AMERICAN UNIVERSITY OF BEIRUT
Faculty of Arts and Sciences
Department of Computer Science
CMPS 240 – Operating Systems
Spring 2025 – 2026

Assignment 1: Process Creation and Basic Inter-Process Coordination

This assignment aims to strengthen your understanding of process creation, parallel execution, and basic coordination between parent and child processes using the C programming language on a UNIX/Linux system. You will make extensive use of system calls such as `fork()`, `wait()`, `waitpid()`, `stat()`, and standard file I/O.

Unless stated otherwise, all programs must:
- Compile using gcc without warnings.
- Follow good programming practices (clear structure, meaningful variable names, comments).
- Be tested on a Linux-based system.

Exercise 1: Directory File Size Analyzer

You are asked to develop a C program called `filesize_analyzer.c` that analyzes the total size of files contained within a given directory by exploiting parallelism through multiple processes. The program takes as input a directory path provided as a command-line argument. The parent process scans the directory and collects the list of regular files it contains (you may use `opendir()` and `readdir()` functions for this purpose). The files are then divided as evenly as possible into three groups. For each group, the parent creates a child process responsible for computing the total size of the files assigned to it. Each child process computes the cumulative size (in bytes) of its assigned files using the `stat()` system call and writes its result to a dedicated output file (*i.e.*, `group1.txt`, `group2.txt`, or `group3.txt`). Once all children terminate, the parent process reads these files, computes global statistics, and displays a comprehensive report.

Your program must:
1. Accept a directory path as a command-line argument.
2. List all regular files in the directory.
3. Divide the files into three groups based on file count.
4. Create three child processes using `fork()`.
5. Wait for all children to complete execution.
6. Read and aggregate results produced by the children.
7. Display the total directory size and identify the group with the largest size.

For all purposes of modularity and organization, you may want to consider implementing the following functions:

```
void listFiles(char *dirPath, char ***files, int *count);
```

This function is responsible for scanning a directory and collecting the names of all regular files it contains. Given a directory path, the function opens the directory using `opendir()` and iterates through its entries using `readdir()`. It identifies regular files and dynamically stores their names in a list. The function allocates memory as needed to store the filenames and updates the total number of files found. Upon return `files` points to a dynamically allocated array of strings, where each string is a filename, and, `count` contains the total number of files discovered in the directory. The function does **not** fork any processes and is executed by the parent process only.

```
void assignFileGroups(char **files, int count, char ***group1, char ***group2,
                      char ***group3);
```

This function divides the list of filenames into three groups of approximately equal size. Given an array of filenames and the total file count, the function partitions the files sequentially into three groups. Each group is stored in a dynamically allocated array of strings. If the number of files is not divisible by three, the remaining files should be distributed as evenly as possible among the groups. Upon return, `group1`, `group2`, and group3 each point to an array of filenames assigned to that group. This function performs no file system access and is executed by the parent process before forking child processes.

```
long getFileSize(char *filename);
```

This function returns the size of a file in bytes. Given a filename, the function uses the `stat()` system call to retrieve file metadata and extracts the file size. If the file cannot be accessed or an error occurs, the function should handle the error appropriately (*e.g.*, print an error message and return `-1`). This function is intended to be used by child processes while computing the total size of their assigned files.

```
void processFileGroup(char **files, int count, char *outputFile, int groupNum);
```

This function is executed by a child process to analyze a subset of files assigned to it. Given a list of filenames and the number of files in the group, the function iterates over the files, retrieves the size of each file using `getFileSize()`, and computes the cumulative size in bytes. The final result is written to the specified output file (*e.g.*, `group1.txt`, `group2.txt`, or `group3.txt`). The function also prints status messages indicating: *i*) The group number being processed, *ii*) the PID of the child process, and, *iii*) the total size computed. After writing the result to the output file, the function terminates the child process using `exit(0)` upon successful completion.

SAMPLE EXECUTION

```
$./filesize_analyzer /home/user/documents
Parent Process (PID: 8000): Analyzing directory /home/user/documents
Total files found: 15

[Parent] Dividing files into 3 groups...
Group 1: 5 files
Group 2: 5 files
Group 3: 5 files

[Child 1, PID: 8001] Processing group 1...
[Child 2, PID: 8002] Processing group 2...
[Child 3, PID: 8003] Processing group 3...
[Child 1, PID: 8001] Total size: 2,458,624 bytes. Written to group1.txt
[Child 2, PID: 8002] Total size: 1,893,440 bytes. Written to group2.txt
[Child 3, PID: 8003] Total size: 3,147,890 bytes. Written to group3.txt
[Parent] All children completed. Reading results...

---- ANALYSIS REPORT ----
Group 1: 2,458,624 bytes (2.34 MB)
Group 2: 1,893,440 bytes (1.81 MB)
Group 3: 3,147,890 bytes (3.00 MB)
-----------------------------------------
Total Size: 7,499,954 bytes (7.15 MB)
Largest Group: Group 3 (3.00 MB)
[Parent] Analysis completed successfully.
```

