OPERATING SYSTEM PROJECT REPORT MEMORY SIMULATOR

Clement Cole, Kuntal Patel

***Index Terms*—Memory, Process Generator, Malloc, Simple Malloc Implementation**

I. Introduction

This project will demonstrate how to implement malloc and free simulation using both dynamic and static partitioning schemes. We start our design by first building a test bench for malloc and free for a list of 50 processes each within the range of 10KB to 2MB and varying in number of cycles in execution. We then measure the system time and compare it to that of our implementation.

II. Implementation of Testing Malloc

Our desigin for the static partitioning Scheme contains the basic unit for each process structure as defined in Listing 1. below. Here each process structure contains a pid, number of cycles, memory size, the total number of blocks and a few member functions just in case to display process information.

Listing 1: Perceptron Struct In C

```
typedef struct process
{
        int pid;
        int cycles;
        int memory_size;
        int blocks;
        int leftover;
        int blocks =
        ceil(float(memory_size)
            /
            float(10240));
        void process_func(){
                printElement(pid ,
                    nameWidth);
                printElement("      ",
                nameWidth);
                printElement(cycles,
                nameWidth);
                printElement("      ",
                nameWidth);
                printElement(memory_size,
                nameWidth);
                cout << endl;
        }
}process;
```

We follow the above design with creating a vector to hold each process in which case we call it the ready queue. As shown in Listing 2., this vector will contain all of the processes that are generated for the purpose of this simulation.

Listing 2: Vector Of Processes

```
vector<process> process_ready_queue;
```

A. Process Generator Design

Next, we define a function to generate our processes and stage them on the ready queue described in Listing 2. Implementation of the process generator is simply setting a process ID equal to the index of iteration, designated cycles of operation and memory size. The latter two are generated randomly. Listing 3. shows the function description for generating each process.

Listing 3: Process Generator

```
void generate_processes()
{
        srand((unsigned) time(0));
        for(int i=0; i < 50; i++) //Generate 50 proceses
        {
                process thisprocess;
                thisprocess.pid = i;
                thisprocess.cycles = rand() %
                    (MAX_PROCESS_SIZE - MIN_PROCESS_SIZE) +
                        MIN_PROCESS_SIZE;
                thisprocess.memory_size = rand() %
                    (MAX_CYCLES-MIN_CYCLES) +
                        MIN_CYCLES;

                process_ready_queue.push_back(thisprocess);
        }
}
```

We then run inside the main function the generate process function to initialize our process generation. After which we use a loop to iterate each process, assigning a memory location to it with the malloc function. The times corresponding to each allocation is recorded for further analysis.

III. Implementation of Static Partitioning Scheme

We started our static memory partitioning scheme by first describing the unit for each basic block of memory. Since the lowest size a process can hold in block size 10KB, our basic block size is 10KB. Listing.4 below gives a description on how each block is described in our implementation.

Listing 4: Basic Block Definition

```
typedef struct basic_block
{
        int basic_block_size;
```

```
        int pid;
        int unique_id;
        bool free;
        //struct basic_block* next;
}basic_block;
```

As shown in Listing 4., each block contains a member of basic block size to indicate the size of the block, a pid value to indicate to what process it is allocated to. A unique id, in case we have to defragment the entire memory system, we will still have to keep track of each block's unique id for future identification. Basically, the structure above contains the metadata for each block, we could have added a few other members including address etc.

1) Implementation Of the Malloc: The malloc function is implemented by traversing the main memory one block after the other, each time checking to see if the block is free or designated to a process. This is verified by checking both the Unique ID of each block with its correspond pid (tells us what Process it belongs to). Listing 5. shows the implementation of the mymallock function which accepts a process type object by reference, in order to save overhead of copying objects from one memory location to another.

Listing 5: Homebrew Malloc Function Implementation

```
void mymalloc(process *thisprocess)
{
        auto j(0);
        long long int diff;
        start_time = rdtsc();
        for(auto& i : MainMemory)
        {
        j++;
        i.free = false;
        i.pid = thisprocess->pid;
        //iter_alloc.push_back(i);
        if (j == thisprocess->blocks)
        {
                end_time =rdtsc();
                printElement(end_time -
                    start_time   , nameWidth);
                break;
        }
        }//Allocated_list.push_back(iter_alloc);
}
```

2) MyFree implementation: Our free function follows a similar paradigm compared to tha of myfree function described above. Listing 6. gives a detailed functional implementation of the free function. The implementation of the myfree() function involves just doing the reverse of what was previously implemented in malloc function. In that we set each pid value within each block as -1 and the value free to true.

Listing 6: Homebrew Free Function Implementation

```
void myfree(process thisprocess)
{
        auto j(0);
        long long int diff;
        long long int end;
        long long int start = rdtsc();
        for(auto& i : MainMemory)
        {
                if((i.free != true) &&
                (i.pid == thisprocess.pid))
                {
                        i.free = true;
                        i.pid = -1;
                        j++;

                }
                if(j == thisprocess.blocks)
                {
                        end = rdtsc();
                        printElement(end -start , nameWidth)
                        cout << endl;
                        break;
                }
        }
}
```

IV. Implementation of Dynamic Partitioning Scheme

The implementation of the dynamic partitioning scheme shares the same data structures as that of the static. However, the only difference is the how each metadata for each block is represented. Here, we use a floor function to find the number of blocks needed for each process. Number of blocks equals the total memory size divided by 10KB. The remainder is then assigned to a leftover memory block. As shown below the malloc function for the dynmaic implementation requires that we defragment our memory unit after every assignment of blocks to a particular process. Listing 7, 8 and 9 shows the implementation of our dynamic malloc, free and defragmentation(Using the STL algorithm of rotate for speed).

