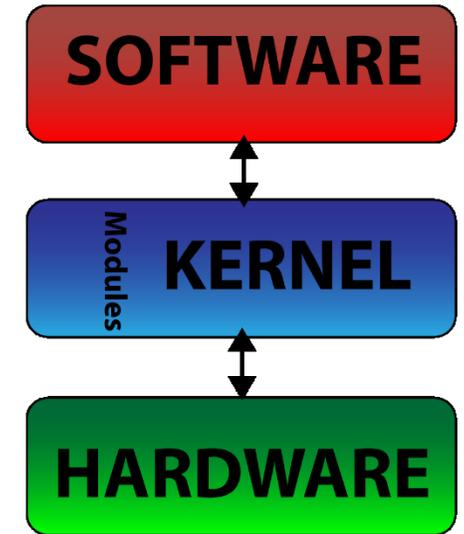
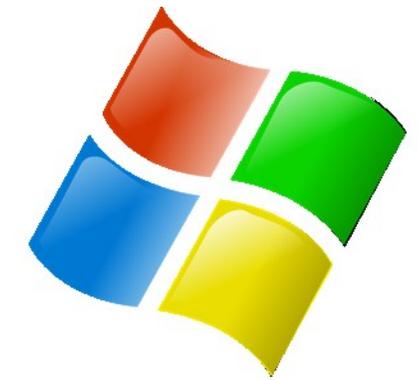




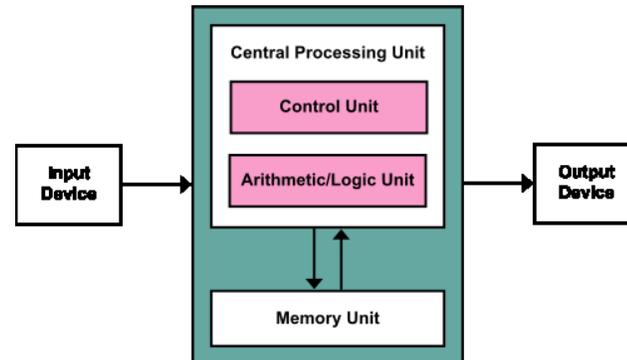
**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

*Corso di Sistemi Operativi  
A.A. 2019/20*

# Processi



Docente:  
**Domenico Daniele  
Bloisi**



Ottobre 2019

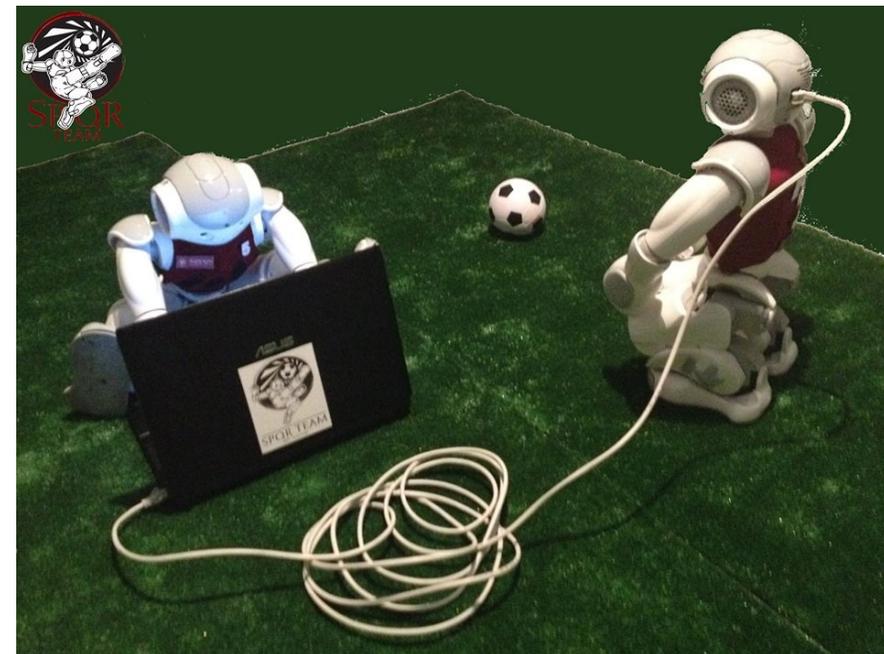
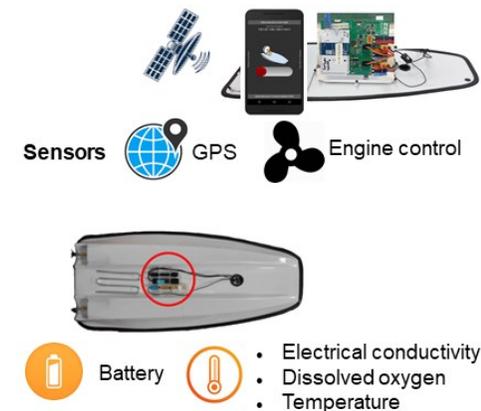
# Domenico Daniele Bloisi

- Ricercatore RTD B  
Dipartimento di Matematica, Informatica  
ed Economia  
Università degli studi della Basilicata

<http://web.unibas.it/bloisi>

- SPQR Robot Soccer Team  
Dipartimento di Informatica, Automatica  
e Gestionale Università degli studi di  
Roma “La Sapienza”

<http://spqr.diag.uniroma1.it>



# Ricevimento

---

- In aula, subito dopo le lezioni
- Martedì dalle 11:00 alle 13:00 presso:  
Campus di Macchia Romana  
**Edificio 3D** (Dipartimento di Matematica,  
Informatica ed Economia)  
**Il piano, stanza 15**

Email: [domenico.bloisi@unibas.it](mailto:domenico.bloisi@unibas.it)



# Programma – Sistemi Operativi

---

- Introduzione ai sistemi operativi
- **Gestione dei processi**
- Sincronizzazione dei processi
- Gestione della memoria centrale
- Gestione della memoria di massa
- File system
- Sicurezza e protezione

# Il concetto di processo

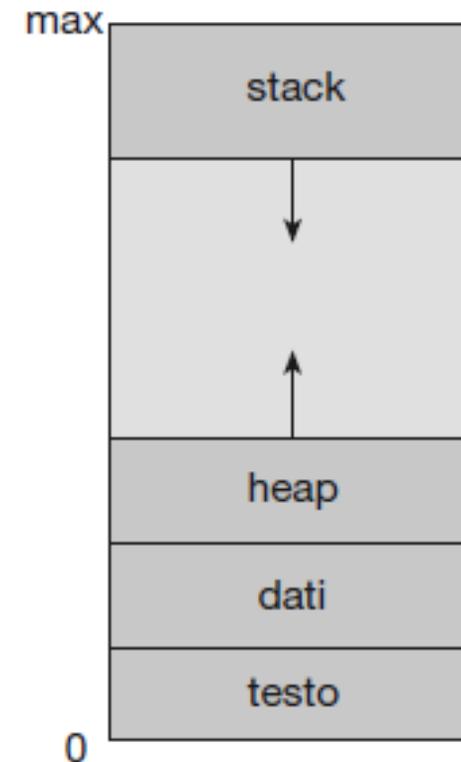
Un **processo** è un programma in esecuzione. La struttura di un processo in memoria è generalmente suddivisa in più sezioni.

