Gestione della Memoria

Introduzione ai Sistemi Operativi

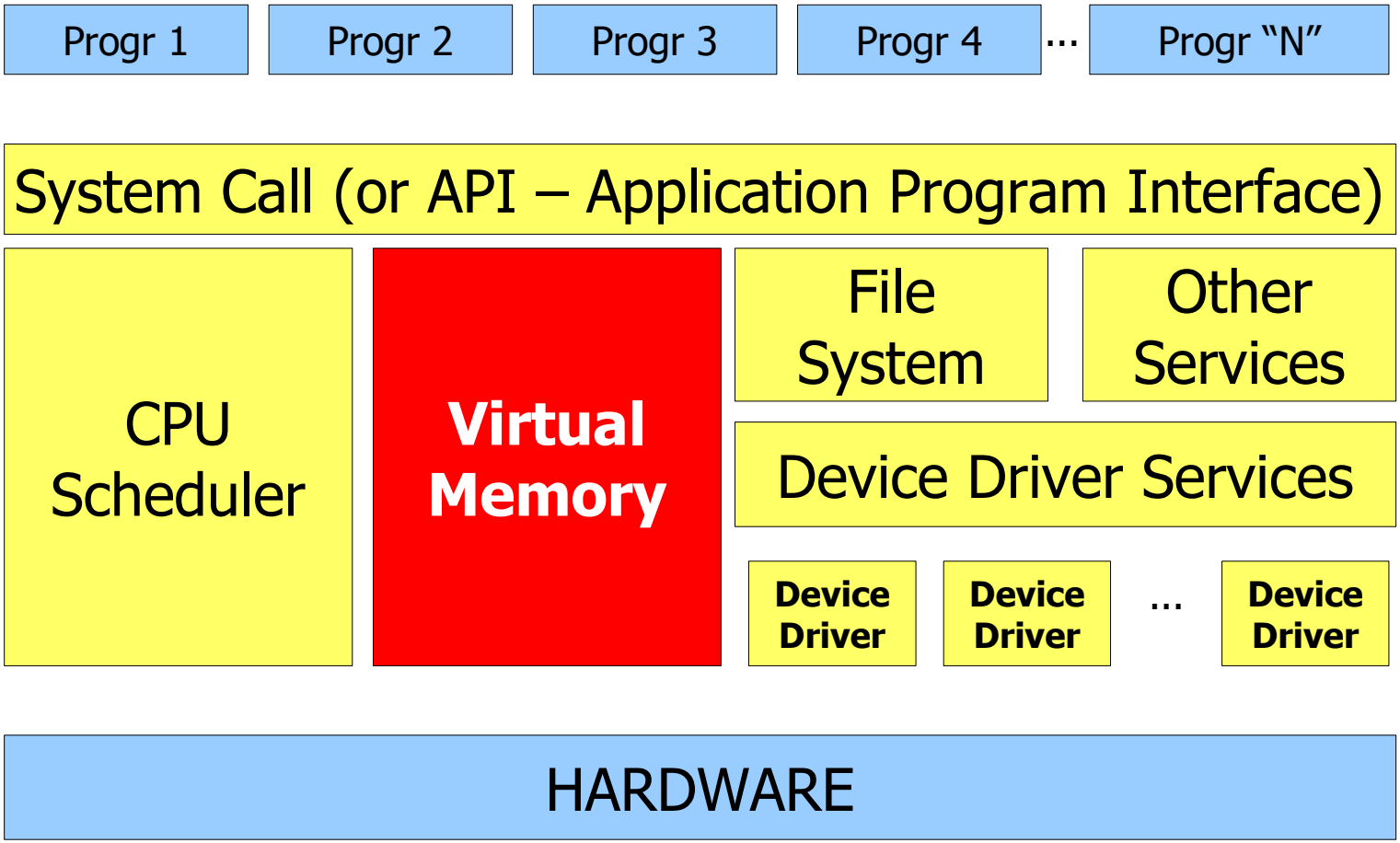
Corso di Informatica

Laurea in Fisica

prof. Ing. Corrado Santoro

A.A. 2008-09

Architettura di un sistema operativo

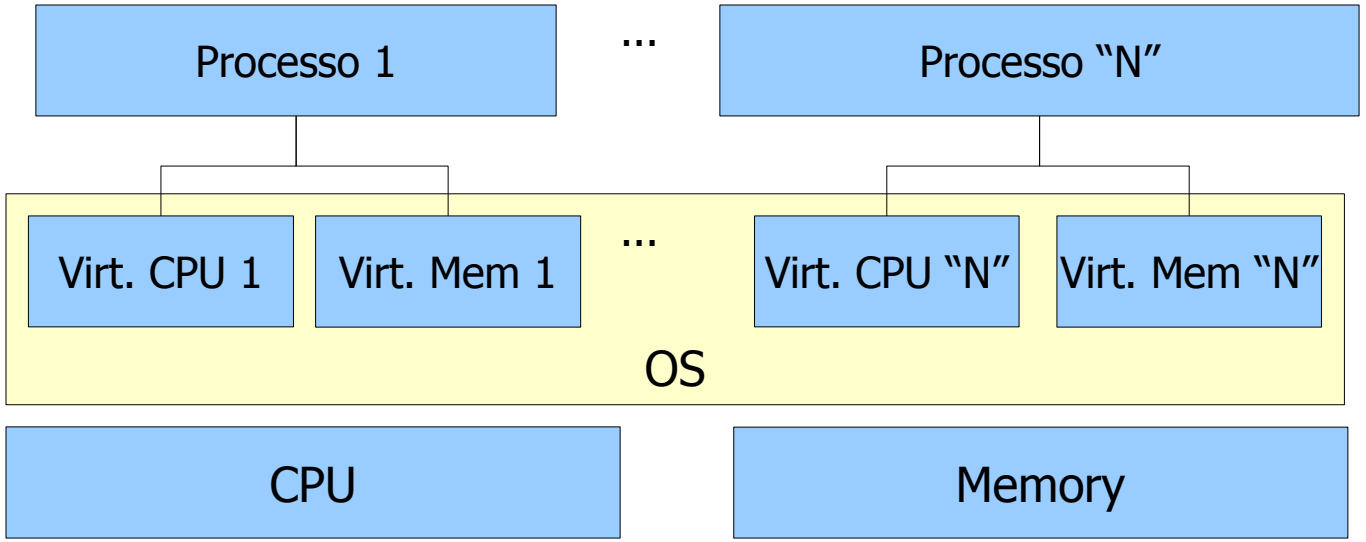


Gestione della memoria

- Ogni programma in esecuzione usa una certa quantità di memoria
- Ogni programma in esecuzione **deve** possedere **la sua** memoria
- La totalità della memoria fisica deve essere opportunamente **"spezzettata"** tra i vari programmi in esecuzione
- Cosa accade se un programma richiede una memoria **maggiore** di quella a lui assegnata?

Gestione della memoria

- Compito del sistema operativo è "virtualizzare" la memoria fisica allo scopo di offrire, ad ogni programma in esecuzione un proprio spazio di indirizzamento **contiguo** e **privato**

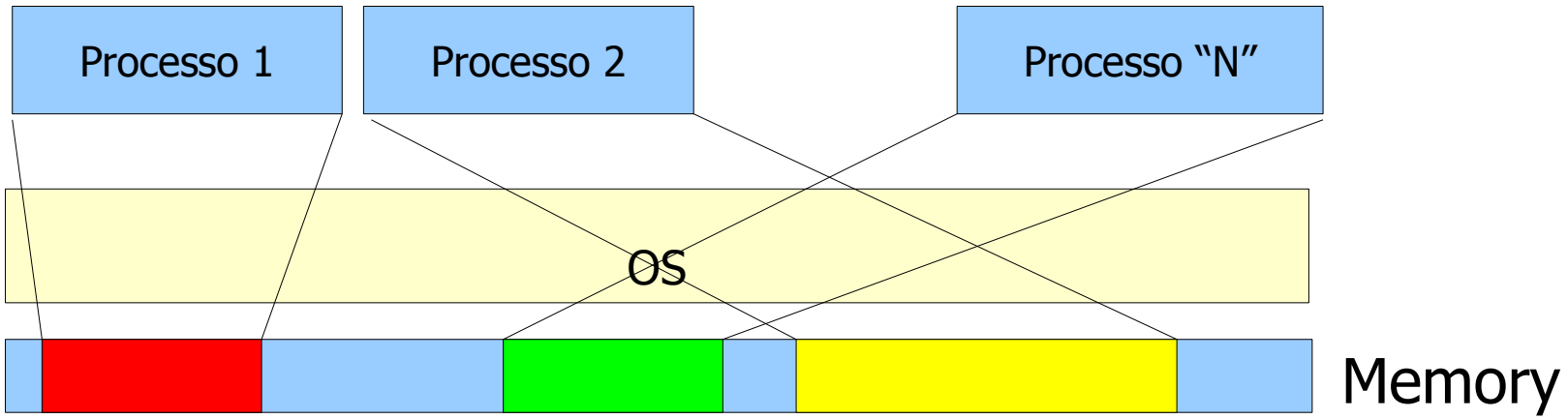


Tecniche di gestione della memoria

- **Allocazione contigua**
  - Molto semplice
  - Poco efficiente
  - Costosa dal punto di vista computazionale
- **Paginazione**
  - Più complessa
  - Molto efficiente
  - Poco costosa computazionalmente
  - Supportata da hardware specifico

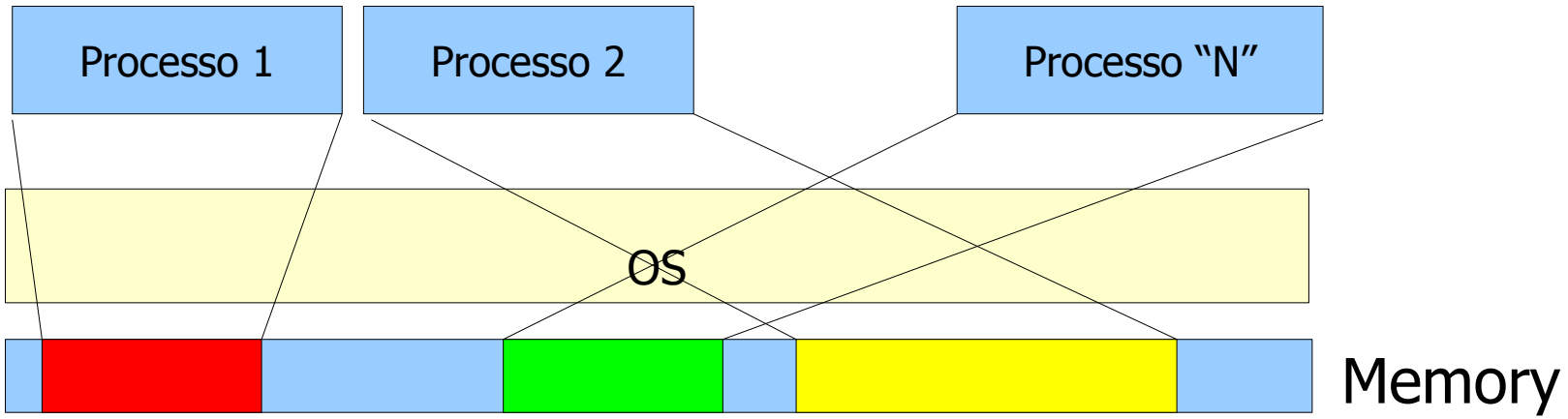
Allocazione contigua

- In funzione delle esigenze dei processi, il sistema operativo suddivide la memoria centrale e ne assegna una partizione ad ogni processo
- Il sistema operativo tiene traccia:
  - delle zone di memoria assegnate
  - dei processi a cui ha assegnato ogni zona
  - delle zone di memoria libere

