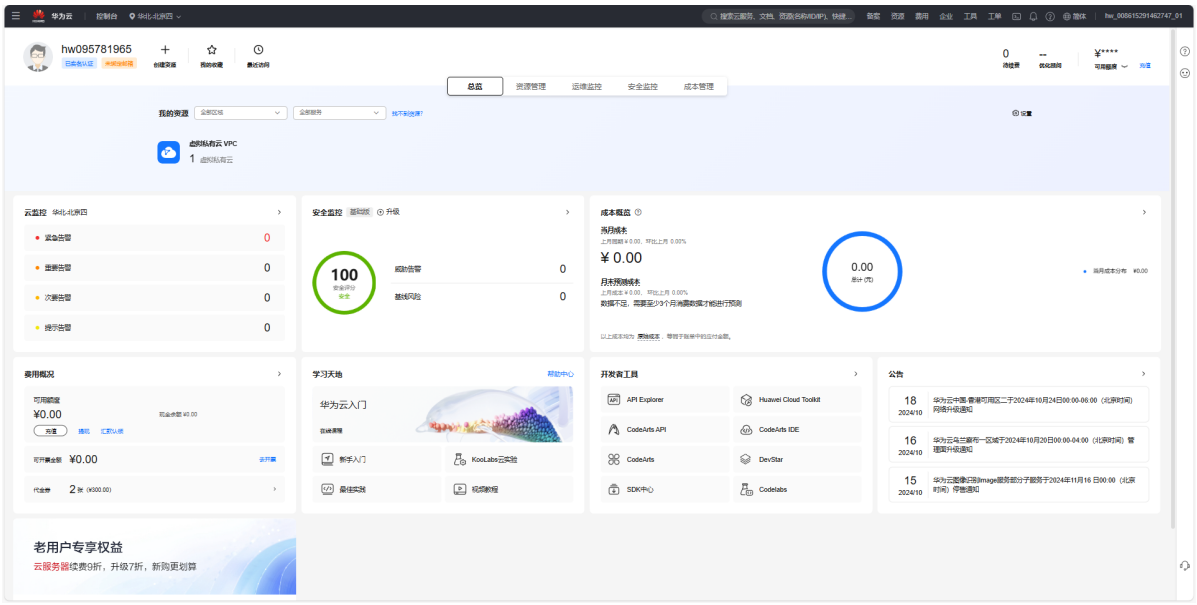
华为元openEuler环境搭建

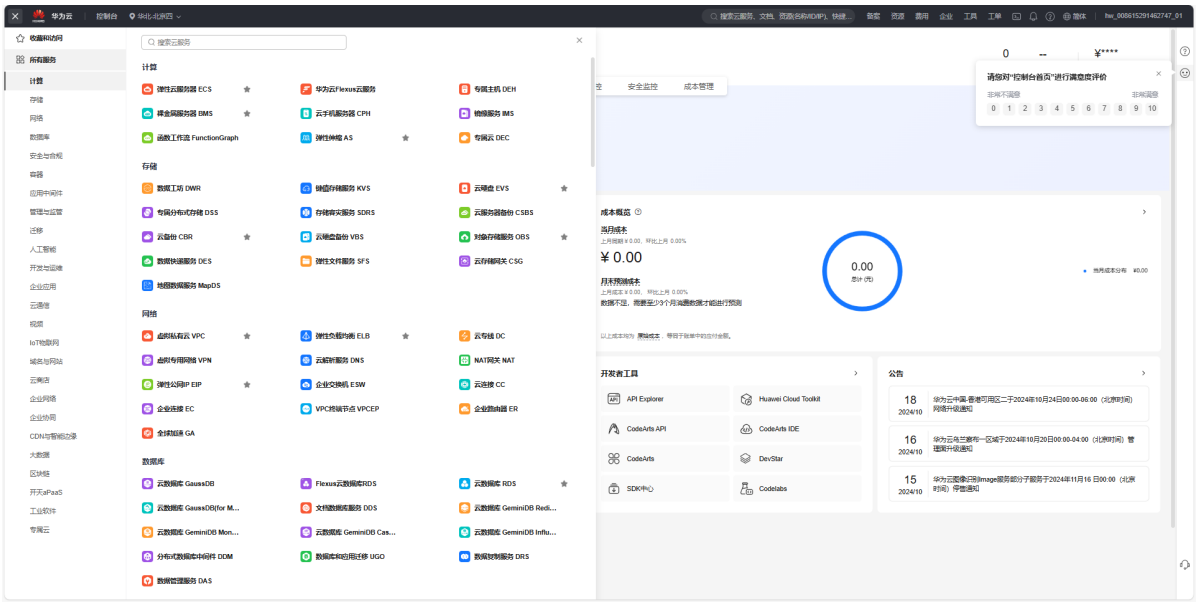
云端布置服务器

登录华为云官网并进入控制台

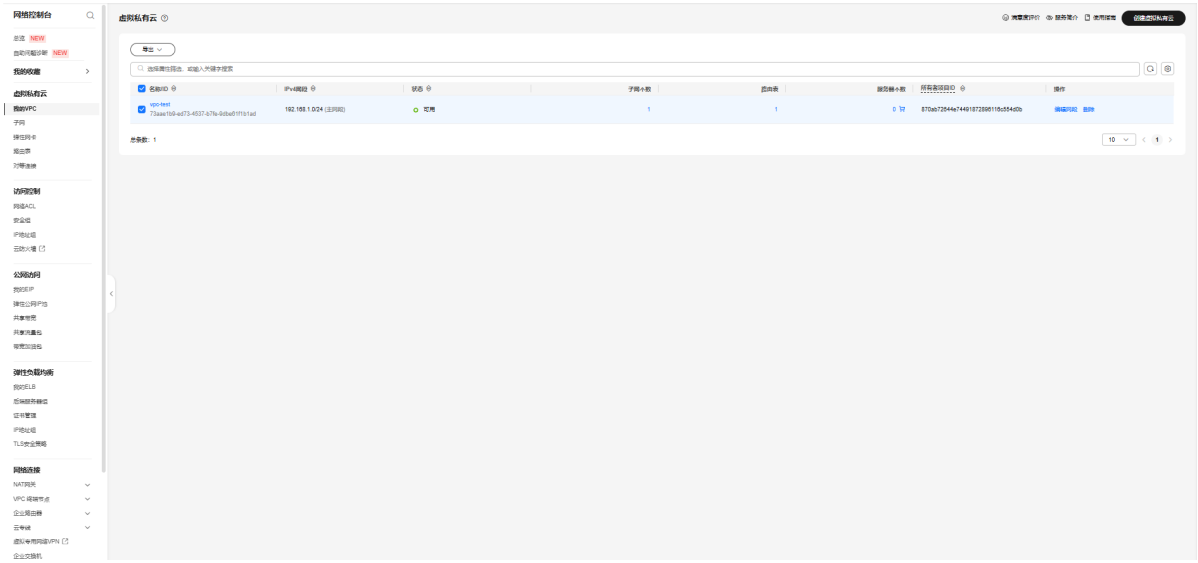


界面与指导书略有不同，默认已是华北-北京四区域

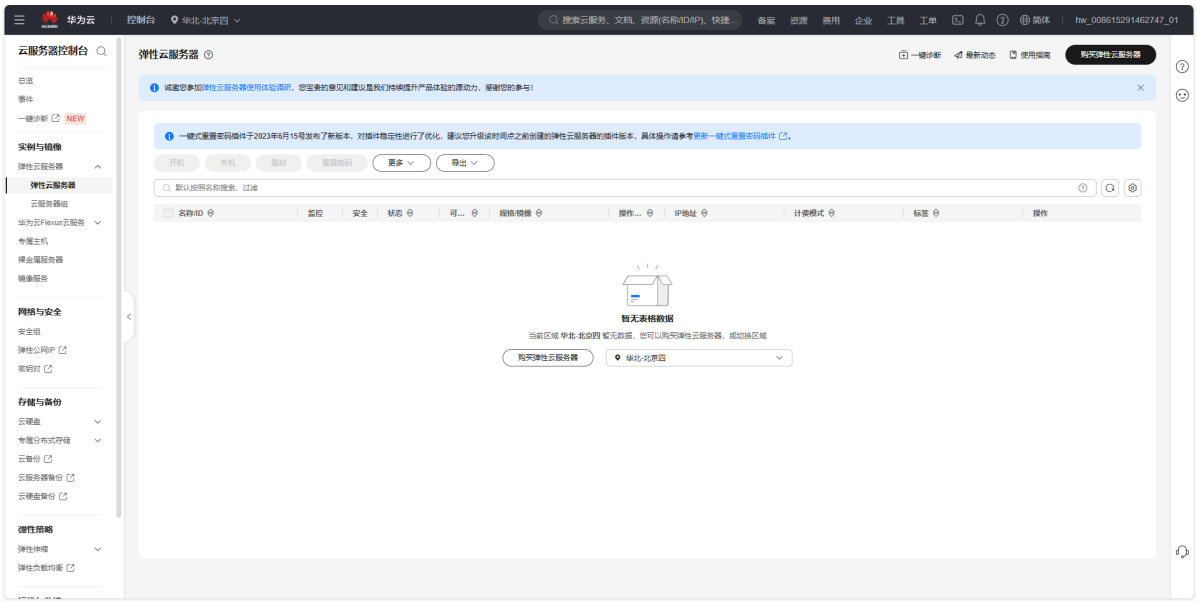
侧边栏与指导书有所不同，通过搜索进入虚拟私有云VPC



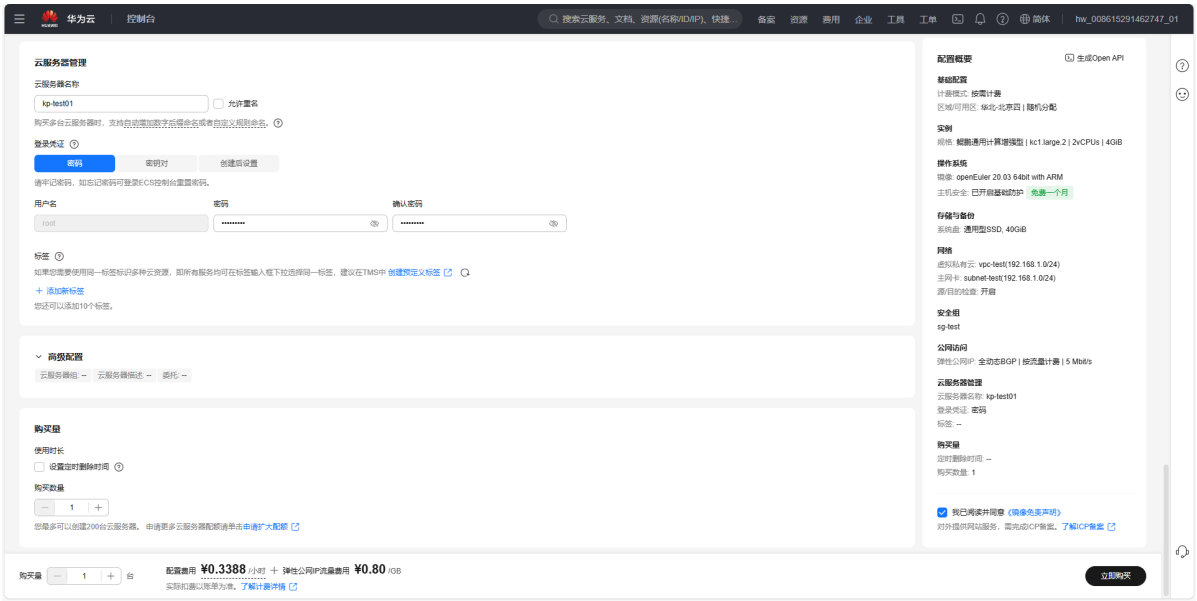
