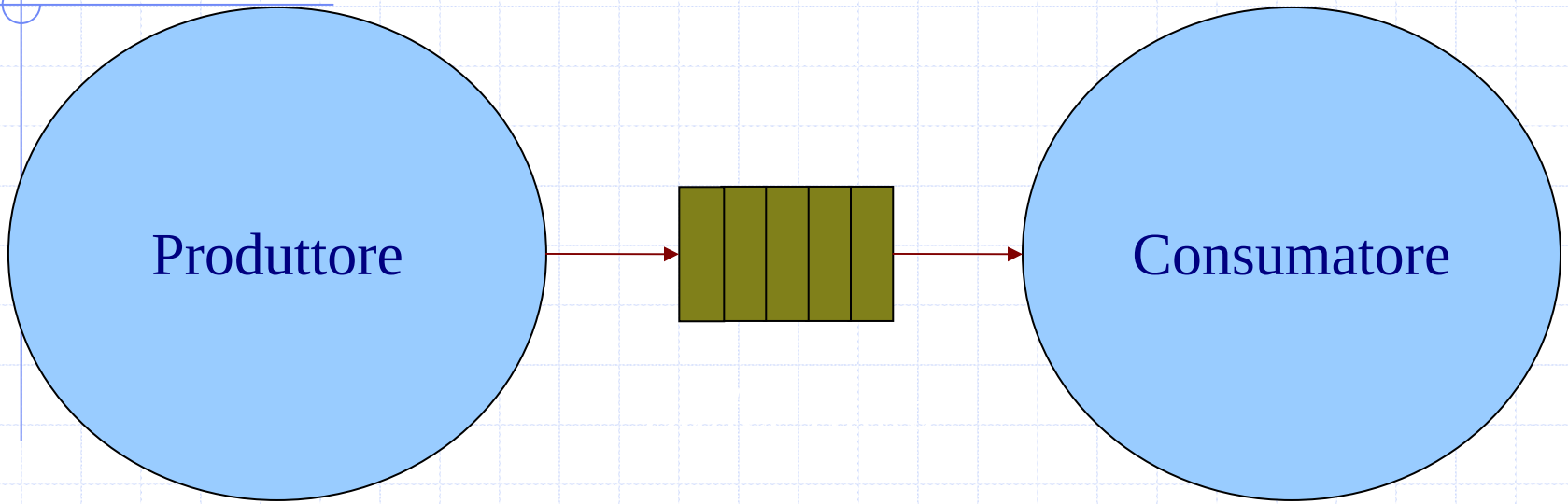
Problemi classici di sincronizzazione

Due esercizi sulla sincronizzazione dei processi

Motivazioni

- Metodo per valutare l'efficacia e l'eleganza di **modelli** e **meccanismi** per la sincronizzazione tra processi
  - **Lettori - scrittori** : accessi concorrenti a basi di dati
    - Chiamato anche "produttore – consumatore" sia al singolare che al plurale (differenza importante!)
  - **Filosofi a cena** : accesso esclusivo a risorse limitate
  - **Barbiere che dorme** : prevenzione di *race condition*
- Problemi pensati per illustrare tipiche situazioni di rischio
  - Stallo con blocco (*deadlock*)
  - Stallo senza blocco (*starvation*)
  - Esecuzioni non predicibili (*race condition*)

Produttore - Consumatore



- Non si scrive su *buffer* pieno
- Non si legge da *buffer* vuoto
- Produttore e consumatore lavorano a velocità diverse

Soluzione 1 (a)

```
#define N …                  /* posizioni del contenitore   */
typedef int semaforo;        /* P decrementa, V incrementa,
                                il valore 0 blocca la P     */

semaforo mutex = 1;
semaforo non-pieno = N;
semaforo non-vuoto = 0;

void produttore(){
   int prod;
   while(1){
      prod = produci();
      P(&non-pieno);
      P(&mutex);
      inserisci(prod);
      V(&mutex);
      V(&non-vuoto);
   }
}
```

```
void consumatore(){
   int prod;
   while(1){
      P(&non-vuoto);
      P(&mutex);
      prod = preleva();
      V(&mutex);
      V(&non-pieno);
      consuma(prod);
   }
}
```

Soluzione 1 (b)

- La correttezza della soluzione 1 dipende dal corretto posizionamento delle P e V in entrambi i processi!
  - Soluzione estremamente fragile
- Cosa accade se invertiamo le P nel codice del processo produttore?

Codice del produttore

```
P(&mutex);     // accesso esclusivo al contenitore
P(&non-pieno); // attesa spazi nel contenitore
```

Soluzione 2 (a)

```
int itemCount // condivisa ma non protetta

procedure producer() {
    while (true) {
        item = produceItem();

        if (itemCount == BUFFER_SIZE) {
            sleep() // si sospende
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1

        if (itemCount == 1) {
            wakeup(consumer);
        }
    }
}
```

```
procedure consumer() {
    while (true) {

        if (itemCount == 0) {
            sleep() // si sospende
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

Soluzione 2 (b)

- **Ausili**
  - La primitiva **sleep** sospende il processo chiamante
  - La primitiva **wakeup** (<*proc*>) invia un segnale di sveglia al processo <*proc*> designato
    - Il segnale si perde se il processo non è sospeso
- **Difetti**
  - *Race condition* su ItemCount che porta a stallo
    - Sospensione non legata atomicamente allo stato del *buffer*
    - Il prerilascio del consumatore prima di **sleep** porta i due processi a diversa percezione dello stato del sistema
      - Da cui deriva stallo

Soluzione 3 (a)

```
monitor PC
 condition non-vuoto, non-pieno;
 integer contenuto := 0;
 procedure inserisci(prod : integer);
 begin
  if contenuto = N then wait(non-pieno);
  <inserisci nel contenitore>;
  contenuto := contenuto + 1;
  if contenuto = 1 then signal(non-vuoto);
 end;
 function preleva : integer;
 begin
  if contenuto = 0 then wait(non-vuoto);
  preleva := <preleva dal contenitore>;
  contenuto :=  contenuto – 1;
  if contenuto = N-1 then signal(non-pieno);
 end;
end monitor;
```

```
procedure Produttore;
begin
 while true do begin
  prod := produci;
  PC.inserisci(prod);
 end;
end;
```

```
procedure Consumatore;
begin
 while true do begin
  prod := PC.preleva;
  consuma(prod);
 end;
end;
```

Soluzione 3 (b)

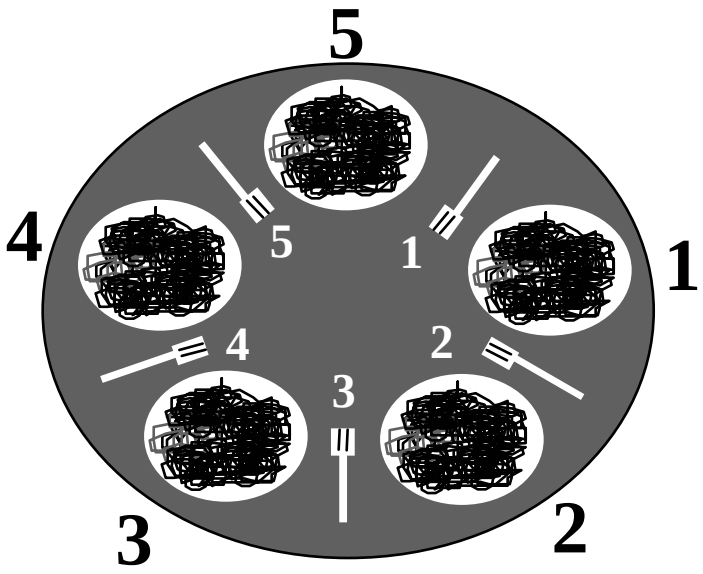
- L'uso del **monitor** elimina il rischio di *race condition*
- Però il programmatore deve comunque scegliere le condizioni logiche giuste per invocare **wait** e **signal** sulle variabili di condizionamento
  - Meno rischio di errore perché c'è un solo **monitor** invece che diversi utilizzatori di primitive di sincronizzazione

Filosofi a cena

- `N` filosofi sono seduti a un tavolo circolare
- Ciascuno ha davanti a se 1 piatto e 1 posata alla propria destra
- Ciascun filosofo necessita di 2 posate per mangiare
- L'attività di ciascun filosofo alterna pasti a momenti di riflessione

Soluzione 1

Soluzione con stallo (*deadlock*)




```
void filosofo (int i){
  while (TRUE) {
    medita();
    P(f[i]);
    P(f[(i+1)%N]);
    mangia();
    V(f[(i+1)%N]);
    V(f[i]);
  };
}
```

L'accesso alla prima forchetta non garantisce l'accesso alla seconda!

Soluzione 2

Soluzione con stallo (*starvation*)

```
void filosofo (int i){
 OK = FALSE;
 while (TRUE) {
  medita();
  while (!OK) {
   P(f[i]);
   if (!f[(i+1)%N]) {
    V(f[i]);
     sleep(T);}
   else {
    P(f[(i+1)%N]);
    OK = TRUE;
   };
  mangia();
  V(f[(i+1)%N]);
  V(f[i]);
 }
```

Un'attesa a durata costante difficilmente genera una situazione differente!

Soluzione 3

- Il problema ammette diverse soluzioni
  1. Utilizzare in soluzione A un semaforo a mutua esclusione per incapsulare gli accessi a entrambe le forchette
     - Funzionamento garantito
  2. In soluzione B, ciascun processo potrebbe attendere un tempo **casuale** invece che fisso
     - Funzionamento non garantito
  3. Algoritmi sofisticati, con maggiore informazione sullo stato di progresso del vicino e maggior coordinamento delle attività
     - Funzionamento garantito

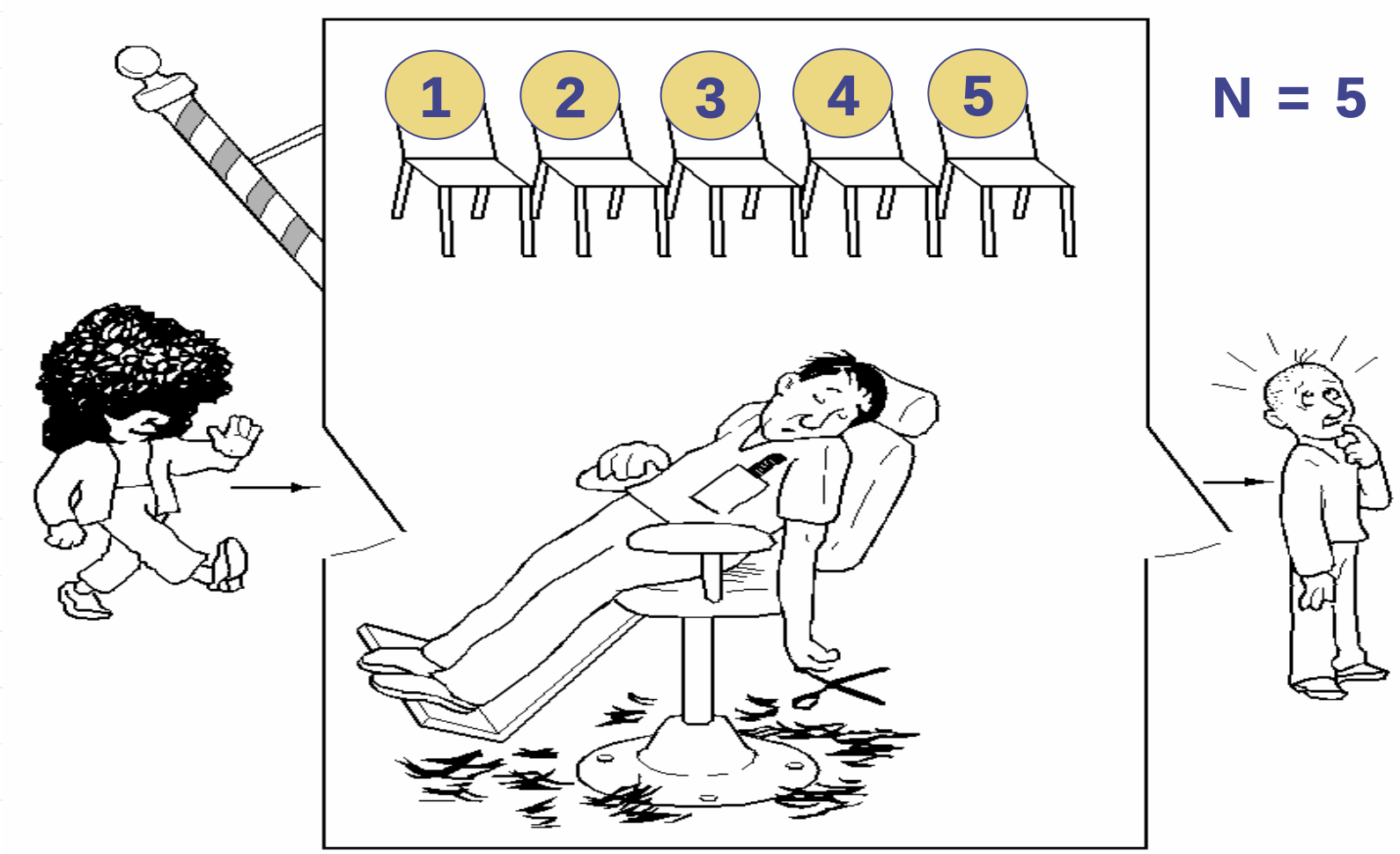
Il barbiere che dorme

Terzo esercizio sulla sincronizzazione dei processi

Specifica del problema

- Un negozio di barbiere con
  - **1** barbiere
  - **1** poltrona per il cliente servito
  - **N** sedie per clienti in attesa
- Protocollo di servizio
  a. Se non vi sono clienti nel negozio il barbiere dorme sulla poltrona
  b. Il primo cliente che entra nel negozio vuoto sveglia il barbiere
  c. I clienti che entrano trovando la poltrona occupata si mettono in attesa su una sedia
  d. Il cliente che non trova una sedia libera va a cercare un altro barbiere senza attendere

Illustrazione del problema



Uno schema di soluzione – 1

- Processo “barbiere”
  - Ripeti
    - { Attendi cliente
    - Servi cliente }
- Processo “cliente”
  - Entra nel negozio
    - Richiedi servizio
      - Se possibile, attendi
        - Ottieni servizio
      - Altrimenti cerca un altro barbiere

Contatore **protetto** di clienti in attesa

Uno schema di soluzione – 2

```
protected body Semaforo is
   entry P when CiSonoClienti is
      begin
         InCoda := InCoda - 1;
         CiSonoClienti := (InCoda > 0);
      end P;
   procedure V (Esito :    out Boolean) is
      begin
         InCoda := InCoda + 1;
         CiSonoClienti := (InCoda > 0);
         Esito := True;
      exception
         --  quando ci sono più clienti di sedie la coda del negozio “tracima”
         --+ causando un overflow nel valore InCoda
         --+ il cui trattamento a programma notifica al cliente esito negativo
         when Constraint_Error =>
            Esito := False;
      end V;
end Semaforo;
```

invocato dal barbiere

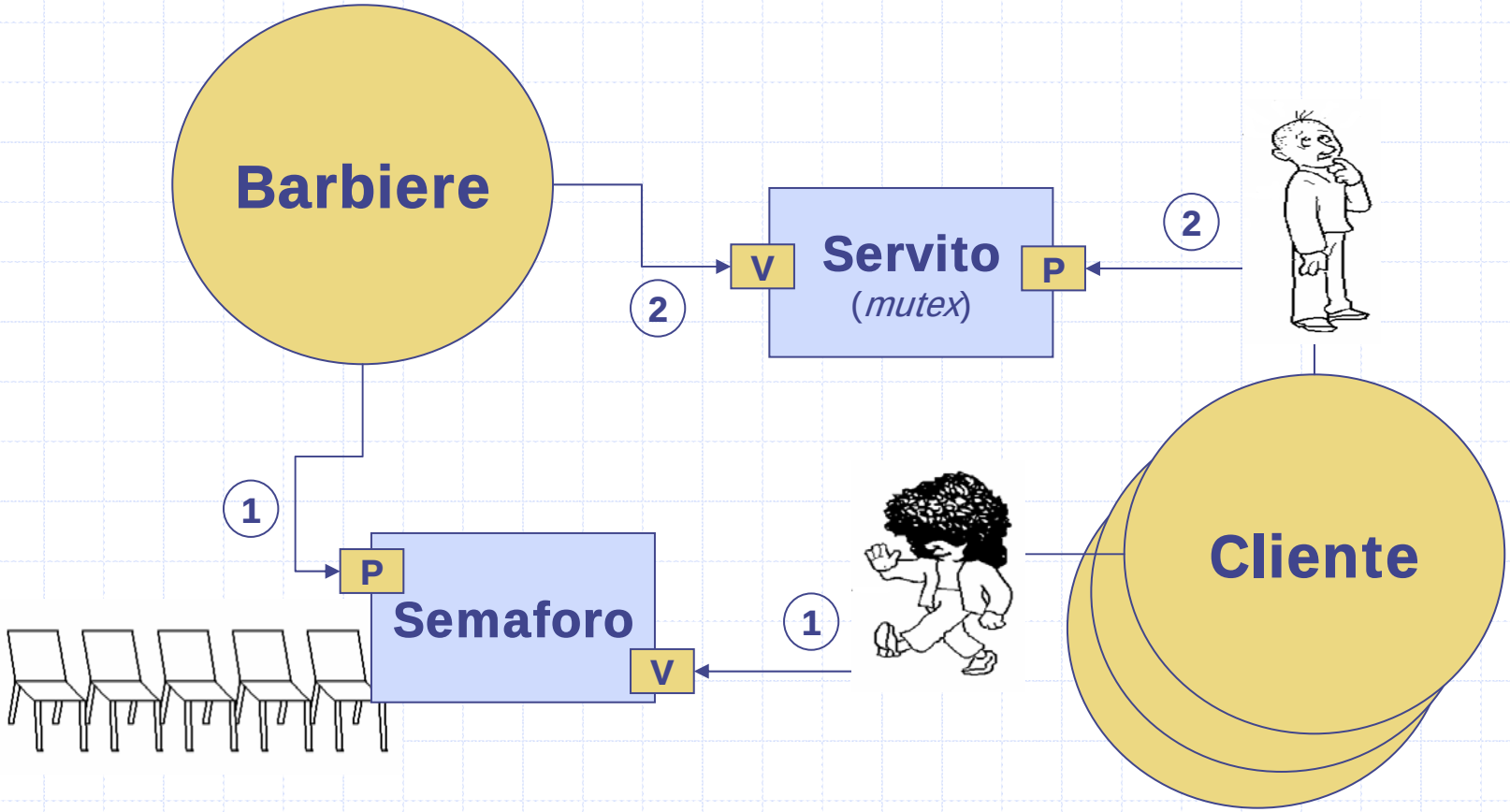
ogni cliente servito libera una sedia

invocato dai clienti

ogni cliente in arrivo occupa una sedia

Un semaforo contatore piuttosto sofisticato

Uno schema di soluzione – 3



Uno schema di soluzione – 4

Processo Barbiere

