学科实践

实验（一）体验Nachos下的并发程序设计

姓　　名	熊恪峥
学　　号	22920202204622
日　　期	2023年3月28日
学　　院	信息学院
课程名称	操作系统

实验（一）体验Nachos下的并发程序设计

目录

1 使用Docker安装Nachos **1**

1.1 构建容器 1

1.2 创建实例 1

1.3 配置VSCode 1

2 实现双向有序链表 **2**

3 体验Nachos线程系统 **3**

3.1 工具宏定义 3

3.2 使用上述工具宏改造双向有序链表 4

3.3 使用RAII自动检测错误 4

3.3.1 什么是RAII 5

3.3.2 使用RAII的方式自动检查并发错误 5

3.4 实验结果 7

4 实验总结 **7**

1 使用Docker安装Nachos

1.1 构建容器

由于Nachos需要使用较旧的工具链，因此在现代操作系统上使用Nachos可能会遇到一些问题。然而专为Nachos安装旧版本的虚拟机比较麻烦，因此为了方便开发、方便阅读和修改代码，可以使用Docker安装Nachos。

首先，由于Docker Hub没有现成的容器，要先编写Dockerfile，然后使用Docker build命令构建容器。如代码 1 然后使用命令

代码 1 Dockerfile

```
ARG UPSTREAM_IMAGE=i386/ubuntu:bionic

FROM ${UPSTREAM_IMAGE}

# install compiler
RUN apt update && \
    apt install -y ed csh gdb build-essential && \
    apt clean -y && \
    rm -rf /var/lib/apt/lists/*

# change workspace
WORKDIR /workspace

ENTRYPOINT [ "/bin/bash" ]
```

```
docker buildx build -t nachos .
```

就可以按照Dockerfile进行构建。

1.2 创建实例

接下来，可以使用Docker创建一个容器的实例来运行Nachos。运行命令

```
docker run -v $(pwd):/workspace -it nachos
```

就可以使用刚才构建的容器创建一个容器的实例，并将容器的/workspace目录映射到当前目录。在Docker Desktop中可以看到刚才创建的容器实例。如图 1所示。 容器在Running状态，说明容器已经启动。在Docker

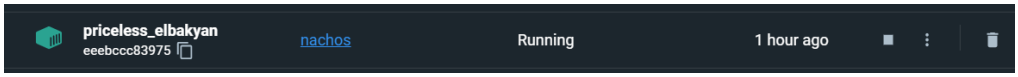


图 1: Docker Desktop

Desktop中可以打开终端、进行配置等操作，但是为了更好地开发，可以使用VSCode连接容器进行开发。

1.3 配置VSCode

VSCode有完善的远程开发功能，可以连接到容器进行开发。在Remote Explorer中选择Containers，刷新，就可以看见刚刚创建的容器实例，如图 2 然后右键点击“Attach to Container”，选择刚才创建的容器实例，就可以连接到容器进行开发了。

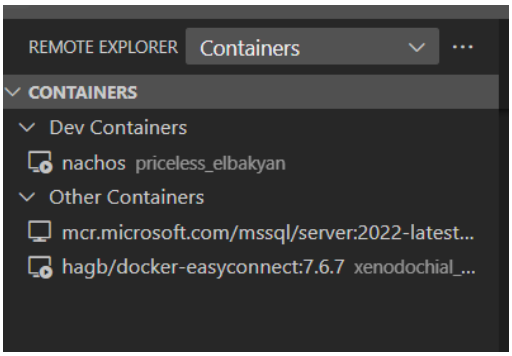


图 2: VSCode Containers

在VSCode终端中可以编译代码，如图 3。绝大多数模块都能成功编译，可以支持进行正常的实验，由于编译器版本还是较高，`tests`目录下有少数几个模块无法编译，但是不影响实验。可以在Makefile中移除它们，或者忽略相关的报错。

```
echo '# IF YOU PUT STUFF HERE IT WILL GO AWAY' >> Makefile
echo '# see make depend above' >> Makefile
make[1]: Leaving directory '/workspace/nachos-linux64/nachos-3.4/code/network'
cd network; make nachos
make[1]: Entering directory '/workspace/nachos-linux64/nachos-3.4/code/network'
g++ -m32 -g -Wall -Wshadow -traditional -I../network -I../bin -I../filesys -I../vm -I../userprog -I../threa
ds -I../machine -DUSER_PROGRAM -DVM -DFILESYS_NEEDED -DFILESYS -DNETWORK -DHOST_i386 -DCHANGED  -c ../threa
ds/dllist.cc
g++ -m32 main.o list.o scheduler.o synch.o synchlist.o system.o thread.o hello.o dllist.o dllist-driver.o u
tility.o threadtest.o interrupt.o stats.o sysdep.o timer.o addrspace.o bitmap.o exception.o progtest.o cons
ole.o machine.o mipssim.o translate.o  directory.o filehdr.o filesys.o fstest.o openfile.o synchdisk.o disk
.o nettest.o post.o network.o switch.o  -o nachos
make[1]: Leaving directory '/workspace/nachos-linux64/nachos-3.4/code/network'
cd bin; make all
make[1]: Entering directory '/workspace/nachos-linux64/nachos-3.4/code/bin'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/workspace/nachos-linux64/nachos-3.4/code/bin'
root@eeebccc83975:/workspace/nachos-linux64/nachos-3.4/code#
```

