The Midterm

Brad Karp
UCL Computer Science

CS 0019
7th February 2019

Midterm Exam

- 21st February, 3:05 PM
- Darwin B40 Lecture Theatre
- 1 hour 25 minutes
- UCL standard calculators permitted
- No other reference materials (books, notes) permitted
- Covers all assigned readings and lectures through 19th February and Courseworks 1 -3
- Please be sure to bring an HB pencil for filling in True/False/Don't Know answer sheet
- Absence can only be excused by unforeseeable extenuating circumstances (through CS ECs process)

Rubric

- **Part I: multi-part short answer questions**
  - A scenario, and multiple questions about that scenario, each of which you answer
  - No choice of questions; all students answer all questions
- **Part II: True/False/Don't Know questions**
  - A scenario and a series of five statements about that scenario
  - For each statement, you must indicate whether the statement is true, false, or that you do not know ("T", "F", or "D")
  - Any number of statements may be true (from 0 through 5)
  - For each true statement you identify as true and false statement you identify as false, you receive 1 mark
  - For each statement whose truth or falsehood you contradict, you lose 1 mark (negative marking discourages guessing)
  - "D" answers neither gain nor lose marks
  - Sum of "raw" marks for Part II normalized across the class

Example Multi-Part Short Answer Question

Consider the disassembly of the x86-64 code for C function f1() shown below (in this lecture, on the next slide).

(a) How many arguments does f1() take? [1 mark]

(b) How much space on the stack does a single invocation of f1() allocate for local variables? [1 mark]

(c) How much of this space on the stack does a single invocation of f1() use for local variables? [2 marks]

(d) Which registers does f1() save on the stack before it makes a function call and does f1() restore from the stack after the function call returns? [2 marks]

(e) For each register in your answer to (d), why does f1() save that register on the stack? [3 marks]

(f) Write the C function that you would expect to compile into f1(). [6 marks]

```
_f1:
    0: 55                      pushq %rbp
    1: 48 89 e5                movq %rsp, %rbp
    4: 48 83 ec 20             subq $32, %rsp
    8: 48 89 7d f0             movq %rdi, -16(%rbp)
    c: 48 83 7d f0 00          cmpq $0, -16(%rbp)
   11: 0f 84 28 00 00 00       je 40 <_f1+0x3F>
   17: 48 8b 45 f0             movq -16(%rbp), %rax
   1b: 48 8b 4d f0             movq -16(%rbp), %rcx
   1f: 48 83 e9 01             subq $1, %rcx
   23: 48 89 cf                movq %rcx, %rdi
   26: 48 89 45 e8             movq %rax, -24(%rbp)
   2a: e8 d1 ff ff ff          callq -47 <_f1>
   2f: 48 8b 4d e8             movq -24(%rbp), %rcx
   33: 48 01 c1                addq %rax, %rcx
   36: 48 89 4d f8             movq %rcx, -8(%rbp)
   3a: e9 08 00 00 00          jmp 8 <_f1+0x47>
   3f: 48 c7 45 f8 00 00 00 00 movq $0, -8(%rbp)
   47: 48 8b 45 f8             movq -8(%rbp), %rax
   4b: 48 83 c4 20             addq $32, %rsp
   4f: 5d                      popq %rbp
   50: c3                      retq
```

Answers to Multi-Part Short Answer Question

(a) One.

(b) 32 bytes.

(c) 24 bytes.

(d) %rbp, %rax

(e) %rbp: callee-saved, used as frame pointer, so must be saved and restored upon entry and exit to hold correct value upon return to caller; %rax: caller-saved, used as return value, so will be clobbered by callee.

(f)
```
long f1(long x)
{
    if (x)
        return x + f1(x - 1);
    else
        return 0;
}
```

Example True/False/Don't Know Question

- Consider the following C structure and its use, which are to be compiled on an x86-64 machine:

```
struct foo {
  char x[5];
  uint16_t i;
  char y;
};
struct foo bar[16];
```

A. **`sizeof(struct foo)`** is 8.

B. Swapping the order of **`x[]`** and **`i`** in the **`struct foo`** declaration changes **`sizeof(struct foo)`**.

C. **`malloc(sizeof(struct foo))`** will return storage aligned neither more coarsely nor more finely than needed by **`struct foo`**.

