1 Warmup

1.1 Hello World

What does C print in the following code?

```c
void* identify(void* arg) {
    pid_t pid = getpid();
    printf("My pid is %d\n", pid);
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &identify, NULL);
    identify(NULL);
    return 0;
}
```

2 Vocabulary

- **thread** - a thread of execution is the smallest unit of sequential instructions that can be scheduled for execution by the operating system. Multiple threads can share the same address space, but each thread independently operates using its own program counter.

- **pthreads** - A POSIX-compliant (standard specified by IEEE) implementation of threads. A pthread_t is usually just an alias for "unsigned long int".

- **pthread_create** - Creates and immediately starts a child thread running in the same address space of the thread that spawned it. The child executes starting from the function specified. Internally, this is implemented by calling the clone syscall.

  ```c
  /* On success, pthread_create() returns 0; on error, it returns an error number, and the contents of *thread are undefined. */
  int pthread_create(pthread_t *thread, const pthread_attr_t *attr, 
                  void *(*start_routine) (void *), void *arg);
  ```

- **pthread_join** - Waits for a specific thread to terminate, similar to waitpid(3).

  ```c
  /* On success, pthread_join() returns 0; on error, it returns an error number. */
  int pthread_join(pthread_t thread, void **retval);
  ```

- **pthread_yield** - Equivalent to thread_yield() in Pintos. Causes the calling thread to vacate the CPU and go back into the ready queue without blocking. The calling thread is able to be scheduled again immediately. This is not the same as an interrupt and will succeed in Pintos even if interrupts are disabled.

  ```c
  /* On success, pthread_yield() returns 0; on error, it returns an error number. */
  int pthread_yield(void);
  ```
3 Problems

3.1 Join

What does C print in the following code?
(Hint: There may be zero, one, or multiple answers.)

```c
void *helper(void *arg) {
    printf("HELPER\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    printf("MAIN\n");
    return 0;
}
```

How can we modify the code above to always print out "HELPER" followed by "MAIN"?
3.2 Stack Allocation

What does C print in the following code?

```c
void *helper(void *arg) {
    int *num = (int*) arg;
    *num = 2;
    return NULL;
}

int main() {
    int i = 0;
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, &i);
    pthread_join(thread, NULL);
    printf("i is %d\n", i);
    return 0;
}
```

3.3 Heap Allocation

What does C print in the following code?

```c
void *helper(void *arg) {
    char *message = (char *) arg;
    strcpy(message, "I am the child");
    return NULL;
}

int main() {
    char *message = malloc(100);
    strcpy(message, "I am the parent");
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, message);
    pthread_join(thread, NULL);
    printf("%s\n", message);
    return 0;
}
```
3.4 Threads and Processes

What does C print in the following code?
(Hint: There may be zero, one, or multiple answers.)

```c
void *worker(void *arg) {
    int *data = (int *) arg;
    *data = *data + 1;
    printf("Data is %d\n", *data);
    return (void *) 42;
}

int main() {
    int data;
    int status;
    data = 0;
    pthread_t thread;

    pid_t pid = fork();
    if (pid == 0) {
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
    } else {  
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
        wait(&status);
    }
    return 0;
}
```

How would you retrieve the return value of worker? (e.g. "42")
3.5 Context Switching

Refer to the “Pintos Context Switch” section at the end of this discussion worksheet to answer these questions:

How many stacks are involved in a context switch? Identify the purpose of each stack.

The value of \texttt{SWITCH\_CUR} is 20. The value of \texttt{SWITCH\_NEXT} is 24. With this information, please draw the stack frame of \texttt{switch\_threads} for a thread that is about to switch the stack pointer to the next thread’s stack. Your stack frame should include the arguments \texttt{cur} and \texttt{next}.

In addition to the code inside \texttt{switch\_threads}, what other actions are required to perform a context switch between 2 user program threads?