图 3: Build Nachos

在`Makefile.common`中的THREAD_C，THREAD_H，和THREAD_O中分别加入 hello.c, hello.h和hello.o，然后在其中实现输出Hello的函数，如代码 2，然后在`ThreadTest`中调用，可见在Nachos中添加的模块也可以正常运行。如图 4。

代码 2 hello.c

```c
#include "copyright.h"
#include "system.h"

void hello()
{
    printf("Hello, nachos! I'm %lld\n", 22920202204622ULL);
}
```

```
root@eeebccc83975:/workspace/nachos-lin
Hello, nachos! I'm 22920202204622
```

图 4: Hello Nachos

2 实现双向有序链表

双向有序链表就是双向的、元素有序的链表。首先，按照上述加入hello的方法加入dllist 相关的文件。然后进行实现。为了保证链表有序，在插入时需要遍历链表、先找到元素应该插入的位置，然后再插入。其余

部分都按照相关文件中的注释进行实现即可。

3 体验Nachos线程系统

由于观察多线程访问中发生的问题需要多个线程、运行多次，而修改代码进行输出、在较多的日志中寻找偶然发生的并发问题是一件相当需要时间的繁琐的事情。因此，需要对代码进行适当的修改和设计，在保持代码可读性、可维护性的前提下插入辅助观察并发错误的业务逻辑，这一点需要较多的设计和改进。

因此，为了良好的完成实验、避免在繁琐的分析和查找中出错，更好地观察到各种并发问题，首先需要更好的工具。因此，我将把我在这一节进行的工作按如下的顺序进行介绍：

1. **工具宏的定义**：使用宏定义来简化在不同位置按需调用Yield的操作，以及输出调用Yield的位置等信息；
2. **使用上述工具宏改造双向有序链表**：应用上述宏定义，方便简洁地实现能通过命令行参数指定代码中调用Yield的位置的功能；
3. **使用RAII自动检测并发错误**：利用现代C++中常见的RAII范式完成自动的并发错误检查器，检查失序等比较难以直接观察和分析的并发错误；
4. **实验结果**：错误内容的截图和讨论；

3.1 工具宏定义

为了方便地观察并发中的现象，可以定义一些宏来使得代码更清晰。首先，为了观察不同位置调度后产生的问题，需要调用Yield，并且为了方便调试，还需要输出调用Yield的位置等信息，以便于观察。因此可以定义一个宏，如代码 3。

代码 3 logyield.h

```
#define DEBUG_YIELD_OUTPUT
#ifdef DEBUG_YIELD_OUTPUT
#define YIELD_AND_REPORT() \
do { \
printf("In function \"%s\":\n  Switch to another thread at code line %d!\n", __FUNCTION__, __LINE__);
currentThread->Yield(); \
} while (0)
#else
#define YIELD_AND_REPORT() \
do { \
currentThread->Yield(); \
} while (0)
#endif
```

在这段代码中，DEBUG_YIELD_OUTPUT作为开关，选择YIELD_AND_REPORT的内容。如果定义了DEBUG_YIELD_OUTPUT，则会输出调用Yield的位置等信息，否则不会输出。为了获取这些位置信息，可以使用编译器预定义的宏__FUNCTION__和__LINE__。它们是调用位置函数的名称和代码行号。在这里，使用了do-while循环，是为了保证宏的语句块的完整性，不会在替换时出错。

这样，在任何位置调用YIELD_AND_REPORT，都会调用Yield并按需要输出调用位置的信息。

代码中许多部分都需要测试是否发生并发问题，因此，首先需要在main.cc中加入3个参数，分别是测试时使用线程的数量t，测试的次数n，和需要Yield的位置标号，如代码 4。 这添加在main.cc95行开始for语句中的switch语句中。

代码 4 main.cc

```
case 't':                   // number of threads
        threadnum = atoi(argv[1]);
        argCount++;
        break;
case 'n':                   // number of items
        itemnum = atoi(argv[1]);
        argCount++;
        break;
case 'e':           // type of error
        errorType = atoi(argv[1]);
        argCount++;
        break;
```

