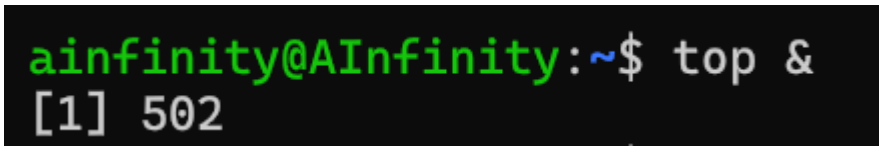
俞楚凡 24302010004 ICS 5.26 lab

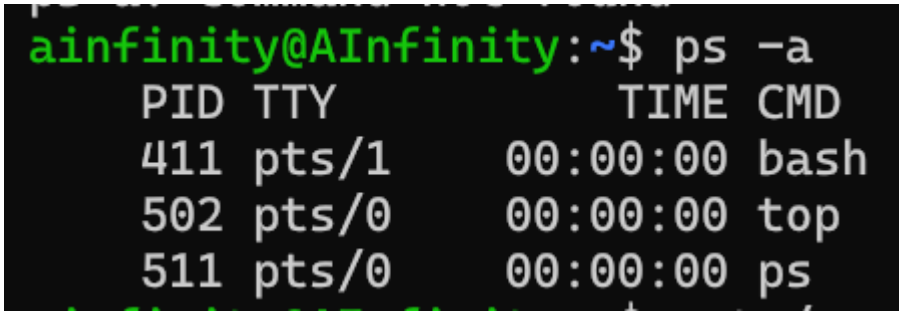
接下来依次叙述实验思路。

1.1.1

我们首先执行 top &，在后台运行 top，便于我们查看后续操作。



随后执行 ps -a，获取 top 的 pid。



随后 执行 cat /proc/502/maps，得到 tops 的内存映射信息。

```
ainfinity@AInfinity:~$ cat /proc/502/maps
564a4e093000-564a4e096000 r--p 00000000 08:20 2104                       /usr/bin/top
564a4e096000-564a4e0ab000 r-xp 00003000 08:20 2104                       /usr/bin/top
564a4e0ab000-564a4e0b2000 r--p 00018000 08:20 2104                       /usr/bin/top
564a4e0b2000-564a4e0b3000 r--p 0001e000 08:20 2104                       /usr/bin/top
564a4e0b3000-564a4e0b5000 rw-p 0001f000 08:20 2104                       /usr/bin/top
564a4e0b5000-564a4e0e1000 rw-p 00000000 00:00 0
564a7dd5c000-564a7dd9f000 rw-p 00000000 00:00 0                          [heap]
7fa893b4b000-7fa893ba4000 r--p 00000000 08:20 12171                      /usr/lib/locale/C.utf8/LC_CTYPE
7fa893ba4000-7fa893ba5000 r--p 00000000 08:20 12177                      /usr/lib/locale/C.utf8/LC_NUMERIC
7fa893ba5000-7fa893ba6000 r--p 00000000 08:20 12180                      /usr/lib/locale/C.utf8/LC_TIME
7fa893ba6000-7fa893ba7000 r--p 00000000 08:20 12170                      /usr/lib/locale/C.utf8/LC_COLLATE
7fa893ba7000-7fa893ba8000 r--p 00000000 08:20 12175                      /usr/lib/locale/C.utf8/LC_MONETARY
7fa893ba8000-7fa893ba9000 r--p 00000000 08:20 12174                      /usr/lib/locale/C.utf8/LC_MESSAGES/SYS_LC_MESSAGES
7fa893ba9000-7fa893bb0000 r--s 00000000 08:20 46697                      /usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache
7fa893bb0000-7fa893bb5000 rw-p 00000000 00:00 0
7fa893bb5000-7fa893bb9000 r--p 00000000 08:20 12817                      /usr/lib/x86_64-linux-gnu/libgpg-error.so.0.34.0
7fa893bb9000-7fa893bcf000 r-xp 00004000 08:20 12817                      /usr/lib/x86_64-linux-gnu/libgpg-error.so.0.34.0
7fa893bcf000-7fa893bd8000 r--p 0001a000 08:20 12817                      /usr/lib/x86_64-linux-gnu/libgpg-error.so.0.34.0
7fa893bd8000-7fa893bd9000 r--p 00023000 08:20 12817                      /usr/lib/x86_64-linux-gnu/libgpg-error.so.0.34.0
7fa893bd9000-7fa893bda000 rw-p 00024000 08:20 12817                      /usr/lib/x86_64-linux-gnu/libgpg-error.so.0.34.0
7fa893bda000-7fa893bde000 r--p 00000000 08:20 13099                      /usr/lib/x86_64-linux-gnu/libzstd.so.1.5.5
7fa893bde000-7fa893c85000 r-xp 00004000 08:20 13099                      /usr/lib/x86_64-linux-gnu/libzstd.so.1.5.5
7fa893c85000-7fa893c92000 r--p 000ab000 08:20 13099                      /usr/lib/x86_64-linux-gnu/libzstd.so.1.5.5
7fa893c92000-7fa893c93000 r--p 000b7000 08:20 13099                      /usr/lib/x86_64-linux-gnu/libzstd.so.1.5.5
7fa893c93000-7fa893c94000 rw-p 000b8000 08:20 13099                      /usr/lib/x86_64-linux-gnu/libzstd.so.1.5.5
7fa893c94000-7fa893c97000 r--p 00000000 08:20 2183                       /usr/lib/x86_64-linux-gnu/liblzma.so.5.4.5
