PROJECT 4

Page Replacement Algorithms

1 PROJECT OVERVIEW

In this project, we will implement different page replacement algorithms, both local and global. We will compare their relative performance using probabilistically generated reference strings.

2 GLOBAL PAGE REPLACEMENT ALGORITHMS

Global page replacement algorithms assume a fixed number of page frames. We consider a single-process system and make the following assumptions:

- The virtual memory of the process consists of \( P \) pages, numbered 0 through \( P - 1 \).
- A reference string \( RS \) is a sequence of integers ranging from 0 to \( P - 1 \). Each element, \( p \), of \( RS \) represents one reference to page \( p \).
- The main memory consists of \( F \) page frames, numbered 0 through \( F - 1 \). This is represented as an array, \( M[F] \). Each entry \( M[f] \) contains the number \( p \) of the page currently residing in the frame \( f \).

Section 8.3.1 (Chapter 8) describes several different global page-replacement algorithms. Each algorithm sequentially reads the elements of \( RS \). For each element \( p \) of \( RS \) the algorithm searches the main memory array for a match, i.e., it tries to find a \( f \) such that \( M[f] = p \). If no match is found, a page fault occurs. The algorithm must select a page frame \( M[i] \) according to the policy it implements and replace the contents of that frame with \( p \), i.e., \( M[i] = p \).

Depending on the algorithm, additional data structures must be provided to implement it. The following list summarizes the needs of the different algorithms:

- \( MIN \) and random replacement require no additional data structures. In the case of \( MIN \), the algorithm searches \( RS \) to find the page to replace. Random replacement must generate a random number between 0 and \( F - 1 \) to choose the page to replace.
Section 3 Local Page Replacement Algorithms

- **FIFO** only maintains a pointer (array index) pointing at the oldest page in memory. The pointer is incremented by one (modulo \( F \)) whenever the current page is replaced.

- **LRU** must maintain an additional array of size \( F \), which implements the queue; this is reordered at each reference such that the referenced page is placed at the end of the queue.

- The **second-chance** algorithm must maintain a pointer similar to the FIFO algorithm. In addition, it needs an array of \( u \)-bits, one for each frame.

- The **third-chance** algorithm must maintain a pointer and three arrays; these represent the \( u \)-bits, the \( w \)-bits, and the marker-bits (the latter correspond to the asterisk in Table 8-1).

Once the algorithms are developed, they can be tested with various reference strings as described in Section 4.

3 LOCAL PAGE REPLACEMENT ALGORITHMS

With local page replacement algorithms (Section 8.3.2, Chapter 8), the number of page frames is not fixed. Instead, we must keep track of the current working set of the process.

We consider a single process and represent its virtual memory as an array \( VM[\tau] \), where \( \tau \) is the number of pages in the virtual memory. The information recorded as part of each element of \( VM \) depends on the algorithm:

- For the **working set** algorithm, \( WS \), each element \( VM[p] \) only records whether the page \( p \) is currently resident, i.e., a member of the working set. To keep track of the current sliding window, we implement another array, say \( WIN[\tau + 1] \), where \( \tau \) is a system constant. This array serves as a queue of constant length; at any point in time, it contains the pages referenced during the last \( \tau + 1 \) time steps. The \( WS \) algorithm uses the two arrays as follows. For each reference \( p \) in \( RS \), it inserts \( p \) at the head of the queue \( WIN \), and it removes the element, \( q \), currently at the end of the queue. It then searches \( WIN \) for the occurrence of \( q \); if this does not appear, the corresponding entry \( VM[q] \) is set to zero, marking it as not resident. Next, the entry \( VM[p] \) is checked; if this is not 1, the algorithm sets it to 1 and records a page fault.

