# Computer Science 15-410/15-605: Operating Systems
## Mid-Term Exam (A), Spring 2023

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. **PLEASE DO NOT WRITE FAINTLY WITH PENCIL. Please write in ink, or, if writing in pencil, please ensure that zero strokes in zero words are faint. Using a mechanical pencil with thin lead is probably unwise.**
4. This is a closed-book in-class exam. You may not use any reference materials during the exam.
5. **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 ...?"**
6. 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.
7. 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** | | |
| **5.** | **10** | | |
| | **65** | | |

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) [3 points] According to the 15-410 orthodoxy, there are three kinds of error. Briefly present them: for each, provide a name, describe in a general high-level sense what should be done in response to that kind of error, and explain why that reponse is what should be done about that kind of error. We are expecting approximately two sentences for each kind.

(b) [2 points] Please present, based on your P2 thread library, an example of *two* of the three kinds of error. For each, briefly describe what the code was trying to do, how the failure qualifies as that particular kind of error, and the action that your P2 implementation should have taken. We are not expecting you to provide code (we are interested in your description/analysis).

(c) [5 points] Describe the "Paradise Lost" phenomenon as used in this class. We are expecting three to five sentences or "bullet points." Your goal is to make it clear to your grader that you understand the concept and can apply it when necessary.

2. [10 points] Consider the following critical-section protocol:

```
#define NTHREAD 32 // a global static limit on the number of threads!
int tid(); // returns thread id from 0..(NTHREAD-1)
int lock_available = 1;
int waiting[NTHREAD] = { 0, };
int atomic_exchange(int *ip, int val); // behaves as expected

void lock() {
    int i = tid(), got_it = 0;

    waiting[i] = 1;
    while (waiting[i] && !got_it) {
      got_it = atomic_exchange(&lock_available, 0);
    }
    waiting[i] = 0;
}

void unlock() {
    int i;

    for (i = 0; i < NTHREAD; ++i) {
        if (waiting[i]) {
            waiting[i] = 0;
            return;
        }
    }
    atomic_exchange(&lock_available, 1);
}
```

(a) [7 points] There is a problem with this protocol. That is, it does not ensure that all three critical-section algorithm requirements are always met. Identify a requirement which is not met and lay out a scenario which demonstrates your claim. Use the format presented in class, i.e.,

| P0 | P1 |
|---|---|
| w[0] = 1; | |
| | w[1] = 1; |

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 possible to answer this question with a brief, clear trace, so you should do what is necessary to ensure that you do.

*Please don't accidentally skip the second part of this problem.*

You may use this page for the critical-section protocol question.

You may use this page as extra space for the critical-section protocol question if you wish.

(b) [3 points] Please *clearly* demonstrate (probably using code, though *very clear* explanatory text might work also) how to fix the problem you described above.

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] Simulation clock.

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 "simulation clock," loosely inspired by some problematic pseudo-code from one of our thread-synchronization lectures. The synchronization objects we have studied so far have the property that some operations stall or block threads (`mutex_lock()`, `cond_wait()`, `sem_wait()`), while other operations liberate threads (`mutex_unlock()`, `cond_signal()`, `sem_signal()`). But other synchronization objects, e.g., barriers, have the property that a single operation may stall threads, liberate threads, or both. You will be implementing such a synchronization object.

The basic idea is that there are multiple simulation threads that want to carry out activities at specific times, and also a single time-keeping thread that declares when new simulated times have arrived. The lecture pseudo-code we discussed suffered from the problem that a simulation thread might be awakened by a specific time but not be able to run before later times occurred. This happened because the code blocked simulation threads but let the clock thread move ahead without knowing that the simulation threads had time to do their work. So this synchronization object blocks *both* worker threads and the clock thread in a taking-turns fashion.

For exam purposes we have simplified the simulation-clock API as follows. First, this simulation-clock object mostly does not support absolute time. Worker threads always ask to be blocked for *relative* time, i.e., "block me for three time steps." Likewise, the clock thread does not declare that absolute times, i.e., "April 18, 1906," or "timestamp 357," have occurred, but instead merely indicates "seven ticks have passed." And to save "typing" the simulation-clock object does not have a function for querying the absolute time, though most likely your implementation will track that. These simplifications do not fundamentally change the nature of good solutions, but they do reduce the amount of code you will need to produce.

