Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Spring 2018

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.	10		
3.	20		
4.	20		
5.	10		
	70		

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. $\boxed{\text{10 points}}$ Short answer.

Give a definition of each of the following terms *as it applies to this course*. We are expecting three to five sentences or "bullet points" for each definition. Your goal is to make it clear to your grader that you understand the concept and can apply it when necessary.

(a) $\boxed{\text{5 points}}$ Recursion

(b) $\boxed{\text{5 points}}$ “Paradise Lost”

2. $\boxed{\text{10 points}}$ Pair matching.

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). As you know, these can be combined to produce higher-level objects such as semaphores or readers/writers locks.

One such object is a "pair matcher." The idea is that some jobs must be worked on by pairs of threads, and the threads need some (dynamic) way to pick a partner to work with. After a pair matcher is initialized, an even number of threads will invoke the `match` operation. The `match` operation involves some amount of thread synchronization, potentially including blocking, and then returns to each thread, in a reasonably timely fashion, the thread identification number of the thread it has been matched with. A match object does not know how many threads will invoke it, though it can depend on the number being even. Once a program is sure that no more threads will invoke the `match` operation on a particular object, the `destroy` operation can and should be invoked.

A small example program using a pair matcher is displayed on the next page. You should assume that the example program is correct. The threads in this program don't do anything useful, since it is just a test. Each time a thread is matched, if it and its partner don't have equally even/odd thread i.d.'s, then it (and its partner) will exit. Even if a thread is repeatedly matched with "the right kind of thread" it will exit after 10 matches; thus, all threads will eventually exit and the program will complete.

On the page after the example program is code for a *broken* pair-matcher implementation. Your job will be to diagnose a bug. When this implementation is used, occasionally a nonsensical result is observed—maybe two threads are partnered with the same other thread, maybe some thread is not issued as a partner to somebody... you will be telling us!

The remainder of this page is intentionally blank.

```
#define NTHREADS 410
int tids[NTHREADS];
pmatch_t matcher;
void *threadbody(void *ignored);

int main(int argc, char** argv)
{
    thr_init(4096); // exam: no failures

    pmatch_init(&matcher); // exam: no failures

    for (int t = 0; t < NTHREADS; t++) {
        tids[t] = thr_create(threadbody, (void *) t); // exam: no failures
    }
    for (int t = 0; t < NTHREADS; t++) {
        thr_join(tids[t], NULL);
    }
    printf("Done\n");
    pmatch_destroy(&matcher);
    thr_exit(0);
}

void *threadbody(void *ignored)
{
    int me = thr_getid();
    int partner;
    int done = 0, rounds = 0;

    while (!done) {
        int coolpartner;

        partner = pmatch_match(&matcher);

        printf("I am %d, my partner is %d\n", me, partner);

        coolpartner = (me&1) == (partner&1);

        if (coolpartner) {
            printf("Whee!  That was so much fun I might do it again.\n");
        }
        if ((++rounds == 10) || !coolpartner) {
            done = 1;
        }
    }
    return 0;
}
```

Below is the broken pair-matcher implementation. Note that because this is "exam-mode code" you should assume that all correct invocations of thread-library primitives always succeed, and that all invocations of the pair-matcher functions will be legal.

```
typedef struct pmatch {
    mutex_t m;
    int t1;
    int t2;
    cond_t xfer;
} pmatch_t;

int pmatch_init(pmatch_t *pmp) {
    mutex_init(&pmp->m);   // exam: no failure
    pmp->t1 = pmp->t2 = -1;
    cond_init(&pmp->xfer); // exam: no failure
    return (0);
}

void pmatch_destroy(pmatch_t *pmp) {
    mutex_destroy(&pmp->m);
    pmp->t1 = pmp->t2 = -2;
    cond_destroy(&pmp->xfer);
}

int pmatch_match(pmatch_t *pmp) {
  int ret;

  mutex_lock(&pmp->m);
  if (pmp->t1 == -1) {
    // we are first
    pmp->t1 = thr_getid();
    cond_wait(&pmp->xfer, &pmp->m);
    ret = pmp->t2;
    pmp->t1 = -1;
    pmp->t2 = -1;
    mutex_unlock(&pmp->m);
  } else {
    // we are second
    pmp->t2 = thr_getid();
    ret = pmp->t1;
    cond_signal(&pmp->xfer);
    mutex_unlock(&pmp->m);
  }
  return (ret);
}
```

First, briefly describe in words something that is wrong with this code; then present a trace which supports your claim.

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 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 the pair-matcher question.

3. 20 points Reducing Deadlock.

