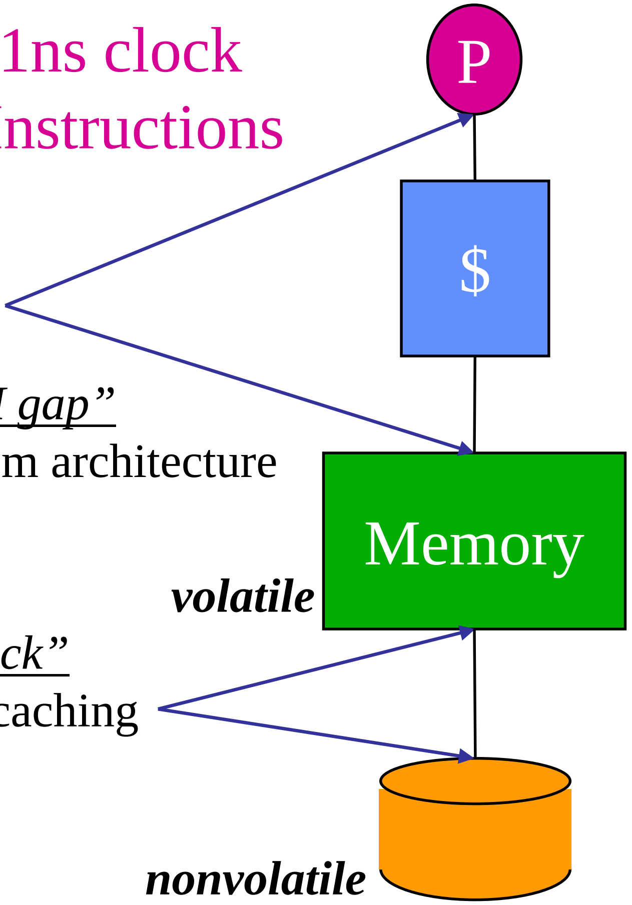
I/O Caching and Page Replacement

Dr. Daniel Andresen

CIS520 – Operating Systems

Memory/Storage Hierarchy 101

Very fast 1ns clock
Multiple Instructions
per cycle



SRAM, Fast, Small
Expensive (cache, registers)

"CPU-DRAM gap"
memory system architecture
(CPS 104)

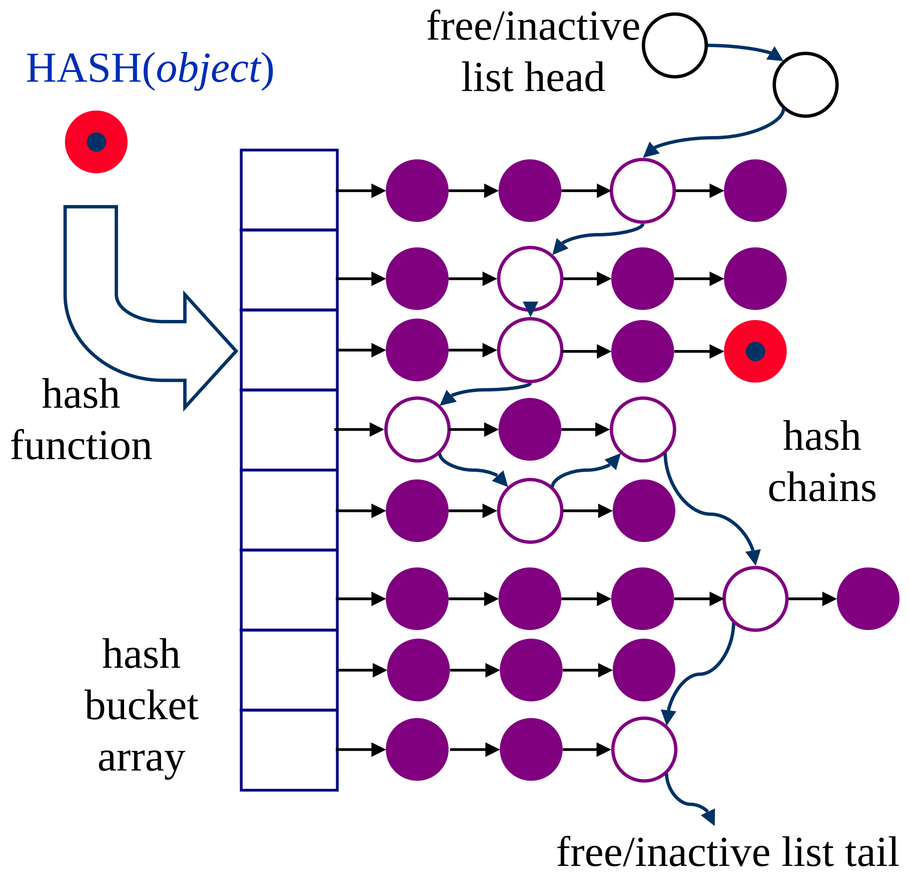
DRAM, Slow, Big,Cheaper
(called *physical* or *main*)
$1000-$2000 per GB or so

"I/O bottleneck"
VM and file caching
(CPS 110)

Magnetic, Rotational,
Really Slow Seeks,
Really Big, Really Cheap
($25 - $40 per GB)

=> Cost Effective Memory System (Price/Performance)

I/O Caching 101



Data items from secondary storage are *cached* in memory for faster access time.

methods:

object = get(tag)

Locate object if in the cache, else find a free slot and bring it into the cache.

release(object)

Release cached object so its slot may be reused for some other object.

