

Sistemi Operativi

Il Facoltà di Ingegneria - Cesena

a.a 2012/2013

docente: Alessandro Ricci

[modulo 2a]

PROCESSI

SOMMARIO

- Nozione di processo
 - stato dei processi
 - PCB e tabella dei processi
- Scheduling di processi
 - tipi di scheduler e code di scheduling
- Operazioni e chiamate di sistema sui processi
 - creazione e terminazione
 - fork
 - comunicazione fra processi (IPC)
 - scambio di messaggi e memoria condivisa
 - esempi
 - pipe
 - memoria condivisa
 - IPC via rete
 - socket
 - esempi in Java API
 - RPC ed RMI
 - esempi in Java API

NOZIONE DI PROCESSO

- Un programma in esecuzione prende il nome di **processo**
 - entità 'attiva', la cui esecuzione è ad opera di un processore (la CPU) e la cui struttura e attività sono descritte da un certo programma o porzione di programma
 - a differenza del processo, il relativo programma è un'entità passiva, come rappresentazione (o insieme di istruzioni) interpretabili dall'esecutore del processo stesso
 - sinonimi: *task*, *job*
- Generalmente c'è una corrispondenza 1:1 fra *applicazioni* e processi
 - ogni applicazione eseguita su un computer è un processo
 - in alcuni casi tuttavia una medesima applicazione può lanciare più processi
- Ogni processo è identificato a tempo di esecuzione da un **PID** (process identifier) assegnatogli dal sistema operativo
 - numero intero monotono crescente

VISUALIZZIAMO I PROCESSI CORRENTI:

PROGRAMMA `ps`

- **ps** è un programma di sistema nei sistemi UNIX che consente di visualizzare lo stato dei processi eseguiti da una shell
 - Varie opzioni: `-a` per visualizzare processi di tutti gli utenti, `-al` per visualizzare dettagli...
- Esempio della slide seguente:
 - da shell viene eseguito il comando `ps`, che visualizza i processi attivi nel contesto della stessa shell.
 - viene poi mandato in esecuzione un programma scritto in C - sorgente **TestProcess.c**, sorgente riportato in seguito
 - per mandare in esecuzione parallela alla shell stessa il programma viene utilizzato `&` al termine della riga
 - il programma semplicemente esegue una `sleep` di 500 secondi (sospendendo la propria esecuzione per 500 secondi)
 - eseguendo nuovamente `ps` (con opzione `-al` per vedere dettagli) si vede il processo in esecuzione
 - ne viene lanciata un'altra istanza
 - notare il PID diverso
 - mediante il comando **kill** viene poi forzata la terminazione di uno dei due processi
 - specificando il pid del processo

PROGRAMMA TestProcess.c

- Codice del programma lanciato nell'esempio precedente (contenuto nel materiale modulo-2a-materiale)

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

int main(int argc, char* argv[]){

    printf("Started.\n");
    sleep(500);
    printf("Completed.\n");

}
```

ESECUZIONI DEL COMANDO ps

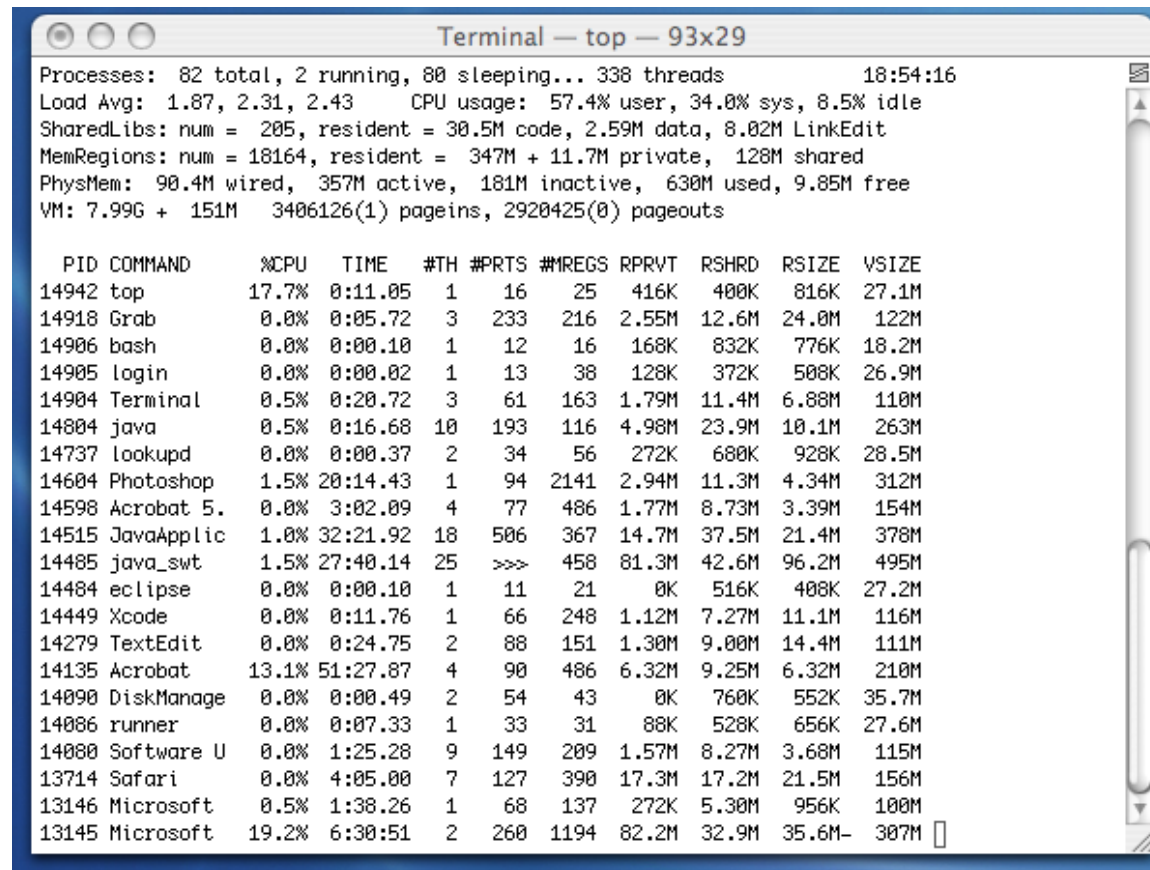
```
Terminal — bash — 144x36
DEIS-Alessandro-Notebook:~/Courses/2006-2007/SISOP/Appunti delle lezioni/modulo-2a-materiale-1.0 aricci$ gcc -o TestProcess.bin TestProcess.c
DEIS-Alessandro-Notebook:~/Courses/2006-2007/SISOP/Appunti delle lezioni/modulo-2a-materiale-1.0 aricci$ ps -all
UID    PID  PPID  CPU  PRI  NI       VSZ    RSS  WCHAN  STAT  TT      TIME COMMAND
0      1784  423   0    31   0       27572  640  -      Ss    p0      0:00.01 login -pf aricci
501    1785  1784  0    31   0       27724  820  -      S     p0      0:00.06 -bash
0      1863  1785  0    31   0       27340  460  -      R+    p0      0:00.00 ps -all
DEIS-Alessandro-Notebook:~/Courses/2006-2007/SISOP/Appunti delle lezioni/modulo-2a-materiale-1.0 aricci$ ./TestProcess.bin &
[1] 1864
DEIS-Alessandro-Notebook:~/Courses/2006-2007/SISOP/Appunti delle lezioni/modulo-2a-materiale-1.0 aricci$ Started.

DEIS-Alessandro-Notebook:~/Courses/2006-2007/SISOP/Appunti delle lezioni/modulo-2a-materiale-1.0 aricci$ ps -all
UID    PID  PPID  CPU  PRI  NI       VSZ    RSS  WCHAN  STAT  TT      TIME COMMAND
0      1784  423   0    31   0       27572  640  -      Ss    p0      0:00.01 login -pf aricci
501    1785  1784  0    31   0       27724  820  -      S     p0      0:00.06 -bash
501    1864  1785  0    31   0       27372  640  -      S     p0      0:00.00 ./TestProcess.bin
0      1865  1785  0    31   0       27340  460  -      R+    p0      0:00.00 ps -all
DEIS-Alessandro-Notebook:~/Courses/2006-2007/SISOP/Appunti delle lezioni/modulo-2a-materiale-1.0 aricci$ ./TestProcess.bin &
[2] 1866
DEIS-Alessandro-Notebook:~/Courses/2006-2007/SISOP/Appunti delle lezioni/modulo-2a-materiale-1.0 aricci$ Started.

DEIS-Alessandro-Notebook:~/Courses/2006-2007/SISOP/Appunti delle lezioni/modulo-2a-materiale-1.0 aricci$ ps -all
UID    PID  PPID  CPU  PRI  NI       VSZ    RSS  WCHAN  STAT  TT      TIME COMMAND
0      1784  423   0    31   0       27572  640  -      Ss    p0      0:00.01 login -pf aricci
501    1785  1784  0    31   0       27724  820  -      S     p0      0:00.07 -bash
501    1864  1785  0    31   0       27372  640  -      S     p0      0:00.00 ./TestProcess.bin
501    1866  1785  0    31   0       27372  640  -      S     p0      0:00.00 ./TestProcess.bin
0      1867  1785  0    31   0       27340  460  -      R+    p0      0:00.00 ps -all
DEIS-Alessandro-Notebook:~/Courses/2006-2007/SISOP/Appunti delle lezioni/modulo-2a-materiale-1.0 aricci$ kill -KILL 1864
DEIS-Alessandro-Notebook:~/Courses/2006-2007/SISOP/Appunti delle lezioni/modulo-2a-materiale-1.0 aricci$ ps -all
UID    PID  PPID  CPU  PRI  NI       VSZ    RSS  WCHAN  STAT  TT      TIME COMMAND
0      1784  423   0    31   0       27572  640  -      Ss    p0      0:00.01 login -pf aricci
501    1785  1784  0    31   0       27724  820  -      S     p0      0:00.07 -bash
501    1866  1785  0    31   0       27372  640  -      S     p0      0:00.00 ./TestProcess.bin
0      1868  1785  0    31   0       27340  460  -      R+    p0      0:00.00 ps -all
[1]- Killed                  ./TestProcess.bin
DEIS-Alessandro-Notebook:~/Courses/2006-2007/SISOP/Appunti delle lezioni/modulo-2a-materiale-1.0 aricci$
```

MONITORAGGIO PROCESSI IN UNIX

- Altro programma per visualizzare processi è **top**
 - permette di *monitorare* lo stato dei processi, visualizzandone le informazioni su standard output, aggiornandole man mano



```
Terminal - top - 93x29
Processes: 82 total, 2 running, 80 sleeping... 338 threads          18:54:16
Load Avg: 1.87, 2.31, 2.43    CPU usage: 57.4% user, 34.0% sys, 8.5% idle
SharedLibs: num = 205, resident = 30.5M code, 2.59M data, 8.02M LinkEdit
MemRegions: num = 18164, resident = 347M + 11.7M private, 128M shared
PhysMem: 90.4M wired, 357M active, 181M inactive, 630M used, 9.85M free
VM: 7.99G + 151M  3406126(1) pageins, 2920425(0) pageouts

  PID COMMAND      %CPU   TIME    #TH  #PRTS  #MREGS  RPRVT  RSHRD  RSIZE  VSIZE
14942 top           17.7%  0:11.05  1    16     25    416K   400K   816K   27.1M
14918 Grab          0.0%   0:05.72  3    233    216    2.55M  12.6M  24.0M  122M
14906 bash          0.0%   0:00.10  1     12     16    168K   832K   776K   18.2M
14905 login         0.0%   0:00.02  1     13     38    128K   372K   508K   26.9M
14904 Terminal      0.5%   0:20.72  3     61    163    1.79M  11.4M  6.88M  110M
14804 java          0.5%   0:16.68  10   193    116    4.98M  23.9M  10.1M  263M
14737 lookupd       0.0%   0:00.37  2     34     56    272K   680K   928K   28.5M
14604 Photoshop     1.5%  20:14.43  1     94   2141    2.94M  11.3M  4.34M  312M
14598 Acrobat 5.    0.0%   3:02.09  4     77    486    1.77M  8.73M  3.39M  154M
14515 JavaApplic    1.0%  32:21.92  18   506   367    14.7M  37.5M  21.4M  378M
14485 java_swt      1.5%  27:40.14  25  >>>   458    81.3M  42.6M  96.2M  495M
14484 eclipse        0.0%   0:00.10  1     11     21     0K    516K   408K   27.2M
14449 Xcode          0.0%   0:11.76  1     66    248    1.12M  7.27M  11.1M  116M
14279 TextEdit      0.0%   0:24.75  2     88    151    1.30M  9.00M  14.4M  111M
14135 Acrobat       13.1%  51:27.87  4     90    486    6.32M  9.25M  6.32M  210M
14090 DiskManage   0.0%   0:00.49  2     54     43     0K    760K   552K   35.7M
14086 runner        0.0%   0:07.33  1     33     31     88K   528K   656K   27.6M
14080 Software U    0.0%   1:25.28  9    149   209    1.57M  8.27M  3.68M  115M
13714 Safari        0.0%   4:05.00  7    127   390    17.3M  17.2M  21.5M  156M
13146 Microsoft     0.5%   1:38.26  1     68    137    272K   5.30M  956K   100M
13145 Microsoft     19.2%  6:30:51  2    260  1194    82.2M  32.9M  35.6M- 307M
```

PARAMETRI D'INGRESSO E VARIABILI D'AMBIENTE

- In generale ad ogni processo è associato un certo insieme di *parametri d'ingresso* e un certo insieme di *variabili d'ambiente*
 - i parametri d'ingresso sono specificati - tipicamente dall'utente - nel momento in cui il programma viene mandato in esecuzione
 - le variabili d'ambiente rappresentano informazioni dell'environment esterno al processo, utili alla sua esecuzione

PARAMETRI DI INGRESSO IN PROGRAMMI C

- In una programma scritto in C, i parametri di ingresso sono recuperabili direttamente a partire dagli argomenti della funzione main
 - il primo è il numero di parametri passati + 1 (include anche il nome dell'applicazione)
 - il secondo è l'array di stringhe che rappresentano i parametri + nome applicazione (che ha indice 0)
- Esempio di programma che stampa in stdout i parametri passati:

```
#include<stdio.h>

int main(int argc, char* argv[]){
    int i;

    printf("Nome del programma lanciato: %s\n",argv[0]);
    printf("Numero parametri dell'applicazione: %d\n",argc-1);
    for (i=1; i<argc; i++){
        printf("Argomento %d : %s \n",i,argv[i]);
    }
}
```

VARIABILI D'AMBIENTE IN POSIX

- Le variabili d'ambiente sono gestite come lista, rappresentata da un array di puntatori a stringhe

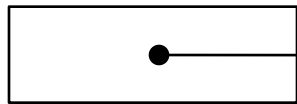
name=value

- che definiscono una variabile d'ambiente di nome *name* e di valore *value*.
 - Esempio:
HOME=/Users/aricci
- Per convenzione i nomi delle variabili d'ambiente sono in maiuscolo
- L'indirizzo della lista delle variabili d'ambiente è contenuto nella variabile globale **environ**, definita come

```
extern char** environ;
```

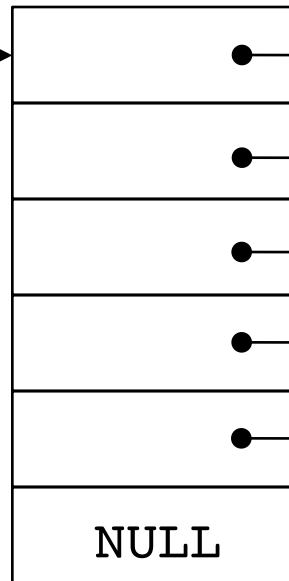
ESEMPIO DI UN POSSIBILE AMBIENTE

Puntatore alle
variabili d'ambiente



`environ`

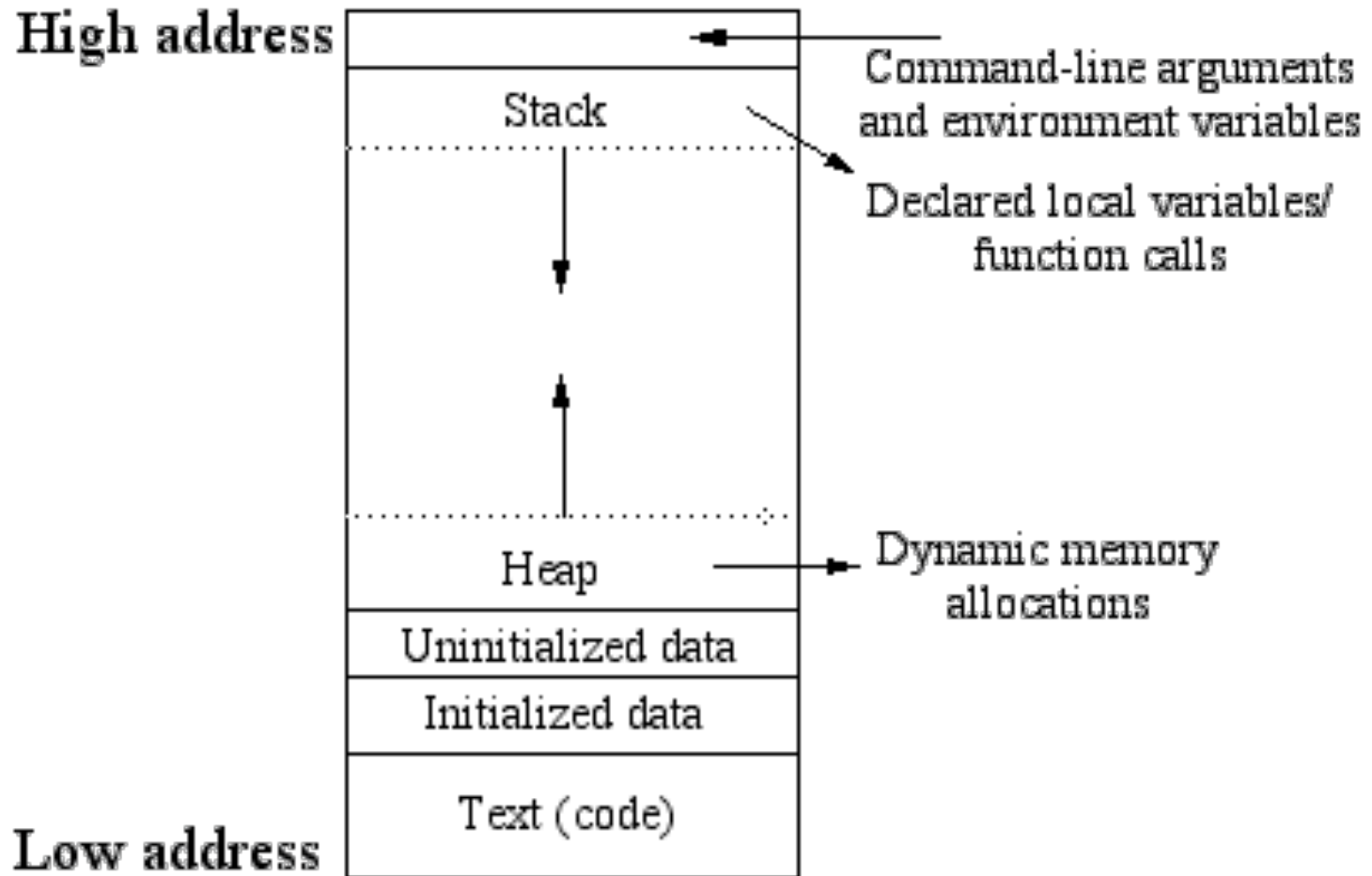
Lista delle variabili
d'ambiente



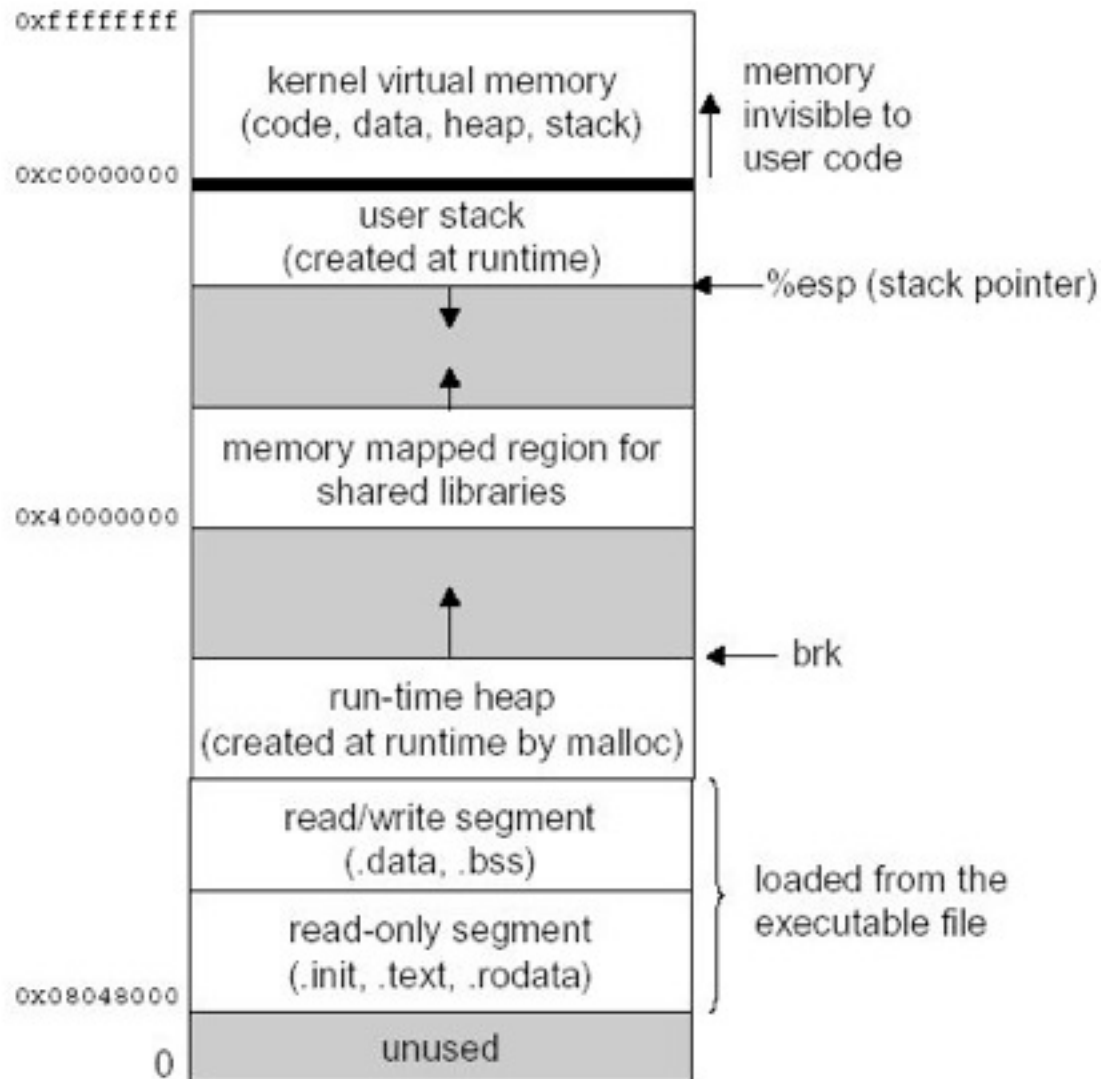
Contenuto (stringhe)
delle variabili

`HOME=/Users/aricci\0`
`PATH=:/bin:/usr/bin\0`
`SHELL=/bin/sh\0`
`USER=aricci\0`
`LOGNAME=aricci\0`

MEMORY LAYOUT DI UN PROCESSO IN UNIX



LAYOUT PROCESSO IN LINUX



UN ESEMPIO

```
#include<stdio.h>
#include<stdlib.h>
int d;

int sum(int a, int b){
    int s;
    printf("[sum] Indirizzo del parametro a: %p \n",&a);
    printf("[sum] Indirizzo della variabile locale s: %p\n",&s);
    return a + b;
}

int main(){
    int a,b;
    a = 5;
    d = 7;

    printf("Indirizzo di d: %p \n",&d);
    printf("Indirizzo di a: %p \n",&a);
    printf("Indirizzo di b: %p \n",&b);
    printf("Indirizzo del main: %p \n",&main);
    printf("Indirizzo di sum: %p \n",&sum);
    printf("Indirizzo di printf: %p \n",&printf);

    b = sum(a,d);
    printf("risultato: %d \n",b);
    exit(0);
}
```

variabile globale
(data seg)

funzione user
(code seg)

funzioni user
(code seg)

variabile allocate
sullo stack

funzioni libreria

RISULTATO DELL'ESECUZIONE

- Mandando in esecuzione il programma è possibile verificare la diversa zona di memoria in cui sono allocate le parti (l'output varia in modo sostanziale da sistema a sistema)

