Systems Programming and Operating Systems	**Induprakas Kodukula**
CS414, Summer 1996	**Class#17, 8/5/96**

1 Class #17

This section examines several policy issues that are implicit in the previous discussion regarding *mapping-based* memory management schemes. In particular, it was mentioned that when a virtual address was generated by the processor, it is not necessary that the physical address corresponding to it be in memory at the time. This leads to the question of how such a problem may be dealt with: should an item on disk be brought into disk only when referenced (i.e. on demand), or its need should be anticipated. In addition, there is also the question of which item to remove from memory if there is no space in memory to hold the new item being brought into memory. For the following discussion, it is assumed that the memory is organized in pages, though this does not take anything away from the generalized treatment.

1.1 Page Replacement Algorithms

The first order of business is to characterize the program behaviour. This can be characterized by a *reference string*, which is simply a sequence of page numbers that the program references as it executes. A page fault occurs when a particular entry in the string is *referenced* and the page does not occur in memory. A memory paging algorithm is one that decides which pages should reside in memory and which pages should not. These algorithms can be divided into two classes:

Unrealizable : A memory management algorithm is unrealizeable if it requires knowledge of the future to decide which pages should be kept in memory.

Realizeable : A memory management algorithm is realizeable if it does not need knowledge of the future for its decisions.

Given a particular reference string, it is possible to come up with an **optimial** algorithm which can at any stage look at the reference string and decide which pages to keep and replace when a new page has to be brought into memory. However, such an algorithm is *unrealizeable* for obvious reasons. However, it is a useful comparison point with respect to *realizeable* algorithms.

Some *realizeable* algorithms are:

random Randomly choose a place to be replaced

FIFO Page to be replaced is the one that was brought into memory the earliest

LRU The page to be replaced is the one that was *least recently used*

Which paging algorithm is actually used by the operating system depends on two factors:

Cost: Simulations have suggested that *FIFO* is upto three times worse and *LRU* is upto two times worse than the *optimal* paging strategy in the number of page faults. *Random*'s performance is actually quite terrible, since in practice, programs exhibit quite a bit of *locality*. For the same reason, *FIFO* performs worse than *LRU*. Only when straight-line code with no loops is used that *FIFO* results in performance comparable to *LRU*

Ease of Implementation: The *random* method of page replacement is trivial to implement and requires no hardware support whatsoever. *FIFO* schemes are also quite easy to implement. A queue of page names has to be maintained. When a new page is fetched, it is added at the end of the queue. When a page fault occurs, the page at the head of the queue is deleted, and the new page is fetched and added at the end of the queue. *LRU* is somewhat harder to implement and requires some amount of hardware support. In particular, some way has to be provided to determine the age of any given page (from the time it was last accessed). There are two ways this is usually done:

> *exponential decay*: A capacitor is charged every time a page is touched. At any future time, the capacitor holds a certain amount of charge which can be converted into a time scale.
>
> *aging register*: In this approach, every page has a hardware register associated with it, called an *aging register*. An aging register is simply a shift register. The leftmost bit of the shift register is set to 1 every time the page is touched. Every certain number of clock cycles, the register is shifted right. Eventually, if the page is not touched for a long time, the contents of the register will become zero. A page can be replaced if its aging register contains the value zero. It should be noted that this approach is not a perfect-LRU scheme, rather it is a good approximation.

1.1.1 Other Approximations to LRU

Another approximation to the LRU mechanism is known as the *clock replacement algorithm*. Every page is associated with a single bit, called a *use-bit*. The *use-bit* is set every time the page is referenced. At the same time, a circular list of page frames is maintained along with a pointer into this circular list. The following algorithm summarizes how this approach works:

```
function Replace(int  ptr)
  Let  ptr be the pointer into the circular list
  if (use_bit[ ptr] = 0)
    replace page at  ptr
  else
    use_bit[ ptr] = 0;
    ptr = ptr ⊕ 1
    Replace( ptr)
  endif
```

The idea behind setting use_bit[*ptr*] to 0 is that each page is examined at most once without getting replaced.

