Gestione dei processi

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

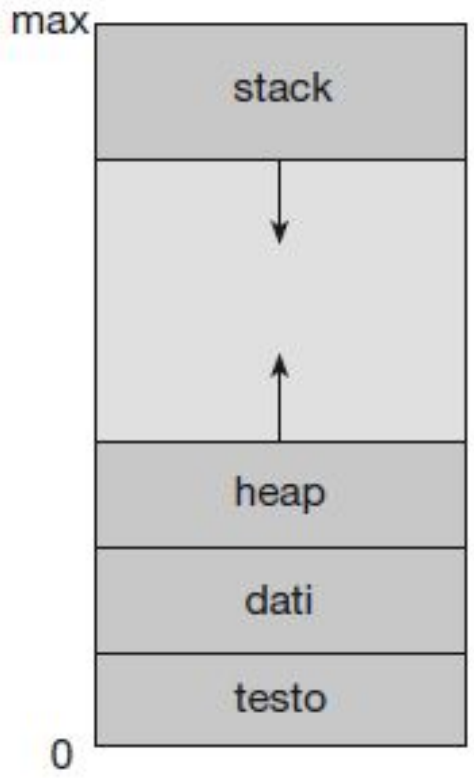


Figura 3.1 Struttura di un processo in memoria.

Stack Allocation C++

- L'allocazione avviene utilizzando blocchi di memoria contigui.
- La dimensione di memoria da allocare viene decisa dal compilatore in fase di compilazione.

```
int main()
{
  // Tutte le variabili seguenti
  // verranno allocate nello stack
  int v;
  int a[5];
  double d = 10.;
}
```

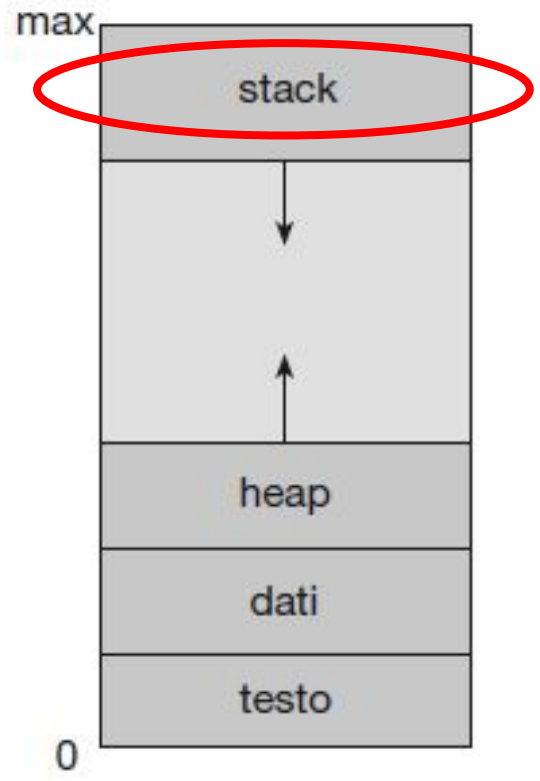


Figura 3.1 Struttura di un processo in memoria.

Heap Allocation C++

- L'allocazione avviene durante l'esecuzione delle istruzioni (runtime memory allocation).
- Un uso poco accorto dell'allocazione dinamica può generare memory leak.

```
int main()
{
  // La memoria per i 5 interi
  // verrà allocata nello heap
  int *a = new int[5];
}
```

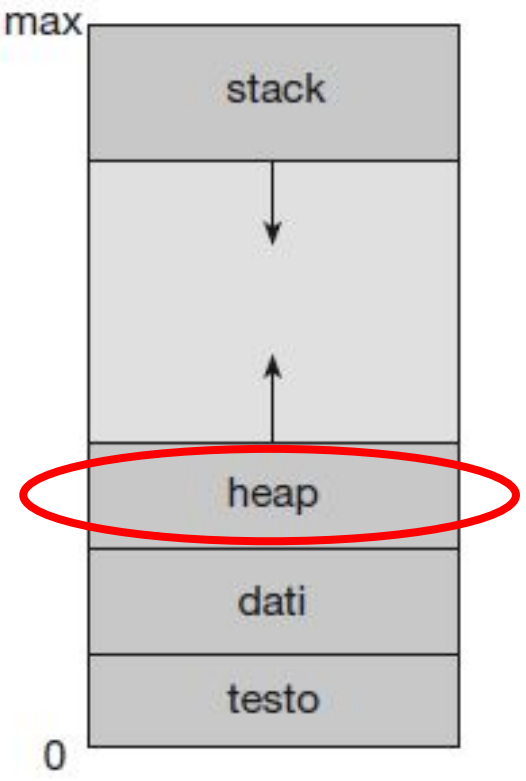


Figura 3.1 Struttura di un processo in memoria.

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 di elaborazione

Terminato. Il processo ha terminato l'esecuzione

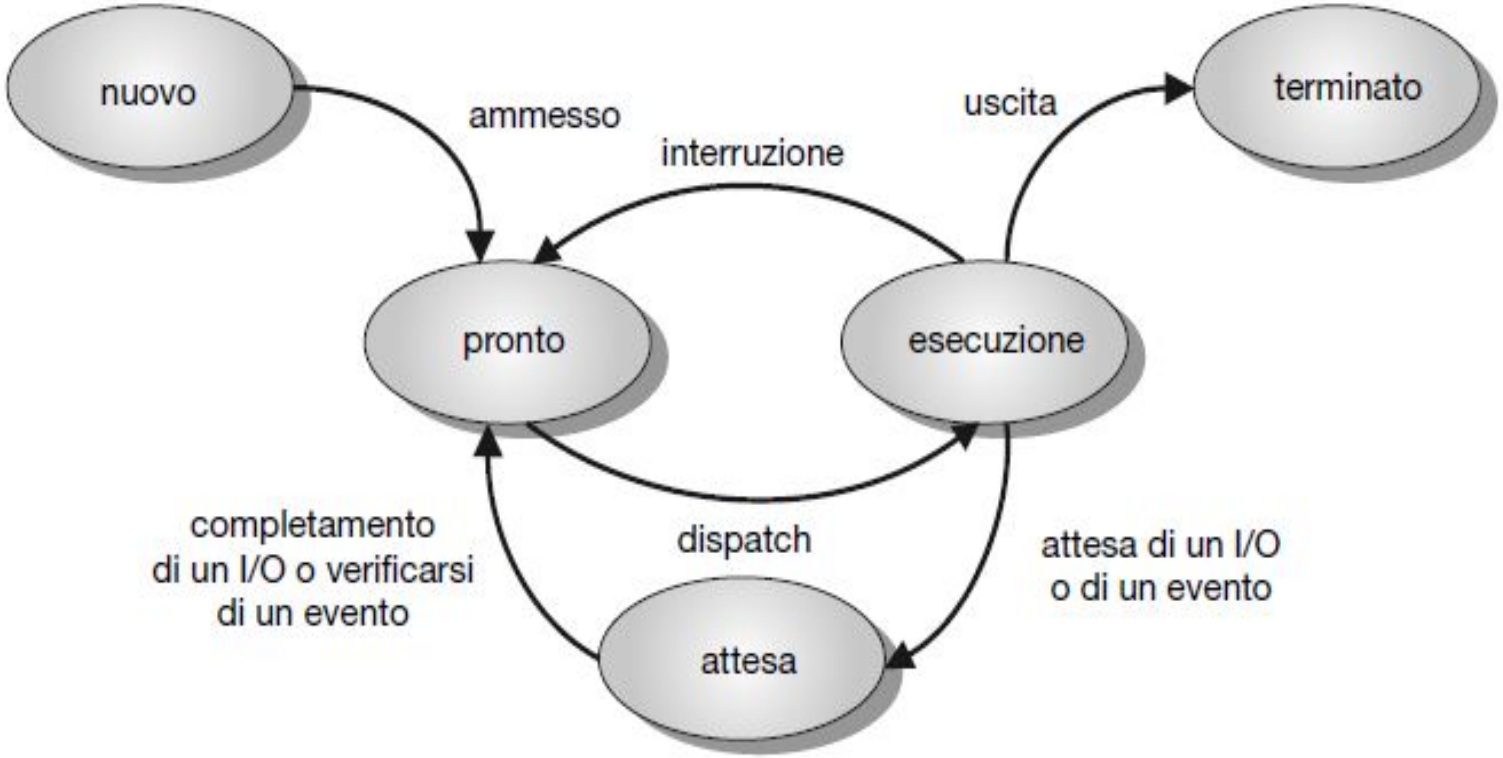


Figura 3.2 Diagramma di transizione degli stati di un processo.

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)

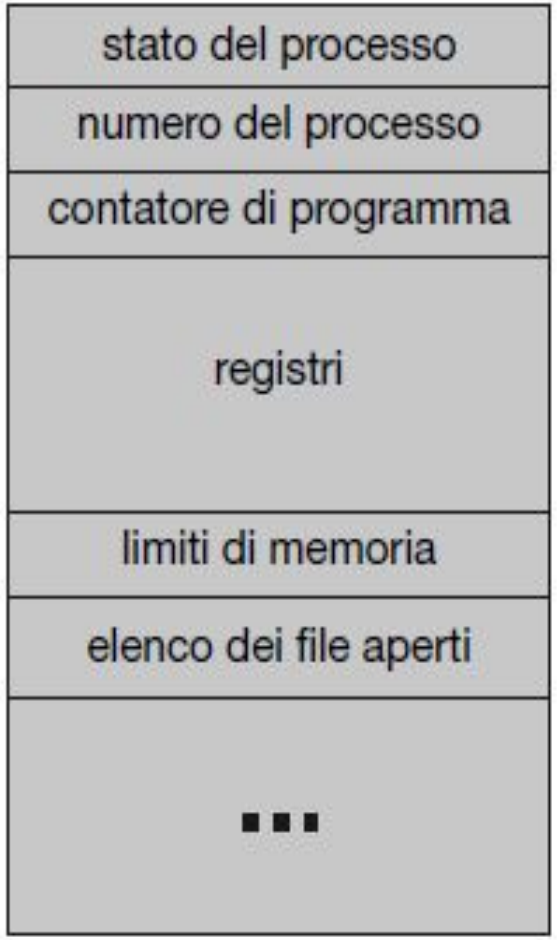
- Il PCB contiene un insieme di informazioni connesse a un processo specifico.

Elementi del 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 di uso generale e registri contenenti i codici di condizione (*condition codes*)



Inoltre:

- **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).

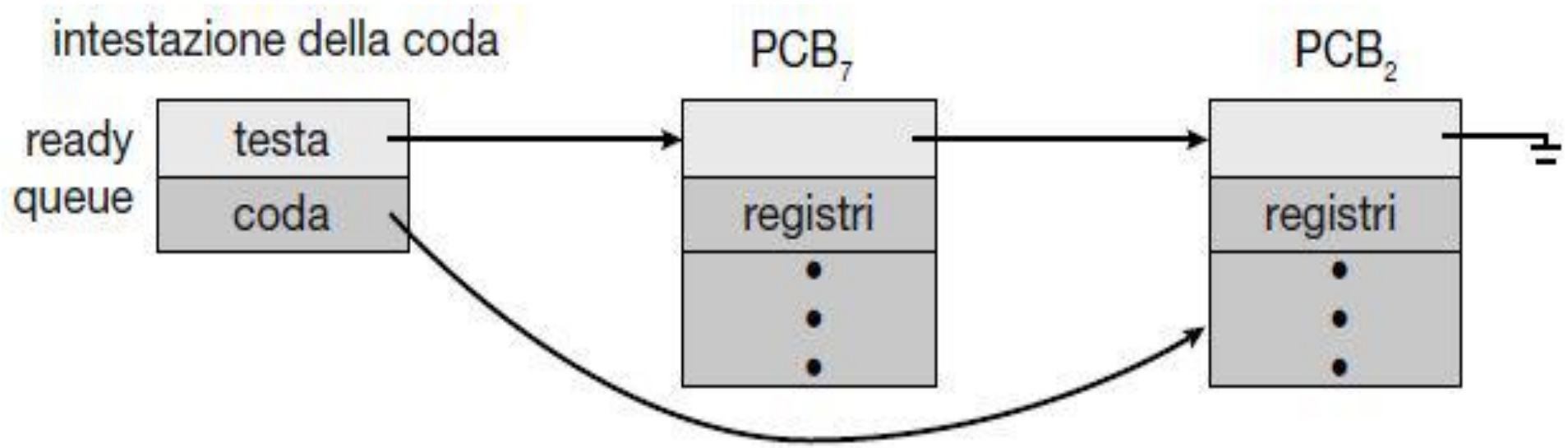
Scheduling dei processi

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

Ready queue

Ogni processo è inserito in una **coda di processi pronti** e in attesa di essere eseguiti, detta **coda dei processi pronti** *(ready queue).*



W

l p

co

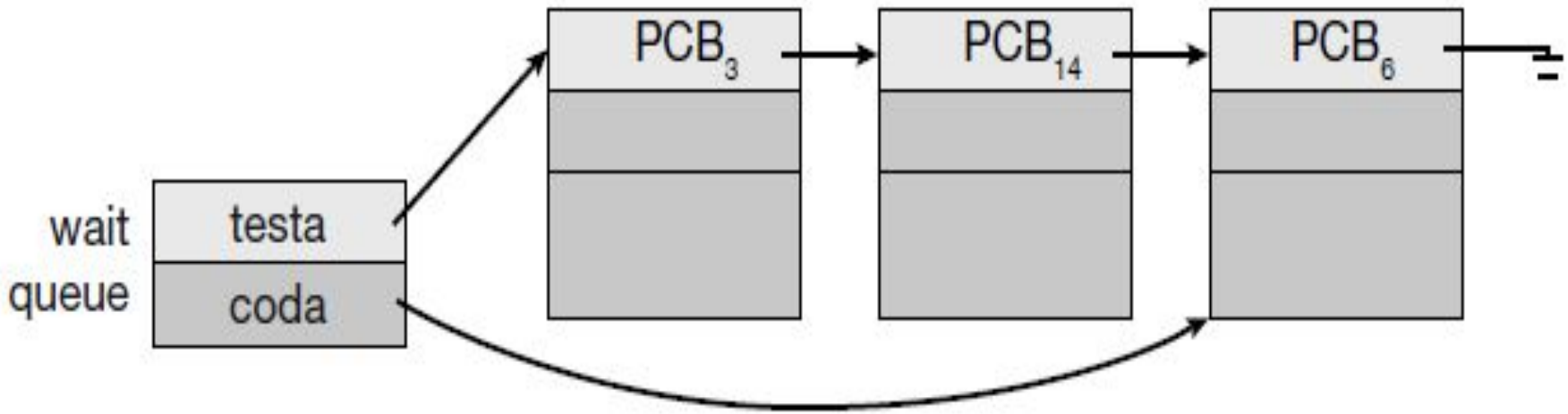


Diagramma di accodamento

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

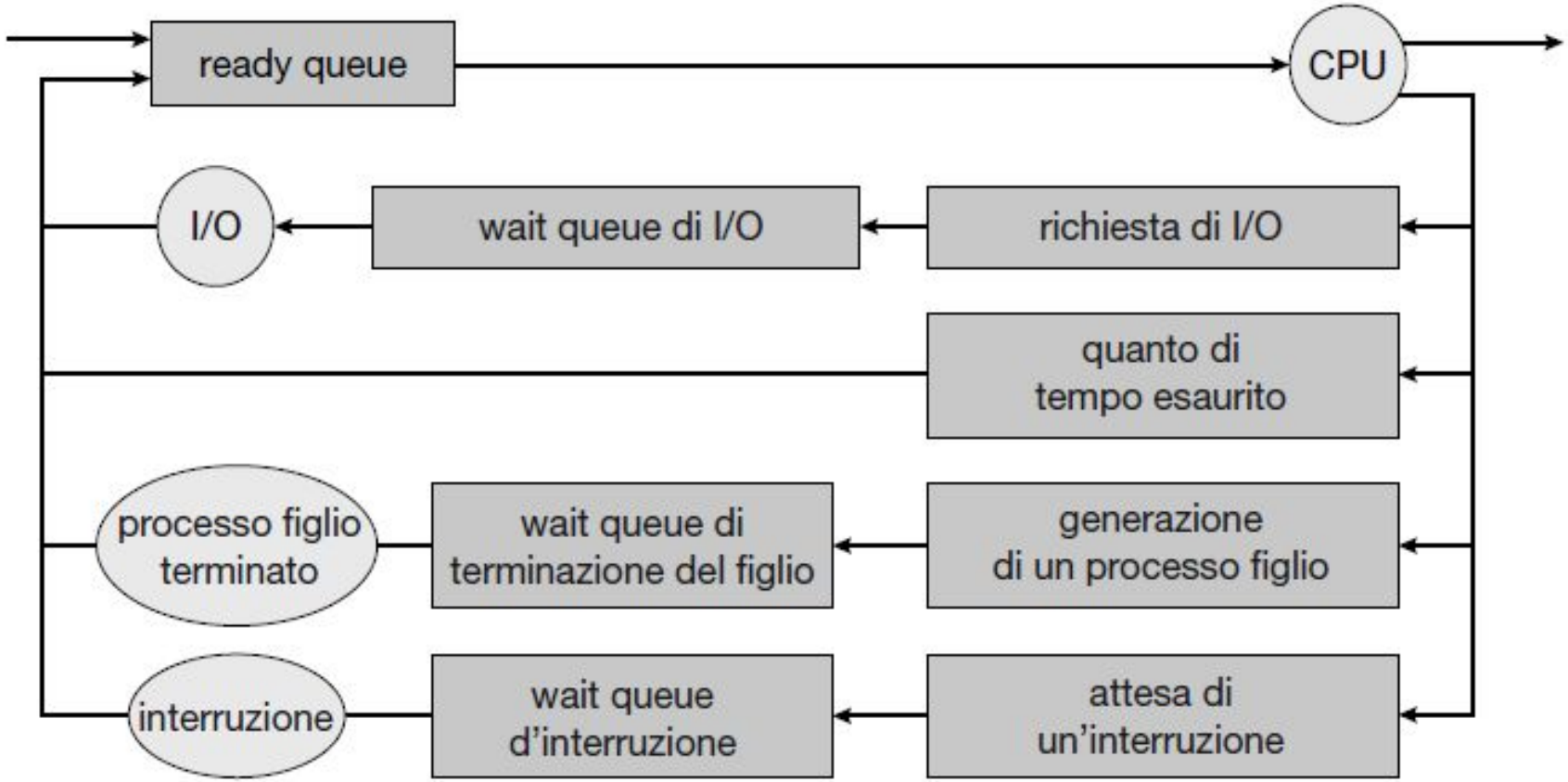


Figura 3.5 Diagramma di accodamento per lo scheduling dei processi.

Context switch

- In presenza di una interruzione (o di una system call), 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**

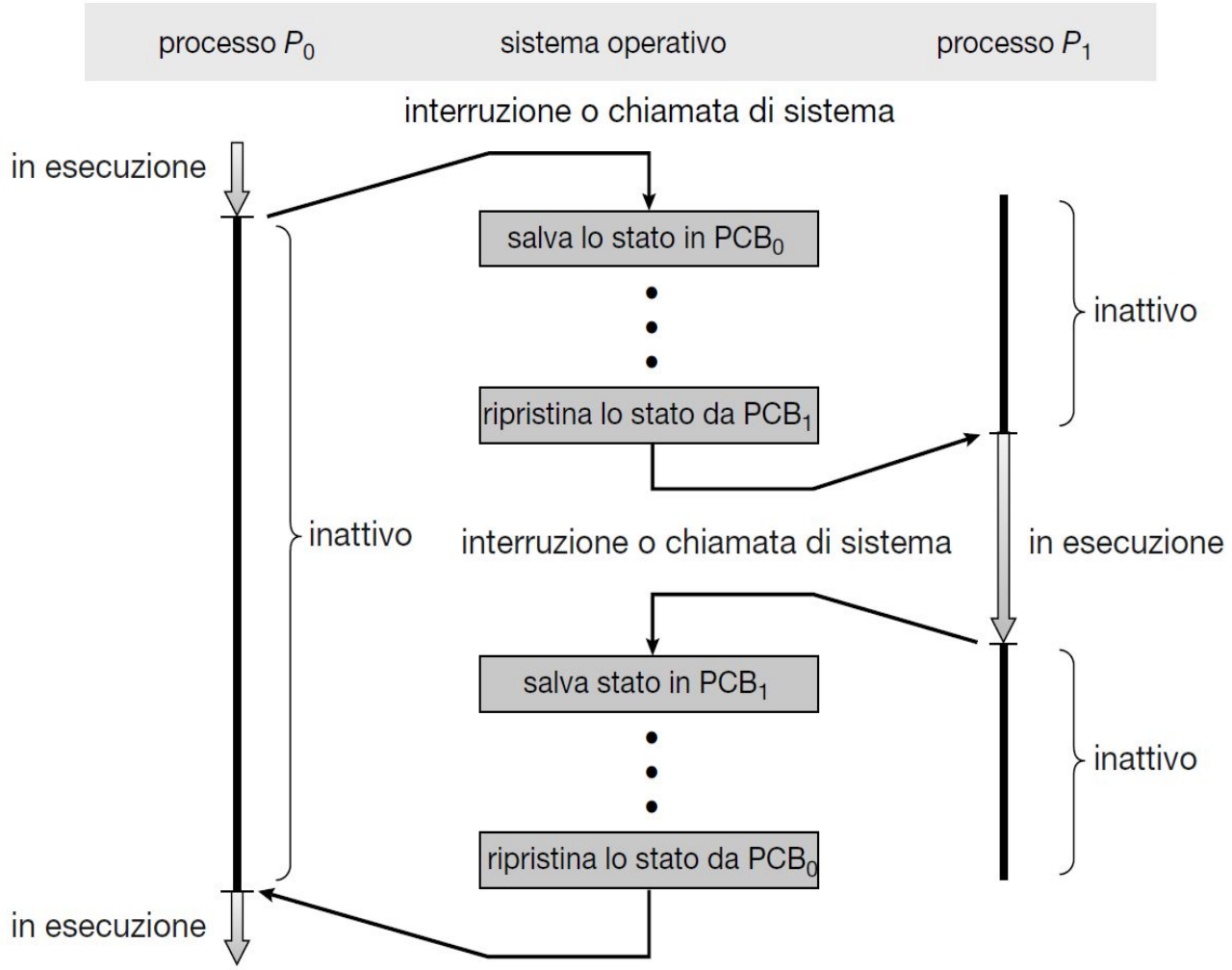


Figura 3.6 Diagramma di cambio di contesto.

Operazioni sui processi

Creazione dei processi

Albero dei processi

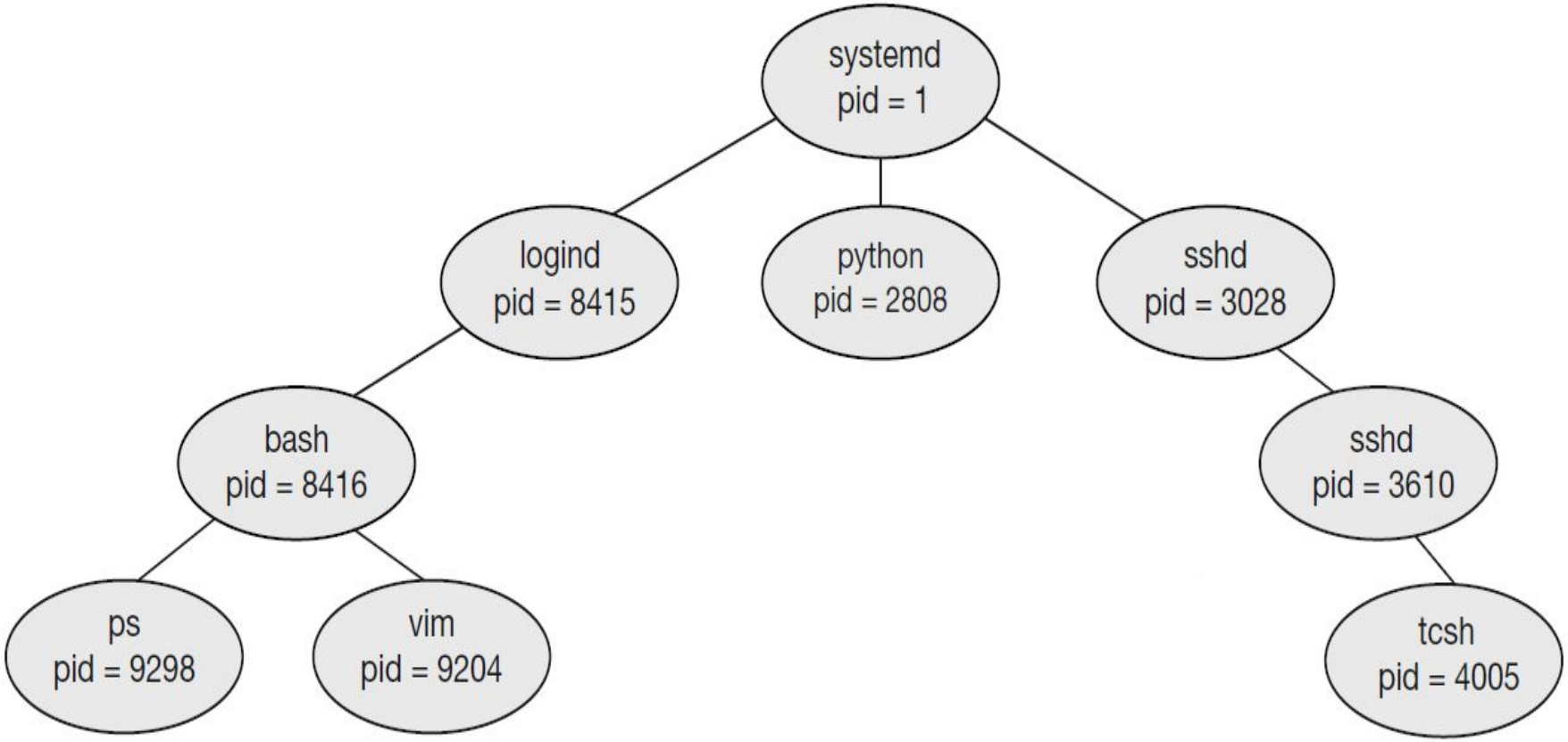


Figura 3.7 Esempio di albero dei processi di un tipico sistema Linux.

fork() - UNIX

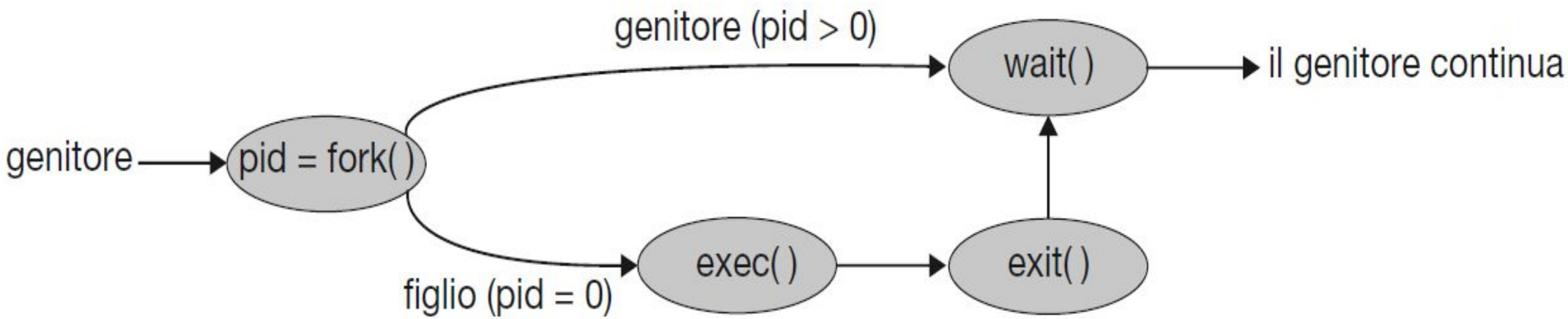


Figura 3.9 Creazione di un processo utilizzando la chiamata di sistema `fork( )`.

