Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (B), Spring 2022

1. **Please read the entire exam before starting to write. This should help you avoid getting bogged down on one problem.**
2. Be sure to put your name and Andrew ID below.
3. This is a closed-book in-class exam. You may not use any reference materials during the exam.
4. If you have a clarification question, please write it down on the card we have provided. Please don't ask us questions of the form "If I answered like this, would it be ok?" or "Are you looking for ...?"
5. The weight of each question is indicated on the exam. Weights of question *parts* are estimates which may be revised during the grading process and are for your guidance only.
6. Please be concise in your answers. You will receive partial credit for partially correct answers, but truly extraneous remarks may count against your grade.

Andrew Username	
Full Name	

Question	Max	Points	Grader
1.	**15**		
2.	**10**		
3.	**15**		
4.	**20**		
	60		

Please note that there are system-call and thread-library "cheat sheets" at the end of the exam.

If we cannot read your writing, we will be unable to assign a high score to your work.

I have not received advance information on the content of this 15-410/605 mid-term exam by discussing it with anybody who took part in the main exam session or via any other avenue.

Signature: _____ Date _____

Please note that there are system-call and thread-library "cheat sheets" at the end of the exam.

If we cannot read your writing, we will be unable to assign a high score to your work.

1. [15 points] Short answer.

(a) [6 points] When designing a body of code, at times one finds oneself thinking, "I wonder if I can assume X?" According to the 15-410 design orthodoxy, immediately upon having such a thought one is required to ask oneself two questions. Please state those questions. It is probably worthwhile to include specific examples and/or to briefly explain why these two replacement questions are important.

You may use this page as extra space for the “assume” question if you wish.

(b) [5 points] Register dump.

Below is a register dump produced by the "Pathos" P2 reference kernel when it decided to kill a user-space thread. Your job is to carefully consider the register dump and:

1. Determine which "wrong register value(s)" caused the thread to run an instruction which resulted in a fatal exception. You should say why/how the wrong value led to an exception, i.e., merely claiming a register has a "wrong" value will not receive full credit.
2. Briefly state the most plausible way you think that register could have taken on that value (i.e., try to describe a bug which could have this effect).
3. Then write a *small* piece of code which would plausibly cause the thread to die in the fashion indicated by the register dump. *This code does not need to implement exactly the set of steps that you identified as "most plausible" above, or result in the same register values; you should aim to achieve "basically the same effect."* Code may be written in either C or assembly language. Your code should assume execution begins in `main()`, which has been passed the typical two parameters in the typical fashion.

Please be sure that your description of the fatality and the code, taken together, clearly support your diagnosis.

```
Registers:
eax: 0x9fb102cd, ebx: 0x00000000, ecx: 0xffffe0d4,
edx: 0xffffef00, edi: 0x0000ffff, esi: 0x00000080,
ebp: 0xffffe0f0, esp: 0xffffe0c8, eip: 0x9fb102cd,
 ss:     0x002b,  cs:     0x0023,  ds:     0x002b,
 es:     0x002b,  fs:     0x002b,  gs:     0x002b,
eflags: 0x00000282
```

You may use this page for your register-dump answer.

(c) [4 points] `readline()` buffer.

In Project 1 the `readline()` call is specified as follows:

> `int readline(char *buf, int len)` –
> Reads a line of characters into a specified buffer.
>
> If the keyboard buffer does not already contain a line of input, `readline()` will spin until a line of input becomes available. If the line is smaller than the buffer, then the complete line, including the newline character, is copied into the buffer. If the length of the line exceeds the length of the buffer, only `len` characters should be copied into `buf[]`.
>
> Available characters should not be committed into `buf[]` until there is a newline character available, so the user has a chance to backspace over typing mistakes.
>
> Characters not placed into the specified buffer should remain available for other calls to `readline()` and/or `readchar()`.
>
> While a `readline()` call is active, the user should receive ongoing visual feedback in response to typing, so that it is clear to the user what text line will be returned by `readline()`.
>
> Returns: the number of characters stored into the specified buffer, or -1 if there is an error, such as `len` being invalid or unreasonably large. In the case of an error, `readline()` makes no changes to the specified buffer.

