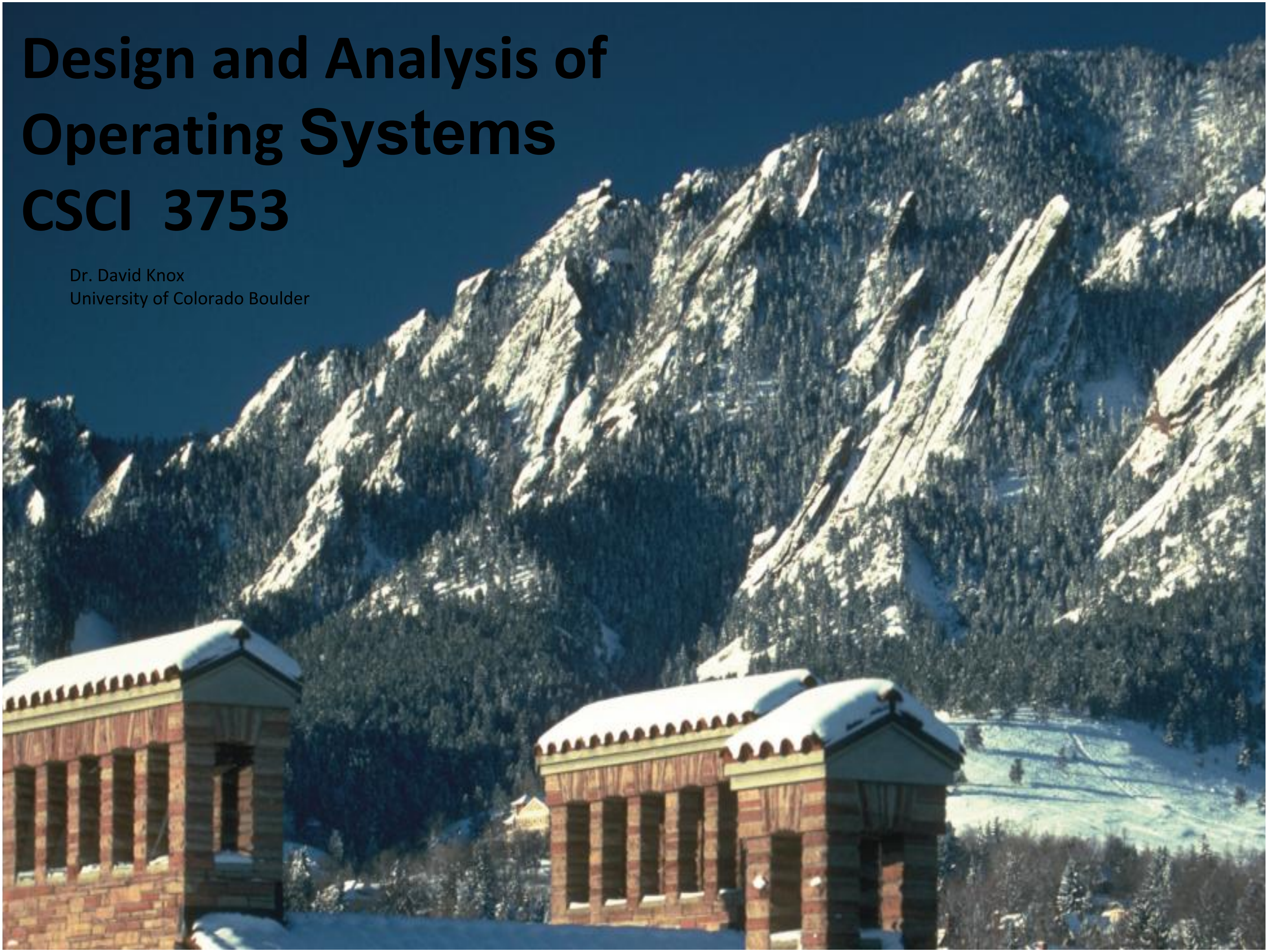
# Design and Analysis of Operating Systems CSCI 3753

