Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Fall 2021

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

Andrew Username	
Full Name	

Question	Max	Points	Grader
1.	**10**		
2.	**10**		
3.	**15**		
4.	**20**		
	55		

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.

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, for 48 hours. It is permissible to discuss the contents of this exam before I leave this exam session, or after the 48-hour period has elapsed.

I certify that my exam submission is my own work, in compliance with the rules stated above.

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] Design process.

When designing a body of code, at times one finds oneself thinking, "I wonder if I should use Approach A or Approach B?" According to the 15-410 design orthodoxy, you should follow a specific process to resolve your question.

As you may know, at present CMU, which traditionally had 15-week Fall and Spring semesters (plus a 12-week Summer), is experimenting with the notion of permanently switching to 14-week semesters (while retaining a 12-week Summer). There are many reasons why this change is being considered and experimented with.

One reason is that, due to oddities of the ways that key holidays are placed, when we have 15-week semesters we sometimes have a Fall that ends very late, followed by a Spring that begins early. When this occurs, students have a short Fall/Spring break, with limited travel time and, for some, elevated travel costs. Faculty members and staff members experience a short break as well, making it difficult to handle administrative tasks which must be done between the semesters. Switching from 15-week semesters to 14-week semesters addresses the years in which time is tight between Fall and Spring.

On the other hand, a 14-week semester is 6% shorter than a 15-week semester. Concerns have been raised about what might happen if instructors try to maintain the original content of classes despite less time being available, and/or what might happen if instructors remove 6% of the content from each class. An additional concern is the potential (unknown) effects of students graduating from CMU with an overall/cumulative reduction of 6% in the amount of time spent on "practice," e.g., programming assignments—whether or not the assignments have been trimmed, it is possible that student learning is reduced merely by less time having been spent on "practice."

> CMU faces an "Approach A or Approach B?" question. Please demonstrate how to apply the recommended 15-410 design practice to this non-15-410 question. For the purposes of this question we will be scoring based on your *description* of the decision, not whether we agree or disagree with what you chose. Begin with a brief description of the problem (one to three sentences) and then show us your design decision—answers that correctly use the 15-410 approved data structure will receive higher scores. If you wish, you may consider factors/concerns beyond the ones raised above by your instructors.

The remainder of this page is intentionally blank.

Use this page for the design-orthodoxy question.

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

2. [10 points] Barrier problem.

After a long night working on P2, you finally decide to get some sleep. Unfortunately, even in your dreams, you cannot escape from synchronization code. In tonight's nightmare, you were forced to implement barriers.

Barriers are a synchronization object that are initialized with some number, N, and allow an arbitrary number of different threads to `wait()` on them. Threads invoking `wait()` are blocked until N threads have done so. At that point, those N threads are unblocked; further calls to `wait()` will cause those threads to block until another set of N threads has accumulated. For any barrier nerds, the above discription is of a "reusable barrier."

In the way of dreams, after you finished implementing barriers, time passed in an instant and suddenly you found yourself looking at a HubBucket issue raised against your barriers, titled "Barriers don't work!?!?!?!?." The frustrated author of the issue kindly included a test that fails when using your barrier implementation. As you finish reading the example, you wake up.

Your dedication to writing high-quality software means you won't be able to go back to sleep until you've learned what bug is in the barriers you dreamed up. Below are listings for the test code submitted in the HubBucket issue and then the barrier implementation itself. Since no one ever files false bug reports, you know that the issue is in the barrier implementation.

