cs5460/6460: Operating Systems

Midterm recap, sample questions

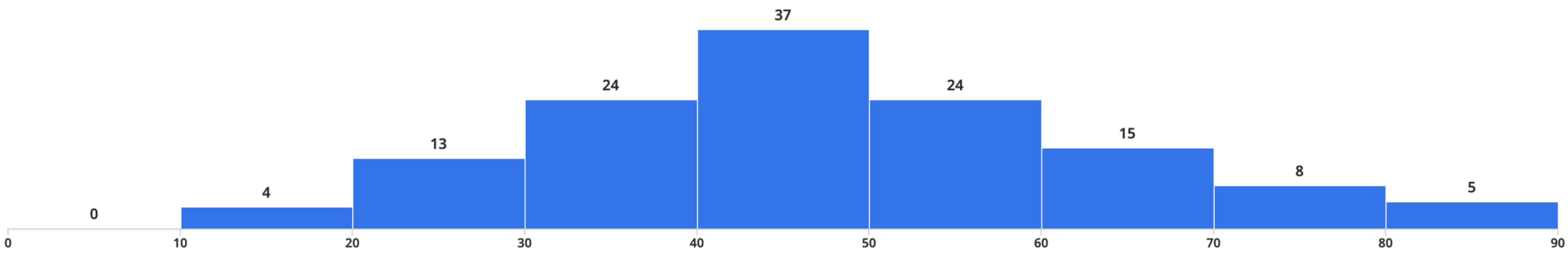
Anton Burtsev

March, 2025


Midterm 2024

Review Grades for Midterm

Grades Published



Histogram of score distribution

Minimum	Median	Maximum	Mean	Std Dev
13.67	46.83	88.0	47.83	15.52

135 Students

Search

Q1 OS interfaces

10 Points

Write a simple UNIX program, `tee` that reads from standard input and writes to standard output and a file specified in command line. For example, if invoked like this:

```
echo Hello | tee foobar.txt
```

"Hello" shows up on the screen and in foobar.txt

"Hello" shows up on the screen and in foobar.txt

```
int pipeP = -1;
if (strcmp(args[i], "|") == 0)
   {
     pipeP = i;
   }
if(pipeP != -1){
args[pipeP] = NULL;

   int pipe_fd[2]; // Stores the output content of the pipeline
   if (pipe(pipe_fd) == -1)
   {
     printf("pipe error");
     exit(0);
   }

   pid_t pid = fork();

   if (pid < 0)
   {
     printf("fork error");
     exit(0);
   }
   else if (pid == 0)
   { // Process the command on the left and store it
     close(pipe_fd[0]);
     dup2(pipe_fd[1], STDOUT_FILENO);
     close(pipe_fd[1]);

     if (execvp(args[0], args) == -1)
     {
       printf("execvp error");
       exit(0);
     }
   }
   else
   { // Run the command on the right using the results on the left
     wait(NULL);
     close(pipe_fd[1]);
     dup2(pipe_fd[0], STDIN_FILENO);
     close(pipe_fd[0]);

     if (execvp(args[pipeP + 1], args + pipeP + 1) == -1)
     {
       printf("execvp error");
       exit(0);
     }
   }
}
```

```
void fork();
void exec(path, args);
void wait();
int open(fd, R|W);
void close(fd);
void pipe(fd[2]);
int dup(fd);
int read(fd, buf, n);
int write(fd, buf, n);
```

Q1 OS interfaces

10 Points

Write a simple UNIX program, `tee` that reads from standard input and writes to standard output and a file specified in command line. For example, if invoked like this:

```
echo Hello | tee foobar.txt
```

"Hello" sh

"Hello" shows up on the screen and in foobar.txt

```
int fp = open(argv[1], O_WRONLY | O_CREATE);
char next;
read(0, &next, 1);

while(next != 0) {
  write(fp, &next, 1);
  write(1, &next, 1);

  read(0, &next, 1);
}
close(fp);
```