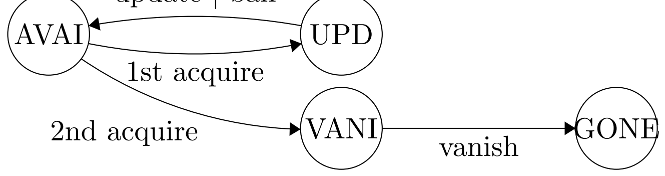
For this problem, we will be considering a parallel program loosely inspired by the "reduce" part of the "map-reduce" paradigm. The program works on a pool of data objects (integers in the example, but in a real system they would be more complicated). Each worker thread repeatedly tries to lock two data objects. If it can, it performs a computational "operation" on the two objects, producing a single value as the answer (the example uses addition, a simple and fast associative operator, but in a real system the operations would be more complicated and would take longer). The worker thread will update one of the objects so its value reflects the result of the operation, and will mark the other object as no longer usable for the rest of the computation. If a worker thread succeeds in locking one object but can't get a lock on a second object, but the computation isn't over yet, it will "bail out" of the attempted computation by releasing the object it has and trying again. Meanwhile, if a thread detects that an object is *about* to be updated, it "reserves" the object, waits for the update to finish, and then proceeds as described above. Eventually all of the objects except for object 0 should be "used up;" object 0 should contain the final result of the computation; and the threads should exit. Unfortunately, this code can deadlock!

Summary:

1. Each thread attempts to acquire two operands, combines them using some operation, and then stores the result in one and invalidates the other.
2. If the thread acquires an operand object "for updating," it will store the result of the operation back into that specific operand. The object's state will go from AVAILABLE to UPDATING and then back to AVAILABLE.
3. If the thread acquires an operand "for vanishing," it means that it will *not* store the result of the operation back into that operand. The object's state will go from AVAILABLE to VANISHING to GONE.
4. If the thread can't acquire a second operand, the state of the first one will go from AVAILABLE to UPDATING and then back to AVAILABLE.
5. In principle, the operation step could be much more expensive than addition. Because of this, a semaphore is used while a value is being updated. Operands which are UPDATING can be "reserved" by another during the computation step.

You may note that `done()` is needlessly inefficient, and that near the end of the computation many of the threads will be wasting effort. However, these poor design decisions allow for the code to be more compact for the purposes of this exam.

Here is a state-transition diagram.



The code will begin with the `main()` function, which is straightforward: it initializes the data, spins up worker threads, joins all of them, and then exits. Then we present some helper functions, followed by the key parts of the code: `operator()`, `acquire()`, and `release()`.

```
#define POOL_SIZE 32
#define NUM_THREADS 10
#define STACK_SIZE 4096

void *operator(void *arg);

typedef enum {
    AVAILABLE, VANISHING, UPDATING, GONE
} state_t;

struct operand {
    mutex_t guard;
    state_t state;
    bool reserved;
    sem_t ready;
    int val;
} pool[POOL_SIZE];

int main(int argc, char** argv) {
    thr_init(STACK_SIZE); // exam: no failures

    for (int i = 0; i < POOL_SIZE; i++) {
        mutex_init(&pool[i].guard); // exam: no failures
        sem_init(&pool[i].ready, 1); // exam: no failures
        pool[i].val = i;
        pool[i].state = AVAILABLE;
        pool[i].reserved = false;
    }

    int tids[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++)
        tids[i] = thr_create(operator, NULL); // exam: no failures
    for (int i = 0; i < NUM_THREADS; i++)
        thr_join(tids[i], NULL);

    for (int i = 0; i < POOL_SIZE; i++) {
        mutex_destroy(&pool[i].guard);
        sem_destroy(&pool[i].ready);
    }
    int actual = pool[0].val;
    int expected = POOL_SIZE * (POOL_SIZE - 1) / 2;
    return actual == expected ? 0 : -1;
}
```

```
bool done(void) {
    for (int i = 1; i < POOL_SIZE; i++) {
        mutex_lock(&pool[i].guard);
        if (pool[i].state != GONE) {
            mutex_unlock(&pool[i].guard);
            return false;
        }
        mutex_unlock(&pool[i].guard);
    }
    return true;
}

int do_operation(int a, int b) {
    int result = a + b;
    printf("(%d, %d) -> %d\n", a, b, result); // exam: no failures
    return result;
}

int acquire(state_t desired, int avoid);
void release(int slot, state_t next_state);

void *operator(void *arg) {
    while (!done()) {
        int i = acquire(UPDATING, -1);
        if (i == -1) {
            continue;
        }
        int j = acquire(VANISHING, i);
        if (j == -1) {
            release(i, AVAILABLE);
            continue;
        }

        int val1 = pool[i].val;
        int val2 = pool[j].val;
        release(j, GONE);
        pool[i].val = do_operation(val1, val2);
        release(i, AVAILABLE);
    }
    return NULL;
}
```

```
int acquire(state_t desired, int avoid) {
    for (int i = 0; i < POOL_SIZE; i++) {
        // pool[0] should only be updated, never consumed
        if (desired == VANISHING && i == 0) {
            continue;
        }
        // skip the slot the caller already holds
        if (i == avoid) {
            continue;
        }

        mutex_lock(&pool[i].guard);

        if (!pool[i].reserved) {
            if (pool[i].state == UPDATING) {
                pool[i].reserved = true;
                mutex_unlock(&pool[i].guard);

                sem_wait(&pool[i].ready);

                mutex_lock(&pool[i].guard);
                pool[i].state = desired;
                pool[i].reserved = false;
                mutex_unlock(&pool[i].guard);
                return i;
            } else if (pool[i].state == AVAILABLE) {
                sem_wait(&pool[i].ready);
                pool[i].state = desired;
                mutex_unlock(&pool[i].guard);
                return i;
            }
        }

        mutex_unlock(&pool[i].guard);
    }
    return -1;
}

void release(int slot, state_t next_state) {
    mutex_lock(&pool[slot].guard);
    pool[slot].state = next_state;
    mutex_unlock(&pool[slot].guard);

    sem_signal(&pool[slot].ready);
}
```