Dr. David Knox
University of Colorado Boulder



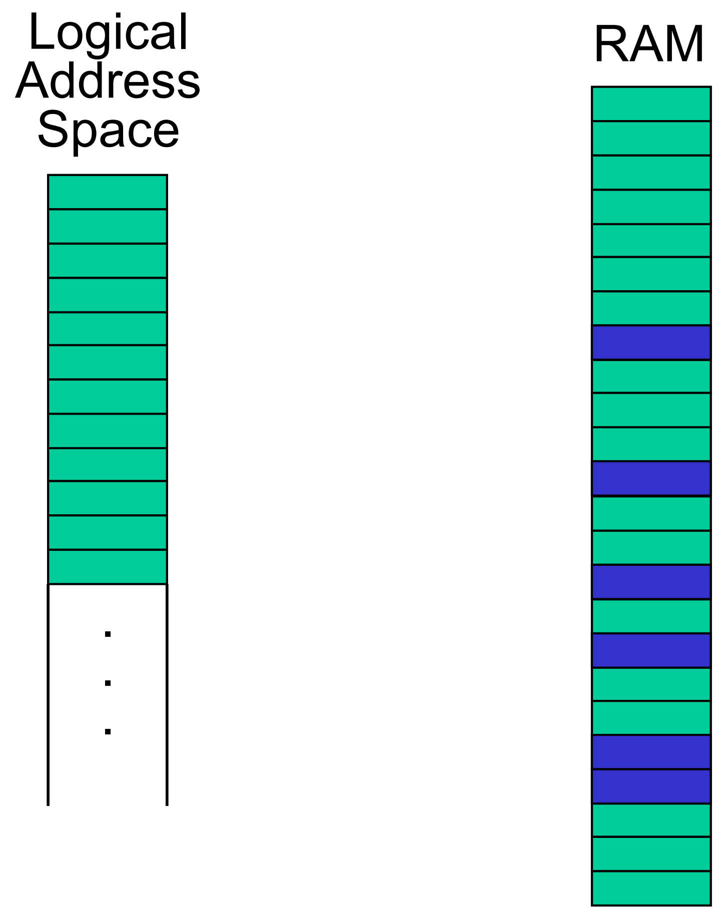
# Design and Analysis of Operating Systems CSCI 3753

## Memory Management Frame Allocation

Dr. David Knox

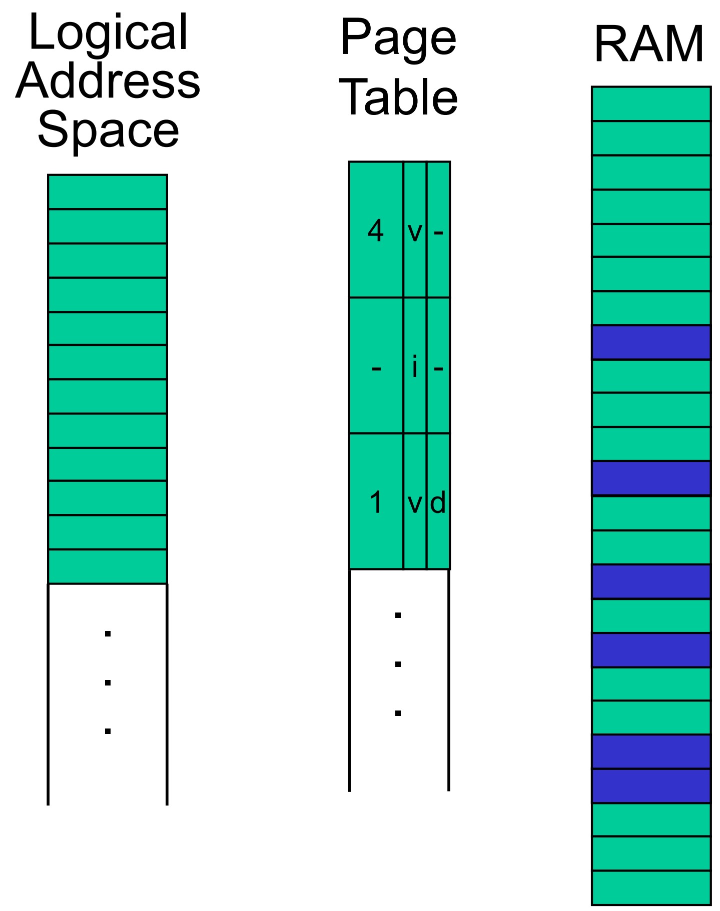
University of Colorado Boulder

# Memory Management
# Frame Allocation



# Techniques to Improve Page Replacement Performance

1. **Use a dirty/modify bit to reduce disk writes**
2. **Choose a smart page replacement algorithm**
   - keeps the most important pages in memory and evicts the least important
3. **Make the search for the least important page be fast**