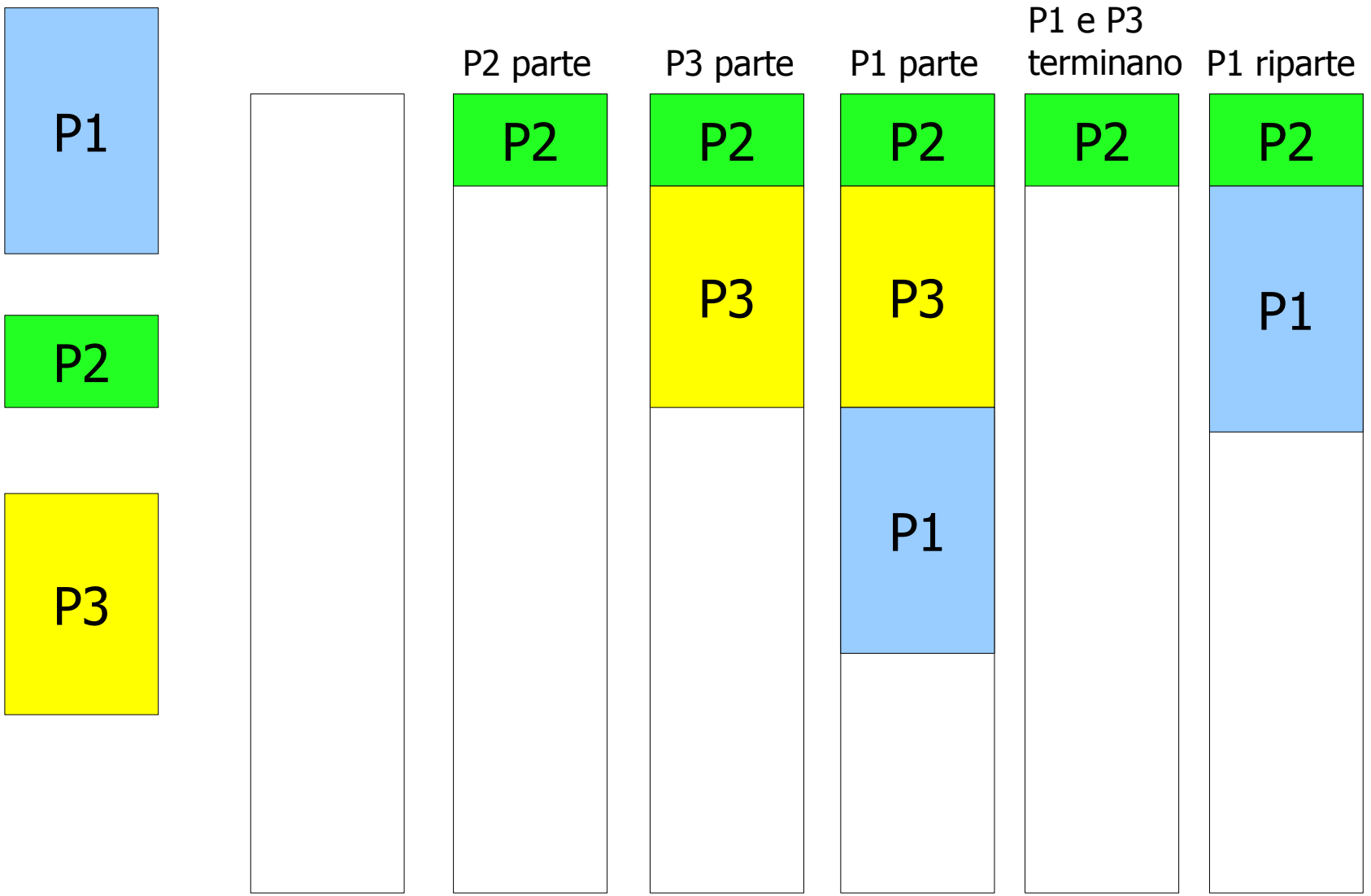


Allocazione contigua

- La memoria è quindi "bucherellata" in
  - zone contigue assegnate ai processi
  - zone libere (buchi)
- La zona di memoria **fisica** assegnata ad un processo è stabilita a **runtime**
- **Quindi per ogni esecuzione dello stesso programma, la zona di memoria assegnata può cambiare**



Allocazione contigua



Memoria e Indirizzi

- Ogni volta che uno stesso programma viene avviato, occuperà/utilizzerà zone di memoria **diverse**
- Tuttavia un programma usa dei riferimenti alla memoria: **indirizzi numerici**
- Allora, se la memoria assegnata **cambia** ad ogni esecuzione ...
- ... gli indirizzi da usare **devono** cambiare ad ogni esecuzione
- Tuttavia gli indirizzi che il programma usa **sono scritti nel codice macchina generato dal compilatore**

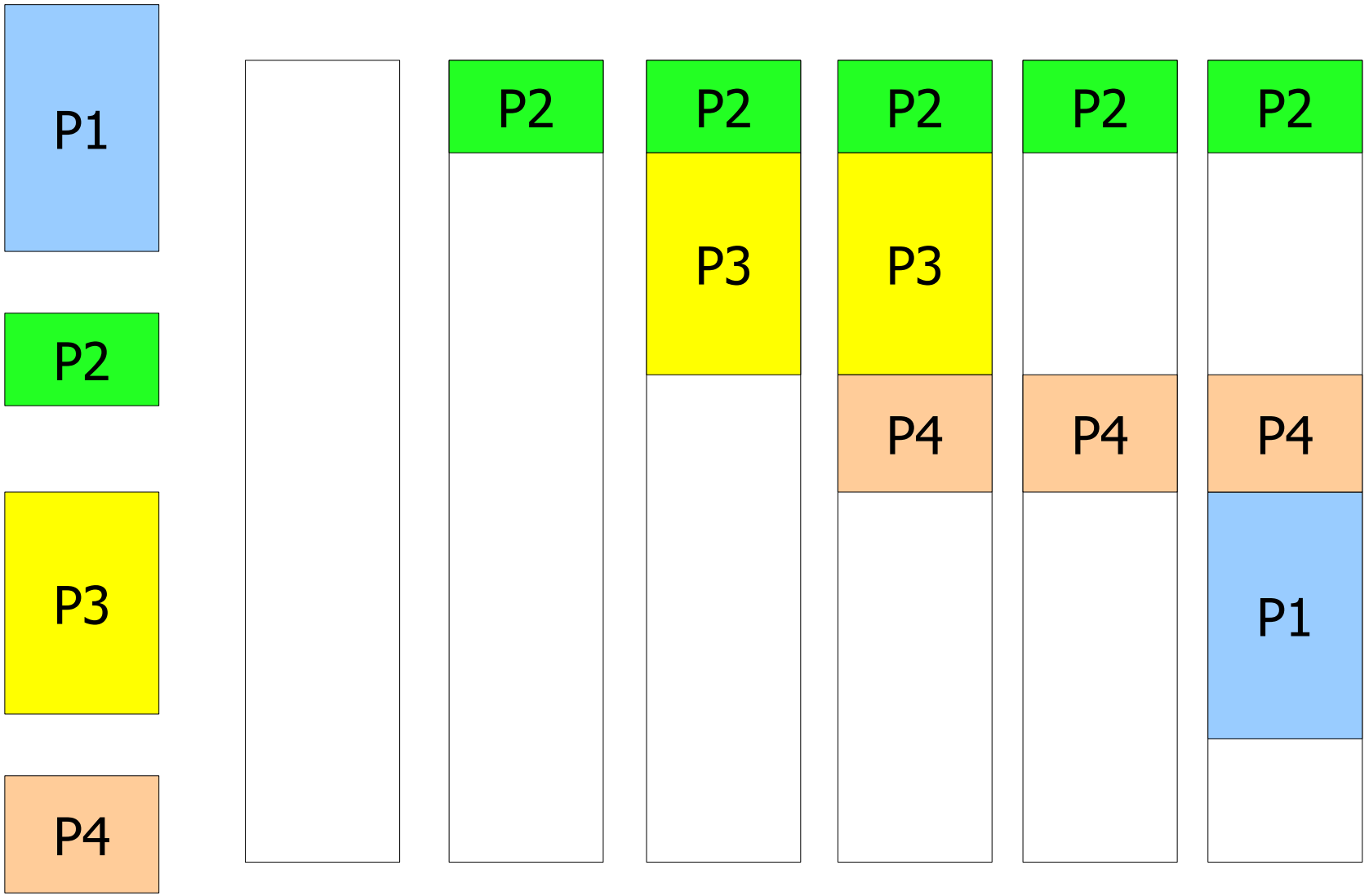
Loading e Relocation

- Il compilatore genera un eseguibile in cui i riferimenti agli indirizzi **non sono assoluti** ma **relativi**
- Il sistema operativo, quando carica il programma, gli assegna una zona di memoria che parte da un certo indirizzo: **base_address**
- Il sistema operativo modifica il codice del programma caricato aggiungendo il base_address ad ogni riferimento in memoria: **(LOAD TIME-)RELOCATION**, **RILOCAZIONE**
- Gli indirizzi vengono quindi trasformati da **relativi** ad **assoluti**

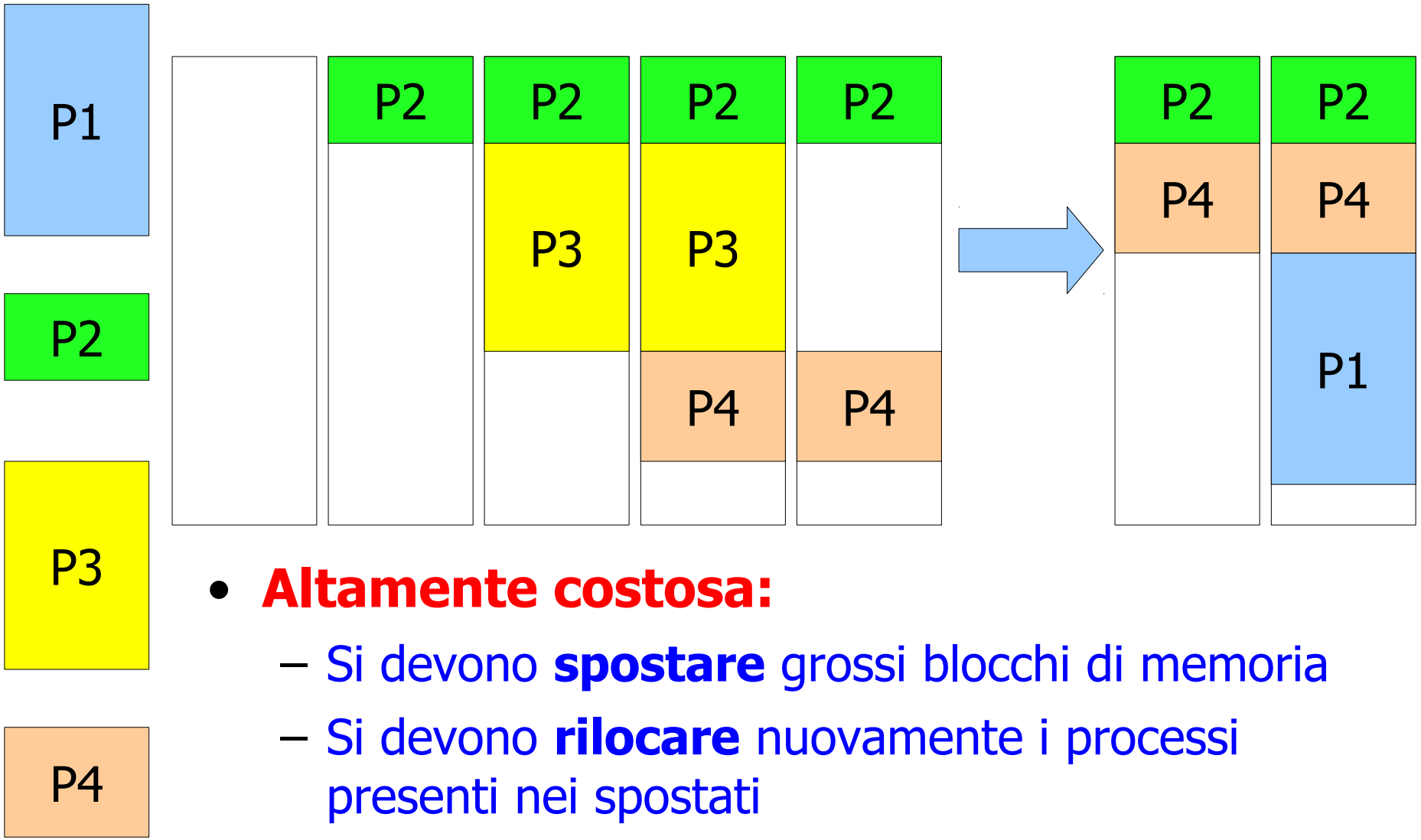
Frammentazione

- Man mano che i processi terminano, la memoria da loro usata viene liberata
- Tuttavia rimane un "buco" (di dimensioni N) che può essere occupato **solo** da un processo di dimensioni <= N
- Con l'andare del tempo, si creano molti buchi piccoli
- Si arriva al paradosso:
  - Processo di dimensioni K
  - Tanti buchi di dimensione $N_i < K$
  - Il processo non può essere caricato nonostante:
    - $\mathbf{\Sigma N_i > K}$