Esempio fork() - UNIX

```c
#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);
                wait(NULL);

                printf("Child Complete\n");
        }

    return 0;
}
```

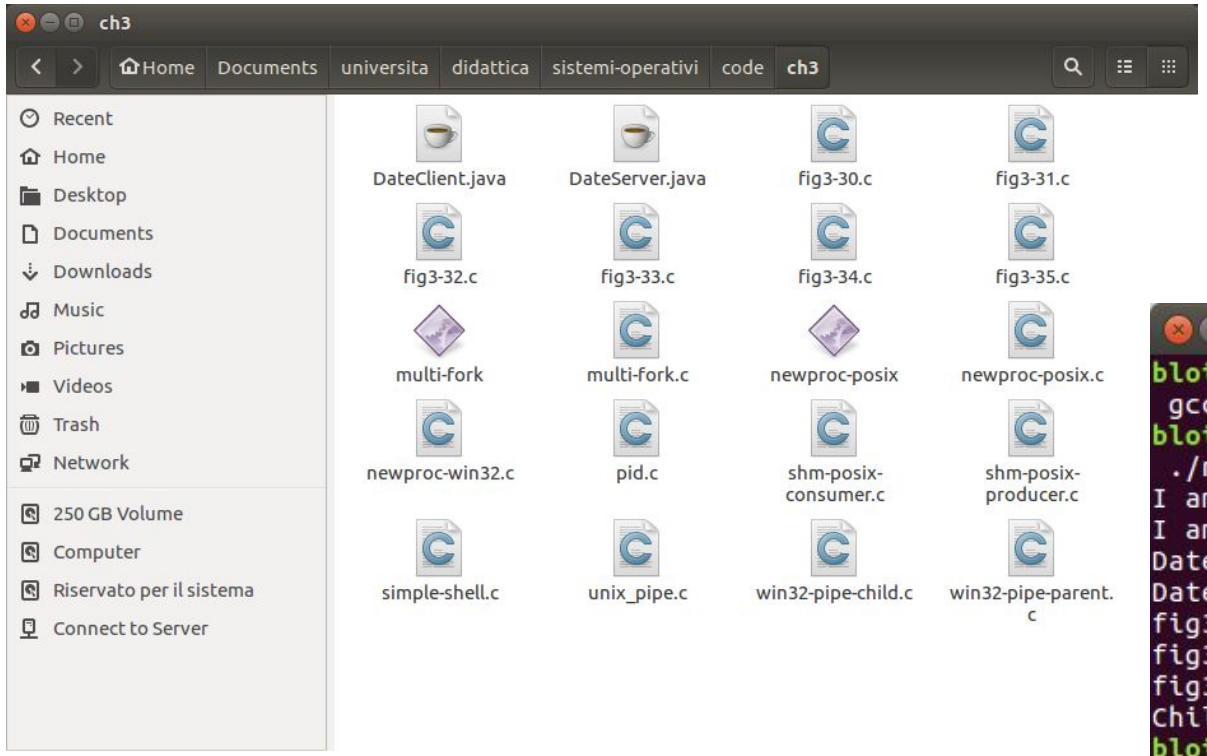
Creazione del processo figlio

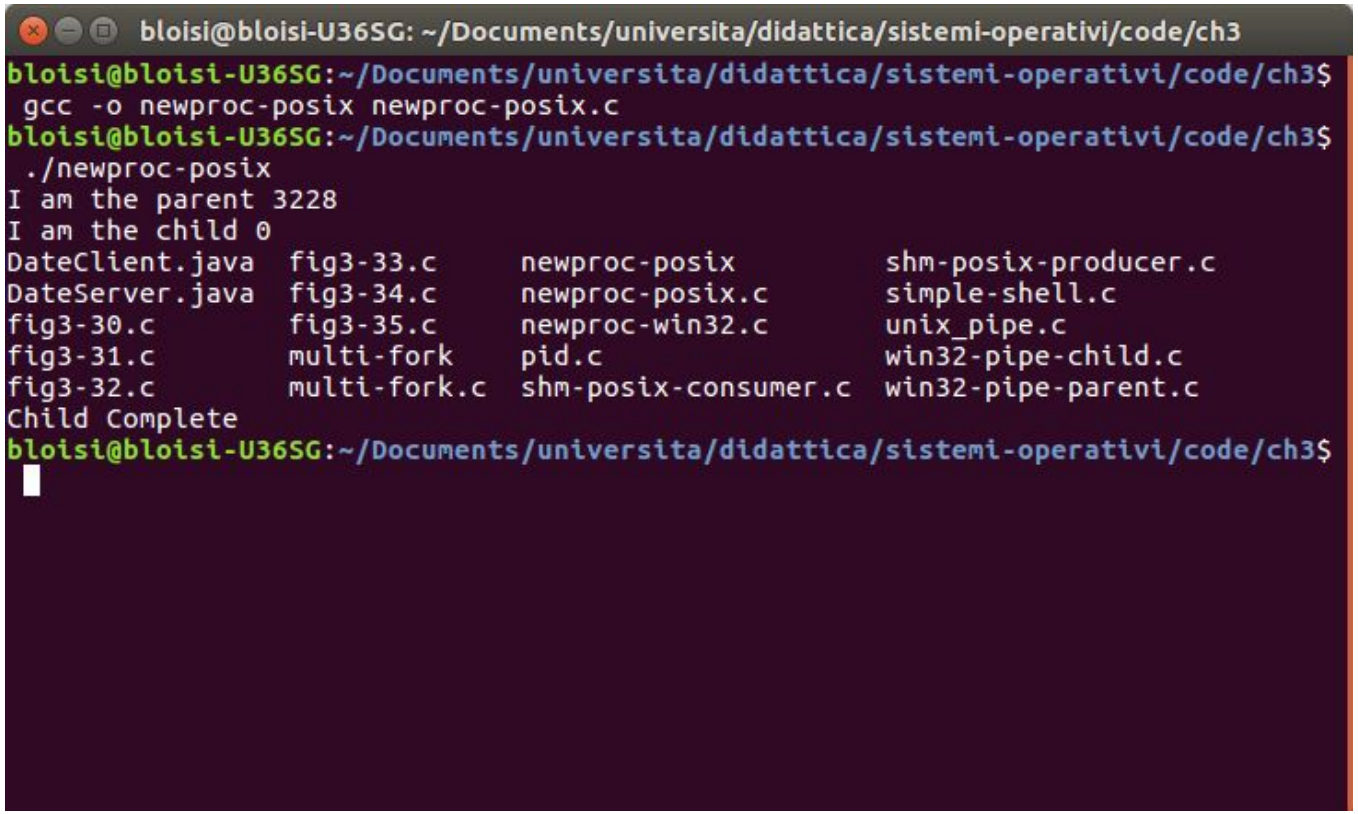
Gestione di un eventuale errore nella creazione del processo figlio

Istruzioni per il processo figlio

Istruzioni per il processo padre

Esecuzione dell'esempio fork()

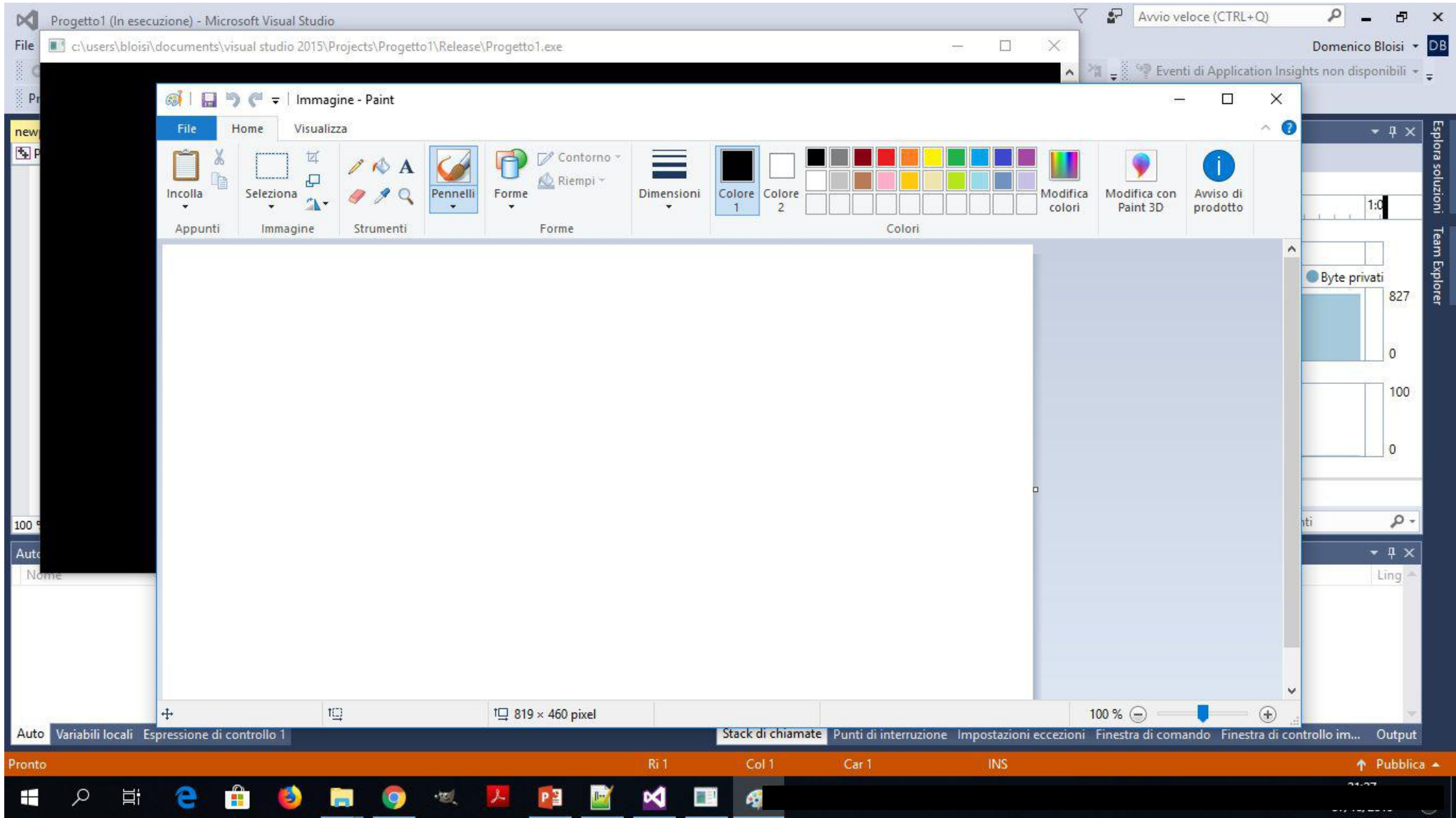




createProcess() Windows

```
#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

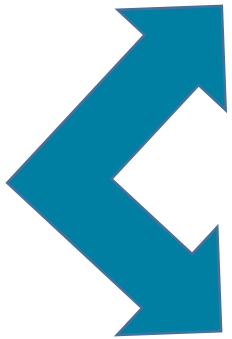
Interprocess communication (IPC)

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

1. **Condivisione di informazioni**
2. **Velocizzazione del calcolo**
3. **Modularità**

Modelli di comunicazione tra processi

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

Memoria condivisa vs. Scambio di messaggi

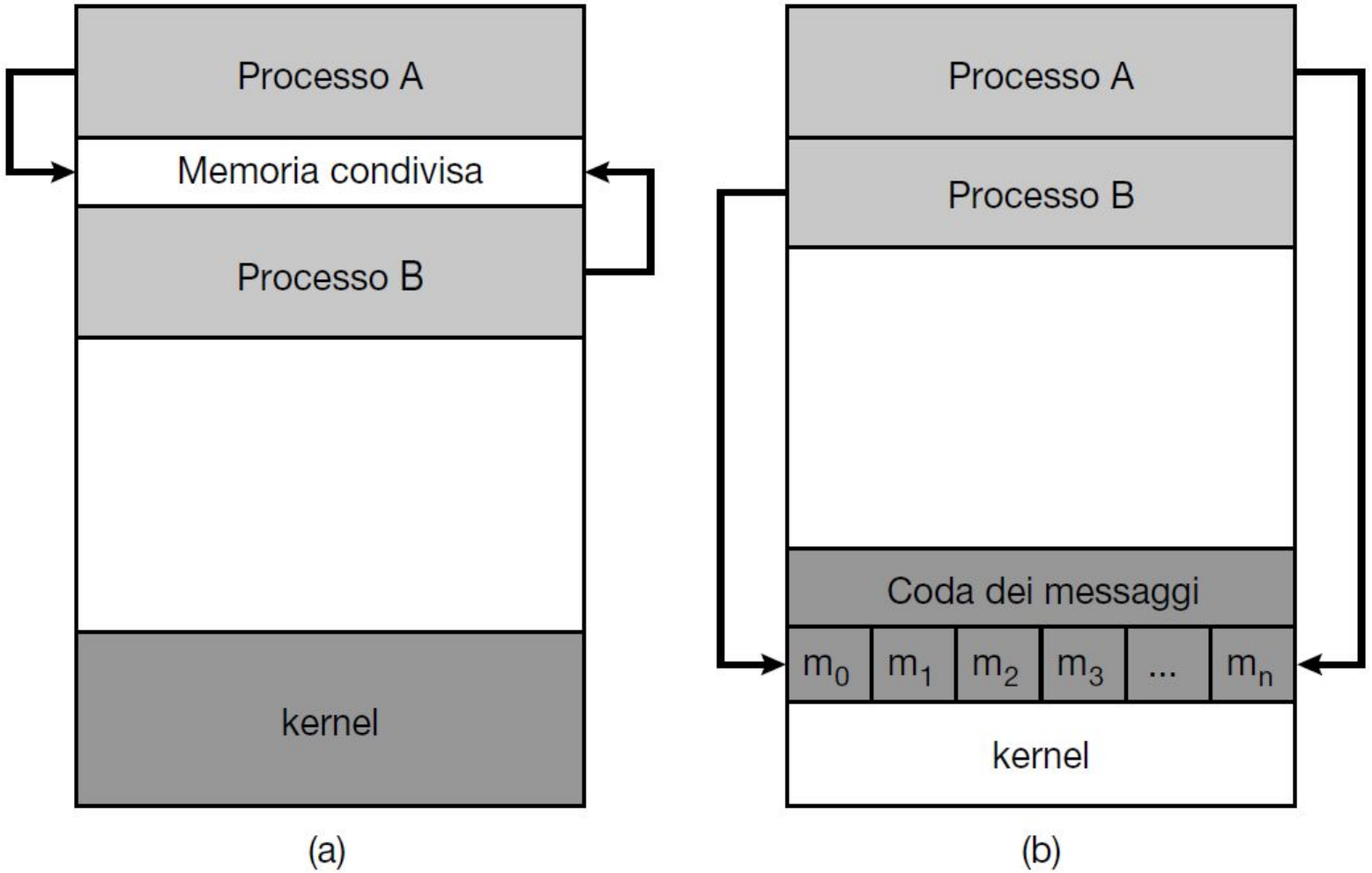
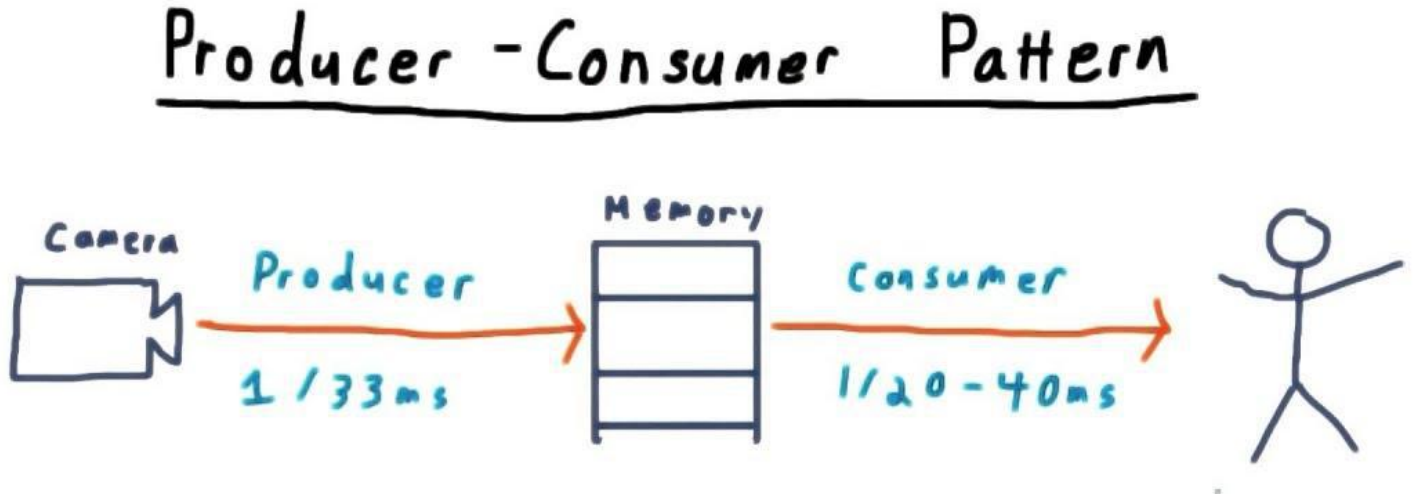


Figura 3.11 Modelli di comunicazione. (a) Memoria condivisa. (b) Scambio di messaggi.

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**



https://youtu.be/VXJSJ6c3ZIs

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'utilizzo 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 bisogno di 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

Il canale di comunicazione

La tipologia di canale di comunicazione scelta permette di ottenere differenti implementazioni per il modello a scambio di messaggi

- 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)**

→ **Invio sincrono**

→ **Invio asincrono**

→ **Ricezione sincrona**

→ **Ricezione asincrona**

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.

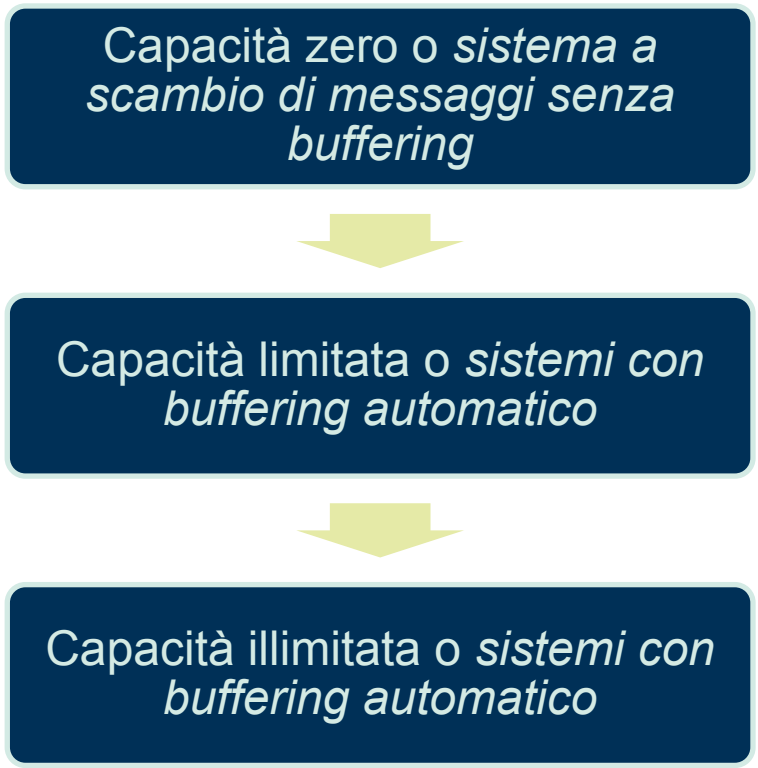
Consumatore con scambio di messaggi

```
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

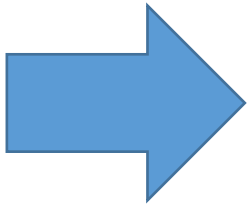
Codice POSIX produttore/consumatore

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 <stlib.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!";
```

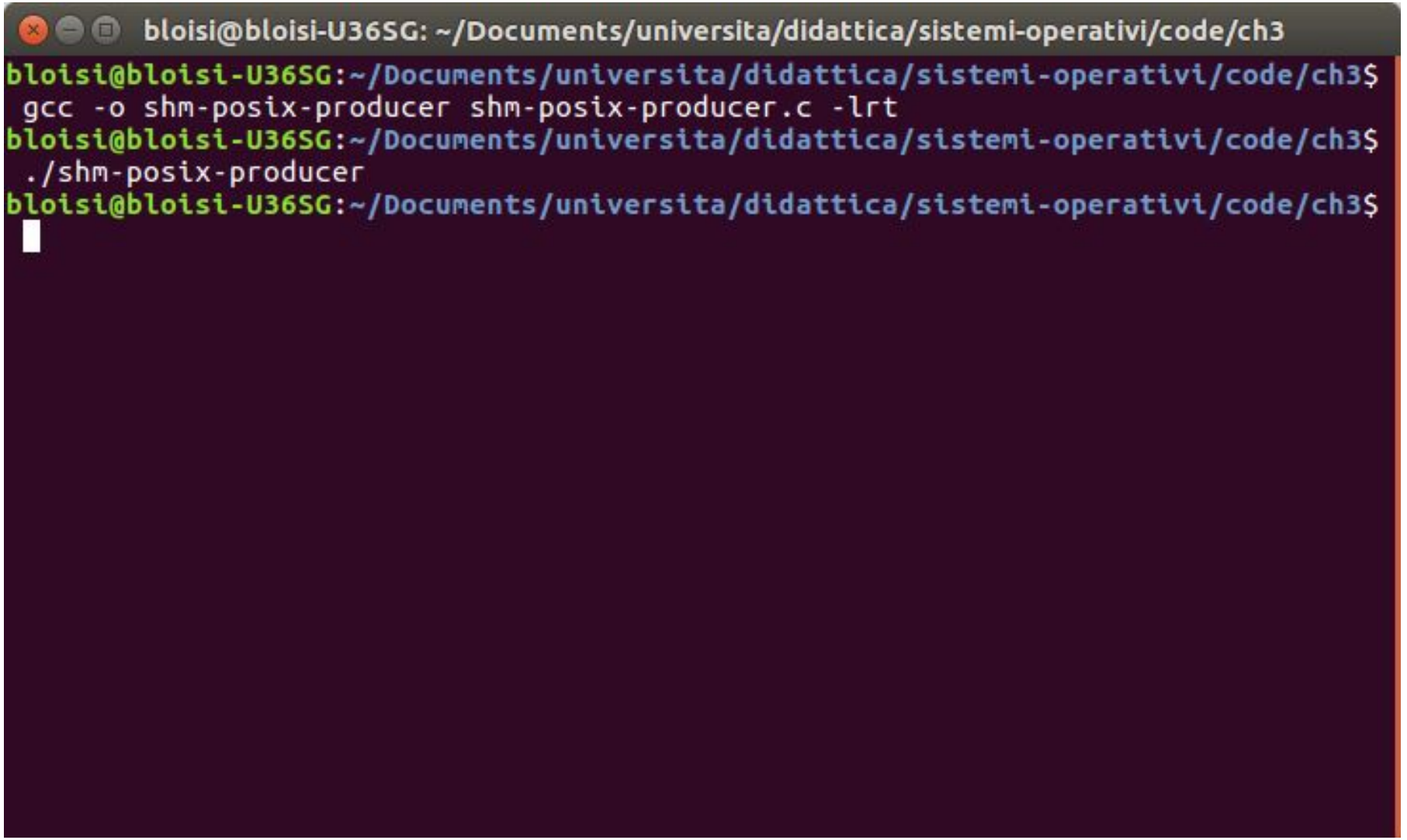


Produttore - memoria condivisa - POSIX

```
/* 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;
}
```

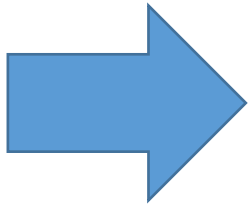
Figura 3.16 Processo produttore che illustra l'API per la memoria condivisa POSIX.

Produttore - memoria condivisa - POSIX



Consumatore - memoria condivisa - POSIX

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

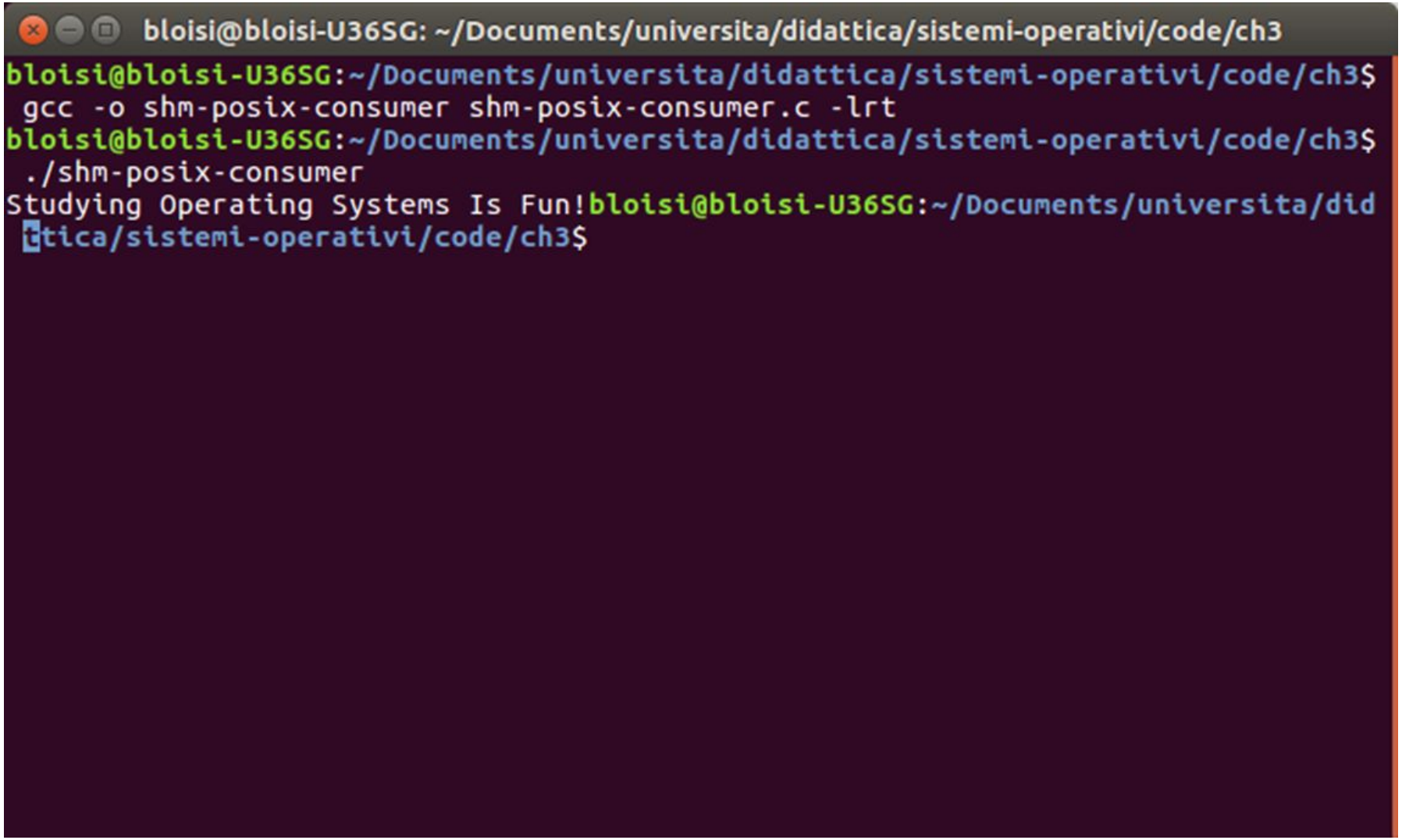


Consumatore - memoria condivisa - POSIX

```
        /* 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;
}
```

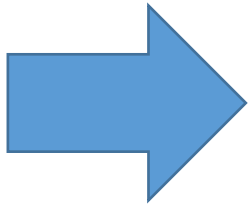
Figura 3.17 Processo consumatore che illustra l'API per la memoria condivisa POSIX.

Consumatore - memoria condivisa - POSIX



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*)

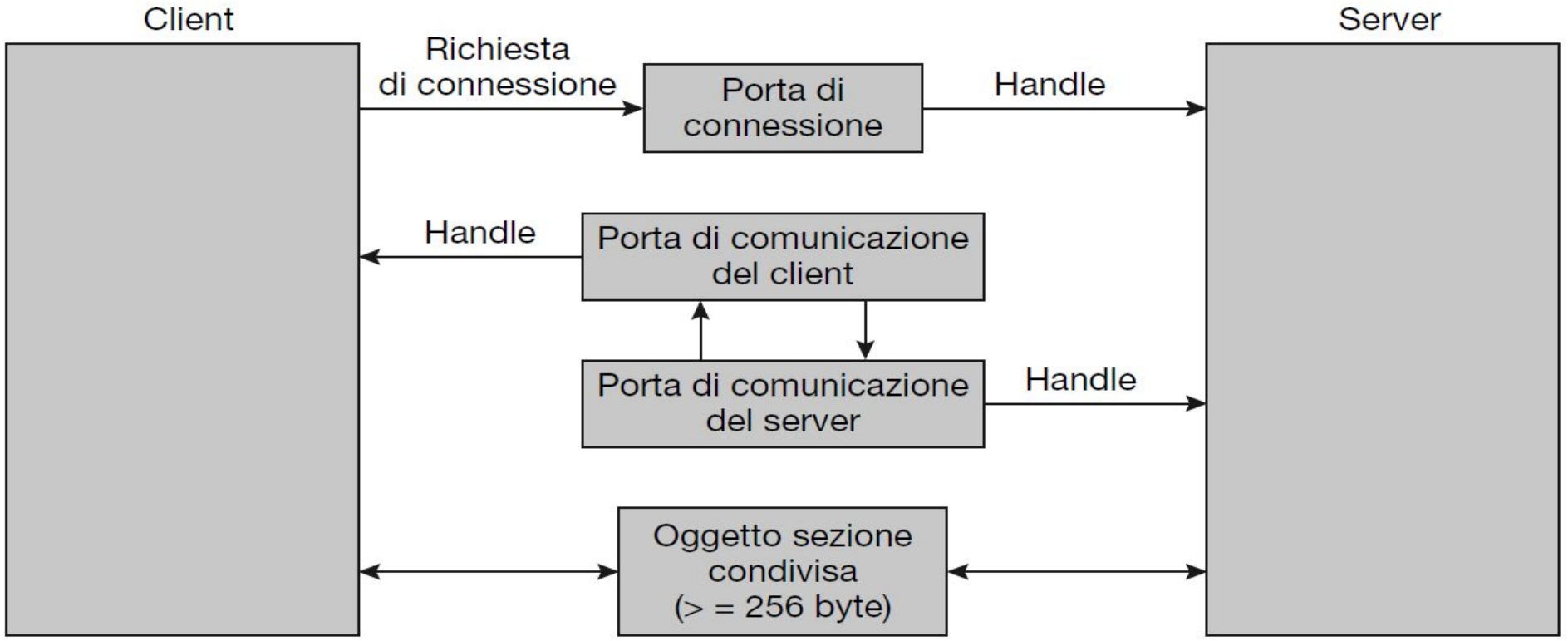


Figura 3.19 Chiamate di procedura locale avanzate in Windows.

Pipe convenzionali

- 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

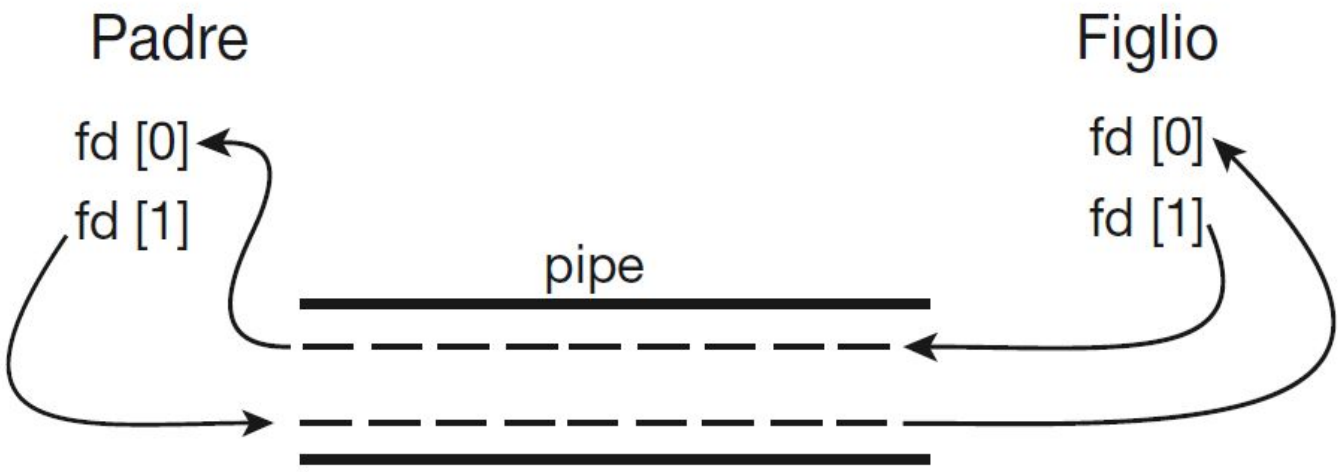
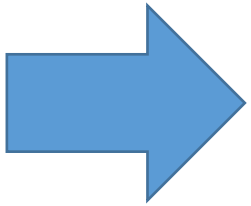


Figura 3.20 Descrittori di file per una pipe convenzionale.

Pipe convenzionali UNIX

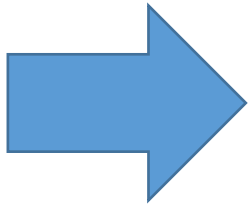
```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#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);
```



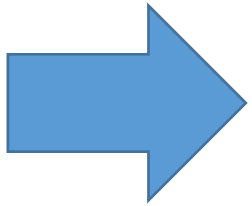
Pipe convenzionali UNIX