```
loop
  Semaforo.P; // dorme in attesa di clienti
    ... // serve il primo cliente in attesa
  Servito.V; // rilascia il cliente servito (tramite un mutex)
end loop;
```

Processo Cliente

```
loop
  Semaforo.V (Esito); // verifica disponibilità di sedie in negozio
    if Esito then
      Servito.P; // attende il servizio
        ... // se ne va servito
    else
        ... // va a cercare un altro barbiere
    end if:
end loop;
```

Due eseguibili di prova

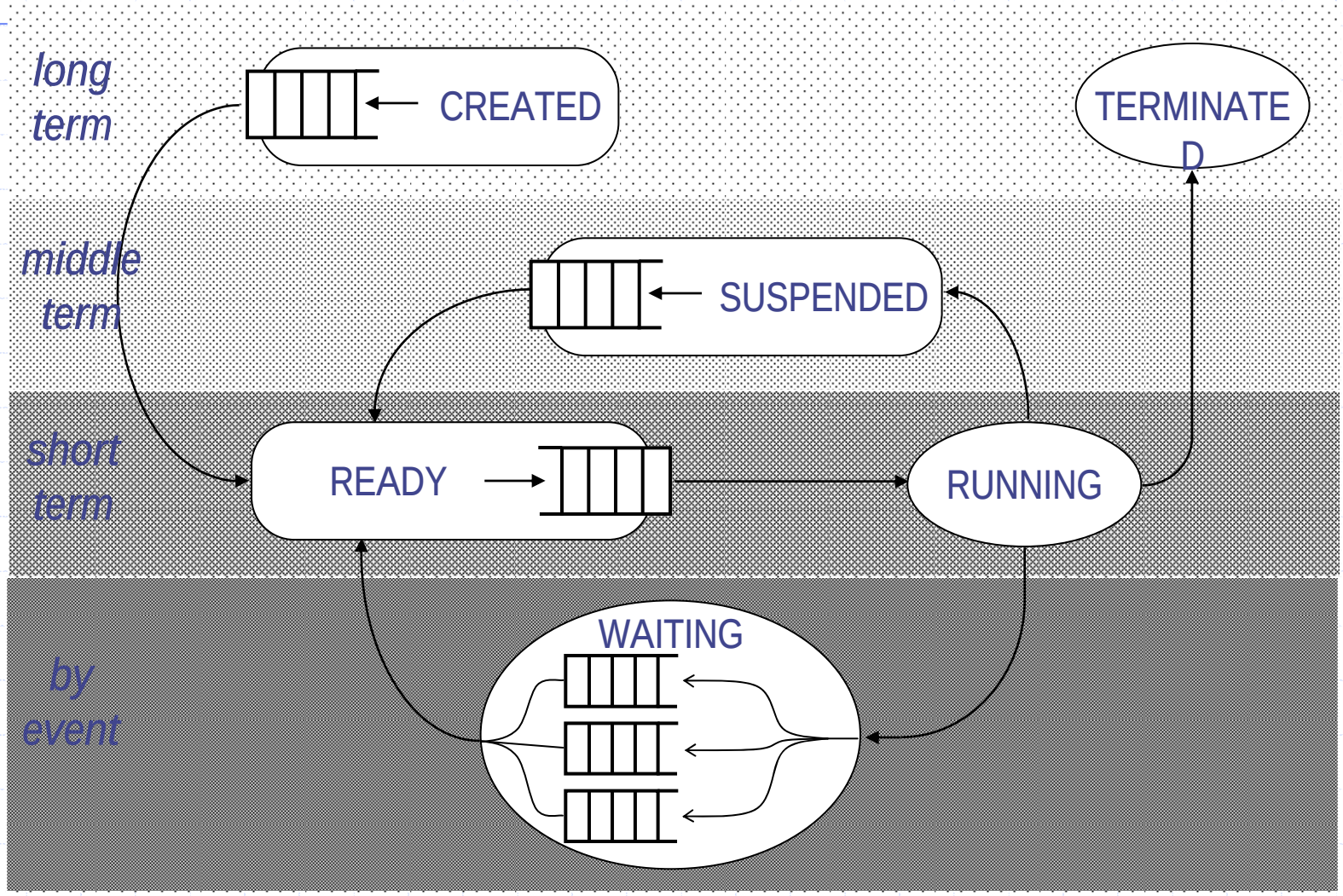
- Un cliente in arrivo ogni 5 secondi
- Tempo di servizio variabile tra 1 e 16 secondi
- **Versione 1**
  - Per MS Windows (☆)
- **Versione 2**
  - Per ambiente Java

Ordinamento dei processi

Materiale preparato da: A. Memo

Rivisitazione e discussione in aula:
Claudio Palazzi – *cpalazzi@math.unipd.it*

Fasi di ordinamento



L’ordinamento dei processi

- Criteri quantitativi di valutazione prestazionale delle politiche di ordinamento
  - **Efficienza di utilizzo**
    - Tempo utile/tempo di gestione
  - ***Throughput***
    - Processi completati per unità di tempo
  - **Tempo di *turn-around***
    - Tempo di completamento
  - **Tempo di attesa**
  - **Tempo di risposta**

Attribuzione della CPU – 1

- Consiste nel selezionare un processo dalla ***ready list*** e attribuirgli la CPU
- L'operazione viene effettuata in modo coordinato dallo ***scheduler*** e dal ***dispatcher***
  - Moduli del nucleo del sistema operativo
    - Lo *scheduler* fissa la politica
    - Il *dispatcher* ne attua le scelte

Attribuzione della CPU – 2

- Alcune politiche di ordinamento
  - ***First Come First Served*** [FCFS]
  - ***Round Robin*** [RR]
  - ***Shortest Job First*** [SJF]
    - Versione base senza prerilascio
    - Diventa ***Shortest Remaining Time Next*** [SRTN] se applicata con prerilascio
  - Con attributo di priorità statica associata ai processi e con prerilascio [FPS]

First Come First Served – 1.1

- La CPU viene assegnata al processo che la richiede per primo
  - Selezione dei processi da una coda FIFO

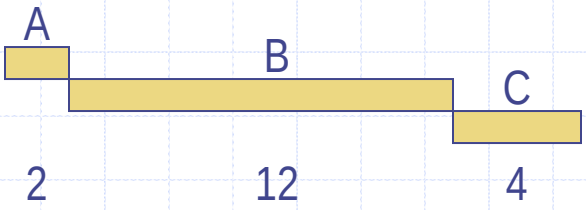
Esempio processo **A**: tempo di esecuzione = 2 [u.t.]

processo **B**: tempo di esecuzione = 12 [u.t.]

processo **C**: tempo di esecuzione = 4 [u.t.]

N.B. trascuriamo per semplicità i tempi di scambio di contesto

First Come First Served – 1.2



TEMPO DI ATTESA

$T_{att}(A) = 0$

$T_{att}(B) = 2$

$T_{att}(C) = 2 + 12 = 14$

$T_{att}(medio) = (0 + 2 + 14) / 3 = 5,3$ [u.t.]

TEMPO DI *TURN AROUND*

$T_{ta}(A) = 2$

$T_{ta}(B) = 2 + 12 = 14$

$T_{ta}(C) = 2 + 12 + 4 = 18$

$T_{ta}(medio) = (2 + 14 + 18) / 3 = 11,3$ [u.t.]

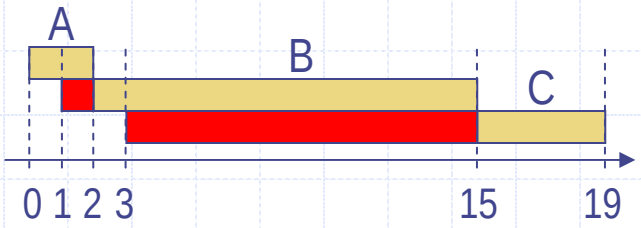
… e il Tempo di Risposta ?

First Come First Served – 2.1

Esempio 2

processo **A**: tempo di arrivo = 0
tempo di esecuzione = 2 [u.t.]
processo **B**: tempo di arrivo = 1
tempo di esecuzione = 12 [u.t.]
processo **C**: tempo di arrivo = 3
tempo di esecuzione = 4 [u.t.]

First Come First Served – 2.2



TEMPO DI ATTESA

$T_{att}(A) = 0$

$T_{att}(B) = 1$

$T_{att}(C) = 11$

$T_{att}(\text{medio}) = (0 + 1 + 11) / 3 = 4{,}3$ [u.t.]

TEMPO DI *TURN AROUND*

$T_{ta}(A) = 2$

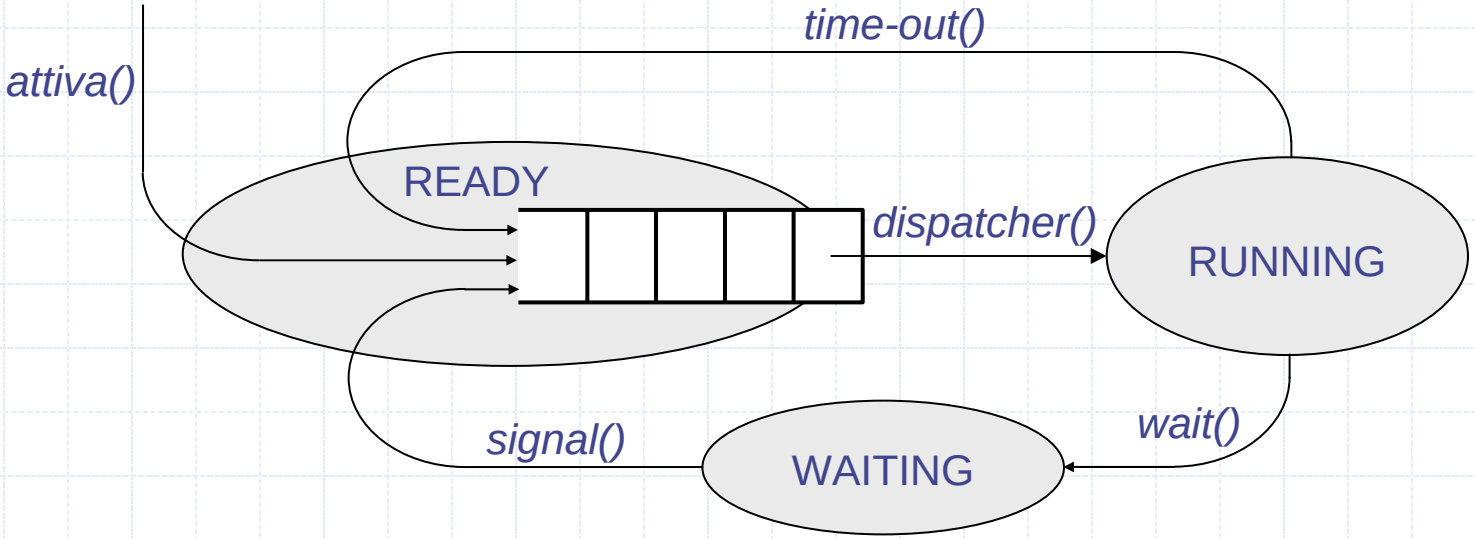
$T_{ta}(B) = 1 + 12 = 13$

$T_{ta}(C) = 11 + 4 = 15$

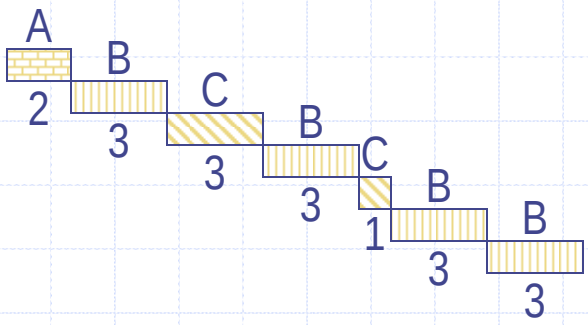
$T_{ta}(\text{medio}) = (2 + 13 + 15) / 3 = 10{,}3$ [u.t.]

Round Robin – 1

- Opera come FCFS ma con prerilascio per esaurimento del quanto di tempo
  - La *ready list* viene trattata come una coda circolare



Round Robin – 2



Tempo di arrivo di A = 0, di esecuzione = 2

Tempo di arrivo di B = 0, di esecuzione = 12

Tempo di arrivo di C = 0, di esecuzione = 4

Quanto di tempo = 3 [u.t.]

TEMPO DI ATTESA

$T_{att}(A) = 0$

$T_{att}(B) = 2 + 3 + 1 = 6$

$T_{att}(C) = 2 + 3 + 3 = 8$

$T_{att}(\text{medio}) = (0 + 6 + 8) / 3 = 4{,}6$ [u.t.]

TEMPO DI *TURN AROUND*

$T_{ta}(A) = 2$

$T_{ta}(B) = 2 + 3 + 3 + 3 + 1 + 3 + 3 = 18$

$T_{ta}(C) = 2 + 3 + 3 + 3 + 1 = 12$

$T_{ta}(\text{medio}) = (2 + 18 + 12) / 3 = 10{,}6$ [u.t.]

Round Robin – 3

Calcolare i tempi di attesa e di *turn-around* medi con un valore di quanto prima di 1 e poi di 5 [u.t.]. Cambierà qualcosa?

Quanto di tempo = 1 [u.t.]

T_{att} (medio) = (2 + 6 + 6) / 3 = 4,6 [u.t.]

T_{ta} (medio) = (4 + 18 + 10) / 3 = 10,6 [u.t.]

Quanto di tempo = 5 [u.t.]

T_{att} (medio) = (0 + 6 + 7) / 3 = 4,33 [u.t.]

T_{ta} (medio) = (2 + 18 + 11) / 3 = 10,3 [u.t.]

Round Robin – 4

```
A   A
 B   B  B  B  BBBBBBBB
  C   C  C  C
```

Quanto di tempo = 1 [u.t.]

```
AaaA
bBbbBbBbBbBBBBBBBB
ccCccCcCcC
```

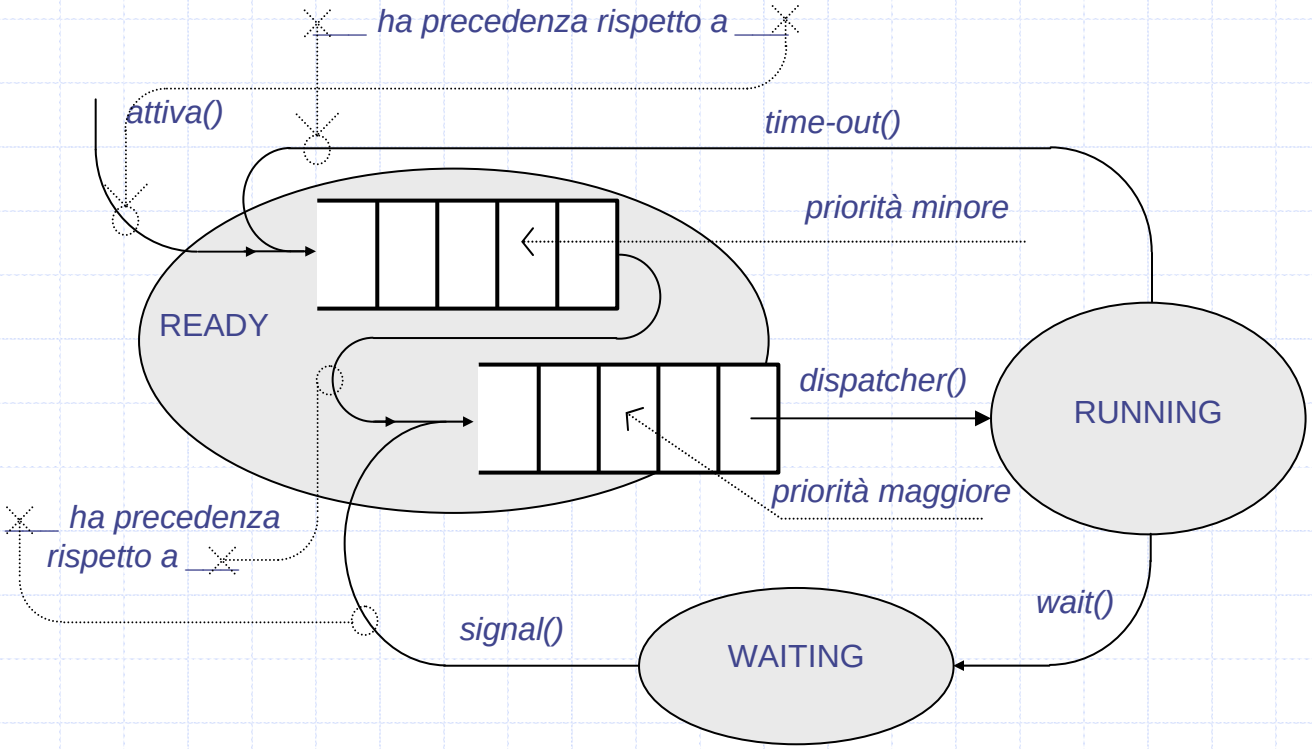
```
AA
   BBBBB       BBBBBBB
          CCCC
```

Quanto di tempo = 5 [u.t.]

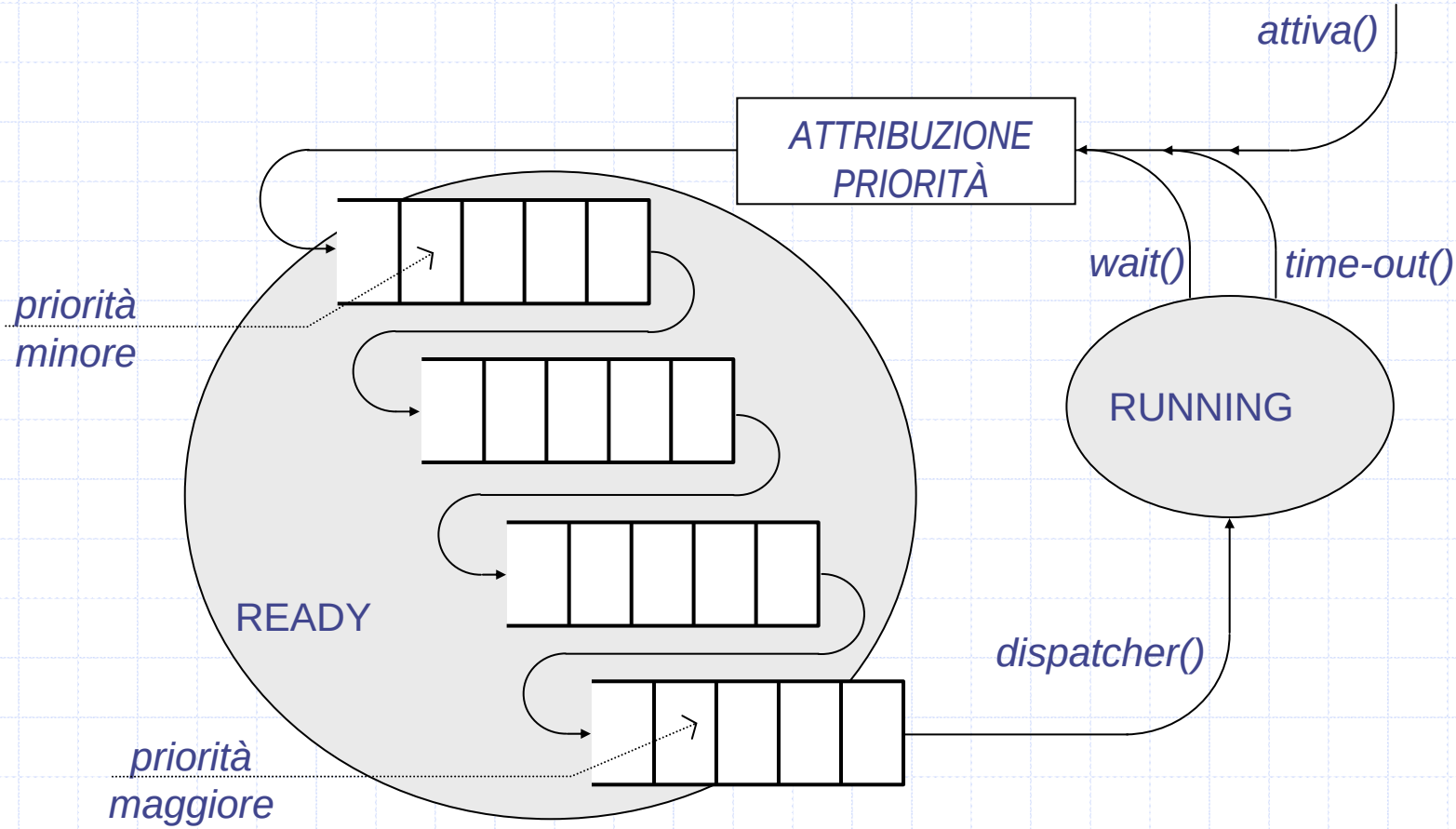
```
AA
bbBBBBBbbbbBBBBBBB
ccccccccCCCC
```

Round Robin con priorità all'I/O

- Processi *I/O bound* e *CPU bound*



Round Robin con priorità multiple – 1



Round Robin con priorità multiple – 2

TEMPO DI ATTESA = 16,50 [u.t.]

TEMPO DI *TURN AROUND* = 22,33 [u.t.]

Processo	Arrivo	Esecuzione	Priorità
A	0	7	2
B	0	6	4
C	0	10	3
D	0	2	3
E	0	7	1
F	0	3	5

Quanto di tempo = 4 [u.t.]

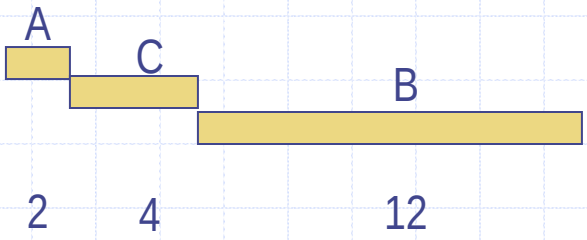
```
EEEE.EEE
aaaa.aaa.AAAA.AAA
cccc.ccc.cccc.ccc.CCCC.cc.CCCC.CC
dddd.ddd.dddd.ddd.dddd.DD
bbbb.bbb.bbbb.bbb.bbbb.bb.bbbb.bb.BBBB.BB
ffff.fff.ffff.fff.ffff.ff.ffff.ff.ffff.ff.FFF
```

Shortest Job First

- Meglio definita come
  - ***"Shortest next-CPU-burst First"***
- La CPU viene assegnata al processo che ha il "*CPU-burst*" successivo più breve
- Può essere realizzata senza uso di prerilascio
- Oppure con prerilascio
  - SRTN (*shortest remaining time next*)

Shortest Job First senza prerilascio

A, B, C arrivano al tempo 0



TEMPO DI ATTESA

$T_{att}(A) = 0$

$T_{att}(B) = 2 + 4 = 6$

$T_{att}(C) = 2$

$T_{att}(\text{medio}) = (0 + 6 + 2) / 3 = 2{,}6$ [u.t.]

TEMPO DI *TURN AROUND*

$T_{ta}(A) = 2$

$T_{ta}(B) = 2 + 4 + 12 = 18$

$T_{ta}(C) = 2 + 4 = 6$

$T_{ta}(\text{medio}) = (2 + 18 + 6) / 3 = 8{,}6$ [u.t.]

Shortest Job First con prerilascio

TEMPO DI ATTESA = 7 [u.t.]

TEMPO DI *TURN AROUND* = 12,83 [u.t.]

Processo	Arrivo	Esecuzione
A	2	7
B	0	6
C	5	10
D	10	2
E	7	7
F	4	3

