环境搭建

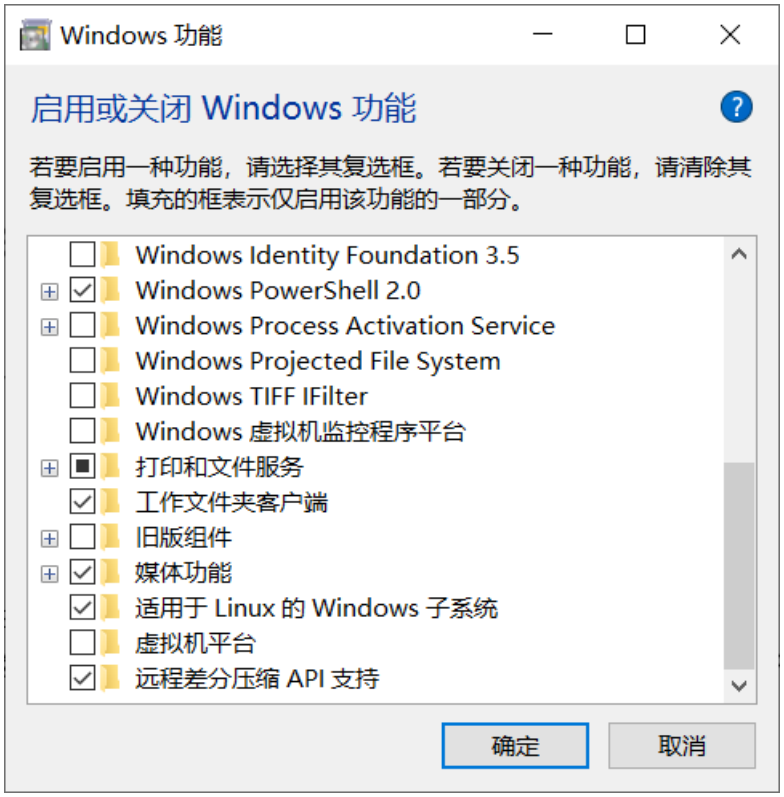
本实验采用 Clion+wsl 的配置。

WSL

wsl，全称 Windows subsystem for linux，有别于虚拟机，它是一种能让 Linux 内核直接运行于 Windows 操作系统的技术，为微软所开发，支持各大常见的 Linux 发行版的内核，但一般不提供桌面环境。

开启

在搜索栏目搜索”启用或关闭 Windows 功能”，勾选“适用于 Linux 的 Windows 子系统”来开启 wsl。



安装

到微软商店搜索 Linux，选择你想要的 Linux 发行版安装，本实验使用的是 Debian。安装过程中会要求设置一些必要的用户信息，按照喜好进行填写即可。

配置 SSH

先得设置 root 密码，因为刚安装的 wsl 的 root 密码是随机的，运行下面的命令来修改：

```
username@localmachine：~$passwd
```

运行下面的命令更新子系统中的 openssh 和安装必要的工具

```
sudo apt update
sudo apt install nano
sudo apt remove openssh-server
sudo apt install openssh-server
sudo apt install net-tools
```

运行下面的命令来编辑 ssh 配置文件

```
sudo nano /etc/ssh/sshd_config
```

去掉注释并修改 PasswordAuthentication 为 yes

```
# To disable tunneled clear text passwords, change to no here!
PasswordAuthentication yes
#PermitEmptyPasswords no
```

安装工具链

运行下面的命令安装必要的工具链条

```
sudo apt install build-essential
sudo apt install cmake
sudo apt install gdb
```