- For the optimal page-replacement algorithm, \( VMIN \), we use the same arrays \( VM \) and \( WIN \). \( VM \) again records which pages are resident; \( WIN \) corresponds to the forward-looking window, i.e., it always contains the \( \tau + 1 \) pages that will be referenced next. At each reference, the page that will be referenced \( \tau \) steps in the future replaces the currently referenced page \( p \) in \( WIN \). If \( p \) is not resident \((VM[p] == 0)\) at the time of the reference, \( VM[p] \) is set to 1, and a page fault is recorded. In addition, all currently resident pages that do not appear in \( WIN \) are marked as nonresident in \( VM \).

- For the **page fault frequency** algorithm, \( PFF \), we extend the array \( VM \) such that each entry is a structure consisting of two fields. One of the fields, say \( VM[p].res \), records whether page \( p \) is currently resident. The other field, say \( VM[p].u \),
corresponds to the use-bit of a page frame. The two fields are used as follows by the PFF algorithm. \( VM[p].u \) is set to 1 at each reference to page \( p \). At the time of a page fault, \( VM[p].res \) of the referenced page \( p \) is set to 1 (making it a member of the resident set). Then, if the time between the current page fault and the last page fault (which the algorithm also must keep track of) is greater than the constant \( \tau \), the system finds all pages \( q \) with \( VM[q].res == 1 \) and \( VM[q].u == 0 \); these pages have not been referenced since the last page fault and are removed from the resident set (\( VM[q].res \) is set to 0). Finally, \( VM[*].u \) of all pages is set to 0; this is done at the end of every page fault, regardless of whether pages have been removed or not.

The three algorithms can be compared with each other and with global page replacement algorithms as described in Section 5.

4 GENERATING REFERENCE STRINGS

Recall that a reference string is a sequence of integers ranging from 0 to \( P - 1 \), where \( P \) is the total number of pages constituting the virtual memory of the process. We can generate reference strings with different properties using random number generators. The main property of a reference string is its degree of locality, which we can control in several different ways.

The simplest possible reference string is one that has no locality. This is a sequence of uniformly distributed random numbers in the range 0 through \( P - 1 \). Such a string is representative of an oversaturated system, where too many processes compete for the CPU; there is no locality of reference—the system is thrashing.

Most programs display a high degree of locality. That means, the same set of pages—the current working set—is being referenced repeatedly. We define the current locus of reference as a region in the virtual memory (not necessarily contiguous) from which addresses are currently being generated by the process.

A typical program behavior consists of periods during which the working set size is stable, punctuated by rapid transitions as the process changes its locus of reference. During each transition, the working set expands rapidly as pages in the new locus are being accessed. After a while, the working set shrinks to a new stable size as the pages of the previous locus are discarded from the working set.

We can model this behavior as follows. First, we assume that the current locus of reference consists of neighboring pages in the virtual memory. This is not the case in a real program, since pages from at least three regions (code, data, and stack) are being accessed. For studying page-replacement algorithms, however, we are interested only in the size of the working set and the frequency with which new pages are included and old ones are discarded. Thus, a contiguous locus of reference is adequate.

We make the simplifying assumption that references within the locus are distributed uniformly. Thus, a locus can be characterized by its starting address (page number \( p \)) and its extent (number of neighboring pages \( e \)). The following diagram illustrates the idea graphically.
Section 5

Performance Evaluations

The locus moves within the virtual memory as the process executes. The motion is gradual, with occasional transitions to new locations. To model the stable periods of gradual motion, we make another simplifying assumption: the locus is moving in the same direction at a constant rate, \( m \), i.e., \( p \) is incremented by 1 every \( m \) references. To model the transition periods, we change the locus probabilistically. That means, each time we generate \( m \) references, the locus moves, with some chosen probability \( t \), to a new location \( p \).

