
Memory management

Context

- Process needs resources for execution
  - CPU cycles, I/O devices, etc.
  - *Memory* resources to store
    - Program code ("text")
    - Data
    - OS structures for process

We will hereafter refer to the set of all process memory resources as process data

Context

- For process to execute, its data must be in *main memory*
  - Recall that main memory is *volatile*
  - Process data stored in *non-volatile* memory, e.g., disk
  - Process data must be *loaded* into main memory from disk

Simple scenario

- Allow only *one process* to be active at a time
  - *Load* process data from disk to main memory
  - Once process is done, *store* its contents to disk

- In reality, a process
  - May wait for I/O → poor *CPU utilization*
  - May not use all memory → poor *memory utilization*

More realistic scenario…

- Just like *CPU* multiplexed among multiple processes…
  - …*memory* must be multiplexed among multiple processes
- What we need
  - *Multiple* processes should reside in memory *at a time*
  - Processes should not *collide* in memory
  - Process should not access other process' memory (*protection*)
  - Processes should be able to *share* memory if desired
- I.e., we need *memory protection* & *controlled overlap*

Memory partitioning

- Basic idea
  - Divide memory into multiple *partitions*
  - *Allocate* processes to partitions
  - Ensure *protection* across partitions
- Several questions arise
  - How should memory be divided into partitions?
  - Which process should be allocated to which partition?
  - Should partitions be allowed to change dynamically?
- Let's consider multiple options…

Static, equal-sized partitions

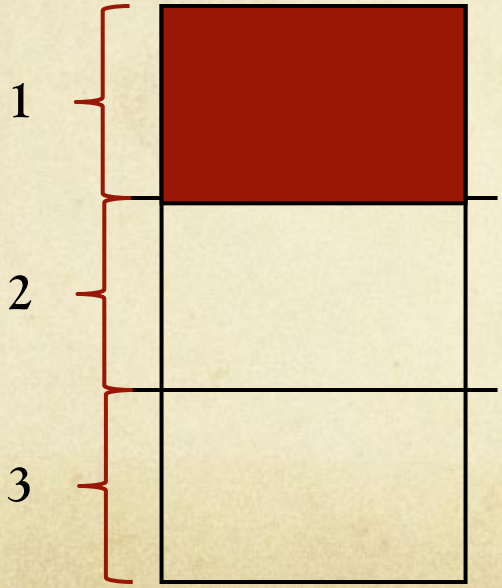
- *Statically* divide main memory into *equal-sized* partitions
- Map *each* process to *one* partition
  - For now, assume mapping does not change

Partitioned memory

Process memory footprints

Static, equal-sized partitions

- *Statically* divide main memory into *equal-sized* partitions
- Map *each* process to *one* partition
  - For now, assume mapping does not change



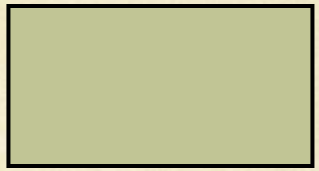
Partitioned memory

Process memory footprints

Static, equal-sized partitions

- *Statically* divide main memory into *equal-sized* partitions
- Map *each* process to *one* partition
  - For now, assume mapping does not change

