

Sistemi Operativi

Laurea in Ingegneria Informatica

Università di Roma Tor Vergata

Docente: Francesco Quaglia



Introduzione

1. Principi di base dei sistemi operativi
2. Prospettiva storica
3. Componenti di un sistema operativo
4. Schema di massima dell'organizzazione di sistemi Unix/Windows
5. Standard ed ambienti d'esecuzione
6. Concetti basici sulla sicurezza del software

Obiettivi di un sistema operativo

- **Semplicità**

Rende lo sviluppo del software più semplice, mascherando le peculiarità delle piattaforme hardware

- **Efficienza**

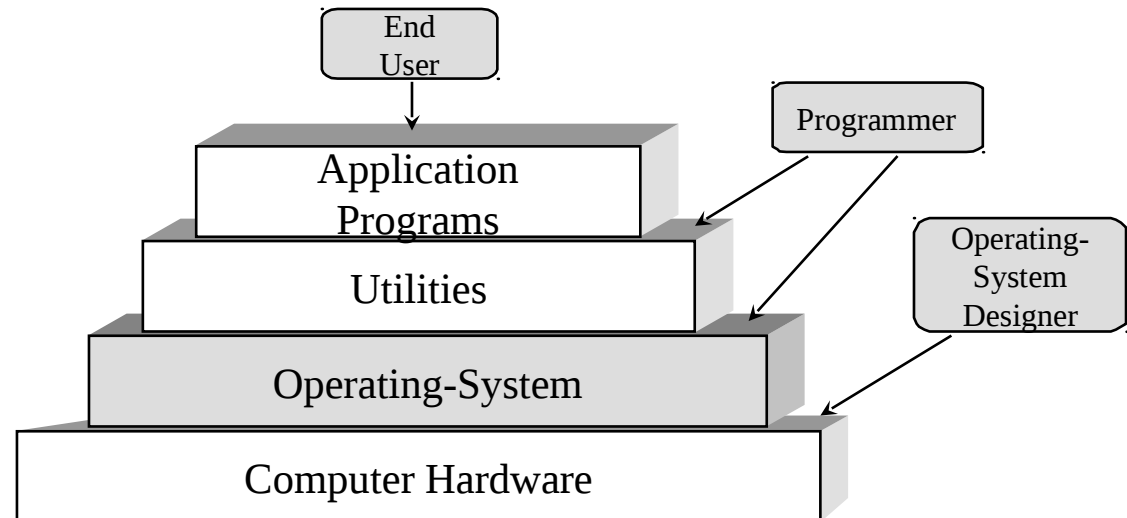
Permette l'ottimizzazione nell'uso delle risorse da parte dei programmi applicativi

- **Flessibilità**

Garantisce la trasparenza verso le applicazioni di modifiche dell'hardware, e quindi la portabilità del software

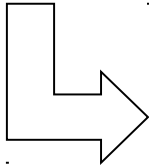
I sistemi operativi sono **componenti software**

Essi esistono perchè rappresentano **una soluzione ragionevole al problema dell'utilizzabilità** di un sistema di calcolo



Virtualizzazione delle risorse

- all'utente ed alle applicazioni vengono mostrate risorse virtuali più semplici da usare rispetto alle risorse reali
- la corrispondenza tra risorse virtuali e risorse reali è mantenuta dal SO in modo trasparente (mascherandone la struttura)
- le risorse reali nella maggioranza dei casi possono essere solo assegnate in uso esclusivo, limitando il parallelismo nella esecuzione delle applicazioni
- la disponibilità di una molteplicità di risorse virtuali, favorisce l'esecuzione concorrente di più applicazioni



Possibilità di soddisfare

- **molteplici utenti in simultanea**
- **utenti che utilizzano molteplici applicazioni in simultanea**

Elaborazione seriale (anni 40-50)

- sistema di calcolo riservato per l'esecuzione di un singolo programma (job) per volta
 - mansioni a carico dell'utente
 1. Caricamento del compilatore e del programma sorgente nella memoria
 2. Salvataggio del programma compilato (programma oggetto)
 3. Collegamento con moduli predefiniti
 4. Caricamento ed avvio del programma eseguibile
-

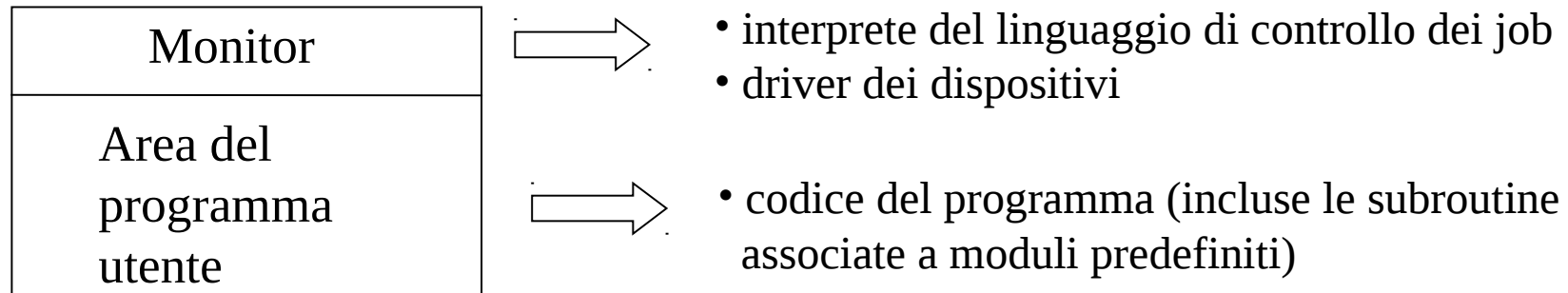
Migliorie

- sviluppo di librerie di funzioni (e.g. per l'I/O)
- sviluppo di linker, loader, debugger

Sistemi operativi batch (anni 50-60)

- tesi al miglioramento dell'utilizzo delle (costose) risorse hardware
- basati sull'utilizzo di un **monitor (set di moduli software)**
 1. Il programma è reso disponibile tramite un dispositivo di input
 2. Il monitor effettua il caricamento del programma in memoria e lo avvia
 3. Il programma restituisce il controllo al monitor in caso di terminazione o errore, ed in caso di interazione coi dispositivi
 4. Il monitor è residente in memoria di lavoro
 5. I moduli richiesti per uno specifico programma vengono caricati dal monitor come subroutine del programma stesso

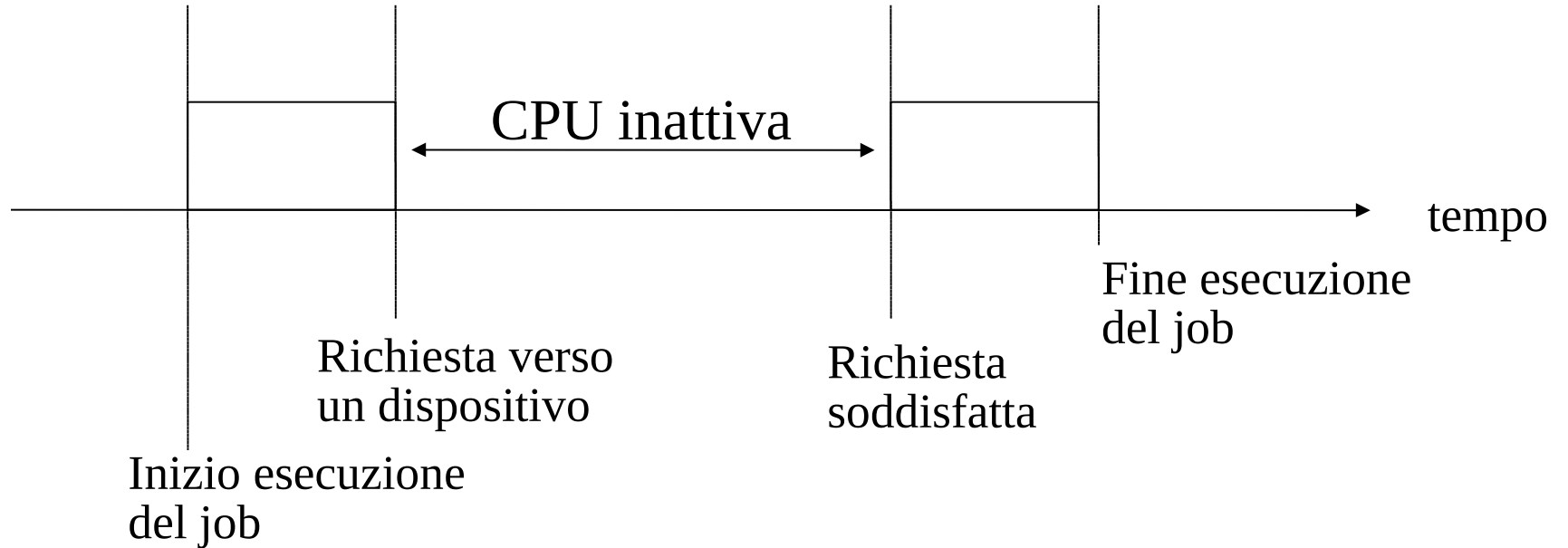
Ripartizione della memoria



Limiti principali dei sistemi batch

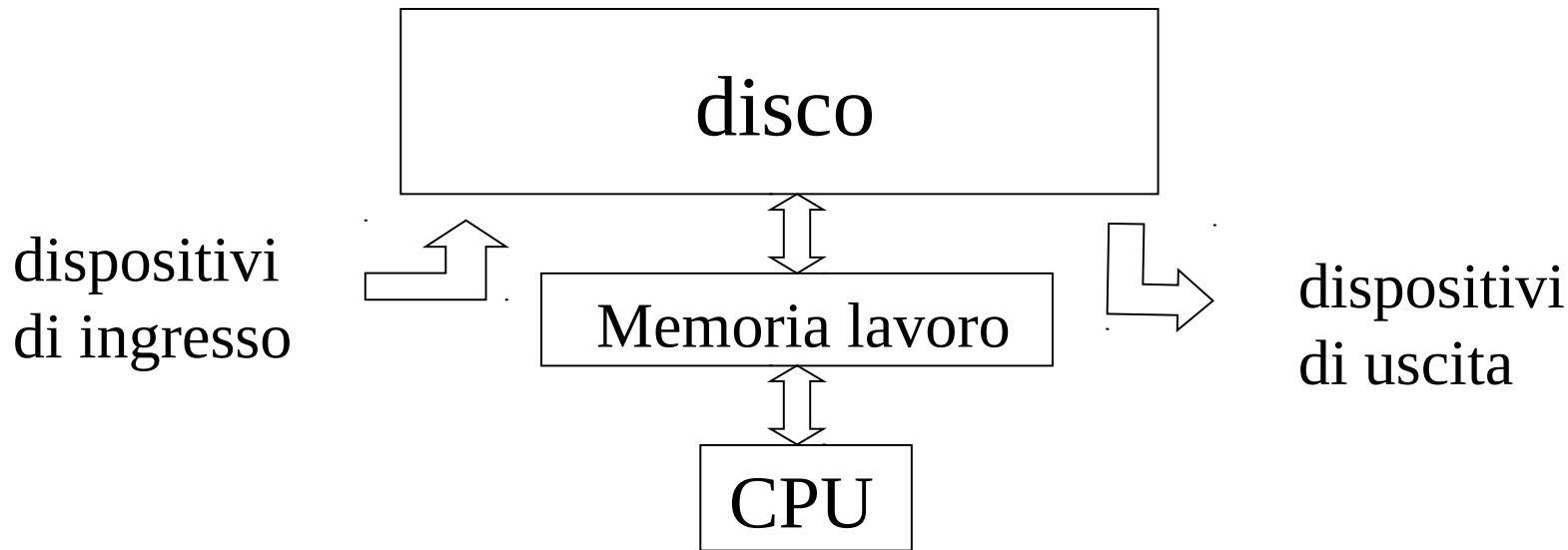
Monoprogrammazione

- un singolo job in esecuzione per volta
- sottoutilizzo della CPU dovuto alla lentezza dei dispositivi



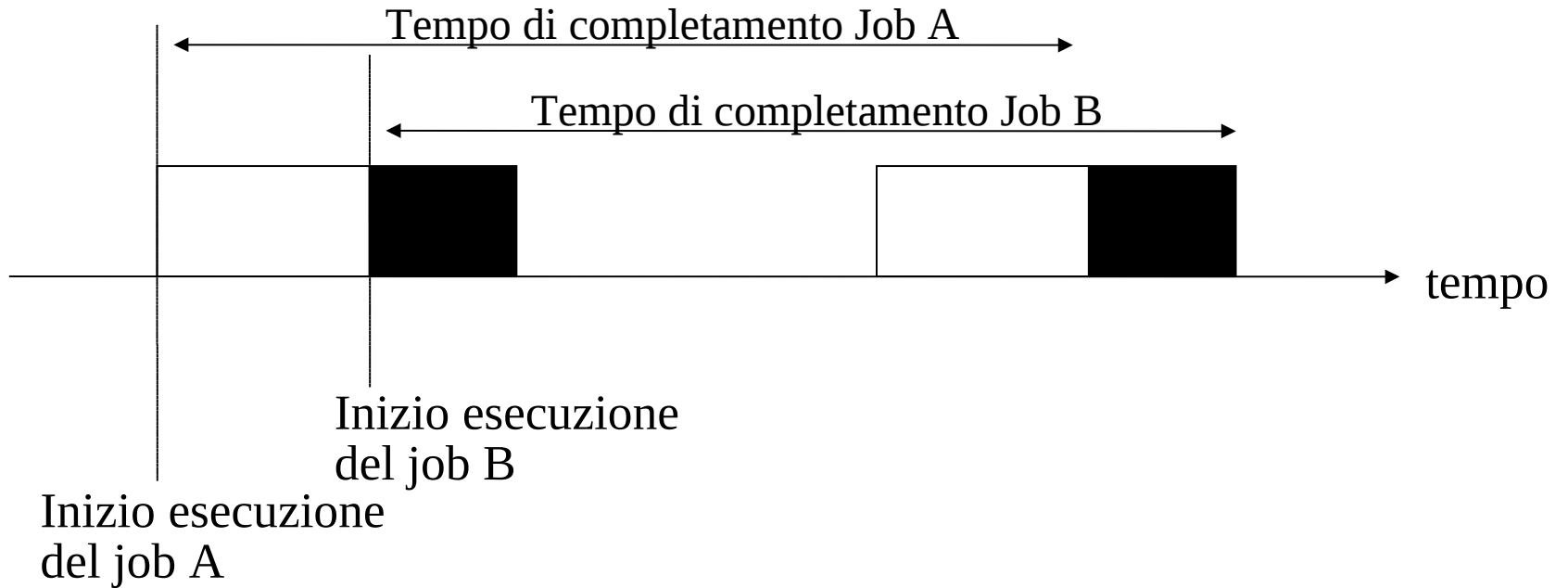
Spooling (simultaneous peripheral operation on-line)

