Dynamic, variable-sized partitions

- Basic idea
  - Start with *un-partitioned* memory (a.k.a. "*free space*")
  - Create partitions *dynamically* based on process *data size*
    - *Number* of partitions changes dynamically
    - *Lengths* of partitions may be different

Dynamic, variable-sized partitions

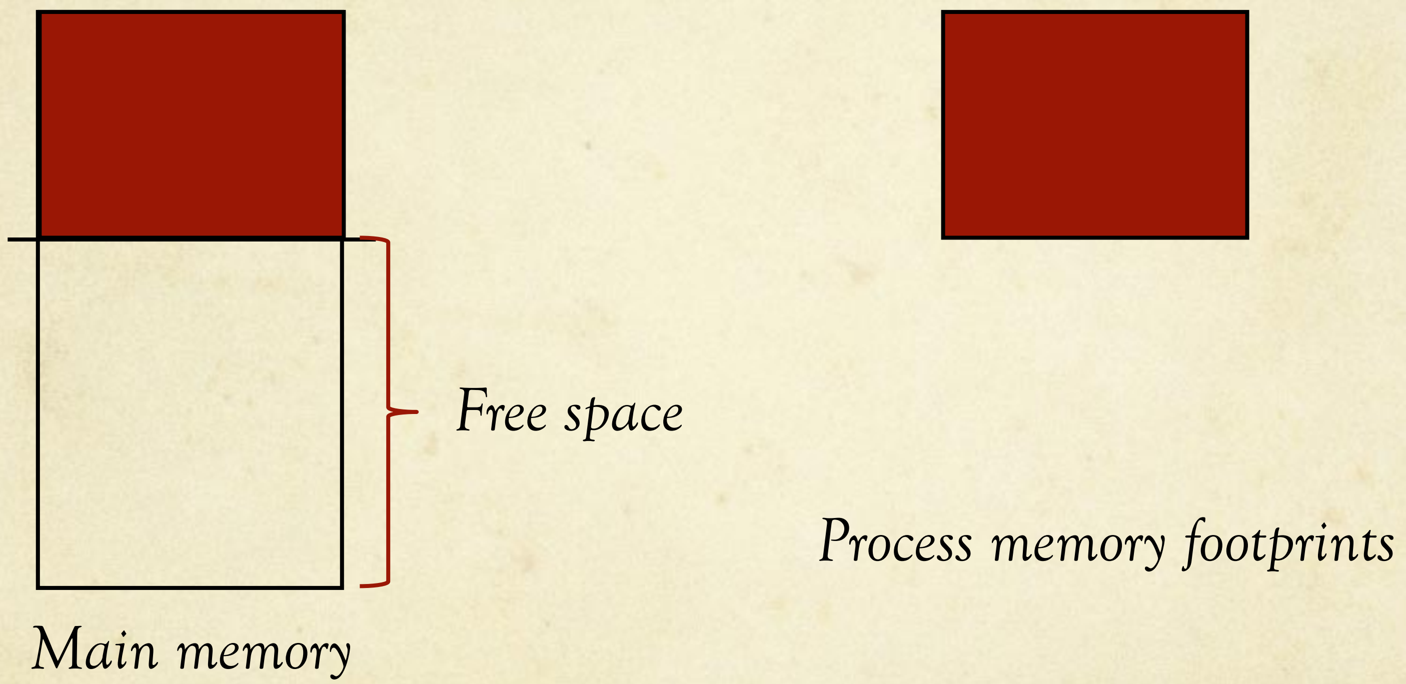
- Place initial partitions *contiguously* in main memory