```
Indirizzo di d: 0x8049778          --> data segment
Indirizzo di a: 0xbf9520b0        --> stack
Indirizzo di b: 0xbf9520ac        --> stack
Indirizzo del main: 0x80483d8     --> text (code seg)
Indirizzo di sum: 0x80483a4       --> text (code seg)
Indirizzo di printf: 0x8048300    --> text (code seg)
[sum] Indirizzo del parametro a: 0xbf952090 --> stack
[sum] Indirizzo della variabile locale s: 0xbf952084 --> stack
risultato: 12
```

- Gli indirizzi in realtà - come verrà spiegato nel modulo relativo alla memoria - sono indirizzi virtuali, ovvero non fisici: sono trasformati in indirizzi fisici in modo diverso a seconda delle tecniche di gestione della memoria adottate
- Per vedere il layout e disassemblato del programma, con indirizzi da rilocare, usare il tool **gdb**

PROCESSI E FILE ESEGUIBILI

- Un file binario *eseguibile* contiene un programma che può essere caricato e mandato in esecuzione come processo
- Esistono **formati** diversi, proposti dalle diverse famiglie dei sistemi operativi
 - su sistemi UNIX/Linux esempi sono **ELF** (recente), **a.out** (storico)
 - ELF è il default su sistemi Linux
 - su sistemi Windows **PE** (Portable Executable)
- In generale un formato definisce un layout con cui sono organizzate le informazioni all'interno del file eseguibile
 - dati, codice e meta-dati

FORMATO a.out

- a.out is a file format used in older versions of Unix-like computer operating systems for executables, object code, and, in later systems, shared libraries.
 - the name stands for *assembler output*.
 - a.out remains the default output file name for executables created by certain compilers/linkers when no output name is specified
 - even though these executables are no longer in the a.out format
- Several variants
 - OMAGIC
 - had contiguous segments after the header, with no separation of text and data. This format was also used as object file format.
 - NMAGIC
 - similar to OMAGIC, however the data segment is loaded on the immediate next page after the end of the text segment, and the text segment was marked read-only.
 - ZMAGIC
 - adds support for demand paging. The length of the code and data segments in the file had to be multiples of the page size.
 - QMAGIC
 - binaries loaded one page above the bottom of the virtual address space, in order to permit trapping of null pointer dereferences via a segmentation fault

a.out LAYOUT

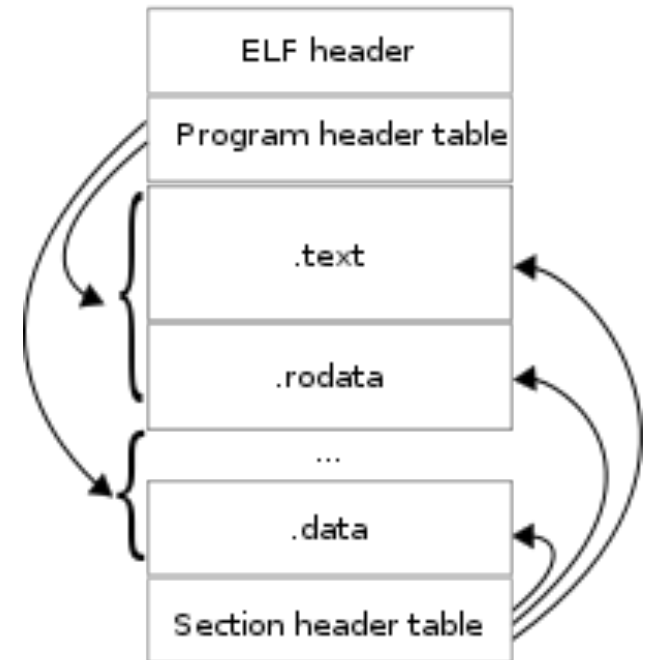
- An a.out file consists of up to seven sections, in the following order:
 - exec header
 - contains parameters used by the kernel to load a binary file into memory and execute it, and by the link editor ld to combine a binary file with other binary files. This section is the only mandatory one.
 - text segment
 - contains machine code and related data that are loaded into memory when a program executes. May be loaded read-only.
 - data segment
 - contains initialized data; always loaded into writable memory.
 - text relocations
 - contains records used by the link editor to update pointers in the text segment when combining binary files.
 - data relocations
 - like the text relocation section, but for data segment pointers.
 - symbol table
 - contains records used by the link editor to cross-reference the addresses of named variables and functions (symbols) between binary files.
 - string table
 - contains the character strings corresponding to the symbol names.

EXECUTABLE AND LINKING FORMAT (ELF)

- Common standard file format for executables, object code, shared libraries, and core dumps.
 - not bound to any particular processor or architecture.
 - adopted by many different operating systems on many different platforms
 - has replaced older executable formats such as a.out and COFF in many Unix-like operating systems such as Linux, Solaris, FreeBSD
 - some adoption in non-Unix operating systems
 - Itanium version of OpenVMS, BeOS, PlayStation
 - extensible
- Utility to analyse an executable file under UNIX env
 - readelf, objdump, file

ELF LAYOUT

- ELF header
 - describes where the various parts are located in the file
- File data
 - **Program** header table, describing zero or more *segments*
 - contain information that is necessary for runtime execution of the file
 - **Section** header table, describing zero or more *sections*
 - contain important data for linking and relocation
 - Data referred to by entries in the program header table, or the section header table



PORTABLE EXECUTABLE (PE)

- File format for executables, object code, and DLLs, used in 32-bit and 64-bit versions of Windows operating systems
 - the term "portable" refers to the format's versatility in numerous environments of operating system software architecture.
- Encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code.
 - including dynamic library references for linking, API export and import tables, resource management data and thread-local storage (TLS) data.
 - on NT operating systems, the PE format is used for EXE, DLL, OBJ, SYS (device driver), and other file types.
- PE is a modified version of the Unix COFF file format.
 - PE/COFF is an alternative term in Windows development.
- Layout
 - consists of a number of headers and sections that tell the dynamic linker how to map the file into memory
- Recently extended by Microsoft's .NET Framework with features to support the Common Language Runtime.

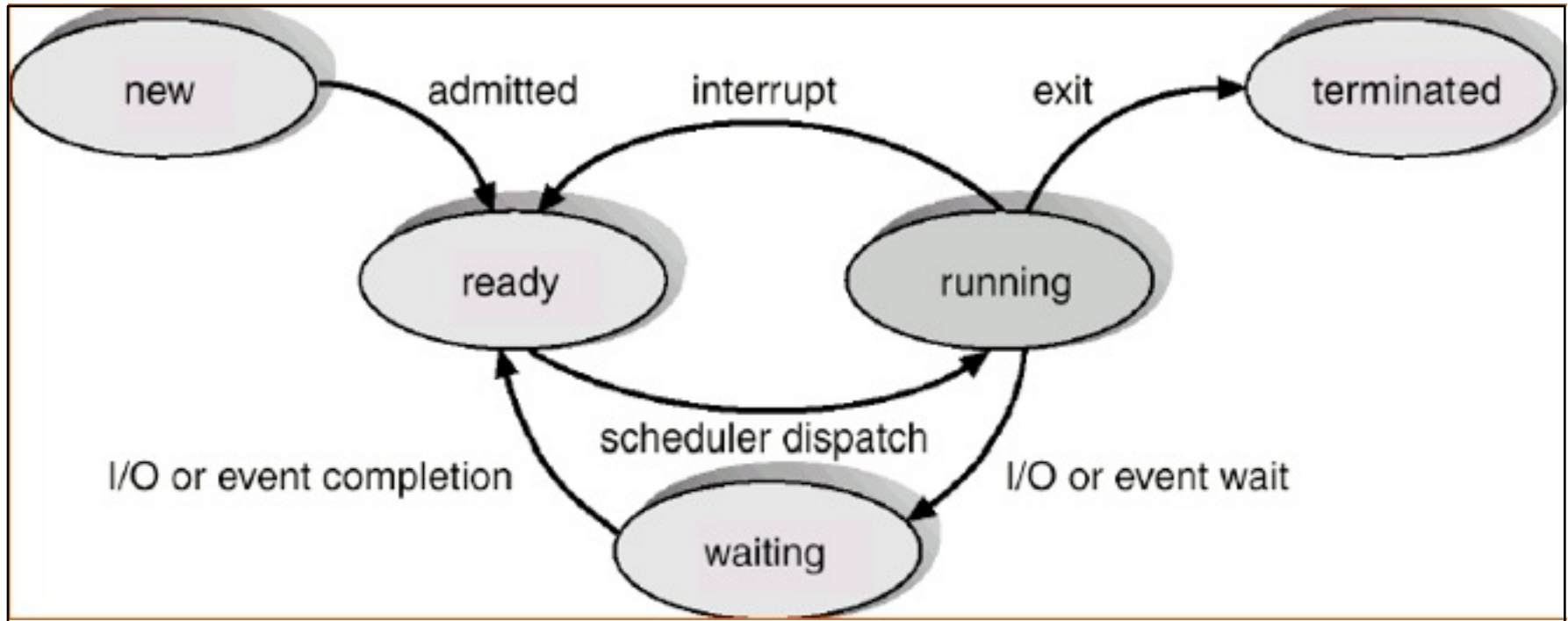
COMPONENTI DI UN PROCESSO

- Le componenti principali di un processo in un S.O. moderno sono:
 - **codice** del programma
 - **contesto di esecuzione**
 - immagine dei registri del processore
 - program counter (PC)
 - **stack**
 - area dati globali
 - heap
 - parametri e variabili d'ambiente

STATO DI UN PROCESSO

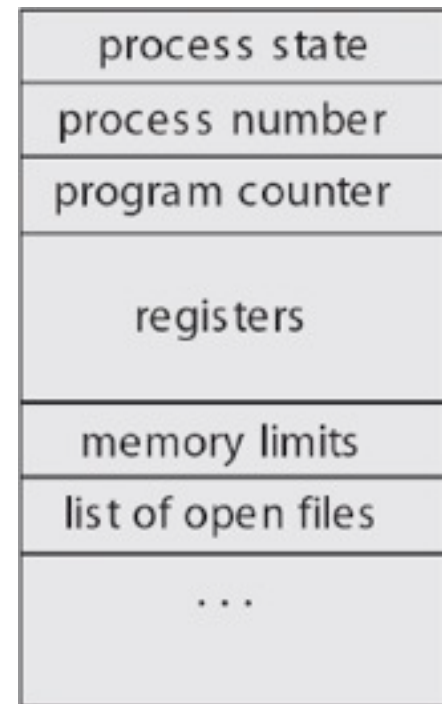
- Dal momento che entra in esecuzione, nel proprio ciclo di vita un processo può cambiare più volte stato
 - lo stato è determinato dal *tipo di attività* che il processo sta svolgendo (es. eseguendo istruzioni CPU, in attesa di operazioni I/O, etc)
- In generale i possibili stati di un processo sono:
 - **new**
 - il processo è appena stato creato
 - **running**
 - il processo sta eseguendo istruzioni
 - **waiting**
 - il processo è in attesa di un evento (es. completamento operazione di I/O o ricezione di un segnale da un altro processo)
 - **ready**
 - pronto ad eseguire istruzioni, in attesa di ricevere un processore
 - **terminated**
 - il processo ha completato l'esecuzione

DIAGRAMMA DEGLI STATI DI UN PROCESSO



PROCESS CONTROL BLOCK (PCB)

- Un sistema operativo ha strutture dati apposite con cui tiene traccia di tutti i processi in esecuzione e del loro stato.
- In particolare per ogni processo viene creato un **Process Control Block** (*PCB*) o *task control block* ove si mantengono informazioni specifiche sul processo e il suo stato.
- In particolare un PCB contiene:
 - id del processo, del processo genitore
 - stato del processo (attivo, sospeso, ..)
 - contesto
 - program counter
 - registri del processore
 - informazioni sullo scheduling del processore
 - informazioni sulla gestione della memoria
 - informazioni di accounting
 - informazioni sulle operazioni di I/O



PCB IN SISTEMI UNIX

- Informazioni contenute nel PCB in sistemi UNIX
 - PID del processo, PID del processo padre (PPID), ID utente (user ID)
 - stato
 - descrittore di evento
 - per un processo in stato di sleeping, indica l'evento per cui il processo è in attesa
 - puntatori alle tabelle delle pagine
 - dimensione delle tabelle delle pagine
 - posizione della user area
 - priorità
 - segnali pendenti

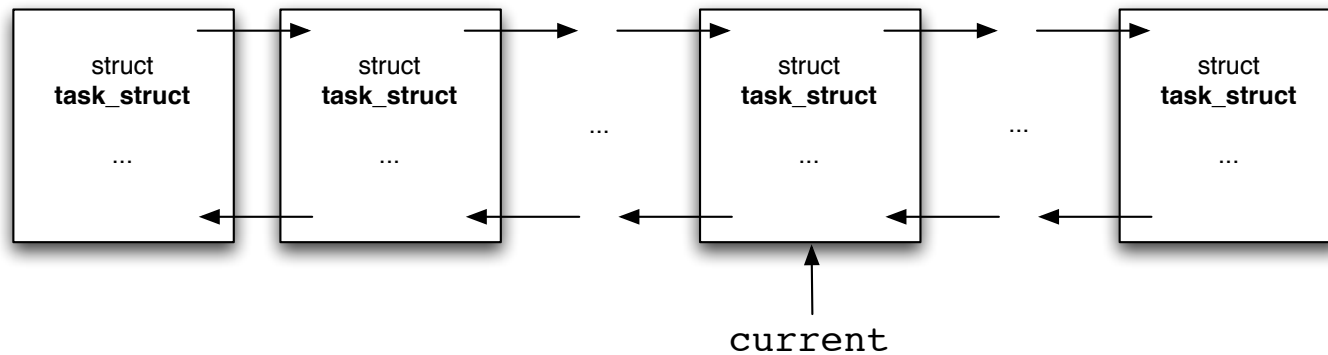
IMPLEMENTAZIONE DI UN PCB: LINUX

- E' rappresentato da una struttura **task_struct** molto corposa
 - dichiarata in `./include/linux/sched.h`
- Alcuni campi:

```
struct task_struct {
    volatile long state;      /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    ...
    pid_t pid;      /* process identifier */
    ...
    struct task_struct *parent; /* processo genitore del processo */
    struct list_head children; /* lista dei figli del processo */
    struct files_struct *files; /* lista dei file aperti */
    struct mm_struct *mm;      /* memoria del processo */
    ...
}
```

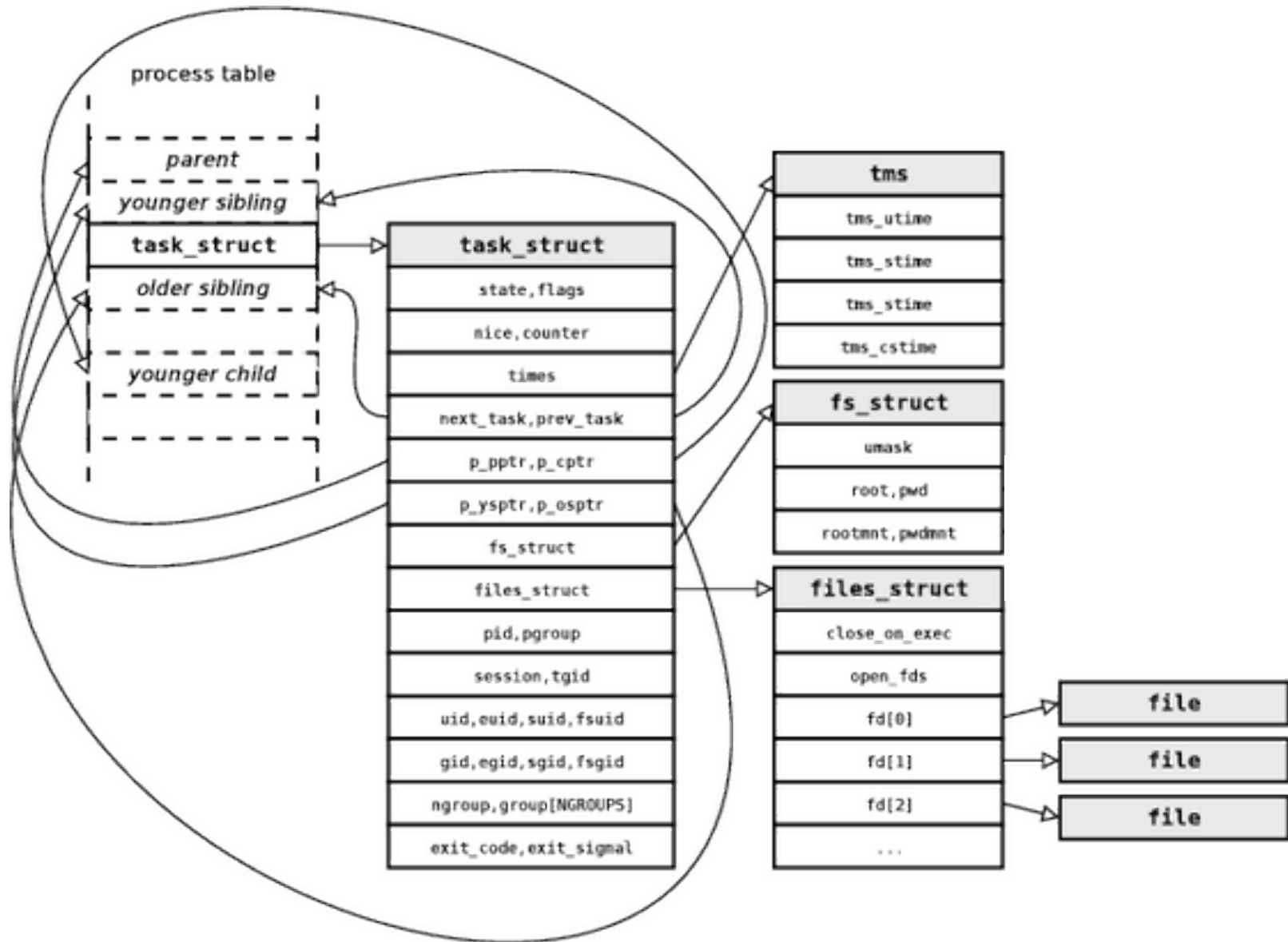
TABELLA DEI PROCESSI

- Struttura dati con cui il kernel tiene traccia di tutti i processi attivi nel sistema
 - ogni entry della tabella è il PCB di un processo attivo
 - tipicamente ha una dimensione massima prefissata
 - limite massimo al numero di processi che possono essere creati
- Esempio Linux
 - la tabella dei processi è implementata come lista doppiamente concatenata



- Il kernel mantiene aggiornato un puntatore **current**, che punta al descrittore del processo correntemente in esecuzione

TABELLA DEI PROCESSI IN LINUX

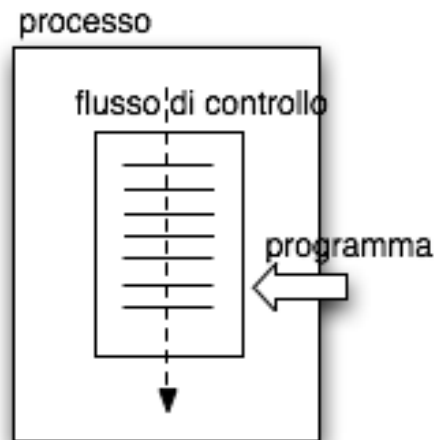


USER AREA IN PROCESSI UNIX/LINUX

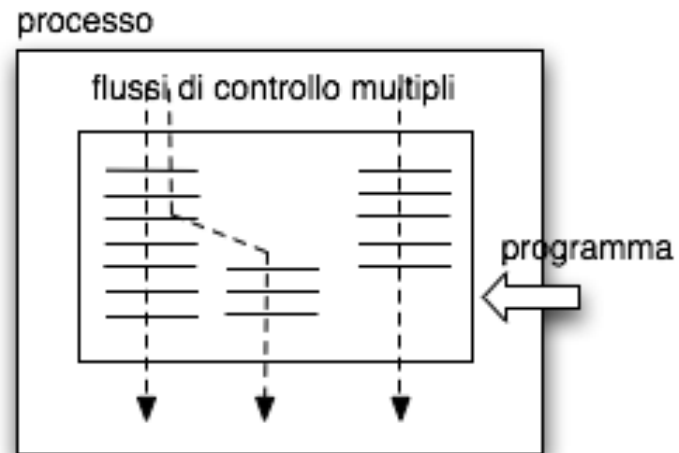
- Nei sistemi Unix, le informazioni di sistema relative ad un processo sono mantenute in parte nella tabella dei processi, in parte in un'area di memoria chiamata **user area**
 - è localizzata al termine della parte superiore dello spazio di indirizzamento del processo
 - è accessibile solo quando il sistema esegue in kernel mode
 - contiene
 - puntatore all'ingresso relativo al processo nella process table
 - setting dei segnali
 - tabella dei file
 - directory di lavoro corrente
 - umask settings
- La user area può essere sottoposta a swap su disco.
 - al contrario la tabella dei processi, che non può essere swappata

PROCESSI E THREADS

- Ad un processo sono associati uno o più flussi di controllo definiti **threads**
 - per flusso di controllo si intende l'esecuzione sequenziale di istruzioni da parte dell'esecutore del processo
- Nei moderni sistemi operativi, un processo può avere più flussi di controllo mediante *multi-threading*
 - argomento molto importante, affrontato nei prossimi moduli



processo con
singolo flusso di controllo



processo con
più flussi di controllo (threads)

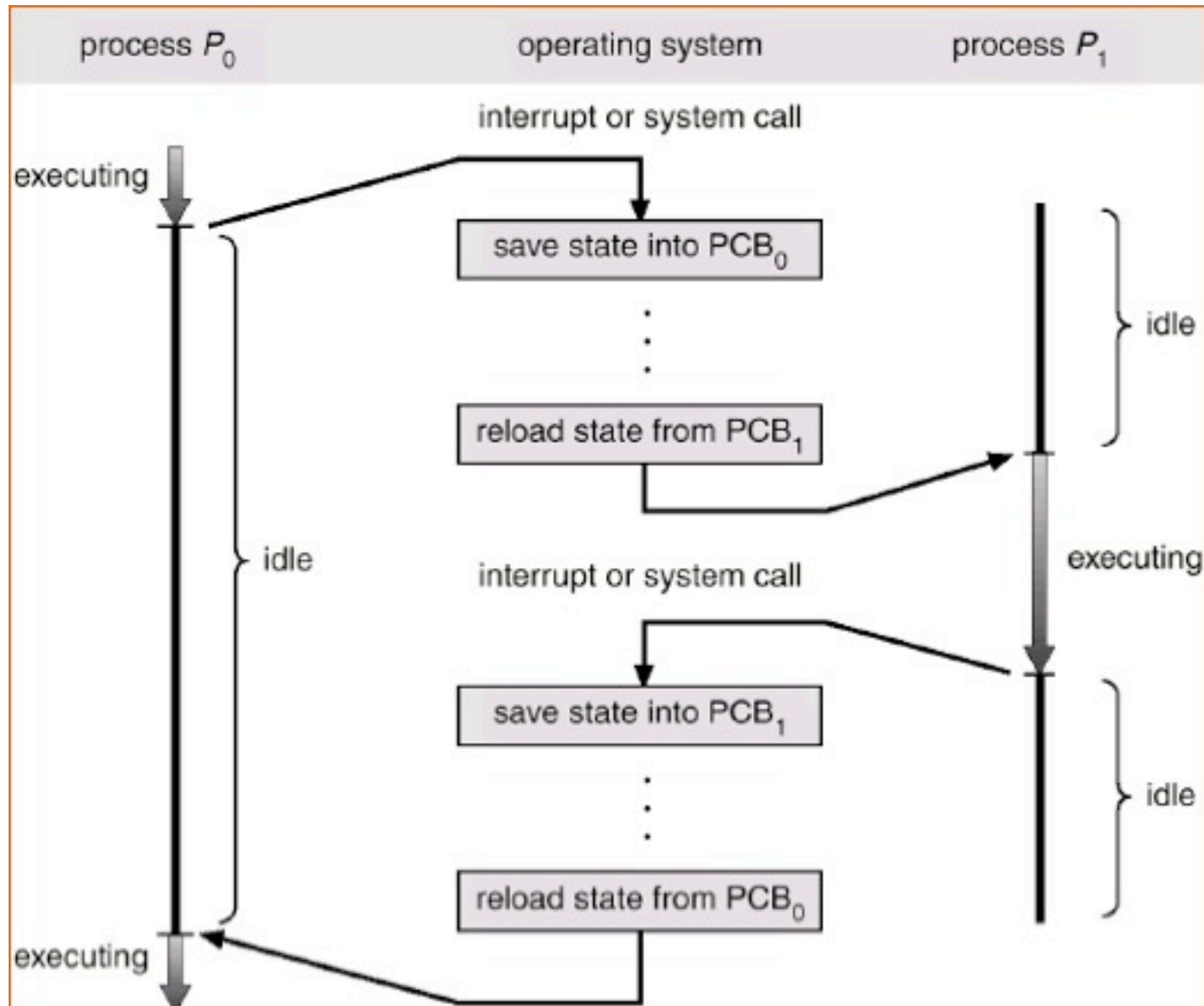
SCHEDULING DEI PROCESSI

- La presenza di più processi in esecuzione concorrentemente richiede strategie adeguate di allocazione della CPU per la loro esecuzione (**process scheduling**)
 - obiettivo del S.O. è massimizzare lo sfruttamento dei processori per l'esecuzione dei processi.
 - Il **process scheduler** è quel componente del sistema operativo che si occupa dello scheduling dei processi
 - ovvero di selezionare man mano a quale processo attivo allocare il processore, secondo certe politiche.
- Su un sistema mono-processore ci sarà sempre solo un processo effettivamente in esecuzione sul processore
 - i multi-core possono essere considerati in questo caso alla stregua dei sistemi multi-processore
- Gli altri eventuali processi attivi attendono che il processore sia disponibile e riallocato
 - l'effetto di simultaneità tipico dei sistemi time-sharing si ottiene allocando a rotazione il processore ai vari processi, per quanti di tempo prefissati

CAMBIO DI CONTESTO

- L'allocazione di un processore da un processo ad un altro da parte del kernel richiede un *cambio di contesto* (**context switch**) in cui
 - viene salvato il contesto corrente di esecuzione nel PCB del processo a cui è tolto il processore
 - viene ripristinato il contesto corrente di esecuzione con il contesto descritto nel PCB a cui è stato allocato il processore
- Il tempo di context switch è di puro overhead, e quindi da minimizzare il più possibile
 - in genere sull'ordine dei micro-secondi

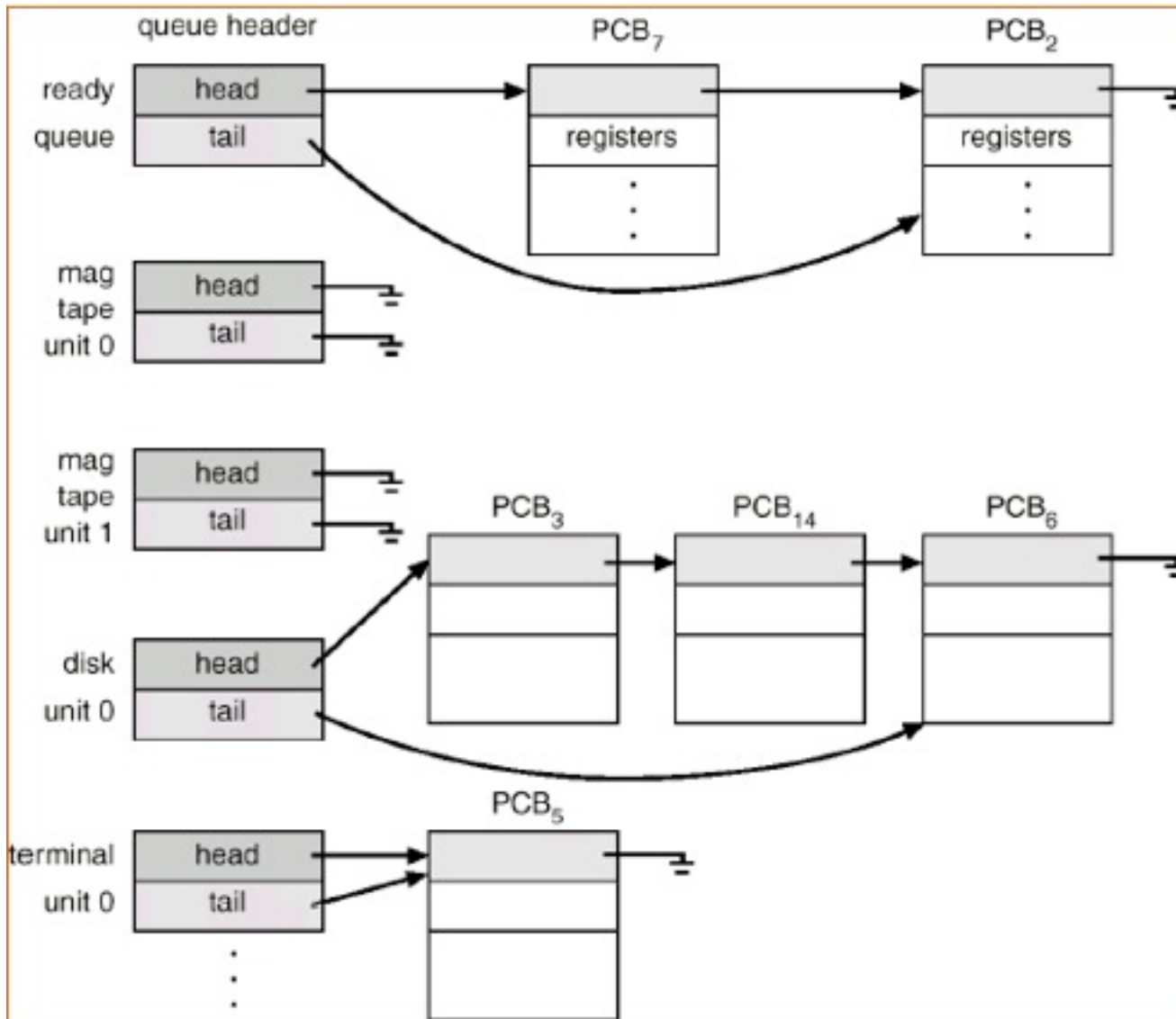
CPU SWITCH DA PROCESSO A PROCESSO



CODE DI SCHEDULING

- Il sistema operativo gestisce la dinamica dei processi mediante delle code, ove vengono inseriti a seconda dello stato in cui sono
 - **job queue**
 - coda che contiene i processi lanciati nel sistema, residenti tuttavia ancora in memoria di massa
 - **ready queue**
 - coda che contiene l'insieme dei processi residenti in memoria centrale, pronti e in attesa di essere eseguiti.
 - **device / wait queue**
 - coda (relativa ad uno specifico device di I/O) che contiene l'elenco di processi in attesa del completamento di qualche operazione di I/O fatta sul device.
- Le code sono tipicamente realizzate come liste semplicemente concatenate, che contengono puntatori ai PCB
- Il sistema operativo sposta processi da coda a coda a seconda delle dinamiche in atto

CODE DI SCHEDULING

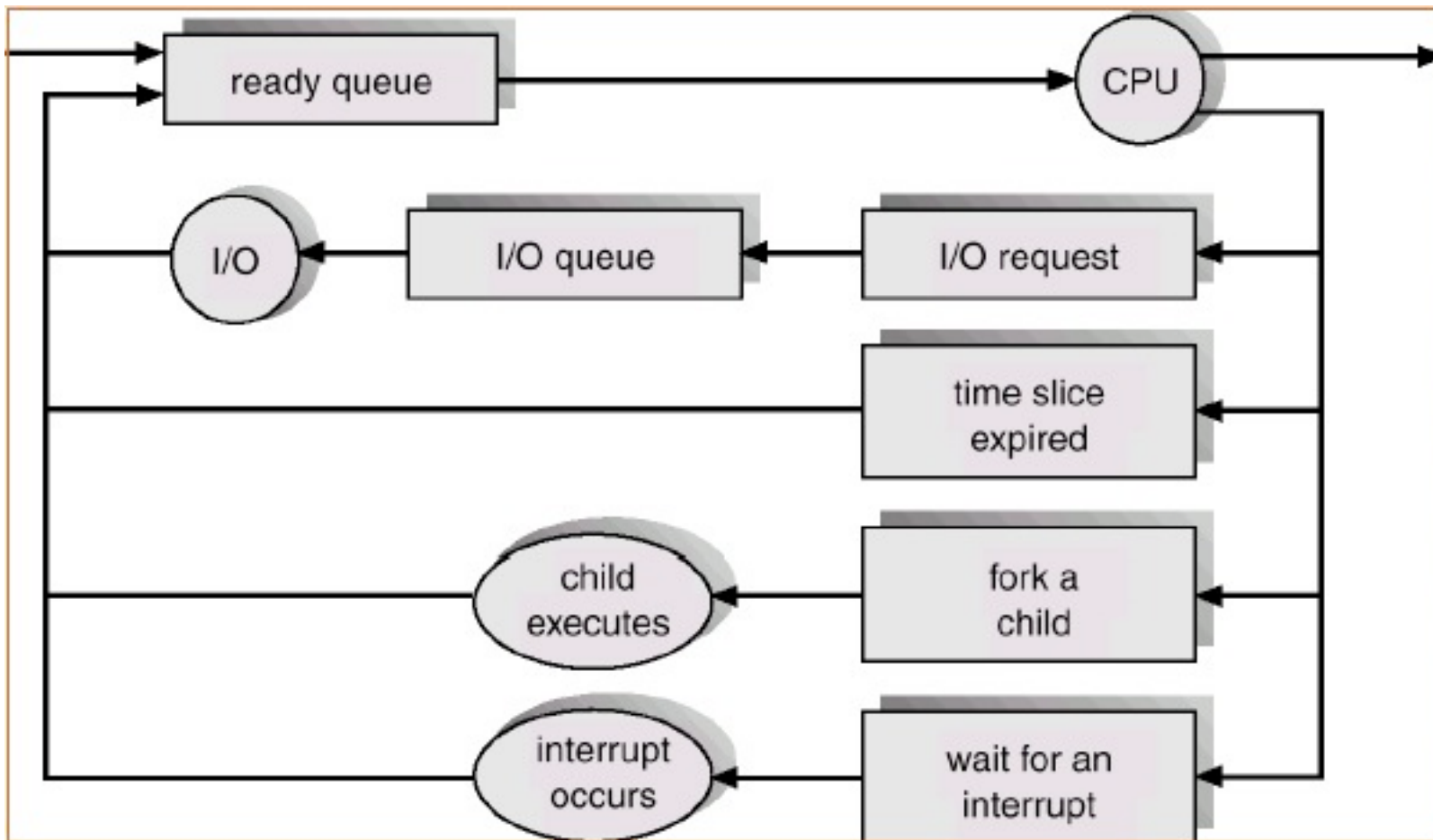


CICLO DI VITA DI UN PROCESSO

- Un processo appena creato è inserito nella ready queue.
- Un processo è **dispatched** quando viene scelto dallo scheduler per essere eseguito (o continuare l'esecuzione), rimuovendolo dalla coda e assegnandolo alla CPU
- In seguito a operazioni di I/O, oppure all'occorrenza di interruzioni, il processo può essere reinserito nella *ready* queue, allocando il processore ad un altro processo
 - in realtà in sistemi operativi di una certa complessità i processi possono essere anche temporaneamente 'congelati' e trasferiti in memoria secondaria, per essere ripristinati successivamente.
 - questo accade tipicamente nei sistemi batch o in sistemi che ottimizzano l'uso della memoria primaria.

DINAMICA: QUADRO RIASSUNTIVO

- Un diagramma riassuntivo della dinamica dei processi relativamente all'inserimento e rimozione dalle varie code nello scheduling è fornito in figura:



SCHEDULER

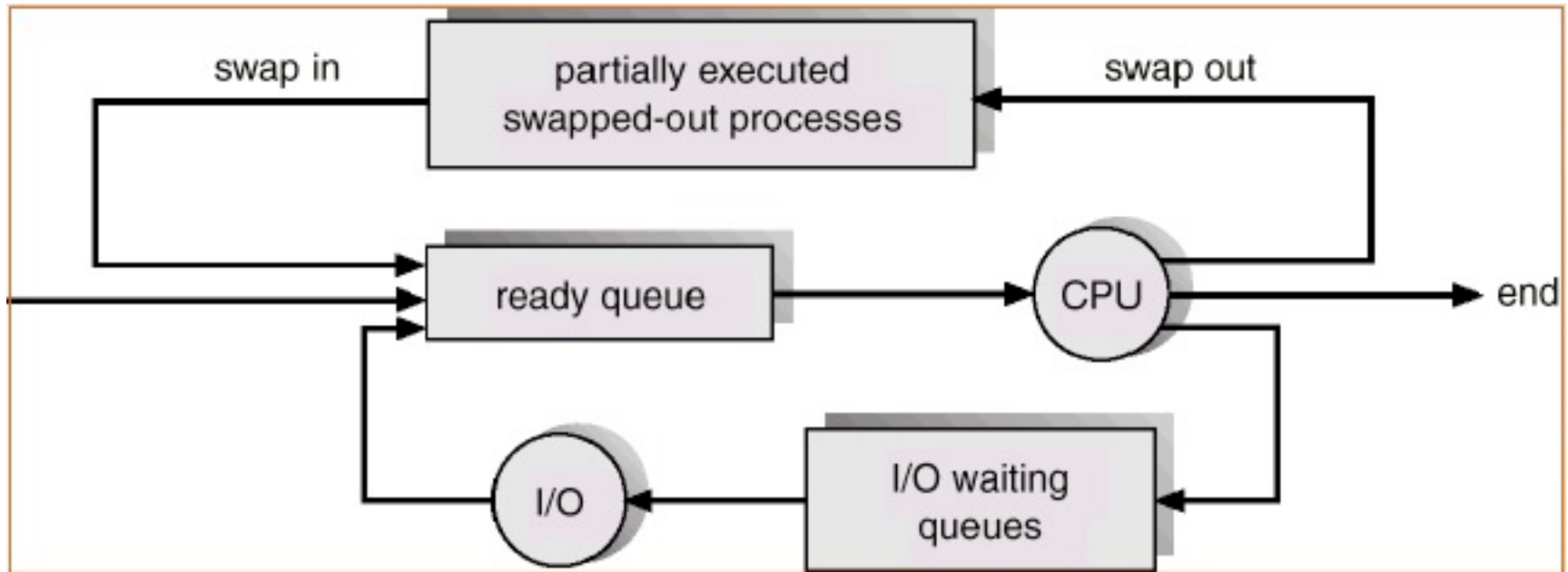
- Ci sono due tipi di scheduler:
 - **short-term scheduler** (o CPU scheduler)
 - seleziona il processo a cui allocare la CPU fra quelli presenti nella ready queue
 - **long-term scheduler** (o job scheduler)
 - seleziona processi residenti in memoria secondaria (dalla job queue) per essere trasferito in memoria centrale
- Lo short-term scheduler entra in azione con una frequenza molto più alta rispetto al long-term scheduler
- Il long-term scheduler controlla e determina il *livello di multiprogrammazione* del sistema, ovvero il numero di processi presenti simultaneamente in memoria centrale.
 - entra in azione tipicamente quando un processo in memoria centrale termina

PROCESSI CPU-BOUND E I/O BOUND

- In proposito i processi sono classificati in
 - **CPU-bound**
 - genera poche richieste di I/O e per lo più sfrutta intensamente la CPU
 - **I/O-bound**
 - genera molte richieste di I/O
- Quindi è fondamentale scegliere un buon mix fra processi CPU-bound e I/O-bound
 - in generale compito del long-term scheduler

MEDIUM TERM SCHEDULER

- In sistemi operativi time-sharing è spesso frequente un altro scheduler, il **medium-term scheduler**:



- Tale scheduler si occupa di fare lo **swapping** di processi in esecuzione - tipicamente inattivi - da memoria centrale a memoria secondaria e viceversa, a seconda delle esigenze.

CHIAMATE DI SISTEMA RELATIVE AI PROCESSI

- I sistemi operativi mettono a disposizione servizi - sempre forniti in forma di system call - per varie operazioni che concernono i processi, in particolare:
 - recupero informazioni di stato
 - creazione
 - terminazione
 - comunicazione
 - sincronizzazione / coordinazione
 - (mobilità)

RECUPERO INFORMAZIONI

- Esistono varie funzioni per recuperare informazioni sui processi
 - ad esempio l'ID dei processi
- Nello standard POSIX ad esempio esiste la chiamata **getpid** con cui è possibile ottenere l'ID di un processo:

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

int main (void){
    printf("hello world from process ID %d \n", getpid());
    exit(0);
}
```

