Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Spring 2014

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 *and also* put your Andrew ID at the top of *each* following page.
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.
7. **Write legibly even if you must slow down to do so!** If you spend some time to *think clearly* about a problem, you will probably have time to write your answer legibly.

Andrew Username	
Full Name	

Question	Max	Points	Grader
1.	**10**		
2.	**15**		
3.	**20**		
4.	**15**		
5.	**10**		
	70		

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

1. [10 points] Short answer.

Define or explain the terms listed below. We are expecting each part of this question to be answered by three to six sentences. Your goal is to make it clear to your grader that you understand the concept *as it applies to this course* and can apply it when necessary. It is typically helpful for you to include an example or two of the concept you are discussing.

(a) [5 points] Deadlock prevention.

(b) [5 points] User mode.

2. [15 points] Deadlock.

It's spring... which means that before long it'll be summer and you'll be working at WhatSnappy-Chatter, a new social-messaging company which is being bought by a large social-media company for $14 quintillion. But there is a problem... before the sale can go through, they need a hotshot CMU student to debug a problem that is blocking the development team.

The company's "app" has some strange architectural features. For example, the app has a small fixed number of threads, and `thr_getid()` has been hacked so it always returns a value between 0 and `NUM_THREADS-1`. You will be working on an internal messaging system which enables these threads to send and receive small messages with each other (for the purposes of this exam, we will assume that the value contained in each message is an `int`—boring, but simple). For some reason which isn't clear to you (maybe a patent??), a core feature of the messaging system is that send calls block until the data has been received by the receiver thread, but it is *very important* for receive operations to return *very quickly* whether or not anything is waiting.

Here are some details about the implementation:

- You may assume that there are only `NUM_THREADS` threads using this system and their thread id's range between 0 and `NUM_THREADS-1`.
- `send_message(msg, to)` should be used to send data (in this case an int) to another thread, and should block until the recieving thread calls `recv_message()`.
- Each thread is given an "inbox" and an "outbox" in two global arrays; each thread's inbox and outbox are located by using the owning thread's id as an index.
- If thread i wishes to send a message to thread j, it will first put the data into its outbox, and then put its thread id in thread j's inbox.
- For thread j to receive this message, it will check its own inbox, see i's thread id, and then find the data in thread i's outbox. At this point the message has been passed successfully.
- In order to avoid deadlock, a thread is not allowed to send a message to itself.
- In order to avoid deadlock, `send_message()` will return an error if thread i tries to send a message to thread j while it already has a pending message from thread j (i.e. thread j is already waiting for thread i to pick up a message it sent).

When reading the code below, you should assume all library and system calls return normally, and the `mailboxes_init()` call is successful.

The remainder of this page is intentionally blank.

```
#define NUM_THREADS 10

typedef struct {
    sem_t sender_lock;
    mutex_t data_lock;
    int from;
} inbox;

typedef struct {
    sem_t send_complete;
    int msg;
} outbox;

inbox inboxes[NUM_THREADS];
outbox outboxes[NUM_THREADS];

int mailboxes_init()
{
    int i;
    for (i = 0; i < NUM_THREADS; i++)
    {
        // assume initialization functions cannot fail
        sem_init(&inboxes[i].sender_lock, 1);
        mutex_init(&inboxes[i].data_lock);
        inboxes[i].from = -1;

        sem_init(&outboxes[i].send_complete, 0);
        outboxes[i].msg = 0;
    }
    return 0;
}
```

```
int recv_message(int *msg)
{
    int my_tid = thr_getid();

    mutex_lock(&inboxes[my_tid].data_lock);
    int sender = inboxes[my_tid].from;
    mutex_unlock(&inboxes[my_tid].data_lock);

    if (sender != -1)
    {
        inboxes[my_tid].from = -1;
        *msg = outboxes[sender].msg;
        sem_signal(&outboxes[sender].send_complete);
        return sender;
    }
    return -1;
}

int send_message(int msg, int to)
{
    int my_tid = thr_getid();

    // DL: a thread is not allowed to send messages to itself
    if (to == my_tid)
    {
        return -1;
    }

    outboxes[my_tid].msg = msg;

    sem_wait(&inboxes[to].sender_lock);
    mutex_lock(&inboxes[to].data_lock);

    inboxes[to].from = my_tid;

    mutex_unlock(&inboxes[to].data_lock);

    // DL: fail to send message if recipient is already waiting on us to read
    mutex_lock(&inboxes[my_tid].data_lock);
    if (inboxes[my_tid].from == to)
    {
        inboxes[to].from = -1;
        mutex_unlock(&inboxes[my_tid].data_lock);
        sem_signal(&inboxes[to].sender_lock);
        return -1;
    }
    mutex_unlock(&inboxes[my_tid].data_lock);

    sem_wait(&outboxes[my_tid].send_complete);
    sem_signal(&inboxes[to].sender_lock);

    return 0;
}
```