Listing 7: Dynamic Malloc Function Implementation

```
void malloc(process &thisprocess)
{

        {
        dynamic_block current_block;
        current_block.basic_block_size =
    (num_of_blocks * 10240);//size of process mm ;
        current_block.pid = pid;
        current_block.free = false;
        Dynamic_Partitioned_Memory.push_back(current_block)

        }
        if (leftover >0)
        {
        dynamic_block leftover_block;
        leftover_block.basic_block_size = leftover;
        leftover_block.pid = pid;
        leftover_block.free = false;
        Dynamic_Partitioned_Memory.push_back(leftover_block
        }
}
```

Listing 8: Dynamic Free Function Implementation

```
void myfree2(process thisprocess)
{
        auto j(0);
        long long int diff;
        long long int end;
        long long int start = rdtsc();
        for(auto& i :
                Dynamic_Partitioned_Memory)
        {
                if((i.free != true) &&
                (i.pid == thisprocess.pid))
                {
                i.free = true;
                i.pid = -1;
                i.basic_block_size = 10240;
                }
                end = rdtsc();
                printElement(end -start,
                        nameWidth);
                cout << endl;
                break;
                }

}
```

Listing 9: STL Rotate Function Implementation

```
void myfree(process thisprocess)
void binary_rotate()
{
        //int i=0;
        std::vector<basic_block>::iterator bound;
        bound = std::partition(MainMemory.begin(),
            MainMemory.end(),
                IsAllocated);
        auto i(0), j(0);
}
```

V. Testing

Our test involves measuring the number of ticks between the allocation of memory from our list of blocks to the respective required block for each process. The measurement is in ticks which represents the number of CPU cycles between each transaction of memory allocation for each process. Below in Fig. 1 shows the results while testing our malloc implemetation against that of the system's malloc.

VI. Observation

As shown in Fig. 1 and 2, the number of clock cycles for the system's implementation of Malloc and Free is almost twice that of the simulation version. This might be due to the low amount of context switches done by our simulated processes vice that implemented by the malloc and free function call to kernel of our linux operating system. Also, another reason will be due to the fact memory allocation for our processes which is done on the fly by the std::allocator, optimized to outperform malloc and new. For instance, when we call a push back function api on a vector object, a new block of memory is allocated.

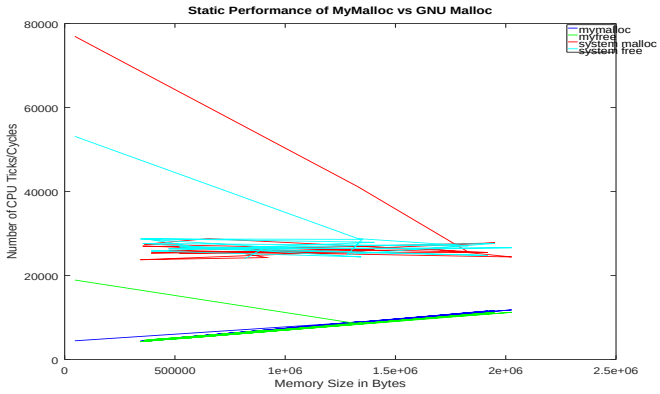

Fig. 1: Results for Static Partition

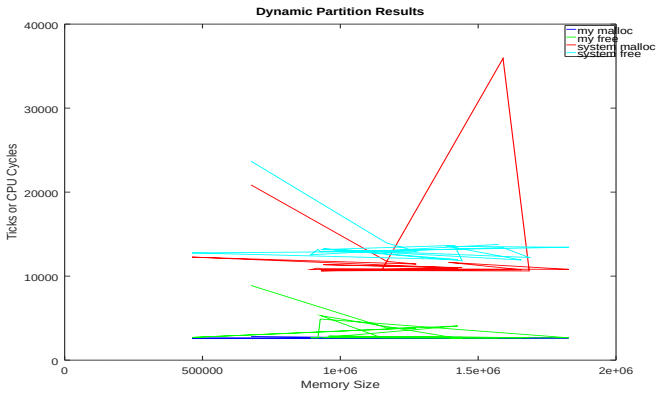

Fig. 2: Results for Dynamic Partition

The size of the overall vector object is then doubled to accommodate future push back calls. In some systems, it

might increase by a factor of 1.5, however over the long run, we would have enough space for the vector object to accommodate any future element hence not requiring any addition copy or move of our current memory location.

VII. Conclusion

Although, our implementation of malloc is hard to compare to that of the system's malloc and free, we think doing this experiment has given us an insight of how implementation of system memory allocation and deallocation can greatly improve or impeded the performance of processes in a computing system. The data locality of our code which enables us to allocate memory to pseudo-processes provided us with better performance compared to malloc/free transactions of memory allocation on the kernel level.