开启 ssh

运行下面的命令来开启 ssh

```
sudo service ssh restart
```

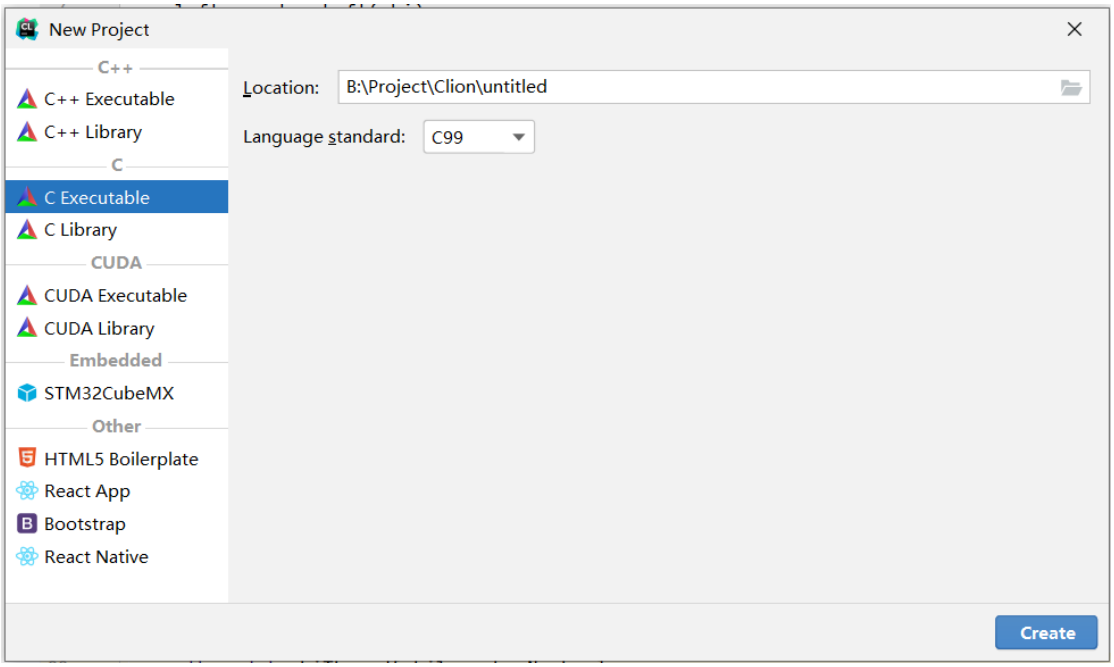
运行下面的命令来查看当前 wsl 的内网 ip：

```
ifconfig
```