(a) $\boxed{\text{4 points}}$ Describe in detail how each of the deadlock ingredients is present. You may find it particularly useful to list *all instances* of each ingredient carefully. This problem is complicated, so properly organizing your thoughts will help you achieve a solution.

(b) $\boxed{\text{14 points}}$ Show *clear, convincing* evidence of deadlock. Begin by *describing the problem* in one or two sentences; then *clearly specify a scenario.* You should provide an execution trace resulting in a deadlock. The deadlock trace may require more threads than you are used to! *Not all of these threads must participate in the actual deadlock.* Use this hint carefully; first try to understand the mechanism by which the deadlock occurs, then fill in what each thread must do. 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 as extra space for the deadlock trace.

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

(c) $\boxed{\text{2 points}}$ Explain in detail (though code is *not* required!) how the program could be modified to not deadlock. 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. $\boxed{\text{20 points}}$ Abortable condition variables.

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). As you know, these can be combined to produce higher-level objects such as semaphores or readers/writers locks.

In this question you will implement a synchronization object called an "abortable condition variable" (abbreviated ACV). It is like a regular condition variable, with two key differences. First, a thread can decide that a failure or emergency state exists and can invoke an operation which causes all threads waiting on an ACV to stop waiting. Second, each time a thread waits on an ACV, the return value from the `wait()` operation indicates whether the wait ended because the condition became true, i.e., because a thread invoked `signal()`, or whether the wait ended due to the declaration of an abort situation.

As an example, consider the following trace which which demonstrates the relationship between `acv_abort()` and `acv_signal()`.

Time	Thread 0	Thread 1	Thread 2
0	i = acv_wait(a)		
1	...wait...	j = acv_wait(a)	
2		...wait...	
3			acv_signal(a)
4			acv_abort(a)
5		j == -1	
6	i == 0		

A small example program using an abortable condition variable is displayed on the next page.

The remainder of this page is intentionally blank.

```
#define NTHREADS 10
#define NROUNDS 100

acv_t   acv;
mutex_t mutex;
bool    terminate = false;
int     counter   = 0;

void* work(void* index_arg);
void* control(void* ignored);

int main(int argc, char** argv) {
    int tids[NTHREADS];

    thr_init(4096);  // exam: no failure
    acv_init(&acv);      // exam: no failure
    mutex_init(&mutex);  // exam: no failure

    tids[0] = thr_create(control, NULL);  // exam: no failure

    for (int t = 1; t < NTHREADS; t++)
        tids[t] = thr_create(work, (void*)t);  // exam: no failure
    for (int t = 0; t < NTHREADS; t++)
        thr_join(tids[t], NULL);

    mutex_destroy(&mutex); // don't need to destroy acv
    thr_exit(0);
}

void* control(void* ignored) {
    char c;
    while ((c = getchar()) != 'q') {
        int wakeupCount;
        if (isdigit(c))
            wakeupCount = c - '0';
        for (int i = 0; i < wakeupCount; i++) // Let some people do work based on user input
            acv_signal(&acv);
    }
    printf("Aborting Computation\n");
    mutex_lock(&mutex);
    terminate = true;
    acv_abort(&acv);
    acv_destroy(&acv);
    mutex_unlock(&mutex);
    return NULL;
}
```

```
void* work(void* index_arg) {
    int index = (int)index_arg;

    int result = 0;
    for (int r = 0; r < NROUNDS && result == 0; r++) {
        // Do work
        sleep(genrand() % 100);
        mutex_lock(&mutex);

        if (terminate) {
            printf("Terminating without abort: %d\n", index);
            mutex_unlock(&mutex);
            break;
        } else {
            int myCount = counter;
            counter++;

            printf(
                "Thread: %d done with round %d count: %d\n", index, r, myCount);
            result = acv_wait(&acv, &mutex);
        }

        mutex_unlock(&mutex);
    }
    if (result != 0)
        printf("Thread: %d aborted!\n", index);

    return NULL;
}
```

Your task is to implement an abortable condition variable with the following interface. Note that you will not need to implement a `broadcast()` operation.