```
    /* 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

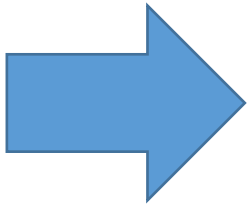
```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#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.

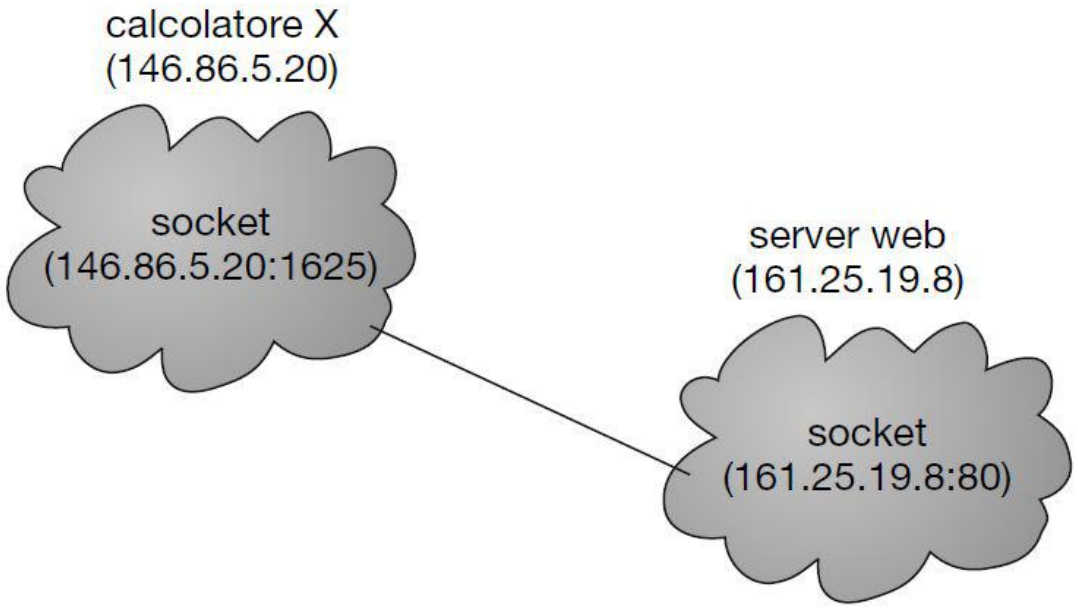


Figura 3.26 Comunicazione tramite socket.

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

Socket in Java

Il linguaggio Java è spesso utilizzato per implementare le socket, poiché Java offre un'interfaccia alle socket più semplice rispetto al C 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

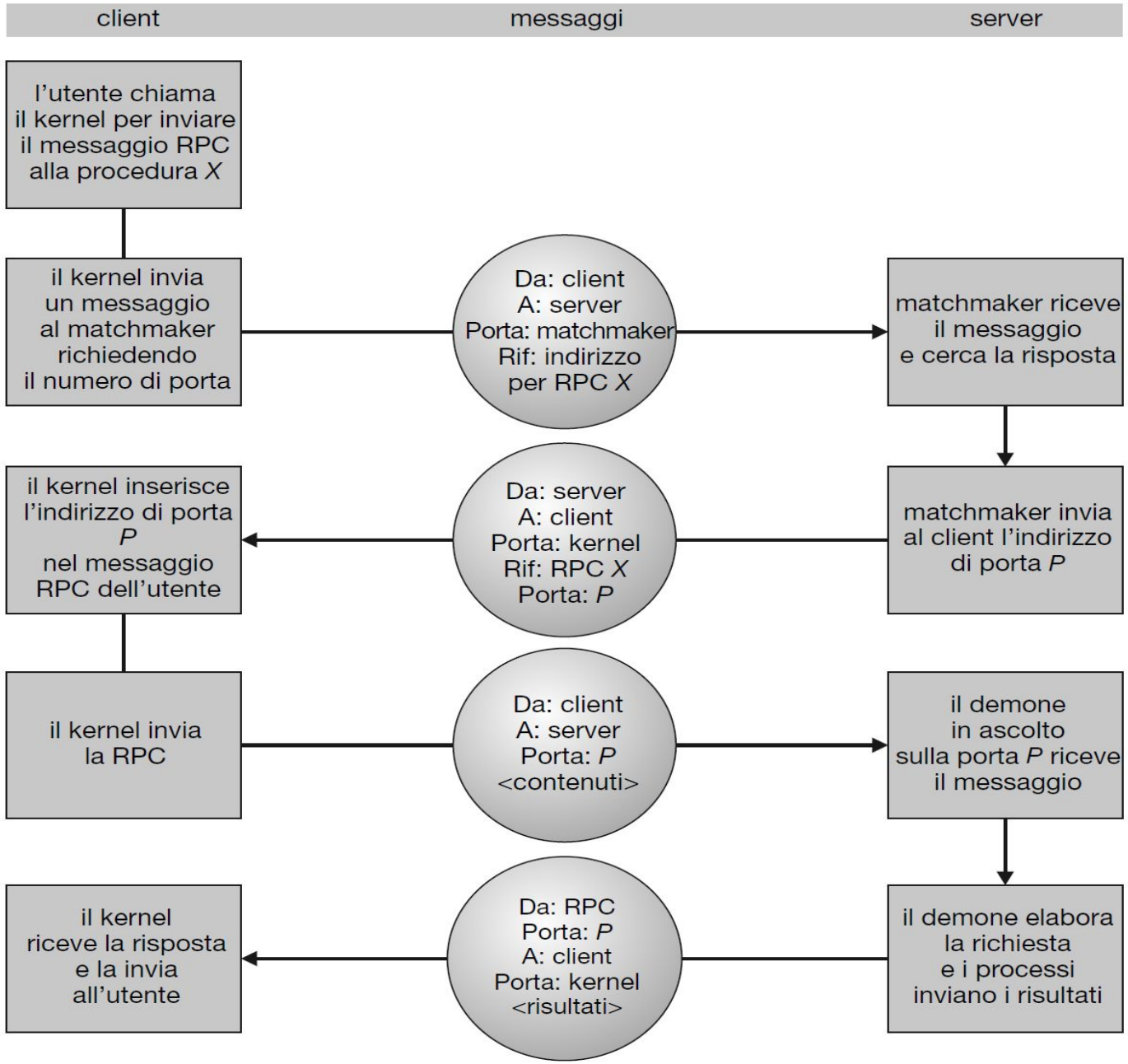
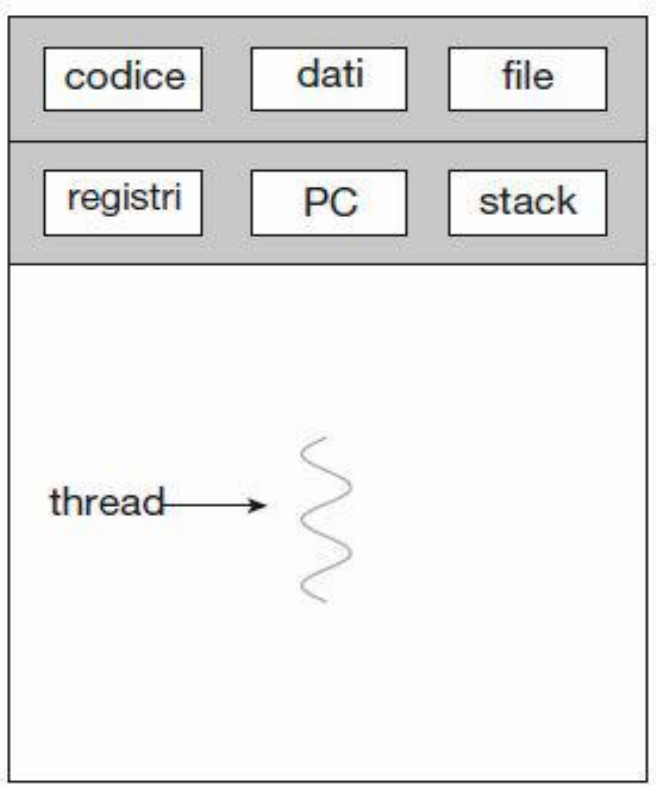


Figura 3.29 Esecuzione di una chiamata di procedura remota (RPC).

Definizione di thread

Un **thread** è l'unità di base d'uso della CPU e comprende

- un identificatore di thread (ID)
- un contatore di programma (PC)
- un insieme di registri
- una pila (*stack*)



processo a singolo thread

Singlethread vs. Multithread process

differenza tra un processo tradizionale, **a singolo thread**, e uno **multithread**.

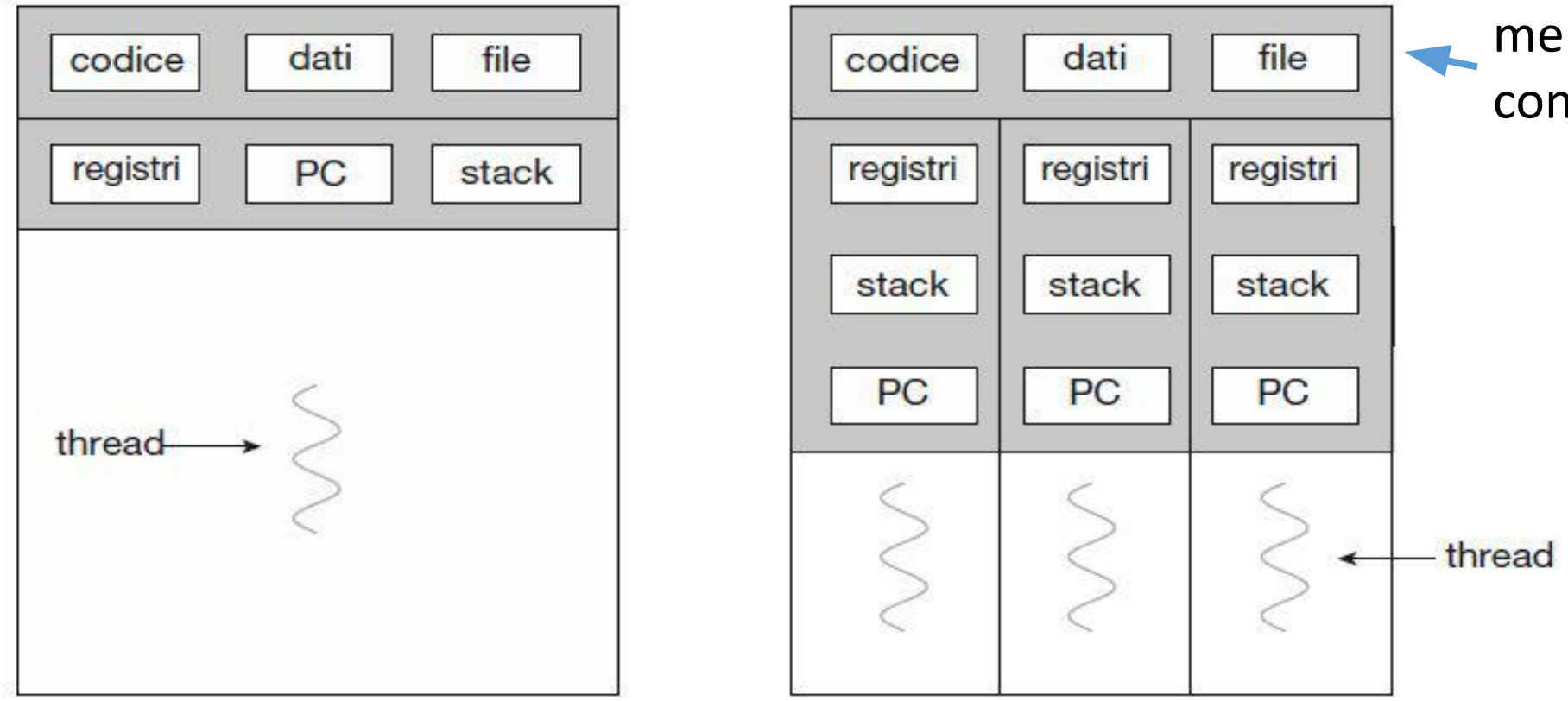


Figura 4.1 Processi a singolo thread e multithread.

Processi mutithread

La maggior parte delle applicazioni per i moderni computer è **multithread**

Esempio web server:

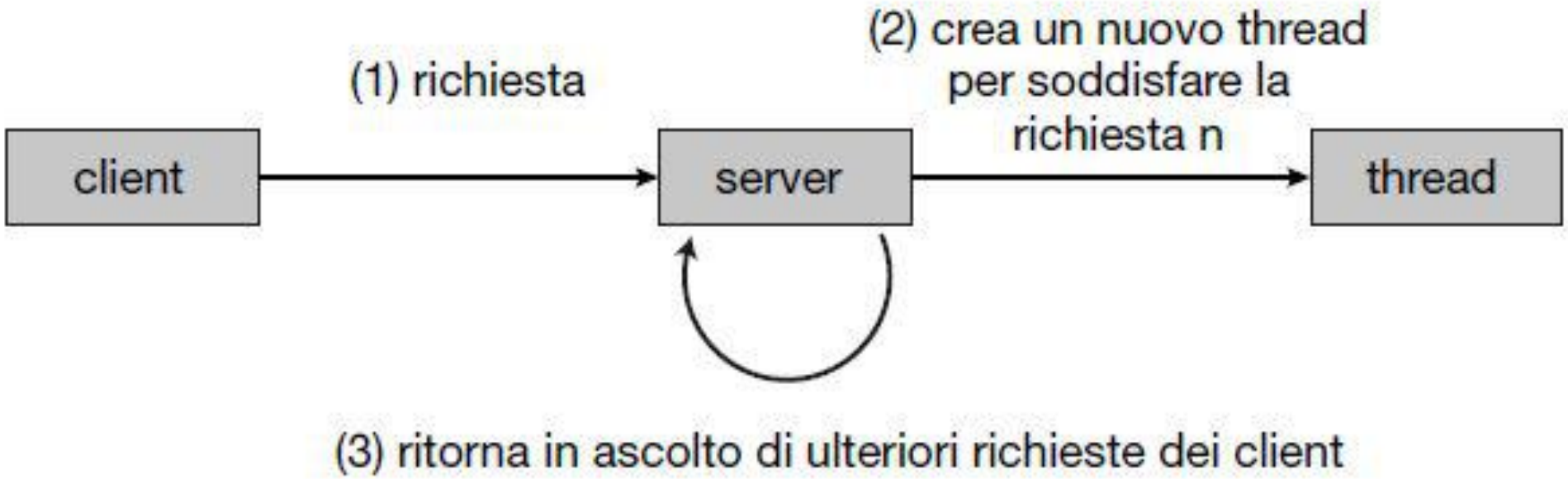


Figura 4.2 Architettura di server multithread.

Programmazione mutithread

VANTAGGI

- Tempo di risposta
- Condivisione delle risorse
- Economia
- Scalabilità

Concorrenza vs. Parallelismo

sistema concorrente → supporta più task permettendo a ciascuno di progredire nell'esecuzione

Figura 4.3 Esecuzione concorrente su un sistema a singolo core.

Esecuzione concorrente su multicore

Su un **sistema multicore** per "esecuzione concorrente" si intende che i thread possono funzionare *in parallelo*, dal momento che il sistema può assegnare thread diversi a ciascun core.

sistema parallelo → può eseguire simultaneamente più di un task

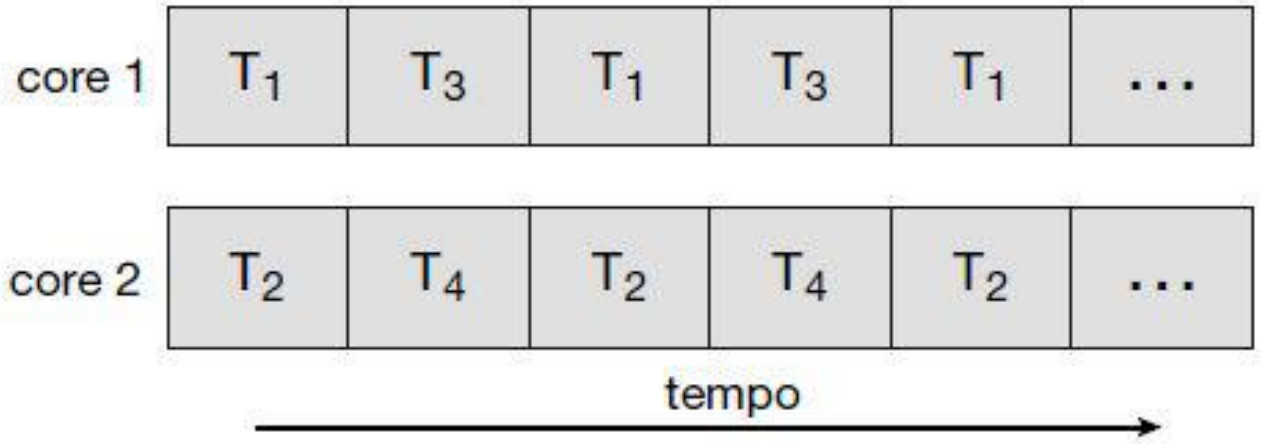
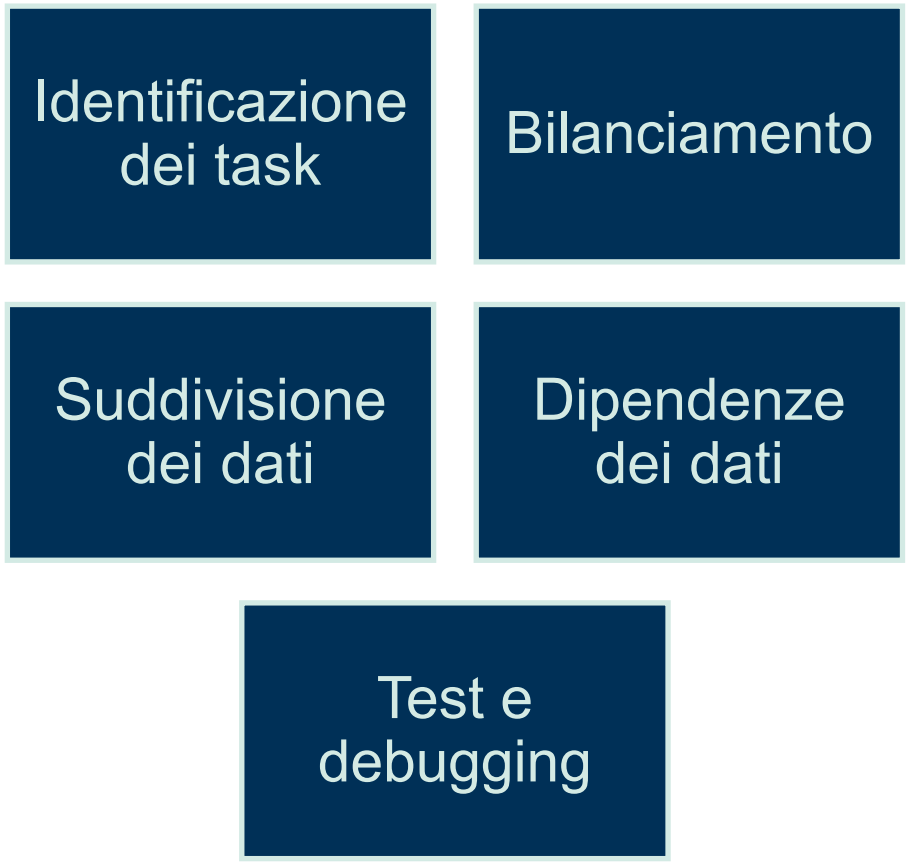


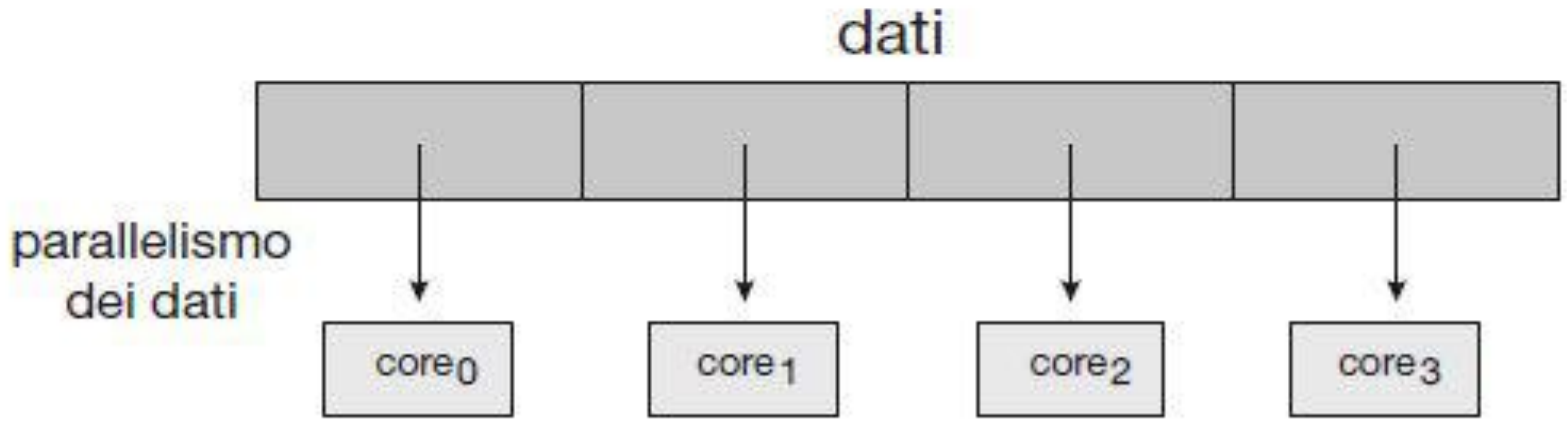
Figura 4.4 Esecuzione parallela su un sistema multicore.

Sfide nella programmazione multicore



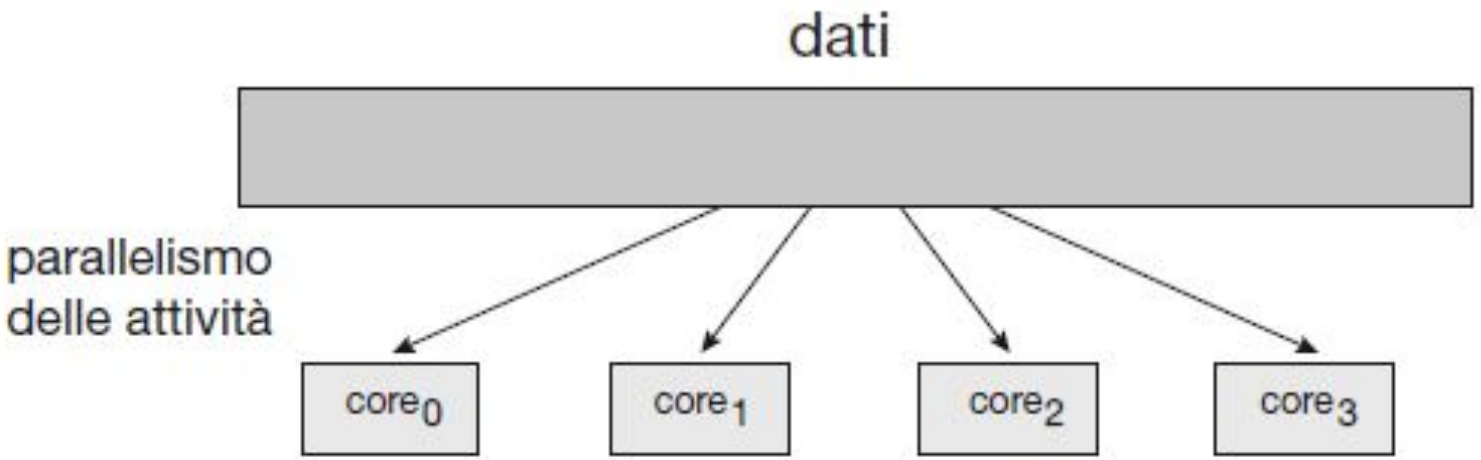
Parallelismo dei dati

parallelismo dei dati → distribuzione di sottoinsiemi dei dati su più core di elaborazione ed esecuzione della stessa operazione su ogni core

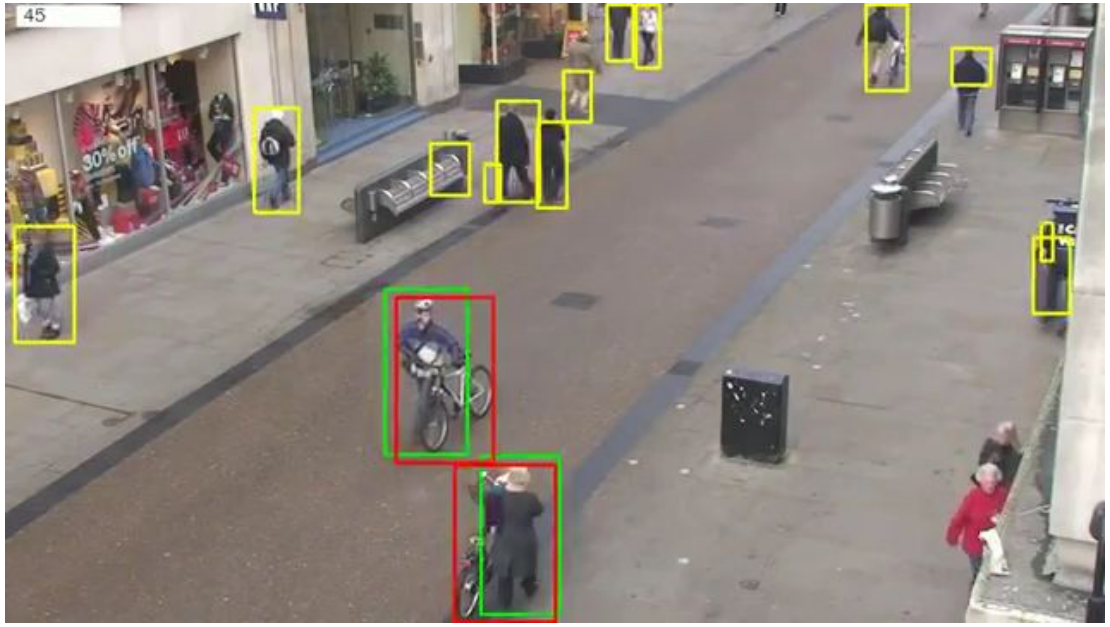


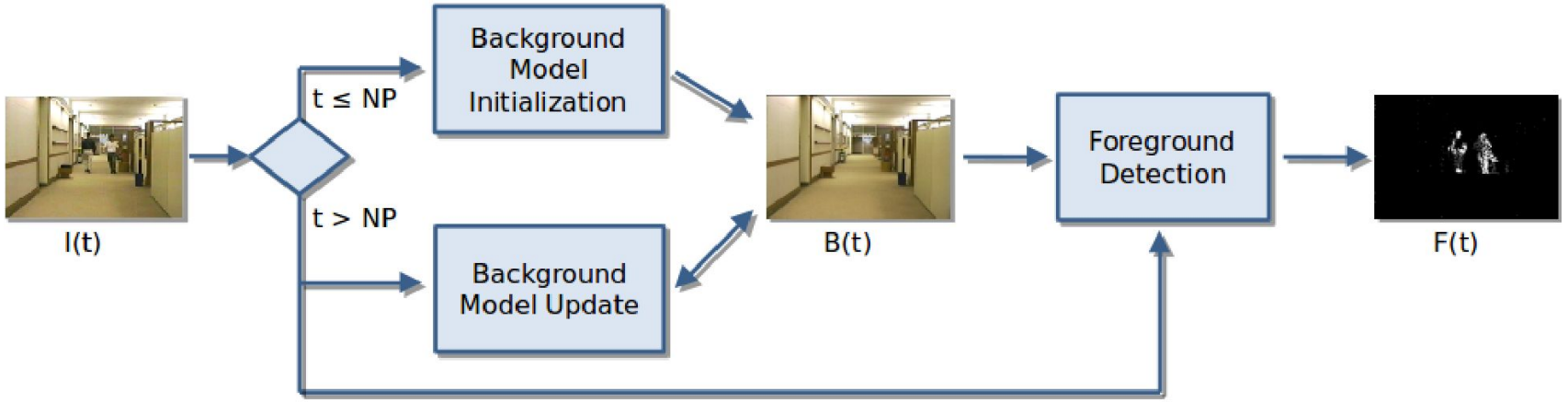
Parallelismo delle attività

thread)



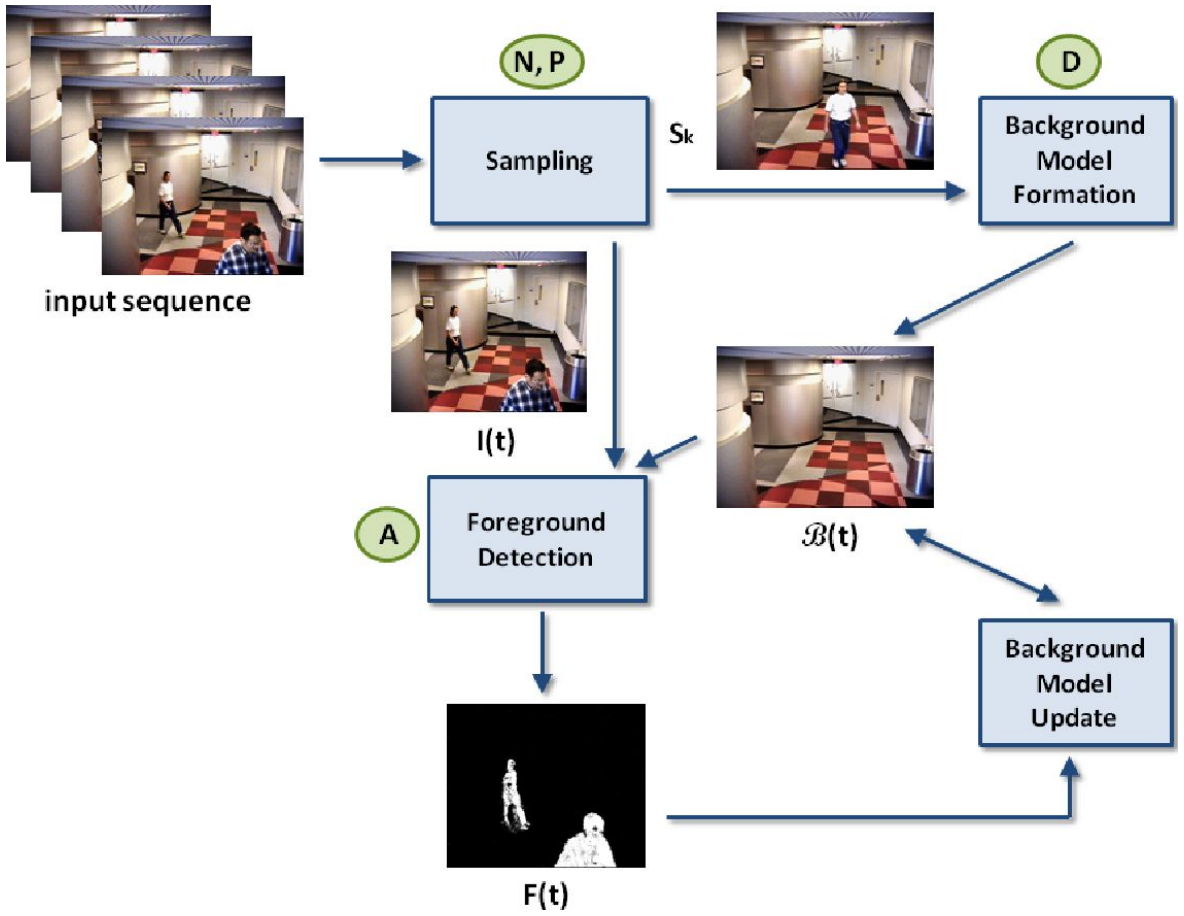
Esempio background subtraction





Esempio background subtraction

Esecuzione con **singolo processo**



Esempio background subtraction

Esecuzione **multicore**

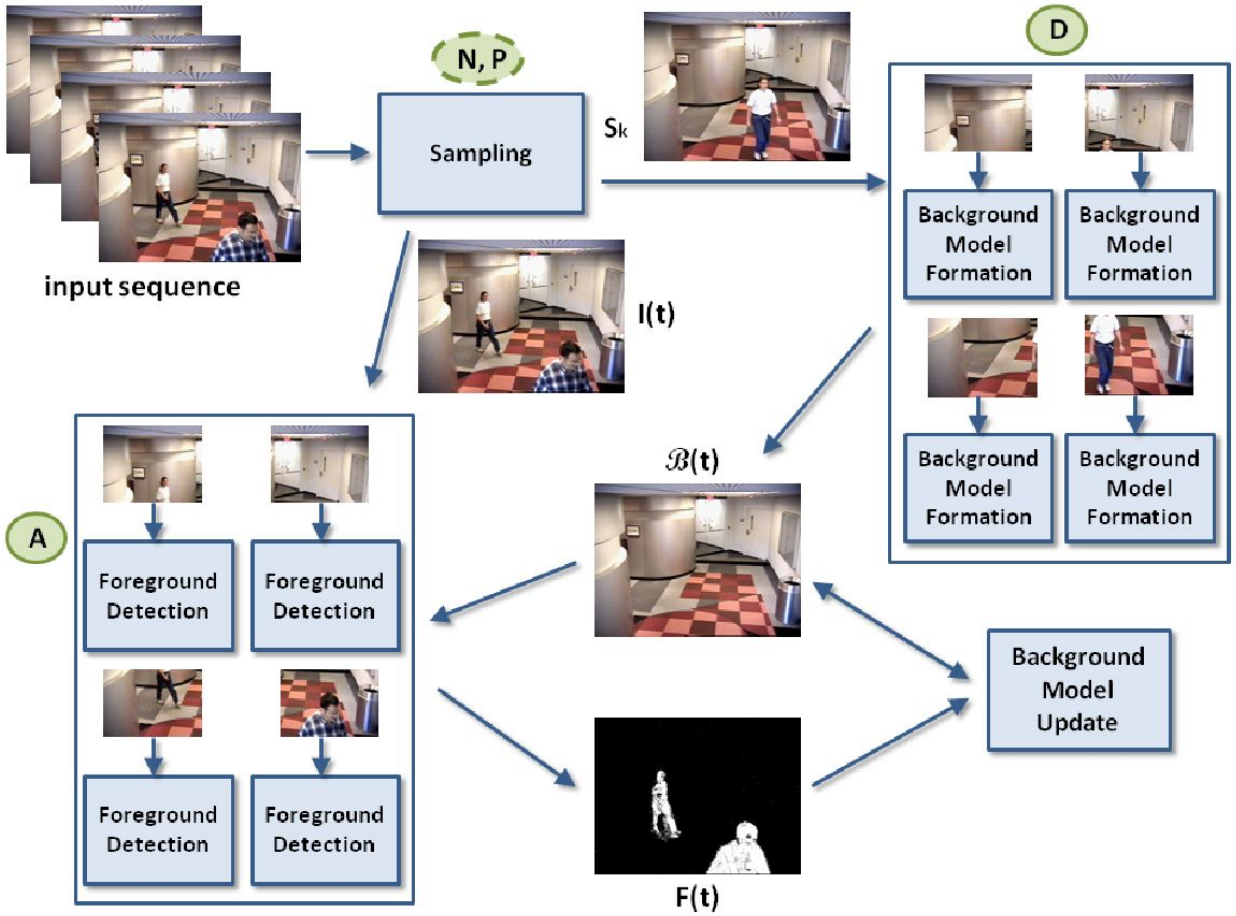


Table 3. Computational load in terms of FPS on different computer display standards for KNN (with TBB), MOG2 (with TBB), IMBS (mono-thread), and IMBS-MT (multi-thread).

Video Standard	Frame Size	KNN	MOG2	IMBS	IMBS-MT
Video CD	352×240	144.702	80.0249	35.13	150.32
HD	1360×768	18.5798	6.2748	12.49	25.31
HD+	1600×900	13.7526	3.8475	8.42	20.72
Full HD	1920×1080	9.7816	2.9693	3.45	13.52

Gestione dei thread

thread a livello utente → gestiti sopra il livello del kernel e senza il suo supporto

thread a livello kernel → gestiti direttamente dal sistema operativo

Modelli di supporto al mutithreading

Modello da molti a uno

Modello da uno a uno

Modello da molti a molti

Modello a due livelli

Modello da molti a uno

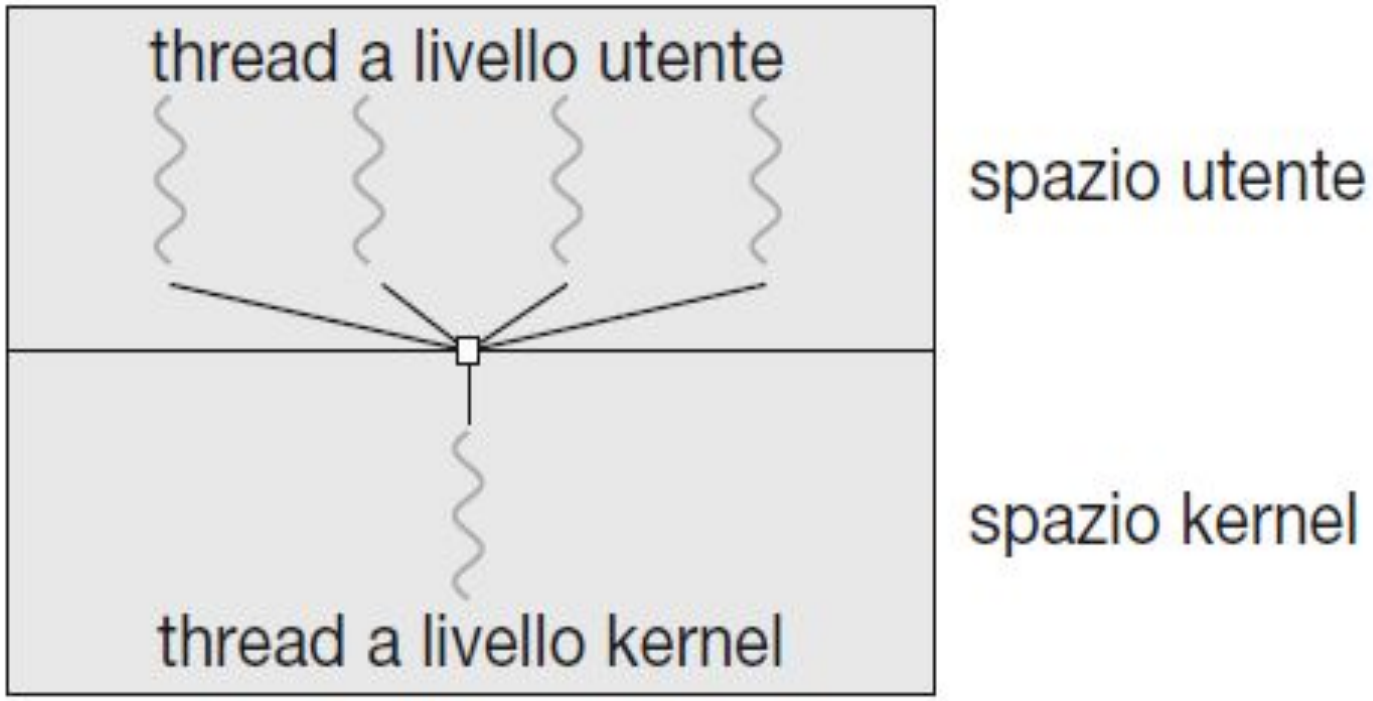


Figura 4.7 Modello da molti a uno.

Modello da uno a uno

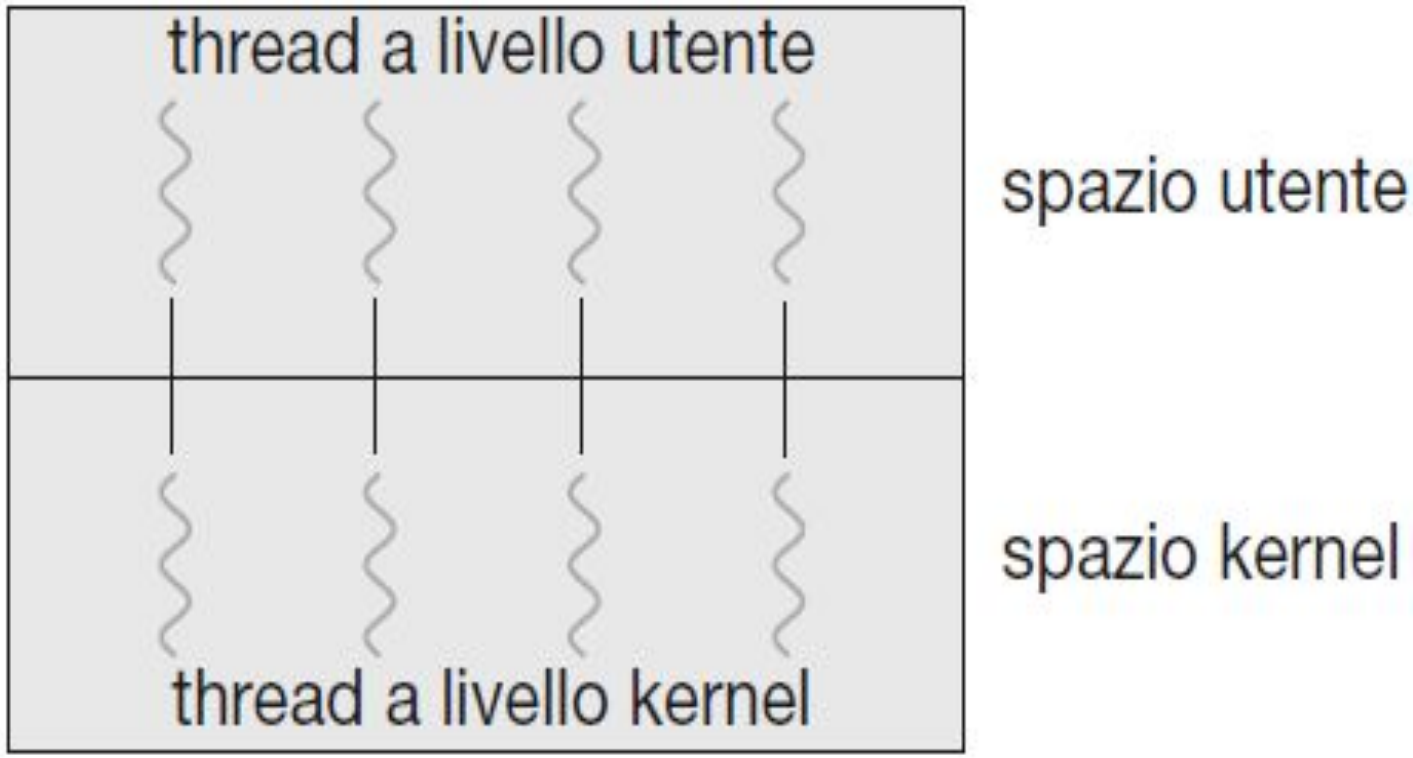


Figura 4.8 Modello da uno a uno.

Modello da molti a molti

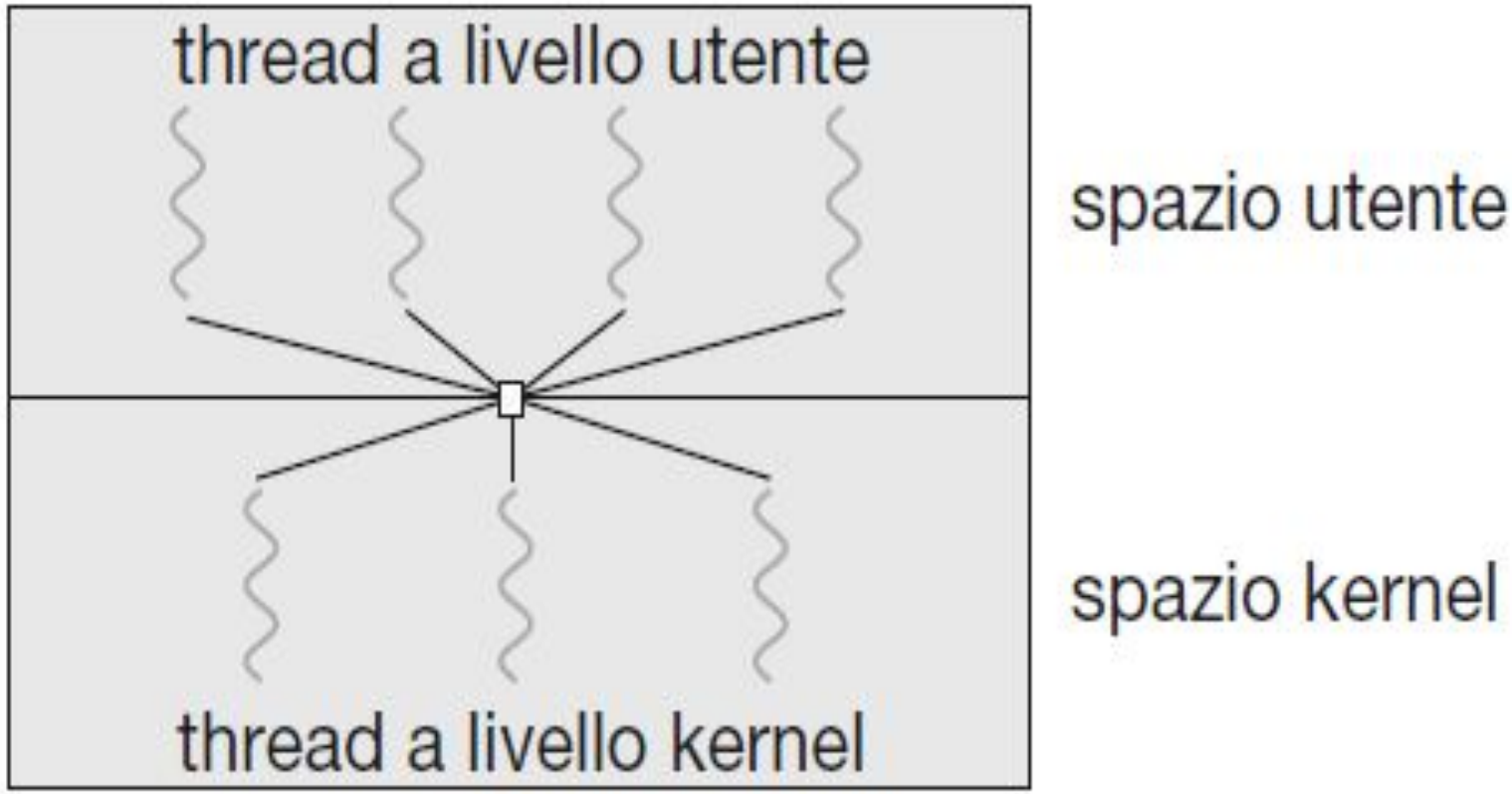


Figura 4.9 Modello da molti a molti.

Modello a due livelli

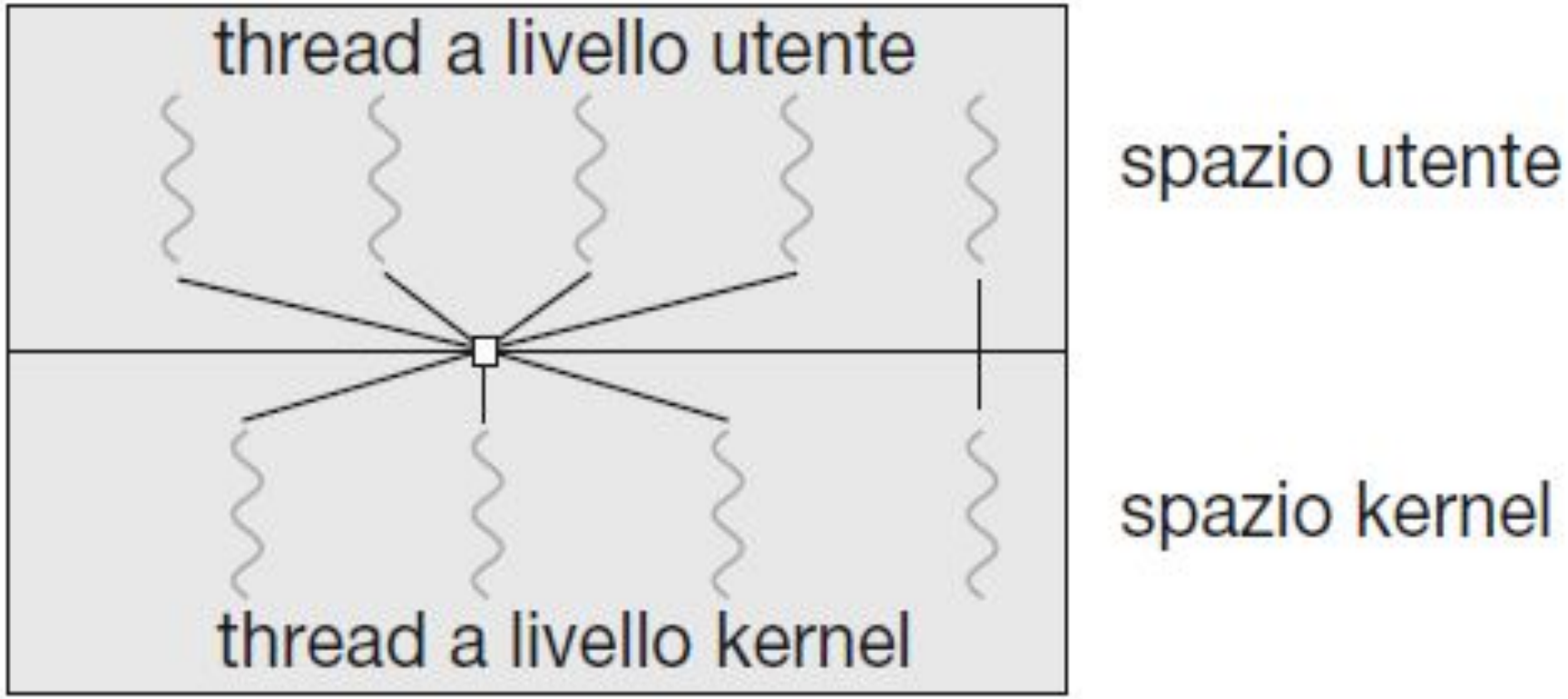


Figura 4.10 Modello a due livelli.

Librerie dei thread

Una **libreria dei thread** fornisce al programmatore una API per la creazione e la gestione dei thread.

Esempi:

- Ptreads POSIX
- Windows
- Java

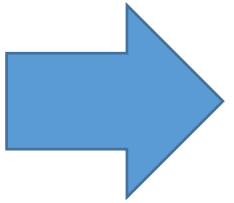
Pthreads

Col termine **Pthreads** ci si riferisce allo standard POSIX (IEEE 1003.1c) che definisce una API per la creazione e la sincronizzazione dei thread.

Esempio API Pthreads

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

#include <stdlib.h>

int sum; /* questo dato è condiviso dai thread */
void *runner(void *param); /* i thread chiamano questa funzione */

int main(int argc, char *argv[])
{
    pthread_t tid; /* identificatore del thread */
    pthread_attr_t attr; /* insieme di attributi del thread */
```

