

Sistemi Operativi

Laurea in Ingegneria Informatica

Universita' di Roma Tor Vergata

Docente: Francesco Quaglia

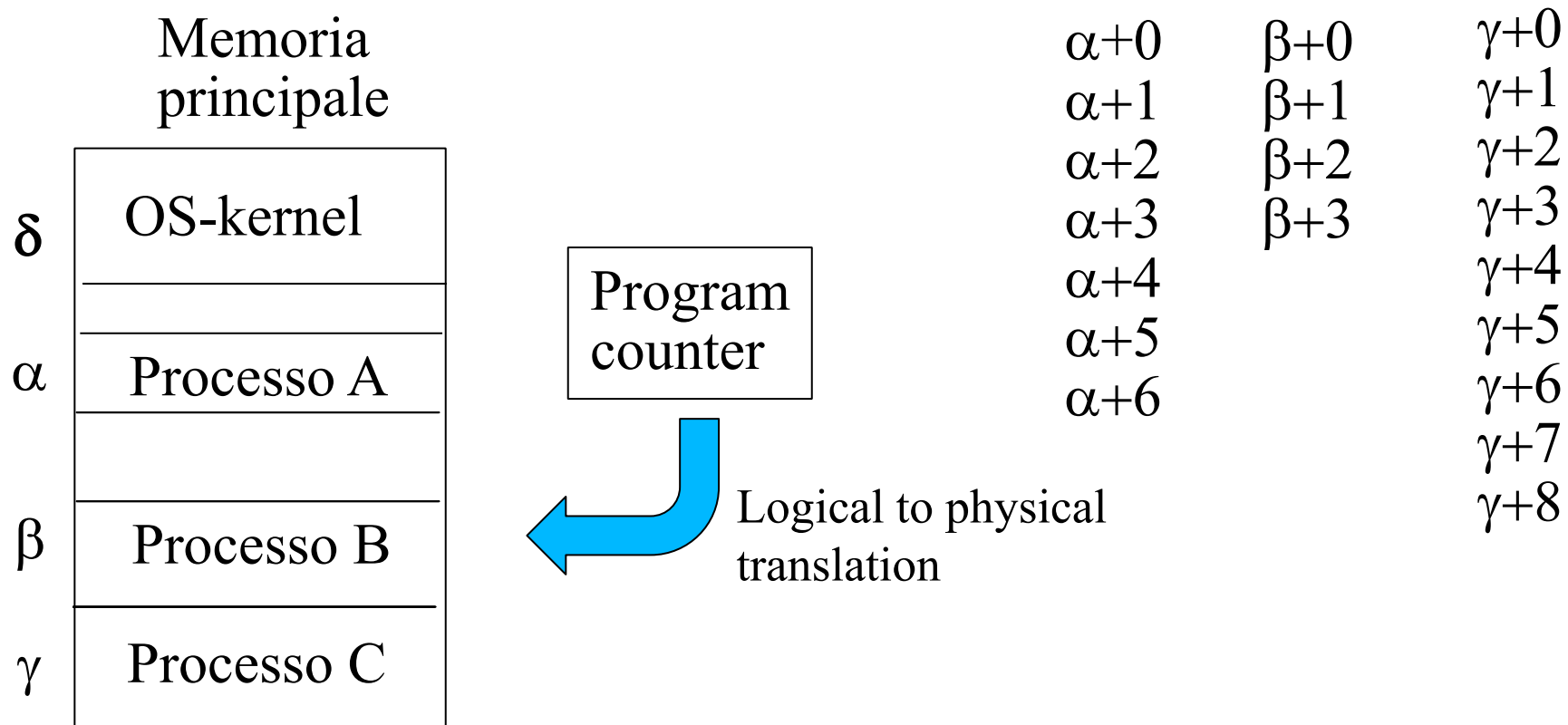


Processi e thread

1. Modelli a stati
2. Rappresentazione di processi
3. Liste di processi
4. Processi in sistemi UNIX/Windows
5. Multi-threading
6. Thread in sistemi UNIX/Windows

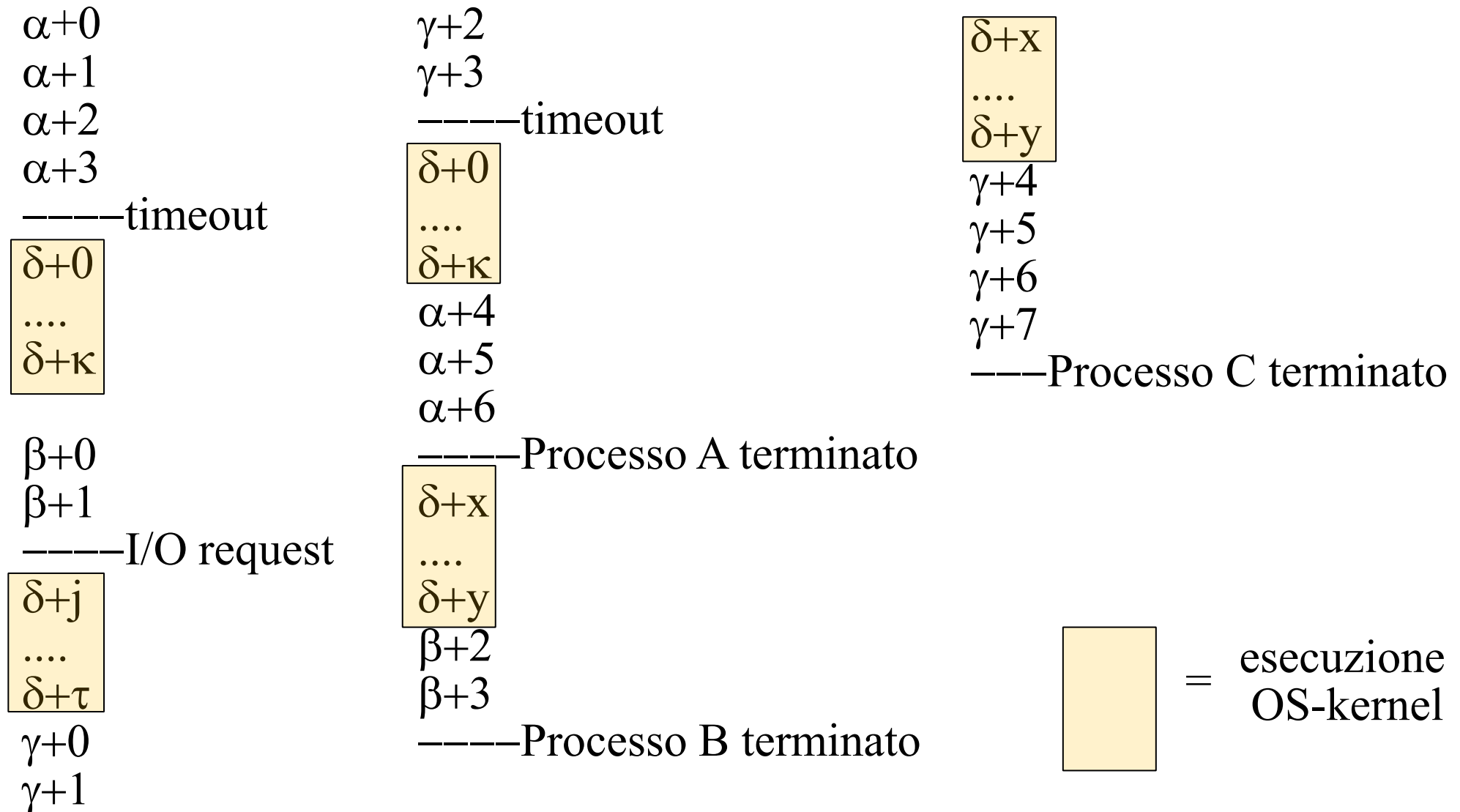
Esecuzione di processi

L'esecuzione di ogni processo puo' essere caratterizzata tramite una sequenza di istruzioni denominata traccia

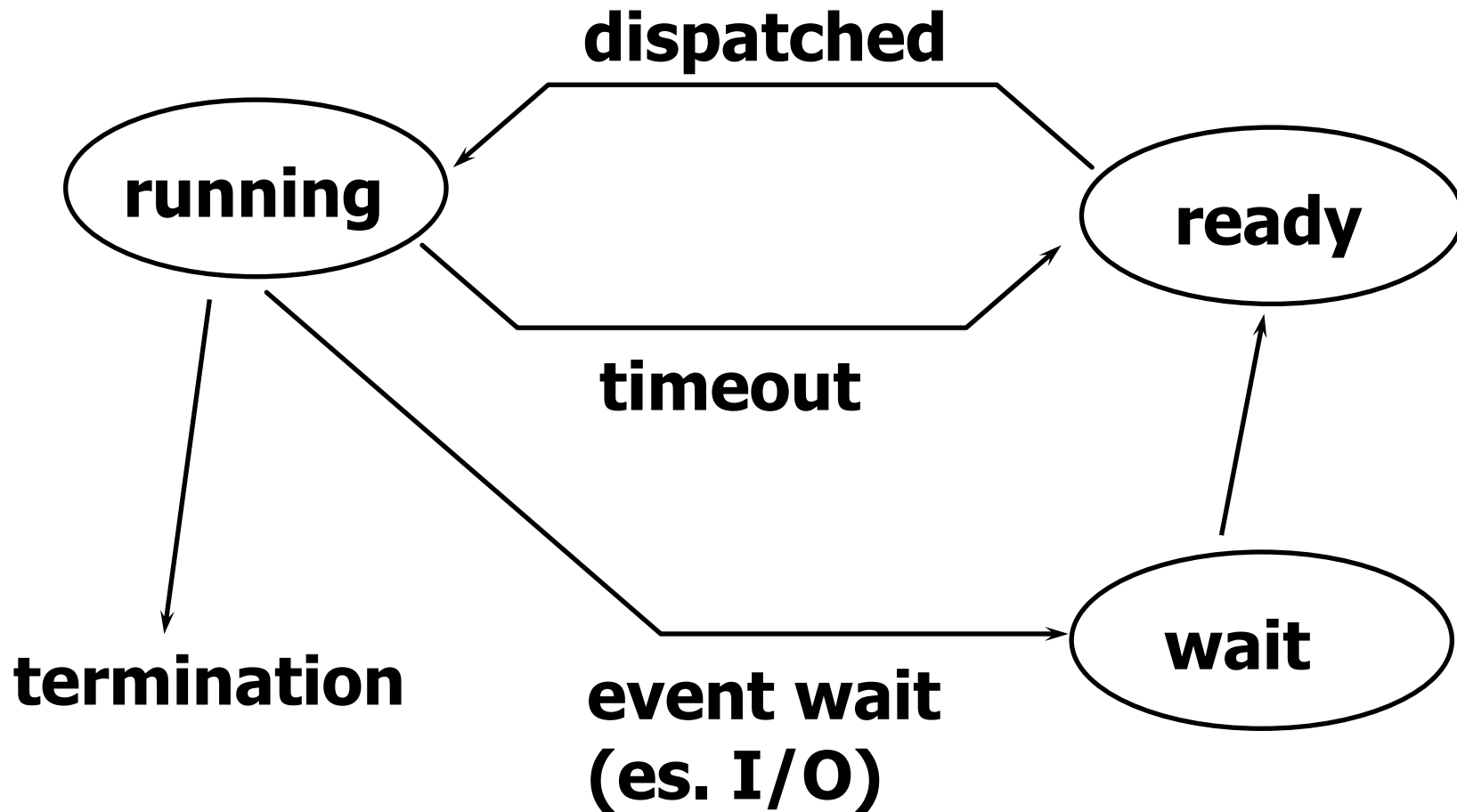


Un sistema operativo Time-Sharing garantisce una esecuzione interleaved delle tracce dei singoli processi

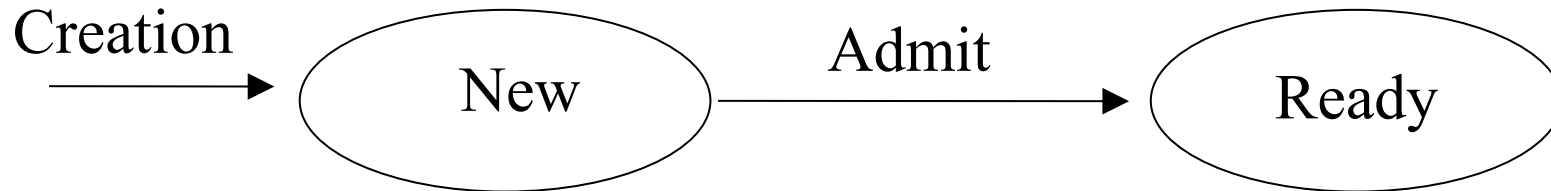
Un esempio di esecuzione interleaved



Stati fondamentali dei processi

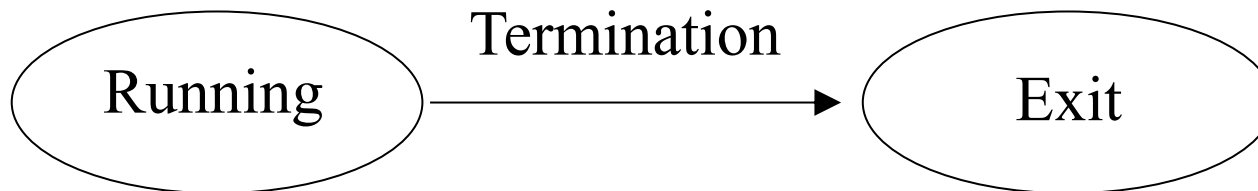


Stati aggiuntivi



New

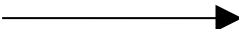
stato in cui il sistema **alloca ed inizializza strutture dati** per la gestione dell'esecuzione del processo