7fa893c97000-7fa893cb9000 r-xp 00003000 08:20 2183                       /usr/lib/x86_64-linux-gnu/liblzma.so.5.4.5
7fa893cb9000-7fa893cc4000 r--p 00025000 08:20 2183                       /usr/lib/x86_64-linux-gnu/liblzma.so.5.4.5
7fa893cc4000-7fa893cc5000 r--p 00030000 08:20 2183                       /usr/lib/x86_64-linux-gnu/liblzma.so.5.4.5
7fa893cc5000-7fa893cc6000 rw-p 00031000 08:20 2183                       /usr/lib/x86_64-linux-gnu/liblzma.so.5.4.5
7fa893cc6000-7fa893cc9000 r--p 00000000 08:20 12888                      /usr/lib/x86_64-linux-gnu/liblz4.so.1.9.4
7fa893cc9000-7fa893ce4000 r-xp 00003000 08:20 12888                      /usr/lib/x86_64-linux-gnu/liblz4.so.1.9.4
7fa893ce4000-7fa893ce6000 r--p 0001e000 08:20 12888                      /usr/lib/x86_64-linux-gnu/liblz4.so.1.9.4
7fa893ce6000-7fa893ce7000 r--p 00020000 08:20 12888                      /usr/lib/x86_64-linux-gnu/liblz4.so.1.9.4
7fa893ce7000-7fa893ce8000 rw-p 00021000 08:20 12888                      /usr/lib/x86_64-linux-gnu/liblz4.so.1.9.4
7fa893ce8000-7fa893cf7000 r--p 00000000 08:20 12791                      /usr/lib/x86_64-linux-gnu/libgcrypt.so.20.4.3
7fa893cf7000-7fa893deb000 r-xp 0000f000 08:20 12791                      /usr/lib/x86_64-linux-gnu/libgcrypt.so.20.4.3
7fa893deb000-7fa893e26000 r--p 00103000 08:20 12791                      /usr/lib/x86_64-linux-gnu/libgcrypt.so.20.4.3
7fa893e26000-7fa893e2b000 r--p 0013e000 08:20 12791                      /usr/lib/x86_64-linux-gnu/libgcrypt.so.20.4.3
7fa893e2b000-7fa893e2f000 rw-p 00143000 08:20 12791                      /usr/lib/x86_64-linux-gnu/libgcrypt.so.20.4.3
7fa893e2f000-7fa893e32000 rw-p 00000000 00:00 0
7fa893e32000-7fa893e35000 r--p 00000000 08:20 1947                       /usr/lib/x86_64-linux-gnu/libcap.so.2.66
7fa893e35000-7fa893e3b000 r-xp 00003000 08:20 1947                       /usr/lib/x86_64-linux-gnu/libcap.so.2.66
7fa893e3b000-7fa893e3d000 r--p 00009000 08:20 1947                       /usr/lib/x86_64-linux-gnu/libcap.so.2.66
7fa893e3d000-7fa893e3e000 r--p 0000b000 08:20 1947                       /usr/lib/x86_64-linux-gnu/libcap.so.2.66
7fa893e3e000-7fa893e3f000 rw-p 0000c000 08:20 1947                       /usr/lib/x86_64-linux-gnu/libcap.so.2.66
7fa893e3f000-7fa893e55000 r--p 00000000 08:20 13012                      /usr/lib/x86_64-linux-gnu/libsystemd.so.0.38.0
7fa893e55000-7fa893ee4000 r-xp 00016000 08:20 13012                      /usr/lib/x86_64-linux-gnu/libsystemd.so.0.38.0
7fa893ee4000-7fa893f12000 r--p 000a5000 08:20 13012                      /usr/lib/x86_64-linux-gnu/libsystemd.so.0.38.0
7fa893f12000-7fa893f1d000 r--p 000d2000 08:20 13012                      /usr/lib/x86_64-linux-gnu/libsystemd.so.0.38.0
7fa893f1d000-7fa893f1e000 rw-p 000dd000 08:20 13012                      /usr/lib/x86_64-linux-gnu/libsystemd.so.0.38.0
7fa893f1e000-7fa893f1f000 rw-p 00000000 00:00 0
```

