Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Fall 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.	**15**		
3.	**15**		
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.

   (a) $\boxed{\text{3 points}}$ 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) $\boxed{\text{4 points}}$ 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).

In lecture we discussed two ways that a virtual memory system could implement least-recently-used page replacement (don't worry, this isn't a virtual-memory implementation question!). The problem statement is: every time a page is referenced, an operation `reference(page_number)` is invoked; every now and then the page daemon invokes `lru_which(void)`, which returns the number of the page which was *least* recently referenced (note that calling `lru_which()` twice in a row should generally return two different page numbers, not the same page number twice).

The two approaches we discussed were a doubly-linked list of blobbies (where each blobby consisted of previous and next pointers and a page number) and a big array of per-page timestamps. In lecture we examined a design matrix in accordance with the 15-410 design orthodoxy, including two time metrics, with the values shown below. For the purposes of this problem, you may assume that those values are accurate even if some seem potentially suspicious (e.g., perhaps the blobbies live in an array, or perhaps there is a hash table somewhere).

	Blobby Queue	Page Stamps
Reference Work	7(?) stores	1 store
Eviction Work	3(?) stores	O(N)

(c) $\boxed{\text{3 points}}$ In the lecture, we didn't decide between these two options, because instead we studied a third option. But if you were required to pick one of these two options which one would you pick, and why? Please note that scoring will be based primarily on the rationale expressed.

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

2. $\boxed{\text{15 points}}$ Bridge problem

As you may know, Pittsburgh is known as "the city of bridges," by which we don't mean the "north bridge" and the "south bridge," but instead the Smithfield Street Bridge, the Fort Duquesne Bridge, etc. In this problem we will be studying a lesser-known bridge, the "Dinky Bridge," so called because it has only one lane (just one—*not* one in each direction). In addition to being narrow, the Dinky Bridge has a weight limit which allows only one car to be driving on it at any given time.

As each car approaches the Dinky Bridge, it invokes either `request(NORTH)` or `request(SOUTH)`; these operations may block the car for some time until the bridge is allocated to the car. When the `request()` operation returns, the car is allowed to drive across the bridge, at which point it invokes `complete()` (which does not take a parameter). So the standard control flow is

```
request(NORTH); drive(); complete();
```

or

```
request(SOUTH); drive(); complete();
```

You can imagine that `drive()` "takes a while," potentially being modelled via `sleep(15*100);`.

In this problem we will ask you to evaluate two proposed implementations of `request()` and `complete()`. While reading each one, you may assume that all synchronization objects are properly initialized and that mutexes begin in the unlocked state. It is probably in your best interest to study *both* implementations before beginning to write your solution to any part of this problem.

```
/* First implementation -- minimal! */

mutex_t bm;
cond_t done;
int available = 1;

void request(int direction) {
  (void) direction; // not used
  mutex_lock(&bm);
  while (!available)
    cond_wait(&done, &bm);
  available = 0;
  mutex_unlock(&bm);
}

void complete(void) {
  mutex_lock(&bm);
  available = 1;
  cond_signal(&done);
  mutex_unlock(&bm);
}
```

```
/* Second implementation - high quality! */

mutex_t bm;
cond_t done;
int next_ticket = 1;
int ready_for = 1;
extern void log_debug(char *s, ...);   // log_debug() is thread-safe

void request(int direction) {
  (void) direction; // neither used nor needed!

  int me, my_ticket;
  me = thr_getid();

  mutex_lock(&bm);
  my_ticket = next_ticket++;

  if (ready_for == my_ticket) {
    mutex_unlock(&bm);
    log_debug("%d happily crossing\n", me);
    return;
  } else {
    // I/O is not a "short instruction sequence"!
    mutex_unlock(&bm);
    log_debug("%d eagerly waiting\n", me);
    mutex_lock(&bm);

    while (ready_for != my_ticket)
      cond_wait(&done, &bm);
    mutex_unlock(&bm);

    log_debug("%d happily crossing\n", me);
  }
}

void complete(void) {
  mutex_lock(&bm);
  ++ready_for;
  cond_signal(&done);
  mutex_unlock(&bm);
}
```

(a) $\boxed{\text{6 points}}$ *Briefly and clearly* state the most problematic thread-synchronization problem you believe the $\boxed{\textit{first}}$ implementation suffers from. Show a *clear and compelling* execution trace which supports your claim.

You may use this page as extra space for the $\boxed{\textit{first}}$ trace if you wish.

(b) $\boxed{\text{7 points}}$ *Briefly and clearly* state the most problematic thread-synchronization problem you believe the $\boxed{\textit{second}}$ implementation suffers from. Show a *clear and compelling* execution trace which supports your claim.

You may use this page as extra space for the $\boxed{\textit{second}}$ trace if you wish.

(c) $\boxed{\text{2 points}}$ Please *clearly* describe a *simple* code change which solves the problem with the second implementation which you identified in part (b). *It should be possible for you to unambiguously describe the change in one sentence*—or maybe two. You may show code if you wish, but you probably shouldn't: we are looking for a *fix* to one of these algorithms, not a new algorithm. Once you have described the change, also briefly describe why it works.

3. $\boxed{\text{15 points}}$ Parallel-sorting deadlock.