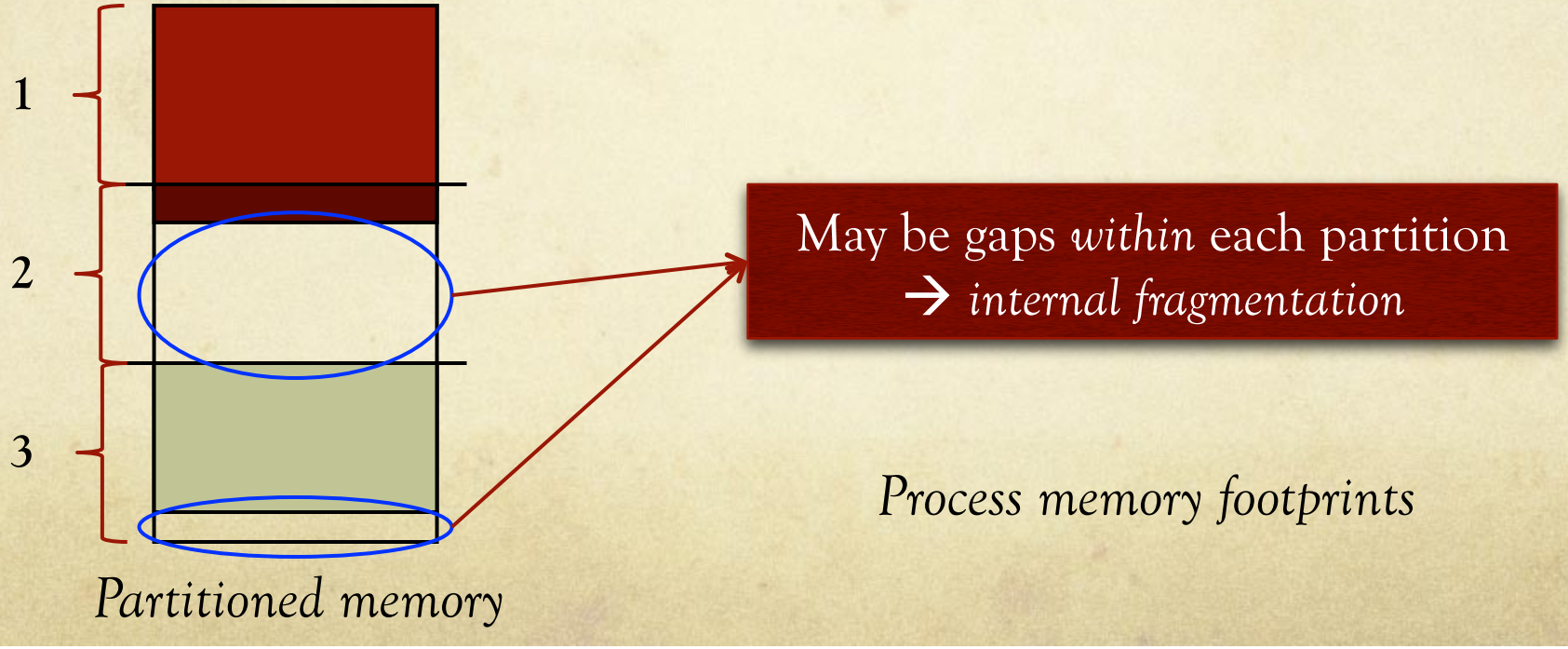
Partitioned memory



Process memory footprints

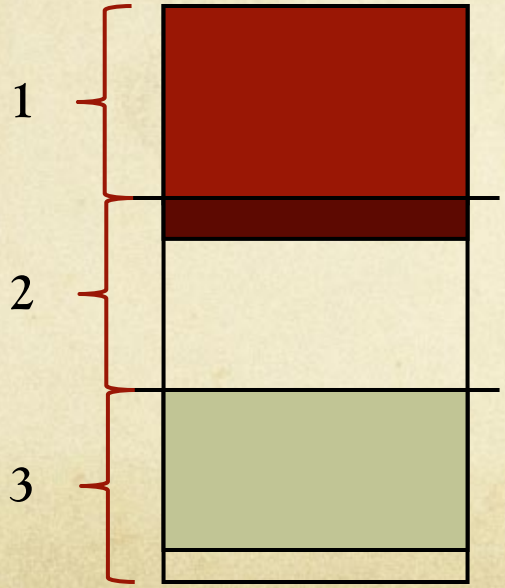
Static, equal-sized partitions

- *Statically* divide main memory into *equal-sized* partitions
- Map *each* process to *one* partition
  - For now, assume mapping does not change



Static, equal-sized partitions

- What if there are more processes than partitions?
  - Map multiple processes to *same* partition
  - *Store* existing process data to disk & *de-allocate* its partition
  - *Allocate* new process to partition & *load* its data from disk

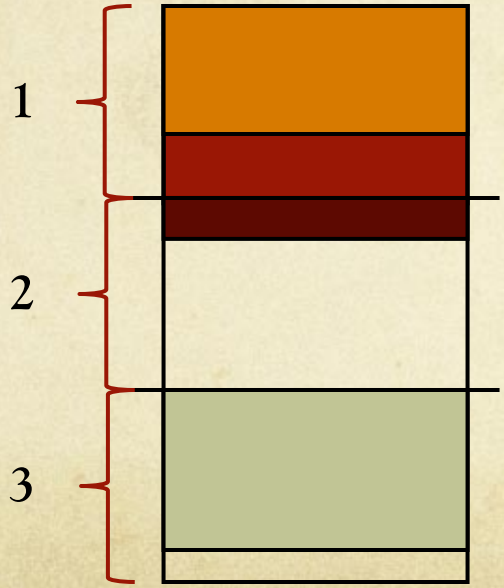


Partitioned memory

Process memory footprints

Static, equal-sized partitions

- What if there are more processes than partitions?
  - Map multiple processes to *same* partition
  - *Store* existing process data to disk & *de-allocate* its partition
  - *Allocate* new process to partition & *load* its data from disk



Partitioned memory

Process memory footprints

Protection

- Ensure process only accesses locations in its partition
- Store start address or *base address* (*BA*) for each process
- Check every address issued by process
  - Must lie between *BA* & (*BA* + *partition size*)

Example

- Memory address range: 0 – 4999; Partitions of size 1000
- Partition to process mapping
  - Partition 1: P1, P6
  - Partition 2: P2, P7
  - Partition 3: P3
  - Partition 4: P4
  - Partition 5: P5
- Is P1 allowed to access address 1004?
  - *No*!
- Is P4 allowed to access address 3000?
  - *Yes*

Discussion

- *Pros*
  - Very simple
- *Cons*
  - Results in *internal* fragmentation
    - Results in *under-utilization* of memory
  - Cannot fit process *larger* than partition size
  - Takes a *one-size-fits-all* kind of approach