Sistemi Operativi e Reti di Calcolatori (SOReCa)

Corso di Laurea in *Ingegneria Informatica e Automatica (BIAR)*

Terzo Anno | Primo Semestre

A.A. 2024/2025

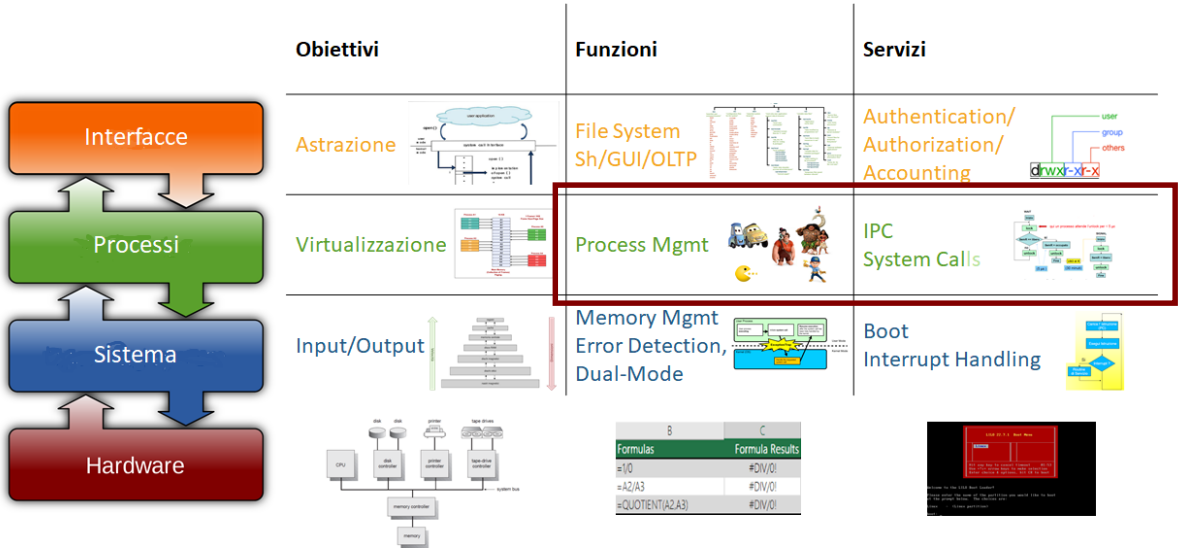
Process, Thread, Concurrency

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Sistemi operativi (3 CFU)

- Il sistema operativo
- Concorrenza e sincronizzazione
- Deadlock
- Inter-process communication (IPC)
- Scheduling
- Memoria centrale e virtuale
- Memoria di massa e File system
- Sicurezza informatica



Lezioni: Settembre - Ottobre

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

I Processi: breve recap

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Processi

Processo: definizione

Processo è un programma in esecuzione:

- Indipendente dagli altri programmi
- Che non deve interferire con gli altri programmi
- Che trae giovamento dalla gestione ottimizzata delle risorse
- Che deve sottostare alla allocazione e condivisione ordinata delle risorse rispetto a spazio/tempo

«**Legum servi** sumus ut **liberi** esse possimus»
[M. T. Cicerone]

Controllo ferreo sull'accesso alle risorse.

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Processi

Processo != Programma

Processo = Esecuzione delle Istruzioni ➔ Entità attiva

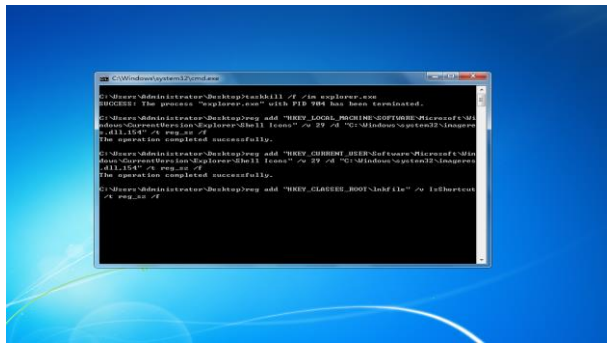
- Contatore di programma
- Valori delle variabili
- Risorse in uso

```
howtogeek@ubuntu: ~
top - 22:47:18 up 33 min,  2 users,  load average: 3.61, 1.51, 0.86
Tasks: 201 total,   7 running, 157 sleeping,   0 stopped,  37 zombie
Cpu(s): 13.5%us,  7.7%sy,  0.0%ni, 78.8%id,  0.0%wa,  0.0%hi,  0.0%si,
Mem:   1024792k total,   940672k used,    84120k free,    84300k buffer
Swap:  1046524k total,     2388k used,  1044136k free,   428356k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1191 root      20   0  106m  70m  11m S 11.2  7.0   0:38.93 Xorg
 2349 howtogee  20   0 72636  12m  10m R  3.7  1.3   0:02.59 metacity
 2381 howtogee  20   0 50276 9564 7492 S  3.7  0.9   0:03.69 bamfdaemon
 2959 howtogee  20   0 90892  24m  18m R  3.7  2.4   2:23.76 gnome-syst
 3551 howtogee  20   0 91544  15m  11m R  1.9  1.6   0:01.42 gnome-term
 3772 howtogee  20   0  2836 1184  880 R  1.9  0.1   0:00.15 top
    1 root      20   0  3628 1708 1348 S  0.0  0.2   0:02.76 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.17 ksoftirqd/
    5 root      20   0     0    0    0 S  0.0  0.0   0:01.11 kworker/u:
```

Programma = Lista di Istruzioni ➔ Entità passiva

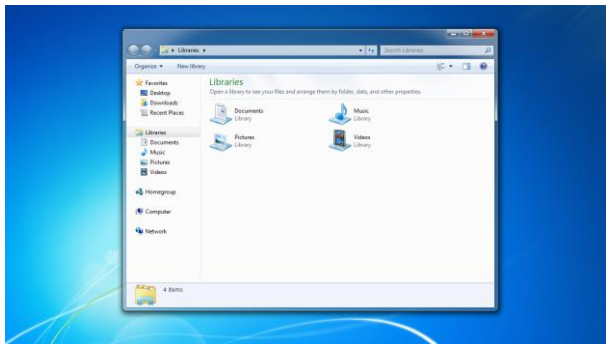
```
def add5(x):
    return x+5

def dotwrite(ast):
    nodename = getNodename()
    label=symbol.sym_name.get(int(ast[0]),ast[0])
    print '    %s [label="%s"' % (nodename, label)
    if isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s"];' % ast[1]
        else:
            print '"]'
    else:
        print '"];'
        children = []
        for n, child in enumerate(ast[1:]):
            children.append(dotwrite(child))
        print '    %s -> {' % nodename,
        for name in children:
            print '%s' % name,
```

due **processi** associati allo stesso programma, sono due differenti istanze di esecuzione dello stesso codice:



0 explorer.exe

1° explorer.exe



N-simo explorer.exe

Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti

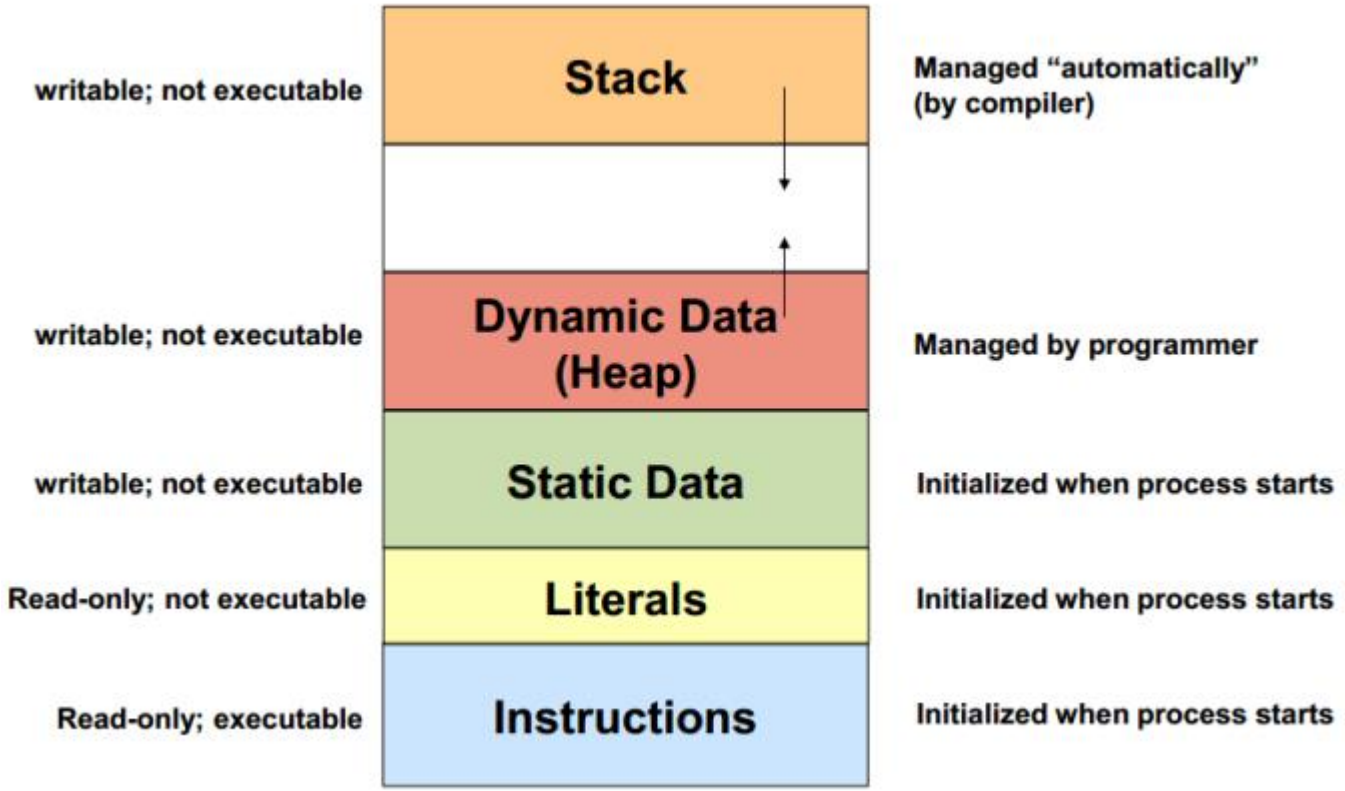
Sapienza
Università di Roma

Operating Systems: Processi

Memory Layout 1/3

La memoria occupata dal processo è composta di:

- **Instructions**: codice del programma (puntato dal Program Counter: **PC**)
- **Literals**: parametri del programma
- **Static Data**: variabili globali in memoria centrale (puntato dallo Stack Pointer: **SP**)
- **Heap** (mucchio): dati dinamici
- **Stack** (pila): funzioni (metodi), variabili locali, indirizzi di rientro

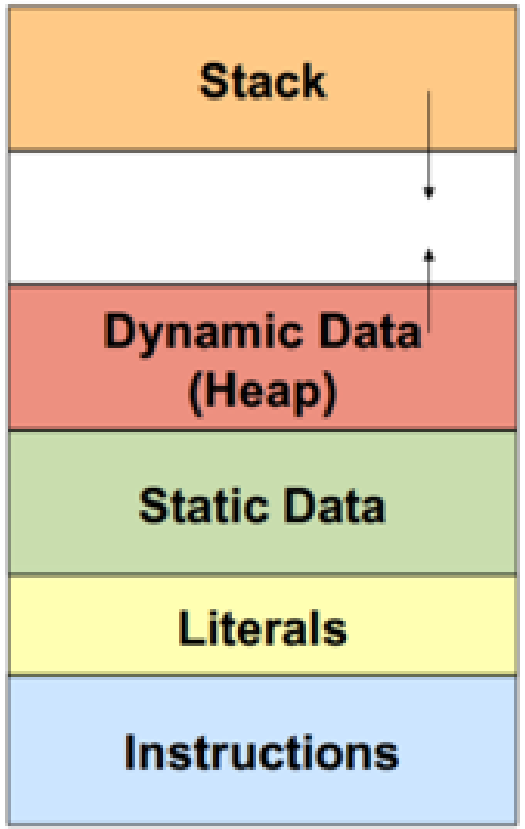


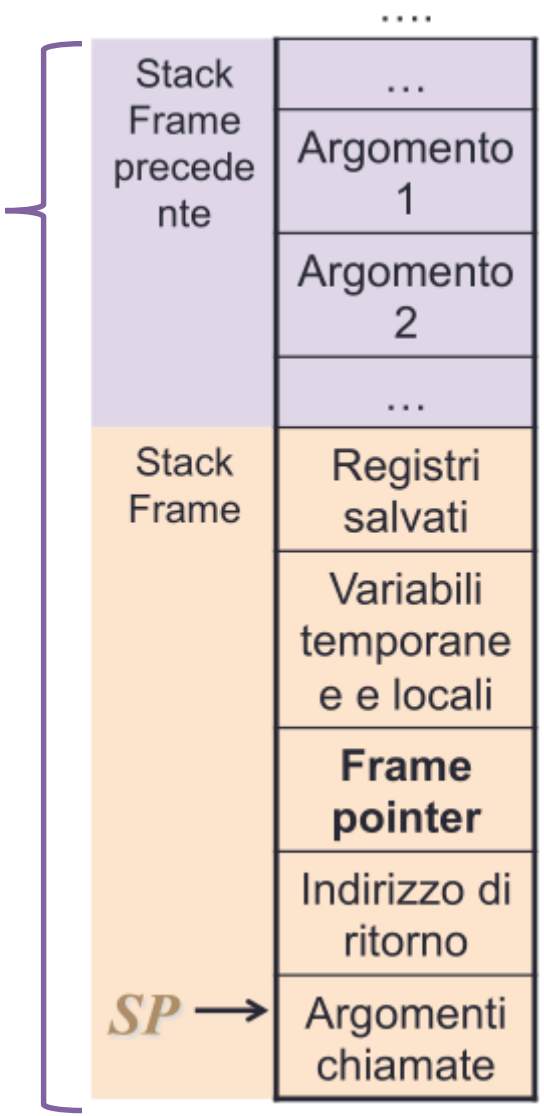
Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Processi

Memory Layout 2/3





Lo **stack** è una pila di dati che tipicamente cresce verso il basso

- I record di attivazione delle funzioni (stack frame) vengono inseriti (pushed) e rimossi (popped) dallo stack
- Lo stack frame contiene dati relativi ad una funzione:
  - Parametri
  - Variabili locali
  - Dati necessari per ripristinare il frame precedente
- Viene puntato dal Frame Pointer (**FP**)

Lo **heap** è una pila di dati che tipicamente cresce verso l'altro

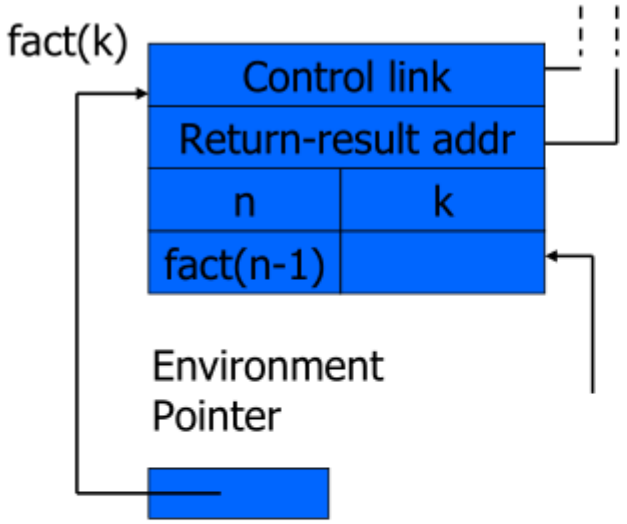
- Il programmatore si occupa di come utilizzare lo heap (non il SO)
- Si deve occupare di allocare (malloc C) e deallocare la memoria
- In Java questo è trasparente al programmatore Garbage Collector)

Se **stack** e **heap** collidono si verificano **errori**: Esistono chiamate di sistema per **ridimensionarli**

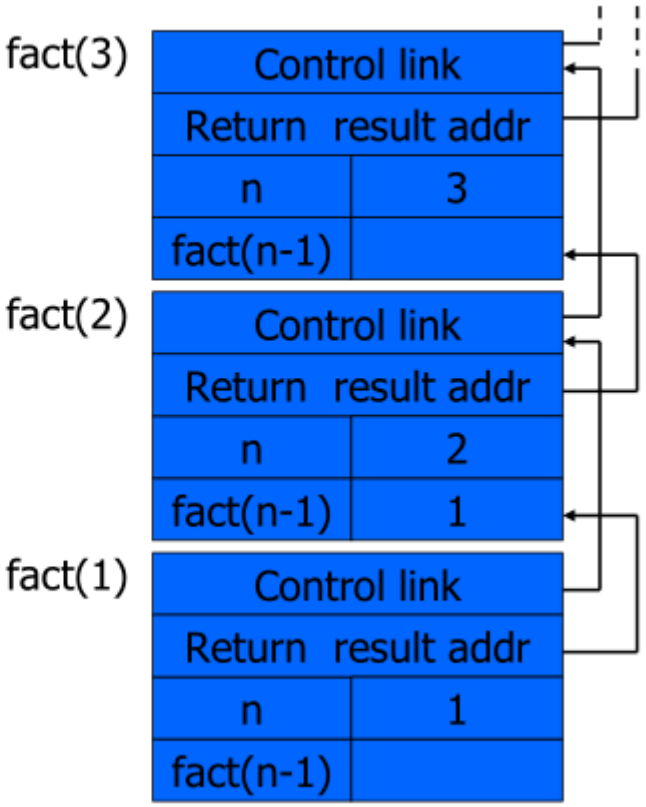
Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

Operating Systems: Processi

Memory Layout 3/3



fact(n) = if n<= 1 then 1
else n * fact(n-1)



- Classico esempio di esecuzione delle funzione fattoriale, implementata in modo iterativo.
- La stessa funzione viene chiamata tante volte (con parameti sempre diversi) finché non si esce dalla condizione di iterazione.

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

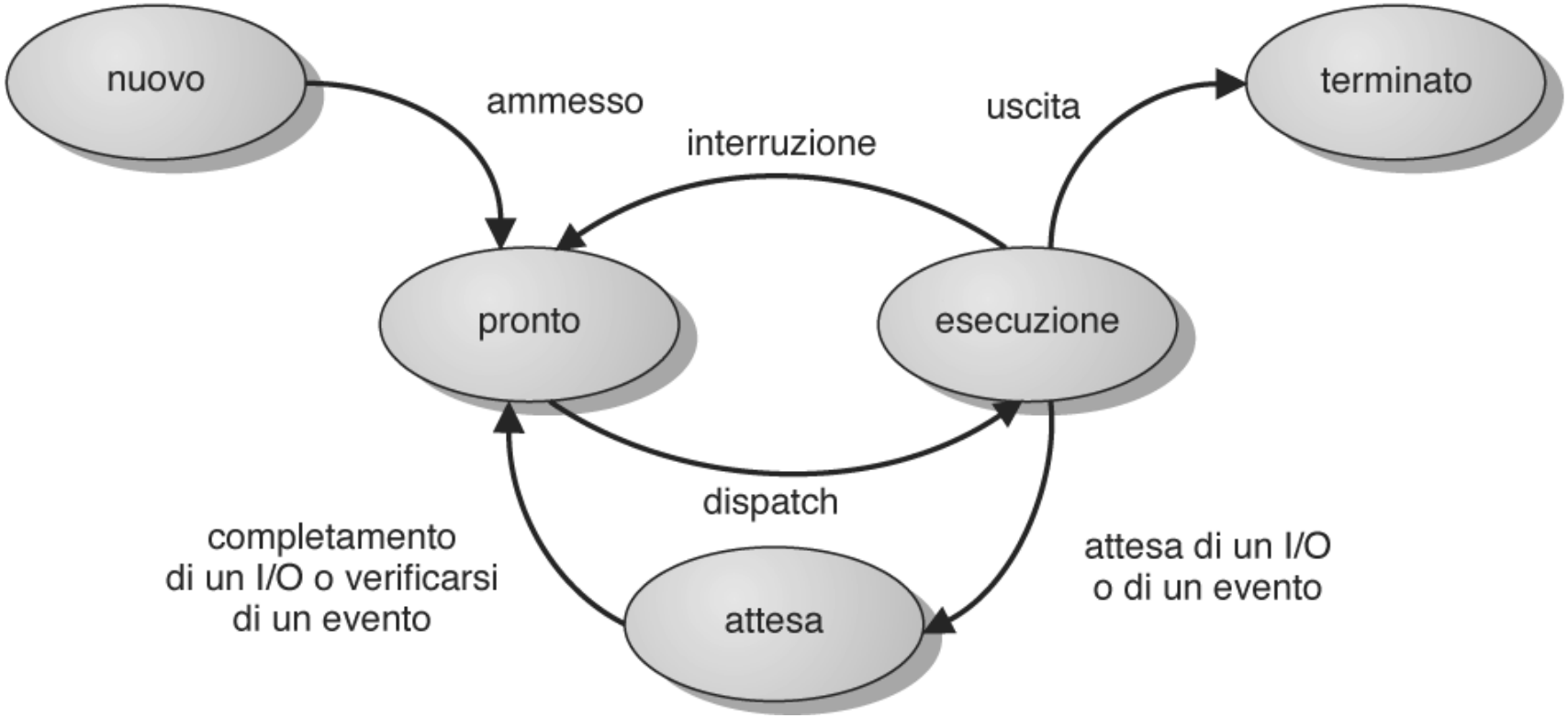
Operating Systems: Processi

Stato

E’ lo stato di uso del processore da parte di un processo

Possibili stati:

- **Nuovo** (new): Il processo è stato creato
- In **esecuzione** (running): le istruzioni vengono eseguite
- In **attesa** (waiting): il processo sta aspettando il verificarsi di qualche evento
- **Pronto** all’esecuzione (ready): il processo è in attesa di essere assegnato ad un processore
- **Terminato** (terminated): il processo ha terminato l’esecuzione

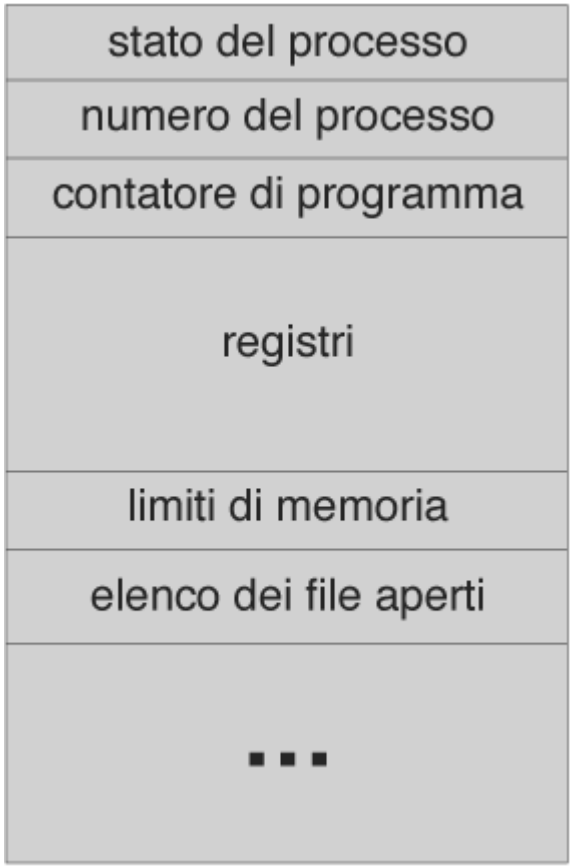


Operating Systems: Processi

PCB

Process Control Block (PCB): Struttura dati del **kernel** che mantiene le informazioni sul processo

- Stato del processo
- Identificatore del processo (Numero)
- Program counter
- i.e. indirizzo istruzione successiva
- Registri della CPU
- Stack Pointer
- Frame Pointer
- I valori devono essere salvati per poter riprendere correttamente l'esecuzione dopo un interruzione
- Informazioni sullo scheduling (es. priorità, etc …)
- Informazioni sulla gestione della memoria (es. tabelle delle pagine, etc …)
- Informazioni di contabilizzazione delle risorse (es. tempo uso CPU, etc …)
- Informazioni sullo stato dell'I/O (es. lista dispositivi assegnati al processo, elenco file aperti, etc …)



Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Processi

PCB: esempio Linux

Process Control Block (PCB): Struttura dati del **kernel** che mantiene le informazioni sul processo

```
struct task_struct {
        long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
        long counter;
        long priority;
        long signal;
        fn_ptr sig_restorer;
        fn_ptr sig_fn[32];
/* various fields */
        int exit_code;
        unsigned long end_code,end_data,brk,start_stack;
        long pid,father,pgrp,session,leader;
        unsigned short uid,euid,suid;
        unsigned short gid,egid,sgid;
        long alarm;
        long utime,stime,cutime,cstime,start_time;
        unsigned short used_math;
/* file system info */
        int tty;                /* -1 if no tty, so it must be signed */
        unsigned short umask;
        struct m_inode * pwd;
        struct m_inode * root;
        unsigned long close_on_exec;
        struct file * filp[NR_OPEN];
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
        struct desc_struct ldt[3];
/* tss for this task */
        struct tss_struct tss;
};
```

Operating Systems: Processi

Scheduling (semplificato): stato pronto ed attesa

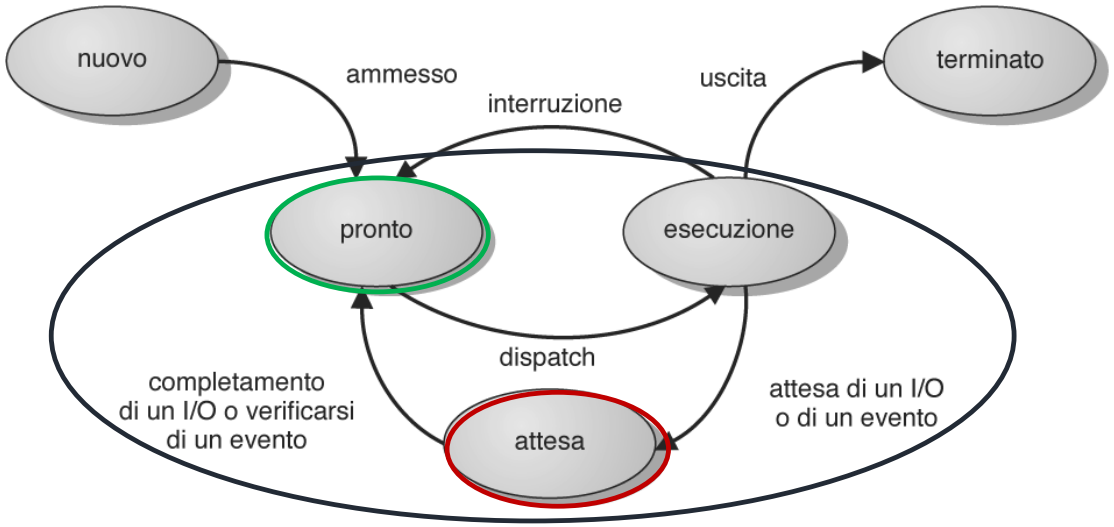
Code di schedulazione dei processi

- Coda di lavori (**job queue**) – contiene tutti i processi nel Sistema

- Coda dei processi pronti (**ready queue**) – contiene tutti i processi che risiedono nella memoria centrale, pronti e in attesa di esecuzione

- Coda della periferica di I/O (**device queues**) – contiene i processi in attesa di una particolare periferica di I/O (una per ogni dispositivo)

- Il processo si muove fra le varie code



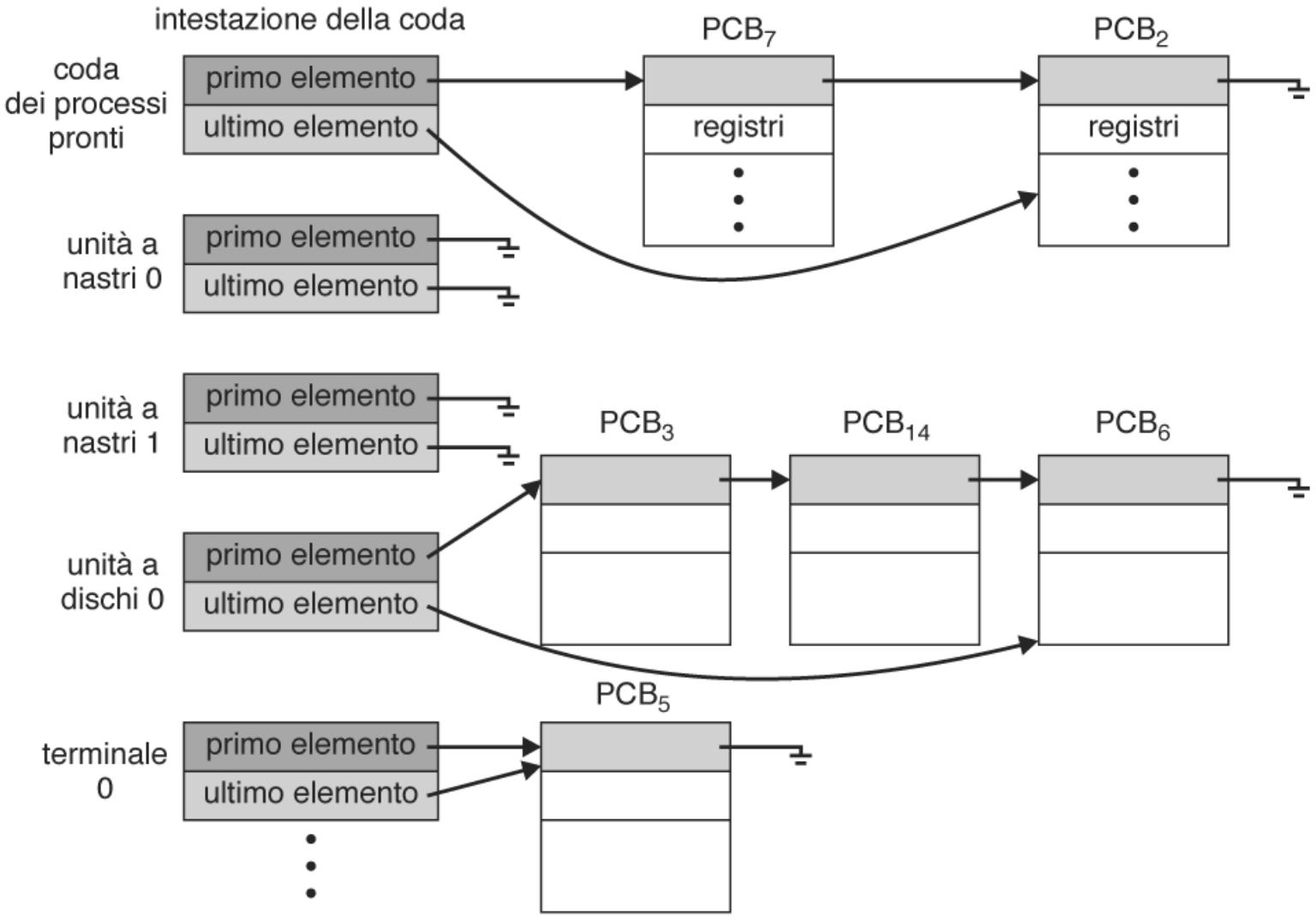
DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Processi

Scheduling

Code di schedulazione dei process nei vari stati



Ogni coda ha due puntatori:

- Primo PCB
- Ultimo PCB

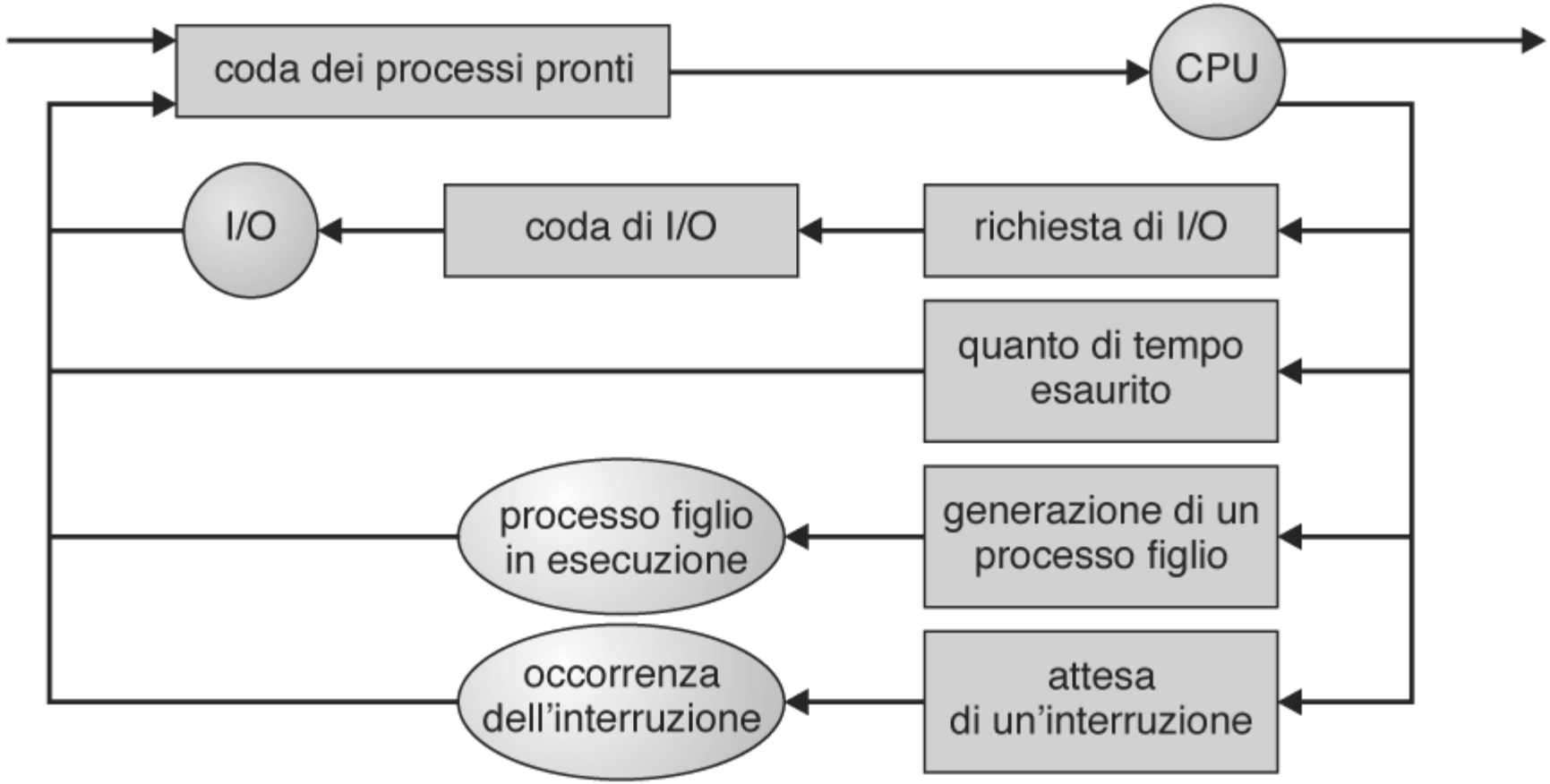
Ogni PCB ha un puntatore al successivo

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Processi

Scheduling: diagramma di accodamento



- Richiesta I/O: il processo torna nella coda pronti al termine dell'I/O
- Interruzione: il processo viene rimesso nella coda dei pronti
- Generazione figlio: il processo torna nella coda dei pronti alla terminazione del figlio
- Quando un processo termina il PCB e le sue risorse sono deallocate

