Advanced Operating Systems
Practical 2: Virtual memory and replacement algorithms

UAH, Departamento de Automática, ATC-SOL
`http://atc1.aut.uah.es`

Theme 2

Abstract

The goal of this practical is to study the page replacement algorithms used in virtual memory management mechanisms, remarking the distinction between the tasks carried out in this context by the hardware and those carried out by the operating system.

1 Introduction

The objective of this practice is to study the mechanisms of virtual memory management. To this end, an MMU (Memory Management Unit) will be simulated along with the corresponding part of the operating system.

1.1 The program `gen_trace`

The program `gen_trace` (generate trace) was created to feed the simulator with a realistic trace of read/write operations. For this purpose, `gen_trace` executes a sorting algorithm on the data of an array, and gives as output a log of the operations carried out by this sorting algorithm. Below is an example of this output:

```
user@host:$ ./gen_trace MER RAN 4
 T8
 R0 W4 R1 W5 R2 W6 R3 W7
 R0 R1 C W4 W5 R2 R3 C
 W6 W7 R4 R6 C W0 R7 C
 W1 R5 C W2 W3 Sorted ;-)
user@host:$
```

The letter R denotes a read operation, and the letter W denotes a write operation. In both cases, the number following the letter indicates the position of the array being accessed. The letter T appears only once, and it indicates the total size of the array. In the example, the size of the array is of 8 elements, and the read/write operations refer the positions 0 through 7. After executing the sorting algorithm, `gen_trace` traverses the array to check if is really sorted, and prints a message. The trace can be considered finished when we reach the letter S (sorted) or the letter O (out of order).

Note that in the above example 8 elements had to be accessed in order to sort only 4 elements. This is because the algorithm used was *mergesort* (sort by merging ordered lists), which needs additional space.

The program `gen_trace` takes three parameters:

1. Sorting algorithm: BUB, INS, SEL, HEA, COM, MER, QUI, o QRP; which indicate, respectively: bubble sort, insertion sort, selection sort, heapsort, combsort, mergesort, quicksort and quicksort with random pivot.
2. Initial state of the array: ASC, DES or RAN, which indicate, respectively: ascending order, descending order and random order —or rather disorder.
3. Number of elements of the array to be sorted —not counting the additional space required by the mergesort algorithm.

1.2 Size of the traces

The length of the traces generated by `gen_trace` will depend on the chosen algorithm, the initial state, and the size of the array to be sorted.

Below is the number of operations needed to perform each algorithm with three different initial states, and with arrays of 10, 100 and 1000 elements:

```
Initial state: ASC
==================
  Size      BUB      INS      SEL      HEA      COM      MER      QUI      QRP

    10       19       19       99      160       46      103      108       85
   100      199      199     9999     2972     2596     1860    10098     1661
  1000     1999     1999   999999    43496    47383    26884  1.0e+06    23792


Initial state: DES
==================
  Size      BUB      INS      SEL      HEA      COM      MER      QUI      QRP

    10      189      145      109      172       73      107      115       88
   100    19899    14950    10099     3111     3149     1900    10150     1819
  1000  2.0e+06  1.5e+06  1.0e+06    44879    52480    26996  1.0e+06    26229


Initial state: RAN
==================
  Size      BUB      INS      SEL      HEA      COM      MER      QUI      QRP

    10      133       84      117      165       79      109       71       78
   100    14950     7763    10197     2998     3376     2080     1761     1675
  1000  1.5e+06   741726  1.0e+06    42956    59179    30674    26701    34214
```

Note that there are some striking differences, both between different algorithms and between different initial states for the same algorithm. The selection algorithm (SEL), for instance, is specially slow in every case, while the heapsort algorithm (HEA) is reasonably fast in every case. On the other hand, the quicksort algorithm (QUI) is the fastest one when the data are initially in random order, but it is very slow when they are initially already sorted. This is because the implementation of quicksort in `gen_trace` always chooses the first element as pivot. The quicksort algorithm with random pivot (QRP) is very fast in these experiments but, if the sequence of random numbers —the sequence used to choose the pivot— is known, an initial state can be generated in order to make the algorithm behave as badly as the normal quicksort.

For more information about sorting algorithms, see the Wikipedia.

2 Working sets

The working set of a program section is the group of memory pages it references.

Throughout the execution of a process, there are periods in which it focuses on a small working set, reusing a few pages for a long time, and there are periods in which many different pages are quickly referenced.

