Sistemi Operativi

Lezione 3
I Processi

Processi (job, task, sequential process)

- Il termine processo è un'astrazione adottata nell'ambito dei sistemi operativi
- denota un programma (sequenza di istruzioni) in esecuzione, nell'ambito di un determinato ambiente esecutivo caratterizzato da:
  - CPU
  - Memoria
  - Risorse di I/O
- I processi sono tra loro scorrelati, cioè le risorse di ciascun processo sono private

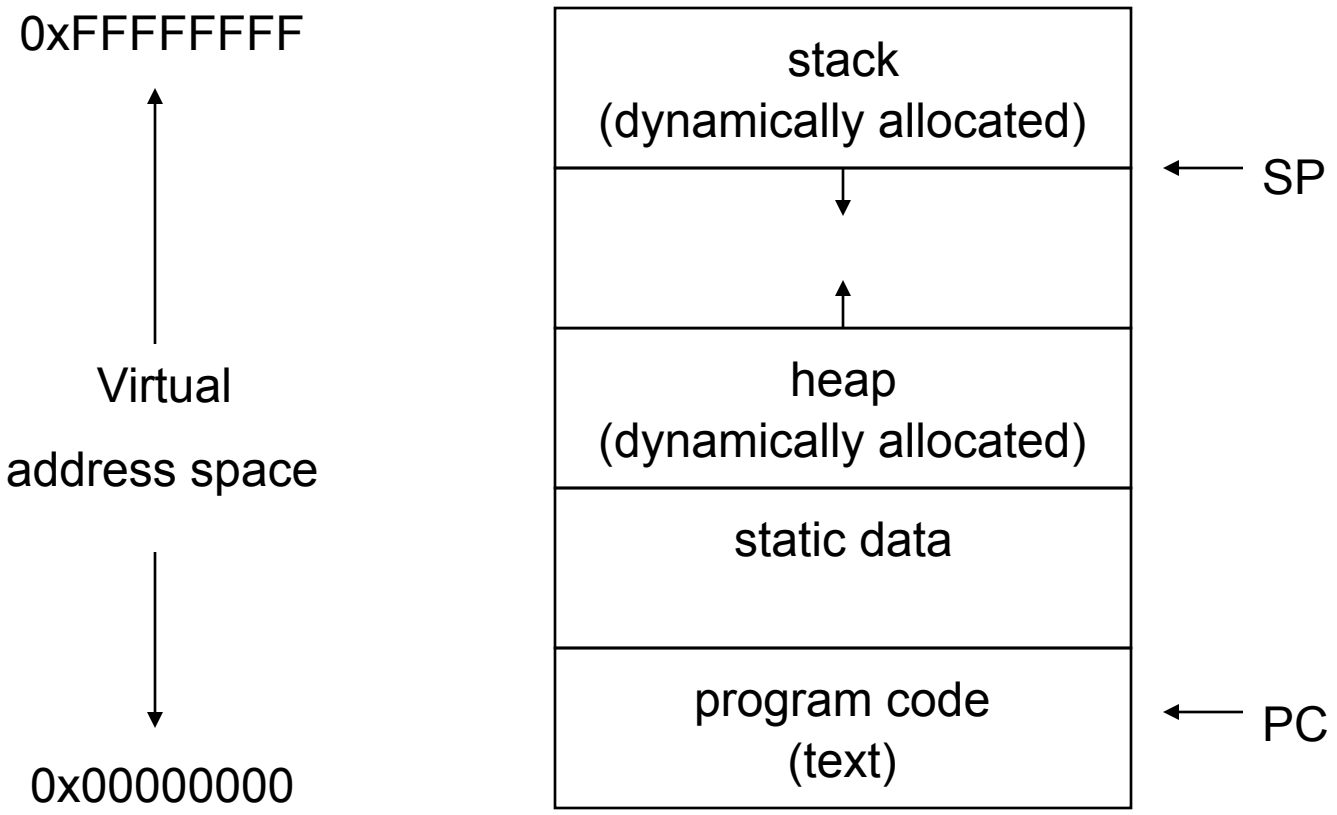
Come viene eseguito un programma?

- Il codice eseguibile viene caricato in memoria
- Viene allocato spazio alle variabili del programma
- Viene allocato uno stack per il processo
- Vengono aggiornati i registri del processore
- Il controllo passa al processo
- Il processo esegue un'iistruzione alla volta:
  - fetch the instruction from memory
  - decode the instruction
  - update the IP
  - execute the instruction

Processi – Contenuti

- Un processo deve quindi comprende (almeno):
  - Uno spazio di indirizzi– di solito protetto e virtuale– all'interno della memoria contrale
  - Il codice da eseguire
  - I dati necessari per l'esecuzione del codice
  - Uno stack e relativo stack pointer (SP)
  - Un program counter (PC)
  - Un insieme di registri – general purpose e di stato
  - Un insieme di risorse del sistema
    - file, network connections, privileges, …

Processi – Spazio d'indirizzamento



Processi

- I processi sono gli oggetti con cui gli utenti di un sistema interagiscono:
  - Web browser
  - Word processors
  - Mail, ecc.

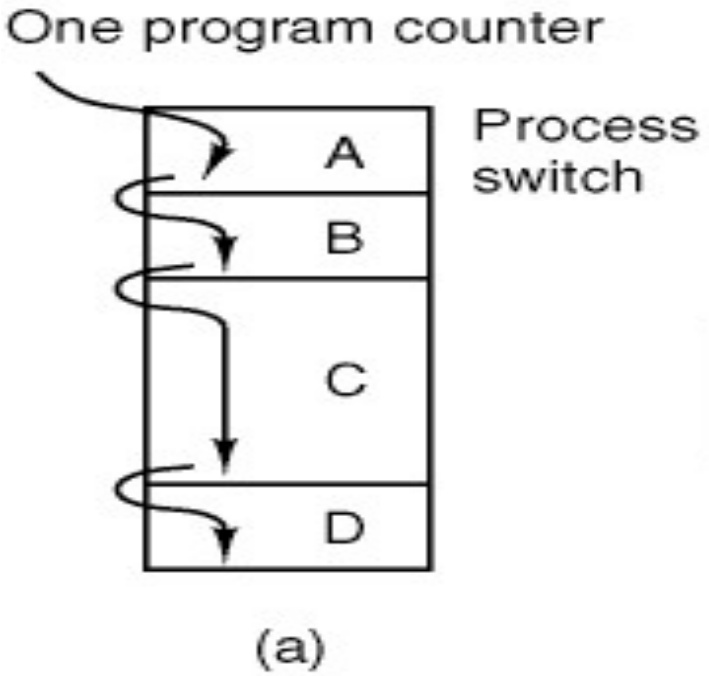
Sistemi mono programmati

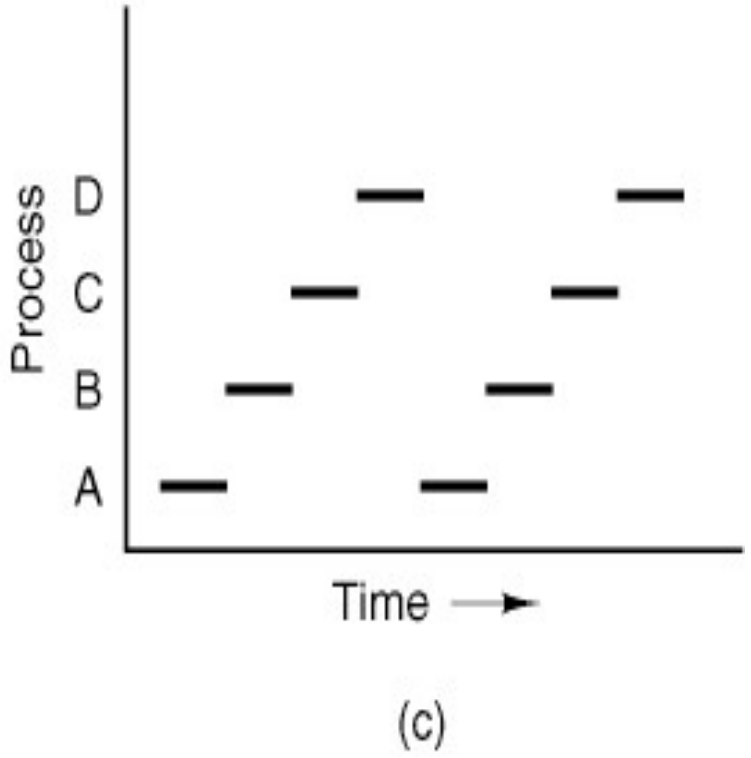
- Viene eseguito un processo per volta
- Non si esegue il nuovo processo sino a che il processo in esecuzione non è terminato
- Esiste un mapping uno a uno tra le risorse del sistema e quelle del processo

