Assignment 4 Project Report

Li Jiaqi 120090727

Overview

In this assignment, I have completed both the basic single-directory File System and the bonus task of tree-structured directory File System. The following content will describe the relevant information of these two tasks I completed.

Environment

OS Version

I use the university's High Performance Cluster (HPC) for testing and running the CUDA program. The nodes run on a CentOS version 7.5.1804.

```
[120090727@node21 ~]$ cat /etc/redhat-release
CentOS Linux release 7.5.1804 (Core)
```

Kernel version

This is the kernel version of the HPC. Other versions should also be OK.

```
[120090727@node21 ~]$ uname -r
3.10.0-862.el7.x86_64
```

CUDA Version

I use the CUDA compiler version 11.7 for compiling the CUDA program.

```
[120090727@node21 ~]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Jun__8_16:49:14_PDT_2022
Cuda compilation tools, release 11.7, V11.7.99
Build cuda_11.7.r11.7/compiler.31442593_0
```

GPU Info

For each node in the HPC, it is equipped with a Quadro RTX 4000 GPU. Each time the program only runs on one allocated node. I have also tested my programs on a RTX3090 GPU.

```
| NVIDIA-SMI 515.65.01    Driver Version: 515.65.01    CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Quadro RTX 4000     Off  | 00000000:AF:00.0 Off |                  N/A |
| 36%   63C    P0    59W / 125W |    250MiB /  8192MiB |    100%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Running the program

Basic task compilation and running

To compile: inside the `source/` folder, there is a file named `slurm.sh`. On the HPC with slurm installed, we can directly use the shell script to compile and run the executable:

```
sbatch slurm.sh
```

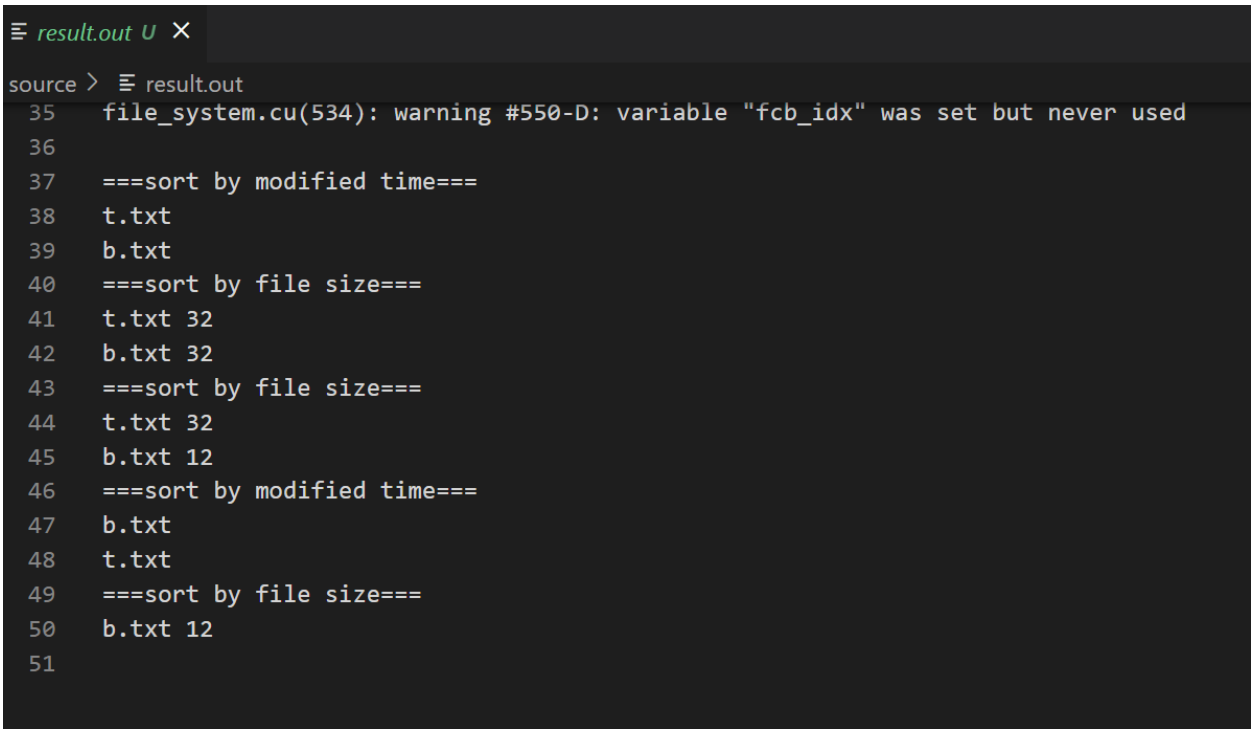
On a device without slurm, one can first compile using