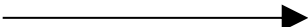


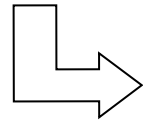
Exit

stato di **rilascio di strutture dati** allocate all'atto della terminazione del processo e di gestione delle azioni necessarie ad una corretta terminazione di processo

Il livello di multiprogrammazione

Il processore e' tipicamente molto piu' veloce dei sistemi di I/O 

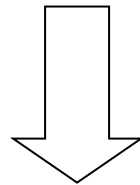
 esiste la possibilita' che la maggior parte dei processi residenti in memoria siano simultaneamente nello stato **Blocked** in attesa di completamento di una richiesta di I/O



Rischio di sottoutilizzo del processore

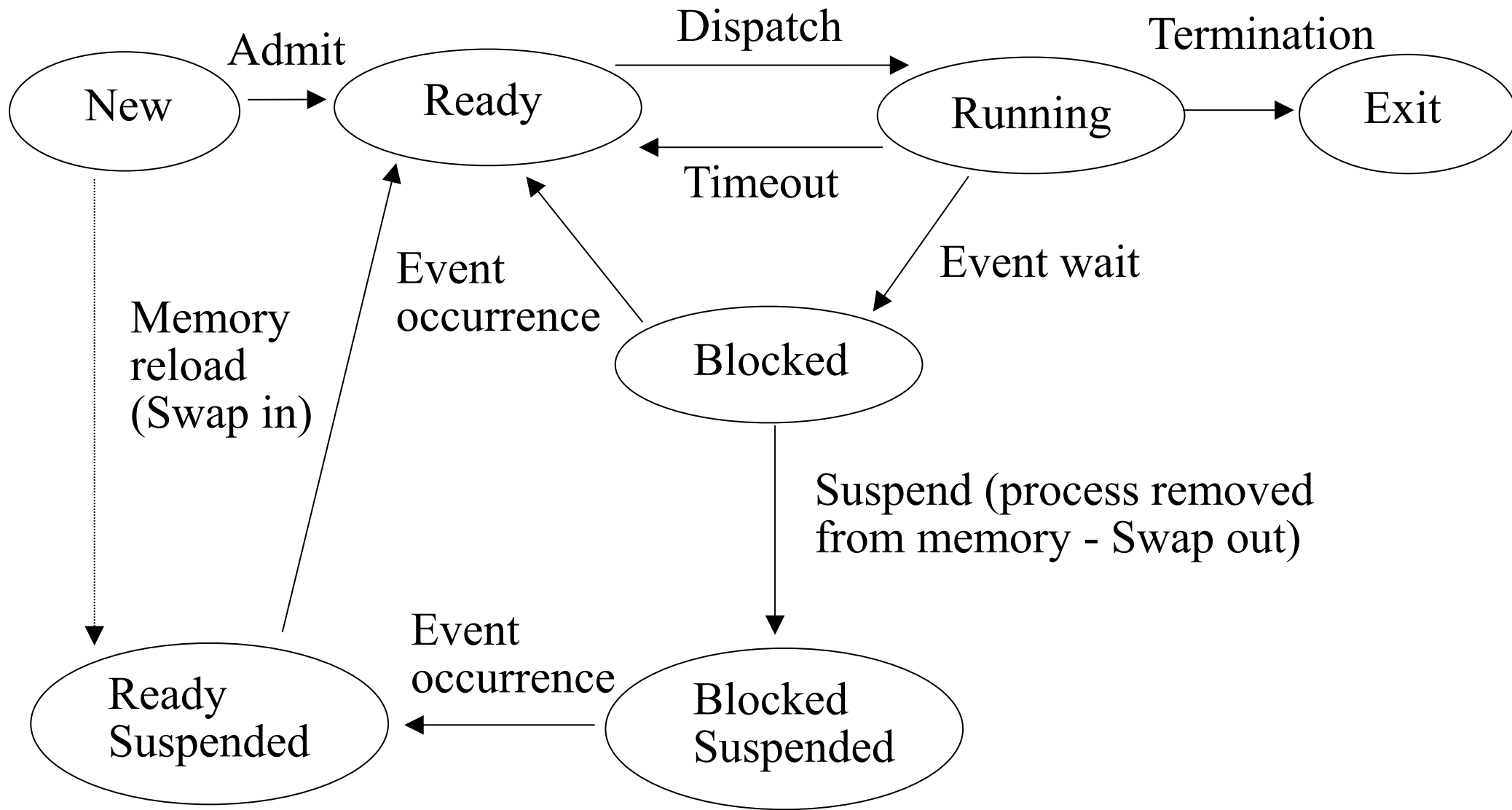
Prevenire il problema implica

- la necessita' di poter mantenere attivi un numero di processi molto elevato, ovvero aumentare il livello di multiprogrammazione



Per superare il limite imposto dal vincolo della quantita' di memoria fisica disponibile si utilizza la tecnica dello **Swapping**

Stati di un processo e swapping



Gestione dei processi: strutture di controllo

Per gestire l'esecuzione di processi il sistema operativo mantiene informazioni sui processi stessi e sulle risorse

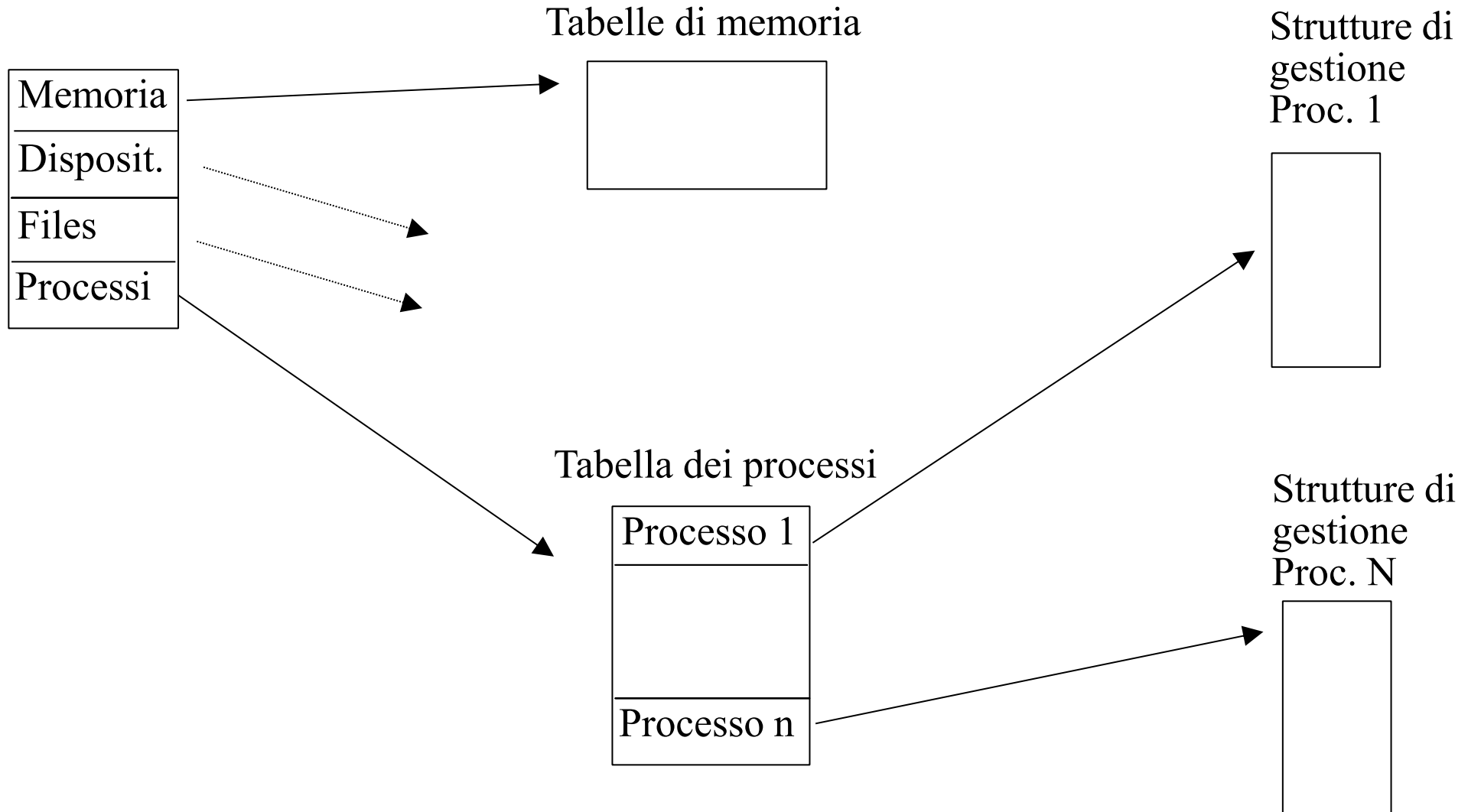


Tabelle di memoria

Includono informazioni su

- l'allocazione della *memoria principale* ai processi
- l'allocazione della *memoria secondaria* ai processi
- la modalita' di *protezione dei singoli segmenti* di memoria (per esempio quali processi possono eventualmente accedere ad alcune regioni di memoria condivisibili)
- tutto cio' che e' necessario a gestire la memoria virtuale nei sistemi in cui essa e' supportata

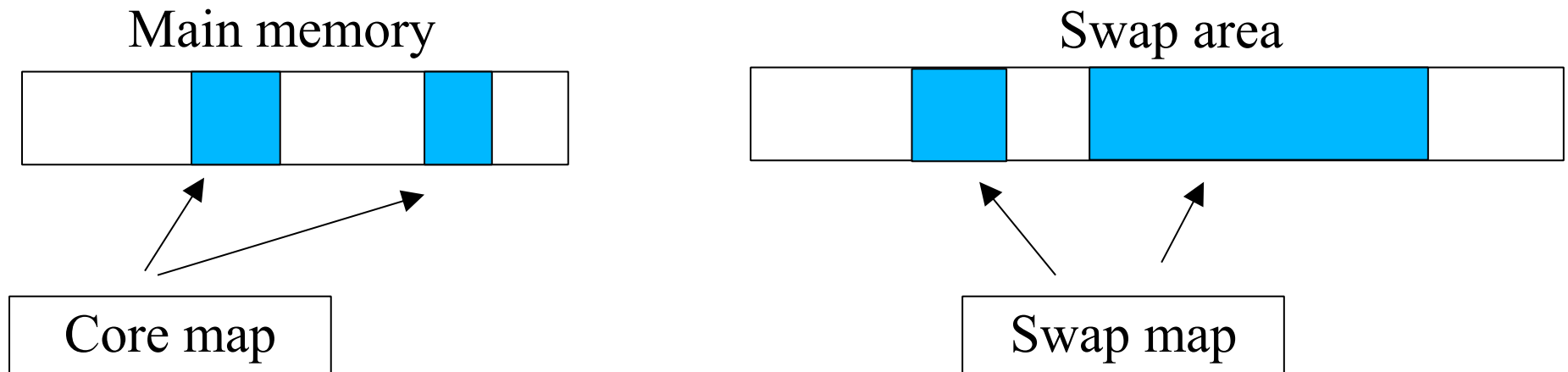


Immagine di un processo

L'immagine di un processo e' definita

- dal *programma* di cui il processo risulta istanza
- dai *dati*, inclusi nella parte modificabile dell'address space
- da uno *stack*, di supporto alla gestione di chiamate di funzione
- da uno *stack di sistema*, di supporto alla gestione di system-call o passaggi in modalita' kernel dovuti ad interrupt
- da una *collezione di attributi* necessari al sistema operativo per controllare l'esecuzione del processo stesso, la quale viene comunemente denominata *Process Control Block (PCB)*

Tale immagine viene tipicamente rappresentata in blocchi di memoria (contigui o non) che possono risiedere o in memoria principale o sull'area di Swap

Una porzione e' mantenuta in memoria principale per controllare efficientemente l'evoluzione di un processo anche quando esso risiede sull'area di Swap

PCB: attributi basici

Identificatori

Del processo in oggetto e di processi relazionati (padre, eventuali figli)

Stato del processo

Posizione corrente nel precedente diagramma di rappresentazione

Privilegi

In termini di possibilita' di accesso a servizi e risorse

Registri (contesto di esecuzione)

Contenenti informazioni associate allo stato corrente di avanzamento dell'esecuzione (es. il valore del program counter, i puntatori allo stack, i registri interni del processore)