Frammentazione

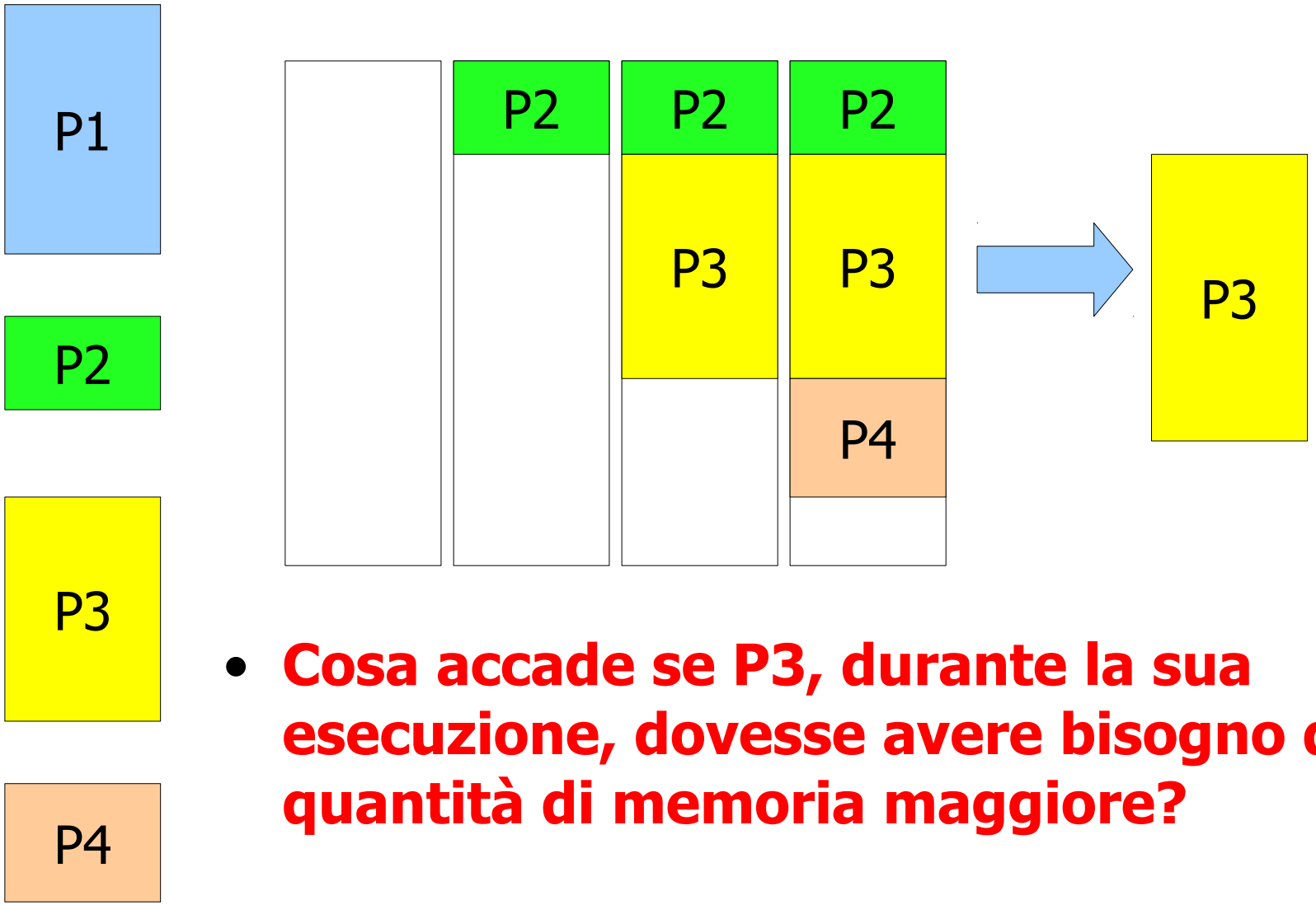


Frammentazione: Compattazione



- **Altamente costosa:**
  - Si devono **spostare** grossi blocchi di memoria
  - Si devono **rilocare** nuovamente i processi presenti nei spostati

Crescita della memoria occupata



- **Cosa accade se P3, durante la sua esecuzione, dovesse avere bisogno di una quantità di memoria maggiore?**

Problemi da risolvere

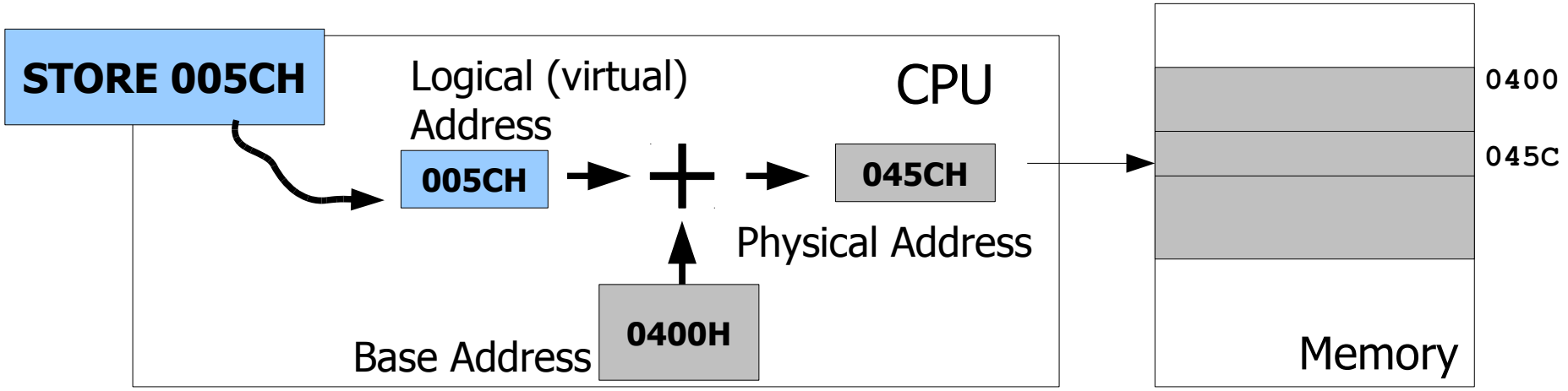
- Rilocazione a load-time (caricamento del programma)
- Rilocazione a run-time (per compattazione blocchi)
- Alto costo della compattazione
- Frammentazione

Gli indirizzi virtuali

- Abbiamo imparato che i programmi usano indirizzi (riferimenti) assoluti (diretti) alla memoria principale
  - STORE 035CH: memorizza un valore nell'indirizzo di memoria 035C esadecimale
- Tuttavia sono proprio gli indirizzi assoluti che provocano i problemi di rilocazione
- Allora... inventiamoci un nuovo modello di indirizzamento che non necessita la rilocazione
- **SOLUZIONE**: indirizzi **indipendenti** dall'effettiva zona di memoria assegnata al processo

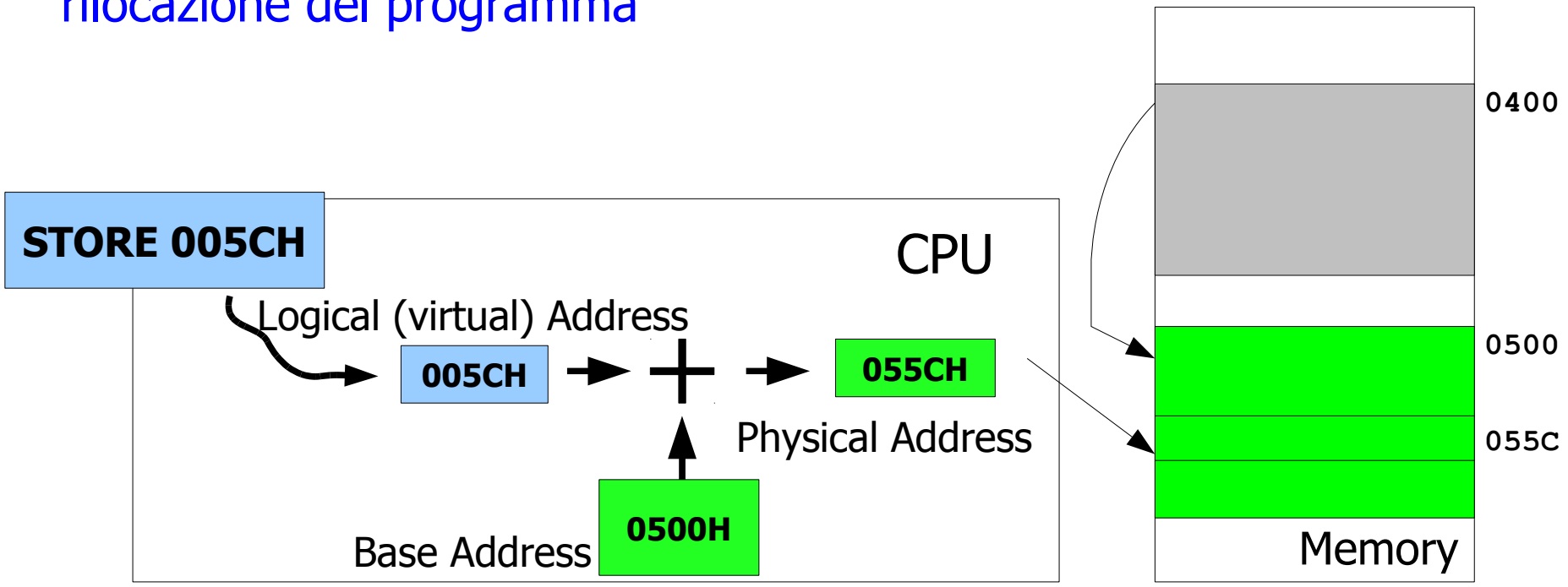
Gli indirizzi virtuali

- Il processo usa **indirizzi virtuali**, indipendenti dalla memoria fisica assegnata, che non cambiano mai
  - Es: STORE 005CH
- Al momento del caricamento del programma, viene assegnato il blocco di memoria e impostato il **base_address in un registro speciale della CPU** (es. blocco che inizia all'indirizzo 0400H)
- Quando il programma fa riferimento ad un indirizzo, la CPU automaticamente **somma il base_address** e ottiene **l'indirizzo fisico**



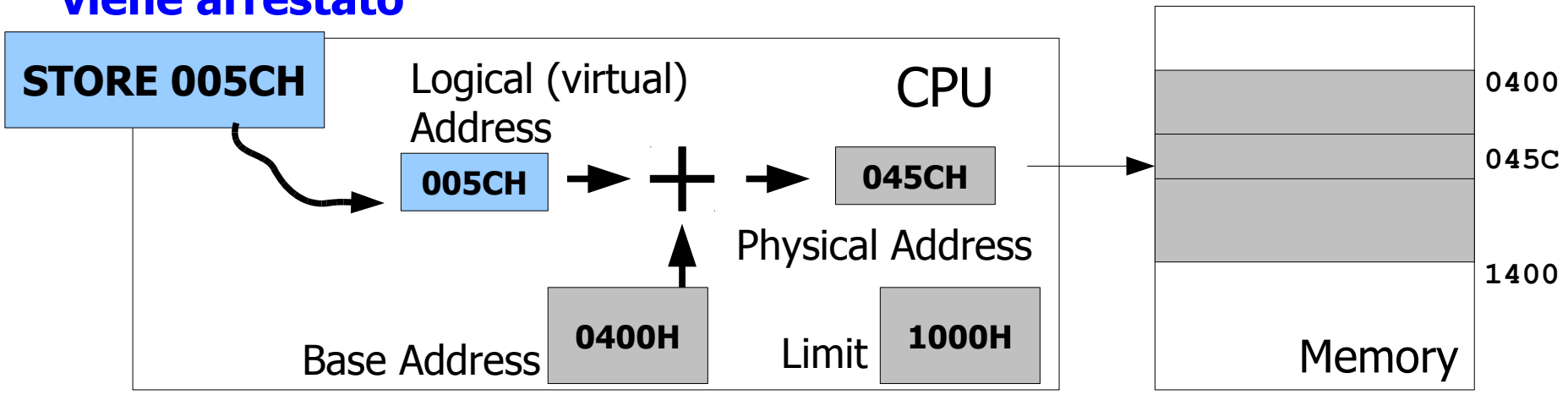
Gli indirizzi virtuali

- Ogni volta che il programma viene caricato, non è necessario rilocarlo ma è sufficiente impostare opportunamente il **base_address**
- Analogamente, se occorre compattare la memoria, è sufficiente aggiornare il **base_address** e non è necessario eseguire la rilocazione del programma



Gli indirizzi virtuali

- Con questo meccanismo è possibile garantire anche la **privatezza**
- Cioè che ogni processo **non possa** accedere allo spazio di memoria di un altro processo
- I processi NON USANO indirizzi fisici, quindi sono vincolati solo al proprio spazio di indirizzamento
- Un ulteriore registro speciale, denominato **limit**, memorizza la dimensione del blocco di memoria assegnato al processo
- Se l'indirizzo fisico generato è **> base_address + limit**, il programma **viene arrestato**