```
nvcc --relocatable-device-code=true main.cu user_program.cu file_system.cu -o test
```

and run `./test` to run the program (might need `srun` in the cluster).

Basic task sample outputs

On the first test program: (The first 36 lines are compiler warnings)



On the second test program: (The first 36 lines are compiler warnings)

source > result.out

```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by file size===
t.txt 32
b.txt 12
===sort by modified time===
b.txt
t.txt
===sort by file size===
b.txt 12
===sort by file size===
*ABCDEFGHIJKLMNOPQR 33
)ABCDEFGHIJKLMNOPQR 32
(ABCDEFGHIJKLMNOPQR 31
'ABCDEFGHIJKLMNOPQR 30
&ABCDEFGHIJKLMNOPQR 29
%ABCDEFGHIJKLMNOPQR 28
$ABCDEFGHIJKLMNOPQR 27
#ABCDEFGHIJKLMNOPQR 26
"ABCDEFGHIJKLMNOPQR 25
!ABCDEFGHIJKLMNOPQR 24
b.txt 12
===sort by modified time===
*ABCDEFGHIJKLMNOPQR
)ABCDEFGHIJKLMNOPQR
(ABCDEFGHIJKLMNOPQR
'ABCDEFGHIJKLMNOPQR
&ABCDEFGHIJKLMNOPQR
b.txt

```

On the third test program: (there are too many lines and we only display the begin, middle and end)

```
===sort by modified time===
t.txt
b.txt
===sort by file size===
t.txt 32
b.txt 32
===sort by file size===
t.txt 32
b.txt 12
===sort by modified time===
b.txt
t.txt
===sort by file size===
b.txt 12
===sort by file size===
*ABCDEFGHIJKLMNOPQR 33
)ABCDEFGHIJKLMNOPQR 32
(ABCDEFGHIJKLMNOPQR 31
'ABCDEFGHIJKLMNOPQR 30
&ABCDEFGHIJKLMNOPQR 29
%ABCDEFGHIJKLMNOPQR 28
$ABCDEFGHIJKLMNOPQR 27
#ABCDEFGHIJKLMNOPQR 26
"ABCDEFGHIJKLMNOPQR 25
!ABCDEFGHIJKLMNOPQR 24
b.txt 12
===sort by modified time===
*ABCDEFGHIJKLMNOPQR
)ABCDEFGHIJKLMNOPQR
(ABCDEFGHIJKLMNOPQR
'ABCDEFGHIJKLMNOPQR
&ABCDEFGHIJKLMNOPQR
b.txt
===sort by file size===
~ABCDEFGHIJKLM 1024
}ABCDEFGHIJKLM 1023
|ABCDEFGHIJKLM 1022
{ABCDEFGHIJKLM 1021
zABCDEFGHIJKLM 1020
yABCDEFGHIJKLM 1019
xABCDEFGHIJKLM 1018
wABCDEFGHIJKLM 1017
vABCDEFGHIJKLM 1016
uABCDEFGHIJKLM 1015
tABCDEFGHIJKLM 1014
```

```
1057    DA 42
1058    CA 41
1059    BA 40
1060    AA 39
1061    @A 38
1062    ?A 37
1063    >A 36
1064    =A 35
1065    <A 34
1066    *ABCDEFGHIJKLMNOPQR 33
1067    ;A 33
1068    )ABCDEFGHIJKLMNOPQR 32
1069    :A 32
1070    (ABCDEFGHIJKLMNOPQR 31
1071    9A 31
1072    'ABCDEFGHIJKLMNOPQR 30
1073    8A 30
1074    &ABCDEFGHIJKLMNOPQR 29
1075    7A 29
1076    6A 28
1077    5A 27
1078    4A 26
1079    3A 25
1080    2A 24
1081    b.txt 12
1082    ===sort by file size===
1083    EA 1024
1084    ~ABCDEFGHIJKLM 1024
1085    aa 1024
1086    bb 1024
1087    cc 1024
1088    dd 1024
1089    ee 1024
1090    ff 1024
1091    gg 1024
1092    hh 1024
1093    ii 1024
1094    jj 1024
```