在双向有序链表的实现中，需要判断存储需要Yield的位置标号，然后调用YIELD_AND_REPORT。这一过程也可以用宏来简化，以减少代码中的重复。如代码5。这样，在需要判断并调用YIELD_AND_REPORT的地方，只

代码 5 logyield.h

```
#define YIELD_ON_TYPE(t) \
if (errorType == t)          \
{ \
        YIELD_AND_REPORT(); \
} \
```

需要调用YIELD_ON_TYPE即可。传入的参数是标识当前位置的数值，它由命令行参数传入并放入变量errorType中。

假设调用了YIELD_ON_TYPE(1)，则会判断errorType是否为1，如果是，则调用。此时，如果在执行nachos时传入了命令行参数-e 1，则会在当前位置调用YIELD_AND_REPORT。

这样有了以上的宏定义，就可以用一行代码来标记当前位置、处理命令行参数，以及按需求调用Yield。

3.2 使用上述工具宏改造双向有序链表

接下来，为了方便观察并发中的现象，需要对双向有序链表进行改造。将上述工具宏实际应用到双向有序链表中。并发的错误通常出现在读写链表节点的前后指针的位置。因此，为了观察这些位置的并发问题，需要在这些位置插入上一节定义的YIELD_ON_TYPE(标识ID)。这里，标识ID的值可以自己定义，只要保证不同的位置使用不同的标识ID即可。在启动Nachos时，将对应的标识ID作为参数传入，就可以在指定的位置调用Yield。

3.3 使用RAII自动检测错误

并发错误具有随机发生、频率不定的特点。为了引发并发错误，就需要运行多个线程、运行多次。在这个过程中，需要打出大量的log来进行分析。这个过程较为繁琐，在大量的log中查找失序等问题并不容易。因此需要更好的方法检测问题。

其中一种方法是在在所有修改结点指针处插入各种检查代码。然而这种侵入式的修改方式需要对代码进行大量的修改，也会影响程序结构使得代码难以理解。并且许多检查代码都是重复的，这会导致代码冗余。因此，需要一种非侵入式的自动检查方案。

3.3.1 什么是RAII

RAII是C++中一种非常重要的编程技巧，它代表“资源获取即初始化”（Resource Acquisition Is Initialization）的缩写。其核心思想是利用对象的构造函数和析构函数，在对象生命周期结束时自动释放资源。通过RAII，程序员可以更加便捷、安全地管理内存、文件句柄、锁、网络连接等各种资源，从而避免因为手动管理资源而引入的错误。

RAII可以帮助避免许多常见的错误，如忘记释放资源、过早或过晚释放资源等。比如，如果程序员手动分配了内存但未在使用后及时释放，就会导致内存泄漏问题。而RAII可以在对象失效时自动调用析构函数，从而正确释放对象占用的资源，避免内存泄漏问题。

实现RAII需要将资源的管理封装在一个类中。例如，标准库对于动态内存的管理，实现了unique_ptr等一系列智能指针智能指针来确保在对象失效时自动释放内存。

RAII不仅可以避免资源泄漏问题，还可以提供异常安全保障。当程序抛出异常时，RAII可以确保已经分配的资源被正确释放，从而避免资源泄漏和内存泄漏等问题。这是因为在异常情况下，C++会自动调用对象的析构函数来释放资源，从而确保程序能够正确处理异常情况。

3.3.2 使用RAII的方式自动检查并发错误

虽然错误检查不是严格意义上的资源管理，但是它可以利用RAII的思想来实现。具体来说，可以将错误检查封装在一个类中，然后在类的构造函数中进行错误检查，而在析构函数中进行错误恢复。这样，当进入相应的作用域中，就会自动进行错误检查，而在作用域结束时，就会自动再次检查，然后按需输出错误状态。这样就可以避免手动检查和恢复错误。这样，在调用端，只需要定义一个RAII对象，然后错误检查就能够自动执行，最大限度减少了错误检查实现的侵入性，使得错误检查更加简单、方便。对于链表失序、指针指向错乱等问题，也能够更好地输出具体的错误信息，从而更加方便地进行分析。

在程序中，我实现了两种粒度的错误检测，分别是对链表的整体检查和对链表节点的检查。链表的整体检查会检查头尾指针、是否有序等，节点级别的检查会检查节点的前后指针、以及前后构成的三元组是否有序。