```
BBBBBB
----ff.FFF
--aaaa.aaa.A.aa.AAAAAA
------.---.-.DD
------.-ee.e.ee.eeeeee.EEEEEEE
-----c.ccc.c.cc.cccccc.ccccccc.CCCCCCCCCC
```

Esercizio risolto – 1

- Cinque processi *batch*, identificati dalle lettere A-E arrivano all'elaboratore allo stesso istante. I processi hanno un tempo di esecuzione stimato di 8, 10, 2, 4 e 8 unità di tempo rispettivamente, mentre le loro priorità (fissate esternamente) sono rispettivamente 2, 4, 5, 1 e 3 (con 5 valore maggiore). Per ognuno dei seguenti algoritmi di ordinamento determinare: (i) il *tempo medio di turn-around* e (ii) il tempo medio di attesa, trascurando i tempi dovuti allo scambio di contesto.
  - *Round Robin* (con quanto di tempo = 2)
  - Con priorità (senza prerilascio)
  - FCFS
  - SJF

Soluzione – 1

RR (time slot = 2 [u.t.])

t_{att} (medio) = 15,6 [u.t.]

t_{ta}(medio) = 22 [u.t.]

Priorità (senza prerilascio)

t_{att} (medio) = 12,4 [u.t.]

t_{ta}(medio) = 18,8 [u.t.]

FCFS

t_{att} (medio) = 14 [u.t.]

t_{ta}(medio) = 20,4 [u.t.]

SJF

t_{att} (medio) = 8,8 [u.t.]

t_{ta}(medio) = 15,2 [u.t.]

Esercizio risolto – 2

- Cinque processi *batch*, identificati dalle lettere A-E, arrivano all'elaboratore agli istanti di tempo 0, 2, 5, 8 e 11 rispettivamente. I processi hanno un tempo di esecuzione stimato di 9, 1, 7, 3 e 5 unità di tempo rispettivamente, mentre le loro priorità (mantenute staticamente) sono rispettivamente 3, 2, 4, 5 e 1 (con 5 valore maggiore). Per ognuna delle seguenti politiche di ordinamento determinare **(i)** il **tempo medio di risposta**, **(ii)** il **tempo medio di *turn-around*** e **(iii)** il **tempo medio di attesa**, trascurando i tempi dovuti allo scambio di contesto.
  - FCFS
  - *Round Robin* 1 (quanto di tempo = 3)
  - *Round Robin* 2 (quanto di tempo = 3) con priorità ma senza prerilascio
    - Nel caso di arrivo di un processo in contemporanea a un'uscita per *time_out*(), si dia precedenza al processo prerilasciato per *time_out*()
  - SJF senza prerilascio
  - SJF con prerilascio

Soluzione – 2 (FCFS)

Processo	Arrivo	Esecuzione	Priorità
A	0	9	3
B	2	1	2
C	5	7	4
D	8	3	5
E	11	5	1

TEMPO DI RISPOSTA = 6,0 [u.t.]

TEMPO DI ATTESA = 6,0 [u.t.]

TEMPO DI *TURN AROUND* = 11,0 [u.t.]

```
AAAAAAAA
--bbbbbbbB
-----ccccccCCCCCCC
--------ddddddddddDDD
----------eeeeeeeeeeEEEEE
```

Soluzione – 2 (*Round Robin*)

- Quanto di tempo = 3

```
AAAaAAAaaaAAA
--bB
-----ccCCCcccccccCCCcccC
--------ddddddDDD
----------eeeeeeeeEEEeEE
```

TEMPO DI RISPOSTA = 3,2 [u.t.]
TEMPO DI ATTESA = 6,0 [u.t.]
TEMPO DI *TURN AROUND* = 11,0 [u.t.]

Soluzione – 2 (*Round Robin*)

Priorità: 3, 2, 4, 5 e 1 (con 5 valore maggiore)

- Con priorità senza prerilascio

TEMPO DI RISPOSTA = 5,6 [u.t.]

TEMPO DI ATTESA = 8,2 [u.t.]

TEMPO DI *TURN AROUND* = 13,2 [u.t.]

```
AAAAAAaaaaaaaaaaAAA
--bbbbbbbbbbbbbbbbB
-----cCCCcccCCCC
--------dDDD
----------eeeeeeeeeEEEEE
```

- Con priorità e prerilascio

```
AAAAAaaaaaaaaaaAAAA
--bbbbbbbbbbbbbbbbB
-----CCCcccCCCC
--------DDD
----------eeeeeeeeeEEEEE
```

TEMPO DI RISPOSTA = 5,2 [u.t.]

TEMPO DI ATTESA = 7,8 [u.t.]

TEMPO DI *TURN AROUND* = 12,8 [u.t.]

Soluzione – 2 (*SJF*)

Priorità: 3, 2, 4, 5 e 1 (con 5 valore maggiore)

- SJF senza prerilascio

TEMPO DI RISPOSTA = 4,8 [u.t.]

TEMPO DI ATTESA = 4,8 [u.t.]

TEMPO DI *TURN AROUND* = 9,8 [u.t.]

```
AAAAAAAAA
--bbbbbbbB
-----ccccccccccccCCCCCCC
--------ddDDD
----------eeEEEEE
```

- SJF con prerilascio

TEMPO DI RISPOSTA = 3,4 [u.t.]

TEMPO DI ATTESA = 3,6 [u.t.]

TEMPO DI *TURN AROUND* = 8,6 [u.t.]

```
AAaAAAAAAA
--B
-----ccccccccccccCCCCCCC
--------ddDDD
----------eeEEEEE
```

Esercizio non risolto – 1

- Si supponga che tre clienti arrivino a una stazione di servizio per fare il pieno di benzina, e che ognuno impieghi il seguente tempo, noto a priori

auto	arrivo	servizio (in minuti)
A	8:00	8
B	8:06	5
C	8:07	2

- Nell'ipotesi che alle 8:00 l'unica pompa di benzina della stazione sia libera, calcolare il tempo medio di attesa e il tempo medio di *turn-around* applicando politiche di ordinamento FIFO, SJF senza prerilascio e SJF con prerilascio.

Esercizio non risolto – 2

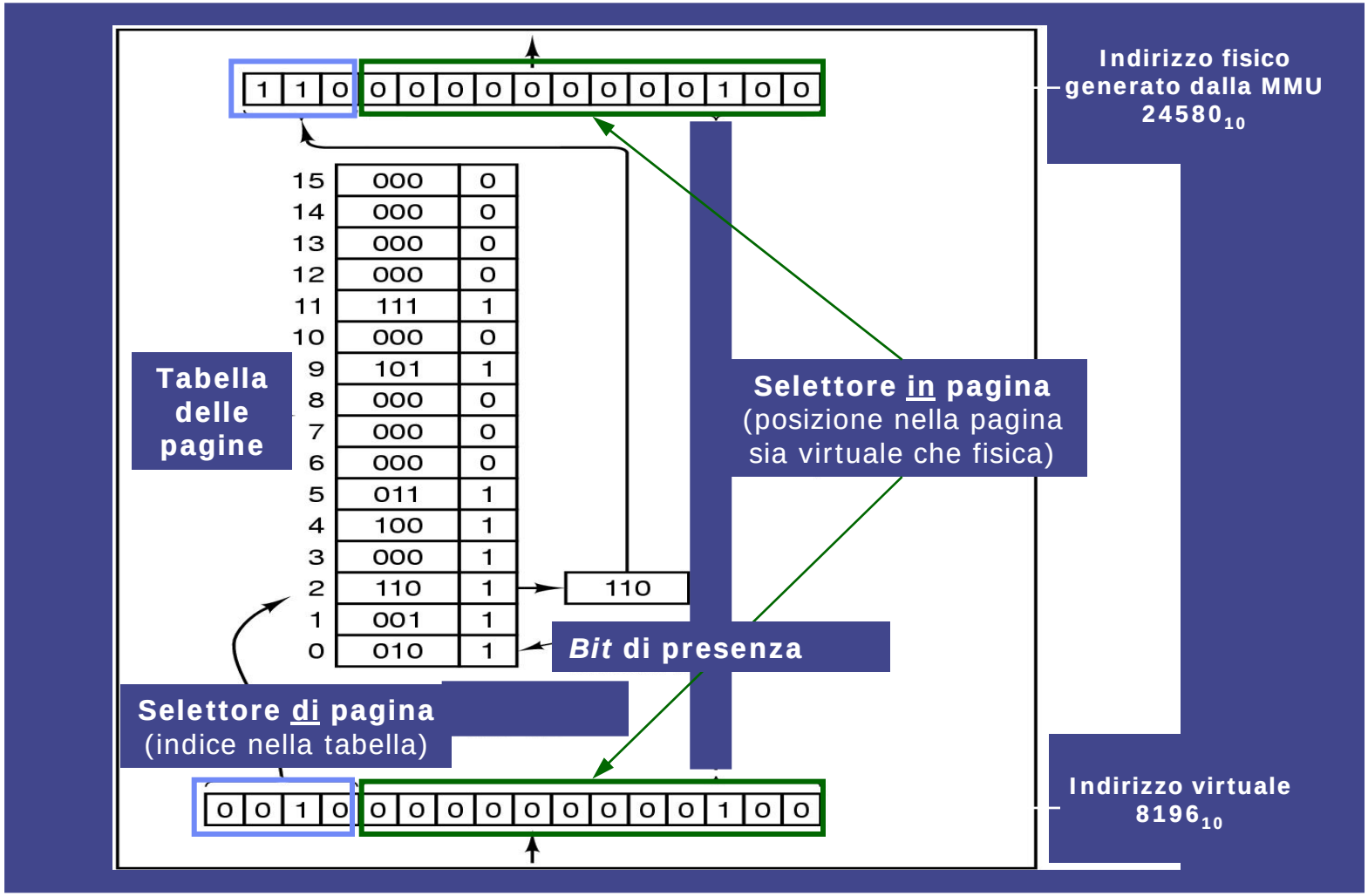
- Cinque processi *batch*, identificati dalle lettere A-E, arrivano all'elaboratore allo stesso istante. I processi hanno un tempo di esecuzione stimato di 10, 6, 2, 4 e 8 unità di tempo rispettivamente, mentre le loro priorità (determinate esternamente) sono rispettivamente 3, 5, 2, 1 e 4 (con 5 valore maggiore). Per ognuno delle seguenti politiche di ordinamento determinare: (i) **il tempo medio di *turn-around*** e (ii) il **tempo medio di attesa**, trascurando i tempi dovuti allo scambio di contesto.
  - *Round Robin* (quanto di tempo = 2)
  - Con priorità esterna senza prerilascio
  - FCFS
  - SJF senza prerilascio

“L’angolo della posta” (discussione su vostri quesiti)

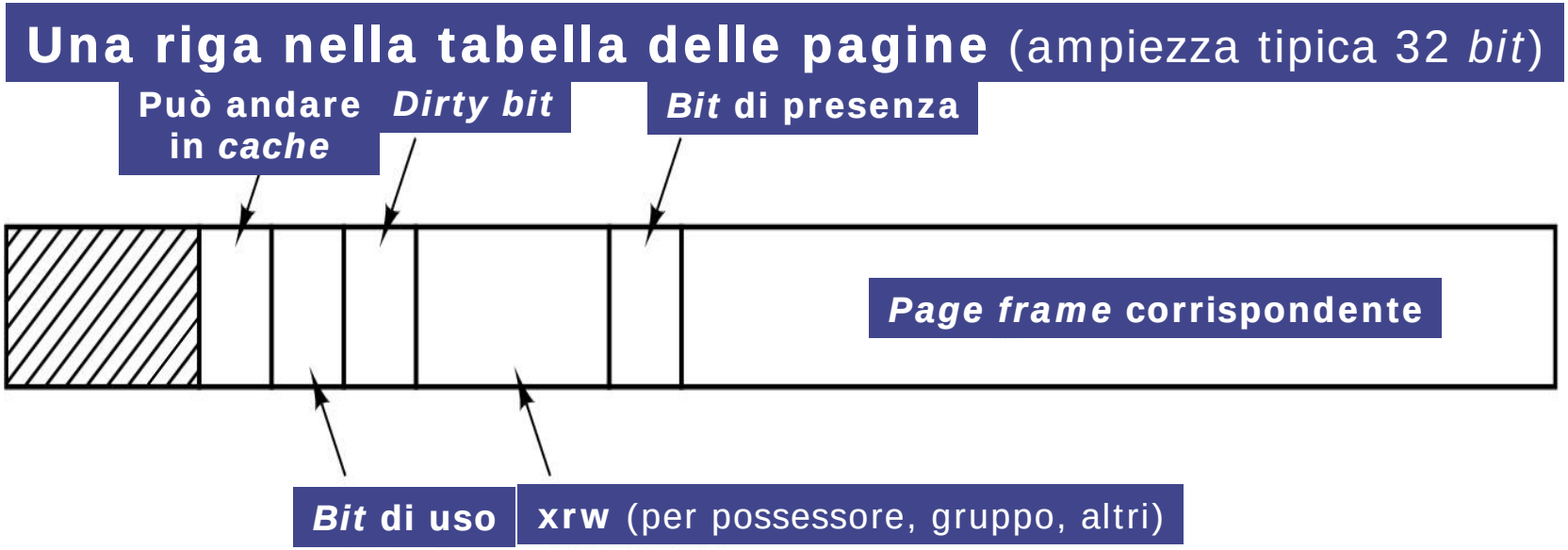
Discussione in aula:

Claudio Palazzi – *cpalazzi@math.unipd.it*

Paginazione: strutture – 1



Paginazione: strutture – 2

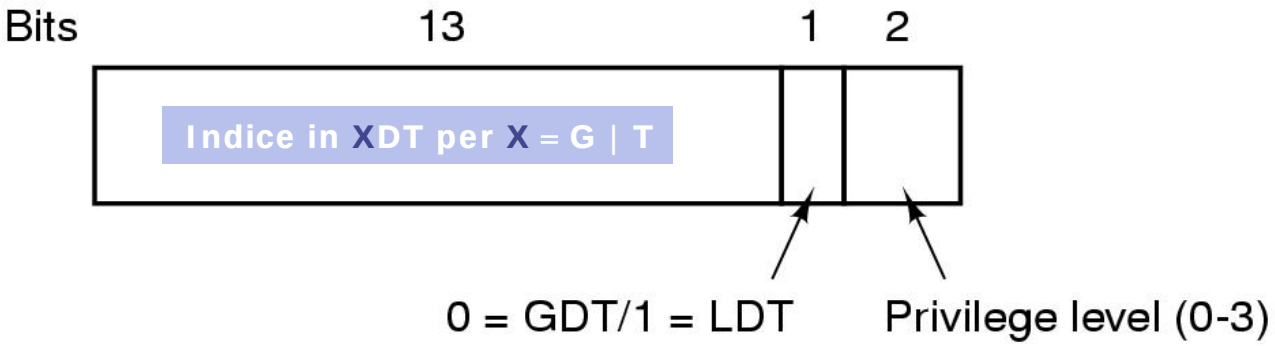


- L'indirizzo di disco ove la pagina si trova quando non è in RAM **non** è nella tabella!
  - La tabella delle pagine serve alla MMU (*hardware*)
  - Il caricamento della pagina da disco viene effettuato dal S/O (*software*)
  - L'informazione dell'uno **non serve** all'altro

Segmentazione: realizzazione – 1

- Vista la grande ampiezza potenziale i segmenti sono spesso **paginati**
- Nel caso del Pentium di Intel
  - Fino a 16 K segmenti indipendenti
    - Di ampiezza massima 4 GB (32 *bit*)
  - Una LDT per processo
    - *Local Descriptor Table*
      - Descrive i segmenti del processo
  - Una singola GDT per l'intero sistema
    - *Global Descriptor Table*
      - Descrive i segmenti del S/O

Segmentazione: realizzazione – 2



- 6 registri di segmento
  - Di cui 1 denota il segmento corrente
- LDT e GDT contengono $2^{13} = 8$ K descrittori di segmento
  - I descrittori di segmento sono espressi su 8 B
    - La **base** del segmento in RAM è espressa su 32 *bit*
    - Il **limite** su 20 *bit* per verificare la legalità dell'*offset* fornito dal processo
      - Consente ampiezza massima a 1 MB (per **granularità** a B)
      - Oppure 1 M pagine da 4 KB ovvero 4 GB (per granularità a pagine)

Segmentazione: realizzazione – 3

- L'indirizzo **lineare** ottenuto da (base di segmento + *offset*) può essere interpretato come
  - Indirizzo **fisico** se il segmento considerato non è paginato
  - Indirizzo **logico** altrimenti
    - Nel qual caso il segmento viene visto come una memoria virtuale paginata e l'indirizzo come virtuale in essa
      - 10 *bit* : indice in catalogo di tabelle delle pagine
        - 2^{10} righe da 32 *bit* ciascuna (base di tabella denotata)
      - 10 *bit* : indice in tabella delle pagine selezionata
        - 2^{10} righe da 32 bit ciascuna (base di *page frame*)
      - 12 *bit* : posizione nella pagina selezionata
        - *Offset* in pagina da 4 KB

“Esercizio 2”

Sia data memoria dotata di 4 *page frame*, inizialmente libere, e 8 pagine di memoria virtuale. Utilizzando la politica FIFO per il rimpiazzo delle pagine, indicare quanti *page fault* si verifichino a fronte della stringa di riferimenti: 0 1 7 2 3 2 7 1 0 3:

A: 4

B: 3

C: 6

D: 2

“Esercizio 3”

Si consideri la seguente serie di riferimenti a pagine di memoria:
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

Si considerino le seguenti politiche di rimpiazzo:

- LRU
- FIFO
- Optimal

Quanti *page fault* avvengono considerando un numero di *page frame* della RAM di 1, 2, 3, 4, 5, 6, 7 ?

“Esercizio 3” – soluzione FIFO

Consideriamo **FIFO** con 3 *page frame*
serie di riferimenti a pagine di memoria:

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6

NEW	1	2	3	4	4	1	5	6	2	1	1	3	7	6	6	2	1	1	3	6
		1	2	3	3	4	1	5	6	2	2	1	3	7	7	6	2	2	1	3
OLD			1	2	2	3	4	1	5	6	6	2	1	3	3	7	6	6	2	1
	P	P	P	P		P	P	P	P	P		P	P	P		P	P		P	P

“Esercizio 3” – soluzione LRU

Consideriamo **LRU** con 3 *page frame*
serie di riferimenti a pagine di memoria:

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6

	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
MRU		1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3
LRU			1	2	3	4	2	1	5	6	6	1	2	3	7	6	3	3	1	2
	P	P	P	P		P	P	P	P	P		P	P	P		P	P			P

“Esercizio 3” – soluzione OPT

Consideriamo **optimal** con 3 *page frame*
serie di riferimenti a pagine di memoria:

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	3
	2	2	2	2	2	2	2	2	2	2	2	7	7	7	2	2	2	2	2
		3	4	4	4	5	6	6	6	6	6	6	6	6	6	1	1	1	6
P	P	P	P			P	P				P	P			P	P			P

“Esercizio 5” (modificato dopo lezione)

Si consideri la matrice (o *array* bidimensionale):