I processi possono essere classificati come:

- processo **I/O-bound** – processo che spende più tempo facendo I/O che elaborazione (molti e brevi utilizzi di CPU)
- processo **CPU-bound** – processo che spende più tempo facendo elaborazione che I/O (pochi e lunghi utilizzi di CPU)

Bilanciamento **I/O-bound** – **CPU-bound** ➔ Buone prestazioni

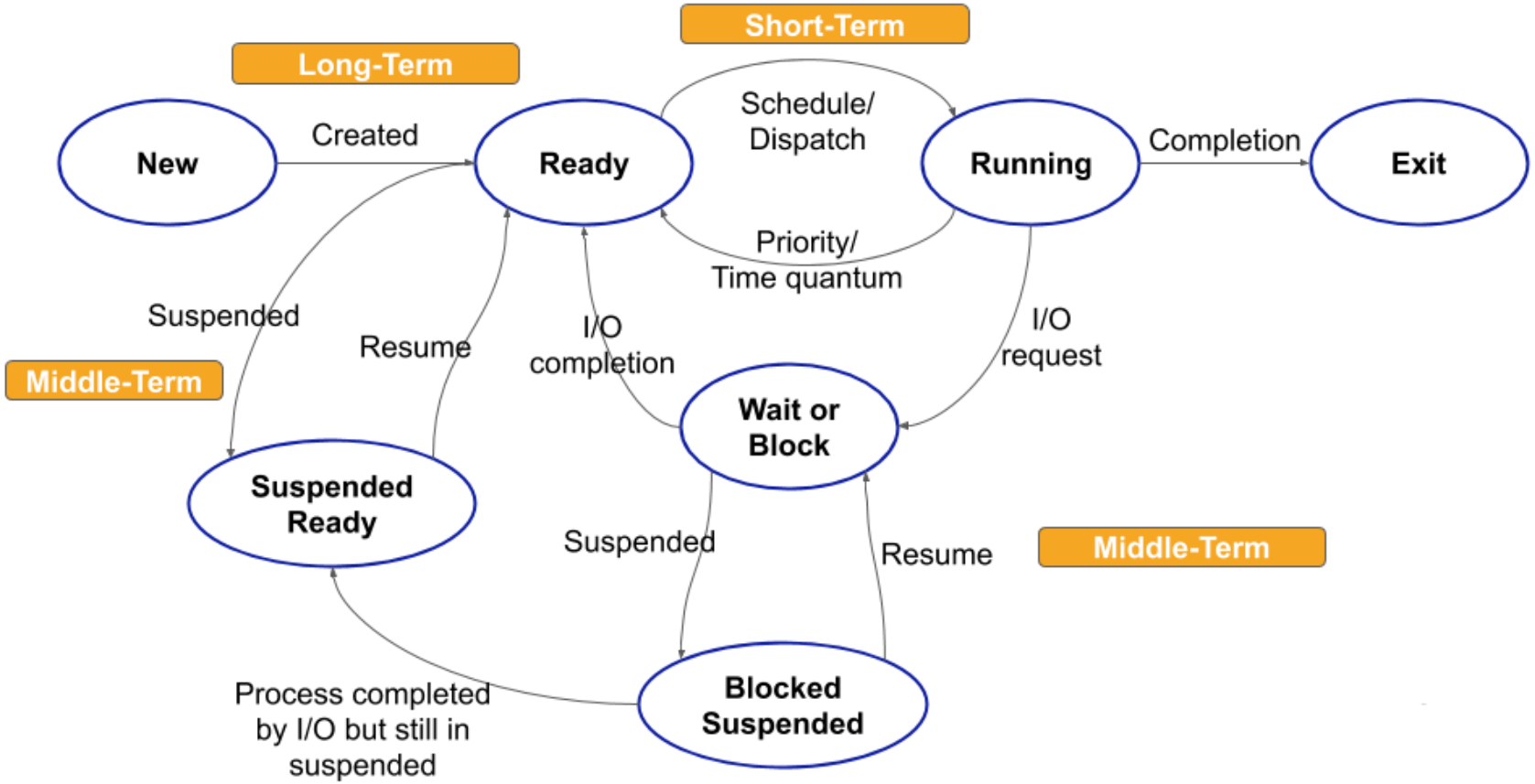
Sbilanciamento **I/O-bound** – **CPU-bound** ➔ Coda "Pronto" o coda "Attesa" vuote

Operating Systems: Processi

Schedulatori

- **Short-Term** (Schedulatore a breve termine): seleziona quale processo (in memoria) deve essere eseguito e alloca la CPU ad esso ogni ms (jiffies)

- **Middle-Term** (Schedulatore a medio termine): esegue lo swapping
  - Rimuove un processo dalla memoria centrale e lo pone in memoria di massa
  - Supportato solo da alcuni SO come i sistemi time-sharing



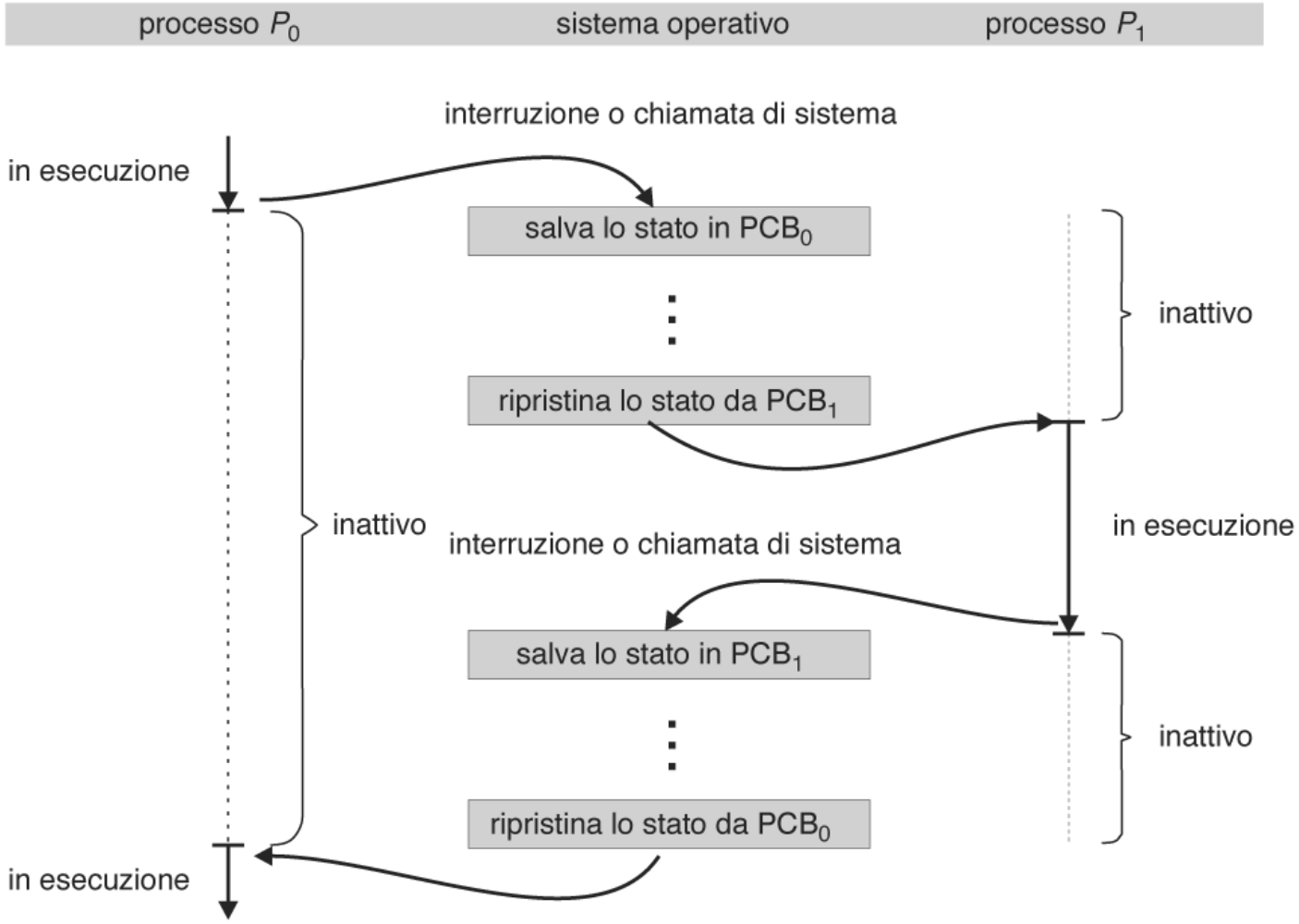
- **Long-Term** (Schedulatore a lungo termine): seleziona quale processo (attualmente in memoria di massa) deve essere inserito nella coda dei processi pronti, con frequenza nell'ordine dei secondi/minuti:
  - Controlla il grado di multi-programmazione
  - Stabile se velocità di creazione processi = velocita terminazione processi
  - Richiamato solo quando un processo abbandona il sistema (termina)
  - Assente nei sistemi UNIX e Windows

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Processi

Context Switch



Context Switch (Cambio di Contesto): passaggio della CPU da un processo ad un altro

Context Switch =
sospensione del processo in esecuzione
+ caricamento del nuovo processo da mettere in esecuzione

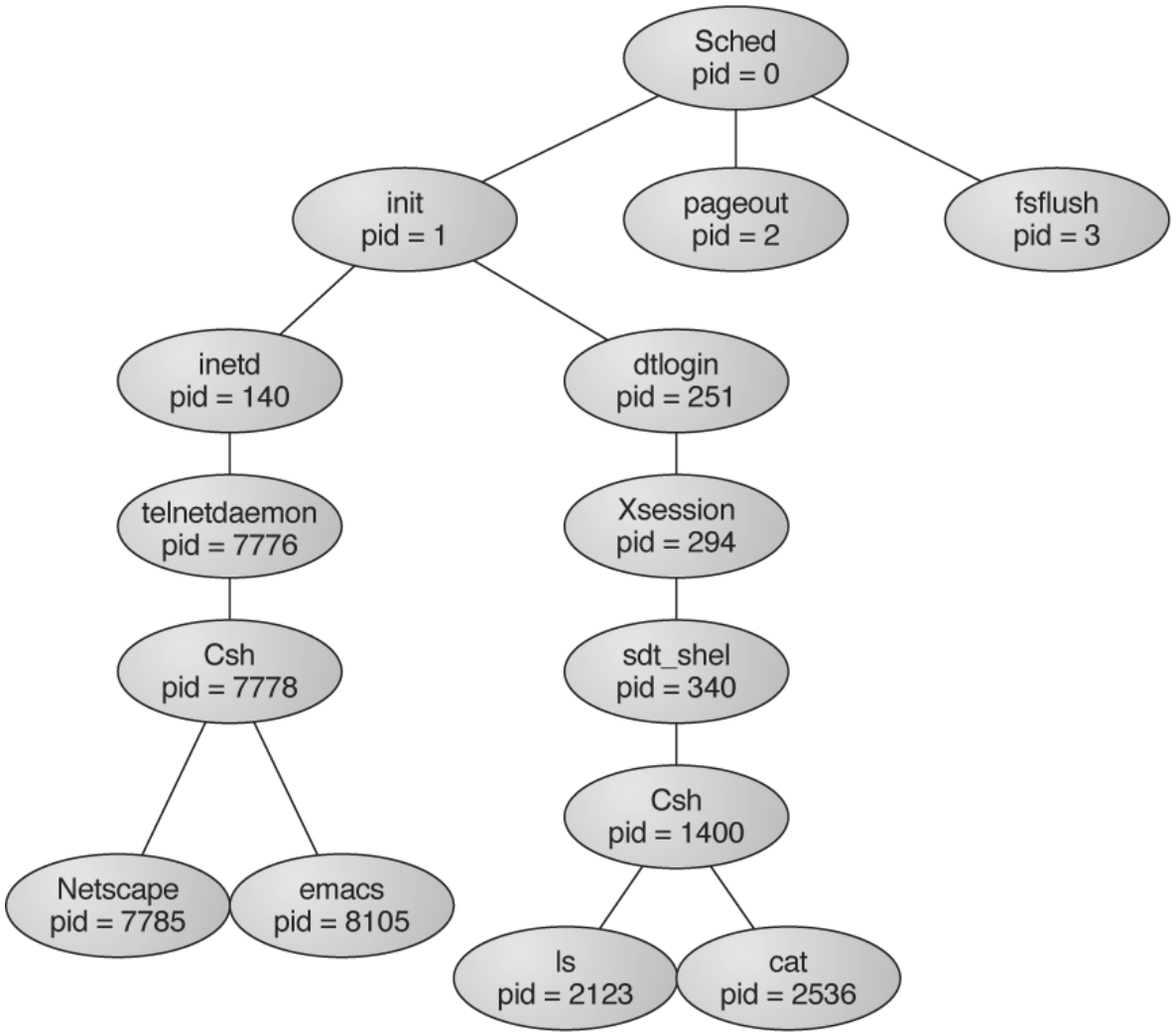
- Il tempo per il cambio di contesto è puro tempo di gestione del sistema (non vengono compiute operazioni utili per la computazione dei processi)
- I tempi (10ms) per i cambi di contesto dipendono dal supporto hardware (e.g. registri disponibili e/o registri dedicati). <

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Processi

Nuovo Processo: Creazione (Generazione) 1/2



Un processo in esecuzione può creare numerosi sottoprocessi usando un'apposita chiamata di sistema (create_process):

- Processo generante = processo padre
- Processo generato = processo figlio

Ogni processo ha un identificatore univoco PID (intero)

In unix si usa la chiamata fork(): nuovo processo

➔ albero di processi (visibile ad es. tramite ps -el)

➔ Inizialmente, il figlio è la copia del padre (es. codice binario, spazio degli indirizzi

➔ Ritorna un valore

- ➔ 0: figlio
- ➔ <>0: padre
- ➔ Exec() sovrascrive lo spazio degli indirizzi del chiamante caricando un nuovo programma

Esecuzione del processo padre dopo la creazione del figlio:

- Continua in modo concorrente, oppure
- Attende la terminazione del figlio ➔ wait()

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI
SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: Processi

Nuovo Processo: Creazione (Generazione) 2/2

- La chiamata fork() crea un nuovo processo. Il valore di ritorno vale
  - 0 per il figlio ➔ 1° else
  - Diverso da 0 per il padre ➔ 2° else

La fork crea una copia dello spazio degli indirizzi del genitore
Entrambi, padre e figlio, continuano l'esecuzione dalla chiamata successiva alla fork

La chiamata wait(NULL) blocca il chiamante fino alla terminazione di un figlio (il primo che termina)

waitpid(…, int pid, …) permette di attendere la terminazione di uno specific figlio

```
int main()
{
  pid_t pid;

  // fork a child process
  pid = fork();

  if (pid < 0) { // error occurred
      fprintf(stderr, "Fork Failed\n");
      exit(-1);
  }
  else if (pid == 0) { // child process
      printf("I am the child %d\n",pid);
      execlp("/bin/ls","ls",NULL);
  }
  else { // parent process
    // parent will wait for the child to
    complete
      printf("I am the parent %d\n",pid);
      wait(NULL);

      printf("Child Complete\n");
      exit(0);
  }
}
```

Operating Systems: Processi

Terminazione del Processo: Normale ed Anomala

Terminazione normale dopo l'ultima istruzione tramite la chiamata exit:

- "figlio" può restituire un valore di stato (di solito un intero) al "padre". Nell'es. tramite la chiamata wait()
- le risorse del processo sono deallocate dal SO
  - File aperti
  - Memoria fisica e virtuale
  - Aree di memoria per I/O

Terminazione in caso di anomalia (aborto)

- Il padre può terminare l'esecuzione di uno dei suoi figli per varie ragioni:
- Eccessivo uso di una risorsa
- Compito non più necessario

Terminazione a cascata:

- se il padre sta terminando, alcuni SO non permettono ad un processo figlio di proseguire
- In unix la terminazione a cascata non esiste: I processi figli vengono "adottati" dal processo init

Mancata terminazione tramite la chiamata wait(), porta ad un processo "zombie": la cui elaborazine è terminate ma cui il padre non ha dato il consenso per la terminazione.

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Unix SysCall: `fork()`, `wait()`, `exit()`

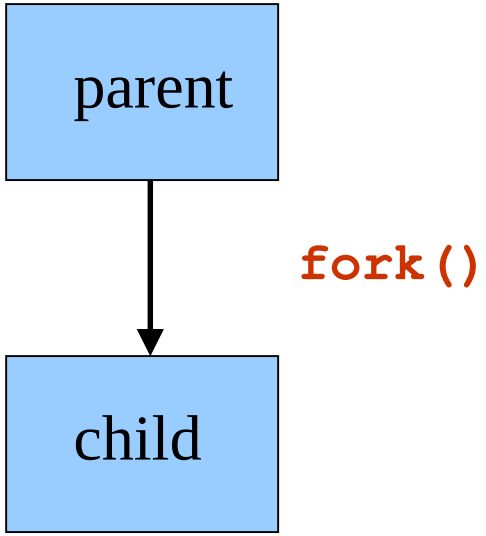
Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Unix SysCall

System Calls: How To Create New Processes

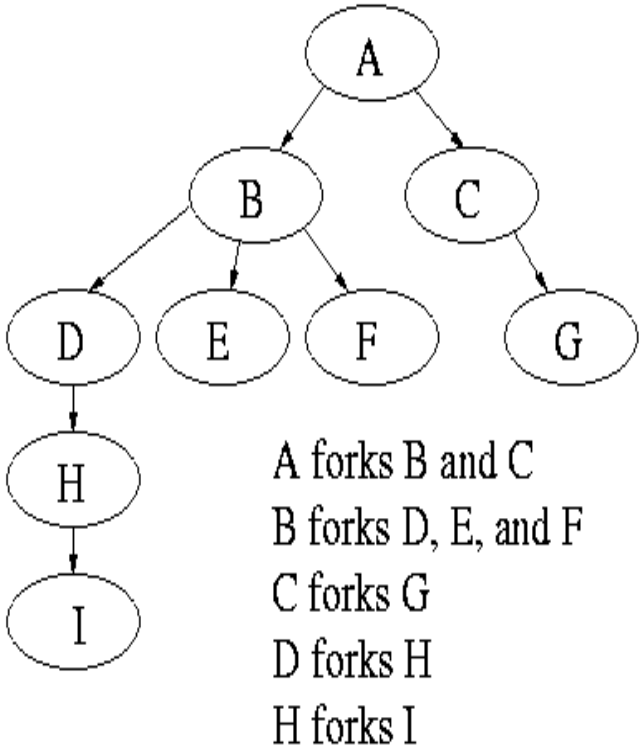
- Underlying mechanism
  - A process runs fork to create a child process
  - Parent and children execute concurrently
  - Child process is a duplicate of the parent process



Operating Systems: Unix SysCall

System Calls: Processes Creation

- After a **fork**, both parent and child keep running, and each can fork off other processes.
- A process tree results. The root of the tree is a special process created by the OS during startup.

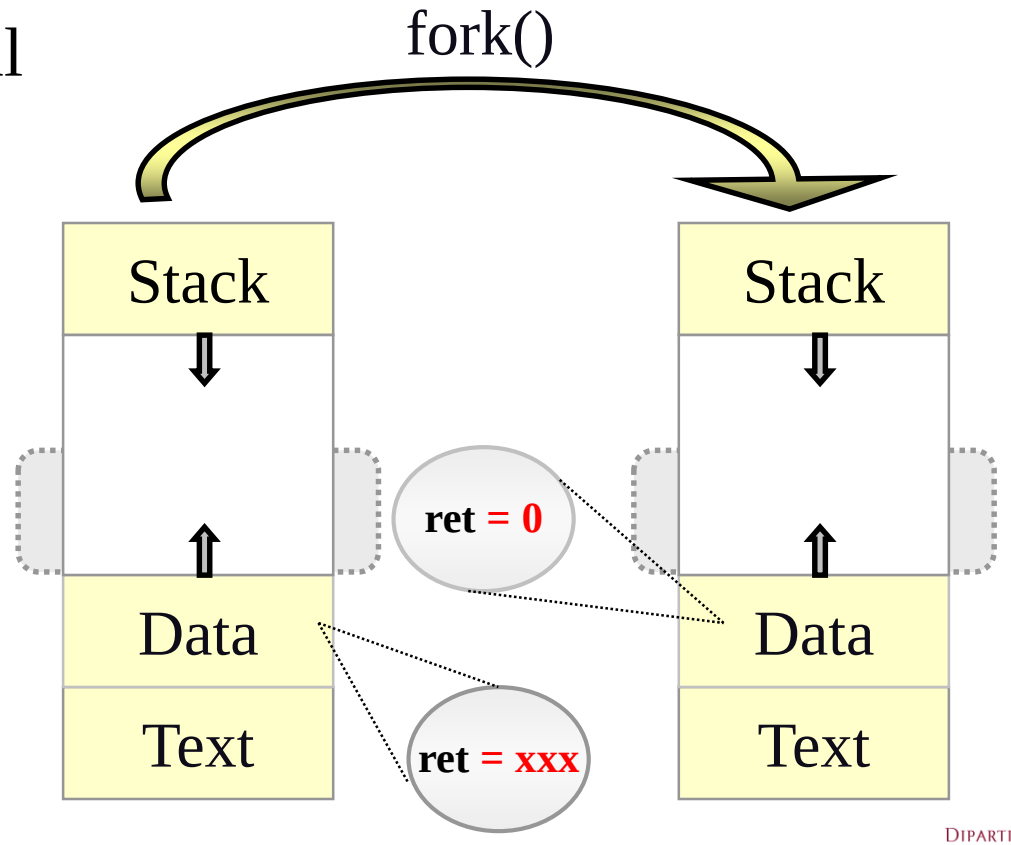


- A process can *choose* to wait for children to terminate. For example, if C issued a **wait**() system call, it would block until G finished.

Operating Systems: Unix SysCall

System Calls: fork() 1/2

- Current process split into 2 processes: parent, child
  - Returns -1 if unsuccessful
  - Returns 0 in the child
  - Returns the child's identifier in the parent



Operating Systems: Unix SysCall

System Calls: fork() 2/2

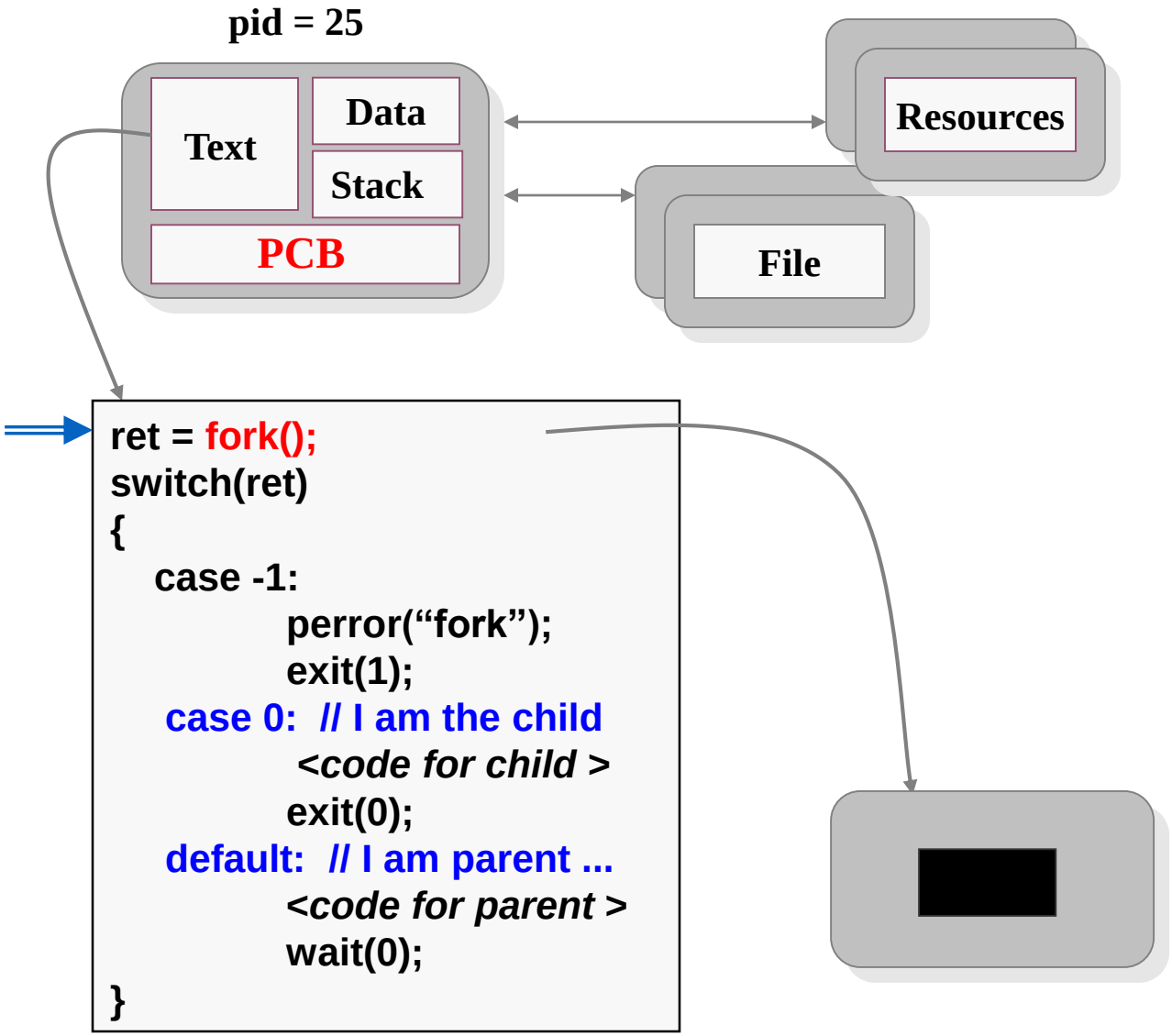
- The child process inherits from parent
  - identical copy of memory
  - CPU registers
  - all files that have been opened by the parent
- Execution proceeds concurrently with the instruction following the fork system call
- The execution context (PCB) for the child process is a copy of the parent's context at the time of the call