It is possible (indeed, one hopes it is frequent!) that the line typed by the user is genuinely shorter than `len` characters. Let's consider that case — maybe the line length is 1, or $len/2$, or $len - 4$. The text above is silent about what to do with the part of `buf[]` which occurs after the newline character is placed.

Please propose an addition to the text of the `readline()` specification that details what — if anything — `readline()` should be required to do with any leftover space in `buf[]`. You *must* justify your answer; a substantial fraction of the point value of the question will be awarded based on the justification. We are not expecting a long, complicated addition to the `readline()` specification, and justifications of one or two sentences may be fine.

The remainder of this page is intentionally blank.

This page is for your `readline()` solution.

2. [10 points] Faulty Mutex

In this question you will examine some mutex code submitted for your consideration by your project partner. You should assume that `atomic_exchange()` works correctly (as described below) and that the `gettid()` system call never "fails." You should also assume "traditional x86-32 memory semantics," i.e., *not* "wacky modern memory."

```
/**
 * Atomically:
 * (1) fetches the old value of the memory location pointed to by "target"
 * (2) places the value "source" into the memory location pointed to by "target"
 * Then: returns the old value (that was atomically fetched)
 *
 * Equivalently:
 *  /* START ATOMIC SEQUENCE */
 *  int previous_target = *target;
 *  *target = source;
 *  /* END ATOMIC SEQUENCE */
 *  return previous_target;
 */
extern int atomic_exchange(int *target, int source);
```

The remainder of this page is intentionally blank.

```
typedef struct {
    int last_requested;  // tid of thread who most recently requested this lock
    int last_owner;      // tid of thread who most recently acquired this lock
    int locked;          // flag indicating lock is currently held
} lock_t;

int mutex_init(lock_t* lock) {
    lock->last_requested = -1;
    lock->last_owner = -1;
    lock->locked = 0;
    return 0;
}

void mutex_lock(lock_t* lock) {
    int me = gettid();
    int before_me = atomic_exchange(&(lock->last_requested), me);

    while (lock->last_owner != before_me) {
        continue;
    }

    while (lock->locked) {
        continue;
    }

    lock->locked = 1;
    lock->last_owner = me;
}

void mutex_unlock(lock_t* lock) {
    lock->locked = 0;
}

/* ... remainder omitted, e.g., mutex_destroy() ... */
```

There is a problem with the mutex code shown above. That is, it does not ensure that all three critical-section algorithm requirements are always met. Identify a requirement which is not met and lay out a scenario which demonstrates your claim. Use the format presented in class, i.e.,

T1	T2
me = 1;	
	me = 2;

...

You may introduce temporary variables or other obvious notation as necessary to improve the clarity of your answer. *Be sure that the execution trace you provide us with is easy to read and conclusively demonstrates the claim you are making.* It is possible to answer this question with a brief, clear trace, so you should do what is necessary to ensure that you do.

This page is for your faulty-mutex solution.

You may use this page as extra space for the faulty-mutex question if you wish.

3. [15 points] Graders' Algorithm.

The Operating Systems course staff are grading a mid-term exam. Wassim and Jeremy are working together to grade a particularly tough question.

Initially the grading procedure is quite simple. Jeremy and Wassim share a large "input stack" of exams to be graded. Each grader repeatedly takes an exam off of the input stack, places it in front of himself, grades that exam's answer to their question, and then places the exam on their shared "output stack."

This procedure doesn't work for very long, however. Since the question they are grading is quite tricky, and since various students provide a wide variety of incorrect answers, sometimes one grader needs to ask the other grader to "sign off" on a proposed score. They agree to extend the grading procedure as follows: if, while grading an exam, one grader needs input from the other grader, he will write a note on the exam, place the exam next to the other grader, and then take the next exam from the input pile and continue grading. Neither Wassim nor Jeremy wishes to appear excessively demanding of the other's time, so they agree on this rule: if one grader wishes to place a half-graded exam next to the other grader, but there is already an exam pending review next to the other grader, the grader requesting a new review will not place the new exam until the one previously waiting has been reviewed. Each time a grader pulls an exam from the shared input stack and successfully grades it solo (without consulting the other grader), he will check his personal "pending review" area to see if it contains an exam. In our model, once an exam has been examined by both graders it is definitely done, i.e., there is never a case when one exam needs to be handed back and forth.

