Name (Last, First): ____

This exam consists of 4 questions on 8 pages; be sure you have the entire exam before starting. The point value of each question is indicated at its beginning; the entire exam is worth 100 points. Individual parts of a multi-part question are generally assigned approximately the same point value: exceptions are noted. This exam is open text and notes. However, you may NOT share material with another student during the exam.

Be concise and clearly indicate your answer. Presentation and simplicity of your answers may affect your grade. Answer each question in the space following the question. If you find it necessary to continue an answer elsewhere, clearly indicate the location of its continuation and label its continuation with the question number and subpart if appropriate.

You should read through all the questions first, then pace yourself.

The questions begin on the next page.

Problem	Possible	Score
1	25	
2	25	
3	25	
4	25	
Total	100	

1. (____________/25 points)

   Short answer questions (continued on next page)

   (a) Recall in Project 0 you could choose between sending results as ASCII characters or as binary data between the children workers and the parent process. Explain a benefit of using ASCII. Explain a benefit of using binary data.

   (b) Write a small C code snippet that dynamically determines the maximum number of threads you can create in Pintos. Assume your code will execute in the main thread after Pintos starts up.

(c) Consider the following partial solution to the Alarm problem in Project 1. Explain any possible problems with this piece of code.

```
timer_sleep (int64_t ticks)
{
  struct thread *t = thread_current ();

  t->wakeup_time = timer_ticks () + ticks;

  list_insert_ordered (&wait_list, &t->timer_elem,
                       compare_threads_by_wakeup_time, NULL);

  sema_down (&t->timer_sema);
}
```

(d) Explain how the BSD scheduler enables low priority threads to eventually run. That is, how does the BSD scheduler prevent starvation? Consider the formula for calculating priority:

```
verb+priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)+.
```

2. (______________/25 points)

   User-level Memory Allocation

   Recall your implementation of the user-level memory allocator API from Project 0.

```
/* include files omitted */
int main(int argc, char *argv[])
{
        uint8_t *a, *b, *c, *d;

        printf("sizeof(struct used_block) = %u\n", sizeof(struct used_block));
        printf("sizeof(struct free_block) = %u\n", sizeof(struct free_block));
        mem_init(memory, 128);










        c = mem_alloc(32);
        printf("c_rel = %u\n", (unsigned) (c - memory));
        exit(0);
}
```

   For this problem you need to add `mem_alloc()` and/or `mem_free()` function calls so that the output of the program is:

```
sizeof(struct used_block) = 4
sizeof(struct free_block) = 12
c_rel = 64
```

   You can only add calls to `mem_alloc()` and `mem_free()`.

3. (____________/25 points)

   Barrier Synchronization

   Barrier synchronization is useful when you need two or more threads to reach a particular point before continuing. This is useful in computations that work in stages. A new function called `barrier_wait()` will block all the threads participating in the barrier until the last thread reaches the barrier. Once the last thread reaches the barrier all the threads can continue. Consider the following sample code:

```
struct barrier tbarrier, gbarrier;

static void
a_thread_func (void *aux) {
  int id = (int) aux;
  int i;

  for (i = 0; i < 2; i++) {
    barrier_wait(&tbarrier)
    print ("Thread %d finished stage %d.", id, i)
  }
  barrier_wait(&gbarrier)
}

void main (void)
{
  int count = 3;
  int i;

  barrier_init(&tbarrier, count)
  barrier_init(&gbarrier, count + 1)

  for (i = 0; i < count; i++) {
    thread_create("Thread", 0, a_thread_func, (void *) i)
  }

  barrier_wait(&gbarrier)
  print ("All done.")
}
```

   The output of this code could be:

```
Thread 0 finished stage 0
Thread 1 finished stage 0
Thread 1 finished stage 1
Thread 0 finished stage 1
All done.
```

   For this problem you are going to provide an implementation of `barrier_init()` and `barrier_wait()`. You also need to provided the definition of the `barrier` struct. The `barrier_init()` function takes a pointer to a `barrier` struct and a `thread_count`, which indicates how many threads will be involved in the barrier. The `barrier_wait()` function takes a pointer to a `barrier` struct. Hint:consider all possible interleavings of the threads to ensure you solution is correct; this problem is harder than it looks. Provide your implementation on the next page. **Explain how it works.**

Solution to Problem 4, Barrier Synchronization

```
struct barrier {







};

void
barrier_init(struct barrier *b, int thread_count)
{








}

void
barrier_wait(struct barrier *b)
{
















}
```

4. (______________/25 points)

   Implementing palloc() for User Programs

   Recall that user programs in Project 2 cannot dynamically allocate memory. For this problem you are going to implement two new system calls, `alloc_pages()` and `free_pages()` that allows a user level program to allocate contiguous pages dynamically and have them installed in the users's virtual address space. For simplicity, you can assume that a user program can only have **one** dynamically allocated region, but the user can request `num_pages` pages to be allocated. Here are the prototypes for `alloc_pages()` and `free_pages()`:

```
void *alloc_pages(int num_pages);
void free_pages();
```

   Some things to consider:

   - `alloc_pages()` returns a pointer to the contigous pages in the user's virtual address space.
   - Where to place the memory request in the user's virtual address space.
   - How to keep track of the request so that it can be deallocated.
   - What is a reasonable limit on `num_pages`?

   For this problem, you only need to provided the kernel implementations of the new system calls. You do **not** need to show how to add them to your `syscall_handler()`. To give you some help, here is the `setup_stack()` code from `process.c`

```
static bool
setup_stack (void **esp)
{
  uint8_t *kpage;
  bool success = false;

  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
      if (success)
        *esp = PHYS_BASE;
      else
        palloc_free_page (kpage);
    }
  return success;
}
```

   Provide your implementations of `alloc_pages()` and `free_pages()` on the next page. **Explain your answer and any assumptions you make.**

Provide your solution to Problem 5, palloc() for user programs here.

Continue your answers here if necessary.