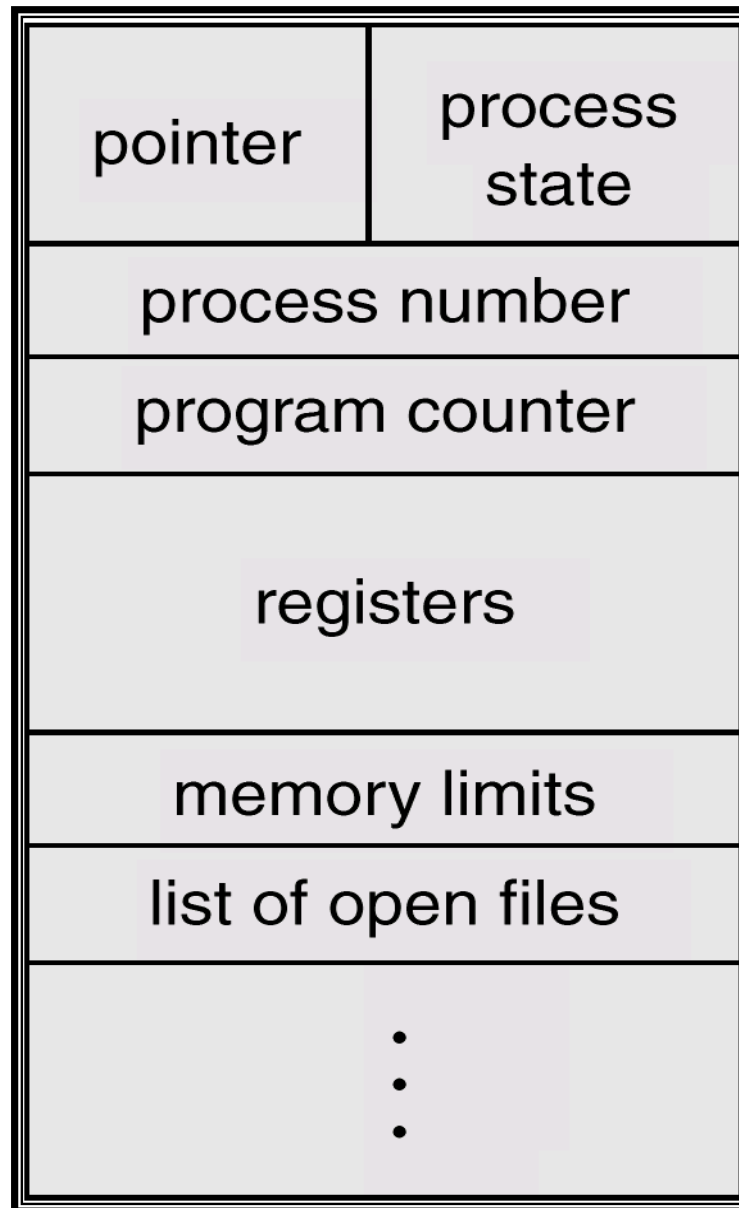
Informazioni di scheduling

Tutto cio' che il sistema necessita di sapere per poter arbitrare l'assegnazione del processore ai processi che si trovano nello stato **Ready** (problema dello *scheduling della CPU*)

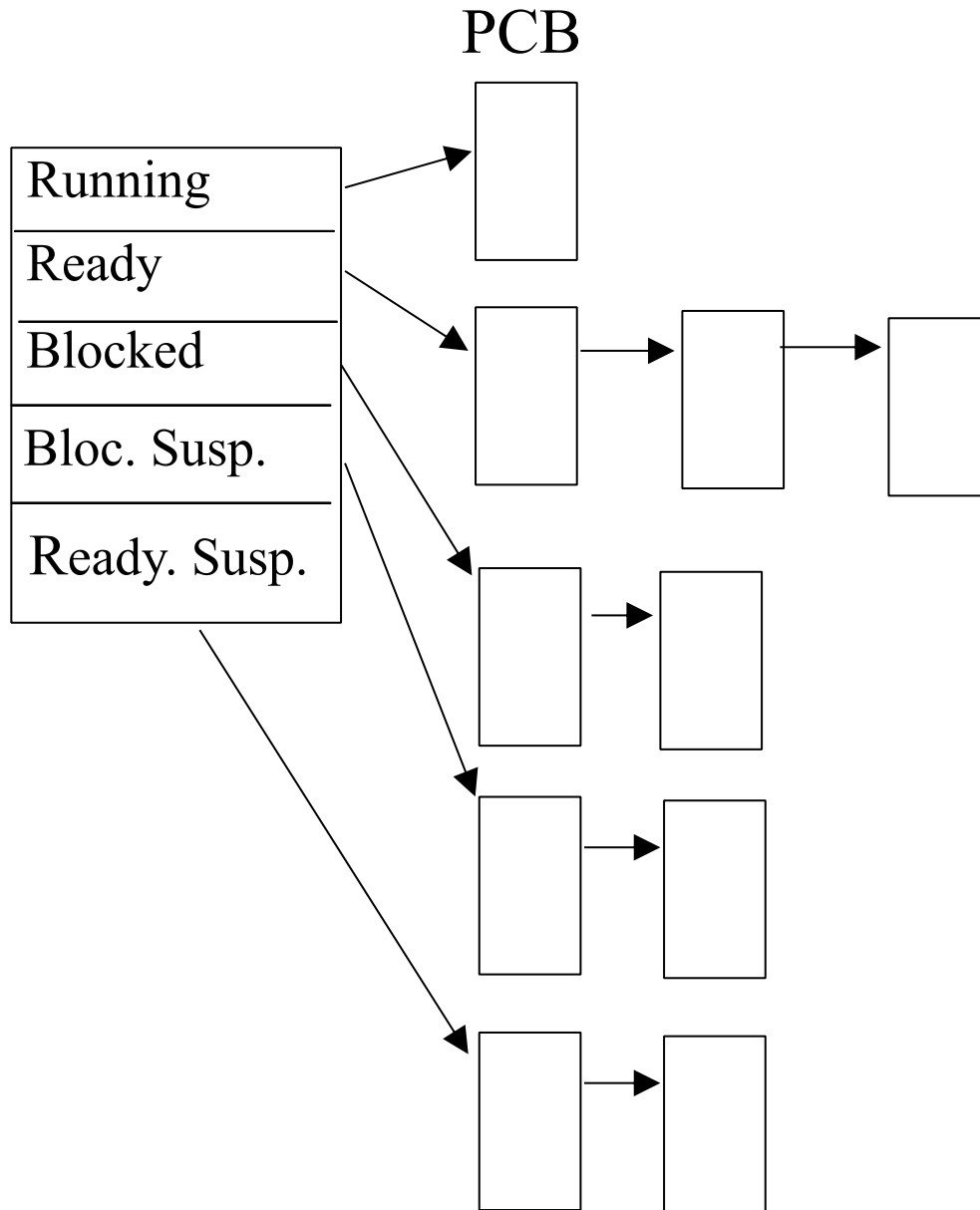
Informazioni di stato

Evento atteso

A Process Control Block (PCB) scheme



Liste di processi e scheduling



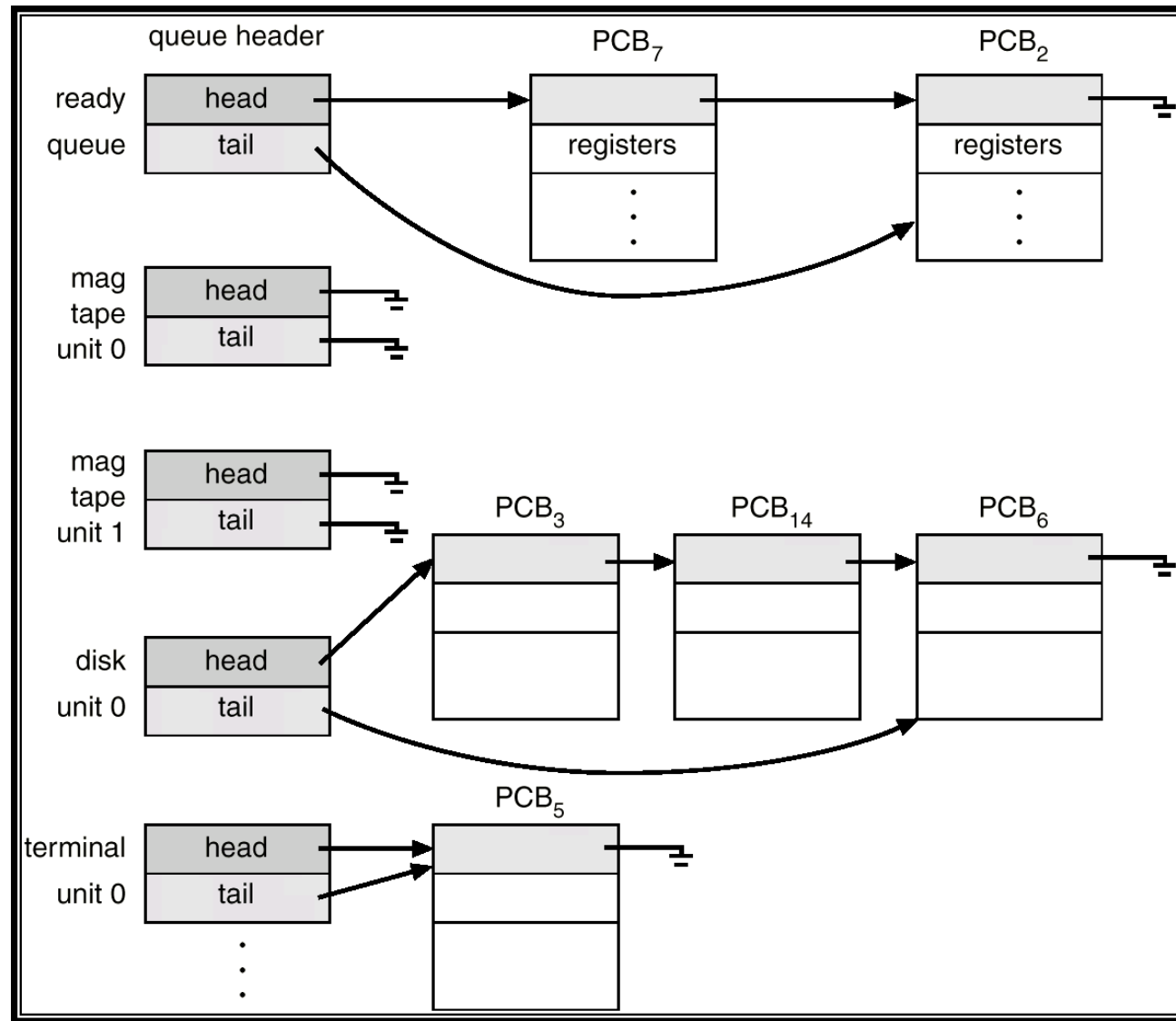
Ready \Rightarrow Running

Problema dello scheduling
di basso livello (*scheduling della CPU*)

Ready Susp. \Rightarrow Ready

Problema dello scheduling
di alto livello (*gestione dello swapping*)

Esempi per la 'ready queue' e per alcune code di I/O



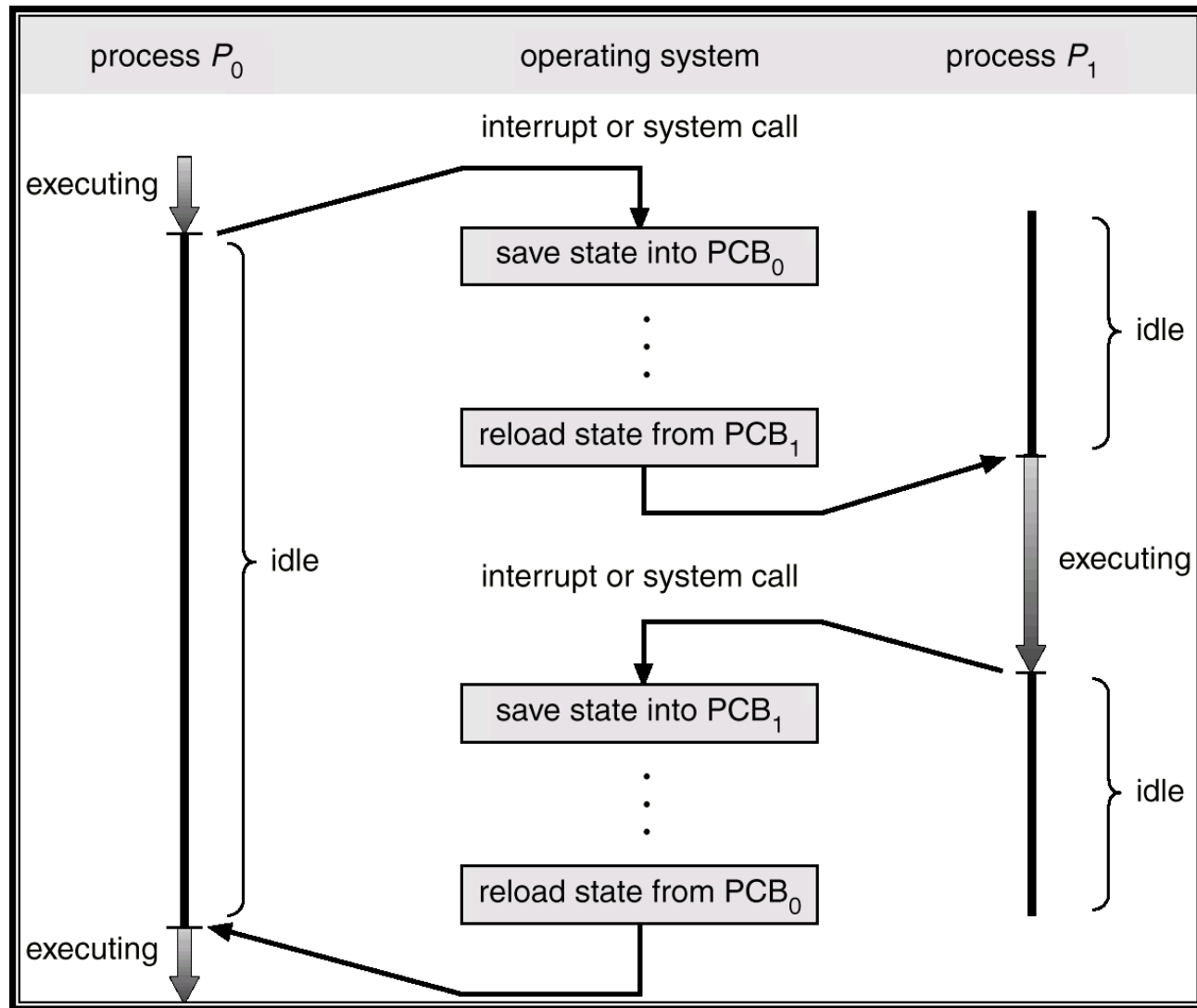
Cambio di contesto

- salvataggio del contesto corrente
 - aggiornamento del PCB del processo corrente (definizione dello stato)
 - inserimento del PCB nella lista/coda adeguata
 - selezione del processo da schedulare
 - aggiornamento del PCB del processo schedulato (cambio di stato)
 - ripristino del contesto del processo schedulato
-