D. In general, for any **`struct str`** whose members are of any C type, whether a basic x86-64 C type (e.g., **`long`**, **`char`**) or a derived type built from such basic types (e.g., array of **`char`**s, **`struct`**, pointer), sorting **`struct str`**'s members in increasing order of alignment in the **`struct str`** declaration yields the smallest possible **`sizeof(struct foo)`**.

Example True/False/Don't Know Question

- Consider the following C structure and its use, which are to be compiled on an x86-64 machine:

```
struct foo {
  char x[5];
  uint16_t i;
  char y;
};
struct foo bar[16];
```

A. **`sizeof(struct foo)`** is 8. **False.**

B. Swapping the order of **`x[]`** and **`i`** in the **`struct foo`** declaration changes **`sizeof(struct foo)`**.

C. **`malloc(sizeof(struct foo))`** will return storage aligned neither more coarsely nor more finely than needed by **`struct foo`**.

D. In general, for any **`struct str`** whose members are of any C type, whether a basic x86-64 C type (e.g., **`long`**, **`char`**) or a derived type built from such basic types (e.g., array of **`char`**s, **`struct`**, pointer), sorting **`struct str`**'s members in increasing order of alignment in the **`struct str`** declaration yields the smallest possible **`sizeof(struct foo)`**.

Example True/False/Don't Know Question

- Consider the following C structure and its use, which are to be compiled on an x86-64 machine:

```
struct foo {
  char x[5];
  uint16_t i;
  char y;
};
struct foo bar[16];
```

A. `sizeof(struct foo)` is 8. **False.**

B. Swapping the order of `x[]` and `i` in the `struct foo` declaration changes `sizeof(struct foo)`. **True.**

C. `malloc(sizeof(struct foo))` will return storage aligned neither more coarsely nor more finely than needed by `struct foo`.

D. In general, for any `struct str` whose members are of any C type, whether a basic x86-64 C type (e.g., `long`, `char`) or a derived type built from such basic types (e.g., array of `char`s, `struct`, pointer), sorting `struct str`'s members in increasing order of alignment in the `struct str` declaration yields the smallest possible `sizeof(struct foo)`.

Example True/False/Don't Know Question

- Consider the following C structure and its use, which are to be compiled on an x86-64 machine:

```
struct foo {
  char x[5];
  uint16_t i;
  char y;
};
struct foo bar[16];
```

A. `sizeof(struct foo)` is 8. **False.**

B. Swapping the order of `x[]` and `i` in the `struct foo` declaration changes `sizeof(struct foo)`. **True.**

C. `malloc(sizeof(struct foo))` will return storage aligned neither more coarsely nor more finely than needed by `struct foo`. **False.**

D. In general, for any `struct str` whose members are of any C type, whether a basic x86-64 C type (e.g., `long`, `char`) or a derived type built from such basic types (e.g., array of `chars`, `struct`, pointer), sorting `struct str`'s members in increasing order of alignment in the `struct str` declaration yields the smallest possible `sizeof(struct foo)`.

Example True/False/Don't Know Question

- Consider the following C structure and its use, which are to be compiled on an x86-64 machine:

```
struct foo {
  char x[5];
  uint16_t i;
  char y;
};
struct foo bar[16];
```

A. **`sizeof(struct foo)`** is 8. **False.**

B. Swapping the order of **`x[]`** and **`i`** in the **`struct foo`** declaration changes **`sizeof(struct foo)`**. **True.**

C. **`malloc(sizeof(struct foo))`** will return storage aligned neither more coarsely nor more finely than needed by **`struct foo`**. **False.**

D. In general, for any **`struct str`** whose members are of any C type, whether a basic x86-64 C type (e.g., **`long`**, **`char`**) or a derived type built from such basic types (e.g., array of **`char`**s, **`struct`**, pointer), sorting **`struct str`**'s members in increasing order of alignment in the **`struct str`** declaration yields the smallest possible **`sizeof(struct foo)`**. **True.**