Q2 Assembly

15 Points

Below is C and assembly code for the `strncpy()` (string copy) function from the xv6 operating system (i.e., `strpy()` copies one string into another). In C strings are represented as continuous arrays of bytes (each character is a byte) that end with a `0` (or NULL) to designate the end of the string.

```
char*
strncpy(char *s, const char *t, int n)
{
  char *os;

  os = s;
  while(n-- > 0 && (*s++ = *t++) != 0)
    ;
  while(n-- > 0)
    *s++ = 0;
  return os;
}
```

```
00000190 <strncpy>:
 190:   55                      push   ebp
 191:   89 e5                   mov    ebp,esp
 193:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 196:   56                      push   esi
 197:   8b 4d 10                mov    ecx,DWORD PTR [ebp+0x10]
 19a:   53                      push   ebx
 19b:   8b 5d 0c                mov    ebx,DWORD PTR [ebp+0xc]
 19e:   89 c2                   mov    edx,eax
 1a0:   eb 19                   jmp    1bb <strncpy+0x2b>
 1a2:   8d b6 00 00 00 00       lea    esi,[esi+0x0]
 1a8:   83 c3 01                add    ebx,0x1
 1ab:   0f b6 4b ff             movzx  ecx,BYTE PTR [ebx-0x1]
 1af:   83 c2 01                add    edx,0x1
 1b2:   84 c9                   test   cl,cl
 1b4:   88 4a ff                mov    BYTE PTR [edx-0x1],cl
 1b7:   74 09                   je     1c2 <strncpy+0x32>
 1b9:   89 f1                   mov    ecx,esi
 1bb:   85 c9                   test   ecx,ecx
 1bd:   8d 71 ff                lea    esi,[ecx-0x1]
 1c0:   7f e6                   jg     1a8 <strncpy+0x18>
 1c2:   31 c9                   xor    ecx,ecx
 1c4:   85 f6                   test   esi,esi
 1c6:   7e 0f                   jle    1d7 <strncpy+0x47>
 1c8:   c6 04 0a 00             mov    BYTE PTR [edx+ecx*1],0x0
 1cc:   89 f3                   mov    ebx,esi
 1ce:   83 c1 01                add    ecx,0x1
 1d1:   29 cb                   sub    ebx,ecx
 1d3:   85 db                   test   ebx,ebx
 1d5:   7f f1                   jg     1c8 <strncpy+0x38>
 1d7:   5b                      pop    ebx
 1d8:   5e                      pop    esi
 1d9:   5d                      pop    ebp
 1da:   c3                      ret
 1db:   90                      nop
 1dc:   8d 74 26 00             lea    esi,[esi+eiz*1+0x0]
```

```
char*
strncpy(char *s, const char *t, int n)
{
  char *os;

  os = s;
  while(n-- > 0 && (*s++ = *t++) != 0)
    ;
  while(n-- > 0)
    *s++ = 0;
  return os;
}
```

```
00000190 <strncpy>:
 190:   55                      push   ebp
 191:   89 e5                   mov    ebp,esp
 193:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 196:   56                      push   esi
 197:   8b 4d 10                mov    ecx,DWORD PTR [ebp+0x10]
 19a:   53                      push   ebx
 19b:   8b 5d 0c                mov    ebx,DWORD PTR [ebp+0xc]
 19e:   89 c2                   mov    edx,eax
 1a0:   eb 19                   jmp    1bb <strncpy+0x2b>
 1a2:   8d b6 00 00 00 00       lea    esi,[esi+0x0]
 1a8:   83 c3 01                add    ebx,0x1
 1ab:   0f b6 4b ff             movzx  ecx,BYTE PTR [ebx-0x1]
 1af:   83 c2 01                add    edx,0x1
 1b2:   84 c9                   test   cl,cl
 1b4:   88 4a ff                mov    BYTE PTR [edx-0x1],cl
 1b7:   74 09                   je     1c2 <strncpy+0x32>
 1b9:   89 f1
 1bb:   85 c9
 1bd:   8d 71 ff
 1c0:   7f e6
 1c2:   31 c9
 1c4:   85 f6
 1c6:   7e 0f
 1c8:   c6 04 0a 00             mov    BYTE PTR [edx+ecx*1],0x0
 1cc:   89 f3                   mov    ebx,esi
 1ce:   83 c1 01                add    ecx,0x1
 1d1:   29 cb                   sub    ebx,ecx
 1d3:   85 db                   test   ebx,ebx
 1d5:   7f f1                   jg     1c8 <strncpy+0x38>
 1d7:   5b                      pop    ebx
 1d8:   5e                      pop    esi
 1d9:   5d                      pop    ebp
 1da:   c3                      ret
 1db:   90                      nop
 1dc:   8d 74 26 00             lea    esi,[esi+eiz*1+0x0]
```

