Programming HW2 - Hierarchical Service Management System

📅 Deadline:

- Soft Deadline: 11/19 (日) 23:59
- Hard Deadline: 11/26 (日) 23:59

Github Classroom Link: https://classroom.github.com/a/6pUnKDNv (https://classroom.github.com/a/6pUnKDNv)
Discussion Link: https://github.com/NTU-SP/SP2023_HW2_release/issues (https://github.com/NTU-SP/SP2023_HW2_release/issues)
Video Link: See Demo & Illustration Section

0. Change Log

- **10/27**: Output Message Clarification
  - If you accept the homework after 10/27, you can ignore this message.

  Due to our mistakes, the output message of the sample code, spec and sample test cases are not consistent. The problem is fixed now. You should print

  - `[service_name] has received [command]` for **Command Echo Message**
  - `[service] and [number of descendents] child services are killed` for **the output message of kill**

- **10/28**: Update the specification for the `spawn` command
  Now the spec clearly states that you should use `exec()` and `dup2()` to implement the command `spawn` .

- **11/3**: Add a new script in **8. Resources** section

1. Problem Description

If you are not interested in the background stories, go to Implementation & Specs

In this homework, we want you to practice inter-process communication by implementing a simplified hierarchical service management system. Before diving deeper into the homework specs, let's first introduce what service management system is and why it is important.

Intro to Service Management System

Service management, in a broader context, pertains to the orchestration and coordination of various services within a system. It ensures seamless operation, optimal resource allocation, and effective communication among different components, ultimately contributing to the overall efficiency and functionality of a system.

For example, consider a scenario where a service experiences a surge in user traffic. In such cases, the service management system can automatically prompt the service to generate additional child services to effectively handle the increased demand. Conversely, during periods of reduced traffic, you may find it advantageous to optimize costs by removing unnecessary services. In this situation, you would utilize the system to gracefully terminate those services, ensuring resources are allocated efficiently. Additionally, there may be instances where the different services may require the transfer of files. This operation is crucial as it enables seamless collaboration between services, ensuring that each one can access and utilize the necessary files for its specific tasks.

Designing A Local Service Management System

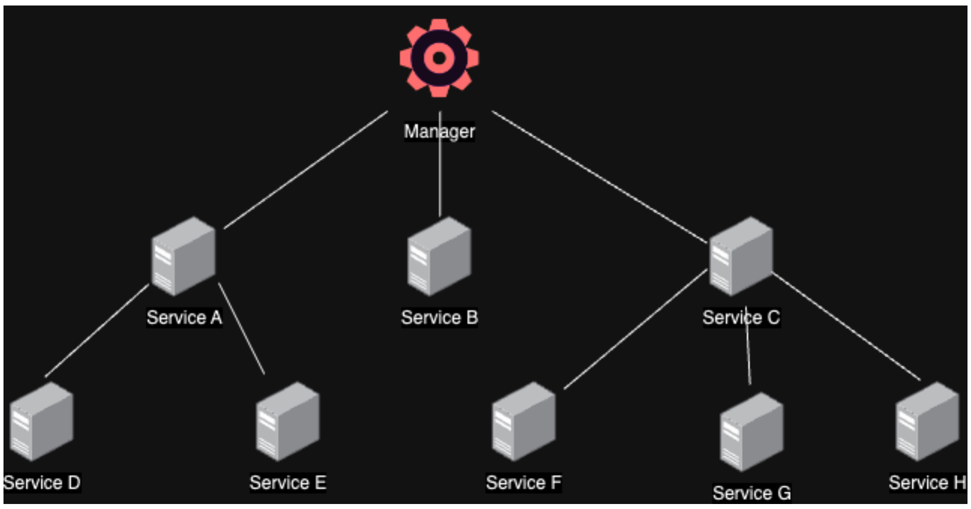
A real-world service management system, often called container orchestration, can be quite complex. It involves many intricate details and considerations. However, for this homework assignment, we're simplifying things. We're focusing on basic functions like adding services, removing services, and transferring authority between them. Also, you won't need to worry about complex network connections. You only need to implement a local service management system where each service operates independently as a process. You'll be using inter-processs communication techniques including forking childs, pipes and FIFOs to make these processes work together smoothly.