Exercise 2: Parallel Keyword Search

In this exercise, you will implement a parallel keyword search utility using multiple processes. The program prompts the user to enter a keyword and a set of text files to be searched. For each file, the parent process spawns a child process responsible for searching for the keyword within that file. Each child scans its assigned file line by line, counts the number of occurrences of the keyword, and reports the result. Each child process exits with status code `0` if the keyword is found at least once, or status code `1` if the keyword does not appear in the file. The parent process waits for all children, collects their exit statuses, and generates a summary report.

Your program must:

1. Prompt the user for a keyword.
2. Accept a list of at least three text files.
3. Create one child process per file.
4. Use `waitpid()` to synchronize with all children.
5. Determine which files contain the keyword.
6. Display the total number of occurrences across all files.

For all purposes of modularity and organization, you may want to consider implementing the following functions:

`int countKeywordOccurrences(char *filename, char *keyword);`

This function counts how many times a given keyword appears in a text file. Given the name of a text file and a keyword, the function opens the file in read mode and scans its contents line by line. It searches for **all occurrences** of the keyword within the file and maintains a running count. Overlapping occurrences do not need to be considered unless explicitly stated. If the file cannot be opened or an error occurs during reading, the function should handle the error appropriately and return `-1`. Upon successful completion, the function returns the total number of occurrences of the keyword found in the file. This function creates no processes and is typically invoked by a child process.

`void searchFile(char *filename, char *keyword);`

This function is executed by a child process to search for a keyword in a single file. Given a filename and a keyword, the function calls `countKeywordOccurrences()` to determine how many times the keyword appears in the file. It then prints a status message that includes: *i*) the PID of the child process, *ii*) the filename being searched, and, *iii*) the number of occurrences found. Based on the result, if one or more occurrences are found, the child process terminates using `exit(0)`. Alternatively, if no occurrences are found, the child process terminates using `exit(1)`. Finally, if an error occurs, the child may terminate using a non-zero exit status different from `1`. This function does not return to the caller; it ends by terminating the child process.

`int waitForChildren(pid_t *pids, int count, char **filenames);`

This function is executed by the parent process to synchronize with all child processes and collect their results. Given an array of child process IDs and the total number of children created, the function waits for each child process to terminate using `waitpid()`. For each terminated child, it retrieves the exit status and determines whether the keyword was found in the corresponding file. The function prints a summary indicating, for each file; that is, whether the keyword was found, as well as the exit status returned by the child process. Additionally, the function computes and returns the number of files in which the keyword was found (i.e., the number of children that exited with status `0`). This function does not create new processes and completes only after all child processes have terminated.

SAMPLE EXECUTION

```
---- Parallel Keyword Search ----
Parent Process PID: 9500

Enter keyword to search: "algorithm"
Enter number of files to search: 4

Files to search:
1. document1.txt
2. notes.txt
3. report.txt
4. summary.txt

[Parent] Creating 4 child processes...

[Child PID: 9501] Searching in document1.txt...
[Child PID: 9502] Searching in notes.txt...
[Child PID: 9503] Searching in report.txt...
[Child PID: 9504] Searching in summary.txt...
[Child PID: 9501] Found 7 occurrences in document1.txt
[Child PID: 9503] Found 12 occurrences in report.txt
[Child PID: 9502] Found 0 occurrences in notes.txt
[Child PID: 9504] Found 3 occurrences in summary.txt

[Parent] All children completed. Collecting results...

---- SEARCH RESULTS ----
Keyword: "algorithm"

Files with keyword:
document1.txt - 7 occurrences
report.txt - 12 occurrences
summary.txt - 3 occurrences

Files without keyword:
notes.txt - 0 occurrences
Summary:

Total files searched: 4
Files containing keyword: 3
Total occurrences: 22

[Parent] Search completed successfully.
```

Exercise 3: Parent-Child Calculator

This exercise focuses on parent–child interaction using `fork()` and exit statuses. You will implement an interactive calculator in which the parent process acts as a coordinator. The parent repeatedly displays a menu of operations, reads two integers and the user's choice, then creates a child process to perform the requested operation. The child process computes the result and terminates, returning the result to the parent via its exit status. The parent retrieves the result using `wait()` or `waitpid()` and displays it. The program continues until the user chooses to exit. Supported operations include addition, subtraction, multiplication, integer division, modulo, maximum, and minimum.

Your program must:

1. Repeatedly display an operations menu.
2. Spawn one child process per requested operation.
3. Retrieve computation results from the child's exit status.
4. Properly handle exit conditions and division by zero.

