


Programming HW2 - PvP Championship

 July 17 Deadline: 11/18 (五) 23:59

⚠ This homework is **much more complicated** than HW1, so TA suggests you to start as soon as possible. 🐱

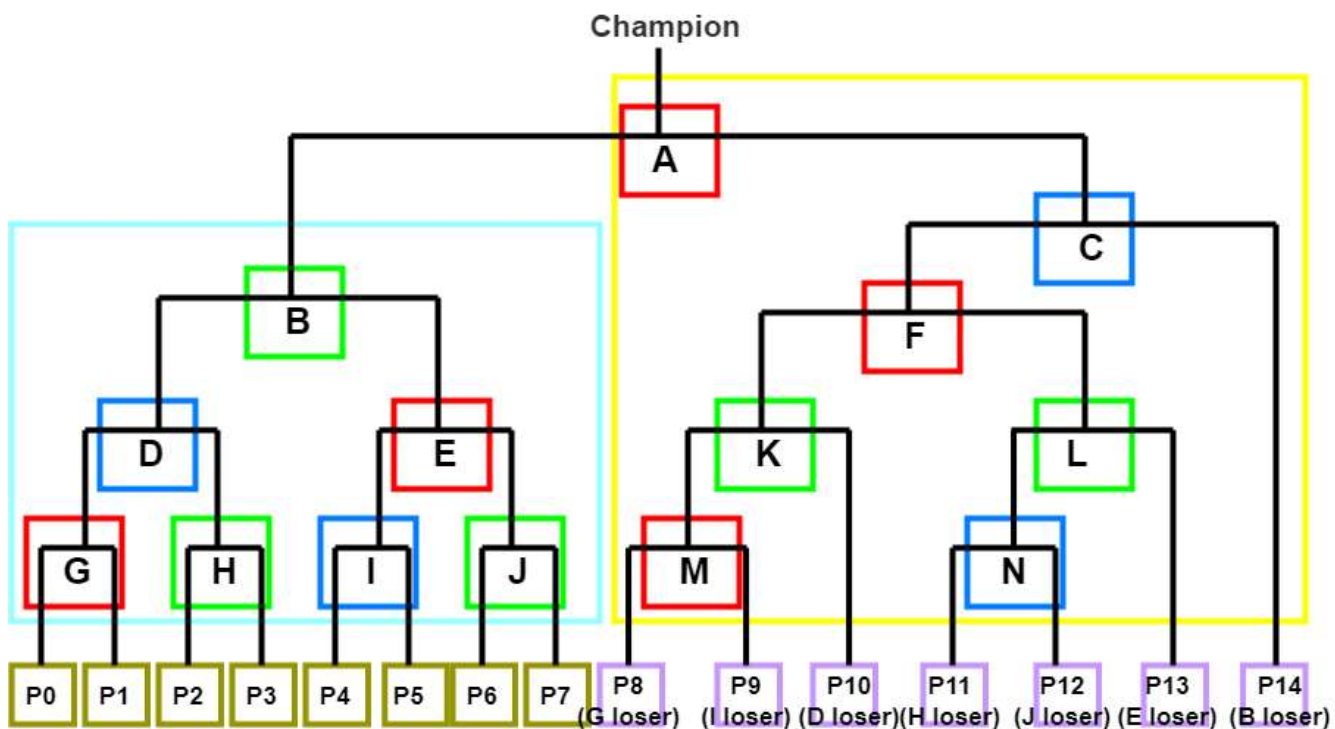
Github Classroom Link (<https://classroom.github.com/a/84niNN9d>).

Video Link (<https://youtu.be/mNVUvgWQezA?t=0>). **(Recommended!)**

Discussion Space Link (https://github.com/NTU-SP/SP_HW2_release/discussions).

Important!!! There are changes in our github repo, please check here (https://github.com/NTU-SP/SP_HW2_release/discussions/6#discussioncomment-4012121), for more information! **[10/30 updated]**

0. Championship Graph 🏆



 Zone A  Real Player  Fire Battle  Grass Battle
 Zone B  Agent Player  Water Battle

1. Problem Description

For you and TA to save our time reading/writing a story for this homework, let's make the long story short. 🤖

There are 8 players (P0 - P7) in this championship. We use a simpler Double Elimination Tournament (https://en.wikipedia.org/wiki/Double-elimination_tournament) for this championship. Each battle has only two players, so there are 14 battles(A - N) in total to generate a champion.

