Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (B), Spring 2020

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.	**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.

I have not received advance information on the content of this 15-410/605 mid-term exam by discussing it with anybody who took part in the main exam session or via any other avenue.

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.

Andrew ID: _____________

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

(a) $\boxed{\text{6 points}}$ When designing a body of code, at times one finds oneself thinking, "I wonder if I can assume X?" According to the 15-410 design orthodoxy, immediately upon having such a thought one is required to ask oneself two questions. Please state those questions. You may wish to include specific examples and/or to briefly explain why these two replacement questions are important.

You may use this page as extra space for the “assume” question if you wish.

(b) $\boxed{\text{4 points}}$ What is meant by “M:N threading”? What might be gained, and what might be lost, by employing such an approach?

You may use this page as extra space for the M:N question if you wish.

2. $\boxed{\text{10 points}}$ In 15-410 we are used to thinking of "the critical-section problem" as being the same thing as "what mutexes solve," but members of other cultures believe that the critical-section problem is best solved by semaphores. It is possible to deploy a general semaphore as a "binary semaphore," meaning a semaphore which always has either zero or one items available. And it is possible to protect a critical section with a binary semaphore instead of a mutex. If one does protect a critical section with a binary semaphore, then it is legitimate to ask whether a particular semaphore implementation does or does not fully solve the critical-section problem.

   Below is some semaphore code. When reading it, please assume that the underlying mutexes do fully solve the critical-section problem, and also that the underlying condition variables behave in the most-appropriate fashion.

The remainder of this page is intentionally blank.

```
typedef struct sem {
    mutex_t m;
    cond_t more;
    volatile int avail;
} sem_t;

int sem_init(sem_t *s, int count)
{
    (void) mutex_init(&s->m);    // exam mode: can't fail
    (void) cond_init(&s->more);  // exam mode: can't fail
    s->avail = count;
    return (0);
}
void sem_destroy(sem_t *s)
{
    cond_destroy(&s->more);      // exam mode: can't fail
    mutex_destroy(&s->m);        // exam mode: can't fail
}

void sem_wait(sem_t *s)
{
    int result;

    // fast path
    mutex_lock(&s->m);
    result = s->avail;
    --s->avail;
    mutex_unlock(&s->m);
    if (result > 0)
        return;

    // slow path
    mutex_lock(&s->m);
    do {
        cond_wait(&s->more, &s->m);
    } while (s->avail < 0);
    mutex_unlock(&s->m);
    return;
}
void sem_signal(sem_t *s)
{
    mutex_lock(&s->m);
    ++s->avail;
    cond_signal(&s->more);
    mutex_unlock(&s->m);
}
```

Unfortunately, this semaphore code does *not* fully solve the critical-section problem. That is, if we were to replace mutex calls in a piece of code with calls to this semaphore code, the resulting behavior would fail to solve one or more requirements of the critical-section problem. Identify a requirement which is not met and 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(&s->m)` as `L`, `mutex_unlock(&s->m)` as `U`, `cond_wait(&s->more,&s->m)` as `W`, and `cond_signal(&s->more)` as `S`.

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.

Execution Trace

Time	T0	T1	T2	T3
0	L			
1		L		
2				

Use this page for the critical-section protocol question.

Andrew ID: ______________

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

3. $\boxed{\text{15 points}}$ "Dead rock"

This semester, 410 students have formed four rock bands. Each band has a (small) set of songs they have practiced; each song requires an eclectic set of instruments.

Naturally the students are familiar with Professor Roger Dannenberg's prominence in the field of computer music and figure that instead of buying instruments they can borrow them from him. Professor Dannenberg's collection contains these instruments.

Accordion	Bagpipe	Cowbell	Drum
4	3	2	5

The bands, their songs, and the instruments needed to play each song are listed in the table below. Note that each instrument, each band name, and each song title can be represented unambiguously by a 1-letter or 1-digit abbreviation.

	Accordion	Bagpipe	Cowbell	Drum
Band 1: One Thread Yielding				
Entry of the Threads	1	0	1	1
Free Memory	0	0	1	3
Band 2: Two Cores				
Groove Gettid	2	0	0	2
Hit The Break Jack	1	1	0	1
Band 3: Three Spare Bits				
Justify My Yield	0	1	1	1
Kill Dash Nine	3	0	2	2
Lock Free Your Heart	1	3	0	2
Band 4: Forth				
Mutex Romance	1	1	0	1
Never Going to Halt	2	2	0	4

There was a "battle of the bands" (popularity competition) last weekend, among these four bands and no others.