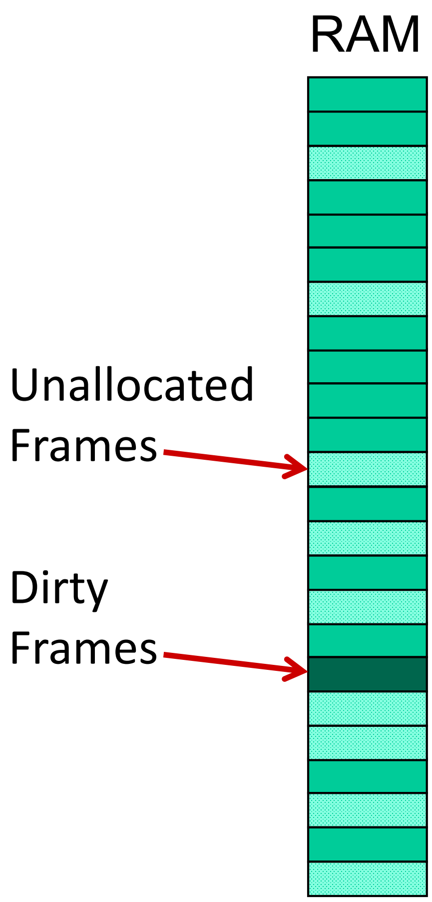


# Techniques to Improve Page Replacement Performance

4. **Page-buffering**
   - read in a frame first and start executing, then select a victim and write it out at a later time - faster perceived performance
   - periodically write out all modified frames to disk when no other activity. Thus most frames are clean so few disk writes on a page fault.

5. **Keep a pool of free frames and remember their content.**
   - Reuse this free frame if the same contents are needed again. Reduces disk reads.

6. **Allocate the appropriate # of frames so a process avoids thrashing – we'll see this next**

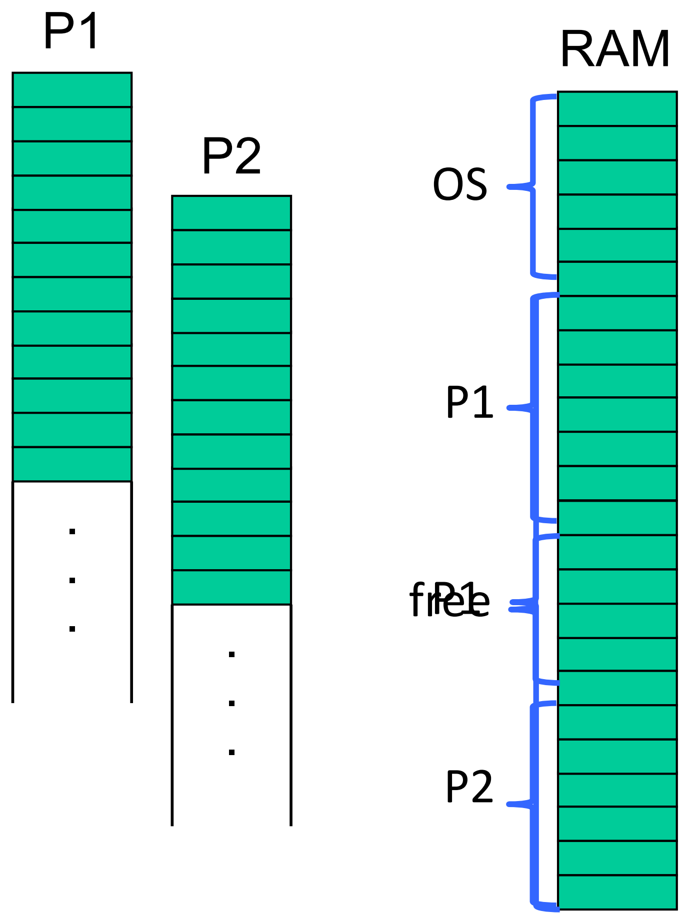


# Allocation of Memory Frames

- **Given that the OS employs paging for memory management, then physical memory is divided into fixed-sized frames or pages.**