$$sum = \sum_{i=1}^{N} i$$



Esempio API pthread

```
    /* imposta gli attributi predefiniti del thread */
    pthread_attr_init(&attr);
    /* crea il thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* attende la terminazione del thread */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* Il thread viene eseguito in questa funzione */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
      sum += i;

    pthread_exit(0);
}
```

Figura 4.11 Programma multithread in linguaggio C che impiega la API Pthreads.

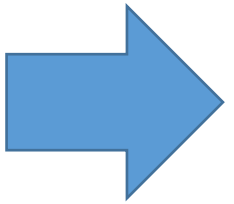
Esempio API pthread

```
#define NUM_THREADS 10

/* array di thread da unire */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Figura 4.12 Codice Pthread per effettuare il join di 10 thread.

Thread in Windows

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* il dato è condiviso tra i thread */

/* il thread viene eseguito in questa funzione separata */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
     Sum += i;
    return 0;
}
```



Thread in Windows

```
int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi (argv [1])

   /* crea il thread */
   ThreadHandle = CreateThread(
    NULL, /* attributi di sicurezza di default */
    0, /* dimensione di default dello stack */
    Summation, /* funzione del thread */
    &Param, /* parametri alla funzione del thread */
    0, /* flag di crezione di default */
    &ThreadId); /* restituisce l'identificatore del thread */

   /* adesso aspetta la fine del thread */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* chiude l'handle del thread */
   CloseHandle(ThreadHandle);

   printf("sum = %d\n",Sum);
}
```

Figura 4.13 Programma multithread in C con l'utilizzo dell'API Windows.

