University of North Texas

Project #2

Memory Management

Marco Duarte

Operating System Design – CSCE 5640

Dr. Armin Mikler

December 4, 2017

Introduction

Whenever a program needs to be run the operating system pulls a process from the input queue, allocates it memory, and loads the program into it to be run. This project aims to simulate the operating system's memory manager by generating a set of "processes" with varying memory requirements and execution time and measuring how long it takes for the set of generated processes to finish executing as well as comparing the times our own memory management systems, which use dynamic and static partitioning schemes, result finish in.

Implementation

Processes

In this simulation, processes are represented by the *process* structure, which is defined as:

```
typedef struct process{
  int id;
  size_t memSize;
  int cycles;
  int start;
  int end;
  char* memory;
} proc;
```

The member *id* represents the process id and is assigned at when the process is generated. *memSize* stores the amount of memory the process needs allocated to it to run which is assigned during processes generation. *cycles*, another member assigned during process generation, stores the amount of cycles the process will take to "execute" to completion. Members *start* and *end* denote at which cycle a process was admitted and at which cycle the process finished executing. Finally, *memory* is a pointer which points to the memory that has been allocated to the process.

Process Generation

The process generator is very simple and short and defined in *processes.h* as:

```
proc* procGen()
{
  static int id = 0;
  proc *temp = (proc*)malloc(sizeof(proc));

  temp->id = id; //set process id
  temp->memSize = (size_t)(random()%(MAXSIZE - MINSIZE + 1) + MINSIZE); //10KB <-> 2MB
  temp->cycles = random()%(MAXCYC - MINCYC + 1) + MINCYC; // 200 <-> 2500 cycles

  id++; //increment id
  return temp;
}
```

This function has a *static int* local variable named *id* that is used to assign the id of the process each time it is called. This function allocates memory for a process, assigns a random size requirement, in the range of 10KB to 2MB, and a random cycle amount, ranging from 200 to 2500 cycles, using the POSIX function call *random()* . MAXSIZE, MINSIZE, MAXCYC, and MINCYC are used as the maximum memory request (2MB) in bytes, the minimum memory request size (10KB) in bytes, the maximum cycles (2500), and the minimum cycles (200), respectively, and are defined in the file *processes.h along with other such constant.* Once the id, size, and cycles of temp are assigned, the function returns the pointer of the structure.

In the file *start.c* in the main function, NUMPROC, defined as 50 in *processes.h*, amount of processes are generated randomly and are output to a file *proc*. Once all the processes have been generated and outputted, the program starts each memory management program (*system, dynamic, static)* to read the *proc* file and simulate their execution. Each memory management system is started individually using *execl()*. Once one starts *start* waits until it finishes before starting the next memory management system. Each memory management system outputs information to a separate file (*system* to *sysOut, dynamic* to *dynOut, static* to *stOut*).

Simulator

Before simulation, each memory management system must allocate space for NUMPROC amount of processes. Then the proc file is opened and the data output from *start* is read and stored into the *proc** *set* variable. Each memory management system program has different partitioning techniques, but the simulation setup is the same for each. The simulation takes place in an infinite loop with each loop representing a CPU cycle. Every 50 cycles a process is admitted into the system and sent to the memory manager for partitioning. In the simulation there is a check to see if the processes have finished executing. If they are not finished, the current system time is read using *gettimeofday()* and then the memory management system's partition function is called. Inside the partitioning functions memory is allocated to processes and are also allowed to execute for 1 cycle, updating their remaining cycle time. After the partition function has returned, the system time is read again. The two read times are subtracted to get the total amount of time spent partitioning and executing which is then added to a total timer. After every CYCPRINT, defined as 1000 in *processes.h*, amount of cycles, the program prints the state of the processes to the file corresponding to the system. Once all of the processes have finished executing the final state of the processes are printed to the file as well as the total amount of time taken to execute and partition them.

System Partitioning

In *systemPartitioning(),* the simulator simply iterates through every process in the *set* array every 50 cycles sending them to the memory manager which calls *malloc* to allocate memory to the process. The manager has two *static int*s,*nextID* and *numFinished*, that keep track of the next id that will be allocated and the number of processes that have finished executing. Once the amount of processes that have finished equals NUMPROC the function returns true and the simulation ends after printing out the final data to file.

Dynamic Partitioning

This memory management system keeps a doubly linked list of process that have been allocated memory. After receiving a pointer that points to BLOCKSIZE (100MB) amount of bytes from *malloc()* and creating the list the simulation begins.

Dynamic Partitioning - myMalloc()

myMalloc() first checks if there is enough memory to be allocated if not a null pointer is returned and the process must try again in 50 cycles. If there is enough memory then *myMalloc()* searches the list for a free block of memory using the first fit scheme.

Dynamic Partitioning - myFree()

myFree() sets the block in the memory list to default values and then shifts the empty block to the end of the list. Once it is moved to the end, the coalesce function is called.