**Sezione di testo** contenente il codice eseguibile

**Sezione dati** contenente le variabili globali

**Heap** memoria allocata dinamicamente durante l'esecuzione del programma

**Stack** memoria temporaneamente utilizzata durante le chiamate di funzioni



**Figura 3.1** Struttura di un processo in memoria.

# Stato del processo

**Nuovo.** Si crea il processo

**Esecuzione** (*running*). Le sue istruzioni vengono eseguite.

**Attesa** (*waiting*). Il processo attende che si verifichi qualche evento.

**Pronto** (*ready*). Il processo attende di essere assegnato a un'unità d'elaborazione.

**Terminato.** Il processo ha terminato l'esecuzione.

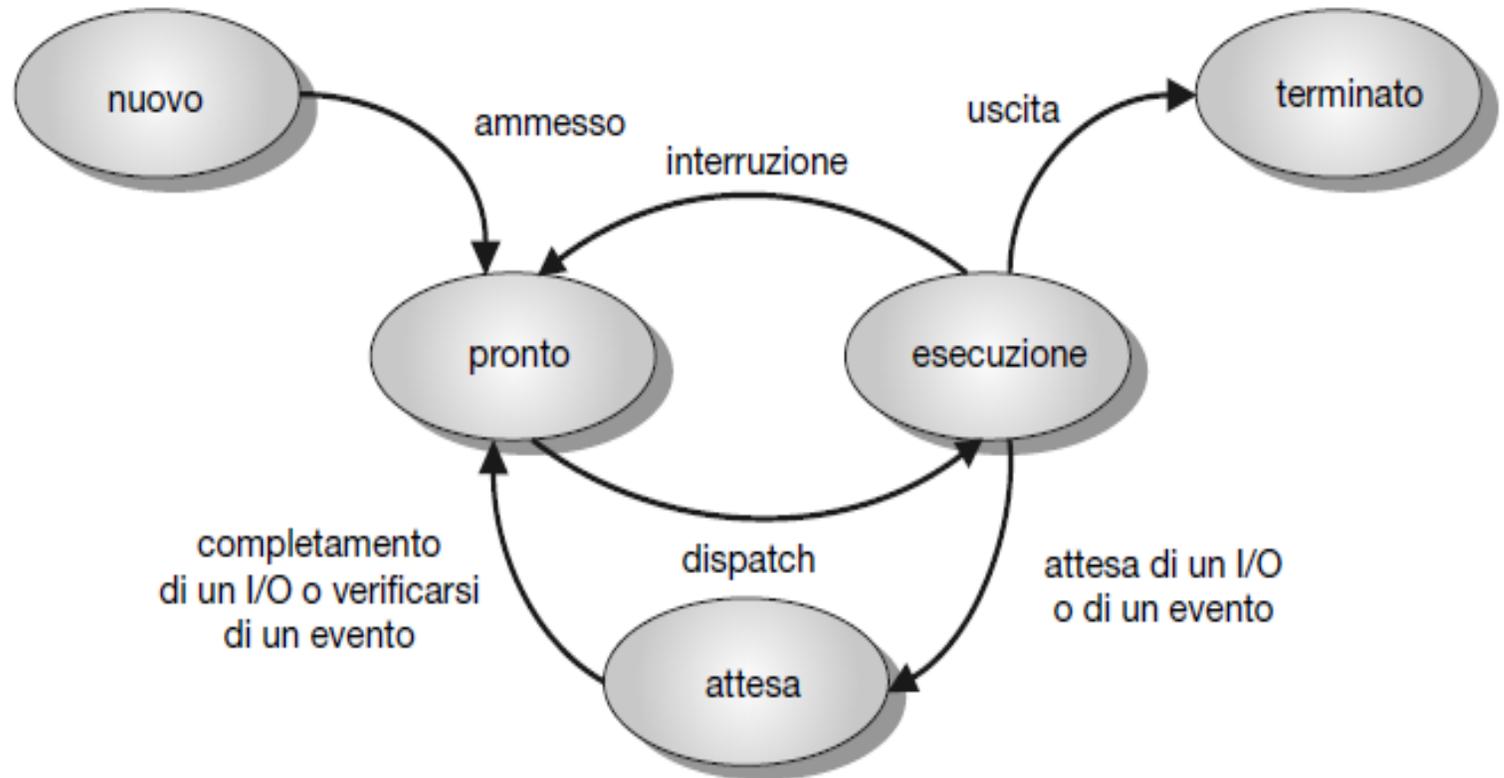


Figura 3.2 Diagramma di transizione degli stati di un processo.

# Process control block (PCB)

---

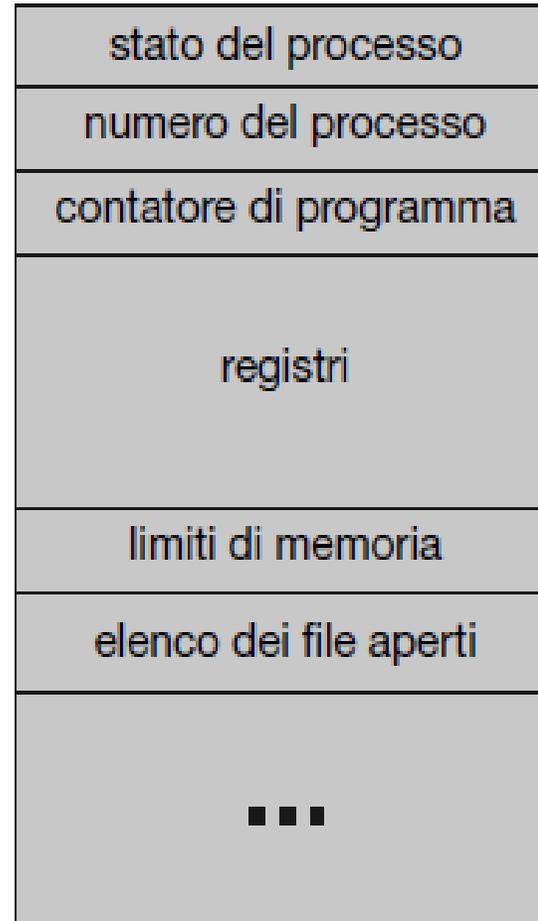
Ogni processo è rappresentato nel sistema operativo da un **blocco di controllo** (*process control block, PCB, o task control block, TCB*) contenente molte informazioni connesse a un processo specifico.

# Process control block (PCB)

**Stato del processo** → nuovo, pronto, esecuzione, attesa, arresto

**Contatore di programma** che contiene l'indirizzo della successiva istruzione da eseguire per tale processo.

**Registri della CPU** → accumulatori, registri indice, puntatori alla cima dello stack (*stack pointer*), registri d'uso generale e registri contenenti i codici di condizione (*condition codes*)



**Informazioni sullo scheduling di CPU**

**Informazioni sulla gestione della memoria**

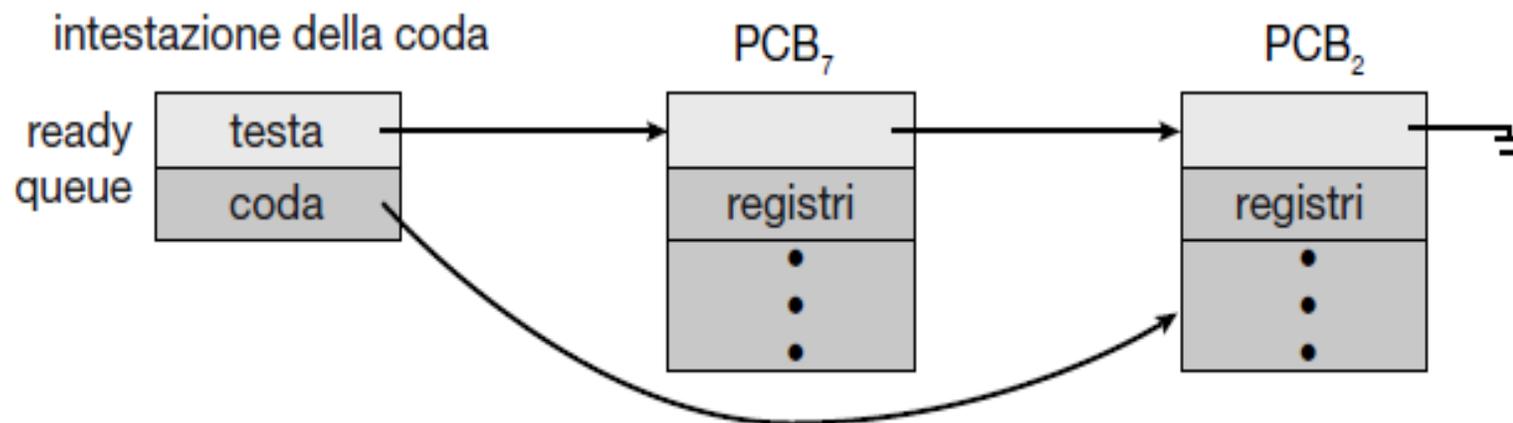
**Informazioni di accounting e sullo stato dell'I/O**

**Figura 3.3** Blocco di controllo di un processo (PCB).

# Scheduling dei processi

Lo **scheduler dei processi** seleziona un processo da eseguire dall'insieme di quelli disponibili.

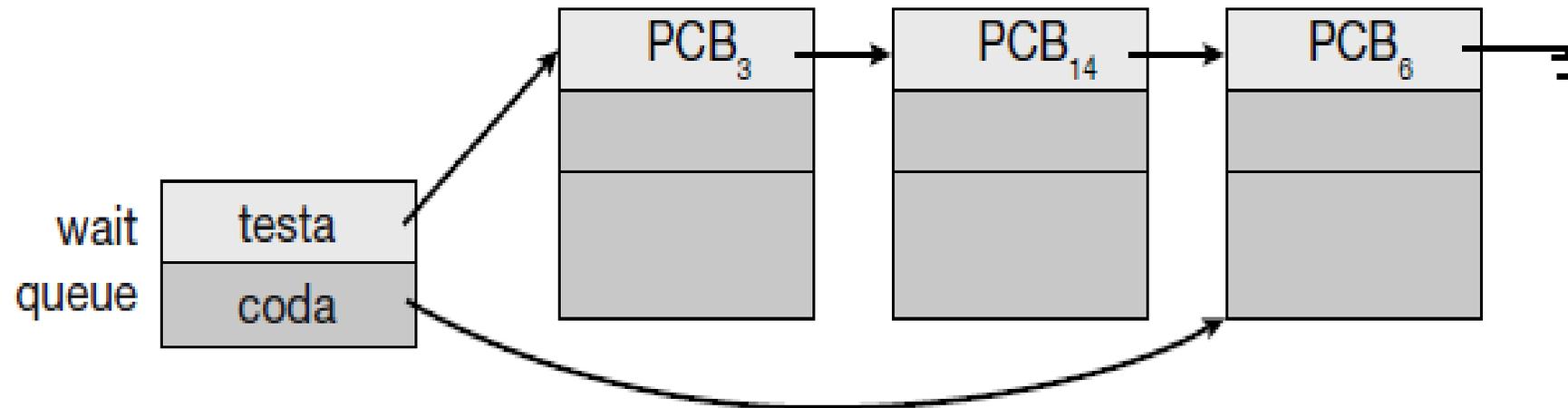
- **Processo I/O bound** → impiega la maggior parte del proprio tempo nell'esecuzione di operazioni di I/O
- **Processo CPU bound** → impiega la maggior parte del proprio tempo nelle elaborazioni
- Ogni processo è inserito in una **coda di processi pronti** e in attesa d'essere eseguiti, detta **coda dei processi pronti** (*ready queue*).