- introduzione di una memoria disco (più veloce dei dispositivi) come buffer tampone per i dispositivi di input/output
- l'input viene anticipato su disco, l'output ritardato da disco
- riduzione della percentuale di attesa della CPU
- contemporaneità di input ed output di job distinti

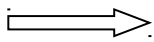


Sistemi batch multiprogrammati (multitasking)

- gestione simultanea di molteplici job
- un job per volta impegna la CPU
- più job possono impegnare contemporaneamente i dispositivi



Ipotesi



job A e job B inoltrano richieste a dispositivi diversi

Monoprogrammazione vs multiprogrammazione

	job1	job2	job3
Tipo	calcolo	calcolo+I/O	calcolo+I/O
Durata	5 min	15 min	10 min
Utilizzo disco	no	no	si
Utilizzo terminale	no	si	no
Utilizzo stampante	no	no	si

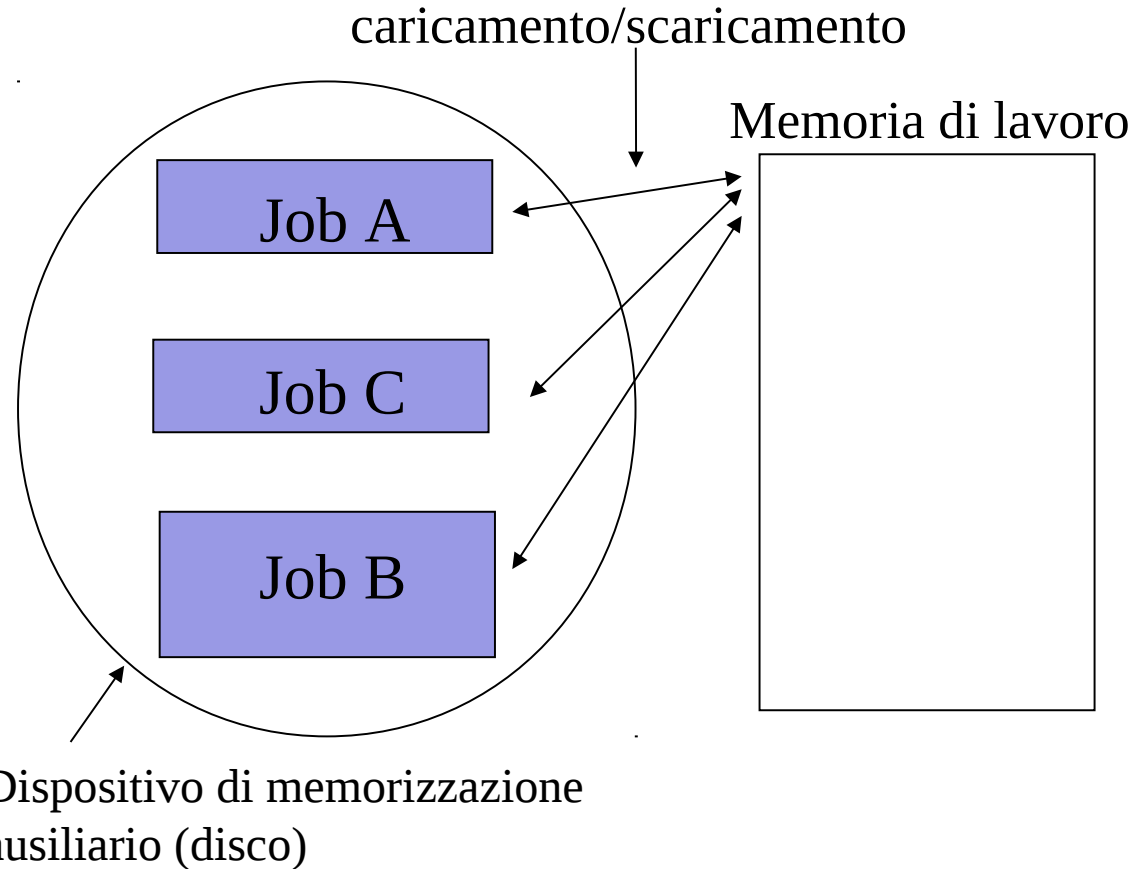
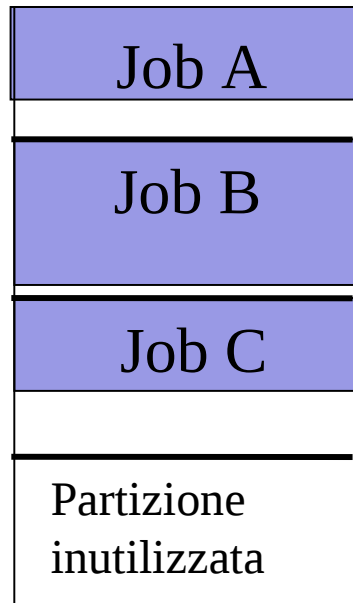
	monoprogrammazione	multiprogrammazione
Utilizzo processore	17%	33%
Utilizzo del disco	33%	67%
Utilizzo stampante	33%	67%
Tempo totale	30 min	15 min
Throughput	6 job/ora	12 job/ora
Tempo di risposta medio	18 min	10 min

Multiprogrammazione e memorizzazione dei job

La memoria di lavoro viene utilizzata per memorizzare i job secondo uno schema di

1. Partizioni multiple
2. Partizione singola

Memoria di lavoro



Limiti principali dei sistemi batch multiprogrammati

Il controllo viene restituito al monitor solo in caso di richiesta verso un dispositivo, terminazione o errore



- rischio di sottoutilizzo delle risorse
- l'esecuzione di job con frequenti richieste al dispositivo può essere penalizzata



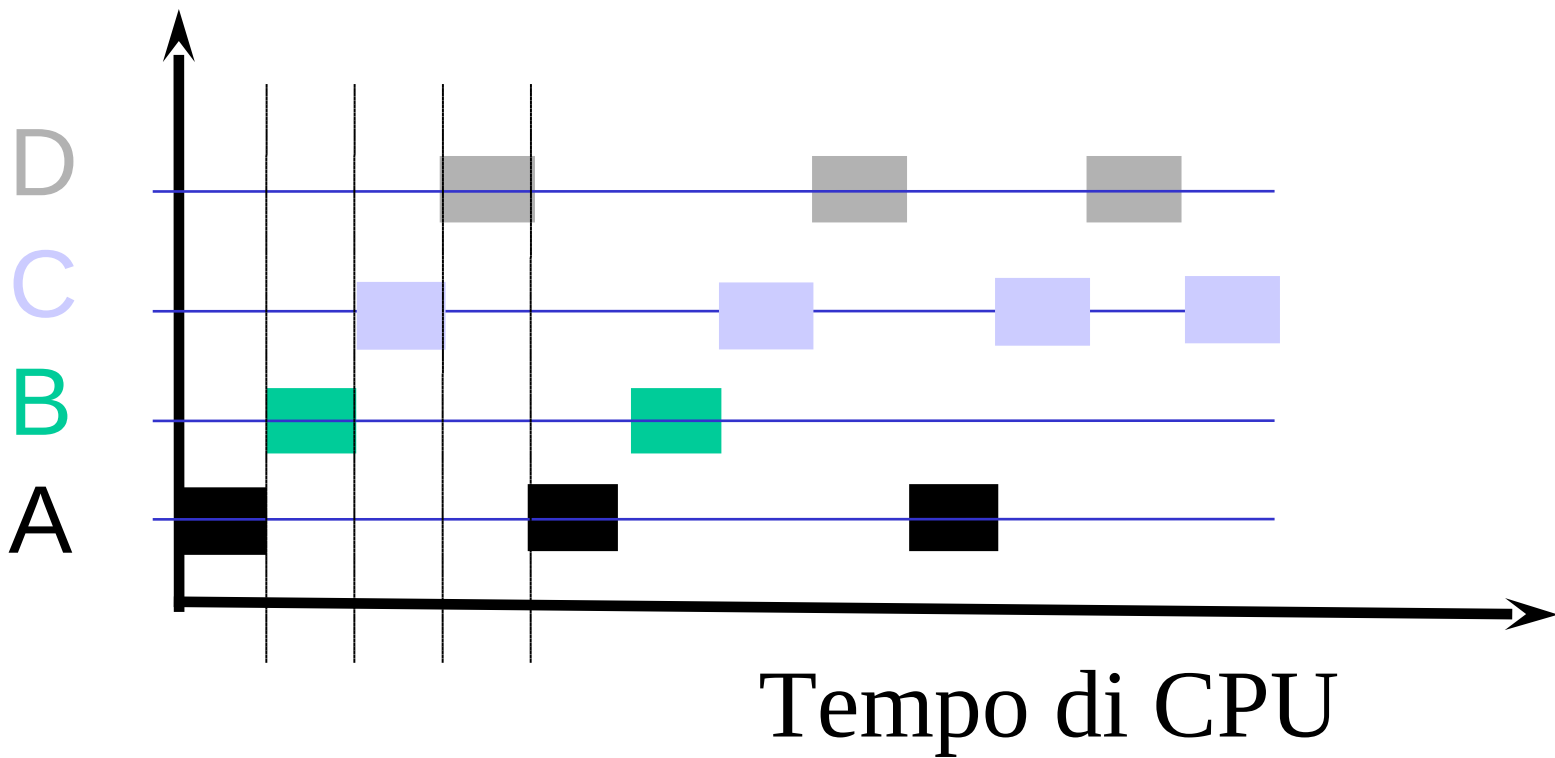
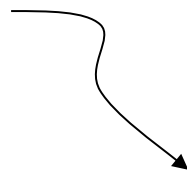
impossibilità di supporto ad applicazioni interattive

Sistemi operativi time-sharing (anni 60'/70')

- il tempo di CPU viene assegnato ai vari job secondo un determinato **algoritmo di scheduling stabilito dal monitor** (e.g. round-robin)
- è possibile che l'esecuzione di un job **venga interrotta** indipendentemente dal fatto che il job effettui una richiesta verso un dispositivo (**preemption**)

Esempio di round-robin

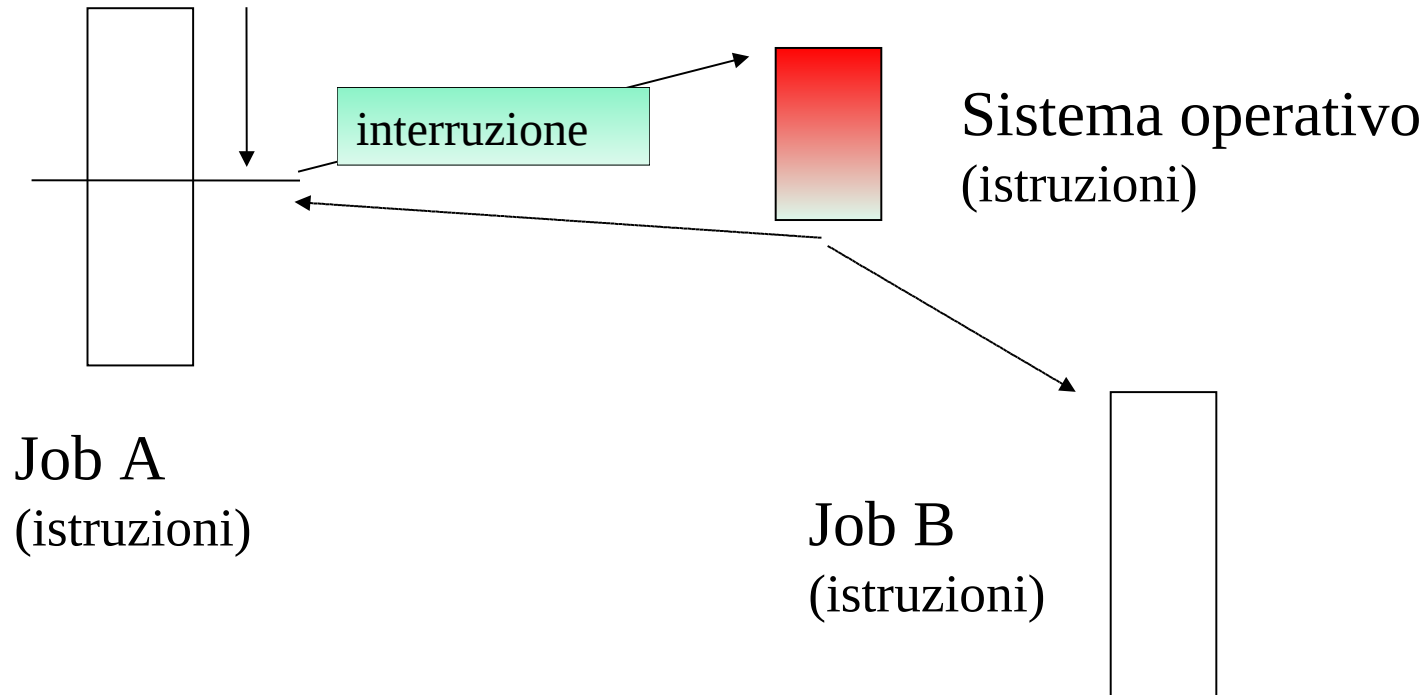
JOBs



Supporti al time-sharing

Interruzioni

- permettono il ritorno del controllo al monitor (ora chiamato sistema operativo)