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

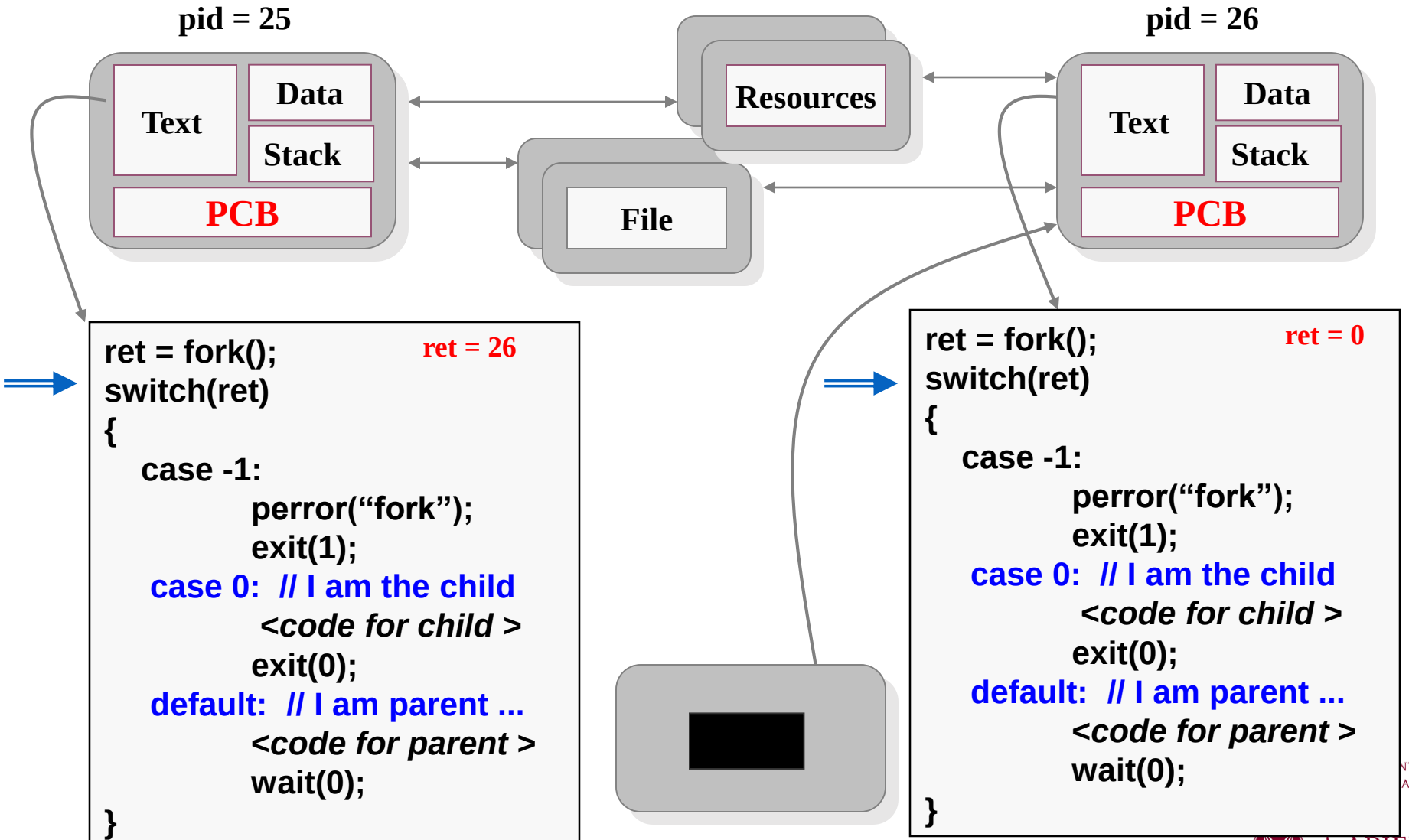
Operating Systems: Unix SysCall

System Calls: how fork() work 1/6



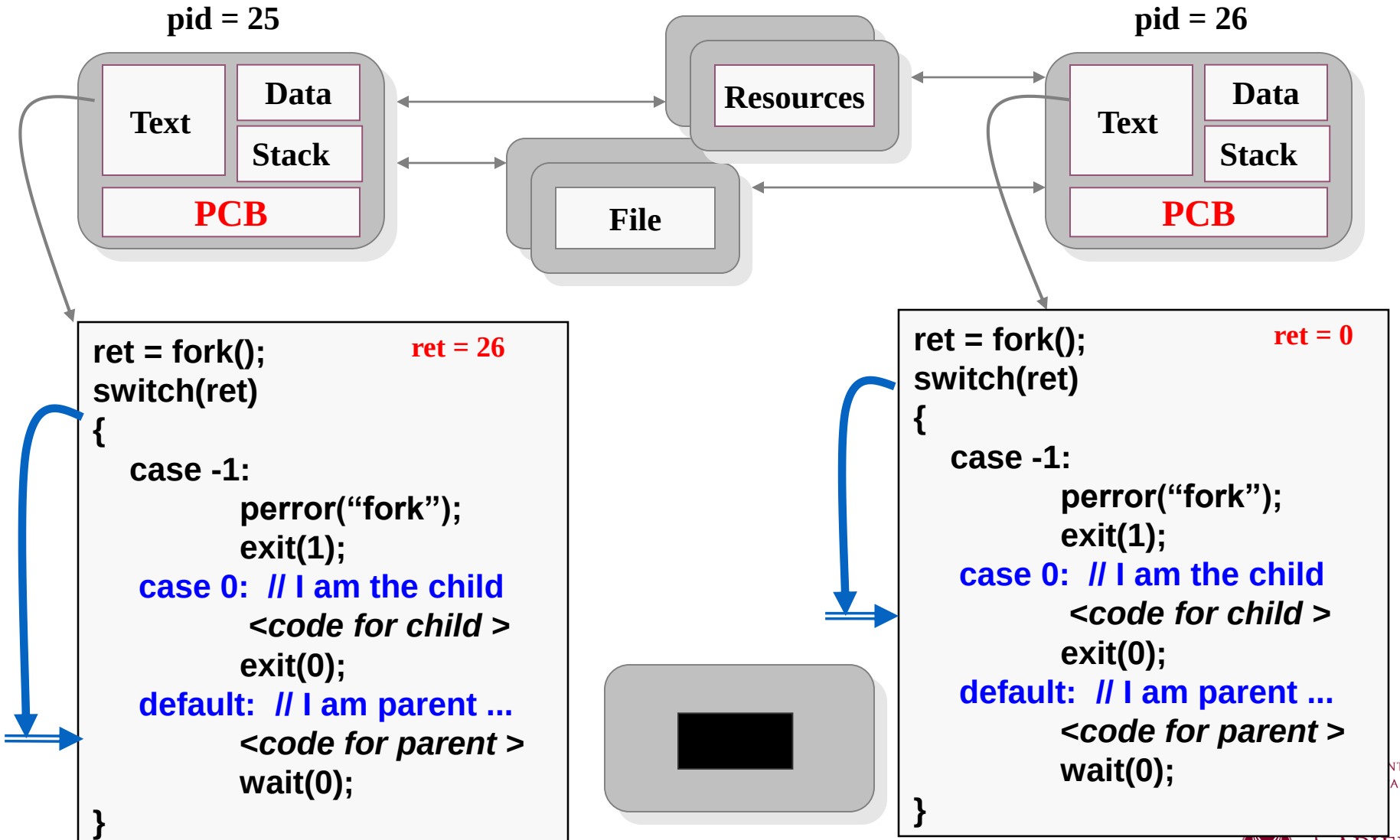
Operating Systems: Unix SysCall

System Calls: how fork() work 2/6



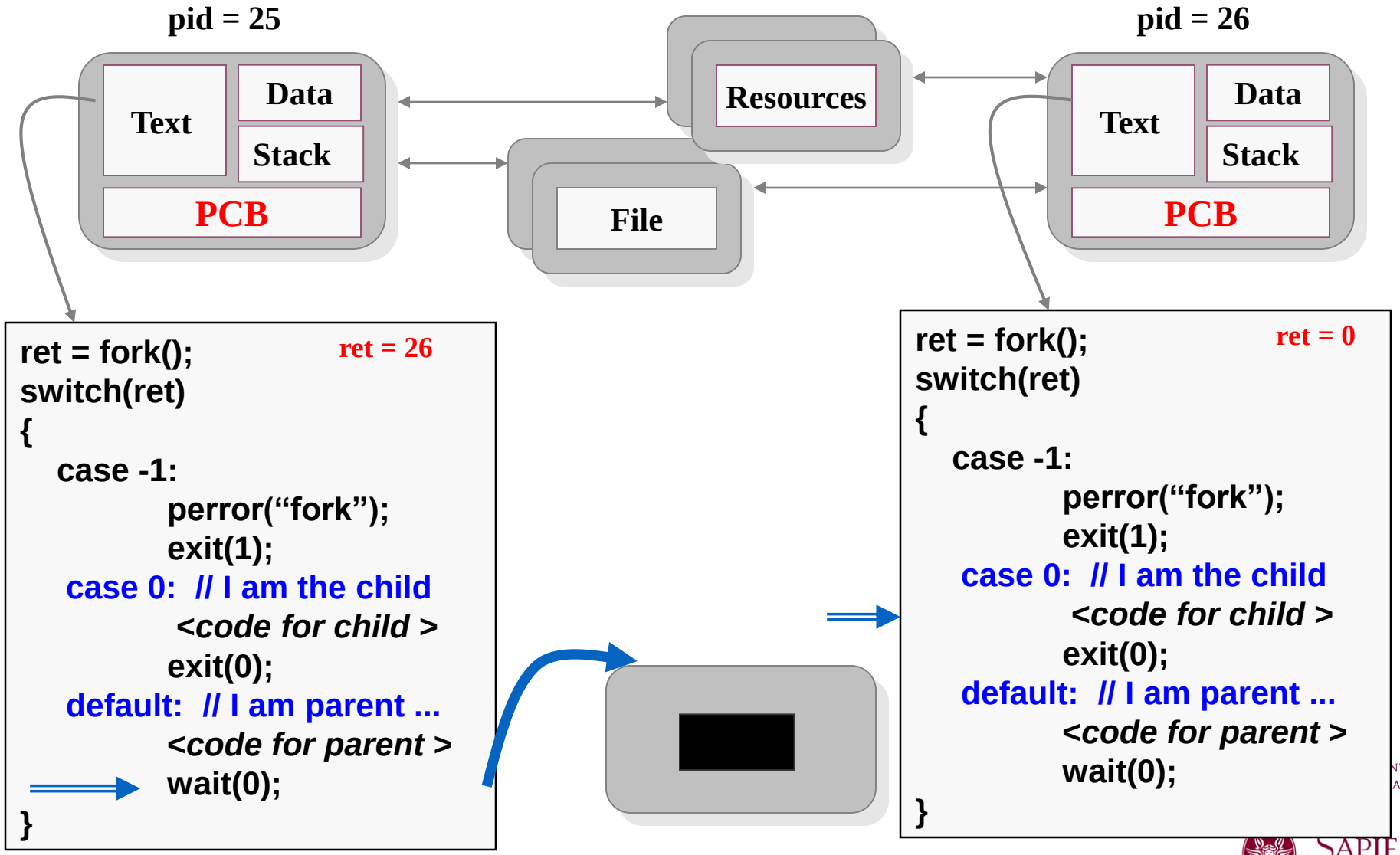
Operating Systems: Unix SysCall

System Calls: how fork() work 3/6



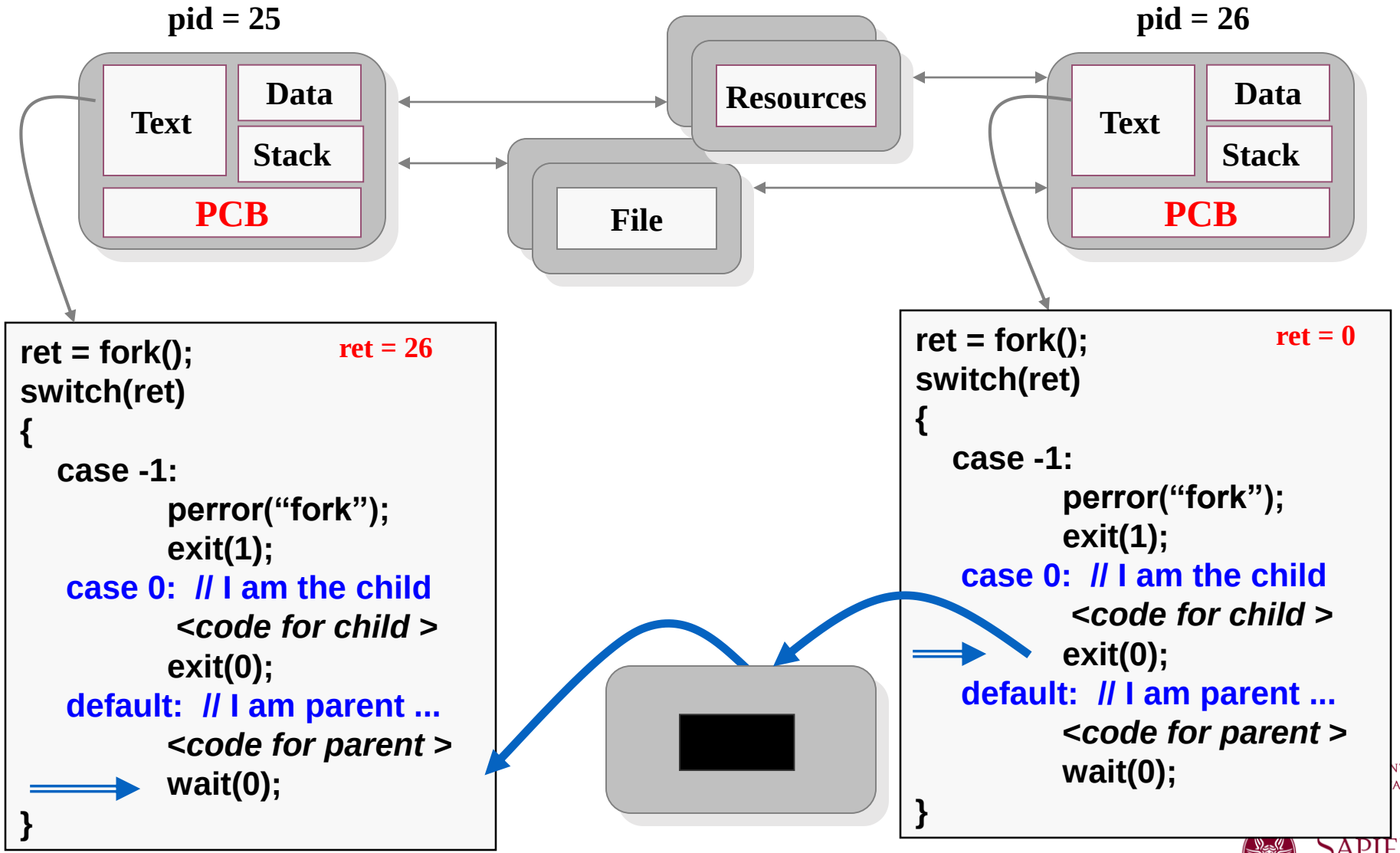
Operating Systems: Unix SysCall

System Calls: how fork() work 4/6



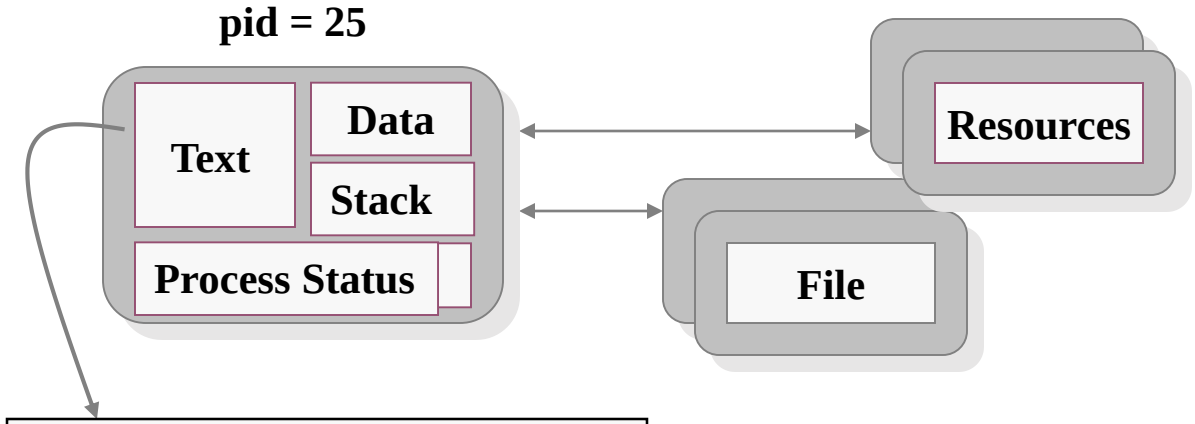
Operating Systems: Unix SysCall

System Calls: how fork() work 5/6

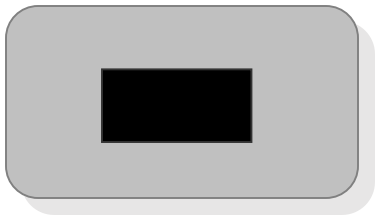


Operating Systems: Unix SysCall

System Calls: how fork() work 6/6




```
ret = fork();                    ret = 26
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
    case 0:  // I am the child
             <code for child >
            exit(0);
    default:  // I am parent ...
            <code for parent >
            wait(0);
            < … >
```



Operating Systems: Unix SysCall

System Calls: exit(), Orderly Termination

- To finish execution, a child may call **exit**(*status*)

- This system call:
  - Saves result = argument of exit
  - Executes all functions specified with **atexit**(*fun*) and **on_exit**(*fun*)
  - Streams are downloaded with **fflush()**
  - Closes all open files, connections
    - (not the ones shared with other processes)

  - Call **_exit**(*status*)

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Unix SysCall

System Calls: _exit(), Finishing Execution

- To finish execution, a child may call **_exit**(*status*)

- This system call:
  - Saves result = argument of exit
  - Deallocates memory
  - If the process has running childs, they are assigned to init
  - Checks if parent is alive
  - If parent is alive, holds the result value until the parent requests it (with wait); in this case, the child process does not really die, but it enters a zombie/defunct state
  - If parent is not alive, the child terminates (dies)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Unix SysCall

System Calls: wait(), waiting Child Finishing

- Parent may want to wait for children to finish
  - Example: a shell waiting for operations to complete
- Waiting for any some child to terminate: **wait**()
  - Blocks until some child terminates
  - Returns the process ID of the child process
  - Or returns -1 if no children exist (i.e., already exited)
- Waiting for a specific child to terminate: **waitpid**()
  - Blocks till a child with particular process ID terminates

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

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

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

I Thread: breve recap

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Processes and Threads

Scheduling and Resources

A process has two characteristics:

- **Scheduling/execution** - follows an execution path that may be interleaved with other processes
- **Resource ownership** - includes a virtual address space to hold the process image

These two characteristics are treated independently by the operating system.

- The unit of dispatching is referred to as a ***thread*** or lightweight process
- The unit of resource ownership is referred to as a process or ***task***

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Thread

Obiettivo

Ogni processo tradizionale (mono-thread) esegue un'attività alla volta ➔ molte attività necessitano di molti processi: `fork()`

Ogni `fork()` implica risorse aggiuntive:

- Risorse di memoria duplicate: Area istruzioni e variabili globali, etc
- Scheduling addizionale: creazione nuovo processo e strutture (PCB)
- Context Switch addizionale: aumento del consumo di CPU senza elaborazioni fattuali

Πηνελόπεια

➔ **condivisione delle risorse** (spazio: memoria e tempo: elaborazione) fra attività analoghe

Operating Systems: Thread

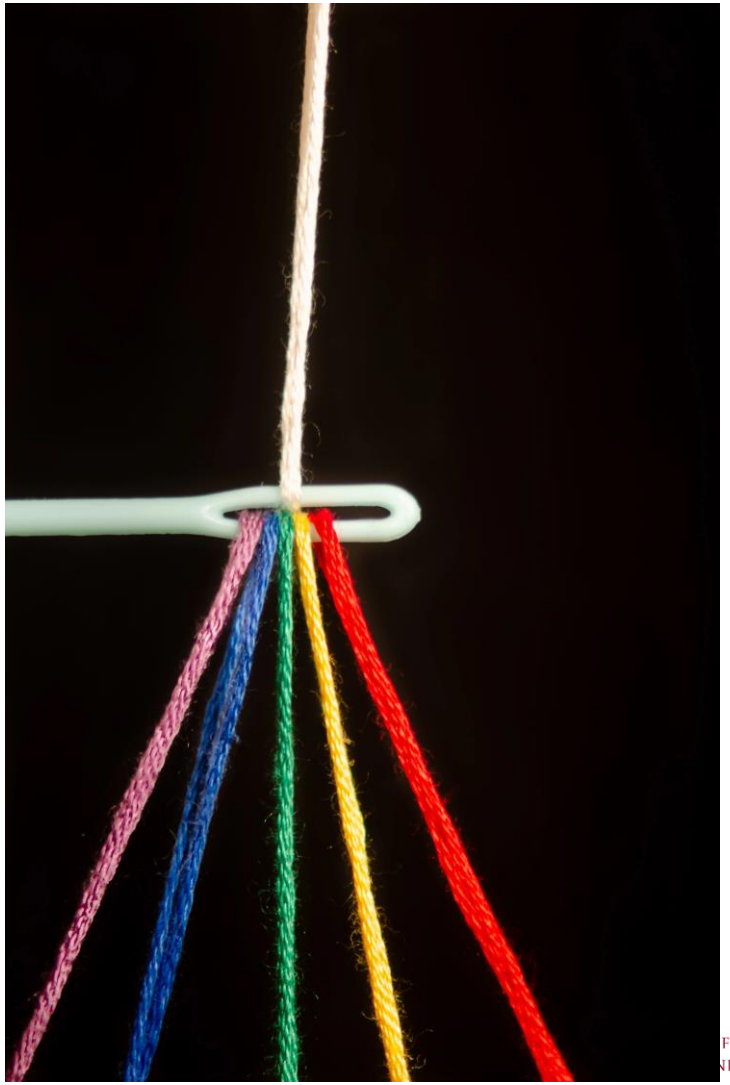
Definizione

Thread: flusso di esecuzione (filo logico) indipendente all'interno dello stesso processo.
Lightweight Process (contesto alleggerito, rispetto ad un processo):

- **Spazio di Indirizzamento**: condiviso con gli altri thread del processo
- **Scheduling**: tramite la struttura di Rappresentazione Thread Control Block (**TCB**) che punta al **PCB** del processo contenitore

Condivisione/Riuso delle risorse.

IBM ha introdotto il supporto per il **multi-threading** (indicati come "sub-tasks") nella variante del **OS/360** denominata Multiprogramming with a Variable number of Tasks (**MVT**), disponibile al pubblico dal **1967** (Terza Generazione, cfr. https://www.britannica.com/technology/IBM-OS-360).
Unix: standard POSIX.1c, Thread (IEEE Std 1003.1c-**1995**).
Es. **Solaris** 2.2 (May **1993**), **Linux** kernel 2.0 (Jun **1996**).



Operating Systems: Multi-Thread

Definizione

The ability of an OS to support multiple, concurrent paths of execution within a single process.

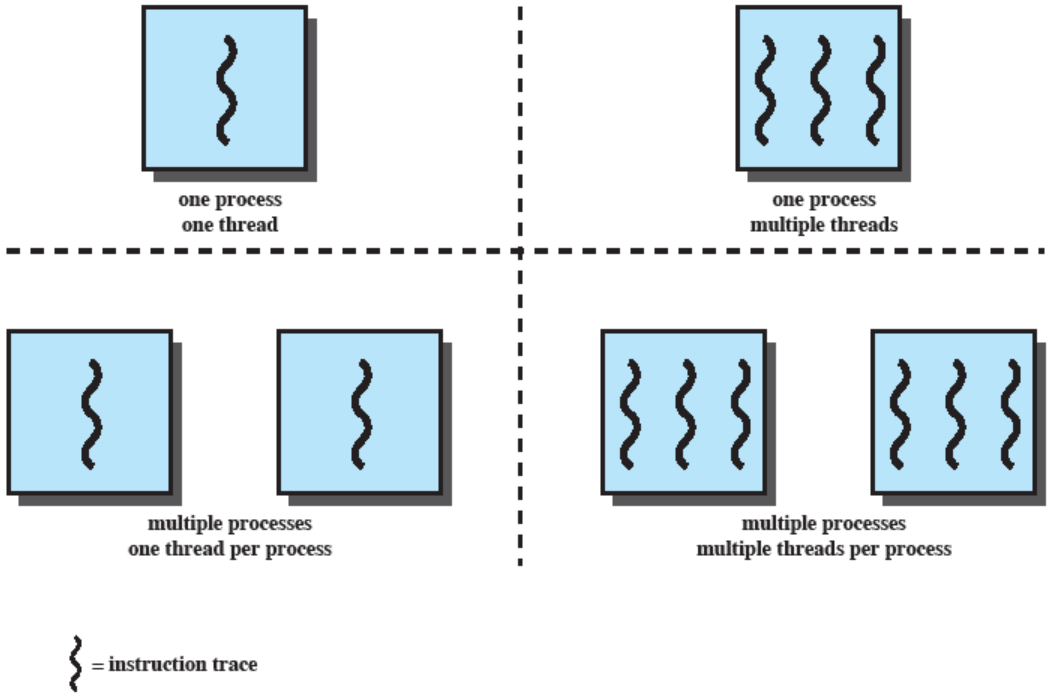


Figure 4.1 Threads and Processes [ANDE97]

One View… One way to view a thread is as an independent program counter operating *within* a process

Operating Systems: Thread

Thread vs Processes

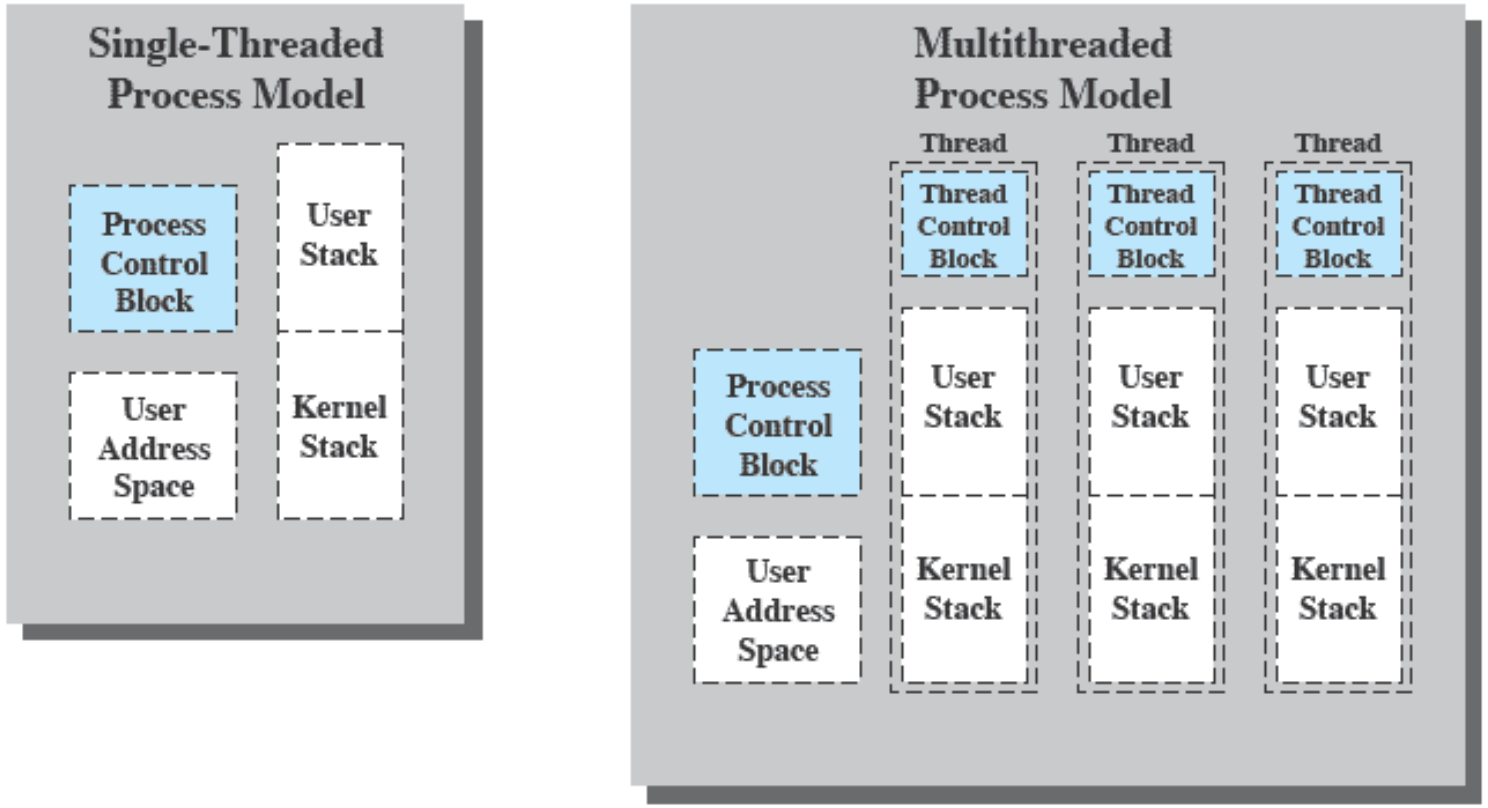
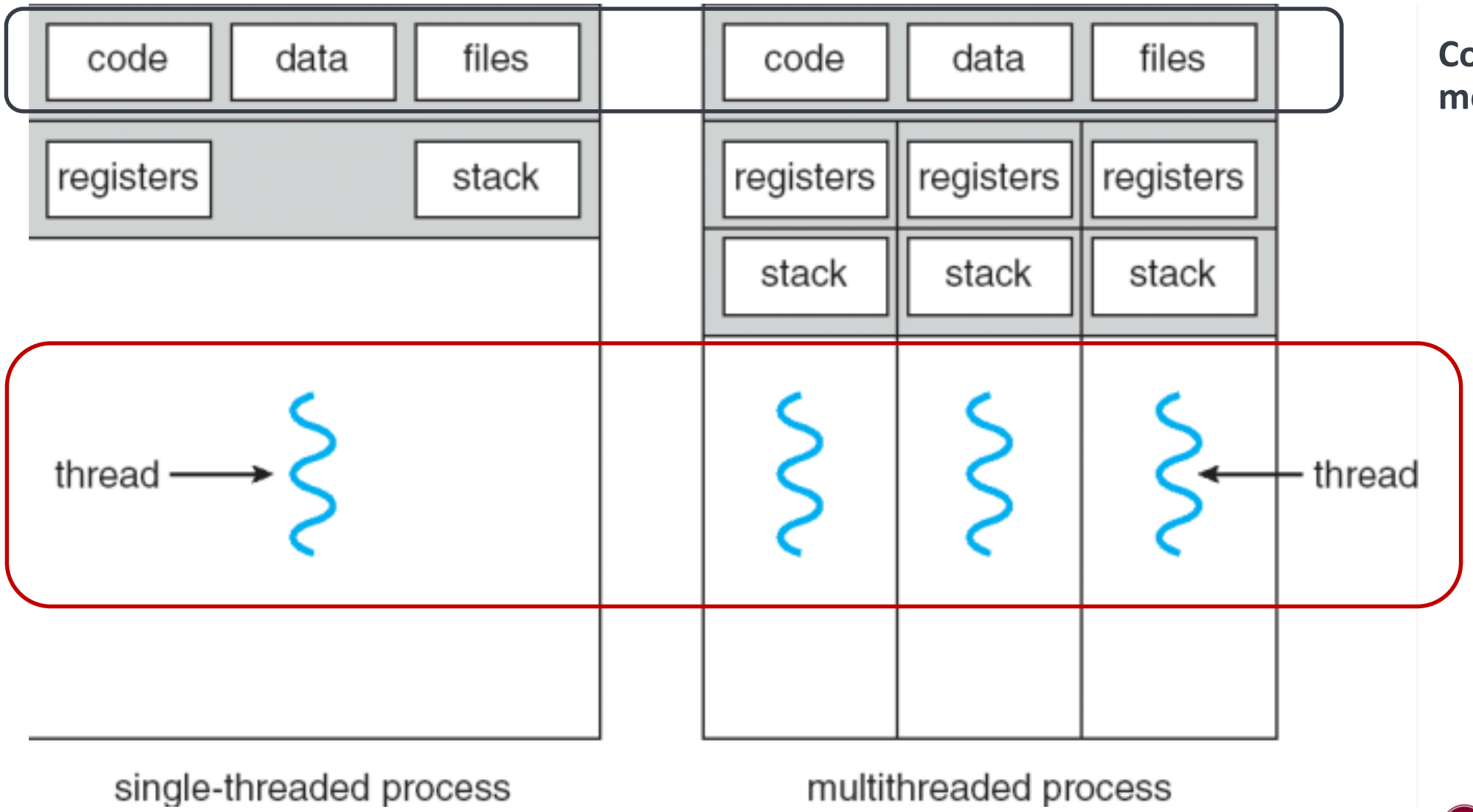


Figure 4.2 Single Threaded and Multithreaded Process Models

Operating Systems: Thread

Definizione



Condivisione delle memoria

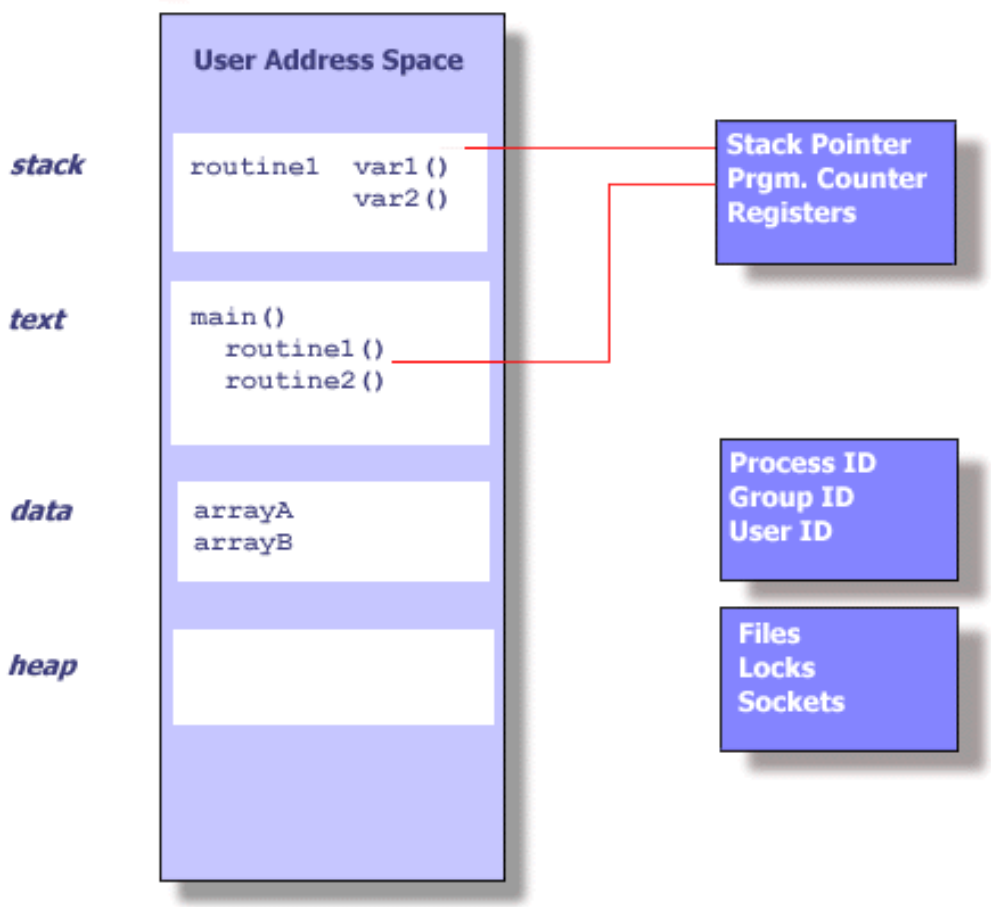
Esecuzione

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

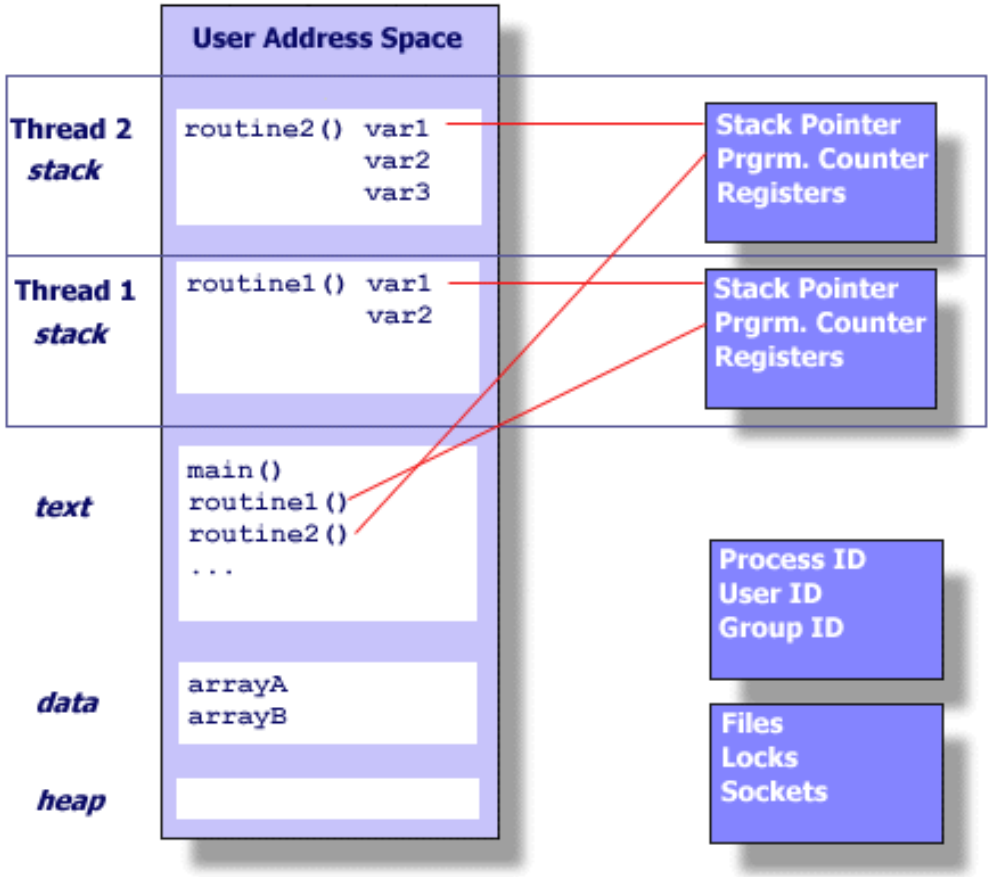
Sapienza Università di Roma

Operating Systems: Thread

Memoria



Processo single-thread (sequenziale)



Processo multi-thread (parallelo)

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Thread

Activities similar to Processes

Threads have **execution states** and may synchronize with one another

- Similar to processes

We look at these **two** aspects of **thread functionality** in turn

- States
- Synchronisation

Thread Execution States

States associated to threads:

- Running, ready, blocked

To change the thread state

- Spawn (another thread)
- Block
  - Issue: can blocking a thread result in blocking some other thread, or even the whole process?
- Unblock
- Finish (thread)
  - Deallocate register context and stacks

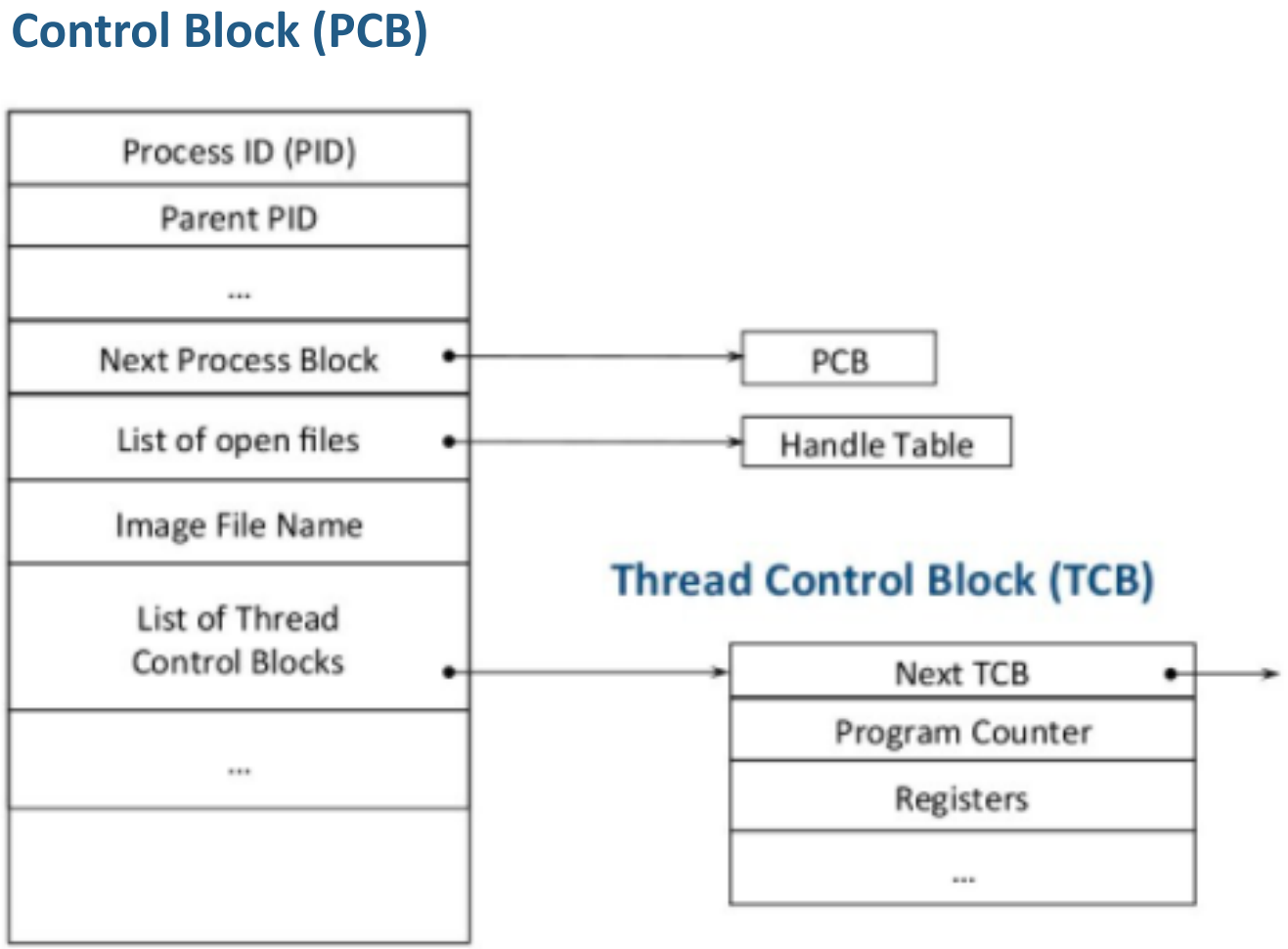
Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Thread