- **How many frames does each process get allocated? How many frames does the OS get allocated versus user processes?**
  - Variety of policies:
    - based on number of frames
    - based on whether frames are allocated locally or globally

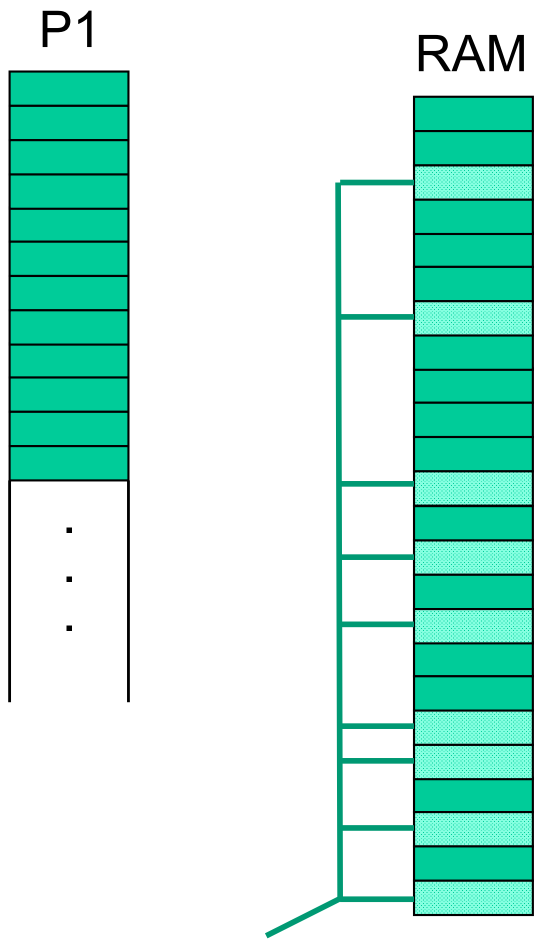
  - Example: given the OS, and processes P1 and P2, and 15 frames of physical memory, how do we allocate the frames among these three entities?



# Memory Allocation Policies

1. **Minimum # frames:**

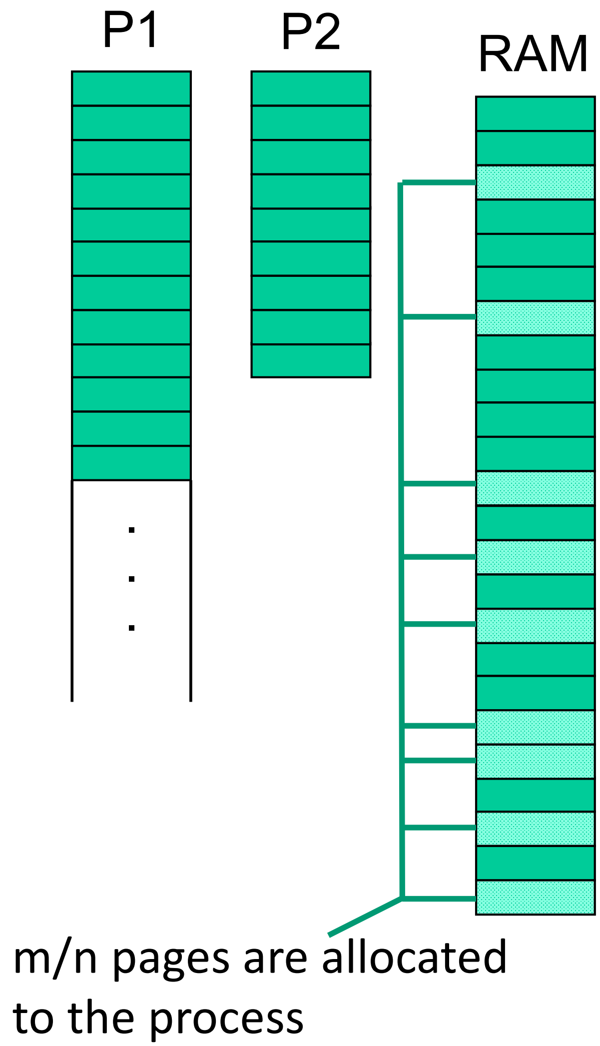
- determine the minimum # of frames that allow a process to allocate. Ideally, this is just one page, i.e. the page in which the program counter is currently executing.
- In practice, some CPU's support complex instructions.
  - multi-address instructions. Each address could belong to a different page.
- Also, there can be multiple levels of indirection in the addressing, i.e. pointers.
  - Each such level of indirection could result in a different page being accessed in order to execute the current instruction. Up to N levels of indirection may be supported, which means may need up to N pages as the minimum.