In this homework, **you have to finish such a local service management system by writing 1 program (`service.c`).** You will have to follow our test case inputs to establish hierarchical services and **implement features** such as adding services (**spawn**), removing services (**kill**), and exchanging messages (**exchange**) to make sure your service management system works as we expect. Detailed explanations of each important feature will be introduced in the next section.

2. Implementation & Specs

- Please strictly follow the implementation guidelines as follows.
- Some details may be omitted on purpose below. Try your best to ensure your program runs smoothly.

Definition of A Hierarchical Service Management System



(A graphical example of a hierarchical service management system)

The picture above serves as an example of a hierarchical service management system. The arrangement of services may differ based on various test cases, but certain aspects remain constant:

Manager

At the topmost level of the service structure, we can see a service referred to as the Manager. Its primary responsibility is to receive commands (in our case, by reading them from standard input) and, if necessary, pass the commands downwards.

Relationship between Services

As you can see from the picture, this is a hierarchical service structure, indicating the presence of a parent-child relationship between services with a one-level gap. For instance, Service A has two child services, namely Service D and Service E. **Both the parent service and its child services should utilize pipes to facilitate inter-process communication**.

As for the Manager, it should read from standard input as it lacks a parent service above it. Please keep this in mind while designing your program.

Structure Design Overview

We expect you to finish a program named `service.c` . The program receives commands, forks processes and executes `service` to create more processes. **Each service is a process** and each service should communicate to each other with pipe (and FIFO if specified).

The root is named **Manager** and reads from `stdin` . As for other services, they should all read from the pipe connected to their own parent. When passing messages through pipe, I/O multiplexing (`select` , `epoll`) is not needed. You can simply use **blocking I/O** for this homework.

In this homework, you need to implement three commands for the service management system:

- **spawn**: asks a specified parent service to create a child service under it
- **kill**: kill a service and all of its descendents
- **Exchange**: Exchange some secret information between two services through the use of FIFOs

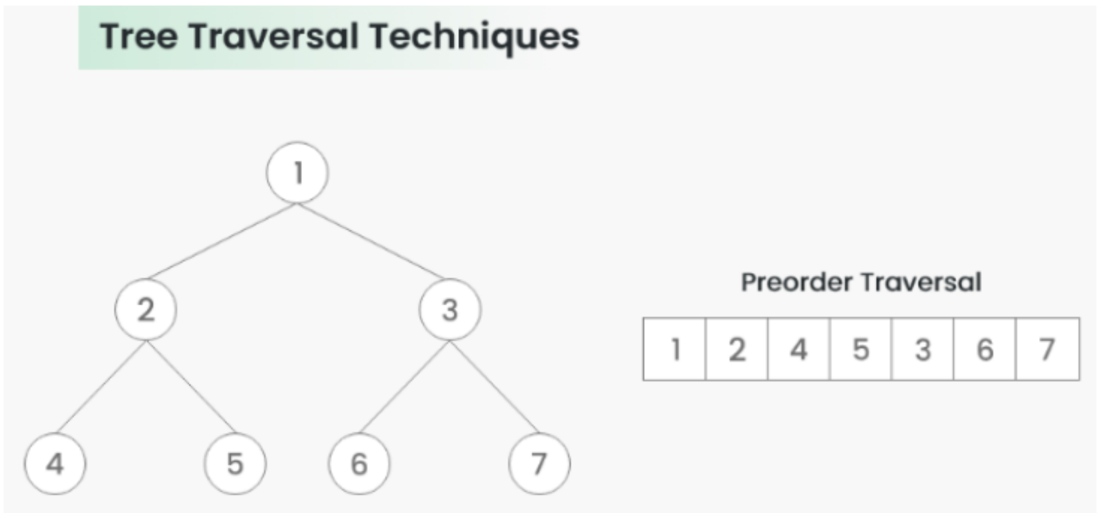
The description above is a very brief introduction to the three commands. Detailed explanations and specifications for the commands are provided in the commands section

Traversal Order for A Hierarchical Service Management System

⚠️ Keep in mind that **a service only has information about its parent and its direct child services**. It doesn't and shouldn't have any other information about other services that don't belong to its parent or child services. Therefore to complete a command, a service may need to **recursively** deliver the message to child services and childs of its child services and so on.