1.2 Requirements of Paging Algorithms

A paging algorithm should satisfy the monotonicity property: If the number of pages increases, the number of page faults should reduce. While it may seem obvious that this may always happen, an example will clarify the point:

Let a reference string for a program be the set $\{1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5\}$. With three page frames and a FIFO replacement algorithm, it is easy to verify that this sequence of page accesses leads to the state of page frames in Figure 2. All the circled entries in the first row correspond to page faults, and it can be seen that there are 9 page faults when there are 3 page frames. However,

1	2	3	4	1	2	5	1	2	3	4	5
1	2	3	4	1	2	5	5	5	3	4	4
	1	2	3	4	1	2	2	2	5	3	3
		1	2	3	4	1	1	1	2	5	5

Figure 1: Page faults with 3 frames, FIFO Policy

1	2	3	4	1	2	5	1	2	3	4	5
1	2	3	4	4	4	5	1	2	3	4	5
	1	2	3	3	3	4	5	1	2	3	4
		1	2	2	2	3	4	5	1	2	3
			1	1	1	2	3	4	5	1	2

Figure 2: Page faults with 4 frames, FIFO policy

as can be seen from Figure **??**, the number of page faults actually *increases* tp 10 if the number of page frames increases by 1. This is is known as the **FIFO stack anomaly** problem.

The problem exhibited by the FIFO algorithm is not a desirable property for a paging algorithm, since it is highly undesirable for the performance of a system to worsen because of increased resources. This notion is formalized by the **stack property** that a paging algorithm should have. This property can be defined as follows:

Let

A be a page replacement algorithm

m be the number of page frames

w be a reference string for this program

S(A,m,w) be the pages in page frames when algorithm A is run with a memory of m pages with a reference string w.

A is said to obey the **stack property** iff $\forall w, S(A,m,w) \subseteq S(A,m+1,w)$.

Theorem 1 *LRU is a* **stack algorithm**

Proof: *S(LRU, m, w)* contains the m most recently used pages. *S(LRU, m+1, w)* contains the m+1 most recently used pages. So, the stack property is trivially satisfied. □

Fact 1 *FIFO does not obey the* **stack property**

1.3 Locality and Working Sets

Program behavious often exhibits locality. In other words, pages referenced in the recent past are a good prediction of pages about to be referenced in the near future. There are two main reasons for locality:

context: At any time, the program is usually in a given module, and instructions and data are fetched from the same locations.

looping: Computationally large programs loop, which is to say they execute the same portion of the code repeatedly, accessing the same data repeatedly, as well. (on a RISC, as many as 10% of the total number of instructions can be branch instructions). As a result, loop code is often accessed repeatedly.

The above considerations give rise to the notion of a **working set**. Loosely, the working set of a program is defined as the set of pages a program is referencing because of locality: $\mathfrak{W}(\mathfrak{w}, \mathfrak{k}, \mathfrak{h})$ is defined for a reference string w as the set of pages that appear at position k thru $k - h + 1$. Informally, at any point in the reference string k, for a given window size h, the working set of a reference string w, $\mathfrak{W}$ is all the pages that have been looked at in the immediate past from position k with a window size of h.

The *working set principle* is the following:

$$\mathfrak{W}(\mathfrak{w}, \mathfrak{k}, \mathfrak{h}) = \mathfrak{W}(\mathfrak{w}, \mathfrak{k} + \mathfrak{1}, \mathfrak{h})$$

i.e. the working set of a program for a given window size changes slowly. This principle can be used as a useful paging strategy:

1. Run a process only if its working set is in the page frames (i.e. do enough prefetching of pages if necessary). This will prevent a lot of page faults when a process just starts up.

2. Never run a program wuth fewer page frames than needed to accomodate its working set.

Paging algorithms can also be classified as **global** or **local**

A page-replacement algorithm is **global** if page allocation is applied to entire memory.

A page-replacement algorithm is **local** if memory is partitioned into various regions (possibly per-process regions) and the effect of a replacement algorithm is limited to regions: a page is deleted from the region that a new page is being brought into.

Finally, **thrashing** is defined as a collapse of performance, where all the processor is doing is paging rather than computing. It is easy to see that with a **global** page replacement algorithm, thrashing is a distinct possibility, where as with a **local** page replacement algorithm, thrashing happens only if the local partition is not big enough to hold the working set of the particular process.