

# Gestione della Memoria



# Scenario

- Sistema di elaborazione con 1 o più CPU
- Un certo quantitativo di memoria
- Una memoria di massa

# Sommario

- Gestione della Memoria Virtuale
- Protezione
- *Loading*

# Una soluzione comune

- Spazio degli indirizzi reali

- `virtualByte mem[M];`  
`// dove M = 512MB, 1GB ...`

- Diviso in pagine:

- `byte P[ X ][4096]` // X dipende dalla memoria

- Spazio degli indirizzi virtuali (1 per processo)

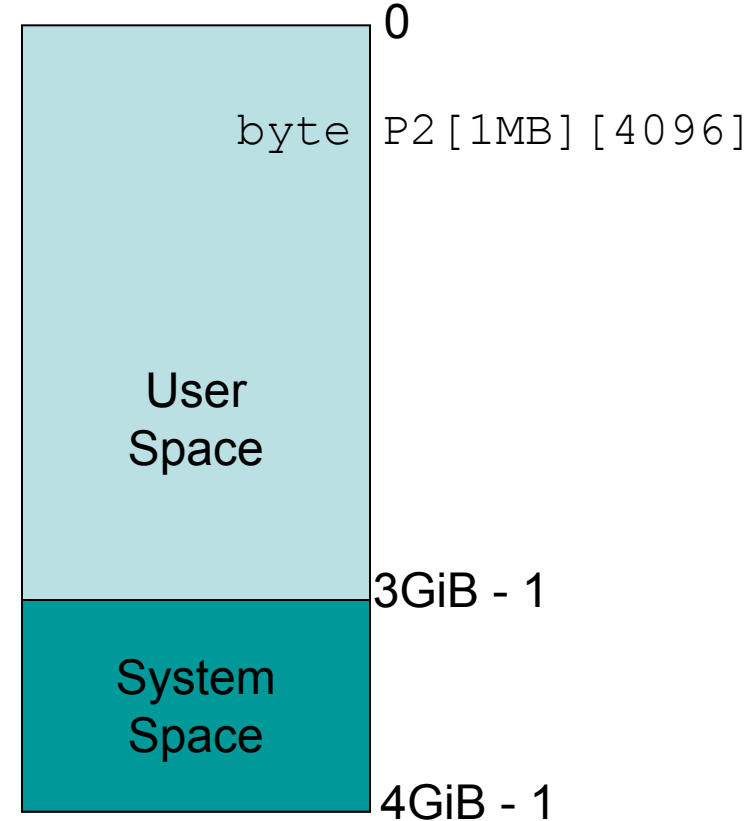
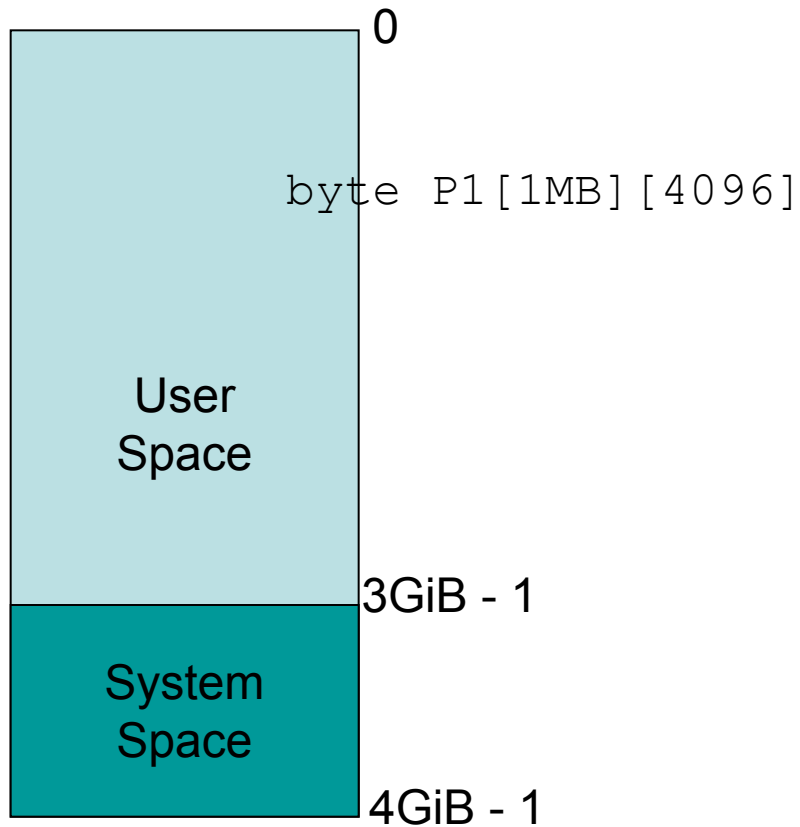
- `virtualByte memPn[N];` //  
`N = 4GB per intel x32`