The smaller the working set, the more likely it is that the pages referenced are present in physical memory frames. On the other hand, the larger the working set, the more likely it is that some pages are removed from their frames to make room for others, which will then provoke page faults when the removed pages are referenced again.

The program `calculate_ws` makes a rudimentary estimate of the working set along an execution of `gen_trace`. Figure 1 shows a graph depicting the output of `calculate_ws` for an execution of `gen_trace` with the mergesort algorithm. The initial peak is due to the fact that this implementation of mergesort starts copying the entire array to be sorted in the temporary array. Then, the working set is considerably reduced due to the order in which sorted lists are built from smaller sorted lists: first, a sorted pair is formed with elements 0 and 1; then, another sorted pair is formed with elements 2 and 3; and then the two pairs are joined —merged— in a list of 4 elements. Then, another list of 4 elements is formed following the same procedure, and merged with the previous one, generating a sorted list of 8 elements, and so on.

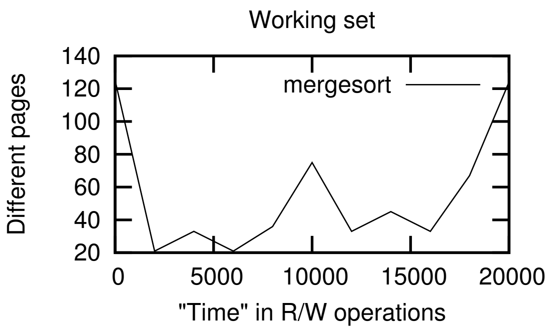

Figure 1: Graph of the working set of an execution of the mergesort algorithm

As the sorted lists grow larger, more different memory locations are quickly accessed, thus enlarging the working set. The last merge of sorted lists accesses all the positions of the array in a short period of time, making the working set as big as in the initial peak.

The program `calculate_ws` takes 5 parameters:

1. Number of elements that fit in a page.
2. Number of operations per interval. The number of different pages referenced during every interval will be counted.
3. Algorithm (as in `gen_trace`).
4. Initial state (as in `gen_trace`).
5. Number of elements of the array to be sorted (as in `gen_trace`).

The output of `calculate_ws` has the ideal format to feed the program `gnuplot`. Generate the graph of the working set of another sorting algorithm as in the following example:

```
user@host:$ ./calculate_ws 16 2000 QRP RAN 1000 >ws_qrp.txt
user@host:$ gnuplot

        G N U P L O T
        Version 4.2 patchlevel 5
        [..]

Terminal type set to 'wxt'
gnuplot> set encoding iso_8859_1
gnuplot> set xlabel "\"Tiempo\" en operaciones de L/E"
gnuplot> set ylabel "P\341ginas diferentes"
gnuplot> set title "Conjunto de trabajo"
gnuplot> plot "ws_qrp.txt" using 1:3 title "QRP" with lines
gnuplot> exit
user@host:$
```

Once entered the `plot` command, the graph should appear in a new window. You can store the commands for `gnuplot` in a text file, so that you do not need to type them again every time you need to paint the graph. In that case, you must run `gnuplot` passing it, as a parameter, the name of the file containing the commands. In addition, you might want to append to the file the command `pause -1 "Press ENTER"` to prevent the graph window from closing immediately.

The program `gnuplot` can also directly record the graphic to an EPS file (*Encapsulated PostScript*). To take a case in point, the graph in Figure 1 has been generated by means of a text file containing the following commands:

```
set terminal postscript portrait enhanced \
    mono dashed lw 1 "Arial" 9

set out "graphic_ws.eps"

set size 0.50, 0.18
set size ratio 0.5

set xlabel "\"Time\" in R/W operations"
set ylabel "Different pages"
set title "Working set"

plot "table_ws.txt" using 1:3 title "mergesort" with lines
```

Compare the evolution of the working set of bubblesort with that of insertion-sort. Use a disordered array of 1000 elements —in random order—, pages of 16 elements and intervals of 100000 operations.

3 Virtual memory management simulator

The rest of this practical will entail completing, and then modifying a program that simulates the operation of an MMU (Memory Management Unit) and the part of the Operating System that manages virtual memory. The simulator is programmed almost entirely, and just needs some additional functions to run.

As in `calculate_ws`, the `main` function of the simulator executes `gen_trace` and interprets its standard output. For every read/write operation, it invokes the function `sim_mmu`, which simulates the access to the specified virtual address.

