

# CSIS0230A Principle of Operating Systems (Class A)

## Tutorial 3

### Making use of signals

Recall that in the password cracking program of the last tutorial, we use shared memory to allow two processes to share the same memory, so that one process can communicate to another that the password is found. It has a disadvantage that some CPU cycles must be used to check the shared memory, so the program slows down a bit. During this tutorial, we will repeat the exercise, this time using signals. In order to do so, we revert all changes that makes use of shared memory, resulting in the following *main()* function:

```
int main() {
    int infile, status;
    char *str="Dearest ", encrypted[8];
    char passwd[8] = "abcxy";
    pid_t pid;

    /* Open the file, read first 8 characters */
    infile = open(ENCRYPTED_FILE, O_RDONLY);
    if (infile == -1) {
        fprintf(stderr, "%s: %s\n", strerror(errno), ENCRYPTED_FILE);
        return -1;
    }
    read(infile, encrypted, 8);
    close(infile);

    pid = fork();
    if (pid) {
        /* Try to decrypt for all characters */
        for (passwd[5]=' '; passwd[5]<=(' '~')/2; ++passwd[5])
            for (passwd[6]=' '; passwd[6]<='~'; ++passwd[6])
                for (passwd[7]=' '; passwd[7]<='~'; ++passwd[7])
                    if (rightPassword(str, passwd, encrypted)) {
                        printf("The password is %.8s\n", passwd);
                        return 0;
                    }
        waitpid(pid, &status, 0);
        if (WIFEXITED(status) && WEXITSTATUS(status)==0)
            return 0;
        printf("Password not found.\n");
    }
    else {
        /* Try to decrypt for all characters */
        for (passwd[5] = (' '~')/2+1; passwd[5]<='~'; ++passwd[5])
            for (passwd[6]=' '; passwd[6]<='~'; ++passwd[6])
                for (passwd[7]=' '; passwd[7]<='~'; ++passwd[7])
                    if (rightPassword(str, passwd, encrypted)) {
                        printf("The password is %.8s\n", passwd);
                        return 0;
                    }
    }
    return 1;
}
```

### Steps:

1. Add code to send the other process the *TERM* signal whenever the password is found.
2. We have prepared three `enMail.txt` files (called `enMail1.txt`, `enMail2.txt` and `enMail3.txt`). For the first mail the password will be found in the parent, for the second it will be found in the child, and for the last the password won't be found at all. Copy each of them to `enMail.txt` and see the resulting behaviour. In particular, note whether `echo $?` will show you the correct exit code of the program.
3. You will find that for the second case, the program exits in a strange way, and the exit code is incorrect. By trying different signal numbers, determine what is meant by the exit code and comparing them to the signal numbers you found in the `signal(7)` manual page.
4. Modify the program to restore the behaviour that the exit code is 0 whenever the program (either parent or child) can find the password.
5. Suggest why it is not in general a good idea to use the `TERM` signal for the purpose, and suggest a better signal number to use. Modify the program accordingly.
6. Now that the program works correctly, let's explore more about other signals. Run your program, and before it finds the password, press `Control-\` to send the `QUIT` signal. By modifying the program, show that both the parent and the child are terminated.
7. Check the directory to check that a `core` file is not generated. Read the man page of `ulimit(1)` and determine the reason.
8. Fix the problem, so that core dumps are created. By using the debugger (call the debugger like `gdb <prog-name> <corefile>`, e.g., `gdb a.out core.12345`, to examine the core file), find what is the last password tried by the program.

### Hints:

- The parent process ID can be found using the `getppid()` system call.
- You should use the `-g` flag (along with the `-lcrypt` flag) when compiling the program to keep information about the names of variables in the executable file, so the the debugger can show symbolic names rather than addresses.
- The following `gdb` commands will be useful: `bt` (show the call stack), `up` (go to the caller), `down` (reverse of `up`) and `p` (print the value of an expression, which may contain variables in the current function).