

Homework 5 Solution

1. We'll list the memory partitions ("holes") left over after adding each process.
 - (a) First-fit: adding the 212 KB process in the 500 KB partition leaves free partitions of sizes 100 KB, 288 KB, 200 KB, 300 KB, 600 KB. Adding the 417 KB process in the 600 KB partition leaves partitions of sizes 100 KB, 288 KB, 200 KB, 300 KB, 183 KB. Adding the 112 KB process in the 176 KB partition (was originally 500 KB) leaves partitions of sizes 100 KB, 176 KB, 200 KB, 300 KB, 183 KB. The 426 KB process cannot be added.
 - (b) Best-fit: adding the 212 KB process to the 300 KB partition leaves free partitions of sizes 100 KB, 500 KB, 200 KB, 88 KB, 600 KB. Adding the 417 KB process to the 500 KB partition leaves partitions of sizes 100 KB, 83 KB, 200 KB, 88 KB, 600 KB. Adding the 112 KB process to the 200 KB partition gives partitions of sizes 100 KB, 83 KB, 88KB, 88 KB, 600 KB. Adding the 426 KB process to the 600 KB partition gives partitions of sizes 100 KB, 83 KB, 88 KB, 174 KB.
 - (c) Worst-fit: adding the 212 KB process to the 600 KB partition leaves free partitions of sizes 100 KB, 500 KB, 200 KB, 300 KB, 388 KB. Adding the 417 KB process to the 500 KB partition leaves partitions of sizes 100 KB, 83 KB, 200 KB, 300 KB, 388 KB. Adding the 112 KB process to the 388 KB partition (was 600 KB) gives partitions of sizes 100 KB, 83 KB, 200 KB, 300 KB, 276 KB. The 426 KB process cannot be added.

Since only best-fit allows all the processes to be added to memory, it makes the most efficient use of memory in this case.

2. "Good" really means the technique has high spatial locality, so that when you read in a page on-demand, most of the data on the page will be used. "Bad" means that you have low spatial locality with lots of scattered memory references. So (a), (c), (f) are "good", (b), (g) are bad (hash tables have very bad locality). (d), (e) are mixed: binary search has bad locality at the start of the search, but improves towards the end (as we narrow in on the target), spatial locality of the code depends on the code in question. Loops have very good spatial locality.
3. The average time to service a page fault will be $0.7 \times 20 + 0.3 \times 8 = 16.4$ milliseconds. We'll take "memory access time" to include the page table access time, i.e. a paged memory access takes 100 nanoseconds. We can't assume a TLB since no TLB hit-rate is mentioned. We want to find a value α so that

$$\alpha \times (16.4 + 2 \times 0.0001) + (1 - \alpha) \times 0.0001 < 0.0002$$

Times are given in milliseconds. The first term is the time taken on a page fault (note two paged memory references: first on the failed reference, second after demand paging). The second term is for memory references to pages which are in memory. Rearranging:

$$\begin{aligned} 16.4002\alpha + 0.0001 - 0.0001\alpha &< 0.0002 \\ 16.4001\alpha + 0.0001 &< 0.0002 \\ 16.4001\alpha &< 0.0001 \\ \alpha &< 0.0000061 = 6.1 \times 10^{-6} \end{aligned}$$

4. (a) The big advantage of copy-on-write is that it defers copying a virtual memory page until it is modified by one of the sharing processes. Therefore, under the traditional Unix fork-exec model of process creation, very little time is spent copying the parent process's address space before the child does its exec. Copy-on-write could also be useful for a parent and child in the absence of an exec, if the two processes write very little of the data inherited by the child at the time of the fork.
 - (b) When a page marked copy-on-write is accessed, the hardware triggers an illegal-access exception, which the OS interprets as meaning that a copy has to be made of the page (n.b. only the OS understands the page is copy-on-write, the hardware only sees an access violation). This combination of access-exception and copy is more expensive than a straight copy before the process is started, and it happens at runtime, not at process setup time, so it interferes with the performance of the process doing the write. Copy-on-write would therefore be a bad choice if the child did not do an exec and ended up writing to a lot of the pages it was sharing with the parent. Copy-on-write reduces the predictability of the memory access. A single memory access can result in a trap, the copying of a full page, and the restart of the process.
5. (a) Will not have an effect: the paging disk is bottlenecked, not the CPU. A faster CPU will result in even lower CPU utilisation.
 - (b) Unclear, but probably not. The effect of adding a bigger paging disk will be to increase the number of virtual pages which can be backed to disk, which may lead the OS to try and put more processes in virtual memory. Then the situation would be likely to get worse!
 - (c) This will probably make the situation worse, more processes will intensify thrashing.
 - (d) This will probably improve performance, there will be less competition for physical page frames.
 - (e) This will improve performance, since each process will now have access to more page frames (i.e. it can accommodate more of its working set).
 - (f) This will improve performance: the disk is evidently bottlenecked, so a faster disk will ease the contention.
 - (g) Unclear, this could actually decrease performance if bringing in a whole set of pages from disk at one time makes thrashing worse.
 - (h) Unclear. A larger page size could make servicing page faults more efficient (fewer pages to load from disk, so reduced access times) if the applications exhibit high spatial locality, but it could also intensify thrashing and merely add to the volume of data to transfer (larger pages).
6. This question can be answered in two ways, depending on whether you assume the matrix is stored in row-major order ($A[1, 1], A[1, 2], \dots$) or column-major order ($A[1, 1], A[2, 1], \dots$). We'll assume row-major order; column-major order will mean that the answers to the questions are exchanged. Row-major order means that each page contains two consecutive rows of the matrix: page 1 has rows 0, 1, page 2 has rows 2, 3, and so on.
 - (a) The matrix is accessed column-by-column, and each page has two entries from a particular column. So accessing all the entries for one column will generate 50 page faults (since they're on 50 different pages). LRU replacement means that we will never have

