



HOEPLI
TECNICA
PER LA SCUOLA

Paolo Camagni
Riccardo Nikolassy

TECNOLOGIE E PROGETTAZIONE DI SISTEMI INFORMATICI E DI TELECOMUNICAZIONI

Nuova Edizione **OPENSCHOOL**

Per l'articolazione INFORMATICA
degli Istituti Tecnici
settore Tecnologico

Edizione **OPENSCHOOL**

1 LIBRO DITESTO

2 E-BOOK+

3 RISORSE ONLINE

4 PIATTAFORMA



HOEPLI

PAOLO CAMAGNI RICCARDO NIKOLASSY

Tecnologie e progettazione di sistemi informatici e di telecomunicazioni

Nuova Edizione OPEN SCHOOL

**Per l'articolazione Informatica
degli Istituti Tecnici settore Tecnologico**

VOLUME 2



EDITORE ULRICO HOEPLI MILANO

Copyright © Ulrico Hoepli Editore S.p.A. 2016
Via Hoepli 5, 20121 Milano (Italy)
tel. +39 02 864871 – fax +39 02 8052886
e-mail hoepli@hoepli.it

www.hoepli.it



Tutti i diritti sono riservati a norma di legge
e a norma delle convenzioni internazionali

Presentazione

Scopo dell'opera è fornire le basi teoriche e pratiche per poter realizzare la **programmazione concorrente**. La **Nuova Edizione Openschool**, concepita secondo le recenti indicazioni ministeriali, è caratterizzata dall'integrazione tra testo cartaceo, libro digitale e risorse digitali a esso collegate (eBook+), ulteriori materiali multimediali disponibili al sito www.hoepliscuola.it e, a discrezione del docente, la piattaforma didattica.

Il volume mantiene la strutturazione della precedente edizione ed è idealmente organizzato in quattro unità di apprendimento:

- ▶ introduzione ai processi paralleli e ai thread;
- ▶ la comunicazione e sincronizzazione tra processi;
- ▶ la specifica dei requisiti di un progetto;
- ▶ la documentazione del software.

Completamente revisionato, il testo recepisce le indicazioni ricevute dai docenti che hanno in uso la precedente edizione. Le parti che comprendono le **lezioni teoriche** sono state revisionate mantenendo sostanzialmente la stessa strutturazione dell'edizione precedente. In questo modo è garantita la continuità didattica per coloro che già hanno impostato il proprio piano di lavoro utilizzando tale organizzazione dei contenuti. Questi sono stati rivisti con commenti e osservazioni che integrano le parti che si sono dimostrate più complesse nel processo di apprendimento da parte degli alunni.

La parte di **laboratorio** è stata ristrutturata aggiungendo **7 nuove lezioni di laboratorio** in modo da frazionare l'acquisizione delle competenze mediante passi successivi più graduati rispetto alla precedente edizione.

In particolare:

- ▶ nella UA1 sono state aggiunte quattro esercitazioni sulla fork dei processi in linguaggio C in modo da rendere più graduale l'approccio labororiale a questo argomento, è stata rivista e ampliata l'esercitazione sui thread ed è stata aggiunta una esercitazione sulla schedulazione dei thread;
- ▶ nella UA2 sono state aggiunte tre esercitazioni sulla sincronizzazione dei thread e sui monitor e una nuova lezione sulla comunicazione tra processi mediante segnali asincroni;
- ▶ nelle UA3 e UA4 è stato aggiornato il software per la realizzazione dei diagrammi UML e per la realizzazione della documentazione.

Come la precedente edizione, ogni sezione è composta da unità di apprendimento suddivise in più lezioni. Ciascuna lezione teorica di ogni unità di apprendimento è affiancata da specifiche lezioni pratiche dove le problematiche teoriche vengono affrontate e risolte sia utilizzando il **linguaggio C** che il **linguaggio Java**.

Metodologia e strumenti didattici

Le finalità e i contenuti dei diversi argomenti affrontati sono descritti dagli **obiettivi generali** e dalle indicazioni **In questa lezione impareremo**; alla fine di ogni lezione, per lo studente sono presenti **esercizi**, anche **interattivi**, di valutazione delle conoscenze e delle competenze raggiunte, suddivisi in **domande a risposta multipla**, a completamento, esercizi con **procedure guidate**.

Caratteristiche della nuova edizione

La Nuova Edizione Openschool consente di:

- ▶ scaricare gratuitamente il libro digitale arricchito (eBook+) che permette in particolare di:
 - eseguire tutte le **esercitazioni** a risposta chiusa in modo **interattivo**;
 - accedere alle **gallerie di immagini**;
 - scaricare gli **approfondimenti tematici**, le **lezioni** e le **unità integrative**;
- ▶ disporre di ulteriori esercitazioni online utilizzabili a discrezione del docente per classi virtuali gestibili attraverso la **piattaforma didattica**.

Aspetti caratterizzanti

- ▶ Testo pienamente in linea con le recenti indicazioni ministeriali in merito alle nuove **caratteristiche tecniche e tecnologiche dei libri misti e digitali** e al loro stretto coordinamento con la **piattaforma didattica**.
- ▶ Totale **duttilità di utilizzo in funzione delle scelte didattiche o dotazioni tecnologiche**:
 - il libro cartaceo consente di svolgere lezioni complete e attività di laboratorio;
 - l'eBook+, le risorse online e la piattaforma offrono il pieno supporto per una didattica multimediale, a discrezione delle scelte del docente.
- ▶ **Lezioni autoconclusive** ricche di esempi ed esercizi, adatte a essere svolte in una lezione o al massimo due.
- ▶ **Teoria ridotta al minimo per privilegiare l'aspetto pratico**.

Materiali online hoepliscuola.it

Sul sito www.hoepliscuola.it sono disponibili numerose risorse online. In particolare, per lo studente: **approfondimenti**, utili integrazioni del testo e un numero elevato di **esercizi** sia per il **recupero e il rinforzo sia per l'approfondimento** degli argomenti trattati. Per il docente, una sezione riservata presenta alcune **unità didattiche per l'approfondimento** delle tematiche affrontate e un insieme di **schede aggiuntive** per la verifica dei livelli di apprendimento degli studenti, nonché **lezioni** (sotto forma di presentazioni in PowerPoint), **utilizzabili** efficacemente **anche con le LIM**. Materiali ed esercizi possono essere usati anche per creare attività didattiche fruibili tramite la piattaforma didattica accessibile dal sito.

Struttura dell'opera

UNITÀ DI APPRENDIMENTO

2

Comunicazione e sincronizzazione

LEZIONE 7

Problemi classici della programmazione concorrente: deadlock, banca e filosofi a cena

ZOOM SU...
Piccole sezioni di approfondimento

DEFINIZIONI
Spiegazione delle proprietà essenziali dei principali termini, funzioni e concetti trattati nel testo

OSSERVAZIONI
Un aiuto per comprendere e approfondire

LABORATORI
Per il rafforzamento dei concetti assimilati, attraverso esercitazioni operative

ESERCIZI
Ampia sezione di esercizi per la verifica delle conoscenze e delle competenze

LEZIONE 8
Esercitazioni di laboratorio

LA FORK IN C

DEFINIZIONI
Spiegazione delle proprietà essenziali dei principali termini, funzioni e concetti trattati nel testo

OSSERVAZIONI
Un aiuto per comprendere e approfondire

LABORATORI
Per il rafforzamento dei concetti assimilati, attraverso esercitazioni operative

ESERCIZI
Ampia sezione di esercizi per la verifica delle conoscenze e delle competenze

LEZIONE 9
Esercitazioni di laboratorio

Verifichiamo le conoscenze

2. Risposte multipla

3. Verifica di fatto

4. AREA didattica

PROVA ADESSO!
Per mettere in pratica, in itinere, quanto appreso nella lezione

LEZIONE 10
Esercitazioni di laboratorio

Verifichiamo le conoscenze

1. Risposte multipla

2. Verifica di fatto

3. Verifica di fatto

4. AREA didattica

LEZIONE 11
Esercitazioni di laboratorio

Verifichiamo le conoscenze

1. Risposte multipla

2. Verifica di fatto

3. Verifica di fatto

4. AREA didattica

L'OFFERTA DIDATTICA HOEPLI

L'edizione **Openschool** Hoepli offre a docenti e studenti tutte le potenzialità di Openschool Network (ON), il nuovo sistema integrato di contenuti e servizi per l'apprendimento.

Edizione **OPENSCHOOL**



LIBRO DI TESTO



Il libro di testo è l'**elemento cardine** dell'offerta formativa, uno strumento didattico **agile e completo**, utilizzabile **autonomamente** o in combinazione con il ricco **corredo digitale** offline e online. Secondo le più recenti indicazioni ministeriali, volume cartaceo e apparati digitali **sono integrati in un unico percorso didattico**. Le espansioni accessibili attraverso l'eBook+ e i materiali integrativi disponibili nel sito dell'editore sono puntualmente richiamati nel testo tramite apposite icone.

eBOOK+



L'eBook+ è la versione digitale e interattiva del libro di testo, utilizzabile su **tablet, LIM e computer**. Aiuta a comprendere e ad approfondire i contenuti, rendendo l'apprendimento più attivo e coinvolgente. Consente di leggere, annotare, sottolineare, effettuare ricerche e accedere direttamente alle numerose **risorse digitali integrative**. ➔ Scaricare l'eBook+ è molto **semplice**. È sufficiente seguire le istruzioni riportate nell'ultima pagina di questo volume.

RISORSE ONLINE



Il sito della casa editrice offre una ricca dotazione di **risorse digitali** per l'approfondimento e l'aggiornamento. Nella pagina web dedicata al testo è disponibile **MyBookBox**, il contenitore virtuale che raccoglie i materiali integrativi che accompagnano l'opera. ➔ Per accedere ai materiali è sufficiente registrarsi al sito www.hoepliscuola.it e inserire il codice coupon che si trova nella terza pagina di copertina. Per il docente nel sito sono previste ulteriori risorse didattiche dedicate.

PIattaforma DIDATTICA



La **piattaforma didattica** è un ambiente digitale che può essere utilizzato in modo duttile, a misura delle esigenze della classe e degli studenti. Permette in particolare di **condividere contenuti ed esercizi** e di partecipare a **classi virtuali**. Ogni attività svolta viene salvata sul **cloud** e rimane sempre disponibile e aggiornata. La piattaforma consente inoltre di consultare la versione online degli eBook+ presenti nella propria libreria. ➔ È possibile accedere alla piattaforma attraverso il sito www.hoepliscuola.it.

Indice

UNITÀ DI APPRENDIMENTO 1

Processi sequenziali e paralleli

L1 Il modello a processi

Il modello a processi.....	2
Stato di processi.....	4
Comandi per la creazione, sospensione e terminazione dei processi.....	6
PCB (Process Control Block).....	7
Verifichiamo le conoscenze.....	9

L2 Risorse e condivisione

Generalità.....	11
Classificazioni.....	13
Grafo di Holt.....	15
Verifichiamo le conoscenze.....	20
Verifichiamo le competenze.....	21

L3 I thread o "processi leggeri"

Generalità.....	22
"Processi pesanti" e "processi leggeri".....	23
Soluzioni adottate: single threading vs multithreading.....	27
Realizzazione di thread.....	27
Thread POSIX.....	29
Stati di un thread.....	29
Utilizzo dei thread.....	30
Verifichiamo le conoscenze.....	31
Verifichiamo le competenze.....	32

L4 Elaborazione sequenziale e concorrente

Generalità.....	33
-----------------	----

Processi non sequenziali e grafo di precedenza.....	35
Scomposizione di un processo non sequenziale.....	38
Verifichiamo le conoscenze.....	42
Verifichiamo le competenze.....	43

L5 La descrizione della concorrenza

Esecuzione parallela.....	44
Fork-join.....	45
Cobegin-coend.....	49
Equivalenza di fork-join e cobegin-coend.....	52
Semplificazione delle precedenze.....	54
Verifichiamo le competenze.....	56

Esercitazioni di laboratorio

 1 L'emulatore Cygwin.....	59
2 L'ambiente di sviluppo Dev-C++.....	
3 La fork in C.....	74
4 Fork annidate ed esecuzione non deterministica.....	80
5 Le funzioni wait() e waitpid().....	86
6 Fork-join e cobegin-coend.....	97
7 I thread in C.....	103
8 Thread e parametri.....	110
9 Thread in ambiente Dev-cpp e linux.....	118
10 I thread in Java: concetti base.....	122
11 Priorità e parametri nei thread Java.....	129
12 I thread Java: i metodi sleep, yield e join.....	134

Puoi scaricare il Lab. 2 anche da



AREA digitale



Esercizi



- ▶ Formalismo dei grafi di Holt
- ▶ I caratteri jolly
- ▶ Why should pid_t be used?
- ▶ Rilevazione dello stato di terminazione
- ▶ Valore di OPTIONS in una waitpid()
- ▶ Casting con warning
- ▶ Un ulteriore esercizio errato sulla memoria condivisa
- ▶ Thread e animazioni
- ▶ Esercizi per il recupero
- ▶ Esercizi per l'approfondimento

UNITÀ DI APPRENDIMENTO 2

Comunicazione e sincronizzazione

L1 La comunicazione tra processi

Comunicazione: modelli software e hardware.....	144
Modello a memoria comune (ambiente globale, global environment).....	145
Modello a scambio di messaggi (ambiente locale, message passing).....	148
Verifichiamo le conoscenze	150

L2 La sincronizzazione tra processi

Errori nei programmi concorrenti.....	152
Definizioni e proprietà.....	154
Proprietà non funzionali: safety e liveness.....	161
Conclusioni.....	162
Verifichiamo le conoscenze	163

L3 Sincronizzazione tra processi: semafori

Premessa: quando è necessario sincronizzare?.....	164
Semafori di basso livello e spin lock().....	165
Semafori di Dijkstra.....	169

Semafori binari vs semafori di Dijkstra	172
Verifichiamo le conoscenze	173
Verifichiamo le competenze	174

L4 Applicazione dei semafori

Semafori e mutua esclusione.....	176
Mutua esclusione tra gruppi di processi.....	178
Semafori come vincoli di precedenza	180
Problema del rendez-vous	181
Verifichiamo le competenze	183

L5 Problemi "classici" della programmazione concorrente: produttori/consumatori

Generalità.....	185
Problema dei produttori/consumatori.....	186
Un produttore, un consumatore e una singola cella di memoria.....	186
Verifichiamo le competenze	192

L6 Problemi "classici" della programmazione concorrente: lettori/scrittori

Problema dei lettori e degli scrittori.....	193
Verifichiamo le competenze	198

L7 Problemi "classici" della programmazione concorrente: deadlock, banchiere e filosofi a cena

Perché si genera un deadlock.....	200
Individuazione dello stallo.....	201
Come affrontare lo stallo.....	204
Esempio classico: problema dei filosofi a cena.....	209
Verifichiamo le conoscenze	210
Verifichiamo le competenze	211

L8 I monitor

Generalità.....	213
Utilizzo dei monitor.....	215
Variabili condizione e procedure di wait/signal.....	215
Emulazione di monitor con i semafori.....	218
Verifichiamo le competenze	220

L9 Lo scambio di messaggi

Generalità.....	221
Canali di comunicazione.....	222
Primitive di comunicazione asimmetrica da-molti-a-uno.....	223
Primitive di comunicazione asimmetrica da-molti-a-molti (cenni).....	225
Verifichiamo le conoscenze.....	226

Esercitazioni di laboratorio

- 1 La comunicazione tra processi mediante segnali asincroni..... 227
- 2 Thread e schedulazione..... 238
- 3 I semafori binari in C..... 243
- 4 La soluzione del deadlock dei filosofi in C con i mutex..... 248
- 5 La soluzione del problema produttori/consumatori con i semafori classici..... 251
- 6 Variabili condizione..... 262
- 7 I monitor con le variabili condition in C..... 269
- 8 I monitor con i semafori in C..... 274
- 9 I semafori in Java..... 279
- 10 I monitor in Java..... 285
- 11 Un esempio con i Java thread:
corsa di biciclette..... 291
- 12 Il deadlock in Java..... 296

AREA digitale



▶ Esercizi



- ▶ Classificazione di Flynn
- ▶ Applicazioni in real time
- ▶ Esempio riepilogativo (non informatico)
- ▶ Grafo di Holt e grafo di attesa
- ▶ Tabella dei segnali UNIX/LINUX
- ▶ Quando viene notificato/gestito un segnale
- ▶ Gruppi di processi
- ▶ Impostare una sveglia
- ▶ Nota per gli utenti MAC
- ▶ Soluzione del problema del produttore/consumatore con i mutex
- ▶ Mutex per la gestione di vincoli di precedenza
- ▶ Esercizi per il recupero
- ▶ Esercizi per l'approfondimento

UNITÀ DI APPRENDIMENTO 3

I requisiti software

L1 La specifica dei requisiti

Premessa.....	300
Requisiti software e stakeholder.....	301
Classificazione dei requisiti.....	304
I requisiti: l'anello debole dello sviluppo software.....	307
Verifichiamo le conoscenze.....	310
Verifichiamo le competenze.....	311

L2 Raccolta e analisi dei requisiti

Premessa.....	312
Tipi di raccolta dei requisiti.....	313
La fase di esplorazione.....	314
Problemi della fase di esplorazione.....	319
Verifichiamo le conoscenze.....	322

L3 Attori, casi d'uso e scenari

Introduzione.....	323
Tipi di scenari.....	327
Descrizione dei casi d'uso.....	328
Relazioni tra casi d'uso.....	331
Documentazione dei casi d'uso.....	334
Verifichiamo le competenze.....	337

L4 La documentazione dei requisiti

Generalità.....	339
Requirements Documents proposto da Sommerville.....	340
Realizzare un efficace documento SRS.....	343
Verifichiamo le competenze.....	346

Esercitazioni di laboratorio

- 1 La realizzazione degli Use Case Diagram con StarUML..... 347
- 2 La realizzazione degli Use Case Diagram con ArgoUML..... 354

AREA digitale



Esercizi



- ▶ Gli stakeholder
- ▶ La norma ISO 13407
- ▶ Non-functional requirements classification
- ▶ Esempio di questionario per lo sviluppo di una piattaforma di e-commerce
- ▶ Consigli per fare un buon Use Case Diagram
- ▶ Esercizi per il recupero
- ▶ Esercizi per il rinforzo

UNITÀ DI APPRENDIMENTO 4

Documentazione del software

L1 La documentazione del progetto

Generalità	358
Standard della documentazione	359
Documentazione del progetto	360
La documentazione esterna	360
Tool di documentazione	366
Verifichiamo le conoscenze	367



L2 La documentazione del codice

- Generalità
 - Il codice sorgente
 - Naming Guidelines
 - Commenti
 - Formattazione
 - Tool di indentazione automatica
- Verifichiamo le conoscenze**

Esercitazioni di laboratorio

- 1 La documentazione automatica con Javadoc 369
- 2 Il software Doxygen 375
- 3 Il controllo delle versioni con SubVersion e TortoiseSVN

Puoi scaricare la Lez. 2 e il Lab. 3 anche da hoepliscuola.it

AREA digitale



Esercizi

Come utilizzare il coupon per scaricare la versione digitale del libro (eBook+) e i contenuti digitali integrativi (risorse online) 390

1

Processi sequenziali e paralleli

- L1 **Il modello a processi**
- L2 **Risorse e condivisione**
- L3 **I thread o “processi leggeri”**
- L4 **Elaborazione sequenziale e concorrente**
- L5 **La descrizione della concorrenza**

Esercitazioni di laboratorio

- ➊ L'emulatore Cygwin; ➋ L'ambiente di sviluppo Dev-C++;
- ➌ La fork in C; ➍ Fork annidate ed esecuzione non deterministica;
- ➎ Le funzioni wait() e waitpid(); ➏ Fork-join e cobegin-coend;
- ➐ I thread in C; ➑ Thread e parametri; ➒ Thread in ambiente Dev-cpp e linux; ➓ I thread in Java: concetti base;
- ➔ Priorità e parametri nei thread Java; ➕ I thread Java: i metodi sleep, yield e join

Conoscenze

- Conoscere i modelli di elaborazione dei processi
- Conoscere ciclo di vita dei processi
- Acquisire il concetto di risorsa condivisa
- Distinguere le richieste e le modalità di accesso alle risorse
- Apprendere l'utilizzo del grafo di Holt per descrivere processi e risorse
- Conoscere la differenza tra processi e thread
- Acquisire il concetto di programmazione concorrente
- Acquisire il concetto di interazione tra processi
- Conoscere le caratteristiche di un linguaggio concorrente

Competenze

- Descrivere l'interazione processi-risorse col grafo di Holt
- Realizzare e semplificare il grafo delle precedenze
- Scrivere programmi concorrenti utilizzando l'istruzione fork-join
- Scrivere programmi concorrenti utilizzando l'istruzione cobegin-coend
- Eseguire un programma C in Cygwin
- Scrivere programmi multiprocessi in linguaggio C

Abilità

- Installare e configurare il software Cygwin
- Compilare i programmi C col compilatore GCC
- Utilizzare in thread in linguaggio C
- Utilizzare in thread in linguaggio Java

AREA digitale



Esercizi



- Formalismo dei grafi di Holt
- I caratteri jolly
- Why should pid_t be used?
- Rilevazione dello stato di terminazione
- Valore di OPTIONS in una waitpid()
- Casting con warning
- Un ulteriore esercizio errato sulla memoria condivisa
- Thread e animazioni
- Esercizi per il recupero
- Esercizi per l'approfondimento



Soluzioni (prova adesso, esercizi, verifiche)
Puoi scaricare il file anche da

Il modello a processi

In questa lezione impareremo...

- ▶ i modelli di elaborazione dei processi
- ▶ il ciclo di vita dei processi

■ Il modello a processi

Nonostante l'evoluzione tecnologica abbia incrementato le capacità computazionali grazie all'aumento della velocità delle **CPU**, la gestione del **processore** ancora oggi deve essere ottimizzata perché spesso presenta **situazioni di criticità**: tutti i moderni **SO** cercano di sfruttare al massimo le potenzialità di parallelismo fisico dell'hardware per minimizzare i tempi di risposta e aumentare il **throughput** del sistema, ossia il **numero di programmi eseguiti per unità di tempo**.

Il **programma**, composto da un insieme di byte contenente le istruzioni che dovranno essere eseguite, è un'**entità passiva** finché non viene caricato in memoria e mandato in esecuzione diventando un **processo**; evolve man mano che le istruzioni vengono eseguite dalla **CPU** e diviene quindi un'**entità attiva**: il **processo** è l'**istanza** di un **programma in esecuzione**.

Possiamo dare come definizione sintetica di **processo** quella riportata di seguito.



PROCESSO

Un **processo** è un'**entità logica (programma)** in evoluzione.

Nel **modello a processi** tutto il software che può essere eseguito su di un calcolatore, compreso il **Sistema Operativo** stesso, è organizzato in un certo numero di **processi sequenziali**

(successivamente chiamati semplicemente **processi**): un unico processore può essere condiviso tra parecchi **processi**, utilizzando un algoritmo di scheduling (di **scheduling**) per determinare quando interrompere l'evoluzione di un processo e servirne un altro.

Questa tecnica di gestione della **CPU** si chiama **multiprogrammazione** e richiede la contemporanea presenza di più programmi in memoria: dato che l'esecuzione di un programma, quindi un processo, è costituita da una successione di fasi di elaborazione della **CPU** e fasi di attesa per l'esecuzione di operazioni su altre risorse del sistema (operazioni di I/O, di caricamento dati, di colloquio con periferiche ecc.) che di fatto lasciano inattiva la **CPU**, la **multiprogrammazione** permette l'evoluzione contemporanea di più **processi** limitando al minimo i tempi morti e sfruttando appieno le potenzialità di calcolo del processore.

Inoltre i **processi** possono essere **indipendenti** oppure **cooperare** per raggiungere un medesimo obiettivo:

- nel primo caso, cioè di **processi indipendenti**, un **processo** evolve in modo autonomo senza bisogno di comunicare con gli altri **processi** per scambiare dati;
- nel secondo caso, due (o più) **processi** hanno la necessità di **cooperare** in quanto, per poter evolvere, necessitano di scambiarsi informazioni.

ESEMPIO

Si pensi a tutti i videogame con due o più giocatori dove ogni giocatore è un **processo**: in questo caso nasce la necessità per i due **processi** di **coordinarsi** per poter comunicare e scambiarsi le informazioni (i **processi** si devono **sincronizzare**).

La possibilità di avere **processi** che **cooperano**, in sintesi, è utile per ottenere:

- **parallelizzazione** dell'esecuzione (per esempio macchine con multi-cpu);
- **replicazione di un servizio** (per esempio connessioni di rete);
- **modularità** di diversi processi per funzioni diverse di una stessa applicazione, come per esempio il correttore ortografico di Word (posso continuare a scrivere mentre corregge, cioè posso compiere più azioni contemporaneamente);
- **condivisione** delle informazioni.

Oltre alla **cooperazione** esiste un'altra forma di **interazione** tra **processi**: due (o più) processi possono **ostacolarsi a vicenda** compromettendo il buon fine delle loro elaborazioni.

È il caso in cui entrambi i **processi competono** per utilizzare la medesima **risorsa**, che magari è in quantità limitata nel sistema: questo tipo di interazione può portare a situazioni indesiderate per uno o per entrambi i **processi** (**blocco individuale o critico**).

ESEMPIO

L'esempio più semplice di **competizione** nella multiprogrammazione è dato dallo **scheduling dei processi**, dove tutti competono per la **CPU**; un'altra **risorsa** che è sempre condivisa è la stampante, e per accedervi i **processi** devono attendere in coda (competono per la **risorsa stampante**).

Possiamo ora dare una definizione per le due situazioni di **interazione tra processi**:



PROCESSI COOPERANTI

Un **processo** è **cooperante** se influenza o può essere influenzato da altri processi in esecuzione nel sistema (un processo condivide dati con altri processi).



PROCESSI DI COMPETIZIONE

Due **processi** sono in **competizione** se possono evolvere indipendentemente ma entrano in conflitto sulla ripartizione delle risorse.

Riassumendo, abbiamo quindi tre **modelli di computazione per i processi**:

- ▷ modello di computazione **indipendente**;
- ▷ modello di computazione con **cooperazione**;
- ▷ modello di computazione con **competizione**

In questo volume analizzeremo le possibili forme di interazione, desiderate e indesiderate, e affronteremo il progetto di applicazioni concorrenti scrivendone la codifica in linguaggio di programmazione.

■ Stato dei processi

Durante il ciclo di vita di un processo è possibile individuare un insieme di situazioni in cui il **processo** può trovarsi, che definiremo come gli **stati di un processo**, associati alla sua evoluzione e alla sua “situazione” rispetto alla **CPU**.

Vediamo dettagliatamente come può trovarsi un **processo** rispetto al processore:

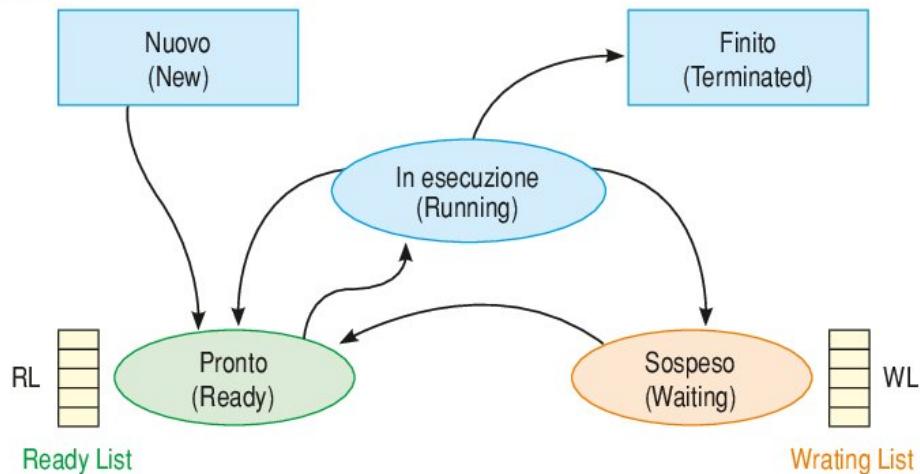
- ▷ **nuovo (new)**: è lo stato in cui si trova un processo appena è stato creato, cioè l'utente richiede l'esecuzione di un programma che risiede sul disco;
- ▷ **esecuzione (running)**: il **processo** sta evolvendo, nel senso che la **CPU** sta eseguendo le sue istruzioni, e quindi a esso è assegnato il processore. Nei sistemi a monoprocesso (quelli analizzati in questo testo) un solo **processo** può essere in questo stato;
- ▷ **attesa (waiting)**: un **processo** è nello stato di attesa quando gli manca una **risorsa** per poter evolvere (oltre alla **CPU**) e quindi sta *aspettando* che si verifichi un evento (per esempio che si liberi la risorsa che gli serve e che il processore gliela assegna);
- ▷ **pronto (ready-to-run)**: un **processo** è nello stato di pronto se ha tutte le **risorse** necessarie alla sua evoluzione tranne la **CPU** (cioè è il caso in cui sta aspettando che gli venga assegnato il suo time-slice di **CPU**);
- ▷ **finito (terminated)**: siamo nella situazione in cui tutto il codice del **processo** è stato eseguito e quindi ha determinato l'esecuzione; il sistema operativo deve ora rilasciare le **risorse** che utilizzava.



STATO DI UN PROCESSO

Con **stato di un processo** intendiamo quindi una tra le cinque possibili situazioni in cui un processo in esecuzione può trovarsi: può assumere una sola volta lo stato di **nuovo** e di **terminato**, mentre può essere per più volte negli altri tre stati.

Vediamo come è possibile rappresentare mediante un grafico orientato i passaggi che un processo esegue tra i possibili stati sopra descritti: questo grafico prende il nome di **diagramma degli stati**.



Seguiamo ora la **vita di un processo** dal principio: al **nuovo** processo viene assegnato un identificatore (**PID, Process IDentifier**) e viene inserito nell'elenco dei processi **pronti** (**RL, Ready List**) in attesa che arrivi il suo turno di utilizzo della **CPU**.

Ogni processo è creato da un altro processo detto **"padre"** mentre lui prende il nome di **"figlio"**: l'unica eccezione è il processo **"init"**, cioè il primo a essere eseguito dal S.O, che non ha nessun **"padre"**.

Quando gli viene assegnata la **CPU**, il processo passa nello stato di **esecuzione**, dal quale può uscire per tre motivi:

- ▶ termina la sua esecuzione, cioè il processo esaurisce il suo codice e quindi **finisce** (**exit**);
- ▶ termina il suo tempo di **CPU**, cioè il suo **quanto di tempo**, e quindi ritorna nella lista dei **processi pronti RL (ready list)**;
- ▶ per poter evolvere necessita di una risorsa che al momento non è disponibile: il processo si **sospende** (**suspend**) e passa nello stato di attesa, insieme ad altri processi, formando la **waiting list WL**.

Quindi durante l'**evoluzione** il **processo** può trovarsi in uno dei seguenti **tre stati**:

- ▶ **in esecuzione** (**running**);
- ▶ **pronto** (**ready**);
- ▶ **sospeso** (**waiting**), anche detto **bloccato** (**blocked**), che è lo stato in cui si trova quando non può ottenere la **CPU** anche se questa è libera, poiché è in attesa di qualche evento esterno o risorsa che al momento non è disponibile.

Dallo stato di **sospeso**, cioè dallo stato di **attesa**, un processo **non** può passare in quello di **esecuzione**: infatti, quando si rende disponibile la risorsa che sta **"aspettando"**, viene spostato dalla **WL** ma viene inserito nelle **RL**, cioè nella lista dei processi pronti ad accedere alla **CPU**. Quando arriverà il suo turno, gli verrà assegnato il processore e solo allora potrà evolvere.

Sospensione per interrupt

Il SO ha un diverso comportamento nel caso che il processo venga sospeso a causa di una interruzione associata a un dispositivo I/O rispetto alla sospensione dovuta allo scadere del **time slice**: gli **interrupt** dovuti ai dispositivi di I/O sono organizzati in classi e a essi viene associata una locazione, spesso vicina alla parte bassa della memoria, chiamata **interrupt vector**, che contiene l'indirizzo della **procedura di gestione delle interruzioni**.

Se si verifica un'interruzione (causata per esempio da problemi sul disco fisso) quando è in esecuzione il processo XXX, il **program counter** di questo processo, la **parola di stato** del programma e la **maschera delle interruzioni** (a volte anche uno o più registri) vengono messi sullo stack corrente dall'hardware dedicato alle interruzioni e la **CPU** salta all'indirizzo specificato nell'**interrupt vector** che provvede a completare il salvataggio dello **stato del processo** prima di eseguire la routine di risposta all'interruzione.

Quando termina la gestione dell'**interrupt**, viene richiamato lo **scheduler** per vedere quale processo deve essere mandato in esecuzione e un piccolo programma assembler esegue il caricamento dei registri e la mappa di memoria per il nuovo **processo corrente**.



CONTEXT SWITCHING

Tutte queste azioni di salvataggio e ripristino vengono chiamate **context switching**.

■ Comandi per la creazione, sospensione e terminazione dei processi

Nei sistemi ◀ *NIX Like ▶ la **creazione** di un processo avviene mediante la funzione:

```
pid fork();
```

All'atto della chiamata viene generato un nuovo **PID** per il figlio e un nuovo descrittore del processo **PCB**, vengono copiati nella memoria del nuovo processo il segmento dati di sistema in modo da avere due copie di segmenti identiche, dato che il "figlio" è un clone del "padre": l'unica differenza è il valore restituito dalla **fork()** stessa.



◀ *NIX Like Con *NIX si intendono i sistemi operativi **UNIX** e **XENIX** che hanno parecchie analogie e con *NIX Like i sistemi operativi che sono loro "somiglianti", come Linux che sappiamo essere direttamente derivato da **UNIX**. ▶

Il codice del processo padre viene condiviso dal figlio dal punto in cui viene invocata la **fork()** e quindi a partire da questa istruzione i due processi evolvono in parallelo eseguendo lo stesso codice: l'istruzione di **fork()** ritornerà il valore a entrambe le istanze del programma:

- 1 la prima istanza è il processo padre, con il valore del **PID** che viene assegnato dal SO al figlio;
- 2 la seconda istanza è il processo figlio appena creato, con valore pari a **0**.

La **terminazione** di un processo avviene mediante la funzione:

```
void exit(int status);
```

Alla sua chiamata vengono eseguite le operazioni di chiusura dei file aperti, viene rilasciata la memoria, viene salvato il valore dell'exit status nel descrittore del processo in modo che potrà essere recuperato dal padre mediante le funzione `wait()` (di seguito descritta).

È importante osservare come il **PID** non viene rilasciato e il descrittore non viene distrutto, ma viene segnalata al processo padre la terminazione di un figlio.

Un **processo padre** può **sospendere la propria attività** in attesa che il figlio termini con l'istruzione:

```
pid wait(int* status);
```

Alla chiamata di questa funzione il **processo padre** si sospende in attesa della terminazione di un **processo figlio**: quando questo avviene, recupera il valore dello stato di uscita specificato dalla funzione `exit()` nel figlio.

In particolare il valore che viene restituito è composto da due byte:

- ▷ nel byte meno significativo della variabile viene indicata la ragione della terminazione, che può essere stata naturale o mediante un segnale;
- ▷ nel byte più significativo della variabile viene scritto il valore dello stato di uscita specificato nella funzione `exit()` del figlio.

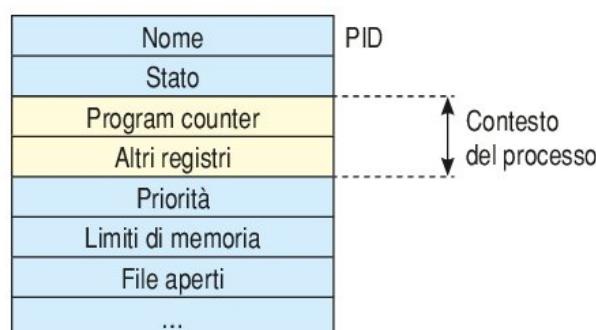
Il **SO** rilascia il **PID** del figlio e rimuove il suo descrittore di processo.

Esiste anche la possibilità che un processo figlio termini prima che il padre abbia invocato la funzione `wait()`, il processo figlio diventa "**defunct**" o "**zombie**": in questo caso il processo è terminato ma il descrittore non può essere rilasciato.

■ PCB (Process Control Block)

Concludiamo questa sintesi sui processi ricordando la struttura del descrittore del processo **PCB (Process Control Block)**:

- ▷ identificatore unico (**PID**);
- ▷ stato corrente;
- ▷ program counter;
- ▷ registri;
- ▷ priorità;
- ▷ puntatori alla memoria del processo;
- ▷ puntatori alle risorse allocate al processo.



Sappiamo che il **program counter** e i registri formano il **contesto del processo**: questi campi prendono anche il nome di **area di salvataggio dello stato della CPU**.

Oltre a queste informazioni sono anche presenti dati che riguardano *informazioni per l'accounting e per lo stato dell'I/O*, che riportano la lista dei file e delle periferiche associati al processo.

Il **descrittore di processo** viene allocato dinamicamente all'atto della creazione e opportunamente inizializzato viene rimosso dopo le operazioni di terminazione del **processo**.

Per implementare il **modello a processi**, il SO mantiene una tabella chiamata **Process Table**, contenente tutti i **PCB** dei singoli processi, in modo da avere sempre a disposizione le informazioni sullo stato del processo, aggiornandola quando il processo passa da uno **stato di esecuzione** a uno **stato di pronto** o **bloccato**, in maniera che possa essere fatto ripartire più tardi come se non fosse mai stato fermato.



Prova adesso!

- Processi attivi in esecuzione
- Priorità dei processi

Individua sul tuo computer l'elenco dei processi attivi e individua le regole di attribuzione della priorità ai processi stessi.

Li puoi individuare a partire dal **Pannello di controllo** tra gli *Strumenti di amministrazione* (System information – Attività in esecuzione).

Verifichiamo le conoscenze



1. Risposta multipla

1 Quale di queste affermazioni è errata?

- a) il programma è un'entità passiva
- b) il programma è un'entità attiva
- c) il processo è un'entità passiva
- d) il processo è un'entità attiva

2 In riferimento alla multiprogrammazione, quale delle seguenti affermazioni è errata?

- a) richiede la contemporanea presenza di più programmi in memoria
- b) permette la contemporanea esecuzione di più processi
- c) permette la contemporanea esecuzione di più istanze di un programma
- d) permette la contemporanea esecuzione di più istanze di un processo

3 Abbiamo tre modelli di computazione per i processi (indica quelli errati):

- a) modello di computazione indipendente
- b) modello di computazione dipendente
- c) modello di computazione con cooperazione
- d) modello di computazione con competizione
- e) modello di computazione con integrazione

4 In quale situazione può trovarsi un processo rispetto al processore (indica quella inesatta)?

- a) nuovo (new)
- b) esecuzione (running)
- c) attesa (sleeping)
- d) pronto (ready-to-run)
- e) finito (terminated)

5 Nella multiprogrammazione più processi possono essere in running contemporaneamente:

- a) solo nei sistemi multiprocessori
- b) solo nei sistemi con schedulazione
- c) sì, sempre
- d) no, mai

6 Dallo stato di esecuzione un processo può uscire per tre motivi (indica quello inesatto):

- a) termina la sua esecuzione
- b) produce un risultato errato (tipo divisione per 0)
- c) termina il suo quanto di tempo
- d) gli manca una risorsa per evolvere

7 In caso di interrupt da periferica, l'hardware effettua il salvataggio:

- a) del program counter
- b) dei dati del processo
- c) della parola di stato del programma
- d) della maschera delle interruzioni
- e) dello stato del processo

8 Cosa significa l'acronimo PCB?

- | | |
|--------------------------|----------------------|
| a) Program Control Block | c) Process CPU Block |
| b) Process Control Block | d) Program CPU Block |



2. Vero o falso

- | | |
|---|-----|
| 1 Con throughput di sistema si intende il numero di programmi eseguiti per unità di tempo. | V F |
| 2 Un processo è un'entità logica in evoluzione. | V F |
| 3 Lo scheduling permette a più programmi di evolvere contemporaneamente. | V F |
| 4 Un processo è nello stato di pronto se ha tutte le risorse necessarie alla sua evoluzione. | V F |
| 5 Un processo è nello stato di attesa quando gli mancano almeno due risorse per poter evolvere. | V F |
| 6 Al nuovo processo viene assegnato un identificatore (PID, Process IDentifier). | V F |
| 7 Dallo stato di sospeso un processo può passare in quello di esecuzione se si rende disponibile la risorsa che sta "aspettando". | V F |
| 8 L'interrupt vector contiene l'indirizzo delle interruzioni. | V F |
| 9 La tabella chiamata Process Table contiene tutti i PCB dei singoli processi. | V F |
| 10 In *nix, la fork ritorna al processo figlio appena creato il valore PID=0. | V F |
| 11 Un zombie è un processo figlio senza il padre. | V F |

3. Completamento

- 1 Disegna il diagramma degli stati di un processo descrivendone le singole transazioni.
-
-
-

Risorse e condivisione

In questa lezione impareremo...

- ▶ il concetto di risorsa condivisa
- ▶ le richieste e le modalità di accesso alle risorse
- ▶ il grafo di Holt per descrivere processi e risorse

■ Generalità

I **processi** sono i programmi in evoluzione e per poter evolvere hanno bisogno delle **risorse** del sistema di elaborazione: possiamo proprio vedere il sistema di elaborazione come composto da un insieme di **risorse** da assegnare ai **processi** affinché questi possano svolgere il proprio compito. Una prima definizione di **risorsa** è la seguente:



RISORSA

Ogni componente riusabile o meno, sia hardware, sia software necessario al processo o al sistema.

Per accedere alle **risorse** i **processi** sono in **competizione** in quanto spesso esse non sono sufficienti per tutti e quindi è necessario “accaparrarsene” per poterle utilizzare; per esempio i **processi competono** per avere a disposizione la memoria **RAM**, per utilizzare l’interfaccia di rete, le stampanti ecc., e soprattutto il processore che, nelle nostre architetture, è singolo e senza di esso nessun **processo** può evolvere.

Le **risorse** possono essere suddivise in **classi** e le **risorse** appartenenti alla stessa classe sono equivalenti, per esempio bytes della memoria, stampanti dello stesso tipo ecc.



CLASSE E ISTANZE

Le **risorse** di una **classe** vengono dette **istanze della classe**; il numero di **risorse** in una **classe** viene detto **molteplicità del tipo di risorsa**.

Quando un **processo** necessita di una **risorsa** generalmente non può richiedere *una particolare risorsa* ma solo una “generica” istanza di quella specifica **classe**: quindi una **richiesta di risorse** viene fatta per una **classe di risorse** e può essere soddisfatta da parte del **SO** assegnando al richiedente una qualsiasi istanza di quel tipo.

In altre parole, la molteplicità di una **risorsa** ci indica il numero massimo di **processi** che la possono usare contemporaneamente: se il numero di **processi** è maggiore della molteplicità di una **risorsa**, questa deve essere **condivisa** tra i **processi** che vi accedono **concorrentemente**.

ESEMPIO

Abbiamo detto che in un **PC** è presente un solo processore, quindi la molteplicità della **risorsa** processore è uguale a uno: il numero massimo di **processi** che possono *evolvere contemporaneamente* è quindi **uno** e quando abbiamo la necessità di far evolvere più **processi** assieme, questi condividono l'unica **istanza** della **risorsa** e **competono** per ottenerla.

Condivisione e gestione

Cerchiamo di chiarire meglio il concetto di **condivisione** prendendo spunto da una “legge ferroviaria” del secolo scorso:

◀ “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone” Legge del Kansas ►



In questo caso i due treni **condividono** l'incrocio e **competono** per poterlo avere a disposizione. Gli esempi nella vita quotidiana di condivisione sono molteplici a partire da quelli di ”natura stradale o ferroviaria” (incroci, posteggi, ponti ecc.) a quelli di natura sociale (“lo stesso bagno”, “lo stesso tetto”, “la stessa bandiera”, “la stessa causa”, la stessa opinione” ecc.).

Possiamo dire che **condividere** è sinonimo di “**avere in comune**” e quando parliamo di **risorse di elaborazione** intendiamo **dispositivi hardware** o **componenti software** che devono essere assegnati alternativamente ai singoli **processi** che le richiedono.

Ma la **condivisione** offre anche un meccanismo che permette ai **processi** di scambiarsi delle informazioni: se per esempio due **processi** condividono una **area di memoria** dove sono definite alcune variabili, possiamo “trasferire” tramite esse il risultato delle elaborazioni del primo processo al secondo processo, cioè possiamo far **cooperare i processi**, e questo tipo di meccanismo prende il nome di **modello di cooperazione a memoria comune**.

È quindi necessaria una **gestione delle risorse** che può essere organizzata in fasi, alcune delle quali sono di natura **statica** e riguardano la loro assegnazione (**pianificazione**), mentre altre sono di natura **dinamica** e sono relative al loro utilizzo (**controllo**):

► **pianificazione** della organizzazione:

- allocazione;
- disponibilità;
- costo;

D controllo delle risorse:

- controllo di accesso (locale o remoto);
- ottimizzazione;
- autenticazione;
- controllo di correttezza operazioni ed eccezioni.

Le attività sopra elencate vengono svolte dal **sistema operativo** e per le risorse di molteplicità finita è necessario controllare gli accessi a ciascuna di esse in modo che il loro utilizzo risulti costruttivo.

Per ogni **risorsa** il **SO** mette a disposizione:

- D un **gestore** della risorsa, che è un programma che ne regola il suo utilizzo;
- D un **protocollo di accesso** alla risorsa, che consiste nella procedura con la quale un processo effettua la *richiesta* della risorsa, la *ottiene* e quindi la *utilizza* e alla fine la *rilascia* affinché gli altri processi la possano utilizzare.

■ Classificazioni

Tra processi e risorse esiste un legame molto stretto:

I processi **competono** nell'accesso alle **risorse**, effettuando delle **richieste** per ottenere l'**assegnazione** di quanto gli necessita per poter **evolvere**.

In merito alla interazione tra risorse e processi possiamo effettuare la classificazione in base:

- D al tipo di **richieste**;
- D alla modalità di **assegnazioni**;
- D alla tipologia delle **risorse**.

Classificazione delle richieste

Le richieste possono essere classificate secondo vari criteri.

A secondo il numero:

- 1 **singola**: la richiesta singola è il caso normale e si riferisce a una singola **risorsa** di una classe definita, cioè un **processo** richiede una **sola risorsa alla volta**.
- 2 **multipla**: si riferisce a una o più classi, e per ogni classe, a una o più **risorse** e deve essere soddisfatta integralmente; è il caso in cui un **processo** richieda contemporaneamente almeno **due risorse** per poter evolvere.

B secondo il tipo di richiesta che effettuano:

- 1 **richiesta bloccante**: è il caso in cui il processo necessita immediatamente di quella **risorsa** e se non gli viene assegnata immediatamente (in quanto occupata e quindi già in situazione di utilizzo da parte di altri processi) si **sospende**, passa nello stato di **attesa** e la sua richiesta viene accodata e riconsiderata dal gestore di quella **risorsa** ogni volta che viene rilasciata dal **processo** che la sta utilizzando.
- 2 **richiesta non bloccante**: in questo caso il **processo** può evolvere ugualmente e nel caso di mancanza di disponibilità gli viene effettuata una notifica che il **processo** richiedente esamina ma continuando la sua evoluzione senza **cioè sospendere** la propria elaborazione.

Classificazione dell'assegnazione

L'**assegnazione** delle risorse avviene in due modalità:

- 1 **statica**: l'assegnazione **statica** di una **risorsa** a un processo avviene al momento della creazione del processo stesso e rimane a esso “**dedicata**” fino alla sua terminazione; l'esempio più significativo è il descrivitore di processo oppure l'area di memoria **RAM** nella quale è caricato (se non viene effettuato lo swapping);
- 2 **dinamica**: è il caso più frequente e generale nella naturale vita di un **processo** che avviene soprattutto per le **risorse condivise** che i processi **richiedono** durante la loro esistenza, le **utilizzano** quando sono a loro assegnate e le **rilasciano** quando non sono più necessarie oppure alla loro terminazione (esempio tipico sono le periferiche di I/O).

Classificazione delle risorse

Anche le risorse possono sottostare a varie classificazioni e tra esse ricordiamo le più importanti:

A in base alla mutua esclusività

- 1 **risorse seriali**: è il caso di **risorse** che **non possono** essere assegnate a più processi **contemporaneamente** ma questi devono attendere il loro turno per poterle utilizzare (si devono mettere in coda, cioè in “serie”, uno dietro all’altro); questo tipo di risorsa si dice che ha **accesso mutuamente esclusivo** da parte dei processi, in quanto quando ne entra in possesso un processo gli altri devono aspettare che questo la rilasci per poterla utilizzare. Esempi tipici di risorsa con accesso seriale sono le stampanti e i **CD-ROM**.
- 2 **risorse non seriali**: ammettono l'accesso **contemporaneo** di più processi e quindi possono considerarsi risorse di molteplicità **infinita**, come per esempio i file a sola lettura, che possono essere letti contemporaneamente da un numero qualsivoglia di processi.

B in base alla modalità di utilizzo

- 1 **risorse prerilasciabili**: si dice **prerilasciabile** o **preemptable** una risorsa che mentre viene utilizzata da un **processo** può essere “liberata”, cioè può essere sottratta al **processo prima** che abbia **terminato di utilizzarla**, senza che questo fatto danneggi il lavoro che stava effettuando e, pertanto, nel momento che gli viene restituita, esso può riprendere il lavoro dal punto in cui è stato interrotto senza “subire danni”.

Il processo che subisce il prerilascio forzato (o anticipato) deve sospendersi; la risorsa prerilasciata sarà successivamente restituita a quel processo che riprenderà la sua evoluzione dal punto in cui l’aveva interrotta.

Affinché una **risorsa** sia prerilasciabile deve avere le seguenti caratteristiche:

- il suo stato non si modifica durante l'utilizzo;
- il suo stato può essere facilmente salvato e ripristinato.

Gli esempi più “classici” di **risorsa preemptive** sono il **processore** e le aree di **memoria**.

Possiamo quindi definire una **risorsa preemptive** o **rilasciabile** come

RISORSA PREEMPTIVE O PRERILASCIABILE

Una risorsa si dice **prerilasciabile** se il suo gestore può **sottrarla** a un processo prima che questo l’abbia effettivamente rilasciata.

- 2 risorse non prerilasciabili:** una risorsa si dice **non prerilasciabile** o **non-preemptive** se una volta assegnata a un **processo** non è possibile “sottrargliela” senza che si provochi un danno al compito che esso sta eseguendo, con il pericolo di dover ripetere completamente la sua esecuzione.

Esempi tipici di risorse **non-preemptive** sono la stampante e il masterizzatore: se interrompiamo il processo che le sta utilizzando, molto probabilmente viene danneggiato, se non del tutto compromesso, il lavoro in fase di svolgimento.

I discorsi che abbiamo fatto si riferiscono alle **risorse** che i **processi** devono **condividere**, cioè **risorse comuni**: un **processo** può inoltre avere delle **risorse private** che esulano da questa trattazione dato che vengono assegnate da **SO** in modo esclusivo al **processo** stesso e non sono neppure visibili agli altri processi e quindi non richiedono di essere gestite in condivisione.

■ Grafo di Holt

Holt nel 1972 ha proposto un sistema di rappresentazione mediante un **grafo** che da lui ha preso il nome (grafo di Holt anche chiamato **grafo di allocazione risorse** o **grafo delle attese**) che permette di rappresentare tutte le situazioni in cui si possono venire a trovare i processi e le richieste di risorse, ed è particolarmente utile, come vedremo nelle prossime lezioni, per individuare situazioni di criticità tra processi e risorse.

Risorse e processi costituiscono due **sottoinsiemi** e sono rappresentati mediante nodi di due tipi:

- di forma **quadrata** le **risorse** (o di forma **rettangolare** nel caso di **classi di risorsa** e/o con **risorsa multipla**);
- di forma **rotonda** (cerchi) i **processi**.

Tra di essi vengono effettuati collegamenti orientati mediante archi:

- l'arco che connette una risorsa a processo indica che la risorsa è **assegnata** al processo



- l'arco che connette un processo a una risorsa indica che il processo **ha richiesto** la risorsa e che non gli viene assegnata dato che al momento della richiesta questa non è disponibile.



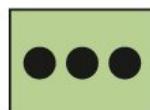
In questo modo si realizza un **grafo orientato diretto** (◀ **directed graph** ▶), con gli archi che hanno una sola direzione, e **bipartito**, in modo che non esistano archi che collegano nodi dello stesso sottoinsieme: gli **archi** possono solo connettere **nodi di tipo diverso**.

◀ **Directed graphs** ▶. The directed graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. ►



Se sono presenti **più istanze** della medesima classe di **risorse**, si effettua un partizione della risorsa stessa indicando la molteplicità con dei **punti** all'interno del box della **risorsa** (**Grafo di Holt generale**).

Un esempio di risorsa con molteplicità 3 è riportato sotto:

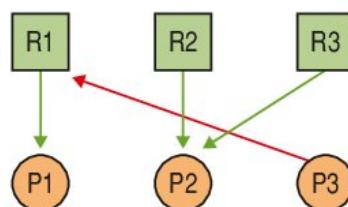


ESEMPIO

Nel primo esempio abbiamo tre processi (p1, p2 e p3) e tre risorse (R1, R2 e R3) con molteplicità 1 che durante la loro evoluzione generano la seguente situazione:

```
P1 richiede R1          //gli viene assegnata
P2 richiede R2          //gli viene assegnata
P2 richiede R3          //gli viene assegnata
P3 richiede R1          //NON gli viene assegnata, P3 rimane in attesa
```

La rappresentazione mediante il grafo di **Holt** è la seguente:



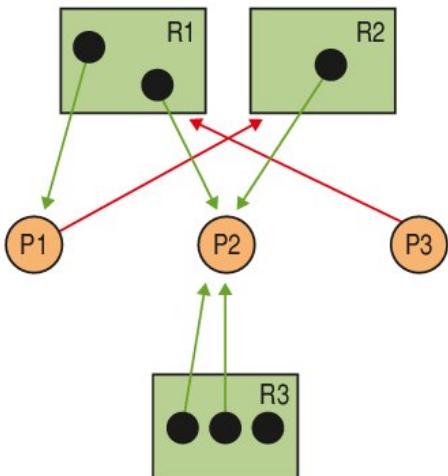
Supponiamo ora di avere classi di risorse con molteplicità diversa, per esempio:

- ▶ la classe R1: molteplicità 2
- ▶ la classe R2: molteplicità 1
- ▶ la classe R3: molteplicità 3

e la situazione è la seguente:

```
P1 richiede R1          //gli viene assegnata
P2 richiede R1          //gli viene assegnata
P2 richiede R2          //gli viene assegnata
P2 richiede R3          //gli viene assegnata
P2 richiede R3          //gli viene assegnata
P3 richiede R1          //NON gli viene assegnata, P3 rimane in attesa
P1 richiede R2          //NON gli viene assegnata, P1 rimane in attesa
```

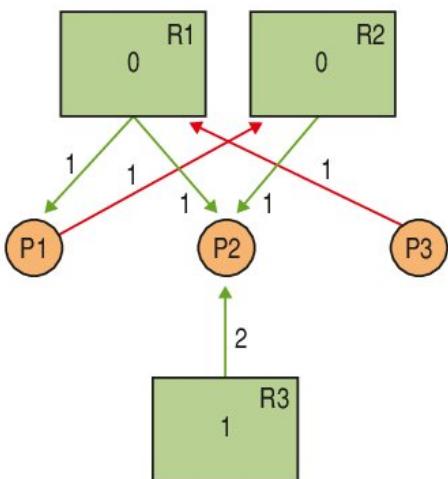
La rappresentazione mediante il grafo di **Holt** è quindi:



Alcuni autori utilizzano una **rappresentazione alternativa** per indicare la **molteplicità di risorsa** utilizzata/richiesta da un processo indicando con un numero sulla **freccia il grado di molteplicità** e all'interno della classe il numero di risorse non ancora assegnate, cioè quante istanze di quella classe sono ancora disponibili.

ESEMPIO

L'esempio precedente sarebbe così rappresentato:



È importante sottolineare come nei grafi di Holt non si rappresentino le richieste che possono essere soddisfatte ma solo **quelle pendenti**: quindi le frecce uscenti dai processi verso le risorse indicano le "risorse mancanti" ai processi per evolvere, che sono in quel momento assegnate ad altri processi.

AREA digitale



Formalismo dei grafi di Holt

Riducibilità di un grafo di Holt

Spesso è utile avere una visione della situazione tra **processi** e **risorse** “spostata in avanti nel tempo”, cioè trasformare il grafo di **Holt** in un grafo chiamato *ridotto* nel quale sono state tolte le situazioni in cui un **processo** è in grado di evolvere e che quindi sicuramente libererà le **risorse** che sta utilizzando a favore degli altri **processi**: siamo nel caso in cui un processo non attende nessuna risorsa e quindi graficamente **ha solo archi entranti** (risorse assegnate) e **non ha archi uscenti** (richieste in sospeso).

Il concetto fondamentale che sta alla base della riduzione di un grafo è la certezza che un processo che non ha bisogno di altre risorse per evolvere **sicuramente prima o poi terminerà** la sua elaborazione rilasciando le risorse che sta utilizzando e quindi non è un ostacolo per i processi che necessitano di quelle risorse e le stanno aspettando: sicuramente nel futuro prossimo le risorse saranno libere e il **grafo ridotto** evidenzia già questa situazione che, come vedremo in seguito, sarà utile perché la maggiore applicazione dei grafi di **Holt** è quella che ci permette di individuare situazioni critiche di blocco del sistema (**stallo**).

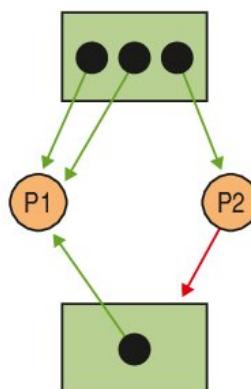


GRAFO RIDUCIBILE

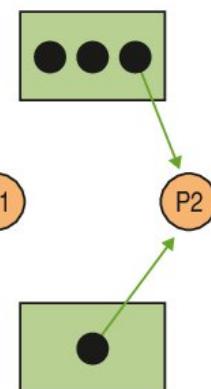
Un grafo di **Holt** si dice **riducibile** se esiste almeno un nodo di tipo processo con solo archi entranti.

ESEMPIO

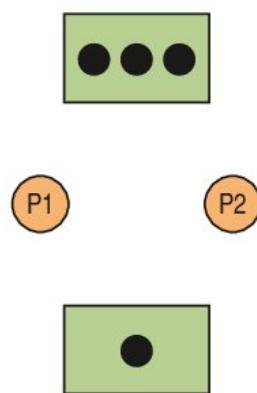
Nel seguente esempio abbiamo il processo P1 che sta utilizzando tre risorse e, dato che non è in attesa di altre, sta sicuramente evolvendo e, sicuramente, presto rilascerà quanto sta utilizzando:



effettuando la **riduzione per P1** si ottiene un nuovo grafo dove si considera terminata l’elaborazione di P1, si eliminano gli archi entranti e si rilasciano le risorse:



Ora anche il processo P2 può evolvere, dato che ha tutte le risorse che gli sono necessarie, e quindi possiamo anche effettuare la *riduzione per P2*, ottenendo:



Verifichiamo le conoscenze

1. Risposta multipla

- 1** Quale tra le seguenti non è una fase di natura statica di gestione delle risorse?
a) allocazione b) disponibilità c) ottimizzazione d) costo
- 2** Quale tra le seguenti non è una fase di natura dinamica di gestione delle risorse?
a) allocazione d) autenticazione
b) controllo di accesso e) controllo di correttezza operazioni
c) ottimizzazione ed eccezioni
- 3** Le richieste di assegnazione delle risorse possono essere classificate:
a) secondo il numero c) secondo il tipo di richiesta che effettuano
b) secondo il tempo di utilizzo d) secondo il tempo di attesa
- 4** Le modalità di assegnazione delle risorse possono essere classificate in:
a) statica c) dinamica e) temporanea
b) permanente d) condivisa f) esclusiva
- 5** Le risorse possono essere classificate:
a) in base al loro costo d) in base alla loro molteplicità
b) in base alla mutua esclusività e) in base alla loro velocità
c) in base alla modalità di utilizzo
- 6** Nei grafi di Holt (indica le affermazioni corrette):
a) risorse e processi costituiscono due sottoinsiemi e sono rappresentati mediante nodi
b) le risorse sono rappresentate mediante nodi di forma quadrata
c) le classi di risorsa sono rappresentate mediante nodi di forma quadrata
d) l'arco che connette una risorsa a processo indica che la risorsa è assegnata al processo
e) l'arco che connette un processo a una risorsa indica che il processo sta usando la risorsa

2. Vero o falso

- 1** Il numero di risorse in una classe viene detto replicazione del tipo di risorsa. **V F**
- 2** Un processo all'occorrenza può richiedere una particolare risorsa di una classe. **V F**
- 3** Se il numero di processi è maggiore della molteplicità di una risorsa, questa viene condivisa. **V F**
- 4** I processi accedono concorrentemente alle risorse condivise. **V F**
- 5** Le fasi di pianificazione che riguardano l'assegnazione di una risorsa sono di natura dinamica. **V F**
- 6** Un processo effettua una richiesta multipla se necessita contemporaneamente di almeno due risorse per poter evolvere. **V F**
- 7** Una richiesta di risorsa è non bloccante se essa può essere utilizzata in modo condiviso. **V F**
- 8** Una periferica di I/O è un esempio tipico di risorsa assegnata staticamente. **V F**
- 9** Un esempio tipico di risorsa con accesso seriale è il CD-ROM. **V F**
- 10** Una risorsa si dice preemptable se può venire sottratta al processo che la sta utilizzando. **V F**
- 11** Nei grafi di Holt si rappresentano le richieste che possono essere soddisfatte. **V F**
- 12** Un grafo di Holt è riducibile se esiste almeno un nodo di tipo processo con solo archi entranti. **V F**

Verifichiamo le competenze

3. Esercizi

- 1 Supponiamo di avere quattro processi A, B, C, D e quattro risorse R, S, T, Q. E supponiamo che il sistema allochi le risorse disponibili al primo processo che le richiede.

La sequenza di richieste è la seguente:

C richiede R
D richiede T
B richiede T
A richiede S
A richiede Q
A richiede R
C richiede T

Rappresenta la situazione mediante un grafo di Holt.

- 2 Supponiamo di avere tre processi A, B, C e tre risorse Q, R, S e che il sistema allochi le risorse disponibili al primo processo che le richiede.

La sequenza di richieste è la seguente:

- 1) A richiede R;
- 2) B richiede Q,
- 3) C richiede S;
- 4) C richiede R;
- 5) A richiede Q;
- 6) B rilascia Q;
- 7) B richiede S;
- 8) A rilascia R;

Rappresenta la situazione mediante un grafo di Holt.

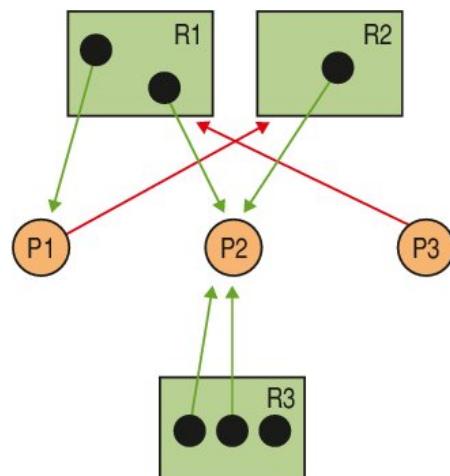
- 3 Supponiamo di avere tre processi A, B, C e tre risorse Q, R, S e che il sistema allochi le risorse disponibili al primo processo che le richiede.

La sequenza di richieste è la seguente:

- 1) A richiede Q;
- 2) C richiede S;
- 3) C richiede Q;
- 4) B richiede R;
- 5) B richiede S;
- 6) A rilascia Q;
- 7) A richiede S;
- 8) C richiede R.

Rappresenta la situazione finale mediante un grafo di Holt.

- 4 Dato il grafo di Holt di figura, per quale processo è possibile farne una riduzione? Disegna il grafo ridotto.



- 5 Supponiamo di avere un sistema dotato di 5 risorse (R, S, T, U, V) seriali, non prerilasciabili, contese da cinque processi (A, B, C, D, E) e che il sistema allochi le risorse disponibili al primo processo che le richiede.

La sequenza di richieste è la seguente:

B richiede R
A richiede R
A richiede S
C richiede T
A richiede T
E richiede U
D richiede V
C richiede U
E richiede R
D richiede T

1) Rappresentare lo stato del sistema con il grafo di Holt?

2) Per quale processo è possibile effettuare una riduzione? Rappresentare il grafo ridotto.

3) Sono possibili altre riduzioni? In caso affermativo procedere con tutte le successive riduzioni possibili.

I thread o “processi leggeri”

In questa lezione impareremo...

- ▶ la differenza tra processi e thread
- ▶ le modalità di utilizzo dei thread nei SO

■ Generalità

Le applicazioni che richiedono l’elaborazione parallela sono di tipologie molto diverse: dai videogiochi al controllo di processo, dall’elaborazione matematica alla gestione dei server internet; in tutte queste situazioni la necessità di **collaborare** e di **condividere risorse** è decisamente diversa per ciascuna applicazione e quindi è opportuno avere a disposizione del progettista più strumenti per poter descrivere nella maniera più appropriata sia le situazioni in cui è richiesto un alto **grado di parallelismo** con molteplici **risorse condivise**, sia quelle in cui la **cooperazione** è molto ridotta e le attività svolte in parallelo sono quasi totalmente indipendenti, con poche interazioni e piccole aree di memoria condivise.

Per loro natura i **processi** sono entità autonome e sono quindi adatti alla descrizione di attività autonome con poche risorse condivise e poco si prestano alla scrittura di applicazioni fortemente cooperanti.

Oltre al **processo** viene definita una nuova entità a esso molto simile ma con particolari caratteristiche che agevolano la risoluzione di problemi con alta cooperazione e condivisione di risorse: i **thread**.

Al **modello a processi** che implementa un insieme di macchine virtuali “una per ciascun processo”, si affianca quindi un **modello a thread**, che definisce un sistema di macchine virtuali che realizzano “**delle attività**” piuttosto che un “**compito completo**”, come descriveremo nel seguito.

In questa lezione riprenderemo i concetti principali sui **processi** e sull'utilizzo delle **risorse** in modo condiviso esposti nelle lezioni precedenti per confrontarli con i **thread** al fine di evidenziarne pregi e difetti per poter individuare le situazioni nelle quali è preferibile l'utilizzo di quest'ultimi rispetto ai **processi**.

■ “Processi pesanti” e “processi leggeri”

I **processi** che abbiamo descritto sino a ora vengono anche definiti **processi pesanti** per distinguerli dai **thread** che sono spesso chiamati **processi leggeri**.

Processi pesanti

Si è detto che il **processo** può essere visto come l'insieme della sua *immagine* e dalle *risorse* che sta utilizzando, dove:

- **L'immagine del processo** è costituita proprio dal *process ID*, *Program Counter*, *Stato dei Registri*, *Stack*, *Codice*, *Dati* ecc.;
- **le risorse possedute** sono i file aperti, processi figli, dispositivi di I/O.,,

Abbiamo definito come **spazio di indirizzamento** l'insieme dell'immagine di un **processo** e le **risorse** da esso possedute: la sua allocazione dipende dalla tecnica di gestione della memoria adottata (per esempio, parte del codice può essere su disco e gestita con tecniche di paginazione e segmentazione o di overloading).



PROPRIETÀ DEI PROCESSI

Una caratteristica fondamentale dei processi è che ciascuno di essi ha un proprio **spazio di indirizzamento**, cioè due processi non condividono nessuna area di memoria: come vedremo in seguito, neppure i processi figli condividono le variabili dichiarate dei rispettivi padri che li hanno generati.

Quando un **processo** si sospende e avviene il cambio di contesto, per rendere operativo il nuovo **processo** le operazioni che il SO deve compiere sono molteplici e complesse in quanto richiedono il salvataggio del contesto del **processo** che si sospende e il ripristino di quello che inizia (o riprende) la sua esecuzione.

L'esecuzione delle operazioni di ▶ **context switch** ▶ richiede tempo utile di **CPU** che, quindi, viene così sprecato per effettuare queste operazioni “non produttive”: questo tempo impiegato prende il nome di **overhead** e può essere definito come “costo di gestione” per realizzare il **multitasking**.

◀ **Context switch** Context switching is the procedure of storing the state of an active process for the CPU when it has to start executing a new one. ►



Per questo motivo al processo viene “assegnato l'aggettivo di **pesante**” perché richiede “pesanti” elaborazioni per passare dallo stato di **pronto** a quello di **esecuzione**.

Naturalmente l'obiettivo di ogni SO è quello di ridurre al minimo l'**overhead** migliorando gli algoritmi di scheduling.

Processi leggeri

Il **processo** in evoluzione può essere visto come l'unione di due componenti:

- ▷ **le risorse che utilizza**, cioè quelle comprese nel suo spazio di indirizzamento;
- ▷ **l'esecuzione del codice**, cioè il flusso di evoluzione del programma che condivide la **CPU** con gli altri processi.

Il **sistema operativo** gestisce questi due componenti in modo indipendente, e questa osservazione è alla base dei **thread**:

- ▷ la *parte del processo* alla quale viene assegnata la **CPU** viene definita **processo leggero (thread)**;
- ▷ la *parte del processo* che possiede le risorse viene definita **processo pesante (processo o task)**.



THREAD

Un **thread** è un **flusso di controllo** che può essere attivato in parallelo ad altri **thread** nell'ambito di uno stesso processo e quindi nell'esecuzione dello stesso programma.

In altre parole, un **thread** è un “segmento di codice” (tipicamente una funzione) che viene eseguito in modo sequenziale all'interno di un **processo pesante** e tutti i **thread** definiti all'interno di un **processo** ne **condividono le risorse**, risiedono nello **stesso spazio di indirizzamento** e hanno accesso a **tutti i suoi dati**.

Inoltre questo codice viene condiviso con gli altri **thread** ed eseguito in parallelo agli altri **thread** mandati in esecuzione dallo stesso **processo** con il vantaggio di **condividere lo spazio di indirizzamento** e quindi le **strutture dati** e le **variabili**.

Il termine “**processo leggero**” (**LightWeight Process LWP**) vuole indicare che l'implementazione di un **thread** è meno onerosa di quella di un vero **processo**, e con **multithreading** si indica la molteplicità di flussi di esecuzione all'interno di un **processo pesante**.

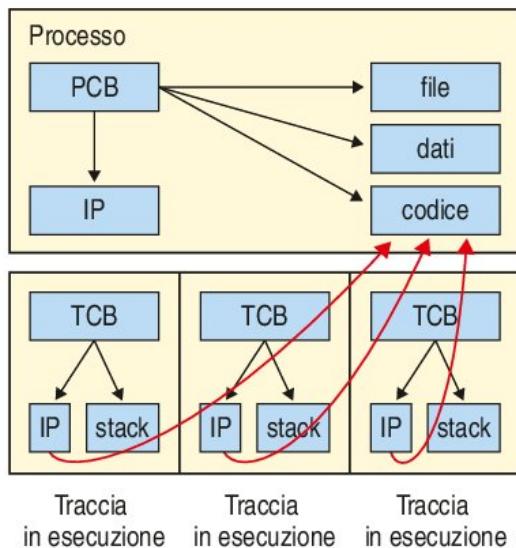
Per evolvere **parallelamente** agli altri **thread** o **processi** che si contendono la **CPU**, il **thread** ha inoltre un insieme di propri elementi che lo caratterizzano, in analogia ai processi tradizionali, chiamato **TCB** (il **Thread Control Block**), costituito da:

- ▷ un **identificatore di thread** (ID);
- ▷ un **program counter**;
- ▷ un **insieme di registri**;
- ▷ uno **stato di esecuzione** (running, ready, blocked);
- ▷ un **contesto** che è salvato quando il **thread** non è in esecuzione;
- ▷ uno **stack** di esecuzione;
- ▷ uno spazio di memoria privato per le **variabili locali**;
- ▷ un puntatore al **PCB** del processo contenitore.

I **thread** non hanno una loro area dove è presente il codice del programma in quanto condividono quello del processo che li genera così come ne condividono l'area dati.

Il **TCB** è quindi simile al **PCB** e contiene i **registri**, lo **stack**, le **variabili "locali"** e lo **stato esecuzione**: dati "globali" e **TCB** "locale" rappresentano lo **stato di esecuzione** del singolo **thread**.

Rappresentiamo graficamente tre **thread** mandati in esecuzione all'interno di un processo di cui eseguono tutti un segmento di codice:



È anche possibile definire variabili "personalizzate" per ogni singolo **thread**, con accesso tramite **chiave**, così che nessun altro **thread** "fratello" o **processo** le possa né vedere né tanto meno modificare.

L'utilizzo dei **thread** offre la possibilità di sfruttare meglio le architetture multiprocessore e di comunicare e scambiare informazioni in modo immediato: inoltre è più vantaggioso avere i **thread** rispetto ai **processi** in quanto le operazioni **context switch** sono più semplici e veloci, dato che i **thread** condividono molti dati e quindi sono in numero minore quelli da salvare e ripristinare.



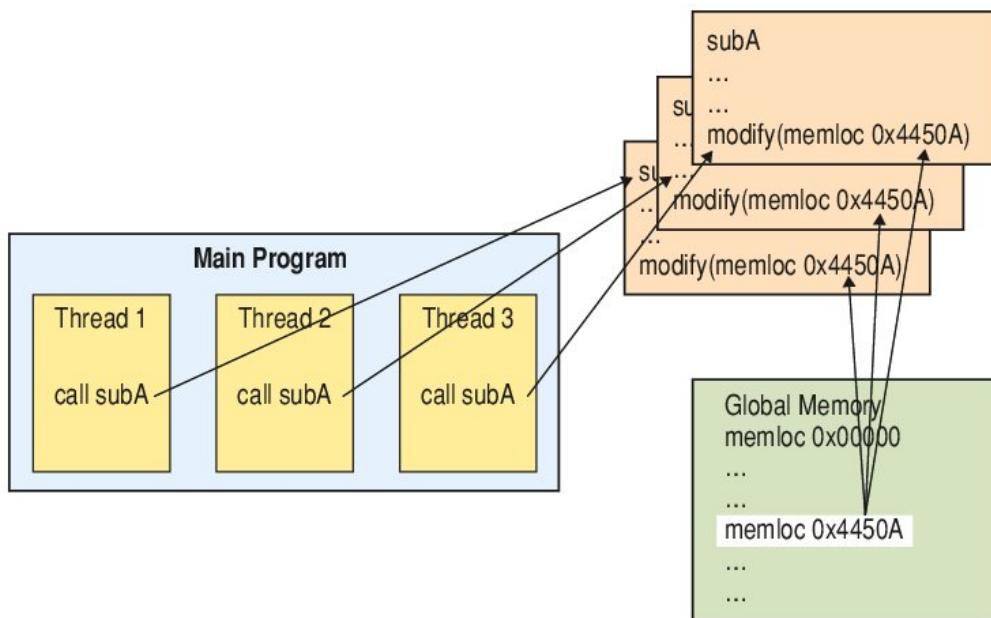
THREAD SAFETY

Un programma o segmento di codice è detto **thread-safe**, o anche che gode della proprietà di **thread safeness**, se è corretto anche nel caso di esecuzioni multiple da parte di più **thread** garantendo che nessun **thread** possa accedere a dati in uno stato inconsistente, cioè durante il loro aggiornamento.

L'esecuzione dei **thread** richiede necessariamente che le routine di libreria debbano essere **rientranti**: i **thread** di un programma interagiscono con il **SO** mediante **system call** che usano dati e tabelle di sistema dedicate al **processo** e queste devono essere progettate in modo da poter essere utilizzate da più **thread** contemporaneamente (**thread safe call**) senza che vengano persi i dati.

ESEMPIO

La funzione `subA` scrive il proprio risultato in una *variabile del processo* e restituisce al chiamante un puntatore a tale variabile.



Se due **thread** di uno stesso processo eseguono “nello stesso istante” la chiamata a due **subA** ognuno setta la variabile con un valore: quale valore sarà letto dai **thread** chiamanti al termine della esecuzione di **subA**?

La tabella seguente riporta il confronto tra **processi** e **thread**, evidenziando per ogni tipologia **pregi** e **difetti**.

	Processi	Thread
Creazione DISTRUZIONE	Richiedono allocazione, copia e deallocazione di grandi quantità di memoria	Richiedono solamente la creazione di uno stack per il thread
ERRORE	Non può danneggiare altri processi	Può danneggiare altri thread e l'intero processo cui appartiene
CODICE	Un processo può modificare il proprio codice mediante il cambiamento di eseguibile	Il codice di un thread è fissato e presente nella sezione text del processo cui appartiene
CONDIVISIONE	È onerosa e deve essere implementata dal programmatore	È automaticamente garantita poiché tutti i thread condividono la memoria del processo cui appartengono
MUTUA ESCLUSIone	La mutua esclusione è garantita automaticamente dall'isolamento proprio dei processi	Deve essere realizzata dal programmatore mediante semafori, mutex ecc.
PRESTAZIONI	Limitate dall'overhead di gestione	Elevate
CONCORRENZA	Limitata dalla difficoltà di comunicazione	Elevate

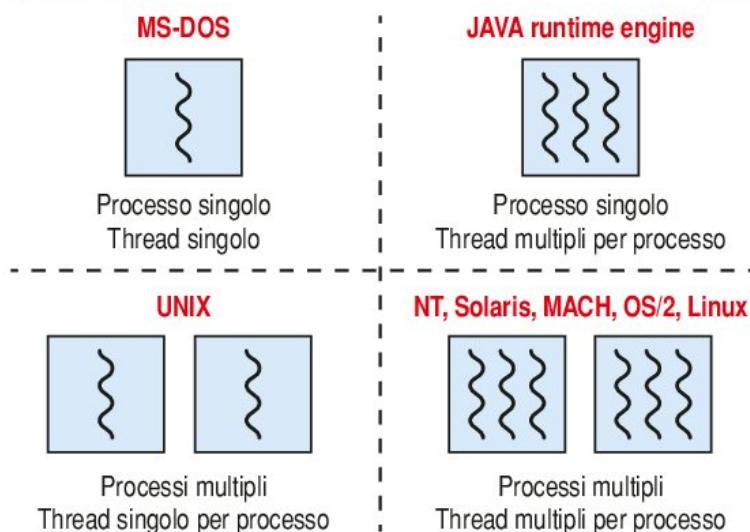
■ Soluzioni adottate: single threading vs multithreading

In base alla capacità di un sistema di gestire **a livello kernel** i **thread**, distinguiamo tra quattro possibili scenari, ottenuti dalla combinazione delle possibili situazioni:

- singolo processo e **thread** singolo;
- singolo processo e **thread** multiplo per processo;
- multiplo processo e **thread** singolo per processo;
- multiplo processo e **thread** multiplo per processo.

Queste quattro situazioni le possiamo riscontrare in quattro diversi ambienti operativi:

- **MS-DOS**: un solo processo utente e un solo **thread**;
- **UNIX**: più processi utente ciascuno con un solo **thread**;
- **supporto run time di Java (JVM)**: un solo processo, più **thread**;
- **Linux, Windows NT, Solaris**: più processi utente ciascuno con più **thread**.



In questo volume faremo quindi riferimento sempre a situazioni con **thread multipli**, pensando a soluzioni con linguaggio **Java** a prescindere dal sistema operativo, e a soluzioni realizzate in **linguaggio C** eseguite su macchine con sistema operativo **Linux** (o in emulazione con **cygwin**).

■ Realizzazione di thread

Gli ambienti operativi hanno due modalità per realizzare un sistema multithreading a seconda che il **kernel** del sistema operativo sia o meno a conoscenza della loro esistenza, cioè se i **thread** vengono realizzati mediante chiamate al **kernel** oppure realizzati da software mediante procedure e librerie. Nel primo caso si parla di **Kernel-Level**, nel secondo di **User-Level**.

User-Level

I **thread a livello utente** sono quelli che vengono implementati grazie a delle librerie apposite (**thread package**) che contengono le funzioni per la creazione, terminazione, sincronizzazione dei **thread** e per realizzare anche i meccanismi per il loro scheduling: quindi nè la schedulazione e neppure lo switching con il cambio di contesto coinvolgono il nucleo, che "ignora" la loro presenza e gestisce solamente i processi.

I **thread** sono quindi di “proprietà e gestione” esclusiva del **processo** che li ha creati, uno alla volta, chiamando le apposite funzioni di libreria.

I vantaggi di questa soluzione sono innanzitutto l’efficienza di gestione in quanto i **tempi di switching** sono molto ridotti dato che non richiedono chiamate al **kernel**, hanno una grande flessibilità e scalabilità, dato che lo scheduling può essere modificato e dimensionato volta per volta in funzione della specifica situazione.

Dato che sono realizzati mediante librerie, possono essere implementati su qualunque sistema operativo e quindi hanno un alto grado di portabilità tra macchine e sistemi diversi.

Tra gli svantaggi il primo che riportiamo è il fatto che se un **thread** effettua una **system call** per esempio per motivi di I/O, oltre che a sospendere se stesso provoca la sospensione del processo che lo ha generato e quindi anche di tutti gli altri **thread** sempre generati dallo stesso processo.

Inoltre non è possibile sfruttare il parallelismo fisico in architetture multiprocessore per **thread** generati dallo stesso processo dato che sono “interni al processo stesso” e quindi assegnati a uno specifico processore.

L’esempio tipico di ambienti **User-Level** è l’ambiente **UNIX** che naturalmente non implementa i **thread**, ma questi vengono generati all’interno dei suoi processi.

Kernel-Level

A livello di nucleo la gestione dei **thread** affidata al **kernel** tramite chiamate di sistema e quindi è il **kernel** che gestisce i **thread** come tutti gli altri processi, li deve schedulare, sospendere e risvegliare assegnandogli le risorse di sistema: a differenza del caso precedente, se un **thread** si sospende può naturalmente evolvere un secondo **thread** generato dallo stesso processo, in quanto sono tra loro schedulati in modo autonomo.

Questo è proprio il vantaggio più significativo di questa seconda soluzione unito al fatto che in architetture multiprocessori si può sfruttare al massimo il parallelismo fisico: inoltre lo stesso **kernel** può essere scritto come un sistema multithread.

Lo svantaggio principale è legato alla efficienza del sistema dovuta ai tempi impiegati dal **kernel** per il cambio di contesto durante la schedulazione che, di fatto, risulta essere più complessa in quanto deve gestire la copresenza di processi e **thread**.

Tra i sistemi operativi che realizzano ambienti **Kernel-Level** ricordiamo **Linux** e **Windows**.

Soluzione mista

Esistono anche soluzioni miste, come quella implementata in **Solaris**, che combina le proprietà di entrambi i meccanismi permettendo di creare a livello utente dei **thread** che solo però preventivamente devono essere definiti a livello di **kernel** (i **thread** utente vengono “mappati” sopra i **thread** a livello **kernel**, quindi non possono essere in numero superiore a essi), e lasciano all’utente le politiche di scheduling e di sincronizzazione.

I principali vantaggi sono che **thread** della stessa applicazione possono essere eseguiti in parallelo su **processori** diversi e che la chiamata al **kernel** da parte di un **thread** non blocca necessariamente il **processo** che lo ha generato.

■ Thread POSIX

Il modello ANSI/IEEE per i **thread** è definito dallo **standard POSIX** (Portable Operating System Interface for Computing Environments), che comprende un insieme di direttive per le interfacce applicative (API) dei **sistemi operativi**.

La definizione di **standard** ha lo scopo di indicare le regole da rispettare per realizzare strumenti compatibili, o meglio, dei programmi applicativi portatili in ambienti diversi: se un programma applicativo utilizza solamente i servizi previsti dalle **API** di **POSIX** può essere portato su tutti i sistemi operativi che implementano tali **API**.

I **thread** di **POSIX** sono chiamati **Pthread** e saranno da noi utilizzati per la realizzazione dei programmi concorrenti semplicemente richiamando la libreria `<pthread.h>`: in essa viene definito il tipo **`pthread_t`** che sarà usato per definire i **thread** nei programmi concorrenti creandoli all'interno di un processo, quindi siamo nella situazione 4 di **thread multipli per processo**.

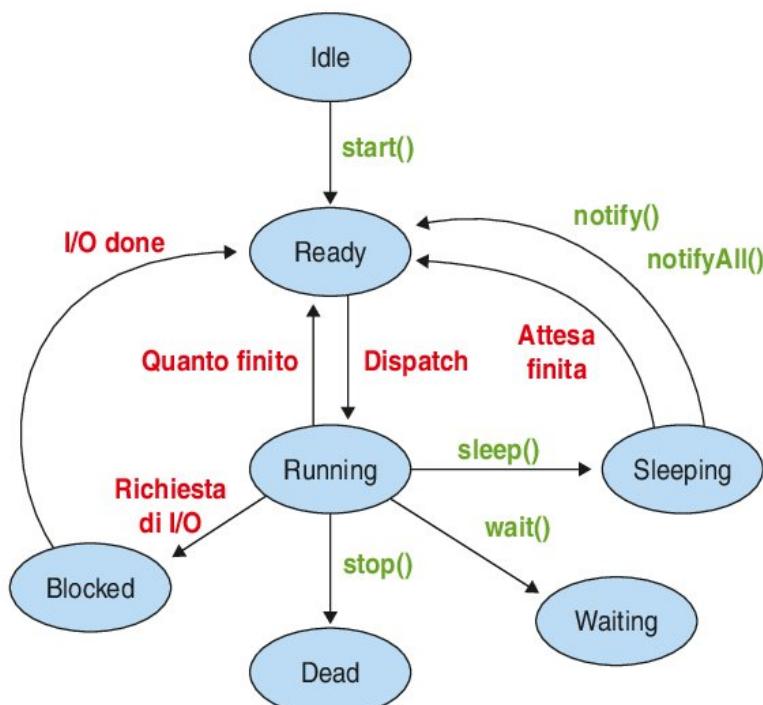
In alcuni compilatori la libreria base offerta dal **linguaggio C** include già le funzionalità relative alla gestione delle **espressioni regolari** che sono definite dallo standard **POSIX**: pertanto in queste situazioni non esiste propriamente una **libreria C** e una **POSIX**: a seconda del compilatore utilizzato è necessario verificare quando è necessario specificare in fase di compilazione l'inclusione della libreria precompilata per la gestione di quella certa funzionalità **POSIX**.

■ Stati di un thread

Come per i **processi**, anche i **thread** durante il loro ciclo di vita passano attraverso diversi stati: inoltre la vita del **thread** è legata alla vita del processo che li genera in quanto se un processo termina questo comporta anche la terminazione di tutti i suoi **thread**: è invece "indipendente" durante la vita anche se è attivo all'interno del processo, cioè evolve indipendentemente dal fatto che il processo sia in esecuzione o in attesa.

Il ciclo di vita dei **thread** è riportato nel diagramma a fianco.

Nel diagramma *le transizioni sotto il controllo del sistema sono indicate in rosso*, quelle effettuate da istruzioni, e quindi dove il controllo è del programma, sono indicate in verde.



Gli stati sono i seguenti:

- **Idle**: prima di essere avviato;
- **Dead**: terminate le sue istruzioni;
- **Blocked**: in attesa di completare l'I/O;
- **Sleeping**: sospeso un periodo;
- **Waiting**: in attesa di un evento;
- **Running**: in esecuzione;
- **Ready**: pronto per l'esecuzione ma in attesa della CPU.

Un **thread** è **Idle** quando non è ancora avviato e viene posto nello stato di **Ready** dove condivide una apposita coda con tutti i thread e i processi che attendono la **CPU** e sono gestiti dalle diverse politiche di scheduling, alternando **Running** a **Ready**.

Dall'esecuzione passa allo stato di **Blocked** in caso di richieste di operazioni di **I/O**, e nello stato di **Waiting** quando esegue chiamate bloccanti che ne causano la sospensione: non appena la causa del blocco è stata rimossa (per esempio sono arrivati i dati dal disco) il **thread** ritorna nello stato di **Ready**.

Dallo stato di **Running** un **thread** può passare anche allo stato di **Sleeping**, che è semplicemente uno stato nel quale effettua dei cicli di attesa (sospensione per un certo tempo) per poi essere riaccodato tra i processi pronti.

Dallo stato di **Running** un **thread** può infine passare allo stato di **Dead**, qualora esso abbia eseguito tutte le proprie istruzioni.

Quando un **thread** si blocca per una richiesta di I/O e passa nello stato di **Blocked** il sistema potrebbe decidere di eseguire un altro **thread** dello stesso processo oppure un **thread** di altri processi pronti, e la scelta del comportamento dipende dal tipo di implementazione

- nei **thread Java** implementati a **livello utente** il sistema non vede gli altri **thread** di quel processo ed esegue un **processo differente**;
- nei **thread C** implementati a **livello kernel** il sistema può gestire lo scheduling a livello **thread** secondo una qualche politica definita nel sistema operativo.

■ Utilizzo dei **thread**

Una delle principali applicazioni dei **thread** è quella di permettere di organizzare l'esecuzione di lavori con attività in **foreground** e in **background**: per esempio, mentre un **thread** gestisce l'I/O con l'utente, altri **thread** eseguono operazioni sequenziali di calcolo in **background**.

Per esempio nei fogli elettronici vengono utilizzati per le procedure di ricalcolo automatico, nei word processor per effettuare la reimpostazione oppure il controllo ortografico del documento che si sta creando, nel web per effettuare le ricerche nei motori o nei database ecc.

Un altro importante utilizzo dei **thread** è quello di implementarli per l'esecuzione di attività asincrone, come per esempio le operazioni di **garbage collection** nella gestione della **RAM** oppure nelle procedure di salvataggio automatico dei dati (**backup schedulati**).

Verifichiamo le conoscenze

1. Risposta multipla

- 1 Il processo può essere visto come l'insieme (due elementi):**
 - a) della sua immagine
 - b) del suo stato
 - c) delle risorse che sta utilizzando
 - d) del valore dei suoi registri
- 2 Quale tra le seguenti affermazioni relativa ai thread definiti all'interno di un processo è falsa?**
 - a) ne condividono le risorse
 - b) ne condividono i registri
 - c) risiedono nello stesso spazio di indirizzamento
 - d) hanno accesso a tutti i suoi dati
- 3 Ogni thread ha un suo (indica la risposta errata):**
 - a) identificatore di thread
 - b) program counter
 - c) insieme di registri
 - d) stato di esecuzione
 - e) segmento di codice
 - f) stack di esecuzione
 - g) spazio di memoria privato per le variabili locali
- 4 Gli ambienti operativi hanno due modalità per realizzare un sistema multithreading:**
 - a) Kernel-Level
 - b) Thread-Level
 - c) System-Level
 - d) User-Level

2. Associazione

- 1 Ai seguenti ambienti operativi associa la capacità gestione a livello kernel:**

1 UNIX	a) singolo processo e thread singolo
2 Supporto run time di Java (JVM)	b) singolo processo e thread multiplo per processo
3 MS-DOS	c) multiplo processo e thread singolo per processo
4 Linux, Windows NT, Solaris	d) multiplo processo e thread multiplo per processo

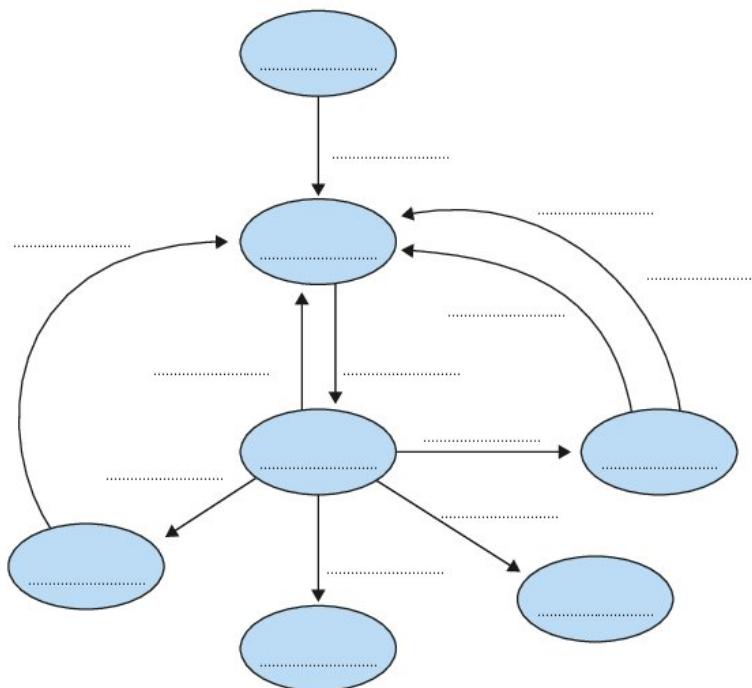
3. Vero o falso

- 1 I processi si prestano alla scrittura di applicazioni fortemente cooperanti.** V F
- 2 Due processi non condividono nessuna area di memoria.** V F
- 3 Due processi condividono le variabili dei loro antenati.** V F
- 4 Con overhead si intende il tempo sprecato per le operazioni di context switch nel multitasking.** V F
- 5 Ogni thread viene eseguito in parallelo agli altri mandati in esecuzione dallo stesso processo.** V F
- 6 Nei thread le operazioni context switch sono più semplici e veloci.** V F
- 7 L'esecuzione dei thread richiede che le routine di libreria debbano essere rientranti.** V F
- 8 Un programma ha la proprietà di thread safeness se è corretto anche nel caso di esecuzioni multiple da parte di più threads.** V F
- 9 I thread a livello utente sono anche chiamati User-Level perché vengono implementati grazie alle librerie del linguaggio utente.** V F
- 10 I thread Kernel-Level sono più efficienti dei thread User-Level.** V F
- 11 I thread Kernel-Level possono meglio sfruttare i multiprocessori rispetto ai User-Level.** V F
- 12 Nelle soluzioni miste possono esistere più threads che processi.** V F

Verifichiamo le competenze

1. Esercizi

- 1 Completa il seguente diagramma degli stati di un thread.



- 2 Descrivi i seguenti passaggi di stato.

Dall'esecuzione allo stato di Blocked.

Dall'esecuzione allo stato di waiting.

Dallo stato di running un thread può passare anche allo stato di sleeping.

Dallo stato di running un thread può passare allo stato di dead.

Elaborazione sequenziale e concorrente

In questa lezione impareremo...

- ▶ il concetto di programmazione concorrente
- ▶ a realizzare il grafo delle precedenze
- ▶ il concetto di interazione tra processi

■ Generalità

La **programmazione imperativa** che si apprende nei primi corsi di informatica ha come riferimento un **esecutore sequenziale** che svolge una sola azione alla volta sulla base di un **programma sequenziale**.



ELABORAZIONE SEQUENZIALE

Con il termine **elaborazione sequenziale** si intende l'esecuzione di un programma sequenziale che genera un processo sequenziale con un ordinamento totale alle azioni che vengono eseguite.

Lo stesso teorema di **Bohm** e **Jacopini** indica nella **sequenza** una delle **tre figure strutturali fondamentali**. L'**elaborazione sequenziale** è quindi un concetto fondamentale nell'informatica in quanto gli **algoritmi** che vengono computati sono composti da una **sequenza finita di istruzioni** in corrispondenza delle quali, durante la loro esecuzione, l'elaboratore passa attraverso una **sequenza di stati** (traccia dell'esecuzione del programma).

Anche l'esecutore dei programmi fino a ora utilizzato è una **macchina sequenziale** che si basa sul modello **Von Neumann**, cioè dotato di una sola unità di elaborazione (singola **CPU**).

Ma gli elaboratori non hanno tutti una sola **CPU** e inoltre, come abbiamo visto nello studio dei **sistemi operativi**, alcune attività del processore possono essere parallelizzate, come per esempio le lunghe fasi di input che provocano enorme spreco di tempo macchina soprattut-

to nei processi ad alta interattività con l'utente; esistono inoltre applicazioni che per loro natura necessitano di attività parallele, come i **server web**, i video games a più giocatori, i robot, e per essi la **codifica sequenziale** delle attività propria della **programmazione sequenziale** non è in grado di descrivere con naturalezza queste situazioni.

Queste situazioni necessitano di un diverso modello in grado di effettuare la programmazione di un **esecutore concorrente**, ovvero di un elaboratore che è in grado di eseguire più istruzioni contemporaneamente.



PROGRAMMAZIONE CONCORRENTE

Con **programmazione concorrente** si indicano le tecniche e gli strumenti impiegati per descrivere il comportamento di più attività o processi che si intende far eseguire contemporaneamente in un sistema di calcolo (**processi paralleli**).

Siamo in una situazione di elaborazione contemporanea reale solo nel caso in cui l'esecutore sia dotato di una **architettura multiprocessore**, cioè con più processori che possono eseguire ciascuno un singolo programma: nei sistemi monoprocessori sappiamo che il **parallelismo** avviene solo virtualmente, grazie alla **multiprogrammazione**, e più processi evolvono "in parallelo" grazie al **quanto di tempo** che viene loro assegnato dalle politiche di scheduling del **sistema operativo**.

Il **sistema operativo** è per eccellenza l'esempio più eclatante di **programmazione concorrente**: il suo compito è quello di assegnare le **risorse** hardware dell'elaboratore ai **processi** utente che ne fanno richiesta, cercando di massimizzarne l'efficienza nella loro utilizzazione. Le attività del **sistema operativo** devono essere eseguite **concorrentemente** in modo da consentire l'esecuzione contemporanea di più programmi utente: ogni attività **interagisce** con le altre sia in **modo indiretto**, occupando delle risorse comuni, sia in **modo diretto**, scambiando informazioni in merito allo stato delle risorse e dei programmi di utente al fine di realizzare la multiprogrammazione.

In un sistema multiprogrammato i programmi d'utente e le singole funzioni svolte dal sistema operativo possono essere considerati come un insieme di processi che **competono** per le stesse risorse.

Quindi un sistema **multiprogrammato** è un sistema **concorrente**, così definito:



SISTEMA CONCORRENTE

Per **sistema concorrente** intendiamo un sistema software implementato su vari tipi di hardware che "porta avanti" **contemporaneamente** una molteplicità di **attività diverse**, tra di loro correlate, che possono **cooperare** a un obiettivo comune oppure possono **competere** per utilizzare risorse condivise.



PROCESSO CONCORRENTE

Due **processi** si dicono **concorrenti** se la prima operazione di uno di essi ha inizio prima del completamento dell'ultima operazione dell'altro.

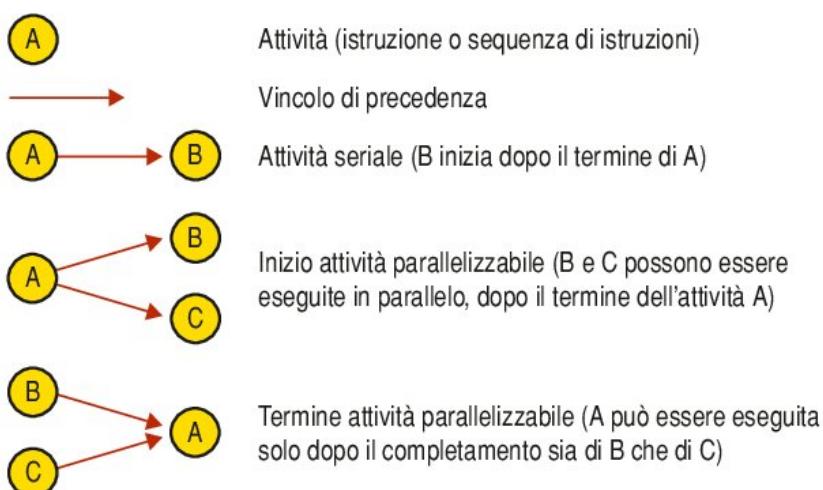
■ Processi non sequenziali e grafo di precedenza

Nei **processi sequenziali** la sequenza degli eventi che costituisce il processo è *totalmente ordinata*: se la rappresentiamo mediante un **grafo** orientato questo risulterà **totalmente ordinato** in quanto la sequenza degli eventi è ben determinata, cioè l'ordine con cui vengono eseguiti e sempre lo stesso. Un **grafo** che descrive l'ordine con cui le azioni (o gli eventi) si eseguono nel tempo prende il nome di **grafo delle precedenze**.

Nei **processi paralleli**, invece, l'ordinamento non è completo, in quanto l'esecutore per alcune istruzioni “è libero” di scegliere quali iniziare prima senza che il risultato sia compromesso: possiamo affermare che nella **elaborazione parallela** l'esecuzione delle istruzioni segue un **ordinamento parziale**.

Per descrivere questa libertà nella evoluzione dei processi concorrenti utilizziamo proprio il **grafo delle precedenze** (o **diagramma delle precedenze**): in un **processo sequenziale** il grafo delle precedenze degenera in una *lista ordinata* mentre in un **processo parallelo** è un **grafo orientato aciclico** e i percorsi alternativi indicano la possibilità di esecuzione contemporanea di più istruzioni.

Per la descrizione del **grafo delle precedenze** vengono utilizzati i seguenti simboli con i rispettivi significati:



ESEMPIO **Grafo delle precedenze**

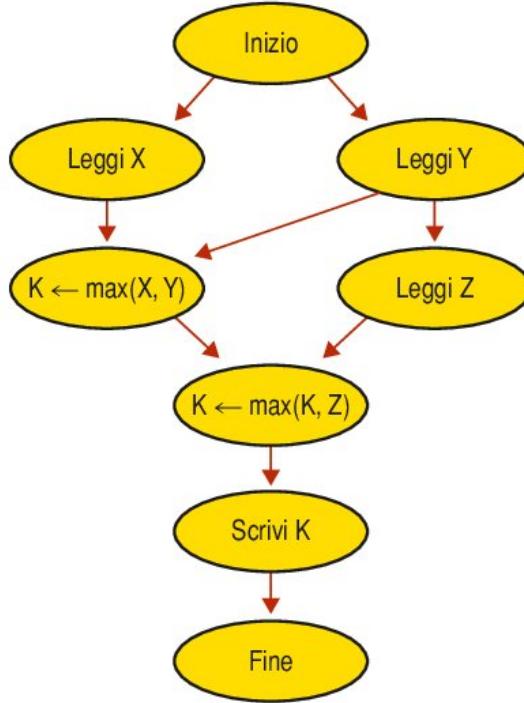
Vediamo un semplice esempio di un algoritmo che legge tre numeri e ne individua il maggiore. Per prima cosa scriviamo il codice sequenziale dell'algoritmo in *pseudocodifica (algoritmo sequenziale)*:

```

inizio
1. leggi X;
2. leggi Y;
3. leggi Z;
4. K ← max(X;Y);
5. K ← max(K;Z);
6. scrivi K;
fine
  
```

Ora riportiamo le istruzioni in un grafo dove esprimiamo i vincoli di effettiva precedenza per l'esecuzione delle istruzioni: *leggi X* e *leggi Y* non devono essere eseguite per forza in questo ordine, anzi, potrebbero anche essere effettuate in parallelo; anche la lettura di *Z* può essere eseguita dopo l'istruzione 4, oppure in parallelo con essa, mentre le istruzioni 5 e 6 devono per forza chiudere la sequenza delle operazioni.

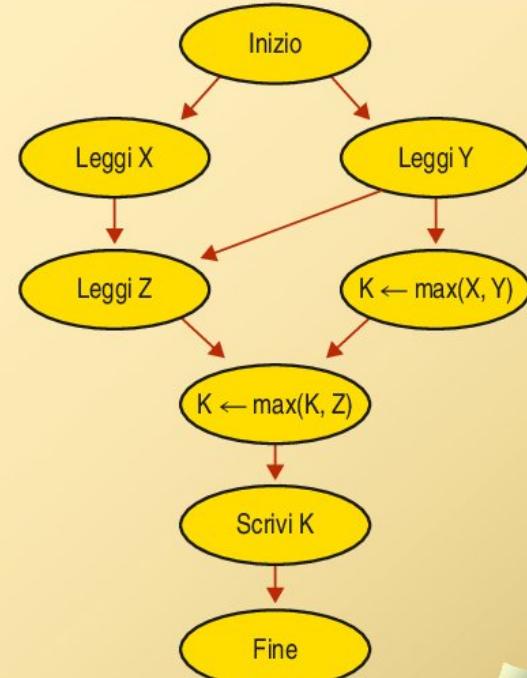
Il grafo è così fatto: ►



Questo non è l'unico diagramma delle precedenze possibile: la lettura di *Z* potrebbe essere eseguita dal ramo che effettua la lettura di *X* e l'operazione di calcolo del massimo tra *X* e *Y* potrebbe essere eseguita al suo posto, ottenendo ►

che è altrettanto logicamente corretto.

I grafi ottenuti sono quindi dei grafi a **ordinamento parziale**.



ESEMPIO**Ordinamento parziale e totale**

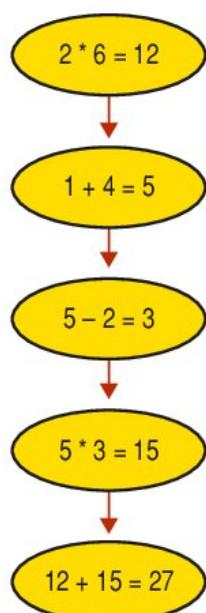
Vediamo un secondo esempio, dove il problema da risolvere è quello di valutare una espressione matematica: $(2 * 6) + (1 + 4) * (5 - 2)$

Le regole di precedenza in questo caso sono dettate dalla matematica, dove dapprima si devono eseguire le operazioni all'interno delle parentesi e successivamente si deve rispettare le precedenze degli operatori ($/$, $*$, $+$ e $-$).

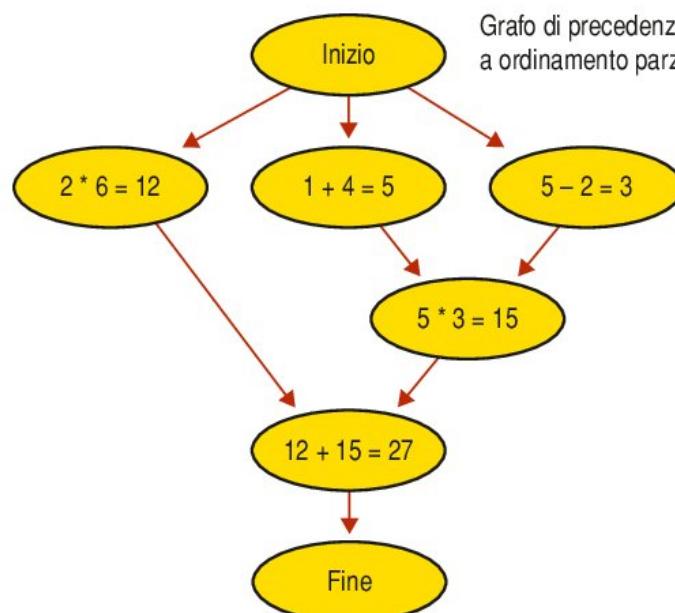
Non esiste l'obbligo di **un ordinamento totale** fra le operazioni da eseguire per il calcolo delle parentesi: potremmo indifferentemente eseguire prima $(1 + 4)$ piuttosto che $(2 * 6)$ o viceversa senza compromettere il risultato finale.

Rappresentiamo la soluzione con il grafo sequenziale a **ordinamento totale** per confrontarla con quello a **ordinamento parziale**, dove introduciamo la parallelizzazione delle attività (che in questo caso sono tutte operazioni algebriche):

Grafo di precedenza
a ordinamento totale



Grafo di precedenza
a ordinamento parziale



Prova adesso!

Date le seguenti espressioni algebriche:

$$1 \ 3+2-8*4-6/2+3 =$$

$$2 \ (3+2)-8*(4-6)/2+3 =$$

$$3 \ (3+2)-8*(4-6)/2+(3-8)*2 =$$

$$4 \ 3*2-(8*4-6)/2+(3-8/2)*2 =$$

A disegnare il grafo sequenziale a ordinamento totale;

B disegnare il grafo sequenziale a ordinamento parziale.

- Grafo delle precedenze
- Ordinamento parziale e totale

■ Scomposizione di un processo non sequenziale

Un **processo non sequenziale (parallelo)** consiste nella elaborazione contemporanea di più **processi** che sono di tipo sequenziale, e quindi possono essere studiati, descritti e programmati singolarmente.



SCOMPOSIZIONE SEQUENZIALE

Un **processo non sequenziale** può essere scomposto in un insieme di **processi sequenziali** che possono essere eseguiti contemporaneamente.

Possiamo quindi affrontare lo studio dei **processi paralleli** scomponendoli in **processi sequenziali** e risolvendoli ciascuno separatamente, con la programmazione classica dei **processi sequenziali**. Per poter correttamente descrivere la **concorrenza** è necessario distinguere le attività che i processi eseguono in due tipologie:

- attività **completamente indipendenti**;
- attività **interagenti**.

Processi indipendenti

La situazione più semplice da gestire è quella nella quale i processi sono tra loro **completamente indipendenti**.



PROCESSI INDIPENDENTI

L'evoluzione di un processo non influenza quella dell'altro.

Quindi sia che vengano eseguiti in sequenza che in parallelo non possono in nessun caso generare situazioni di funzionamento problematiche: l'unica accortezza è quella di **rispettare** per ogni processo l'**ordine stabilito delle operazioni**.

ESEMPIO **Scomposizione in processi indipendenti**

Supponiamo di avere il seguente segmento di codice:

```

inizio
1. leggi X;
2. leggi Y;
3. K ← SQRT(X);
4. W ← log(Y);
5. scrivi K;
6. scrivi W;
fine
  
```

Possiamo scomporre il programma in due segmenti completamente indipendenti:

segmento 1

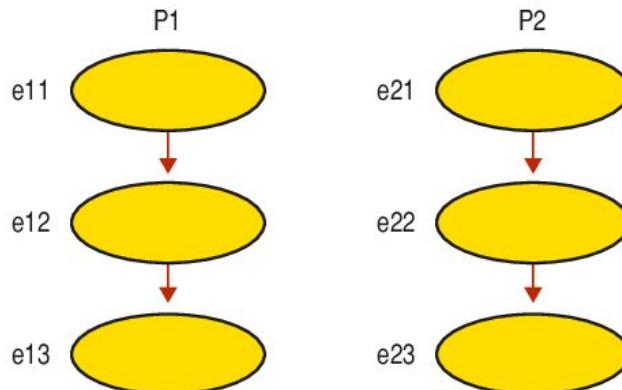
1. leggi X;
3. K ← SQRT(X);
5. scrivi K;

segmento 2

2. leggi Y;
4. W ← log(Y);
6. scrivi W;

Otteniamo due processi che eseguono ciascuno tre elaborazioni che *non hanno nulla in comune*, quindi ciascuno dei due processi può evolvere autonomamente senza interessarsi di quello che sta facendo l'altro. ►

Il **grafo di precedenze** può essere scomposto in due **grafi completamente autonomi**.



Due **processi indipendenti** devono lavorare su un **insieme privato** di variabili e ogni variabile che ciascuno di essi modifica non può essere utilizzata da nessun altro processo: l'unico caso in cui due **processi indipendenti** utilizzano una **variabile comune** è quello in cui su tale variabile effettuano solo **operazioni di lettura**.

L'ultima osservazione che possiamo fare su un insieme di processi concorrenti disgiunti è che il risultato di ciascuno di essi è **indipendente dalla velocità** con la quale viene eseguito ma dipende unicamente dai dati di ingresso.

Processi interagenti

La seconda situazione è quella in cui i due (o più) **processi** non possono evolvere in modo completamente autonomo perché **devono interagire** o volontariamente o involontariamente e quindi la rappresentazione nel **grafo delle precedenze** dovrà necessariamente avere degli elementi comuni.

È possibile classificare le modalità di interazione tra processi in base alla **loro conoscenza** o meno della presenza degli altri.

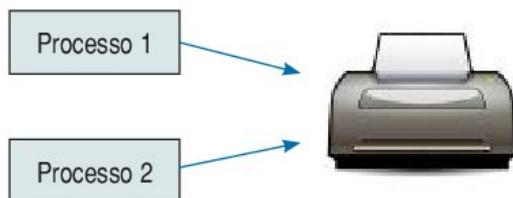
A processi **totalmente ignari**

In questo caso i processi sono indipendenti e non sono stati progettati per lavorare assieme ma “evolvono” in un ambiente comune: interagiscono tra loro in **competizione** sulle **risorse** e si devono quindi **sincronizzare**.

Questa situazione viene gestita dal sistema operativo, che deve essere arbitro della loro evoluzione effettuando la **sincronizzazione** all'accesso delle risorse ogni volta che i processi le richiedono.

ESEMPIO **Processi in competizione**

I processi sono in competizione per l'accesso a una stampante



oppure per l'utilizzo di una tabella di dati o di file che non può essere letta da un processo mentre viene modificata da un altro.

B processi indirettamente a conoscenza uno dell'altro

è questa la situazione nella quale i **processi** sono a conoscenza dell'esistenza degli altri ma non ne conoscono il nome (o il **PID**) e non possono comunicare direttamente tra loro ma devono **cooperare** per qualche motivo e possono scambiarsi i dati utilizzando risorse comuni, come aree di memoria condivisa.

Il sistema operativo deve offrire dei **meccanismi di sincronizzazione** che rendano possibile la cooperazione;

PID In *nix, a PID is a process ID. It is generally a 15 bit number, and it identifies a process or a thread. ►



C processi direttamente a conoscenza uno dell'altro

in questa situazione i **processi** devono **cooperare** per qualche motivo ma **comunicano** tra loro conoscendo i propri nomi: possono quindi effettuare direttamente lo scambio di informazioni mediante invio di **messaggi esplicativi**.

Il **sistema operativo** deve offrire dei **meccanismi di comunicazione** che rendano possibile la cooperazione.

Mecanismi di comunicazione e sincronizzazione tra entità (anticipazioni)

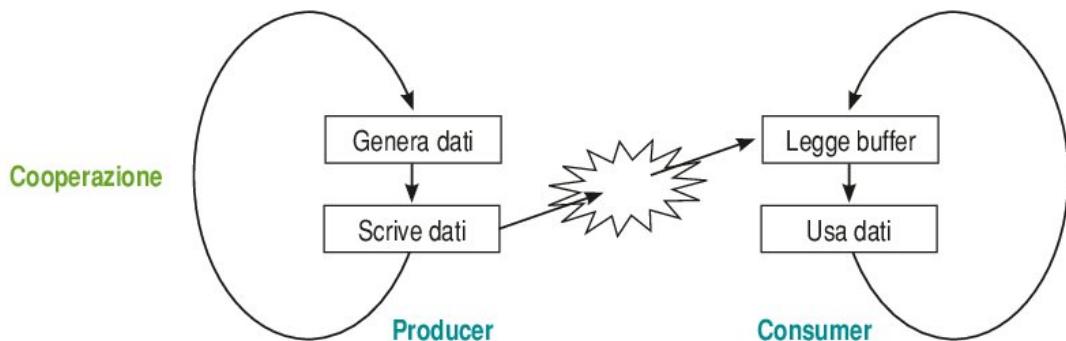
Negli ultimi due casi i **processi** devono **cooperare**, quindi il sistema operativo deve offrire i **meccanismi di sincronizzazione** in modo da garantire il corretto funzionamento regolando e gestendo i vincoli sull'ordine con cui devono essere eseguite le operazioni.

Una scorretta **sincronizzazione dei processi** dà luogo a una particolare categoria di errori, gli **errori dipendenti dal tempo**: questo tipo di errori spesso sono di difficile individuazione anche perché legati alla velocità relativa dei singoli processi e alla loro evoluzione autonoma, e quindi potrebbero manifestarsi o meno a seconda della casistica delle alternative di computazione in base alle diverse istanze di esecuzione.

Un **errore dipendente dal tempo** potrebbe **non ripetersi** anche riavviando il sistema e riportandosi nelle **medesime condizioni** nelle quali si è manifestato una prima volta.

È quindi di fondamentale importanza la scelta di **tecniche corrette di sincronizzazione** che possono essere realizzate in tre modalità differenti:

- A** attraverso l'utilizzo di **aree dati comuni** (memoria condivisa, in inglese **Shared Memory**): un **processo produce** un dato (**producer o produttore**) e lo scrive nella memoria condivisa (**buffer comune**) in modo che l'altro processo (**consumer o consumatore**) lo possa leggere e utilizzare.



Spesso questa situazione viene regolata mediante un meccanismo chiamato **monitor**: questo si occupa di gestire la memoria in modo **sincronizzato** ricevendo dati creati dal **produttore** e gestendo le richieste di lettura e di risposta del **consumatore**.