Main memory

Process memory footprints

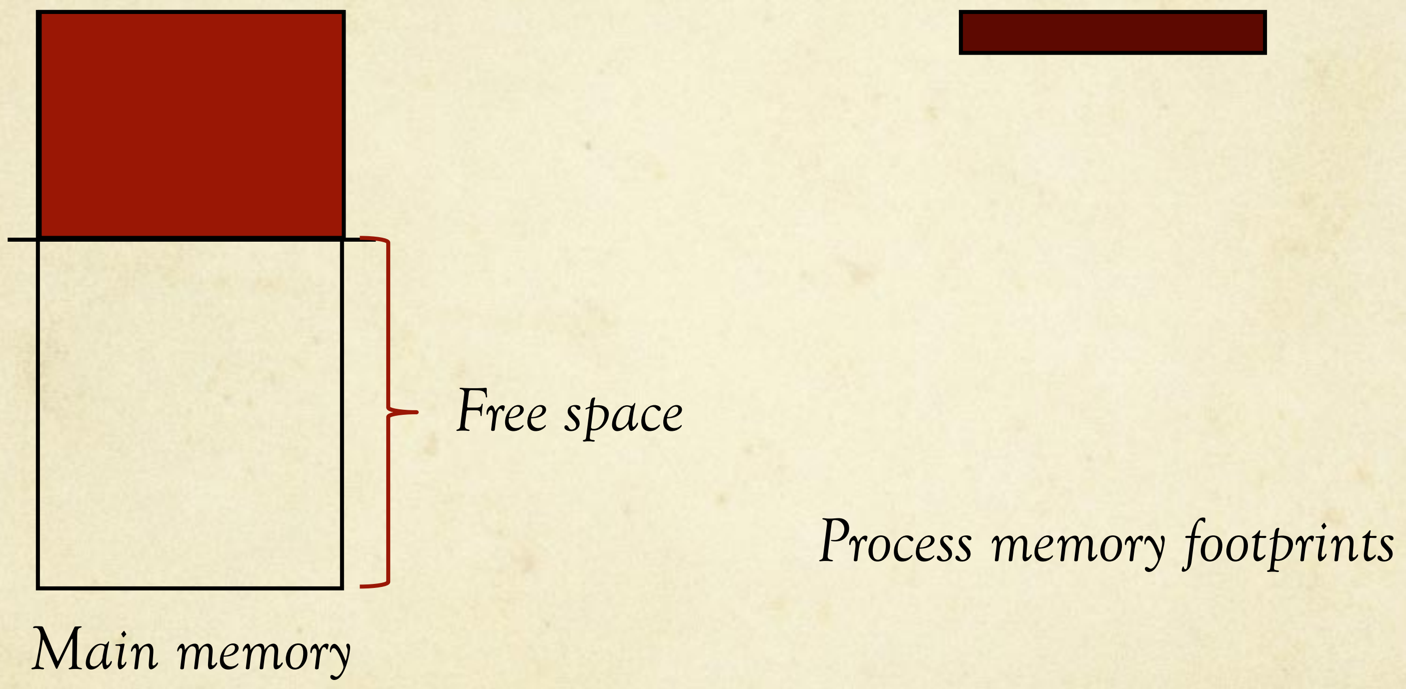
Dynamic, variable-sized partitions

- Place initial partitions *contiguously* in main memory



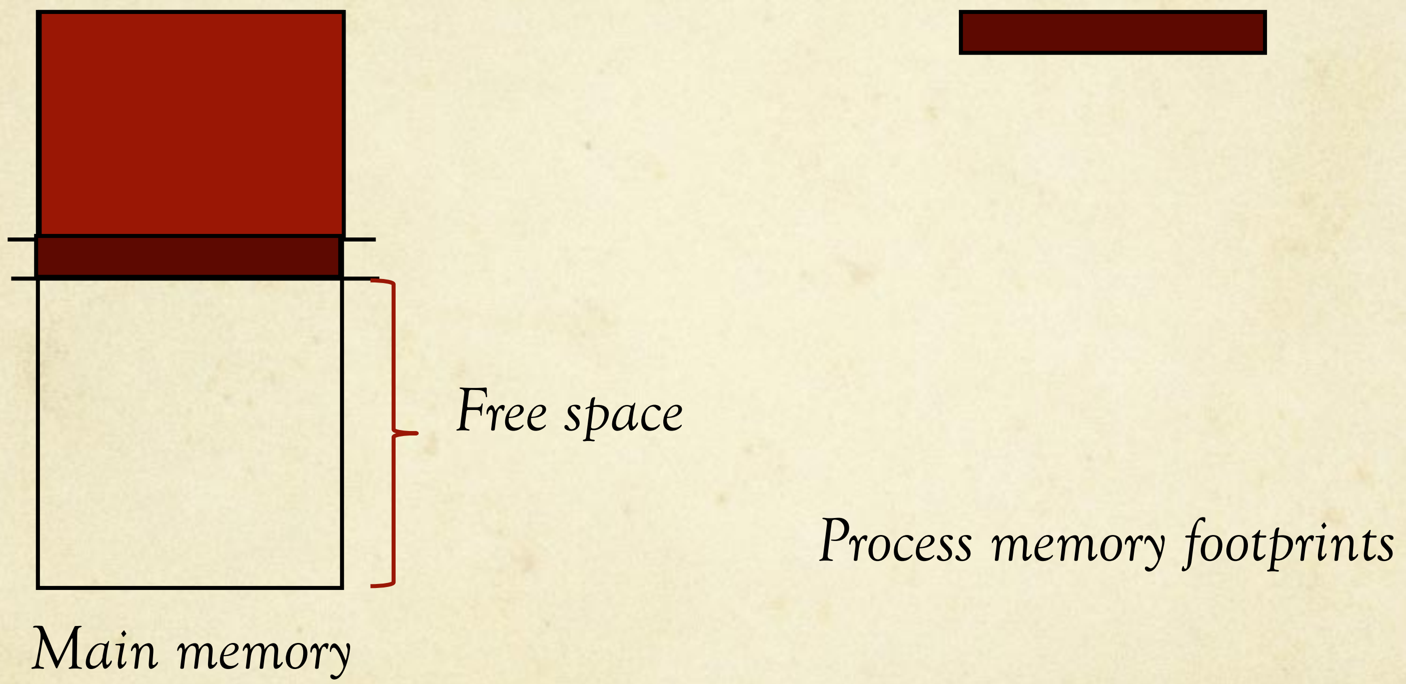
Dynamic, variable-sized partitions

- Place initial partitions *contiguously* in main memory