Funzionalità delle interruzioni

- un'interruzione trasferisce il controllo ad una “routine di interrupt”
- tipicamente questo avviene attraverso un **interrupt vector**, che contiene gli indirizzi delle rispettive routine nel codice del sistema operativo
- un'architettura che gestisca gli interrupt deve fornire supporti per il salvataggio dell'indirizzo della prossima istruzione del programma interrotto (PC) e del valore dei registri di CPU
- ogni sistema operativo moderno è **interrupt driven**

Uno schema

Interrupt
(o trap)

Il firmware salva informazioni
“core” sullo stato della CPU in una
zona nota di memoria

Il controllo passa ad una
routine di gestione di livello
sistema (questa esegue il
salvataggio dello stato
rimanente di CPU se
necessario)

Il firmware associa
l'interrupt/trap alla routine
di gestione in base
all'interrupt vector – questo
è chiamato IDT (Interrupt
Descriptor Table) in
processori x86

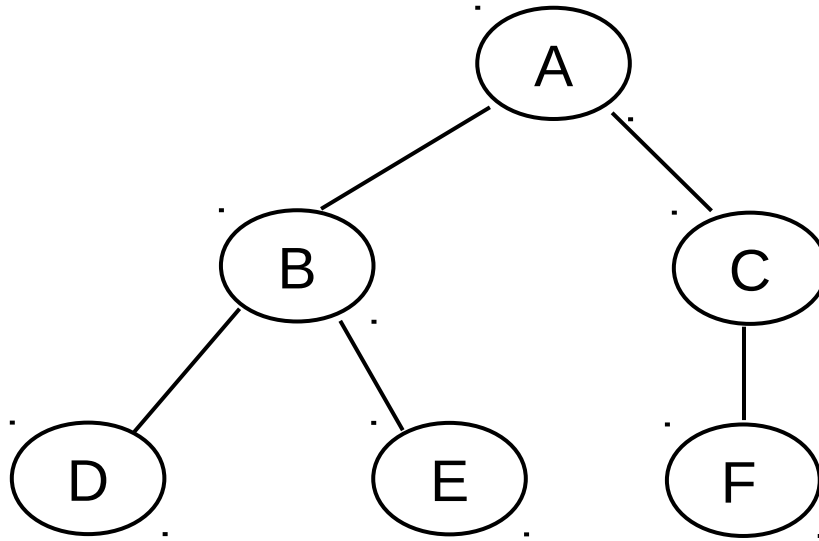
Sistemi real-time

- il sistema operativo deve far sì che un programma esegua specifici task entro dei limiti temporali prestabiliti (deadlines)
- **Hard real-time:** le deadlines devono essere assolutamente rispettate
 - utile per esempio per controllo di processi industriali
 - dati e istruzioni vengono tipicamente memorizzate in componenti ad accesso veloce o memorie a sola lettura (ROM)
 - tipicamente non supportato dai sistemi operativi time-sharing
- **Soft real-time:** le deadlines dovrebbero essere rispettate
 - utile in applicazioni che richiedono funzionalità di sistema avanzate (ad esempio applicazioni multimediali)

Processi

- introdotti per monitorare e controllare in modo sistematico l'esecuzione dei programmi
- un processo è un **programma in esecuzione** con associati:
 1. i dati su cui opera
 2. un **contesto** di esecuzione, ovvero le informazioni necessaria al sistema operativo per schedularlo
- il sistema operativo utilizza particolari strutture dati per mantenere tutta l'informazione relativa a ciascun processo, ed il suo stato corrente
- quando un processo va in esecuzione sulla CPU può **interagire** sia con il sistema operativo che con altri processi
- anche il sistema operativo può essere basato su un insieme di processi

Gerarchie di processi



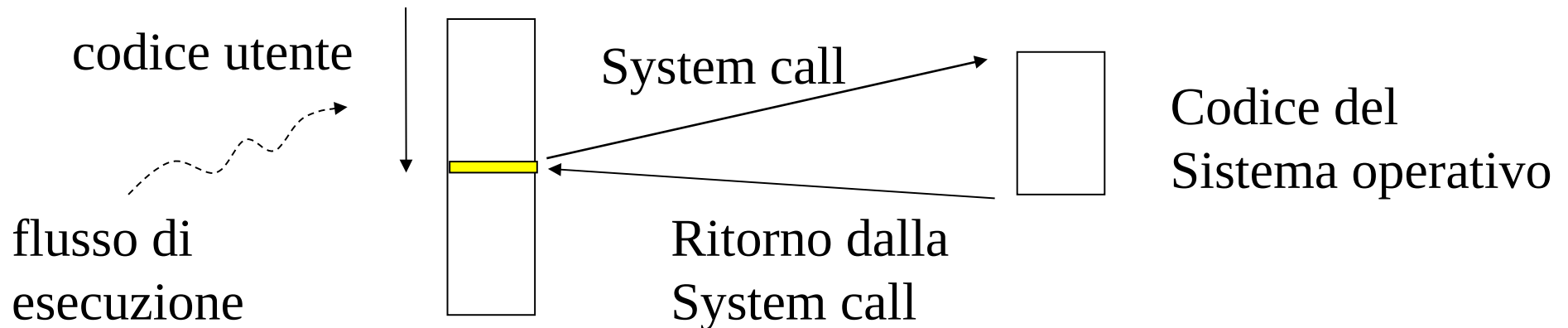
- un processo può creare altri processi (**child**)
- in base ai rapporti di parentela esistono gerarchie di processi

Servizi classici di un sistema operativo

- Gestione dei processi
- Gestione della memoria primaria e secondaria
- Gestione dei File
- Gestione dell'I/O (inclusi device di rete)
- Protezione delle risorse

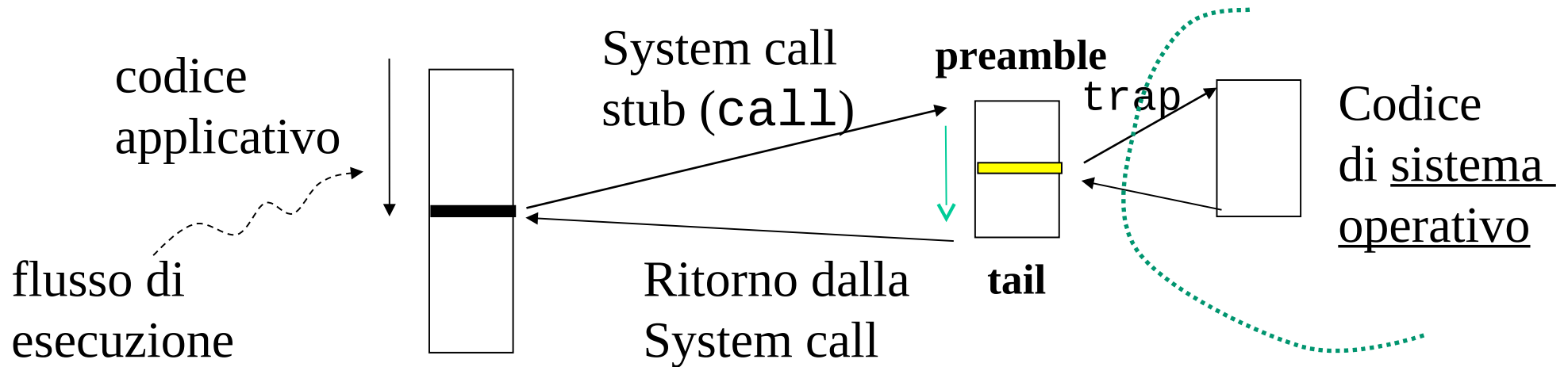
Accesso ai servizi di sistema: meccanismo delle “system call”

- il meccanismo delle “system call” fornisce un’interfaccia per l’accesso al software del sistema operativo
- originariamente tale interfaccia era accessibile solo per programmazione in linguaggio assembly
- alcuni linguaggi di alto livello (ad esempio C e C++) permettono l’invocazione diretta di una system call



Supporti per le system call

- il reale supporto per le system call sono le istruzioni di trap
- di conseguenza invocare a livello applicativo una system call implica la presenza nel codice applicativo di istruzioni di trap
- in tecnologia convenzionale (e.g. linguaggio C) questo avviene invocando “stub” offerti da librerie di programmazione



Anatomia di una system call

- viene mostrata al codice applicativo come una semplice funzione di libreria
- la sua reale implementazione è tipicamente “machine dependent”
- richiede quindi esplicita programmazione ASM per poter far uso di
 - ✓ istruzioni di trap
 - ✓ istruzioni di manipolazione dei registri

Un esempio

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



header including the system call specification

```
....
```

```
int main( ... ) {
```

```
....
```

```
syscall_name(....)
```

```
.....
```

```
}
```



```
syscall_name(....){  
    ...//preamble (partially asm)  
    asm ( machine code block )  
    ...//tail (partially asm)  
}
```

Codice C verso codice macchina

compile with “gcc -c -fomit-frame-pointer”
inspect with “objdump”

```
int f(int x){
```

```
    if (x == 1) return 1;  
    return 0;
```

```
}
```

```
000000000000000000 <f>:
```

```
  0:  89 7c 24 fc
```

```
  4:  83 7c 24 fc 01
```

```
  9:  75 07
```

```
  b:  b8 01 00 00 00
```

```
 10:  eb 05
```

```
 12:  b8 00 00 00 00
```

```
 17:  c3
```

```
    mov     %edi, -0x4(%rsp)
```

```
    cmpl   $0x1, -0x4(%rsp)
```

```
    jne    12 <f+0x12>
```

```
    mov    $0x1, %eax
```

```
    jmp    17 <f+0x17>
```

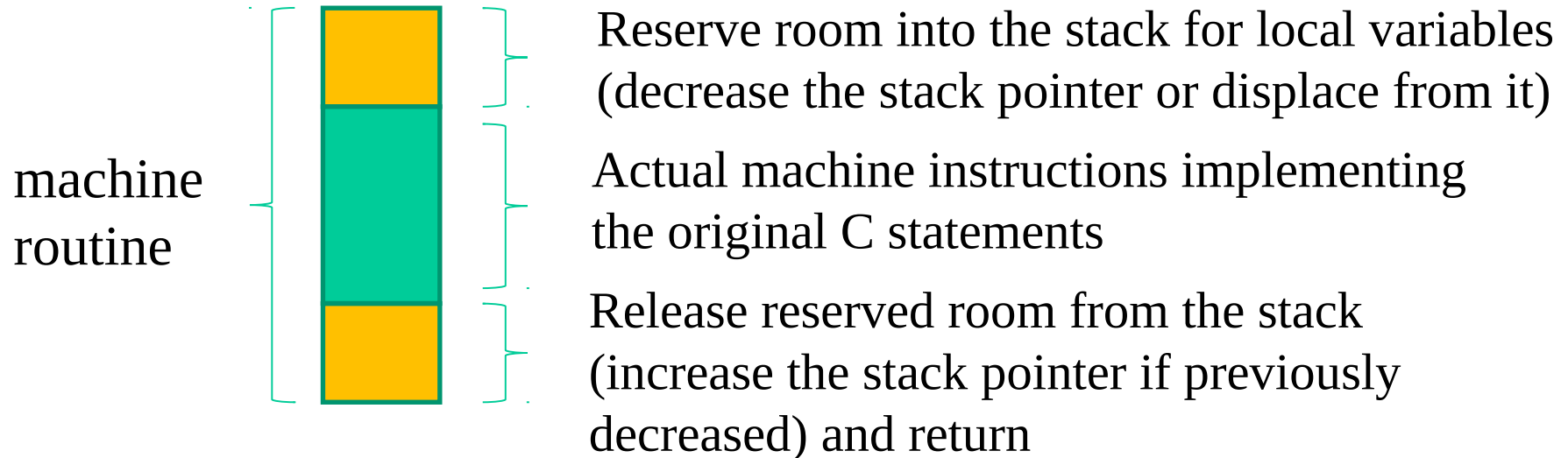
```
    mov    $0x0, %eax
```

```
    retq
```

