Computer Science 15-410/15-605: Operating Systems
Mid-Term Exam (B), Fall 2025

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

(a) $\boxed{\text{6 points}}$ If condition variables are built and used correctly, three properties should hold. Each of those properties is necessarily somewhat probabilistic, but each one should be true a large fraction of the time. List the three rules; for each one, provide a brief statement of why that rule is important when building an application.

(b) $\boxed{\text{2 points}}$ Which component of an operating system typically has a “top half” and a “bottom half”? How do those “halves” differ, and why?

(c) $\boxed{\text{2 points}}$ Briefly state two differences between a trap and an interrupt.

2. $\boxed{\text{15 points}}$ Pausable semaphores.

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.

Something we largely do *not* cover in this class, that is increasingly important to workers in the field of operating systems, is managing heat and power. It is one thing to correctly compute a required answer, but it is often important to perform a computation while remaining within a heat and/or power budget. One tool that might be used to balance computation versus heat/power constraints is a "pausable semaphore." That object would provide the standard semaphore services allowing a pool of threads to acquire and release logical items, but it would *also* allow a power-management thread to, from time to time, suspend and resume threads when they are in the act of acquiring and releasing those logical items. While most of the threads are invoking `psem_signal()` and `psem_wait()` on a pausable semaphore, a *single designated* power-management thread can invoke `psem_pause()` and `psem_resume()` (in strict alternation: each `pause` by the single power-management thread will be followed by exactly one `resume` by the single power-management thread, and then later there may be more `pause/resume` sequences by the power-management thread as appropriate). While a pausable semaphore is paused, `signal()` and `wait()` operations *both* cause threads to block; when the pausable semaphore is resumed, threads "artifically" blocked by the pause operation resume, and `signal()` and `wait()` operate normally. Like a regular semaphore, a pausable semaphore object does not know how many threads will invoke it.

A small example program using a pausable semaphore is displayed on the next page.

The remainder of this page is intentionally blank.

```
// Assumes that printf() of a short single-line message is atomic.
// Test code:  not required to exhibit reasonable synchronization behavior.

#define WIDTH 8 // number of coprocessors
#define THREADS 16 // number of clients

#define RUN_TICKS 10
#define SLEEP_TICKS 5
#define TOTAL_TICKS 15410

static psem_t limiter;

void *threadbody(void *vid) {
  int id = (int) vid;
  int base;

  while (1) {
    psem_wait(&limiter);
    printf("Thread %d running...fans will be ON\n", id);
    base = get_ticks();
    while (get_ticks() < base + RUN_TICKS)
      continue;
    printf("Thread %d done for now\n", id);
    psem_signal(&limiter);
    sleep(SLEEP_TICKS);
  }
  return (0);
}

int main(void) {
  thr_init(8192); // exam: cannot fail
  psem_init(&limiter, WIDTH); // exam: cannot fail

  printf("Starting %d threads\n", THREADS);

  for (int n = 0; n < THREADS; ++n)
    affirm(thr_create(threadbody, (void *)n) > 0); // exam: cannot fail

  int base, ticks;

  base = get_ticks();
  while ((ticks = get_ticks()) < base + TOTAL_TICKS) {
    sleep(ticks % 67); // heating is likely now
    psem_pause(&limiter);
    sleep(ticks % 23); // cool down a bit
    psem_resume(&limiter);
  }

  printf("All done!\n");
  task_vanish(0);
  panic("task_vanish() didn't???");
}
```

Below is a proposed implementation of pausable semaphores.

```
typedef struct {
  mutex_t m;

  sem_t sem;

  volatile bool is_resuming;
  cond_t resume_cvar;

  volatile bool is_paused;
  volatile int paused_count;
  cond_t paused_cvar;
} psem_t;

int psem_init(psem_t *psp, int count) {
  mutex_init(&psp->m);          // can't fail
  cond_init(&psp->paused_cvar); // can't fail
  sem_init(&psp->sem, count);   // can't fail

  psp->is_resuming = false;
  psp->is_paused = false;

  psp->paused_count = 0;

  return 0;
}

void psem_destroy(psem_t *psp) { /* omitted for exam brevity */ }
```