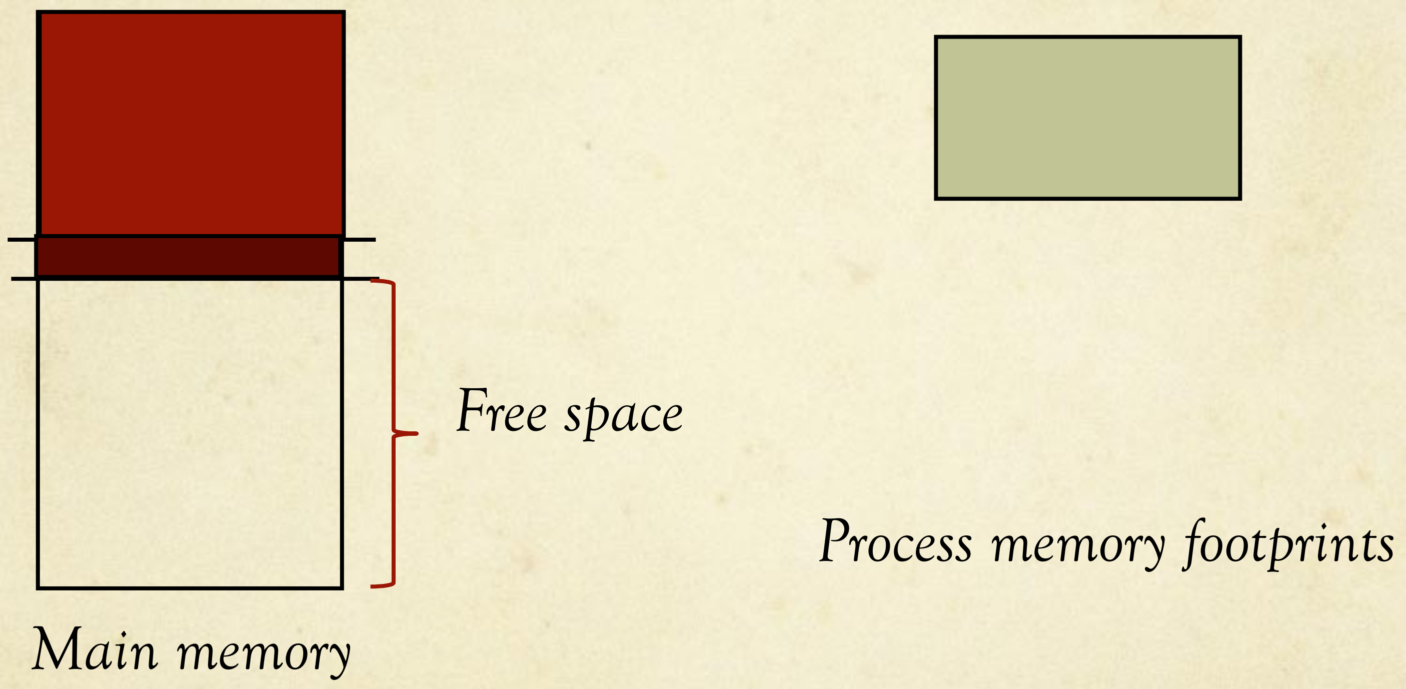
Dynamic, variable-sized partitions

- Place initial partitions *contiguously* in main memory



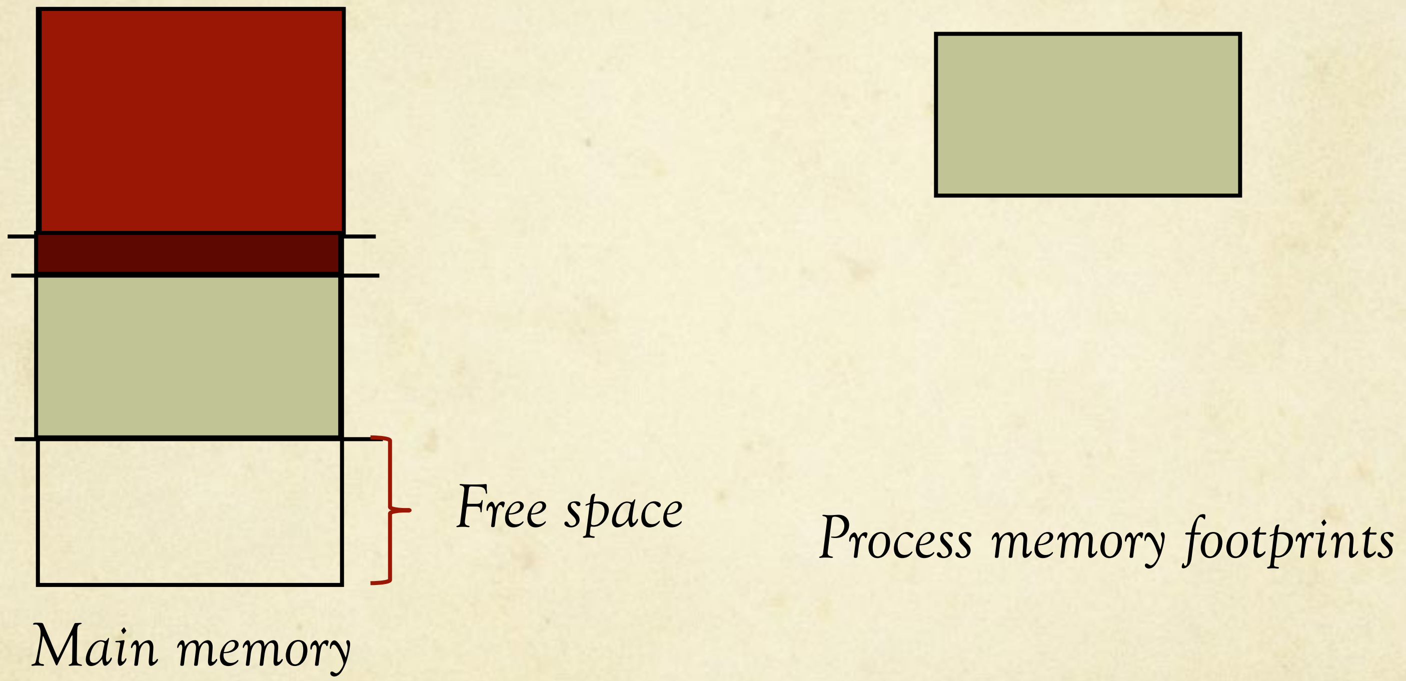
Dynamic, variable-sized partitions

- Place initial partitions *contiguously* in main memory