Think of the whole service management system as a tree structure, we regulate that commands must be passed in the same order as specified by the Pre-Order Traversal. Detailed explanations and regulations are given in the following:

- If a service has only one direct child service, when it needs to relay a command downward, it simply passes it to its sole child service.
- If a service has two or more child services, it should relay commands **in the order they were created.** Essentially, the earlier a child service was created, the earlier it should receive commands from its parent.
    - For instance, in the diagram below, Service 1 has two child services, 2 and 3. Assuming Service 2 was created before Service 3, whenever Service 1 attempts to pass a command to its child services, it should first pass it to Service 2, and then to Service 3.
- The sequence of command forwarding should align with the Pre-Order Traversal. You should generate a Command Echo Message each time a service receives a command, allowing us to confirm the traversal order.
    - Detailed information for Command Echo Message can be referred to here.
- You could check sample test cases and demo video () for concrete examples.



Hint: Considering the second point above, because the order of command delivery corresponds to the creation sequence of child services, using a linked list to store information about child services could be a beneficial approach. However, the implementation detail is up to you!

Creating a Service

Each service is created with a name assigned to it. After being created (executed), it randomly generates a secret value (for the use of proceeding `exchange` command). The provided code skeleton has handled the secret value generation part. We have also implement some utility functions for you to output the messages.

- Execution format
  - `./service [service_name]` , where `service` is the binary executable of your `service.c`
  - Example: The root is named `Manager` . Therefore in the very beginning, you should start the service management system program by executing `./service Manager` .
- **Service Creation Message**
  - Whenever a service is being created, that is, executing `./service [service_name]` , it should print out the following message:

    ```
    [service_name] has been spawned, pid: [service_pid], secret: [service_secret
    ```

  - Example: After Manager is created, it may (below values are just for example) print out:

    ```
    Manager has been spawned, pid: 1001, secret: 29103921
    ```

- **Command Echo Message**
  - Whenever a service receives a command from `stdin` or its parent, the first thing is to print out the command echo message with the following message format:

    ```
    [service_name] has received [command]
    ```

  - Example: When service A has received command: `spawn B E` , it should print out the command echo message first and handle the command after:

    ```
    A has received spawn B E
    ```

  - It's important to note that, as demonstrated in the example, even if the command's target isn't service A, upon receiving it, service A must still print out the message. This step is crucial for us to verify that your traversal order aligns with our intended implementation.
- Specification

    - A service name consist of at most **15 letters and numbers (`strlen(service_name) < 16`)**
    - The name of a service is unique
    - The root service is always named `Manager`
    - While judging, every service won't accidentally exit.
    - For precise input range, pleace refer to Genereal Implementation Specifications.
    - You need to return the correct pid for us to verify the process tree structure when judging

Commands of A Hierarchical Service Management System

As mentioned earlier, the Manager service is responsible for reading commands from standard input and passing them downwards (if necessary) to other services to complete tasks. Each command can be considered a function within a hierarchical service management system. In this section, we will provide detailed explanations of the **three commands** we expect you to implement.

General Specifications

- There will only be valid commands, i.e. `spawn, kill, exchange` .
    - Each command ends with a new line `\n` .
    - The number of arguments of each command will be correct.
    - However, **there may be non-existent services in the arguments of a command.** Therefore you should perform some error handling mechanism if needed.
    - You should transmit the command to the children in the order which they were spawned.
    - Once all target(s) has/have received the command, you shouldn't pass the command to the children. In other words, you should stop relaying the command as early as possible. This is called "Early Stop" rule in this homework. You will lose some points if you cannot follow the rule.
        - Example: In sample test case 3 (https://pastebin.com/ACpP5ErW). Only `Manager, A, C, B` have received `exchange A B` .
- Every output message should be printed to **stdout and ends with `\n` .** We will provide detailed explanations and concrete examples to let you know which message to print in the next section.