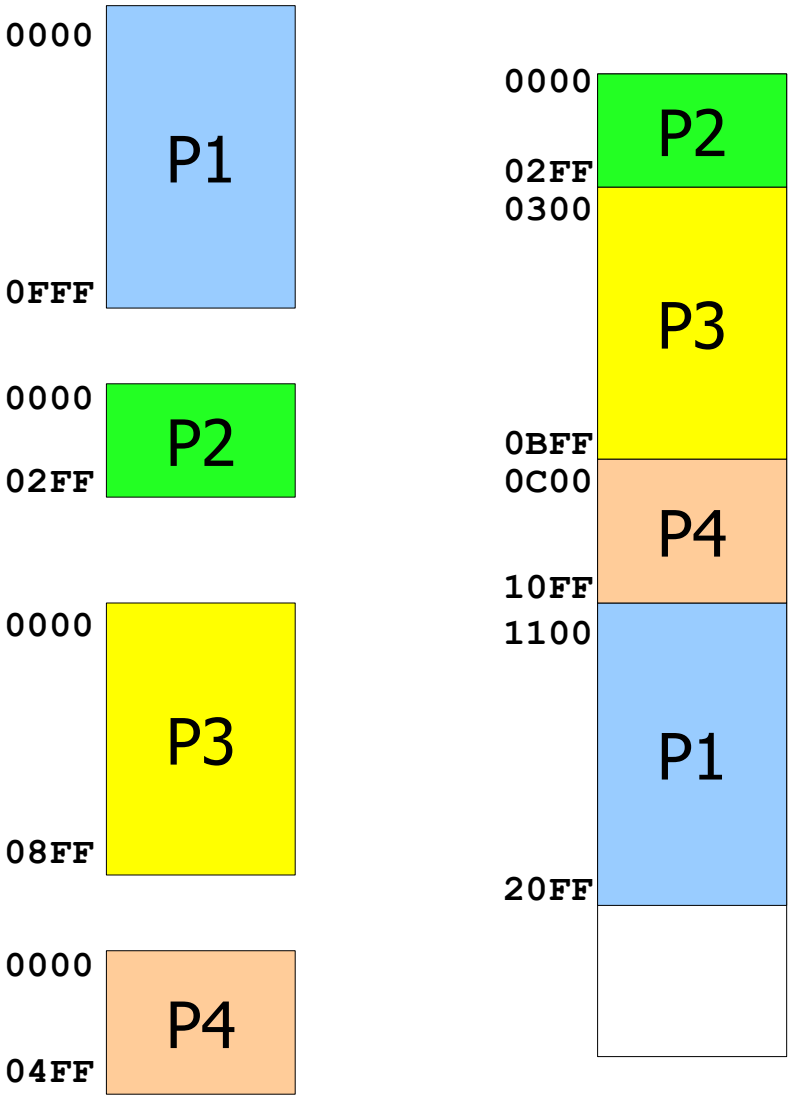
Indirizzi virtuali e processi

- Il base address e il limit sono informazioni **dipendenti dal processo**
- Ogni processo ha i suoi **base address** e **limit**
- Lo scheduler, quando effettua il context switch, deve anche aggiornare i registri speciali della CPU con i nuovi valori di **base address** e **limit**

Problemi da risolvere

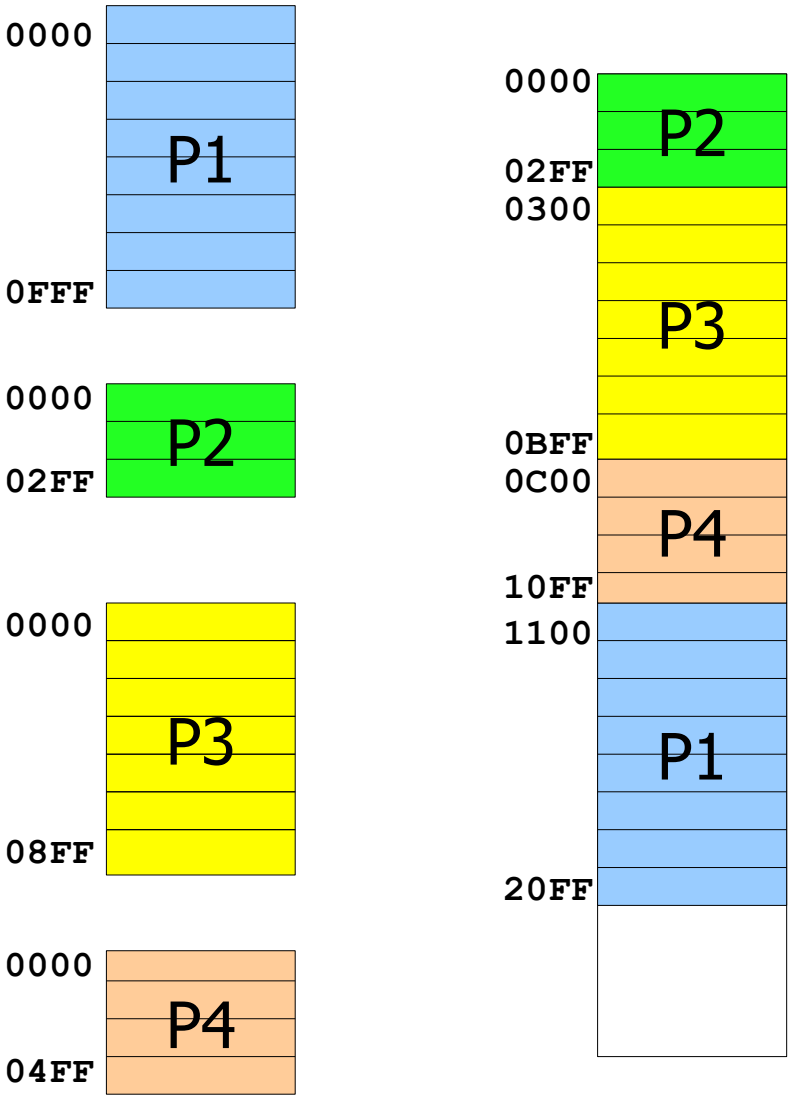
- ✔ Rilocazione a load-time (caricamento del programma)
- ✔ Rilocazione a run-time (per compattazione blocchi)
- ✔ Privatezza dei blocchi di memoria
- Alto costo della compattazione
- Frammentazione

Indirizzi virtuali vs. fisici



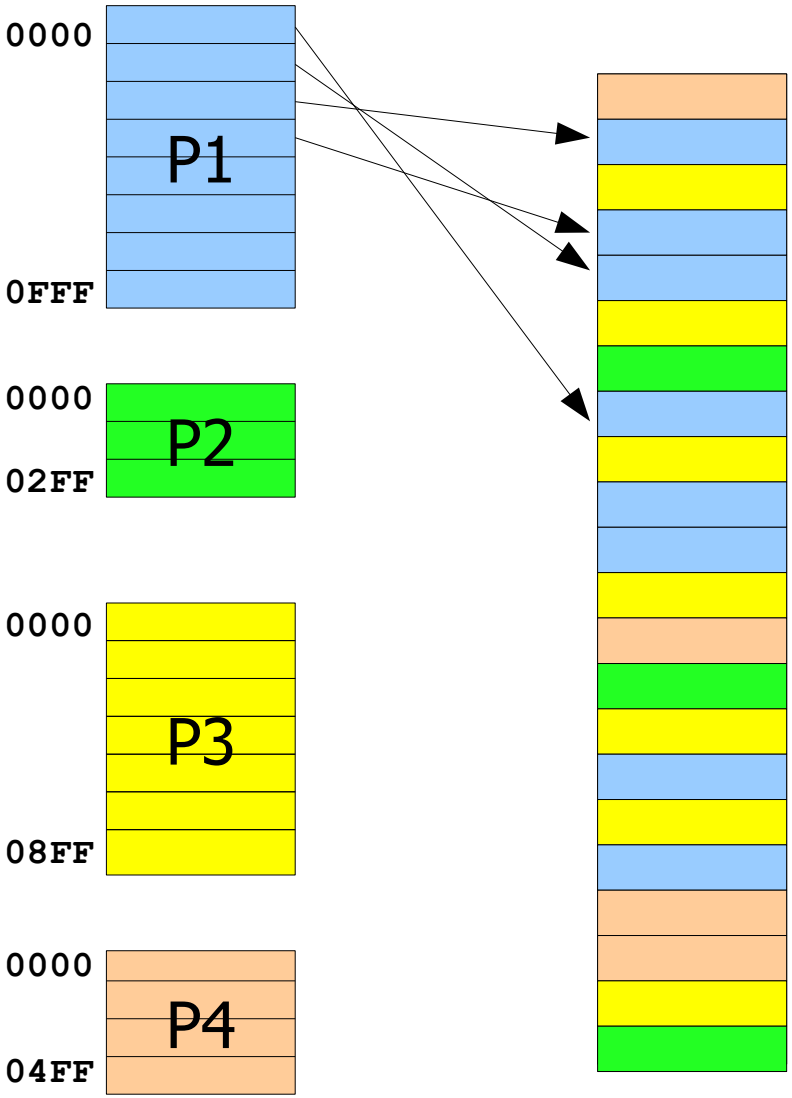
- Vi è un mapping 1:1 tra l'**intero spazio logico**/virtuale del processo e l'**intero blocco di memoria fisica** assegnato

Verso il paging (paginazione)



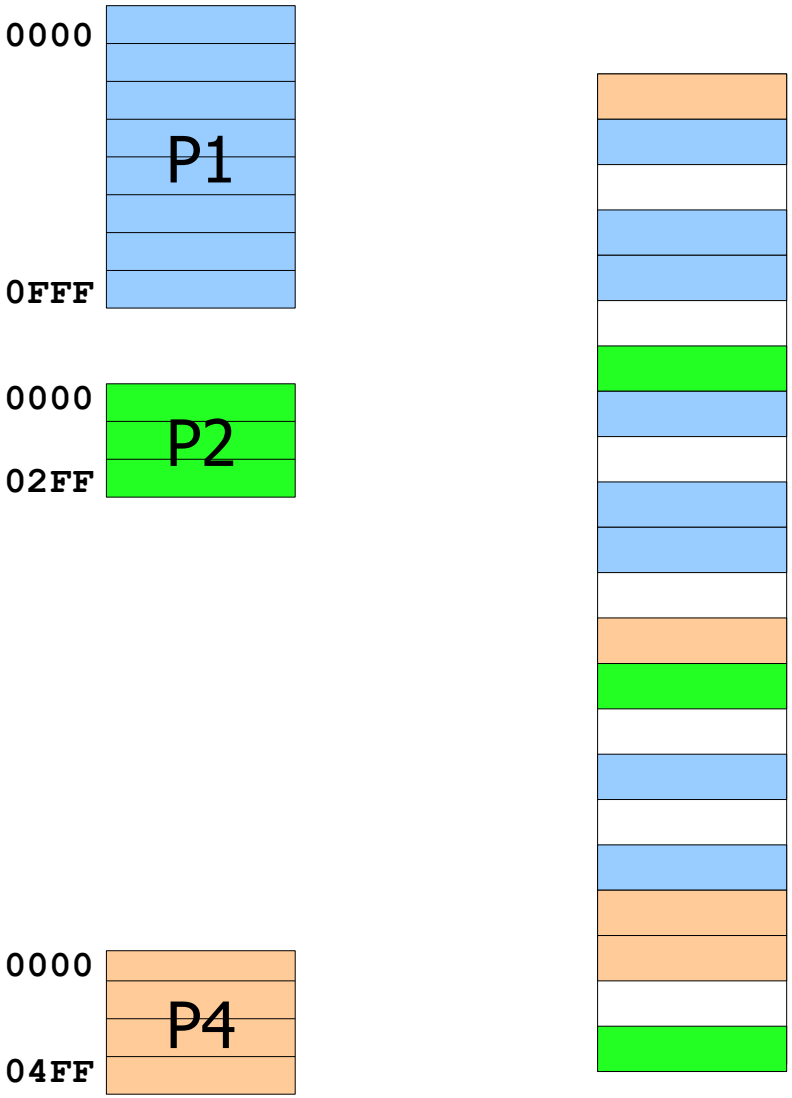
- Ora supponiamo di **suddividere** lo spazio logico e fisico in blocchetti di **dimensione fissa**
- ... mantenendo lo stesso tipo di mapping tra blocco logico e blocco fisico

Verso il paging (paginazione)