In this homework, **you have to finish such a championship by writing 2 programs(battle.c and player.c)**. You don't have to customize your championship. Just follow the picture above to finish your programs.

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.
- For all battles and players processes, they have to write down logs to their log files when they pass/receive the PSSM to/from parents, children, or siblings.

2.1 battle.c

- The compiled executables must be named `battle`.
- Running the program in Battle Exec Format.
- There will be **14** battles (from A to N) in the game, a battle starts whenever two players both send the first PSSM to the battle, so different battles may run parallelly.
- For non-root battles (B - N), it should read from its parent via **stdin**, and write to its parent via **stdout**.
- For the root battle (A), it should print the Champion Message via **stdout** when the battle ends.
- For battles (A - N), it should communicate with its two children via **pipes**. Two pipes should be created for each child. One for writing to the child. The other for reading from the child. So **a battle will create four pipes in total**.
- **Four modes** in a battle's lifecycle:

- Init Mode
 - In this mode, battle creates its log file, create pipes, fork children..., etc. Then it enters Waiting Mode.
- Waiting Mode
 - After the battle initializes itself, it enters Waiting Mode. It is blocked by the pipes until it receives PSSMs from both children.
- Playing Mode
 - In Playing Mode, the battle receives children's PSSM, updates them, then sends them back to each children repeatedly until one of the player's $HP \leq 0$.
- Passing Mode
 - In Passing Mode, the battle is responsible for passing PSSM between its parent and child. The passing mode ends when all its descendant players' $HP \leq 0$. After that, **the battle should terminate itself immediately.**

- Log File

- Please checkout [here](#) first for general log format.
- Each battle creates a log file named

`log_battle[battle_id].txt`

- `battle_id`: the battle's ID
 - Valid Format: letter A - N (with upper case)
 - Example: `log_battleG.txt`
- There are **two scenarios** that a battle needs to **append** the message to the log file
 1. **Right after** the battle receives PSSM from the target (parent / child).

You should guarantee that the log content is the same as the value in PSSM before/after an IPC.

- Format:

`[self's ID],[self's pid] pipe from [target's ID],[target's pid]
[real_player_id],[HP],[current_battle_id],[battle_ended_flag]\n`

- Example:

`G,1006 pipe from 0,2000 0,3,G,0\n`

2. Right before the battle sends PSSM to the target (parent / child).

- Format:

`[self's ID],[self's pid] pipe to [target's ID],[target's pid]
[real_player_id],[HP],[current_battle_id],[battle_ended_flag]\n`

- Example:

G,1006 pipe to 0,2000 0,3,G,0\n

- Implementation: (in order)

1. Create the log file.
2. Fork and exec two children, then wait for PSSMs from both of them.
3. Receive two childrens' statuses in the form of PSSM.
4. Two players perform a round of the battle.
5. Check if Playing Mode is finished, then send two PSSMs back to each children respectively.
 - If Playing Mode is not finished, repeat the step 3 to 5.
 - Otherwise, after sending the last PSSMs to both children, the battle wait for the loser child's death, then change to Passing Mode.
6. In Passing Mode, the battle still receive winner child's PSSM, but instead of updating the status by itself, it just pass the PSSM to its parent, then wait for the response from it. For everytime the parent pass the PSSM back, the battle should check if the descendant player is dead.
 - If not, passes the PSSM to its child.
 - Otherwise, it pass the last PSSM to its child, wait for its death, then kill itself immediately.