- B** attraverso lo **scambio di messaggi** un **processo** trasmette le informazioni all'altro processo: il meccanismo a disposizione è realizzato con dei meccanismi simili a semplici **operazioni** di I/O che prendono il nome di **InterProcess Communication (IPC)**. La **comunicazione** diretta viene realizzata con due **primitive** dove ogni **processo** deve specificare il “nome” dell’altro **processo** con il quale deve comunicare

```
send(processo_destinatario, messaggio)
receive(processo_mittente, messaggio)
```

Verifichiamo le conoscenze

1. Risposta multipla

- 1 Un'attività di un programma interagisce con le altre in modo diretto:**
 - a) occupando delle risorse comuni
 - b) scambiando informazioni
- 2 Un'attività di un programma interagisce con le altre in modo indiretto:**
 - a) occupando delle risorse comuni
 - b) scambiando informazioni
- 3 In un sistema concorrente le diverse attività tra di loro correlate (indica l'affermazione errata):**
 - a) possono cooperare
 - b) possono competere
 - c) possono ostacolarsi
 - d) nessuna delle precedenti
- 4 Il grafo delle precedenze:**
 - a) può essere una lista ordinata
 - b) può essere una lista non circolare
 - c) può essere grafo ciclico
 - d) può essere grafo orientato aciclico
- 5 I grafi a ordinamento parziale:**
 - a) sono incompleti
 - b) sono errati
 - c) descrivono un programma parallelo
 - d) descrivono espressioni con più soluzioni

2. Associazione

- 1 Abbina ai simboli il loro significato.**

- | | | |
|---------|--|--------------------------------------|
| 1 | | a) inizio attività parallelizzabile |
| 2 | | b) termine attività parallelizzabile |
| 3 | | c) attività seriale |

3. Vero o falso

- | | |
|---|---|
| 1 Un esecutore sequenziale svolge una sola azione alla volta di un programma sequenziale. | V F |
| 2 Un processo sequenziale ha un ordinamento totale alle azioni che vengono eseguite. | V F |
| 3 Un elaboratore in grado di eseguire più istruzioni contemporaneamente si dice concorrente. | V F |
| 4 I processi paralleli vengono descritti con la programmazione concorrente. | V F |
| 5 Per computare i processi paralleli sono necessarie macchine multiprocessore. | V F |
| 6 Nei processi sequenziali la sequenza degli eventi è totalmente ordinata. | V F |
| 7 Il grafo delle precedenze descrive l'ordine con cui le azioni si eseguono. | V F |
| 8 Il grafo delle attività descrive l'ordine con cui le azioni si eseguono. | V F |
| 9 Nei processi paralleli l'ordinamento delle istruzioni non è completo. | V F |
| 10 Generalmente esiste un solo diagramma delle precedenze possibile. | V F |

Verifichiamo le competenze

1. Esercizi

- 1** Costruisci un grafo delle precedenze che esprima il massimo grado di parallelismo nel calcolo delle seguenti espressioni dopo avere letto da input il valore di ogni variabile:

- a) $(A+B)*(C+D)$
- b) $(-B-\sqrt{B^2-4AC})/(2A)$
- c) $2P=2A+2B \quad S=A*B-B*C-C*D$
- d) $(-B-\sqrt{B^2-4AC})/(2A)$

- 2** Costruisci il grafo delle precedenze dell'algoritmo che legge un numero e ne visualizza le prime 4 potenze utilizzando solo l'operatore di moltiplicazione.

Per esempio, l'algoritmo sequenziale è il seguente:

```

inizio
1. leggi X;
2. scrivi X;
3. X2 ← X * X;
4. scrivi X2;
5. X3 ← X2 * X;
6. scrivi X3;
7. X4 ← X2 * X2;
8. scrivi X4;
fine

```

- 3** Dato il seguente segmento di codice parallelizzabile:

```

A ← 10
B ← 20
G ← 2+B
D ← A+3
N ← A+3
E ← A+B
F ← D+N
H ← 2+N
L ← G+F+H
M ← 100+L

```

disegna il grafo delle precedenze che esprima il massimo grado di parallelismo.

- 4** Dato il seguente segmento di codice parallelizzabile:

```

A ← 10
B ← A+1
G ← 20+D
D ← A+3
N ← A+3
E ← B+D
F ← G+E+N
H ← 2+N
L ← G+F+H
M ← 100+L

```

disegna il grafo delle precedenze che esprima il massimo grado di parallelismo.

La descrizione della concorrenza

In questa lezione impareremo...

- ▶ l'istruzione fork-join
- ▶ l'istruzione cobegin-coend

■ Esecuzione parallela

L'esecuzione di un **processo non sequenziale** richiede un sistema specifico che permetta la codifica e l'esecuzione dei programmi, cioè necessita di:

- ▶ un elaboratore non sequenziale;
- ▶ un linguaggio di programmazione non sequenziale.

Elaboratore non sequenziale

L'elaboratore non sequenziale è una macchina che deve essere in grado di eseguire più operazioni contemporaneamente e, come abbiamo detto, sostanzialmente questo si può ottenere in due modi differenti:

- ▶ architettura parallela, cioè sistemi multielaboratori;
- ▶ sistemi monoproessori multiprogrammati.

Nei primi il **parallelismo è fisico**, mentre nei secondi il **parallelismo è virtuale**.



Nei nostri esempi tratteremo sempre il secondo caso in modo da poter scrivere e collaudare i programmi sui nostri PC che hanno tutti un sistema operativo multiprogrammato.

Linguaggi non sequenziali

Per scrivere **programmi non sequenziali** (o concorrenti) è inoltre necessario avere a disposizione dei particolari costrutti che permettano la descrizione delle **attività parallele**: non tutti i linguaggi hanno tali figure strutturali e quelli che consentono la descrizione delle attività concorrenti prendono il nome di **linguaggi di programmazione concorrente** (o **non sequenziale**).

La scrittura di un **programma concorrente** si basa sul concetto di **scomposizione sequenziale**.

Quindi il programmatore deve dapprima individuare le attività parallelizzabili e descriverle in termini di sequenze di istruzioni (blocchi sequenziali) e, successivamente mediante i linguaggi concorrenti, descrivere come tali moduli possono essere eseguiti in parallelo.



SCOMPOSIZIONE SEQUENZIALE

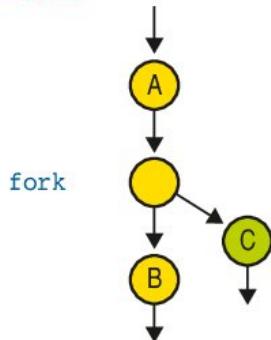
Un processo non sequenziale può essere scomposto in un insieme di processi sequenziali che possono essere eseguiti contemporaneamente.

I **linguaggi di programmazione concorrenti** di alto livello contemplano nuove istruzioni come il costrutto **fork-join** e **cobegin-coend**, che permettono di dichiarare, creare, attivare e terminare processi sequenziali.

■ Fork-join

Le istruzioni **fork** e **join** furono introdotte nel 1963 da **Dennis e VanHorne** per descrivere l'esecuzione parallela di segmenti di codice mediante la scomposizione di un **processo** in due **processi** e la successiva "riunione" in un unico **processo**.

Fork



In riferimento al **grafo delle precedenze**, la **fork** corrisponde alla biforcazione di un nodo in due rami: l'esecuzione di una **fork** coincide con la creazione di un processo che inizia la propria esecuzione **in parallelo** con quella del processo chiamante.

Con un **linguaggio di pseudocodifica** è possibile tradurre il **grafo** nel seguente segmento di codice:

```

/* processo padre: */
inizio
{
  ...
  A: <istruzioni>;
  p2 = fork figliol;
  B: <istruzioni>;
  ...
}
fine.
  
```

```
/* codice nuovo processo:*/
void figlio1()
{
    C: <istruzioni>;
}
```

Join

La **join** è l'istruzione che viene eseguita quando il processo creato tramite la **fork**, ha terminato il suo compito, si sincronizza con il processo che lo ha generato e termina la sua esecuzione. ►

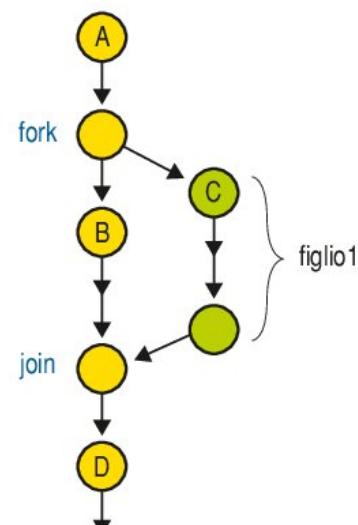
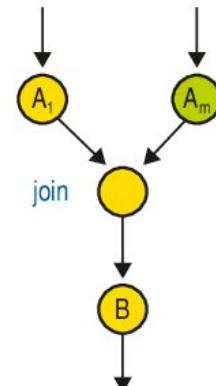
Con un linguaggio di pseudocodifica è possibile tradurre il grafo nel seguente segmento di codice:

```
...
join p2;
B:<istruzioni>
...
```

Il programma completo rappresentato nel diagramma a lato ► viene codificato in pseudolinguaggio con:

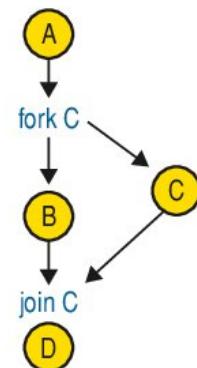
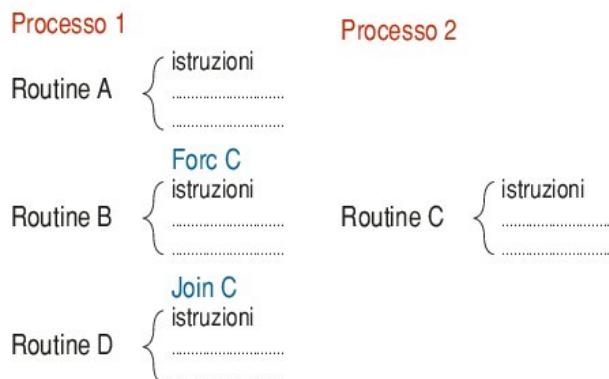
```
/* processo padre: */
inizio
...
    A:<istruzioni>;
    p2 = fork figlio1; // inizia l'elaborazione parallela
    B: <istruzioni>;
    join p2;           // termina l'elaborazione parallela
    D:<istruzioni>;
...
fine

/* codice nuovo processo:*/
void figlio1 ()
{
    C:<istruzioni>;
}
```



Se ipotizziamo che ogni nodo sia una *routine*, possiamo meglio comprendere il funzionamento della istruzione **fork** con il seguente schema, dove sono messi in evidenza i due **processi**:

- P1 esegue la Routine A;
- con la **fork** attiva il **processo 2** e in **parallelo** si eseguono la routine B e la routine C;
- P1 attende con la **join** la terminazione di P2;
- P1 esegue la routine C.



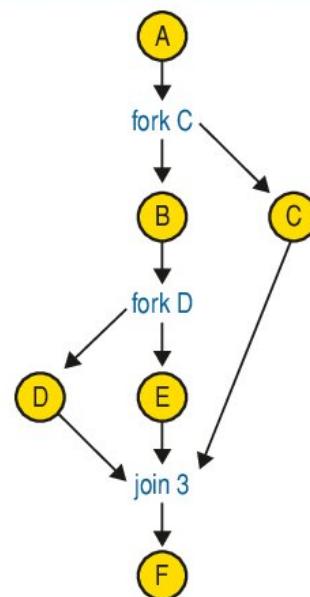
Join (count)

In questa definizione, la **fork** restituisce un identificatore di processo, che viene successivamente accettato da una **join**, in modo che sia specificato con quale processo si intende sincronizzarsi. Lo svantaggio di questa definizione è che la **join** può ricongiungere solo due flussi di controllo.

Esiste anche una formulazione estesa dell'istruzione **join**:

```
join(count);
```

dove **count** è una variabile intera non negativa e indica il numero di processi che si “riuniscono” in quel punto: viene utilizzata nel caso di terminazione congiunta di un numero superiore a due processi, come nel seguente grafo delle precedenze ▶



che viene codificato con questo segmento di programma:

```
/* processo padre: */
inizio
...
A:<istruzioni>;
  p2 = fork C;           // inizia l'elaborazione parallela con P2
```

```

B: <istruzioni>;
    p3 = fork D;           // inizia l'elaborazione parallela con P3
E: <istruzioni>;
join 3;                  // termina l'elaborazione parallela di 3 processi
F:<istruzioni>;
...
fine.

```

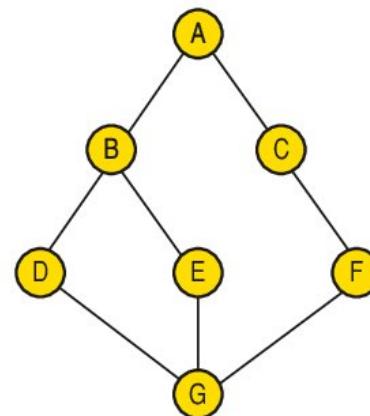
Nella letteratura la sintassi della pseudocodifica a volte è leggermente differente da quella appena presentata: spesso viene semplificata omettendo le <istruzioni> e indicando semplicemente con le lettere maiuscole A,B,C oppure con S1,S2,..., Sn il blocco o la sequenza di istruzioni, e tale simbologia sarà adottata anche da noi per le prossime codifiche.

A volte viene anche messa una label per indicare dove il processo termina ed effettua la **join**, come nel seguente esempio: ►

```

begin
    cont : = 3 ;
    A ;
    FORK E1 ;
    B ;
    FORK E2 ;
    D ;
    goto E3 ;
E1 : C ;
    F ;
    goto E3 ;
E2 : E ;
E3 : JOIN cont ;
    G ;
end.

```



Il linguaggio di shell **UNIX/Linux** ha due **system call** simili ai costrutti di alto livello definiti da **Dennis e VanHorne**:

- **fork()**: è l'istruzione utilizzata per creare un nuovo processo;
- **wait()**: corrisponde alla istruzione **join** e viene utilizzata per consentire a un processo di attendere la terminazione di un processo figlio.

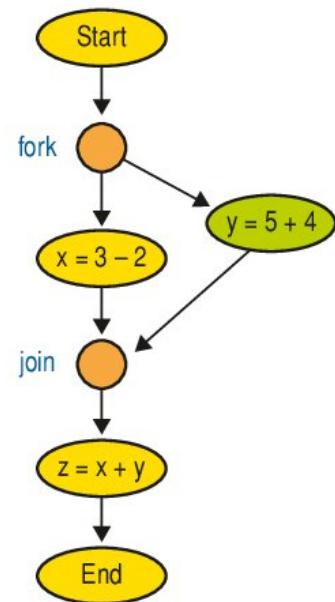
Sono però concettualmente diverse dato che sono *chiamate di sistema* mentre quelle da noi descritte sono istruzioni di un ipotetico *linguaggio concorrente di alto livello*: possono essere comunque utilizzate didatticamente per realizzare programmi concorrenti e alcune codifiche in **linguaggio C** che le utilizzano in ambiente **Linux** sono riportati a titolo di esempio in una lezione dedicata al laboratorio a conclusione di questa unità di apprendimento.

ESEMPIO

Scriviamo un programma parallelo che esegue la seguente espressione matematica:

$$z = (3-2) * (5+4)$$

Le operazioni tra parentesi possono essere parallelizzate ottenendo il seguente grafo: ▶



che viene codificato in:

```

/* processo padre: */
inizio
  p2 = fork(figlio1);      // inizia l'elaborazione parallela
  x=3-2;
  join p2;                // termina l'elaborazione parallela
  z=x+y;
  ...
fine

/* codice nuovo processo:*/
int figlio1()
{
  y=5+4;
  return y
}
  
```

■ Cobegin-coend

Il secondo costrutto che utilizziamo per descrivere la concorrenza è il **cobegin-coend**: è un costrutto che permette di indicare il punto in cui N processi iniziano contemporaneamente l'esecuzione (**cobegin**) e il punto che la terminano, confluendo nel processo principale (**coend**).

La sintassi del comando è:

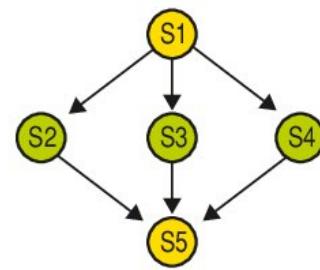
```

cobegin
  <elenco delle attività parallele>
coend
  
```

ESEMPIO

Il grafo delle precedenze di figura: ►
può essere descritto mediante il costrutto **cobegin-coend** con la
seguente pseudocodifica:

```
inizio
  S1
  cobegin
    S2      //attività parallele
    S3
    S4
  coend
  S5
fine
```

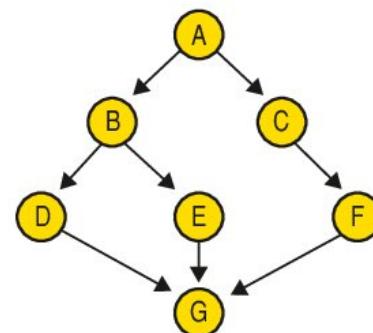


Al termine dell'esecuzione della sequenza S1 vengono attivati tre processi che eseguono in parallelo le sequenze di istruzioni S2, S3, S4: il processo padre S1 si sospende e rimane in attesa della loro terminazione.

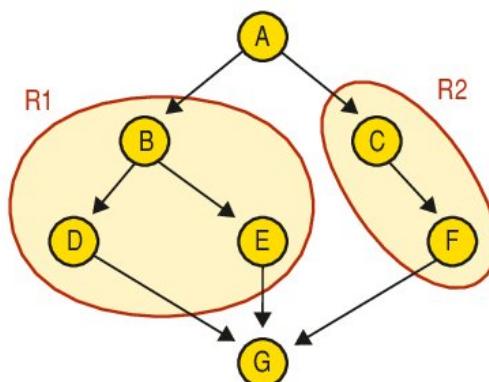
All'interno dei singoli processi possono a loro volta essere presenti delle istruzioni di **cobegin-coend**, cioè è possibile avere l'annidamento di più costrutti **cobegin-coend**.

ESEMPIO

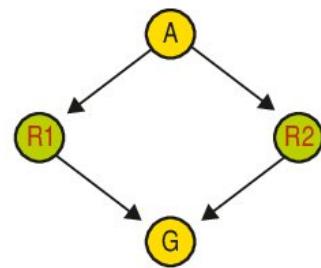
Codifichiamo il grafo delle precedenze utilizzando il costrutto **cobegin-coend** della figura a fianco. ►



Osservando il grafo delle precedenze possiamo notare come è sostanzialmente composto da due rami che possono essere evidenziati come nella seguente figura



Chiamando R1 e R2 le due sequenze di istruzioni, il grafo può essere semplificato come segue: ►



Il codice in pseudocodifica è quindi il seguente:

Processo P1	Processo R2	Processo R3
inizio A cobegin R1 R2 coend G fine	inizio B cobegin D E coend fine	inizio C F fine

Il costrutto **cobegin-coend** è decisamente più semplice da utilizzare e il codice che si ottiene è più strutturato e quindi più comprensibile di quello scritto con le istruzioni di **fork-join** che trasformano il codice in strutture che assomigliano ai vecchi programmi che utilizzavano l'istruzione di salto incondizionato "goto label", censurata dalla programmazione strutturata per la generazione di ▲ "spaghetti code" ▶ incomprensibili.

◀ **Spaghetti code** Spaghetti code is a derogatory term for computer programming that is unnecessarily convoluted, and particularly programming code that uses frequent branching from one section of code to another (using many GOTOs or other "unstructured" constructs). Spaghetti code sometimes exists as the result of older code being modified a number of times over the years. ►

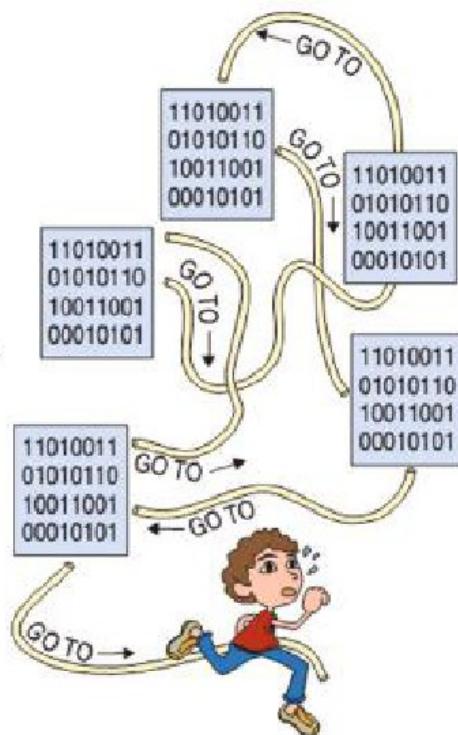


CAMMINO PARALLELO

Indichiamo con cammino parallelo una qualunque sequenza di nodi delimitata da un nodo **COBEGIN** e un nodo **COEND**.



Nell'esempio precedente è possibile individuare due **cammini paralleli**, uno esterno e uno annidato al suo interno.



■ Equivalenza di fork-join e cobegin-coend

Il costrutto **fork-join** può essere sostituito col costrutto **cobegin-coend** solo se non ci sono strutture annidate, nel tal caso l'unica possibilità di “conversione” è quella che ha la terminazione congiunta di tutti i **processi**: nel caso opposto, invece, sempre tutti i programmi codificati con **cobegin-coend** possono essere anche codificati con **fork-join**.



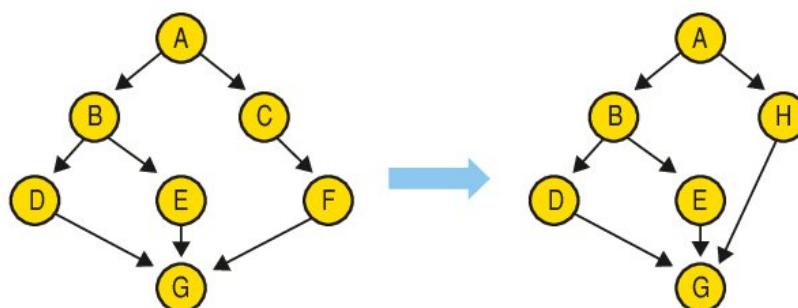
EQUIVALENZA TRA FORMALISMI

Qualunque programma parallelo può essere descritto utilizzando il costrutto **fork-join**, mentre non tutti i programmi paralleli possono essere descritti col costrutto **cobegin-coend**.

ESEMPIO

Da cobegin-coend a fork-join

Scriviamo mediante l'utilizzo di **fork-join** l'esempio che presenta due **cobegin-coend** annidati.



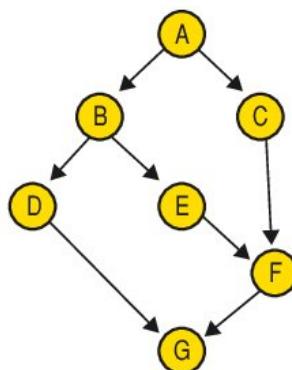
Dopo aver posto $H = C+F$, la pseudocodifica è la seguente:

```
/* processo padre: */
inizio
...
A
p2 = fork(H);           // inizia l'elaborazione parallela di B e H
B
p4 = fork(E);           // inizia l'elaborazione parallela di D e E
D
join p4;                // termina l'elaborazione parallela di D e E
join P2;                // termina l'elaborazione parallela del primo fork
...
fine.

/* processo figlio H */
inizio
C
F
fine
/* processo figlio E */
inizio
E
fine
```

ESEMPIO**Da fork-join a cobegin-coend**

Ipotizziamo ora di dover descrivere mediante **cobegin-coend** la situazione rappresentata nel seguente grafo delle precedenze.



Possiamo notare come il processo che esegue A si sospende per mandare in esecuzione due processi (**cobegin** P_B e P_C) dei quale il primo, a sua volta, si sospende per mandare in esecuzione altri due processi (**cobegin** P_D e P_E): è però impossibile determinare i corrispondenti **coend** in quanto i processi figli interagiscono tra di loro generando nuovi processi che non terminano assieme: quindi P_B non termina assieme a P_C .

**GRAFO STRUTTURATO**

Un grafo per poter essere espresso soltanto con **cobegin** e **coend** deve essere tale che detti X e Y due nodi del grafo, tutti i cammini paralleli che iniziano da X terminano con Y e tutti quelli che terminano con Y iniziano con X (o, più sinteticamente, ogni 'sottografo' deve essere del tipo one-in/one-out). In questo caso il grafo si dice **strutturato**.

Il grafo dell'esempio precedente non presenta questa caratteristica e quindi, non essendo **strutturato**, costituisce un esempio impossibile da descrivere soltanto con **cobegin** e **coend**. La codifica con il costrutto **fork-join** è invece fattibile, ma è necessario introdurre le istruzioni di salto in quanto le **fork** si intersecano tra loro:

```

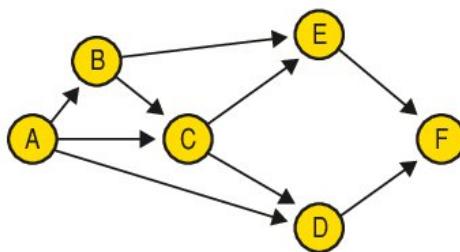
/* processo padre: */
inizio
...
A
p2=fork(C);           // inizia l'elaborazione parallela di B e C
  B
  p3=fork(E);         // inizia l'elaborazione parallela di D e E
    D
    goto L1
  join C;            // aspetta la terminazione di C
L1:join E;
  // aspetta la terminazione di E join C
G
...
fine.
  
```

La **join** etichettata con L1 non avviene tra gli stessi due processi che hanno fatto la prima **fork**, ma tra il primo processo e un processo generato dalla **join** tra due figli, il processo che esegue E è quello che esegue C.

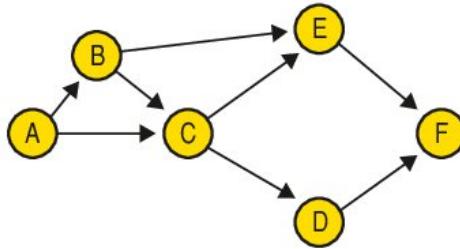
■ Semplificazione delle precedenze

Prima di affrontare la scrittura di un programma parallelo è sempre necessario soffermarsi ad analizzare il grafo delle precedenze per cercare di semplificarlo eliminando le **precedenze implicite**.

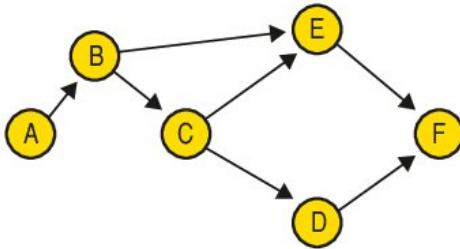
In presenza di **precedenze implicite** è possibile togliere un arco in quanto questo risulta ridondante ed è quindi possibile semplificare il grafo: vediamo un esempio e semplifichiamo il grafo riportato nella figura seguente:



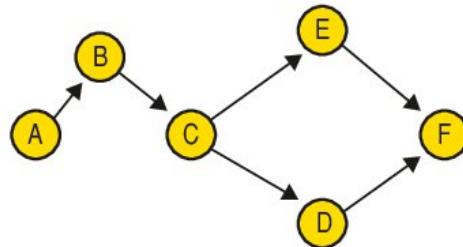
Possiamo osservare che il nodo D ha come precedenza il nodo A e anche il nodo C ha come precedenza il nodo A: quindi il nodo D ha il nodo A come precedenza diretta ma deve attendere l'elaborazione di C che a sua volta dipende da A; è possibile eliminare la precedenza tra A → D che risulta essere implicita in quella tra C → D: il grafo risulta essere trasformato nel seguente:



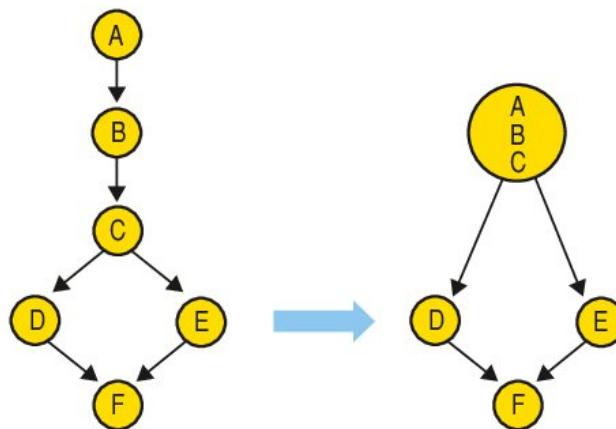
Analogo discorso può essere fatto tra B, che ha come precedenza il nodo A, e il nodo C, che anch'esso ha come precedenza il nodo A: quindi il nodo B ha il nodo A come precedenza diretta e dato che deve attendere l'elaborazione di B che a sua volta dipende da A è possibile eliminare la precedenza tra A → C che risulta essere implicita in quella tra B → C: il grafo risulta essere trasformato nel seguente:



Analogo discorso può essere fatto tra E, che ha come precedenza il nodo B, e il nodo C, che anch'esso ha come precedenza il nodo B: quindi il nodo E ha il nodo B come precedenza diretta e dato che deve attendere l'elaborazione di C, che a sua volta dipende da B, è possibile eliminare la precedenza tra B → E che risulta essere implicita in quella tra C → E; il grafo risulta essere trasformato nel seguente:



Ridisegnandolo, osserviamo che le tre operazioni A, B e C sono **sequenziali**, possono quindi essere raggruppate in un solo nodo, come si può vedere nel seguente disegno.

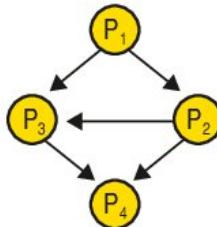


Il nostro grafo di partenza può essere trasformato in quest'ultimo, dove si vede che l'unica operazione che può essere parallelizzata è quella dei nodi D ed E: parallelizzare il primo grafo non porterebbe nessun vantaggio in termini di tempo di elaborazione.

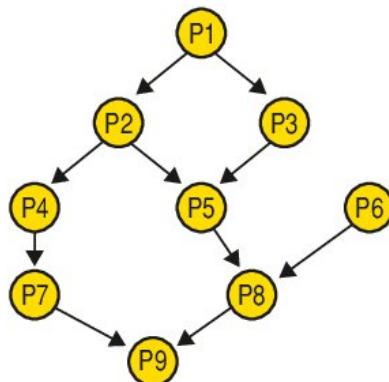
Verifichiamo le competenze

1. Esercizi

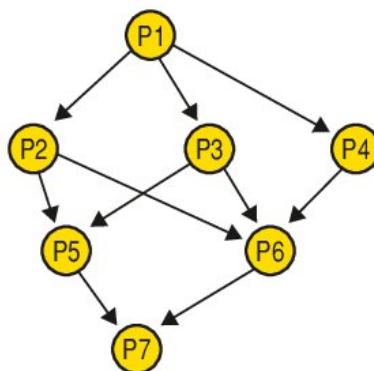
- 1 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio utilizzando il costrutto fork-join.



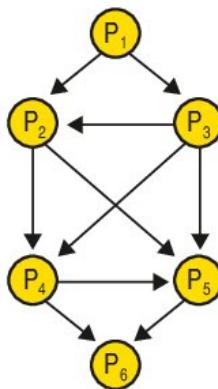
- 2 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio utilizzando il costrutto fork-join.



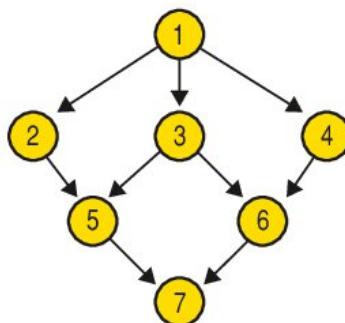
- 3 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio utilizzando il costrutto fork-join.



- 4** Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio utilizzando il costrutto fork-join.

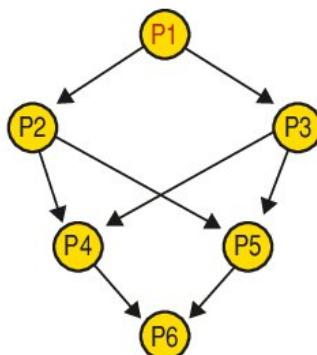


- 5** Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio utilizzando il costrutto fork-join.



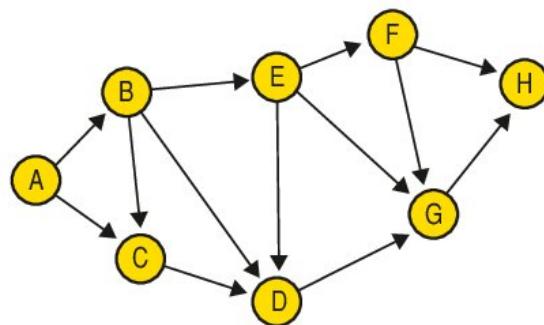
- 6** Realizza un programma con cobegin-coend che implementi Mergesort.

- 7** Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio:
- utilizzando il costrutto fork-join
 - utilizzando il costrutto cobegin-coend



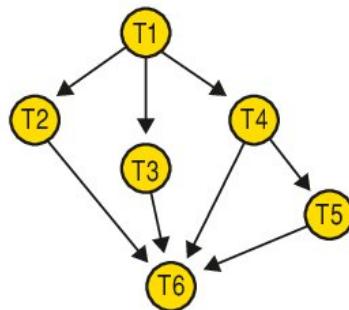
8 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio:

- a) utilizzando il costrutto fork-join
- b) utilizzando il costrutto cobegin-coend

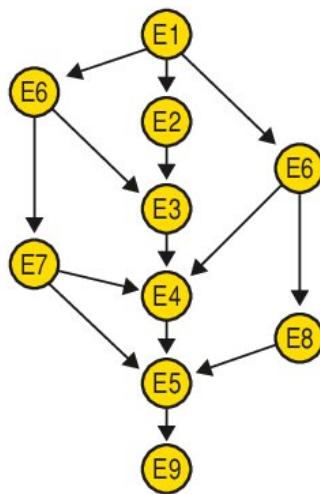


9 Partendo dal seguente grafo delle precedenze scrivi il programma concorrente in pseudolinguaggio:

- a) utilizzando il costrutto fork-join
- b) utilizzando il costrutto cobegin-coend



10 Codifica in pseudolinguaggio il seguente grafo di precedenze utilizzando il costrutto fork-join.



ESERCITAZIONI DI LABORATORIO 1

L'EMULATORE CYGWIN

■ Cos'è Cygwin

La più concisa (e al tempo stesso esauriente) spiegazione di cosa sia ▲ **Cygwin** ▶ la si trova sul sito Internet dove esso viene distribuito (<http://www.cygwin.com>):

Cygwin permette di avere su una macchina **Windows** la potenza e la flessibilità di una shell ▲ *NIX Like ▶ e quindi di utilizzare molti dei software tipici del mondo **Linux**.

◀ **Cygwin** "Cygwin is a Linux-like environment for Windows. It consists of two parts:

- ▶ A DLL (cygwin1.dll) which acts as a Linux API emulation layer providing substantial Linux API functionality.

A collection of tools which provide Linux look and feel." ▶



◀ *NIX Like Con *NIX si intendono i sistemi operativi **UNIX** e **XENIX** e con *NIX Like si intendono i sistemi operativi che sono loro "somiglianti", come **Linux** che sappiamo essere direttamente derivato da **UNIX**. ▶

Il vantaggio di utilizzare un ambiente emulato invece di uno "nativo" consiste nel non dover installare sulla propria macchina un nuovo sistema operativo, con tutte le problematiche che questa operazione solitamente comporta.

Naturalmente è diverso avere a disposizione una macchina **Linux** vera e una emulata, soprattutto in termini di risorse in quanto l'emulatore occupa buona parte della memoria **RAM** disponibile, ma per i programmi che verranno scritti durante questo corso, molto piccoli e "leggeri", l'ambiente emulato non si differenzia dall'ambiente reale, sia nel linkaggio che nella compilazione e nell'esecuzione.

Perché installiamo Cygwin

Naturalmente se stiamo lavorando su una macchina *NIX Like o ne abbiamo una partizione sul nostro PC non abbiamo bisogno di installare questo programma. Ma per eseguire una parte dei programmi concorrenti, in particolare quelli che eseguono la **fork**, è necessario avere a disposizione un sistema operativo *NIX like: durante l'installazione di **Cygwin** scaricheremo anche il compilatore **C** (il compilatore **GCC**), in modo da avere a disposizione "sotto **Windows**" un emulatore di macchina **Linux** completa per poter effettuare le nostre prove.

■ Installare Cygwin

Per installare **Cygwin** è necessario utilizzare un computer connesso a Internet: nella home page del sito <http://www.cygwin.com> si può individuare il link all'installer di **Cygwin** (<http://www.cygwin.com/setup.exe>, contraddistinto dall'etichetta "setup.exe").



È necessario scegliere la versione specifica per il proprio PC:

- **setup-x86.exe** per macchine a 32 bit;
- **setup-x86_64.exe** per macchine a 64 bit.

Nel momento in cui scrivo questa esercitazione, la versione di **Cygwin** più recente è quella comprendente la **DLL** in versione **2.0.4**. Le istruzioni che seguono faranno dunque riferimento all'installazione di questa specifica versione, ma sono identiche a quelle per le versioni precedenti.

Un semplice click con il pulsante sinistro del mouse su questo collegamento e una finestra come quella riportata di seguito chiederà che cosa si desidera fare con il file "**setup.exe**" cui si sta cercando di accedere.

Se per esempio utilizziamo una macchina 64 bit con il browser **Chrome**, cliccando su **setup-x86_64.exe** scarichiamo il programma di installazione, come si vede nella figura seguente:

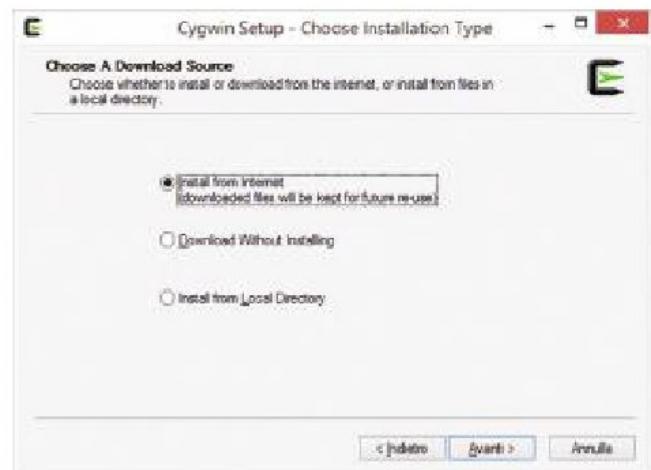


Ora cliccando su  si avvia l'esecuzione dell'installazione dell'ambiente **Cygwin**: viene visualizzata la seguente schermata:

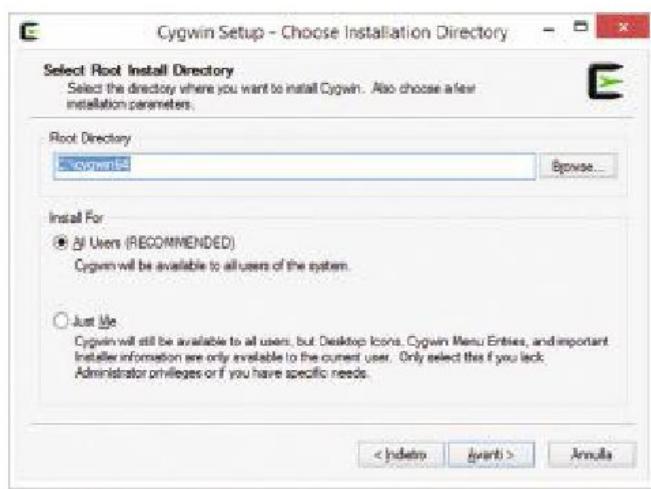


Cliccando **[Avanti]** inizia l'installazione vera e propria dell'emulatore.

Alla successiva finestra confermiamo quanto ci viene proposto, cioè la prima opzione (“*Install from Internet*”) e clicchiamo il pulsante **Avanti** per continuare. ►



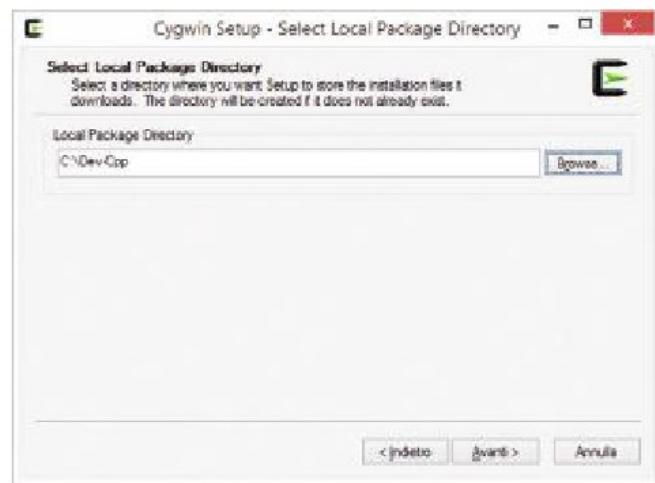
È necessario indicare in quale directory installare **Cygwin**: questa cartella sarà in seguito la root del nostro sistema e quindi deve essere facilmente raggiungibile; si consiglia di lasciare quella da lui proposta di default, cioè “**C:\cygwin64**”. ►



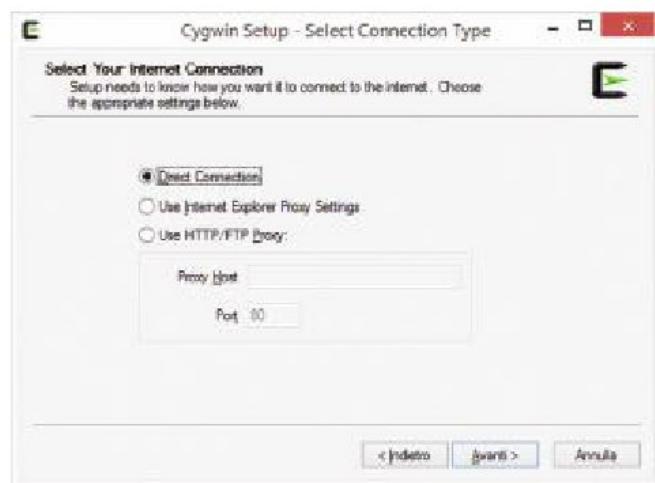
Senza modificare l'altra impostazione (“*Install For All Users*”) clicchiamo sul pulsante **[Avanti]** per continuare.

L'installer di **Cygwin**, a questo punto, ci chiede dove vogliamo copiare i file che vengono scaricati per il download, proponendoci la conferma della directory dove abbiamo copiato il file di setup.

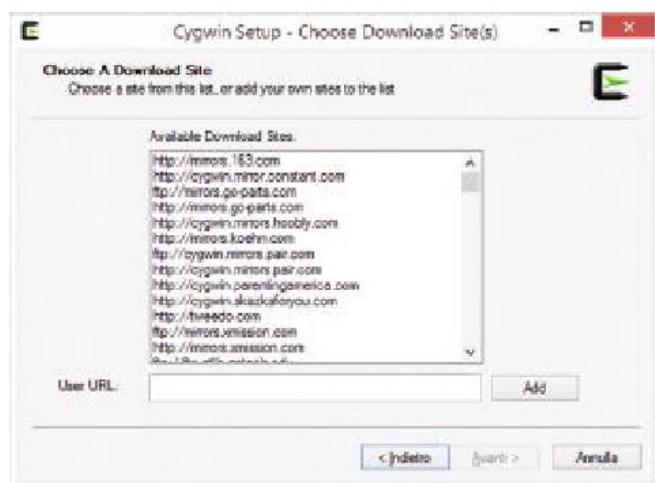
Confermiamo oppure mettiamo una directory esistente (nel nostro caso abbiamo messo quella del compilatore **Dev-Cpp** che, come vedremo, useremo come editor per i file sorgenti), e proseguiamo come al solito cliccando sul pulsante **[Avanti]**.



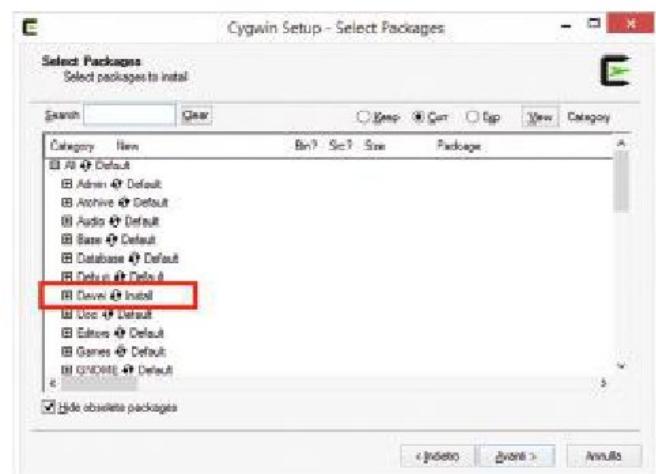
Selezioniamo ora come sarà effettuato il download: in caso di normale connessione via modem si conferma quanto proposto e si procede sempre cliccando sul pulsante **[Avanti]**. ►



Dobbiamo ora selezionare un server da cui scaricare i file: scelto uno a piacere (si consiglia uno europeo) si prosegue sempre cliccando sul pulsante **Avanti**. ►



La successiva schermata ci permette di selezionare i pacchetti che intendiamo installare: per utilizzare il compilatore **GCC** è necessario modificare le impostazioni di default e includere il “kit” per gli sviluppatori (developers) cliccando su **Devel** in modo che a fianco sia presente *Install*, come nella seguente immagine. ►

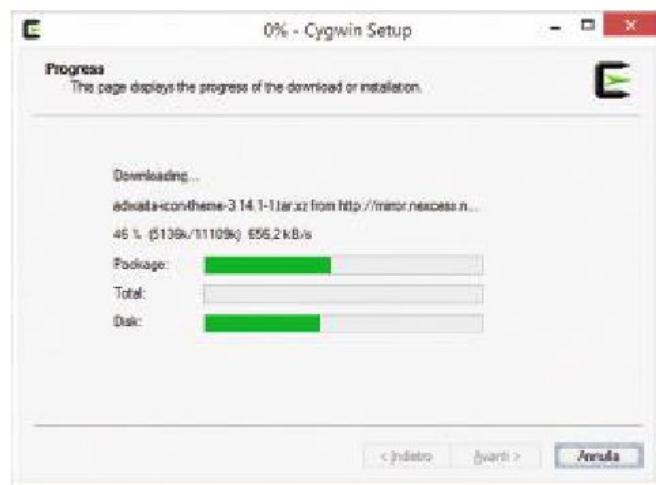
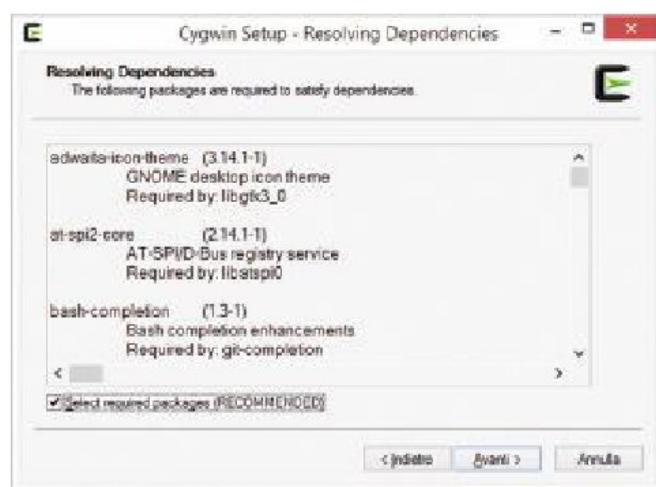


Allo stesso modo si può selezionare il download della documentazione, abilitando nella riga successiva **Doc**: si prosegue cliccando sul pulsante **Avanti**.

Se si presenta la seguente videata si prosegue cliccando nuovamente sul pulsante **Avanti**. ►

Nulla da modificare nella schermata seguente sempre cliccando sul pulsante **Avanti**.

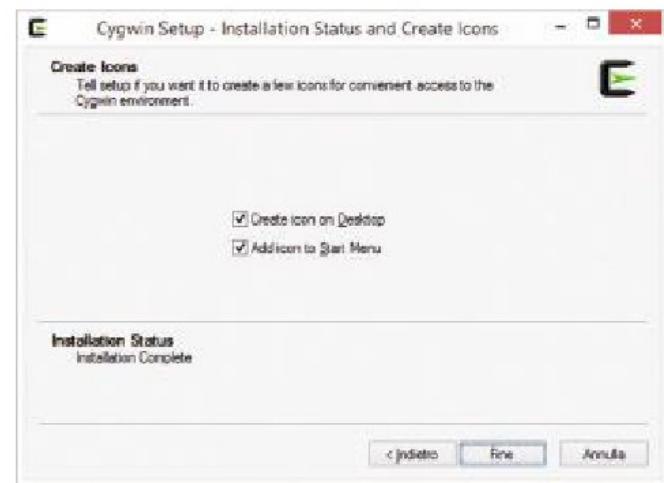
Inizia ora il download e l'installazione vera e propria: sullo schermo ora è presente la seguente videata: ►



L'installazione completa dell'emulatore, che ha dimensioni di circa 3,62 GB, richiede dai 20 ai 40 minuti, a seconda della velocità della connessione.

Al termine dell'installazione di **Cygwin** viene visualizzata la seguente schermata che chiede dove posizionare i collegamenti per lanciare l'applicazione: accettiamo e confermiamo cliccando su **Fine**. ►

Siamo ora pronti a utilizzare l'emulatore, a scrivere i programmi e compilarli con **GCC**.

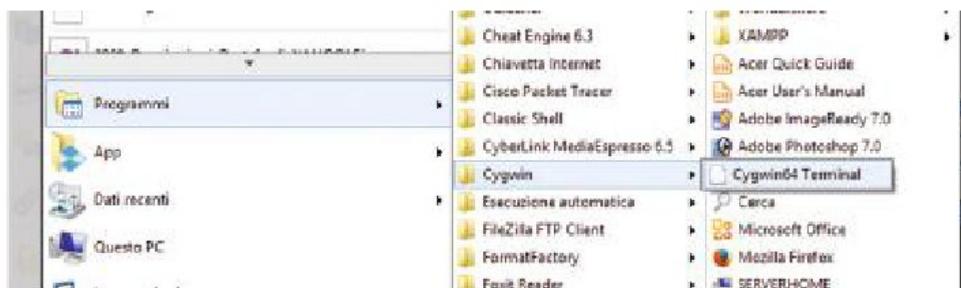


■ Il primo programma in **GCC**

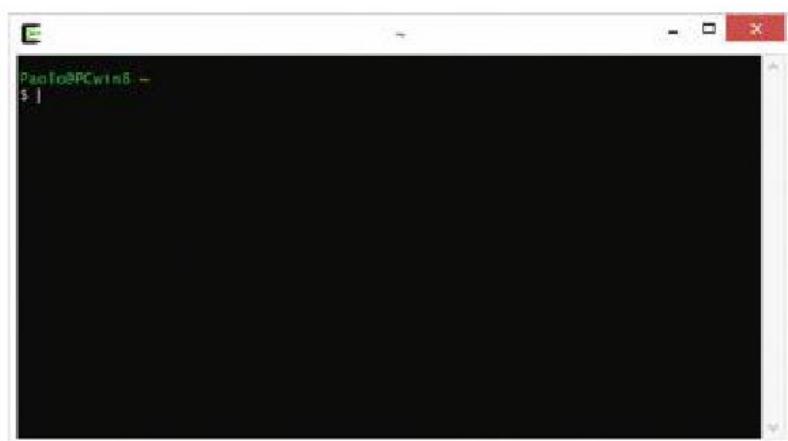
Ora che abbiamo il nostro ambiente di esecuzione, proviamo a scrivere il nostro primo programma, a compilarlo con **GCC** e a mandarlo in esecuzione.

Per prima cosa avviamo l'emulatore cliccando sull'icona presente sul desktop: ►

oppure dal menu di avvio, nell'opzione **Programmi**.



Ci si presenta una finestra nera, come quella di figura, tipica dello shell dei comandi degli ambienti testuali (come il **DOS**).



Il nome che viene proposto è il *nome_utente* assegnato in **Windows** dell'utente che ha installato **Cygwin**.

Se digitiamo il comando

```
help
```

viene visualizzata una schermata con l'elenco di tutti i comandi disponibili.

```
PaoTo@PCwin8 ~/ua1
$ help
GNU bash, versione 4.3.39(2)-release (x86_64-unknown-cygwin)
Questi comandi della shell sono definiti internamente. Digitare "help" per consultare questa lista.
Digitare "help nome" per saperne di più sulla funzione "nome".
Usare "info bash" per saperne di più sulla shell in generale.
Usare "man -k" o "info" per saperne di più su comandi non presenti nella lista.

Un asterisco (*) vicino a un nome significa che il comando è disabilitato.

spec_job [&]
(( espressione ))
. nomefile [argomenti]
:
[ arg... ]
[[ espressione ]]
alias [-p] [nome]=valore ...
bg [spec_job ...]
bind [-lpsvPSVX] [-m keymap] [-f filename] >
break [n]
builtin [comando-int-shell [arg ...]]
caller [espr]
case PAROLA in [MODELLO [| MODELLO]...]) CON>
ed [-L|[-P [-e]] [-o]] [dir]
```

Per avere il dettaglio di un singolo comando è sufficiente digitare il nome subito dopo **help**; per esempio il comando:

```
help cd
```

Visualizza la seguente schermata di dettaglio sul comando **cd**:

```
PaoTo@PCwin8 ~
$ help cd
cd: cd [-L|[-P [-e]] [-o]] [dir]
      Change the shell working directory.

      Change the current directory to DIR.  The default DIR is the value of the
      HOME shell variable.

      The variable CDPATH defines the search path for the directory containing
      DIR.  Alternative directory names in CDPATH are separated by a colon (:).
      A null directory name is the same as the current directory.  If DIR begins
      with a slash (/), then CDPATH is not used.

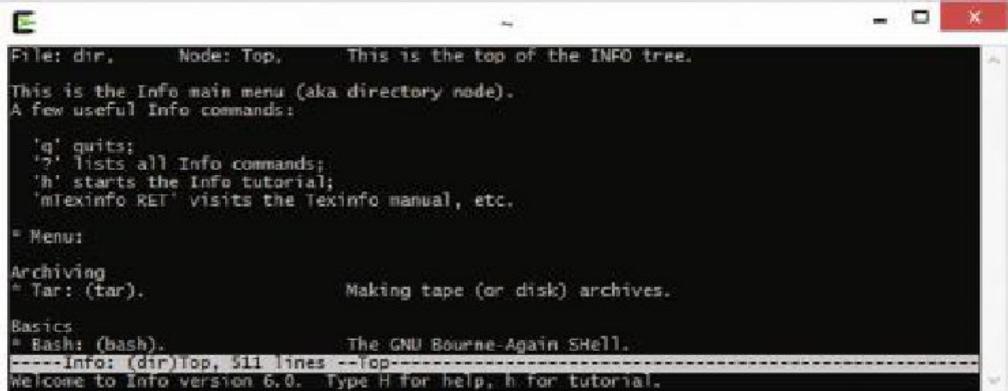
      If the directory is not found, and the shell option 'cdable_vars' is set,
      the word is assumed to be a variable name.  If that variable has a value,
      its value is used for DIR.

      Options:
        -L      force symbolic links to be followed: resolve symbolic links in
               DIR after processing instances of '..'
        -P      use the physical directory structure without following symbolic
               links: resolve symbolic links in DIR before processing instances
               of '..'
```

Esiste anche un manuale online che descrive ogni comando: la sua attivazione avviene mediante l'istruzione:

```
info
```

È più completo e offre per ogni istruzione sia un **help** immediato che un tutorial con esempi:



```

File: dir,      Node: Top,      This is the top of the INFO tree.
This is the Info main menu (aka directory node).
A few useful Info commands:
  'q' quits;
  '?' lists all Info commands;
  'h' starts the Info tutorial;
  'texinfo RET' visits the Texinfo manual, etc.

* Menu:
Archiving
* Tar: (tar).           Making tape (or disk) archives.

Basics
* Bash: (bash).          The GNU Bourne-Again SHell.
-----info: (dir)Top, 511 lines --Top-----
Welcome to Info version 6.0. Type H for help, h for tutorial.
  
```

Eseguiamo alcuni tra i comandi che utilizzeremo per la nostre esigenze operative.

Il primo comando che proviamo è quello che ci permette di muoverci nell'albero delle directory, cioè il comando **cd**: spostiamoci dalla nostra directory alla radice digitando

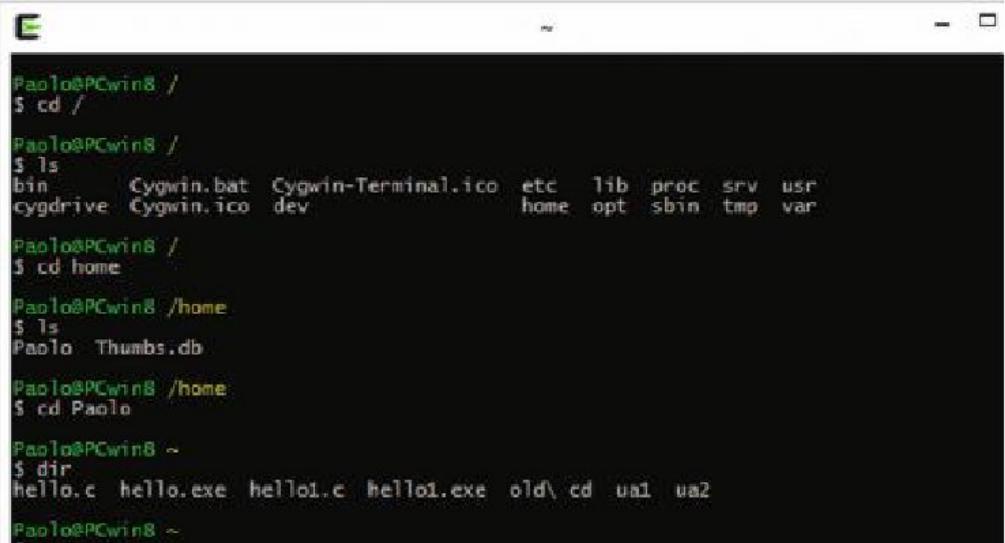
```
cd /
```

Quindi ne visualizziamo il contenuto col comando

```
ls /
```

Entriamo poi nella directory **home** col comando **cd home** e ne visualizziamo il contenuto: è presente la sola cartella col nostro nome, ci entriamo e visualizziamo l'elenco dei file presenti sempre con **ls** (oppure anche con il comando **dir**).

In figura possiamo vedere l'esito dei comandi sopra elenctati. ►



```

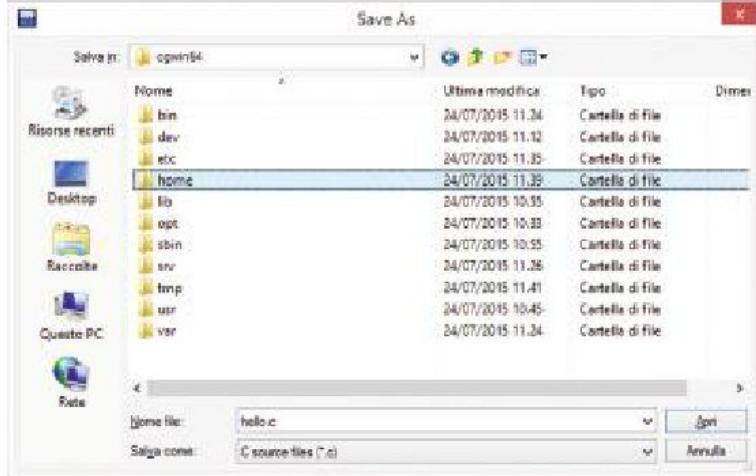
Paolo@PCwin8 /
$ cd /
Paolo@PCwin8 /
$ ls
bin      Cygwin.bat  Cygwin-Terminal.ico  etc      lib      proc    srv    usr
cygdrive  Cygwin.ico  dev                  home     opt      sbin   tmp    var
Paolo@PCwin8 /
$ cd home
Paolo@PCwin8 /home
$ ls
Paolo  Thumbs.db
Paolo@PCwin8 /home
$ cd Paolo
Paolo@PCwin8 ~
$ dir
hello.c  hello.exe  hello1.c  hello1.exe  old\cd  val  va2
Paolo@PCwin8 ~
$ 
  
```

Per scrivere il codice C è necessario usare un editor: si può usare il semplice **blocco note**, come nell'esempio di figura: ►

```
hello.c - Blocco note
File Modifica Formato Visualizza 
#include <stdio.h>
int main(){
    printf ("Hello mondo Cygwin \n\n");
    return 0;
}
Linea 9, colonna 1
```

... e salvare il file nella cartella personale (nel mio caso **paulo**) all'interno della subdirectory **home** della sotto cartella nella quale abbiamo installato **Cygwin**, nel nostro caso "**C:\cygwin64**": ►

e salviamolo con il nome **Hello.c**.



È anche possibile utilizzare come editor **Dev-cpp**, scaricabile gratuitamente all'indirizzo <http://www.bloodshed.net/devcpp.html> oppure dalla cartella **materiali** nella sezione riservata al presente volume sul sito www.hoepliscuola.it.

Se ora digitiamo **dir** al prompt dei comandi di **Cygwin** troviamo il nostro file sorgente pronto per essere compilato con **GCC**: lo compiliamo col comando

```
gcc Hello.c -o hello
```

Digitando nuovamente **dir** al prompt ora visualizzeremo due file: oltre al file sorgente troviamo il file eseguibile che mandiamo in esecuzione col comando:

```
./hello.exe
```

L'output e la sequenza dei comandi fin qui elencati è riportata nella successiva videata:

```
PaoLo@PCWin8 /home/paolo
$ ls
hello.c hellotest.c old cd parametri.c progetto1 progettoJava ual uel
$ cd /home/paolo
$ gcc hello.c -o hello.exe
$ ./hello.exe
PaoLo@PCWin8 /home/paolo
$ ls
hello.c hellotest.c parametri.c progetto1 progettoJava ual uel
hello.exe old cd parametri1 uel
$ ./hello
Hello mondo Cygwin
```

AREA digitale

I caratteri jolly

Abbiamo realizzato il nostro primo programma in C in ambiente **Cygwin** utilizzando il compilatore **GCC**!

GCC è un tool ricco di opzioni. Il manuale ufficiale lo si può trovare all'indirizzo: <http://gcc.gnu.org/onlinedocs/>.

Per esempio, per conoscere la versione di **GCC** che si sta utilizzando è possibile utilizzare il comando `gcc --ver`.



Prova adesso!

I programmi concorrenti che scriveremo necessitano di un insieme di librerie, riportate nella seguente figura che andremo ad aggiungere all'inizio di tutti i nostri codici C.

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <sys/wait.h>
4 #include <fcntl.h>
5 #include <stdio.h>
6 #include <unistd.h>
7 #include <stdlib.h>
-
```

Predisponi l'ambiente di sviluppo da te utilizzato in modo che alla creazione di un nuovo programma le istruzioni di figura siano già inserite automaticamente.

Scrivi, compila e manda in esecuzione un programma che legge una frase e la scrive in un file di testo.



Zoom su...

I COMANDI PER LE DIRECTORY

Partiamo dal presupposto che la **shell di Linux** riconosce due tipi di percorsi dei file: **assoluto** o **relativo**. Il percorso **assoluto** indica in modo completo la destinazione da raggiungere, partendo dalla **radice**. La **radice**, indicata con lo slash (/) è la **directory** principale del sistema, ogni altro **file** o **directory** è contenuto al suo interno. Un percorso assoluto inizia quindi sempre con la barra e indicherà la posizione di un file all'interno della **root**, per esempio:

```
/esempio/prova
```

Il percorso **relativo** indica il percorso di un file partendo dalla **posizione corrente** e pertanto non inizia con lo slash (/). Per esempio il comando seguente:

```
../prova
```

Indica un percorso che identifica la directory **prova** come "sorella" rispetto alla directory corrente, secondo lo **schema ad albero** delle directory, in quanto il simbolo del **punto punto** (..) indica la directory del livello superiore superiore da cui accedere alla directory **prova**, che risulterà in tal modo "sorella" di quella in cui ci troviamo.

Appena accediamo al sistema tramite cygwin siamo posizionati nella directory **/home** dell'utente, per identificare la posizione con il comando **pwd**:

```
E
Paolo@PCwin8 /home
$ pwd
/home
```

Vediamo adesso il comando che consente di visualizzare il contenuto di una directory, si tratta di **ls**, abbreviazione di **list**. Permette ottenere un elenco sul terminale dei file e delle sottocartelle presenti nella directory corrente, per esempio:

```
E
Paolo@PCwin8 /home
$ pwd
/home

Paolo@PCwin8 /home
$ ls
cygwin Paolo
```

Per poter visualizzare il contenuto della directory corrente possiamo anche usare il comando **long listing** format, che fornisce un elenco più completo e ricco di informazioni sui file e le directory, contenute, la sintassi è: **ls -l**:

```
E
/home
Paolo@PCwin8 /home
$ ls -l
totale 8
drwx-----+ 1 Paolo mkpasswd 0 ott 7 21:05 cygwin
drwx-----+ 1 Paolo mkpasswd 0 ott 7 20:47 Paolo

Paolo@PCwin8 /home
$ |
```

Nei percorsi relativi possiamo usare due simboli, il primo si chiama **current** e identifica la directory corrente (.) mentre il secondo, chiamato **parent**, identifica la directory del livello superiore di quella corrente (..). Possiamo in tal modo identificare una directory o un file utilizzando i caratteri punto (.) e punto punto (..).

Per cambiare la directory corrente si usa il comando **cd** seguito dal percorso della directory da raggiungere. Proviamo per esempio a cambiare la directory corrente da **/home/Paolo** alla directory **/home/paolo/Documenti**. Possiamo operare in due modi, usando un percorso **assoluto** oppure **relativo**.

Nel primo caso il comando sarà il seguente:

```
cd /home/Paolo/Documenti
```

È sempre necessario lasciare uno spazio tra il comando **cd** e il successivo parametro.

Come possiamo notare il percorso della directory in cui ci troviamo adesso è il seguente: `~/Documenti`, ricordando che il simbolo tilde (`~`) indica la directory `home` dell'utente (in questo caso **Paolo**)

```
Paolo@PCwin8 ~
$ cd /home/Paolo/Documenti

Paolo@PCwin8 ~/Documenti
$
```

Proviamo ad eseguire la medesima operazione (cambio di directory corrente), questa volta usando un percorso relativo, il comando sarà il seguente, ipotizzando di essere posizionati nella directory `/home`:

```
cd Paolo/Documenti
```

Il risultato è il medesimo, ossia ci troviamo adesso nella directory `/home/Paolo/Documenti`:

```
Paolo@PCwin8 ~
$ cd Paolo/Documenti

Paolo@PCwin8 ~/Documenti
$
```

Per tornare alla directory `/home` possiamo anche usare il comando `cd` senza parametri.

CREARE, COPIARE E CANCELLARE DIRECTORY

I comandi che consentono di **creare**, **cancellare** e **copiare** le directory sono i seguenti:-

- ▶ **mkdir**: **crea** una nuova **directory**;
- ▶ **rmdir**: **cancella** una **directory**, solo se vuota;
- ▶ **rm**: **cancella** una **directory** o un **file**;
- ▶ **cp**: **copia** una **directory** o un **file**.

Creare una nuova directory significa assegnare un nome a una cartella nel percorso desiderato, in grado di contenere file oppure altre cartelle. Copiare una directory significa copiarne anche tutto il contenuto. Cancellare una directory con **rm** significa eliminarla per sempre, mentre con **rmdir** possiamo eliminare solo una directory non vuota. Vediamo innanzi tutto come creare una nuova directory con il comando **mkdir**:

```
mkdir prova
```

In questo caso abbiamo creato la directory **prova** all'interno del **percorso corrente**. L'esempio che segue mostra come creare la directory **prova** dopo esserci posizionati nella directory `home`, creiamo la directory **prova** e visioniamo il risultato con **ls -l**:

```
Paolo@PCwin8 ~
$ mkdir prova

Paolo@PCwin8 ~
$ ls -l
totale 0
drwxr-xr-x+ 1 Paolo mkpasswd 0 ott 8 09:30 Documenti
drwxr-xr-x+ 1 Paolo mkpasswd 0 ott 8 09:44 prova
```

Desideriamo creare una directory come sotto cartella di **Documenti**, di nome **lavori**, il suo percorso assoluto sarà il seguente:

```
/home/Paolo/Documenti/lavori
```

Digitiamo il comando e quindi osserviamo se la directory è stata effettivamente creata, entrando in **Documenti** con **cd**:

```
Paolo@PCwin8 ~
$ mkdir /home/Paolo/Documenti/lavori

Paolo@PCwin8 ~
$ cd Documenti

Paolo@PCwin8 ~/Documenti
$ ls -l
totale 0
drwxr-xr-x+ 1 Paolo mkpasswd 0 ott 8 09:48 lavori
```



Prova adesso!

- Usare i comandi **mkdir**, **cd**, **ls**

- 1 Crea una directory di nome **amici** nella directory **Documenti**, con un percorso relativo.
- 2 Adesso prova a posizionarti nella directory appena creata sopra e visualizzane il contenuto.

Per **copiare** una cartella si utilizza il comando **cp -r** seguito dal percorso della directory da copiare e dal percorso di dove destinarla. Per esempio vogliamo copiare la cartella **lavori** nella directory **/home**. Il comando necessario, che utilizza percorsi assoluti, è il seguente:

```
cp -r /home/Paolo/Documenti/lavori /home
```

Otteniamo:

```
Paolo@PCwin8 ~/Documenti
$ cp -r /home/Paolo/Documenti/lavori /home

Paolo@PCwin8 ~/Documenti
$ cd /home

Paolo@PCwin8 /home
$ ls -l
totale 0
drwx-----+ 1 Paolo mkpasswd 0 ott 7 21:05 cygwin
drwxr-xr-x+ 1 Paolo mkpasswd 0 ott 8 09:55 lavori
drwx-----+ 1 Paolo mkpasswd 0 ott 8 09:44 Paolo
```

Il parametro **-r** è necessario per copiare le directory, mentre invece per copiare solo i file non dovrà essere utilizzato.

Per eliminare le directory possiamo usare due comandi, **rmdir** che elimina la directory solo se è vuota e **rm -r** che elimina la directory indipendentemente dal suo contenuto. Entrambi i comandi prevedono che il percorso della directory da eliminare venga scritto di seguito dopo lo spazio. Proviamo a eliminare la directory **lavori** presente nel percorso **/home**:

```
E /home
Paolo@PCwin8: /home
$ ls -l
totale 0
drwx-----+ 1 Paolo mkpasswd 0 ott 7 21:05 cygwin
drwxr-xr-x+ 1 Paolo mkpasswd 0 ott 8 09:55 lavori
drwx-----+ 1 Paolo mkpasswd 0 ott 8 09:44 Paolo

Paolo@PCwin8: /home
$ rmdir /home/lavori

Paolo@PCwin8: /home
$ ls -l
totale 0
drwx-----+ 1 Paolo mkpasswd 0 ott 7 21:05 cygwin
drwx-----+ 1 Paolo mkpasswd 0 ott 8 09:44 Paolo
```

Possiamo anche eliminare più cartelle indicandone i nomi di seguito separati dallo spazio, per esempio per eliminare due directory digitiamo i due percorsi subito dopo il comando di cancellazione:

```
rm -r /home/prova /home/Paolo/Documenti/lavori
```

ESEMPIO Creare una directory e un file e cancellarli

Adesso proviamo a creare una directory, scrivere al suo interno un file e cancellarla con il comando **rm -r** che elimina la directory e il suo contenuto. Prima di tutto creiamo la directory **esempio** nel percorso **/home** e verifichiamone l'avvenuta creazione:

```
E
Paolo@PCwin8: ~
$ mkdir esempio

Paolo@PCwin8: ~
$ ls -l
totale 0
drwxr-xr-x+ 1 Paolo mkpasswd 0 ott 8 09:48 Documenti
drwxr-xr-x+ 1 Paolo mkpasswd 0 ott 8 10:02 esempio
drwxr-xr-x+ 1 Paolo mkpasswd 0 ott 8 09:44 prova
```

Adesso posizioniamoci nella directory **esempio** e creiamo un file di nome **pippo** attraverso l'utile comando **touch** che crea un file vuoto secondo una tecnica che viene chiamata **on the fly** (al volo):

```
E ~/esempio
Paolo@PCwin8: ~
$ cd esempio

Paolo@PCwin8: ~/esempio
$ touch pippo

Paolo@PCwin8: ~/esempio
$ ls -l
totale 0
-rw-r--r-- 1 Paolo mkpasswd 0 ott 8 10:13 pippo
```

Proviamo ora a copiare il file **ippo** nella directory **/home** mediante il comando **cp** e ne verifichiamo l'avvenuta copia mediante **ls -l**:

```

Paolo@PCwin8 ~
$ cd esempio
Paolo@PCwin8 ~/esempio
$ touch ippo

Paolo@PCwin8 ~/esempio
$ cp ippo /home/Paolo
Paolo@PCwin8 ~/esempio
$ cd ..
Paolo@PCwin8 ~
$ ls -l
totale 0
drwxr-xr-x+ 1 Paolo mkpasswd 0 ott 8 09:48 Documenti
drwxr-xr-x+ 1 Paolo mkpasswd 0 ott 8 10:32 esempio
-rw-r--r--+ 1 Paolo mkpasswd 0 ott 8 10:33 ippo
drwxr-xr-x+ 1 Paolo mkpasswd 0 ott 8 09:44 prova

```

Adesso usciamo dalla directory, tornando alla directory del livello superiore (**/home/Paolo**) con il comando **cd..**; da qui digitiamo il comando di cancellazione e verifichiamo l'avvenuta eliminazione, ricordando che il comando **rm -r** elimina sia la directory che tutto il suo contenuto:

```

Paolo@PCwin8 ~/esempio
$ cd ..

Paolo@PCwin8 ~
$ rm -r esempio

Paolo@PCwin8 ~
$ ls -l
totale 0
drwxr-xr-x+ 1 Paolo mkpasswd 8 ott 8 09:48 Documenti
drwxr-xr-x+ 1 Paolo mkpasswd 8 ott 8 09:44 prova

```

Come possiamo notare attraverso il comando digitato sopra abbiamo eliminato sia la directory **esempio** che il file **ippo** contenuto in essa.



Prova adesso!

- Utilizzare i comandi **rm -r**, **cd**, **cp**, **ls**

- 1 Posizionati nella directory **/home** e crea una sottodirectory di nome **prova_adesso**.
- 2 Posizionati nella directory **prova_adesso**.
- 3 Torna alla directory **/home**.
- 4 Crea una sotto cartella della directory **/home** chiamata **prova**.
- 5 Posizionati adesso nella directory **prova**.
- 6 Posizionati nella directory **prova_adesso** usando un percorso relativo.
- 7 Crea un file nella directory **prova**.
- 8 Copia la directory **prova** sotto alla directory **prova_adesso**.

ESERCITAZIONI DI LABORATORIO 3

LA FORK IN C



I codici sorgenti sono nel file **C_fork.rar** scaricabile dalla cartella materiali nella sezione del sito www.hoepiscuola.it riservata al presente volume.

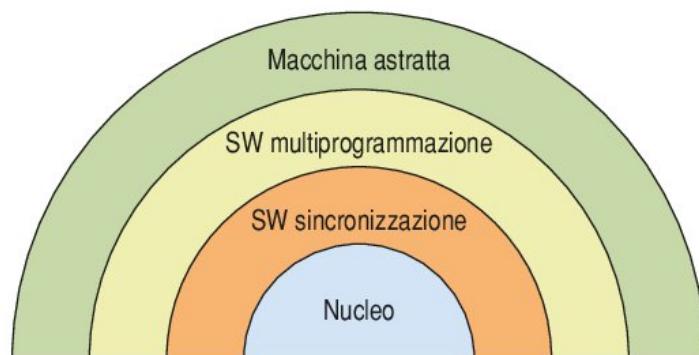
■ Macchina astratta

Per scrivere e collaudare programmi concorrenti è necessario avere a disposizione una **macchina concorrente** in grado di eseguire più processi sequenziali contemporaneamente e un **linguaggio di programmazione** con il quale descrivere algoritmi non sequenziali.

Il **linguaggio C** permette di descrivere programmi composti da un insieme di **processi sequenziali asincroni interagenti** e in questa esercitazione analizzeremo come scrivere e collaudare processi paralleli, cioè algoritmi che per loro natura hanno una sequenza delle attività con **ordinamento parziale**, dove l'esecutore per alcune istruzioni “è libero” di scegliere quali iniziare prima senza che il risultato sia compromesso: può quindi anche “portare avanti” l'elaborazione di questi segmenti di codice in **parallelo**.

Il nostro esecutore, essendo un normale personal computer, non ha tante unità di elaborazione quanti sono i processi da svolgere e quindi realizzerà una macchina concorrente astratta con tecniche software o direttamente grazie alle primitive del sistema operativo.

Quindi nei sistemi monoprocessoressi la macchina parallela è una macchina astratta che possiamo vedere strutturata nei seguenti livelli: ►



Il nucleo corrisponde all'esecutore del programma compilato in linguaggio concorrente che dispone di un insieme di meccanismi di sincronizzazione che permettono la realizzazione di applicazioni multiprogrammate.

In questa esercitazione inizieremo a realizzare gerarchie di processi mediante l'istruzione **fork** e nella prossima esercitazione implementeremo i costrutti fondamentali descritti ad alto livello nella lezione 5 della corrente unità di apprendimento:

- il costrutto **fork-join**;
- il costrutto **cobegin-coend**.

■ L'istruzione fork

Scriviamo ora il primo programma che genera un processo figlio eseguendo una operazione di **fork**. La sintassi dell'istruzione è la seguente:

```
int fork();
```

che per essere eseguita necessita dell'inclusione della seguente libreria:

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

La funzione **fork** serve per **creare** un processo figlio identico al processo padre e tutti i segmenti del padre vengono duplicati nel figlio al momento della esecuzione **fork**.

Il processo figlio viene creato con questa istruzione e viene mandato in esecuzione proprio dall'istruzione successiva: quindi dopo l'istruzione **e fork** avremo due processi in evoluzione, il padre e il figlio.

La **fork** ha come parametro di ritorno un dato **integer** che ha valori differenti nei due processi:

- nel processo padre è il **PID** del processo appena creato;
- nel processo figlio è uguale a 0.

Solo la variabile pid di ritorno dalla fork ha valori diversi nel padre e nel figlio in quanto il processo figlio è una "copia perfetta" del processo padre, inclusa anche la sua area dati: quindi anche il processo figlio procede l'elaborazione con gli esiti delle elaborazioni del processo padre.

È possibile sfruttare il valore della variabile **pid** per distinguere il padre dal figlio in modo da far eseguire a ciascuno le istruzioni desiderate: dato che entrambi condividono il codice, cioè dato che dopo l'esecuzione della **fork** vengono eseguite parallelamente le istruzioni da padre e figlio, se inseriamo un'istruzione di selezione:

```
if (pid == 0)
```

questa darà risultato **VERO** se è il processo figlio a eseguirla mentre darà risultato **FALSO** se è il processo padre che la esegue: quindi, dopo questo test, nei due rami della selezione inseriamo le istruzioni specifiche che devono essere eseguite o dal padre oppure dal figlio.

Terminazione di un processo

Un processo può terminare:

- **involontariamente**, se per esempio cerca di effettuare operazioni illegali oppure mediante una interruzione;

- **volontariamente**, o perché ha finito tutte le istruzioni oppure con la chiamata alla funzione **exit()**.

La funzione **exit()** ha la seguente sintassi:

```
void exit(int status) ;
```

Prevede un parametro (**status**) mediante il quale il **processo** che termina può comunicare al padre informazioni sul suo stato di terminazione; generalmente è un intero a 16 bit ed ha i seguenti significati:

- se il byte meno significativo di **status** è zero, il più significativo rappresenta lo stato di terminazione (**terminazione volontaria**, per esempio il valore indicato nella funzione **exit**);
 ► in caso contrario, il byte meno significativo di **status** descrive il segnale che ha terminato il figlio (**terminazione involontaria**) e il suo valore viene “forzato” dal sistema operativo.

Se il valore di **status** è minore di 256 la terminazione è volontaria e, quindi, l'elaborazione è corretta e il valore di **status** è quello che viene passato dal figlio come parametro: è possibile utilizzare il secondo byte per comunicare dati al processo padre sempre tenendo presente che il range dei valori possibili si limita a 0-255.

Vediamo un semplice esempio, evidenziando il codice che viene eseguito dai due processi.

Codice eseguito dal processo padre	Codice eseguito dal processo figlio
<pre>int main(){ int pid; printf("1) prima della fork \n"); pid = fork(); // creo processo figlio ---> printf(" 2) dopo della fork \n"); if (pid == 0){ printf(" 3) sono il processo figlio\n"); exit(1); } else{ printf(" 3) sono il processo padre\n"); exit(0); // termina padre } }</pre>	<pre>int main(){ int pid; printf("1) prima della fork \n"); pid = fork(); printf(" 2) dopo della fork \n"); if (pid == 0){ printf(" 3) sono il processo figlio\n"); exit(1); } else{ printf(" 3) sono il processo padre\n"); exit(0); // termina padre } }</pre>

Completiamo la codifica inserendo le direttive per il compilatore:

```
fork1.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int pid;
6     printf( "1) prima della fork \n" );
7     pid = fork(); // crea processo figlio
8     printf( " 2) dopo della fork \n" );
9     if (pid == 0){
10         printf( " 3) sono il processo figlio\n" );
11         exit(1); // termina il processo figlio
12     }
13     else{
14         printf( " 3) sono il processo padre\n" );
15         exit(0); // non necessaria
16     }
17 }
```

Avviandone l'esecuzione otteniamo il seguente risultato sullo schermo:



```

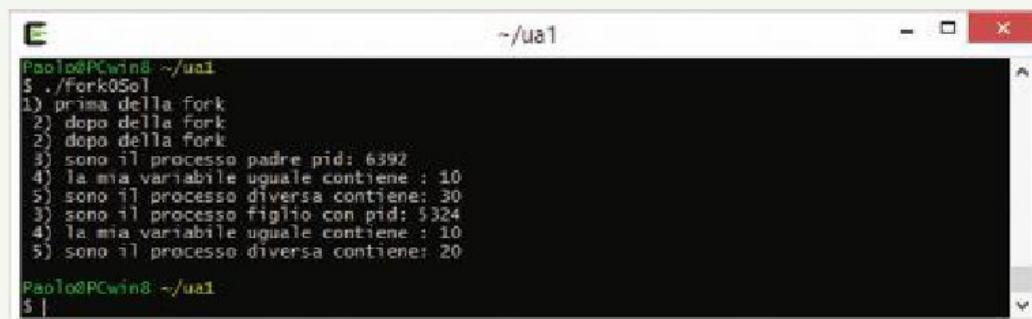
Paolo@PCwin8 ~/ual
$ gcc fork0.c -o fork0
Paolo@PCwin8 ~/ual
$ ./fork0
1) prima della fork
2) dopo della fork
3) sono il processo padre
2) dopo della fork
3) sono il processo figlio
Paolo@PCwin8 ~/ual
$ 
```

Si vede chiaramente come l'istruzione 8 venga eseguita due volte, sia dal processo padre che dal processo figlio: successivamente i due processi "scelgono" il ramo opportuno dell'istruzione **if** di riga 9 in base al valore della variabile **pid** che, ripetiamo, per il processo figlio ha valore 0.



Prova adesso!

- 1** Inserisci alcune variabili nel programma e assegna loro valori a piacere prima e dopo l'istruzione **fork**.
- 2** Modifica ora il programma facendo il test non sul **pid == 0** ma confrontandolo con il valore che viene generato alla sua creazione, in modo da "invertire" i due rami della selezione.
- 3** All'interno dei due rami, oltre alla frase di saluto visualizza anche il valore del **pid** utilizzando la funzione **getpid()**, in modo da ottenere un output simile al seguente:



```

Paolo@PCwin8 ~/ual
$ ./fork0Sol
1) prima della fork
2) dopo della fork
2) dopo della fork
3) sono il processo padre pid: 6392
4) la mia variabile uguale contiene : 10
5) sono il processo diversa contiene: 30
3) sono il processo figlio con pid: 5324
4) la mia variabile uguale contiene : 10
5) sono il processo diversa contiene: 20
Paolo@PCwin8 ~/ual
$ 
```

Confronta il tuo codice con quello riportato nel file **fork0Sol.c**.

■ PID del padre e del figlio

Ogni processo figlio “si ricorda” il **PID** del **processo** padre che prende il nome di **parent pid** o **PPID**. Il linguaggio C mette a disposizione due funzioni che permettono di sapere per il processo corrente quale è il proprio **PID** e quale è il **PID** del suo genitore:

```
int getpid()           //PID proprio
int getppid()          //PID del padre
```

Utilizziamole per completare l’esempio precedente:

```
fork2.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int pid;
6     printf( "1) prima della fork \n" );
7     pid = fork();           // creo processo figlio
8     printf( "2) dopo della fork \n" );
9     if (pid == 0){
10         printf( "3) sono il processo figlio con pid:%d.", getpid() );
11         printf( " Il mio papi ha pid: %d\n", getppid() );
12         exit(1);           // termina il processo figlio
13     }
14     else{
15         printf( "3) sono il processo padre con pid:%d.", getpid() );
16         printf( " Il mio papi ha pid: %d\n", getppid() );
17         exit(0);           // non necessaria
18     }
19 }
```

Avviandone l’esecuzione otteniamo il seguente risultato sullo schermo:

```
Paolo@PCwin8 ~/uad/c_fork
$ gcc fork2.c -o fork2
Paolo@PCwin8 ~/uad/c_fork
$ ./fork2
1) prima della fork
2) dopo della fork
3) sono il processo Figlio con pid:9604. Il mio papi ha pid: 5484
```

È anche possibile visualizzare **PID** e **PPID** su tutti i processi che sono in esecuzione digitando da console il comando:

PID	PPID	PGID	WINPID	TTY	UID	STIME	COMMAND
4092	10336	4092	9256	pty0	197609	15:58:44	/usr/bin/ps
10336	4368	10336	2220	pty0	197609	15:57:53	/usr/bin/bash
4368	1	4368	4368	?	197609	15:57:53	/usr/bin/mintty

Modifichiamo ora il codice aggiungendo una istruzione del figlio rispetto al padre; a tal fine utilizziamo la seguente funzione

```
void sleep(int secondi)
```

che “addormenta” il processo per un numero parametrico di secondi. Inseriamo quindi l’istruzione 11:

```

8   |    if (pid == 0){
9   |      printf( " 3) sono il processo figlio con pid:%d.", getpid() );
10  |      sleep(3);           // ritardo di 3 secondi
11  |      printf( " Il mio papi ha pid: %d\n", getppid()); // termina il processo figlio
12  |
13  |      exit(1);
14 }
```

Aggiungiamo a ogni istruzione la funzione `getpid()` in modo da poter individuare il processo che la esegue e mandiamo in esecuzione il programma ottenendo:

```

Paolo@PCwin8 ~/ua1/l3_c_fork
$ gcc fork3.c -o fork3
Paolo@PCwin8 ~/ua1/l3_c_fork
$ ./fork3
1) prima della fork 5992
2) dopo della fork 5992
3) sono il processo padre con pid:5992. Il mio papi ha pid: 5796
2) dopo della fork 3208
Paolo@PCwin8 ~/ua1/l3_c_fork
$ 3) sono il processo figlio con pid:3208. Il mio papi ha pid: 1

```

Osserviamo come il **processo padre** termina prima del **processo figlio** lasciandolo, di fatto, **orfano**: lo possiamo riconoscere dal fatto che il programma principale è terminato e sullo schermo appare il **prompt** dei comandi e solo in un successivo tempo viene visualizzata la riga scritta dal processo figlio con l’istruzione 12.

Inoltre il processo figlio in questa riga “dichiara” di avere come padre il processo 1, ma questo è il **PID** del processo `init()` che lo ha “adottato”.

Vedremo come rimediare a questa situazione nelle prossime lezioni.



Prova adesso!

- Istruzione `fork`
- Identificatore di processo `PID`

Scrivi un programma dove un padre genera un numero di processi figli definiti dall’operatore mediante la lettura di un valore intero mediante una funzione `faiFiglio()`.

Visualizza per ciascuno di essi il proprio **PID** e quello del padre.

Confronta la tua soluzione con quella presente nel file `fratelliSol.c`.

Quindi prova a togliere l’istruzione `exit(0)` all’interno del segmento eseguito dal figlio: cosa ti aspetti che venga visualizzato? Perche?

ESERCITAZIONI DI LABORATORIO 4

FORK ANNIDATE ED ESECUZIONE NON DETERMINISTICA



Info

I codici sorgenti sono nel file **C_fork.rar** scaricabile dalla cartella materiali nella sezione del sito www.hoeplicuola.it riservata al presente volume.

Riprendiamo il codice dell'esercizio **fork2.c** e modifichiamolo in modo da effettuare tre **fork()**, come si può osservare nel seguente programma alle istruzioni 5, 6 e 7:

```
fork4.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int pid1, pid2, pid3;
5     pid1 = fork();                                // creo processo figlio
6     pid2 = fork();                                // creo processo figlio
7     pid3 = fork();                                // creo processo figlio
8     if ((pid1 == 0) || (pid2 == 0) || (pid3 == 0)) { // se uno è 0 è un figlio
9         printf( " Sono il processo figlio con pid:%d.", getpid() );
10        printf( " Il mio papà ha pid: %d\n", getppid() );
11        sleep(1);                                 // attesa per non creare orfani
12        exit(1);                                 // termina il processo figlio
13    }
14    else{
15        sleep(2);
16        printf( "Sono il processo padre con pid:%d. \n", getpid() );
17    }
18    return 0;
19 }
```



Prova adesso!

Prima di mandare in esecuzione il programma, rispondi alle seguenti domande:

- ▶ Cosa ti aspetti di vedere sullo schermo?
- ▶ Quanti figli vengono generati?
- ▶ Perché?

Una esecuzione produce il seguente output, dove sono stati colorati i PID uguali per meglio individuarli:

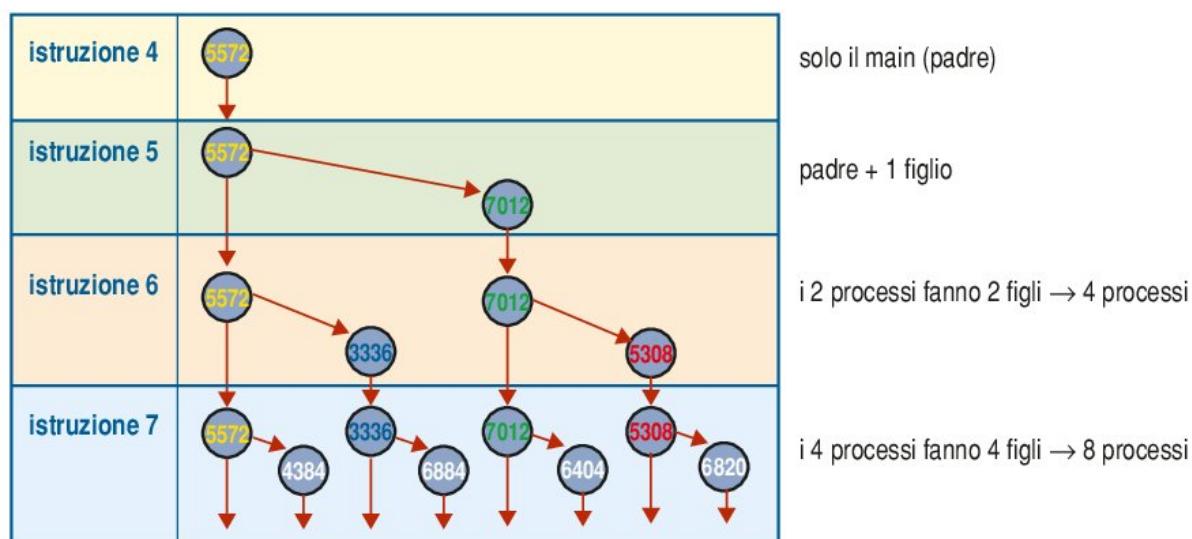
```

Paolo@PCwin8 ~/ua1/13_c_fork
$ ./fork4
Sono il processo figlio con pid:4384. Il mio papi ha pid: 5572
Sono il processo figlio con pid:5308. Il mio papi ha pid: 7012
Sono il processo figlio con pid:6820. Il mio papi ha pid: 5308
Sono il processo figlio con pid:3336. Il mio papi ha pid: 5572
Sono il processo figlio con pid:6884. Il mio papi ha pid: 3336
Sono il processo figlio con pid:7012. Il mio papi ha pid: 5572
Sono il processo figlio con pid:6404. Il mio papi ha pid: 7012
Sono il processo padre con pid:5572.

Paolo@PCwin8 ~/ua1/13_c_fork
$ 
```

Vengono quindi generati **sette processi figli** (e non tre come probabilmente hai erroneamente risposto).

Per comprendere la dinamica del programma disegniamo graficamente l'elenco delle **fork** e assegniamo a ogni processo il suo **PID**:



La prima **fork()** (istruzione 5) genera un processo che a sua volta esegue la seconda **fork()** (istruzione 6), quindi i primi due processi generano altri due figli che a loro volta eseguiranno la **fork()** della istruzione 7: in totale sono (padre + figlio) + 2 processi + 4 processi per un totale di otto processi (2^3 processi).

Se quindi si facessero 10 **fork()** di seguito si genererebbero 2^{10} figli!

Trasformiamo ora il programma che genera più figli in modo da creare una gerarchia di N livelli, cioè dove viene letto come parametro il numero di antenati desiderati e ciascuno crea il proprio discendente passandogli il parametro decrementato.

A tal fine effettuiamo una **chiamata ricorsiva** all'interno del segmento di codice **faiFiglio()**, in modo da richiamare la funzione che esegue la **fork**:

```

10  void faiFiglio (int quanti){
11      int pid;
12      pid = fork();           // creo processo figlio
13      if (pid == 0){
14          mettiSpazi(quanti);
15          printf( "Sono il processo figlio con pid:%d.", getpid() );
16          printf( "Il mio papi ha pid: %d\n", getppid());
17          if (quanti > 0)
18              faiFiglio(quanti-1);    // il figlio diventa il padre
19          else
20              exit(0);
21      }
22      else{
23          mettiSpazi(quanti);
24          printf( "Sono il processo padre con pid:%d.\n", getpid() );
25      }
26  }

```

Per meglio visualizzare sullo schermo la gerarchia introduciamo una semplice funzione che indenta gli output che ogni processo produce in modo da “visualizzare anche graficamente” la gerarchia.

```

[*] gerarchia.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define NRFIGLI 4
4  void mettiSpazi(int quanti){      // per meglio vedere la gerarchia
5      int x;
6      for (x = 0; x <= quanti; x++)
7          printf(" ");
8  }

```

Avviandone l'esecuzione otteniamo il seguente risultato sullo schermo:

```

Paolo@PCwin8 ~/ua1
$ ./gerarchia
    Sono il processo padre con pid:32.
    Termino elaborazione processo con pid:32.
    Sono il processo figlio con pid:8072.Il mio papi ha pid: 32

Paolo@PCwin8 ~/ua1
$     Sono il processo padre con pid:8072.
    Termino elaborazione processo con pid:8072.
    Sono il processo figlio con pid:5612.Il mio papi ha pid: 8072
    Sono il processo padre con pid:5612.
    Termino elaborazione processo con pid:5612.
    Sono il processo figlio con pid:4888.Il mio papi ha pid: 5612
    Sono il processo padre con pid:4888.
    Termino elaborazione processo con pid:4888.
    Sono il processo figlio con pid:6916.Il mio papi ha pid: 4888
    Sono il processo padre con pid:6916.
    Termino elaborazione processo con pid:6916.
    Sono il processo figlio con pid:5236.Il mio papi ha pid: 6916

```

Osserviamo come l'output sullo schermo evidenzi il fatto che il **main()** termina prima dei processi figli che lui stesso ha generato.



Prova adesso!

- Istruzione fork ricorsiva
- Identificatore di processo PID

Per ottenere un output "ordinato" è necessario introdurre dei ritardi nella esecuzione dei processi, in modo che i padri "attendano" la terminazione dei figli.
Confronta la tua soluzione con quella presente nel file `gerarchiaSol.c`.

Attenzione! Con l'introduzione della funzione `sleep()` **NON** stiamo effettuando alcuna **sincronizzazione**: abbiamo semplicemente rallentando il processo padre in modo che termini dopo il proprio processo figlio.

I meccanismi di **sincronizzazione** verranno descritti a partire dalla prossima lezione.

Esecuzione non deterministica

L'esecuzione di un programma parallelo viene influenzata dallo scheduler del sistema operativo e quindi l'output sullo schermo può essere diverso per ogni sua esecuzione.
Per verificarlo scriviamo un semplice programma.



Prova adesso!

- Generazione processi
- Esecuzione sequenziale

Scrivi un programma concorrente dove un processo padre genera solamente due processi figli numerando ogni istruzione di output in modo da poterne "seguire" la cronologia di esecuzione sullo schermo.

Mandalo in esecuzione più volte confrontando il risultato.

Una possibile implementazione è riportata di seguito:

```
fork5.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 void main(){
4     int pid, pid1, pid2;
5     pid1 = fork();
6     if(pid1 == 0){      // processo figlio
7         printf("1) sono il primo processo figlio con pid: %i\n", getpid());
8         exit(1);        // termina primo processo figlio
9 }
```

```

10 } else{
11     printf("2) sono il processo padre\n");
12     printf("3) ho creato un processo con pid: %i\n", pid1);
13     printf("4) il mio pid e' invece: %i\n", getpid());
14     pid2 = fork();
15 } if (pid2 == 0){ // secondo processo figlio
16     printf("5) sono il secondo processo figlio con pid: %i\n", getpid());
17     exit(2); // termina secondo processo figlio */
18 }
19 else {
20     printf("6) sono il processo padre\n");
21     printf("7) ho creato un secondo processo con pid: %i\n", pid2);
22 }
23 }
24 }
```

Una prima esecuzione dà il seguente output:

```

~/ua1/l3_c_fork
$ gcc fork5.c -o fork5
Paolo@PCwin8 ~/ua1/l3_c_fork
$ ./fork5
1) sono il primo processo figlio con pid: 1896
2) sono il processo padre
3) ho creato un processo con pid: 1896
4) il mio pid e' invece: 8152
5) sono il processo padre
6) ho creato un secondo processo con pid: 7904
5) sono il secondo processo figlio con pid: 7904
Paolo@PCwin8 ~/ua1/l3_c_fork
$ |
```

Una seconda esecuzione dà il seguente output:

```

~/ua1/l3_c_fork
Paolo@PCwin8 ~/ua1/l3_c_fork
$ ./fork5
2) sono il processo padre
3) ho creato un processo con pid: 1652
4) il mio pid e' invece: 100
1) sono il primo processo figlio con pid: 1652
6) sono il processo padre
7) ho creato un secondo processo con pid: 8160
5) sono il secondo processo figlio con pid: 8160
Paolo@PCwin8 ~/ua1/l3_c_fork
$ |
```

Possiamo osservare che entrambe le esecuzioni non rispettano l'ordinamento sequenziale delle istruzioni e ogni esecuzione ha un ordine diverso, confermando come la schedulazione giochi un ruolo fondamentale nella evoluzione dei processi.

Per ottenere esecuzioni deterministiche è necessario introdurre la sincronizzazione, come vedremo nella prossima lezione.



Prova adesso!

- Generazione processi
- Esecuzione sequenziale

Scrivi un programma che utilizza l'istruzione di **switch** per intercettare l'esecuzione dei due processi prendendo il **PID** come selettore.

Confronta la tua soluzione con quella presente nel file **fork6.c**.

Successivamente modifica il programma **fork5.c** in modo che ciascun figlio generi un processo in modo da avere una gerarchia a tre livelli.

Manda in esecuzione più volte il programma osservando la sequenza non deterministica delle istruzioni.

Confronta la tua soluzione con quella presente nel file **fork5Sol.c**.

ESERCITAZIONI DI LABORATORIO 5

LE FUNZIONI WAIT() E WAITPID()



I codici sorgenti sono nel file **C_fork_join.rar** scaricabile dalla cartella materiali nella sezione del sito www.hoepliscuola.it riservata al presente volume.

■ La funzione wait()

Nella esercitazione precedente abbiamo visto come l'elaborazione parallela di più processi necessita di un meccanismo di sincronizzazione affinché il primo processo che termina il proprio compito possa "comunicare" agli altri il risultato della propria elaborazione altrimenti l'esecuzione avviene in modo NON deterministico, a causa delle politiche di schedulazione del **sistema operativo**.

Sia il costrutto **fork-join** che il **cobegin-coend** necessitano di un meccanismo mediante il quale un processo possa "aspettare" gli altri processi che non hanno ancora finito di produrre i loro risultati.

L'istruzione che permette a un processo padre di attendere e leggere lo stato di terminazione del processo figlio è la system call **wait()**, che ha la seguente sintassi:

```
pid_t wait(int *status);
```

Il parametro **status** è l'indirizzo della variabile in cui viene memorizzato lo stato di terminazione del figlio mentre il valore di ritorno è il **pid** del processo terminato oppure un codice di errore (<0): la variabile indirizzata da **status** è quindi utilizzata come secondo parametro di ritorno dato che viene letto dal processo padre.

pid_t è un tipo di dato definito nel linguaggio C per rappresentare l'ID del processo: sostanzialmente è un numero intero con segno.

Per utilizzare il tipo **pid_t** può essere necessario includere come **header files** la libreria **sys/types.h**

In generale il tipo **pid_t** viene utilizzato quando si vuole svincolare il codice dal tipo di macchina/sistema operativo oppure quando non si conosce come esso viene implementato, in quanto potrebbe essere un **short int**, un **int**, un **long int**, oppure un **long long int**: nella libreria **GNU** è un semplice **int**, e quindi la funzione può essere anche così scritta:

```
int wait(int *status);
```

AREA digitale

Why should pid_t be used?

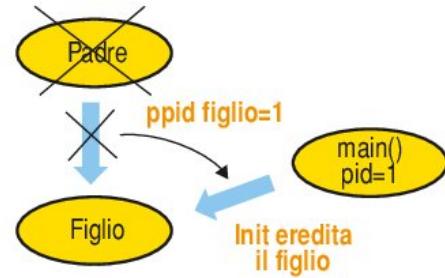
Vediamo gli effetti della chiamata di una **wait()**: dato che un processo che chiama la funzione **wait()** può avere figli in esecuzione abbiamo alcune possibili situazioni:

- se nessun figlio è terminato, il processo si sospende in attesa della *terminazione del primo di essi*;
- se almeno un figlio è già terminato, il processo padre NON **si sospende** per quel figlio e **wait()** ritorna immediatamente il suo stato di terminazione;
- se non esiste neanche un figlio, **wait()** NON è suspensiva e ritorna un codice di errore (valore ritornato < 0).

Terminazioni particolari di un processo

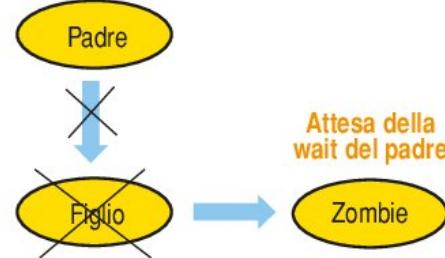
A Terminazione del padre

Se il processo che termina ha figli in esecuzione, il processo **main()** adotta i figli dopo la terminazione del padre e nella process structure di ogni figlio al **pid** del processo padre viene assegnato il **valore 1**.



B Terminazione del figlio

Se il processo termina prima che il padre ne rilevi lo stato di terminazione il figlio passa nello stato **zombie**, tranne quando si trova nella situazione precedente, cioè se è stato "adottato" dal **main()** che rileva automaticamente il suo stato di terminazione.



Cerchiamo di comprendere le diverse situazioni mediante esempi.

AREA digitale

Rilevazione dello stato di terminazione

■ Alcuni esempi di attesa processi con wait

Come primo esempio il processo padre genera un solo processo figlio e lo attende con `wait()`.

ESEMPIO

Scriviamo un primo programma che genera un figlio e utilizziamo la funzione `wait()` in modo che il padre ne attenda la terminazione indicando se la sua esecuzione è terminata in modo corretto.

```
forkwait0.c
1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <fcntl.h>
4 #include <stdlib.h>
5
6 int main() {
7     int status;
8     int pid = fork();           // creazione del figlio
9     if (pid == 0) {
10         printf("Sono il figlio, il mio pid e': %d. ", getpid());
11         printf("Il mio papi ha pid: %d\n", getppid());
12         exit(69);
13     }
14     else {
15         printf("Sono il padre, il mio pid e': %d. ", getpid());
16         printf("L'exit di mio figlio (%d) e': %d\n", wait(&status), status );
17         return 0;
18     }
19 }
```

Compiliamo il programma e mandiamolo in esecuzione due volte.

```
Paolo@PCwin8 ~/uai
$ gcc forkwait0.c -o fw0.exe
Paolo@PCwin8 ~/uai
$ ./fw0
Sono il Figlio, il mio pid e': 1292. Il mio papi ha pid: 6704
Sono il padre, il mio pid e': 6704. L'exit di mio figlio (1292) e': 0

Paolo@PCwin8 ~/uai
$ ./fw0
Sono il Figlio, il mio pid e': 7740. Il mio papi ha pid: 4352
Sono il padre, il mio pid e': 4352. L'exit di mio figlio (7740) e': 0
```

Possiamo vedere come a ogni esecuzione il PID assegnato ai processi è sempre diverso ma il padre termina sempre dopo il figlio.

Il codice può essere migliorato distinguendo e commentando le due possibili situazioni di terminazione, come riportato nel seguente esempio:

```
forkwait1.c
1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 int main(){
5     int pid, status;
6     pid = fork();
7     if (pid == 0){           // ramo eseguito dal solo processo figlio
8         printf("codice eseguito dal figlio \n");
9         exit(0);
10    }
11    else{                  // ramo eseguito dal solo processo padre
12        pid = wait(&status); // attesa terminazione
13        printf("terminato processo figlio n.%d\n", pid);
14        if ((char)status == 0) // controllo terminazione figlio
15            printf("terminazione volontaria con stato %d\n", status );
16        else
17            printf("terminazione errata con segnale %d\n", (char)status);
18    }
19 }
```

Proviamo ora a togliere l'istruzione **wait()** dalla riga 16 del primo esempio:

```
14 else {
15     printf("Sono il padre, il mio pid e': %d. ", getpid());
16     printf("L'exit di mio figlio e': %d\n", status );
17 }
18 }
```

Compiliamo il programma e mandiamolo in esecuzione due volte.

```

Paolo@PCwin8 ~\ual
$ gcc forkwait0x.c -o fw0x.exe
Paolo@PCwin8 ~\ual
$ ./fw0x
Sono il padre, il mio pid e': 4048. L'exit di mio figlio e': 0
Sono il figlio, il mio pid e': 6780. Il mio papi ha pid: 4048

Paolo@PCwin8 ~\ual
$ ./fw0x
Sono il padre, il mio pid e': 6608. L'exit di mio figlio e': 0
Sono il figlio, il mio pid e': 6684. Il mio papi ha pid: 6608

Paolo@PCwin8 ~\ual
$
```

Notiamo che il processo padre termina prima del figlio: per effettuare meglio questa osservazione scriviamo un programma che genera più figli.

Come secondo esempio il padre genera più figli e li attende con **wait()**.

ESEMPIO

Scriviamo un programma dove il processo padre procede a istanziare quattro figli:

```
forkwait5.c
1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <fcntl.h>
4 #include <stdlib.h>
5
6 #define NFIGLI 4
7
8 int main() {
9     int status, pid, xj
10    for (x= 0; x < NFIGLI; x++) {
11        pid = fork();           // creazione del figlio
12        if (pid == 0) {         // nel figlio pid è uguale a 0
13            printf(" Sono il figlio, il mio pid e': %d. ", getpid());
14            printf(" Il mio ppid ha pid: %d\n", getppid());
15            exit(0);
16        }
17        else {                // nel padre ha valore > 0
18            printf("Sono il padre, il mio pid e': %d. ", getpid());
19            printf("Il pid del figlio corrente e': %d. \n", pid);
20            printf("L'exit di mio figlio (%d) e': %d\n\n", wait(&status), status );
21        }
22    }
23    return 0;
24 }
```

Una sua esecuzione produce il seguente output:

```
Paolo@PCwin8 ~/ual
$ gcc forkwait5.c -o fw5.exe
Paolo@PCwin8 ~/ual
$ ./fw5
Sono il padre, il mio pid e': 4136. Il pid del figlio corrente e': 7868.
Sono il figlio, il mio pid e': 7868. Il mio ppid ha pid: 4136
L'exit di mio figlio (7868) e': 0

Sono il padre, il mio pid e': 4136. Il pid del figlio corrente e': 6372.
Sono il figlio, il mio pid e': 6372. Il mio ppid ha pid: 4136
L'exit di mio figlio (6372) e': 0

Sono il padre, il mio pid e': 4136. Il pid del figlio corrente e': 6684.
Sono il figlio, il mio pid e': 6684. Il mio ppid ha pid: 4136
L'exit di mio figlio (6684) e': 0

Sono il padre, il mio pid e': 4136. Il pid del figlio corrente e': 3944.
Sono il figlio, il mio pid e': 3944. Il mio ppid ha pid: 4136
L'exit di mio figlio (3944) e': 0

Paolo@PCwin8 ~/ual
$
```

Il processo padre esegue quattro volte sia la generazione del figlio che il ramo else della istruzione 12, cioè esegue 4 istruzioni di **wait()** e quindi attende uno per uno i quattro figli che ha generato: le coppie padre-figlio vengono eseguite sequenzialmente.

Ora togliamo l'istruzione di riga 20 dove è presente la **wait()**:

```
17 else {                  // nel padre ha valore > 0
18     printf("Sono il padre, il mio pid e': %d. ", getpid());
19     printf("Il pid del figlio corrente e': %d. \n", pid);
20     // printf("L'exit di mio figlio (%d) e': %d\n\n", wait(&status), status )
21 }
```

Mandiamo in esecuzione il programma ottenendo:



```

Paolo@PCwin8 ~/ua1
$ gcc forkwait5a.c -o fw5.exe
Paolo@PCwin8 ~/ua1
$ ./fw5
Sono il padre, il mio pid e': 7640. Il pid del figlio corrente e': 7184.
Sono il figlio, il mio pid e': 7184. Il mio papi ha pid: 7640
Sono il padre, il mio pid e': 7640. Il pid del figlio corrente e': 3872.
Sono il figlio, il mio pid e': 3872. Il mio papi ha pid: 7640
Sono il padre, il mio pid e': 7640. Il pid del figlio corrente e': 4988.
Sono il figlio, il mio pid e': 4988. Il mio papi ha pid: 7640
Sono il padre, il mio pid e': 7640. Il pid del figlio corrente e': 6756.
Sono il figlio, il mio pid e': 6756. Il mio papi ha pid: 7640
Paolo@PCwin8 ~/ua1
$ |

```

È immediato verificare come il processo padre termini sempre prima del corrispondente processo figlio.



Prova adesso!

- Istruzione fork()
- Istruzione wait()
- Istruzione sleep()

Modifica il programma inserendo dei cicli di attesa nel segmento eseguito dai figli utilizzando l'istruzione **sleep(int x)** che sospende "addormenta" per x secondo l'esecuzione del processo che la esegue. Fai in modo che ogni figlio "dorma" per un numero diverso di secondi. Confronta la tua soluzione con quella presente nel file **forkwait5aSol.c**
 Verifica infine cosa succede se si dimentica di mettere **exit()** nel ramo eseguito dal figlio (riga 15).
 Confronta la tua soluzione con quella presente nel file **forkwait5bSol.c**

ESEMPIO

Come terzo esempio modifichiamo il codice in modo che sia un processo a generare più figli e attendere solo il primo che termina con una unica **wait()**: per garantirci la casualità nel tempo di elaborazione di ogni figlio introduciamo un ritardo parametrico con valore generato dalla funzione **rand()**.

```

forkwait2.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int pid, pid1, pid2, status, x;
5     srand(time(NULL));
6     pid1 = fork();
7     if(pid1 == 0){           // primo processo figlio
8         x = (rand() % 4) + 1;
9         printf("1) sono il primo processo figlio con pid: %i sleep: %d\n", getpid(), x);
10        sleep(x);
11        exit(1);            // termina primo processo figlio
12    }

```

```

13 } else{
14     pid2 = fork();
15     if (pid2 == 0){           // secondo processo figlio
16         x = (rand() % 2) + 1;
17         printf("2) sono il secondo processo figlio con pid: %i sleep: %d\n", getpid(),x);
18         sleep(x);
19         exit(2);             // termina secondo processo figlio
20     }
21     else {
22         printf("3) padre in attesa del primo figlio che termina\n");
23         pid = wait(&status); // attesa terminazione
24         printf("4) per primo termina il figlio con pid: %d\n", pid);
25         return 0;
26     }
27 }
28 }
```

Mandiamo in esecuzione più volte il programma e osserviamo che nella prima esecuzione il figlio che termina per primo è il primo che è stato generato mentre negli altri due casi è il secondo.

```

Paolo@PCwin8 ~/uai/c_fork_join
$ gcc forkwait2.c -o fw
Paolo@PCwin8 ~/uai/c_fork_join
$ ./fw
1) sono il primo processo figlio con pid: 7704 sleep: 3
3) padre in attesa del primo figlio che termina
2) sono il secondo processo figlio con pid: 7780 sleep: 1
4) per primo termina il figlio con pid: 7780

Paolo@PCwin8 ~/uai/c_fork_join
$ ./fw
1) sono il primo processo figlio con pid: 7792 sleep: 2
3) padre in attesa del primo figlio che termina
2) sono il secondo processo figlio con pid: 7788 sleep: 2
4) per primo termina il figlio con pid: 7792

Paolo@PCwin8 ~/uai/c_fork_join
$ ./fw
1) sono il primo processo figlio con pid: 6016 sleep: 1
3) padre in attesa del primo figlio che termina
2) sono il secondo processo figlio con pid: 6276 sleep: 1
4) per primo termina il figlio con pid: 6016
```

Funzione `waitpid()`

È possibile sospendere il processo padre in attesa di uno specifico processo utilizzando la funzione:

```
int waitpid(int pid, int *status, int options);
```

che ha tre parametri:

- 1 **pid**: serve a indicare quale processo si deve aspettare, e può essere:
 - < -1 attende della terminazione di un qualunque processo figlio il cui **PGID** (identifi-

cativo di ▶ process group ▶) è uguale al valore assoluto del parametro (per esempio, -5764 indica che il process group è 5764);

- ▶ -1: attende la terminazione di un qualunque processo figlio, in modo analogo alla wait();
 - ▶ 0: attende un qualunque processo figlio il cui PGID è uguale a quello del processo chiamante;
 - ▶ > 0: attende la terminazione del processo figlio con uno specifico valore di PID.
- 2 **status**: permette di memorizzare il valore di ritorno del processo che si sta attendendo;
- 3 **options**: è utilizzato per specificare opzioni avanzate che esulano dai nostri scopi (per noi vale sempre 0).



◀ Process group Appartengono allo stesso process group, e quindi hanno PGID uguale, tutti i processi discendenti dallo stesso antenato padre oppure i processi raggruppati esplicitamente con la funzione setpgrp(). ▶

È possibile quindi fare in modo che una processo si sospenda in attesa o di un generico figlio del quale indica il PID, oppure di un figlio appartenente a un gruppo da lui definito oppure di un qualunque figlio della sua dinastia.

Nel caso in cui il figlio sul quale viene fatta la wait() avesse terminato la sua esecuzione la funzione ritorna immediatamente il valore 0.

Nel caso si verifichi un errore il valore di ritorno è -1 mentre in tutti gli altri casi il valore di ritorno è il PID del processo atteso.

AREA digitale



Valore di OPTIONS in una waitpid()

Riscriviamo il programma precedente facendo in modo che il padre termini attendendo il processo più lungo, che realizziamo mediante appositi valori della funzione sleep().

```
forkwait4.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int pid, pid1, pid2, status, xj
5     srand(time(NULL));
6     pid1 = fork();
7     if(pid1 == 0){           // primo processo figlio
8         printf("1) sono il primo processo figlio con pid: %i \n", getpid());
9         sleep(4);
10        exit(1);           // termina primo processo figlio
11    }
12    else{
13        pid2 = fork();
14        if (pid2 == 0){      // secondo processo figlio
15            printf("2) sono il secondo processo figlio con pid: %i \n", getpid());
16            sleep(1);
17            exit(2);         // termina secondo processo figlio
18        }
19        else {
20            printf("3) padre in attesa del figlio piu' lento \n");
21            pid = waitpid(pid1, &status , 0); // attesa terminazione
22            printf("4) finalmente termina il figlio con pid: %d\n", pid);
23            return 0;
24        }
25    }
26 }
```

Ora qualunque esecuzione ci visualizza una situazione come quella sotto riportata dove il processo padre termina solo dopo la terminazione del processo più lento che è sicuramente il primo figlio dato che ha un ritardo provocato dalla istruzione 9 di `sleep(4)`.