The two key operations are `sc_wait()` and `sc_tick()`. The job of `sc_wait()` is to block the invoking thread until an indicated number of ticks have passed from "the current time." One job of `sc_tick()` is to awaken however many threads (maybe zero blocked threads, maybe all of them) wanted to be awakened before or at the time that has arrived. Because `sc_tick()` can move the simulation time forward by any number of ticks, it may awaken some threads later than they had asked for. The *other* job of `sc_tick()` is to block the invoking clock thread until all of the awakened simulation worker threads have had a chance to run—this is the tricky part! Each of the awakened worker threads does some work and then *must call back* to the synchronization-clock object; when all have done so, the `sc_tick()` operation returns to the invoking clock thread. For reasons we will not discuss, "call back" is not a separate operation.

Because each simulation worker thread will be invoking `sc_wait()` over and over, in the common case a simulation worker will acknowledge one awakening by invoking `sc_wait()` the next time. There are two special cases: each thread's *first* invocation of `sc_wait()` does not acknowledge a previous awakening, and each thread's *final* invocation of `sc_wait()` does acknowledges its final awakening but does not block. To simplify your work, it is the responsibility of each

worker thread to specify with each invocation of `sc_wait()` whether it is an initial invocation, a continuing invocation, or a final invocation.

Also note that because of the API design this simulation-clock object can operate by tracking merely the *number* of acknowledgements, without needing to track which threads are issuing acknowledgements. Finally, as a convenience, the code using a simulation-clock object pre-declares the maximum number of simulation worker threads that will blcok on it at the same time. This is not typical of synchronization objects in general (e.g., `cond_init()` does not do this) but it may make your job simpler.

It is expected that the number of threads that a given simulation-clock object is tracking on will typically be "fairly small," and it is ok if a simulation clock experiences "reasonable slowdown" if the number of simultaneously blocked threads is larger than expected.

The remainder of this page is intentionally blank.

Below are declarations for the simulation-clock functions. The next page has sample code that uses them.

```
typedef enum {
    SC_FIRST,    // no ack, just wait
    SC_CONT,     // ack, then wait
    SC_LAST      // ack only, then return
} sc_mode_t;

// sc_init() initializes a simulation-clock object in the provided memory,
// indicating the maximum number of worker threads that can simultaneously
// block on it.  It is expected that this number is typically under 1,000.
//
// sc_init() returns a negative error code less than zero, or zero for success.

int sc_init(sc_t *sp, int nthreads);

// sc_destroy() deactivates a simulation-clock object.  The invoking thread
// must ensure that no threads are using the object.

void sc_destroy(sc_t *sp);

// sc_wait() acknowledges a previous wakeup and/or blocks until the indicated
// future simulation time has arrived (or passed).  Each thread must specify
// SC_FIRST the first time it calls sc_wait() on a particular sc_t, must
// specify SC_CONT to acknowledge a previous wakeup and block again, and
// SC_LAST to acknowledge a wakeup without blocking.  In the SC_LAST case
// the delta parameter is ignored.  In other cases the delta value must
// be non-negative.
//
// sc_wait() returns the simulation timestamp at which it was awakened,
// which may be later than the time that the thread had requested, or
// a negative error code less than zero.

int sc_wait(sc_t *sp, sc_mode_t mode, int delta);

// sc_tick() applies the provided positive delta to the simulation time
// and then awakens all threads who requested to be awakened at or before
// the new simulation time.
//
// sc_tick() blocks until that number of threads acknowledge via sc_wait(),
// and then returns the number of threads that were awakened.
//
// sc_tick() may return a negative error code less than zero.

int sc_tick(sc_t *sp, int delta);
```

```
#define NTHREAD 42
#define EVENTS 10
#define MAX_DELTA 13
static sc_t clock;
static mutex_t nrunm;
static int nrunning = 0;

void *threadbody(void *vid)
{
  int id = (int) vid;
  int delta = (id * 10) % MAX_DELTA, mode = SC_FIRST;

  mutex_lock(&nrunm); ++nrunning; mutex_unlock(&nrunm);

  for (int n = 0; n < (EVENTS-1); ++n) {
    printf("Thread %d sleeping %d ticks\n", id, delta);
    int result = sc_wait(&clock, mode, delta);
    printf("Thread %d awake again at %d\n", id, result);
    sleep(2); // fake "simulation work"
    mode = SC_CONT;
  }
  sc_wait(&clock, SC_LAST, -99);
  mutex_lock(&nrunm); --nrunning; mutex_unlock(&nrunm);
  return (0);
}

int main(void)
{
   int tid[NTHREAD], done = 0, delta = 7;

   thr_init(8192);
   sc_init(&clock, NTHREAD); // exam: cannot fail
   mutex_init(&nrunm); // exam: cannot fail

   affirm (sc_tick(&clock, 1) == 0); // "Just checking!"

   for (int n = 0; n < NTHREAD; ++n)
     tid[n] = thr_create(threadbody, (void *)n); // exam: cannot fail

   do {
     int awoke = sc_tick(&clock, delta);
     printf("Woke %d threads\n", awoke);
     sleep(1); // game time runs slower than real time
     mutex_lock(&nrunm);
     if (nrunning == 0)
       done = 1;
     mutex_unlock(&nrunm);
   } while (!done);

   for (int n = 0; n < NTHREAD; ++n)
     thr_join(tid[n], NULL);
   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 (etc.) are strictly FIFO if you wish.

The remainder of this page is intentionally blank.

*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!*

(a) [5 points] Please declare a `struct sc` and implement:

- int sc_init(sc_t *sp, int nthreads) and
- void sc_destroy(sc_t *sp).

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

