Università degli Studi di Bologna
Facoltà di Ingegneria

Corso di

Reti di Calcolatori T

Integrazione di Sistemi e Consapevolezza nell'uso delle risorse

Antonio Corradi, Luca Foschini
Michele Solimando, Marco Torello

Anno Accademico 2020/2021

Integrazione di sistema

Programmazione tradizionale

Programmazione moderna

Programmazione concentrata e monolitica

Programmazione devops a microservizi
DEVelopment OPerationS

Programmazione applicativa

Programmazione di sistema e consapevole delle risorse (resource-aware)

Programmazione imperativa

Programmazione dichiarativa

Programmazione di sistema

Accanto alla **programmazione applicativa**, fatta per raggiungere obiettivi specifici e costituita da applicazioni che rispondono a **esigenze utente** specifiche, i servizi applicativi non potrebbero funzionare senza un **supporto di programmi** che consentono di preparare l'ambiente e forniscono le **funzionalità di servizio**, che chiamiamo **componenti di sistema (non solo sistema operativo)**

In generale, i **componenti di sistema** tendono ad eseguire **molte volte, molto spesso, e devono essere ottimizzati**

In particolare nei sistemi distribuiti, tendiamo ad avere sempre più **componenti di sistema** che si devono **coordinare tra di loro in modo dinamico e veloce**

Programmazione concentrata monolitica

La esigenza di avere componenti in un sistema è stata spesso espressa in **modo monolitico e concentrato**: un programma è sempre lo stesso, e contiene sempre **tutte le funzioni** necessarie

(le funzioni possono essere ottimizzate)

Se attiviamo più volte lo stesso programma con più incarnazioni o processi, possiamo avere molti possibili servizi applicativi contemporaneamente

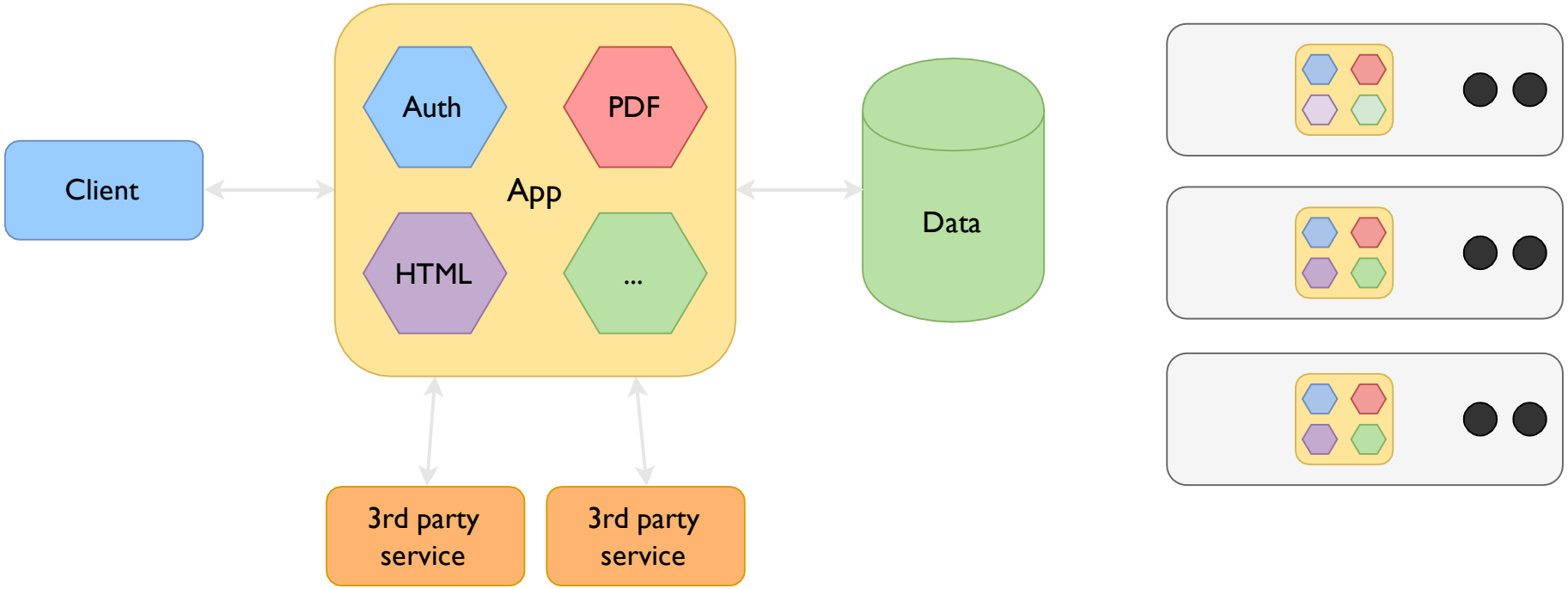
Però più processi comportano la ripetizione delle parti di supporto, in modo non scalabile