# Scheduling dei processi

---

- I processi in attesa di un determinato evento vengono collocati in una **codice di attesa**, o *wait queue*



# Diagramma di accodamento

Una comune rappresentazione dello scheduling dei processi è data da un **diagramma di accodamento**

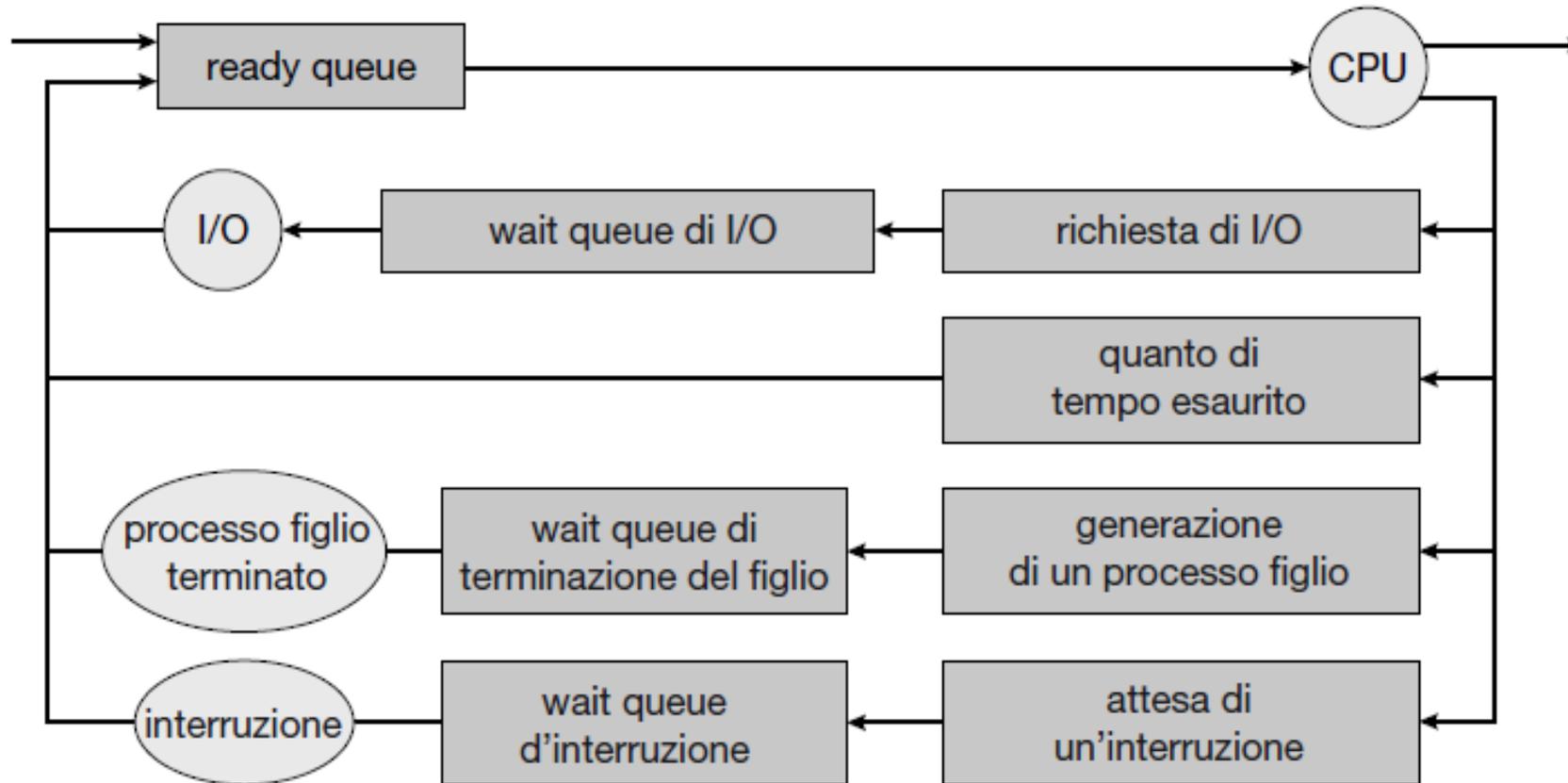


Figura 3.5 Diagramma di accodamento per lo scheduling dei processi.

# Context switch

- In presenza di una interruzione (evento comune), il sistema deve salvare il **contesto del processo corrente**, per poterlo poi ripristinare quando il processo stesso potrà ritornare in esecuzione.
- Si esegue un **salvataggio dello stato** e, in seguito, un corrispondente **ripristino dello stato**.
- Tale procedura è detta **cambio di contesto**

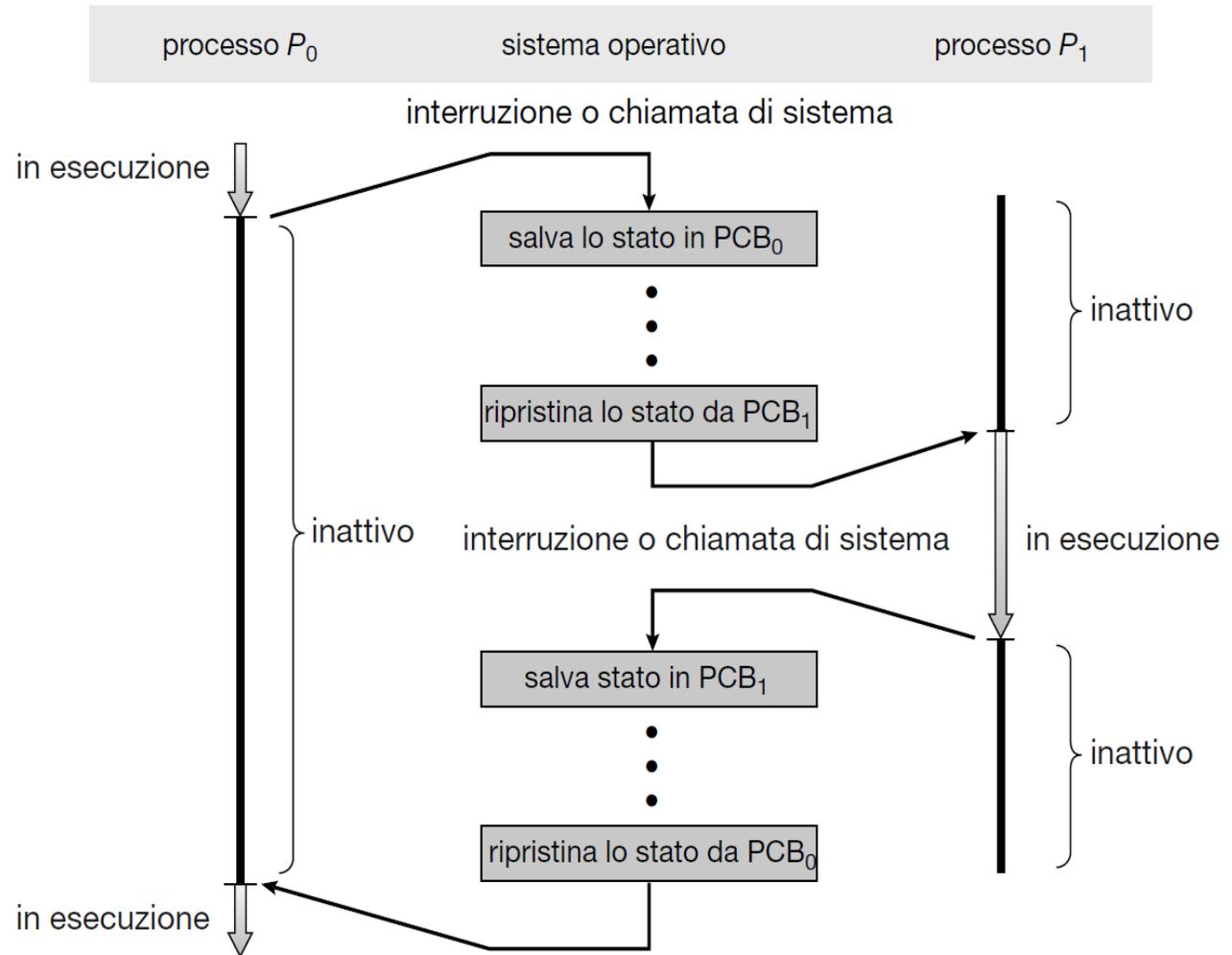
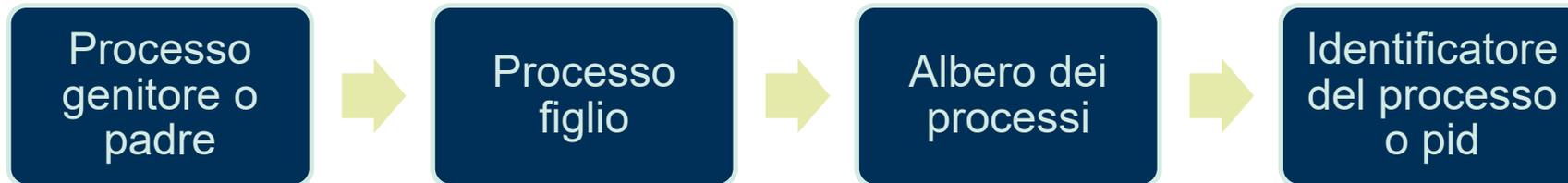


Figura 3.6 Diagramma di cambio di contesto.

