Computer Science 15-410: Operating Systems
Mid-Term Exam (A), Fall 2011

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.	**20**		
3.	**15**		
4.	**20**		
5.	**10**		
	75		

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

Andrew ID: _____

1. [10 points] Short answer.

   (a) [5 points] Define "rodata." Give examples of what it is used for. Briefly explain a bad thing that could happen if we didn't have "rodata."

   (b) [5 points] Define: "internal fragmentation." We are expecting an answer of three to five sentences.

2. [20 points] Deadlock?

Harry Q. Bovik is hosting a video game playing party. He has 2 wireless controllers, 2 wired controllers, and 2 gaming consoles. Each gaming console can have only 1 wired controller connected to it at once, but it can have up to 4 controllers total. All of Harry's games are two-player games. As people arrive at the party, they form groups of two. Each group of two then attempts to allocate one wired controller and one wireless controller using the code below. (Two wireless controllers would also work, but no one tries to allocate that because they don't want to hog the wireless controllers.) Harry doesn't know how many people will come to the party, so assume there could be an arbitrary number of threads each running the `acquire2_and_play()` function. For the purposes of the exam, assume that Harry will properly call all initialization functions before everyone arrives and all destroy functions after everyone leaves, and that no function calls will fail.

The remainder of this page is intentionally blank.

```
#define TOTAL_WIRED 2
#define TOTAL_WIRELESS 2

int wired_avail = TOTAL_WIRED;
int wireless_avail = TOTAL_WIRELESS;
mutex_t controller_mutex;
cond_t controller_cond;

#define ACQUIRE(avail, want) do {\
    while ((avail) && (want)) {\
        (avail)--;\
        (want)--;\
    }\
} while(0)

#define RELEASE(avail, release, cond) do {\
    while (release) {\
        (avail)++;\
        (release)--;\
        cond_signal(cond);\
    }\
} while(0)

void acquire_controllers(int wired_want, int wireless_want) {
    mutex_lock(&controller_mutex);
    while (wired_want || wireless_want) {
        ACQUIRE(wired_avail, wired_want);
        ACQUIRE(wireless_avail, wireless_want);
        if (wired_want || wireless_want)
            cond_wait(&controller_cond, &controller_mutex);
    }
    mutex_unlock(&controller_mutex);
}

void release_controllers(int wired_release, int wireless_release)
{
    mutex_lock(&controller_mutex);
    RELEASE(wired_avail, wired_release, &controller_cond);
    RELEASE(wireless_avail, wireless_release, &controller_cond);
    mutex_unlock(&controller_mutex);
}

// No need to lock around play_game(): once we allocate our
// controllers, a console must be available
void acquire2_and_play(void *unused) {
    acquire_controllers(1, 1);
    play_game();
    release_controllers(1, 1);
}
```

Suggestions for working on this problem:

1. Read *both* parts of the problem before working on *either* part. Also, if you get "stuck" on one part, maybe working on the other part for a while will help.
2. When tracing the execution of the code, we recommend a tabular format very similar to this:

Old Avail	TID	Request	Holds	New Avail	Status	Comment
(2,2)	0	(1,1)	(1,1)	(1,1)	gaming	
(1,1)	1	(1,1)	(1,1)	(0,0)	gaming	
(0,0)	2	(1,1)	(0,0)	(0,0)	blocked on (1,1)	
(0,0)	0	(-1,-1)	(0,0)	(1,1)	done	
(1,1)	2	(1,1)	(1,1)	(0,0)	gaming	

   This format should help you think about the execution as a series of requests while keeping track of what is acquired by each thread over time. Releasing a resource can be conveniently represented as acquiring a negative resource. The tuples represent some number of wired and wireless controllers, in that order (the same order as parameters to `acquire_controllers()`). The first line in the example represents that 2 wired controllers and 2 wireless controllers are available initially, then Thread 0 requests 1 wired controller and 1 wireless controller. The thread acquires everything requested, leaving 1 wired and 1 wireless controller available, and begins gaming. Because you should list every request that occurs, the "new avail" column of one row should always be equal to the "old avail" column of the next row. The comments column can contain whatever explanations you think would make your trace more clear. *Be sure that any execution trace or argument you provide us with is easy to read and conclusively demonstrates the claim you are making.*
3. 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!
4. **Finally, a hint: you should consider the can-deadlock and cannot-deadlock possibilities as being equally likely.**

The remainder of this page is intentionally blank.