a page in memory when we do a repeated access (the pages in memory will be (0, 1), then (1, 2), ..., (98, 99), (99, 0), and so on), so there will be $50 \times 100 = 5000$ page faults.

- (b) The matrix is accessed row-by-row, and each page contains two rows, we get a page fault every two rows, i.e. 50 page faults in all.

7. Here are the results (number of page faults) in a table:

<i>page frames</i>	<i>LRU</i>	<i>FIFO</i>	<i>optimal</i>
1	20	20	20
2	18	18	15
3	15	16	11
4	10	14	8
5	8	10	7
6	7	10	7
7	7	7	7

8. (a) Unless the code for the shared library is compiled correctly, it will end up with embedded virtual addresses, which will not be valid if the library is in the “wrong location”. Unless all processes map the shared library to the same set of virtual pages, the shared library will not work.
- (b) Rather than compiling the code to use absolute addressing, the shared library should use relative addressing (e.g. relative to the program counter), so that the code will be relocatable – it will work at whatever virtual address it is mapped to.
9. (a) Segments are potentially of different sizes, and a segment must be loaded into a free contiguous memory region of sufficient size. When selecting segments to evict from physical memory, there are at least two competing constraints to consider. First, as in page replacement algorithms, we would like to throw out segments that will not be needed again for the longest time into the future. On the other hand, we would like to throw out segments that decrease fragmentation. For example, there may be a small fragment in the middle of two holes, which could become one contiguous piece of memory. In general, we would prefer to evict smaller pages, so as not to incur an expensive writeback. But, we would also like to evict as few pages as possible, in order to reduce the segment fault frequency. We also have the option of relocating segments in memory for purposes of defragmentation. This is advantageous when a few moves can defragment large portions of memory. However, as memory utilization increases, the defragmentation problem becomes more difficult to solve with segment relocations, and paging is more reasonable.
- (b) The closest equivalent to LRU paging in the domain of segments is to evict least recently used segments (possibly non-contiguous, and possibly more than one due to the varying segment sizes) until the total amount of (unfragmented) free memory is sufficient to store the segment that we would like to load. We then defragment the physical memory by relocating segments (a potentially expensive operation!), and load the segment into the single remaining memory “hole”.
- (c) By paging the segments we allow segments to occupy non-contiguous pages of memory. Thus, the problem of “paging” in a segment reduces to the problem of paging in a set of pages (belonging to that segment), which we can perform using our earlier paging techniques. We usually page in the segment pages on demand, since there is

no need to load the entire segment address at once. And, since pages are of fixed size, we have no external fragmentation in the physical memory. However, segments do not necessarily end on page boundaries, so there will be some wasted space within the pages – internal fragmentation.

10. (a) The virtual pages of a process usually represent the lion's share of the memory that it consumes. However, there are other process-related data structures that can add up when there are hundreds of processes in the system. This includes page tables, open file tables, buffers, process accounting information and even the PCB. All of these can be written to disk (or discarded in the case of read buffers) to allow more memory for other active processes. All that must remain in memory is some reference to the information on disk, so that it can be reconstituted in memory at a later time to continue execution.
- (b) One could take all the data structures for the process and persist them to disk, perhaps leaving some scheduling information about when the process should be revived. Without the scheduling information, the operating system would need to continuously check the disk to determine whether to revive the process or, worse yet, to revive the process only to allow it to yield again. Persisting the process information to disk is only beneficial if the process can remain there for a long time.

Most operating systems do *not* take this approach (of many long-lived idle processes). They rely, instead, on a single timer process (known as *cron*d in Unix) that will execute others at specific times. Those processes run quickly to perform their maintenance functions and quit, thus consuming no memory at all in the interim. It is a lot simpler to implement an application like *cron*d than to add the above-mentioned complexity into the OS kernel, so the payoff is not worthwhile.