之后根据指导书创建虚拟私有云vpc-test



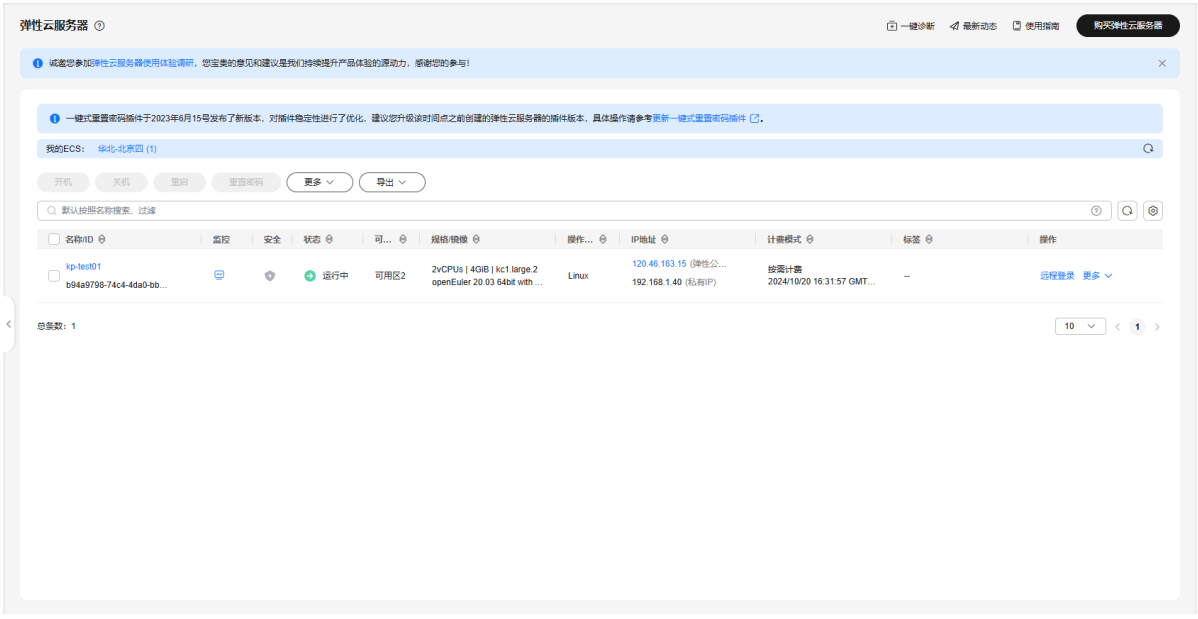
进入弹性云服务器ECS界面，购买弹性云服务器

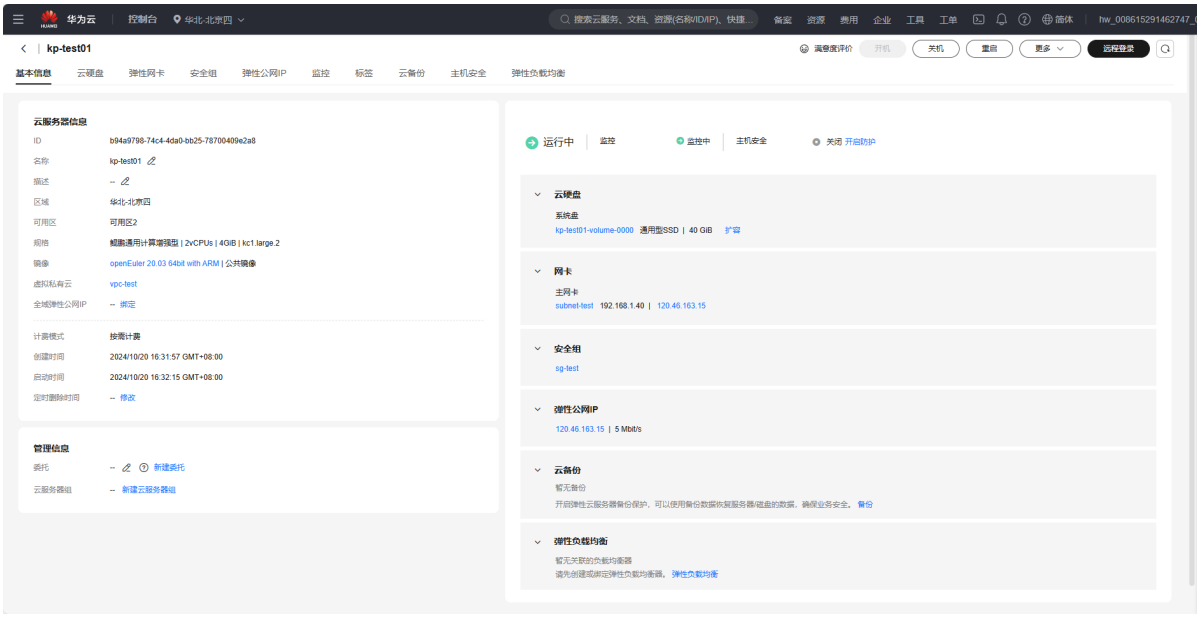


依据指导书配置信息配置好环境系统，如下图

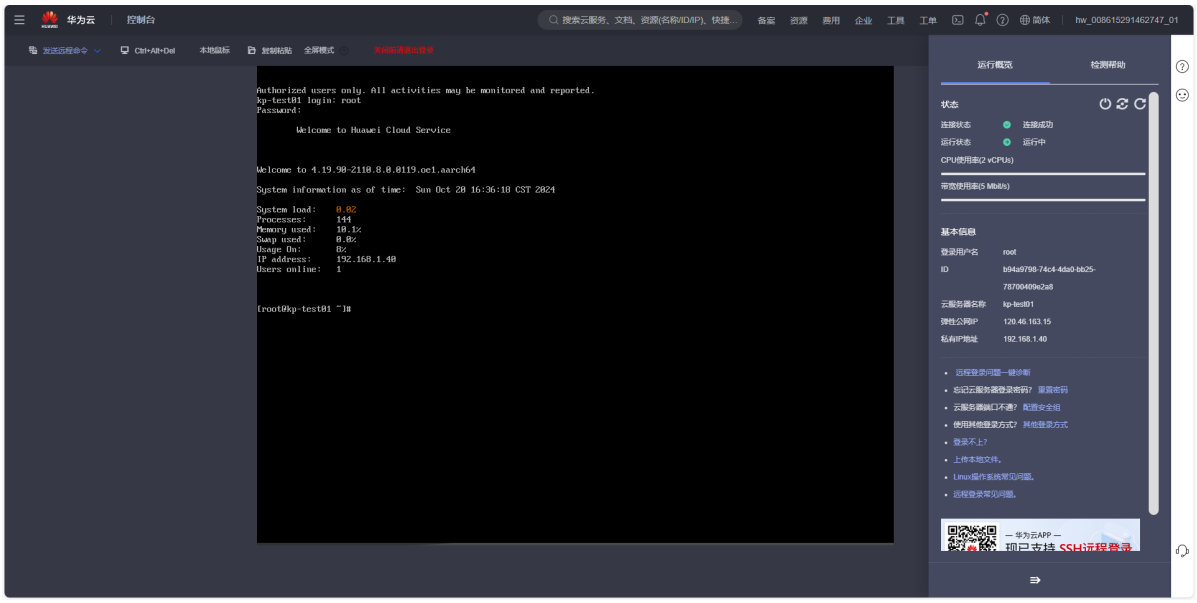


如图所示，创建成功

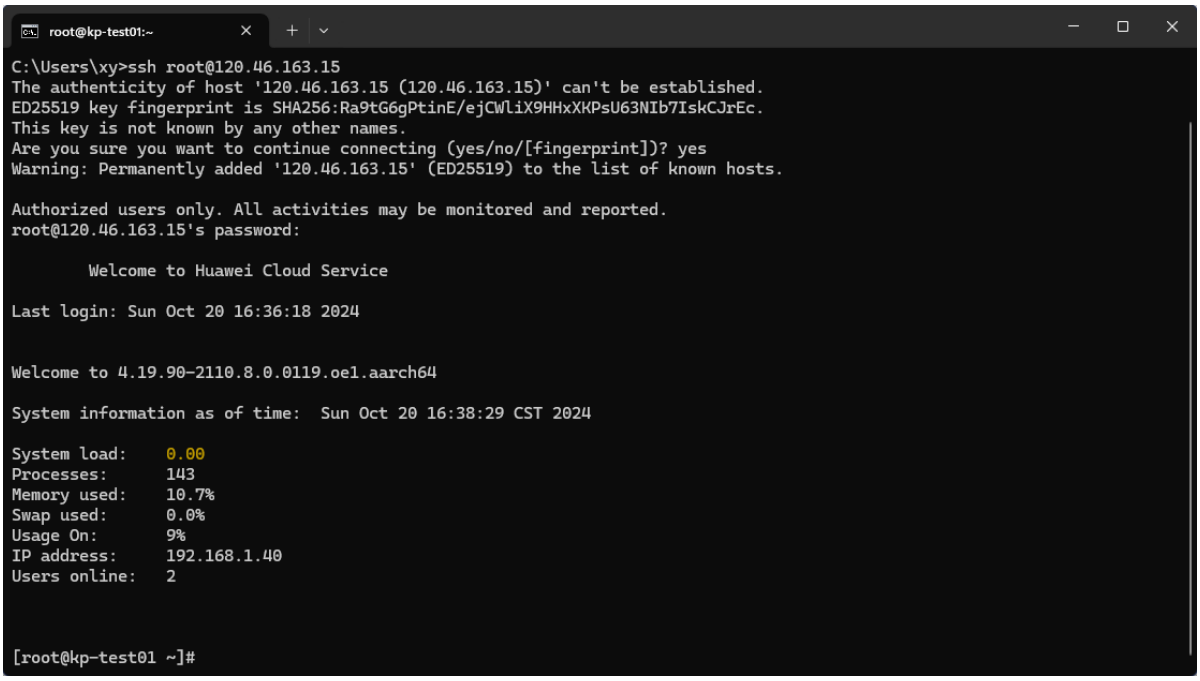




远程登陆连接服务器，同时由于使用web网页中提供的vpc登录，不需要手动输入命令：ssh root@[公网IP]