几下 wifi0 下的 inet 的值。

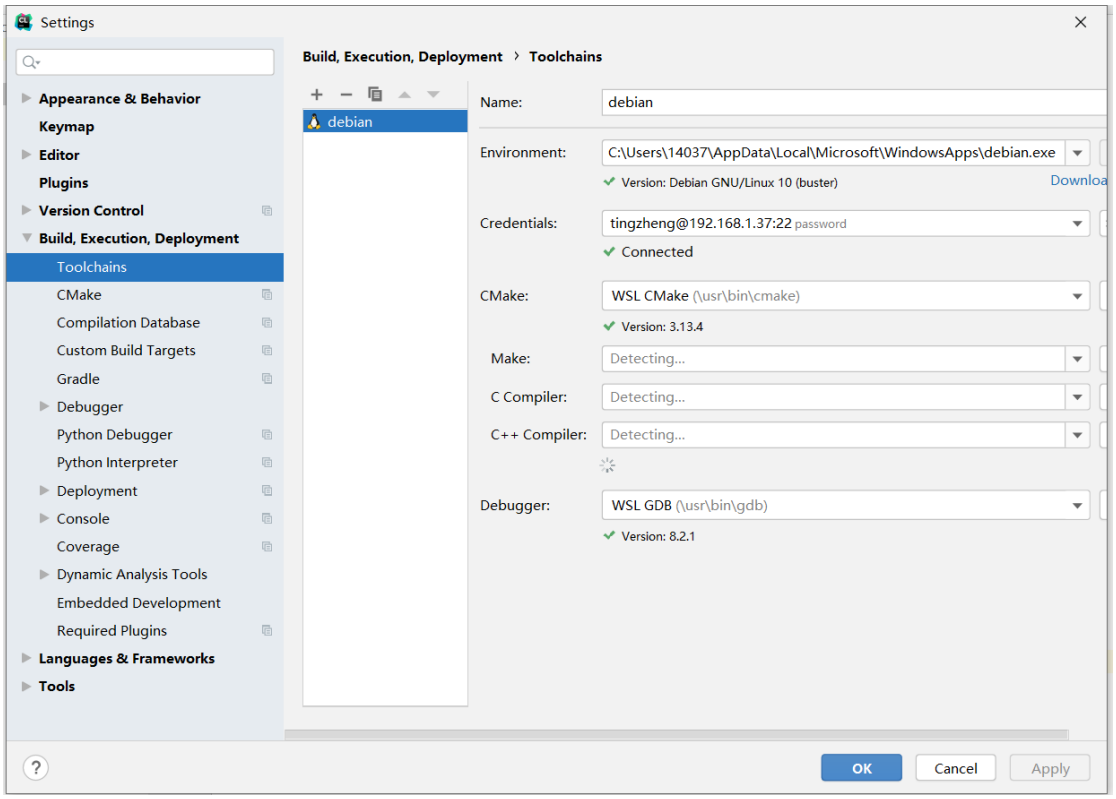
Clion 配置

Clion 的安装就不再赘述。我们先创建一个 c 项目：



找到 Setting 里的 Toolchains，添加一个 wsl 环境其中：

- Environment：wsl 环境，也就是 Windows 下 wsl 所在地，自动配置
- Credentials：ssh 登录，使用刚才的 ip 和你用户的账户密码登录
- CMake：之前已经安装，自动检测
- Debugger：调试器，为 gdb，之前已经安装，自动检测



思路设计

哲学家吃面

题目要求是模拟哲学家吃面，我们就来回顾一下哲学家吃面问题，哲学家吃面问题可以阐述为:
几个哲学家围坐在一个圆形桌子周围，每个哲学家的左手边和右手边都有一个叉子，哲学家不是在思考就是在吃面，但是如果哲学家如果要吃面，就需要拿到左手右手两个叉子。如果只拿到一只叉子，由于人的利己性，哲学家都不愿意放弃自己手里的叉子。存在一种情况，每个哲学家都同时拿起自己左边或者右边的叉子，此时桌上所有的哲学家就都拿不到两只叉子，因为每个哲学家都不愿意放弃自己已经拿到的叉子，导致其它哲学家无法凑齐叉子。如果每个哲学家同时开始抢叉子，那么我们可以知道，一个哲学家选择一边叉子的概率是:

$$P_{phi}(fork) = \frac{1}{2}$$

那么他们选择同一边的概率是:

$$P_{deadlock} = \prod_{phi}^{N} P_{phi}(left|right) = \left(\frac{1}{2}\right)^{N}$$

这种大家都抢不到的现象就叫做**死锁**。

死锁的必要条件

我们如果把哲学家当作计算机中的线程的话，如果要死锁发生，就必然满足下面的条件:

1）互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

2）请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

3）不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

4）环路等待条件：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，…，Pn}中的 P0 正在等待一个 P1 占用的资源；P1 正在等待 P2 占用的资源，……，Pn 正在等待已被 P0 占用的资源。
我们的所有解决死锁问题的方法都是围绕: “如何破坏死锁的必要条件“的思路来设计的。

避免死锁的方法

避免死锁的方法不是唯一的，在本次实验中，我只介绍一种，我们可以破坏条件 2，也就是请求和保持条件，我们可以设定当一个线程已经拥有一个资源时，如果它需要获取的第二个资源被其它线程占用，该线程就必须放弃获取这个资源并且放弃之前已经获得的资源，在代码层面，我们可以使用 thread_mutex_trylock 函数来获取第二个资源，与 thread_mutex_lock 函数不同的是，当 thread_mutex_trylock 所操作的 mutex 被其它线程占用时，本线程并不会进入阻塞状态，而是返回一个错误报告，我们可以利用它来实现我们的算法。

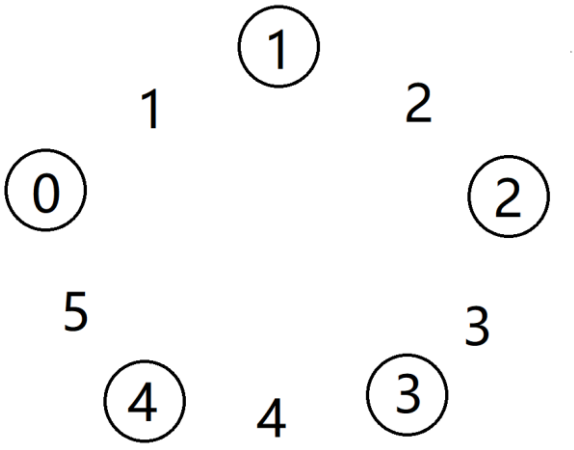
自定义哲学家数量

题目要求可以自定义哲学家数量，这里关系到两个值，一个是哲学家的数量，一个是叉子的数量，哲学家的数量体现在线程的数量上，叉子的数量体现在保存叉子的数据结构的长度上，对于线程数量，我们只在一个地方使用它，所以我们可以使用还没被初始化的变量作为它的长度，那么我们就可以直接使用数组储存线程，但是由线程的创建过程可知，我们需要在 pthread_init 函数和线程运行函数中同时使用叉子数组，所以我们无法用未初始化的变量作为长度，也就无法直接使用数组，我们有以下几种方案