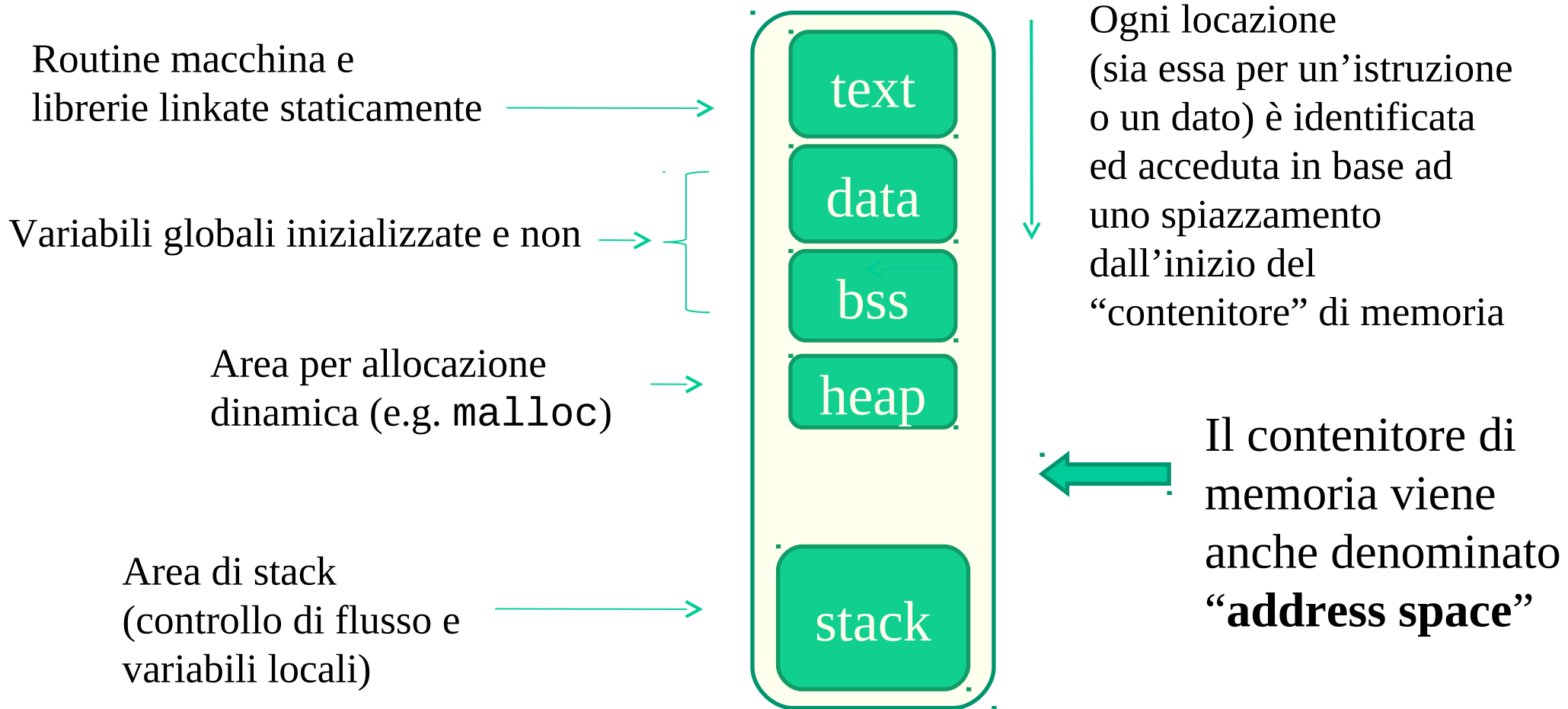
AT&T syntax

Alcune notazioni x86 e tipica strutturazione del codice macchina

- `rsp` – stack pointer
- `edi` – registro di CPU general purpose
- `retq` – control return to caller (PC saved into the stack)
- `mov` – data movement instruction
- many others



Visione globale della memoria accessibile ai programmi applicativi



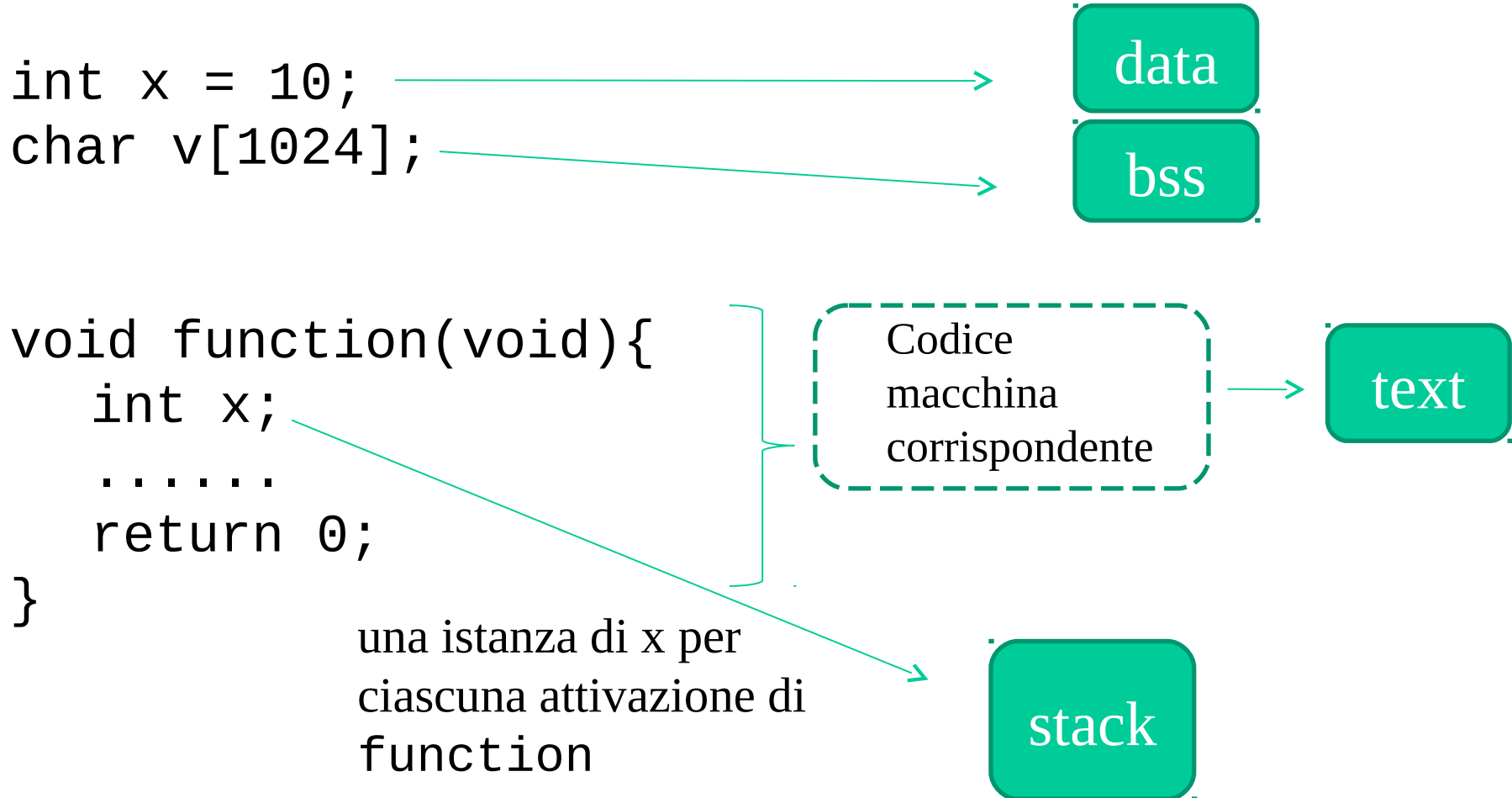
Formati

- In sistemi Unix la strutturazione delle varie sezioni per ogni singolo programma è specificata attraverso un formato chiamato ELF (Executable and Linkable Format)
- In sistemi Windows la strutturazione è specificata tramite un formato denominato PE (Portable Executable), basato su COFF (Common Object File Format), tramite il quale si descrive quindi il contenuto effettivo di un file exe
- E' compito dei tool di compilazione generare ELF/exe a partire dai sorgenti del programma e dalle direttive di compilazione fornite:
 - ✓ Specifica di quali moduli vanno inclusi in compilazione
 - ✓ Specifica di quali sezioni (ad esempio non di default) debbano essere incluse

Corrispondenze sorgente-eseguibile

- 1) ad ogni funzione C compilata e linkata staticamente corrisponderà un'unica routine di istruzioni macchina collocata nella sezione "testo"
- 2) ad ogni variabile globale dichiarata dal programmatore o all'interno di librerie linkate staticamente corrisponderà una locazione (di taglia appropriata) nella sezione "dati"
- 3) la sezione "heap" permette uso di ulteriore memoria, ad esempio tramite la libreria `malloc`
- 4) in tal caso è la stessa libreria `malloc` che tiene traccia di quali aree (buffer) dell'heap sono stati consegnati in uso all'applicazione o rilasciati dall'applicazione
- 5) la sezione "heap" è espandibile fino alla saturazione dell'intero contenitore di memoria dell'applicazione

Un semplice esempio



Variabili puntatore

- sono collocate nello spazio di indirizzamento esattamente con le stesse regole di variabili non puntatore (ad esempio interi, floating point)
- possono registrare il valore di un indirizzo di memoria
- tale indirizzo identifica un punto qualsiasi dello spazio di indirizzamento e rappresenta quindi un semplice spiazzamento all'interno del contenitore di memoria dell'applicazione

```
int *x;  
float *y;  
double *z;
```

```
void function(void){  
    x = y = z;  
}
```

abilitato di default in gcc
(si inibisce con, e.g.
-Werror)

Aritmetica dei puntatori

- i puntatori possono essere coinvolti in espressioni in cui si definisce un indirizzo di memoria come spiazzamento a partire da essi
- l'aritmetica dei puntatori definisce la modalità di calcolo del riferimento a memoria in un'espressione "indirizzo" in base alla tipologia di puntatore
- il valore dello spiazzamento sarà funzionale alla tipologia di puntatore

```
int *x;  
double *y;
```

```
void function(void){  
    x = y;  
    if ( x+1 == y+1 ) return 1;  
    return 0;  
}
```

Predicato mai verificato



Classico utilizzo dei puntatori

- per notificare ad una funzione dove “consegnare” o “leggere” in memoria le informazioni
 - ✓ ad esempio, `scanf ()` consegna informazioni in memoria usando puntatori come parametri di input
- per scandire la memoria e accedere/aggiornare i valori
- per accedere ad informazioni nell'heap
- data un'espressione “indirizzo” gli operatori principali per accedere all'effettivo contenuto di memoria sono 2:
 - * indirazione (prefisso all'espressione)
 - [] spiazzamento ed indirazione (suffisso all'espressione)

Array e puntatori

- il nome di un array di fatto corrisponde ad una espressione indirizzo, al pari di una variabile puntatore
- questa espressione ha un valore che corrisponde all'indirizzo di memoria dove l'array è collocato
- **NOTA:** un array puo essere collocato in parti differenti dello spazio di indirizzamento (data, stack ..) dipendendo da come esso è dichiarato
- i nomi di array a differenza dei puntatori sono **RVALUES**
- ovvero, essi (i nomi!!!) non hanno nessuna locazione di memoria associata (non sono delle variabili)
- i nomi di array non possono quindi comparire in espressioni di assegnazione come destinazioni

Alcuni esempi

```
int *x;
```

```
int y[128];
```

```
x = y;
```

```
x = y+10;
```

```
*x = *(y+10);
```

```
*(y+10) = *x;
```

assegnazioni lecite

```
y = x;
```

assegnazioni non lecite (y è un rvalue!!!)

Richiami su scanf

- vuole una stringa di formato che identifica un numero generico di informazioni da consegnare al chiamante e la relativa tipologia
- per ogni informazione da consegnare specificata come parametro dalla stringa di formato, `scanf ()` vuole il relativo indirizzo di memoria dove effettuare la consegna
- la specifica dell'informazione da consegnare determina il numero di byte che verranno consegnati al rispettivo indirizzo, il quale può anche non essere noto (come per le stringhe)

```
int x;  
  
void get_input_value(void){  
    scanf( "%d" , &x );  
}
```

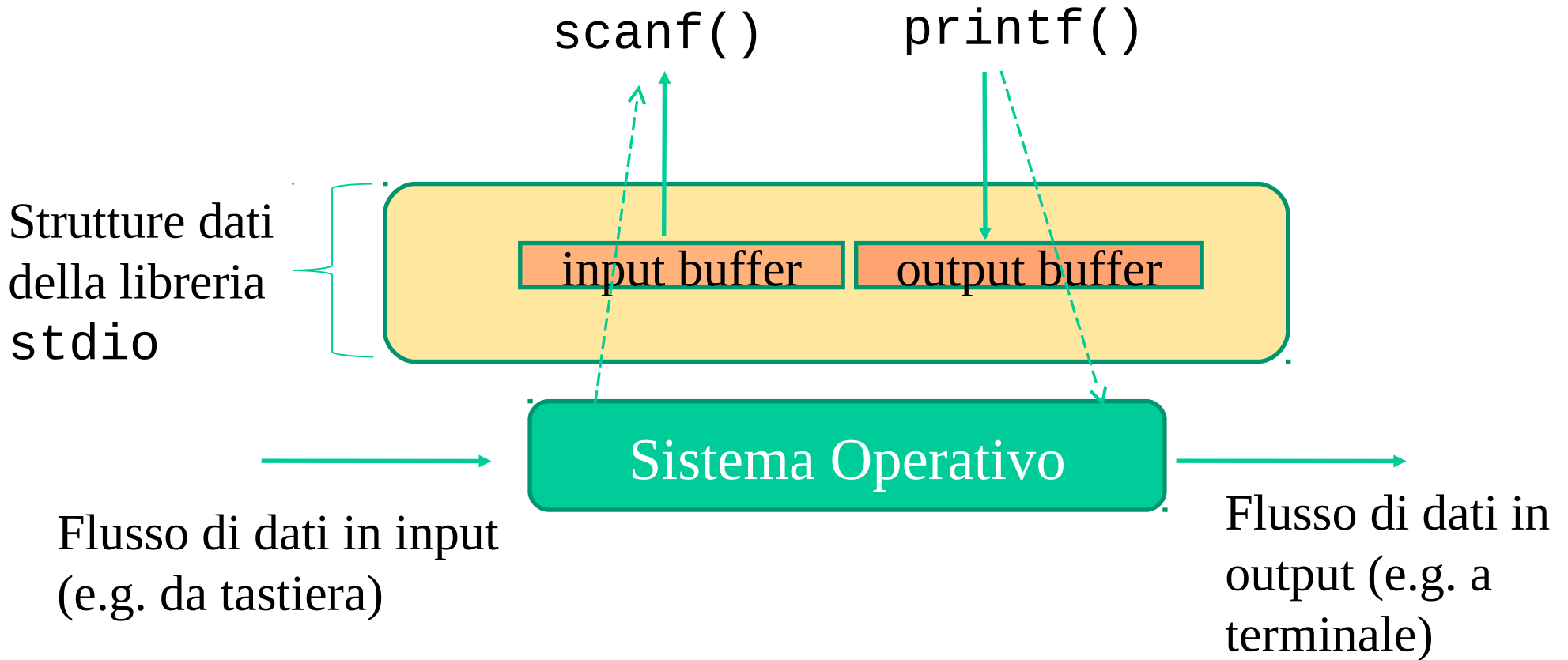
Richiami su scanf

- **NOTA:** il valore di ritorno di scanf () indica il numero delle informazioni realmente consegnate, in base all'ordine di consegna definito dalla stringa di formato!!!
- la funzione scanf () lavora secondo uno schema bufferizzato
- ovvero i dati che consegna sono preventivamente prelevati dalla sorgente tramite specifiche system call offerte dal sistema operativo, e bufferizzati in aree di memoria gestite dalla stessa libreria di I/O
- dati bufferizzati e non ancora consegnati rimangono validi per prossime consegne
- c'è possibilità di iterazione degli errori di formato!!!