- Diviso in pagine:

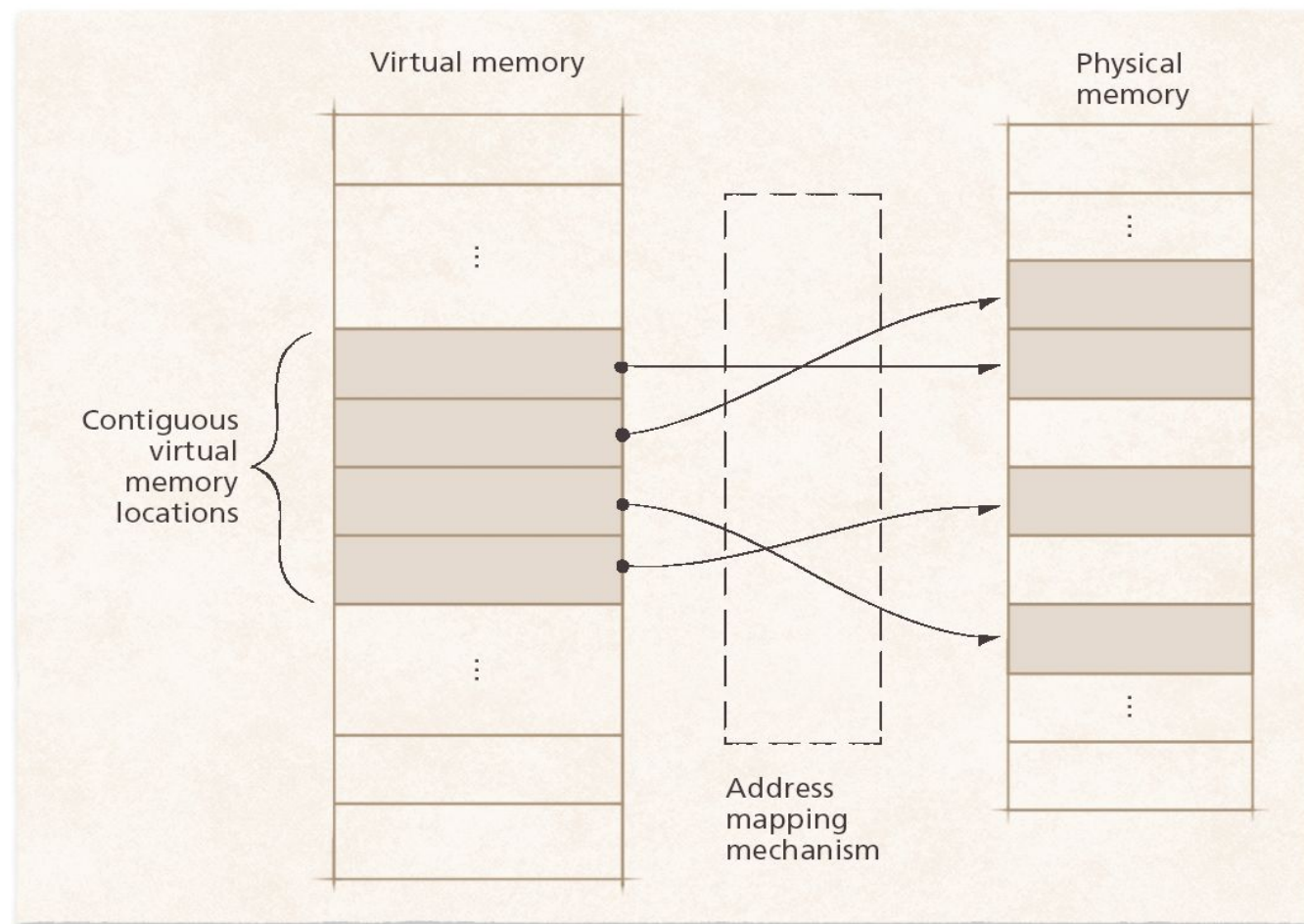
- `byte Pn[ 1MB ][4096]`

# Ogni processo viene “illuso”...

- ....di avere a disposizione 4 GiB (OS a 32bit) o  $2^{64}$  bit (OS a 64bit)