```

Paolo@PCwin8 ~/ua1/c_fork_join
$ gcc forkwait4.c -o fw
Paolo@PCwin8 ~/ua1/c_fork_join
$ ./fw
1) sono il primo processo figlio con pid: 7688
2) sono il secondo processo figlio con pid: 5728
3) padre in attesa del figlio piu' lento
4) finalmente termina il figlio con pid: 7688
Paolo@PCwin8 ~/ua1/c_fork_join
$ |

```



Prova adesso!

- Istruzione `fork()`
- Istruzione `wait()` e `waitpid()`
- Istruzione `sleep()`

Scrivi un programma dove il processo figlio scrive una frase in un file e il padre ne attende la terminazione della scrittura e successivamente la legge dal file e la visualizza.

Confronta la tua soluzione con quella riportata nel file [forkwait4Sol.c](#)

Quindi modifica il programma aggiungendo tre figli che scrivano in sequenza una frase nel file: il padre visualizza il contenuto del file quando ha terminato il figlio più lento.

Scrivi infine un programma in cui un processo padre P crea due processi figli (prima F1 e poi F2) e ne attende la terminazione in ordine inverso.

Ciascuno dei due figli F1 e F2, a sua volta, crea due processi figli (il figlio F1 crea prima G1, poi H1 e il figlio F2 crea prima G2, poi H2) e ne attende la terminazione nell'ordine in cui li ha creati.

Confronta la tua soluzione con quella riportata nel file [forkwait4aSol.c](#)

■ Funzione predefinite per rilevare lo stato

Per interpretare correttamente il valore della variabile `status` è necessario conoscere la rappresentazione di `status`: come precedentemente detto lo standard [POSIX.1](#) utilizza 16 bit e mette a disposizione delle macro (definite nell'header file `<sys/wait.h>`) per l'analisi dello stato di terminazione.

In particolare

► `bool WIFEXITED(status)`: restituisce vero se il processo figlio è **terminato volontariamente**. In questo caso la macro:

`int WEXITSTATUS(status)`

restituisce lo stato di terminazione che passato con la funzione `exit(status)`;

► `bool WIFSIGNALED(status)`: restituisce vero se il processo figlio è **terminato involontariamente**.

riamente. In questo caso la macro:

int WTERMSIG(status)

restituisce il numero del segnale che ha causato la terminazione.

Tutti i programmi “dovrebbero” utilizzare queste funzioni per interpretare correttamente i risultati delle elaborazioni: riscriviamo il primo esempio utilizzando queste macro.

```
forkwait6.c
1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 int main(){
5     int pid, status;
6     pid = fork();
7     if (pid == 0){           // ramo eseguito dal solo processo figlio
8         printf("eseguito dal figlio \n");
9         exit(0);
10    }
11    else{                  // ramo eseguito dal solo processo padre
12        pid = wait(&status); // controllo stato del processo figlio
13        if(WIFEXITED(status))
14            printf("Term. volontaria di %d con stato %d\n", pid, WEXITSTATUS(status));
15        else
16            if(WIFSIGNALED(status))
17                printf("terminazione involontaria per segnale %d\n", WTERMSIG(status));
18    }
19 }
```

Vediamo un esempio più articolato dove creiamo più figli.

ESEMPIO

Scriviamo un programma dove un processo padre crea un numero N di figli e ciascun figlio scrive 10 volte una lettera dell’alfabeto, rispettivamente il primo figlio scrive le “a”, il secondo le “b”, e via di seguito.

Si richiede di visualizzare un output ordinato, dove le “a” precedono le “b”, le “c” ... ecc.

Naturalmente non ha senso risolvere questo esercizio con le **fork** in quanto dobbiamo rendere sequenziale processi paralleli e quindi, di fatto, rendere nullo il grado di parallelismo.

```
forkwait7.c
1 #include <sys/wait.h>
2 #include <stdio.h>
3 #include <fcntl.h>
4 #include <stdlib.h>
5
6 #define NRFIGLI 4
7 #define VOLTE 10
8
9 int main() {
10     int status, pid, x, y, z;
11     char c;
12     for (x = 0; x < NRFIGLI; x++) {
13         pid = fork();           // creazione del figlio
14         if (pid == 0) {          // nel figlio pid è uguale a 0
15             c = 'a' + x;
```

```

16    for (y = 0; y < VOLTE; y++){
17        printf ("%c",c); fflush(0);
18        sleep(1);
19    }
20    printf ("\n");
21    exit(0);
22 }
23 else // nel padre ha valore > 0
24     pid = wait(&status); // controllo stato del processo figlio
25     if(WIFEXITED(status))
26         printf("Term. volontaria di %d con stato %d\n", pid, WEXITSTATUS(status));
27     else
28         if(WIFSIGNALED(status))
29             printf("terminazione involontaria per segnale %d\n", WTERMSIG(status));
30 }
31 }
```



Prova adesso!

- Istruzione wait
- Macro di controllo dello status

Scrivi un programma che genera una **gerarchia di processi** ricevendo dalla linea di comando tre parametri

`<flag> <#_figli> <#_nipoti >`

Dove rispettivamente:

`<flag>`: permette di indicare il tipo di concorrenza:

- ▶ 0: figli in parallelo;
- ▶ 1: esecuzione figli in sequenza;

`<#_figli>`: numero di figli di primo livello nella gerarchia;

`<#_nipoti >`: numero di nipoti che ciascun figlio deve fare.

Esempi di parametri

- ▶ 0 1 2: parallelo , un figlio con due nipoti
- ▶ 1 2 2 4: seriale, due figli rispettivamente uno con 2 nipoti e uno con 4
- ▶ 1 3 1 2 3: seriale, tre figli rispettivamente uno con 1, 2 e 3 nipoti

```

E ~ /ua1/c_fork_join
Paolo@PCwin8 ~/ua1/c_fork_join
$ ./fj 1 2 1 2
PADRE: Creato figlio (pid=4060)
Figlio 4060: Creato nipote (pid=8928)
Nipote 8928, figlio di 4060
PADRE: terminazione volontaria del Figlio 8928 con stato 0
PADRE: terminazione volontaria del figlio 4060 con stato 0
PADRE: Creato figlio (pid=5628)
Figlio 5628: Creato nipote (pid=7384)
Nipote 7384, figlio di 5628
PADRE: terminazione volontaria del Figlio 7384 con stato 0
Figlio 5628: Creato nipote (pid=8832)
Nipote 8832, figlio di 5628
PADRE: terminazione volontaria del figlio 8832 con stato 0
PADRE: terminazione volontaria del figlio 5628 con stato 0
Paolo@PCwin8 ~/ua1/c_Fork_Join
$ |
```

Confronta la tua soluzione con quella riportata nel file `forkwait7Sol.c`

ESERCITAZIONI DI LABORATORIO 6

FORK-JOIN E COBEGIN-COEND



I codici sorgenti sono nel file **C_fork_join.rar** scaricabile dalla cartella materiali nella sezione del sito www.hoepiscuola.it riservata al presente volume.

■ Il costrutto fork-join

Utilizzando le funzioni **fork()** e **wait()** realizziamo il costrutto **fork-join** per l'esecuzione di calcoli paralleli dove il processo figlio passa il risultato al processo padre per generare il risultato finale.

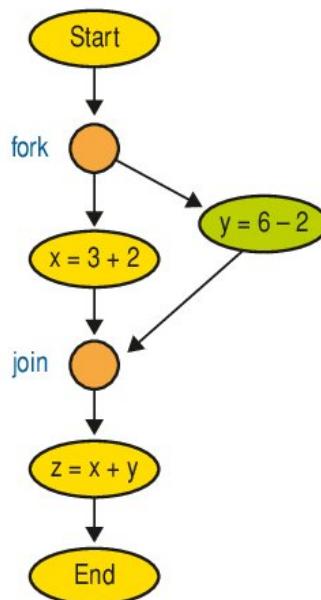
ESEMPIO

Risolviamo per esempio la seguente espressione matematica

$$z = (3+2) * (4-6)$$

scriviamo un programma che la esegue col massimo parallelismo.

Per prima cosa realizziamo il grafo delle precedenze parallelizzando le operazioni indicate tra parentesi:



Una pseudocodifica del programma è la seguente:

```
/* processo padre: */
start
    p2 = fork figlio1; // inizia l'elaborazione parallela
    x=3+2;
    join p2;           // termina l'elaborazione parallela
    z=x+y;
...
end.

/* codice nuovo processo:*/
int figlio1()
{
    y=6-2;
    return y
}
```

Separiamo il codice che effettua i calcoli matematici dal codice che controlla l'evoluzione dei processi: una elaborazione viene fatta dal processo figlio mentre due elaborazioni sono effettuate dal processo padre.

```
forkjoin1.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int calcoli_figlio1(){
5      int k;
6      printf("1.1 elab. parallela processo figlio \n");
7      k = 3 + 2;
8      return k;
9  }
10
11 int calcoli_padre1(){
12     int k;
13     printf("1 elab. parallela processo padre \n");
14     k = 6 - 2;
15     return k;
16 }
17
18 int calcoli_padre2(int a, int b){
19     int k;
20     printf("2 elab. finale padre \n");
21     k = (a + b);
22     return k;
23 }
```

Ricordiamo che è possibile leggere il valore di ritorno **ret_value** della funzione

```
int wait(*ret_value)
```

mediante la funzione

```
int WEXITSTATUS(ret_value)
```

e sfruttiamo questa particolarità per effettuare il “ritorno” del valore calcolato dalle funzioni al processo padre, come possiamo vedere nella istruzione 37, se tale valore è inferiore al numero 255.

```

24
25  int main( ){
26      int x, y, z , retv;
27      pid_t pid;
28      pid = fork();           // inizio elaborazione parallela
29      if (pid == 0){
30          x = calcoli_figlio(); // esecuzione parallela calcoli figlio
31          exit(x);            // termina processo figlio
32      }
33      else{
34          y = calcoli_padre(); // esecuzione parallela calcoli padre
35      }
36      printf(.. join: padre aspetta \n");
37      wait(&retv);           // join : il padre aspetta il figlio
38      x = WEXITSTATUS(retv); // prende il risultato del figlio
39      z = calcoli_padre2(x, y); // esegue gli ultimi calcoli
40      printf("-> risultato finale z = %d", z );
41  }

```

Mandiamo in esecuzione il programma e otteniamo il seguente output:

```

Paolo@PCwin8 ~/uai/c_fork_join
$ ./fj
1 elab. parallela processo padre
.. join: padre aspetta
1.1 elab. parallela processo figlio
2 elab. finale padre
-> risultato finale z = 9
Paolo@PCwin8 ~/uai/c_fork_join
$ |

```

Utilizzare il parametro di **status** per “comunicare” con il processo padre non è sicuramente una soluzione percorribile nel caso che si voglia trasferire il risultato di elaborazioni: anche solo un numero negativo manderebbe in difficoltà questa soluzione.

Gli esercizi presenti in questa lezione sono quindi di carattere esclusivamente teorico: vedremo nel prossima esercitazione come comunicare tra **processi** mediante aree di **memoria condivisa**.



Prova adesso!

- Costrutto fork-join

Dopo aver realizzato il diagramma delle precedenze, scrivi i programmi che implementando il costrutto **fork-join** eseguono le seguenti operazioni con il massimo grado di parallelismo.

$$5 * [(2 + 4) * (7 + 3)] - 10$$

$$(3 + 2) * (5 - 7) + (8 - 3)$$

$$(3 + x) - (5 - y) * (7 * y + 3) \quad // x viene letto nel padre mentre y e z nel main()$$

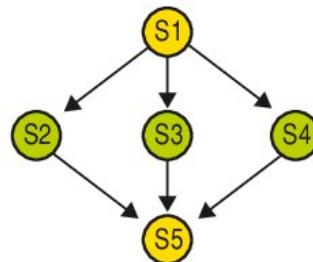
Confronta il tuo codice con quello presente nel file **forkjoin1Sol.c**.

■ Cobegin-coend

Mentre il costrutto **fork-join** permette di realizzare qualsiasi grafo di precedenza fra task, il costrutto **cobegin-coend** è più restrittivo e a volte non permette di descrivere tutte le situazioni di concorrenza.

Il linguaggio C non ha una istruzione specifica per questo costrutto, ma è semplicemente realizzabile mediante le istruzioni **fork()** e **wait()** appena descritte.

L'esempio di figura prevede l'esecuzione parallela di tre processi

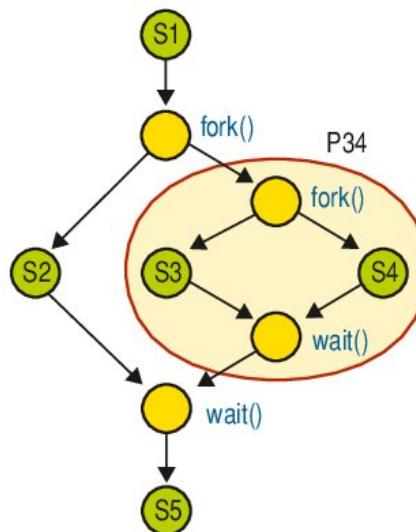


descritta con la seguente pseudocodifica

```

S1;
cobegin
  S2;
  S3;
  S4;
coend
S5;
  
```

Può essere vista come la combinazione di più fork-wait() come riportata nel seguente diagramma



La codifica in linguaggio C del primo segmento che esegue la prima **fork** è il seguente:

```
cobegin1.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main( ){
5     int status;
6     pid_t pid_figlio;
7     printf ("S1 (padre1) - pid = %d\n", getpid());
8     printf( " fork esterna - \n" );
9     if ((pid_figlio = fork()) == -1) // prima fork
10        printf( "fork non riuscita !" );
11    else
12        if (pid_figlio == 0){
13            printf(" figlio1: pid = %d, padre pid = %d\n", getpid(), getppid());
14            ramo34(); // esecuzione fork processi 3 e 4 (P34)
15        }
}
```

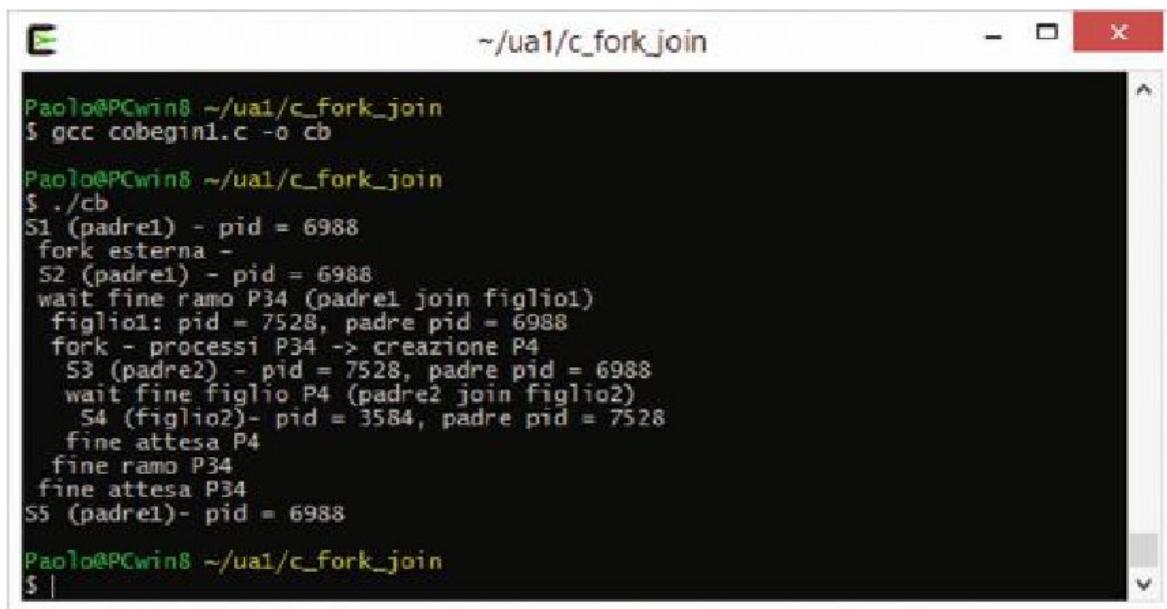
Il ramo di destra ora deve eseguire la seconda fork in modo che venga eseguito il segmento S3 in parallelo col segmento S4:

```
26 ramo34(){
27     int status;
28     pid_t pid4;
29     printf( " fork - processi P34 -> creazione P4 \n" );
30     if ((pid4 = fork() ) == -1 )
31        printf( " fork non riuscita !" );
32    if (pid4 == 0){
33        printf(" S4 (figlio2)- pid = %d, padre pid = %d\n", getpid(), getppid() );
34    }
35    else{
36        printf(" S3 (padre2) - pid = %d, padre pid = %d\n", getpid(), getppid() );
37        printf(" wait fine figlio P4 (padre2 join figlio2) \n" );
38        waitpid(pid4, &status , 0); // attesa fork interna
39        printf(" fine attesa P4 \n" );
40        printf(" fine ramo P34 \n" );
41    }
42    exit( 0 );
43 }
```

Contemporaneamente viene eseguito il ramo di sinistra, cioè il segmento S2, e alla fine il **main()** si mette in attesa della terminazione del ramo di destra, per poi poter eseguire l'ultima parte di codice, il segmento S5.

```
15 }
16 else{ // esecuzione ramo padre
17     printf(" S2 (padre1) - pid = %d\n",getpid());
18     printf(" wait fine ramo P34 (padre1 join figlio1) \n" );
19     waitpid(pid_figlio, &status , 0); // attesa prima fork
20     printf(" fine attesa P34 \n" );
21     printf("S5 (padre1)- pid = %d\n",getpid());
22     exit( 0 );
23 }
24 }
```

Abbiamo indentato manualmente l'output in modo da individuare sullo schermo le diverse istruzioni di **join** e **wait**:



```

Paolo@PCwin8 ~/ua1/c_fork_join
$ gcc cobegin1.c -o cb

Paolo@PCwin8 ~/ua1/c_fork_join
$ ./cb
S1 (padre1) - pid = 6988
fork esterna -
S2 (padre1) - pid = 6988
wait fine ramo P34 (padre1 join figlio1)
figlio1: pid = 7528, padre pid = 6988
fork - processi P34 -> creazione P4
S3 (padre2) - pid = 7528, padre pid = 6988
wait fine figlio P4 (padre2 join figlio2)
S4 (figlio2)- pid = 3584, padre pid = 7528
fine attesa P4
fine ramo P34
fine attesa P34
S5 (padre1)- pid = 6988

Paolo@PCwin8 ~/ua1/c_fork_join
$ |

```

Non è agevole scambiare informazioni tra padre e figlio in quanto i processi non condividono variabili: al momento della **fork** tutte le variabili del padre vengono duplicate nel figlio e se effettuiamo il passaggio dei parametri per indirizzo spesso ci troviamo di fronte a situazioni complesse.

Nella prossima lezione vedremo un meccanismo più semplice ed efficace per far comunicare tra loro i processi.



Prova adesso!

- Costrutto cobegin-coend

Dopo aver realizzato il diagramma delle precedenze, scrivi i programmi che implementando il costrutto **cobegin-coend** eseguono le seguenti operazioni:

$$5 * [(2 + 4) * (7 + 3)] - 10$$

$$(3 + 2) * (5 - 2) * (8 - 3)$$

$$(2 + x) * (3 + y) * (7 * z + 4) - 10 \quad // x, y e z vengono letti da input nel main()$$

Confronta il tuo codice con quello presente nel file **cobegin1Sol.c**



ESERCITAZIONI DI LABORATORIO 7

I THREAD IN C



Info

I codici sorgenti sono nel file **C_pthread.rar** scaricabile dalla cartella **materiali** nella sezione del sito www.hoepiscuola.it riservata al presente volume.

■ L'implementazione dei thread in LINUX

Per utilizzare i **pthread** in un programma C è necessario includere la libreria:

`<pthread.h>`

In questo modo potremmo utilizzare i **thread** definiti dallo standard **POSIX, Portable Operating System Interface for Computing Environments**, che prendono proprio il nome di “**pthread**” e sono stati definiti con l’obiettivo di avere uno standard di riferimento che permetta la portabilità dei programmi in ambienti diversi che utilizzano le interfacce applicative (**API**) dei sistemi operativi.

Linux non è completamente **POSIX** compatibile per quanto riguarda la gestione dei segnali, e dalla versione **RedHat** si è sviluppato il supporto nativo ai thread per **Linux (NPTL: Native POSIX Thread Linux)** che oltre ad avere vantaggi in termini di prestazioni è maggiormente aderente allo standard **POSIX**.

■ Creazione di thread: **pthread_create**

La chiamata per creare un **thread** è definita da questo prototipo:

```
int pthread_create(pthread_t *ptID, pthread_attribute att, void *routine, void *arg)
```

Dove:

- **ptID**: è il puntatore alla variabile che contiene l’identificativo del **thread** che viene creato, il **thread_ID (tID)**;
- **att**: è il puntatore all’eventuale vettore (o alla variabile) contenente i parametri (od il parametro) da passare al **thread** creato (per esempio la priorità); noi utilizzeremo sempre gli attributi di default e quindi indicheremo nella chiamata **NULL**;

- **routine**: è il puntatore alla funzione che contiene il codice che il **thread** deve eseguire; tale funzione deve avere il seguente prototipo:

```
void *codice_thread (void *argomento);
```

- **arg**: è il puntatore all'eventuale vettore (o alla variabile) contenente i parametri (od il parametro) da passare al **thread** creato e può essere posto a **NULL** nel caso di assenza di parametri.

void * permette di passare alla funzione argomenti di natura e struttura diversi: nella funzione eseguita dal **thread** è possibile, tramite **cast**, tradurre il puntatore generico in un puntatore ad una struttura dati che contiene tutti i parametri necessari per quella specifica applicazione.

La funzione restituisce 0 in caso che vada a buon fine la creazione di un nuovo **thread** e il **TID** del **thread** appena creato è salvato all'interno di **pthread_t**, altrimenti ritorna un codice di errore diverso da zero secondo le convenzioni di **<sys/errno.h>**.

Come sappiamo il **thread** condivide con il **processo invocante** le seguenti aree:

- codice;
- dati;
- mappe di memoria.

In seguito vedremo come utilizzare questa condivisione per scambiare dati tra processo e **thread** e tra i singoli **thread**.

Vediamo un primo esempio dove effettuiamo la creazione di un **thread** con **pthread_create()**:

```
creazione.c
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void *codice_thread(void * arg) { // codice che esegue il thread
6     pid_t pid;
7     pid = getpid(); // ritorna l'identificatore del processo PID
8     pthread_t tid;
9     tid = pthread_self(); // ritorna identificato del thread (il valore di ptid)
10    printf("Sono il thread %i del processo %i\n", tid, pid);
11 }
12
13 int main (){
14     pthread_t ptid; // identificatore thread alla creazione
15     printf ("il pid del main e' %d\n", (int) getpid ());
16     pthread_create (&ptid, NULL, &codice_thread, NULL);
17     return (void*)0; // return 0;
18 }
```

Compiliamo e mandiamo in esecuzione il programma, che visualizza una situazione simile alla seguente:

```
Paolo@PCwin8 ~/ua1/c_pthread
$ gcc creazione.c -o crea
Paolo@PCwin8 ~/ua1/c_pthread
$ ./crea
il pid del main e' 6672
Paolo@PCwin8 ~/ua1/c_pthread
$
```

Non visualizzando l'output della riga 10 possiamo ipotizzare che il codice del **thread** non venga eseguito: molto probabilmente il processo padre termina prima che avvenga la schedulazione del **thread** e quindi la sua terminazione comporta la terminazione anche del figlio "prima che nasca".

Per essere sicuri che entrambi vengano eseguiti è necessario introdurre un ritardo nel padre in modo da "permettere" la nascita del figlio stesso: modifichiamo il codice, come di seguito riportato, inserendo un secondo di ritardo nel processo padre.

```

13 int main (){
14     pthread_t miothread;
15     printf ("il pid del main e' %d\n", (int) getpid ());
16     if (pthread_create (&miothread, NULL, &codice_thread, NULL)==0) // tutto ok
17         sleep(1);           // ritardo per garantire l'esecuzione del thread
18     else
19         printf ("errore nella creazione del thread!" );
20     return 0;
21 }
```

In esecuzione ora otteniamo il seguente output:

```

Paolo@PCwin8 ~/ua1/c_pthread
$ ./crea
il pid del main e' 6672
Sono il thread 296080 del processo 6672
Paolo@PCwin8 ~/ua1/c_pthread
$
```

dove possiamo riconoscere oltre al processo padre anche l'esecuzione del **thread** figlio.

Il **PID** visualizzato è sempre quello del processo padre, dato che il **thread** vive all'interno del padre.

È sempre meglio verificare il buon fine della creazione del **thread** introducendo il controllo sul parametro di ritorno della creazione del **thread**, come fatto con l'istruzione 16 del listato precedente.



Prova adesso!

- Creazione di thread

Scrivi un programma che crea tre thread che, rispettivamente, visualizzano sullo schermo ciascuno il suono di una campana: DIN , DON oppure DAN.

Manda più volte in esecuzione il programma osservando l'output dopo aver introdotto un ritardo casuale in ciascun **thread**.

Confronta la tua soluzione con quella riportata nel file **campaneSol.c**

Terminazione di thread: pthread_exit

Esistono due modalità per terminare l'esecuzione di un **thread**:

- A** quella tradizionale che avviene alla fine dell'esecuzione del codice della funzione, cioè la classica

```
return();
```

- B** utilizzando la chiamata alla funzione

```
void pthread_exit(void *retval);
```

con **retval** che è il puntatore alla variabile che contiene il valore di ritorno e, come vedremo, potrà essere letto dal processo padre con la funzione **join**.

È anche possibile terminare il **thread** utilizzando la classica funzione **exit()**.

Si consiglia di utilizzare sempre la chiamata di **pthread_exit()** oppure **return()** evitando l'uso di **exit()** per la terminazione regolare di un **processo/thread** riservando questa per i casi di uscita forzata e immediata (errore irrecuperabile).

Il valore di ritorno **retval** della funzione **pthread_exit()** è anch'esso di tipo **void *** in modo da permettere di passare alla funzione chiamante argomenti di natura e struttura diversi: ritornando un puntatore generico il processo chiamante effettuerà tramite cast la conversione in un puntatore a una struttura dati specifica per quella applicazione nella quale il thread ha inserito i risultati che vuole passare al chiamante.

Abbiamo due possibili strade per "produrre" il parametro di ritorno, a seconda della dimensione del dato:

- **si può** convertire direttamente il dato presente in **void *** se il valore da restituire è un intero oppure è un tipo più piccolo della dimensione dell'intero;
- **si deve** convertire il puntatore al dato **void *** se il valore da restituire ha una dimensione maggiore di quella di un intero.

È da preferirsi quindi la soluzione che prevede sempre di effettuare la conversione tra puntatori in quanto è più elegante e più portabile: inoltre non dipende in alcun modo dalla dimensione dei dati dei vari tipi in caso di esecuzione su piattaforme hardware diverse.

Se dobbiamo restituire un valore costante intero possiamo utilizzare la seguente notazione:

```
11 |
12 |     pthread_exit( (int*) 0 ); // ritorna un numero intero
13 | }
```

dove ritorniamo il numero 0 al programma che lo ha generato: vedremo in seguito come leggere tale parametro utilizzando la funzione **join**.

Dal momento che la funzione **void pthread_exit()** non ha nessun parametro di ritorno, il codice successivo alla chiamata di questa funzione è "codice morto" che non verrà mai eseguito.

■ Cancellazione di un thread

È possibile cancellare un **thread** prima che termini la sua esecuzione mediante il comando:

```
int pthread_cancel(pthread_t tid);
```

richiede come parametro in ingresso il **tid** del **thread** che deve essere terminato e restituisce come parametro in uscita:

- 0 se OK;
- un codice d'errore se l'istruzione non va a buon fine.

■ Attesa di thread: **pthread_join**

La modalità corretta per attendere la terminazione di un **thread** (invece che introdurre una istruzione di **sleep(tempo)** con un valore del tempo impostato “a caso”) è quella di utilizzare la funzione

```
int pthread_join(pthread_t mioThread, void **par Ritorno);
```

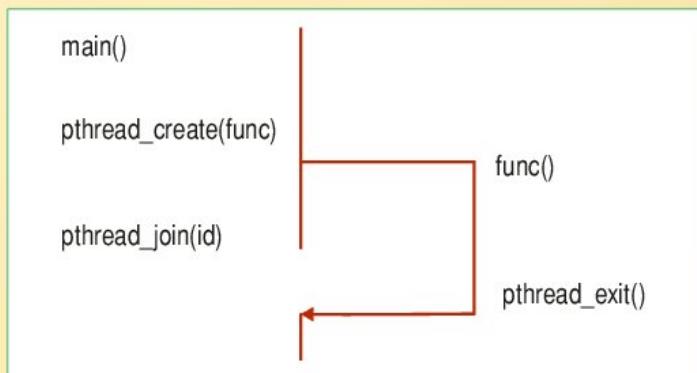
Dove:

- **mioThread**: è il **pid** del **thread** del quale si vuole attendere la terminazione;
- **par_Ritorno**: in questa variabile viene memorizzato il valore di ritorno del **thread** che il **thread** terminato ha passato a **pthread_exit(parametro)**: può quindi avere valore **NULL** in caso di assenza di parametro di ritorno.

È possibile mettere un unico processo/thread in attesa della terminazione di un altro thread.

Come valore di ritorno **int** la funzione restituisce 0 in caso di successo, altrimenti un codice di errore diverso da 0, secondo le convenzioni di **<sys/errno.h>**.

La funzione **pthread_join()** è equivalente alla **waitpid()** dei processi fatta con la **fork()**; ha un comportamento come quello riportato in figura:



Riscriviamo ora il codice dell'esempio **campaneSol.c** facendo in modo di ottenere la sequenza ordinata del suono delle campane utilizzando la funzione **pthread_join()**.

Il codice dei tre figli è per esempio il seguente:

```
campaneloin.c
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 void* thread1 (void* arg){
6     sleep(rand() % 1);
7     printf ("DIN ");
8     fflush(stdout);
9 }
10 void* thread2 (void* arg){
11     sleep(rand() % 2);
12     printf ("DON ");
13     fflush(stdout);
14 }
15 void* thread3 (void* arg){
16     sleep(rand() % 3);
17     printf ("DAN ");
18     fflush(stdout);
19 }
```

Il codice del padre è molto semplice: crea i tre figli e si mette in attesa della loro terminazione.

```
18 int main (){
19     srand(time(NULL));
20     pthread_t miothread1, miothread2, miothread3 ;
21     printf ("il pid del main e' %d\n", (int) getpid ());
22     pthread_create (&miothread1, NULL, &thread1, NULL);
23     pthread_create (&miothread2, NULL, &thread2, NULL);
24     pthread_create (&miothread3, NULL, &thread3, NULL);
25     pthread_join (miothread1, NULL);
26     pthread_join (miothread2, NULL);
27     pthread_join (miothread3, NULL);
28     return 0;
29 }
```

Mandiamo in esecuzione più volte il programma e otteniamo risultati diversi a ogni sua istanza.

```
Paolo@PCwin8 ~/u1/c_pthread
$ ./pt
il pid del main e' 2112
DAN DON DIN
Paolo@PCwin8 ~/u1/c_pthread
$ ./pt
il pid del main e' 6980
DON DIN DAN
Paolo@PCwin8 ~/u1/c_pthread
$ ./pt
il pid del main e' 7340
DON DAN DIN
Paolo@PCwin8 ~/u1/c_pthread
$ |
```

Per ottenere un risultato deterministico è necessario introdurre delle rettifiche che lasciamo come esercizio per il lettore.

Riportiamo in una tabella riepilogativa le similitudini esistenti tra processi e **thread**.

Processi	Thread
fork	pthread_create
exit	pthread_exit
waitpit	pthread_join
kill	pthread_kill
getpid	pthread_self



► Quando compiliamo un programma che utilizzi i thread in ambiente Linux nativo, dobbiamo necessariamente richiamare la libreria **pthread** all'interno della linea di comando come riportato di seguito: ►

```
C_pthread : bash - Konsole
File Edit View Bookmarks Settings Help
utente@AspireXC705-Linux: ~ > cd sviluppo
utente@AspireXC705-Linux: ~/sviluppo > cd ual
utente@AspireXC705-Linux: ~/sviluppo/ual > cd C_pthread
utente@AspireXC705-Linux: ~/sviluppo/ual/C_pthread > gcc -pthread creazione.c -o crea
utente@AspireXC705-Linux: ~/sviluppo/ual/C_pthread > ./crea
il pid del main e' 3758
Sono il thread 429135616 del processo 3758
C_pthread : bash
```



Prova adesso!

- Costrutto fork-join
- Costrutto cobegin-coend

Realizza mediante i thread gli stessi programmi che implementavano il costrutto fork-join e cobegin-coend fatti per i processi.

In particolare:

- A utilizzando il costrutto **fork-join** realizzare le seguenti operazioni con il massimo grado di parallelismo:

$$5 * [(2 + 4) * (7 + 3)] - 10 \\ (3 + x) - (2 + y) * (7 * y + 3) \quad // x e y vengono letti da input nel main()$$

- B utilizzando il costrutto **cobegin-coend** realizzare le seguenti operazioni con il massimo grado di parallelismo:

$$(3 + 2) * (5 - 2) * (8 - 3) \\ (2 + x) * (3 + x) * (7 * y + 4) - 10 \quad // x e y vengono letti da input nel main()$$

ESERCITAZIONI DI LABORATORIO 8

THREAD E PARAMETRI

■ Passare parametri a un thread

Abbiamo due possibili alternative per scambiare dati tra il programma principale e i **thread** che da esso vengono creati:

- Ⓐ passaggio diretto
- Ⓑ passaggio indiretto in memoria condivisa

Col **passaggio diretto** vengono sfruttate le possibilità offerte rispettivamente dalla funzione:

```
int pthread_create(pthread_t *ptID, pthread_attribute att, void *routine, void *arg)
```

utilizzando l'argomento **void *arg** per passare un valore dal “creatore” al **thread**, e dalla funzione:

```
int pthread_exit(void *status)
```

per ritornare un valore dal **thread** al “creatore” utilizzando il parametro **void *status**.

Ricordiamo che l'argomento e il valore di ritorno sono puntatori generici, quindi di tipo **void**, per offrire la massima flessibilità di scambio variabili.

In tal modo nella funzione eseguita dal **thread** è possibile:

- ▷ trasformare in ingresso tramite **casting** il puntatore generico in un puntatore a una specifica struttura dati (che contiene tutti i parametri richiesti dalla specifica applicazione);
- ▷ ritornare un puntatore generico il quale, sempre tramite **casting**, sarà convertito in un puntatore a una struttura dati che contiene tutti i risultati che il **thread** vuole trasmettere al processo chiamante.

■ Parametro passato da principale a thread

Come primo esempio passiamo al momento della creazione del **thread** un numero intero come ultimo parametro mediante la variabile **ingresso**:

```
38 int main() {
39     pthread_t tid;
40     int ingresso; // utilizzata come variabile di ingresso condivisa
41     void *ritorno; // utilizzata come variabile di ritorno condivisa
42     ingresso = 1;
43     pthread_create(&tid, NULL, thread, (void*) &ingresso); // &ingresso al thread
```

La corretta istruzione per leggere il valore del parametro è la seguente:

```
7 void *thread(void *arg) {
8     int dato, *punta_dato;
9     // 1) lettura diretta del valore passato come parametro
10    dato = *(int*) arg;
11 }
```

che può essere fatta anche in due istruzioni convertendo dapprima il puntatore e successivamente dereferenziandolo:

```
12 // 2) lettura in due passaggi
13 punta_dato = (int*) arg;           // cast da void* ad int*
14 dato = *punta_dato;              // dereferenziazione e lettura valore
15 }
```

Con la stessa procedura possiamo leggere un parametro di qualsiasi formato: nel seguente esempio leggiamo una stringa.

```
5 void *thread1(void *arg) {
6     char *nome;
7     nome = (char*) arg;           // scrivo nell'argomento arg
8 }
```

che viene passata mediante la seguente istruzione di creazione:

```
18 char *ingresso = "Thread 1";
19 pthread_create(&tid, NULL, thread1, (void*) ingresso); // &ingresso al thread
20 }
```



Prova adesso!

- Passaggi parametri a thread

Scrivi un programma che crea un **thread** per ciascuno dei sette nani e ne visualizza il nome sullo schermo utilizzando il seguente segmento di codice:

```
31 nome[0] = "Cucciolo";
32 nome[1] = "Pisolo";
33 nome[2] = "Eolo";
34 nome[3] = "Dotto";
35 nome[4] = "Mammolo";
36 nome[5] = "Gongolo";
37 nome[6] = "Brontolo";
38 for (t = 0; t < NUM_THREADS; t++) {
39     printf("Creazione thread %d\n", t);
40     rc = pthread_create(&threads[t], NULL, codice_thread, (void *) &t);
41 }
```

Controlla la tua soluzione con quella presente nel file **naniSol.c**

Quindi modifica il programma **campane.c** passando ai singoli **thread** il nome della campana che deve suonare.

Controlla la tua soluzione con quella presente nel file **campaneParametroSol.c**.

■ Parametro di ritorno da thread a chiamante

Passare dati da un **thread** al chiamante richiede attenzione in quanto è necessario

- prenderne l'indirizzo con l'operatore **&**;
- convertire l'indirizzo in **void*** col casting (**(void*)**);
- restituire tale valore passandolo con **pthread_exit()** o con l'istruzione **return()**.

ESEMPIO

Scriviamo come esempio un programma dove il **thread** effettua il lancio di un dado e ritor-
na il valore generato mediante la funzione **pthread_exit()**.

```
dadoGlobale.c
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <pthread.h>
4
5 int parametroOUT;
6 void* codice(void *arg){ // thread che genera un dado
7     srand(time(NULL));
8     parametroOUT = (rand() % 6) +1;
9     pthread_exit( (void*) &parametroOUT );
10 }
11
12 int main () {
13     int dadoEstratto, *risultato;
14     pthread_t t1;
15     pthread_create(&t1, NULL, codice, NULL);
16     pthread_join(t1, (void*) &risultato);
17     printf("dato estratto: %d\n", *risultato);
18     return 0;
19 }
```

Alcune esecuzioni danno il seguente risultato:

```
Paolo@PCwin8 ~/u1/c_pthread
$ gcc dadoGlobale.c -o dg
Paolo@PCwin8 ~/u1/c_pthread
$ ./dg
dato estratto: 2
Paolo@PCwin8 ~/u1/c_pthread
$ ./dg
dato estratto: 5
Paolo@PCwin8 ~/u1/c_pthread
$ ./dg
dato estratto: 4
```

Il dato da restituire non deve essere contenuto in una variabile automatica della funzione in esecuzione nel **thread** dato che le variabili automatiche risiedono sullo stack del **thread** che al termine della sua esecuzione viene rilasciato e quindi l'indirizzo restituito punta a una zona di memoria non più esistente.



Prova adesso!

- Utilizzo funzione pthread_exit()
- Utilizzo funzione pthread_join()

Sposta l'istruzione 5 di dichiarazione della variabile globale all'interno del codice del **thread**, come riportato in figura.

```

4
5 void* codice(void *arg){           // thread che genera un dado
6     int parametroOUT;             // dichiarata come locale (errore)
7 }
```

Manda in esecuzione il programma e osserva i risultati: come puoi giustificarli?

Come puoi modificare ulteriormente il codice per ottenere "almeno" un valore di ritorno al programma chiamante?

Confronta la tua soluzione con quelle riportata nel file **dadoErratoSol.c**

■ Passaggio dati mediante condivisione di memoria

Abbiamo visto come sia abbastanza complesso ricevere da un **thread** i risultati della sua elaborazione e nel caso ci fossero più valori di tipo diverso si rende necessaria la definizione di una struttura di dati: inoltre abbiamo visto che affinché si possa riportare l'indirizzo la struttura deve essere definita al di fuori del **thread**.

Nel programma precedente il **thread codice** passa come parametro di ritorno al **main** il puntatore alla variabile che però deve essere dichiarata nel **main** altrimenti non esiste al termine della esecuzione del **thread**: in altre parole passiamo al **main** il puntatore di una variabile che è definita nel **main**!

A questo punto possiamo semplificare queste operazioni e sfruttare direttamente lo stesso spazio di indirizzamento comune che i **thread** condividono senza passare alcuna variabile, né in ingresso e né in uscita, cioè tutti i **thread** vedono le stesse **variabili globali** e se uno di essi ne modifica una tale modifica è vista anche dall'altro **thread**.

In tutti i nostri esempi l'accesso alla **variabile globale** non viene regolamentato da alcun meccanismo di gestione della concorrenza: questo potrebbe provocare anomalie e malfunzionamenti, come discusso nella prossima **unità didattica**.

Scriviamo un esempio riepilogativo dove una **variabile globale** viene modificata da un **thread** in maniera causale elaborando un dato letto come parametro in ingresso e ritornando 0 oppure 1 in base al valore di questa variabile.

```
parametri.c
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <pthread.h>
4 int globale = 30;
5
6 void *thread(void *arg) {
7     int dato;
8     // elaborazione parametro in ingresso
9     dato = *(int*) arg + 20;           // scrivo nell'argomento arg
10
11    // introduco casualità del risultato
12    srand(time(NULL));
13    globale = globale * (rand() % 7) + dato;
14
15    // alternativa di terminazione condizionata
16    if (globale < 100)
17        pthread_exit( (int*) 0);      // ritorno numero 0
18    else
19        pthread_exit( (int*) 1);      // ritorno numero 1
20 }
21
```

Il `main()` passa il parametro **ingresso** alla creazione del **thread** e aspetta la sua terminazione leggendo il parametro **ritorno**: quindi visualizza i loro valori.

```
** 22 int main() {
23     pthread_t tid;
24     int ingresso;    // utilizzata come variabile di ingresso condivisa
25     void *ritorno;   // utilizzata come variabile di ritorno condivisa
26     ingresso = 10;
27     pthread_create(&tid, NULL, thread, (void*) &ingresso); // &ingresso al thread
28     if ( pthread_join(tid, &ritorno) == 0 ) {                // legge valore ritorno
29         printf(" valore di globale : %d\n", globale);
30         printf(" valore di ingresso: %d\n", ingresso);
31         printf(" uscita di ritorno : %d\n", ritorno);
32         exit(0);
33     }
34     else
35         printf ("errore join del thread! " );
36     exit(-1);
37 }
```

Due possibili esecuzioni danno il seguente risultato:

```
Paolo@PCwin8 ~/uai/c_pthread ~
Paolo@PCwin8 ~/uai/c_pthread $ ./parametri
valore di globale : 60
valore di ingresso: 10
uscita di ritorno : 0

Paolo@PCwin8 ~/uai/c_pthread $ ./parametri
valore di globale : 120
valore di ingresso: 10
uscita di ritorno : 1

Paolo@PCwin8 ~/uai/c_pthread $
```



Prova adesso!

- Condivisione di variabili globali

Scrivi un programma che calcola la somma dei primi 2^{N-1} numeri interi con N letto da input creando N **thread**.

Confronta la tua soluzione con quella presente nel file **potenzaSol.c**

Scrivi un programma che riceve in ingresso un numero N intero da linea di comando, genera N **thread** e ne aspetta la terminazione: ogni **thread** genera un numero casuale X, attende X secondi e quindi lo incrementa a una variabile globale **totale**.

Al termine della elaborazione viene visualizzato il valore di questa variabile.

Confronta la tua soluzione con quella presente nel file **addizioniSol.c**

■ Errato utilizzo delle variabili globali

Scriviamo ora un semplice programma che crea sette **thread** dove ciascuno di essi visualizza sullo schermo il nome di uno dei sette nani (Cucciolo, Pisolo, Eolo, Dotto, Mammolo, Gongolo, Brontolo) definendo **globale** il vettore con i nomi dei nani e passando a ciascun **thread** il rispettivo identificatore **tid**.

Invece di generare sette valori distinti di **tid**, si usa una variabile comune e la si condivide fra **thread** passandola come parametro: tale parametro viene dereferenziato a una variabile intera usata dal **thread** per leggere e visualizzare il proprio nome da una array globale **nome[]**.

```
nani.c
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>           // per utilizzare costanti predefinite
5
6 #define NUM_THREADS 7          // nr dei thread da far partire
7 char *nome[NUM_THREADS];      // vettore contenente i nomi dei nani
8
9 void *cod_thread(void *tid){
10     int tid_int;
11     tid_int = *(int*) tid;    // leggo parametro da puntatore void
12     // generazione ritardo casuale
13     srand(time(NULL));
14     sleep(rand() % 2);
15     printf(" Ciao da %d: %s \n", tid_int, nome[tid_int] ); // stampa nome
16     pthread_exit(NULL);       // uscita del thread senza valori di ritorno
17 }
```

Il main() riempie con i nomi dei nani il vettore **nome[]**, quindi crea i sette **thread** passando a ciascuno il rispettivo **pid**.

```

19 int main(int argc, char *argv[]){
20     pthread_t threads[NUM_THREADS]; // vettore identificatori dei thread
21     int tid, rc;
22     nome[0] = "Cucciolo";
23     nome[1] = "Pisolo";
24     nome[2] = "Eolo";
25     nome[3] = "Dotto";
26     nome[4] = "Mammolo";
27     nome[5] = "Gongolo";
28     nome[6] = "Brontolo";
29
30     for (tid = 0; tid < NUM_THREADS; tid++) {
31         printf("Creazione thread %d\n", tid);
32         rc = pthread_create(&threads[tid], NULL, cod_thread, (void *) &tid);
33         // sleep(1); // necessario: la velocità della macchina crea problemi sul valore di t
34         if (rc) {
35             printf ("ERRORE: il codice di errore di ritorno da pthread_create() e': %d\n", rc);
36             exit(EXIT_FAILURE);
37         }
38     }
39     pthread_exit(NULL); // terminazione corretta del programma
40 }
41

```

Poiché `pthread_create()` è asincrona, il processo principale fa in tempo a impostare `tid = 7`, che è la condizione di uscita della istruzione 30, prima che siano partiti tutti i `thread` e quindi come risultato avremo che alcuni `thread` (uno sicuramente!) stampa il contenuto della cella `nome[7]` che, nella migliore delle ipotesi, è `NULL`.

```

Paolo@PCwin8 ~/ua1/c_pthread
$ ./nani
Creazione thread 0
Creazione thread 1
  Ciao da 1: Pisolo
Creazione thread 2
  Ciao da 2: Eolo
Creazione thread 3
  Ciao da 3: Dotto
Creazione thread 4
  Ciao da 4: Mammolo
Creazione thread 5
  Ciao da 5: Gongolo
Creazione thread 6
  Ciao da 6: Brontolo
  Ciao da 7: (null)
Paolo@PCwin8 ~/ua1/c_pthread
$ 

```

Per eliminare il malfunzionamento è necessario introdurre un ritardo nel programma principale (istruzione 33) in modo da permettere al figlio di leggere correttamente il proprio nome.



Prova adesso!

- Condivisione di variabili globali

Scrivi un programma che crea otto **thread** e, rispettivamente, ciascuno visualizza sullo schermo il saluto in una lingua diversa senza usare variabili globali come contatori ma definendo globale il vettore con i messaggi di saluto.

```

32  /* I messaggi nelle otto lingue */
33  messaggio[0] = "English: Hello World!";
34  messaggio[1] = "French: Bonjour, le monde!";
35  messaggio[2] = "Spanish: Hola al mundo";
36  messaggio[3] = "Italiano: Ciao mondo!";
37  messaggio[4] = "German: Guten Tag, Welt!";
38  messaggio[5] = "Russian: Zdravstvyyte, mir!";
39  messaggio[6] = "Japan: Sekai e konnichiwa!";
40  messaggio[7] = "Latin: Orbis, te saluto!";
..
```

Manda più volte in esecuzione il programma osservando l'output dopo aver introdotto un ritardo casuale in ciascun **thread** e modifica il programma per renderlo immune da errori. Confronta la tua soluzione con quella riportata nel file **multilinguaSol.c**

AREA digitale



Un ulteriore esercizio errato
sulla memoria condivisa

AREA digitale



Esercizi per il recupero / Esercizi per il rinforzo

ESERCITAZIONI DI LABORATORIO 9

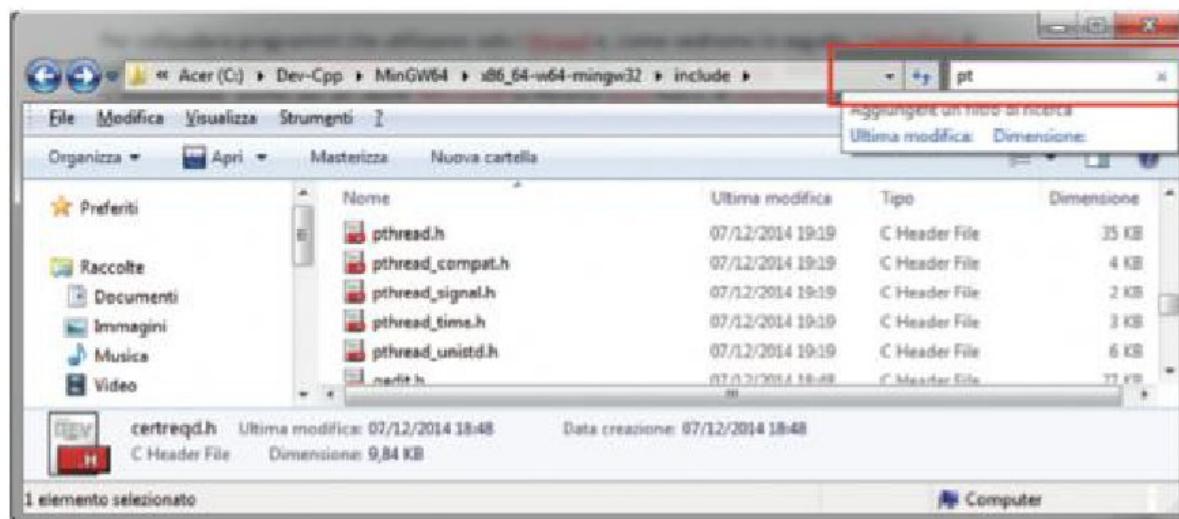
THREAD IN AMBIENTE DEV-CPP E LINUX

■ La libreria pThread.h di Dev-cpp

Nelle lezioni precedenti abbiamo eseguito i programmi con **Cygnus**, cioè in emulazione **linux**, anche perché le **fork/join** sono specifiche del sistema operativo ***nix**.

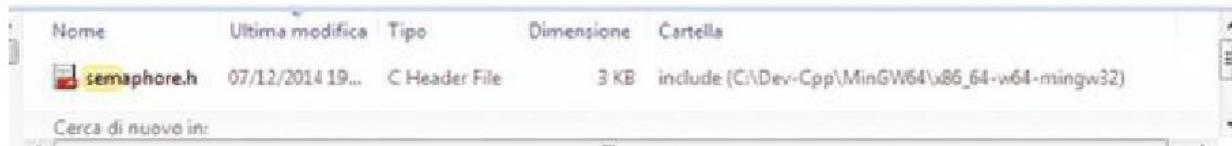
Per collaudare programmi che utilizzano solo i **thread** e, come vedremo in seguito, i **semafori**, è sufficiente **Windows** in quanto l'ambiente di sviluppo **Dev-cpp** dalla **versione 4.9** mette a disposizione anche per gli utenti **Microsoft** sia la libreria **posix** tipica di **unix/linux** che altre librerie utili per realizzare la **programmazione multithread e concorrente**.

Per verificare la presenza di queste librerie è sufficiente posizionarsi nella directory **include** nel percorso indicato e ricercarle, come nella seguente figura, digitando come iniziale **"pt"**:



Se non fossero presenti nel tuo PC sono scaricabili dalla cartella **software** nella sezione del sito www.hoepliscuola.it riservata al presente volume.

Sono anche disponibili le librerie per realizzare i programmi presenti nelle successive unità didattiche del presente volume; in particolare è disponibile la libreria **semaphore.h**



che ci permetterà di scrivere i programmi che implementano i meccanismi di sincronizzazione.

Ricordiamo però che in ambiente **windows** non è possibile effettuare le **fork** e quindi chi volesse riscrivere gli esercizi della unità di apprendimento 1 deve utilizzare la libreria **<windows.h>** e le funzioni **create_thread()**.

ESEMPIO

Vediamo un esempio scritto ed eseguito direttamente in **Dev-cpp**, dove un processo scrive un numero in una variabile a intervalli di un secondo e un processo la visualizza sullo schermo.

Nel codice, dopo aver incluso le solite librerie, definiamo la costante **TANTI** che useremo per indicare parametricamente il numero di ripetizioni desiderate e la variabile **globale** che verrà modificata all'interno del **main** e visualizzata nel **thread**.

```
contatore1.c
1 #include <pthread.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #define TANTI 10
6 int globale;
7
```

Il codice del **thread** si limita a una sola operazione, cioè quella di visualizzare con intervallo di un secondo per **TANTI** volte il contenuto della variabile **globale**, condivisa con il **main()**:

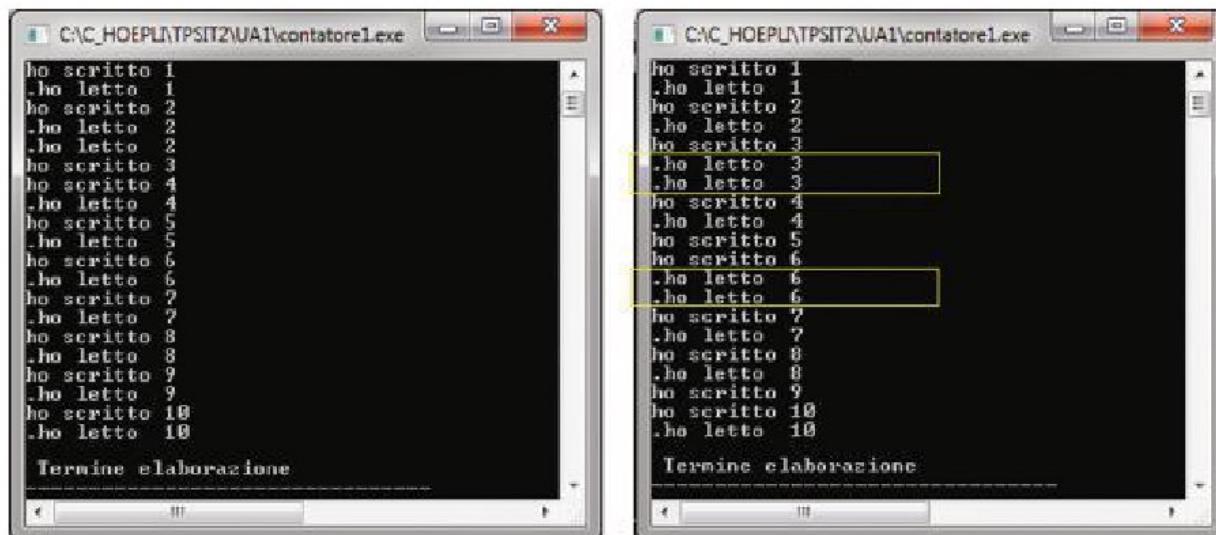
```
8 void *cosaFaThread(void *arg) {
9     int i, locale;
10    for (i = 0; i < TANTI; i++) {
11        printf("\n ho letto %d", globale); // visualizza contenuto var. condivisa
12        sleep(1);
13    }
14    return NULL;
15 }
```

Il `main()`, dopo aver definito il `thread`, lo crea con la funzione di riga 20 passandogli come parametro il codice che deve eseguire, inizia a incrementare la variabile `globale`, effettuando anch'esso una pausa di un secondo dopo ogni incremento.

```

17 int main(void) {
18     pthread_t mioThread;
19     int i;
20     if (pthread_create (&mioThread, NULL, cosaFaThread, NULL)){
21         printf("errore nella creazione del thread.");
22         abort();
23     }
24     for (i = 0; i < TANTI; i++) {
25         globale = globale + 1;           // scrive valore nella var. condivisa
26         printf("\nho scritto %d", globale);
27         fflush(stdout);
28         sleep(1);
29     }
30     if (pthread_join (mioThread, NULL)){
31         printf("errore nel join del thread.");
32         abort();
33     }
34     printf("\n\n Termine elaborazione ");
35     exit(0);
36 }
```

Compiliamo e mandiamo in esecuzione due o tre volte il programma direttamente da **Dev-cpp**: otterremo probabilmente risultati sempre diversi, come i due riportati nella seguente immagine:



L'esecuzione non è quindi deterministica e l'output dipende dalla schedulazione: se ripetiamo più volte l'esecuzione del programma è probabile che otterremo sempre output diversi!

■ Esecuzione in linux

Potrebbe venire il sospetto che il non determinismo sia legato all'algoritmo di schedulazione del **sistema operativo** e che, quindi, cambiando sistema operativo, possa “scomparire”.

Per toglierci questo dubbio mandiamo in esecuzione lo stesso programma in **linux**: otterremo anche in questo ambiente un output simile a quanto riportato nella immagine seguente.

```

C_pthread : bash - Konsole
File Edit View Bookmarks Settings Help
utente@AspireXC705-Linux:~/sviluppo/utel/C_pthread > gcc contatore1.c -lpthread -o prova
utente@AspireXC705-Linux:~/sviluppo/utel/C_pthread > ./prova

ho scritto 1
.ho letto 1
ho scritto 2
.ho letto 2
ho scritto 3
.ho letto 3
ho scritto 4
.ho letto 4
ho scritto 5
.ho letto 5
ho scritto 6
.ho letto 6
ho scritto 7
.ho letto 7
.ho letto 7
ho scritto 8
.ho letto 8
.ho scritto 9
.ho letto 9
ho scritto 10

Termino elaborazione
C_pthread : bash

```

Quindi già alla prima esecuzione possiamo constatare che il problema è identico a quello riscontrato con l'esecuzione in **Windows**: pertanto possiamo affermare che il “difetto” non è un problema legato al tipo di **sistema operativo**!

La soluzione definitiva viene effettuata introducendo la **sincronizzazione**, cioè facendo in modo che il **main()** non scriva un nuovo valore nella variabile fino a che questo non sia stato visualizzato dal **thread**.

Nelle prossime lezioni introdurremo un meccanismo che ci permette di coordinare le attività utilizzando strumenti appropriati come per esempio i **semafori**.



Prova adesso!

- Esecuzione in Dev-cpp
- Malfunzionamenti nei thread

Esegui ora lo stesso programma sul tuo calcolatore in modo da avere una esecuzione dello stesso codice su di una macchina diversa.

Modifica il valore del tempo di attesa in modo da ottenere, se possibile, sempre lo stesso risultato: discuti e commenta i risultati ottenuti.

ESERCITAZIONI DI LABORATORIO 10

I THREAD IN JAVA: CONCETTI BASE



Info

I codici sorgenti sono nel file **java_thread.rar** scaricabile dalla cartella **materiali** nella sezione del sito www.hoepliscuola.it riservata al presente volume.

■ I thread in Java

Il linguaggio **Java** già dalla sua prima definizione mette a disposizione dell'utente **classi** che implementano i **thread** in forma primitiva ed è anche questo un motivo per cui viene utilizzato come linguaggio di progetto nei corsi per **sistemi operativi**.

Esistono tre possibilità per definire un **thread** in **Java**:

- 1 creare un **main()**;
- 2 creare una **sottoclasse** della classe **Thread**;
- 3 implementare l'interfaccia **Runnable**.

La prima modalità di creazione di un **thread** è implicita nella realizzazione di un programma **Java** in quanto ogni programma **Java** contiene almeno un singolo **thread**, corrispondente all'esecuzione del metodo **main()** sulla **JVM**.

La seconda modalità consiste nel creare i **Thread** come oggetti di sottoclassi della classe **Thread**.

La terza modalità consiste nell'utilizzo dell'interfaccia **Runnable** che risulta avere maggior flessibilità del caso precedente.

In questa esercitazione vedremo come scrivere e come generare un **thread** dinamicamente attivando concorrentemente la sua esecuzione all'interno del programma.

Dato che in un'applicazione si può avere un unico **main()** non sarà possibile utilizzare il primo metodo per creare applicazioni concorrenti: dobbiamo comunque tenere sempre presente che l'esecuzione di un **main** implica la creazione di un **thread** che verrà schedulato insieme agli altri **thread** che definiremo in seguito.

■ Thread come oggetti di sottoclassi della classe Thread

Java mette a disposizione la classe **Thread** (fornita dal package `java.lang`), fondamentale per la gestione dei **thread**, in quanto in essa sono definiti i metodi per avviare, fermare, sospendere, riattivare e così via, come si vede dal diagramma della classe.

Il diagramma delle classi è il seguente:

Classe Thread		
Attributi	Metodi costruttori	Metodi modificatori
<pre>int MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY</pre>	<pre>Thread() Thread(Runnable target) Thread(String name) Thread(ThreadGroup name, Runnable target) Thread(ThreadGroup group, String name) Thread(Runnable target, String name) Thread(ThreadGroup group, Runnable target, String name) Thread(ThreadGroup group, Runnable target, String name, long stackSize)</pre>	<pre>int activeCount() Thread currentThread() void dumpStack() int enumerate(Thread tarray[]) boolean holdsLock(Object obj[]) boolean interrupted() void sleep(long millis) void sleep(long millis, int nanos) void yield() ClassLoader get/setContextClassLoader() String get/setName() int get/setPriority() ThreadGroup getThreadGroup() boolean isAlive() boolean is/setDataemon() boolean isInterrupted() String toString() void checkAccess() void destroy() void interrupt() void join() void join(long millis) void join(long millis, int nanos) void run() void start()</pre>

La prima modalità di creazione di thread in Java viene realizzata ereditando da tale classe una sottoclasse che utilizzi il metodo `run()` con la seguente segnatura:

```
public class MiaClasseThread extends thread{
    public void run(void){
        // metti qui il tuo codice
    }
}
```

Il metodo `run` definisce l'insieme di istruzioni **Java** che ogni **thread** “creato come oggetto della classe” **Thread** eseguirà concorrentemente con gli altri **thread**.

Nella classe **Thread** l'implementazione del suo metodo `run` è vuota, quindi in ogni sottoclasse derivata deve essere ridefinito (override) con le istruzioni che costituiscono il corpo del **thread**, cioè tutte le istruzioni che lo scheduler deve eseguire quando il **thread** viene attivato.

ESEMPIO ***Il thread dei 7 nani***

Un esempio completo è il seguente:

```
public class ContaINanil extends Thread{
    public void run() {
        setName("settenani");
        System.out.println(Thread.currentThread().getName());
        for(int i = 0; i < 7; i++){
            System.out.print((i + 1) + " ");
        }
    }
}
```

Questa semplice classe definisce un segmento di codice che, dopo aver assegnato il nome “**settenani**” al **thread** con il metodo **setName()** e averlo visualizzato sullo schermo, conta da 1 a 7.

Per l'esecuzione di un **thread** che esegua questo codice si deve scrivere una classe di prova che ne definisca un'istanza e ne avvii l'esecuzione: deve essere chiamato il metodo **start()** che invoca il metodo **run()** (il metodo **run()** non può essere chiamato direttamente, ma solo attraverso **start()**).

```
public class ProvaNanil{
    public static void main(String args[]){
        ContaINanil thr1 = new ContaINanil();
        thr1.start();
        // o in una unica istruzione: new ContaINanil().start();
        System.out.println(Thread.currentThread().getName());
    }
}
```

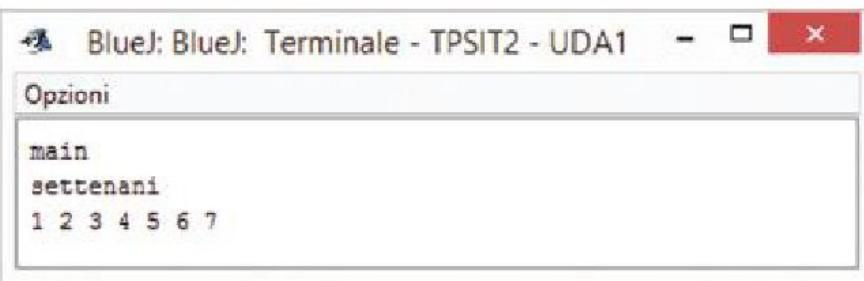
Effettuiamo cioè la definizione di un oggetto in una classe di prova e attiviamo il **thread** con il metodo **start()** che manda in esecuzione il metodo **run()** prima definito.

La creazione di un oggetto **thread** NON determina l'esecuzione: il **thread** è ancora in uno stato simile allo stato di attesa, cioè è in attesa di essere avviato!

In questo modo abbiamo creato due **thread** concorrenti:

- il **thread principale**, associato al **main**;
- il **thread thr1** creato dinamicamente dal precedente, con l'esecuzione dell'istruzione **thr1.start()** che lancia in concorrenza l'esecuzione del metodo **run()** del nuovo **thread**.

Una esecuzione produce sullo schermo il seguente output:



The screenshot shows a terminal window titled "BlueJ: BlueJ: Terminale - TPSIT2 - UDA1". The output text is as follows:

```
Opzioni
main
settenani
1 2 3 4 5 6 7
```

A ogni **thread** viene associato automaticamente un nome: il primo **thread** ha nome **main** ed è inconfondibile dato che deve essere unico; ai successivi **thread** viene assegnato di default un nome con un indice progressivo (per esempio **Thread-6**) a meno che il programmatore provveda alla sua inizializzazione esplicitamente, come fatto da noi.

Il **thread** appena creato continua la sua evoluzione fin tanto che viene completata l'esecuzione del codice, ovvero finché il contatore ha raggiunto e visualizzato il numero 7; quindi termina spontaneamente.

In altri casi, potremmo avere necessità di fermare espressamente l'evoluzione del **thread** tramite un altro **thread**: tale meccanismo viene realizzato semplicemente con l'invocazione del metodo **stop()**

```
thr1.stop();
```

ESEMPIO Due thread in esecuzione

Vediamo un secondo esempio dove inseriamo un costruttore per assegnare il nome al **thread**:

```
public class ContaINani2 extends Thread{
    public ContaINani2(String nome){          // costruttore
        super();
        setName(nome);
    }

    public void run(){
        for(int i=0;i<7;i++){
            System.out.println((i+1)+" "+getName());
        }
    }
}
```

Modifichiamo ora la classe di prova in modo che vengano create le due istanze con i nomi diversi:

```
public class ProvaNani2{
    public static void main(String args[]){
        Thread thr1 = new ContaINani2("topolino");
        Thread thr2 = new ContaINani2("pippo");
        thr1.start();
        thr2.start();
    }
}
```

Tre esecuzioni producono sullo schermo il seguente output:

The figure displays three separate terminal windows, each titled "Blue: Blue: Terminale - ...". Each window contains a list of names under the heading "Opzioni". The first window lists: 1 topolino, 1 pippo, 2 topolino, 2 pippo, 3 topolino, 4 topolino, 3 pippo, 5 topolino, 4 pippo, 5 pippo, 6 topolino, 7 topolino, 6 pippo, 7 pippo. The second window lists: 1 pippo, 1 topolino, 2 topolino, 2 pippo, 3 topolino, 3 pippo, 4 topolino, 4 pippo, 5 topolino, 5 pippo, 6 topolino, 6 pippo, 7 topolino, 7 pippo. The third window lists: 1 pipp... (truncated). This illustrates how different scheduling can lead to different execution orders.

Si può osservare come gli effetti della schedulazione producano risultati diversi.



Prova adesso!

- 1 Modifica la classe di prova in modo che vengano create le sette istanze con i nomi diversi, per esempio quelli dei sette nani.
- 2 Avvia l'esecuzione e osserva e commenta l'output.

■ Thread come classe che implementa l'interfaccia Runnable

La terza possibilità consiste nell'utilizzo dell'interfaccia **Runnable**: con questo meccanismo si ha maggiore flessibilità rispetto a quello appena descritto.

La creazione della **classe** è simile al metodo precedente e le operazioni da eseguire sono le seguenti:

- ▶ codificare il metodo **run()** nella classe che implementa l'interfaccia **Runnable**;
- ▶ creare un'istanza della classe tramite **new**;
- ▶ creare un'istanza della classe **Thread** con un'altra **new**, passando come parametro l'istanza della classe che si è creata;
- ▶ invocare il metodo **start()** sul **thread** creato, producendo la chiamata al suo metodo **run()**.

ESEMPIO **Campane non sincronizzate**

Scriviamo un esempio classico, le “campane che suonano”.

Costruiamo la **classe Campana** e come parametri del costruttore indichiamo il suono che deve emettere e quante volte dovrà essere eseguito.

```

class Campana implements Runnable{
    String suono;
    int volte;
    public Campana(String suono,int volte){
        this.suono = suono;
        this.volte = volte;
    }
    public void run(){
        for(int k = 0; k < volte; k++) {
            System.out.print ((k + 1)+suono+" ");
        }
    }
}

```

Quindi, definiamo la classe che crea tre **thread** e li manda in esecuzione:

```

public class Dindondan{
    public static void main(String args[]){
        //prima modalità di definizione
        Runnable caml = new Campana("din", 5);
        Thread thrl = new Thread(caml);
        thrl.start();
        //seconda modalità di definizione
        Thread thr2 = new Thread(new Campana("don", 5));
        thr2.start();
        // terza modalità di definizione
        new Thread(new Campana("dan", 5)).start();
    }
}

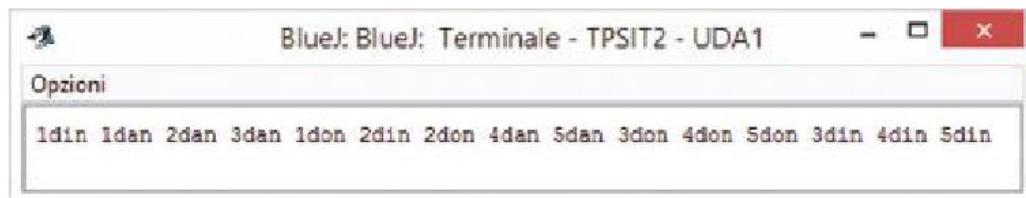
```

Facciamo una prima osservazione sulla differenza del codice dei metodi **run()**: in questo caso non si è potuto utilizzare il metodo **getName()** per stampare il nome del **thread** (e usare tale nome come suono delle campane): infatti, gli oggetti della classe **Campana** non appartengono alla classe **Thread** ma implementano solamente l'interfaccia **Runnable** che ha un unico metodo **run()**.

Si sono dovuti creare successivamente nel **main()** oggetti di tipo **Thread** passandogli come parametro del costruttore l'oggetto **Runnable** creato in precedenza.

Una caratteristica comune del metodo **run()** è invece l'assenza di parametri, sia in ingresso sia in uscita. Questo comporta che, qualora ci sia la necessità di passare parametri al **thread**, dobbiamo servirci del costruttore, passare tali dati come parametri e scriverli nelle variabili di istanza della classe. Il metodo **run()**, così come tutti i metodi della classe, ha visibilità di tali variabili di classe e quindi è possibile accedervi, leggerle e modificarle.

Mandiamo ora in esecuzione la classe **Dindondan** e otteniamo il seguente output:



The screenshot shows a terminal window titled "BlueJ: BlueJ: Terminale - TPSIT2 - UDA1". The window contains the text "Opzioni" and a scrollable text area displaying the output of the program: "1din 1dan 2dan 3dan 1don 2don 4dan 5dan 3don 4don 5don 3din 4din 5din".

mentre una successiva esecuzione dà il seguente risultato, diverso dal precedente:

```
BlueJ: BlueJ: Terminale - TPSIT2 - UDA1
Opzioni
1dan 2dan 3dan 4dan 5dan 1din 2din 3din 1don 4din 2don 5din 3don 4don 5don
```

Quindi, l'output non è sempre uguale bensì dipende dalla schedulazione e dai **processi/thread** che al momento sono nella lista dei processi pronti: questa modalità è quindi **non deterministica** ed è una delle cause che rendono complicata la gestione della **concorrenza**. Se abbiamo bisogno della perfetta “rotazione” dei suoni delle campane, e cioè din seguito da don, seguito da dan, dobbiamo implementare i meccanismi di **sincronizzazione** e **cooperazione** che vedremo in seguito.

Anche se può sembrare più complessa, questa seconda modalità è preferibile per realizzare i **thread**: infatti generalmente le classi che stiamo definendo sono ereditate da altre e, poiché Java non permette l'**ereditarietà multipla**, per avere anche le caratteristiche dei **thread** l'unica soluzione è quella di usare **implements** (come già visto per la gestione degli eventi).



Prova adesso!

- 1 Modifica il programma precedente cercando di ottenere sullo schermo una sequenza ben determinata di suoni, per esempio tre serie di “din-don-dan” semplicemente introducendo dei ritardi per esempio con il metodo sleep(secondi), che blocca per un tempo specificato l'esecuzione di un **thread**.
- 2 Manda in esecuzione più volte il programma per essere sicuro di aver ben sincronizzato (!?) il sistema: quali osservazioni puoi fare?
- 3 Modifica ora il programma introducendo una nuova campana dal suono “dun”, che deve essere alternata a ciascuna altra campana, cioè deve sempre seguire una delle tre campane precedenti.
Per esempio una sequenza possibile è la seguente:
“din-dun” - don-dun - din-dun - dan-dun”

ESERCITAZIONI DI LABORATORIO 11

PRIORITÀ E PARAMETRI NEI THREAD JAVA

■ Passaggio di parametri a un thread

Il metodo `run()` che attiva l'esecuzione di un `thread` non ha parametri formali e quindi non è possibile passare valori attuali per diversificare le esecuzioni in modo da far fare eseguire operazioni diverse in base a valori diversi. Non è possibile neppure utilizzare il metodo `start()`: anch'esso non ha parametri.

La **prima soluzione** che ci permette di passare parametri a un `thread` si ottiene utilizzando i parametri del metodo costruttore. Vediamo un esempio dove si vuole fa visualizzare ad un `thread` i numeri pari e un altro i numeri dispari fino al raggiungimento di un valore N.

Utilizziamo una variabile booleana (**pari**) che determinerà se il processo deve stampare i numeri pari o i numeri dispari. ►

```
public class PariDispari extends Thread{
    private int massimo;
    private boolean pari;
    private int ritardo = 500;

    public PariDispari (int finale, boolean pari){
        massimo = finale;
        this.pari = pari;
    }
}
```

Il metodo `run()` deve semplicemente controllare la variabile e comportarsi di conseguenza: per verificare il corretto funzionamento insieme al numero visualizziamo anche il nome del `thread` che ne effettua la stampa.

```
11  public void run(){
12      String chisono;
13      chisono = Thread.currentThread().getName();
14      for (int xx = 0; xx < massimo; xx++){
15          if(pari){           // se è il thread che deve stampare i numeri pari
16              if(xx % 2 == 0) // numero pari
17                  System.out.println(chisono+"-pari "+xx);
18          }
19          else                // se è il thread che deve stampare i numeri dispari
20              if (xx % 2 != 0) // numero dispari
21                  System.out.println(chisono+"-dispari "+xx);
22          try {Thread.sleep(ritardo);}
23          catch (InterruptedException e){System.out.println(e);}
24      }
25  }
```

Il primo **thread** deve essere attivato in un contesto in cui il flag pari viene inizializzato a **true** mentre il secondo deve avere lo stesso flag con il valore **false**.

Utilizziamo una variabile d'istanza per ogni dato che intendiamo “personalizzare” nei **thread** e passiamo tali parametri nel metodo costruttore, avendo letto il numero **n** dei numeri da visualizzare in input come argomento del **main()**:

```

1 public static void main(String[] args){
2     if (args.length != 1){
3         System.out.println("Deve restituire il numero intero");
4         return;
5     }
6     int n = Integer.parseInt(args[0]);
7     Thread TP = new PariDispari (n + 1, true); // thread che conta i pari
8     Thread TD = new PariDispari (n + 1, false); // thread che conta i dispari
9     System.out.println(">Contate fino a " + n);
10    TP.start(); // avvio esecuzione thread
11    TD.start();
12    try{
13        TP.join(); // attesa terminazione thread
14        TD.join();
15    }
16    catch(Exception e){}
17    System.out.println("<-Fine conteggio!");
18 }
```

L'esecuzione produrrà un output simile al seguente: ►

Per ottenere l'alternanza corretta della visualizzazione dei numeri abbiamo dovuto inserire un ritardo: sappiamo che questo NON è un meccanismo di sincronizzazione bensì un “artificio” per far schedulare i **thread** alternativamente dal sistema operativo; vedremo come correggere questo esercizio nella prossima unità di apprendimento.

```
->Contate fino a 9
Thread-5-pari 0
Thread-6-dispari 1
Thread-5-pari 2
Thread-6-dispari 3
Thread-5-pari 4
Thread-6-dispari 5
Thread-5-pari 6
Thread-6-dispari 7
Thread-5-pari 8
<-Fine conteggio!
```

La **seconda soluzione** si ottiene mediante una **classe** nella quale si scrive il metodo costruttore con encapsulato la creazione del **thread** in modo da rendere possibile il passaggio di un parametro: definiamo una **classe Contatore()** con il costruttore che riceve due parametri, ne assegna i valori alla variabili locali e crea un **thread**.

```

1 class Contatore extends Thread{
2     private int massimo;
3     private boolean pari;
4     private Thread PID;
5     private int ritardo = 500;
6
7     public Contatore (int finale, boolean pari){
8         massimo = finale;
9         this.pari = pari;
10        PID = new Thread(this);
11        PID.start();
12    }
13 }
```

Il metodo `run()` è lo stesso dell'esercizio precedente: ci rimane ora da definire la classe di prova che è riportata di seguito:

```
# public class ProvaConta{
#   public static void main(String[] args){
#     int n = 10 ;
#     System.out.println(">Contate fino a " + n);
#     Thread TP = new Contatore (n, true);
#     Thread TD = new Contatore (n, false);
#   }
# }
```

L'output è praticamente identico al precedente.

Le due modalità portano al medesimo risultato, quindi possono essere impiegate indifferentemente: va tuttavia osservato come in questo secondo caso la dichiarazione del thread è distinta dalla sua attivazione.



Prova adesso!

- 1 Calcola con N thread differenti i valori del fattoriale di N numeri naturali generati casualmente in [1,10] verificando che non ci siano ripetizioni.
- 2 Memorizzali in una matrice condivisa e visualizzala sullo schermo.

■ Priorità di un thread

I **thread** nella loro evoluzione si comportano come veri e propri processi in termini di occupazione di **risorse** nella schedulazione della **CPU** secondo le politiche della **round robin** del sistema operativo: ogni singolo **thread** partecipa alla partizione di tempo con tutti gli altri, acquisendone il relativo **time slice**.

Tutti i **thread** hanno lo stesso tempo di **CPU** a disposizione, non essendoci motivazioni particolari per attribuire esecuzioni privilegiate.

In **Java** ogni **thread** eredita, all'atto della sua creazione, la priorità del processo padre: è possibile modificare le politiche di allocazione delle risorse assegnando ai **thread** **priorità diverse** mediante l'inizializzazione di un valore intero compreso tra 1 e 10 (1 è il valore associato al minimo livello di priorità mentre 10 è il massimo).

Il metodo da utilizzare è il seguente:

```
public final void setPriority(int priorita)
```

Il corrispondente metodo **get** restituisce il valore attualmente impostato nel **thread**:

```
public int getPriority(int priorita)
```

Sono definite le costanti riportate nella tabella.

Identificatore	Valore
MIN_PRIORITY	1
NORM_PRIORITY	5
MAX_PRIORITY	10

Vediamo un esempio dove due **thread** con diversa priorità stampano a video il proprio nome:

```

1 public class Priorita extends Thread{
2     private String chisono;
3
4     public Priorita (String nome){
5         setChiseno(nome);
6
7     }
8
9     public String getChiseno(){
10        return chiseno;
11    }
12
13    public void setChiseno(string nome){
14        chiseno = nome;
15    }
16
17    public void run(){
18        int conta = 0;
19        while(conta < 5000000){
20            conta++;
21            if((conta % 1000000) == 0)
22                System.out.println("Thread #"+chiseno+", conta = "+conta);
23        }
24    }
25 }
```

Il metodo **main()** passa come parametro ai due processi la “loro importanza” settandone uno al valore massimo (**MAX_PRIORITY**) e uno al minimo (**MIN_PRIORITY**):

```

1 public static void main (String [] args)
2 {
3     Thread TA = new Priorita ("IMPORTANTE (10)");
4     Thread TB = new Priorita ("poco importante");
5     TA.setPriority(Thread.MAX_PRIORITY);
6     TB.setPriority(Thread.MIN_PRIORITY);
7     TA.start();
8     TB.start();
9 }
```

Un'esecuzione genera il seguente output: ▶

```

Thread #IMPORTANTE (10), conta = 1000000
Thread #IMPORTANTE (10), conta = 2000000
Thread #poco importante, conta = 1000000
Thread #IMPORTANTE (10), conta = 3000000
Thread #IMPORTANTE (10), conta = 4000000
Thread #poco importante, conta = 2000000
Thread #IMPORTANTE (10), conta = 5000000
Thread #poco importante, conta = 3000000
Thread #poco importante, conta = 4000000
Thread #poco importante, conta = 5000000

```

Un caso particolare potrebbe essere quello in cui tanti **thread** ad alta priorità ostacolano l'evoluzione di un **thread** a bassa priorità: affronteremo tale argomento quando ci occuperemo della cooperazione tra i **thread**.

È comunque disponibile un metodo (**yield()**) con cui si può temporaneamente ridurre la priorità di un **thread** per fare in modo che anche quelli con bassa priorità possano evolvere.

Un particolare **thread** a bassa priorità è il **garbage collector**: esso rientra in una famiglia di **thread** particolari, detti **thread deamon**, che sono servizi di sistema attivati quando la **CPU** ha un basso numero di risorse utilizzate.



Prova adesso!

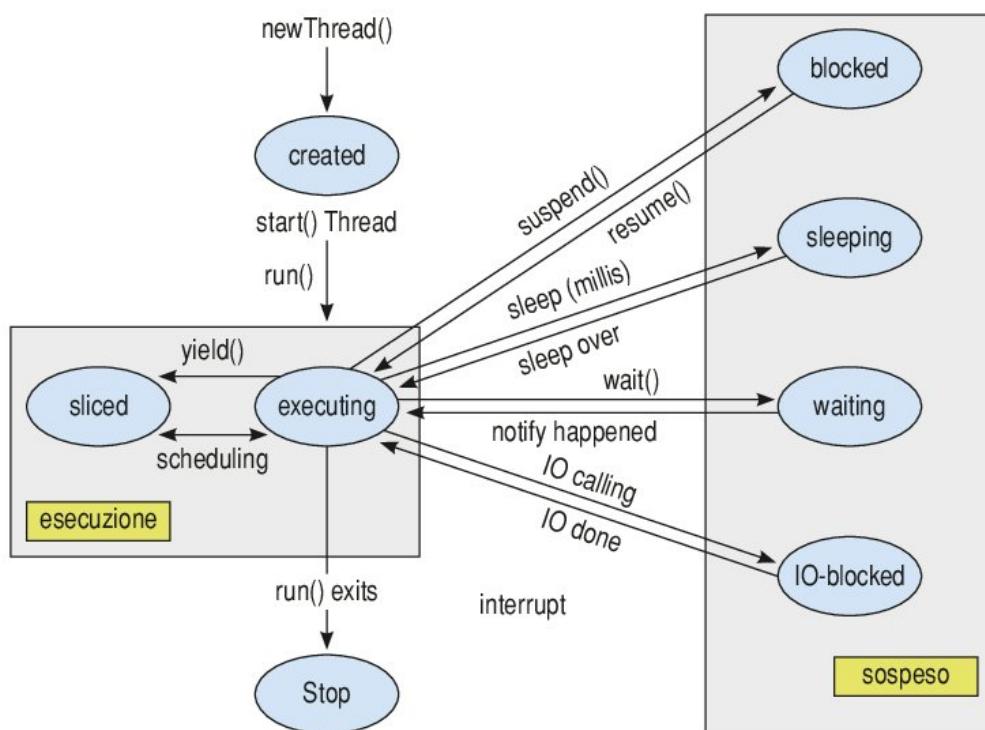
- 1 Realizza un programma in Java che attivi tre contatori indipendenti e ne visualizzi i valori (totali o parziali) in un intervallo massimo di 5 secondi.
- 2 Il contatore deve incrementare una proprietà interna (pubblica o privata) ed effettuare una visualizzazione a ogni ciclo.
- 3 Quindi a ogni contatore assegna una priorità iniziale diversa.
- 4 Analizza i risultati proponendo le relative considerazioni.

ESERCITAZIONI DI LABORATORIO 12

I THREAD JAVA: I METODI SLEEP, YIELD E JOIN

■ Metodi per i cambiamenti di stato

Nella lezione 3 abbiamo descritto i diversi stati che può assumere un **thread**: nella figura seguente riportiamo il diagramma completo degli stati di un **thread Java** indicando i nomi dei diversi metodi che possono causare il passaggio di stato.



Descriviamoli succintamente:

- **start()** fa partire l'esecuzione di un **thread**: la **JVM** invoca il metodo **run()** del **thread** appena creato;
- **stop()** forza la **terminazione** dell'esecuzione di un **thread**;
- **suspend()** blocca l'esecuzione di un **thread** in attesa di una successiva operazione di **resume()**; non libera le risorse impegnate dal **thread**;
- **resume()** riprende l'esecuzione di un **thread** precedentemente **sospeso**;

- `sleep(long t)` blocca per un **tempo specificato** (`time`) l'esecuzione di un **thread**;
- `join()` blocca il **thread** chiamante in attesa della **terminazione** del **thread** di cui si invoca il metodo;
- `yield()` sospende l'esecuzione del **thread** invocante, lasciando il controllo della **CPU** agli altri **thread** in coda d'attesa.

Vediamo alcuni esempi del loro utilizzo.

Rilascio del processore

È consigliabile l'utilizzo di `stop()`, `suspend()` e `resume()` dato che non sono sicuri in quanto agiscono in modo diretto su un **thread** senza effettuare alcun controllo del suo stato o della sua evoluzione; possono quindi determinare situazioni di **stallo** e di **deadlock** (lezione 7 dell'unità di apprendimento 2):

- se fermiamo il **thread** appena questi ha acquisito il controllo di una risorsa, quest'ultima rimane a lui allocata e quindi indisponibile a tutto il sistema;
- se successivamente un **thread** avesse necessità di utilizzare tale risorsa rimarrebbe "perennemente" in attesa;
- se fermiamo il **thread** mentre sta effettuando un'operazione critica e non la termina, ci portiamo in una situazione di inconsistenza.

Nessun problema per il metodo `sleep(int)`, in quanto sospende solo temporaneamente il **thread** e lo riattiva dopo `int` millisecondi ripartendo dalla stessa istruzione dalla quale è stato interrotto.

Il metodo `yield()`, invece, consente di trasferire il controllo del processore a un altro **thread** rimettendosi in coda, in attesa di poter riacquisire l'esecuzione: questa situazione prende il nome di **coroutining**.

■ Il metodo `sleep()`

Il metodo `sleep()` è un **metodo statico** che sospende il **thread** corrente e viene utilizzato per "far fare una pausa" al **thread** senza che questo utilizzi cicli del processore, cioè senza **attesa attiva**. La pausa di un **thread** può essere **interrotta** da un altro **thread** con il metodo `resume()`: in tale situazione viene sollevata un'eccezione **InterruptedException** e, quindi, è necessario che `sleep()` sia eseguito in un blocco **try-catch**.

Vediamo un semplice esempio dove introduciamo anche un meccanismo per riconoscere i **thread**.

ESEMPIO

Indichiamo come costante il tempo di **attesa** che viene espresso in millisecondi e lo poniamo al valore di 200.

Ad ogni **thread** che viene avviato il sistema assegna un nome interno, che è costituito dalla parola **thread** seguita da un numero progressivo: questo individua univocamente i **thread** ed è possibile leggerlo mediante il metodo:

```
Thread.currentThread()
```

Viene restituito un dato di tipo **Thread** e lo memorizziamo in una variabile di classe **padre** che quindi conterrà l'identificatore del "padre di tutti i **thread**".

```

1 public class EsempioSleep1 extends Thread{
2     static int tanti = 2;
3     static int attesa = 200;
4     private Thread padre;
5
6     public EsempioSleep1(){
7         padre = Thread.currentThread();
8     }
9
10    public void run(){
11        for (int x = 0; x < tanti; x++){
12            System.out.println("Nuovo thread");
13            printNome();
14            try {Thread.sleep(attesa);} catch (InterruptedException e) {return;}
15        }
16        System.err.println("\n" + getName() + " finito");
17    }

```

Usiamo l'identificatore del **thread** per scrivere sullo schermo il “nome” del **thread** mediante il metodo **printNome()**: all'interno di questo confrontiamo con **this** il contenuto della variabile **padre** in modo da visualizzare sullo schermo se il codice viene eseguito dal **thread** padre oppure dal figlio.

```

1 public void printNome(){
2     Thread nome = Thread.currentThread();
3     if (nome == padre){
4         System.out.println("Thread padre");
5     }
6     else{
7         if (nome == this)
8             System.out.println("Nuovo thread");
9         else
10            System.out.println("Thread ignoto");
11    }
12}

```

Facciamo ripetere un gruppo di istruzioni per un numero di volte indicato dalla variabile **tanti** sia nel **thread** che nel **main()**, e tra le istruzioni inseriamo l'istruzione che visualizza il nome del **thread** e la funzione **sleep(atteca)**:

- ▷ nel metodo **run()** all'interno di un il costrutto **try-cath**;
- ▷ nel **main()** rimandiamo la gestione dell'eccezione con la clausola **throws**.

```

1 public static void main(String[] args) throws InterruptedException{
2     new EsempioSleep1().start();
3     for(int x = 0; x < tanti; ++x){
4         System.out.println("Main thread");
5         Thread.sleep(atteca);
6     }
7 }

```

Solo un **thread** può essere in pausa e quindi non è possibile mettere in pausa un altro **thread** contemporaneamente al primo: se un secondo **thread** viene messo in **sleep()** automaticamente viene risvegliato quello che stava dormendo.

Una esecuzione del programma è la seguente:

```
Main thread
Nuovo thread
Nuovo thread
Main thread
Nuovo thread
Nuovo thread
Thread-1 finito
```

Dato che nel metodo `run()` abbiamo inserito come istruzione 17 di output il device `err` ci viene visualizzato il nome del **thread** nella finestra che riporta gli errori.

■ Il metodo `yield()`

Con il metodo `yield()` il **thread** non viene interrotto immediatamente: gli si segnala la necessità che esso si sospenda, lasciandogli però la scelta del momento opportuno, che sarà un momento non critico.

ESEMPIO

Vediamo un esempio che simula una partita di ping-pong iniziando con la definizione di una **classe** `Racchetta` che nel metodo `run()` scrive il contenuto della variabile `pallina`, si sospende per un secondo (per facilitare la visualizzazione sullo schermo) e quindi cede l'esecuzione a un altro **thread** con il metodo `yield()`.

```
class Racchetta implements Runnable{
    String pallina;
    public Racchetta(String pallina){
        this.pallina = pallina;
    }
    public void run(){
        while (true){
            System.out.println(pallina);
            try{
                Thread.sleep(1000); // ritardo solo per visualizzazione
            } catch (InterruptedException e) {}
            Thread.yield(); // cede l'esecuzione ad un altro thread
        }
    }
}
```

La **classe** che crea i due **thread** è la seguente:

```
public class PingPong{
    public static void main(String args[]){
        Thread thrl = new Thread(new Racchetta("ping"));
        thrl.start();
        // secondo giocatore
        Thread thr2 = new Thread(new Racchetta("pong"));
        thr2.start();
    }
}
```

L'echo sullo schermo è una sequenza infinita di scritte "ping pong" che si ripetono con ritardo di un secondo impostato dalla funzione `sleep()`.

L'osservazione dell'echo sullo schermo ci mostra però un comportamento indesiderato dato che non viene rispettata l'alternanza del ping con il pong: il problema è dovuto alla assenza di **sincronizzazione** e i singoli **thread** sono schedulati dal sistema operativo in modo "casuale". Nella prossima UA vedremo come regolare correttamente l'esecuzione dei **thread**.



■ Il metodo `join()`

Quando un processo manda in esecuzione un **thread**, continua anch'esso la propria evoluzione: si è detto che il `main()` è un **thread** e sino a ora lo si è utilizzato per creare e mandare in esecuzione altri **thread** che successivamente vivono di vita propria (ossia: anche se il processo, o **thread**, che li ha generati termina, questi continuano la propria esecuzione).

In alcuni casi, potrebbe risultare necessario attendere la terminazione di un **thread** prima di continuare l'esecuzione, il che implica sospendere un **thread** in attesa che un secondo **thread** termini per riprendere quindi l'esecuzione del primo **thread**. **Java** mette a disposizione l'operazione `join()` da eseguirsi sul **thread** che si intende aspettare.

ESEMPIO

Vediamo per esempio un `main()` che manda in esecuzione due **thread** per poi sospendersi in attesa della terminazione di entrambi; in linea di principio, il codice è il seguente:

```
public static void main(String [] a){
    ...
    //avvio l'esecuzione dei due thread
    thr1.start();
    thr2.start();
    try{
        //attendo la loro terminazione
        thr1.join();
        thr2.join();
    }
    catch(InterruptedException e){
        System.out.println(e);
    }
}
```

Ma anche il metodo `join()` può generare una `InterruptedException` e quindi deve essere eseguito in un segmento `try/catch`.

Realizziamo prima il codice eseguito dai due **thread**: per esempio, facciamo scrivere il loro nome e li facciamo attendere un secondo con il metodo **sleep()** prima di terminare:

```

public class Aspettali implements Runnable{
    String mionome;
    int tempodormi;
    Aspettali(String chisono, int quanto){
        mionome = chisono;
        tempodormi = quanto;
    }
    public void run(){
        try{
            System.out.println("il thread "+mionome+" ora si sospende ");
            Thread.sleep(tempodormi);
            System.out.println("il thread "+mionome+" si sveglia e termina ");
        }
        catch(InterruptedException e){
            System.out.println(e);
        }
        return;
    }
}

```

Nel metodo **main()**, oltre che attivare due **thread**, inseriamo anche le istruzioni che ci permettono di verificare il tempo totale di esecuzione del **main()** e quindi calcoliamo il tempo totale di esecuzione del programma. Ci serviamo di un metodo statico della classe **System: static long currentTimeMillis()**.

Java è stato realizzato e implementato dapprima su sistema Unix, dal quale quindi "eredita" il conteggio del tempo: "l'inizio del mondo Unix" è la mezzanotte del 1° gennaio 1970 e il conteggio del tempo avviene contando i millisecondi trascorsi da tale data.

Il metodo **static long currentTimeMillis()** fornisce proprio tale valore.

```

public static void main(String [] a){
    long start = 0, stop = 0, delta = 0;
    int quanti = 5000;
    Thread thr1 = new Thread(new Aspettali("ali", quanti));
    Thread thr2 = new Thread(new Aspettali("baba", quanti));
    System.out.println("Ogni thread attende "+ quanti +" millisecondi");
    //leggo il tempo alla creazione dei thread
    start = System.currentTimeMillis();
    //avvio l'esecuzione dei due thread
    thr1.start();
    thr2.start();
    try{ //attendo la loro terminazione
        thr1.join();
        thr2.join();
        //leggo il tempo alla fine dei thread
        stop = System.currentTimeMillis();
        System.out.println("il main riprende l'elaborazione ");
    }
    catch(InterruptedException e){
        System.out.println(e);
    }
    delta = (stop - start) / 1000;
    System.out.println("L'elaborazione è durata: "+delta+" secondi");
}
}

```

Il risultato dell'elaborazione è il seguente:

```
Ogni thread attende 5000 millisecondi
il thread baba ora si sospende
il thread alii ora si sospende
il thread alii si sveglia e termina
il thread baba si sveglia e termina
il main riprende l'elaborazione
L'elaborazione è durata: 5 secondi
```

Il tempo totale di attesa è di circa 5 secondi e non 10, che è la somma delle attese dei due processi: abbiamo così verificato che `sleep()` non è un'attesa attiva, cioè non occupa tempo di CPU!



Prova adesso!

Utilizzando questo meccanismo, ovvero ricorrendo a `join()`, realizza correttamente l'esercizio delle campane facendo cioè in modo che a ogni `din` segua un `don`, a ogni `don` segua un `dan` e via di seguito!

■ Un calcolo parallelo con `join()`

Scriviamo un programma che ci permette di valutare una espressione in modo parallelo. Supponiamo di dover risolvere:

$$k = 3 * (a-1) + 2 * (b-2) * 3 * (c-3)$$

e di far eseguire i singoli addendi a thread diversi:

```
thr1 → x = 3 * (a-1)
thr2 → y = 2 * (b-2)
thr3 → z = 3 * (c-3)
```

quindi:

$$k = (\text{thr1})\text{join}(\text{thr2})\text{join}(\text{thr3})$$

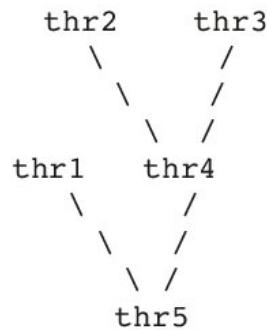
la somma finale viene ottenuta con due somme parziali, quindi nuovamente utilizzando due **thread** che aspettano i risultati parziali prima di poter operare:

```
thr4 → t = y * z
thr5 → k = x + t
```

cioè:

```
thr4 = (\text{thr2})\text{join}(\text{thr3})
thr5 = (\text{thr1})\text{join}(\text{thr4})
```

Il **grafo delle precedenze** si può rappresentare nel modo seguente



Scriviamo una **classe** apposita che contenga i dati iniziali **a**, **b**, **c** sui quali effettuare le elaborazioni e dove i singoli **thread** scrivano i risultati dei loro calcoli: ►

```

public class Buffer {
    public double x, y, z, t, k, a, b, c; // variabili condivise
    public Buffer() { // costruttore
        x = 0; y = 0; z = 0; t = 0; k = 0;
        a = 0; b = 0; c = 0;
    }
    public Buffer(double aa, double bb, double cc) {
        x = 0; y = 0; z = 0; t = 0; k = 0;
        a = aa;
        b = bb;
        c = cc;
        System.out.println(" I parametri valgono: a = " + a + " b = " + b + " c = " + c);
    }
}
  
```

Scriviamo una apposita **classe** per ciascun **thread** in modo che il costruttore legga dal buffer i dati che deve elaborare e il metodo **run()** effettui i calcoli e scriva il risultato sempre nel buffer. A titolo di esempio riportiamo quella che esegue la prima operazione parallela, cioè quella dalla quale istanziamo il **thr1**:

```

public class Operazione1 extends Thread {
    /* Calcola x = 3 * (a - 1)
     *   x = c * (a - b) */
    Buffer dati;
    private double b = 1; // costante1
    private double c = 3; // costante2
    private double a; // parametro

    public Operazione1(Buffer d) {
        dati = d; // passaggio dall'oggetto con i dati
        a = dati.a; // valore di 'a'
    }

    public void run() {
        dati.x = c * (a - b);
        System.out.println(" Ho calcolato x : " + dati.x);
    }
}
  
```

Il programma principale si limita a creare i **thread**:

```

public class CalcoloParallelo {
    public static void main(String[] args) {
        double a, b, c;
        a = 2;
        b = 3;
        c = 4;
        System.out.println("Devo calcolare: 3*(a-1) + 2*(b-2) + 3*(c-3)");
        Buffer parziali = new Buffer(a, b, c); // init parametri
        Operazione1 thr1 = new Operazione1(parziali); // singoli thread
        Operazione2 thr2 = new Operazione2(parziali);
        Operazione3 thr3 = new Operazione3(parziali);
        Operazione4 thr4 = new Operazione4(parziali);
        Operazione5 thr5 = new Operazione5(parziali);
    }
}
  
```

a mandarli in esecuzione e ad attenderne la loro terminazione:

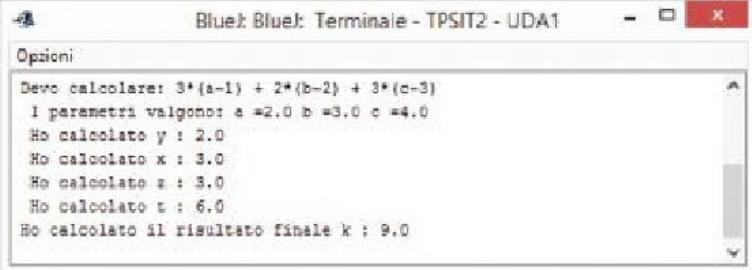
```

17    thr1.start();           // calcolo concorrente
18    thr2.start();
19    thr3.start();
20    try { thr2.join(); }
21    catch(InterruptedException e) {System.out.println("Errore thr2");}
22    try { thr3.join(); }
23    catch(InterruptedException e) {System.out.println("Errore thr3");}
24
25    thr4.start();
26    try { thr1.join(); }
27    catch(InterruptedException e) {System.out.println("Errore thr1");}
28    try { thr4.join(); }
29    catch(InterruptedException e) {System.out.println("Errore thr4");}
30
31    thr5.start();
32    try { thr5.join(); }
33    catch(InterruptedException e) {System.out.println("Errore thr5");}
34
35}
36

```

In **Java** è il programmatore che deve prestare attenzione a scrivere il codice in modo che venga rispettato l'ordine di avvio e di terminazione per la corretta realizzazione dei costrutti **fork-join** desiderati.

Una esecuzione del programma visualizza i seguenti risultati:



The terminal window displays the following output:

```

Blue: Blue: Terminale - TPSIT2 - UDA1
Opzioni
Devo calcolare: 3*(a-1) + 2*(b-2) + 3*(c-3)
I parametri valgono: a =2.0 b =3.0 c =4.0
Ho calcolato y : 2.0
Ho calcolato x : 3.0
Ho calcolato z : 3.0
Ho calcolato t : 6.0
Ho calcolato il risultato finale k : 9.0

```

AREA digitale

 Thread e animazioni



Prova adesso!

- Costrutto fork-join

Dopo aver realizzato il diagramma delle precedenze, scrivi i programmi che implementando il costrutto **fork-join** eseguono le seguenti operazioni con il massimo grado di parallelismo leggendo come parametro i valori per i coefficienti **a**, **b** e **c**.

$$\begin{aligned}
 & 5 * [(2a + 4) * (7b + 3)] - 10c \\
 & (3 + 2a) * (5b - 7) + (8 - 3c) \\
 & (3 + a) - (5 - 2c) * (7by + 3) + 2a
 \end{aligned}$$

2

Comunicazione e sincronizzazione

- L1 La comunicazione tra processi
- L2 La sincronizzazione tra processi
- L3 I semafori
- L4 Applicazione dei semafori
- L5 Il problema dei produttori/consumatori
- L6 Il problema dei lettori/scrittori
- L7 Il problema del deadlock: banchiere e filosofi a cena
- L8 I monitor
- L9 Lo scambio di messaggi

Esercitazioni di laboratorio

- 1 La comunicazione con segnali asincroni; 2 Thread e schedulazione
- 3 I semafori binari in C; 4 La soluzione del deadlock in C; 5 La soluzione del problema produttori/consumatori con i semafori; 6 Variabili condizione; 7 I monitor con le variabili condition in C; 8 I monitor con i semafori in C; 9 I semafori in Java; 10 I monitor in Java; 11 Un esempio con i Java thread; 12 I deadlock in Java

Conoscenze

- Conoscere il modello ad ambiente globale e locale
- Comprendere l'esigenza di sincronizzazione
- Comprendere il concetto di indivisibilità di una primitiva
- Sapere il funzionamento dei semafori di Dijkstra
- Avere il concetto di regione critica e di mutua esclusione
- Sapere la differenza tra interleaving e overlapping
- Comprendere le condizioni di Bernstein
- Avere il concetto di starvation e di deadlock
- Comprendere le proprietà di safety, di fairness e di liveness

Competenze

- Individuare le tipologie di errori nei processi paralleli
- Definire e utilizzare i semafori di basso livello e spin lock()
- Utilizzare gli strumenti di sincronizzazione per thread in C
- Utilizzare le condition variable in C
- Implementare i monitor in C
- Utilizzare gli strumenti di sincronizzazione per thread in C
- Implementare i monitor in Java

Abilità

- Risolvere le situazioni di starvation
- Risolvere le situazioni di deadlock
- Risolvere i problemi produttore/consumatore in C
- Risolvere il problema dei filosofi in C
- Risolvere i problemi produttore/consumatore in Java
- Risolvere il problema dei filosofi con il linguaggio in Java

AREA digitale



Esercizi



Classificazione di Flynn

Applicazioni in real time

Esempio riepilogativo
(non informatico)

Grafo di Holt e grafo di attesa

Tabella dei segnali UNIX/LINUX

Quando viene notificato/gestito un
segnaile

Gruppi di processi

Impostare una sveglia

Nota per gli utenti MAC

Soluzione del problema del
produttore/consumatore con
i mutex

Mutex per la gestione di vincoli di
precedenza

Esercizi per il recupero

Esercizi per l'approfondimento



Soluzioni (prova adesso, esercizi, verifiche)

Puoi scaricare il file anche da hoepliscuola.it

La comunicazione tra processi

In questa lezione impareremo...

- ▶ il modello ad ambiente globale o a memoria condivisa
- ▶ il modello ad ambiente locale o a scambio di messaggi

■ Comunicazione: modelli software e hardware

In un ambiente di **processi concorrenti** le situazioni e le possibilità nelle quali i processi devono **comunicare** tra loro sono molteplici.

Innanzitutto la possibilità può essere offerta dalla tipologia della architettura hardware e, come già detto, noi ci occuperemo solo di architetture monoprocessoressi, di quelle cioè che nella classificazione di **Flynn** si basano sul modello **Von Neumann** (modello sequenziale con una sola capacità di esecuzione **SISD Singol Instruction Singol Data**).

Ma anche in questo caso, cioè considerando solo macchine monoprocessoressi, in un sistema di rete è connessa una molteplicità di macchine di **von Neumann** che lavorano su flussi di esecuzione paralleli e quindi interagiscono tra loro **condividendo risorse comuni**.

Inoltre su una macchina monoprocessoressi il **SO multitasking** permette l'esecuzione contemporanea di più **processi** che competono per accedere alle **risorse comuni**.

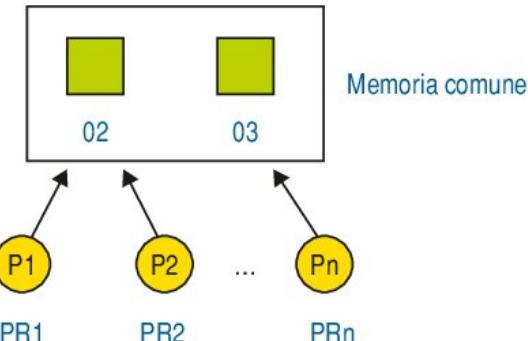
Possiamo individuare due modelli di **interazione concorrente** a prescindere dalla soluzione hardware, cioè sia che stiamo analizzando una architettura distribuita oppure una situazione di multitask su macchina **SISM**:

- ▶ modello a **memoria comune** (ambiente **globale**, global environment);
- ▶ modello a **scambio di messaggi** (ambiente **locale**, message passing).

Entrambi i modelli si basano sul concetto di interazione tra i due elementi che costituiscono il sistema: i **processi interagiscono** per entrare in possesso (utilizzare) delle **risorse** (oggetti).

■ Modello a memoria comune (ambiente globale, global environment)

Il **modello a memoria comune** trova naturale impiego nelle architetture in cui esiste un'unica memoria comune a tutti i **processi** (o processori), per esempio su macchine **monoprocessoressi** con processi multitasking.



È invece complesso (e costoso) condividere memoria nei sistemi di elaborazione distribuiti, nei quali si preferisce utilizzare il meccanismo a **scambio di messaggi**.

Allocazione delle risorse ai processi

Il **modello a memoria comune** è il solo caso nel quale possono verificarsi problemi per l'accesso a essa da parte di due (o più) **processi** che ne potrebbero richiedere l'attribuzione contemporaneamente.

Il **sistema operativo** associa a ogni **risorsa** un apposito **gestore di risorsa** (o **allocatore**), cioè un segmento di codice che gestisce tutte le richieste fatte dai diversi **processi** che necessitano di utilizzare la specifica **risorsa** della quale ne è l'**allocatore**, cioè può assegnarla a un **processo** o negarla a seconda dello stato in cui si trova (per esempio può essere "occupata", cioè in uso a un altro **processo**).

Sostanzialmente possiamo riassumere i suoi compiti in:

- 1 deve mantenere aggiornato lo **stato di allocazione della risorsa**;
- 2 deve **fornire i meccanismi** ai processi che hanno il diritto di utilizzare tale risorsa di accedervi, prenderne possesso, operare su di essa e alla fine del suo utilizzo di "liberarla" per gli altri **processi**;
- 3 deve **implementare la strategia di allocazione** della risorsa definendo a quale **processo** e per quanto tempo assegnare la **risorsa**.

In generale un allocatore si interfaccia con i processi mediante due procedure:

- al momento in cui un **processo** ha bisogno di una risorsa fa una richiesta di assegnazione (**acquisizione**) mediante la prima procedura;
- la seconda viene chiamata dal **processo** che sta utilizzando la risorsa quando termina di averne bisogno e intende "renderla libera" restituendola al sistema operativo (**rilascio**).

ESEMPIO

Pensiamo per esempio a un **processo** che necessita di stampare un documento: in questo caso il **gestore della risorsa** è lo **spooler di stampa** che all'atto di una richiesta la pone in una coda di attesa e la soddisfa quando un **processo** termina di stampare e lascia libera la stampante.

Tipologie di allocazione delle risorse nel modello ad ambiente globale

Come **risorsa** intendiamo qualunque oggetto, fisico o logico, di cui un **processo** necessita per portare a termine il suo compito: le **risorse** vengono raggruppate in classi e una classe identifica le **risorse** che hanno in comune tutte e sole le operazioni che un **processo** può eseguire per operare su ciascun componente di quella classe.

Anche una istanza di una struttura dati allocata nella *memoria comune* è una risorsa e, come vedremo, sarà proprio tramite questa che più processi o **thread** si scambiano informazioni.

Possiamo classificare le risorse in **private** e **comuni**, in base al modello di macchina e al loro tipo di allocazione.

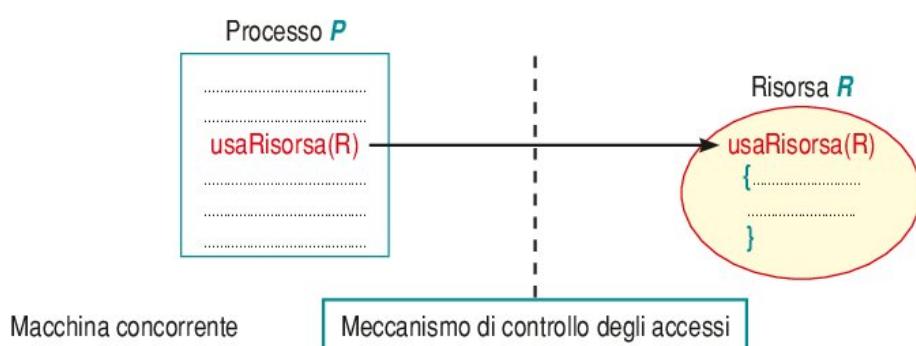
Risorse dedicate (visibili in ogni istante da un solo processo)		Risorse condivise (visibili in ogni istante anche da più processi contemporaneamente)
risorse allocate staticamente	risorse private	risorse comuni
risorse allocate dinamicamente	risorse comuni	risorse comuni

Per ogni risorsa il relativo **gestore** definisce istante per istante i processi che hanno il diritto di operare su di essa.

Le risorse **allocate staticamente** vengono definite prima che il programma inizi la propria esecuzione e quindi il loro gestore è il **programmatore**.

- ▶ Nel caso di **risorse dedicate**, sia allocate staticamente che dinamicamente, non è necessario nessun controllo da parte del programmatore per quanto riguarda la **sincronizzazione** dato che queste sono di utilizzo esclusivo di un singolo **processo** e vengono assegnate e gestite dal **sistema operativo**.
- ▶ Nel caso di **risorse condivise** il programmatore, utilizzando i costrutti del linguaggio di programmazione, stabilisce le regole di visibilità e quindi quali **processi** possono operare sui dati comuni, in quali istanti possono accedere alla **risorsa**, definendo le modalità di **sincronizzazione**.

Gli accessi a una **risorsa condivisa** devono avvenire in modo **non divisibile**: le funzioni che utilizzano un dato condiviso (**sezioni critiche**) devono essere programmate utilizzando i meccanismi di **sincronizzazione** offerti dal linguaggio di programmazione e supportati dalla macchina **concorrente**.





MUTUA ESCLUSIONE

L'accesso a una **risorsa** si dice **mutuamente esclusivo** se a ogni istante, al massimo un **processo** può accedere a quella **risorsa**.

Vediamo nelle diverse situazioni come interagiscono i processi nella situazione di **condivisione delle risorse**, cioè quando **competono**, quando **cooperano** oppure quando **interferiscono** nelle due situazioni di allocazione **statica** e **dinamica**.

Competizione

La **competizione** è una interazione tra **processi prevedibile** e **NON desiderata**.

Nel caso di risorse **condivise e allocate staticamente** la competizione tra processi avviene al momento dell'accesso alla risorsa: è necessario garantire l'accesso esclusivo (**mutua esclusione**) alla stessa e il compito è del programmatore che, utilizzando le primitive disponibili dal linguaggio di programmazione, scrive le funzioni di accesso in modo che solo un **processo** alla volta possa utilizzare il dato condiviso.

Nel caso di risorse **dedicate e allocate dinamicamente** la responsabilità di gestione è demandata al gestore e la competizione tra processi avviene al momento della richiesta di utilizzo indirizzata al gestore stesso che provvederà a gestirle in **mutua esclusione**.

La soluzione della **competizione** avviene mediante la **sincronizzazione indiretta o implicita**, gestita dal **sistema operativo**.

Cooperazione

La **cooperazione** è una interazione tra **processi prevedibile** e **desiderata** dato che è insita nella logica del programma.

Nel caso di risorse **condivise e allocate staticamente** la **cooperazione** tra processi avviene utilizzando tale risorsa per scambiarsi le informazioni, cioè un **processo** (o più processi) scrive un dato nella risorsa (**produttori**) e un altro **processo** (o più processi) legge successivamente dalla risorsa condivisa (**consumatori**).

Nel caso di risorse **dedicate e allocate dinamicamente** l'unica possibilità di **cooperazione** si ha se una risorsa assegnata a un **processo** viene assegnata a un secondo processo quando il primo termina di utilizzarla: il gestore deve essere al corrente della **cooperazione** in modo da non azzerare il dato "prodotto" dal primo **processo** prima che venga "consumato" dal secondo **processo**.

La soluzione della **cooperazione** avviene mediante **sincronizzazione diretta o esplicita**, gestita dal **sistema del programmatore** mediante **scambio di informazioni**.

Interferenza

L'interferenza è dovuta alla **competizione** che avviene tra **processi** che utilizzano senza le opportune autorizzazioni **risorse condivise** oppure a una erronea soluzione dei problemi di **competizione** e di **cooperazione**.

L'interferenza è una interazione tra **processi NON prevedibile** e **NON desiderata**.

Per esempio possiamo essere in una delle due seguenti situazioni:

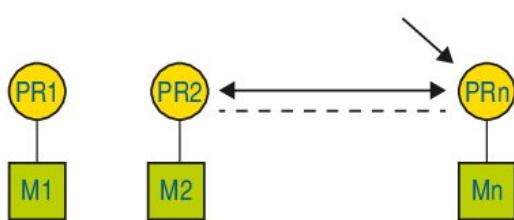
- P1 chiama P2 senza necessità e lo rallenta;
- P1 chiama P2 ma al momento sbagliato.

Generalmente i malfunzionamenti sono legati alla velocità relativa dei **processi** e quindi i risultati sono "dipendenti dal tempo": di conseguenza è molto complesso effettuare il debug.

■ Modello a scambio di messaggi (ambiente locale, message passing)

Nel modello ad **ambiente locale** ogni processo può accedere esclusivamente alle risorse allocate nella propria memoria (**virtuale**) locale che non può essere modificata direttamente dagli altri processi.

Non avendo una memoria condivisa, i processi non possono utilizzare la memoria per il coordinamento delle loro attività e lo strumento di **comunicazione** e **sincronizzazione** diventa lo **scambio di messaggi**.



Il modello a **scambio di messaggi** rappresenta la naturale astrazione di un sistema privo di memoria comune, in cui a ciascun processore è associata una memoria privata.

Il sistema è visto come un insieme di processi, ciascuno operante in un ambiente locale che non è accessibile a nessun altro processo.

Esistono due possibili modalità per implementare questo modello:

- Ⓐ utilizzare linguaggi che prevedono costrutti esplicativi per realizzare lo scambio di messaggi, come il **CSP (Communicating Sequential Processes)** proposto da ▲ **Tony Hoare** ▷;
- Ⓑ utilizzare la "chiamata di procedura remota", come il **DP (Distributed Processes)**, proposto da ▲ **Brinch Hansen** ▷.

▲ **Hoare & Hansen** Hoare is a British computer scientist best known for the development (in 1960, at age 26) of Quicksort. He also developed the formal language Communicating Sequential Processes (CSP) to specify the interactions of concurrent processes (including the dining philosophers problem).



Hansen is a pioneer in the development of operating system principles and parallel programming languages(ex. the parallel programming languages Concurrent Pascal) Dijkstra, Hoare, and Brinch Hansen suggested a parallel programming concept in 1971: the monitor. ▷

È inoltre necessario effettuare una classificazione dei **modelli a scambio di messaggi**: per esempio possiamo distinguere tra comunicazione **sincrona** e **asincrona**:

- Ⓐ nel caso di comunicazione **asincrona** la comunicazione da parte del processo mittente avviene senza che questo rimanga in attesa di una risposta da parte del processo destinatario;

- B nel caso di comunicazione **sincrona** lo scambio di informazioni può avvenire solo se mittente e destinatario sono pronti a “parlarsi” e quindi è necessario che si sincronizzino, e questa interazione prende il nome di ▲ “rendez-vous” ▷:
 ► **stretto**: se si limita alla trasmissione di un messaggio dal mittente al destinatario;
 ► **esteso**: se il destinatario, una volta ricevuto il messaggio, deve inviare una risposta al mittente.

◀ “Rendez-vous” A rendez-vous is a synchronization or meeting point between the task calling another task’s entry and the called task. The first task to the rendez-vous will suspend until another task gets to the same rendezvous. ▷



È possibile effettuare una seconda classificazione dei **modelli a scambio di messaggi**, distinguendo tra comunicazione **asimmetrica** e **simmetrica**:

- A nel caso di comunicazione **asimmetrica** il mittente nomina esplicitamente il destinatario ma questo non nomina esplicitamente il mittente;
 B nel caso di comunicazione **simmetrica** entrambi si nominano in modo esplicito.

Possiamo quindi avere per esempio queste due classiche situazioni:

- comunicazione di tipo **simmetrico** e sincrono a **rendez-vous stretto** tipico del **CSP**;
 ► comunicazione di tipo **asimmetrico** e sincrono a **rendez-vous esteso** tipico del **DP**.

Le applicazioni di queste proposte trovano il loro utilizzo fondamentalmente nei sistemi multiprocessori: un esempio tipico è quello noto come modello **client-server**.

Il meccanismo che viene quindi utilizzato dai processi è quello prima descritto di **scambio di messaggi** e solo tramite questi sono possibili tutti i tipi di interazione tra i processi.

Modello client-server

Ogni risorsa del sistema è accessibile a un solo processo che prende il nome di **processo servitore** (o **server**) e quando un processo deve utilizzarla (**processo cliente**) non può accedervi direttamente ma deve chiedere al processo server di effettuare lui stesso le operazioni desiderate sulla risorsa e di comunicargli successivamente l’esito delle elaborazioni. Definiamo quindi:



SERVITORE/CLIENTE

Servitore: entità computazionale in grado di eseguire una specifica prestazione per altre entità (in grado cioè di offrire un servizio).

Cliente: entità computazionale che richiede a un servitore l’esecuzione di una specifica prestazione.

Il modello **cliente-servitore** è il principio su cui è possibile costruire applicazioni distribuite: tipico esempio è la rete **Internet**, dove un cliente risiede in un nodo e la risorsa è disponibile in un altro; il cliente deve fare la richiesta a una servitorea locale della risorsa per poterla utilizzare, come per esempio per accedere a un database o a un file, e il servente esegue sulla risorsa le richieste comunicando l’esito attraverso messaggi (che generalmente sono pagine **HTML** o trasferimento di file).

AREA digitale

Classificazione di Flynn

Verifichiamo le conoscenze

1. Risposta multipla

1 Il parallelismo dell'esecuzione di processi concorrenti:

- a) è solo virtuale
- b) è solo reale
- c) è reale su sistemi multiprocessing e virtuale su sistemi multiprocessor
- d) è reale su sistemi multiprocessor e virtuale su sistemi multiprocessing
- e) nessuna delle precedenti

2 Nella classificazione di Flynn il personal computer è

- | | |
|----------------------|----------------------|
| a) una macchina SISD | c) una macchina MISD |
| b) una macchina SIMD | d) una macchina MIMD |

3 Quali dei seguenti accoppiamenti è esatto?

- a) modello a memoria comune o ambiente locale
- b) modello a memoria comune o ambiente globale
- c) modello a scambio di messaggi o ambiente locale
- d) modello a scambio di messaggi o ambiente globale

4 I compiti del gestore della risorsa sono (indicare quello errato):

- a) deve mantenere aggiornato lo stato di allocazione della risorsa
- b) deve sospendere i processi che utilizzano la risorsa per molto tempo
- c) deve implementare la strategia di allocazione della risorsa
- d) deve fornire i meccanismi ai processi che hanno il diritto di utilizzare tale risorsa

5 Le risorse allocate staticamente sono:

- | | |
|------------------------|------------------------|
| a) sempre private | c) sempre comuni |
| b) private se dedicate | d) comuni se condivise |

6 Le risorse allocate dinamicamente sono:

- | | |
|------------------------|------------------------|
| a) sempre private | c) sempre comuni |
| b) private se dedicate | d) comuni se condivise |

7 Nel modello a scambio di messaggi le risorse comuni:

- | | |
|--------------------------------|--|
| a) sono allocate staticamente | c) non ci sono risorse condivise |
| b) sono allocate dinamicamente | d) sono allocate in maniera dipendente dall'architettura |

8 Quali tra i seguenti sono tipi di comunicazione nei modelli a scambio di messaggi?

- | | |
|-----------------------------------|-----------------------------------|
| a) sincrono a rendez-vous stretto | c) asincrono a rendez-vous esteso |
| b) sincrono a rendez-vous stretto | d) asincrono a rendez-vous esteso |

2. Vero o falso

- 1 L'acronimo SIMD indica Single Instruction Multiprocessor Data.** V F
- 2 È comodo condividere memoria nei sistemi di elaborazione distribuiti.** V F
- 3 Il sistema operativo associa a ogni risorsa un apposito gestore di risorsa.** V F
- 4 Si può considerare risorsa anche una istanza di una struttura dati allocata nella memoria comune.** V F
- 5 Il gestore definisce istante per istante i processi che hanno il diritto di operare su di essa.** V F

- 6** Il gestore delle risorse allocate staticamente è il programmatore. V F
- 7** Gli accessi a una risorsa condivisa possono avvenire in modo non divisibile. V F
- 8** L'accesso si dice mutualmente esclusivo se solo un processo può accedere a quella risorsa. V F
- 9** Il caso più complesso di gestione è quello di risorse condivise e allocate dinamicamente. V F
- 10** Il linguaggio come il CSP proposto da Hansen prevede costrutti esplicativi per realizzare lo scambio di messaggi. V F
- 11** Nella comunicazione simmetrica il mittente nomina esplicitamente il destinatario e viceversa. V F

3. Completamento

multitask • interazione concorrente • scambio di messaggi • SISM • memoria comune • architettura distribuita • interazione • processi • risorse • sincronizzazione • risorsa condivisa • non divisibile • sezioni critiche • esteso • allocate staticamente • mutualmente esclusivo • stretto • sincronizzino • condivise • condivise • mutua esclusione • senza • allocate staticamente • rendez-vous • sincrona • asincrona

- 1** Possiamo individuare due modelli di a prescindere dalla soluzione hardware, cioè sia che stiamo analizzando una oppure una situazione di su macchina ;
 - modello a
 - modello a
- 2** Entrambi i modelli si basano sul concetto di tra i due elementi che costituiscono il sistema: i interagiscono per entrare in possesso (utilizzare) delle (oggetti).
- 3** Gli accessi a una devono avvenire in modo: le funzioni che utilizzano un dato condiviso (.....) devono essere programmate utilizzando i meccanismi di offerti dal linguaggio di programmazione e supportati dalla macchina concorrente.
- 4** L'accesso a una risorsa si dice se a ogni istante, al massimo un processo può accedere a quella risorsa.
- 5** Nel caso di risorse e la competizione tra processi avviene al momento dell'accesso alla risorsa: è necessario garantire l'accesso esclusivo (.....).
- 6** Nel caso di risorse e la cooperazione tra processi avviene utilizzando tale risorsa per scambiarsi le informazioni
- 7** Nel caso di comunicazione la comunicazione da parte del processo mittente avviene che questo rimanga in attesa di una risposta da parte del processo destinatario;
- 8** Nel caso di comunicazione lo scambio di informazioni può avvenire solo se mittente e destinatario sono pronti a "parlarsi" e quindi è necessario che si , e questa interazione prende il nome di "":
 -: se si limita alla trasmissione di un messaggio dal mittente al destinatario;
 -: se il destinatario, una volta ricevuto il messaggio, deve inviare una risposta al mittente.

La sincronizzazione tra processi

In questa lezione impareremo...

- ▶ le tipologie di errori nei processi paralleli
- ▶ le motivazioni della sincronizzazione
- ▶ le proprietà richieste ai programmi concorrenti

■ Errori nei programmi concorrenti

La **programmazione concorrente** nasconde maggiori insidie della normale programmazione monoprogrammata in quanto introduce la possibilità di commettere **errori dipendenti dal tempo**, nei confronti dei quali le normali tecniche di debugging non sono efficaci dato che, oltre alla **correttezza logica**, ai programmi è anche richiesta la **correttezza temporale**.

È molto diverso effettuare il testing di un **programma concorrente** rispetto a uno sequenziale:

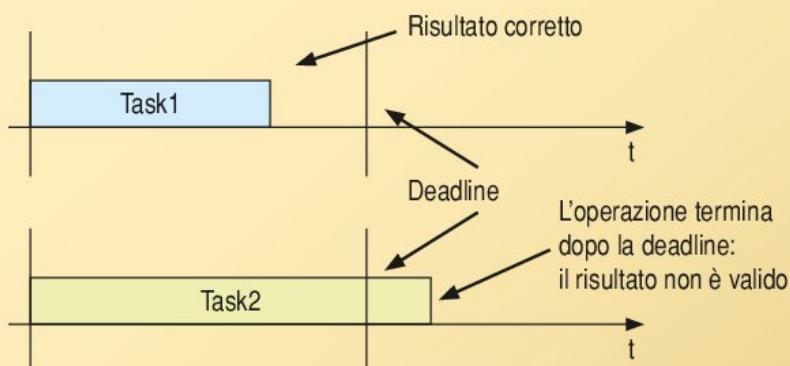
- ▶ il **programma sequenziale** produce diversi risultati ma solo in funzione di dati di input diversi e il programmatore, mediante casi di prova, può verificare i comportamenti del programma confrontando gli output con i risultati attesi: non è possibile in questo modo dimostrare la correttezza totale del programma, ma se viene accuratamente scelto l'insieme dei dati di prova possiamo avere buone garanzie sulla correttezza del nostro lavoro;
- ▶ nei **programmi concorrenti**, oltre agli errori sequenziali eliminabili come prima descritto, sono possibili gli errori legati ai tempi di esecuzione e di **schedulazione nella CPU** che non si possono determinare ed eliminare tramite testing ma devono essere evitati tramite una programmazione particolarmente accurata, individuando “dove il codice” potrebbe essere causa di possibili situazioni di errore, che quasi sempre è connesso con la condivisione e comunicazione di dati tra **processi concorrenti**.

La soluzione estrema che permette di eliminare i problemi dipendenti dal tempo è quella di introdurre ritardi nelle elaborazioni tali che possano escludere la generazione di problemi dovuti alla loro interazione; tuttavia questa non è una soluzione, è un “artificio non accettabile” oltre che rischioso in quanto non offre nessuna certezza che renda l'esecuzione corretta.

La maggior parte dei **sistemi concorrenti** ha inoltre la necessità di sfruttare al massimo le potenzialità del sistema di calcolo e richiede di conciliare l'affidabilità del software all'esigenza di massimizzare l'efficienza di elaborazione, soprattutto nei sistemi a **hard real time**, dove è assolutamente indispensabile il rispetto delle **deadlines temporali** (**timing constraint**).

Abbiamo due vincoli da soddisfare indicati generalmente come "correttezza temporale":

- A determinismo:** i risultati devono essere uguali per ogni esecuzione, indipendentemente dalla sequenza di schedulazione;
- B timing constraint:** i risultati devono essere prodotti entro certi limiti temporali fissati (**deadlines**) (specifica dei sistemi **real time**).



◀ **Deadlines temporali** È l'istante temporale entro cui il processo deve terminare la propria esecuzione e produrre un risultato. ►



◀ **Hard real time** A hard real-time system (also known as an immediate real-time system) is hardware or software that must operate within the confines of a stringent deadline. Examples of hard real-time systems include components of pacemakers, anti-lock brakes and aircraft control systems. ►

Gli **errori dipendenti dal tempo** sono causati da una scorretta **sincronizzazione dei processi** e costituiscono una particolare categoria di errori: si verificano in corrispondenza a determinate velocità relative dei processi e non si riproducono quindi necessariamente riavviando il sistema con le stesse condizioni iniziali.

Riassumiamo le caratteristiche degli errori dipendenti dal tempo:

- ▶ **irriplicabili:** possono verificarsi con alcune sequenze e non con altre;
- ▶ **indeterminati:** esito ed effetti dipendono dalla sequenza;
- ▶ **latenti:** possono presentarsi solo con sequenze rare;
- ▶ **dificili da verificare e testare:** perché le tecniche di verifica e testing si basano sulla riproducibilità del comportamento.

È quindi di fondamentale importanza la scelta di tecniche corrette di sincronizzazione.

AREA digitale
Applicazioni in real time

Se una **risorsa** è allocata come **dedicata** non è necessaria la sincronizzazione mentre se una risorsa è condivisa è necessario assicurare che gli accessi avvengano in modo **non divisibile**, cioè che le operazioni che un processo deve effettuare sulla risorsa, per esempio l'aggiornamento di un dato, non vengano interrotte neppure dallo scheduler ma si possa garantire l'accesso in **mutua esclusione** finché il processo stesso non decide di rilasciarla al termine del suo utilizzo in modo da rendere disponibile il risultato dell'elaborazione quando questo è significativo. L'insieme delle operazioni che devono essere ininterrompibili devono essere programmate come **sezioni critiche** utilizzando i meccanismi di sincronizzazione offerti dal linguaggio di programmazione.

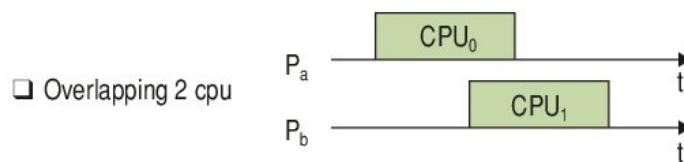
■ Definizioni e proprietà

Prima di procedere con lo studio dei meccanismi che permettono di realizzare la **mutua esclusione** è necessario introdurre alcune definizioni e rivedere quelle incontrate sino a questo punto della trattazione per completarne la formalizzazione.

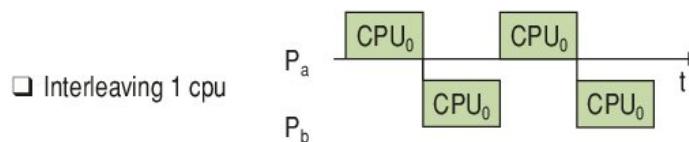
Interleaving e overlapping

Innanzitutto per avere la **concorrenza** è necessario che due programmi siano eseguiti in parallelo, che può essere *parallelismo reale*, nel caso di più processori, o apparente, in macchine multiprogrammate monoprocessoressi.

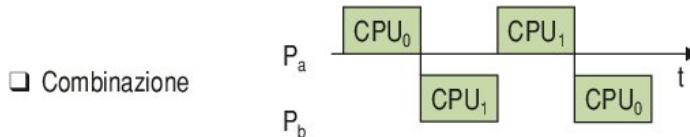
- 1 Nei **sistemi multiprocessore** più processi vengono eseguiti simultaneamente su **processori** diversi e quindi sono “sovrapposti nel tempo”: in questo caso si definisce **overlapping** la **sovraposizione** temporale di processi.



- 2 Nella macchina con un **singolo processore** i processi sono “alternati nel tempo” ma con velocità tali da dare l'impressione di avere un multiprocessore: in questo caso si definisce **interleaving** la situazione di **alternanza casuale** che possono avere i processi a causa delle diverse modalità di **schedulazione** (che può portare a errori dipendenti dal tempo).



- 3 Potrebbe anche esserci una situazione in cui sono presenti **più macchine**, ciascuna **multitasking** e quindi avere una **combinazione delle due situazioni** sopra descritte di **interleaving/overlapping**.



Il caso 1 implica che ogni **azione** sia svolta da un distinto esecutore fisico (**processore**) ed equivale a un **PARALLELISMO REALE**:

p processori per $a=p$ azioni

Negli altri due casi siamo in situazioni in cui si dispone di un **numero p** di processori inferiore al **numero a** di **azioni** da eseguire ed equivale a un quasi-parallelismo o a un **PARALLELISMO VIRTUALE**:

p processori per $a>p$ azioni

Condizioni di Bernstein

Scrivere quindi un programma concorrente non è facile: abbiamo visto come è possibile rappresentare un programma sequenziale in termini di precedenze ma non sappiamo che caratteristiche devono avere due operazioni che possono essere eseguite in parallelo senza che generino **errori dipendenti dal tempo**.

I vincoli che devono soddisfare due istruzioni per essere eseguite concorrentemente sono le **condizioni di Bernstein**.

Prima di elencarle è necessario introdurre il concetto di **dominio** e di **rango** di una procedura, che si ottiene dall'unione dei **domini** e **ranghi** delle singole istruzioni.



DOMINIO E RANGO DI UNA ISTRUZIONE O PROCEDURA

Indicando A, B, ..., X, Y, ... una variabile o, più generalmente, un'area di memoria, una istruzione K:

- si ottiene da una o più aree di memoria, che indichiamo come **domain(K)** (dominio di K);
- modifica il contenuto di una o più aree di memoria, che indichiamo con **range(K)** (rango di K).

ESEMPIO **Dominio e rango**

La seguente procedura P:

```
procedura P
inizio
  X ← A - X;
  Y ← A * B;
fine
```

utilizza tre variabili, A, B e Y, quindi si ha **domain(P) = {A, B, X}**
modifica due variabili, X e Y, quindi si ha **range(P) = {X, Y}**

Possiamo ora definire le:



CONDIZIONI DI BERNSTEIN

Due istruzioni i_a e i_b possono essere eseguite concorrentemente se valgono le seguenti condizioni, dette **condizioni di Bernstein**:

- $\text{range}(\text{istruzione A}) \cap \text{range}(\text{istruzione B}) = \emptyset$
- $\text{range}(\text{istruzione A}) \cap \text{domain}(\text{istruzione B}) = \emptyset$
- $\text{domain}(\text{istruzione A}) \cap \text{range}(\text{istruzione B}) = \emptyset$

Non si impone alcuna condizione sulla intersezione dei domini delle due istruzioni.

ESEMPIO Verifica delle condizioni di Bernstein

Vediamo tre semplici esempi dove analizziamo due istruzioni alla volta:

1	Istruzione	Domain(istruzione)	Range(istruzione)
A	$X \leftarrow Y + 5;$	Y	X
B	$X \leftarrow Y - 3;$	Y	X

violano la condizione 1:

- $\text{range}(A) \cap \text{range}(B) = X$ che è diverso dall'insieme vuoto \emptyset

2	Istruzione	Domain(istruzione)	Range(istruzione)
A	$X \leftarrow Y + 2;$	Y	X
B	$Y \leftarrow X - 1;$	X	Y

violano la condizione 2 e la condizione 3:

- $\text{range}(\text{istruzione A}) \cap \text{domain}(\text{istruzione B}) = X$ che è diverso dall'insieme vuoto \emptyset
- $\text{domain}(\text{istruzione A}) \cap \text{range}(\text{istruzione B}) = Y$ che è diverso dall'insieme vuoto \emptyset

3	Istruzione	Domain(istruzione)	Range(istruzione)
A	scrivi (X)	X	\emptyset
B	$X \leftarrow X + Y + 3;$	X, Y	X

violano la condizione 3:

- $\text{domain}(\text{istruzione A}) \cap \text{range}(\text{istruzione B}) = X$ che è diverso dall'insieme vuoto \emptyset

Se due (o più) istruzioni soddisfano le **condizioni di Bernstein** il risultato è indipendente dalla particolare sequenza di esecuzione eseguita dai processori (interleaving) e sarà quindi identico alla loro esecuzione seriale.

Quando anche una sola condizione viene violata si ottengono errori generati dal tempo dovuti al fenomeno dell'**interferenza**.

ESEMPIO **Errore dovuto all'interleaving**

Un programma di magazzino utilizza due funzioni costituite da due istruzioni per aggiornare il totale di un prodotto a seconda che venga comprato o venga venduto.

```
procedura Scarica(x)
inizial
    TotS ← TANTI - x;
    TANTI ← TotS;
fine
```

```
procedura Carica(y)
inizial
    TotC ← TANTI + Y;
    TANTI ← TotC;
fine
```

Individuiamo per ogni coppia di istruzioni il *domain* e il *range* per verificare le **condizioni di Bernstein**:

domain(Scarica)= {TANTI, X, TotS}, range(Scarica)= {TANTI, TotS}
domain(Carica)= {TANTI, Y, TotC}, range(Carica)= {TANTI, TotC}

È immediato osservare che sono violate tutte e tre le condizioni e quindi molto probabilmente avremo errori dipendenti dal tempo.

Come verifica, supponiamo che contemporaneamente vengano vendute 3 unità e prodotte 5 unità a partire da una giacenza iniziale di 10 pezzi.

Analizziamo tre possibili **sequenze di interleaving**:

TANTI=10 TotS ← TANTI - 3; TotC ← TANTI + 5; TANTI ← TotC; TANTI ← TotS; (TANTI =7)	TANTI=10 TotS ← TANTI - 3; TANTI ← TotC; TotC ← TANTI + 5; TANTI ← TotS; (TANTI =12) ESECUZIONE SEQUENZIALE	TANTI=10 TotC ← TANTI + 5; TotS ← TANTI - 3 TANTI ← TotS; TANTI ← TotC; (TANTI =15)
--	--	--

Solo nella seconda sequenza sono rispettati i vincoli temporali ed è quindi l'unica sequenza che genera il risultato esatto.

Nella seconda soluzione i due segmenti di codice sono stati eseguiti in sequenza, cioè non sono stati interrotti: quindi l'esecuzione in mutua esclusione delle istruzioni che modificano variabili condivise deve essere eseguita in modo tale da non essere interrotta (istruzioni **atomiche**).

Mutua esclusione e sezione critica

Consideriamo due processi in competizione per l'uso esclusivo di una **risorsa comune**: non è prevedibile sapere l'istante di tempo nel quale uno di essi utilizzerà la **risorsa**, ma **bisogna** garantire che quando ne entrerà in possesso lo farà in modo **esclusivo**, cioè la risorsa verrà utilizzata da “un processo alla volta” che la rilascerà al temine delle operazioni che la coinvolgono.



MUTUA ESCLUSIONE

Si ha mutua esclusione quando non più di un processo alla volta può accedere a una **risorsa comune**.

La regola di **mutua esclusione** impone che le operazioni con le quali i **processi** accedono alle variabili comuni **non si sovrappongano nel tempo**: inoltre **nessun vincolo è imposto sull'ordine** con il quale le operazioni sulle variabili vengono eseguite.



SEZIONE CRITICA

La sequenza di istruzioni con la quale un processo accede e modifica un insieme di variabili condivise prende il nome di **sezione critica**.

Nei nostri programmi le **risorse comuni** condivise saranno le **variabili globali** che verranno utilizzate dai diversi processi per scambiarsi le informazioni (**modello a memoria condivisa**).

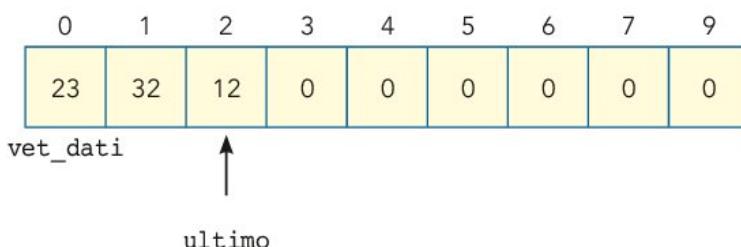
Vediamo un esempio nel quale due processi si scambiano dati attraverso la **memoria condivisa** per comprendere meglio perché è di fondamentale importanza regolare l'accesso a tale risorsa in modo **esclusivo**.

ESEMPIO

Errore causato dalla interrompibilità delle operazioni

Due processi P1 e P2 condividono una regione di memoria dove il produttore genera un numero e lo inserisce nella prima posizione libera di un vettore **vet_dati[x]** e il consumatore legge l'ultimo dato inserito azzerando il contenuto della cella.

Oltre che il vettore i processi condividono anche la variabile indice **ultimo** che contiene il valore dell'ultima cella occupata:



Riportiamo un segmento di codice per ciascuno di essi:

Produttore	Consumatore
<pre>funzione inserisci(nuovo_dato) inizio ultimo ← ultimo+1 vet_dati[ultimo] ← nuovo_dato fine</pre>	<pre>funzione preleva() inizio letto r vet_dati[ultimo] vet_dati[ultimo]≤ 0 ultimo r ultimo -1 fine</pre>

Entrambi i segmenti di codice accedono alle variabili condivise e modificano sia il contenuto del vettore che dell'indice: una esecuzione contemporanea potrebbe portare a situazioni errate, come nel caso seguente.

A causa dello scadere del **time slice** un processore potrebbe eseguire nel tempo le istruzioni in questa sequenza:

```
t0 Prod ultimo ← ultimo+1
t1 Cons letto ← vet_dati[ultimo]
t2 Cons ultimo ← ultimo -1
t3 Cons vet_dati[ultimo] ← 0
t4 Pros vet_dati[ultimo] ← nuovo_dato
```

Il consumatore andrebbe a prelevare un dato non ancora prodotto (quindi un valore uguale a zero) e aggiornerebbe l'indice facendo in modo che il produttore “sovrapponga” il nuovo dato a uno preesistente, non ancora prelevato.

Nel nostro esempio le **sezioni critiche** sono associate alle istruzioni che accedono alle variabili condivise **vet_dati[x]** e **ultimo** presenti nelle due funzioni descritte, **inserisci()** e **preleva()**.

La regola di **mutua esclusione** stabilisce che in ogni istante una risorsa o è libera oppure è assegnata a uno e un solo processo: in particolare per avere la **mutua esclusione** devono essere soddisfatte le seguenti **quattro condizioni**:

- ▷ nessuna coppia di **processi** può trovarsi simultaneamente nella **sezione critica**;
- ▷ l'accesso alla **regione critica** non deve essere regolato da alcuna assunzione temporale o dal numero di **CPU**;
- ▷ nessun **processo** che sta eseguendo codice al di fuori della **regione critica** può bloccare un **processo** interessato a entrarvi;
- ▷ nessun **processo** deve attendere indefinitamente per poter accedere alla **regione critica**.

Starvation e deadlock

Un **errata sincronizzazione** può portare al fallimento delle elaborazioni, genera situazioni di incoerenza dei dati, e può portare a situazioni di **blocco dei processi**:

- ▷ **starvation** (o **blocco individuale**): si verifica quando un processo rimane in attesa di un evento che non accadrà mai, e quindi non può portare a termine il proprio lavoro,
- ▷ **deadlock** (**stallo** o **blocco multiplo**): si verifica quando due o più **processi** rimangono in attesa di eventi che non potranno mai verificarsi a causa di condizioni cicliche nel possesso e nella richiesta di risorse.

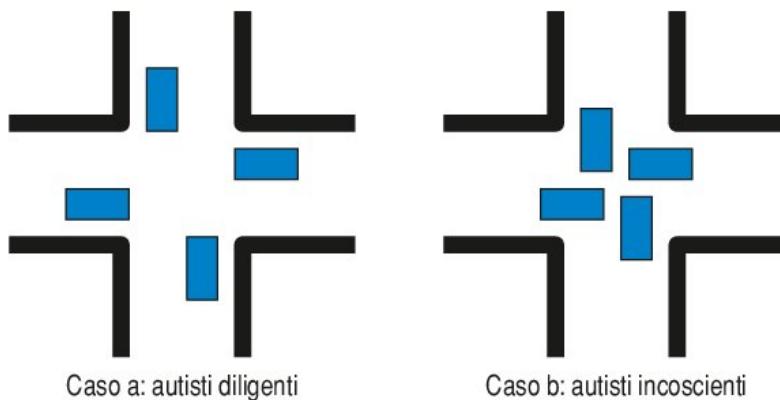
Vediamo un terzo esempio dove la mutua esclusione permette di risolvere il problema della interferenza ma può causare il blocco permanente dei processi.

ESEMPIO **Blocco critico o deadlock**

Supponiamo di avere un incrocio con quattro strade, senza che questo venga regolato dalla presenza di un vigile e neppure di un semaforo.

La situazione di blocco si può verificare se contemporaneamente un automezzo giunge da ogni strada all'incrocio e tutti gli autisti si comportano allo stesso modo:

- A** danno la precedenza a destra, come previsto dal regolamento della strada;
- B** si apprestano ad attraversare l'incrocio senza curarsi degli altri automezzi.



In entrambe le situazioni si verifica un **deadlock** in quanto tutti gli automezzi rimangono bloccati senza possibilità di soluzione, cioè si ostacolano a vicenda creando la "morte" contemporanea e collettiva.

Diverso invece il caso nel quale una risorsa è occupata per un solo processo in modo continuativo: è il caso che capita quando siete in coda a uno sportello e continuano ad arrivare "furbi" che passano davanti, impedendovi di avanzare verso l'impiegato: in questo caso si tratta di **starvation**.

Possiamo riassumere in una tabella le diverse situazioni di interazione a seconda della natura dei processi:

Tipo	Relazione	Meccanismo	Problemi di controllo
Processi "ignari" uno dell'altro	Competizione	Sincronizzazione	Mutua esclusione Deadlock Starvation
Processi con conoscenza indiretta l'uno dell'altro	Cooperazione (sharing)	Sincronizzazione	Mutua esclusione Deadlock Starvation
Processi con conoscenza diretta l'uno dell'altro	Cooperazione (comunicazione)	Comunicazione	Deadlock Starvation

Esistono molte soluzioni a questo problema, a seconda della **politica** di accesso alla **risorsa** che viene scelta, che può favorire nell'accesso una classe di **processi** oppure un'altra.

■ Proprietà non funzionali: safety e liveness

Nella soluzione dei problemi con processi interagenti non è quindi sufficiente programmare in modo da soddisfare solamente gli **aspetti funzionali** per affermare che un programma è **corretto**: per esempio, non basta verificare che i calcoli diano risultati esatti è che l'algoritmo sia giusto per essere sicuri che sempre "tutto funzioni".

È necessario tener presente anche degli aspetti **NON funzionali** che garantiscono la correttezza in ogni situazione di schedulazione e di interazione, cioè per ogni possibile "storia di esecuzione" del programma stesso: tra queste ricordiamo il rispetto del tempo di esecuzione di un'applicazione in un sistema real-time, la sicurezza di un'applicazione, ma anche l'assenza di **deadlock** e di **starvation**.

Premessa: il **sistema operativo** deve garantire la proprietà di **fairness**, cioè deve sempre mandare in esecuzione qualsiasi **processo** soddisfacendo prima o poi tutte le richieste di esecuzione così da non essere lui la causa di eventuali situazioni di **starvation**.

Le **proprietà non funzionali** fondamentali che un **programma concorrente** deve soddisfare sono due che descriviamo qui di seguito: la prima riguarda le **risorse (safety)** mentre la seconda riguarda i **processi (liveness)**.

Safety

La **safety** è una proprietà che riguarda lo **stato delle risorse**: con questa proprietà si intende una condizione che deve sempre verificarsi per il buon fine dell'esecuzione del **processo** oppure una condizione di pericolo che non si deve mai verificare per la sicurezza del sistema. In altre parole, se un sistema avanza, questo va "nella direzione voluta" senza eseguire azioni indesiderate, cioè "**nothing bad ever happens**"!

I **processi** non devono "interferire" fra di loro nell'accesso alle **risorse condivise** e i meccanismi di **sincronizzazione** servono a garantire la proprietà di **safety** in modo che tutte le risorse del sistema siano sempre in uno **stato consistente**: si devono eliminare le **interferenze indesiderate** tra i **processi**.

Tra le molte possibili cause di errore e di interferenza tra processi che possono violare la **safety** ricordiamo:

- ▶ la presenza di corse critiche causate da una non corretta **mutua esclusione**;
- ▶ l'accesso e le azioni su stati di **risorse non consistenti**;
- ▶ la violazioni dell'**atomicità** di certe operazioni (**lock** e **P()** su semafori);
- ▶ esecuzione di azioni che dovrebbe essere proibite ma permesse dal linguaggio di programmazione utilizzato per scrivere i programmi;
- ▶ esecuzione di operazioni su valori e dati non aggiornati, mantenuti in cache e quindi non consistenti.

Molti linguaggi di programmazione sequenziale sono **type-safe** perché il compilatore controlla il corretto uso dei tipi di dato, ma questo controllo viene fatto staticamente, a **compiled-time**: i sistemi **multi-threaded** introducono la **dimensione del tempo** e quindi non è sufficiente il controllo statico ma si devono quindi utilizzare tecniche di programmazione che preservino la consistenza degli oggetti evitando interferenze a **run-time**.

Liveness

La **liveness** è una proprietà che riguarda le attività, cioè i **processi**, in merito alla loro “esecuzione ed evoluzione”: si deve garantire che il processo che avanza porterà a termine in modo corretto il proprio lavoro, cioè “**something good eventually happens**”. A tal fine i meccanismi di sincronizzazione devono fare in modo che un processo non aspetti “indeterminatamente” che una **risorsa** venga rilasciata oppure che **tutti** i processi si “**blocchino**” in attesa di eventi che non possono verificarsi: è la proprietà che esclude situazioni di **deadlock**.

La **liveness** garantisce che prima o poi **tutti i processi** entrano in uno stato corretto e progradiscono verso il completamento, cioè si deve assicurare la corretta **cooperazione** tra i **processi** evitando situazioni di **deadlock** e di **starvation**.

Tra le molte possibili cause di errore e **interferenza** tra **processi** che possono violare la **liveness** di una applicazione possono essere:

- ▷ errori di **sincronizzazione**;
- ▷ errori con i segnali tra **processi cooperanti**;
- ▷ fallimenti di un **processo**: si attende un segnale da un altro **processo** che è andato in crash;
- ▷ **livelock**: continuo tentativo di una azione che fallirà sempre;
- ▷ **lockout**: chiamata di una operazione che non sarà mai disponibile;
- ▷ esaurimento di una risorsa necessaria per evolvere.

Ciascuna di queste situazioni può portare o alla **starvation** di un **processo** oppure addirittura al **deadlock** di tutto il sistema.

Ricapitolando, in caso di violazione delle proprietà sopra descritte si generano problemi di:

- ▷ **violazione fairness**: si ha l’assegnazione di risorse in modo non equo a tutti processi;
- ▷ **violazione liveness**:
 - **starvation**: una classe di processi non entra mai in possesso della risorsa;
 - **deadlock**: blocco di tutti i processi che si ostacolano a vicenda.

■ Conclusioni

È sicuramente più complesso scrivere **programmi concorrenti** rispetto ai **programmi sequenziali** in quanto non basta essere sicuri della correttezza dei singoli moduli ma è necessario garantire il loro corretto funzionamento per ogni possibile combinazione di interazioni (volute o indesiderate) che questi possono avere.

Le **condizioni di Bernstein** ci permettono di individuare nel nostro codice dove possono verificarsi problemi dipendenti dal tempo, ma non sempre è semplice garantire la **mutua esclusione** e soprattutto effettuare il test e il debug di applicazioni che presentano **race condition**. Si noti inoltre che **safety** e **liveness** sono proprietà “in contrasto” tra loro: la realizzazione di una corretta applicazione concorrente deve bilanciare le diverse esigenze di progettazione e introdurre sincronizzazioni troppo forti che possono ridurre la concorrenza di esecuzione impattando sulle prestazioni (per esempio, l’uso di **attese attive** su eventi che devono accadere o in attesa di segnali da parte di altri processi comporta spreco di risorse, soprattutto di tempo CPU).

Nelle prossime lezioni descriveremo gli strumenti e le tecniche per scrivere programmi paralleli e garantire l’assenza di situazioni di **starvation** o di **deadlock**.

AREA digitale

 Esempio riepilogativo
(non informatico)

Verifichiamo le conoscenze



1. Risposta multipla

- 1** Come "correttezza temporale" si intendono due vincoli da soddisfare:
 - a) determinismo
 - b) sincronizzazione
 - c) timing constraint
 - d) cooperazione
- 2** Indica quale non è una caratteristica degli errori dipendenti dal tempo:
 - a) indeterminati
 - b) irriplicabili
 - c) inconsistenti
 - d) latenti
- 3** Quale tra le seguenti non è una condizione di Bernstein?
 - a) range(istruzione A) > range(istruzione B) = \emptyset
 - b) range(istruzione A) > domain(istruzione B) = \emptyset
 - c) domain(istruzione A) > range(istruzione B) = \emptyset
 - d) domain(istruzione A) \cap domain(istruzione B) = \emptyset
- 4** Una errata sincronizzazione può portare:
 - a) all'interleaving
 - b) all'overlapping
 - c) alla starvation
 - d) al deadlock



2. Vero o falso

- 1** La programmazione concorrente introduce errori dipendenti dal tempo. V F
- 2** La velocità della CPU è una delle cause degli errori dipendenti dal tempo. V F
- 3** L'introduzione di ritardi risolve il problema degli errori dipendenti dal tempo. V F
- 4** Nei sistemi real time i risultati devono essere prodotti entro la deadlines. V F
- 5** Una risorsa allocata come risorsa dedicata necessita di sincronizzazione iniziale. V F
- 6** Nei sistemi hard real time siamo in una situazione di real time stretto. V F
- 7** L'interleaving si genera da una errata schedulazione dei processi. V F
- 8** L'overlapping si genera da una errata schedulazione temporale dei processi. V F
- 9** La violazione di una condizione di Bernstein crea l'interferenza e problemi legati al tempo. V F
- 10** La proprietà di liveness esclude situazioni di deadlock. V F

3. Risposte aperte

- 1** Dai una definizione di mutua esclusione.
- 2** In che cosa consiste una regione critica?
- 3** In che cosa consiste l'interleaving?
- 4** In che cosa consiste l'overlapping?
- 5** Descrivi le condizioni di Bernstein.
- 6** In che cosa consiste la starvation?
- 7** In che cosa consiste il deadlock?
- 8** In che cosa consiste la proprietà di safety?
- 9** In che cosa consiste la proprietà di fairness?
- 10** In che cosa consiste la proprietà liveness?

Sincronizzazione tra processi: semafori

In questa lezione impareremo...

- ▶ a definire e utilizzare i semafori di basso livello e spin lock()
- ▶ il concetto di indivisibilità di una primitiva
- ▶ il funzionamento dei semafori di Dijkstra

■ Premessa: quando è necessario sincronizzare?

Nel caso di **processi** interagenti, siano essi in **competizione**, cioè che chiedono l'uso di una risorsa comune riusabile e di molteplicità finita per i **propri scopi**, oppure siano in **cooperazione** per raggiungere un obiettivo comune, possono verificarsi **casi di interferenza**.

La strategia da adottare per gestire l'**interferenza** è diversa per ogni situazione e dipende dalla tollerabilità che il sistema può permettersi degli effetti di una errata **sincronizzazione**, dalla possibilità di individuare agevolmente le eventuali situazioni e di poterle correggere ripristinando le situazioni precedenti ed eventualmente ripetendo le **operazioni critiche**.

Possiamo classificare in quattro gruppi la casistica di situazioni possibili e per ogni gruppo indicare l'azione che è necessario effettuare:

Conseguenze	Esempio	Strategie
inaccettabili	incrocio stradale	evitare ogni interferenza
trascurabili	applicazioni non critiche	ignorare
rilevabili e controllabili	iteratori	rilevare ed evitare
rilevabili e recuperabili	rete ethernet	rilevare e ripetere

Noi ci occuperemo sia del primo caso, cioè di situazioni in cui la **competizione** tra i processi ci porta a **interferenze** che possono provocare situazioni inaccettabili e che quindi devono

essere gestite con la **mutua esclusione**, sia del caso in cui i processi collaborano e quindi devono scambiarsi informazioni attraverso aree di memoria condivisa e con accesso regolato da meccanismi di **sincronizzazione**.

Il **frammento di programma** che utilizza la **risorsa R** che deve essere gestita in **mutua esclusione** si dice **sezione critica** o **regione critica** rispetto alla **risorsa R**: è necessario garantire che un **processo** acceda alla risorsa da solo, cioè che quindi esegua la **sezione critica** con la certezza di essere l'unico utilizzatore che volta per volta esegue il codice che utilizza la **risorsa**.

La **sezione critica** viene gestita in modo che:

- A** un **processo** che deve accedere a una **regione critica** deve **chiedere l'autorizzazione** eseguendo una serie di istruzioni che, nel caso la risorsa fosse libera, gli garantiscono il suo utilizzo esclusivo per tutta la durata della sua elaborazione; se la **risorsa** fosse occupata il gestore ne impedisce l'accesso gestendo la richiesta, per esempio, con una coda di attesa;
- B** quando un **processo** termina di utilizzare una **risorsa**, ha quindi terminato l'esecuzione delle istruzioni della **sezione critica**, deve effettuare un insieme di **operazioni per rilasciarla** in modo che possa essere utilizzata dagli altri processi.

Mediante l'utilizzo di **primitive** che regolino l'accesso e il rilascio della **risorsa** è necessario garantire che:

- A** la **risorsa** o è libera oppure è utilizzata da un solo processo (condizione di **mutua esclusione**);
- B** i **processi** devono sempre poter accedere alla **risorsa** richiesta e portare a termine il proprio lavoro (**condizione di fairness**);
- C** i **processi** non devono avere cicli di ritardo non necessari che possano rallentare l'accesso alla **regione critica** da parte di un altro **processo**.

Dobbiamo scrivere i nostri programmi in modo da garantire la **serializzazione** dell'uso della **risorsa** e l'utilizzo della stessa per un tempo finito, in modo che tutti i **processi** che ne hanno bisogno prima o poi la possano utilizzare.

I meccanismi che permettono di regolare l'accesso alla **regione critica** risolvendo di fatto il problema della **mutua esclusione** sono essenzialmente tre:

- **gli spin lock** (o **semafori binari**);
- **P(S) & V(S)** ovvero i **semafori di Dijkstra**;
- **i monitor**.

In questa lezione affronteremo il caso di richiesta di **risorsa** singola da parte di soli due **processi** ma il concetto è immediatamente estendibile a un numero qualsivoglia di **processi**: più complesso è invece il caso di risorse multiple richieste contemporaneamente dagli stessi **processi**, che discuteremo nella prossima lezione.

■ Semafori di basso livello e spin lock()

Il primo meccanismo che analizziamo è quello che associa a ogni risorsa una **variabile x** che in base al suo valore assume il seguente significato:

- **x = 1** **risorsa libera**, cioè nessun **processo** la sta utilizzando;
- **x = 0** **risorsa occupata** da un **processo**.

Quindi il **flag** fa la funzione di un **semaforo**:

- **x = 1** semaforo **verde**, è possibile accedere alla **risorsa**;
- **x = 0** semaforo **rosso**, la **risorsa** è occupata ed è necessario mettersi in attesa che si liberi.

La variabile **x semaforo** può assumere solamente il valore 0 oppure 1: l'implementazione di questi **semafori** può essere realizzata con **variabili booleane** che prendono il nome di **spin lock**.

Vediamo come esprimere in pseudocodifica i segmenti di codice che effettuano rispettivamente l'**allocazione** e il **rilascio** di una **risorsa**.

Allocazione di una risorsa: **lock()**

La **primitiva** (o funzione) che permette di allocare una **risorsa** prende il nome di **lock()**. Possiamo indicare la sua sintassi nel seguente formato:

```
lock(x);
```

dove **x** è il semaforo associato alla risorsa che desideriamo utilizzare.

La **primitiva lock()** deve:

- testare il **semaforo** per verificarne il suo colore;
- se è **verde**, modificarne il valore a **rosso**;
- se è **rosso**, aspettare che diventi **verde** per poi metterlo a **rosso**.

In pseudocodifica una possibile realizzazione è la seguente:

```
funzione lock(x)
inizia
    ripeti           // ciclo di attesa sul semaforo nel caso che sia rosso
        finche x=1   // esci dal ciclo a semaforo verde
        x ← 0;         // metti il semaforo a rosso
    fine
```

L'osservazione che possiamo fare a questa funzione è che il ciclo di attesa viene eseguito ripetendo continuamente il test sul **semaforo** fino a quando diventa verde (**x=1**): durante la fase di attesa un **processo** "consuma inutilmente **CPU**" e questo tipo di situazione prende il nome di **attesa attiva**.

Rilascio di una risorsa: **unlock()**

La **primitiva** (o funzione) che permette di rilasciare una **risorsa** prende il nome di **unlock()**. Possiamo indicare la sua sintassi nel seguente formato:

```
unlock(x);
```

dove **x** è il **semaforo** associato alla **risorsa** che desideriamo utilizzare.

La primitiva `unlock()` deve semplicemente modificare il valore del **semaforo** da rosso a verde, quindi:

```
funzione unlock(x)
inizia
    x ← 1;           // metti il semaforo a verde
fine
```

La **mutua esclusione** si ottiene facendo precedere la `lock(x)` a una **sezione critica** e facendola seguire da una `unlock()`.

```
...
lock(x);
< sezione critica >
unlock(x);
...
```

Problema della indivisibilità

Per come abbiamo scritto l'istruzione di `lock()` non possiamo garantire l'acceso seriale a una risorsa, in quanto potrebbe verificarsi una situazione di **interleaving** indesiderata.

Vediamo per esempio la seguente situazione:

- 1 ipotizziamo che il semaforo sia verde $x = 1$;
- 2 il processo P1 effettua la `lock(x)` fino a verificare la condizione di test ma viene sospeso prima che ne possa modificare il valore;
- 3 un secondo processo P2 effettua anch'esso la `lock(x)` e, trovando il semaforo verde, lo mette a rosso e inizia a utilizzare la risorsa;
- 4 quando viene risvegliato il processo P1 esegue l'istruzione che pone a rosso il semaforo (che di fatto però è già rosso) e anch'esso utilizza la risorsa.

Risulta perciò violata la **mutua esclusione** in quanto i due **processi** si trovano contemporaneamente a utilizzare la **risorsa**.

Per evitare questa situazione è necessario rendere **indivisibile** l'esecuzione delle istruzioni della funzione `lock(x)`: la soluzione potrebbe essere quella di **disabilitare le interruzioni** all'inizio e al termine di questa funzione in modo che diventi ininterrompibile.

Per esempio il codice potrebbe essere il seguente:

```
funzione lock(x)
inizia
<disabilitare le interruzioni>
    ripeti           // ciclo di attesa sul semaforo nel caso che sia rosso
        finche x=1   // esci dal ciclo a semaforo verde
        x ← 0;       // metti il semaforo a rosso
<abilitare le interruzioni>
fine
```

Analogo problema potrebbe verificarsi per **interleaving** anche sulla primitiva di **unlock(x)** e quindi anch'essa viene eseguita a interruzioni disabilitate.

```
funzione unlock(x)
inizia
<disabilitare le interruzioni>
    x ← 1;           // metti il semaforo a verde
<abilitare le interruzioni>
fine
```

Continuare ad **abilitare e disabilitare le interruzioni** porta però a compromettere le prestazioni del sistema: se osserviamo attentamente le due primitive ci accorgiamo che il problema si verifica a causa dell'interruzione tra il test del semaforo e il suo settaggio al nuovo valore. Introduciamo a livello macchina (quindi hardware) una nuova istruzione, che chiamiamo **TestAndSet(x)**, che controlla e modifica il valore di un bit in modo ininterrompibile, e una funzione **Set(x)** che ne effettua il semplice settaggio, da utilizzarsi nella **unlock()**.

Riscriviamo le due primitive utilizzando queste due nuove istruzioni:

```
funzione lock(x)
inizia
    ripeti           // ciclo di attesa sul semaforo nel caso che sia rosso
        <ritardo>
        finche TestAndSet(x) // esci dal ciclo settando il semaforo a rosso
    fine
```

e

```
funzione unlock(x)
inizia
    set(x);          // metti il semaforo a verde
fine
```

Nonostante questo miglioramento rimane da risolvere il problema della **attesa attiva**; inoltre non abbiamo la garanzia esplicita che un processo non attenda indefinitamente su di un semaforo che, anche se diviene verde, viene ripetutamente assegnato ad altri processi e non a lui.

Gli **spinlock** non vengono quindi utilizzati come meccanismo di sincronizzazione ma sono alla base della realizzazione di primitive più complesse.



Zoom su...

TSL

I primi a notare la necessità di avere una istruzione indivisibile furono i progettisti dell'OS/360 che aggiunsero nel linguaggio macchina del sistema l'istruzione **TEST AND SET LOCK (TSL)**.

■ Semafori di Dijkstra

E.W. Dijkstra nel 1968 ha proposto due primitive che permettono la soluzione di qualsiasi problema di interazione fra processi, che sono:

- la primitiva **P(S)**, che riceve in ingresso un numero intero S non negativo (**semaforo**), che viene utilizzata per accedere alla **risorsa**;
- la primitiva **V(S)**, che riceve anch'essa in ingresso un numero intero S non negativo (**semaforo**), che viene utilizzata per rilasciare la **risorsa**.

Introduciamo quindi un nuovo tipo di dato, il **semaphore**, dove una sua istanza non è altro che una variabile intera non negativa alla quale è possibile accedere solo tramite le due primitive **P(S)** e **V(S)**.

ESEMPIO **Il primo semaforo**

Dichiarazione di un oggetto di tipo **semaphore**:

```
semaphore s = vi;
```

dove vi ($vi \geq 0$) è il *valore iniziale* e il **valore 0** corrisponde al **rosso**.

L'idea di **Dijkstra** è quella di disciplinare l'accesso alle risorse mediante code e rendendo inattive le situazioni di attesa: se un semaforo è rosso il processo che fa il test (richiama la primitiva **P(S)**) si sospende e viene messo nella coda dei processi, in attesa che si liberi quella risorsa mentre il processo che rilascia la risorsa invoca la primitiva **V(S)** che modifica il valore del semaforo S e risveglia il primo processo presente nella coda di attesa.

Quindi due o più processi possono **cooperare** attraverso semplici segnali, in modo tale che un processo possa essere bloccato in specifici punti del suo programma finché non riceve un segnale da un altro **processo**.

ESEMPIO **Semafori ferroviari**

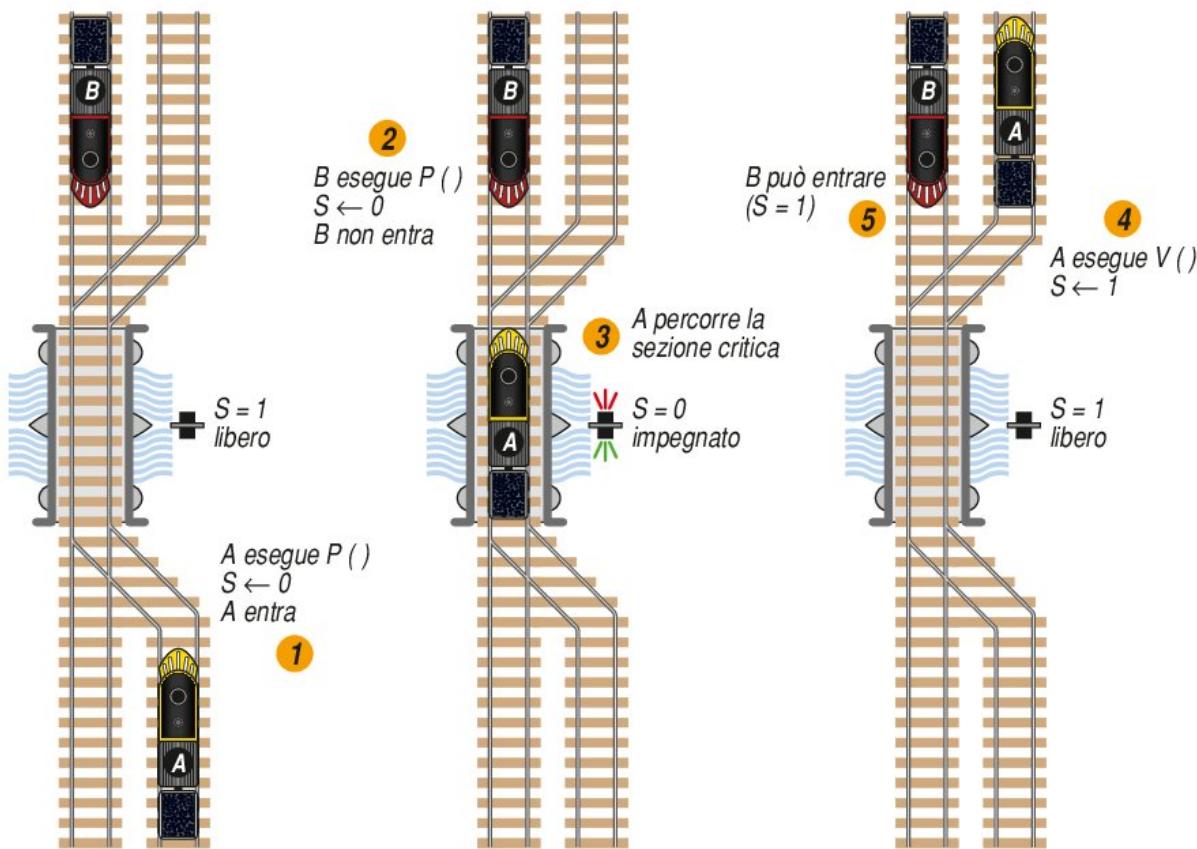
Il nome “**semaforo**” proviene dalla segnalazione ferroviaria e facciamo un esempio di gestione rimanendo in ambito ferroviario: supponiamo di avere la situazione rappresentata in figura, dove una linea a due binari a un certo punto deve confluire su di un ponte con binario unico.

Il ponte è la **risorsa** condivisa che dovrà essere gestita in **mutua esclusione**.

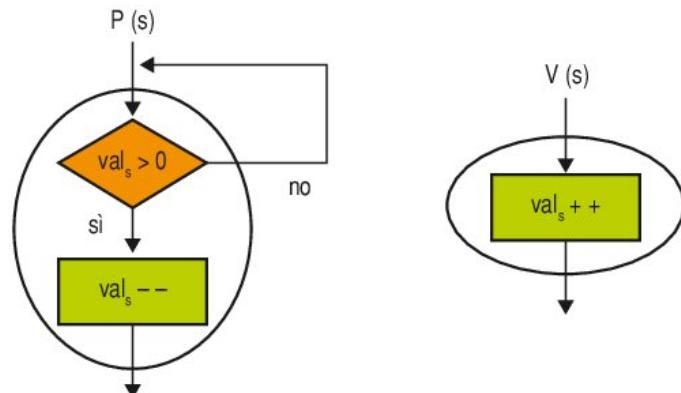
Associamo al ponte un semaforo che viene controllato e gestito dai macchinisti dei treni: quando un treno A si avvicina al ponte osserva il semaforo:

- 1 il macchinista di A controlla lo stato del semaforo (esegue una **P(S)**): lo trova spento, quindi lo accende ($S = 0$) e inizia a transitare sul ponte;
- 2 anche il treno B sopraggiunge al ponte ma trovando il semaforo acceso esegue anch'esso una **P(S)**, si ferma e rimane in attesa;
- 3 A continua la sua corsa attraversando il ponte;
- 4 una volta raggiunta l'altra sponda, spegne il semaforo ($S=1$);

- 5 B ora vede il semaforo spento e lo accende ($S=0$) e inizia pure lui ad attraversare il ponte.



Lo **schema a blocchi** delle due istruzioni è quello a fianco: ▶



La pseudocodifica delle due primitive è la seguente:

```

funzione P(s)
inizia
  se S=0           // risorsa occupata - semaforo rosso
    allora
      <il processo viene posto nella coda di attesa>
    altrimenti
      S  $\leftarrow$  S-1;   // accedi alla risorsa
  fine

```

e

```
funzione V(S)
inizia
  se <è presente un processo in attesa>
    allora
      <poni il primo processo nello stato di pronto>
      S ← S+1;           // rilascia la risorsa
  fine
```

Un **processo** che esegue una **P(S)** può avanzare solo se trova $S > 0$, altrimenti deve accodarsi per attendere passivamente una **V(S)**; un **processo** che esegue una **V(S)**, se non esistono processi in attesa dentro la coda, ha come unico effetto quello di incrementare S , altrimenti risveglia uno dei processi che attendono nella coda: successivamente, in entrambe le situazioni, continua la propria evoluzione.

Naturalmente le primitive **P(S)** e **V(S)** devono essere indivisibili per evitare la sequenza di interleaving indesiderata e la procedura di assegnazione della risorsa ai processi in attesa viene stabilita in modo tale da garantire la proprietà di fairness, per esempio con una coda **FIFO**.

In un sistema uniprocessore si realizzano introducendo spinlock e/o disabilitando/riabilitando gli interrupt all'inizio/fine di ciascuna di esse (dato che sono implementate direttamente dal **sistema operativo** e l'intervallo temporale in cui gli interrupt sono disabilitati è molto breve). La realizzazione completa delle due primitive è la seguente:

```
funzione P(S)
inizia
  <disabilita le interruzioni>
  LOCK(SX)
  se S=0
    allora
      <il processo viene posto nella coda di attesa>
    altrimenti
      S ← S-1;           // accedi alla risorsa
  UNLOCK(SX)
  <abilita le interruzioni>
fine
```

```
funzione V(S)
inizia
  <disabilita le interruzioni>
  LOCK(SX)
  se <è presente un processo in attesa>
    allora
      <poni il primo processo nello stato di pronto>
      S ← S+1;           // rilascia la risorsa
  UNLOCK(SX)
  <abilita le interruzioni>
fine
```

Si può verificare lo **spin lock SX** utilizzato per garantire l'indivisibilità delle due primitive: è necessario solamente nei sistemi **multi-processore** mentre per i sistemi **uniprocessore** è sufficiente disabilitare le interruzioni.



Zoom su...

ORIGINE DI P E V

Originariamente queste primitive avevano il nome di **wait** e **signal**: **Dijkstra** li sostituì con le iniziali **P** e **V** di due termini olandesi:

- ▶ **V** è l'iniziale dal termine olandese **verhogen**: aumentare, alzare di livello;
- ▶ **P** è l'iniziale dal termine olandese **proberen**: provare, testare.

■ Semafori binari vs semafori di Dijkstra

I semafori di **Dijkstra** vengono anche chiamati **semafori generalizzati** (o a conteggio) per distinguerli dai **semafori binari**.

In letteratura si trovano spesso i nomi **down(S)** e **up(S)** per indicare rispettivamente **P(S)** e **V(S)** dato che l'istruzione **P(S)** decrementa di 1 il valore del semaforo (**down(S)**) e l'istruzione **V(S)** invece la incrementa (**up(S)**).

Naturalmente un semaforo di **Dijkstra** può essere utilizzato come semplice semaforo binario utilizzando solamente i valori 0 e 1.

Molteplicità di una risorsa

I **semafori a conteggio** vengono utilizzati per controllare l'accesso a una risorsa disponibile in un numero finito di esemplari: noi vedremo nel seguito una applicazione nel caso di utilizzo di un array di NUM celle come **risorsa condivisa**, ciascuna di esse destinata a contenere una variabile da condividere e quindi il **semaforo** verrà inizializzato al valore NUM per indicare che sono disponibili NUM risorse e quindi il **semaforo** rimane verde finché non sono state tutte “occupate”.

Al test sul **semaforo S** possiamo avere due situazioni:

- 1 **S = X** dove $x \leq \text{NUM}$ è il numero di esemplari di risorsa liberi: se $X > 0$ il processo può accedere come in presenza di semaforo verde;
- 2 **S = 0** risorse occupate, cioè 0 risorse libere e quindi il semaforo è rosso.

Le istruzioni di **P(S)** e **V(S)** incrementano e decrementano la molteplicità di risorsa libera:

- ▶ quando viene effettuata una **P(S)** su semaforo che ha valore > 0 ne viene decrementato di 1 il suo valore, dato che viene occupata “una sua parte”;
- ▶ quando viene effettuata una **V(S)** su un semaforo viene incrementato di una unità il suo valore dato che ne viene resa disponibile “una parte” per gli altri **processi**.

Verifichiamo le conoscenze

1. Risposta multipla

- 1** In uno spin lock, a seconda del valore di x abbiamo:
 - a) $x = 1$ risorsa libera
 - b) $x = 0$ risorsa libera
 - c) $x = 1$ risorsa occupata
 - d) $x = 0$ risorsa occupata
- 2** L'istruzione `TestAndSet(x)`:
 - a) modifica il valore di un bit in modo ininterrompibile
 - b) modifica il valore di un byte in modo ininterrompibile
 - c) modifica il valore di x in modo ininterrompibile
 - d) tutte le affermazioni precedenti sono valide: dipende dalla applicazione
- 3** L'istruzione `TestAndSet(x)`:
 - a) risolve il problema della attesa attiva
 - b) risolve il problema della starvation
 - c) risolve il problema dell'interleaving
 - d) non risolve alcun problema
- 2** Quale gruppo di primitive esegue la stessa funzione?
 - a) `P(S)` `WAIT(S)` `DOWN(S)`
 - b) `P(S)` `SIGNAL(S)` `UP(S)`
 - c) `P(S)` `WAIT(S)` `UP(S)`
 - d) `P(S)` `SIGNAL(S)` `DOWN(S)`

2. Associazione

- 1** Associa a ciascuna conseguenza la strategia necessaria:

Conseguenze	Strategie
inaccettabili	ignorare
trascurabili	evitare ogni interferenza
rilevabili e controllabili	rilevare e ripetere
rilevabili e recuperabili	rilevare ed evitare

3. Vero o falso

- 1** Se la mancanza di sincronizzazione provoca danni trascurabili la miglior strategia è quella di ignorare il problema.
- 2** La serializzazione può essere ottenuta introducendo cicli di ritardo.
- 3** L'istruzione di lock su uno spin può generare una attesa attiva.
- 4** Per evitare il verificarsi di interleaving la primitiva `lock(x)` si esegue a interruzioni disabilitate.
- 5** La primitiva di `unlock(x)`, essendo una sola istruzione, non richiede interruzioni disabilitate.
- 6** Dijkstra disciplina l'accesso alle risorse mediante code e rendendo inattive le situazioni di attesa.
- 7** Nel semaforo di Dijkstra il valore iniziale 0 corrisponde al rosso.
- 8** Le primitive `P(S)` e `V(S)` non necessitano di essere indivisibili in quanto sono atomiche.
- 9** La `P(S)` su un semaforo rosso sospende il processo e lo mette in una coda.
- 10** La `V(S)` su un semaforo verde mette il semaforo a rosso senza sospendere il processo.

V F
 V F
 V F
 V F
 V F
 V F
 V F
 V F
 V F
 V F

Verifichiamo le competenze

1. Esercizi

- 1 Commenta e completa i seguenti segmenti di codice.

```
funzione lock(x)          // .....
inizia
<disabilitare le interruzioni>
    ripeti           // .....
    finche x=1       // .....
    x ← 0;           // .....
<abilitare le interruzioni>
fine
```

```
funzione unlock(x)        // .....
inizia
<disabilitare le interruzioni>
    x ← 1;           // .....
<abilitare le interruzioni>
fine
```

```
funzione P(S)             // .....
inizia
<.....>
LOCK(SX)
    se S=0
        allora
        <.....>
        altrimenti
            S ← S-1;      // .....
UNLOCK(SX)
<.....>
fine
```

```
funzione V(S)
inizia
<.....>
LOCK(SX)
    se <.....>
        allora
        <.....>
        S ← S+1;          // .....
UNLOCK(SX)
<.....>
fine
```

- 2** Il seguente programma concorrente, a seconda dell'interleaving fra i due processi, può stampare una o più stringhe. Indicare quante sono le possibilità e visualizzarle. **Nota:** tutti i semafori sono generali e inizializzati a 0.

```
process Prol {
    s1.V();
    print ("F");
    s1.P();
    s2.V();
    s1.P();
    print ("C");
}
process Pro2 {
    print ("G");
    s2.P();
    print ("D");
    s1.V();
}
```

- 3** Che cosa fa questo programma concorrente? (s1, s2 semafori, entrambi inizializzati a 0).

```
process P {
    s1.V();
    s2.P();
    print ("O")
    s2.P();
    print ("D")
    s1.V();
}
process Q {
    s1.P();
    print ("L");
    s2.V();
    s2.V();
    s1.P();
    print ("E");
}
```

- 4** Considerare i seguenti processi:

```
process P {
    print ("P");
    print ("E");
    print ("E");
}
process Q {
    print ("R");
    print ("S");
}
```

Modificare il codice dei processi in modo che stampino unicamente le parole PERSE e PERES (tramite semafori).

- 5** a) Al termine di questo programma, quali sono i valori possibili della variabile condivisa val?
 b) È possibile che i processi P o Q restino bloccati indefinitamente?

```
// DEFINIZIONE
shared val = 0;
shared Semaphore sp =
new Semaphore(2);
shared Semaphore sq =
new Semaphore(1);
shared Semaphore mutex =
new Semaphore(1);
}
process P {
    int kp = 2;
    while (kp > 0) {
        sp.P();
        mutex.P();
        val = val+1;
        sq.V();
        kp--;
        mutex.v();
    }
}
process Q {
    int kq = 3;
    while (kq > 0) {
        sq.P();
        mutex.P();
        val = val*2;
        sp.V();
        kq--;
        mutex.v();
    }
}
```

Applicazione dei semafori

In questa lezione impareremo...

- ▶ a realizzare la mutua esclusione mediante i semafori
- ▶ a regolare l'accesso multiplo tramite i semafori
- ▶ a utilizzare i semafori per realizzare i vincoli di precedenza

■ Semafori e mutua esclusione

Il **semaforo** viene utilizzato come strumento di sincronizzazione tra processi **concorrenti** che intendono **cooperare**, come per esempio nelle situazioni **produttore-consumatore**; per garantire la mutua esclusione nel caso di un singolo produttore e un singolo consumatore è sufficiente utilizzare semafori binari, cioè associare alla variabile il valore 1 per libero e 0 per occupato (come per gli **spinlock**).

Nella letteratura l'istanza di un semaforo viene generalmente indicata con l'identificatore **mutex** che deriva dalla contrazione dell'inglese **mutual exclusion** (mutua esclusione), e anche nella nostra trattazione utilizzeremo questa terminologia.

Scriviamo lo schema del codice di due **processi** che accedono **concorrentemente** a una **risorsa condivisa**:

```
programma concorrente MutuaEsclusione
semaphore mutex = 1;
...
Processo operazione1()
inizio
...
P(mutex);                                /* prologo */
<corpo della funzione produci>
V(mutex);                                /* epilogo */
...
fine
```

```

Processo operazione2 ()
...
inizio
P(mutex);                                /* prologo */
<corpo della funzione consuma>
V(mutex);                                /* epilogo */
...
fine

inizialia /* principale*/
...
cobegin
    operazione1 ()
    operazione2 ()
coend
...
fine

```

È possibile dimostrare che la soluzione proposta risolve correttamente il problema della mutua esclusione, in quanto soddisfa le condizioni necessarie:

- ▶ le sezioni critiche devono essere eseguite in modo **mutuamente esclusivo**;
- ▶ non si devono verificare situazioni in cui i processi impediscono mutuamente la prosecuzione della loro esecuzione (**deadlock**);
- ▶ quando un processo si trova all'esterno di una sezione critica **non può rendere impossibile** l'accesso alla stessa sezione ad altri processi.

ESEMPIO *Prenotazione posti al cinema*

In una sala cinematografica con più casse per la vendita dei biglietti, il numero di posti è continuamente aggiornato e memorizzato in una variabile intera condivisa **postiLiberi** (inizializzata a 200). Lo spettatore richiede **quanti** posti (> 0): se è possibile soddisfare la richiesta, i posti vengono occupati aggiornando **postiLiberi** altrimenti viene data una segnalazione visiva. La **regione critica** viene gestita con un **semaforo** che garantisce la **mutua esclusione** alla risorsa condivisa (**postiLiberi**): una possibile pseudocodifica è la seguente:

```

programma concorrente MutuaEsclusione
semaphore mutex = 1;
int postiLiberi=200;
...
Processo occupa(int quanti)
inizio
    P(mutex);                                /*prologo*/
    se postiLiberi>quanti
        allora
            postiLiberi= postiLiberi-quanti
        altrimenti
            scrivi("posti non disponibili")
    V(mutex);                                /*epilogo*/
fine

```

■ Mutua esclusione tra gruppi di processi

In alcuni casi è consentito a più **processi** di eseguire contemporaneamente la stessa operazione su una **risorsa** (per esempio la lettura di dati) ma affinché un **processo** possa eseguire una operazione diversa da quella che stanno eseguendo tutti gli altri è necessario che tutti questi finiscano di utilizzare la **risorsa condivisa** e che questa quindi risulti “libera da ogni processo”.

Indichiamo con **operazioneK** il prototipo della generica operazione che ha la seguente struttura:

```
procedura operazioneK ()
inizio
    <operazioni preliminari>           // prologo
    <corpo della funzione>
    <operazioni conclusive>           // epilogo
fine
```

Le **operazioni preliminari** che il **processo** deve eseguire (**prologo**) sono quelle inerenti al controllo degli accessi, cioè il **processo** che ha chiamato l'operazione **operazioneK** si deve sospendere se sulla **risorsa** sono in esecuzione operazioni diverse previste nel codice di **operazioneK**; nelle altre situazioni, cioè quando la **risorsa** è libera oppure utilizzata da un altro **processo** che fa la medesima operazione, si deve consentire al **processo** di accedere alla risorsa.

Le **operazioni conclusive** nel caso in cui il **processo** che sta uscendo è l'ultimo (o il solo) presente nella **regione critica** consistono nel liberare la **risorsa** per le altre attività, altrimenti si deve semplicemente aggiornare il numero dei **processi** presenti.

Nelle istruzioni del **prologo** e **dell'epilogo** è quindi necessario utilizzare una *variabile di conteggio* dei **processi** presenti contemporaneamente nella **regione critica**, che chiameremo **ProcessiDentro** e anch'essa, dato che è in condivisione con gli altri **processi**, viene aggiornata all'interno di sezioni critiche che dovranno essere gestite in mutua esclusione: si introduce un nuovo semaforo che viene utilizzato appositamente per regolare l'accesso dei processi che devono modificare il contatore.

Scriviamo un primo affinamento della procedura, introducendo la nuova variabile **mutexPDK** che è il semaforo che regola l'aggiornamento della variabile **ProcessiDentroK** della **operazioneK**:

```
semaphore mutex =1, mutexPDK = 1;
int ProcessiDentroK = 0;
procedura operazioneK ( )
inizial
    P(mutexPDK)
        <controllo e aggiornamento ProcessiDentroK >           // prologo
    V(mutexPDK)
        <corpo della funzione operazioneK >                   / parte centrale
    P(mutexPDK)
        <controllo e aggiornamento ProcessiDentroK >           // epilogo
    V(mutexPDK)
fine
```

Le operazioni effettuate dal prologo sono di seguito elencate e commentate:

```
P(mutexPDK)                                //accesso in mutua esclusione alla RC
    ProcessiDentroK++;
    if (ProcessiDentroK ==1)                  //aggiorna numero processi
        P(mutex)                            //se è il primo processo che accede
    V(mutexPDK)                           //mette a rosso il semaforo sulla risorsa
                                            //libera accesso in mutua esclusione alla RC
```

Le operazioni effettuate dall'epilogo sono di seguito elencate e commentate:

```
P(mutexPDK)                                //accesso in mutua esclusione alla RC
    ProcessiDentroK--;
    if (ProcessiDentroK ==0)                  //aggiorna numero processi
        V(mutex)                            //se l'ultimo che utilizza la risorsa
    V(mutexPDK)                           //mette a verde il semaforo sulla risorsa
                                            //libera accesso in mutua esclusione alla RC
```

Il codice completo è il seguente:

```
semaphore mutex =1, mutexPDK = 1;
int ProcessiDentroK = 0;
procedura operazioneK( )
inizia
// prologo
    P(mutexPDK)
        //controllo e aggiornamento ProcessiDentroK
        ProcessiDentroK++;                //aggiorna numero processi
        if (ProcessiDentroK ==1)          //se è il primo processo che accede
            P(mutex)                    //mette a rosso il semaforo sulla risorsa
    V(mutexPDK)
// parte centrale
< corpo della funzione operazioneK >
// epilogo
    P(mutexPDK)
        // controllo e aggiornamento ProcessiDentroK
        ProcessiDentroK--;              //aggiorna numero processi
        if (ProcessiDentroK ==0)          //se l'ultimo che utilizza la risorsa
            V(mutex)                    //mette a verde il semaforo sulla risorsa
    V(mutexPDK)
fine
```

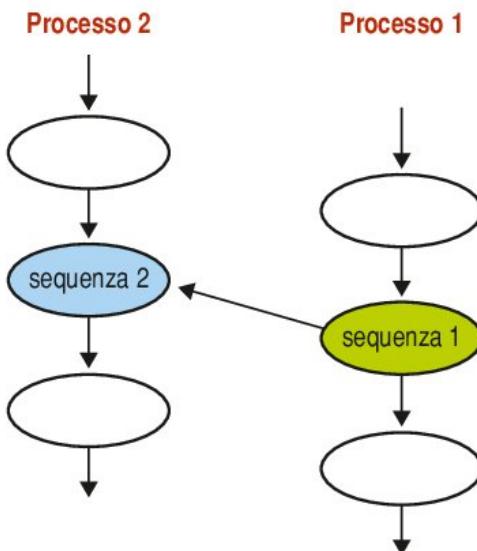
È immediata l'estensione nel caso di un insieme di attività simili da eseguirsi contemporaneamente: basta scrivere una procedura per ogni operazione e aggiungere un contatore e un semaforo riservato per regolarne l'accesso a ciascuna di esse.

Deve invece rimanere unico il semaforo **mutex** che regola l'accesso alla risorsa comune elaborata all'interno del **<corpo della funzione operazioneK>**.

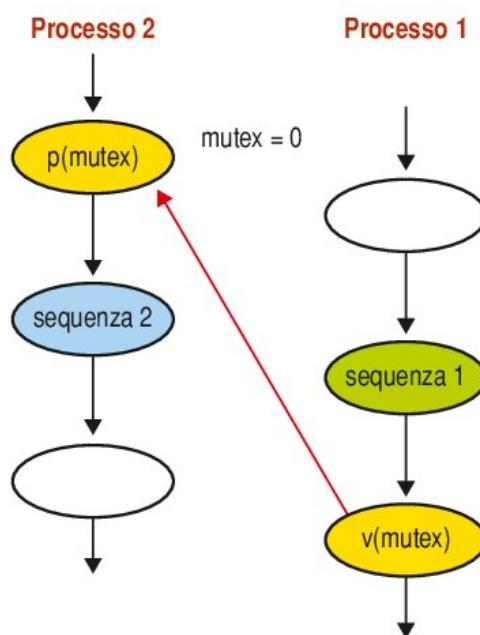
■ Semafori come vincoli di precedenza

I **semafori binari** possono anche essere utilizzati per stabilire dei **vincoli di precedenza** sull'esecuzione di gruppi di operazioni in processi paralleli.

Supponiamo di avere due sequenze di istruzioni **sequenza1** e **sequenza2** che devono essere eseguite da due processi diversi necessariamente in successione, cioè il secondo processo esegue la **sequenza2** solo dopo che il primo processo ha eseguito la **sequenza1**.

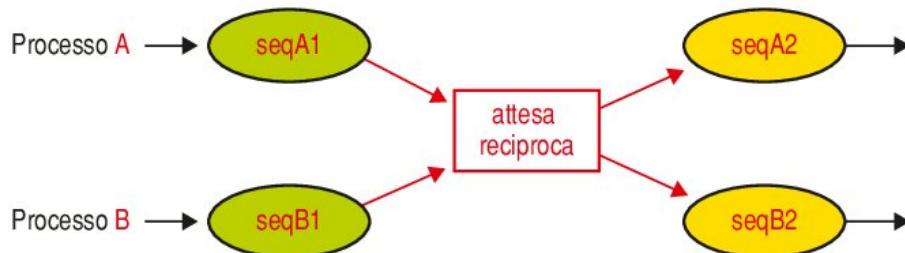


Introduciamo un semaforo **mutex** che inizializziamo a rosso (**mutex=0**) e che viene messo a verde dal **processo1** solo dopo che ha eseguito la **sequenza1**: il secondo processo, prima di eseguire la **sequenza2**, effettua un test su questo semaforo che troverà **verde** solo dopo l'esecuzione della **sequenza2** altrimenti, trovandolo **rosso**, aspetta il **processo1**.



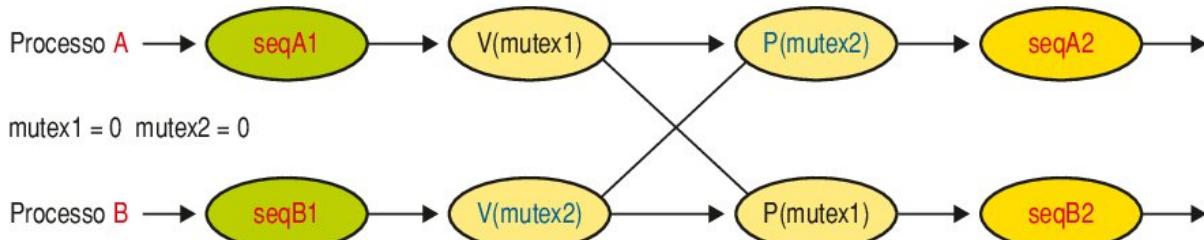
■ Problema del rendez-vous

I semafori binari possono anche essere utilizzati per gestire dei “rendez-vous” tra due processi. Per esempio, supponiamo che due processi debbano eseguire due sequenze per ciascuno ma per ogni processo la seconda sequenza deve essere eseguita dopo che anche l’altro processo ha eseguito la prima: in altre parole il processo più veloce deve attendere il processo più lento.



La sequenza **seqA2** deve essere eseguita dopo la sequenza **seqA1** e la sequenza **seqB2** deve essere eseguita dopo le sequenze **seqA1** e **seqB1**.

Introduciamo due semafori in modo che i due processi si possano scambiare “segnali temporali in modo simmetrico”: quando un processo giunge “all’appuntamento” segnala all’altro di esserci arrivato e lo attende. Poniamo all’inizio dell’esecuzione i due semafori a rosso e facciamo in modo che ogni processo metta a verde il semaforo che ferma la prosecuzione dell’elaborazione dell’altro processo e successivamente testa il semaforo che è gestito dall’altro processo che gli permette di proseguire, come riportato nel seguente diagramma:



Se arriva prima il **processoA** al **rendez-vous**, dopo aver messo a verde il **mutex1** trova **mutex2** rosso e quindi si ferma fino a che sopraggiunge il **processoB** a mettere a verde il **mutex2**, risvegliando il **processoA** che riprende la sua evoluzione: il **processoB** può proseguire perché trova a verde il semaforo **mutex1** settato dal **processoA** come prima operazione al suo arrivo alla zona di sincronizzazione, quindi ora entrambi possono procedere eseguendo rispettivamente la **seqA2** e la **seqB2**.

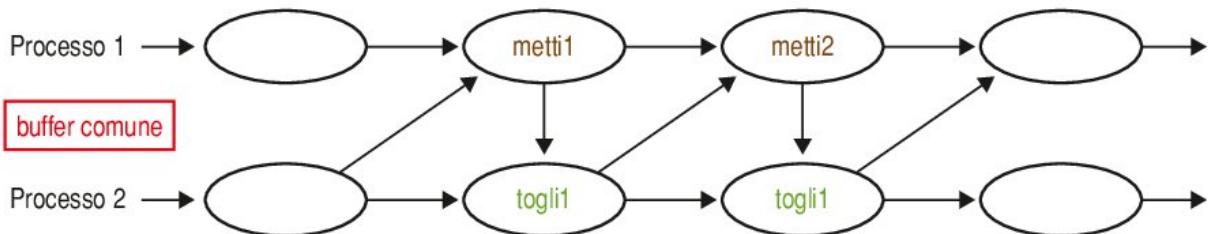
ESEMPIO Rendez-vous prolungato

Scriviamo un esempio dove due processi, per poter proseguire, devono scambiarsi alcuni dati utilizzando un **buffer** condiviso: il primo processo scrive un dato alla volta, ma per poter scrivere il secondo dato deve attendere che il secondo processo prelevi il primo, e così via. Individuiamo i vincoli di sincronizzazione:

- i processi devono effettuare l’accesso al buffer comune in mutua esclusione;
- il processo P2 può prelevare un dato solo dopo che P1 lo ha inserito;
- il processo P1, prima di inserire un nuovo dato, deve attendere che P2 abbia estratto il precedente.

Graficamente indichiamo solamente che devono essere scambiati due dati, ma il programma che scriviamo non ha limiti di funzionamento sul numero di dati da scrivere e leggere.

Graficamente la situazione è la seguente



La realizzazione della sincronizzazione prevede l'utilizzo di due semafori:

- ▶ **mutexBV**: per realizzare l'attesa del processo1 in caso di buffer pieno;
- ▶ **mutexBP**: per realizzare l'attesa del processo2 in caso di buffer vuoto.

Inizialmente il buffer è vuoto e i due semafori sono così settati:

```
buffer = <vuoto>
mutexBV= 1 //all'inizio il semaforo che indica buffer vuoto è verde
mutexBP= 0 //all'inizio il semaforo che indica buffer pieno è rosso
```

Il **processo** che scrive effettua il test sul **semaforo mutexBV** che indica se il **buffer** è vuoto, quindi se lo trova verde lo mette a rosso, riempie il **buffer** e setta a verde il **semaforo mutexBP** indicando che ora il **buffer** è pieno, pronto per la lettura:

```
Procedura invio(dato)
inizio
...
P(mutexBV)                                // se il buffer è vuoto riempilo
inserisci(dato)
V(mutexBP)                                 // indica che il buffer è pieno
...
fine
```

Il **processo** che deve leggere il contenuto del **buffer** per prima cosa testa il semaforo **mutexBP** che indica se il **buffer** è pieno, quindi lo svuota e successivamente setta a verde il **semaforo mutexBV** indicando che il **buffer** è vuoto, pronto per una nuova scrittura:

```
Procedura ricezione( )
inizio
...
P(mutexBP)                                // se il buffer è pieno svuotalo
estrai(dato)
V(mutexBV)                                 // indica che il buffer è vuoto
...
fine
```

Verifichiamo le competenze

1. Esercizi

- 1 Considera i seguenti processi,

```
Risorse condivise
semaphore S=1;
int x=0;

Processo P1{
    while (true) do
    begin
        down(S);
        x:=x+1;
        up(S);
    end
}

Processo P2{
    while (true) do
    begin
        down(S);
        write(x);
        up(S);
    end
}
```

e supponi che i processi vengano eseguiti concorrentemente sulla stessa CPU.

Quali sono le sezioni critiche in questo programma? Vale la proprietà di mutua esclusione?
Descrivi il comportamento e i possibili output del programma concorrente.

- 2 Scrivi lo pseudocodice dove due processi devono scambiarsi reciprocamente dei dati per poter proseguire utilizzando un buffer condiviso: il primo processo che scrive un dato prima di proseguire deve attendere che il secondo processo lo prelevi e inserisca a sua volta un dato destinato a lui.
- 3 Scrivi lo pseudocodice di tre processi concorrenti che risolvono il seguente problema:
si vogliono sommare gli elementi di un array 'buf' di 20 elementi, cioè fare l'operazione $buf[0]+buf[1]+\dots+buf[19]$. L'operazione viene fatta però in modo concorrente, in modo tale cioè che il primo thread sommi gli elementi di posizione pari dell'array, mentre il secondo sommi gli elementi di posizione dispari. Il terzo thread sommi i risultati dei primi due. Quanti semafori sono necessari? Come devono essere inizializzati i semafori? Scrivi una pseudocodifica della soluzione.
- 4 Un conto corrente richiede che alla variabile saldo si acceda in modo concorrente da parte di due processi Preleva e Versa e quindi si deve regolare l'accesso in mutua esclusione: completa il codice riportato di seguito in modo da garantire il corretto funzionamento della sincronizzazione sapendo che la banca non permette scoppio sul conto corrente.

```
Risorse condivise
int saldo = 300;
semaphore mutex1=.....;

Processo Preleva() {
    int movimento;
    .....
    .....
    .....
    movimento = saldo;
    movimento = movimento-500;
    saldo      = movimento;
    scrivi ("effettuato prelievo di 500 euro")
    .....
    .....
}
```

```

Processo Versa() {
    int movimento;
    .....
    .....
    movimento = saldo;
    movimento = movimento+500;
    conto      = movimento;
    scrivi ("deposito di 500 euro")
    .....
    .....
}

```

- 5** Un cinema con capienza di MAX persone non dispone di un sistema di prenotazione, e quindi un cliente arriva alla biglietteria dove sono presenti due casse, aspetta il suo turno e, se trova un posto libero, può accedere alla proiezione.

Al termine della proiezione due hostess intervistano gli spettatori richiedendo un parere espresso con un numero da 1 a 5.

Si richiede di gestire le sincronizzazioni e di scrivere il codice per visualizzare ordinatamente la classifica dei pareri ricevuti.

- 6** Si deve gestire l'ufficio prenotazioni/cassa di un ospedale: i pazienti prendono un ticket con un numero progressivo da un dispenser in base a 5 categorie di servizio/prestazione, etichettate con le lettere da A a F dove le richieste di tipo A hanno priorità più alta rispetto alle altre, e così via, in ordine decrescente fino alla F. Sono disponibili tre sportelli che devono smaltire il lavoro in base alla priorità e al numero d'ordine del paziente, garantendo l'assenza di starvation (per esempio, limitando l'attesa massima a 20 utenti).

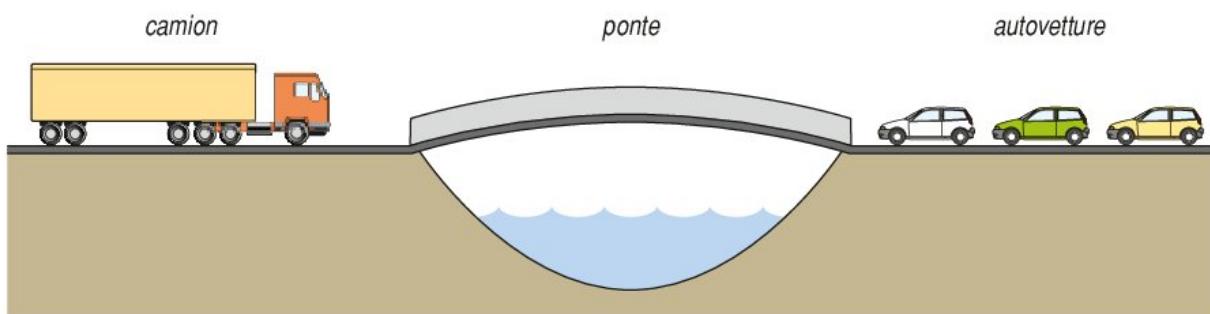
Scrivi una possibile soluzione in pseudocodifica.

- 7** Si consideri la situazione rappresentata in figura dove due strade sono unite da un ponte a un unico senso di marcia e quindi il traffico lo attraversa a senso unico alternato.

Non viene stabilita alcuna precedenza, quindi il primo che sopraggiunge al ponte, se lo trova libero oppure se altre macchine lo stanno attraversando nella suo stesso senso di marcia, lo può attraversare: un'autovettura che giunge dalla direzione opposta deve attendere che tutti quelli che sono sul ponte abbiano finito di attraversarlo.

Descrivi la situazione regolando la sincronizzazione in modo tale che al massimo possano passare 100 autovetture consecutive per senso di marcia.

Successivamente introduci la condizione che possano sopraggiungere sia autovetture che camion ma che, per il loro peso, questi ultimi debbano attraversare il ponte da soli.



Problemi “classici” della programmazione concorrente: produttori/consumatori

In questa lezione impareremo...

- ▶ le caratteristiche del problema produttori/consumatori
- ▶ a risolvere il problema produttori/consumatori

■ Generalità

Esiste un certo numero di problemi “classici” della **programmazione concorrente** che rappresentano la casistica completa delle diverse situazioni di **concorrenza**; possono essere suddivisi in tre gruppi, e ne riportiamo il nome con il quale sono indicati nella letteratura informatica:

- ▶ **produttore/consumatore** (producer/consumer)
 - 1 produttore – 1 consumatore
 - buffer limitato o circolare (bounded buffer)
 - n produttori – n consumatori
- ▶ **lettori e scrittori** (readers/writers)
- ▶ **filosofi a cena** (dining philosophers)

In questa lezione descriveremo il primo dei tre problemi nelle sue diverse situazioni mentre rimandiamo gli altri alle successive lezioni.

Ricordiamo che una **soluzione al problema della sincronizzazione** di **processi** è ammissibile se soddisfa le seguenti 4 condizioni:

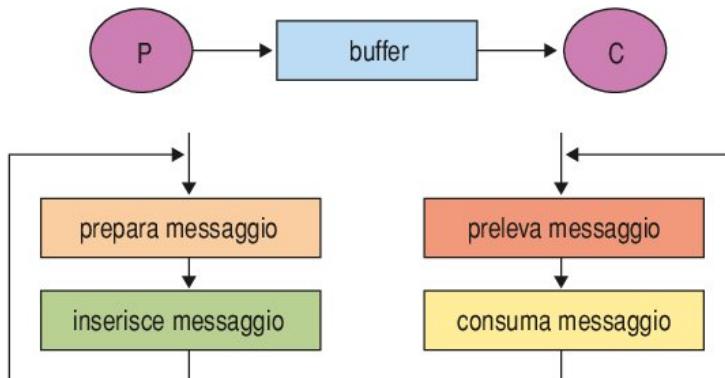
1. deve garantire attesa finita per ogni processo del sistema;
2. non bisogna fare assunzioni sull’ambiente di esecuzione;
3. non deve subire condizionamenti dai **processi** esterni alla sezione critica;
4. deve soddisfare i requisiti per la gestione della **sezione critica (SC)**, cioè:
 - ▶ **accesso in mutua esclusione**: solo un **processo** alla volta deve essere all’interno della **SC**, fra tutti quelli che hanno una **SC** per la stessa **risorsa condivisa**;

- **assenza di deadlock:** uno scenario in cui tutti i processi restano bloccati definitivamente non è ammissibile;
- **assenza di delay non necessari:** un processo fuori dalla SC non deve ritardare l'ingresso della SC da parte di un altro processo;
- **eventual entry (assenza di starvation):** ogni processo che lo richiede, prima o poi entra nella SC.

■ Problema dei produttori/consumatori

Un gruppo di problemi fondamentali nell'ambito della **programmazione concorrente** sono quelli conosciuti come **produttori/consumatori**: questo è un modello classico per lo studio della **sincronizzazione** nei **sistemi operativi** dove più processi devono coordinarsi per scambiarsi delle informazioni.

Il **produttore** è un **processo** che “genera una informazione” e la memorizza in un area di memoria condivisa dove un **processo** utilizzatore, il **consumatore**, che necessita di questa informazione per poter proseguire la sua evoluzione, la andrà a “prelevare”.



In questo problema individuiamo due possibili problematiche legate alla **sincronizzazione**:

- **problema di cooperazione:** si deve garantire che il **produttore** NON possa scrivere sul **buffer** se questo è pieno e che il **consumatore** non possa leggere due volte lo stesso messaggio:
 - **sequenza corretta** di accesso: inserimento-prelievo-inserimento-prelievo...
 - **sequenze errate** di accesso: prelievo-prelievo-inserimento...
- **problema di competizione:** il **buffer** è la struttura dati condivisa e quindi il **produttore** e il **consumatore** NON devono accedere contemporaneamente a esso: bisogna garantire la **mutua esclusione** nell'accesso al **buffer**.

■ Un produttore, un consumatore e una singola cella di memoria

Nella sua formulazione più semplice sono presenti solo due **processi**, il primo, il **produttore**, che vuole inviare informazioni scrivendo un dato (o un messaggio) in un'**area di memoria condivisa** (in questo esempio una singola variabile) e il secondo, il **consumatore**, che vuole prelevare i valori prodotti dal primo e “consumarli”.

ESEMPIO**Produzione di un dato e stampa singola**

Un tipico esempio è la gestione della stampa, dove vengono inviati i dati al processo che si occupa di stamparli. Il sistema è soggetto ai seguenti vincoli:

- il **produttore** può depositare un dato alla volta e non può scrivere un nuovo messaggio se il consumatore non ha ancora raccolto il precedente;
- il **consumatore** può leggere un dato alla volta e, naturalmente, solo dopo che sia stato prodotto dal processo **produttore** e non deve leggere due volte lo stesso valore;
- nel sistema non devono verificarsi situazioni di **deadlock**.

I due **processi** devono comunicarsi rispettivamente la disponibilità del dato da leggere e l'avvenuta lettura di dato in modo da dare la possibilità di produrre il successivo: ciascuno **processo** deve attendere un segnale da parte dell'altro **processo** per poter riprendere l'elaborazione.

Una possibile soluzione in pseudocodifica che utilizza un solo semaforo è la seguente:

```

semaphore accedi = 1      // all'inizio il buffer è vuoto (sem.verde)
int buffer;               // all'inizio vuoto
char aChiTocca = "P"     // P per il produttore C per il consumatore

Processo produci(int dato)
inizio
  P(accedi)              // quando il semaforo è verde
  if aChiTocca == "P"    // tocca al produttore
    buffer = dato
    aChiTocca = "C"      // aggiorna il turno
  V(accedi)              // segnala che il processo ha finito
fine

Processo consuma()
inizio
  P(accedi)              // quando il semaforo è verde
  if aChiTocca=="C"     // tocca al consumatore (il buffer è pieno)
    <consuma il dato>
    aChiTocca = "P"      // aggiorna il turno
  V(accedi)              // segnala che il processo ha finito
fine

```

La soluzione proposta è apparentemente corretta ma non rispetta il terzo requisito per l'accesso alla **SC** in quanto sono presenti **delay non necessari**: è richiesto che “un processo fuori dalla **sezione critica**” non deve ritardare l'ingresso nella **sezione critica** di un altro **processo** “avente più diritto di lui”.

La causa di questo problema è dovuta dal fatto che nella nostra soluzione vengono sempre risvegliati tutti i **processi** sospesi che, risvegliandosi, competono per accedere al buffer per vedere se tocca a loro “effettuare l'operazione”: facendo una analogia nella vita quotidiana, è come se noi andiamo continuamente a guardare nella casella di posta per vedere se è arrivato un documento.

Miglioriamo l'efficienza del sistema introducendo un vincolo temporale, cioè permettendo l'accesso alla **sezione critica** al **consumatore** solo dopo che un **produttore** ha inserito un dato:

- ▷ il **processo produttore** dovrà risvegliare solo il **consumatore**;
- ▷ il **processo consumatore** dovrà risvegliare solo il **produttore**.

In tale modo eliminiamo lo spreco di tempo e quindi i ritardi inutili all'accesso della **risorsa condivisa**.

Una possibile soluzione in pseudo codifica che utilizza **due semafori** è la seguente:

```

semaphore pieno = 0           // all'inizio il buffer non è pieno (sem. rosso)
semaphore vuoto = 1          // all'inizio il buffer è vuoto      (sem. verde)

int buffer;

Processo produci(int dato)
inizio
    P(vuoto)                  // quando il buffer è vuoto
    buffer = dato
    V(pieno)                  // segnala che il buffer è pieno
fine

Processo consuma()
inizio
    P(pieno)                  // quando il buffer è pieno
    <consuma il dato>
    V(vuoto)                  // segnala che il buffer è vuoto
fine

```

Il funzionamento è abbastanza semplice: vengono utilizzati **due semafori**, il primo che indica quando il buffer è pieno e quindi verrà settato dal **produttore**, il secondo che indica quando il dato è stato consumato, e sarà quindi settato dal **consumatore**: alla fine delle operazioni ogni processo deve avvisare l'altro del cambiamento di stato e lo farà resettando il **semaforo** opportuno:

▷ semaphore vuoto

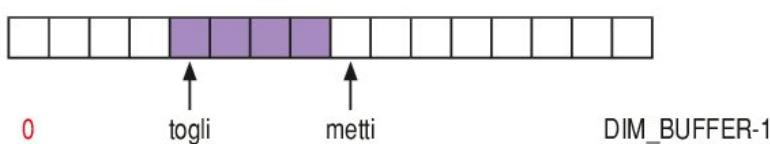
- **produttore** lo mette a 0 quando inizia a inserire un dato (il buffer è pieno)
- **consumatore** lo mette a 1 quando ha tolto un dato (il buffer è vuoto)

▷ semaphore pieno

- **produttore** lo mette a 1 quando ha inserito un dato (il buffer è pieno)
- **consumatore** lo mette a 0 quando inizia a togliere il dato (il buffer è vuoto)

Un produttore e un consumatore e buffer circolare

Nel caso più generale la memoria condivisa può contenere più dati (o messaggi) e viene realizzata per esempio con un buffer circolare implementato con un vettore di lunghezza **DIM_BUFFER**.



Introduciamo due puntatori, **metti** e **togli**, che individuano rispettivamente la prima porzione vuota e piena del buffer: inizialmente viene posto **metti = togli**.

I due puntatori **metti** e **togli** che si spostano attorno all'array realizzano una coda circolare FIFO con una dimensione massima prefissata (DIM_BUFFER): vengono utilizzati in modo ciclico, per esempio con l'operatore **% (modulo)**.

Una codifica del **processo produttore** è la seguente:

```
Processo produci(int dato)
inizio
    P(vuoto)                      // quando il buffer ha almeno uno spazio vuoto
        buffer[metti] = dato;
        // aggiornamento indice: quando si arriva DIM_BUFFER si ritorna a 0
        metti = (metti + 1) % DIM_BUFFER;
    V(pieno)                        // segnala che il buffer è pieno
fine
```

Una codifica del **processo consumatore** è la seguente:

```
Processo consuma()
inizio
    P(pieno)                      // quando il buffer è pieno
        <consuma il dato>
        toigli = (togli+1)%DIM_BUFFER;
    V(vuoto)                        // segnala che il buffer è vuoto
fine
```

Se utilizziamo i **semafori binari** questa soluzione però presenta un inconveniente: il produttore scrive il secondo dato solo dopo che il consumatore lo ha prelevato, rendendo di fatto inutile il buffer circolare.

Dato che ogni cella del vettore è una risorsa, siamo in presenza di risorse con molteplicità > 1 ed è quindi necessario modificare il semaforo di accesso al buffer sostituendolo con un **semaforo generalizzato** che "tenga il conteggio" dei posti liberi nel buffer, così che il produttore possa inserire successivi dati senza che vengano consumati (al limite ne potrà inserire DIM_BUFFER prima che il consumatore inizi a leggere); è inoltre necessario sostituire anche il secondo semaforo sempre con un **semaforo generalizzato** in modo da "ricordare" quanti nuovi dati non ancora letti sono presenti nel buffer.

È sufficiente introdurre la modifica dell'inizializzazione del programma:

```
semaphore vuoto = DIM_BUFFER      // all'inizio il buffer ha posti liberi
semaphore pieno = 0                // dati da leggere presenti nel buffer
buffer array[DIM_BUFFER]
```

Più produttori e più consumatori

Estendiamo ora al caso generale, dove possono essere presenti più **produttori** e più **consumatori**. Questa nuova situazione può comportare nuove necessità di **sincronizzazione** in quanto ora non è sufficiente la **sincronizzazione** appena descritta, poiché abbiamo due tipi di problematiche:

- **sincronizzare i produttori e i consumatori**, così come nel caso precedente, verificando che se qualunque dei produttori volesse produrre ma non ci fosse la disponibilità di un buffer libero, questo deve sospendersi e attendere che uno qualunque dei consumatori consumi un dato liberando così lo spazio di un buffer;
- **sincronizzare gli n produttori** fra di loro e gli m **consumatori** fra di loro perché in questa condizione anche le variabili togli e metti sono condivise e quindi divengono di fatto risorse comuni.

Se per esempio due produttori contemporaneamente effettuano la **P(V)** ed entrano nella **regione critica**, molto probabilmente scriveranno nella stessa cella ed eseguiranno in modo errato l'operazione:

```
metti = (metti + 1) % DIM_BUFFER;
```

Dobbiamo risolvere questo conflitto nei **produttori** così come lo ritroviamo simile anche nei **consumatori**, che potrebbero leggere contemporaneamente lo stesso dato.

Inizialmente definiamo i **semafori** e il **buffer** comune:

```
semaphore vuoto = DIM_BUFFER      // all'inizio il buffer ha posti liberi
semaphore pieno = 0                // dati da leggere presenti nel buffer
buffer array[DIM_BUFFER]
```

Per i **produttori** useremo quindi una procedura come quella di seguito riportata che, dopo aver accertata l'esistenza di almeno un posto libero nel buffer, **serializza** tra i **produttori** l'accesso alla variabile **metti** in modo da garantire la **mutua esclusione** al suo accesso da parte di coloro che richiedono di aggiornarla.

```
Processo produci(int dato)
semaphore mutexP = 1              // semaforo che regola i produttori
int tempo;                      // per salvare posizione corrente del buffer
inizio
P(vuoto)                         // quando il buffer ha spazio libero
P(mutexP)                          // blocca l'accesso alla RC agli altri produttori
tempo = metti                     // salva il valore corrente di metti
metti = (metti + 1) % DIM_BUFFER; // aggiornamento dell'indice
V(mutexP)                          // libera l'accesso alla RC agli altri produttori
buffer[metti] = dato              // inserisci il nuovo dato
V(pieno)                           // segnala che il buffer è pieno
```

La **serializzazione** avviene inserendo una **regione critica**.

Per i **consumatori** useremo quindi una procedura simile, di seguito riportata:

```
Processo consuma()
int tempo;
semaphore mutexC = 1           // semaforo che regola i consumatori
inizio
P(piemo)                      // quando il buffer è pieno
P(mutexP)                      // blocca l'accesso alla RC agli altri produttori
tempo = togli                  // salva il valore corrente di metti
togli = (togli+1) % DIM_BUFFER; // libera l'accesso alla RC agli altri produttori
V(mutexP)                      // segnala che il buffer è vuoto
<consuma il dato>
V(vuoto)                        // segnala che il buffer è vuoto
fine
```

Osserviamo come l'operazione di scrittura messaggio nel buffer non sia stata serializzata in quanto più produttori contemporaneamente devono poter fare il proprio "deposito": per serializzare anche questa operazione basta scambiare tra loro i semafori, cioè testare prima il mutexP.mutexC e successivamente i semafori pieno/vuoto.

Bisogna stare attenti a non precludere il **parallelismo** del riempimento del **buffer** da parte dei **produttori**, soprattutto se le operazioni di scrittura nel **buffer** sono piuttosto lente: se così fosse, tutti gli altri **produttori** rimarrebbero in attesa per lungo tempo anche in presenza di posti liberi, e anche tutti i **consumatori** rimarrebbero fermi, non avendo nulla da "consumare".

Per questo motivo sono state portate al di fuori della **regione critica** le operazioni di deposito e di prelievo dei dati.

Verifichiamo le competenze

1. Esercizi

- 1 Scrivi una funzione che restituisca il numero di celle correntemente occupate nel buffer, nell'ipotesi che non sia disponibile una primitiva per interrogare il valore dei semafori a conteggio. (Suggerimento: non effettuare incrementi circolari dei due indici togli e metti e gli accessi circolari alle celle del buffer).
- 2 In un sistema i tempi di produzione e consumazione sono casuali ma della medesima durata massima per produttori e consumatori.
Descrivi come cambia il tempo di attesa passiva sulle quattro invocazioni della primitiva P() al variare:
 - del numero di produttori;
 - del numero di consumatori;
 - delle dimensioni del buffer.
- 3 Si vuole realizzare un sistema produttori/consumatori dove le operazioni di inserimento e prelievo risultino non-bloccanti, ovvero dove l'operazione di inserimento su un buffer pieno e di prelievo da un buffer vuoto invece che causare la sospensione restituisca immediatamente un errore. Impostare una soluzione a questo problema utilizzando i semafori di Dijkstra.

Si consideri il seguente pseudocodice:

```

mutex = 1
 pieno = 0
 vuoto =10
 Processo P1(){
   while(true){
     elemento=produc();
     P(mutex);
     P(pieno);
     inserisci(elemento);
     V(vuoto);
     V(mutex);
   }
 }

 Processo P2(){
   while(true){
     P(mutex);
     P(vuoto);
     valore=rimuovi();
     V(pieno);
     V(mutex);
     consuma(valore);
   }
 }

```

I due processi concorrenti si passano i dati attraverso un buffer condiviso con il meccanismo del produttore/consumatore. Il semaforo 'mutex' è inizializzato a 1, il semaforo ' pieno' è inizializzato a 0 e 'vuoto' è inizializzato a 10, che è la dimensione del buffer.

Discuti il funzionamento della sincronizzazione e, nel caso ci sia un errore di programmazione o di inizializzazione, proponi una soluzione corretta.

Problemi "classici" della programmazione concorrente: lettori/scrittori

In questa lezione impareremo...

- ▶ le caratteristiche del problema lettori/scrittori
- ▶ a risolvere il problema lettori/scrittori

■ Problema dei lettori e degli scrittori

In un sistema sono presenti due gruppi di processi, i **lettori** e gli **scrittori**:

- ▶ i **lettori** non possono modificare i dati, ma solo prelevare in modo non distruttivo il contenuto di un buffer per elaborarlo;
- ▶ gli **scrittori** possono invece produrre un dato e quindi aggiornare il contenuto del buffer.

La situazione è diversa da quella analizzata nella lezione precedente dei **produttori/consuttori** in quanto in questo caso più **lettori** possono contemporaneamente utilizzare un dato senza “danneggiarlo”: quindi un lettore può accedere a un dato anche se è già presente un altro lettore, mentre lo scrittore deve accedere in **modo esclusivo** come nell'esempio della lezione precedente.

Esiste un vincolo di **mutua esclusione** fra **lettori** e **scrittori** nell'uso della **risorsa** e inoltre gli **scrittori** tra loro si contendono la **risorsa**, che quindi deve essere gestita in **mutua esclusione**.

Esistono diverse soluzioni a questo **problema**, in base alle ipotesi che vengono fatte sulle possibili situazioni di funzionamento.

Prima soluzione

Ipotizziamo che:

- ▶ i **lettori** possano accedere alla risorsa anche se ci sono **scrittori** in attesa;
- ▶ uno **scrittore** possa accedere alla risorsa solo se questa è libera.

Anche se l'ipotesi fatta sembra sensata, analizzandola attentamente ci accorgiamo che potrebbe portare a una situazione di **starvation** da parte degli **scrittori**, in quanto se i **lettori** continuano ininterrottamente a leggere, non potranno mai accedere in scrittura sul buffer.

Questa soluzione viene lo stesso implementata considerando il fatto che la **starvation** si verifica raramente anche considerando le velocità relative dei lettori rispetto agli scrittori, la frequenza delle operazioni e il tempo necessario a eseguirle.

Le procedure necessarie a realizzare la **sincronizzazione** di questo sistema sono quattro:

- ▷ una di richiesta lettura;
- ▷ una di fine lettura;
 - fra queste due si interporrà l'operazione di lettura vera e propria;
- ▷ una di richiesta scrittura;
- ▷ una di fine scrittura;
 - fra queste si interporrà l'operazione di scrittura.

Introduciamo una variabile `numLettori` necessaria per conoscere il numero di lettori che contemporaneamente stanno utilizzando il buffer comune, che sono cioè all'interno della **Regione Critica lettura/scrittura**: questa variabile deve essere modificata in competizione fra i lettori e per un suo corretto funzionamento deve essere manipolata (incrementata e decrementata) in **mutua esclusione** regolata dal semaforo **mutex**.

```
semaphore mutex = 1           // regola l'accesso alla variabile numLettori
semaphore sincro = 1          // regola l'accesso al buffer per scrivere/leggere
numLettori = 0                // numero di lettori all'interno della RC
```

```
procedura inizioLettura
inizio
  P(mutex)                   // mutua esclusione accesso al contatore
  numLettori++                // aggiornamento numero lettori entrati nella RC
  if (numLettori==1)          // se è il primo lettore che accede al buffer
    then P(sincro)            // blocca gli scrittori
  V(mutex)                    // libera l'accesso alla RC agli altri produttori
fine
```

Analizziamo il funzionamento dei due semafori, **mutex** e **sincro**:

- ▷ **mutex** serve per regolare i lettori che possono accedere contemporaneamente ai dati; per poter incrementare il contatore deve accedere in modo esclusivo e bloccare gli altri lettori;
- ▷ **sincro** serve per sincronizzare i lettori dagli scrittori: il primo lettore che riesce ad accedere al buffer deve verificare o meno la presenza di uno scrittore nella RC di lettura/scrittura e quando riesce ad accedere blocca gli solo gli scrittori mettendo a rosso il semaforo **sincro**; non blocca i successivi lettori perché questa istruzione viene eseguita solo quando si verifica `numLettori=1`.

```

procedura fineLettura
inizio
  P(mutex)           // mutua esclusione accesso al contatore
  numLettori--        // aggiornamento numero lettori entrati nella RC
  if (numLettori)==0  // nessun lettore dentro la RC
    then V(sincro)   // sblocca gli scrittori
  V(mutex)           // libera l'accesso al contatore ai lettori
fine

```

La funzione di rilascio è molto simile: **mutex** regola sempre l'accesso al contatore e quando l'ultimo **lettore** sta lasciando la RC di lettura/scrittura metterà a verde il semaforo **sincro** in modo da “liberare” l'accesso a chiunque voglia successivamente accedere ai dati.

Quindi **sincro** non viene sbloccato ogni volta che un **lettore** esaurisce il suo lavoro, ma solo quando *tutti* i **lettori** hanno terminato.

```

procedura inizioScrittura
inizio
  P(sincro)          // blocca sia scrittori che lettori
fine

```

```

procedura fineScrittura
inizio
  V(sincro)          // sblocca sia scrittori che lettori
fine

```

Le procedure di **inizioScrittura** e **fineScrittura** si realizzano con una sola istruzione sul semaforo **sincro**, che regola l'accesso al buffer di lettura/scrittura; quando un processo inizia a scrivere lo mette a rosso bloccando ogni altro accesso e quando finisce di scrivere lo mette a verde permettendo a qualsiasi altro processo di accedervi.

Queste funzioni devono essere chiamate all'interno dei processi che, nel caso di buffer circolare, devono inoltre provvedere anche alla sincronizzazione della risorsa finita, come descritto nell'esempio dei **produttori/consumatori**.

Poniamoci ora due domande:

1 quando un **lettore** rimane in attesa?

Un **lettore** si **sospende** per due motivi:

- A** perché c'è un processo **scrittore** che sta scrivendo;
- B** perché un altro **lettore** ha chiamato **inizioLettura** e quindi ha chiamato in causa la **regione critica** relativa alla variabile **numLettori**.

2 che cosa succede se entra prima un **lettore**?

Dipende dalla applicazione:

- A** se i **lettori** devono leggere qualcosa che “nessuno **scrittore** ha ancora scritto”, allora non funziona dato che lo **scrittore** non potrà mai accedere in quanto nella **RC** è presente un **lettore** che da essa non uscirà mai;

- B** se stiamo regolando l'accesso a un archivio già esistente, non ci sono problemi dato che “qualcosa” da leggere è già presente e quindi il **lettore**, prima o poi, lascerà la risorsa condivisa.

La soluzione è comunque sbilanciata a favore dei **lettori** dato che, potendo accedere contemporaneamente, potrebbero provocare una lunga lista di attesa degli **scrittori** che devono attendere che non sia presente più nessun **lettore**.

Seconda soluzione

Una seconda soluzione che migliora “la vita” agli **scrittori** e **lettori** è quella presentata di seguito dove uno **scrittore**, prima di terminare le sue operazioni, cerca di “passare il testimone” a un altro **scrittore** invece che ai **lettori**.

```
semaphore mutex      = 1          // regola l'accesso alla variabile numLettori
semaphore mutex2    = 1          // regola l'accesso alla variabile numScrittori
semaphore mutex1    = 1          // regola l'accesso degli scrittori uno alla volta
semaphore sincro   = 1          // regola l'accesso al buffer per scrivere/leggere
numLettori         = 0          // numero di lettori all'interno della RC
numScrittori       = 0          // numero di scrittori all'interno della RC
```

La procedura **inizioScrittura** è praticamente identica alla precedente procedura **inizioLettura** in modo da contare il numero degli scrittori che desiderano accedere alla RC.

```
procedura inizioScrittura
inizio
  P(mutex2)                      // mutua esclusione accesso al contatore
  numScrittori++                  // aggiornamento numero scrittori entrati nella RC
  if (numScrittori==1)            // se è il primo lettore che accede al buffer
    then P(sincro)                // blocca i lettori
  V(mutex2)                      // libera l'accesso al contatore altri scrittori
  P(mutex1)                      // blocca l'accesso alla RC agli altri scrittori
fine
```

Il semaforo **mutex2** serve per garantire la mutua esclusione sulla variabile **numScrittori** così come il semaforo **mutex** regolava l'accesso alla variabile **numLettori** nel caso prima descritto.

Il semaforo **sincro** ha lo stesso compito della prima soluzione, cioè realizza la **mutua esclusione** fra la ‘categoria’ lettori e la ‘categoria’ scrittori: l'unica differenza rispetto alla prima soluzione è che mentre i lettori possono essere contemporaneamente dentro la RC lettura/scrittura, gli scrittori devono susseguirsi uno alla volta, cioè in mutua esclusione.

Questo accesso serializzato viene realizzato con un ulteriore semaforo, **mutex1**, che viene posto a rosso quando il primo scrittore sta scrivendo e successivamente messo a verde dalla procedura di **fineScrittura**.