```
2063    XA 62
2064    WA 61
2065    VA 60
2066    UA 59
2067    TA 58
2068    SA 57
2069    RA 56
2070    QA 55
2071    PA 54
2072    OA 53
2073    NA 52
2074    MA 51
2075    LA 50
2076    KA 49
2077    JA 48
2078    IA 47
2079    HA 46
2080    GA 45
2081    FA 44
2082    DA 42
2083    CA 41
2084    BA 40
2085    AA 39
2086    @A 38
2087    ?A 37
2088    >A 36
2089    =A 35
2090    <A 34
2091    *ABCDEFGHIJKLMNOPQR 33
2092    ;A 33
2093    )ABCDEFGHIJKLMNOPQR 32
2094    :A 32
2095    (ABCDEFGHIJKLMNOPQR 31
2096    9A 31
2097    'ABCDEFGHIJKLMNOPQR 30
2098    8A 30
2099    &ABCDEFGHIJKLMNOPQR 29
2100    7A 29
2101    6A 28
2102    5A 27
2103    4A 26
2104    3A 25
2105    2A 24
2106    b.txt 12
2107
```

One the fourth test case: (there are too many lines and we only screenshot the beginning and end)

source > result.out

```
triggering gc
===sort by modified time===
1024-block-1023
1024-block-1022
1024-block-1021
1024-block-1020
1024-block-1019
1024-block-1018
1024-block-1017
1024-block-1016
1024-block-1015
1024-block-1014
1024-block-1013
1024-block-1012
1024-block-1011
1024-block-1010
1024-block-1009
1024-block-1008
1024-block-1007
1024-block-1006
1024-block-1005
1024-block-1004
1024-block-1003
1024-block-1002
1024-block-1001
1024-block-1000
1024-block-0999
1024-block-0998
1024-block-0997
1024-block-0996
1024-block-0995
1024-block-0994
1024-block-0993
1024-block-0992
1024-block-0991
1024-block-0990
1024-block-0989
1024-block-0988
1024-block-0987
1024-block-0986
1024-block-0985
1024-block-0984
1024-block-0983
1024-block-0982
1024-block-0981
```

source > result.out

```
1009    1024-block-0041
1010    1024-block-0040
1011    1024-block-0039
1012    1024-block-0038
1013    1024-block-0037
1014    1024-block-0036
1015    1024-block-0035
1016    1024-block-0034
1017    1024-block-0033
1018    1024-block-0032
1019    1024-block-0031
1020    1024-block-0030
1021    1024-block-0029
1022    1024-block-0028
1023    1024-block-0027
1024    1024-block-0026
1025    1024-block-0025
1026    1024-block-0024
1027    1024-block-0023
1028    1024-block-0022
1029    1024-block-0021
1030    1024-block-0020
1031    1024-block-0019
1032    1024-block-0018
1033    1024-block-0017
1034    1024-block-0016
1035    1024-block-0015
1036    1024-block-0014
1037    1024-block-0013
1038    1024-block-0012
1039    1024-block-0011
1040    1024-block-0010
1041    1024-block-0009
1042    1024-block-0008
1043    1024-block-0007
1044    1024-block-0006
1045    1024-block-0005
1046    1024-block-0004
1047    1024-block-0003
1048    1024-block-0002
1049    1024-block-0001
1050    1024-block-0000
1051
```

Bonus Task Compilation and running

Because the bonus task shares the same template structure with the basic task, the compilation and running steps are exactly the same as above, which means we could use:

```
sbatch slurm.sh
```

Bonus task sample output

The first 91 lines are compiler warnings.

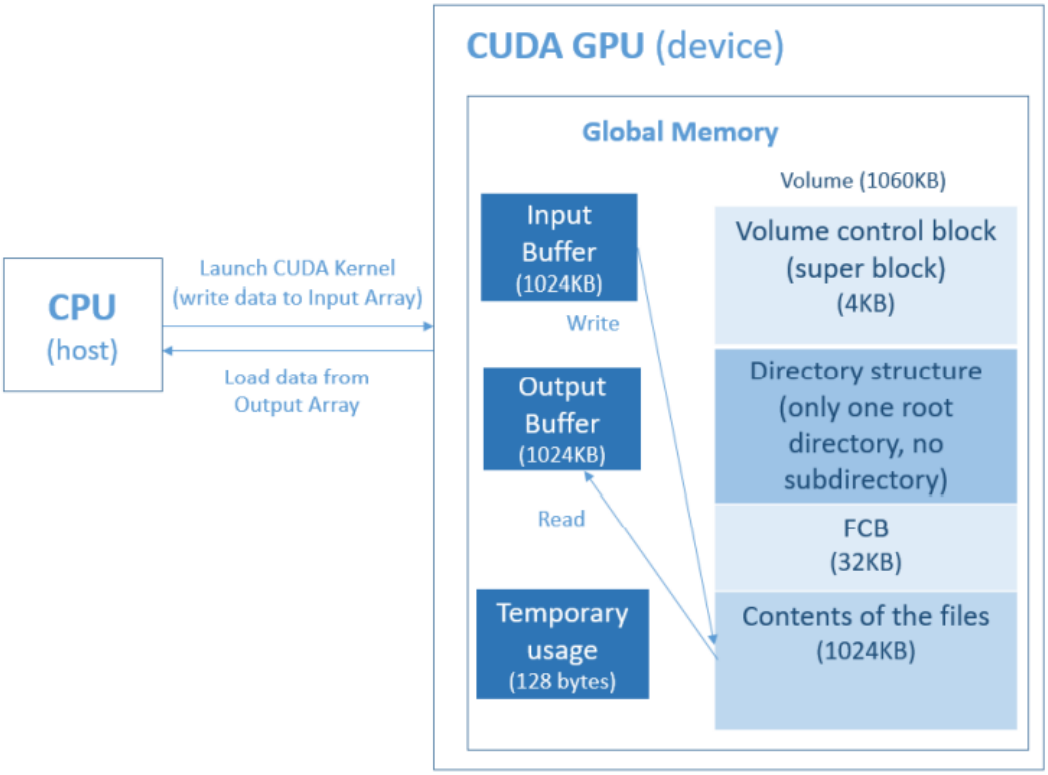
bonus > result.out