In order to perform a context switch, the kernel must copy all of a thread’s registers onto the CPU’s registers. How is the %eip (instruction pointer) register copied onto the CPU? Identify which instruction is responsible for this.
3.6 Interrupt Handlers

Refer to the “Pintos Interrupt Handler” section at the end of this discussion worksheet to answer these questions:

What do the instructions pushal and popal do?

The interrupt service routine (ISR) must run with the kernel’s stack. Why is this the case? And which instruction is responsible for switching the stack pointer to the kernel stack?

The pushal instruction pushes 8 values onto the stack (32 bytes). With this information, please draw the stack at the moment when “call intr_handler” is about to be executed.

What is the purpose of the “pushl %esp” instruction that is right before ”call intr_handler”?

Inside the intr_exit function, what would happen if we reversed the order of the 5 pop instructions?
3.7 Pintos Context Switch

```assembly
 allocate

 ; Switch from CUR, which must be the running thread, to NEXT,
 ; which must also be running, switch_threads(), returning CUR in
 ; NEXT's context.

 ; This function works by assuming that the thread we're switching
 ; into is also running switch_threads(). Thus, all it has to do is
 ; preserve a few registers on the stack, then switch stacks and
 ; restore the registers. As part of switching stacks we record the
 ; current stack pointer in CUR's thread structure.

 .globl switch_threads
 .func switch_threads
 switch_threads:
     # Save caller's register state.
     #
     # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
     # but requires us to preserve %ebx, %ebp, %esi, %edi. See
     # [SysV-ABI-386] pages 3-11 and 3-12 for details.
     #
     # This stack frame must match the one set up by thread_create()
     # in size.
     pushl %ebx
     pushl %ebp
     pushl %esi
     pushl %edi

     # Get offsetof (struct thread, stack).
     .globl thread_stack_ofs
     mov thread_stack_ofs, %edx

     # Save current stack pointer to old thread's stack, if any.
     movl SWITCH_CUR(%esp), %eax
     movl %esp, (%eax, %edx, 1)

     # Restore stack pointer from new thread's stack.
     movl SWITCH_NEXT(%esp), %ecx
     movl (%ecx, %edx, 1), %esp

     # Restore caller's register state.
     popl %edi
     popl %esi
     popl %ebp
     popl %ebx
     ret
 .endfunc
```
3.8 Pintos Interrupt Handler

```
/**
 * An example of an entry point that would reside in the interrupt
 * vector. This entry point is for interrupt number 0x30.
 */

.func intr30_stub
intr30_stub:
  pushl %ebp /* Frame pointer */
  pushl $0 /* Error code */
  pushl $0x30 /* Interrupt vector number */
  jmp intr_entry
.endfunc

/* Main interrupt entry point.

An internal or external interrupt starts in one of the
intrNN_stub routines, which push the `struct intr_frame`
frame_pointer, error_code, and vec_no members on the stack,
then jump here.

We save the rest of the `struct intr_frame` members to the
stack, set up some registers as needed by the kernel, and then
call intr_handler(), which actually handles the interrupt.

We "fall through" to intr_exit to return from the interrupt.
*/

.func intr_entry
intr_entry:
  /* Save caller’s registers. */
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  /* Set up kernel environment. */
  cld /* String instructions go upward. */
  mov $SEL_KDSEG, %eax /* Initialize segment registers. */
  mov %eax, %ds
  mov %eax, %es
  leal 56(%esp), %ebp /* Set up frame pointer. */

  /* Call interrupt handler. */
  pushl %esp
  .globl intr_handler
  call intr_handler
  addl $4, %esp
.endfunc
```
/* Interrupt exit. 
Restores the caller’s registers, discards extra data on the 
stack, and returns to the caller. 
This is a separate function because it is called directly when 
we launch a new user process (see start_process() in 
userprog/process.c). */
.globl intr_exit
.func intr_exit
intr_exit:
    /* Restore caller’s registers. */
popal
    popl %gs
    popl %fs
    popl %es
    popl %ds
    /* Discard ‘struct intr_frame’ vec_no, error_code, 
frame_pointer members. */
    addl $12, %esp
    /* Return to caller. */
    iret
.endfunc