You have been called in because of a disagreement in the development team. In particular, the author of the messaging library claims that, no matter how it is used by its client threads, it won't "internally deadlock": either the send and receive operations will complete in a reasonable way, or at least one thread invoking `send_message()` will receive a return code of -1.

Unfortunately, the code shown above *can* deadlock. Show *clear, convincing* evidence of deadlock. Your evidence should include a "tabular execution trace," a well-annotated process/resource graph, or both. Missing, unclear, or unconvincing traces will result in only partial credit. Note that `send_message()` returning an error code does *not* count as deadlock.

You may use this page as extra space for the deadlock question if you wish.

3. [20 points] "Channel locking"

In your Project 2 thread library, you implemented various standard synchronization primitives such as mutexes, condition variables, semaphores, reader/writer locks. These primitives can be used to implement higher-level constructs such as inter-thread mailboxes, as discussed in an earlier question. However, a different view is recently gaining in popularity: in this view, a concurrency-management system should provide a rich set of thread-safe message-transfer operations, and threads should synchronize *only* by sending messages, not by using crude "mutexes" and "condition variables." Thinking similar to this is part of the rationale of two recent programming languages, Go and Rust.

In this question you will be asked to build a "backwards compatibility layer" which implements mutexes and condition variables in terms of an existing channel library.

- `int chan_init(chan_t *chan, int capacity)` — initializes the channel
- `void chan_send(chan_t *chan, void *data)` — sends the value `data` on channel `chan`
- `void* chan_recv(chan_t *chan)` — consumes one message sent by `chan_send`
- `void chan_destroy(chan_t *chan)` — destroys the channel

Notice that `chan_init()` takes a capacity as an argument. This capacity argument corresponds to the size of an internal buffer (measured in messages, not bytes) that the channel manages. If this capacity is zero, then the channel is initialized as a *fully synchronous* channel. This means that whether a thread is doing `chan_send()` or `chan_recv()`, that thread will block until there is a thread performing the opposite operation on the same channel. For example, if `T0` calls `chan_send()` on a channel initialized with capacity `0`, then `T0` will block until `T1` calls `chan_recv()` on the same channel; then both threads will return.

On the other hand, if the capacity is non-zero, then the channel is initialized as "mostly" asynchronous. As long as the internal buffer is not full, `chan_send()` will simply copy the value into the buffer and return, most likely before a thread has even begun to receive the value. If `chan_send()` encounters a full buffer it will block. Some noteworthy points about channels:

- `chan_send()` will always block if capacity is set to 0 and a corresponding `chan_recv()` isn't already blocked waiting for it
- If capacity is non-zero, then a return from `chan_send()` does not imply a corresponding `chan_recv()` has occured yet
- `chan_recv()` will block unless/until either a buffered value or a corresponding `chan_send()` operation is available, regardless of the capacity set in `chan_init()`
- You may assume that channels are strictly FIFO if you wish.
- Each message sent is received by at most one receiver.

Your mission has two parts. You will first implement mutexes while using just channels as your synchronization primitives. Then you will implement condition variables using just channels and/or mutexes. You may *not* use other atomic or thread-synchronization operations, such as, but not limited to: reader/writer locks, `deschedule()`/`make_runnable()`, or any atomic instructions (`XCHG`, `LL/SC`). 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. **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).

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 on these pages. If we cannot understand the solution you provide here, your grade will suffer!

(a) [5 points] Please declare your `struct mutex_t` here. If you need an auxilary structure, you may fill that in as well

```
typedef struct mutex {









} mutex_t;

typedef struct m_aux {




} m_aux_t;
```

(b) [5 points] Now please write `mutex_init()`, `mutex_destroy()`, `mutex_lock()` and `mutex_unlock()`

...space for mutex implementation...