```
procedura fineScrittura
inizio
    V(mutex1)                      // libera l'accesso agli eventuali scrittori
    P(mutex2)                      // mutua esclusione accesso al contatore
    numScrittori--                  // aggiornamento numero lettori entrati nella RC
    if(numScrittori==0)             // nessuno scrittore ne dentro la RC ne in attesa
        then V(sincro)            // sblocca i lettori
    V(mutex2)                      // libera l'accesso al contatore agli scrittori
fine
```

Il problema ora si è duplicato: possono bloccarsi sia gli scrittori che i lettori, cioè abbiamo la possibile **starvation** per entrambe le categorie.

È necessario alternare lettori e scrittori per evitare la **starvation**:

- ▶ si deve rendere **impossibile** l'acquisizione senza limite dei processi **lettori**, anche se la risorsa è in possesso di un certo numero di **lettori**, se c'è almeno uno **scrittore** in attesa;
- ▶ si deve impedire agli scrittori di **passarsi la risorsa** tra loro, non considerando richieste di **lettori** già accordate.

Verifichiamo le competenze

1. Esercizi

1 Variante del problema lettori/scrittori

Si consideri una variante del problema dei lettori/scrittori con i seguenti due lettori:

```

Processo Lettore1(){
    while(true){
        P(mutex);
        nrlettori++;
        if(nrlettori==1)
            printf("primolettore");
        V(mutex);
        leggi_archivio();
        P(mutex);
        nrlettori--;
        if(nrlettori==0)
            printf("ultimolettore");
        V(mutex);
    }
}

Processo lettore2(){
    while(true){
        P(mutex);
        nrlettori++;
        if(nrlettori==1)
            printf("primolettore");
        V(mutex);
        leggi_archivio();
        P(mutex);
        nrlettori--;
        if(nrlettori==0)
            printf("ultimolettore");
        V(mutex);
    }
}

```

Descrivi il funzionamento indicando se il meccanismo è "equo" oppure privilegia una categoria di processi. Volendo riscriverla su di una piattaforma che non ha le primitive semaforiche, come si possono realizzare le sezioni critiche? Modifica il codice in tale senso.

2 Situazione: la Toilette Unisex

Un ristorante ha un'unica toilette sia per uomini che per donne: realizza un'applicazione concorrente nella quale ogni utente della toilette (uomo o donna) è rappresentato da un processo e il bagno come una risorsa. La politica di sincronizzazione tra i processi dovrà garantire che:

- nella toilette non vi siano contemporaneamente uomini e donne;
- nell'accesso alla toilette, le donne abbiano la priorità sugli uomini.

Si supponga che la toilette abbia una capacità limitata a N persone.

3 Situazione: ponte con utenti grassi e magri

A un ponte pedonale che collega le due rive di un fiume possono accedere due tipi di utenti: utenti magri e utenti grassi.

Il ponte ha una capacità massima MAX che esprime il numero massimo di persone che possono transitare contemporaneamente su di esso ma è talmente stretto che il transito di un grasso in una particolare direzione impedisce l'accesso al ponte di altri utenti (grassi e magri) in direzione opposta.

Realizza una politica di sincronizzazione delle entrate e delle uscite dal ponte che tenga conto delle specifiche date e che favorisca gli utenti magri rispetto a quelli grassi nell'accesso al ponte.

4 Situazione: BAR dello stadio

In uno stadio è presente un unico bar a disposizione di tutti gli spettatori che assistono alle partite di calcio. Oltre ai giornalisti, al bar accedono sia i tifosi della squadra locale che quelli della squadra ospite.

Il bar ha una capacità massima NRMAX di posti, che esprime il numero massimo di persone che il bar può accogliere contemporaneamente: per motivi di sicurezza, nel bar non è consentita la presenza contemporanea di tifosi di squadre opposte.

Il bar chiude per le operazioni di pulizia dopo dieci minuti dall'inizio della partita, riapre prima dell'intervallo, quindi chiude dopo dieci minuti dall'inizio del secondo tempo e riapre prima della fine della partita: nei periodi di chiusura potrebbero essere presenti alcune persone nel bar e in questo caso il barista attenderà l'uscita delle persone presenti nel bar, prima di procedere alla pulizia.

Scrivi le classi opportune necessarie per coordinare barista, giornalisti e tifosi delle opposte fazioni facendo in modo che si dia la precedenza di accesso al bar ai tifosi della squadra ospite.

5 Situazione: barbiere dormiente

In un salone lavora un solo barbiere, ci sono NRSEDIE=5 sedie per accogliere i clienti in attesa e una sedia di lavoro per il servizio dei clienti.

Il barbiere e i clienti sono processi e la poltrona è una risorsa, che può essere assegnata a un cliente per il taglio dei capelli, oppure utilizzata dal barbiere per dormire: infatti, se non ci sono clienti, il barbiere si addormenta sulla sedia di lavoro.

Un cliente quando arriva deve svegliare il barbiere, se addormentato, o accomodarsi su una delle sedie in attesa che finisce il taglio corrente. Se nessuna sedia è disponibile, il cliente preferisce non aspettare e lascia il negozio.

Dopo che un cliente ha occupato la poltrona, il barbiere esegue il taglio dei capelli e al termine:

- se ci sono clienti in attesa del proprio turno, riattiva il primo;
- altrimenti occupa la poltrona e si addormenta.

Scrivi le classi opportune necessarie per coordinare il barbiere e i clienti.

6 Situazione: travaso del vino

Un gruppo di persone collabora per travasare il contenuto di un grosso tino in due botti: solo Armando può prelevare vino dal tino e ogni collaboratore gli consegna il proprio contenitore vuoto (passando come parametro la sua capienza), aspetta che venga riempito e quindi si avvia verso le botti, per effettuare il travaso. Le persone in arrivo presso il tino devono attendere che Armando le serva e anche per versare il vino in una botte devono attendere in una coda, dato che una sola persona alla volta può versare il vino in una botte, fino a che questa è piena.

Le dimensioni del tino e delle botti vengono acquisite dal sistema come parametri, ma la somma delle botti è equivalente alla capienza del tino: quando Armando non ha nessun contenitore da riempire, si pone in uno stato di riposo.

Osservazione: se un trasportatore riempie la prima botte con una parte del contenuto che sta portando si mette in coda per terminare il travaso nella stessa botte.

Problemi “classici” della programmazione concorrente: deadlock, banchiere e filosofi a cena

In questa lezione impareremo...

- ▶ il concetto di deadlock
- ▶ a riconoscere le situazioni di deadlock
- ▶ a risolvere le situazioni di deadlock
- ▶ l'algoritmo del banchiere proposto da Dijkstra

■ Perché si genera un deadlock

Con **deadlock** (o **stallo**) indichiamo quelle situazioni nelle quali due (o più) **processi** si ostacolano a vicenda impedendo reciprocamente di portare a termine il proprio lavoro: il **deadlock** viene anche chiamato “abbraccio mortale” in quanto l'**interferenza** porta al fallimento di entrambi e... “alla loro morte”.



I **processi** “coinvolti” nel **deadlock** ostacolano anche gli eventuali altri **processi** presenti nel sistema, in quanto le **risorse** a loro allocate non possono essere utilizzate da nessun altro processo che le necessita per poter evolvere.

Non tutti i problemi di **concorrenza** possono portare a situazioni di **deadlock**: affinché ci sia un **deadlock** nel sistema devono verificarsi contemporaneamente **quattro condizioni** di seguito descritte, che sono state dimostrate da **Coffman** nel 1971 come le condizioni **necessarie e sufficienti** affinché esista la possibilità di verificarsi una situazione di **blocco critico** o **deadlock**.

Affinché ci sia un **deadlock** è necessaria la presenza di:

- 1 mutua esclusione**: le risorse coinvolte devono essere seriali, cioè ogni risorsa o è assegnata a un solo processo o è libera;
- 2 assenza di prerilascio (preemption)**: le risorse coinvolte non possono essere prerilasciate, ovvero devono essere rilasciate volontariamente dai processi che le controllano solo quando hanno terminato di usarle;
- 3 richieste bloccanti** (detta anche “◀ hold and wait ▶”): le richieste devono essere bloccanti, e un processo che ha già ottenuto alcune **risorse** può chiederne ancora altre;
- 4 attesa circolare**: devono essere presenti nel sistema almeno due processi e ciascuno di essi è in attesa di una risorsa occupata dall'altro.

◀ **Hold and wait** A process or thread holding a resource while waiting to get hold of another resource. ►

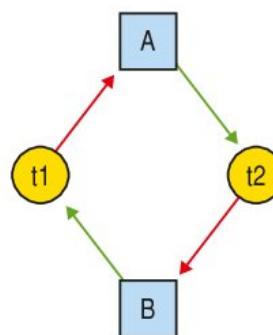


In questa lezione analizzeremo le tecniche che possono essere utilizzate per gestire il problema del **deadlock**.

■ Individuazione dello stallo

Per individuare le situazioni di **stallo** possiamo utilizzare i grafi di **Holt**, o grafi di allocazione (**Resource Allocation Graphs RAG**), descritti nella lezione 2 dell'unità di apprendimento 1: con questi è possibile verificare se una data sequenza di richieste-acquisizioni porta allo **stallo**.

Analizziamo la situazione riportata nel grafo della figura seguente:



Lo interpretiamo nel modo seguente:

- ▶ i due **processi** (o **thread**) **t1** e **t2** sono in attesa di bloccare rispettivamente la **risorsa** A e B;
- ▶ le risorse A e B sono allocate rispettivamente ai **thread** **t2** e **t1**.

Osserviamo come il **thread** **t1** richiede una **risorsa** A che è occupata da **t2** e il **thread** **t2** richiede una **risorsa** B che è occupata da **t1**!

Il **grafo di Holt** ci permette di individuare o escludere i **deadlock** grazie ai seguenti teoremi:



I TEOREMA SUL GRAFO DI HOLT

Se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili lo **stato è di deadlock** se e solo se il grafo di **Holt** contiene un ciclo.



II TEOREMA SUL GRAFO DI HOLT

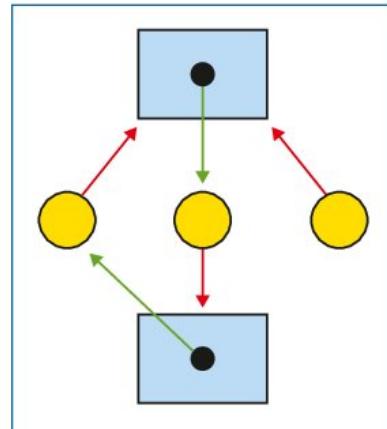
Se le risorse sono ad accesso mutualmente esclusivo, seriali e non prerilasciabili lo **stato non è di deadlock se e solo se** il grafo di **Holt** è completamente riducibile, cioè esiste una sequenza di passi di riduzione che elimina tutti gli archi del grafo.

Nel caso in cui le risorse sono presenti con molteplicità superiore a 1, la presenza di un ciclo nel caso di Holt non è condizione sufficiente per avere deadlock.

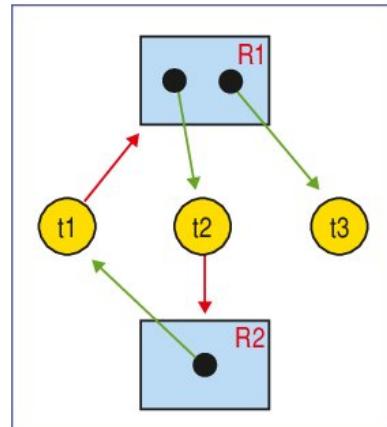
AREA digitale

Grafo di Holt e grafo di attesa

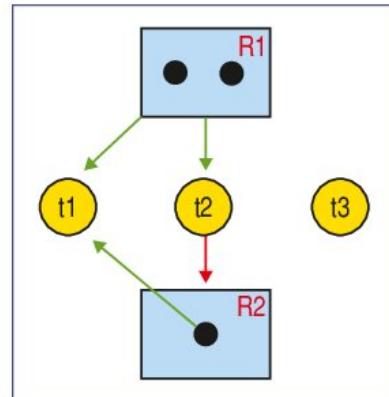
Lo possiamo verificare con i seguenti esempi:



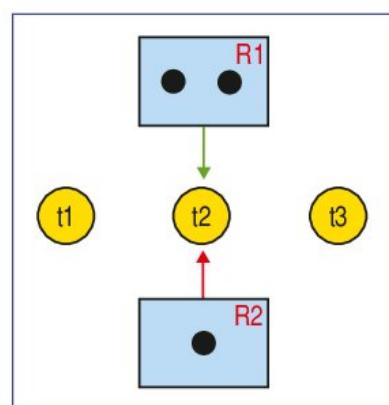
In questa situazione la presenza di **un ciclo** indica la *presenza di un deadlock*. ►



In questa situazione anche se è presente **un ciclo** non sussiste la situazione di **deadlock** in quanto possiamo ridurre il grafo, dato che il **thread t3** può evolvere e quindi prima o poi rilascerà la risorsa **R1** che verrà assegnata al **thread t1**: ▶

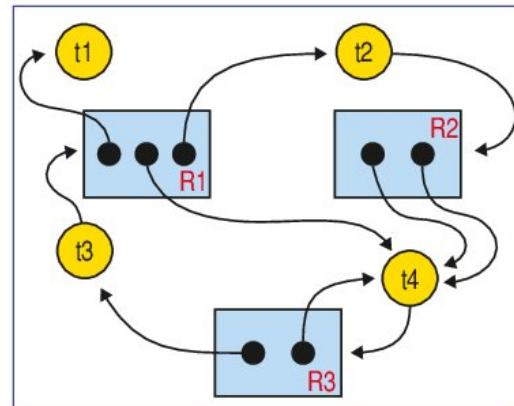


Questo potrà evolvere e rilasciare entrambe le risorse al **thread t2** che può portare a termine le sue elaborazioni. ▶

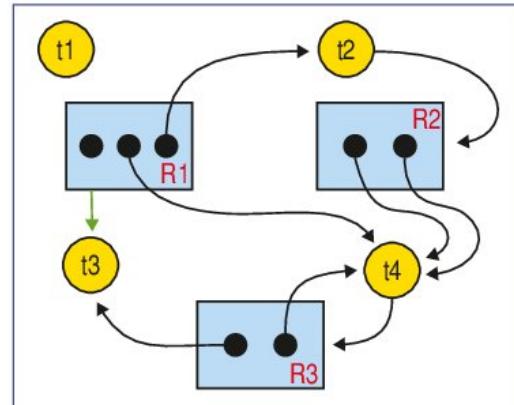


ESEMPIO Riduciamo un grafo-complesso

Analizziamo un esempio più articolato per scoprire se è possibile che si verifichi un **deadlock**. ▶



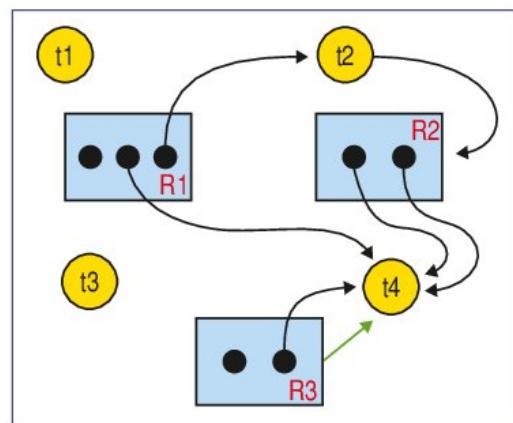
Per prima cosa osserviamo che il **thread t1** ha assegnato l'unica risorsa che richiede e quindi è in grado di portare a termine il proprio lavoro, quindi lo riduciamo: ▶



Dopo aver effettuato la riduzione di **t1**, possiamo riassegnare la **R1** da lui posseduta a **t3**.

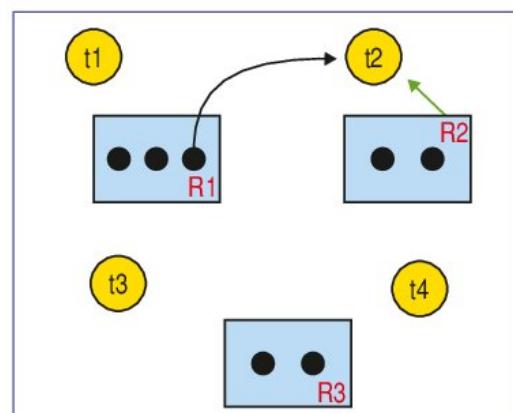
Ora è il **thread t3** che può evolvere e quindi lo riduciamo: ▶

Dopo aver effettuato la riduzione di **t3**, possiamo riassegnare la **R3** da lui posseduta a **t4**.

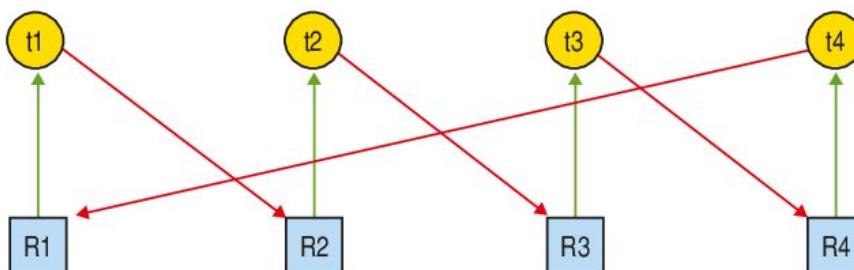


A questo punto anche il **t4** ha a disposizione tutte le risorse che gli necessitano, e lo possiamo ridurre: ▶

Dopo la riduzione di **t4** possiamo effettuare anche la riduzione di **t2**: quindi il grafo è stato completamente ridotto, come richiesto dal II teorema, e quindi possiamo affermare con certezza l'assenza dello stallo.



Le situazioni di **deadlock** possono coinvolgere anche più di 2 **thread**, come mostrato dal grafo riportato di seguito, dove esiste una situazione che coinvolge ciclicamente 4 **thread**.



■ Come affrontare lo stallo

Lo **stallo** può essere affrontato sostanzialmente in quattro diverse strategie:

- 1 detection e recovery**: riconoscerlo ed eliminarlo;
- 2 avoidance**: evitarlo con specifiche politiche di allocazione;
- 3 prevention**: impedirlo rimuovendo una delle quattro condizioni necessarie affinché si verifichi;
- 4 ignorare il problema**.

1 Individuare ed eliminare lo stallo: detection e recovery

Il sistema deve essere continuamente monitorato per riconoscere le situazioni **stallo**, utilizzando i **grafi di Holt**, e quando si individua un problema si interviene per eliminarlo.

La risoluzione delle situazioni indesiderate può avvenire in modo:

- 1 manuale:** è richiesto all'operatore di intervenire per risolvere la situazione a favore di uno dei processi in modo da sbloccare il sistema;
- 2 automatica:** il sistema operativo è in grado di risolvere automaticamente lo stallo applicando meccanismi opportuni che utilizzano specifiche politiche prestabilite.

Le soluzioni per entrambe le modalità sono le seguenti:

- A terminazione dei processi:** è il modo più semplice e allo stesso tempo più drastico per risolvere la situazione di **stallo**: si forza la terminazione di un **processo** alla volta (*terminazione parziale*) osservando se man mano viene risolta la situazione di stallo e, in caso negativo, si procede alla terminazione di tutti i **processi** (*terminazione totale*) e si sceglie quale deve essere il primo **processo** che deve essere fatto ripartire;
- B prerilascio di una risorsa (preemption):** viene forzato il prerilascio di una **risorsa** da parte di uno dei **processi in stallo** in modo che questa possa essere acquisita da un **processo** che, grazie a quella allocazione, possa evolvere; questa operazione, però, non può essere effettuata per tutti i tipi di **risorse** (si pensi per esempio a una stampante);
- C checkpoint/rollback:** viene fatto periodicamente il salvataggio su disco dello stato dei **processi** in determinati istanti ben definiti (**checkpoint**): in caso di **deadlock**, si ripristina (**rollback**) uno o più **processi** a uno stato precedente, fino a quando il **deadlock** non scompare.

Due osservazioni sui metodi sopra descritti:

- **terminare i processi** può essere costoso: per esempio se "uccidiamo" un **processo** che è in esecuzione da parecchio tempo verrebbe "ignorato e perso" completamente tutto il lavoro da esso effettuato; inoltre, se il processo viene terminato in una **sezione critica**, si rischia di lasciare le **risorse** in uno **stato inconsistente**;
- **fare preemption** non sempre è possibile in automatico e può richiedere **interventi manuali**.

2 Evitare lo stallo: avoidance

Si cerca di evitare lo **stallo** analizzando in anticipo l'utilizzo che il **processo** farà delle **risorse** che richiederà nella sua **evoluzione** in modo da controllare se tra queste operazioni può sorgere il pericolo del verificarsi di un deadlock: se questo può avvenire, si ritarda l'esecuzione di questo **processo**.

Per individuare se un processo può portare al **deadlock** si introduce il concetto di **stato sicuro** e **sequenza sicura** di esecuzione dei **processi**:



STATO SICURO

Un sistema è in uno **stato sicuro** (**safe**) soltanto se esiste una sequenza di completamento sicuro per i propri processi.

In questo caso si possono allocare le **risorse** per ogni **processo** della sequenza in un ordine preciso e continuare a evitare un **deadlock**.



SEQUENZA SICURA

Una sequenza di processi $\langle P_1, P_2, \dots, P_n \rangle$ è **sicura** se per ogni P_i le richieste di risorse che P_i può fare possono essere soddisfatte da:

- ▷ risorse attualmente disponibili;
- ▷ risorse detenute da tutti i processi che lo precedono nella sequenza.

Infatti se un **processo** richiede solo **risorse** che vengono utilizzate da **processi** che lo precedono nella sequenza, quando sarà il suo turno le risorse saranno disponibili per la sua esecuzione.

Se il sistema è in uno **stato sicuro** non ci sarà mai la possibilità di **deadlock**, mentre se non lo è questo potrebbe verificarsi: quindi per evitare il **deadlock** occorre assicurarsi che il sistema non entri mai in uno stato non sicuro.

Una possibile soluzione: l'algoritmo del banchiere

Esistono algoritmi che garantiscono la permanenza sempre in stati sicuri e tra tutti ricordiamo il cosiddetto **algoritmo del banchiere**, proposto da Dijkstra nel 1965.

Il nome deriva dal metodo utilizzato da un ipotetico banchiere che ha un capitale fisso e deve gestire un gruppo prefissato di clienti che richiedono del credito: non tutti i clienti avranno bisogno dello stesso credito simultaneamente ma tutti devono dichiarare in anticipo al banchiere la massima somma della quale hanno necessità che, necessariamente, deve essere inferiore al capitale posseduto dal banchiere e può essere richiesta in una sola volta o in più volte.

La traduzione informatica del problema è che un insieme di processi richiede al gestore il massimo numero di risorse in anticipo.

Le operazioni che possono eseguire i clienti sono due:

- Ⓐ chiedere un prestito una o più volte, ma con somma totale massima non superiore a quanto dichiarato in anticipo;
- Ⓑ restituire il prestito in un tempo finito.

Come si regola il banchiere per dare i prestiti in modo da riuscire a soddisfare tutti i clienti e facendo loro aspettare la disponibilità del denaro in ogni caso in un tempo finito?

Quando un cliente richiede un prestito, prima di concederlo il banchiere controlla se questa operazione può portare la banca in uno **stato sicuro** e solo in questo caso la richiesta viene accettata.

Per la banca si considera uno **stato sicuro** la situazione per cui i soldi rimanenti in cassa, dopo aver soddisfatto la richiesta corrente, permettono di soddisfare almeno una successiva richiesta massima da parte di un cliente che ancora non ne ha fatte: in questo modo si garantisce che sempre almeno un altro cliente possa essere servito completamente e quindi in un tempo finito restituirà i soldi in modo da poterli prestare agli altri clienti.

Si può facilmente verificare che se si soddisfa questa condizione l'insieme degli stati generano una **sequenza sicura**, e quindi è garantito che non ci saranno situazioni di **deadlock**.

In questo algoritmo avviene la richiesta di una risorsa singola: se invece si ipotizza che la banca possa fare prestiti in diverse valute si ottiene il caso generale di sistema con classi di risorse multiple.

Nei **sistemi operativi** questo algoritmo viene così applicato:

ogni processo deve dichiarare il massimo numero di risorse che gli sono necessarie e a ogni richiesta di una nuova risorsa l'algoritmo deve verificare cosa succede nel caso in cui venga soddisfatta questa richiesta, cioè se questa porta il sistema a uno stato sicuro o insicuro: quindi si calcola la quantità di risorse rimanenti nel caso che questo processo venga servito e si valuta se queste risorse possono soddisfare la massima richiesta di almeno un processo che ancora deve essere servito: in tal caso l'allocazione viene accordata, altrimenti viene negata.

3 Prevenire lo stallo: prevention

Con “**prevenzione dello stallo**” intendiamo le operazioni che ci permettono di evitare che si verifichi il **deadlock** intervenendo sulle quattro condizioni necessarie che lo provocano eliminandone una.

Le metodologie utilizzate per la prevenzione sono:

Eliminare la condizione di “risorse seriali”

Questa tecnica si realizza tramite strumenti che permettono di escludere la necessità di **mutua esclusione** permettendo la condivisione di risorse. Un esempio classico è quello che viene realizzato sulle stampanti mediante gli spool, dove ogni processo “crede” di avere la stampante ma in realtà si trova sempre in una situazione di attesa: il problema, di fatto, non viene eliminato ma “spostato” su di un’altra risorsa.

Inoltre questa tecnica non è di possibile realizzazione per tutte le tipologie di risorse.

Eliminare la condizione di “hold & wait”

Per escludere che un processo si impossessi di una risorsa e la mantenga occupata quando è in attesa di una seconda risorsa si provvede a effettuare quella che si chiama **allocazione totale**, cioè si impone che un processo richieda tutte le risorse all'inizio della computazione.

Anche questo meccanismo in generale non è sempre realizzabile in quanto spesso i processi non sanno dall'inizio di quali risorse necessitano e, inoltre, riservandole tutte per un processo si provoca l'arresto di altri processi che magari necessitano solo di una risorsa e potrebbero portare “indisturbati” a compimento il proprio lavoro: si riduce quindi il parallelismo introducendo problemi di possibile **starvation**.

Effettuare la preemption

Si obbliga un processo a rilasciare le risorse che possiede quando ne richiede un'altra che in quel momento non è disponibile e il processo sarà fatto ripartire soltanto quando può riguadagnare sia le risorse precedentemente possedute sia quelle che sta richiedendo.

Anche questa soluzione spesso non è realizzabile, come per esempio nel caso che la prima risorsa posseduta da un processo sia la stampante e il processo sia già a metà stampa: cosa potrebbe succedere se fosse forzato a cederla?

Con la prevention il deadlock viene eliminato strutturalmente.

Una possibile alternativa è quella di controllare quando un processo richiede una risorsa occupata se il processo che la sta occupando in quel momento stia evolvendo oppure attendendo a sua volta una ulteriore risorsa: solo in questo caso si forza il rilascio così da sbloccare il nuovo processo.

Eliminare la condizione di attesa circolare

Si introduce il concetto di **allocazione gerarchica** delle risorse attribuendo alle classi di risorse dei valori di priorità e imponendo che ogni processo in ogni istante possa allocare solamente risorse di priorità superiore a quelle che già possiede: se invece ha bisogno di una risorsa a priorità inferiore, deve prima rilasciare tutte le risorse con priorità uguale o superiore a quella desiderata.

In questo modo le risorse devono essere richieste seguendo un ordine prestabilito che viene opportunamente definito dall'amministratore del sistema facendo in modo che siano impossibili i deadlock.

Si può semplicemente verificare che questo meccanismo è efficace in quanto previene il deadlock ma introduce gravi rallentamenti portando il sistema a una alta inefficienza: l'indisponibilità di una risorsa ad alta priorità ritarda processi che già detengono risorse ad alta priorità.

4 Ignorare il problema

L'ultima soluzione per risolvere il problema dello stallo è quella di applicare l'**algoritmo detto "dello struzzo"**, cioè di "nascondere la testa sotto la sabbia" e di fare finta che non esista il problema, cioè ipotizzare che i **deadlock** non si possano mai verificare.

La motivazione dell'esistenza di "questa tecnica" sicuramente "poco scientifica" si basa sulla considerazione che spesso è troppo costoso mettere in atto le precauzioni sopra descritte e, basandosi su analisi statistiche, si preferisce ignorare il problema e affrontarlo poi solo nel momento in cui "in un caso remoto" potesse succedere (generalmente effettuando il **reset completo** di tutto il sistema).

Questa tecnica è quella utilizzata sia dal sistema operativo **Unix** che dalla **Java Virtual Machine** e da molti sistemi per basi di dati (**ORACLE**, **DB2**, **Informix**, **MySQL**).

Nei **sistemi transazionali**, dato che per motivi di efficienza non è possibile effettuare una transazione alla volta, cioè serializzare gli accessi, il **controllo di concorrenza** avviene in modo "**pessimistico**", cioè vengono assegnati dei **lock** sui dati:

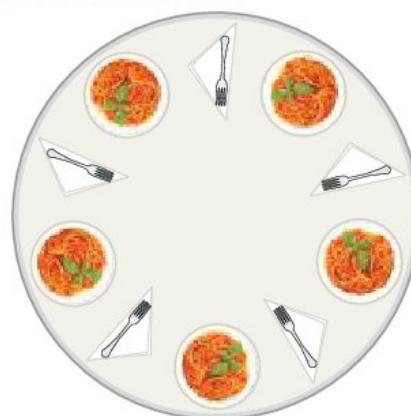
- ▷ i **read-lock** possono essere **condivisi**;
- ▷ i **write-lock** sono **esclusivi**.

I **lock** concessi sono memorizzati nella **tabella dei lock**: lo studio dettagliato della realizzazione della "transazione ben formata" rispetto al locking e delle anomalie di aggiornamento viene affrontato nei corsi di "**progetto dei database**".

■ Esempio classico: problema dei filosofi a cena

Un problema classico che doverosamente deve essere ricordato, sempre dovuto a **Dijkstra** che lo propose alla "comunità informatica" nel 1965, è quello che viene riportato col nome di "problema dei filosofi a cena".

Una possibile formulazione è la seguente:
cinque filosofi sono seduti attorno a un tavolo circolare, ciascuno di essi ha di fronte un piatto di spaghetti che necessita di **due forchette** per poter essere mangiato e sul tavolo vi sono in totale solo **cinque forchette** disposte come nel disegno: ▶



Ogni filosofo ha un comportamento ripetitivo, che alterna due fasi:

- ▶ una fase in cui pensa, lasciando le forchette sul tavolo;
- ▶ una fase in cui mangia, per la quale ha bisogno di avere in ciascuna mano una forchetta.

La prima considerazione che possiamo fare immediatamente è che i filosofi non possono mangiare tutti insieme: dato che ci sono solo cinque forchette solo due filosofi alla volta possono nutrirsi (altrimenti servirebbero dieci forchette).

La seconda è che due filosofi vicini di posto non possono mangiare contemporaneamente perché condividono una forchetta e, pertanto, quando uno mangia, l'altro è costretto ad attendere che i suoi vicini finiscano per potersi impossessare delle risorse.

Vediamo un esempio di funzionamento, supponendo che il filosofo quando ha fame e smette di pensare si comporti nel seguente modo:

- 1 come prima mossa prende la forchetta a sinistra del suo piatto;
- 2 quindi prende quella che è alla destra del suo piatto;
- 3 mangia finché è sazio;
- 4 quindi rimette a posto, sul tavolo, le due forchette.

Cosa succede se contemporaneamente i cinque filosofi prendono la forchetta di sinistra? I filosofi muoiono di fame, in quanto il sistema andrebbe in **deadlock**: ciascun filosofo rimarrebbe con una sola forchetta in mano in attesa di un evento che non si potrà mai verificare!

Una soluzione è quella precedentemente descritta della "allocazione totale": ogni filosofo verifica se entrambe le forchette sono disponibili e solo in questo caso le acquisisce contemporaneamente, altrimenti rimane in attesa; in questo modo non si può verificare deadlock in quanto è stata rimossa la condizione di **hold & wait**.



SPAGHETTI O RISO?

La versione del problema che è stata sopra riportata è quella originale dall'autore: la leggenda narra che durante un soggiorno in Italia, successivo alla sua pubblicazione, a **Dijkstra** venne insegnato a mangiare gli spaghetti con una sola forchetta e, quindi, la versione fu modificata introducendo filosofi orientali con piatti di riso al posto degli spaghetti da mangiare con le bacchette (**chopstick**) al posto di forchette!

Verifichiamo le conoscenze

1. Risposta multipla

1 Affinché ci sia un deadlock è necessaria la presenza di (indicare quella errata):

- a) mutua esclusione
- b) presenza di prerilascio
- c) preemption
- d) richieste bloccanti
- e) attesa circolare

2 Con risorsa seriale si intende:

- a) risorse richieste una dopo l'altra
- b) risorse disponibili in serie
- c) risorse allocate una dopo l'altra
- d) nessuna delle precedenti

3 Una richiesta bloccante è anche detta:

- a) "hold and wait"
- b) "look and wait"
- c) "look and signal"
- d) "keep and signal"
- e) "keep and wait"
- f) "wait and signal"

4 Quale delle seguenti affermazioni è falsa in merito ai grafi di Holt?

- a) il grafo di Holt contiene un ciclo se e solo se il grafo wait-for contiene un ciclo
- b) il I Teorema sul grafo di Holt dice quando uno stato è di deadlock
- c) il II Teorema sul grafo di Holt dice quando uno stato è di deadlock
- d) la presenza di un ciclo è condizione necessaria ma non sufficiente per avere un deadlock
- e) la presenza di un ciclo è condizione sufficiente ma non necessaria per avere un deadlock

5 Quale tra le seguenti non è una strategia per affrontare lo stallo?

- a) detection e recovery
- b) avoidance
- c) prevention e recovery
- d) prevention

6 I concetti di stato sicuro e sequenza sicura di esecuzione si introducono nella strategia di:

- a) detection e recovery
- b) avoidance
- c) prevention e recovery
- d) prevention

2. Vero o falso

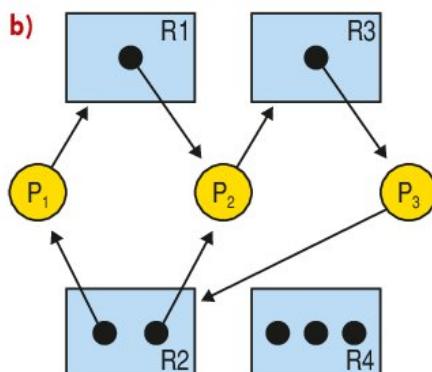
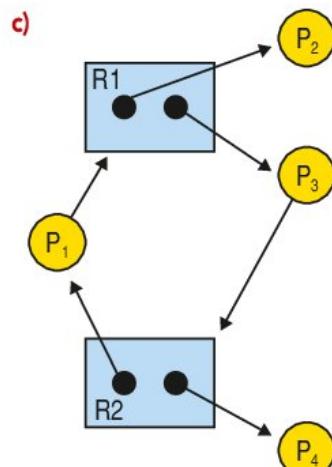
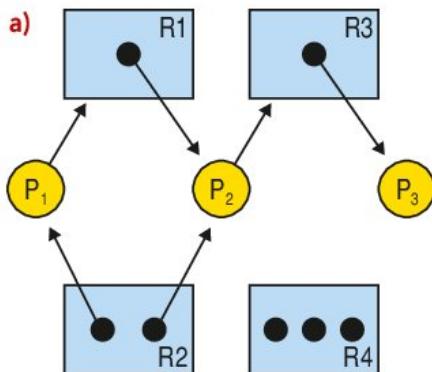
- 1 L'attesa circolare è condizione sufficiente al verificarsi del deadlock.
- 2 L'attesa circolare è condizione necessaria al verificarsi del deadlock.
- 3 La mutua esclusione è condizione sufficiente al verificarsi del deadlock.
- 4 Il deadlock si può verificare anche quando una sola delle condizioni necessarie è soddisfatta.
- 5 La situazione di deadlock è rappresentata dall'esistenza di un ciclo in un grafo di attesa.
- 6 Il Teorema sul grafo di Holt dice quando uno stato è di deadlock.
- 7 La terminazione totale è un metodo efficiente per eliminare lo stallo.
- 8 Nell'algoritmo del banchiere un processo può sperimentare un'attesa di durata indefinita.
- 9 Il rollback costituisce un metodo per la risoluzione del deadlock.
- 10 L'uso dei semafori garantisce il programmatore nei confronti del deadlock.
- 11 Due processi che non accedono mai a risorse comuni non possono mai essere coinvolti in una situazione di stallo.

V	F
V	F
V	F
V	F
V	F
V	F
V	F
V	F
V	F
V	F
V	F

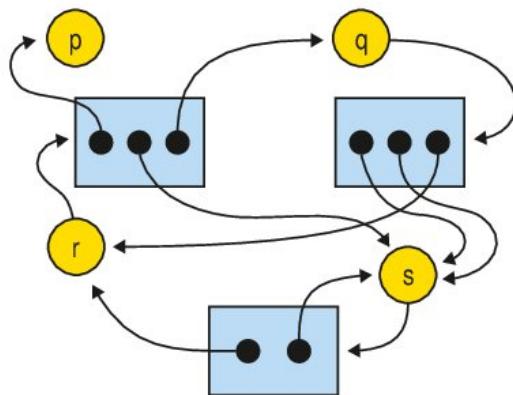
Verifichiamo le competenze

1. Esercizi

- 1 Nei seguenti grafi delle allocazioni è possibile che si verifichi un deadlock?
Determina la risposta individuando gli eventuali cicli presenti.



- 2 Nel seguente grafo delle allocazioni è possibile che si verifichi un deadlock?
Determina la risposta effettuando le opportune riduzioni.



- 3** Dati tre processi A,B,C che osservano il paradigma di "possesso e attesa" e tre risorse singole Q, R, S, utilizzabili in mutua esclusione e senza possibilità di prerilascio, supponiamo che il gestore assegni le risorse al processo richiedente la sola condizione che la risorsa sia disponibile.

Inizialmente tutte le risorse sono disponibili.

Si considerino le seguenti sequenze di richieste e rilasci:

Sequenza S1:

- | | |
|------------------|------------------|
| 1) A richiede Q; | 5) B richiede S; |
| 2) C richiede S; | 6) A rilascia Q; |
| 3) C richiede Q; | 7) A richiede S; |
| 4) B richiede R; | 8) C richiede R. |

Sequenza S2:

- | | |
|------------------|------------------|
| 1) A richiede R; | 5) A richiede Q; |
| 2) B richiede Q; | 6) B rilascia Q; |
| 3) C richiede S; | 7) B richiede S; |
| 4) C richiede R; | 8) A rilascia R. |

Queste sequenze provocano stallo? Motivare le risposte rappresentando nel grafo lo stato al termine di ciascuna sequenza.

- 4** Un sistema è dotato di 5 risorse (R,S,T,U,V) seriali, non prerilasciabili, contese da cinque processi (A,B,C,D,E): il sistema alloca le risorse disponibili al primo processo che le richiede.

La sequenza di richieste è la seguente:

A richiede R	C richiede T	B richiede R	D richiede V	E richiede R
A richiede S	A richiede T	E richiede U	C richiede U	D richiede T

Dopo l'ultima richiesta il sistema è in deadlock?

In caso affermativo dire quali sono i processi la cui terminazione forzata permetterebbe di far terminare correttamente il maggior numero di processi.

In caso negativo dire quale ulteriore richiesta porterebbe il sistema in deadlock.

Rappresentare lo stato del sistema con il grafo di Holt e motivare la risposta.

- 5** Algoritmo del banchiere multivaluta

Si estenda il problema del banchiere ipotizzando che questo debba fare prestiti usando valute diverse (euro, dollari, yen ecc.) e si discuta la politica di allocazione che garantisce l'assenza di deadlock.

- 6** Considerate i seguenti processi:

```

Risorse condivise
semaphore S=0, M=1;
int x = 1;

Processo P1{
    while (true) do
    begin
        down(M);
        x:=2*x;
        up(M);
        up(S);
    end;
}

Processo P2{
    while (true) do
    begin
        down(S);
        write(x);
        up(S);
        down(M);
    end;
}

```

- 1) I due processi possono andare in deadlock? Motivare la risposta.
- 2) Uno dei due processi può essere eseguito infinite volte mentre l'altro è bloccato? Motivare la risposta.
- 3) Descrivere tutti gli output del programma.

I monitor

In questa lezione impareremo...

- ▶ il concetto di monitor
- ▶ come utilizzare i monitor

■ Generalità

I **semafori** sono un meccanismo molto potente per realizzare la sincronizzazione dei **processi** ma il loro utilizzo è spesso molto rischioso e talvolta difficoltoso in quanto sono primitive ancora di “basso livello” e il programmatore può causare situazioni di blocco infinito (**deadlock**) o anche esecuzioni erronee di difficile verifica (**race condition**) per un errato o improprio posizionamento delle primitive di **P()** e **V()**.

Viene lasciata al programmatore troppa responsabilità nell’utilizzo di queste strutture di controllo così delicate.

Per ovviare a problemi di questa natura nei linguaggi evoluti di alto livello, come per esempio il **Concurrent Pascal**, **Ada** e **Java**, si sono introdotti costrutti linguistici “a più alto livello” per realizzare il controllo esplicito delle **regioni critiche**, dove è il compilatore che introduce il codice necessario al controllo degli accessi: questo meccanismo fu chiamato **monitor**, proposto da **Hoare** nel 1974 e da **Brinch-Hansen** nel 1975.

La definizione di **monitor di Hoare** è la seguente.



MONITOR DI HOARE

Costrutto sintattico che associa un insieme di procedure/funzioni (**public** o **entry**) a una struttura dati comune a più processi, tale che:

- ▶ le operazioni **entry** sono le **sole operazioni permesse** su quella struttura;
- ▶ le operazioni **entry** sono **mutuamente esclusive**: un solo processo per volta può essere attivo nel monitor.

Un **monitor**:

- ▶ protegge i dati da accessi poco strutturati;
- ▶ garantisce che i dati condivisi siano elaborati solo attraverso interfacce ben definite.

In altre parole il **monitor** definisce la **regione critica** e mette a disposizione sottoprogrammi che possono accedere a variabili e strutture dati interne a esso: un solo **processo** (o **thread**) alla volta può essere attivo entro il **monitor** e può quindi richiamare queste procedure (in questo caso si dice che il **processo** è nel **monitor**).

Un **thread** che richiama una procedura di **monitor**, quando un altro **thread** è già nel **monitor**, viene bloccato su un'opportuna coda associata al **monitor** e quando un **thread** all'interno del **monitor** si blocca, un altro **thread** deve poter accedere al **monitor**.

La struttura di un **monitor** è la seguente.

```
monitor <nome_monitor>{
    <dichiarazione delle variabili locali private>
    <inizializzazione delle variabili locali>

    /* definizione delle funzioni e procedure entry*/
    procedura entry1()          /* getter/setter delle variabili private */
        {...}
    procedura entry2()          /* getter/setter delle variabili private */
        {...}
    ...
    procedura entryN()          /* getter/setter delle variabili private */
        {...}
}
```

Possiamo riconoscere in questo meccanismo il concetto di **incapsulamento** proprio della programmazione **orientata agli oggetti**, dove per accedere agli attributi privati è necessario utilizzare metodi **setter/getter**.

L'inizializzazione delle variabili locali viene eseguita una sola volta prima dell'esecuzione di qualunque **procedura entry** e tali variabili mantengono il loro valore tra successive esecuzioni delle procedure del **monitor**, sono cioè **variabili permanenti**.

A queste variabili si accede solo mediante le procedure e le funzioni **entry** che sono definite entro il **monitor**: queste procedure potrebbero anche richiamare altre procedure sempre definite dentro il **monitor** ma non accessibili dall'esterno, chiamate **procedure non entry** (sono procedure non public).

In un **monitor** è attiva una sola procedura alla volta e questa proprietà è garantita dai meccanismi del supporto a tempo di esecuzione del linguaggio di programmazione corrente e il codice necessario è inserito dal compilatore direttamente nel programma eseguibile.

■ Utilizzo dei monitor

Il **monitor** viene utilizzato per effettuare il controllo degli accessi a una **risorsa condivisa tra processi concorrenti** in accordo a determinate **politiche di gestione**: le variabili locali definiscono lo stato della **risorsa** associata al **monitor** e i **processi** possono aggiornare lo stato della risorsa mediante le procedure **entry**.

Una istanza di tipo **monitor** (per esempio **miaRisorsa**) viene creata direttamente dopo la definizione del tipo **monitor** desiderato, come una qualunque variabile: nel nostro pseudocodice utilizziamo una annotazione **Java-like** e quindi la “manipoliamo” mediante la **dot notation**.

```
monitor TipoRisorsa          /* dichiaro il tipo monitor */
{
    <dichiarazione variabili e procedure del monitor>
}
TipoRisorsa miaRisorsa;      /* creo una istanza/oggetto di tipo monitor */
...
miaRisorsa.entry4()          /* chiamata di entry4 sull'istanza di monitor */
```

Dot notation The popular object-oriented programming languages (most notably C++ and Java) use the industry-standard “dot notation” to refer to objects and their properties. This notation takes the form of **objectname.variable**. ▶



La garanzia di **mutua esclusione** da sola può non bastare per consentire **sincronizzazione** intelligente; infatti l'accesso, e quindi l'assegnazione della risorsa, avviene secondo due livelli di controllo:

- 1 il **primo livello** è quello che garantisce la **mutua esclusione** facendo in modo che un solo **processo** alla volta possa eseguire le **entry** (funzioni pubblici) e quindi avere accesso alle variabili comuni del **monitor**: nel caso di richieste contemporanee effettuate da più processi mentre uno di essi è “dentro il monitor”, gli altri vengono sospesi e messi nella **entry queue**;
- 2 il **secondo livello** controlla l'**ordine** con il quale i processi hanno accesso alla **risorsa**: alla chiamata della procedura, se non viene verificata una condizione logica che assicura l'ordinamento, il **processo** viene sospeso, viene posto nella **entry queue** e viene liberato il **monitor**.

I **monitor** devono essere presenti “nativi” nel linguaggio di programmazione e sono stati progettati per sistemi con memoria comune; non funzionano in ambiente distribuito.

■ Variabili condizione e procedure di wait/signal

La **condizione di sincronizzazione** è costituita da variabili locali al **monitor** e da variabili proprie del **processo**, passate come parametri.

Nel caso in cui la condizione non sia verificata, la sospensione del processo avviene utilizzando variabili di un nuovo tipo, detto **condition variables** (condizione), che non sono dei semplici contatori ma rappresentano una **coda** nella quale i **processi** si **sospendono**.

Sulle variabili di tipo condition si accede solamente mediante due procedure, `wait()` e `signal()`, che hanno come parametro la variabile sulla quale devono operare.

Wait ()

L'invocazione dell'operazione `wait()` avviene mediante la notazione

```
miaCondition.wait() // forza l'attesa del processo chiamante
```

dove `miaCondition` è la variabile `condition` che deve essere testata: il processo che la esegue si sospende e viene posto nella coda associata a tale variabile e libera il monitor.

Signal ()

L'invocazione dell'operazione `signal()` avviene mediante la notazione

```
miaCondition.signal() // risveglia il processo in attesa
```

dove `miaCondition` è la variabile `condition` dove si vuole liberare un processo in attesa: se la coda a esso associata contiene almeno un processo, questo viene risvegliato e riprenderà l'esecuzione da dentro il `monitor`, a partire dall'istruzione seguente dalla `wait()` che lo aveva sospeso.

Se non sono presenti processi in coda l'operazione non ha nessun effetto.

L'esecuzione della `signal` deve essere l'ultima istruzione eseguita dal `processo` all'interno del `monitor` in modo che se viene risvegliato un processo in attesa sullo stesso monitor questo lo trovi libero.

La chiamata di una signal fa sì che almeno due processi possono evolvere contemporaneamente, quello che l'ha eseguita e quello risvegliato dalla coda: nella proposta di **Hoare** chi fa la `signal` si sospende immediatamente e va subito in esecuzione il processo appena risvegliato (`signal and wait`) mentre nella proposta di **Hansen** la scelta viene lasciata alla naturale gestione dello scheduler (`signal and continue`) semplicemente mettendo il processo che viene risvegliato nello stato di pronto.

Vediamo un primo esempio di utilizzo del `monitor` come allocatore di una `risorsa`.

ESEMPIO Monitor utilizzato come semaforo

Utilizziamo il `monitor` come un `semaforo`, che permette o meno l'accesso a una risorsa. All'interno del monitor sono presenti i dati condivisi che servono alle entry per regolare l'accesso a:

- ▶ una `variabile booleana` che rappresenta lo stato (libero, occupato) della risorsa;
- ▶ una `variabile condition` per gestire la coda di attesa.

Le funzioni (entry) `richiedi()` e `rilascia()` sono utilizzate solo per garantire l'accesso esclusivo alla risorsa da parte dei processi e la `mutua esclusione` tra le operazioni `richiedi()` e `rilascia()` fa in modo che lo stato della risorsa venga esaminato in modo mutuamente esclusivo dai processi.

Solo se un processo ottiene l'accesso tramite la `entry` può utilizzare la risorsa che è “fuori dal monitor” e, al termine del suo utilizzo, sempre tramite una entry, la rende disponibile agli altri processi.

Una pseudocodifica è la seguente:

```

monitor alocaRisorsa
    boolean occupato = false      // variabile privata del monitor (semaforo)
    condition libero            // variabile condition
    procedura entry richiedi()
        inizio
            if(occupato)           // se il semaforo è rosso
                libero.wait       // sospendo il processo in coda
                occupato = true     // metto il semaforo a rosso
        fine
    procedura entry rilascia()
        inizio
            occupato = false      // se il semaforo è verde
            libero.signal        // risveglio il processo in coda
        fine
    fine monitor

```

Osserviamo come la variabile **occupato** venga utilizzata come un **semaforo**, la entry **richiedi()** equivale alla esecuzione di un **P()** mentre la entry **rilascia()** a quella di un **V()**: l'accesso esclusivo viene però gestito ad alto livello e il programmatore non si deve preoccupare di realizzare la **mutua esclusione** ma semplicemente di richiamare queste entry all'interno dei suoi **processi**.

Scriviamo ora il segmento di codice che “utilizza la risorsa”

```

alocaRisorsa miaRisorsa;                      // istanza del tipo monitor
...
miaRisorsa.richiedi();                          // se nessuno usa la risorsa
< uso della risorsa>;                      // elaborazione in mutua esclusione
miaRisorsa.rilascia();                         // risvegli eventuali proc. in coda
...

```

Vediamo un secondo esempio di utilizzo del **monitor** come gestore di una **risorsa**.

ESEMPIO **Monitor per il problema dei produttori/consumatori**

Utilizziamo il **monitor** per risolvere il problema dei **produttori e consumatori** dove la risorsa condivisa è un buffer realizzato con un solo valore integer.

Per poter accedere alla variabile **buffer** mettiamo a disposizione due **entry**, **metti** e **togli**, definite all'interno del **monitor**.

La struttura del monitor è la seguente:

```

monitor ProducConsuma
    condition riempito, svuotato
    integer buffer

```

```

procedura entry metti(<dato>
inizio
    if buffer <> 0           // è presente un dato
        then svuotato.wait;   // si sospende nella coda
    <inserisci il dato>;    // quando è vuoto il buffer, mette il dato
    riempito.signal;         // risveglia il consumatore
fine

procedura entry preleva()
inizio
    if buffer == 0           // il buffer è vuoto
        then riempito.wait   // attendi che venga riempito
    <preleva dal contenitore> // togli il dato dal buffer
    svuotato.signal          // risveglia il produttore
fine
fine monitor

```

Scriviamo ora uno schema delle procedure **produttore** e **consumatore** che utilizzano il **monitor** appena descritto:

```

processo produttore()
inizio
    ...
    <produci dato>
    buffer.metti(<dato>)
    ...
fine

processo consumatore()
inizio
    ...
    <dato>=buffer.preleva()
    ...
fine

```

Svuotato è in pratica una **coda** dove gli scrittori attendono che il buffer venga "svuotato" dai lettori per poterlo riempire, e riempito è una coda di lettori in attesa di un dato da leggere.

■ Emulazione di monitor con i semafori

Nei linguaggi come il **C** che non hanno il costrutto **monitor** questo è possibile "costruirlo" mediante **semafori**, anche se non sempre la soluzione è di semplice realizzazione: per ogni istanza di **monitor** il compilatore deve assegnare un **semaforo mutex** per garantire la **mutua esclusione** all'accesso del **monitor** e un ulteriore **semaforo semVar** da associare a ogni variabile di tipo **condition** che viene definita all'interno del **monitor** da utilizzarsi come un contatore per tener conto del numero di **processi sospesi** su di esso.

Vediamo un esempio di codice per le funzioni di **wait()** e di **signal()** su di una variabile **var**:

```
wait(var)
inizio
contaVar++
V(mutex)
P(semC)
P(mutex)
fine

signal(var)
inizio
if(contaVar >0)
then
inizio
contaVar--
V(semS)
fine
fine
```

Il codice descritto è quello della proposta di **Hansen**, cioè **signal and continue**: la codifica per la proposta di **Hoare**, cioè **signal and wait**, è la seguente:

```
wait(var)
inizio
contaVar++
V(mutex)
P(semC)
fine

signal(var)
inizio
if(contaVar >0)
then
inizio
contaVar--
V(semS)
P(mutex)
fine
fine
```

Vedremo la loro completa implementazione in linguaggio C e alcuni esempi di utilizzo nelle apposite lezioni 6 e 7 di laboratorio.

Verifichiamo le competenze

1. Esercizi

- 1 Utilizzando i monitor realizza un meccanismo di sincronizzazione in modo tale che la risorsa condivisa venga assegnata a quello tra tutti i processi sospesi che la userà per il periodo di tempo inferiore.
- 2 Descrivi, utilizzando i monitor, il problema produttore/consumatore dove la variabile condivisa è un buffer circolare, a partire dal seguente schema di monitor:

```
Monitor buffer
{
    condition vuoto, pieno;
    int buffer[DIM];
    int contatore=0;
    int preleva=0, inserisci=0;

    entry riempি(int valore) {
        . . .
    }

    entry svuota() {
        . . .
    }
fine monitor
}
```

- 3 Mediante un monitor realizza la seguente politica di allocazione della risorsa:
 - 1) un nuovo lettore non può acquisire la risorsa se c'è uno scrittore in attesa;
 - 2) tutti i lettori sospesi al termine di una scrittura hanno priorità sul successivo scrittore.
- Utilizza il seguente schema di monitor:

```
monitor lettori_scrittori
{
    int num-lettori=0, occupato=0;
    condition ok_lettura, ok_scrittura;

    entry inizio_lettura() {
        . . .
    }
    entry fine_lettura() {
        . . .
    }
    entry inizio_scrittura() {
        . . .
    }
    entry fine_scrittura() {
        . . .
    }
fine monitor
}
```

Lo scambio di messaggi

In questa lezione impareremo...

- ▶ le tipologie dei meccanismi a scambio di messaggi
- ▶ le primitive send() e receive()

■ Generalità

Nel **modello ad ambiente locale** ogni **processo** può accedere esclusivamente alle risorse allocate nella propria memoria (virtuale) locale che non può essere modificata direttamente dagli altri **processi**.

Non avendo una **memoria condivisa**, i **processi** non possono utilizzare la memoria per il coordinamento delle loro attività e lo strumento di comunicazione e sincronizzazione diventa lo **scambio di messaggi**.

In questo modello ogni **processo** può accedere **esclusivamente** alle risorse allocate nella **propria memoria (virtuale) locale** e ogni risorsa del sistema è accessibile a **un solo processo** che può operare su di essa (**processo server**), eseguire le operazioni che gli vengono richieste dai processi applicativi (**processi clienti**) che ne hanno bisogno.

In questo modello architetturale di macchina concorrente il concetto di **gestore** di una risorsa coincide con quello di **processo server**.

Client e **server** comunicano tra loro esclusivamente interagendo con un **meccanismo di scambio di messaggi**.

Esistono due possibili modalità per implementare questo modello:

- Ⓐ utilizzare linguaggi che prevedono costrutti esplicativi per realizzare lo scambio di messaggi, come il **CSP** (Communicating Sequential Processes) proposto da **Tony Hoare**;

- B** utilizzare la “chiamata di procedura remota”, come il DP (Distributed Processes), proposto da Brinch Hansen.

In questa lezione descriveremo per ciascuno le caratteristiche essenziali e le primitive che ne permettono il funzionamento.

■ Canali di comunicazione

Prima di analizzare le modalità e le primitive che permettono di scambiare messaggi a due processi remoti, è necessario definire il collegamento logico mediante il quale due processi comunicano, che prende il nome di canale. Per poter inviare un messaggio un processo deve indicare quale canale il sistema operativo deve utilizzare per effettuare la trasmissione, quindi è compito del nucleo del sistema operativo fornire al processo come primitiva l’astrazione del concetto di canale, che, nel caso di un sistema distribuito, sarà la rete di comunicazione (per esempio Internet), mentre nel caso di un sistema multiprocessori può essere un bus locale.

Il programmatore non deve curarsi dell’hardware e quindi deve avere a disposizione costrutti linguistici ad alto livello che gli permettano di definire logicamente i canali, e linguaggi di programmazione che gli offrano le primitive che gli permettono di utilizzarli per programmare le varie interazioni tra i processi della sua specifica applicazione.

Un canale è caratterizzato da tre parametri:

- A** la tipologia del canale, intesa come direzione del flusso dei dati che un canale può trasferire, che può essere:
 - **monodirezionale**: il flusso di dati viene trasmesso in una sola direzione;
 - **bidirezionale**: viene usato sia per inviare che per ricevere informazioni.
- B** la designazione del canale e dei processi sorgente e destinatario di ogni comunicazione, che può essere:
 - nel caso di comunicazione **asimmetrica** il mittente nomina esplicitamente il destinatario ma questo non nomina esplicitamente il mittente:
 - **port**: canale **asimmetrico da-molti-a-uno**, i processi clienti specificano il destinatario delle loro richieste e il processo servitore è pronto a ricevere messaggi da qualunque cliente;
 - **mailbox**: canale **asimmetrico da-molti-a-molti** processi cliente che inviano richieste non a un particolare servitore, ma a uno qualunque scelto tra un insieme di servitori equivalenti;
 - nel caso di comunicazione **simmetrica** entrambi si nominano in modo esplicito:
 - **link**: canale **simmetrico da-uno-a-uno**;
- C** il tipo di sincronizzazione fra i processi comunicanti.
 - comunicazione **asincrona**;
 - comunicazione **sincrona**:
 - semplice o stretta;
 - estesa.

Comunicazione asincrona

Nel caso di comunicazione asincrona la comunicazione da parte del processo mittente avviene senza che questo rimanga in attesa di una risposta da parte del processo destinatario, quindi il processo mittente continua la sua esecuzione immediatamente dopo che il messaggio è stato inviato.

La spedizione del messaggio non è un **punto di sincronizzazione**, è quindi necessario utilizzare questo schema di interazione quando lo scambio esplicito di messaggi non deve contenere informazioni da elaborare, in quanto è complesso verificarne l'esito e comportarsi di conseguenza.

Comunicazione sincrona

Nel caso di comunicazione **sincrona** lo scambio di informazioni può avvenire solo se mittente e destinatario sono pronti a “parlarsi” e quindi è necessario che si sincronizzino, e questa interazione prende il nome di “**rendez-vous**”:

A rendez-vous semplice o stretto

Si limita alla trasmissione di un messaggio dal mittente al destinatario: il primo dei due processi comunicanti che esegue l'invio oppure è in necessità di ricevere un dato si sospende in attesa che l'altro sia pronto a eseguire l'operazione attesa. Il messaggio che viene inviato/ricevuto contiene informazioni corrispondenti allo stato attuale del processo mittente semplificando la realizzazione di meccanismi di sincronizzazione: questo modello è il **CSP (Communicating Sequential Processes)** proposto da **Tony Hoare**;

B rendez-vous esteso

In questo caso il destinatario, una volta ricevuto il messaggio, deve inviare una risposta al mittente e quindi il **processo mittente** rimane in attesa fino a che il ricevente non ha terminato di svolgere l'azione richiesta. Viene realizzato nel modello **client-server** mediante **RPC**, cioè **chiamata a procedura remota**; in questo caso il parallelismo risulta essere ridotto ma la **sincronizzazione tra i processi** è palese e quindi risulta semplice la verifica dei programmi. Utilizza questo meccanismo il **DP (Distributed Processes)**, proposto da **Brinch Hansen**.

■ Primitive di comunicazione asimmetrica da-molti-a-uno

Per dichiarare un canale possiamo per esempio utilizzare la seguente notazione sintattica:

```
port <tipo> <identificatore>;
```

Per esempio, se vogliamo definire un canale per trasferire messaggi che sono semplicemente numeri interi scriviamo:

```
port int canale1;
```

Con la parola riservata **port** si identifica un canale asimmetrico da-molti-a-uno nell' host ricevente: gli host mittenti, per inviare a questo un messaggio, utilizzano la **dot notation** come di seguito riportato:

```
<nome del processo>.<identificatore del canale>
```

Se sull'host dove si è definito il canale **canale1** si vuole inviare un messaggio al processo **processo12**, si scrive semplicemente:

```
processo12.canale1;
```

Primitiva di invio: send()

Per inviare un messaggio si utilizza la primitiva **send()** con la seguente notazione sintattica:

```
send(<valore>) to <porta>;
```

dove

- ▶ <porta>: identifica in modo univoco il canale a cui inviare il messaggio;
- ▶ <valore>: identifica il contenuto del messaggio, che deve essere dello stesso tipo definito dalla <porta>.

ESEMPIO **Invio di un messaggio**

Trasmettiamo come messaggio il numero 666 al **processo12** tramite il **canale1** dal quale il destinatario può ricevere.

```
send(666) to processo12.canale1;
```

La primitiva **send()** non è bloccante nella comunicazione asincrona mentre per realizzare scambi di messaggi sincroni è necessario che sia bloccante, cioè il processo che l'ha eseguita si sospenda in attesa di una risposta da parte del destinatario.

Primitiva di ricezione: receive()

Per ricevere un messaggio si utilizza la primitiva **receive ()** con la seguente notazione sintattica:

```
receive(<variabile>) from <porta>;
```

dove

- ▶ <porta>: identifica in modo univoco la porta dell'host ricevente sulla quale si attende il messaggio;
- ▶ <variabile>: è l'identificatore della variabile dove verrà posto il messaggio appena ricevuto, che deve essere dello stesso tipo definito dalla <porta>.

ESEMPIO **Ricezione di un messaggio**

Un processo è in attesa sul **canale1** di un messaggio di tipo intero che verrà memorizzato nella variabile **dato**:

```
receive (dato) from canale1;
```

Se non ci sono messaggi sul canale questa primitiva sospende il processo che l'ha chiamata, altrimenti preleva il primo messaggio dal canale e lo assegna alla variabile dato e inoltre memorizza in una apposita variabile un valore del tipo predefinito **process** che identifica il nome del processo mittente.

Vediamo un esempio completo di produttore/consumatore sincrono realizzato mediante le primitive appena definite.

```
Processo consumatore1
inizio
port int canale1;
.
.
receive (dato) from canale1;
.
.
fine

Processo produttore1
inizio
int messaggio
.
.
<produci messaggio>
.
.
send(messaggio) to consumatore1.canale1;
.
.
fine
```

La primitiva **receive()**, se non ci sono messaggi sul canale bloccante, costituisce un punto di sincronizzazione, ma introduce il problema **dell'attesa attiva** in quanto il **processo** ricevente rimane in loop in attesa che arrivi il messaggio.

Un problema può sorgere nel caso di un **host** in attesa di ricezione di diverse richieste di servizio e tra di queste deve privilegiarne alcune rispetto alle altre: è necessario introdurre più canali di ingresso, ciascuno dedicato a un tipo di richiesta diverso, e avere a disposizione la primitiva che permetta di indicare su quali canali attendere, per esempio in base dello **stato interno** della risorsa che il processo sta gestendo. Il meccanismo di **ricezione** ideale è il seguente:

- ▶ deve consentire al processo **server** di verificare la disponibilità di messaggi su un insieme di canali potendo anche assegnare a essi una priorità di scelta di ricezione;
- ▶ deve poter abilitare la ricezione di un messaggio da un qualunque **canale** non appena questo divenga disponibile, e nel caso di più messaggi pronti in contemporanea, lasciare al **processo** la scelta di quale privilegiare, in base alle proprie politiche di gestione;
- ▶ nel caso che nessun canale presenti messaggi disponibili deve bloccare il processo in attesa che arrivi un messaggio **su** di un canale qualsiasi.

■ Primitive di comunicazione asimmetrica da-molti-a-molti (cenni)

In questa situazione abbiamo molti **processi** che possono inviare messaggi a più destinatari e per realizzarla è necessario utilizzare una **mailbox** al posto dei **canali di comunicazione**. Le primitive possono essere così modificate:

- A per inviare un messaggio si utilizza la primitiva **send()** con la seguente notazione sintattica:

```
send(<mailbox>,<messaggio>)
```

- B per ricevere un messaggio si utilizza la primitiva **receive()** con la seguente notazione sintattica:

```
receive (<mailbox>,<messaggio>)
```

La realizzazione dei programmi **client/server** e la modalità di gestione delle porte sono oggetto di studio nel quinto anno di corso e saranno descritte nel terzo volume di questa collana.

dove

- ▶ <mailbox> indica il nome dell'area di memoria destinata a contenere i messaggi;
- ▶ <messaggio> indica il dato che viene trasmesso.

Verifichiamo le conoscenze

1. Completamento

sorgente • simmetrica • scambio di messaggi • destinatario • asincrona •
 CSP (Communicating Sequential Processes • DP (Distributed Processes) • asimmetrica •
 sincrona • simmetrico • asimmetrico • chiamata di procedura remota • asimmetrico

- 1** Esistono due possibili modalità per implementare questo modello:
 - a) utilizzare linguaggi che prevedono costrutti esplicativi per realizzare lo come il proposto da Tony Hoare;
 - b) utilizzare la come il proposto da Brinch Hansen.

- 2** Un canale è caratterizzato da tre parametri:
 - a) la **tipologia del canale**, intesa come direzione del flusso dei dati che un canale può trasferire, che può essere:
 - : il flusso di dati viene trasmesso in una sola direzione;
 - : viene usato sia per inviare che per ricevere informazioni.
 - b) la **designazione del canale e dei processi** e di ogni comunicazione, che può essere:
 - nel caso di comunicazione il mittente nomina esplicitamente il destinatario ma questo non nomina esplicitamente il mittente:
 - port: canale da a i processi clienti specificano il destinatario delle loro richieste e il processo servitore è pronto a ricevere messaggi da qualunque cliente;
 - mailbox: canale da a processi cliente che inviano richieste non a un particolare servitore, ma a uno qualunque scelto tra un insieme di servitori equivalenti;
 - nel caso di comunicazione entrambi si nominano in modo esplicito.
 - link: canale da a
 - c) il **tipo di sincronizzazione** fra i processi comunicanti.
 - comunicazione;
 - comunicazione:
 - semplice o;
 -;

2. Risposta aperta

- 1** Che cos'è un canale?
- 2** Come avviene la comunicazione asincrona?
- 3** Come avviene la comunicazione sincrona?
- 4** Che cosa si intende per rendez-vous stretto?
- 5** Che cosa si intende per rendez-vous esteso?
- 6** Quali primitive vengono utilizzate per realizzare la comunicazione mediante lo scambio di messaggi?
- 7** Che cosa si intende per RPC?
- 8** In che cosa consiste la proposta CSP di Hoare?
- 9** In che cosa consiste il meccanismo DP proposto da Brinch Hansen?

ESERCITAZIONI DI LABORATORIO 1

LA COMUNICAZIONE TRA PROCESSI MEDIANTE SEGNALI ASINCRONI

■ Comunicazione asincrona tra processi

I **processi** talvolta devono gestire eventi inaspettati o impredicibili quali errori run-time, come per esempio la divisione per 0, oppure richieste specifiche da parte di un utente come la terminazione (**<Ctrl-C>**) o la sospensione (**<Ctrl-Z>**) di un **processo** oppure, infine, la gestione di un **processo figlio**, cioè di come terminarlo in modo forzato oppure inviargli un particolare messaggio.

Tra questi eventi da gestire sono presenti anche situazioni accidentali impreviste come per esempio la mancanza di corrente elettrica o la rottura di un componente hardware.

Per la gestione di **eventi asincroni** di questo tipo **Unix** offre il meccanismo dei **segnali**: sono chiamati **interrupt software** e la loro elaborazione può comportare l'interruzione del flusso regolare del **processo**.

Ogni **segna**le ha un identificatore numerico e nella libreria **signal.h** vengono definite delle costanti simboliche **SIGxxx** che identificano i vari **segnali**.

Nei programmi che utilizzano i **segnali** è necessario quindi includere la libreria

```
#include <signal.h>
```

ESEMPIO

- | | |
|----------------|--|
| SIGINT | rappresenta un messaggio di interrupt |
| SIGTTIN | rappresenta un messaggio di terminazione per l'input |
| SIGALRM | suona la sveglia! |

Il numero di **segnali** predefiniti varia da 15 a 35 a seconda della versione di **Unix**: in allegato ne riportiamo una tabella.

AREA digitale



Tabella dei segnali UNIX/
LINUX

Quando il **SO** si rende conto che si è verificato l'evento associato a un segnale **SIGxxx** invia il **segnale** in questione al **processo** destinatario che attiva il bit corrispondente al segnale in un vettore di bit (**bitmap**) presente nel descrittore del processo (**PCB**) appositamente predisposto per memorizzare i “**segnali pendenti**”.

L'invio del **segnale** quindi consiste nell'attivazione del bit corrispondente nel **bitmap** e il **kernel** verifica la presenza o meno di “**segnali pendenti**” per un processo quando questo passa da **kernel** a **user mode** oppure quando effettua un passaggio di stato che coinvolge la situazione di **sleep**, sia quando viene **sospeso** che quando viene **riattivato**.

I **segnali** non sono quindi immediati come gli **interrupt hardware** e quindi possono essere sentiti da un **processo** con un certo ritardo: tale situazione potrebbe creare problemi in caso di applicazioni in **real time stretto**.

Quando viene generato un **segnale** il **SO** esegue un **signal handler** che gestisce l'evento associato al **segnale** che lo ha generato e, al termine della sua esecuzione, riprende il flusso di controllo: il programmatore può associare una propria funzione personalizzata a ogni **segnale** e in questo caso il **SO** la manda in esecuzione al posto dell'**handler** di default.

AREA digitale

 Quando viene notificato/gestito un segnale

Gli **handler** di default stabiliti dal kernel eseguono una delle seguenti possibili quattro azioni:

- ▶ **terminare** il processo (dump o quit);
- ▶ **ignorare** il segnale e lo cancella (ignore);
- ▶ **sospendere** il processo (suspend);
- ▶ **riprendere** l'esecuzione del processo (resume).

Nel seguito noi riscriveremo gli **handler** in modo da sfruttare questa potenzialità per realizzare meccanismi di **comunicazione tra i processi**.

■ Condizioni che generano un segnale

Esistono tre possibili situazioni che generano un segnale:

1 pressione di tasti speciali sulla tastiera.

È possibile inviare un segnale a un **processo in foreground** premendo **<Ctrl-C>** o **<Ctrl-Z>** dalla tastiera: quando il driver di un terminale riconosce che è stata premuta una di queste sequenze di tasti invia un segnale a tutti i **processi** nel **gruppo del processo** in foreground:

- ▶ la sequenza **<Ctrl-C>** invia il segnale **SIGINT**;
- ▶ la sequenza **<Ctrl-Z>** invia il segnale **SIGSTP**.

AREA digitale

 Gruppi di processi

2 eccezioni hardware

La generazione di un interrupt hardware viene catturata dal **kernel** che invia un **segnale** corrispondente al **processo** che è in esecuzione in quell'istante: per esempio, la divisione per 0 invia il segnale **SIGFPE** mentre un errato riferimento alla memoria produce un segnale **SIGSEGV**.

3 invio segnali da shell o da processo

I **segnali** possono essere inviati a un **processo** o dalla **shell** dei comandi del SO digitando il comando **kill** oppure richiamando la funzione **kill()** all'interno del codice del programma utente scritto in linguaggio di programmazione C.

1) **kill [-<segnalet>] <pid>**: comando inviato dallo shell

ESEMPIO

Il comando

```
kill -INT 169
```

invia il segnale **SIGINT** al processo il cui **pid** è 169

Il comando

```
kill -1
```

visualizza l'elenco di tutti i segnali, come si può vedere dalla successiva immagine:

```

Paolo@PCwin8: ~
$ kill -1
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGEMT      8) SIGPPE       9) SIGKILL     10) SIGBUS
11) SIGSEGV    12) SIGSYS      13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGURG     17) SIGSTOP     18) SIGSTP      19) SIGCONT     20) SIGCHLD
21) SIGTTIN    22) SIGTTOU     23) SIGIO       24) SIGNCPU     25) SIGNFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGPWR      30) SIGUSR1
31) SIGUSR2     32) SIGRTMIN    33) SIGRTMIN+1   34) SIGRTMIN+2   35) SIGRTMIN+3
36) SIGRTMIN+4   37) SIGRTMIN+5   38) SIGRTMIN+6   39) SIGRTMIN+7   40) SIGRTMIN+8
41) SIGRTMIN+9   42) SIGRTMIN+10  43) SIGRTMIN+11  44) SIGRTMIN+12  45) SIGRTMIN+13
46) SIGRTMIN+14  47) SIGRTMIN+15  48) SIGRTMIN+16  49) SIGRTMAX-15  50) SIGRTMAX-14
51) SIGRTMAX-13  52) SIGRTMAX-12  53) SIGRTMAX-11  54) SIGRTMAX-10  55) SIGRTMAX-9
56) SIGRTMAX-8   57) SIGRTMAX-7   58) SIGRTMAX-6   59) SIGRTMAX-5   60) SIGRTMAX-4
61) SIGRTMAX-3   62) SIGRTMAX-2   63) SIGRTMAX-1   64) SIGRTMAX

```

2) **int kill(pid_t pid, int signum)**: comando inviato dal **codice C**.

Con questo comando un processo invia il segnale con valore **signum** al processo con **PID pid**.

ESEMPIO

```
kill(169, SIGINT)
```

invia il segnale **SIGINT** al processo il cui **pid** è 169, come il precedente comando inviato dallo **shell**.

4 condizioni software

I **timer software**, come quelli definiti con la funzione **alarm()**, segnalano la loro scadenza programmata inviando un messaggio al processo destinatario: anche se questa situazione non è “propriamente asincrona”, dato che è prevedibile in quanto prestabilita, il messaggio viene inviato con la modalità dei **segnali asincroni**.

Permessi

Un **processo** non può inviare messaggi a un qualunque altro **processo**: è possibile inviare il **segnale** nel caso che i processi mittente e destinatario hanno lo **stesso proprietario**, cioè se “appartengono” alla stessa linea gerarchica: devono avere lo stesso **pgid**.

Il comportamento di **kill(pid, signum)** varia a seconda del valore di **pid**:

- ▶ **pid > 0**: il segnale è inviato al processo **pid**;
- ▶ **pid = 0**: invia il segnale a tutti i processi nel gruppo del mittente;
- ▶ **pid = -1**: se il mittente ha per proprietario il **superuser**, invia il segnale a tutti i processi, mittente incluso mentre se il mittente non ha per proprietario un **superuser**, invia il segnale a tutti i processi nello stesso gruppo del mittente, con esclusione del mittente;
- ▶ **pid < -1**: invia il segnale a tutti i processi nel gruppo **pgid**.

La funzione restituisce valore **0** se invia con successo almeno un segnale, altrimenti restituisce **-1**.

■ Gestire i segnali con la funzione **signal()**

Abbiamo detto che a ogni **segnale** viene associato di default una routine di risposta che viene richiamata dal **SO** quando un **processo** riceve un segnale (**handler** del segnale).

La funzione **signal()** viene utilizzata per definire il comportamento di un processo (**myHandler**) in seguito alla ricezione di uno specifico segnale identificato dall’argomento **signum**, cioè proprio per definire “l’**handler personalizzato**”.

```
void (*signal (int signum, void (*myHandler)(int)) )
```

dove:

- ▶ il *primo parametro* della funzione **signal** è il numero **signum** del segnale da gestire;
- ▶ il *secondo parametro* può assumere uno dei tre seguenti valori:
 - **SIGN_IGN**: indica che il segnale dev’essere ignorato;
 - **SIGN_DFL**: indica che deve essere usato l’**handler** di default fornito dal nucleo;
 - l’indirizzo di una funzione ***myHandler** definita dall’utente che ritorna un valore intero.

In questo ultimo caso dopo la sua esecuzione viene installato nel **SO** un nuovo **myHandler()** come **signal handler** per il segnale con numero **signum**: se la chiamata ha successo restituisce il numero del precedente **signal handler** associato a **signum** altrimenti **-1**.

Dato che il nome di una **funzione** è un *puntatore a funzione*, una funzione in una dichiarazione di parametro formale viene interpretata dal compilatore come un puntatore: quindi nel prototipo di **signal()** si può togliere “*****” e la funzione può essere scritta semplicemente come nei seguenti esempi:

ESEMPIO

- 1 per ignorare un segnale

```
signal(SIGUSR1, SIGN_IGN) // da qui in poi viene ignorato il SIGUSR1
```

2 per definire un gestore del segnale

```
void myHandler (int segnale)
{
    ...
    printf("Ricevuto segnale %d \n", segnale);
    ... // exit() oppure si restituisce il controllo al chiamante
}
main()
...
signal(SIGINT, myHandler);
... // da qui in avanti il processo mette in esecuzione la
// funzione myHandler ogni volta che viene premuta la
// sequenza CTRL-C da tastiera
```

3 per ripristinare il gestore di default

```
...
signal(SIGINT, SIG_DFL);
... // da qui in avanti il processo ripristina l'azione
// di default ogni volta che viene premuta la
// sequenza CTRL-C da tastiera
```

Non tutti i segnali sono riprogrammabili: per esempio, per i segnali **SIGKILL** e **SIGSTOP** non è possibile modificarne il comportamento di default.

ESEMPIO

Scriviamo un programma dove intercettiamo la sequenza **<Ctrl-C>**, e quindi il segnale **SIGINT**, scrivendo un gestore ad hoc che non fa terminare il processo ma visualizza una scritta sullo schermo.

Il codice è il seguente:

```
segnali1.c
1 #include <stdio.h>
2 #include <signal.h>
3 void gestore(int segnale){
4     printf(.. ricevuto segnale %d\n", segnale);
5     sleep(2);           // qui non sento il Ctrl-C
6     printf(.. terminazione handler segnale %d\n", segnale);
7 }
8
9 void main(argc, argv){
10    int k;
11    signal(SIGINT, gestore); // sostituzione handler <Ctrl-C>
12    for(k = 1; k < 7 ; k++) { // cicli di ritardo
13        printf("%d\n", k );
14        sleep(1);
15    }
16 }
```

Mandiamolo in esecuzione e proviamo a inviare da tastiera sequenze <Ctrl-C> ottenendo un output simile a quello riportato di seguito.



```

E ~/ua2/c_segnali
Paolo@PCwin8 ~/ua2/c_segnali
$ gcc segnali1.c -o s1
Paolo@PCwin8 ~/ua2/c_segnali
$ ./s1
1
2
.. ricevuto segnale 2
.. terminazione handler segnale 2
3
.. ricevuto segnale 2
.. terminazione handler segnale 2
4
5
6
Paolo@PCwin8 ~/ua2/c_segnali
$ |

```

Continuando a inviare da tastiera sequenze <Ctrl-C> sicuramente qualche digitazione viene effettuata mentre è in esecuzione la funzione `gestore()` da noi realizzata.

È facile osservare come la digitazione dei tasti viene ignorata mentre è in esecuzione il codice del nostro `handler` ridefinito dato che il processo non viene terminato.



Zoom su...

DIFFERENZA TRA SLEEP() E PAUSE()

Il **linguaggio C** mette a disposizione due primitive particolari per "sospendere" temporaneamente l'esecuzione di un **processo**: la seconda che presentiamo si presta particolarmente a essere utilizzata per realizzare meccanismi di comunicazione/sincronizzazione tra **processi** in quanto sospende il **processo** fino a quando non gli perviene un **segnaletico**:

1 sospensione temporizzata

```
int sleep(numsecondi);
```

Sospende il **processo** per **numsecondi**, ma il **processo** riparte prima di **numsecondi** se riceve un **segnaletico**.

2 attesa di un segnale

```
int pause(void);
```

Sospende il **processo** per un tempo indeterminato in attesa che gli arrivi un **segnale**.



Prova adesso!

- Utilizzo di `signal()`
- Disabilitazione e abilitazione di <Ctrl-C>

Scrivi un programma che realizza un pezzo di "codice critico" rendendolo immune da interruzioni dovute a <Ctrl-C> (segnale **SIGINT**).

Il processo deve procedere nel modo seguente:

- ▶ salva il precedente valore dell'**handler**, indicando che il segnale deve essere ignorato;
- ▶ esegue la "regione di codice protetta";
- ▶ ripristina il valore originale dell'**handler**.

Per esempio, il risultato di una sua possibile esecuzione è il seguente:

```

E:\ ~\ua2/c_segnali
Paolo@PCwin8 ~\ua2/c_segnali
$ ./segna1
Qui sento messaggio da tastiera Ctrl-C!
Ora non mi interrompe il Ctrl-C !
Qui sento messaggio da tastiera Ctrl-C
Fine elaborazione!
Paolo@PCwin8 ~\ua2/c_segnali
  
```

Confronta la tua soluzione con quella presente nel file [segna1Sol.c](#).

■ Scambiare segnali tra processo padre e i figli

Vediamo ora il comportamento dei **processi** figli di fronte alla ridefinizione degli **handler** e di come è possibile utilizzare i **segnali** per comunicare tra **processi**.

Dopo una `fork()`, il **processo** figlio eredita le **politiche di gestione dei segnali** del padre.

Lo possiamo vedere dal semplice esempio seguente: eseguiamo una `fork()` e mettiamo sia il padre che il figlio in attesa di un segnale da **SIGINT** da tastiera:

```

segna12.c
 1 #include <signal.h>
 2 #include <stdio.h>
 3 void gestore (int signum) {
 4     printf ("-> il processo %d sente il segnale %d \n", getpid (), signum);
 5 }
 6 int main (void) {
 7     signal (SIGINT, gestore); /* handle di Ctrl-C */
 8     if (fork () == 0)
 9         printf("figlio PID %d - gruppo PGID %d aspetta ... \n", getpid (), getpgid ());
10     else
11         printf("padre PID %d - gruppo PGID %d aspetta ... \n", getpid (), getpgid ());
12     pause(); /* i processi rimangono in attesa di un segnale */
13 }
  
```

Digitando <Ctrl-C> otteniamo la seguente situazione:

```

Paolo@PCwin8 ~/ua2/c_segnali
$ gcc segnali3.c -o s2
Paolo@PCwin8 ~/ua2/c_segnali
$ ./s2
figlio PID 6796 - gruppo PGID 7156 aspetta ..
padre PID 7156 - gruppo PGID 7156 aspetta ...
-> il processo 7156 sente il segnale 2
-> il processo 6796 sente il segnale 2
Paolo@PCwin8 ~/ua2/c_segnali
$ 
```

Entrambi i **processi** sentono il **segnale** ed eseguono il gestore che abbiamo definito.

Gruppo di processi

Ogni processo è membro di un **gruppo di processi** al quale viene associato un identificatore unico chiamato **pgid** (process group ID). Abbiamo detto che un **processo figlio** eredita il **gruppo di appartenenza** dal padre, ma è anche possibile definire dei gruppi autonomi e cambiare gruppo a un **processo**.

La **system call** **setpgid()** permette al **processo** invocante di *cambiare gruppo* di appartenenza a un **processo**:

```
int setpgid(pid_t pid, pid_t pgrp_id)
```

assegna valore **pgrp_id** al **process group id** del processo con **PID pid**:

- ▶ se **pid** è 0, cambia il valore del **process group ID** del processo invocante;
- ▶ se **pgrp_id** è 0, assegna il **process group ID** del processo con **PID pid** al processo invocante. Restituisce 0 se ha successo mentre se fallisce ritorna -1.

Quando un processo vuole creare un **proprio gruppo di processi**, distinto dagli altri gruppi del sistema, tipicamente passa il proprio **PID** come argomento per **setpgid()**.

```
setpgid(0, getpid()) // assegna come PGID il proprio PID
```

Non bisogna confondere il **PPID** con il **PGID**:

- ▶ **getppid()**: ritorna il valore del **PID** del padre del **processo**;
- ▶ **getpgid()**: ritorna il valore del **ID** del **gruppo** al quale appartiene il **processo**.

ESEMPIO

Scriviamo un esempio dove un **processo** lascia il **gruppo** creandone uno personale: in tal modo verifichiamo che non riceve più segnali dal terminale, a differenza dell'esempio precedente.

```
segnali3.c
1 #include <signal.h>
2 #include <stdio.h>
3 void gestore (int signum) {
4     printf ("-> il processo %d sente il segnale %d \n", getpid (), signum);
5 }
```

```

6 int main (void) {
7     int i;
8     printf ("il processo padre ha PID %d e PGID %d \n", getpid (), getpgid (0));
9     signal (SIGINT, gestore);           /* modifico handler di SIGINT */
10    if (fork () == 0)
11        setpgid (0, getpid ());        /* il figlio viene messo in un altro gruppo */
12        printf ("il processo PID %d PGID %d aspetta ..\n", getpid (), getpgid (0));
13    for (i = 1; i <= 3; i++) {         /* aspetta per tre volte .. */
14        printf ("il processo %d e' vivo \n", getpid ());
15        sleep(2);
16    }
17    return 0;
18 }
```

Possiamo vedere che il **processo** padre sente il **segnale** da tastiera mentre il **processo** figlio non lo riconosce: inoltre, nel caso riportato nella immagine seguente, il processo padre termina mentre il processo figlio rimane vivo fino a che non lo si interrompe con un segnale **<Ctrl-C>** inviato dallo shell dei comandi.

```

E ~ /ua2/c_segnali
Paolo@PCwin8 ~/ua2/c_segnali
$ gcc segnali3.c -o s3
Paolo@PCwin8 ~/ua2/c_segnali
$ ./s3
il processo padre ha PID 7368 e PGID 7368
il processo PID 7368 PGID 7368 aspetta ..
il processo 7368 e' vivo
il processo PID 7104 PGID 7104 aspetta ..
il processo 7104 e' vivo
-> il processo 7368 sente il segnale 2
il processo 7368 e' vivo
il processo 7104 e' vivo
il processo 7368 e' vivo
-> il processo 7368 sente il segnale 2
Paolo@PCwin8 ~/ua2/c_segnali
$ il processo 7104 e' vivo
^C
Paolo@PCwin8 ~/ua2/c_segnali
$
```



Prova adesso!

- Utilizzo di **signal()**
- Modifica gruppo a un processo

Scrivi un programma dove viene creato un figlio e viene riscritto l'handler della tastiera **SIGTTIN** (segnale 21) facendo in modo che questo venga sentito dal figlio.

Devi ottenere come output qualcosa simile a quello riportato di seguito:

```

E ~ /ua2/c_segnali
Paolo@PCwin8 ~/ua2/c_segnali
$ gcc segnali4Sol.c -o s4
Paolo@PCwin8 ~/ua2/c_segnali
$ ./s4
Introduci una stringa : -> il processo 6896 sente il segnale 21
ciao
6896 dice che hai inserito ciao
Il padre 5284 ora puo' terminare
Paolo@PCwin8 ~/ua2/c_segnali
$
```

Suggerimento: "sposta" il figlio dal gruppo del processo di controllo del terminale. Confronta la tua soluzione con quella riportata nel file **segnali4Sol.c**

■ Segnali da padre a figlio

Utilizziamo ora il meccanismo dei **segnali** per controllare l'evoluzione dei figli creati da un **processo** padre; scriviamo un programma che crea due figli che entrano in un ciclo infinito e mostrano un messaggio ogni secondo; il padre esegue le seguenti operazioni:

- 1 aspetta 2 secondi;
- 2 sospende il primo figlio, mentre il secondo figlio continua l'esecuzione;
- 3 aspetta altri 3 secondi;
- 4 riattiva il primo figlio;
- 5 aspetta altri 2 secondi;
- 6 termina entrambi i figli.

I segnali che il padre invia ai figli sono:

- **SIGSTOP**: sospende il processo;
- **SIGCONT**: riattiva l'esecuzione di un processo sospeso;
- **SIGINT**: termina un processo.

Il codice del programma è il seguente:

```
segna15.c
1 #include <signal.h>
2 #include <stdio.h>
3
4 int main (void) {
5     int pid1;
6     int pid2;
7     pid1 = fork ();
8     if (pid1 == 0) {           // crea primo figlio
9         while (1) {           // ogni secondo ripete la scritta/
10            printf ("pid1 e' vivo \n"); // all'infinito
11            sleep (1);
12        }
13    }
14    pid2 = fork ();
15    if (pid2 == 0) {           // crea secondo figlio
16        while (1) {           // ogni secondo ripete la scritta
17            printf ("pid2 e' vivo \n"); // all'infinito
18            sleep (1);
19        }
20        sleep (1);           // 1 padre riposa 2 secondi
21        kill (pid1, SIGSTOP); // 2 sospende il primo figlio
22        sleep (4);           // 3 padre riposa 2 secondi
23        kill (pid1, SIGCONT); // 4 riattiva il primo figlio
24        sleep (2);           // 5 padre riposa 2 secondi
25        kill (pid1, SIGINT); // 6 Kill del primo figlio
26        kill (pid2, SIGINT); // 6 Kill del secondo figlio
27    return 0;
28 }
```

Mandandolo in esecuzione possiamo individuare facilmente l'intervallo di tempo dove il primo figlio è sospeso: ►

```
Paolo@PCwin8 ~/ua2/c_segnali
$ gcc segna15.c -o s5
Paolo@PCwin8 ~/ua2/c_segnali
$ ./s5
pid1 e' vivo
pid2 e' vivo
pid1 e' vivo
pid2 e' vivo
pid2 e' vivo => sospeso figlio 1
pid2 e' vivo
pid2 e' vivo
pid2 e' vivo
pid1 e' vivo
pid1 e' vivo
pid2 e' vivo
pid1 e' vivo
pid2 e' vivo
```

Per realizzare la **comunicazione tra processi** è comodo utilizzare i segnali **SIGUSR1** e **SIGUSR2**: questi non hanno un significato specifico e sono lasciati alla implementazione del programma utente: come azione di default, se non vengono ridefiniti, terminano il processo.

AREA digitale

 Impostare una sveglia



Prova adesso!

- Utilizzo di **signal()**
- Modifica gruppo a un processo

- 1 Scrivi un programma in cui un processo padre P crea anzitutto N figli F1, ..., FN (dove N è un argomento a linea di comando), quindi invia a ciascun figlio un segnale **SIGINT**. Ciascun figlio Fi deve eseguire una **sleep()** di un numero di secondi pari al proprio indice e quindi terminare.
Confronta la tua soluzione con quella riportata nel file **segnali6Sol.c**
- 2 Scrivi un programma dove un processo crea un figlio e, successivamente utilizzando **kill()** e **pause()**, padre e figlio si scambiano il controllo alternativamente:
 - ▶ **kill(<pid>, SIGUSR1)** viene utilizzato per inviare il segnale **SIGUSR1** ai processi;
 - ▶ **pause()** sospende il processo finché non gli arriva un segnale qualunque.

È necessario riscrivere l'**handler** di **SIGUSR1** che utilizzeremo per incrementare un contatore che ci permette di visualizzare quante volte questa funzione viene richiamata, che riportiamo di seguito:

```
segnali7Sol.c
1 #include <stdio.h>
2 #include<signal.h>
3 #define TANTI 5
4 int volte = 0; // contatore delle esecuzioni
5
6 void gestore() {
7     printf(" -> pid %d, ricevuto segnale %d volte\n", getpid(), ++volte);
8 }
```

Confronta la tua soluzione con quella riportata nel file **segnali7Sol.c**

Osservazioni:

- per quale ragione è necessaria la chiamata alla funzione **pause()**?
Prova a rimuoverla e guarda come si comporta il programma.
- successivamente ripristina la **pause()** e modifica il programma in modo che il padre a ogni ciclo mandi un numero di segnali al figlio pari a (**volte+1**).
Come ti aspetti che si comporti il figlio in risposta a questi segnali?

- 3 Scrivi un programma che scrive un messaggio su standard output ogni volta che riceve i segnali **SIGINT** o **SIGUSR1**: il programma non deve mai terminare spontaneamente. Lancia il programma e usando il comando **kill()** della shell prova a inviargli quei due segnali e, infine, a terminarlo con un altro segnale.

ESERCITAZIONI DI LABORATORIO 2

THREAD E SCHEDULAZIONE

■ Un risultato non atteso

In questa lezione vedremo un esercizio che offre lo spunto per fare una riflessione sulla schedulazione dei **processi** e dei **thread** ed evidenzia come sia necessaria la **sincronizzazione**.

L'obiettivo che ci proponiamo è quello di incrementare una variabile condivisa chiamata **globale** sia all'interno del codice del programma principale che all'interno di un **thread** da lui creato: scriviamo un programma in modo che entrambi la incrementino di una unità per 20 volte.

Nel codice, dopo aver incluso le solite librerie, definiamo la variabile **globale** che verrà modificata all'interno del **main** e del **thread**.

contatore1.c

```

1 #include <pthread.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 int globale;
```

Il programma che viene eseguito dal **thread** fa alcune operazioni locali, tra le quali anche quella di sospendersi per un secondo (e ne scopriremo la motivazione in seguito), prima di aggiornare il valore della variabile **globale**:

```

7 void *cosaFaThread(void *arg) {
8     int i, locale;
9     for (i = 0; i < 20; i++) {
10         locale = globale; /* lettura variabile globale */
11         locale = locale + 1;
12         printf(".");
13         fflush(stdout);
14         sleep(1);
15         globale = locale; /* incremento variabile comune */
16     }
17     return NULL;
18 }
```

Il **main()**, dopo aver definito il **thread**, lo crea con la funzione di riga 23 passandogli come parametro il codice che deve eseguire e quindi inizia a incrementare la variabile **globale**, introducendo una pausa dopo ogni incremento.

Sia il **main()** che il **thread** visualizzano sullo schermo un "simbolo" che permette all'utente di identificare la loro esecuzione, così da poterne osservare la loro **schedulazione**.

```

20 int main(void) {
21     pthread_t mioThread;
22     int i;
23     if (pthread_create(&mioThread, NULL, cosaFaThread, NULL)) {
24         printf("errore nella creazione del thread.");
25         abort();
26     }
27     for (i = 0; i < 20; i++) {
28         globale = globale+1;
29         printf("o"); /* "marca" di incremento fatto dal main */
30         fflush(stdout);
31         sleep(1);
32     }
33     if (pthread_join(mioThread, NULL)){
34         printf("errore nel join del thread.");
35         abort();
36     }
37     printf("\nglobale vale %d\n", globale);
38     exit(0);
39 }
```

Ora osserviamo alcune esecuzioni del programma riportate nella seguente immagine:

```

Paolo@PCwin8 ~/ua2/c_semafori
$ gcc contatore1.c -o conta
Paolo@PCwin8 ~/ua2/c_semafori
$ ./conta
o.o.o..00..00..0.0.0..0.0.00.0..0.
globale vale 21

Paolo@PCwin8 ~/ua2/c_semafori
$ ./conta
o.o.o..0.0.0.0.0..0.0..0.00.0.0.0..0
globale vale 22

Paolo@PCwin8 ~/ua2/c_semafori
$ ./conta
o.o.o..0.0.0..0..0..0.0..0.00.0..0.0..0
globale vale 21

Paolo@PCwin8 ~/ua2/c_semafori
$
```

Si può vedere che ogni esecuzione è diversa dalla precedente e che ciascuna fornisce un risultato finale errato, cioè il valore finale di **globale** è pari a 21, 22 oppure anche 23 contro il risultato atteso di 40!



Prova adesso!

- Schedulazione dei thread
- Condivisione di memoria

Esegui ora lo stesso programma sul tuo calcolatore in modo da avere una esecuzione dello stesso codice su di una macchina diversa: otterrai sempre sequenze di caratteri diverse per ciascuna esecuzione del programma.

Modifica i valori delle istruzioni di `sleep(x)` e prova a discutere i risultati inattesi ottenuti.

■ Problemi di schedulazione e di sincronizzazione

Osservando i risultati dell'esempio precedente possiamo individuare due tipologie di problemi:

- A** problemi dovuti alla **sincronizzazione**;
- B** problemi dovuti alla **schedulazione**.

Possiamo affermare che i problemi dovuti alla **sincronizzazione** sono la causa del risultato errato che si presenta sullo schermo: la variabile **globale** nel **main()** viene incrementata direttamente mentre nel **thread** questo avviene utilizzando una variabile **locale**, cioè dopo aver fatto “alcune istruzioni” inserite volutamente per generare un ritardo: il nuovo valore non viene generato direttamente sulla variabile globale ma questa, prima viene copiata in una variabile locale, quindi incrementata e, infine, dopo una pausa di 1 secondo, assegnata alla variabile **globale** che si **sovrappone** al valore che essa ha in quel momento, valore che nel frattempo è stato modificato dal programma principale.

Scrivendo programmi concorrenti che sfruttano i **thread** è doveroso evitare inutili effetti collaterali simili a quello appena visto: cosa si può fare per eliminare questo problema?

Effettuiamo un primo intervento: dato che il problema si verifica perché **globale** è stata copiata in **locale** possiamo provare a evitare l'uso di una variabile **locale** e “operare” direttamente la variabile **globale** nel **thread**

```

7 void *cosaFaThread(void *arg) {
8     int i;
9     for (i = 0; i < 20; i++) {
10         globale = globale + 1;
11         printf(".");
12         fflush(stdout);
13         sleep(1);
14     }
15     return NULL;
16 }
```

ma i risultati, anche se sono diversi, non producono il valore atteso di 40!

```

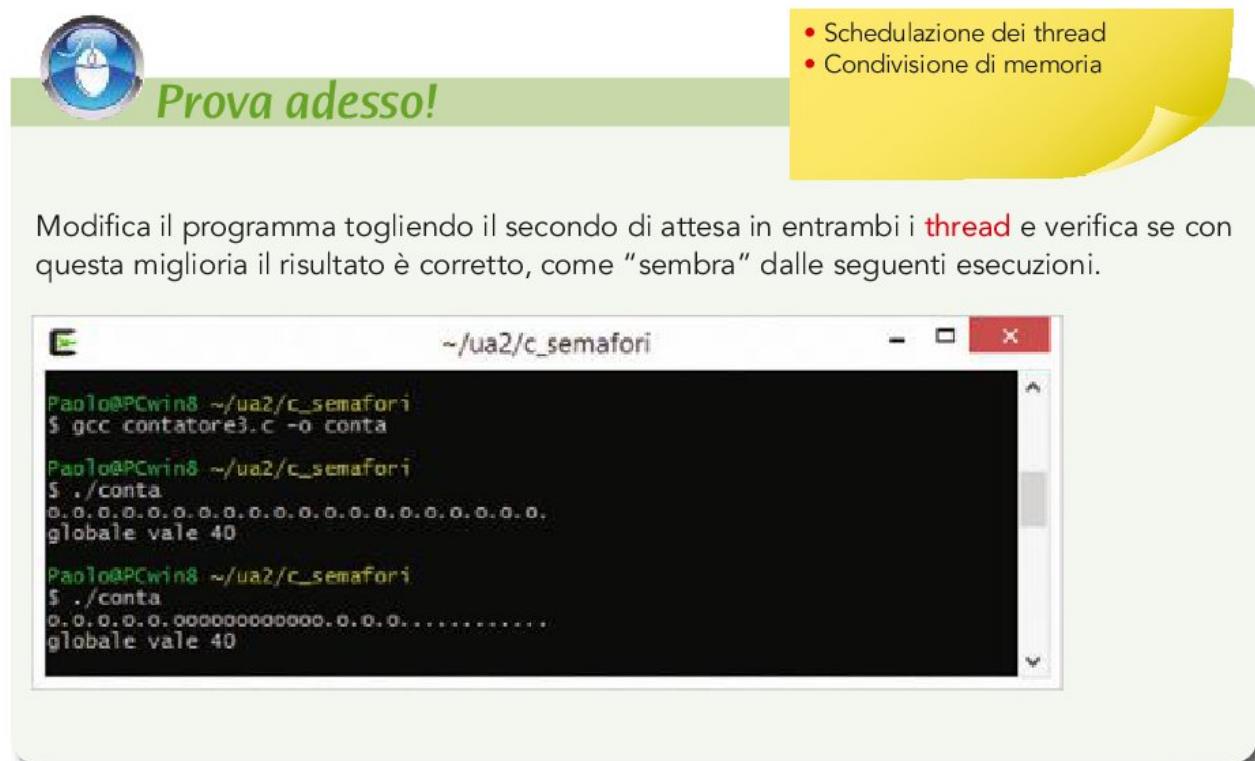
Paolo@PCwin8 ~/ua2/c_semafori
$ gcc contatore2.c -o conta
Paolo@PCwin8 ~/ua2/c_semafori
$ ./conta
0.0.0..0.0.0.0.0.0..0.0.0.0.0.
globale vale 38

Paolo@PCwin8 ~/ua2/c_semafori
$ ./conta
0.0.0..0.0.0.0.0..0.0..0.0.0.0.
globale vale 39

Paolo@PCwin8 ~/ua2/c_semafori
$ ./conta
0..0.0.0.0.0.0.0..0.0..0.0.0.0.0.
globale vale 38

```

Per giustificare questa situazione è necessario ricordare che i **thread** vanno in esecuzione in maniera simultanea e stiamo lavorando su di una macchina uniprocessore, dove il **kernel** realizza il **multitasking** assegnando volta per volta **time slice** di **CPU** ai diversi **processi**: possiamo immaginare che entrambi siamo in esecuzione contemporaneamente e mentre un **thread** “si riposa” per un secondo nel frattempo la variabile condivisa viene modificata dall’altro **thread**.



Con queste ultime rettifiche otteniamo il risultato “apparentemente corretto” se mandiamo in esecuzione il programma su macchine lente: ma anche su di esse, se appena si aumenta la complessità dell’operazione eseguita dal **thread** al posto del semplice incremento di una unità di una variabile, il problema si ripresenterà: lo si può verificare inserendo una semplice moltiplicazione (operazione di riga 12) come sotto riportato (oppure aggiungendo qualche ulteriore operazione algebrica a seconda della velocità del proprio processore).

```
7  void *cosaFaThread(void *arg) {
8      int i, x, y;
9      for (i = 0; i < 20; i++) {
10          globale = globale + 1;
11          x = globale;           /* inutili, solo per rallentare l'esecuzione */
12          y = globale * x;
13          printf(".");
14          fflush(stdout);
15      }
16      return NULL;
17 }
```

Per ottenere un risultato “sempre corretto” è necessario introdurre alcuni accorgimenti in modo che un **thread** comunichi all’altro di “bloccarsi” fintanto che non termina di effettuare tutte le operazioni sulle variabili condivise: è quindi necessaria la **sincronizzazione esplicita**.



Prova adesso!

- Schedulazione dei thread
- Condivisione di memoria

Scrivi un programma che crea due **thread** “**somma1**” e “**somma2**” che accedono alle variabili **datoA** e **datoB** di una struttura dati globale **condivisa** incrementandole rispettivamente di una unità per 5 volte e 10 volte.

Il valore finale delle due variabili verrà stampato a video dal **main()** che per prima cosa definisce e crea le due istanze del **thread**, passandogli come parametro la funzione che devono eseguire: il **main()** rimane quindi in attesa che entrambi i **thread** terminino la loro esecuzione mediante la funzione **pthread_join()** e come ultime operazioni visualizza sullo schermo i valori finali che assumono le due variabili.

```
somma1Sol.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define VOLTE 5           // costante usata per incrementare le variabili
4
5 struct dati {           // struttura Condivisa fra i thread
6     int datoA;
7     int datoB;
8 } condivisa;
```

Confronta il tuo codice con quello presente nel file **somma1Sol.c**.

Dato che anche in questo esercizio l’accesso alla struttura non viene regolamentato da alcun meccanismo di gestione della concorrenza, cosa ti aspetti come output?

Aggiungi un nuovo **thread** e manda in esecuzione il programma con tre **thread** in contemporanea.

Quindi modifica il codice delle operazioni eseguite da ciascun **thread** introducendo qualche operazione che richiede un maggior tempo di elaborazione (esempio una divisione oppure una radice quadrata): verifica che il nuovo output può avere “dei problemi”.

Come nell’esercizio precedente è necessario introdurre alcuni accorgimenti per fare in modo che un **thread** comunichi a tutti gli altri di “bloccarsi” fintanto che non abbia terminato di effettuare tutte le operazioni sulle variabili condivise: la soluzione avviene introducendo i **mutex** che permettono di gestire la **sincronizzazione**.

ESERCITAZIONI DI LABORATORIO 3

I SEMAFORI BINARI IN C



Info

I codici seguenti sono reperibili nel file **C_semafori.rar** scaricabile dalla cartella **materiali** nella sezione del sito www.hoepliscuola.it riservata al presente volume.

■ Realizzazione dei semafori binari

Il linguaggio C ha nella libreria **<pthread.h>** il concetto di **semaforo MUTEX** (Mutual Exclusion), un semaforo binario che ha valore **zero** se è **occupato** (rosso) e **uno** se è **libero** (verde).

Per prima cosa dobbiamo creare una variabile **semaforo**:

```
pthread_mutex_t semaforo;
```

è una variabile di tipo **semaforo** a cui sarà associato un valore (0,1) e una coda di **thread** sospesi sul **mutex** in attesa di “avere il verde”.

Il **semaforo** deve essere inizializzato prima di essere usato ed è possibile farlo in due modalità.

Staticamente

Al momento della creazione viene utilizzato il seguente codice per settarlo **occupato**, altrimenti, di default il **semaforo** è settato come **libero**:

```
pthread_mutex_t semaforo = PTHREAD_MUTEX_INITIALIZER;
```

Il significato cambia a seconda della opzione di inizializzazione, ma delle diverse possibilità noi utilizzeremo solo quella sopra indicata.

Le possibili forme di inizializzazione statica di un **mutex** sono:

- **PTHREAD_MUTEX_INITIALIZER**: il semaforo parte dallo stato **occupato**;
- **PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP**: crea un semaforo binario ricorsivo, ripetendo il tentativo di accesso alla sezione critica per ‘n’ volte, per cui poi dovremo fare l’unlock ‘n’ volte;
- **PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP**: se il semaforo è bloccato ritorna il codice di errore EDEADLK
- **PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP**: per i “fast mutex”.

Nella nostra trattazione noi utilizzeremo solo la prima costante, cioè inizializzeremo sempre il **semaforo** rosso contemporaneamente alla sua creazione.

Dinamicamente

È disponibile la seguente funzione di libreria che consente di inizializzarlo dinamicamente:

```
pthread_mutex_init (pthread_mutex_t *semaforo, pthread_mutex_attr *attributi)
```

La funzione di inizializzazione necessita di due parametri:

- ▶ un **semaforo**, cioè un puntatore a una variabile di tipo **semaforo**;
- ▶ gli attributi per l'inizializzazione del **semaforo**, non necessari per il settaggio di default (parametro **NULL**).

ESEMPIO Inizializzazione dinamica di un mutex

La seguente istruzione inizializza il semaforo libero

```
pthread_mutex_init(&bufferVuoto, NULL);
```

Funzioni P() e V()

Le funzioni che permettono di sospendere il **thread** in attesa che la **risorsa** associata al **semaforo** sia libera oppure per liberare il **semaforo** sono le seguenti:

```
int pthread_mutex_lock (pthread_mutex_t *semaforo) // P(S) o wait(S)
```

se il **semaforo** è occupato (ovvero è **zero**) il **thread** viene messo nella coda di attesa a esso associata, altrimenti occupa direttamente il **semaforo** e ritorna 0.

Per rilasciare la **sezione critica** e quindi liberare il **semaforo** si utilizza la funzione

```
int pthread_mutex_unlock (pthread_mutex_t *semaforo)
```

che provvede a porre il **semaforo** al valore uno (cioè verde) oppure, se ci sono **processi** in coda, ne risveglia il primo.

ESEMPIO Incremento di una variabile condivisa

Scriviamo un programma C in cui una variabile globale **count** viene incrementata di uno per 30 volte da un **thread** creato dal programma e poi incrementata di uno per 20 volte all'interno del **main()**.

Come meccanismo di gestione della **concorrenza** utilizziamo un **mutex** per garantire la “**mutua esclusione per sezione critica**”.

Dopo aver incluso le librerie e le costanti, definiamo le variabili globali che sono:

- ▶ una variabile intera **condivisa** che contiene il dato;
- ▶ il semaforo **mutex** che ci permette di regolare gli accessi a **condivisa**.

```

mutex1.c

1 #include <pthread.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #define TANTI1 30
5 #define TANTI2 20
6
7 int condivisa = 0; // variabile globale condivisa da main e thread
8 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // definizione semaforo
9
10 void *cod_thread(void *arg){ // funzione che viene eseguita dal thread
11     int x, dato;
12     for(x = 0; x < TANTI1; x++) {
13         pthread_mutex_lock(&mutex); // entra nella sez. critica
14         dato = condivisa; // aggiorna la variabile condivisa
15         dato++;
16         condivisa = dato; // copiandola prima in var. locale
17         printf(".");
18         fflush(stdout);
19         pthread_mutex_unlock(&mutex); // esce dalla sez. critica
20     }
21 }

```

Il **main()** crea il **thread** e successivamente compete con esso per l'aggiornamento della variabile **condivisa**:

```

23 int main(void) {
24     pthread_t tidi;
25     int x, err;
26     if((err = pthread_create(&tid1, NULL, cod_thread, NULL)) != 0) {
27         printf("errore nella creazione thread: %s\n", strerror(err));
28         exit(1);
29     }
30     for(x = 0; x < TANTI2; x++) {
31         pthread_mutex_lock(&mutex); // entra dalla sez. critica
32         condivisa++; // aggiorna direttamente condivisa
33         printf("o");
34         fflush(stdout);
35         pthread_mutex_unlock(&mutex); // esce dalla sez. critica
36     }
37     if(pthread_join(tidi,NULL)) {
38         printf("errore: %s\n",strerror(err));
39         exit(1);
40     }
41     printf("\nVariabile condivisa uguale a: % d\n",condivisa);
42     exit(0);
43 }

```

Riportiamo una esecuzione in **Cygwin** e una esecuzione in **Dev-cpp**:



```

Paolo@PCwin8 ~/ua2/c_semafori
$ ./mutex1
.....
Variabile condivisa uguale a: 50

```




```

C:\cygwin64\home\Paolo\ua2\c_semafori\mutex1.exe
.....
Variabile condivisa uguale a: 50

```

Scriviamo ora un secondo esempio dove vengono creati due **thread** che aggiornano una variabile condivisa finché questa non raggiunge il valore 10: per modificare le sequenze di esecuzioni introduciamo nei due **thread** un ritardo casuale, come si può vedere nella istruzione di riga 16.

```
mutex2.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #define TANTI 10
5 pthread_mutex_t mutex; // semaforo condiviso tra i threads
6 int buffer = 0; // variabile condivisa
7
8 void *cod_thread1 (void *arg) {
9     int accessi1 = 0; // num. di accessi del thread 1 alla sez critica
10    while(buffer < TANTI) { // fino a TANTI
11        pthread_mutex_lock(&mutex); // accesso sez. critica
12        accessi1++;
13        buffer++;
14        printf("accessi di T1: %d valore buffer: %d \n", accessi1, buffer);
15        pthread_mutex_unlock(&mutex); // rilascio sez. critica
16        sleep ((int) (5.0 * rand()/(RAND_MAX + 1.0))); // riposo casuale
17    }
18    pthread_exit (0);
19 }
```

Il codice del secondo **thread** è praticamente identico, tranne che le righe 9 e 14, mentre il **main()** si limita alla creazione dei **thread** e alla attesa della loro terminazione.

```
34 main(){
35     pthread_t tid1, tid2;
36     pthread_mutex_init (&mutex, NULL); // semaforo iniz. a verde
37
38     if (pthread_create(&tid1, NULL, cod_thread1, NULL) < 0) {
39         fprintf (stderr, "errore nella creazione del thread 1\n");
40         exit (1);
41     }
42     if (pthread_create(&tid2, NULL, cod_thread2, NULL) < 0) {
43         fprintf (stderr, "errore nella creazione del thread 2\n");
44         exit (1);
45     }
46     pthread_join (tid1, NULL);
47     pthread_join (tid2, NULL);
48 }
```

Una possibile esecuzione produce il seguente output:

```
Paolo@PCwin8 ~/ua2/c_semafori
$ ./mu2
accessi di T1: 1 valore buffer: 1
accessi di T2: 1 valore buffer: 2
accessi di T1: 2 valore buffer: 3
accessi di T2: 2 valore buffer: 4
accessi di T2: 3 valore buffer: 5
accessi di T1: 3 valore buffer: 6
accessi di T1: 4 valore buffer: 7
accessi di T2: 4 valore buffer: 8
accessi di T2: 5 valore buffer: 9
accessi di T1: 5 valore buffer: 10

Paolo@PCwin8 ~/ua2/c_semafori
$
```



Prova adesso!

- Utilizzo dei mutex
- Condivisione dei dati in memoria

Scrivi un programma C che crei due **thread** "somma1" e "somma2" che accedono entrambi alle variabili **test.a** e **test.b** di una struttura dati **test** condivisa incrementandole di 1 per 10 volte (20 volte).

Il valore finale delle due variabili verrà stampato a video dal main.

Realizzare come meccanismo di gestione della **concorrenza** quello della "**mutua esclusione per struttura**", dove la struttura **mutex** va allocata **dinamicamente**.

Confronta la tua soluzione con quella presente nel file **mutex3.c**

Scrivi un programma che crea due **thread** che dopo un intervallo di tempo casuale aggiornano metà elementi di un array condiviso inserendo il proprio **pid**.

Si rende necessario l'utilizzo di un lock in modo da avere l'accesso esclusivo all'array e l'aggiornamento dell'indice dell'array. Confronta la tua soluzione con quella presente nel file **mutex4.c**

ESERCITAZIONI DI LABORATORIO 4

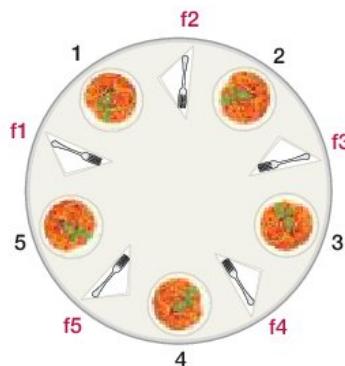
LA SOLUZIONE DEL DEADLOCK DEI FILOSOFI IN C CON I MUTEX

■ Filosofi e deadlock

Si vuole realizzare in linguaggio C la soluzione del problema dei 5 filosofi utilizzando i **semafori binari (mutex)**. Ciascun filosofo, numerato con un valore da 1 a N, può compiere tre attività:

- ▶ pensare
- ▶ mangiare
- ▶ aspettare, nel caso non trovasse disponibili le forchette, numerate per ciascun filosofo:
 - A** forchetta di destra = $(x + 1) \% N$
 - B** forchetta di sinistra = $(x - 1 + N) \% N$

Schematicamente nel caso di n = 5 filosofi abbiamo la situazione rappresentata in figura.



Associamo a questi tre stati dei valori costanti e indichiamo inoltre il filosofo di destra e quello di sinistra sempre con le due costanti DESTRA e SINISTA, che verranno comode nel seguito del programma sia per individuare i filosofi adiacenti che le posate da utilizzare:

```

5 #define N 5
6 #define PENSA 0
7 #define ATTESA 1
8 #define MANGIA 2
9 #define DESTRA (x + 1) % N
10 #define SINISTRA (x - 1 + N) % N

```

I dati condivisi sono le variabili che indicano lo stato dei filosofi: vengono memorizzate in un vettore e a ciascuna di esse si associa un semaforo per potervi accedere in mutua esclusione:

```
12 pthread_mutex_t mutex, mutex_f[N]; // semafori per RC e forchette
13 int stato[N]; // variabile condivisa con lo stato dei filosofi
```

Le due attività prevalenti dei filosofi sono quelle di pensare e mangiare; le simuliamo con due funzioni che fanno compiere queste azioni per una durata causale:

```
15 void pensa(int x){
16     printf("\nFILOSOFO %d: sto pensando ... ", x);
17     sleep(rand() % N); // pensa per un tempo casuale tra 0 e N
18 }
19
20 void mangia(int x){
21     printf("\nFILOSOFO %d: sto mangiando ... ", x);
22     sleep(rand() % N); // mangia per un tempo casuale tra 0 e N
23 }
```

Per sapere se il generico filosofo x -esimo può mangiare è necessario osservare lo stato dei filosofi adiacenti: se il filosofo di destra non sta mangiando e quello di sinistra non sta mangiando, sicuramente le forchette sono libere e quindi il filosofo corrente può mangiare; utilizziamo il semaforo associato al filosofo mettendolo a rosso quando inizia a mangiare e rilasciandolo quando termina di mangiare.

Quando il filosofo termina di mangiare richiamiamo la seguente funzione:

```
25 void posa(int x){
26     pthread_mutex_lock(&mutex); // accesso in mutua esclusione
27     stato[x] = PENSA; // aggiorna il suo stato
28     pthread_mutex_unlock(&mutex_f[x]); // libera la sua risorsa
29     pthread_mutex_unlock(&mutex); // rilascia la regione critica
30 }
```

Quando invece vuole iniziare a mangiare richiamiamo una funzione con una struttura simile alla seguente:

```
39 void prendi(int x){
40     pthread_mutex_lock(&mutex); // accesso in mutua esclusione
41     printf("\nFILOSOFO %d: ho fame e aspetto le forchette ... ", x);
42     stato[x] = ATTESA;
43     controlloPosate(x); // controlla e attende le posata
44     printf("\nFILOSOFO %d: ...ora prendo le forchette e mangio", x);
45     pthread_mutex_lock(&mutex_f[x]); // occupa la sua risorsa
46     pthread_mutex_unlock(&mutex); // rilascia la regione critica
47 }
```

La funzione `controlloPosate(x)` effettua i controlli sulle posate libere e si comporta di conseguenza.



Prova adesso!

Realizza la funzione controlloPosate(x) in modo da garantire il corretto funzionamento in assenza di starvation e deadlock; per simulare la "vita del filosofo" utilizzando la seguente funzione:

```

49 void *filosofo(void *x){
50     int k = (int) x;
51     while(1){
52         pensa(k);
53         prendi(k);
54         mangia(k);
55         posa(k);
56     }
57 }
```

Il sistema viene mandato in esecuzione dal seguente **main()**:

```

59 void main(){
60     int x;
61     srand((int) time(NULL)); // inizializza il seme x random
62     pthread_t filo[N]; // definizione dei thread
63     pthread_mutex_init(&mutex, NULL); // inizializzazione semafori
64     for(x = 0; x < N; x++)
65         pthread_mutex_init(&mutex_f[x], NULL);
66     for(x = 0; x < N; x++){
67         pthread_create(&filo[x], NULL, (void *)filosofo, (void *)x); // crea i thread
68         sleep(1);
69     }
70 }
```

Confronta la tua soluzione con quella presente nel file [filosofi.c](#)

Infine modifica il numero di filosofi e manda in esecuzione il programma: cosa puoi osservare?

ESERCITAZIONI DI LABORATORIO 5

LA SOLUZIONE DEL PROBLEMA PRODUTTORI/CONSUMATORI CON I SEMAFORI CLASSICI

■ Semafori in C

La prima versione di **POSIX** metteva a disposizione del programmatore solo due primitive per la **sincronizzazione** dei **thread** in processi multipli: **mutex** e le **variabili condizione**.

Con lo standard **IEEE POSIX 1003.1b** (1993) vengono introdotti come estensione real-time anche i **semafori classici**, mediante l'inclusione dell'**header**:

```
#include <semaphore.h>
```

In questa libreria sono presenti molteplici funzioni che operano su un nuovo tipo di dato, il **semaforo**

```
sem_t <nome semaforo>
```

I **semafori** presenti in **POSIX1.b** possono essere **named** e **unnamed**: noi utilizzeremo i **semafori unnamed** che possiedono funzioni simili ai **mutex** e per utilizzarli in applicazioni multiprocesso basta allocarli nella **memoria condivisa**.

Descriviamo sinteticamente le principali operazioni su variabili di tipo **sem_t**.

- 1 Per inizializzare un **semaforo unnamed** si usa la funzione:

```
int sem_init(sem_t *sem, int pshared, unsigned int valore )
```

Dove:

sem_t *sem è il puntatore al semaforo da inizializzare;

int pshared nell'attuale implementazione deve essere posto a 0; in implementazioni future potrà essere posto a 1 nel caso che il **semaforo** dovrà essere condiviso tra più processi.

L'effetto di questa istruzione è quello di inizializzare il semaforo **sem** con il valore **valore**: come al solito avrà valore di ritorno 0 in caso di successo e -1 in caso di fallimento: deve sempre essere fatta dal programmatore.

- 2** Per attendere indefinitamente l'**acquisizione** di un semaforo, cioè per effettuare l'operazione che esegue una **P(sem)**, si ha a disposizione la funzione:

```
int sem_wait(sem_t *sem)      // equivale alla operazione di P(sem)
```

Il funzionamento della **sem_wait()** è quello della **P()** di Dijkstra: il semaforo può essere considerato come un intero e la funzione **sem_wait()** esegue un test per verificarne il valore:

- se il semaforo è minore o **uguale a 0** (**semaforo rosso**), la **sem_wait()** si sospende in coda, forzando un cambio di contesto a favore di un **processo pronto**;
- se il semaforo presenta un valore maggiore o **uguale a 1** (**semaforo verde**), la **sem_wait()** decrementa tale valore e ritorna al chiamante, che può quindi procedere nella sua elaborazione.

- 3** Per tentare di **acquisire** un semaforo ma continuare l'evoluzione se questo è rosso, esiste la funzione "non bloccante":

```
int sem_trywait (sem_t *sem)
```

particolarmente utile nelle situazioni dove potrebbero esserci possibili deadlock.

- 4** Per **rilasciare** un semaforo, cioè per eseguire un segue una **V(sem)**, si usa:

```
int sem_post (sem_t *sem)
```

L'operazione di **sem_post()** incrementa il contatore del **semaforo** e se dopo tale operazione il valore risulta ancora minore od **uguale a zero** significa che altri processi hanno effettuato la **wait_sem()** ma hanno trovato il semaforo rosso e sono stati posti in attesa: quindi si sveglia il primo in coda.

- 5** Per controllare il valore di un **semaforo** si usa la funzione:

```
int sem_getvalue (sem_t *sem, int *sem_val);
```

Il valore del semaforo viene ritornato nella variabile ***sem_val**

- 6** Per distruggere un semaforo si usa la funzione:

```
int sem_destroy (sem_t *sem)
```



ESEMPIO

Vediamo un primo esempio dove riscriviamo il codice che risolve il problema del **produttore/consumatore**, in particolare dove un **produttore** e un **consumatore** si alternano per un numero di **VOLTE** nella scrittura/prelievo di un dato da **buffer** comune.