1. Spawn

The command `spawn [parent_service] [new_child_service]` notifies `[parent_service]` to create a child service (fork a child process) and execute `./service [new_child_service]` with `exec()` .

- Command format
  `spawn [parent_service] [new_child_service]`
- Output Message
    - Because `spawn` creates a service, and whenever there is a service being created, the newly created service should print out the previously mentioned service creation message after created.
    - After `[new_child_service]` is spawned, **Manager** should also print out the following message after the newly created service prints out its **Service Creation Message**.

      ```
      [parent_service] has spawned a new service [new_child_service]
      ```

        - **This message should be printed after the newly created service printing out its creation message**. Therefore you should use pipe to guarantee the order of messages.

    - If `[parent_service]` doesn't exist, **`Manager`** should print the following message **after the child print the Command Echo Message**.

      ```
      [parent_service] doesn't exist
      ```

- Specification
    - **(10/28) update:** After forking the child service, you should use `dup2()` to duplicate `parent_read_fd` to file descriptor 3 and `parent_write_fd` to file descriptor 4, where
        - `parent_read_fd` is the file descriptor through which the child service read the message from parent.
        - `parent_write_fd` is the file descriptor through which the child service write the message to parent.
    - **(10/28) update:** You should call `exec()` to execute `service [new_child_service]` after finishing the duplication of the file descriptors. You lose some points if you skip calling `exec()` .
    - You should properly **close all unused file descriptors**. That is, a service should only have these file descriptors:
        - **0: stdin**
        - **1: stdout**
        - **2: stderr**
        - **3: `parent_read_fd`** , if it is not `Manager`
        - **4: `parent_write_fd`** , if it is not `Manager`
        - A pipe fd for read and a pipe fd for write **for each child**.
    - For the precise input range, pleace refer to Genereal Implementation Specifications.
- Tips
    - `FD_CLOEXEC` flag and `pipe2()` may be useful.
    - Use two pairs of pipe. One for the parent to write, the child to read, the other for the parent to read, the child to write.

2. Kill

The command `kill [service]` notifies `[service]` to **terminate itself and all its descendents.**

- Command format
  `kill [service]`
- Output Message
    - **After `[service]` is terminated, `Manager`** should print the following message:

      ```
      [service] and [number of descendents] child services are killed
      ```

        - `kill Manager` is an exception. You should print `Manager and [number of descendents] child services are killed` **after all child processes of `Manager` have terminiated** and then gracefully terminate `Manager` itself.
        - Print "child service**s are** killed" even the number of descendents is 0 or 1.
    - **If `[service]` doesn't exist, `Manager`** should print the following message.

      ```
      [service] doesn't exist
      ```

- Specification
    - **There should be no zombie process.** That is to say, a parent service should properly call **`wait()`** or **`waitpid()`** for all terminated child services.

        - **Double fork is discouraged.** You'll lose some points if you employ double fork.
    - **Don't use the system call `kill()`.** Instead, pass message to the chlidren via pipe to infer them that they are meant to exit.
    - For the precise input range, pleace refer to Genereal Implementation Specifications.
- Tips
    - Remember to close the pipes of a child.
    - Call `read()` to the pipe before chlidren close the pipe or terminates.

3. Exchange

The command `exchange [service_a] [service_b]` creates FIFOs. Next, `service_a` and `service_b` exchange their secret values via the created FIFOs. The secret values of the two services will swap after executing the command.