# Operazioni sui processi

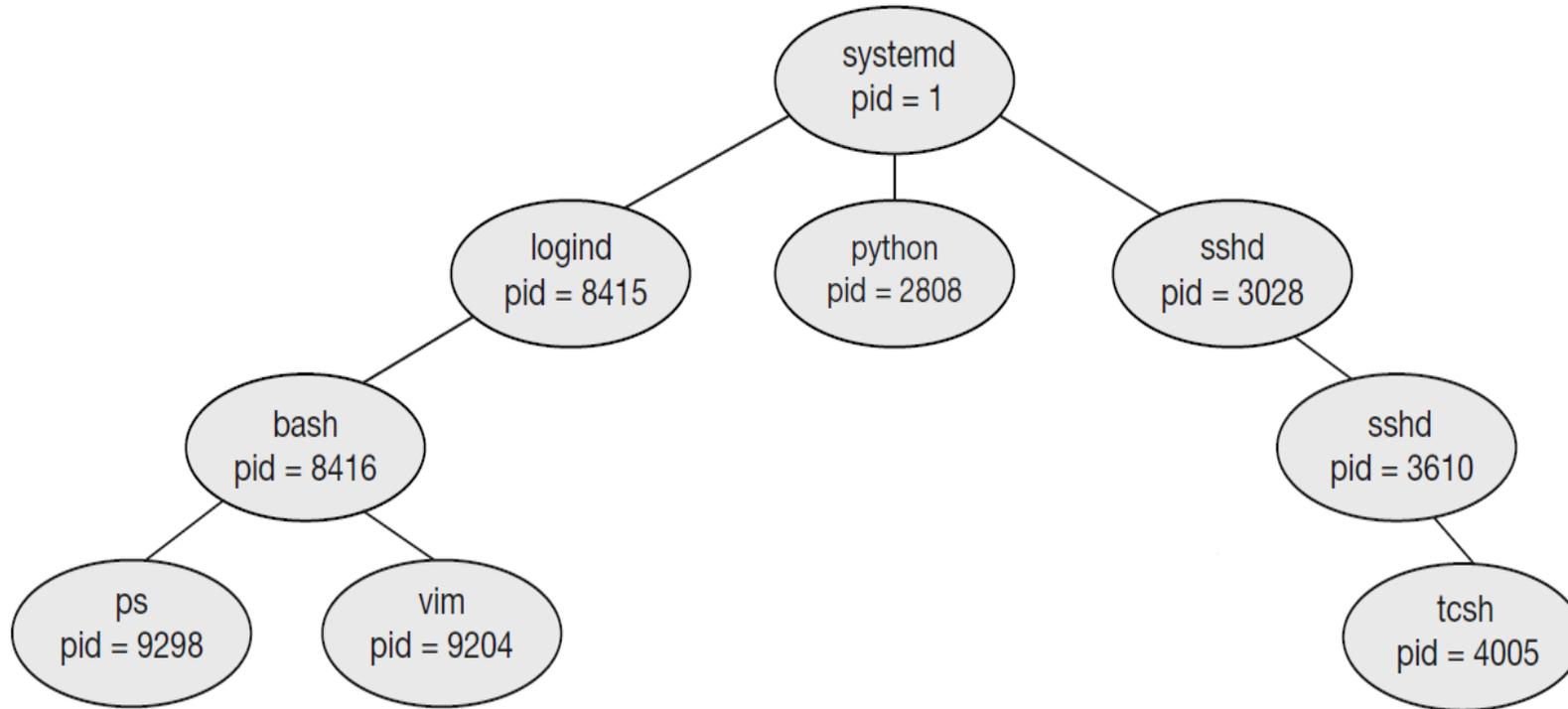
---

## Creazione dei processi



# Albero dei processi

---



**Figura 3.7** Esempio di albero dei processi di un tipico sistema Linux.

# fork() - UNIX

---

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

int main()
{
    pid_t pid;

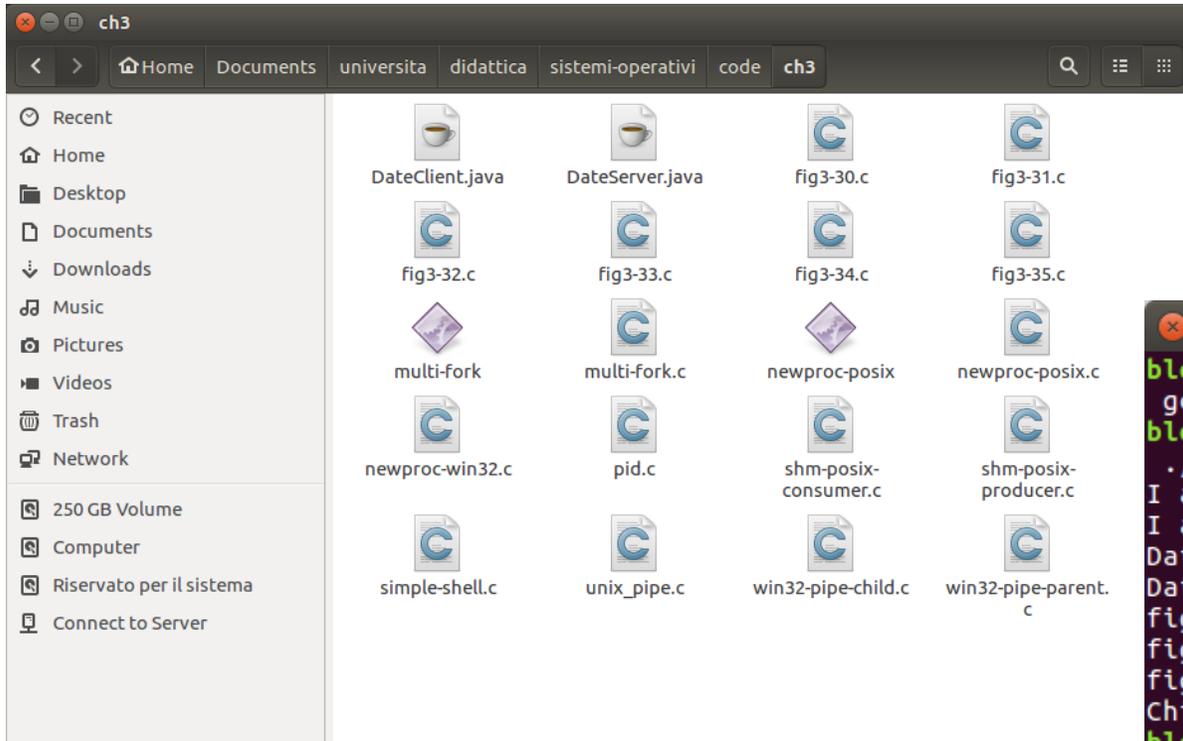
    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed\n");
        return 1;
    }
    else if (pid == 0) { /* child process */
        printf("I am the child %d\n",pid);
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        printf("I am the parent %d\n",pid);
    }

