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 = tid; i < ARRAY_LEN; i += N) {
      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) {
  #pragma omp parallel for
  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?

There is a race condition for the `result` variable.

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) {
    #pragma omp critical
    result ^= A[i] - B[i];
  }
  return result;
}
```

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

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

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

$$\begin{aligned}\text{Speedup} &= \frac{1}{(1-\text{frac}_{\text{optimized}}) + \frac{\text{frac}_{\text{optimized}}}{\text{factor}_{\text{improvement}}}} \\ &= \frac{1}{(1-0.8) + \frac{0.8}{8}} \\ &= \frac{1}{0.2 + 0.1} \\ &= 3.333\text{x speedup!}\end{aligned}$$

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.

$$\begin{aligned}\text{Speedup} &= \frac{1}{(1-\text{frac}_{\text{optimized}}) + \frac{\text{frac}_{\text{optimized}}}{\text{factor}_{\text{improvement}}}} \\ &= \frac{1}{(1-0.8) + \frac{0.8}{9999999...}} \\ &= \frac{1}{0.2} \\ &= 5\text{x maximum speedup!}\end{aligned}$$

1.7 What does the above result tell you about using parallelism to optimize programs?

Programs can only be as fast as their serial portion.

2 Virtual Memory Potpourri

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: 14 bits

PPN: 10 bits

Offset: 10 bits

(Q1.2) Number of PTEs: 2^{14} PTEs

(Q1.3) Page Table Size: 2^{16} Bytes

16 MiB of virtual memory means we have 16 MiB VM / 1 KiB pages = $16 * 2^{20}/2^{10} = 2^{14}$ virtual pages which means our VPN = 14 bits.

1 MiB of physical memory means we have 1 MiB VM / 1 KiB pages = $2^{20}/2^{10} = 2^{10}$ physical pages which means our PPN = 10 bits.

1 KiB pages means each page is 2^{10} bytes so we have have an offset of 10 bits in our address.

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

VPN: 20 bits

PPN: 6 bits

Offset: 9 bits

(Q1.2) Number of PTEs: 2^{20} PTEs

(Q1.3) Page Table Size: 2^{22} Bytes

512 MiB of virtual memory means we have 512 MiB VM / 512 B pages = $512 * 2^{20}/512 = 2^{20}$ virtual pages which means our VPN = 20 bits.

32 KiB of physical memory means we have 32 KiB VM / 512 B pages = $32 * 2^{10}/2^9 = 2^6$ physical pages which means our PPN = 6 bits.

512 B pages means each page is 2^9 bytes so we have have an offset of 9 bits in our address.

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

VPN: 20 bits

PPN: 18 bits

Offset: 12 bits

(Q1.2) Number of PTEs: 2^{20} PTEs

(Q1.3) Page Table Size: 2^{22} Bytes

4 GiB of virtual memory means we have 4 GiB VM / 4 KiB pages = $4 * 2^{30}/4 * 2^{10} = 2^{20}$ virtual pages which means our VPN = 20 bits.

1 GiB of physical memory means we have 1 GiB VM / 4 KiB pages = $2^{30}/4 * 2^{10} = 2^{18}$ physical pages which means our PPN = 18 bits.

4 KiB pages means each page is $4 * 2^{10}$ bytes so we have have an offset of 12 bits in our address.

2.2 For the above systems, how many entries are in each the page table?

Since the VPN is the offset in the page table, there must be an entry for each VPN. Thus, Number of PTEs = $2^{\text{VPN Bits}}$.

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.

Page Table Size = Number of Entries × Size of Entry = Number of Entries × 2^2 Bytes

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?

If each page table is 2^{30} Bytes and each page can store $4 * 2^{10}$, then it takes 2^{30} Bytes $/4 * 2^{10} = 2^{18}$ physical pages 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: `0x51B`

Offset: `0x38`

The VPN is the upper-most 12 bits of the address while the offset is the lower-most 8 bits of the address.

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

PPN: `0xB1`

Offset: `DC`

The PPN is the upper-most 8 bits of the address while the offset is the lower-most 8 bits of the address.