```
int A[][] = new int[100][100];
```

Si assuma che la posizione **`A[0][0]`** di tale matrice sia posta alla locazione **200** di una memoria paginata con **3** pagine di dimensione **200** B.

Si assuma che un valore **`int`** occupi **1** B.

Si assuma che le pagine siano inizialmente tutte vuote e che il processo (il cui codice occupa esattamente **200** B) abbia la seguente esecuzione:

```
for (int i = 0; i < 100; i++){
   for (int j = 0; j < 100; j++){
      A[i][j] = 0;}}
```

Quanti *page fault* saranno generati usando LRU?

“Esercizio 5” – soluzione (1/2)

La matrice `A[i][j]` è scritta linearmente in memoria:

`A[0][0],A[0][1],A[0][2],…,A[0][99],A[1][0],`
`A[1][1],…,A[99][99]`

Il processo azzera le celle della matrice proprio nel loro ordine di memorizzazione.

Quindi inizialmente verranno caricati nelle pagine
Pagina 0: Il programma (che occupa esattamente 200 B)
Pagina 1: Celle da A[0][0] a A[1][99] incluse
Pagina 2: Celle da A[2][0] a A[3][99] incluse

“Esercizio 5” – soluzione (2/2)

A ogni iterazione del programma la memoria viene acceduta per leggere l’istruzione successiva
→ la pagina relativa al processo verrà continuamente acceduta

I primi 200 azzeramenti faranno accesso a Pagina 1, i successivi 200 a Pagina 2; l’ulteriore azzeramento (cella `A[4][0]`) causerà un *page fault* ; la pagina usata meno recentemente è Pagina 1 che verrà sostituita con le celle da `A[4][0]` a `A[5][99]` incluse. Arrivati all’azzeramento di `A[6][0]` sarà Pagina 2 ad essere stata usata meno di recente
E così via, causando in tutto 1 *page fault* iniziale per caricare il processo in Pagina 0 e 50 *page fault* di Pagina 1 e 2 (25 ciascuno) per caricare le porzioni di matrice.

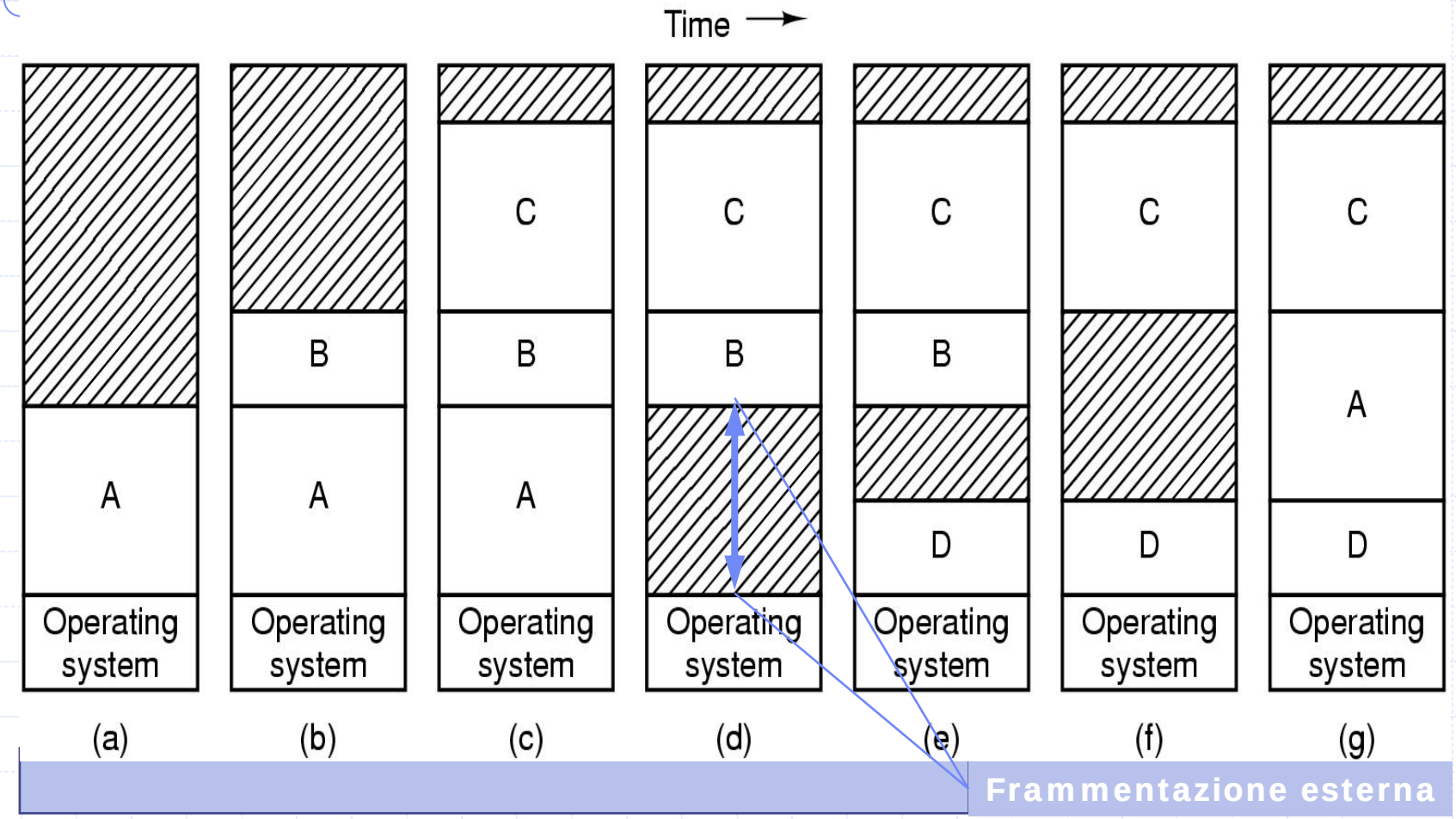
TOTALE = 51 *page fault*

Gestione della memoria

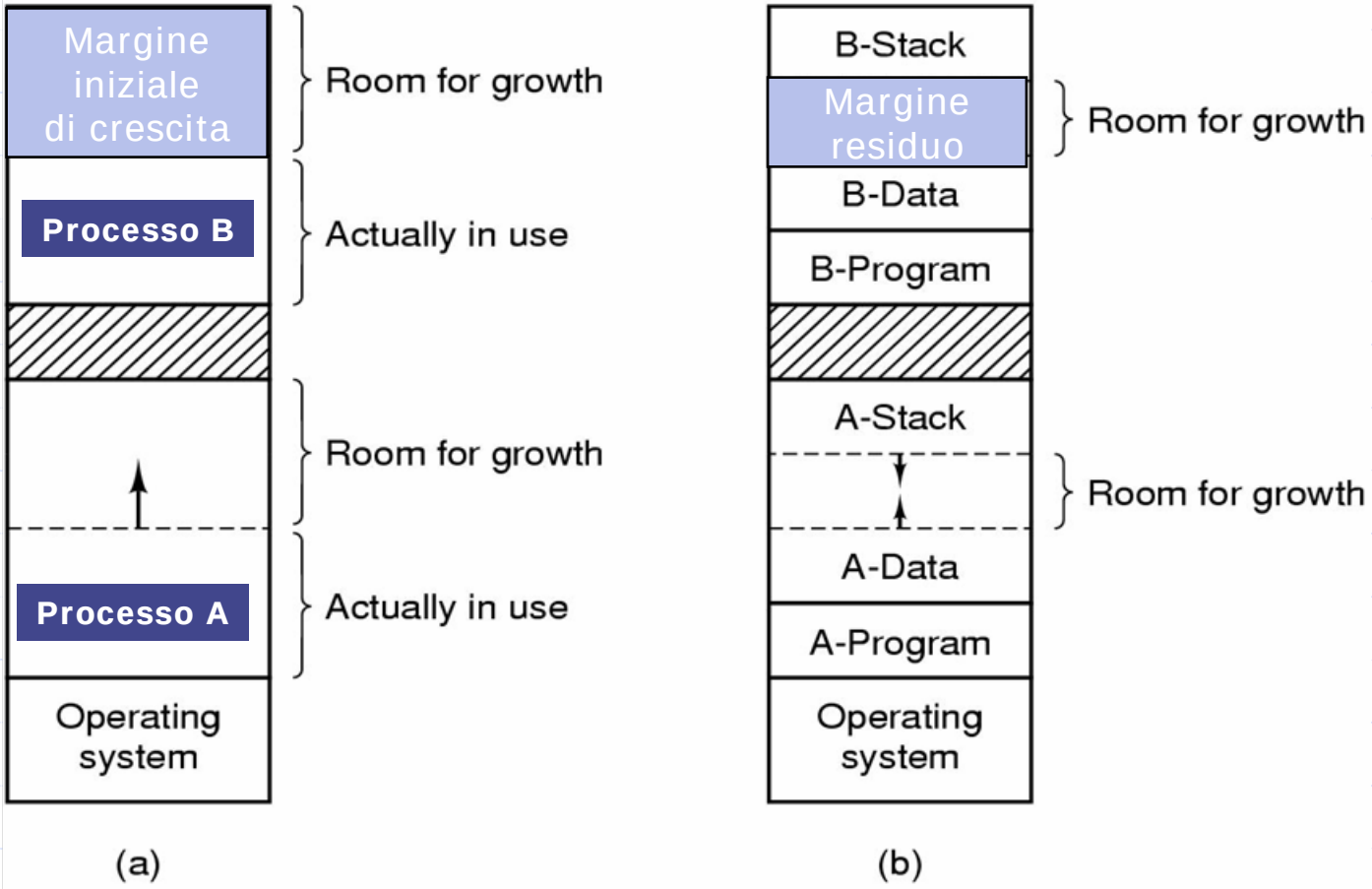
Ricapitolazione e discussione in aula:

Claudio Palazzi – *cpalazzi@math.unipd.it*

Swapping – 1


Time
C
B
A
D
Operating system
(a)
(b)
(c)
(d)
(e)
(f)
(g)
Frammentazione esterna

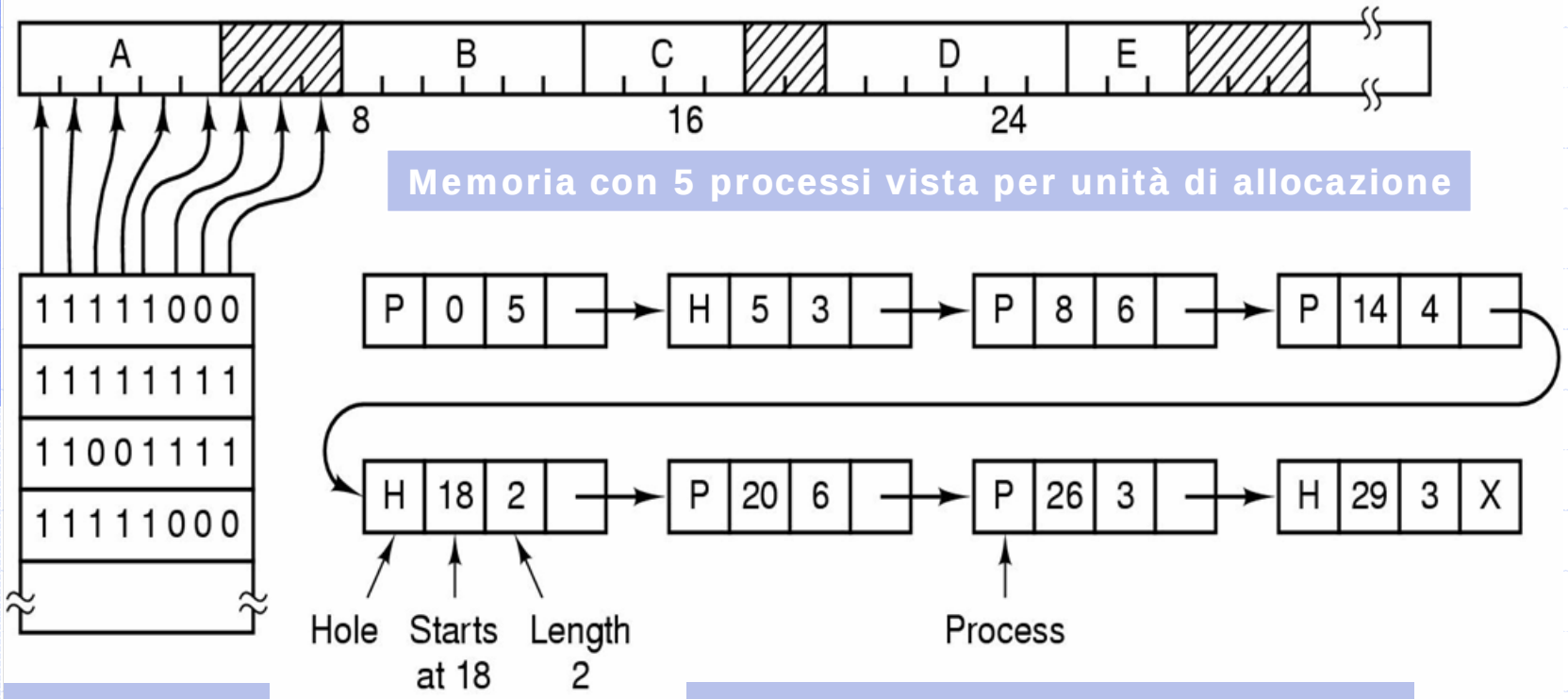
Swapping – 2


Margine iniziale di crescita
Room for growth
Processo B
Actually in use
Room for growth
Processo A
Actually in use
Operating system
(a)
B-Stack
Margine residuo
Room for growth
B-Data
B-Program
A-Stack
Room for growth
A-Data
A-Program
Operating system
(b)

Strutture di gestione – 1

- Quando la memoria principale viene allocata dinamicamente è essenziale tenere traccia del suo stato d'uso
- Due strategie principali
  - (A) **Mappe di *bit***
  - (B) **Liste collegate**

Strutture di gestione – 2



Memoria con 5 processi vista per unità di allocazione

Struttura di gestione a *bitmap*

Mappa di allocazione
(per processo o per vuoto)
meglio tenere mappe separate!

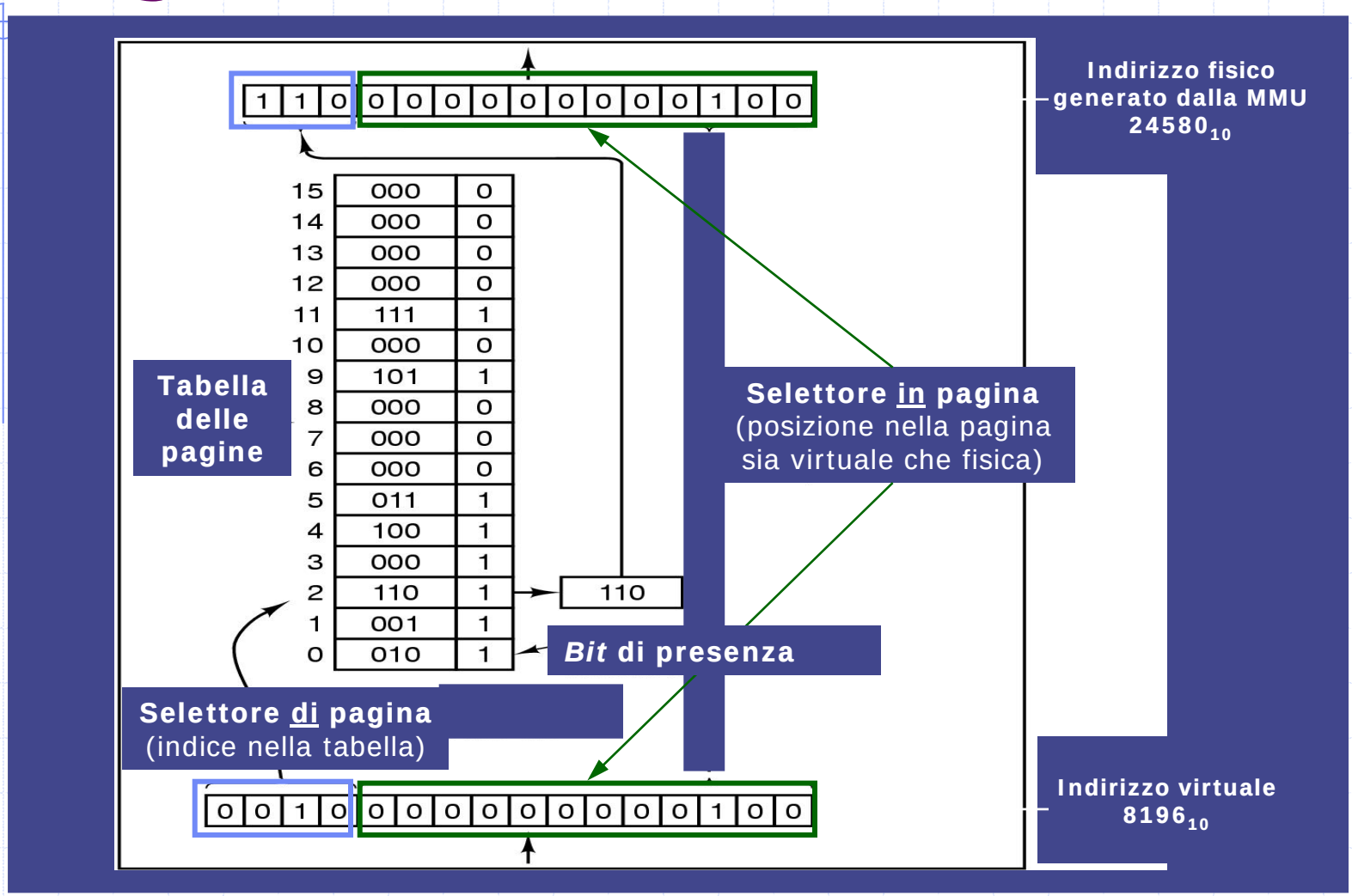
Strutture di gestione – 3

- Due strategie principali
  - (A) **Mappe di *bit***
    - Memoria vista come insieme di **unità di allocazione** (1 *bit* per unità)
      - Unità piccole → struttura di gestione grande
        - Esempio: Unità da 32 *bit* e RAM ampia 512 MB → struttura ampia 128 M *bit* = 16 MB → 3.1 % (= 1/32)
  - (B) **Liste collegate**
    - Nella sua versione più semplice la memoria è vista a segmenti
      - Segmento = processo oppure spazio libero tra processi
      - Ogni elemento di lista rappresenta un segmento
        - Ne specifica punto di inizio, ampiezza e successore
        - Liste ordinate per indirizzo di base

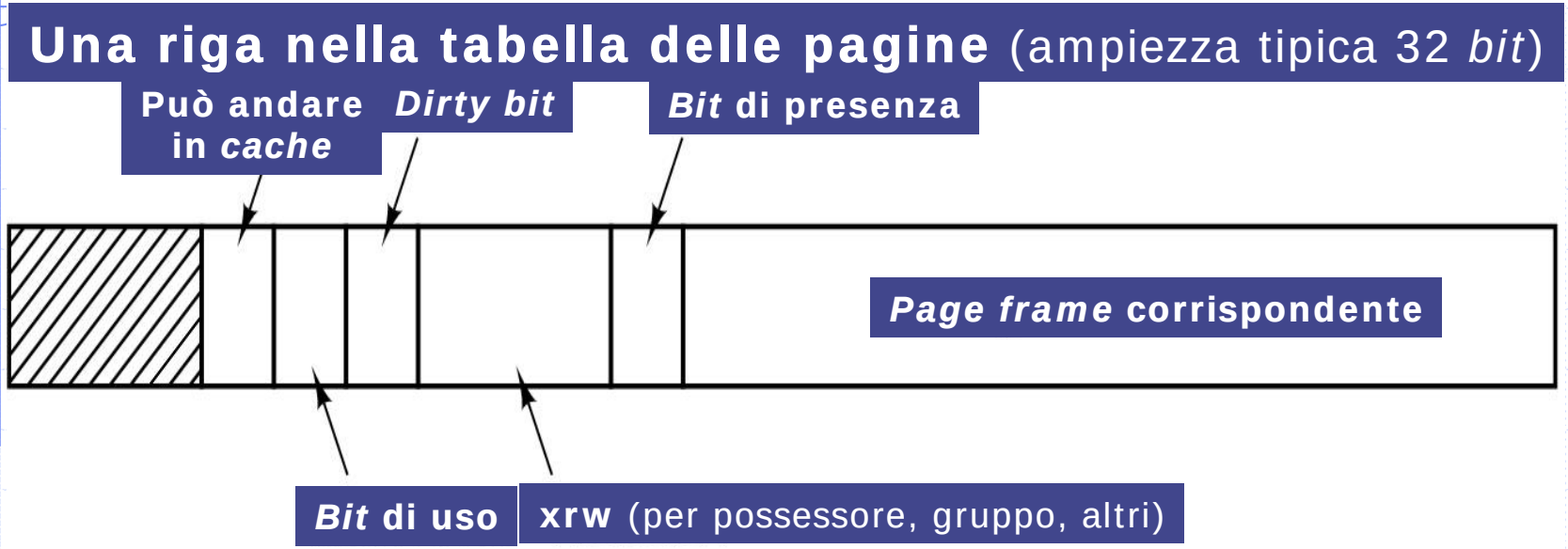
Strutture di gestione – 4

- Varie strategie di allocazione
  - ***First fit*** : il primo segmento libero ampio abbastanza
  - ***Next fit*** : come *First fit* ma cercando sempre avanti
  - ***Best fit*** : il segmento libero più adatto
  - ***Worst fit*** : sempre il segmento libero più ampio
  - ***Quick fit*** : liste diverse di ricerca per ampiezze "tipiche"

Paginazione: strutture – 1



Paginazione: strutture – 2



- L'indirizzo di disco ove la pagina si trova quando non è in RAM **non** è nella tabella!
  - La tabella delle pagine serve alla MMU (*hardware*)
  - Il caricamento della pagina da disco viene effettuato dal S/O (*software*)
  - L'informazione dell'uno **non serve** all'altro

Paginazione: rimpiazzo – 1

- **NRU** (*not recently used*)
  - Per ogni *page frame* vengono aggiornati
    - *Bit* M (*modified*), inizializzato a 0 dal S/0
    - *Bit* R (*referenced*), posto a 0 **periodicamente** dal S/O per stimare la frequenza d'uso
  - Le pagine nei *page frame* sono classificate in
    - **Classe 0**: non riferita, non modificata
    - **Classe 1**: non riferita, modificata
    - **Classe 2**: riferita, non modificata
    - **Classe 3**: riferita, modificata
  - NRU sceglie una pagina **a caso** nella classe non vuota a indice più basso

Paginazione: rimpiazzo – 2

- **FIFO**
  - Rimuove la pagina di ingresso più antico in RAM
    - Basta una lista ordinata di *page frame*
      - Ogni inserimento viene marcato in coda e la rimozione avviene dalla testa
- ***Second chance***
  - Corregge FIFO rimpiazzando solo le pagine con *bit* R = 0
    - Altrimenti il *page frame* viene considerato come appena caricato, posto in fondo alla coda e R viene posto a 0
    - Degenera in FIFO quando tutti i *page frame* siano stati recentemente riferiti

Paginazione: rimpiazzo – 3

- **Orologio**
  - Come SC ma i *page frame* sono mantenuti in una lista **circolare**
    - L'indice di ricerca si muove come una lancetta
- **LRU** (*least recently used*)
  - Necessita di *hardware* dedicato
- **Aging** (*not frequently used* modificato)
  - Realizzabile a *software*
  - Per ogni *page frame* aggiorna periodicamente un "contatore" C che cresce di più se R = 1
    - Non incrementa C con R ma gli inserisce R a sinistra
  - Approssima LRU con differenze importanti
    - Valuta solo periodicamente (a grana grossa)
    - Usando N *bit* per C perde memoria dopo N aggiornamenti

Paginazione: rimpiazzo - 4

- **WS approssimato**
  - Simile all'*Aging*
    - Ogni *page frame* in RAM ha un attributo temporale che indica se a un dato istante appare come riferita (R = 1)
      - Tale attributo prende il valore **t** del **tempo virtuale corrente** all'arrivo di un *page fault*
      - R e M sono posti a 1 dall'*hardware*
      - R è posto a 0 (se non in uso) da un controllo periodico e al *page fault*
    - Al *page fault* sono rimpiazzabili le pagine con R = 0 e valore di attributo **antecedente** all'intervallo (**t** – Δt, **t**)
      - Se all'istante **t** tutti i *page frame* avessero R = 1 verrebbe rimpiazzata una pagina scelta a caso, con M = 0
    - Nel caso peggiore bisogna scandire l'**intera** RAM!

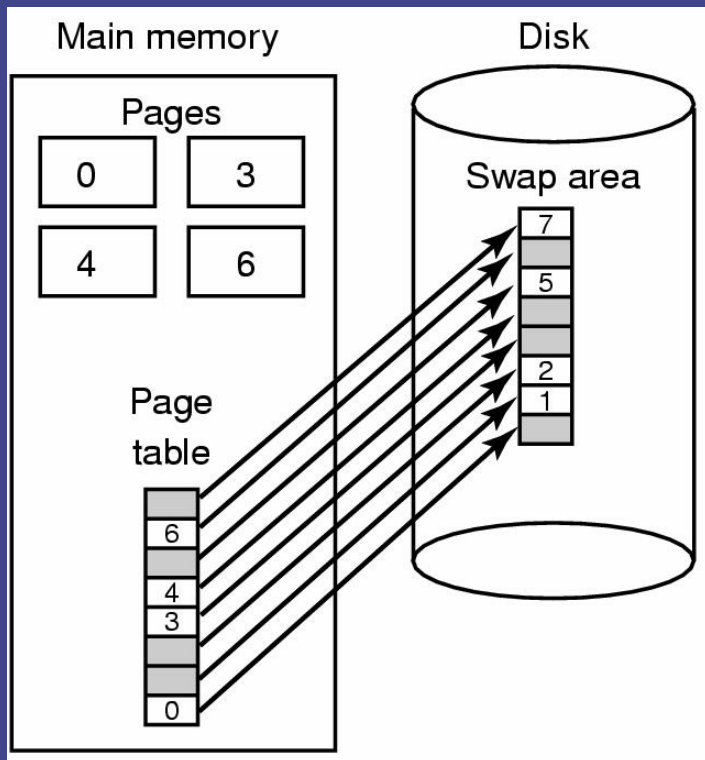
Paginazione: rimpiazzo - 5

- **WS approssimato con orologio**
