Operating Systems

Homework Assignment# 3

[30 points]

[Note: Some problems *may* contain less information than needed to solve it. In that case, you need to make some assumptions to continue the solution. In other problems more information may be given than what is needed to solve the problem. You can neglect that extra information.]

---

1. A 2D long integer array, A[64][64], is stored in memory in row-major order (i,e, A[0][0] is followed by A[0][1], then A[0][2], … etc). Assume the whole physical memory can hold only four pages and that the size of each page is 512 bytes. Remember that A[x][y] where x is the row number and y is the column number. Let's assume the page replacement policy used by the OS is first-in-first-out.

Piece A	Piece B

for (int j = 0; j < 64; j++)
  for (int i = 0; i < 64; i++)
  A[i][j] = 0;
	for (int i = 0; i < 64; i++)
	  for (int j = 0; j < 64; j++)
	  A[i][j] = 0;

a) [4] Which of the two pieces, piece A and piece B, of code results in more page faults? Justify your choice.
[1] Piece A results in more page faults.
[6] This is because it does not access the array sequentially. The first elements it accessed is A[0][0], the second elements is A[1][0] which is stored in memory after A[0][1], A[0][2], … A[0][63]. This makes it in a different page than A[0][0].

b) [2] How many elements of array A can each virtual page hold? Show your calculations to get full credit.
64 elements
Each element is long int → 8 bytes. A page takes 512 bytes.
So, a page can hold 512/8 = 64 elements

c) [2] How many elements of array A can each physical page (sometimes called page slot) hold? Show your calculations to get full credit.
The page slot is the same size as the virtual page. So, same answer as part b) above.

d) [2] How many pages faults does piece A of code cause? Show your calculations to get full credit.
[1] 64x64 = 4096 page faults
[1] The array is stored as row based. And we saw, in b) above that a page can hold 64 elements. The code in piece A access the array column wise. That is, each two consecutive accesses are a full row apart. Since each row has 64 elements and since each page holds 64 elements → each two consecutive accesses are a page apart. That is, for example, A[0][0] and A[1][0] are a page apart in memory. This means each access brings a full page to memory and hence causes a page fault.

e) [2] How many pages faults does piece B of code cause? Show your calculations to get full credit.
[1] 64 page faults

[1] The array is stored as row based. And we saw, in b) above that a page can hold 64 elements. This means we get a page fault each time we access a page for the first time. Then, we get 63 page hits. So, 1 page fault every 64 accesses. We have 64x64 accesses → 64 page faults.

---

2. Suppose we have a system with four CPUs. Each CPU can execute one thread at a time. Two processes start. No other processes exist in the system. Neglect the OS system overhead and its usages of CPUs.

a) [4] What is the minimum number of CPUs that these two processes can use, assuming none of them will be blocked or exits? Explain.
[1] Two CPUs.
[3] The two processes can be running at the same time. To get the minimum, let's assume each process consists of one thread only (or use several user-level threads). This means they can both use two CPUs.

b) [4] What is the maximum number of CPUs that these two processes can use? Explain.
[1] All four CPUs can be used.
[3] This can happen when the sum of the threads in the two processes >= number of CPUs and all the threads are kernel-level threads.

---

3. [10] Suppose you wrote a multithreaded program where you spawn x threads. During execution time we may see *less than x* threads, from that process, executing in parallel. That is, as a programmer, you want, for example, five threads to be executing in parallel. But, during execution, you see that there are less than five threads really executing. State *five* scenarios that may cause this to happen.

- [2] There are less cores in the hardware than x. So, the OS cannot schedule x threads at the same time. And, even if there are x cores, this process may not be running alone. Threads from other processes, and the OS itself, may be taking some of the cores.
- [2] Those x threads are user-level threads. Hence, the OS treats them as one thread and schedules them on one core.
- [2] Some of the x threads are blocked, for I/O or page fault or any other reason, so the OS removes those blocked threads from the core(s) and schedule other threads/processes.
- [2] The user-level threads to kernel-level threads may be M:N where N < M. Hence the x threads will be mapped to less than x kernel-level threads.
- [2] One, or more threads exited.

[If you mentioned several reasons like: page fault and a thread is blocked, I/O request and the thread is blocked, etc as different reasons, it will be taken as correct]
