Operating Systems **Name (Print):**
Winter 2017
Final
3/20/2018
Time Limit: 4:00pm - 6:00pm

- **Don't forget to write your name on this exam.**
- **This is an open book, open notes exam. But no online or in-class chatting.**
- **Ask me if something is not clear in the questions.**
- **Organize your work**, in a reasonably neat and coherent way, in the space provided. Work scattered all over the page without a clear ordering will receive very little credit.
- **Mysterious or unsupported answers will not receive full credit**. A correct answer, unsupported by explanation will receive no credit; an incorrect answer supported by substantially correct explanations might still receive partial credit.
- If you need more space, use the back of the pages; clearly indicate when you have done this.

Problem	Points	Score
1	20	
2	15	
3	25	
4	5	
5	15	
6	5	
7	5	
Total:	90	

1. File systems

   Ben tries to understand the xv6 code of the `filewrite()` function (see the listing from the xv6 book below). He specifically looks at the definition of the `max` variable at line `5767` and tries to understand the logic behind it.

```
5750 // Write to file f.
5751 int
5752 filewrite(struct file *f, char *addr, int n)
5753 {
5754   int r;
5755
5756   if(f->writable == 0)
5757     return -1;
5758   if(f->type == FD_PIPE)
5759     return pipewrite(f->pipe, addr, n);
5760   if(f->type == FD_INODE){
5761     // write a few blocks at a time to avoid exceeding
5762     // the maximum log transaction size, including
5763     // i-node, indirect block, allocation blocks,
5764     // and 2 blocks of slop for nonaligned writes.
5765     // this really belongs lower down, since writei()
5766     // might be writing a device like the console.
5767     int max = ((LOGSIZE-1-1-2) / 2) * 512;
5768     int i = 0;
5769     while(i < n){
5770       int n1 = n - i;
5771       if(n1 > max)
5772         n1 = max;
5773
5774       begin_op();
5775       ilock(f->ip);
5776       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
5777         f->off += r;
5778       iunlock(f->ip);
5779       end_op();
5780
5781       if(r < 0)
5782         break;
5783       if(r != n1)
5784         panic("short filewrite");
5785       i += r;
5786     }
5787     return i == n ? n : -1;
5788   }
5789   panic("filewrite");
5790 }
```

(a) (10 points) Explain the role of the `max` variable and why it is defined the way it is (provide a detailed explanation for the formula and a high level explanation for why `max` is needed, i.e., what will go wrong without it).

To make sure that all writes to the file system leave it in a consistent state, in case of a power outage the `filewrite()` function splits large writes submitted by the user into multiple file system transactions. The `max` variable limits the maximum size for each transaction to make sure that it fits into the file system log. The file system log can accomodate `LOGSIZE` blocks. To make sure that each transaction is smaller than the `LOGSIZE` we need to count the number of blocks that can possibly be written by each transaction.

A minimal (one byte) write to an inode (`writei()` function) might trigger writes to the following blocks: 1) an allocation block that will be updated to reflect the fact tha the indirect block for the inode is allocated if the write is the first write outside of the first 12 blocks covered by the direct blocks (an allocation has to write to the block that holds the allocation bitmap), 2) a write to the indirect block itself, and 3) an update of the block that holds the inode since now it has a pointer to the new indirect block. This means that in the worst case a single write might trigger 3 additional block writes. Hence, the total max blocks we can accomodate in one transaction is no more than `LOGSIZE - 1 - 1 - 1`.

Note, however, that each individual block write might trigger an allocation of the block, hence in the worst case for each block written there will be an additional write to the allocation block (we have to divide the remaining log blocks by half). And hence the max number of blocks we can write should be: `(LOGSIZE - 1 - 1 - 1)/2`. Finally, the write that can fit in n blocks might be misaligned and require updates to n + 1 blocks. However if the write starts in the middle of a block it means that it was already allocated by the previous write (xv6 does not have `lseek()` system call that allows users to write into random locations in the file, hence the file is always written sequentially). Therefore, we only need to subtract one block from the original size of the log, and the max number of blocks we can allow to write without running of space in the log is: `(LOGSIZE - 1 - 1 - 1 - 1)/2`.