Cambio di modo di esecuzione

- accesso a privilegi di livello superiore passando al modo kernel
- possibilità di esecuzione di istruzioni non ammesse in modo utente
- modo kernel caratterizzato (e quindi riconoscibile) da settaggio di bit di stato del processore , e.g. CPL (Current Privilege Level) bits in x86

Esempio di cambio di contesto tra processi



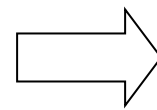
Modi di esecuzione e contesto di processo

Cause di cambio di contesto

- interruzione di clock (time-sharing), viene attivato lo scheduler per cedere il controllo ad un altro processo
- interruzione di I/O, con possibile riattivazione di un processo a piu' alta priorita'
- fault di memoria (per sistemi a memoria virtuale), con deattivazione del processo corrente

Cause di cambio di modo di esecuzione

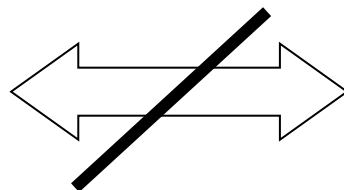
- attivazione di una funzione kernel
- gestione di una routine di interruzione



Salvataggio/ripristino di
porzione di contesto

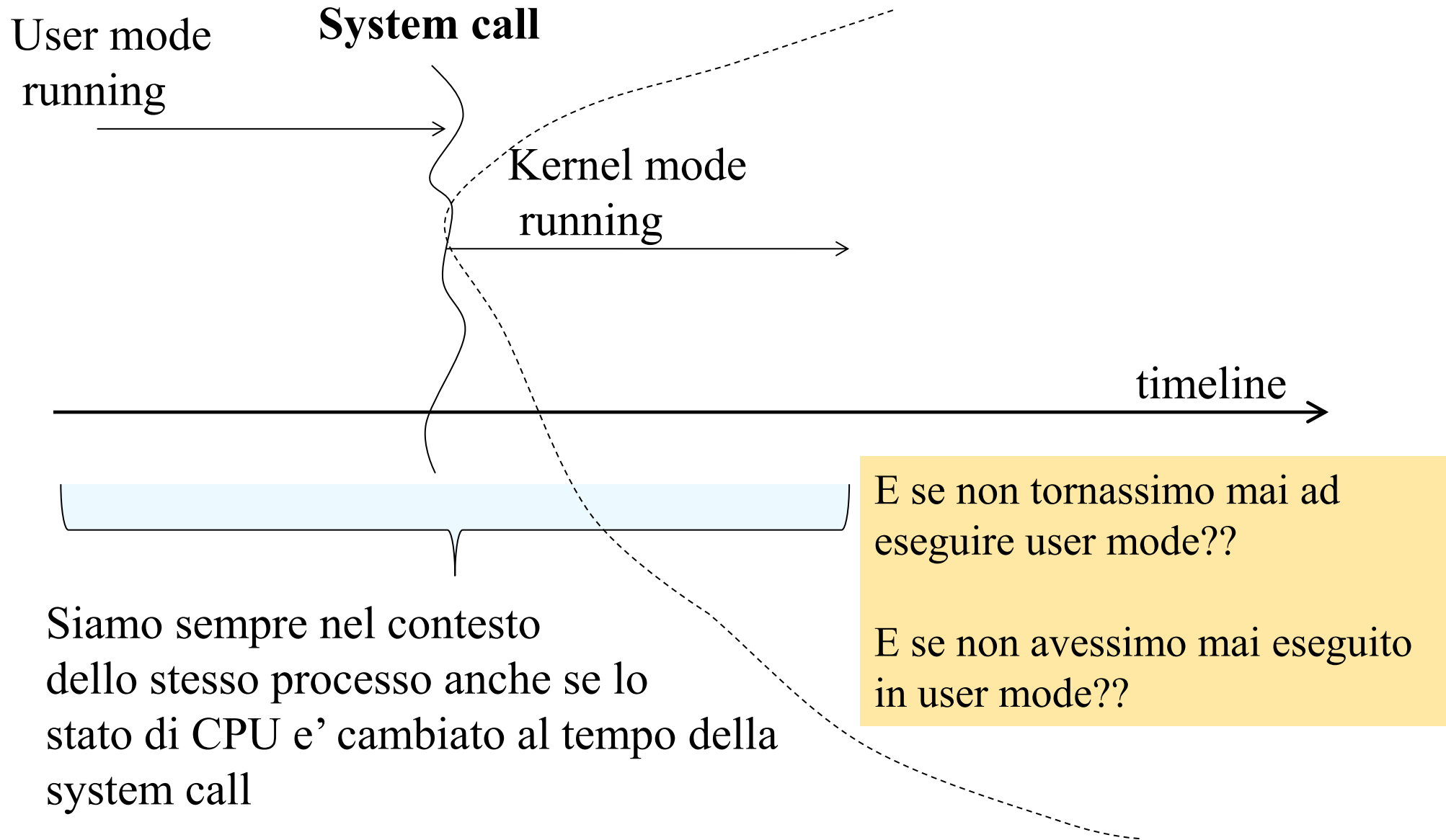
Nessuna implicazione diretta

Cambio di modo

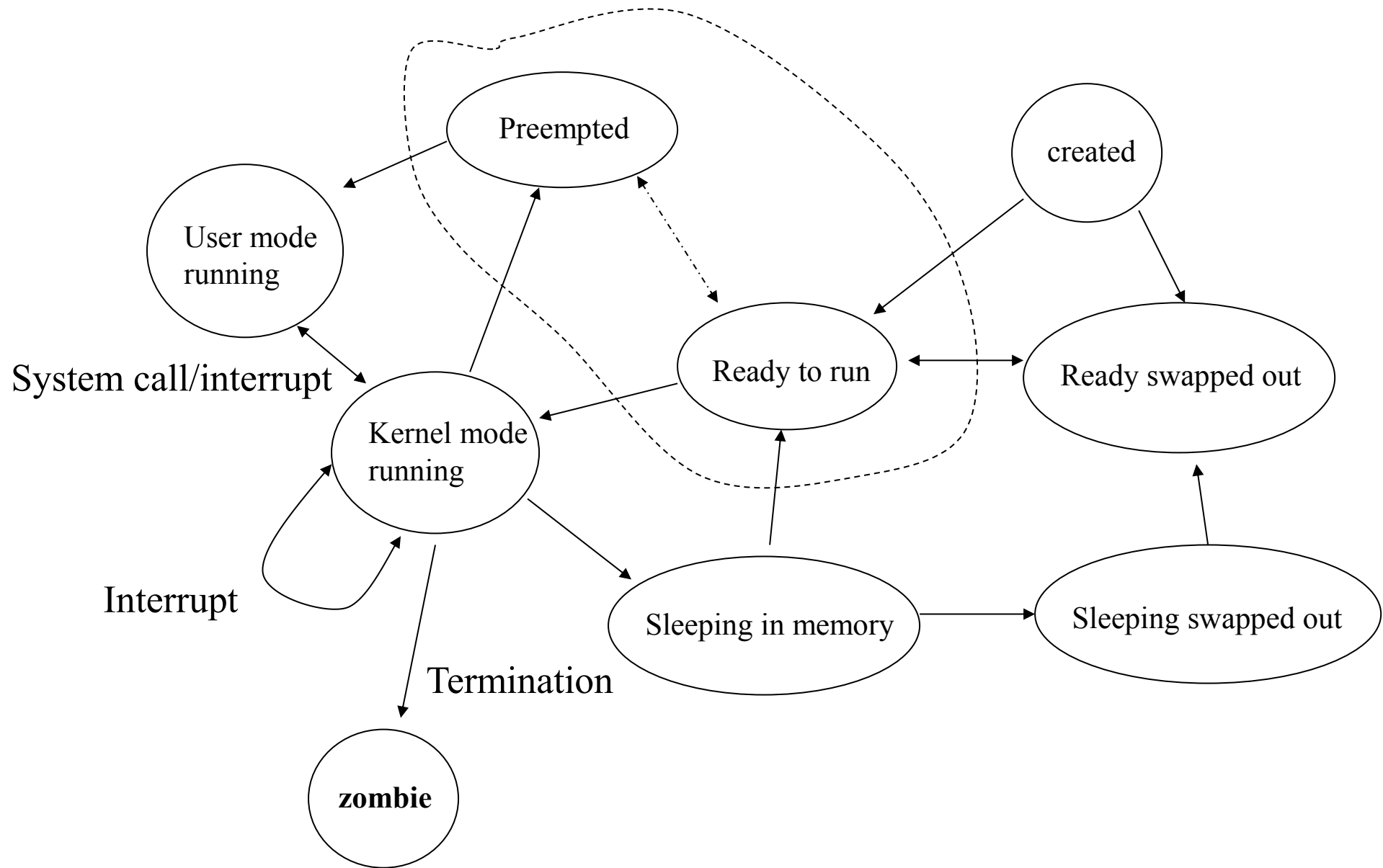


Cambio di contesto

Una timeline ed alcune osservazioni



Reference UNIX state diagram



Classica immagine di processo in sistemi UNIX

Testo

Dati

Stack utente

Memoria condivisa

Contesto utente

Program counter

Registro di stato del processore

Stack pointer

Registri generali

Contesto registri

Entry nella tabella dei processi

U area (area utente)

Tabella indirizzamento (memoria virtuale)

Stack del modo kernel

Contesto sistema

Entry della tabella dei processi: campi principali

Stato del processo

Identificatori d'utente (reale ed effettivo)

Identificatori di processi (pid, id del genitore)

Descrittore degli eventi (valido in stato sleeping)

Affinità di processore (insieme di processori utili per
l'esecuzione)

Priorità

Segnali (mandati al processo ma non ancora gestiti)

Timer (monitoring)

Stato della memoria (swap in/out)

U area: campi principali

Identificatori d'utente (effettivo/reale)

Array per i gestori di segnali

Terminale

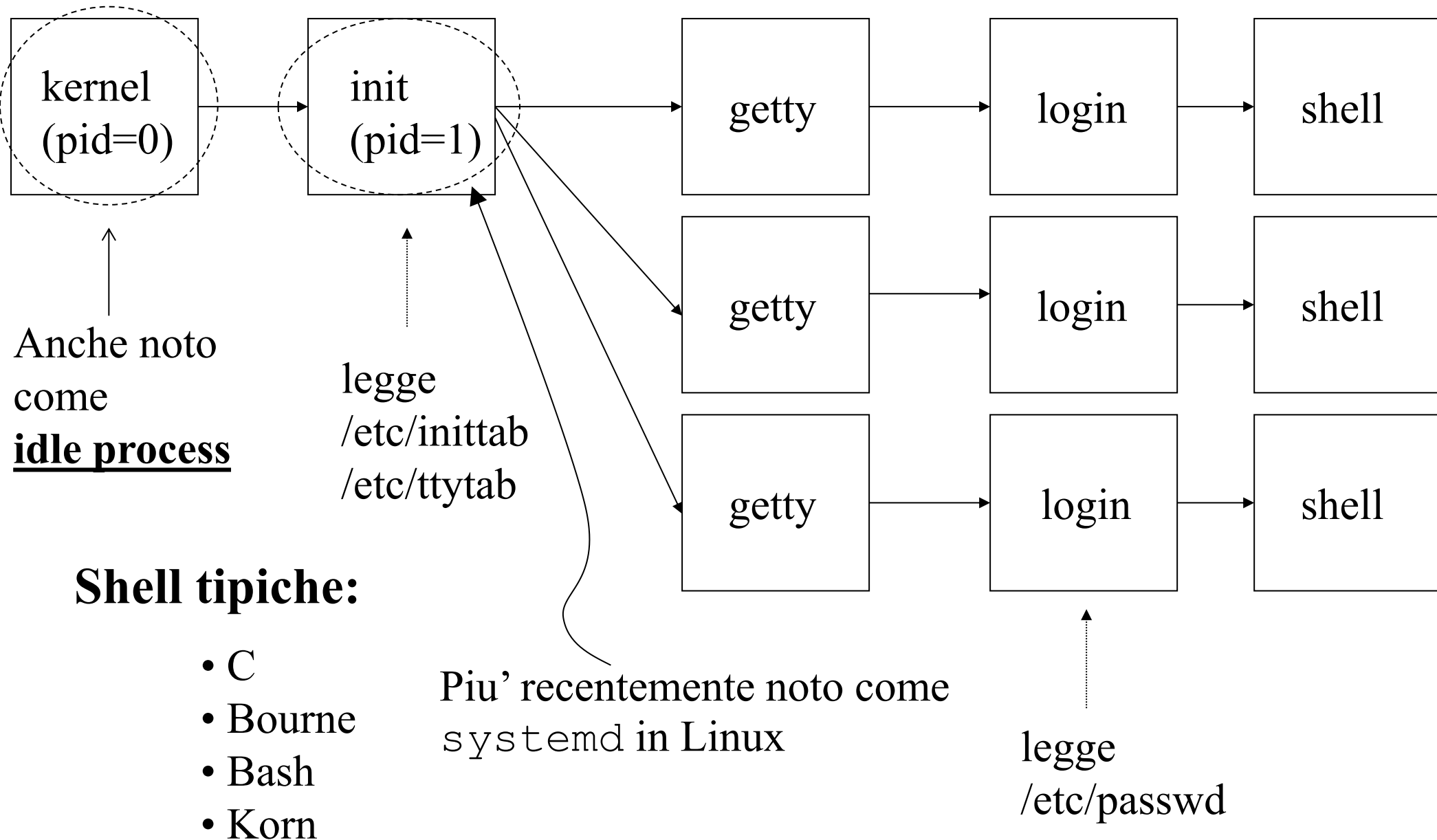
Parametri di I/O (es. indirizzi dei buffer)