N pages are needed to be loaded to support process

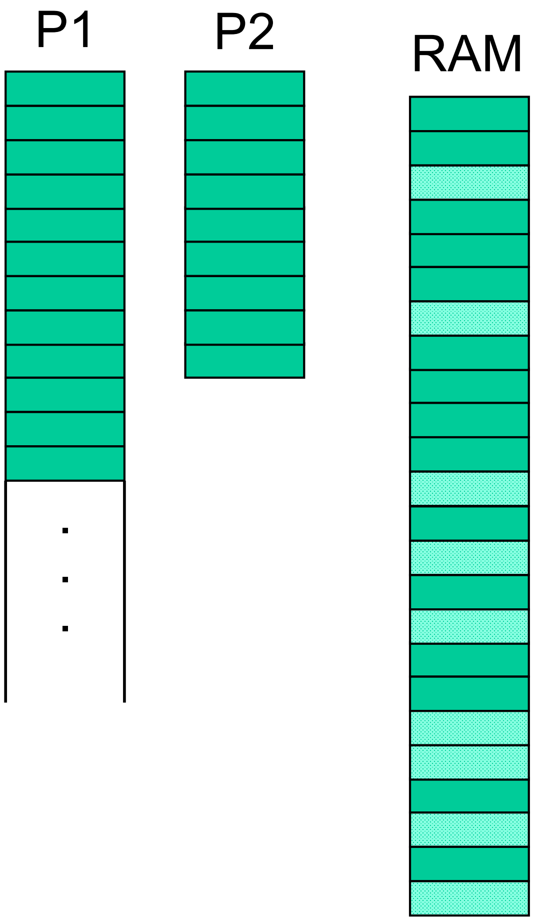
# Memory Allocation Policies

2. **Equal allocation:**
   - split m frames equally among n process
   - m/n frames per process
   - problem: doesn't account for size of processes, e.g. a large database process versus a small client process whose size is << m/n
   - needs to be adaptive as new process enter and the value of n fluctuates



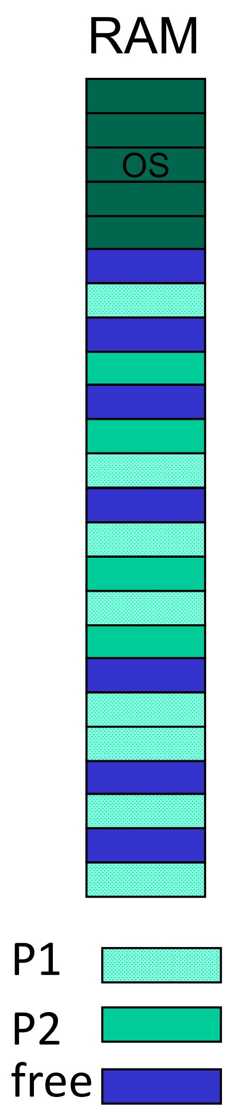
# Memory Allocation Policies

3. **Proportional allocation**
   - allocate the number of frames relative to the size of each process
   - Let $S_i$ = size of process $P_i$
   - $S = \Sigma S_i$
   - Allocate $a_i = (S_i / S) * m$ frames to process $P_i$
   - proportion $a_i$ can vary as new processes start and existing processes finish
   - Also, if size is based on the code size, or address space size, then that is not necessarily the number of pages that will be used by a process



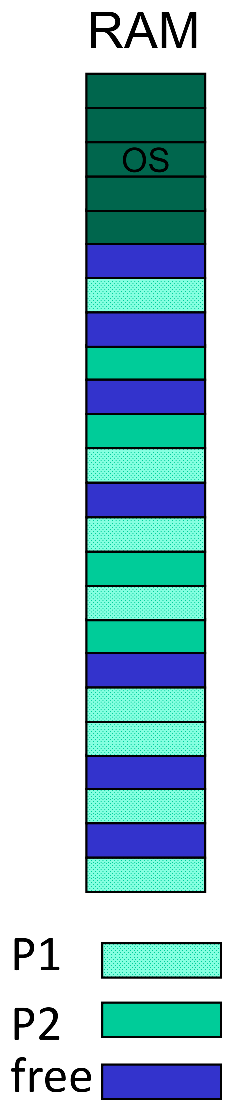
# Local vs Global Allocation

- **In local allocation/page replacement, a process is assigned a fixed set of N memory pages for the lifetime of the process**
  - When a page needs to be replaced, it is chosen only from this set of N pages
  - Easy to manage
  - Processes isolated from each other (not fully true)



