Simulation of Page Replacement Algorithms

Felix Wiemann

Contents

1 Abstract

This is a facharbeit about the simulation of page replacement algorithms. This means that it focuses mainly on the *simulation*, not on the page replacement algorithms themselves. Basic knowledge of memory management systems is presumed.

For a good introduction to the topic, I recommend reading [Tanenbaum, 2001], which comprises a chapter about memory management.

2 Page replacement algorithms

This section gives a very short overview of the page replacement algorithms which can be simulated with the scripts and their implementation on real hardware.

2.1 Optimal

The optimal algorithm replaces the page which will not be accessed for the longest time.

To do this, it needs to look into the future, thus it is unimplementable in reality. However, it is quite useful for comparison purposes in simulations, because in a simulated environment, it is possible to predefine the order in which pages are accessed. Using the optimal algorithm, you can see how "real" algorithms perform compared to the optimum.

2.2 Least Recently Used (LRU)

The "least recently used" (LRU) algorithm replaces the page which has not been accessed for the longest time.

It performs very well, but it is difficult to implement. There are several implementations of it:

1. Every time a page is read, it is moved to the head of the list. The next page to replace is the one at the tail of the list. It is very hard to perform this task so fast that it does not slow down the whole system, because memory accesses occur very frequently.

2. We use a bit matrix consisting of $n * n$ bits, where n is the number of physical pages in memory—refer to [Tanenbaum, 2001] for details on this implementation. If there are many pages, the bit matrix grows very big.

3. Every time a page is read, the current time is stored in a variable of this page. The next page to swap out is the one with the lowest time of last access. This is hardly implementable because the timer needs more than

32 bit to be fine-granular enough and finding the page with the lowest access time is very expensive.

2.3 First In First Out (FIFO)

The FIFO (first-in-first-out) algorithm is extremely simple: It removes the page with the earliest creation time.

This can be implemented using a list: New pages are inserted at the head of the list, and the page at the tail is swapped out.

Another implementation is using a ring (usually referred to as "clock"): Every time a page has to be replaced, the page the pointer points at is swapped out and at the same place the new page is swapped in. After this, the pointer moves to the next page.

The FIFO algorithm's performance is rather bad.

2.4 Second Chance

The second-chance algorithm is very similar to FIFO. However, it interferes with the accessing process: Every page has, in addition to its "dirty bit", a "referenced bit" (r-bit). Every time a page is accessed, the r-bit is set.

The replacement process works like FIFO, except that when a page's r-bit is set, instead of replacing it, the r-bit is unset, the page is moved to the list's tail (or the pointer moves to the next page, resp.) and the next page is examined.

Second Chances performs better than FIFO, but it is still far from optimal.

2.5 Aging

The aging algorithm is somewhat tricky: It uses a bit field of w bits for each page in order to track its accessing profile.

Every time a page is read, the *first* (i.e. most significant) bit of the page's bit field is set. Every n instructions all pages' bit fields are right-shifted by one bit.

The next page to replace is the one with the lowest (numerical) value of its bit field. If there are several pages having the same value, an arbitrary page is chosen.

The aging algorithm works very well in many cases, and sometimes even better than LRU, because it looks behind the last access. It furthermore is rather easy to implement, because there are no expensive actions to perform when reading a page. However, finding the page with the lowest bit field value usually takes some time. Thus, it might be necessary to predetermine the next page to be swapped out in background.

2.6 Not Recently Used (NRU)

The NRU (not-recently-used) algorithm uses an r-bit for every page. Every time a page is read, the r-bit is set. Periodically, all r-bits are unset. When a page fault occurs, an arbitrary page with r-bit unset is swapped out.

NRU is actually Aging with a bit field width of 1, and it does not perform very well.

3 Simulation of page replacement algorithms

Now we start with the really interesting part: The *simulation* of page replacement algorithms.

3.1 How to simulate

The scripts to simulate the page replacement algorithms are written in Python. They have been tested with Python-2.2.3, but other versions might work as well.

There are four modules, `mms`, `algorithms`, `access` and `simulate`, and `demo`, a demonstration script.

After reading this section I recommend executing `demo.py` in order to get an idea of how it works in practice.

3.1.1 MMS

The module `mms` contains a generic (abstract) class `MMS`, which represents a memory management system. All page replacement algorithms are classes derived from `MMS`.

`MMS`' constructor takes one parameter, the number of physical pages. (The number of virtual pages is unlimited.)

The method `read` reads a page. It takes one parameter, the virtual page number. If the page is already contained in memory, it does nothing. (Of course, this behavior may be changed by derived classes.) If the page is not contained in memory, the method `pagefault` is called.