Richiami su `printf`

- utilizza una stringa di formato per definire la struttura del messaggio da produrre – anche in questo caso secondo uno schema bufferizzato!!!
- la stringa di formato può contenere argomenti
- per ogni argomento, un ulteriore parametro è richiesto in input da parte di `printf()`
- questo parametro definisce l'espressione il cui valore dovrà comparire al posto del corrispondente argomento nel messaggio da produrre
- **NOTA:** il valore di ritorno di `printf()` corrisponde al numero di byte effettivamente costituenti il messaggio prodotto
- C standard codifica il tipo `char` in ASCII (American Standard Code for Information Interchange) – 1 byte per carattere

Lo schema di bufferizzazione



“Svuotare” il buffer di input/output

- in linea di principio potrebbe essere utilizzato il servizio `fflush()` offerto dalla libreria `stdio`
- però in C questo servizio ha un comportamento definito solo per quel che riguarda lo svuotamento del buffer di output (denominato `stdout`)
- per il buffer di input (denominato `stdin`) conviene ridefinire il comportamento di tale servizio in modo esplicito e funzionale al proprio scopo
- a tal fine si può utilizzare la direttiva di precompilazione `#define`
- Un esempio per l'eliminazione di un'intera linea di input dal buffer

```
#define fflush(stdin) while(getchar() != '\n')
```

Strutture (struct)

- sono un costrutto del C che permette di accorpare in un'area di memoria informazioni di natura eterogenea
- un impiego classico è quello di creare una tabella di informazioni eterogenee in termini di tipologia e quantità di memoria richiesta per la loro rappresentazione
- nel caso di accesso ai campi della `struct` tramite puntatore si può utilizzare l'operatore '`->`' suffisso all'espressione indirizzo
- nel caso si conosca il nome della variabile di tipo `struct` si può accedere ai campi con l'operatore '`.`' suffisso al nome della variabile

Un esempio

```
#include <stdio.h>

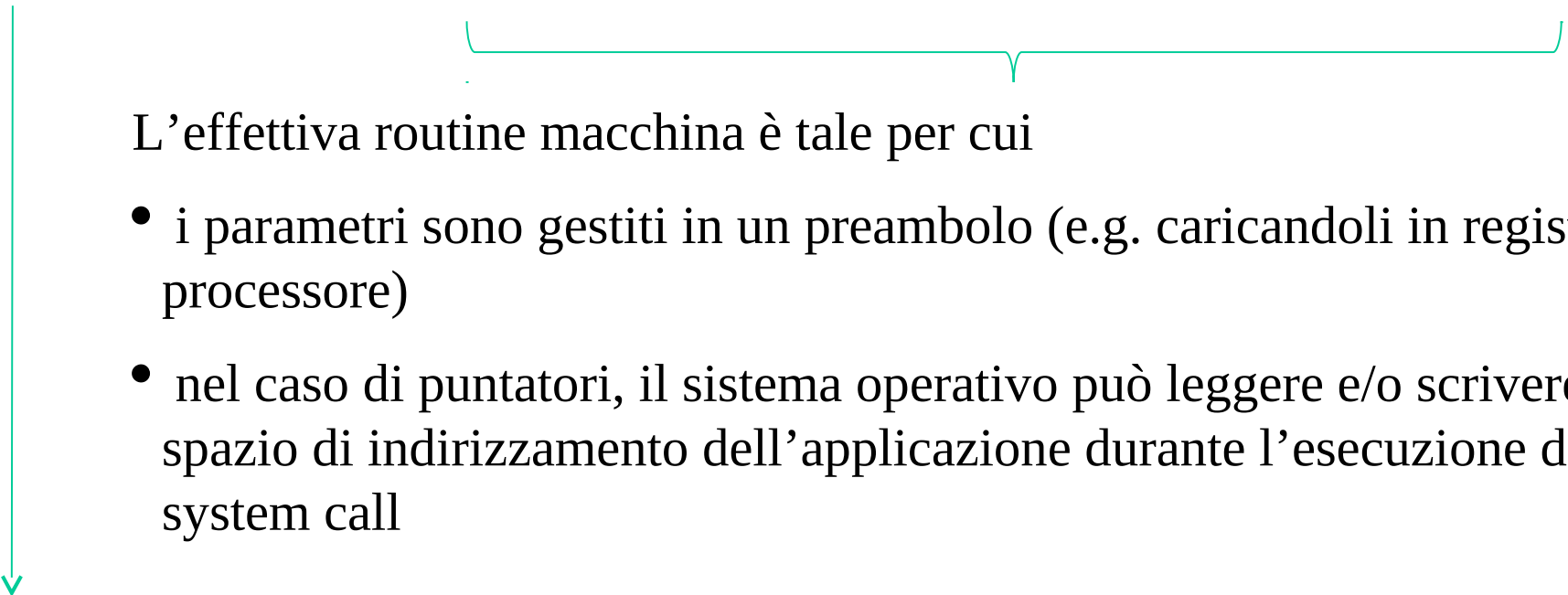
typedef struct _table_entry{
    int x;
    float y;
} table_entry;

table_entry t;
table_entry *p = &t;

void main(void){
    printf("please provide me with an INT and a FLOAT\n");
    scanf("%d %f",&(t.x),&(t.y));
    // scanf("%d %f",&(p->x),&(p->y)); //equivalent to previous line
    // scanf("%d %f",&((&t)->x),&((&t)->y)); //again equivalent
    printf("x is %d - y is %f\n",t.x,t.y);
}
```

Tornando allo “stub” delle system call

```
int syscall_name(int , void *, struct struct_name * ...)
```

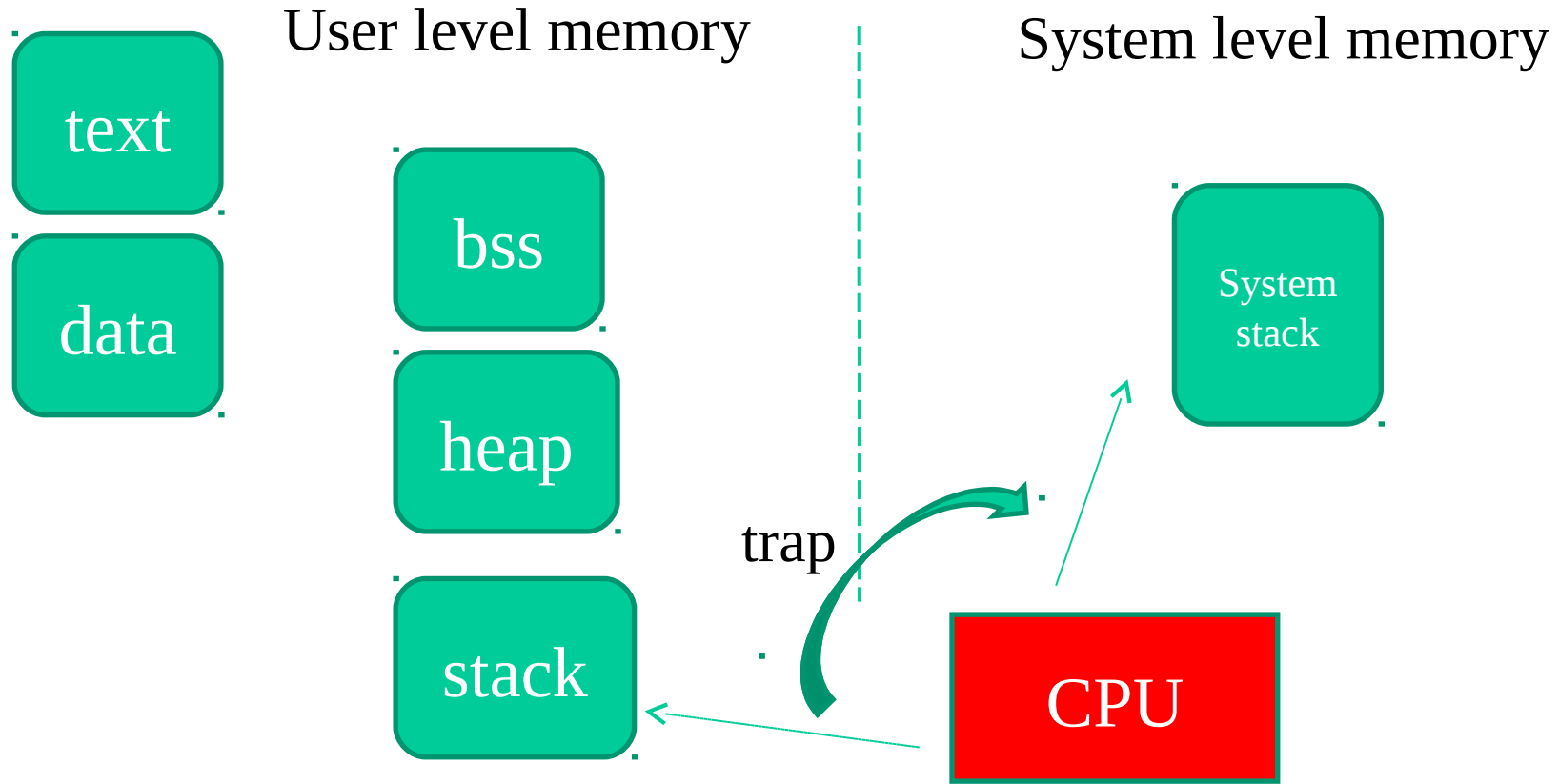


L'effettiva routine macchina è tale per cui

- i parametri sono gestiti in un preambolo (e.g. caricandoli in registri di processore)
- nel caso di puntatori, il sistema operativo può leggere e/o scrivere nello spazio di indirizzamento dell'applicazione durante l'esecuzione della system call

- ✓ il valore di ritorno è definito dalla “coda” della routine macchina
- ✓ questo generalmente dipende dal valore scritto dal sistema operativo in qualche registro di processore

Stack di sistema



Il cambio di stack avviene quando uno stub di system call esegue l'istruzione di trap che passa il controllo al sistema operativo

Librerie standard: gli standard di linguaggio

- gli standard di linguaggio definiscono servizi standard per la programmazione in una data tecnologia (e.g. ANSI-C)
- questi sono offerti dalle librerie standard, basate su specifica di interfaccia e semantica di esecuzione
- ogni funzione di libreria invocherà a sua volta le system call (necessarie per la sua mansione) proprie del sistema operativo su cui si opera
- permettono la portabilità del software su piattaforme di natura differente



Standard di sistema

- gli standard di sistema definiscono i servizi offerti da uno specifico sistema per la programmazione secondo una data tecnologia (e.g. linguaggio C)
- per sistemi Unix abbiamo lo standard Posix
- per sistemi Windows abbiamo le Windows-API (WinAPI)
- lo standard per una famiglia di sistemi definisce tipicamente
 - 1) il set delle system call offerte ai programmatori (tramite libreria)
 - 2) il set di funzioni di libreria specifiche per quel sistema (che a loro volta possono appoggiarsi sulle system call native del sistema in oggetto)

Kernel di un sistema operativo

- con la denominazione kernel si intende l'insieme dei moduli software di base di un sistema operativo
- ad esempio il kernel contiene
 - i moduli accessibili tramite le system call
 - i moduli per la gestione interrupt/trap
- il sistema operativo vero e proprio mette a disposizione moduli software aggiuntivi a quelli interni al kernel, come ad esempio programmi per l'interazione con gli utenti (**command interpreters**)
- comunemente tali programmi si appoggiano a loro volta sui moduli del kernel