  - *Page frame* organizzati in lista circolare
    - Come per l'orologio semplice
    - Ma con le informazioni del WS approssimato
  - Una "lancetta" indica il *page frame* corrente
    - Al *page fault* se R = 1 la lancetta avanza e R = 0
    - Se R = 0 si valuta l'attributo temporale
      - Se fuori da **w**(**k**,**t**) e con M = 0 allora rimpiazzo
      - Altrimenti il *page frame* va in una coda di trasferimento su disco e la lancetta avanza
        - Alla ricerca di un *page frame* rimpiazzabile direttamente
        - Quando **N** pagine in coda si trasferisce su disco
    - Se nessun *page frame* è rimpiazzabile allora si sceglie una pagina con M = 0 altrimenti quella cui punta la lancetta

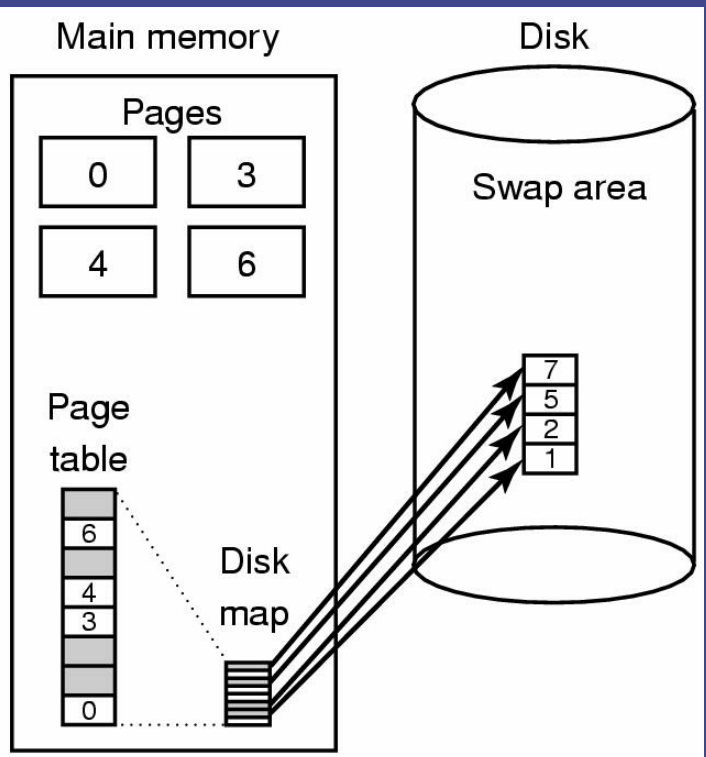
Paginazione: rimpiazzo - 6

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Paginazione: realizzazione – 1



Area di *swap* pre-assegnata e mappata automaticamente dalla tabella delle pagine

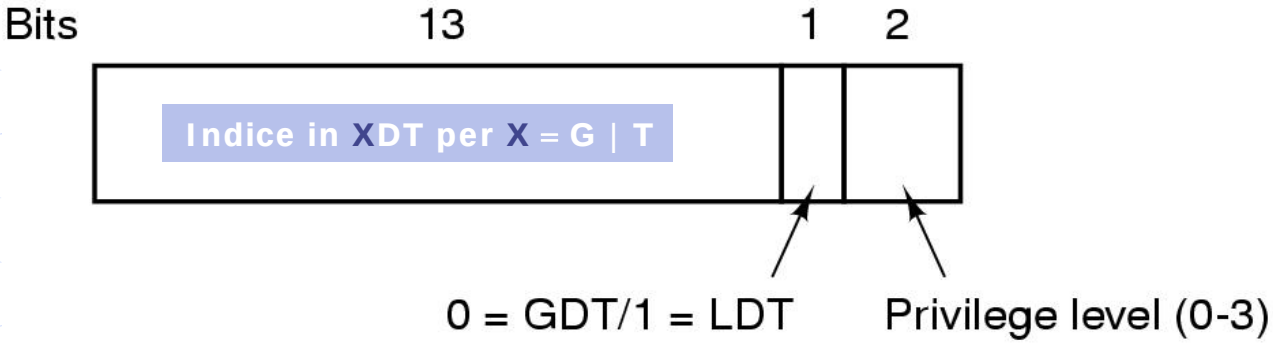


Area di *swap* assegnata a richiesta e mappata esplicitamente dalla tabella delle pagine

Segmentazione: realizzazione – 1

- Vista la grande ampiezza potenziale i segmenti sono spesso **paginati**
- Nel caso del Pentium di Intel
  - Fino a 16 K segmenti indipendenti
    - Di ampiezza massima 4 GB (32 *bit*)
  - Una LDT per processo
    - *Local Descriptor Table*
      - Descrive i segmenti del processo
  - Una singola GDT per l'intero sistema
    - *Global Descriptor Table*
      - Descrive i segmenti del S/O

Segmentazione: realizzazione – 2



- 6 registri di segmento
  - Di cui 1 denota il segmento corrente
- LDT e GDT contengono $2^{13} = 8$ K descrittori di segmento
  - I descrittori di segmento sono espressi su 8 B
    - La **base** del segmento in RAM è espressa su 32 *bit*
    - Il **limite** su 20 *bit* per verificare la legalità dell'*offset* fornito dal processo
      - Consente ampiezza massima a 1 MB (per **granularità** a B)
      - Oppure 1 M pagine da 4 KB ovvero 4 GB (per granularità a pagine)

Segmentazione: realizzazione – 3

- L'indirizzo **lineare** ottenuto da (base di segmento + *offset*) può essere interpretato come
  - Indirizzo **fisico** se il segmento considerato non è paginato
  - Indirizzo **logico** altrimenti
    - Nel qual caso il segmento viene visto come una memoria virtuale paginata e l'indirizzo come virtuale in essa
      - 10 *bit* : indice in catalogo di tabelle delle pagine
        - 2^{10} righe da 32 *bit* ciascuna (base di tabella denotata)
      - 10 *bit* : indice in tabella delle pagine selezionata
        - 2^{10} righe da 32 bit ciascuna (base di *page frame*)
      - 12 *bit* : posizione nella pagina selezionata
        - *Offset* in pagina da 4 KB

Esercizio 1

Dato un sistema di *swapping* e una memoria con zone disponibili di ampiezza: 8, 4, 14, 18, 17, 9, 12, 15 KB, in questo ordine, indicare quale area venga prescelta dalla politica ***Next Fit*** a fronte della richiesta di caricamento di un segmento di ampiezza 3 KB dopo aver caricato un segmento ampio 12 KB:

A: 4 KB

B: 18 KB

C: 14 KB

D: 9 KB.

Esercizio 1 - soluzione

Dato un sistema di *swapping* e una memoria con zone disponibili di ampiezza: 8, 4, 14, 18, 17, 9, 12, 15 KB, in questo ordine, indicare quale area venga prescelta dalla politica ***Next Fit*** a fronte della richiesta di caricamento di un segmento di ampiezza 3 KB dopo aver caricato un segmento ampio 12 KB:

A: 4 KB

B: 18 KB

C: 14 KB

D: 9 KB

E con ***First fit? Best fit? Worst fit?***

Esercizio 2

Sia data memoria dotata di 4 *page frame*, inizialmente libere, e 8 pagine di memoria virtuale. Utilizzando la politica FIFO per il rimpiazzo delle pagine, indicare quanti *page fault* si verifichino a fronte della stringa di riferimenti: 0 1 7 2 3 2 7 1 0 3:

A: 4

B: 3

C: 6

D: 2

Esercizio 2 - soluzione

Sia data memoria dotata di 4 *page frame*, inizialmente libere, e 8 pagine di memoria virtuale. Utilizzando la politica FIFO per il rimpiazzo delle pagine, indicare quanti *page fault* si verifichino a fronte della stringa di riferimenti: 0 1 4 7 3 7 4 1 0 3:

A: 4
B: 3
C: 6
D: 2

Posso decidere anche per altre politiche di rimpiazzo (NRU, *Second Chance*, LRU, ecc.)? Oppure ho bisogno di altre informazioni (in caso affermativo, quali?) ?

Esercizio 3

Si consideri la seguente serie di riferimenti a pagine di memoria:
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

Si considerino le seguenti politiche di rimpiazzo:
- LRU
- FIFO
- Optimal

Quanti *page fault* avvengono considerando un numero di *page frame* della RAM di 1, 2, 3, 4, 5, 6, 7 ?

Esercizio 3 - soluzione

# *page frame*	FIFO	LRU	*Optimal*
1	20	20	20
2	18	18	15
3	16	15	11
4	14	10	8
5	10	8	7
6	10	7	7
7	7	7	7

Esercizio 4.1

Sia dato un sistema di gestione della memoria principale basato su segmentazione con indirizzi logici e fisici espressi su 16 *bit*. Si consideri la tabella dei segmenti riportata di seguito, ove il prefisso 0x denota l'uso di notazione Esadecimale:

Segmento	Base	Limite
$0x0$	$0x0219$	$0x600$
$0x1$	$0x2300$	$0x014$
$0x2$	$0x0090$	$0x100$
$0x3$	$0x1327$	$0x580$
$0x4$	$0x1952$	$0x096$
...	...	...
$0xF$	...	...

Si mostri graficamente la mappa di memoria corrispondente, indicando anche il suo grado percentuale di occupazione.

Esercizio 4.1 - soluzione

Con un po' di calcoli:

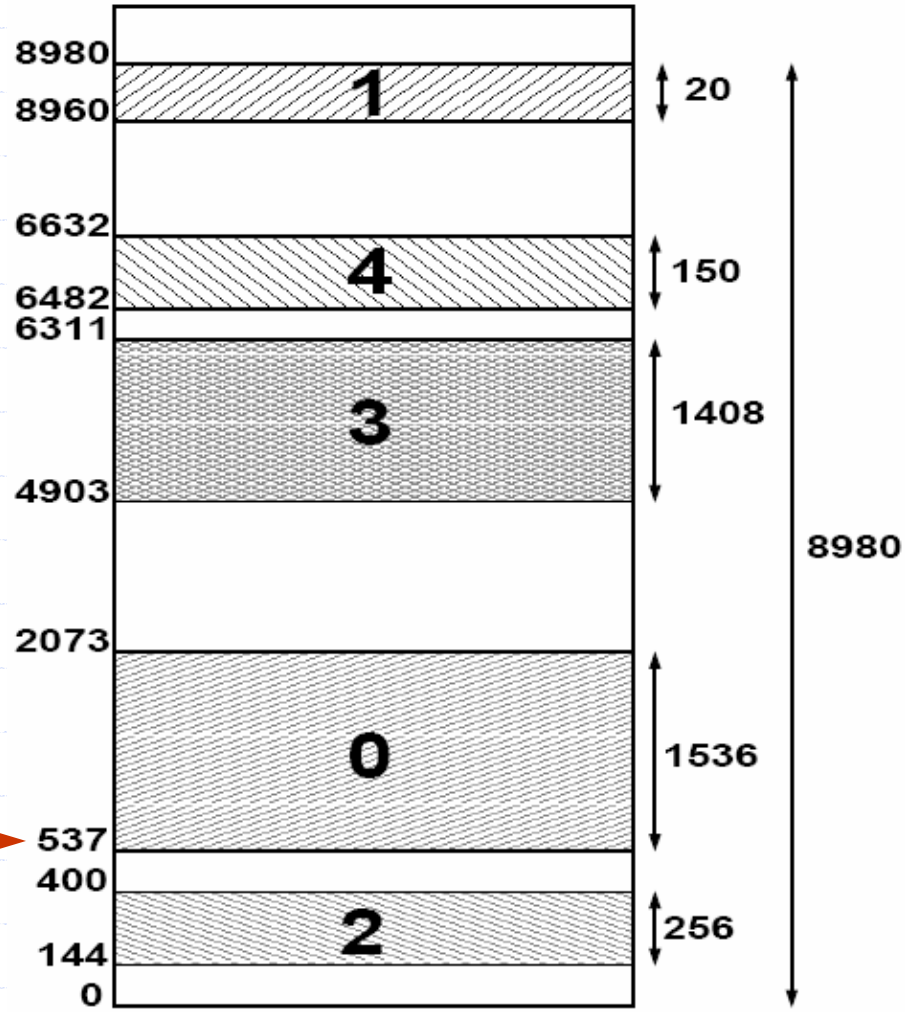
0x219 =

$2 \times 256 + 1 \times 16 + 9 \times 1 =$

537

Eccetera …

L'area di memoria occupata va dall'indirizzo 0 all'indirizzo 8980. L'ampiezza complessiva dei segmenti in tale zona misura 3370 B con grado di occupazione: **37, 53%**.



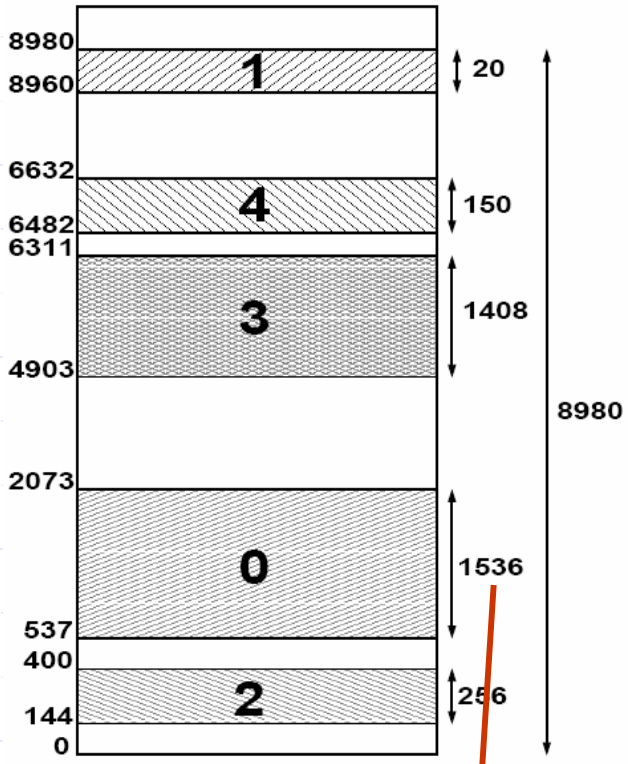
Esercizio 4.2

Si fornisca l'indirizzo fisico corrispondente ai seguenti indirizzi virtuali espressi in notazione decimale:

a.	$< 0, 1984 >$
b.	$< 1, 18 >$
c.	$< 2, 250 >$
d.	$< 3, 1400 >$
e.	$< 4, 112 >$

Esercizio 4.2 - soluzione

Ricordando che ...



Allora:

indirizzo virtuale	indirizzo fisico
$< 0, 1984 >$	Errore: indirizzo illegale.
$< 1, 18 >$	$8960 + 0018 = 8978$
$< 2, 250 >$	$0144 + 0250 = 0394$
$< 3, 1400 >$	$4903 + 1400 = 6303$
$< 4, 112 >$	$6482 + 0112 = 6594$

1984 > 1536

Esercizio 4.3

Indicare il modo nel quale i segmenti di indice 0–4 possano essere mappati su disco, assumendo blocchi di ampiezza 1 KB, calcolando anche la quantità percentuale di memoria *sprecata* di conseguenza.

Esercizio 4.3 - soluzione

Blocchi di ampiezza 1 KB = 1024 B, quindi:

Segmento	Ampiezza	# Blocchi occupati
0	1536 B	2
1	20 B	1
2	256 B	1
3	1408 B	2
4	150 B	1
Totale	3370 B	7

Totale B $\times$ Blocchi = 1024 B $\times$ 7 = 7168 B

Totale spreco = (7168 – 3370) B = 3798 B

Spreco di memoria: 3798 B / 7168 B = **53%**

Esercizio 5 (modificato dopo lezione)

Si consideri la matrice (o *array* bidimensionale):