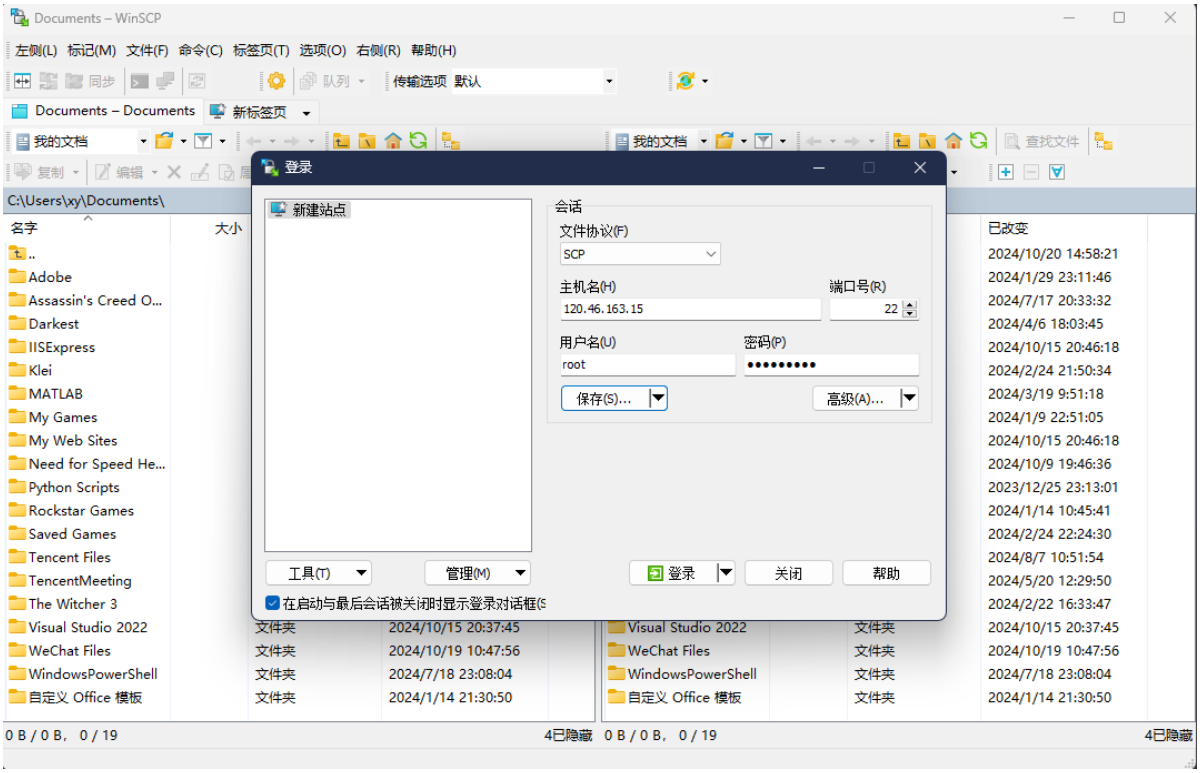
如图为windows中cmd登录



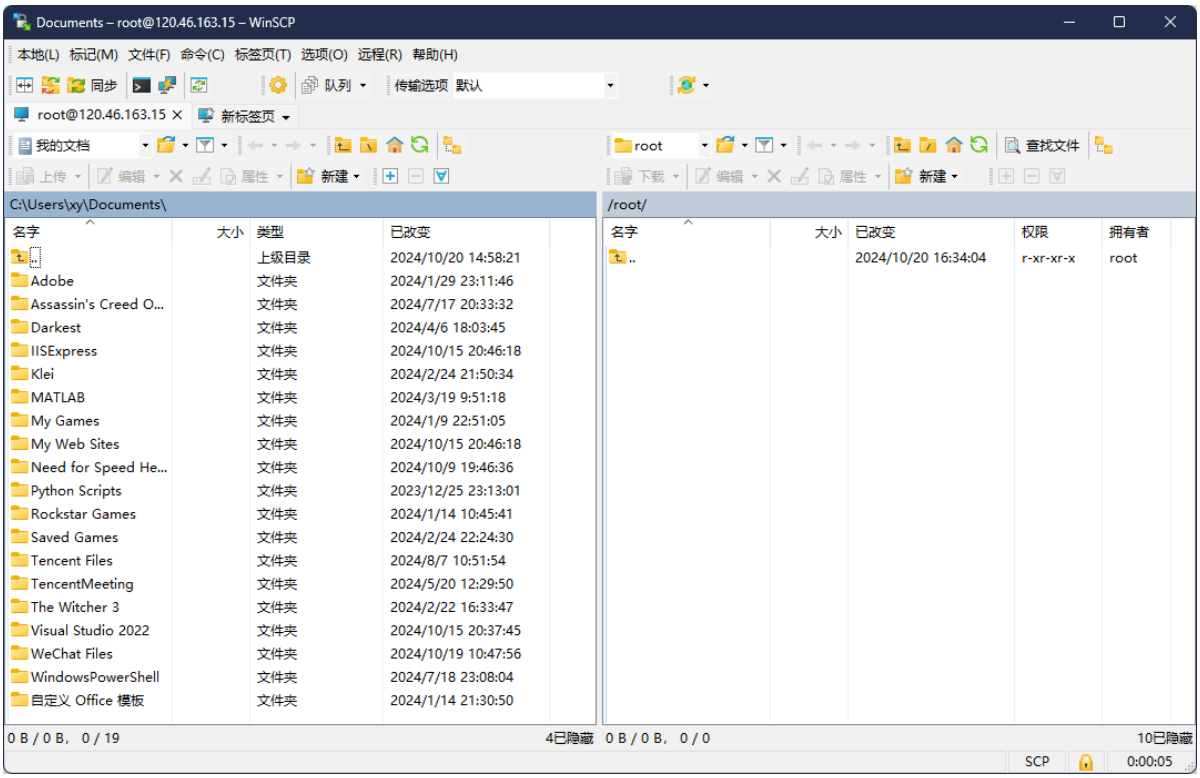
此时由于没有了解过winscp，不知道winscp从哪里使用，查阅以下资料：

- 文件传输工具WinSCP下载安装教程_winscp安装教程-CSDN博客

根据资料，下载安装winscp



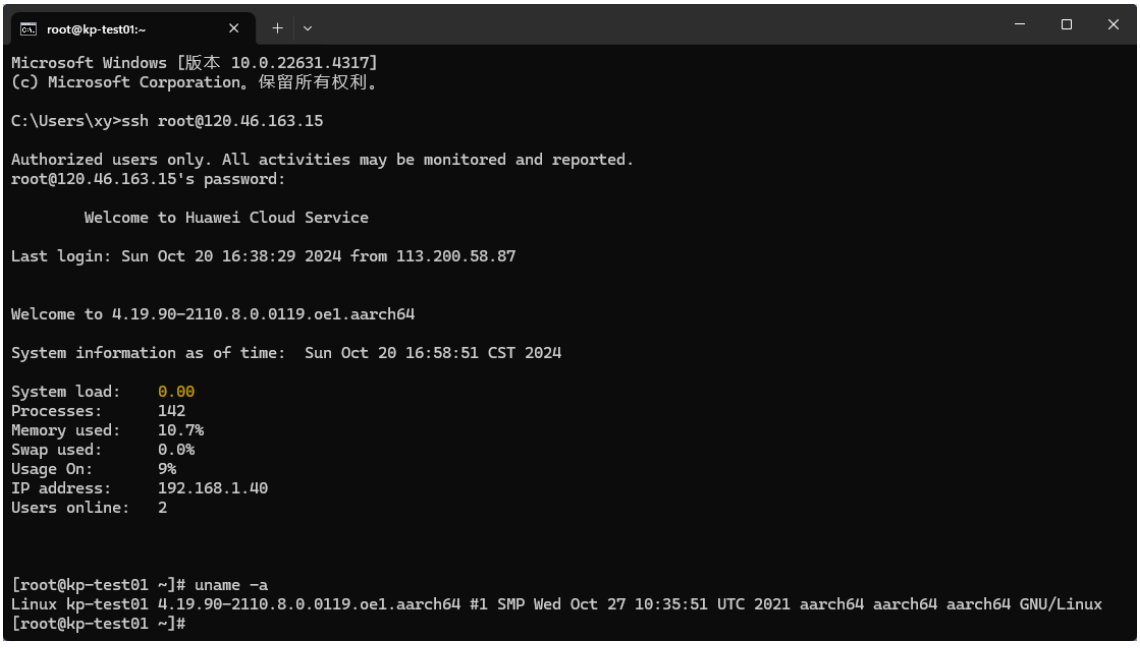
配置好连接站点



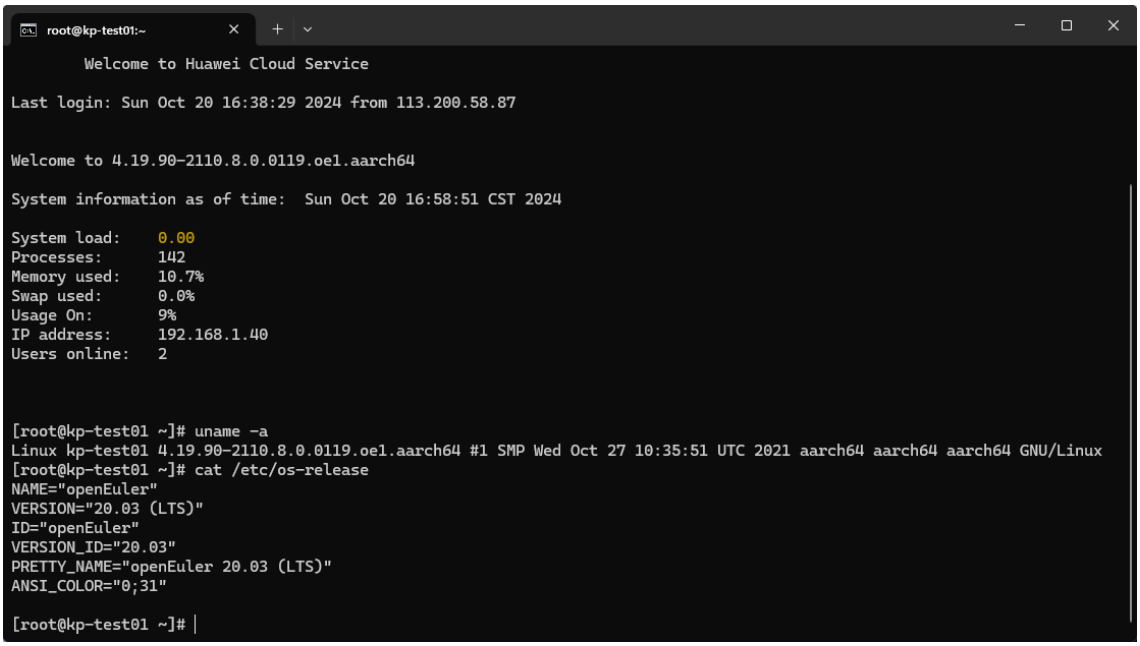
连接成功

在cmd远程登陆界面查看服务器信息

1. uname –a：用以查看总体架构

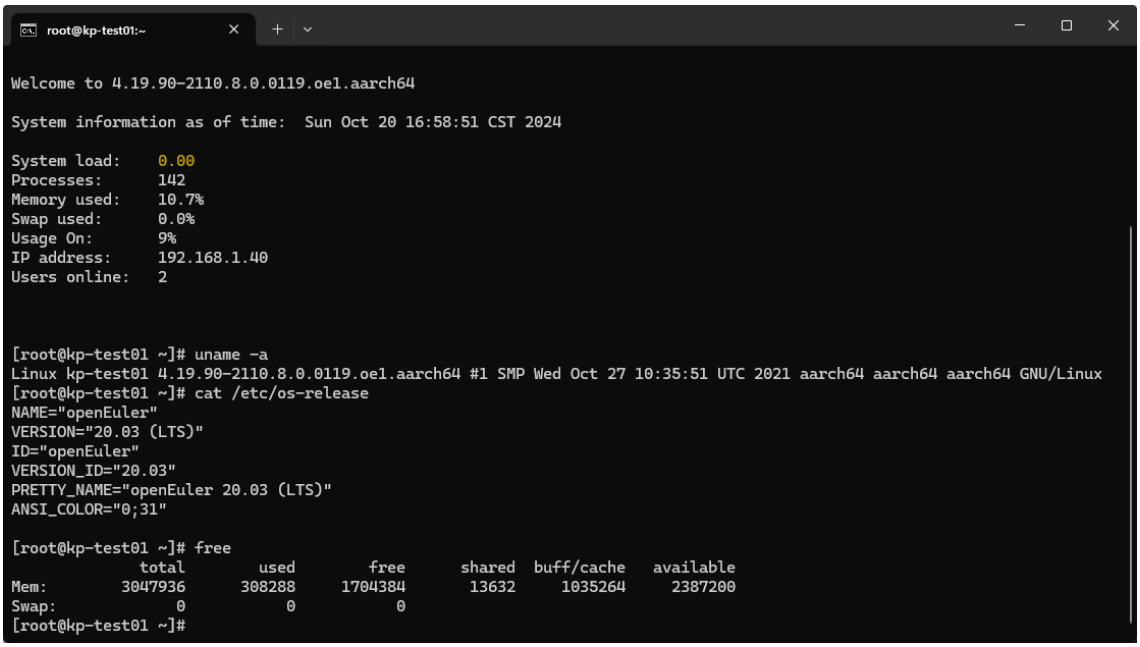


2. cat /etc/os-release：查看操作系统信息



可以看见为openEuler20.03系统

3. free： 查看内存信息



可以看见mem的总大小，及各种用途的不同大小。由swap为0知系统没有交换空间。

此处不知道mem具体代表什么空间，及free命令查询得大小的单位，查阅以下资料了解

- ## Linux free命令详解 [1] [2] [3]

  free命令是Linux系统中用于显示当前系统的内存使用情况的工具，包括物理内存、交换内存（swap）以及内核缓冲区内存。它能够提供关于系统内存使用情况的快照，帮助用户了解系统资源的分配。

