Comp111 Assignment 4 - Paging

Overview

In the case of limited physical memory, processes must contend for pages. A common strategy for dealing with paging is "lookahead", in which one tries to predict what a process will do from its pattern of references.

In this assignment, you will write a paging strategy for a simulator. The simulator assumes that there are several processors sharing a pool of memory. You must assign which pages to assign to what processes, and when. You are graded on the latency of your solution, compared to that of others, i.e., how long your processes have to block. While paging in memory that it needs, a process is blocked; paging and processing for other processes occur concurrently while paging is occurring.

Objective

Your objective is to write a pager that will optimally swap in and out pages of memory so that a small pool of processes completes as quickly as possible. Your pager has the following form:

```
#define MAXPROCPAGES 20   /* max pages per individual process */
#define MAXPROCESSES 20   /* max number of processes in runqueue */
#define PAGESIZE 128      /* size of an individual page */
#define PAGEWAIT 100      /* wait for paging in */
#define PHYSICALPAGES 100 /* number of available physical pages */

typedef struct pentry {
   int active;
   int pc;
   int npages;
   int pages[MAXPROCPAGES]; /* 0 if not allocated, 1 if allocated */
} Pentry ;

// you write this:
void pageit(Pentry processes[MAXPROCESSES]); /* your routine */
// you use these simulated "system calls":
extern int pageout(int process, int page); /* move a page out */
extern int pagein (int process, int page); /* move a page in */
```

You write the routine `pageit`, which calls my routines `pagein` and `pageout` to accomplish paging. Your routine `pageit` gets a page map for each process as well as the program counter for it. By tracking the program counter, you might be able to figure out which page to page in next. Note: the process is getting nowhere if the program counter points to a page outside memory that has been paged in. This is true if `pages[pc/PAGESIZE]==0`, and is the initial state of the simulation.

This would be easy if there were an unlimited number of physical frames; in fact there are a small number and you'll have to be clever about how you page processes in and out. There are 100 physical pages and 20 processors, each of which runs one process. So you are extremely memory starved.

Note that your pager has no knowledge of the actual nature of processes, or even which processes are assigned to which processors. Processes are randomly generated, where the randomization determines branching structure and the probability of branching. To make things easier, only instruction memory is considered.

Hints

First, some overall comments:

- *I have no idea at this point what the best solution is.* I change some of the simulator conditions each time I run this assignment, mostly to invalidate old solutions. This time, there is no delay for paging something out. This changes a lot of things and allows some new strategies (and better performance than could be achieved last time). Another thing I changed is that no process is repeated this time, so "process profiling" is not as useful as before (though not totally useless, either).
- *The problem is designed to punish elaborate solutions in favor of simple and cleanly implemented strategies.* If you find yourself writing a ton of code you are probably on the wrong track. Instead, *use the visualizer* to find out where your inefficiencies are hiding. *Spend your time testing and thinking* rather than just coding. Many short solutions solve the problem well. Which one is best?
- *This problem exhibits strongly diminishing returns* as you continue to work on it. While your earlier ideas will improve performance very much, your later changes will make very slight improvements. The difference between a 10 and a 9 will be very small. To compensate for that, your solution will be compared to all other solutions for the same (secret) set of random seeds. Thus every solution will be tested on the same scenerios. These scores will be averaged and your final score will be based upon the average. Then, in the week after the main duedate, I will expose the seeds to the class, so that you can make another try with more information.

The following facts may help you:

- `pagein` and `pageout` return 1 if they succeed in *starting* a paging process or if a paging process is already started, 0 if not (e.g., if the request is invalid or the page is currently paging out). The "swapping in" process ends 100 ticks later; the only indication you get is that the contents of the Pentry change.
- When you call `pagein`, it takes 100 ticks for the page to become available (ostensibly, from disk).
- When you call `pageout`, the page is flushed immediately; then you can do a subsequent `pagein` to reuse the flushed page.
- There is no harm whatever to use `pagein` to probe for whether a page is available to page in; if it returns 1, the page is paging in; if it returns 0, it is not available to page in. Repeatedly calling `pagein` until it succeeds is sufficient.
- Same for `pageout`; it only returns 0 if the page is already paged out.

Strategies you might use for paging:

- Least-recently-used (LRU): when flushing a page; flush the one that's least recently accessed. Yes, that means you have to keep track of when. `pageit` is called once per clock tick so you can keep track of time in a global variable. Flush the least-recently accessed page over *all* processes.
- Predictive: when flushing a page, flush the one that you *predict* will not be accessed for the longest time.