完整信息不列在此处。从信息中，我们可以知道：表格是虚拟内存映射表，完整列出了.text，.rodata,.data/.bss，堆栈等模块的内存位置信息。

使用 whereis 查找 top 的完整路径名：

```
ainfinity@AInfinity:~$ whereis top
top: /usr/bin/top /usr/share/man/man1/top.1.gz
```

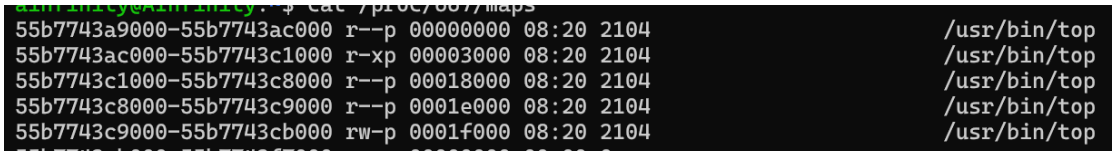
然后我们可以使用 readelf -S 查看完整节头信息。

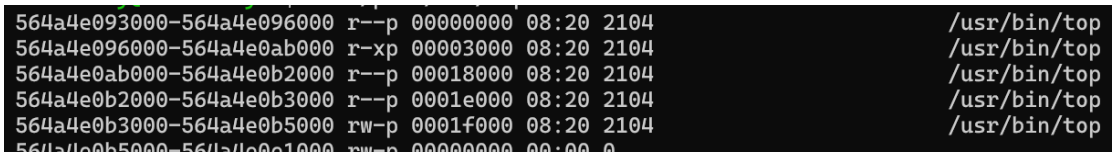
```
ainfinity@AInfinity:~$ readelf -S /usr/bin/top
There are 32 section headers, starting at offset 0x206d0:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000350  00000350
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000370  00000370
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             00000000000003a0  000003a0
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             00000000000003c4  000003c4
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003e8  000003e8
       0000000000000034  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           0000000000000420  00000420
       0000000000000c90  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           00000000000010b0  000010b0
       000000000000061e  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           00000000000016ce  000016ce
       000000000000010c  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          00000000000017e0  000017e0
       00000000000000c0  0000000000000000   A       7     3     8
  [10] .rela.dyn         RELA             00000000000018a0  000018a0
       0000000000000ae0  0000000000000018   A       6     0     8
  [11] .rela.plt         RELA             0000000000002380  00002380
       0000000000000b28  0000000000000018  AI       6    26     8
  [12] .init             PROGBITS         0000000000003000  00003000
       000000000000001b  0000000000000000  AX       0     0     4
  [13] .plt              PROGBITS         0000000000003020  00003020
       0000000000000780  0000000000000010  AX       0     0     16
  [14] .plt.got          PROGBITS         00000000000037a0  000037a0
       0000000000000040  0000000000000010  AX       0     0     16
  [15] .plt.sec          PROGBITS         00000000000037e0  000037e0
       0000000000000770  0000000000000010  AX       0     0     16
  [16] .text             PROGBITS         0000000000003f50  00003f50
       0000000000013752  0000000000000000  AX       0     0     16
  [17] .fini             PROGBITS         00000000000176a4  000176a4
       000000000000000d  0000000000000000  AX       0     0     4
  [18] .rodata           PROGBITS         0000000000018000  00018000
       0000000000005194  0000000000000000   A       0     0     32
  [19] .eh_frame_hdr     PROGBITS         000000000001d194  0001d194
       000000000000025c  0000000000000000   A       0     0     4
  [20] .eh_frame         PROGBITS         000000000001d3f0  0001d3f0
       0000000000000ca8  0000000000000000   A       0     0     8
  [21] .tbss             NOBITS           000000000001f470  0001e470
       000000000000010c  0000000000000000 WAT       0     0     4
  [22] .init_array       INIT_ARRAY       000000000001f470  0001e470
       0000000000000008  0000000000000008  WA       0     0     8
```