(b) (10 points) Finally, Ben thinks he understands the role of `max`, so he tries to explain it to Alice. Alice, however, is a mature xv6 hacker and she immediately spots an error in Ben's logic. She is quick to point out the bug in the xv6 code arguing that the definition of `max` above is incorrect. She quickly looks up the most recent version of xv6 and finds out that the bug she spotted is fixed. The most recent version of xv6 defines the `max` like:

```
// write a few blocks at a time to avoid exceeding
// the maximum log transaction size, including
// i-node, indirect block, allocation blocks,
// and 2 blocks of slop for non-aligned writes.
// this really belongs lower down, since writei()
// might be writing a device like the console.
int max = ((MAXOPBLOCKS-1-1-2) / 2) * 512;
```

Explain what is the bug that Alice has found, i.e., why the fix above is important and what can go wrong with the old definition?

Alice has previously looked at the logging implementation in xv6, hence she knows that xv6 allows concurrent transactions of MAXOPBLOCKS (10 blocks) each. She realizes that if multiple transactions of `(LOGSIZE - 1 - 1 - 1 - 1)/2` happen in parallel they might exhaust the space in the log. Hence, she suspects that the definition of `max` should be changed to limit the maximum write size based on the size of individual transaction, not the entire log.

2. Processes and boot

(a) (10 points) Explain how the first xv6 process and the boot shell process are created upon boot.

The first process is created inside the `userinit()` function, that is called from `main()`. `userinit()` allocates a new process from the process table and initializes it's user memory. Specifically it loads a small stub of code (initcode.S) that is already compiled in the kernel. This stub contains a short sequence of code that will execute the `exec()` system call passing "/init" as an argument. Exec will replace the memory of the first process with the memory image of the init program. Init will fork creating the boot shell.

(b) (5 points) At what point the first process starts executing? I.e., what is the function in the xv6 kernel that starts execution of the first process.

While the `userinit()` allocates the first process and initializes its memory it does not start running until the kernel finish initialization and enters the scheduler from the mpmain() function. The scheduler enters the scheduling loop, finds the first RUNNABLE process in the process list and context switches into it. At this process the first process starts running and invokes the `exec("/init")` system call.

3. Anatomy of a process and context switching

   Ben decides to implement user-level threads. His plan is to create a new `thread` data structure that describes a thread. He then thinks he can allocate a new stack for each thread and implement a function `u_thread_create()` which creates a new thread. The `u_thread_create()` has the following signature (it's exactly like the `thread_create()` function in our Homework 4 "Kernel Threads" assignment, but just takes an additional `thread` argument):

```
int u_thread_create(struct thread *t, void (*fn)(void *), void *arg, void *stack);
```

   The `u_thread_create()` call creates a new user thread that runs inside the same process (in contrast to the kernel thread implementation, the `u_thread_create()` doesn't invoke a single system call, but instead just executes the function that is passed as an argument (`fn`) on the already allocated stack (`stack`). The function pointed by the `fn` pointer takes a void pointer as an argument (it's passed inside `u_thread_create()` as `arg`). The new user thread runs until it explicitly yields execution back to the parent with the `u_yield()` call. The `u_yield()` call doesn't do a single system call, but saves execution of the thread on the stack and switches back to the parent process (i.e., it continues execution at the line immediately following the `u_thread_create()` invocation.

   Ben can then create and run multiple user threads like this (the `u_yield_to()` function yields execution back to a specific thread pointed by the `struct thread` argument):

```
#include <stdio.h>
#include <stdlib.h>
void do_work(void *arg) {
   int i;
   for (i = 0; i < 2; i++) {
       printf("I'm in %s\n", (char *)arg);
       u_yield();
   }
};
int main(int argc, char *argv[]) {
    void *stack1, *stack2;
    struct thread t1, t2;
    char a1[] = "Thread 1";
    char a2[] = "Thread 2";

    stack1 = malloc(4096);
    stack2 = malloc(4096);

    u_thread_create(&t1, do_work, (void*)a1, stack1);
    u_thread_create(&t2, do_work, (void*)a2, stack2);

    while(t1.state == RUNNABLE || t2.state == RUNNABLE) {
        if (t1.state == RUNNABLE)
            u_yield_to(&t1);
        if (t2.state == RUNNABLE)
            u_yield_to(&t2);
```

```
    }
    printf("Threads finished\n");
    return 0;
}
```

(a) (5 points) What output the program above will produce (assume that Ben got everything right and standard output is connected to the terminal).

The program will produce

```
I'm in Thread 1
I'm in Thread 2
I'm in Thread 1
I'm in Thread 2
...
Threads finished
```

(b) (5 points) Ben times execution of his program by adding the `uptime()` system call at the beginning and the end of the `main()` function, but doesn't see any improvement compared to a normal program (no user-level threads, just run `do_work()` twice). He then changes `do_work()` to compute factorial and other computationally intensive functions instead of simply printing on the console, and yet he sees no performance improvement. Explain why the performance stays the same although multiple user-level threads are running?