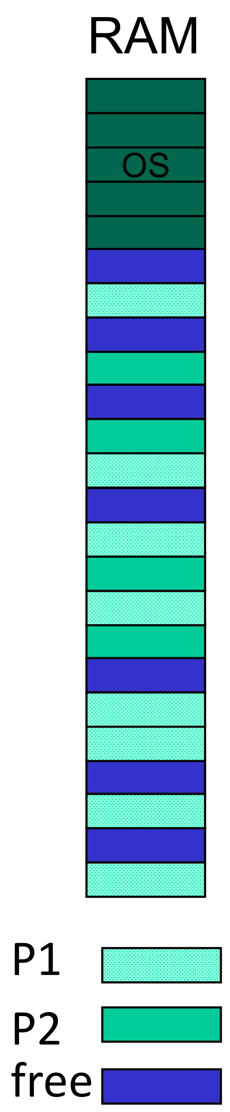
# Local vs Global Allocation

- **Problems with local allocation:**
  1. the behavior of processes may change as they execute
     - Sometimes they'll need more memory, sometimes less
     - local replacement doesn't allow a process to take advantage of unused pages in another process
     - Want a more adaptive allocation strategy that would allow a page fault to trigger the page replacement algorithm to increase its page allocation



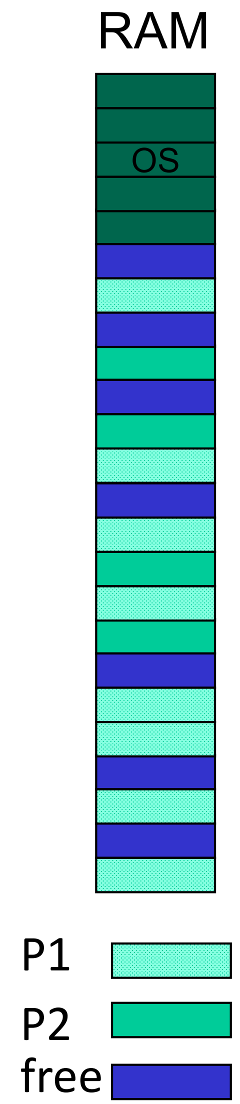
# Local vs Global Allocation

- **Problems with local allocation:**
  2. Local replacement would seem to isolate a process's paging behavior from other processes
     - Amount of memory allocated to each process is fixed
     - However, isolation is not perfect:
     - If a process P1 page faults frequently, then P1 will queue up many requests to read/write from/to disk.
     - If another process P2 needs to access the disk, e.g. to page fault, then even if P2 page faults less frequently, P2 will still be slowed down by P1's many queued up reads and writes to disk



# Local vs Global Allocation

- **Global allocation and page replacement**
  - Pool all pages from all processes together
  - When a page needs to be replaced/evicted, choose it from the global pool of pages
  - Linux follows this model
  - Global allocation and page replacement allows the # pages allocated to a process to fluctuate dramatically
    - Good: system adapts to let a process utilize “unused” pages of another process, leading to better memory utilization overall and better system throughput
    - Bad: a process cannot control its own page fault rate under global replacement, because other processes will take its allocated frames, potentially increasing its own page fault rate



# Thrashing

- **Describes situation of repeated page faulting**
  - this significantly slows down performance of the entire system, and is to be avoided
- **occurs when a process' allocated # of frames < size of its recently accessed set of frames**
  - each page access causes a page fault
  - must replace a page, load a new page from disk
  - but then the next page access also is not in memory, causing another page fault, resulting in a domino effect of page faults - this is called *thrashing*
  - a process spends more time page faulting to disk than executing – is I/O bound, causing a severe performance penalty