Open the file `sim_paging.h` and read carefully the declaration of the structure type `spage`. The structures of this type represent entries of the page table. The page table, of course, is simulated with an array of `spage` structures. The field `present` indicates whether the page is loaded in physical memory —occupying a frame—. If this field is 0, the rest of the fields are considered invalid. If it is 1, then the field `frame` stores the number of the physical frame where the page is loaded, and the field `modified` indicates whether the page has been written since it was loaded, or if it has only been read. The fields `referenced` and `timestamp` will help while simulating systems with, respectively, FIFO (First In → First Out) with 2^{nd} chance replacement and LRU (Least Recently Used) replacement.

The fields `present`, `modified` and `referenced` are stored occupying one byte each to make the code of the simulator more readable. In a more realistic simulation, they should be packed occupying just one bit each.

The page table must be known and maintained by both hardware and operating system. In the simulator, the functions `sim_mmu` and `reference_page` —implemented in `sim_pag_random.c`— play the role of the hardware. The rest of the functions simulate the behaviour of the operating system.

See the function `reference_page`. Like all the functions covered here, it receives a pointer `S` that points to a structure that stores the state of the simulated system. Among other things, this structure contains a pointer to the page table, and some memory reference counters. In addition, the function receives the number of the page being accessed, and the type of operation (R/W). The function `reference_page` increments the counter of read operations or the counter of write operations —whichever corresponds to the type of operation— and, if the operation is write, it activates the `modified` flag in the corresponding entry of the page table.

Complete `sim_mmu` following the next instructions. First, `sim_mmu` must calculate the page number and offset basing on the virtual address. The page size is stored in the field `pagsz`[1] of the structure pointed by `S`.

```
page   = virtual_addr / S->pagsz;    // Quotient
offset = virtual_addr % S->pagsz;    // Remainder
```

In a more realistic simulation, the chosen page size would be a power of 2. As a result, the above calculation would simply consist in splitting the virtual address —stored in binary— into two pieces.

Then, `sim_mmu` must check that the access to the specified address is legal:

```
if (page<0 || page>=S->numpags)
{
    S->numillegalrefs ++;
    return ~0U;
}
```

After calculating and checking the address, the page table must be consulted to see if the page is presently loaded in physical memory. If it is not, `sim_mmu` must raise a page fault trap, that is, interrupt the process and invoke the operating system to resolve the problem. In the simulator, the function `handle_page_fault` will play the role of this routine of the operating system, dealing with loading the page in a memory frame.

```
if (!S->pgt[page].present)                      // Not present:
    handle_page_fault (S, virtual_addr);        // PAGE FAULT
```

Once the operating system has loaded the page in a frame and modified accordingly the page table, the hardware can resume the memory access operation.

The next step is to translate the virtual address to physical address:

```
// Now it is present

frame = S->pgt[page].frame;                // Calculate physycal
physical_addr = frame * S->pagsz +         // address
                offset;
```

Again, if the page size was a power of 2, the multiplication and the sum would be simply reduced to the operation of concatenating two binary numbers.

[1] In the simulation, every memory position contains one element of the array to be sorted. The page size is specified as the number of elements that fit in a page. The number of bits or bytes per element is irrelevant in this practical.

Also, the page must be marked as referenced:

```
reference_page (S, page, op);
```

Now `sim_mmu` just has to dump the memory access information on the screen if the user ordered to execute the simulator in mode D (detailed):

```
if (S->detailed)
    printf ("\t%c %u==P%d(M%d)+%d\n",
            op, virtual_addr, page, frame, offset);
```

Next we will study the role of the operating system on the virtual memory management. In this regard, during the execution of the process, the entry point to the operating system is the page fault handler. But before approaching it we will study other data structures that the operating system needs to manage.

If the page table was the only information available, the operating system would have to make expensive searches through the entire table to perform the following operations:

- Determine if a frame is free or occupied
- Find out which page is stored in a given frame
- Find a free frame —assuming they exist
- Choose a frame to replace the page that occupies it —when there is no free frame

The operating system solves this problem by maintaining a table of frames. The MMU does not need to know the existence of the table of frames because this table is maintained and used exclusively by the operating system.

See the declaration of the struct type `sframe` in `sim_paging.h`. The field `page` indicates the number of the page that is stored in the frame. The field `next` serves to maintain the free frames arranged in a linked list. It stores the number of the next frame in the list.

The linked list of free frames is circular. The field `listfree` of the `ssystem` structure — pointed by `S`— stores the number of the last frame of the list. The first element of the list can be reached by following the `next` field of the last one, because the list is circular. The function `init_tables` ensures that, at the beginning, the list of free frames contains all the frames. When the list is emplty, the value of `S->listfree` will be -1.