```
92   ===sort by modified time===
93   t.txt
94   b.txt
95   ===sort by file size===
96   t.txt 32
97   b.txt 32
98   ===sort by modified time===
99   app d
100  t.txt
101  b.txt
102  ===sort by file size===
103  t.txt 32
104  b.txt 32
105  app 0 d
106  ===sort by file size===
107  ===sort by file size===
108  a.txt 64
109  b.txt 32
110  soft 0 d
111  ===sort by modified time===
112  soft d
113  b.txt
114  a.txt
115  /app/soft
116  ===sort by file size===
117  B.txt 1024
118  C.txt 1024
119  D.txt 1024
120  A.txt 64
121  ===sort by file size===
122  a.txt 64
123  b.txt 32
124  soft 24 d
125  /app
126  ===sort by file size===
127  t.txt 32
128  b.txt 32
129  app 17 d
130  ===sort by file size===
131  a.txt 64
132  b.txt 32
133  ===sort by file size===
134  t.txt 32
135  b.txt 32
136  app 12 d
```

Program Design

Basic task design

In this CUDA program, we implement a single-directory file system using a limited GPU memory pool. The memory usage strictly obeys the one in the instruction, that no extra global memory is maintained or used. Temporary usage for function stack is limited.



For the task, we allocate a volume of 1060kb with the 4kb volume control block using bitmap, 32kb for 1024 FCBs, each FCB is 32 bytes. The content of files use 1024kb, divided into storage blocks each 32 bytes.

FCB Structure

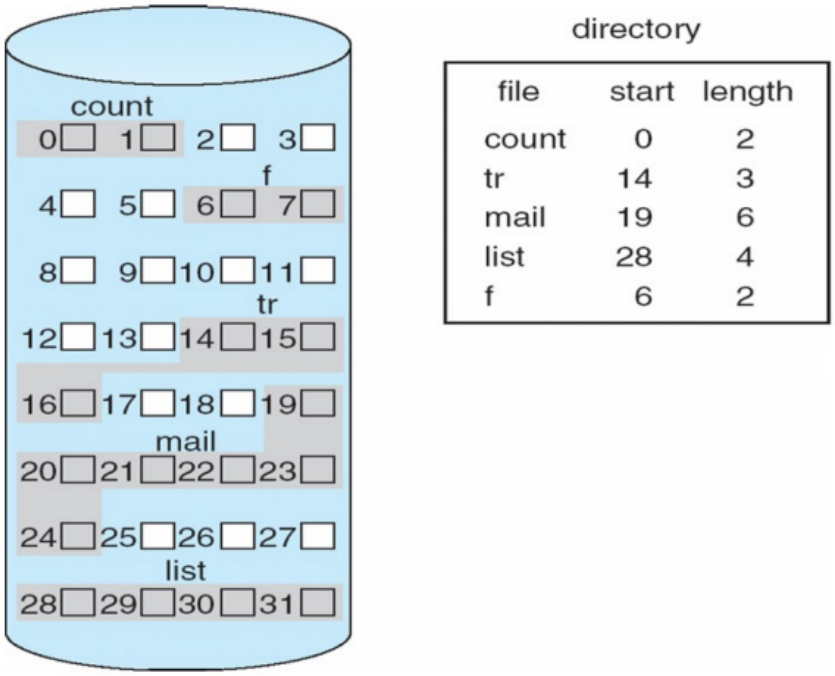
Here is the FCB structure I used. Note that this does not create extra memory space, we just turn the specific portion of volume, originally `uchar*` to `FCB*` for better information storage and retrieval. I have tested that the `sizeof(FCB)` is 32, which is exactly the size of desired FCB. The attributes are self-explanatory.

```
// **32 bytes** File control block. We will turn the FCB part in volume into (FCB*)
// sizeof(FCB) is 32
struct FCB {
  char filename[20];  // maximum size of filename is 20 bytes
  u32 size; // the size of the file **in bytes**
  u16 modified_time;  // the last modified time
  u16 creation_time;
  u16 start_block_idx;  // the index of the first of its contiguous blocks
  bool is_on;
};
```

Allocation strategy

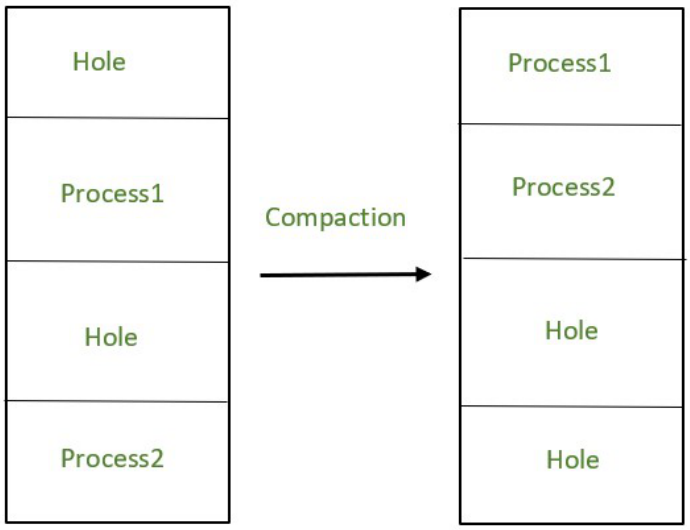
The maximum file size allowed is the total content size of files, 1024 KB. I use dynamic, contiguous allocation together with compaction algorithm to maintain the FS. The dynamic scheme is to allow storing this maximum size of file.

The contiguous allocation is one such that adjacent blocks store the file content sequentially, as illustrated in the figure:



The compaction algorithm is utilized when there is fragmentation and a newly written file cannot file enough space. In my compaction algorithm implementation, I maintain a pointer to the first unused block and the first used block moving forward together. They

are constantly swapped, which in effect “compacts” all the used blocks to the front. In the mean time, we update the FCB’s content block attribute.



Compaction algorithm

For the superblock, I use the bitmap, which uses one bit for one content block to indicate on or off.

Designing the APIs

All the required APIs `fs_open`, `fs_read`, `fs_write`, `fs_gsys` (including LS_S, LS_D, rm operations) are implemented. The `fs_open` returns an fp, which is the index of the FCB in the FCB array. Another interesting point is for the LS_D and LS_S operations. I did not use external storage for these two sorting operations. Instead, my implementation is simple, which is in each time, traverse all files and find the largest element that is not printed. This does not need to re-place the blocks. The following figure shows my implementation of `LS_D`.

