Homework 3

CS 250: Computer Architecture

Kian Kasad

Problem 1

(5 points) Describe three different ways "caching" as a general concept is used in hardware and in software.

Caching is used…

1. to speed up access to physical memory by storing recently/frequently used data in a faster storage media located on the CPU die called the memory cache;
2. to speed up access to frequently-used RAM pages by storing them in memory rather than in the swap space (on disk);
3. to speed up web browsing by storing frequently-accessed pages or assets locally so they don't need to be re-fetched frequently.

Problem 2

(5 points) When allocating memory initialized to zero, explain…

(a) how this ordeal is accelerated by using virtual memory;
(b) how the page table is prepared and what are the values of the page bits initially;
(c) what happens during the page fault.

(a) All zero-initialized pages in virtual memory map to the same zero-initialized page in physical memory. This way, when a new zero-initialized page is allocated, a page table entry needs to be allocated to point to the physical zero page, but a new page does not need to be cleared until the page is modified (copy-on-write).

(b) The page bits are all zero initially to represent a page filled with zeroes, except for the read permission bit, which is set.

(c) During the page fault, the operating system allocates a new page in physical memory and clears it, i.e. sets all bits to 0. It then replaces the entry in the page table with the number of this new page, makes it modifiable, and re-executes the instruction which triggered this page fault.

Problem 3

(5 points) When `fork(2)` is called to create a new process, explain…

(a) how this ordeal is accelerated by using virtual memory;
(b) how the page table is prepared and what are the values of the page bits initially;
(c) what happens during the page fault.

(a) Spawning a child process is accelerated by using virtual memory because the entire virtual memory space of the parent process does not need to be copied. For a process using lots of memory, this could be very expensive. Instead, only the page table needs to be copied, and the pages themselves can be copied as they are modified (copy-on-write).

(b) The page table initially has the same values as the parent process's page table, except all pages are marked as non-modifiable so the operating system can implement copy-on-write behavior for the forked process.

(c) When a newly-forked process modifies one of its pages, a page fault is triggered because the page is marked non-modifiable when the new process is created. The operating system will then handle this page fault by:
   i. creating a new page which is a copy of the one modified;
   ii. replacing the page requested with this new page in the process's page table;
   iii. marking the new page as modifiable;
   iv. retrying the instruction which modified this page so that it can now proceed with the copied-on-demand page.

Problem 4

(5 points) Explain how the following operations may benefit or may not benefit from memory cache:
(a) Accessing local variables.
(b) Sorting using bubble sort.
(c) Sorting using heap sort.

(a) Accessing local variables will benefit from caching due to their high locality of reference. Local variables are located in a fixed place in memory and are typically accessed frequently within a function. Thus they are likely to be cached on first use and are likely to stay in the cache due to their frequent use. This makes reading/writing their values fast, because the underlying memory does not need to be accessed.

(b) Sorting using bubble sort will benefit from caching because it uses many sequential operations. It accesses each element of the array being sorted in order, resulting in high locality of reference. Thus it is likely that each element is already in the cache from looking up the previous element, resulting in fast operations.

(c) Heap sort will not benefit much from caching, because it accesses array elements in a non-uniform (specifically non-sequential) way. Since heap sort performs large jumps around the array, it is unlikely that elements being accessed are already cached, since it is rare that two elements being compared/swapped are nearby in the array. As a result, caching does not help much.

Problem 5

(20 points) Given the truth table in Table 1,
(a) draw a Karnaugh map,
(b) find the maximum groups of 1, 2, 4, 8 elements in the Karnaugh map,
(c) write the boolean function for A and write it as a sum of products.

---

w	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
x	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
y	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
z	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$A(w, x, y, z)$	1	0	0	1	1	1	1	1	0	0	1	1	0	0	1	0

Table 1: Initial truth table for Problem 5.

(a) See Table 2.

(b) See Table 2.

(c) $A(w, x, y, z) = w'x + w'y'z' + x'yz + wyz'$

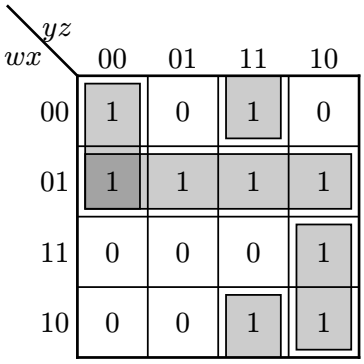

Table 2: Karnaugh map for $A(w, x, y, z)$.