- Command format

  ```
  exchange [service_a] [service_b]
  ```

- Output Message
    - After the services get the secret value from each other, they should respectively print the messages with the following orders:
        1. `service_a`

           ```
           [service_a] has acquired a new secret from [service_b], value: [service_b
           ```

        2. `service_b`

           ```
           [service_b] has acquired a new secret from [service_a], value: [service_a
           ```

        3. `Manager`
            - `Manager` should print the following message after `service_a` and `service_b` have exchanged their secret.:

              ```
              [service_a] and [service_b] have exchanged their secrets
              ```

- Specification
    - **`[service_a]` and `[service_b]` are guaranteed to exist and be different.**
    - Create two FIFOs **under the current directory**. The names of the two FIFOs should be
        - `[service_a]_to_[service_b].fifo`
        - `[service_b]_to_[service_a].fifo`
    - You should **close the file descriptors** of the FIFOs and **remove** the FIFOs before `Manager` prints the message `[service_a] and [service_b] have exchanged their secrets`.
    - For the precise input range, pleace refer to Genereal Implementation Specifications.
- Tips
    - Opening a FIFO for read/write will block until another process open it for write/read. While opening FIFOs, be careful of the order.

General Implementation Specifications

Input Range

> The input size won't be large.

- There will be at most `30` services (including Manager) alive simultaneously.
- Each service will have at most `8` child services be alive simultaneously.
- The height of the whole service tree will be at most `10` (starts from Manager).
- The length of a service name `strlen(service_name) < 16` .

Cautions

- You should strictly follow the instructions while implementing the command delivery mechanism. **Don't attempt to bypass the design. As doing so may result in a significant point deduction, and in some cases, a score of 0 if you bypass the command delivery design utilizing pipes**. Below are some examples (not limited to) of bypassing the design.
    - Maintain the whole tree structure in `Manager` and send the commands to the descendents via FIFOs.
    - Record the number of descendents in every service and return the number when executing a `kill` command.
    - Record the name of every service in `Manager` and check whether a service exists.

To be clear, we want you to perform every command and deliver a command between services gracefully and successfully through the use of pipes with Depth First Search and Pre-Order Traversal. That is to say, for each service, please **DO NOT** try to remember any information about other services other than its child services.

3. Judging & Grading

- If your program doesn't block or sleep, there is no need to worry about the time limit.
- We'll give you as many points as possible, even you don't pass all test cases.

General (1.5 points)

- Order of your message delivery mechanism **(0.5 points)**
    - We'll check your standard output after completing every command
- Early stop as soon as all targets have received the command
    - `spawn` and `kill` commands **(0.5 points)**
    - `exchange` command **(0.5 points)**

Spawn (2.5 points)

- Behavior **(2 points)**
    - Fork
        - We'll check the output of `pstree` .
    - Correctly open necessary pipes for parent and child service
- Correctly close unused file descriptors **(0.5 points)**
    - Parent
    - Child

Kill (2 points)

- Behavior **(1 point)**
    - Gracefully terminate the service and all of its descendents
        - We'll check the output of `pstree` and `ps` after running this command
- Correctly release the resources **(0.75 points)**
    - Correctly close unused file descriptors
    - No zombie process after executing this command

- Correctly count the number of killed processes **(0.25 points)**

Exchange (1 point)

- Behavior **(0.75 points)**
    - Sucessfully use FIFOs to swap the secrets
- Correctly close the file descriptors and remove the FIFOs after the services have swapped their secrets. **(0.25 points)**

Report (1 point)

- Briefly describe your approach or design for integrating Pipes (FIFOs) and Pre-Order Traversal in order to achieve the defined command delivery mechanism.
    - You can mention some challenges you've encountered, for example:
        - How do you ensure the order of the output messages?
        - How do you count the number of descendents being killed?
- The report should be in PDF format with no more than 2 pages

4. Submission

Code Submission

You should use GitHub classroom to submit your homework as before. The folder structure on github classroom should at least be:

```
├- sample-testcases
├- service.c
└- Makefile
```

You can submit other `.c`, `.h` files, as long as they can be compiled to one executable named `service` with `make`.

Do not submit files generated by Makefile. You should make clean before you submit. You will lose **0.25 point** as a punishment if you submit those files.

Report Submission

Your report should be in PDF format and you should upload your report to NTU COOL.