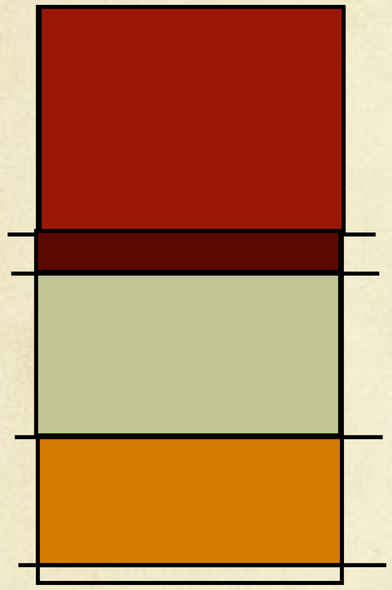
Dynamic, variable-sized partitions

- Place initial partitions *contiguously* in main memory



Dynamic, variable-sized partitions

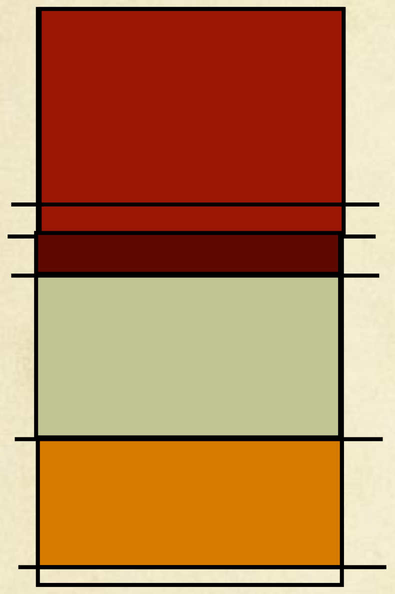
- Place initial partitions *contiguously* in main memory