```
int A[][] = new int[100][100];
```

Si assuma che la posizione **`A[0][0]`** di tale matrice sia posta alla locazione **200** di una memoria paginata con **3** pagine di dimensione **200** B.

Si assuma che un valore **`int`** occupi **1** B.

Si assuma che le pagine siano inizialmente tutte vuote e che il processo (il cui codice occupa esattamente **200** B) abbia la seguente esecuzione:

```
for (int i = 0; i < 100; i++){
   for (int j = 0; j < 100; j++){
      A[i][j] = 0;}}
```

Quanti *page fault* saranno generati usando LRU?

Esercizio 5 – soluzione (1/2)

La matrice `A[i][j]` è scritta linearmente in memoria:

`A[0][0],A[0][1],A[0][2],…,A[0][99],A[1][0],`
`A[1][1],…,A[99][99]`

Il processo azzera le celle della matrice proprio nel loro ordine di memorizzazione.

Quindi inizialmente verranno caricati nelle pagine
Pagina 0: Il programma (che occupa esattamente 200 B)
Pagina 1: Celle da A[0][0] a A[1][99] incluse
Pagina 2: Celle da A[2][0] a A[3][99] incluse

Esercizio 5 – soluzione (2/2)

A ogni iterazione del programma la memoria viene acceduta per leggere l'istruzione successiva
→ la pagina relativa al processo verrà continuamente acceduta

I primi 200 azzeramenti faranno accesso a Pagina 1, i successivi 200 a Pagina 2; l'ulteriore azzeramento (cella `A[4][0]`) causerà un *page fault* ; la pagina usata meno recentemente è Pagina 1 che verrà sostituita con le celle da `A[4][0]` a `A[5][99]` incluse.
Arrivati all'azzeramento di `A[6][0]` sarà Pagina 2 ad essere stata usata meno di recente
E così via, causando in tutto 1 *page fault* iniziale per caricare il processo in Pagina 0 e 50 *page fault* di Pagina 1 e 2 (25 ciascuno) per caricare le porzioni di matrice.

TOTALE = 51 *page fault*

Il *file system*: esercizi

Discussione in aula:

Claudio Palazzi – *cpalazzi@math.unipd.it*

Appello AE-2 del 19/3/2003

Quesito 3. Un *file system* utilizza un vettore di *bit* (*bitmap*), con posizioni numerate da sinistra verso destra, per indicare i blocchi liberi presenti in una data partizione di disco. La formattazione della partizione libera tutti i blocchi tranne il 1°, che viene assegnato alla *directory* radice. In tale sistema, l'assegnazione di blocchi liberi a *file* considera sempre prima i blocchi di indice minore, trascurando ogni considerazione di contiguità. Si mostri il contenuto della parte iniziale di tale vettore dopo ciascuna delle seguenti operazioni:

- Scrittura del *file* A, ampio 6 blocchi
- Scrittura del *file* B, ampio 5 blocchi
- Rimozione del *file* A
- Scrittura del *file* C, ampio 8 blocchi
- Rimozione del *file* B

Si mostri poi l'evoluzione del contenuto di tale vettore a fronte della medesima sequenza di operazioni nel caso in cui il sistema volesse assicurare la massima contiguità dei blocchi assegnati ad ogni *file*.

Soluzione

Soluzione 3 (punti 5). Seguiamo l'evoluzione del contenuto della maschera nel primo caso, concentrandoci sulle sue prime posizioni:

Dopo la formattazione:	1000000000000 ...
Scrittura del *file* A:	1111111000000 ...
Scrittura del *file* B:	1111111111000 ...
Rimozione del *file* A:	1000001111000 ...
Scrittura del *file* C:	1111111111110 ...
Rimozione del *file* B:	1111111000110 ...

Vediamone ora l'evoluzione nel secondo caso, nel quale prevalgono considerazioni di contiguità dei blocchi assegnati ai *file*:

Dopo la formattazione:	1000000000000 ...	
Scrittura del *file* A:	1111111000000 ...	allocazione contigua senza frammentazione esterna
Scrittura del *file* B:	1111111111000 ...	allocazione contigua senza frammentazione esterna
Rimozione del *file* A:	1000001111000 ...	
Scrittura del *file* C:	1000001111111111110 ...	allocazione contigua con frammentazione esterna
Rimozione del *file* B:	1000000000111111110 ...	

Appello AE-2 del 19/3/2003

Quesito 4. Il progettista di un sistema operativo ha deciso di usare nodi indice (*i-node*) per la realizzazione del proprio *file system*, stabilendo che essi abbiano la stessa dimensione di un blocco, fissata a 512 *byte*. Il progettista ha poi deciso che un nodo indice primario contenga 12 campi di indirizzo di blocchi di disco e 2 campi puntatori a nodi indice di primo e secondo livello di indirezione rispettivamente. Sapendo che gli indirizzi di blocco sono espressi su 32 *bit*, si vuole allocare un *file* logicamente composto da 10.000 *record* da 80 *byte* ciascuno, imponendo che un *record* non possa essere suddiviso su due blocchi. Calcolare quanti blocchi verranno utilizzati per allocare il *file* dati e quanti per gestire la sua allocazione tramite nodi indice. Determinare inoltre l'occupazione totale in memoria secondaria risultante da tale strategia di allocazione.

Soluzione

Soluzione 4 (punti 5). Richiamiamo i dati del problema:
N_R = numero di *record* che compongono il *file* = 10.000
D_R = dimensione di un *record* = 80 *byte*
D_I = dimensione di un indirizzo = 4 *byte*
D_B = dimensione di un blocco = 512 *byte*

N_{RB} = numero di *record* per blocco = intD_B/DR) = int512/80) = int6,4) = 6
N_{BF} = numero di blocchi occupati dal *file* = 1+intN_R/N_{RB}) = 1+int10000/6) = 1.667
N_{IB} = numero di indirizzi in un blocco = D_B/D_I = 512/4 = 128
N_{ID} = numero di indirizzi in X blocchi a doppia indirezione = X^2

I blocchi da indirizzare sono N_{BF} = 1.667

- di questi, 12 possono essere indirizzati direttamente dal nodo indice principale
- dei rimanenti 1.667-12 = 1.655, N_{IB} (cioè 128) sono indirizzabili ad indirezione singola tramite l'indirizzo ad indirezione singola del nodo indice principale
- dei rimanenti 1.655-128 = 1.527, si possono utilizzare blocchi indiretti di 128 con indirezione doppia, come mostrato in figura.

Con le indicazioni riportate in figura possiamo concludere che:

- per allocare il *file* dati sono necessari NBF blocchi, cioè 1.667 blocchi
- per gestire l'allocazione del *file* sono necessari:
  - 1 blocco per il nodo indice principale
  - 1 blocco di indirizzi ad indirezione singola
  - 1+12 blocchi per l'indirezione doppia

  per un totale di 1+1+1+12 = 15 blocchi
- l'occupazione totale in memoria secondaria vale 1.667 + 15 = 1.682 blocchi da 512 *byte*, per un totale di 1.682 o 512 = 861.184 = *byte* = 841 kB

Appello AE-2 del 14/9/2005

Quesito 1 (punti 8). Sia data una memoria secondaria di ampiezza 64 GB, organizzata in blocchi di ampiezza 1 kB. Dopo aver calcolato la dimensione minima di un indice di blocco per tale memoria, sotto il vincolo che essa debba essere un multiplo di 8 (*bit*), si determini la dimensione massima di *file* ottenibile nel caso pessimo di contiguità nulla sotto le seguenti ipotesi:

1. *file system* di tipo NTFS, con *record* ampi 1 kB, 408 B riservati all'attributo dati nel *record* principale ed 800 B nei *record* di estensione, utilizzando esattamente 2 *record*;
2. *file system* di tipo Extfs, con *i-node* ampi 1 kB, nodo principale contenente 16 indici di blocco ed 1 indice di I e di II indirezione, utilizzando l'intero *i-node* principale.

Calcolate le dimensioni richieste, si determini per ciascun tipo di *file system*, il rapporto inflattivo determinato dalla sua organizzazione strutturale, ossia l'onere proporzionale dovuto alla memorizzazione delle strutture di rappresentazione rispetto a quella dei dati veri e propri.

Soluzione

Soluzione 1 (punti 8). Essendo la memoria secondaria ampia 64 GB e i blocchi ampi 1 kB, è immediato calcolare che siano necessari $\lceil \frac{64\ GB}{1\ kB} \rceil = 64\ M = 2^6 \times 2^{20} = 2^{26}$ indici, la cui rappresentazione binaria banalmente richiede 26 *bit*. Stante il vincolo che la dimensione dell'indice debba essere un multiplo di 8 *bit*, la dimensione dell'indice deve salire a 32 *bit* (4 B). Vediamo ora quale possa essere la dimensione massima di *file* ottenibile sotto le ipotesi fissate dal quesito.