I/O cache*: a hash table with an integrated free/inactive list (i.e., an ordered list of eviction candidates).*

Rationale for I/O Cache Structure

Goal: maintain *K slots* in memory as a cache over a collection of *m* items on secondary storage (*K* << *m*).

1. What happens on the first access to each item?

   Fetch it into some slot of the cache, use it, and leave it there to speed up access if it is needed again later.

2. How to determine if an item is resident in the cache?

   Maintain a *directory* of items in the cache: a hash table.

   Hash on a unique identifier (*tag*) for the item (fully associative).

3. How to find a slot for an item fetched into the cache?

   Choose an unused slot, or select an item to replace according to some policy, and *evict* it from the cache, freeing its slot.

Mechanism for Cache Eviction/Replacement

Typical approach: maintain an ordered *free/inactive list* of slots that are candidates for reuse.

- *Busy* items in *active use* are not on the list.

  E.g., some in-memory data structure holds a pointer to the item.

  E.g., an I/O operation is in progress on the item.

- The best candidates are slots that do not contain valid items.

  Initially all slots are free, and they may become free again as items are destroyed (e.g., as files are removed).

- Other slots are listed in order of *value* of the items they contain.

  These slots contain items that are valid but *inactive*: they are held in memory only in the hope that they will be accessed again later.

Replacement Policy

The effectiveness of a cache is determined largely by the policy for ordering slots/items on the free/inactive list.

defines the *replacement policy*

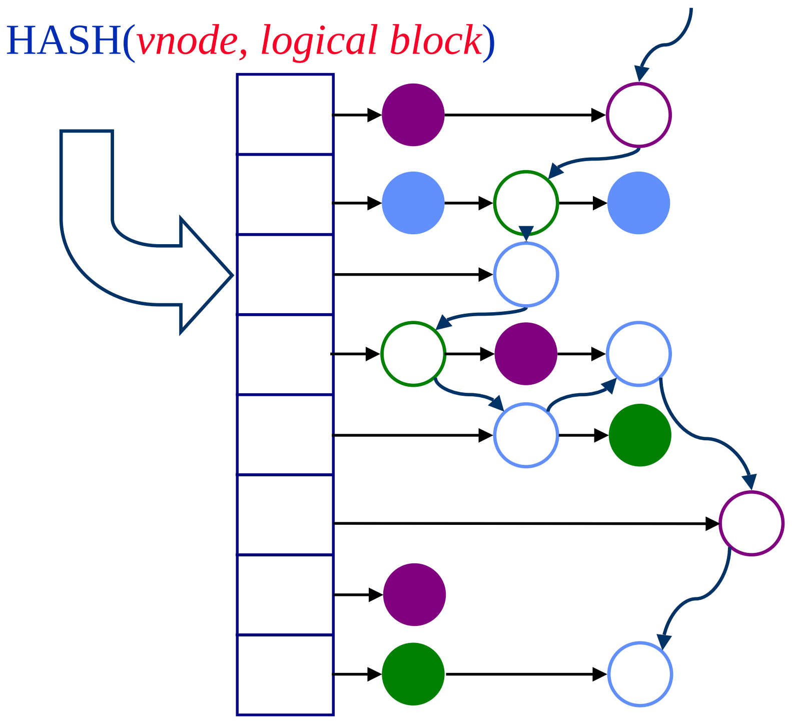
A typical cache replacement policy is ***L**east **R**ecently **U**sed*.

- Assume *hot* items used recently are likely to be used again.
- Move the item to the tail of the free list on every ***release***.
- The item at the front of the list is the *coldest* inactive item.

Other alternatives:

- FIFO: replace the *oldest* item.
- MRU/LIFO: replace the most recently used item.

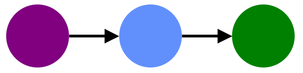
Example: File Block Buffer Cache



Buffers with valid data are retained in memory in a *buffer cache* or *file cache*.

Each item in the cache is a *buffer header* pointing at a buffer .

Blocks from different files may be intermingled in the hash chains.



System data structures hold pointers to buffers only when I/O is pending or imminent.

- *busy bit* instead of refcount
- most buffers are "free"

Most systems use a pool of buffers in kernel memory as a staging area for memory<->disk transfers.

Why Are File Caches Effective?

1. *Locality of reference*: storage accesses come in clumps.
   - *spatial locality*: If a process accesses data in block B, it is likely to reference other nearby data soon.

     (e.g., the remainder of block B)

     example: reading or writing a file one byte at a time
   - *temporal locality*: Recently accessed data is likely to be used again.
2. *Read-ahead*: if we can predict what blocks will be needed soon, we can *prefetch* them into the cache.
   - most files are accessed sequentially

Handling Updates in the File Cache

1. Blocks may be modified in memory once they have been brought into the cache.

   Modified blocks are *dirty* and must (eventually) be written back.

2. Once a block is modified in memory, the write back to disk may not be immediate (*synchronous*).

   - *Delayed writes* absorb many small updates with one disk write.

     How long should the system hold dirty data in memory?

   - *Asynchronous writes* allow overlapping of computation and disk update activity (*write-behind*).

     Do the **write** call for block ***n+1*** while transfer of block ***n*** is in progress.

   - Thus file caches also can improve performance for writes.

The Page Caching Problem

Each thread/process/job utters a stream of page references.

- *reference string:* e.g., *abcabcdabce*..

The OS tries to minimize the number of faults incurred.