Q2.1

5 Points

What happens if you replace instruction at address `190` with a `nop` instruction? (`nop` does nothing, i.e., it advances the instruction pointer to the next instruction but does not affect memory or registers).

```
00000190 <strncpy>:
 190:   55                      push   ebp
 191:   89 e5                   mov    ebp,esp
 193:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 196:   56                      push   esi
 197:   8b 4d 10                mov    ecx,DWORD PTR [ebp+0x10]
 19a:   53                      push   ebx
 19b:   8b 5d 0c                mov    ebx,DWORD PTR [ebp+0xc]
 19e:   89 c2                   mov    edx,eax
 1a0:   eb 19                   jmp    1bb <strncpy+0x2b>
 1a2:   8d b6 00 00 00 00       lea    esi,[esi+0x0]
 1a8:   83 c3 01                add    ebx,0x1
 1ab:   0f b6 4b ff             movzx  ecx,BYTE PTR [ebx-0x1]
 1af:   83 c2 01                add    edx,0x1
 1b2:   84 c9                   test   cl,cl
 1b4:   88 4a ff                mov    BYTE PTR [edx-0x1],cl
 1b7:   74 09                   je     1c2 <strncpy+0x32>
 1b9:   89 f1
 1bb:   85 c9
 1bd:   8d 71 ff
 1c0:   7f e6
 1c2:   31 c9
 1c4:   85 f6
 1c6:   7e 0f
 1c8:   c6 04 0a 00
 1cc:   89 f3
 1ce:   83 c1 01
 1d1:   29 cb
 1d3:   85 db
 1d5:   7f f1
 1d7:   5b                      pop    ebx
 1d8:   5e                      pop    esi
 1d9:   5d                      pop    ebp
 1da:   c3                      ret
 1db:   90                      nop
 1dc:   8d 74 26 00             lea    esi,[esi+eiz*1+0x0]
```

Q2.1

5 Points

What happens if you replace instruction at address `190` with a `nop` instruction? (`nop` does nothing, i.e., it advances the instruction pointer to the next instruction but does not affect memory or registers).

> The previous value of ebp will not be preserved on the stack. When the strcpy tries to "pop ebp" at 0x1d9, an incorrect value on top of the stack will be loaded into ebp. Then the ret instruction at 0x1da will try to jump to an incorrect address and very likely segfault.

```
00000190 <strncpy>:
 190:   55                      push   ebp
 191:   89 e5                   mov    ebp,esp
 193:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 196:   56                      push   esi
 197:   8b 4d 10                mov    ecx,DWORD PTR [ebp+0x10]
 19a:   53                      push   ebx
 19b:   8b 5d 0c                mov    ebx,DWORD PTR [ebp+0xc]
 19e:   89 c2                   mov    edx,eax
 1a0:   eb 19                   jmp    1bb <strncpy+0x2b>
 1a2:   8d b6 00 00 00 00       lea    esi,[esi+0x0]
 1a8:   83 c3 01                add    ebx,0x1
 1ab:   0f b6 4b ff             movzx  ecx,BYTE PTR [ebx-0x1]
 1af:   83 c2 01                add    edx,0x1
 1b2:   84 c9                   test   cl,cl
 1b4:   88 4a ff                mov    BYTE PTR [edx-0x1],cl
 1b7:   74 09                   je     1c2 <strncpy+0x32>
 1b9:   89 f1                   mov    ecx,esi
 1bb:   85 c9                   test   ecx,ecx
 1bd:   8d 71 ff                lea    esi,[ecx-0x1]
 1c0:   7f e6                   jg     1a8 <strncpy+0x18>
 1c2:   31 c9                   xor    ecx,ecx
 1c4:   85 f6                   test   esi,esi
 1c6:   7e 0f                   jle    1d7 <strncpy+0x47>
 1c8:   c6 04 0a 00             mov    BYTE PTR [edx+ecx*1],0x0
 1cc:   89 f3                   mov    ebx,esi
 1ce:   83 c1 01
 1d1:   29 cb
 1d3:   85 db
 1d5:   7f f1
 1d7:   5b
 1d8:   5e                      pop    esi
 1d9:   5d                      pop    ebp
 1da:   c3                      ret
 1db:   90                      nop
 1dc:   8d 74 26 00             lea    esi,[esi+eiz*1+0x0]
```