User threads that Ben created run inside the same process, hence the kernel schedules them on the same CPU and only one of them can run at every given moment in time.

(c) (15 points) Provide code for the `u_thread_create()` and `u_yield()` functions (you can use pseudocode for C and ASM as long as semantics of operations is clear).

We provide a complete working solution below, obviously a much simpler draft is be accepted. Below we provide examples for two possible implementations:

Implementation #1

```
#define RUNNABLE  1
#define EXITED    2

struct thread {
  void  *ebp;
  void  *esp;
  int   state;
};

struct thread parent;
struct thread *current;

void _uswitch(void *from, void *to) __attribute__((returns_twice));

__asm__ ("      .text                    \n\t"
         "      .align  16               \n\t"
         "      .globl  _uswitch         \n\t"
         // 8(%esp): thread_to, 4(%esp): thread_from
         "_uswitch:                      \n\t"
         "      mov 4(%esp), %eax        \n\t" // load thread_from into eax
         "      mov 8(%esp), %ecx        \n\t" // load thread_to into ecx
         "      push %ebx                \n\t" // save callee saved registers: ebx, ed
         "      push %edi                \n\t"
         "      push %esi                \n\t"
         "      movl %ebp, 0(%eax)       \n\t" // save EBP
         "      movl %esp, 4(%eax)       \n\t" // save ESP
         // Thread state is saved, switch
         "      mov 0(%ecx), %ebp        \n\t" // load thread_to's ebp into ebp
         "      mov 4(%ecx), %esp        \n\t" // load thread_to's esp into esp
         "      pop %esi                 \n\t" // restore callee saved registers
         "      pop %edi                 \n\t"
         "      pop %ebx                 \n\t"
         "      ret                      \n\t" // return
        );

void uexit(void) {
  current->state = EXITED;
  _uswitch(current, &parent);
}

void u_thread_create(struct thread *t, void *fnc, void*arg, void *stack) {
```

```
    current = t;
    t->state = RUNNABLE;
    t->esp = stack + 4096;

    // push the argument on the stack
    t->esp -= sizeof(void*);
    *(void**)t->esp = arg;

    // when fnc returns, return into uexit()
    t->esp -= sizeof(void*);
    *(void**)t->esp = uexit;

    // The new thread will return into fnc from _uswitch
     t->esp -= sizeof(void*);
    *(void**)t->esp = fnc;

    // Fake the return stack for _uswitch_save
    t->esp -= sizeof(void*);
    *(void**)t->esp = 0;     // ebx

    t->esp -= sizeof(void*);
    *(void**)t->esp = 0;     // edi

    t->esp -= sizeof(void*);
    *(void**)t->esp = 0;     // esi

    _uswitch(&parent, t);
}

void u_yield() {
    _uswitch(current, &parent);
}

void u_yield_to(struct thread *t) {
    current = t;
    _uswitch(&parent, t);
}
```