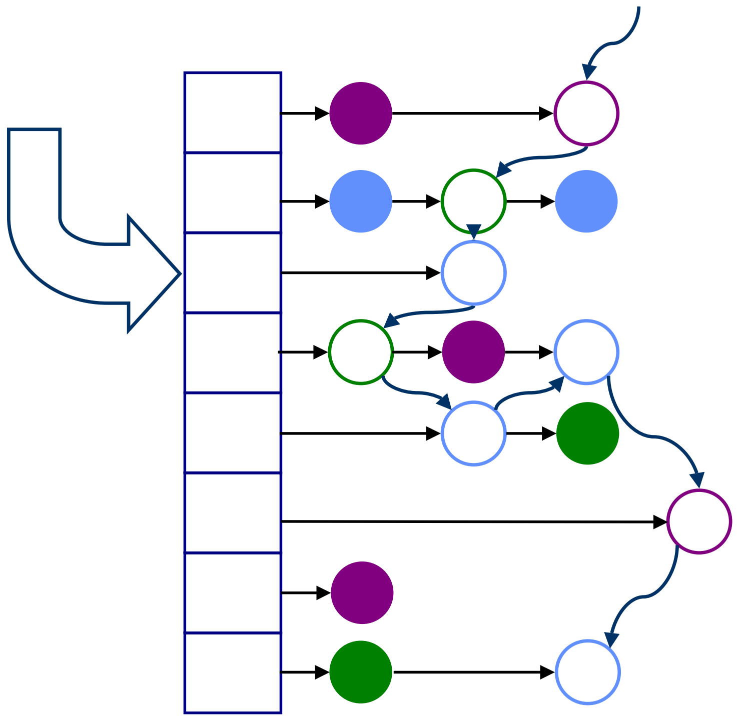
- The set of pages (the *working set*) actively used by each job changes relatively slowly.
- Try to arrange for the *resident set* of pages for each active job to closely approximate its working set.

Replacement policy is the key.

- On each page fault, select a victim page to evict from memory; read the new page into the victim's frame.
- Most systems try to approximate an LRU policy.

VM Page Cache Internals

HASH(*memory object/segment, logical block*)



1. Pages in active use are mapped through the page table of one or more processes.

2. On a fault, the global object/offset hash table in kernel finds pages brought into memory by other processes.

3. Several page queues wind through the set of active frames, keeping track of usage.

4. Pages selected for eviction are removed from all page tables first.

Managing the VM Page Cache

Managing a VM page cache is similar to a file block cache, but with some new twists.

1. Pages are typically referenced by page table (*pmap*) entries.

   Must *pmap_page_protect* to invalidate before reusing the frame.

2. Reads and writes are implicit; the TLB hides them from the OS.

   How can we tell if a page is dirty?

   How can we tell if a page is referenced?

3. Cache manager must run policies periodically, sampling page state.

   Continuously push dirty pages to disk to “launder” them.

   Continuously check references to judge how “hot” each page is.

   Balance accuracy with sampling overhead.

The Paging Daemon

Most OS have one or more system processes responsible for implementing the VM page cache replacement policy.

- A *daemon* is an autonomous system process that periodically performs some housekeeping task.

The *paging daemon* prepares for page eviction before the need arises.

- Wake up when free memory becomes low.
- Clean dirty pages by pushing to backing store.

  prewrite or *pageout*
- Maintain ordered lists of eviction candidates.
- Decide how much memory to allocate to file cache, VM, etc.

LRU Approximations for Paging

Pure LRU and LFU are prohibitively expensive to implement.

- most references are hidden by the TLB
- OS typically sees less than 10% of all references
- can't tweak your ordered page list on every reference

Most systems rely on an *approximation* to LRU for paging.

- periodically *sample* the reference bit on each page

  visit page and set reference bit to zero

  run the process for a while (the *reference window*)

  come back and check the bit again
- reorder the list of eviction candidates based on sampling

FIFO with Second Chance

Idea: do simple FIFO replacement, but give pages a “second chance” to prove their value before they are replaced.

- Every frame is on one of three FIFO lists:

  active, *inactive* and *free*

- Page fault handler installs new pages on tail of active list.
- “Old” pages are moved to the tail of the inactive list.

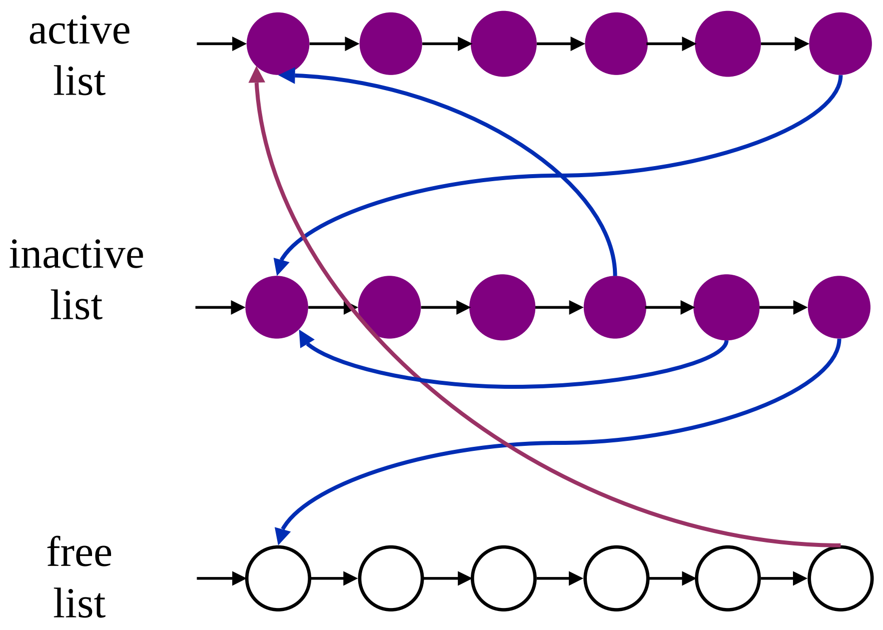
  Paging daemon moves pages from head of active list to tail of inactive list when demands for free frames is high.