Complete the function `handle_page_fault` following the next instructions. First, calculate the number of the page that provoked the fault. Increase the count of page faults and, if the simulator is running in mode D (detailed), show a message.

```
S->numpagefaults ++;
page = virtual_addr / S->pagsz;

if (S->detailed)
    printf ("@ PAGE FAULT in P%d!\n", page);
```

If there are free frames, simply grab one out of the list —the first one is the handiest—, and occupy it with the requested page:

```
if (S->listfree!=-1)     // If there are free frames:
{
    last = S->listfree;          // Last of the list
    frame = S->frt[last].next;   // Take the next on (the 1st)

    if (frame==last)             // If they are the same frame,
        S->listfree = -1;        // then it was the only free one
    else
        S->frt[last].next = S->frt[frame].next; // Otherwise,
                                                 // bypass
    occupy_free_frame (S, frame, page);
}
```

If there are no free frames, we must choose an occupied frame and remove the page contained in it, in order to store the requested page instead. The replacement policy —the algorithm that selects the victim page— and the actual replacement operation are implemented separately.

```
else                     // If there are _no_ free frames:
{
    victim = choose_page_to_be_replaced (S);

    replace_page (S, victim, page);
}
```

See the code of the function `replace_page`. It is noteworthy that in a real operating system, this routine is not only responsible for updating the tables and load the new page in the frame, but also for dumping the victim page back to disc if it has been modified while it was in memory.

Program the function `occupy_free_frame`. You only need to establish the page-frame link and mark adequately the bits of the page. In a real system, this function would also read the page from disc to put in the frame.

3.1 Random replacement

Edit the file `Makefile` and add the program `sim_pag_random` to the target `all`. Compile and execute the simulator:

```
user@host:$ make
user@host:$ ./sim_pag_random 1 3 HEA DES 4 D
```

The previous command specifies a page size of just one element, a physical memory size of three pages, mode D (detailed) and execution of `gen_trace` with parameters `HEA DES 4` (heapsort algorithm, initial state of descending order, and array of four elements to be sorted). In the current set-up of the laboratory, the resulting report should agree with the next one:

```
---------- GENERAL REPORT ----------

Read references:          20
Write references:         17
Page faults:              7
Page dumps to disc:       2

---------- PAGES TABLE ---------

      PAGE    Present      Frame   Modified
       0         1           1         1
       1         1           0         1
       2         1           2         1
       3         0           -         -

---------- FRAMES TABLE ----------

     FRAME       Page    Present   Modified
       0          1         1          1
       1          0         1          1
       2          2         1          1

--------- REPLACEMENT REPORT ---------

Random replacement (no specific information)

-------------------------------------

PAGE FAULTS: --->> 7 <<---
```

3.2 LRU replacement

What follows is a version of the simulator with other replacement policy. Make a copy of the file `sim_pag_random.c` and call the new file `sim_pag_lru.c`. Edit the file `Makefile` and add the program `sim_pag_lru` to the target `all`. Modify the comment in the header of `sim_pag_lru.c` to make it fit the name of the file.

The LRU (Least Recently Used) replacement policy consists in selecting as victim of the replacement the least recently used page in the hope that it will not be referenced in the near future either. Implement this policy making the following modifications:

1. Add instructions to the function `reference_page` to store the value of the clock in the field `timestamp` of the accessed page, and then increment the value of the clock. Add also a check to print a warning message when the clock overflows back to 0.
2. In `choose_page_to_be_replaced`, implement a sequential search for the occupied frame with the lowest timestamp. Replace the word "random" with "LRU" in the message of the call to `printf`. Erase the function `myrandom`.
3. In `print_page_table`, add a column displaying the value clock−timestamp for the pages present in memory.
4. Make `print_replacement_report` show the value of the clock and the minimum and maximum *timestamp*s of the pages present in memory.

Compile and execute the new program `sim_pag_lru`. Verify that the results make sense. You can use the numbers of page faults shown in the LRU column of Table 1 as a reference.

Table 1: Page faults depending on the replacement algorithm

Parameters	Random	LRU	FIFO	FIFO2nd	Optimal
16 3 HEA DES 100	280	282	283	285	171
16 8 HEA DES 1000	3101	2436	2877	2642	1360
16 32 HEA DES 10000	15614	10802	13427	11211	8792
16 3 MER DES 100	158	104	119	118	83
16 8 MER DES 1000	1111	905	907	898	722
16 32 MER DES 10000	10190	9549	9458	9530	8146

3.3 FIFO replacement

Make a new copy of `sim_pag_random.c` an call it `sim_pag_fifo.c`. Repeat the initial steps of the previous section, this time to create the program `sim_pag_fifo`.