Thread in Java – Java Executor

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
  private int upper;
  public Summation(int upper) {
    this.upper = upper;
                                                    }

  /* Il thread viene eseguito in questo metodo */
  public Integer call() {
    int sum = 0;
    for (int i = 1; i <= upper; i++)
      sum += i;

    return new Integer(sum);
  }
}
```

Thread in Java – Java Executor

```
public class Driver
{
  public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);

    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));

    try {
      System.out.println("sum = " + result.get());
    } catch (InterruptedException | ExecutionException ie) { }
  }
}
```

Figura 4.14 Esempio di utilizzo dell'API Java Executor.

Threading implicito

threading implicito → trasferimento della creazione e della gestione del threading *dagli sviluppatori di applicazioni ai compilatori e alle librerie di runtime*

Threading implicito

Approcci alternativi per la progettazione di programmi multithread in grado di sfruttare i processori multicore attraverso il threading implicito

gruppi di thread *(thread pool)* → creare un certo numero di thread alla creazione del processo e organizzarlo in un gruppo (*pool*) in cui attenda di eseguire il lavoro che gli sarà richiesto

Thread pool in Java

```
import java.util.concurrent.*;

public class ThreadPoolExample
{
public static void main(String[] args) {
  int numTasks = Integer.parseInt(args[0].trim());

  /* Crea il gruppo di thread */
  ExecutorService pool = Executors.newCachedThreadPool();

  /* Esegue ogni attività con un thread distinto del gruppo */
  for (int i = 0; i < numTasks; i++)
    pool.execute(new Task());

  /* Termina il gruppo quando tutti i thread hanno completato
     l'attività */
  pool.shutdown();
}
```

Figura 4.15 Creazione di un gruppo di thread in Java.

fork – join

Nel metodo fork-join il *thread padre* crea (*fork*) uno o più *thread figli*, attende che i figli terminino e si uniscano a esso (*join)* e a quel punto può recuperare e combinare i risultati ottenuti dai figli.

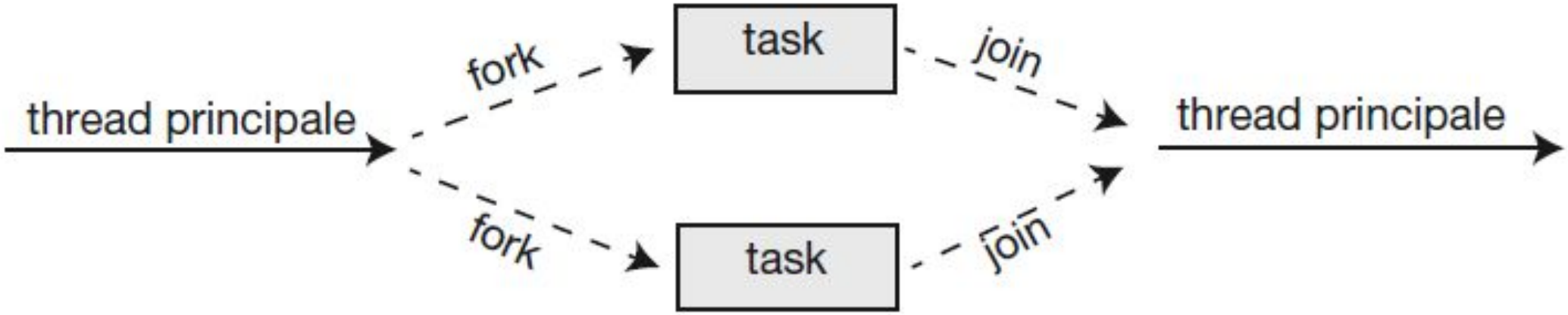


Figura 4.16 Parallelismo fork-join.

fork – join in Java

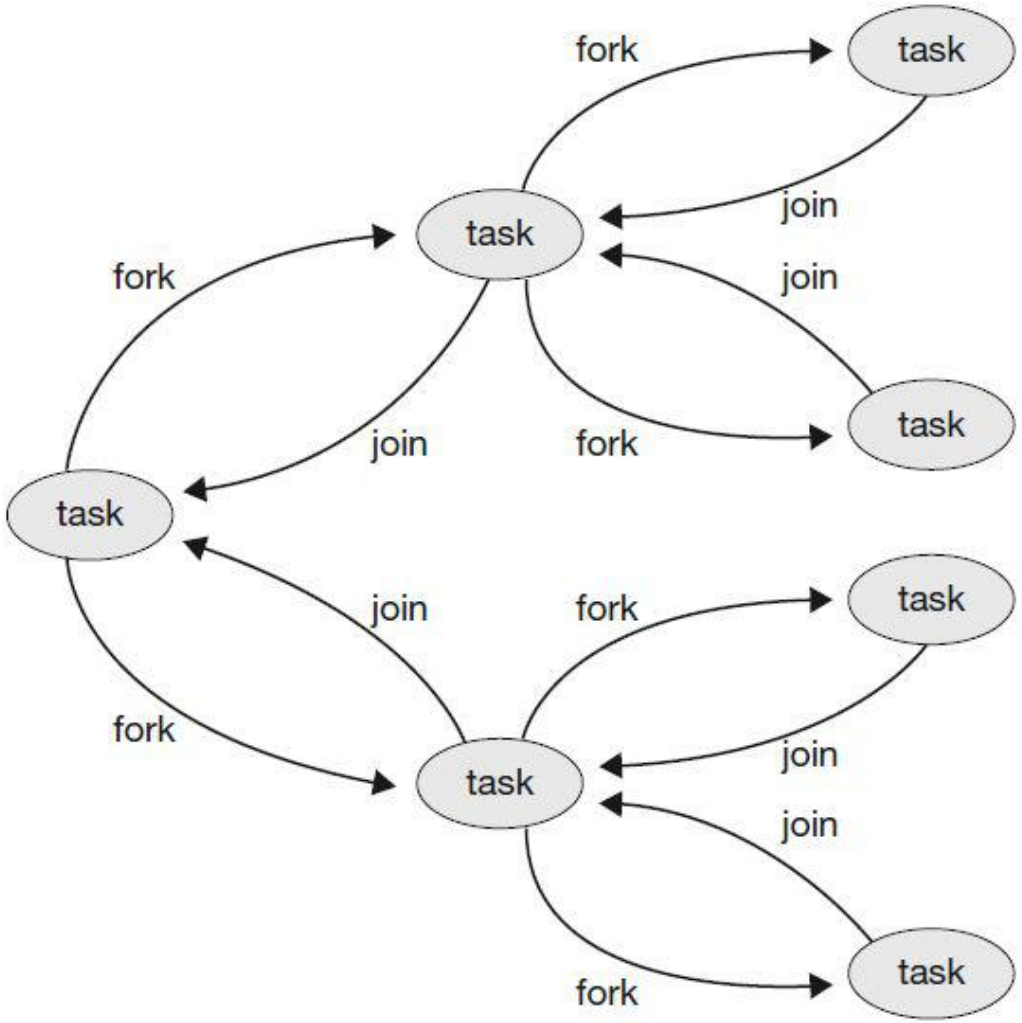
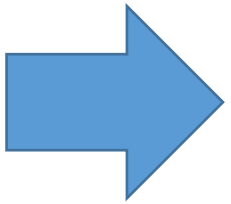


Figura 4.17 Fork-join in Java.

fork – join in Java

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
  static final int THRESHOLD = 1000;

  private int begin;
  private int end;
  private int[] array;

  public SumTask(int begin, int end, int[] array) {
    this.begin = begin;
    this.end = end;
    this.array = array;
  }
```



fork – join in Java

```
protected Integer compute() {
  if (end - begin < THRESHOLD) {
    int sum = 0;
    for (int i = begin; i <= end; i++)
      sum += array[i];
    return sum;
  }
  else {
    int mid = (begin + end) / 2;

    SumTask leftTask = new SumTask(begin, mid, array);
    SumTask rightTask = new SumTask(mid + 1, end, array);

    leftTask.fork();
    rightTask.fork();

    return rightTask.join() + leftTask.join();
  }
 }
}
```

Figura 4.18 Esempio di computazione fork-join in Java.

fork – join in Java

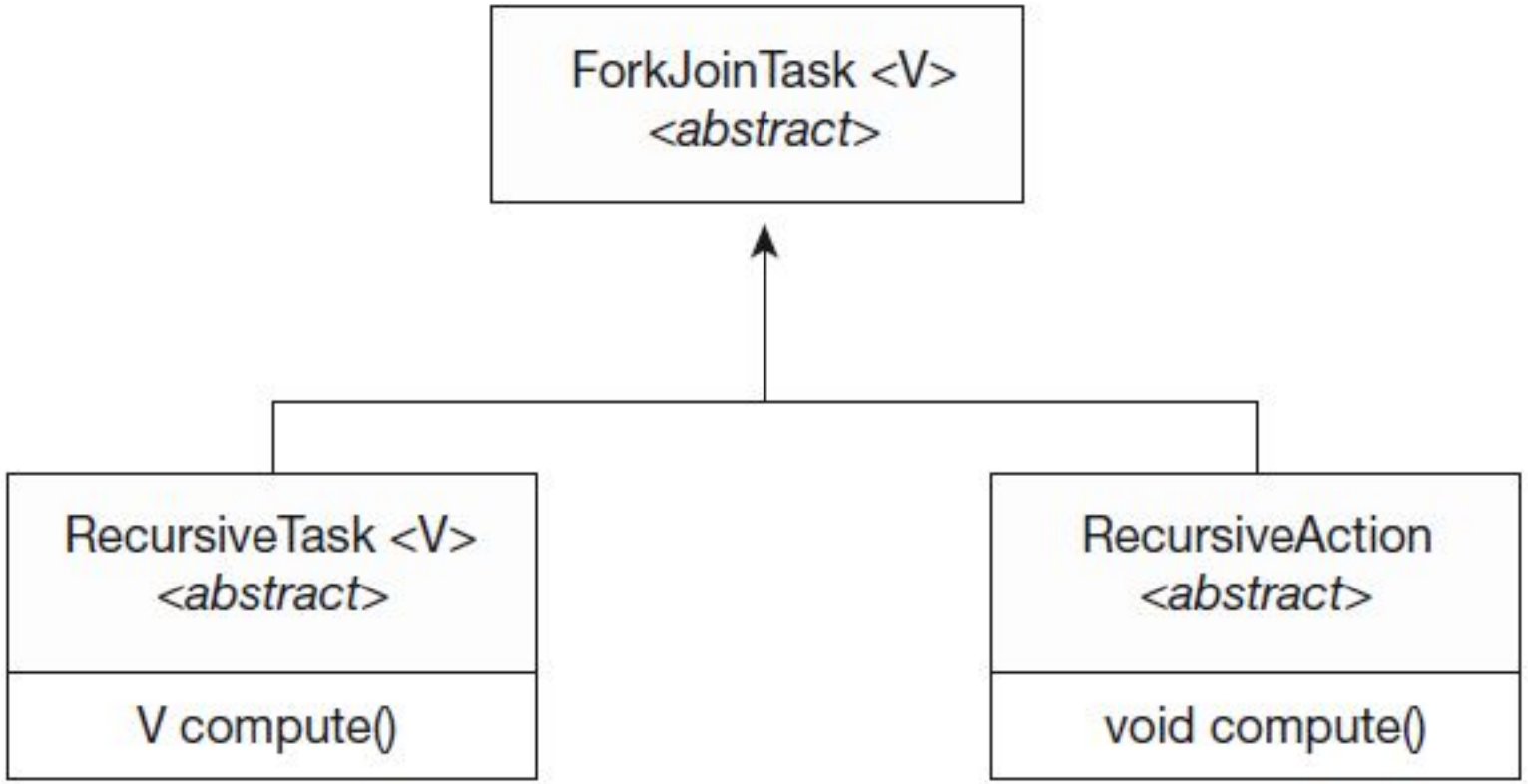


Figura 4.19 Diagramma UML della classe Java per il fork-join.

OpenMP / GCD / TBB

- **OpenMP** è un insieme di direttive del compilatore e una API per programmi scritti in C, C++ o FORTRAN che fornisce il supporto per la programmazione parallela in ambienti a memoria condivisa.

- **Grand Central Dispatch** (GCD) è una tecnologia per i sistemi operativi macOS e iOS di Apple. È una combinazione di estensioni del linguaggio C, una API e una libreria di runtime che permette agli sviluppatori di applicazioni di individuare sezioni di codice da eseguire in parallelo.

- **Intel Threading Building Blocks** (TBB) *è* una libreria di template che supporta la progettazione di applicazioni parallele in C++ e non richiede alcun compilatore o supporto linguistico speciale.

Programmazione mutithread

PROBLEMATICHE

- Chiamate di sistema fork() ed exec()
- Gestione dei segnali
- Cancellazione dei thread
- Dati locali dei thread
- Attivazione dello scheduler

Cancellazione di thread

I thread possono essere terminati utilizzando la **cancellazione asincrona** o la **cancellazione differita**.

La cancellazione asincrona interrompe immediatamente un thread, anche se si trova a metà di un aggiornamento.

La cancellazione differita informa un thread che dovrebbe terminare la sua esecuzione, ma gli consente di terminare in modo ordinato.
Nella maggior parte dei casi la cancellazione differita è preferibile alla terminazione asincrona.

Cancellazione in ptheads

pthreads permette di disabilitare o abilitare la **cancellazione dei thread**

Modalità	Stato	Tipo
Off	Disabilitato	-
Differita	Abilitato	Differito
Asincrona	Abilitato	Asincrono

Windows vs. Linux

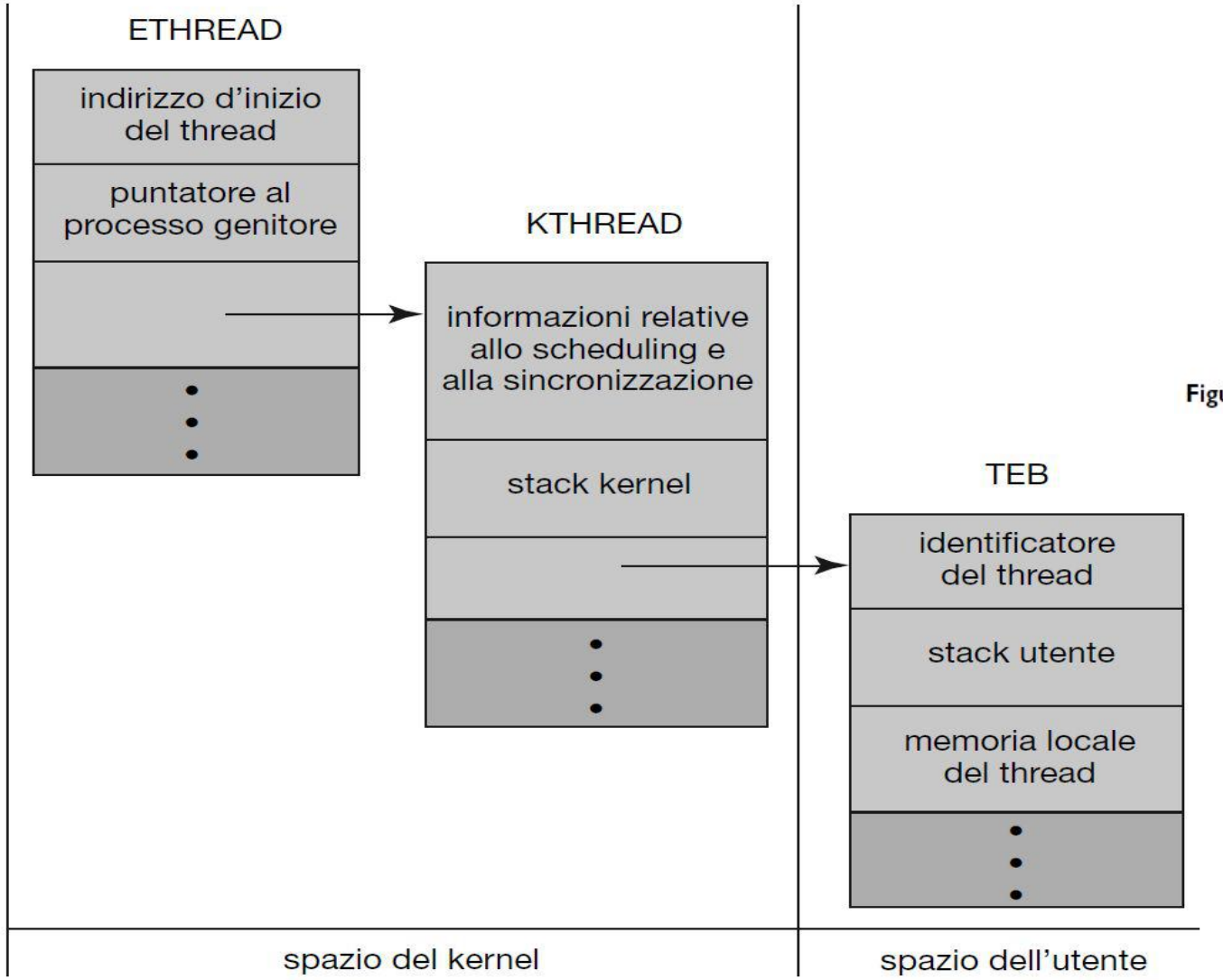


Figura 4.21 Strutture dati di un thread Windows.

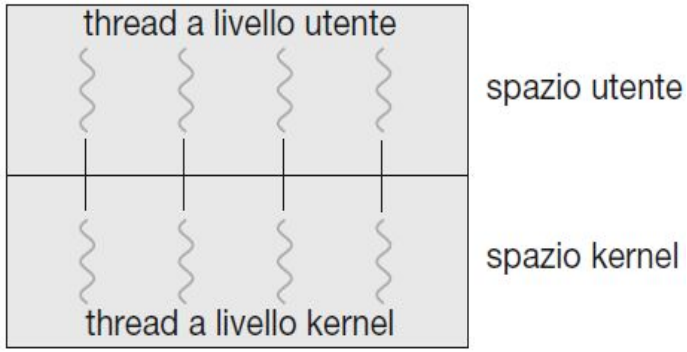


Figura 4.8 Modello da uno a uno.

Windows vs. Linux

A differenza di molti altri sistemi operativi, **Linux** non distingue tra processi e thread, ma si riferisce a ciascuno come un task.

La chiamata di sistema clone() di Linux può essere utilizzata per creare task che si comportano in maniera più simile ai processi o più simile ai thread.

clone()

Quando clone() è invocata, riceve come parametro un insieme di indicatori (*flag*), al fine di stabilire quante e quali risorse del task genitore debbano essere condivise dal task figlio

flag	significato
CLONE_FS	Condivisione delle informazioni sul file system
CLONE_VM	Condivisione dello stesso spazio di memoria
CLONE_SIGHAND	Condivisione dei gestori dei segnali
CLONE_FILES	Condivisione dei file aperti

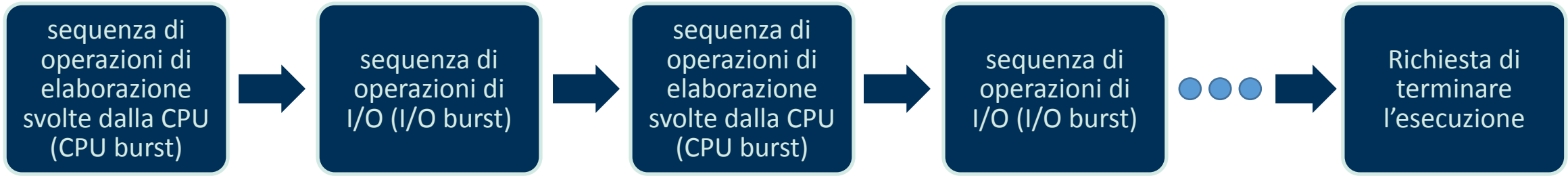
Figura 4.22 Alcuni dei flag passati quando viene invocata la funzione `clone()`.

Scheduling

- Lo scheduling è una funzione fondamentale dei sistemi operativi.
- Si sottopongono a scheduling quasi tutte le risorse di un calcolatore.
- La CPU è una delle risorse principali e il suo scheduling è alla base della progettazione dei sistemi operativi.

Ciclicità delle fasi d'elaborazione e di I/O

L'esecuzione di un processo consiste in un ciclo di elaborazione



Ciclicità delle fasi d'elaborazione e di I/O

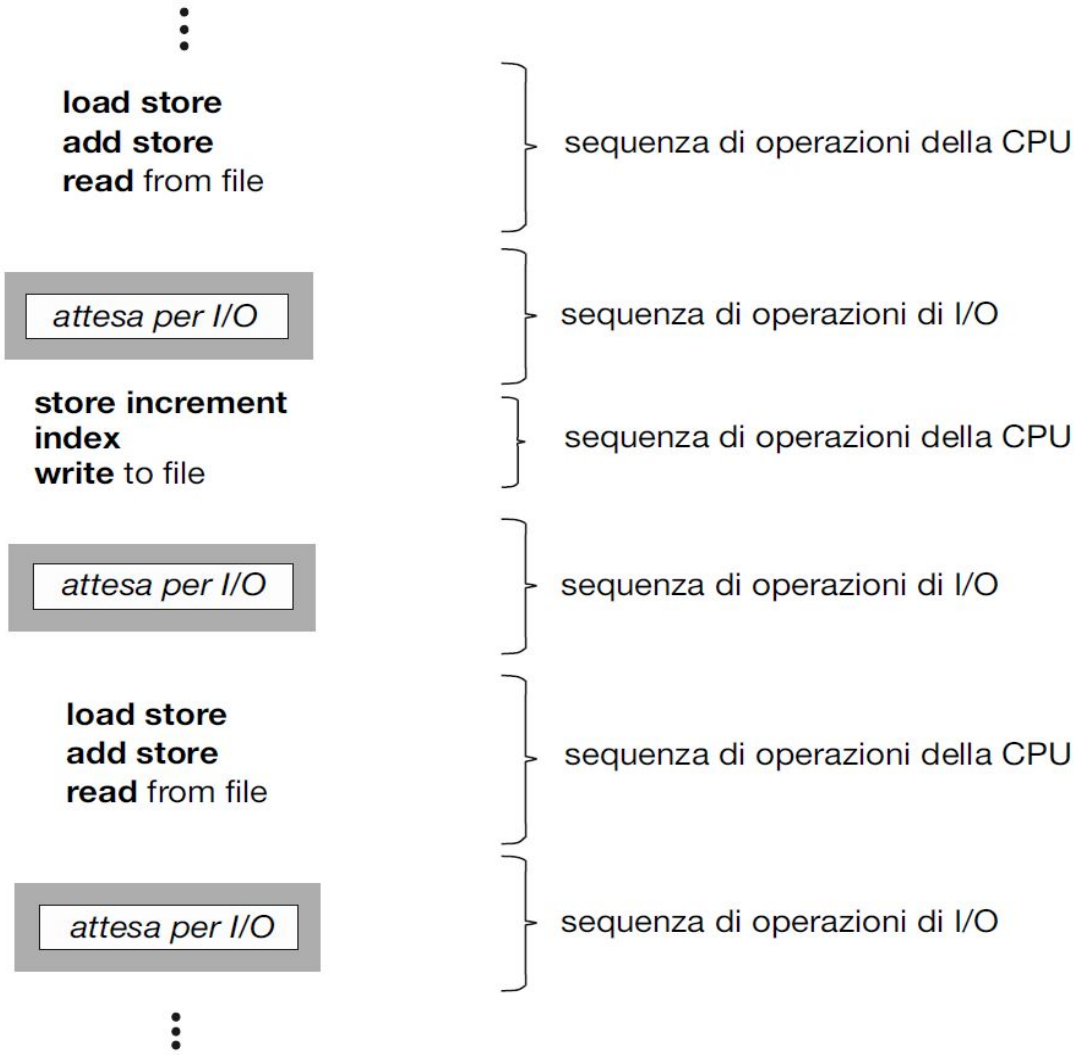
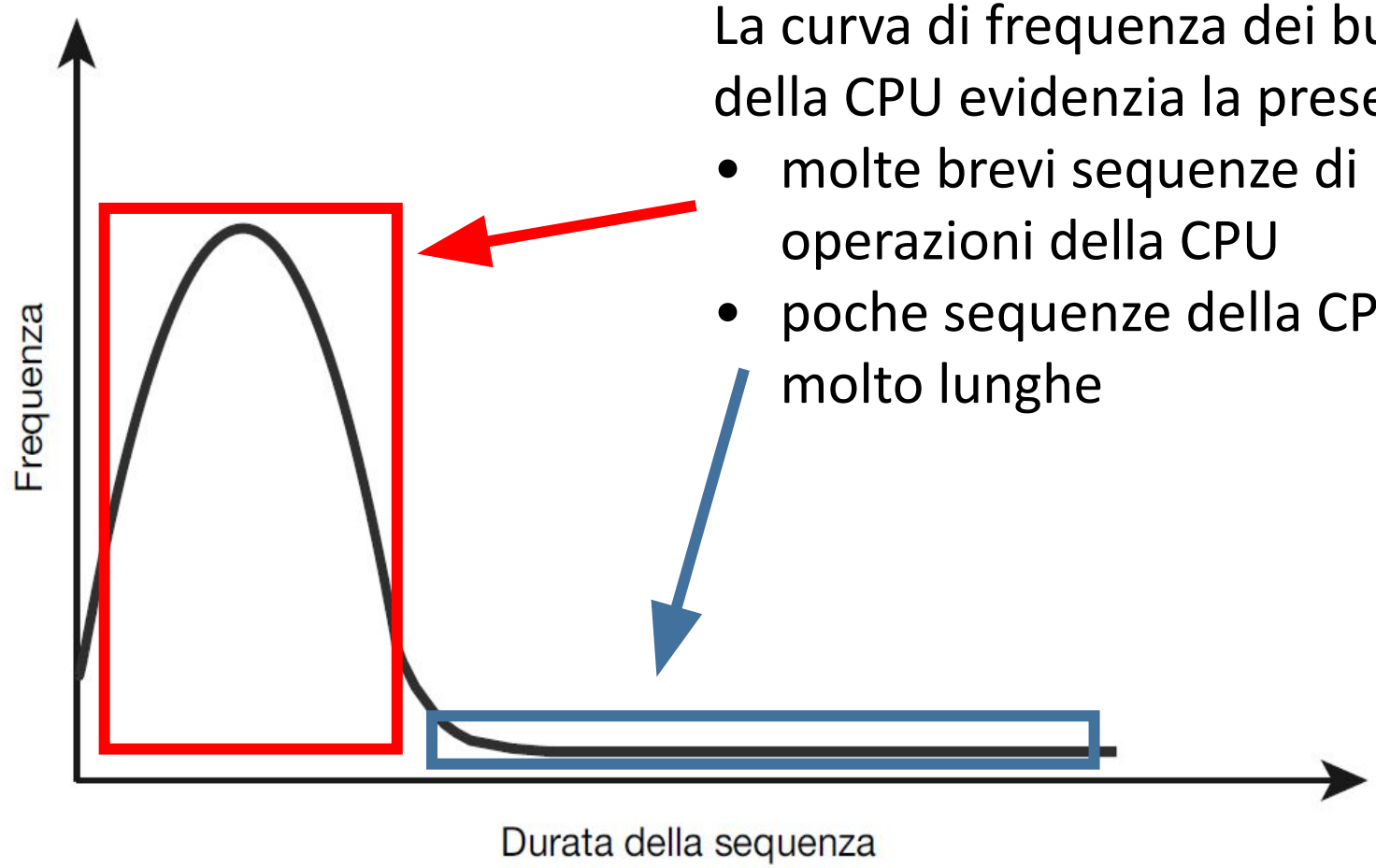


Figura 5.1 Serie alternata di sequenze di operazioni della CPU e di sequenze di operazioni di I/O.

Ciclicità delle fasi d'elaborazione e di I/O



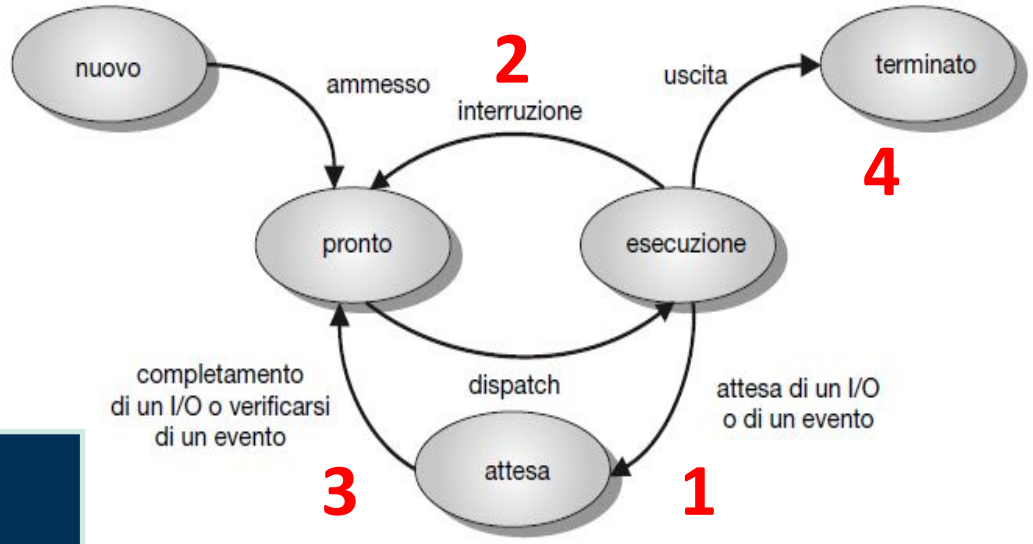
La curva di frequenza dei burst della CPU evidenzia la presenza di:

- molte brevi sequenze di operazioni della CPU
- poche sequenze della CPU molto lunghe

Figura 5.2 Diagramma delle durate delle sequenze di operazioni della CPU.