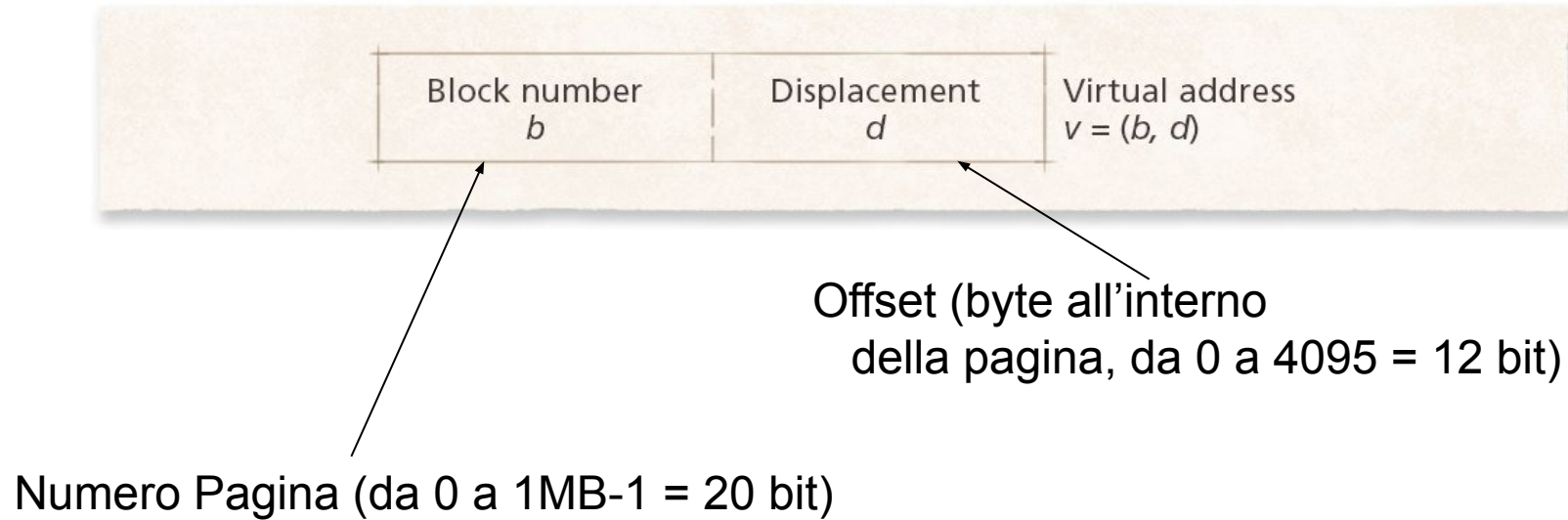


# Dov'è il trucco?





# Suddivisione dell'indirizzo



# Gli indirizzi di memoria sono virtuali

- Assumiamo di avere un solo processo P1, e di avere 4GiB di memoria fisica.
- Immaginiamo di poter esaminare il microcodice delle istruzioni macchina

```
- mov AL,indirizzo      ** LETTURA (LDRB in Arm) **  
- mov indirizzo, AL     ** SCRITTURA (STRB in Arm) **
```

```
...  
AL = memP1[indirizzo]; // lettura  
...  
oppure...  
...  
memP1[indirizzo] = AL; // scrittura  
...
```

```
byte& virtualByte::operator[] (long indirizzo)  
{  
    pagina = indirizzo >> 12;  
    offset = indirizzo % 4096;  
    return P[pagina][offset];  
}
```



# Purtroppo...

- C'è più di un processo.
- L'array mem può addirittura essere più piccolo di 4GiB;

## Soluzione

- Introduzione di una tabella delle “Page Entries”: una per ogni processo.

```
typedef pagina byte[4096];
pagina P[X];    // X dipende dalla memoria RAM reale.
class PageEntry {
    bool inMemoria;
    bool acceduto;
    bool scritto;
    long posizione;
    long posizioneSuDisco;
    bool leggibile;
    bool scrivibile;
    bool eseguibile;
}

class process {
    ....
    PageEntry pageDirectory[1MB];
    virtualByte mem(4GB, pageDirectory);
}
```

# Il vero operator[] è simile a...

```
byte& virtualByte::operator[] (long indirizzo)
{
    pagina = indirizzo >> 12;
    offset = indirizzo % 4096;
    PageEntry pe = pageDirectory[pagina];
    if (!pe.inMemoria)
    {
        // page fault.
        pe.posizione = trovaPaginaLibera();
        loadPage(pe.posizioneSuDisco); // swap in
        pe.scritto = false; // pagina fresca presa da disco
    }
    pe.acceduto = true;
    if (operazione di Scrittura)
        pe.scritto = true;
    return P[pe.posizione][offset];
}
```

# Swap in e Swap out

- Ci sono dei thread ad altissima priorità che si occupano di
  - Caricare in anticipo le pagine che si prevede siano usate (Swap in)
  - Eliminare le pagine non usate (swap out)
- Linux: è il demone `kswapd`
- Windows: pool di thread di sistema