(a) [10 points] Can the code shown above deadlock? If so, provide an execution sequence using the tabular form shown above. If deadlock is *not* possible, provide a clear and concise argument that it cannot happen; your reasoning should be convincing enough that it could be included with the code as documentation, and should probably mention one or more of the four "deadlock ingredients" by name. You should assume that the only code that allocates or frees the game controllers is the function `acquire2_and_play()`.

For his next party, Harry buys some 3-player video games, although he does not buy any new equipment. The existing code does not change, but now the system will run an arbitrary mixture of threads running the old `acquire2_and_play()` function and threads running a new `acquire3_and_play()` function (for reading convenience, both are shown below; you should assume that the only code that allocates or frees the game controllers is in those two functions). Again, notice that the players try to be polite, always allocating as few wireless controllers as possible.

```
// No need to lock around play_game(): once we allocate our
// controllers, a console must be available
void acquire2_and_play(void *unused) {
    acquire_controllers(1, 1);
    play_game();
    release_controllers(1, 1);
}
// As in acquire2_and_play, no need to synchronize around play_game();
void acquire3_and_play(void *unused) {
    acquire_controllers(1, 2);
    play_game();
    release_controllers(1, 2);
}
```

(b) [10 points] Can the code for this new party deadlock? If so, provide an execution sequence using the same format as before. If deadlock is *not* possible, provide a clear and concise argument that it cannot happen; your reasoning should be convincing enough that it could be included with the code as documentation, and should probably mention one or more of the four "deadlock ingredients" by name.

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

3. [15 points] Race conditions.

You are working on implementing a program which will generate, for testing purposes, requests to a remote file-system server. Your program will fetch files from the server, store them in a local cache, and perform some local computations on the file contents to make sure the server is providing you with correct answers. For now you are not concerned with path names and directories, because it's easier to consider the file system as just an "array of files," indexed by the file-number (inode number) that the server uses to store each file.

- At various times, threads will select some remote file (by picking a number at random), download the contents of that file from the server, and "register" the file in the local cache using `register_inode()`.
- At various times, threads will "check out" (obtain references to) files from the cache using `use_inode()`; once a file is referenced, it must not be removed from the cache until everybody using it has released it with `release_inode()`.
- The local cache must be garbage-collected from time to time: when it contains "too many" files, some files should be deleted. The files that should be deleted are those which are least-recently-used ("LRU").

When reading the code below, you should assume that all system calls succeed, and all global data structures are initialized elsewhere and before the code is run. Assume also that users of this code are well-behaved, e.g., nobody will "release" a cached file that they hadn't previously "checked out."

The following functions' implementations are not shown, but behave "as expected."

```
/** Downloads an inode from the server to local storage -- SLOW */
int fetch_inode(int inode);

/** Removes an inode from local storage -- SLOW */
void remove_inode(int inode);

/** Adds a new item to the LRU list (at the back) */
void lru_add(lru_t *lru, q_node *node);

/** Moves an LRU-list item to the back of the LRU list */
void lru_mark_used(lru_t *lru, q_node *node);

/** Removes an LRU-list item from the list */
void lru_remove(lru_t *lru, q_node *node);

/** Gets the front node of the LRU list */
q_node* lru_front(lru_t *lru);
```

