Clock Extensions

- **Replace multiple pages at once**: expensive to run replacement algorithm and to write single block to disk. Find multiple victims each time.
- **Nth chance**: The clock algorithm is also called the "second-chance" algorithm, as the pages with the accessed bit equal to one, are given a second chance. A straight-forward extension to this is the Nth chance algorithm, wherein a page is given N chances before evicting it. Here is the Nth chance algorithm:
    - With each page, OS maintains a counter to indicate the number of sweeps that page has gone through.
    - On page fault, OS checks accessed bit:
        - If 1, then clear it, and also clear the counter.
        - If 0, then increment the counter; if count == N, replace page.

  Large N implies better approximation to LRU, e.g., N = 1000 is a very good LRU approximation. However, a large N implies more work by the OS before a page can be replaced.

  N = 1 implies the default clock algorithm.

- **Treat dirty pages preferentially**: Dirty pages require more work for eviction, and so given a choice, it is cheaper to replace clean pages. A common approach is to give dirty pages one more chance over clean pages, e.g., N=1 for clean pages, N=2 for dirty pages (and write back to disk in a batched manner when N=1).

LRU analysis

In general, LRU works well, but there are certain workloads where LRU performs very poorly. One well-known workload is a "scan" wherein a large memory region (larger than physical memory) is accessed in quick succession. e.g.,

```
for (i = 0; i < MAX_PHYSICAL_MEM_PAGES + 1; i++) {
  access(page i);
}
```

Consider an example where there are 5 pages in physical memory, and 6 pages of the address are repeatedly scanned:
ABCDEFABCDEFABCDEFA...
What happens with LRU? All accesses are misses. In this case, the page that is replaced happens to be the one that is likeliest to be accessed closest in future. In other words, the future is opposite of the past. Any other algorithm (e.g., Random) would perform better than LRU in this case.

Such scans are often quite common, as some thread may want to go through all its data. And so, plain LRU (or CLOCK) does not work as well in practice. Instead a variant of LRU that considers both recency and frequency of access to a page, is typically used in an OS (e.g., 2Q, CLOCK with Adaptive Replacement, etc.).

A related idea called LRU-K tracks the K-th reference to a page in the past, and replaces the page with oldest K-th reference (or one that does not have K-th reference). This algorithm is resistant to scans, i.e., a scan will not pollute the cache with cold pages. This is expensive to implement for an OS, but LRU-2 is often used in databases.

Global or Local Replacement

Per-process or global replacement?

In global replacement, all pages from all processes are grouped into a single replacement pool. Each process competes with all the other processes for page frames.

In per-process replacement, each process has a separate pool of pages. A page fault in one process can only replace one of that process's frames. This avoids interference with other processes. But, how to decide how many page frames to allocate to each process? Bad decision implies inefficient memory usage.

One of two common approaches used in practice:

1. Use global replacement
2. Use per-process replacement but "slowly" change the number of frames allocated (quota) to each process depending on usage. The rate of change of the quota is much slower than the cache replacement rate, but adjusts for the case where one process is using less memory while the other process is running out of its allocated quota.

Thrashing and Working Sets

Normally, if a thread takes a page fault and must wait for the page to be read from disk, the OS runs a different thread while the I/O is occurring.

But what happens if memory gets overcommitted? Suppose the pages being actively used by multiple processes do not all fit in the physical memory. Each page fault causes one of the active pages to be moved to disk, so another page fault is expected soon. The system will spend most of its time reading and writing pages instead of executing useful instructions.

This is a situation where the demand paging illusion breaks down. The OS wanted to provide the size of disk and the access time of main memory.

However, at this point, it is providing the access time of disk. This situation is called *thrashing*; it was a serious problem in early demand paging systems.

Dealing with thrashing:

- If a single process is too large for memory, there is nothing the OS can do. That process will simply thrash.
- If the problem arises because of the sum of many processes:
    - Figure out how much memory each process needs.
    - Change scheduling priorities to run processes in groups that fit comfortably in memory: must shed load.

Working Sets

- Informally, a working set of a process is the collection of pages that a process is using actively, and which must thus be memory resident for the process to make progress (and avoid thrashing).
- If the sum of all working sets of all runnable threads exceeds the size of memory, then stop running some of the threads for a while.
- Divide processes into two groups: active and inactive:
    - When a process is active, its entire working set must always be in memory: never execute a thread whose working set is not resident.
    - When a process becomes inactive, its working set can migrate to disk
    - Threads from inactive processes are never scheduled for execution.
    - The collection of active processes is called the *balance set*.
    - Gradually move processes in and out of the balance set.
    - As working sets change, the balance set must be adjusted.

How to compute working sets?

- Denning proposed a parameter T; all pages referenced in the last T seconds comprise the working set. Note that recency of access (and not frequency of access) is used to determine the page's value (as in LRU).
- Computation of working set can be done by maintaining an *idle time* with each page.
- Clock algorithm extended to also increment the idle time, each time the hand passes through the page and the page is found "not-accessed". If the page is found accessed, the idle time is reset to zero.
- Pages with idle times less than T are in the working set.

Difficult questions for the working set approach:

- How long should T be? typically tens of seconds to minutes.
- Need to handle changes in working sets, and manage balance set.
- How to account for memory shared between processes? e.g., let processes that share memory be swapped in and out together.

Page Fault Frequency: another approach to prevent thrashing

- Use per-process replacement. Monitor the rate at which page faults occur in each process.
- If the rate gets too high for a process, assume that its memory is overcommitted; increase the size of its memory pool.
- If the rate gets too low, assume that its memory pool can be reduced in size.
- If the sum of all memory pools don't fit in memory, deactivate some processes.

Thrashing was more of a problem with shared computers (e.g., mainframes). With personal computers, users can either buy more memory or manage the balance set by hand (e.g., terminate some processes manually). Also, memory has become cheaper and plentiful, and so thrashing is less of a problem in general today.

Memory-mapped files

Modern operating systems allow file I/O to be performed using the VM system.

- Memory-mapped files (e.g., mmap())
- Programmer's perspective: file lives in memory, access through pointer dereferences
- Advantages: elegant, no overhead for system calls (can be especially significant for fast storage devices), use madvise() for prefetching limit.