2.2 player.c

- The compiled executables must be named `player` .
- Running the program in Player Exec Format.
- For each player, it should read from its parent via **stdin**, and write to its parent via **stdout**.
- Two types of players
 - Real Player (P0 - P7)
 - It reads initial status from **player_status.txt** and join battles first. When the Real Player lose in a battle, it will transfer its PSSM to the corresponding Agent Player via **fifo**.
 - Agent Player (P8 - P14)
 - It waits for a PSSM via fifo in the beginning. After receiving loser's PSSM from the corresponding battle, the Agent Player then works like a Real Player.
- Log File
 - Please checkout here first for general log format.
 - Each Real Player creates a log file named

`log_player[player_id].txt`

- player_id: Real Player's ID
 - Valid Format: integer, $0 \leq \text{player_id} \leq 7$
 - Example: log_player0.txt
- For Agent players, they inherit the Real Player's log file and continue to append logs to the same log file.
- There are **four scenarios** that a player needs to append the message to the log file:
 1. Right after the player receives PSSM from the target (parent).

- Format:

```
[self's ID],[self's pid] pipe from [target's ID],[parent's pid]
[real_player_id],[HP],[current_battle_id],[battle_ended_flag]\n
```

- Example:

```
0,2000 pipe from G,1006 0,3,G,0\n
```

2. Right before the player sends PSSM to the target (parent).

- Format:

```
[self's ID],[self's pid] pipe to [target's ID],[target's pid]
[real_player_id],[HP],[current_battle_id],[battle_ended_flag]\n
```

- Example:

```
0,2000 pipe to G,1006 0,3,G,0\n
```

3. Right before the **Real Player** sends PSSM to the target (agent player). (**Notice that this log format is slightly different to others**)

- Format:

```
[self's ID],[self's pid] fifo to [target's id] [real_player_id],[HP]\n
```

- Example:

```
0,2000 fifo to 8 0,3\n
```

4. Right after the **Agent Player** receives PSSM from the target (real player). (**Notice that this log format is slightly different to others**)

- Format:

```
[self's ID],[self's pid] fifo from [target's id] [real_player_id],
[HP]\n
```

- Example:

8,2007 fifo from 0 0,3\n

- Implementation: (in order)

1. Check if the player is a Real Player or an Agent Player by args.

- Create a fifo named:

```
player[K].fifo
```

- K: Agent Player's ID

- Valid Format: integer, $8 \leq K \leq 14$

- Example: `player8.fifo`

- Either Real players or Agent players can create the fifo file. For more information, please check [here](https://github.com/NTU-SP/SP_HW2_release/discussions/12) (https://github.com/NTU-SP/SP_HW2_release/discussions/12). **[11/11 updated]**

- If it's **Real Player** (P0 - P7):

- Read the player's initial status from **player_status.txt**. A sample file is [here](#). Then create PSSM from the file.

- If it's **Agent Player** (P8 - P14):

- Block at reading until someone writes a PSSM to the fifo.

2. Send the PSSM to its parent.

3. Receive the updated status from parent. Check if `battle_ended_flag = 1`.

- If not, repeat the step 2 and 3.

- Otherwise,

- If it's a winner:

- Recover itself with **half blood**. (If it's not an integer after recovering, round down it to an integer.)

- Example: P0's **initial HP from** `player_status.txt` is 7. After Battle G, P0's HP = 4, so we recover half of its HP to 5 by the formula: $5(\text{after recover}) = 4(\text{before recover}) + (7(\text{initial HP}) - 4(\text{before recover})) // 2$

- If it's a loser in Zone A:

- Recover itself with **full blood**.

- Example: P1's initial HP from `player_status.txt` is 8. After Battle G, P0's HP = 0, so we recover full of its HP to 8.

- Write the PSSM to the Agent Player corresponds to `current_battle_id` by **fifo**.

- Exit itself.

- If it's a loser in Zone B:

- No more chances for the player.

- Exit itself.

3. Several Definitions

Battle Exec Format

```
./battle [battle_id] [parent_pid]
```

- It requires **two** arguments
 - battle_id: the id of the battle
 - Valid Format: letter A - N (with upper case)
 - parent_pid: the pid of the parent battle, **if there is no parent battle (root battle A), then parent_pid = 0**
 - Valid Format: Non-negative integer

Player Exec Format

```
./player [player_id] [parent_pid]
```

- It requires **two** arguments
 - player_id: the id of the player
 - Valid Format: integer, $0 \leq \text{player_id} \leq 14$
 - parent_pid: the pid of the parent battle.
 - Valid Format: Non-negative integer