Timer (monitoring in modalita' utente)

Valore di ritorno di system calls

Tabella dei descrittori di file

Sistemi UNIX: avvio tradizionale



Comandi di shell: nome-comando [arg1, arg2,, argn]

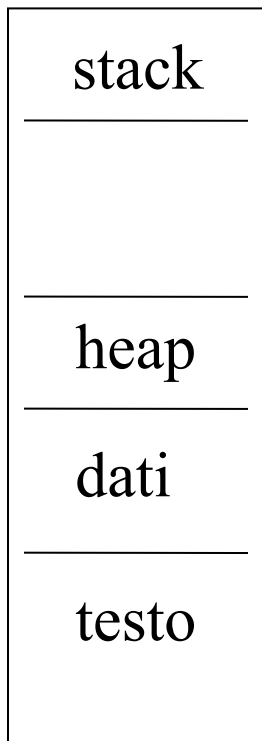
Creazione di un processo

pid_t fork(void)

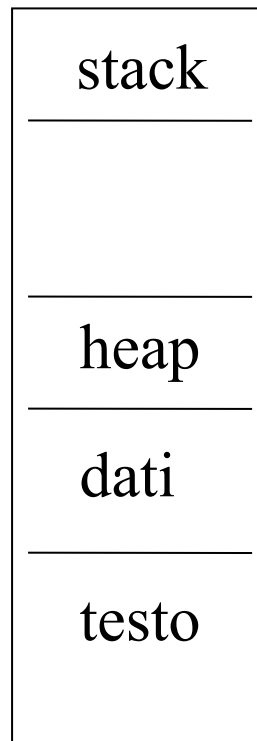
Descrizione invoca la duplicazione del processo chiamante

Restituzione 1) nel chiamante: pid del figlio, -1 in caso di errore
2) nel figlio: 0

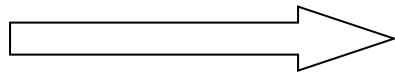
Processo padre



Processo figlio



fork()



Entrambi i processi
ripartono
dall'istruzione
successiva alla trap
al kernel dovuta alla
fork()

Sincronizzazione parent/child

`pid_t wait(int *status)`

Descrizione invoca l'attesa di terminazione di un generico proc. figlio

Parametri codice di uscita nei secondi 8 bit meno significativi puntati da status

Restituzione -1 in caso di fallimento

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

```
void main(int argc, char **argv){
```

```
    pid_t pid;    int status;
```

```
    pid = fork();
```

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

```
        printf("processo figlio");
```

```
        exit(0);
```

```
    }
```

```
    else{
```

```
        printf("processo padre, attesa terminazione figlio");
```

```
        wait(&status);
```

```
    }
```

```
}
```

Terminazione su richiesta
(definizione esplicita di un
codice di uscita)


Accesso al valore del PID e 'wait' selettivo

SYNOPSIS

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


```
pid_t getpid(void);
```

Proprio PID



```
pid_t getppid(void);
```

Parent PID




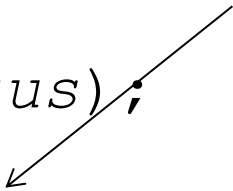
SYNOPSIS

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

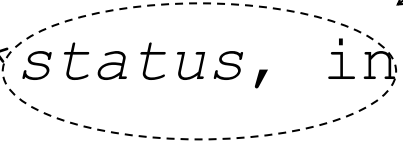
```
pid_t wait(int *status);
```

PID del processo da attendere

Parametrizzazione dell'esecuzione



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



```
#include <stdio.h>

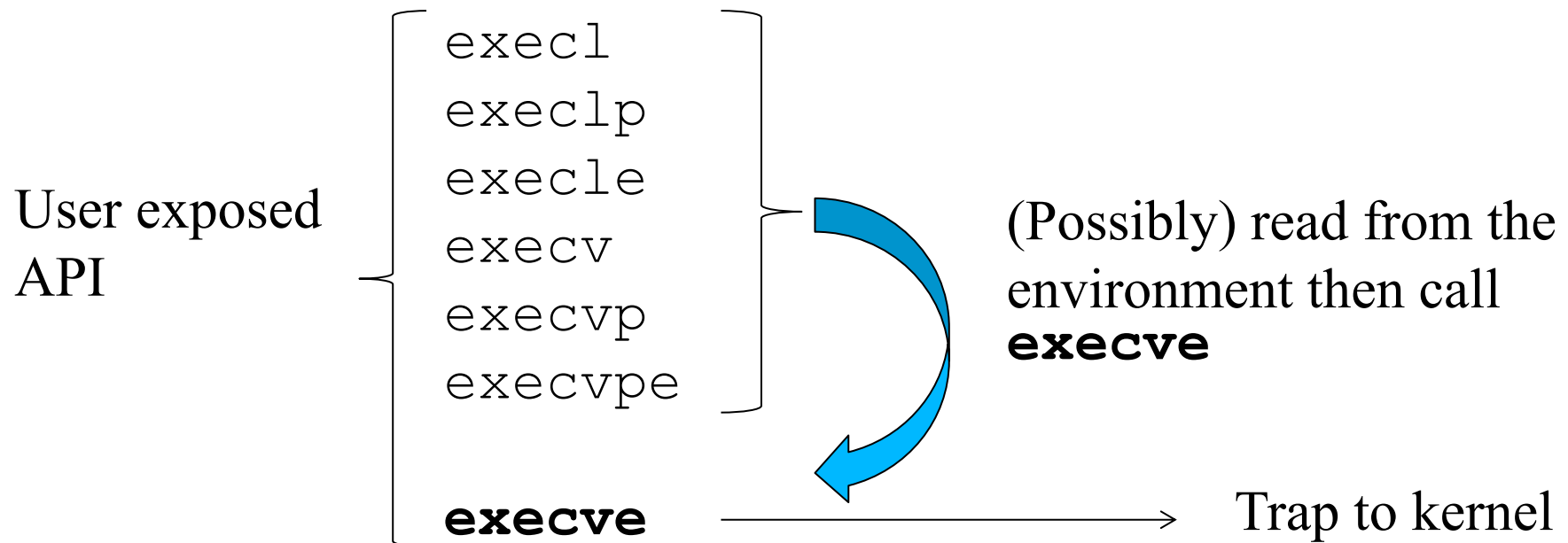
void main(int argc, char **argv){
    pid_t pid;    int status, result;
    pid = fork();

    if(pid == -1) ){
        printf("errore nella chiamata fork()");
        exit(-1);
    }

    if ( pid == 0 ){
        printf("processo figlio");
        exit(0);
    }
    else{
        printf("processo padre, attesa terminazione figlio");
        result = wait(&status);
        if(result == -1) printf("errore nella chiamata wait()");
    }
}
```

Definizione di immagini di memoria: famiglia di chiamate exec

- L'attivazione di un programma eseguibile generico (non un clone dello stato del programma correntemente in esecuzione) avviene su sistemi Unix tramite la famiglia di chiamate exec
- Esse sono tutte specificate nello standard di sistema Posix
- Ma solo una di esse e' una vera system call
- Le dipendenze nello standard di sistema Posix sono le seguenti:



Synopsis

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

```
extern char **environ;
```

```
int execl(const char *path, const char *arg, ...);
```

```
int execlp(const char *file, const char *arg, ...);
```

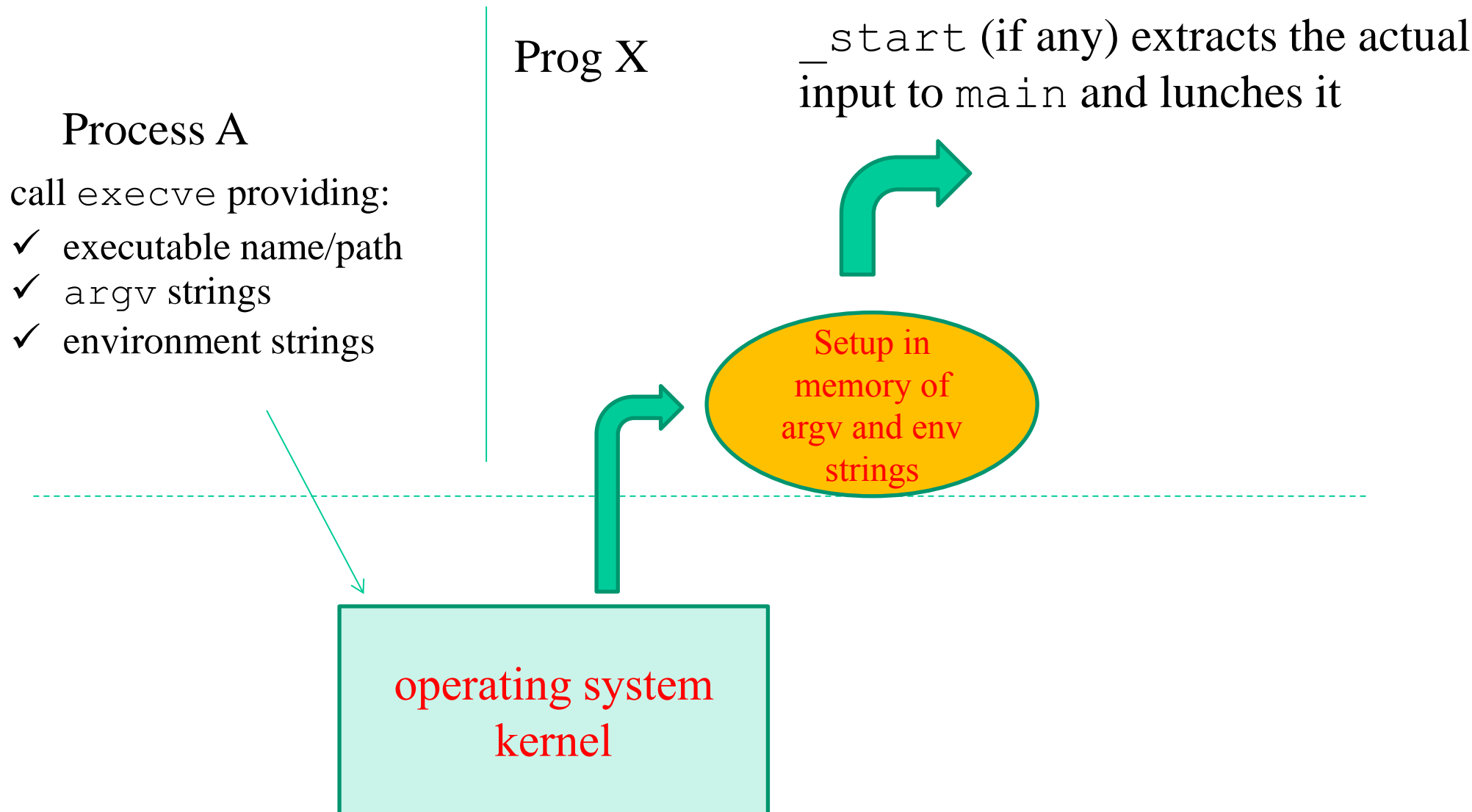
```
int execl_e(const char *path, const char *arg, ..., char * const envp[]);
```

```
int execv(const char *path, char *const argv[]);
```

