

- L'apertura di una pipe comporta ottenere sia il descrittore per la lettura che quello per la scrittura
 - Bisogna poi chiudere quello che non viene usato
- L'apertura di una FIFO avviene in lettura o in scrittura
- Nella gestione si differenziano solo nella fase di apertura e chiusura
 - Lettura e scrittura avvengono nello stesso modo

