CS61C Summer 2025 — TLP, Virtual Memory — Discussion 12

1 OpenMProgramming

Consider the following C function:

```
#define ARRAY_LEN 1000

void mystery(int32_t *A, int32_t *B, int32_t *C) {
  for (int i = 0; i < ARRAY_LEN; i += 1) {
    C[i] = A[i] - B[i];
  }
}
```

1.1 Manually rewrite the loop to split the work equally across N different threads.

```
#define ARRAY_LEN 1000

void mystery(int32_t *A, int32_t *B, int32_t *C) {
  #pragma omp parallel
  {
    int N = OMP_NUM_THREADS;
    int tid = omp_get_thread_num();

    for (int i = ____________; i < ______________; i += _____________) {
      C[i] = A[i] - B[i];
    }
  }
}
```

1.2 Now, split the work across `N` threads using a `#pragma` directive:

```
#define ARRAY_LEN 1000

void mystery(int32_t *A, int32_t *B, int32_t *C) {

  ----------------------------------------
  for (int i = 0; i < ARRAY_LEN; i += 1) {
    C[i] = A[i] - B[i];
  }
}
```

1.3 Instead of saving the product to an array `C`, we now want to XOR the subtraction of all the elements of `A` and `B`.

```
#define ARRAY_LEN 1000

int mystery(int32_t *A, int32_t *B) {
  int result = 0;
  #pragma omp parallel for
  for (int i = 0; i < ARRAY_LEN; i += 1) {
    result ^= A[i] - B[i];
  }
  return result;
}
```

What is the issue with the above implementation and how can we fix it?

1.4 Solve the problem above in two different methods using OpenMP:

(a)
```
int mystery(int32_t *A, int32_t *B) {
  int result = 0;
  #pragma omp parallel for
  for (int i = 0; i < ARRAY_LEN; i += 1) {

    _____________________________________
    result ^= A[i] - B[i];
  }
  return result;
}
```

(b)
```
int mystery(int32_t *A, int32_t *B) {
  int result = 0;

  _____________________________________
  for (int i = 0; i < ARRAY_LEN; i += 1) {
    result ^= A[i] - B[i];
  }
  return result;
}
```

$\boxed{1.5}$ Assume we run the above `mystery` function with 8 threads. The parallel portion accounts for 80% of the program and is 8x as fast as the naive implementation. Use Amdahl's Law to calculate the speedup of the full program where

$$\text{Speedup} = \frac{1}{\left(1 - \text{frac}_{\text{optimized}}\right) + \frac{\text{frac}_{\text{optimized}}}{\text{factor}_{\text{improvement}}}}$$

$\boxed{1.6}$ What is the maximum speedup we can achieve if we use unlimited threads in the parallel section for an infinite performance increase? Assume the parallel portion still accounts for 80% of our program.

$\boxed{1.7}$ What does the above result tell you about using parallelism to optimize programs?

2 Virtual Memory Potpourri

$\boxed{2.1}$ For the following address spaces, how many bits are in the Virtual Page Number (VPN), Physical Page Number (PPN), and Page Offset?

(a) A system with 16 MiB of virtual memory, 1 MiB of physical memory, 1024 B pages

VPN:

PPN:

Offset:

(Q1.2) Number of PTEs:

(Q1.3) Page Table Size:

(b) A system with 512 MiB of virtual memory, 32 KiB of physical memory, 512 B pages

VPN:

PPN:

Offset:

(Q1.2) Number of PTEs:

(Q1.3) Page Table Size:

(c) A system with 4 GiB of virtual memory, 1 GiB of physical memory, 4 KiB pages

VPN:

PPN:

Offset:

(Q1.2) Number of PTEs:

(Q1.3) Page Table Size:

$\boxed{2.2}$ For the above systems, how many entries are in each the page table?

$\boxed{2.3}$ For the above systems, calculate the size of the page table (in bytes) in memory given each Page Table Entry is 4 bytes.

$\boxed{2.4}$ If a Page Table's size is 2^{30} Bytes and each page is 4 KiB, how many physical pages are needed to store the page table?

2.5 Given a system with 12-bit VPNs, 8-bit PPNs, and 8-bit offsets:

(a) What is the Virtual Page Number (VPN) and the page offset of the *virtual address* `0x51B38`?

VPN:

Offset:

(b) What is the Physical Page Number (PPN) and the page offset of the *physical address* `0xB1DC`?

PPN:

Offset:

2.6 What are three specific benefits of using virtual memory?

3 Page Table Walk

Assume we have 16-bit VPNs, 12-bit PPNs, 8-bit page offsets, and 32-bit page table entries (PTEs). The first six entries of the page table are shown below.

Page Table	**Valid?**	**Dirty?**	**PPN**
`0xB61C 0483`			
`0xFB83 A61C`			
`0x8483 3F01`			
`0x7ABC 4103`			
`0xC012 F7CB`			
`0x15DA C203`			
...			

...where each page table entry (PTE) is formatted as:

1 Valid Bit	1 Dirty Bit	18 Status Bits	12 PPN Bits

3.1 Of the first 6 entries in the TLB, fill out the above table for each entry. List whether the PTE is a valid mapping. If so, list translate its corresponding physical page and if the page is clean/dirty.

3.2 For each of the following virtual addresses, answer whether accessing will result in a 1) Page Table Hit or 2) Page Fault, and translate to its corresponding physical address. Each access occurs independently, not sequentially. The next available free page has PPN 0x42D.

(a) `0x000429`

(b) `0x00018D`

(c) `0x000345`

3.3 Recall that the Page Table Base Register (PTBR) stores the physical address of our page table. For this program, the PTBR = `0x10000`. What is the physical address of the page table entry `0xC012F7CB`?

3.4 We want to reserve the first 10 pages of physical memory to be read-only. How can we modify our page table to accomplish this?

4 Page Table with TLB

4.1 A processor has 16-bit addresses, 256 byte pages, and an 8-entry fully associative TLB with LRU replacement (the LRU field is 3 bits and encodes the order in which pages were accessed, 0 being the most recent). The TLB for the current process is the initial state given below, and we have three free physical pages. Assume that all current page table entries are in the initial TLB. Write out the physical addresses of each location accessed and fill in the final state of the TLB according to the following access pattern. **Free Physical Pages**: `0x17`, `0x18`, `0x19`

Initial TLB

VPN	PPN	Valid	Dirty	LRU
`0x01`	`0x11`	1	1	0
`0x00`	`0x00`	0	0	7
`0x10`	`0x13`	1	1	1
`0x20`	`0x12`	1	0	5
`0x00`	`0x00`	0	0	6
`0x11`	`0x14`	1	0	4
`0xac`	`0x15`	1	1	2
`0xff`	`0xff`	1	0	3

Final TLB

VPN	PPN	Valid	Dirty	LRU

Access Pattern:

1. `0x11f0` (Read)
2. `0x1301` (Write)
3. `0x20ae` (Write)
4. `0x2332` (Write))
5. `0x20ff` (Read)
6. `0x3415` (Write)