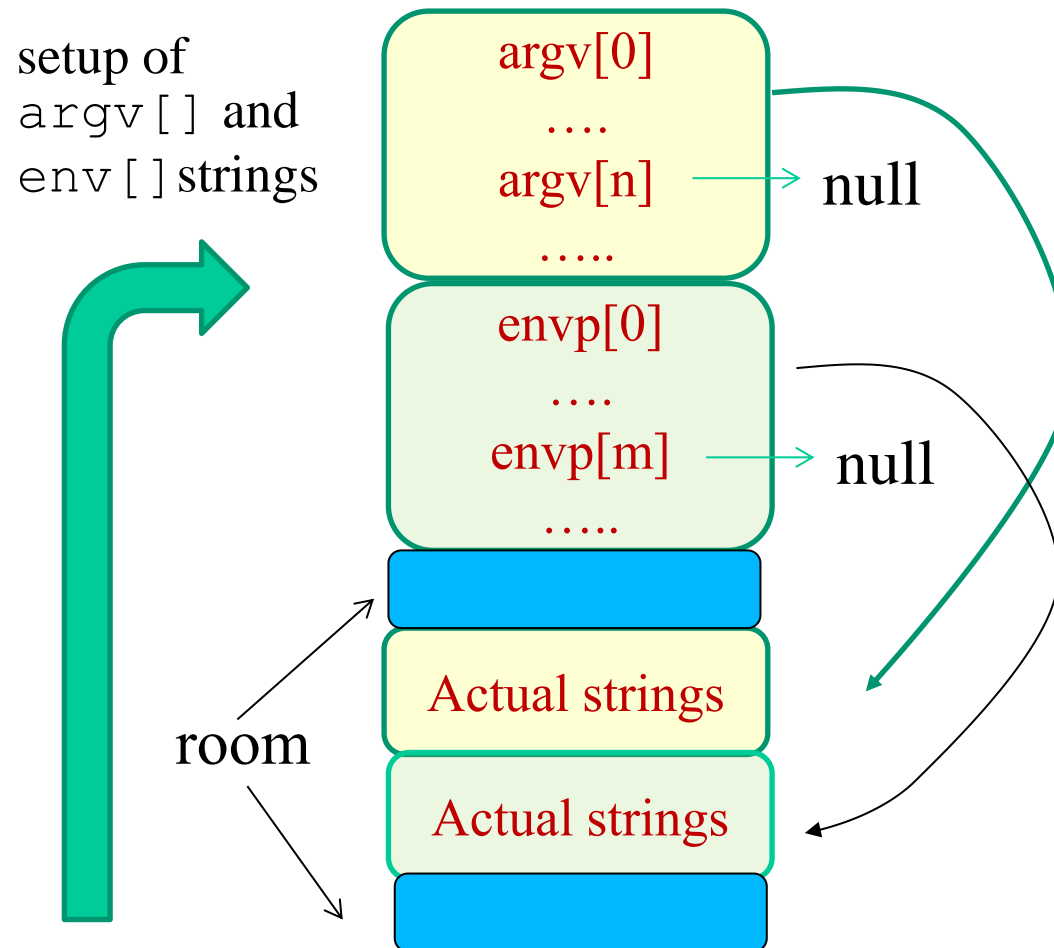
```
int execvp(const char *file, char *const argv[]);
```

```
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Environment setup



Collocazione di argv ed env



API per accedere ad argomenti ed ambiente

argv[] }

main

envp[] }

char ** environ;

getenv

putenv

setenv

unsetenv



**from
unistd.h**

execl

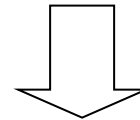
```
int execl(char *file_name, [char *arg0, ... ,char *argN,] 0)
```

Descrizione invoca l'esecuzione di un programma

Parametri 1) *file_name: nome del programma
 2) [*arg0, ... ,*argN,]: parametri della funzione main()

Restituzione -1 in caso di fallimento

- l'esecuzione avviene per sostituzione del codice (valido per tutte le chiamate della famiglia)
- 'cerca' il programma da eseguire solo nel direttorio corrente
- la funzione execlp() cerca in tutto il path valido per l'applicazione che la invoca



Mantenuto nella variabile d'ambiente PATH

Definizione di parametri del main() a tempo di esecuzione: **execv**

```
int execv(char *file_name, char **argv)
```

Descrizione invoca l'esecuzione di un programma

Parametri 1) *file_name: nome del programma
 2) **argv: parametri della funzione main()

Restituzione -1 in caso di fallimento

- **‘cerca’ il programma da eseguire solo nel direttorio corrente**
- **la variante `execvp()` cerca in tutto il path valido per l'applicazione che la invoca, secondo uno schema ‘fail-retry’**

Un esempio semplice

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    execlp("ls", "ls", 0);
    printf("La chiamata execlp() ha fallito\n")
}
```

Nota

Il risultato dell'esecuzione dipende dalla composizione della variabile di ambiente PATH !!!!!

Una semplice shell di comandi UNIX like

- Comandi **interni** hanno il codice cablato nel programma shell
 - Comandi **esterni** corrispondono a codice non cablato nel programma shell attivo
-

```
#include <stdio.h>
void main() {
    char comando[256];
    pid_t  pid; int status;
    while(1) {
        printf("Digitare un comando: ");
        scanf("%s",comando);
        pid = fork();
        if ( pid == -1 ) {
            printf("Errore nella fork.\n");
            exit(1);
        }
        if ( pid == 0 )
            execlp(comando,comando,0);
        else wait(&status);
    }
}
```

- Esegue comandi senza parametri
- Comandi **interni** (ovvero su variabili d'ambiente) non hanno effetto
- Di fatto esegue, con effetto, solo comandi **esterni**

Variabili d'ambiente: alcuni dettagli

PWD	direttorio di lavoro corrente
HOME	directory principale d'utente
PATH	specifica di ricerca di eseguibili
DISPLAY	specifica di host dove reindirizzare l'output grafico
LOGNAME	login dell'utente

Acquisizione valori di variabili d'ambiente

```
char *getenv(char *name)
```

Descrizione richiede il valore di una variabile d'ambiente

Parametri *name, nome della variabile d'ambiente

Restituzione NULL oppure la stringa che definisce il valore della variabile

Definizione di variabili d'ambiente

```
int putenv(char *string)
```

Descrizione setta il valore di una variabile d'ambiente

Parametri *string, nome della variabile d'ambiente + valore da assegnare (nella forma “nome=valore”)

Restituzione 0 in caso di success – valore diverso da zero in caso di fallimento

- se la variabile d'ambiente non esiste viene anche creata
- la congiunzione di valori avviene attraverso il carattere ':'

ES.  PATH=/user/local/bin:/bin:/home/quaglia/bin

Settaggio/eliminazione di variabili d'ambiente

```
int setenv(char *name, char *value, int overwrite)
```

Descrizione crea una variabile d'ambiente e setta il suo valore

Parametri

- 1) *name: nome della variabile d'ambiente
- 2) *value: valore da assegnare
- 3) overwrite: flag di sovrascrittura in caso la variabile esista

Restituzione 0 in caso di successo, -1 in caso di fallimento

```
int unsetenv(char *name)
```

Descrizione elimina una variabile d'ambiente

Parametri *name: nome della variabile d'ambiente

Restituzione 0 in caso di successo, -1 in caso di fallimento

Passaggio di variabili d'ambiente su **exec**

```
int execve(char *file_name, char **argv, char **envp)
```

Descrizione invoca l'esecuzione di un programma

Parametri

- 1) *file_name: nome del programma
- 2) **argv: parametri della funzione main()
- 3) **envp: variabili d'ambiente

Restituzione -1 in caso di fallimento

Quando si esegue una fork(), le variabili d'ambiente del processo padre vengono ereditate totalmente dal processo figlio

Gestione basica di variabili d'ambiente da shell

- **bash shell**

- ✓ `export NAME=VAL`
- ✓ `unset NAME`
- ✓ `$NAME` richiama il valore attuale

- **tcsh shell**

- ✓ `setenv NAME VAL`
- ✓ `unsetenv NAME`
- ✓ `$NAME` richiama il valore attuale

User vs kernel environment

- Le variabili d'ambiente sono dati utilizzate solo in user space
- Il kernel di sistemi Unix (ma anche Windows) non utilizza tali valori per governare il comportamento delle system call
- Il kernel fa solo il setup in user-space di tali valori (iniziali)
- Il kernel mantiene invece variabili “di configurazione” (**pseudo ambiente**) per ogni processo attivo
- Queste servono a determinare il comportamento dell'esecuzione **modo kernel** di una system call
- Esempi sono
 - ✓ VFS (virtual file system) pwd (see the `chdir()` system call)
 - ✓ VFS (virtual file system) root
 - ✓ Kernel level user ID
- E' compito del software applicativo (e.g. di libreria) mantenere la consistenza tra ambiente e configurazione a livello kernel in caso vi siano dati omologhi

Oggetti

In NT/2000 ogni entita' e' un oggetto

Gli oggetti si distinguono per

- tipo
 - attributi dell'oggetto
 - servizi
-

I servizi definiscono cio' che e' possibile richiede al sistema operativo per quel che riguarda un oggetto di un determinato tipo

Oggetti di tipo processo: attributi

ID del processo

Descrittore della sicurezza

Priorita' di base (dei thread del processo)

Affinita' di processore (insieme di processori utili per l'esecuzione)

Limiti di quota

Tempo di esecuzione (totale di tutti i thread del processo)

Contatori di I/O

Contatori di memoria virtuale

Porte per le eccezioni (canali verso il gestore dei processi)

Stato di uscita

Oggetti di tipo processo: servizi

Creazione di processi

Apertura di processi

Richiesta/modifica di informazioni di processi

Terminazione di processo

Allocazione/rilascio di memoria virtuale

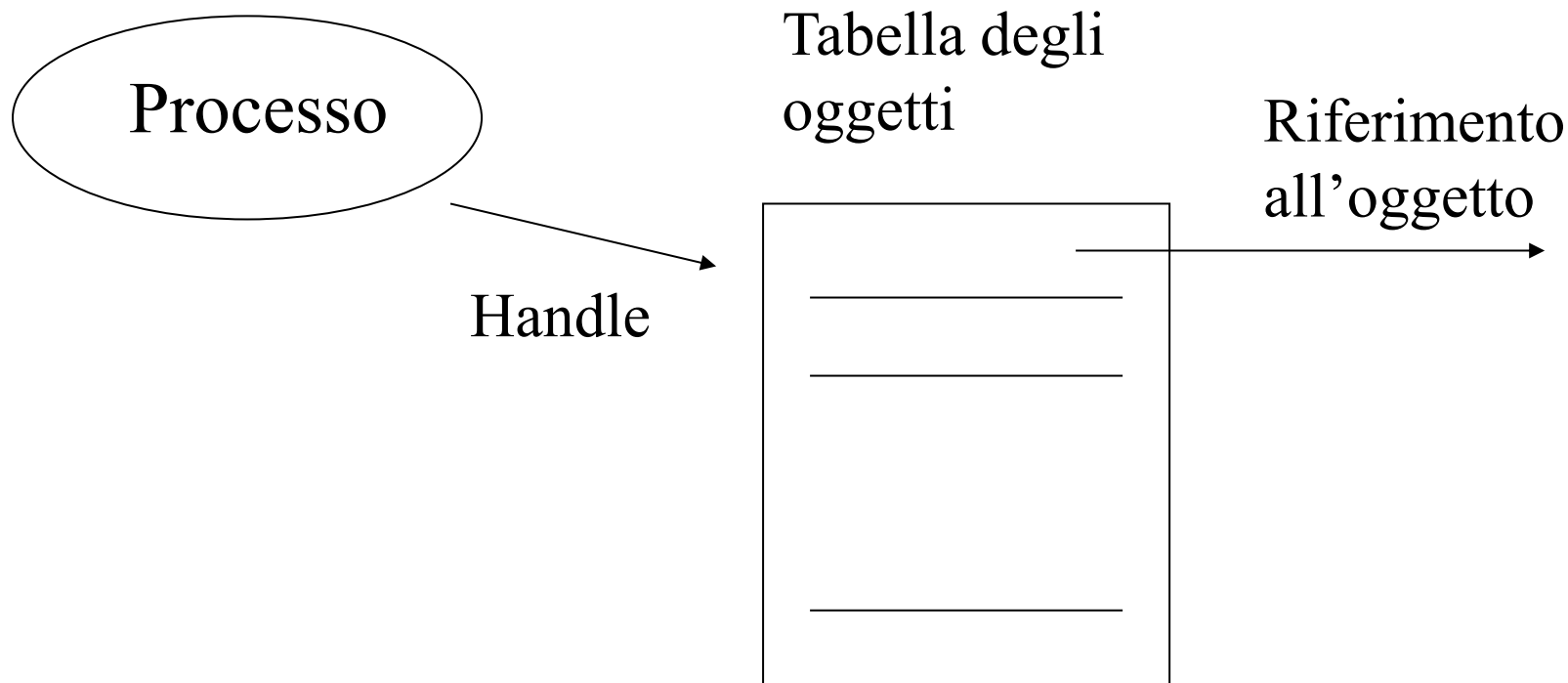
Lettura/scrittura di memoria virtuale

Protezione di memoria virtuale

Blocco/sblocco di memoria virtuale

Handle e tabella degli oggetti

- Ogni processo accede ad oggetti tramite un **handle** (maniglia)
- L'handle implementa un riferimento all'oggetto tramite una tabella degli oggetti propria del processo



Attributi di sicurezza: ereditabilita' degli handle

```
typedef struct _SECURITY_ATTRIBUTES {  
    DWORD nLength;  
    LPVOID lpSecurityDescriptor;  
    BOOL bInheritHandle;  
} SECURITY_ATTRIBUTES
```

Descrizione

- struttura dati che specifica permessi

Campi

- nLength: va settato SEMPRE alla dimensione della struttura
- lpSecurityDescriptor: puntatore a una struttura SECURITY_DESCRIPTOR
- bInheritHandle: se uguale a TRUE un nuovo processo puo' ereditare l'handle a cui fa riferimento questa struttura

Creazione di un processo

```
BOOL CreateProcess( LPCTSTR lpApplicationName,  
                    LPTSTR lpCommandLine,  
                    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
                    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
                    BOOL bInheritHandles,  
                    DWORD dwCreationFlags,  
                    LPVOID lpEnvironment,  
                    LPCTSTR lpCurrentDirectory,  
                    LPSTARTUPINFO lpStartupInfo,  
                    LPPROCESS_INFORMATION lpProcessInformation)
```

Descrizione

- invoca la creazione di un nuovo processo (creazione di un figlio)

Restituzione

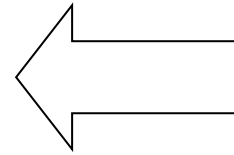
- nel chiamante: un valore diverso da zero in caso di successo, 0 in caso di fallimento.