  ### free命令的基本使用

  free命令的基本语法非常简单，只需要在终端中输入`free`即可。默认情况下，free命令会以KB为单位显示内存的使用情况，包括以下几个关键指标 [1] [2]：

  - **total**: 显示系统总的可用物理内存和交换空间大小。
  - **used**: 显示已经被使用的物理内存和交换空间。
  - **free**: 显示还有多少物理内存和交换空间可用。
  - **shared**: 显示被共享使用的物理内存大小。
  - **buffers/cached**: 显示被buffer和cache使用的物理内存大小。
  - **available**: 显示还可以被应用程序使用的物理内存大小。

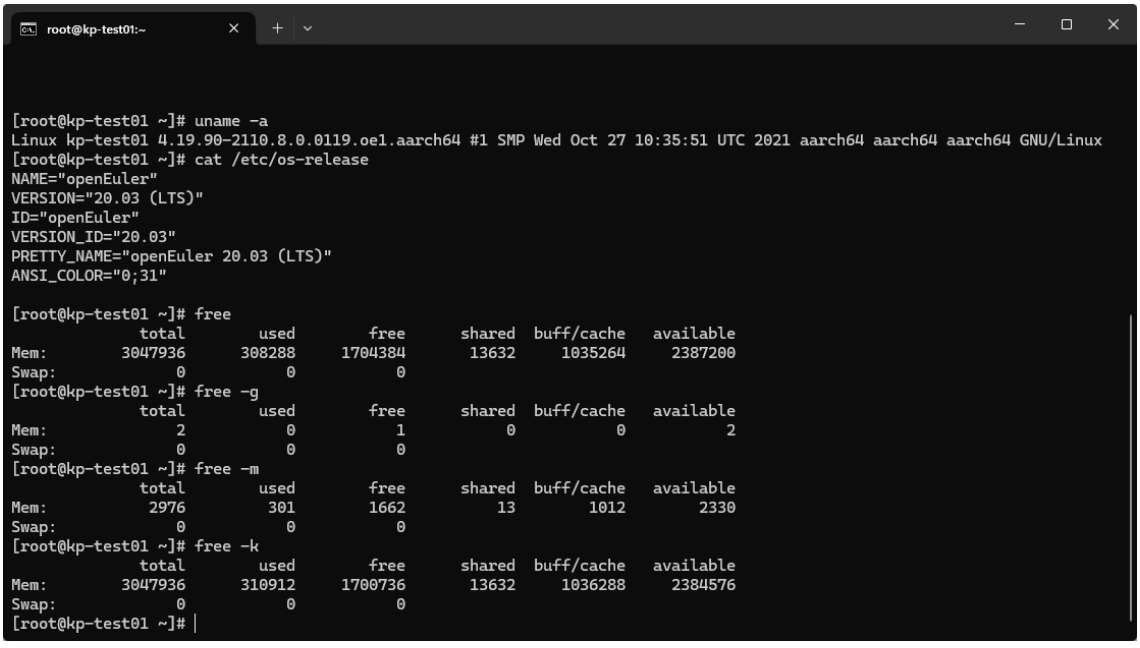
  ### free命令的选项

  free命令提供了多个选项，可以更详细地控制输出的信息。以下是一些常用的选项 [2] [3]：

  - `-b`: 以Byte为单位显示内存使用情况。
  - `-k`: 以KB为单位显示内存使用情况。
  - `-m`: 以MB为单位显示内存使用情况。
  - `-g`: 以GB为单位显示内存使用情况。
  - `-h`: 以适于人类可读的方式显示内存使用情况。
  - `-t`: 显示内存总和列。
  - `-s <间隔秒数>`: 持续观察内存使用状况。
  - `-V`: 显示版本信息。

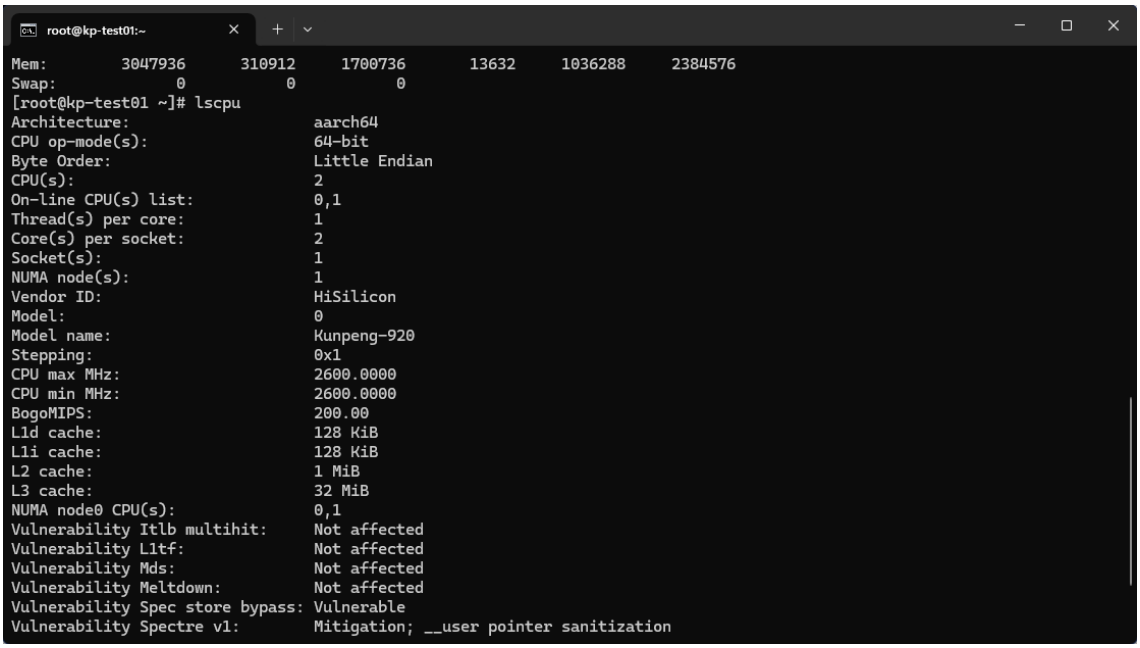
  例如，使用`free -h`命令可以以易于阅读的格式显示内存使用情况，其中`-h`选项会自动为数字添加合适的单位，如GB或MB。