Lo scheduler

Lo scheduler interviene nelle seguenti situazioni:



1. un processo passa dallo stato di esecuzione allo stato di attesa

2. un processo passa dallo stato di esecuzione allo stato pronto

3. un processo passa dallo stato di attesa allo stato pronto

4. un processo termina

Scheduling con e senza prelazione

1. un processo passa dallo stato di esecuzione allo stato di attesa
2. un processo passa dallo stato di esecuzione allo stato pronto
3. un processo passa dallo stato di attesa allo stato pronto
4. un processo termina

Schemi di scheduling:

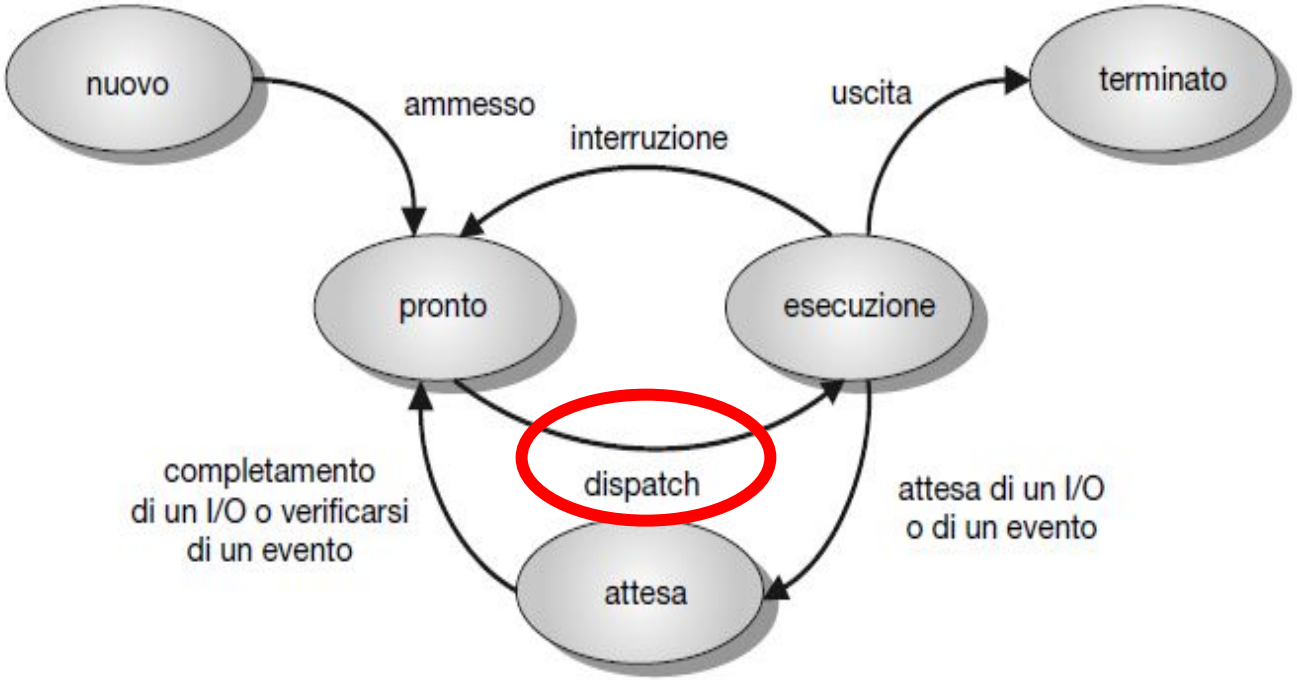
- **senza prelazione (*nonpreemptive*) o *cooperativo (cooperative);***
- **con prelazione (*preemptive*)**

Solo casi 1 e 4

Race condition

Dispatcher

Dispatcher → altro elemento coinvolto nella funzione di scheduling della CPU



Responsabile per:

- Context switch
- Passaggio alla modalit utente
- Riavviare esecuzione processo utente

Latenza di Dispatcher

Latenza di dispatch ⟶ tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro

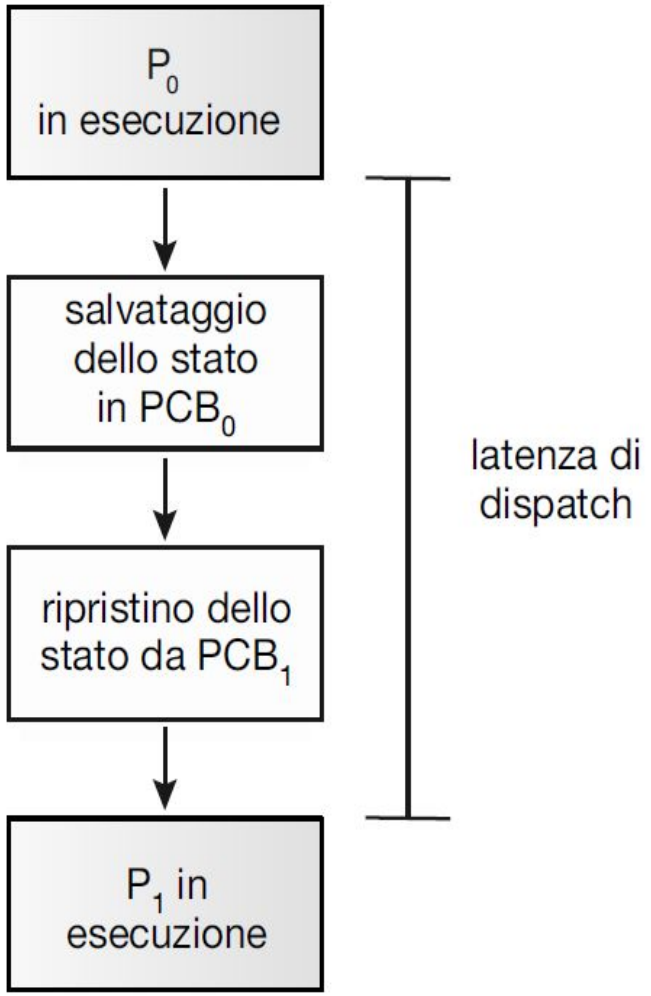
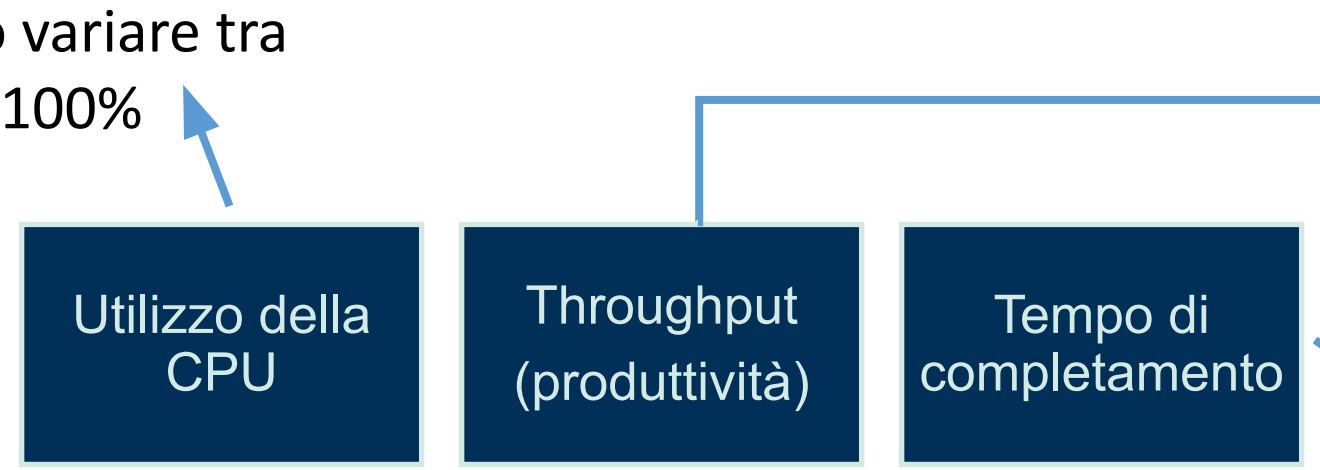


Figura 5.3 Ruolo del dispatcher.

Criteri di scheduling



Numero di processi completati nell'unità di tempo

Tempo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione

Somma degli intervalli di attesa passati nella ready queue

Tempo che intercorre tra l'effettuazione di una richiesta e la prima risposta prodotta

Algoritmi di scheduling

Esistono molti **algoritmi di scheduling della CPU** differenti:

- Scheduling in ordine d'arrivo (first-come, first-served, FCFS)
- Scheduling per brevità (shortest-job-first, SJF)
- Scheduling circolare (round-robin, RR)
- Scheduling con priorità
- Scheduling a code multilivello
- Scheduling a code multilivello con retroazione

First-Come, First-Served (FCFS)

Il più semplice algoritmo di scheduling della CPU è l'algoritmo di **scheduling in ordine d'arrivo *(scheduling first-come, first-served o FCFS)***

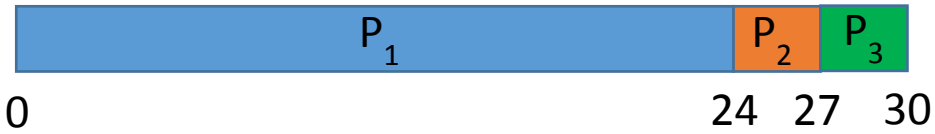
- La CPU si assegna al processo che la richiede per primo
- Senza prelazione
- Tempo medio di attesa spesso abbastanza lungo
- Effetto convoglio

Esempio FCFS

Processo	Tempo di CPU (CPU burst) in ms
P_1	24
P_2	3
P_3	3

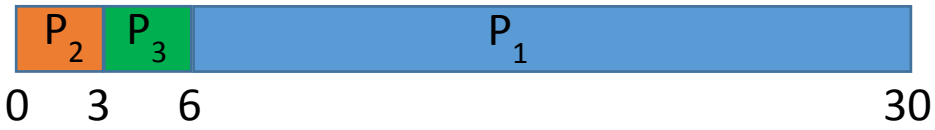
Ordine di arrivo: P_1 , P_2 , P_3

Diagramma di Gantt



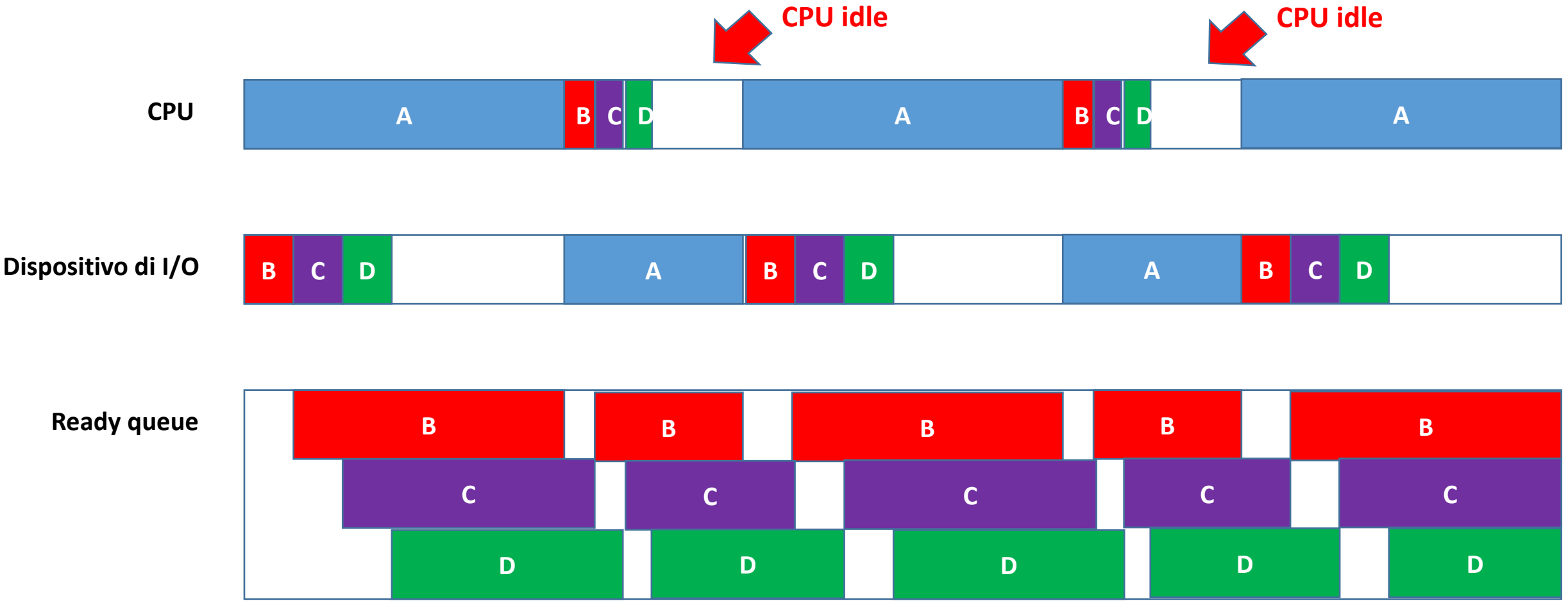
Tempo di attesa = (0 + 24 + 27) / 3 = 17 ms

Ordine di arrivo: P_2 , P_3 , P_1



Tempo di attesa = (0 + 3 + 6) / 3 = 3 ms

Effetto convoglio in FCFS



Shortest-Job-First (SJF)

L'**algoritmo di scheduling per brevità** (*shortest-job-first,* ***SJF)*** assegna la CPU al processo che ha la più breve lunghezza della successiva sequenza di operazioni della CPU.

↓

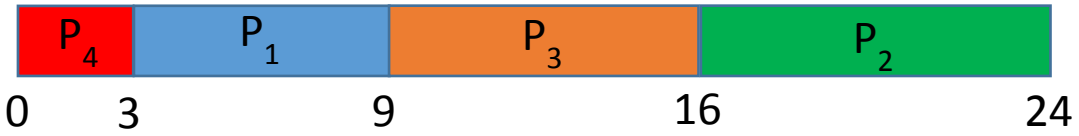
shortest next CPU burst

↓

Si esamina la lunghezza della successiva sequenza di operazioni della CPU del processo e non la sua lunghezza totale

Esempio SJF

Processo	Tempo di CPU (CPU burst) in ms
P_1	6
P_2	8
P_3	7
P_4	3

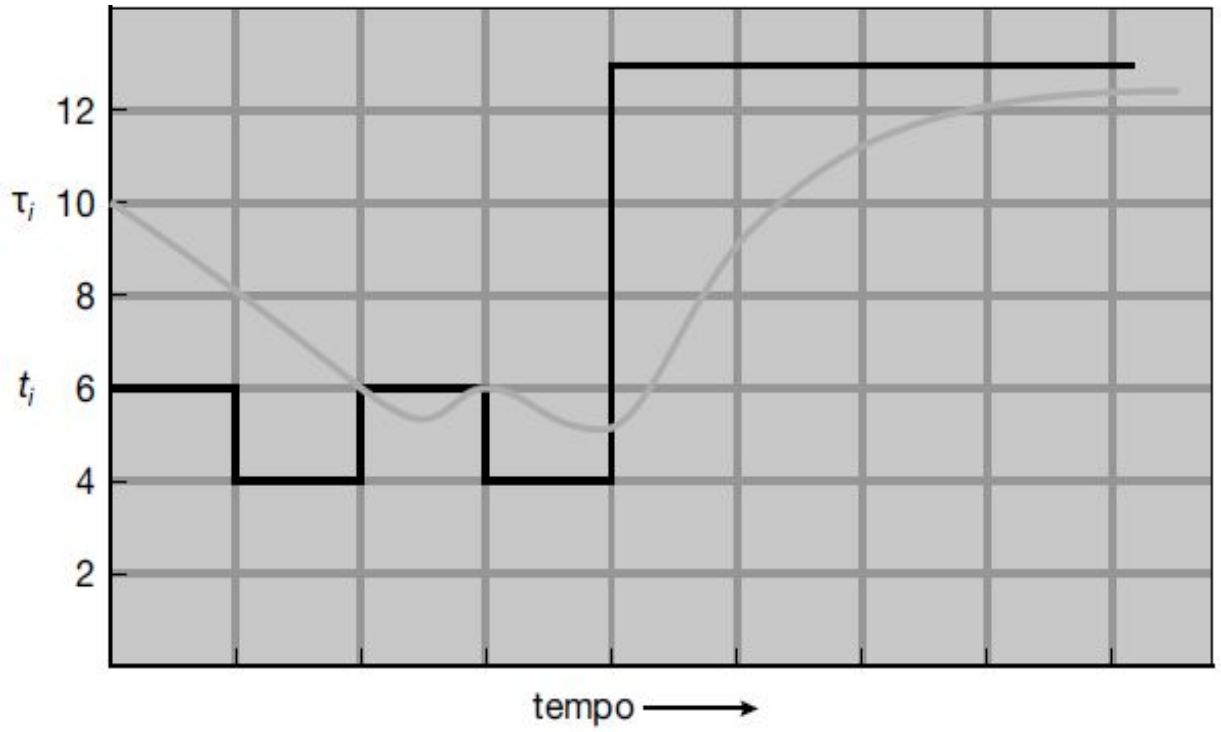


Tempo di attesa = (3 + 16 + 9 + 0) / 4 = 7 ms

Predizione in SJF

L'**algoritmo di scheduling per brevità** (*shortest-job-first,* ***SJF)*** non è attuabile in realtà, poiché non possiamo conoscere in anticipo la lunghezza della prossima CPU burst

Possiamo però provare a predire il suo valore usando, per esempio, una media esponenziale sulle precedenti osservazioni



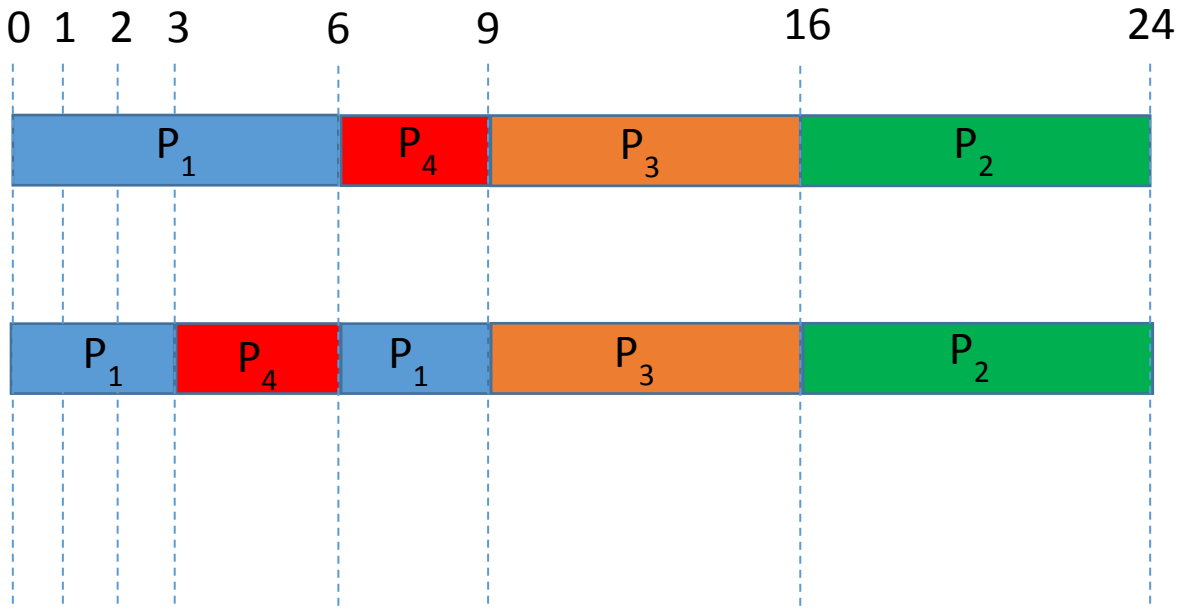
CPU burst (t_i)	6	4	6	4	13	13	13	...
"predizione" (τ_i)	10	8	6	6	5	9	11	12	...

Figura 5.4 Predizione della lunghezza della successiva sequenza di operazioni della CPU (CPU burst).

SJF con prelazione

Processo	Tempo di CPU (CPU burst) in ms	Istante di arrivo
P_1	6	0
P_2	8	1
P_3	7	2
P_4	3	3

prelazione



Tempo di attesa =
(0 + 15 + 7 + 3) / 4 = 6,25 ms

Tempo di attesa =
(3 + 15 + 7 + 0) / 4 = 6,25 ms

Round-Robin (RR)

L'**algoritmo di scheduling circolare (*round-robin, RR*)** è simile allo scheduling FCFS (in ordine di arrivo), ma aggiunge la capacità di prelazione in modo che il sistema possa commutare fra i vari processi.

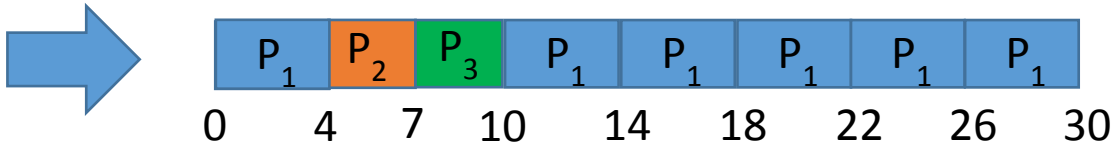
quanto di tempo o **porzione di tempo *(time slice)*** piccola quantità fissata del tempo della CPU ricevuta da un processo

Esempio RR

Processo	Tempo di CPU (CPU burst) in ms
P_1	24
P_2	3
P_3	3

Quanto di tempo = 4 ms

Ordine di arrivo: P_1 , P_2 , P_3



Tempo di attesa = (6 + 4 + 7) / 3 = 5,67 ms

Dimensione del quanto di tempo

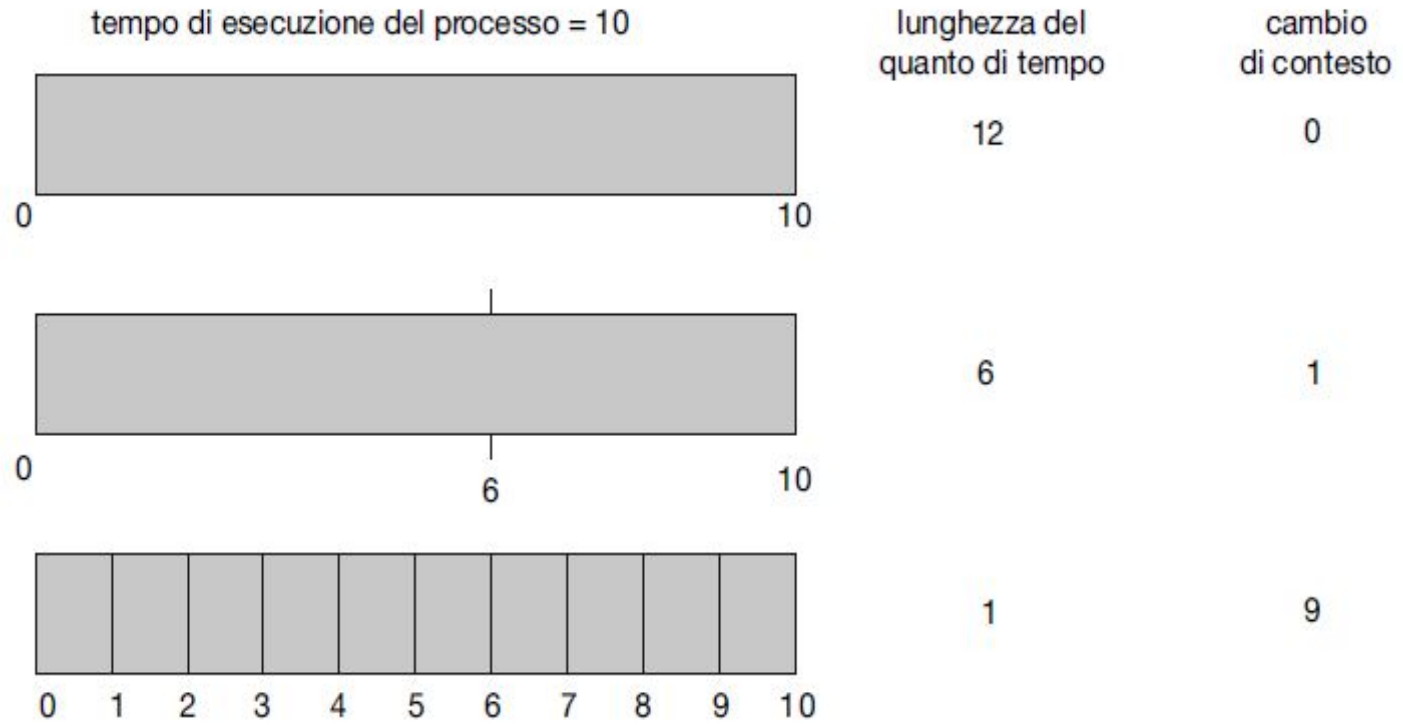
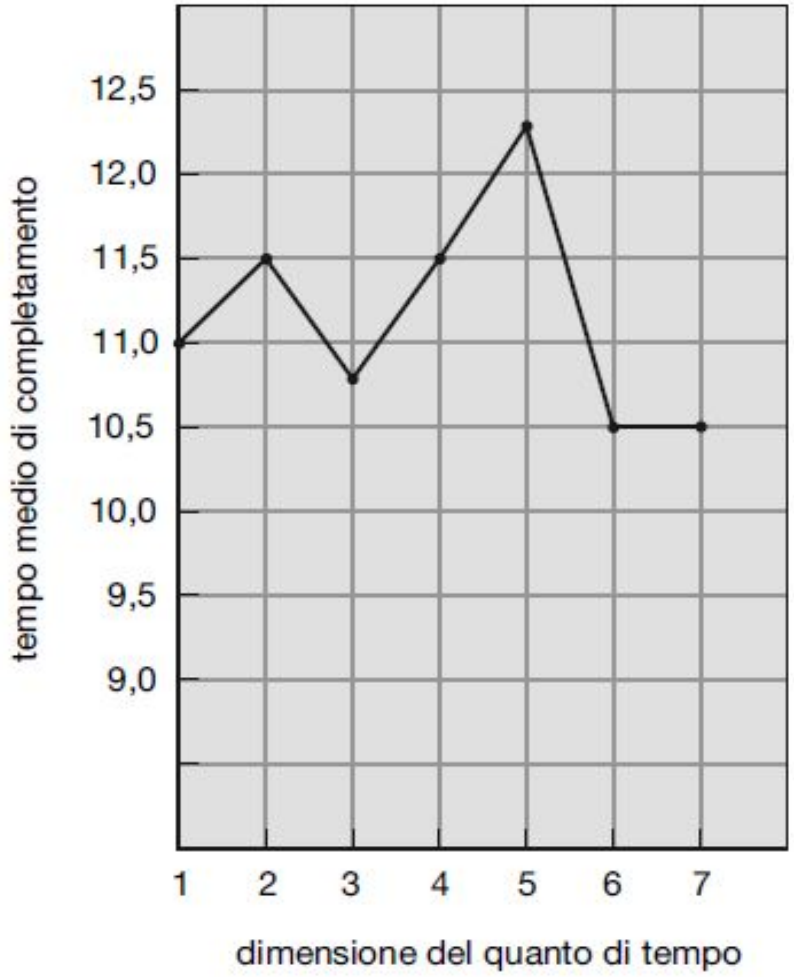


Figura 5.5 Aumento del numero dei cambi di contesto al diminuire del quanto di tempo.

Tempo di completamento

Il **tempo di completamento *(turnaround time)*** dipende dalla dimensione del quanto di tempo: com'è evidenziato nella a lato, il tempo di completamento medio di un insieme di processi non migliora necessariamente con l'aumento della dimensione del quanto di tempo.



processo	tempo
P_1	6
P_2	3
P_3	1
P_4	7

Figura 5.6 Variazione del tempo di completamento in funzione del quanto di tempo.

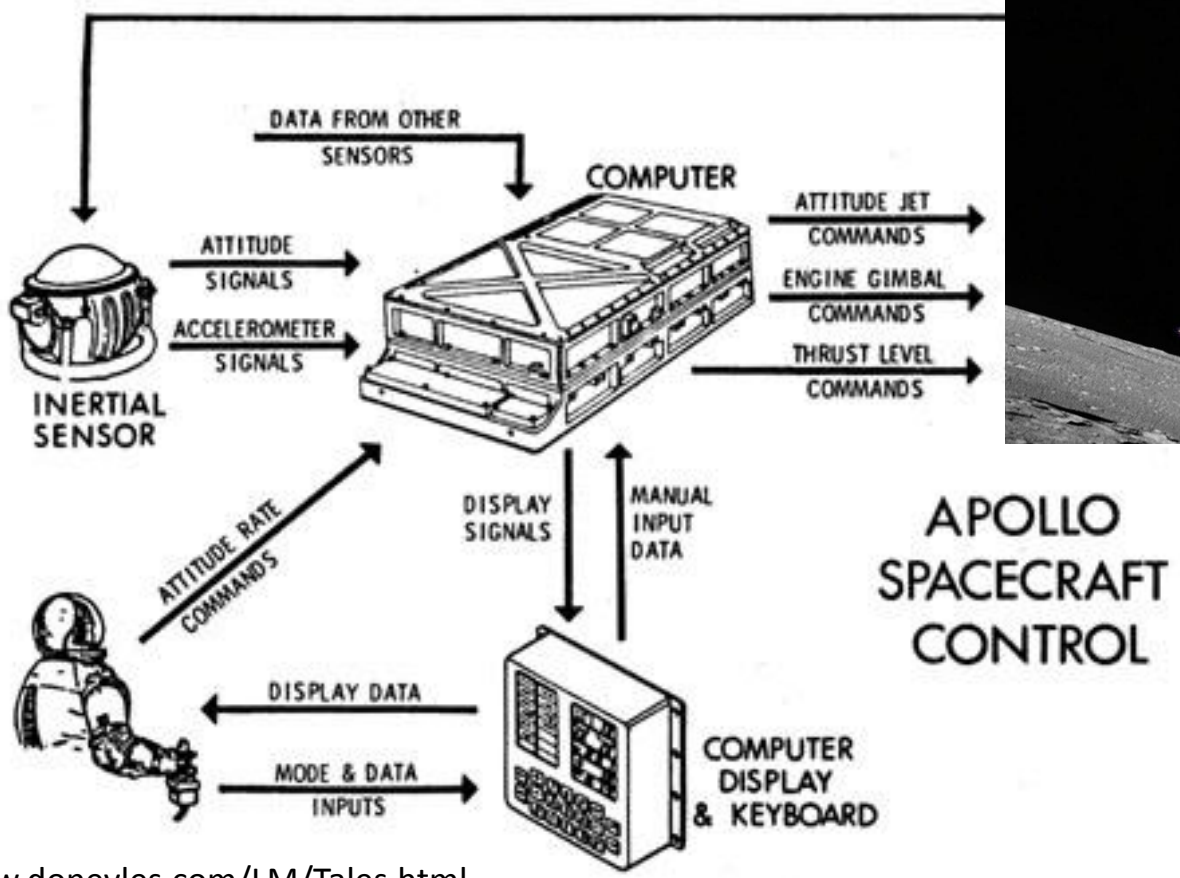
Scheduling con priorità

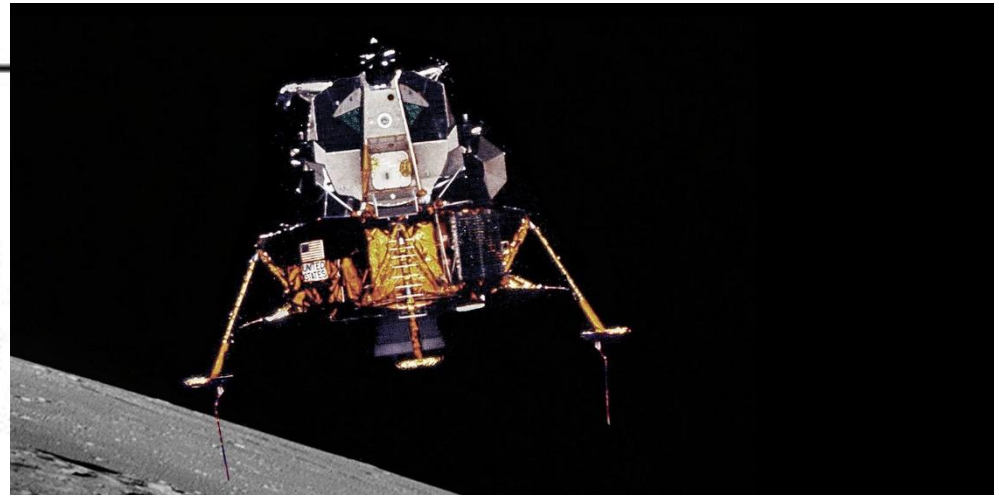
Algoritmo di scheduling con priorità → associa una priorità a ogni processo e assegna la CPU al processo con priorità più alta

- Le priorità sono fissate da un intervallo fisso di numeri
- Per esempio, numeri bassi indicano priorità alte
- Può generare starvation (attesa indefinita) → soluzione aging (invecchiamento)

Scheduling con priorità - Apollo 11

Algoritmo di scheduling con priorità → associa una priorità a ogni processo e assegna la CPU al processo con priorità più alta





https://www.doneyles.com/LM/Tales.html

Scheduling con code multilivello

scheduling a code multilivello

È spesso più semplice disporre di code separate per ciascuna priorità distinta e lasciare che lo scheduling con priorità si occupi semplicemente di selezionare il processo nella coda con priorità più alta

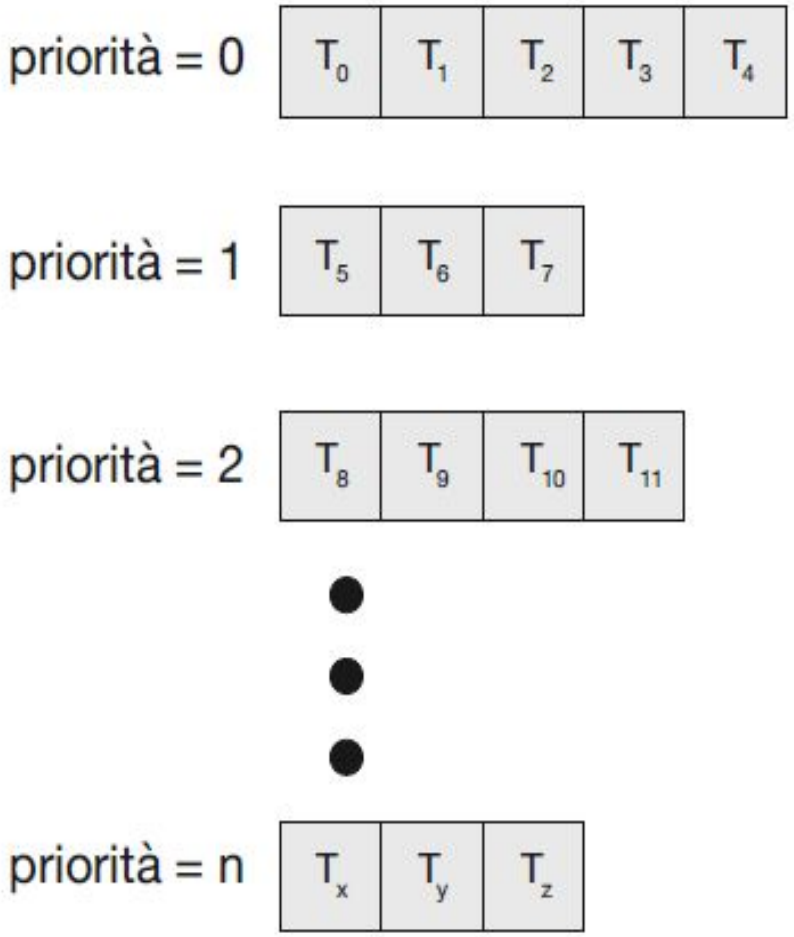


Figura 5.7 Code distinte per ogni priorità.