Multiprogrammazione (su sistemi mono)

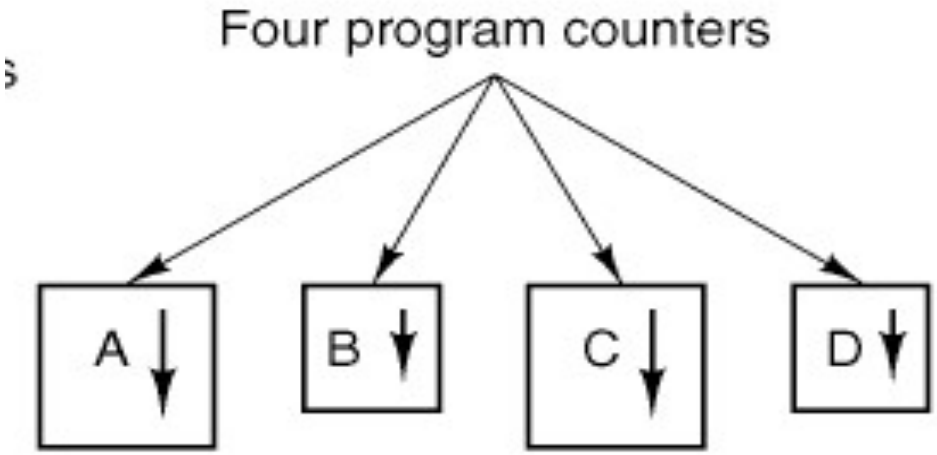
- Presenza contemporanea di più processi
- Una CPU può eseguire un solo processo alla volta
- UN PROCESSO È IN ESECUZIONE quando il suo ambiente esecutivo è la macchina fisica
- Va introdotto un meccanismo che consente di "cambiare in corsa" un processo in esecuzione con un altro
- Mapping molti a uno tra i processi ed il processore

Pseudo parallelismo





Parallelismo



(b)

Le operazioni su un processo

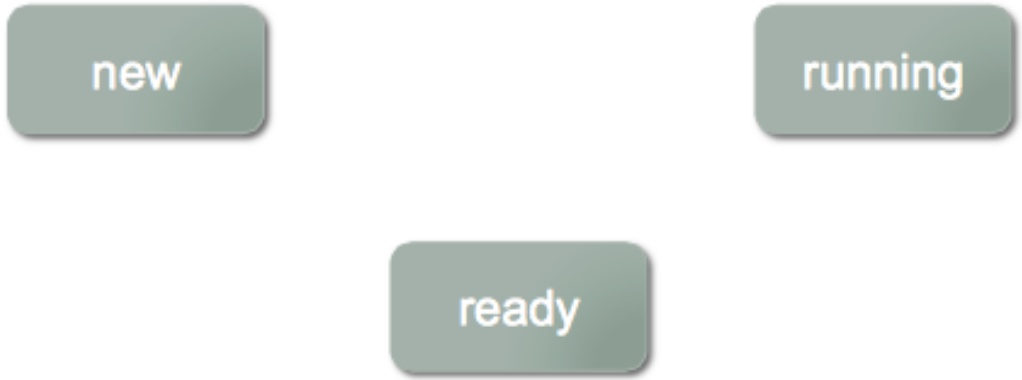
- Compito primario del sistema operativo è quello di gestire i processi
  - Mantenere l'illusione di più processi in esecuzione contemporaneamente
  - dando ogni processo una porzione di tempo sulla CPU
  - Gestire le diverse fasi del ciclo di vita di processo dalla sua creazione alla sua terminazione

Creazione di un processo

- Un processo viene creato a seguito di:
  - una richiesta esplicita da parte dell'utente;
  - una richiesta esplicita da parte di un processo (vedi fork())
  - Durante l'inizializzazione del sistema
    - Tramite l'esecuzione di un'apposita routine di sistema

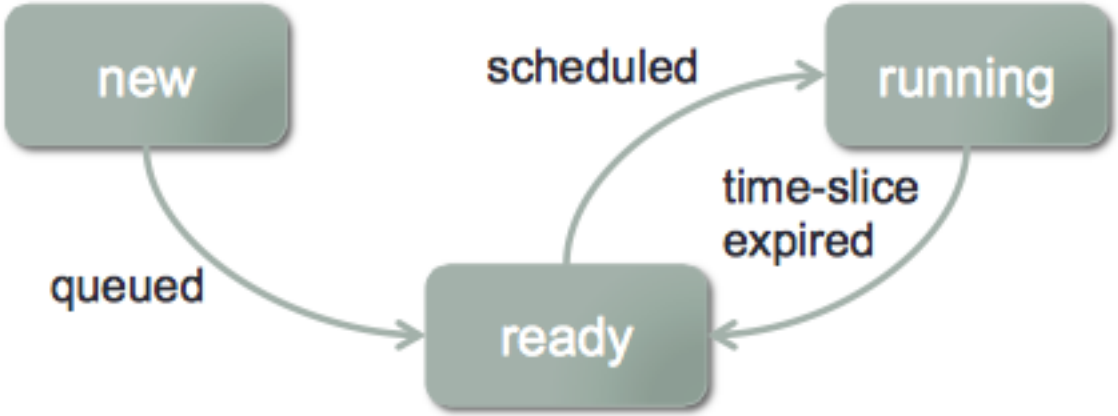
Ciclo di vita dei processi

- Generalmente , vi sono più processi che competono per la CPU
  - Solo un processo può essere in esecuzione su ogni CPU alla volta
  - Quando un processo è in " running “, significa che detiene la CPU !
  - Altri processi che potrebbero essere eseguiti, ma attualmente non hanno la CPU , sono in stato di ”ready"



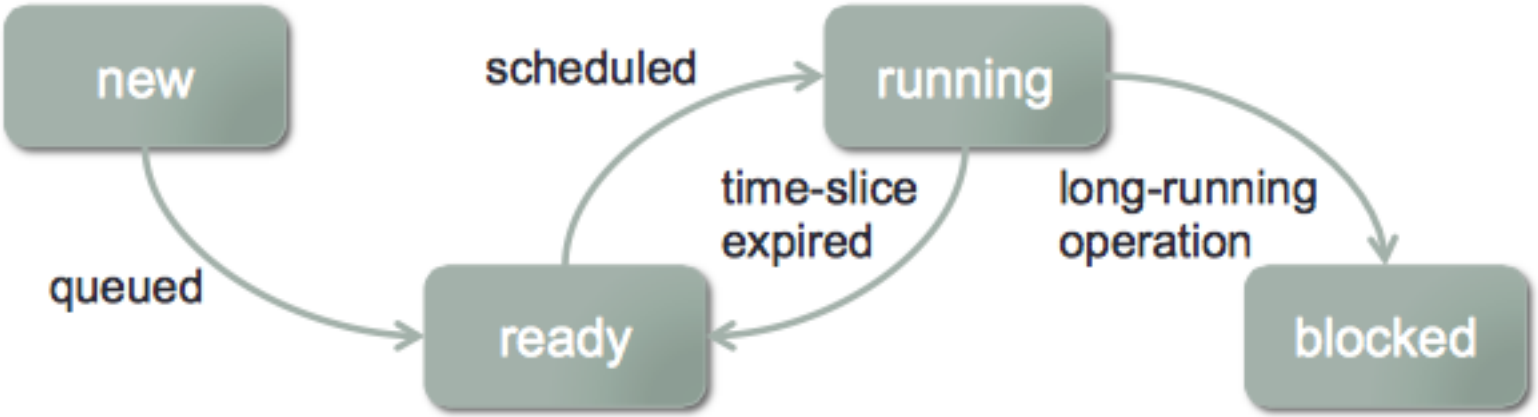
Ciclo di vita dei processi

- Processo nuovi non necessariamente ottengono subito al CPU
- Il SO gestisce il “passaggio” della CPU da un processo all’altro