Main memory

Process memory footprints

Dynamic, variable-sized partitions

- Place initial partitions *contiguously* in main memory

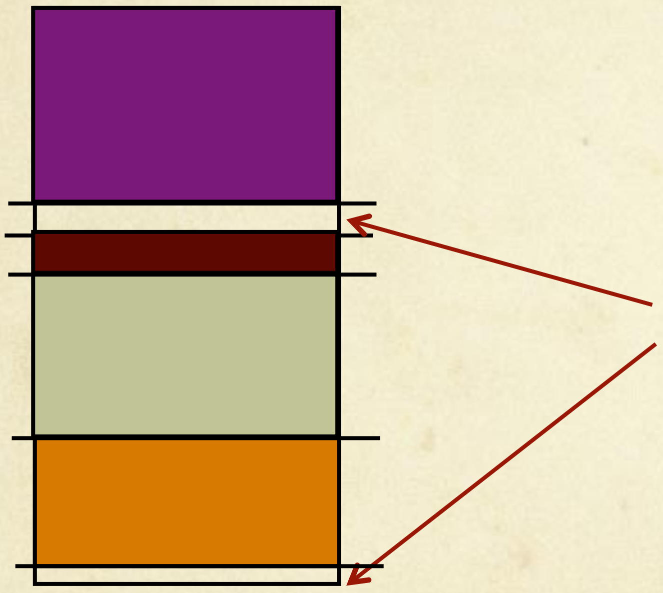


Main memory

Process memory footprints

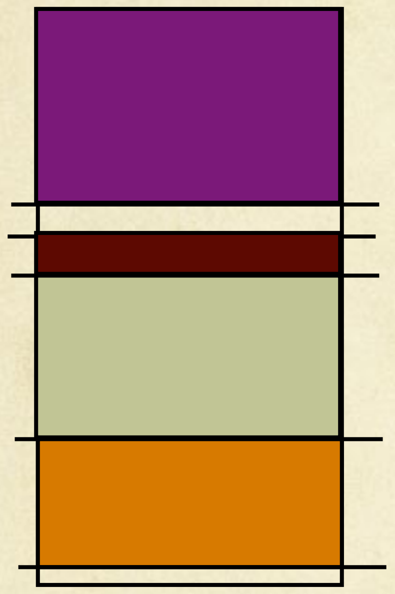
Dynamic, variable-sized partitions

- Place initial partitions *contiguously* in main memory


Main memory

Dynamic, variable-sized partitions

- Place initial partitions *contiguously* in main memory



Main memory

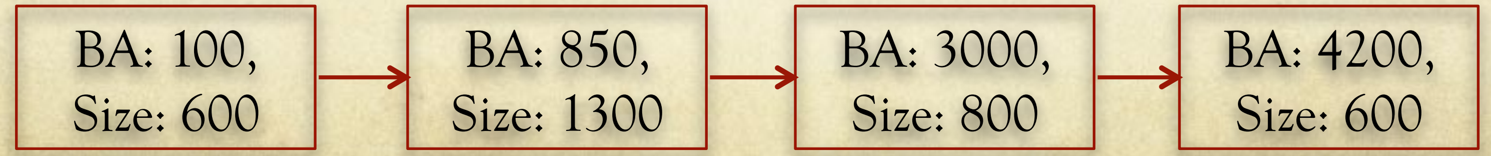
External fragmentation

- As processes get swapped in & out of main memory
  - Memory ends up with "*free spaces*"/"*holes*" b/w partitions

Dynamic, variable-sized partitions

- Consequences
  - Need *mechanism* to *keep track* of free spaces!
  - Need *policy* for *partition placement*

- Possible *mechanism* to keep track of free spaces
  - Maintain *linked list* of free spaces
  - Each list element stores *base address* & *size* of free space



