Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Fall 2016

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.	**15**		
4.	**20**		
5.	**10**		
	70		

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

Andrew ID: _____

After I leave this exam session, I will not discuss the contents of this 15-410/605 midterm with *anybody*, whether or not in this class, whether or not present in this exam session with me, before 18:00 on Wednesday, October 19th.

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. [10 points] Short answer.

   (a) [5 points] Briefly explain the differences between Pebbles tasks and threads.

(b) [5 points] Define "thread-safe function" and explain the specific properties that make a function thread-safe.

2. [15 points] Faulty Condition Variables

As leader of the "Free the UC Stones" movement sweeping Carnegie Mellon, you are running a covert operation in which a team of threads cooperates to insert and remove virtual stones into and out of a bucket. Each team is configured with a balanced set of parameters— maybe one insertion thread adds 20 stones, while two removal threads remove 10 stones each; or maybe two insertion threads each add 15 stones, while one removal thread removes 30 stones. Meanwhile, the size of the bucket (buffer) is an independent parameter: maybe it's a 5-stone bucket/buffer, maybe it's 10 stones—whatever. Of course you recognize this as an instance of the classic "producer/consumer" prpoblem, and write up some sample code using condition variables.

Meanwhile, your OS partner recently attended a tech talk given by a company called Purple Valley. Purple Valley has a real-time micro-kernel which makes kernel-implemented semaphores available to user-space programs via `sem_wait()` and `sem_signal()` primitives. Your partner bridges the gap between your cvar-based code and the semaphore-based kernel by coding up some semaphore-based cvars; the semaphores are used as a tool to block and unblock threads.

Of course, something goes wrong, and your partner is asleep, so you need to find the problem. You are to assume that the producer/consumer code shown below is correct, and also assume that the Purple Valley real-time micro-kernel semaphores are correct, i.e., you are to assume the problem is in your partner's cvar code.

Refer to the following code for the producer-consumer problem. Again, assume it works correctly.

The remainder of this page is intentionally blank.

```
#define BUCKET_SIZE 5       // or something else
#define LIFTING_SLEEP 7     // or something else
#define DISPOSAL_SLEEP 9    // or something else

static mutex_t count_lock; // correctly initialized elsewhere
static int nstones = 0;
static cond_t removed;     // correctly initialized elsewhere
static cond_t added;       // correctly initialized elsewhere

void *insertion_thread(void *vstones) {
    int stones = (int) vstones;
    for (int s = 0; s < stones; s++) {
        /* Pick up a stone.  Heavy!  Takes a while. */
        sleep(LIFTING_SLEEP);
        /* Now wait until there is an empty slot, insert stone, notify. */
        mutex_lock(&count_lock);
        while (nstones == BUCKET_SIZE)
            cond_wait(&removed, &count_lock);
        ++nstones;
        cond_signal(&added);
        mutex_unlock(&count_lock);
    }
    return (0);
}

void *removal_thread(void *vstones) {
    int stones = (int) vstones;
    for (int s = 0; s < stones; s++) {
        /* Wait for a stone, claim it, notify. */
        mutex_lock(&count_lock);
        while (nstones == 0)
            cond_wait(&added, &count_lock);
        --nstones;
        cond_signal(&removed);
        mutex_unlock(&count_lock);
        /* Dispose of stone.  Heavy!  Takes a while. */
        sleep(DISPOSAL_SLEEP);
    }
    return (0);
}
```

Now refer to the condition-variables code below—do *not* assume this code works correctly! Note that certain functions, such as `cond_broadcast()` and `cond_destroy()`, are not relevant to this problem, and so their implementation has been omitted.

```
/* cond.c */

typedef struct cond {
  mutex_t waiters_lock;
  int waiters;
  sem_t s;
} cond_t;

int cond_init(cond_t *cv) {
    mutex_init(&cv->waiters_lock);
    sem_init(&cv->s, 0); // 0 for blocking purposes
    cv->waiters = 0;
    return (0);
}

void cond_wait(cond_t *cv, mutex_t *mp) {
    mutex_unlock(mp);
    mutex_lock(&cv->waiters_lock);
    ++cv->waiters;
    mutex_unlock(&cv->waiters_lock);
    sem_wait(&cv->s); // block
    mutex_lock(mp);
}

void cond_signal(cond_t *cv) {
    mutex_lock(&cv->waiters_lock);
    if (cv->waiters) {
        --cv->waiters;
        sem_signal(&cv->s); // unblock one
    }
    mutex_unlock(&cv->waiters_lock);
}
```