Control Block: PCB e TCB

- spazio di memoria
- risorse private (con le corrispondenti tabelle dei descrittori)



- stato della computazione (registri, stack, PC…)
- attributi (schedulazione, priorità)
- descrittore di thread (tid, priorità, segnali pendenti, …)
- memoria privata: Thread Specific Data (TSD)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

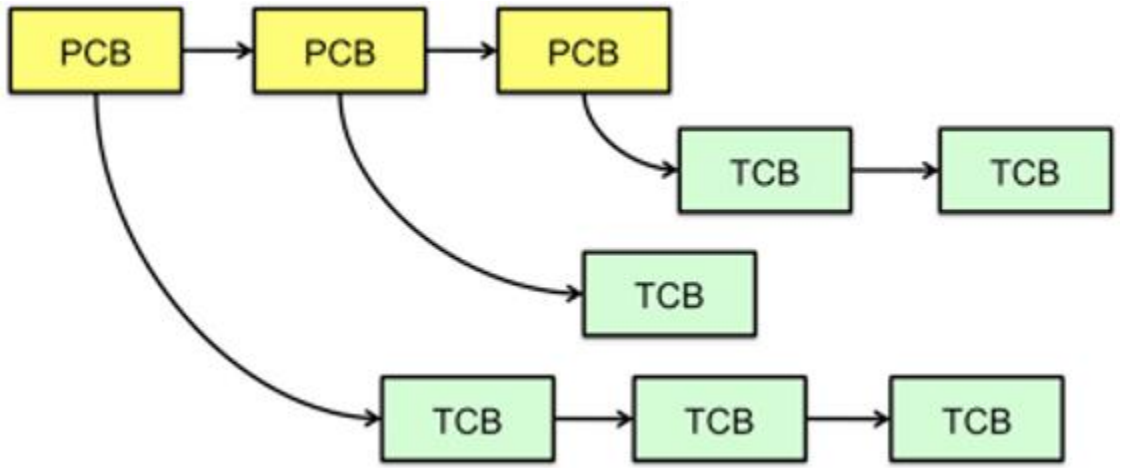
Sapienza Università di Roma

Operating Systems: Thread

TCB: esempio Linux

Thread Control Block (TCB): Struttura dati del **kernel** che mantiene le informazioni sul thread

```
struct tcb {
        short           tcb_state;      /* TCP state                    */
        short           tcb_ostate;     /* output state                 */
        short           tcb_type;       /* TCP type (SERVER, CLIENT)    */
        int             tcb_mutex;      /* tcb mutual exclusion         */
        short           tcb_code;       /* TCP code for next packet     */
        short           tcb_flags;      /* various TCB state flags      */
        short           tcb_error;      /* return error for user side   */

        IPaddr          tcb_rip;        /* remote IP address            */
        short           tcb_rport;      /* remote TCP port              */
        IPaddr          tcb_lip;        /* local IP address             */
        short           tcb_lport;      /* local TCP port               */
        struct          netif           *tcb_pni; /* pointer to our interface */

        tcpseq          tcb_suna;       /* send unacked                 */
        tcpseq          tcb_snext;      /* send next                    */
        tcpseq          tcb_slast;      /* sequence of FIN, if TCBF_SNDFIN */
        long            tcb_swindow;    /* send window size (octets)    */
        tcpseq          tcb_lwseq;      /* sequence of last window update */
        tcpseq          tcb_lwack;      /* ack seq of last window update */
        int             tcb_cwnd;       /* congestion window size (octets) */
        int             tcb_ssthresh;   /* slow start threshold (octets) */
        int             tcb_smss;       /* send max segment size (octets) */
        tcpseq          tcb_iss;        /* initial send sequence        */

        int             tcb_srt;        /* smoothed Round Trip Time     */
        int             tcb_rtde;       /* Round Trip deviation estimator */
        int             tcb_persist;    /* persist timeout value        */
        int             tcb_keep;       /* keepalive timeout value      */
        int             tcb_rexmt;      /* retransmit timeout value     */
        int             tcb_rexmtcount; /* number of rexmts sent        */

        tcpseq          tcb_rnext;      /* receive next                 */
        tcpseq          tcb_rupseq;     /* receive urgent pointer       */
        tcpseq          tcb_supseq;     /* send urgent pointer          */

        int             tcb_lqsize;     /* listen queue size (SERVERs)  */
        int             tcb_listenq;    /* listen queue port (SERVERs)  */
        struct tcb *tcb_pptcb;  /* pointer to parent TCB (for ACCEPT)   */
        int             tcb_ocsem;      /* open/close semaphore         */
        int             tcb_dvnum;      /* TCP slave pseudo device number */

        int             tcb_ssema;      /* send semaphore               */
        u_char          *tcb_sndbuf;    /* send buffer                  */
        int             tcb_sbstart;    /* start of valid data          */
        int             tcb_sbcount;    /* data character count         */
        int             tcb_sbsize;     /* send buffer size (bytes)     */

        int             tcb_rsema;      /* receive semaphore            */
        u_char          *tcb_rcvbuf;    /* receive buffer (circular)    */
        int             tcb_rbstart;    /* start of valid data          */
        int             tcb_rbcount;    /* data character count         */
        int             tcb_rbsize;     /* receive buffer size (bytes)  */
        int             tcb_rmss;       /* receive max segment size     */
        tcpseq          tcb_cwin;       /* seq of currently advertised window */
        int             tcb_rseqq;      /* segment fragment queue       */
        tcpseq          tcb_finseq;     /* FIN sequence number, or 0    */
        tcpseq          tcb_pushseq;    /* PUSH sequence number, or 0   */
};
```



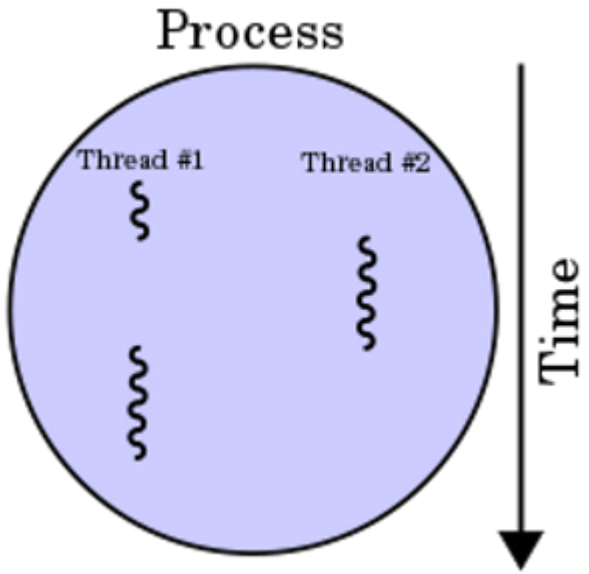
Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Thread

Threat concorrenti e paralleli

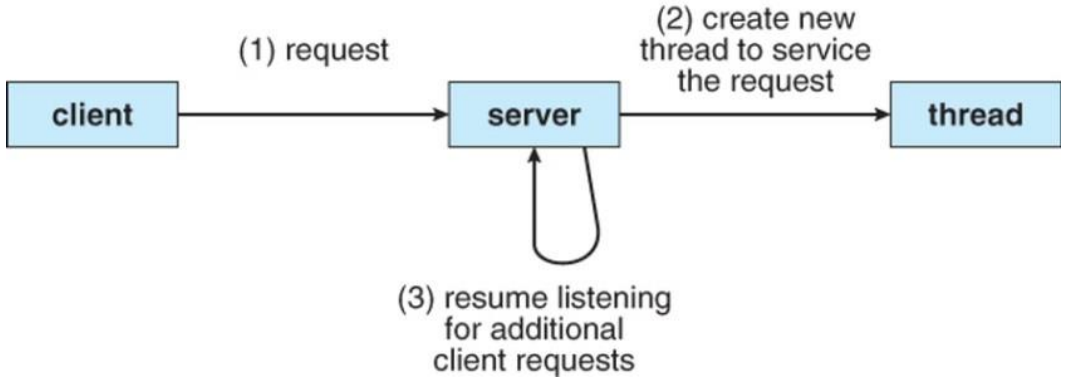
Concurrent Thread



Es. MS Word ha più thread che eseguono diversi compiti in parallelo, tramite thread:

- prendere l'input da tastiera
- controllo ortografico
- conteggio dell'ortografia
- …

Parallel Thread



Es. il web server è un sistema multihreaded dove ogni richiesta è gestita da un hread separato.

Se il web server fosse single-thrad potrebbe servire solo un client alla volta. Il tempo di attesa per la risposta potrebbe crescere a dismisura.

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

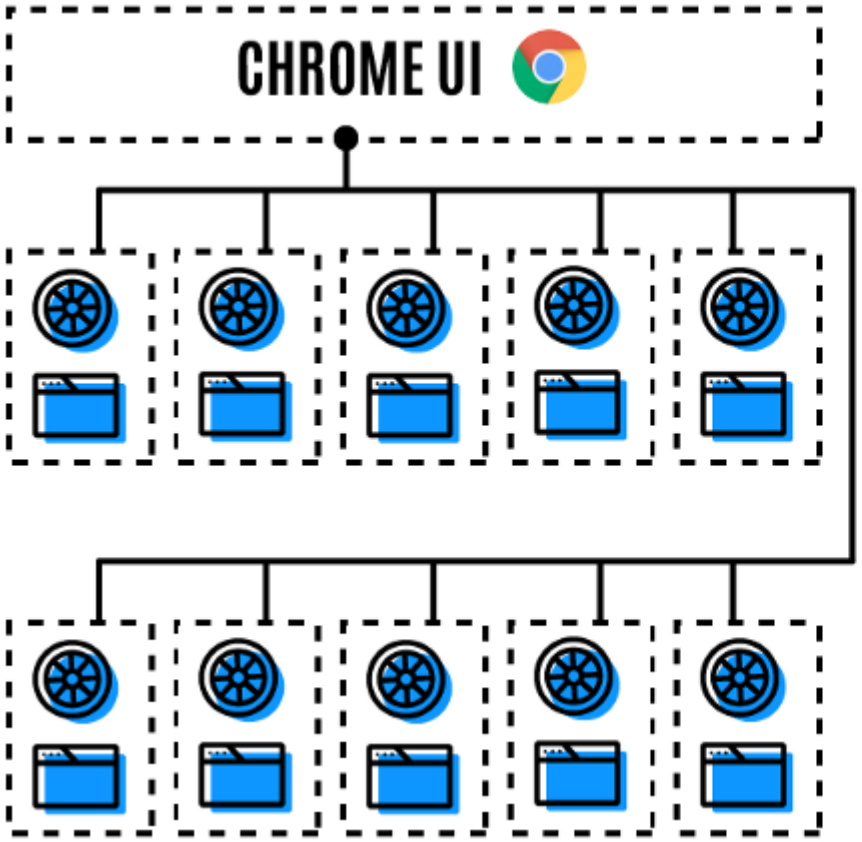
SAPIENZA UNIVERSITÀ DI ROMA

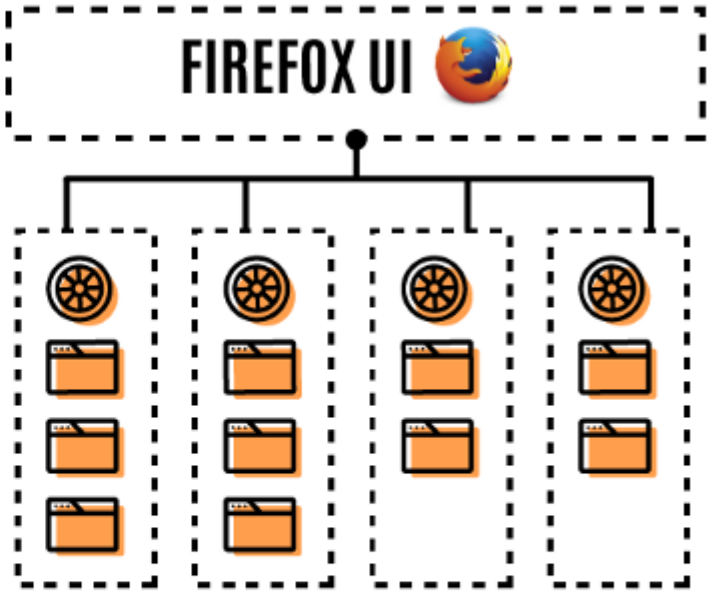
Operating Systems: Thread

Thread in processi utente: esempi

Ad ogni tab corrisponde un processo separato. All'interno di questo sono attivati **thread concorrenti** (trasporto dati, rendering, etc).

Alte performance ma consumo di memoria e batterie





I primi 4 tab sono implementati in processi diversi. I successivi sono implementati tramite **thread paralleli**.

Prestazioni minori ma risparmio di risorse.

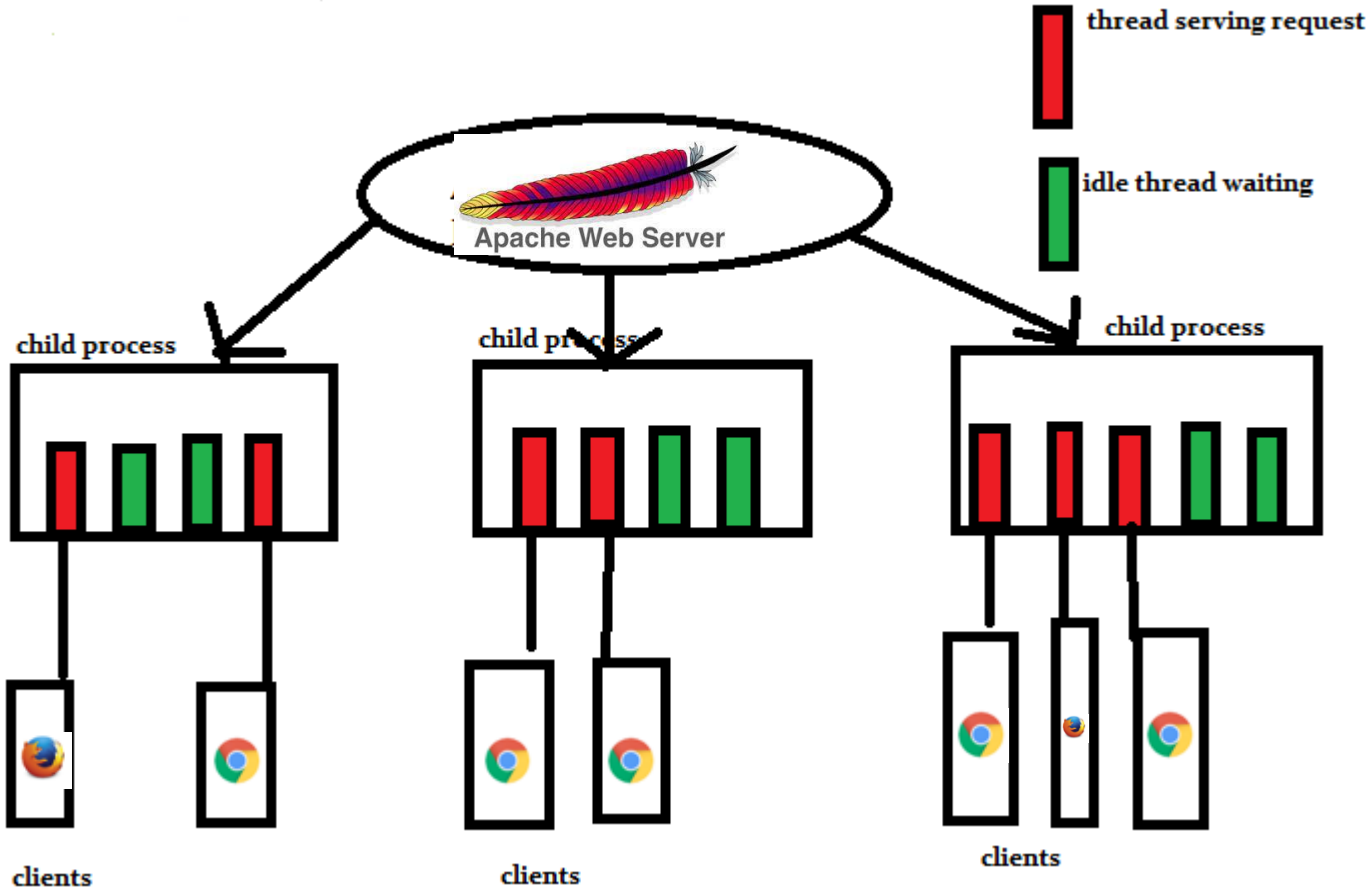
DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: Thread

Thread in processi server (daemon): esempio

Il web server di Apache (httpd: HTTP Daemon) crea un certo numero di figli ed in ognuno di essi viene creato un certo numero di **thread paralleli**.



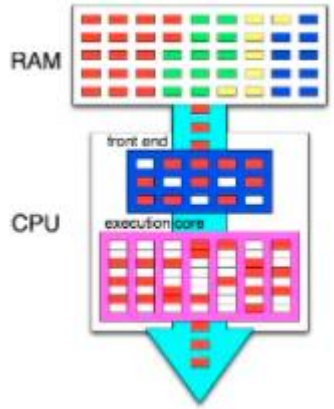
Operating Systems: Thread

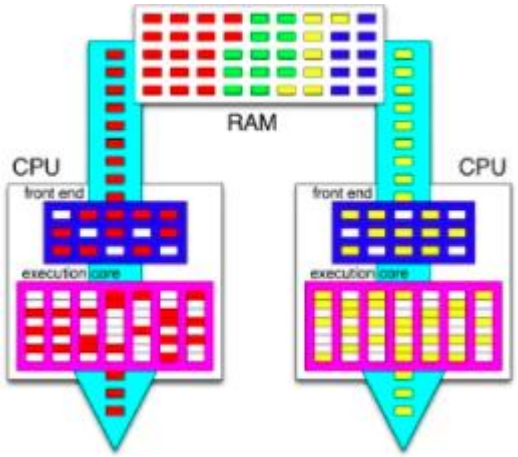
Vantaggi/Svantaggi

Ambito	Vantaggi	Svantaggi
Tempi di Attesa	Un programma può continuare la computazione anche se un suo thread è bloccato (e.g. attesa I/O)	
Condivisione delle Risorse	• condivisione delle informazioni tramite IPC (memoria condivisa o messaggi) • condivisione della memoria e delle risorse del processo che li genera (i.e. molti thread nello stesso spazio di indirizzi)	Difficoltà di ottenere risorse private: per ottenere memoria privata all'interno di un thread è possible ricorrere ad appositi meccanismi
Concorrenza	Creazione e context switch molto più leggeri rispetto a quelli dei processi	Pericolo di interferenza • la condivisione delle risorse accentua il pericolo di interferenza • gli accessi concorrenti devono essere sincronizzati di modo da evitare interferenze (thread safeness)
Scalabilità	I thread possono essere eseguiti in parallelo su architettura multiprocessore	

Operating Systems: Thread

Programmazione Multicore





Single-Core: In un sistema con un'unica unità di calcolo l'esecuzione concorrente consiste nell'interfoliazione (interleaving) dei thread

Multi-Core (es. SMP: Symmetric MultiProcessor) l'esecuzione concorrente consente ai thread di essere eseguiti in parallelo. I SO e le applicazioni devono essere appositamente progettate:

- **SO**: devono fornire schedulatori che utilizzano diverse unità di calcolo
- **Applicazioni**: devono essere progettate tenendo in considerazione diversi fattori:
  - **Separazione dei task**: individuazione dei task che possono essere eseguiti in parallelo
  - **Bilanciamento dei task**: devono eseguire compiti che richiedono all'incirca tempo e risorse paragonabili
  - **Suddivisione dei dati**: devono essere definiti dei dati specifici per i vari task
  - **Dipendenza dei dati**: l'accesso ai dati condivisi deve essere fatto in modo sincronizzato (thread safeness)
  - **Test e debugging**: a causa dei diversi flussi di esecuzione test e debugging sono più difficili

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

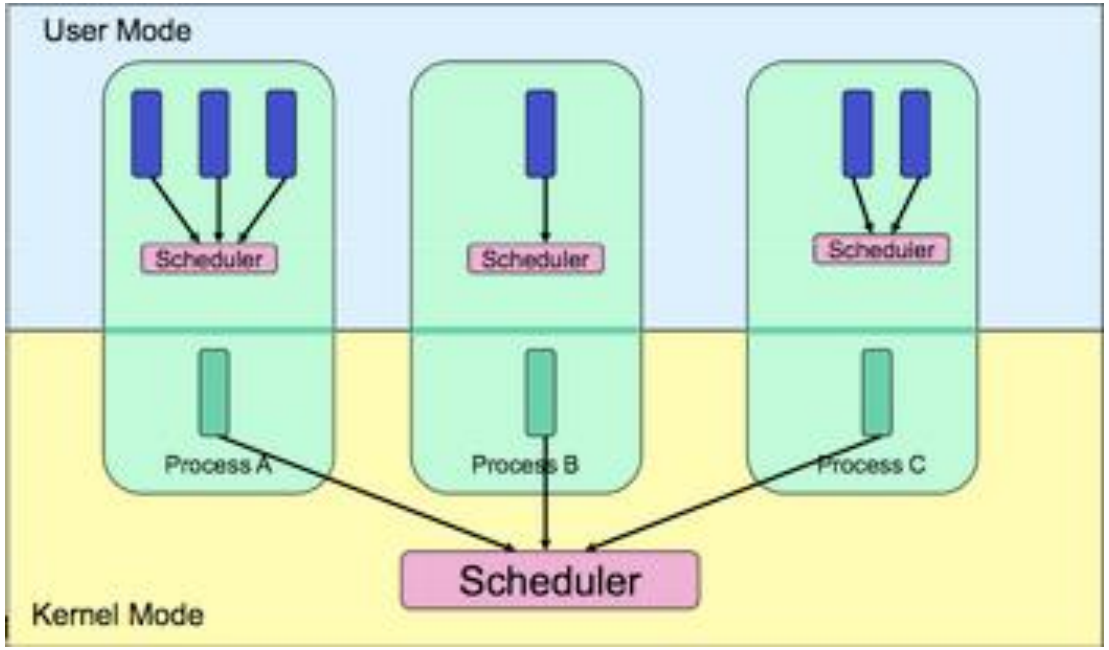
SAPIENZA UNIVERSITÀ DI ROMA

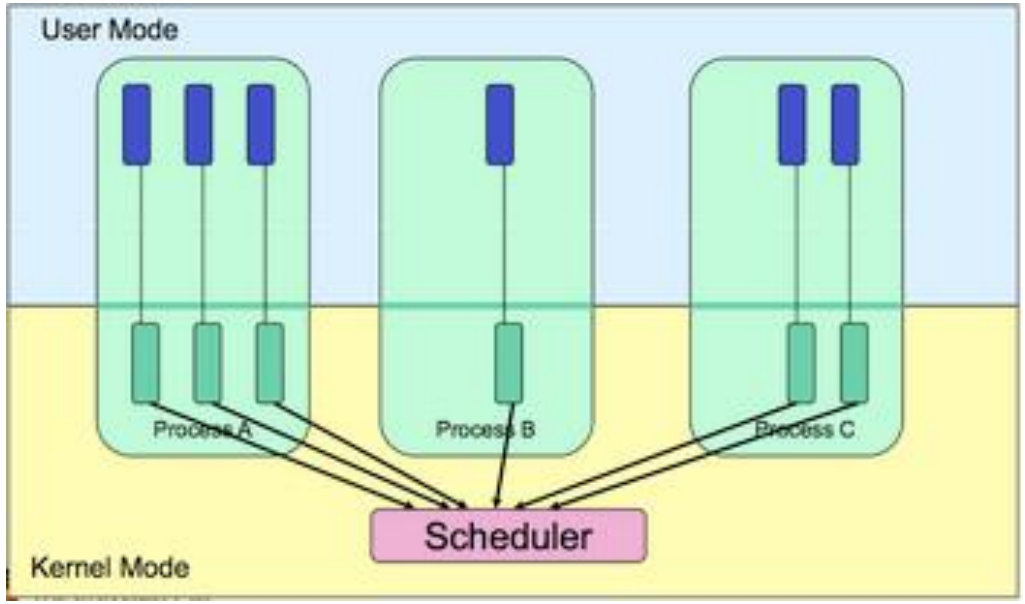
Operating Systems: Thread

Implementation Categories: ULT vs KLT

User-Level Thread: thread all'interno del processo utente gestiti da una libreria specifica (da un supporto run-time)

- sono gestiti senza l'aiuto del kernel
- lo switch non richiede chiamate al kernel





Kernel-Level Thread: gestione dei thread affidata al kernel tramite chiamate di sistema

- gestione integrata processi e thread dal kernel
- lo switch è provocato da chiamate al kernel

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

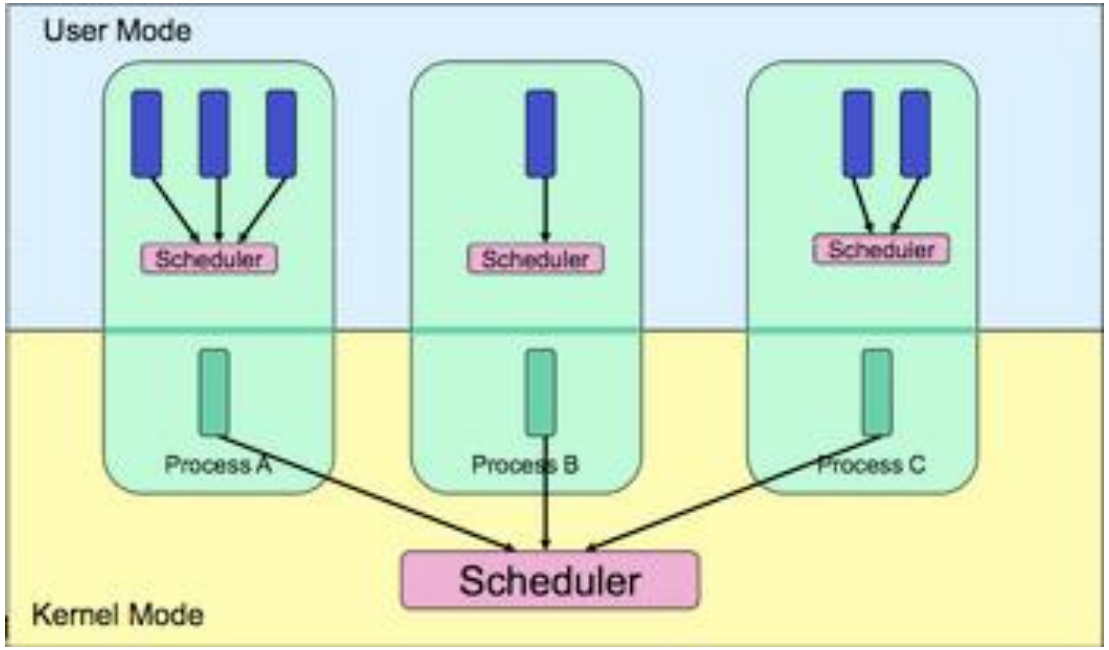
SAPIENZA UNIVERSITÀ DI ROMA

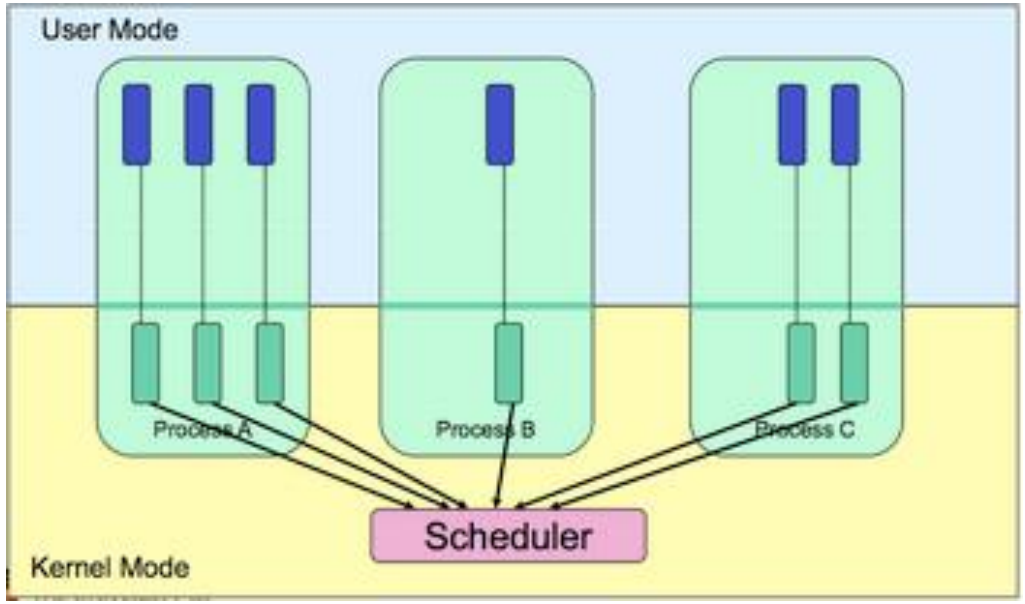
Operating Systems: Thread

Modello di Thread: User vs Kernel Level

User-Level Thread: thread all'interno del processo utente gestiti da una libreria specifica (da un supporto run-time)

- sono gestiti senza l'aiuto del kernel
- lo switch non richiede chiamate al kernel





Kernel-Level Thread: gestione dei thread affidata al kernel tramite chiamate di sistema

- gestione integrata processi e thread dal kernel
- lo switch è provocato da chiamate al kernel

Chiamati anche:

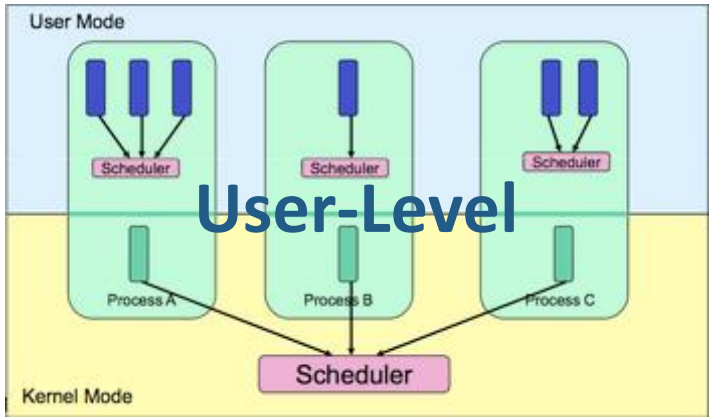
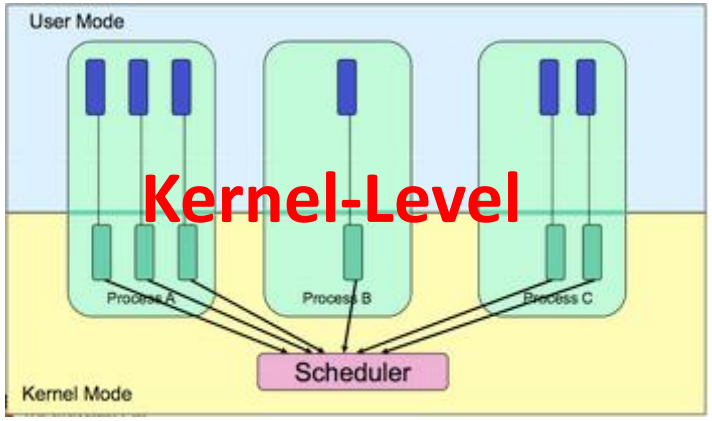
- kernel-supported threads
- lightweight processes

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

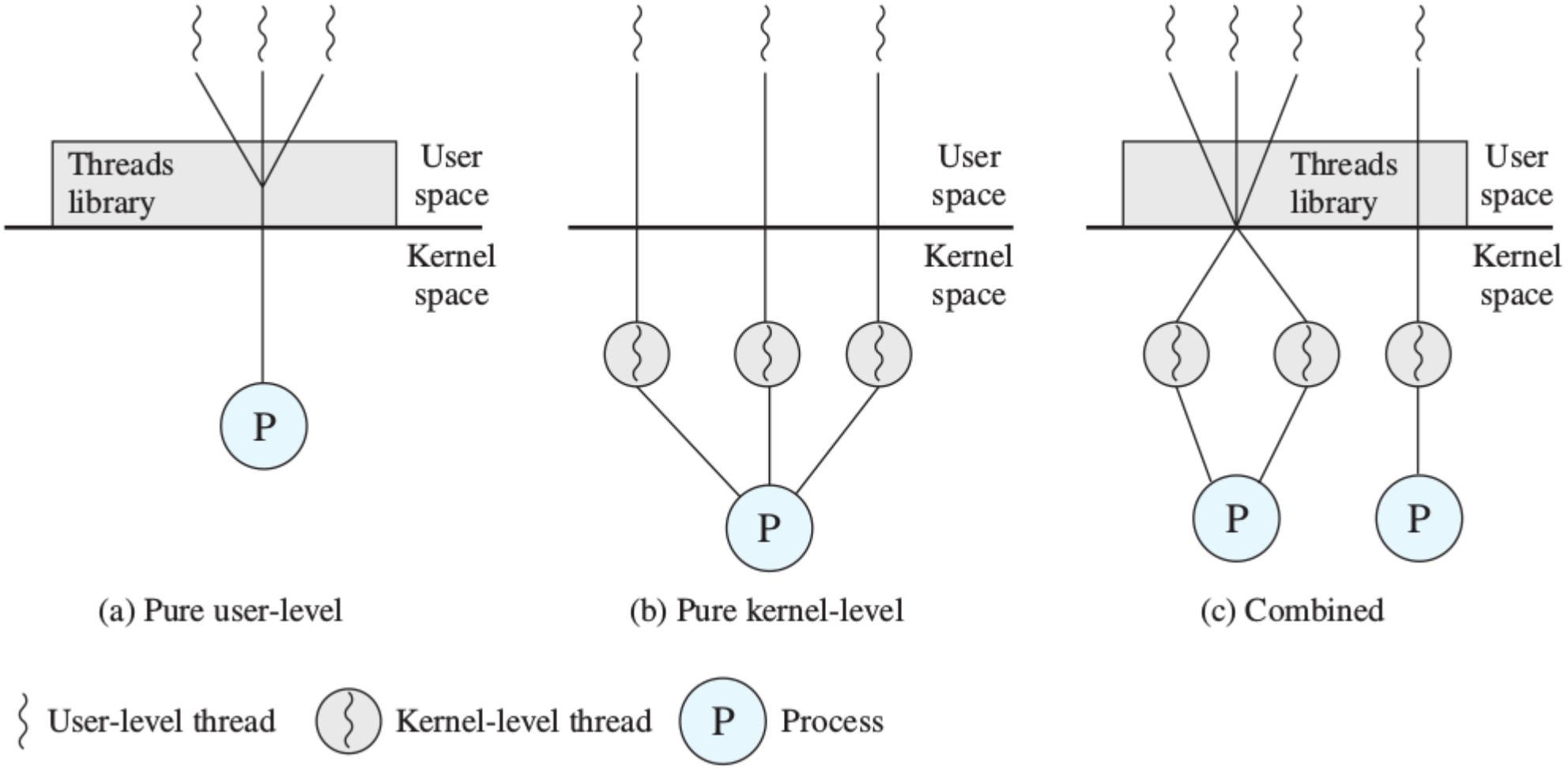
Operating Systems: Thread

User vs Kernel Level

Ambito	Vantaggi	Svantaggi
	• Lo switch non coinvolge il kernel, e quindi non ci sono cambiamenti della modalità di esecuzione • Maggiore libertà nella scelta dell'algoritmo di scheduling che può anche essere personalizzato • Poiché le chiamate possono essere raccolte in una libreria, c'è maggiore portabilità tra SO	• Una chiamata al kernel può bloccare tutti i thread di un processo, indipendentemente dal fatto che in realtà solo uno dei suoi thread ha causato la chiamata bloccante • In sistemi a multiprocessore simmetrico (SMP) due processori non possono essere associati a due thread del medesimo processo
	• Il kernel può eseguire più thread dello stesso processo anche su più processori • Il kernel stesso può essere scritto multithread	• Lo switch coinvolge chiamate al kernel e questo comporta un costo • L'algoritmo di scheduling è meno facilmente personalizzabile • Meno portabile