采用两种不同的粒度的检测，是因为链表结构的完整操作才能保证链表级别的不变性，例如整体的有序性。然而，在链表操作之内的细分步骤中，局部的节点操作也有其不变性需要检查。例如前后局部的有序性、指针指向的一致性。当然，在实践中，可以观察到链表级的错误检查没有发现任何问题，而节点级的错误检查则提供了一些有用的信息。

以节点级的错误检查为例，它的实现如代码 7。在构造函数中，会检查节点的前后指针是否一致，并且检查前后节点的三元组是否有序。在析构函数中，会再次检查节点的前后指针是否一致，并且检查前后节点的三元组是否有序。例如要将其使用在链表的插入操作中，可以在插入操作的每个局部使用大括号{和}标记一个作用域，然后在其中定义一个RAII对象，从而在作用域结束时自动进行错误检查。使用这种方法的实例如插入操作中的代码 6所示。使用这种方法检查，最好将作用域限定为能够维持节点所需要维持的不变性的最小范围。这样，就更容易发现错误。

代码 6 使用示例

```
{
        RAIINodeGuard _1(*last, "last");
        RAIINodeGuard _2(*element, "element");
        last->next = element;
        YIELD_ON_TYPE(19);
        element->prev = last;
        element->next = NULL;
        YIELD_ON_TYPE(20);
        last = element;
}
```

代码 7 节点级的错误检查

```
RAIINodeGuard::RAIINodeGuard(DLLElement& ele, char* name)
        : ele_(&ele), name_(name)
{
        printf("Thread %s enter guarded section for %s. ", currentThread->getName(), name_);
        if (ele_->prev || ele_->next)
        {
                if (!ele_->prev)
                {
                        printf("It's first node.");
                }
                if (!ele_->next)
                {
                        printf("It's last node.");
                }
        }

        printf("\n");
        test();
}

RAIINodeGuard::~RAIINodeGuard()
{
        test();
        printf("Thread %s exit guarded section for %s\n", currentThread->getName(), name_);
}

void RAIINodeGuard::test()
{
        if (ele_->prev && ele_->prev->next != ele_)
        {
                printf("%s has wrong previous relationship.\n", name_);
                ASSERT(false);
        }

        if (ele_->next && ele_->next->prev != ele_)
        {
                printf("%s has wrong next relationship.\n", name_);
                ASSERT(false);
        }

        int diff1 = ele_->prev ? ele_->key - ele_->prev->key : 0;
        int diff2 = ele_->next ? ele_->next->key - ele_->key : 0;
        if (diff1 && diff2 && diff1 * diff2 < 0)
        {
                printf("%s has wrong order. Values are: ", name_);
                if (ele_->prev)
                {
                        printf("%d ", ele_->prev->key);
                }

                printf("%d ", ele_->key);

                if (ele_->next)
                {
                        printf("%d ", ele_->next->key);
                }

                printf("\n");

                ASSERT(false);
        }
}
```

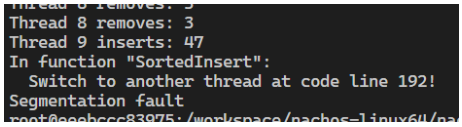
3.4 实验结果

使用以上的措施，就可以方便地进行错误检查。在实验中，我尝试了不同的Yield位置之后，发现了四类错误，它们分别是：

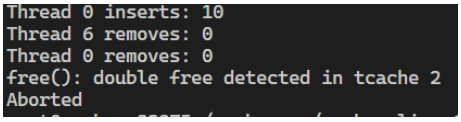
1. 空指针造成的Segmentation Fauld
2. 两次释放同一段内存导致的Double Free问题
3. 链表指针的错乱
4. 链表的失序

输出错误信息如图 5。其中，只有借助上述RAII错误检查才能发现后两种错误。可见，将错误检查进行一定程度的自动化有利于更好地调试并发程序。

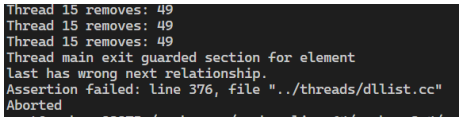
图 5: 错误信息



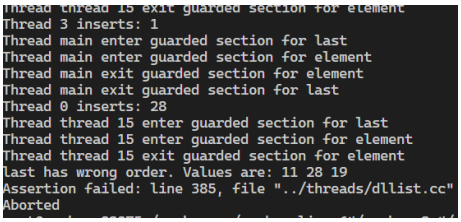
(a) 空指针造成的Segmentation Fauld



(b) Double Free问题



(c) 链表指针的错乱



(d) 链表的失序

4 实验总结

在本次实验中，我通过Docker实现了更方便的环境配置。实现双向有序链表，并借助宏、RAII等功能更方便地观察了并发的问题。对并发可能产生的问题有了进一步的理解。