Professor Dannenberg had planned to lend instruments to the band members, but he was out of town and his return flight was delayed by bad weather. So the 410 students broke into his lab and each band borrowed the instruments necessary to play its first song (E, G, J, and M, respectively). Each band is stubborn, and will demand to go on stage and be allowed to play all of its songs in some order, acquiring instruments as necessary, before being willing to yield the stage or return any instruments to the professor's collection.

Professor Dannenberg, not only a musician but also a noted expert on deadlock, is worried that the naive resource allocations made by the students, plus their stubborn policy, may have placed the entire music festival at risk of deadlock.

(a) $\boxed{\text{4 points}}$ Given the state of affairs described above, i.e., each band holds the instruments needed to play their first song, is there a safe sequence? If so, list a sequence of bands in order. If not, explain why there is not.

At Professor Dannenberg's urging, the 410 students decide to return all of the instruments to him and adopt an improved instrument-management protocol. After finishing each song and deciding on the next, they calculate how many instruments of each type they have which they will not need for the next song, and how many they will need which they do not have. For example, Band 1 switching from song E to song F would need to release one accordion and acquire two drums.

After calculating the change in resource requirements, they proceed as follows:

1. Release unneeded Accordions
2. Release unneeded Bagpipes
3. Release unneeded Cowbells
4. Release unneeded Drums
5. Acquire additional Accordions
6. Acquire additional Bagpipes
7. Acquire additional Cowbells
8. Acquire additional Drums

Please note that each of the above steps is atomic, meaning "Acquire 2 drums" happens all at once (at a time when two or more drums are available).

Initially each band is playing no song, and may decide to "play no song" (i.e., take a break) at any time. As far as the management protocol is concerned, playing no song is equivalent to playing a song requiring no instruments.

(b) $\boxed{\text{8 points}}$ Consider Bands 2, 3, and 4, and their songs as listed in the table above. Do not consider Band 1 (Band 1 is playing no song). Assume there are three stages, so the three bands can play simultaneously. Can this algorithm deadlock?

If so, list a sequence of songs starting from a state in which no band is playing any song and leading to deadlock in the tabular format indicated below and draw a process/resource graph showing the deadlock.

Band	Song	Operation	Instrument	Count
1	F	Acquire	Cowbell	1

If not, list the four deadlock requirements and briefly indicate for each one whether the proposed management protocol has, or lacks, that ingredient.

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

(c) $\boxed{\text{3 points}}$ Now consider this variant on the protocol of Part B: each group, after playing a song, takes a break (thus releasing all instruments) and then acquires instruments for their next song as described in the Part B protocol. Given all four bands, can this algorithm deadlock? Justify your answer.

4. $\boxed{\text{20 points}}$ Conditions.

Suddenly, you wake up. There's a message in your inbox urgently demanding to know where "the new test code" is. "What new test code???" you wonder. Then you remember... after a challenging but rewarding semester in 15-410, you were asked to serve as a teaching assistant. When you made the mistake of criticizing the comprehensiveness of the thread-library test suite, Professor Eckhardt responded by asking you to write some condition-variable test code.

In particular, you are asked to write a program which tests whether the following (multi-part) property is true of a condition-variable implementation: if two threads wait on the same condition variable, they will both stop running, and if the condition variable is then signaled twice, the threads will awaken in FIFO order (the same order in which they stopped running). Of course, it is extremely difficult for a simple test program to *completely* verify that condition variables *always* block and unblock threads in a *strictly* FIFO fashion, but the goal of this test code is to catch with high probability implementations which are consistently broken.

Your program should, with very high probability, print the string "PASS" if the condition variable implementation you are testing has the property described above and "FAIL" if it does not (if your program prints multiple "PASS" or "FAIL" strings, that is acceptable as long as all are "PASS" or all are "FAIL", i.e., no mixtures!).

Professor Eckhardt has told you that the 15-410 test suite already contains extensive tests for mutexes, so while writing your condition-variable test you should assume that the underlying mutexes behave perfectly. However, the student condition-variable code you are testing might synchronize threads incorrectly in multiple ways: perhaps threads are not reliably blocked; perhaps threads are not reliably unblocked; perhaps threads are unblocked in an unreasonable order; etc. Your test program should be written so that, with high probability, it produces an answer in less than sixty seconds, and, with high probability, the "PASS" or "FAIL" indication is accurate. What your test program does after printing its verdict doesn't matter: your test program is allowed to hang, crash, etc., because once the test harness has seen your answer it will shut Simics down.