- lanciando più volte il programma (dopo averlo compilato) si ottiene un output del tipo:

hello world from process ID 10590

hello world from process ID 10591

CREAZIONE DI PROCESSI

- E' tipicamente definita in modo gerarchico
 - la creazione di un processo avviene ad opera del processo definito genitore (*parent*) e i processi generati vengono definiti figli (*children*)
 - i processi figli possono a loro volta creare processi, divenendo loro stessi processi parent
 - si creano quindi degli alberi di processi (*process tree*)
- La creazione di un processo implica l'allocazione di un certo insieme di risorse (memoria in primis) allo scopo
- Processi genitori e figli possono condividere o meno - a seconda dei vari S.O. - risorse
- In particolare nei sistemi operativi moderni ogni processo *ha il proprio spazio di indirizzamento di memoria*
 - **non c'è condivisione di memoria fra processi**
 - per condividere memoria è necessario sfruttare opportuni meccanismi forniti dal sistema operativo
 - shared memory, descritta in seguito

CREAZIONE DI UN PROCESSO: `fork`

- Nei sistemi UNIX (POSIX) **fork** è la chiamata di sistema per creare un nuovo processo

```
pid_t      fork(void);
```

- L'esecuzione della chiamata comporta la creazione un nuovo processo, con uno spazio di memoria (dati, stack) che è *copia esatta*, al momento della creazione, di quello del padre
 - quindi un ulteriore flusso di controllo (del processo figlio)
- A livello di programma, il processo figlio parte ad eseguire direttamente alla chiamata della `fork`
 - quindi sia il padre, sia il figlio continuano l'esecuzione dal punto in cui è stata chiamata la `fork`
- Per distinguere padre dal figlio:
 - nel caso del processo figlio, la `fork` restituisce come valore di ritorno 0
 - nel caso del processo padre, la `fork` restituisce un numero intero pari all'*identificatore di processo del figlio*

ESEMPIO

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

int main(int argc, char* argv[]){
    pid_t pid;

    /* crea un altro processo */
    pid = fork();

    if (pid<0){
        printf("Fork failed.\n");
        exit(1);
    } else if (pid==0){ /* processo figlio */
        printf("[NEWPROC] started \n");
        sleep(5);
        printf("[NEWPROC] completed \n");
    } else { /* processo genitore */
        printf("[PARENT] waiting for child proc %d\n",pid);
        wait(NULL);
        printf("[PARENT] child proc completed. \n");
        exit(0);
    }
}
```

-il processo padre crea un processo figlio con **fork**

-Il processo figlio aspetta 5 secondi, poi termina

-Il padre aspetta la terminazione del figlio (con **wait**) e poi esce

ATTESA TERMINAZIONE DEI FIGLI

– **wait**

- sospende l'esecuzione del processo padre fino a quando *uno qualsiasi dei processi figli* ha terminato la propria esecuzione

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

pid_t wait(int *stat_loc);
```

– **waitpid:**

- sospende l'esecuzione del processo padre fino alla terminazione di uno specifico processo figlio

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

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

SECONDO ESEMPIO (testfork2.c)

- Processo padre che crea N figli
 - ogni figlio stampa in standard output il proprio PID e attende un certo numero (casuale) di millisecondi prima di terminare
 - il padre aspetta la terminazione di tutti i figli e poi termina


```

#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<time.h>

int getrand(int base, int range);
void proc();

int main(int argc, char** argv){
    pid_t pid;
    int i;
    int nprocs;

    nprocs = atoi(argv[1]);
    printf("[PARENT] Spawning
           %d processes \n",nprocs);
    for (i = 0; i < nprocs; i++){
        pid = fork();
        if (pid < 0){
            printf("Error in process
                   creation (%d)\n",i);
        } else if (pid == 0){
            proc();
        }
    }

    printf("[PARENT] Waiting
           for child processed...\n");
    for (i = 0; i < nprocs; i++){
        wait(NULL);
    }
    printf("[PARENT] All processes
           terminated.\n");

    exit(0);
}
...

```

```

...

int getrand(int base, int range){
    int c = clock();
    srand(c);
    return base + rand()%range;
}

void proc(){
    pid_t mypid;
    int dt;

    mypid = getpid();
    dt = getrand(500,1500);

    printf("Process %d started
           - waiting for %d ms.\n",mypid,dt);
    usleep(dt*1000);
    printf("Process %d completed.\n",mypid);
    exit(0);
}

```

ISOLAMENTO

- I processi *non* condividono memoria
 - nei sistemi UNIX alla creazione con `fork` il processo figlio ha una *copia* dell'immagine completa di memoria del processo padre

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

int value = 0;

int main(int argc, char** argv){
    pid_t pid;

    pid = fork();
    if (pid == 0){
        /* processo figlio */
        value++;
        exit(0);
    } else if (pid > 0){
        /* padre */
        /* aspetta terminazione del figlio */
        wait(NULL);
        printf("%d\n",value);
        exit(0);
    }
}
```

In output viene stampato (dal padre) il valore 0

- se ci fosse stata condivisione di memoria, padre e figlio avrebbero condiviso la variabile `value` e il valore in output sarebbe stato 1, dopo l'incremento del figlio

ESECUZIONE DI UN PROGRAMMA ESTERNO: `exec`

- Nei sistemi UNIX (POSIX) la famiglia di chiamate di sistema **exec** permettono di caricare e mandare in esecuzione un programma eseguibile

```
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl_e(const char *path, const char *arg , ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

– **execl**

- carica in memoria ed esegue un programma, nel contesto del processo corrente
- varianti: `execv`, `execl_e`, `execlp`, `execvp`
 - a seconda che siano specificati anche parametri al programma da eseguire e le variabili d'ambiente

CREAZIONE PROCESSO ED ESECUZIONE COMANDO SHELL

```
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>

int main(int argc, char* argv[]){
    pid_t pid;

    /* crea un altro processo */
    pid = fork();

    if (pid<0){
        fprintf(stderr,"Fork fallita.\n");
        exit(1);
    } else if (pid==0){ /* processo figlio */
        execl("/bin/ls","");
    } else { /* processo genitore */
        wait(NULL);
        printf("Processo figlio terminato.\n");
        exit(0);
    }
}
```

-il processo padre crea un processo figlio con **fork**

-Il processo figlio carica ed esegue con **execlp** il programma di sistema 'ls' (che ha come effetto il listing della directory corrente)

-Il padre aspetta la terminazione del figlio (con **wait**) e poi esce

DEMONI

- Con il termine **demone** (daemon) si indica - in particolare nel mondo UNIX - un processo il cui tempo di vita è molto lungo
 - tipicamente un demone viene eseguito al bootstrap del sistema e termina al shutdown del sistema stesso
 - i demoni sono eseguiti *in background* (i processi normali invece in *foreground*), senza la possibilità di interagire mediante terminale con gli utenti
 - i demoni sono utilizzati frequentemente 'come server' di servizi (es: Web Server) e svolgere attività giornaliere di supporto al sistema
- Per vedere i demoni attivi nel sistema: **ps -ajx**
- Tra i demoni più noti:
 - `/sbin/init` (servizi fondamentali del OS)
 - `inetd` (in ascolto di richieste di connessione di rete - servizi di rete)
 - `lpd` (servizi di stampa)
 - `sendmail` (servizi di email)
 - `pagedaemon` (servizio di gestione della memoria paginata)
 - ...

ESEMPIO: DEMONE UNIX IN C (daemon.c)

```
...
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <fcntl.h>

void daemon_body();
int init_daemon();

int main(){
    int status;
    status = init_daemon();
    if (status == 0){
        exit(0);
    } else {
        exit(1);
    }
}

...
...
int init_daemon(){
    pid_t pid;
    if ((pid = fork()) < 0){ /* errore */
        printf("Installing daemon... FAILED\n.");
        return -1;
    } else if (pid!=0) { /* parent */
        printf("Installing daemon... OK.\n");
        return 0;
    }
    setsid(); /* diventa owner della sessione*/
    daemon_body();
    return 1;
}

void daemon_body(){
    time_t t;
    char *st;
    while (1){
        sleep(10); /* 10 secondi */
        time(&t);
        st = ctime(&t);
        printf("\n Current Time: %s ",st);
    }
}
}
```

CREAZIONE PROCESSI IN WIN32

- La creazione di processi nei sistemi Windows è possibile mediante **CreateProcess**
 - a differenza della **fork** il nuovo processo non eredita codice/dati dal padre, ma deve essere specificato il programma da eseguire
 - funzione con molti parametri
- Dichiarazione:

```
BOOL WINAPI CreateProcess(  
    __in_opt LPCTSTR lpApplicationName,  
    __inout_opt LPTSTR lpCommandLine,  
    __in_opt LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    __in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    __in BOOL bInheritHandles,  
    __in DWORD dwCreationFlags,  
    __in_opt LPVOID lpEnvironment,  
    __in_opt LPCTSTR lpCurrentDirectory,  
    __in LPSTARTUPINFO lpStartupInfo,  
    __out LPPROCESS_INFORMATION lpProcessInformation  
);
```

ESEMPIO

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

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    // init strutture
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // usa linea di comando
        "C:\\WINDOWS\\system32\\mspaint.exe", // cmd line
        NULL, // non ereditare il process handle
        NULL, // non ereditare il thread handle
        FALSE, // disabilita l'ereditarietà per gli handle
        0, // nessun flag di creazione specifico
        NULL, // utilizza l'ambiente del processo padre
        NULL, // utilizza il path del processo padre,
        &si,
        &pi))
    {
        fprintf(stderr, "Creazione del processo fallita.");
        return -1;
    }

    // il padre aspetta la terminazione del figlio
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Processo figlio terminato.\n");

