Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (A), Spring 2026

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.	**20**		
4.	**15**		
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. [10 points] Short answer.

   (a) [4 points] List, and briefly explain, the "three kinds of error."

(b) [4 points] What are the two kinds of thread cancellation? Briefly state one inconvenient feature of each kind.

(c) [2 points] Briefly define "starvation" as the term applies to this class.

2. [15 points] Rendezvous

Concurrent programs use a variety of synchronization objects beyond the standard mutex and condition variable. For Project 2 you implemented semaphores and readers-writers locks, but applications use other synchronization objects as well.

One such object is the "rendezvous," which combines synchronization with data communication. Two threads wishing to synchronize "meet" at the same rendezvous object, each with a data item they wish to convey to the other thread. The first thread to arrive suspends its execution until the second arrives; when the second arrives, the two values are exchanged; the "meet" operation then returns to each thread indicating the value which was provided by the other thread.

The Plan 9 operating system exports rendezvous as a kernel primitive serving essentially the same purpose as `deschedule()` and `make_runnable()` do in Pebbles.

Here is some toy code demonstrating the usage of a rendezvous.

```
// "Exam mode": all creation code is assumed not to fail

rdv_t r;

void* worker(void* data) {
    int initial, other;
    initial = (int)data;

    other = rdv_meet(&r, initial); // swap values with someone else

    // We assume that printing a short string is "atomic enough".
    printf("[%d] Sent %d, received %d\n", thr_getid(), initial, other);
    return NULL;
}

int main(void) {

    int tids[NUM_THREADS];

    thr_init(PAGE_SIZE);
    rdv_init(&r);

    for (int i = 0; i < NUM_THREADS; i++)
        tids[i] = thr_create(worker, (void *) i);

    for (int i = 0; i < NUM_THREADS; i++)
        thr_join(tids[i], NULL);

    thr_exit(NULL);
    return 0;
}
```

Below is a proposed implementation of `rendezvous()` using P2 thread-library operations.

```
#include "thread.h"
#include "stdbool.h"

typedef struct rdv {

    mutex_t m;

    int entry_count;
    cond_t entry;

    bool has_waiter;
    cond_t waiter;

    int share_1;
    int share_2;

} rdv_t;

int rdv_init(rdv_t *r) {
    mutex_init(&r->m);      // exam mode: can't fail
    cond_init(&r->entry);   // exam mode: can't fail
    cond_init(&r->waiter);  // exam mode: can't fail

    r->entry_count = 0;
    r->has_waiter = false;
    r->share_1 = 0;
    r->share_2 = 0;
    return 0;
}
```

The remainder of this page is intentionally blank.

```
int rdv_meet(rdv_t *r, int share) {

    mutex_lock(&r->m);
    r->entry_count++;
    if (r->entry_count > 2) {
        // Block until we're allowed entry
        cond_wait(&r->entry, &r->m);
    }

    if (!r->has_waiter) {
        // We are the first to arrive
        r->has_waiter = true;
        r->share_1 = share;

        // Block until the 2nd thread arrives
        cond_wait(&r->waiter, &r->m);

        // Grab and update values
        int retval = r->share_2;
        r->entry_count -= 2;
        r->has_waiter = false;

        mutex_unlock(&r->m);
        // Ready for the next pair!
        cond_signal(&r->entry);
        cond_signal(&r->entry);

        return retval;

    } else {
        // We are the second to arrive
        r->share_2 = share;
        int retval = r->share_1;

        // Wake the waiter up
        mutex_unlock(&r->m);
        cond_signal(&r->waiter);

        return retval;
    }
}
```

Unfortunately, there is (at least) one synchronization problem in the code presented above. The problem is found in the rendezvous code presented to you (in other words, we are not looking for claims that `cond_wait()` 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. You may wish to refer to the "cheat sheets" at the end of the exam.

Please identify a synchronization problem in the rendezvous code. Then *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.*

This page can be used for your rendezvous answer.

This page can be used for your rendezvous answer if you wish.

3. [20 points] Theme-park ride synchronization.

In anticipation of the overwhelming box office success of the new *Snow White* remake, Disney has built a new ride at Disney World based on this movie. They expect thousands of visitors to the ride, but, due to the limited number of seats, only 143 visitors can be visiting the attraction at the same time. Other eager visitors must wait until someone exits the ride before they can enter.