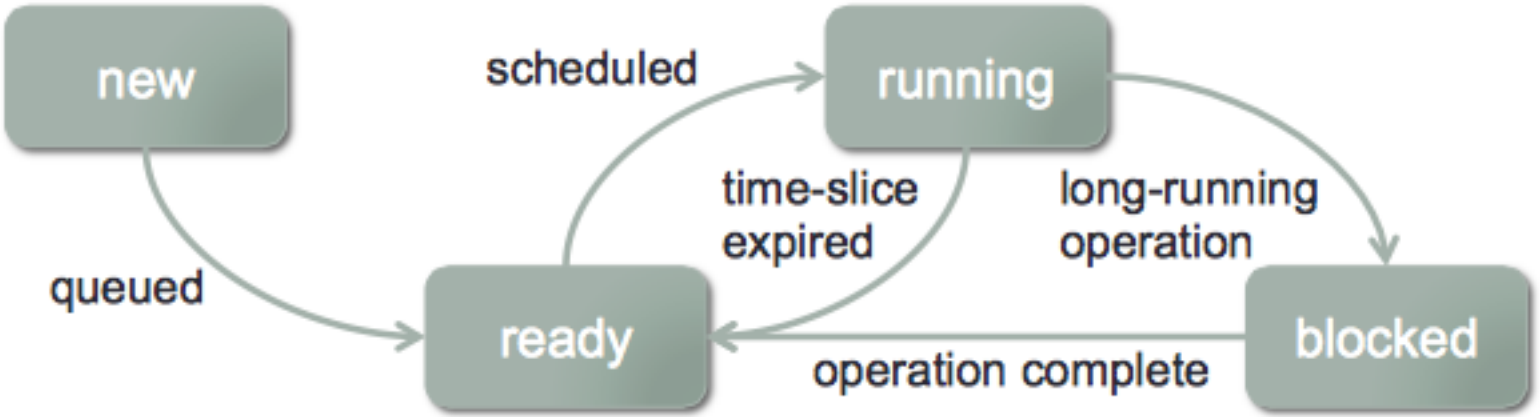
Cliclo di vita dei processi

- I processi devono spesso svolgere operazioni lente (long-running operation)
- In questo caso So provvede a togliere loro la CPU per cederla ad altri processi



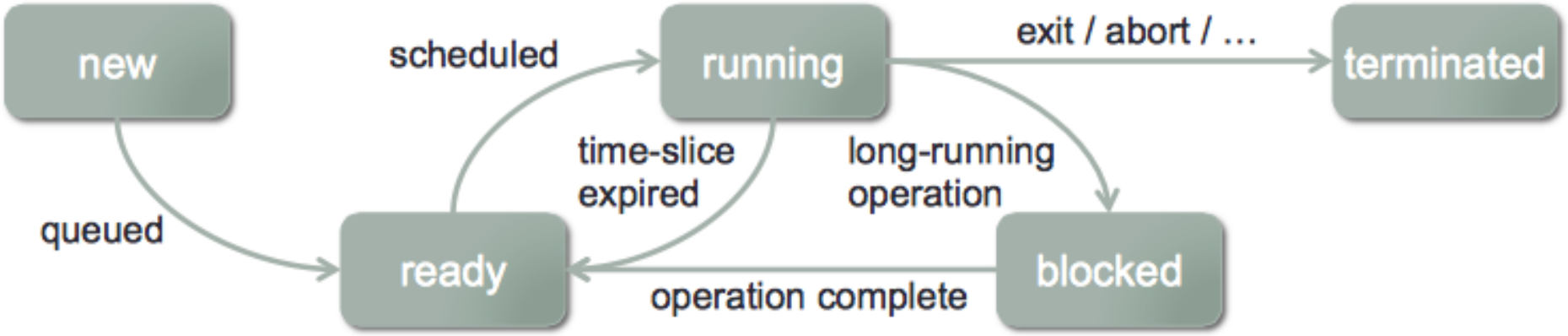
Ciclo di vita dei processi

- Quando l'operazione lenta termina, il processo che l'ha eseguita viene rimesso in ready queue



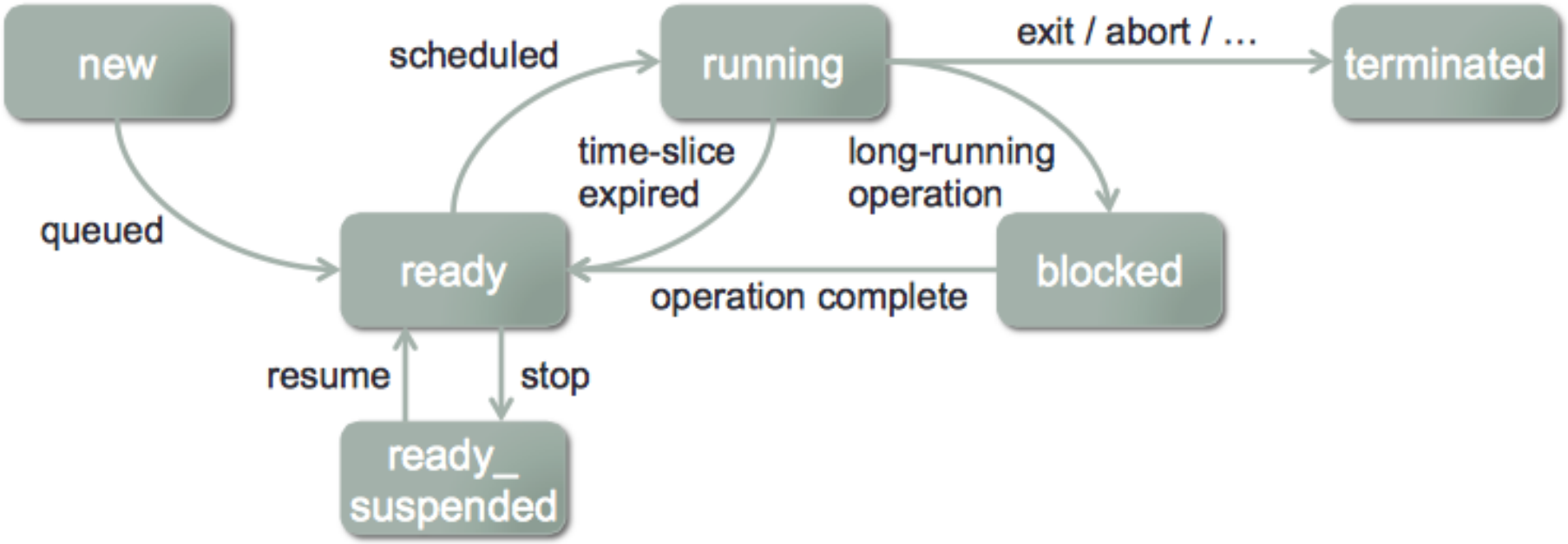
Ciclo di vita di un processo

- Primo o poi i processi terminano :
  - Terminazione normale (esecuzione dell'istruzione exit)
  - Scadenza del tempo di permanenza nel sistema
  - Memoria non disponibile
  - Violazione delle protezioni/Errori durante l'esecuzione



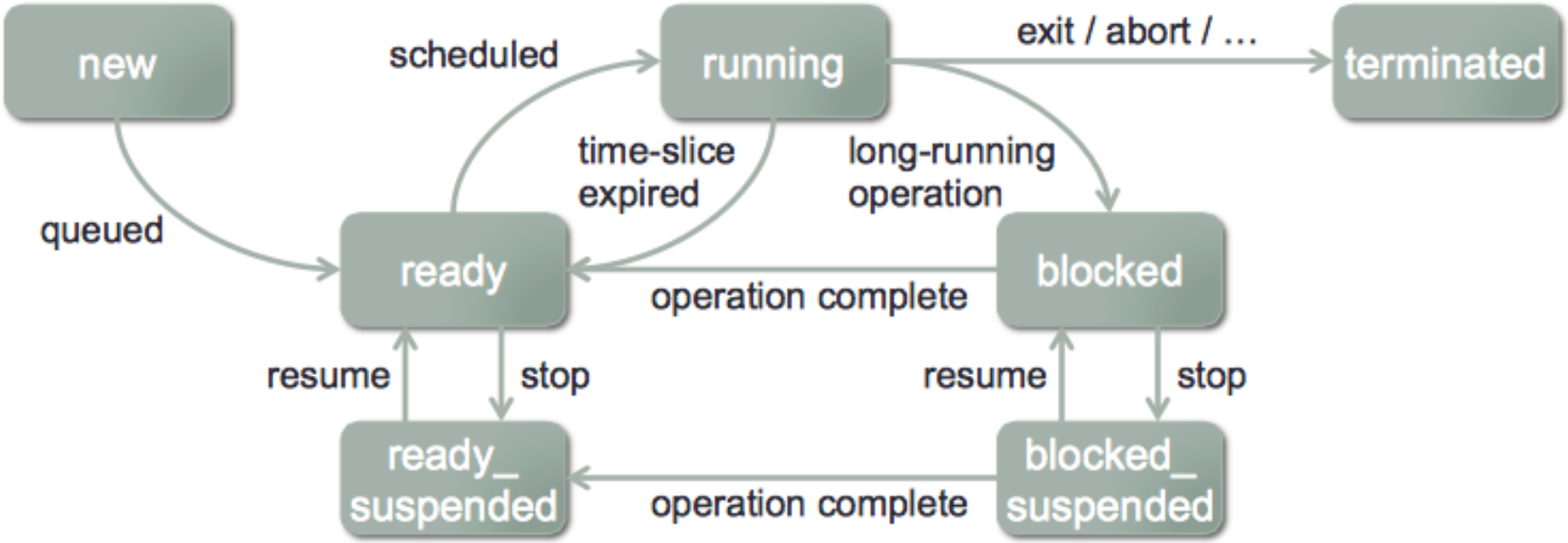
Ciclo di vita di un processo

- Quelli che abbiamo visto sono gli stati fondamentali presenti in tutti i SO, esistono poi ulteriori stati "facoltativi"
- Suspend/resume
  - un utente sospende un processo con Ctrl-Z
  - Un processo può sospendere un altro processo con