```
semProCon1.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #define VOLTE 5
6 // memoria condivisa
7 static sem_t mio_sem;
8 int buffer = 0; // var. condivisa con il dato
9 int conta = 0; // var. condivisa per nr ripetizioni
```

Il **thread produttore** esegue la **P()** sul **semaforo** ed entra nella sezione critica dove controlla se è il suo turno (**buffer vuoto**) e in tal caso scrive un numero progressivo (**conta + 1**) nel **buffer**; quindi esce dalla sezione critica eseguendo la **V()**:

```
11 void *produc (void * arg){ // variabile di ctr del ciclo
12     int avanti = 1;
13     while(avanti) {
14         sem_wait (&mio_sem); // P() sul semaforo mio_sem
15         if (conta < VOLTE){
16             if (buffer == 0){ // se il dato è stato consumato
17                 buffer = conta + 1; // aggiorna il dato per il lettore
18                 printf("T1: scritto %d \n", buffer);
19             }
20         } else {
21             avanti = 0; // fine ripetizione del ciclo
22             printf("T1 ha finito \n");
23         }
24         sem_post(&mio_sem); // V() sul semaforo mio_sem
25     }
26     pthread_exit (0);
27 }
```

Il **thread consumatore** esegue la **P()** sul semaforo ed entra nella sezione critica dove controlla se è il suo turno (**buffer pieno**) e in tal caso legge (e consuma mettendolo uguale a 0) il dato presente nel **buffer**; quindi esce dalla sezione critica eseguendo la **V()**:

```
30 void *consum (void * arg){ // variabile di ctr del ciclo
31     int avanti = 1;
32     while(avanti){
33         sem_wait (&mio_sem); // P() sul semaforo mio_sem
34         if (conta < VOLTE){
35             if (buffer > 0){
36                 printf("T2: letto %d \n", buffer);
37                 conta = buffer; // incremento le volte
38                 buffer = 0; // "consumo" il dato
39             }
40         } else {
41             avanti = 0; // fine ripetizione VOLTE del ciclo
42             printf("T2 ha finito \n");
43         }
44         sem_post(&mio_sem); // V() sul semaforo mio_sem
45     }
46     pthread_exit (0);
47 }
```

Il `main()` inizializza il semaforo `mio_sem` a verde, crea i due `thread` e si pone in attesa della loro terminazione.

```

50 void main () {
51     pthread_t tid1, tid2;
52     sem_init(&mio_sem, 0, 1);           // all'inizio semaforo verde
53
54     if (pthread_create(&tid1, NULL, produci, NULL) < 0){
55         fprintf (stderr, "errore nella creazione di thread 1\n");
56         exit (1);
57     }
58     if (pthread_create(&tid2, NULL, consumi,NULL) < 0) {
59         fprintf (stderr, "errore nella creazione di thread 2\n");
60         exit (1);
61     }
62     pthread_join (tid1, NULL);
63     pthread_join (tid2, NULL);
64 }
```

Compiliamo e mandiamo in esecuzione il programma ottenendo la seguente schermata:

```

~/ua2/c_semafori
Paolo@PCwin8 ~/ua2/c_semafori
$ gcc semProConi.c -o pcl
Paolo@PCwin8 ~/ua2/c_semafori
$ ./pcl
T1: scritto 1
T2: letto 1
T1: scritto 2
T2: letto 2
T1: scritto 3
T2: letto 3
T1: scritto 4
T2: letto 4
T1: scritto 5
T2: letto 5
T1 ha finito
T2 ha finito
Paolo@PCwin8 ~/ua2/c_semafori
$
```



Prova adesso!

- Semafori binari
- Un produttore – Due consumatori

- 1 Scrivi un programma dove un produttore scrive tante VOLTE un carattere sempre diverso in un buffer e 2 consumatori a turno lo leggono e lo visualizzano facendo in modo che il produttore inserisca un nuovo carattere solo dopo che entrambi i consumatori lo hanno letto.
- 2 Confronta la tua soluzione con quella riportata nel file [sem1Prod2Cons.c](#)



Prova adesso!

- Semafori binari
- Un produttore – N consumatori

- 1 Scrivi un programma dove un produttore scrive tante VOLTE un carattere sempre diverso in un buffer e N consumatori a turno lo leggono e lo visualizzano: il produttore può produrre un nuovo carattere solo dopo che almeno uno dei consumatori ha letto quello precedente.
- 2 Confronta la tua soluzione con quella riportata nel file [sem1ProdNConsChar.c](#)

AREA digitale

 Soluzione del problema del produttore/consumatore con i mutex

■ Semafori per la gestione di vincoli di precedenza

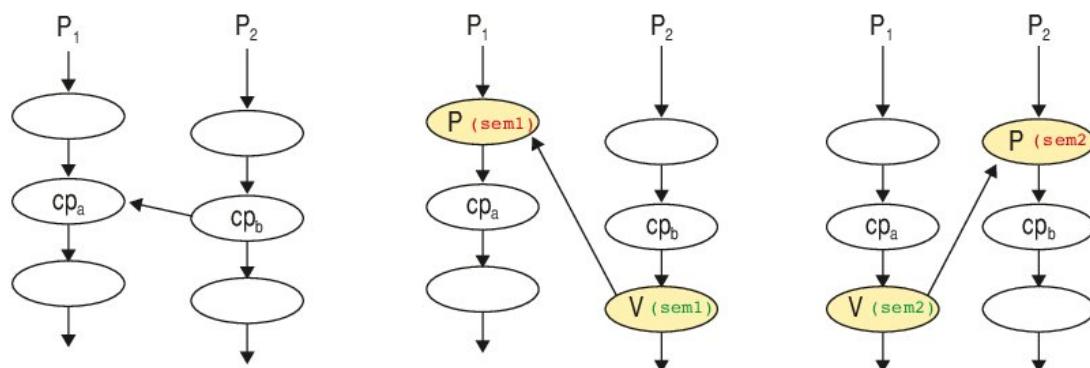
La soluzione del problema **produttore/consumatore** ottenuta con un solo **semaforo** concede a tutti i **processi** di accedere al **buffer** senza vincoli temporali, cioè i **processi consumatori** possono accedere al **buffer** anche se nessun **produttore** è presente nel sistema: per come abbiamo implementato il meccanismo di **sincronizzazione** questi **processi consumatori** consumano risorse per niente in quanto prima di sapere se è il loro turno accedono alla sezione critica per effettuare il controllo sul buffer in modo da accertarsi se questo è pieno/vuoto e si regolano di conseguenza.

Questa strategia non è sicuramente funzionale in quanto occupiamo la **sezione critica** inutilmente al pari di una **attesa attiva**: facendo una analogia nella vita quotidiana, è come se noi andiamo continuamente a guardare nella casella di posta per vedere se è arrivato un documento.

Miglioriamo l'efficienza del sistema introducendo un vincolo temporale, cioè permettendo l'accesso alla regione critica ai **consumatori** solo dopo che un **produttore** ha inserito un dato. Per realizzare questo meccanismo inseriamo nel sistema **due semafori**:

- un primo **semaforo sem1** associato “agli scrittori” che inizializziamo a zero (verde);
- un secondo **semaforo sem2** per i lettori che inizializziamo a 1 (rosso).

In riferimento allo schema seguente:



la sequenza delle operazioni è:

- un produttore P1 prima di eseguire op_a , esegue $P(sem1)$ e blocca altri produttori: accede in mutua esclusione alla SC (semaforo rosso per gli scrittori);
- dopo aver eseguito op_a , P1 esegue $V(sem2)$: risveglia i consumatori “lasciando dormire” gli altri produttori;
- un consumatore P2 prima di eseguire op_b , esegue $P(sem2)$ e blocca altri consumatori: accede in mutua esclusione alla SC;
- dopo aver eseguito op_b , P2 esegue $V(sem2)$: vengono risvegliati i produttori “lasciando dormire” gli altri consumatori.

Riassumendo, per realizzare il meccanismo di **mutua esclusione** utilizzando due semafori abbiamo:

- **producì**: semaforo che è verde quando la memoria non contiene dati utili e quindi il **produttore** può scrivere in essa;
- **leggi**: semaforo che è verde quando la memoria contiene dati utili e quindi il **produttore** può accedere per la lettura.

Lo schema generale di **sincronizzazione** è il seguente:

```
semaforo produci = verde;
semaforo leggi   = rosso;
produttore {
    P(producì);
    ... produci ...
    V( leggi)
}
consumatore {
    P(leggi);
    ... consuma ...
    V(producì);
}
```

Nel nostro esempio facciamo eseguire dieci volte la produzione e il consumo del dato: questo si limita a essere un numero intero che contiene proprio il contatore delle iterazioni effettuate.

Dato che il **produttore** testa il **semaforo** verde, sarà il primo ad accedere alla regione condivisa ed a depositare un dato: finito di scrivere *sveglierà* il **consumatore** modificando il valore del semaforo **leggi** al quale è associata la lista di attesa dei **consumatori** e solo allora il primo **processo** in coda potrà quindi accedere alla **regione critica**, inizierà a leggere il dato e alla fine della lettura metterà a verde il **semaforo** di **producì**, permettendo al **produttore** di inserire un nuovo dato... e così via per dieci iterazioni.

In questo primo esempio creiamo un solo **produttore** e un solo **consumatore**: vedremo nel seguito come completare la casistica aumentando sia il numero dei **produttori** che dei **consumatori**.

La codifica in C richiede l'importazione di quattro librerie, la definizione dei due **semafori** e la definizione delle variabili condivise:

```
semProCon2.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #define MAX 5
6
7 static sem_t produci, leggi;
8 int buffer = 0; // variabile condivisa con il dato
9 int volte = 0; // variabile condivisa per nr ripetizioni
```

Il codice dei **thread produttore** non richiede spiegazioni aggiuntive in quanto è simile alla versione precedente tranne che per le istruzioni 14 e 23 dove vengono settati i due **semafori**:

```
14 sem_wait(&produc); // P() sul semaforo produci dei produttori
15 if (volte < MAX){
16     buffer = volte + 1;
17     printf("T1: scritto %d \n", buffer);
18 }
19 else{
20     avanti = 0; // fine ripetizione del ciclo
21     printf("T1 ha finito \n");
22 }
23 sem_post(&leggi); // V() sul semaforo leggi dei consumatori
```

Analogo discorso per il **thread consumatore** per le istruzioni 31 e 40.

```
31 sem_wait(&leggi); // P() sul semaforo leggi dei consumatori
32 if (volte < MAX){
33     printf("T2: letto %d \n", buffer);
34     volte = buffer; // incrementa le volte
35 }
36 else{
37     avanti = 0; // fine ripetizione del ciclo
38     printf("T2 ha finito \n");
39 }
40 sem_post(&produc); // V() sul semaforo produci dei produttori
```

Riportiamo la prima parte del **main()** dove vengono inizializzati i semafori dei quali quello che indica il dato pronto viene inizializzato staticamente a **rosso**:

```
44
45 int main () {
46     pthread_t tid1, tid2;
47     sem_init(&produc, 0, 1); // verde per la produzione
48     sem_init(&leggi, 0, 0); // rosso per la consumazione
49 }
```

Compiliamo e mandiamo in esecuzione il programma ottenendo lo stesso output del primo esempio, ma in questa realizzazione i **processi** non consumano inutilmente tempo **CPU!**

AREA digitale



Mutex per la gestione di vincoli di precedenza



Prova adesso!

- Semafori sem_t
- Utilizzo di P() e V()
- Sincronizzazione

- 1 Scrivi un programma dove un thread1 esegua un insieme di operazioni (chiamate fase 1) e al suo termine risvegli un thread2 che a sua volta esegue un insieme di operazione (chiamate fase2) regolando la loro sequenzialità mediante un **semaforo**.
- 2 Confronta la tua soluzione con quella presente nel file **semSequenza1.c**
- 3 Scrivi un programma dove tre processi ciclicamente incrementano a turno una variabile rispettando la sequenza P1 -> P2 -> P3
- 4 Confronta la tua soluzione con quella presente nel file **semCiclico1.c**
- 5 Scrivi un programma dove due processi ciclicamente "giocano a ping-pong" rilanciandosi per tante VOLTE la pallina sincronizzandosi con due semafori, sem1 e sem2.
- 6 Confronta la tua soluzione con quella presente nel file **semPingPong1.c**
- 7 Scrivi un programma per sincronizzare tre processi che rispettivamente visualizzano il suono di una campana in modo da ottenere la sequenza infinita di "DIN" "DON" "DAN".
- 8 Confronta la tua soluzione con quella presente nel file **semCampane1.c**

■ N produttori e N consumatori

Il problema dei **produttori/consumatori** nel caso più generale può essere così formulato: alcuni processi **produttori** utilizzando un buffer di scambio devono continuamente inviare messaggi ad altri processi **consumatori** che li elaborano nello stesso ordine in cui li ricevono:

- i **produttori** scrivono i messaggi nel buffer, uno alla volta;
- i **consumatori** leggono i messaggi dal buffer, ciascuno un messaggio diverso.

I due tipi di processi devono essere opportunamente sincronizzati in modo **da evitare** queste situazioni:

- un **consumatore** legge un messaggio senza che un **produttore** ne abbia depositato alcuno;
- un **produttore** sovrascrive un messaggio scritto prima che un **consumatore** sia riuscito a leggerlo;
- un **messaggio** viene letto più di una volta dai **consumatori**.

Scriviamo un codice parametrico in modo da poter personalizzare con delle costanti:

- il numero dei produttori: **NUM_THREAD_P**;
- il numero dei consumatori: **NUM_THREAD_C**;
- il numero di messaggi che devono essere prodotti: **VOLTE**.

Per la sincronizzazione utilizziamo tre **semafori**:

- un **MUTEX** per regolare l'acceso ai dati condivisi;
- un **semaforo vuoto** per regolare i **produttori**;
- un **semaforo pieno** per regolare i **consumatori**.

semifutesPC.c

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5
6 #define VOLTE 33      // elementi da produrre
7 #define NUM_THREAD_P 3 // numero produttori
8 #define NUM_THREAD_C 8 // numero consumatori
9 #define FALSO 0
10 #define VERO 1
11
12 char buffer = 'a'-1;    // scritto dai produttori e letto dai consumatori
13 int scritti = 0;        // modificato dai produttori
14 int letti = 0;          // modificato dai consumatori
15 int fine = FALSO;      // modificato dai produttori quando hanno prodotto VOLTE
16
17 sem_t pieno, vuoto;    // semafori per la sincronizzazione
18 pthread_mutex_t mutex; // semaforo per la sezione critica
19

```

Il codice del **thread produttore** è costituito da:

- A** un ciclo a condizione finale che ripete la produzione degli elementi così strutturato;
 - una istruzione genera casualmente un tempo di attesa che “simula il tempo in cui un **produttore** produce il suo prodotto”;
 - l’istruzione di **P(vuoto)** che regola l’accesso nel caso in cui il buffer sia disponibile per “inserire” la produzione oppure sospende il **thread** in attesa della **wait** (istruzione 26);
 - il controllo del **mutex** che garantisce la mutua esclusione alle risorse condivise;
 - il test per verificare se è necessario produrre nuovi messaggi oppure i **produttori** hanno “finito il loro lavoro”;
 - la produzione del messaggio, che si limita a incrementare di una unità il carattere presente nel **buffer** e ad aggiornare il contatore dei messaggi prodotti;
 - viene infine risvegliata la coda dei **consumatori** con la **V(pieno)** con la istruzione 38 e rilasciato il **mutex** della regione critica con l’istruzione 39;
- B** dalla coda viene risvegliato eventualmente un **thread** produttore in attesa in modo da cauterarsi per quelle situazioni in cui alla fine di tutta la produzione ci fossero altri produttori in attesa di un evento che non accadrebbe mai: quindi prima di terminare la sua vita un produttore deve risvegliare un “collega produttore” affinché possa anch’esso terminare (istruzione 46).

```

20 void *produci(void *threadid){
21     long tid;
22     tid = (long) threadid;
23     printf("\n Ciao a tutti: sono il thread produci #%ld!", tid);
24     do{
25         sleep(rand() % 3);           // prepara la "produzione"
26         sem_wait(&vuoto);         // blocca altri produttori
27         //-----
28         pthread_mutex_lock(&mutex); // entra nella sez. critica
29         if (scritti < VOLTE){
30             buffer++;              // nuovo dato prodotto - condiviso
31             scritti++;              // contatore produzione -condiviso
32             printf("\n produci #%ld %d", tid, scritti);
33             fflush(stdout);
34         }
35         else{
36             fine = VERO;           // scrive su memoria condivisa fine produzione
37         }
38         sem_post(&pieno);          // risveglia i consumatori
39         pthread_mutex_unlock(&mutex); // esce dalla sez. critica
40         //}
41     }while(scritti < VOLTE);      // manca ancora produzione
42
43     // ---parte finale ---
44     printf("\n produci #%ld ha finito ", tid);
45     fflush(stdout);
46     sem_post(&vuoto);           // necessario se C < P per far terminare i produttori
47     return NULL;
48 }

```

Il codice del **consumatore** è molto simile dove viene dapprima testato il **semaforo** per verificare se il buffer è pieno e di seguito testato il **mutex** che regola la sezione critica.

Per meglio controllare l'evoluzione del programma introduciamo alcune istruzioni di output che visualizzano il contenuto delle variabili.

```

49 void *consuma(void *threadid){
50     long tid;
51     tid = (long) threadid;
52     printf("\n Ciao a tutti: sono il thread consuma #%ld!", tid);
53     char miobuffer = '\x0';
54     do{
55         sleep(rand() % 3);           // si riposa ...
56         sem_wait(&pieno);        // blocca altri consumatori
57     //...
58         pthread_mutex_lock(&mutex);    // entra nella sez. critica
59         if (letti < scritti){
60             miobuffer = buffer;      // legge variabile condivisa
61             letti++;                // modifica variabile condivisa
62             printf("\n consuma #%ld ha letto %c: %d di %d", tid, miobuffer, letti, scritti);
63             fflush(stdout);
64         }
65         pthread_mutex_unlock(&mutex); // esce dalla sez. critica
66         sem_post(&vuoto);          // risveglia i produttori
67     //...
68     } while ((fine == FALSO) );    // non prodotti tutti o non consumati tutti
69     // ---parte finale ---
70     printf("\n consuma #%ld ha finito ", tid);
71     fflush(stdout);
72     sem_post(&pieno);           // necessario se P < C
73     return NULL;               // sem_post(&vuoto); // necessario se C < P
74 }
```

Dopo aver definito due array per memorizzare i tid dei **thread** produttori e **consumatori**, il **main()** inizializza i **semafori** ponendo a verde quelle dei produttori ed a rosso quello dei **consumatori**. Quindi avvia tutti i **thread** e si mette in attesa della loro terminazione.

```

76 int main(void){
77     pthread_t thread_P[NUM_THREAD_P]; // vettore dei produttori
78     pthread_t thread_C[NUM_THREAD_C]; // vettore dei consumatori
79     long tc, tp ;
80     printf("\nInizio elaborazione...");
81     sem_init(&pieno, 0, 0);           // inizia a falso (rosso)
82     sem_init(&vuoto, 0, 1);          // inizia a vero (verde)
83     for (tp = 0; tp < NUM_THREAD_P; tp++) {
84         if (pthread_create(thread_P+tp, NULL, produci, (void *)tp+1) < 0){
85             fprintf (stderr, "errore nella creazione del thread P%d\n", tc);
86             exit (1);
87         }
88     }
89     for (tc = 0; tc < NUM_THREAD_C; tc++){
90         if (pthread_create(thread_C+tc, NULL, consuma, (void *)tc+1) < 0){
91             fprintf (stderr, "errore nella creazione del thread C%d\n", tc);
92             exit (1);
93         }
94     }
95     for (tp = 0; tp < NUM_THREAD_P; tp++) // attesa terminazione produttori
96         pthread_join(thread_P[tp], NULL);
97     for (tc = 0; tc < NUM_THREAD_C; tc++) // attesa terminazione onsumatori
98         pthread_join(thread_C[tc], NULL); // bisognerebbe distruggere i 3 semafori
99     printf("\nFine elaborazione !");    // prima di terminare l'elaborazione
100 }
```

Una esecuzione con 2 produttori, 4 consumatori e 7 messaggi da produrre dà il seguente output: ►

```

Inizia elaborazione ...
Ciao a tutti: sono il thread produci #2!
Ciao a tutti: sono il thread produci #1!
Ciao a tutti: sono il thread consuma #1!
Ciao a tutti: sono il thread consuma #2!
Ciao a tutti: sono il thread consuma #3!
Ciao a tutti: sono il thread consuma #4!
    produci #2 1
consuma #4 ha letto a: 1 di 1
    produci #1 2
consuma #2 ha letto b: 2 di 2
    produci #1 3
consuma #1 ha letto c: 3 di 3
    produci #2 4
consuma #3 ha letto d: 4 di 4
    produci #2 5
consuma #2 ha letto e: 5 di 5
    produci #1 6
consuma #4 ha letto f: 6 di 6
    produci #1 7
    produci #1 ha finito
consuma #3 ha letto g: 7 di 7
    produci #2 ha finito
    consuma #1 ha finito
    consuma #3 ha finito
    consuma #4 ha finito
    consuma #2 ha finito
Fine elaborazione !

```



Prova adesso!

- N produttori – N consumatori
- Buffer circolare
- Lettori e scrittori

- 1 Scrivi un programma dove NUM_THREAD_P produttori scrivono tante VOLTE un carattere sempre diverso in un buffer circolare di dimensione DIMBUFFER e NUM_THREAD_C consumatori a turno lo leggono e lo visualizzano.
Confronta la tua soluzione con quella riportata nel file [semMutexCirco.c](#). È possibile utilizzare un unico semaforo? Discuti la soluzione proposta nel file [semMutexCirco1.c](#)
- 2 Quattro amici fanno una scommessa, il premio per il vincitore è di poter fare bere uno dei rimanenti amici. Chi vince la scommessa sceglie casualmente a chi tocca bere il cocktail tutto d'un fiato mentre gli altri stanno a guardare. Alla fine, viene proposta un'ulteriore scommessa e il gioco va avanti all'infinito. Rappresentare il problema utilizzando i semafori.
- 3 Realizza un programma che sincronizzi la situazione lettori/scrittori, dove più lettori possono contemporaneamente accedere alla risorsa condivisa mentre gli scrittori devono alternarsi, nei seguenti casi:
 - ▶ privilegiando i lettori;
 - ▶ privilegiando gli scrittori;
 - ▶ alternando il più possibile lettori e scrittori.

L'uso di **semafori** a livello di programma è comunque ostico e rischioso:

- ▶ il posizionamento improprio delle P può causare situazioni di blocco infinito (**deadlock**) o anche esecuzioni erronee di difficile verifica (**race condition**);
- ▶ è indesiderabile lasciare all'utente il pieno controllo di strutture così delicate;
- ▶ linguaggi evoluti di alto livello offrono strutture esplicite di controllo delle regioni critiche come i **monitor**.

AREA digitale

Esercizi per il recupero / Esercizi per l'approfondimento

[hoepliscuola.it](#)

ESERCITAZIONI DI LABORATORIO 6

VARIABILI CONDIZIONE

■ Pthread condition variable

Il linguaggio C ha nella libreria `<pthread.h>` oltre ai **semafori** descritti nelle lezioni precedenti un ulteriore strumento adatto per “sospendere” i **thread** in attesa della verifica di situazioni particolari: le **variabili condizione** (**condition variable**).

Le **variabili condizione** sono un meccanismo che viene utilizzato per sospendere l'esecuzione di un **thread** in attesa che si verifichi un certo evento.

Le **variabili condizione** sono molto diverse dai **semafori di sincronizzazione**, anche se semanticamente fanno la stessa cosa: a ogni **condition** viene associata una **coda** per la sospensione dei **thread** ma la **variabile condizione** non ha uno **stato** (libero o occupato), rappresenta solo una **coda** nella quale i **thread** possono essere sospesi e, quindi, le operazioni fondamentali sono la **sospensione e la riattivazione** che vengono effettuate tramite due primitive (**wait** e **signal**).

Le primitive delle **condition** si preoccupano di rilasciare e acquisire la **mutua esclusione** prima di bloccarsi e dopo essere state sbloccate, ma una **variabile condizione** non fornisce la **mutua esclusione**: ha sempre bisogno di un **mutex** per poter **sincronizzare** l'accesso ai dati mentre i **semafori** non necessitano della presenza di altri meccanismi.

Lo schema di funzionamento della **sincronizzazione** è il seguente: abbiamo detto che una **variabile condition** è **sempre** associata a un **mutex** quindi la prima operazione che esegue il **thread** è quella di ottenere il **mutex** e di testare il predicato:

- se il predicato è verificato allora il **thread** esegue le sue operazioni e rilascia il **mutex**;
- se il predicato non è verificato, in modo atomico, il **mutex** viene rilasciato (implicitamente) e il **thread** si blocca sulla **variabile condition**.

Successivamente un **thread** bloccato riacquisisce il **mutex** nel momento in cui viene svegliato da un altro **thread**.

Linux garantisce inoltre che i **thread** bloccati su una **condition** vengano sbloccati quando questa cambia di stato.

■ Definizione di una variabile condizione

Attraverso le **variabili condizione** è però possibile implementare condizioni più complesse che i **thread** devono soddisfare per essere eseguiti: oltre che a essere un nuovo tipo di variabile (**pthread_cond_t**) hanno anche attributi di un nuovo tipo (**pthread_condattr_t**).

È possibile effettuare la creazione di una **variabile condizione** e la contestuale inizializzazione con quella che prende il nome di **inizializzazione statica**.

Inizializzazione statica di una variabile condizione

Una **variabile condizione** è creata e inizializzata mediante la seguente istruzione simile a quella già utilizzata per i **mutex**:

```
pthread_cond_t miavc = PTHREAD_COND_INITIALIZER;
```

dove

- **miavc** è il nome di una **variabile condizione**;
- **PTHREAD_COND_INITIALIZER** è una macro di inizializzazione definita nella libreria `<pthread.h>` che setta gli attributi di **miavc** con i valori di default.

Inizializzazione dinamica di una variabile condizione

È anche possibile inizializzare una **variabile condizione** in fase run-time mediante la seguente istruzione:

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr)
```

dove:

```
pthread_cond_t *cond
```

puntatore a un'istanza della variabile **condition** (condizione di sincronizzazione):

```
pthread_condattr_t *cond_attr
```

puntatore a una struttura che contiene gli attributi della condizione: può anche assumere il valore **NULL** e in questo caso viene inizializzato con i valori di default.

Distruzione di una variabile condizione

Al termine della elaborazione è necessario deallocare lo spazio in memoria, così come abbiamo visto per i **semafori**: viene richiamata la seguente funzione:

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

Affinché la funzione dia un esito positivo e ritorni il valore 0 non devono essere presenti **thread** in attesa sulla variabile **cond** altrimenti viene ritornato un valore diverso da 0.

■ Primitive fondamentali: wait e signal

Le primitive fondamentali sulle **variabile condizione** sono due: **wait()** per la **sospensione** e **signal()** per la **riattivazione**.

Attesa su una variabile condizione: wait

Per mettersi in attesa di una **certa condizione** all'interno di un blocco di dati condivisi e protetti da un **mutex** un **thread** utilizza la seguente **system call**:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

dove i parametri sono:

```
pthread_cond_t *cond
```

è il puntatore all'istanza di **condition** che rappresenta la condizione di sincronizzazione:

```
pthread_mutex_t *mutex
```

è l'indirizzo di un **semaforo** di mutua esclusione necessario alla corretta consistenza dei dati regolando l'accesso alla regione critica agli altri **thread** dopo aver messo nella coda di attesa il **thread** corrente.

La funzione ha sempre 0 come valore di ritorno.

Alla chiamata di **wait()** il **mutex** viene sbloccato mentre il **thread** chiamante **si blocca** **sem pre** sulla **variabile condition**, in attesa di essere risvegliato da un segnale (una successiva **signal()**).

La presenza del **mutex** fra i parametri garantisce che, al momento del bloccaggio del **thread** che esegue la **wait()**, esso venga posto automaticamente a verde, eliminando a monte possibili errori di programmazione che potrebbero condurre a condizioni di **deadlock**.

Quando il **thread** sospeso riceve un segnale di risveglio, entra in competizione per accedere a bloccare il **mutex** prima di riprendere a eseguire le successive istruzioni della **sezione critica**.

Per effettuare una **wait()** e quindi attendere su una **variabile condition** un **thread** deve aver precedentemente effettuato una **lock()** sul **mutex** e non sulla **variabile condition**, che **non è** un "oggetto booleano":

```
if (miaCV)
    then ...
if (risorseLibere == 0)
    then wait(&miaCV, &mutex)           // è errata e senza significato
                                            // è formalmente corretta
```

Risveglio di una variabile condizione: signal

La **signal()** non si preoccupa di liberare la **mutua esclusione**: fra i suoi parametri, infatti, non c'è il **mutex** ma solo la **variabile condition**, quindi il **mutex** deve essere rilasciato esplicitamente dal programmatore con una successiva istruzione altrimenti si potrebbe produrre una condizione di **deadlock**.

Una **variabile condizione** può essere svegliata in due modi: **standard** oppure **broadcast**.

a) Standard: sblocca un solo thread bloccato

La modalità standard, che è quella più utilizzata, sblocca il **thread** a priorità più alta che è in attesa da più tempo.

```
int pthread_cond_signal (pthread_cond_t *cond)
```

Se esistono **thread** sospesi nella coda associata a **cond** ne viene risvegliato il primo ma se non vi sono **thread** in attesa sulla condizione la **signal()** non produce alcun effetto e viene persa. Come unico parametro ha

```
pthread_cond_t *cond
```

che è il puntatore all'istanza di **condition** che rappresenta la condizione di sincronizzazione. La funzione ha sempre 0 come valore di ritorno.

b) Broadcast: sblocca tutti i thread bloccati

Con la chiamata alla seguente funzione:

```
int pthread_cond_broadcast(pthread_cond_t *cond)
```

vengono sbloccati tutti i **thread** sospesi sulla variabile **condition** rispettando l'ordine di arrivo **FIFO** in caso siano presenti **thread** con medesima priorità.

Come unico parametro ha

```
pthread_cond_t *cond
```

che è il puntatore all'istanza di **condition** che rappresenta la condizione di sincronizzazione. La funzione ha sempre 0 come valore di ritorno.

Ne è sconsigliato l'utilizzo dato che il **thread** svegliato deve rivalutare la condizione in quanto l'altro **thread** potrebbe non aver testato la condizione e nel frattempo questa potrebbe essere stata cambiata.

Osservazioni sul risveglio

Quando un **thread A** effettua una **signal()** e risveglia un **thread B**, questo riprenderà la sua attività all'interno della **sezione critica**, poiché è proprio all'interno della **SC** che si è sospeso: ma il **thread A** che lo ha risvegliato non ha ancora lasciato la **SC** quindi, per un "breve intervallo di tempo", non viene rispettato il **principio della mutua esclusione** all'interno della **SC**, dato che in essa sono presenti contemporaneamente due **thread**.

L'**intervallo di tempo** di convivenza dipende dall'approccio operativo che viene adottato dal programmatore per il comportamento del **thread A** dopo che ha risvegliato il **thread B**:

- 1** se viene seguito l'approccio di **Brinch Hansen** il **thread A** viene fatto uscire immediatamente dalla **SC** sbloccando il **mutex**;
- 2** se viene seguito l'approccio di **Hoare** il **thread A** viene posto in attesa finché il **thread B** svegliato non usa più la **risorsa**.

Generalmente si usa il primo approccio in modo da limitare il più possibile la “convivenza”.

■ Sincronizzazione con condition variable

Per realizzare la sincronizzazione di **thread** con **variabili condizione** per prima cosa le definiamo e inizializziamo:

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t libero=PTHREAD_COND_INITIALIZER;
```

Lo schema della procedura/funzione di un **thread** che si metta in attesa sulla **variabile condition** è il seguente:

```
pthread_mutex_lock(&mutex);           // accede alla sc
...
pthread_cond_wait(& condVar, &mutex); // accede alla sc
...
pthread_mutex_unlock(&mutex);        // esce dalla sc
```

Lo schema della procedura/funzione di un **thread** che libera la situazione di attesa sulla **variabile condition** è il seguente:

```
pthread_mutex_lock(&mutex);
...
pthread_cond_signal(&condVar, &mutex);
...
pthread_mutex_unlock(&mutex);
```

Vediamo un esempio completo di codice dove un primo **thread** si sospende su un semaforo in attesa che un secondo **thread** lo “risvegli” al termine della sua elaborazione.

ESEMPIO Accesso alla SC regolato da condition variable

Definiamo due semafori, un **mutex** e una **condition variable**:

varCondition1.c
<pre>1 #include <stdio.h> 2 #include <stdlib.h> 3 #include <pthread.h> 4 5 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // mutex 6 pthread_cond_t condVar = PTHREAD_COND_INITIALIZER; // condition variable</pre>

Il codice dei due **thread** rispecchia lo schema prima descritto:

```

8  void produci(void *parametro) {
9      printf("Avvio della esecuzione del %s.\n", (char *)parametro);
10     sleep(2);           // pausa di comodo
11     printf("Thread 1 tenta di entrare nella sezione critica.\n");
12     pthread_mutex_lock(&mutex);
13     printf("Thread 1 nella sezione critica - ha bloccato il mutex.\n");
14     printf("Thread 1 produce ....\n");
15     sleep(1);           // pausa di comodo
16     printf("Thread 1 si sospende sulla condition variable.\n");
17     pthread_cond_wait(&condVar, &mutex);
18     printf("Thread 1 riprende l'esecuzione.\n");
19     printf("Thread 1 ora esce dalla sezione critica - rilascia mutex\n");
20     pthread_mutex_unlock(&mutex);
21     printf("Thread 1 puo' terminare.\n");
22 }
```

All'avvio il primo **thread** aspetta due secondi prima di entrare nella **sezione critica**: quindi effettua il test sulla **condition variable** **condVar** e, trovandola a valore falso, si sospende in attesa che questa venga modificata da qualche altro **thread**.

```

24 void consuma(void *parametro) {
25     printf(" Avvio della esecuzione del %s.\n", (char *)parametro);
26     sleep(5);           // pausa di comodo
27     printf(" Thread 2 tenta di entrare nella sezione critica.\n");
28     pthread_mutex_lock(&mutex);
29     printf(" Thread 2 nella sezione critica - ha bloccato mutex.\n");
30     printf(" Thread 2 consuma ....\n");
31     sleep(1);           // pausa di comodo
32     printf(" Thread 2 sblocca chi e' in attesa sulla condition variable.\n");
33     pthread_cond_signal(&condVar);
34     printf(" Thread 2 ora esce dalla sezione critica - rilascia mutex.\n");
35     pthread_mutex_unlock(&mutex);
36     printf(" Thread 2 puo' terminare.\n");
37 }
```

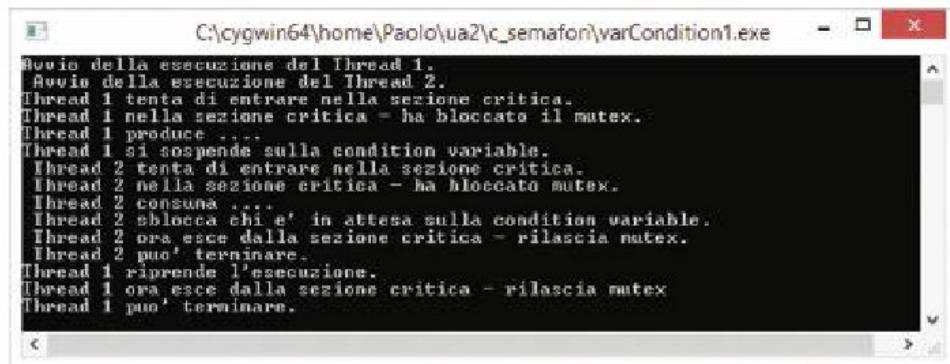
Il secondo **thread** aspetta cinque secondi prima di entrare nella **sezione critica** e dentro essa modifica il valore della **condition variable** **condVar** in modo da risvegliare il primo **thread** che era sospeso su di essa; quindi termina l'elaborazione.

Il programma **main** si limita a creare i due **thread**, a mandarli in esecuzione e ad attendere la terminazione:

```

40 void main(){
41     pthread_t thread1, thread2;
42     char *nome1 = "Thread 1";
43     char *nome2 = "Thread 2";
44     if(pthread_create(&thread1, NULL, &produci, nome1) != 0) {
45         perror("Errore nella creazione del primo thread.\n");
46         exit(1);
47     }
48     if(pthread_create(&thread2, NULL, &consuma, (void*)nome2) != 0) {
49         perror("Errore nella creazione del secondo thread.\n");
50         exit(1);
51     }
52     pthread_join(thread1, NULL);
53     pthread_join(thread2, NULL);
54 }
```

Un'esecuzione è la seguente:



```
C:\cygwin64\home\Paolo\ua2\c_semafon\varCondition1.exe
Avvio della esecuzione del Thread 1.
Avvio della esecuzione del Thread 2.
Thread 1 tenta di entrare nella sezione critica.
Thread 1 nella sezione critica - ha bloccato il mutex.
Thread 1 produce ...
Thread 1 si sospende sulla condition variabile.
Thread 2 tenta di entrare nella sezione critica.
Thread 2 nella sezione critica - ha bloccato mutex.
Thread 2 consuma ...
Thread 2 sblocca chi e' in attesa sulla condition variabile.
Thread 2 ora esce dalla sezione critica - rilascia mutex.
Thread 2 puo' terminare.
Thread 1 riprende l'esecuzione.
Thread 1 ora esce dalla sezione critica - rilascia mutex.
Thread 1 puo' terminare.
```



Prova adesso!

- Variabile condition
- Sincronizzazione thread

Completa il codice presente nel file **varCondition2.c** dove un gruppo di **thread** eseguono alcune operazioni (da noi simulate con un tempo di attesa random) e al loro termine si pongono tutti in attesa degli altri, cioè aspettano gli altri **thread** su una "barriera" prima di poter proseguire l'elaborazione.

Quando tutti i **thread** sono pronti per proseguire, viene inviato un segnale che ne permette la ripresa della elaborazione.

ESERCITAZIONI DI LABORATORIO 7

I MONITOR CON LE VARIABILI CONDITION IN C

■ Monitor

L'uso di **semafori** a livello di programma è ostico e rischioso dato che il posizionamento improprio delle **P()** può causare situazioni di blocco infinito (**deadlock**) oppure esecuzioni erronee di difficile verifica (**race condition**): è preferibile utilizzare i **monitor**.

Nei **monitor** il programmatore definisce la **regione critica** mentre il compilatore inserisce il codice necessario al controllo degli accessi.

Il **linguaggio C** non mette a disposizione i **monitor** ma il programmatore, sapendo che questo è composto da “*un aggregato di dati*” condivisi tra processi, *dalle procedure* che operano su di essi e dalle *primitive per la sincronizzazione* tra processi che lo usano, può realizzarlo “da sé” definendolo con dati strutturati o, meglio ancora, con oggetti in **linguaggio C++**.

Quindi la “creazione e gestione del monitor” avviene manualmente a opera del programmatore che deve però tenere ben presente durante la sua realizzazione che:

- ▶ i dati “interni al monitor” sono *potenzialmente accessibili* direttamente da tutti i **processi**;
- ▶ la **mutua esclusione** delle funzioni/procedure entry deve essere garantita esplicitamente dal programmatore mediante lock/unlock su un mutex associato al “monitor”;
- ▶ è necessario che il programmatore effettui l’accesso al **monitor** esclusivamente attraverso le **funzioni/procedure entry**.

Un problema che rimane in comune sia con i **monitor** che con la gestione con **semafori** è il fatto che per la loro realizzazione è necessario avere **memoria condivisa** e quindi questi costrutti non sono applicabili ai *sistemi distribuiti* come le **reti di calcolatori** dato che non hanno memoria fisica condivisa.

■ Monitor con variabili condizione

Risolviamo come esempio il classico problema **produttore/consumatore** nel primo caso, cioè quello in cui è presente un solo **produttore** e un solo **consumatore** dove per sospendere e riprendere i **processi** ci serviamo delle **condition variable**.

Dopo l'inclusione delle librerie indichiamo il numero di dati da produrre come costante manifesta **MAX**, cioè per quante volte il produttore inserirà un dato in una variabile condivisa: per indicare il termine della produzione si inserirà il valore **FINE** (costante posta a -1):

```
moniterVC1.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 // costanti comuni
5 #define MAX 4           // numero di dati da produrre
6 #define FINE (-1)        // valore del dato di fine produzione (tappo)
```

Definiamo un record contenente la regione di memoria condivisa (**int buffer**) e i semafori per accedere a essa, che sono un **mutex** e due semafori **condition**, col codice seguente:

```
8  typedef struct{
9      int buffer;           // area di memoria condivisa
10     pthread_mutex_t mutex; // mutex per la mutua esclusione
11     pthread_cond_t pieno; // condition var per accedere a consumare
12     pthread_cond_t vuoto; // condition var per accedere a produrre
13 } monitor;             // dichiarazione della variabile comune
14 monitor bufferComune;
```

Definiamo ora le primitive che possono operare su questa struttura dati: innanzi tutto scriviamo la funzione che provvede alla loro inizializzazione che sarà chiamata dal **main()** prima della creazione dei **thread** (istruzione 79).

```
16 void inizializza (monitor *mpc){ // entry per inizializzare il buffer
17     pthread_mutex_init(&mpc->mutex, NULL);
18     pthread_cond_init (&mpc->pieno, NULL);
19     pthread_cond_init (&mpc->vuoto, NULL);
20     mpc->buffer=0;
21 }
```

La **entry inserisci()** inizia e termina con la gestione della **mutua esclusione** regolata dal **mutex**: quindi controlla se il buffer è pieno con l'istruzione 26 e in tale caso si sospende: se il buffer è vuoto inserisce il proprio dato e avvisa il consumatore modificando il valore del **mpc-> pieno** con istruzione 31; quindi rilascia la **sezione critica** ponendo il **mutex** a verde:

```
23 void inserisci (monitor *mpc, int data){ // entry per l'insersione
24     pthread_mutex_lock (&mpc->mutex); // mutex di mutua esclusione a rosso
25     //-
26     if (mpc->buffer != 0){ // se il buffer è pieno
27         pthread_cond_wait(&mpc->vuoto, &mpc->mutex); // si pone in attesa e libera mutex
28     }
29     if (mpc->buffer == 0) // se il buffer è vuoto
30         mpc->buffer = data; // scrivi il dato
31     pthread_cond_signal(&mpc->pieno); // risveglia eventuali consumatori
32     //-
33     pthread_mutex_unlock(&mpc->mutex); // mutex di mutua esclusione a verde
34 }
```

Quando il **thread** si risveglia è sempre opportuno effettuare nuovamente il controllo del buffer, come quello da noi fatto con l'istruzione 29: nel caso di N produttori e N consumatori, se questi venissero risvegliati tutti contemporaneamente, qualche altro **thread** potrebbe aver prodotto prima del **thread** corrente, dato che tutti vengono sospesi all'interno di una sezione critica.

Anche la **entry inserisci()** inizia e termina con la gestione della **mutua esclusione** regolata dal **mutex**: quindi si controlla se il buffer è vuoto e nel caso in cui nessun dato è stato prodotto il **thread** si pone in attesa sul corrispondente semaforo pieno (istruzione 41) mentre se il dato è disponibile questo viene prelevato, si svuota il buffer e segnala al **produttore** che può riprendere l'attività (istruzione 47):

```

36 int estrai (monitor *mpc){           // entry per l'estrazione
37     int data;
38     pthread_mutex_lock(&mpc->mutex);    // mutex di mutua esclusione a rosso
39     //
40     if (mpc->buffer == 0){            // se il buffer è vuoto
41         pthread_cond_wait(&mpc->pieno, &mpc->mutex); // si pone in attesa e libera mutex
42     }
43     if (mpc->buffer != 0){           // se il buffer è pieno
44         data = mpc->buffer;          // leggi il dato
45         mpc->buffer = 0;             // svuota il buffer
46     }
47     pthread_cond_signal(&mpc->vuoto); // risveglia eventuali produttori
48     //
49     pthread_mutex_unlock(&mpc->mutex); // mutex di mutua esclusione a verde
50     return data;
51 }
```

Quello che abbiamo realizzato sino a ora è proprio il **monitor**, con la sua area dati e le primitive (**entry**) che permettono ai **thread** di potervi accedere.

Queste due **entry** vengono chiamate rispettivamente dai processi **produttore** e **consumatore** che, come si può vedere dai codici seguenti, non si preoccupano della **sincronizzazione** ma si limitano a utilizzare le funzioni messe a disposizione dal **monitor**.

```

54 void *produttore (void *data){           // nel nostro caso nessun parametro
55     int n;
56     for (n = 1; n <= MAX; n++){
57         printf (" produttore: dato %d ..prodotto\n", n);
58         inserisci (&bufferComune, n); // richiama la entry del monitor
59         printf (" produttore: dato %d ..inserito\n", n);
60     }
61     inserisci (&bufferComune, FINE);
62     return NULL;
63 }
64
65 void *consumatore (void *data){           // nel nostro caso nessun parametro
66     int dato;
67     do {
68         dato = estrai (&bufferComune); // richiama la entry del monitor
69         if (dato != FINE)
70             printf("consumatore: dato %d ...consumato\n", dato);
71     } while (dato != FINE);
72     return NULL;
73 }
```

Anche il main non si cura della sincronizzazione, ma si limita a definire i due **thread**, a mandarli in esecuzione e ad attendere la loro terminazione.

```

75 int main (){
76     pthread_t pid1, pid2;
77     void *retval;           // valore di ritorno
78
79     inizializza(&bufferComune);
80     pthread_create(&pid1, NULL, produttore, 0); // creazione di due threads
81     pthread_create(&pid2, NULL, consumatore, 0);
82
83     pthread_join(pid1, &retval); // attesa terminazione threads
84     pthread_join(pid2, &retval);
85     return 0;
86 }
```

Se ora mandiamo in esecuzione il programma diverse volte in ambiente windows troviamo probabilmente sempre esecuzioni diverse ma, dato che **wait** e **signal** sono invocate in mutua esclusione, non si può verificare la situazione di **race condition** e quindi queste situazioni sono esclusivamente legate alla schedulazione dei processi da parte del sistema operativo:

```
produttore: dato 1 ..prodotto
produttore: dato 1 ..inserito
produttore: dato 2 ..prodotto
consumatore: dato 1 ...consumato
produttore: dato 2 ...consumato
produttore: dato 2 ..inserito
produttore: dato 3 ..prodotto
produttore: dato 3 ..inserito
produttore: dato 4 ..prodotto
produttore: dato 4 ..inserito
consumatore: dato 3 ...consumato
consumatore: dato 4 ...consumato
```

```
produttore: dato 1 ..prodotto
produttore: dato 1 ..inserito
produttore: dato 2 ..prodotto
consumatore: dato 1 ...consumato
produttore: dato 2 ...consumato
produttore: dato 3 ..prodotto
consumatore: dato 2 ...consumato
produttore: dato 3 ..inserito
produttore: dato 3 ..prodotto
produttore: dato 4 ..inserito
produttore: dato 4 ..prodotto
consumatore: dato 3 ...consumato
consumatore: dato 4 ...consumato
```

```
produttore: dato 1 ..prodotto
produttore: dato 1 ..inserito
produttore: dato 2 ..prodotto
consumatore: dato 1 ...consumato
produttore: dato 2 ...consumato
produttore: dato 3 ..prodotto
produttore: dato 3 ..inserito
produttore: dato 4 ..prodotto
produttore: dato 4 ..inserito
consumatore: dato 3 ...consumato
consumatore: dato 4 ...consumato
```

Basta introdurre una piccola pausa nel produttore tra un prodotto e il successivo per garantire la corretta schedulazione dei **thread**, ottenendo il seguente risultato:

```
produttore: dato 1 ..prodotto
produttore: dato 1 ..inserito
consumatore: dato 1 ...consumato
produttore: dato 2 ..prodotto
produttore: dato 2 ..inserito
consumatore: dato 2 ...consumato
produttore: dato 3 ..prodotto
produttore: dato 3 ..inserito
consumatore: dato 3 ...consumato
produttore: dato 4 ..prodotto
produttore: dato 4 ..inserito
consumatore: dato 4 ...consumato
```

Il problema non si pone su macchine native **unix/linux** oppure **iOS**, come già descritto nelle precedenti lezioni.



Prova adesso!

- Monitor con variabile condition
- Problema produttori/consumatori
- Buffer circolare

Scrivi un programma C che crea due **thread**, chiamati produttore e consumatore. La risorsa condivisa è un buffer circolare di dimensione **BUFFER_SIZE** (per esempio 20) il cui stato è identificato dalla seguenti variabili:

- 1 numero di elementi contenuti: **contaDati**;
- 2 puntatore alla prima posizione libera: **posScrittura**;
- 3 puntatore al primo elemento occupato: **posLettura**;
- 4 tempo per la preparazione di un dato: **DELEYP**
- 5 tempo per consumare un dato: **DELEYC**

Il produttore inserisce **TANTI** numeri interi in maniera sequenziale e il consumatore li estrae sequenzialmente, in ordine di inserimento, per stamparli.

Il programma dovrà prevedere:

- ▶ un meccanismo di accesso controllato alla risorsa buffer da parte dei due **thread** (**mutex** per il controllo della mutua esclusione nell' accesso al buffer);
- ▶ una sincronizzazione tra il **thread** produttore e consumatore in caso di:
 - A** buffer pieno: definizione di una **var condition** per la sospensione del produttore se il buffer è pieno (**nonPieno**);
 - B** buffer vuoto: definizione di una **var condition** per la sospensione del consumatore se il buffer è vuoto (**nonVuoto**).

Per esempio per il **monitor** puoi utilizzare la seguente struttura:

```

8  typedef struct{
9    int buffer[BUFFER_SIZE];
10   pthread_mutex_t lock;
11   int posLettura;                                // indice primo dato da leggere
12   int posScrittura;                            // indice primo dato da scrivere
13   int contaDati;
14   pthread_cond_t nonVuoto;
15   pthread_cond_t nonPieno;
16 }prodcons;
17 prodcons bufferComune;
```

Confronta la tua soluzione con quella presente nel file **monitorVC2.c**

Modifica l'esempio precedente (1 solo produttore e 1 solo consumatore) per generalizzare il problema a N produttori e N consumatori che utilizzano sempre un buffer circolare con dimensione **BUFFER_SIZE**: per esempio:

```

#define BUFFER_SIZE 20      // dimensione buffer circolare .
#define NUM_THREAD_P 5     // numero produttori
#define NUM_THREAD_C 10    // numero consumatori
```

Effettua le necessarie modifiche al monitor e ai processi di prova nonché al **main()** che deve generare rispettivamente:

- ▶ 5 processi produttori e 10 processi consumatori;
- ▶ 10 processi produttori e 5 processi consumatori;
- ▶ 10 processi produttori e 10 processi consumatori.

Discuti i risultati che hai ottenuto.

ESERCITAZIONI DI LABORATORIO 8

I MONITOR CON I SEMAFORI IN C

■ Monitor con semafori

Per realizzare il **monitor** al posto delle **variabili condition** è anche possibile utilizzare i **semafori** visti nelle lezioni di laboratorio nr. 3 e nr. 4.

Risolviamo di seguito il problema dove N produttori (definiti in una costante **NUM_THREAD_P**) e N consumatori (definiti in una costante **NUM_THREAD_C**) condividono un buffer nel quale ogni produttore inserisce un numero ATESTA di dati, indicato nelle costanti manifeste.

La prima parte dichiarativa è la seguente:

monitorSem2.c

```

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <pthread.h>
4. #include <semaphore.h>
5. #define ATESTA 10          // numero di dati da produrre per processo
6. #define FINE (-1)        // valore del dato di fine produzione (tappo)
7. #define NUM_THREAD_P 2   // numero produttori
8. #define NUM_THREAD_C 4   // numero consumatori
9. #define FALSO 0
10. #define VERO 1
11. #define DELAY 2

```

Scriviamo ora il codice del **monitor**, dove nella struttura dati inseriamo i **semafori** e le **variabili condivise**: indichiamo “di seguito” i tre prototipi delle **procedure entry** che saranno utilizzate dai **thread** per manipolare i tali dati condivisi.

```

13 //----- inizio monitor -----
14 typedef struct{
15     int buffer;           // area di memoria condivisa
16     int scritti;          // modificato dai produttori
17     int letti;            // modificato dai consumatori
18     int globale;          // tot. da produrre = ATESTA*NUM_THREAD_P
19     pthread_mutex_t mutex; // mutex per la mutua esclusione
20     sem_t pieno;          // condition var per accedere a consumare
21     sem_t vuoto;          // condition var per accedere a produrre
22 } monitor;
23 monitor bufferComune;    // dichiarazione della variabile comune
24
25 void inizializza(monitor *miopc); // entry per inizializzare il buffer
26 void metti(monitor *miopc, int data); // entry per l'inserimento
27 int estrai(monitor *miopc); // entry per l'estrazione
28 //----- fine monitor -----

```

Il problema con un singolo produttore e singolo consumatore è riportato tra gli esempi nel file **monitorSem1.c**

La prima procedura effettua l'inizializzazione dei semafori e delle variabili condivise.

```

29
30 void inizializza(monitor *miopc){           // entry per inizializzare il buffer
31     miopc->buffer = 0;
32     miopc->scritti = 0;                      // aggiornato dai produttori
33     miopc->letti = 0;                        // aggiornato dai consumatori
34     miopc->globale = ATESTA * NUM_THREAD_P ; // tot. prodotti da inserire
35     sem_init(&miopc->pieno, 0, 0);           // falso: inizio buffer non pieno
36     sem_init(&miopc->vuoto, 0, 1);            // vero : inizio buffer vuoto
37 }

```

Nella entry che inserisce i dati effettuiamo alcune modifiche rispetto a quella scritta in precedenza che utilizzava le **var condition**: innanzi tutto spostiamo i semafori, in quanto come prima operazione ci mettiamo “eventualmente” in attesa nel caso in cui il buffer sia pieno (istruzione 40); solo successivamente, quando il buffer è vuoto, lo utilizziamo entrando nella sezione critica e quindi mettendo a rosso il **mutex** (istruzione 42):

```

39 void metti(monitor *miopc, int data){        // entry per l'inserimento
40     sem_wait(&miopc->vuoto);                // aspetta che sia vuoto il buffer
41     //-----
42     pthread_mutex_lock(&miopc->mutex);      // mutex di mutua esclusione a rosso
43     if (miopc->buffer == 0) {                  // se il buffer è vuoto
44         if (miopc->scritti == miopc->globale) // già scritti tutti
45             data = FINE;                      // valore TAPPO per i lettori
46         miopc->buffer = data;                 // scrivi il dato
47         miopc->scritti++;                   // contatore produzione -condiviso
48     }
49     pthread_mutex_unlock(&miopc->mutex);    // mutex di mutua esclusione a verde
50     //-----
51     sem_post(&miopc->pieno);               // risveglia eventuali consumatori
52 }

```

Analogo discorso per la entry che toglie il dato dal buffer: prima si controlla se il dato è pronto ed eventualmente lo si attende (istruzione 56); successivamente si entra nella sezione critica ponendo a rosso il **mutex** (istruzione 58) ed effettuando tutte le operazioni sulle variabili condivise:

```

54 int estrai(monitor *miopc){                // entry per l'estrazione
55     int data = FINE;
56     sem_wait(&miopc->pieno);              // si pone in attesa buffer pieno
57     //-----
58     pthread_mutex_lock(&miopc->mutex);    // mutex di mutua esclusione a rosso
59     if(miopc->buffer > 0){                  // se il buffer è pieno
60         data = miopc->buffer;              // leggi il dato
61         miopc->letti++;                  // contatore produzione -condiviso
62         miopc->buffer = 0;                 // svuota il buffer
63     }
64     // fase terminale
65     if (miopc->letti >= miopc->globale){ // letto tutto ciò che bisognava leggere
66         if (miopc->letti > miopc->globale){ // il dato letto non è valido
67             data = FINE;                  // scrivi per terminare i consumatori
68         }
69         sem_post(&miopc->pieno);          // risveglia altri eventuali consumatori
70         // per farli terminare anch'essi
71     }
72     pthread_mutex_unlock(&miopc->mutex); // mutex di mutua esclusione a verde
73     //-----
74     sem_post(&miopc->vuoto);            // risveglia eventuali produttori
75     fflush(stdout);
76     return data;
77 }

```

In entrambe le entry introduciamo le istruzioni che controllano la terminazione delle operazioni:

- ▶ di scrittura (istruzioni 44 e 45): quando sono stati prodotti tutti i dati, inseriamo il carattere convenzionale di terminazione (FINE) che abbiamo posto a -1;
- ▶ di lettura (istruzioni 65 e 67): quando sono stati letti tutti i dati, inseriamo come dato letto il carattere convenzionale di terminazione (FINE).

Il codice eseguito dal **produttore** è riportato di seguito e non presenta particolarità degne di osservazioni:

```

79  void *produttore(void *threadid){
80      long tid;
81      tid = (long) threadid;
82      printf("\n Ciao a tutti: sono il thread produci #%ld!", tid);
83      int n, dato;
84      for (n = 1; n <= ATESTA; n++){
85          sleep(rand() % DELAY);           // ritardo mentre produce il dato
86          dato = tid * 100 + n;
87          printf ("\n .. %d produce il dato %d ", tid, dato);
88          metti (&bufferComune, dato);
89          printf("\n .. %d ha inserito dato %d", tid, dato);
90          fflush(stdout);
91      }
92      printf ("\n >>>>> fine produttore %d", tid);
93      return NULL;
94  }
```

Analogo discorso per il codice eseguito dal **consumatore**:

```

96  void *consumatore (void *threadid){
97      long tid;
98      tid = (long) threadid;
99      printf ("\\n Ciao a tutti: sono il thread consuma #%ld!", tid);
100     char miobuffer = '\x00';
101     int dato;
102     do {
103         printf("\n ... %d pronto x consumare ", tid );
104         dato = estrai(&bufferComune);
105         if (dato != FINE) {
106             printf("\n .... %d consumato il dato %d", tid, dato);
107             fflush(stdout);
108         }
109     } while (dato != FINE);
110     printf ("\n >>>>> fine consumatore %d", tid);
111     return NULL;
112 }
```

Leggermente più articolato il **main()**: avendo messo parametrico il numero dei **thread** è necessario memorizzarli in due vettori, rispettivamente uno per i **produttori** (**thread_P**) e uno per i **consumatori** (**thread_C**).

A ciascun **thread** viene passato un numero identificatore così da poter seguire la sua evoluzione: tale numero viene anche utilizzato per individuare il dato da lui prodotto (istruzione 86).

```

114 void main(){
115     int tc, tp;
116     srand((int) time(NULL));
117     inizializza(&bufferComune);
118     pthread_t thread_P[NUM_THREAD_P];
119     pthread_t thread_C[NUM_THREAD_C];
120     printf ("\nInizio elaborazione: creazione thread produttori e consumatori.");
121
122     for (tp = 0; tp < NUM_THREAD_P; tp++){           // produttori
123         if (pthread_create(thread_P + tp, NULL, produttore, (void *)tp + 1) < 0){
124             fprintf(stderr, "errore nella creazione del thread P%d\n", tp);
125             exit(1);
126         }
127     }
128     for (tc = 0; tc < NUM_THREAD_C; tc++){           // consumatori
129         if (pthread_create(thread_C + tc, NULL, consumatore, (void *)tc + 1) < 0){
130             fprintf(stderr, "errore nella creazione del thread C%d\n", tc);
131             exit(1);
132         }
133     }
134     printf ("\nAttesa terminazione della elaborazione ... ");
135     for (tp = 0; tp < NUM_THREAD_P; tp++)
136         pthread_join(thread_P[tp], NULL);
137     for (tc = 0; tc < NUM_THREAD_C; tc++)
138         pthread_join(thread_C[tc], NULL);
139
140     printf("\nFine elaborazione !");
141     fflush(stdout);
142 }
...

```

Terminano il programma i due cicli di attesa con le **join** rispettivamente dei **produttori** e **consumatori**.

Una esecuzione con 2 **produttori** che producono ciascuno 2 dati e due **consumatori** è la seguente:

```

Inizio elaborazione: creazione thread produttori e consumatori.
Ciao a tutti: sono il thread produci #1!
Ciao a tutti: sono il thread produci #2!
Ciao a tutti: sono il thread consuma #1!
... 1 pronto x consumare
Ciao a tutti: sono il thread consuma #2!
... 2 pronto x consumare
Attesa terminazione della elaborazione ...
... 2 produce il dato 201
... 2 ha inserito dato 201
.... 1 consumato il dato 201
... 1 pronto x consumare
... 1 produce il dato 101
... 1 ha inserito dato 101
.... 2 consumato il dato 101
... 2 pronto x consumare
... 1 produce il dato 102
... 1 ha inserito dato 102
>>>>>>> fine produttore 1
2 ha inserito dato 202
>>>>>>> fine produttore 2
.... 1 consumato il dato 102
... 1 pronto x consumare
.... 2 consumato il dato 202
... 2 pronto x consumare
>>>>>>> fine consumatore 1
>>>>>>> fine consumatore 2
Fine elaborazione !

Process exited after 2.137 seconds with return value 0
Premere un tasto per continuare . .

```



Prova adesso!

- Monitor con semafori sem_t
- Problema produttore/consumatori
- Buffer circolare

Scrivi un programma C che crea due gruppi di **thread produttore e consumatore**.

La risorsa condivisa è un **buffer circolare** di dimensione **BUFFER_SIZE** (per esempio 20) il cui stato è identificato dalle seguenti variabili:

- 1 numero di elementi contenuti: **contaDati**;
- 2 puntatore alla prima posizione libera: **posScrittura**;
- 3 puntatore al primo elemento occupato: **posLettura**;
- 4 tempo per la preparazione di un dato: **DELEYP**;
- 5 tempo per consumare un dato: **DELEYC**.

Il **produttore** inserisce **TANTI** numeri interi in maniera sequenziale e il consumatore li estrae sequenzialmente, in ordine di inserimento, per stamparli.

Mentre le **var condition** gestivano "automaticamente" il **mutex**, con i **semafori** è necessario settarlo manualmente a ogni sospensione/risveglio, per esempio con il seguente codice:

```

42 void inserisci (monitorC *monitor, int data){ // entry per l'inserimento
43     pthread_mutex_lock (&monitor->lock); // mutex di mutex esclusione a rosso
44     /*
45     while (monitor->contaDati == BUFFER_SIZE){ // controlla che il buffer non sia pieno
46         pthread_mutex_unlock (&monitor->lock); // mutex di mutex esclusione a verde
47         sem_wait(&monitor->vuoto); // si pone in attesa di spazio nel buffer
48         pthread_mutex_lock (&monitor->lock); // mutex di mutex esclusione a rosso
49     }

```

Confronta la tua soluzione con quella presente nel file **monitorSemCirco.c**.

ESERCITAZIONI DI LABORATORIO 9

I SEMAFORI IN JAVA



Info

I codici seguenti sono reperibili nel file **Java_sincronizzazione_rar** scaricabile dalla cartella **materiali** nella sezione del sito www.hoepiscuola.it riservata al presente volume.

■ Modello ad ambiente globale: interazione tra thread

In Java i **thread** hanno come naturale meccanismo di comunicazione il modello ad **ambiente globale**: vediamo un primo esempio di come i **thread** possono interagire tra loro comunicando nell'area dati comune del processo che li ha generati (ambiente globale) mediante due modalità:

- innanzitutto definiamo un oggetto di tipo integer nell'area dati del processo padre e passiamo il reference di tale oggetto al costruttore dei **thread**;
- successivamente sfruttiamo le regole di scooping accedendo direttamente dai **thread** alle singole variabili.

Per garantire l'unicità di variabili e oggetti che dovranno essere condivisi è necessario dichiararli con la clausola **static**, cioè come **variabili di classe**.

In questo primo esempio comunichiamo attraverso entrambe le modalità descritte, e cioè:

- direttamente nell'area dati indirizzando la variabile **x** della **classe InComune1**;
 - passando ai **thread** un reference di un oggetto **Contatore** creato nel processo padre (conta).
- Scriviamo dapprima la **classe Contatore**:

```

1 class Contatore{
2     int valore;
3     int passo;
4     Contatore(int valore, int passo){
5         this.valore = valore;
6         this.passo = passo;
7         System.out.println("\nIl contatore e' nato e vale " + this.getValore());
8     }
9     void incrementa(){
10         valore += passo;
11     }
12     void decrementa(){
13         valore -= passo;
14     }
15     int getValore(){
16         return valore;
17     }
18 }
```

Nel costruttore del **Contatore** è stata inserita l'istruzione 7 **println()** unicamente per visualizzare dove viene creata e inizializzata la variabile **valore** (istruzione 33 del thread **main()**).

Proprio con l'istruzione 33 del metodo **main()** creiamo un oggetto Contatore e passiamo ad entrambi i **thread** il suo reference in modo che possano condividerlo:

```

public class InComune{
    // a) ambiente globale visibili da ogni thread
    protected static int x = 100;

    public static void main(String [] args){
        // b) oggetto comune passato al thread
        Contatore conta = new Contatore(0,1);
        Thread thri = new UnThread1("Yogi",conta);
        Thread thr2 = new UnThread1("Bubu",conta);
        thri.start();
        thr2.start();
    }
}

```

Vediamo ora la **classe UnThread1**: nella dichiarazione delle variabili di classe è presente l'identificatore dell'oggetto della **classe Contatore** e il reference di tale oggetto viene passato come parametro mediante il costruttore della **classe UnThread1** (insieme al nome da assegnare all'oggetto stesso, cioè al thread stesso).

Il corpo del **thread**, cioè il metodo **run()**, è costituito da un ciclo infinito (che poi nell'esempio viene interrotto alla nona interazione!) che:

- ▷ dapprima visualizza il contenuto delle variabili locali e delle variabili globali;
- ▷ quindi invoca **conta.incrementa()** (che è un metodo della **classe Contatore**) sull'oggetto comune: il risultato è l'incremento del valore della variabile **valore** (attributo di tale oggetto):

```

class UnThread1 extends Thread{
    // stato dell'oggetto
    private int inizia = 0;           // Variabile interne di servizio inizializzate dal costruttore
    private int delta = 1;
    private String micidome = "";
    Contatore conta;
    // costruttore
    UnThread1(String nomethread, Contatore conta){
        this.micidome = nomethread;
        this.conta = conta;
    }
    public void run(){
        for(;;){                     // ripeti per sempre
            System.out.println(micidome+"inizia="+inizia+", x="+InComune.x+", contatore="+conta.getValore());
            inizia += delta;
            InComune.x++;           // modifica direttamente le variabili della classe che lo crea riconosciute
            conta.incrementa();     // modifica x
            try {Thread.sleep(1500);} // dorme
            catch (InterruptedException e){System.out.println(e);}
            if(inizia > 5)           // termina dopo 5 ripetizioni
                return;
        }
    }
}

```

```

Il contatore e' nato e vale 0
Bubu:inizializza=0, x=100, contatore=0
Yogi:inizializza=0, x=100, contatore=0
Yogi:inizializza=1, x=102, contatore=2
Bubu:inizializza=1, x=102, contatore=2
Yogi:inizializza=2, x=104, contatore=4
Bubu:inizializza=2, x=104, contatore=4
Yogi:inizializza=3, x=106, contatore=6
Bubu:inizializza=3, x=106, contatore=6
Yogi:inizializza=4, x=108, contatore=8
Bubu:inizializza=4, x=108, contatore=8
Yogi:inizializza=5, x=110, contatore=10
Bubu:inizializza=5, x=110, contatore=10

```

Avviandone una esecuzione otteniamo il seguente output: ▶

Osserviamo come le variabili interne di conteggio **inizia** sono “locali” ai singoli **thread** in quanto ogni **thread** incrementa la propria, mentre sia la variabile **valore** dell’oggetto **conta** sia la variabile **x** della classe **InComune1** ad ogni iterazione di ciascun **thread** “subiscono” un incremento, cioè sono *effettivamente comuni ai due thread*.

L’output può però anche mostrare la presenza di qualche problema di **interleaving**: se ripetiamo più volte l’esecuzione del programma probabilmente avremo sempre sequenze diverse. È necessario introdurre i meccanismi che permettono la corretta gestione della risorsa condivisa.

■ Il qualificatore **synchronized**

Nei **thread** la risorsa condivisa è costituita dalle variabili globali e quindi implementiamo le primitive per accedere ad esse e utilizzarle. Tutta la libreria di oggetti di **Java** è **thread-safe** e quindi tutte le classi che scriveremo genereranno oggetti che godono di tale proprietà, permettendoci l’accesso condiviso.

CLASSE THREAD-SAFE

Una classe è detta **thread-safe** se vi si può accedere contemporaneamente da un insieme di **thread**.

In **Java** i **thread** utilizzano i monitor di **Hoare** come meccanismo di sincronizzazione nel modello ad **ambiente globale**: ogni istanza di una classe **thread-safe** (e quindi di ciascuna classe) ha associato un **monitor** e i metodi di tale classe sono eseguiti in mutua esclusione. L’implementazione avviene modificando la notazione dei metodi nel modo seguente.

```

1  public class Sincronizzata {
2      ...
3
4      public synchronized void metodo1(...) {
5          // sezione critica
6          ...
7      }
8
9      ...
10     public synchronized void metodo2(...) {
11         // sezione critica
12         ...
13     }
14 }
```

Se il metodo “è lungo” potrebbe di conseguenza risultare lunga l’attesa degli altri all’ingresso del monitor; è anche possibile restringere la **regione critica** a una porzione del metodo, cioè a un singolo blocco di istruzioni, mediante il costrutto:

```

public void metodo3(...){
    // sezione non critica
    ...
    synchronized(this) {
        // sezione critica
        ...
    }
}
```

Il qualificatore **synchronized** garantisce che solo un **thread** alla volta possa eseguire tale metodo, mandando gli altri in uno stato di attesa: l'insieme dei **thread** in attesa per l'utilizzo di un oggetto prende il nome di **wait-set**.

■ Realizzazione dei semafori in Java

Java non ammette **semafori** esplicativi come elementi del linguaggio, ma proprio per la sua natura di linguaggio a oggetti permette di costruire una classe che realizza i **semafori** di Dijkstra.

A tal fine la classe **Object** mette a disposizione due metodi, **wait()** e **notify()**, che ci permettono di scrivere le primitive **P(S)** e le **V(S)**:

- l'operazione **wait()** sospende l'esecuzione del **thread** e lo colloca nella lista in attesa del verificarsi di una condizione; il **thread** rilascia la risorsa e rilascia la mutua esclusione;
- l'operazione **notify()** segnala che si è verificata una modifica in una condizione, e lo segnala alla lista dei **thread** sospesi su quella condizione, o meglio su quell'oggetto (**wait-set**), in modo tale che uno di essi possa riprendere l'esecuzione dallo stato di attesa.

È necessario poi porre particolare attenzione a quanto di seguito indicato:

- l'istruzione **wait()** deve essere inserita in un costrutto try/catch in quanto può essere sospesa dal metodo interrupt che genera un'eccezione **InterruptedException**;
- sia **wait()** che **notify()** devono essere chiamate da un metodo **synchronized**, altrimenti si incorre nell'eccezione **IllegalMonitorStateException**.

Scriviamo ora la **classe Semaforo** dove realizziamo le due primitive **P()** e **V()** con due metodi ad accesso in mutua esclusione tramite la clausola **synchronized**:

```

1 class Semaforo{
2     int valore;                                // 0 = rosso
3     public Semaforo(int v){
4         valore = v;
5     }
6
7     synchronized public void P(){
8         while (valore == 0){                      // semaforo rosso
9             try { wait();}                         // il thread mi sospende
10            catch(InterruptedException e){}
11        }
12        valore--;                                // pone il semaforo a rosso
13    } //end P
14
15     synchronized public void V(){
16         valore++;                                // pone semaforo a verde
17         notify();                               // risveglia thread in coda
18    } //end V
19 }
```

Il **costruttore** ci permette di inizializzare il **semaforo** a rosso oppure a verde: utilizziamo questa classe per implementare una situazione tipica di produttore/consumatore.

ESEMPIO **Un produttore e un consumatore regolato da semaforo**

Realizziamo una semplice situazione di produttore/consumatore, dove il produttore scrive in sequenza N numeri e il consumatore li visualizza sullo schermo.

Dichiariamo una variabile globale **buffer** che servirà memoria condivisa per i due **thread** e due **semafori** (**pieno**, **vuoto**) che ne permetteranno l'accesso: rispettivamente il primo che segnala quando il buffer è pieno e il secondo che segnala quando il buffer è vuoto.

I semafori vengono passati come parametri nel costruttore dei due **thread** nel **main()** della classe di prova: ►

```

1 public class ProdConsSem {
2     protected static int buffer;           // variabile condivisa globale
3     public static void main(String args[]){
4         Semaforo pieno = new Semaforo(0);    //inizialmente rosso
5         Semaforo vuoto = new Semaforo(1);    //inizialmente vuoto
6         Producidoato prod = new Producidoato(pieno, vuoto);
7         ConsumaDato cons = new ConsumaDato(pieno, vuoto);
8         prod.start();
9         cons.start();
10        System.out.println("main: termine avvio thread.");
11    }
12 }
```

Il **thread Producidoato** si limita a scrivere un dato nel buffer quando trova verde il **semaforo** che indica buffer vuoto: altrimenti si sospende e, al termine della operazione di scrittura, setta a verde il **semaforo** che risveglia il **consumatore**.

```

1 class Producidoato extends Thread{
2     Semaforo pieno;
3     Semaforo vuoto;
4     int tanti = 5;                      // numero di elementi da produrre
5     final int attesa = 500;              // tempo di riposo/attesa
6     public Producidoato(Semaforo s1, Semaforo s2){
7         pieno = s1;
8         vuoto = s2;
9     }
10
11
12     public void run(){
13         for (int k = 0; k < tanti; k++){ // per tutti i dati da produrre
14             vuoto.P();                // produce un numero
15             Producidoato.buffer = k;
16             System.out.println("Scrittore: dato scritto :" + k);
17             pieno.V();
18             try {Thread.sleep(attesa);}
19             catch (Exception e){}
20         }
21         System.out.println("Scrittore: termine scrittura dati.");
22     } //fine run
23 }
```

Il **thread ConsumaDato** si limita ad aspettare che un dato sia pronto da leggere e quando lo ha consumato setta a verde il semaforo che risveglia il **produttore**. ►

```

1 class ConsumaDato extends Thread {
2     Semaforo pieno;
3     Semaforo vuoto;
4     int dato;
5     public ConsumaDato(Semaforo s1, Semaforo s2){
6         pieno = s1;
7         vuoto = s2;
8     }
9
10
11     public void run() {
12         while (true){
13             pieno.P();
14             dato = Producidoato.buffer; // consuma un numero
15             System.out.println("Lettore: dato letto "+dato);
16             vuoto.V();
17         }
18     } //fine run
19 }
```

Mandando in esecuzione il programma, si ottiene il seguente output: ►

Possiamo osservare che in questa implementazione si verifica una situazione indesiderata: il **thread consumatore** non termina mai!

Sarà quindi necessario "rivedere" la strutturazione del programma.

```
Main: termine avvio thread.
Scrittore: dato scritto :0
Lettore: dato letto 0
Scrittore: dato scritto :1
Lettore: dato letto 1
Scrittore: dato scritto :2
Lettore: dato letto 2
Scrittore: dato scritto :3
Lettore: dato letto 3
Scrittore: dato scritto :4
Lettore: dato letto 4
Scrittore: termine scrittura dati.
```



Prova adesso!

- 1** Modifica il programma precedente facendo in modo che il consumatore termini la sua esecuzione dopo che ha letto l'ultimo dato prodotto dal consumatore.
- 2** Modifica il programma utilizzando come buffer un oggetto di tipo contatore utilizzando la classe Contatore definita nel primo esempio.
- 3** Modifica il programma precedente in modo che siano presenti due produttori e due consumatori: il primo produttore incrementa il dato di una sola unità mentre il secondo lo raddoppia. A video i consumatori visualizzano oltre al valore letto anche il proprio nome.
- 4** Realizza un programma per sincronizzare un produttore che riempie un buffer circolare di 10 elementi e tre consumatori che estraggono dalla coda un elemento ciascuno.
Il produttore incrementa un contatore a ogni scrittura e il consumatore lo visualizza sullo schermo, assieme al proprio identificatore.
- 5** Successivamente realizza una situazione dove N produttori e M consumatori utilizzano una risorsa con memoria limitata, per esempio composta da 10 elementi. Manda in esecuzione una possibile situazione di tre produttori e tre consumatori che rispettivamente inseriscono un numero progressivo e lo leggono dal buffer visualizzandolo sullo schermo.
- 6** Realizza un programma per sincronizzare tre processi che rispettivamente visualizzano il suono di una campana in modo da ottenere la sequenza infinita di "DIN" "DON" "DAN".
- 7** Realizza un programma che sincronizzi la situazione lettori/scrittori, dove più lettori possono contemporaneamente accedere alla risorsa condivisa mentre gli scrittori devono alternarsi, nei seguenti casi:
 - A** privilegiando i lettori
 - B** privilegiando gli scrittori
 - C** alternando il più possibile lettori e scrittori

AREA digitale



Esercizi per il recupero / Esercizi per l'approfondimento

 hoepliscuola.it

ESERCITAZIONI DI LABORATORIO 10

I MONITOR IN JAVA

■ Monitor Java

Il linguaggio **Java** si presta alla implementazione del **monitor** in quanto basta definirlo come classe e sfruttare i metodi come primitive entry che accedono alle risorse condivise: naturalmente questi metodi devono essere qualificati con **synchronized** in modo da essere eseguiti singolarmente.

I progettisti di **Java** hanno deciso di utilizzare i **monitor** come prevalente metodo di sincronizzazione reputando questo meccanismo "il più elegante" per aggiungere primitive di sincronizzazione in un linguaggio.

Quindi la struttura di un **monitor** è la seguente:

```
public class MioMonitor {  
    < variabili condivise>  
    ...  
    public synchronized void metodo1(...) {  
        // sezione critica  
        ...  
    }  
    public synchronized void metodo2(...) {  
        // sezione critica  
        ...  
    }  
    public void metodo3(...){  
        // sezione non critica  
        ...  
        synchronized(this){  
            // sezione critica  
            ...  
        }  
    }  
    ...  
}
```

Quando si invocano le **entry** sincronizzate su di un oggetto **MioMonitor**, questo viene bloccato e quindi abbiamo **mutua esclusione** su di esso, dato che nessun altro processo può invocare nessun altro metodo dello stesso **oggetto**.

Le primitive `wait()` e `notify()` invocate all'interno di metodi `synchronized` evitano il verificarsi di **race condition**:

- ▶ `wait()` invocata senza parametri resta in attesa fino a che non viene risvegliata da una `notify()`: in alternativa è possibile specificare un parametro che indica il numero di milisecondi da attendere prima del risveglio: `wait(long timeout)`;
- ▶ `notify()` sblocca un processo ma non lo esegue subito, ossia permette al codice che ha chiamato la `notify` di continuare la sua esecuzione.

La primitiva `notify()` di Java sblocca uno dei processi in attesa, ma il chiamante non sa quale, dipende dalla JVM: esiste una ulteriore primitiva, la `notifyAll()`, che sblocca tutti i processi in attesa sulla condizione; semplifica molto la programmazione, ma non è molto efficiente.

Le tre primitive `wait()`, `notify()` e `notifyAll()` possono solo essere chiamate se il processo che le invoca possiede in quel momento il **monitor**.

Vediamo un primo semplice esempio di costruzione e utilizzo di un **monitor** e delle primitive `wait()` e `notify()`.

ESEMPIO

Il seguente programma definisce come **monitor** una sveglia costituita da due metodi:

- 1 il primo metodo, `dormi()`, mette in attesa chi lo invoca per un tempo passato come parametro;

```

1 class Sveglia{                                // monitor
2     private int ritardo;
3     synchronized void dormi(int quanto){
4         ritardo = quanto;
5         System.out.println(" 1 impostata la sveglia tra: " + ritardo/1000 + " secondi");
6         try{
7             wait();
8         }catch(InterruptedException e){
9             e.printStackTrace();
10        }
11    } // fine synchronized

```

- 2 il secondo metodo, `risveglia()`, aspetta il tempo “stabilito” per il sonno del “dormiglione” e quindi lo risveglia con `notify()` all'istruzione 39:

```

1 synchronized void risveglia(){
2     System.out.println(" 2 metodo risveglia ...");
3     try{
4         Thread.sleep(ritardo);
5     }catch(InterruptedException e){
6         e.printStackTrace();
7     }
8     notify();
9     System.out.println(" 3 notificata sveglia");
10 }
11 } // fine synchronized

```

Un oggetto **Sveglia** viene condiviso tra il `main()` che, di fatto, è anch'esso un **thread** come gli altri, e un secondo **thread**, il “dormiglione”.

Il funzionamento desiderato è il seguente:

- il **main()** crea un **thread** e viene posto “a dormire”, passandogli una sveglia e il tempo che deve dormire;
- il **main()** si riposa in modo che venga schedulato e si “addormenti” con la **sleep()**: quindi risveglia il **thread**;
- il **thread** dormiente viene risvegliato e termina la sua esecuzione.

```

1 class SvegliaChiDorme{
2     static int delay = 2000;
3     public static void main(String[] args){
4         Sveglia s = new Sveglia();           // monitor: sveglia temporizzata
5         Dormiente d = new Dormiente(s, delay); // thread: dorme e viene risvegliato dal main
6         d.start();
7         // attesa che il dormiglione ... dorma
8         try{
9             Thread.sleep(delay);
10        }catch(InterruptedException e){
11            e.printStackTrace();
12        }
13        synchronized(s){
14            s.risveglia();
15            System.out.println("Main-> Fine elaborazione !");
16        } // fine synchronized
17    } // fine main
18 } // fine class SvegliaChiDorme

```

Se nel **main()** non si aspettasse qualche secondo con l'**istruzione 9** questo occuperebbe il **monitor** prima che il processo dormiente esegua la **wait()** e quindi eseguirebbe il **notify()** in modo prematuro: il tale situazione la **wait()** verrebbe invocata dal dormiente solo successivamente alla terminazione del **main()** e il dormiente rimarrebbe in una attesa permanente!

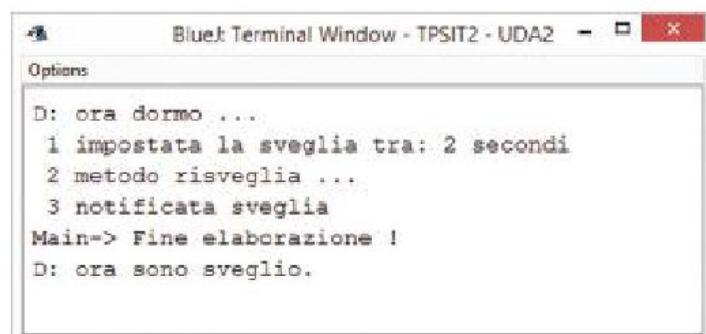
```

1 class Dormiente extends Thread {
2     Sveglia s;
3     int durataSonno;
4     Dormiente(Sveglia s, int durataSonno ) { //costruttore
5         this.s = s;
6         this.durataSonno = durataSonno;
7     } // fine costruttore
8     public void run(){
9         System.out.println("D: ora dormo ...");
10        s.dormi(durataSonno);
11        System.out.println("D: ora sono sveglio.");
12    } // fine run
13 } // fine classe dormiente

```

Il codice del dormiente è estremamente semplice: riceve come parametro per quanto tempo deve dormire nel costruttore e invoca il metodo della sveglia che lo “mette a dormire”.

Una esecuzione del programma mostra video il seguente output. ►



Vediamo ora un esempio completo di soluzione del problema produttore/consumatore.

ESEMPIO

Per prima cosa realizziamo un **monitor** che regola l'accesso a una risorsa: questa classe verrà in seguito utilizzata sia nella risoluzione delle diverse tipologie del problema **produttori/consumatori** che in quello **lettori/scrittori**.

Nel nostro monitor **Buffer** condividiamo due variabili:

- ▶ **dato**: contiene il dato prodotto da consumare;
 - ▶ **disponibile**: variabile booleana che segnala se il dato è nuovo, pronto da essere consumato.
- Scriviamo due **entry** per sincronizzare gli accessi:
- ▶ **togli()**: verifica se il dato è disponibile altrimenti si mette in attesa: quando può consumare “svuota il buffer” e modifica lo stato della disponibilità, quindi risveglia un **thread** sospeso;
 - ▶ **metti()**: verifica se il dato precedente è stato prelevato e in caso negativo si mette in attesa: quindi appena può inserisce un dato, aggiorna lo stato della variabile booleana e risveglia un **thread** sospeso.

```

1 class Buffer {
2     private int dato;                                //variabile contenente il dato
3     private boolean disponibile = false;            //indica se il dato e' disponibile
4
5     public synchronized int togli(){
6         while(disponibile == false){
7             try{wait();}                                //svuota il buffer
8             catch(InterruptedException e){}
9         }
10        disponibile = false;                         //svuota il buffer
11        notify();                                    //avvisa un processo tra i sospesi
12        return dato;
13    }
14
15    public synchronized void metti (int valore) {
16        while(disponibile == true){                  //non e' stato svuotato il buffer
17            try{wait();}                                //viene inserito un dato
18            catch(InterruptedException e){}
19        }
20        dato = valore;                            //si segnala la sua disponibilita'
21        disponibile = true;                         //avvisa un processo tra i sospesi
22        notify();
23    }
24 }
```

Utilizziamo questo **monitor** per risolvere il problema nel caso di un **produttore** e un **consumatore**: nel **main()** vengono creati due **thread** ai quali viene passato come parametro un oggetto “monitor” appena creato, il **buffer**.

```

1 public class UnProd_UnCons{
2     public static void main(String args[]){
3         int tanti = 5;                                // quantita' da produrre
4         Buffer buffer = new Buffer();                  // monitor condiviso
5         ProducI prodi = new ProducI(buffer, 1, tanti); // produttore di nome I
6         Consuma consl = new Consuma(buffer, 1, tanti); // consumatore di nome I
7         prodi.start();
8         consl.start();
9     }
10 }
```

Le classi **Producì** e **Consumà**, di seguito riportate, hanno un costruttore che riceve come parametro anche un numero intero che viene utilizzato per dare un “nome” ai singoli **thread**: in questo caso, avendo un solo **thread** produttore e consumatore, a entrambi viene passato il valore 1 come parametro e nel caso di N **thread** ci servirà per seguirne le singole evoluzioni.

```

1 class Producì extends Thread {
2     private Buffer buffer;           // monitor per il dato condiviso
3     private int nome;               // per dare un nome al produttore
4     private int tanti;              // num. dei dati da consumare
5     public Producì(Buffer buffer, int numero, int quanti){
6         this.nome = numero;
7         this.buffer = buffer;
8         this.tanti = quanti;
9     }
10    public void run(){
11        for(int xx = 0; xx < tanti; xx++){
12            buffer.metti(xx);
13            System.out.println("Produttore Nr. " + nome + " mette: " + xx);
14        }
15    }
16 }
```

Come terzo parametro ricevono entrambi la variabile **quanti** che contiene il numero degli elementi da produrre e da consumare, utilizzata nel metodo **run()**.

```

1 class Consumà extends Thread{
2     private Buffer buffer;           // monitor per il dato condiviso
3     private int nome;               // identificativo del consumatore
4     private int tanti;              // num. dei dati da consumare
5     public Consumà(Buffer buffer, int numero, int quanti){
6         this.nome = numero;
7         this.buffer = buffer;
8         this.tanti = quanti;
9     }
10    public void run(){
11        for(int xx = 0; xx < tanti; xx++){
12            int valore = buffer.tegli();
13            System.out.println("Consumatore Nr. " + nome+ " prende : " + valore);
14        }
15    }
16 }
```

Mandando il programma in esecuzione si ottiene l'output riportato a fianco: ►

L'output a schermo potrebbe anche non visualizzare “cronologicamente” gli eventi dato che è fortemente dipendente dalla schedulazione: è comunque evidente che un dato per essere letto deve essere stato prodotto, a prescindere dall'ordine dell'echo che i diversi **thread** presentano a schermo.

```

Bluet Terminal Window - TPSIT2 - UDA2 - □ X
Produzione Nr. 1 mette: 0
Consumatore Nr. 1 prende : 0
Produzione Nr. 1 mette: 1
Consumatore Nr. 1 prende : 1
Produzione Nr. 1 mette: 2
Consumatore Nr. 1 prende : 2
Produzione Nr. 1 mette: 3
Consumatore Nr. 1 prende : 3
Produzione Nr. 1 mette: 4
Consumatore Nr. 1 prende : 4

```



Prova adesso!

- 1** Modifica l'esempio precedente per generalizzare il problema nel caso che siano presenti due produttori e tre consumatori. Quindi sostituisci il buffer con un nuovo monitor, bufferCircolare con dimensione BUFFER_SIZE, e utilizza due variabili (`int readpos, writepos;`) per leggere/scrivere nel buffer:
 - ▷ `writepos` è il puntatore alla prima posizione libera;
 - ▷ `readpos` è il puntatore al primo elemento occupato.
- 2** Modifica l'esempio precedente per generalizzare il problema a N produttori e N consumatori che utilizzano sempre un buffer circolare con dimensione BUFFER_SIZE.
Effettua le necessarie modifiche al monitor e ai processi di prova nonché al main per:
 - ▷ 5 processi produttori e 10 processi consumatori;
 - ▷ 10 processi produttori e 5 processi consumatori;
 - ▷ 10 processi produttori e 10 processi consumatori.
- 3** Realizza un sistema in cui un produttore generi i numeri di Fibonacci (il primo è 1, il secondo è 2, ogni numero successivo è la somma dei due precedenti) introducendo il risultato in un buffer; il consumatore legge il numero e:
 - ▷ se il numero è corretto, prosegue regolarmente la lettura finché arriva al termine n-esimo;
 - ▷ se il numero non è stato aggiornato... aspetta;
 - ▷ al termine, visualizza tutti i numeri della sequenza.
- 4** Realizza un programma che sincronizzi la situazione lettori/scrittori, dove più lettori possono contemporaneamente accedere alla risorsa condivisa mentre gli scrittori devono alternarsi, nei seguenti casi:
 - ▷ privilegiando i lettori;
 - ▷ privilegiando gli scrittori;
 - ▷ alternando il più possibile lettori e scrittori.
- 5** Realizza un'applicazione concorrente per la gestione dello studio di un veterinario, unico per cani e gatti, dove ogni utente del veterinario (cane o gatto) è rappresentato da un **thread** e il suo studio come una risorsa. La politica di sincronizzazione tra i processi dovrà garantire che:
 - ▷ nello studio non vi siano contemporaneamente cani e gatti.
 - ▷ nell'accesso allo studio, i gatti abbiano la priorità sui cani.

Si supponga inoltre che lo studio abbia una capacità limitata a N animali. Il numero C di cani, quello dei gatti G e la capacità N dovranno essere passati alla riga di comando.
- 6** In un'autocarrozzeria si devono verniciare alcune auto: per ogni auto si deve alternare un certo numero di fasi di ceratura con fasi di lucidatura che vengono eseguite da due operai che sono specializzati, rispettivamente, nel processo di ceratura e in quello di lucidatura. Si deve simulare il comportamento dall'autocarrozzeria in JAVA:
 - ▷ si attivino due **thread** che simulino il comportamento dei due operai;
 - ▷ si definisca l'oggetto condiviso Auto, tramite cui interagiscono i due **thread**, con i metodi per la corretta sincronizzazione tra i due **thread**.

Il programma di prova Autocarrozzeria deve creare un oggetto Auto, attivare i due **thread**, passando a ciascuno di essi l'oggetto condiviso, lasciarli lavorare per un certo numero di secondi, quindi interromperli.

AREA digitale



Esercizi per il recupero / Esercizi per l'approfondimento

 hoepliscuola.it

ESERCITAZIONI DI LABORATORIO 11

UN ESEMPIO CON I JAVA THREAD: CORSA DI BICICLETTE

■ Un esempio completo: thread e grafica

Si vuole realizzare un programma che simula una corsa dove sei ciclisti si sfidano nella volata finale, visualizzando sullo schermo l'avanzamento di ogni concorrente fino a che tutti giungono al traguardo e la classifica finale.

Descrizione della soluzione

Realizziamo questo sistema definendo per ogni ciclista un **thread** che si muove con velocità costante per un certo intervallo di tempo (per esempio 10 unità di tempo) e successivamente, in modo casuale, viene modificata tale velocità per un nuovo intervallo di tempo e così via fino a che percorre lo spazio che lo separa dal traguardo del campo di gara.

Iniziamo a codificare la classe **CiclistaInGara** e il suo costruttore:

```

1 public class CiclistaInGara implements Runnable {
2     Ciclista ciclista;
3     GaraCiclistica campo;
4     int velocita;           // numero di pixel di spostamento al secondo: range 1-4
5     Thread t;
6     int conta;             // ogni 10 spostamenti cambio la velocità
7     int posizione;         // coordinate X in espressa in pixel
8
9     public CiclistaInGara(Ciclista c, GaraCiclistica g){
10         ciclista = c;           // identificativo del ciclista
11         campo = g;            // campo di gara - pista
12         conta = 0;
13         velocita = 2;
14         t=new Thread(this);
15         t.start();
16         posizione = 0;
17     }
18
19     public void run(){
20         //muove il ciclista lungo il percorso, cambiando la velocità
21         int dimImmagine = 75; // dimensione della JPG del corridore
22         int dimPista = 960;
23         while((ciclista.getCordx() + dimImmagine) < dimPista){ // corsa non finita
24             if((conta % 10) == 0) // ogni 10 spostamenti
25                 velocita = (int)(Math.random()*4 + 1); // modifica la velocità
26             ciclista.setCordx(ciclista.getCordx() + velocita);
27             conta++;
28             try{Thread.sleep(75);} // delay
29             catch(Exception e){}
30             campo.repaint();
31         }
32         //scrivo in che posizione è arrivato nella classifica finale
33         posizione = campo.getPosizione();
34         campo.controlloArrivo();
35     }
36 }
```

Il metodo **run()** genera la posizione del ciclista e ne varia la velocità ogni 10 spostamenti: ▶

Codifichiamo una classe **GaraCicistica** che effettua la gestione della corsa implementando il campo di gara, definendo e inizializzando i singoli **thread**.

L'intestazione e gli attributi della classe sono i seguenti:

```

1 import java.awt.*;
2 import javax.swing.*;
3
4 public class GaraCicistica extends JFrame{
5     int posizione;
6     Ciclista[] partecipanti;
7     CiclistainGara[] thread_partecipanti;
8     Campo pista;
9     Graphics offscreen;           // per la gestione del doppio buffering
10    Image buffer_virtuale;
11}

```

Il costruttore definisce la dimensione della pista, definisce e crea i sei partecipanti assegnando a ciascuno di essi la rispettiva corsia.

```

12 public GaraCicistica(){
13     //posiziono e disegno il percorso
14     super("Gara Cicistica");
15     setSize(1000,645);
16     setLocation(10,40);
17     setDefaultCloseOperation(EXIT_ON_CLOSE);
18     pista = new Campo();
19     partecipanti= new Ciclista[6];
20     thread_partecipanti= new CiclistainGara[6];
21     posizione = 1;
22     //aggiungo le immagini dei concorrenti e li associo ai Thread
23     int partenza = 15;           // posizioni della prima corsia
24     for(int xx = 0; xx < 6; xx++){
25         partecipanti[xx]= new Ciclista(partenza, xx + 1);
26         thread_partecipanti[xx]= new CiclistainGara(partecipanti[xx],this);
27         partenza=partenza + 100;      // posizione verticale - corsia del ciclista
28     }
29     // visualizzo la gara
30     this.add(pista);
31     setVisible(true);
32 }

```

Aggiungiamo due metodi che utilizzano le variabili comuni (posizione e arrivati) e quindi li definiamo **synchronized**:

- A il primo assegna la posizione al concorrente che ha appena raggiunto il traguardo;

```

14     public synchronized int getPosizione(){
15         return posizione++;
16     }

```

- B il secondo verifica se tutti i concorrenti hanno raggiunto il traguardo e, in tal caso, ri-chiama la visualizzazione della classifica.

```

17     public synchronized void controllaArrivo(){
18         boolean arrivati = true;
19         for(int xx = 0; xx < 6; xx++){
20             if(thread_partecipanti[xx].posizione == 6){
21                 arrivati = false;
22             }
23         }
24         if(arrivati){
25             visualizzaClassifica();
26         }
27     }

```

La classifica viene realizzata dal seguente metodo:

```

49 public void visualizzaClassifica(){
50     JLabel[] arrivi;
51     arrivi = new JLabel[6];
52     JFrame classifica=new JFrame("Classifica");
53     classifica.setSize(500,500);
54     classifica.setLocation(250,150);
55     classifica.setBackground(Color.BLUE);
56     classifica.setDefaultCloseOperation(EXIT_ON_CLOSE);
57     classifica.getContentPane().setLayout(new GridLayout(6,1));
58     //visualizza l'ordine di arrivo
59     for(int xx = 1; xx < 7; xx++){
60         for(int yy = 0; yy < 6; yy++){
61             if(partecipanti[yy].posizione == xx){
62                 arrivi[yy]=new JLabel(xx + " classificato ciclista in "+(yy+1)+" corsia");
63                 arrivi[yy].setFont(new Font("Times New Roman",Font.ITALIC,20));
64                 arrivi[yy].setForeground(Color.blue);
65                 classifica.getContentPane().add(arrivi[yy]);
66             }
67         }
68     }
69     classifica.setVisible(true);
70 }
```

Concludono la classe **GaraCiclistica** i metodi relativi al disegno sullo schermo mediante la tecnica del doppio buffering necessaria per eliminare lo sfarfallio di immagini in movimento:

```

1 public void update(Graphics g){
2     paint(g);
3 }
4
5 public void paint(Graphics g){
6     if (partecipanti != null){
7         Graphics2D screen = (Graphics2D)g;
8         buffer_virtuale = createImage(1000, 240);
9         offscreen = buffer_virtuale.getGraphics();
10        Dimension d = getSize();
11        pista.paint(offscreen);
12        for(int xx = 0; xx < 6; xx++){
13            partecipanti[xx].paint(offscreen);
14        }
15        screen.drawImage(buffer_virtuale, 0, 30, this);
16        offscreen.dispose();
17    }
18 }
```

e il metodo il main, ridotto alla sola creazione di un oggetto della classe:

```

1 public static void main(String[] s){
2     GaraCiclistica m = new GaraCiclistica();
3 }
4 }
```

Riportiamo per completezza la classe **Ciclista** che si occupa del caricamento e della assegnazione delle immagini ai rispettivi ciclisti: ▼

```

1 import java.awt.*;
2 import javax.swing.*;
3
4 public class Ciclista extends JPanel{
5     int cordx;
6     int cordy;
7     Image img;
8
9     public Ciclista(int yy,int xx){
10         cordx = 0;
11         cordy = yy;
12         setSize(50, 51);
13         Toolkit tk=Toolkit.getDefaultToolkit();
14         switch (xx){ // ogni ciclista ha la maglia di colore diverso
15             case 1: {img = tk.getImage("bici1.JPG");break;}
16             case 2: {img = tk.getImage("bici2.JPG");break;}
17             case 3: {img = tk.getImage("bici3.JPG");break;}
18             case 4: {img = tk.getImage("bici4.JPG");break;}
19             case 5: {img = tk.getImage("bici5.JPG");break;}
20             case 6: {img = tk.getImage("bici6.JPG");break;}
21         }
22         MediaTracker mt=new MediaTracker(this); // oggetto per la gestione di un num. arbitrario
23         mt.addImage(img, 1); // di immagini in parallelo
24         try{mt.waitForID(1);}
25         catch(InterruptedException e){}
26     }

```

e mette a disposizione i metodi per la gestione della coordinata x del ciclista stesso:

```

17 public void setCordx(int n){
18     cordx = n;
19 }
20
21 public int getCordx(){
22     return cordx;
23 }
24
25 public void paint(Graphics g){
26     g.drawImage(img, cordx, cordy, null);
27 }
28

```

Completa il programma la classe **Campo** che disegna il “campo di gara”, cioè la pista:

```

1 import javax.swing.*;
2 import java.awt.*;
3
4 public class Campo extends JPanel {
5     public void paint(Graphics g){
6         g.setColor(Color.green);
7         g.fillRect(0, 0, 1000, 645);
8         //Linee laterali
9         g.setColor(Color.white);
10        g.fillRect(0, 0, 1000, 10);
11        g.fillRect(0, 100, 1000, 10);
12        g.fillRect(0, 200, 1000, 10);
13        g.fillRect(0, 300, 1000, 10);
14        g.fillRect(0, 400, 1000, 10);
15        g.fillRect(0, 500, 1000, 10);
16        g.fillRect(0, 600, 1000, 10);
17        //Traguardo
18        g.fillRect(960, 0, 5, 645);
19        g.fillRect(970, 0, 5, 645);
20        g.fillRect(980, 0, 5, 645);
21    }
22 }

```

Nell'immagine seguente è mostrato un momento della corsa.



E una possibile classifica finale: ►

	Classifica
1	classificato ciclista in 5 corsia
2	classificato ciclista in 1 corsia
3	classificato ciclista in 6 corsia
4	classificato ciclista in 4 corsia
5	classificato ciclista in 2 corsia
6	classificato ciclista in 3 corsia



Prova adesso!

- 1 Modifica il programma sopra descritto in una corsa con staffetta: ogni ciclista non appena raggiunge il traguardo passa il testimone a un nuovo ciclista che effettua il percorso nel senso opposto fino a raggiungere il punto di partenza che diviene il nuovo traguardo.
- 2 Si aggiungano i nomi delle squadre e si visualizzi nella classifica il tempo totale e i tempi parziali di percorrenza dei singoli concorrenti.

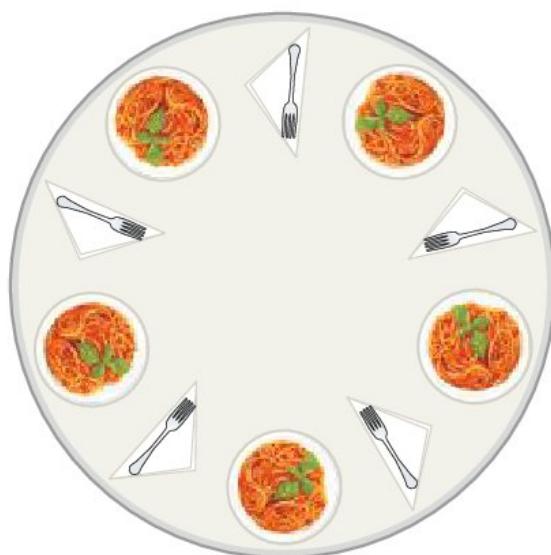
ESERCITAZIONI DI LABORATORIO 12

IL DEADLOCK IN JAVA

■ Filosofi e deadlock

Si vuole realizzare in linguaggio **Java** la soluzione del problema dei 5 filosofi utilizzando un **monitor** Tavolo sul quale sono posti i piatti e le forchette.

Ciascun filosofo, numerato con un valore da 0 a N (possiamo poi provare il caso di N filosofi) può compiere tre attività: pensare, mangiare e, nel caso non trovasse disponibili le forchette, aspettare. ►



La classe **Filosofo** è la seguente, dove viene passato al costruttore l'oggetto tavola e il nome che assegniamo al filosofo:

```

1 class Consumo extends Thread{
2     private Buffer buffer;           // monitor per il dato condiviso
3     private int nome;               // identificativo del consumatore
4     private int tanti;              // num. dei dati da consumare
5     public Consumo(Buffer buffer, int numero, int quanti){
6         this.nome = numero;
7         this.buffer = buffer;
8         this.tanti = quanti;
9     }
10    public void run(){
11        for(int xx = 0; xx < tanti; xx++){
12            int valore = buffer.togli();
13            System.out.println("Consumatore Nr. " + nome+ " prende : " + valore);
14        }
15    }
16 }
```

Nel metodo **run()** sono evidenziate le due attività del filosofo: quando vuole iniziare a mangiare accede al monitor con la entry **tavola.prendiForchette()** che verifica se il filosofo **nrFil** è in grado di prendere **contemporaneamente** entrambe le forchette, in modo da evitare la situazione di deadlock dato che in questo modo è stata rimossa la condizione di **hold & wait**.

Quindi mangia per un determinato tempo e successivamente rilascia le forchette utilizzando una seconda entry del monitor, la **tavola.lasciaForchette()** e inizia a pensare, per poi ripetere all'infinito queste due operazioni.

La classe di prova crea i cinque filosofi, una istanza della classe **monitor Tavola** e avvia l'esecuzione dei singoli **thread**:

```

1 public class FilosofiCena{
2     public static void main(String[] args){
3         Filosofo filosofi[];
4         filesofi = new Filosofo[5];
5         Tavola tavola = new Tavola();           // creazione del 5 filosofi
6         for(int n = 0; n < 5; n++)
7             filosofi[n] = new Filosofo(tavola, n);
8
9         for(int n = 0; n < 5; n++)           // avvio dei 5 filosofi
10            filosofi[n].start();
11    }
12 }
```



Prova adesso!

Realizza il monitor Tavola dove sono presenti le forchette e le due procedure/entry che permettono ai filosofi di prendere e rilasciare le forchette, secondo il seguente schema:

```
class Tavola{  
    int forchette[];           // risorse condivise  
    public Tavola() {  
        forchette=new int[5];  
        <inizializza le forchette>  
    }  
  
    public synchronized void prendiForchette(int nrFilosofo){  
        <se le forchette sono occupate>  
        <il filosofo si sospende>  
        <altrimenti prende le forchette>  
    }  
  
    public synchronized void lasciaForchette(int nrFilosofo){  
        <rilascia le forchette>  
        notifyAll();           // risveglia i filosofi in attesa  
    }  
}
```

Ora modifica il numero di filosofi e manda in esecuzione il programma: cosa puoi osservare?

3

I requisiti software

- L1 **La specifica dei requisiti**
- L2 **Raccolta e analisi dei requisiti**
- L3 **Attori, casi d'uso e scenari**
- L4 **La documentazione dei requisiti**

Esercitazioni di laboratorio

- ① La realizzazione degli Use Case Diagram con StarUML;
- ② La realizzazione degli Use Case Diagram con ArgoUML

Conoscenze

- Comprendere l'importanza della fase di analisi
- Avere il concetto di requisito utente e di sistema
- Avere il concetto di fase di esplorazione
- Conoscere le tecniche di esplorazione
- Avere il concetto di scenario e caso d'uso
- Comprendere le caratteristiche SRS

Competenze

- Individuare i requisiti utente
- Individuare i requisiti di sistema
- Utilizzare le tecniche di esplorazione
- Individuare gli scenari d'uso
- Analizzare il documento di Specifica dei Requisiti Software (SRS)
- Acquisire la struttura di un SRS

Abilità

- Saper descrivere in UML i casi d'uso
- Saper descrivere in UML il diagramma di contesto
- Saper documentare i casi d'uso
- Saper compilare il documento di Specifica dei Requisiti Software
- Validare le specifiche di un SRS

AREA *digitale*



Esercizi



Gli stakeholder

La norma ISO 13407

Non-functional requirements classification

Esempio di questionario per lo sviluppo di una piattaforma di e-commerce

Consigli per fare un buon Use Case Diagram

Esercizi per il recupero

Esercizi per l'approfondimento



Soluzioni (prova adesso, esercizi, verifiche)

Puoi scaricare il file anche da

La specifica dei requisiti

In questa lezione impareremo...

- le diverse tipologie di requisiti software
- i requisiti utente e i requisiti di sistema
- i requisiti funzionali e i requisiti non funzionali

■ Premessa

Nel **ciclo di vita del processo** informatico le prime fasi che seguono lo **studio di fattibilità** riguardano la **raccolta** e l'**analisi dei requisiti**.

Con lo **studio di fattibilità** viene effettuata una valutazione preliminare di **costi** e **benefici** nella quale si stabilisce se è conveniente avviare il progetto oppure si possono individuare opzioni o alternative più adeguate, come utilizzare e adeguare altri prodotti presenti in commercio, e quindi valutare le **risorse umane** e **finanziarie** necessarie.

Questa fase generalmente si conclude con la presentazione di una offerta al cliente (preventivo o documento di fattibilità nel quale sono indicati i costi, i tempi ed eventualmente le modalità di sviluppo per ogni alternativa proposta).

La formulazione del preventivo **impegna** la software house alla sua realizzazione e al rispetto di quanto in esso descritto: generalmente questo viene compilato da un **Software Architect Senior**.

Nella successiva fase, quella di **specifiche dei requisiti**, possiamo riconoscere le seguenti attività:

- **analisi del problema**: lo scopo dell'analisi del problema è quello di comprendere cosa deve fare il sistema software che si vuole realizzare; devono essere individuate le necessità degli utenti, la natura dell'ambiente operativo, le condizioni di mercato e/o di settore operativo, le condizioni di funzionamento;
- **definizione** delle funzionalità, dell'operatività, dei vincoli interni ed esterni alla azienda, delle prestazioni e di ogni caratteristica richiesta al sistema per soddisfare la necessità del cliente;

- ▶ redazione di un documento di **Specifiche dei Requisiti Software (SRS – Software Requirement Specification)** che, in poche parole, non è altro che la trasformazione dei requisiti in un documento formalizzato che raccoglie le specifiche tecniche e funzionali che caratterizzano il sistema;
- ▶ **convalida delle specifiche:** prima di procedere alla realizzazione il **SRS** viene analizzato e rivisto con il committente per validare ogni singola specifica individuata.

La **specifiche dei requisiti** risponde alla domanda "che **cosa** deve fare il sistema?" cioè stabilisce le funzionalità del sistema **senza** interessarsi di "come" queste funzioni verranno realizzate.

L'**analisi** e l'individuazione dei **requisiti** è di fondamentale importanza per poter sviluppare un software di alta qualità: un errore in questa fase comporta necessariamente la presenza anche di molti errori nel sistema finale.

L'insieme delle attività che si "occupano" di requisiti prende anche il nome di **ingegneria dei requisiti (requirements engineering)**.

In questa lezione inizieremo ad analizzare il concetto di **requisito** e ne studieremo alcune possibili classificazioni che ci permetteranno di comprenderne meglio le caratteristiche al fine di acquisire gli elementi essenziali per poter poi redigere correttamente l'**SRS**.

■ Requisiti software e stakeholder

Un **requisito** (dal latino *requisitus*, richiesto) è una proprietà richiesta, oppure auspicabile, del prodotto: il **documento dei requisiti** ha lo scopo di accogliere in forma organica una descrizione di tutte le proprietà desiderate e dalla sua formulazione dovrebbe essere chiaro se un requisito esprime una proprietà obbligatoria, oppure soltanto suggerita o auspicabile.

Possiamo dare una prima definizione di **requisito** (◀ **Software requirements** ▶):



REQUISITO

Ogni informazione (ottenuta in qualche modo) circa le funzionalità, i servizi, le modalità operative e di gestione del sistema da sviluppare.

◀ **Software requirements** Def. IEEE Std 610.12 (1990):

- (A) A condition or capability needed by a user to solve a problem or achieve an objective.
- (B) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
- (C) A documented representation of a condition or capability as in definition (A) or (B). ►



ESEMPIO

Per esempio, per un sito web di commercio elettronico potremmo identificare, fra gli altri, i seguenti cinque requisiti:

- ▶ il sito *deve* permettere all'utente di inserire nel carrello d'acquisto i prodotti di cui sta valutando l'acquisto;
- ▶ il carrello *deve* poter contenere almeno 15 prodotti contemporaneamente;

- ogni scheda prodotto contenuta nel catalogo *deve* contenere una fotografia a colori del prodotto, il suo nome, il nome del produttore, il prezzo e una descrizione sintetica ma completa, 5 righe di testo al massimo;
- il pagamento *deve* essere effettuato sia con carte di credito elettroniche che tradizionali;
- l'intero processo di acquisto di un prodotto *dovrebbe* richiedere al massimo 5 minuti.

La **definizione dei requisiti** rappresenta l'analisi completa dei **bisogni** dell'utente e del **dominio** del problema allo scopo di definire quello che il sistema deve fare.

La raccolta dei requisiti è spesso considerata l'attività più difficile, perché richiede la collaborazione tra più gruppi di partecipanti con differenti background: questa fase deve coinvolgere (**engagement**) diverse persone sia del team di sviluppo (programmatori, sistemisti ecc.) che dell'azienda del cliente (committente, end-users, managers ecc.) e, a volte, può anche richiedere la consulenza di terze parti, come consulenti esterni, per esempio, per l'analisi della legislazione e/o di realizzazioni pre-esistenti.

Le diverse persone coinvolte in tale processo sono chiamate gli ► **stakeholder** ▶.



STAKEHOLDER

"Gli **stakeholder** – o portatori di interesse – sono tutte quelle persone o gruppi che influenzano e/o sono influenzati dalle attività di un'organizzazione, dai suoi prodotti o servizi e dai relativi risultati di performance". (Freeman, 1984)

◀ **Stakeholder** A person, group or organization that has interest or concern in an organization. Stakeholders can affect or be affected by the organization's actions, objectives and policies. ▶



Sono da considerarsi **stakeholder** tutte le persone in qualche modo interessate alla messa in opera del sistema, a ogni livello della organizzazione.

All'inizio del progetto gli **stakeholder** difficilmente hanno una chiara idea di quello che esattamente vogliono e usano un proprio linguaggio tecnico (tipico del dominio dell'attività dell'azienda) che generalmente non è noto all'analista che deve quindi per prima cosa studiare il loro lessico specifico per poter dialogare e comprendere correttamente quanto gli viene esposto.

D'altra parte il cliente e gli utenti finali sono esperti nel loro dominio ma hanno generalmente poca (o nulla) esperienza nello **sviluppo del software** e idee spesso confuse e vaghe sulle funzionalità che il nuovo sistema vuole realizzare.

Inoltre sia le funzioni svolte che i fabbisogni individuali sono diversi a ogni livello della azienda e spesso gli **stakeholder** effettuano richieste diverse che possono anche sfociare in requisiti in conflitto tra loro.

Ogni **stakeholder** può fornire una descrizione astratta e imprecisa del sistema che l'analista deve essere in grado di elaborare per ottenere una descrizione dettagliata e matematica dello stesso.

AREA digitale
Gli stakeholder

Nella **analisi dei requisiti** bisogna tener presente non solo le esigenze dell'azienda ma anche i fattori esterni al sistema stesso, come le esigenze sopravvenute a causa di modifiche apportate nella organizzazione aziendale conseguenti all'adeguamento a nuove politiche di mercato o a obblighi legislativi.

Anche per questo motivo i **requisiti** non sono "stabili", ma possono cambiare sia durante la **fase di analisi** che in tutto il **ciclo di sviluppo** della applicazione informatica.

La fase di analisi dei requisiti è particolarmente delicata e l'introduzione di errori in questa fase può portare anche al fallimento dell'intero progetto: in ogni caso questi errori sono difficili da individuare e spesso vengono scoperti "troppo tardi", nelle ultime fasi del processo di sviluppo del software, quando l'intervento per risolverli risulta essere molto oneroso.

I principali rischi sono quelli di "dimenticare" o ignorare una funzionalità, di implementare in modo errato o incompleto una richiesta, di realizzare interfacce utenti poco intuitive e difficili da usare.