  Clear the refbit and protect the inactive page to “monitor” it.

- Pages on the inactive list get a “second chance”.

  If referenced while inactive, *reactivate* to the tail of active list.

Illustrating FIFO-2C



I. Restock inactive list by pulling pages from the head of the active list: clear the ref bit and place on inactive list (*deactivation*).

II. Inactive list scan from head:
1. Page has been referenced? Return to tail of active list (*reactivation*).

2. Page has not been referenced? *pmap_page_protect* and place on tail of free list.

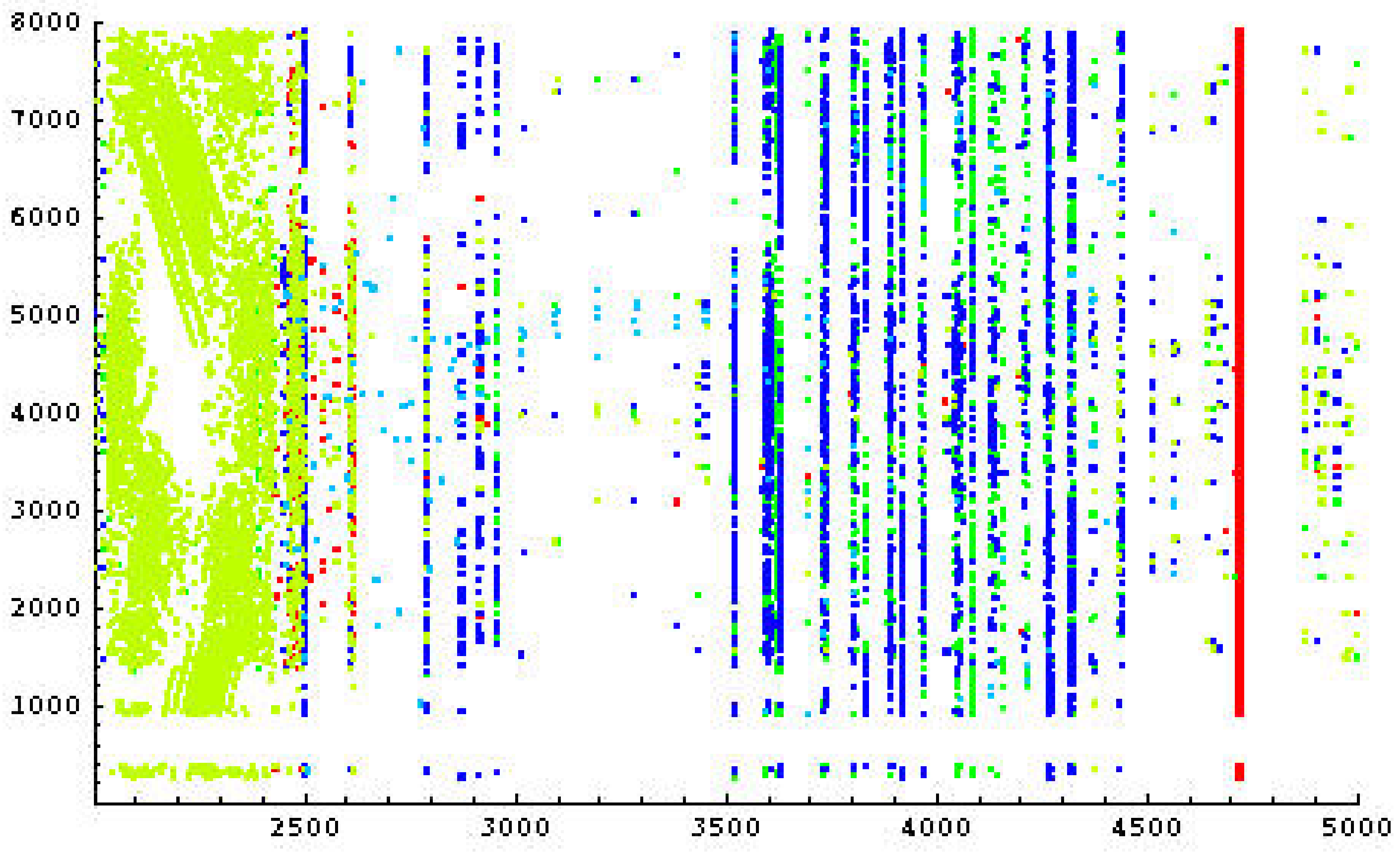
3. Page is dirty? Push to backing store and return it to inactive list tail (*clean*).

Paging daemon typically scans a few times per second, even if not needed to restock free list.

Consume frames from the head of the free list (*free*).

If free shrinks below threshhold, kick the paging daemon to start a scan (I, II).

FIFO-2C in Action (FreeBSD)



What Do the Pretty Colors Mean?

This is a plot of selected internal kernel events during a run of a process that randomly reads/writes its virtual memory.