For this problem, we will be considering a parallel sorting algorithm, though not a particularly good one. The program provided seeks to sort a randomly-generated array of size `SLOTS`. It spools up `NTHREADS` threads, each of which runs for a fixed number of iterations. In each iteration, a thread attempts to acquire two different slots with indices `x` and `y`. After acquiring them, it swaps them if necessary, then releases them. While acquiring the first slot, the thread will block if it has already been acquired. For anti-deadlock purposes, while acquiring the second slot, the thread may decide to release the first slot and start over. Unfortunately, this sorting program can deadlock!

You will find that `main()` does not do anything particularly interesting: it initializes the thread library, `rand_lock`, and array, then creates and joins the worker threads. You will also find that `rand_int()` is not particularly interesting; it simply generates a random number in a thread-safe manner (`genrand()` is not thread-safe).

```
int main() {
    thr_init(4096); // exam: no failure
    sgenrand(get_ticks());
    mutex_init(&rand_lock); // exam: no failure

    for (int i = 0; i < SLOTS; i++) {
        mutex_init(&array[i].mtx); // exam: no failure
        cond_init(&array[i].cvar); // exam: no failure
        array[i].owner = -1;
        array[i].waiters = 0;
        array[i].value = rand_int();
    }

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

    int inversions = 0;
    for (int i = 0; i < SLOTS; i++) {
        for (int j = i+1; j < SLOTS; j++)
            if (array[i].value > array[j].value)
                inversions++;
        mutex_destroy(&array[i].mtx);
        cond_destroy(&array[i].cvar);
    }
    printf("inversions: %d\n", inversions);

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

```
#define SLOTS 25
#define NTHREADS 20
#define ITERS 100

#define MAX(x,y) (((x) < (y)) ? (y) : (x))
#define MIN(x,y) (((x) < (y)) ? (x) : (y))

typedef struct {
    int owner;
    unsigned int value;
    int waiters; // bit-vector
    mutex_t mtx;
    cond_t cvar;
} slot_t;

static slot_t array[SLOTS];

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

void swap_slots(unsigned int x, unsigned int y) {
    int less = MIN(array[x].value, array[y].value);
    int more = MAX(array[x].value, array[y].value);
    array[x].value = x < y ? less : more;
    array[y].value = x < y ? more : less;
}

void release(int idx) {
    slot_t *s = &array[idx];
    mutex_lock(&s->mtx);
    s->owner = -1;
    mutex_unlock(&s->mtx);
    cond_broadcast(&s->cvar);
}
```

```
bool acquire(int desired_idx, int owned_idx, int id) {
    slot_t *desired = &array[desired_idx];
    slot_t *owned = owned_idx == -1 ? NULL : &array[owned_idx];

    int acquired = true;
    mutex_lock(&desired->mtx);

    if (desired->owner != -1) {
        desired->waiters |= (1 << id);
        while (desired->owner != -1) {
            if (owned && (owned->waiters & (1 << desired->owner))) {
                acquired = false;
                break;
            }
            cond_wait(&desired->cvar, &desired->mtx);
        }
        desired->waiters &= ~(1 << id);
    }

    if (acquired)
        desired->owner = id;

    mutex_unlock(&desired->mtx);
    return acquired;
}

void *sorter(void *arg) {
    int id = (int)arg;
    for (int iter = 0; iter < ITERS; iter++) {
        unsigned int x = rand_int() % SLOTS;
        unsigned int y = rand_int() % SLOTS;
        if (x == y) continue;

        acquire(x, -1, id); // first grab can't fail
        if (acquire(y, x, id)) {
            swap_slots(x, y);
            release(y);
        }
        release(x);
    }
    return NULL;
}
```

(a) $\boxed{\text{4 points}}$ Show *clear, convincing* evidence of deadlock. Begin by *describing the problem* in one or two sentences; then *clearly specify a scenario.* Explicitly indicate how each necessary deadlock ingredient is present in the scenario you describe.

(b) $\boxed{\text{8 points}}$ Now provide an execution trace resulting in a deadlock. 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 question if you wish.

(c) $\boxed{\text{3 points}}$ 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		

Here is a second illustative trace.

Time	Thread 0	Thread 1	Thread 2
0	i = acv_wait(a)		
1	...wait...	acv_signal(a)	
2		thr_create(T2)	
3			j = acv_wait(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.

```
#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 awakened. `acv_abort()` should not return until the abortable condition variable is no longer in use by any of the waiting threads. After a condition variable has been aborted, it is illegal for other threads to call `acv_wait()`, `acv_signal()`, `acv_abort()` on that condition variable.

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`.

The standard C run-time model, and the standard Unix process model, support various memory regions (because C and Unix grew up together, the similarity between the models is not a coincidence). As a result, a programmer can allocate memory in various "places," (or perhaps memory of various "kinds"). For the purposes of this problem, consider a programmer who wishes to allocate an array of ten `int` variables, with all elements initialized to zero.

(a) $\boxed{\text{3 points}}$ Show code which will allocate a ten-`int` array initialized to zeroes in one area/region/place.

(b) $\boxed{\text{3 points}}$ Show code which will allocate a ten-`int` array initialized to zeroes in a second area/region/place.

(c) $\boxed{\text{3 points}}$ Show code which will allocate a ten-`int` array initialized to zeroes in a *third* area/region/place.

(d) $\boxed{\text{1 point}}$ Of the allocations you showed above, which one is the *most reliable*? Why?

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.