(a) [12 points] Depending on the exactly how/when the insertion and removal threads invoke the `cond_signal()` and `cond_wait()` shown above, one or more condition variables end up working incorrectly and insertion and/or removal threads can get stuck. Please construct a trace of how a thread can get stuck. If possible, try to use one insertion thread (called "I") and one extractor thread (called "E"), though we will consider traces involving other sets of threads. Use the execution trace format presented in class, e.g.,

I	E
–waiters	
	++waiters

At the bottom of your trace, summarize in a sentence or two what "the problem" is.

You may use this page for your cvar trace if you wish.

(b) [3 points] The UC Stones can be freed only with your help! Write code that replaces the area of problematic implementation which you identified in the condition variable code. We are expecting under 15 lines of code (though we will consider any solution you provide).

3. [15 points] Deadlock.

In a parallel universe, the 15-410 book report assignment (remember that???) requires students to do research using a small library maintained by the course staff. In this library, a book may refer to material that is more thoroughly discussed in some other book, so students may need to check out multiple books to fully understand a concept and complete the assignment. Each student is instructed to research some topic chosen from the many described in the official course textbook, *Operating Systems: Design and Implementation*, using the related books as additional sources.

The good news is that the course library has a copy of the main textbook for each student. Sadly, due to budget constraints, the library has only a few copies of each of the other books. In the 15-410 library, all copies of a single book are stored in a "pile" of books of that type—there is one pile of the official course textbook, one pile of the book on scheduling algorithms, etc.

To ensure that the book report project would run smoothly, the TAs wrote a multi-threaded simulation of the student/library system. Some notes on the simulation are below.

- Each distinct book owned by the library is assigned an `id` from 0 to `NTITLES-1`.
- The library stores books that are not currently checked out in "piles." There are `NTITLES` piles, each storing all checked-in copies of a given book.
- Each student is simulated by a thread that concurrently interacts with the library. All students follow the same algorithm.
- At various times, a student may check out a book or return a book, but at any point in time no student may have more than `BPSIZE` books checked out at once ("BPSIZE" stands for "backpack size").
- At the start of the semester, every student checks out the course textbook (`id = 0`).
- At various points in time, if a student has fewer than `BPSIZE` books checked out, the student will check out a book referenced by a book that student is currently reading.
- Once a student has `BPSIZE` books checked out, the student will return the book checked out the furthest in the past (books are returned in FIFO order).

The code for the simluation follows.