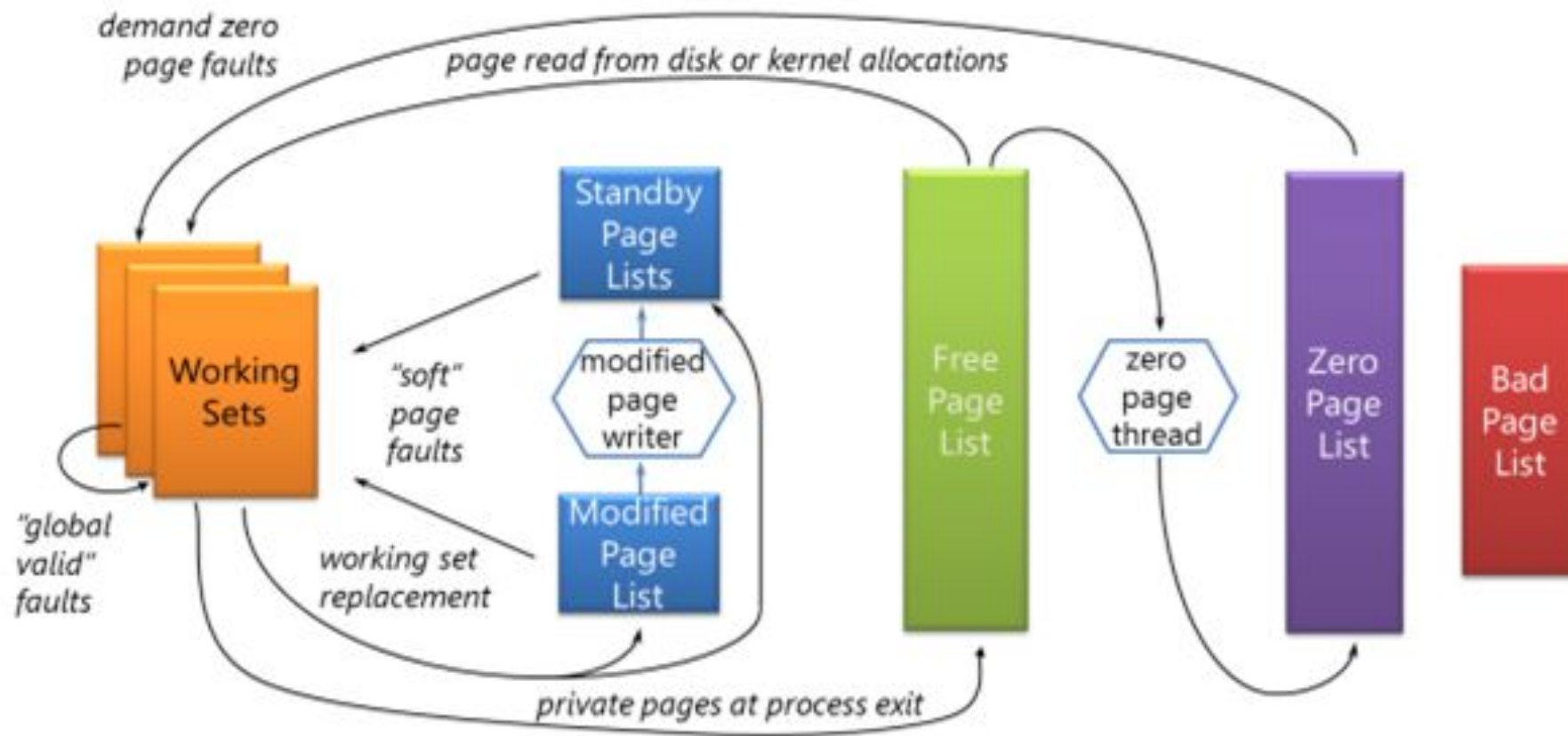
# Swap out

- Semplice algoritmo: le pagine stanno in una lista FIFO. Quando una pagina  $p$  è acceduta, viene settato il bit  $p.\text{acceduto}$  a 1, e viene messa in testa alla FIFO.
- Un thread “pulitore”, elimina periodicamente dalla fine della coda (swap out) quelle pagine  $p$  in cui trova  $p.\text{acceduto} == 0$ . Pone  $p.\text{acceduto} = 0$  per tutte le altre pagine
- Le pagine accedute periodicamente tendono ad evitare di essere “swapped out”.
- Eccezioni: pagine marcate come inamovibili, working set
- Algoritmo reale.. un po’ più complicato.
- Il “pulitore” è tanto più aggressivo tanto più c’è meno memoria.

# Swap out (Trimming)

```
void swapout (PageEntry p)
{
    if (p.scritto)
        p.posizioneSuDisco
        =scriviSuDisco (p.posizione);
    p.inMemoria = false;
}
```