Champion Message

```
Champion is P[player_id]\n
```

- player_id: the ID of the player
 - Valid Format: 0 - 7. Note that **if the champion is an Agent Player, it should print the original Real Player's ID** that is inherited by the Agent Player.
 - Example: Champion is P0\n

Player Status Structured Message (PSSM)

A structure to pass between processes. **You cannot modify the structure by yourself**, or TA's program may not work fine with your program.

```

typedef enum {
    FIRE,
    GRASS,
    WATER
} Attribute;

typedef struct {
    int real_player_id; // ID of the Real Player, an integer from 0 - 7
    int HP; // Healthy Point
    int ATK; // Attack Power
    Attribute attr; // Player's attribute
    char current_battle_id; //current battle ID
    int battle_ended_flag; // a flag for validate if the battle is the last round or
not, 1 for yes, 0 for no
} Status;

```

Here we give you some hints about PSSM:

1. You will not modify **real_player_id**, **ATK** and **attr** after the PSSM's creation by real players. You can see them as **const variables**.
2. Only `battles` will modify the value of `current_battle_id`.
3. Only `battles` will modify the value of `battle_ended_flag` from 0 to 1.
4. Only `players` will modify the value of `battle_ended_flag` from 1 to 0.

Log Format

- It's because that we can't know what data you pass between processes when we test your programs, so we need you to **write logs before/after all IPCs**. It's also helpful for you to debug your programs via these log files.
- There will be 8 + 14 log files in the championship. 8 for real players, 14 for battles.
- To sum up log formats for battles and players,

```
[self's info] ["pipe"/"fifo"] ["from"/"to"] [target's info] [PSSM contents]\n
```

- There are **5 main arguments** in the log format. Main arguments are splitted by a space character " ". Each main argument may have several sub-arguments. Sub-arguments are splitted by a comma character ",".
 - self's info: 2 sub-arguments (self's ID, self's PID)
 - self's ID: self's ID in this championship
 - Valid Format: letter A-N (with upper case) or number 0-14
 - self's PID: self's process ID (can use `getpid()` system call to reach it)
 - Valid Format: a positive integer
 - "pipe"/"fifo": communication type, 1 sub-argument
 - "pipe" for communicates between parents or childs
 - "fifo" for communicates between siblings.

- "from"/"to": communication direction, 1 sub-argument
 - "from": A message received from the other process.
 - "to": A message sent to the other process.
- target's info: the other process's info, 1 or 2 sub-arguments (target's ID, target's PID)
 - target's ID: parent/child/sibling's ID in this championship
 - Valid Format: letter A-N (with upper case) or number 0-14
 - target's PID: parent/child's process ID, **you don't need this argument when IPC with siblings, because you have no chance to get its pid.**
 - Valid Format: a positive integer
- PSSM contents: the PSSM you pass in IPC, 2 or 4 sub-arguments (real_player_id, HP, current_battle_id, battle_ended_flag)
 - real_player_id: the value of `real_player_id` in the PSSM
 - Valid Format: number 0-7
 - HP: the value of `HP` in the PSSM
 - Valid Format: integer
 - current_battle_id: the value of `current_battle_id` in the PSSM. **you don't need this argument when IPC with siblings**
 - Valid Format: letter A-N (with upper case)
 - battle_ended_flag: the value of `battle_ended_flag` in the PSSM. **you don't need this argument when IPC with siblings**
 - Valid Format: 0 or 1

Zone A / B

For battles in **Zone A**, players who lose in the battle still have a chance to be the champion, however, players who lose in the battle in **Zone B** are eliminated.

Battle Rules

- Each battle contains N rounds ($N \geq 1$) to generate a winner/loser.
- For each round, two players both attack each other once. So there are **two attacks in one round**.
- The player who has **lower HP can attack first**. If their HP are the same, the player who has lower **Real Player ID** starts first.
- **Once one of the player's $HP \leq 0$, the round is terminated immediately**, and the battle is finished, too. So there is no chance that two players's HP are both ≤ 0 in a round.
- If two players' HP are both still ≥ 0 , the battle has to return PSSMs to players, wait for them to send back, and play the next round until one of the player's $HP \leq 0$.
- Before each round starts, the battle should set two PSSM's `current_battle_id` = the battle's ID.