Un suggerimento per identificare i **requisiti** rilevanti in un processo di progettazione di sistemi informatici viene dato dalla normativa **ISO 13407**, anche conosciuta come **human-centered design process (UCD)**.

La filosofia di base si può riassumere in una frase: "*fondare il progetto sui reali bisogni degli utenti*".

Secondo l'**ISO 13407**:



"La progettazione centrata sull'essere umano (**human-centred design**) è un approccio allo sviluppo dei sistemi interattivi specificamente orientato alla creazione di sistemi usabili. È un'attività multi-disciplinare che incorpora la conoscenza e le tecniche dei fattori umani e dell'ergonomia. L'applicazione dei fattori umani e dell'ergonomia alla progettazione dei sistemi interattivi ne potenzia l'efficacia e l'efficienza, migliora le condizioni del lavoro umano e contrasta i possibili effetti avversi dell'uso sulla salute, sulla sicurezza e sulle prestazioni. Applicare l'ergonomia alla progettazione dei sistemi richiede che si tenga conto delle capacità, delle abilità, delle limitazioni e delle necessità umane. I sistemi human-centred supportano gli utenti e li motivano a imparare. I benefici possono includere una maggiore produttività, una migliore qualità del lavoro, riduzione dei costi di supporto e di addestramento e una migliore soddisfazione dell'utente".

Possiamo sintetizzare in un insieme di punti le indicazioni che la normativa ci indica come guida alla individuazione dei **requisiti** ed quindi alla definizione degli obiettivi di un sistema applicativo usabile; è necessario cioè definire:

- ▶ le prestazioni richieste al nuovo sistema in relazione agli obiettivi operativi ed economico/finanziari;
- ▶ i requisiti normativi o legislativi rilevanti, compresi quelli relativi alla sicurezza e alla salute;
- ▶ la comunicazione e la cooperazione fra gli utenti e gli altri attori rilevanti;
- ▶ le attività degli utenti, inclusa la ripartizione dei compiti, il loro benessere e le loro motivazioni;
- ▶ le prestazioni dei diversi compiti;
- ▶ la progettazione dei flussi di lavoro e dell'organizzazione;
- ▶ la gestione del cambiamento indotto dal nuovo sistema, incluse le attività di addestramento e il personale coinvolto;

- ▶ la fattibilità delle diverse operazioni, comprese quelle di manutenzione;
- ▶ la progettazione dei posti di lavoro e la interfaccia uomo-computer.

Prima di affrontare lo studio di una metodologia standardizzata per la definizione dei **requisiti** è necessario introdurre una loro classificazione.



La norma ISO 13407

■ Classificazione dei requisiti

I **requisiti software** possono essere classificati secondo due diversi punti di vista:

- ▶ **livello di dettaglio:**
 - requisiti **utente** (**user requirements**);
 - requisiti di **sistema** (**system requirements**);
- ▶ **tipo di requisito** che rappresentano:
 - requisiti **funzionali**;
 - requisiti **non funzionali**;
 - requisiti **di dominio**.

Livello di dettaglio

Analizzando i **requisiti** sulla base del livello di dettaglio, per ciascuna delle due categorie individuate abbiamo livelli di astrazione e formalismo diversi:

- ▶ i **requisiti utente** sono quelli che “osserva il cliente”, cioè le esigenze sentite dall’utente finale e descritte con il linguaggio del cliente: vengono sottoposti al team di sviluppo che ne propone una soluzione, a volte con diverse alternative (sono anche chiamati **requisiti aperti**);
- ▶ i **requisiti di sistema** sono quelli imposti da vincoli esistenti, come per esempio l’utilizzo di apparecchiature esistenti all’interno della azienda o di interfacciamento con sistemi aziendali già in funzione o anche di natura fiscale e/o legislativa (rispetto della normativa sulla sicurezza e sulla privacy, contabilità e bilancio secondo la normativa CEE ecc.). Sono solitamente molto strutturati e vincolanti, scritti in linguaggi tecnici e/o semi-formali e/o formali e non lasciano margini di “inventiva” data la loro natura (sono requisiti più restrittivi o **requisiti chiusi**): spesso non sono conosciuti all’utente ma noti solo al programmatore.

ESEMPIO

In un progetto dove vi è un unico requisito utente elenchiamo i possibili requisiti di sistema:

Requisito utente:

- ▶ l’applicazione deve permettere di rappresentare e visualizzare file esterni prodotti da altri pacchetti software.

Requisito di sistema:

- ▶ l’utente deve poter definire il tipo dei file esterni;
- ▶ l’utente deve poter associare a ogni file esterno il prodotto che lo ha generato;
- ▶ a ogni tipo di file deve essere associata una specifica icona per visualizzarlo sullo schermo;
- ▶ l’icona che rappresenta il tipo di file esterno deve poter essere scelta dall’utente;
- ▶ selezionando l’icona, il sistema deve mandare in esecuzione una applicazione in grado di visualizzare il contenuto del file.

Tipo di requisito

Se invece analizziamo i requisiti sulla base del tipo, abbiamo tre possibili raggruppamenti:

► i requisiti **funzionali** (*Functional Requirement*) descrivono le funzionalità che il sistema deve avere e tutti i servizi che dovrà offrire agli utenti.

I requisiti funzionali devono essere:

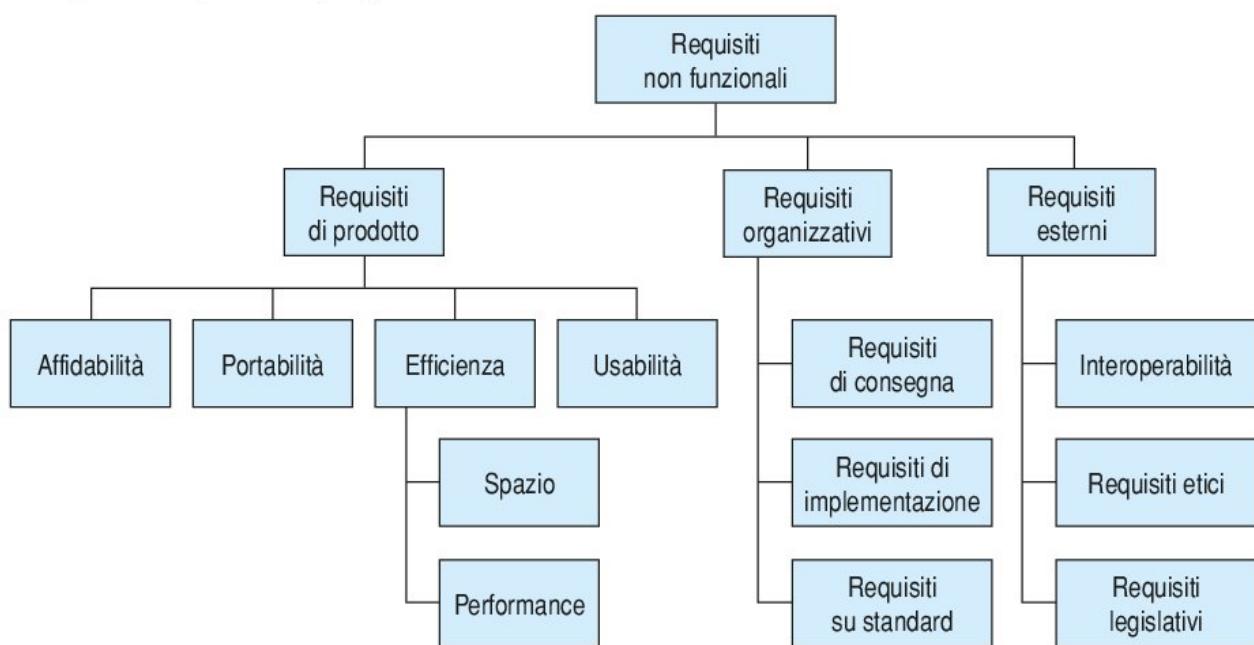
- **completi**: devono indicare tutti i servizi richiesti dagli utenti;
- **coerenti**: i requisiti non devono avere definizioni contraddittorie.

Abbiamo già detto come per grossi sistemi sia difficile ottenere requisiti completi e coerenti dato che spesso i vari stakeholders hanno esigenze diverse, spesso in contrasto;

► i requisiti **non funzionali** (*Non Functional Requirement*) sono quelli imposti dalle modalità operative, dal ciclo di vita del prodotto”, nel caso di un sistema di produzione, oppure dalla “catena del freddo” nel caso di un prodotto alimentare surgelato, o dalle precedenze nel triage in un pronto soccorso ecc., o del rispetto della normativa vigente: il sistema deve rispettare questi vincoli perché imposti dalla organizzazione e dall'esterno.

Un esempio tipico è il tempo di risposta che deve avere il sistema per ciascuna operazione richiesta dall'utente oppure la possibilità di utilizzare il sistema sia su una piattaforma fissa (PC) che mobile (*smartphone*).

Per i requisiti non-funzionali sono state proposte diverse classificazioni: riportiamo di seguito quella di **Sommerville** dove definisce un primo livello composto da tre tipologie di requisiti: di *prodotto*, *organizzativi* ed *esterni*.



► i requisiti **di dominio** (*Domain Requirements*) sono dipendenti dal dominio in cui il sistema deve operare, come la riservatezza nel caso di dati sensibili, le leggi della fisica e/o della tecnologia nel caso di impianti industriali, la sicurezza sul lavoro specifica per ogni tipo e settore di attività ecc.

Un esempio tipico è la richiesta di login (**user-id** e **password**) per accedere a un'area dati protetta.

AREA digitale

 Non-functional requirements classification

ESEMPIO

Analizziamo la seguente situazione per individuare le tipologie di requisiti.

Carta di credito

Una banca rilascia ai suoi clienti una carta di credito con la quale è possibile effettuare il pagamento degli acquisti e, presso uno sportello, effettuare di prelievo di contanti, visualizzare il saldo e l'estratto conto. Il sistema deve garantire un tempo di risposta inferiore al minuto, deve essere sviluppato su architettura X86 e deve essere disponibile a persone portatori di Handicap. Le operazioni di pagamento possono essere fatte entro un limite massimo mensile e quelle allo sportello richiedono una autenticazione tramite un codice segreto memorizzato sulla carta. Il sistema deve essere facilmente espandibile e adattabile alle future esigenze bancarie.

Già nella descrizione del progetto possiamo evidenziare la differenza tra requisiti funzionali, non funzionali e di dominio: i primi li indichiamo **in verde**, i secondi **in rosso** e gli ultimi **in blu**:

*Una banca rilascia ai suoi clienti una carta di credito con la quale è possibile effettuare il pagamento degli acquisti e, presso uno sportello, effettuare di prelievo di contanti, visualizzare il saldo e l'estratto conto. Il sistema deve garantire un **tempo di risposta inferiore al minuto**, e deve essere sviluppato su architettura X86 e deve essere disponibile a persone portatori di Handicap. Le operazioni di pagamento possono essere fatte entro un **limite massimo mensile** e quelle allo sportello richiedono una **autenticazione tramite un codice segreto memorizzato sulla carta**. Il sistema deve essere facilmente espandibile e adattabile alle future esigenze bancarie.*

Una **seconda classificazione** è chiamata modello **FURPS** (del 1987), ottenuto come acronimo di:

- ▶ **Functionality**: caratteristiche e capacità del programma, funzioni fornite, sicurezza dell'intero sistema;
- ▶ **Usability** (usabilità): legata alla semplicità di apprendimento e utilizzo del sistema da parte degli utenti (convenzioni per le interfacce utenti; organizzazione dell'Help in linea; tipologia e livello della manualistica);
- ▶ **Reliability** (affidabilità): il sistema deve garantire nel tempo le funzioni richieste per un determinato range di situazioni e fornire risposte sempre corrette (frequenza e severità delle failure, accuratezza degli output, capacità di recupero dalle failure, predicitività del programma);
- ▶ **Performance**: sono i requisiti legati al tempo di risposta del sistema, cioè la velocità nel fornire i risultati, la quantità di risorse utilizzate, il throughput e quindi l'efficienza;
- ▶ **Supportability** (supportabilità): l'adattabilità del sistema alle modifiche che devono essere apportate durante il suo esercizio (manutenibilità, estendibilità, adattabilità, compatibilità e configurabilità).

A questi vanno aggiunti i vincoli (o pseudo-requisiti):

- ▶ **implementazione**: vincoli sull'implementazione del sistema, incluso l'uso di tool specifici, linguaggi di programmazione, o piattaforme hardware;
- ▶ **interfacce**: vincoli imposti da sistemi esterni, incluso sistemi legacy e formati di cambio;
- ▶ **operazioni**: vincoli sull'amministrazione e sulla gestione del sistema;
- ▶ **packaging**: vincoli sulla consegna del sistema (vincoli sui mezzi di installazione);
- ▶ **legali**: riguardano licenze, regolamentazioni e certificazioni.

■ I requisiti: l'anello debole dello sviluppo software

Un celebre studio effettuato dallo **Standish Group** su un campione di 8000 progetti riporta risultati sconcertanti: 16% di successi, 53% di fallimenti parziali (gravi problemi sulla funzionalità, sui costi e/o sui tempi), 31% di fallimenti completi (progetto cancellato).

Se si analizzano nel dettaglio le motivazioni (riportate nella seguente tabella) per le quali i progetti falliscono

Motivo del fallimento	Percentuale
Requisiti incompleti	13,1
Mancato coinvolgimento dell'utente	12,4
Mancanza di risorse	10,6
Attese irrealistiche	9,9
Mancanza del supporto della direzione	9,3
Cambiamento dei requisiti	8,7
Mancanza di pianificazione	8,1
Non serviva più	7,5

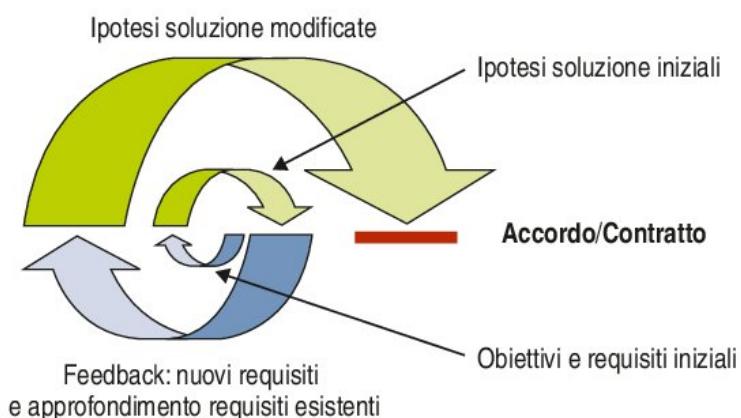
tra i primi otto fattori ne individuiamo cinque relativi a problemi connessi con i **requisiti**: requisiti incompleti, mancato coinvolgimento degli utenti, attese irrealistiche, cambiamento dei requisiti in corso d'opera, progetto cancellato perché non più utile.

L'intero ambito dei rapporti tra i **progettisti di sistemi** e i loro **clienti** risulta essere la causa prevalente del fallimento di un progetto software (51,6 % sull'80% delle principali cause).

La maggior parte delle volte il **fallimento del sistema** viene "scoperto" solo alla messa in opera del sistema, individuando una non conformità rispetto ai **requisiti** attesi o concordati che ne determina la mancata accettazione del prodotto da parte del committente: solo in questo momento emergono i conflitti di interpretazione che incrinano i rapporti tra le parti.

Le principali cause che determinano questo ritardo nell'individuazione del problema sono:

- A insufficiente azione di interazione e discussione tra gli attori coinvolti;
- B insufficiente individuazione dei conflitti tra **requisiti**, sempre presenti in ogni progetto;
- C mancato coinvolgimento degli utenti nella verifica dei **requisiti** individuati dagli analisti;
- D mancato coinvolgimento degli utenti per esprimere feedback sulle ipotesi prospettate dai progettisti;



- E** insufficiente controllo sull'evoluzione dei **requisiti** durante l'intero ciclo di vita del sistema;
- F** rinvio della presa in carico del nuovo **requisito** sopraggiunto in corso d'opera a una release successiva;
- G** mancato accordo sui contenuti e/o i costi e/o i tempi durante la rinegoziazione per l'aggiunta di un **requisito** sopraggiunto in corso d'opera.

È importante introdurre una metodologia di specifica da utilizzarsi in tutte le **fasi di analisi**: la raccolta dei requisiti deve necessariamente essere fatta prima che i progettisti inizino a proporre qualche soluzione, in quanto sono essenziali sia durante la **fase di realizzazione** che dopo che il sistema è stato realizzato, cioè durante la **fase di collaudo**.

Negli ultimi anni sono nati strumenti dedicati alla gestione dei requisiti (**◀ requirements management tools ▶**), che verranno trattati nella prossima lezione.

Il sistema finale deve essere raggiunto con una specifica soluzione che risponda in modo adeguato ai requisiti espressi dal committente e dagli altri attori coinvolti, deve avere caratteristiche comprensibili per tutti gli utenti, deve essere di semplice utilizzo e con funzionalità chiare, prive di ambiguità.

◀ Requirements management tools A requirements management tool is a software system that helps you manage the various manually intensive tasks in the requirements development and requirements management processes. Good requirements management tools (RM Tools) can help your team to save time and increase their productivity among other benefits. ►



Verifica di requisiti

I requisiti **funzionali** sono abbastanza semplici da verificare: basta effettuare il collaudo del sistema con gli utilizzatori e verificare se tutte le loro aspettative sono state soddisfatte.

Anche i requisiti di **dominio** possono essere verificati agevolmente: si collaudano le interazioni del sistema con il “resto del mondo”, cioè l'integrazione con il software aziendale preesistente, con terze parti e si verifica il rispetto delle normative di settore.

I requisiti **non funzionali**, invece, spesso indicati in modo generico dall'utente, possono risultare non quantificabili e difficili da verificare: a differenza dei precedenti, per i quali è possibile dire in “modo binario” se sono rispettati o meno, per questi generalmente è possibile dare un valore quantitativo del grado di soddisfacimento del requisito.

In tabella sono riportati, a titolo di esempio, degli indicatori quantificabili per alcune proprietà:

Proprietà	Misura
Velocità	Numero di transazioni per secondo Tempo di risposta di un evento Tempo di refresh dello schermo
Dimensione	Occupazione disco fisso programma Occupazione disco fisso per i dati Occupazione memoria RAM

Proprietà	Misura
Semplicità d'uso	Tempo richiesto per la formazione Dimensione della documentazione Numero di pagine di help on line
Affidabilità	Frequenza delle failure Gravità di una failure Accuratezza degli output Capacità di recupero dalle failure Predicibilità del programma
Robustezza	Tempo di riavvio dopo un guasto Percentuale di eventi che possono causare un crash del sistema Probabilità di danneggiamento dei dati
Portatilità	Percentuale di istruzioni dipendenti dai dispositivi HW Percentuale di istruzioni dipendenti dal SO



Prova adesso!

Data la seguente situazione, si richiede di individuare ogni tipo di requisito e, per i requisiti non funzionali, di predisporre una tabella per la valutazione quantitativa:

Si deve realizzare un sistema per archiviare i dati di una biblioteca, in particolare quelli relativi ai libri, ai giornali, alle riviste, ai video, ai nastri audio e ai CD-ROM.

Il sistema dovrà permettere agli utenti di fare delle ricerche per titolo, autore, genere o codice ISBN grazie a una interfaccia realizzata per un browser Internet.

L'accesso al sistema deve avvenire mediante autenticazione con smart-card e ogni prestito può durare al massimo 30 giorni per i libri, 7 giorni per i rimanenti articoli disponibili: un utente può avere contemporaneamente in consegna fino a 10 articoli.

Il sistema dovrà gestire almeno 20 transazioni per secondo e deve essere consultabile anche mediante smartphone.

Verifichiamo le conoscenze



1. Risposta multipla

- 1 Nella fase di specifica dei requisiti possiamo riconoscere le seguenti attività (indica quella errata):**
 - a) analisi del problema
 - b) definizione delle funzionalità
 - c) definizione del preventivo
 - d) redazione di un documento SRS
 - e) convalida delle specifiche

- 2 Quale tra le seguenti figure non si può considerare uno stakeholder?**

a) committente	e) programmatore
b) end-users	f) sistemisti
c) managers	g) nessuno di quelli elencati
d) analista	

- 3 Secondo il livello di dettaglio i requisiti software possono essere classificati in:**

a) requisiti funzionali	d) requisiti non funzionali
b) requisiti utente	e) requisiti di dominio
c) requisiti di sistema	

- 4 Secondo il tipo di requisito che rappresentano, i requisiti software possono essere classificati in:**

a) requisiti funzionali	d) requisiti non funzionali
b) requisiti utente	e) requisiti di dominio
c) requisiti di sistema	

- 5 Quali tra i seguenti requisiti non appartiene al primo livello della classificazione di Sommerville?**

a) di prodotto	c) esterni
b) di sistema	d) organizzativi

- 6 Una classificazione è chiamata modello FURPS, ottenuto come acronimo di (indica quello errato):**

a) Usability	d) Portability
b) Functionality	e) Supportability
c) Reliability	



2. Vero o falso

- 1 La specifica dei requisiti risponde alla domanda "che cosa deve fare il sistema?".** V F
- 2 La validazione dei requisiti risponde alla domanda "come lo deve risolvere il sistema?".** V F
- 3 La definizione dei requisiti rappresenta anche l'analisi del dominio del sistema.** V F
- 4 Le diverse persone coinvolte in tale processo sono chiamate gli stakeholder.** V F
- 5 I requisiti devono essere stabili per tutta la durata dello sviluppo.** V F
- 6 Per i requisiti non-funzionali ricordiamo la classificazione di Sommerville.** V F
- 7 Un esempio di requisiti di sistema è la richiesta di login (user-id e password).** V F
- 8 La verifica dei requisiti non funzionali può essere espressa mediante un valore quantitativo.** V F

Verifichiamo le competenze

3. Esercizi

Date le seguenti situazioni, si richiede di individuare ogni tipo di requisito e per i requisiti non funzionali di predisporre una tabella per la valutazione quantitativa

- 1 Si deve realizzare un software per la gestione di un telefono cellulare, in particolare per poter inserire un testo in due circostanze:
 1) composizione di SMS;
 2) immissione nuovo nome/numero in rubrica.

Esistono delle regole generali che disciplinano l'immissione del testo, per esempio:

- come si cambia fra maiuscole e minuscole;
- come si attiva/disattiva la modalità T9.

Gli SMS devono essere raggruppati per destinatario (sia quelli spediti che quelli ricevuti) e ricercati anche in base al giorno di trasmissione/ricezione.

La rubrica deve permettere l'immissione di più numeri per lo stesso nominativo, indicando per ciascuno di essi il gestore telefonico e il numero di minuti di conversazione effettuata.

L'accesso alla rubrica deve essere protetto da password.

- 2 Si deve realizzare un sistema per la gestione di un ufficio postale: descrivilo in termini di un insieme composto da due moduli funzionali (Posta e Banca) e per ciascuno di essi indica:

- i servizi che offre agli utenti;
- la modalità con cui gli utenti possono usufruirne.

Dopo aver individuato le possibili interazioni tra i due moduli, considera la possibilità di utilizzo di una parte di tali servizi tramite web: individua quali funzionalità possono essere comuni e quali limitazioni necessariamente devono essere definite.

- 3 Si deve realizzare un sistema per un ristorante per gestire i clienti, i tavoli (con il relativo numero di posti), le prenotazioni (effettuate dai clienti per un certo giorno e ora, e per un certo numero di persone). Alle prenotazioni vengono assegnati uno o più tavoli (divisi in fumatori/non fumatori), i camerieri (che servono i clienti al tavolo) e il conto (composto dalle singole portate ordinate).

Dei clienti interessano il nome e il numero di telefono, mentre dei camerieri interessano, nome e anni di servizio. Ogni portata ha un suo costo unitario previsto dal listino e al cliente viene presentato il conto dove vengono indicate le singole portate col loro nome, il prezzo unitario e la quantità ordinata che permette di calcolare il totale per portata e il conto totale da pagare.

- 4 Si deve realizzare un sistema software per una azienda costituita da diversi dipartimenti dove sono addetti un certo numero di impiegati con mansioni differenti: ogni impiegato è in forza solo a uno specifico dipartimento, e di lui sono memorizzati i dati anagrafici e gli stipendi percepiti.

Ogni dipartimento ha un suo nome, un direttore, un numero di telefono, la data di afferenza di ognuno degli impiegati che vi lavorano per poter calcolare i costi mensili, in termini di risorse umane.

Gli impiegati partecipano inoltre a vari progetti aziendali, dei quali viene indicato il nome e il budget: in questo caso lo stipendio non è a carico del dipartimento ma del progetto.

- 5 Si vuole progettare un sistema per la gestione di un campionato di calcio. Il sistema deve consentire la creazione del calendario delle partite e la designazione degli arbitri, con codice fiscale, nome, cognome e città di nascita. Il calendario è composto da un certo numero X di gironi, ognuno dei quali è composto da Y giornate, a loro volta composte da partite tra K squadre. Delle partite interessano il numero progressivo in schedina, la data, l'arbitro designato, le squadre (casa e ospite) e il risultato finale (che l'arbitro avrà il compito di memorizzare nel sistema, a fine gara, tramite un SMS).

Ogni squadra ha un proprio dirigente e un allenatore che decide quali giocatori convocare per le varie partite: le squadre sono individuate da un nome, dalla città e regione di provenienza. Il sistema deve anche gestire la classifica generale e dei marcatori, consultabile da Internet sia con un PC sia mediante uno smartphone.

Raccolta e analisi dei requisiti

In questa lezione impareremo...

- la fase di esplorazione
- le tecniche di esplorazione

■ Premessa

Per poter effettuare la stesura del documento **SRS** di Specifica dei Requisiti Software è necessario effettuare la raccolta dei requisiti (◀ **Products requirements** ▶) e la loro analisi.

Abbiamo visto che queste due fasi rientrano nella **ingegneria dei requisiti** (**requirements engineering**), che possiamo ricordare come una sequenza di quattro attività:

- **raccolta dei requisiti;**
- **analisi dei requisiti;**
- **stesura della documentazione** dei requisiti **SRS**;
- **verifica e approvazione dei requisiti.**

La **raccolta dei requisiti** è considerata l'attività più difficile, perché richiede la collaborazione tra più gruppi di partecipanti con differenti background.

◀ **Product requirements** "The most difficult part of building a software system is to decide, precisely, what must be built. No other part of the work can undermine so badly the resulting software if not done correctly. No other part is so difficult to fix later." (Fred Brook) ▶



◀ **Fred Brook** Frederick Phillips Brooks Jr., (nato il 19 aprile 1931) è un ingegnere informatico, conosciuto per lo sviluppo del sistema operativo **OS/360** della famiglia di computer **IBM System/360**: ne descrive il processo nel libro **The Mythical Man-Month**. Brooks attualmente si occupa di ingegneria del software e ha ricevuto numerosi riconoscimenti, tra cui la **National Medal of Technology** nel 1985 e il **Premio Turing** nel 1999. ▶



In questa lezione ci occuperemo dei primi due punti, cioè delle **raccolta e dall'analisi** dei **requisiti**, che prende anche il nome di **fase di esplorazione**.

■ Tipi di raccolta dei requisiti

A seconda del tipo di progetto la **raccolta di requisiti** degli **attori** potrebbe essere integrata da altre analisi; sostanzialmente abbiamo tre tipi di realizzazione:

- **greenfield engineering**: questa è la situazione in cui lo sviluppo parte da zero e nella azienda non esiste un precedente sistema da sostituire o integrare: la “fonte” dei **requisiti** è l’azienda committente, nella figura del cliente e degli **stakeholder** (attori); lo sviluppatore, però, deve anche valutare e ricercare la disponibilità sul mercato di un prodotto in grado di soddisfare le esigenze individuate da confrontare e/o proporre in alternativa allo sviluppo ex-novo;
- **re-engineering**: in questa situazione esiste già nella azienda un sistema che deve essere riprogettato perché tecnologicamente obsoleto rispetto alle nuove tecnologie o per estendere le funzionalità del sistema: l'**analisi del sistema** esistente è alla base del nuovo progetto perché si possono analizzare i suoi pregi e difetti, possono essere evidenziate le funzionalità che devono “migrare” nel nuovo sistema e possono essere valutate, per migliorare quelle che non soddisfacevano gli utenti, e integrate con le nuove esigenze e con le nuove possibilità offerte dalla tecnologia;
- **interface engineering**: è questa la situazione dove si è nella impossibilità di sostituire completamente il software esistente ma si deve adeguare almeno l’interfaccia utente con i nuovi ambienti operativi: le funzionalità non sono “toccate” e quindi non è necessario raccogliere nuovi requisiti perché rimane inalterato nello “strato inferiore” il ► **sistema legacy** ► con i suoi servizi che vengono interfacciati con il nuovo ambiente, **riprogettando** semplicemente le **interfacce**.



► **Sistema legacy** Il termine inglese **legacy** è utilizzato per indicare qualcosa di valore ottenuto attraverso un’eredità. Associato a un sistema informativo, pertanto, il termine **legacy** ne evidenzia le caratteristiche di “valore aziendale” e di “provenienza dal passato”. Nell’ambito dei sistemi informativi il termine **legacy** non sta a indicare qualcosa di vecchio ma, al contrario, uno strumento su cui l’azienda fa affidamento anche per il futuro: la convivenza fra questi sistemi e quelli realizzati con nuovi paradigmi e tecnologie è un dato imprescindibile. ►

ESEMPIO

Situazione

Il Comune di XYZ intende automatizzare la gestione delle informazioni relative alle contravvenzioni elevate sul suo territorio. In particolare, intende dotare ogni vigile di un dispositivo palmare che gli consenta di comunicare al sistema informatico il veicolo a cui è stata comminata la contravvenzione, il luogo in cui è stata elevata e la natura dell’infrazione. Il sistema informatico provvederà a notificare, tramite posta ordinaria, la contravvenzione al cittadino interessato.

Il Comune bandisce una gara per la realizzazione e manutenzione del sistema, che viene vinta dalla Ditta ABC.

Quali sono gli attori coinvolti in questa applicazione software?

- **Committente**: Comune di XYZ

- **Esperto del domino:** funzionario del Comune, o altro professionista designato, esperto del Codice della Strada
 - **Utenti finali:** vigili
 - **Progettista**
 - **Analista**
 - **Programmatore**
 - **Manutentore**
- } // personale della ditta ABC

■ La fase di esplorazione

La raccolta dei requisiti è anche detta **fase di esplorazione** in quanto il problema da risolvere deve essere proprio esplorato, cioè “sviscerato” in ogni sua minima componente, analizzato e affrontato in ogni suo minimo aspetto per individuare tutti i bisogni e definirne le priorità.

La letteratura anglosassone attribuisce a questa fase i termini *elicitation* o *discovery* cioè *elicitazione* e *scoperta* poiché spesso i requisiti vengono “scoperti” in quanto sono difficili da essere individuati.

Per poter produrre il **documento dei requisiti** è necessario raccogliere il maggior numero possibile d’informazioni sugli obiettivi e sulle necessità riguardo al sistema da costruire, e quindi procedere con l’esplorazione del sistema.

- Il primo passo della fase di esplorazione è quello di **esaminare le richieste del committente**, cioè colui che ha richiesto lo sviluppo del progetto, che ha approvato e sottoscritto il preventivo ed è di fatto il principale **referente** (cliente).
- Il secondo passo della fase di esplorazione consiste nella definizione degli **stakeholder (attori)**, cioè di tutti coloro che, in un modo o nell’altro, hanno qualche interesse nel prodotto, o la cui attività sarà influenzata, direttamente o **indirettamente**, da esso: si procede con il loro coinvolgimento (**engagement**) al progetto.

Lo stakeholder engagement

Engagement è un sostantivo che significa coinvolgimento, ma allo stesso tempo richiama al concetto del “dedicarsi, occuparsi”, nel senso che gli interlocutori coinvolti nello sviluppo partecipano attivamente al processo.

Gli **ingegneri del software** devono avviare un processo di dialogo con gli attori che inneschi una comunicazione interattiva in modo da confrontare reciprocamente le diverse aspettative di ciascuno di loro per impostare o rivedere politiche e strategie dell’impresa, integrando eventualmente quello che emerge da questa attività: spesso il punto di vista dei dipendenti è diverso da quello della azienda e frequentemente solo chi operativamente esegue determinate attività è a conoscenza di particolari criticità.

È importante dare a questa fase il giusto peso, senza sottovalutarne l’importanza: **coinvolgere** vuol dire comunicare interattivamente, confrontarsi, dialogare, saper ascoltare, prendere degli impegni verso gli attori e non significa fare dei sondaggi, proporre soluzioni, comunicare delle scelte fatte da altri. Gli **attori** devono sentirsi “importanti” per la realizzazione del progetto (come, di fatto, lo sono) e devono far parte a tutti gli effetti del gruppo di lavoro.

È però anche importante individuare criteri di selezione che garantiscono la rappresentatività e inclusività degli **stakeholder** affinché il processo porti a risultati utili alla definizione dei requisiti e quindi al successivo sviluppo del progetto.

È necessario conoscere tutti i “**punti di vista**” (**viewpoint**), cioè analizzare il sistema dalle prospettive di tutti i possibili suoi utilizzatori e/o componenti che possono interagire con esso per individuarne tutti i **requisiti** e comprendere come questi possono essere realizzati.

La **raccolta dei requisiti** dagli **stakeholder** viene anche chiamata **requirements elicitation**.

Tecniche di esplorazione

La tabella seguente riporta le tecniche principali che possono essere utilizzate nella fase di esplorazione per la raccolta dei requisiti.

Strumenti	Obiettivi	Vantaggi	Svantaggi
Interviste individuali	Esplorare determinati aspetti del problema e determinati punti di vista.	L'intervistatore può controllare il corso dell'intervista, orientandola verso quei temi sui quali l'intervistato è in grado di fornire i contributi più utili.	Richiedono molto tempo. Gli intervistati potrebbero evitare di esprimersi con franchezza su alcuni aspetti delicati.
Focus group	Mettere a fuoco un determinato argomento, sul quale possono esserci diversi punti di vista.	Fanno emergere le aree di consenso e di conflitto. Possono far emergere soluzioni condivise dal gruppo.	La loro conduzione richiede esperienza. Possono emergere figure dominanti che monopolizzano la discussione.
Osservazioni sul campo	Comprendere il contesto delle attività dell'utente.	Permettono di ottenere una consapevolezza sull'uso reale del prodotto che le altre tecniche non danno.	Possono essere difficili da effettuare e richiedere molto tempo e risorse.
Suggerimenti spontanei degli utenti	Individuare specifiche necessità di miglioramento di un prodotto.	Hanno bassi costi di raccolta. Possono essere molto specifici.	Hanno normalmente carattere episodico.
Questionari	Rispondere a domande specifiche.	Si possono raggiungere molte persone con poco sforzo.	Vanno progettati con grande accuratezza, in caso contrario le risposte potrebbero risultare poco informative. Il tasso di risposta può essere basso.
Analisi della concorrenza e delle best practices	Individuare le soluzioni migliori adottate nel settore di interesse.	Evitare di “reinventare la ruota” e ottenere vantaggio competitivo.	L'analisi di solito è costosa (tempo e risorse).
Scenari e casi d'uso	Descrivere ogni singola operazione che il sistema deve effettuare.	Sono poi utilizzati in fase di collaudo per verificare le funzionalità del sistema.	Gli intervistati spesso non sono in grado di descrivere le criticità.

Interviste individuali

La migliore fonte per reperire i requisiti è quella delle interviste individuali.

L'indagine inizia con l'intervista al cliente/committente che è il maggior referente del progetto (anche perché “è lui che paga”) e quindi ha chiari gli obiettivi che devono essere perseguiti e i tempi di realizzazione del sistema.

Sarà inoltre il cliente stesso a indicare al team di analisi gli **stakeholder** più significativi per ogni categoria di personale, con i quali procedere con le interviste: sarà sempre lui che, alla fine della stesura del documento, verrà coinvolto nella revisione e validazione.

Più persone vengono intervistate e più possibilità ci sono di avere nuove indicazioni utili ma anche di ottenere suggerimenti contraddittori, dato che ciascuno osserva un requisito solo dal proprio punto di vista e lo esprime in termini diversi: è essenziale valutare caso per caso le persone intervistare e selezionarle in modo opportuno, dopo averle suddivise in gruppi omogenei di utenti.

ESEMPIO

Modalità di individuazione degli attori

L'individuazione degli attori può essere fatta seguendo il seguente campione di domande:

- chi o cosa usa il sistema? Che compito svolge?
- chi ottiene informazioni dal sistema e a chi ne fornisce?
- quali gruppi di utenti eseguono le principali funzioni del sistema?
- quali gruppi di utenti eseguono le funzioni secondarie, come la manutenzione e l'amministrazione?
- sono presenti funzioni o azioni che vengono eseguite a intervalli prestabiliti?
- sono presenti funzioni svolte da una sola persona?
- quali sono i dipendenti che da più anni sono nella organizzazione?
- chi sono gli attori esterni all'organizzazione che interagiscono con essa?
- con quale sistema hardware o software il sistema interagisce?

Bisogna anche tenere conto del tempo a disposizione per questa fase, quindi è necessario saper individuare quali sono le persone che sicuramente hanno informazioni importanti e quali, invece, possono essere consultate solo in fase di verifica del sistema utilizzando un semplice questionario.

Strutturazione delle interviste

È possibile effettuare interviste con diversi **livelli di strutturazione**:

- A** **interviste non strutturate**: sono dei dialoghi dove l'intervistatore pone un insieme di domande aperte prestabilite allo **stakeholder** e dalle risposte può prendere spunti per avviare una indagine approfondita su eventuali nuovi requisiti o aspetti diversi di operazioni già individuate.

ESEMPIO

Domande da effettuare agli stakeholder

Il colloquio può iniziare semplicemente con una sequenza di domande del tipo:

- quali sono le attività che svolgi regolarmente?
- descrivi il flusso normale delle tue attività.

- descrivi cosa può andar male nell'esecuzione del flusso normale.
- cosa non funziona nel sistema attuale?
- come intendi utilizzare il sistema?
- come credi che il sistema debba essere utilizzato?
- cosa ti aspetti quando il sistema parte?
- ...

B interviste strutturate: sono simili ai questionari (che tratteremo in seguito) ma sono somministrati verbalmente dall'intervistatore e consistono in un insieme di domande specifiche su di una particolare funzione o requisito che è già stato individuato ma richiede una indagine su tutti i possibili utilizzatori.

Proprio per la loro natura strutturata sono utilizzate per ricavarne dati statistici facilmente interpretabili.

C interviste semi-strutturate: sono una via di mezzo delle prime due, cioè sono interviste che contengono sia risposte aperte che domande specifiche a risposta chiusa.

Non è semplice condurre una buona intervista: un fattore importante è l'esperienza dell'intervistatore che deve avere ottime capacità di relazione e deve essere in grado di mettere l'intervistato a proprio agio, garantirne l'anonimato delle risposte, parlare con lo stesso linguaggio dell'intervistato, senza suggerire le soluzioni ma cercando di individuare i requisiti. La maggior difficoltà è quella di individuare i requisiti impliciti che le persone considerano ovvi, e quindi non evidenziano nel colloquio poiché spesso rivelano come le operazioni dovrebbero essere svolte "in teoria" e non come sono svolte "in pratica".

Questionari

È il metodo più semplice per ottenere con bassi costi delle informazioni in forma strutturata facilmente elaborabili in modo automatico per ottenere delle statistiche.

Sono composti da un insieme di domande alle quali gli utenti rispondono scegliendo una risposta tra le cinque proposte: *completamente d'accordo, d'accordo, incerto, in disaccordo, in completo disaccordo* (scala di **Likert**). A ciascuna risposta è associato un numero compreso fra 1 e 5 e con questi valori si potrà calcolare la media delle risposte a ciascun gruppo di affermazioni correlate a uno stesso argomento.

Possono essere utilizzati solo in **fase di consuntivazione** per ottenere un giudizio sul sistema appena realizzato, prima della sua messa in funzione, anche perché sono facilmente realizzabili e somministrabili a tutti gli utenti (esistono software che realizzano questionari da sottoporre online producendo direttamente i risultati). L'attendibilità delle risposte, però, generalmente è bassa.

AREA digitale

 Esempio di questionario per lo sviluppo di una piattaforma di e-commerce

Focus group

I **focus group** sono interviste di gruppo che vengono gestite sul modello dei **brainstorming** partendo con l'analisi di un singolo argomento e cercando di far esprimere tutti i partecipanti in modo da ricercare gli elementi di condivisione e di contrasto.

◀ Brainstorming Brainstorming is a group or individual creativity technique by which efforts are made to find a conclusion for a specific problem by gathering a list of ideas spontaneously contributed by its member(s). ▶



Il mediatore deve essere in grado di gestire il gruppo, soprattutto nei casi di divergenza di opinioni, facendo in modo che la discussione non degeneri in lite e annotando ogni osservazione utile per la definizione e/o puntualizzazione dei requisiti.

Osservazioni sul campo

L'osservazione sul campo consiste nell'affiancare l'utente nelle sue operazioni quotidiane per visionare direttamente quali operazioni svolge nella attività quotidiana: spesso l'attore fatica a descrivere il proprio lavoro, soprattutto se questo utilizza strumenti software dei quali non ha particolari conoscenze tecniche ma solo operative.

Spesso l'utente non è in grado di evidenziare le criticità delle sue funzioni e quindi l'analisi congiunta "sul campo" con il responsabile dello sviluppo permette di scoprire dettagli importanti per la definizione dei requisiti.

È una attività molto onerosa, che richiede molto tempo e una conoscenza preventiva delle singole attività per individuare quando è più significativo effettuare l'osservazione; inoltre vengono osservate le funzioni "già presenti" e quindi difficilmente permette di individuare nuove funzioni che potrebbero migliorare il lavoro.

Suggerimenti spontanei degli utenti

Gli utenti suggeriscono come il prodotto deve essere realizzato anche nella fase delle **interviste**, ma inoltre possono ulteriormente contribuire fornendo indicazioni su appositi forum che vengono aperti sul web contestualmente all'inizio della attività di sviluppo: gli utenti segnalano spontaneamente miglioramenti desiderabili e condividono o meno i suggerimenti proposti dagli altri utenti.

Analisi della concorrenza e delle best practice

Tra le attività svolte nella fase di esplorazione dei requisiti è di fondamentale importanza il confronto con prodotti concorrenti simili già presenti sul mercato per "attingere" dai ▶ **best practice** ▶ qualche indicazione preziosa nonché compararne i prezzi, i punti di forza e riconoscere i punti di debolezza.



◀ **Best practice** Il termine di best practice rientra nel più ampio processo di benchmarking: metodo di confronto finalizzato a identificare, comprendere e adattare pratiche particolarmente significative (riconosciute come best practice o high performance) messe in atto da altre organizzazioni, al fine di migliorare le prestazioni della propria attività. ▶

Questa operazione è particolarmente difficile da effettuarsi nelle situazioni di **sviluppo ad hoc** altamente specializzato dove è difficile, se non impossibile, trovare altre realizzazioni già implementate: in generale, però, parte del sistema da sviluppare lo si può trovare già realizzato e quindi da queste realizzazioni si possono trarre preziose indicazioni.

La ricerca di soluzioni già esistenti o simili deve essere fatta all'inizio del progetto, non appena sono definiti gli obiettivi essenziali; è importante anche sottoporre le soluzioni adottate da terzi agli **stakeholder** in modo che possano valutarle e commentarle per verificare la loro possibile integrazione nel nuovo sistema e la loro applicabilità nel contesto corrente.

Casi d'uso

Sono la descrizione degli **esempi d'uso del sistema**, cioè delle singole operazioni e attività che devono essere implementate nel sistema; per ciascun **caso d'uso** viene specificata la

normale sequenza di eventi necessaria per compiere quell'operazione e le eventuali sequenze di eventi alternative in caso di errore per le situazioni non previste o per i "casi limite di funzionamento".

Per la loro importanza, che non si limita alla fase di analisi dato che i casi d'uso sono una componente fondamentale dalla fase di verifica e collaudo del sistema, verranno trattati dettagliatamente nella prossima lezione, a loro riservata.

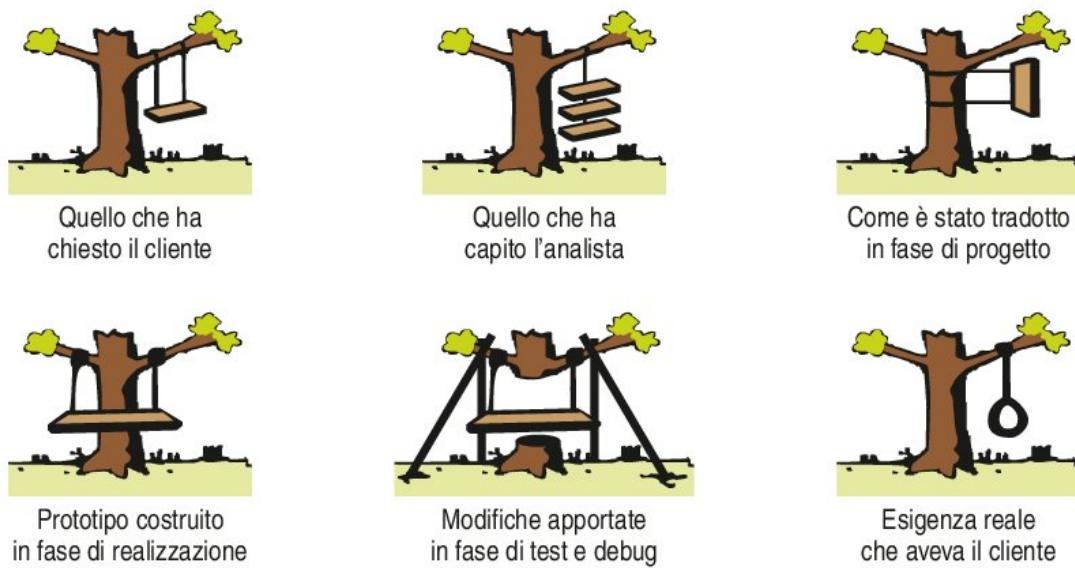
Riepilogo

Possiamo schematizzare con la seguente figura la **fase di esplorazione**, dove tutto il materiale raccolto è ancora allo stato di *appunti* e deve essere formalizzato nel documento **SRS**.



■ Problemi della fase di esplorazione

La metafora più comune per la gestione dei requisiti nei progetti software è quella conosciuta come la **best practice** metafora dell'altalena, che aiuta a comprendere perché le parole (influenzate dal gergo) non sono indicate per descrivere il problema.



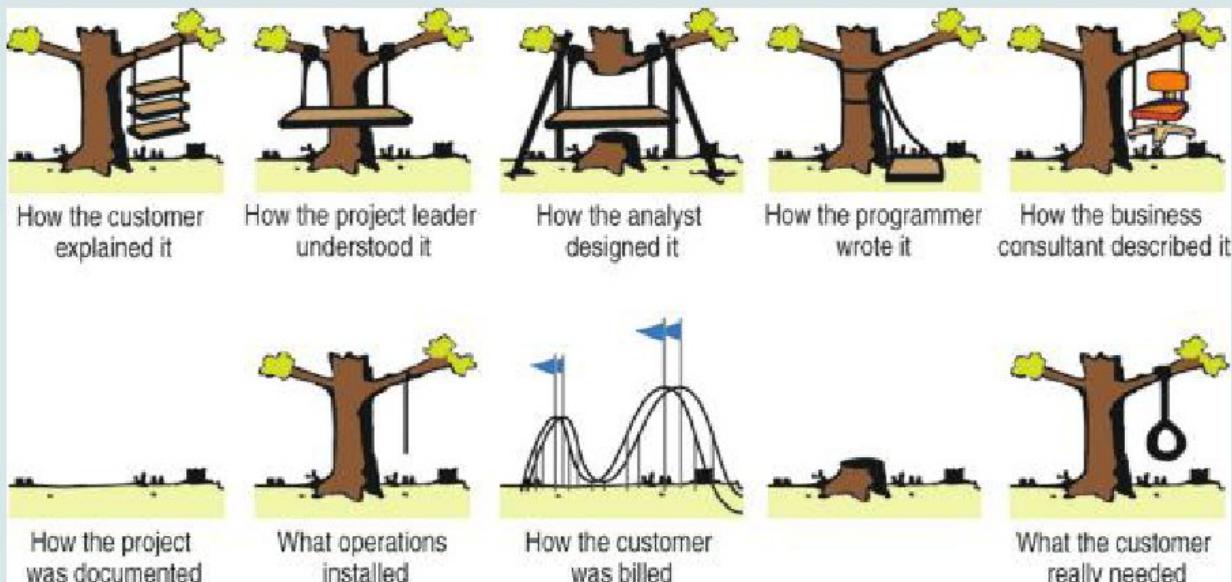
È chiaro come l'esigenza descritta dal cliente "con sue parole" venga interpretata diversamente da tutti gli interlocutori nelle diverse fasi del ciclo di vita dello **sviluppo del prodotto**.

to: è interessante infine osservare come lo stesso cliente *abbia una diversa percezione* di quello che realmente gli serve rispetto a quello di cui “dichiara di avere bisogno”.

La difficoltà di comprensione delle esigenze del cliente rende difficile la realizzazione del sistema.



◀ Riportiamo per completezza una nuova formulazione più completa della “metafora dell’altalena”. ▶



Un’altra metafora che ben si presta alla “acquisizione corretta dei requisiti” è il “telefono senza fili”, un gioco diffuso tra i bambini “prima che si diffondessero i cellulari” (il gioco forse si gioca ancora, ma probabilmente avrà cambiato nome).

Il gioco è semplice: i bambini si dispongono in linea oppure in cerchio; il primo sussurra una frase nell’orecchio del vicino, che a sua volta la ripete al vicino successivo, e così via fino all’ultimo bambino che, per concludere, deve dire la frase ad alta voce. Il divertimento deriva dal fatto che la frase riportata dall’ultimo giocatore è praticamente sempre molto diversa da quella di partenza, a causa del combinarsi e sommarsi di errori successivi di interpretazione (anche introdotti di proposito!).

La gestione dei requisiti è analoga: più lunga è la catena che collega chi ha l’esigenza, il requisito, a chi lo deve soddisfare, maggiore la probabilità di fraintendimenti e incomprensioni. Più persone lavorano in sequenza sui requisiti, più aumentano i costi, i tempi, i rischi, la litigiosità.

La **fase di esplorazione** è estremamente delicata e, come abbiamo visto, non sempre è di semplice realizzazione in quanto spesso presenta notevoli difficoltà, che possono essere suddividere in quattro tipologie:

► **problemi di ambito:** durante le interviste è difficile individuare il livello di dettaglio e di approfondimento che deve essere rispettato, rischiando da una parte di non entrare abba-

stanza nello specifico e tralasciare magari aspetti e dettagli che invece sono di importanza essenziale nell'impostazione del sistema, oppure di “sconfinare” in aree esterne a quelle specifiche di competenza dello **stakeholder** interrogato delle quali ha solo una conoscenza parziale e quindi potrebbe esprimere valutazioni dannose;

► **problemi di comprensione:** la difficoltà di comunicazione è stata già più volte sottolineata ed è dovuta dal fatto che gli **stakeholder** utilizzano un linguaggio tecnico differente da chi raccolgono i requisiti e tende a non dare il giusto peso e la corretta importanza alle attività che svolgono, trascurando gli aspetti che ritengono ovvi e limitandosi a descriverle superficialmente. Va sempre tenuto presente che gli **stakeholder** non sono esperti di informatica e spesso le loro richieste sono di difficile realizzazione oppure potrebbero avere costi elevati non contemplati dal budget del cliente;

► **problemi di conflitto:** il medesimo requisito può essere descritto in modo differente da **stakeholder** che appartengono a gruppi diversi e, al limite, le descrizioni potrebbero essere anche incompatibili tra di loro. I conflitti devono emergere in questa fase e devono essere affrontati in modo da trovare una “mediazione” che soddisfi tutti i contendenti;

► **problemi di volatilità:** i requisiti individuati non rimangono stabili per tutto il ciclo di vita del progetto, anzi, è anche possibile che si evolvano e mutino anche all'interno della singola fase di analisi. Le cause principali di questa dinamicità sono legate a **eventi esterni** come l'evoluzione tecnologica, i mercati, le leggi e gli obblighi fiscali, oppure a **eventi interni**, come il ricambio del management o la ristrutturazione dell'organizzazione del committente, acquisizioni o vendite societarie, nuove strategie di mercato, di produzione ecc.

È difficile stabilire la correttezza e la completezza dei requisiti, specialmente prima che il sistema venga realizzato e quindi collaudato: dato che la specifica dei requisiti serve come base per lo sviluppo, questi devono essere rivisti con grande attenzione sia dal committente che dal team di progettazione e validati prima che si prosegua con le successive fasi di realizzazione.

Verifichiamo le conoscenze

1. Risposta multipla

- 1 Quale tra le seguenti non è una attività della ingegneria dei requisiti?**
 - a) raccolta dei requisiti
 - b) analisi dei requisiti
 - c) stesura della documentazione dei requisiti SRS
 - d) collaudo dei requisiti
 - e) approvazione dei requisiti

- 2 Quale tra le seguenti non è una tipologia di progetto?**
 - a) greenfield engineering
 - b) re-engineering
 - c) reverce engineering
 - d) interface engineering

- 3 Quale tra le seguenti tecniche di esplorazione fa emergere aree di conflitto?**
 - a) Interviste individuali
 - b) Focus group
 - c) Osservazioni sul campo
 - d) Suggerimenti spontanei degli utenti
 - e) Questionari
 - f) Analisi della concorrenza e delle best practices
 - g) Scenari e casi d'uso

- 4 Quali tra le seguenti tecniche di esplorazione richiedono molto tempo?**
 - a) Interviste individuali
 - b) Focus group
 - c) Osservazioni sul campo
 - d) Suggerimenti spontanei degli utenti
 - e) Questionari
 - f) Analisi della concorrenza e delle best practices
 - g) Scenari e casi d'uso

- 5 Quale problemi non sono connessi alla fase di esplorazione?**
 - a) problemi di ambito
 - b) problemi di comprensione
 - c) problemi di sviluppo
 - d) problemi di conflitto
 - e) problemi di volatilità

2. Vero o falso

- 1 Se esiste già nella azienda un sistema che deve essere riprogettato si parla di re-engineering.**
- 2 Il termine legacy è utilizzato per indicare qualcosa di valore ottenuto attraverso un'eredità.**
- 3 Nell'ambito dei sistemi informativi il termine legacy sta a indicare qualcosa di vecchio.**
- 4 Con Stakeholder Engagement si intende la scelta degli attori.**
- 5 La raccolta di informazioni dagli stakeholder può essere fatta sotto forma di sondaggio.**
- 6 La migliore fonte per reperire i requisiti è quella delle interviste individuali.**
- 7 Agli stakeholder è necessario effettuare interviste strutturate.**
- 8 L'attendibilità delle risposte ai questionari è generalmente buona.**
- 9 I requisiti individuati non rimangono stabili per tutto il ciclo di vita del progetto.**
- 10 I requisiti individuati non possono variare nella fase di analisi.**

V	F
V	F
V	F
V	F
V	F
V	F
V	F
V	F
V	F
V	F

Attori, casi d'uso e scenari

In questa lezione impareremo...

- a individuare gli scenari d'uso
- a descrivere i casi d'uso in UML
- a documentare i casi d'uso

■ Introduzione

Abbiamo visto come i **requisiti funzionali** catturano il comportamento atteso del sistema in termini di servizi, compiti e funzioni: la metodologia più semplice che viene utilizzata per rappresentare i **requisiti funzionali** è quella che si basa sui **casi d'uso**, utilizzata soprattutto nello sviluppo **object-oriented**.

Durante la **fase di analisi** la tecnica dei **casi d'uso** viene utilizzata per catturare il comportamento esterno del sistema da sviluppare, senza dover specificare come tale comportamento viene realizzato, cioè considerando il sistema come una “scatola nera” (**black-box**).

Viene descritto in un diagramma il comportamento, offerto o richiesto, di una funzione desiderata dal sistema individuando chi (o che cosa) compie l'azione o interagisce col sistema (**attore**) e che cosa viene fatto (**caso d'uso**).

In sintesi, due domande ci aiutano in questa fase di progetto:

- chi interagisce con il sistema? => **attori**;
- cosa fanno? => **casi d'uso**.

Descriviamoli nel dettaglio, data la loro fondamentale importanza per il buon esito della **analisi dei requisiti**.

Attore (actor)

Con **attori** si indicano tutte le entità, interne od esterne, che interagiscono col sistema: queste devono essere in grado di prendere delle decisioni autonome fornendo informazioni in input e/o ricevendo informazione in output; non necessariamente sono “esseri umani” in

quanto anche altri sistemi possono interagire con il sistema che si sta progettando e, quindi, anch'essi sono degli **attori**.

Un **attore** può quindi essere singolo utente, una classe omogenea di utenti, un “ruolo” che gli utenti possono svolgere, un altro sottosistema, una parte del sistema non ancora sviluppata, un trigger temporale, ...



ATTORE

Un **attore** specifica un ruolo assunto da un utente o altra entità che interagisce con il sistema nell'ambito di un'unità di funzionamento (**caso d'uso**).

Gli **attori** eseguono **casi d'uso**: nella fase di analisi si cercano prima gli **attori** e per ciascuno di essi si descrivono i singoli **casi d'uso**.

Caso d'uso (use cases)

Il termine **use cases** viene introdotto da **Ivar Jacobson** per il metodo **Objectory** con questo “nuovo nome”, ma non è altro che la tecnica consolidata che si basa sullo studio degli **scenari** di operatività degli utilizzatori di un sistema.



USE CASES

Un **caso d'uso** è una sequenza di transazioni in un **sistema** il cui compito è di conseguire un risultato di valore misurabile per un singolo **attore** del sistema. (Jacobson 1992).

In altre parole i **casi d'uso** sono le funzionalità che il **sistema** mette a disposizione dei suoi utilizzatori, cioè i “modi” in cui il **sistema** può essere utilizzato, cosa ci si aspetta dall’operatività del **sistema** (“**what?**”), i compiti che svolge, i servizi che offre, ecc.

Un **caso d'uso** è avviato da un **attore** con un particolare obiettivo e si conclude con successo quando l’obiettivo è raggiunto.

Vedremo come viene descritta nel **caso d'uso** la sequenza di interazioni tra **sistema** e **attori** necessarie a realizzare il servizio richiesto anche indicando le possibili sequenze e/o alternative con cui può essere raggiunto l’obiettivo.

Una **specifica situazione** nell'esecuzione del **caso d'uso** prende il nome di **scenario** e la lista completa di tutti i **casi d'uso** e di tutti gli **scenari** specifica tutti i differenti modi di usare il sistema.

ESEMPIO

Descriviamo una generica la situazione di un negozio online: a titolo d'esempio elenchiamo alcuni possibili casi d'uso:

- ▷ registrare un nuovo utente
- ▷ modificare i dati di un utente
- ▷ acquistare un prodotto
- ▷ ricercare un prodotto nel catalogo
- ▷ inserire un nuovo prodotto in catalogo
- ▷ modificare i dati di un prodotto

Ciascuno di questi **casi d'uso** corrisponde a una precisa funzionalità che un “soggetto” deve eseguire con il nuovo sistema: possiamo individuare in questo esempio quattro diversi **attori** che devono interagire con il sistema software, cioè l'*Utente Visitatore*, l'*Utente registrato*, l'*Amministratore del sistema* e il *Sistema Bancario*, e ciascuno di questi attori eseguirà specifici **casi d'uso**, anche comuni a più attori.

In particolare nel nostro esempio avremo:

- A** l'Utente Visitatore
 - ricercare un prodotto nel catalogo
 - registrarsi come nuovo cliente
- B** l'Utente registrato
 - modificare i dati di un utente
 - ricercare un prodotto nel catalogo
 - acquistare un prodotto
- C** l'Amministratore del sistema
 - inserire un nuovo prodotto in catalogo
 - modificare i dati di un prodotto
- D** Sistema Bancario
 - regolare l'acquisto di un prodotto



Prova adesso!

- Casi d'uso
- Attori

Dati i seguenti **casi d'uso** per un sistema di vendita online di biglietti ferroviari:

- | | |
|---|--|
| <ul style="list-style-type: none"> ► consultare gli orari ► comperare un biglietto ► viaggiare | <ul style="list-style-type: none"> ► prenotare uno o più posti ► scrivere un reclamo ► aggiornare gli orari |
|---|--|

individua gli attori e per ciascuno di essi dettagli gli specifici **casi d'uso**.

Se un caso d'uso coinvolge più attori, quello che persegue lo scopo che il caso d'uso deve soddisfare sarà considerato l'**attore principale**: nel nostro esempio nel caso acquistare un prodotto l'Utente Registrato è l'attore principale mentre il Sistema Bancario è l'**attore secondario**.

Scenario



SCENARIO

Uno **scenario** rappresenta una **particolare interazione** tra uno o più **attori** e il sistema, cioè uno **scenario** è un'istanza di un **caso d'uso**.

◀ **Scenario** “A narrative description of what people do and experience as they try to make use of computer systems and applications” [M. Carroll, Scenario-based Design, Wiley, 1995] ▶



Uno **scenario d'uso** è quindi la descrizione che uno specifico **attore** fornisce, in linguaggio comune, del suo utilizzo del sistema, o meglio, della “sequenza di operazioni” che egli effettua per una possibile situazione di utilizzo del sistema.

Per descrivere completamente un **caso d'uso** è opportuno individuare i diversi **scenari**, cioè “i diversi percorsi possibili”: non potendo “naturalmente” descrivere tutti gli scenari **possibili** è necessario almeno individuare i **casi più significativi** così da poter descrivere al meglio almeno le diverse interazioni che ci possono essere fra più **casi d'uso**.

Il diagramma dei **casi d'uso (use case diagrams)** è il primo tipo di diagramma ad essere creato in un processo o ciclo di sviluppo, nell’ambito dell’analisi dei requisiti dato che serve a modellare i **requisiti funzionali** di un sistema.

Osserviamo infine che oltre che per descrivere i **requisiti** d’utente nelle fasi iniziali dell’analisi del sistema, gli **scenari** vengono anche utilizzati al termine della realizzazione per la validazione dell’architettura durante la fase di collaudo del sistema, per verificarne il “soddisfacimento” dei **requisiti**.

ESEMPIO **Un semplice scenario: negozio online**

Vediamo per esempio uno **scenario** riferito a un **negozi online** su Internet.

Scenario: Acquisto di un Prodotto

Il cliente naviga nel catalogo e raccoglie gli articoli desiderati in un carrello della spesa “virtuale”. Quando il cliente desidera pagare, descrive la modalità di spedizione e fornisce la necessaria informazione riguardante la propria carta di credito prima di confermare l’acquisto. Il sistema controlla se la carta di credito è valida e conferma l’acquisto sia immediatamente che con un successivo messaggio di posta elettronica.

Questo scenario descrive come potrebbero andare le cose, ma se per esempio la carta di credito fosse scaduta la parte terminale dello scenario sarebbe diversa:

...

Il sistema controlla se la carta di credito è valida ed essendo scaduta invia un messaggio di posta elettronica segnalando il problema al cliente.

Potrebbe esserci un altro problema sempre legato al pagamento, per esempio se il credito non fosse sufficiente, modificando nuovamente la parte terminale dello scenario:

...

Il sistema controlla se la carta di credito è valida ed essendo il credito insufficiente invia un messaggio di posta elettronica segnalando il problema al cliente.

ESEMPIO **Negozi online (seguito)**

Considerando l’esempio precedente del **negozi online**, il **caso d'uso** **Acquisto di un Prodotto** ha i seguenti tre **scenari** corrispondenti a:

- ▷ successo della transazione;
- ▷ invalidità della carta di credito;
- ▷ insufficienza del credito sulla carta.

Uno scenario può anche essere descritto mediante un "dialogo" che rappresenta l'interazione tra gli utilizzatori e il sistema:

- il **cliente** richiede l'elenco dei prodotti
- il **sistema** propone i prodotti disponibili
- il **cliente** sceglie i prodotti che desidera
- il **sistema** fornisce il costo totale dei prodotti selezionati
- il **cliente** conferma l'ordine
- il **sistema** comunica l'accettazione dell'ordine

È importante focalizzare l'attenzione sulle funzionalità generate dalla interazione e non dalle attività oppure da come queste vengono svolte all'interno del sistema.

■ Tipi di scenari

Esistono diversi tipi di scenari:

- **As-is scenario**: serve per descrivere la situazione attuale esistente nell'organizzazione e solitamente viene utilizzato durante il **re-engineering**: agli utenti viene detto di descrivere il sistema che stanno utilizzando;
- **Visionary scenario**: viene usato principalmente dal committente tipicamente nelle situazioni di **greenfield engineering** o nel **re-engineering** per descrivere il sistema futuro o desiderato;
- **Evaluation scenario**: descrive i task che gli utenti dovrebbero svolgere nel sistema per i quali sarà poi valutato (esempio: emettere un biglietto, prenotare un posto, validare un pagamento ecc.);
- **Training scenario**: è un tutorial che indica passo per passo come un neofita può imparare la corretta interazione con il sistema (esempio: i passi necessari per emettere un biglietto: selezionare lo spettacolo, scegliere l'orario, cercare il posto, cercare la tariffa ecc.).

La **definizione degli scenari** può essere il primo passo nella **definizione dei requisiti** che avvicina i progettisti e gli sviluppatori agli **stakeholder**: la descrizione delle situazioni d'uso concrete è la base su cui lavorare per sviluppare il nuovo prodotto che dovrà soddisfare le esigenze reali, tipiche delle attività di ogni giorno.

La descrizione dettagliata degli scenari con esempi pratici aiuta ogni componente del gruppo di lavoro a comprendere senza pericolo di ambiguità le funzionalità desiderate.

Gli **scenari** possono essere utilizzati in diverse attività del ciclo di vita del software: come vedremo i **casi d'uso** saranno fondamentali anche nella fase di verifica e collaudo in quanto il sistema deve implementare tutti i casi d'uso descritti negli scenari.

Individuazione degli scenari

L'**individuazione degli scenari** è sicuramente più difficile da effettuarsi nel caso che il sistema sia da realizzarsi ex novo, cioè se non esiste nella organizzazione: nel caso di **greenfield engineering** non bisogna aspettarsi che il cliente proponga e descriva gli scenari ma questa operazione deve essere fatta in modo coordinato tra team di sviluppo e cliente usando tecniche evolutive e incrementali.

Si può partire con semplici domande del tipo:

- quali sono i compiti primari che ciascun attore vuole che esegua il sistema?
- quali dati saranno creati, memorizzati, modificati, cancellati, o aggiunti dall'utente nel sistema?
- di quali cambiamenti esterni l'attore deve informare il sistema? Quanto spesso? Quando?
- di quali cambiamenti o eventi deve essere informato l'attore dal sistema? Entro quanto?

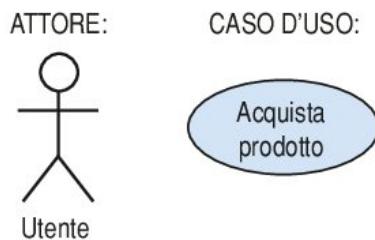
Quindi si riesamina assieme al cliente e agli **stakeholder** quanto è emerso da una “prima analisi” e si procede successivamente mediante affinamenti successivi: la continua interazione con il cliente aiuta gli analisti che man mano descrivono gli scenari acquisiscono una maggiore conoscenza dei requisiti.

L'**individuazione degli scenari** è più agevole se il sistema già esiste, cioè nelle situazioni di **interface engineering** o **reengineering**: in questo caso si parte da situazioni esistenti, ed è però necessario insistere sull'osservazione dettagliata delle funzioni per non ignorare particolari che gli **stakeholder** considerano scontati perché consueti e spesso eseguiti in automatico ma che sono ignoti agli sviluppatori.

■ Descrizione dei casi d'uso

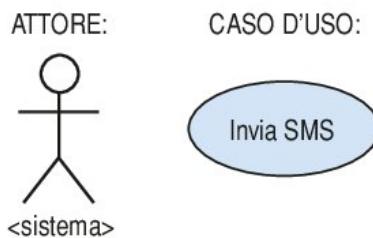
Utilizziamo i diagrammi **UML** per la rappresentazione del caso d'uso: con questi possiamo descrivere i singoli scenari e anche indicare le relazioni esistenti tra essi.

Gli **attori** sono rappresentati con omini stilizzati e il nome dell'attore viene indicato sotto l'omino, i **casi d'uso** con ellissi e il loro nome è indicato all'interno dell'ellisse, come indicato nella seguente figura:



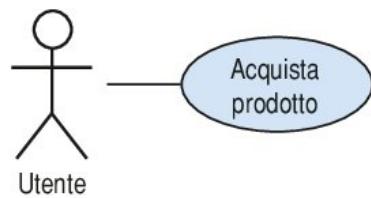
Anche se vengono indicati graficamente con degli omini sappiamo che gli **attori** coinvolti in un caso d'uso possono anche non essere umani: per esempio, se un sistema si interfaccia con un altro mediante una linea telefonica per comunicare o ricevere dati oppure per richiedere un servizio non necessariamente è necessaria la presenza umana.

In questo caso nel diagramma al posto del nome dell'attore viene genericamente indicato **<sistema>** per indicare che l'attore non è umano.



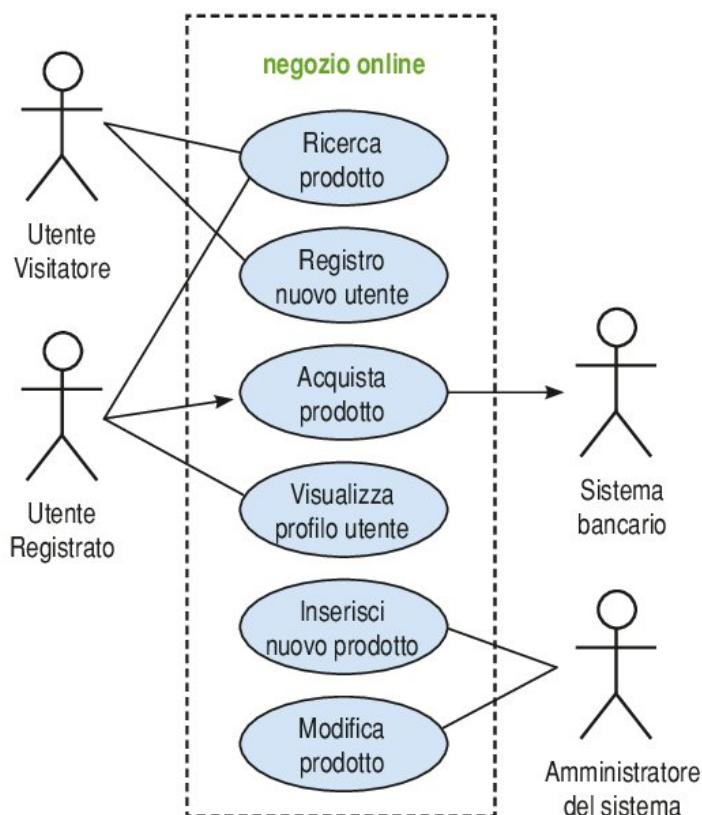
Tra un attore e un caso d'uso viene indicata l'associazione mediante un segmento che li congiunge: in questo modo viene indicata una (o più) delle tre possibili situazioni:

- ▶ l'attore esegue il caso d'uso;
- ▶ l'attore fornisce informazioni al caso d'uso;
- ▶ l'attore riceve informazioni dal caso d'uso.



In questo modo si è ottenuto un componente di quello che prende il nome di **Use Case Diagram (UCD)**: l'**Use Case Diagram** mostra come i casi d'uso sono collegati con gli attori e tra di loro.

Completiamo l'**Use Case Diagram** del nostro esempio negozio online con tutti i casi d'uso che abbiamo prima individuato, ottenendo:



La direzione della freccia non specifica la direzione dei flussi dei dati e viene messa solamente nei casi in cui possono esserci ambiguità in presenza di più **attori** per individuare l'**attore principale**.



DIAGRAMMA DI CONTESTO

Un diagramma che rappresenta tutti i casi d'uso di un sistema si chiama **diagramma di contesto** del sistema, perché indica i "confini" dello stesso e tutti gli attori che lo utilizzano.

Il sistema viene indicato con un riquadro che contiene i casi d'uso e ne riporta il nome, come nel nostro esempio.

Vediamo un secondo esempio.

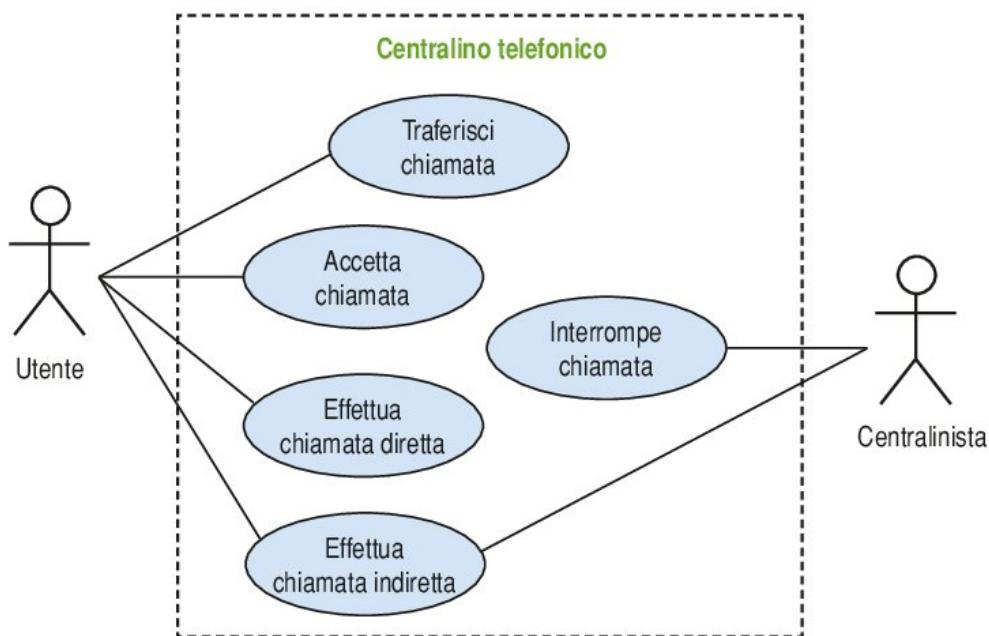
ESEMPIO **Centralino e chiamate telefoniche**

Si vuole realizzare un sistema di gestione delle chiamate telefoniche per un ufficio dove oltre alla linea diretta è presente anche un centralino per chi non conosce il numero degli interni.

Gli attori in questo sistema sono solamente due, gli **utenti** e la **centralinista**, ed elenchiamo i casi d'uso:

- l'utente effettua una chiamata diretta;
- la chiamata dell'utente viene accettata;
- la chiamata dell'utente viene trasferita;
- l'utente effettua una chiamata indiretta tramite centralinista;
- il centralinista interrompe una chiamata.

L'**Use Case Diagram** è il seguente:



Prova adesso!

• Diagramma dei casi d'uso

- 1 Realizza il diagramma dei casi d'uso di un sistema di vendita online dei biglietti del treno.
- 2 Realizza il diagramma dei casi d'uso di un sistema di prenotazione posti per un concerto.
- 3 Realizza il diagramma dei casi d'uso di un sistema di vendita dischi vinile e CD, indicando le diverse possibili alternative di spedizione dei prodotti al cliente.

■ Relazioni tra casi d'uso

Anche nella stesura degli **use case** si applica la metodologia top-down, partendo da un elenco dove la descrizione è generica e ad alto livello per poi raggiungere mediante affinamenti (e decomposizioni) un livello di dettaglio soddisfacente.

Nel processo di affinamento si possono individuare “segmenti” già “sviluppati” in altri progetti oppure individuare delle relazioni tra più **use case** e descriverle in modalità grafico nei diagrammi **UML**.

- A** È possibile **riutilizzare** gli **use case**, secondo due modalità.

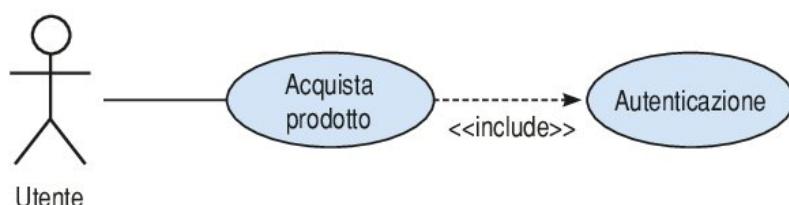
1 Inclusione (inclusion): questa modalità permette di utilizzare i passi appartenenti a una sequenza di un *use case* e inglobarli in un altro *use case*, cioè serve a rappresentare un caso d'uso che ne utilizza un altro.

Per rappresentare graficamente l'inclusione si traccia una freccia tratteggiata che punta sul caso da cui dipende l'altro caso, con la etichetta `<<include>>`.

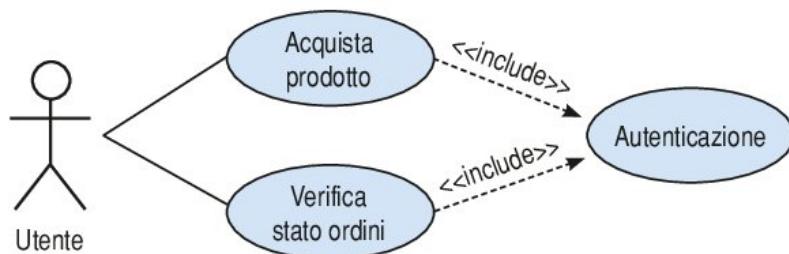
La parola **include** racchiusa tra doppie parentesi formate dai simboli “`<<` e `>>`” prende il nome di **stereotipo**: `<<include>>` è uno stereotipo.

ESEMPIO

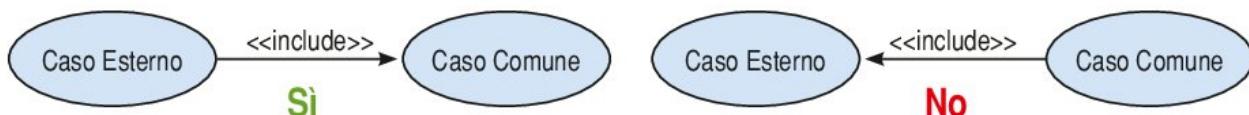
Nel nostro esempio abbiamo indicato genericamente la funzione *Acquista Prodotto* con la sequenza di operazioni che permettono di effettuare l'acquisto: questa attività è scomponibile in diverse parti, tra le quali sicuramente è presente la fase di *Autenticazione*, fase che potrebbe anche essere utilizzata in un ulteriore caso d'uso come la *Verifica dello stato dell'ordine* (da noi non considerata): quindi Autenticazione sarà indicata separatamente in un altro caso d'uso a se stante e nella descrizione dei casi d'uso che lo includono viene indicato col seguente diagramma:



Se ora aggiungiamo anche il caso d'uso *Verifica dello stato dell'ordine* otteniamo:



Un errore tipico è nel posizionamento della freccia, che deve andare dal caso più esterno al caso comune:



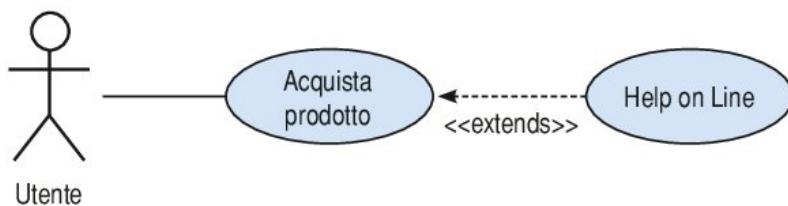
2 Estensione (extension): tramite questo metodo è possibile creare un nuovo *use case* semplicemente aggiungendo dei passi a un *use case* esistente: in questo modo è possibile integrare l'*use case* aggiungendo comportamenti alternativi o eccezionali (opzionali) che estendono il caso d'uso generale.

Anche in questo caso si utilizza una linea tratteggiata con una freccia finale insieme con uno stereotipo che mostra la parola <<extends>> tra parentesi.

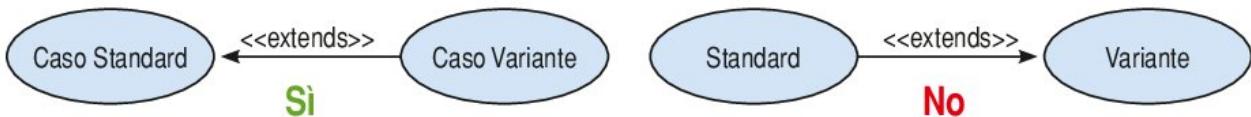
La relazione **extends** va utilizzata quando abbiamo uno *use case* simile a un altro che però fa qualcosa in più.

ESEMPIO

Aggiungiamo per esempio al nostro *NegozioOnline* un *Help on Line* che, per esempio, può essere richiamato durante l'acquisto di un prodotto: la situazione nel diagramma è la seguente



Anche in questo caso un errore tipico è nel posizionamento della freccia, che deve andare dal caso variante al caso standard:



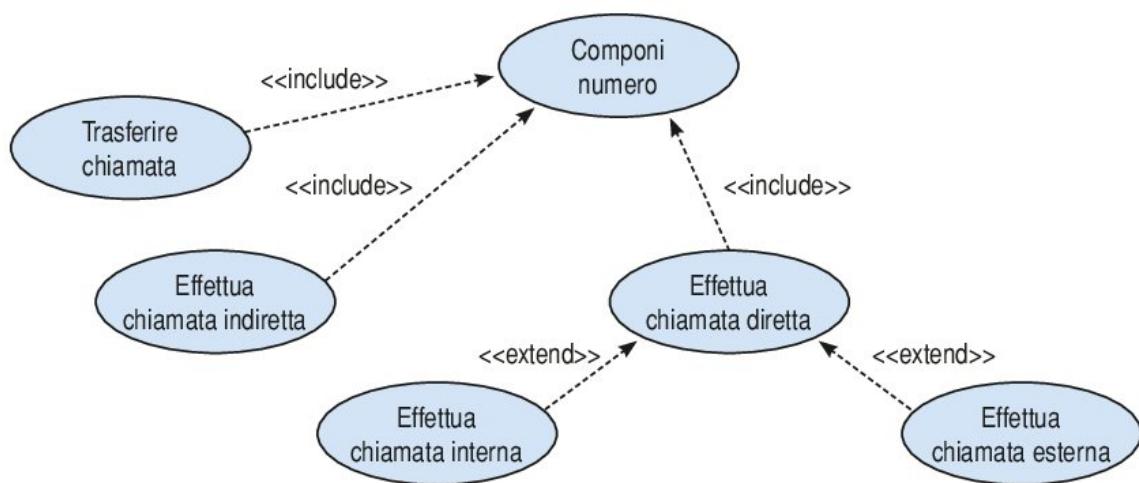
È bene osservare una semplice regola per effettuare un utilizzo corretto (e moderato!) delle inclusion ed extension:

- utilizzare **includes** quando si sta ripetendo la descrizione di un comportamento presente già in uno o più differenti *use case* e il suo utilizzo è **sempre richiesto**;
- utilizzare **extends** quando si descrive una variazione al comportamento normale (**utilizzo opzionale**).

ESEMPIO

Esempio centralino telefonico (seguito)

Se rianalizziamo l'esempio del centralino telefonico potremmo descrivere le attività in termini di inclusione ed estensione come di seguito riportato, dove l'unica attività è quella di effettuare la chiamata e le altre attività sono "varianti" o "possibilità" della stessa operazione:



B Generalizzazione (generalization)

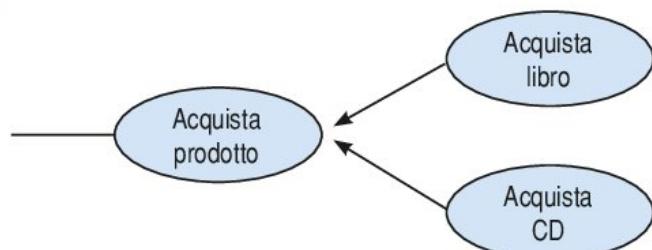
Un *use case* può inoltre ereditare le caratteristiche di un altro *use case* (come nel caso di ereditarietà delle classi nella OOP): lo *use case* figlio eredita il comportamento e il significato dal padre e in più aggiunge le sue caratteristiche specifiche.

Come per le classi la relazione di ereditarietà deve soddisfare la relazione IS-A, nel caso di *use case* è possibile applicare lo *use case* figlio dove è possibile applicare il padre.

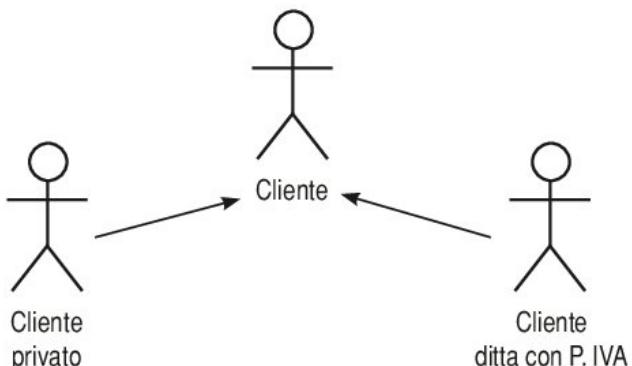
La **generalizzazione** viene rappresentata con una linea continua che ha un triangolo aperto che parte dai figli e punta al padre.

ESEMPIO Negozio online (seguito)

Nell'esempio del *NegozioOnline* potrei avere due tipi di prodotti, libri e CD: per descrivere che possono essere entrambi delle "specializzazioni" della funzione *Acquista prodotto* (che è il caso generale) utilizzo la seguente notazione:



La relazione di **generalizzazione** può esistere anche tra **actors**: sempre nell'esempio del *NegozioOnline* potrei avere due tipi di clienti, gli utenti privati o le ditte (con partita IVA), che il sistema deve trattare in modo differente (scontrino fiscale per i primi e fattura per i secondi); li rappresento col seguente diagramma:



■ Documentazione dei casi d'uso

La descrizione dei **casi d'uso** del sistema costituisce un capitolo molto importante del documento di specifica dei requisiti: a ogni caso d'uso mostrato nel diagramma deve essere associata una descrizione per renderlo comprensibile a tutti coloro che lo devono poi leggere e verificare, tra i quali sono presenti gli **stakeholder** che non hanno conoscenze informatiche. Quindi si utilizza il linguaggio naturale in modo semplice e chiaro, in modo da non creare ambiguità. Lo schema adottato da tutti i progettisti è quello riportato di seguito, dove in ogni caso d'uso si descrivono tutti gli scenari, a partire da quello principale che è quello che accade nella grande maggioranza dei casi, detto **basic flow** o “**happy path**”:

Nome del caso d'uso	<xyz>
Descrizione	<operazione effettuata>
Scopo	<sintesi della attività>
Attori	<descrizione degli attori coinvolti nel caso d'uso>
Attore primario	<se un attore ha prevalenza sugli altri sull'esito del caso d'uso>
Use Case d'extend	<use case che potrebbe seguire a quello corrente>
Scenario principale	<nome dello scenario principale>
1 Entry condition (precondizione):	è un vincolo che il sistema deve rispettare affinché il caso d'uso possa iniziare; si incomincia con “questo caso d'uso inizia quando...”;
2 Flusso di Eventi:	<descrizione in linguaggio naturale informale>;
3 Exit condition (postcondizione):	<è una condizione che è verificata quando il caso d'uso termina (può essere diversa a seconda dello scenario effettivamente seguito): si incomincia con “questo caso d'uso termina quando...”
Scenari alternativi	<descrizione delle modalità alternative di esecuzione del caso>
1 Flussi alternativi:	<descrizione di cosa accade in tutte le situazioni di errore, cioè quando non va a buon fine il caso principale>
2 Eccezioni:	<descrizione di cosa accade in tutte le situazioni di errore, cioè quando non va a buon fine il caso principale>
Requisiti Speciali:	<sono eventuali requisiti non funzionali (cioè non relativi alle funzionalità del sistema nell'assolvere al caso d'uso) e i vincoli>
Extension Points:	<sono relazioni con eventuali altri casi d'uso correlati>
Frequenza stimata di utilizzo:	<per decidere le priorità nel piano di sviluppo>
Criticità:	<per stimare il rischio legato al requisito>
Specializza il caso d'uso (opzionale)	<nome del caso d'uso generico del quale il caso d'uso corrente costituisce una <i>specializzazione</i> >
Generalizza il caso d'uso (opzionale)	<nome del caso d'uso specifico del quale il caso d'uso corrente costituisce una <i>generalizzazione</i> >

Lo schema descritto è una interpretazione di quello specificato da **Ivar Jacobson**, ideatore assieme a **Grady Booch** e **James Rumbaugh** dell'**UML**.

ESEMPIO

Descrizione degli scenari del Negozio Online

Vediamo sempre nell'esempio **NegozioOnline** che come scenario principale abbiamo il successo dell'acquisto mentre gli scenari indesiderati (**alternative flows**) possono per esempio essere:

- ▶ carta di credito non accettata;
- ▶ disponibilità di credito non sufficiente;
- ▶ collegamento con i servizi interbancari interrotto;
- ▶ disponibilità di un prodotto terminata.

La scheda di documentazione è ad esempio la seguente:

Nome del caso d'uso	Acquisto di uno o più prodotti
Descrizione	Un utente registrato effettua un acquisto on-line
Scopo	Scelta prodotto, aggiornamento carrello, pagamento
Attori	Utente, Sistema Bancario
Attore principale	Utente
Use Case d'extend	Organizzazione spedizione prodotti, fatturazione
Scenario principale	Acquisto andato a buon fine
1 Entry condition (precondizione):	per poter fare un acquisto on-line l'utente deve essere registrato;
2 Flusso di Eventi:	<ol style="list-style-type: none"> 1. Il cliente ricerca nel catalogo e inserisce nel carrello uno o più articoli 2. Il cliente va "alla cassa" 3. Il sistema presenta il conto degli articoli selezionati 4. Il cliente inserisce le informazioni per la spedizione (indirizzo, tempo di consegna) 5. Il sistema fornisce il conto totale, comprese le spese di spedizione 6. Il cliente inserisce le informazioni riguardo la sua carta di credito 7. Il sistema autorizza l'acquisto 8. Il sistema conferma il perfezionamento con successo dell'ordine 9. Il sistema invia una e-mail di conferma dell'acquisto all'indirizzo indicato dal cliente
3 Exit condition (postcondizione):	se il cliente conferma l'ordine esso viene passato al magazzino, se invece l'utente lo annulla il sistema rimane inalterato
Scenari alternativi	
1 Flussi alternativi:	nessuno
2 Eccezioni	<p>Eccezione a) Carta di credito non valida</p> <ul style="list-style-type: none"> ▶ Il sistema al passo 7 del Basic Flow non autorizza l'acquisto. ▶ Il sistema avverte l'utente e gli consente di reinserire i dati <p>Eccezione b) Carta di credito non valida</p> <ul style="list-style-type: none"> ▶ Il sistema al passo 7 del Basic Flow non autorizza l'acquisto. ▶ Il sistema avverte l'utente del credito esaurito e gli consente di reinserire i dati di una diversa carta di credito. <p>Eccezione c) Collegamento con i servizi interbancari interrotto</p> <ul style="list-style-type: none"> ▶ Il sistema salva il carrello dell'utente ▶ lo invita a riprovare a perfezionare l'ordine in seguito <p>Eccezione d) <altre situazioni></p>
Requisiti Speciali:	Il sistema deve garantire che l'inoltro dell'ordine al magazzino avvenga entro le 24 ore successive alla sua conferma. Gli articoli richiesti dagli utenti devono essere presenti in magazzino almeno il 90% delle volte
Extension Points:	sistema di consegna della merce
Frequenza stimata di utilizzo:	un utente al minuto
Criticità:	tempo di risposta

◀ Alistair Cockburn ▶, autore di un libro sui **casi d'uso**, dà le seguenti indicazioni:

Molti si sentono colpevoli se lo scenario principale di un caso d'uso è breve, così lo allungano per arrivare a quindici, o anche trentacinque righe. Personalmente, io non ho mai scritto uno scenario principale più lungo di nove passi. Non che il nove sia un numero magico; il fatto è che, quando ho individuato i sotto-goal a un giusto livello e ho eliminato i dettagli che riguardano la progettazione, mi restano sempre meno di nove passi. A volte, lo scenario principale di un caso d'uso può essere anche di soli tre passi.

Il valore maggiore di un caso d'uso non è nello scenario principale, ma nei comportamenti alternativi. Lo scenario principale può occupare da un quarto a un decimo della lunghezza totale di un caso d'uso, perché ci possono essere molte alternative da descrivere. Se lo scenario principale fosse lungo trentacinque passi, l'intero caso d'uso occuperebbe dieci pagine, e sarebbe troppo lungo da leggere e da comprendere. Se lo scenario principale contiene da tre a nove passi, la descrizione complessiva potrebbe essere di solo due o tre pagine, il che è più che sufficiente.

Se potete evitare di includere troppi dettagli dell'interfaccia utente, i casi d'uso saranno molto più facili da leggere. E i casi d'uso leggibili possono in effetti venire letti. Casi d'uso lunghi e illeggibili vengono soltanto firmati – di solito con sgradevoli conseguenze sul progetto, alcuni mesi più tardi.

AREA digitale

Consigli per fare un buon Use Case Diagram



◀ Alistair Cockburn is a Senior Consultant with Cutter Consortium's Agile Product & Project Management practice. He is consulting fellow at Humans and Technology, where he helps clients succeed with object-oriented projects, including corporate strategy, project setup, staff mentoring, process development, technical design, and design quality. ▶



Prova adesso!

- Scheda dei dati d'uso

Individuato il seguente scenario come principale tra quelli descritti nel caso d'uso del sistema di vendita online dei biglietti treni, realizza la scheda completa.

Nome del caso d'uso	Consultare gli orari ferroviari
Fasi scenario principale	<ol style="list-style-type: none"> L'internauta inserisce le città di partenza e di destinazione, le date di partenza e ritorno (se biglietto andata/ritorno). Il sistema visualizza la lista dei viaggi che corrispondono alla selezione. L'internauta seleziona un viaggio. Il sistema visualizza i dettagli del viaggio. L'utente può acquisire i biglietti del viaggio selezionato. Viene chiamato l'use case "Comprare un biglietto"
Scenari alternativi	<p>2a Il sistema segnala l'inesistenza di un viaggio corrispondente alla scelta dell'internauta.</p> <p>2b Il sistema visualizza, secondo i casi, la mancanza o un errore nell'inserimento dell'informazione per: la data di partenza, la città di partenza, la città di destinazione, la data di ritorno (se biglietto andata/ritorno)</p>

Verifichiamo le competenze

Per ciascuna situazione descritta in seguito viene richiesto di:

- 1** definire gli attori
- 2** definire gli use case, diagramma e documentazione, aiutandosi con il seguente elenco di domande:
 - chi sono gli attori principali e secondari
 - quali sono gli obiettivi degli attori
 - quali sono le precondizioni
 - quali sono le principali azioni richieste/svolte dagli attori
 - quali eccezioni vanno considerate nello svolgimento della storia
 - quali variazioni vanno considerate nelle interazioni con gli attori
 - quali sono le informazioni acquisite, prodotte e modificate dagli attori

1. Esercizi

1 Supermercato

Viene richiesto di sviluppare un prodotto per offrire ai clienti di un supermercato un sistema di vendita online con il quale i clienti facciano via web le stesse cose che fanno quando visitano il supermercato fisico.

Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.

Dopo aver individuato il caso d'uso più importante, descrivilo completamente con lo schema di Jacobson.

2 Supermercato bis

In riferimento all'esercizio precedente relativo al supermercato, descrivi la specifica del caso d'uso 'aggiorna carrello', dove il cliente può eseguire due operazioni:

- aggiungere un nuovo prodotto al carrello;
- modificare la nuova quantità di un articolo esistente;
- rimuovere l'articolo dal carrello.

Descrivi la specifica di questo nuovo caso d'uso.

3 Supermercato ter

In riferimento all'esercizio precedente relativo al supermercato, viene introdotto un sistema di consegna a domicilio della spesa per pensionati.

Descrivi la specifica di questo nuovo caso d'uso.

4 Medico Generico

Un medico vuole realizzare un sistema per gestire i propri pazienti in modo da non dover chiedere al paziente la sua situazione e poter annotare di volta in volta cure, situazione, osservazioni ecc.

Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.

Dopo individuato il caso d'uso più importante, descrivilo completamente con lo schema di Jacobson.

5 Medico Generico bis

In riferimento all'esercizio precedente relativo al medico, viene introdotto un sistema di prenotazione online degli appuntamenti.

Descrivi la specifica di questo nuovo caso d'uso.

6 Medico Generico ter

In riferimento all'esercizio precedente relativo al medico, viene introdotto un sistema di richiesta ricette ripetitive mediante email.

Descrivi la specifica questo nuovo caso d'uso.

7 Museo

I visitatori di un museo possono accedervi comprando un biglietto venduto da un addetto alla biglietteria o usando biglietti acquistati precedentemente e possono richiedere di essere accompagnati da una guida. È previsto uno sconto per alcune categorie di utenti, per i gruppi e le scolaresche.

Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.

Dopo aver individuato il caso d'uso più importante, descrivilo completamente con lo schema di Jacobson.

8 Museo bis

In riferimento all'esercizio precedente relativo al museo, viene introdotto un sistema di prenotazione on-line per le scuole.

Descrivi la specifica di questo nuovo caso d'uso.

9 Biglietteria ferroviaria

Il sistema deve permettere ai viaggiatori di comprare i biglietti (andata e/o ritorno) e gli abbonamenti (settimanale, mensile, annuale); le tariffe vengono aggiornate da un sistema centralizzato.

Oltre alla biglietteria dove è presente un bigliaio, è in funzione anche un distributore automatico di biglietti che però ha alcune possibili problematiche nell'emissione dei biglietti (mancanza di carta, di resto ecc.).

Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.

Dopo aver individuato il caso d'uso più importante, descrivilo completamente con lo schema di Jacobson.

10 Biglietteria bis

In riferimento all'esercizio precedente relativo alla biglietteria, viene introdotto un sistema di prenotazione on-line per i biglietti di I classe e le tratte superiori a 50 km.

Descrivi la specifica di questo nuovo caso d'uso.

11 Biglietteria ter

In riferimento all'esercizio precedente relativo alla biglietteria, viene introdotto un sistema di sconti per le famiglie.

Descrivi la specifica di questo nuovo caso d'uso.

12 Sportello servizi comunali

Si vuole realizzare un sistema di sportello automatico per offrire alcuni servizi ai cittadini, come la stampa di certificati e il pagamento di tributi comunali (IMU, multe ecc.).

Il cittadino viene autenticato mediante la lettura della tessera sanitaria oppure con la digitazione del codice fiscale seguito dal PIN personale.

Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.

Descrivi il caso d'uso della stampa certificati con lo schema di Jacobson.

13 Sportello servizi comunali bis

Descrivi gli scenari di successo e di insuccesso relativi al pagamento di una multa in contanti con errori dovuti al codice della multa non corretto, al time-out del sistema, all'annullamento esplicito dell'operazione, alla mancanza di fondi.

Descrivi la specifica di questo nuovo caso d'uso.

14 Sportello servizi comunali ter

Tra i documenti comunali è anche prevista la richiesta della cartografia tecnica regionale (CTR) nelle scale 1:5.000, 1:10.000 e 1:25.000 nei formati cartaceo o elettronico relativa alla zona in cui il cittadino abita.

Nel caso di richiesta del formato elettronico, questa viene inviata immediatamente per e-mail all'utente mentre nel caso di richiesta cartacea viene plottata in un tempo successivo alla richiesta e comunicata all'utente sempre per e-mail la disponibilità della copia da ritirarsi presso l'ufficio tecnico.

La quota da pagare è diversa a seconda che la richiesta provenga da un privato cittadino o da un tecnico professionista.

Identifica gli attori, i casi d'uso e disegna il diagramma di contesto.

Dopo individuato il caso d'uso più importante, descrivilo completamente con lo schema di Jacobson.

La documentazione dei requisiti

In questa lezione impareremo...

- il documento di Specifica dei Requisiti Software (SRS)
- le caratteristiche e la struttura di un SRS
- la validazione e convalida delle specifiche di un SRS

■ Generalità

Il lavoro di **individuazione dei requisiti** termina con la stesura del documento di **Specifiche dei Requisiti Software(SRS)**: questo è un documento ufficiale e descrive quello che è richiesto allo sviluppatore del sistema come risultato del lavoro di analisi tra il cliente, l'utente e lo sviluppatore. Un errore nell'**SRS** produrrà errori nel sistema finale: la rimozione di un difetto scoperto dopo il rilascio del sistema può costare fino a 100 volte quella di un difetto individuato in fase di progettazione.

Nel documento **SRS** deve essere presente una analisi approfondita dell'utente, delle sue reali necessità, l'elenco dei singoli requisiti specificati e completati con i relativi **casi d'uso**.

Abbiamo quindi tre tipologie di informazioni che devono “emergere” nell'analisi:

► analisi degli **utenti**:

- a quali categorie di utenti è destinato il prodotto?
- quali sono le loro caratteristiche e le loro priorità?

► analisi dei **bisogni**:

- quali sono le necessità di ciascuna categoria di utenti?
- quali sono prioritari?

► analisi del **contesto d'uso**:

- quali saranno i diversi contesti d'uso del prodotto da parte delle diverse categorie di utenti?
- quali sono prioritari?

Per la buona riuscita dello sviluppo del sistema l'**SRS** deve essere di alta qualità in quanto questo documento aiuta lo sviluppatore a comprendere il dominio di applicazione, riduce i tempi di sviluppo e fornisce un punto di riferimento per la convalida del prodotto.

Nel **documento dei requisiti (Requirements Documents)**, quindi, dopo una parte generale che descrive il sistema e le informazioni sulla richiesta del committente vengono raccolti in modo organizzato i **requisiti** individuati e su come ha avuto luogo la loro determinazione, e vengono allegati i diagrammi d'uso e le rispettive schede che li descrivono.

Per redigere un efficace documento **SRS** gli sviluppatori seguono i suggerimenti dello standard **IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specification**. Esistono anche altri modi per scrivere **SRS** di qualità, comunque lo Standard 830 è quello più diffuso ed è quello al quale anche noi faremo riferimento.

■ Requirements Documents proposto da Sommerville

Esistono diversi standard che propongono la forma del documento **SRS** e la metodologia di compilazione: noi descriveremo la proposta di **Sommerville** che si ispira allo **standard IEEE/ANSI 830-1998**.

1 Introduzione

1. Scopo del documento dei requisiti
2. Scopo del prodotto
3. Definizione, acronimi ed abbreviazioni (glossario)
4. Riferimenti
5. Overview dell'intero documento

2 Descrizione generale

1. Prospettive sul prodotto
2. Funzioni del prodotto
3. Caratteristiche degli utenti
4. Vincoli generali
5. Assunzioni e dipendenze

3 Requisiti specifici

1. Definizione dei **Requisiti funzionali** (requisiti utente)
2. Definizione dei **Requisiti non funzionali** (requisiti utente)
3. Architettura: strutturazione in sottosistemi a cui riferire i requisiti
4. Specifiche dei **Requisiti di sistema**
5. Modelli del sistema
6. Evoluzione del sistema

4 Appendici

1. Individuazione ed eventuale descrizione della piattaforma hardware
2. Requisiti di DataBase
3. Piani di Test

5 Indici

All'interno del documento vengono utilizzati i seguenti termini **IEEE**:

- **contratto (Contract)**: il documento (legale) stilato tra committente e fornitore (la specifica dei requisiti potrebbe/dovrebbe essere la parte integrante del contratto);
- **committente (Customer)**: colui che paga il prodotto che di solito (ma non necessariamente) è colui che decide i requisiti;
- **fornitore (Supplier)**: chi produce il prodotto / sviluppa per il committente;
- **utente (User)**: la persona che usa e interagisce direttamente col sistema (spesso si identifica col committente).

Per ciascuna funzionalità descritta nella sezione 3.1 devono essere specificati gli input, l'output e l'azione elaborativa.

Per le altre componenti del SRS non sono necessarie ulteriori spiegazioni.

Vediamo, in un esempio, come viene effettuata la descrizione di una singola specifica funzionalità (requisito utente) che, come più volte detto, deve rispondere alla seguente domanda: *Cosa deve fare il sistema software?*

ESEMPIO Negozio Online: funzionalità Inserimento articolo

Descriviamo la funzionalità di inserimento di un articolo nel carrello del **NegozioOnline** descritto nelle precedenti lezioni:

3.x Inserisci articolo nel carrello

Utenti della funzionalità:

clienti registrati

Introduzione:

la funzionalità consente l'immissione di un nuovo articolo nel carrello di acquisto

Precondizione:

l'utente si è precedentemente autenticato come cliente registrato

Input:

codice articolo: obbligatorio

quantità: obbligatorio (di default proposto 1)

Elaborazione:

viene presentata una videata di ricerca degli articoli a più livelli e modalità

selezionato un articolo, si visualizza un form per l'immissione dei dati di input

viene effettuata la lettura dei dati di input

viene verificato che la quantità sia ammissibile

viene verificata la disponibilità dell'articolo e eventualmente segnalati i tempi di attesa

Output:

aggiunta di una riga nel carrello

segnalazione di errore nel caso di articolo già presente

Sempre del nostro **NegozioOnline** vediamo un altro esempio di funzionalità, quella relativa all'elenco dei clienti attivi.

ESEMPIO Negozio Online: funzionalità Elenco Clienti Attivi

3.y Elenco Clienti Attivi

Utenti della funzionalità:

Titolare

Introduzione

La funzionalità visualizza la lista dei clienti che hanno effettuato acquisti nell'ultimo anno con il totale degli acquisti effettuati e con possibilità di stampa.

Consente, se richiesto, di selezionare un cliente, di visualizzarne la scheda personale e l'elenco degli articoli acquistati ed eventualmente di stamparla.

Precondizione

L'utente si è precedentemente autenticato come titolare del negozio.

Input

Nome titolare – Facoltativo
 Cognome titolare – Facoltativo
 Città – Facoltativo
 Codice Fiscale – Facoltativo
 P.I.V.A. – Facoltativo
 E-mail – Un solo indirizzo. Facoltativo

Un dato tra quelli elencati deve essere obbligatoriamente inserito per effettuare la ricerca

Elaborazione

Il sistema presenta una lista con l'elenco dei clienti con la possibilità di ricerca.
 Selezionando un cliente dalla lista è possibile richiedere la visualizzazione o la stampa della sua scheda riportante tutti i dati anagrafici e contabili del cliente.

Output

La lista con l'elenco dei clienti, visualizzata a schermo o stampata, riporta per ciascuno di essi:
 Nome – Cognome – Città – Numero Acquisiti – Data ultimo acquisto – Totale fatturato



Prova adesso!

Descrivi le seguenti funzionalità:

- inserimento di un cliente con verifica e-mail;
- cancellazione di una voce dal carrello;
- elenco degli articoli, con indicazione delle movimentazioni di carico/scarico.

◀ SRS standard IEEE/ANSI 830-1998

Table of contents

1. Introduction

- 1.1. Purpose (Scopo del documento)
- 1.2. Scope (Scopo del prodotto)
- 1.3. Definitions, Acronyms and Abbreviations
- 1.4. References
- 1.5. Overview (Descrizione generale del resto del documento)

2. General Description

- 2.1. Product Perspective
- 2.2. Product Functions
- 2.3. User Characteristics
- 2.4. General Constraints
- 2.5. Assumptions and Dependencies

3.1.1.4. Output

3.1.2. Functional Requirement 2

...

3.1.3. Functional Requirement n

...

3.2. External Interface Requirements

3.2.1. User Interfaces

3.2.2. Hardware Interfaces

3.2.3. Software Interfaces

3.2.4. Communications Interfaces

3.3. Performances Requirements

3.4. Design Constraints

3.4.1 Standard Compliance

3.4.2. Hardware Limitation

3.5. Software System Attributes

3.5.1. Security

3.5.2. Maintainability

...

3.6. Other Requirements

3.6.1. Logical Database Requirements

3.6.2. Operations

3.6.3. Site Adaptation requirements



3. Specific Requirements

- 3.1. Functional Requirements
- 3.1.1. Functional Requirement 1
- 3.1.1.1. Introduction
- 3.1.1.2. Inputs
- 3.1.1.3. Processing

Il documento completo [IEEE830.pdf](#) è disponibile nella cartella **materiali** della sezione dedicata a questo volume del sito <http://www.hoeplicuola.it/>.

■ Realizzare un efficace documento SRS

Il documento **SRS** riporta in modo corretto e non ambiguo le specifiche dei requisiti del software ma non descrive alcun dettaglio progettuale o implementativo e non impone vincoli addizionali al prodotto software, quali per esempio qualità, oggetto di documenti specifici.

Lo standard **IEEE 830** descrive alcune caratteristiche fondamentali di cui un efficace **SRS** deve essere in possesso, a partire dalla validazione dei requisiti.

Validazione dei requisiti

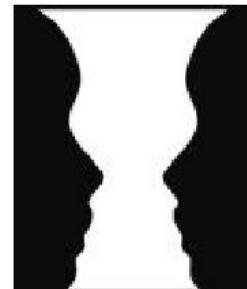
Durante il processo di sviluppo i requisiti devono essere continuamente validati da cliente e utenti; la validazione dei requisiti richiede di controllare:

- ▶ **correttezza:** un **SRS** è corretto se, e solo se, il sistema software soddisfa appieno ciascun requisito in esso riportato; una specifica è corretta se rappresenta accuratamente il sistema che il cliente richiede e che gli sviluppatori intendono sviluppare;
- ▶ **completezza:** una specifica è completa se tutti i possibili scenari per il sistema sono descritti, inclusi i comportamenti eccezionali dovuti a una qualunque combinazione di ingressi (validi o non validi); deve inoltre riportare i riferimenti alle figure, diagrammi e tabelle, la definizione dei termini e delle unità di misure utilizzati;
- ▶ **consistente:** nessun requisito deve essere in conflitto con altri requisiti;
- ▶ **coerenza:** i requisiti non devono contraddirsi tra di loro;
- ▶ **chiarezza:** una specifica è chiara se non è possibile interpretarla in due modi diversi; non possiamo quindi avere ambiguità di descrizione, dovute alla terminologia o semplicemente alla scarsa oppure elevata presenza di dettagli.

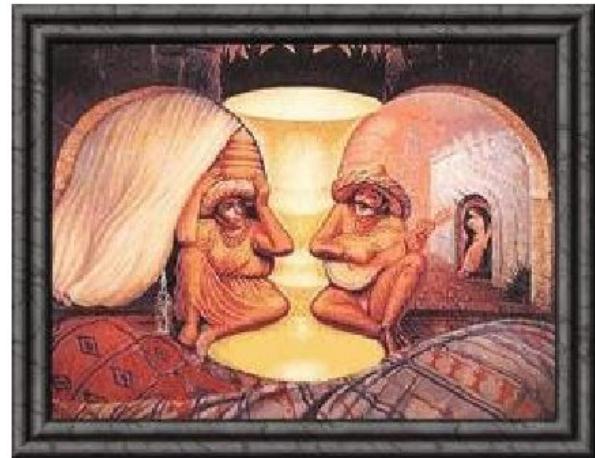
ESEMPIO

Ambiguità e dettagli

Il seguente disegno mostra un caso di ambiguità: se ci si focalizza sui contorni neri si vede il profilo di due volti, se si osserva il bianco si individua un vaso. ►



Anche il livello di dettaglio può portare all'introduzione di ambiguità: vediamo una versione "arricchita" dello stesso disegno dove è possibile individuare molte più ambiguità: ►



(da Giorgio Zanetti, zanblog.it).

- ▶ **realismo:** la specifica dei requisiti è realistica se può essere implementata tenendo conto dei vincoli presenti nel dominio del sistema;
- ▶ **modificabilità:** la struttura e lo stile dell'**SRS** devono essere tali da consentire facili modifiche, preservando consistenza e completezza (un **SRS** con ridondanze non si modifica facilmente);
- ▶ **verificabilità:** la specifica dei requisiti è verificabile se, e soltanto se, una volta che il sistema è stato costruito, test ripetuti possono essere delineati per dimostrare che il sistema soddisfa i requisiti. Un requisito è verificato se esiste un procedimento (di costo compatibile) tramite il quale una persona o una macchina può stabilire se il software soddisfa il requisito.

Esempi di requisiti **non verificabili**

- il programma deve funzionare bene
- il programma dovrebbe avere una buona interfaccia
- il tempo di risposta deve essere di norma 10 sec.

Esempi di requisiti **verificabili**

- il tempo di risposta all'evento X deve stare entro 10 sec.
- il tempo di risposta all'evento Y deve stare entro 10 sec. nel 60% dei casi

- ▶ **tracciabilità:** la definizione **IEEE** riporta “se l'origine di ciascun requisito è chiara e può essere referenziata nello sviluppo futuro ogni funzione del sistema deve poter essere individuata e ricondotta al corrispondente requisito funzionale”.

È necessario tracciare le dipendenze tra i requisiti e le funzioni del sistema, oltre che per effettuare lo sviluppo di test per poter successivamente introdurre eventuali cambiamenti del sistema.

La tracciabilità può essere realizzata assegnando un numero univoco a ciascun requisito

<tipo-requisito><numero-requisito>

e collegando i requisiti ai vari aspetti del sistema, una volta che anch'essi sono stati catalogati e numerati, mediante una tabella a doppia entrata come la seguente:

		Aspetti del sistema o del suo ambiente						
		A1	A2	A3	A4	A5	...	An
Requisiti	R1		✓		✓			
	R2		✓	✓				
...								
Rm	✓			✓	✓			

Il documento dei requisiti dovrebbe essere organizzato in modo tale che le modifiche possano essere eseguite senza la riscrittura del documento.



TRACCIABILITÀ DI UN SRS

Un SRS è tracciabile se:

- ▶ è chiara l'origine di ogni requisito (tracciatura all'indietro, ai documenti precedenti);
- ▶ ogni requisito ha un nome o un numero (tracciatura in avanti per i requisiti futuri).

La convalida delle specifiche

La stesura del **SRS** deve seguire pari passo lo sviluppo del prodotto software dato che generalmente non è possibile specificare tutti i dettagli durante la fase iniziale di raccolta dei requisiti anche perché i requisiti possono cambiare in corso d'opera (sistemi complessi possono avere anche anni come tempi di sviluppo e nel frattempo gli obiettivi/bisogni di una organizzazione possono cambiare).

Anche se la stesura del documento viene realizzata scrupolosamente spesso si commettono errori che, se scoperti in fase avanzata di realizzazione, possono provocare enormi danni (ritardo nello sviluppo e quindi lievitazione dei costi); i tipi di errori più frequenti sono:

- **requisiti non chiari:** i requisiti sono espressi male o sono state omesse delle informazioni; in questo caso i requisiti devono essere riscritti;
- **informazioni mancanti:** le informazioni mancano dal documento dei requisiti; l'ingegnere dei requisiti deve raccogliere delle altre informazioni;
- **conflitto tra requisiti:** sono presenti contraddizioni fra i vari requisiti oppure tra i requisiti e l'ambiente operativo; una negoziazione con il cliente può aiutare a rimuovere il conflitto;
- **requisiti non realistici:** il cliente dovrebbe essere consultato, i requisiti cancellati, modificati e resi più realistici.

L'individuazione di queste indicazioni viene fatta aiutandosi con una **check list** che solitamente comprende non più di 10 domande da sottoporre all'utente finale, come quelle riportate nel seguente esempio.

ESEMPIO **Check list per la convalida dei requisiti**

Esempio di check list

- 1 Sono state definite tutte le risorse hardware?
- 2 È stato specificato il tempo di risposta delle funzioni?
- 3 Sono state specificate tutte le interfacce esterne, hardware, software e relativamente ai dati?
- 4 Sono state specificate tutte le funzioni richieste dall'utente?
- 5 È possibile testare ogni requisito?
- 6 Sono state specificate tutte le risposte a condizioni eccezionali?
- 7 È stato definito lo stato iniziale del sistema?
- 8 Sono state specificate le future possibili modifiche?

Non è semplice individuare gli errori presenti in un documento **SRS** e a tal fine è importante effettuare frequenti **revisioni** e **ispezioni** coinvolgendo sempre un rappresentante di tutti gli attori compreso l'autore del documento, il cliente, un progettista, un esperto di qualità: prima di organizzare le riunioni specifiche è necessario consegnare una copia del documento da analizzare a tutti i partecipanti invitandoli a rivedere il documento prima della riunione per poter segnalare in quella sede eventuali osservazioni.

È anche molto utile far leggere il documento a qualcuno diverso dall'autore per coglierne potenziali problemi di ambiguità o incompletezza (**Tecnica di Reading**).

Il primo metodo per eliminare gli errori resta comunque sempre quello "di non farli": concludiamo quindi questa trattazione indicando alcuni semplici consigli per ottenere una specifica precisa e priva di ambiguità:

- evitare termini troppo generici o troppo precisi;
- mantenere un livello di astrazione costante;
- riferirsi allo stesso concetto sempre nello stesso modo così da evitare l'uso di sinonimi (termini diversi con il medesimo significato) e omonimi (termini uguali con differenti significati);
- usare frasi brevi e semplici possibilmente uniformandone la struttura;
- dividere il testo in paragrafi e dedicare ogni paragrafo alla descrizione di una specifica entità della realtà modellata, evidenziandola in ogni paragrafo.