The next page contains an implementation of the LRU cache; the first part of the question includes some sample code which uses the LRU cache.

```
/* Our filesystem has a max of 1000 nodes */
#define INODE_LIMIT 1000
#define GC_LIMIT 10
#define LRU_LIMIT (INODE_LIMIT - GC_LIMIT)

typedef struct q_node {
  int id;
  struct q_node *prev;
  struct q_node *next;
} q_node;

typedef struct {
  int refcount; // -1 for "not present"
  q_node node;
  mutex_t lock;
} inode_t;

inode_t inodes[INODE_LIMIT];
int lru_entries;
cond_t gc_cvar;
mutex_t lru_lock;
lru_t lru;

/* 1 = inode already here, 0 = added */
int register_inode(int inode_id) {
  inode_t *inode = &inodes[inode_id];
  mutex_lock(&inode->lock);
  /* Is this inode already registered? */
  if (inode->refcount != -1) {
    mutex_unlock(&inode->lock);
    return 1;
  }
  inode->refcount = 0;
  mutex_unlock(&inode->lock);

  /* Insert inode into the LRU */
  mutex_lock(&lru_lock);
  lru_entries++;
  lru_add(&lru, &inode->node);
  if (lru_entries > LRU_LIMIT) {
    cond_signal(&gc_cvar);
  }
  mutex_unlock(&lru_lock);
  return 0;
}

int use_inode(int inode_id) {
  return alter_inode(inode_id, 1);
}

int release_inode(int inode_id) {
  return alter_inode(inode_id, -1);
}
```

```
/* Private, only called internally */
static int alter_inode(int inode_id, int delta) {
  inode_t *inode = &inodes[inode_id];
  mutex_lock(&inode->lock);
  if (inode->refcount == -1) {
    mutex_unlock(&inode->lock);
    return -1;
  }
  inode->refcount += delta;
  mutex_unlock(&inode->lock);

  /* Update the LRU */
  mutex_lock(&lru_lock);
  lru_mark_used(&lru, &inode->node);
  if (lru_entries > LRU_LIMIT) {
    cond_signal(&gc_cvar);
  }
  mutex_unlock(&lru_lock);
  return 0;
}

void garbage_collect() {
  q_node *node, *next;
  int id;
  mutex_lock(&lru_lock);
  cond_wait(&gc_cvar, &lru_lock);
  for (node = lru_front(&lru);
       lru_entries > LRU_LIMIT && node != NULL;
       node = next) {
    next = node->next;
    inode_t *inode = &inodes[node->id];
    mutex_lock(&inode->lock);
    if (inode->refcount != 0) {
      mutex_unlock(&inode->lock);
      continue;
    }
    id = node->id;
    inode->refcount = -1;
    lru_remove(&lru, node);
    lru_entries--;
    mutex_unlock(&inode->lock);
    mutex_unlock(&lru_lock);
    remove_inode(id); /* This takes awhile */
    mutex_lock(&lru_lock);
  }
  mutex_unlock(&lru_lock);
}

void gc_thread() {
  while (1) {
    garbage_collect();
  }
}
```

(a) [4 points] Consider the following function, designed to "check out" a file from the cache, first fetching it from the server if necessary.

```
int open_inode_handle(int inode) {
  int found_it;

  /* Is the inode in the local cache? */
  if (use_inode(inode) >= 0)
    return 1;

  /* Oops, must download from the server */
  if (fetch_inode(inode) < 0)
    return -1;

  /* If registration "fails", don't worry:  some other thread
   * must have randomly chosen the same file we did, and we
   * both downloaded it.  That's silly, but it is unlikely to
   * happen, so we don't care:  all we care about is that it
   * got cached by one of us.
   */
  register_inode(inode);

  /* Should not fail, it's registered! */
  found_it = use_inode(inode);
  assert(found_it >= 0);
  return found_it;
}
```

What situation (perhaps an unlikely one) could cause the `assert()` to fail? Describe a small code change that would solve this problem.