```
printf("===sort by modified time===\n");


int last_item_time = (1<<15); // trace the time of last printed file
// print the most recent modified file before the last item
for (int i = 0; i < file_count; i++)
{
  int latest_modified_time = 0;
  FCB latest_fcb;
  for (int j = 0; j < fs->FCB_ENTRIES; j++)
  {
    FCB fcb = START_OF_FCB[j];
    if (fcb.is_on && (fcb.modified_time > latest_modified_time) && (fcb.modified_time < last_item_time))
    {
      latest_fcb = fcb;
      latest_modified_time = fcb.modified_time;
    }

  }
  last_item_time = latest_fcb.modified_time;
  printf("%s\n", latest_fcb.filename);
  // printf("%s   time%d\n", latest_fcb.filename, last_item_time);
}
```

The LS_S is more tricky but still uses the above idea. We do three traverses in total. First we traverse each item to get the largest unprinted size of files. Then we traverse to get the count of the largest size files. Then find the file with the file size of largest_file_size and the **earliest created time** among all unprinted items.

Bonus task design

The bonus task is based upon the basic task with modification to add files for directories.

Firstly, we need to add a new attribute to trace the current working directory. I add to the `fs` struct.

```
struct FileSystem {
    uchar *volume;
    int SUPERBLOCK_SIZE;
    int FCB_SIZE;
    int FCB_ENTRIES;
    int STORAGE_SIZE;
    int STORAGE_BLOCK_SIZE;
    int MAX_FILENAME_SIZE;
    int MAX_FILE_NUM;
    int MAX_FILE_SIZE;
    int FILE_BASE_ADDRESS;

    int cwd;    // current working directory's fcb index, **not** block index
};
```

Then, the FCB should be different to record each file's directory index.

We squeeze the `is_on` and `is_dir` 1-bit attribute to the first two bits of `u32 size`. So this is 32 bytes again.

```
// File control block
// this is 32 bytes. We turn the FCB portion in the volume into (FCB*). So no extra space
struct FCB {
    char filename[20];  // maximum size of filename is 20 bytes
    u32 size;   // the size of the file **in bytes**
    u16 modified_time;  // the last modified time
    u16 creation_time;
    u16 start_block_idx;    // the index of the first of its contiguous blocks

    // if the FCB is a directory, it stores its parent dir index.
    // if the FCB is root directory, its parent dir index will be -1
    int16_t dir_idx;    // the index of the directory that the file is in; for a dir, it is its own idx
};
```

My implementation does not use extra global memory. Other implementations are similar to basic task. Because we record the file contents or subdirectory names in the content of directory, we need to traverse and check the filename match and the `dir_idx` match the cwd.

All required operations are supported, including the extra command `MKDIR`, `PWD`, `CD`, `RM_RF` and `CD_P`, in addition to the ones in basic task.

My implementation also supports absolute addressing to increase robustness.

Project reflection and conclusion

Several problems I met in this assignment gave me valuable experience in solving them. The first is

about data structure used to implement FCB in bonus. At first I thought 32 bytes is not enough and want to implement a doubly linked list in the bonus. However, later I found that I could squeeze the FCB on/off bits, and can use a filename match traversing strategy instead of linked list. This allows mroe efficient storage utilization.

I think this project is a valuable experience for learning the FS, including dynamic allocation, contiguous allocation, compaction. I also learn the technique of writing CUDA programs, which are somewhat like C/C++ but have restricted access to some standard library routines, like `memcpy`.