Ciclo di vita di un processo

- Un processo sospeso può essere in stato di blocked



Evoluzione dei Processi (3)

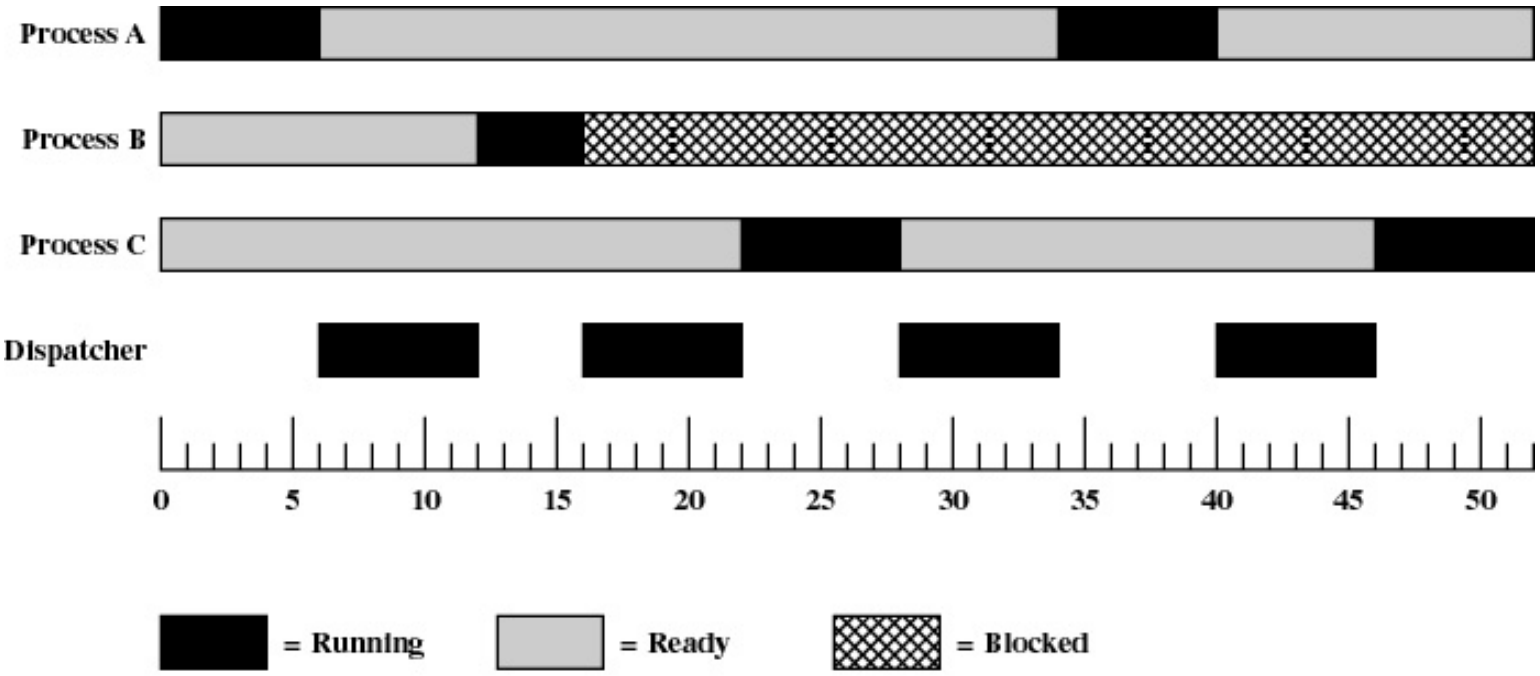
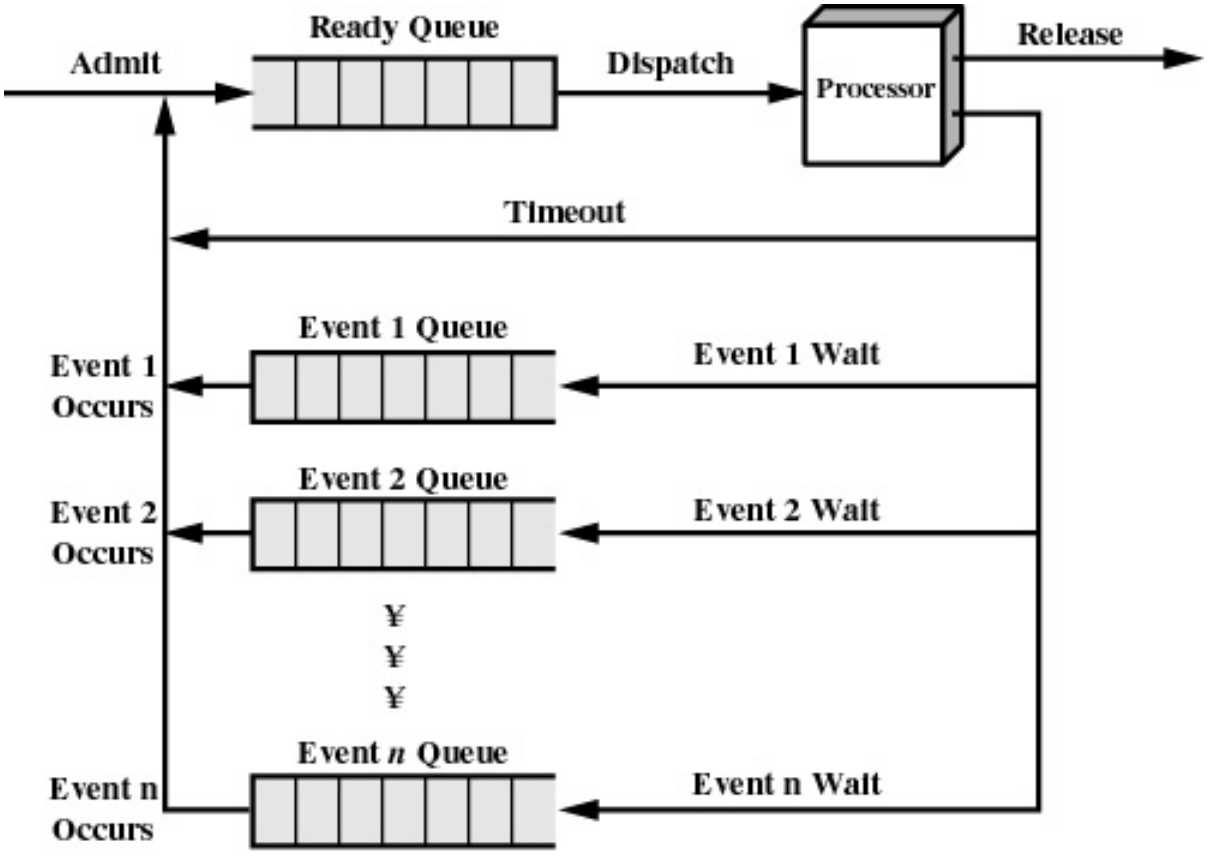