Nel **distribuito poi questa architettura** mostra ancora più limiti

Problemi della programmazione monolitica

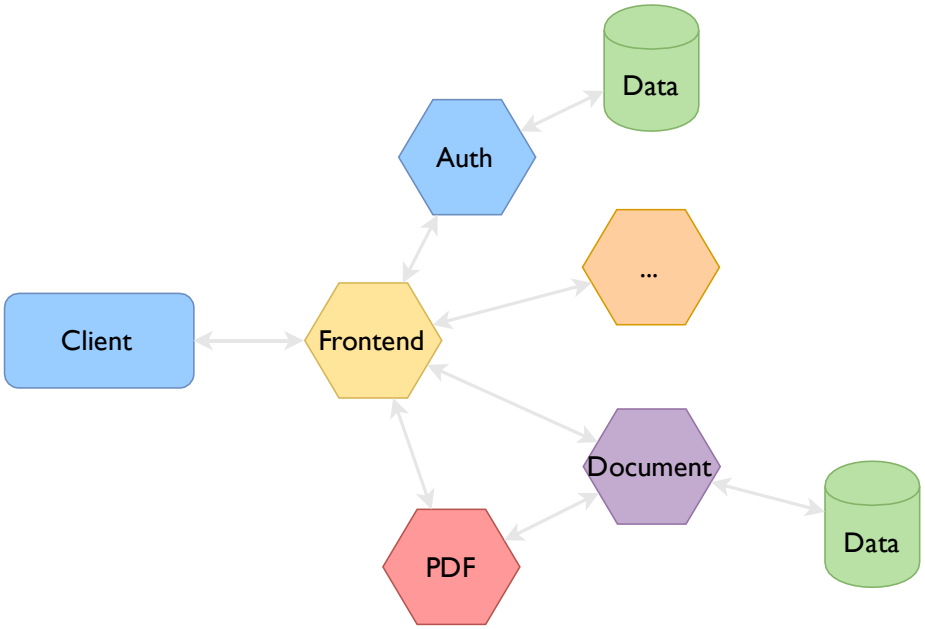
Considerando più processi, in generale **dobbiamo pensare a strutture ripetute per ogni processo, sia per la parte di supporto, sia per la parte applicativa…**

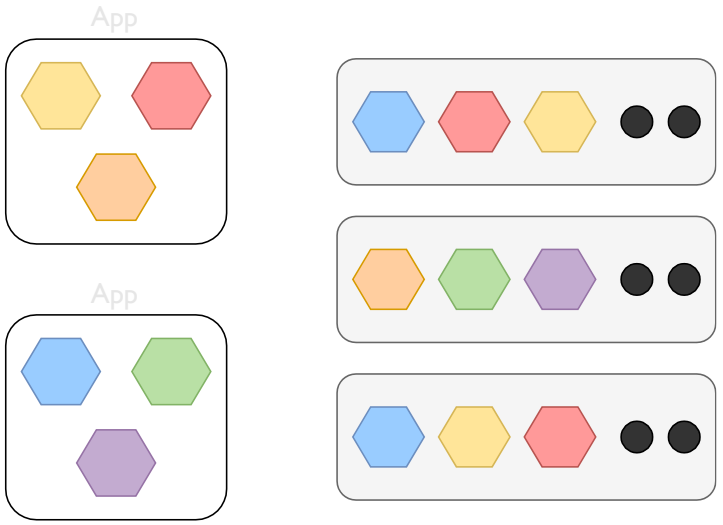
Vedi l'esempio sotto



Programmazione a componenti piccoli

La esigenza di avere **componenti**, che si devono coordinare tra di loro **in modo dinamico e veloce e ottimizzato per l'uso**, ha portato a definire **programmi piccoli, efficienti e molto capaci di interagire in modo dinamico** che si possano comporre al bisogno e con costi limitati (**microservizi**)





Devops e Microservizi

I **Microservizi** sono parte della **programmazione di sistema,** in quanto i **piccoli componenti** sono anche **contenitori** sia della parte applicativa, sia della parte di sistema

Questa tendenza che impacca insieme la **applicazione** con tutto quello che è **necessario per supportarla** in un unico servizio piccolo (**microservice**) è una direzione fortemente propagata del filone detto **DEVOPS**, ossia tutta la tecnologia che mette attenzione **sulla parte di supporto e lo considera integrante e da integrare nella definizione della applicazione**

Una applicazione a microservizi è fatta a componenti applicativi e di supporto insieme

Consapevolezza delle Risorse

In generale, **ogni programma che si debba mettere in esecuzione ha bisogno di risorse di sistema di basso e alto livello**

Quali? **Processori, Processi, Memoria, Dati, Codice, File**…

Risorse di **vario tipo e proprietà**

Anche nei sistemi concentrati (unico processore)

Da PERSEGUIRE

questo significa fare **programmi piccoli, efficienti e molto capaci di interagire in modo dinamico**

Uso delle **risorse della architettura in modo ottimale**, senza sprechi non necessari con buon **controllo delle situazioni di errore**

Da NON FARE

NON usare **risorse complesse e pesanti e difficili da portare** tra architetture e da implementare

Modelli di architettura della soluzione

In generale, nella espressione dei programmi ci sono due paradigmi ispiratori:

Programmazione Imperativa

Programmazione Dichiarativa

La prima descrive i linguaggi tradizionali e anche ad oggetti con la **specifica di algoritmi precisi di soluzione**

La seconda fa riferimento a modi diversi di **esprimere la computazione in modo meno deterministico**

Modelli di architettura della soluzione

Programmazione imperativa (codice e dati)

In questo paradigma che ispira tutti i linguaggi che avete visto, **le istruzioni specificano completamente l'algoritmo** da risolvere senza 'libertà' di cercare soluzioni non specificate

L'algoritmo viene specificato come sequenza (più o meno) precisa di passi che vengono **eseguiti una dopo l'altra** e che agiscono su **dati, intesi come contenitori** in memoria delle informazioni

(in genere è il sistema operativo che inserisce poi la possibilità di concorrenza …)

Vedi C, C++, Java, C#, …

Tutti questi **linguaggi usano stato** (inteso come **variabili - e loro tipi**) che sono manipolate dagli algoritmi

Modelli di architettura della soluzione

Programmazione dichiarativa

Questo paradigma porta ad esprimere soluzioni che devono soddisfare **una serie di requisiti specificati dall'utilizzatore**, senza arrivare ad una specifica completa di **algoritmo**, che spesso può esplorare soluzioni molteplici e viene guidato **da non determinismo e da modalità ad elevato innestamento funzionale o logico garantito da un motore** alla base del supporto

I linguaggi sono basati su diversi paradigmi,

vedi LISP, Prolog, Scala,…

La specifica esecuzione viene lasciata alla libertà di supporto del **motore di base**, che potrebbe essere logico, funzionale, ecc.