- **x-axis**: time in milliseconds (total run is about 3 seconds)
- **y-axis**: each event pertains to a physical page frame, whose PFN is given on the y-axis

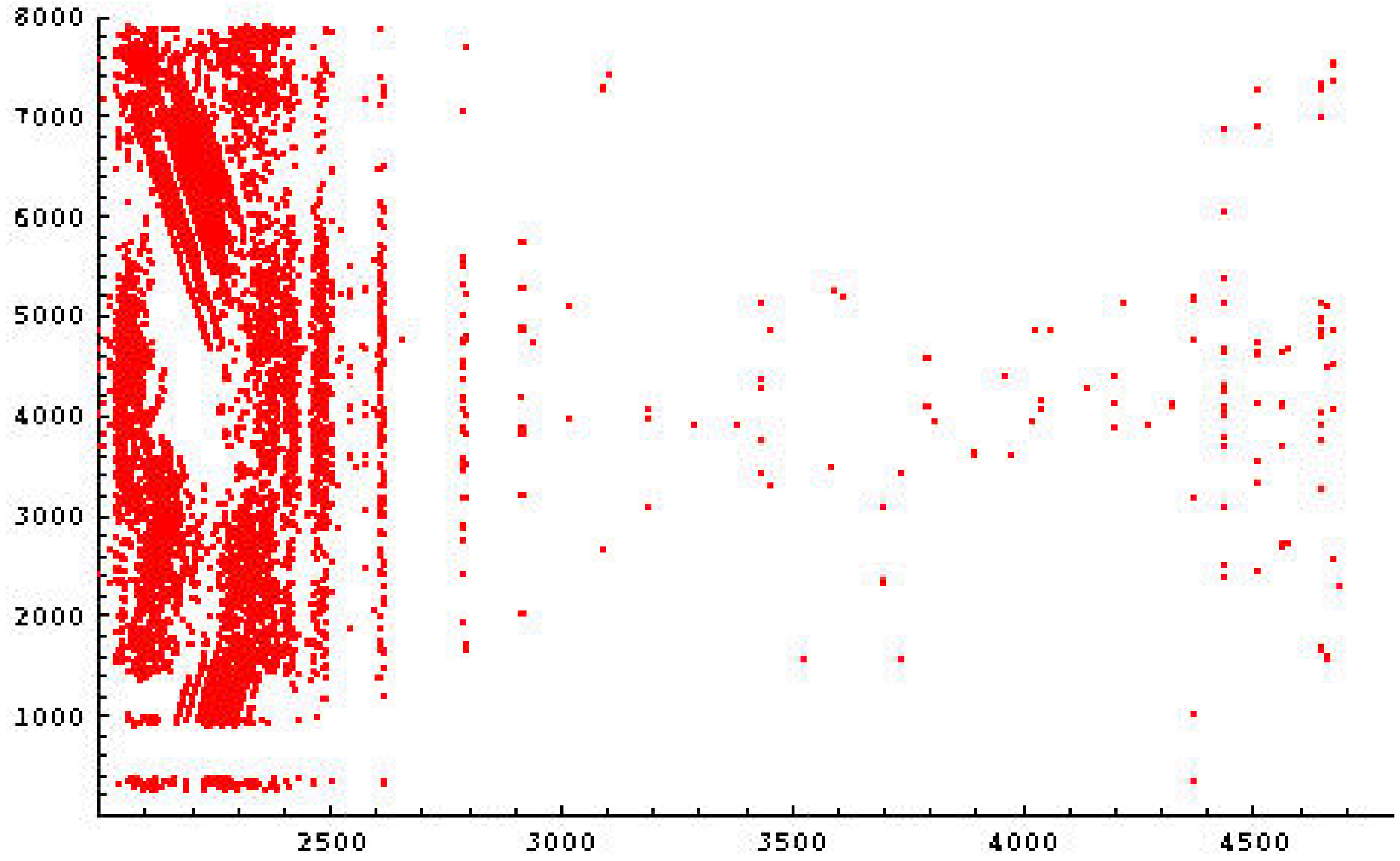
  The machine is an Alpha with 8000 8KB pages (64MB total)

  The process uses 48MB of virtual memory: force the paging daemon to do FIFO-2C bookkeeping, but little actual paging.
- **events**: page *allocate* (yellow-green), page *free* (red), *deactivation* (duke blue), *reactivation* (lime green), page *clean* (carolina blue).