5. Demo & Illustration

Sample Test Cases

The following test cases are packed into a folder named `sample-testcases` in the GitHub classroom template repo. What needs to be noticed is that the pid and secret may vary to your answer because these values are indeterministic between each execution.

- Sample-Test-Case-1 (https://pastebin.com/bqQSP0ET). This example contains:
    - basic usage
    - killing descendents
    - service not existing
- Sample-Test-Case-2 (https://pastebin.com/FdWDbfRh). This example contains:
    - exchanging secret between a service and its descendants
- Sample-Test-Case-3 (https://pastebin.com/ACpP5ErW). This example contains:
    - exchanging secret between a service and its sibling

- Sample-Test-Case-4 (https://pastebin.com/1YJ4FS8b). This example contains:
  - demo of traversal order

Task Illustration and Demo

- Video:
  - Command Illustration (https://drive.google.com/file/d/15jiETaDsU-7-6E8Zu-U8rSHV2RlEZAYR/view?usp=share_link)
  - Traversal Order Explanation (https://drive.google.com/file/d/1CAV8mQa-KRTZxT1g4ch7JUREjZ_OoNz4/view?usp=sharing)
- Slides (https://docs.google.com/presentation/d/1oOLgjWjwiOTKbiuljP5bIfSWE2LwUT9hIsrZmn17jTA/edit?usp=sharing)

6. FAQ

> We will update some Frequently Asked Questions from students to this section.

7. Reminders

- Plagiarism is **STRICTLY** prohibited.
  - Both copying and being copied result in a score of zero.
  - Previous or classmates' work will be considered as plagiarism.
  - Discussion is allowed, but the code should be entirely self-written.
  - Avoid letting others view your code. It's your responsibility to protect your work.
  - The method of checking goes beyond comparing source code similarity (e.g., changing `for` to `while` or modifying variable names will also be detected).

- Late policy (D refers to formal deadline, 11/19 23:59)
  - If you submit your assignment on **D+1** or **D+2**, your score will be multiplied by 0.85.
  - If you submit your assignment between **D+3** and **D+5**, your score will be multiplied by 0.7.
  - If you submit your assignment between **D+6** and **11/26**, your score will be multiplied by 0.5.
  - Late submission after **11/26 23:59** will not be accepted.
- If you have any question, we strongly encourage you to:
  - evoke issues in SP2023_HW2_release (https://github.com/NTU-SP/SP2023_HW2_release)
  - discuss on Discord channel
  - ask during TA hours.
- Please start your work as soon as possible, **do NOT** leave it until the last day!

8. Resources

Here are some supplementary resources provided by TA. These resources may contain bugs and TAs won't frequently update these resources. Consider these resources as tools that may be helpful. **Do not argue about your grade due to issues with the resources**.

Output Transformation Script (Maintainer: TA 魏晧融)

script link (https://pastebin.com/7dSzq7rs), **last update: 10/27**

- Usage: `python3 [script_name.py] [output]`

The script transforms lines in `[output]` such as

```
Manager has been spawned, pid: 1286249, secret: 60098046
```

into

```
Manager has been spawned, pid: [Manager-pid], secret: [Manager-secret]
```

and print to stdout to make you check your output more easily.

Zombie Process Removal

You are encouraged to remove zombie processes on the server.

- Run `htop -u $(whoami)` or `ps -ux` to check whether there are zombie processes.
- Run `pkill -9 service` to remove zombie processes effortlessly.

FD Number Check Script (Maintainer: TA 魏皓融)

script link (https://pastebin.com/qEWuTEcL),**last update:11/10, fix some typos, no functionality change**

- Usage: `python3 [script_name.py] [testcase_name] [0/1]`
  - A detailed explanation is in the comment of the script

The script check the number of file descriptors of your `service` . You are expected to finish the functionality of `service` first, otherwise, the script may not work correctly.

In general, the number of the file descriptors is `5 + 2*(num_of_children)` for every child service and `3+2*(num_of_children)` for `Manager` .