- If it is the last round in this battle, you should set `battle_ended_flag = 1` in **both** PSSMs to tell two players the battle has been finished. Otherwise, the flag should be 0.
- Check if any player's attribute matches the battle's attribute. **If they are the same, the player's attack power(ATK) is doubled** in this battle.
- One Attack Power(ATK) can reduce the opponent's one Healthy Point(HP).
- Here we give an example:
 - player 0:
 - ATK: 2
 - HP: 7
 - Attribute: FIRE
 - player 1:
 - ATK: 3
 - HP: 8
 - Attribute: GRASS
 - battle G:
 - Attribute: FIRE
 - Flow:
 1. Battle G receives two PSSMs from P0 and P1.
 2. Battle G sets two PSSMs' `current_battle_id = 'G'` . **Round 1 starts.**
 3. $\text{attr}_{P0} = \text{attr}_G$. So ATK_{P0} **is doubled** from 2 to 4.
 4. $\text{HP}_{P0} < \text{HP}_{P1}$, P0 attacks first.
 $4(\text{P1's after HP}) = 8(\text{P1's before HP}) - 4(\text{P0's ATK})$
 5. P1 attacks second. $4(\text{P0's after HP}) = 7(\text{P0's before HP}) - 3(\text{P1's ATK})$
 6. Two players are still alive, so Battle G sets `battle_ended_flag = 0` for both PSSMs, then sends back two updated PSSMs. **Round 1 ends.**
 7. Battle G receives two PSSMs from P0 and P1.
 8. Battle G sets two PSSMs' `current_battle_id = 'G'` . **Round 2 starts.**
 9. $\text{attr}_{P0} = \text{attr}_G$. So ATK_{P0} **is doubled** from 2 to 4.
 10. $\text{HP}_{P0} = \text{HP}_{P1}$, but $0(\text{P0's Real player ID}) < 1(\text{P1's Real player ID})$, so P0 attacks first. $0(\text{P1's after HP}) = 4(\text{P1's before HP}) - 4(\text{P0's ATK})$
 11. $\text{HP}_{P1} \leq 0$, so the battle ends immediately. P0 wins/P1 loses this battle. Battle G sets `battle_ended_flag = 1` for both PSSMs, then sends back two updated PSSMs.
Round 2 ends. The battle ends, too.

4. Sample Execution

Input

player_status.txt

This file would be the same directory with two executables in our test case. There are 8 lines in this file. **The i th line represents player (i-1)'s initial status.** For each line, there are three items, which **represents the player's HP, ATK, Attribute, current_battle_id, battle_ended_flag** respectively (in order). The items are splitted by a space character.

There will be five arguments in a line, no invalid input exists in our testcases.

- HP: Player i's HP.
 - Valid Format: an integer that > 0 and < 65536 **[10/28 updated]**
- ATK: Player i's ATK.
 - Valid Format: an integer that > 0 and < 65536 **[10/28 updated]**
- Attribute: Player i's attribute.
 - Valid Format: a string, "FIRE", "GRASS" or "WATER"
- current_battle_id:
 - Valid Format: will always be player i's parent id.
- battle_ended_flag
 - Valid Format: will always be 0

Below is a sample input:

```
7 2 FIRE G 0\n
8 3 GRASS G 0\n
4 4 FIRE H 0\n
1 4 GRASS H 0\n
12 2 WATER I 0\n
3 4 GRASS I 0\n
7 3 WATER J 0\n
10 2 WATER J 0\n
```

Sample Executables: battle / player

We provide two sample executables named `battle` and `player` for you to test with your program. Note that it does not mean you can get full points by successfully running with these two programs. We will check if you strictly follow the steps by checking the log files step by step.

Output

1. A champion message from battle A's stdout at the end of the championship.
2. 8 + 14 log files.

For PPT sample execution, please refer to the video (<https://youtu.be/mNVUvgWQezA?t=1595>).