Parametri

- `lpApplicationName`: stringa contenente il nome del file da eseguire
- `lpCommandLine`: stringa contenente l'intera riga di comando del programma.
- `lpProcessAttributes`: puntatore a una struttura `SECURITY_ATTRIBUTES` che specifica se l'handle del nuovo processo puo' essere ereditata da processi figli
- `lpThreadAttributes`: puntatore a una struttura `SECURITY_ATTRIBUTES` che specifica se l'handle del primo thread del nuovo processo puo' essere ereditata da processi figli.
- `bInheritHandles`: se e' `TRUE` ogni handle ereditabile del processo padre viene automaticamente ereditato dal processo figlio
- `dwCreationFlags`: opzioni varie (per es. La priorit )
- `lpEnvironment`: Puntatore a una struttura contenente l'ambiente del processo. `NULL` eredita l'ambiente del processo padre
- `lpCurrentDirectory`: stringa contenente la directory corrente del processo
- `lpStartupInfo`: Puntatore a una struttura `STARTUPINFO`
- `lpProcessInformation`: puntatore a struttura `PROCESS_INFORMATION` che riceve informazioni sul processo appena creato.

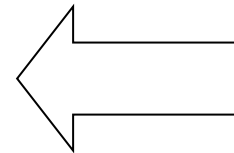
Strutture dati

```
typedef struct _PROCESS_INFORMATION {  
    HANDLE hProcess;  
    HANDLE hThread;  
    DWORD dwProcessId;  
    DWORD dwThreadId;  
} PROCESS_INFORMATION;
```



windows.h

```
typedef struct _STARTUPINFO {  
    DWORD cb;  
    .....  
    .....  
    .....  
} STARTUPINFO
```



windows.h

Un esempio

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

int main(int argc, char *argv[]) {
    BOOL newprocess;  STARTUPINFO si; PROCESS_INFORMATION pi;
    memset(&si, 0, sizeof(si));  memset(&pi, 0, sizeof(pi)); si.cb = sizeof(si);
    newprocess = CreateProcess(".\\figlio.exe",
                              ".\\figlio.exe pippo pluto",
                              NULL,
                              NULL,
                              FALSE,
                              NORMAL_PRIORITY_CLASS,
                              NULL,
                              NULL,
                              &si,
                              &pi);

    if (newprocess == FALSE) { printf("Chiamata fallita\n") };
}
```

ASCII vs UNICODE

- ANSI-C e standard successivi si basano su codifica ASCII
- Stessa cosa e' vera per sistemi operativi della famiglia Unix (seppur UTF-8 viene utilizzato sui terminali)
- Windows utilizza codifica UNICODE (4 byte per carattere)
- E' compito del programmatore e/o dell'ambiente di compilazione risolvere la dicotomia
- Di fatto gli stub delle system call Windows che trattano di stringhe di caratteri hanno sempre versioni duali, una ASCII una UNICODE
- Ogni stub di system call ha quindi 3 forme, una anonima, in termini di codifica dei caratteri, e due non anonime
- E' compito del settaggio di compilazione determinare il mapping della forma anonima su quella non anonima

Tornando a CreateProcess

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

```
    ...
```

```
    ...
```

```
    CreateProcess (...)
```

```
    ...
```

```
    ...
```

```
}
```

Compile with
UNICODE directive

CreateProcessW (...)

This requires
already
UNICODE
strings as input

Compile with
ASCII directive

CreateProcessA (...)

Simple redirections via `#ifdef` directives

Modalita' generale per la codifica dei caratteri

Basata sull'uso di TCHAR e sulla macro _UNICODE:

```
#ifdef _UNICODE
typedef wchar_t TCHAR;
#else
typedef char TCHAR;
#endif
```

Ad esempio:

TCHAR name[MAX_SIZE]

typedef char TCHAR;

char name[MAX_SIZE]

ASCII

typedef wchar_t TCHAR;

wchar_t name[MAX_SIZE]

UNICODE

Notazioni in espressioni i tipo carattere

```
char oneChar = 'x';
```

```
wchar_t oneChar = L'x'
```

Esempi di API per la manipolazione di stringhe in ASCII e UNICODE

C++

```
void TEXT(  
    LPTSTR string  
);
```

Parameters

string

Pointer to the string to interpret as UTF-16 or ANSI.

TCHAR string

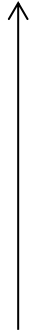


Return value

This macro does not return a value.

U esempio di uso

```
TCHAR message[] = TEXT ("Ciao a tutti!")
```



Puntatore riutilizzabile come input alla macro TEXT ()

Typedef per la gestione delle stringhe in WINAPI

Typedef	Definition
CHAR	<code>char</code>
PSTR or LPSTR	<code>char*</code>
PCSTR or LPCSTR	<code>const char*</code>
PWSTR or LPWSTR	<code>wchar_t*</code>
PCWSTR or LPCWSTR	<code>const wchar_t*</code>

Un altro esempio di configurazione manuale: aspetti di sicurezza

Definendo la macro `_CRT_SECURE_NO_WARNINGS` prima di includere header file permette di riconfigurare gli header stessi

Ad esempio, `stdio.h` viene ad offrire realmente l'accesso a funzioni classiche quali `scanf` e `gets` che altrimenti sarebbero non accessibili

Accesso al valore del PID

```
DWORD WINAPI GetCurrentProcessId(void);
```

```
DWORD WINAPI GetProcessId( _In_ HANDLE Process );
```

Terminazione di un processo e relativa attesa

```
VOID ExitProcess(UINT uExitCode)
```

Descrizione

- Richiede la terminazione del processo chiamante

Argomenti

- uExitCode: valore di uscita del processo e di tutti i thread terminati da questa chiamata
-

```
DWORD WaitForSingleObject(HANDLE hHandle,  
                           DWORD dwMilliseconds)
```

Descrizione

- effettua la chiamata wait ad un mutex

Parametri

- hHandle: handle al mutex
- dwMilliseconds: timeout

Restituzione

- WAIT_FAILED in caso di fallimento

Catturare il valore di uscita di un processo

```
int GetExitCodeProcess (  
    HANDLE hProcess,  
    LPDWORD lpExitCode  
)
```

Descrizione

- richiede lo stato di terminazione di un processo

Parametri

- hProcess: handle al processo
- lpExitCode: puntatore all'area dove viene scritto il codice di uscita

Questa system call e' **non bloccante**, e ritorna il valore **STILL_ACTIVE** nel caso in cui il processo target sia ancora attivo

Terminazione su richiesta

C++

```
BOOL WINAPI TerminateProcess(  
    _In_ HANDLE hProcess,  
    _In_ UINT    uExitCode  
);
```

Parameters

hProcess [in]

A handle to the process to be terminated.

The handle must have the **PROCESS_TERMINATE** access right. For more information, see [Process Security and Access Rights](#).

uExitCode [in]

The exit code to be used by the process and threads terminated as a result of this call. Use the **GetExitCodeProcess** function to retrieve a process's exit value. Use the **GetExitCodeThread** function to retrieve a thread's exit value.

Variabili di ambiente

LPTCH WINAPI GetEnvironmentStrings(void);

Descrizione

- acquisizione del valore delle variabili di ambiente

Parametri

- Nessuno

Ritorno

- Puntatore al blocco (sequenza di stringhe) di variabili d'ambiente

DWORD WINAPI GetEnvironmentVariable(_In_opt_ LPCTSTR lpName,
_Out_opt_ LPTSTR lpBuffer, _In_ DWORD nSize);

BOOL WINAPI SetEnvironmentVariable(_In_ LPCTSTR lpName,
_In_opt_ LPCTSTR lpValue);

BOOL WINAPI FreeEnvironmentStrings(_In_ LPTCH lpzEnvironmentBlock);

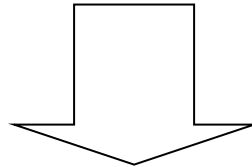
Thread

La nozione di processo ingloba

- il concetto di spazio di indirizzamento proprietario del processo ed il concetto di risorse assegnate al processo stesso
- il concetto di traccia di istruzioni (relazionate al dispatching)

Nei moderni sistemi operativi le due cose possono essere disaccoppiate

- l'unita' base per il dispatching viene denominata thread
- l'unita' base proprietaria di risorse resta il processo in senso classico



Ogni processo puo' essere strutturato come un insieme di thread, ciascuno caratterizzato da una propria traccia di esecuzione

Esempi di sistemi Multithreading sono: NT/2000/....., Solaris/Linux MacOS....

Ambienti multithreading

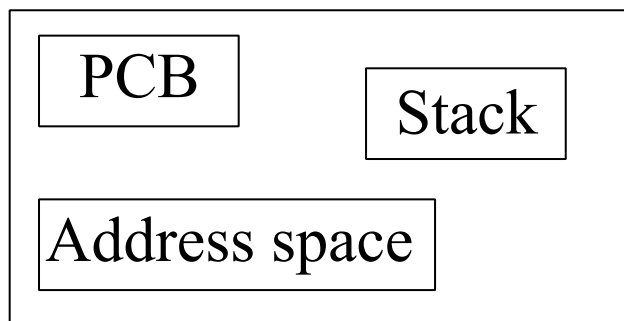
Connotazione di un processo

- spazio di indirizzamento virtuale (immagine del processo)
- protezione e permessi di accesso a risorse (files etc.)

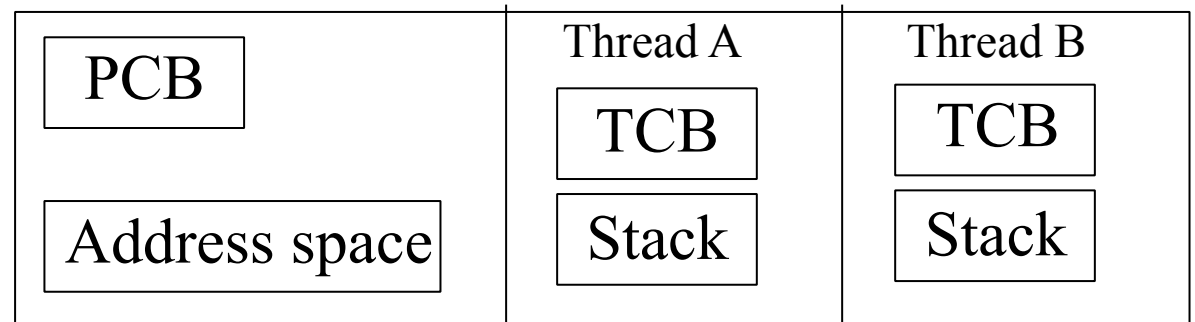
Connotazione di un thread

- stato corrente (Running, Ready, etc.)
 - stack di esecuzione e variabili locali al thread (distinte dalla memoria di processo)
 - in caso il thread non sia nello stato Running, un contesto salvato (valore del registro program counter etc.)
-

Modello classico



Modello multithreading



Windows: attributi principali di oggetti thread

ID del thread

Contesto del thread

Priorita' base del thread (legata a quella di processo)

Priorita' dinamica del thread

Affinita' di processore (insieme di processori utili per
l'esecuzione)

Tempo di esecuzione

Contatore dei blocchi

Stato di uscita

Windows: servizi principali di oggetti thread

Creazione di thread

Apertura di thread

Richiesta/modifica di informazioni di thread

Terminazione di thread

Lettura di contesto

Modifica di contesto

Blocco

Nessun servizio sulla memoria (virtuale)

Creazione di un thread

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES  
                    lpThreadAttributes,  
                    SIZE_T dwStackSize,  
                    LPTHREAD_START_ROUTINE lpStartAddress,  
                    LPVOID lpParameter,  
                    DWORD dwCreationFlags,  
                    LPDWORD lpThreadId)
```

Descrizione

- invoca l'attivazione di un nuovo thread

Restituzione

- un handle al nuovo thread in caso di successo, NULL in caso di fallimento.