```
typedef struct sc {









} sc_t;


typedef struct aux {









} aux_t;
```

...space for int sc_init(sc_t *sp, int nthreads) void sc_destroy(sc_t *sp)...

(b) [15 points] Now please implement:

- int sc_wait(sc_t *sp, sc_mode_t mode, int delta), and
- int sc_tick(sc_t *sp, int delta).

...space for simulation clock implementation...

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

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

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

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

Your friend, Grave O'Danger, is excited about a new idea for running C functions. He believes that performance will be improved by running certain functions inside their own independent stack regions (which we could call "stacklets")—for some reason he believes that "the cache will be less polluted" if some functions run "far away from" other functions. Instead of running `f(x)` the usual way, he wants to invoke `far_call(f,x)`; the `far_call()` function will allocate some memory for a new stacklet, and "somehow" invoke `f()` so that it runs on the new stacklet. Once `f()` returns, regular execution will resume on whatever stack was used before `far_call()` was invoked.

Grave has begun hacking together a crude prototype of `far_call()` as follows.

```
typedef (*intfun)(int);

extern wrapper(char *stack_high, intfun f, int arg); // TBD

#define FAR_SIZE 4096

int far_call(intfun f, int arg) {
    int result;
    unsigned char *stack_low, *stack_high;

    stack_low = malloc(FAR_SIZE); // Guaranteed to be 8-byte aligned

    if (stack_low) {
        stack_high = stack_low + FAR_SIZE - 8;
        result = wrapper(stack_high, f, arg);
        free(stack_low);
    } else {
        result = f(arg); // oh well
    }
    return (result);
}
```

Grave understands that the `wrapper()` function will need to be written in assembly code, but so far all of his attempts have resulted in program crashes. Because you are a helpful stack expert, your mission will be to write the wrapper for him. You should assume the standard "Linux x86-32" stack convention that we have been using.

Please write code for the `wrapper()` function. In addition to the code, you may provide documentation, such as a *clear* supporting diagram, which we will try to use to better understand your code. However, your score will be derived from your code as we understand it (i.e., diagrams without code will not score very highly).

You may use this page for your `wrapper()` 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 );

#define RWLOCK_READ  0
#define RWLOCK_WRITE 1
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.