Implement the FIFO replacement policy. This policy consists in removing pages in the same order in which they were loaded. That is, the page that enters first, leaves first (First In → First Out). To achieve this, maintain a circular linked list of the occupied frames like the list shown in Figure 2. The field `listoccupied` of the structure `ssystem` will point to the last element.

Make the next modifications:

1. As in the previous section, delete the elements that correspond to the random replacement policy.
2. In `occupy_free_frame`, append the frame to the list of occupied frames —insert it between the last and the first one, and then make `listoccupied` point to the newly added element—. Take into account that the list is initially empty, and the value of `listoccupied` is -1.
3. In `choose_page_to_be_replaced`, choose the page stored in the first frame of the list —the one that follows the last one— as the victim. Then move it to the end of the list —make `listoccupied` point to the chosen frame.
4. In `print_replacement_report`, show the list of occupied frames.

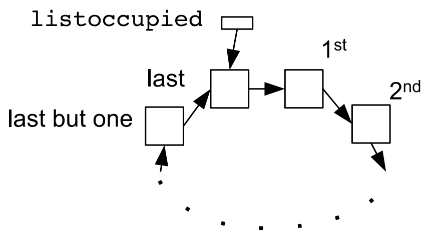

Figure 2: Circular list of occupied frames

Compile the program and execute it in mode D (detailed) with a reduced number of pages and frames, in order to verify that the replacement is carried out in FIFO order. Then execute it with the parameters of any example from Table 1 to check that the numbers of page faults are equal.

3.4 FIFO replacement with second chance

Make a copy of `sim_pag_fifo.c` an call it `sim_pag_fifo2ch.c`. Repeat the initial steps of the previous section, this time to create the program `sim_pag_fifo2ch`.

Modify `sim_pag_fifo2ch.c` to implement the replacement policy FIFO with 2^{nd} chance. This policy consists in granting a second chance to the first frame of the FIFO queue, but only if the page has been referenced since the last time it was the first in the queue. That is, pardon this page, moving the frame to the end of the list, but resetting the reference bit of the page.

Make the next modifications:

1. In `reference_page`, set to 1 the reference bit of the page.
2. In `choose_page_to_be_replaced`, program a loop that skips the frames of referenced pages —resetting to 0 their reference bit and making `S->listoccupied` advance— until finding a frame whose page has a 0 in its reference bit —this will be the victim of the replacement.
3. In `print_page_table` and `print_frames_table`, add a column showing the reference bit of the pages.
4. Modify also `print_replacement_report` to show the reference bit of the page stored in every frame.

Compile the program and execute it in mode D (detailed) with a reduced number of pages and frames, in order to verify that the replacement is carried out in FIFO with 2nd chance order. Then execute it with the parameters of any example from Table 1 to check that the numbers of page faults are equal.

3.5 Optimal replacement

The best possible replacement policy is to replace, in each case, the page that will take longer to be referenced. For this, the operating system would need to predict the future accurately and efficiently[2].

Obviously, the optimal replacement is not feasible in a real system. However, it is interesting to simulate because its behaviour is the ideal model to which a good replacement algorithm should come close.

The program `sim_pag_optimal` simulates the optimal replacement by "cheating", that is, storing the whole trace in memory before starting the simulation in order to know in every step what will happen next, and decide accordingly.

Execute `sim_pag_optimal` in mode D (detailed) with a reduced number of pages and frames, and watch the result. For example:

```
user@host:$ ./sim_pag_optimal 1 1 BUB RAN 2 D | grep @
@ PAGE FAULT in P0!
@ Hosting P0 in F0
@ PAGE FAULT in P1!
@ Choosing P0 (will take 1 operations to be used again) of F0 for replacing it
@ Replacing victim P0 with P1 in F0
@ PAGE FAULT in P0!
@ Choosing P1 (will take 1 operations to be used again) of F0 for replacing it
@ Replacing victim P1 with P0 in F0
@ PAGE FAULT in P1!
@ Choosing P0 (won't be used anymore) of F0 for replacing it
@ Writing modified P0 back (to disc) to replace it
@ Replacing victim P0 with P1 in F0
```

[2]The most efficient way to accurately predict the future is to wait until it happens. For now. In the future we'll see...