对比分析两者区别我们可以发现：readelf -S 获得的是程序在磁盘上的内存布局，也即虚拟地址，而 proc map 显示的是真实的内存中的地址。Readelf 显示所有节，而 proc maps 只显示实际加载到内存中的部分。

1.2

打开两个 shell，分别执行 top &命令。随后分别 proc maps 查看内存映射。





可以很明显的看到，每次代码段的内存地址都不一样了。这是因为随机化地址空间布局导致的。操作系统为了抵御攻击，会在每次运行程序的时候给程序加上一个随机的内存偏移值。这样，相同的程序多次运行内存位置也会变得不一样。

1.3

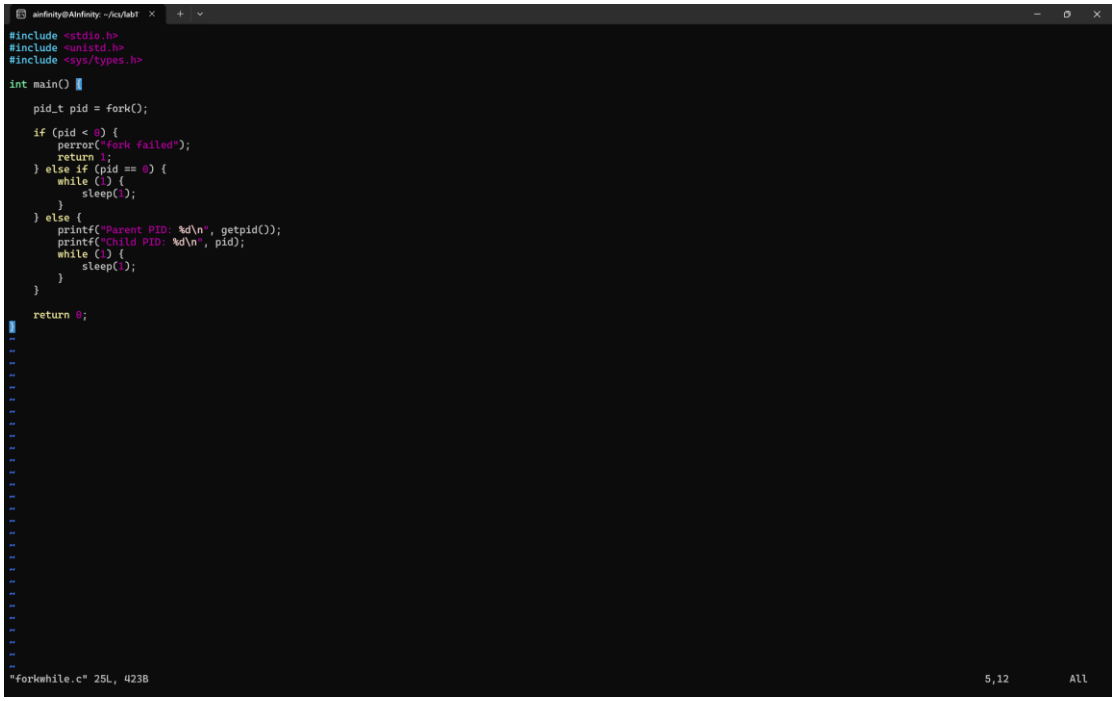
/proc 文件系统是一个虚拟文件系统，它向用户提供了一个接口来查看内核的数据结构。实际上，/proc 并不包含真正的文件，而是由内核在运行时动态生成的内容。它的主要目的是提供一种机制，使得用户空间的应用程序可以与内核进行通信和交互。