可知mem为物理内存，swap为交换内存，并可以用不同参数显示输出信息，以下是用GB、MB、KB显示的信息



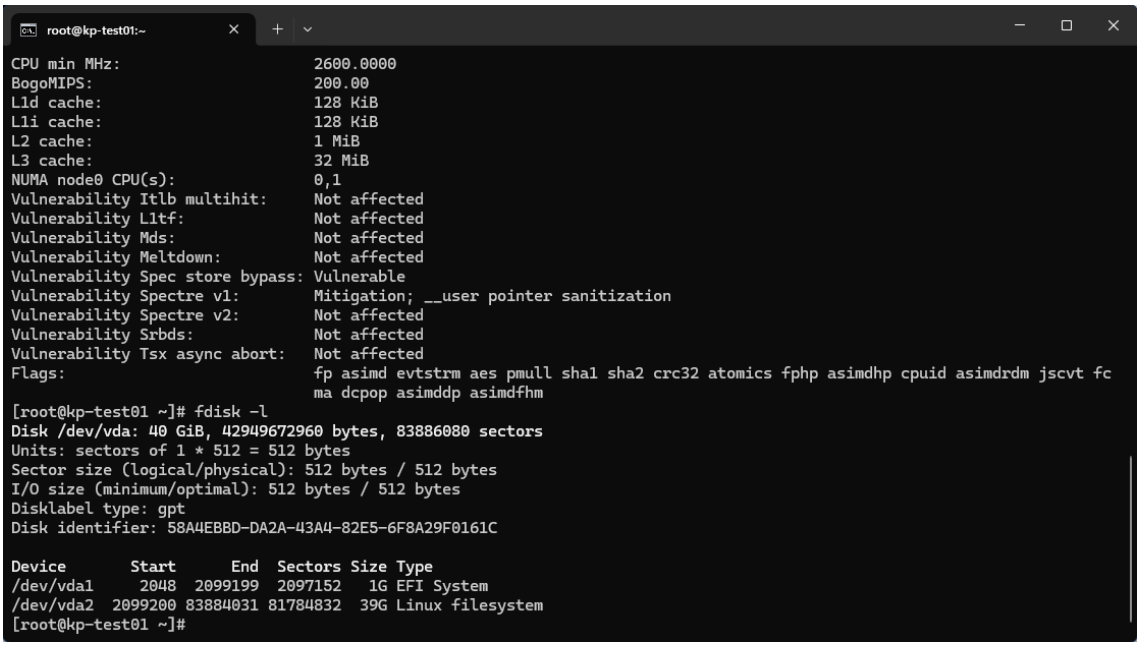
可见物理内存为2976MB，且free命令无法显示小数信息。

4. lscpu：查看cpu信息



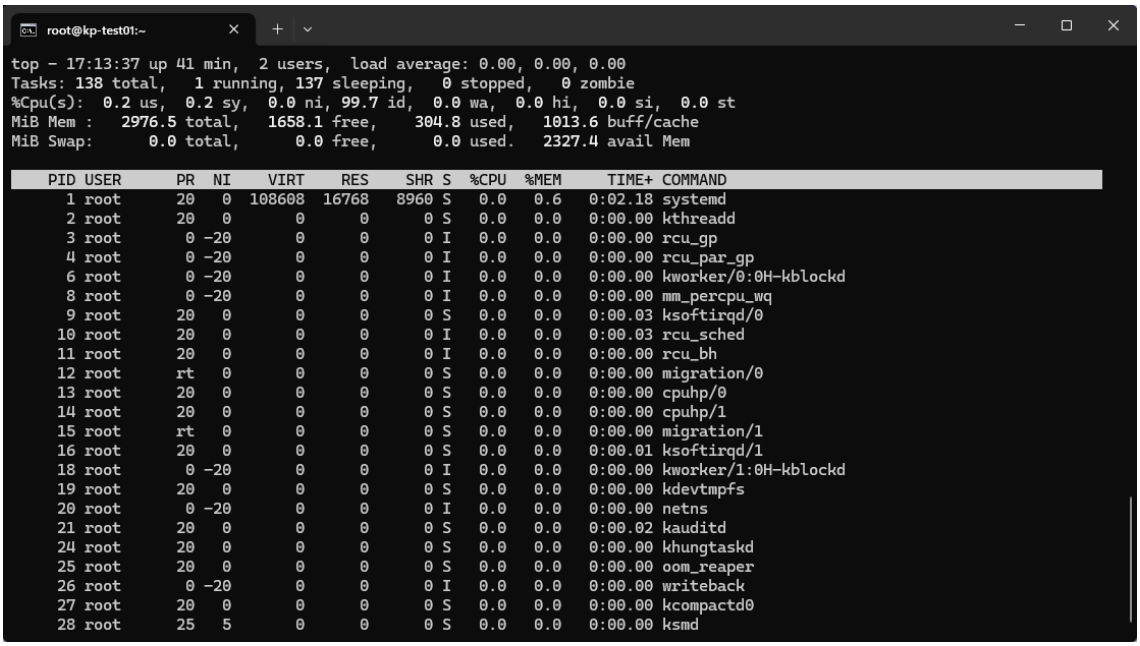
与配置时所用的cpu核数一致。

5. fdisk -l：查看磁盘信息

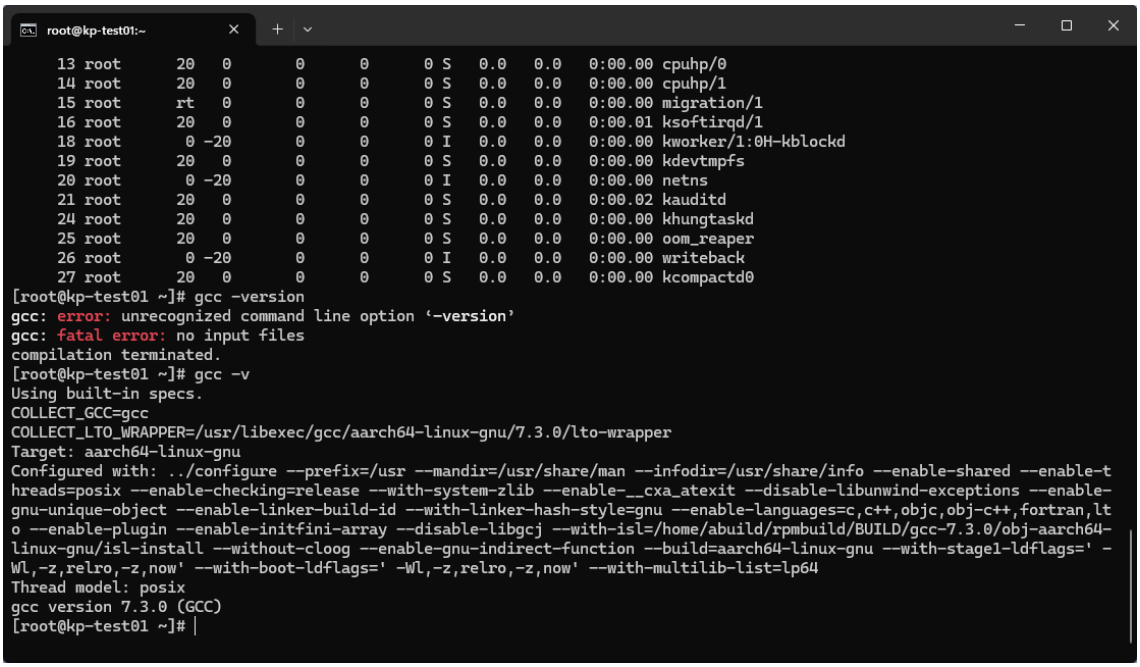


可见总共40GB空间，其中1G为系统启动区域所用，文件格式为EFI，另外39G为正常的Linux的文件系统。

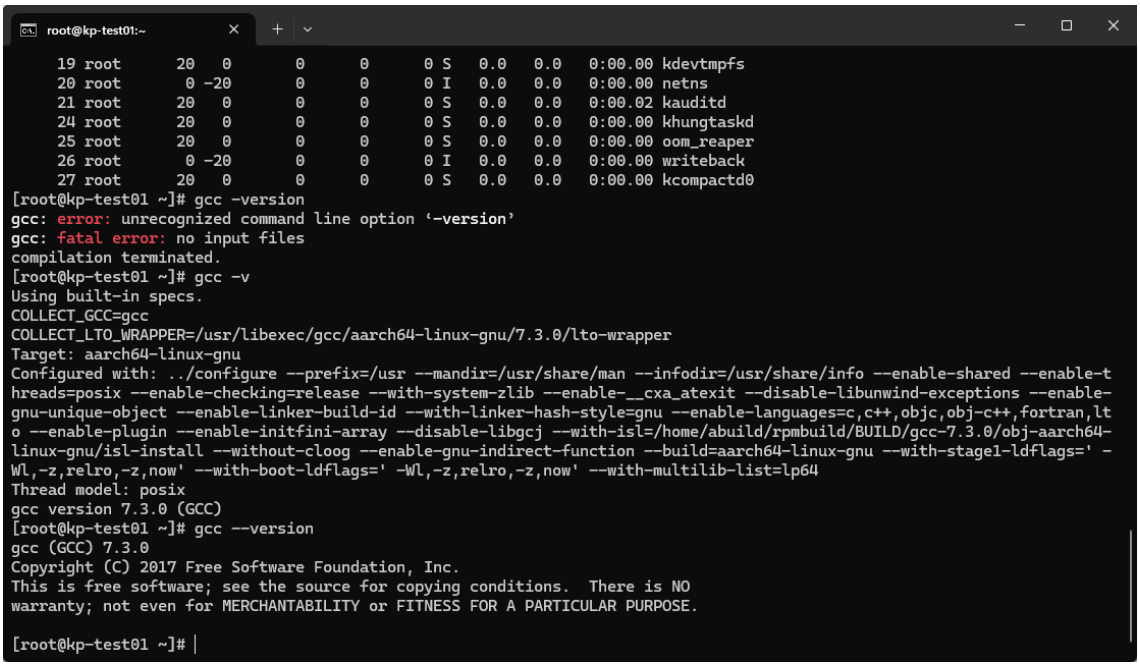
6. top：查看系统资源使用信息，如下图：



7. gcc -version：查看gcc编译器版本



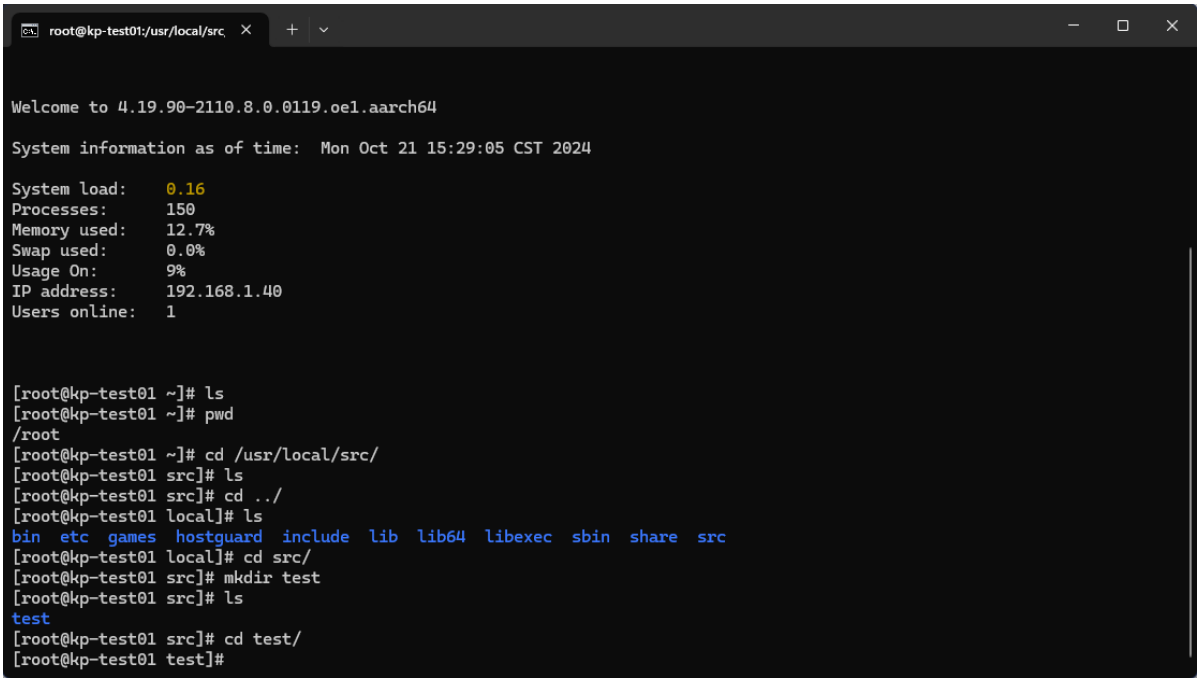
可见实验指导书有误，查询时需要使用**gcc -v**而不是gcc -version



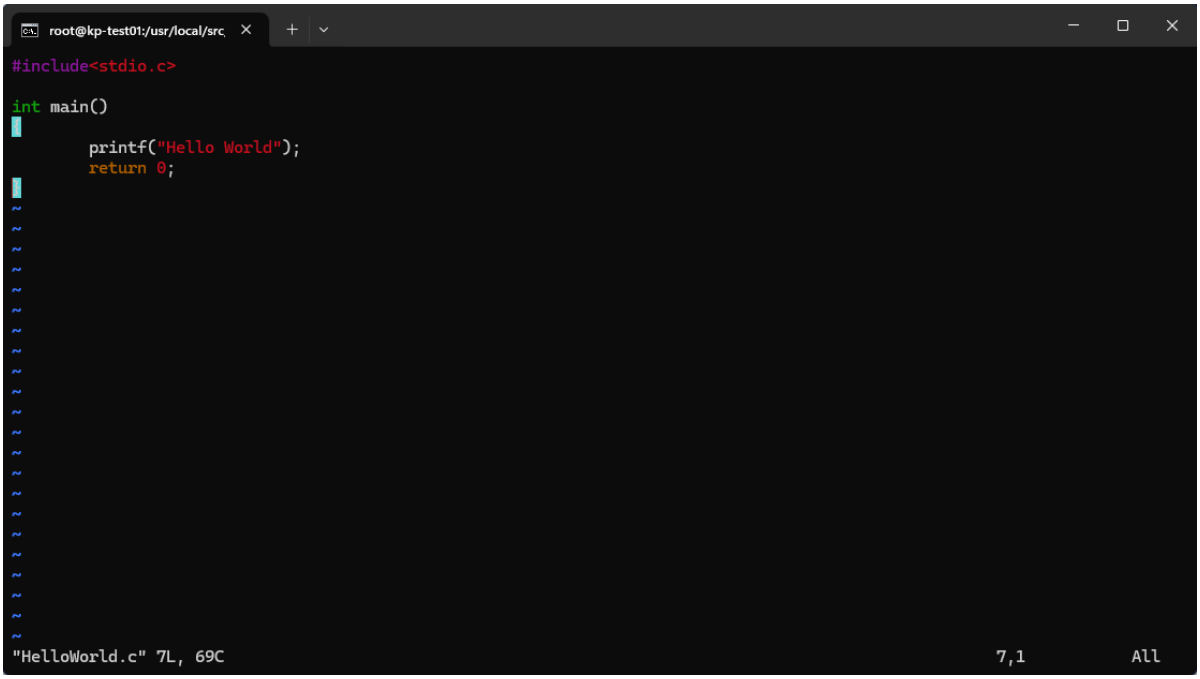
或者可以使用**gcc --version**

使用当前环境编译并运行简单程序

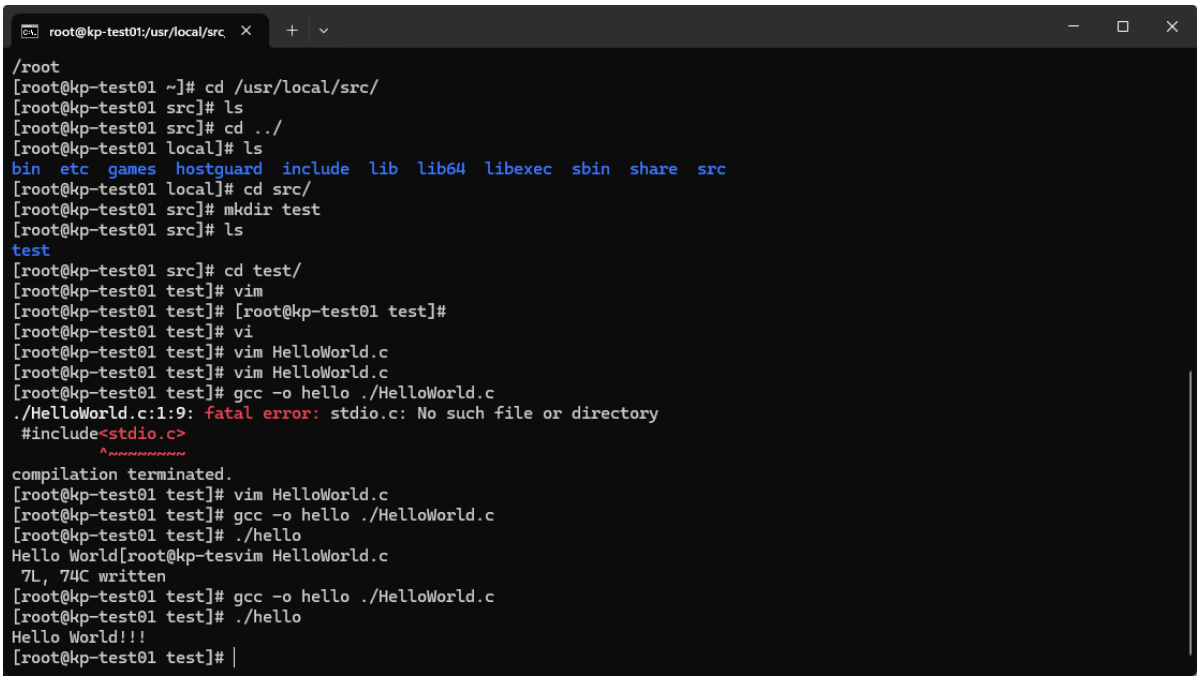
使用cmd连接ECS云服务器，使用cd命令移动到对应文件夹下并创建test文件夹：