    printf("Child Complete\n");
}

return 0;
}
```

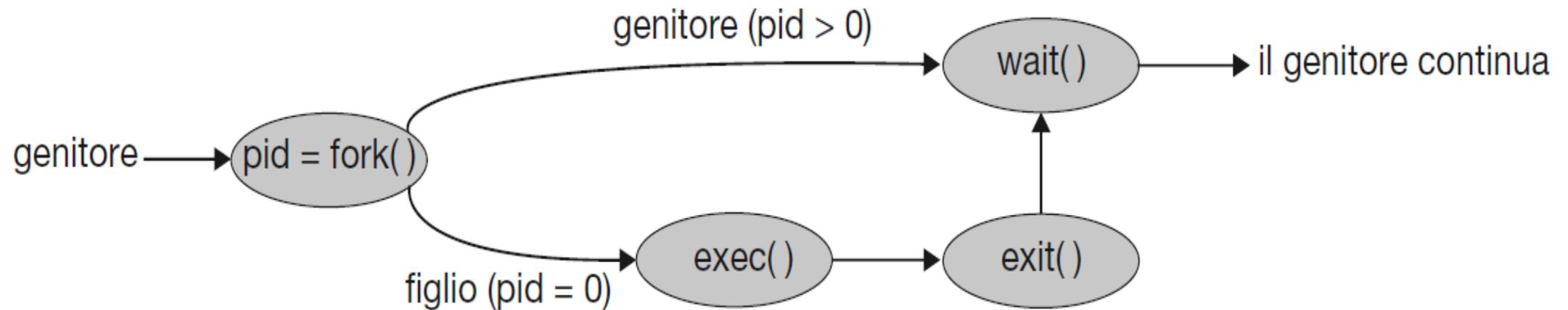
# fork() - UNIX



```
bloisi@bloisi-U36SG: ~/Documents/universita/didattica/sistemi-operativi/code/ch3
bloisi@bloisi-U36SG:~/Documents/universita/didattica/sistemi-operativi/code/ch3$
gcc -o newproc-posix newproc-posix.c
bloisi@bloisi-U36SG:~/Documents/universita/didattica/sistemi-operativi/code/ch3$
./newproc-posix
I am the parent 3228
I am the child 0
DateClient.java  fig3-33.c      newproc-posix      shm-posix-producer.c
DateServer.java  fig3-34.c      newproc-posix.c    simple-shell.c
fig3-30.c        fig3-35.c      newproc-win32.c    unix_pipe.c
fig3-31.c        multi-fork     pid.c              win32-pipe-child.c
fig3-32.c        multi-fork.c  shm-posix-consumer.c win32-pipe-parent.c
Child Complete
bloisi@bloisi-U36SG:~/Documents/universita/didattica/sistemi-operativi/code/ch3$
```

# fork() - UNIX

---



**Figura 3.9** Creazione di un processo utilizzando la chiamata di sistema `fork()`.

# createProcess() Windows

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

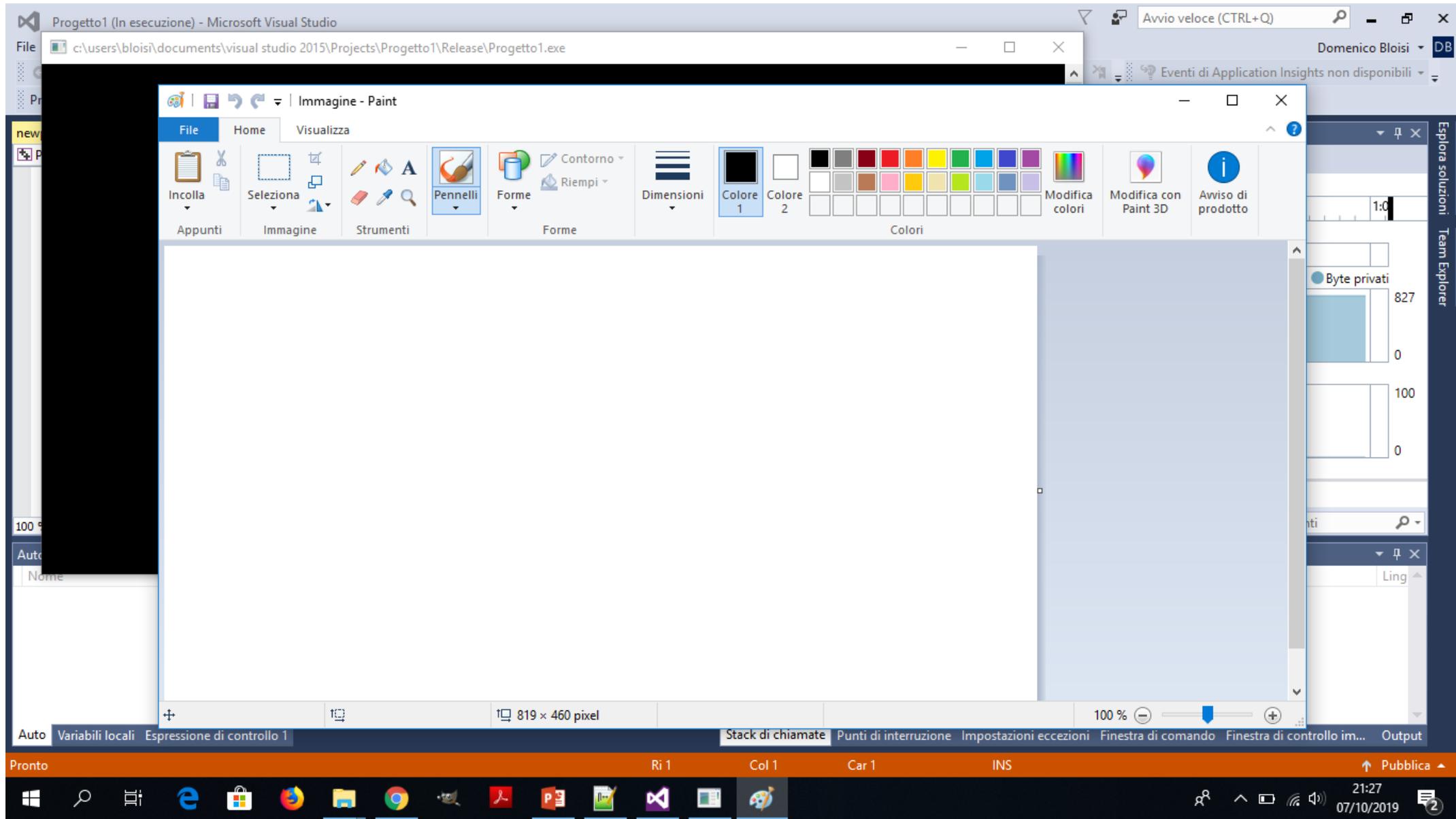
int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* alloca la memoria */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* genera processo figlio */
    if (!CreateProcess(NULL, /* usa riga di comando */
        "C:\\WINDOWS\\system32\\mspaint.exe", /* riga di comando */
        NULL, /* non eredita l'handle del processo */
        NULL, /* non eredita l'handle del thread */
        FALSE, /* disattiva l'ereditarieta' degli handle */
        0, /*nessun flag di creazione */
        NULL, /* usa il blocco ambiente del genitore */
        NULL, /* usa la directory esistente del genitore */
        &si,
        &pi))
    {
        fprintf(stderr, "generazione del nuovo processo fallita");
        return -1
    }
    /* il genitore attende il completamento del figlio */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("il processo figlio ha terminato");

    /* rilascia gli handle */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

# createProcess() - Windows



# Terminazione dei processi

---

Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato usando la **chiamata di sistema exit()**.

Un processo genitore può porre termine all'esecuzione di uno dei suoi processi figli per diversi motivi:

Il processo figlio ha ecceduto nell'uso di alcune tra le risorse che gli sono state assegnate.

Il compito assegnato al processo figlio non è più richiesto.

Il processo genitore termina e il sistema operativo non consente a un processo figlio di continuare l'esecuzione in tale circostanza

# Gerarchia dei processi Android

---



# Interprocess communication (IPC)

---

Un ambiente che consente la *comunicazione tra processi* (IPC, *Interprocess Communication*) può essere utile per diverse ragioni.



**Condivisione d'informazioni**  
**Velocizzazione del calcolo**  
**Modularità**

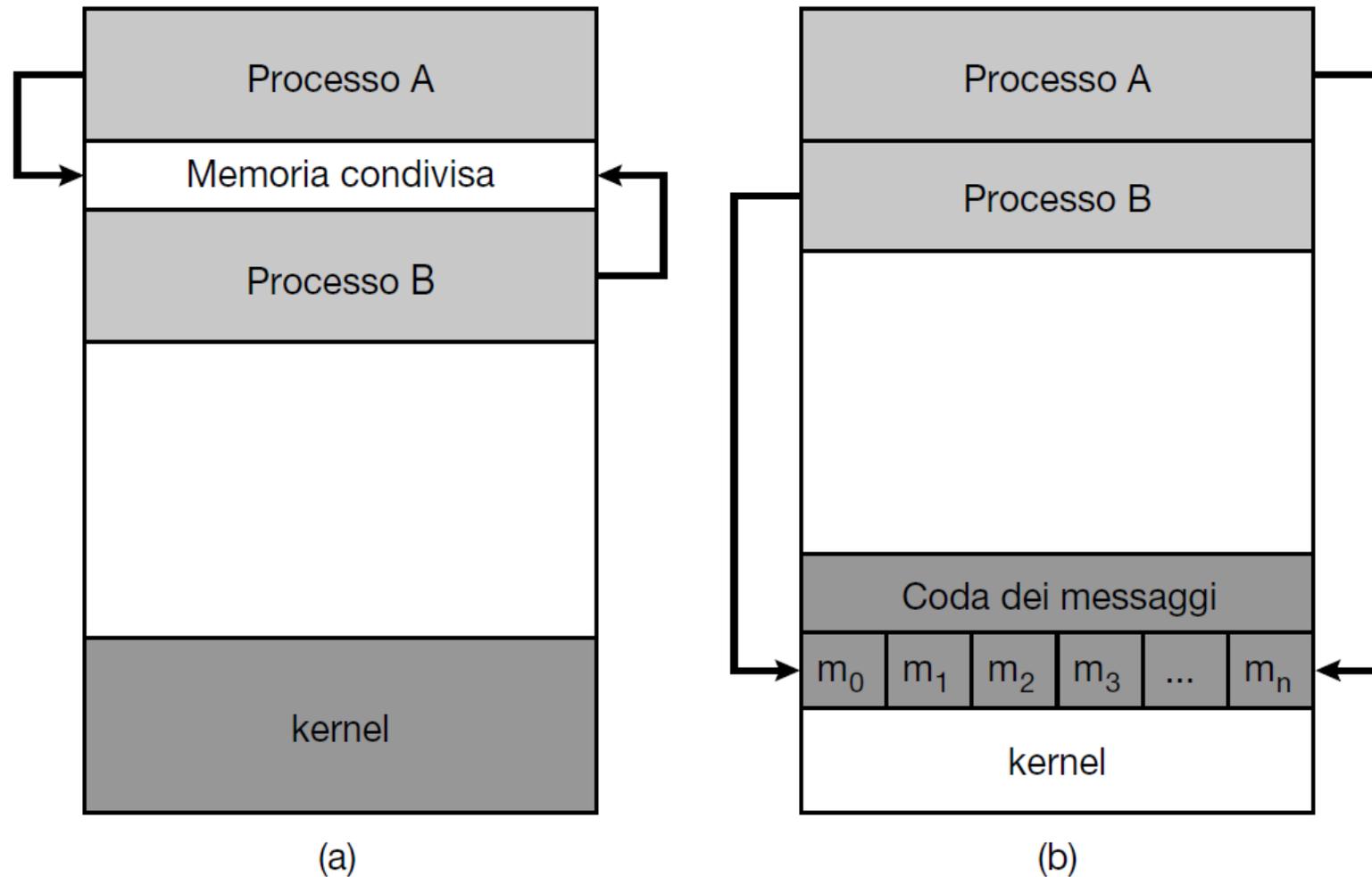
I modelli fondamentali della *comunicazione tra processi*



**a memoria condivisa**  
**a scambio di messaggi**

- Nei sistemi operativi sono diffusi entrambi i modelli; spesso coesistono in un unico sistema.

# Modelli di comunicazione



**Figura 3.11** Modelli di comunicazione. (a) Memoria condivisa. (b) Scambio di messaggi.

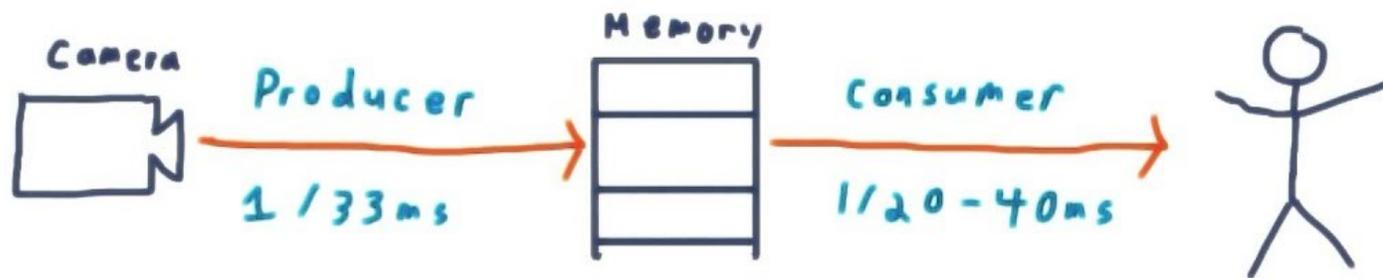
# IPC in sistemi a memoria condivisa

## PROBLEMA DEL PRODUTTORE/CONSUMATORE

Utile per illustrare il concetto di cooperazione tra processi

Un processo **produttore** produce informazioni che sono consumate da un processo **consumatore**

### Producer - Consumer Pattern



# Produttore con memoria condivisa

---

