OPERATING SYSTEM

Abhishek Kumar Sah (19CS10004)
Saumyak Raj (19CS30040)

1 April 2022

—

Assignment 5

In this assignment, we have to build our Memory Management System, which requests a chunk of memory from the Operating System and then manages it instead of relying on it. This has many benefits for application developers who need complete memory control, and efficiency is the key. Such methods are generally used in database servers.

What the programmer needs to do:

The programmer does not need to do much to use this library. The steps he needs to follow are as follows:

1. First import the library
2. Declare a variable of type VMemory (for example, vmem)
3. Call the function vmem.createMem. This will allocate the memory and bookkeeping data structures and initialize everything needed for memory management.
4. At the beginning of every function call, he has to call vmem.fn_begin().
5. At the end of every function call, he must call vmem.fn_end().

He has to make sure that he follows these things; the rest of the library will automatically handle them all.

At the start of the library

The VMemory allocates the given amount of memory at the program's start. This is the place where all the variables are stored. The library uses an additional overhead for the PageTable and Stack, which is all of the sizes of MAX_VAR. All these are allocated in the beginning.

To implement the solution, several data structures are built. They are as follows:

1. VMemory
2. PageTableEntry
3. Stack
4. TYPE

VMemory

The data structure has the following declaration.

```
Struct Vmemory {
PageTableEntry* pgTable;
char* memory;
size_t memsize;
size_t count = 0;
size_t offset = 0;
size_t memory_holes = 0;
sem_t mutex;

pointer last_valid = -1;

Stack var_stack;
```

```
pthread_t tid;
pthread_attr_t attr;

pthread_t g_tid;
pthread_attr_t g_attr;

FILE* memory_footprint_fd;
std::chrono::steady_clock::time_point start;

static size_t getsize(TYPE type, int len);

int createMem(size_t size = MEMSIZE, size_t max_var = MAX_VAR);

pointer createVar(TYPE var_type);
pointer createArr(TYPE var_type, size_t size);

int assignVar(pointer adr, int val);
int assignArr(pointer adr, int index, int val);

int readVar(pointer adr);
int readArr(pointer adr, int index);

void freeElem(pointer adr);

void fn_begin();
void fn_end(int return_something = false, int return_value = 0,
TYPE return_type = INTEGER);

pointer returned_value();

size_t get_memory_footprint();

~Vmemory();
};
```

It contains all the values and methods needed for memory creation and bookkeeping.

1. pgTable: It is an array of PageTableEntry that stores information about individual variables
2. memory: It is the pointer to the memory allocated to the library
3. count: The count of the number of variables declared. It will also work as the pointer to a variable
4. offset: The position in the memory where the next variable will be allocated.
5. memory_holes: It is bytes of memory holes in the program at any point in time
6. mutex: it is used for locking the program in the critical section
7. var_stack: it is the stack of variable pointers

PageTableEntry :

The data structure has the following declaration.

```
struct PageTableEntry {
    char* ptr;
    bool valid_bit;
    TYPE var_type;
    int len;
};
```

The createMem method of the VMemory creates an array of PageTableEntry. An individual PageTableEntry stores the information of a single declared variable.

1. ptr: stores the pointer to the variable declared in the memory
2. valid_bit: stores if the variable is still being used or deleted by the garbage collector
3. var_type: stores the type of the variable; it can be of types defined in enum TYPE.
4. len: stores the len of the variable, for a single variable, it is 1, and for arrays, it is the size of the array

Stack:

The data structure has the following declaration.

```
struct Stack {
    int* arr;
    int top_pos = -1;
    size_t size();


    Stack();
    void push(int data);
    int empty();
    void pop();
    int top();

    ~Stack();
};
```

This is a simple array implementation of the stack data structure, with a max size of MAX_VAR, the maximum number of variables assumed to be declared in the program.

TYPE:

This has the following declaration.

```
enum TYPE {
    INTEGER,
```

```
    CHARACTER,
    MID_INTEGER,
    BOOLEAN,
};
```

This declares all the possible data types used in the program. If the variable is an array, its datatype will remain the same, but its len will change, which is declared in PageTableEntry.

Structure of Page Table

Our Page Table is a linear array of PageTableEntry, and it maps the variables to the exact location they are stored in the memory, the memory array. As new variables are declared, the count in VMemory is incremented, and this count is used to identify an individual variable. As a one-dimensional array is used, there is a maximum limit to the number of variables that can be declared.

Handling variables

We have four types of variables, namely, Integer (4 bytes), Boolean (1 bit), Character (1 byte), and Medium Integer (3 bytes). Though all the variables are of different sizes, the same amount, i.e., 4 bytes, is used for all the variable types. All the data types must be word-aligned to improve the performance, where one word = 4 bytes.