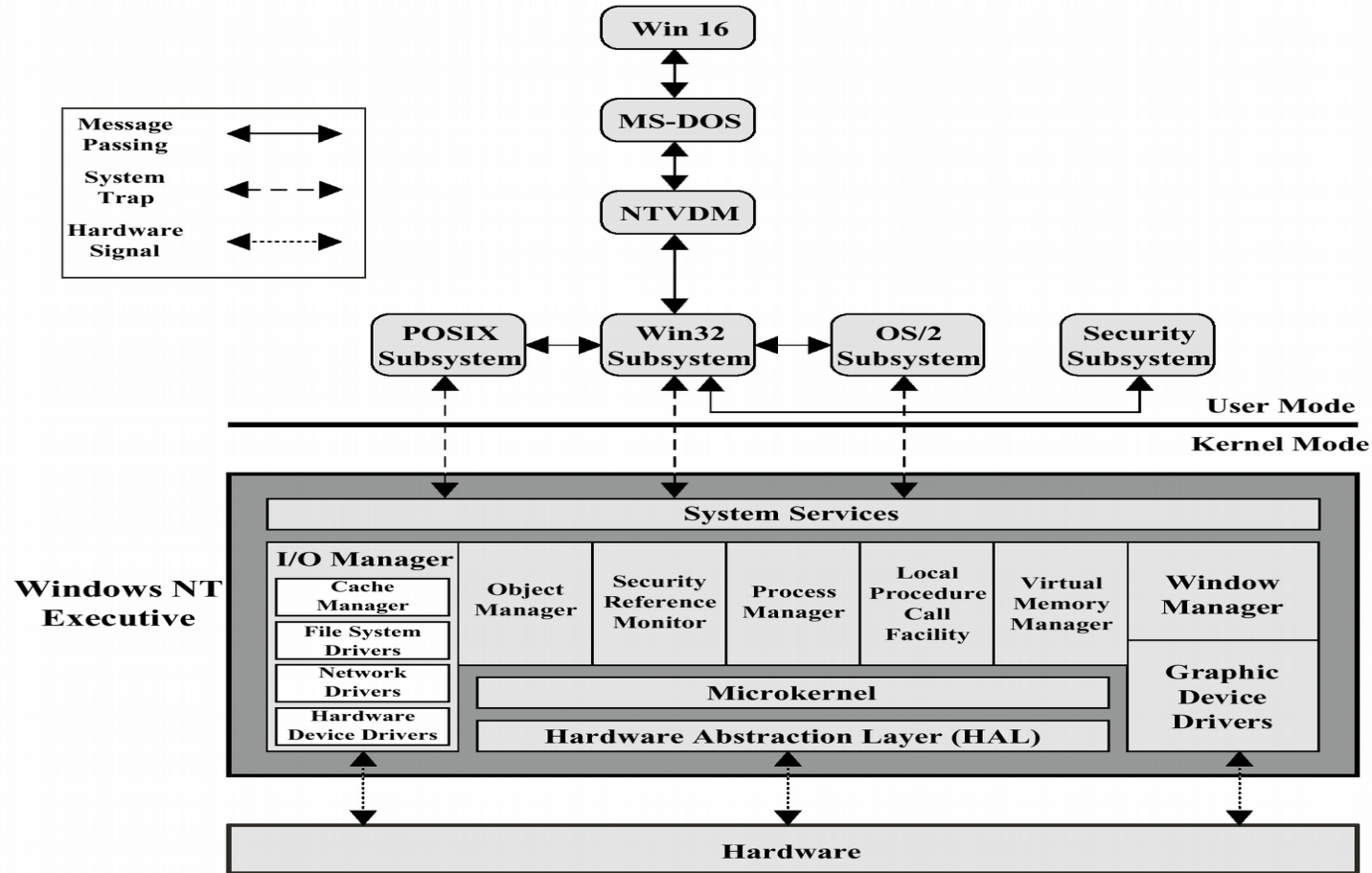
Architettura a microkernel

- solo un insieme ristretto di funzionalità sono implementate nel **kernel**:
 - gestione delle interruzioni
 - gestione basica della memoria
 - comunicazione tra processi
 - scheduling
- gli altri servizi del Sistema Operativo sono realizzati da processi che girano in “modo utente”
- maggiore flessibilità, estensibilità e portabilità
- minore efficienza a causa della maggiore frammentazione

Windows NT/2000/.../7/10/...

- NT fu sviluppato da Microsoft per superare Windows 3.1 ancora basato sul DOS
- disponibile su varie piattaforme (Intel, DEC Alpha)
- stessa interfaccia di Windows 95/98
- single-user multitasking
- Windows 2000 con architettura NT ma multi-user
- supporta piattaforme multiprocessore
- supporta applicazioni sviluppate per altri sistemi operativi, DOS, POSIX, Windows 95
- architettura orientata agli oggetti
- architettura microkernel modificata

Architettura di NT



Windows NT Executive

- identifica il **kernel** del sistema operativo
- **Hardware Abstraction Layer**: strato di disaccoppiamento fra Sistema Operativo e piattaforma hardware
- **Microkernel**: scheduling, gestione delle interruzioni, gestione del multiprocessing
- **Executive Services**: moduli relativi ai vari tipi di servizi offerti dall'Executive
- **I/O Manager**: gestisce code di I/O sulla base delle richieste dei processi
- **Windows Manager**: gestisce l'interfaccia grafica a finestre
- **System Services**: interfaccia verso i processi che girano in modalità user

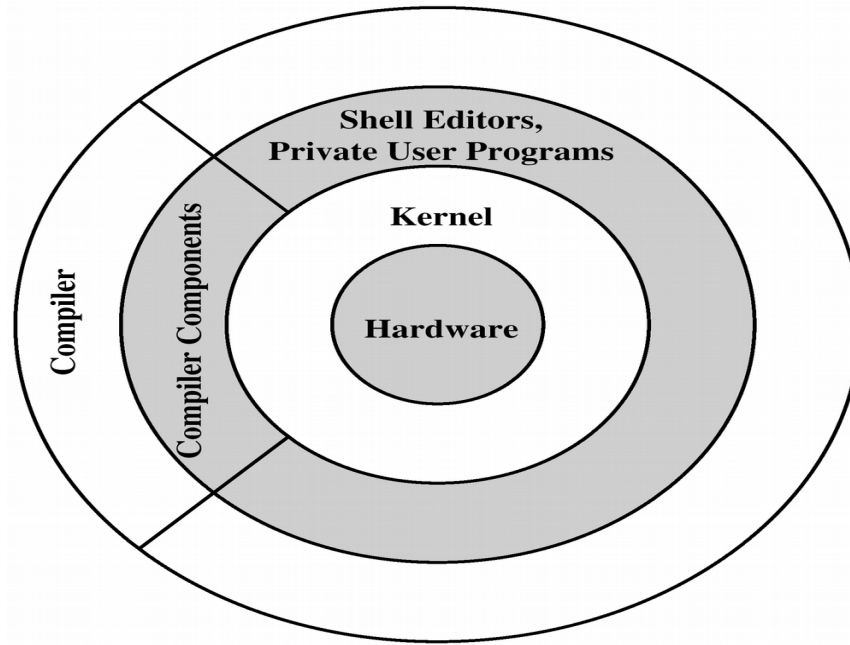
Sottosistemi d'ambiente

- NT era strutturato supportare applicazioni scritte anche per altri sistemi operativi
- interfacce multiple verso le applicazioni
- garantisce la portabilità del software
- **Win32**: API di Windows 95 e NT
- **MTVDM**: MS-DOS NT Virtual DOS Machine
- **OS/2**: (in memoria di ...)
- **Win16**: Windows a 16 bit
- **POSIX**: interfaccia standard di chiamate di sistema, basata su UNIX e supportata da vari sistemi operativi

UNIX

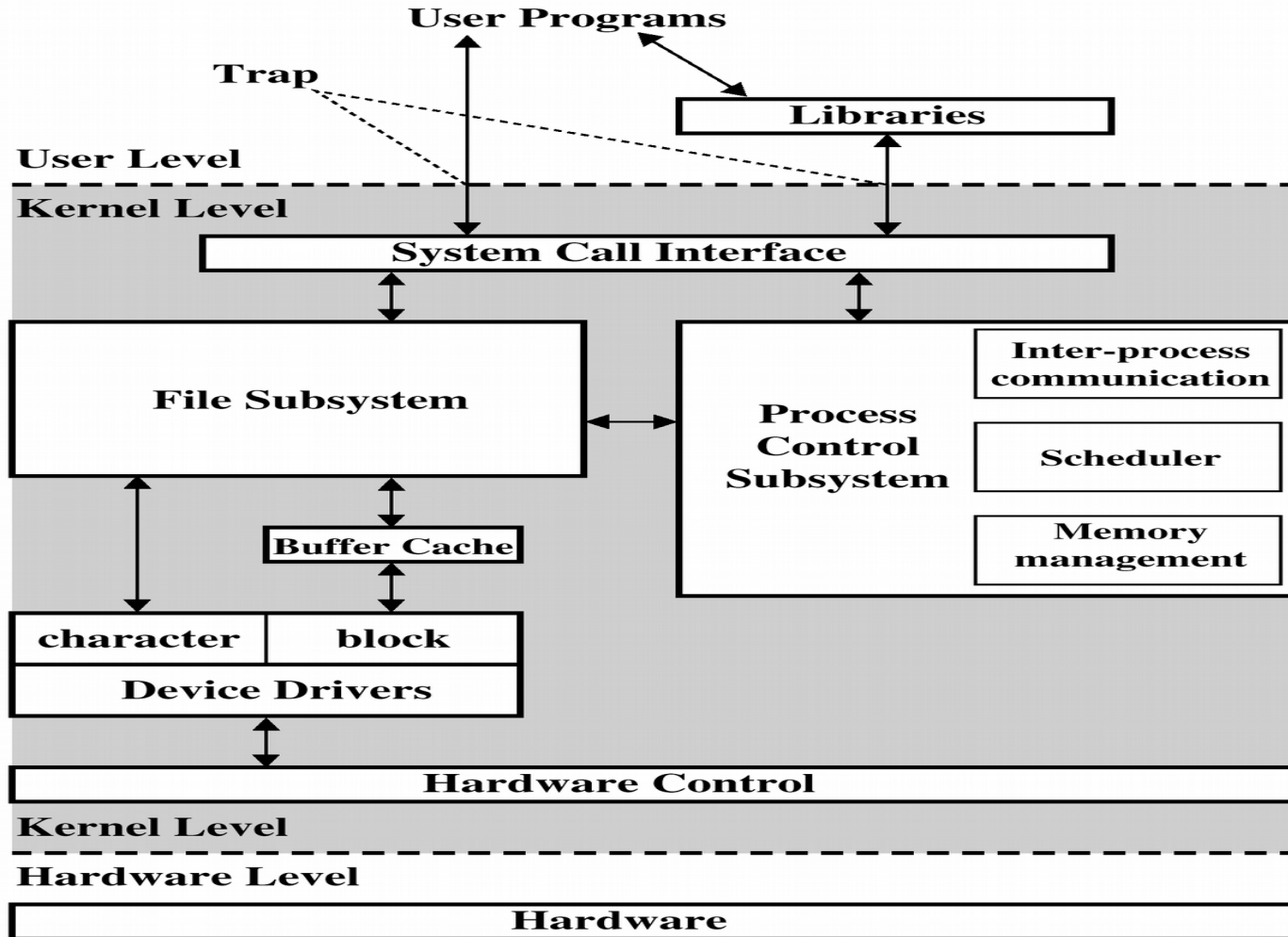
- sviluppato da ATT nei Bell Labs su piattaforma PDP 7 (1970)
- riscritto completamente in linguaggio C (appositamente definito)
- codice di dominio pubblico (per alcune versioni)
- architettura aperta, cioè indipendente dalla piattaforma
- diffusione iniziale (gratuita) negli ambienti universitari
- versione UNIX BSD di Berkely
- si afferma come piattaforma aperta di riferimento
- sistema operativo (inizialmente) tipico dei server e delle workstation di fascia alta

Architettura di UNIX



- struttura stratificata
- il Kernel costituisce il sistema operativo vero e proprio
- molte parti del sistema vengono eseguite tuttavia in modalità user

Architettura di riferimento del Kernel



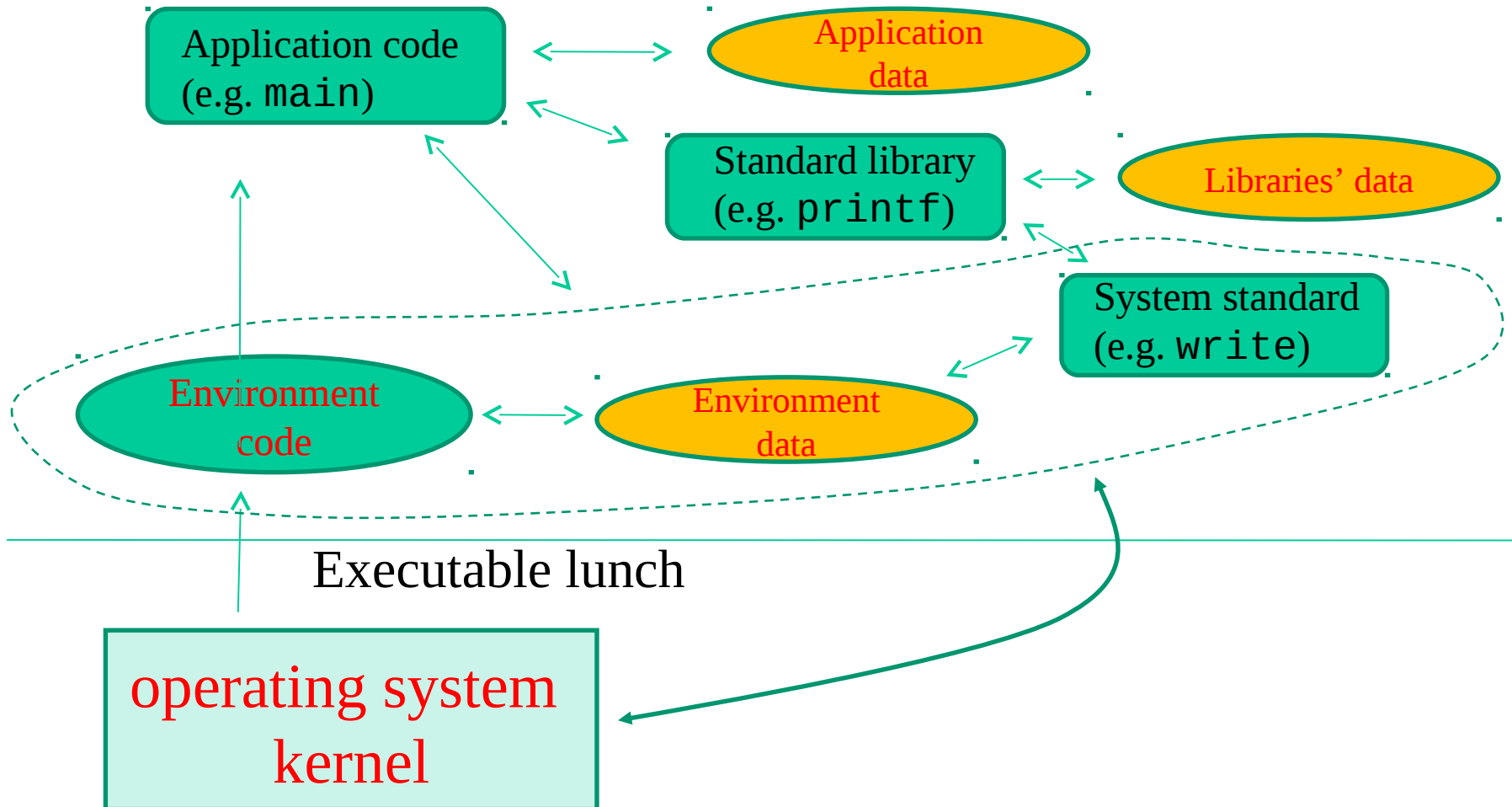
Il concetto di ‘ambiente’