使用vim创建HelloWorld.c文件并编写代码：



使用命令“gcc -o hello HelloWorld.c”编译运行



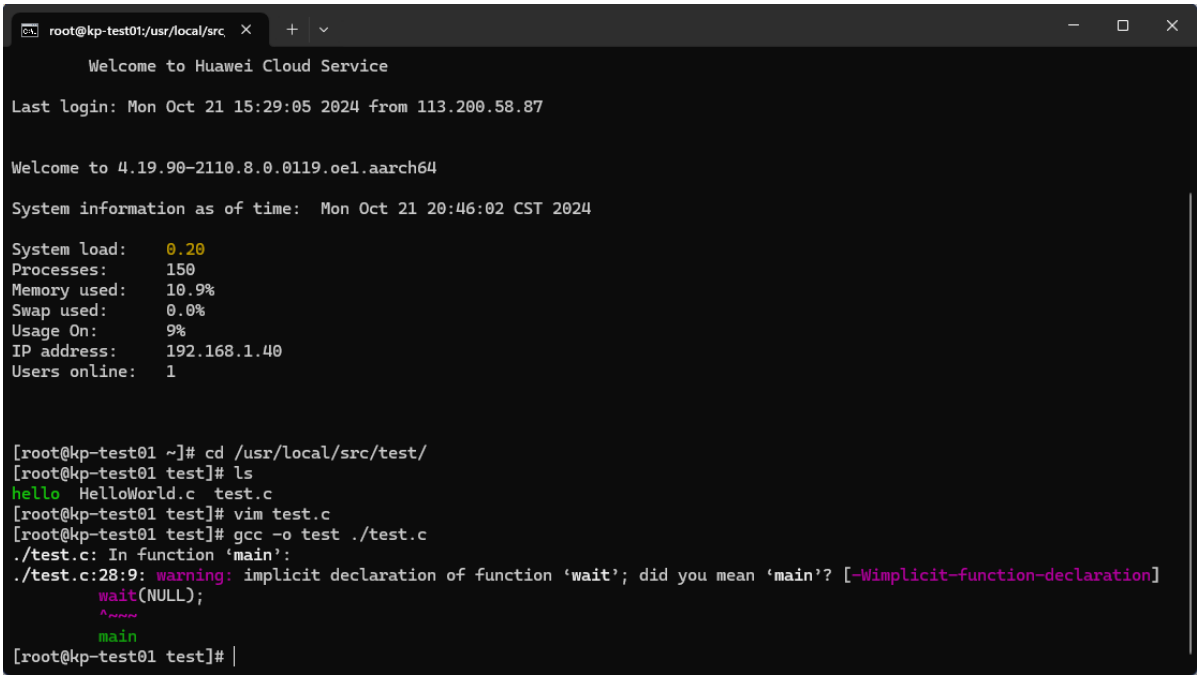
因输入失误出错，改正后正常。

进程相关实验

实验步骤一、二

本地使用vscode将图1-1程序写为test.c文件，并通过winscp传输到/usr/local/src/test/目录下，如图：

编译时出现问题：

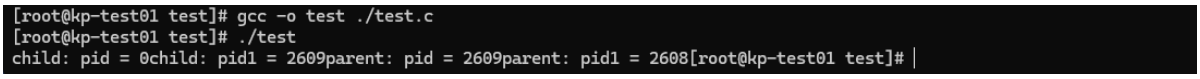


查阅资料知，在使用wait函数时需要添加头文件

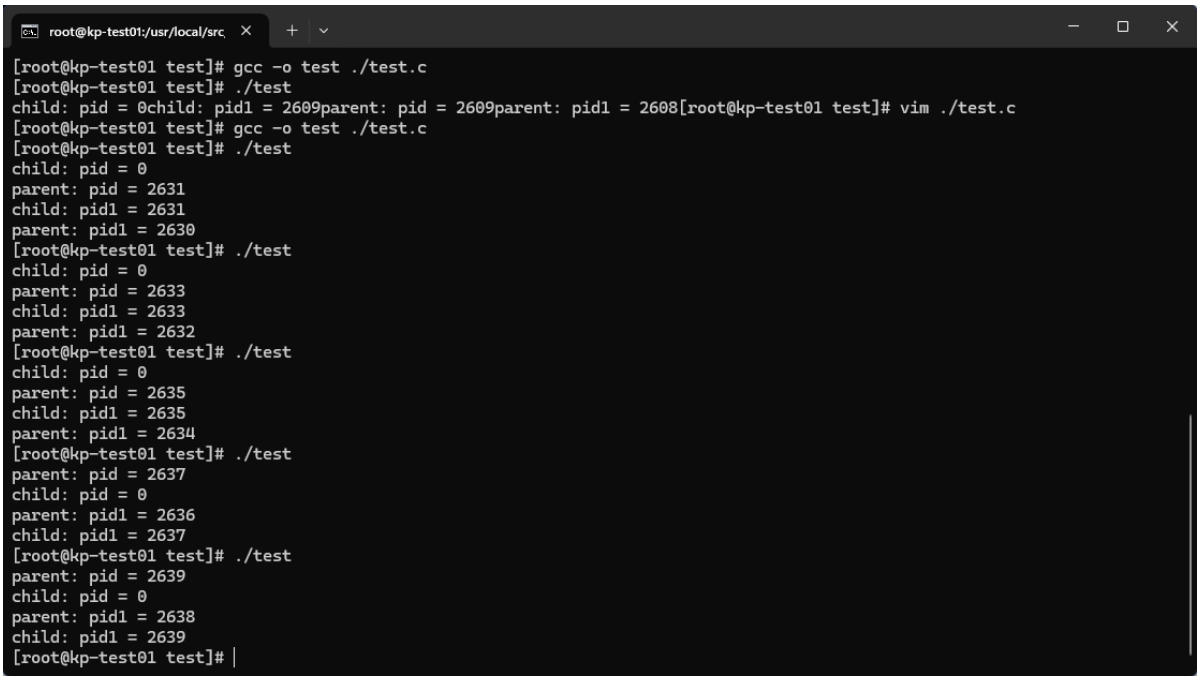
```
#include<sys/wait.h>
```

- 了解C语言中的wait()系统调用-CSDN博客

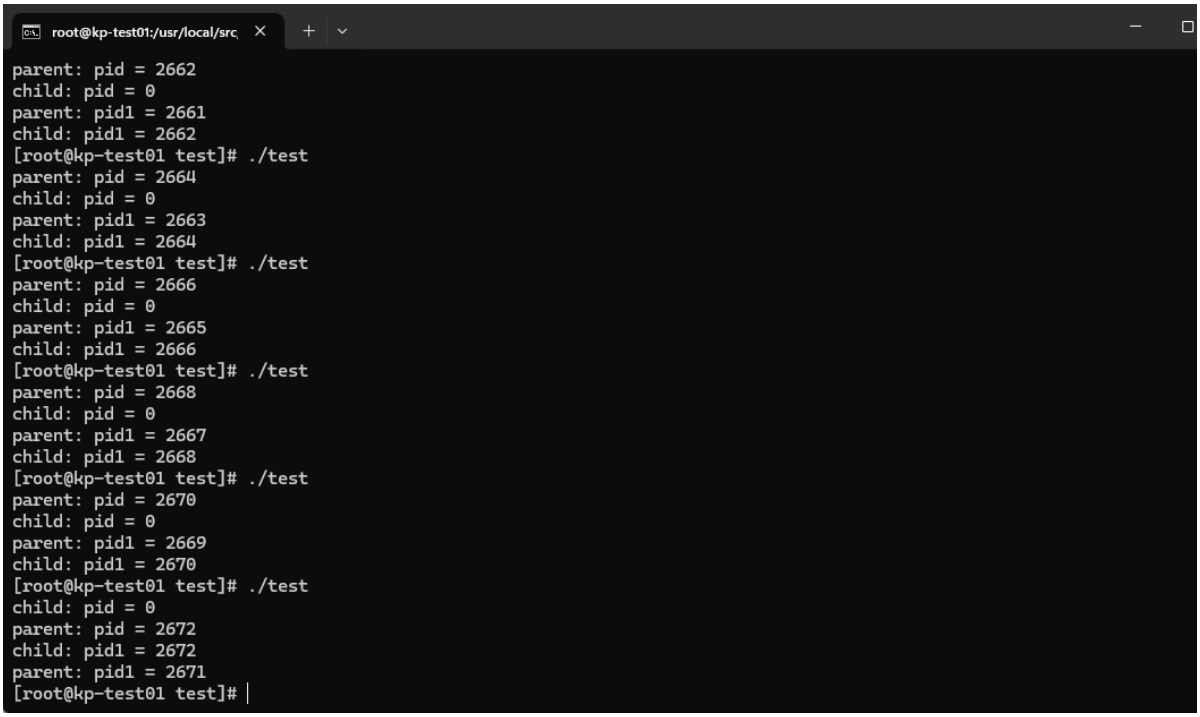
成功消除警告，运行结果如下：



为代码输出添加换行符后，多次运行结果如下：

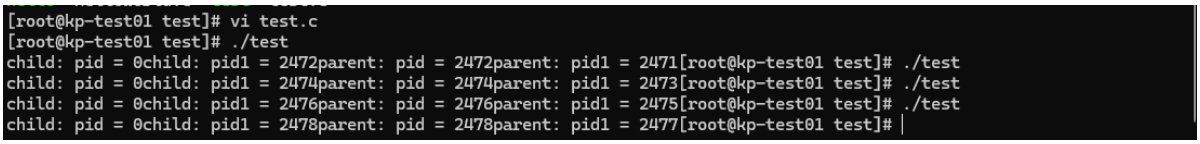


去掉wait函数后，运行结果如下：

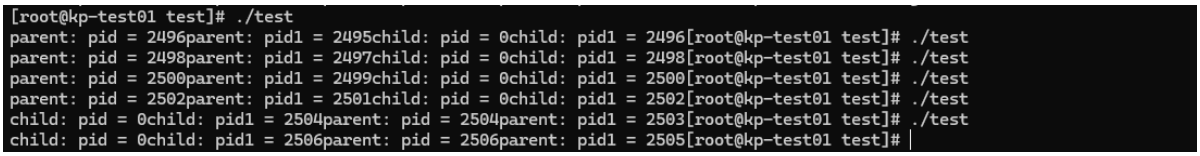


观察到删除前和删除后无明显区别，考虑到添加换行符后，标准输出缓冲区会在每次输出后自动刷新，不利于观察代码的执行效果，将换行符删去后重新执行：

去掉wait函数前



去掉wait函数后



理论分析

可见，无论去不去掉wait函数，子进程中pid值总为0，pid1值总为子进程pid；父进程中，pid值为子进程pid，pid1为父进程pid。考虑到对于fork函数，当当前进程为子进程时，返回值为0，当前进程为父进程时，返回值为子进程pid，同时，对于getpid函数，获取的总为当前进程的pid，输出符合编程逻辑。

