Operating Systems Lab - Week 4: exercise - with answers

This lab is about low-level input-output and processes. You will start to see why studying C is important from an OS perspective. You will practice with IO process control facilities. In particular, this exercise asks you to write

- a program that reads and writes files using `stdio.h` functions for *formatted IO* and
- a program that creates a given number of child processes using `fork`.

Try to reproduce the formatted output shown in the examples *exactly*, e.g. pay attention to all capitalization details and empty spaces.

1 Input-output

Write a program, called `inputOutput.c`, that

- takes two file names, e.g. `fileIn.txt` and `fileOut.txt`, as *command-line* arguments,
- copies what the user writes on the terminal after the program has started into the first file, `fileIn.txt`, and
- makes a *capitalised version* of the text saved in `fileIn.txt` into the second file, `fileOut.txt`.

1.1 Command line inputs

To make your program accept and parse command line arguments, you need the formalism mentioned in Week 3 lab introduction. You can find an example of how an input-dependent main in the last section of Week 3's lab exercise. Try to understand what the following program does

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  if (argc < 2) return -1;
  char *fileNameIn = argv[1];
  FILE *fileHandleIn = fopen(fileNameIn, "w");
  printf("fdIn=%d\n", fileno(fileHandleIn));
  printf("sizeof(fileHandleIn)=%lu\n", sizeof(*fileHandleIn));
  return 0;
}
```

How do you run the corresponding executable? Try different file names to see if the output is affected and if the file identifier, `fileno(fileHandleIn)`, changes over different runs.

Answer: The output is not affected because the name is stored in the struct as a string and referred to through a pointer to its first character. `fdIn` does not change over different runs. Make the program accept two files instead of one and print their identifiers and the size of their file handles. Why can't you use `printf` to print `fileHanldeIn` directly?

Answer:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  if (argc < 3) return -1;
  char *fileNameIn = argv[1];
  char *fileNameOut = argv[2];
  FILE *fileHandleIn = fopen(fileNameIn, "w");
  FILE *fileHandleOut = fopen(fileNameOut, "w");
  printf("fdIn=%d\n", fileno(fileHandleIn));
```

```
  printf("sizeof(fileHandleIn)=%lu\n", sizeof(*fileHandleIn));
  printf("fdOut=%d\n", fileno(fileHandleOut));
  printf("sizeof(fileHandleOut)=%lu\n", sizeof(*fileHandleOut));
  fclose(fileHandleIn);
  fclose(fileHandleOut);
  return 0;
}
```

`fileHandleIn` is a structure of type `FILE` and cannot be printed with a single call of `printf`.

1.2 Parse the user input with `scanf`

The following program uses the `stdio.h` function `fscanf` to print a capitalized version of the user input on `stdout`.

```
#include <stdio.h>
int upper(int c) {
  if (c >= 'a' && c<= 'z')
    return c — 'a' + 'A';
  else
    return c;
}
int capitalise(char *q) {
  int c;
  int i = 0;
  while((c = *(q+i)) != '\0') {
    *(q+i) = upper(c);
    i++;
  }
  return i;
}
int main() {
  char s[10];
  while (fscanf(stdin, "%10s", s)==1) {
    capitalise(s);
    fprintf(stdout, "s=%s", s);
  }
}
```

Compile and run the program to understand how `fscanf` works. Note that

- `fscanf` is triggered by both `'\n'` and `' '`,
- to avoid overflow or other memory problems, you need to specify the maximum number of characters to be stored in the buffer through the format specifier `%10s`, and
- `capitalize` returns the length of the string and the changes in `s` are not discarded when it returns.

Look at C online manualto see how to use its return value to exit the `while`-loop.

Answer: `fscanf` returns -1 when it reads a non-valid character, e.g. `EOF`.

1.3 Write the original input on `fileIn.txt`

Use `scanf` and the file handle associated with the first file to write the user input on `fileIn.txt`. Note that `stdin` in the program above is a file handle and, to write on an open file, you can use

```
fprintf(fileHandle, "%s\n", q);
```

where `fileHandle` is a *pointer* to the structure of type `FILE` associated with the open file.