- tipicamente i programmi applicativi vengono compilati secondo regole ‘canoniche’ tali da generare eseguibili formati da:
 - il codice applicativo vero e proprio (e.g. `main`)
 - un set di altri moduli software denominati ambiente di esecuzione
- per sistemi Unix e toolchain di compilazione convenzionale il modulo software che prende il controllo quando un eseguibile è lanciato non è il `main`
- esso è una funzione speciale denominata `_start`
- le funzioni di ambiente eseguono (anche) task preliminari atti a far sì che il codice applicativo esegua poi correttamente, e sotto certe precondizioni

Utilizzo dell'ambiente

- raccordo tra il sistema operativo ed il codice applicativo allo startup di una applicazione
 - ✓ infatti codice ANSI-C (o ISO-C) può essere compilato ed eseguito su sistemi operativi differenti proprio grazie al collegamento (in compilazione e linking) con ambienti differenti
- come mezzo per pilotare l'esecuzione di specifiche funzioni di libreria (tipicamente quelle afferenti allo standard di sistema)
 - ✓ infatti l'ambiente è formato da una collezione di moduli software e anche da informazioni di stato (valori di variabili e locazioni di memoria)

Uno schema



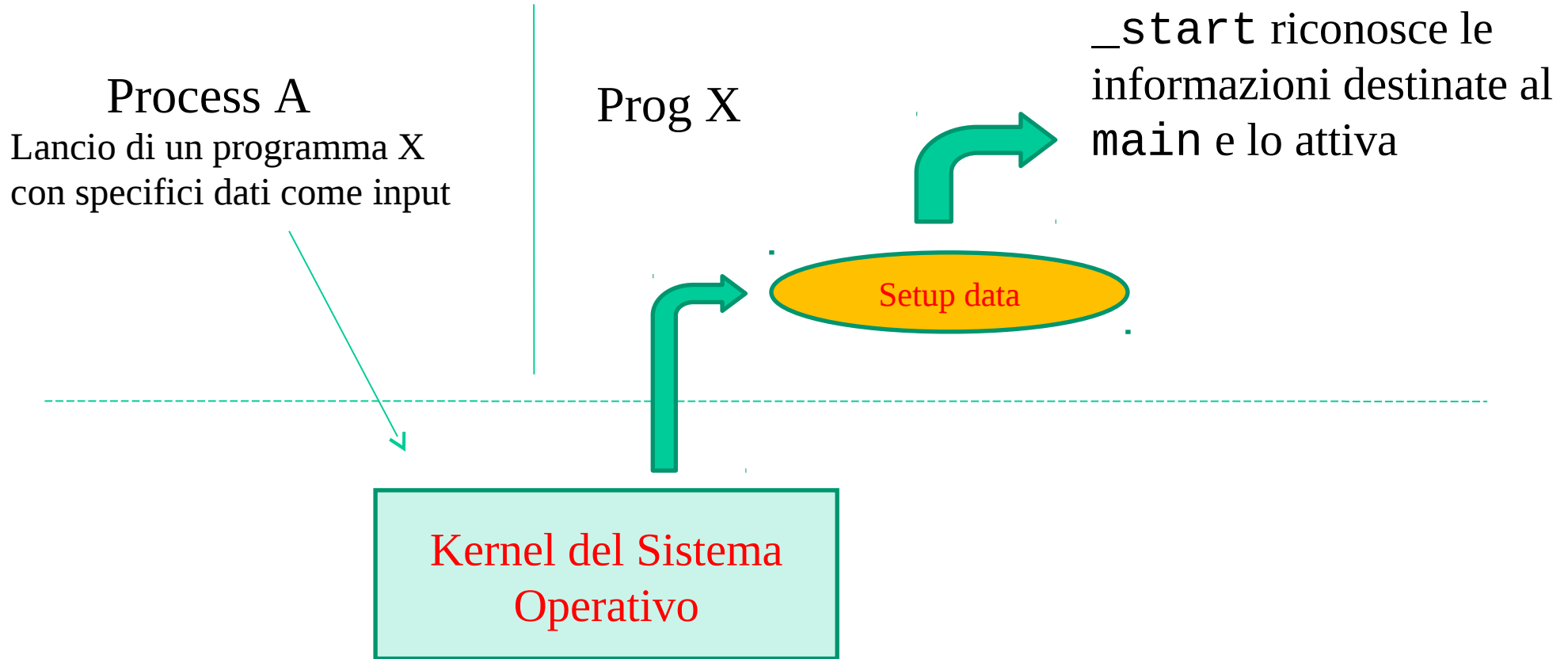
Prescindere dall'ambiente

- in principio è possibile
- è quindi anche possibile codificare un ambiente ex-novo
- è possibile anche compilare ed eseguire programmi C che non includono il modulo `main`
- ad esempio, su sistemi Posix se il programma include una funzione `_start` allora questa verrà identificata in compilazione come la funzione di partenza utilizzando il flag `-nostartfiles`
- altrimenti la funzione identificata come quella di startup sarà la prima presente nella sezione `.text` dell'eseguibile

Funzionalità principali dell'ambiente

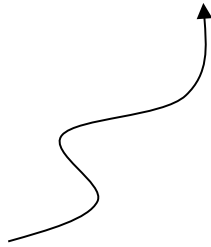
- raccordo tra il kernel del sistema operativo ed il `main` per quel che concerne il passaggio di parametri al codice applicativo
- infatti, come vedremo in dettaglio, su ogni sistema operativo lanciare uno specifico programma implica eseguire una system call apposita
- questà ricevera in input informazioni che definiscono quali e quanti parametri passare
 - ✓ al codice applicativo vero e proprio
 - ✓ all'ambiente di esecuzione in cui esso vive

Uno schema



Parametri della funzione main

main (int argc, char *argv[])

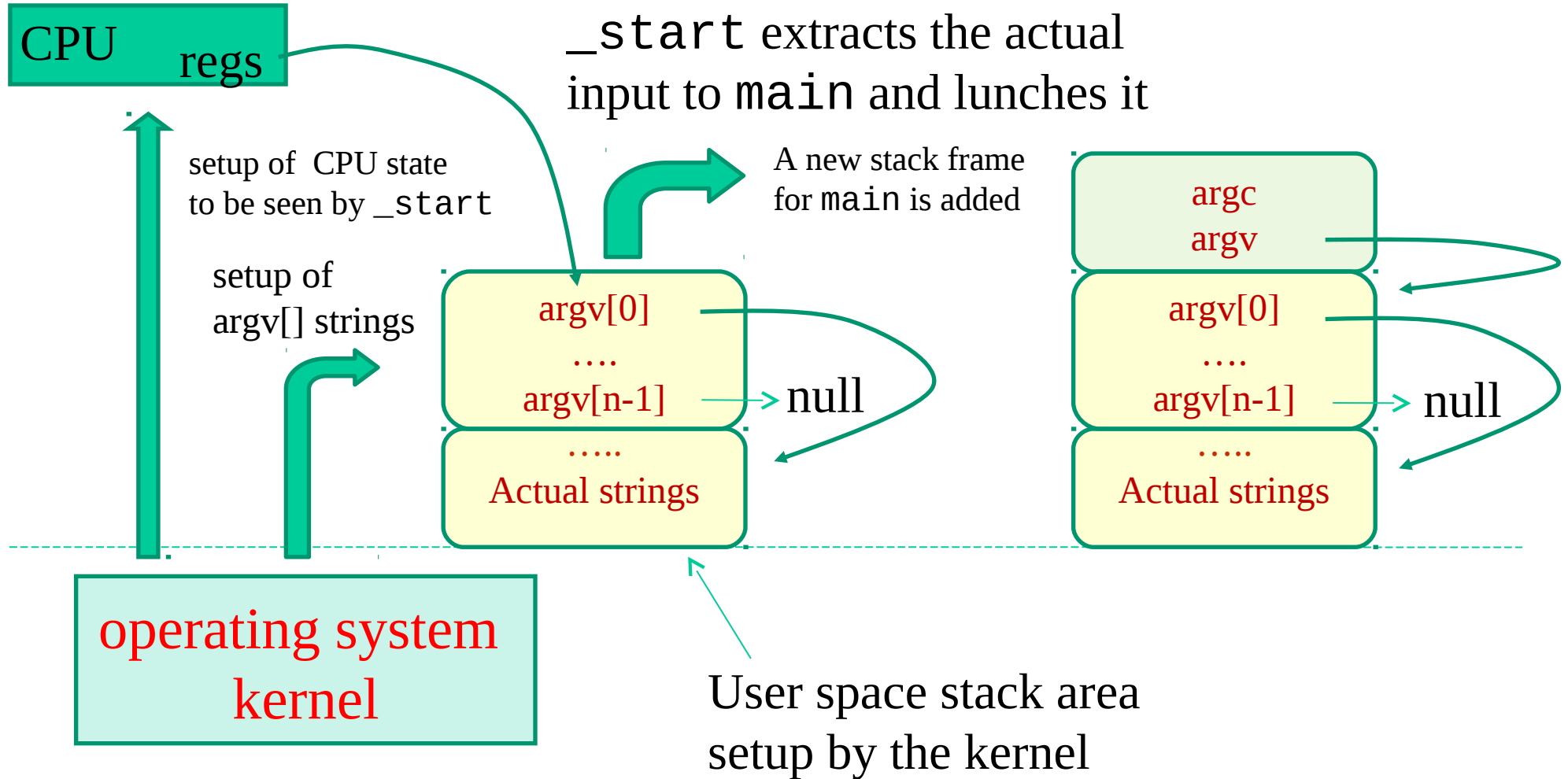


numero di elementi validi
nell'array



array di puntatori a carattere
(quindi a stringhe)

Effettivo schema di passaggio dei parametri



Ancora sulle system call

- indipendentemente dalla tipologia di sistema operativo, molte system call prevedono come parametri di input **descrittori (o handle)**
- un descrittore/handle è un codice operativo (e.g. una chiave)
- viene utilizzato dal kernel per identificare in modo veloce ed efficiente l'istanza di struttura dati coinvolta nell'esecuzione della system call
- quindi l'entità (gestita dal kernel) coinvolta nell'operazione