Operating Systems: Thread

ULT e/o KLT



Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

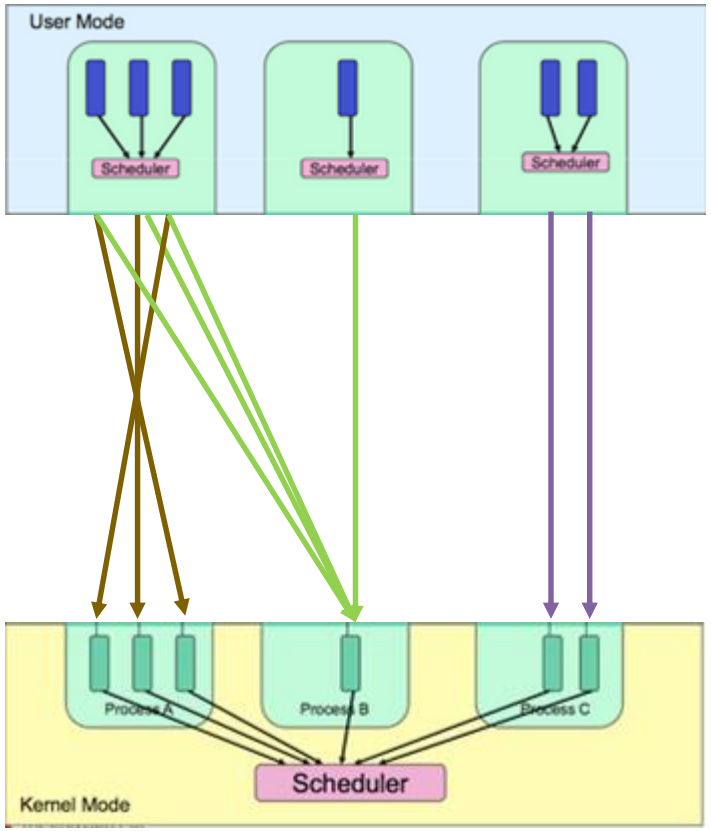
Operating Systems: Thread

Modello multi-Thread

I **thread a livello utente**

vengono messi in relazione per permettere l'accesso alle risorse

con **I thread a livello kernel**



Esistono diversi tipi di relazione tra thread a livello utente e thread a livello kernel:

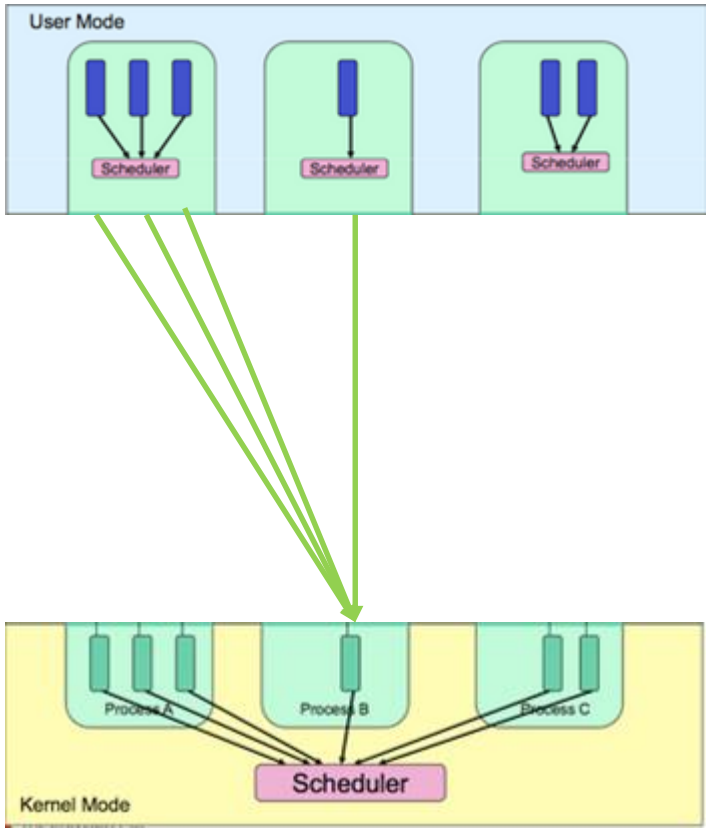
- **Molti-a-uno**
- **Uno-a-uno**
- **Molti-a-molti**
- **A due livelli**

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: Thread

multi-Thread: Molti-a-Uno



Molti-a-uno

riunisce molti thread di livello utente in un unico kernel thread

Vantaggi

- Gestione dei thread efficiente

Svantaggi

- Intero processo bloccato se un thread invoca una chiamata di sistema bloccante
- Un solo thread può accedere al Kernel (Impossibile eseguire thread in parallelo)

Esempi:

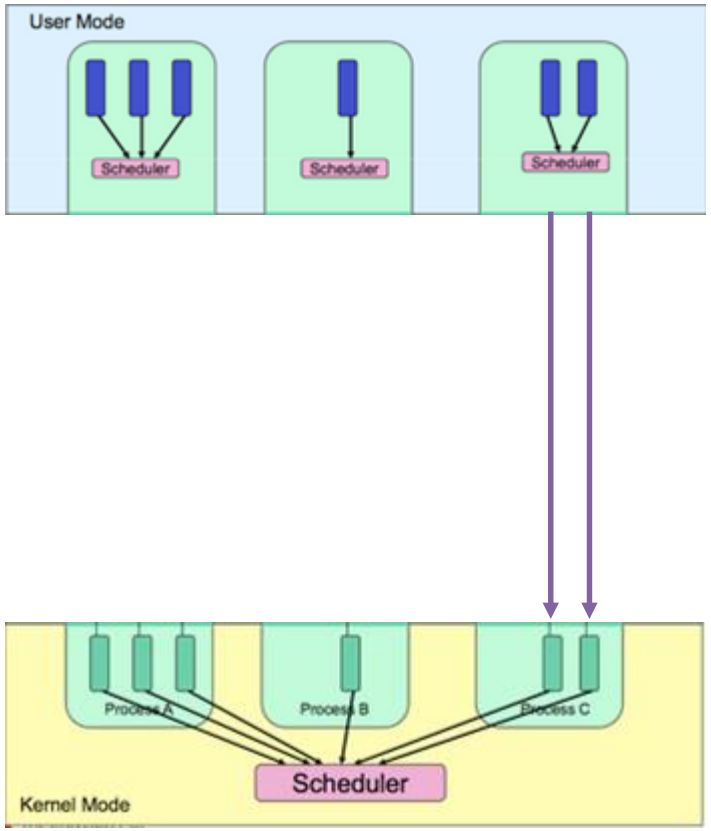
- Solaris Green Threads
- GNU Portable Threads

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Thread

multi-Thread: Uno-a-Uno



Uno-a-uno

mappa ciascun thread utente in un kernel thread

Vantaggi

- Maggiore concorrenza (possibili thread in parallelo e le chiamate bloccanti non bloccano tutti i thread)

Svantaggi

- La creazione di molti thread a livello kernel compromette le prestazioni dell'applicazione (es. limite del numero di thread a livello kernel)

Esempi:

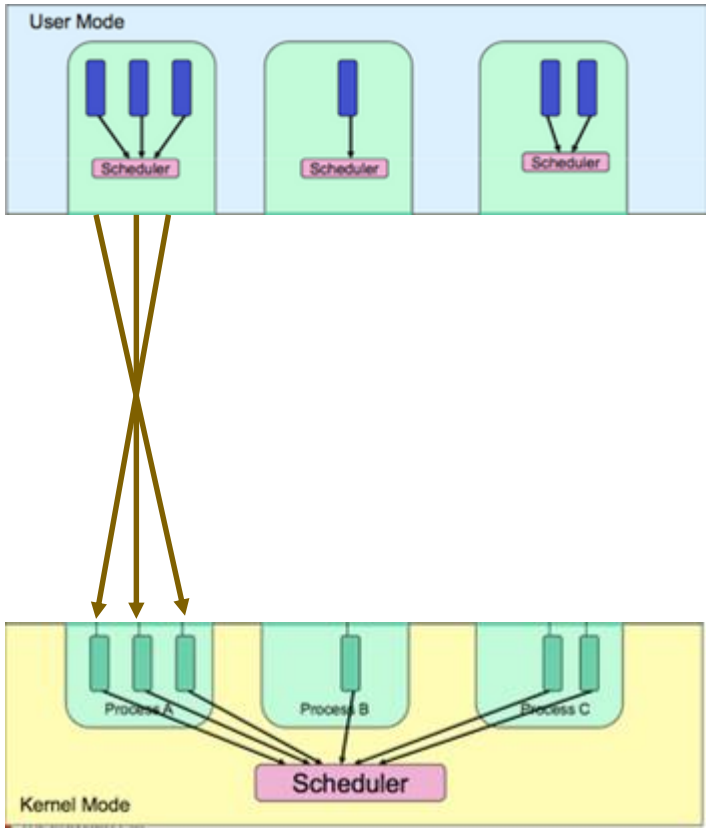
- Windows NT/XP/2000
- Linux
- Solaris 9

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Thread

multi-Thread: Molti-a-Molti



Molti-a-molti

Permette di aggregare molti thread a livello utente verso un numero più piccolo o equivalente di kernel thread

Vantaggi

- Risolve le limitazioni dei modelli precedenti (Il numero max di processi a livello Kernel può essere personalizzato in base all'architettura; n_max maggiore per multicore)

Esempi:

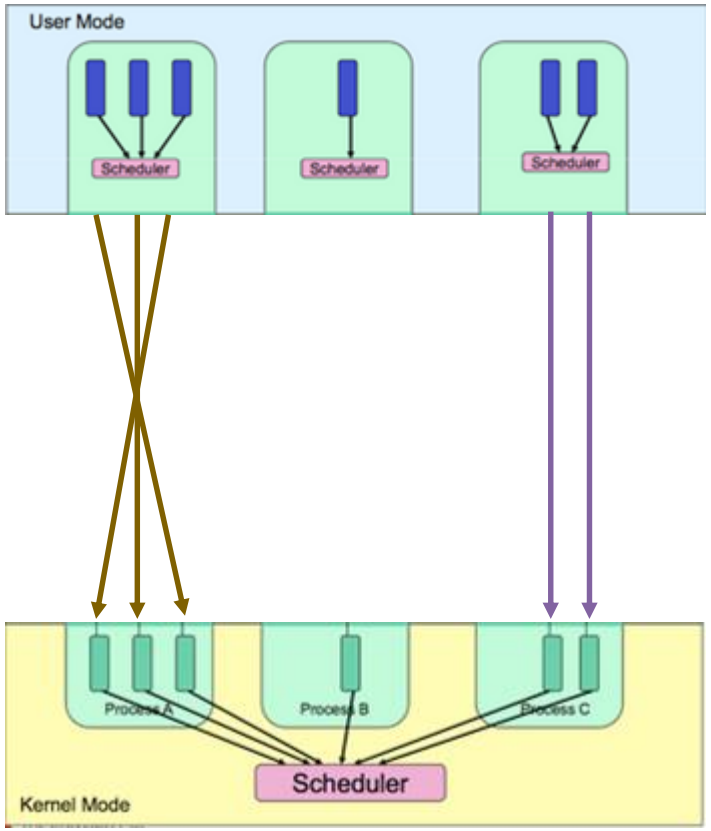
- Solaris, versioni precedenti alla 9
- Windows NT/2000 (pacchetto ThreadFiber)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Thread

multi-Thread: a Due Livelli



A due livelli

Simile al modello molti-a-molti, eccetto che permette anche di associare un thread di livello utente ad un kernel thread

Esempi:

- IRIX
- HP-UX
- Tru64 UNIX

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

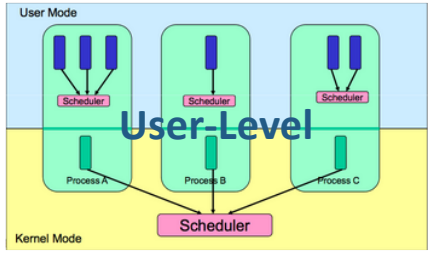
Operating Systems: Thread

Librerie per i Thread

La gestione dei thread è effettuata attraverso specifiche librerie

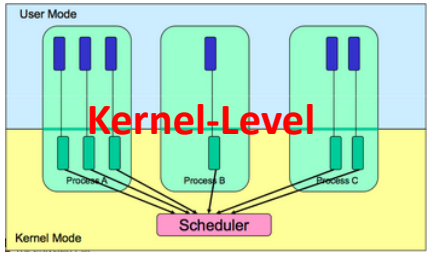
Libreria collocata a livello utente (POSIX Pthreads)

- Codice e strutture dati per la libreria risiedono nello spazio utente
- Invocare una funzione della libreria non significa invocare una chiamata di sistema



Libreria collocata a livello kernel (POSIX Pthreads, Win32 thread)

- Codice e strutture dati per la libreria risiedono nello spazio del kernel
- Invocare una funzione della libreria significa invocare una chiamata di sistema



N.B. Java thread (dipende dal sistema operativo ospitante)

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: POSIX Thread

Librerie per i Thread: Pthread – User Level Thread

standard **POSIX** (IEEE 1003.1c) API per la creazione e la sincronizzazione dei thread

- Fornisce una specifica per il comportamento della libreria dei thread
- i progettisti di sistemi operativi possono implementare la specifica nel modo che desiderano
- Frequente nei sistemi operativi di UNIX (Solaris, Linux, Mac OS X)

Pthreads are C language programming types

```
#include <pthread.h>
```

`pthread_t`: specifica l'identificatore di un thread
`pthread_attr`: specifica gli attributi di un thread (e.g. proprietà per lo scheduling)
`pthread_attr_init(&pthread_attr)`: inizializza gli attributi di un thread ai valori di default
`pthread_create(pthread_t *tid, pthread_attr *attr, void*(), void*argv)`: crea un user thread al quale associare gli attributi inizializzati, la funzione da eseguire e gli argomenti
`pthread_join (pthread_t *tid, <value>)`: mette in pausa il thread corrente fino alla terminazione del thread tid. Se il secondo parametro non è nullo viene usato per restituire il valore di ritorno del thread

Operating Systems: Thread

System Call fork() ed exec()

Solitamente, vi sono varie versioni della funzione `fork()` a seconda che si desideri:

- duplicare solo il thread che la invoca
- Duplicare tutti i thread del processo ospitante

In linux esiste la funzione `clone()`: una nuova chiamata di sistema versatile che può essere utilizzata per creare un nuovo thread di esecuzione. A seconda delle opzioni passate, il nuovo thread di esecuzione può aderire alla semantica di un processo UNIX, un thread POSIX, una via di mezzo o qualcosa di completamente diverso (come un contenitore diverso). È possibile specificare tutti i tipi di opzioni che determinano se la memoria, i descrittori di file, i vari spazi dei nomi, i gestori dei segnali e così via vengano condivisi o copiati. La funzione `fork()` è attualmente implementata come wrapper che richiama `clone()`.

La funzione `exec()`, viceversa, funziona analogamente a quanto visto prima della introduzione dei thread: rimpiazza l'intero processo, quindi anche tutti i thread.

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Thread

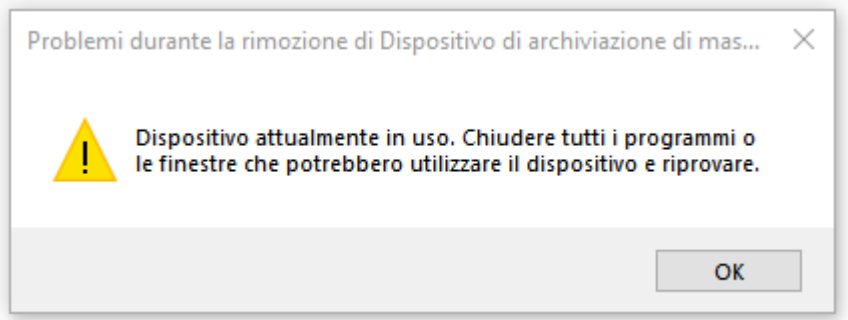
Cancellazione dei Thread

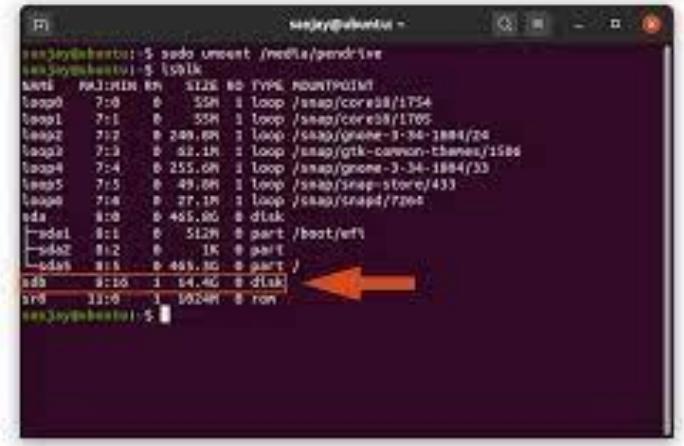
Terminazione di un thread prima che abbia completato l'esecuzione

Può avvenire in due differenti scenari:

- **Cancellazione asincrona**: un thread termina immediatamente il thread target (se il thread sta operando su un file potrebbe portare ad una situazione in cui la risorsa non è libera nonostante il processo sia terminato)

- **Cancellazione differita**: il thread target può periodicamente controllare se deve terminare, cosi da terminare in modo opportuno (se il thread sta operando su un file la risorsa viene liberata opportunamente)

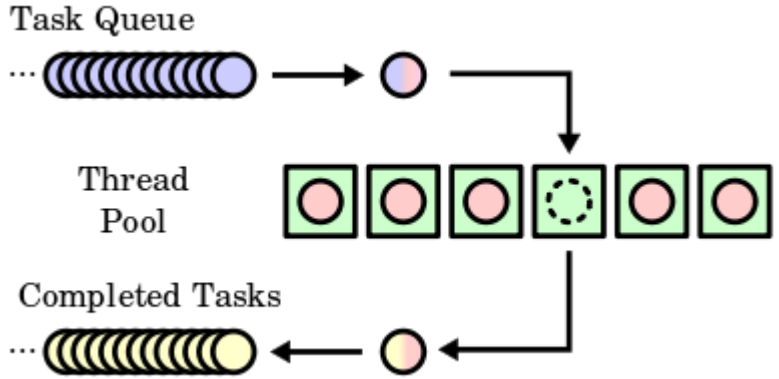




Operating Systems: Thread

Thread Pool

Thread Pool: creare un gruppo di thread (pool) all'avvio del processo ed assegnarli un lavoro quando richiesto. Al completamento del lavoro il thread torna nel gruppo d'attesa.



Vantaggi:

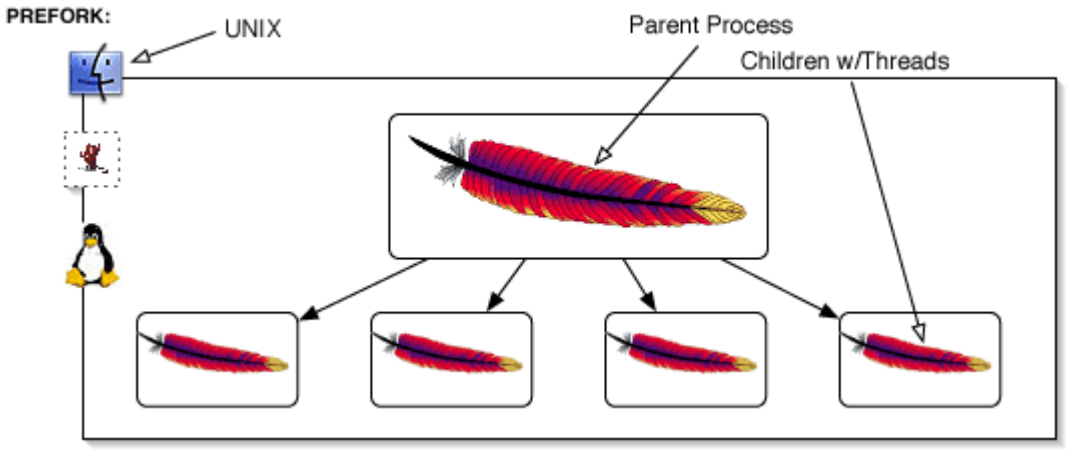
- Servire la richiesta all'interno di un thread esistente è tipicamente più veloce che attendere la creazione di un thread
- Un pool di thread limita il numero di thread esistenti in contemporanea (Il numero massimo può essere adattato a runtime in architetture raffinate)

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

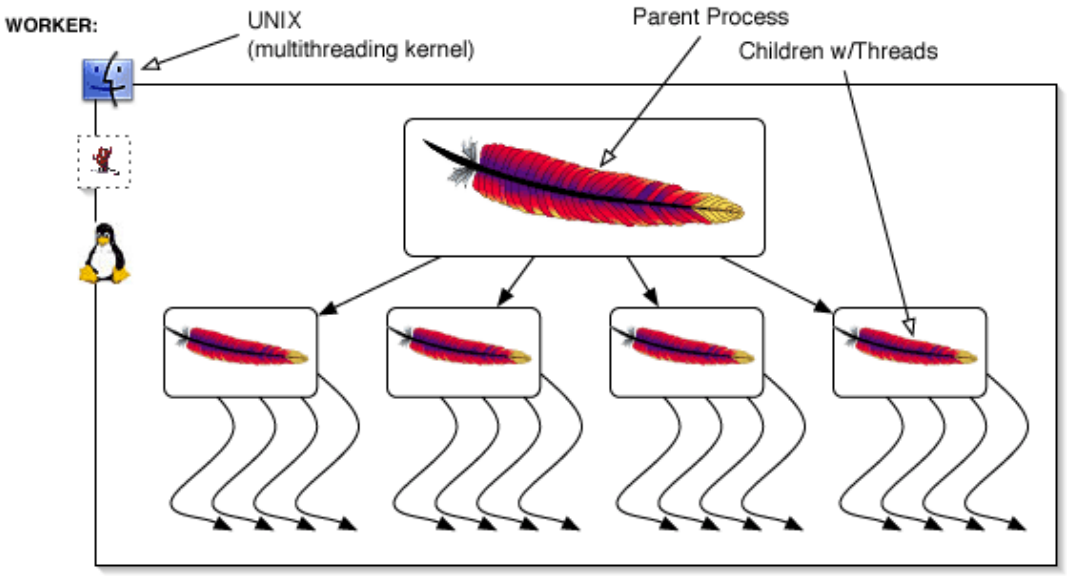
SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Thread

Thread Pool: es. Apache



Prefork (no thread): Apache 1.x.y si avvia come processo server padre e da esso vengono generati i processi figli. Il server mantiene alcuni figli generati in esecuzione, per gestire le richieste in arrivo, e il server può uccidere i processi figli se ne vengono avviati troppi, evitando che il server vada in crash.



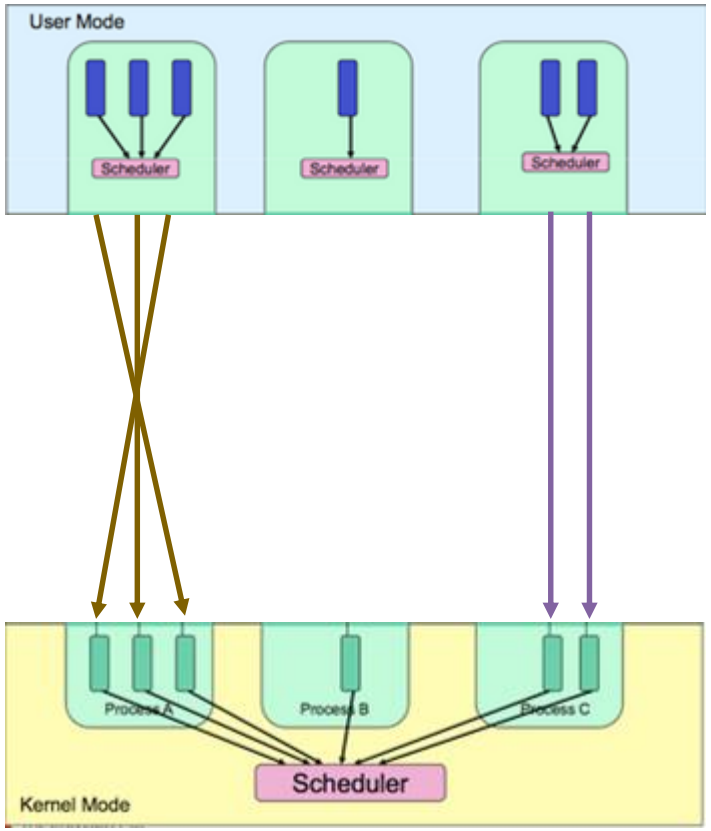
Worker (thread): Apache 2.x.y fa uso del worker, molto simile al modulo PREFORK, ma è un MPM (multi-Processing Module) ibrido thread/processo. Questo significa che si comporta come PREFORK generando figli, e questi figli generano un numero statico di thread. Il worker ha, quindi, un thread per ogni figlio che ascolta la rete, e ogni processo può servire molti processi contemporaneamente.

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Thread

Scheduling: Attivazione 1/3



Le applicazioni multi-thread con modello molti-a-molti o a due livelli richiedono una comunicazione con il kernel:

- per mantenere l'appropriato numero di kernel thread allocati (prestazioni)
- per far sì che il kernel informi l'applicazione di certi eventi (segnali), come il blocco di un thread dell'applicazione (ad es. per un evento di I/O)

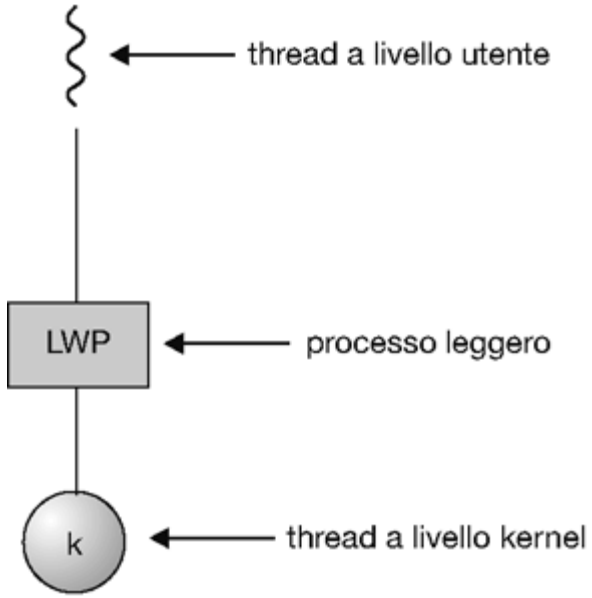
Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Thread

Scheduling: Attivazione 2/3

Struttura dati posta tra i thread kernel e i thread utente: **LightWeight Process** (**LWP**)



Il LWP è visto dalla libreria dei thread utente come un processore virtuale su cui schedulare i thread utente

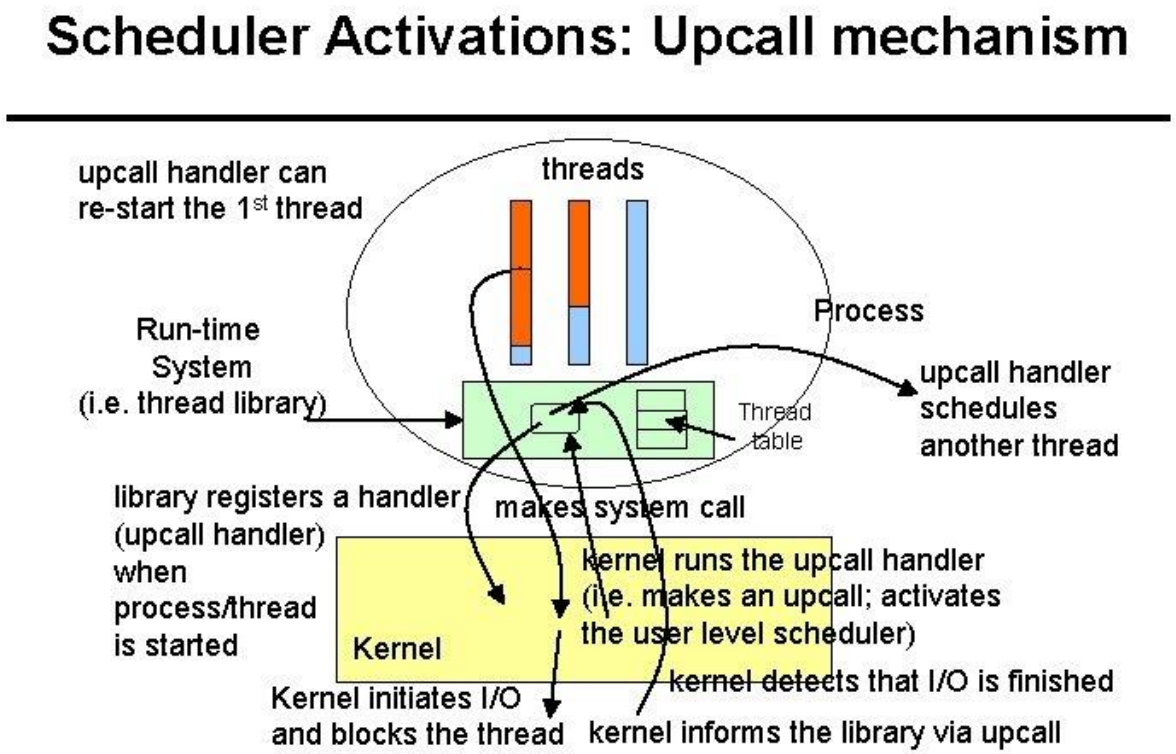
Il blocco di un thread kernel (e.g. I/O) causa il blocco del LWP ➔ si bloccano i thread utente schedulati sul LWP ➔ necessario un LWP per ogni chiamata di sistema concorrente bloccante
(e.g. se un'applicazione ha bisogno di fare 5 letture da file servono 5 LWP)

Operating Systems: Thread

Scheduling: Attivazione 3/3

I thread kernel possono comunicare eventi alla libreria dei thread tramite le **upcall** (chiamate al thread)

Le **upcall** sono gestite a <u>livello utente</u> con un gestore di upcall in esecuzione “just in time” (cioè quando avviene una comunicazione dal kernel all’applicazione) su di un LWP



Operating Systems: Thread

es. Thread in Windows

Implementa la mappatura uno-a-uno. Ogni thread contiene:

- Un identificatore del thread
- Il contesto del thread: set di registri (stato del processore), User stack, Kernel stack, area di memoria privata

ETHREAD (kernel): blocco di esecuzione del thread

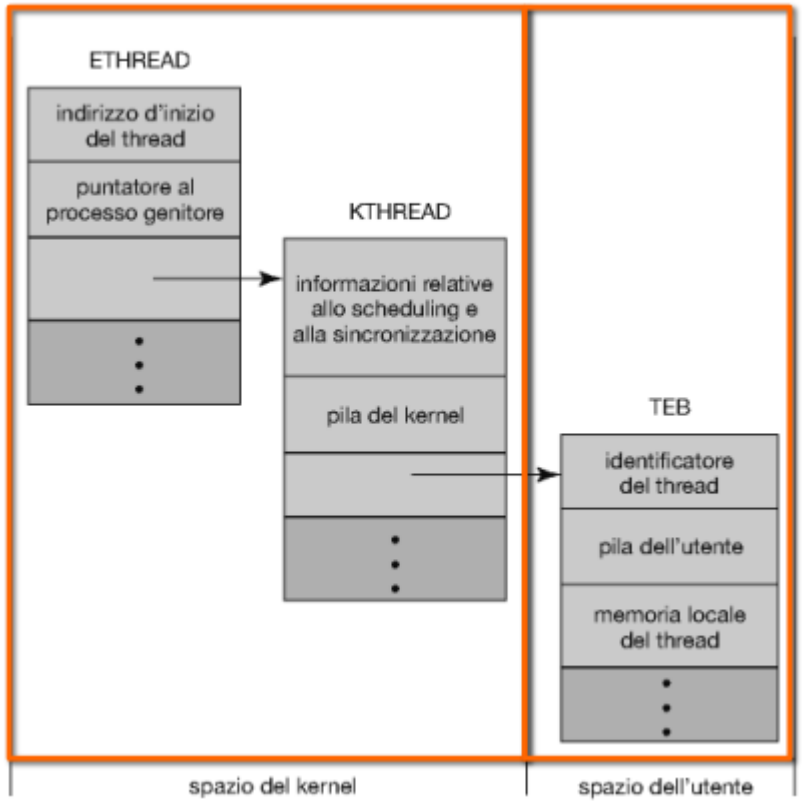
- Puntatore al processo padre
- Indirizzo della funzione eseguita dal thread