# Windows Memory Lifecycle

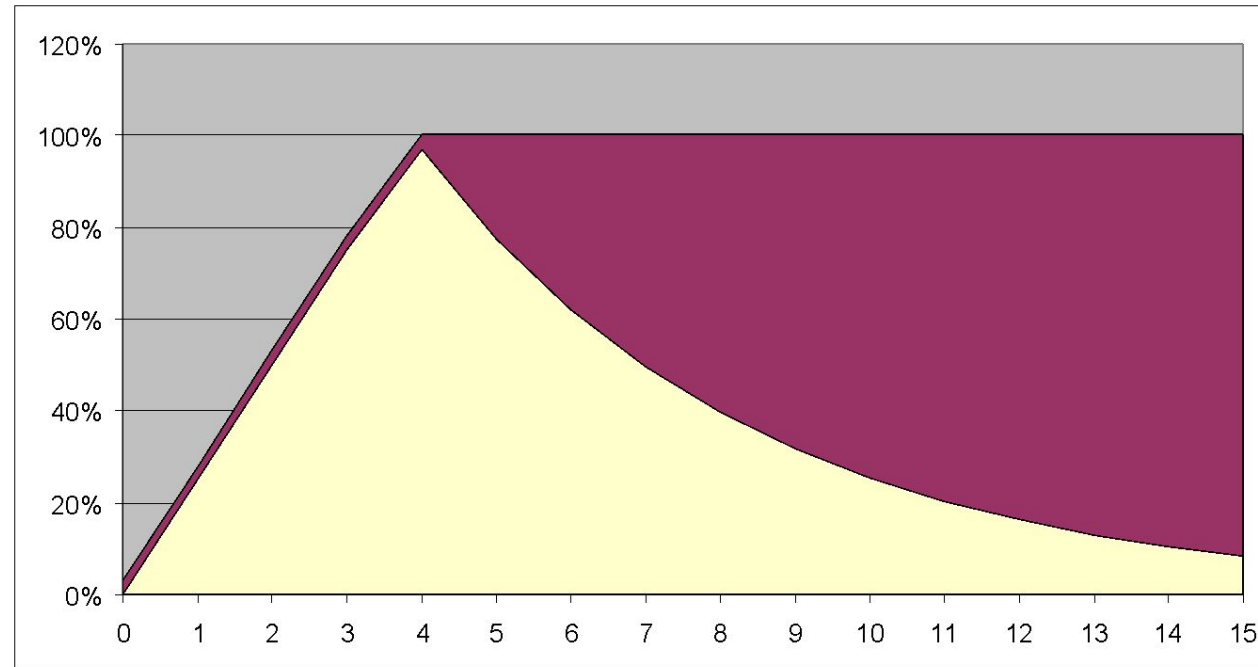


# Strumenti di diagnostica

- Windows
  - vmmap □ mappa per singolo processo
  - rammap □ mappa di sistema
- Linux
  - top
  - vmstat
  - htop