Answer:

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  if (argc < 3) return -1;
  char *fileNameIn = argv[1];
  char *fileNameOut = argv[2];
  FILE *fileHandleIn = fopen(fileNameIn, "w");
  FILE *fileHandleOut = fopen(fileNameOut, "w");
  printf("fdIn=%d\n", fileno(fileHandleIn));
  printf("sizeof(fileHandleIn)=%lu\n", sizeof(*fileHandleIn));
  printf("fdOut=%d\n", fileno(fileHandleOut));
  printf("sizeof(fileHandleOut)=%lu\n", sizeof(*fileHandleOut));
  char q[10];
  while(fscanf(stdin, "%10s", q) == 1)
        fprintf(fileHandleIn, "%s\n", q);
  fclose(fileHandleIn);
  fclose(fileHandleOut);
}
```

1.4 Capitalize and copy the content of `fileIn.txt` into `fileOut.txt`

Complete the following program using the correct file handles. The completed program should behave as described at the beginning of this section.

```c
#include <stdio.h>
int upper(int c) {
  if (c >= 'a' && c<= 'z')
    return c - 'a' + 'A';
  else
    return c;
}
int capitalise(char *q) {
  int c;
  int i = 0;
  while((c = *(q+i)) != '\0') {
    *(q+i) = upper(c);
    i++;
  }
  return i;
}
int main(int argc, char *argv[]) {
  if (argc < 3) return -1;
  char *fileNameIn = argv[1];
  char *fileNameOut = argv[2];
  FILE *fileIn = fopen(..., "w");
  char q[10];
  while(fscanf(..., "%10s", q) == 1) {
        fprintf(..., "%s", q);
  }
  fclose(...);
  ... = fopen(..., "r");
  FILE *fileOut = fopen(..., "w");
  while(fscanf(..., "%10s", q) == 1) {
    capitalise(q);
    fprintf(..., "%s", q);
  }
  fclose(...);
  fclose(...);
  return 0;
}
```

Note that the program does not print anything on screen. A run with the following user input.

```
one
Two

three four Five


and
s
i
x!
```

write

```
oneTwothreefourFiveandsix!
```

on `fileIn.txt` and

```
ONETWOTHREEFOURFIVEANDSIX!
```

on `fileOut.txt`.

Answer:

```c
#include <stdio.h>
int upper(int c) {
  if (c >= 'a' && c<= 'z')
    return c — 'a' + 'A';
  else
    return c;
}
  int capitalise(char *q) {
  int c;
  int i = 0;
  while((c = *(q+i)) != '\0') {
    *(q+i) = upper(c);
    i++;
  }
  return i;
}

int main(int argc, char *argv[]) {
  if (argc < 3) return —1;
  char *fileNameIn = argv[1];
  char *fileNameOut = argv[2];
  FILE *fileHandleIn = fopen(fileNameIn, "w");
  char q[10];
  while(fscanf(stdin, "%10s", q) == 1)
    fprintf(fileHandleIn, "%s", q);
  fclose(fileHandleIn);
  fileHandleIn = fopen(fileNameIn, "r");
  FILE *fileHandleOut = fopen(fileNameOut, "w");
  while(fscanf(fileHandleIn, "%10s", q) == 1) {
    capitalise(q);
    fprintf(fileHandleOut, "%s", q);
  }
  fclose(fileHandleIn);
  fclose(fileHandleOut);
  return 0;
}
```

2 fork

Write a program, `nChildren.c`, where a parent process creates N child processes through `fork`, waits for them to complete a task, and exits. We suggest you use the following *standard library* functions:

1. `int printf(const char *format, ...)` defined in `stdio.h` and described in Section 12.12 of C online manual,
2. `pid_t getpid(void)` defined in `unistd.h` and described in Section 26.3 of C online manual,
3. `pid_t fork(void)` defined in `unistd.h` and described in Section 26.4 of C online manual,
4. `unsigned int sleep(int sec)` defined in `unistd.h` and described in Section 21.7 of C online manual,
5. `pid_t wait(int *status)` defined in `sys/wait.h` and described in Section 26.6 of C online manual, and
6. `int WEXITSTATUS(int status)` defined in `sys/wait.h` and described in Section 26.7 of C online manual.

Do not forget to include the corresponding headers (`stdio.h`, `unistd.h`, `wait.h`), write

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

at the very beginning of your code.

2.1 Command line argument

Again define `main` so that the program accept a single *character digit*, `N` as a command-line parameter, i.e. let `main` be

