

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

- 1.1 Responsibilities of the OS include loading programs, handling services, multiplexing resources, and combining programs together for efficiency.
- 1.2 The purpose of supervisor mode is to isolate certain instructions and routines from user programs.
- 1.3 User programs call into OS routines using system calls.
- 1.4 A thread is another name for a process.
- 1.5 SIMD is a form of instruction-level parallelism.
- 1.6 SIMD is ideal for flow-control heavy tasks (i.e. tasks with many branches/if statements).
- 1.7 Intel's SIMD intrinsic instructions invoke large registers available on the architecture in order to perform one operation on multiple values at once.

2 Forking

- 2.1 One of the many responsibilities of the OS is to load new programs, and in order to do this it creates a new process and loads in the program to execute. In Linux, the system call to create a new process is `fork()`. `fork()` creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process. In the parent process, `fork()` returns the process ID of the child or -1 if the fork has failed. In the child process, it returns 0.

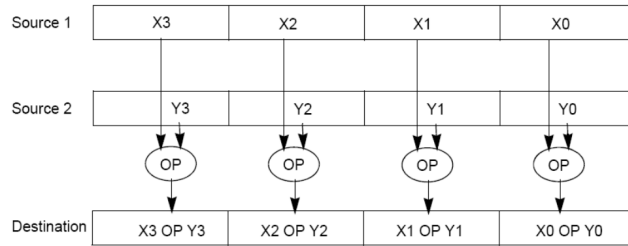
Use this information to complete the code block below, which creates a child process to change the value of `y` while the parent process changes the value of `x`. Assume any call to `fork()` is successful.

```
int x = 10;
int y = 0;
int pid = _____;
if(_____){
    y++
}
else{
    x--;
}
```

- 2.2 After the code segment completes, what will be the values of `x` and `y` for the parent?
- 2.3 After the code segment completes, what will be the values of `x` and `y` for the child?

3 Data-Level Parallelism

The idea central to data level parallelism is vectorized calculation: applying operations to multiple items (which are part of a single vector) at the same time.



Some machines with x86 architectures have special, wider registers, that can hold 128, 256, or even 512 bits. Intel intrinsics (Intel proprietary technology) allow us to use these wider registers to harness the power of DLP in C code.

Below is a small selection of the available Intel intrinsic instructions. All of them perform operations using 128-bit registers. The type `__m128i` is used when these registers hold 4 ints, 8 shorts or 16 chars; `__m128d` is used for 2 double precision floats, and `__m128` is used for 4 single precision floats. Where you see “epiXX”, epi stands for **e**xtended **p**acked **i**nteger, and XX is the number of bits in the integer. “epi32” for example indicates that we are treating the 128-bit register as a pack of 4 32-bit integers.

- `__m128i _mm_set1_epi32(int i)`:
Set the four signed 32-bit integers within the vector to `i`.
- `__m128i _mm_loadu_si128(__m128i *p)`:
Load the 4 successive ints pointed to by `p` into a 128-bit vector.
- `__m128i _mm_mullo_epi32(__m128i a, __m128i b)`:
Return vector $(a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3)$.
- `__m128i _mm_add_epi32(__m128i a, __m128i b)`:
Return vector $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- `void _mm_storeu_si128(__m128i *p, __m128i a)`:
Store 128-bit vector `a` at pointer `p`.
- `__m128i _mm_and_si128(__m128i a, __m128i b)`:
Perform a bitwise AND of 128 bits in `a` and `b`, and return the result.
- `__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)`:
The `i`th element of the return vector will be set to `0xFFFFFFFF` if the `i`th elements of `a` and `b` are equal, otherwise it’ll be set to 0.

Notice: On this worksheet, we are using the *unaligned* versions of the commands that interface with memory (i.e. `storeu/loadu` vs. `store/load`). This is because the `store/load` commands require that the address we are loading at is aligned at some byte boundary (and not necessarily just word-aligned), whereas the unaligned versions have no such requirements. For instance, `_mm_store_si128` needs the address to be aligned on a 16-byte boundary (i.e. is a multiple of 16). There is extra work that needs to be done to achieve these alignment requirements, so for this class, we just use the unaligned variants.

3.1 You have an array of 32-bit integers and a 128-bit vector as follows:

```
1 int arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
2 __m128i vector = _mm_loadu_si128((__m128i *) arr);
```

For each of the following tasks, fill in the correct arguments for each SIMD instruction, and where necessary, fill in the appropriate SIMD function. Assume they happen independently, i.e. the results of Part (a) do not at all affect Part (b).