Q2.2

5 Points

Same as above, but now you put two `nop` instructions instead of instruction at address `1b7`

```
00000190 <strncpy>:
 190:   55                      push   ebp
 191:   89 e5                   mov    ebp,esp
 193:   8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
 196:   56                      push   esi
 197:   8b 4d 10                mov    ecx,DWORD PTR [ebp+0x10]
 19a:   53                      push   ebx
 19b:   8b 5d 0c                mov    ebx,DWORD PTR [ebp+0xc]
 19e:   89 c2                   mov    edx,eax
 1a0:   eb 19                   jmp    1bb <strncpy+0x2b>
 1a2:   8d b6 00 00 00 00       lea    esi,[esi+0x0]
 1a8:   83 c3 01                add    ebx,0x1
 1ab:   0f b6 4b ff             movzx  ecx,BYTE PTR [ebx-0x1]
 1af:   83 c2 01                add    edx,0x1
 1b2:   84 c9                   test   cl,cl
 1b4:   88 4a ff                mov    BYTE PTR [edx-0x1],cl
 1b7:   74 09                   je     1c2 <strncpy+0x32>
 1b9:   89 f1                   mov    ecx,esi
 1bb:   85 c9                   test   ecx,ecx
 1bd:   8d 71 ff                lea    esi,[ecx-0x1]
 1c0:   7f e6                   jg     1a8 <strncpy+0x18>
 1c2:   31 c9                   xor    ecx,ecx
 1c4:   85 f6                   test   esi,esi
 1c6:   7e 0f                   jle    1d7 <strncpy+0x47>
 1c8:   c6 04 0a 00             mov    BYTE PTR [edx+ecx*1],0x0
 1cc:   89 f3                   mov    ebx,esi
 1ce:   83 c1 01
 1d1:   29 cb
 1d3:   85 db
 1d5:   7f f1
 1d7:   5b
 1d8:   5e
 1d9:   5d
 1da:   c3
 1db:   90
 1dc:   8d 74 26 00
```

Q2.2

5 Points

Same as above, but now you put two `nop` instructions instead of instruction at address `1b7`

The first while loop will not exit when it encounters a null character at the end of t, causing it to copy extra garbage data into s instead of zeroing out the remainder of s.

```
00000190 <strncpy>:
 190:   55
 191:   89 e5
 193:   8b 45 08
 196:   56
 197:   8b 4d 10
 19a:   53                      push   ebx
 19b:   8b 5d 0c                mov    ebx,DWORD PTR [ebp+0xc]
 19e:   89 c2                   mov    edx,eax
 1a0:   eb 19                   jmp    1bb <strncpy+0x2b>
 1a2:   8d b6 00 00 00 00       lea    esi,[esi+0x0]
 1a8:   83 c3 01                add    ebx,0x1
 1ab:   0f b6 4b ff             movzx  ecx,BYTE PTR [ebx-0x1]
 1af:   83 c2 01                add    edx,0x1
 1b2:   84 c9                   test   cl,cl
 1b4:   88 4a ff                mov    BYTE PTR [edx-0x1],cl
 1b7:   74 09                   je     1c2 <strncpy+0x32>
 1b9:   89 f1                   mov    ecx,esi
 1bb:   85 c9                   test   ecx,ecx
 1bd:   8d 71 ff                lea    esi,[ecx-0x1]
 1c0:   7f e6                   jg     1a8 <strncpy+0x18>
 1c2:   31 c9                   xor    ecx,ecx
 1c4:   85 f6                   test   esi,esi
 1c6:   7e 0f                   jle    1d7 <strncpy+0x47>
 1c8:   c6 04 0a 00             mov    BYTE PTR [edx+ecx*1],0x0
 1cc:   89 f3                   mov    ebx,esi
 1ce:   83 c1 01                add    ecx,0x1
 1d1:   29 cb                   sub    ebx,ecx
 1d3:   85 db                   test   ebx,ebx
 1d5:   7f f1                   jg     1c8 <strncpy+0x38>
 1d7:   5b                      pop    ebx
 1d8:   5e                      pop    esi
 1d9:   5d                      pop    ebp
 1da:   c3                      ret
 1db:   90                      nop
 1dc:   8d 74 26 00             lea    esi,[esi+eiz*1+0x0]
```