```
int main(int argc, char **argv) {
  int N = *argv[1] - '0';
  ...
}
```

What is `**argv`, why is `int main(int argc, char **argv)` equivalent to `int main(int argc, char *argv[])`? Why can you use `int N = *argv[1] - '0';` to convert the input into an integer?

Answer: `char **argv` is a pointer to pointer and is equivalent to a *string array* as it is initialized with a list of constant strings (the user input). `*argv[1]` is a `char` containing a numerical character, and can be converted into an integer by subtracting the right offset, i.e. the ASCII code of `'0'`.

2.2 Write a task function

The task of all children consists of

- printing the process identifier on the terminal using `printf` and `getpid` and
- sleeping for $n\%(N-1)$ seconds using `sleep`.

Your function should not return any value, i.e. you should declare it as `void`, and accept two parameters, the sleeping time and the process *label*. You can use the following structure

```
void sleepingFunction(int sec, int j) {
  printf("%dth child (pid=%d) sleeps for %d sec\n", ..., ..., ...);
  sleep(...);
}
```

Answer:

```
void sleepingFunction(int sec, int j) {
  printf("%dth child (pid=%d) sleeps for %d sec\n", j, getpid(), sec);
  sleep(sec);
}
```

2.3 Generate N children with fork

You can generate a given number of child processes with a loop. Add a return statement just after the children have performed their task to avoid an uncontrolled generation of child-of-child processes. Also, make your program print a message on the terminal when one of the children terminates using `printf`and the child's label $j = 1, \ldots, N$. For example, you can complete and add to your main the following lines

```
pid_t pid;
for (int j = 0; j < N; j++) {
  if ((pid = fork()) ...) {
    sleepingFunction(... , ...);
    printf("%dth child exits \n", ...);
    return j + 1;
  }
}
```

where N is the integer that you get from `argv`.

Answer:

```
int main(int argc, char **argv) {
  int N = *argv[1] - '0';
  int K = 3;
  pid_t pid;
  for (int j = 0; j < N; j++) {
    if ((pid = fork())== 0) {
      //printf("%d", N/(2 * (j + 1)));
      sleepingFunction(N * (j % K), j + 1);
      printf("%dth child exits \n", j + 1);
      return j + 1;
    }
  }
```

2.4 Child-parent inter-process communication

Before exiting, the parent prints on the screen the order in which the children have terminated. The parents should also wait for all N children to terminate, which can be done by calling `wait` N times. Note that`pid_t wait(int *status)` returns the process identifier of the child that terminates and writes the return value of the child that terminates at the address passes as `status` parameter. To interpret the content of that address, you can call `WEXITSTATUS`, with the value stored at that address as an input. For example,

```
int status;
pidChild = wait(&status);
pidReturnValue = WEXITSTATUS(status);
```

where `pidReturnValue` is what we have called the *child label* above.

2.5 Print the order of arrival

Finally, the parent should print the order of the reaped children at the very end. To do this, save the return values of `wait` and `WEXITSTATUS` into two integer arrays and print their content just before the parent process terminates using

```
for (int k = 0; k < N; k++)
  printf("%dth child(pid=%d) exited %dth \n", orderVector[k], pidVector[k], k + 1);
```

Answer: Here is a possible version of the final program,

```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
void sleepingFunction(int sec, int j) {
  printf("%dth child (pid=%d) sleeps for %d sec\n", j, getpid(), sec);
  sleep(sec);
}
int main(int argc, char **argv) {
  int N = *argv[1] - '0';
  int K = 3;
  pid_t pid;
  for (int j = 0; j < N; j++) {
    if ((pid = fork())== 0) {
      //printf("%d", N/(2 * (j + 1)));
      sleepingFunction(N * (j % K), j + 1);
      printf("%dth child exits \n", j + 1);
      return j + 1;
    }
  }
  int status;
  int pidVector[N];
  int orderVector[N];
  for (int k = 0; k < N; k++) {
    pidVector[k] = wait(&status);
    orderVector[k] = WEXITSTATUS(status);
  }
  for (int k = 0; k < N; k++)
    printf("%dth child(pid=%d) exited %dth \n", orderVector[k], pidVector[k], k + 1);
}
```

Example If $N = 3$ a run of the program should produce an output analogous to[1]

```
cim-ts-node-01$ ./a.out 3
1th child (pid=3691411) sleeps for 0 seconds
2th child (pid=3691412) sleeps for 1 seconds
1th child exits
3th child (pid=3691413) sleeps for 0 seconds
3th child exits
2th child exits
1th child(pid=3691411) exited 1th
3th child(pid=3691413) exited 2th
2th child(pid=3691412) exited 3th
```

[1] Of course, you should expect different values for the process identifiers.