For all purposes of modularity and organization, you may want to consider implementing the following functions:

`void performOperation(int num1, int num2, int operation);`

This function is executed by the **child process** to perform a single arithmetic or comparison operation. Given two integer operands and an operation identifier, the function determines which operation to perform based on the value of `operation` (as selected from the menu). It then computes the corresponding result. Supported operations include: *a*) addition, *b*) subtraction, *c*) multiplication, *d*) integer division, *e*) modulo, *f*) maximum of two numbers, and, *g*) minimum of two numbers. After computing the result, the function terminates the child process using `exit()`, returning the result as the exit status. Since exit statuses are limited to 8 bits, results exceeding 255 must be handled appropriately (*e.g.*, using modulo 256 or another documented encoding strategy). This function does not return to the caller and should not perform any user input or looping.

`void displayMenu();`

This function displays the calculator's operation menu to the user. The function prints a clearly formatted list of available operations along with their corresponding numeric choices. It does not read user input and does not return any value. Input handling is the responsibility of the parent process. This function may be called repeatedly by the parent process within the main execution loop.

`int getResult(int status);`

This function extracts the computation result returned by a child process. Given the status value obtained from `wait()` or `waitpid()`, the function verifies that the child process terminated normally and extracts the exit status using the appropriate macros (*e.g.*, `WIFEXITED()` and `WEXITSTATUS()`). The function returns the decoded result of the operation. If the child did not terminate normally or an error is detected, the function should handle the situation appropriately (e.g., print an error message and return a sentinel value). This function is executed by the parent process and does not create any processes.

SAMPLE EXECUTION

```
---- Parent-Child Calculator ----

Parent PID: 9000

Operations Menu:
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modulo
6. Maximum
7. Minimum
8. Exit

Choose operation (1-8): 1

Enter first number: 15
Enter second number: 7

[Parent] Creating child process for Addition

[Child PID: 9001] Calculating 15 + 7
[Child PID: 9001] Result: 22, exiting...

[Parent] Child 9001 completed.

Result: 15 + 7 = 22

Choose operation (1-8): 8

[Parent] Exiting calculator. Goodbye!

Total calculations performed: 1
```

Exercise 4: Process-Based String Processor

In this exercise, you will design a program that performs multiple string-processing tasks concurrently using several child processes. The parent process reads a string of up to 100 characters from the user and stores it in a shared input file. It then creates five child processes simultaneously. Each child process performs a different string operation and writes its result to a dedicated output file. The parent process waits for all children to complete, then reads and displays all results. The string operations are: *a*) counting vowels and consonants, *b*) converting the string to uppercase, *c*) reversing the string, *d*) counting the number of words, *e*) removing all spaces.

Your program must:

1. Create five child processes concurrently.
2. Use files to share input and collect output.
3. Synchronize using `wait()` or `waitpid()`.
4. Display all results clearly after completion.

For all purposes of modularity and organization, you may want to implement the following functions:

`void countVowelsConsonants(char *str, char *outputFile);`

This function analyzes the input string and counts the number of vowels and consonants. Given a null-terminated string str, the function scans all alphabetic characters. It counts vowels (a, e, i, o, u — case-insensitive) and consonants (all other letters). Non-letter characters (spaces, digits, punctuation) should be ignored. The function writes the results to the file named by `outputFile` (*e.g.*, in a clear format such as two labeled lines: vowels count and consonants count). The function performs no user input and does not print the final report to the screen.

`void convertToUppercase(char *str, char *outputFile);`

This function converts the input string to uppercase. Given `str`, the function produces a new version of the string where all lowercase letters are converted to uppercase (using `toupper()` or equivalent). All other characters (spaces, punctuation, digits) remain unchanged. The resulting uppercase string is written to `outputFile`.

`void reverseString(char *str, char *outputFile);`

This function reverses the input string. Given `str`, the function generates a reversed version of the string (characters in reverse order). The reversal should include spaces and punctuation exactly as they appear in the original string (*i.e.*, it is a character-level reversal, not a word-level reversal). The reversed string is written to `outputFile`.

`void countWords(char *str, char *outputFile);`

This function counts the number of words in the input string. A "word" is defined as a maximal sequence of non-space characters separated by one or more spaces. The function should correctly handle multiple consecutive spaces and leading/trailing spaces. It writes the word count to `outputFile` in a clear, labeled format (*e.g.*, "Number of words: X").

`void removeSpaces(char *str, char *outputFile);`

This function removes all spaces from the input string. Given `str`, the function produces a new string that contains the same characters in the same order **except that all space characters (' ') are removed**. Other whitespace characters may be treated as-is unless otherwise specified. The resulting string (with no spaces) is written to `outputFile`.