Figure 3.6 Process States for Trace of Figure 3.3

Evoluzione dei processi



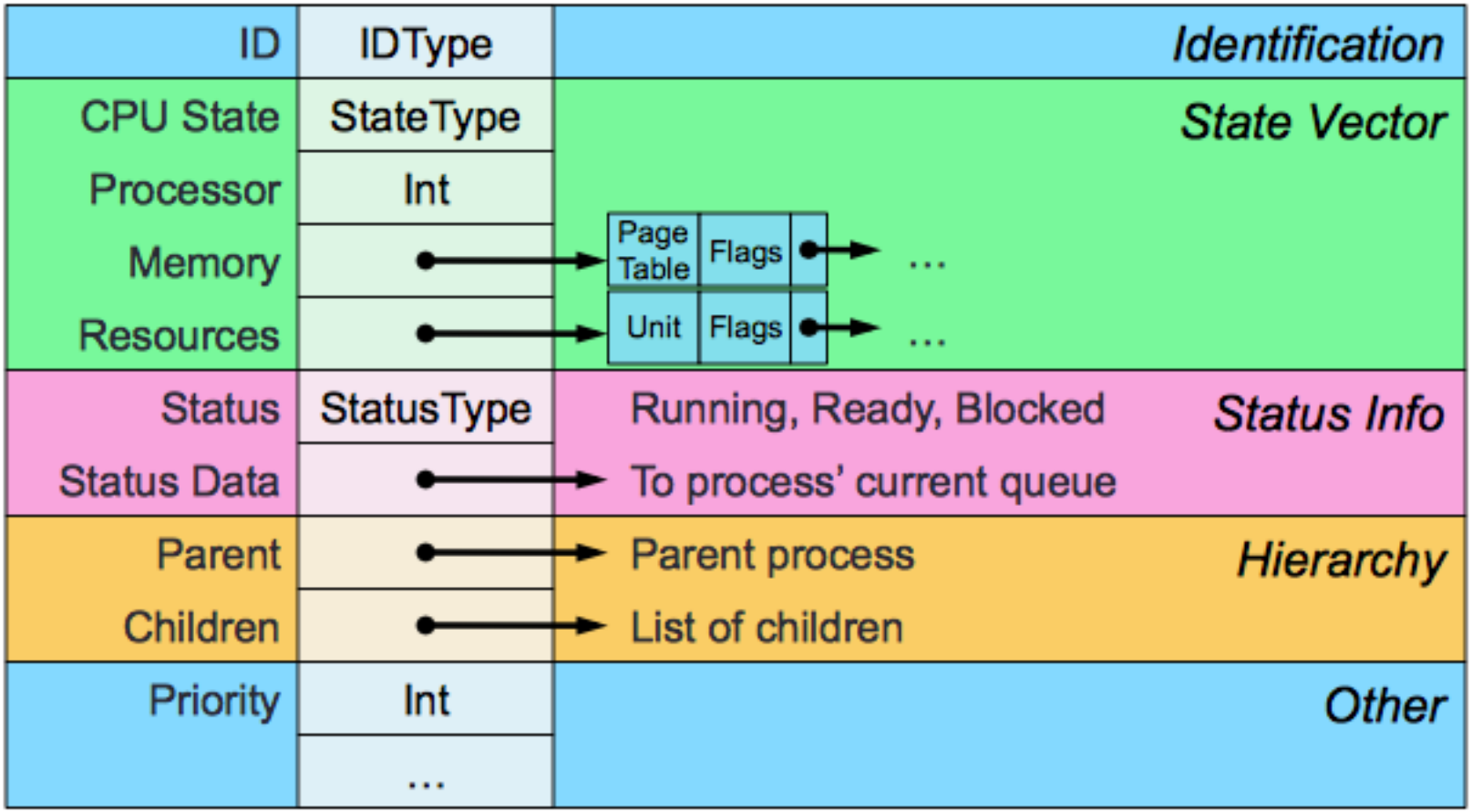
(b) Multiple blocked queues

Context switch (1)

- Per poter implementare il ciclo di vita un processo è necessario individuare un meccanismo che consenta di “togliere” il processore dal processo che lo sta utilizzando a favore di un altro processo
- La sospensione di un processo in esecuzione, a favore di un altro processo avviene attraverso l’operazione di context switch
- Il contesto di un processo è il valore delle variabili che caratterizzano il suo ambiente esecutivo (vedi slide successive)

Process Controlo Block

- I contesto di un processo è memorizzato in un'apposita struttura dati del kernel nota come PCB



Identification

- Le informazioni per l'identificazione del processo contengono:
  - Identificatore del processo
  - Identificatore del processo padre
  - Identificatore dell'utente proprietario del processo

State vector (1)

- Lo stato del processore
  - Tutti i registri del processore
  - PC (EIP)
  - Condition codes (EFLAGS)
  - Variabili di stato: flag di interrupt, execution mode
  - Stack pointer: puntatori allo stack associato al processo
- Gestione della Memoria
  - Tabelle delle pagine o dei segmenti
- Risorse utilizzate
  - File aperti e/o creati

Status info (3)

- Process state: running, ready, waiting, halted.
- Eventi: identificativo dell'evento di cui il processo è eventualmente in attesa

Other

- Priorità di scheduling
- Informazioni per l'algoritmo di scheduling: tempo di permanenza nel sistema, tempo di CPU,
- Variabili per la comunicazione tra processi

Context Switch

- Quando il kernel decide che deve essere eseguito un altro processo effettua un context switch, per forzare il sistema ad operare nel contesto del nuovo processo
- Per effettuare un context switch, il kernel:
  - salva le informazioni sufficienti a ripristinare, in un tempo successivo, il contesto del processo sospeso
  - carica il contesto del nuovo processo sulla macchina fisica

Quando si effettua un context switch

- Il context swith è un'operazione interrupt driven che viene svolta dal kernel a seguito di:
  - Clock interrupt
  - I/O interrupt
  - Memory fault
  - Eccezioni

Context switch in dettaglio

- Durante un context switch le operazioni svolte sono:
  - Salvataggio del contenuto dei registri di CPU
  - Ripristino del contenuto dei registri di CPU con i nuovi contenuti
  - Esecuzione di interrupt handler
  - Esecuzione dello scheduler
  - Flush della pipeline del processore
  - Flush della TLB

Il costo di un context swith

- Molto difficile da determinare perché in funzione del tipo di processore
- Mediamente il costo varia tra i 2 e i 7 microsecondi
- Può però arrivare a punte di 50 microsecondi

context switch

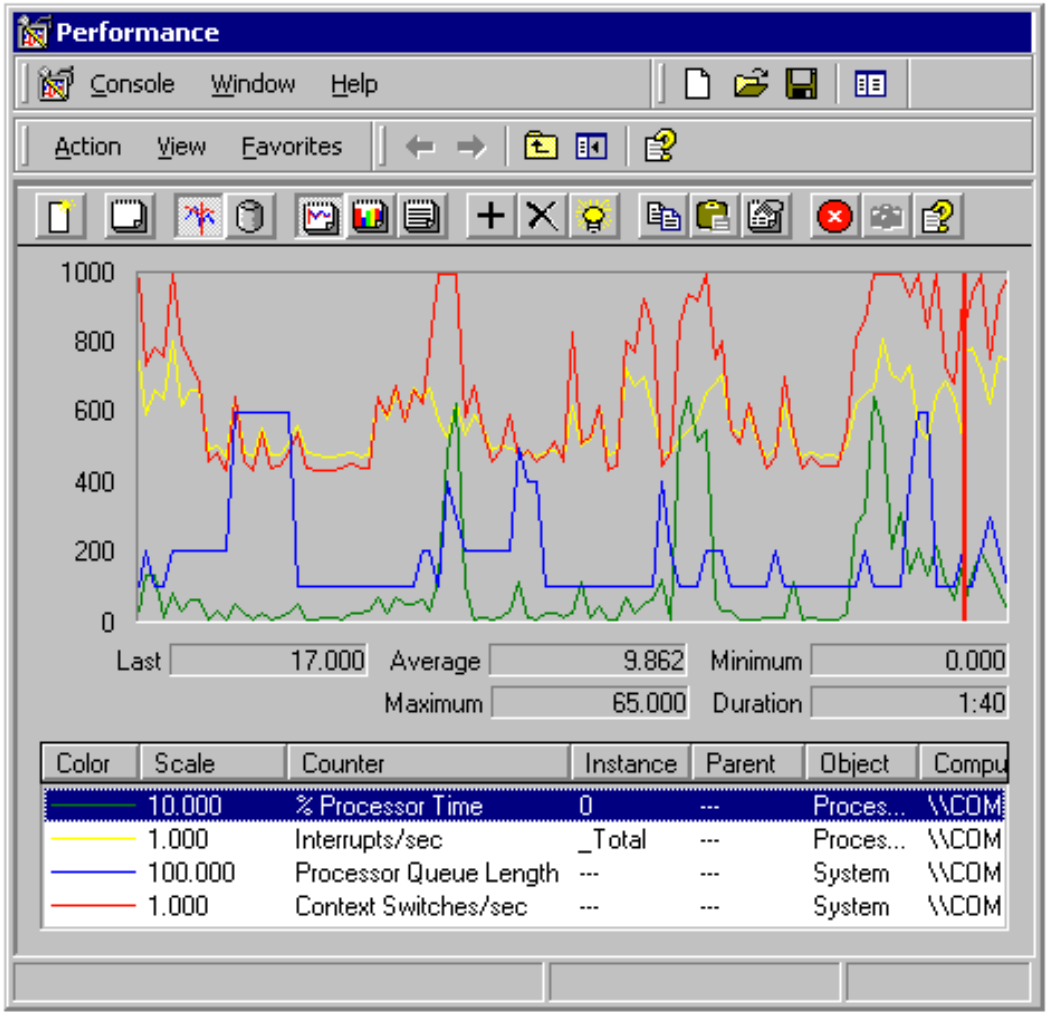


Figure 7.5 Systemwide Context Switches During a Processor Bottleneck

Mode switch

- Durante l'esecuzione di un processo vi sono delle transizioni da user mode a kernel mode (es. syscall, page fault)
- Durante queste transizioni il kernel salva le informazioni necessarie per riprendere successivamente l'esecuzione del processo momentaneamente sospeso
- In questo caso si parla di mode change ma non di context switch