Once they settle on this protocol, Andrew decides to code it up for simulation purposes. Here is what he comes up with. Note that this simulation doesn't move exam *objects* around; instead, each exam is represented by a small integer.

```
// locked_printf() is a version of printf() with internal locking

/* Begin: exam-grading data structures */
typedef struct grader {
    volatile int waiting;
    cond_t removed;
} grader_t;

int total_exams;
mutex_t table_lock;
volatile int table_exams;  // "input stack"
volatile int graded_exams; // "output stack"
grader_t graders[2];
/* End: exam-grading data structures */

// grader-thread "body function"
void *run_grader(void *id);

// examine_exam_number() code is not shown.
// It returns a negative number when a grader can't assign a final score.
extern int examine_exam_number(int en);
```

```
int main(int argc, char *argv[])
{
    total_exams = 50; // should come from command line
    affirm(thr_init(64*1024) == 0);

    affirm(mutex_init(&table_lock) == 0);
    affirm(cond_init(&graders[0].removed) == 0);
    affirm(cond_init(&graders[1].removed) == 0);
    table_exams = total_exams; graded_exams = 0;
    graders[0].waiting = -1;
    graders[1].waiting = -1;

    affirm(thr_create(run_grader, (void*) 0) > 0);
    affirm(thr_create(run_grader, (void*) 1) > 0);
    thr_exit(0);
    exit(0); // placate compiler
}

// This can take a while, so we enter and leave with the table unlocked.
int handle_exam(int i, int j, int e)
{
    if (examine_exam_number(e) >= 0) {
        // I was able to assign a score myself!
        // Note that this is ALWAYS true if I am the second reader.
        locked_printf("Grader %d graded exam %d.\n", i, e);
        mutex_lock(&table_lock);
        ++graded_exams;
        mutex_unlock(&table_lock);
        return 1;
    } else {
        // I wrote down some comments, now my partner will finish grading.
        locked_printf("Grader %d perplexed by exam %d.\n", i, e);
        mutex_lock(&table_lock);
        while (graders[j].waiting > 0) {
            cond_wait(&graders[j].removed, &table_lock);
        }
        graders[j].waiting = e;
        mutex_unlock(&table_lock);
        locked_printf("Grader %d handed off exam %d.\n", i, e);
        return 0;
    }
}
```

```
void *run_grader(void *vid)
{
    int i = (int) vid;
    int j = 1 - i;

    while (graded_exams < total_exams) {
        int e;
        int succeeded = 1;

        mutex_lock(&table_lock);

        if (table_exams > 0) {
            e = table_exams--;
            mutex_unlock(&table_lock);
            succeeded = handle_exam(i, j, e);
            mutex_lock(&table_lock);
        }

        if (succeeded && graders[i].waiting > 0) {
            e = graders[i].waiting;
            graders[i].waiting = -1;
            mutex_unlock(&table_lock);
            cond_signal(&graders[i].removed);
            handle_exam(i, j, e);
        } else {
            mutex_unlock(&table_lock);
        }
    }

    locked_printf("Grader %d is DONE GRADING!\n", i);
    thr_exit(0);
    exit(0); // placate compiler
}
```

Suggestions for working on this problem:

1. When tracing the execution of the code, we recommend a tabular format very similar to this:

Grader 0	Grader 1
	wait;
lock;	
...	
unlock;	
signal(1)	

2. It is *strongly recommended* that you write down a draft version of any execution trace using the scrap paper provided at the end of the exam, or on the back of some other page, before you start to write your solution on the next page. If we cannot understand the solution you provide, your grade will suffer!

(a) [8 points] Unfortunately, this exam-grading procedure can deadlock. Show a *clear, convincing* execution trace that yields a deadlock (missing, unclear, or unconvincing traces will result in only partial credit).

You may use this page as extra space for the first part of the exam-grading question if you wish.

(b) [7 points] Briefly describe how to fix or restructure the code so that it does not deadlock. It is not necessary for your answer to include code for it to receive full credit if it is clear and convincing (be sure to indicate how your solution addresses one or more deadlock ingredient(s)).

4. [20 points] Event manager.