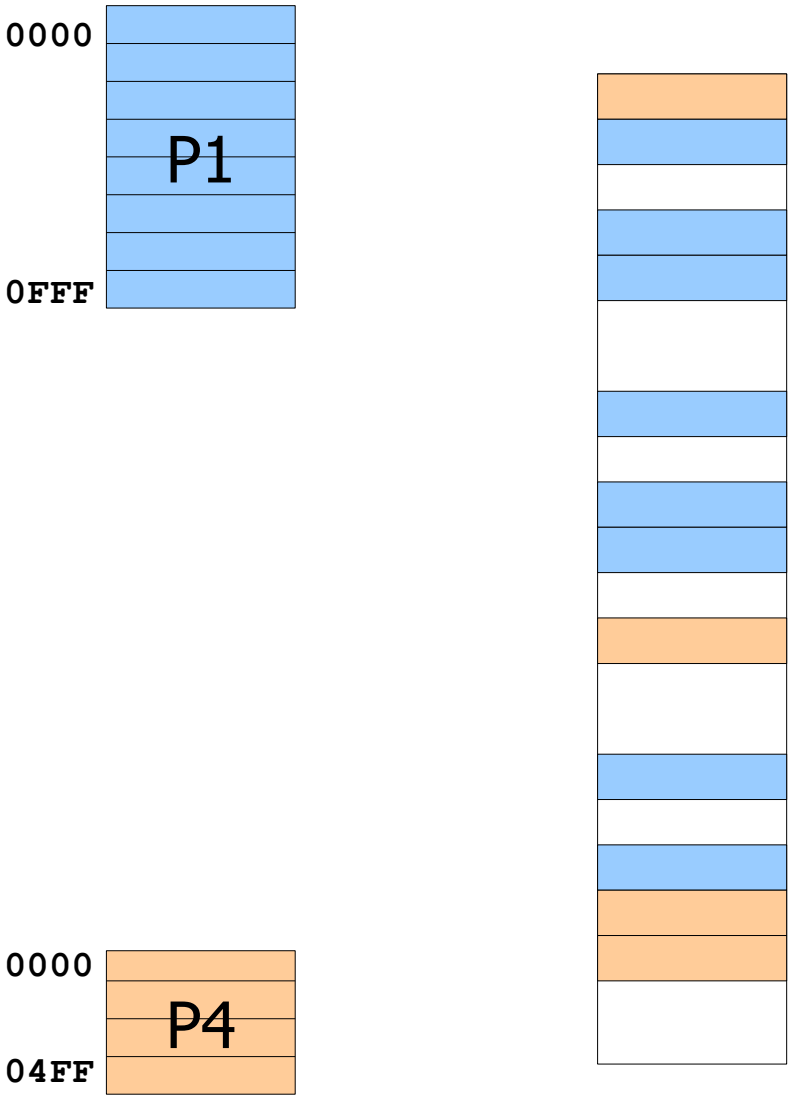
- Ora supponiamo di poter effettuare il mapping tra i **singoli blocchetti** e non più tra i blocchi di memoria interi
- I blocchetti di memoria fisica **non devono più essere contigui**
- **Si eliminano i problemi di rilocazione a run-time e frammentazione**

Va via P3 ...



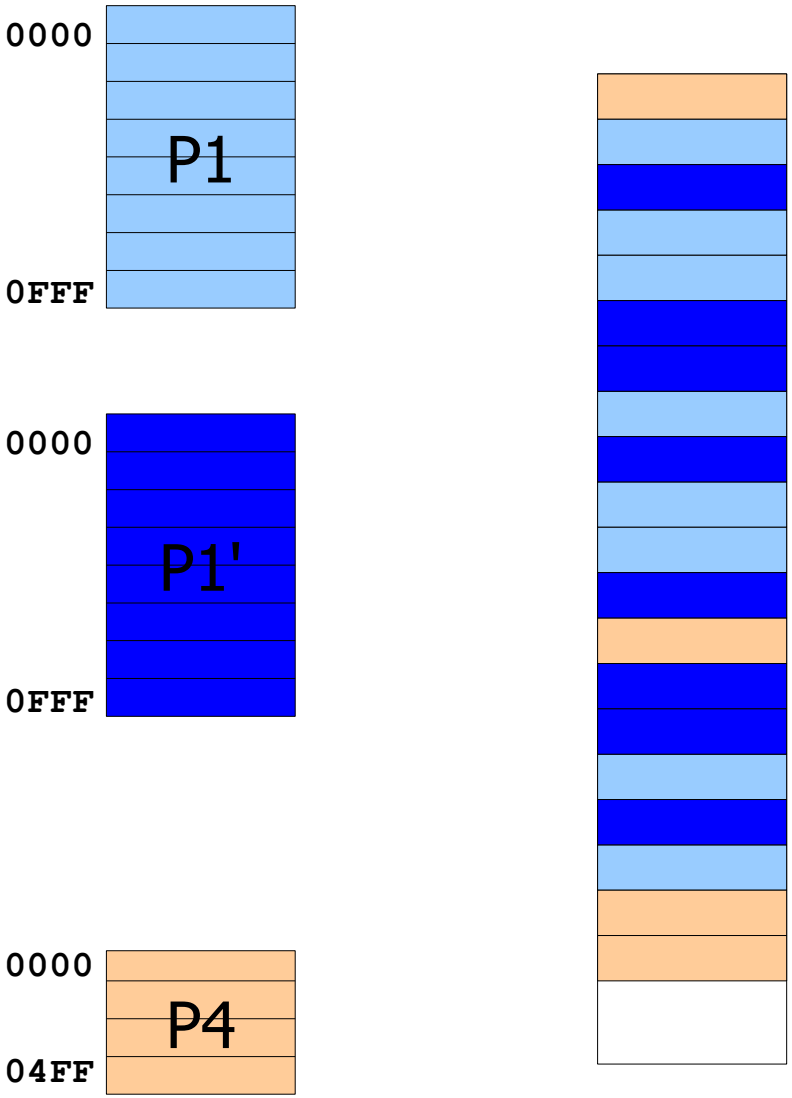
- Liberiamo 7 "blocchettini"

Va via P2 ...



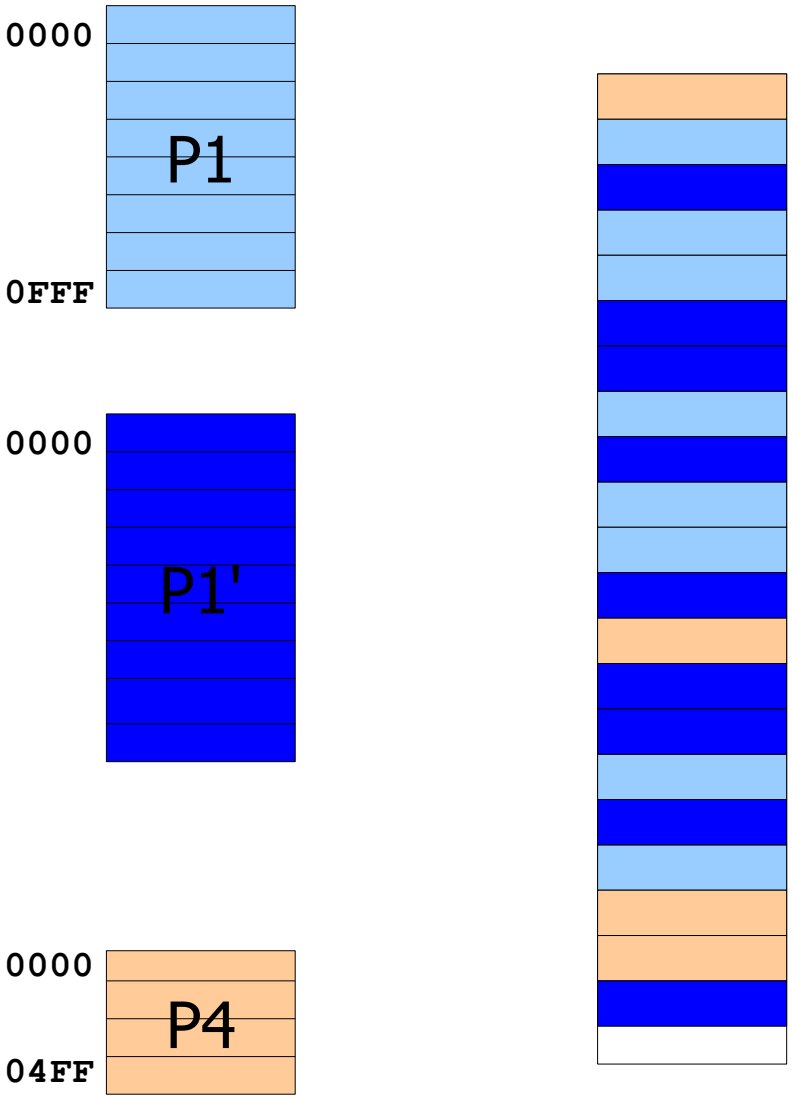
- Liberiamo altri 3 “blocchettini”

Arriva un'altra istanza di P1 ...



- Ci servono 8 "blocchettini", **ma non è necessario che siano contigui**
- **Non si sono rese necessarie né la rilocazione né la compattazione**

P1' cresce ...



- ... e vorrebbe un altro po' di memoria
- ... gli assegnamo un altro blocchettino che prima era libero
- **Non si sono rese necessarie né la rilocazione né la compattazione**

(Quasi) tutto risolto!

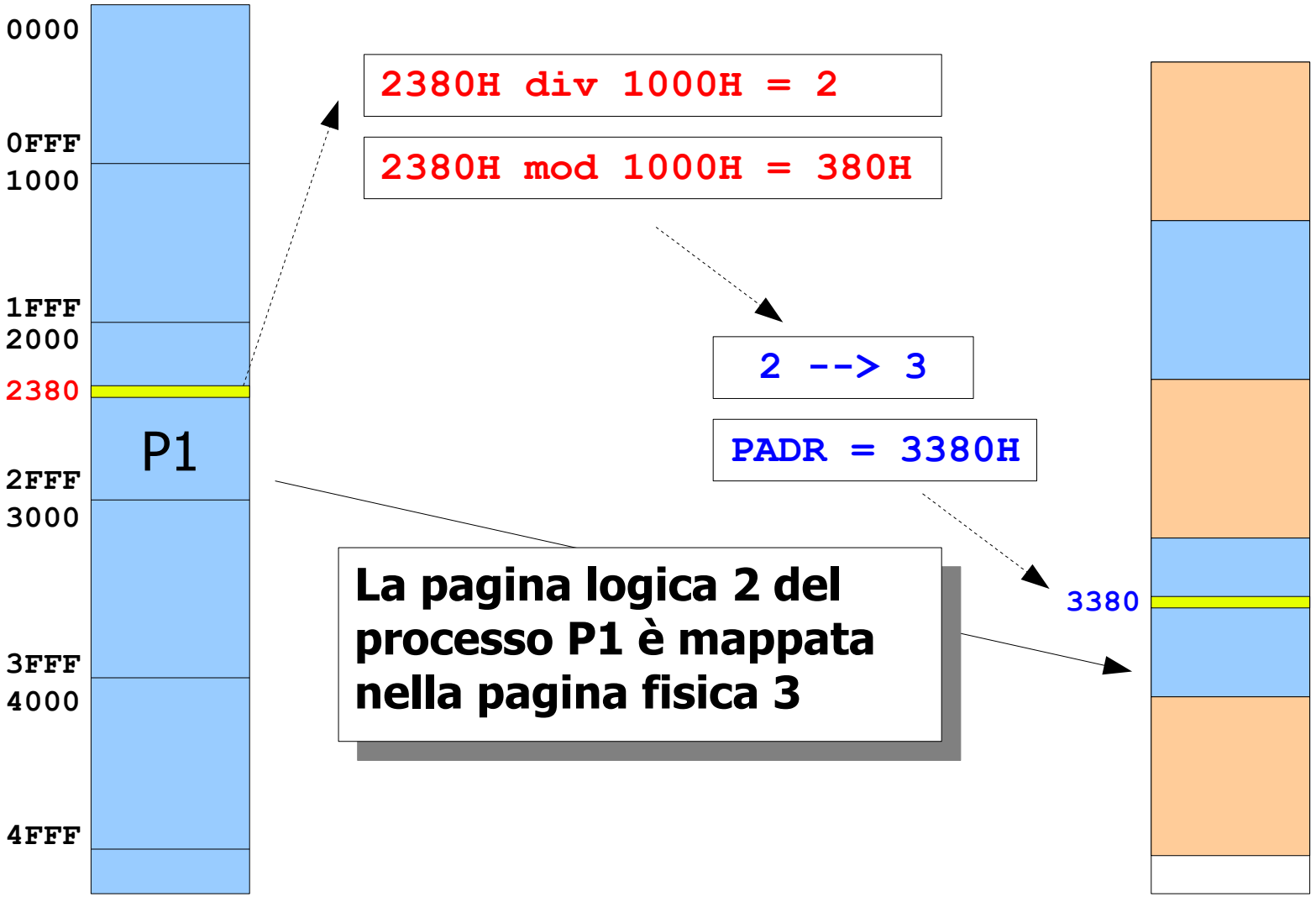
- ✔ Rilocazione a load-time (caricamento del programma)
- ✔ Rilocazione a run-time (per compattazione blocchi)
- ✔ Privatezza dei blocchi di memoria
- ✔ Alto costo della compattazione
- ✔ Frammentazione

- Cosa sono questi "blocchetti"?
- Serve un meccanismo di mapping degli indirizzi (da logico a fisico) che tenga conto dei "blocchetti"