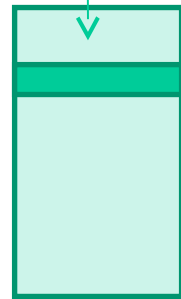
Uso di descrittori/handle

Process X
is the caller

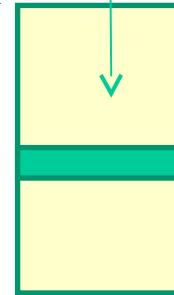
syscall_A(descriptor/handle

syscall_B(descriptor/handle

Kernel level



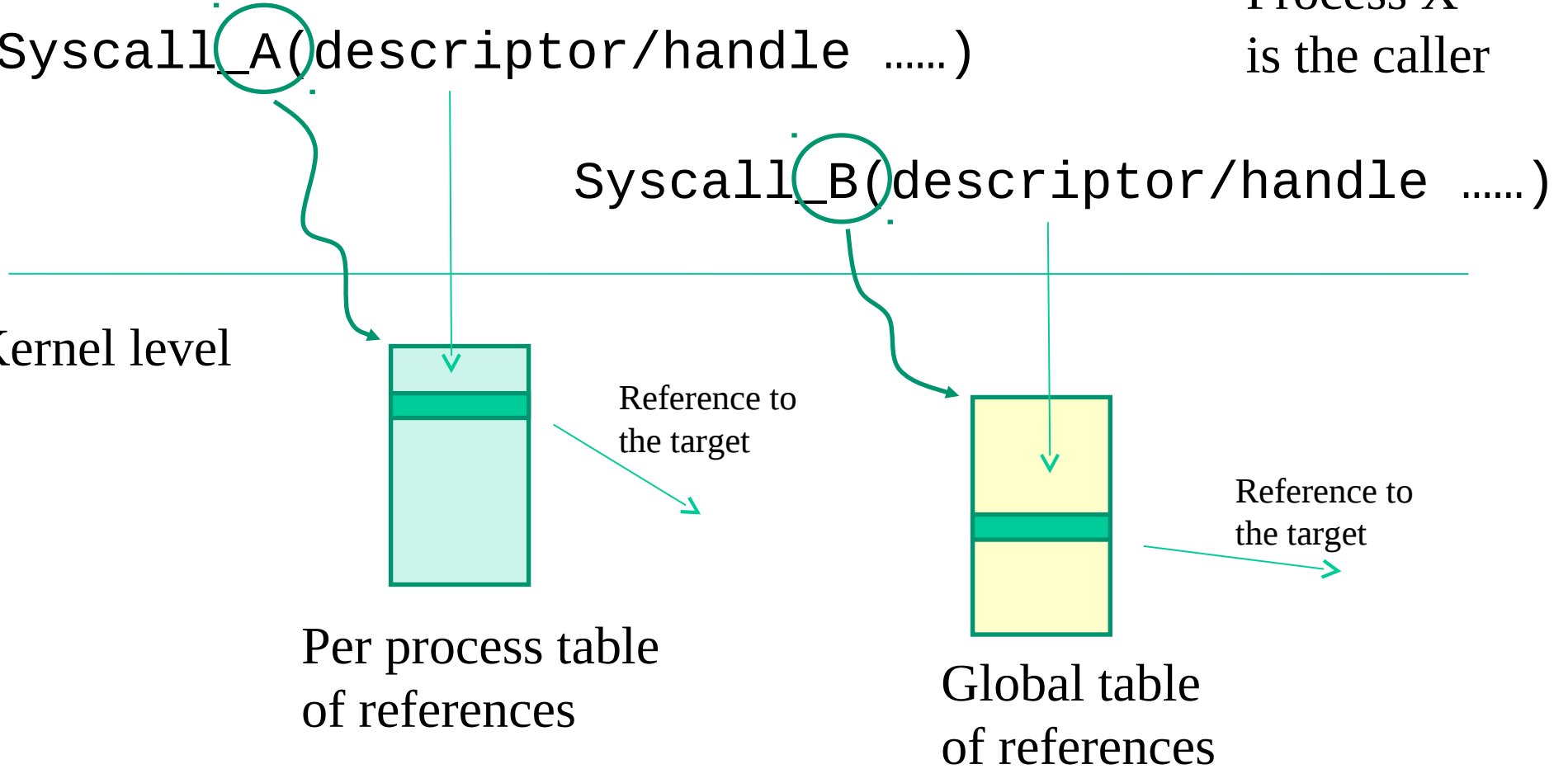
Per process table
of references



Global table
of references

Reference to
the target

Reference to
the target



Output dal kernel

- il vero output dal kernel si basa esclusivamente su
 - ✓ valore di ritorno di una system call, scritto dal kernel in un registro di processore prima del ritorno da trap
 - ✓ side-effect in memoria dovuti a passaggio di puntatori come parametri di system call
- ma il valore di ritorno è di fatto una maschera di bit manipolabile dal blocco “tail” di uno stub di accesso al kernel
- la vera informazione di output dal kernel quindi viene aumentata tramite operazioni di stub che aggiornano aree di memoria accessibili al codice applicativo
 - ✓ `errno` in Posix
 - ✓ `GetLastError()` in WinAPI

Aspetti basici di sicurezza

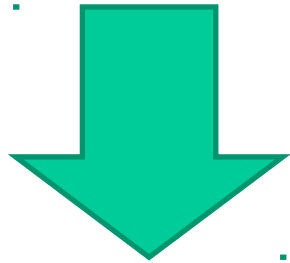
- linguaggi come il C permettono un controllo “assoluto” sullo stato della memoria riservata per le applicazioni
- vi è quindi la possibilità di accedere ad una qualsiasi area di memoria logicamente valida (per esempio tramite puntatori)
- alcune funzioni standard accedono a memoria tramite schemi di puntamento+spiazzamento con spiazzamento in taluni casi non deterministico
- il rischio è il così detto buffer overflow il quale può portare il software a comportamenti che deviano dalla specifica

Esempi di funzioni standard rischiose o deprecate

`scanf()`

`gets()`

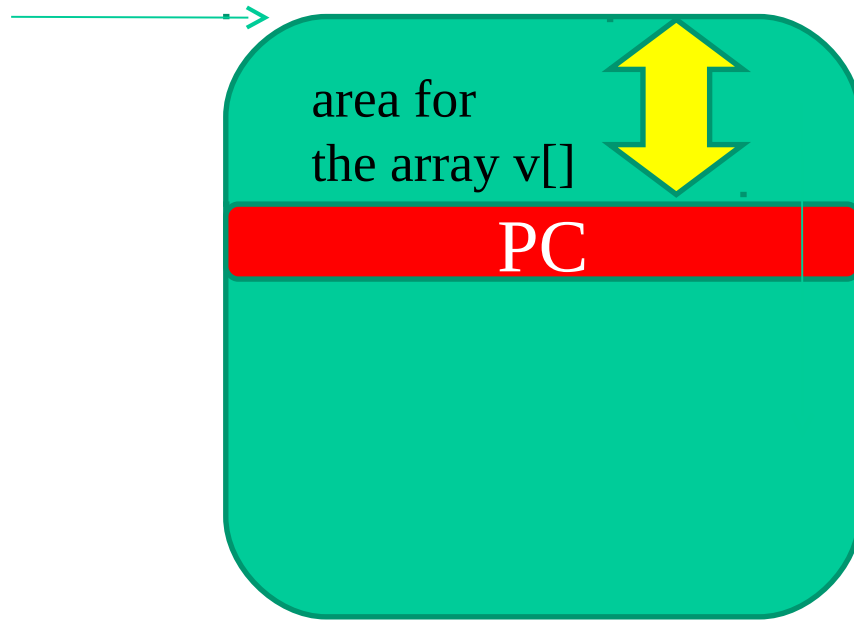
Alcune librerie mettono a disposizione varianti con miglioramenti di aspetti di sicurezza (**specifica della taglia dei buffer per ogni tipologia di dato da gestire**)



`scanf_s()`

Classico esempio di overflow dello stack

stack pointer
as seen by f()



Stack area

```
void f(){  
    char v[128];  
    .....  
    scanf("%s", v);  
    .....  
}
```

Strings longer than
128 will overflow
the buffer v[]

Risk of destroying
PC value

Altre possibilità con `scanf ()`

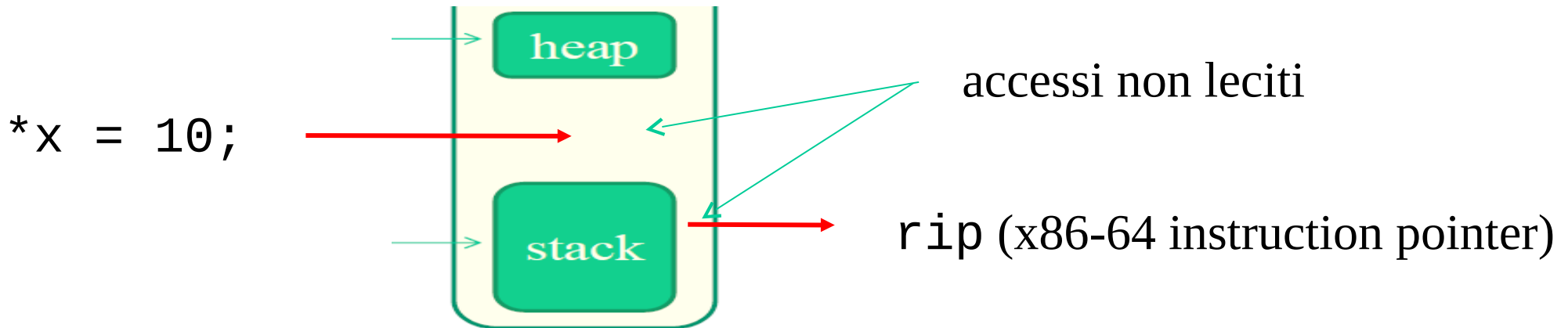
- `scanf ()` mette disposizione il modificatore di memoria “m” suffisso al tipo di dato da acquisire
- questo indica che l’area di memoria dove inserire l’input dovrà essere allocata a carico di `scanf ()`
- in tal caso viene resituito in un parametro l’indirizzo di tale area

```
char *p;  
scanf( "%ms", &p );
```

- alternativamente is può utilizzare un valore numerico al posto di “m” per indicare il numero di byte da trattare nell’operazione

Errori di segmentazione (segmentation fault)

- sono legati ad accesso a zone dello spazio di indirizzamento correntemente non valide
- sono anche legati ad accessi allo spazio di indirizzamento in modalità non conforme alle regole che il sistema operativo impone
 - ✓ `.text` è configurato read/execute
 - ✓ `.stack` è tipicamente configurato read/write (ma non execute, almeno su processori moderni, e.g. x86-64, e senza flag di compilazione `-z execstack`)

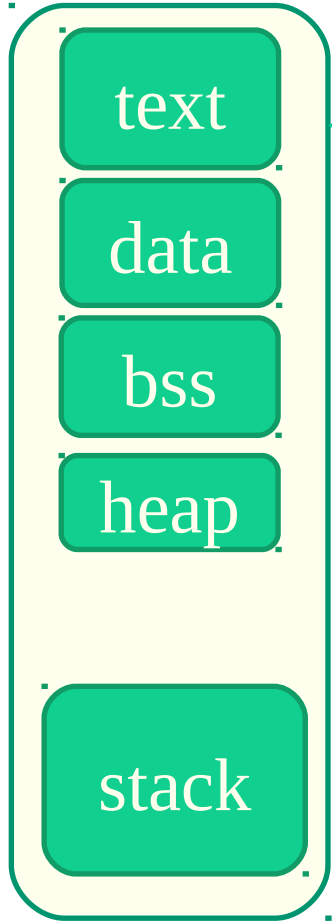


Randomizzazione (ASLR – Address Space Layout Randomization)

- in version più recenti di tool di compilazione, piattaforme hardware e sistemi operativi viene anche applicata la randomizzazione di parti dello spazio di indirizzamento (come lo stack) oppure dell'intero contenuto dell'address space
- questa consiste nel collocare le parti (.text, .data etc.) a spiazamento randomico (entro determinati limiti) a partire dall'inizio (o dalla fine) del contenitore quando l'applicazione viene attivata
- eventuali indirizzi di funzioni o dati noti a tempo di compilazione non coincideranno quindi con i relativi indirizzi nello spazio di indirizzamento e quindi daranno luogo a una difficoltà superiore per eventuali attaccanti che possano sfruttare delle vulnerabilità del software

Uno schema

disposizione
tradizionale



Posizione di un oggetto
nel contenitore definita a tempo
di compilazione

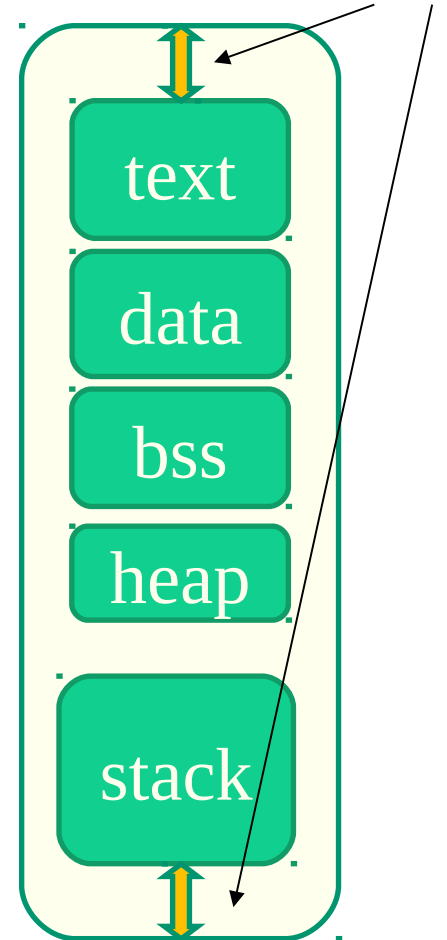
Posizione di un oggetto
nel contenitore definita a tempo
di attivazione dell'applicazione

randomizzazione



accesso a dati ed istruzioni basato
su tecnica `rip` - relative

offset
randomico



Generare codice indipendente dalla posizione

- gcc per mette la generazione di codice indipendente dalla posizione tramite i flag di compilazione `-pie -fPIE` (Position Independent Executable)
- codice PIE viene poi randomizzato, in termini di posizione nello spazio di indirizzamento, allo startup dipendendo dalla configurazione del kernel
- in Linux, si può abilitare o disabilitare la randomizzazione utilizzando il pseudo file `/proc/sys/kernel/randomize_va_space` (valore 0 disabilita la randomizzazione, valore 2 la attiva)
- in Windows (versioni recenti a partire da, e.g., 7) il supporto kernel per la randomizzazione è sempre attivo, e per generare codice PIE bisogna utilizzare l'opzione di compilazione `DYNAMICBASE` offerta da, e.g., Visual Studio
- ... le librerie dinamiche sono tipicamente codice PIE ...

Stack protector

- Un ulteriore meccanismo per l'amento del livello di sicurezza è lo stack protector
- Esso è basato sulla scrittura di un valore denominato “canary tag” sulla stack area subito prima del valore del program counter per il ritorno
- Prima di ritornare si verifica se il valore originariamente scritto è ancora presente
- In caso negativo si passa il controllo a un blocco di codice che ci fornisce in output l'indicazione di una corruzione dello stack e che poi termina l'applicazione
- Lo stack protector è includibile/escludibile a livello di compilazione in gcc utilizzando il flag stack-protector
- In ogni caso, non risolve completamente le problematiche di sovrascrittura del valore del program counter

Uno schema

struttura della funzione chiamata:

CT = valore del Canary Tag

%fs:0x28 = indirizzo di memoria di CT



stack area

```
mov  %fs:0x28, %rax
mov  %rax, -0x8(%rsp)
...
...
mov  -0x8(%rsp),%rax
xor  %fs:0x28,%rax
je   RETURN
callq CORRUPTION-HANDLER
RETURN: retq
```