2.6 What are three specific benefits of using virtual memory?

- Illusion of access to entire address space (bridges memory and disk in memory hierarchy).
- Avoids memory address conflict between programs by simulating a separate full address space for each process, so that the linker/loader don't need to know about other programs.
- Enforces protection between processes and even within a process (e.g. read-only pages set up by the OS).

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`	Valid	Clean	`0x483`
`0xFB83 A61C`	Valid	Dirty	`0x61C`
`0x8483 3F01`	Valid	Clean	`0xF01`
`0x7ABC 4103`	Invalid	N/A	N/A
`0xC012 F7CB`	Valid	Dirty	`0x7CB`
`0x15DA C203`	Invalid	N/A	N/A
...			

...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.

For each of the Page Table Entries, we can rewrite the hex as binary and decode according to the PTE format. For example, the first entry `0xB61C 0483` is `0b1011_0110...0100_1000_0011`.

- The first MSB is defined as the valid bit which is `1` so our entry is valid.
- The second MSB is `0` which means our entry is not dirty.
- The last 12 bits are for the PPN mapping which (if PTE is valid) corresponds to the physical page allocated to that virtual page. For this mapping, the `PPN= 0x483`.

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`

Answer: Page Table Hit, `PA = 0x7CB29`

Our virtual address is split up into `[VPN | Offset]` bits. We have 8-bit offsets so our `offset =` 0x29 `and our` VPN = `0x0004` which corresponds to entry `0xC012F7CB`. Because this is a valid mapping, our `PPN = 0x7CB` which means our physical address is `0x7CB29`.

(b) `0x00018D`

Answer: Page Table Hit, `PA = 0x61C8D`

Following the above procedure: `offset =` 0x8D `and` VPN = `0x0001` which corresponds to entry `0xFB83A61C`. Because this is a valid mapping, our `PPN = 0x61C` which means our physical address is `0x61C8D`.

(c) `0x000345`

Answer: Page Fault, `PA = 0x42D45`

Following the above procedure: `offset =` 0x45 `and` VPN = `0x0003` which corresponds to entry `0x7ABC4103`. This is an invalid mapping which means we have not yet mapped this VPN to a physical page. The next available free page has `PPN = 0x42D` which means our physical address is `0x42D45`. Although not shown, this PTE will be updated with this new mapping.

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`?

`0x10010`. From the PTBR, we know that the first PTE is at `0x10000`. Since each PTE is 4 bytes and the required entry is at the 4th index, `Physical address of the entry = PTBR + offset = 0x10000 + (4 PTEs) = 0x10000 + 16 bytes = 0x10010`.

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?

We can add a Read-Only status bit to the metadata which, if true, disallows writing to a specific physical page.

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
0x01	0x11	1	1	5
0x13	0x17	1	1	3
0x10	0x13	1	1	6
0x20	0x12	1	1	1
0x23	0x18	1	1	2
0x11	0x14	1	0	4
0xac	0x15	1	1	7
0x34	0x19	1	1	0

Access Pattern:

1. `0x11f0` (Read)

   Hit. PA: `0x14f0`, LRUs: `1, 7, 2, 5, 6, 0, 3, 4`

2. `0x1301` (Write)

   Miss. Map VPN `0x13` to next available free page which is PPN `0x17`. PA: `0x1701` and set valid and dirty bits. LRUs: `2, 0, 3, 6, 7, 1, 4, 5`

3. `0x20ae` (Write)

   Hit. Set dirty bit because of the write. PA: `0x12ae`. LRUs: `3, 1, 4, 0, 7, 2, 5, 6`

4. `0x2332` (Write))

   Miss. Map VPN `0x23` to next available free page which is PPN `0x18` and set valid and dirty. PA: `0x1832`. LRUs: `4, 2, 5, 1, 0, 3, 6, 7`

5. `0x20ff` (Read)

   Hit. PA: `0x12ff`. LRUs: `4, 2, 5, 0, 1, 3, 6, 7`

6. `0x3415` (Write)

   Miss and replace last entry. Map VPN `0x34` to the last free page which is `0x19` and set valid and dirty bit. PA: `0x1915`. LRUs `5, 3, 6, 1, 2, 4, 7, 0`