(c) [5 points] Please declare your `struct cond_t` here. If you need an auxilary structure, you may fill that in as well. Regardless of what you have written for the mutex implementation, we will grade your condition-variable solution under the assumption that your mutex implementation is correct.

```
typedef struct cond {









} cond_t;

typedef struct aux {



} aux_t;
```

(d) [5 points] Now please write `cond_init()`, `cond_destroy()`, `cond_signal()`, `cond_wait()` and `cond_broadcast()`

...space for condition variable implementation...

4. [15 points] Peterson's Algorithm

In this question you will examine some critical-section protocol code and imagine how it might be executed on a somewhat strange processor. First, the code. *In the assembly-language listing, the two-digit hex numbers in brackets show the address of the first byte of each instruction; we will use these addresses to refer to specific instructions.*

```
volatile int want[2] = { 0, 0 };
volatile int turn = 0;

void peterson_enter(int i)
{
    int j = 1 - i;

    want[i] = 1;
    turn = j;

    while (want[j] && (turn == j))
        continue;
    return; // critical section acquired!
}
void peterson_leave(int i)
{
    want[i] = 0;
}
```

```
        .bss
        .align 4
want:
        .zero   8
turn:
        .zero   4

        .text
        .globl  peterson_enter
peterson_enter:
[00]    pushl   %ebp
[01]    movl    %esp, %ebp
[03]    pushl   %ecx
[04]    pushl   %edx
[05]    movl    8(%ebp), %ecx
[08]    movl    $1, %edx
[0d]    subl    %ecx, %edx
[0f]    movl    $1, want(,%ecx,4)
[1a]    movl    %edx, turn
.L1:
[20]    movl    want(,%edx,4), %eax
[27]    testl   %eax, %eax
[29]    je      .L2
[2b]    movl    turn, %eax
[30]    cmpl    %edx, %eax
[32]    je      .L1
.L2:
[34]    popl    %edx
[35]    popl    %ecx
[36]    popl    %ebp
[37]    ret

        .globl  peterson_leave
peterson_leave:
[38]    pushl   %ebp
[39]    movl    %esp, %ebp
[3b]    movl    8(%ebp), %eax
[3e]    movl    $0,want(,%eax,4)
[49]    popl    %ebp
[4a]    ret
```

For the purposes of this question, you may assume the code above works on single-processor x86-32 machines. Part of the reason it *does* work is a feature of most modern CPU architectures, namely "precise interrupts." If the code presented above is interrupted by a clock tick, the hardware will include, in the trap frame pushed on the appropriate stack, a value of `%EIP`. This value represents a guarantee from the hardware that all instructions prior to `%EIP` have *completed* execution and that the instruction pointed at by `%EIP`, as well as all instructions subsequent to that instruction *have not executed*. The idea is that, at some later point, the saved value of `%EIP` will be restored into the actual program counter, which will cause the interrupted code stream to resume where it left off. In a sense, the interrupt "happened between" two instructions, and the `%EIP` value stored on the stack is the address of the instruction after the interrupt happened.

While the interrupt model described above is convenient for programmers, it is based on a *very* old-fashioned model of processor architecture, in which the processor first fetches an instruction; then figures out what needs to be done; then does it; then updates the program counter to point to the subsequent instruction; and finally checks to see whether an interrupt is pending.

However, as you are probably aware, modern processors execute multiple instructions in parallel (typically at least two, but often four or more). As a result, when an interrupt is detected some instructions have been fetched but not yet begun execution; some instructions have been fetched and are executing; and of course some instructions have been fetched and executed to completion. When different instructions take different amounts of time to complete, it can happen that an instruction later in program order has finished even though an instruction earlier in program order hasn't yet. For example, `[05] movl 8(%ebp), %ecx` might take a while since it needs to get something from RAM, but it appears before `[08] movl $1, %edx` which is trivial to execute, so it is possible that the second `MOVL` might complete before the first.

Modern processors which support "precise interrupts" (including the x86!) contain elaborate logic that can delay the launch of some instructions and *revert the effects* of others, so that when an interrupt is detected it is possible to cleanly split the instruction stream into a leading sequence of instructions that are 100% complete, plus `%EIP` pointing at the beginning of a sequence of instructions that appear not to have executed at all.