The remainder of this page is intentionally blank.

```
void psem_wait(psem_t *psp) {
  mutex_lock(&psp->m);

  // To be fair to previously paused threads, wait here until they are resumed
  while (psp->is_resuming) {
    cond_wait(&psp->resume_cvar, &psp->m);
  }

  if (psp->is_paused) {
    psp->paused_count++;
    cond_wait(&psp->paused_cvar, &psp->m);
  }

  mutex_unlock(&psp->m);

  sem_wait(&psp->sem);
}

void psem_signal(psem_t *psp) {
  mutex_lock(&psp->m);

  // To be fair to previously paused threads, wait here until they are resumed
  while (psp->is_resuming) {
    cond_wait(&psp->resume_cvar, &psp->m);
  }

  if (psp->is_paused) {
    psp->paused_count++;
    cond_wait(&psp->paused_cvar, &psp->m);
  }

  mutex_unlock(&psp->m);

  sem_signal(&psp->sem);
}
```

The remainder of this page is intentionally blank.

```
void psem_pause(psem_t *psp) {
  mutex_lock(&psp->m);

  psp->is_paused = true;

  mutex_unlock(&psp->m);
}

void psem_resume(psem_t *psp) {
  mutex_lock(&psp->m);

  // While resuming is not finished, new threads should wait
  psp->is_resuming = true;

  int resume_count = psp->paused_count;

  psp->paused_count = 0;
  psp->is_paused = false;

  mutex_unlock(&psp->m);

  while (resume_count > 0) {
    cond_signal(&psp->paused_cvar);
    resume_count--;
  }

  // Paused threads are resumed, now everyone can compete
  psp->is_resuming = false;
  cond_broadcast(&psp->resume_cvar);
}
```

There is at least one synchronization problem with the above pausable-semaphore implementation. Briefly summarize one such problem and then present a trace, using the format presented in class, that supports your claim.

Be sure that the execution trace you provide us with is easy to read and conclusively demonstrates the claim you are making. You may introduce temporary variables or other obvious notation as necessary to improve the clarity of your answer. *You should report a problem with code that is visible to you rather than assuming a problem in code that you have not been shown.* It is possible to answer this question with a clear trace, so you should do what is necessary to ensure that you do. It is *strongly recommended* that you write down a draft version of any execution trace using the scrap paper provided at the end of the exam, or on the back of some other page, *before* you begin to write your solution on the next page. If we cannot understand the solution you provide, your grade will suffer!

This page is for your pausable-semaphore solution.

This page can be used for your pausable-semaphore solution.

This page can be used for your pausable-semaphore solution if you wish.

3. $\boxed{\text{20 points}}$ Super semaphores

The semaphore is a thread-synchronization primitive based on the metaphor of a pool of identical resources; threads request a resource from the pool by calling `sem_wait()`. In some situations, threads frequently need multiple resources at the same time. In this problem, your job will be to implement a version of semaphores where both `sem_wait()` and `sem_signal()` are extended to include a count indicating the number of resources being acquired or released, respectively. To simplify your job, users of these "super semaphores" are restricted as follows: once a thread acquires resources using `sem_wait()`, it is *guaranteed* not to call `sem_wait()` again until it has invoked `sem_signal()` one or more times in such a way that it has released all resources from this particular "super semaphore." Also, it is guaranteed that a single thread will never request more resources from a single "super semaphore" than the peak number of resources that the "super semaphore" has ever held. Thus, usage examples "A" and "B" below are invalid. If you wish, you may optionally *state as an assumption* that a "super semaphore" will never have more resources available than when it was initialized (*if you state that assumption*, then usage example "C" below is invalid). For exam purposes, you may assume that "guarantees" are *always* true and your code does not need to verify they are true or react if they aren't true.

Example A (invalid!) Thread 0
`sem_init(s,3)`
`sem_wait(s,2)`
`sem_signal(s,1)`
`sem_wait(s,1) // still holding 1!`

Example B (invalid!) Thread 0
`sem_init(s,3)`
`sem_wait(s,4) // stuck!`

Example C (??) Thread 0
`sem_init(s,3)`
`sem_signal(s,1) // 4 > 3`

Here is a sample trace of legal operation (many other traces are possible!).