```
item next_produced;
while (true) {
    /* produce un elemento in next_produced */
    while (((in + 1) % BUFFER_SIZE) == out); /* non fa niente */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

**Figura 3.12** Processo produttore con l'uso della memoria condivisa.

# Consumatore con memoria condivisa

---

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* non fa niente */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consuma l'elemento in next_consumed */
}
```

**Figura 3.13** Processo consumatore con l'utilizzo della memoria condivisa.

# IPC in sistemi a scambio di messaggi

Lo **scambio di messaggi** è un meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza condividere lo stesso spazio di indirizzi.

È una tecnica particolarmente utile negli ambienti distribuiti, dove i processi possono risiedere su macchine diverse connesse da una rete → **web chat**

canale di comunicazione



comunicazione  
diretta o  
indiretta

gestione  
automatica o  
esplicita del  
buffer

comunicazione  
sincrona o  
asincrona

# Sincronizzazione tra processi

---

La comunicazione tra processi avviene attraverso chiamate delle primitive `send()` e `receive()`

Lo scambio di messaggi può essere

- **sincrono (o bloccante)**
- **asincrono (o non bloccante)**



# Produttore con scambio di messaggi

---

```
message next_produced;

while (true) {
    /* produce un elemento in next_produced */

    send(next_produced);
}
```

**Figura 3.14** Processo produttore con l'utilizzo dello scambio di messaggi.

# Consumatore con scambio di messaggi

---

```
message next_consumed;

while (true) {
    receive(next_consumed);

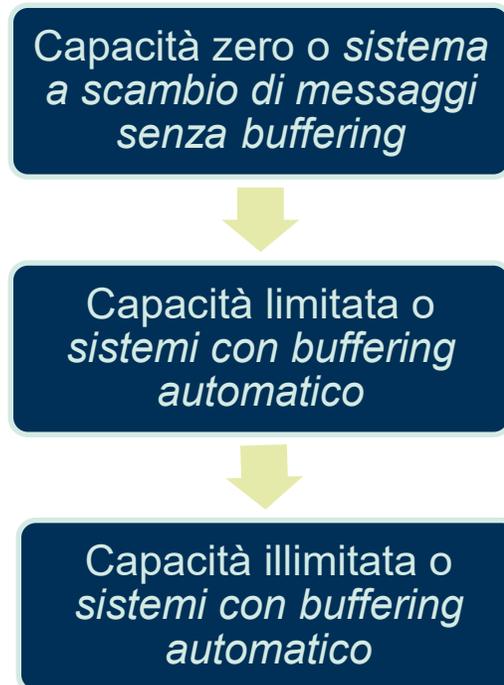
    /* consuma l'elemento in next_consumed */
}
```

**Figura 3.15** Processo consumatore con l'utilizzo dello scambio di messaggi.

# Code di messaggi

---

I messaggi scambiati tra processi comunicanti risiedono in **code** temporanee. Esistono tre modi per realizzare queste code.



# Esempi di sistemi IPC (comunicazione tra processi)

---

1. La API POSIX  
basata sulla  
memoria  
condivisa

2. Lo scambio di  
messaggi nel  
sistema  
operativo Mach

3. La  
comunicazione  
fra processi in  
Windows

4. Le pipe, canali  
di  
comunicazione  
tra processi

# Codice POSIX produttore/consumatore

---

Vedremo ora il codice POSIX di due programmi che usano la **memoria condivisa** per implementare il modello produttore-consumatore.

Il produttore definisce un oggetto memoria condivisa e scrive sulla memoria condivisa, il consumatore legge dalla memoria condivisa.

# Produttore - memoria condivisa - POSIX

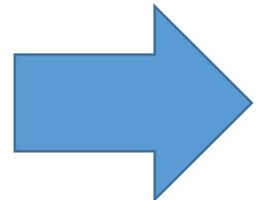
---

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()

{
/* dimensione, in byte, dell'oggetto memoria condivisa */
const int SIZE 4096;
/* nome dell'oggetto memoria condivisa */
const char *name = "OS";
/* stringa scritta nella memoria condivisa */
const char *message_0 = "Hello";
const char *message_1 = "World!";
```



# Produttore - memoria condivisa - POSIX

---

```
/* descrittore del file di memoria condivisa */
int fd;
/* puntatore all'oggetto memoria condivisa */
char *ptr;

/* crea l'oggetto memoria condivisa */
fd = shm_open(name, O_CREAT | O_RDWR, 0666);

/* configura la dimensione dell'oggetto memoria condivisa */
ftruncate(fd, SIZE);

/* mappa in memoria l'oggetto memoria condivisa */
ptr = (char *)
    mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

/* scrive sull'oggetto memoria condivisa */
sprintf(ptr, "%s", message_0);
ptr += strlen(message_0);
sprintf(ptr, "%s", message_1);
ptr += strlen(message_1);

return 0;
}
```

**Figura 3.16** Processo produttore che illustra l'API per la memoria condivisa POSIX.

# Produttore - memoria condivisa - POSIX

---

```
bloisi@bloisi-U36SG: ~/Documents/universita/didattica/sistemi-operativi/code/ch3
bloisi@bloisi-U36SG:~/Documents/universita/didattica/sistemi-operativi/code/ch3$
gcc -o shm-posix-producer shm-posix-producer.c -lrt
bloisi@bloisi-U36SG:~/Documents/universita/didattica/sistemi-operativi/code/ch3$
./shm-posix-producer
bloisi@bloisi-U36SG:~/Documents/universita/didattica/sistemi-operativi/code/ch3$
█
```

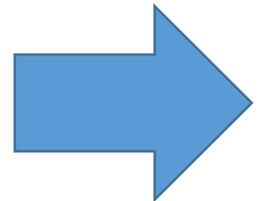
# Consumatore - memoria condivisa - POSIX

---

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* dimensione, in byte, dell'oggetto memoria condivisa */
const int SIZE 4096;
/* nome dell'oggetto memoria condivisa */
const char *name = "OS";
/* descrittore del file di memoria condivisa */
int fd;
/* puntatore all'oggetto memoria condivisa */
void *ptr;
```



# Consumatore - memoria condivisa - POSIX

---

```
/* apre l'oggetto memoria condivisa */
fd = shm_open(name, O_RDONLY, 0666);

/* mappa in memoria l'oggetto memoria condivisa */
ptr = (char *)
    mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

/* legge dall'oggetto memoria condivisa */
printf("%s", (char *)ptr);

/* rimuove l'oggetto memoria condivisa */
shm_unlink(name);

return 0;
}
```

**Figura 3.17** Processo consumatore che illustra l'API per la memoria condivisa POSIX.

# Consumatore - memoria condivisa - POSIX

---

```
bloisi@bloisi-U36SG: ~/Documents/universita/didattica/sistemi-operativi/code/ch3
bloisi@bloisi-U36SG:~/Documents/universita/didattica/sistemi-operativi/code/ch3$
gcc -o shm-posix-consumer shm-posix-consumer.c -lrt
bloisi@bloisi-U36SG:~/Documents/universita/didattica/sistemi-operativi/code/ch3$
./shm-posix-consumer
Studying Operating Systems Is Fun!bloisi@bloisi-U36SG:~/Documents/universita/did
attica/sistemi-operativi/code/ch3$
```

# Esempio - scambio messaggi - Mach

---

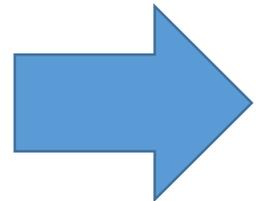
```
#include<mach/mach.h>

struct message {
    mach_msg_header_t header;
    int data;
};

mach_port_t client;
mach_port_t server;

    /* Codice del client */
struct message message;

// costruisce l'intestazione
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;
```



# Esempio - scambio messaggi - Mach

---

```
// invia il messaggio
mach_msg(&message.header, // intestazione del messaggio
        MACH_SEND_MSG, // invia un messaggio
        sizeof(message), // dimensione del messaggio spedito
        0, // dimensione massima del messaggio ricevuto – non necessario
        MACH_PORT_NULL, // nome della porta di ricezione - non necessario
        MACH_MSG_TIMEOUT_NONE, // nessun timeout
        MACH_PORT_NULL // nessuna porta di notifica
);