Q2.3

5 Points

Same as above, but now you put `nop` instead of the instruction at address `1d7`

```
00000190 <strncpy>:
 190:   55
 191:   89 e5
 193:   8b 45 08
 196:   56
 197:   8b 4d 10
 19a:   53
 19b:   8b 5d 0c
 19e:   89 c2
 1a0:   eb 19
 1a2:   8d b6 00 00 0
 1a8:   83 c3 01
```

Q2.3

5 Points

Same as above, but now you put `nop` instead of the instruction at address `1d7`

At 0x1d8, strdpy will pop the last pushed value into esi, but this value is the saved value of ebx -- this is not correct. It will similarly load the saved value of esi into ebp, and leave the saved value of ebp on the stack. The ret instruction at 0x1da will then try to jump to the saved value of ebp, and will very likely segfault.

```
 1ab:   0f b6 4b ff             movzx  ecx,BYTE PTR [ebx-0x1]
 1af:   83 c2 01                add    edx,0x1
 1b2:   84 c9                   test   cl,cl
 1b4:   88 4a ff                mov    BYTE PTR [edx-0x1],cl
 1b7:   74 09                   je     1c2 <strncpy+0x32>
 1b9:   89 f1                   mov    ecx,esi
 1bb:   85 c9                   test   ecx,ecx
 1bd:   8d 71 ff                lea    esi,[ecx-0x1]
 1c0:   7f e6                   jg     1a8 <strncpy+0x18>
 1c2:   31 c9                   xor    ecx,ecx
 1c4:   85 f6                   test   esi,esi
 1c6:   7e 0f                   jle    1d7 <strncpy+0x47>
 1c8:   c6 04 0a 00             mov    BYTE PTR [edx+ecx*1],0x0
 1cc:   89 f3                   mov    ebx,esi
 1ce:   83 c1 01                add    ecx,0x1
 1d1:   29 cb                   sub    ebx,ecx
 1d3:   85 db                   test   ebx,ebx
 1d5:   7f f1                   jg     1c8 <strncpy+0x38>
 1d7:   5b                      pop    ebx
 1d8:   5e                      pop    esi
 1d9:   5d                      pop    ebp
 1da:   c3                      ret
 1db:   90                      nop
 1dc:   8d 74 26 00             lea    esi,[esi+eiz*1+0x0]
```

Q3

20 Points

Below is a code snippet of the `cat()` function from the xv6 `cat` utility

```c
#include "types.h"
#include "stat.h"
#include "user.h"

char buf[512];

void
cat(int fd)
{
  int n;

  while((n = read(fd, buf, sizeof(buf))) > 0) {
    if (write(1, buf, n) != n) {
      printf(1, "cat: write error\n");
      exit();
    }
  }
  if(n < 0){
    printf(1, "cat: read error\n");
    exit();
  }
}
```