To reward its most loyal customers, Disney wishes to give priority access to DisneyPlus subscribers. Those who have a DisneyPlus "Premium" subscription (highest level) will be granted entry first when a space becomes available, regardless of their place in line (i.e., they get highest-priority access to the attraction). The next-highest preference is for those with a DisneyPlus "Basic" subscription (middle level). Finally, someone without any subscription (lowest level) will be granted entry only if no one at higher subscription levels are waiting when a seat becomes available.

Your mission is to build a synchronization object to implement Disney's priority scheme for ride entry, under the assumption that each visitor is represented by a thread. The synchronization mechanism needs to work for a wide range of rides, which can vary in the number of seats and the number of priority levels. For example, the large *Snow White* ride might have 143 seats and 3 priority levels, while riding an individual pony might be modeled as a "ride" with exactly 1 seat (the pony's saddle), though it might still have the 3 priority levels described above. It would even be possible to model a restroom as a "ride" with 5 seats, though it might be best for it to have just 1 priority level.

Generally, the number of visitors to a given attraction/ride may be large (potentially thousands), but the number of priority levels is expected to be quite small (one, three, maybe 10, but not 100 or 1,000). The API for the priority-based ride synchronization mechanism is summarized below, and an example program showing usage is given on the next page.

```
/// @brief   Initialize ride_t.
/// @param   levels  Number of priority levels (>= 1)
/// @param   seats   Number of seats (>= 1)
/// @return          0 on success, -1 on failure
int ride_init( ride_t *r, int levels, int seats );

/// @brief   Wait to enter a ride/attraction.
///          This will block the thread while no seat is available.
/// @note    For now, strict priority service.
/// @param   r       Ride to wait for a seat on
/// @param   level   Priority level of waiting customer (greater level gets a seat sooner)
void ride_enter( ride_t *r, int level );

/// @brief   Depart from the ride, freeing a seat.
///          If some riders are waiting, one will be issued a seat.
/// @note    For now, strict priority service.
/// @param   r       Ride departing from
void ride_leave( ride_t *r );

/// @brief   Destroy ride_t.
/// @note    It is illegal to destroy a ride object that is in use.
/// @param   r       Object to destroy
void ride_destroy( ride_t *r );
```

```
/** @file riders.c */
#define NUM_LEVELS 3
#define NUM_SEATS 12
#define NUM_VISITORS 500

ride_t sample_ride;

volatile int park_is_closing = 0;

void *visitor( void *level ) {

    int tid = thr_getid();

    while (!park_is_closing) {
        printf("Thread %d: about to wait\n", tid);
        ride_enter(&sample_ride, (int) level);

        printf("Thread %d: riding!\n", tid);
        sleep(17);

        ride_leave(&sample_ride);
        printf("Thread %d: done!!\n", tid);

        sleep(187 % (int) level); // get a snack
    }
    return (level);
}

// exam: creation operations "won't fail"
int main(void)
{
    thr_init(5*1024); // exam
    ride_init( &sample_ride, NUM_LEVELS, NUM_SEATS ); // exam

    // generate visitors, with different subscription levels
    for (int i=0; i<NUM_VISITORS; i++)
        thr_create( visitor, (void*) (i % NUM_LEVELS)); // exam

    sleep(100*60*60*8); // Park is open for 8 hours

    park_is_closing = 1;

    thr_exit(0);
    return (0);
}
```

In this problem you will implement the ride-sharing synchronization mechanism on the page(s) that follow. Don't forget to include your definition of `ride_t`. You may use any of the P2 synchronization primitives in your implementation, and you may assume that they work correctly. You may assume that the mutex implementation has bounded waiting (e.g., is FIFO) and does not block threads. You may likewise assume a P2-compliant condition-variable implementation, which you may assume to be strictly-FIFO if you wish. You may *not* use low-level operations such as `deschedule()`/`make_runnable()` or any atomic instructions (`XCHG`, `LL/SC`). **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). **However, 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. You may use non-synchronization-related thread-library routines in the "`thr_xxx()` family," e.g., `thr_getid()`.

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 on the next page, your grade will suffer!

The remainder of this page is intentionally blank.

Please first declare a `struct ride` and implement `ride_init()`, then implement `ride_enter()` and `ride_leave()`, and then `ride_destroy()`.

This page can be used for your ride priority solution.

This page can be used for your ride priority solution if you wish.

This page can be used for your ride priority solution if you wish.

4. [15 points] N:1 threads.