(a) Multiply vector by itself, and set vector to the result.

```
1 vector = _____(_____, _____);
```

(b) Add 1 to each of the first 4 elements of the arr, resulting in arr = {2, 3, 4, 5, 5, 6, 7, 8}

```
1 __m128i vector_ones = _mm_set1_epi32(_____);
2 __m128i result = _mm_add_epi32(_____, _____);
3 _mm_storeu_si128(_____, _____);
```

(c) Add the second half of the array to the first half of the array, resulting in arr = {1 + 5, 2 + 6, 3 + 7, 4 + 8, 5, 6, 7, 8} = {6, 8, 10, 12, 5, 6, 7, 8}

```
1 __m128i result = _mm_add_epi32(_mm_loadu_si128(_____), _____);
2 _mm_storeu_si128(_____, _____);
```

(d) Set every element of the array that is not equal to 5 to 0, resulting in arr = {0, 0, 0, 0, 5, 0, 0, 0}. Remember that the first half of the array has already been loaded into vector.

```
1 __m128i fives = _____(_____);
2 __m128i mask = _____(_____, _____);
3 __m128i result = _____(_____, _____);
4 _mm_storeu_si128(_____, _____);
5 vector = _mm_loadu_si128(_____);
6 mask = _____(_____, _____);
7 result = _____(_____, _____);
8 _mm_storeu_si128(_____, _____);
```

3.2 SIMD-ize the following function, which returns the product of all of the elements in an array. Things to think about: When iterating through a loop and grabbing elements 4 at a time, how should we update our index for the next iteration? What if our array has a length that isn't a multiple of 4? Can we always SIMD-ize an entire array? What can we do to handle this tail case?

```
static int product_naive(int n, int *a) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product *= a[i];
    }
    return product;
}
```

```

static int product_vectorized(int n, int *a) {
    int result[4];
    __m128i prod_v = _____;
    for (int i = 0; i < _____; i += ___) { // Vectorized loop
        prod_v = _____;
    }
    __mm_storeu_si128(_____, _____);
    for (int i = _____; i < _____; i++) { // Handle tail case
        result[0] *= _____;
    }
    return _____;
}

```

4 Thread-Level Parallelism

As powerful as data level parallelization is, it can be quite inflexible, as not all applications have data that can be vectorized. Multithreading, or running a single piece of software on multiple hardware threads, is much more powerful and versatile.

OpenMP provides an easy interface for using multithreading within C programs. Some examples of OpenMP directives:

- The `parallel` directive indicates that each thread should run a copy of the code within the block. If a for loop is put within the block, **every** thread will run every iteration of the for loop.

```
#pragma omp parallel
{
    ...
}
```

NOTE: The opening curly brace needs to be on a newline or **else** there will be a compile-time error!

- The `parallel for` directive will split up iterations of a for loop over various threads. Every thread will run **different** iterations of the for loop. The following two code snippets are equivalent.

<pre>#pragma omp parallel for for (int i = 0; i < n; i++) { ... }</pre>	<pre>#pragma omp parallel { #pragma omp for for (int i =0; i < n; i++) { ... } }</pre>
--	---

There are two functions you can call that may be useful to you:

- `int omp_get_thread_num()` will return the number of the thread executing the code
- `int omp_get_num_threads()` will return the number of total hardware threads executing the code

4.1 For each question below, state and justify whether the program is **sometimes incorrect**, **always incorrect**, **slower than serial**, **faster than serial**, or **none of the above**. Assume the default number of threads is greater than 1. Assume no thread will complete before another thread starts executing. Assume `arr` is an `int[]` of length `n`.

(a) // Set element `i` of `arr` to `i`

```
#pragma omp parallel
{
    for (int i = 0; i < n; i++)
        arr[i] = i;
}
```

(b) // Set `arr` to be an array of Fibonacci numbers.

```
arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i++)
    arr[i] = arr[i-1] + arr[i - 2];
```

(c) // Set all elements in `arr` to 0;

```
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    arr[i] = 0;
```

(d) // Set element `i` of `arr` to `i`;

```
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    *arr = i;
    arr++;
```

4.2 What potential issue can arise from this code?

```
1 // Decrements element i of arr. n is a multiple of omp_get_num_threads()
2 #pragma omp parallel
3 {
4     int threadCount = omp_get_num_threads();
5     int myThread = omp_get_thread_num();
6     for (int i = 0; i < n; i++) {
7         if (i % threadCount == myThread) arr[i] -= 1;
8     }
9 }
```