- 选择一种没有空间限制的数据结构，储存叉子，比如说链表，但是链表查询不太好
- 在初始化 fork 数组的时候使用已经定义好的叉子数量最大值，算是折中方案
- 使用指针，这个我还没搞明白，可能可以

哲学家序号与叉子序号的关系

我们可以预见的是，我们需要根据哲学家的序号，得到哲学家左边和右边叉子的序号，那么他们的关系如何呢？我们假设它们的位置关系如下，圆圈代表哲学家，不带圆圈的代表叉子。



我们可以找到如下规律：

- 一个哲学家左边叉子的序号总是等于它本身的序号加一
- 除了 0 号哲学家，其它哲学家的右边叉子序号等于它本身的序号
- 0 号哲学家的右边的序号等于哲学家的数量

值得一提的是，叉子和哲学家的序号关系并不是唯一的，我只是给出一种比较简单的。

程序设计

在本节中，我将利用 pthread.h 中的几个函数来模拟哲学家吃面的过程，源代码托管在这里。

Cmake 配置

由于 pthread 库并不被 cmake 原生支持，我们需要对 cmake 进行一些配置，我们需要在项目的 CMakeList.txt 下添加：

```
find_package(Threads REQUIRED)
target_link_libraries(philosopher Threads::Threads)
```

其中 **philosopher** 为项目名称。

配置头文件

配置头文件允许用户设定一些必要的信息，比如说：

- 最大哲学家数量：由于上文提到的局限，我们不能创建无限多个哲学家
- 非法输入内容：当输入一个非法的哲学家数量时，应该设置的哲学家数量的内容

代码可以为：

```
#define philosopherNumberMax 100
#define validInputCode -1
```

工具模块

工具模块的任务是为其它函数提供必要的工具，这里的工具有：

- 设置哲学家数量函数
- 获得某个哲学家左边叉子序号函数
- 获得某个哲学家右边叉子序号函数

根据思路设计中的**哲学家序号与叉子序号的关系**中的内容，我们可以很容易编出程序：

下面是设定哲学家数量的函数，代表哲学家的数量的变量 philosopherNumber 被定义在这个函数所在.c 文件所对应的头文件里。

```
/*
 * 设定哲学家的数量，哲学家的数量将存入数组中供其它程序调用
 * @param number: 哲学家的数量
 */
void setPhilosopherNumber(int number)
{
    if(number <= philosopherNumberMax)
    {
```

```
            philosopherNumber = number;
            printf("philosopher number has been set as %d!\n", number);
        } else
        {
            printf("Philosopher Number bigger than PhilosopherNumberMax %d, valid!
reset!\n", philosopherNumberMax);
            return ;
        }
}
```

下面是返回哲学家左边和右边叉子序号的函数:

```
/*
 * 返回一个哲学家的左边叉子的序号
 * @param philosopherCode  哲学家的序号
 * return  一个整型变量，是该哲学家左边叉子的序号
 */
int returnLeft(int philosopherCode)
{
    return philosopherCode + 1;
}



/*
 * 返回一个哲学家的右边叉子的序号
 * @param philosopherCode  哲学家的序号
 * return  一个整型变量，是该哲学家右边叉子的序号
 */
int returnRight(int philosopherCode)
{
    if(philosopherCode != 0)
    {
        return philosopherCode;
    }
    return philosopherNumber;
}
```

输入模块

输入模块的作用是接收并处理用户输入的哲学家的数量。它需要有两个函数，一个是接收函数，这个函数需要有一定的判别非法输入的能力，还有一个是更具输入的合法数字设定哲学家数量的函数，这里需要调用之前工具模块的函数：

这个是获得用户输入的函数

```
int getInput()
{
    int data;
    if(scanf("%d",&data))
    {
        return data;
    }
    return validInputCode;
}
```

这个是设置哲学家数量的函数：

```
void setPhilosopherNumberWithInput()
{
    int res;
    res = getInput();
    if(res != validInputCode)
    {
        printf("you got an philosopher number: %d\n ", res);
        setPhilosopherNumber(res);
    }
    else {
        printf("Input code valid\n");
    }
}
```

哲学家模块

在这个模块中，需要完成线程的创建，线程函数（吃面过程）的编写。
在给线程传参数的不能穿 循环遍历 i 的 地址，因为 i 在主线程中 ，被多个线程共享，所以不是唯一的。那么如何让每个线程 都有独自拥有自己的顺序编号呢？有两种方法：

