Solved Exercises 4

1. Solve Problem **8.19** from the textbook.

Periodically reallocate some of the memory pages used by the program with the lowest page transfer activity to the program with the highest activity. This should not be done too frequently. Otherwise, the time cost in doing the reallocation would outweigh its potential performance benefit. A reasonable strategy is to reallocate pages only when the difference in the page transfer rate exceeds some threshold.

2. Solve Problem **8.20** from the textbook.

Continuing the execution of an instruction interrupted by a page fault requires saving the entire state of the processor, which includes saving all registers that may have been affected by the instruction as well as the control information that indicates how far its execution has progressed. The alternative of re-executing the instruction from the beginning is simpler. It requires that partial results, if any, generated during instruction execution be stored in temporary locations, so that they can be safely discarded if the execution of that instruction is interrupted. This is particularly easy to do in a RISC-style processor, as the results of the instruction are recorded only in the very last execution step, as described in Chapter 5, and there are no side effects.

3. Solve Problem **8.21** from the textbook.

A page fault can occur part-way during the execution of an instruction that has an operand on a different page. When this happens, the operating system has to suspend execution of the program containing this instruction and start a DMA operation to transfer the required page. (It may also perform a context switch — see Chapter 4.) It resumes execution of the suspended program after the required page has been read from the disk. The difficulty that arises in this case is that the instruction that caused the page fault has been partially executed.

There are two ways to proceed. When execution of the program is suspended, the processor may discard the instruction that caused the page fault together with any partial results that may have been produced. It adjusts the program counter, so that the discarded instruction is executed again when the suspended program resumes execution. Alternatively, the full state of the processor may be saved at the time the page fault occurs, including how many execution steps have been completed for the instruction that caused the page fault and the contents of any temporary registers it is using. With this information, the processor can later resume execution of the instruction at the point of suspension.

4. Consider the **Good** and **Bad** code examples shown on **Slide 55** of the **"Memory"** lecture notes, where one **4KB** page was allocated for the array **int a[1024][1024]**. What would be the page fault rate in each example, if we allocate 512 **4KB** pages for the array? What would be the page fault rate in each example, if we allocate only one **4KB** page for the array, but let **M** = **N** = **512**?

If we allocate 512 **4KB** pages for the **int a[1024][1024]** array, the **Good** example will have 1 page fault per 1024 row element accesses (**p ≈ 0.1%**), and the **Bad** example will have 1 page fault per 1 row element access (**p = 100%**). In other words, increasing the number of allocated **4KB** pages from 1 to 512 provides no benefit for this particular application code.

If we allocate one **4KB** page for the **int a[512][512]** array (i.e., one page holds 2 rows), then the **Good** example will have 1 page fault per 512+512 row element accesses (**p ≈ 0.1%**), and the **Bad** example will have 1 page fault per 1+1 row element accesses (**p = 50%**).

5. Consider a C code fragment transposing **`float original[M][N]`** into **`float transposed[N][M]`**, where **M = 512** and **N = 256**:

```
for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++) {
            transposed[j][i] = original[i][j];
      }
}
```

Storing both **`original`** and **`transposed`** requires 512*256*4 + 256*512*4 = **1MB** of memory (each **`float`** array element is a 4-byte number). If the cache (assume fully associative) is smaller than 1MB, the above code example will result in many misses, considerably slowing down program execution. Alternatively, one can perform blocked transposition, i.e., partition the matrices into smaller blocks and transpose matrix elements block by block. This may significantly outperform the original transposition code, if block data can fit into the cache.

Rewrite the above code using **32×32** blocks for matrix transposition. Assuming that such blocking results in the best performance, what can you say about the size of the cache?

Matrix transposition with **m×m** square blocks (**m = 32**):

```
for (p = 0; p < M/m; p++) {
      for (q = 0; q < N/m; q++) {
            for (i = p*m; i < (p+1)*m; i++) {
                  for (j = q*m; j < (q+1)*m; j++) {
                        transposed[j][i] = original[i][j];
                  }
            }

      }
}
```

Storing a **32x32** block of 32-bit numbers requires 32*32*4=**4KB** of memory. The cache size should be at least 4KB+4KB=**8KB** to hold block numbers from both matrices.