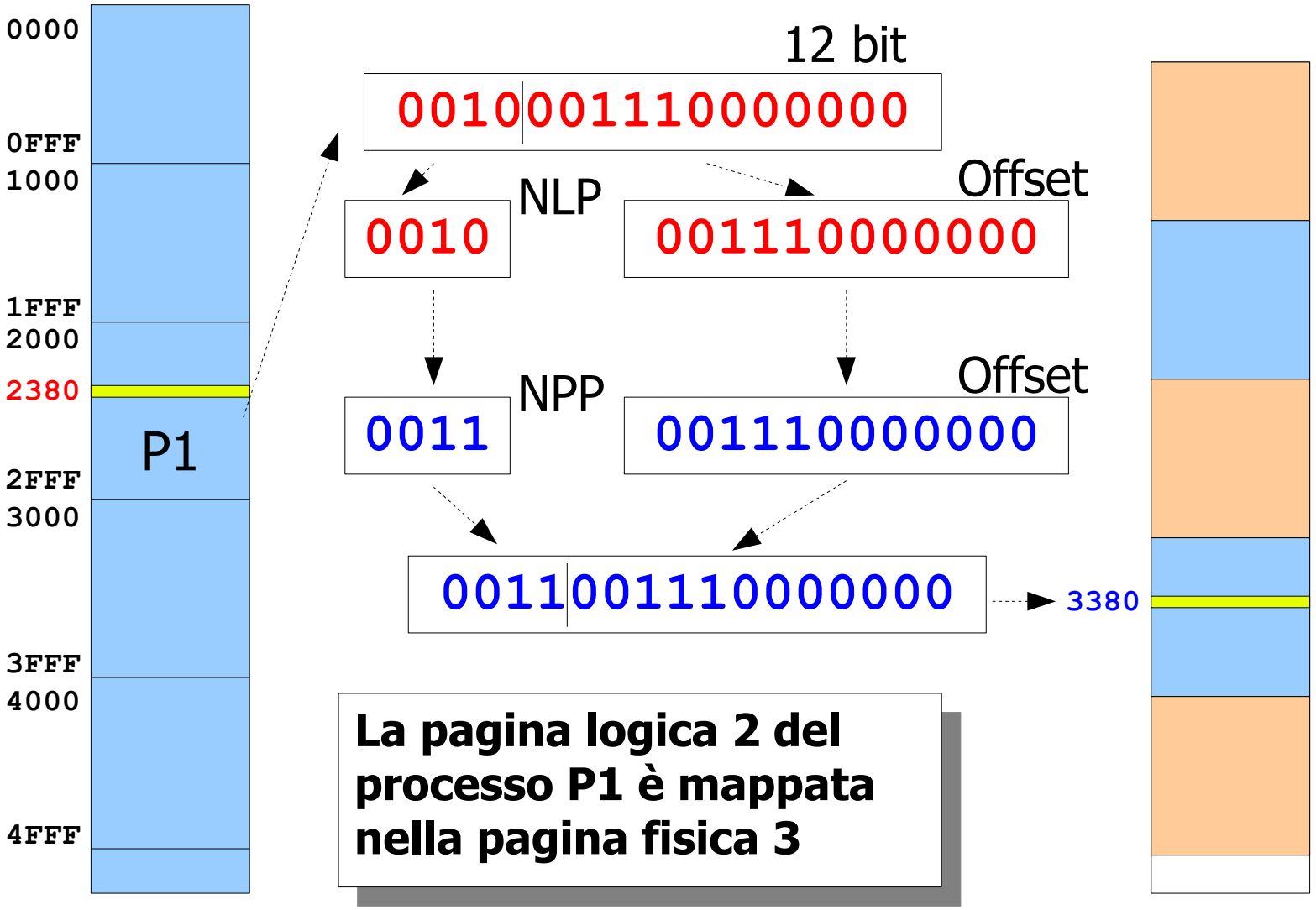
Il paging (paginazione)

- Ogni "blocchetto" è denominato **pagina**
- La pagina ha una dimensione prefissata (una potenza del 2)
  - In genere è 4 Kbytes = 4096 bytes = 2^{12}
- L'indirizzo logico/virtuale **VADR** viene suddiviso in due parti:
  - Numero pagina logica, **NLP = int (VADR / 4096)**
  - Offset (displacement), **D = VADR mod 4096**
- La CPU (in realtà un componente specifico della CPU) traduce il **NLP** in **numero di pagina fisica**, **NPP**, in base alla corrispondenza tra "blocchetto logico" e "blocchetto fisico"
- La CPU genera l'indirizzo fisico PADR combinando NPP e D
  - **PADR = NPP * 4096 + D**

Paging

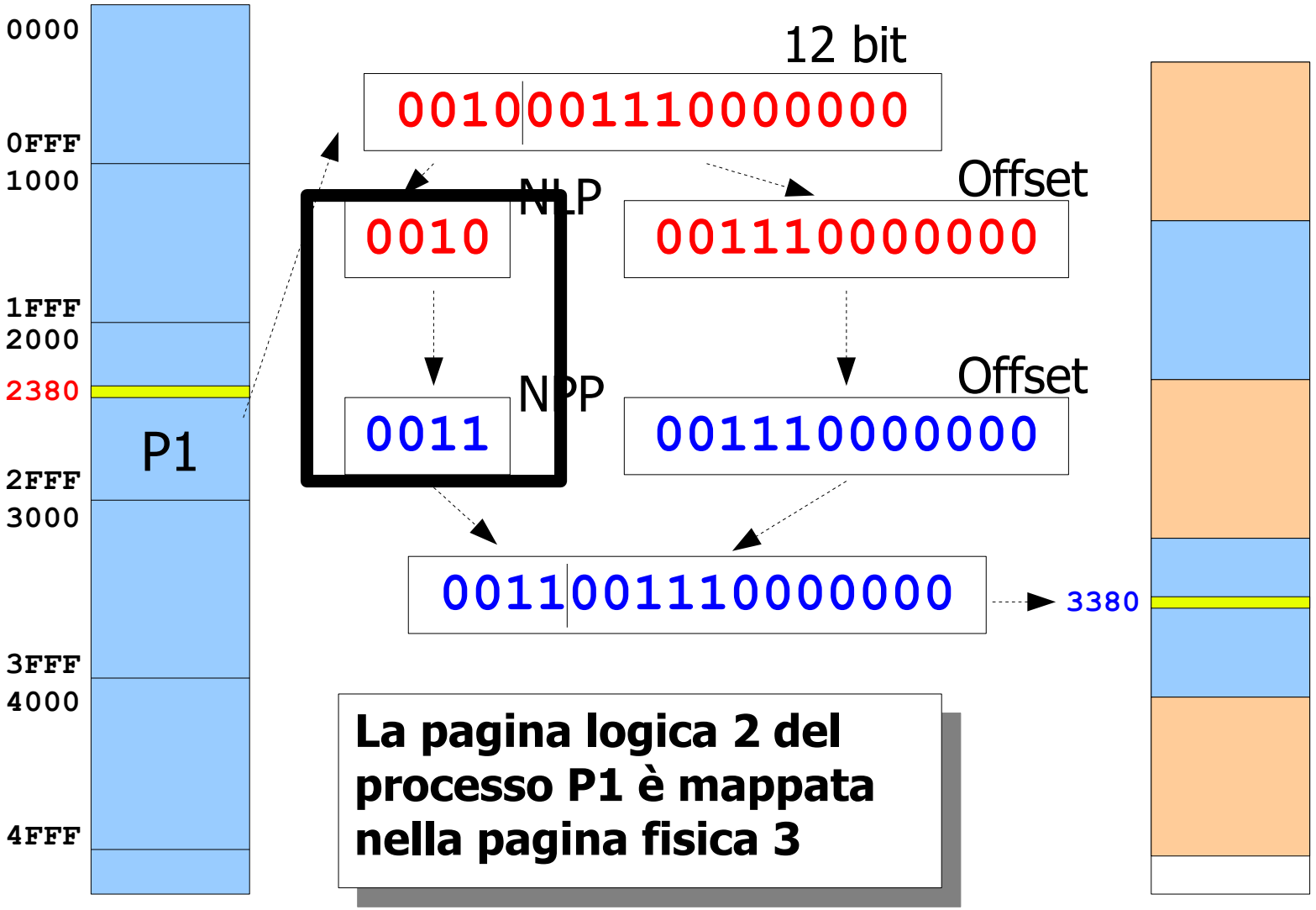

0000
0FFF
1000
1FFF
2000
2380
P1
2FFF
3000
3FFF
4000
4FFF
2380H div 1000H = 2
2380H mod 1000H = 380H
2 --> 3
PADR = 3380H
3380
La pagina logica 2 del processo P1 è mappata nella pagina fisica 3

Paging



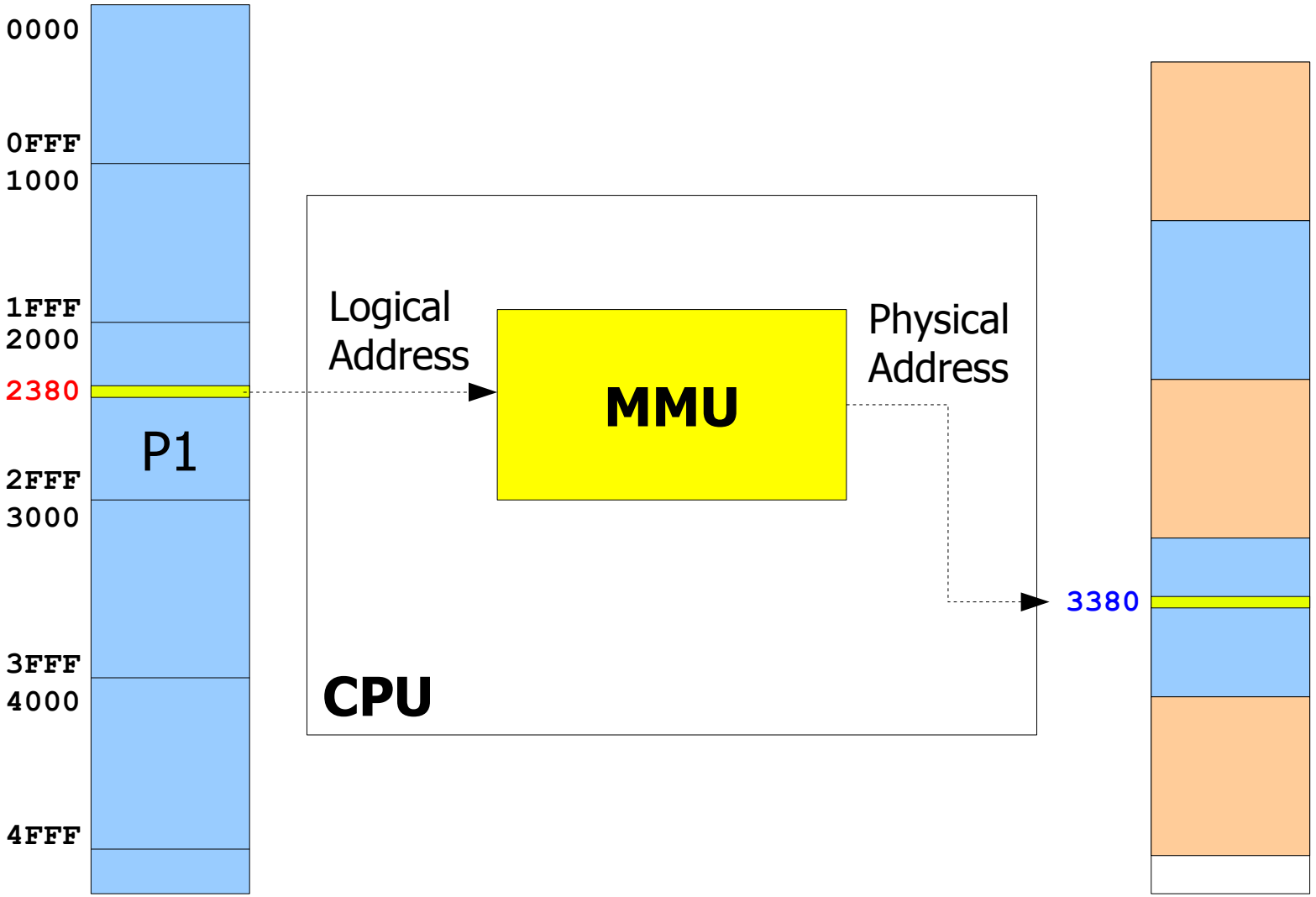
La pagina logica 2 del processo P1 è mappata nella pagina fisica 3