Noi non ci occuperemo di questa

Architettura delle soluzioni

Nei sistemi distribuiti ancora di più si devono considerare le risorse messe in gioco ed impegnate

Sviluppo statico della applicazione

Supporto dinamico durante la esecuzione

Architettura logica della soluzione

Architettura del supporto alla applicazione

Modello di sviluppo integrato

Integrazione dei sistemi e dei componenti

Sistemi a livelli e consapevolezza reciproca

ARCHITETTURA logica e deployment

Per una applicazione

1) *Design dell'algoritmo di soluzione*
2) *Codifica in uno o più linguaggi di programmazione*

   dobbiamo usare le specifiche dei linguaggi e definire le **risorse logiche del programma**

3) **Obiettivo: arrivare ad eseguire**

Per arrivare alla esecuzione, dobbiamo stabilire quali sono le **risorse hardware e fisiche** su cui potere eseguire

Dobbiamo attuare una corrispondenza **tra la parte logica e la parte concreta di architettura**

Questa architettura scelta con la sua **specifica configurazione** è la scelta di **deployment**

Il **deployment** è la fase in cui decidiamo come **allocare le risorse logiche alle risorse fisiche disponibili** (che abbiamo preparato decidendo **una configurazione di esecuzione**)

ARCHITETTURA SUPPORTO

Fasi Statiche (prima della esecuzione)

Da una applicazione partiamo dall'algoritmo

Analisi, sviluppo, e progetto dell'algoritmo e sua codifica in uno o più linguaggi di programmazione

Una volta prodotta ***una forma eseguibile***, noi vogliamo eseguire quel programma

Fino a qui non abbiamo nessuna esecuzione

Fasi dinamiche

Dobbiamo decidere su quale architettura concreta e poi caricare il programma per quella configurazione (**deployment**)

Poi cominciare la **reale esecuzione,** da cui **ottenere dati e la migliore performance**

Noi siamo interessati alla esecuzione e magari anche a diverse possibili architetture diverse, senza rifare le fasi statiche

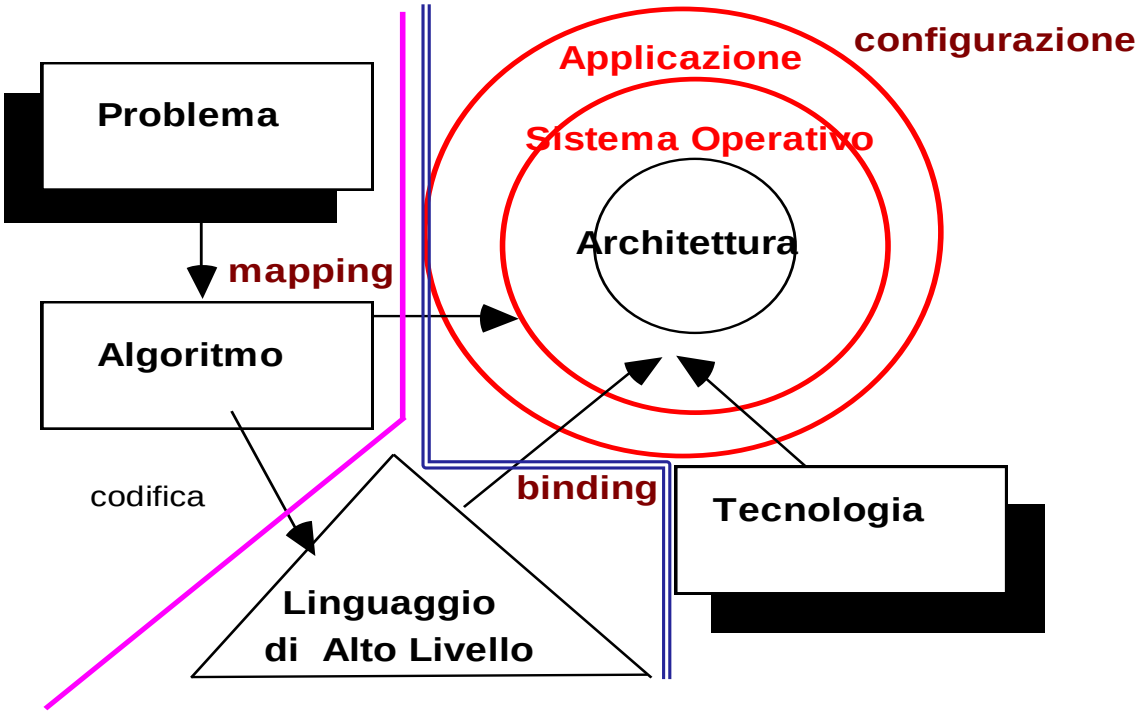
ARCHITETTURA SUPPORTO

Per una applicazione vorremmo avere fasi dinamiche molto efficienti e con la possibilità di usare al meglio le risorse e di non sprecarle

Magari molte esecuzioni diverse

Nel corso massimo interesse per le fasi dinamiche

Deployment ed Esecuzione (molteplici)



INTEGRAZIONE DEI SISTEMI

Una esecuzione efficiente dipende da una comprensione integrale a tutti i livelli

Dalla applicazione fino a tutti i livelli del supporto

CONSAPEVOLEZZA del SUPPORTO

FINO ALLA APPLICAZIONE

Nel caso di programmi di sistema è necessario acquisire una capacità di mettere in relazione i **rapporti tra i diversi livelli** e come i livelli richiedono risorse fino alla interazione con le ricorse locali

Livello Applicazione

Livello supporto linguaggio (C, Java, …)

Protocolli di comunicazione

Sistema Operativo

Risorse locali

Principi di progetto

Regole di base al progetto di sistema

Semplicità

Efficienza

Orientamento ai requisiti

Strutture dati

Algoritmi

Risorse e loro controllo

Protocolli di interazione

Interazione delle applicazioni e loro controllo

ARCHITETTURA SUPPORTO

Una esecuzione efficiente richiede anche una **abilità nella parte di progetto** per avere **componenti** di cui si possano fare **deployment efficienti in ogni ambiente**

Nel caso di programmi di integrazione di sistema (non applicativi) che devono essere **parte del supporto** e magari **eseguiti milioni di volte**, allora il progetto deve mirare a:

- **Semplicità delle strutture dati e degli algoritmi**
- **Minimizzazione del costo della configurazione**
- **Minimizzazione dell'impegno sulle risorse**
- **Minimizzazione overhead**
- **Capacità di interazione con l'utente**
- **Prevenzione errori**
- **Capacità di controllo della esecuzione**
- …

SEMPLICITÀ DELLE STRUTTURE DATI

Semplicità delle strutture dati

- Strutture statiche

Le variabili in memoria **non dinamica** sono predefinite e non hanno costi aggiuntivi durante la esecuzione (overhead della allocazione dinamica delle aree sullo stack e dall'heap)

- Struttura dati limitate

Le variabili con memoria limitata (array) non hanno costi in eccesso per memoria non usata (allocazione di liste linkate)

- Strutture dati semplici e non troppo astratte

Le strutture dati semplici (array e **non lista linkata** o **hash list o hash table)** non a volte hanno **capacità espressiva non richiesta**

- Controllo delle risorse allocate

- Prevenzione errori

- Capacità di controllo della esecuzione
- Capacità di interazione con l'utente
- …

SEMPLICITÀ DELLE STRUTTURE DATI (1)

Vediamo come cambiano le performance in base **alla complessità delle strutture** utilizzate per implementare un algoritmo in C

ESEMPIO a: Scrivere un programma che memorizza una serie di interi (100.000) e verificarne le performance

```
startTime = System.currentTimeMillis();
  for (int i=0; i<99999; i++) arr[i]=i+1;
stopTime = System.currentTimeMillis(); elapsedTime = stopTime - startTime;
System.out.println("Fill Simple Array: "+elapsedTime + ", start="+startTime+
", stop="+stopTime);
/*Fill Simple Array: 2 millisecondi, start=1536829145957,
stop=1536829145959*/
```

```
startTime = System.currentTimeMillis();
  for (int i=0; i<99999; i++) arrL.add(i+1);
stopTime = System.currentTimeMillis();
elapsedTime = stopTime - startTime;
System.out.println("Fill Array List: "+elapsedTime + ", start="+startTime+
", stop="+stopTime);
/*Fill Array List: 18 millisecondi, start=1536829145960,
stop=1536829145978*/
```

SEMPLICITÀ DELLE STRUTTURE DATI (2)

Vediamo come cambiano le performance in base alla **complessità delle strutture dati**
ESEMPIO b: Scrivere un programma che conti tutte le occorrenze di un valore in un struttura

```
startTime = System.currentTimeMillis();
  for (int i = 0; i < arr.length; i++) if (arr[i]==5000)  occ++;
stopTime = System.currentTimeMillis();
System.out.println("Occurrences = "+occ);
elapsedTime = stopTime - startTime;
System.out.println("Read Simple Array "+elapsedTime + ", start="+startTime+
", stop="+stopTime);
/*Read Simple Array 2 millisecondi,
start=1536829938252, stop=1536829938254*/
```

```
startTime = System.currentTimeMillis();
System.out.println("Occurrences = " +
    java.util.Collections.frequency(arrL,5000));
stopTime = System.currentTimeMillis();
elapsedTime = stopTime - startTime;
System.out.println("Read Array List "+elapsedTime + ", start="+startTime+ ",
stop="+stopTime);
/*Read Array List 11 millisecondi,
start=1536829938254, stop=1536829938265*/
```

File

Un **file** è una **sequenza di dati, di dimensione grandi a piacere (idealmente infinita)**
Questa risorsa NON SI può tenere tutta in memoria

Alcuni fatti sui file:

- **I file sono stream di byte**
- **Alcuni file sono testo:** ossia contengono **solo caratteri ASCII** (e non byte qualunque) e sono fatti di linee di caratteri (di dimensione limitata), ossia sequenze di caratteri ASCII intramezzate da fine linea (‘/n’), se ben fatti
- **I file sono risorse non trasportabili tra processori, perché sono radicate sul disco su cui risiedono**; si può portarne il contenuto con opportuni applicazioni che possiamo codificare con algoritmi

File

Il modello è quello di UNIX
Un file come **risorsa che fa riferimento alla memoria permanente di un nodo**
Un file è memorizzabile su disco

Dettagli
Il file non contiene una fine del file (un carattere o una sequenza ad-hoc) ma un descrittore ne dice la **lunghezza**
Quando interagiamo con il file, **in lettura, le azioni hanno il modo di farci capire che siamo arrivati alla fine del file** e non possiamo leggere ulteriori caratteri, **tramite risultati specifici che ci fanno capire** che abbiamo raggiunto la fine del file (la condizione **EOF** non è una eccezione)

Primitive sui file e dispositivi

In generale, il modello di lavoro è quello di azioni richieste e attuate dal sistema di supporto

Il sistema per le azioni mette a disposizione **operazioni primitive** sui file e dispositivi che sono visti come **stream esterni ai linguaggi**

Il **modello è quello detto** **Open, Read / Write, Close**

che agiscono **in modo atomico sugli stream** e non in modo ***mediato da una libreria*** (vedi API precedenti)

Con altre azioni primitive, possiamo anche intervenire sui dispositivi e gestirli e controllarli in modo più granulare

(select(), ioctl(), fctl(),….)