In lecture we talked about two fundamental operations in concurrent programming: brief mutual exclusion for atomic sequences (provided in Project 2 by mutexes) and long-term voluntary descheduling (provided by condition variables). As you know, these can be combined to produce higher-level objects such as semaphores or readers/writers locks, which you implemented in P2. But concurrent programs may use a variety of other synchronization objects.

In this question, you will implement a synchronization object called an "event manager." The basic idea is that there is a collection of objects, and various threads wish to operate on those objects at various times. It is fine (and actually *desirable*) if two threads can operate on two different objects in parallel, but it is disastrous if two threads operate on the same object at the same time. If two or more threads *do* try to operate on the same object in an overlapping fashion, the first thread performs the operation and is informed that it has done so; any other threads wait until the operation is complete, and then are informed that the operation was completed by the first thread. So each "event manager" object wants to allow multiple threads when possible, but also must serialize threads when necessary.

Operations on objects are represented by function pointers. It is expected that typically if two threads operate on the same target object that they will use the same function to do so, but the "event manager" does not enforce that expectation.

So there might be two functions, `void (*fuel)(void *objptr)` and `void (*orbit)(void *objptr)` which could be applied to rocket objects; the event manager would try to run `(*fuel)(rocket_A)` and `(*fuel)(rocket_B)` in parallel but would need to serialize `(*orbit)(rocket_A)` against `(*orbit)(rocket_A)`: the first thread to invoke `(*orbit)(rocket_A)` would trigger the launch sequence and liftoff and, presumably, monitor the progress of the rocket until it reached orbit; the second thread to invoke `(*orbit)(rocket_A)` would wait a while and then be told "somebody already did it."

The "event manager" doesn't "understand" anything about function pointers and "understands" object pointers only in terms of equality testing. An "event manager" doesn't know up front the identities of the objects in a set and, indeed, threads can start invoking functions on new objects, and stop invoking functions on previously-used objects, without informing the "event manager." When a thread asks an "event manager" to invoke a function on an object, the "event manager" returns a status code indicating whether it was the first thread, and performed the operation, or whether it was a non-first thread, and waited. Once all threads operating/waiting on an object are done, the "event manager" retains no memory of the object: two invocations of `(*orbit)(rocket_A)` that are separated by "long enough" will result in two launches (assuming the rocket in question is reusable and refueled, etc.).

It is expected that the number of objects that a given "event manager" object is tracking operation invocations on will typically be "fairly small," and it is ok if an "event manager" experiences "reasonable slowdown" if the number of simultaneous operation requests is atypically large.

The remainder of this page is intentionally blank.

Below are prototypes for the "event manager" functions.

```
typedef enum {
    EM_RAN,
    EM_WAITED
} em_status_t;

// Initializes a new event manager object.
// It is illegal for an application to use the event manager before
// it has been initialized or to initialize one when it is already
// initialized and in use.  em_init() shall return 0 on success or
// a negative error code on failure.
// Because this is an exam, you may assume that allocating and
// initializing the necessary state will succeed.
int em_init(em_t *em);

// Destroys the given event manager object.
// It is illegal for a program to invoke em_destroy() if any
// threads are operating on it.
void em_destroy(em_t *em);

// Checks if any operation is running on the target object.
// If so, wait until the operation is done and then return EM_WAITED.
// If not, run the operation (*fn)(target) and return EM_RAN.
em_status_t em_request(em_t *em, void (*fn)(void*), void *target);

/* ********************************************** */
/* map.h
 * You may use these functions in your solution.
 * They do NOT contain locking.
 * For the purposes of this exam, assume map functions cannot fail.
 */

// initializes a map (to empty)
int map_init(map_t *m);

// adds mapping from key, to val -- only one mapping per key is legal
void map_insert(map_t *m, void *key, void *val);

// lookup - returns non-zero if key was found and *val_ptr was filled in
int map_lookup(map_t *m, void *key, void **val_ptr);

// removes mapping from key
void map_remove(map_t *m, void *key);

// frees any resources used by map
void map_destroy(map_t *m);
```

Assumptions:

1. You may use regular Project 2 thread-library primitives: mutexes, condition variables, semaphores, readers/writer locks, etc.
2. You may *not* use other atomic or thread-synchronization synchronization operations, such as, but not limited to: `deschedule()`/`make_runnable()`, or any atomic instructions (`XCHG`, `LL/SC`).
3. You may assume that callers of your routines will obey the rules. **But you must be careful that you obey the rules as well!**
4. You must comply with the published interfaces of synchronization primitives, i.e., you cannot inspect or modify the internals of any thread-library data objects.
5. You may not use assembly code, inline or otherwise.
6. **For the purposes of the exam, you may assume that library routines and system calls don't "fail"** (unless you indicate in your comments that you have arranged, and are expecting, a particular failure).
7. You may **not** rely on any data-structure libraries such as splay trees, red-black trees, queues, stacks, or skip lists, lock-free or otherwise, that you do not implement as part of your solution. However, you **may** use the map functions documented above.
8. You may use non-synchronization-related thread-library routines in the "`thr_xxx()` family," e.g., `thr_getid()`. You may wish to refer to the "cheat sheets" at the end of the exam. If you wish, you may assume that `thr_getid()` is "very efficient" (for example, it invokes no system calls). You may also assume that condition variables are strictly FIFO if you wish.

The remainder of this page is intentionally blank.

Please declare a `struct em` and implement:

- `int em_init(em_t *em)`,
- `void em_destroy(em_t *em)`, and
- `em_status_t em_request(em_t *em, void (*fn)(void*), void *target)`

If you wish, you may also declare an auxiliary structure, `struct aux`, *but this is strictly optional.*

It is strongly recommended that you rough out an implementation on the scrap paper provided at the end of the exam, or on the back of some other page, before you write anything here. If we cannot understand the solution you provide on this page, your grade will suffer!

```
typedef struct em {









} em_t;


typedef struct aux {









} aux_t;
```

...space for “event manager” implementation...

You may use this page as extra space for your "event manager" solution if you wish.

You may use this page as extra space for your "event manager" solution if you wish.

You may use this page as extra space for your "event manager" solution if you wish.

System-Call Cheat-Sheet

```
/* Life cycle */
int fork(void);
int exec(char *execname, char *argvec[]);
void set_status(int status);
void vanish(void) NORETURN;
int wait(int *status_ptr);
void task_vanish(int status) NORETURN;

/* Thread management */
int thread_fork(void); /* Prototype for exam reference, not for C calling!!! */
int gettid(void);
int yield(int pid);
int deschedule(int *flag);
int make_runnable(int pid);
int get_ticks();
int sleep(int ticks); /* 100 ticks/sec */
typedef void (*swexn_handler_t)(void *arg, ureg_t *ureg);
int swexn(void *esp3, swexn_handler_t eip, void *arg, ureg_t *newureg):

/* Memory management */
int new_pages(void * addr, int len);
int remove_pages(void * addr);

/* Console I/O */
char getchar(void);
int readline(int size, char *buf);
int print(int size, char *buf);
int set_term_color(int color);
int set_cursor_pos(int row, int col);
int get_cursor_pos(int *row, int *col);

/* Miscellaneous */
void halt();
int readfile(char *filename, char *buf, int count, int offset);

/* "Special" */
void misbehave(int mode);
```

If a particular exam question forbids the use of a system call or class of system calls, the presence of a particular call on this list does not mean it is "always ok to use."

Thread-Library Cheat-Sheet

```
int mutex_init( mutex_t *mp );
void mutex_destroy( mutex_t *mp );
void mutex_lock( mutex_t *mp );
void mutex_unlock( mutex_t *mp );

int cond_init( cond_t *cv );
void cond_destroy( cond_t *cv );
void cond_wait( cond_t *cv, mutex_t *mp );
void cond_signal( cond_t *cv );
void cond_broadcast( cond_t *cv );

int thr_init( unsigned int size );
int thr_create( void *(*func)(void *), void *arg );
int thr_join( int tid, void **statusp );
void thr_exit( void *status );
int thr_getid( void );
int thr_yield( int tid );

int sem_init( sem_t *sem, int count );
void sem_wait( sem_t *sem );
void sem_signal( sem_t *sem );
void sem_destroy( sem_t *sem );

int rwlock_init( rwlock_t *rwlock );
void rwlock_lock( rwlock_t *rwlock, int type );
void rwlock_unlock( rwlock_t *rwlock );
void rwlock_destroy( rwlock_t *rwlock );
void rwlock_downgrade( rwlock_t *rwlock );
```

If a particular exam question forbids the use of a library routine or class of library routines, the presence of a particular routine on this list does not mean it is "always ok to use."