        /* Codice del server */

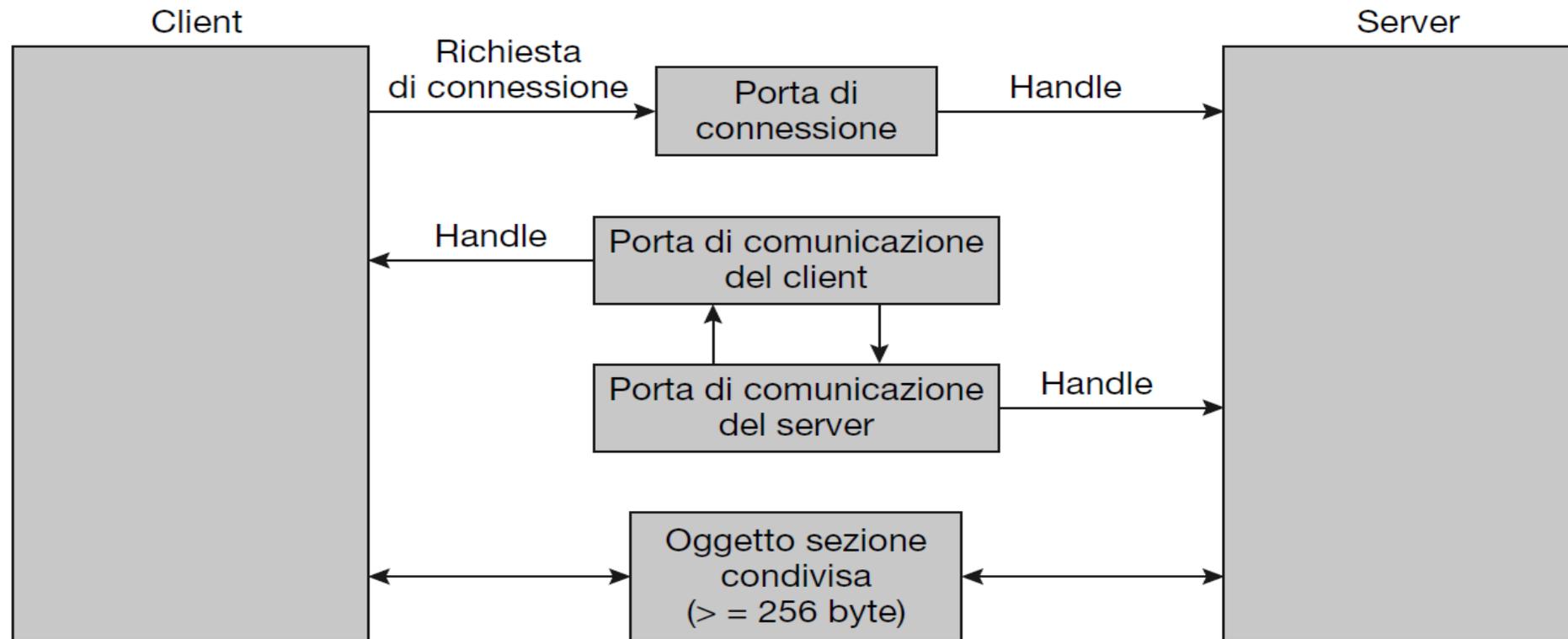
struct message message;

// riceve un messaggio
mach_msg(&message.header, // intestazione del messaggio
        MACH_RCV_MSG, // riceve un messaggio
        0, // dimensione del messaggio spedito
        sizeof(message), // dimensione massima del messaggio ricevuto
        server, // nome della porta di ricezione
        MACH_MSG_TIMEOUT_NONE, // nessun timeout
        MACH_PORT_NULL // nessuna porta di notifica
);
```

**Figura 3.18** Esempio di scambio di messaggi in Mach.

# Windows – Comunicazione tra processi

La funzione di scambio di messaggi di Windows è detta **chiamata di procedura locale avanzata** (*advanced local procedure call, ALPC*)

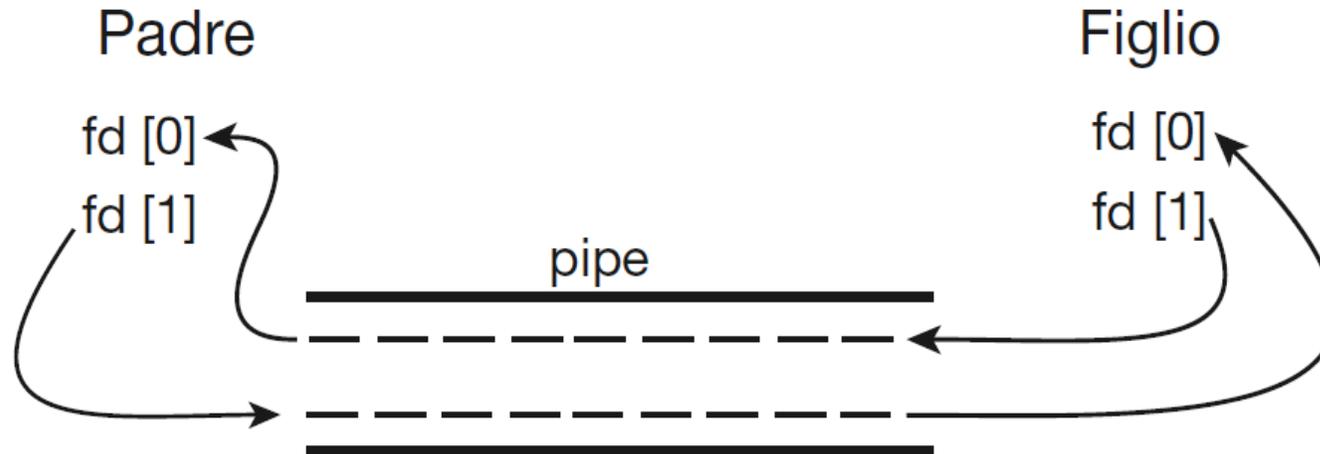


**Figura 3.19** Chiamate di procedura locale avanzate in Windows.

# Pipe convenzionali – Named pipe

Una **pipe** agisce come canale di comunicazione tra processi.

Le **pipe convenzionali** permettono a due processi di comunicare secondo una modalità standard chiamata del **produttore-consumatore**



**Figura 3.20** Descrittori di file per una pipe convenzionale.

# Pipe convenzionali UNIX

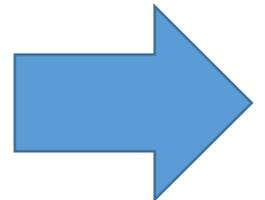
---

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define _BUFFER_SIZE 25
#define _READ_END 0
#define _WRITE_END 1

int main(void)
{
char write_msg[_BUFFER_SIZE] = "Greetings";
char read_msg[_BUFFER_SIZE];
int fd[2];
pid_t pid;
```

In questo programma, il processo padre crea una **pipe** e in seguito esegue una chiamata `fork()`, generando un processo figlio.



# Pipe convenzionali UNIX

---

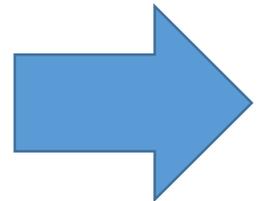
```
/* crea la pipe */
if (pipe(fd) == -1) {
    fprintf(stderr, "Pipe failed");
    return 1;
}

/* crea tramite fork un processo figlio */
pid = fork();

if (pid < 0) { /* errore */
    fprintf(stderr, "Fork Failed");
    return 1;
}

if (pid > 0) { /* processo padre */
    /* chiude l'estremità inutilizzata della pipe */
    close(fd[READ_END]);

    /* scrive sulla pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
}
```



# Pipe convenzionali UNIX

---

```
    /* chiude l'estremità della pipe dedicata alla scrittura */
    close(fd[WRITE_END]);

}
else { /* processo figlio */
    /* chiude l'estremità inutilizzata della pipe */
    close(fd[WRITE_END]);

    /* legge dalla pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read %s", read_msg);

    /* chiude l'estremità della pipe dedicata alla lettura */
    close(fd[READ_END]);
}

return 0;
}
```

# Pipe anonime Windows

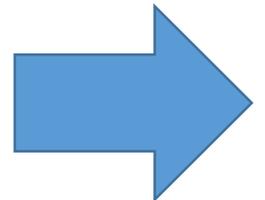
---

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char message[BUFFER_SIZE] = "Greetings";
DWORD written;
```

In questo programma, il processo padre crea una **pipe anonima** per comunicare con il proprio figlio.



# Pipe anonime Windows (padre)

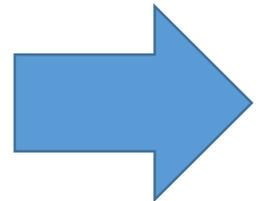
---

```
/*imposta gli attributi di sicurezza in modo che le pipe siano
ereditate */
SECURITY_ATTRIBUTES sa = {sizeof(SEcurity_ATTRIBUTES),NULL,TRUE};
/* alloca la memoria */
ZeroMemory(&pi, sizeof(pi));

/* crea la pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
    fprintf(stderr, "Create Pipe Failed");
    return 1;
}

/* prepara la struttura START_INFO per il processo figlio */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* reindirizza lo standard input verso l'estremità della pipe
dedicata alla lettura */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;
```



# Pipe anonime Windows (padre)

---

```
/* non permette al processo figlio di ereditare l'estremità
   della pipe dedicata alla scrittura */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* crea il processo figlio */
CreateProcess(NULL, "child.exe", NULL, NULL,
  TRUE, /* inherit handles */
  0, NULL, NULL, &si, &pi);

/* chiude l'estremità inutilizzata della pipe */
CloseHandle(ReadHandle);

/* il padre scrive sulla pipe */
if (!WriteFile(WriteHandle, message, BUFFER_SIZE, &written, NULL))
  fprintf(stderr, "Error writing to pipe.");

/* chiude l'estremità della pipe dedicata alla scrittura */
CloseHandle(WriteHandle);

/* attende la terminazione del processo figlio */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

# Pipe anonime Windows (figlio)

---

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

/* riceve l'handle di lettura della pipe */
ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

/* il figlio legge dalla pipe */
if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
    printf("child read %s",buffer);
else
    fprintf(stderr, "Error reading from pipe");