`void displayResults();`

This function is executed by the **parent process** after all child processes have terminated. It reads the contents of the output files generated by the children: `child1_output.txt`, `child2_output.txt`, `child3_output.txt`, `child4_output.txt`, `child5_output.txt`. It then prints a clear, well-formatted report to the screen that includes: *i*) he original input string, and, *ii*) the result of each child's operation (properly labeled). This function does not fork processes and does not modify the output files; it only reads and displays results.

SAMPLE EXECUTION

```
---- Process-Based String Processor ----
Parent Process PID: 10000
Enter a string: Hello World from Operating Systems

[Parent] Input saved to input.txt
[Parent] Creating 5 child processes...

[Child 1, PID: 10001] Counting vowels and consonants...
[Child 2, PID: 10002] Converting to uppercase...
[Child 3, PID: 10003] Reversing string...
[Child 4, PID: 10004] Counting words...
[Child 5, PID: 10005] Removing spaces...
[Child 1, PID: 10001] Completed! Result written to child1_output.txt
[Child 3, PID: 10003] Completed! Result written to child3_output.txt
[Child 2, PID: 10002] Completed! Result written to child2_output.txt
[Child 5, PID: 10005] Completed! Result written to child5_output.txt
[Child 4, PID: 10004] Completed! Result written to child4_output.txt

[Parent] All children completed. Reading results...

---- RESULTS ----
Original String:
"Hello World from Operating Systems"

Child 1 - Vowels & Consonants:
Vowels: 11
Consonants: 18

Child 2 - Uppercase:
"HELLO WORLD FROM OPERATING SYSTEMS"

Child 3 - Reversed:
"smetsyS gnitarepO morf dlroW olleH"

Child 4 - Word Count:
Number of words: 5

Child 5 - No Spaces:
"HelloWorldfromOperatingSystems"

[Parent] Process completed successfully.
Total children spawned: 5
```

Exercise 5: Solving a Maze

Solve the following maze problem by using forks at every location in the 8x8 matrix where there are different directions that you need to explore and print visited locations in the 8x8 matrix and the PID of the process that visited that location.

```
  0 1 2 3 4 5 6 7
 0# # # # # # # #
 1# S . . # . . #
 2# # # . # . # #
 3# . . . . . # #
 4# . # # # . # #
 5# . . . # . . #
 6# # # . . . . #
 7# # # # E # # #
```

The diagram above shows a maze, where `'S'` denotes Start location, `'#'` denotes walls, dots `'.'` represent open paths, and the `'E'` at the bottom is the Exit. The numbers denote the `(x,y)` coordinates of the maze, with the Start at coordinates `(1,1)` and Exit at coordinates `(4,7)`.

Solving a maze is simply checking every possible direction (North, South, East, and West) and fork processes to handle each direction where an open path is present. It is also necessary to keep track of visited positions, to avoid infinite loops. For this problem it is acceptable to assume there are different paths to the maze exit. All you need to find are the exit coordinates.

SAMPLE 8X8 INPUT MATRIX

Legend:

```
S = Start
E = Exit
# = Wall
. = Open path

# # # # # # # #
# S . . # . . #
# # # . # . # #
# . . . . . # #
# . # # # . # #
# . . . # . . #
# # # . . . . #
# # # # E # # #
```

SAMPLE EXECUTION

```
$ ./ maze
PID 7302 visiting (1,1)
PID 7303 visiting (2,1)
PID 7304 visiting (3,1)
PID 7306 visiting (3,2)
PID 7308 visiting (3,3)
PID 7309 visiting (4,3)
PID 7310 visiting (5,3)
PID 7312 visiting (5,4)
PID 7314 visiting (5,5)
PID 7315 visiting (6,5)
PID 7317 visiting (6,6)
PID 7320 visiting (5,6)
PID 7323 visiting (4,6)
PID 7324 visiting (4,7)
PID 7324 FOUND EXIT at (4,7)
```

Use the sample template shown below when developing your solution:

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

#define N 8

typedef struct {
    int x;
    int y;
} Coordinate;

char maze[N][N] = {
    {'#','#','#','#','#','#','#','#'},
    {'#','S','.','.','#','.','.','#'},
    {'#','#','#','.','#','.','#','#'},
    {'#','.','.','.','.','.','#','#'},
    {'#','.','#','#','#','.','#','#'},
    {'#','.','.','.','#','.','.','#'},
    {'#','#','#','.','.','.','.','#'},
    {'#','#','#','#','E','#','#','#'}
};

int maze_debug = 1;
int visited[N][N] = {0};

int search(Coordinate c) {
    ...
    ...
    ...
    return 0;
}

int main(void) {
    Coordinate start = {1, 1};
    search(start);
    return 0;
}
```