# Thrashing



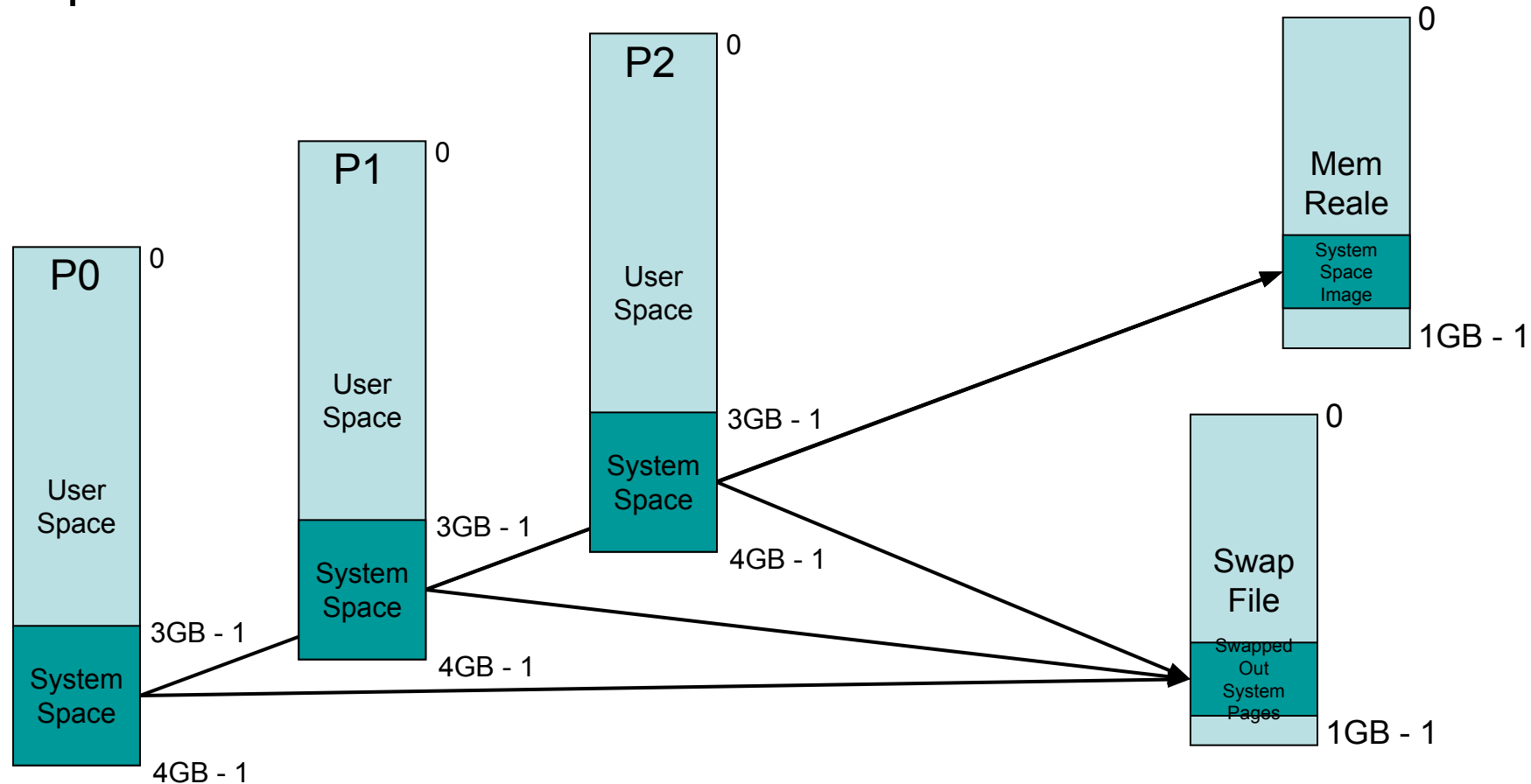
Ipotesi:

Thread tutti uguali: occupazione 256MiB, 25% occupazione media processore

Memoria centrale: 1GiB

# Protezione

- E' fisicamente impossibile (o quasi) che un processo acceda alla memoria di un altro processo



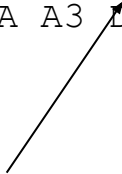
# Protezione – 2

- I processi tuttavia condividono lo stesso system space
  - Ci sono pagine fisiche riferite da page entries di più processi
  - L'accesso in lettura/scrittura può essere proibito tramite i valori di  
`p.leggibile` e `p.scrivibile`
  - Si può proibire anche l'eseguibilità (DEP: Data execution prevention)

# Loading e rilocalizzazione


004010CA	8B 1D 50 8D 42 00	mov	ebx,dword ptr [a (00428d50)]
004010D0	A1 D0 99 42 00	mov	eax,[b (004299d0)]
<b>004010D5</b>	<b>F7 E3</b>	mul	eax,ebx
004010D7	4B	dec	ebx
004010D8	75 <b>FB</b>	jne	ciclo (004010d5)
004010DA	A3 D0 99 42 00	mov	[b (004299d0)],eax