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```


PROCESS_INFORMATION

```
typedef struct _PROCESS_INFORMATION {  
    HANDLE hProcess;  
    HANDLE hThread;  
    DWORD  dwProcessId;  
    DWORD  dwThreadId;  
}PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

hProcess

A handle to the newly created process. The handle is used to specify the process in all functions that perform operations on the process object.

hThread

A handle to the primary thread of the newly created process. The handle is used to specify the thread in all functions that perform operations on the thread object.

dwProcessId

A value that can be used to identify a process. The value is valid from the time the process is created until all handles to the process are closed and the process object is freed; at this point, the identifier may be reused.

dwThreadId

A value that can be used to identify a thread. The value is valid from the time the thread is created until all handles to the thread are closed and the thread object is freed; at this point, the identifier may be reused.

OGGETTI E HANDLE IN WINDOWS

- Windows utilizza un approccio OO per rappresentare ed interagire con le risorse del sistema
 - pur essendo Win32 API non ad oggetti...
- Qualsiasi risorsa creata è rappresentata un opportuno oggetto
 - processi, file, thread, etc.
- Un'applicazione può accedere alle informazioni di un oggetto ed interagirvi richiedendo/ottenendo un object **handle**
 - tipo HANDLE
 - da utilizzare come parametro nelle funzioni parte delle API per interagire con l'oggetto

TERMINAZIONE DI UN PROCESSO

- La terminazione di un processo può avvenire in varie circostanze:
 - quando viene eseguito l'ultima istruzione del suo programma e/o viene effettuata una chiamata di sistema apposita, chiamata **exit**
 - specificando un **exit status**, che può essere letta dal processo genitore (mediante `wait`) e che contiene informazioni circa la corretta terminazione o meno del processo
 - quando un processo esterno (tipicamente il genitore) ne richiede la terminazione, mediante system call **abort**
- In alcuni sistemi la terminazione di un processo implica automaticamente anche la terminazione di tutti i figli (**cascading termination**)
- Nei sistemi UNIX *non* è c'è cascading termination
 - può quindi capitare che un processo termini prima che i propri figli abbiano completato la propria esecuzione
 - in tal caso un processo figlio si dice **orfano** e viene automaticamente “adottato” dal processo **init** (che ha PID = 1)

PROCESSI ZOMBI

- Quando un processo termina, tutta la memoria e le risorse associate vengono liberate
- Tuttavia, nei sistemi UNIX l'entry nella tabella dei processi rimane in modo tale che il processo genitore del processo terminato possa leggere l'exit status mediante una **wait**
- Fintanto che il processo padre non esegue la wait, il processo figlio viene definito **zombie**
 - è terminato, ma la sua entry c'è ancora...
- Eseguendo la wait, l'entry viene correttamente rimossa
- Quindi è fondamentale che un processo esegua la wait per ogni processo figlio che ha creato
 - per evitare di riempire la tabella dei processi con entry di processi già terminati

COMUNICAZIONE FRA PROCESSI

COMUNICAZIONE FRA PROCESSI

- Abbiamo visto che I sistemi operativi moderni supportano l'esecuzione di più processi simultaneamente, sulla medesima CPU (processi concorrenti)
- Processi concorrenti in esecuzione simultaneamente possono essere:
 - **indipendenti**
 - se non hanno nessuna forma di dipendenza, ovvero la presenza / azioni degli uni non ha alcun effetto su quella degli altri
 - **cooperativi**
 - se le azioni di un processo hanno o possono avere un qualche effetto o sono essere affette dalle azioni di altri processi.
 - ad esempio la condivisione di risorse quali memoria, device, et.
- L'esecuzione concorrente di processi cooperativi richiede la presenza di servizi/meccanismi che supportino tale cooperazione, in particolare la **comunicazione** fra processi e la **sincronizzazione** delle loro azioni, come meccanismi primitivi di coordinazione.

COOPERAZIONE FRA PROCESSI: VANTAGGI

- A fronte di una maggiore complessità, sono tuttavia molteplici i benefici che si hanno progettando e realizzando ambienti che supportino la cooperazione fra processi. In particolare:
 - **information/resource sharing**
 - più processi possono essere interessati ad accedere e manipolare informazioni comuni
 - allo scopo è necessario fornire meccanismi e servizi che abilitino l'accesso e uso concorrente di risorse
 - **computation speedup**
 - Per i problemi scomponibili in attività distinte, indipendenti o da coordinare, l'utilizzo di processi cooperativi può portare a notevoli benefici in termini di performance, in particolare su sistemi multi-processore
 - **modularity**
 - L'utilizzo di processi (e thread) permette di aumentare notevolmente la modularità del sistema, definendo moduli che incapsulano non solo uno stato e operazioni (come oggetti), ma anche attività e flusso di controllo.

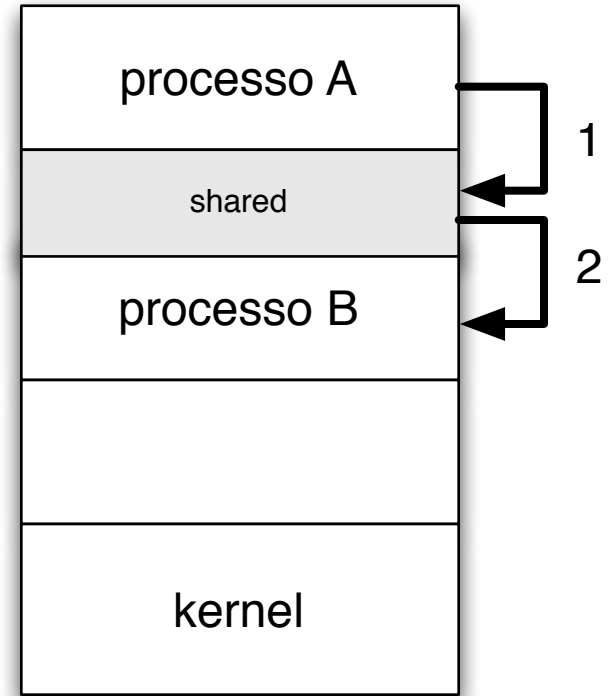
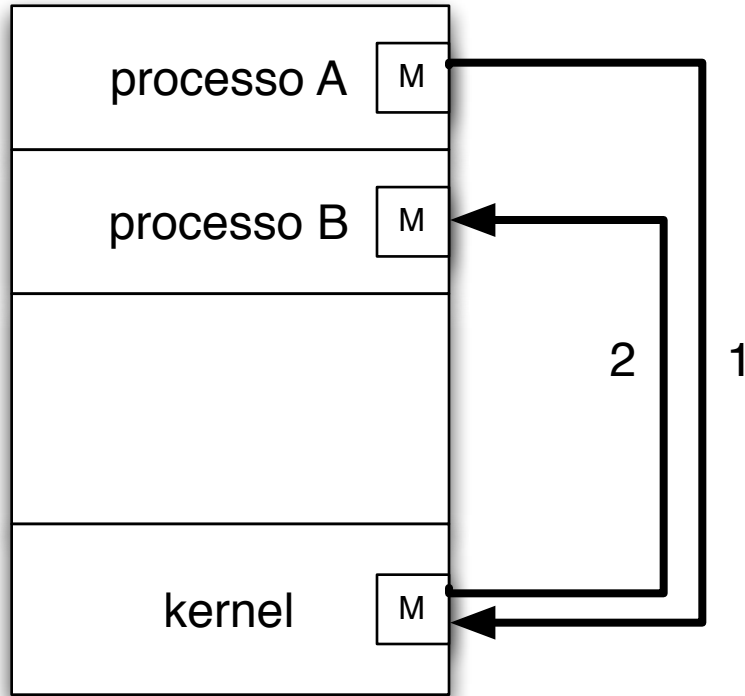
INTER-PROCESS COMMUNICATION (IPC)

- La **comunicazione** fra processi è un meccanismo fondamentale per realizzare processi cooperativi.
 - nel contesto di S.O. per comunicazione si intende *scambio di dati*.
 - processi distinti non condividono proprie aree di memoria, in virtù di principi di protezione (e incapsulamento) descritti in precedenza: servono dunque meccanismi di supporto per supportare la comunicazione
- I sistemi operativi forniscono un insieme base di servizi (primitive, chiamate di sistema,...) per **IPC (inter-process communication)**, che abilita la comunicazione fra processi
 - meccanismi IPC frequentemente supportano la comunicazione fra processi che possono essere anche distribuiti, ovvero risiedere su nodi distinti e connessi via rete.

MODELLI DI COMUNICAZIONE

- Esistono due principali modelli di comunicazione:
 - *comunicazione basata su scambio di messaggi e*
 - *comunicazione basata su memoria condivisa*
- **Scambio di messaggi** (*message passing*)
 - i processi comunicano mediante l'invio di messaggi, fra un mittente (sender) e un destinatario (receiver)
 - medium di comunicazione sono tipicamente canali stabiliti fra due o più processi, oppure mail box ove si depositano e recuperano messaggi
 - detti anche modelli *a memoria o ambiente locale*
- **Memoria condivisa** (*shared memory*)
 - i processi comunicano mediante l'accesso a memoria condivisa, che possono scrivere e leggere concorrentemente
 - la memoria condivisa è il medium di comunicazione
 - detti anche modelli *a memoria o ambiente comune*

MESSAGE PASSING VS SHARED MEMORY

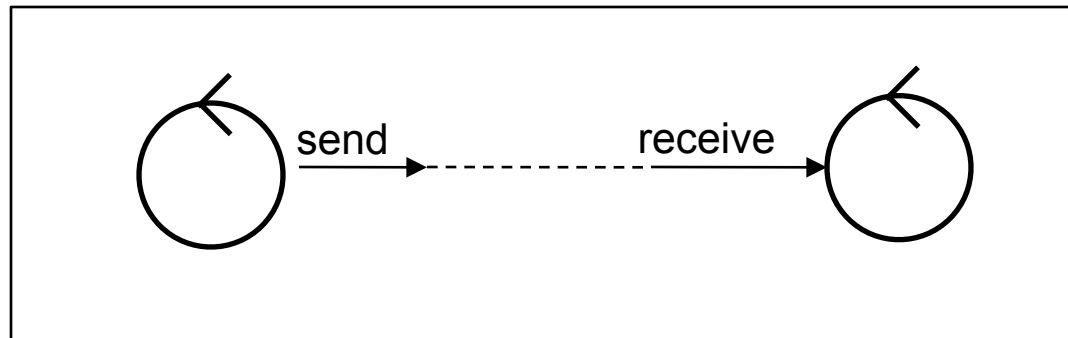


SCAMBIO DI MESSAGGI

- I processi comunicano inviando e ricevendo messaggi, specificando il destinatario/mittente del messaggio.
 - si parla di comunicazione *multicast* quando l'invio di un messaggio da un mittente ha come target un insieme di riceventi, tipicamente aggregati in gruppi
 - per *broadcast* si intende un multicast esteso a tutti i processi a prescindere dal gruppo
- Primitive classiche
 - **send** per invio di messaggi
 - **receive** per ricezione di messaggi
- In questo modello, se P e Q vogliono comunicare, devono:
 - 1) stabilire un *communication link*
 - 2) scambiare messaggi mediante send / receive
- La comunicazione può essere:
 - **diretta** o **indiretta**
 - **sincrona** o **asincrona**
 - con buffering automatico o esplicito

COMUNICAZIONE DIRETTA

- I processi comunicano specificando direttamente il destinatario/ sorgente dei messaggi, esplicitamente.
 - ai astraie da medium che realizza la comunicazione
- Le primitive sono:
 - **send**(P, message) –invio un messaggio al processo P
 - **receive**(Q, message) –ricevo un messaggio dal processo Q
- Questo schema realizza una comunicazione *simmetrica*.
- Esiste anche la variante *asimmetrica*, per molti aspetti più flessibile:
 - send(P, message) –invio un messaggio al processo P
 - receive(?id, message) –ricevo un messaggio da una qualsiasi processo. id verrà istanziato dinamicamente all'identificatore del processo mittente

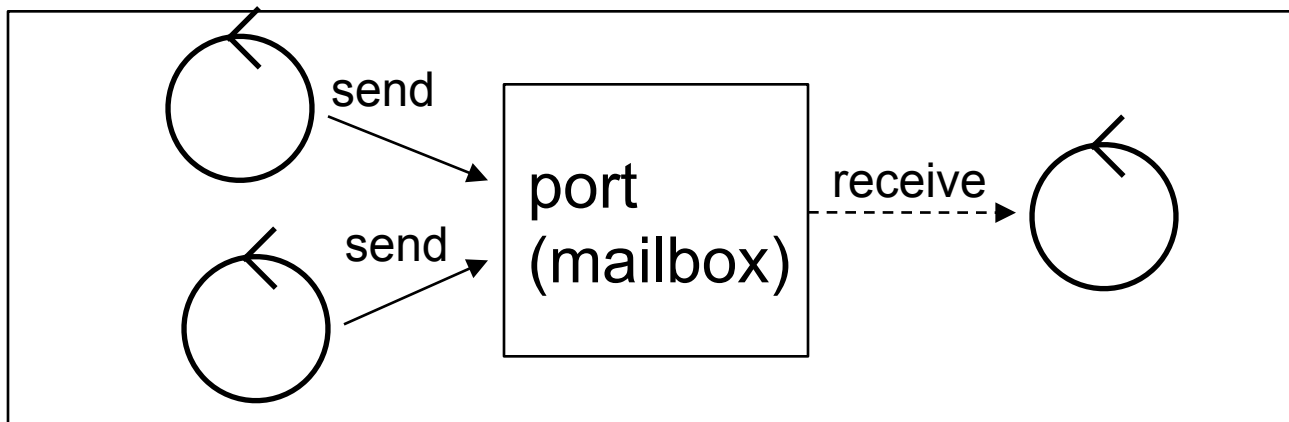


PROPRIETA' E LIMITI DELLA COMUNICAZIONE DIRETTA

- Proprietà tipiche dei link di comunicazione in caso di comunicazione diretta:
 - link stabiliti automaticamente
 - ogni link è dedicato ad una e sola coppia di processi comunicanti
 - un link può essere unidirezionale o bidirezionale (caso tipico)
- La comunicazione diretta ha seri limiti in termini di modularità:
 - cambiando l'identificatore di un processo è necessario cambiare il codice in tutti i processi con cui il processo comunica.

COMUNICAZIONE INDIRETTA MEDIANTE PORTE

- In questo approccio (il più diffuso) la comunicazione avviene indirettamente, tramite **mailbox** o porte (**port**) come medium di comunicazione.
 - ogni mailbox ha un proprio identificatore.
 - i processi comunicano depositando/recuperando informazioni da porte condivise .
 - politiche di gestione di messaggi (es: FIFO)
- Le mailbox o porte sono tipicamente gestite a livello di sistema operativo, che fornisce servizi per
 - creazione di una nuova mailbox
 - eliminazione di una mailbox esistente
 - inserimento di un messaggio nella mailbox A: `send(A, message)`
 - recupero di un messaggio della mailbox A: `receive(A, message)`



COMUNICAZIONE INDIRETTA: PROPRIETA'

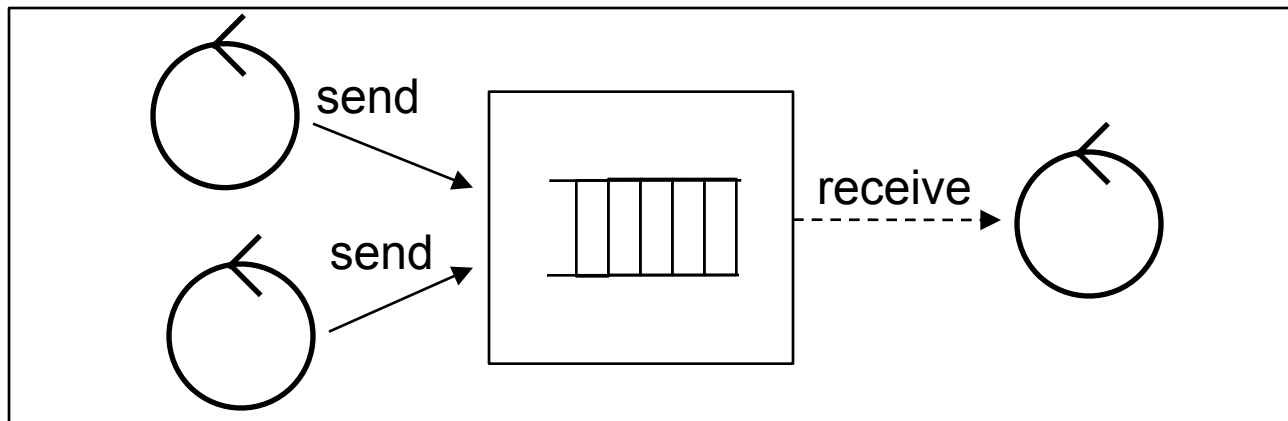
- Proprietà tipiche dei link di comunicazione in caso di comunicazione indiretta:
 - link è stabilito dal momento che i processi condividono una mailbox
 - un link può essere associato a più processi
 - ogni coppia di processi può condividere più link
 - link sia unidirezionali che bidirezionali
- Vari aspetti da decidere nella progettazione.
 - esempio: come gestire lo scenario di comunicazione in cui abbiamo un sender e più receiver che usano la medesima mailbox (link)
 - permettere ad un solo processo alla volta di eseguire una receive
 - lasciare al sistema la selezione non deterministica del receiver. Il mittente è notificato di quale sia stato il receiver selezionato
 - politiche di buffering
 - dimensione buffer della mailbox

COMUNICAZIONE SINCRONA/ASINCRONA

- Lo scambio di messaggi può essere **sincrono** o **asincrono**
 - a cui corrispondono primitive (chiamate di sistema) sincrone/asincrone
- Varianti
 - **synchronous send**
 - invio sincrono, il mittente si blocca fin quando il messaggio è ricevuto
 - **synchronous receive**
 - ricezione sincrona, il ricevente si blocca fin quando non c'è un messaggio disponibile
 - **asynchronous send**
 - invio asincrono, il mittente invia il messaggio e prosegue
 - **asynchronous receive**
 - ricezione asincrona, il ricevente non si blocca per ricevere, ma continua ed eventualmente controlla successivamente l'eventuale arrivo di informazioni

BUFFERING

- Sia nel caso di comunicazione diretta, sia indiretta, i messaggi scambiati da processi comunicanti vengono depositati in code temporanee, che fungono da buffer.
- Ci sono varie possibilità di realizzare tali code, legate in particolare alla dimensione (capacità) del loro buffer:
 - 1. capacità zero - il mittente deve aspettare che il ricevente recuperi il messaggio (**rendezvous**)
 - 2. capacità limitata - esiste un numero massimo di messaggi. Se il buffer è pieno, il mittente aspetta
 - 3. capacità illimitata - il mittente non aspetta mai



COMUNICAZIONE VIA PIPE (UNIX)

- Una dei primi meccanismi di comunicazione introdotti
- In questo modello di comunicazione il medium di comunicazione è dato dalle **pipe** (tubi), canali in cui da un lato processi produttori di informazioni possono inserire/scrivere byte, dall'altro processi consumatori possono recuperare/leggere
 - meccanismo fondamentale di comunicazione nelle shell
- Nella versione di base una pipe:
 - è **monodirezionale**
 - flusso dati in una sola direzione
 - possono essere usati da processi padre/figlio o con genitore comune
- Sono gestite come se fossero file
 - una volta create, vi si inseriscono informazioni mediante primitive che scrivono su file (**write**), e si recuperano / leggono mediante primitive di lettura (**read**)

FORMA DI SINCRONIZZAZIONE

- Le pipe hanno una capacità (settabile con opportune primitive)
 - *lettura da pipe vuota* comporta la **sospensione** del processo in lettura
 - *scrittura su una pipe piena* comporta la sospensione del processo in scrittura
 - lettura o scrittura su una pipe chiusa ritorna errore
- Quindi possono essere usate come meccanismi primitivi di sincronizzazione

USO DI PIPE IN C

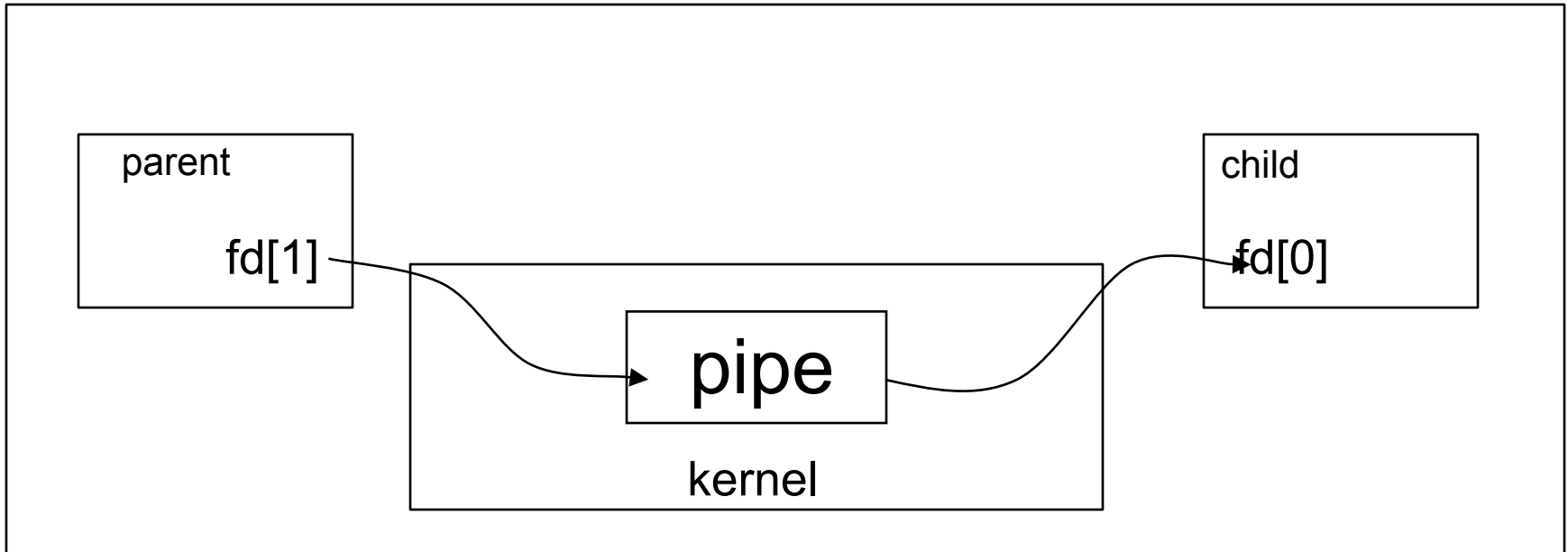
- Una pipe si crea mediante la funzione `pipe` dichiarata in `<unistd.h>`:

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

- L'invocazione comporta la creazione di una pipe e l'apertura di due file descriptor, restituiti in `fd[0]` e `fd [1]`, che rappresentano input e output del canale
- Un processo scrive in output e un processo legge dall'input
 - `fd[0]` è aperto in lettura
 - `fd[1]` è aperto in scrittura
- Un processo che usa la pipe solo in lettura *deve* chiudere il canale in scrittura
 - `close(fd[1])`
- Un processo che usa la pipe solo in scrittura *deve* chiudere il canale in lettura
 - `close(fd[0])`

PIPE FRA PROCESSI

- Esempio di uso di pipe per la comunicazione fra padre (che nella figura scrive in output) e figlio (che legge in input)



PRIMO ESEMPIO

```
#include<unistd.h>
#include<stdio.h>
#define MAXLINE 1024

int main(void){
    int fd[2], n;
    pid_t pid;
    char line[MAXLINE];
    if (pipe(fd)<0){
        fprintf(stderr,"pipe error.\n"); exit(1);
    }
    pid = fork();
    if (pid<0){
        fprintf(stderr,"fork error.\n"); exit(2);
    }
    if (pid>0){ // parent
        printf("PARENT PROCESS: sending data...\n");
        close(fd[0]);
        write(fd[1],"hello world \n",14);
    } else {
        printf("CHILD PROCESS: receiving data...\n");
        close(fd[1]);
        n = read(fd[0],line,MAXLINE);
        printf("MESSAGE from PARENT: %s \n",line);
    }
    exit(0);
}
```

Nell'esempio due processi - padre e figlio - comunicano mediante una pipe creata dal padre. Il padre scrive informazioni nella pipe e il figlio le legge

NOTE

- Al fine di condividere pipe fra processi (padri e figli) devono essere create prima di eseguire le fork
- Le operazioni di lettura e scrittura su pipe sono *atomiche*
 - per cui più processi possono scrivere, ad esempio, sulla stessa pipe senza creare interferenze
 - *in realtà sono atomiche solo se non si richiede la scrittura o lettura di quantità di dati superiori alla dimensione del buffer utilizzato nella pipe*
 - *tipicamente di 8192 bytes, definito in limits.h*

SECONDO ESEMPIO

- Conteggio del numero di occorrenze di un elemento in un array usando 2 processi

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

#define NELEMS 10
int elems[NELEMS] = { 3, 2, 4, 1, 6, 1, 2, 8, 2, 10};

void worker(int from, int to, int value, int ch[2]){
    int i, count = 0;
    close(ch[0]);
    for (i = from; i < to; i++){
        if (elems[i]==value){
            count++;
        }
    }
    write(ch[1], &count, sizeof(count));
    close(ch[1]);
    exit(1);
}

...

int main(void){
    pid_t pid0, pid1;
    int ch[2];
    int count, val;

    int value = 2;

    pipe(ch);
    pid0 = fork();
    if (pid0 == 0){
        worker(0, NELEMS/2, value, ch);
    } else if (pid0 > 0){
        pid1 = fork();
        if (pid1 == 0){
            worker(NELEMS/2, NELEMS, value, ch);
        } else {
            /* padre */
            close(ch[1]);
            count = 0;
            while (read(ch[0], &val, sizeof(val)) > 0){
                printf("received %d \n", val);
                count += val;
            }
            printf("total count: %d\n", count);
            wait(NULL);
            wait(NULL);
        }
    }
    exit(0);
}
```


NOTE SUL SECONDO ESEMPIO

- In questo caso al momento della creazione i processi figli hanno una copia dell'array, dal momento che l'array è dichiarato come variabile globale
 - l'array comunque *non* è condiviso, ogni processo ha la propria copia
- Chiusura dei canali
 - padre e figli subito chiudono la parte del canale che non serve loro
 - alla fine del loro lavoro chiudono anche la parte del canale che hanno usato
- Funzione di sincronizzazione dell'operazione read eseguita dal padre
 - si blocca se la pipe è vuota, ma non è ancora stata chiusa
 - si sblocca restituendo un valore negativo nel caso in cui sia stata chiusa in scrittura da tutti i processi figli che la condividono

TERZO ESEMPIO

- Come l'esempio precedente, tuttavia senza utilizzare array come variabili globali

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

#define SEND(ch, msg) write(ch[1],&msg,sizeof(msg))
#define RECEIVE(ch, msg) read(ch[0],&msg,sizeof(msg))

typedef struct task_msg {
    int len;
    int elems[5];
    int value;    // valore da cercare
} task_msg_t;

void master(int in[2], int out0[2], int out1[2]);
void worker(int in[2], int out[2]);
...
```

```
...
int main(void){
    pid_t pid0,pid1;
    int ch0[2];
    int ch1[2];
    int ch[2];

    pipe(ch);
    pipe(ch0);
    pipe(ch1);
    pid0 = fork();
    if (pid0 == 0){
        close(ch1[0]);
        close(ch1[1]);
        worker(ch0,ch);
    } else if (pid0 > 0){
        pid1 = fork();
        if (pid1 == 0){
            close(ch0[0]);
            close(ch0[1]);
            worker(ch1,ch);
        } else {
            master(ch,ch0,ch1);
        }
    }
    exit(0);
}
```

TERZO ESEMPIO

```
void master(int in[2], int out0[2], int out1[2]){
    int elems[10] = { 3, 2, 4, 1, 6, 1, 2, 8, 2, 10};
    task_msg_t t0,t1;
    int i, count, val;

    close(in[1]);
    close(out0[0]);
    close(out1[0]);

    t0.value = 2;
    t0.len = 5;
    for (i = 0; i < 5; i++){
        t0.elems[i] = elems[i];
    }
    SEND(out0,t0);

    t1.value = 2;
    t1.len = 5;
    for (i = 0; i < 5; i++){
        t1.elems[i] = elems[i+5];
    }
    SEND(out1,t1);

    count = 0;
    while (RECEIVE(in,val)>0){
        printf("received %d \n",val);
        count+=val;
    }
    printf("total count: %d\n",count);
    wait(NULL);
    wait(NULL);
    exit(0);
}
```

```
void worker(int in[2], int out[2]){
    int i, count = 0;
    task_msg_t t;

    close(out[0]);
    close(in[1]);

    RECEIVE(in,t);
    for (i = 0; i < t.len; i++){
        if (t.elems[i]==t.value){
            count++;
        }
    }
    SEND(out,count);

    close(out[1]);
    close(in[0]);
    exit(0);
}
```

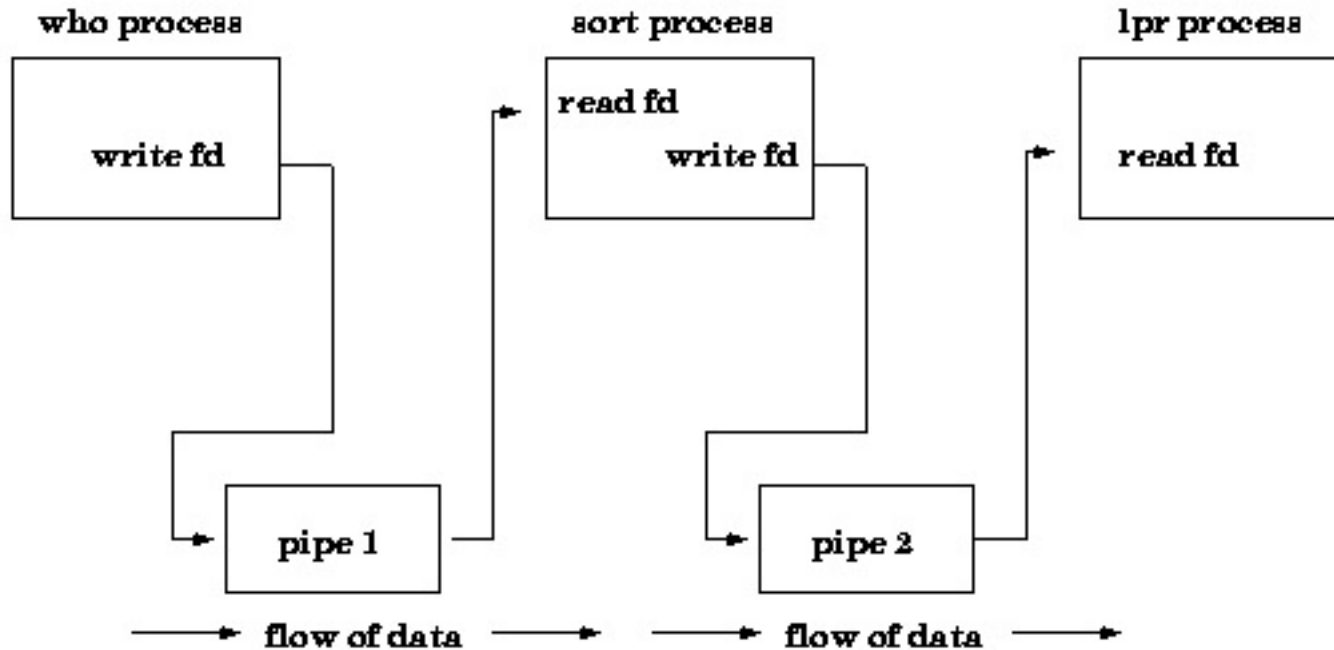
NOTE SUL TERZO ESEMPIO

- Valori dell'array comunicati via pipe dal master ai due worker
- Utilizzo di un canale come message box per i processi
 - utilizzato in scrittura dai processi che vogliono spedire messaggi ad un processo
 - utilizzato in lettura dal solo processo proprietario della message box

PIPE E SHELL

- Le pipe sono usate in modo pervasivo in UNIX
- Ad esempio nella combinazione di comandi eseguiti da shell:

```
who | sort | lpr
```



VARIANTI DELLE PIPE

- FIFO
 - pipe il cui utilizzo è esteso a processi che non condividono genitori diretti comuni
- STREAM PIPE
 - pipe bidirezionali
- NAMED STREAM PIPE
 - pipe bidirezionali indentificate da un nome, utilizzabili da qualsiasi coppia di processi

SHARED MEMORY

- Modello di comunicazione indiretta, che avviene mediante l'utilizzo da parte dei processi comunicanti di una porzione di memoria condivisa (shared memory)
- E' un modello di IPC molto performante (per ciò che concerne ambienti non distribuiti), in quanto non comporta la copia di dati fra processi comunicanti
- Tuttavia richiede meccanismi di sincronizzazione per coordinare l'accesso concorrente dei processi ai dati condivisi
 - tipicamente semafori, illustrati nel modulo su sincronizzazione e coordinazione

SHARED MEMORY IN UNIX

- In ambito Unix la memoria condivisa viene gestita a **segmenti**, e si hanno a disposizione funzioni per ottenere un segmento di memoria condivisa specificando le dimensioni e per accedervi
 - i segmenti sono identificati da un numero intero positivo, rilasciato dal kernel
 - dopo essere stato creato, un segmento di memoria condivisa può essere usato da uno o più processi che lo inseriscono (**attach**) fra i propri segmenti di memoria
- Il kernel utilizza la struttura dati `struct_ds` per mantenere le informazioni circa un segmento creato:

```
struct shmid_ds {
    int shm_segsz;    /* dimensioni in byte */
    pid_t shm_cpid;  /* pid del creatore */
    ulong shm_nattch; /* numero di attach correnti */
    ....
}
```


API PER UTILIZZO DI SHARED MEMORY (1/4)

- Le funzioni C messe a disposizione per la gestione della shared memory sono descritte in `<sys/ipc.h>`, `<sys/shm.h>` e `<sys/types.h>`
- Fra le funzioni: **creazione di un segmento** di shared memory:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

int shmget(key_t key, int size, int flag)
```

crea un segmento di memoria condivisa della dimensione specificata, restituendone un identificatore

- *key* è una costante intera che fornisce informazioni specifiche sul meccanismo di IPC stabilito
 - nel caso di memoria condivisa generalmente si usa `IPC_PRIVATE`, che comporta la creazione di una nuova struttura IPC per la gestione del segmento
- *flag* è una maschera di bit che indica se il segmento è di lettura (`SHM_R`), scrittura (`SHM_W`) o entrambi (`SHM_R | SHM_W`)

API PER UTILIZZO DI SHARED MEMORY (2/4)

- La funzione che permette di eseguire tutte le varie operazioni di gestione del segmento è

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

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

- Restituisce 0 se OK, -1 se problemi
- cmd può essere la costante
 - IPC_STAT - recupera le info sul segmento, memorizzandole nella struttura puntata da buf
 - IPC_SET - setta proprietà del segmento, specificate nella struttura puntata da buf
 - IPC_RMID - rimuove il segmento dal sistema. Se il numero di attach non è 0 allora il segmento viene solo scollegato, altrimenti viene distrutto
 - SHM_LOCK - lock del segmento (può farlo solo il superuser)
 - SHM_UNLOCK - unlock del segmento (solo superuser)

API PER UTILIZZO DI SHARED MEMORY (3/4)

- Dopo esser stata creato, un segmento deve essere *agganciato* (**attached**) dai vari processi per poterci scrivere / leggere. Allo scopo c'è la funzione:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

void* shmat(int shmid, void* addr, int flag);
```

- La funzione aggancia un segmento ad un processo
 - restituisce il puntatore alla shared memory se OK, oppure -1 se problemi
 - addr specifica l'indirizzo di memoria nello spazio di indirizzamento del processo ove attaccare il segmento
 - se addr è NULL, allora il segmento viene agganciato al primo indirizzo libero scelto dal kernel (scelta consigliata).
 - flag è utilizzato solo se addr non è null, in caso contrario è 0

API PER UTILIZZO DI SHARED MEMORY (4/4)

- Infine è fornita una funzione con cui si 'sgancia' (**detach**) il segmento dal processo

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

int shmdt(void* addr);
```

- Restituisce 0 se OK, -1 se problemi
- Scollega il segmento di memoria condivisa collocato all'indirizzo restituito dalla chiamata `shmat`

ESEMPIO

- Come esempio, nel programma `testsm.c` processo padre e processo figlio condividono un segmento di shared memory della dimensione specificata come parametro in ingresso
 - Il figlio scrive nella memoria condivisa, nella fattispecie un array di elementi interi
 - il padre - terminato il figlio - legge e visualizza in standard output

SORGENTE testsm.c

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char* argv[]) {
    int shmid;
    char* shmptr;
    int* array;
    int size,i;
    pid_t pid;

    if (argc<2) {
        fprintf(stderr,"Arguments: <SharedMemorySegment Size>\n");
        exit(1);
    }
    size = atoi(argv[1]);
    shmid = shmget(IPC_PRIVATE, size, (SHM_R | SHM_W)); /* crea il segmento */
    if (shmid<0){
        fprintf(stderr,"Impossible to create a shared
                    memory segment of size %d \n",size);
        exit(2);
    }

    printf("The shared memory segment id is  %d.\n",shmid);
    ...
}
```

SORGENTE testsm.c: PADRE

```
...
pid = fork();

if (pid>0){ /* PADRE */
    shmptr = shmat(shmid,0,0); /* aggancia il segmento */

    if (shmptr== (void*)-1){
        fprintf(stderr,"Error in attaching the shared memory segment.\n");
        exit(3);
    }

    printf("PARENT: shared memory attached at %x \n",shmptr);
    array=(int*)shmptr;

    wait(NULL); /* ASPETTA CHE IL FIGLIO ABBIA CONCLUSO */

    printf("PARENT: array content:\n");
    for (i=0; i<10; i++){
        printf("%d ",array[i]); /* legge la memoria condivisa */
    }
    printf("...\n");

    if (shmctl(shmid,IPC_RMID,0)<0){ /* RIMOZIONE */
        fprintf(stderr,"Error in removing shared memory.\n");
        exit(4);
    }
} else ...
```

SORGENTE testsm.c: FIGLIO

```
...
} else if (pid==0){ /* FIGLIO */
    shmptr = shmat(shmid,0,0); /* aggancia il segmento */

    if (shmptr== (void*)-1){
        fprintf(stderr,"Error in attaching the shared memory segment.\n");
        exit(3);
    }

    printf("CHILD: shared memory attached at %x \n",shmptr);
    array=(int*)shmptr;

    printf("CHILD: filling the array.\n");
    for (i=0; i<10; i++){ /* scrive sulla memoria condivisa */
        array[i] = i*2;
    }
    exit(0);
}
}
```


ESECUZIONE DI `testsm.c`

- Si compila e linka al solito con

```
$ gcc testsm.c -o testsm
```

- Quindi si esegue:

```
$ ./testsm 1000000
```

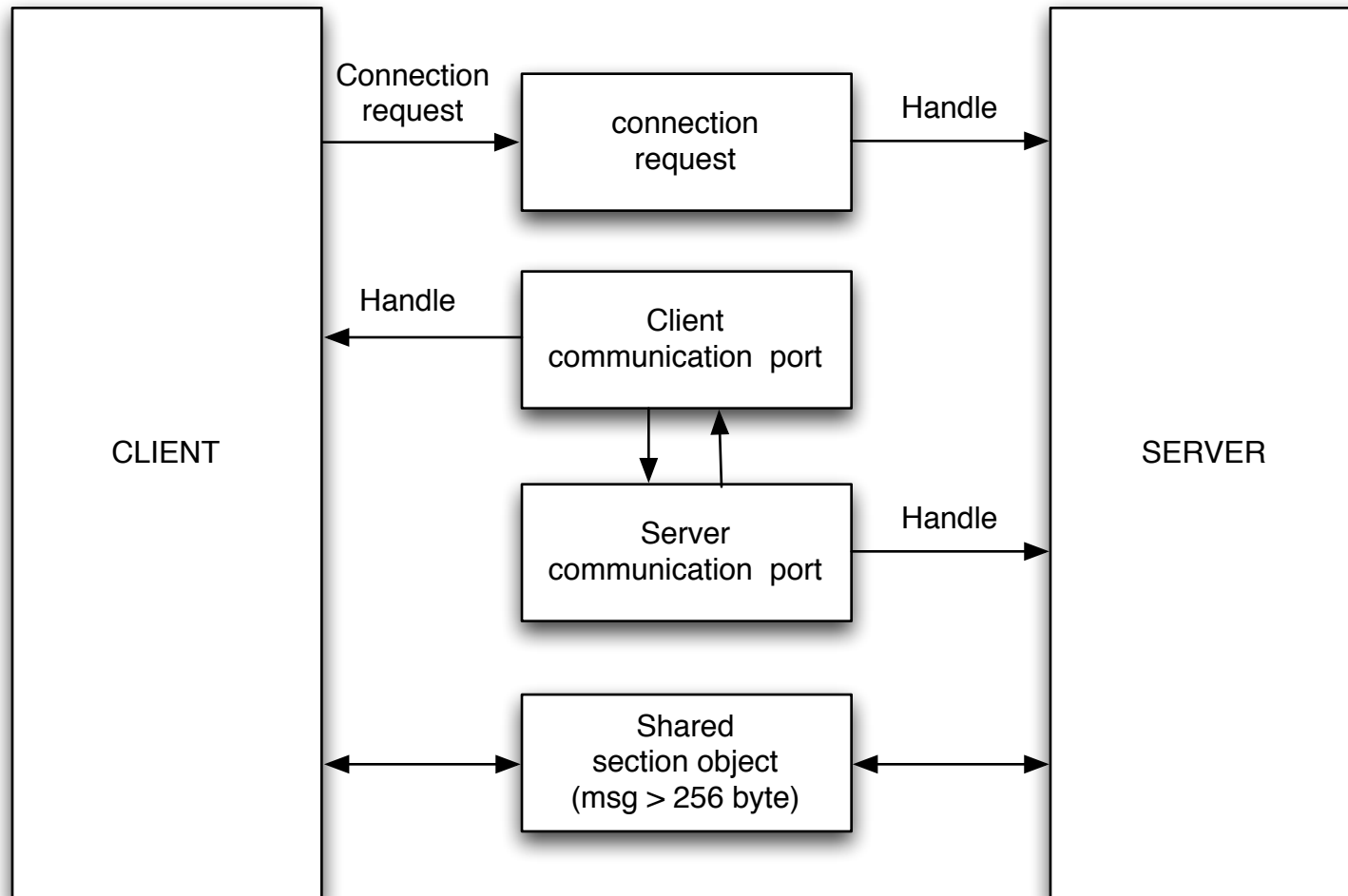
- Risultato (gli indirizzi di allocazione del segmento possono cambiare da sistema a sistema):

```
The shared memory segment id is 524291.  
PARENT: shared memory attached at 28000  
CHILD: shared memory attached at 28000  
CHILD: filling the array.  
PARENT: array content:  
0 2 4 6 8 10 12 14 16 18 ...
```

IPC SU WINDOWS XP

- Supporto per comunicazione basata su message-passing mediante un meccanismo chiamato **local procedure call (LPC)**
 - schema *client-server* ma in locale
- La comunicazione avviene attraverso dei *port object*, che fungono da canali di comunicazione
 - connection port
 - visibili a tutti i processi, servono per creare canali di comunicazione
 - communication port
 - utilizzati per comunicazione
- Passi per far comunicare un client A e un server B
 - un client apre un handle ad una specifica connection port e invia una richiesta di connessione
 - il server crea due porte private e restituisce l'handle di una delle due al client
 - il client e il server utilizzano i corrispondenti port handle per inviare e ricevere messaggi
- Due tecniche per lo scambio di messaggi
 - per piccoli messaggi (< 256), viene utilizzata la coda associata ad ogni porta, trasferendo l'informazione da una porta all'altra
 - per messaggi di grandi dimensioni, viene creata una regione di memoria condivisa (**section object**)

LPC SU WINDOW XP

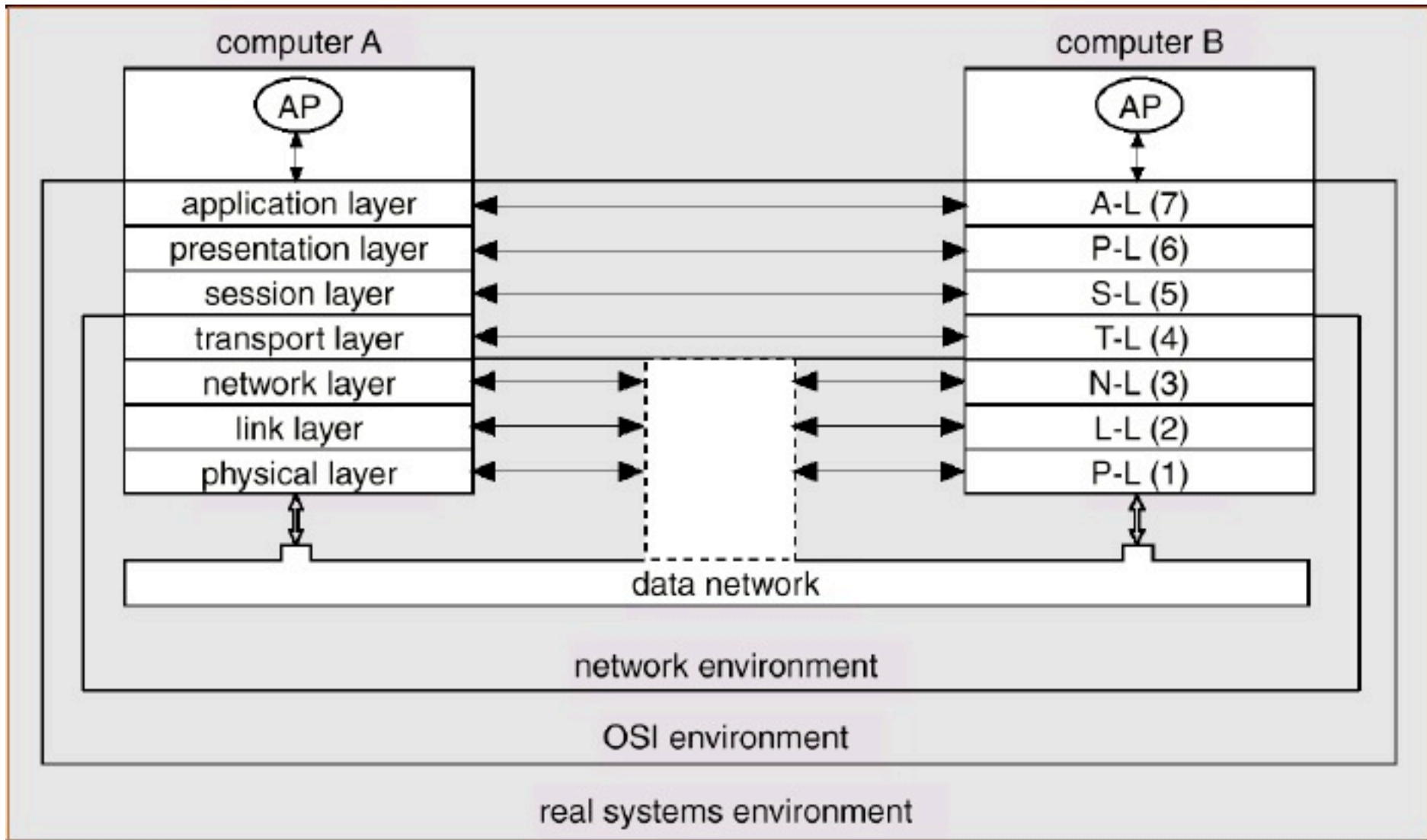


COMUNICAZIONE VIA RETE

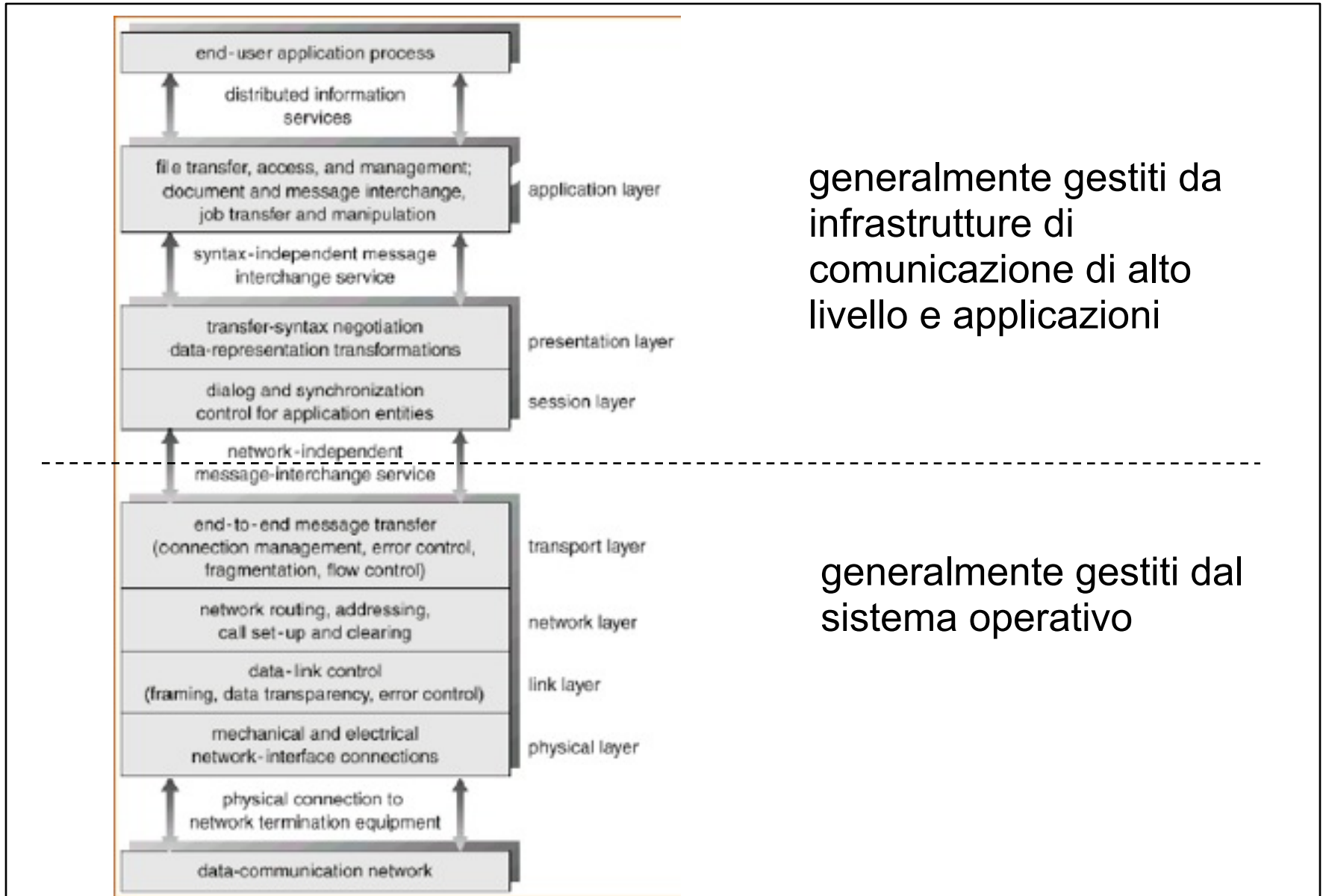
COMUNICAZIONE VIA RETE

- Modelli / meccanismi molto diffusi per la comunicazione inter-processo sono quelli utilizzati più in generale per la comunicazione in sistemi distribuiti, ove i processi (thread) possono risiedere su computer distinti (o macchine virtuali distinte), collegati in rete
- Tali modelli in genere prevedono protocolli necessari per gestire l'identificazione dei processi in un ambito distribuito (**naming**), e il trasporto del messaggio da computer a computer
 - tale trasporto richiede tipicamente l'instradamento delle informazioni su un percorso di rete (**routing**) che può includere più computer 'di passaggio' (*hop*), che agiscono da instradatori e smistatori
 - questo aspetto verrà analizzato nel dettaglio nel corso di Reti di calcolatori: in questa sede ne consideriamo solo gli aspetti utili come modello di comunicazione fra processi
- Fra i modelli di comunicazione distribuita vedremo socket, RPC, RMI (RPC in ambito oggetti) e qualche accenno a CORBA come generalizzazione di RMI

LIVELLI DI RETE



LIVELLI ISO/OSI PER PROTOCOLLI DI RETE



generalmente gestiti da infrastrutture di comunicazione di alto livello e applicazioni

generalmente gestiti dal sistema operativo

SOCKET

















- Una **socket** è definita come in generale porta di comunicazione (*communication endpoint*), gestita dal sistema operativo.
 - due processi possono comunicare mediante una coppia di socket, una per ogni processo
 - la comunicazione avviene creando un communication link fra due socket e consiste nell'invio/ricezione di insiemi di byte.
- Protocollo IP
 - Il protocollo standard-de-facto più diffuso e utilizzato nei S.O. moderni per la comunicazione inter-processo distribuita è basato su socket che comunicano su infrastruttura di rete Internet
 - livello di rete
 - nel protocollo IP una socket è caratterizzata dall'**indirizzo IP** del computer sui cui risiede e da un numero intero (32 bit), che identifica una porta

INDIRIZZI IP

- L'indirizzo IP è un intero di 4 byte con cui si identifica univocamente un *host*
 - l'indirizzo IP è rappresentato spesso come 4 byte scritti in notazione decimale, separati da punti: es: 137.204.107.188.
- Tale notazione permette più agilmente di determinare informazioni topologiche, ovvero la locazione del computer nelle varie reti, sottoreti, domini.
 - ad esempio: 137.204.107.188 è un PC della sottorete DEIS di Cesena (107), del dominio dell'università di Bologna (204), in una dorsale italiana (137).
- Al posto dell'indirizzo IP è possibile usare anche (quando è definito) un nome logico
 - es: <http://corsi.ing2.unibo.it>, a cui è associato l'indirizzo IP fisico.
- L'associazione nome logico / fisico è mantenuta e risolta dinamicamente da appositi server in rete chiamati DNS (Domain Name Servers)

BLOCCHI DI INDIRIZZI PREDEFINITI

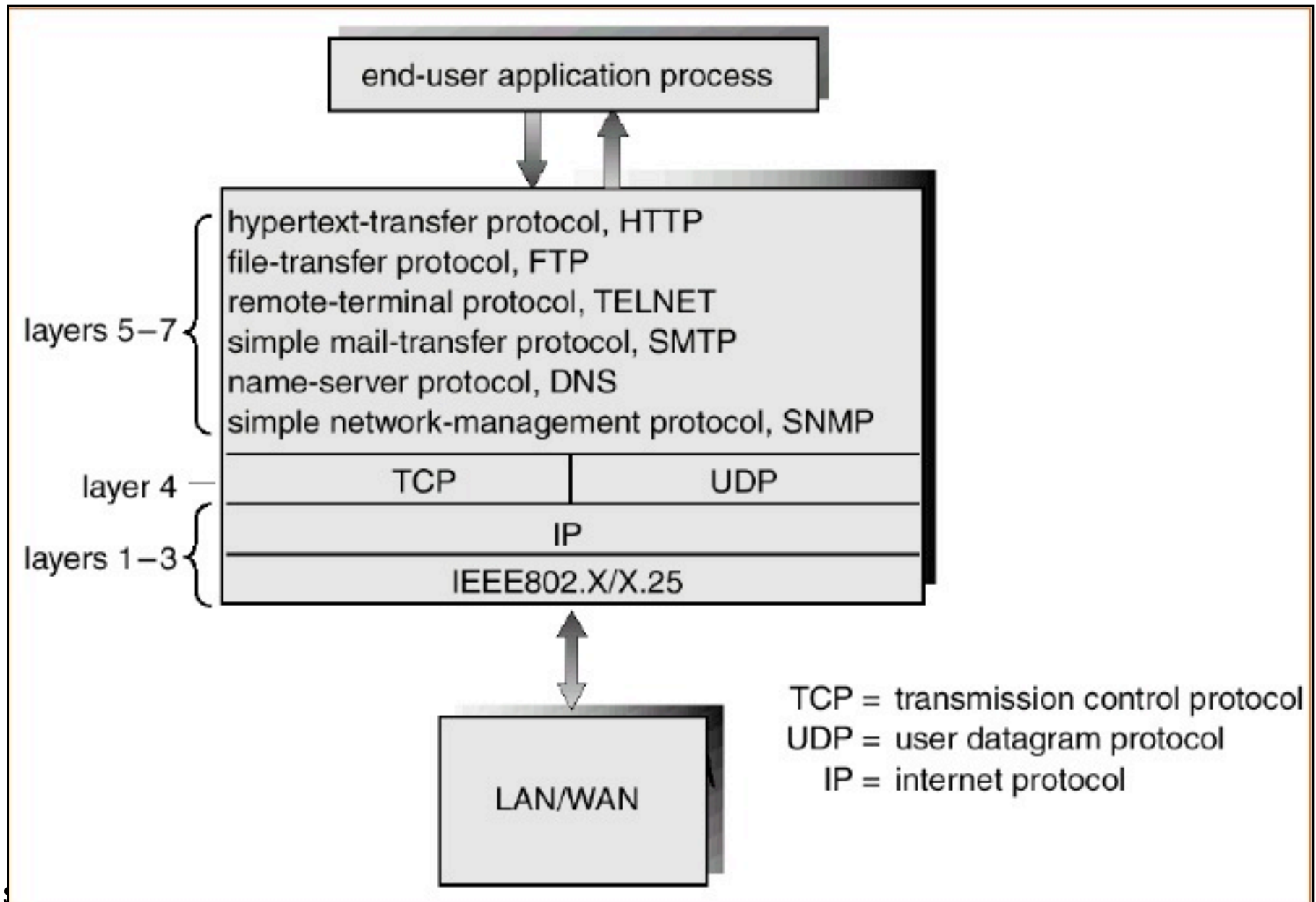
Reserved address blocks

CIDR address block	Description	Reference
0.0.0.0/8	Current network (only valid as source address)	RFC 1700 
10.0.0.0/8	Private network	RFC 1918 
14.0.0.0/8	Public data networks (per 2008-02-10, available for use ^[1])	RFC 1700 
127.0.0.0/8	Loopback	RFC 3330 
128.0.0.0/16	Reserved (IANA)	RFC 3330 
169.254.0.0/16	Link-Local	RFC 3927 
172.16.0.0/12	Private network	RFC 1918 
191.255.0.0/16	Reserved (IANA)	RFC 3330 
192.0.0.0/24	Reserved (IANA)	RFC 3330 
192.0.2.0/24	Documentation and example code	RFC 3330 
192.88.99.0/24	IPv6 to IPv4 relay	RFC 3068 
192.168.0.0/16	Private network	RFC 1918 
198.18.0.0/15	Network benchmark tests	RFC 2544 
223.255.255.0/24	Reserved (IANA)	RFC 3330 
224.0.0.0/4	Multicasts (former Class D network)	RFC 3171 
240.0.0.0/4	Reserved (former Class E network)	RFC 1700 
255.255.255.255	Broadcast	

SOCKET TCP e UDP

- Sul protocollo a livello di rete IP esistono diversi protocolli che determinano la natura del communication link che è possibile stabilire fra due socket (*il livello di trasporto*).
- I due principali utilizzati nelle applicazioni sono
 - **UDP (modalità connection-less)** - per la realizzazione di comunicazione basate su scambio di messaggi, di dimensione limitata, senza garanzie di consegna e ordine
 - **TCP (modalità connection-oriented)** - per la realizzazione di connessioni con invio di flusso di informazioni, con garanzie di ordinamento e consegna
- A questo livello esistono altri protocolli utilizzati per lo più per amministrazione / controllo della rete stessa (es: ICMP), oppure per applicazioni specifiche (es: streaming)
- Lo stack di protocolli (UDP | TCP) / IP è gestito direttamente dal sistema operativo e sfruttato dalle applicazioni costruite sopra

STACK DI PROTOCOLLI TCP/IP e UDP



SCAMBIO MESSAGGI VIA UDP

- Mediante il protocollo UDP è possibile realizzare la comunicazione basata su scambio di messaggi
- I messaggi prendono il nome di datagrammi (**datagram**), e sono costituiti da pacchetti di byte di dimensione variabile
 - dimensione dei messaggi limitata (64kb)
 - nessuna garanzia di consegna del messaggio
 - nessuna garanzia nell'ordinamento invio / ricezione messaggi
 - molto performante

DINAMICHE DELLA COMUNICAZIONE VIA SOCKET UDP

- Per scambiare messaggi via socket UDP processi devono creare una o più socket, conoscere l'indirizzo delle socket a cui vogliono inviare messaggi mediante una delle socket, ricevere messaggi su una delle socket
- Per scambiare messaggi attraverso socket UDP i sistemi operativi mettono a disposizione servizi principalmente per
 - creare socket (e distruggerla), legandola ad una porta locale
 - ricevere un messaggio da una socket
 - inviare un messaggio (mediante una socket) ad una socket di un dato indirizzo
- La richiesta di ricezione di un messaggio su una socket è sincrono, e comporta la sospensione del processo (thread) fino a quando non sia disponibile un messaggio
 - varianti temporizzate

SOCKET UDP IN JAVA

- In Java il package `java.net` mette a disposizione le API per far la comunicazione fra processi via socket
- Le classi in generale utilizzate per
 - **InetAddress** - rappresenta un indirizzo IP
 - **InetSocketAddress** - rappresenta l'indirizzo completo di una socket (IP + porta)
- In particolare per la comunicazione via socket UDP le principali classi interessate sono
 - **DatagramPacket** - rappresenta un datagramma
 - **DatagramSocket** - rappresenta una socket per inviare o ricevere messaggi
 - metodi per inviare e ricevere datagrammi
 - **MulticastSocket** - rappresenta socket multicast
- Vedere per dettagli la documentazione Java

ESEMPIO: SCAMBIO MESSAGGI VIA UDP

- Esempio: due processi - sender e receiver - che comunicano mediante invio di messaggi tramite socket UDP
- L'applicazione receiver (classe modulo2a.Receiver) ascolta messaggi - in questo caso stringhe - in arrivo sulla porta specificata eventualmente come parametro della applicazione (di default la 8888) e quindi li stampa in standard output
 - Il messaggio 'stop' fa terminare il processo
- L'applicazione Sender invia un determinato messaggio testuale ad un determinato indirizzo:porta

Receiver (1/2)

