

## 433-332 Project 2 - 2003

The aim of this project is to increase your familiarity with issues in memory management. The project is divided into two parts. Part A is practical and Part B is theoretical.

### Part A

Your task is to write a simulator which takes process of different sizes and loads them into memory using one of three different algorithms and when needed, swaps processes out to create a sufficiently large hole for a new process to come into memory.

The three different algorithms to be used are first fit, best fit and worst fit. Assume that memory is partitioned into contiguous segments, where each segment is either occupied by a process or is a hole (a contiguous area of free memory). The *free list* is a list of all the holes. Holes in the free list are kept in ascending order of memory address. Adjacent holes in the free list should be merged into a single hole.

The three algorithms to be used for placing a process in memory are:

- *First fit*: First fit searches the free list from the beginning, and uses the first hole large enough to satisfy the request. If the hole is larger than necessary, it is split, with the process occupying the lower address range portion of the hole and the remainder being put on the free list.
- *Best fit*: Chooses the smallest hole from the free list that will satisfy the request. If multiple holes meet this criterion, choose the earliest one in the free list. If the hole is larger than necessary, it is split, with the process occupying the lower address range portion of the hole and the remainder being put on the free list.
- *Worst fit*: Chooses the largest hole from the free list that will satisfy the request. If multiple holes meet this criterion, choose the earliest one in the free list. If the hole is larger than necessary, it is split, with the process occupying the lower address range portion of the hole and the remainder being put on the free list.

The textbook describes these in more detail on page 200.

The details of the behaviour of the simulation are now described.

A *process size file* is a sequence of entries which describe an initial queue of processes waiting to be swapped into memory from disk. The first entry is the head of the queue and the last is the tail of the queue. Each entry consists of a (process-id, memory-size) pair. For example:

```

4 98
2 33
1 1000
3 5

```

This models an initial queue where process 4 is at the head of the queue and is 98 MB in size, process 2 is second in the queue and is size 33 MB, process 1 is third in the queue and is size 1000 MB, etc.

Points to note:

- Each process id is a unique positive integer.
- Each process size is a positive integer  $\leq m$  (the main memory size).

You may assume the input file being read in will always be in the correct format.

The simulation should behave as follows:

- Parse the process file to obtain the initial queue of processes waiting to be swapped into memory.
- Assume memory is initially empty.
- Load the processes from the queue into memory, one by one, according to one of the three algorithms.
- If a process needs to be loaded, but there is no hole large enough to fit it, then processes should be swapped out, one by one, until there is a hole large enough to hold the process needing to be loaded.
- If a process needs to be swapped out, choose the one which has been in memory the longest (measured from the time it was most recently placed in memory).
- After a process has been swapped out, it is placed at the end of the queue of processes waiting to be swapped in.
- Once a process has been swapped out for the fourth time, we assume the process has finished and it is not re-queued. Note that not all processes will be swapped out for four times.
- The simulation should terminate once no more processes are waiting to be swapped into memory.

Your program should print out a line of the following form after each time a process has been swapped into memory

```
15 loaded, numprocesses=3, numholes=2, memusage=77%
```

where '15' refers to the id of the process just loaded, 'numprocesses' refers to the number of processes currently in memory and 'numholes' refers to the number of holes currently in memory. 'memusage' is an integer referring to the percentage of memory currently occupied by processes.

Your program must be called *swap* and the name of the process size file should be specified at run time using a '-f' *filename* option. The placement algorithm to be used should be specified using a '-a' *algorithm\_name* option, where *algorithm\_name* is one of {first,best,worst}. The size of main memory should be specified using a '-m' *memsize* option, where *memsize* is an integer.

## Part B

This part requires answers to the following questions. Place your answers in a file named *comments*.

- (2 marks) The amount of disk space that must be available for page storage is related to the maximum number of processes,  $n$ , the number of bytes in the virtual address space,  $v$  and the number of bytes of RAM,  $r$ . Give an expression for the worst case disk space requirements, showing working. How realistic is this amount ?
- (2 marks) A programmer who writes programs to exhibit good locality can expect marked improvement in the execution efficiency of their programs. List and justify two high level language features or data structures that should be emphasised for achieving good locality. List and justify two that should be avoided.
- (1 mark - hard) Let the number of segments of memory that are occupied with processes be  $N$ . Let the number of segments that are holes be  $H$ . Show that, on average,  $H = 0.5 * N$ . List any assumptions you have made.

## Assessment

**Marking:** This project is worth 15% of your final mark for the subject. Your submission will be tested and marked with the following criteria:

- 6 points for the program giving the expected output.
- 4 points for the coding style, design and documentation in Part A. Efficiency and clarity of code are both important.
- 5 points for answers to Part B

**Individual Work:** You are reminded that all submitted assignment work in this subject is to be your own individual work. Students submitting the work of

others for assessment will be penalised by the Department, and risk prosecution under the University's Discipline Regulations. Students who allow other students access to their work will also be penalised, even if they themselves are sole authors of the program in question. *All work must be your own and no-one else's.* Automated similarity checking software will be used to compare submissions.

**Subject Assessment Policy:** The two programming projects for 332 comprise 30% of the marks. The exam is worth 70% of the marks. There are also two hurdle requirements: you must obtain a total of at least 15/30 for the projects and 35/70 for the exam. Students who fail either the exam or the project hurdle will have their mark adjusted so as to ensure that they fail the subject as a whole by the amount by which they failed the hurdle. For example, a student with a mark of 25/30 for the projects and 28/70 in the exam (seven marks short of the hurdle) will be assigned a final mark of 43.

**Questions and further information:** The subject WWW home page will be kept updated with any further information relevant to the project and will be considered as part of the specification for the project. Questions about the project specification should be directed to the newsgroup cs.332

**Submission:** You must submit program file(s), Makefile and comments file, using `submit 332 2`, not later than Monday, 19th of May, by 5:00pm. Late submissions will incur a deduction of 3 marks per day (or part thereof). Please include your name, login id and student number in a comment at the top of each file.