Note that, as an interface-compliant test program, *you cannot inspect or modify the internals of any thread-library data objects*–because your test code must run against every student thread library, you are strictly a *client* of the abstractions provided by each library. However, because this is test code, you may use normally-distasteful code constructs such as yield loops or even spin-waiting if you must; that said, solutions with smaller quantities of distasteful code are likely to receive higher scores.

Because you think that some day a future version of your test might test multiple condition variables in parallel, you decide to encapsulate the housekeeping data structures (e.g., condition variables, mutexes, ints, chars, doubles, ...) used by your test in a `struct test`; in the interests of keeping the test code simple, you decide to limit the struct to containing six fields.

Hint: it is a very good idea to write a draft version of your solution on scrap paper before writing anything in the "official answer" space.

(a) $\boxed{\text{5 points}}$ Please declare your `struct test` here. Also write a function `void test_init(test_p tp)` to initialize a test.

```
typedef struct test {













} *test_p;

void test_init(test_p tp)
{













}
```

(b) $\boxed{\text{15 points}}$ Now please write your test code. The suggested structure is one to three small helper functions and a `main()` function.

You may use this page as extra space for your “condition variable test” solution if you wish.

You may use this page as extra space for your “condition variable test” solution if you wish.

5. $\boxed{\text{10 points}}$ Nuts and Bolts: SprintFast.

One day you overhear members of the course staff excitedly discussing who has achieved the highest score while playing an odd-sounding pseudo-athletic computer game. It sounds as if, when the game is run, a virtual player collects some number of mysterious performance-enhancing pills called "stairoids," and then runs a fixed distance at some rate of speed, with higher speeds obviously being more desirable.

A member of the course staff accidentally leaves behind a printout of the source code for the game. When you review the code, you are perplexed, because it is not immediately obvious how any player could achieve a speed of more than 1 km/s. But you definitely heard some TAs reporting high numbers and "maximum buff-ness." Eventually you come to suspect that the members of the course staff are reporting scores based on cheating in the race by exploiting the stairoid-collecting/race-running program.

Below you will find the source code for the game program and, in case it is useful, a disassembly of what seems like a key function. Following those materials we will ask you questions about how the cheating is taking place.

(Hint: Despite the problem being about a race, there are no race conditions here.)

The remainder of this page is intentionally blank.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

typedef struct {
    int stairoid_counter;
    char name[128];
} player_t;

#define EMIT(s) write(STDOUT_FILENO, s, strlen(s))

void collect_stairoids(int *stairoids) {
    player_t player;
    player.stairoid_counter = 0;

    EMIT("Enter ur name:\n");
    if (scanf("%127s", player.name) < 0) {
        EMIT("Cheeter! No stairoids 4 u!!\n");
        *stairoids = 0;
        return;
    }

    EMIT("Collecting...\n");
    player.stairoid_counter += 1;

    char msg[64];
    sprintf(msg, "Congrats %s! You got %d units of stairoid!\n",
                 player.name, player.stairoid_counter);
    EMIT(msg);

    *stairoids = player.stairoid_counter;
}

int sprint(int stairoids) {
    if (stairoids == 0) {
        EMIT("Natural talent!\n");
    } else if (stairoids < 10) {
        EMIT("You're buff!\n");
    } else {
        EMIT("Buff overflow\n");
    }
    return stairoids;
}
```

```
int main(void) {
    int stairoids;
    collect_stairoids(&stairoids);
    int speed = sprint(stairoids);

    printf("Your speed was %d km/s!\n", speed);