Several other observations that might help:

- When a process exits, *all* of its pages are released instantly. You can use this to gain access to pages.
- You are resource starved and deadlock is possible. To avoid deadlock, make sure that *you* release resources from some process to make resources available for others.
- You cannot tell when processes exit other than by observing that the pc is 0 and that the whole page table is clear.

Helper files

There are a number of helper files available to you in /comp/111/assignments/a4. In this assignment I am going to give you the source to the simulator so you'll know exactly what's going on inside it. This is contained in /comp/111/assignments/a4/t4.c and includes /comp/111/assignments/a4/t4.h. There is also an example program /comp/111/assignments/a4/a4.c that does rather badly in performance.

Grading

The simulator computes the ratio of paging to useful work. You are graded on the smallness of ratio that you can achieve. The beginning solution is downright awful and there is a lot of room for improvement. Smaller ratios are better; larger ratios are worse. Your solution will be run several times and the average used to compute your grade. The assignment is worth 10 points, including 8 points performance and 2 points style.

Getting started

There is a brain-dead solution in `/comp/111/assignments/a4/a4.c`. It demonstrates the interface but has terrible performance. To get a copy,

```
mkdir a4
cd a4
cp /comp/111/assignments/a4/a4.c a4.c
```

Testing

To test your solution, if it is `a4.c`, type

```
ln -s /comp/111/assignments/a4/Makefile Makefile
make
./a.out
```

This will create symbolic links in your directory for tester files `t4.c`, `t4.h`, `programs.c`, etc.

The tester in this assignment is a white-box tester; you have source code. Hopefully this will be useful. Please pay attention to the news for bug fixes and enhancements. Some observations may be helpful:

- `a.out -help` gives options available. These include printouts of what the tester is doing over time.
- If you create a deadlock, the tester detects it and prints a dump of all state.
- The "programs" being run are stored as branch tables. I simulate them without doing any real computation.
- Each time through, 40 randomly generated programs are each run once. The programs do *not* share memory and accesses to data memory are not simulated (I know that this is not realistic). They are scheduled in random order, back to back with other processes.
- You may work on any station; the code is fairly portable and I am not grading you on computation time. However, please be considerate and work on `comp111-01` to `comp111-06` so as not to overwhelm public servers.

The tester options

The tester has several options that can help you debug your program. If you run `./a.out -help` it will print the following.

```
-all       log everything
-load      log loading of processes
-unload    log unloading of processes
-branch    log program branches
-page      log page in and out
-seed 512  set random seed to 512
-procs 4   run only four processors
-dead      detect deadlocks
-csv       generate output.csv and pages.csv for graphing
```

- `-seed 21` means to generate exactly the same processes each time, based upon the seed. If you do not choose a seed, one will be chosen randomly. Thus you can repeat experiments that don't work.
- `-all` logs everything the tester does to `stderr`. This is a lot of output. You can also choose to log single things, like `-load` (loads and unloads of processes), `-branch` (whether branches in the code are taken), `-page` (paging in or out).
- `-csv` generates two (large) files, output.csv and pages.csv, for use as input to the output visualizer `see.R` included with the assignment.

Running see.R

When you are trying to tune performance (and not when doing basic debugging) it helps to be able to visualize what is happening. The R program `see.R` takes as input two files `output.csv` and `pages.csv`, and makes a visual representation of the performance of your algorithm. To run it:

- Obtain a current version of R (runs on Windows, Linux, and MacOS, and is installed on the servers).
- Move `output.csv` and `pages.csv` to a local workstation. Also make a copy of `see.R` in the same directory.
- Invoke R and `setwd` to the directory containing `output.csv` and `pages.csv`. In windows, from the R command line, one would do:

  ```
  > getwd()
  [1] "C:/Users/Alva/Documents"
  > setwd('../Desktop')
  > getwd()
  [1] "C:/Users/Alva/Desktop"
  >
  ```

  Where presumably `output.csv`, `pages.csv`, and `see.R` are located.
- Type `source('see.R')` to run the visualizer.

In the visualizer

- Timelines from left to right indicate the state of a processor (20 processors).
- Clicking on a timeline displays the detailed trace for a processor's program counter.
- red means blocked, green means unblocked.

Submitting assignments

To submit this assignment, if your program is "a4.c", ssh to one of the comp111 machines and type

```
provide comp111 a4 a4.c
```

You will see your results with the command "progress comp111".