Parametri

- `lpThreadAttributes`: puntatore a una struttura `SECURITY_ATTRIBUTES` che specifica se l'handle del nuovo thread puo' essere ereditata
- `dwStackSize`: dimensione dello stack. 0 e' il valore di default
- `lpStartAddress`: puntatore della funzione (di tipo `LPTHREAD_START_ROUTINE`) che deve essere eseguita dal nuovo thread
- `lpParameter`: parametro da passare alla funzione relativa al thread
- `dwCreationFlags`: opzioni varie
- `lpThreadId`: puntatore a una variabile che conterra' l'identificatore del thread

Terminazione di un thread

```
VOID ExitThread(DWORD dwExitCode)
```

Descrizione

- invoca la terminazione di un thread

Parametri

- dwExitCode specifica il valore di uscita del thread terminato

Restituzione

- non ritorna in caso di successo

Catturare il valore di uscita di un thread

```
int GetExitCodeThread(  
    HANDLE hThread,  
    LPDWORD lpExitCode  
)
```

Descrizione

- richiede lo stato di terminazione di un thread

Parametri

- hThread: handle al processo
- lpExitCode: puntatore all'area dove viene scritto il codice di uscita

Restituzione

- 0 in caso di fallimento

Un esempio

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

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

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

```
void ThreadFiglio() {
```

```
    int x;
```

```
    printf("thread figlio, digita un intero per farmi terminare:");
```

```
    scanf("%d",&x);
```

```
    ExitThread(x);
```

```
}
```

```
int main(int argc, char *argv[]) {
```

```
    HANDLE hThread;          DWORD hid;  DWORD exit_code;
```

```
    hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadFiglio,  
                             NULL, NORMAL_PRIORITY_CLASS, &hid);
```

```
    if (hThread == NULL) printf("Chiamata fallita!\n");
```

```
    else { WaitForSingleObject(hThread,INFINITE);
```

```
            GetExitCodeThread(hThread,&exit_code);
```

```
            printf("thread figlio terminato con codice %d\n",exit_code);
```

```
    }
```

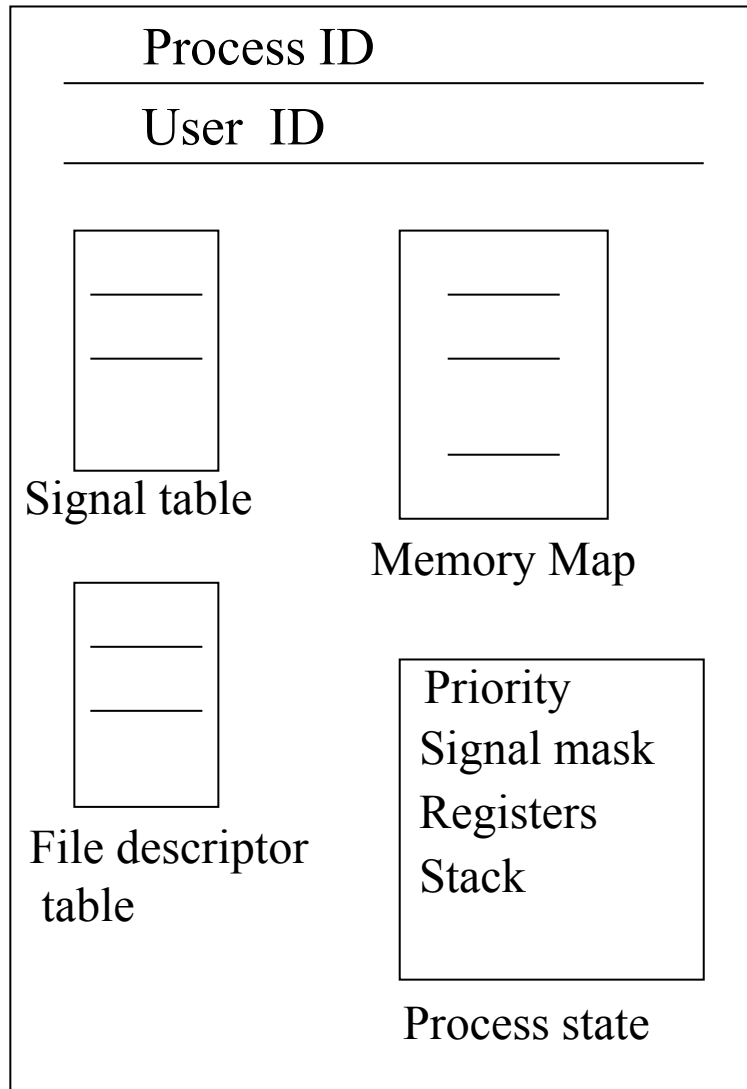
```
    printf("thread padre, digita un intero per farmi terminare:");
```

```
    scanf("%d",&exit_code);
```

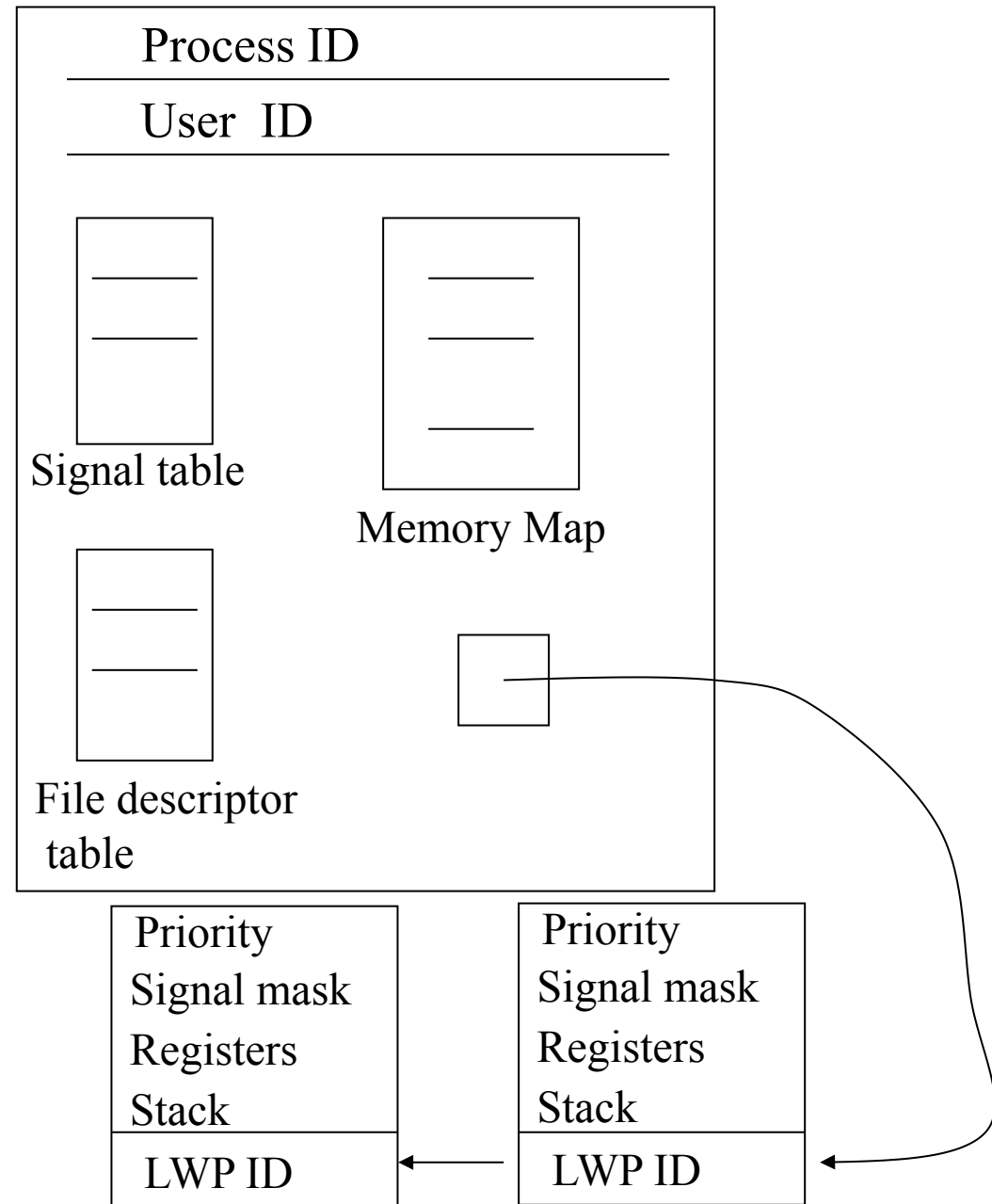
```
}
```

Thread in sistemi UNIX - l'esempio Solaris

Classical UNIX process



Solaris 2.x process



Gestione basica dei thread con Posix

```
int pthread_create(pthread_t *tid, pthread_attr_t *attr, void *(*func)(void*), void *arg)
```

Descrizione invoca l'attivazione di un thread

Parametri

- 1) *tid: buffer di informazioni sul thread
- 2) *attr: buffer di attributi (NULL identifica il default)
- 3) (*func): puntatore a funzione per il thread
- 4) *arg: puntatore al buffer degli argomenti

Restituzione un valore diverso da zero in caso di fallimento

pthread_t e' un **unsigned int**

```
void pthread_exit(void *status)
```

Descrizione invoca la terminazione del thread chiamante

Parametri *status: puntatore al buffer contenente il codice di uscita

Restituzione non ritorna in caso di successo

Sincronizzazione ed identità

```
int pthread_join(pthread_t tid, void **status)
```

Descrizione invoca l'attesa di terminazione di un thread

Parametri

- 1) tid: identificatore del thread (indicativo)
- 2) **status: puntatore al puntatore al buffer contenente il codice di uscita

Restituzione -1 in caso di fallimento, altrimenti l'identificatore del thread terminato


```
pthread_t pthread_self(void)
```

Descrizione chiede l'identificatore del thread chiamante

Restituzione -1 in caso di fallimento, altrimenti l'identificatore del thread


Terminazione in applicazioni multi-thread

Ambienti a processi

`void exit(int)` 

Terminazione dell'unico thread attivo, e quindi dell'intero processo

Ambienti a processi

`void exit(int)` 

Terminazione del thread corente, non necessariamente dell'intero processo

System-call

`void exit_group(int)`



Terminazione dell'intero processo

System-call

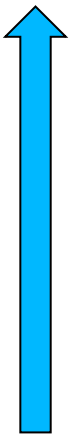
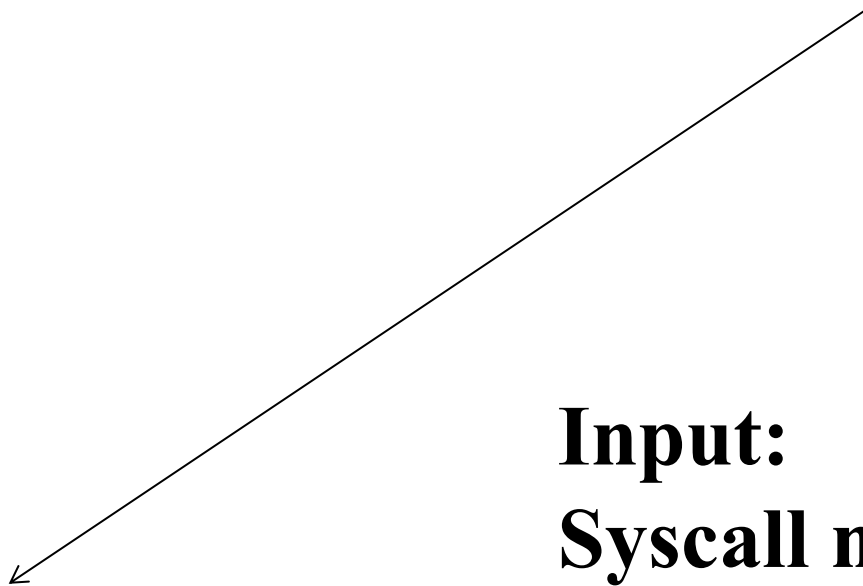
**`exit()` e' mappata su `exit_group()` in `stdlib`
per conformita' a sistemi legacy**

Chiamata esplicita delle system call UNIX tramite stub

Supportata tramite il costrutto `syscall (...)`

Output:
Syscall return-value

Input:
Syscall num, Arg 0, Arg 1



Il kernel: un ambiente nativamente multi-thread

- Tecnologie multi-thread facevano parte della strutturazione di kernel dei sistemi operativi ben prima di renderle disponibile agli sviluppatori applicativi
- Esistevano infatti già percorsi di esecuzione concorrenti privi di immagine user-level (ovvero ‘processi’ concorrenti senza user space code/data/stack, detti kernel threads)
- Un esempio su tutti è l’idle-process
- Altri esempi includono demoni kernel level di natura disparata (e.g. kswapd in sistemi Linux)
- Tutti i kernel threads vivono nello stesso spazio di indirizzamento logico (il kernel stesso, inclusi testo e dati)

Librerie rientranti e non

- Un libreria si definisce rientrante se offre servizi “**thread safe**”
- Questo implica la possibilità di usare la libreria in ambiente multi-thread senza problemi di consistenza sullo stato della libreria stessa (dovuti alla concorrenza)
- Consultare sempre il manuale per verificare se la funzione di libreria che si intende utilizzare è rientrante o non
- Molte librerie comunemente usate sono implementate come rientranti per default (e.g. `printf`, `scanf`, `malloc` ...)
- In taluni casi per motivi di performance (o di sicurezza) esistono versioni duali delle librerie, una non rientrante ed una rientrante
- Le funzioni della versione rientrante sono in genere identiche in specifica di interfaccia a quelle non rientranti, ma hanno il suffisso `_r` nella segnatura)