而对于去掉wait函数前，父进程结束前，需要等待子进程结束并回收，故父进程的输出总在子进程的输出之后，与实际相符合。对于去掉wait函数之后，无论父子进程，都有可能先于对方结束，故输出既有可能是父进程在前，也有可能是子进程在前。与实际输出结果相符合。

实验步骤三

增加全局变量，修改代码如下：

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

int value = 0;

int main()
{
    pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0)
    {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0)
    {
        pid1 = getpid();
        printf("child: pid = %d", pid);
        printf("child: pid1 = %d", pid1);
        value++;
        printf("child value: %d", value);
    }
    else
    {
        pid1 = getpid();
        printf("parent: pid = %d", pid);
        printf("parent: pid1 = %d", pid1);
        wait(NULL);
        value--;
        printf("parent value: %d", value);
    }

    return 0;
}
```

定义全局变量value，赋予其初始值为0。在子进程中，让value值加一并输出，在父进程中，让value值减一并输出。

运行结果如下：

```
[root@kp-test01 test]# gcc -o test ./test.c
[root@kp-test01 test]# ./test
child: pid = 0child: pid1 = 2611child value: 1parent: pid = 2611parent: pid1 = 2610parent value: -1[root@kp-test01 test]# ./test
child: pid = 0child: pid1 = 2613child value: 1parent: pid = 2613parent: pid1 = 2612parent value: -1[root@kp-test01 test]#
```

可见，在子进程中，value输出值为1，父进程中，value输出值为-1。可见父子进程中，全局变量不共享，全局变量在fork之后，会被复制到子进程中。

步骤四

在return前增加对全局变量的修改并输出，代码如下：

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

int value = 0;

int main()
{
    pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0)
    {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0)
    {
        pid1 = getpid();
        printf("child: pid = %d", pid);
        printf("child: pid1 = %d", pid1);
        value++;
        printf("child value: %d", value);
        printf("child *value: %d", &value);
    }
    else
    {
        pid1 = getpid();
        printf("parent: pid = %d", pid);
        printf("parent: pid1 = %d", pid1);
        wait(NULL);
        value--;
        printf("parent value: %d", value);
        printf("parent *value: %d", &value);
    }

    value += 17;
    printf("before return value: %d", value);
    printf("before return *value: %d", &value);

    return 0;
}
```

在父子进程中，增加输出value地址，同时在return前，将value值加17并输出结果与value地址。运行结果如下：

```
[root@kp-test01 test]# vi test.c
[root@kp-test01 test]# gcc -o test ./test.c
[root@kp-test01 test]# ./test
child: pid = 0child: pid1 = 2651child value: 1child *value: 4325468before return value: 18before return *value: 4325468parent: pid = 2651parent: pid1 = 2650parent value: -1parent *value: 4325468before return v
alue: 16before return *value: 4325468[root@kp-test01 test]#
```

可见在子进程中，return前输出的value值为18，而在父进程中，return前输出的value值为16。进一步证明了父子进程并不共享全局变量，子进程中全局变量是父进程中全局变量的复制。

同时可见父子进程中，value的地址都一样，但这并不说明这是同一个变量，而是因为fork后，子进程会复制父进程的整个地址空间，而输出的value的地址仅是value的虚拟地址，只表明value在父子进程中在各自的虚拟地址空间中存储在相同的地址处。

步骤五

编写system_call.c及修改test.c代码如下：

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

int main()
{
    pid_t pid = getpid();
    printf("systemcall pid = %d\n", pid);
    return 0;
}
```

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

int main()
{
    pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0)
    {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0)
    {
        pid1 = getpid();
        printf("child: pid1 = %d\n", pid1);
        system("./system_call");
    }
    else
    {
        pid1 = getpid();
        printf("parent: pid1 = %d\n", pid1);
        wait(NULL);
    }

    return 0;
}
```

如图，在system_call.c中，代码获取当前进程的pid并输出，在test.c文件中，输出父子进程的pid，并在子进程中使用system函数调用编译好的system_call文件。运行结果如下图所示：

```
[root@kp-test01 test]# gcc -o system_call ./system_call.c
[root@kp-test01 test]# gcc -o test ./test.c
./test.c: In function ‘main’:
./test.c:22:9: warning: implicit declaration of function ‘system’ [-Wimplicit-function-declaration]
         system("./system_call");
         ^~~~~~
[root@kp-test01 test]# vi test.c
[root@kp-test01 test]# gcc -o test ./test.c
[root@kp-test01 test]# ./test
parent: pid1 = 2766
child: pid1 = 2767
systemcall pid = 2768
[root@kp-test01 test]#
```

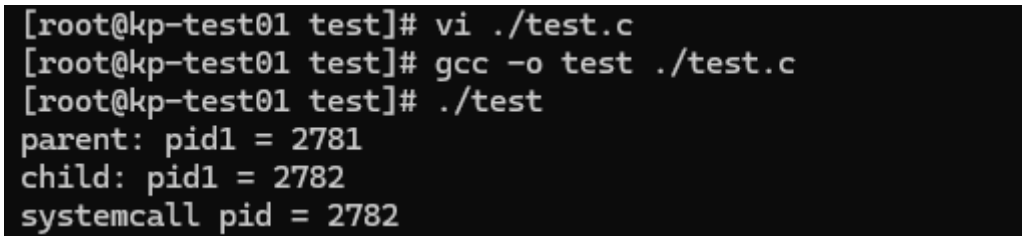
第一次运行报错原因在于没有添加#include <stdlib.h>库，添加后问题解决，成功编译。

在运行结果中，父进程pid为2766，子进程pid为2767，调用system_call程序运行后，输出的pid为2768。

修改为使用exec族函数调用system_call程序，修改代码如下：

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

int main()
{
    pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0)
    {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0)
    {
        pid1 = getpid();
        printf("child: pid1 = %d\n", pid1);
        if (execl("./system_call", "system_call", NULL) == -1)
        {
            perror("execl failed");
            exit(1); // 出现错误时退出子进程
        }
    }
    else
    {
        pid1 = getpid();
        printf("parent: pid1 = %d\n", pid1);
        wait(NULL);
    }

    return 0;
}
```

选择使用execl函数，当其返回值为-1表示调用失败时，使用exit函数退出子进程。运行结果如下：



如图可见，父进程pid为2781，子进程pid为2782，调用system_call程序输出pid为2782。

分析可知，使用system函数调用时，程序运行在新的进程下，使用exec族函数调用时，程序运行在子进程下。

查资料可知，`system()` 函数和 `exec` 函数族（如 `execl()`、`execvp()` 等）都可以用来在 C 程序中调用外部程序。`system()` 函数调用外部程序时，会启动一个新的子 shell 来执行命令。它相当于在命令行中手动输入命令。`system()` 函数会阻塞调用进程，直到外部程序执行完毕。当 `system()` 返回时，调用进程仍然保持原样，不会被替换。

而对于`exec`函数族，直接用外部程序替换当前进程，并不创建新的 shell。调用 `exec` 后，原进程的代码和数据都会被替换为外部程序，除非调用 `exec` 失败，程序控制永远不会返回到调用点。`exec` 不会返回，除非调用失败（即，当前进程被替换为外部程序的进程映像）。`exec` 系列函数直接运行指定的程序，而无需启动一个新的 `shell`。

修改代码，在调用外部程序后继续打印子进程pid，如下：

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

int main()
{
    pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0)
    {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0)
    {
        pid1 = getpid();
        printf("child: pid1 = %d\n", pid1);
        if (execl("./system_call", "system_call", NULL) == -1)
        {
            perror("execl failed");
            exit(1); // 出现错误时退出子进程
        }
        printf("child: pid2= %d\n", pid1);
    }
    else
    {
        pid1 = getpid();
        printf("parent: pid1 = %d\n", pid1);
        wait(NULL);
    }

    return 0;
}
```

运行结果如下：

```
[root@kp-test01 test]# vi ./test.c
[root@kp-test01 test]# gcc -o test ./test.c
[root@kp-test01 test]# ./test
parent: pid1 = 2851
child: pid1 = 2852
systemcall pid = 2853
child: pid2= 2852
[root@kp-test01 test]# gcc -o test ./test.c
[root@kp-test01 test]# vi ./test.c
[root@kp-test01 test]# gcc -o test ./test.c
[root@kp-test01 test]# ./test
parent: pid1 = 2865
child: pid1 = 2866
systemcall pid = 2866
```

第一次使用`system`，第二次使用`exec`族函数。可见在使用`exec`函数后，不会在执行pid2的打印代码，与预期符合。

线程相关实验

步骤一

创建两个子线程，分别对全局变量counter进行100000次加减100操作，代码如下：

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>


int counter = 0;

void *thread1_f()
{
  for(int i=0;i<100000;i++)
  {
    counter += 100;
  }
  pthread_exit(NULL);
}

void *thread2_f()
{
  for(int i=0;i<100000;i++)
  {
    counter -= 100;
  }
  pthread_exit(NULL);
}

int main()
{
    pthread_t thread1, thread2;
    if(!pthread_create(&thread1, NULL, thread1_f, NULL))
    {
        printf("threaad1 create success\n");
    }

    if(!pthread_create(&thread2, NULL, thread2_f, NULL))
    {
        printf("threaad2 create success\n");
    }

    printf("variable result: %d", counter);

}
```

之后使用gcc编译运行，在此过程中报错，提示pthread相关函数没有定义，查阅资料得知，在编译时需要链接`pthread`库，资料来源如下：

- https://blog.csdn.net/jay_zzs/article/details/106380659

修改后报错消失，但是观察后发现，运行得到的结果中，result全为大于零的正数，考虑到可能是代码中没有及时在主线程回收子线程，导致子线程提前结束导致，修改代码，在主线程返回前使用`pthread_join`回收子线程。

```
[root@kp-test01 exper1]# vim 1-6.c
[root@kp-test01 exper1]# gcc -o 1-6 ./1-6.c
/usr/bin/ld: /tmp/cc42jQnl.o: in function `main':
1-6.c:(.text+0xc8): undefined reference to `pthread_create'
/usr/bin/ld: 1-6.c:(.text+0xf8): undefined reference to `pthread_create'
collect2: error: ld returned 1 exit status
[root@kp-test01 exper1]# gcc -pthread -o 1-6 ./1-6.c
[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 666900[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 668700[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 743400[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 662500[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 680100[root@kp-test01 exper1]# ./1-6
```

修改后代码与第二次运行结果如下图：

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>


int counter = 0;

void *thread1_f()
{
  for(int i=0;i<100000;i++)
  {
    counter += 100;
  }
  pthread_exit(NULL);
}

void *thread2_f()
{
  for(int i=0;i<100000;i++)
  {
    counter -= 100;
  }
  pthread_exit(NULL);
}

int main()
{
    pthread_t thread1, thread2;
    if(!pthread_create(&thread1, NULL, thread1_f, NULL))
    {
        printf("threaad1 create success\n");
    }
```

```
    if(!pthread_create(&thread2, NULL, thread2_f, NULL))
    {
        printf("threaad2 create success\n");
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("variable result: %d", counter);
    return 0;

}
```

```
[root@kp-test01 exper1]# gcc -pthread -o 1-6 ./1-6.c
[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: -720400[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 414900[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: -10500[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: -613600[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: -423200[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 173100[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: -289400[root@kp-test01 exper1]#
```

可见每一次运行结果都不一样，既有正数也有负数。在原代码中，两个子线程都需要访问`counter`全局变量，但由于计算机中访问时需要先复制存储器中值，修改后在写回，可能导致两个线程在同时访问时，取得的值都是另一个线程还没有修改的，导致其中某一个计算结果无效。