The remainder of this page is intentionally blank.

```
// Termination lemma: the 3rd group of 32 will be released by 1/2 of the first 64
#define NUM_FENCES 64
#define JUMPS_PER_FENCE 2
#define NUM_SHEEP (NUM_FENCES + (NUM_FENCES / 2))
#define STACK_SIZE (1 << 11)

// Atomically adds src to dst, and returns the old dst.
extern int atomic_add(int *dst, src);

static int fence_jumps[NUM_FENCES];

int main(void)
{
    thr_init(STACK_SIZE); // Exam mode: Can't fail.

    barrier_t brr;
    barrier_init(&brr, NUM_FENCES); // Exam mode: Can't fail.

    // Dream up some sheep.
    int sheep[NUM_SHEEP];
    for (unsigned i = 0; i < NUM_SHEEP; i++) {
        // Exam mode: Can't fail.
        sheep[i] = thr_create(&count_sheep, (void*) &brr);
    }

    // Count some sheep :)
    for (unsigned i = 0; i < NUM_SHEEP; i++) {
        thr_join(sheep[i], NULL); // Exam mode: can't fail.
    }
    for (unsigned j = 0; j < NUM_FENCES; j++) {
        affirm_msg(fence_jumps[j] == JUMPS_PER_FENCE,
                   "Uh-oh! We seem to have miscounted our sheep!");
    }

    barrier_destroy(&brr);
    thr_exit(NULL);
}

void *count_sheep(void *brr)
{
    static int jump_counter = 0;

    int jump_id;
    do {
        barrier_wait(brr);
        jump_id = atomic_add(&jump_counter, 1);
        fence_jumps[jump_id % NUM_FENCES]++;
    } while ((jump_id < NUM_FENCES) && (jump_id & 0x1));

    return NULL;
}
```

```
#include <barrier.h>

typedef struct {
    mutex_t counter_lock;
    cond_t barrier_wait;
    unsigned wait_max;
    unsigned wait_remaining;
} barrier_t;

/* Initializes the given barrier with the given count. */
int barrier_init(barrier_t *brr, unsigned count)
{
    affirm(brr);
    affirm(count);

    mutex_init(&brr->counter_lock); // Exam mode: can't fail.
    cond_init(&brr->barrier_wait);  // Exam mode: can't fail.
    brr->wait_max = count;
    brr->wait_remaining = count;

    return 0;
}

/* Destroys the given barrier. */
void barrier_destroy(barrier_t *brr)
{
    affirm(brr);
    affirm(brr->wait_max == brr->wait_remaining);

    cond_destroy(&brr->barrier_wait);
    mutex_destroy(&brr->counter_lock);
}

/* Blocks until brr->wait_max threads are in barrier_wait */
void barrier_wait(barrier_t *brr)
{
    affirm(brr);

    mutex_lock(&brr->counter_lock);
    if (--brr->wait_remaining > 0) {
        cond_wait(&brr->barrier_wait, &brr->counter_lock);
        mutex_unlock(&brr->counter_lock);
    } else {
        brr->wait_remaining = brr->wait_max;
        mutex_unlock(&brr->counter_lock);
        cond_broadcast(&brr->barrier_wait);
    }
}
```

(a) [7 points] Finding the problem.

The above barrier implementation can misbehave in a way that causes the affirmation in the test code to fail. Assume that the problem is in the barrier implementation, e.g., we are not looking for claims that `mutex_lock()` might be implemented incorrectly. You should assume that invocations of thread-library primitives (e.g., `mutex_lock()`) succeed rather than detecting inconsistency or otherwise failing. If you wish, you may assume cvars are strictly FIFO. You may wish to refer to the "cheat sheets" at the end of the exam.

If you have correctly identified a sychronization problem, you will be able to *briefly and clearly* summarize it and show a *clear and compelling* execution trace using the tabular execution format from the lectures and homework assignment. Confusing descriptions and unclear execution traces will be read as evidence of incomplete understanding and will be graded as such. This means that it is to your advantage to *think your answers through before beginning to write.* It is wise to write a "rough-draft" trace on the scrap paper provided at the end of the exam, or on the back of some other page, before you write anything here. If we cannot understand the solution you provide on this page, your grade will suffer!

First briefly state what can go wrong in one to three sentences. Then present a trace which demonstrates your claim. Use the format presented in class as shown below. You may use more or fewer columns or lines in your trace. You may introduce temporary variables or other obvious notation as necessary to improve the clarity of your answer. If you wish, you may abbreviate:

- `mutex_lock(&brr->counter_lock)` as `L`,
- `mutex_unlock(&brr->counter_lock)` as `U`,
- `cond_wait(&brr->barrier_wait,&brr->counter_lock)` as `W`, and
- `cond_broadcast(&brr->barrier_wait)` as `B`.

The permissibility of those abbreviations does not imply that your trace should consist of *only* `L`, `U`, `W`, and `B`.

> *Be sure that the execution trace you provide us with is easy to read and conclusively demonstrates the claim you are making.*

Execution Trace

Time	T0	T1	T2
0	L		
1		L	
2			

Use this page for the barrier question.

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

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

(b) [3 points] Fixing the problem.

Briefly explain one way to resolve the problem you identified above. You may present a small amount of code if you wish, though any clear explanation will suffice. If your resolution has any notable drawbacks, mention them.

3. [15 points] Grading deadlock.

When Project 2 thread libraries are submitted, a long and arduous grader-assignment process begins. This is a non-trivial problem because each TA should not grade some P2's (for example, a TA shouldn't grade his or her roommate's P2). Each TA has a function which maps from group numbers to a boolean can-grade/cannot-grade decision. The instructor suggests an initial assignment of groups to TA's, but the TA's need to swap groups back and forth until everybody's grading assignment is acceptable. In real life, each TA grades multiple P2's, but because it will simplify the code we will counterfactually assume that each TA grades exactly one P2. We will also assume that group numbers and TA numbers both start from zero (and thus both end at `NUM_TAS-1`).