```
public class Receiver {

    public static void main(String[] args) throws Exception {
        int port = 8888;
        String s = getOption(args, "-port");
        if (s!=null){
            try {
                port = Integer.parseInt(s);
            } catch (Exception ex){
                System.out.println("Porta errata.");
                System.exit(1);
            }
        }
        try {
            doWork(port);
        } catch (Exception ex){
            System.out.println("Socket non disponibile.");
            System.exit(2);
        }
    }

    private static void doWork(int port) throws java.net.SocketException {...}

    private static String getOption(String[] args,String prefix){
        for (int i=0; i<args.length; i++)
            if (args[i].equals(prefix)){
                return args[i+1];
            }
        return null;
    }
}
```

Receiver (2/2)

```
private static void doWork(int port) throws java.net.SocketException {
    log("Start working (port "+port+").");

    boolean stop = false;
    DatagramSocket socket = new DatagramSocket(port);
    byte[] msgBuffer = new byte[1024*64];
    DatagramPacket currentMsg = new DatagramPacket(msgBuffer,msgBuffer.length);

    while (!stop) {
        try {
            socket.receive(currentMsg);
            int nbytes = currentMsg.getLength();
            byte[] data = currentMsg.getData();
            String msg = new String(data,0,nbytes);
            log("New message: "+msg+" from "+currentMsg.getSocketAddress());
            if (msg.equals("stop")){
                stop = true;
            }
        } catch (Exception ex){
            log("Errore nella comunicazione.");
        }
    }
    log("Shutdown.");
}
```

Sender

```
package modulo2a;
import java.net.*;
public class Sender {
    public static void main(String args[]){
        if (args.length<2){
            System.err.println("Arguments:  <IPAddress:Port>  <Msg>");
            System.exit(1);
        }
        try {
            byte[] info = args[1].getBytes();
            InetAddress address = getAddress(args[0]);
            DatagramPacket msg = new DatagramPacket(info,info.length,address);
            DatagramSocket socket = new DatagramSocket();
            socket.send(msg);
            System.out.println("Message sent.");
        } catch (Exception ex){
            ex.printStackTrace();
            System.err.println("\nMessage not sent. ");
        }
    }

    static InetAddress getAddress(String addr) throws Exception {
        int i = addr.indexOf(':');
        String ip_address = addr.substring(0,i);
        int port = Integer.parseInt(addr.substring(i+1,addr.length()));
        return new InetAddress(ip_address,port);
    }
}
```

TEST DEL SISTEMA

- E' possibile testare il sistema anche lanciando i due processi (applicazioni) sullo stesso host, e specificando localhost come indirizzo (logico) IP
 - Esempio di lancio di un Receiver:

```
java modulo2a.Receiver -port 5000
```
 - Esempio di lancio di un Sender:

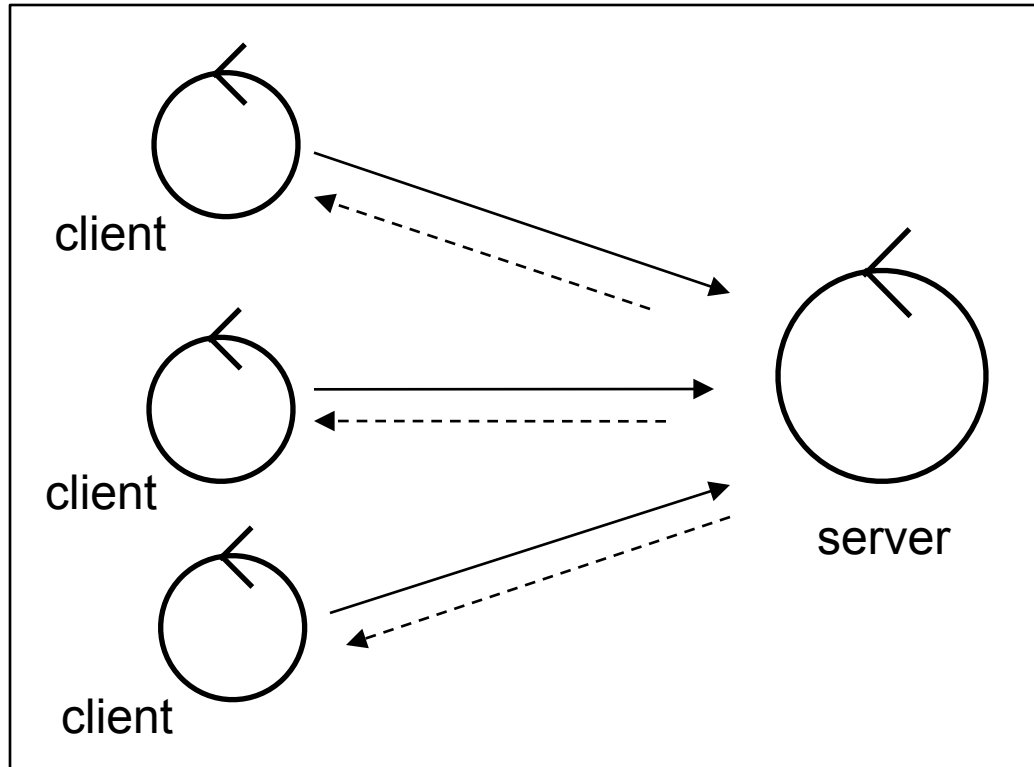
```
java modulo2a.Sender localhost:5000 'this is a test'
```
- (Nota: sender e receiver devono essere lanciati da shell diverse (oppure dalla stessa shell in parallelo), dal momento che il receiver non termina subito)

INVIO OGGETTI E DATI COME MESSAGGI

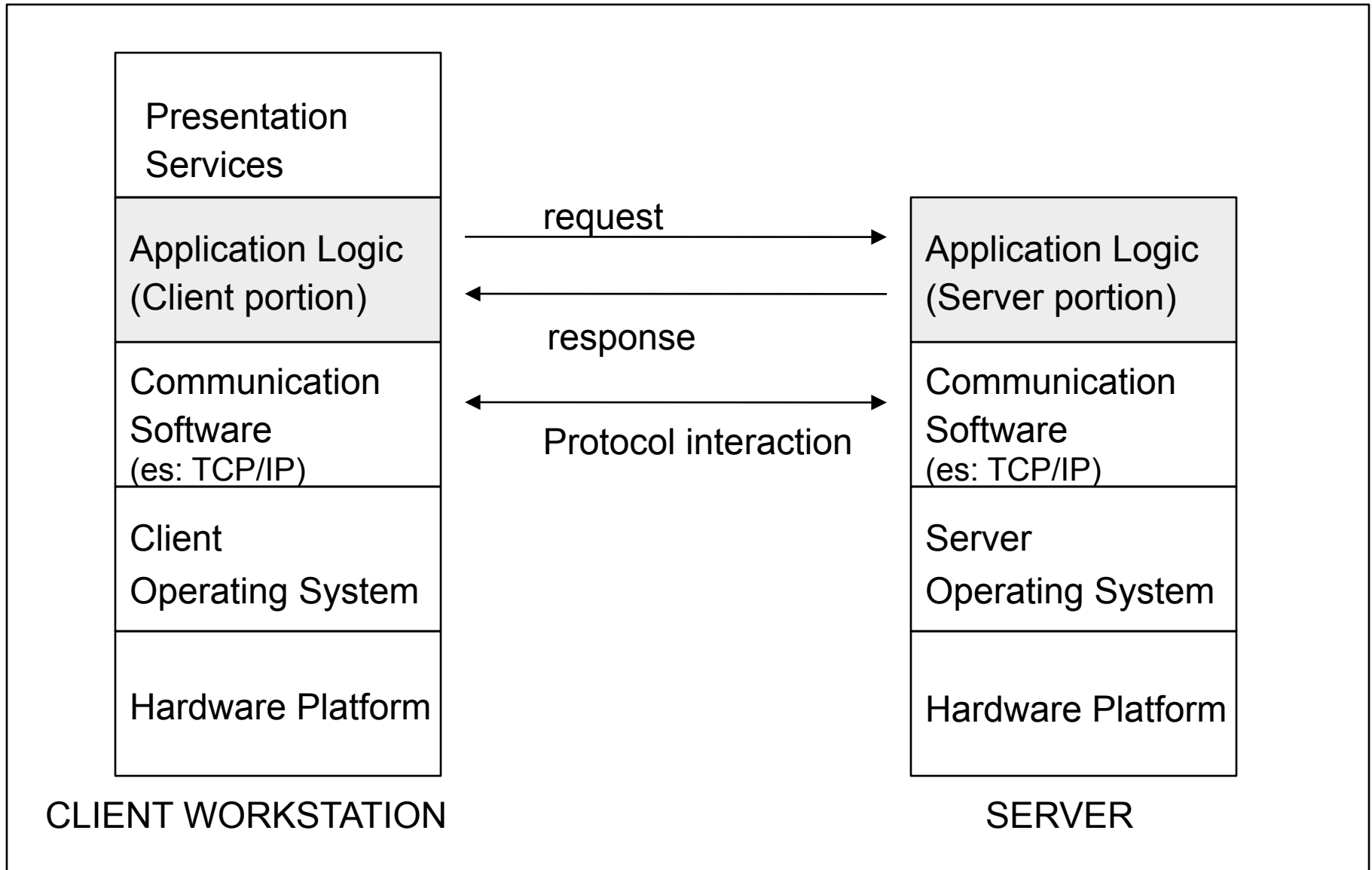
- E' possibile sfruttare gli stream di incapsulamento di Java per inviare - impacchettati come byte - tipi di dati primitivi (sfruttando `java.io.DataOutputStream` e `java.io.DataInputStream`) e oggetti *serializzabili* (sfruttando `ObjectOutputStream` e `ObjectInputStream`)
- L'effetto è di scambiare messaggi via socket UDP che contengono direttamente valori 'tipati' e oggetti Java

COMUNICAZIONE CLIENT-SERVER

- La comunicazione è una componente fondamentale per la realizzazione di **sistemi client-server**, in cui uno o più processi client interagiscono con uno o più server (servitori) al fine di ottenere qualche *servizio*.
 - Esempi: Web Server, Time Service, File Servers, ...



ARCHITETTURA CLIENT-SERVER



SISTEMI CLIENT-SERVER CON SOCKET

- Un processo server può essere in ascolto su una determinata socket.
- Processi client contattano il server richiedendo una connessione alla porta dove è in ascolta il server.
- Ad esempio:
 - (server) Web Server su corsi.ing2.unibo.it è in ascolto sulla porta 80
 - (client) Web browser su un computer di indirizzo 137.204.107.189
- Al client che chiede la connessione alla porta del server viene automaticamente assegnata una porta locale (es: 2000), e quindi creata una socket che il client può usare per ricevere e inviare informazioni al server.

CLIENT/SERVER VIA TCP

- A differenza del protocollo di trasporto UDP, mediante il protocollo **TCP (connection-oriented socket)** si stabilisce un vero e proprio canale o **stream** fra due socket, con garanzie in termini di qualità del servizio
 - garanzie di consegna
 - ordine invio / ricezione preservato
 - nessun limite nell'invio delle informazioni
- L'interazione via protocollo TCP è tipicamente client-server
 - un processo / thread che funge da server è in attesa di ricevere connessioni su una **server socket**
 - un processo client contatta il server chiedendo una connessione alla server socket
 - il server ascolta la richiesta, creando una nuova socket e creando una nuova connessione con la socket del client

SOCKET TCP IN JAVA

- Per la comunicazione via socket TCP le principali classi interessate nel package `java.net` sono
 - **ServerSocket** - rappresenta una server socket
 - `accept`
 - **Socket** - rappresenta una socket client
 - metodi per inviare e ricevere informazioni come stream di byte
 - Supporto Java con stream di incapsulamento per inviare e ricevere dati primitivi e oggetti
- Lo schema generale di comunicazione (caso semplice):
 - il server crea un oggetto **ServerSocket** su una determinata porta locale (es: 6120) e vi si mette in ascolto mediante l'operazione **accept**, al fine di ricevere connessioni, sospendendo la propria esecuzione
 - il client crea un oggetto **Socket** (specificando o meno la porta locale) e ne chiede la connessione alla socket server mediante l'operazione **connect**
 - alla ricezione della richiesta, `accept` si sblocca e ritorna un oggetto di classe `Socket` che può essere usata per la comunicazione con il client

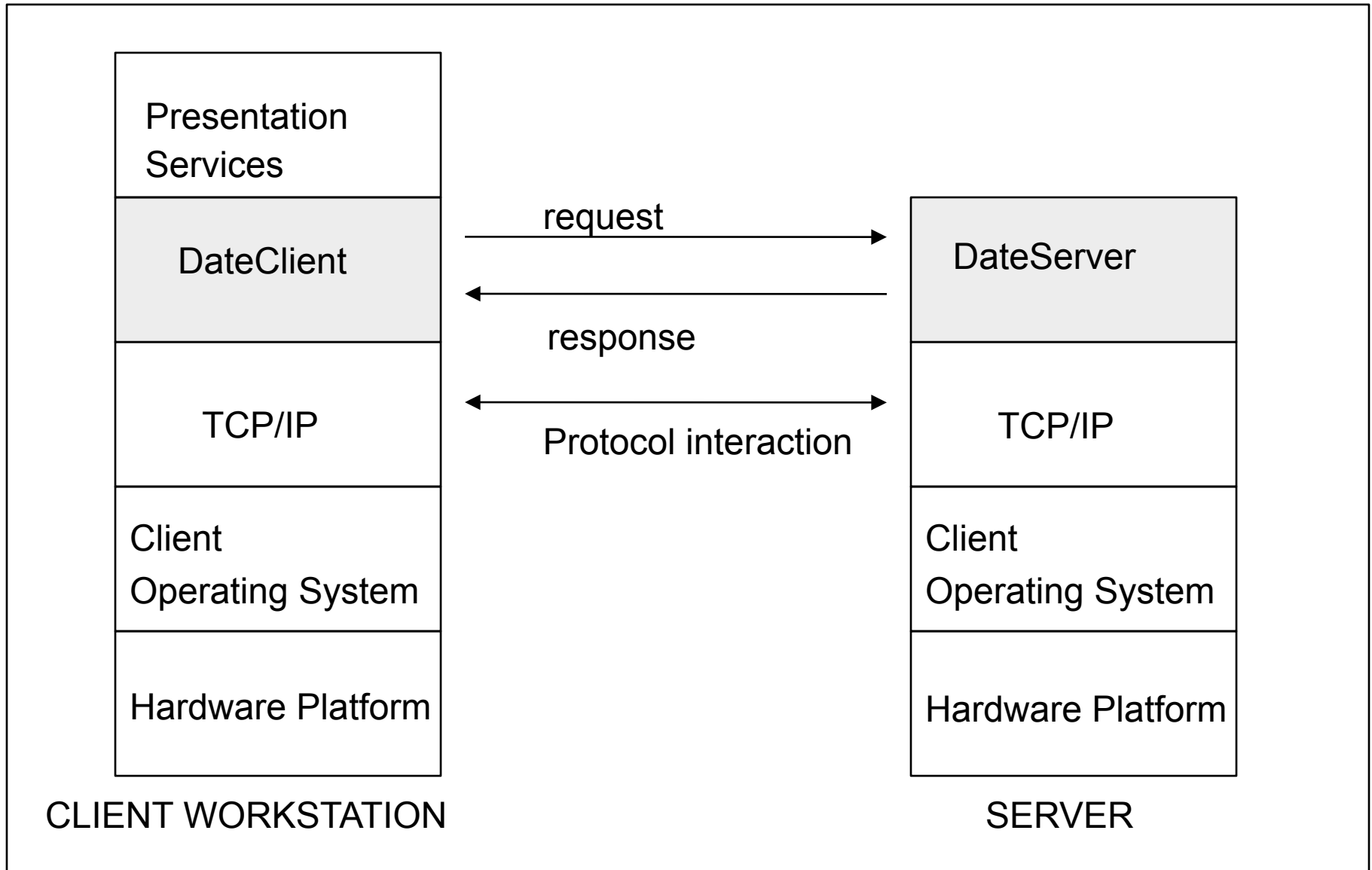
SOCKET E STREAM

- Ad ogni socket TCP (istanza della classe Socket) sono associati due stream, uno stream di input e uno stream di output
 - lo stream di input è un oggetto di classe `java.io.InputStream` e si recupera mediante metodi `getInputStream`
 - lo stream di output è un oggetto di classe `java.io.OutputStream` e si recupera mediante metodi `getOutputStream`
- La ricezione di byte sulla socket avviene leggendo dallo stream di input (mediante metodi `read`). Analogamente l'invio di byte sulla socket avviene scrivendo sullo stream di output (mediante metodi `write`)
 - nel caso in cui si eseguano delle letture su stream in cui non ci sono dati disponibili, il processo si blocca in attesa di dati disponibili
- E' frequente l'utilizzo di stream di incapsulamento, per inviare e ricevere direttamente dati 'tipati' e oggetti
 - vedere `DataInputStream / DataOutputStream` e `ObjectInputStream / ObjectOutputStream`

ESEMPIO: UN SERVIZIO DATE

- Come semplice esempio realizziamo un servizio Date: client contattano un server per ottenere informazioni sulla data (incluso il tempo) corrente
 - Il processo server è realizzato dalla classe modulo2a.DateServer
 - Il client è realizzato dalla classe modulo2a.DateClient
- Il server crea una server socket e rimane in attesa di ricevere connessioni sulla porta 6013. All'arrivo di una richiesta, viene creata (automaticamente dalla accept) una socket per dialogare con il client. Il dialogo in questo caso si risolve con l'invio di una stringa (usando il metodo writeUTF fornito dallo stream di incapsulamento DataOutputStream costruito sull'output stream della socket) che rappresenta la data corrente (nel formato UCT, inclusiva del tempo).
- Il client crea una socket che connette all'indirizzo specificato come argomento, alla porta 6013, da cui legge via input stream wrappato con un DataInputStream una stringa (la data) che visualizza poi in output

SERVIZIO DATE



PROCESSO SERVER

```
package modulo2a;
import java.net.*;
import java.io.*;
public class DateServer {
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);
            log("Waiting for request on port "+sock.getLocalPort());
            while (true){
                Socket client = sock.accept();
                try {
                    log("New request from "+client.getRemoteSocketAddress());
                    DataOutputStream out = new DataOutputStream(client.getOutputStream());
                    out.writeUTF(new Date().toString());
                } catch (Exception ex){
                    log("Wrong request.");
                } finally {
                    client.close();
                }
            }
        } catch (Exception ex){ ex.printStackTrace();}
    }
    private static void log(String msg){
        System.out.println("[DATE SERVER] "+msg);
    }
}
```

PROCESSO CLIENT

```
package modulo2a;

import java.io.*;
import java.net.*;

public class DateClient {
    public static void main(String[] args) throws Exception {
        if (args.length<1){
            System.err.println("Arguments: <DateServerAddress>");
            System.exit(1);
        }
        Socket sock = new Socket();
        sock.connect(new InetSocketAddress(args[0],6013));

        DataInputStream input = new DataInputStream(sock.getInputStream());
        String data = input.readUTF();
        System.out.println("Server date: "+data);
        sock.close();
    }
}
```

TEST

- Per testare il sistema, prima installare il server (ad esempio su localhost):

```
java modulo2a.DateServer
```

- Poi in una shell a parte eseguire

```
java modulo2a.DateClient localhost
```

- Nel caso in cui si lanci da un host diverso, si deve specificare l'indirizzo IP reale del server:

```
java modulo2a.DateClient corsi.ing2.unibo.it
```


SISTEMI CLIENT-SERVER BASATI SU SCAMBIO DI MESSAGGI

- Vediamo il medesimo esempio realizzato in termini di scambio di messaggi mediante socket UDP
- Strategia per il server: crea una socket legata alla porta 6013 e attende di ricevere su essa messaggi di richiesta. Arrivato un messaggio di richiesta, crea un messaggio di risposta in cui inserisce informazioni sulla data corrente e quindi si rimette in attesa di una nuova richiesta
- Strategia per il client: crea una socket mediante la quale invia un messaggio di richiesta alla socket del server, e quindi attende (per max 5 secondi) il messaggio di risposta. Arrivato il messaggio di risposta, ne estrae il contenuto informativo (la data) e la visualizza in standard output

DateServerUDP (1/2)

```
package modulo2a;

import java.net.*;
import java.util.*;

public class DateServerUDP {
    public static void main(String[] args) {
        try {
            DatagramSocket socket = new DatagramSocket(6013);
            // packet for receiving client request (empty datagrams)
            DatagramPacket receivePacket = new DatagramPacket(new byte[1],1);

            log("Waiting for request on port "+socket.getLocalPort());
            while (true){

                // wait client msg request
                socket.receive(receivePacket);

                try {
                    InetAddress remoteAddr = receivePacket.getAddress();
                    int remotePort = receivePacket.getPort();
                    log("New request from "+remoteAddr+":"+remotePort);

                    ...
                }
            }
        }
    }
}
```

DateServerUDP (2/2)

```
    ...
    // record current date as an array of bytes
    String currentDate = new Date().toString();
    byte[] data = currentDate.getBytes("ISO-8859-1");

    // send the response
    DatagramPacket responsePacket =
        new DatagramPacket(data,data.length,remoteAddr,remotePort);
    socket.send(responsePacket);
} catch (Exception ex){
    log("Wrong request.");
}
}
} catch (Exception ex){
    ex.printStackTrace();
}
}

private static void log(String msg){
    System.out.println("[DATE SERVER] "+msg);
}
}
```

DateClientUDP (1/2)

```
package modulo2a;

import java.io.*;
import java.net.*;

public class DateClientUDP {
    private static int PORT = 6013;
    private static int BUFFER_SIZE = 256;

    public static void main(String[] args) throws Exception {
        if (args.length<1){
            System.err.println("Arguments: <DateServerAddress>");
            System.exit(1);
        }
        DatagramSocket socket = null;
        try {
            InetAddress serverAddress = InetAddress.getByName(args[0]);

            socket = new DatagramSocket();
            socket.setSoTimeout(5000);

            ...
        }
    }
}
```

DateClientUDP (2/2)

```
...
// request packet
DatagramPacket requestPacket =
    new DatagramPacket(new byte[0],0,serverAddress,PORT);
// packet where to receive the response
byte[] buffer = new byte[BUFFER_SIZE];
DatagramPacket responsePacket =
    new DatagramPacket(buffer,buffer.length);

// send the request
socket.send(requestPacket);
// receive the response
socket.receive(responsePacket);

// extract info
byte[] data = responsePacket.getData();
String currentDate = new String(data,"ISO-8859-1");
System.out.println("Server date: "+currentDate);