ESEMPIO di DATI di SISTEMA (3)

La **tabella dei file descriptor** è **statica**

Scrivere un programma **che mostri il numero massimo di descrittori di I/O** che il processo può creare

Ad **ESEMPIO**:

```
while(true)
{       printf("Opening file n. %d ...\n", i);
        if((open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, 00640))<0)
        { perror("Error opening file \n"); exit(1); }
        i++;
}
```

- Arrivati al limite dei descrittori il sistema lancia 'Too many open files'

Una **volta eseguite le operazioni necessarie su un file aperto, è fondamentale chiudere il file** per liberarne il descrittore

```
while(true)
{   printf("Opening file n. %d ...\n", i);
    if((fileDescr=open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, 00640))<0)
    { perror("Error opening file \n"); exit(1);
    }
    /*… operations on file …*/
    printf("Closing file ...\n");
    close(fileDescr); i++;
}
```

SEMPLICITÀ DEGLI ALGORITMI

Semplicità degli algoritmi

- **Algoritmi semplici**

  Eliminare controlli ridondanti per ridurre i costi in eccesso
  (vedi funzioni sulle stringhe in C che **non** controllano formato)

- **Poco innestamento di procedure/funzioni non necessarie**

  Eliminare l'impegno sullo stack che viene anche da innestamento funzioni (ogni invocazione impegna il record di attivazione sullo stack)

- **Eliminare ripetizioni**

  Uso di **variabili** per **tenere lo stato di una esecuzione** invece di ripetere la computazione più volte, una volta per necessità (array che viene scorso una prima volta per contare occorrenze e poi una seconda volta viene scandito ogni volta per fare una azione per ciascuna di queste – pensare a 100.000 elementi e magari milioni di volte o più)

- Capacità di controllo della esecuzione
- Capacità di interazione con l'utente
- …

SEMPLICITÀ DEGLI ALGORITMI (1)

- Prendiamo come esempio l'algoritmo che calcola la sommatoria, realizzato in modo **ricorsiva** vs **iterativo**

```
int sommatoriaRic(int i)
{   if(i!=0)
      return i+sommatoriaRic(i-1);
    else return i;
}
```

```
int sommatoriaIt(int i)
{ int acc = 0, j=0;
    for(j=i; j>0; j--)
        acc=acc+j;
    return acc;
}
```

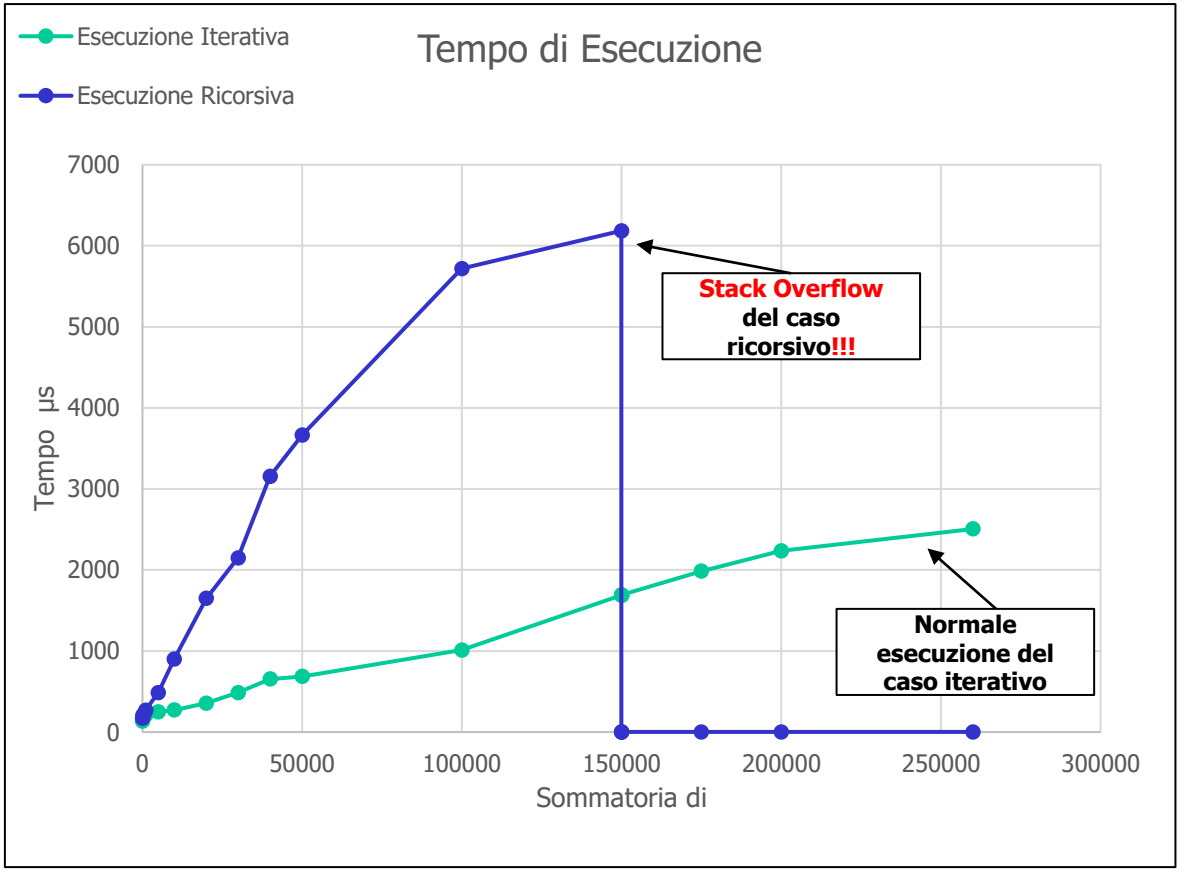
- Esiste un valore per il quale le due funzioni si comporteranno in maniera diversa: cosa succede alla versione **ricorsiva**?

SEMPLICITÀ DEGLI ALGORITMI (2)

Stampa dei risultati:

- Eseguendo diversi test durante i quali è stato aumentato il valore per il quale si deve calcolare la sommatoria, si ottiene la situazione del **grafico a lato**

- Oltre ai tempi, possiamo notare come il **caso ricorsivo riempie la risorsa stack** molto prima di quello iterativo, *a parità di valori da calcolare*



Funzioni in C per stringhe

Il C non definisce le funzioni per I/O, ma, dove possibile, le delega al sistema operativo per maggiore flessibilità

le strutture dati sono definite dal linguaggio, ma le funzioni sono lasciate al sistema operativo o alle librerie

Obiettivo: lavorare con stringhe sequenze di caratteri

- le stringhe sono acceduta tramite puntatori (char *)
- le stringhe di caratteri vengono memorizzate in **array di caratteri terminate dal carattere '\0'** (NULL - valore decimale zero)