Q3
20 Points

Below is a code sni

Q3.1 Memory allocation
10 Points

For each variable used in the program above, explain where (stack/heap/data/bss section) this variable is allocated.

```
#include "types.h"
#include "stat.h"
#include "user.h"

char buf[512];

void
cat(int fd)
{
  int n;

  while((n = read(fd, buf, sizeof(buf))) > 0) {
    if (write(1, buf, n) != n) {
      printf(1, "cat: write error\n");
      exit();
    }
  }
  if(n < 0){
    printf(1, "cat: read error\n");
    exit();
  }
}
```

Q3
20 Points

Below is a code sni

```
#include "type
#include "stat
#include "user.h"

char buf[512];

void
cat(int fd)
{
  int n;

  while((n = read(fd, buf, sizeof(buf))) > 0) {
    if (write(1, buf, n) != n) {
      printf(1, "cat: write error\n");
      exit();
    }
  }
  if(n < 0){
    printf(1, "cat: read error\n");
    exit();
  }
}
```

Q3.1 Memory allocation
10 Points

For each variable used in the program above, explain where (stack/heap/data/bss section) this variable is allocated.

buf: bss
fd: stack
n: stack

Q3

20 Points

Below is a code sni

```
#include "type
#include "stat
#include "user.h"

char buf[512];

void
cat(int fd)
{
  int n;

  while((n = read(fd, buf, sizeof(buf))) > 0) {
    if (write(1, buf, n) != n) {
      printf(1, "cat: write error\n");
      exit();
    }
  }
  if(n < 0){
    printf(1, "cat: read error\n");
    exit();
  }
}
```

Q3.1 Memory allocation

10 Points

For each variable used in the program above, explain where (stack/heap/data/bss section) this variable is allocated.

buf: bss
fd: stack
n: stack

Well, not entirely correct…

- What is missing?

Q3
20 Points

Below is a code snip

Q3.2
10 Points

Which lines of the code above require relocation if loaded at a different memory address. Explain your answer (1 point for each non-trivial line)

```c
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 char buf[512];
6
7 void
8 cat(int fd)
9 {
10   int n;
11
12   while((n = read(fd, buf, sizeof(buf))) > 0) {
13     if (write(1, buf, n) != n) {
14       printf(1, "cat: write error\n");
15       exit();
16     }
17   }
18   if(n < 0){
19     printf(1, "cat: read error\n");
20     exit();
21   }
22 }
```

Q3
20 Points

Below is a code snip

```
1 #include "types
2 #include "stat.]
3 #include "user.]
4
5 char buf[512];
6
7 void
8 cat(int fd)
9 {
10   int n;
11
12   while((n = rea
13     if (write(1
14       printf(1,
15       exit();
16     }
17   }
18   if(n < 0){
19     printf(1, "
20     exit();
21   }
22 }
```

Q3.2
10 Points

Which lines of the code above require relocation if loaded at a different memory address. Explain your answer (1 point for each non-trivial line)

Line 12:
read() function would require relocation as it is an external function
buf would require relocation as it is a global variable

Line 13:
write() function would require relocation as it is an external function
buf would require relocation as it is a global variable

Line 14:
printf() function would require relocation as it is an external function
"cat: write error\n" (treated as a string constant by C compiler) would require relocation

Line 15:
exit() function would require relocation as it is an external function

Line 19:
printf() function would require relocation as it is an external function
"cat: read error\n" (treated as a string constant by C compiler) would require relocation

Line 20:
exit() function would require relocation as it is an external function

Q4

20 Points

Imagine you have an x86 machine which has all the same instructions as we discussed in class, besides that it does not have `call`, `ret`, `push` and `pop`. Imagine you're in control of the compiler and can generate any assembly code you like.

Q4.1

10 Points

Explain how can you support function invocations? Show an example of the assembly code that invokes the `int foobar(int a, int b, int c)` function.

Q4
20 Points

Imagine you hav
besides that it dc
and can generate

Q4.1
10 Points

Explain how can y
invokes the `int f`

Without push and pop, we'd have to use the mov instruction to move values to the appropriate addresses on the stack (esp holds the memory address to the top of the stack).

Without the call instruction, we'd have to modify and use the value in the eip register to "push" as return address on the stack before transferring control to the callee.