(b) [8 points] Describe two race conditions that involve the functions `use_inode()`, `release_inode()`, `alter_inode()`, and `garbage_collect()`. (For partial credit, describe one; if you can think of more than two, try to describe the two "worst" ones.)

(c) [3 points] Explain how to improve the implementation of `use_inode()`, `release_inode()`, `alter_inode()`, and/or `garbage_collect()` to avoid the problems you identified. It should be possible for you to describe each of your proposed changes *convincingly* in one to three sentences.

You may use this page as extra space for your race-condition solution if you wish.

4. [20 points] "Big Reader Locks"

As hardware and software becomes increasingly parallelized, we expect more and more performance out of concurrent systems. One performance problem arises when two CPUs access shared memory in a way that forces them to reconcile their individual cached representations of that memory. For example, when code running on one CPU writes to a memory location that is in another CPU's cache, the two caches must communicate so that the second CPU either receives the new value or at least forgets the stale value it has. It's worse if code running on one CPU *reads* a memory location for which the most up-to-date value is stored in another CPU's cache and is "dirty," i.e., not yet written back to RAM: somehow the first CPU must get the second CPU to share the data, and it must stall its read operation until the new value becomes available. If many threads running on many CPU's are updating and fetching a single cache line, the memory-bus traffic which makes this work is expensive; programmers often refer to this as "cache-line bouncing" because which cache "owns" the cache line (i.e., has permission to store a new value into it) shifts rapidly from one CPU to another.

In Project 2 you implemented reader/writer locks, which are designed to enable a large number of threads to simultaneously read some or all of a data structure. Many simultaneous readers is generally considered reasonable, since any number of CPUs can simultaneously cache pieces of a data structure without lots of bus traffic *as long as nobody is making changes*. However, the reader/writer locks you implemented for Project 2 probably are not "good citizens" when it comes to multiprocessor cache behavior: if many threads running on many CPUs try to lock one of your rwlocks at the same time, we suspect that some part of your `struct rwlock` will be forced to bounce around to all of the caches (perhaps all lockers start off by acquiring a mutex inside the rwlock?). Likewise, if many readers concurrently *unlock* an rwlock, some parts of the lock will probably bounce around quite a bit. This is ironic since rwlocks are supposed to enable threads to work *without* cache-line bouncing.

In this question we will ask you to design a special kind of rwlock, with a slightly different calling interface, which minimizes cache-line bouncing in the case where: (1) many threads frequently lock and unlock the rwlock in read mode and (2) write-mode lockers are rare. This lock will be called a "big reader lock" ("brlock"). In order to achieve blazingly fast performance for the reader threads, you are authorized to (1) use more memory than a regular rwlock typically uses, and (2) make locking a brlock in write mode *much* more expensive than locking it in read mode.

The basic idea is that we won't worry about cache-line bouncing when a brlock is initialized, destroyed, or locked in write mode: scanning and/or mutating the entire brlock, including rudely yanking it out of everybody else's caches, is fine during these operations. Also, if *multiple* threads try to write-lock a brlock at the same time, it is fine if they bounce everything around for a long time (multiple write-locking threads is not a usage pattern we care about). But *the key part of your mission* is this: any time a thread locks or unlocks a brlock in read mode, it should not *write* to any memory which any other reader thread will *read or write*, nor should it *read* any memory which any other reader thread will *write*. The brlock API helps make this feasible by requiring each thread locking or unlocking a brlock to indicate which part of the brlock "belongs" to it, as follows. When a brlock is created, the creator specifies how many threads will use it and arranges for each of those threads to be issued a "seat number" ranging from 0 to $T - 1$. Each time a thread locks or unlocks the brlock, it passes in its "seat number."

Here is the interface.

- `void brlock_init(struct brlock *b, int num_seats)` - Initializes a big-reader lock to provide the given number of seats.