Tutta una serie di funzioni su stringhe

int strlen(const char *s); char *strcat(char *dest, const char *src);

int strcmp(const char *s1, const char *s2);

char *strcpy(char *s1, const char *s2);

char *strstr(const char *ss, const char *s);

Le funzioni non **controllano le terminazioni** per efficienza e funzionano correttamente solo per stringhe ben fatte

USO MINIMO DELLE RISORSE

Minimizzazione dell'impegno sulle risorse

Nel caso di **programmi di sistema che devono essere ripetuti molte / moltissime volte** è molto importante avere il **controllo della risorse usate**

- Non ripetere **operazioni semplici** (usare variabili di stato come si fa con **stile imperativo**)
- Non **allocare e deallocare variabili all'interno di cicli, ma definirle all'esterno del ciclo**
- Non **lasciare memoria non utilizzata**

- Considerare bene i processi
- Capacità di controllo della esecuzione
- Capacità di interazione con l'utente

USO MINIMO DELLE RISORSE

- **ESEMPIO allocazione di memoria**: scrivere un programma che riempia lo spazio di Heap allocando dati (senza eseguire mai una **free**…)

```
struct Struttura { int a, b;};

int main(void) {
  struct Struttura *b;
  srand(time(NULL));
//random num gen
  for (;;)
  { b = malloc(sizeof *b);
    b->a=rand(); b->b=rand();
  }
  return 0;
}
```

```
struct Struttura { int a, b;};
int main(void) {
  struct Struttura *b;
  srand(time(NULL));
  for (;;)
  { free(b);
    b = malloc(sizeof *b);
    b->a=rand(); b->b=rand();
  }
  return 0;
}
```

Se vogliamo che il programma **non** fallisca dobbiamo **liberare memoria prima di allocarla** di nuovo

- Usare **dmesg** per verificare la memoria occupata dal processo:

  Out of memory: Kill process 5208 (o) score 803 or sacrifice child
  Killed process 5208 (o) **total-vm:14224868kB**, anon-rss:7300564kB, file-rss:0kB, shmem-rss:0kB

USO DELLE RISORSE PROCESSI

Minimizzazione dell'impegno sulle risorse

Nel caso di processi e programmi che interagiscono con l'utente

- Controllare gli **argomenti di invocazione** e procedere solo se corretti
- Verificare gli **input dell'utente** e non procedere in caso di errore (per non fare azioni inutili e costose)
- Consumare gli **stream (di input)** e le risorse in modo corretto
- Interagire **usando risorse di sistema limitate**
- Progettare bene i processi
- Capacità di controllo della esecuzione
- Capacità di interazione con l'utente
- …

USO DELLE RISORSE PROCESSI (1)

- **ESEMPI**
  - **Controllo Parametri**: per non avere incongruenze durante l'esecuzione di un programma occorre sempre controllare ***NUMERO***, ***TIPO*** **e** ***RANGE*** consentito dei parametri inseriti dall'utente
  - Per esempio scrivere un programma che accetti come input la dimensione **INTERA** del buffer da usare durante il trasferimento di un file

```
int num, dimBuffer=0;
if(argc!=2)
{  printf("Usage Error: %s DimBuffer\n", argv[0]); exit(1); }
…
//CONTROLLO SECONDO PARAMETRO INT
while( argv[1][num]!= '\0')
{    if ( (argv[1][num] < '0') || (argv[1][num] > '9') )
     {   printf("Secondo argomento non intero\n");
         printf("Usage Error: %s DimBuffer\n", argv[0]);
         exit(2);
     }
     num++;
}
dimBuffer = atoi(argv[1]);
```

USO DELLE RISORSE PROCESSI (2)

ESEMPI

– **Dimensionamento Buffer**: scrivere un programma che usi un buffer per il trasferimento di un file

```
#define dim 100
int nread, fdsorg, dimbuff, sd, …; char buff[dim]; …
if((fdsorg=open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, 00640))<0)
 { perror("Error opening file \n");  exit(1); }
while((nread=read(fd_sorg, buff, dimBuffer))>0) write(sd,buff,nread);
// Invio
```

– **Richiedere un intero all'utente in modo ripetuto fino a EOF durante l'esecuzione del programma (prevenire errore e ciclo infinito)**

```
int ok; char ch;
while ((ok = scanf("%d", &ch)) != EOF)
{  if (ok != 1) // errore di formato
   { //CONTROLLO
     do {c=getchar(); printf("%c ", c);}
     while (c!= '\n');
     printf("Inserisci un int), EOF per terminare: ");
     continue;
   }
// opera sull'intero
}
```

USO DELLE RISORSE PROCESSI (3)

Uso di **strutture dinamiche anziché statiche**

- Quando lo stack delle chiamate eccede i limiti di memoria predefiniti per lo stack del processo, si verifica un errore di tipo **Stack Overflow**
- Anche le chiamate a funzione utilizzano una risorsa di sistema limitata. La causa più comune di stack overflow è l'eccessiva profondità (chiamate innestate) o ricorsione eccessiva all'interno di un programma

Ad **ESEMPIO**:

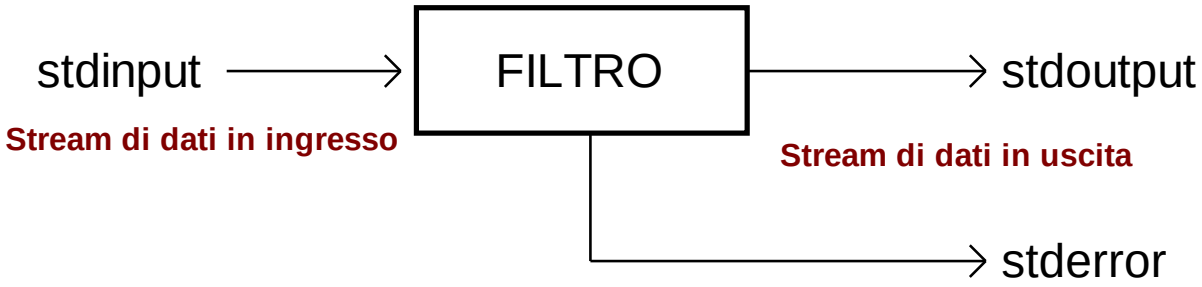
- Scrivere un programma che mostri il numero massimo di chiamate innestate consentito

```
void ricorsiva (int i)
{   printf("Nesting level: %d\n", i);
    ricorsiva(++i);
}

int main(int argc, char* argv[])
{   printf("Testing nested lvls\n");
    ricorsiva(0);
    printf("Test is over\n");
    exit(0);
}
```