C:
foobar(1, 2, 3);

Assembly:
Before:
push 0x3
push 0x2
push 0x1
call foobar

After:
mov [esp-0x4], 0x3
mov [esp-0x8], 0x2
mov [esp-0xc], 0x1
mov eax, eip
add eax, 0x14
mov [esp-0x10], eax
sub esp, 0x10
jmp foo
add esp, 0xc

Q4
20 Points

Imagine you have an x86 machine which has all the same instructions as we discussed in class, besides that it does not have `call`, `ret`, `push` and `pop`. Imagine you're in control of the compiler and can generate any assembly code you like.

Q4.2
10 Points

How will you maintain the stack frame and return from the function? Show assembly code that maintains the stack frame inside `foobar()` and returns from it.

Q4
20 Points

Imagine you have an :
besides that it does n
and can generate any

Q4.2
10 Points

How will you maintain
maintains the stack fr

Just as in A4.1, we'd have to use mov instructions to main stack frames in lieu of the push/pop mechanism.
To return, we'd use the value at address which is (four bytes) before the frame pointer of the callee since we know that that is where we'd find the return address on the stack when we reach the epilogue of the callee.
Since ecx and edx are caller-saved (in xv6-32), we can use it as temporary registers without having to restore them in the callee.

Assembly:
Before:

```
foobar:
push ebp
mov ebp, esp
...
pop ebp
ret
```

After:

```
foobar:
sub esp, 0x4
mov [esp], ebp
mov ebp, esp
...
mov ecx, ebp
mov ebp, [ecx]
mov edx, [ecx+0x4]
add esp, 0x4  \\"Pop" ebp
add esp, 0x4  \\"Pop" return address
jmp edx
```

Q5 Page tables

25 Points

Q5.1

10 Points

Consider the following 32-bit x86 page table setup.

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Specify all virtual address ranges mapped by this page table (don't forget to mention the physical ranges to which each virtual range is mapped), e.g., `virt: [a - b] -> phys: [x - z]`

Q5 Page tables

25 Points

Q5.1

10 Points

Consider the follo

virt: [0x0 - 0xfff] -> phs: [0x00003000 - 0x00003fff]
virt: [0x1000 - 0x1fff] -> phs: [0x00004000 - 0x00004fff]
virt: [0x400000 - 0x400fff] -> phs: [0x00006000 - 0x00006fff]
virt: [0x401000 - 0x401fff] -> phs: [0x00007000 - 0x00007fff]

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Specify all virtual address ranges mapped by this page table (don't forget to mention the physical ranges to which each virtual range is mapped), e.g., `virt: [a - b] -> phys: [x - z]`

Q5.2
15 Points

Using the same format for describing the page table as in the question above construct a page table that maps virtual addresses from 0x0 to 1MB (0x10_0000) and from 2GB (0x8000_0000) to 2GB + 1MB (0x8010_0000) to physical addresses from 0x0 to 1MB (0x10_0000). You're free to choose where your PTD and PT pages are located in physical memory.

Q5.2
15 Points

Using the same format for describing the page table as in the question above construct a page table that maps virtual addresses from 0x0 to 1MB (0x10_0000) and from 2GB (0x8000_0000) to 2GB + 1MB (0x8010_0000) to physical addresses from 0x0 to 1MB (0x10_0000). You're free to choose where your PTD and PT pages are located in physical memory.

CR3 holds 0x200000
The Page Directory Page at physical address 0x200000:
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 512: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero

The Page Table Page at physical address 0x00001000 (which is PPN 0x1):
PTE 0 : PPN=0x00000, PTE_P, PTE_U, PTE_W
PTE 1 : PPN=0x00001, PTE_P, PTE_U, PTE_W
...
PTE 256 : PPN=0x00100, PTE_P, PTE_U, PTE_W
... all other PTEs are zero

The Page Table Page at physical address 0x00002000 (which is PPN 0x2):
PTE 0 : PPN=0x00000, PTE_P, PTE_U, PTE_W
PTE 1 : PPN=0x00001, PTE_P, PTE_U, PTE_W
...
PTE 256 : PPN=0x00100, PTE_P, PTE_U, PTE_W
... all other PTEs are zero

Thank you!