# Thrashing

- **Thrashing under local replacement: an example**
  - a single process P1 exhibits poor locality in code and/or data but processes P2 through PN exhibit strong locality in code/data.
  - Suppose P1 is not allocated enough memory frames, but P2 through PN have enough frames.
  - P1 will thrash, but local replacement confines the thrashing to P1. P2 through PN will not thrash.
  - The isolation is not perfect. P1's disk reads and writes will slow down other processes that have disk I/O

# Thrashing

- **Thrashing under global replacement**
  - a process needs more frames, page faults and takes frames away from other processes, which then take frames from others, ... - domino effect
  - as processes queue up for the disk, CPU utilization drops
  - OS can add fuel to the fire:
    - suppose OS has the policy that if it notices CPU utilization dropping, then it thinks that the CPU is free, so it restarts some processes that have been frozen in swap space
    - these new processes page fault more, causing CPU utilization to drop even further

# Thrashing

- **Solution 1: Build a *working set* model.**
  - Programs tend to exhibit *locality* of behavior, i.e. they tend to access/reuse same code/data pages
  - Find a set of recently accessed pages that captures this locality
  - To measure locality, define a window size $\Delta$ of the $\Delta$ most recent page references
    - Within the set of pages in $\Delta$, the working set is the set of *unique* pages
    - pages recently referenced in working set will likely be referenced again
  - Then allocate to a process the size of the working set

# Thrashing

- **working set solution (continued):**
  - Example: Assume Δ = 4 and we have a page reference sequence of

2, 7, 3, 2, 0, 1, 2, 1, 2, 1, 0, 5, ...

t1

t2

working set at t1,
WS1 = {0,2,3,7}
working set size = 4

working set at t2, WS2 = {1,2}
working set size = 2

- at time t1, process only needs to be allocated 4 frames; at time t2, process only needs 2 frames
- Is Δ = 4 the right size window to capture the locality?

# Thrashing

- **working set solution (cont.)**
  - Choose window Δ carefully
    - if Δ is too small, Δ won't capture the locality of a process
    - if Δ is too large, then Δ will capture too many frames that aren't really relevant to the local behavior of the process
      - – which will result in too many frames being allocated to the process

# Thrashing

- **working set solution (cont.)**
  - Once $\Delta$ is selected, here is the working set solution:
    1. Periodically compute a working set and working set size $WSS_i$ for each process $P_i$
    2. Allocate at least $WSS_i$ frames for each process
    3. Let demand $D = \Sigma\ WSS_i$ . If $D > m$ total # of free frames, then there will be thrashing. So swap out a process, and reallocate its frames to other processes.
  - This working set strategy limits thrashing

# Thrashing

- **approximate working set with a timer & a reference bit**
  - set a timer to expire periodically
  - any time a page is accessed, set its reference bit
  - at each expiration of the timer, shift the reference bit into the most significant bit of a record kept with each page,
    - a byte records references in the last 8 timer epochs
  - If any reference bit is set during the last N timer intervals, then the page is considered part of the working set, i.e. if record > 0
  - re-allocate frames based on the newly calculated working sets. Thus, it is not necessary to recalculate the working set on every page reference.

# Thrashing

- **Solution 2: Instead of using a working set model, just directly measure the page fault frequency (PFF)**
  - When PFF > upper threshold, then increase the # frames allocated to this process
  - When PFF < lower threshold, then decrease the # frames allocated to this process (it doesn't need so many frames)

# Linux Global Page Replacement

- **Doesn't explicitly use working sets to avoid thrashing**
- **Instead, applies Clock-like LRU to entire pool of all process' pages**
  - The evicted page is the least recently used globally over all processes, not just one process
    - This should be the best candidate in the entire system to remove, hence globally optimal by minimizing thrashing – won't need it again soon, as would occur for thrashing
  - As a process needs more pages, its effective working set expands and it takes pages from others
    - But the evicted page is the global LRU – least likely to be used again soon by the entire system, hence minimizing thrashing

# Linux Global Page Replacement

- **Hence, Linux's approach:**
  - Effectively and adaptively allocates in an implicit way the working set to each process
- **Thrashing can still occur if the sum of all process' working sets exceeds size of physical memory**
  - In this case, Linux has a mechanism to terminate processes – rarely used

# Design and Analysis of Operating Systems CSCI 3753

Dr. David Knox
University of Colorado Boulder