Each TA's state is a current (tentative) assignment and a flag indicating whether the TA is happy with that assignment. Every time a new tentative assignment is made to a TA, the TA's `happy` flag is set to false because only the TA can confirm that the assignment is acceptable. When a TA receives an unacceptable tenative assignment, he or she selects another TA at random and swaps assignments with that TA (both are marked unhappy; each one will independently evaluate the new tentative assignment). The assignment procedure completes when all TA's are happy at the same time.

Meanwhile, the instructor is monitoring the progress of the TA's. The instructor goes from TA to TA checking whether each one is happy. If all the TAs are happy *at the same time*, then the instructor terminates the assignment procedure and "returns" to doing other things. Any time the instructor finds an unhappy TA, the instructor takes a break (to give the TA's time to shuffle groups around) and later begins another "everybody is happy" verification round, resuming where the procedure left off last time.

```
struct ta {
    mutex_t lock;
    int group;
    int happy;
};

volatile struct ta tas[NUM_TAS];
volatile int done = 0;

/** @brief Returns true if a specific TA can grade a specific group */
int can_grade(int i, int group);

/** @brief Selects a random number from 0..(NUM_TAS-1) which is not equal to i
  * @param i The identity of the TA
  */
int get_other(int i);
```

The remainder of this page is intentionally blank.

```
void* ta (void* self_param) {
    int self = (int)(self_param);

    while (!done) {
        mutex_lock(&tas[self].lock);
        tas[self].happy = can_grade(self, tas[self].group);

        if (!tas[self].happy) {
            // I'm unhappy, find someone to swap with
            int other = get_other(self);

            // acquire both locks in the right order
            if (other < self)
                mutex_unlock(&tas[self].lock);
            mutex_lock(&tas[other].lock);
            if (other < self)
                mutex_lock(&tas[self].lock);

            // swap groups, marking the other person as unhappy
            tas[other].happy = 0;
            int tmp_group = tas[other].group;
            tas[other].group = tas[self].group;
            tas[self].group = tmp_group;

            mutex_unlock(&tas[other].lock);
        }

        mutex_unlock(&tas[self].lock);
        sleep(100); // attend a class, eat, maybe actually sleep
    }
    return 0;
}
```

The remainder of this page is intentionally blank.

```
int main() {
    int i;

    thr_init(4096); // exam, so this can't fail

    for (i = 0; i < NUM_TAS; i++) {
        mutex_init(&tas[head].lock); // exam, so this can't fail
        tas[i].group = i; // tentatively assign group i to TA i
        tas[i].happy = 0;
    }
    for (i = 0; i < NUM_TAS; i++)
        thr_create(ta, (void*)i); // exam, so this can't fail

    // instructor runs on the main thread, checking until all TA's are happy
    int head = 0;
    int tail, gotsome;

    while (!done) {
        tail = head;
        gotsome = 0;

        do {
            mutex_lock(&tas[head].lock);
            if (!tas[head].happy)
                break;
            gotsome = 1;
            head = (head + 1) % NUM_TAS;
        } while (head != tail);

        if (gotsome && (head == tail)) {
            done = 1;
            tail = (head + 1) % NUM_TAS;
        }

        while (tail != head) {
            mutex_unlock(&tas[tail].lock);
            tail = (tail + 1) % NUM_TAS;
        }
        mutex_unlock(&tas[head].lock);

        if (!done)
            sleep(100); // teach a class, eat, maybe actually sleep
    }
    thr_exit(0);
    return (0);
}
```

Suggestions for working on this problem:

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

TA 0	TA 1
lock(0)	
...	
unlock(0)	
	lock(0)

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

(a) [12 points] Unfortunately, this P2 assignment procedure can deadlock. Show a *clear, convincing* execution trace that yields a deadlock (missing, unclear, or unconvincing traces will result in only partial credit).

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

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

(b) [3 points] Briefly describe how to fix or restructure the code so that it does not deadlock. It is not necessary for your answer to include code for it to receive full credit if it is clear and convincing (be sure to indicate how your solution addresses one or more deadlock ingredient(s)).

4. [20 points] Message buffers.

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

In this question, you will implement a synchronization object called a "message buffer." A message buffer has some fixed capacity chosen at initialization. The capacity may be large or small depending on the requirements of the application. Each message has a corresponding "topic": a non-negative integer which can be used to identify groups of messages in the system.

