2. Processes and System Calls [6 marks; 3 each]

A process, P1, wants to terminate another target process, P2, using the `kill` system call (with the `SIGKILL` argument, which instructs the target process to exit).

(a) Why can't the operating system locate and destroy the thread struct for P2 as part of the implementation of the `kill` system call?

When the kernel executes the kill system call on behalf of P1, it marks the process P2 as "ready to exit". Whn P2 starts running the next time after a context switch, it would exit (recall, processes move to the exit state only from the running state).
The OS runs on a per-thread stack associated with the currently running thread. Destroying the thread structure destroys the per-thread stack as well. So the OS would crash if the P2 stack was destroyed when the OS was running in the context of the P2 thread.

(b) Briefly explain how P2 is actually terminated when P1 calls `kill`, and why this solves the problem identified in (a). A high-level description is fine – you do not need to refer to specific parts of OS/161 code.

When P2 starts running after a context switch, it will call exit. This removes P2 from the run queue so it is not run any further, but its thread structure and stack are not destroyed yet. Instead the P2 thread is marked a "zombie", i.e., dead but not destroyed thread. When the OS performs a context switch to start running another thread (and thus starts using the other thread's stack), it destroys any zombie threads. This is safe because the OS is not running in the context of any zombie thread.

Question 4. OS161 Process Management [10 MARKS]

Part (a) [4 MARKS] Process management systems calls in Unix-based systems include fork(), exec(), waitpid() and exit(). A student has written code that uses these system calls to launch a child process, have the child execute a program named "hello" (with no arguments), and have the parent wait for the child to complete.

Is the code shown below correct? If not, explain the problem on the right side in one short sentence. Then fix the code by crossing out the lines to be deleted in the code below and then adding new code on the right side.

```
int done = 0;
int rc = fork()

if (rc == 0) { // child
  char *argv[2];
  argv[0] = strdup("hello");
  argv[1] = NULL; // important!
  execv(argv[0], argv);
  done = 1;
} else if (rc > 0) { // parent
  while (done == 0) { // spin
  }
}
```

The variable `done` is not a shared variable and so when the child writes to it, it will not be updated in the parent. Remove the `while (done == 0)` loop, and replace by `waitpid(rc)`.

Part (b) [4 MARKS] Describe how or where an operating system such as OS161 gets the following information about a process.

(a) Where to start executing a program.

The executable stores the start address of a program.

(b) The first available heap memory address.

The executable stores the end address of the data region. The heap starts at a page boundary after the data region.

(c) The initial value for the stack pointer for the process.

The stack top is set to 0x8000000.

(d) The arguments passed to a program.

The `execv()` system call passes the arguments to the program.

Part (c) [2 MARKS] Process A has issued a system call and is running in kernel mode. Process B is in BLOCKED state. Give one reason why the kernel running in the context of Process A cannot "kill" Process B by invoking thread_exit() and thread_destroy() on Process B.

Process B could have allocated a resource and is expecting another thread/interrupt handler to update the resource before unblocking this thread. If Process B was killed and the resource deallocated, then the other thread may overwrite this resource. For example, if a thread is blocked on a disk read, and it is killed, and the page deallocated and later reallocated to another thread, then the disk interrupt handler would write to this reallocated page (which the other thread is not expecting). Hence, a process exits only while it is running.

Question 2. System Calls [12 MARKS]

For each program shown below, we have listed five possible outputs. Circle all the outputs (zero, one or more) that may be produced by the program. For each wrong answer, 1/2 mark will be deducted. These programs are called `whatA`, `whatB`, `whatC` and `whatD`. Assume that all system calls execute successfully.

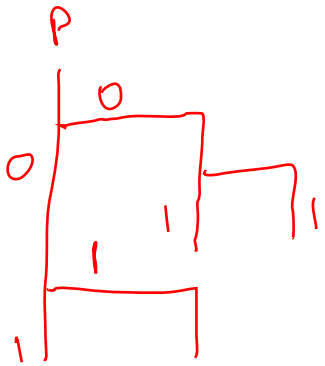
Part (a) [3 MARKS] whatA

```
int main() {
        int i = 0;
        char *argv[2] = { "./whatA", NULL };

        printf("%d ", i);
        i = i + 1;
        if (i >= 2)
                exit(0);
        execv(argv[0], argv);
}
```

A: 0 1 2
B: 0 0 0
C: 0 0 0 0 0 0 ... (forever)
D: 0 1 2 0 1 2 ... (forever)
E: none of the above

Part (b) [3 MARKS] whatB

```
int main() {
        int i = 0;

        while (i < 2) {
                fork();
                printf("%d ", i);
                i = i + 1;
        }
}
```



A: 0 1 0 1
B: 0 0 1 1 1 1
C: 0 1 0 1 1 1
D: 0 1 1 0 1 1
E: 0 1 1 1 0 1

Part (c) [3 MARKS] whatC

```
int main() {
        int i = 0;

        while (i < 2) {
                int pid = fork();
                if (pid)
                        waitpid(pid, NULL, 0);
                printf("%d ", i);
                i = i + 1;
        }
}
```

A: 0 1 0 1
B: 0 0 1 1 1 1
C: 0 1 0 1 1 1
(D): 0 1 1 0 1 1
E: 0 1 1 1 0 1

Part (d) [3 MARKS] whatD

```
int i = 0;

void sig(int num) {
        printf("-1 ");
        exit(0);
}

int main() {
        int pid;

        signal(SIGUSR1, sig);
        pid = fork();
        if (pid) {
                i = i + 1;
                kill(pid, SIGUSR1);
                waitpid(pid, NULL, 0);
        }
        printf("%d ", i);
}
```

(A): -1 1
(B): 0 1
C: 1 0
D: -1 0 1
(E): 0 -1 1