However, some machines in the 1990's were manufactured with a different interrupt model, called "imprecise interrupts." When an imprecise-interrupts processor begins an interrupt handler, the hardware has stored, instead of a single `%EIP` value describing all completed and not-started instructions, a *list* of instructions and the completion status of each! On such a machine the equivalent of `IRET` loads this *list* back into the processor; the processor will execute the not-yet-executed instructions on the list, skip the already-executed instructions, and then resume regular operation.

In this problem we will assume that the processor can execute up to four instructions in parallel; when an interrupt or exception occurs, the processor will thus save *four* `%EIP` values along with a true/false flag indicating which instructions have completed. Here is a simple case.

Instruction	Done?
[04]	Yes
[05]	No
[08]	Yes
[0d]	No

For the purposes of this exam we will assume that the four instructions reported on are sequential in program order.

When the state table above is used to resume the interrupted thread, the processor will skip the `PUSHL` instruction that was previously completed, will perform the `MOVL` instruction which was not formerly completed, will skip the `MOVL` instruction that *was* completed, and will perform the `SUBL` instruction that was not formerly completed. From the point of view of that thread, all instructions appear to happen in order. True, there was a period of time during which one of the instructions was done "too soon," but in some sense this is "ok" because the thread *wasn't running* during that time, and any time it *is* running its instructions are completed in order as far as it can tell.

Note that we are assuming the processor properly delays instructions in order to avoid conditions that ECE people call "data hazards" and "control hazards." For example, when the instruction `[08] movl $1, %edx` is followed by `[0d] subl %ecx, %edx`, obviously the processor must delay the launch of the `SUBL` instruction until the result of the `MOVL` is available, because the `MOVL` places a value in `%EDX` which must present before the `SUBL` can operate on it in `%EDX`. However, when the instruction `[0f] movl $1, want(,%ecx,4)` is followed by the instruction `[1a] movl %edx, turn` it is not necessary to delay the second instruction because the instructions are using different registers and different memory locations, so neither one depends on the result of the other. Hazards are complicated, but luckily for this question your intuition about which instructions fundamentally cannot be run in parallel should be sufficient.

There is a problem with the Peterson's solution code shown above when it is run on a single-processor machine with imprecise interrupts. In this environment, the code 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. In your solution, you should assume two threads sharing a single processor interrupted only by timer ticks—but you may declare an interrupt whenever you want, regardless of whether nor not a fixed interval has occurred since the last interrupt.

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

Note that this question has two parts.

Note that this question has two parts.

Note that this question has two parts.

The remainder of this page is intentionally blank.

(a) [10 points] In a sentence or two, *identify* the critical-section property you will demonstrate the violation of, and briefly *summarize* how imprecise interrupts will cause that property to fail. Then show a *two-thread trace* which backs up your claim. Your trace does not need to show every instruction; you may use an address range to indicate when a sequence of instructions is executed in order. For example, `[38-4a]` would indicate that a thread executed all of `peterson_leave()` without being interrupted. When you do declare an interrupt, be sure you specify four `%EIP` values and a corresponding completion-status value for each. Here is an example.

Thread 0	Thread 1
00-01	
intr: 03:y 04:n 05:n 08:n	
	00-04
...	...

You may use this page as extra space for the "imprecise Peterson's" trace if you wish.

(b) [5 points] Suggest a way to fix the problem you identified in your trace. You may invent a new kind of instruction if you wish.

5. [10 points] Process model.

In this question we will discuss whether various Pebbles system calls are expected to block threads or are expected to generally complete without blocking. In order to clarify the issue, you should probably imagine that a Pebbles-compliant kernel is running on multiple processors (this actually happened during Spring 2012 and might happen again). Also note that "block" is not the same concept as "might require a lock"—as you will soon experience directly, *many* system calls require some locking.

(a) [2 points] Explain briefly what it means for a system call to block a thread, or for a thread to be blocked in a system call.

For each Pebbles system call listed below (in alphabetical order), briefly argue that the system call either *should generally not block threads* (for some stated reason(s)) or *is expected to block threads* (in some specified scenario(s)).

(b) [2 points] `deschedule()`

(c) [2 points] `gettid()`

(d) [2 points] `make_runnable()`

(e) [2 points] `swexn()`

You may use this page as extra space for the blocking question 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."

