**3.8 (15 points)** Describe the difference between short-term, medium-term, and long-term scheduling.

---

**short-term:** Selects a process which is in the ready state and allocates it to a CPU to actually begin execution (running). Thus, a short-term scheduler is the fastest form of CPU scheduler, as it must make the decision on which process to execute next.

**medium-term:** Decides which processes to remove from memory. Can suspend processes by moving them to secondary storage (e.g. hard drive) rather than taking up space on the system RAM. For example if a process is waiting for a I/O to complete, it may be moved to swap space by the medium-term scheduler to free up system resources for other programs.

**long-term:** Determines which programs are to actually be processed by admitting processes (new→ready) from the queue and loading them into memory.

**3.9 (10 points)** Describe the actions taken by a kernel to context-switch between processes.

---

The system will receive a clock interrupt, letting it know that it is time to perform a context switch, all registers (including the program counter, stack pointer, etc) are saved to the Process Control Block. Once the state of the program has been saved, the OS calls the scheduler to determine the next process to be executed. The OS then fetches the PCB of that process, and loads all registers to restore the state that that process left off. The CPU can then continue execution of this new process in user mode.

**3.12 (25 points)** Including the initial process, how many processes are created by the program shown below.

```c
#include <stdio.h>
#include <unistd.h>

int main() {
        for (int i=0; i<4; i++) {
                fork();
        }
```

```
        return 0;
}
```

---

16 processes are created by the program.

To see this, note that the parent creates 4 child processes (one at each index of the for loop). Thus, we so far have 5 processes total, including the parent.
Consider the process created at index 0, when this process starts, it will run the remaining 3 iterations of the for loop, and create 3 child processes of its own. The first of those will create 2 (one of which will create another 1), the second will create 1 child process, and the last will terminate the loop when it is created and not make another child. Thus the child at index 0 has 7 children/grandchildren.
The second child (index 1) process will create 2 child process for each iteration, one of which will create 1 more and the last will again terminate the loop, for a total of 3.
The third child (index 2) will create 1 child process, with no children, for a total of 1 children/grandchildren.
The fourth and final child (index 3) immediately terminates the lop, for a total of 0 children/grandchildren.
Thus, 5 + 7 + 3 + 1 + 0 = 16 processes in total.

**3.14 (25 points)** Using the program below, identify the values of pid at lines A, B, C, and D (Assume that the actual pids of the parent and child are 2600 and 2603, respectively).

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main() {
        pid_t pid, pid1;
        pid = fork(); // fork a child process

        if (pid < 0) {
                fprintf(stderr, "Fork Failed!\n");
                return 1;
```

```
        } else if (pid == 0) {
                // child process
                pid1 = getpid();
                printf("child: pid\t= %d\n", pid); /* A */
                printf("child: pid1\t= %d\n", pid1); /* B */
        } else {
                // parent process
                pid1 = getpid();
                printf("parent: pid\t= %d\n", pid); /* C */
                printf("parent: pid1\t= %d\n", pid1); /* D */
                wait(NULL);
        }

        return 0;
}
```

---

**A:** 0 (pid = fork() returns 0 to child process)
**B:** 2603 (pid1 = getpid() returns id of current process, i.e. the child)
**C:** 2603 (pid = fork() returns child pid to the parent)
**D:** 2600 (pid1 = getpid() returns id of current process, i.e. the parent)

**3.17 (25 points)** Using the program shown below, explain what the output will be at lines X and Y.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

#define SIZE 5
int nums[SIZE] = {0, 1, 2, 3, 4};

int main() {
        pid_t pid = fork();

        if (pid == 0) {
```

```
            // child process
            for (int i=0; i<SIZE; i++) {
                    nums[i] *= -i;
                    printf("CHILD: %d ", nums[i]); /* LINE X */
            }
    } else if (pid > 0) {
            // parent process
            wait(NULL);

            for (int i=0; i<SIZE; i++) {
                    printf("PARENT: %d ", nums[i]); /* LINE Y */
            }
    }

    return 0;
}
```

---

**LINE X:** The child process multiplies each value of the array by the opposite of the values index. Thus the child will print out the results of {0*-0, 1*-1, 2*-2, 3*-3, 4*-4}. Which will be the values {0, -1, -4, -9, -16}.

CHILD: 0 CHILD: -1 CHILD: -4 CHILD: -9 CHILD: -16

**LINE Y:** The child modifies the values of the array in it's own memory space (even though it is global, child process still have their own global memory space), not the parents, thus the parent will print out the results the array was initialized to when 'nums' was declared. That is, {0, 1, 2, 3, 4}.

PARENT: 0 PARENT: 1 PARENT: 2 PARENT: 3 PARENT: 4