The remainder of this page is intentionally blank.

```
#define NUMSTUDENTS 20
#define NTITLES 4
#define BPSIZE (NTITLES - 1)
#define MAX_REFERENCES (NTITLES - 1) // references per book
#define MAX_PILE_SIZE NUMSTUDENTS
#define MAX_PILES NTITLES

typedef struct {
    unsigned int id;
    const char *title;
    unsigned numreferences;
    unsigned references[MAX_REFERENCES];
} book_t;

typedef struct {
    mutex_t lock;
    unsigned int size;
    book_t books[MAX_PILE_SIZE];
    cond_t returned;
} pile_t;

pile_t library[MAX_PILES];

void add_to_pile(pile_t *pile, book_t book) {
    mutex_lock(&pile->lock);
    assert(pile->size < MAX_PILE_SIZE);
    pile->books[pile->size++] = book;
    cond_signal(&pile->returned);
    mutex_unlock(&pile->lock);
}

book_t remove_from_pile(pile_t *pile) {
    book_t book;
    mutex_lock(&pile->lock);
    while (pile->size == 0) {
        cond_wait(&pile->returned, &pile->lock);
    }
    book = pile->books[--pile->size];
    mutex_unlock(&pile->lock);
    return book;
}
```

```
void library_init() {
    // init pile data structures
    for (int i = 0; i < MAX_PILES; i++) {
        mutex_init(&library[i].lock);
        cond_init(&library[i].returned);
        library[i].size = 0;
    }

    book_t books[NTITLES] = {
        {
            .id = 0,
            .title = "Operating Systems: Design and Implementation",
            .numreferences = 3,
            .references = { 1, 2, 3 }
        }, {
            .id = 1,
            .title = "Synchronization",
            .numreferences = 1,
            .references = {2}
        }, {
            .id = 2,
            .title = "Scheduling Algorithms",
            .numreferences = 1,
            .references = {3}
        }, {
            .id = 3,
            .title = "Interprocess Communication",
            .numreferences = 1,
            .references = {1}
        }
    };
    unsigned counts[NTITLES] = { NUMSTUDENTS, 2, 2, 2 };

    for (int p = 0; p < NTITLES; p++)
        for (int b = 0; b < counts[p]; b++)
            add_to_pile(&library[p], books[p]);
}

book_t checkout_book(int id) {
    return remove_from_pile(&library[id]);
}

void return_book(book_t book) {
    add_to_pile(&library[book.id], book);
}
```

```
void *student(void *arg) {
    unsigned int checked_out;
    book_t books[BPSIZE];
    int holding[NTITLES] = {0}; // boolean: do we have a copy of each title?

    checked_out = 0;
    books[checked_out++] = checkout_book(0);

    /* study tirelessly */
    while (1) {
        if (checked_out < BPSIZE) {
            /* check out some reference we don't currently have */
            for (int b = 0; b < checked_out; b++) {
                for (int r = 0; r < books[b].numreferences; r++) {
                    if (!holding[books[b].references[r]]) {
                        books[checked_out++] =
                            checkout_book(books[b].references[r]);
                        holding[books[b].references[r]] = 1;
                        goto done;
                    }
                }
            }
        } else {
            /* Return first book and slide the others to front */
            holding[books[0].id] = 0;
            return_book(books[0]);
            for (int b = 1; b < BPSIZE; b++)
                books[b-1] = books[b];
            checked_out--;
        }
done:
        continue;
    }
}

int main() {
    thr_init(PAGE_SIZE);
    library_init();
    for (int i = 0; i < NUMSTUDENTS; i++) {
        thr_create(student, (void *)0);
    }

    while(1) {
        yield(-1);
    }
}
```

(a) [10 points] Unfortunately, the code shown above can deadlock. Show *clear, convincing* evidence of deadlock. Begin by *describing the problem* in one or two sentences; then *clearly specify a scenario*. Your description should state, for example, the minimum number of threads needed to form the deadlock you envision, and should justify that number; you should describe which books are held/requested by which threads, and justify your claims (perhaps, but not necessarily, through the use of one or more traces).

If you cannot describe a particular exact deadlock, or are having trouble describing how it would occur, you may receive partial credit by describing which deadlock ingredients are and/or are not exhibited by the code above. *It is to your advantage to use scrap paper or the back of some page to experiment with draft traces, so that the answer you write below is easy for us to read.*

You may use this page for your deadlock answer if you wish.

(b) [5 points] Explain in detail (though code is *not* required!) how the course staff could prevent the students from deadlocking during their research. Be sure to explain (in a theoretical / conceptual sense) why your solution works. *Solutions judged as higher-quality by your grader will receive more points.* This means that it is probably better to "genuinely fix" some problem than to replace a sensible assumption/parameter with an unrealistic assumption/parameter, though we will consider any solution you clearly describe.

4. [20 points] Condition locks.

In lecture we talked about two fundamental operations in concurrent programming: brief mutual exclusion for atomic sequences (provided in P2 by mutexes) and long-term voluntary descheduling (provided by condition variables). In this question you will implement "condition locks," a synchronization object that combines (or, perhaps, conflates), these two notions.

Like mutexes, condition locks prevent multiple threads from simultaneously executing a critical section. Unlike traditional mutexes, however, threads attempting to acquire a condition lock pass in a "predicate function" describing a condition that must hold before the cond lock can be acquired. Thus, for example, a consumer thread in a producer/consumer system might specify a predicate that returns true when a buffer contains at least one item; that would result in the consumer waiting/stalling until the buffer contains an item and then acquiring the buffer lock; presumably the consumer would copy an item out and then drop the lock. Meanwhile, a producer thread might use a different predicate function that returns true when the buffer has at least one free slot. It is typical for a single cond lock to be operated on by threads of multiple types, where threads of one type specify the same predicate function but threads of different types specify different predicate functions.