KTHREAD (kernel): blocco di kernel del thread

- Informazioni relative allo scheduling e sincronizzazione
- Pila del kernel

Thread Environment Block TEB (user): blocco di ambiente del thread

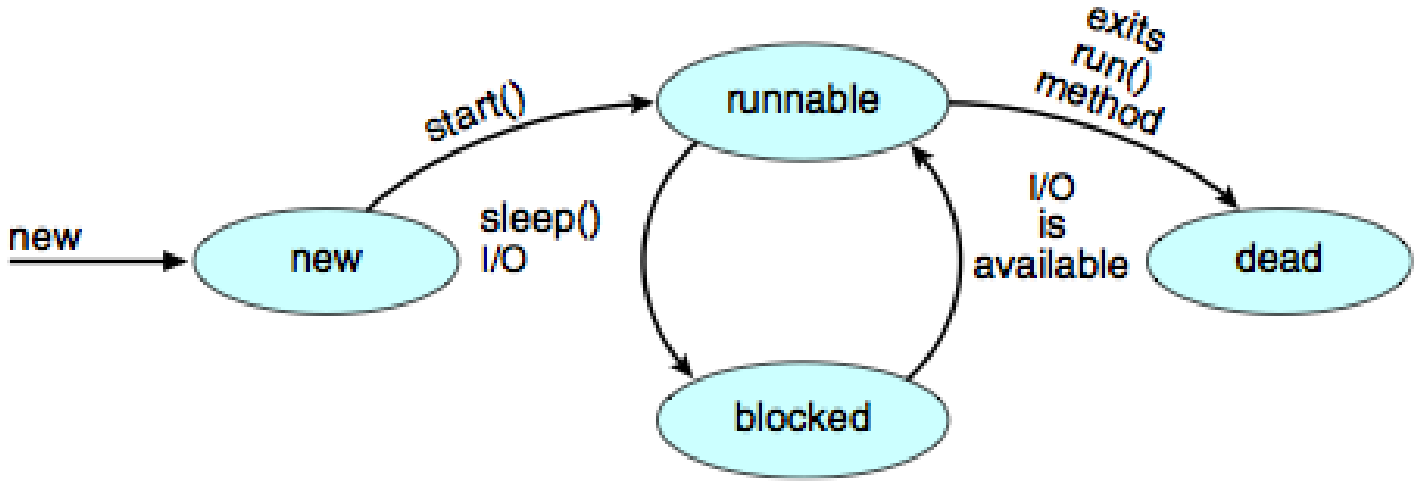
- Pila dell'utente
- ID del thread
- Vettore per dati specifici del thread non condivisi (memoria locale)



Operating Systems: Thread

es. Thread in Java

I thread di Java sono gestiti dalla **JVM**



I thread di Java possono essere creati tramite:

- L'estensione della classe **Thread**
- L'interfaccia **Runnable**

Operating Systems: Thread

es. Thread in Linux

Linux li definisce come ***task*** piuttosto che *thread*

La creazione di un thread avviene attraverso la chiamata di Sistema `clone()`
Questa permette di stabilire il "grado di condivisione" tramite flag parametri

Flag	Significato
CLONE_FS	Condivisione delle informazioni sul file system
CLONE_VM	Condivisione dello stesso spazio di memoria
CLONE_SIGHAND	Condivisione dei gestori dei segnali
CLONE_FILES	Condivisione dei file aperti

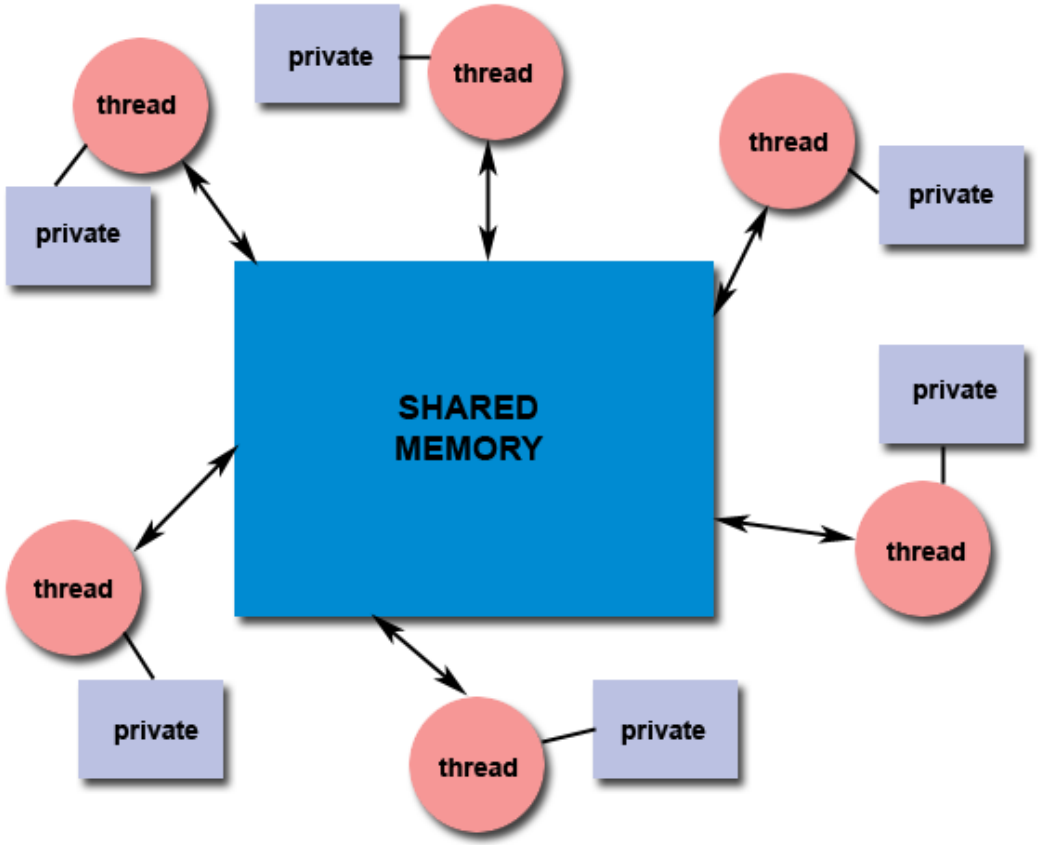
se nessun flag è impostato, non c'è alcuna condivisione e l'effetto di `clone()` è simile a quello della `fork()` (funzione "wrapper")

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Thread

Shared-memory Model



All threads have access to the *same* global, shared memory

Threads also have their own private data

Programmers are responsible for synchronizing access to (i.e., protecting) globally shared data

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

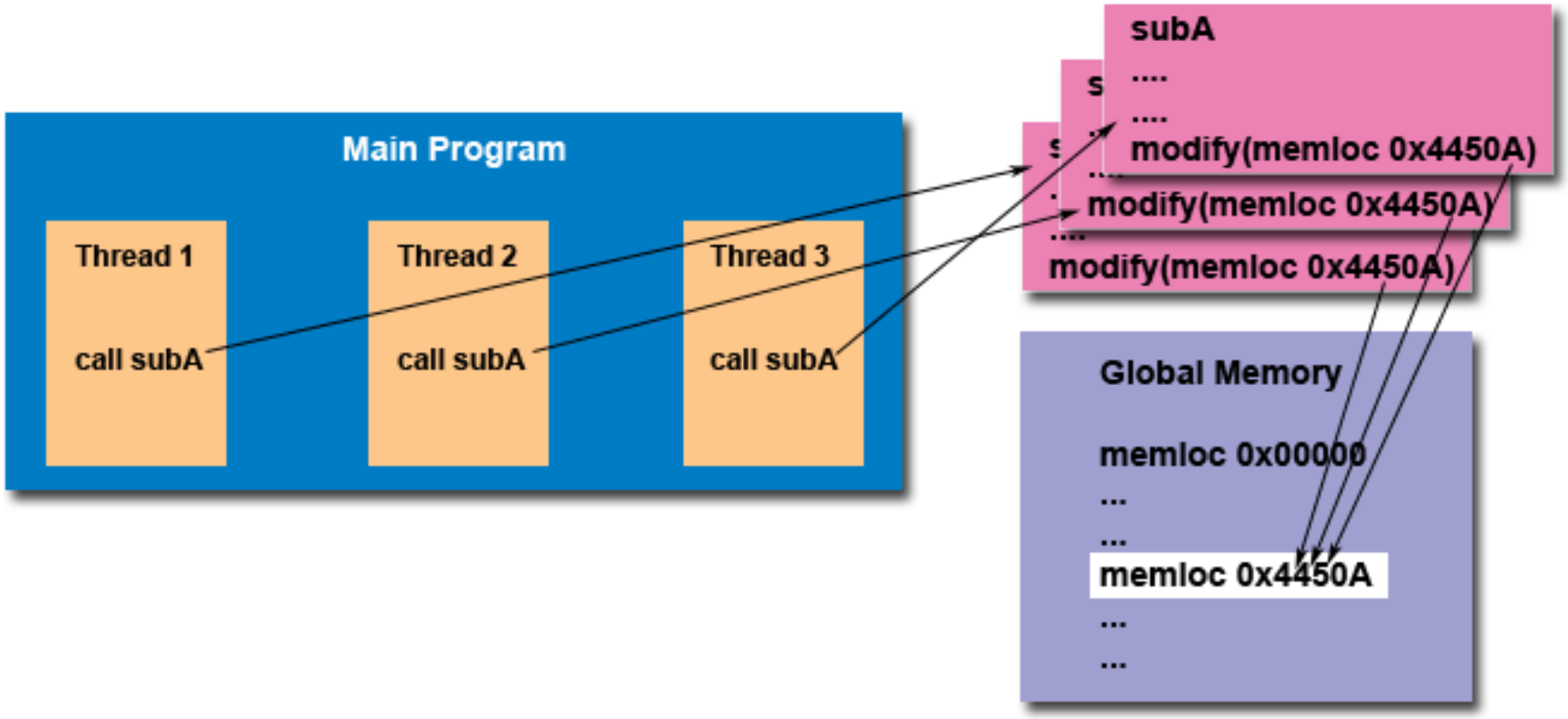
Sapienza Università di Roma

Operating Systems: Thread

Thread Safety 1/2

A code is thread-safe when multiple threads can execute it simultaneously without unintended interactions

- (without *clobbering* shared data)
- (without creating *race conditions*)



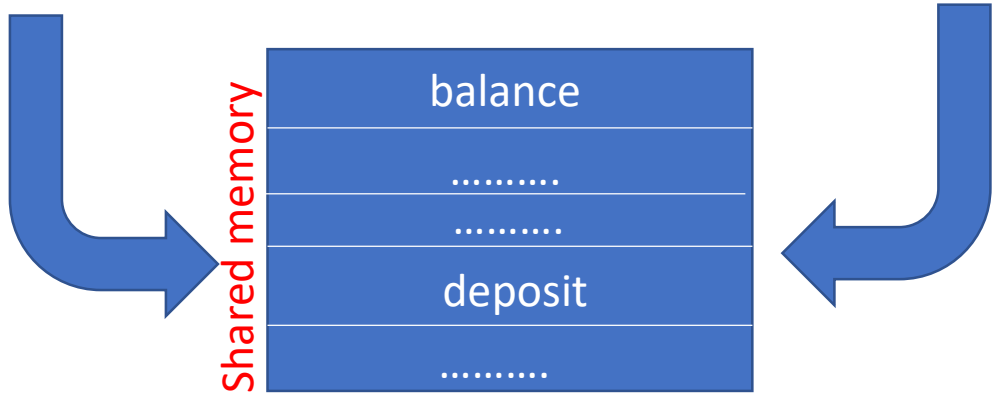
Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Thread

Thread Safety 2/2

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200



Example: an application creates several threads, each of which makes a call to the same library routine:

- The library routine accesses/modifies a global structure or location in memory
- As each thread calls this routine, it is possible that they may try to modify this structure/location at the same time
- If the routine does not employ some sort of *synchronization mechanism* to prevent data corruption, then it is not thread-safe

Concurrency: mutual exclusion and synchronization

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Concorrenza

Multiple Processes

Operating System design is concerned with the management of processes and threads:

- **Multiprogramming**: The management of multiple processes within a uniprocessor system
- **Multiprocessing**: The management of multiple processes within a multiprocessor
- **Distributed Processing**: The management of multiple processes executing on multiple, distributed computer systems. The recent proliferation of clusters is a prime example of this type of system

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

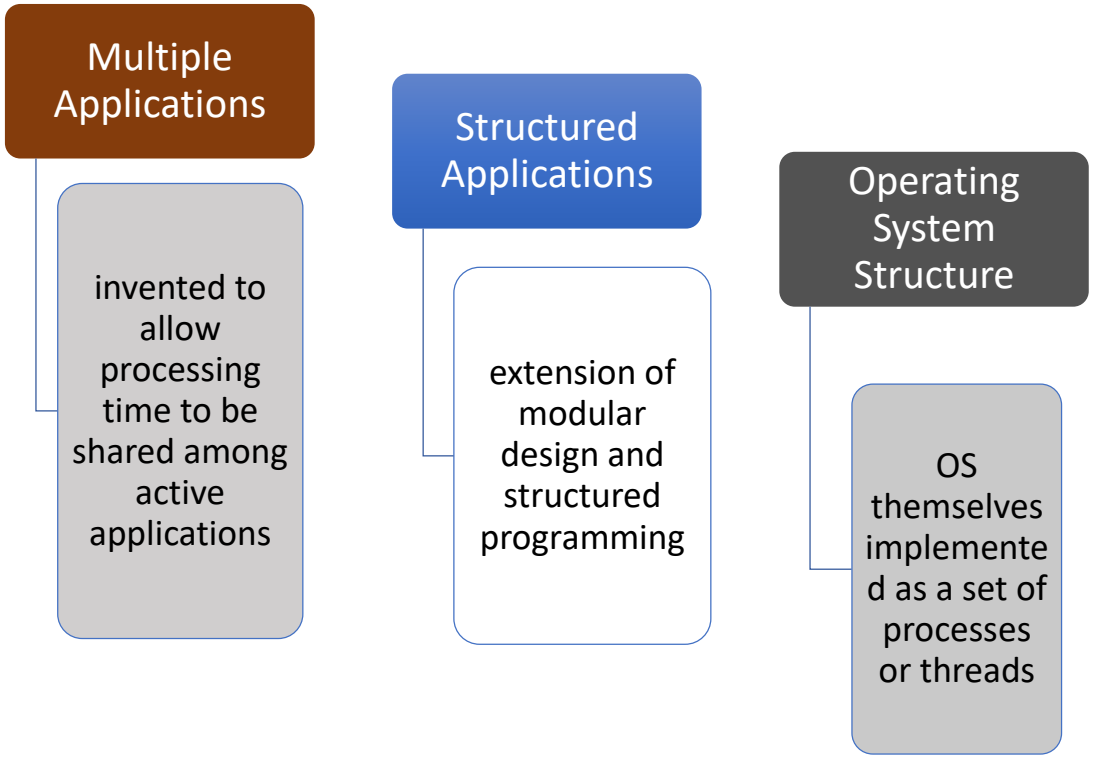
Sapienza Università di Roma

Operating Systems: Concorrenza

I 3 contesti in cui si presenta

Operating System design is concerned with the management of processes and threads:

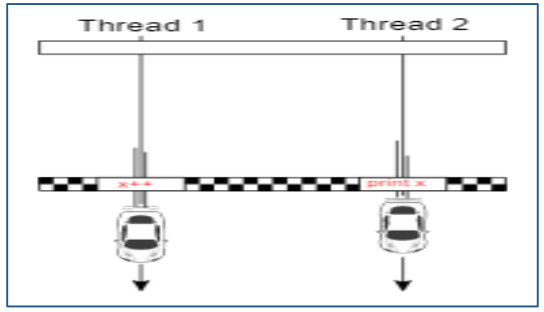
- **Multiple applications**: Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications.
- **Structured applications**: As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes.
- **Operating system structure**: The same structuring advantages apply to systems programs, and we have seen that operating systems are themselves often implemented as a set of processes or threads.

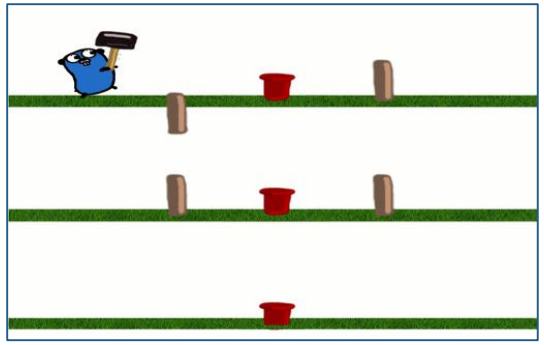


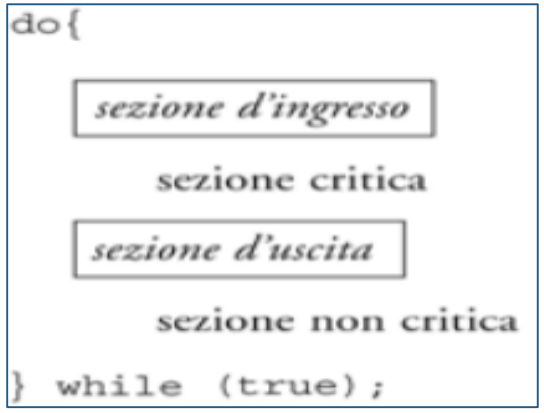
DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: Concorrenza
Definizioni







In un Sistema multiprogrammato, multitasking, multithread

- **Race Condition**: (Corsa Critica): situazione in cui il risultato della elaborazione su una risorsa condivisa da più processi, di cui almeno uno vi scrive, dipende da come essi si alternano (run-time: non predicibile, in funzione della schedulazione, degli eventi esterni, degli interrupt, etc).

- **Mutual Exclusion**: (Mutual Esclusione): proibire che più di un processo acceda contemporaneamente alla risorsa condivisa

- **Critical Region** (Sezione Critica): parte del codice di un processo in cui avviene l'accesso alla risorsa condivisa. Per impedire l'ingenerarsi di Race Condition devono essere rispettati i seguenti:
  1. Mutual Exclusion: Un solo processo può essere in una regione critica
  2. Velocità Relative: Le assunzioni su velocità e numero di processori sono ininfluenti
  3. Progresso: Nessun processo la cui elaborazione è al di fuori della sezione critica può fermare altri processi
  4. Bounded Waiting: Nessun processo deve aspettare indefinitamente per entrare nella sua regione critica

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI
SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: Concorrenza

Termini Chiave 1/2

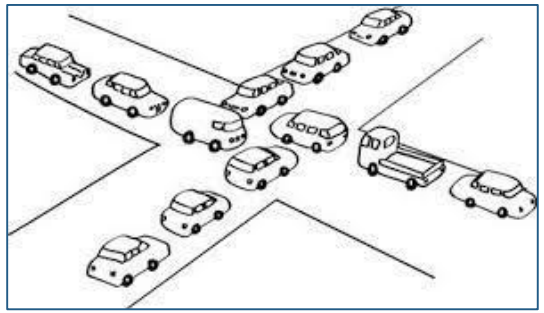
Race Condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution. The "loser" of the race is the process that updates last and will determine the final value of the variable. 
Mutual Exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources
Critical Section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.

Output of process must be independent of the speed of execution of other concurrent processes

Atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes

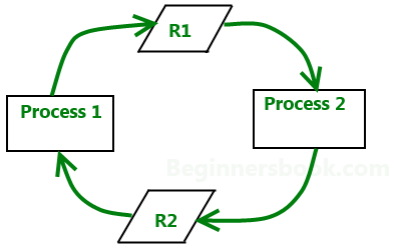
Operating Systems: Concorrenza

Critical Region: Definizioni



In un Sistema multiprogrammato, multitasking, multithread

- **Deadlock**: (Stallo): situazione in cui due o più processi non possono procedere con la prossima istruzione, perché ciascuno attende l'altro.



Process P1 holds resource R2 and requires R1 while
Process P2 holds resource R1 and requires R2.

- **LiveLock**: (Stallo Attivo): situazione in cui due o più processi cambiano continuamente il proprio stato, l'uno in risposta all'altro, senza fare alcunché di "utile"

- **Starvation** (Inedia): un processo, pur essendo ready, non viene mai scelto dallo scheduler

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Concorrenza

Termini Chiave 2/2

Deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
Livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work
Starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Concorrenza

Sezione Critica e Sincronizzazione

Le tematiche inerenti la concorrenza che il SO deve risolvere sono:

- **How**: formato e modalità di scambio informazioni fra processi

➔ vedi “Inter Process Communication” (IPC)

- **Where**: garantire che I processi non si intralcino a vicenda (Race Condition - Critical Region)

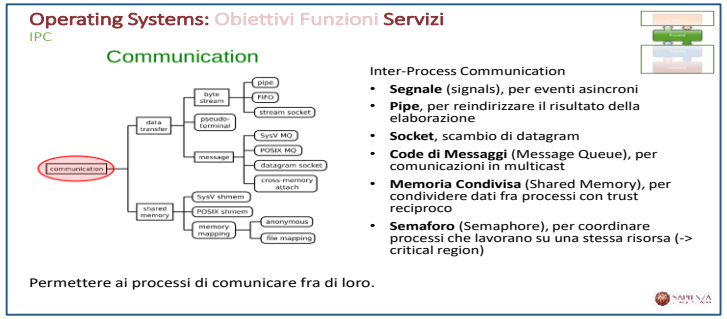
➔ **Mutual Exclusion** (without Deadlock|Livelock, Starvation)

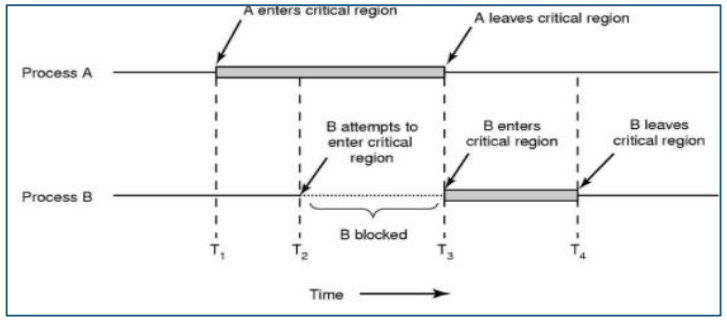
- **When**: sequenziamento corretto qualora vi sia dipendenza (es. Producer-Consumer: Bounded Buffer), **indipendente** da **interruzioni**, **context-switch** (multi-programming, multi-threading) e **velocità di elaborazione** (SMP: Symmetric Multi Processor, CPU throttling)

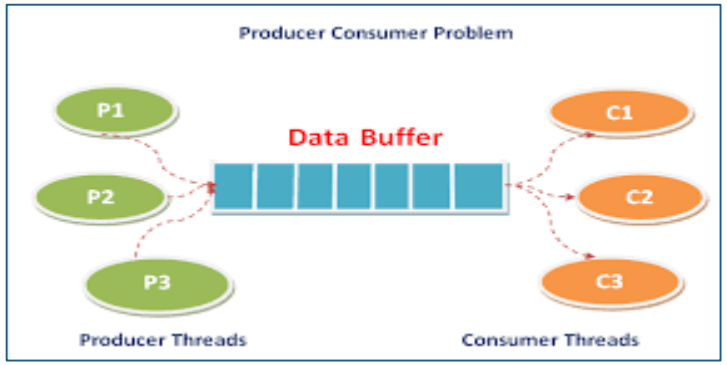
➔ **Turn Variable**: solo per 2 processi concorrenti

➔ Hardware Support: Interrupt Disabling | TSL (XCHG)

➔ Sleep Wake-Up: sincronizzazione con lo scheduler







DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: Concorrenza

Operating System Concerns

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

- **Mutual Exclusion** (without Deadlock|Livelock, Starvation)
- **Turn Variable**: solo per 2 processi concorrenti
- Hardware Support: Interrupt Disabling | TSL (XCHG)

- **Mutual Exclusion** (without Deadlock|Livelock, Starvation)
- Sleep Wake-Up: sincronizzazione con lo scheduler

- Sleep Wake-Up: sincronizzazione con lo scheduler

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

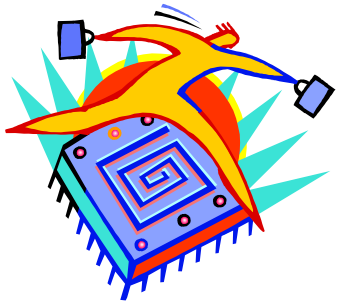
Operating Systems: Concorrenza

Principii e Difficoltà

Principii

Interleaving and overlapping

- can be viewed as examples of concurrent processing
- both present the same problems



[Uniprocessor] The relative speed of execution of processes cannot be predicted; depends on:

- activities of other processes
- the way the OS handles interrupts
- scheduling policies of the OS

Difficoltà

- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Concorrenza

Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
  - for example: I/O devices, memory, processor time, clock

In the case of competing processes, three control problems must be faced:

- **the need for mutual exclusion**
- **deadlock**
- **starvation**

Concurrency: Mutual Exclusion

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Concorrenza

Mutua Esclusione: Schema di Dijkstra (1965)

Algoritmo per la gestione della mutua esclusione nell'accesso ad una risorsa condivisa (sezione critica).

Sequenza di blocchi del programma:

```
NCS (non in sezione critica)
<trying protocol>
CS (sezione critica)
<exit protocol>
NCS (non in sezione critica)
```

In un sistema concorrente esiste uno "scheduler" centralizzato che permette ad un solo processo alla volta di entrare in esecuzione e quindi di evolvere secondo il suo codice.
Lo scheduler esegue una **linearizzazione** di tutte le istruzioni elementari effettuate dai vari processi. La linearizzazione creata dalla singola esecuzione dell'algoritmo è chiamata "schedule".

Assunzioni:

- i processi comunicano leggendo e scrivendo variabili condivise
- la lettura e la scrittura di una variabile è una azione atomica

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Concorrenza

Mutua Esclusione: Fasi e Condizioni di gestione con Flag e Turni

Fasi del Processo coinvolto in una mutua esclusione

Process Elaboration Type
Process Flag: Critical Status
Process Turn

NCS	➔ TRY	➔ CS	➔ EXIT	➔ RS
IDLE	WAIT	ACTIVE	ACTIVE	IDLE
?	NO	YES	YES	?

- **NCS**: Non Critical Section
- **TRY**: Test di verifica delle condizioni di ingresso
- **CS**: Critical Section
- **EXIT**: configurazione delle condizioni per gli altri processi
- **RS**: Remainder Section

- IDLE: Non Interessato
- WAIT: Interessato ma impossibilitato
- ACTIVE: Interessato, in Critical Section

Condizioni:

- **Mutual Exclusion**: solo un processo in **CS** od **EXIT** ➔ solo 1 processo con (ACTIVE, YES)
- **Progress**: processi in **NCS** ed **RS** il più possibile ➔ molti processi con (IDLE, ?)
- **Bounded Waiting**: processi in **TRY** il meno possibile ➔ pochi processi con (WAIT, NO)

Operating Systems: Mutual Exclusion

Illustrazione

Control of competition inevitably **involves the OS** because it is the OS that allocates resources. In addition, the **processes themselves will need to be able to express the requirement for mutual exclusion in some fashion, such as locking a resource prior to its use**. Any solution will involve some support from the OS, such as the provision of the locking facility.

```
          PROCESS 1 */

void P1
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

```
        /* PROCESS 2 */

void P2
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

• • •

```
        /* PROCESS n */

void Pn
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

The figure illustrates the mutual exclusion mechanism in abstract terms. There are *n processes to be executed concurrently.* Each process includes (1) a critical section that operates on some resource Ra, and (2) additional code preceding and following the critical section that does not involve access to Ra. Because all processes access the same resource Ra, it is desired that only one process at a time be in its critical section. To enforce mutual exclusion, two functions are provided: entercritical and exitcritical . Each function takes as an argument the name of the resource that is the subject of competition. Any process that attempts to enter its critical section while another process is in its critical section, for the same resource, is made to wait.

It remains to examine specific mechanisms for providing the functions entercritical and exitcritical . For the moment, we defer this issue while we consider the other cases of process interaction.

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Mutual Exclusion

Requisiti

1. Must be **enforced**
2. A process that halts in its noncritical section must do so without interfering with other processes
3. No deadlock or starvation
4. A process must not be denied access to a critical section when there is no other process using it
5. No assumptions are made about relative process speeds or number of processes
6. A process remains inside its critical section for a *finite time* only

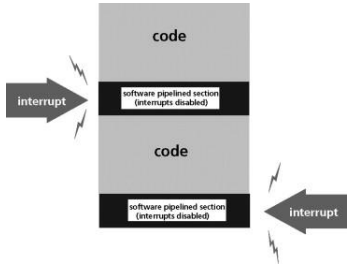
Operating Systems: Mutual Exclusion

Mutual Exclusion with Busy Waiting

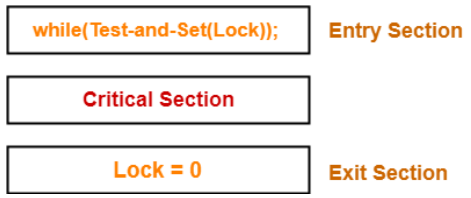
➔ **Busy Waiting**: i processi fuori dalla sezione critica continuano a testare la variabile fino a che lock non ridiventi 0;
➔ **Check Asincrono**: lo Scheduler testa lo stato «Pronto» che non rispecchia anche la usabilità della Sezione Critica

- **Turn Variable**: (Variabile di Blocco): stato inerente la esecuzione di sezioni critiche. Variabile condivisa che definisce lo stato (0: risorsa libera; 1: risorsa in uso) di uso di una risorsa: segnala se è in uso da parte di un processo (nella sua sezione critica).
  ➔ **Strict Alternation**: l'utilizzo di una sola variabile permette di descrivere solo 2 stati (processo A in Critical Region – processo B in Critical Region). Esiste anche l'evenienza che nessuno sia in critical region;

- **Interrupt Disabling**: (Disabilitazione delle Interruzioni): il processo fa una esplicita richiesta di disabilitazione di tutte le interruzioni, durante la esecuzione della sezione critica, in modo da non incorrere in inconsistenza di dati sulla risorsa condivisa.
  ➔ Non funziona sui multi-core perché la disabilitazione degli interrupt ha effetto solo su una CPU alla volta:



- **Special HW Instruction**: (istruzioni speciali implementate nell'hardware): per assicurare l'atomicità nella esecuzione assembly della istruzione, questa viene implementata in.
  ➔ Necessaria conoscenza dell'hardware sottostante;
  ➔ Ridotta portabilità del codice.



DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Mutual Exclusion

Mutual Exclusion with Busy Waiting: Turn Variable 1/2

- **Turn Variable** (Variabile di Turno): definisce il turno di uso della risorsa. Annovera anche la situazione in cui nessun processo è interessato alla sezione critica. Da usarsi assieme alla variabile lock. Inizialmente sviluppato da Dekker e documentato da Dijkstra nel 1965, è stato ulteriormente ottimizzato da Peterson nel 1981.
  - ➔ **Busy Waiting**: i processi interessati ma alla sezione critica ma al di fuori di essa continuano a testare la variabile fino a che lock non ridiventi 0;

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

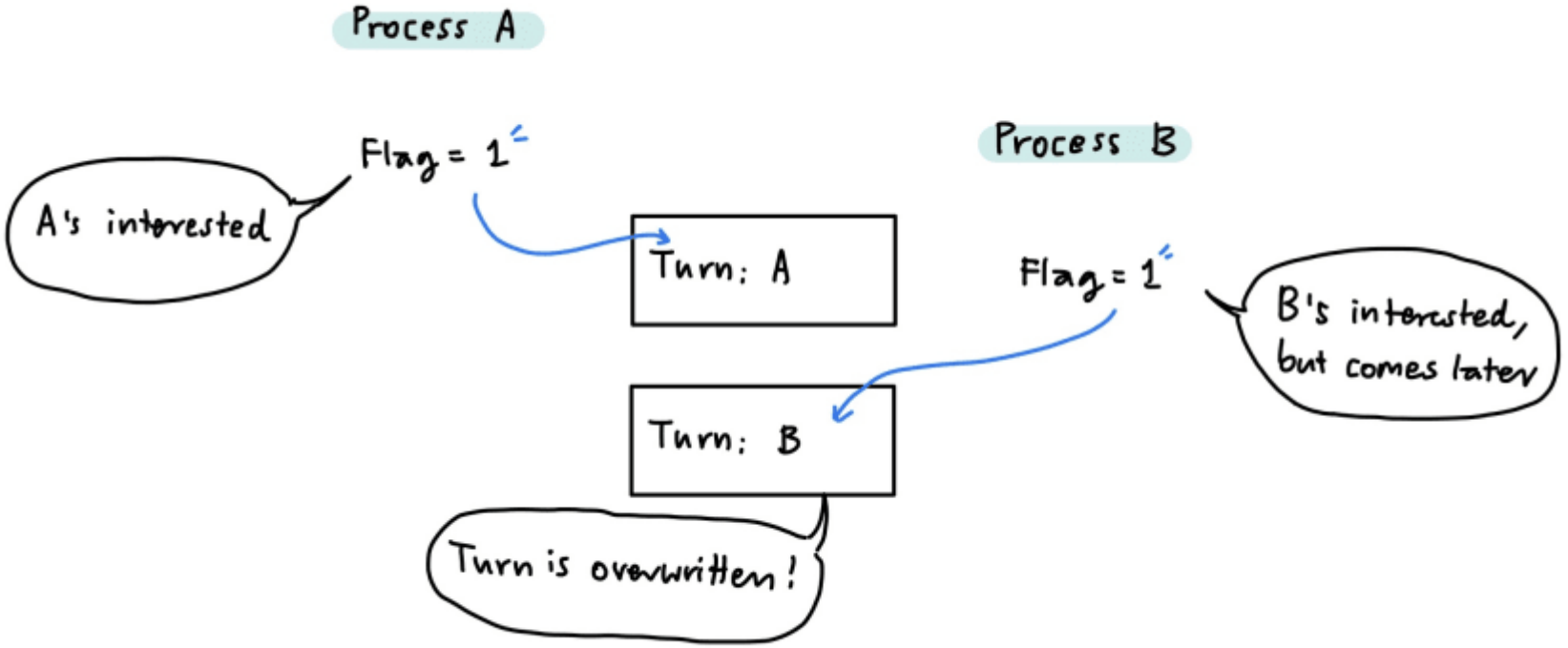
Figure 2-24. Peterson's solution for achieving mutual exclusion.

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Mutual Exclusion

Mutual Exclusion with Busy Waiting: Turn Variable 2/2



Turn Variable L'algoritmo controllerà

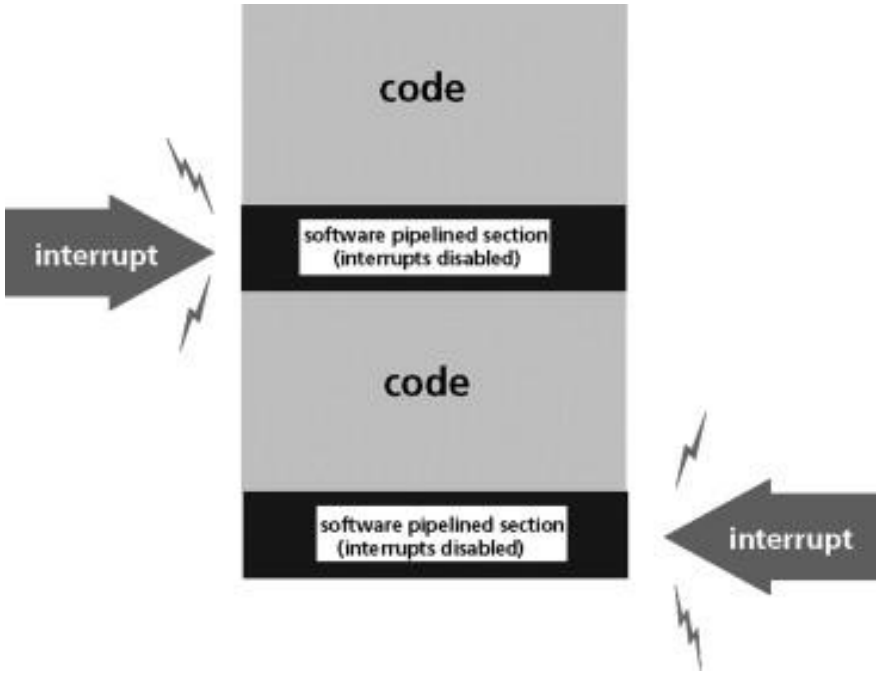
1. prima se i flag di entrambi i processi A e B sono 1 per vedere se sono entrambi interessati.
2. di chi è il turno.
3. Se sono entrambi interessati e il turno è del processo B, allora il processo A può entrare nella regione critica per primo

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: Mutual Exclusion

Mutual Exclusion with Busy Waiting: HW: Interrupt Disabling



- **Advantages**
  - **uniprocessor** system only
  - disabling interrupts guarantees mutual exclusion

- **Disadvantages:**
  - the efficiency of execution could be noticeably degraded
  - this approach will **not work** in a **multiprocessor** architecture
  - **Safely** operated only by **kernel** (**dangerous** at **user-space**)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Mutual Exclusion: Hardware Support

```
int compare_and_swap(int* reg, int oldval, int newval)
{
  ATOMIC();
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  END_ATOMIC();
  return old_reg_val;
}
```

Hardware Support for Mutual Exclusion

```
int compare_and_swap(int* reg, int oldval, int newval)
{
  ATOMIC();
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  END_ATOMIC();
  return old_reg_val;
}
```

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
  while (true) {
    while (compare_and_swap(&bolt, 0, 1) == 1)
        /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . . ,P(n));

}
```

(a) Compare and swap instruction

Operating Systems: Mutual Exclusion

Mutual Exclusion with Busy Waiting 3/3

- **TSL** (Test and Set Lock): operazione in **HW**, definita nella CPU. Mette il contenuto di [lock] nel registro ed inserisce un valore non nullo (es. PID) in [lock]
  - ➔ **Busy Waiting**: i processi interessati ma alla sezione critica ma al di fuori di essa continuano a testare la variabile fino a che lock non ridiventi 0;

```
enter_region:
        TSL REGISTER,LOCK     | copy lock to register and set lock to 1
        CMP REGISTER,#0       | was lock zero?
        JNE enter_region      | if it was nonzero, lock was set, so loop
        RET                   | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0          | store a 0 in lock
        RET                   | return to caller
```

- **Atomic Operation** (Operazione Atomica): sequenza indivisibile di comandi. Nessun altro processo può vedere uno stato intermedio della sequenza o interrompere la sequenza. Le operazioni HW sono, ovviamente, atomiche.