Using the above assumptions, we can generate a reference string as follows:

- select a memory size (\( P \)), a starting location (\( p \)), an extent (\( e \)), a rate of motion (\( m \)), and a probability of transition (\( t \));
- repeat the following steps until a reference string of desired length is generated {
  - pick \( m \) random numbers in the range \( p \) through \( p+e \) and write them to a file (each number represents one reference);
  - generate a random number \( 0 \leq r \leq 1 \);
  - if \( r < t \) generate new \( p \) /* transition */
  - else increment \( p \) by 1 (modulo \( P \)); } /* stable period */

The values of \( p \) in the above algorithm are chosen as uniformly distributed random numbers in the range \([0..P-1]\). The values \( P, e, m, \) and \( t \) are constants that allow us to vary the properties of the generated reference strings. The value of \( P \) is the assumed size of the allocated virtual memory. When choosing values for \( e \), keep in mind that the typical size of a working set is in the range of a few tens to perhaps 100 pages. When choosing values for \( m \), keep in mind that a typical program references any page only a few hundred times, on average. When choosing the value of \( t \), keep in mind that the stable periods typically last for thousands of instructions.

5 PERFORMANCE EVALUATIONS

The following is a suggested list of possible experiments that can be run using the different page replacement algorithms.

1. Using different reference strings and different memory sizes, answer the following questions:

   (a) How much is gained by using FIFO over a random replacement? How much is gained by using LRU over FIFO?

   (b) How close is LRU to the optimal (MIN)?

   (c) How effective are the second-chance and third-chance algorithms in approximating the LRU algorithm? For the third-chance algorithm, the reference string \( RS \) must differentiate between read and write accesses. This can be done during the generation of \( RS \) (e.g., by representing write accesses as negative numbers), or by augmenting the page-replacement algorithm itself. In either case, choose a constant \( 0 \leq W \leq 1 \), and assume that, with probability \( W \), any given request is a write operation.

For all cases, present your results in terms of the average number of page faults for a given \( RS \). For case (c), also estimate the overhead (in terms of numbers of operations) generated by different algorithms.
2. For the local page replacement algorithms, determine the following:

   (a) How close is WS to the optimal replacement VMIN?
   (b) How does the page fault rate (average number of page faults for a given reference string) vary with different values of $\tau$ for WS, VMIN, or PFF?
   (c) How close is PFF to WS?

The main measure in all cases is again the average number of page faults for a given RS.

3. Compare a local page replacement algorithm with a global replacement algorithm. For this comparison, choose a local page replacement algorithms and a value for $\tau$; determine:

   (a) the average number of frames, $nf$, needed to accommodate the working set
   (b) the average number of page faults, $pf_{loc}$

Then use the number $nf$ as the memory size for a global replacement algorithm, such as $LRU$; determine the number of page faults, $pf_{glob}$, for this algorithm and compare it with $pf_{loc}$.

6 SUMMARY OF SPECIFIC TASKS

1. Develop a set of page replacement algorithms (global, local, or a combination of both).
2. Develop a program to generate reference strings with different characteristics.
3. Compare the performance of the different algorithms using the reference strings.
4. Document the results of your experiments in a report.

7 IDEAS FOR ADDITIONAL TASKS

1. Implement the variation of the $LRU$ that uses aging registers (emulated in software), as described in Section 8.3.1 (Chapter 8). Determine a suitable size of the registers for different scenarios. As a guideline, note that the registers should not all be zero at the time of a page fault to be of any use.
2. Extend the experiments to cover multiprogramming systems. Assume $n$ processes, each maintaining its own locus of reference. Generate a reference string for each process. Assume all processes are to execute in a RR fashion. That means, during each quantum, the current process generates a number, $q$, of references from its reference string. After $q$ references or whenever a page fault occurs, execution switches to the next process. References from the new process reference string are used until again $q$ references have been used or a page fault occurs.

Using the above scheme, study the effects of load control and thrashing. That means, assume a time, $s$, it takes to service a page fault. Thus, the faulting process remains blocked (cannot issue any references) for $s$ number of steps following a page fault. Then vary $s$, $n$, and $q$; record and plot the frequency of page faults for a chosen algorithm as a function of these variables. Also record the periods of time during which the CPU is idle, i.e., when all processes are blocked waiting for their page fault to be serviced; when does thrashing occur?