If the memory is full, `pagefault` calls `swapout`, which is not defined in `MMS` and has to be implemented by derived classes. After this, `pagefault` calls `create`, which creates a new page in memory. Derived classes might implement additional behavior for `create`.

The method `output` is simply used for outputting the virtual page numbers in a human-readable format. By default, it outputs the next page to swap out last, however this behavior might be undesirable for some non-stack algorithms like Second Chance, which should overwrite `output` for better convenience.

Every time a page is swapped in (or out), `pagefault` increments `swapin` (or `swapout`, resp.), so that you can measure the algorithms' performances.

3.1.2 Algorithms

The module `algorithms` contains from `mms.MMS` derived classes which implement various page replacement algorithms.

The classes `FIFO`, `SecondChance` and `LRU` are self-explanatory.

`Optimal` needs to get a list of virtual page numbers which will we accessed, either as second parameter of the constructor or as assignment to the `alist` attribute. The algorithm will only be optimal if the order of read pages is the same as in `alist`.

`Aging` may get the bit field width and shifting frequency as second and third constructor-parameter or as parameters to the method `configure`. ("Shifting frequency" means the number of read-instructions between the shifts.) Defaults for both are provided.

`NRU` is derived from `Aging` and uses a bit field length of 1, regardless of its configuration.

3.1.3 Access

The module `access` provides helper functions to simulate memory accessing behavior of real programs.

Some general notes for all functions in this module: `mem` is a list of virtual page numbers representing the whole allocated memory. `ws` is a list of virtual page numbers representing a program's working set and may be contained in `mem`. `probab` is a probability in the range of 0 to 1. `rand` is a random number generator, e.g. the module `random` or an instance of `random.Random`.

The function `access(mem, ws, rand, probab)` returns a page. With a probability of `probab`, the returned page is contained in `ws`. Otherwise, it is contained in `mem` but not in `ws`.

`makealist(mem, ws, rand, probab, length)` returns a list of `length` pages to access. Basically, it works like `access`, except that *exactly* `round((1 - probab) * length)` pages are contained in `mem` but not in `ws` and all other pages are contained in `ws`, so that random inaccuracies are avoided.

`wsmake(mem, rand, size)` returns a working set of `size` pages which are all contained in `mem`.

`wsmove(mem, ws, rand)` removes one page from `ws` and adds a different, randomly chosen page from `mem` to `ws`.

`wsextend(mem, ws, rand)` adds a random page from `mem` to `ws`.

`wsshrink(ws, rand)` removes a random page from `ws`.

None of the four `ws`-functions touch `mem`, and, when adding a page to `ws`, they ensure that no page is contained more than once in `ws`.

3.1.4 Simulate

The module `simulate` has only one function, `simulate(mms, alist, skip=0)`, which invokes `mms.read` for all pages in `alist`. It returns the ratio of page faults

to accesses, leaving the first `skip` accesses out of consideration. `skip` defaults to 0.

3.2 What to simulate

This section deals with what exactly to simulate.

3.2.1 Hardware

If the hardware is still being designed, an important point is how to support the algorithm.

E.g., without the ability to perform any action after reading a page, there is not much more left than FIFO.

If the hardware is able to reorder elements of a list at every read-instruction, it is possible to implement LRU.

If there are not many pages, it might be a good idea to choose the bit-matrix implementation of LRU or to supply automatically shifting bit-fields for every page in order to support Aging.

3.2.2 Software

Now we can start creating profiles of the programs which will run on the system. E.g., we could generate access lists by tracking a (real) program's memory access behavior.

Or we could analyse the source of the programs in order to answer questions like these: Does the program have a working set at all? (If it does not, we simply do not need to care of a decent page replacement algorithm.) If it does, how does the working set change over time? Does it have a constant size? Does it move? And so on. Based on these parameters, we can create access lists which approximate the program's memory accessing behavior using the `access` module.

3.2.3 Algorithms

Based on the previously defined hardware environment, we select algorithms which can run in that environment, or we even implement some more complex algorithms specific to our hardware.