Operating Systems: Mutual Exclusion

Mutual Exclusion with Busy Waiting: XCHG and CMPXCHG instructions on Intel

- **XCHG** (per x86) `XCHG EAX, [lock]`. Mette il contenuto di [lock] nel registro EAX ed inserisce un valore non nullo (es. PID) in [lock]

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        CMP REGISTER,#0           | was lock zero?
        JNE enter_region          | if it was non zero, lock was set, so loop
        RET                       | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0              | store a 0 in lock
        RET                       | return to caller
```

- **CMPXCHG** (per 486 ed oltre) `CMPXCHG Dest, Src`. Se Dest = [AL, AX, EAX], mette il contenuto di Src in Dest. Altrimenti mette il contenuto di Dest in [AL, AX, EAX]. Effettua le 2 istruzioni XCHG e CMP nello stesso tempo (1 delle 6 istruzioni annoverate nel 80486 e non nel 80386).

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Mutual Exclusion: Hardware Support

- Exchange instruction

```
void exchange (int *register, int *memory)
{
   int temp;
   temp = *memory;
   *memory = *register;
   *register = temp;
}
```

Operating Systems: Mutual Exclusion

Mutual Exclusion with Busy Waiting: exchange instruction

Exchange instruction

- exchange the content of a register with that of a memory location
- Implemented in Pentium and Itanium (IA64) architectures

- Exchange instruction

```
void exchange (int *register, int *memory)
{
  int temp;
  temp = *memory;
  *memory = *register;
  *register = temp;
}
```

```
/* program mutualexclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
  while (true) {
    int keyi = 1;
    do exchange (&keyi, &bolt) while (keyi != 0);
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . ., P(n));
}
```


DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: Mutual Exclusion

Mutual Exclusion with Busy Waiting: Compare&Swap instruction

Compare&Swap Instruction

- also called a “compare and exchange instruction”
- a **compare** is made between a memory value and a test value
- if the values are the same a **swap** with the new value occurs in memory
- carried out atomically
- Available in x86, IA64, sparc, IBM architectures

```
int compare_and_swap(int* reg, int oldval, int newval)
{
  ATOMIC();
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  END_ATOMIC();
  return old_reg_val;
}
```

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
  while (true) {
    while (compare_and_swap(&bolt, 0, 1) == 1)
        /* do nothing */;
    /* critical section */;
    bolt = 0;
    /* remainder */;
  }
}
void main()
{
  bolt = 0;
  parbegin (P(1), P(2), . . . ,P(n));

}
```


Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Mutual Exclusion

Hardware Support: Special Machine Instruction recap

- **Advantages**
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - Simple and easy to verify
  - It can be used to support multiple critical sections; each critical section can be defined by its own variable

- **Disadvantages:**
  - **Busy-waiting** is employed: while a process is waiting for access to a critical section it continues to consume processor time
  - **Starvation** is *possible* when a process leaves a critical section and more than one process is waiting
  - **Deadlock** is *possible* (e.g., because of process priority)
  - Not available in any architecture

    Developer must be aware of the architecture

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: IPC

Mutual Exclusion with Sleep and Wakeup 1/4

➔ **UnBusy Freeze**: i processi fuori dalla sezione critica, ma interessati ad essa, non sono in stato «Pronto»;

➔ **Check Sincrono**: lo stato «Pronto» rispecchia anche la usabilità della Sezione Critica

Busy Waiting: fa sprecare tempo di CPU prezioso

➔ **Blocking Calls**: liberare la CPU per altri processi. Magari proprio quello che deve liberare la risorsa

- **Sleep** blocks process
- **Wakeup** unblocks process

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Concurrency: Semafori

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Semafori

Meccanismi Comuni 1/2

Semaphore	A construct based on an integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: *initialize*, *decrement*, and *increment*. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process
Binary Semaphore	A semaphore that takes on only the values 0 and 1
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1)
Condition Variable	A data type that is used to block a process or thread until a particular condition is true

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

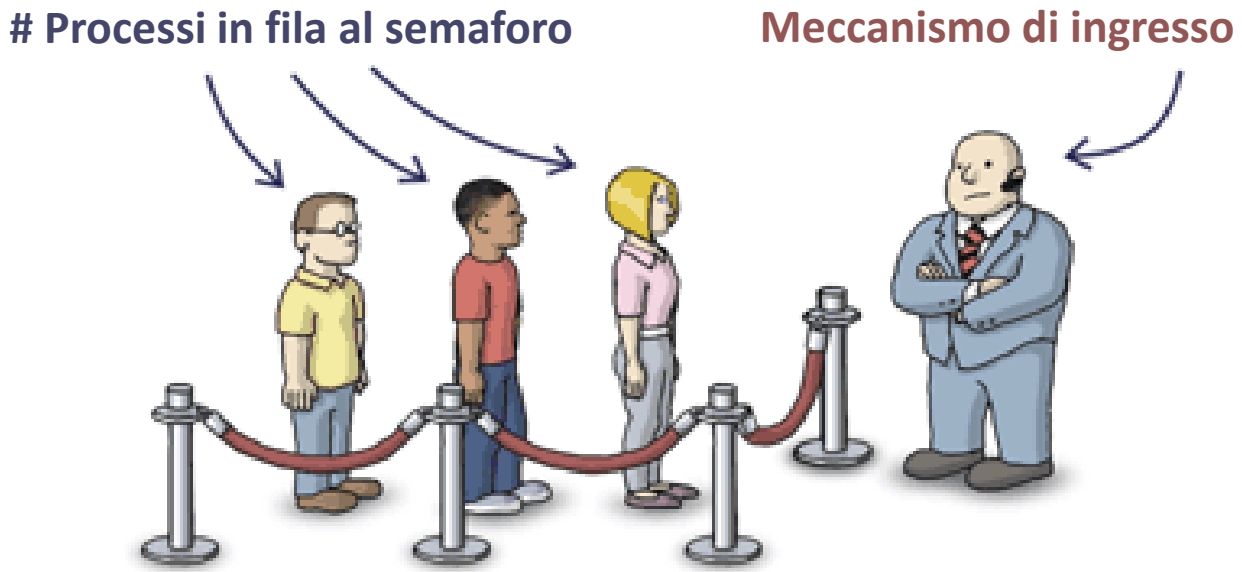
Operating Systems: Semafori

Up/Down: Semafori 1/3

Semaphore (**Semaforo**): variabile intera, usata per mettere a dormire i processi (sleep) o svegliarli (wakeup)
Operazioni: sono definite 3 operazioni, le sole ammesse per manipolare il semaforo:

- **init()**: abilitazione del semaforo (valore non negativo)
- **down()** [`semWait()`]: generalizzazione di sleep(); controlla il semaforo. Se non è zero, decrementa il semaforo. Se zero, il processo va a dormire;
- **up()** [`semSignal()`]: generalizzazione di wakeup(); controlla il semaforo. Se non è zero, incrementa il semaforo. Se zero, il processo va a dormire.

➔ **Operazioni Atomiche**: garantisce che, una volta che sono partite le operazioni sul semaforo, nessun altro processo po' accedere fino a che l'operazione non sia stata completata (o bloccata).



Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti
Sapienza Università di Roma

Operating Systems: Semafori

Up/Down: Semafori 2/3


```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                          /* TRUE is the constant 1 */
                item = produce_item( );         /* generate something to put in buffer */
                down(&empty);                   /* decrement empty count */
                down(&mutex);                   /* enter critical region */
                insert_item(item);              /* put new item in buffer */
                up(&mutex);                     /* leave critical region */
                up(&full);                      /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* infinite loop */
                down(&full);                    /* decrement full count */
                down(&mutex);                   /* enter critical region */
                item = remove_item( );          /* take item from buffer */
                up(&mutex);                     /* leave critical region */
                up(&empty);                     /* increment count of empty slots */
                consume_item(item);             /* do something with the item */
        }
}
```

Produttore/Consumatore: sono definiti 3 semafori.

- **full**: conteggia gli slot che sono riempiti; inizialmente full == 0;
- **empty**: conteggia gli slot che sono vuoti; inizialmente empty == N;
- **mutex**: protegge la variabile che contiene gli item prodotti e consumati.

Operazioni Atomiche: implementate tramite System Call

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Semafori

Up/Down: Semafori 3/3


```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                          /* TRUE is the constant 1 */
                item = produce_item( );         /* generate something to put in buffer */
                down(&empty);                   /* decrement empty count */
                down(&mutex);                   /* enter critical region */
                insert_item(item);              /* put new item in buffer */
                up(&mutex);                     /* leave critical region */
                up(&full);                      /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* infinite loop */
                down(&full);                    /* decrement full count */
                down(&mutex);                   /* enter critical region */
                item = remove_item( );          /* take item from buffer */
                up(&mutex);                     /* leave critical region */
                up(&empty);                     /* increment count of empty slots */
                consume_item(item);             /* do something with the item */
        }
}
```

Produttore/Consumatore: sono definiti 3 semafori.

- **full**: garantisce che non vengano effettuate scritture a buffer pieno (sovrascritture);
- **empty**: garantisce che non siano effettuate letture su porzioni del buffer che non contengono dati prodotti;
- **mutex**: usato per mutua esclusione. Garantisce che solo un processo alla volta possa leggere/scrivere sul buffer

Multiprocessori: ogni semaforo, inoltre, è protetto anche da una istruzione TSL, di modo che solo un processore alla volta possa accedervi (sperabilmente, durante il Busy-Waiting il semaforo diventa libero).

Operating Systems: Concorrenza

Semafori


```
semaphore S = 1;
    int X = N;

    [Process 1]                    [Process 2]
        int Y;                         int Z;
        semWait(S);                    semWait(S);
        Y = X*2;                       Z = X+1;
        X = Y;                         X = Z;
        semSignal(S);                  semSignal(S);
```

- Sequenza Process1 – Process 2: X = 2N + 1
- Sequenza Process2 – Process1: X = 2(N+1)

SemWait (down): fondamentale per difendere la mutua esclusione sulla scrittura di X. Senza di esso, a run-time, potrebbe succedere che il valore scritto da uno venga sovrascritto dall'altro, portando a valori non previsti nella corretta computazione:
X = N+1
X = 2 N

SemSignal (up): fondamentale per uscire dalla mutua esclusione consentendo anche all'altro processo di entrarvi.
Senza di esso ➔ Starvation

Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Semafori

Considerazioni

There is no way to know before a process decrements a semaphore whether it will block or not

There is no way, on a uniprocessor system, to know which process will continue immediately after a semSignal when two processes are running concurrently

You don’t know whether another process is waiting so the number of unblocked processes may be zero or one

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Semafori

Definizione della Primitiva per semaforo a interi


```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

formal definition of the primitives for semaphores. The semWait and semSignal primitives are assumed to be atomic.

Operating Systems: Semafori

Definizione della Primitiva per semaforo binario


```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
            s.value = zero;
      else {
               /* place this process in s.queue */;
               /* block this process */;
      }
}
void semSignalB(semaphore s)
{
      if (s.queue is empty())
            s.value = one;
      else {
               /* remove a process P from s.queue */;
               /* place process P on ready list */;
      }
}
```

A binary semaphore may only take on the values 0 and 1 and can be defined by the following three operations:

In principle, it should be easier to implement the binary semaphore, and it can be shown that it has the same expressive power as the general semaphore (see Problem 5.16). To contrast the two types of semaphores, the nonbinary semaphore is often referred to as either a counting semaphore or a general semaphore .

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Semafori

Strong/Weak Semaphores

A **queue** is used to hold processes waiting on the semaphore

Strong Semaphores

- the process that has been blocked the longest is released from the queue first (FIFO)

Weak Semaphores

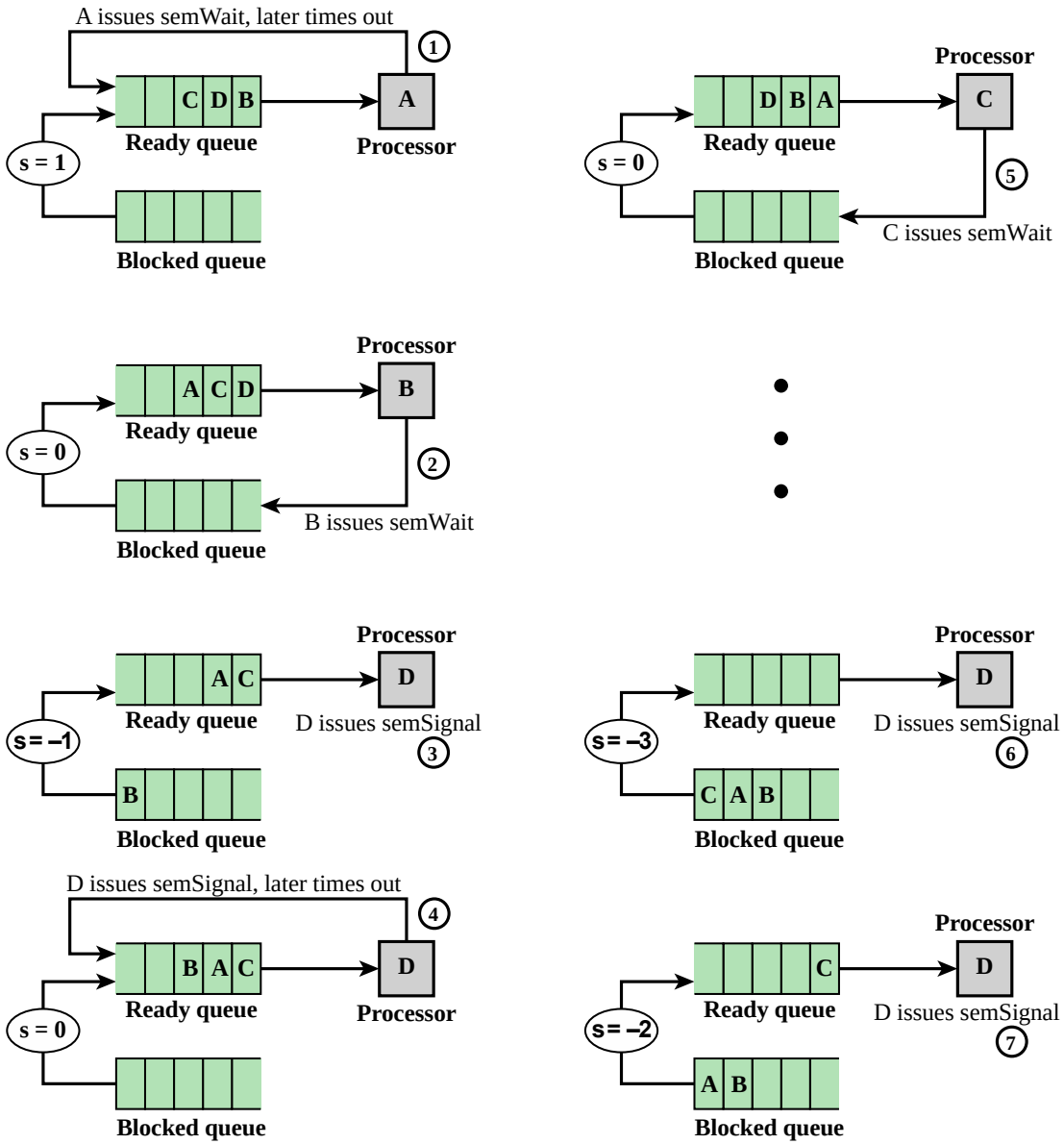
- the order in which processes are removed from the queue is not specified

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Semafori

Strong/Weak Semaphores: implementation



Here processes **A**, **B**, and **C depend on** a result from process **D**.

1) Initially (1), A is running; **B, C, and D are ready**; and the semaphore count is 1, indicating that one of D's results is available. When A issues a semWait instruction on semaphore *s,* the semaphore decrements to 0, and A can continue to execute; subsequently it rejoins the ready queue.
2) Then B runs (2), eventually issues a semWait instruction, and is blocked,
3) allowing D to run (3). When D completes a new result, it issues a semSignal instruction,
4) which allows B to move to the ready queue (4).
5) D rejoins the ready queue and C begins to run (5) but is blocked when it issues a semWait instruction.
6) Similarly, A and B run and are blocked on the semaphore, allowing D to resume execution (6).
7) When D has a result, it issues a semSignal , which transfers C to the ready queue.
8) Later cycles of D will release A and B from the Blocked state.

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Semafori

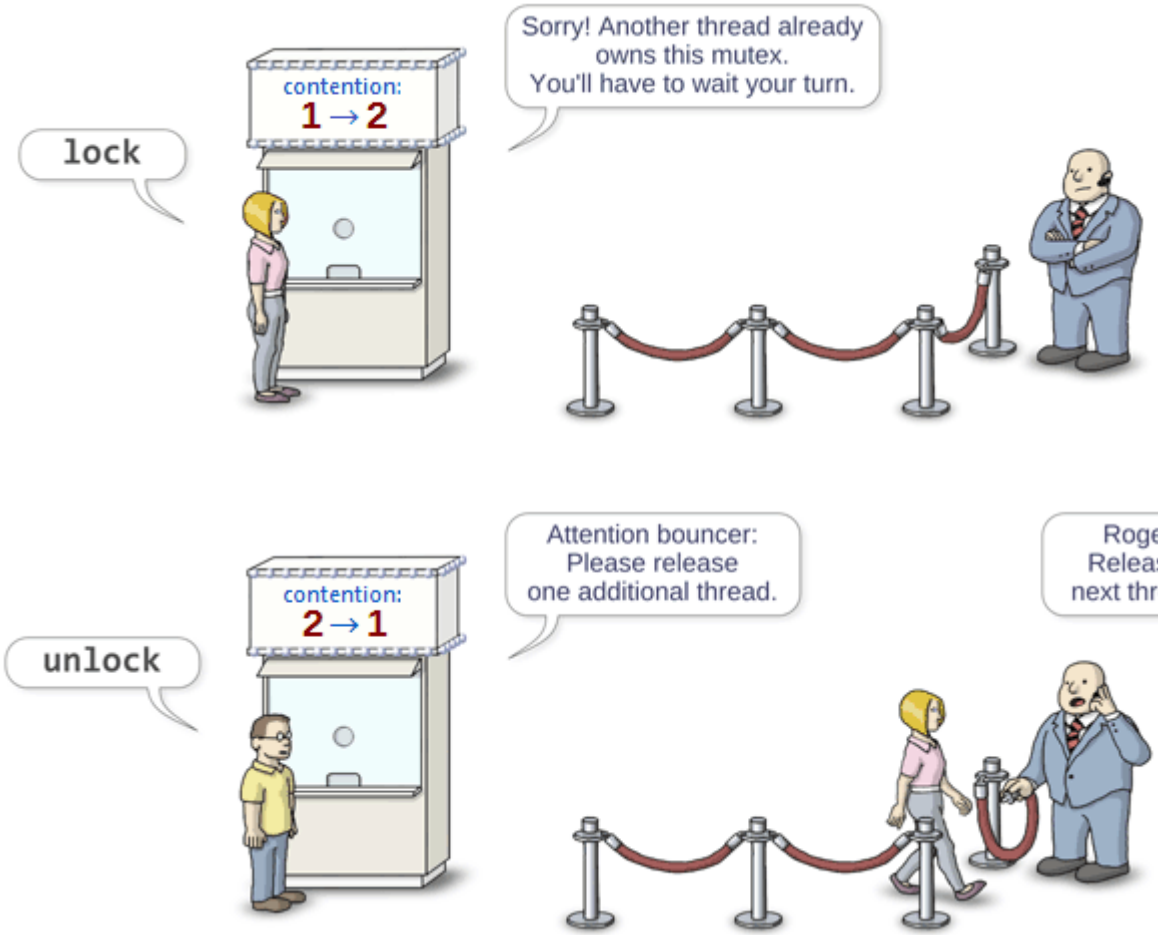
MutEx 1/2

No-Count: non è necessario effettuare un conteggio degli spazi riempiti/vuoti

2 Processi: solo 2 processi condividono la risorsa

➔ **Mutual Exclusion**: variabile binaria. Versione semplificata del semaforo. Solo 2 stati:

- **Locked**
- **Unlocked**



DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: Semafori

MutEx 2/2


```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Simile al TSL ma la funzione thread_yeald() mette il processo in sleep. Istruzioni eseguibili in user-space

➔ no System Call

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Semafori

Mutual Exclusion Using Semaphores

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

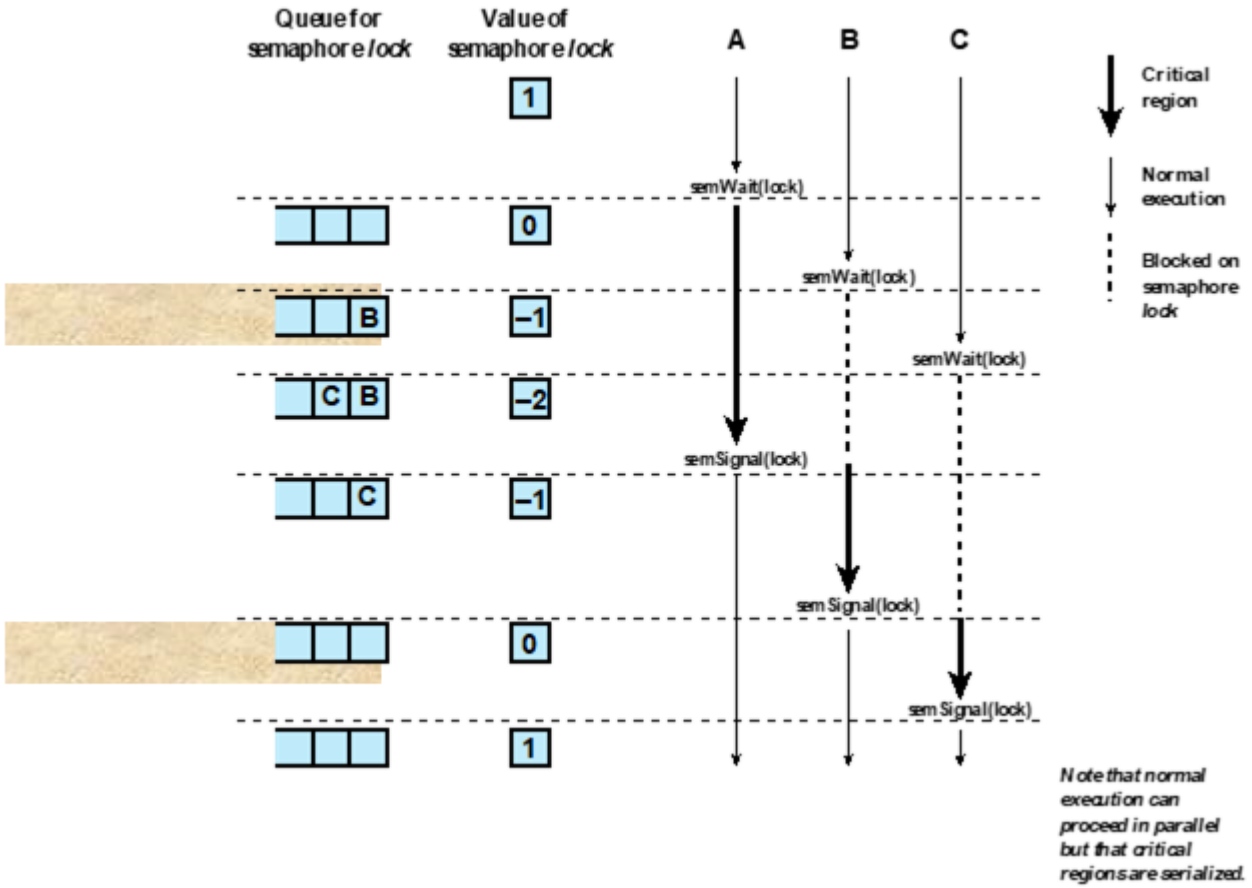
For the mutual exclusion, **strong semaphores** guarantee **freedom** from **starvation**, while **weak semaphores** do **not**.
We will assume **strong semaphores** because they are **more convenient** and because this is the **form** of semaphore **typically provided** by operating systems.

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Semafori

Accessing shared data protected by a Semaphore: example with 3 processes



3 processes (A, B, C) access a shared resource protected by the semaphore lock .
Process A executes semWait (lock) ; because the semaphore has a value of 1 at the time of the semWait operation, A can immediately enter its critical section and the semaphore takes on the value 0.
While A is in its critical section, both B and C perform a semWait() operation and are blocked pending the availability of the semaphore.
When A exits its critical section and performs semSignal (lock) , B, which was the first process in the queue, can now enter its critical section.

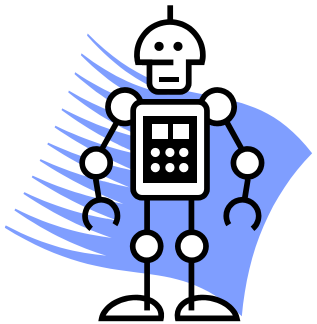
Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Semafori

Implementation of Semaphores

- Imperative that the `semWait` and `semSignal` operations be implemented as atomic primitives
  - Manipulating a semaphore is a mutual exclusion problem
- Can be implemented in hardware or firmware
- Software schemes such as Dekker's or Peterson's algorithms can be used
- Use one of the hardware-supported schemes for mutual exclusion



Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Semafori

Implementation of Semaphores 1/2: Compare&Swap

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set s.flag to 0) */;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
          /* do nothing */;
      s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
      }
      s.flag = 0;
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Semafori

Implementation of Semaphores 2/2: Interrupts

```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow interrupts */;
   }
   allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Semafori

Posix semaphores (c)

- `int sem_init(sem_t * sem, int pshared, unsigned value); //initialization`
- `int sem_wait(sem_t *sem); //wait`
- `int sem_post(sem_t *sem); //signal`
- `int sem_destroy(sem_t *sem); //destruction`

- Parameters:
  - `sem`: the semaphore
  - `pshared`: 0 if semaphore is shared among threads, 1 if shared among processes
  - `value`: the starting value (how many resources we can share)
- return -1 in case of error, 0 otherwise
  - Set errno to allow identification of the error type

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Semafori

Posix semaphores (c): example

```
...
#include <semaphore.h>

sem_t sem;

void* thread_fun(void* arg){
   sem_wait(&sem);
   //critical section
   sem_post(&sem);
}

int main (){
   sem_init(&sem,0,1);
   pthread_t t1,t2;
   pthread_create(&t1,NULL,thread,NULL);
   pthread_create(&t2,NULL,thread,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   sem_destroy(&sem);
   return 0;
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Terminologia

Condition Variable

Soluzione al problema della mutua esclusione tra processi, gestita con variabile:

```
int bolt = 0;
    void P(int i)
    {
        while (bolt == 1) /* do nothing */;
        bolt = 1;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
```

- Starvation:
- Mutua Exclusion: requisite base
- Deadlock

Hints

- Risorse contemporaneamente necessarie?
- Atomicità del transito in sezione citica?
- Rilascio della risorsa?

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Concurrency: other mechanisms

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Concorrenza – altri meccanismi

Meccanismi Comuni 2/2

Monitor	A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are critical sections. A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes / Messages	A means for two processes to exchange information and that may be used for synchronization.
Barriers	When a process reaches the barrier, it is blocked until all processes have reached the barrier. This allows groups of processes to synchronize.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

Operating Systems: Concorrenza – altri meccanismi

Monitor

Programming language construct that provides equivalent functionality to that of semaphores and is easier to control

Implemented in a number of programming languages, including:

- Concurrent Pascal,
- Pascal-Plus,
- Modula-2,
- Modula-3,
- Java

Has also been implemented as a program library

Software module consisting of one or more procedures, an initialization sequence, and local data

Operating Systems: Concorrenza – altri meccanismi

Monitor Characteristics

1. Local data variables are accessible only by the monitor's procedures and not by any external procedure

2. Process enters monitor by invoking one of its procedures

3. Only one process may be executing in the monitor at a time

1. e 2. sono simili a caratteristiche di oggetti dei linguaggi orientato agli oggetti, dove è facilmente implementabile un monitor come un oggetto con caratteristiche speciali.

3. è necessaria per garantire la mutua esclusione. Una struttura di dati condivisa può essere protetta inserendola in un monitor che fornisce una funzione di esclusione reciproca per l'accesso alla risorsa.

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Concorrenza – altri meccanismi

Event Flags: Synchronization

Achieved by the use of **condition variables** that are contained within the monitor and accessible only within the monitor

Condition variables are operated on by two functions:

- cwait(c): suspend execution of the calling process on condition c
- csignal(c): resume execution of some process blocked after a cwait on the same condition

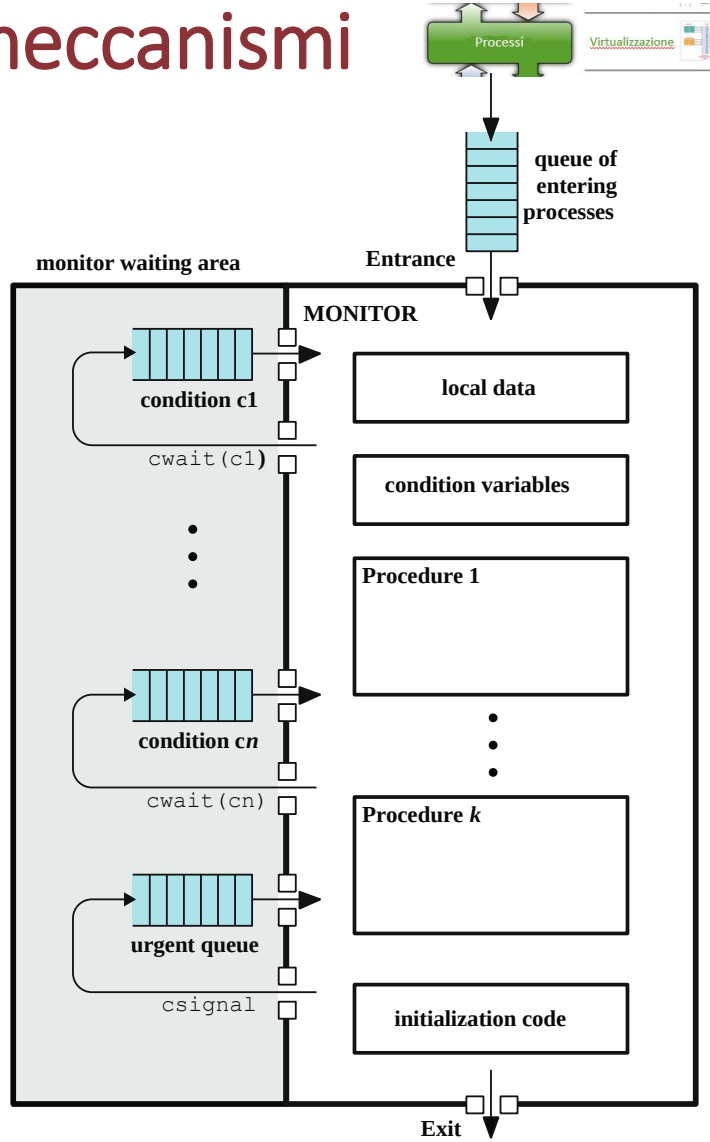


Figure 5.15 Structure of a Monitor

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message. passing

- When processes interact with one another, two fundamental requirements must be satisfied:

synchronization	communication
• to enforce mutual exclusion	• to exchange information

- Message Passing is one approach to providing both of these functions
  - works with distributed systems *and* shared memory multiprocessor and uniprocessor systems

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: Send-Receive; Ack, AuthN

Informazioni scambiate tra processi (anche su macchie differenti)

Funzioni Primitive: sono definite 2 operazioni:

- **Send**(destination, &message); A process sends information in the form of a message to another process designated by a destination

- **Receive**(source,&message). A process receives information by executing the receive primitive, indicating the source and the message

➔ **Problematiche**:

- Acknowledgement: conferma di arrivo messaggio.
- Authentication: identità del sender.

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: Queue

Lo scambio di messaggi è fondamentale nel caso di sistemi distribuiti.

Interazione fra processi: requisiti

- **Mutua Esclusione** ➔ Sincronizzazione
- **Cooperazione** ➔ Scambio di Informazioni

} Code di messaggi

Risorsa consumabile (non riutilizzabile)

Lo scambio di messaggi prevede che:

- **Shared Resources**: non ci siano risorse condivise (per questo viene detta anche shared nothing, nessuna condivisione)
- **Lock**: non siano inclusi meccanismi quali l'uso dei lock o analoghi per ottenere la mutua esclusione poiché l'uso dei soli messaggi permette, ai processi che devono interagire tra loro, di garantirsi la mutua esclusione e di scambiare informazioni per la cooperazione.