    return 0;
}
```

```
08048c54 <collect_stairoids>:
 8048c54:    55                      push   %ebp
 8048c55:    89 e5                   mov    %esp,%ebp
 8048c57:    81 ec e8 00 00 00       sub    $0xe8,%esp
 8048c5d:    c7 85 74 ff ff ff 00    movl   $0x0,-0x8c(%ebp)
 8048c64:    00 00 00
 8048c67:    c7 04 24 2c 8a 0c 08    movl   $0x80c8a2c,(%esp)
 8048c6e:    e8 4d ae 00 00          call   8053ac0 <strlen>
 8048c73:    89 44 24 08             mov    %eax,0x8(%esp)
 8048c77:    c7 44 24 04 2c 8a 0c    movl   $0x80c8a2c,0x4(%esp)
 8048c7e:    08
 8048c7f:    c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048c86:    e8 55 21 01 00          call   805ade0 <__libc_write>
 8048c8b:    8d 85 74 ff ff ff       lea    -0x8c(%ebp),%eax
 8048c91:    83 c0 04                add    $0x4,%eax
 8048c94:    89 44 24 04             mov    %eax,0x4(%esp)
 8048c98:    c7 04 24 3c 8a 0c 08    movl   $0x80c8a3c,(%esp)
 8048c9f:    e8 ac 0b 00 00          call   8049850 <__isoc99_scanf>
 8048ca4:    85 c0                   test   %eax,%eax
 8048ca6:    79 32                   jns    8048cda <collect_stairoids+0x86>
 8048ca8:    c7 04 24 42 8a 0c 08    movl   $0x80c8a42,(%esp)
 8048caf:    e8 0c ae 00 00          call   8053ac0 <strlen>
 8048cb4:    89 44 24 08             mov    %eax,0x8(%esp)
 8048cb8:    c7 44 24 04 42 8a 0c    movl   $0x80c8a42,0x4(%esp)
 8048cbf:    08
 8048cc0:    c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048cc7:    e8 14 21 01 00          call   805ade0 <__libc_write>
 8048ccc:    8b 45 08                mov    0x8(%ebp),%eax
 8048ccf:    c7 00 00 00 00 00       movl   $0x0,(%eax)
 8048cd5:    e9 93 00 00 00          jmp    8048d6d <collect_stairoids+0x119>
 8048cda:    c7 04 24 5f 8a 0c 08    movl   $0x80c8a5f,(%esp)
 8048ce1:    e8 da ad 00 00          call   8053ac0 <strlen>
 8048ce6:    89 44 24 08             mov    %eax,0x8(%esp)
 8048cea:    c7 44 24 04 5f 8a 0c    movl   $0x80c8a5f,0x4(%esp)
 8048cf1:    08
 8048cf2:    c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048cf9:    e8 e2 20 01 00          call   805ade0 <__libc_write>
 8048cfe:    8b 85 74 ff ff ff       mov    -0x8c(%ebp),%eax
 8048d04:    83 c0 01                add    $0x1,%eax
 8048d07:    89 85 74 ff ff ff       mov    %eax,-0x8c(%ebp)
 8048d0d:    8b 85 74 ff ff ff       mov    -0x8c(%ebp),%eax
 8048d13:    89 44 24 0c             mov    %eax,0xc(%esp)
 8048d17:    8d 85 74 ff ff ff       lea    -0x8c(%ebp),%eax
 8048d1d:    83 c0 04                add    $0x4,%eax
 8048d20:    89 44 24 08             mov    %eax,0x8(%esp)
 8048d24:    c7 44 24 04 70 8a 0c    movl   $0x80c8a70,0x4(%esp)
```

```
8048d2b:    08
8048d2c:    8d 85 34 ff ff ff       lea    -0xcc(%ebp),%eax
8048d32:    89 04 24                mov    %eax,(%esp)
8048d35:    e8 e6 0a 00 00          call   8049820 <_IO_sprintf>
8048d3a:    8d 85 34 ff ff ff       lea    -0xcc(%ebp),%eax
8048d40:    89 04 24                mov    %eax,(%esp)
8048d43:    e8 78 ad 00 00          call   8053ac0 <strlen>
8048d48:    89 44 24 08             mov    %eax,0x8(%esp)
8048d4c:    8d 85 34 ff ff ff       lea    -0xcc(%ebp),%eax
8048d52:    89 44 24 04             mov    %eax,0x4(%esp)
8048d56:    c7 04 24 01 00 00 00    movl   $0x1,(%esp)
8048d5d:    e8 7e 20 01 00          call   805ade0 <__libc_write>
8048d62:    8b 95 74 ff ff ff       mov    -0x8c(%ebp),%edx
8048d68:    8b 45 08                mov    0x8(%ebp),%eax
8048d6b:    89 10                   mov    %edx,(%eax)
8048d6d:    c9                      leave
8048d6e:    c3                      ret
```

(a) $\boxed{\text{6 points}}$ Please draw the stack of the `collect_stairoids()` function right before the last line of C code is executed. Your picture should show each relevant part or item with a level of detail appropriate to understanding the behavior of the code (and the misbehavior of the TAs).

(b) $\boxed{\text{3 points}}$ Describe an input that will result in you being awarded a speed of more than 1 km/s. For full credit, specifying a shorter input is better.

(c) $\boxed{\text{1 point}}$ Briefly explain why your input results in a speed of more than 1 km/s.

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.