Ureg Cheat-Sheet

```
#define SWEXN_CAUSE_DIVIDE       0x00  /* Very clever, Intel */
#define SWEXN_CAUSE_DEBUG        0x01
#define SWEXN_CAUSE_BREAKPOINT   0x03
#define SWEXN_CAUSE_OVERFLOW     0x04
#define SWEXN_CAUSE_BOUNDCHECK   0x05
#define SWEXN_CAUSE_OPCODE       0x06  /* SIGILL */
#define SWEXN_CAUSE_NOFPU        0x07  /* FPU missing/disabled/busy */
#define SWEXN_CAUSE_SEGFAULT     0x0B  /* segment not present */
#define SWEXN_CAUSE_STACKFAULT   0x0C  /* ouch */
#define SWEXN_CAUSE_PROTFAULT    0x0D  /* aka GPF */
#define SWEXN_CAUSE_PAGEFAULT    0x0E  /* cr2 is valid! */
#define SWEXN_CAUSE_FPUFAULT     0x10  /* old x87 FPU is angry */
#define SWEXN_CAUSE_ALIGNFAULT   0x11
#define SWEXN_CAUSE_SIMDFAULT    0x13  /* SSE/SSE2 FPU is angry */

#ifndef ASSEMBLER

typedef struct ureg_t {
    unsigned int cause;
    unsigned int cr2;   /* Or else zero. */

    unsigned int ds;
    unsigned int es;
    unsigned int fs;
    unsigned int gs;

    unsigned int edi;
    unsigned int esi;
    unsigned int ebp;
    unsigned int zero;  /* Dummy %esp, set to zero */
    unsigned int ebx;
    unsigned int edx;
    unsigned int ecx;
    unsigned int eax;

    unsigned int error_code;
    unsigned int eip;
    unsigned int cs;
    unsigned int eflags;
    unsigned int esp;
    unsigned int ss;
} ureg_t;

#endif /* ASSEMBLER */
```

Useful-Equation Cheat-Sheet

$$\cos^2\theta + \sin^2\theta = 1$$

$$\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta$$

$$\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta$$

$$\sin 2\theta = 2\sin\theta\cos\theta$$

$$\cos 2\theta = \cos^2\theta - \sin^2\theta$$

$$e^{ix} = \cos(x) + i\sin(x)$$

$$\cos(x) = \frac{e^{ix} + e^{-ix}}{2}$$

$$\sin(x) = \frac{e^{ix} - e^{-ix}}{2i}$$

$$\int \ln x\, dx = x\ln x - x + C$$

$$\int_0^\infty \sqrt{x}\, e^{-x}\, dx = \frac{1}{2}\sqrt{\pi}$$

$$\int_0^\infty e^{-ax^2}\, dx = \frac{1}{2}\sqrt{\frac{\pi}{a}}$$

$$\int_0^\infty x^2 e^{-ax^2}\, dx = \frac{1}{4}\sqrt{\frac{\pi}{a^3}} \text{ when } a > 0$$

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dt$$

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},\, t) = \hat{H}\Psi(\mathbf{r}, t)$$

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},\, t) = -\frac{\hbar^2}{2m}\nabla^2\Psi(\mathbf{r},\, t) + V(\mathbf{r})\Psi(\mathbf{r},\, t)$$

$$E = hf = \frac{h}{2\pi}(2\pi f) = \hbar\omega$$

$$p = \frac{h}{\lambda} = \frac{h}{2\pi}\frac{2\pi}{\lambda} = \hbar k$$

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\varepsilon_0}$$

$$\nabla \cdot \mathbf{B} = 0$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

$$\nabla \times \mathbf{B} = \mu_0\mathbf{J} + \mu_0\varepsilon_0\frac{\partial \mathbf{E}}{\partial t}$$

If you wish, you may tear this page off and use it for scrap paper. But be sure not to write anything on this page which you want us to grade.

If you wish, you may tear this page off and use it for scrap paper. But be sure not to write anything on this page which you want us to grade.