Consider an N:1 thread library, aka "green threads." In such an environment, `thr_create()` does not invoke the `thread_fork` system call, perhaps because the operating system doesn't support multi-threaded applications.

If an application author wishes to use threads, those threads must be "logical," meaning that switching among the logical threads is the responsibility of library code in the application. In general there is some piece of code, perhaps called `int __tswitch(int tid)`. In the common case of round-robin scheduling of logical threads inside the application, various application and/or library code would invoke `__tswitch(-1)`, which would manipulate a thread queue inside the application, putting the current thread's control block at the end of the queue, pulling the thread off of the front of the queue, and then calling some assembly code to save the calling thread's registers somewhere and restore the switched-to thread's registers from somewhere. It is also possible that somebody might invoke `__tswitch()` with a thread i.d. other than `-1` to switch to a particular target thread, if possible.

In this question we will ask you to consider, and describe, behavioral characteristics of the six thread-library functions listed on page 26 whose names match `thr_*` (as opposed to the other standard thread-library functions). In particular, we will ask which of those six functions *should generally not* invoke `__tswitch()`, which of those functions *must sometimes or always* invoke `__tswitch()` (or a standard thread-library routine that invokes `__tswitch()`), and which of those functions *legitimately might* invoke `__tswitch()`, but might not, or might not always. We expect that each of the six `thr_*` functions will be placed in exactly one category.

The remainder of this page is intentionally blank.

(a) [4 points] Of the six `thr_*` functions listed on page 26, indicate which *should generally not* invoke `__tswitch()` (and should generally not invoke a standard thread-library routine that invokes `__tswitch()`). Briefly explain each answer.

(b) [4 points] Of the six `thr_*` functions listed on page 26, indicate which *must sometimes or always* invoke `__tswitch()` (or must sometimes or always invoke a standard thread-library routine that invokes `__tswitch()`). Briefly explain each answer.

(c) [4 points] Of the six `thr_*` functions listed on page 26, indicate which *legitimately might or might not* invoke `__tswitch()` (or a standard thread-library routine that invokes `__tswitch()`). For each function you list, briefly explain in which circumstance(s) it plausibly makes sense for that function to directly or indirectly invoke `__tswitch()`, and in which circumstance(s) it plausibly makes sense for the function to *not* directly or indirectly invoke `__tswitch()`.

(d) [3 points] Considering the `thr_*` function(s) that you placed in the "must sometimes or always" category, it is arguably fairly likely that the identified `thr_*` function(s) might not *directly* invoke `__tswitch()`, but might instead call a standard thread-library function that invokes `__tswitch()`. Note that it is plausible that the standard thread-library function that invokes `__tswitch()` might manipulate the thread queue before and/or after invoking `__tswitch()`. State the name of one standard thread-library function listed on page 26 that you believe is particularly likely to serve as this kind of "middle man," and explain your reasoning. (If you wish to name and justify more than one, that is fine.)

5. 10 points `Nuts and bolts`

Consider the following C code written for the Pebbles environment:

```
/** @file crt0.c */
#include <stdlib.h>
#include <syscall.h>

extern int main(int argc, char *argv[]);
extern void install_autostack(void *stack_high, void *stack_low);

void _main(int argc, char *argv[], void *stack_high, void *stack_low) {
  install_autostack(stack_high, stack_low);
  exit(main(argc, argv));
}
```

```
/** @file foo.c */
#include <stdlib.h>
#include <stdio.h>
#include <syscall.h>

static void foo(int argc, char *argv[]) {
  static int i;

  // Checkpoint Charlie

  for (i = 0; i < argc; i++) {
    printf("%s", argv[i]);
    // printf(argv[i]); sometimes this version crashes, not clear why
  }
  printf("\n");
}

int main(int argc, char *argv[]) {
  foo(argc, argv);
  return 0;
}
```

The program was launched through the shell with the following command:

```
[410-shell]$ foo a b unreal_mode
```

Draw a picture of the stack when the program reaches the comment labeled *Checkpoint Charlie*. Your diagram should include everything from the stack pointer up to *and including* the `argv` strings that will be printed. Label each memory location you are filling with its address (you may "draw" memory in bytes or words as you see fit). You may choose any "plausible" addresses you wish, as long as they make sense. Assume a 32-bit machine with any plausible byte order.

Space for the stack question

You may use this page as extra space for the stack question 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 */
```

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.