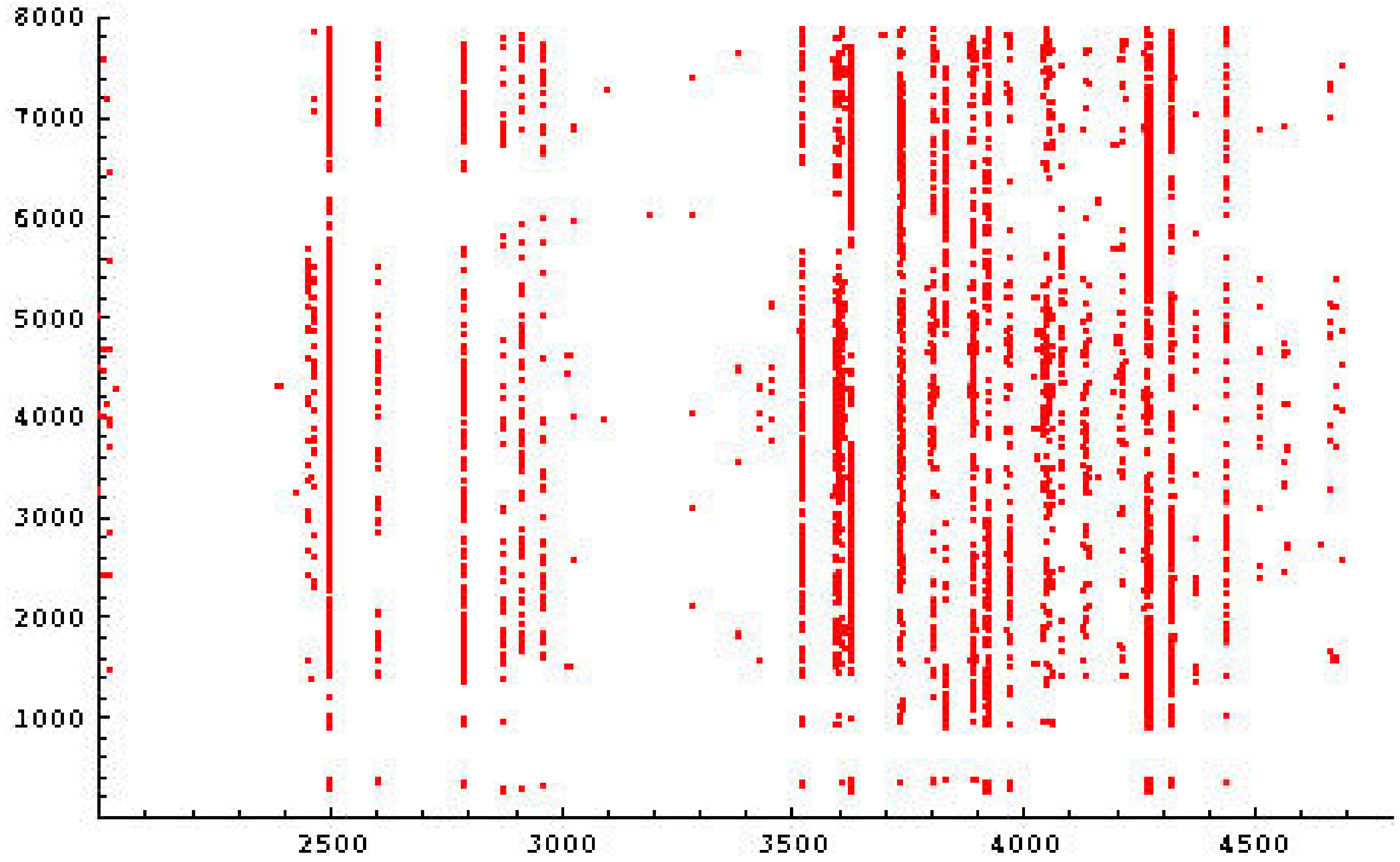
What to Look For

- Some low physical memory ranges are reserved to the kernel.
- Process starts and soaks up memory that was initially free.
- Paging daemon frees pages allocated to other processes, and the system reallocates them to the test process.
- After an initial flurry of demand-loading activity, things settle down after most of the process memory is resident.
- Paging daemon begins to run more frequently as memory becomes overcommitted (dark blue deactivation stripes).
- Test process touches pages deactivated by the paging daemon, causing them to be reactivated.
- Test process exits (heavy red bar).