Some restrictions must be placed on the code of predicate functions. In particular, every predicate must examine only data items which are protected by the relevant cond lock. This implies that the result (true/false) of a predicate function must change only because of actions performed while somebody holds the cond lock. In addition, because a cond lock controls access by multiple threads, every predicate function must compute the right answer no matter which thread executes it.

In the other direction, cond locks guarantee that they will invoke a predicate function only when it can safely/accurately compute its answer—in other words, a predicate function will be invoked only under the oversight of the lock.

Your task is to implement condition locks with the following interface:

- `int condlock_init(condlock_t *clp)` — initialize a condition lock.
- `void condlock_lock_when(condlock_t *clp, int (*predicate)(void *), void *arg)` — Acquire the condition lock once the predicate is true. Whenever `predicate` is called, `arg` will be passed as its argument.
- `void condlock_unlock(condlock_t *clp)` — Unlock the condition lock.
- `void condlock_destroy(condlock_t *clp)` — Destroy the condition lock. You will not need to implement this.

To provide the simplest possible example, given the interface above it is straightforward to implement a `condlock_lock_always()` function that unconditionally acquires the lock: simply pass a predicate that always returns true.

```
static int condlock_true_pred(void *arg) { return 1; }

void condlock_lock_always(condlock_t *lock) {
    condlock_lock_when(lock, condlock_true_pred, NULL);
}
```

A small example showing the use of condition locks to manage asynchronous message passing follows.

```
condlock_t queue_lock;
queue_t some_queue;

static int ready_to_receive(void *pqueue) {
    queue_t *queue = (queue_t *)pqueue;
    return !queue_is_empty(queue);
}
static int clear_to_send(void *pqueue) {
    queue_t *queue = (queue_t *)pqueue;
    return !queue_is_full(queue);
}

msg_t receive_message(void) {
    condlock_lock_when(&lock, ready_to_receive, &some_queue);
    // Since condlock_lock_when() returned, we know that
    // ready_to_receive must be true and thus queue_is_empty is false.
    assert(!queue_is_empty(&some_queue));
    msg_t msg = queue_dequeue(&some_queue);
    condlock_unlock(&lock); // may enable a send_message() thread
    return msg;
}

void send_message(msg_t msg) {
    condlock_lock_when(&lock, clear_to_send, &some_queue);
    queue_enqueue(&some_queue, msg);
    condlock_unlock(&lock); // may enable a receive_message() thread
}
```

The remainder of this page is intentionally blank.

Assumptions:

1. You may use regular Project 2 thread-library primitives: mutexes, condition variables, semaphores, readers/writer locks, etc.
2. You may assume that callers of your condition lock routines will obey the rules—for example, predicates will depend only on state protected by the lock. **But you must be careful that you obey the rules as well!**
3. 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`).
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.
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).

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 the next page. If we cannot understand the solution you provide, your grade will suffer!

(a) [5 points] Please declare your `condlock_t` here. If you need one (or more) auxilary structures, you may declare it/them here as well.

```
typedef struct {

} condlock_t;
```

(b) [15 points] Now please implement `int condlock_init()`, `void condlock_lock_when()`, and `void condlock_unlock()`,

. . . space for condition lock implementation . . .

. . . space for condition lock implementation . . .

5. [10 points] `Nuts and bolts`

   (a) [4 points] Write a sequence of up to five x86-32 instructions which, when run, will have the effect of placing the address of one of the instructions (you get to pick which one) into `%eax`. It is OK for you to destroy the values in other registers.

(continued on next page)

(b) [6 points] Consider the following C code:

```
int g;                /* 1 */

void foo() {
    int i=1;          /* 2 */
    static int s;     /* 3 */
}

void bar() {
    static int s=1;   /* 4 */
}

int main(int argc, char **argv) /* 5, 6 */
{
    return 0;
}
```

For each variable, indicate with a check mark where in memory it is stored at runtime. Assume `-O0`, where all variables are initially stored in memory rather than registers.

	text	data	bss	heap	stack
1. `g`					
2. `i`					
3. `s`					
4. `s`					
5. `argv`					
6. `argv[0]`					

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.