Filtri (ridirezione e piping)

Un **filtro** è un modello di ordinato e ben fatto che prevede un **programma che riceve in ingresso da un input** e **produce risultati su output (uno o più)**



Il filtro deve consumare tutto lo stream in ingresso e portare in uscita il contenuto filtrato
Possibilità di ridirezione e di piping

```
comando  > file1 < file2
rev < file1 | sort | tee file | rev | more
```

Input/Output in C

C non definisce l'input/output ma lo prende dal **sistema operativo**, che prevede una **gestione integrata di I/O e dell'accesso ai file**

Per I/O, azioni di base del Sistema Operativo:
primitive di accesso **read() / write()**
con semantica di atomicità (ossia di mutua esclusione e di non interruzione)

Per I/O, anche libreria:

Input/Output **a caratteri**	**getc(), putc()**
Input/Output **a stringhe di caratteri**	**gets(), puts()**
Input/Output **con formato per tipi diversi**	**scanf(), printf ()**
Input/Output **con strutture FILE**	**fopen(), fclose ()**

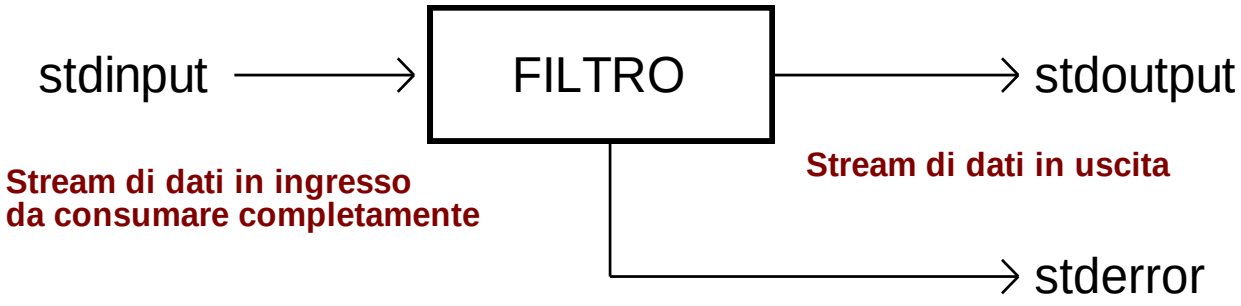
I/O azioni e API

L'input è sempre più complesso dell'output

Si deve **gestire** la **fine dello stream di Input,** ossia si deve gestire la **fine del file (che è un evento atteso e necessario, non una eccezione) e consumare sempre tutto l'input**

I programmi che non lo fanno sono scorretti e non possono essere usati facilmente in composizione

Per questo progettiamo sempre **programmi filtri (e processi) corretti che devono tenere conto delle risorse in gioco e favorirne la gestione con sequenze di azioni opportune (API)**



Input: protocollo

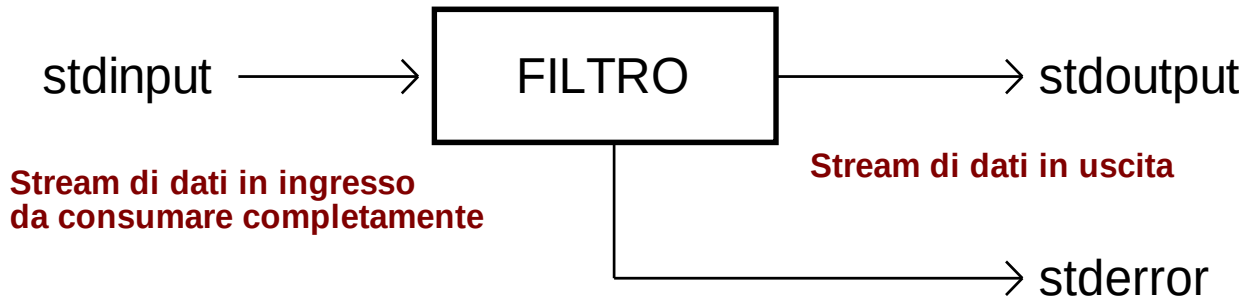
L'input è uno stream di dati (spesso file testo, ossia costituito da soli caratteri ASCII con fine linea, come quelli che derivano da tastiera)

Spesso i file che vengono dati come input sono di questo tipo

Protocollo di input

Il processo deve leggere l'input secondo le necessità applicative (a caratteri, a blocchi di caratteri, a sequenze) tipicamente in modo iterativo… **fino a non incontrare la fine dello stream**

Il non consumo di parte dello stream è assolutamente scorretto specie in catene o in composizione di processi



Funzioni di I/O a caratteri ASCII

int getchar(void); legge un carattere e **restituisce il carattere letto convertito in int** o **EOF in caso di end-of-file** o errore

int putchar(int c); scrive un carattere e restituisce il carattere scritto o **EOF** in caso di errore

Esempio: Programma che copia da input ad output:

```
#include <stdio.h>
main()
{ int c;
   while ((c = getchar()) != EOF)  putchar(c);
}
```

ATTENZIONE: La funzione getchar() (a causa della driver) comincia a restituire caratteri solo quando è stato battuto un carriage return (invio) e il sistema operativo li ha memorizzati