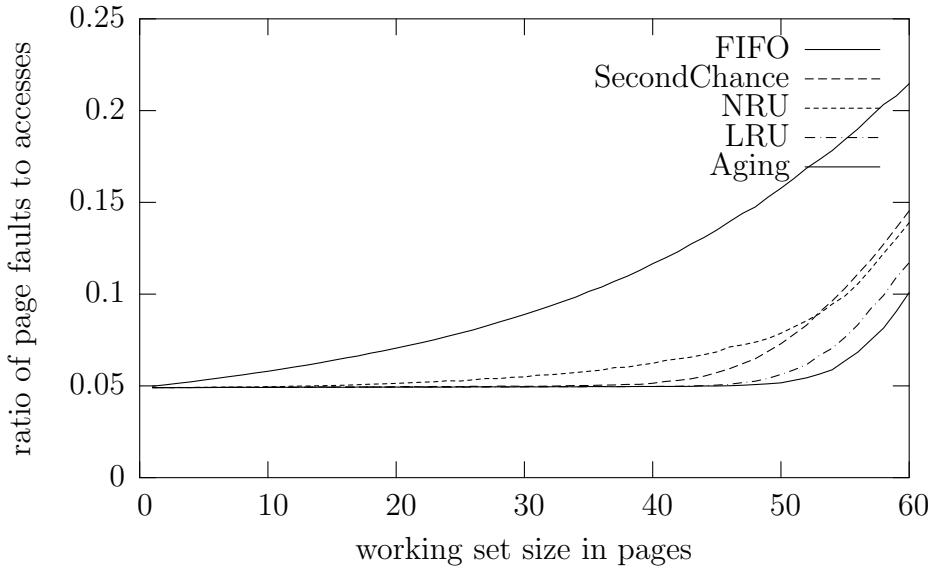
Then we simulate the algorithms using the `simulate` module.

3.2.4 Example 1

The hardware has 64 pages of memory and shall be suitable for applications with a working set size of up to 60 pages. The applications will use the working set at about 95% of their accesses.

First, let us presume that our hardware has unlimited abilities, so that we can compare all algorithms in order to see if there is a cheap possibility to implement a good page replacement algorithm.

We create a diagram. Read the source of `makediagram1.py` on how to exactly do this.



FIFO performs really bad, Second Chance and NRU (both using only the r-bit) perform better and the complex algorithms LRU and Aging are by far the best ones.

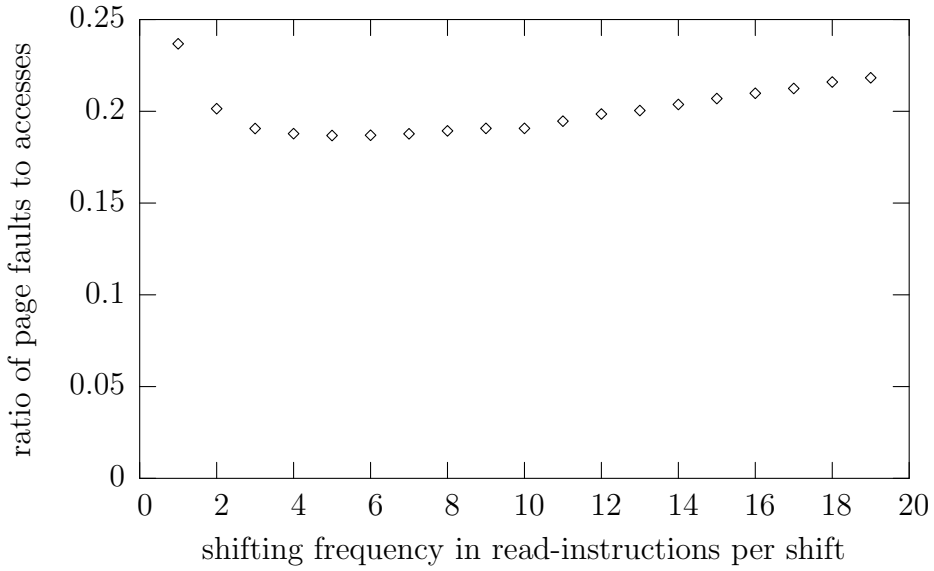
3.2.5 Example 2

Another example:

Aging is a very good algorithm and could be supported by a hardware we are currently designing. Our hardware has a memory size of 6 pages, and the bit fields utilized by the Aging algorithm consist of 4 bits.

The applications which will run on our system will have a working set size of 5 pages and a very large allocated virtual memory.

Now we want to know which shifting frequency is best for this environment. We create a diagram using `makediagram2.py`.



Obviously, the best performance is achieved when shifting every 5 read-instructions.

4 Summary

These scripts help you to find the best page replacement algorithm for a specific environment.

We did not simulate any real existing system environments, as this would mean reinventing the wheel and there are too many of them. These scripts are intended to be used at the designing phase of *new* hardware and operating systems.

5 Bibliography

[Tanenbaum, 2001] Tanenbaum, A. S., Modern Operating Systems, Upper Saddle River 2001[2]