In questa tecnica si sfrutta un canale di comunicazione logico sul quale viaggia un messaggio destinato ad un indirizzo.

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: design issues

Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

Format
- Content
- Length
  - fixed
  - variable

Queueing Discipline
- FIFO
- Priority

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: Queue Synchronization

Primitive di Sincronizzazione

Due processi possono interagire fra di loro scambiandosi messaggi, cosa che fanno usando le due primitive:

- send()
- receive()

Fornite dal sistema IPC (Inter Process Communication) presente nel sistema operativo

- send (destinazione, messaggio)
- receive (sorgente, messaggio)

Send: bloccante o non bloccante
Receive: bloccante, non bloccante o con test di messaggio d'arrivo.

Indirizzamento: diretto o indiretto.
Formato di un messaggio: lunghezza fissa o variabile.
Gestione delle code dei messaggi può essere FIFO o a priorità.

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: Synchronization

Communication of a message between two processes implies synchronization between the two

When a receive primitive is executed in a process there are two possibilities:

- if there is no waiting message the process is blocked until a message arrives or the process continues to execute, abandoning the attempt to receive
- if a message has previously been sent the message is received and execution continues

the receiver cannot receive a message until it has been sent by another process

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: Queue Send-Receive

criteri normalmente usati

send		receive
bloccante	rendez-vou	bloccante
non bloccante	Permette ad un processo mittente di mandare messaggi a molti riceventi. E' il caso di un server che deve fornire servizi a molti altri processi	bloccante
non bloccante		non bloccante

In ognuno dei casi ci possono essere problemi:

send non bloccante

- *può consumare risorse (tempo, memoria, ecc...)*
- *il programmatore deve controllare l'arrivo a destinazione del messaggio*

receive bloccante

- *il processo può rimanere bloccato se il messaggio non arriva*

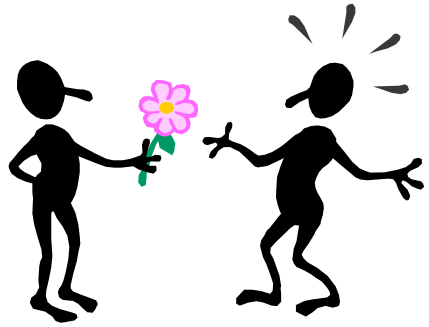
DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: Synchronization - Blocking

- Both sender and receiver are blocked until the message is delivered
- Sometimes referred to as a *rendezvous*
- Allows for tight synchronization between processes



Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: Synchronization – non-Blocking Send

Nonblocking send, blocking receive

- sender continues on but receiver is blocked until the requested message arrives
- most useful combination
- sends one or more messages to a variety of destinations as quickly as possible
- example -- a service process that exists to provide a service or resource to other processes

Nonblocking send, nonblocking receive

- neither party is required to wait

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Concorrenza – altri meccanismi

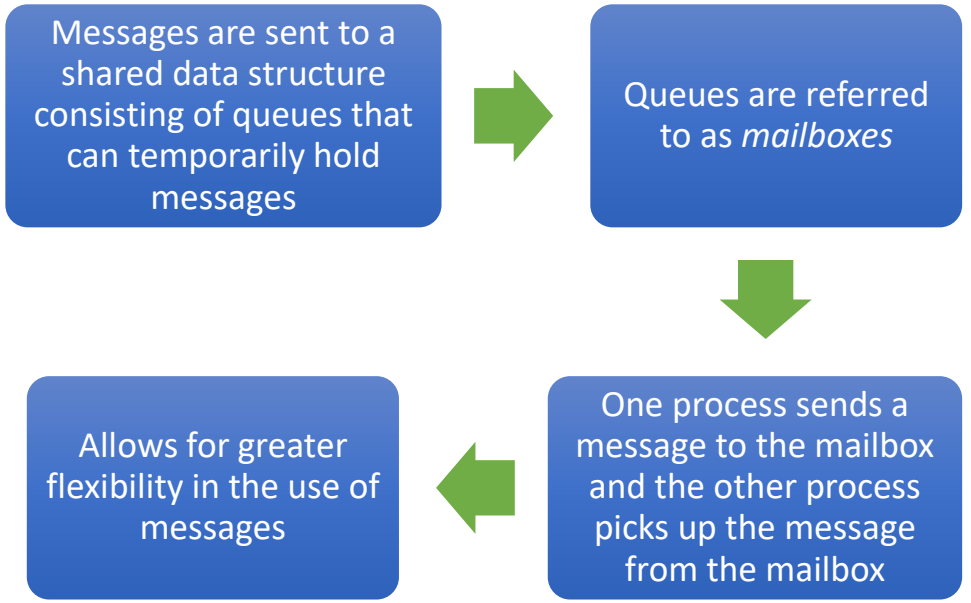
Mailboxes/Message: Addressing

Schemes for specifying processes in `send` and `receive` primitives fall into two categories:

Direct Addressing

- **Send primitive** includes a **specific identifier** of the destination process
- Receive primitive can be handled in one of two ways:
  - require that the process explicitly designate a sending process → effective for cooperating concurrent processes
  - implicit addressing → source parameter of the receive primitive possesses a value returned when the receive operation has been performed

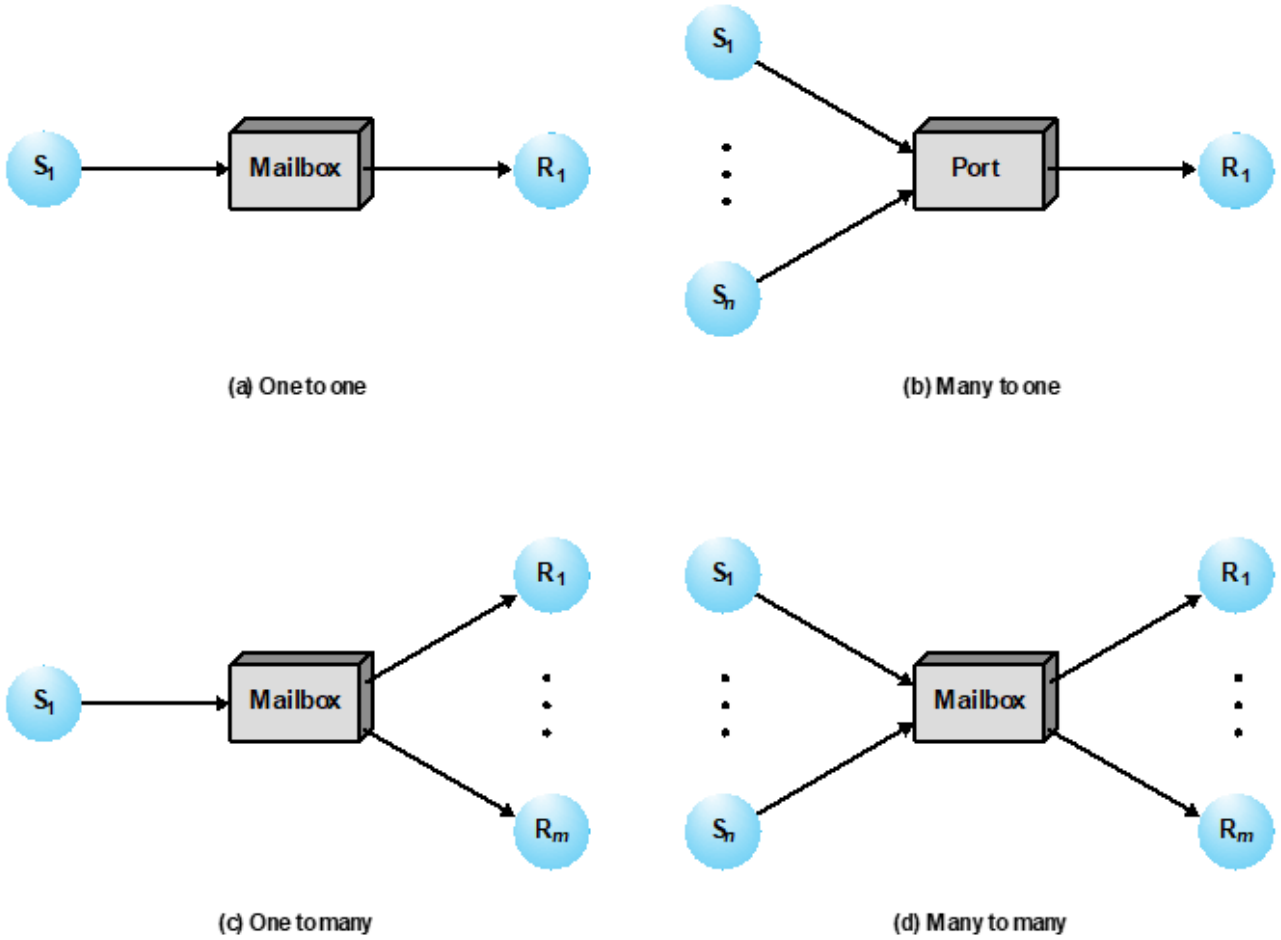
Indirect Addressing



Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti
Sapienza Università di Roma

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: Addressing (Indirect Process Communication)



a) A **one-to-one** relationship allows a private communications link to be set up between two processes. This *insulates their interaction from erroneous interference from other processes*.

b) A **many-to-one** relationship is useful for *client/server* interaction; one process provides service to a number of other processes. In this case, the mailbox is often referred to as a *port* .

c) *A **one-to-many*** relationship allows for one sender and multiple receivers; it is useful for applications where a message or some information is to be *broadcast* to a set of processes.

d) A **many-to-many** relationship allows multiple server processes to provide concurrent service to multiple clients.

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA UNIVERSITÀ DI ROMA

Operating Systems: Concorrenza – altri meccanismi

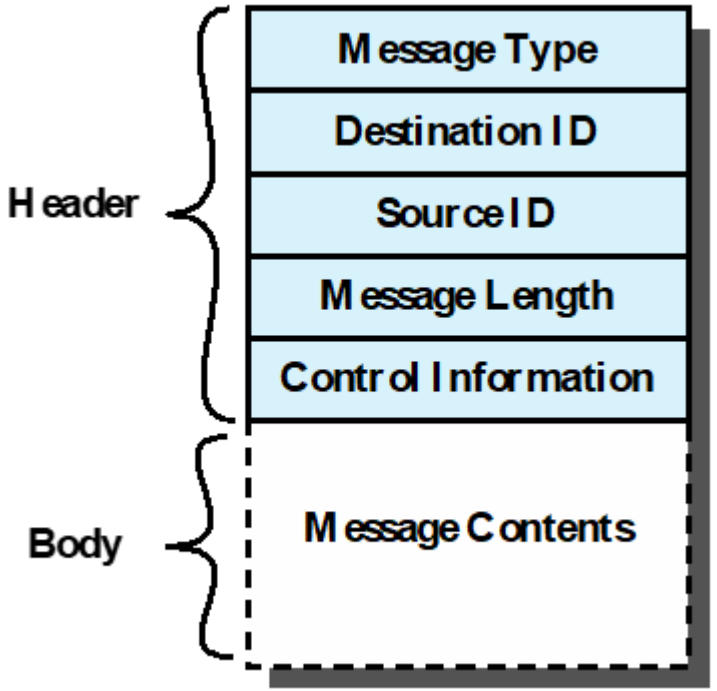
Mailboxes/Message: Format

The format of the message depends on the objectives of the messaging facility and whether the facility runs on a single computer or on a distributed system.

For some operating systems, designers have preferred short, fixed-length messages to minimize processing and storage overhead.

If a large amount of data is to be passed, the data can be placed in a file and the message then simply references that file.

A more flexible approach is to allow variable-length messages.



Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: Mutual Exclusion

Assume the use of the blocking receive primitive and the nonblocking send primitive. A set of concurrent processes share a mailbox, box , which can be used by all processes to send and receive.

The mailbox is initialized to contain a single message with null content. A process wishing to enter its critical section first attempts to receive a message. If the mailbox is empty, then the process is blocked. Once a process has acquired the message, it performs its critical section and then places the message back into the mailbox. Thus, the message functions as a token that is passed from process to process.

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true) {
      receive (box, msg);
      /* critical section   */;
      send (box, msg);
      /* remainder    */;
    }
}
void main()
{
    create_mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

The preceding solution assumes that if more than one process performs the receive operation concurrently, then:

- If there is a message, it is delivered to only one process and the others are blocked,

or

- If the message queue is empty, all processes are blocked; when a message is available, only one blocked process is activated and given the message.

These assumptions are true of virtually all message-passing facilities.

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: Mutual Exclusion

Soluzione al problema della mutua esclusione tra processi, gestita con messaggi:

```
const message null = /* null message */;
    mailbox box;
    void P(int i) {
      message msg;
      while (true) {
        receive(box, msg);
        /* critical section */;
        send(box, msg);
        /* remainder */;
    } }

    void main() {
      box = create_mailbox();
      nbsend(box, null);
      parbegin (P(1), P(2), . . ., P(n));
    }
```

- Deadlock in funzione di n
- Mutua Exclusion: requisiti

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: Concorrenza – altri meccanismi

Mailboxes/Message: Mutua Esclusione

Soluzione al problema della mutua esclusione tra processi, gestita con messaggi:

```
const message null = /* null message */;
    mailbox box;
    void P(int i) {
      message msg;
      while (true) {
        receive(box, msg);
        /* critical section */;
        nbsend(box, msg);
        /* remainder */;
    } }

    void main() {
      box = create_mailbox();
      nbsend(box, null);
      parbegin (P(1), P(2), . . ., P(n));
    }
```

- Deadlock in funzione di n
- Mutua Exclusion: requisite base

Hints

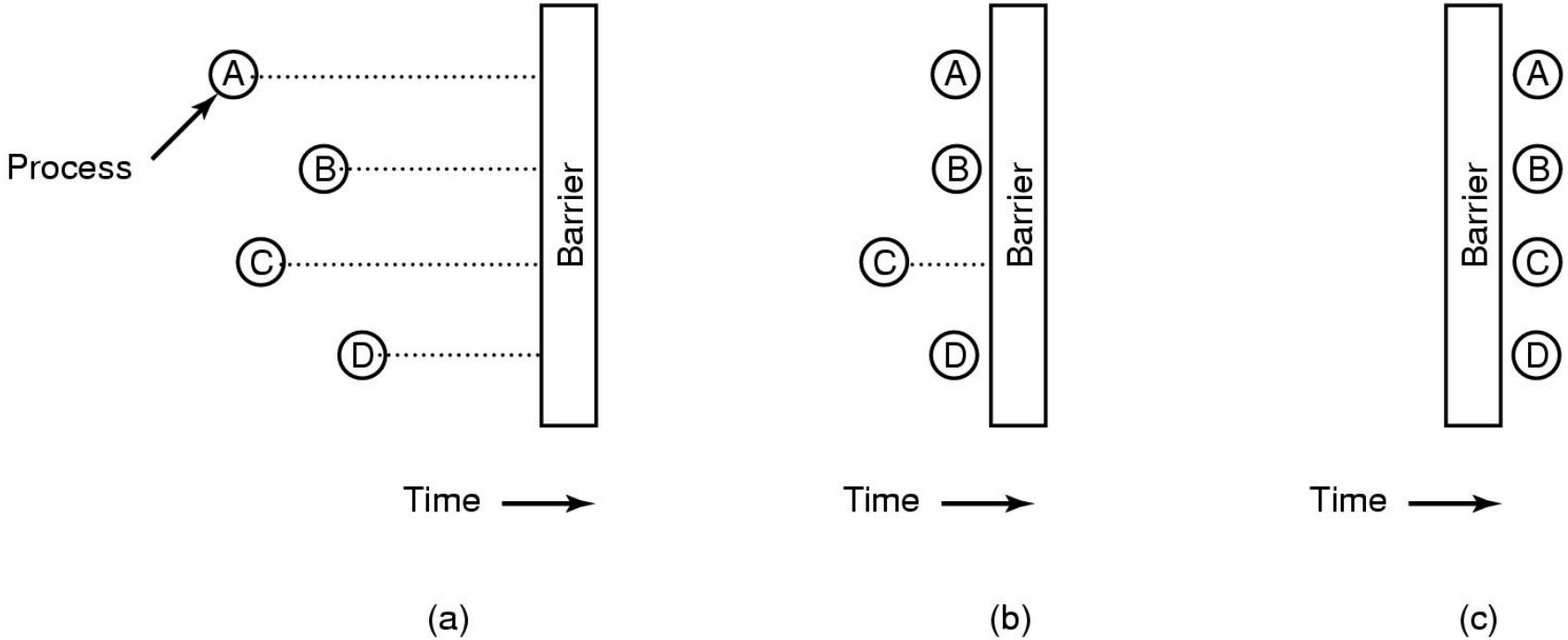
- funzioni bloccanti

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: Concorrenza – altri meccanismi

Barriers



Sincronizzare gruppi di processi.

Operating Systems: Concorrenza – altri meccanismi

SpinLock

Uno spin lock è un modo per proteggere una risorsa condivisa dalla modifica di due o più processi contemporaneamente. Usato nel kernel dei sistemi operativi.

Il primo processo che tenta di modificare la risorsa "acquisisce" il blocco e continua sulla sua strada, facendo ciò che era necessario con la risorsa.

Tutti gli altri processi che successivamente tentano di acquisire il blocco vengono arrestati; si dice che "ruotano in posizione" in attesa che il blocco venga rilasciato dal primo processo, da cui il nome spin lock. ➔ Busy Waiting

➔ **OK: Sezioni Critiche Corte** per architetture multiprocessore. Non ha senso in architetture mono-processore: blocca l'unico processore attivo (allora, forse conviene disabilitare gli interrupts). Spinlock utile negli SMP: evita il context switch.

➔ **NO: Sezioni Critiche Lunghe** usare al posto semafori o messaggi , soprattutto in architetture SMP. Anche su sistemi monoprocessore se il Context Switch è meno dannoso del Busy Waiting

Per approfondire: https://0xax.gitbooks.io/linux-insides/content/SyncPrim/linux-sync-1.html

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Concurrency: producer-consumer

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Operating Systems: IPC
Resource Sharing

La condivisione di risorse può quindi essere implementata con 3 metodi diversi

1. istruzioni hardware (TSL, XCHG) ➔ busy waiting
2. sincronizzazione (semafori) ➔ risorsa riusabile
3. message passing (sincronizzazione + informazione) ➔ risorsa consumabile

Se si può implementare un'applicazione con uno qualsiasi dei 3 metodi, allora lo si può fare anche con gli altri 2
un particolare meccanismo potrà rivelarsi più conveniente degli altri, in termini di:

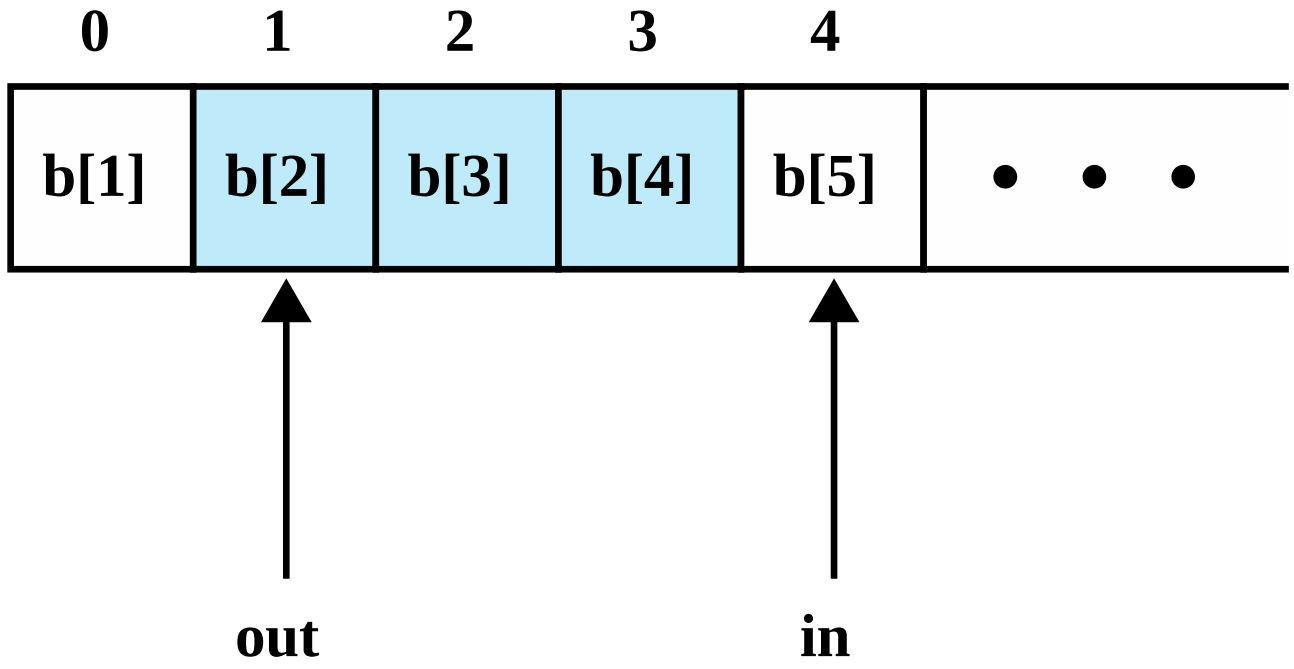
- **Facilità di sviluppo**: considerazioni inerenti il number di processi concorrenti
- **Prestazioni**: considerazioni inerenti il Busy Waiting
- Gestione:

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Producer/Consumer Problem

General Statement:	one or more producers are generating data and placing these in a buffer
	one or more consumer are taking items out of the buffer one at a time
	only one producer or consumer may access the buffer at any one time
The Problem:	ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer



Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

Infinite-Buffer P/C - An incorrect solution

```
/* program producerconsumer */
int n = 0;
binary_semaphore s = 1, delay = 0;
```

```
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Infinite-Buffer P/C - An incorrect solution

```
/* program producerconsumer */
int n = 0;
binary_semaphore s = 1, delay = 0;
```

```
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
```

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Consuming an item that does not exist...

Producer	Consumer	s	n	Delay
		1	0	0
`semWaitB(s)`		0	0	0
`n++`		0	1	0
if `(n==1)` `(semSignalB(delay))`		0	1	1
`semSignalB(s)`		1	1	1
	`semWaitB(delay)`	1	1	0
	`semWaitB(s)`	0	1	0
	`n--`	0	0	0
	`semSignalB(s)`	1	0	0

Consumer is descheduled before it can test:
if (n==0) semWaitB(delay)
So the producer enters CS and increases n...

Consuming an item that does not exist...

	`semSignalB(s)`	1	0	0
`semWaitB(s)`		0	0	0
`n++`		0	1	0
if `(n==1) (semSignalB(delay))`		0	1	1
`semSignalB(s)`		1	1	1
	if `(n==0) (semWaitB(delay))`	1	1	1

Later on we get **n == -1**: we are reading an invalid element!

```
// consumer
semWaitB(s)
take(), n--
semSignalB(s)
consume()
if (n==0) ...
```

`semWaitB(s)`	0	1	1
`n--`	0	0	1
`semSignalB(s)`	1	0	1
if `(n==0) (semWaitB(delay))`	1	0	0
`semWaitB(s)`	0	0	0
`n--`	0	–1	0
`semSignalB(s)`	1	–1	0

Consuming an item that does not exist...

	Producer	Consumer	s	n	Delay
1			1	0	0
2	`semWaitB(s)`		0	0	0
3	`n++`		0	1	0
4	**if** `(n==1) (semSignalB(delay))`		0	1	1
5	`semSignalB(s)`		1	1	1
6		`semWaitB(delay)`	1	1	0
7		`semWaitB(s)`	0	1	0
8		`n--`	0	0	0
9		`semSignalB(s)`	1	0	0
10	`semWaitB(s)`		0	0	0
11	`n++`		0	1	0
12	**if** `(n==1) (semSignalB(delay))`		0	1	1
13	`semSignalB(s)`		1	1	1
14		**if** `(n==0) (semWaitB(delay))`	1	1	1
15		`semWaitB(s)`	0	1	1
16		`n--`	0	0	1
17		`semSignalB(s)`	1	0	1
18		**if** `(n==0) (semWaitB(delay))`	1	0	0
19		`semWaitB(s)`	0	0	0
20		`n--`	0	–1	0
21		`semSignalB(s)`	1	–1	0

Infinite-Buffer P/C - Another incorrect solution

```
/* program producerconsumer */
int n = 0;
binary_semaphore s = 1, delay = 0;
```

```
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Infinite-Buffer P/C - Another incorrect solution

```
/* program producerconsumer */
int n = 0;
binary_semaphore s = 1, delay = 0;
```

```
void consumer()
{
      semWaitB(delay);
      while (true) {
            semWaitB(s);
            take();
            n--;
            consume();
            if (n==0) semWaitB(delay);
            semSignalB(s);
      }
}
```

Deadlock

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

A possible fix (using binary semaphores only)

```
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)  {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
```

We make a local copy m = n of the variable, so we can read the correct value once we left the CS

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

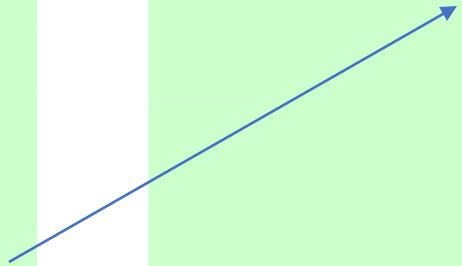
Sapienza Università di Roma

Infinite-Buffer P/C – General semaphores

```
/* program producerconsumer */
semaphore n = 0, s = 1;
```

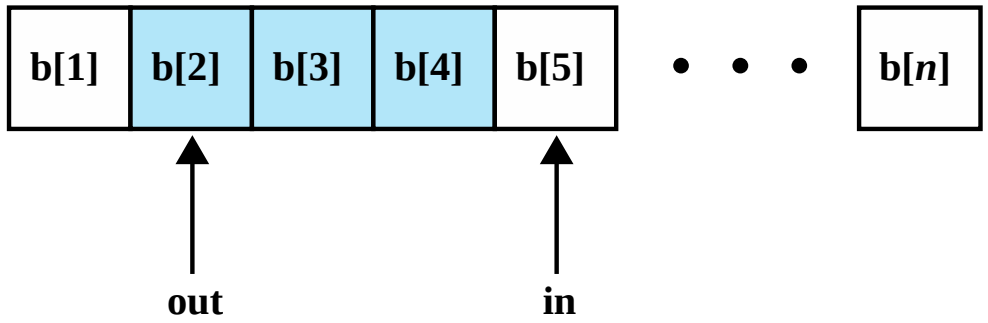
```
void producer()
{
   while (true) {
         produce();
         semWait(s);
         append();
         semSignal(s);
         semSignal(n);
   }
}
```

```
void consumer()
{
   while (true) {
         semWait(n);
         semWait(s);
         take();
         semSignal(s);
         consume();
   }
}
```

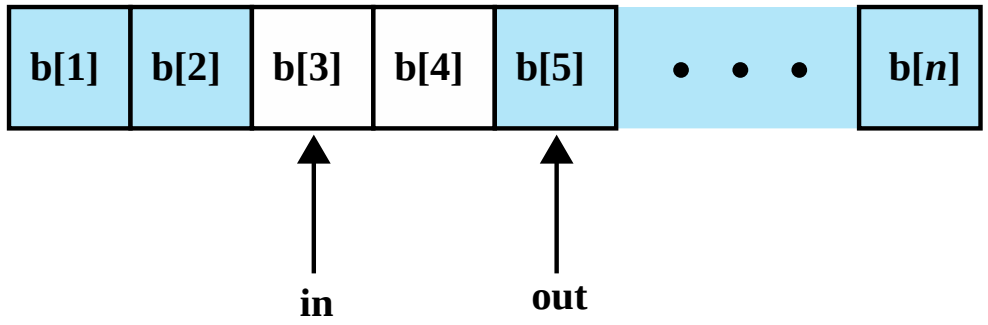


DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA



(a)



(b)

Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

Bounded-buffer Producer/Consumer

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
```

```
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Bounded-buffer Producer/Consumer

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
```

```
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
```

Dipartimento di Ingegneria informatica
automatica e gestionale Antonio Ruberti

Sapienza
Università di Roma

Bounded-buffer Producer/Consumer

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
```

```
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
```

```
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
```

What if I swap pairs of adjacent semWait or semSignal operations???
What if I have only a producer or only a reader???

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti
Sapienza Università di Roma

Bounded-buffer P/C using Monitors

```
void producer()
{
     char x;
     while (true) {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true) {
       take(x);
       consume(x);
     }
}
```

Monitors:

- local data accessible only by the monitor's procedures
- only one process may be executing in the monitor at a time
- programmer can decide when a procedure running in the monitor should stop or resume depending on a condition predicate

Thus, consume() and produce() do not need to know how take() and append() are implemented inside the monitor...

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Bounded-buffer P/C using Monitors

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;  /* condition variables for synchronization */
```

Init: `nextin = nextout = count = 0;`

```
void append (char x)
{
     if (count == N) cwait(notfull);
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);
}
```

Wait if buffer is full, notify pending consumer process on exit (if any)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Bounded-buffer P/C using Monitors

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;  /* condition variables for synchronization */
```

Init: `nextin = nextout = count = 0;`

```
void take (char x)
{
     if (count == 0) cwait(notempty);
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;
     csignal(notfull);
}
```

Wait if buffer is empty, notify pending producer process on exit (if any)

(By the way, this algorithm supports *multiple consumers* too...)

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
      receive (mayproduce, pmsg);
      pmsg = produce();
      send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
      receive (mayconsume, cmsg);
      consume (cmsg);
      send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce,
null);
    parbegin (producer, consumer);
}
```

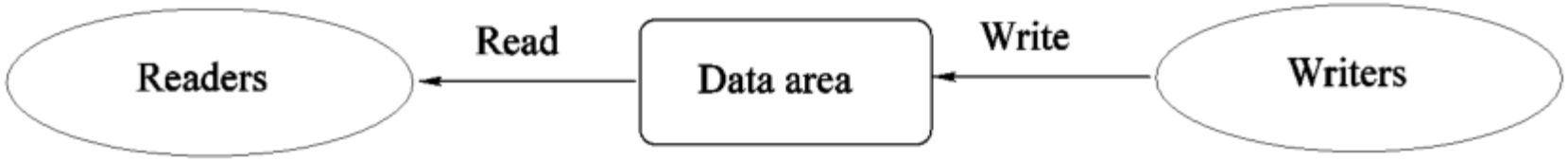
Figure 5.21

A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

Init. Mayproduce is full

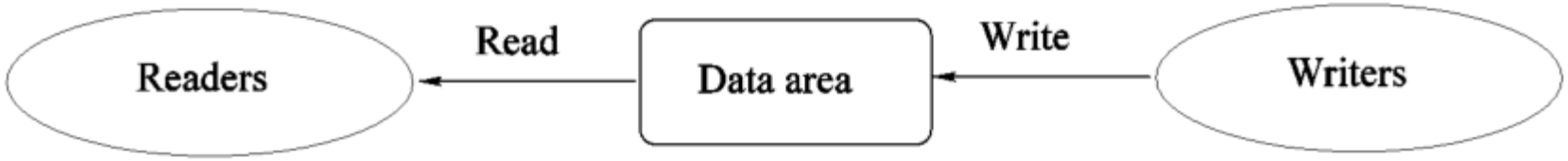
Init. Mayconsume is empty

Operating Systems: IPC
Readers/Writers Problem 1/4



- Una base di dati è condivisa fra vari processi concorrenti
- Lettori: accedono solo in lettura; non modificano la base di dati
- Scrittori: possono sia leggere che scrivere
- Problema:
- permettere a lettori multipli di leggere contemporaneamente (non crea interferenze)
- permettere ad un solo scrittore alla volta di accedere alla base di dati (accesso esclusivo)

Operating Systems: IPC
Readers/Writers Problem 1/7



- Una base di dati è condivisa fra vari processi concorrenti
- Lettori: accedono solo in lettura; non modificano la base di dati
- Scrittori: possono sia leggere che scrivere
- Problema:
- permettere a lettori multipli di leggere contemporaneamente (non crea interferenze)
- permettere ad un solo scrittore alla volta di accedere alla base di dati (accesso esclusivo)
- se è in atto una scrittura, non è permessa alcuna lettura

Operating Systems: IPC
Readers/Writers Problem 2/7

Variante 1 (priorità ai lettori):

- nessun lettore deve essere tenuto in attesa, a meno che uno scrittore abbia già ottenuto l'accesso alla base di dati condivisa, o anche
- Nessun lettore deve essere messo in attesa solo perché uno scrittore sta aspettando di scrivere
- È soggetta però al problema della starvation degli scrittori

Variante 2 (priorità agli scrittori):

- Uno scrittore deve poter svolgere la scrittura il più presto possibile, o anche
- Nessun nuovo lettore deve iniziare la lettura se uno scrittore è in attesa di scrivere
- È soggetta però al problema della starvation dei lettori

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: IPC

Readers/Writers Problem: semafori 3/7


```
/* program readersandwriters */
int readcount = 0;
semaphore x = 1, wsem = 1;
```

Requirements:

- any number of readers may read simultaneously
- only one writer at a time may write
- when a writer is writing, no reader is allowed to read

```
while (true) {
  semWait (wsem);
  WRITEUNIT();
  semSignal (wsem);
}
```

writer()

- acquire exclusive lock using binary semaphore wsem

Operating Systems: IPC
Readers/Writers Problem semafori 4/7


```
while (true) {
  semWait (x);
  readcount++;
  if (readcount == 1) semWait (wsem);
  semSignal (x);
  READUNIT();
  semWait (x);
  readcount--;
  if (readcount == 0) semSignal (wsem);
  semSignal (x);
}
```

reader()

- binary semaphore x to safely update variable readcount
- when there is only one reader (x==1) lock wsem (no writes!)
- once the read is performed, unlock wsem if no reader is active (i.e., readcount==0) => *writers may starve*, extensions needed!

Operating Systems: IPC

Readers/Writers Problem semafori 5/7

Readers only in the system	•*wsem* set •no queues
Writers only in the system	•*wsem* and *rsem* set •writers queue on *wsem*
Both readers and writers with read first	•*wsem* set by reader •*rsem* set by writer •all writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*
Both readers and writers with write first	•*wsem* set by writer •*rsem* set by writer •writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*

State of the Process Queues

For readers, one additional semaphore is needed.

A long queue must not be allowed to build up on rsem ; otherwise writers will not be able to jump the queue.

Therefore, only one reader is allowed to queue on rsem , with any additional readers queuing on semaphore z , immediately before waiting on rsem .

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma

Operating Systems: IPC

Readers/Writers Problem semafori 6/7

```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Writers Have Priority

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Operating Systems: IPC

Readers/Writers Problem: Message passing 7/7


```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
 }
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive (writerequest, msg);
               writer id = msg.id;
               count = count - 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```

Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti

Sapienza Università di Roma