Problem 6

(20 points) Write in x86-64 assembly the code to implement the Fibonacci sequence.

```c
int fib(int n) {
  if (n == 0 || n == 1) {
    return n;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}
```

Since the C code uses `int` for variables, I use mostly 4-byte instructions/registers here. I still use 8 bytes for pushing and popping so that the stack is aligned properly, but the upper 4 bytes don't get used when I manipulate the values.

```asm
.section .text
.global fib
fib:
    pushq %rbp              # Set up stack frame
    movq %rsp, %rbp
    testl $(~1), %edi       # Test if (n & ~1) is 0, i.e. n is 0 or 1
    movl %edi, %eax
    jz return
    decl %edi               # Decrement %edi to get (n - 1)
    pushq %rdi              # Save (n - 1) on the stack (twice for alignment)
    pushq %rdi
    call fib                # Call fib(n - 1)
    popq %rdi               # Retrieve saved (n - 1) into %rdi
    pushq %rax              # Save result of fib(n - 1)
    decl %edi               # Decrement %edi to get (n - 2)
    call fib                # Call fib(n - 2)
    popq %rdx               # Retrieve saved result of fib(n - 1) and add to
    addl %edx, %eax         # fib(n - 2) result
return:
    leave                   # Restore stack and return from function
    ret                     # At this point, the return value
                            # is already in %eax
```

Problem 7

(20 points) Write the x86-64 assembly code that your compiler will generate for the following expressions. Assume that `x`, `y`, and `z` are global variables of type `long`.

(a) `y = z + 3 * x`

(b) `y = (x > 1)`

(c)
```
x = 0;
while (x < y) {
  x = x + 1;
}
```

(a)
```
movq z(%rip), %rbx # push z
movq $3, %r10      # push 3
movq x(%rip), %r13 # push x
imulq %r13, %r10   # 3 * x
addq %r10, %rbx    # result + z
movq %rbx, y(%rip) # y = result
```

(b)
```
movq x(%rip), %rbx # push x
movq $1, %r10      # push 1
cmpq %r10, %rbx    # compare x to 1
setg %bl           # set to 1 iff >
movzbq %bl, %rbx   # extend to 8 bytes
movq %rbx, y(%rip) # y = result
```

(c)
```
movq $0, x(%rip)   # x = 0;
while_start_0:     # start of while loop
movq x(%rip), %rbx # push x
movq y(%rip), %r10 # push y
cmpq %r10, %rbx    # compare x to y
setl %bl           # set to 1 iff <
movzbq %bl, %rbx   # extend to 8 bytes
testq %rbx, %rbx   # check condition
jz while_end_0     # exit loop if false
movq x(%rip), %rbx # push x
movq $1, %r10      # push 1
addq %r10, %rbx    # x + 1
movq %rbx, x(%rip) # x = result
jmp while_start_0  # go back to loop condition
while_end_0:       # end of while loop
```

Problem 8 *(20 points)* We want to implement a simplified `switch()` statement in the compiler. E.g.:

```
switch (x) {
  case 1: y = 5; break;
  case 2: y = 4; break;
  default: y = 3; break;
}
```

It will work similar to the switch statement in C, but simplified. In the example, given the value of `x`, it will compare if `x == 1`, and if true, execute `y = 5`. Otherwise, it will go to the next comparison, `x == 2`, and if true, execute `y = 4`. If all of these fail, it will execute the default `y = 3`. Fill up the actions in the grammar. Add comments to document your code. There is always "`break;`" at the end of each option.

This solution requires the definition of a variable `int current_switch;` in the preamble in order to support nested switch statements. Additionally, a final `RCURLY` had to be added to the end of the `switch_statement` pattern, because that was missing and would cause a syntax error.

```
switch_statement:
    SWITCH LPARENT expression {
        /* Save the previous value of current_switch so that it can be restored
         * at the end of this switch statement. This ensures that
         * current_switch always contains the label number of the current
         * switch statement (if inside one). This allows us to avoid problems
         * with nested switches because the inner switch will change
         * current_switch, then restore it to its previous value (i.e. that of
         * the outer switch) before we return to continue processing the outer
         * switch statement. */
        $<my_nlabel>1 = current_switch;
        current_switch = nlabel++;
    } RPARENT LCURLY case_list {
        /* Get rid of the expression being compared now that we've finished
         * processing all the cases. */
        top--;
    } DEFAULT COLON statement BREAK SEMICOLON {
        fprintf(fasm, "switch_end_%d:\n", current_switch);
        /* Restore current_switch. */
        current_switch = $<my_nlabel>1;
    } RCURLY
;

case_item:
    CASE expression COLON {
        $<my_nlabel>1 = nlabel++;
        fprintf(fasm, "\tcmpq %%%s, %%%s\n", regStk[top - 2], regStk[top - 1]);
        top--;
        fprintf(fasm, "\tjnz case_end_%d\n", $<my_nlabel>1);
    } statement BREAK SEMICOLON {
        fprintf(fasm, "\tjmp switch_end_%d\n", current_switch);
        fprintf(fasm, "case_end_%d:\n", $<my_nlabel>1);
    }
;

case_list:
    case_item
    | case_list case_item
;
```