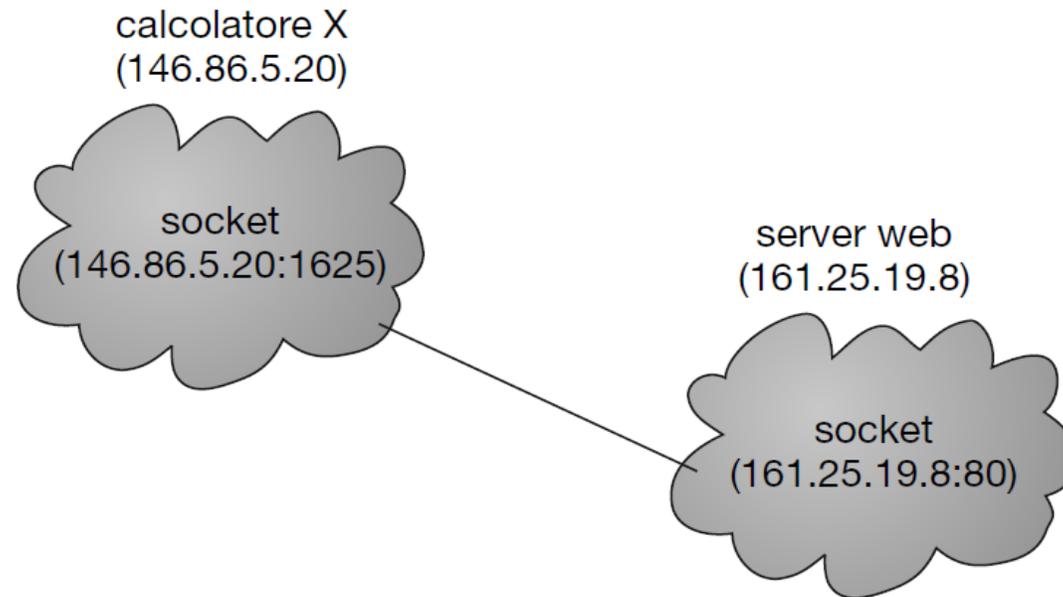
return 0;
}
```

# Socket

---

Una coppia di processi che comunica attraverso una rete usa una coppia di **socket**, una per ogni processo.

Ogni socket è identificata da un indirizzo IP concatenato a un numero di porta.



Le **socket** generalmente impiegano un'architettura client-server

**Figura 3.26** Comunicazione tramite socket.

# Socket in Java

---

Useremo il **linguaggio Java per implementare le socket**, poiché Java offre un'interfaccia alle socket più semplice e dispone di una ricca libreria di utilità di networking.



# Server

---

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

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            /* si pone in ascolto di richieste di connessione */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* scrive la Data sulla socket */
                pout.println(new java.util.Date().toString());

                /* chiude la socket */
                /* ritorna in ascolto di nuove richieste */
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

**Figura 3.27** Server che fornisce al client la data corrente.

# Client

---

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

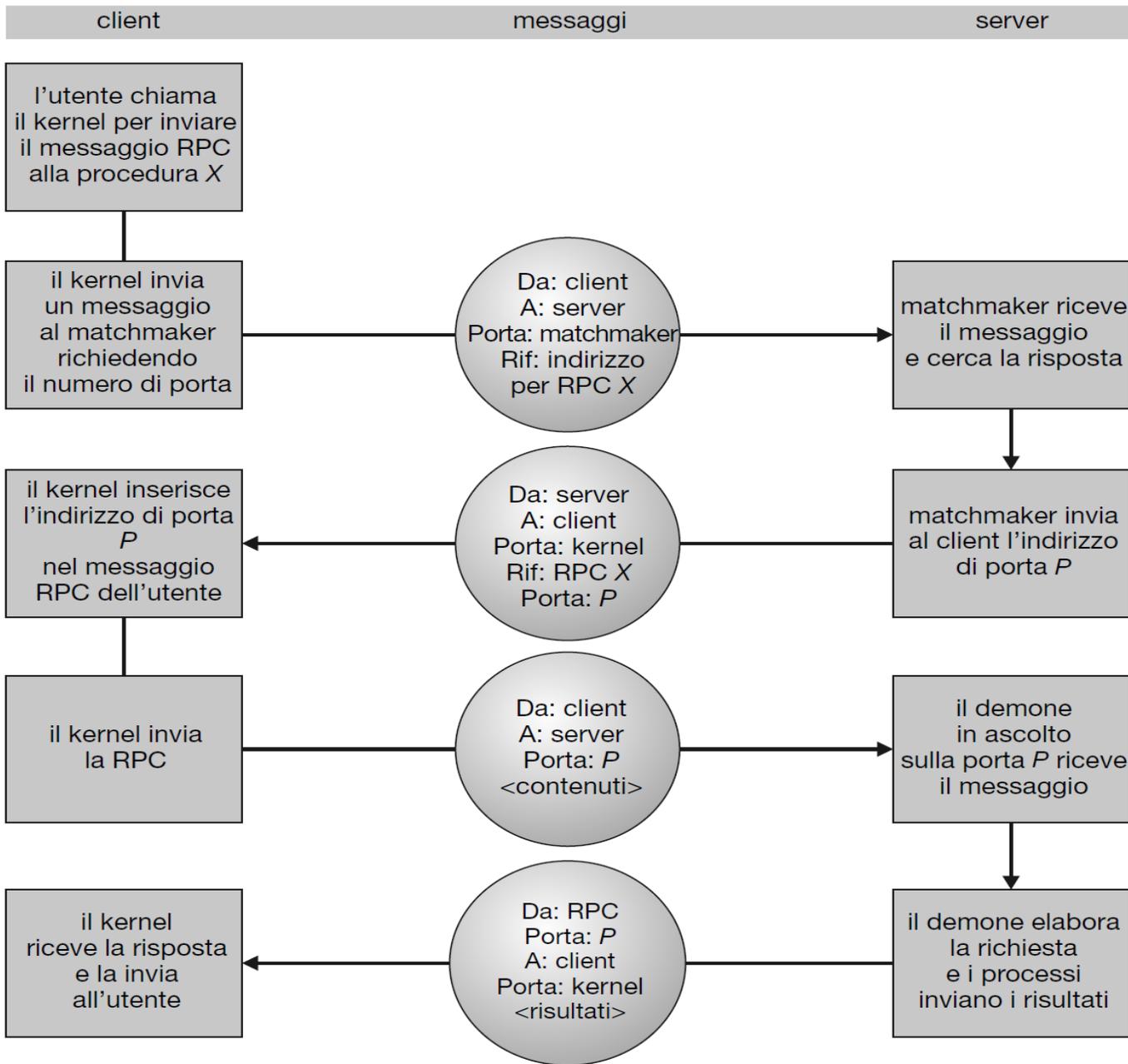
public class DateClient
{
    public static void main(String[] args) {
        try {
            /* si collega alla porta su cui ascolta il server */
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* legge la data dalla socket */
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* chiude la socket */
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

**Figura 3.28** Client che riceve dal server la data corrente.



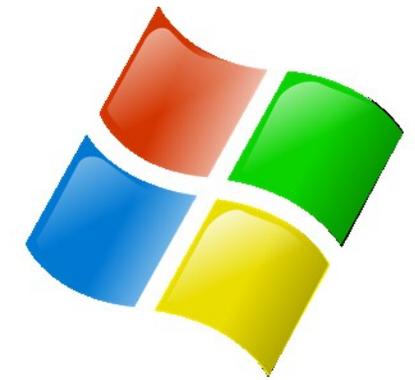
# Chiamate di procedure remote

Figura 3.29 Esecuzione di una chiamata di procedura remota (RPC).



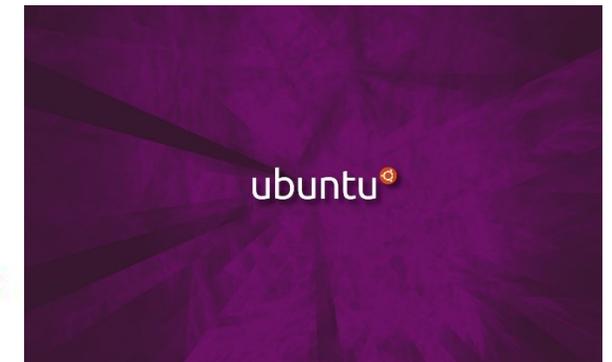
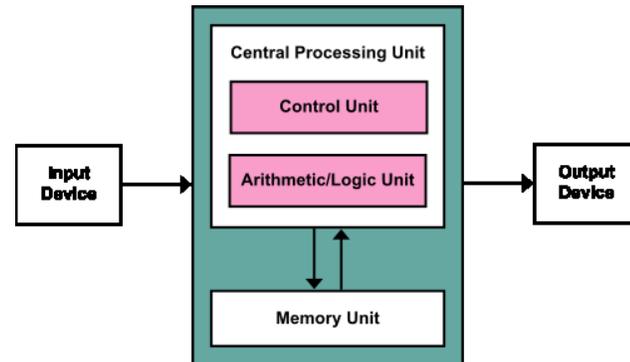
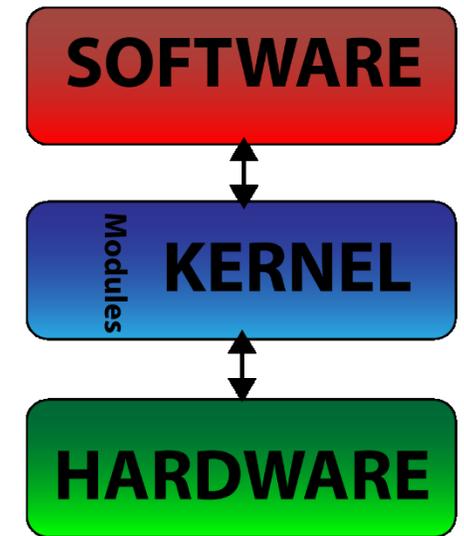
**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

*Corso di Sistemi Operativi  
A.A. 2019/20*



# Processi

Docente:  
**Domenico Daniele  
Bloisi**



Ottobre 2019