- 当然可以在堆上开辟空间存储顺序编号呗。自己有自己的顺序编号的空间各自独立。
- 就是参数是 void* 可以直接将循环变量 i 直接传给 void* ,由于 arg 是每个线程 栈空间上的变量 故此 属于各个子线程，然后在使用的时候强转回 int，因为 void* 和 int 刚好 都是 4 字节，这样做是安全的。

```
int start(){
    pthread_t phiThread[philosopherNumber];

    for (int a = 0; a < philosopherNumber; a++)
    {
        pthread_mutex_init(&forkp[a], NULL);
        printf("thread %d inited\n", a);
    }

    for (int b = 0; b < philosopherNumber; b++)
    {
        int* temp = (int*)malloc(sizeof(int));
        *temp = b;
        pthread_create(&phiThread[b], NULL, eat_think, (void *)temp);
        printf("thread %d created\n", b);
    }

    for (int c = 0; c <philosopherNumber; c++)
        pthread_join(phiThread[c], NULL);
    return 0;
}
```

在下面这个函数中，我们规定了每个哲学家抢叉子的过程:

```
_Noreturn void *eat_think(void *arg)
{
    int phi = *(int *)arg;
    int left,right;
    left = returnLeft(phi);
    right = returnRight(phi);

    for(;;){
        sleep(rand()%4);//思考 0~3 秒
        pthread_mutex_lock(&forkp[left]);//拿起左手的叉子
        printf("哲学家 %d 拿起左手的叉子 %d\n", phi, left);
        if (pthread_mutex_trylock(&forkp[right]) == EBUSY){//拿起右手的叉子
            pthread_mutex_unlock(&forkp[left]);//如果右边叉子被拿走放下左手的叉子
            printf("哲学家 %d 放下左手的叉子 %d\n", phi, left);
            continue;
        }
        printf("哲学家 %d 拿起右手的叉子 %d\n", phi, right);
        printf("哲学家 %d 在吃饭\n",phi);
        printf("--------------------------------\n");
        sleep(rand()%4);//吃 0~3 秒
        pthread_mutex_unlock(&forkp[left]);
```

```
            pthread_mutex_unlock(&forkp[right]);
        }
}
```

我们可以看到，当哲学家拿起左边的叉子后，他要去拿另外一个叉子，这个时候，他不采用 thread_mutex_lock 而采用 thread_mutex_trylock, 原因是他要判断第二个叉子是否有人拿了，如果有人拿，这个函数将返回 **EBUSY，**那么按照设定，他将丢弃左边的叉子，这就破坏了请求和保持条件，就不会发生死锁，如果我们要模仿会发生死锁的情况，可以把 thread_mutex_trylock 换回 thread_mutex_lock。

主函数

主函数的内容是一个获取哲学家数量函数，和一个开始吃面函数：

```
int main() {
    setPhilosopherNumberWithInput();
    start();
    return 0;
}
```

测试

我们把程序改成回发生死锁的，我选择输入 10 个哲学家。

```
10
10
you got an philosopher number: 10
 philosopher number has been set as 10!
```

可以观察到，大约结果 180 次操作，整个程序的所有线程就全部阻塞了，也就是说发生了死锁！

```
哲学家 1 在吃饭
---------------------------------
哲学家 1 拿起左手的叉子 2
哲学家 1 拿起右手的叉子 1
哲学家 1 在吃饭
---------------------------------
哲学家 1 拿起左手的叉子 2
哲学家 1 拿起右手的叉子 1
哲学家 1 在吃饭
---------------------------------
哲学家 9 拿起左手的叉子 10
哲学家 1 拿起左手的叉子 2
哲学家 0 拿起左手的叉子 1
```

我把程序改回不发生死锁的情况，同样设置 10 个哲学家，我们发现，程序一直为发生全部哲学家都阻塞的情况，整个系统一直未发生死锁，我们成果避免了死锁。

```
10
10
you got an philosopher number: 10
 philosopher number has been set as 10!
```

测试结果可以在此处查看，其中没有发生死锁的部分，我只截取了一部分输出，实际上，输出是无限多的，因为不会发生死锁。

参考

- 解决 Debian9 下 ifconfig command not found
- pthread_create 传递参数
- Ubuntu - 本地 SSH 连接 WSL【WSL 第二弹】
- Clion 如何编译支持 pthread
- Linux 系统编程：循环创建 N 个子线程并顺序输出