11111011 = -5



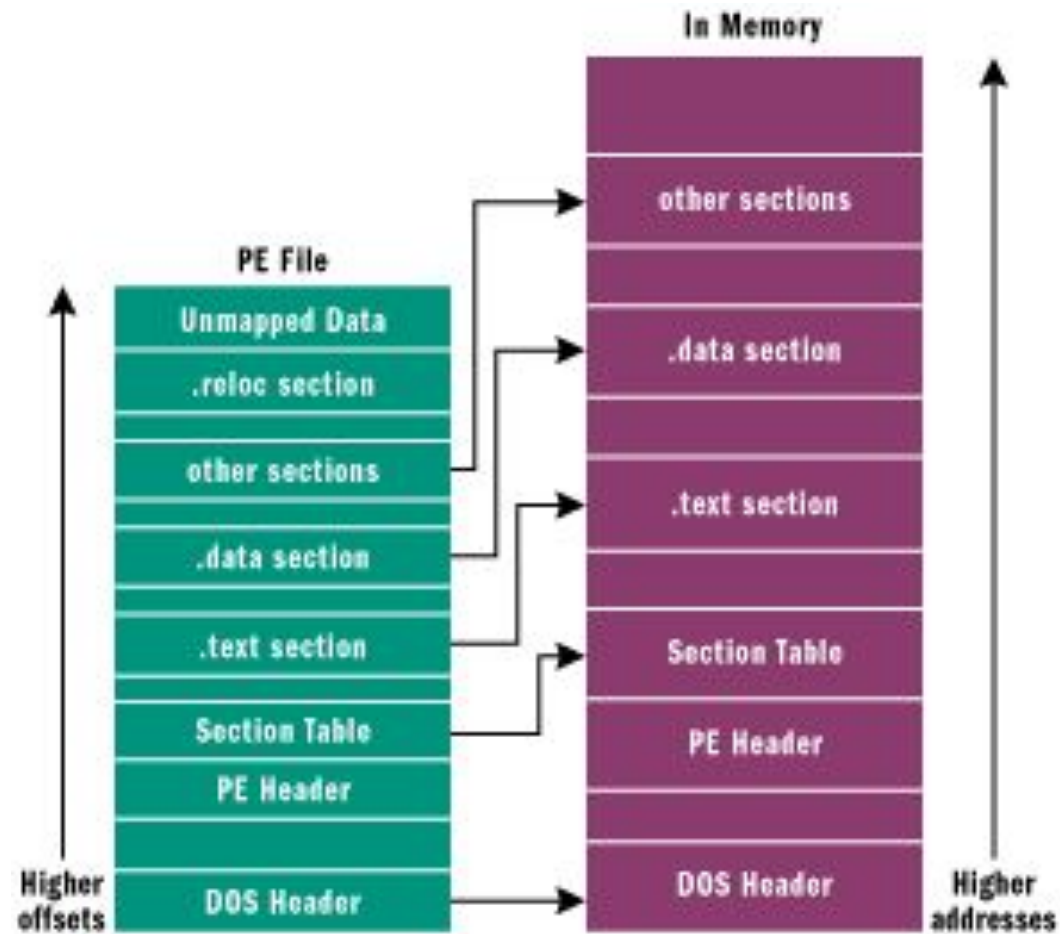
Codice  
indipendente dalla  
posizione

# Codice non rilocabile

004010EA B9 E0 99 42 00	mov	ecx,offset cout (004299e0)
004010EF E8 8C 01 00 00	call	ostream::operator<< (00401280)
004010F4 8B C8	mov	ecx,eax
004010F6 E8 0A FF FF FF	call	@IAT+0(ostream::operator<<) (00401005)
		
00401005 E9 36 01 00 00	jmp	ostream::operator<< (00401140)
0040100A E9 E1 01 00 00	jmp	ostream::operator<< (004011f0)
0040100F E9 8C 01 00 00	jmp	endl (004011a0)
00401014 E9 27 00 00 00	jmp	main (00401040)
00401019 E9 22 02 00 00	jmp	flush (00401240)

ASLR ☐ Address space layout randomization

# PE : Portable Executable



# Mobile OSes

- Android: cambio di paradigma
  - No swap space
  - Un processo può essere killed per poter liberare memoria
  - OOM Manager (Out of Memory Manager) diventa modulo cruciale