Handling arrays

The arrays are handled differently from the variables to improve efficiency and minimize redundant space. Only the minimum required amount of memory is given to the array, but some redundant bits are allocated at the end of the array to keep the addresses word-aligned. For example, if a character array is created of size 10, 12 bytes will be allocated, and the last 2 bytes are redundant to keep the addresses word-aligned. All the values are still read and written in the size of 4 bytes.

Working of Garbage Collector

The garbage collector of the solution works in two parts. First, at the end of every function, and second, running parallelly in a thread that wakes up at a certain interval and performs the compaction algorithm.

End of Function: The var_stack stores the pointer to the variables declared in the function. At the end of the function, when fn_end() is called, all the variables in the current function are popped and marked as invalid, and the memory allocated is freed. The count in VMemory is also decremented so that the library does not waste the page table entries. This is done at the end of every function call. This is efficient because the variables declared in the function must not be accessed in the parent function, so they must be marked invalid. Now, the garbage collection is just three extra O(1) operations. This also removes the necessity of running the compaction algorithm because all the local variables will always be freed, and memory holes will never be created at the end of a function. The benefits of doing this at the end of the function are much higher than the possible delay.

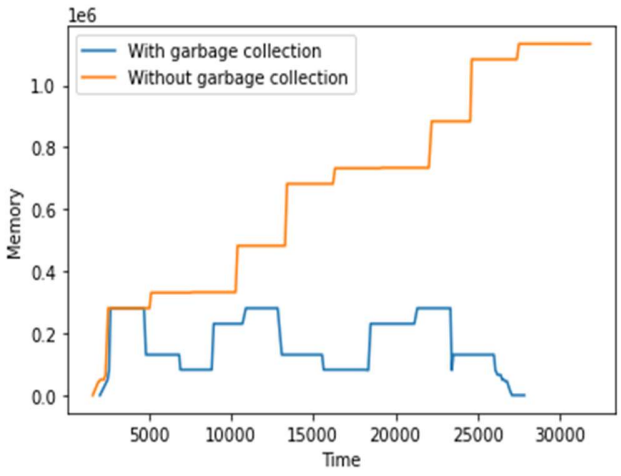
Garbage Collector running in parallel: There is a garbage collector running in parallel which wakes up at a certain interval of time. The primary duty it performs is to compact the memory. But since memory is always compacted at the end of every function call, this is needed significantly less often. It is only required when some large memory is freed in the function call itself.

Algorithm for compaction: The compaction algorithm is a simple linear algorithm—the algorithm notes where the last valid data was placed in the memory. The algorithm traverses the page table entries; if the variable is invalid, it does nothing. If it is valid, it copies the variable's data to the end of the position where the last valid data was placed. There will be no memory holes at the end of the algorithm. But this garbage collector does not change the count (pointer) to the variables because the variables are still declared in the scope.

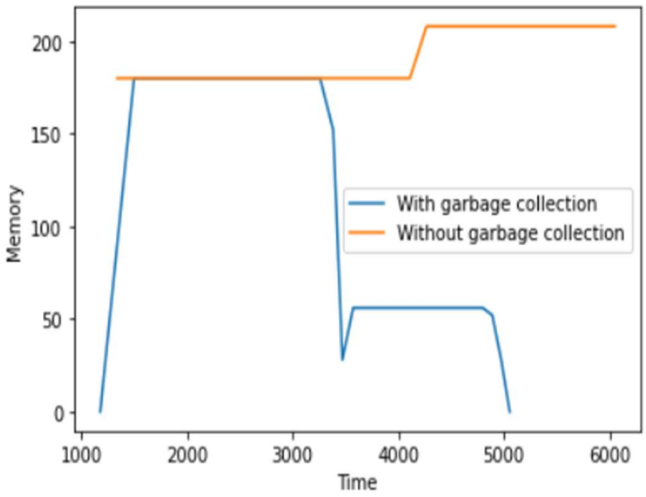
Use of Locks:

Semaphore locks have been used in all the library functions because when any variable is read or updated, the garbage collector must not run. After all, then it can corrupt the memory. So all the library functions and the garbage collector are locked. The garbage collector will be called only if the user code is running on the CPU.

Impact of Garbage Collector:



Effect of garbage collector for demo1



Effect of garbage collector for demo1

We can see here that when the garbage collector is disabled the memory used is just increasing for both demo1 and it is never compacted.
The memory used includes memory used by variables, page table, and variable stack.