Polices for partition placement

- *First fit* (*FF*)
  - Scan list, choose *first* free space *large enough* for partition
  - *Pros*: simple; fast
  - *Cons*: partitions may *crowd* initial regions of main memory
- *Next fit* (*NF*)
  - Similar to first fit; scanning starts where previous scan ended
  - *Pros*: more distributed allocation

Polices for partition placement

- *Best fit* (*BF*)
  - Scan entire list; choose *smallest* free space *large enough*
  - Cons: slow; leaves many small unusable free spaces
- *Worst fit* (*WF*)
  - Scan entire list; choose *largest* free space that fits process
  - *Pros*: leaves larger free spaces
  - Cons: slow

First fit

Memory address range: 0 – 4999

Process memory requirements

P1: 500	P3: 750
P2: 800	P4: 1200
P5: 900	

$$\begin{tikzcd}
\boxed{\begin{matrix}\text{BA: 100,}\\ \text{Size: 600}\end{matrix}} \arrow[r] & \boxed{\begin{matrix}\text{BA: 850,}\\ \text{Size: 1300}\end{matrix}} \arrow[r] & \boxed{\begin{matrix}\text{BA: 3000,}\\ \text{Size: 800}\end{matrix}} \arrow[r] & \boxed{\begin{matrix}\text{BA: 4200,}\\ \text{Size: 600}\end{matrix}}
\end{tikzcd}$$

P2 Address 850

$$\begin{tikzcd}
\boxed{\begin{matrix}\text{BA: 100,}\\ \text{Size: 600}\end{matrix}} \arrow[r] & \boxed{\begin{matrix}\text{BA: 1650,}\\ \text{Size: 500}\end{matrix}} \arrow[r] & \boxed{\begin{matrix}\text{BA: 3000,}\\ \text{Size: 800}\end{matrix}} \arrow[r] & \boxed{\begin{matrix}\text{BA: 4200,}\\ \text{Size: 600}\end{matrix}}
\end{tikzcd}$$

P1 Address 100

$$\begin{tikzcd}
\boxed{\begin{matrix}\text{BA: 600,}\\ \text{Size: 100}\end{matrix}} \arrow[r] & \boxed{\begin{matrix}\text{BA: 1650,}\\ \text{Size: 500}\end{matrix}} \arrow[r] & \boxed{\begin{matrix}\text{BA: 3000,}\\ \text{Size: 800}\end{matrix}} \arrow[r] & \boxed{\begin{matrix}\text{BA: 4200,}\\ \text{Size: 600}\end{matrix}}
\end{tikzcd}$$

Best fit

Memory address range: 0 – 4999

Process memory requirements

P1: 500 P3: 750

P2: 800 P4: 1200

P5: 900

$$\begin{tikzcd}
\boxed{\text{BA: 100, Size: 600}} \arrow[r] & \boxed{\text{BA: 850, Size: 1300}} \arrow[r] & \boxed{\text{BA: 3000, Size: 800}} \arrow[r] & \boxed{\text{BA: 4200, Size: 600}}
\end{tikzcd}$$

P2 Address 3000

$$\begin{tikzcd}
\boxed{\text{BA: 100, Size: 600}} \arrow[r] & \boxed{\text{BA: 850, Size: 1300}} \arrow[r] & \boxed{\text{BA: 4200, Size: 600}}
\end{tikzcd}$$

P1 Address 100

$$\begin{tikzcd}
\boxed{\text{BA: 600, Size: 100}} \arrow[r] & \boxed{\text{BA: 850, Size: 1300}} \arrow[r] & \boxed{\text{BA: 4200, Size: 600}}
\end{tikzcd}$$

Discussion

- *Pros*
  - More *flexible* than static partitioning
- *Cons*
  - Management more *complex* than with static partitioning
  - Has *external fragmentation* (extent varies with policy)

Now consider this…

- How should a program specify memory addresses?
  - Absolute addresses?
  - Definitely not for *dynamic* partitioning!
  - Probably not even for *static* partitioning → loses portability

In practice…

- Program uses offsets *relative* to start address of 0
    - I.e., program uses *logical* address
    - Range of logical addresses for process: **logical address space**
- System *translates* relative offset into *absolute* address
    - I.e., system generates *physical* address
    - Range of absolute addresses of process: **physical address space**
- Translation mechanism
    - Maintain *base address* & *limit* for process partition
    - Add base address to logical address issued by program
    - Check whether result is within limit

Consequence of relative addressing

- Process *need not* be mapped to same physical partition all the time...
- ...partition *relocation* is possible
- Very useful, especially when using *dynamic* partitioning