Dynamic Partitioning - coalesce()

coalesce() compacts the memory to reduce fragmentation by offsetting the addresses by the removed block's offset.

Static Partitioning

This memory management system keeps a doubly linked list of BLOCKSIZE/PARTITION (100MB/5MB) amount blocks. The memory is split up into single blocks and an allocated process gets the whole block even if the required memory size is much less than the partition size.

Static Partitioning - myMalloc()

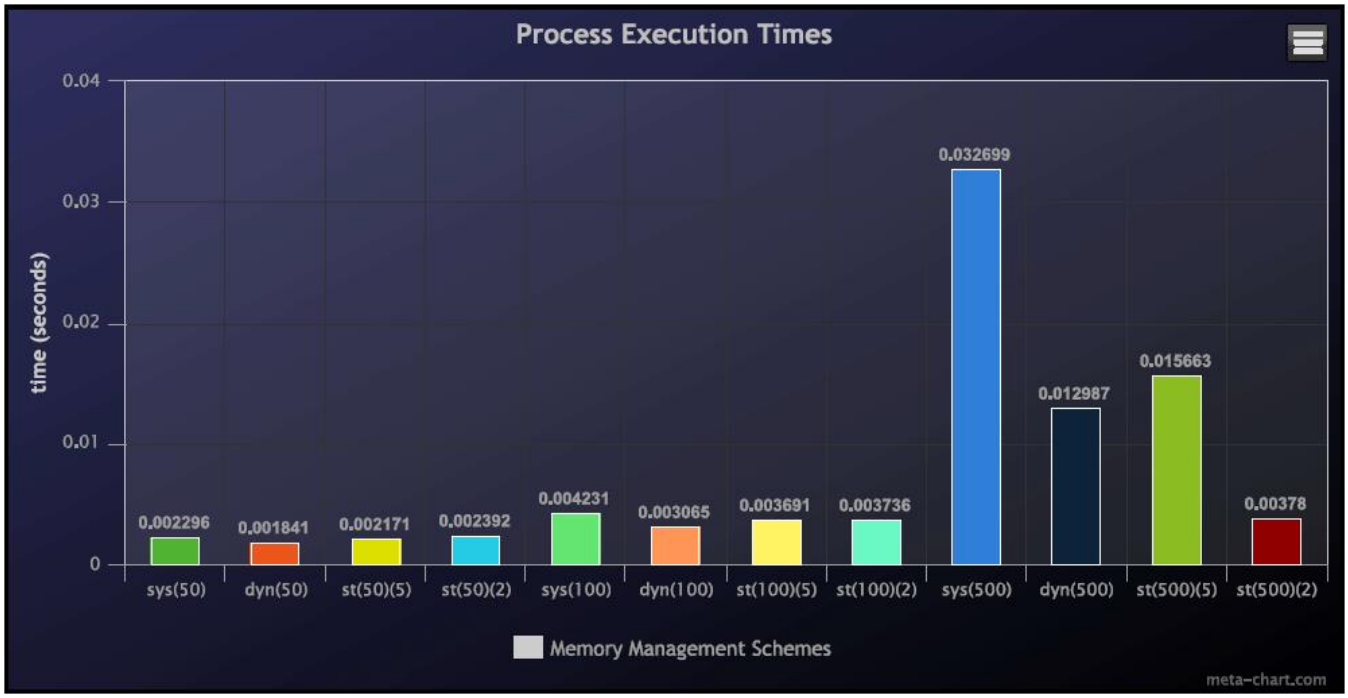
Since, based on the maximum size of the requests, there will always be enough memory available, the management system only checks if there is a free block available. If there is not a free block *myMalloc()* returns a null pointer and the process must try again in 50 cycles. If there is a free block the process receives a pointer to the block.

Static Partitioning - myFree()

myFree() is very simple. It searches for the block that needs to be removed and sets it to default values and frees the process' allocated memory

Results

The memory management systems were tested each with fifty, one hundred, and five hundred processes ten times each and the average of there execution times is taken. Static partitioning is also tested with a partition size of five megabytes and 2 megabytes. These results are shown in the graph below



In almost every case dynamic partitioning had a lower average of execution times than any other memory management system. Static partitioning also was able to get lower average times than the system partition. Surprisingly, static partitioning with a partition size of two megabytes had a much lower average execution time than any other memory management system, even dynamic partitioning. Due to having less context switches than system partitioning the dynamic and static partitioning were able to execute the set of processes faster.

Limitations and Assumptions

The simulation in this project runs under the assumption that the processes run in the order that they are generated in and that all allocated processes run concurrently.

Conclusion

By allocating a large amount of memory at the beginning of the program and partitioning it as needed instead of calling *malloc()* each time new memory is required the programs are able to save context switching time thus reducing the amount of time it takes to execute.