Paging

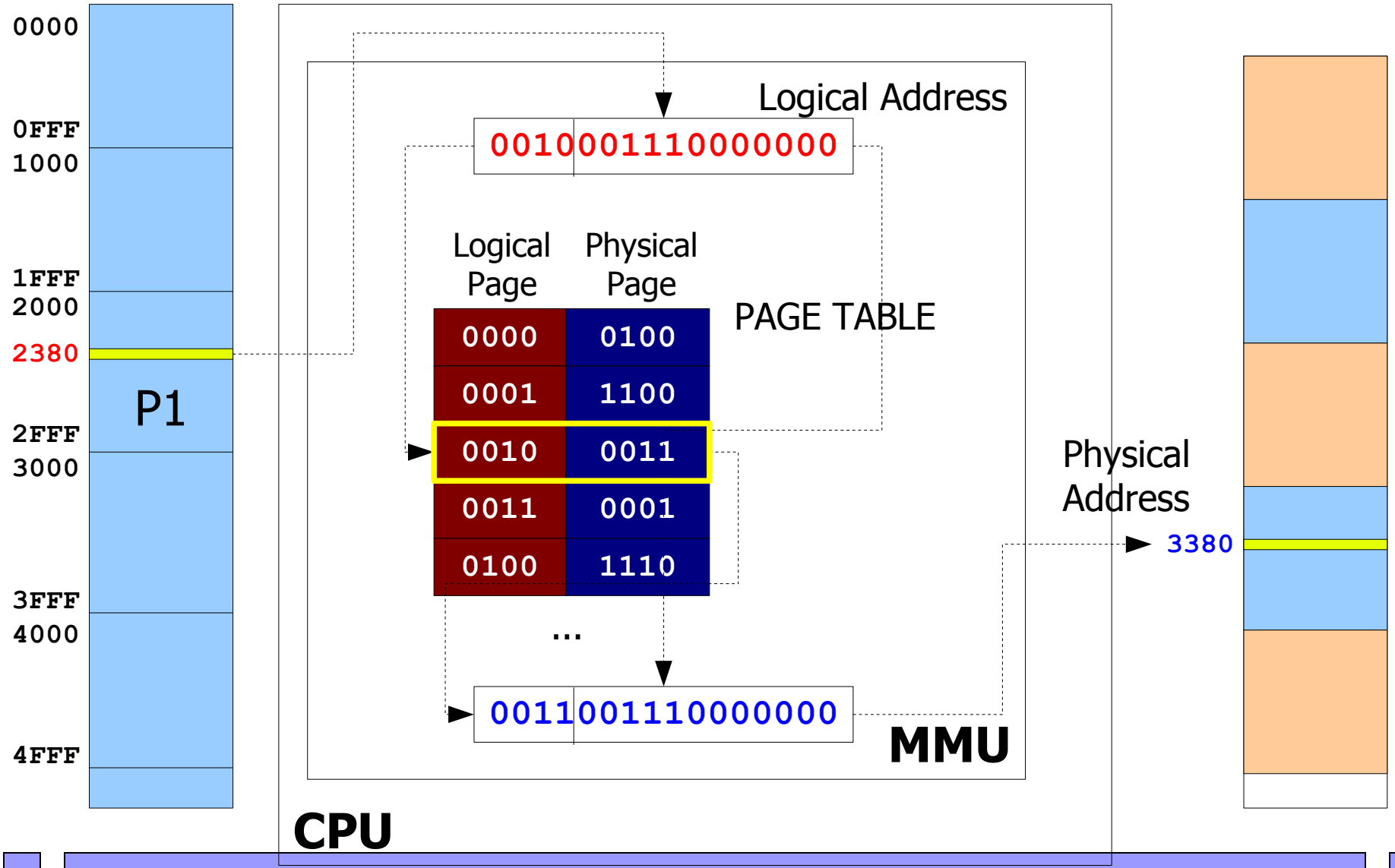


La pagina logica 2 del processo P1 è mappata nella pagina fisica 3

Memory Management Unit



Memory Management Unit



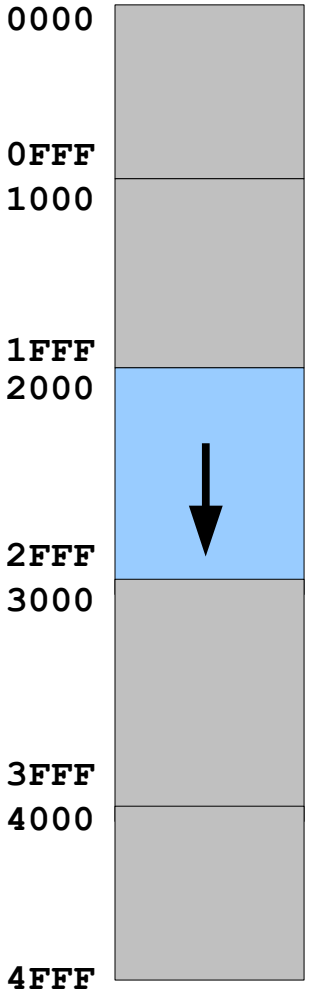
Paging: riassunto

- Gli spazi di indirizzamento logico e fisico sono suddivisi in pagine di dimensioni prefissate (potenze del 2)
- Il processo deve leggere o scrivere una cella di memoria
- La CPU riceve un **indirizzo logico** e lo passa alla MMU
- La MMU
  - separa numero di **pagina logica** e **offset**
  - tramite la tabella delle pagine, trasforma il numero di pagina **logica** in **numero di pagina fisica**
  - accoda l'offset e genera l'**indirizzo fisico**
- L'indirizzo fisico viene generato sul bus e l'operazione di lettura/scrittura viene infine eseguita

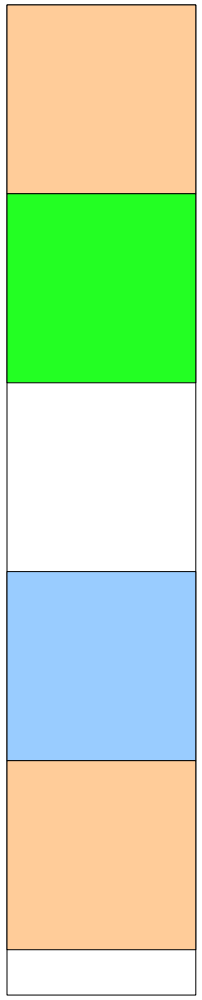
Demand Page

- Adesso abbiamo il controllo dello spazio di indirizzi a livello di pagina
- Tuttavia quando un processo gira esegue le istruzioni contenute in una pagina specifica ...
- ... con l'andare avanti dell'esecuzione, si passa ad eseguire il codice della pagina successiva, etc. etc.
- IN DEFINITIVA, AL PROCESSO SERVE UNA PAGINA PER VOLTA
- ALLORA SE ABBIAMO NECESSITA' DI MEMORIA, E NON C'E' PIU' SPAZIO, POSSIAMO MOMENTANEAMENTE **TOGLIERE AD UN PROCESSO LE PAGINE CHE NON STA USANDO**

Demand Page

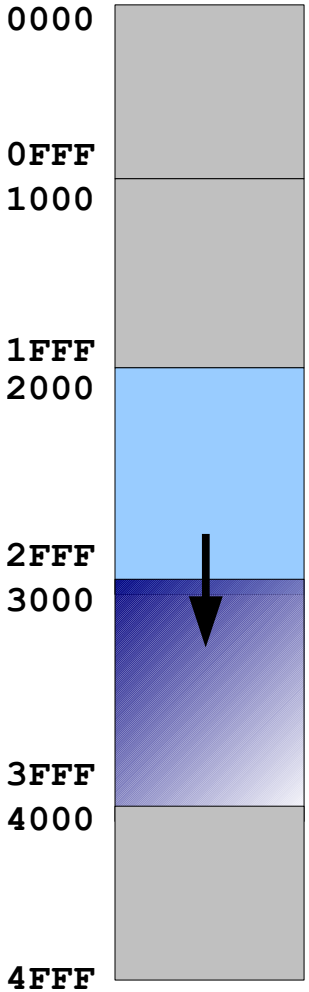


Il processo sta eseguendo il codice presente nella pagina (logica) 2 che è mappata nella pagina (fisica) 3



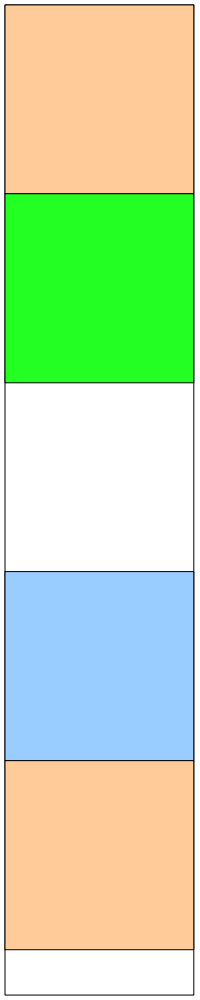
Le altre pagine del processo NON SONO PRESENTI IN MEMORIA FISICA

Demand Page

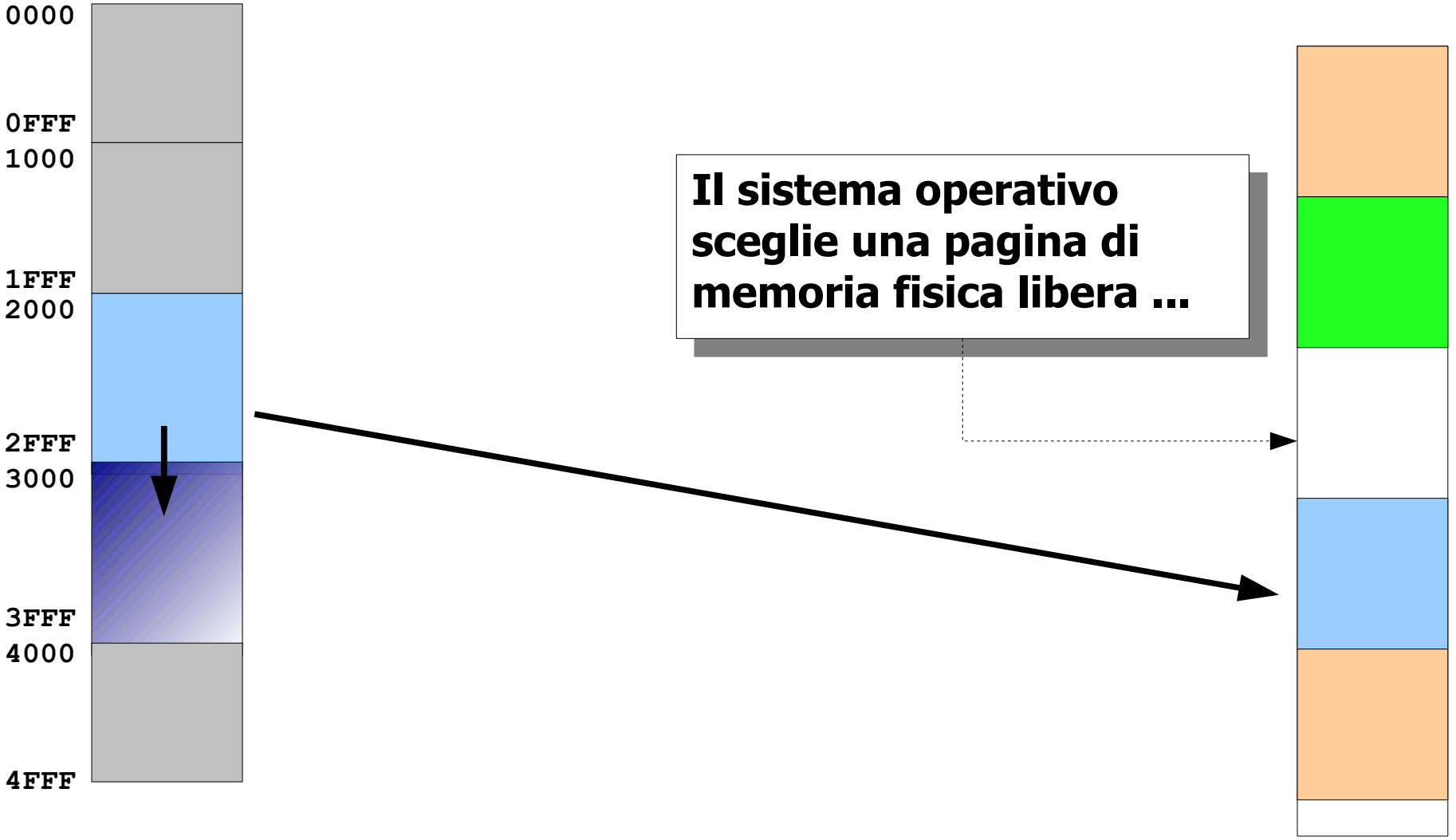


Il processo dovrebbe proseguire con il codice presente nella pagina (logica) 3 che NON E' PRESENTE IN MEMORIA