/proc 的作用相当广泛，可以查看到进程状态，内存映射，CPU 和内存统计，网络统计，模块列表等。

/proc/meminfo 可以查看系统内存信息。

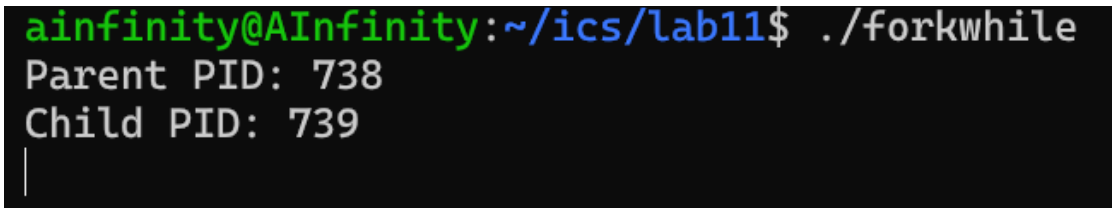
/proc/modules 列出了当前所有的内核模块。

2.1

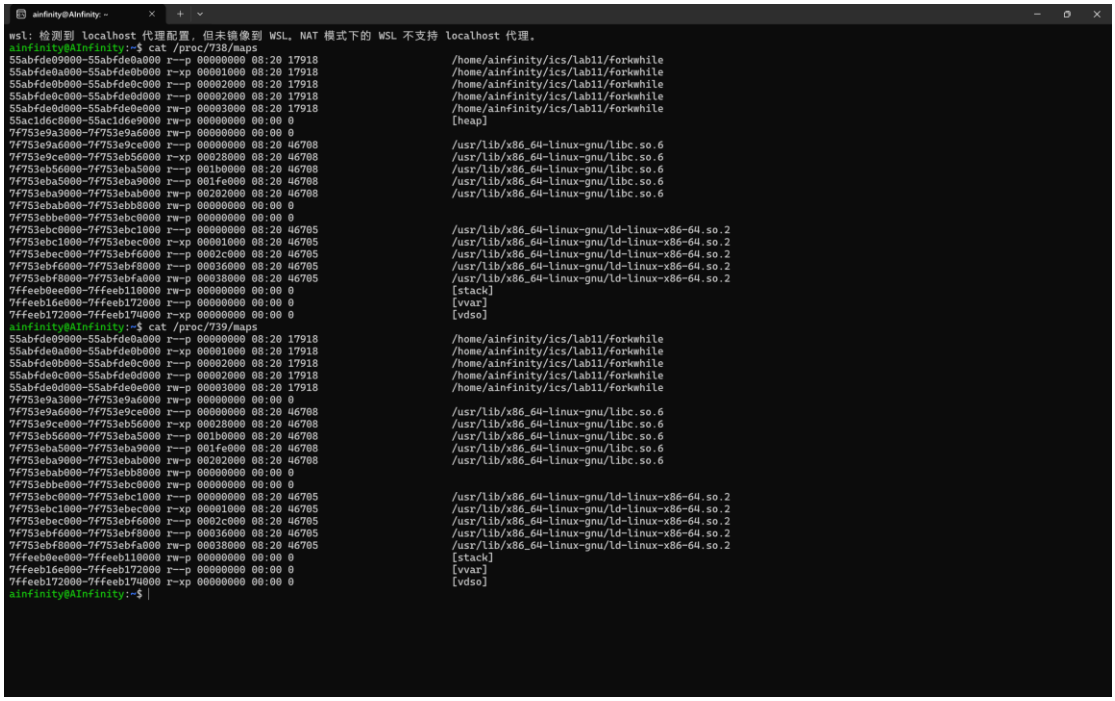


编写程序如图所示。程序创建了一个子程序，并且使父子程序都进入死循环。

启动程序。

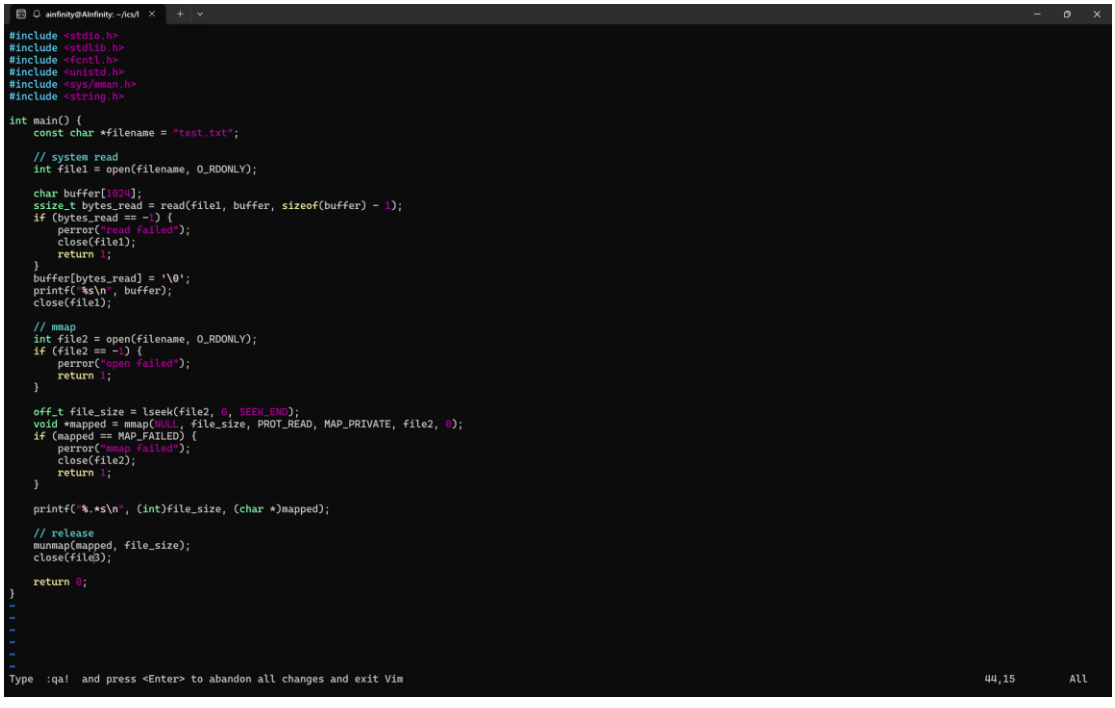


启动另外一个终端，分别对两个进程 proc map。

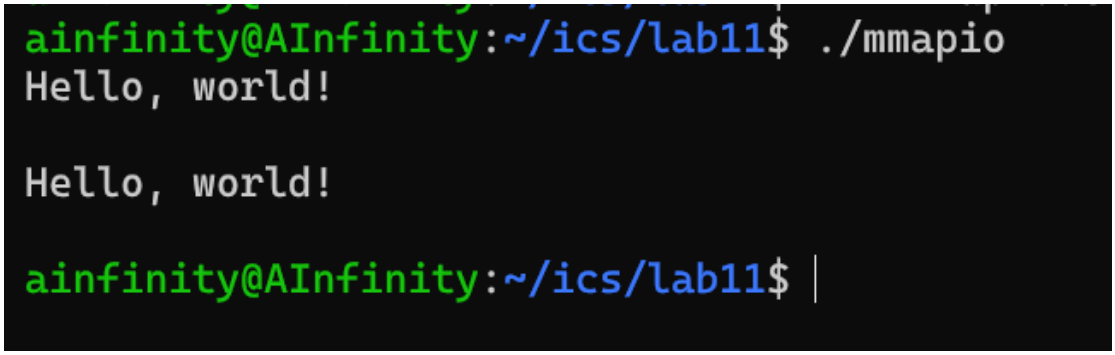


可以看到，子进程和父进程的虚拟内存空间几乎完全相同。这是因为，fork()在创建子进程的时候会复制父进程的页表，在子进程未对数据进行写操作时，二者共享同一块内存页面。由于写时复制技术，只有当子进程尝试对内存写入数据时，才会触发缺页中断，系统复制页码进行隔离。

2.2



编写代码如图所示。代码分别通过 mmap 和 read 两种方式实现了文件读取。输出结果如下：



Mmap 在性能上和使用上都要优于 read，因为它不需要显式调用函数 read，并且利用页机制进行按页加载，并且不需要手动定义缓冲区，而是由操作系统自动管理。但是 mmap 需要手动进行页对齐，更容易出错。

2.3

在 linux 中，进程间通信的方式有很多。列举如下：

1. 共享内存
   共享内存允许多个进程共享同一块内存区域。这是由虚拟内存机制实现的。让多个进程链接到相同的物理页即可。
2. 消息队列
   通过队列结构来发送或接受消息的数据传输方式。
3. 信号量
   信号量是一种用于控制多个进程对共享资源访问的计数器。
4. 管道
   创建管道后，一端用于输入数据，一端用于输出数据。