Andrew ID: _____

Typing Rules Cheat-Sheet

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \to \tau \mid \mu\alpha.\tau \mid \forall\alpha.\tau \\ e & ::= & x \mid \lambda x{:}\tau.e \mid e\,e \mid \mathsf{fix}(x{:}\tau.e) \mid \mathsf{fold}_{\alpha.\tau}(e) \mid \mathsf{unfold}(e) \mid \Lambda\alpha.e \mid e[\tau] \end{array}$$

$$\frac{}{\Gamma,\, \alpha\,\mathbf{type} \vdash \alpha\,\mathbf{type}}\ \text{istyp-var} \qquad \frac{\Gamma \vdash \tau_1\,\mathbf{type} \quad \Gamma \vdash \tau_2\,\mathbf{type}}{\Gamma \vdash t_1 \to t_2\,\mathbf{type}}\ \text{istyp-arrow}$$

$$\frac{\Gamma,\, \alpha\,\mathbf{type} \vdash \tau\,\mathbf{type}}{\Gamma \vdash \mu\alpha.\tau\,\mathbf{type}}\ \text{istyp-rec} \qquad \frac{\Gamma,\, \alpha\,\mathbf{type} \vdash \tau\,\mathbf{type}}{\Gamma \vdash \forall\alpha.\tau\,\mathbf{type}}\ \text{istyp-forall}$$

$$\frac{}{\Gamma,\, x : \tau \vdash x : \tau}\ \text{typ-var} \qquad \frac{\Gamma,\, x : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash \tau_1\,\mathbf{type}}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ \text{typ-lam} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\,e_2 : \tau_2}\ \text{typ-app}$$

$$\frac{\Gamma,\, x : \tau \vdash e : \tau \quad \Gamma \vdash \tau\,\mathbf{type}}{\Gamma \vdash \mathsf{fix}(x{:}\tau.e) : \tau}\ \text{typ-fix}$$

$$\frac{\Gamma \vdash e : [\mu\alpha.\tau/\alpha]\tau \quad \Gamma,\, \alpha\,\mathbf{type} \vdash \tau\,\mathbf{type}}{\Gamma \vdash \mathsf{fold}_{\alpha.\tau}(e) : \mu\alpha.\tau}\ \text{typ-fold} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \mathsf{unfold}(e) : [\mu\alpha.\tau/\alpha]\tau}\ \text{typ-unfold}$$

$$\frac{\Gamma,\, \alpha\,\mathbf{type} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \text{typ-tlam} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau \quad \Gamma \vdash \tau'\,\mathbf{type}}{\Gamma \vdash e[\tau'] : [\tau'/\alpha]\tau}\ \text{typ-tapp}$$

$$\frac{}{\lambda x{:}\tau.e\,\mathbf{value}}\ \text{val-lam} \qquad \frac{}{\mathsf{fold}_{\alpha.\tau}(e)\,\mathbf{value}}\ \text{val-fold} \qquad \frac{}{\Lambda\alpha.\tau\,\mathbf{value}}\ \text{val-tlam}$$

$$\frac{e_1 \;\mapsto\; e_1'}{e_1\,e_2 \;\mapsto\; e_1'\,e_2}\ \text{steps-app}_1 \qquad \frac{e_1\,\mathbf{value} \quad e_2 \;\mapsto\; e_2'}{e_1\,e_2 \;\mapsto\; e_1\,e_2'}\ \text{steps-app}_2$$

$$\frac{e_2\,\mathbf{value}}{(\lambda x{:}\tau.e_1)\,e_2 \;\mapsto\; [e_2/x]e_1}\ \text{steps-app-}\beta$$

$$\frac{}{\mathsf{fix}(x{:}\tau.e) \;\mapsto\; [\mathsf{fix}(x{:}\tau.e)/x]e}\ \text{steps-fix}$$

$$\frac{e \;\mapsto\; e'}{\mathsf{unfold}(e) \;\mapsto\; \mathsf{unfold}(e')}\ \text{steps-unfold}_1 \qquad \frac{}{\mathsf{unfold}(\mathsf{fold}_{\alpha.\tau}(e)) \;\mapsto\; e}\ \text{steps-unfold}_2$$

$$\frac{e \;\mapsto\; e'}{e[\tau] \;\mapsto\; e'[\tau]}\ \text{steps-tapp}_1 \qquad \frac{}{(\Lambda\alpha.e)[\tau] \;\mapsto\; [\tau/\alpha]e}\ \text{steps-tapp}_1$$

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.