Verifichiamo le competenze

1. Esercizi

1 Situazione: NegozioOnline

In riferimento al sistema NegozioOnline descritto nelle precedenti lezioni, descrivi le seguenti funzionalità nel documento SRS secondo lo schema proposto dallo standard IEEE 830:

- 3.e Registrazione di un nuovo cliente
- 3.f Cancellazione di un vecchio cliente
- 3.g Variazione dei dati di un nuovo cliente
- 3.h Inserimento di un nuovo articolo in magazzino
- 3.i Eliminazione di un vecchio articolo in magazzino

2 Situazione: biblioteca scolastica

Il sistema dovrà archiviare i dati di una biblioteca scolastica dove oltre ai libri, ai giornali e alle riviste, è presente una sezione multimediale con video, nastri audio e CD-ROM.

Il sistema dovrà permettere agli utenti di fare delle ricerche per titolo, autore, categoria o codice ISBN mentre il bibliotecario deve poter stampare l'elenco dei ritardi di consegna per poter fare i dovuti solleciti.

Il sistema dovrà riconoscere l'utente attraverso la stessa smart-card che lo identifica all'interno della scuola, utilizzata per esempio per rilevare le presenze e i ritardi

Dopo aver individuato e realizzato i diagrammi dei casi d'uso, descrivi le seguenti funzionalità nel documento SRS secondo lo schema proposto dallo standard IEEE 830:

- 3.m Inserimento di una nuova categoria
- 3.n Cancellazione di un alunno
- 3.o Inserimento di un nuovo libro
- 3.p Elenco dei ritardi di consegna per i libri in base a una data specifica
- 3.q Elenco dei ritardi di consegna per una classe dell'istituto

3 Situazione: gite scolastiche

Il sistema richiesto serve per gestire le gite scolastiche e deve essere in grado di memorizzare nuovi itinerari con le seguenti informazioni: destinazione, giorni, tipo, descrizione, numero minimo e massimo di partecipanti, costo, anno di corso, optional.

Possono essere presenti più escursioni/itinerari per la medesima destinazione

I tipi si distinguono in base alla regione e/o alla nazione di destinazione.

Gli optional riguardano il vitto, se è compreso, escluso, solo cena ecc.

Il sistema deve permettere di prenotare un itinerario da parte di una o più classi, in base al numero richiesto di partecipanti: all'atto della prenotazione devono essere raccolte le autorizzazioni da parte dei genitori per gli alunni minorenni e un acconto pari al 10% dell'importo.

Il sistema deve permettere di modificare/cancellare un itinerario solo se non sono state fatte prenotazioni: deve anche permettere di annullare una escursione prenotata in caso di motivazioni gravi (eventi atmosferici, bellici, motivi didattici ecc.).

Un singolo alunno può rinunciare a una gita portando la documentazione medica: in questo caso il costo totale a saldo deve essere ripartito sul resto della comitiva.

4 Dopo aver individuato e realizzato i diagrammi dei casi d'uso, descrivi le seguenti funzionalità nel documento SRS secondo lo schema proposto dallo standard IEEE 830:

- 3.k Inserimento di un nuovo itinerario per una vecchia destinazione
- 3.l Inserimento di un nuovo itinerario per una nuova destinazione
- 3.m Cancellazione di un alunno da una gita
- 3.n Stampa del programma di un particolare itinerario scelto in base alla destinazione
- 3.o Elenco delle escursioni per tutte le classi di un anno di corso
- 3.p Elenco delle escursioni effettuate da una classe nei diversi anni

ESERCITAZIONI DI LABORATORIO 1

LA REALIZZAZIONE DEGLI USE CASE DIAGRAM CON STARUML

■ Generalità

Esistono molti prodotti che permettono di realizzare gli schemi **UML**, tra i quali ricordiamo:

- ▶ **StarUML**, scaricabile gratuitamente all'indirizzo <http://staruml.io/download> oppure dalla cartella **materiali** della sezione dedicata a questo volume del sito <http://www.hoepliscuola.it/>;
- ▶ **Visual Paradigm for UML Community Edition**, scaricabile gratuitamente all'indirizzo http://www.visual-paradigm.com/download/0sp1/support_uml2_xmi.html;
- ▶ **ArgoUML**, pacchetto open source di modellizzazione dei diagrammi con lo standard **UML 1.4 diagrams** scaricabile gratuitamente all'indirizzo <http://argouml.tigris.org> (richiede JVM Java);
- ▶ **Microsoft Visio**, strumento a pagamento per la creazione di diagrammi che offre anche la possibilità di condivisione dei progetti sul Web in tempo reale.

In questa esercitazione presenteremo **StarUML**, un progetto open source per lo sviluppo rapido, flessibile, estensibile di diagrammi **UML**, secondo il paradigma **MDA** (Model Driven Architecture).



Zoom su...

MODEL DRIVEN ARCHITECTURE

La metodologia **Model Driven Architecture (MDA)** fornisce un insieme di linee guida (standard) intese a permettere un approccio integrato allo sviluppo del software, dove i modelli sono considerati parte integrante del processo di implementazione. Il suo utilizzo generalmente ha numerosi vantaggi in termini sia di flessibilità per l'implementazione, l'integrazione e la manutenzione, il collaudo e la simulazione, sia per quanto riguarda la portabilità, la interoperabilità e riusabilità in un ampio arco di tempo.

StarUML è ricco di funzioni, flessibile ed estendibile grazie alla sua architettura a plug-in e alla disponibilità di apposite **API**; con una semplice interfaccia **IDE** consente la composizione dei diagrammi **UML** di base: **Use Case**, **Class**, **Sequence**, **Collaboration**, **Statechart**, **Activity**, **Component** e **Deployment**.

Attualmente **StarUML** è aggiornato all'**UML 2.0** dato che segue continuamente le modifiche che l'**OMG** (◀ **Object Management Group** ▶) apporta alle specifiche **UML**.

◀ **Object Management Group** **OMG's** mission is to develop, with our worldwide membership, enterprise integration standards that provide real-world value. **OMG** is also dedicated to bringing together end-users, government agencies, universities and research institutions in our communities of practice to share experiences in transitioning to new management and technology approaches like **Cloud Computing**. ▶

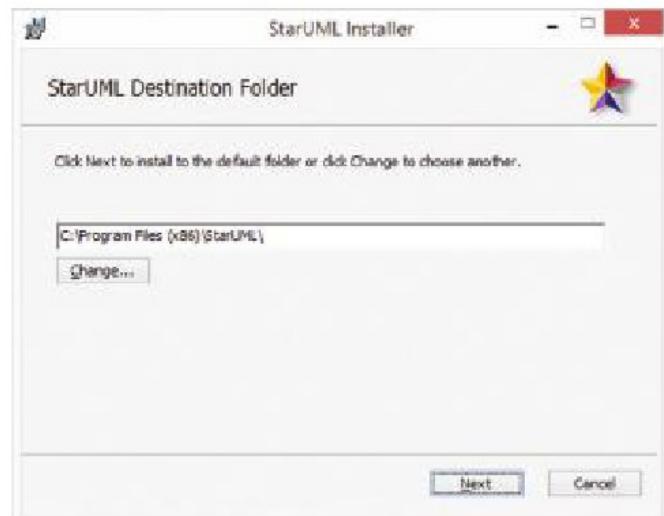


È scritto in **Delphi** e con l'installazione degli appositi plug-in, **StarUML** fornisce pieno supporto ai linguaggi più diffusi quali **C/C++**, **Java**, **Visual Basic**, **Delphi**, **JScript**, **VBScript**, **C#**, **VB.NET** e offre funzionalità uniche come quelle che consentono la creazione automatica di documenti della suite **Microsoft Office** (**Word**, **Excel**, **PowerPoint**).

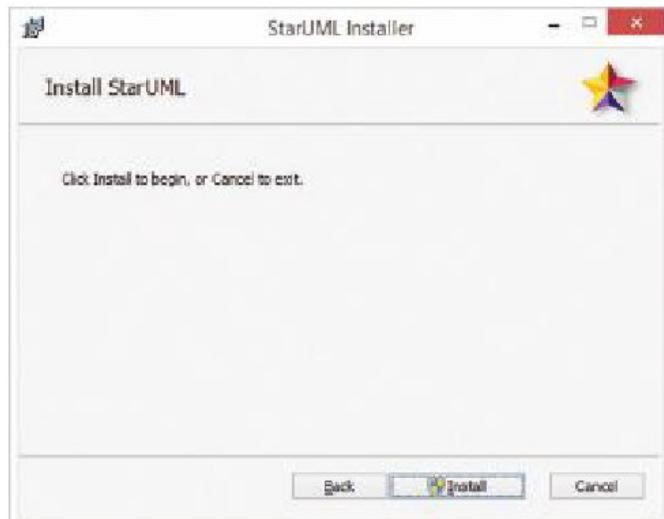
È apprezzata particolarmente la possibilità di effettuare la **generazione di codice** a partire dai diagrammi.

Installare StartUML

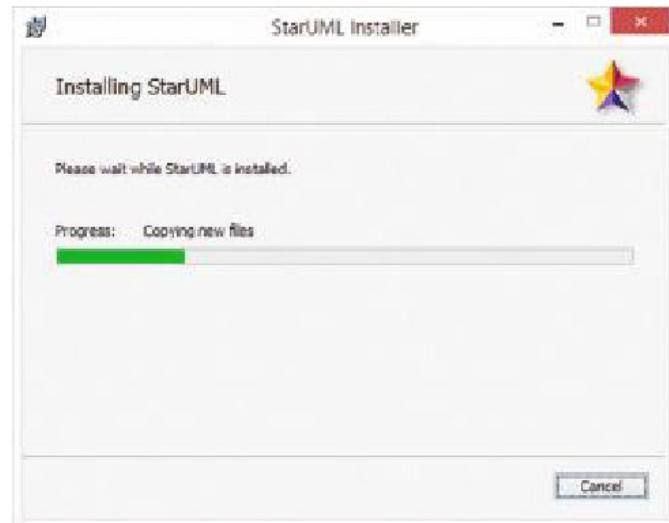
Dopo aver scaricato il **Staruml-v2.5.1.exe** (oppure una versione più recente), mandiamolo in esecuzione con un doppio click; la prima videata che ci viene presentata è la classica finestra per la scelta del percorso di installazione: ▶



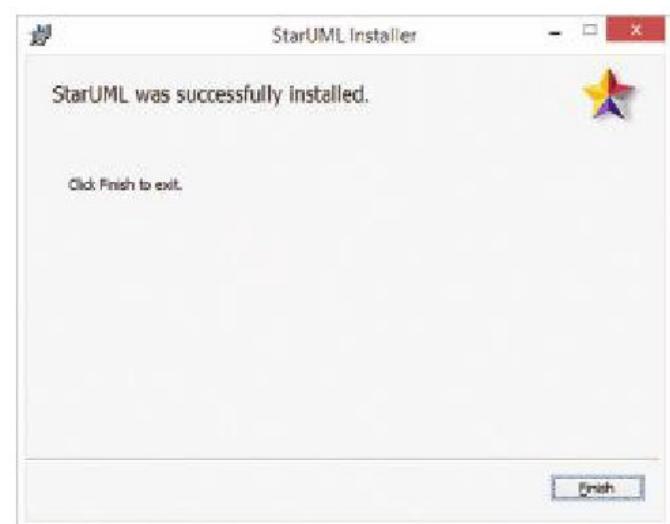
Procediamo senza modificare quella proposta in automatico e confermiamo anche la successiva richiesta di conferma di inizio installazione: ▶



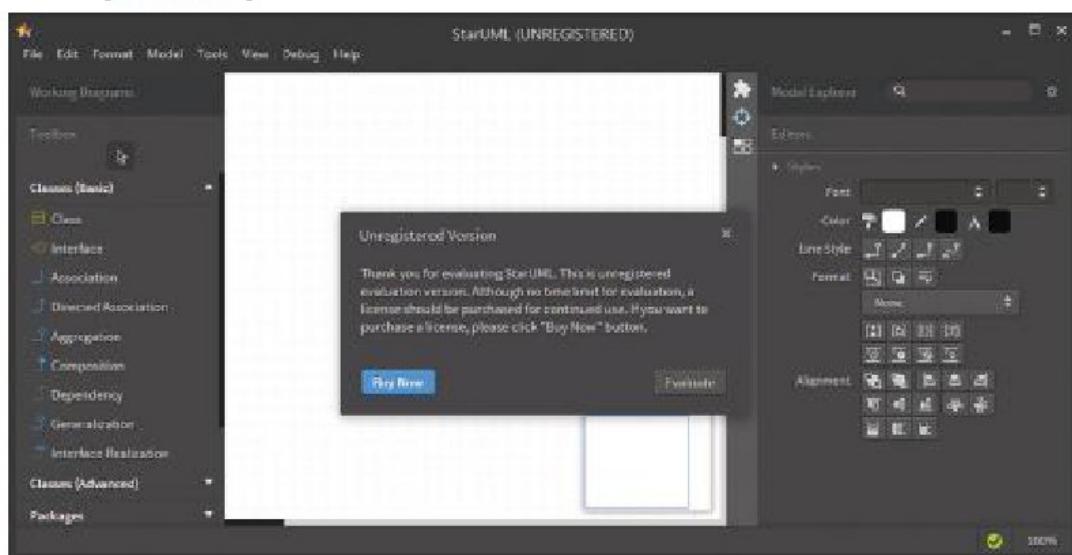
Cliccando su [Install] si avvia l'estrazione e la copia dei file del programma: ►



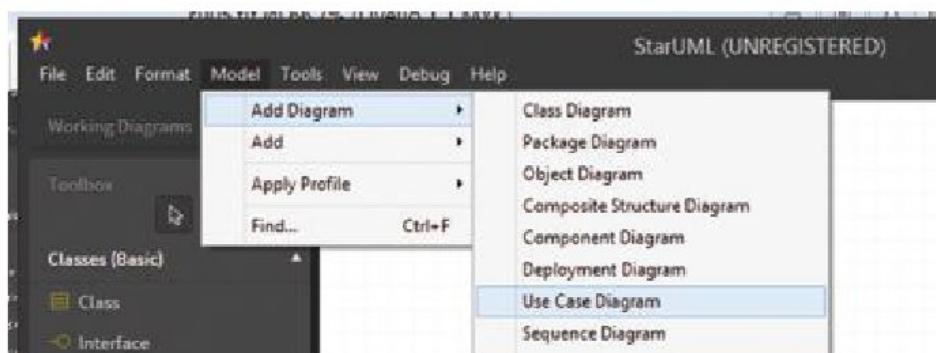
Se tutto va a buon fine viene visualizzata la seguente videata: ►



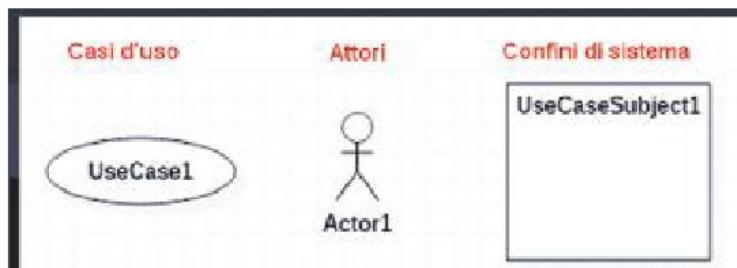
Avviamo il programma **StarUML** e selezioniamo l'opzione di valutazione del programma cliccando su **[Evalutate]**:



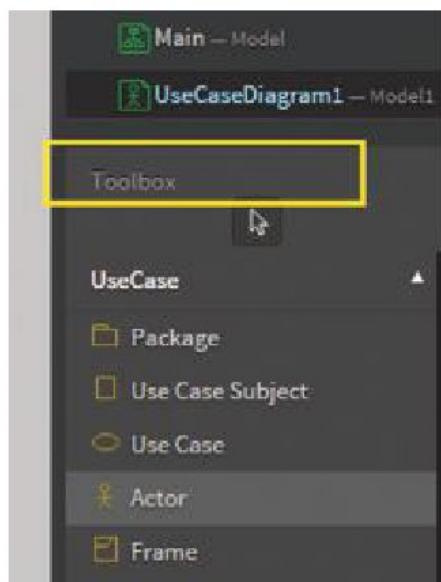
Realizziamo come primo esempio i **casi d'uso** dell'esempio del **negozi virtuale** descritto nella lezione 4: dal menu superiore **[Model]** scegliamo di creare un nuovo diagramma con **[Add Diagram]** e ne selezioniamo la tipologia, cioè **[Use Case Diagram]**:



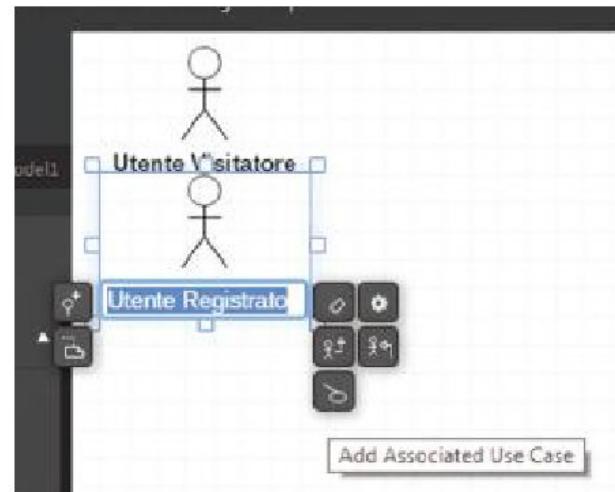
Utilizziamo i seguenti simboli grafici di **StarUML** per realizzare i nostri diagrammi:



Li troviamo nel **Toolbox Panel** di sinistra appena selezioniamo **Use Case Diagram**:



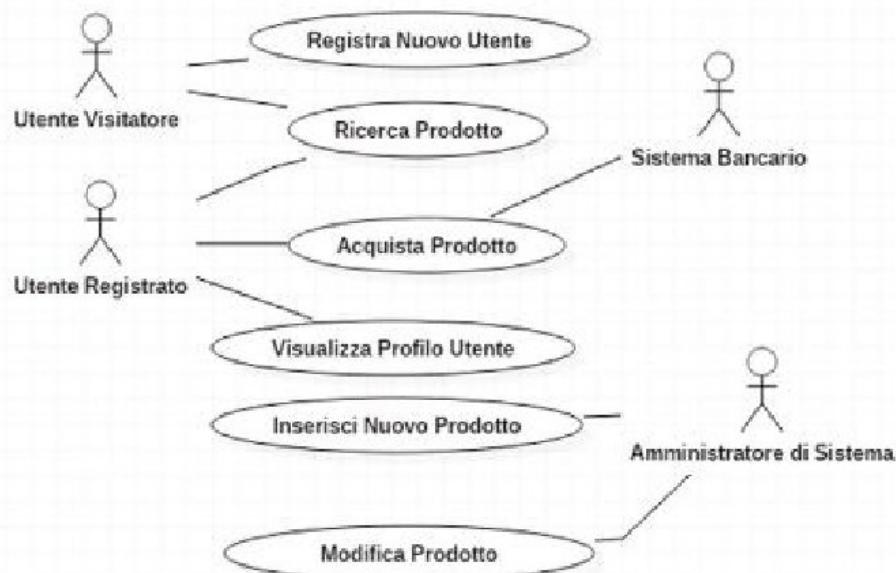
Inseriamo gli elementi nell'area di lavoro a partire dagli **attori**, selezionando l'icona sul **Toolbox Panel** e successivamente cliccando sull'area di lavoro nella posizione in cui si desiderano collocarli.



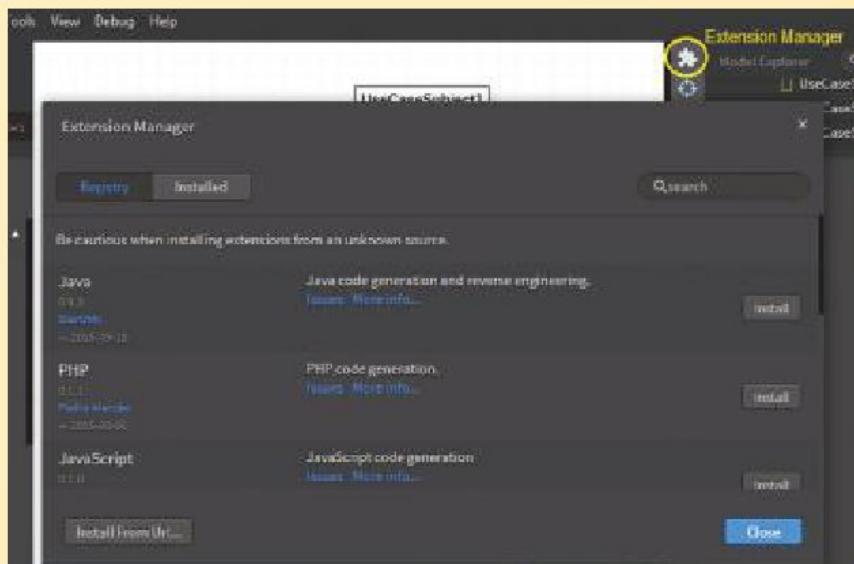
Mentre collichiamo un nuovo **Actor** ci vengono visualizzate "icone di editor" che ci permettono di introdurre gerarchie di attori oppure di aggiungere attributi o semplicemente di collegare il nostro **attore** al **caso d'uso**:



Con alcuni passaggi otteniamo le schema seguente:



Dalla versione 2.0 **StarUML** ha aggiunto la possibilità di importare prodotti di "terze parti" attivando la finestra **Extension Manager** e cliccando sull'apposita icona per generare automaticamente a partire dai diagrammi **UML** il codice sorgente in un linguaggio di programmazione:



Il pacchetto **StarUML** offre inoltre molteplici possibilità: si consiglia di approfondirne la conoscenza consultando la documentazione on-line presente ai seguenti indirizzi:

- StarUML Tutorial: <http://www.cs.sjsu.edu/faculty/pearce/modules/tutorials/uml/index.htm>
- StarUML 2: <https://github.com/staruml/staruml-dev-docs/wiki>
- StarUML 2 Blog: <http://blog.staruml.io/>
- StarUML Support: <http://staruml.io/support>



Prova adesso!

Disegna il diagramma dei casi d'uso nelle seguenti situazioni.

1 Sistema biblioteca

In una biblioteca sono stati individuati i seguenti requisiti per il bibliotecario:

- solo il bibliotecario può effettuare il prestito dei libri;
- il bibliotecario riceve i libri restituiti dagli utenti;
- effettua la gestione dei libri;
- classifica i libri nuovi, ripone i libri sugli scaffali, segnala i libri danneggiati, ordina nuovi libri.

2 Sistema ordini

In un sistema che raccoglie gli ordini di acquisto sono stati individuati i seguenti requisiti:

- ▶ i clienti registrati effettuano gli ordini;
- ▶ i clienti effettuano il pagamento comunicando i dati dell'acquisto al sistema che procede a processare l'ordine;
- ▶ se la merce è disponibile, l'ordine viene evaso, altrimenti si colloca in attesa;
- ▶ quando un fornitore consegna della merce si verifica se sono presenti ordini in attesa per procedere con l'evasione;
- ▶ si individua quali ordini possono essere evasi;
- ▶ si effettuano le assegnazioni della nuova merce agli ordini in attesa e la merce rimanente viene sistemata in magazzino.

3 Negozio di Musica

Un negozio di musica vende anche libri e riviste musicali.

Si intende automatizzare l'intero processo, dall'approvvigionamento alla vendita, a partire dalla gestione del magazzino, dove vengono richieste le seguenti funzionalità:

- ▶ la verifica della disponibilità di un prodotto (caso d'uso *VerificaDispProdotto*);
- ▶ l'effettuazione di un ordine al fornitore qualora la disponibilità non sia sufficiente (caso d'uso *EffettuaOrdineFornitore*), che richiede la seguente sequenza di attività:
 - ▶ creazione di un ordine al fornitore (inizialmente vuoto);
 - ▶ aggiunta del/dei prodotto/i da ordinare;
 - ▶ invio dell'ordine al fornitore.

4 Sistema bancario

Nella realizzazione di un sistema bancario sono stati individuati i requisiti di due funzionalità di seguito descritte; disegna il diagramma dei casi d'uso.

4.1) Apertura conto corrente

Scenario base:

- 1 il cliente si presenta in banca per aprire un nuovo c/c
- 2 l'addetto riceve il cliente e fornisce spiegazioni
- 3 il cliente accetta le condizioni e fornisce i propri dati
- 4 l'addetto verifica se il cliente è censito in anagrafica
- 5 l'addetto crea il nuovo conto corrente
- 6 l'addetto segnala il numero di conto al cliente

ESERCITAZIONI DI LABORATORIO 2

LA REALIZZAZIONE DEGLI USE CASE DIAGRAM CON ARGOUML

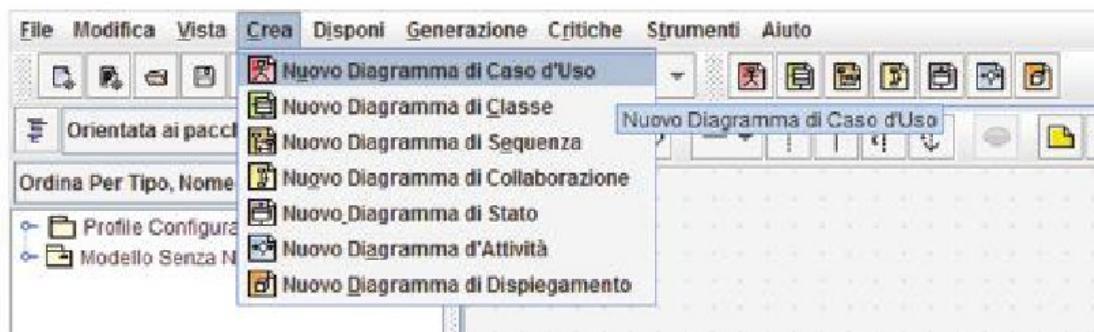
■ Premessa

Nella [esercitazione 1 di laboratorio](#) della [unità di apprendimento 5](#) del [primo volume](#) dedicato agli argomenti della classe III abbiamo utilizzato [ArgoUML](#) come software per la realizzazione dei diagrammi delle classi [UML](#).

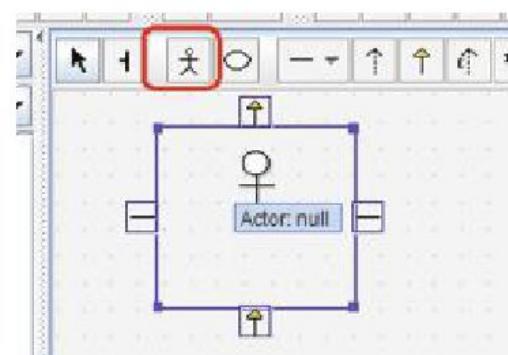
In questa esercitazione utilizzeremo [ArgoUML](#) per disegnare il diagramma dei [casi d'uso](#), cioè l'[Uses case diagram](#).

■ Il diagramma dei casi d'uso

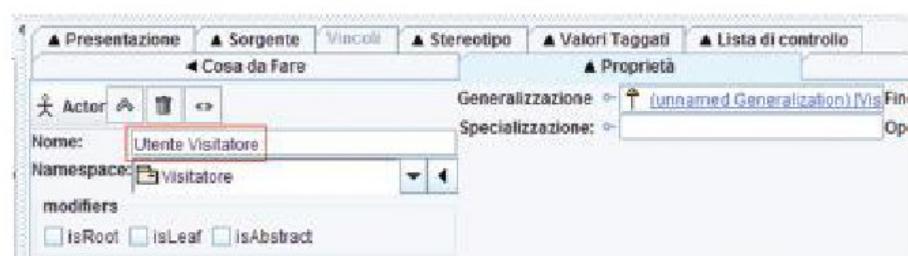
Come prima cosa selezioniamo dal menù [\[Crea\]](#) la tipologia di diagramma che si intende realizzare, e nel nostro caso proprio il primo, cioè [\[Nuovo Diagramma di Casi d'Uso\]](#).



Inseriamo nell'area di lavoro un [attore](#) selezionando l'icona dal menù superiore all'area stessa.



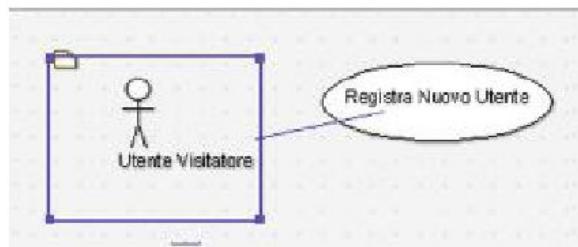
Dopo aver inserito un **Actor** gli possiamo assegnare il **Nome** selezionando la tab **[Proprietà]** della sezione inferiore dello schermo:



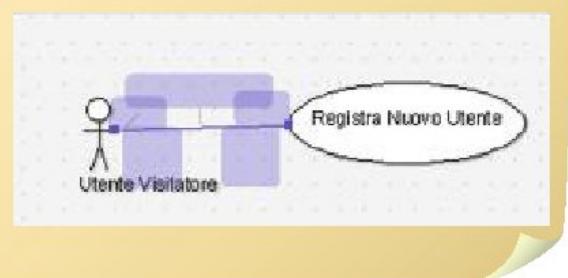
Ora inseriamo un **caso d'uso** selezionando l'icona opportuna dalla barra superiore del menù:



Selezioniamo la linea presente nella casella di sinistra per collegarlo all'attore, in modo da ottenere la seguente figura:



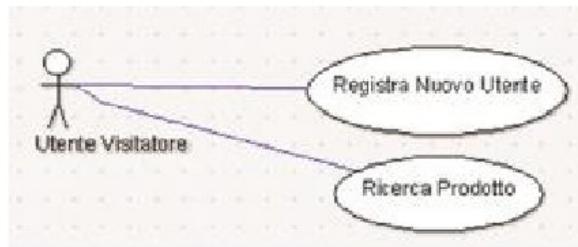
Sulla relazione è anche possibile aggiungere informazioni quali il nome e la molteplicità:



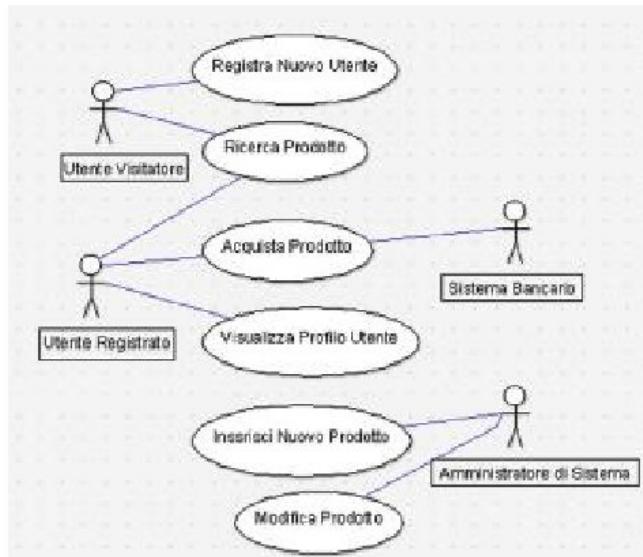
Aggiungiamo ora un secondo **caso d'uso**:



e lo colleghiamo al medesimo attore:



Con alcuni semplici passaggi si ottiene lo stesso diagramma dei **casi d'uso** della esercitazione precedente.



Prova adesso!

- Diagramma dei casi d'uso in ArgoUML

- 1 Installa il programma ArgoUML.
- 2 Disegna i diagrammi dei **casi d'uso** per tutti gli esercizi presenti alla pag. 353 della precedente esercitazione di laboratorio.
- 3 Quindi realizza il diagramma dei casi d'uso di un sistema di vendita online dei biglietti ferroviari.



4

Documentazione del software

L1 La documentazione del progetto

L2 La documentazione del codice

Esercitazioni di laboratorio

- 1 La documentazione automatica con Javadoc; 2 Il software Doxygen; 3 Il controllo delle versioni con SubVersion e TortoiseSVN

Conoscenze

- Comprendere la necessità di documentare
- Sapere quali sono i documenti necessari in un progetto
- Conoscere il concetto di documentazione interna ed esterna
- Apprendere le modalità per realizzare la documentazione esterna di sistema e utente
- Acquisire una tecnica di documentazione del codice
- Conoscere i principali tool di documentazione automatica del codice

Competenze

- Saper organizzare la documentazione del progetto
- Saper definire uno standard di documentazione
- Saper formattare il codice
- Saper effettuare la documentazione del codice

Abilità

- Utilizzare Javadoc come strumento di documentazione automatica
- Installare e utilizzare Doxygen come strumento di documentazione automatica
- Installare e configurare Subversione TortoiseSVN
- Utilizzare TortoiseSVN per effettuare il controllo delle versioni

AREA digitale



Esercizi



Soluzioni (prova adesso, esercizi, verifiche)

Puoi scaricare il file anche da

La documentazione del progetto

In questa lezione impareremo...

- ▶ la necessità di documentare
- ▶ la documentazione esterna di sistema
- ▶ la documentazione esterna utente
- ▶ i tool di documentazione del codice

■ Generalità

Documentare il software è sicuramente una attività non gradita dagli sviluppatori e spesso viene eseguita dagli stessi solo al termine dello sviluppo unicamente con lo scopo di completare il pacchetto software prima della consegna al cliente: è invece importante che venga effettuata contemporaneamente alle diverse fasi di realizzazione del progetto.

Infatti la **documentazione**, man mano che viene prodotta, diviene un supporto al **processo di sviluppo** del software e quindi deve essere redatta durante lo svolgimento del progetto stesso e non al suo termine solo al fine di completare il “corredo” del pacchetto software.

La stesura della **documentazione** deve quindi seguire il più possibile le fasi definite nel **processo di sviluppo del sistema** e deve prevedere la compilazione di un insieme di documenti in relazione temporale con il completamento delle stesse.

Lo sforzo dedicato alla **documentazione del software** non deve essere visto come un’attività costosa e noiosa, alla quale associare la minima priorità, poiché i **benefici di una buona documentazione** sono sempre tangibili: non a caso molti **fattori di qualità** sono direttamente o indirettamente influenzati in maniera positiva dalla presenza di **documentazione**.

Non esiste uno **standard** che descrive rigorosamente i documenti che devono essere prodotti: in questa lezione si è cercato di riunire tutta la serie di elaborati cartacei che dovrebbero essere prodotti durante lo sviluppo di un prodotto software di medie dimensioni, come indicato dai maggiori teorici dell’**ingegneria del software**.

■ Standard della documentazione

È importante definire un proprio **standard** nella produzione della **documentazione** da mantenere per tutti i documenti che vengono prodotti; elenchiamo di seguito un insieme di punti che devono essere definiti dal **responsabile del progetto** prima dell'inizio del progetto stesso.

A Standard per la **produzione di un documento**:

- per i documenti: descrivono la struttura, il contenuto e l'editing del documento;
- per la presentazione di un documento: definiscono i font, stili, l'uso di un logo....

ESEMPIO

Tutti i documenti dovranno contenere:

- il nome e il logo della società;
- l'elenco dei nominativi dei redattori, con le rispettive firme e la data;
- l'elenco dei nominativi di chi ha approvato il documento con le rispettive firme e la data;
- l'oggetto del contenuto ed eventualmente un sommario.

Inoltre tutte le pagine dei documenti devono essere numerate progressivamente, indicando sempre anche il totale delle pagine (esempio pag. 4 di 10).

B Standard per la **manutenzione di un documento**:

- per l'identificazione di un documento: come i documenti sono codificati per essere univocamente identificati;
- per l'aggiunta di un documento: come i documenti vanno codificati e validati; come gestire le versioni con il registro delle funzionalità apportate con i vari riferimenti, il numero e la data;
- per l'aggiornamento di documenti: definire come le modifiche di una precedente versione si riflettono in un documento, con i vari riferimenti, il numero e la data.

ESEMPIO

È possibile realizzare un layout dei documenti con ad esempio una sezione di intestazione dove inserire queste informazioni, come nell'esempio di figura:

	Progetto FATTURAZIONE PA	Mod.014
		Rev.1.0 del 12/12/2010
ORGANIGRAMMA		Pagina 1 di 14

C Standard per la **distribuzione di un documento**:

- standard per lo scambio/condivisione di documenti: come i documenti vanno memorizzati e scambiati/condivisi tra differenti sistemi di documentazione.

ESEMPIO

Per esempio, si può stabilire che per poter permettere lo scambio di documenti elettronici prodotti usando differenti sistemi e computer, sia in modo diretto che per l'inoltro per mezzo di e-mail, il loro sia lo standard **XML** (questa motivazione viene anche dal fatto che i documenti devono essere archiviati e la vita di un sistema di word processing potrebbe essere più breve di quella del software che si sta documentando).

Nella documentazione un aspetto fondamentale è quello di permettere la **tracciabilità** delle modifiche apportate al software o alla architettura del sistema: se è agevole risalire a tutta la "cronologia" che ha portato a un modifica o alla realizzazione di codice si riduce notevolmente lo sforzo necessario per comprenderlo.

■ Documentazione del progetto

La bontà di un progetto viene valutata anche in base alla documentazione acclusa: occorre pertanto che la documentazione sia redatta in forma standard, completa e comprensibile e, a parità di contenuto e di chiarezza, è preferibile che la documentazione sia il più possibile concisa perché un testo lungo non invoglia la lettura.

La **documentazione del progetto** si compone di tre parti:

- il manuale per l'utente;
- la documentazione tecnica (che comprende anche il codice sorgente);
- le prove di collaudo.

Possiamo classificare la documentazione di un progetto secondo quattro modalità:

A esterna o interna:

- **esterna:** separata dal programma vero e proprio;
- **interna:** sotto forma di commenti o help contenuti nel file sorgente;

B in linea o fuori linea:

- **in linea:** richiamabile su richiesta durante l'esecuzione del programma,
- **fuori linea:** sotto forma di manuale consultabile separatamente;

C globale o locale:

- **globale:** riguardante il programma nel suo complesso;
- **locale:** sotto forma di commenti inseriti nel codice sorgente in punti specifici del programma;

D per l'utente o per il programmatore:

- **rivolta all'utente:** per chi usa il programma, generalmente organizzata in:
 - manuale d'uso;
 - manuale di installazione;
 - help in linea;
- **rivolta alla programmazione:** è specifica per il personale tecnico interessato alla struttura del programma per lo sviluppo o per la sua manutenzione, e comprende
 - documentazione inherente il progetto e il management del progetto;
 - documentazione del codice.

La documentazione di un sistema quindi non riguarda soltanto il codice sorgente che costituisce il programma vero e proprio ma anche:

- le specifiche funzionali e non funzionali del sistema;
- le scelte di progetto riguardanti l'architettura generale, le strutture degli archivi e dei dati in memoria centrale e i principali algoritmi utilizzati;
- le modalità d'uso del programma;
- i collaudi effettuati e i loro risultati.

In questa lezione descriviamo la **documentazione esterna**, mentre la **documentazione interna** sarà trattata nella prossima lezione.

■ La documentazione esterna

Dopo l'approvazione del preventivo si avvia la realizzazione di un sistema software e viene nominato un direttore generale responsabile del progetto (**project manager**).

La prima attività del **project manager** è proprio la stesura dell'elenco dei documenti necessari allo svolgimento del progetto che devono tenere traccia scritta di tutte le attività

inerenti al progetto stesso. Questa documentazione deve consentire alla direzione di pianificare e controllare lo svolgimento del progetto ma anche di analizzare a posteriori ciò che è accaduto, in un'ottica di miglioramento sia dell'organizzazione che del processo di produzione del software. Possiamo individuare due gruppi di documenti:

1 documentazione inerente il management del progetto:

- ▷ organigramma
- ▷ diario di progetto
- ▷ verbale
- ▷ piano di progetto
- ▷ norme di progetto
- ▷ offerta
- ▷ contratto per lo sviluppo, la fornitura del software e l'assistenza all'avviamento
- ▷ piano di gestione della qualità
- ▷ relazione finale

2 documentazione del progetto:

- ▷ analisi del dominio
- ▷ analisi dei requisiti (documento SRS)
- ▷ specifica architetturale
- ▷ specifica di dettaglio
- ▷ piano delle prove
- ▷ risultati delle prove
- ▷ manuale d'uso

Ogni documento redatto dal gruppo di progetto deve comparire nel diario di progetto che il project manager mantiene costantemente aggiornato anche perché le eventuali verifiche ispettive terranno conto solo ed esclusivamente dei documenti elencati nel diario di progetto.

Descriviamoli succintamente uno per uno.

Documentazione inerente il management del progetto

Organigramma

È un documento che contiene l'elenco dei membri del gruppo di progetto con la descrizione dei ruoli assegnati a ciascuno di essi e viene redatto dal project manager prima dell'inizio delle attività. Ogni ruolo viene descritto indicando le competenze di ciascun membro del progetto ed è soggetto a modifiche dato che nel corso dell'evoluzione del progetto l'assegnazione di qualche ruolo potrebbe essere rivista.

Viene sottoscritto da tutti i membri del team in modo che ciascuno conosca i propri incarichi. Non è soggetto a numerazione di versioni in quanto è a uso solo del **project manager**: in casi di variazione dell'organigramma può essere sostituito con un nuovo documento.

Diario di progetto

Rappresenta il **registro ufficiale** della documentazione del progetto e contiene l'elenco di tutti i movimenti relativi all'archivio dei documenti del progetto e del software che si sta sviluppando. Ogni movimento viene registrato indicando la data in cui avviene, una descrizione comprendente il suo riferimento univoco (codifica) e la versione del documento (o del prodotto), il nominativo e il ruolo del destinatario o del ricevente.

Tutti i documenti entranti nell'archivio devono essere approvati dal project manager.

Verbale

Tutte le riunioni del gruppo devono essere verbalizzate, sia che vengano fatte solo dal gruppo di sviluppo sia che prevedano la presenza del committente e/o degli stakeholder.

Ogni riunione viene convocata specificando precedentemente l'ordine del giorno e comunicandolo a tutte le persone che devono parteciparvi: lo svolgimento di una riunione segue tale ordine e nel verbale, dopo l'indicazione della data, del luogo in cui si è tenuta la riunione e dell'elenco delle persone presenti, si riporta sinteticamente quanto emerso dalla discussione indicando nominalmente gli interventi in modo da poterli tracciare.

Al termine della riunione viene redatto un verbale che riporti il nome del compilatore e che, dopo una rilettura, viene fatto firmare da tutti i presenti.

È buona norma aggiungere come ultimo punto dell'ordine del giorno la voce "Varie ed eventuali" in modo da poter affrontare anche la discussione di esigenze sopravvenute successivamente alla convocazione della riunione.

Piano di progetto

Il **project manager** deve compilare e tenere aggiornato il piano del progetto che contiene le attività pianificate, in particolare:

- 1 la definizione degli obiettivi;
- 2 l'analisi dei rischi;
 - ▷ piano gestione rischi;
 - ▷ sintesi rischi individuati;
 - ▷ strategie di prevenzione;
- 3 descrizione del modello di processo di sviluppo e delle singole fasi (**milestones**);
- 4 stima dei costi;
- 5 attività di progetto (Work Breakdown Structure, Diagramma di Gantt);
- 6 consuntivo attività;
- 7 strumenti utilizzati.

Il piano di progetto si divide in due parti, **pianificazione**, che serve per stimare realisticamente le risorse, i costi e i tempi necessari alla realizzazione del progetto e **consuntivazione**, per confrontare e avere un riscontro tangibile tra le attività effettuate e quelle previste, individuare i punti di discordanza e rendere possibile la pianificazione delle attività future.

Il piano di progetto è corredata da un documento che tiene conto, nel corso del tempo, delle variazioni che sono state apportate, il **registro delle modifiche**, indicando la causa, la data e un numero progressivo, che indica la versione del piano e lo identifica univocamente.

Norme di progetto

Il **project manager** ha anche il compito di redarre un documento che contiene le norme che devono essere rispettate durante le svolgimenti del progetto; è generalmente strutturato in due parti:

- 1 **Convenzioni generali**
 - ▷ **documentazione del progetto**: convenzioni utilizzate per stilarla, per identificarla, per archiviarla, diffonderla, approvarne i contenuti e integrarli.
 - ▷ **comunicazione**: come viene organizzata la comunicazione tra i membri del gruppo di sviluppo e verso l'esterno, le regole di utilizzo della posta elettronica, le modalità di comunicazione e di gestione dei problemi.
 - ▷ **organizzazione dello spazio di lavoro**: come organizzare i file di progetto, dove archiviarli, come viene strutturata la directory e le varie cartelle, la modalità di accesso e condivisione, l'indirizzo IP o URL del server che conterrà i codici sorgenti e la documentazione, la modalità di aggiornamento e integrazione dei file e dei documenti ecc.

► **uso degli strumenti:** elenco degli strumenti utilizzati, delle piattaforme di sviluppo, dei compilatori, delle convenzioni usate per l'analisi, la progettazione, la codifica e la verifica del prodotto, e inoltre per la gestione delle versioni.

2 Norme di sviluppo

► **norme di analisi:** norme che riguardano le attività inerenti lo svolgimento delle attività di analisi;

► **norme di progettazione:** modalità per lo svolgimento delle attività di progettazione e descrizione delle convenzioni adottate per la stesura della documentazione (per esempio, il formato dei diagrammi UML, le modalità di specifica dei singoli componenti ecc).

► **norme di codifica:** sono previste singole sezioni dedicate ai diversi linguaggi di programmazione usati nel progetto, dove elencare le convenzioni e gli standard di codifica (regole per l'indentazione, il naming delle variabili, l'intestazione dei file, le modalità di utilizzo degli strumenti di documentazione automatica utilizzati: per esempio [Javadoc](#) o [Doxygen](#) descritti in seguito).

Offerta

L'**offerta** (o **preventivo**) è il documento che viene sottoposto al cliente prima dell'inizio dello sviluppo del sistema e deve essere firmato dallo stesso per accettazione: viene stilata generalmente dal **project manager** e da un commerciale e comprende una breve descrizione del prodotto offerto, i dettagli economici, l'analisi dei requisiti e i tempi di consegna con l'indicazione delle eventuali penali.

Contratto per lo sviluppo, la fornitura del software e l'assistenza all'avviamento

Al momento della firma per accettazione dell'offerta viene redatto il **contratto per lo sviluppo, la fornitura del software e l'assistenza all'avviamento** che, oltre alle Condizioni Generali e agli elementi essenziali del contratto, riporta le clausole accidentali raccomandate:

- A** limiti delle responsabilità della software house (oltre a quanto previsto dalla legge);
- B** modalità di utilizzo corretto del software prodotto;
- C** proprietà dei materiali (software) e degli elaborati;
- D** tipo di fornitura del codice (sorgente e/o oggetto);
- E** criteri, modalità, obiettivi del collaudo;
- F** termini di consegna;
- G** eventuali penali;
- H** periodo di prova del software a decorrere dalla data di collaudo;
- I** tipo di assistenza nel periodo di avviamento;
- L** attività del cliente che costituiranno parte delle obbligazioni contrattuali del cliente:
 - 1** indicazione del responsabile del progetto che è referente per il cliente;
 - 2** definizione completa dei requisiti di progetto (specifiche di sistema, specifiche software);
 - 3** partecipazione al collaudo e approvazione;
 - 4** attività di installazione e configurazione sistema (ambiente di produzione e ambiente di test) se non facenti parte esplicitamente delle attività di progetto del professionista.

Piano di gestione della qualità

A seconda delle diverse certificazioni di qualità possedute dalla azienda svilupatrice deve essere definito il piano della qualità comprendente le politiche e gli obiettivi prefissi, in esso devono essere indicati gli strumenti e le procedure di controllo indicando tempi, tecniche, metodi (Management reviews, Technical reviews, Inspections, Audits), azioni da intraprendere in caso di non conformità e anomalie.

Relazione finale

Alla conclusione dello sviluppo e del collaudo del sistema viene redatta dal **project manager** la **relazione finale**, che contiene la descrizione delle funzionalità implementate, dell'architettura del software e dell'hardware dove è stato installato e collaudato il sistema, le informazioni sui linguaggi di programmazione e sull'ambiente di sviluppo impiegati, la descrizione delle strutture dei dati realizzate, dell'interfaccia software per l'utente e delle risorse utilizzate dal progetto, la documentazione del collaudo fatto in presenza del committente e da lui sottoscritto come approvazione della fornitura.

Alla relazione finale viene generalmente allegato il manuale operativo per il corretto utilizzo del programma e dei dati da esso generati e viene proposto al cliente un **contratto di manutenzione** del software per specificare gli accordi economici sullo sviluppo successivo per le migliorie, per lo sviluppo di funzionalità per esigenze tecnico/organizzative al momento non previste e gli eventuali adeguamenti legislativi e un **contratto di assistenza**, che può essere per esempio costituito da un "pacchetto" di giornate di assistenza oltre che alla assistenza telefonica o fatta mediante accesso remoto.

Documentazione del progetto

Analisi del dominio

Nell'analisi del dominio sono presenti i dati necessari a conoscere l'ambito di sviluppo del prodotto, una breve descrizione della azienda e del progetto da realizzare, la descrizione dei compiti e delle procedure e una analisi dettagliata del dominio, a partire dalla definizione del glossario contenente le definizioni, gli acronimi e le abbreviazioni d'uso comune nel settore d'intervento, le caratteristiche dei clienti e degli utenti, la descrizione della concorrenza e dei prodotti software competitori da loro utilizzati.

Analisi dei requisiti

Il documento di **Specifiche dei Requisiti Software (SRS)** è stato descritto nella lezione 4 dell'unità di apprendimento 3 e descrive quello che è richiesto allo sviluppatore del sistema come risultato del lavoro di analisi tra il cliente, l'utente e lo sviluppatore.

Specifiche architettoniche

In questo documento viene descritta l'**architettura del sistema**, la sua decomposizione in moduli funzionali con la definizione delle informazioni che vengono scambiate tra di essi e con l'ambiente esterno, la descrizione dell'articolazione del programma in sotto programmi con particolare riguardo all'albero di chiamata dei sotto programmi stessi. Nel caso di programmazione a oggetti vengono descritte le singole classi, le gerarchie, la modellizzazione dinamica e comportamentale dei singoli componenti. Vengono inoltre descritte le interfacce utente e i dati presenti in ciascuna di esse.

Specifiche di dettaglio

Nella specifica di dettaglio si riprendono i **componenti software** (modulo, funzioni o classi) identificati nel disegno e vengono descritte le loro specifiche funzionalità, l'algoritmo utilizzato, il significato delle variabili passate come parametri, le eventuali variabili globali utilizzate, le variabili locali di cui fa uso e quelle di particolare significatività e i singoli casi d'uso. Viene quindi allegato il codice sorgente e il modello logico del database.

Piano delle prove

Definiamo dapprima con **batteria di prove** una sequenza di singoli casi di prova correlati che prevede la verifica di una funzionalità del sistema in corrispondenza di determinati input, che deve essere così documentata:

- ▶ **classificazione della prova:** identificativo, titolo, elenco delle componenti coinvolte nella verifica, descrizione degli obiettivi in termini di funzionalità, livello d'importanza della prova.
- ▶ **istruzioni per il verificatore:** descrizione dei dati in ingresso, singoli o di sequenze particolari;
- ▶ **risultati attesi:** risultati attesi e/o comportamenti in casi estremi;
- ▶ **istruzioni di ripristino:** modalità di recovery in caso di fallimento della prova.

Il piano delle prove è un documento che può essere articolato in due sezioni:

- ▶ **definizione delle prove di integrazione:** generalmente il sistema deve interagire con altri preesistenti, sia interni che esterni, e quindi devono essere pianificate le **batterie di prove** e descritta la strategia di integrazione; devono essere descritti i **driver** e gli **stub** utilizzati per le prove di modulo e per le prove condotte a ogni passo d'integrazione, eventualmente allegando i codici sorgenti;
- ▶ **definizione delle prove di collaudo** (del sistema completo): dopo aver verificato singolarmente le unità funzionali del sistema è necessario pianificare le batterie di prove per la verifica delle funzionalità nel suo complesso e il soddisfacimento dei requisiti software prefissati.

Il collaudo termina con la stesura del documento di accettazione nel quale il cliente sottoscrive il suo soddisfacimento per il prodotto che gli è stato consegnato.

Risultati delle prove

Per ogni caso/batteria di prove viene prodotto un report da parte del verificatore che riporta la data del test, il tipo di prova identificato con il riferimento univoco, i dati di prova utilizzati e l'esito, commentando eventuali risultati inattesi e situazioni indesiderate ed eventualmente fornendo le indicazioni per una eventuale ripetizione della prova.

Manuale d'uso (o manuale utente)

Il manuale utente ha lo scopo di mettere l'utente in grado di utilizzare il programma senza problemi e deve essere redatto in un linguaggio semplice, di facile comprensione per ciascun utente del sistema, tenendo conto che sicuramente ci saranno persone senza conoscenze informatiche approfondite. La documentazione utente comprende:

- ▶ **documentazione cartacea:**
 - manuale d'uso;
 - manuale di installazione, se il prodotto viene pacchettizzato e installato dal cliente;
- ▶ **documentazione elettronica:**
 - help on-line;
 - eventuale sito di riferimento.

Disegni, schemi e sequenze guidate nella descrizione delle operazioni agevolano la loro comprensione: più la documentazione è chiara ed esaustiva e meno gli utenti telefoneranno per richiedere assistenza operativa!

Sia il formato cartaceo che quello elettronico devono contenere le sezioni seguenti:

- ▶ **introduzione:** dopo una breve descrizione dell'utilizzo del manuale vengono descritti i servizi forniti dal programma e l'ambiente hardware/software in cui il sistema funziona (tipo di calcolatore, memoria di massa e di lavoro necessarie, tipo e versione del sistema operativo, tipo e versione del linguaggio di programmazione utilizzato) e una descrizione delle modalità di avvio del programma;
- ▶ **istruzioni d'uso:** la descrizione di ogni singola funzione e di ogni singola voce del menu (se esiste), possibilmente integrata con un esempio guidato dove vengono elencate le singole azioni richieste (e permesse) all'utente; per ogni caso devono essere elencate le possibili situazioni di errore con l'elenco delle cause e l'indicazione dei possibili interventi necessari alla loro risoluzione;
- ▶ **appendice:** viene riportato il glossario, l'elenco dei messaggi di errore comuni a tutto il sistema, una sezione di come risolvere le principali situazioni critiche.

■ Tool di documentazione

Per aiutare gli sviluppatori a produrre la documentazione del codice sono presenti **tool di documentazione automatica** che analizzano il codice individuando appositi contrassegni (tag) predisposti dal programmatore. Come vedremo nella prossima lezione, l'aggiunta di commenti all'interno del codice è una metodologia di documentazione interna che viene effettuata per facilitarne la lettura, per spiegarne il funzionamento e per favorire la collaborazione tra diversi sviluppatori: utilizzando una notazione specifica per questi commenti si ottiene il doppio beneficio di commentare il codice predisponendo gli elementi necessari per poi ottenere automaticamente la generazione della documentazione dei programmi sorgenti.

È però necessario porre molta attenzione per rispettare gli standard previsti dal generatore automatico che si intende utilizzare, sia nella forma che nella completezza necessaria per produrre un documento esaustivo, chiaro e ben dettagliato.

È consigliabile produrre questa documentazione durante lo sviluppo, anche perché la persona più indicata per descrivere un programma è proprio chi lo sta scrivendo: in questo modo il documento viene costantemente aggiornato, in linea con l'evoluzione del software e risulta pronto "contemporaneamente" col codice da collaudare.

Può quindi essere utilizzato dai collaudatori durante le operazioni di test, permettendo di risparmiare notevoli risorse, sia in termini di tempo che naturalmente di denaro.

Nessun programmatore ama scrivere relazioni e documenti: avere uno strumento di generazione automatica delle documentazioni del codice lo libera da un compito "noioso" e poco gratificante e gli permette di utilizzare "meglio" il proprio tempo.

Alcuni tool permettono di personalizzare lo "strumento di generazione automatica" arricchendo l'insieme dei tag in modo da ottenere una documentazione più approfondita, completa e aderente ai fabbisogni del team di sviluppo: è infatti possibile aggiungere elementi specifici per i collaudatori affinché possano effettuare le varie fasi di test con riferimenti a esempi concreti.

Tool esistenti

I principali tool appositamente sviluppati per effettuare la generazione automatica della documentazione sono **Javadoc** e **Doxygen**.

► **Javadoc** è stato sviluppato direttamente dalla **Sun Microsystems** nel 1990 durante lo sviluppo del linguaggio Java per documentare proprio la sua progettazione e venne distribuito insieme al kit di sviluppo per la documentazione di progetti scritti in questo linguaggio. Col passare degli anni il tool è cresciuto assieme al linguaggio, divenendo sempre più ricco e completo, e aggiungendo la possibilità per l'utilizzatore di personalizzare il contenuto e il formato di output della documentazione.

► **Doxygen** è un sistema multipiattaforma per la generazione di documentazione tecnica di un qualsiasi prodotto software e fu sviluppato come progetto *Open Source* a partire dal 1997 su proposta di **Dimitri van Heesch**.

A differenza di **Javadoc** utilizzabile solo per Java, **Doxygen** tool può essere utilizzato con codice sorgente scritto in diversi linguaggi: C++, C, Java, Objective C, Python, IDL, PHP e C#.

Sia **Doxygen** che **Javadoc** analizzano il codice sorgente ed estraggono informazioni dalla dichiarazione di strutture dati e da commenti scritti con una particolare sintassi, generando un ipertesto o un formato **HTML** oppure anche un formato pdf: li descriveremo entrambi.

Verifichiamo le conoscenze



1. Risposta multipla

- 1 Quale tra i seguenti non è uno standard utilizzato nella produzione della documentazione?**
 - a) standard per la produzione di un documento
 - c) standard per la pubblicazione di un documento
 - b) standard per la manutenzione di un documento
 - d) standard per la distribuzione di un documento

- 2 Quale tra i seguenti non è uno standard di manutenzione di un documento?**
 - a) l'identificazione di un documento
 - c) l'aggiunta di un documento
 - b) l'editing di un documento
 - d) l'aggiornamento di documenti

- 3 La documentazione di un sistema riguarda anche (indicare quello inesatto):**
 - a) le specifiche funzionali e non funzionali del sistema
 - b) le scelte di progetto riguardanti l'architettura generale
 - c) lo studio di fattibilità
 - d) le strutture degli archivi e dei dati in memoria centrale e i principali algoritmi utilizzati
 - e) le modalità d'uso del programma
 - f) i collaudi effettuati e i loro risultati

- 4 Quale tra i seguenti documenti non appartiene alla documentazione del management del progetto?**

a) organigramma	f) offerta
b) diario di progetto	g) contratto per lo sviluppo, la fornitura del software e l'assistenza all'avviamento
c) verbale	h) piano di gestione della qualità
d) analisi del dominio	i) relazione finale
e) norme di progetto	

- 5 Quale tra i seguenti documenti non appartiene alla documentazione del progetto?**

a) analisi dei requisiti (documento SRS)	e) piano delle prove
b) specifica architettonica	f) risultati delle prove
c) specifica di dettaglio	g) manuale d'uso
d) piano di progetto	

- 6 Il piano del progetto contiene le attività pianificate, in particolare (indicare quello non presente):**
 - a) la definizione degli obiettivi
 - b) l'analisi dei rischi
 - c) la descrizione del modello di processo di sviluppo e delle singole fasi
 - d) la stima dei costi
 - e) l'attività di progetto (Work Breakdown Structure, Diagramma di Gantt)
 - f) il piano dei collaudi
 - g) il consuntivo attività
 - h) gli strumenti utilizzati



2. Vero o falso

- 1 La documentazione è un supporto al processo di sviluppo del software.**
- 2 La documentazione quindi deve essere redatta durante lo svolgimento del progetto.**
- 3 La documentazione di un sistema riguarda soltanto il codice sorgente.**
- 4 L'offerta rientra tra la documentazione inherente il management del progetto.**

V F
V F
V F
V F

- 5** Il piano di gestione della qualità non è tra i documenti del management del progetto.
- 6** Tra le norme di progetto rientra l'elenco degli strumenti utilizzati.
- 7** Nel contratto per lo sviluppo sono presenti i tempi di consegna e le eventuali penali.
- 8** Col contratto per lo sviluppo viene proposto al cliente un contratto di manutenzione.
- 9** Alla relazione finale viene generalmente allegato il manuale operativo.
- 10** Nel manuale utente sono presenti gli esiti delle prove delle funzionalità del prodotto.

V	F
V	F
V	F
V	F
V	F
V	F

3. Completamento

per l'utente o per il programmatore • in linea o fuori linea • globale o locale • le prove di collaudo • il manuale per l'utente • la documentazione tecnica • in linea • locale • globale • fuori linea • rivolta all'utente • esterna o interna • interna • esterna • rivolta alla programmazione • i loro risultati • funzionali • manuale d'uso • help in linea • manuale d'installazione • non funzionali • dei dati • l'architettura generale • algoritmi utilizzati • degli archivi

- 1** La documentazione del progetto si compone di tre parti:

- a)
- b)
- c)

- 2** Possiamo classificare la documentazione di un progetto secondo quattro modalità:

- a) o :
 - : separata dal programma vero e proprio;
 - : sotto forma di commenti o help contenuti nel file sorgente;
- b) o :
 - : richiamabile su richiesta durante l'esecuzione del programma,
 - : sotto forma di manuale consultabile separatamente;
- c) o :
 - : cioè riguardante il programma nel suo complesso;
 - : sotto forma di commenti inseriti nel codice sorgente in punti specifici del programma;
- d) o :
 - : per chi usa il programma
 - : è specifica per il personale tecnico interessato alla struttura del programma per lo sviluppo o per la sua manutenzione

- 3** La documentazione rivolta all'utente è generalmente organizzata in:

- a)
- b)
- c)

- 4** La documentazione di un sistema comprende anche:

- a) documentazione :
 - manuale ;
 - manuale ;
- b) documentazione :
 - ;
 - eventuale

ESERCITAZIONI DI LABORATORIO 1

LA DOCUMENTAZIONE AUTOMATICA CON JAVADOC

Javadoc è stato realizzato direttamente dalla Sun Microsystems nel 1990, durante lo sviluppo del linguaggio Java, per documentare proprio la sua progettazione e, alla fine di essa, venne distribuito insieme al kit di sviluppo per la documentazione di progetti scritti in questo linguaggio. Nel passare degli anni il tool è cresciuto assieme al linguaggio, divenendo sempre più ricco e completo, e aggiungendo la possibilità all'utilizzatore di personalizzare il contenuto e il formato di output della documentazione.

■ Generazione Automatica di Documentazione Tecnica

Javadoc è uno strumento che permette di documentare i sorgenti di un programma all'interno dei sorgenti stessi: anziché scrivere la documentazione di un programma in un file separato, il programmatore inserisce nel codice sorgente dei commenti con una particolare sintassi, che verrà mostrata in seguito.

Tali commenti vengono estratti dal programma Javadoc che li converte in un formato più semplice per la consultazione (generalmente in formato HTML); viene effettuato un parsing dei file sorgenti, ovvero un'analisi di tale codice, tramite il compilatore Java.

Il Javadoc può essere eseguito su interi package, su singoli file o su entrambi.

Nel caso di documentazione di un package è possibile memorizzare in un'apposita cartella (una sottocartella che deve avere il nome "doc-files") altri file che possono essere importati nella documentazione, come immagini, esempi di codice o altri file HTML.

Avviando l'esecuzione del **parser** del codice di Javadoc, viene analizzato il file sorgente, individuati i tag specifici che identificano i commenti e generato un ipertesto HTML di una o più pagine che descrivono package, classi, interfacce, costruttori, metodi e attributi.

Parser A natural language parser is a program that works out the grammatical structure of sentences, for instance, which groups of words go together (as "phrases") and which words are the subject or object of a verb. ►



Nella documentazione sono presenti sezioni riepilogative, pagine di indice, l'elenco dei metodi e altri strumenti per facilitarne la consultazione.

Il processo di generazione di **Javadoc** consiste in tre semplici fasi:

- ▶ l'inserimento dei commenti di spiegazione del codice sorgente che il programmatore esegue manualmente contemporaneamente alla scrittura dello stesso;
- ▶ la predisposizione di eventuali file da associare al documento, inserendo gli opportuni comandi nelle sezioni dove vuole che questi vengano inseriti;
- ▶ la generazione vera e propria dell'ipertesto, tramite il comando **Javadoc**.

ESEMPIO

Per generare la documentazione della classe **Prova.java** all'interno della cartella è sufficiente digitare al prompt dei comandi la seguente istruzione:

```
javadoc Prova.java -d doc
```

Sono molteplici le possibilità offerte come opzione per la compilazione della documentazione che, oltre che sul manuale di riferimento presente sul sito della **SUN**, sono ottenibili direttamente dalla linea di comando digitando semplicemente **Javadoc**.

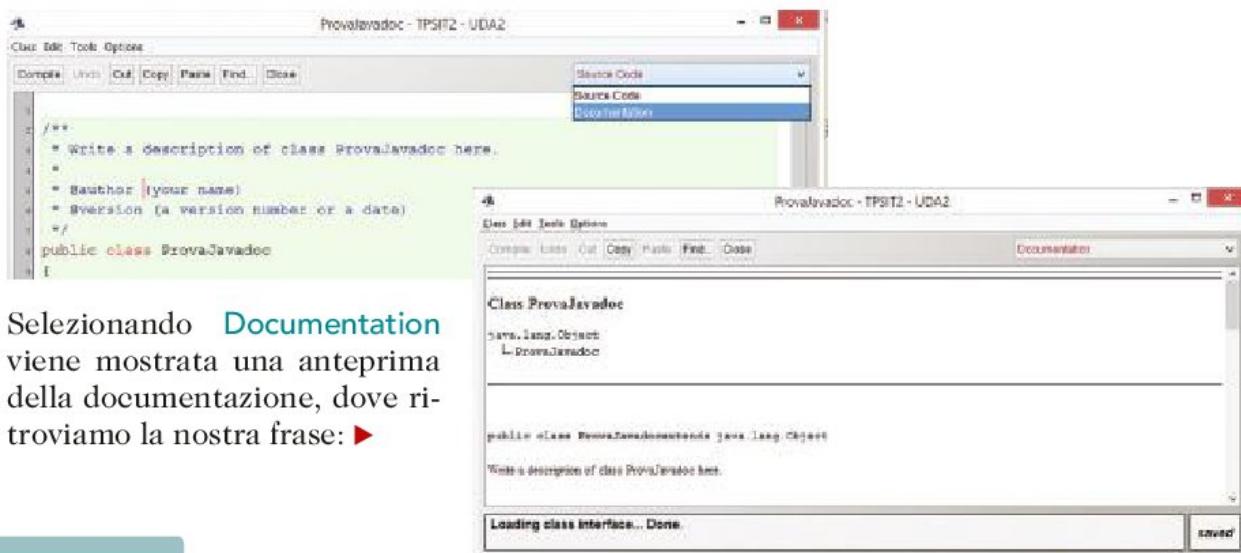
Il generatore di documentazione è stato sviluppato specificamente per Java, ma con alcuni semplici accorgimenti e adattamenti è possibile utilizzarlo anche per la documentazione di altri linguaggi di programmazione, come per esempio dei sorgenti **C++**.

Realizzazione di un Javadoc

Vediamo la sintassi dei più comuni comandi: in generale un commento **Javadoc** è un testo **HTML** racchiuso tra i tag `/**` e `*/`.

Scriviamo un esempio di classe Java utilizzando come editor **Bluej** che è particolarmente comodo per apprendere **Javadoc** in quanto permette di avere una anteprima della documentazione direttamente dalla finestra di editor del codice sorgente.

ESEMPIO



Selezionando **Documentation** viene mostrata una anteprima della documentazione, dove ritroviamo la nostra frase: ▶

La documentazione **minima** dovrebbe comprendere la descrizione di ciascun **package**, **classe**, **interfaccia**, **metodo pubblico**, **attributo pubblico** e quindi va inserito un commento prima di ciascuno di essi.

All'interno del commento **Javadoc** abbiamo quindi tre possibili situazioni di commento **Javadoc**: ►

e nell'ultimo è possibile inserire uno o più tag, che iniziano con il simbolo **@**; riportiamo di seguito i tag più utilizzati.

```
Monosezione - monolinea :
/** descrizione */

Monosezione - multilinea :
/*
 * descrizione
 */

Multisezione - multilinea :
/**
 * descrizione
 * @tag1 commento per il tag1
 * @tag2 commento per il tag2
 */
```

Tags per documentazione di classi

@author [nome]

Aggiunge “Author:” seguito dal nome specificato: è possibile inserire più autori, presentati in ordine cronologico.

@version [versione]

Aggiunge “Version:” seguita dalla versione specificata.

@see [riferimento]

Aggiunge “See Also” seguito dal riferimento indicato.

Tags per documentazione di metodi

@param [nome del parametro] [descrizione]

Aggiunge il parametro specificato e la sua descrizione alla sezione “Parameters:” del metodo corrente: per ogni parametro va inserito un tag rispettando lo stesso ordine del metodo.

@returns [descrizione]

Aggiunge “Returns:” seguito dalla descrizione specificata indica il tipo restituito e l'insieme dei valori possibili. Non previsto per costruttori e per metodi che non restituiscono alcun valore (void).

@throws [nome completo della classe][descrizione]

Aggiunge “Throws:” seguito dal nome della classe specificata (che costituisce l'eccezione) e dalla sua descrizione. Sono ammessi più tag **@exception** per ognuna delle eccezioni che compaiono nella sua clausola throws, presentate in ordine alfabetico.

ESEMPIO

Inseriamo nel nostro esempio questi tag per commentare un metodo costruttore:

```
/**
 * Questo è un commento <em>Javadoc</em>.
 * Gli spazi e gli asterischi a inizio riga
 * <strong>sono</strong> sempre ignorati.
 * @author Paolo Riccardo
 * @version 1.0 del 25/12/2012
 * @see http://www.java.com/en/java_in_action/bluej.jsp
 * @since java.7.0
 */
```

```
public class ProvaJavadoc
{
    /** unico attributo: anni per imparare il linguaggio Java */
    private int anni;

    /**
     * Costruttore per gli oggetti della classe ProvaJavadoc
     * @param numero numero di anni di studio
     */
    public ProvaJavadoc(int numero)
    {
        // inizializza l'attributo della classe
        anni = numero;
    }
}
```

e visualizziamo ora il documento generato:

Since:	java 7.0
Version:	1.0 dal 25/12/2012
Author:	Paolo, Riccardo
See Also:	http://www.java.com/en/java_in_action/bluej.jsp
Constructor Summary	
ProvaJavadoc(int numero)	Costruttore per gli oggetti della classe ProvaJavadoc

Oltre ai dati dell'intestazione abbiamo l'iperclick al costruttore, che viene riportato in una apposita sezione con il commento da noi inserito: cliccando su di esso veniamo "portati" a una sezione di dettagli del metodo costruttore:

ESEMPIO

Constructor Detail	
ProvaJavadoc	
public ProvaJavadoc (int numero)	
Costruttore per gli oggetti della classe ProvaJavadoc	
Parameters:	
numero - numero di anni di studio	

Aggiungiamo un semplice metodo che somma al nostro attributo un valore numerico e otteniamo un reference nella sezione dedicata ai metodi:

Method Summary	
int	summa(int y)
un semplice esempio di metodo: incrementa l'attributo della classe	

e il dettaglio cliccando sul link.

Method Detail

```
somma
public int somma(int y)
    un semplice esempio di metodo: incrementa l'attributo della classe

    Parameters:
        y - il valore da sommare
    Returns:
        la somma degli anni e del valore passato come parametro
```

Nella tabella seguente sono riportati tutti i tag disponibili con una breve descrizione:

Tag	Descrizione
@author	viene seguito da una stringa che identifica l'autore dell'entità documentata
@version	specificava la versione dell'entità documentata, spesso indicata con la versione del software nella quale è stata introdotta; è seguita da un testo descrittivo
@deprecated	identificava un'entità destinata a scomparire nelle future versioni del software ma ancora in uso; viene seguito da una descrizione testuale che solitamente indica l'entità con cui verrà rimpiazzata
@exception @throws	indicavano un'entità che può sollevare eccezioni, sono sinonimi e sono seguiti dal nome della classe, ovvero il nome dell'eccezione che può essere generata, e da una descrizione testuale
@param	indica un parametro passato nel costruttore dell'entità documentata; è seguito dal nome del parametro e da una descrizione testuale
@return	specificava cosa può essere restituito come risultato dopo l'esecuzione dell'entità associata; viene seguito da una descrizione testuale
@see	indica un link o un riferimento in relazione con l'entità documentata, può essere espresso in diversi modi: seguito da semplice testo, seguito da link html, oppure, nel caso di riferimenti ad altre entità della documentazione, nella forma "package.classe#metodo testovisualizzato"

Una volta commentati in modo adeguato tutti i sorgenti è possibile specificare le impostazioni per la generazione della **Javadoc**: prendono il nome di “option” e possono specificare per esempio la presenza della pagina di overview, un CSS diverso da quello standard, frammenti di codice html da porre in cima o alla fine di ogni pagina creata, il titolo del documento, le cartelle o i file sorgenti, la cartella in cui la **Javadoc** verrà creata, la visibilità o meno di elementi pubblici, protetti o privati.

L'opzione sicuramente più interessante è quella che permette la specifica di una “**Doclet**” diversa da quella standard: le **Doclet** sono estensioni di **Javadoc** che permettono di gestire a piacimento le varie fasi di generazione della documentazione.

ESEMPIO

Si supponga di volere la documentazione nel formato **PDF**: è necessario specificare una **doclet** in grado di istruire **Javadoc** come produrre un **PDF**: la doclet **PdfDoclet** è una delle più semplici e complete tra le molteplici disponibili (<http://pdfdoclet.sourceforge.net/>).

Scaricata la libreria (la **pdfdoclet-1.0.2-all.jar** è disponibile anche tra i materiali della sezione del sito www.hoepiscuola.it riservato a questo volume), è sufficiente specificarne il nome tra le opzioni del comando di generazione:

```
javadoc -doclet com.tarsec.javadoc.pdfdoclet.PDFDoclet
-docletpath pdfdoclet-1.0.2-all.jar -pdf <nome file pdf> prova.java
```

per ottenere un PDF simile a quello di figura:



Per ulteriori opzioni si rimanda alla documentazione specifica di **PdfDoclet**.

Per queste possibilità e per tutte le modalità di utilizzo del tool **Javadoc** rimanda alla documentazione di riferimento del “linguaggio” all’indirizzo: <http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/index.html>



Prova adesso!

Riprendi gli esempi e gli esercizi scritti in Java nelle unità didattiche precedenti e completali con i tag **Javadoc** creando l’ipertesto completo:

- 1 Unità didattica 1: lezione 10-11
- 2 Unità didattica 1: lezione 12-13
- 3 Unità didattica 2: lezione 9
- 4 Unità didattica 2: lezione 10
- 5 Unità didattica 2: lezione 11

ESERCITAZIONI DI LABORATORIO 2

IL SOFTWARE DOXYGEN

■ Cos'è Doxygen

La spiegazione sintetica ed esauriente di cosa sia **Doxygen** la si trova sul sito Internet dove esso viene distribuito gratuitamente (<http://www.doxygen.org>):



Doxygen is a documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors), Fortran, VHDL, PHP, C#, and to some extent D.



It can help you in three ways:

- 1 It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in L^AT_EX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.
- 2 You can **configure** doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. You can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.
- 3 You can also use doxygen for creating normal documentation.

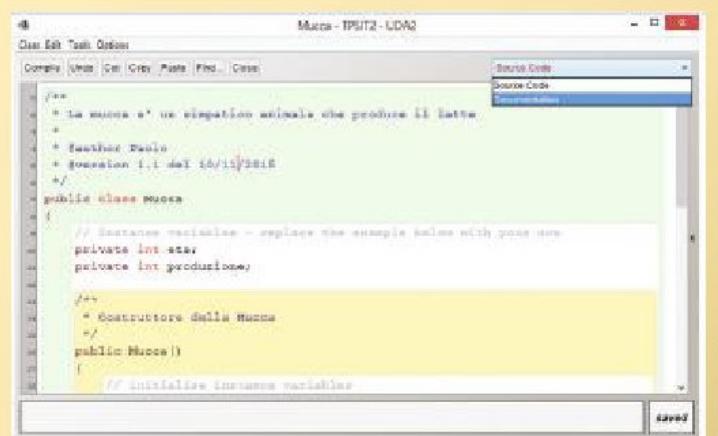
Doxygen is developed under Linux and Mac OS X, but is set-up to be highly portable. As a result, it runs on most other Unix flavors as well. Furthermore, executables for Windows are available. ►

Doxygen è un tool per la generazione automatica della documentazione di codice sorgente (C, Java ecc.) partendo dai singoli commenti presenti nel codice stesso, opportunamente formattati.

La formattazione avviene inserendo dei commenti nel codice per impaginare una documentazione utilizzando particolari tag.

I programmati **Java** hanno già a disposizione uno strumento simile che produce la documentazione automatica: **Javadoc**, che da **Bluej** si genera direttamente selezionando l'opzione **Documentation**. ►

Doxygen è molto simile a **Javadoc** e utilizza una sintassi dei comandi praticamente uguale: anch'esso genera un ipertesto **HTML** ma aggiunge molte funzionalità tra cui la generazione dei file in formato ◀ **Latex** ▶ e la creazione, a partire da esso, di un documento: il **Reference Manual**.



◀ **Latex** Latex (scritto anche **L_AT_EX**) è un linguaggio di markup usato per la preparazione di testi basato sul programma di composizione tipografica **T_EX**. ►



■ Installare Doxygen

In alternativa, per installare **Doxygen** è necessario utilizzare un computer connesso a Internet: nella home page del sito <http://www.doxygen.org> si può individuare il link per effettuare il download sia del programma che del manuale.

Se avete installato **Cygwin** non c'è bisogno di installare **Doxygen** in quanto è già presente nell'ambiente: per verificarlo basta digitare **doxygen** al prompt e si visualizzerà la seguente schermata:

```

PaxToMPCarin@PaxToMPCarin: ~
$ doxygen
error: Doxyfile not found and no input file specified!
Doxygen version 1.8.10
Copyright (C) 1997-2005 Dieter Klaesche

You can use doxygen in a number of ways:
1) Use doxygen to generate a template configuration file:
   doxygen [-s] -w [configName]
   If -s is used for configName doxygen will write to standard output.

2) Use doxygen to update an old configuration file:
   doxygen [-s] -w [configName]

3) Use doxygen to generate documentation using an existing configuration file:
   doxygen [configName]
   If -s is used for configName doxygen will read from standard input.

4) Use doxygen to generate a template file controlling the layout of the
   generated documentation:
   doxygen -T [layoutFileName.wml]

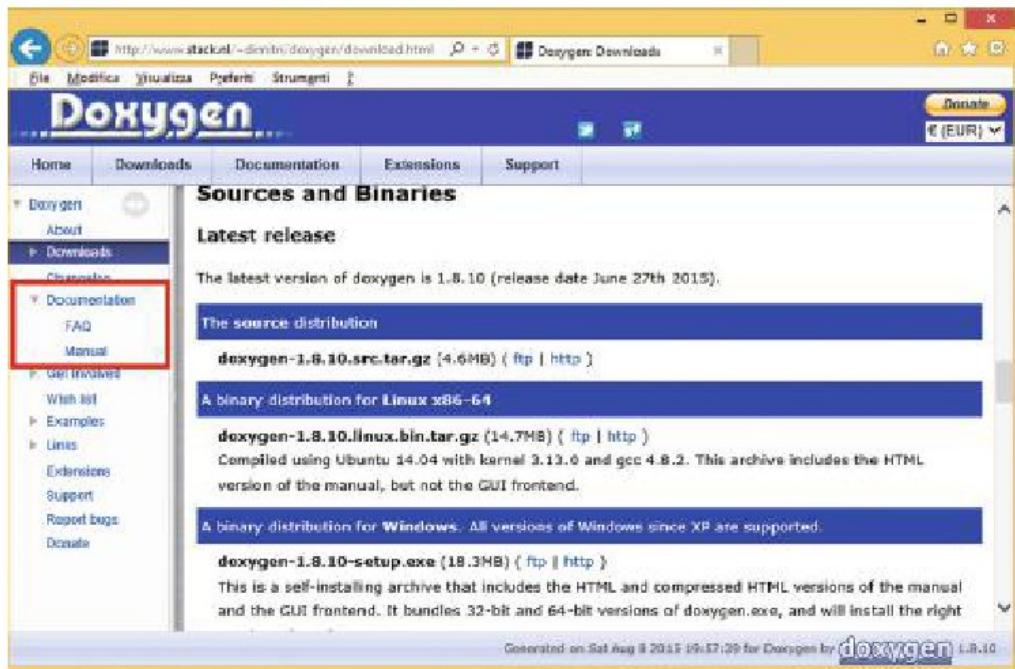
5) Use doxygen to generate a template style sheet file for RTF, HTML or LaTeX.
   RTF:      doxygen -w rtf stylesheetFile
   HTML:     doxygen -w html headerFile footerFile stylesheetFile [configFile]
   LaTeX:   doxygen -w latex headerFile footerFile stylesheetFile [configFile]

6) Use doxygen to generate a rtf extensions file
   RTF:      doxygen -w rtf extensionsFile
If -s is specified the contents of the configuration items in the config file will be omitted.
If configName is omitted 'Doxyfile' will be used as a default.

-v print version string
PaxToMPCarin@PaxToMPCarin: ~
$ 

```

È anche possibile scaricare il programma dal sito www.hoeplicsuola.it nella cartella materiali nella sezione riservata al presente volume.



Nel momento in cui viene scritta questa esercitazione, la versione di **Doxygen** più recente è quella comprendente [doxygen-1.8.10-setup.exe](#). Le istruzioni che seguono faranno dunque riferimento all'installazione di questa specifica versione.

A binary distribution for Windows. All versions of Windows since XP are supported.

[doxygen-1.8.10-setup.exe \(18.3MB\)](#) { ftp | http }

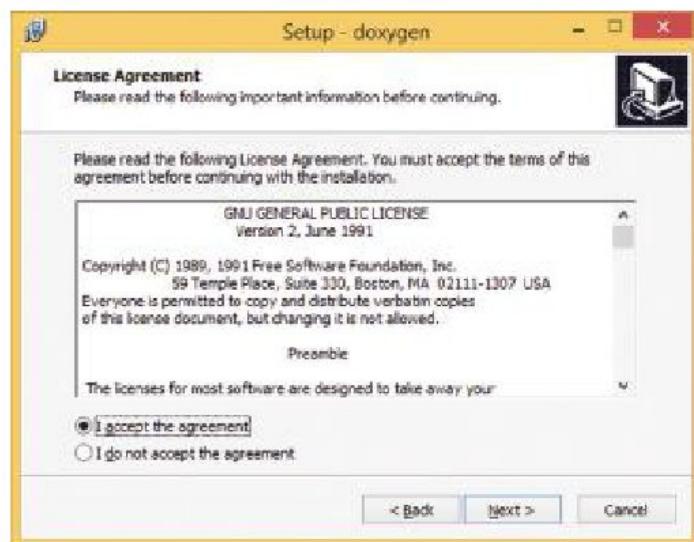
This is a self-installing archive that includes the HTML and compressed HTML versions of the manual and the GUI frontend. It bundles 32-bit and 64-bit versions of doxygen.exe, and will install the right one based on the OS.

If you are allergic to installers and GUIs, haven't sufficient bandwidth, or don't have administrator privileges you can also download the [32-bit doxygen binary in a zip \(4.7MB\)](#) { ftp | http } or the [64-bit version \(5.3MB\)](#) { ftp | http }

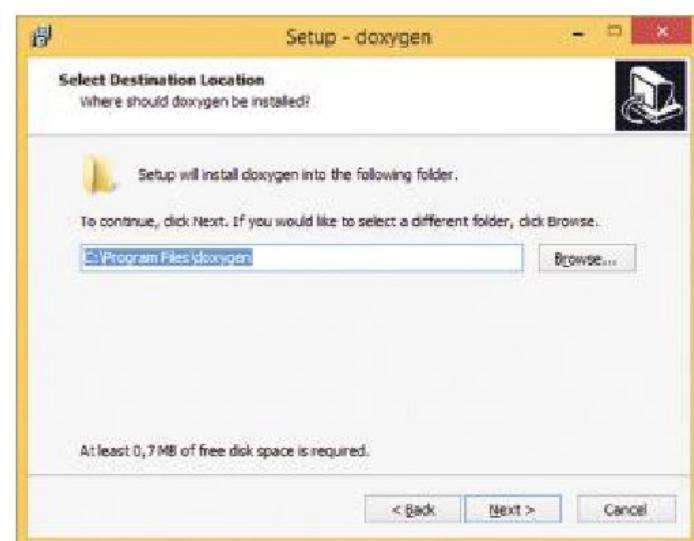
Un semplice click con il pulsante sinistro del mouse su [html](#) inizia il processo di installazione: come prima passo è necessario autorizzare l'esecuzione del programma di installazione cliccando [\[Next>\]](#) nella finestra di protezione che ci si presenta davanti. ►



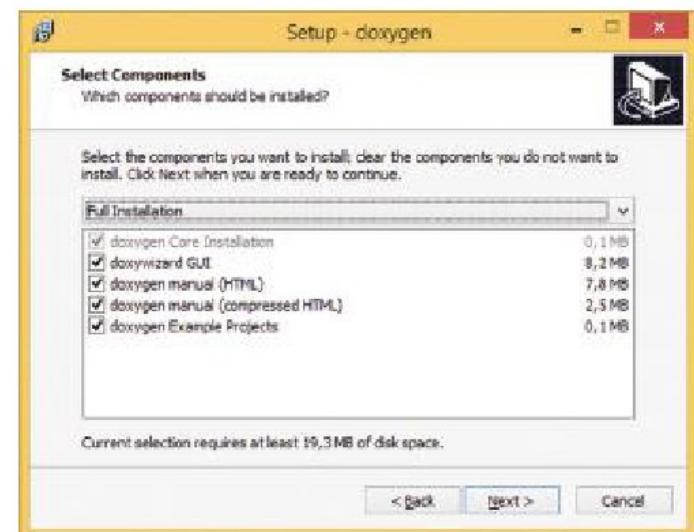
Noi salveremo una copia nel nostro **PC**, per esempio nella cartella **hoepli**, per poi avviare l'installazione, che proporrà la seguente schermata: ►



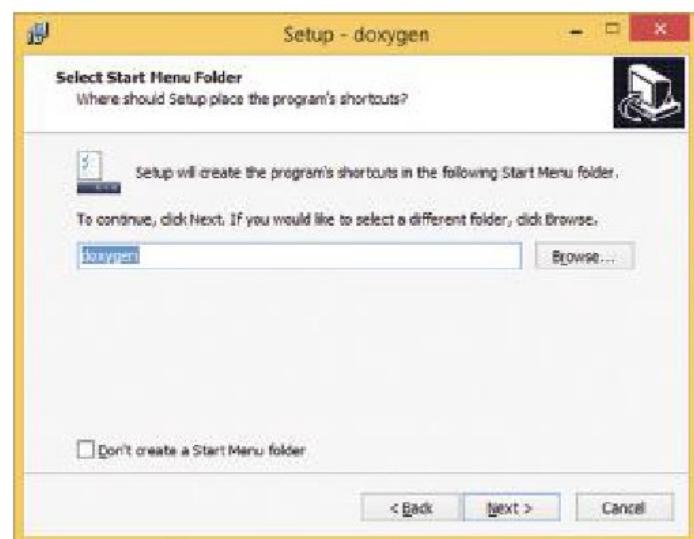
Accettando i termini della licenza d'uso si avvia l'installazione scegliendo la directory di destinazione: ►



Lasciamo settati tutti i flag per effettuare l'installazione completa di tutte le opzioni: ►



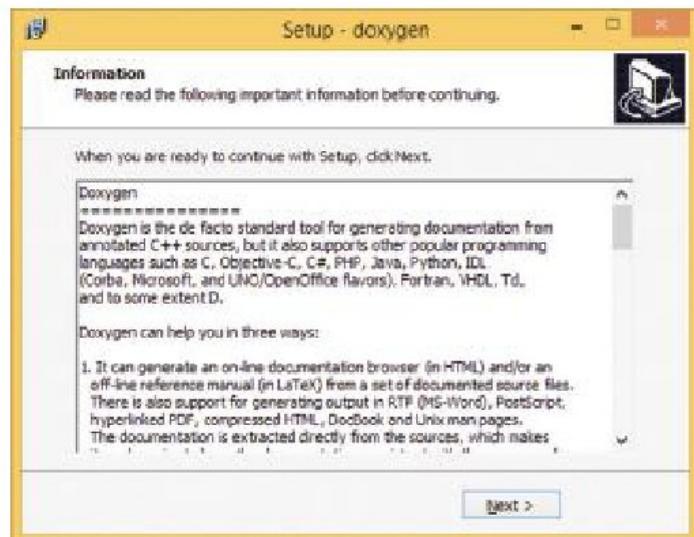
e creiamo una **shorcut** dal nostro menu delle applicazioni di **window**, utile per mandare successivamente in esecuzione il programma: ►



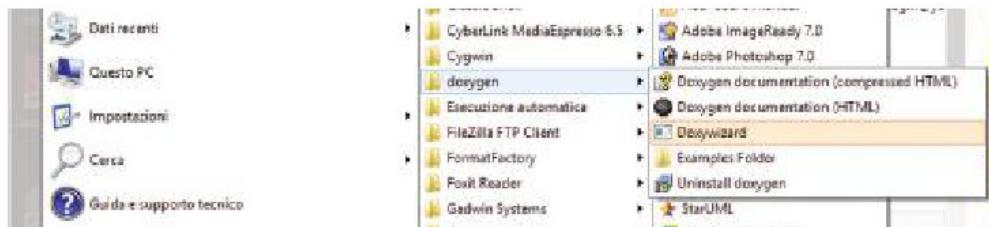
ora siamo pronti per installare il programma: clicchiamo sul pulsante **Install** dalla successiva videata ►



e dopo aver eletto le informazioni sulla versione completiamo l'installazione sempre tramite il pulsante **Install**. ►

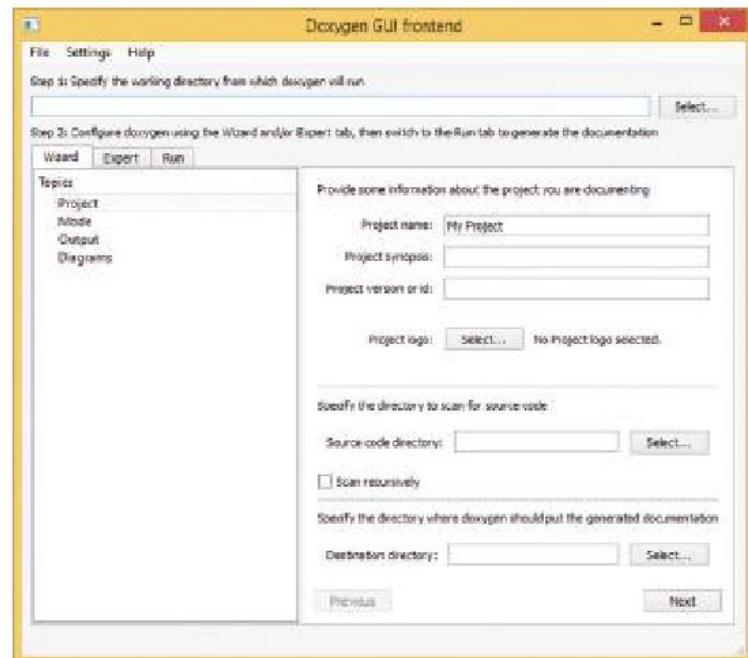


Al termine dell'installazione possiamo mandare in esecuzione il programma dal menu programmi, selezionando il link **Doxywizar**.



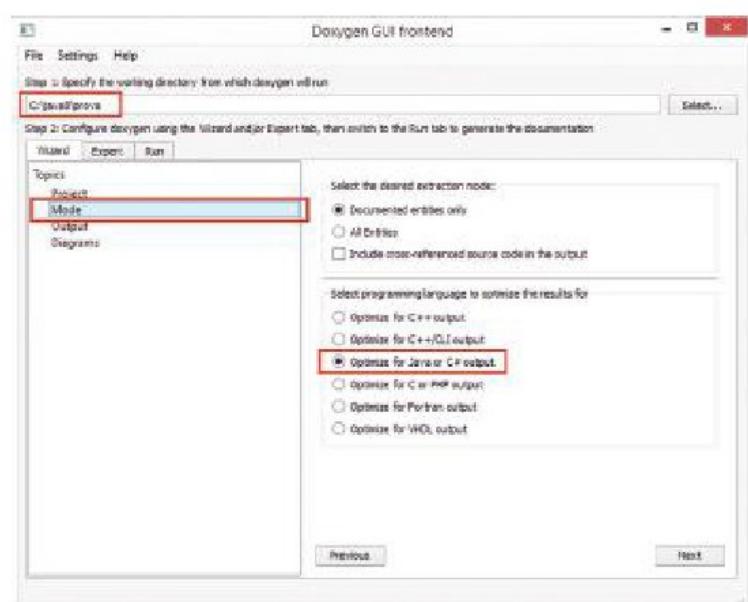
■ Generazione della documentazione da window

La videata iniziale è la seguente: ►

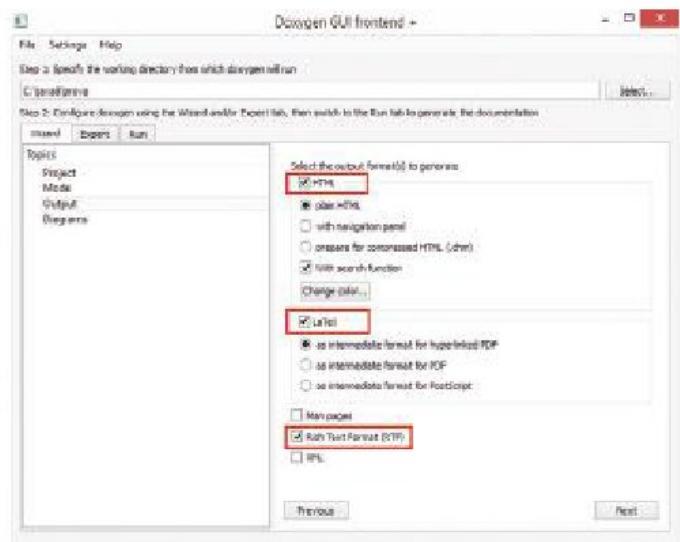


È possibile assegnare un nome al progetto e indicare le directory dove **Doxygen** deve essere mandato in esecuzione (cartella di riferimento, nel nostro esempio è la cartella **prova** dove è presente un singolo file **Java**, *Mucca.java*), dove sono presenti i codici sorgenti e dove deve essere memorizzata la documentazione prodotta.

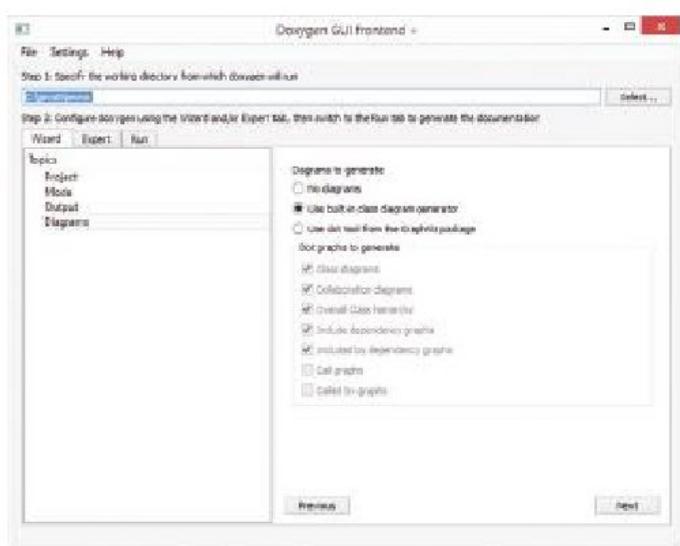
Cliccando su **NEXT** o selezionando **mode** nella finestra **wizard** è possibile scegliere il linguaggio di programmazione: ►



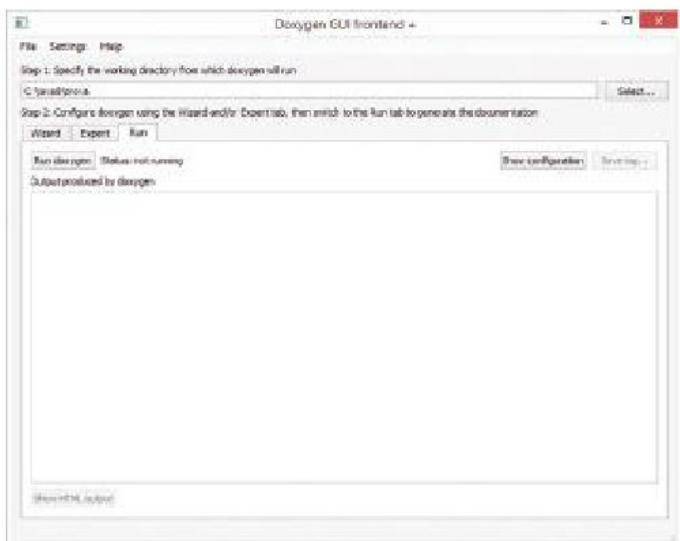
Nella successiva finestra si “spunta” il formato nel quale si desidera venga prodotta la documentazione: ►



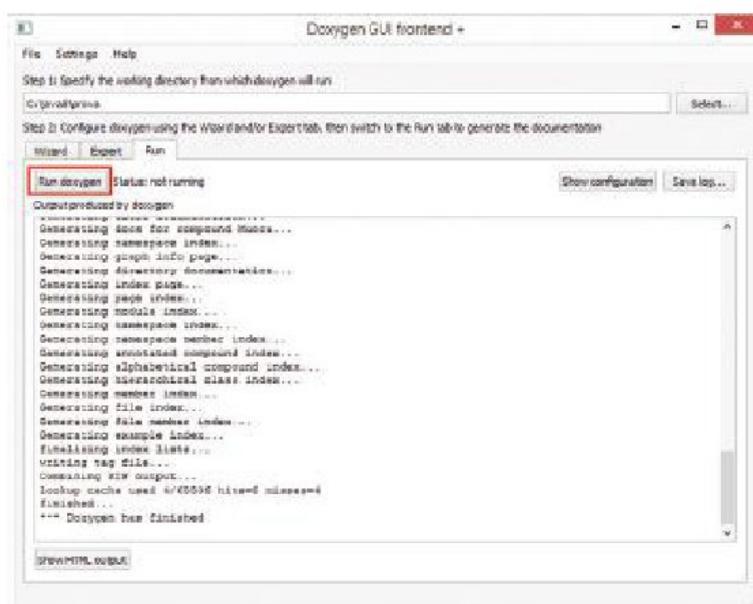
La successiva videata permette di indicare a **Doxygen** se devono essere generati anche dei diagrammi ►



L'ultima **tab**, quella con titolo **RUN**, è la finestra di esecuzione dove viene avviata la generazione della documentazione e viene visualizzato l'echo dei comandi eseguiti dai singoli passi dell'evoluzione di **Doxigen**: clicchiamo il pulsante **Run doxygen** ►



e otteniamo una esecuzione simile alla seguente ►



Al termine possiamo direttamente visualizzare l'output generato in formato **HTML** cliccando sull'apposito pulsante: se invece apriamo la cartella **prova** possiamo osservare che sono presenti tre nuove cartelle, una per ogni formato di documentazione che abbiamo precedentemente richiesto: ►

Name	Ultima modifica	Tipo
html	23/11/2015 17:29	Cartella di file
latex	23/11/2015 17:29	Cartella di file
rtf	23/11/2015 17:29	Cartella di file
src	23/11/2015 17:23	Cartella di file
Mucca.class	30/09/2012 12:09	File CLASS
Mucca	30/09/2012 13:47	File JAVA

Visioniamo il codice **HTML** e nel browser ci viene proposto il seguente ipertesto (molto simile a quello prodotto da **Javadoc**): ►

The screenshot shows a web browser window with the URL 'file:///C:/java8/prova/html/class_mucca.html'. The page content is as follows:

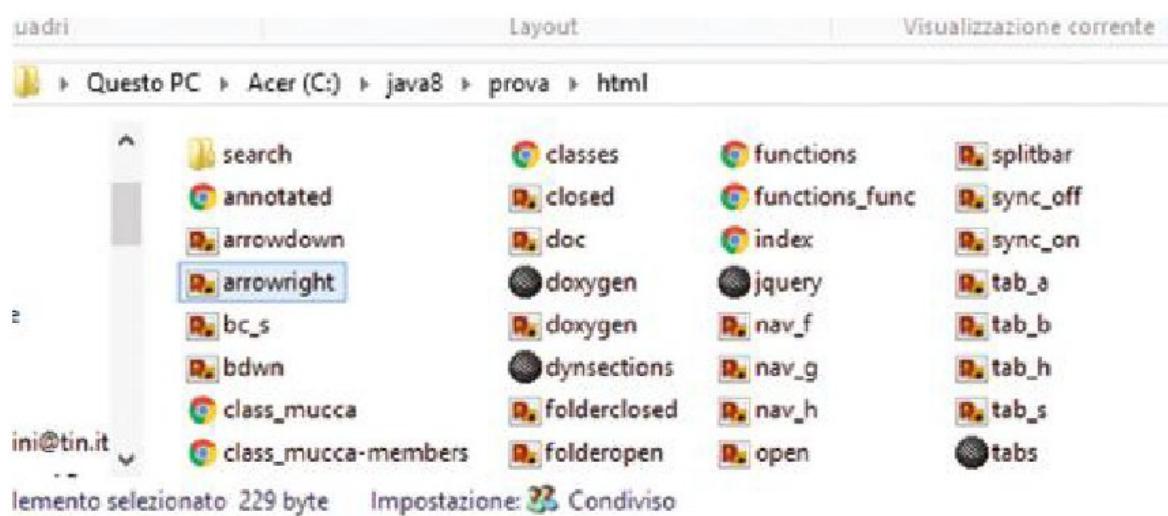
- Public Member Functions**
 - `Mucca ()`
 - `int mangi (int libri_latte)`
 - `boolean mangia (int kg_feso)`
- Detailed Description**

La mucca è un simpatico animale che produce il latte
- Author**

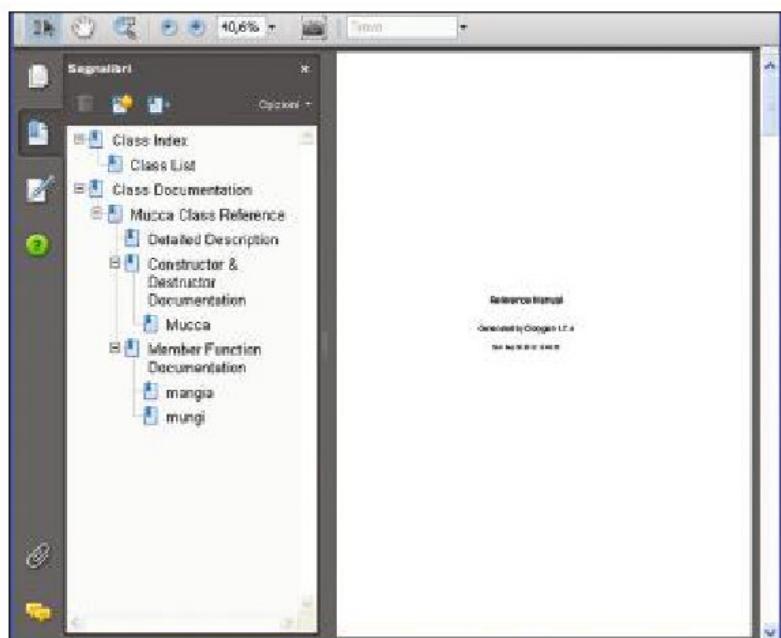
Paolo
- Version**

1.0 del 10/10/2012
- Constructor & Destructor Documentation**
 - `Mucca.Mucca ()`
 - Coefficiente della Mucca**

Se visioniamo il contenuto della cartella **HTML** troviamo tutti i file



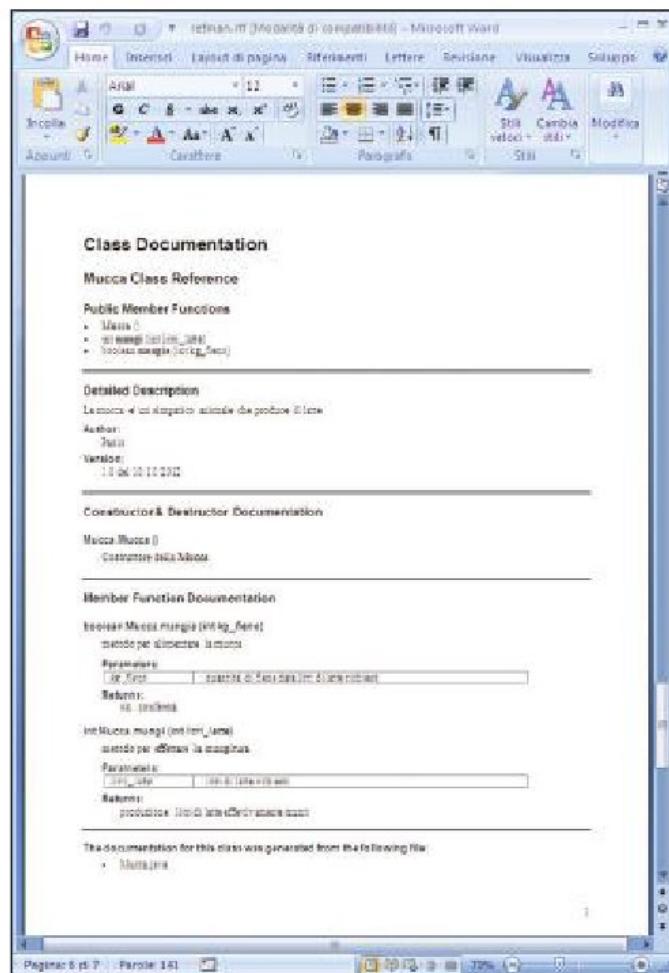
Entriamo nella sottodirectory **latex** e mandiamo in esecuzione il file make.bat: viene compilato automaticamente il manuale in **PDF**, come quello riportato di seguito. ►



Come ultima operazione entriamo nella sottodirectory **rtf**



e troviamo la documentazione anche in forma **rtf**, leggibile ed editabile con **MS Word**. ▶



■ Generazione della documentazione da linux/Cygwin

Creiamo un nuovo progetto e predisponiamo il file di configurazione di default con il comando

```
doxygen -g <nomefile>
```

dove sostituiamo <nomefile> col nome che vogliamo assegnare al nostro file, per esempio **config_DG**. ▶

Il file **config_DG** può essere aperto in qualunque editor di codice (per esempio **Dev-Cpp**) oppure anche con **MS Word**.

```
Paolo@PCwin8 ~
$ mkdir progetto1
Paolo@PCwin8 ~
$ cd progetto1
Paolo@PCwin8 ~/progetto1
$ doxygen -g config_DG

Configuration file 'config_DG' created.
Now edit the configuration file and enter
    doxygen config_DG
to generate the documentation for your project

Paolo@PCwin8 ~/progetto1
$ dir
config_DG
```

Si prosegue editando il file di configurazione appena generato settando i seguenti parametri:

```
PROJECT_NAME      = nome del progetto
OUTPUT_DIRECTORY = cartella in cui generare la documentazione
INPUT            = percorso del codice sorgente
FILE_PATTERNS    = Estensioni da considerare, per esempio .h,.cpp, .cc ecc...
```

La documentazione verrà generata in formato html nella cartella “[html](#)”.



Per generare la documentazione automatica in qualsiasi momento è sufficiente digitare

```
doxygen <nomefile>
```

Prima di generare la documentazione è necessario inserire nel codice la descrizione di classi, metodi, e attributi, come descritto in seguito.

■ Elenco dei comandi

I commenti possono essere:

```
// Commento su linea singola
/*
 * commento su più linee
*/
```

I commenti utilizzano alcuni particolari tag di formattazione che, uniti a particolari parole riservate, permettono di far risaltare nella documentazione diverse informazioni, tipo il nome della funzione, l'autore, la data di creazione o modifica, i parametri, il file sorgente di appartenenza o più semplicemente un breve commento o nota.

Vediamo ora alcuni tipi di tag, che devono essere scritti compresi tra

```
/*
< riga inizianta con TAG per essere visualizzata nella documentazione>
*/
```

Tag	Descrizione	Esempio		
\fn	serve per identificare il nome della funzione	\fn	int somma(int a, int b)	
\brief	permette di visualizzare un commento	\brief	funz. che somma i valori interi a e b	
\param	permette di identificare sulla documentazione un parametro di funzione e di applicarvi una descrizione	\param	int a: Primo valore da sommare \param	int b: Secondo valore da sommare
\return	permette di descrivere ciò che ritorna la funzione	\return	int: la funzione ritorna la somma int dei due parametri	
\date	permette di visualizzare sulla documentazione una data, per esempio quella di creazione o modifica	\date	12/10/2010	

Tag	Descrizione	Esempio
\author	permette di stampare il nome dell'autore dell'ultima modifica o chi ha creato la funzione	\author Zio Pino
\file	permette di visualizzare sulla documentazione il file sorgente su cui si trova la funzione in questione	\file primoesempio.c
\version	permette di visualizzare sulla documentazione la versione del file	\version 4.1
\bug	permette di visualizzare sulla documentazione l'ultimo bug trovato	\bug problema corretto Out of Range
\class	permette di visualizzare sulla documentazione il nome della classe che si commenta	\class <nome della classe>
\warning	permette di visualizzare sulla documentazione l'ultimo bug trovato	\warning la funzione ritorna int, ma la somma può dare un double
\example	permette di specificare un file di esempio. Nell'html verrà richiamato un file sorgente di esempio	\example nomefile_esempio.c

Per realizzare una documentazione semplice ma completa è sufficiente avere l'accortezza di inserire i tag di commento almeno in quattro posizioni:

- 1 all'inizio di ogni file inserire le seguenti righe:

```
/*
 @file      fileName.cc .cpp .h ecc ecc ecc
 @author    name, mail
 @version   1.0
 */
```

- 2 prima della dichiarazione di una classe:

```
/** Descrizione classe...
 @code
 ... eventuale codice di esempio ...
 @endcode
 */
```

- 3 prima di ogni metodo (o ridefinizione o template):

```
/**
 Descrizione metodo...
 @param    a parametro 1
 @param    b parametro 2
 @return   valore di ritorno
 @throws   eccezioni...
 */
```

- 4 prima di ogni attributo:

```
/** descrizione */
```

In questo modo si otterrà una documentazione completa ed esaustiva, come quella riportata nei successivi esempi.

ESEMPIO Documentiamo una classe C

Come primo esempio creiamo la documentazione per un semplice **file C**: creiamo una directory **progettoC** e scriviamo il seguente programma salvandolo nel file **primofile.C** ►

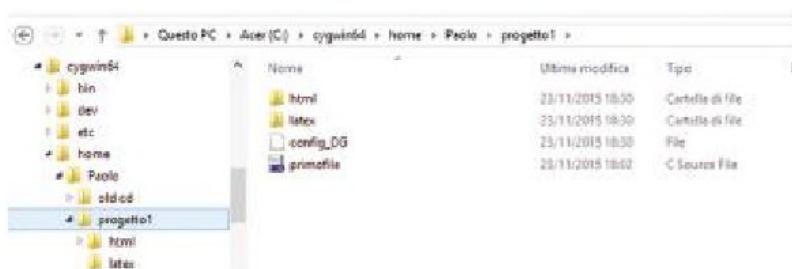
```
primofile.c
1 // file primofile.c
2 // brief file di esempio
3 // date il file è stato creato il 10/11/2015
4
5 /**
6 * \fn int somma(int a, int b)
7 * \brief funzione che somma i parametri a e b
8 * \param int a: Primo valore da sommare
9 * \param int b: Secondo valore da sommare
10 * \return int: la funzione ritorna la somma int dei due parametri
11 * \date 10/11/2015
12 * \author Paolo Comogni
13 * \file primofile.c
14 */
15
16 int somma(int a, int b)
17 {
18     return a + b;
19 }
```

Digitando il comando

```
doxygen config_DG
```

si genera la documentazione sia in formato **html** che **Latex**: esplorando la cartella **html** ►

selezioniamo **index.html** per visualizzarlo in un browser, come riportato di seguito: ►



primofile.c File Reference

Functions

`int somma (int a, int b);`
funzione che somma i parametri a e b... More...

Function Documentation

`int somma (int a,
 int b
)`

funzione che somma i parametri a e b

Parameters

`int a:` Primo valore da sommare
`int b:` Secondo valore da sommare

Returns:

`int:` la funzione ritorna la somma int dei due parametri

ESEMPIO Documentiamo una classe Java

Come secondo esempio generiamo la documentazione per la **classe Mucca.java** di seguito riportata:

The screenshot illustrates the process of generating Java documentation. On the left, a code editor window titled "Mucca" shows the source code for the `Mucca` class. The code includes Javadoc comments for the constructor, the `mungo` method, and the `mangia` method. On the right, a web browser window titled "My Project Mucca Class" displays the generated documentation. The main page is titled "Mucca Class Reference". It features a "Public Member Functions" section listing the constructor `Mucca()`, the `mungo` method (with parameter `litri_latte`), and the `mangia` method (with parameter `kg_fieno`). Below this is a "Detailed Description" section containing the class's purpose and author information.

```

/*
 * La mucca e' un simpatico animale che produce il latte
 *
 * Author: Paolo
 * Version: 1.0 del 10/10/2012
 */
public class Mucca {
    // instance variable - rappresenta la mucca bolla with your cow
    private int eta;
    private int produzione;

    /**
     * Costruttore della mucca
     */
    public Mucca() {
        // inizializza instance variabili
        eta = 3;
    }

    /**
     * metodo per effettuare la mangitura
     *
     * @param litri_latte : litri di latte richiesti
     * @return produzione : litri di latte effettivamente mungiti
     */
    public int mungo(int litri_latte) {
        // per ogni litro restituito
        return produzione;
    }

    /**
     * metodo per alimentare la mucca
     *
     * @param kg_fieno : quantita' di fieno data intera al latte mangiato
     */
    public boolean mangia (int kg_fieno) {
        boolean ok;
        ok=true;
        return ok;
    }
}


```

viene prodotta la seguente documentazione ►

GraphViz

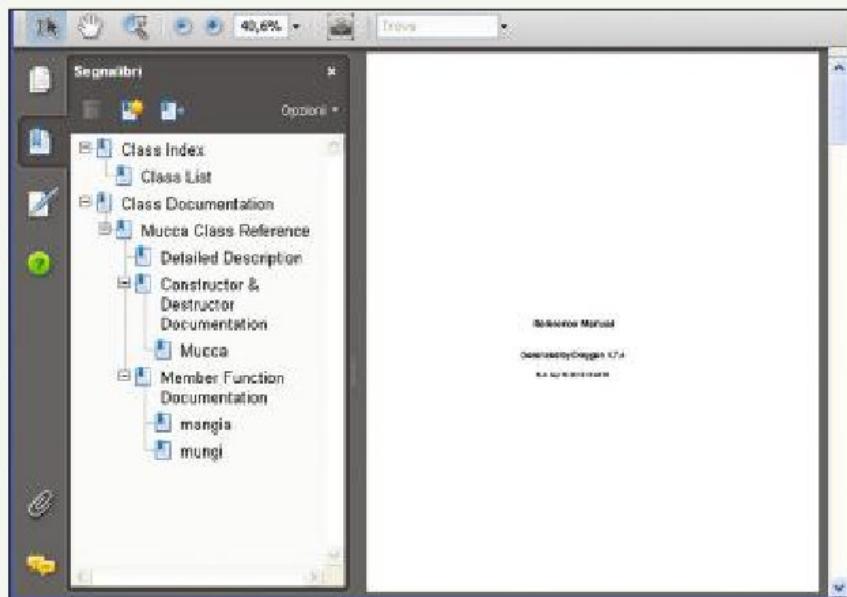
GraphViz è un tool che deve esser scaricato a parte e che può essere utilizzato separatamente o integrato al **Doxygen**.

In breve ci permette di creare un grafo che mostra dove una funzione viene chiamata e da chi oppure che cosa chiama al suo interno.



Prova adesso!

- 1 Scarica il file Mucca.java, aggiungi almeno altri tre metodi e copiala in una nuova directory di cygwin\home\tuonome chiamandola **progettoJava**
- 2 lancia
doxygen -g documenta
- 3 quindi digita il comando
doxygen documenta
- 4 entra nelle sottodirectory create e osserva (ls) i vari output generati
- 5 visualizza con un browser il file index.html
- 6 entra nella sottodirectory latex e digita semplicemente **make**: viene compilato automaticamente il manuale in **PDF**, come quello riportato di seguito.



- 7 Per comprendere le potenzialità di Doxygen, guarda i seguenti esempi di documentazione:
<http://api.kde.org/>
<http://www.opensg.org/doxygen/>
<http://www.abisource.com/doxygen/index.html>



VERSIONE
SCARICABILE
EBOOK

e-ISBN 978-88-203-6941-5

www.hoepliscuola.it

Ulrico Hoepli Editore S.p.A.
via Hoepli, 5 - 20121 Milano
e-mail hoepliscuola@hoepli.it