L'evento di terminazione è EOF == -1

Funzioni di I/O a stringhe

Obiettivo: fornire e lavorare con stringhe ben fatte

- le stringhe di caratteri vengono memorizzate in array di caratteri terminate dal carattere '\0' (NULL - valore decimale zero)

char *gets (char *s); legge una stringa e restituisce la stringa come indirizzo del primo carattere, se ok; **in caso di end-of-file o errore ritorna stringa nulla** (**0 ossia carattere NULL**)

int puts (char *s); scrive una stringa, in caso di errore restituisce **EOF**

Esempio (lo stesso di prima):

```
#include <stdio.h>
main() { char s[81];
while (gets(s))  puts(s); }
```

- gets sostituisce **new line** con NULL, puts aggiunge **new line** alla stringa

Le gets non dà problemi se si leggono caratteri all'interno della memoria fornita dall'utente (se a livello di Sistema Operativo si garantisce terminazione corretta, nessun problema)

I/O a stringhe: gets e fgets

char * gets (char *s);

legge una stringa e **non controlla** che la memoria sia congrua (OVERFLOW) in caso diamo troppi caratteri

char* fgets (char *s, int #car, FILE *f);
char* fgets (char *s, int #car, stdin);

legge una stringa dal file specificato, se errore o EOF, restituisce NULL

Se non forniamo il fine linea o terminiamo prima della dimensione specificata, la fgets riceve un carattere di meno e inserisce il carattere NULL all'ultima posizione (inserisce anche il '/n')

In caso di situazioni **in cui il sistema operativo controlli la massima sequenza possibile e tenendo conto di quella, la gets non è problematica**

Funzioni di I/O con formato

Si forniscono funzioni per la lettura/scrittura di dati formattati di ogni tipo con un numero variabile dei parametri (stringhe di formato tra "%"

int printf (char *format, expr1, expr2, ..., exprN);

scrive una serie di valori in base al format, restituendo il numero di caratteri scritti, oppure **EOF** in caso di errore

int scanf (char *format, &var1, &var2, ..., &varN);

legge una serie di valori in base alle specifiche contenute nella format memorizzando i valori nelle variabili passate per riferimento, e restituendo **il numero di valori letti e memorizzati**, oppure **EOF in caso di end-of-file**

(qui **EOF == -1, se inferiore ai valori attesi, problema**)

Esempi:

```
int k;
scanf("%d",&k);
printf("Il quadrato di %d e' %d",k,k*k);
```

Formati comuni (e %)

signed int **%d** %hd %ld unsigned int **%u** (decimale) %hu %lu

ottale **%o** %ho %lo esadecimale **%x** %hx%lx

float **%e** %f %g double **%le** %lf %lg

carattere singolo **%c** stringa di caratteri **%s**

puntatori (indirizzi) **%p**

*Per l'output dei **caratteri di controllo** si usano: '\n', '\t', etc.*

Per l'output del carattere '%' si usa: %% \% non funziona!

```
int a;
printf("Dai un carattere e ottieni il valore dec. \
   ottale e hex  "); a = getchar();
printf("\n%c vale %d in decimale, %o in ottale e %x in \
   hex.\n",a, a, a, a);
}
```

Errore lettura in scanf

scanf, in caso di errore di formato di lettura, ritorna una indicazione del **numero di variabili correttamente lette** e non **consuma lo stream di input** e **lascia il puntatore di I/O fermo** ad una **posizione iniziale** e …

Quindi il valore sbagliato rimane nell'input e deve essere **invece consumato**, specie in caso di ciclo di letture (tipicamente fino a fine linea)

Lettura di un intero (gestendo errore)

```
while ((ok = scanf("%i", &num1)) != EOF /* finefile */ )
{if (ok != 1) // errore di formato
 do {c=getchar(); printf("%c ", c);} while (c!= '\n');
 printf("Inserisci valore intero, EOF per terminare: ");
} …
```

In alternativa la può consumare i caratteri `gets(str);`

In genere scanf **non consuma il fine linea** (con formati interi, float, …) che va tolto **manualmente dall'input**

Primitive sui file e dispositivi

In generale, sono preferibili le **azioni primitive** sui file che agiscono **in modo atomico sugli stream** e non in modo *mediato da una libreria* (vedi API precedenti)

int read (int fd, char * buff, int len) legge i caratteri e ritorna il numero di caratteri letti. In caso di errore, restituisce un valore negativo.
In caso **di fine file restituisce 0**

int write (int fd, char * buff, int len) scrive i caratteri e ritorna il numero di caratteri scritti. In caso di errore, restituisce un valore negativo.

Con le azioni primitive, possiamo anche intervenire sui dispositivi e gestirli e controllarli in modo granulare (select(), ioctl(), fctl(),….)

Le azioni mediate attraverso le struttura dai **FILE** sono meno dirette e agiscono appunto attraverso la libreria del sistema operativo

fopen (), fread (), fwrite (), fgets (), fputs (), fscanf (), fprintf (), fclose()

APPENDICE A: Limiti di sistema

I sistemi operativi (**Unix** e Unix-like) hanno bisogno di memoria per gestire i file aperti e memorizzano un descrittore astratto (**file descriptor**) per accedere **a file e risorse di I/O** (p.e., **pipe** e **socket**)

Il seguente esempio mostra il numero massimo di descrittori che un processo può gestire

Nel sistema operativo ci sono file che riportano i limiti nell'utilizzo di alcune risorse:

- Il **file /etc/security/limits.conf** riporta le seguenti regole:
  - **<domain>** → username, group, o wildcard
  - **<type>**
    - *Hard* → impostati da super-user e imposti dal Kernel. L'utente non può modificare il vaore delle risorse messe a disposizione dal sistema
    - *Soft* → questi limiti possono essere impostati dall'utente nel rispetto dei corrispondenti limiti Hard impostati dal sistema
  - **<item>** → **fsize** (maximum filesize); **nofile** (maximum number of open files); **nproc** (maximum number of processes) …
  - **<value>** → il valore della rule
- Il comando per visualizzare e modificare (solo per sessione corrente) i limiti è **ulimit** (man ulimit)
  - **ulimit** -[Hard|Soft] <limit_name> <limit_value>