***File system* di tipo** NTFS : Dei 408 B riservati all'attributo dati nel *record* principale, $2 \times 4 = 8$ B saranno riservati alla coppia {base, indice}, mentre i rimanenti $408 - 8 = 400$ B potranno essere utilizzati per indicare le sequenze contigue di caso peggiore (dunque tutte ampie 1 blocco). Poiché ciascuna sequenza richiede una coppia di indici {inizio, fine}, pari a $2 \times 4 = 8$ B, il *record* principale potrà ospitare $\lfloor \frac{400\ B}{8\ B} \rfloor = 50$ sequenze ampie 1 blocco. Il *record* di estensione dispone invece di 800 B per la memorizzazione di $\lfloor \frac{800\ B}{8\ B} \rfloor = 100$ ulteriori sequenze. Ne segue che, sotto le ipotesi del quesito, la dimensione massima di *file* consentita da NTFS è pari a: $(50 + 100)$ blocchi $\times$ 1kB/blocco $= 150$ kB, al costo di 2 *record*, ciascuno ampio 1 kB. Il rapporto inflattivo in questo caso è dunque pari a: $\frac{2\ kB}{150\ kB} = 1.33\%$.

***file system* di tipo** Extfs : In questo caso, utilizzando tutti i campi dell'*i-node* principale, abbiamo a disposizione:

- 16 indici diretti di blocco, al costo di 1 blocco poiché un *i-node* occupa lo stesso spazio di un blocco;
- 1 indice di I indirezione, il quale punta ad un *i-node* interamente utilizzato per contenere indici diretti di blocco, che consente di esprimere fino a: $\lfloor \frac{1\ kB}{4\ B} \rfloor = \lfloor \frac{2^{10}}{2^2} \rfloor = 2^8 = 256$ indici di blocco, al costo di 1 ulteriore blocco;
- 1 indice di II indirezione, il quale punta ad un *i-node* speciale, interamente utilizzato per contenere puntatori ad *i-node* di I livello, che dunque consente di esprimere 256 puntatori a strutture ciascuna contenente fino a 256 indici diretti di blocco, per un totale di $256^2 = (2^8)^2 = 2^{16} = 65.536$ blocchi, al costo di $1 + 256 = 257$ ulteriori blocchi.

Conseguentemente, Extfs consente di rappresentare *file* di dimensione massima pari a: $(16 + 256 + 65.536) \times 1\ kB = 65.808\ kB = 64\ MB + 272\ kB$ (438,72 volte maggiore di quanto ottenuto con NTFS) per un rapporto inflattivo pari a: $\frac{(1+1+257) \times 1\ kB}{65.808\ kB} = 0,39\%$ (3,4 volte inferiore a quanto ottenuto con NTFS).

La considerazione ovvia da trarre da queste considerazioni è che NTFS) è particolarmente penalizzato dalle condizioni di scarsa o nulla contiguità. (Per contro, ad Extfs la contiguità non giova in alcun modo.)

Accesso al Disco

Sia dato un disco organizzato in blocchi di dimensione fissa avente le seguenti caratteristiche: ogni traccia del disco è composta da 100 KB; il numero di giri al minuto (RPM) è 7200; il *seek time* medio è di 10 ms. Si assuma che tutti i *file* salvati abbiano dimensione 2 KB.
Considerando blocchi di dimensione, rispettivamente, 1 KB, 2 KB, 4 KB, calcolare:

1) lo spazio su disco sprecato per ogni blocco, qualora i blocchi abbiano dimensione, rispettivamente, di 1 KB, 2 KB, 4 KB;

2) il tempo medio necessario per copiare un blocco in memoria considerando ininfluente il tempo di trasferimento fisico dei dati sul *bus*

Accesso al Disco – Soluzione – 1

1) Lo spazio su disco sprecato per ogni blocco equivale a:
   caso dim. blocco = 1 KB: spreco = 0 KB
   caso dim. blocco = 2 KB: spreco = 0 KB
   caso dim. blocco = 4 KB: spreco = 2 KB

2) Il tempo richiesto è dato dalla somma del *seek time*, più il tempo di rotazione del disco per far sì che la testina si posizioni all'inizio del blocco da copiare, più il tempo per copiare il blocco su *bus* (il trasferimento fisico sul *bus* è, per assunzione, ininfluente)
   tempo di *seek* = 10 ms (per assunzione)
   tempo **medio** di rotazione per posizionare la testina =
   = [tempo di rotazione completa] / 2 =
   = [1 / (7200 rpm / 60 s/m)] / 2 ≈ 4,167 ms

Accesso al Disco – Soluzione – 2

2) (cont.)

tempo per copiare **un** blocco su *bus* =
[tempo per copiare 1 B] × dimensione blocco in B =
= [8,33 ms / 100 KB] × dimensione blocco in B

dunque:
- blocco 1 KB: 0,0833 ms (ma servono 2 blocchi per file di 2 KB)
- blocco 2 KB: 0,1666 ms (basta 1 blocco per file di 2 KB)
- blocco 4 KB: 0,3332 ms (basta, e avanza, 1 blocco per file di 2 KB)

TOTALE tempo necessario:

blocco 1 KB: 10 ms + 4,167 ms + (2 * 0,0833) ms = **14,3336 ms**
blocco 2 KB: 10 ms + 4,167 ms + 0,1666 ms = **14,3336 ms**
blocco 4 KB: 10 ms + 4,167 ms + 0,3332 ms = **14,5002 ms**

Esercizi di Ricapitolazione

Discussione in aula:

Claudio Palazzi – *cpalazzi@math.unipd.it*

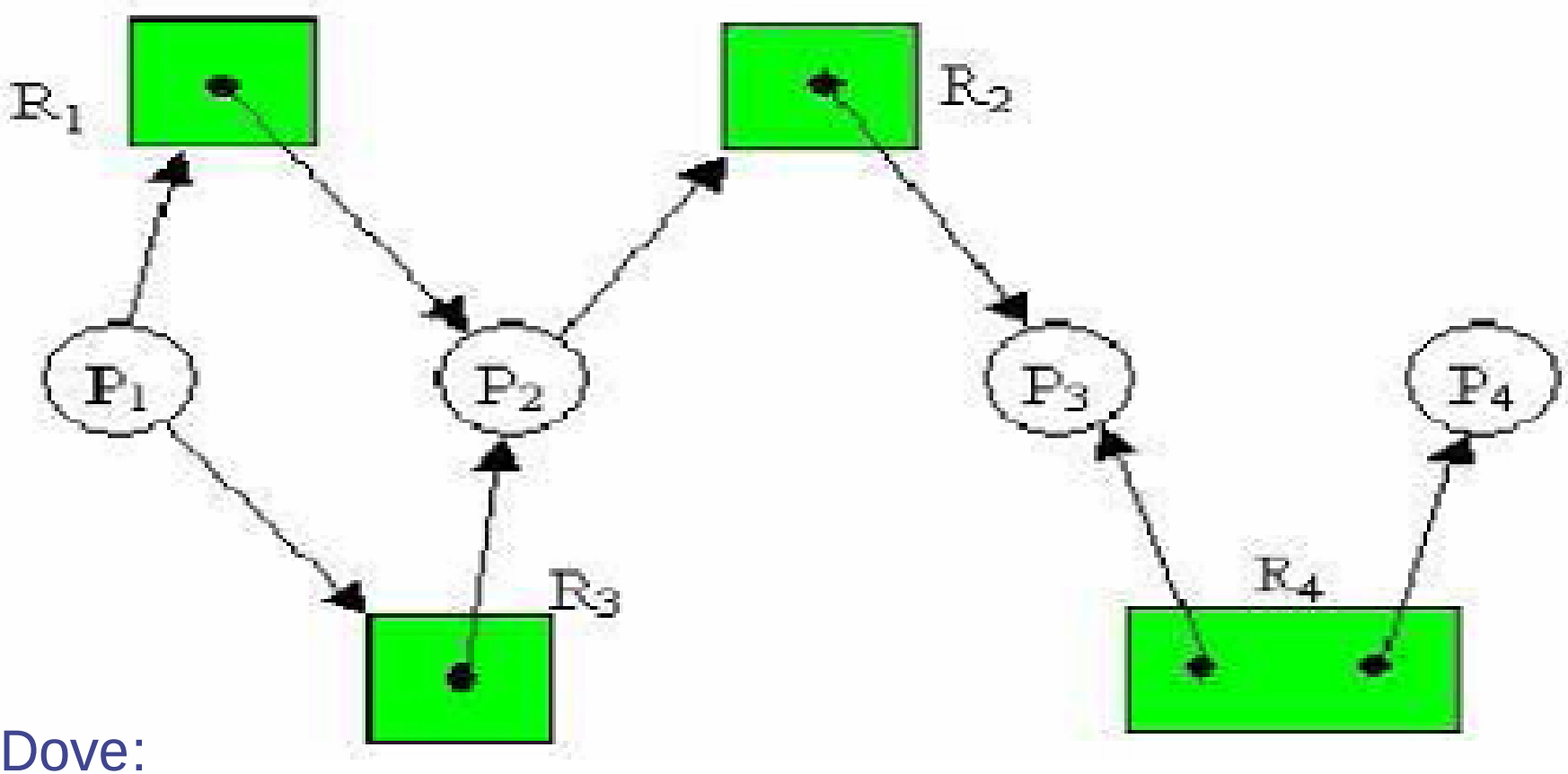
Appello AE-2 del 5/4/2005

Sia dato il sistema descritto dalla seguente rappresentazione insiemistica delle assegnazioni di risorsa, dove *P* denota l'insieme dei processi, *R* l'insieme delle risorse R_i^j di indice i e molteplicità j, *E* l'insieme delle richieste di accesso di processi in *P* a risorse in *R*:

$$
\begin{aligned}
P &= \{P_1, P_2, P_3, P_4\} \\
R &= \{R_1^1, R_2^1, R_3^1, R_4^2\} \\
E &= \{P_1 \rightarrow R_1, P_1 \rightarrow R_3, P_2 \rightarrow R_2, \\
&\quad R_1 \rightarrow P_2, R_2 \rightarrow P_3, R_3 \rightarrow P_2, R_4 \rightarrow P_3, R_4 \rightarrow P_4\}
\end{aligned}
$$

a) Verificare se il sistema si trova in condizioni di stallo
b) Verificare se la situazione in a) cambia se P_3 richiede R_3

Soluzione – Grafo di Allocazione



Dove:

- P1 richiede R1 e R3
- P2 possiede R1 e R3 e richiede R2
- P3 possiede R2 e R4
- P4 possiede R4

Soluzione – Grafo di Allocazione

a) Per verificare se il sistema sia in stallo si verifica la presenza di eventuali percorsi “chiusi”.
   In questo caso non ve ne sono e dunque il sistema NON e’ in stallo.

b) Viene creato il percorso chiuso P2, R2, P3, R3 che comprende solo risorse unarie (stallo per P2 e P3). Siccome P2 e’ in stallo, non riesce a concludere e a liberare R1 ed R3 e quindi anche P1 entra in stallo. P4 può invece concludere.

Appello S0 del 13/12/2006

Sia dato il sistema con *P* insieme dei processi, *R* insieme delle risorse R_i^j di indice i e molteplicità j, *E(P)* insieme delle richieste di accesso a risorse in *R* emesse da processi in *P* e **attualmente pendenti**, e *E(R)* insieme degli accessi **attualmente soddisfatti** di processi in *P* a risorse in *R*:

$$\mathcal{P} = \{P_1; P_2; P_3; P_4; P_5; P_6\}$$

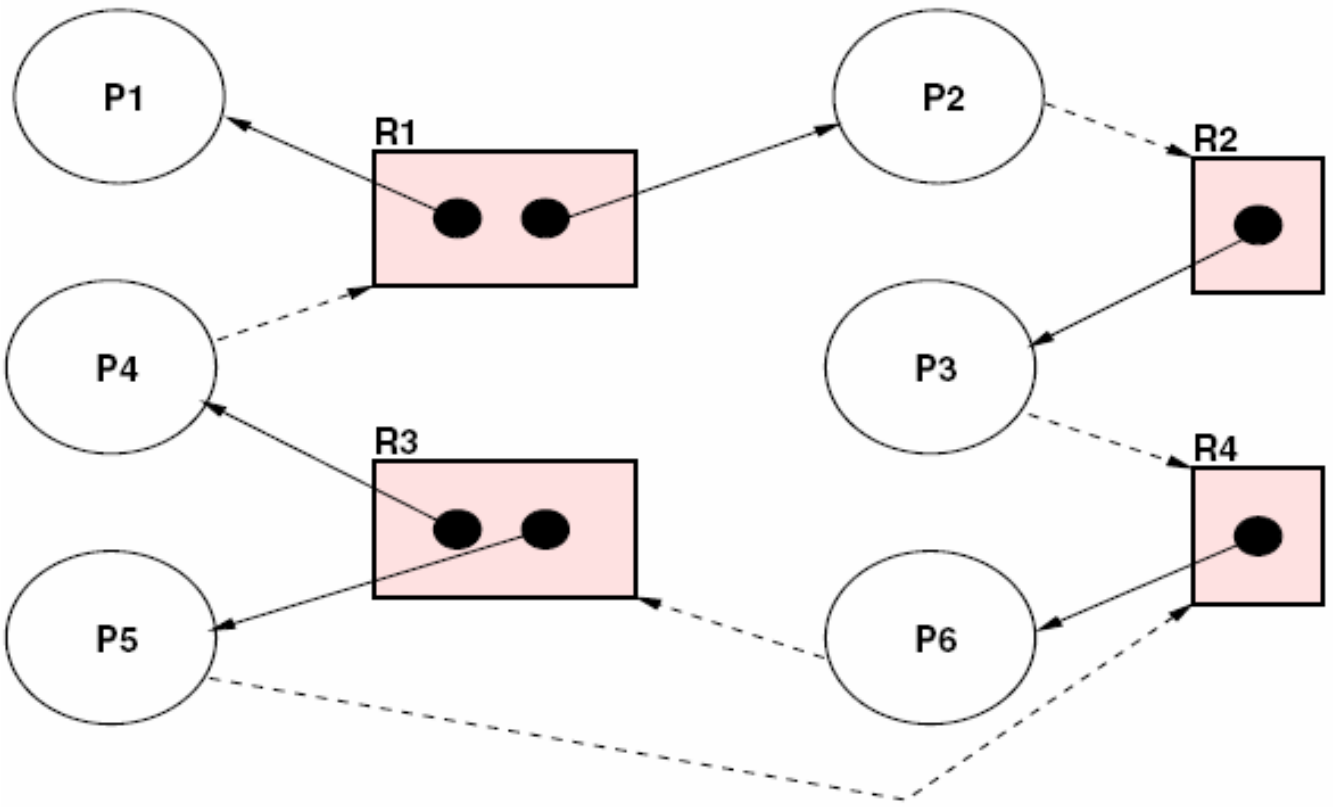
$$\mathcal{R} = \{R_1^2; R_2^1; R_3^2; R_4^1\}$$

$$\mathcal{E}(\mathcal{P}) = \{P_2 \rightarrow R_2; P_3 \rightarrow R_4; P_4 \rightarrow R_1; P_5 \rightarrow R_4; P_6 \rightarrow R_3\}$$

$$\mathcal{E}(\mathcal{R}) = \{R_1 \rightarrow (P_1, P_2); R_2 \rightarrow P_3; R_3 \rightarrow (P_4, P_5); R_4 \rightarrow P_6\}$$

a) si analizzi il grafo di allocazione delle risorse, determinando se il sistema i trovi attualmente in situazione di stallo o meno.
b) si studi l'evoluzione successiva dello stato del sistema se il processo P1 richiedesse la risorsa R2.

Soluzione – Grafo di Allocazione



a) Ci sono due percorsi chiusi:
   - P5, R4, P6, R3, P5 **(percorso 1)**
   - P2, R2, P3, R4, P6, R3, P4, R1, P2 **(percorso 2)**

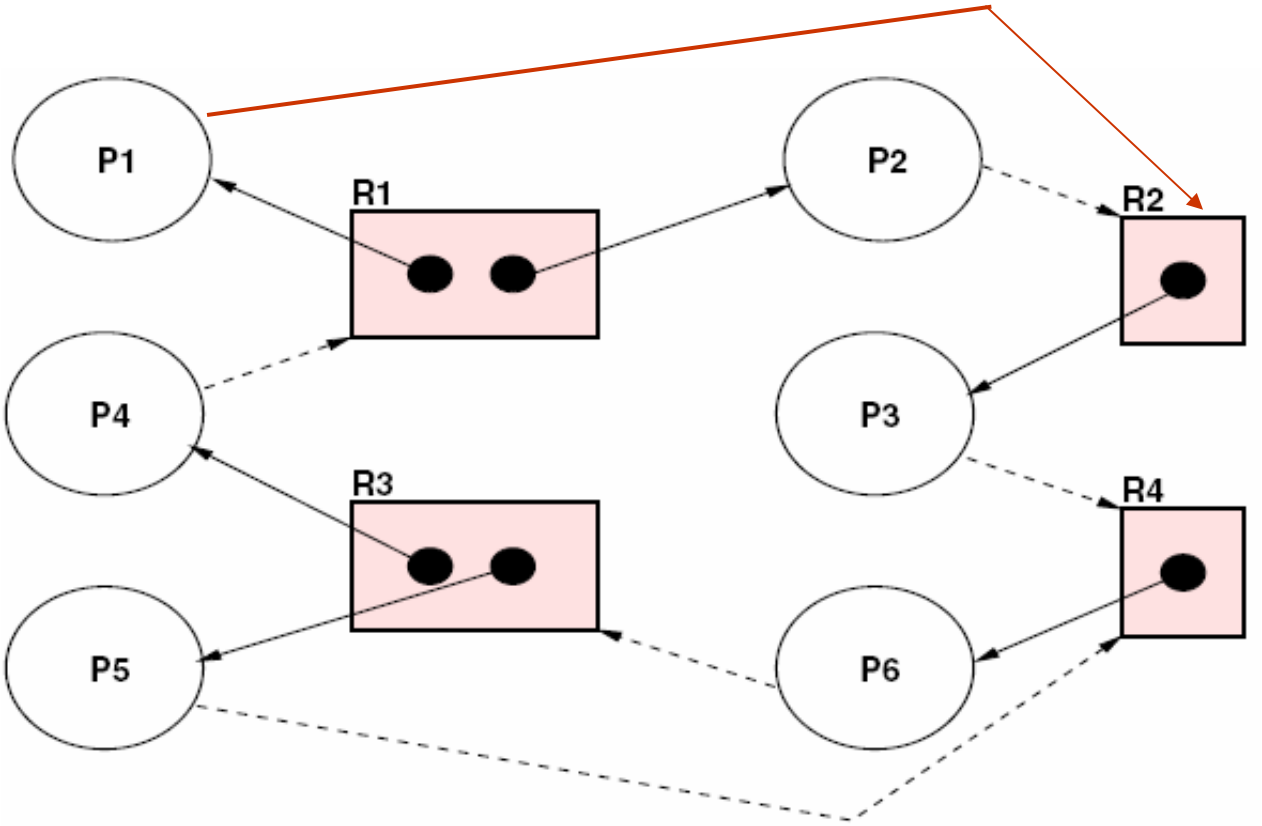
   ma in entrambi è presente risorsa completamente assegnata ma di molteplicità > 1, con una risorsa che potrebbe liberarsi (non posso quindi trarre conclusioni a priori)

Soluzione – Grafo di Allocazione

a) **Continuo..** Analizzando in dettaglio:
Il percorso 1 condivide risorsa R3 (a molteplicità > 1) con il percorso 2. Quindi sono entrambi bloccati o non-bloccati. Lo stallo del percorso 1 dipende dall'evoluzione dello stato di R3, la quale però fa parte anche del percorso 2 e dipende dalla sua evoluzione. Il percorso 2 dipende anche dalla risorsa R1 (a molteplicità >1). R1 è in possesso anche del processo P1, che non fa parte di dei due percorsi chiusi. P1 può quindi proseguire e terminare la sua esecuzione, liberando poi R1, che sarà quindi acceduta da P4 evitando lo stallo (graficamente, la freccia tra P4 ed R1 cambia verso). P4 può quindi terminare e liberare R1 ed R3. In particolare, R3 potrà poi essere acceduta da P6, il quale avanzerà fino a completamento e liberando R4. P5 potrà così accedere ad R4 e completarsi. Conclusione: **non c'e' stallo**.

Soluzione – Grafo di Allocazione