Time	Thread 0	Thread 1	Thread 2	Thread 3
0	sem_init(s,4)			
1	sem_wait(s,4)			
2	...run...	sem_wait(s,1)		
3		...wait...	sem_wait(s,3)	
4			...wait...	sem_wait(s,3)
5				...wait...
6	sem_signal(s,2)			
7		...run...		
8	sem_signal(s,2)			
9			..run...	
10			sem_signal(s,3)	
11				...run...
12		sem_signal(s,1)		

(Continued on next page)

Your mission

In this problem you will implement "super semaphores" using mutexes and condition variables. You may *not* use other atomic or thread-synchronization synchronization operations, such as, but not limited to: semaphores (obviously), reader/writer locks, `deschedule()/make_runnable()`, or any atomic instructions (`XCHG`, `LL/SC`). You may assume that there is a mutex implementation available for use, with the same interface as provided with P2; you may assume that this 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. **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!

There are multiple "legal" solutions (ones that operate defensibly or "reasonably well" for every valid execution sequence). Some solutions that are not only legal but also "nice" are too long and complicated for us to expect them as exam solutions. Observe that, compared to regular semaphores, these "super semaphores" have "interesting" states, in which multiple resources are available and multiple threads, with potentially different needs, are waiting. If you can spend a little more design time (say, ten minutes) to get a reasonable-length solution which is not only legal but also handles some of the "interesting" states in a "nice" fashion as opposed to a "crude" fashion, you will receive a higher score (we expect valid answers to fall into three different classes, which may be scored differently). But don't spend *too* much time thinking about "nice"... getting something that works is more important, plus there are other questions on the exam!

The remainder of this page is intentionally blank.

Please declare a `struct sem` and implement:

- `int sem_init(sem_t *sem, int initial)`,
- `void sem_wait(sem_t *sem, int request)`,
- `void sem_signal(sem_t *sem, int release)`

You do not need to implement `sem_destroy()`.

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

```
typedef struct sem {
















} sem_t;


typedef struct aux {









} aux_t;
```

...space for super-semaphore implementation...

You may use this page as space for your super-semaphore solution if you wish.

You may use this page as space for your super-semaphore solution if you wish.

4. $\boxed{\text{10 points}}$ Process model.

In this question we will consider whether various Pebbles system calls are expected to block threads or are expected to generally complete without blocking. In order to clarify the issue, you should probably imagine that a Pebbles-compliant kernel is running on multiple processors (this actually happened during Spring 2012 and might happen again). Also note that "block" is not the same concept as "might require a lock"—as you will soon experience directly, *many* system calls require some locking.

(a) $\boxed{\text{2 points}}$ Explain briefly what it means for a system call to block a thread, or for a thread to be blocked in a system call.

For each Pebbles system call listed below (in alphabetical order), briefly argue that the system call either *should generally not block threads* (for some stated reason(s)) or *is expected to block threads* (in some specified scenario(s)).

(b) $\boxed{\text{2 points}}$ `deschedule()`

(c) $\boxed{\text{2 points}}$ `gettid()`

(d) $\boxed{\text{2 points}}$ `make_runnable()`

(e) $\boxed{\text{2 points}}$ `new_pages()`

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

5. $\boxed{\text{10 points}}$ `Nuts and bolts`

(a) $\boxed{\text{4 points}}$ Write a sequence of up to five x86-32 instructions which, when run, will have the effect of placing the address of one of the instructions (you get to pick which one) into `%eax`. It is OK for you to destroy the values in other registers.

(continued on next page)

(b) $\boxed{\text{6 points}}$ Consider the following C code:

```
int g;              /* 1 */

void foo() {
    int i=1;        /* 2 */
    static int s;   /* 3 */
}

void bar() {
    static int s=1; /* 4 */
}

int main(int argc, char **argv) /* 5, 6 */
{
    return 0;
}
```

For each variable, indicate with a check mark where in memory it is stored at runtime. Assume `-O0`, where all variables are initially stored in memory rather than registers.

	text	data	bss	heap	stack
1. `g`					
2. `i`					
3. `s`					
4. `s`					
5. `argv`					
6. `argv[0]`					

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.