mode switch in dettaglio

- Durante un context switch le operazioni svolte sono:
  - Salvataggio di alcuni dei registri di CPU
  - Ripristino del contenuto dei registri di CPU con i nuovi contenuti
  - Esecuzione di interrupt handler
  - Flush della pipeline del processore
- Un mode switching costa mediamente un centinaio di nano secondi

Processi: aspetti realizzativi

Interrupt/Eccezioni

- Meccanismi introdotti per interrompere il ciclo fetch-decode-execute delle Cpu e consentire l'esecuzione di attività alternative al processo in esecuzione
- Il modo con cui l'hw reagisce ai due eventi sono molto simili, vediamo di seguito le modalità di gestione degli interrupt

Interrupt handling

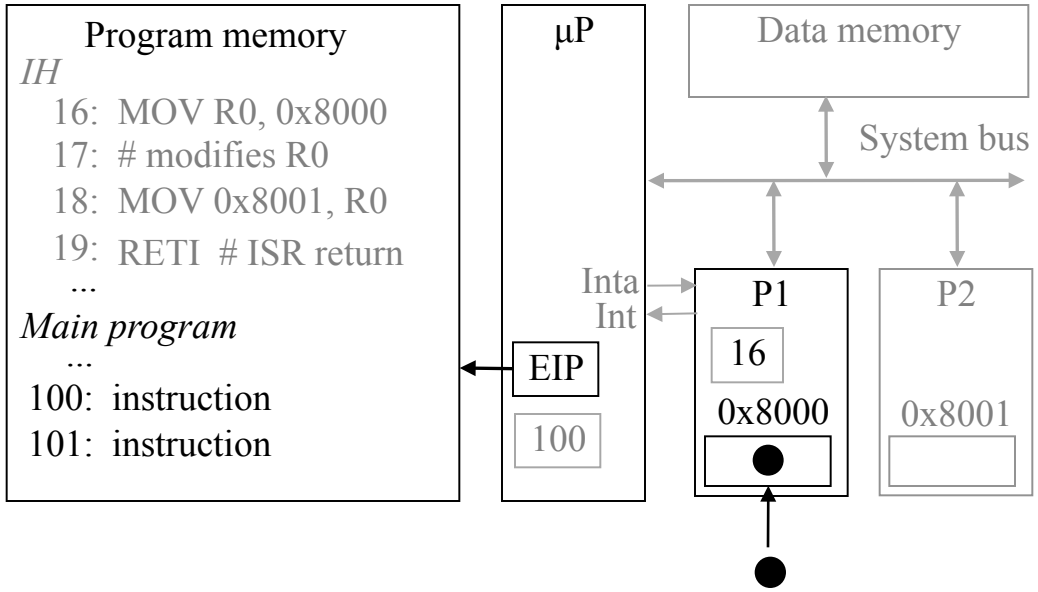
- Interrupt controller segnala l'occorrenza di un interrupt, e passa il numero dell'interrupt (vettore)
- Il processore usa il vettore dell'interrupt per decidere quale handler attivare
- Il processore interrompe il processo corrente PROC, e ne salva lo stato (contesto)
- Il processore salta a un interrupt handler
- Quando l'interrupt è stato gestito, lo stato di PROC viene ripristinato e PROC riprende l'esecuzione da dove era stato sospeso

Interrupt handling

(a): CPU sta eseguendo l'istruzione 100 di un programma

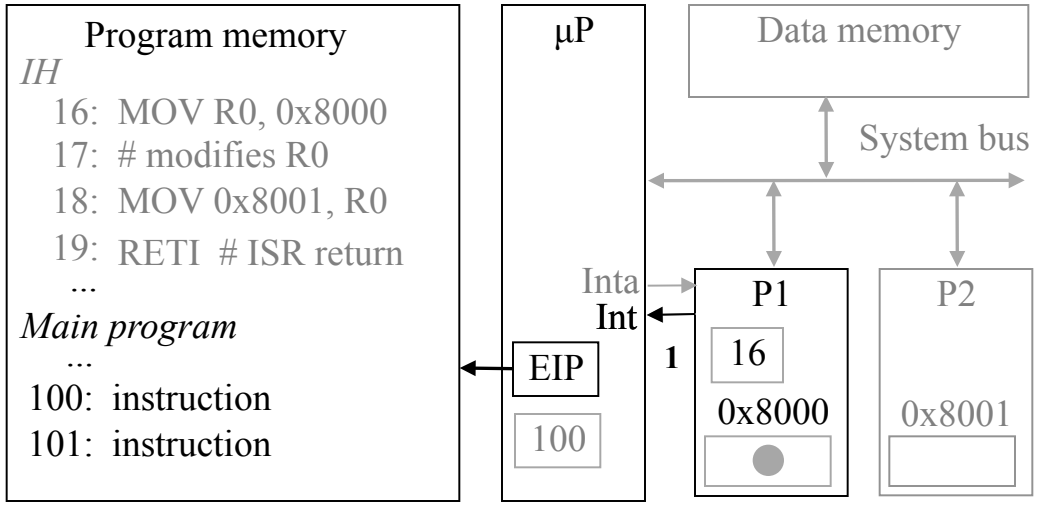
Concorrentemente

(b): P1 acquisisce dei dati in un suo registro di indirizzo 0x8000



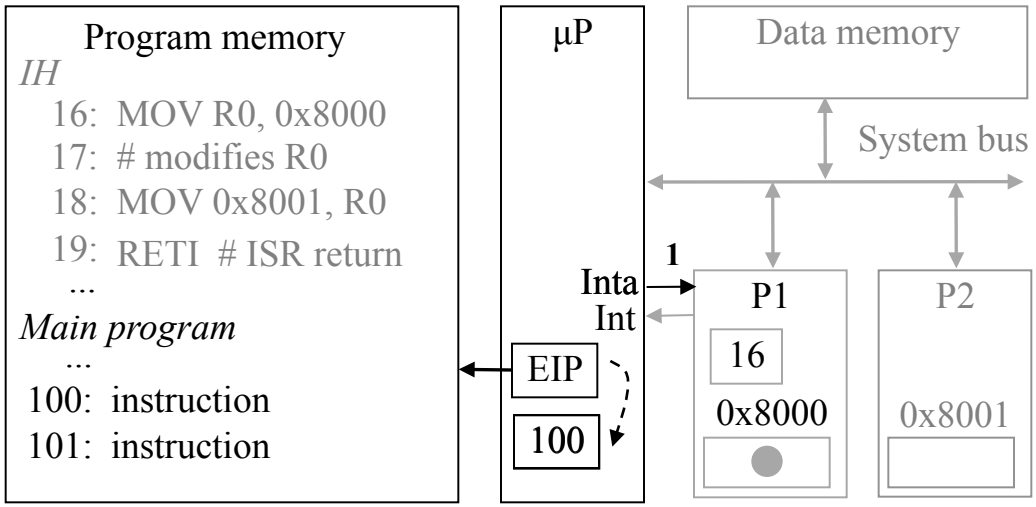
Interrupt handling

2: P1 asserisce il segnale *Int* per richiedere l'intervento del microprocessore



Interrupt handling

3: Dopo aver completato l'istruzione in esecuzione, il processore sente il segnale INT asserito, salva il valore di EIP e asserisce ***Inta***