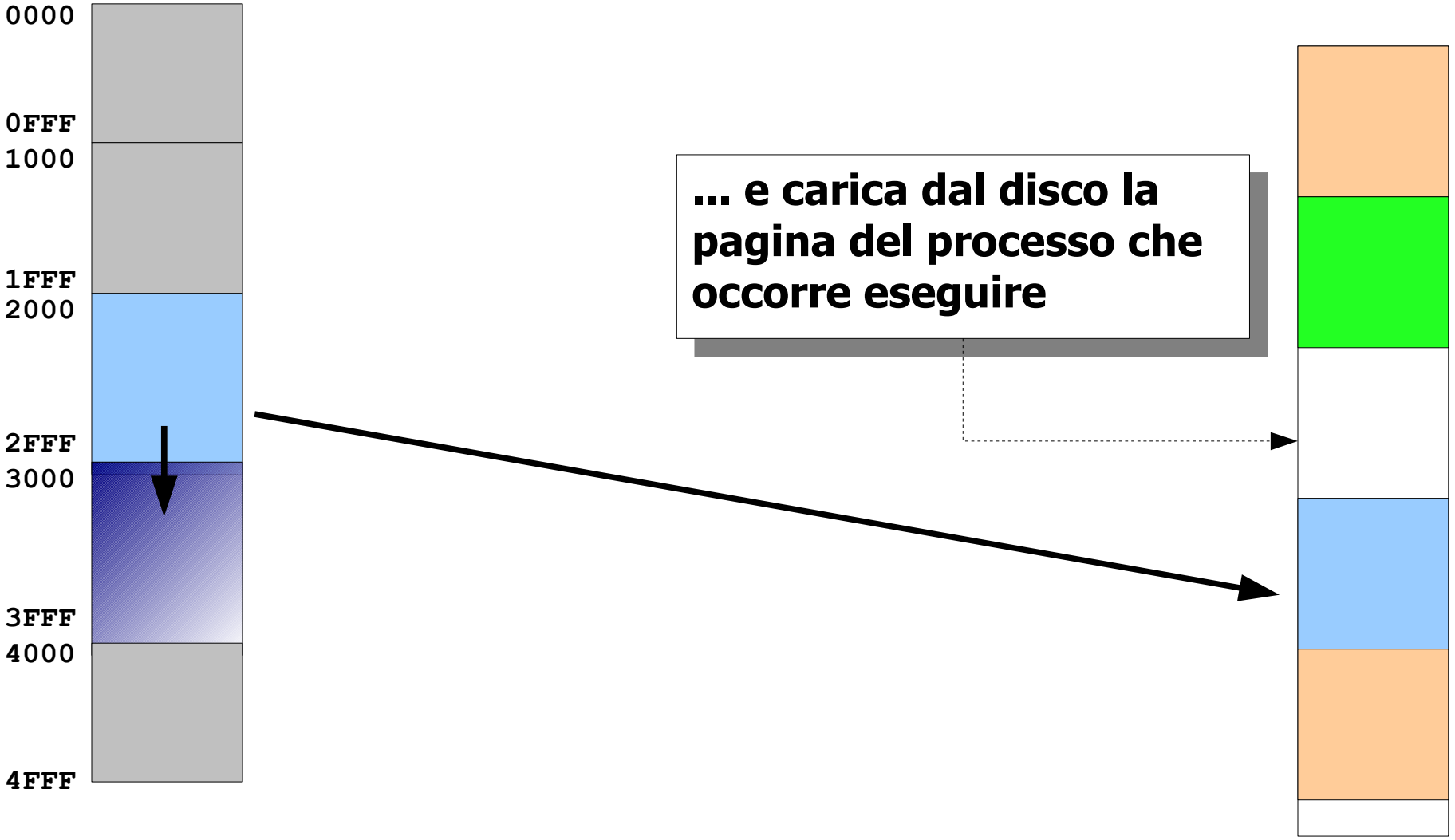
Avviene un evento denominato PAGE FAULT che attiva l'esecuzione del sistema operativo



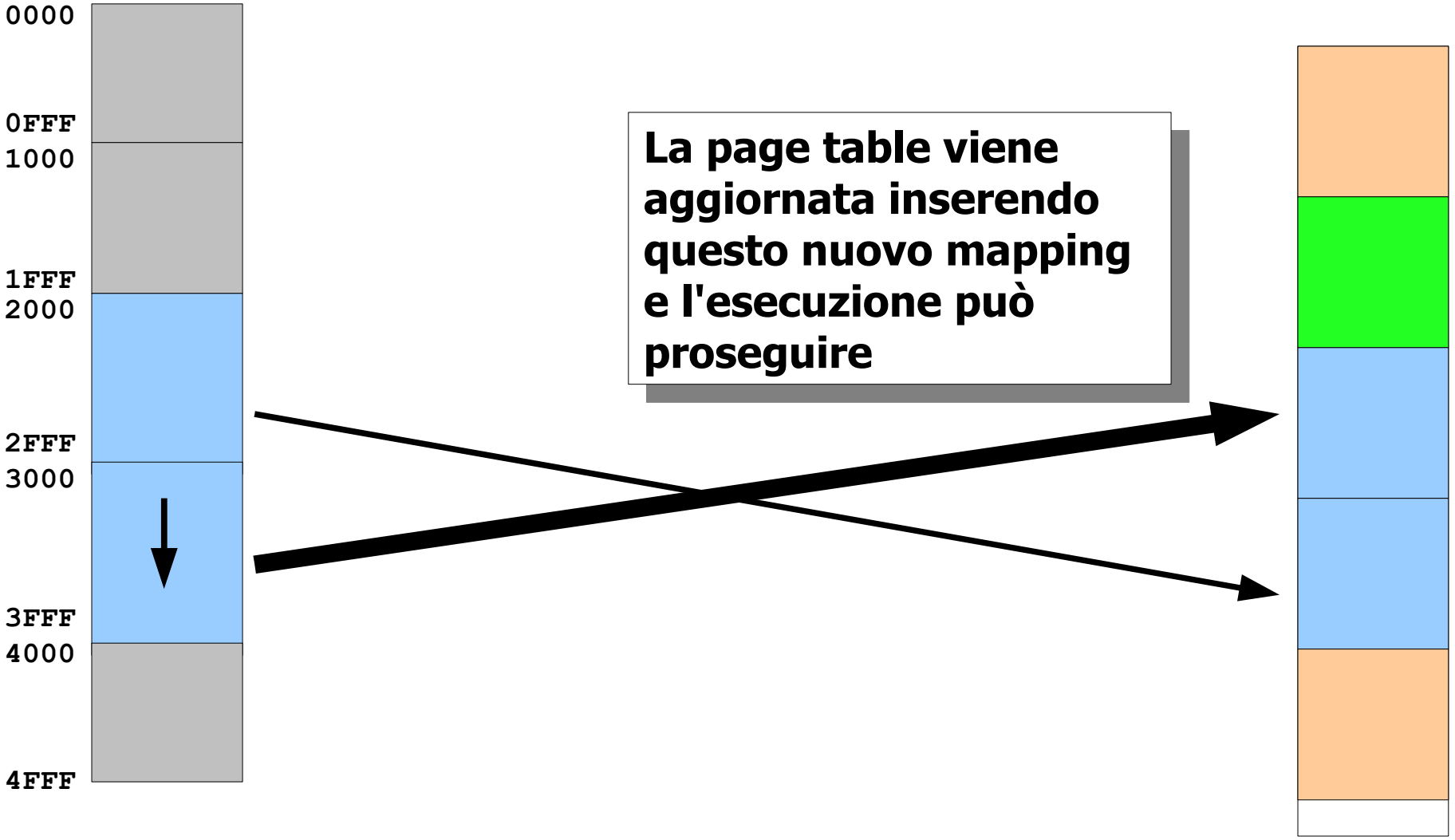
Demand Page


0000
0FFF
1000
1FFF
2000
2FFF
3000
3FFF
4000
4FFF
Il sistema operativo sceglie una pagina di memoria fisica libera ...

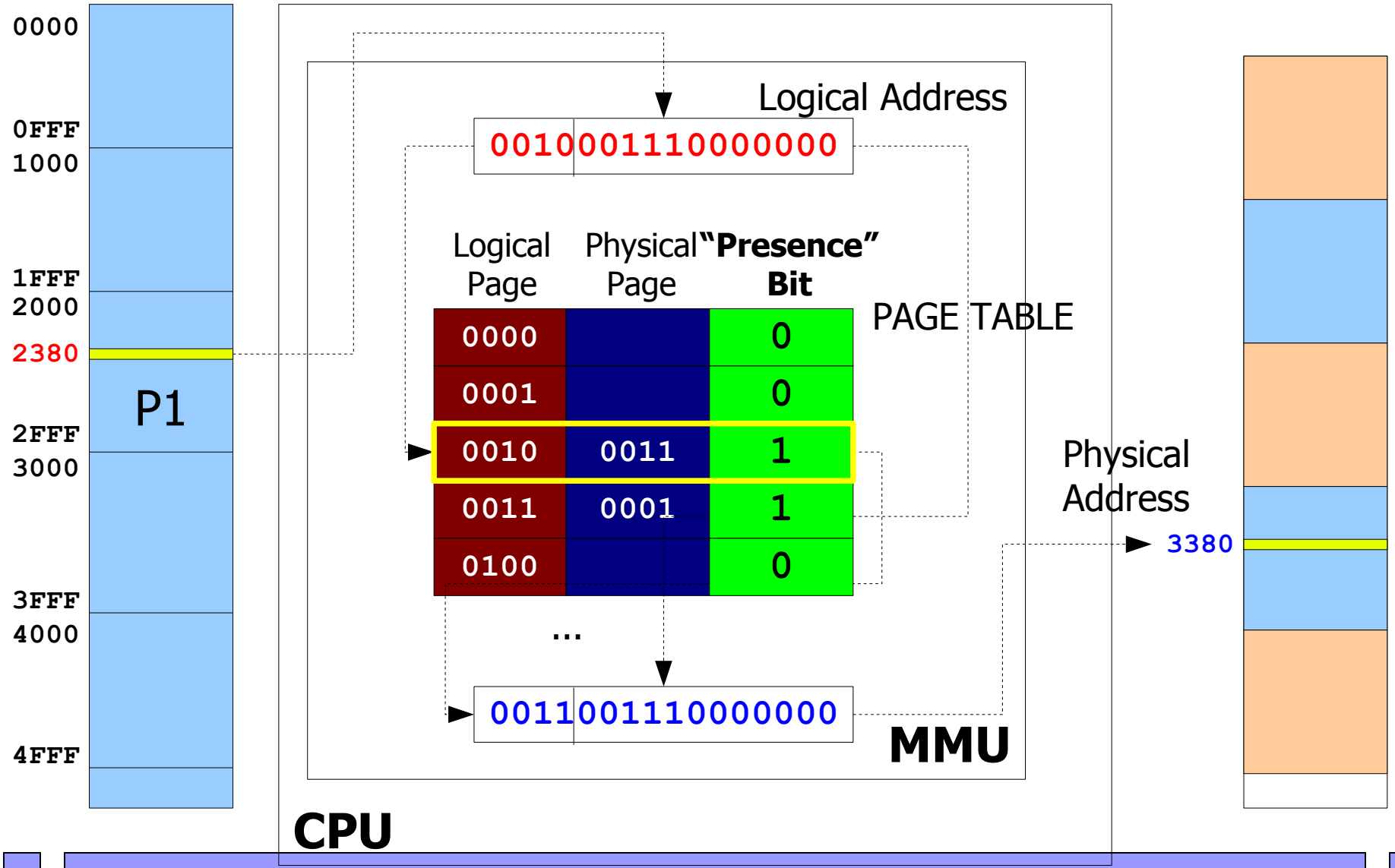
Demand Page


0000
0FFF
1000
1FFF
2000
2FFF
3000
3FFF
4000
4FFF
... e carica dal disco la pagina del processo che occorre eseguire

Demand Page


0000
0FFF
1000
1FFF
2000
2FFF
3000
3FFF
4000
4FFF
La page table viene aggiornata inserendo questo nuovo mapping e l'esecuzione può proseguire

Memory Management Unit



Infine ...

- Quando occorre caricare una pagina da disco ...
- Se ci sono pagine fisiche libere, allora nessun problema
- Ma se tutta la memoria è occupata? Occorre "scaricare" una pagina
- Si sceglie una pagina occupata
  - **la si "toglie" al processo proprietario**
  - **la si assegna al nuovo processo**
- Quali pagine sono candidate?
  - **Ovviamente quelle dei processi in WAIT**
  - E poi le altre ...

Conclusioni

- Il paging è la soluzione ottimale per la gestione della memoria
- Si basa sul suddividere la **memoria logica e fisica** in **pagine** di dimensione fissata
- Un componente della CPU, la **MMU (Memory Management Unit)** traduce l'indirizzo virtuale in indirizzo fisico tramite la **page table**
- La page table riporta la corrispondenza tra **pagina logica** e **pagina fisica**
- Grazie alla MMU, non tutte le pagine devono essere caricate in memoria, ma solo quelle utili al momento
- La page table possiede un **bit presente/assente** che indica se la pagina è presente o meno in memoria