Directory Structure

Below is the file structure in the directory. **Comments behind each file name is the answer respected to the questions below:**

1. Should this file be generated by `make` ?
 - Make sure your `Makefile` compile necessary files correctly on CSIE Linux1 workstation, or your score will be **0**.
2. Should this file be generated by your programs?
 - Make sure your program **generate the files in the correct location**, or our judge won't able to find the files, and your score will be **0**.
3. Should this file be removed by `make clean` ?
 - Make sure your `Makefile` can clean unnecessary files with `make clean` , or you will lose **0.25 point**.
4. Should this file be pushed to your github repo?
 - You will lose 1 point if you submit unnecessary files.

```

github_repo/
|
|----sample/      // N/N/N/Y
|   |
|   |--battle     // sample executable given by TAs
|   |
|   |--player     // sample executable given by TAs
|
|----Makefile     // N/N/N/Y
|
|----README.md    // N/N/N/Y
|
|----battle.c     // N/N/N/Y
|
|----player.c     // N/N/N/Y
|
|----status.h     // N/N/N/Y, PSSM structure definition file
|
|----other .c .h files // N/N/N/Y
|
|----player_status.txt // N/N/N/Y, sample input file given by TAs
|
|----battle       // Y/N/Y/N
|
|----player       // Y/N/Y/N
|
|----log files    // N/Y/Y/N
|
|----fifo files   // N/Y/Y/N

```

5. Grading 100

Warning

- Please strictly follow the implementation guidelines, or TA's program may not work successfully with yours and you will lose points.
- Make sure to be aware of your programs when you are writing this homework on CSIE workstations. We encourage you to make attempts; however, if you do the following behaviors, you will lose points in this homework:
 - **Leave too many zombie processes on the workstation.** (OK with first time, -2 points for the rest of each times)
 - Each user in CSIE workstation can create at most 512 processes. If you use all the quota, you may have problems with creating login session in workstation. To solve this problem, you have to contact ta217@csie.ntu.edu.tw. **Please be careful!**
 - **Utilize too many CPU resources by doing a busy waiting task for a long time.**

- Watch out and clean your zombie processes regularly, if you leave them in a while loop for over 24 hours, TA217 / SP TAs will notify you first. If you don't take any actions, you will lose 2 points each time.
- **Do damages maliciously on CSIE workstations.**
 - We will give you a ZERO.
- Here's a useful command to clean all your processes on CSIE workstations (include your terminal):

```
pkill -9 -f [Your_Student_ID]
```

- Your_Student_ID: Your student ID with lowercase.
- Example: `pkill -9 -f r11922002`
- Also, we encourage you to write and debug your homework diversely on linux2 ~ linux15 and only do the final test on linux1, since their environments are almost the same.
- Please strictly follow the output format above, or you will lose the point of each task, respectively.

Items

1. *Your battle works smoothly.*

- (2 points) Battles with only one round work successfully.
 - Single Round: Each battle generates a winner/loser with only one round.
- (2 points) Battles with multiple rounds work successfully.
- We will test if your `battle.c` can successfully communicate with TA's `player.c` and outputs the correct champion and log files.
- *Work successfully* means that
 - all log files are correct
 - champion output is correct
 - battles terminate themselves immediately after Passing Mode
 - work fine with TA's `player.c`.

2. *Your pLayer works smoothly.*

- (0.5 points) Battles in Zone A work successfully.
- (1.5 points) Battles in Zone A, B work successfully.
 - fifo communication is correct.
- We will test if your `player.c` could successfully communicate with TA's `battle.c` and outputs correct log files.
- *Work successfully* means that
 - all log files are correct
 - players terminate themselves immediately after they die.
 - work fine with TA's `battle.c`.

3. (1 points) Your own *battle.c* and *player.c* work together smoothly.

- We will test if your own program can run on its own and outputs the correct champion and log files.

4. (1 points) Report Questions

5. If you don't wait your zombie processes, your maximum score in this homework will be no higher than **6 points**.

6. Report Question

For report questions, please write down your answer and submit the file **in PDF format** to NTUCOOL. The deadline is the same as the programming part.

1.

Please briefly tell us what bugs do you encounter during writing this homework. If no, tell us the most challenging part in this homework. (0.4pt)


2.

In this homework, you have to implement a fifo between two processes. Please tell us how you implement it in your code. Is your implementation good enough to prevent dead lock / race condition / busy waiting? If no, what is the better way to do it? (0.3pt)

3.