Implementation #2

```
#define RUNNABLE  1
#define EXITED    2

struct thread {
  void  *eip;
  void  *ebp;
  void  *esp;
  int   state;
};

struct thread parent;
struct thread *current;

void _uswitch(void *from, void *to) __attribute__((returns_twice));
void _uswitch_save(void *from, void *to) __attribute__((returns_twice));

__asm__ ("      .text                     \n\t"
         "      .align  16                \n\t"
         "      .globl  _uswitch_save     \n\t"
         // 8(%esp): thread_to, 4(%esp): thread_from
         "_uswitch_save:                  \n\t"
         "      mov 4(%esp), %eax         \n\t" // load thread_from into eax
         "      mov 8(%esp), %ecx         \n\t" // load thread_to into ecx
         "      push %ebx                 \n\t" // save callee registers
         "      push %edi                 \n\t"
         "      push %esi                 \n\t"
         "      push %ecx                 \n\t" // push thread_to as second arg
         "      push %eax                 \n\t" // load thread_from as first arg
         "      call _uswitch             \n\t"
         "      add   $0x8,%esp           \n\t" // release space used for args
         "      pop %esi                  \n\t" // restore callee registers
         "      pop %edi                  \n\t"
         "      pop %ebx                  \n\t"
         "      ret                       \n\t"
        );

__asm__ ("      .text                     \n\t"
         "      .align  16                \n\t"
         "      .globl  _uswitch          \n\t"
         "_uswitch:                       \n\t"
         // 8(%esp): thread_to, 4(%esp): thread_from
         "      movl 4(%esp), %eax        \n\t" // load thread_from into eax
         "      movl 0(%esp), %ecx        \n\t" // load return address into esi
         "      movl %ecx, 0(%eax)        \n\t" // EIP (our return address)
         "      movl %ebp, 4(%eax)        \n\t" // EBP
         "      movl %esp, 8(%eax)        \n\t" // ESP
```

```
        "      addl $4, 8(%eax)          \n\t" // return address + 4
        // Thread state is saved, switch
        "      mov 8(%esp), %eax         \n\t"
        "      mov 4(%eax), %ebp         \n\t"
        "      mov 8(%eax), %esp         \n\t"
        "      jmp *0(%eax)              \n\t"
       );

void uexit(void) {
  current->state = EXITED;
  _uswitch_save(current, &parent);
}

void u_thread_create(struct thread *t, void *fnc, void*arg, void *stack) {
    current = t;
    t->state = RUNNABLE;
    t->eip = fnc;
    t->esp = stack + 4096;

    t->esp -= sizeof(void*);
    *(void**)t->esp = arg;

    t->esp -= sizeof(void*);
    *(void**)t->esp = uexit;

    _uswitch_save(&parent, t);
}

void u_yield() {
    _uswitch_save(current, &parent);
}

void u_yield_to(struct thread *t) {
    current = t;
    _uswitch_save(&parent, t);
}
```

4. Memory management

   (a) (5 points) In the question above (user-level threads) Ben's code allocates memory for two stacks with `malloc()`, but it never calls `free()`. Is Ben's code causes a memory leak in the system? Support your argument.

   Not really. While the memory remains allocated until the process exits, the kernel cleans up all memory allocated by the process immediately after it terminates.

5. Fork, and console synchronization

Ben writes the following program.

```
int main(int argc, char *argv[])
{
  int pid = fork();
  char *msg = "aaa\n";

  if (pid == 0) {
        msg = "bbb\n";
        write(1, msg, 4);
        sleep(1);
  }

  write(1, msg, 4);
  sleep(1);
  wait();
  exit();
}
```

(a) (5 points) What are the possible outputs of the above program?

- aaa bbb bbb
- bbb aaa bbb
- bbb bbb aaa

(b) (10 points) Ben argues with Alice that there will never be an output with interleaving characters? E.g., "ababba" Is Ben correct? Explain your answer?

Yes. Ben is correct. The `write()` call follows the following call sequence, `write()` -> `filewrite()` -> `consolewrite()`. The function `consolewrite()` acquires the console lock such that only one thread can be writing to the console. So, there won't be any interleaving characters.

6. Synchronization

Alice creates a program to test her understanding of multithreaded locks. Below is a part of the program that describes her locking implementation.

```
struct mutex_lock m1;
struct mutex_lock m2;

void do_work(void *arg){
  mutex_lock(&m1);
  mutex_lock(&m2);
    //do something
  mutex_unlock(&m2);
  mutex_unlock(&m1);
}

void do_work2(void *arg){
  mutex_lock(&m2);
  mutex_lock(&m1);
    //do something
  mutex_unlock(&m1);
  mutex_unlock(&m2);
}

int main(int argc, char *argv[])
{
  ...
  mutex_init(&m1);
  mutex_init(&m2);
  t1 = thread_create(do_work2, (void*)&b1, s1);
  t2 = thread_create(do_work, (void*)&b2, s2);
  ...
}
```

(a) (5 points) Assuming the program compiles, do you see anything wrong with her locking strategy? Explain your answer.

Yes, the locking order is incorrect. Thread1 acquires lock in the following order: m1 followed by m2, whereas Thread2 acquires the lock m2 followed by m1. When these two threads are run in parallel, there is a possibility of a deadlock as thread1 can acquire m1 and wait for m2 and thread2 can acquire m2 and wait for m1.

7. CS238P. I would like to hear your opinions about CS238P, so please answer the following questions. (Any answer, except no answer, will receive full credit.)
   (a) (1 point) Grade CS238P on a scale of 0 (worst) to 10 (best)?

   (b) (2 points) Any suggestions for how to improve CS238P?

   (c) (1 point) What is the best aspect of CS238P?

   (d) (1 point) What is the worst aspect of CS238P?