Scheduling con code multilivello

Un **algoritmo di scheduling a code multilivello** può anche essere utilizzato per suddividere i processi in diverse code in base al tipo di processo.

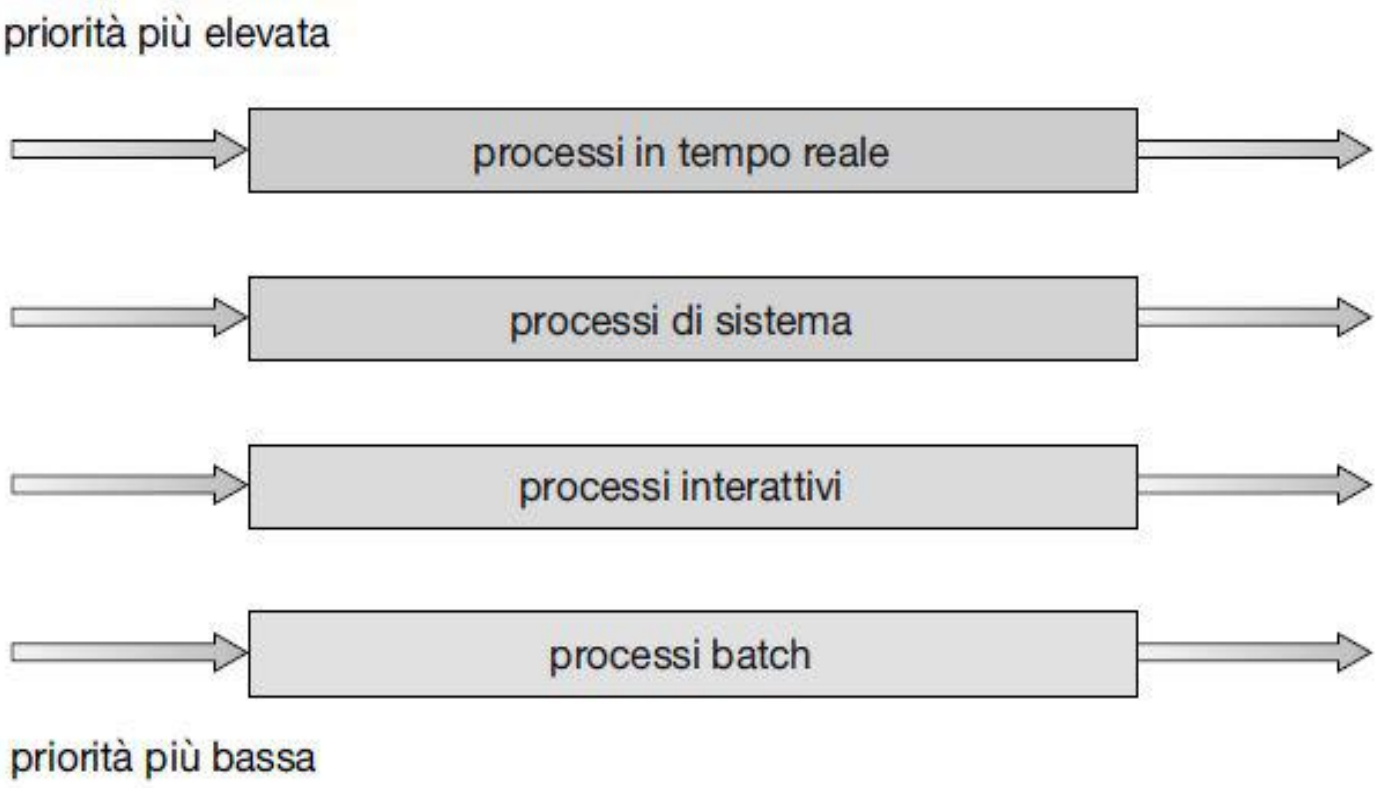


Figura 5.8 Scheduling a code multilivello.

Scheduling a code multilivello con retroazione

Lo **scheduling a code multilivello con retroazione (*multilevel feedback queue scheduling*)** permette ai processi di spostarsi fra le code.

È l'algoritmo più complesso.

È caratterizzato dai seguenti parametri:

- numero di code;
- algoritmo di scheduling per ciascuna coda;
- metodo usato per determinare quando spostare un processo in una coda con priorità maggiore;
- metodo usato per determinare quando spostare un processo in una coda con priorità minore;
- metodo usato per determinare in quale coda si deve mettere un processo quando richiede un servizio.

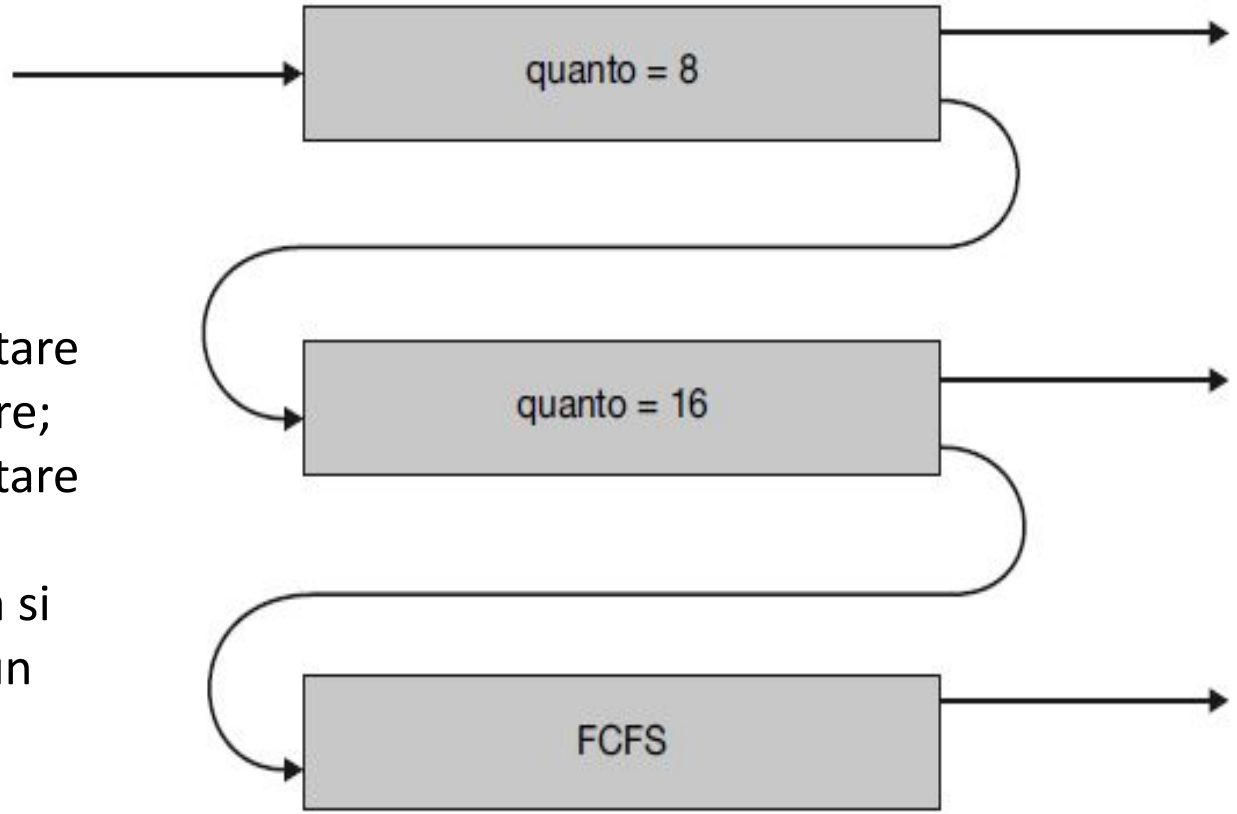


Figura 5.9 Code multilivello con retroazione.

Scheduling dei thread

a livello utente → ambito della contesa ristretto al processo
(process-contention scope, PCS)

e

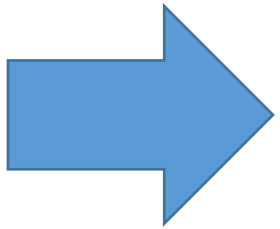
a livello kernel → ambito della contesa allargato al sistema
(system-contention scope, SCS)

- Nel caso del PCS, lo scheduling è solitamente basato sulle priorità: lo scheduler sceglie per l'esecuzione il thread con priorità più alta.

- La **API pthread di POSIX** consente di specificare *PCS* o *SCS* nella fase di generazione dei thread.

API di scheduling PThreads

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
  int i, scope;
  pthread_t tid[NUM_THREADS];
  pthread_attr_t attr;

  /* ottiene gli attributi di default */
  pthread_attr_init(&attr);

  /* per prima cosa appura l'ambito della contesa */
  if (pthread_attr_getscope(&attr, &scope) != 0)
    fprintf(stderr, "Impossibile appurare l'ambito della contesa\n");
  else {
    if (scope == PTHREAD_SCOPE_PROCESS)
      printf("PTHREAD_SCOPE_PROCESS");
    else if (scope == PTHREAD_SCOPE_SYSTEM)
      printf("PTHREAD_SCOPE_SYSTEM");
    else
      fprintf(stderr, "Valore d'ambito della contesa non ammesso.\n");
   }
```



API di scheduling PThreads

```
  /* imposta l'algoritmo di scheduling a PCS o SCS */
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* genera i thread */
  for (i = 0; i < NUM_THREADS; i++)
     pthread_create(&tid[i],&attr,runner,NULL);

  /* adesso aspetta la terminazione di tutti i thread */
  for (i = 0; i < NUM_THREADS; i++)
     pthread_join(tid[i], NULL);
}

/* ogni thread inizia l'esecuzione da questa funzione */
void *runner(void *param)
{
  /* fai qualcosa ... */