Any number of "senders" may place messages into the buffer, but they are forced to wait if the buffer has run out of space. Any number of "receivers" may request an available message with a given topic anywhere in the buffer, returning promptly whether or not such a message exists. The message buffer need not guarantee that messages are distributed in the same order in which they are received. Below is documentation for the message-buffer functions.

```
// The message buffer shall be initialized with the given capacity for
// uncollected messages. It is illegal for an application to use the message
// buffer before it has been initialized or to initialize a message buffer when
// it is already initialized and in use.  mb_init() shall return 0 on success or
// a negative error code on failure.  Because this is an exam, you may assume
// that allocating and initializing the necessary state will succeed.
int mb_init(mb_t *mb, int size);

// The message buffer shall be destroyed.  It is illegal for a program to invoke
// mb_destroy() if any threads are operating on it.
void mb_destroy(mb_t *mb);

// The message buffer shall take receipt of the given message with its
// corresponding topic.  If the message buffer is full (i.e., it already
// holds `size` uncollected messages), then this function blocks until
// space is available in message buffer.  Once the message buffer takes
// receipt of the message, mb_send() shall return promptly.
void mb_send(mb_t *mb, int topic, void *msg);

// The message buffer shall make available a message with the given topic,
// if there is such a message in the buffer.  If there is no such message,
// mb_recv() shall return -1 promptly.  If such a message is found, it shall
// be removed from the buffer and stored at `msg_dest`, and 0 shall be
// returned promptly.
int mb_recv(mb_t *mb, int topic, void **msg_dest);
```

On the next page is sample code that uses the message-buffer functions.

```
static mutex_t rand_lock;
static unsigned int rand_uint(void) {
    mutex_lock(&rand_lock);
    unsigned int res = genrand();
    mutex_unlock(&rand_lock);
    return res;
}

#define NUM_ROUNDS 200
#define NUM_SENDERS 4
#define NUM_PINGERS 3
#define NUM_PONGERS 2

#define TOPIC_PING 17
#define TOPIC_PONG 38

static volatile bool all_done = false;

static void *sender(void *arg) {
    mb_t *mb = arg;
    int tid = gettid();

    for (int send = 0; send < NUM_ROUNDS / NUM_SENDERS; send++) {
        void *msg = (void *)(rand_uint());
        int topic = rand_uint() % 2 ? TOPIC_PING : TOPIC_PONG;
        printf("sender-%05d: will send on topic %d msg %p\n", tid, topic, msg);
        mb_send(mb, topic, msg);
    }

    return NULL;
}

static void *recver(mb_t *mb, int topic, const char *tag) {
    int tid = gettid();

    while (true) {
        void *msg;
        if (mb_recv(mb, topic, &msg) == 0) {
            printf("%ser-%05d: recved msg %p\n", tag, tid, msg);
        }
        else if (all_done)
            break;
    }

    return NULL;
}
```

```
static void *pinger(void *arg) {
    return recver(arg, TOPIC_PING, "PING");
}

static void *ponger(void *arg) {
    return recver(arg, TOPIC_PONG, "PONG");
}

int main() {
    thr_init(4096); // assume this succeeds
    sgenrand(get_ticks());
    mutex_init(&rand_lock); // assume this succeeds

    unsigned size = rand_uint() % 5 + 1;
    printf("using %d mailboxes\n", size);
    mb_t mb;
    mb_init(&mb, (int)size); // assume this succeeds

    int sender_tids[NUM_SENDERS];
    int pinger_tids[NUM_PONGERS];
    int ponger_tids[NUM_PINGERS];
    for (int i = 0; i < NUM_SENDERS; i++)
        sender_tids[i] = thr_create(sender, &mb); // assume this succeeds
    for (int i = 0; i < NUM_PINGERS; i++)
        pinger_tids[i] = thr_create(pinger, &mb); // assume this succeeds
    for (int i = 0; i < NUM_PONGERS; i++)
        ponger_tids[i] = thr_create(ponger, &mb); // assume this succeeds

    for (int i = 0; i < NUM_SENDERS; i++)
        thr_join(sender_tids[i], NULL);
    all_done = true; // done sending
    for (int i = 0; i < NUM_PINGERS; i++)
        thr_join(pinger_tids[i], NULL);
    for (int i = 0; i < NUM_PONGERS; i++)
        thr_join(ponger_tids[i], NULL);

    mb_destroy(&mb);
    mutex_destroy(&rand_lock);
    thr_exit(0);
}
```

Assumptions:

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

The remainder of this page is intentionally blank.

Please declare a `struct mb` and implement:

- `int mb_init(mb_t *mb, int size)`,
- `void mb_destroy(mb_t *mb)`,
- `void mb_send(mb_t *mb, int topic, void *msg)`, and
- `int mb_recv(mb_t *mb, int topic, void **msg_dest)`

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

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

```
typedef struct mb {









} mb_t;
```

```
typedef struct aux {









} aux_t;
```

...space for message-buffer implementation...

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

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

System-Call Cheat-Sheet

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

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

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

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

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

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

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

Thread-Library Cheat-Sheet

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

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

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

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

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

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

Ureg Cheat-Sheet

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

#ifndef ASSEMBLER

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

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

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

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

#endif /* ASSEMBLER */
```

Useful-Equation Cheat-Sheet

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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