This is a closed book exam with one 2-sided page of notes permitted. It is intended to be a 50
minute exam. You have 80 minutes to complete it. The number at the beginning of each question
indicates the points for that question. Write all of your answers directly on this paper. Make
your answers as concise as possible. If there is something in a question that you believe is open to
interpretation, please raise your hand to request clarification. When told to open the exam, put
your login on every page and check that you have them all. (Final page is for reference.)

By my signature below, I swear that this exam is my own work. I have not obtained answers or
partial answers from anyone. Furthermore, if I am taking the exam early, I promise not to discuss
it with anyone prior to completion of the regular exam, and otherwise I have not discussed it with
anyone who took the early alternate exam.

X ______________________________

Grade Table (for instructor use only)

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. (25 points) Operating Systems Concepts

(a) (20 points) Choose either true or false for the below questions. You do not need to provide justifications.

i. (2 points) A user program can request an operating system service by issuing a system call.
   - True
   - False

ii. (2 points) Threads in a single process are prevented by the operating system from accessing the stacks of other threads.
    - True
    - False

iii. (2 points) A pintos kernel thread enters thread_switch with interrupts disabled, so thread_switch enables them before switching to the next thread.
     - True
     - False

iv. (2 points) The scheduler places RUNNING threads on a queue according to a scheduling algorithm.
    - True
    - False

v. (2 points) In operating systems like Unix, different operations are used for accessing files, transferring information through sockets, and communication between processes.
    - True
    - False

vi. (2 points) In operating systems like Unix, different operations are used for setting file transfers and network communication.
    - True
    - False

vii. (2 points) Interrupt handlers can call pthread_mutex_lock because it is atomic.
     - True
     - False

viii. (2 points) Disabling interrupts always protects a critical section of code.
      - True
      - False

ix. (2 points) A successful call to fork() always returns twice - once in the child process and once in the parent process.
    - True
    - False

x. (2 points) Path resolution is implemented by the kernel.
    - True
    - False
(b) (5 points) Which of the following is true about Round Robin Scheduling? Select all the choices that apply.

- Cache performance is likely to improve relative to FCFS.
- If no new threads are entering the system all threads will get a chance to run in the cpu every $\text{QUANTA} \times \text{SECONDS\_PER\_TICK} \times \text{NUMTHREADS}$ seconds. (Assuming $\text{QUANTA}$ is in ticks).
- This is the default scheduler in Pintos.
- If quanta is constantly updated to become the number of cpu ticks since boot, Round Robin becomes FIFO.
- If all threads in the system have the same priority, Priority Schedulers must behave like round robin.

2. (25 points) Processes and Threads
For the following consider the following nearly identical process based and thread based programs. For each question, answer YES or NO and provide a short, crisp explanation. Assume that functions will be well-behaved (e.g., malloc will succeed) and that printf() is atomic. (We will answer half the first part to provide an example.)
void process() {
    pid_t pid;
    int status, rd;
    int j = 1;
    buf = strcpy(malloc(100), "boring");
    pid = fork();
    if (pid != 0) {
        /* parent */
        printf("P Parent: j=%d\n", j);
        i = 162;
        j = 2;
        printf("P Parent: j=%d\n", j);
        printf("P Parent: %s\n", buf);
        j = 4;
        wait(&status);
    } else {
        /* child */
        printf("P Child: i=%d, j=%d\n", i, j);
        printf("P Child: i=%d, j=%d\n", i, j);
        j = 3;
        strcpy(buf, "cool");
        exit(0);
    }
}

void thread() {
    pthread_t tid;
    int j = 1;
    buf = strcpy(malloc(100), "boring");
    pthread_create(&tid, NULL, tfun, NULL);
    printf("T Child: i=%d, j=%d\n", i, j);
    printf("T Child: i=%d, j=%d\n", i, j);
    j = 3;
    strcpy(buf, "cool");
    pthread_exit(NULL);
}

(a) (5 points) Can the output of the process-based program ever include (meaning they appear in this order, possibly with other output interleaved) the following? Why? (We are going to answer this one for you!)

P Parent: j=1
P Child: i=100, j=1
P Parent: j=2

YES. The child process may get switched in after the parent prints its initial value of its stack variable j and the parent later switch back in. The child obtained a copy of the parent’s stack and heap at the fork and it has changed neither when it first prints.
Can the output of the thread-based program ever include the following? Why?

T Parent: \( j = 1 \)
T Child: \( i = 100, j = 1 \)
T Parent: \( j = 2 \)

(b) (10 points) (be careful) Can the output of the process-based program ever include the following? Why?

P Parent: \( j = 1 \)
P Parent: \( j = 2 \)
P Child: \( i = 100, j = 1 \)

Can the output of the thread-based program ever include the following? Why?