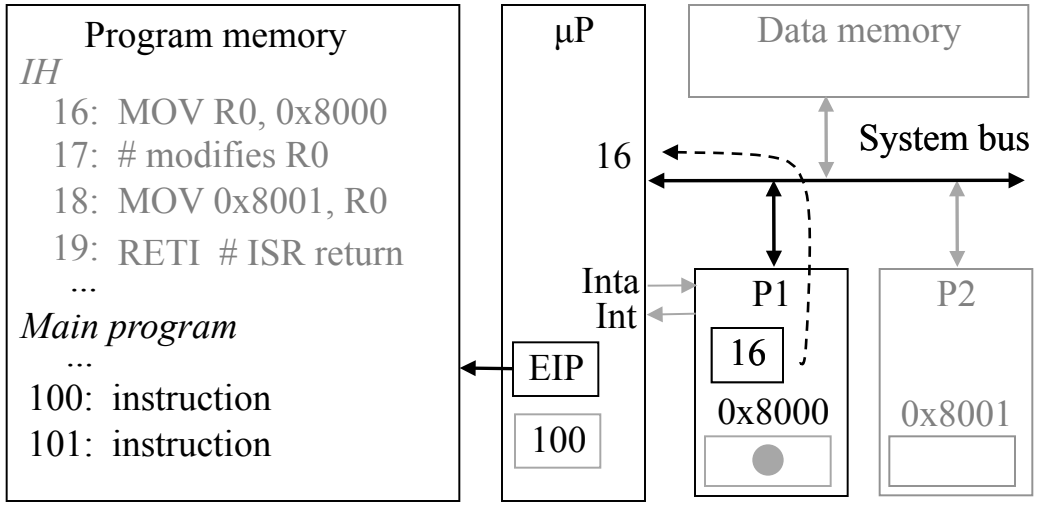


Interrupt Handling

4(a): P1 rileva *Inta* e abbassa il segnale di Int

4(b): il processore riasserisce *Inta*

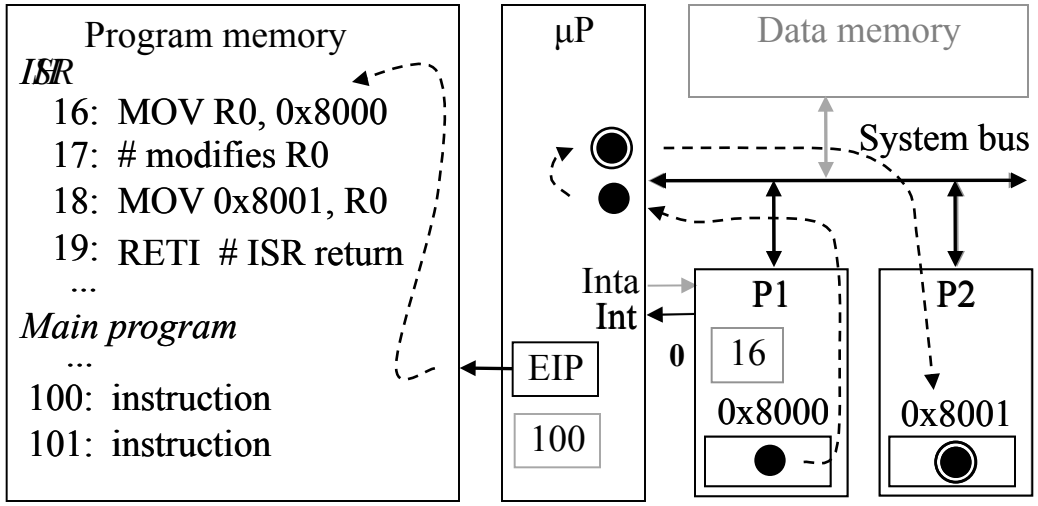
4(c): P1 rileva di nuovo Inta e pone il vettore dell'interrupt (**16**) sul bus dati



Interrupt handling

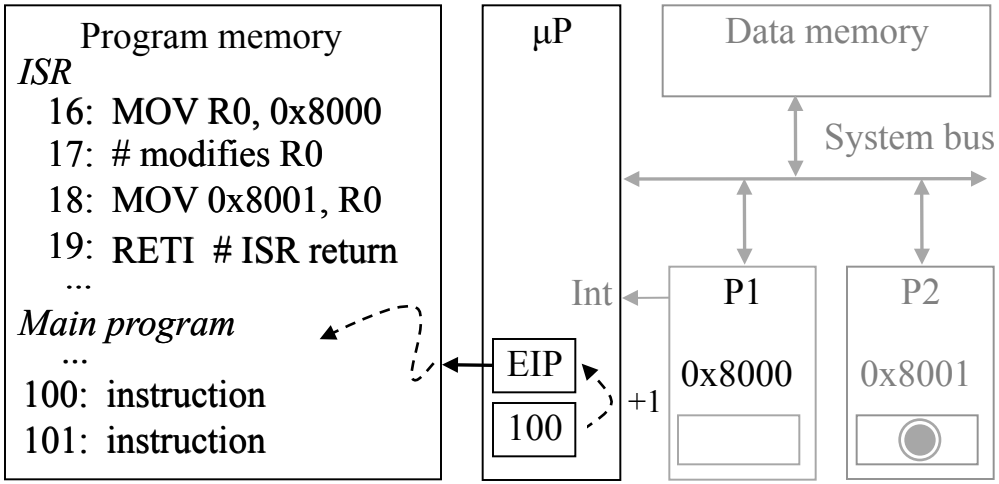
5(a): il processore salta all'interrupt handler associato all'interrupt 16.

L'handler legge il dato da 0x8000, lo modifica e lo riscrive all'indirizzo 0x8001.

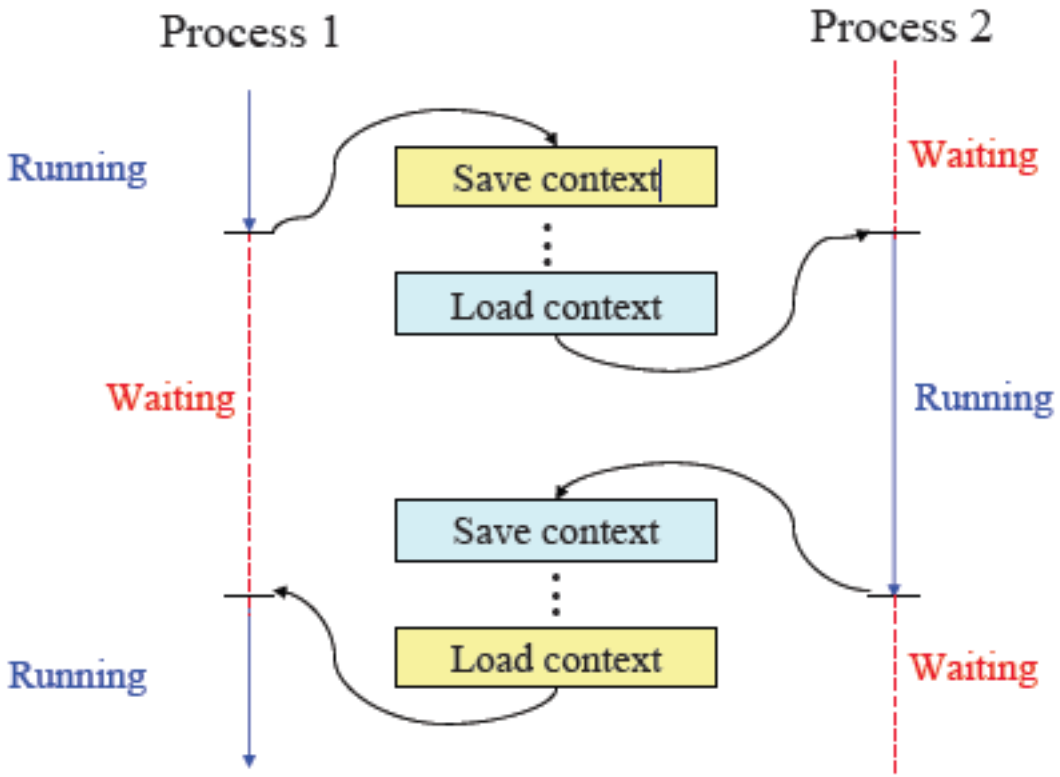


Interrupt handling

6: Alla fine l'handler, esegue l'istruzione RETI che ripristina il valore di EIP a 100+1=101, da dove il processore riprende l'esecuzione



