CS2506 Operating Systems II Lab 2 1/03/2022
Main memory management

Laboratory Goals / Objectives

Students will investigate kernel services for the management of the main memory by designing several algorithms in pseudo-code and implementing them with Python/Java. By the end of the lab, an understanding of the memory kernel services should be demonstrated, by providing a working simulation of the algorithms implementation and screenshots with results of the execution.

1. Memory kernel services

The OS kernel manages free (not allocated) memory blocks or page frames which can be allocated to processes.

If the system is using pages of equal size, one bit/page frame can show the state of each page, free or not – this solution is called *free bitmap*. In some cases, it is more effective to use memory blocks of different size; a block can be a multiple of page frames. Free memory blocks can be managed by using a *linked list*.

When dealing with pages, allocation of memory to processes is easy – take first of the list, or when the page becomes free, append it to the list. If memory is allocated in variable-sized blocks, the list needs to be searched to find a suitable block.

When a process requires memory space, if more than one block can match the request, then the decision of the kernel service regards which one to select? There are several strategies such as first fit, next fit, best or worst fit, that however result in fragmentation.

When the memory is fully allocated, new memory requests can be met by replacing, either locally, or globally, pages already in use. For example, the working set is a common solution for replacing local pages. Another possibility is to move pages that are allocated but not currently in use to the disk.

The kernel can implement the functionality mentioned above in one service or in several distinct but cooperating services – for example, the allocation service uses the output of the free memory service.

Memory allocation

- *First fit* takes the first block from the list which is greater than or equal to the requested size. If the request cannot be met, it fails. This policy tends to cause allocations to be clustered towards the low memory addresses – the effect is that the low memory area gets fragmented, while the upper memory area tends to have larger free blocks.

- *Next fit* starts the search with the free block that is next on the list to the last allocation. During the search the list is treated as a circular one. If returned to the initial starting point without any allocation, the process fails. This strategy leads to a more evenly allocation of free memory.
- *Best fit* allocates the free block that is closest in size to the request. Like first fit, best fit tends to create significant external fragmentation, but keeps large blocks available for requests of larger sizes.
- *The buddy system.* All blocks are a power of 2 in size. Let n be the size of the request. Locate a block of at least n bytes and return it to the requesting process:
    - If n < smallest allocation unit, set n to be the smallest size.
    - Round n up to the nearest power of 2. Select the smallest k such that $2^k \geq n$.
    - If there is no free block of size 2^k, then recursively allocate a block of size 2^{k+1} and split it into two free blocks of size 2^k.
    - Return the first free block of size 2^k in response to the request.
- Each time a block is split, a pair of buddies is created; they'll either be split or paired together. It is easy to determine (by looking at bit k+1) which is the buddy of a block. The block that is de-allocated is a buddy to a free block; they are merged in order to create a larger free one.

Page replacement algorithms

- *First In First Out* assumes that pages are used for a finite period of time after which they become "old". The page selected here is the one that has been in memory the longest period of time. Implementation is done by using a queue – all new pages are added to the tail of the queue.
- *Second chance* is an extension of FIFO: when a page is pulled off the head of the queue, the accessed (A) bit is examined. If it's 0, the page is swapped out, else the bit is cleared and the page is reinserted at the tail of the queue. A second examination of the queue will produce available pages.
- *The clock algorithm* is similar to the second chance: two clock hands are moving synchronously above pages; the first hand points to the page currently examined. The second checks it again. If it has not been accessed within that time, it can be swapped out. The distance between the two hands determines how long the page is given to be accessed. In many implementations of the two-handed clock, the hands are not moved only when a page fault occurs. The pair of hands periodically advances some number of pages and keep a record of those with A = 0.

- *The working set* of pages corresponds to the number of pages a specific process is using at a time. There are variations, and two thresholds (called also watermarks) can be considered. They are the upper and lower bounds of the working set. If a process has allocated more pages than its upper threshold for the working set, it is a good candidate for page swapping. Contrary, if by taking a page the lower threshold is passed, it makes sense to swap all the pages. The two thresholds can be selected based on the page fault frequency. If a process generates page faults too often, it needs more pages; if it doesn't generate page faults for a time, probably it has too many pages.

2. Lab work

This lab work is about programming and testing a set of memory management algorithms for (1) keeping track of free memory, (2) allocation of memory to processes and (3) page replacement, if the entire memory space is allocated. Students can choose the algorithms they wish to implement for each service, for example one using the linked list for free memory consisting of variable-size blocks, first fit or the buddy system for allocation and the second chance replacement.

The following assumptions can be considered:

- The memory user space is 4 MB and a page is 4 KB.
- The main memory is organized in blocks of different sizes, for example, 16 blocks of 2 pages, 16 blocks of 4 pages, etc, up to the total of 4MB.
- Working scenario: initially, all blocks are free, but as memory requests arrive and are processed, they'll be allocated until there will be no free memory. From that moment, a replacement strategy needs to be used.
- A memory request will include the process id and the number of KB that is requested.
- All memory requests arrive in a queue and are processed according to their arrival order - FIFO.
- During the execution, some pages are de-allocated and become free again – the explanation is that processes finished their execution and their pages were released.
- Sometimes, it might be impossible to meet the memory requests. This situation can be dealt with as an error or a deferred operation.

Tasks to do

Task 1 First decide on the size of the main memory, the size of a page and the organization of the main memory in blocks of different sizes in terms of number of pages – how many blocks of what size. From now on, this is the memory organization managed by the kernel.
Choose an algorithm for free memory tracking, one for memory allocation and one for page replacement. Explain your algorithms choices. Determine how they interact and the data structures they will use. Use a diagram for illustration.

Task 2 Outline the memory management algorithms in pseudo-code. This will be your design. You should consider all aspects related to the interaction of the three services.

if process ref not found, it's added to memory i think

Task 3 Provide an implementation of your memory management algorithms in Python or Java. Add comments to your code.

Task 4 Test your implementation by simulating the execution of your algorithms. The simulation scenario should include a sequence of memory allocation and replacement requests that are processed in their order of arrival. By executing the simulation, the output should be some text tracing the memory management operations until completion – capture screenshots to be included in the report. Check and print the fragmentation of the main memory during simulation.

Submission

Return your results to Tasks 1-4 above, including the pseudo-code, the Python/Java code and **testing screenshots** in a pdf file, by the deadline.
Place your name and student number at the top of each class file. Place comments at the method level; use one-line comments inside method bodies to describe more complex statements if you feel they are not obvious.

Tasks 1 and 2 have 2 marks allocated each; tasks 3 and 4 are worth 3 marks each. The total for this lab is 10 marks.

For the completion of this lab, you will have three weeks:

- Task 1 and 2 should be completed by the end of the first week – 8/03.
- Task 3 should be completed by the end of the second week – 15/03.
- Task 4 should be completed by the end of the third week – 22/03. Report as pdf file only submitted by Canvas.