  pthread_exit(0);
}
```

Figura 5.10 API di scheduling Pthread.

Scheduling per sistemi multiprocessore

Il termine **multiprocessore** si applica attualmente alle seguenti architetture di sistema:

- CPU multicore
- Core multithread
- Sistemi NUMA (accesso non uniforme alla memoria)
- Sistemi multiprocessore eterogenei

Scheduling per sistemi multiprocessore

L'approccio standard per supportare i multiprocessori è la **multielaborazione simmetrica (SMP)**, in cui ciascun processore è in grado di autogestirsi

La SMP offre due possibili strategie per organizzare i thread da selezionare per l'esecuzione:

(a) tutti i thread possono trovarsi in una ready queue comune;

(b) ogni processore può avere una propria coda privata di thread.

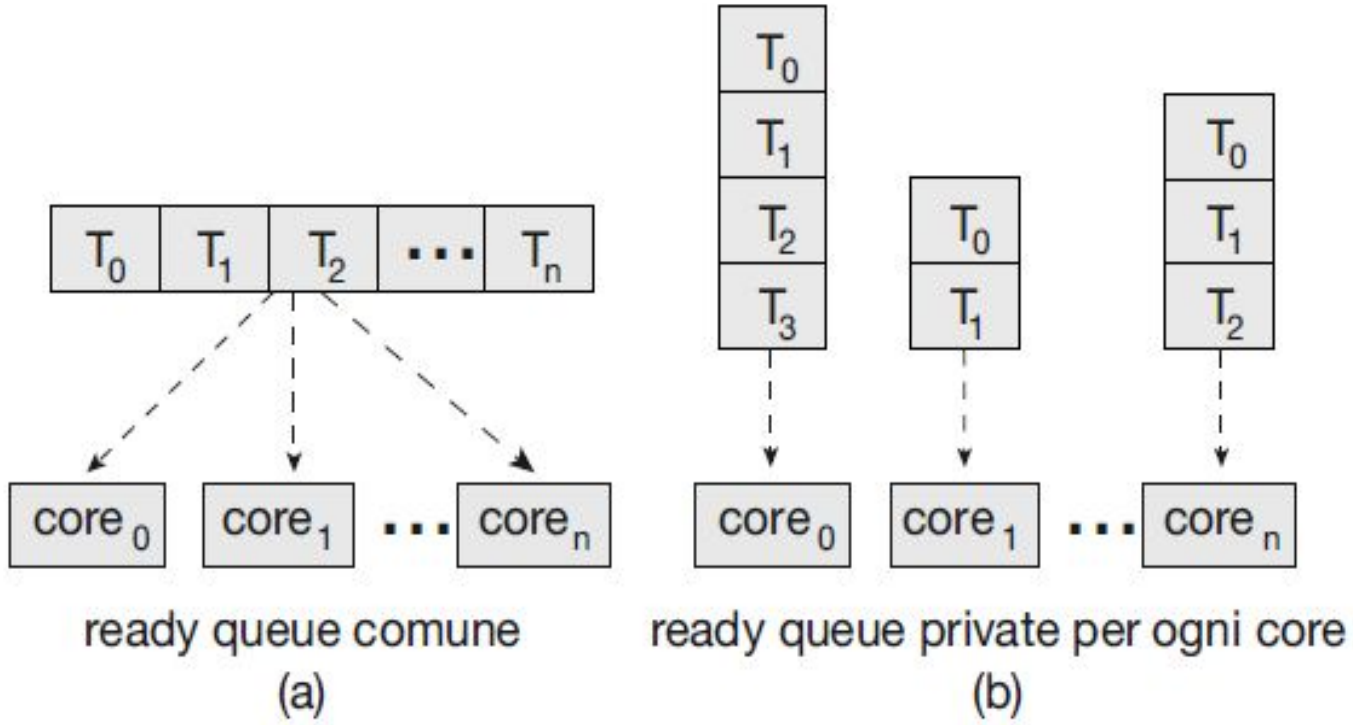


Figura 5.11 Organizzazione di ready queue.

Scheduling per sistemi multicore

Inserire più core di elaborazione in un unico chip fisico → **processore multicore**

Quando un processore accede alla memoria, una quantità significativa di tempo trascorre in attesa della disponibilità dei dati → **stallo della memoria**

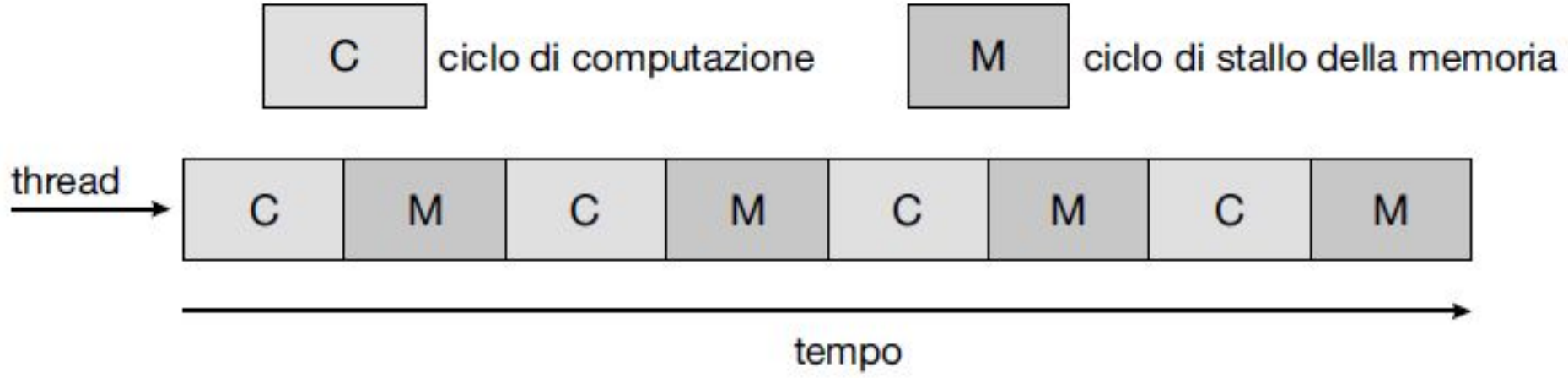


Figura 5.12 Stallo della memoria.

Scheduling per sistemi multicore

La Figura 5.13 mostra un processore a due thread nel quale l'esecuzione dei thread 0 e 1 sono avvicendate nel tempo.

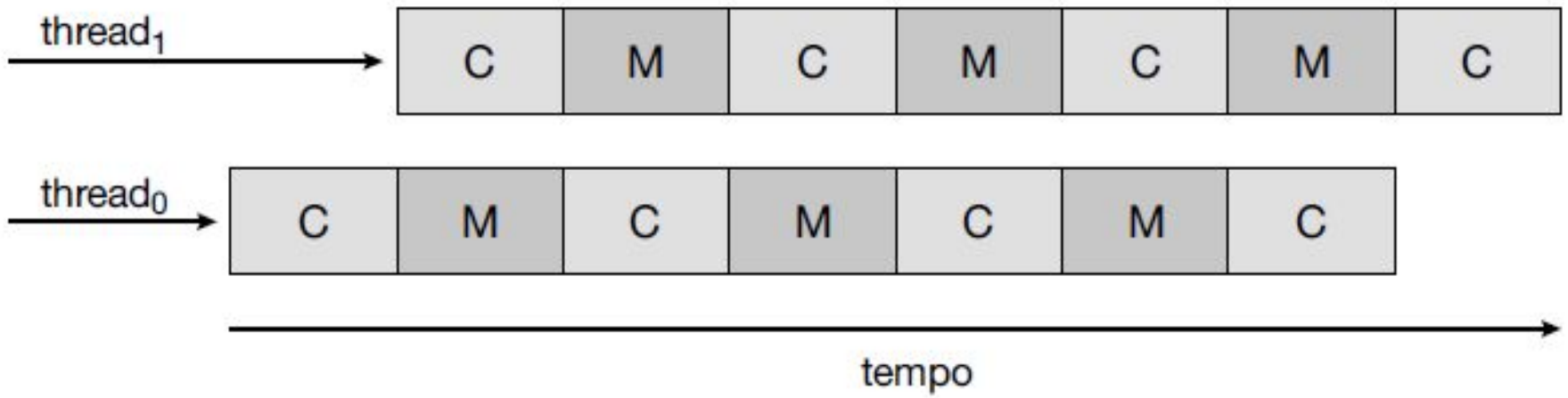


Figura 5.13 Sistema multicore e multithread.

Hyper-threading

chip multithreading (CMT) o hyper-threading

un processore contiene quattro *core di elaborazione*, ognuno dei quali contiene due *thread hardware*: dal punto di vista del sistema operativo sono presenti otto **CPU logiche**.

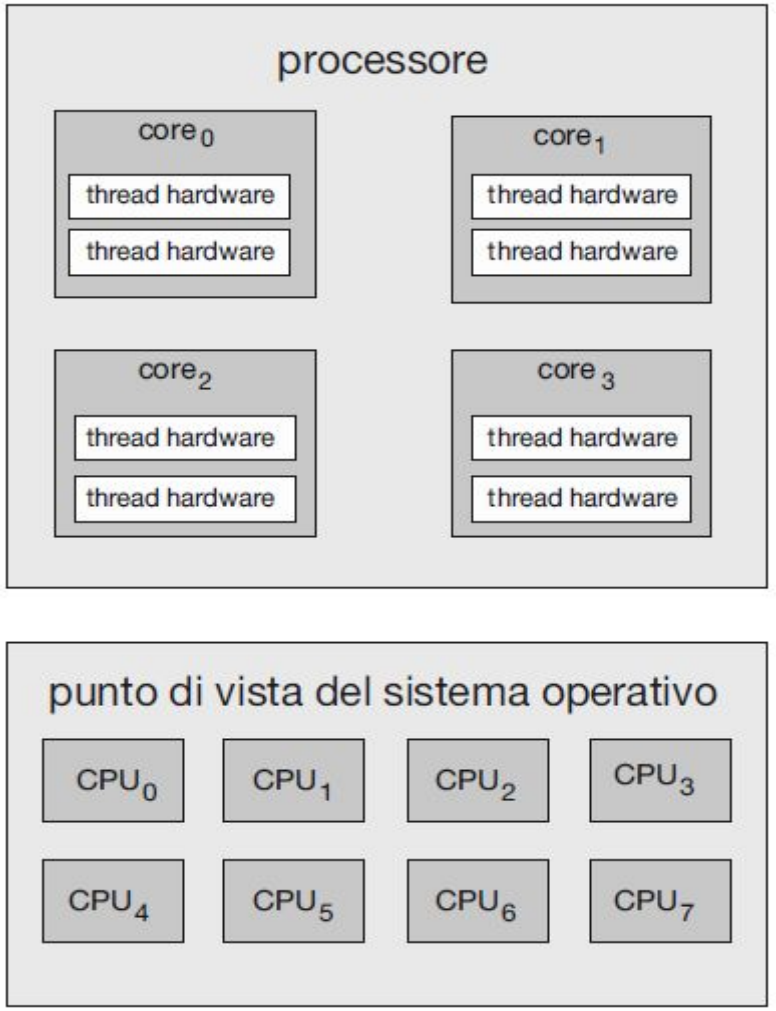
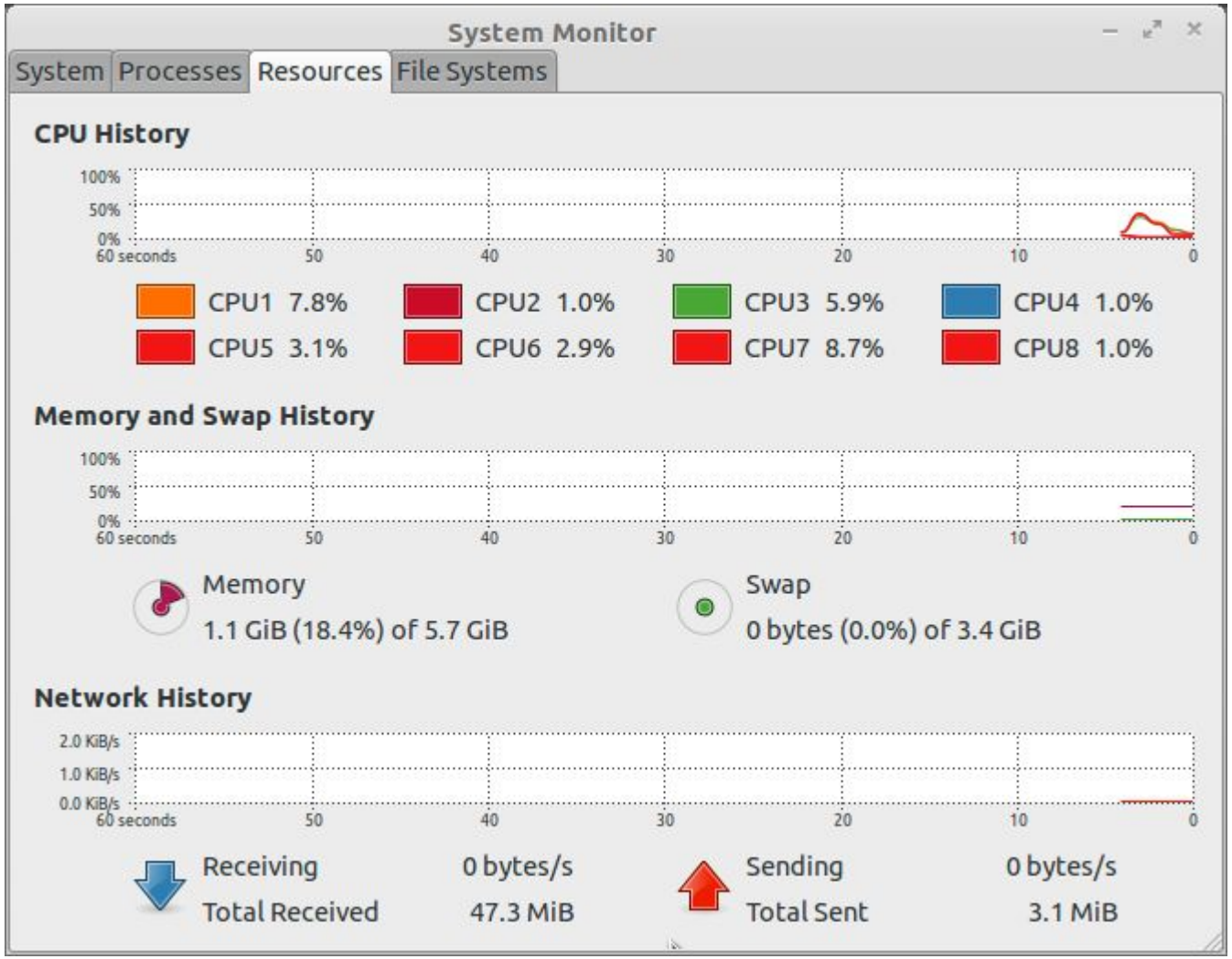


Figura 5.14 Chip multithreading.

Esempio Gnome System Monitor



Scheduling a due livelli

Un processore *multithreaded* e *multicore* richiede due diversi livelli di scheduling, come mostrato nella Figura 5.15, che illustra un core di elaborazione dual-threaded.

I due diversi livelli di scheduling mostrati nella Figura 5.15 non sono necessariamente mutuamente esclusivi

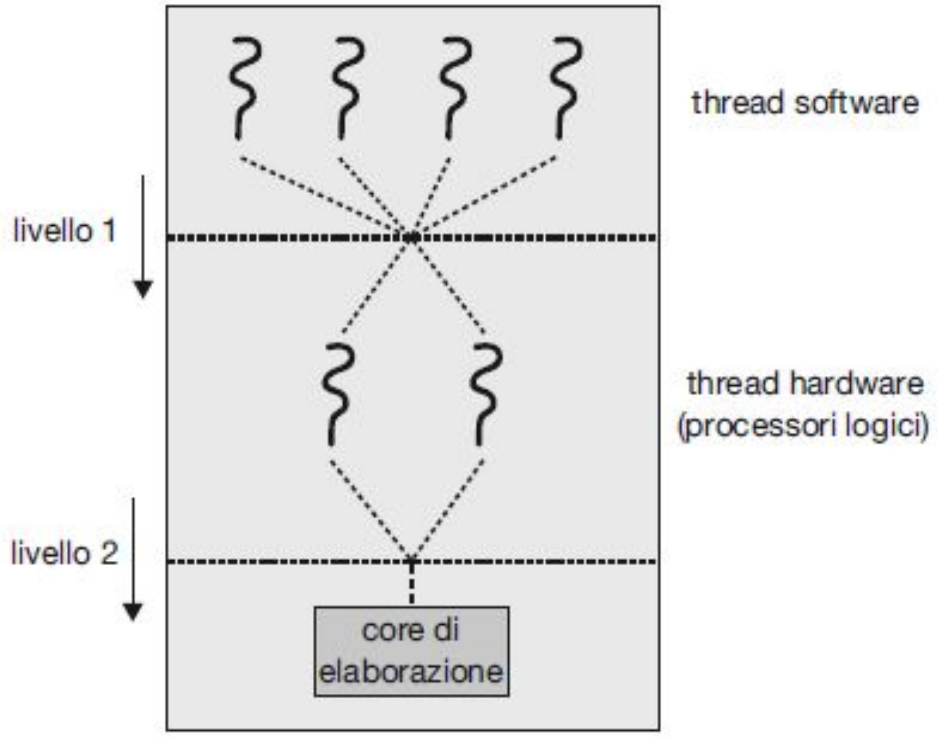


Figura 5.15 Due livelli di scheduling.

NUMA e scheduling della CPU

predilezione per il processore *(processor affinity)*, → un processo ha una predilezione per il processore su cui è in esecuzione.

- **predilezione debole *(soft affinity)***
- **predilezione forte *(hard affinity)***

La Figura 5.16 mostra un'architettura con accesso non uniforme alla memoria (**NUMA**) in cui sono presenti due chip fisici di processore, ciascuno con la propria CPU e memoria locale.

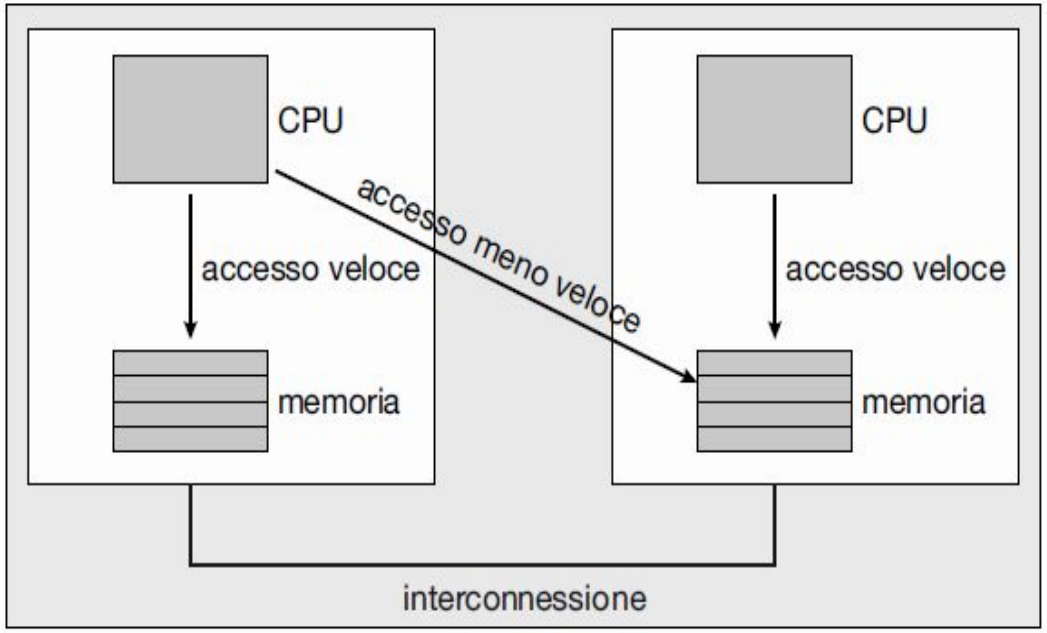


Figura 5.16 NUMA e lo scheduling della CPU.

Scheduling real-time della CPU

I sistemi real-time sono per loro natura guidati dagli eventi: generalmente, il sistema attende che si verifichi un evento in tempo reale.

Figura 5.17 Latenza relativa all'evento.

Scheduling real-time della CPU

sistemi soft real-time

non offrono garanzie sul momento in cui un processo critico sarà eseguito, ma assicurano solamente che sarà data precedenza a quest'ultimo piuttosto che ad altri processi non critici.

sistemi hard real-time

hanno vincoli più rigidi: i task vanno eseguiti entro una scadenza prefissata ed eseguirli dopo tale scadenza è del tutto inutile

Latenza nei sistemi real-time

Le categorie di latenza che influiscono sul funzionamento dei sistemi real-time sono:

- **Latenza relativa alle interruzioni**
- **Latenza relativa al dispatch**

Latenza relativa alle interruzioni

Latenza relativa alle interruzioni

si riferisce al periodo di tempo compreso tra la notifica di un'interruzione alla CPU e l'avvio della routine che gestisce l'interruzione (ISR)

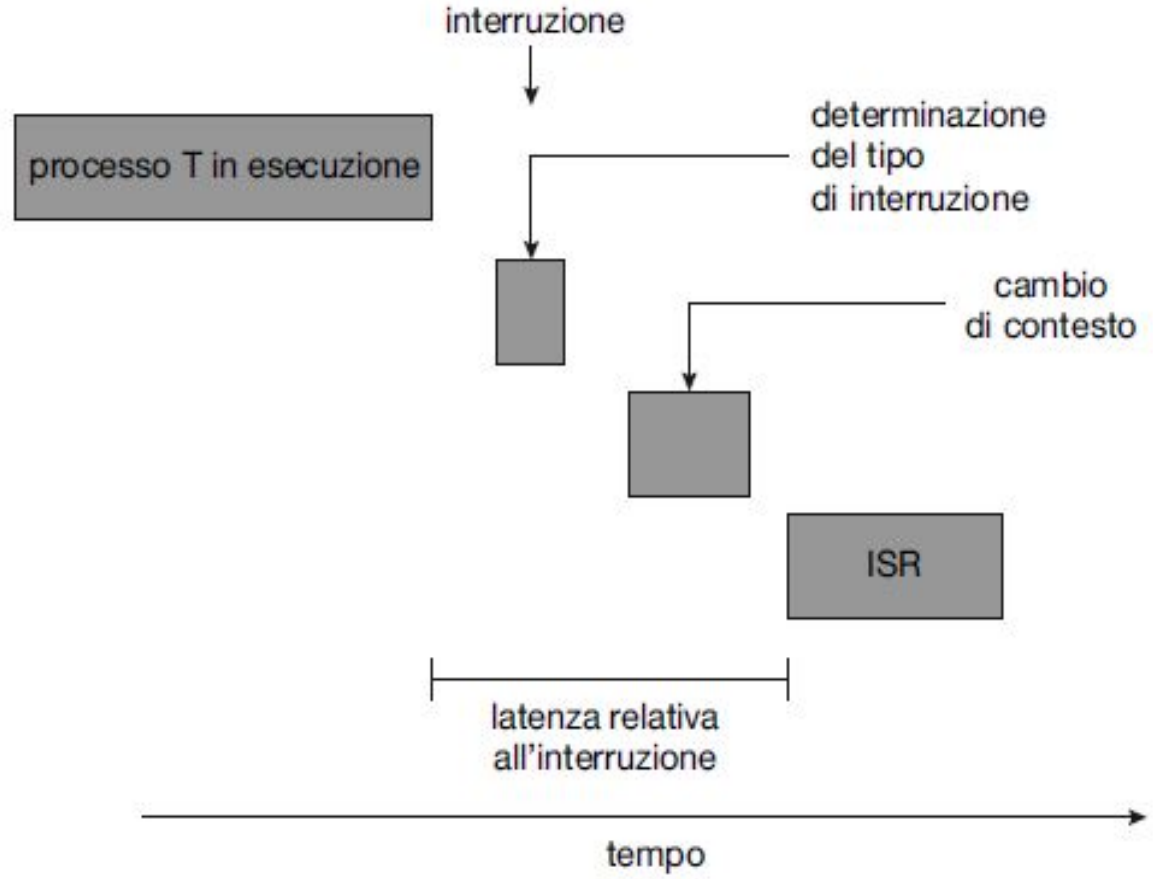


Figura 5.18 Latenza relativa alle interruzioni.

Latenza

latenza relativa al dispatch: periodo di tempo necessario al dispatcher per bloccare un processo e avviarne un altro

La *fase di conflitto della latenza di dispatch* consiste di due componenti:

1. prelazione di ogni processo in esecuzione nel kernel;

2. cessione, da parte dei processi a bassa priorità, delle risorse richieste dal processo ad alta priorità.

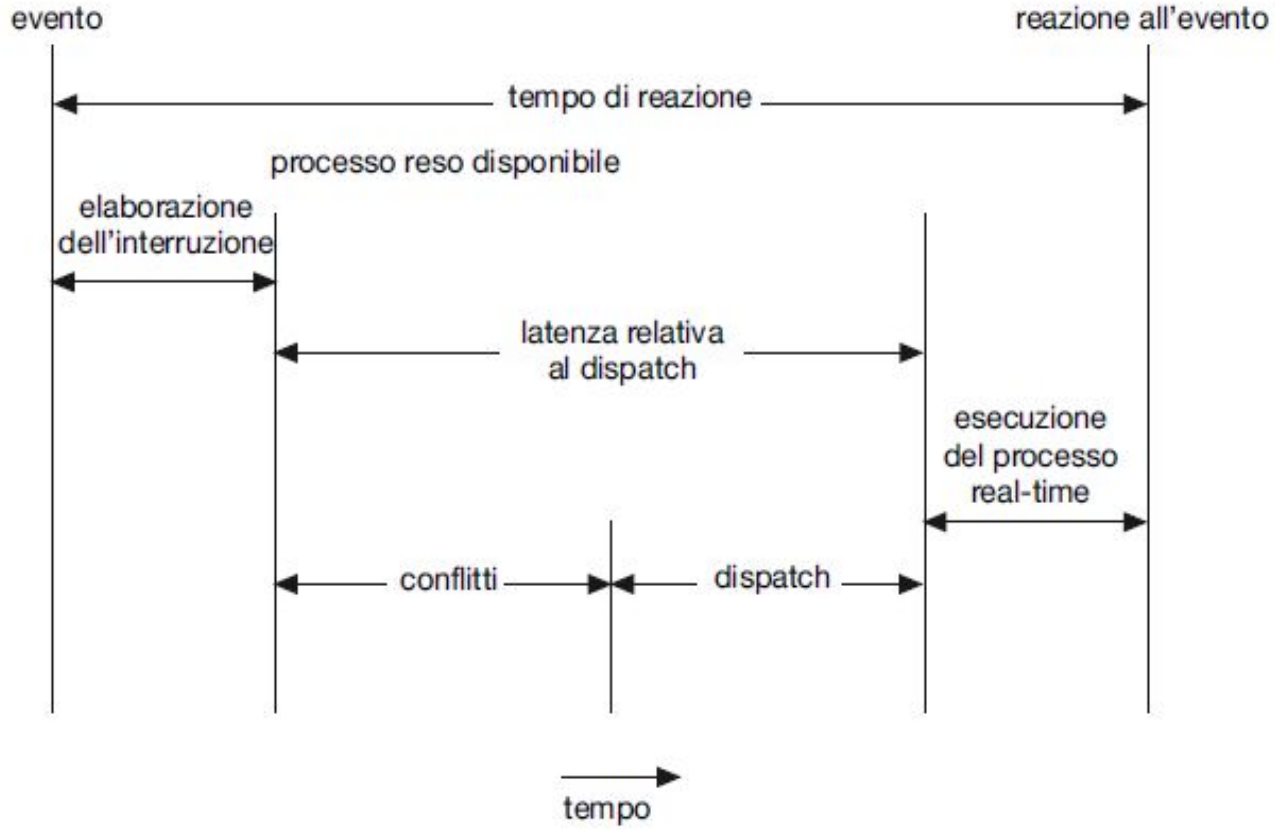


Figura 5.19 Latenza relativa al dispatch.

Scheduling basato sulla priorità

- Gli algoritmi di scheduling con priorità assegnano a ogni processo una priorità in base alla loro importanza
- I processi sono considerati **periodici**, nel senso che richiedono la CPU a intervalli costanti di tempo (periodi)

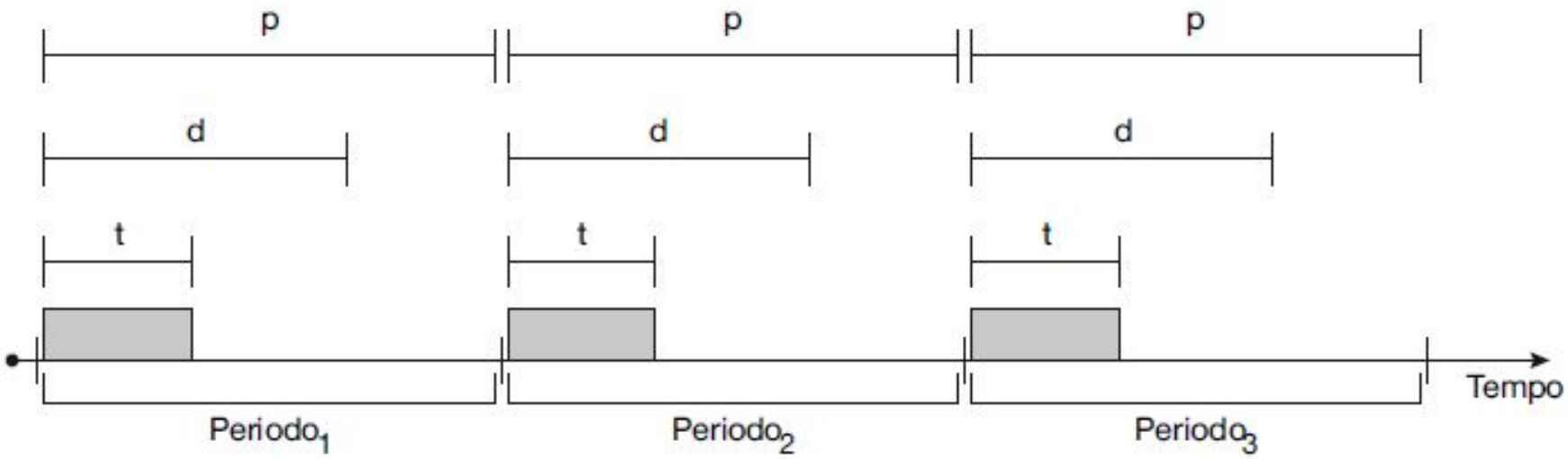


Figura 5.20 Processo periodico.

Scheduling con priorità proporzionale alla frequenza

Ciascun task periodico si vede assegnare una priorità inversamente proporzionale al proprio periodo:

- più breve è il periodo, più alta la priorità;
- più lungo il periodo, più bassa la priorità.

P_1 con periodo $p_1 = 50$
e tempo di elaborazione
$t_1 = 20$

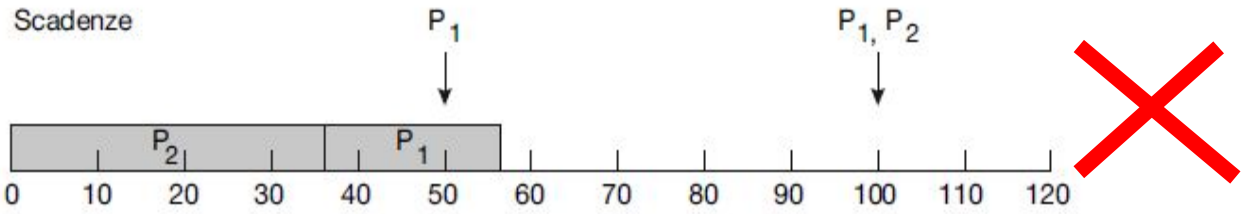


Figura 5.21 Scheduling dei task in caso P_2 abbia priorità maggiore di P_1.

P_2 con periodo $p_2 = 100$
e tempo di elaborazione
$t_2 = 35$

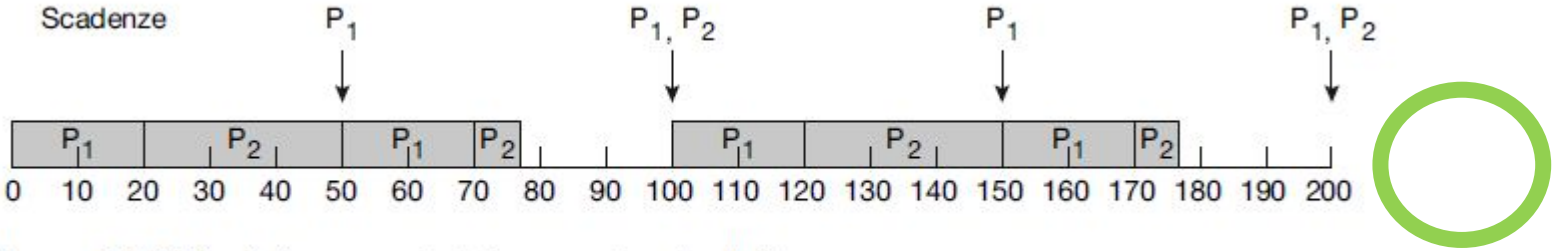


Figura 5.22 Scheduling con priorità proporzionale alla frequenza.

Scheduling EDF

Lo **scheduling EDF** *(earliest-deadline-first, ossia "per prima la scadenza più ravvicinata")*, attribuisce le priorità dinamicamente, sulla base delle scadenze.

· Più vicina è la scadenza, maggiore è la priorità

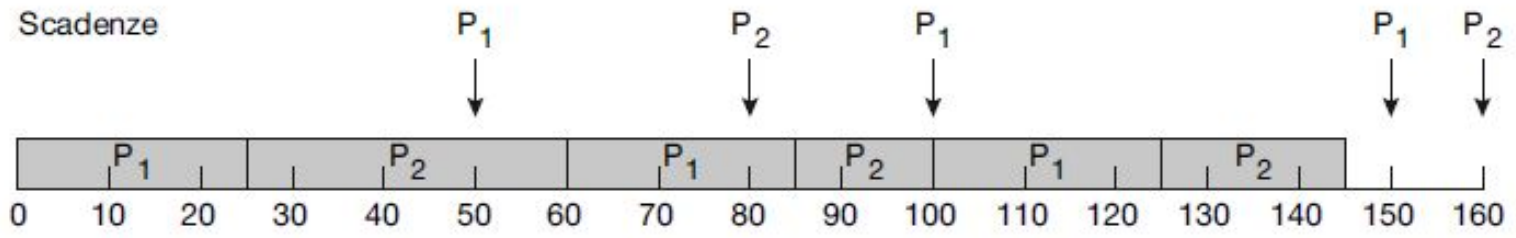
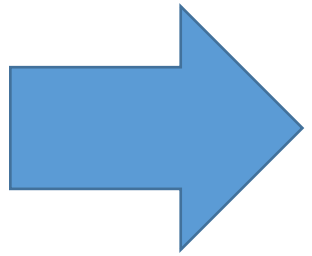


Figura 5.24 Scheduling EDF.

A differenza dell'algoritmo con priorità proporzionale alla frequenza, lo scheduling EDF non postula la periodicità dei processi, e non prevede neanche di impiegare sempre la stesso tempo della CPU per ogni burst.

Scheduling real-time POSIX

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
  int i, policy;
  pthread_t tid[NUM_THREADS];
  pthread_attr_t attr;

  /* appura gli attributi di default */
  pthread_attr_init(&attr);

  /* appura la politica di scheduling corrente */
  if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
    fprintf(stderr, "Unable to get policy.\n");
  else {
    if (policy == SCHED_OTHER)
     printf("SCHED_OTHER\n");
    else if (policy == SCHED_RR)
     printf("SCHED_RR\n");
    else if (policy == SCHED_FIFO)
     printf("SCHED_FIFO\n");
  }
```



Scheduling real-time POSIX

```
  /* imposta la politica di scheduling – FIFO, RR o OTHER */
  if (pthread_attr_setschedpolicy(&attr, SCHED_OTHER) != 0)
    fprintf(stderr, "Unable to set policy.\n");

  /* genera i thread */
  for (i = 0; i < NUM_THREADS; i++)
     pthread_create(&tid[i],&attr,runner,NULL);

  /* ora attende la terminazione di ciascun thread */
  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* ciascun thread comincia l'esecuzione da questa funzione */
void *runner(void *param)

{
   /* fai qualcosa ... */

   pthread_exit(0);
}
```

Figura 5.25 Scheduling real-time con la API POSIX.

Criteri di scheduling per OS

Scheduling in Linux

Nei sistemi Linux lo scheduling si basa sulle **classi di scheduling**

Per decidere quale task eseguire, lo scheduler seleziona il task con priorità più alta appartenente alla classe di scheduling a priorità più elevata.

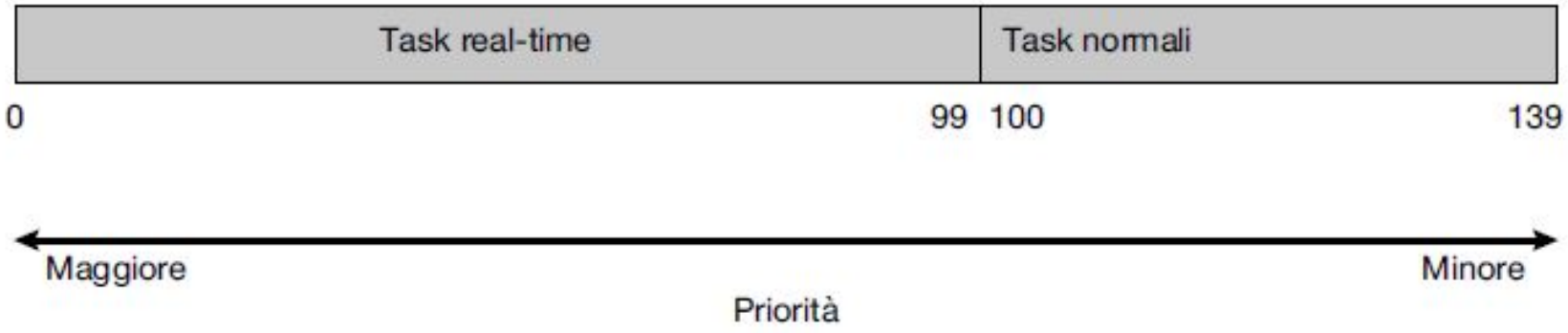


Figura 5.26 Priorità di scheduling in Linux.

Dominio di scheduling in Linux

Un **dominio di scheduling** è un insieme di core che può essere bilanciato l'uno con l'altro, come mostrato nella Figura 5.27.

I core in ciascun dominio di scheduling sono raggruppati in base al modo in cui condividono le risorse del sistema.

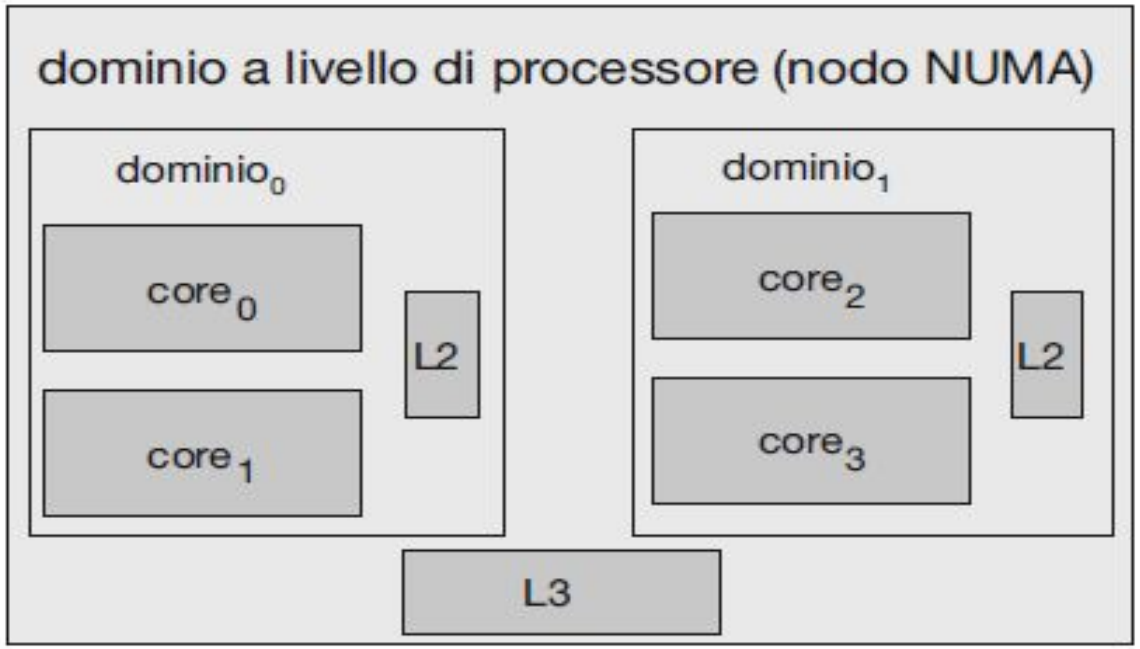


Figura 5.27 Bilanciamento del carico NUMA-aware nello scheduler CFS di Linux.

Scheduling in Windows

- Lo scheduler di Windows assicura che si eseguano sempre i thread a più alta priorità.
- La porzione del kernel che si occupa dello scheduling si chiama ***dispatcher***
- Le priorità sono suddivise in due classi:
  - la **classe variable** → raccoglie i thread con priorità da 1 a 15,
  - la **classe real-time** → raccoglie i thread con priorità tra 16 e 31
- La priorità di ciascun thread dipende dalla priorità della classe cui appartiene e dalla priorità relativa che il thread ha all'interno della stessa classe.

Classi di priorità nello scheduling in Windows

	real-time	high	above_ normal	normal	below_ normal	idle_ priority
time_critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above_normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below_normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figura 5.28 Priorità dei thread in Windows.

Valutazione degli algoritmi

metodi di valutazione dell'algoritmo
di scheduling della CPU

Uso di simulazioni

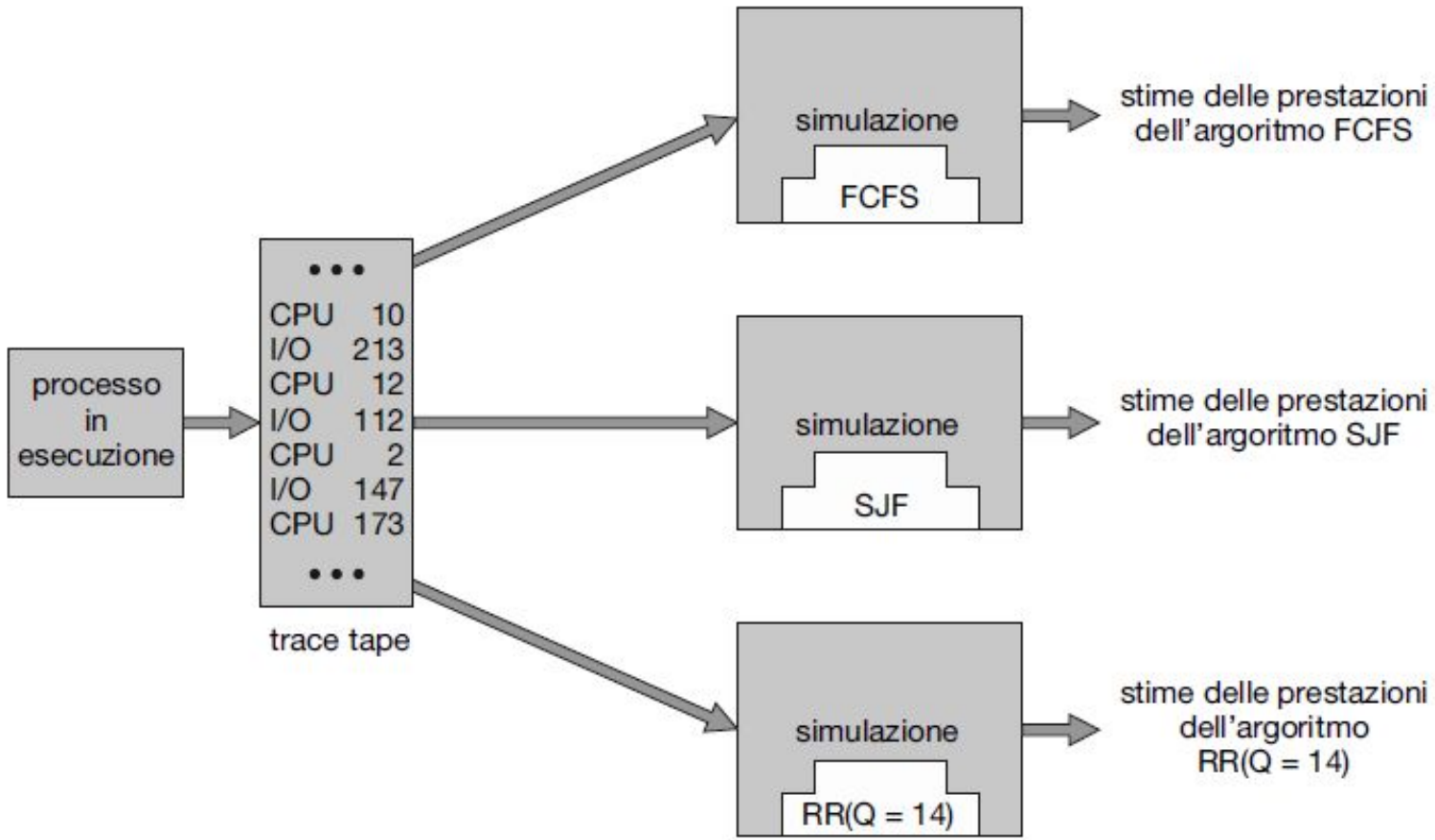


Figura 5.31 Valutazione di algoritmi di scheduling della CPU tramite una simulazione.