Context Switch



Strutture Dati

- Per poter effettuare questa operazione efficacemente, il sistema operativo usa una particolare struttura dati, detta Process Control Block (PCB)
- All'avvio di ogni processo il sistema provvede a costruire il PCB per il nuovo processo ed inserirlo nella tabella dei processi
- Il PCB è deallocato quando il processo termina
- Esiste un PCB distinto per ogni processo
- Il PCB è struttura dati del kernel e risiede nella zona di memoria corrispondente

JOS PCB (ENV)

```
struct Env {
    struct Trapframe env_tf;    // Saved registers
    struct Env *env_link;       // Next free Env
    envid_t env_id;             // Unique environment identifier
    envid_t env_parent_id;      // env_id of this env's parent
    enum EnvType env_type;      // Indicates special system enviro
    unsigned env_status;        // Status of the environment
    uint32_t env_runs;          // Number of times environment has
    int env_cpunum;             // The CPU that the env is running on

    // Address space
    pde_t *env_pgdir;           // Kernel virtual address of page dir

    // Exception handling
    void *env_pgfault_upcall;   // Page fault upcall entry point

    // Lab 4 IPC
    bool env_ipc_recving;       // Env is blocked receiving
    void *env_ipc_dstva;        // VA at which to map received page
    uint32_t env_ipc_value;     // Data value sent to us
    envid_t env_ipc_from;       // envid of the sender
    int env_ipc_perm;           // Perm of page mapping received
};
```

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

Envs array (Process Table)

```
// Mark all environments in 'envs' as free, set their env_ids to 0,
// and insert them into the env_free_list.
// Make sure the environments are in the free list in the same order
// they are in the envs array (i.e., so that the first call to
// env_alloc() returns envs[0]).
//
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    int i;
    for (i = 0; i != NENV - 1; ++i) {
        envs[i].env_id = 0;
        envs[i].env_link = &envs[i + 1];
    }
    envs[NENV - 1].env_link = NULL;
    env_free_list = &envs[0];

    // Per-CPU part of the initialization
    env_init_percpu();
}
```

env_create

```
// Allocates a new env with env_alloc, loads the named elf
// binary into it with load_icode, and sets its env_type.
// This function is ONLY called during kernel initialization,
// before running the first user-mode environment.
// The new env's parent ID is set to 0.
//
void
env_create(uint8_t *binary, size_t size, enum EnvType type)
{
    // LAB 3: Your code here.
    struct Env *e;

    int result = env_alloc(&e, 0);
    if (result == -E_NO_FREE_ENV)
        panic("env_create: no free environment (exceeding NENVS)");
    else if (result == -E_NO_MEM)
        panic("env_create: not enough memory");

    load_icode(e, binary, size);  //load the binary and set EIP to its entry point

    e->env_type = type;
```

env_alloc

```
// Allocates and initializes a new environment.
// On success, the new environment is stored in *newenv_store.
//
int
env_alloc(struct Env **newenv_store, envid_t parent_id)
{    int32_t generation;
     int r;
     struct Env *e;
     if (!(e = env_free_list))
          return -E_NO_FREE_ENV;
// Allocate and set up the page directory for this environment.
     if ((r = env_setup_vm(e)) < 0)
          return r;
// Generate an env_id for this environment.
     e->env_id = generation | (e - envs);
// Set the basic status variables.
     e->env_parent_id = parent_id;
     e->env_type = ENV_TYPE_USER;
     e->env_status = ENV_RUNNABLE;
     e->env_runs = 0;
```

Stato processi

- ENV_FREE
- ENV_RUNNABLE (READY)
- ENV_RUNNING
- ENV_NOT_RUNNABLE (WAITING)
- ENV_DYING (Zombie)

Context Switch

- Salva lo stato del processore sul PCB del processo in esecuzione
- Modifica lo stato del suddetto PCB in - ready, blocked, ecc.
- Seleziona dalla tabella dei processi un altro processo per l'esecuzione
- Ripristina con le informazioni contenute nel PCB del nuovo processo lo stato del processore e le strutture per la gestione della memoria

Creazione di un processo via syscall

- La system call fork() crea una copia del processo chiamante
  - *Al termine dell'esecuzione di una fork() da parte di un processo A saranno in esecuzione due processi: il processo A e il processo da lui generato (figlio)*
  - Il processo figlio eredita dal padre una copia esatta del codice, stack, file descriptor, heap, variabili globali, e program counter
  - Il figlio riceve un nuovo *pid, time, signals, file locks, …*

- fork() restituisce
  - -1in caso di errore
  - 0 al processo figlio
  - il PID del figlio al processo padre

Esempio

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

int main(void) {
    pid_t x;
    x = fork();
    if (x == 0)
        printf("In child: fork() returned %ld\n", (long) x);
    else
       printf("In parent: fork() returned %ld\n", (long) x);
}
```

Creazione di processi

- I processi padre generano i processi figli che a loro volta generano altri processi in questo modo si crea una gerarchia di processi
  - UNIX: process group
- Windows non possiede nozioni di gerarchia di processi
  - Tutti i processi sono uguali

Esempi

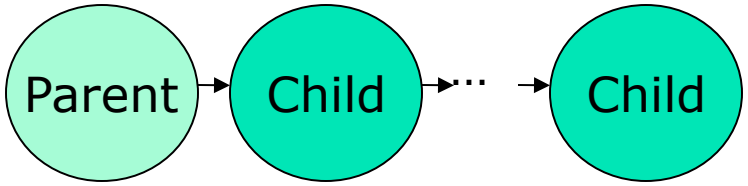
```
pid_t childpid = 0;
for (i=1;i<n;i++)
   if (fork() != 0) kill;
```

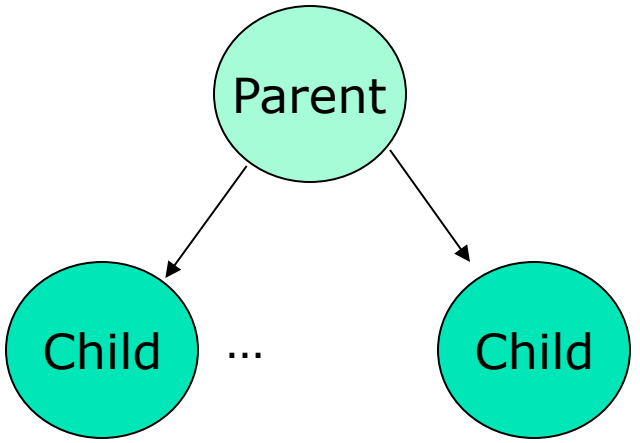
```
pid_t childpid = 0;
for (i=1;i<n;i++)
    if (fork() == 0) kill;
```

Esempi

```
pid_t childpid = 0;
for (i=1;i<n;i++)
  if (fork() != 0) kill;
```

```
pid_t childpid = 0;
for (i=1;i<n;i++)
  if (fork() = 0) kill;
```





Terminazione di un processo

- Normale (volontaria)
  - Al termine della procedura main()
  - exit(0)
- Per errore (volontaria)
  - exit(2) o abort()
- Errore imprevisto (involontaria)
  - Divisione per 0, seg fault, exceeded resources
- Killed (involontaria)
  - Signal: kill(procID)

Operazioni di terminazione

- Quando un processo termina:
  - I file aperti vengono chiusi
  - I file Tmp sono cancellati
  - Le risorse dei processi figli sono deallocate
    - File descriptor, memoria, semafori, ecc.
- Il processo padre viene notificato via signal
- Lo stato di terminazione (Exit status) è disponibile al genitore attraverso la syscall wait()

System call: wait(), waitpid()

- *wait()* il genitore si sospende in attesa che qualche processo figlio termini
- *wait()* il pid e un codice di ritorno sono restituiti al genitore
- *waitpid()* il genitore si mette in attesa della terminazione di un determinato figlio

errno	Cause
ECHILD	Caller has no unwaited-for children
EINTR	Function was interrupted by signal
EINVAL	Options parameter of waitpid was invalid

Esempio

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

pid_t childpid;

childpid = wait(NULL);
if (childpid != -1)
   printf("waited for child with pid %ld\n",
             childpid);
```

What's a Zombie Process?

- When a process dies on Linux, it isn't all removed from memory immediately — its process descriptor stays in memory (the process descriptor only takes a tiny amount of memory).
- The process's status becomes EXIT_ZOMBIE and the process's parent is notified that its child process has died with the SIGCHLD signal.
- The parent process is then supposed to execute the wait() system call to read the dead process's exit status and other information. After wait() is called, the zombie process is completely removed from memory.
- This normally happens very quickly, so you won't see zombie processes accumulating on your system. However, if a parent process isn't programmed properly and never calls wait(), its zombie children will stick around in memory until they're cleaned up.