T Parent: \( j = 1 \)
T Parent: \( j = 2 \)
T Child: \( i = 100, j = 1 \)

(c) (10 points) Can the output of the process-based program ever include the following? Why?

P Parent: cool

Can the output of the thread-based program ever include the following? Why?

T Parent: cool
3. (25 points) **Synchronized Objects**

(a) (15 points) Assuming malloc is threadsafe, modify (by showing where code needs to be inserted) the following unbounded stack abstraction to make `ppush` and `ppop` threadsafe using pthreads operations. We have started it for you by adding the lock to the pdl struct. (You do not need to eliminate the busy-wait. It only needs to be threadsafe, not highly efficient. You may not need to fill every line indicated.)

(b) (10 points) Fill in `dequeue` to create a thread-safe function that waits until the stack is non-empty, removes and item and returns it.

typedef struct item {
    struct item *next;
    void *val;
} item_t;

typedef struct pdl {
    item_t *head;

    -------------------------
} pdl_t;

pdl_t *new_pdl() {
    pdl_t *s = malloc(sizeof(pdl_t));
    s->head = NULL;
    pthread_mutex_init(&s->pdl_lock, NULL);
    return s;
}

void ppush (pdl_t *s, void *val) {

    --------------------------
    item_t *new_item;

    --------------------------
    new_item = malloc(sizeof(item_t));
    new_item->val = val;
    new_item->next = s->head;
    s->head = new_item;

    --------------------------
}

void dequeue (pdl_t *s) {

    --------------------------
    item_t *new_item;

    --------------------------
    new_item = malloc(sizeof(item_t));
    new_item->next = s->head;
    new_item->val = s->head->val;
    s->head = new_item->next;

    --------------------------
}
void *ppop(pdl_t *s) {

_______________________________
item_t *pop;
pop = s->head;
void *res = NULL;
if (pop) {

_______________________________
_______________________________
res = pop->val;
s->head = pop->next;
free(pop);
}

_______________________________
return res;
}

bool empty_pdl(pdl_t *s) {
    return (s->head == NULL);
}

void *dequeue(pdl_t *s) {
    void * res;

_______________________________
_______________________________
_______________________________
_______________________________
    pthread_yield();
_______________________________
_______________________________
return res;
}
4. (25 points) **Operating System Implementation: Pintos**

You design a new scheduler, you call it TFS. The idea is relatively simple, if a thread makes it to the end of a round robin scheduling quanta it sets its own priority to \(-1\) times the number of ticks it has ever spent in the cpu.

You may make the following assumptions in this problem.

- Priority scheduling in Pintos is functioning properly,
- Priority donation is not implemented.
- Alarm is not implemented.
- `thread_set_priority` is never called by the thread
- You may ignore the limited set of priorities enforced by pintos (priority values may span from \(\text{INT_MIN}\) to \(\text{INT_MAX}\)).

i. (4 points) Below is the declaration of `struct thread`, what field(s) would we need to add to make TFS possible. You may not need all the blanks.

```c
struct thread {
    /* Owned by thread.c. */
    tid_t tid; /* Thread identifier. */
    enum thread_status status; /* Thread state. */
    char name[16]; /* Name (for debugging purposes). */
    uint8_t *stack; /* Saved stack pointer. */
    int priority; /* Priority. */
    struct list_elem allelem; /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem; /* List element. */

    #ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir; /* Page directory. */
    #endif

    _____________________________ /* What goes here? */
    _____________________________ /* What goes here? */
    _____________________________ /* What goes here? */

    /* Owned by thread.c. */
    unsigned magic; /* Detects stack overflow. */
};
```
ii. (8 points) A what is needed for `thread_tick()` for TFS to work properly. You may not need all the blanks.

```c
void
thread_tick (void)
{
    struct thread *t = thread_current ();

    /* Update statistics. */
    if (t == idle_thread)
        idle_ticks++;
    #ifdef USERPROG
    else if (t->pagedir != NULL)
        user_ticks++;
    #endif
    else
        kernel_ticks++;

    /* Enforce preemption. */
    if (++thread_ticks >= TIME_SLICE) {
        intr_yield_on_return ();

    }

}
```
iii. (5 points) In a few short words, describe the overall behavior of this scheduler. How is it different from regular round robin or FIFO scheduler.

iv. (4 points) What could you do to "trick" this scheduler to giving a thread more cpu time. (Hint: thread_ticks is set to 0 every time the scheduler picks a new thread)

v. (4 points) What is a flaw in this scheduler when you have a steady state of threads being created, and very little threads destroyed. (Hint: Think about what happens when the scheduler has run for a really long time).
/******************************** Pthreads ********************************/
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                        const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);

/****************************** Strings & Processes ******************************/
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);

/****************************** Pintos ******************************/