b) Qualora P1 richiedesse R2, allora si crea un altro circuito chiuso (quale?). P1 non riesce a concludere e a liberare R1 e ad aprire in cascata i circuiti chiusi.
Conclusione: **c'e' stallo**.



Appello SO del 18/7/2007

Si consideri un *file system* residente su una partizione di disco con dimensione dei blocchi logici e fisici di 512 B, dimensione dei *file* non superiori a 512 blocchi, e con tutte le informazioni su ciascun *file* già presenti in memoria principale. Per ciascuno dei tre metodi di allocazione visti a lezione (contigua, concatenata, indicizzata):

a) si illustri come gli indirizzi logici vengono fatti corrispondere agli indirizzi fisici
b) assumendo che l'ultimo accesso sia stato fatto al blocco logico 10, si determini quanti blocchi fisici debbano essere letti dal disco per accedere al blocco logico 4.

Soluzione

Allocazione contigua : il *file* è denotato dall'indice del primo blocco fisico e dalla sua ampiezza in blocchi; vista la corrispondenza di ampiezza tra blocchi logici e fisici, ogni posizione interna al *file* (blocco logico e *offset* in esso) ha una corrispondenza diretta sul disco (blocco fisico e *offset*).

Allocazione concatenata : il *file* è denotato dagli indici del primo e dell'ultimo blocco fisico; una parte dei dati di ogni blocco contiene il puntatore al blocco successivo. La posizione interna al *file* espressa in (blocco logico i, *offset* o) viene dunque tradotta mediante l'attraversamento di i posizioni nella lista concatenata a partire dalla testa.

Soluzione

Allocazione indicizzata : il *file* è denotato da un blocco speciale (detto appunto “indice”), che contiene gli indici dei blocchi fisici ove risiedono i dati. La posizione interna al *file* espressa in (blocco logico i, *offset* o) viene dunque tradotta localizzando il blocco fisico denotato dalla posizione i entro il blocco indice e la posizione o al suo interno. (Come noto, il blocco indice può essere realizzato come una tabella concatenata, tipo FAT, oppure come un blocco contiguo dedicato, tipo *i-node*.)

Esercizi di Ricapitolazione Parte II

Discussione in aula:

Claudio Palazzi – *cpalazzi@math.unipd.it*

Appello AE-2 del 14/9/2005

Quesito 1 (punti 8). Sia data una memoria secondaria di ampiezza 64 GB, organizzata in blocchi di ampiezza 1 kB. Dopo aver calcolato la dimensione minima di un indice di blocco per tale memoria, sotto il vincolo che essa debba essere un multiplo di 8 (*bit*), si determini la dimensione massima di *file* ottenibile nel caso pessimo di contiguità nulla sotto le seguenti ipotesi:

1. *file system* di tipo NTFS, con *record* ampi 1 kB, 408 B riservati all'attributo dati nel *record* principale ed 800 B nei *record* di estensione, utilizzando esattamente 2 *record*;
2. *file system* di tipo Extfs, con *i-node* ampi 1 kB, nodo principale contenente 16 indici di blocco ed 1 indice di I e di II indirezione, utilizzando l'intero *i-node* principale.

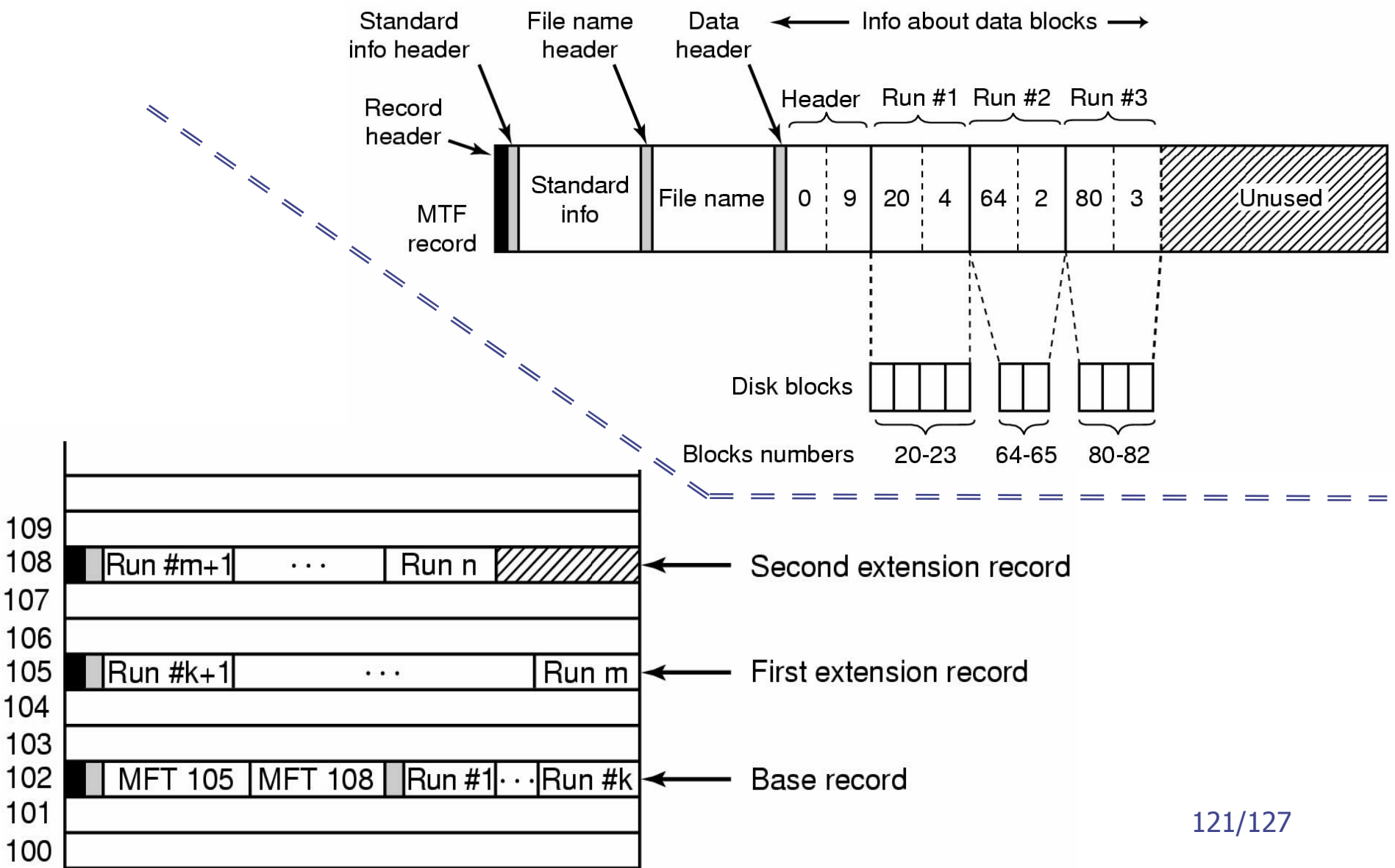
Calcolate le dimensioni richieste, si determini per ciascun tipo di *file system*, il rapporto inflattivo determinato dalla sua organizzazione strutturale, ossia l'onere proporzionale dovuto alla memorizzazione delle strutture di rappresentazione rispetto a quella dei dati veri e propri.

Soluzione

Soluzione 1 (punti 8). Essendo la memoria secondaria ampia 64 GB e i blocchi ampi 1 kB, è immediato calcolare che siano necessari $\lceil \frac{64\ GB}{1\ kB} \rceil = 64\ M = 2^6 \times 2^{20} = 2^{26}$ indici, la cui rappresentazione binaria banalmente richiede 26 *bit*. Stante il vincolo che la dimensione dell'indice debba essere un multiplo di 8 *bit*, la dimensione dell'indice deve salire a 32 *bit* (4 B). Vediamo ora quale possa essere la dimensione massima di *file* ottenibile sotto le ipotesi fissate dal quesito.

***File system* di tipo** NTFS : Dei 408 B riservati all'attributo dati nel *record* principale, $2 \times 4 = 8$ B saranno riservati alla coppia {base, indice}, mentre i rimanenti $408 - 8 = 400$ B potranno essere utilizzati per indicare le sequenze contigue di caso peggiore (dunque tutte ampie 1 blocco). Poiché ciascuna sequenza richiede una coppia di indici {inizio, fine}, pari a $2 \times 4 = 8$ B, il *record* principale potrà ospitare $\lfloor \frac{400\ B}{8\ B} \rfloor = 50$ sequenze ampie 1 blocco. Il *record* di estensione dispone invece di 800 B per la memorizzazione di $\lfloor \frac{800\ B}{8\ B} \rfloor = 100$ ulteriori sequenze. Ne segue che, sotto le ipotesi del quesito, la dimensione massima di *file* consentita da NTFS è pari a: $(50 + 100)$ blocchi $\times$ 1kB/blocco = 150 kB, al costo di 2 *record*, ciascuno ampio 1 kB. Il rapporto inflattivo in questo caso è dunque pari a: $\frac{2\ kB}{150\ kB} = 1.33\%$.

Soluzione (descrizione *record* MFT)



Soluzione

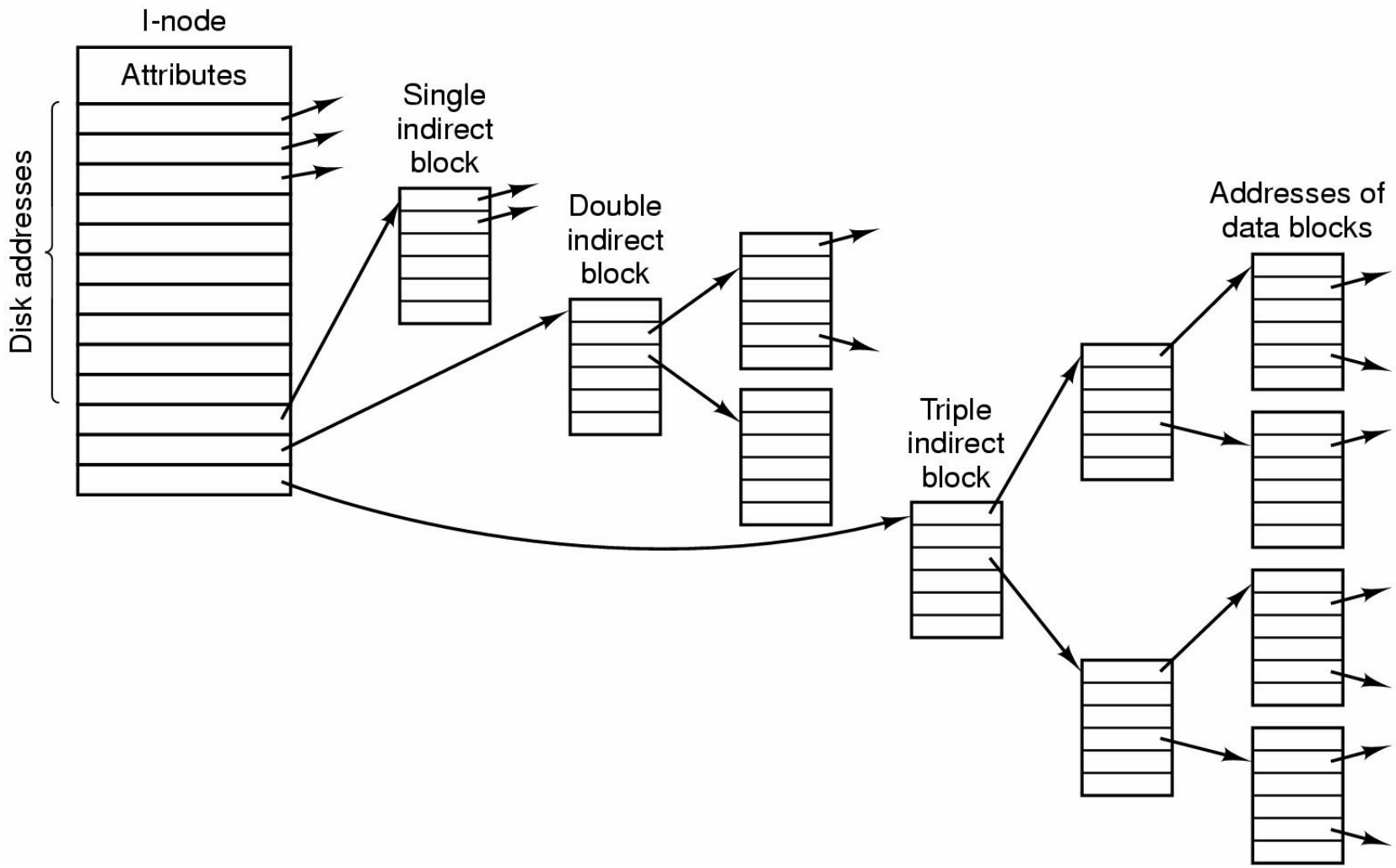
***file system* di tipo** Extfs : In questo caso, utilizzando tutti i campi dell'*i-node* principale, abbiamo a disposizione:

- 16 indici diretti di blocco, al costo di 1 blocco poiché un *i-node* occupa lo stesso spazio di un blocco;
- 1 indice di I indirezione, il quale punta ad un *i-node* interamente utilizzato per contenere indici diretti di blocco, che consente di esprimere fino a: $\lfloor \frac{1\ kB}{4\ B} \rfloor = \lfloor \frac{2^{10}}{2^2} \rfloor = 2^8 = 256$ indici di blocco, al costo di 1 ulteriore blocco;
- 1 indice di II indirezione, il quale punta ad un *i-node* speciale, interamente utilizzato per contenere puntatori ad *i-node* di I livello, che dunque consente di esprimere 256 puntatori a strutture ciascuna contenente fino a 256 indici diretti di blocco, per un totale di $256^2 = (2^8)^2 = 2^{16} = 65.536$ blocchi, al costo di $1 + 256 = 257$ ulteriori blocchi.

Conseguentemente, Extfs consente di rappresentare *file* di dimensione massima pari a: $(16 + 256 + 65.536) \times 1\ kB = 65.808\ kB = 64\ MB + 272\ kB$ (438,72 volte maggiore di quanto ottenuto con NTFS) per un rapporto inflattivo pari a: $\frac{(1+1+257) \times 1\ kB}{65.808\ kB} = 0,39\%$ (3,4 volte inferiore a quanto ottenuto con NTFS).

La considerazione ovvia da trarre da queste considerazioni è che NTFS) è particolarmente penalizzato dalle condizioni di scarsa o nulla contiguità. (Per contro, ad Extfs la contiguità non giova in alcun modo.)

Soluzione (descrizione *i-node*)



Esercizio "*Keeping Track of Free Blocks*" (pag. 471 del libro di testo)

Sia dato un disco di 16 GB diviso in blocchi ampi 1 KB. Si considerino due possibili strutture per tener traccia dei blocchi liberi: lista concatenata e *bitmap*. Nel primo caso, ogni elemento della lista è costituito a sua volta da un blocco, il quale contiene indici di blocco (di 32 *bit* ciascuno), dei quali l'ultimo è riservato per l'indicazione del prossimo blocco di lista libera. Nel secondo caso l'uso di un *bit* 1 o 0 definisce se il corrispondente blocco sia libero o utilizzato.
Si calcoli l'occupazione di memoria delle due strutture.

Soluzione

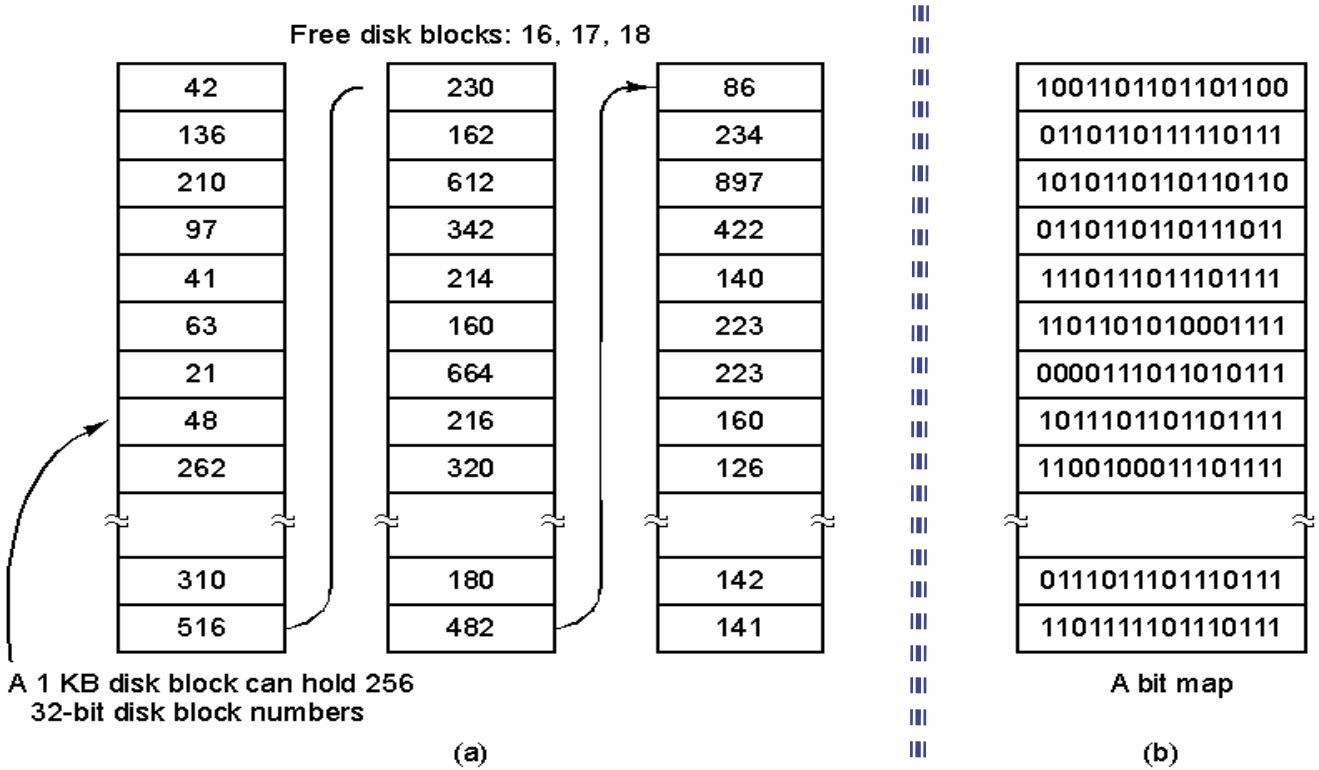
16 GB = 2^{34} B diviso in blocchi da 1 KB = 2^{10} B/blocco
ovvero 2^{34} B / 2^{10} B/blocco = 2^{24} blocchi = 16 M blocchi.

Ogni blocco può contenere 1 KB / 4 B/indice = 256 indici di blocco di cui 1 viene usato come collegamento al "blocco di indici" successivo nella lista. Ne rimangono dunque 255 utilizzabili per rappresentare i blocchi liberi.

Per rappresentare una lista di **massima ampiezza** servono dunque:
16 M indici / 255 indici/blocco = 65793,0039.. ≈ **65794** blocchi
cioè poco più di 64 K × 1 KB = **64 MB**

Con la struttura a *bitmap* sono invece **sempre** necessari 2^{24} *bit* = 2^{21} B =
= 2^{11} KB = **2 MB**

Soluzione



Riformulare la soluzione:

- variando la dimensione del blocco
- senza conoscere a priori la dimensione di ogni indice (32 *bit*)

Risposta a Quesiti degli Studenti

- SWAPPING: allocazione zone di memoria con politica *Next Fit*

- *Scheduling* Processi: gestione coda di "*ready*/pronti" per politica di ordinamento *Round Robin*