  The `num_seats` argument to `brlock_init()` specifies the maximum number of threads that will be using the brlock. Each relevant thread will keep track of a "seat number" (from `0` to `num_seats-1`), which indicates (in an abstract sense) which subset of the big-reader lock's internal memory the reader thread is free to access and modify without causing inter-processor memory conflicts.
- `void brlock_lock(struct brlock *b, int my_seat, int mode)` - Acquires the big-reader lock in the given mode (`BRLOCK_READ` or `BRLOCK_WRITE`) from the given seat.
- `void brlock_unlock(struct brlock *b, int my_seat)` - Releases the big-reader lock from the given seat (according to the mode in which that thread acquired it).
- `void brlock_destroy(struct brlock *b)` - Destroys the brlock. (You don't need to implement this function on the exam.)

Your mission is to avoid (or at least lessen) cache conflicts among readers, as described in the "mission statement" above. For the purposes of this exam question you may assume that cache lines are 4 bytes (one word),[1] so concurrent writes to adjacent words will not cause bouncing.

While implementing brlocks, you may use mutexes, condition variables, and semaphores, but you may *not* use rwlocks, or any atomic operations, such as `xchg`, `xadd`, or `cmpxchg`. Condition variables and semaphores guarantee FIFO ordering, but mutexes may not. For exam purposes you may assume that all library functions and other associated functions will not fail.

The remainder of this page is intentionally blank.

[1] This is false. A more realistic size would be 64 bytes, but it would be painful for you to worry about cache-aligning your data structures on the exam.

A pseudo-code usage example follows.

```
struct brlock b;
#define NUM_READERS 8192

void *reader_thread(void *arg)
{
    int my_seat = (int)arg;

    while (1) {
        void *data;

        brlock_lock(&b, my_seat, BRLOCK_READ);
        data = read_global_table(); // fetch or compute something from the table
        brlock_unlock(&b, my_seat);
        process_data(data); // think for a while about what we fetched/computed
    }
}

int main(int argc, char *argv[])
{
    int i, my_seat;

    brlock_init(&b, NUM_READERS+1);

    for (i = 0; i < NUM_READERS; i++) {
        thr_create(reader_thread, (void *)i);
    }

    my_seat = i;

    while (1) {
        char input[BUF_SIZE];

        readline(sizeof (input), input); // Blocks for user input--a long time!

        // This part is *very* expensive!
        // Luckily, it happens rarely and only one thread does it.
        brlock_lock(&b, my_seat, BRLOCK_WRITE);
        rewrite_global_table(input);
        brlock_unlock(&b, my_seat);
    }
}
```

(a) [5 points] Please declare your `struct brlock` here. Also implement `brlock_init()`. If you wish, you may also declare one or two small/auxiliary structs.

Hint: A `struct brlock` which contains some sort of single internal mutex which protects all readers' state will *necessarily* cause cache-line bouncing. Make sure you understand why, and why this is undesirable, before designing your brlock internals. Of course, you will probably also wish to consider whether your implementation starves readers, starves writers, or deadlocks.

```
struct brlock {









};

void brlock_init(struct brlock *b, int num_seats)
{









}
```

(b) [15 points] Now please write your code for `brlock_lock()` and `brlock_unlock()`.

You may use this page as extra space for your brlock solution if you wish.

You may use this page as extra space for your brlock solution if you wish.

5. [10 points] `Nuts & Bolts`.

In the Project 1 environment, where all code runs in kernel mode, when an interrupt, exception, or other "surprise" happens, the CPU pushes a "trap frame" onto the existing stack (there is no "stack switch" as happens during a user-to-kernel transition in Project 2 or Project 3).

List the three elements of the Project 1 (kernel-only) trap frame. Briefly say why each of those elements is saved onto the stack by the CPU (for example, you should probably be able to suggest something that would go wrong if that element were *not* saved and later restored).

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 ls(int size, char *buf);

/* "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.