步骤二

为使得两个子进程可以互斥的访问`counter`全局变量，使用信号量实现互斥锁，`修改``1-6.c`代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>


int counter = 0;
sem_t mutex;
```

```c
void *thread1_f()
{
  for(int i=0;i<100000;i++)
  {
    sem_wait(&mutex);
    counter += 100;
    sem_post(&mutex);
  }
  pthread_exit(NULL);
}

void *thread2_f()
{
  for(int i=0;i<100000;i++)
  {
    sem_wait(&mutex);
    counter -= 100;
    sem_post(&mutex);
  }
  pthread_exit(NULL);
}

int main()
{

    pthread_t thread1, thread2;
    sem_init(&mutex, 0, 1);
    if(!pthread_create(&thread1, NULL, thread1_f, NULL))
    {
        printf("threaad1 create success\n");
    }

    if(!pthread_create(&thread2, NULL, thread2_f, NULL))
    {
        printf("threaad2 create success\n");
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    sem_destroy(&mutex);
    printf("variable result: %d", counter);
    return 0;
}
```

编译后运行结果如下：

```
variable result: -289400[root@kp-test01 exper1]# gcc -pthread -o 1-6 ./1-6.c
[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 0[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 0[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 0[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 0[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 0[root@kp-test01 exper1]# ./1-6
threaad1 create success
threaad2 create success
variable result: 0[root@kp-test01 exper1]#
```

可见成功实现互斥锁，最终全局变量的结果为0,符合预期。分析原因可知，使用信号量时，在`main`函数起始，将其初始化为1,即只允许同时一个线程进行访问，其他线程在访问时将被阻塞，只有在该线程使用完`counter`全局变量后，才能够继续访问，实现互斥访问。

步骤三

分别使用system函数以及exec族函数调用system_call程序，修改代码如下：

```c
/* 1-8.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/types.h>


void *thread1_f()
{
  pid_t tid = syscall(SYS_gettid);
  pid_t pid = getpid();
  printf("thread1 tid: %d, pid: %d\n", tid, pid);
  system("./system_call");
  printf("thread1 systemcall return\n");
  pthread_exit(NULL);
}

void *thread2_f()
{
  pid_t tid = syscall(SYS_gettid);
  pid_t pid = getpid();
  printf("thread2 tid: %d, pid: %d\n", tid, pid);
  system("./system_call");
  printf("thread2 systemcall return\n");
  pthread_exit(NULL);
}
```

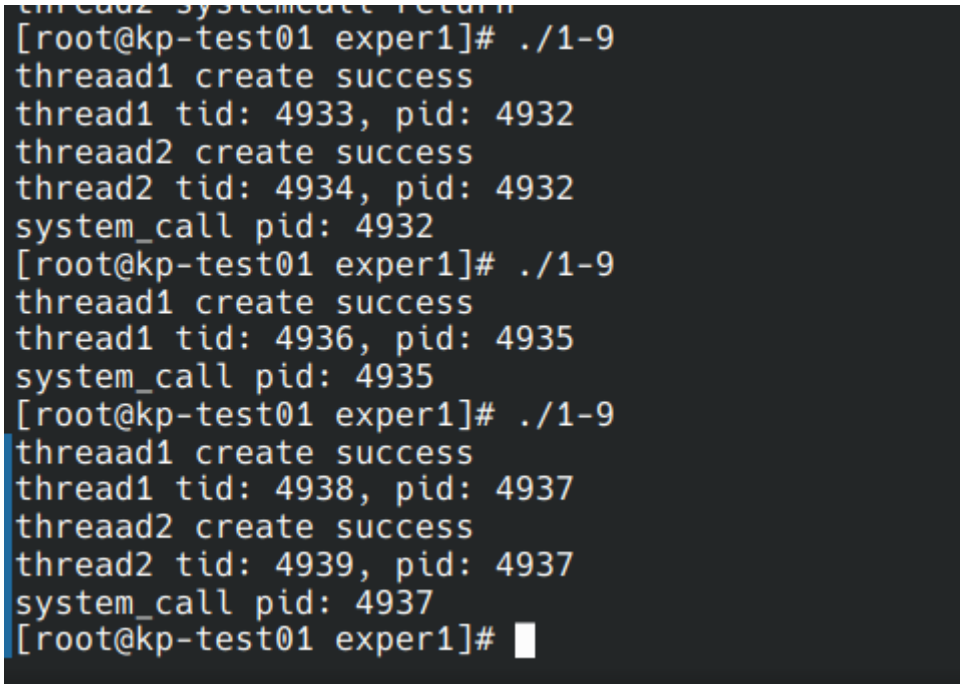
```c
int main()
{

    pthread_t thread1, thread2;
    if(!pthread_create(&thread1, NULL, thread1_f, NULL))
    {
        printf("threaad1 create success\n");
    }

    if(!pthread_create(&thread2, NULL, thread2_f, NULL))
    {
        printf("threaad2 create success\n");
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```

```c
/* 1-9.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/types.h>


void *thread1_f()
{
  pid_t tid = syscall(SYS_gettid);
  pid_t pid = getpid();
  printf("thread1 tid: %d, pid: %d\n", tid, pid);
  if (execl("./system_call", "system_call", NULL) == -1)
  {
    perror("execl failed\n");
    exit(1); // 出现错误时退出子进程
  }
  printf("thread1 systemcall return\n");
  pthread_exit(NULL);
}

void *thread2_f()
{
  pid_t tid = syscall(SYS_gettid);
  pid_t pid = getpid();
  printf("thread2 tid: %d, pid: %d\n", tid, pid);
  if (execl("./system_call", "system_call", NULL) == -1)
  {
    perror("execl failed\n");
    exit(1); // 出现错误时退出子进程
  }
  printf("thread2 systemcall return\n");
  pthread_exit(NULL);
```

```
}

int main()
{

    pthread_t thread1, thread2;
    if(!pthread_create(&thread1, NULL, thread1_f, NULL))
    {
        printf("threaad1 create success\n");
    }

    if(!pthread_create(&thread2, NULL, thread2_f, NULL))
    {
        printf("threaad2 create success\n");
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```

分别编译后运行有如下结果：

```
[root@kp-test01 exper1]# ./1-8
threaad1 create success
thread1 tid: 4918, pid: 4917
threaad2 create success
thread2 tid: 4919, pid: 4917
system_call pid: 4921
system_call pid: 4920
thread1 systemcall return
thread2 systemcall return
[root@kp-test01 exper1]# ./1-8
threaad1 create success
thread1 tid: 4923, pid: 4922
threaad2 create success
thread2 tid: 4924, pid: 4922
system_call pid: 4926
system_call pid: 4925
thread2 systemcall return
thread1 systemcall return
[root@kp-test01 exper1]# ./1-8
threaad1 create success
thread1 tid: 4928, pid: 4927
threaad2 create success
thread2 tid: 4929, pid: 4927
system_call pid: 4930
system_call pid: 4931
thread1 systemcall return
thread2 systemcall return
[root@kp-test01 exper1]#
```



分析可见，无论是对system调用还是exec族函数调用，子线程的pid都相同，tid都不相同，而两部分代码不同主要体现在两个方面：

- 对于system调用，两个子线程都分别调用`system_call`程序。而对于exec族函数调用，只成功调用了一次system_call程序，且两个子线程都没有返回“systemcall return”字符串。
- 对于system调用，调用程序的进程与子线程的进程不同。而对于exec族函数调用，调用system_call的进程pid与子线程的pid一致。

查阅相关资料可知，`system`函数会创建一个新的子进程，并在该进程中执行指定的命令，调用完成后返回到父进程。具体来说，它先通过`fork`创建一个子进程，然后在子进程中调用`/bin/sh -c`来执行传入的命令字符串。而`exec`不创建新的进程，而是在当前进程中执行新程序。这意味着，一旦调用了`exec`函数，当前进程的代码将被新程序的代码替代，当前进程不再返回到原来的代码。

自旋锁实验

步骤一

补充后自旋锁代码如下：

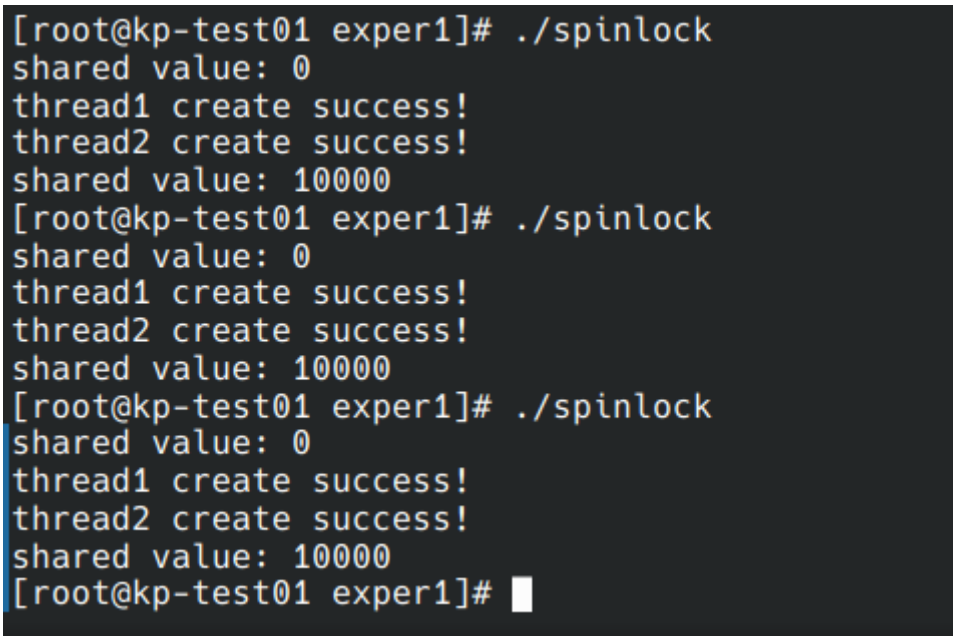
```
/**
*spinlock.c
*in xjtu
*2023.8
*/
#include <stdio.h>
#include <pthread.h>
// 定义自旋锁结构体
typedef struct {
int flag;
} spinlock_t;
// 初始化自旋锁
void spinlock_init(spinlock_t *lock) {
lock->flag = 0;
}
```

```
// 获取自旋锁
void spinlock_lock(spinlock_t *lock) {
while (__sync_lock_test_and_set(&lock->flag, 1)) {
// 自旋等待
}
}
// 释放自旋锁
void spinlock_unlock(spinlock_t *lock) {
__sync_lock_release(&lock->flag);
}
// 共享变量
int shared_value = 0;
// 线程函数
void *thread_function(void *arg) {
spinlock_t *lock = (spinlock_t *)arg;
for (int i = 0; i < 5000; ++i) {
spinlock_lock(lock);
shared_value++;
spinlock_unlock(lock);
}
return NULL;
}
int main() {
pthread_t thread1, thread2;
spinlock_t lock;
// 输出共享变量的值
printf("shared value: %d\n", shared_value);
// 初始化自旋锁
spinlock_init(&lock);
// 创建两个线程
if(!pthread_create(&thread1, NULL, thread_function, &lock))
{
    printf("thread1 create success!\n");
}
if(!pthread_create(&thread2, NULL, thread_function, &lock))
{
    printf("thread2 create success!\n");
}
// 等待线程结束
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
// 输出共享变量的值
printf("shared value: %d\n", shared_value);

return 0;
}
```

如代码所示，创建两个线程，分别在子线程创建前与结束后输出共享的shared_value值，同时两个子线程运行线程函数，分别对`shared_value`执行5000次加一操作。

步骤二

将`spinlock.c`编译后运行有：



可见在运行结果，第一次输出`shared_value`值为0,第二次输出`shared_value`值为10000。表示两个子线程分别正确的对`shared_value`执行了加操作，成功利用自旋锁完成了对两个线程访问共享变量的互斥操作。