In this homework, we use fifo to communicate between real players and agent players. Is it possible to implement the communication with pipe instead of fifo? Why or Why not? Please briefly explain your answer. (0.3pt)

Programming HW2 - PvP Championship

 [HackMD \(https://hackmd.io?utm_source=view-page&utm_medium=logo-nav\)](https://hackmd.io?utm_source=view-page&utm_medium=logo-nav)

7. Submission

7.1 To GitHub

Your source code should be submitted to GitHub before deadline. The submission should include at least three files:

- Makefile
- battle.c
- player.c
- other .c, .h files

The submission should not include files belows:

- log files
- executables generated by your `make` command
- fifo files

You can submit other `.c`, `.h` files, as long as they can be compiled into two executable files named `battle` and `player` with Makefile. TA will clone your repository and test your code with last commit before deadline on main branch, which is the default branch of GitHub.

Furthermore, TAs will test your code on `linux1.csie.ntu.edu.tw`, so make sure your codes will work as expected on this server. If you print some debugging messages in server, it's suggested that you comment these debugging messages out, otherwise it may slow down your code and thus failing to pass testdata.

Last but not least,

- Do not submit unnecessary files. You should make clean before you submit.
- Github classroom seems to have a problem that we can't record your last commit before deadline. Therefore, TA has to clone your github repo manually after the deadline (11/18 23:59). **You should not do any operation on your github repo at (11/19 00:00 - 01:00).** Or we cannot guarantee that your repo is at the right commit.

7.2 To NTU COOL

Your report should be submitted to NTU COOL before deadline.

8. Where to Ask Questions (with recommended order) ?

1. Evoke QAs in Github discussion NTU-SP/SP HW2 release (https://github.com/NTU-SP/SP_HW2_release/discussions/categories/q-a).
 - For this homework, we **change the asking space from Github Issue to Github Discussion**.
 - How to open a Github Discussion (<https://youtu.be/rgCMmg9IBNw?t=210>).
2. Send mails to `ntusp2022@gmail.com`.
 - Before sending mails, please make sure that you have **read the SPEC carefully**, and **no similar questions were asked** in the discussion space, or TAs will be very tired and upset. 😞
 - **How to send an Email properly**
 - Mail title should be like: [SP Programming HW2] TA hour reservation
 - It's better to use your `@ntu.edu.tw`, `@g.ntu.edu.tw`, `@csie.ntu.edu.tw` email accounts to send the mail.

- You should write down your student ID, name at the end of the mail.
- Please mind your manners.
- Use punctuation marks to make your email easier to read.
- If you want to ask questions about code in mail, please tell us how to reproduce your problem. Attach a screenshot, a video recording, or your code file will be better for us to understand your situation.
- Below is an example format:

Title: [SP Programming HW2] TA hour reservation
 Sent By: r11922002@csie.ntu.edu.tw
 Contents:

Dear SP TAs,

Can I have a TA hour appointment at Tuesday 13:00 - 14:00?
 I encounter some problems about fork/exec. Thank you!

Best,
 R11922002 資工碩一 李宥靈

- If you don't follow the upper rules, we may ignore your email. (Most of you guys are good, though. 😊)

3. Ask during TA hours

- Belows are TA hours for this Programming Assignment. Please make appointments by email ntusp2022@gmail.com before visiting.
 - Tuesday 13:00 - 14:00 at Room 404, CSIE building
 - Wednesday 17:30 - 18:30 at Room 302, CSIE building
 - Friday 15:00 - 16:00 at Room 404, CSIE building

9. Reminders !!

- You have to do `fflush()` or `setbuf(NULL)` if you use `printf()` to write to the pipe/fifo.
- Plagiarism is STRICTLY prohibited.
- Late policy (D refers to formal deadline, 11/18)
 - 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 11/30, your score will be multiplied by 0.5.
 - Late submission after 11/30 will not be accepted.
- It's suggested that you should check return value of system calls like `fork`, `execl`, etc.
- Reading manuals of system calls may be tedious, but it's definitely helpful.

- The sample executables are compiled on the CSIE workstation. Therefore, it should only be able to run on Linux Machines. (Note: the sample executables are used to help you and you are not required to use it.)