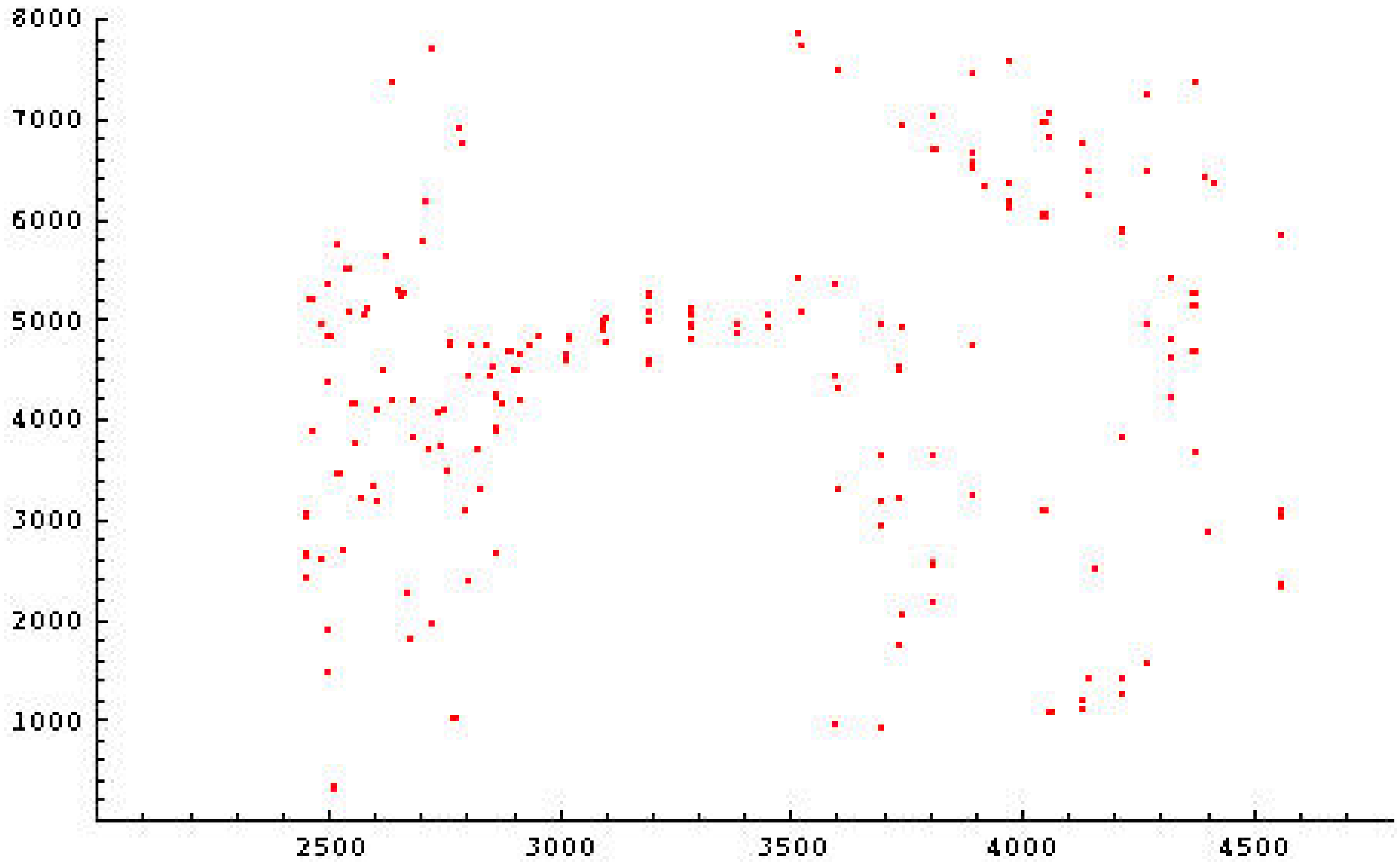
page alloc


8000
7000
6000
5000
4000
3000
2000
1000
2500
3000
3500
4000
4500

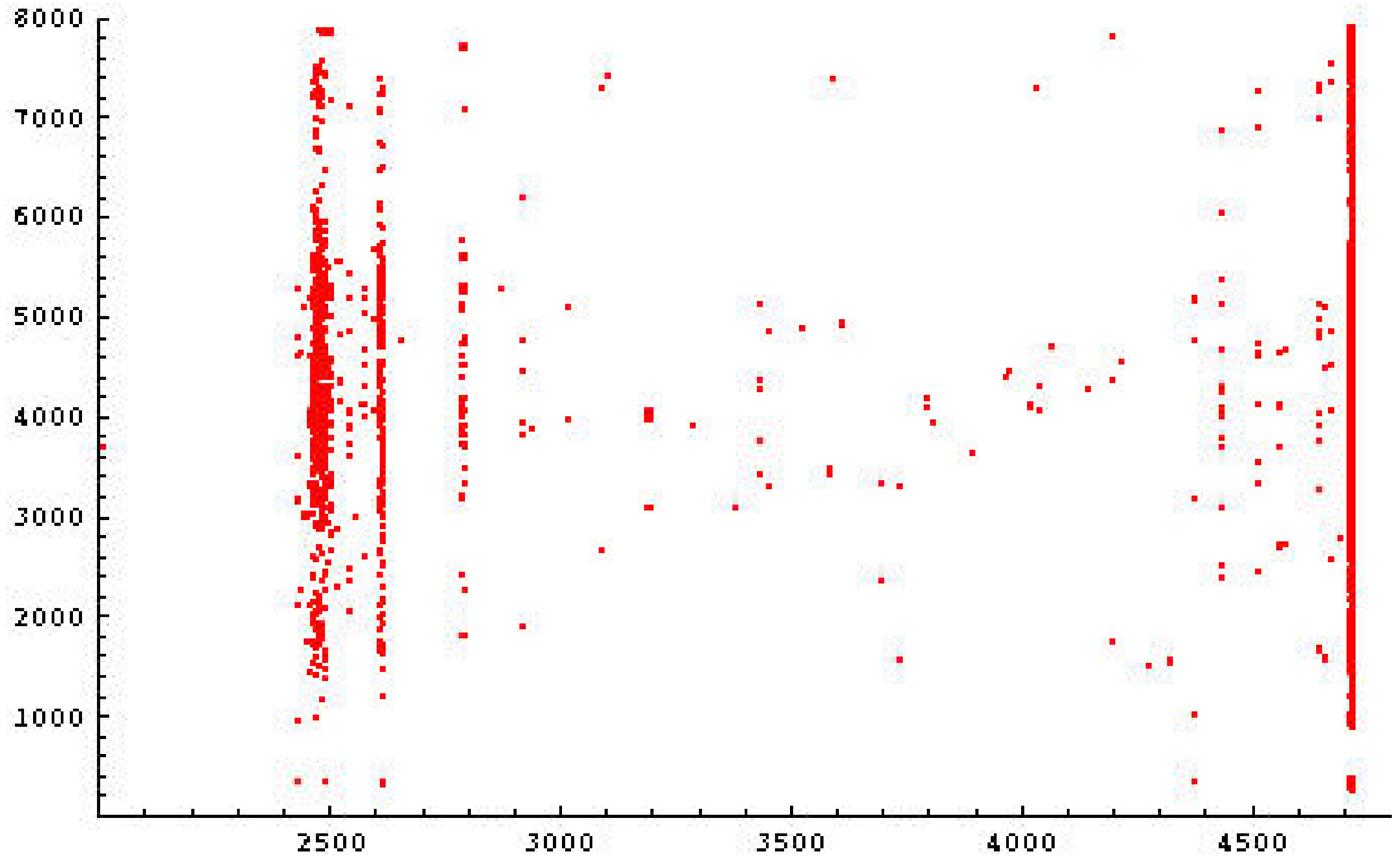
deactivate


8000
7000
6000
5000
4000
3000
2000
1000
2500
3000
3500
4000
4500

clean

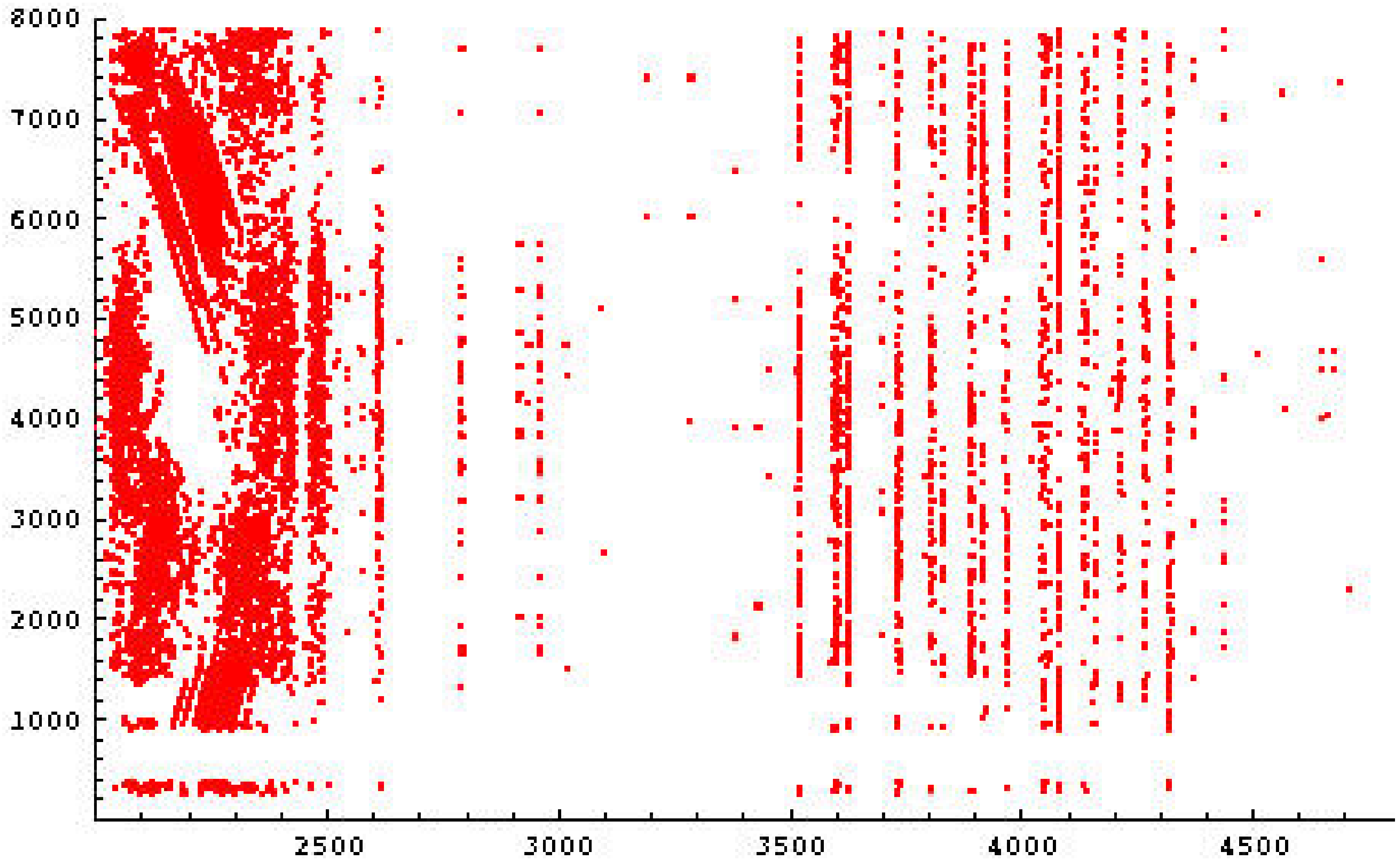

8000
7000
6000
5000
4000
3000
2000
1000
2500
3000
3500
4000
4500

free



8000
7000
6000
5000
4000
3000
2000
1000
2500
3000
3500
4000
4500

activate



8000
7000
6000
5000
4000
3000
2000
1000
2500
3000
3500
4000
4500

Viewing Memory as a Unified I/O Cache

A key role of the I/O system is to manage the page/block cache for performance and reliability.

- tracking cache contents and managing page/block sharing
- choreographing movement to/from external storage
- balancing competing uses of memory

Modern systems attempt to balance memory usage between the VM system and the file cache.

- Grow the file cache for file-intensive workloads.
- Grow the VM page cache for memory-intensive workloads.
- Support a consistent view of files across different style of access.
  - *unified buffer cache*

Pros and Cons of Paged Virtual Memory

Demand paging gives the OS flexibility to manage memory...

- programs may run with pages missing
  - unused or “cold” pages do not consume real memory
  - improves degree of multiprogramming
- program size is not limited by physical memory
  - program size may grow (e.g., stack and heap)

…but VM takes control away from the application.

- With traditional interfaces, the application cannot tell how much memory it has or how much a given reference costs.
- Fetching pages on demand may force the application to incur I/O stalls for many of its references.

More Issues for VM Paging

1. synchronizing shared pages
2. clustered reads/writes from backing store, and prefetching
3. adapting replacement strategies (e.g., switch to MRU)
4. trading off memory between the file (UBC) and VM caches
5. trading off memory usage among processes
6. parameterizing the paging daemon:

   Keep the paging devices fully utilized if pages are to be pushed, but don’t swamp the paging device.

   Balance LRU accuracy with reactivation overhead.

Synchronization Problems for a Cache

1. What if two processes try to *get* the same block concurrently, and the block is not resident?
2. What if a process requests to write block ***A*** while a *put* is already in progress on block ***A***?
3. What if a *get* must replace a dirty block ***A*** in order to allocate a buffer to fetch block ***B***?

   This will happen if the block/buffer at the head of the free list is dirty.

   What if another process requests to *get* ***A*** during the *put*?

4. How to handle read/write requests on shared files atomically?

   Unix guarantees that a *read* will not return the partial result of a concurrent *write*, and that concurrent writes do not interleave.