- `int acv_init(acv_t *a)`
  The abortable condition variable shall be initialized. It is illegal for an application to use the abortable condition variable before it has been initialized or to initialize an abortable condition variable when it is already initialized and in use. `acv_init` shall returns 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 (thus, this declaration shows the function returning a value so that the declaration matches what a non-exam implementation would declare).
- `void acv_destroy(acv_t *a)`
  The abortable condition variable shall be destroyed. It is illegal for a program to invoke `acv_destroy()` if any threads are operating on it. A common pattern is to call `acv_abort` before calling `acv_destroy`
- `int acv_wait(acv_t *a, mutex_t *mp)`
  The abortable condition variable shall wait until signaled (`acv_signal`) or aborted (`acv_abort`). The mutex `mp` should be released when waiting and reacquired upon returning. The mutex should be reacquired even if the wait was aborted. `acv_wait` shall return 0 if successfully signaled (`acv_signal`) or a negative value if aborted (`acv_abort`).
- `void acv_signal(acv_t *a)`
  The abortable condition variable shall be signaled waking up a single waiting thread if one exists.
- `void acv_abort(acv_t *a)`
  The abortable condition variable shall be aborted. All threads waiting on the abortable condition variable should be woken. `acv_abort` should not return until the abortable condition variable is no longer in use by any of the waiting threads. It is illegal for other threads to call `acv_wait`, `acv_signal`, or another `acv_abort` after the condition variable has been aborted.

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 routines will obey the rules. **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). You may also assume that condition variables are strictly FIFO if you wish.

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) $\boxed{\text{5 points}}$ Please declare your `acv_t` here. If you need one (or more) auxilary structures, you may declare it/them here as well.

```
typedef struct {







} acv_t;
```

(b) $\boxed{\text{15 points}}$ Now please implement `acv_init()`, `acv_wait()`, `acv_signal()`, `acv_abort()`, and `acv_destroy()`.

. . . space for abortable condition variable implementation . . .

. . . space for abortable condition variable implementation . . .

. . . space for abortable condition variable implementation . . .

5. $\boxed{\text{10 points}}$ `Nuts & Bolts`.

Below are excerpts from the Pebbles specifications of the `fork()` and `exec()` system calls, lightly edited for brevity.

- `int fork(void)` - Creates a new task. The new task receives an exact, coherent copy of all memory regions of the invoking task. The new task contains a single thread which is a copy of the thread invoking `fork()` except for the return value of the system call. If `fork()` succeeds, the invoking thread will receive the ID of the new task's thread and the newly created thread will receive the value zero. Errors are reported via a negative return value, in which case no new task has been created.
- `int exec(char *execname, char **argvec)` - Replaces the program currently running in the invoking task with the program stored in the file named `execname`. The argument `argvec` points to a null-terminated vector of null-terminated string arguments.

  The number of strings in the vector and the vector itself will be transported into the memory of the new program where they will serve as the first and second arguments of the the new program's `main()`, respectively. Before the new program begins, `%EIP` will be set to the "entry point" (the first instruction of the `main()` wrapper, as advertised by the ELF linker). The stack pointer, `%ESP`, will be initialized appropriately so that the `main()` wrapper receives four parameters:

  1. `int argc` - count of strings in `argv`
  2. `char *argv[]` - argument-string vector
  3. `void *stack_high` - highest legal (byte) address of the initial stack
  4. `void *stack_low` - lowest legal (byte) address of the initial stack

You will note that the text for `fork()` specifies (in a very brief fashion) the values *all* general-purpose registers in the new program, but the text for `exec()` specifies values for only some of the registers. In this question we will explore this curious difference.

The remainder of this page is intentionally blank.

(a) $\boxed{\text{6 points}}$ Explain why it is ok for the description of `exec()` to ignore lots of general-purpose register values. Your explanation should probably include some specific examples.

(b) $\boxed{\text{4 points}}$ Because the description of `exec()` is silent about the values of some registers, kernel implementors have the opportunity to make a natural mistake with potential security implications. Explain.

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 */
```

Typing Rules Cheat-Sheet

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

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

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

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

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

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

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

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

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

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

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

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

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

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.