} catch (IOException ex){ System.err.println(ex);
} finally {
    if (socket!=null){ socket.close();}
}
}
```

APPLICAZIONI DI RETE SU UDP E TCP/IP

- La maggior parte delle applicazioni distribuite presenti su sistemi operativi utilizza protocolli TCP/IP e UDP per la comunicazione
 - TCP/IP è utilizzato per architetture client/server, in cui ci sia la necessità di avere una certa qualità del servizio nella comunicazione
 - UDP è utilizzato per comunicazioni più agili e dinamiche, in cui non sono ben prefissati ruoli di client e server (esempio: applicazioni **peer-to-peer**)
- La maggior parte dei protocolli di più alto livello (livello sessione / applicazione) sono costruiti su TCP/IP e UDP
 - es: HTTP, FTP (File transfer protocol), SFTP, SMTP...
- Fra le applicazioni note che sfruttano protocolli TCP/IP
 - ssh
 - rsh
 - telnet
 - mailer
 - web browser (HTTP)
 - ...

LIMITI DELL'APPROCCIO VIA SOCKET

- La comunicazione via socket, per quanto molto efficiente e diffusa, è ad un livello di astrazione relativamente basso rispetto a quello utilizzato dai linguaggi di programmazione e architetture per le realizzazioni di software di una certa complessità.
- Allo scopo nel tempo sono state ideate e sviluppate infrastrutture di comunicazione ad un livello di astrazione più alto, simile a quello dei linguaggi di programmazione procedurali e ad oggetti, astruendo da aspetti di comunicazione di basso livello.
- In particolare le informazioni scambiate sono descritte non come insieme di byte (come nel caso delle socket), ma ad un livello di astrazione più alto, direttamente come valori o riferimenti di un determinato tipo (interi, stringhe, array, classi e interfacce nei linguaggi OO)

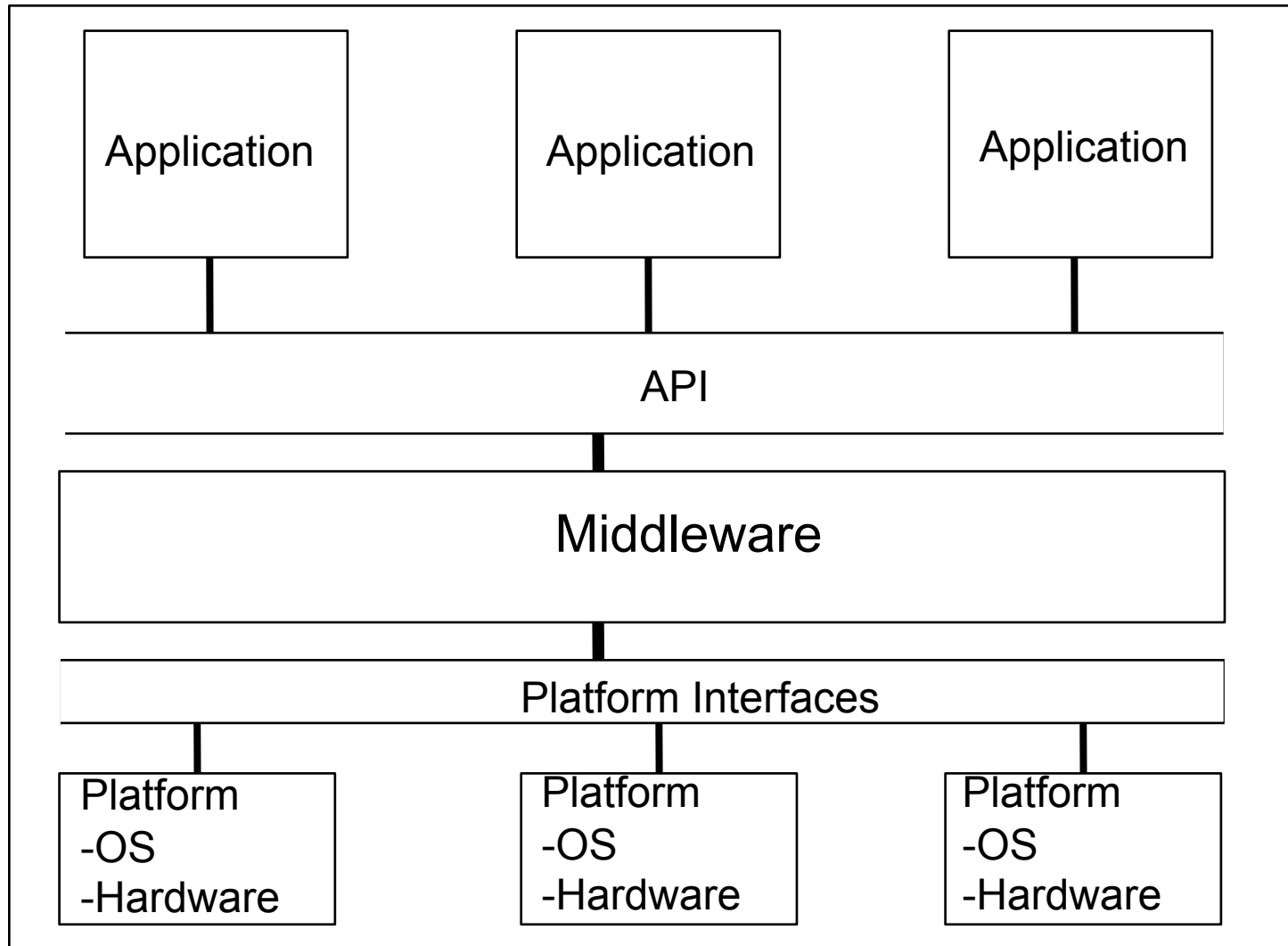
INFRASTRUTTURE DI ALTO LIVELLO

- Le più diffuse realizzano in ambito distribuito meccanismi di computazione e comunicazione di base:
 - le chiamate a procedura remota (**remote procedure call, RPC**) permettono di realizzare una chiamata a procedura in cui il chiamante e chiamato sono processi distinti, il client e il server.
 - invocazione remota di metodo in Java (**remote method invocation, RMI**) permettono di realizzare l'invocazione di metodo fra un oggetto client e oggetto server (che fornisce il metodo) che risiedono su processi diversi.
 - invocazione remota di metodo in linguaggi OO in generale (**CORBA**), che supporta invocazione remota di metodo a prescindere dal linguaggio OO utilizzato
- Questi utilizzabili sono realizzati come **servizi** forniti da **infrastrutture o middleware di comunicazione**, forniti o direttamente a livello di sistema operativo oppure come librerie / servizi costruiti sopra.
 - In generale tutti sono implementati sfruttando meccanismi IPC di base, come le socket.

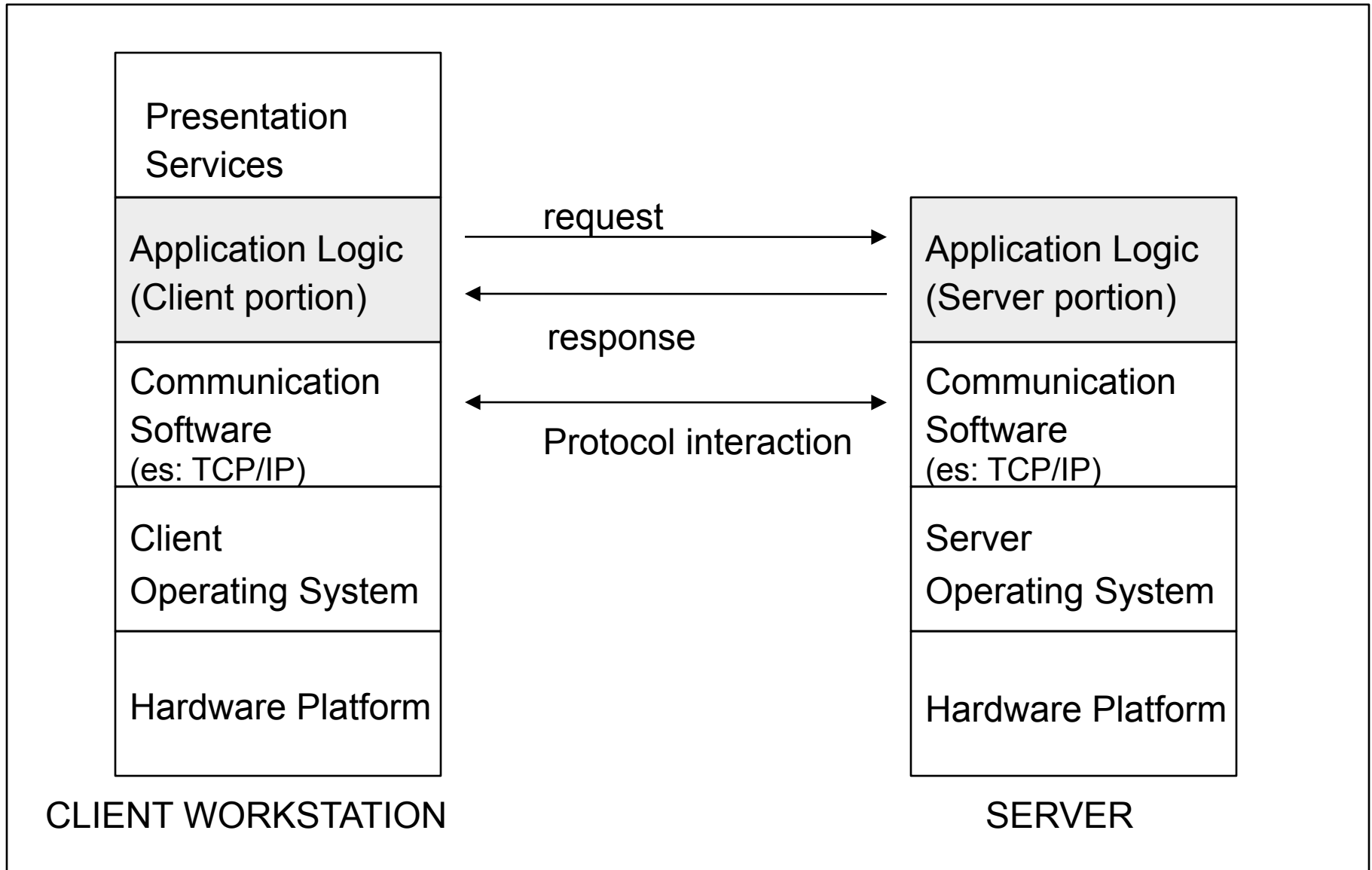
MIDDLEWARE

- La nozione di **middleware** è fondamentale per la realizzazione di sistemi software complessi distribuiti
- In generale per infrastruttura o middleware si intende uno strato intermedio su cui poggiano le applicazioni/sistemi (distribuiti) che eroga un certo insieme di servizi che i componenti dei sistemi sfruttano
- Possono esserci vari tipi di middleware, a seconda dei servizi che erogano: comunicazione, coordinazione, mobilità, etc
- Fra i middleware più noti:
 - RMI e CORBA per interazione e gestione di oggetti distribuiti
 - Web Services per realizzare architetture basate su servizi
- L'argomento viene approfondito nel corso di Sistemi distribuiti

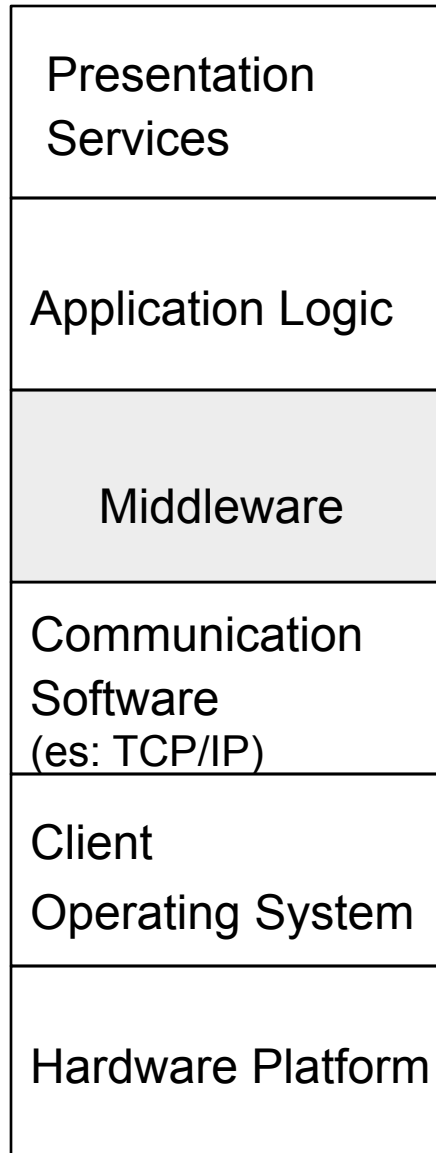
ARCHITETTURA LOGICA DEL MIDDLEWARE



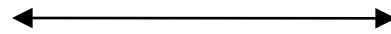
DAL CLIENT-SERVER...



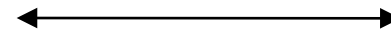
SOLUZIONE CON MIDDLEWARE



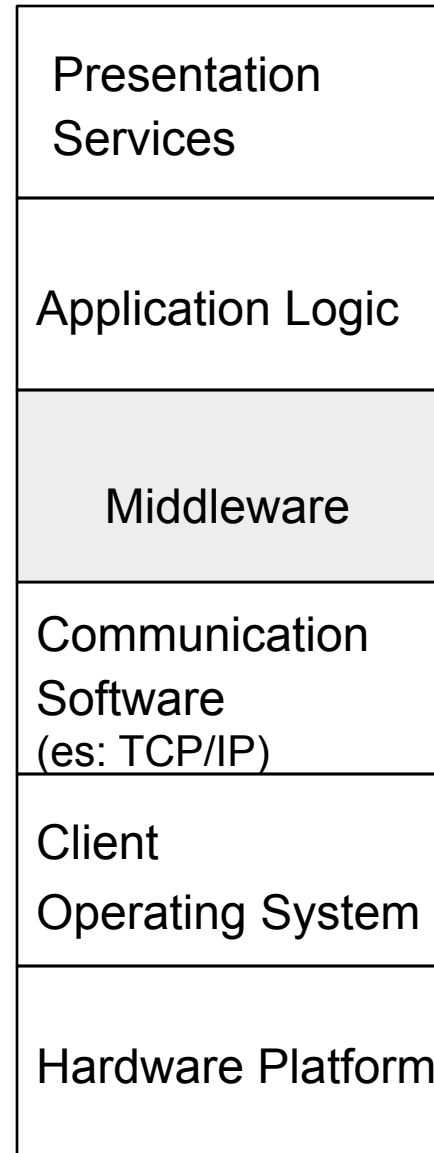
Middleware node



Middleware
interaction

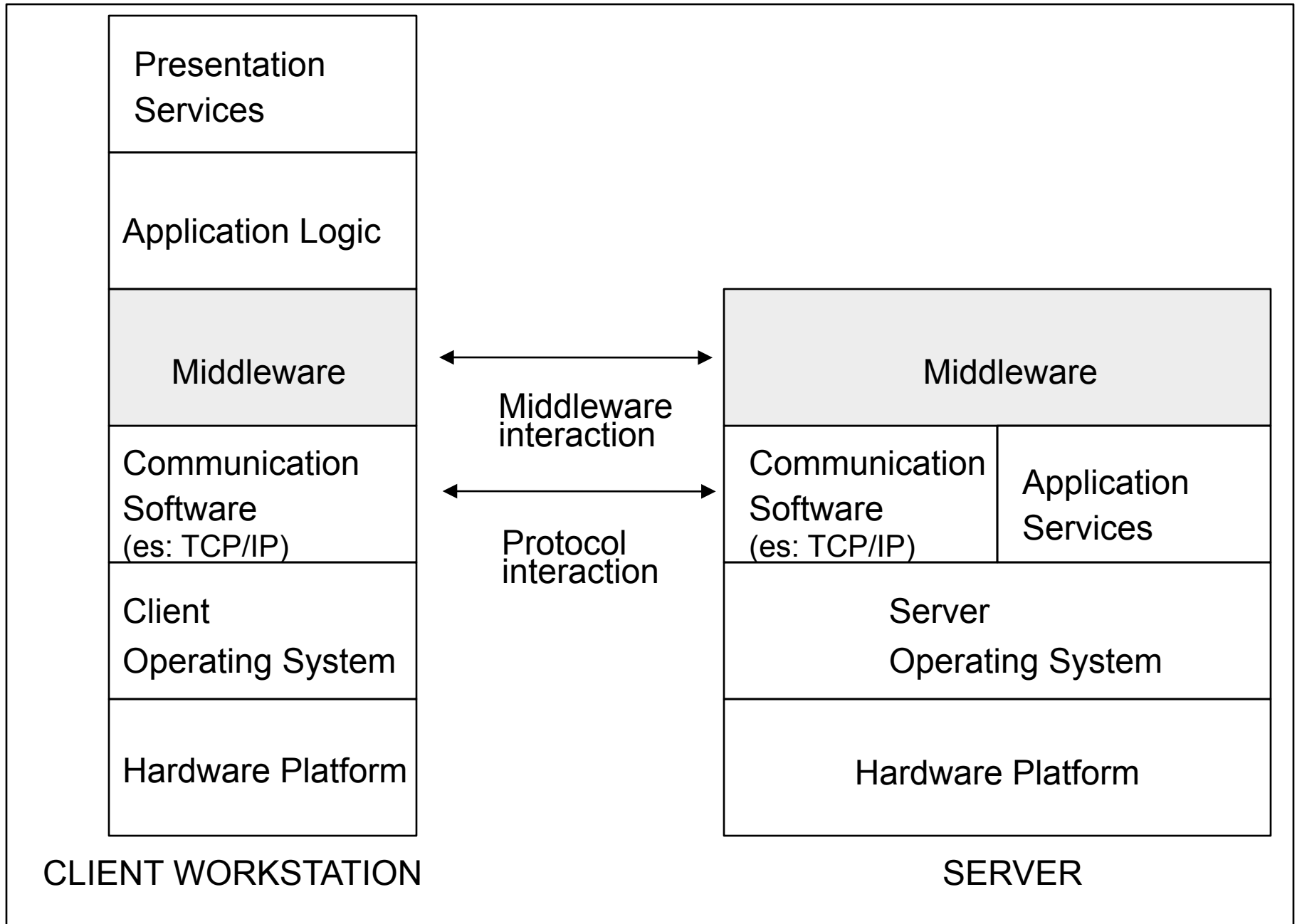


Protocol
interaction



Middleware node

SOLUZIONE CON MIDDLEWARE



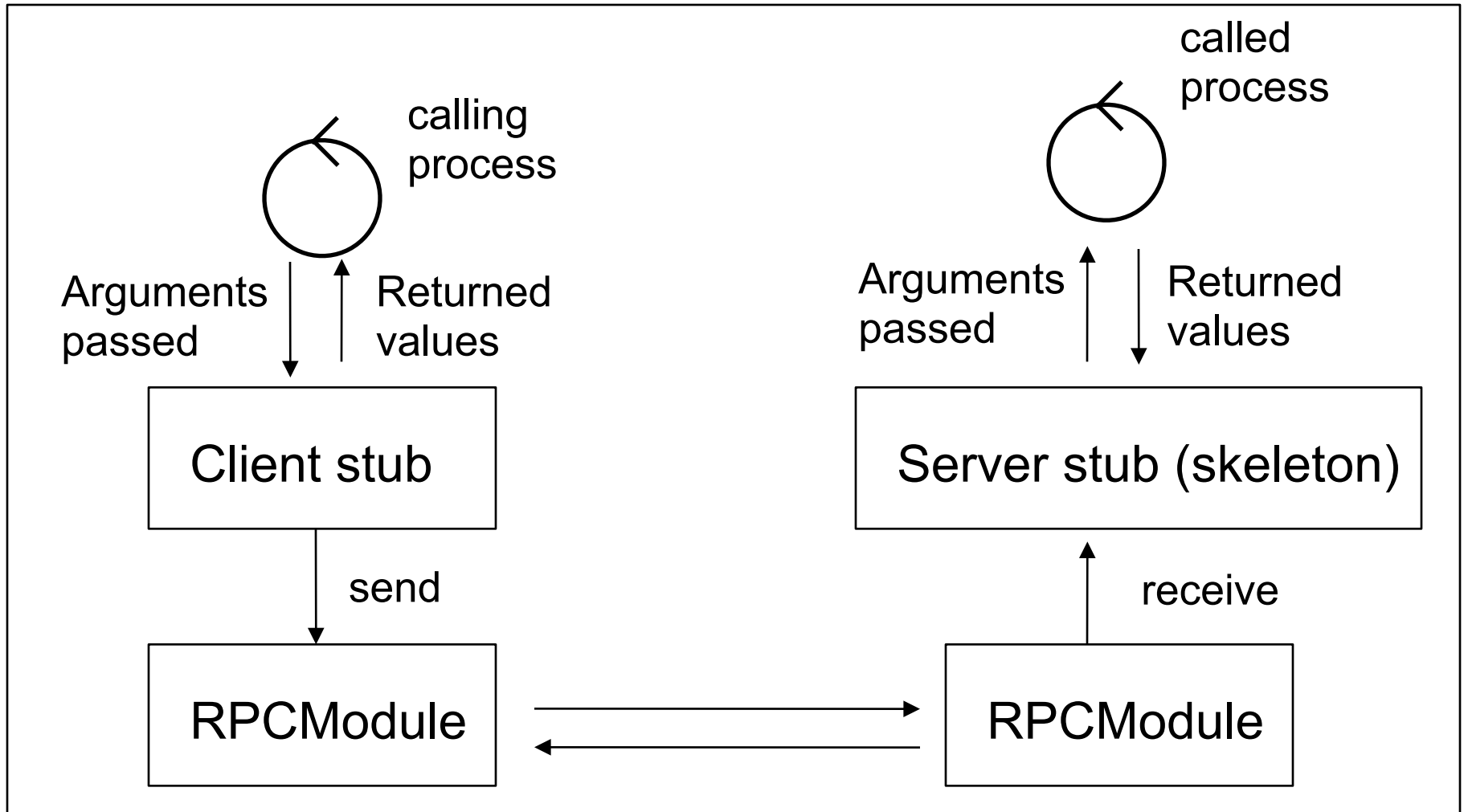
REMOTE PROCEDURE CALL (RPC)

- Permette di realizzare chiamate a procedura (con procedure scritte in linguaggi procedurali, come il C) avendo processi distinti - e tipicamente distribuiti in rete - come chiamante (processo che invoca la procedura) e chiamato (processo che fornisce la procedura).
- Quindi il client invoca una procedura come se fosse locale: l'infrastruttura nasconde i dettagli relativi a tutta la comunicazione e coordinazione via rete fra nodo locale e nodo ove risiede la procedura.

COMPONENTI DI UN SISTEMA RPC

- A livello di implementazione, un sistema RPC si realizza mediante alcune parti infrastrutturali che permettono di astrarre dalla presenza della rete:
 - demone nel nodo server in attesa di ricevere richieste di esecuzione di procedure
 - **stub** - entità lato client che agisce da **proxy** per il servizio lato server: l'invocazione di procedura del client è raccolta (in modo trasparente) dallo stub, che provvede a contattare via rete il server ove risiede la procedura, e quindi inoltrare la richiesta con le informazioni necessarie (in particolare i parametri della procedura). La trasformazione ad opera dello stub dei valori dei parametri in pacchetti di byte prende il nome di **marshaling**.
 - **skeleton** (o server-side stub) - entità lato server che riceve la richiesta del client, esegue la chiamata a procedura e quindi invia i risultati al client (allo stub). In particolare, uno skeleton riceve una richiesta inoltrata dallo stub, ripristina i parametri a partire dalle informazioni passate via rete (**unmarshaling**) e quindi esegue la procedura.

MECCANISMO RPC

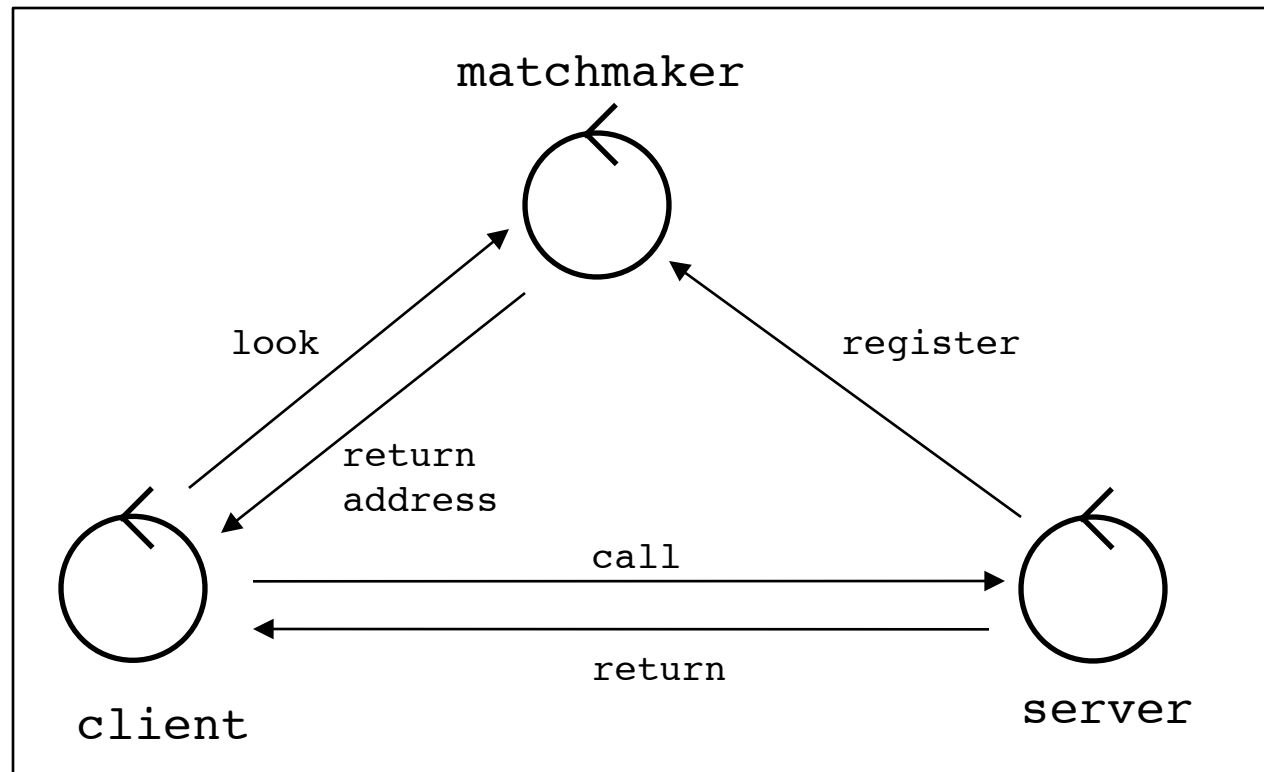


XDR

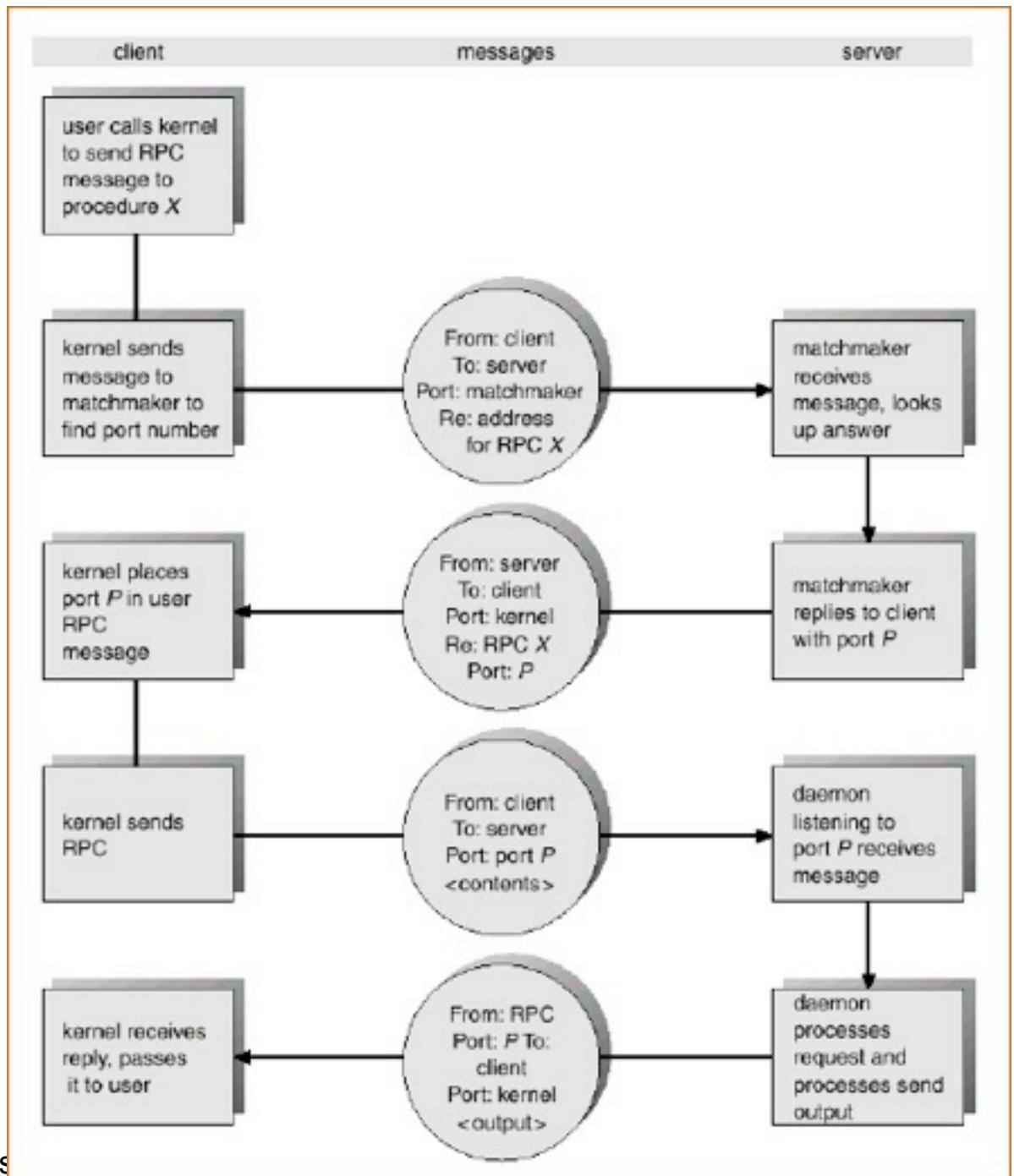
- I sistemi RPC definiscono in genere una rappresentazione dei dati (**XDR, eXternal Data Representation**) scambiata indipendente dallo specifico hardware, sistema operativo o linguaggio - in modo da permettere RPC fra sistemi eterogenei.
- Alcuni sistemi operativi forniscono le RPC come meccanismi forniti direttamente a livello di kernel.
 - Ad esempio Windows XP.
- Spesso sono realizzati sui meccanismi IPC di base forniti dal kernel, come l'invio di messaggi.

RPC MATCHMAKER

- Un altro componente importante in una infrastruttura RPC è dato dal **matchmaker**, il componente che fornisce il servizio di 'pagine-gialle', ovvero mantiene l'associazione fra nome di una procedura e suo indirizzo.
- Questo permette di realizzare un **binding** dinamico fra client e server.

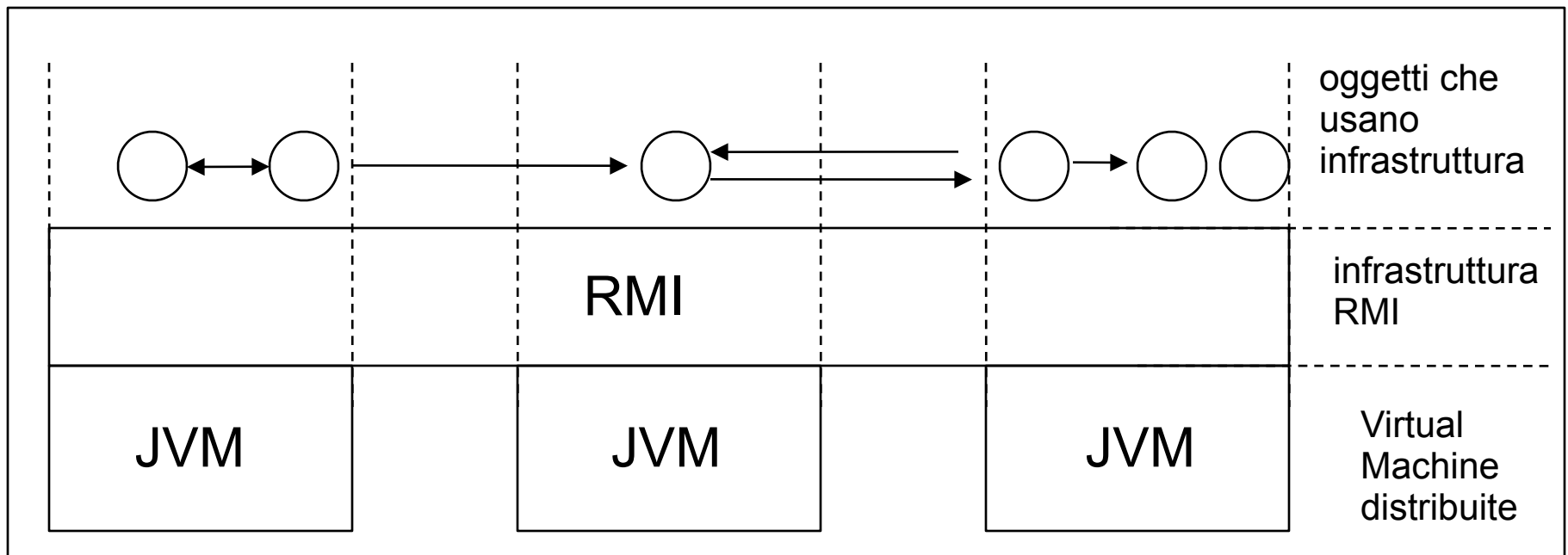


RPC: DINAMICA

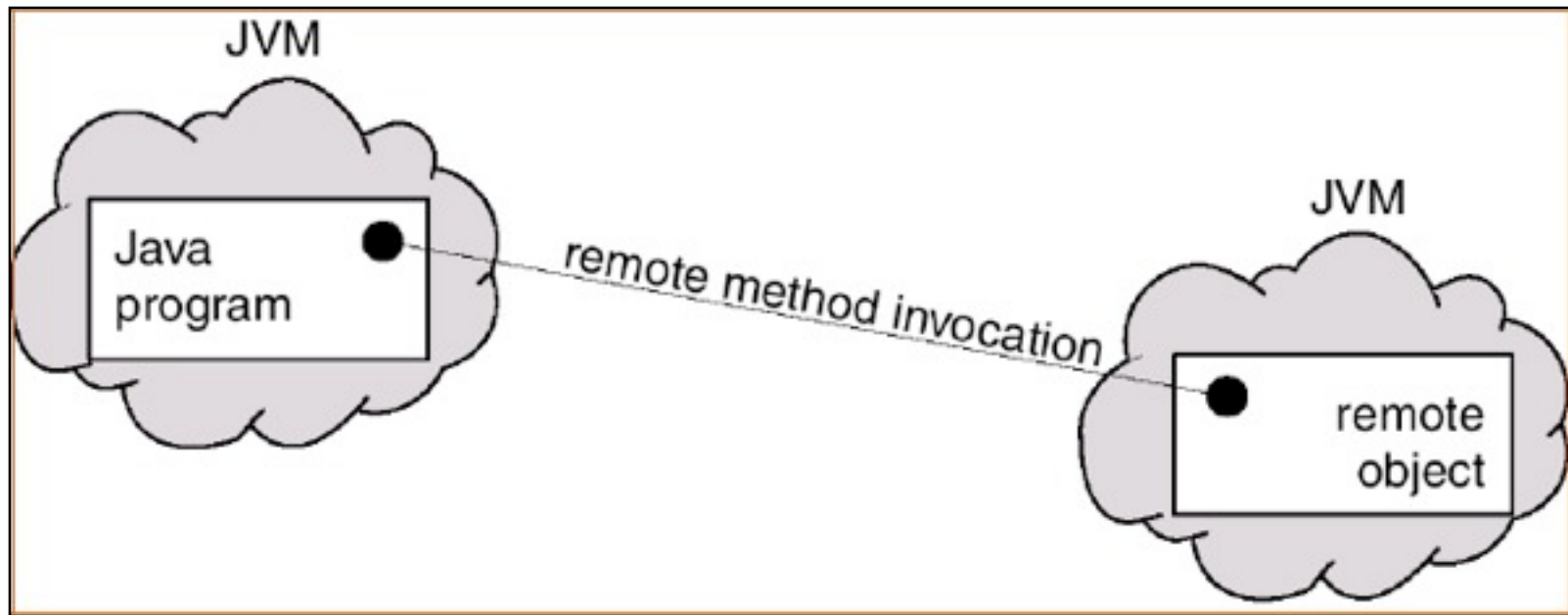


REMOTE METHOD INVOCATION (RMI)

- Infrastruttura di comunicazione in Java che permette di realizzare invocazione di metodi fra oggetti che appartengono a JVM distinte, generalmente su computer distinti
 - Obiettivo: rendere al programmatore il più possibile trasparente l'ambiente distribuito, mantenendo lo stile object oriented
 - “RPC” in ambito Object-oriented



INVOCAZIONE REMOTA DI METODO



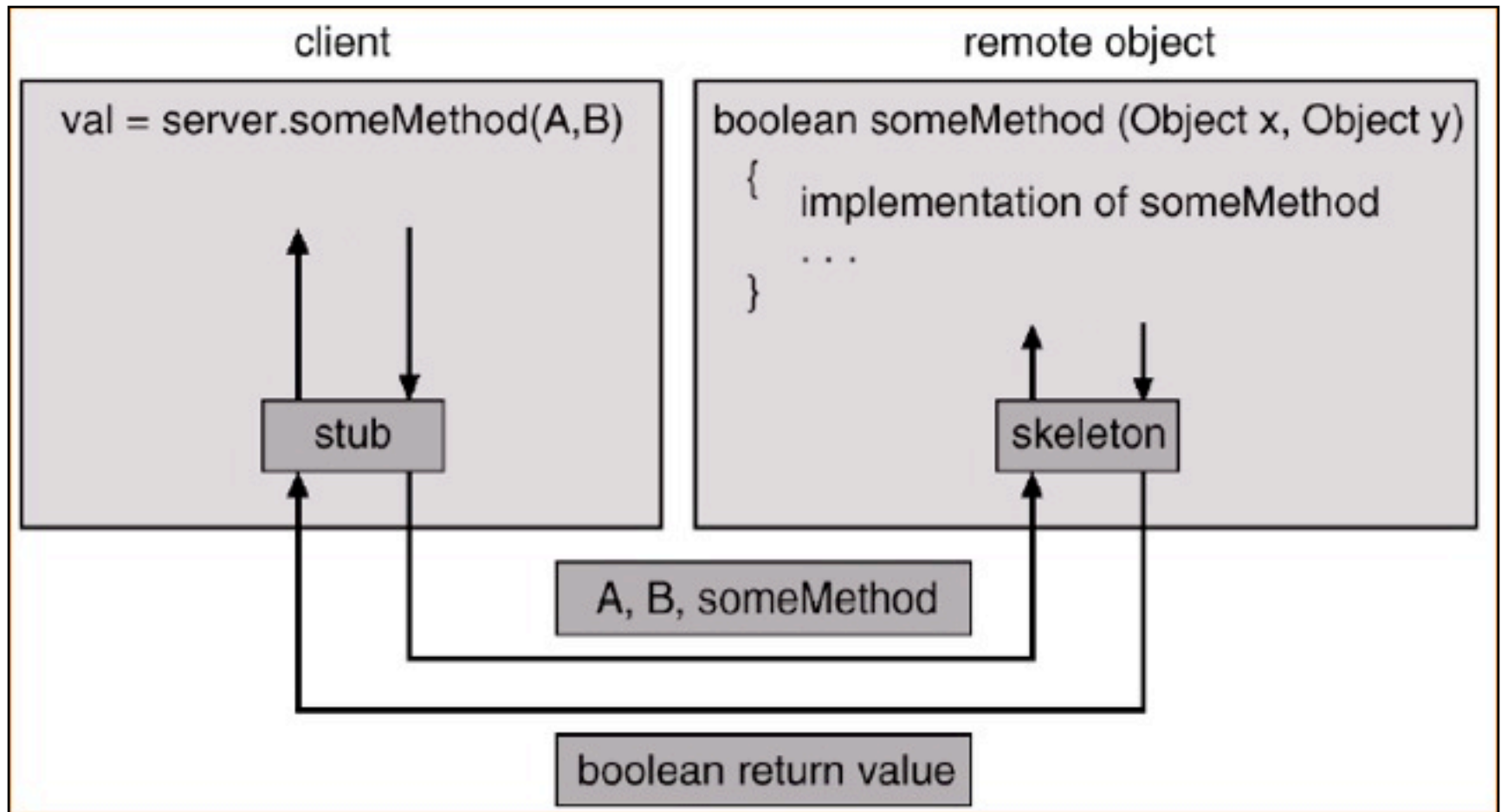
COMPONENTI DI UN SISTEMA RMI (1/2)

- Analogamente ad un servizio RPC a livello architetturale l'infrastruttura RMI è costituita a runtime da un demone residente su ogni nodo, da stub e skeleton
- lo **stub** è il proxy per l'oggetto remoto, residente su lato client
 - Quando un client invoca un metodo sull'oggetto remoto, l'invocazione è eseguita in realtà sullo stub, che provvede a creare un oggetto (**parcel**) con le informazioni relative al metodo invocato, contenente il nome del metodo e parametri impacchettati (marshaled).
 - Il parcel viene quindi inviato al server sfruttando protocolli di rete di più basso livello (tipicamente TCP/IP), in modo che sia ricevuto dallo skeleton
- lo **skeleton** è la parte su lato server, responsabile di eseguire concretamente il metodo sull'oggetto
 - riceve il parcel, spacchetta i parametri e invoca il metodo sull'oggetto. Ottenuti gli eventuali risultati, lo skeleton quindi impacchetta le informazioni e le rispedisce allo stub

COMPONENTI DI UN SISTEMA RMI (2/2)

- Demone RMI
 - Gestisce le richieste che arrivano da remoto, mediante un insieme di thread che provvedono a ricevere richieste, e a consegnarle
 - Fornisce servizi per la registrazione di oggetti che si vogliono rendere accessibili da remoto
 - Si installa lanciando il programma **rmiregistry**

DINAMICA DI UNA INTERAZIONE RMI



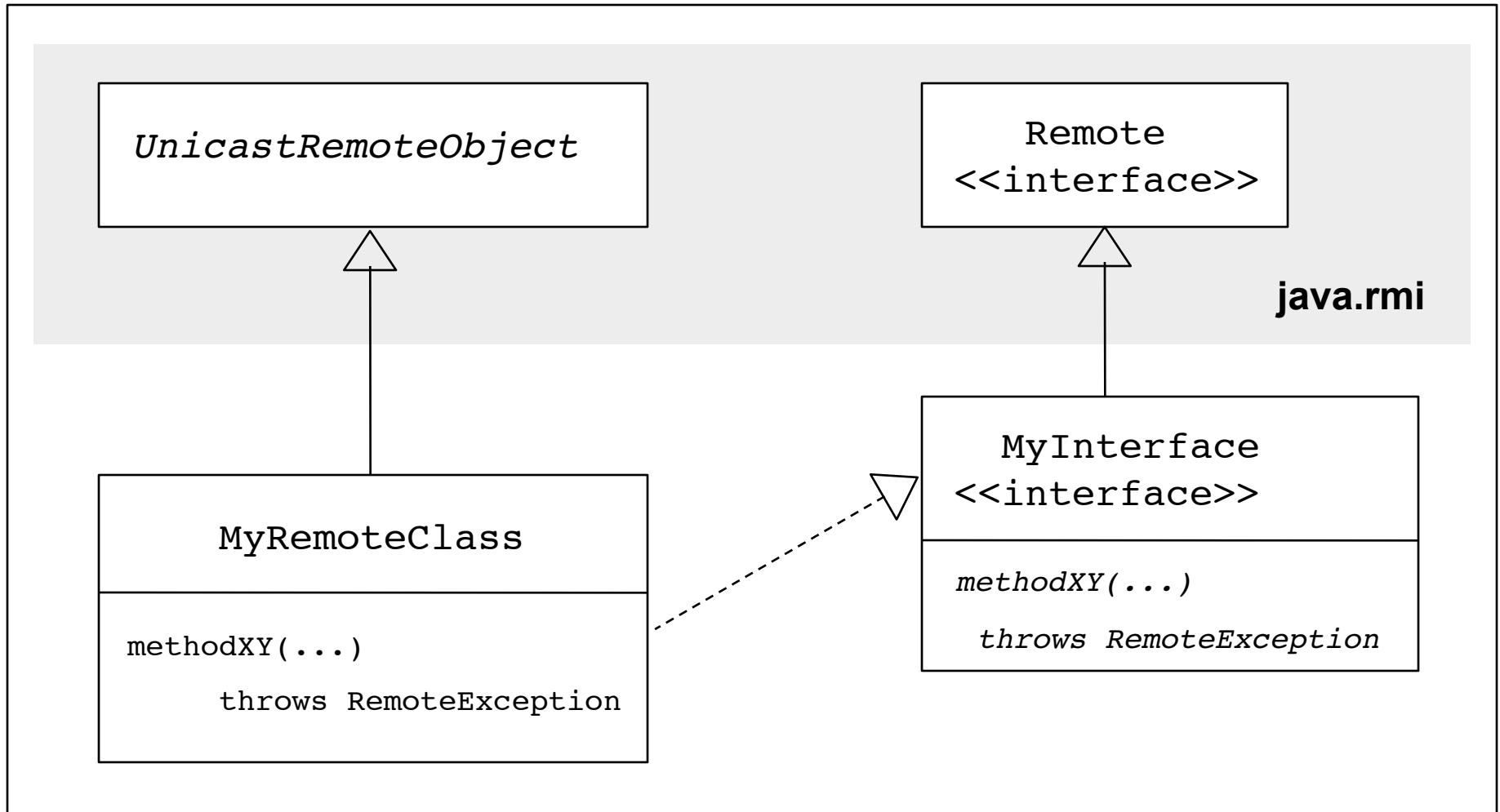
PASSAGGIO DI PARAMETRI

- Il livello di astrazione fornito da RMI è tale per cui stub e skeleton non sono utilizzati direttamente: il programmatore scrive programmi che invocano metodi remoti esattamente (o quasi) come se invocasse metodi locali
- Ci sono tuttavia differenze sostanziali nel modo in cui si realizza il passaggio di parametri
 - Se i parametri passati, impacchettati (marshalled), sono oggetti *locali*, non remoti, allora viene passati una copia di tali oggetti, sfruttando la **serializzazione**
 - esecuzione di metodi su tali oggetti da parte del server NON ha effetto sugli oggetti stessi lato client
 - Se invece i parametri passati sono invece anch'essi oggetti remoti, allora viene passato il loro riferimento
 - esecuzione di metodi su tali oggetti da parte del server ha effetto sugli oggetti stessi lato client
 - Idem vale per gli argomenti di ritorno

INTERFACCIA E CLASSE DI UN OGGETTO REMOTO

- In realtà l'approccio RMI è 'quasi' trasparente: affinché oggetti siano raggiungibili anche nel distribuito è necessario che siano progettati già come tali, ovvero implementino / estendano interfacce / classi di base fornite nel package `java.rmi`.
- In particolare, affinché un oggetto sia raggiungibile anche da remoto deve implementare una interfaccia che specifichi quali metodi devono poter essere invocati da remoto
 - L'interfaccia deve estendere l'interfaccia `java.rmi.Remote`
 - Ogni metodo 'remoto' deve poter lanciare l'eccezione `java.rmi.Exception`, che rappresenta la presenza di problemi nella comunicazione di I/O distribuita
- La classe di un oggetto remoto deve implementare l'interfaccia remota ed estendere una classe fornita nella libreria di classi RMI, `java.rmi.UnicastRemoteObject`

DIAGRAMMA UML



ESEMPIO: SERVIZIO DATE IN RMI

- Con riferimento al servizio Date visto in precedenza con le socket, vediamo la realizzazione in RMI
- Anzitutto definiamo l'interfaccia dell'oggetto remoto:

```
package modulo2a;

import java.rmi.*;
import java.util.Date;

public interface IRemoteDate extends Remote {

    Date getDate() throws RemoteException;

}
```

IMPLEMENTAZIONE DELLA CLASSE DELL'OGGETTO REMOTO RemoteDateImpl

```
package modulo2a;

import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

public class RemoteDateImpl extends UnicastRemoteObject
    implements IRemoteDate {

    public RemoteDateImpl() throws RemoteException{
    }

    public Date getDate() throws RemoteException {
        return new Date();
    }
}
```

REGISTRAZIONE DI UN OGGETTO REMOTO

- Affinché un oggetto sia raggiungibile da remoto deve essere **registrato** a runtime sul server, rendendolo 'visibile' a tutti gli oggetti che fruiscono dell'infrastruttura RMI,
- L'oggetto si registra specificando un nome *Name*
 - Il nome completo che identifica l'oggetto a livello distribuito è una URI del tipo: `rmi:// <IPAddress> / <Name>`
 - Esempio: `rmi:// alicelab.ing2.unibo.it / DateService`
`rmi:// 137.204.107.69 / pippoTheCounter`
- La registrazione avviene mediante i servizi statici forniti dalla classe `java.rmi.Naming`, che rappresenta l'API per interagire con il match-maker dell'infrastruttura RMI
 - metodi `bind` e `rebind`

ESEMPIO: DateServiceInstaller

```
package modulo2a;

import java.rmi.*;

public class DateServiceInstaller {
    public static void main(String[] args) {
        try {
            IRemoteDate dateService = new RemoteDateImpl();
            Naming.rebind("DateService",dateService);
            System.out.println("Date service installed.");
        } catch (Exception ex){
            ex.printStackTrace();
        }
    }
}
```

LOOKUP DI UN OGGETTO

- Lato client, si sfruttano i servizi statici della classe Naming (**lookup** in particolare) per fare il binding di un oggetto data la URI che lo identifica unicamente a livello di rete, secondo il formato visto in precedenza
 - Esempio: `rmi: // alicelab.ing2.unibo.it / DataService`

ESEMPIO: DateServiceClient

```
package modulo2a;

import java.rmi.*;
import java.util.*;

public class DateServiceClient {
    public static void main(String[] args) {
        try {
            if (args.length!=1){
                System.err.println("Argument: <Date Service URI> -
                    e.g. rmi://127.0.0.1/DateService");
                System.exit(1);
            }
            IRemoteDate dateService =
                (IRemoteDate)Naming.lookup(args[0]);
            Date date = dateService.getDate();
            System.out.println("The date is "+date);
        } catch (Exception ex){
            System.err.println(ex);
        }
    }
}
```

ESECUZIONE DEL PROGRAMMA: PASSI (1/2)

- Supponiamo di installare il DateService in locale. I passi fondamentali per l'esecuzione sono:
 - compilazione di tutti i sorgenti
 - generazione dei file stub e skeleton mediante il tool **rmic**, che richiede il nome della classe che implementa l'oggetto remoto (RemoteDateImpl nel nostro caso)

```
rmic modulo2a.RemoteDateImpl
```

L'esecuzione con successo del comando porta alla creazione di due classi compilate, RemoteDateImpl_Skel.class e RemoteDateImpl_Stub.class

- E' importante che questi due file siano presenti sia lato client, sia lato server: in realtà ciò può essere evitato con il caricamento dinamico della classe utilizzando RMI

ESECUZIONE DEL PROGRAMMA: PASSI (2/2)

- (.passi..)
 - Usando una shell separata, installare il servizio (l'oggetto remoto) eseguendo DateServiceInstaller:

```
java modulo2a.DateServiceInstaller
```

- Eseguire il client, specificando il nome del servizio come argomento

```
java modulo2a.DateServiceClient  
rmi://127.0.0.1/DateService
```

- Risultato dell'esecuzione:

```
$ java modulo2a.DateServiceClient rmi://localhost/DateService  
The date is Mon Feb 14 04:22:16 CET 2005
```

CORBA

- CORBA (Common Object Request Broker Architecture) nasce come middleware (infrastruttura) per l'interazione fra oggetti distribuiti in rete, scritti in linguaggi diversi
 - Java RMI è stato studiato per invocazione remota di oggetti Java, CORBA invece è stato ideato per supportare oggetti scritti in qualsiasi linguaggio OO
- Tra gli obiettivi
 - supporto per eterogeneità
 - trasparenza
- Cuore del sistema è l'ORB (**Object Request Broker**), ovvero il componente presente in ogni nodo CORBA che fornisce servizi per il corretto funzionamento del middleware
 - servizi di naming, registering, lookup
 - raccoglie le richieste di invocazione di metodo e permette loro di arrivare sull'oggetto target

WEB SERVICES

- Stack di protocolli per realizzare sistemi client-server e *peer-to-peer* eterogenei e interoperabili
 - HTTP come principale protocollo di trasporto
 - SOAP come protocollo per specificare i messaggi da scambiare
 - XML come linguaggio di riferimento per rappresentare le informazioni
- Sviluppatisi nell'ultimo decennio, sta divenendo l'approccio di riferimento oggi per realizzare sistemi distribuiti