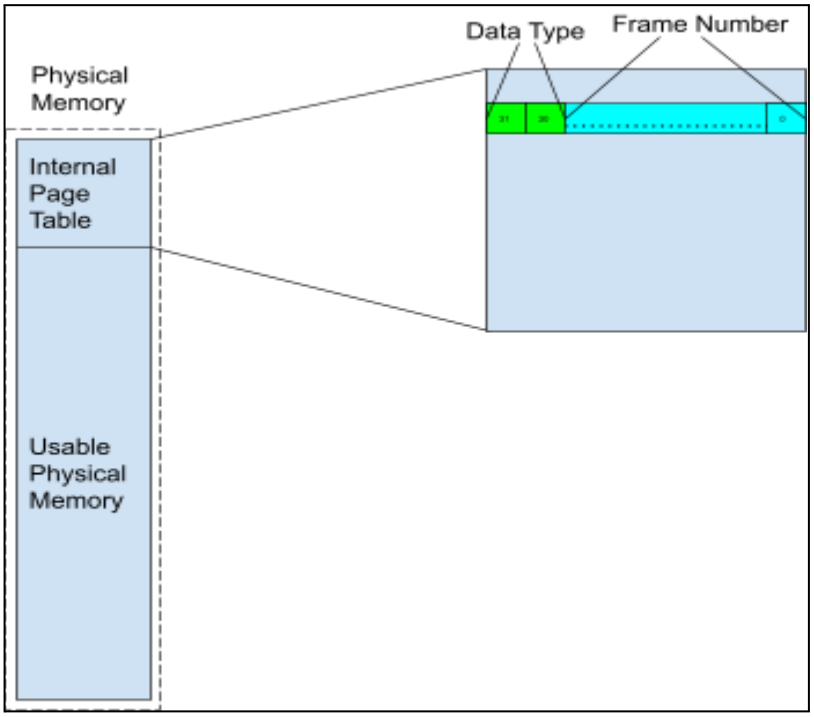
Assignment 05
Creating Memory Management Systems

(A) Structure of the Internal Page Table

The page table is a part of the memory.



The internal page table is maintained as a stack which grows downwards. Each entry in the page table consists of the first two bits as the data type and the rest of the bits as the frame sequence number.

The page table entry is of size 4 bytes (32 bits).
The first two bits are used to encode the data type. This is done so that while reading a frame in the physical memory via the page table we know exactly how many bytes are to be read.

The last 30 bits are used to encode the frame number which points to a frame in the physical memory.

Let the memory size requested by the user be x words ($4x$ bytes).

Let the total number of frames in physical memory be y words ($4y$ bytes).
Then the page table will require y words ($4y$ bytes).

The global stack has an overhead of 4 words (16 bytes) for bookkeeping parameters and 11 words (44 bytes) per stack element. Total overhead = $16 + 44\rho$ bytes

The list of holes has an overhead of 6 words (24 bytes) for bookkeeping parameters and 6 words (24 bytes) per hole. Total overhead = $24 + 24\sigma$ bytes

Now $x = y + y + 4 + 11\rho + 6 + 6\sigma$

$x = 2y + 11\rho + 6\sigma + 10$

We know that the number of stack elements ρ cannot be greater than twice the number of frames in the worst case as a new base pointer may be needed for every new symbol.

$\rho \leq 2y$

We know that the number of holes σ cannot be greater than the number of frames

$\sigma \leq y$

Therefore,

$2y + 11\rho + 6\sigma + 10 \leq 2y + 22y + 6y + 10 \equiv 30y + 10$

$x \leq 30y + 10$

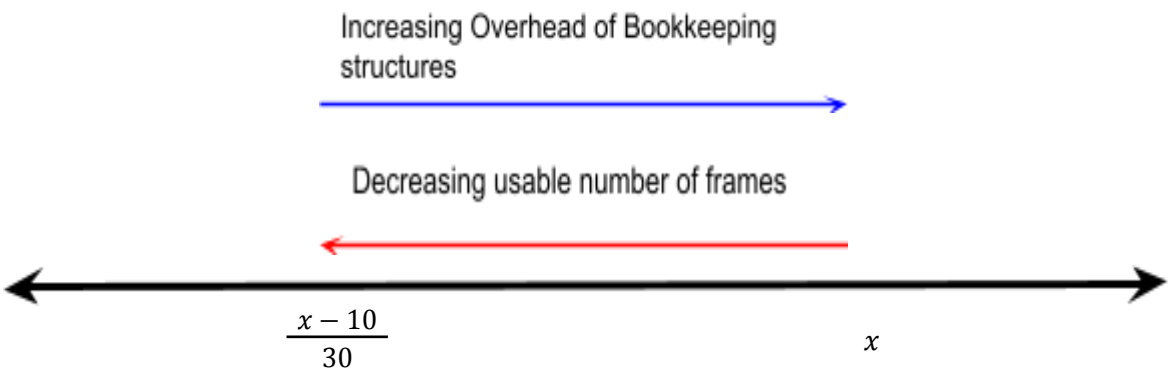
$y \geq \frac{x - 10}{30}$

y cannot be greater than the number of memory words requested.

$y \leq x$

Choosing the minimum y gives the least amount of overhead and restricts the number of frames to y.

$$y = \frac{x - 10}{30}$$



In this report we have chosen y with the least amount of overhead.

$$y = \frac{x - 10}{30}$$

(B) Additional Data Structures or Functions

DEFINES ACCESSIBLE TO END-USER

```
#define FUNCTION_SCOPE  1
#define LOCAL_SCOPE     0
#define EMPTY           0
#define GC_VOLUNTARY    1
#define GC_AUTOMATIC    0
#define LOG_STRUCTURE_UPDATES   0
#define GC_AUTO_SLEEP_T         10000       // 0.01 sec
#define GC_AUTO_COMPAC_GAP      100000      // 0.1 sec
#define GC_AUTO_MAS_GAP         100000      // 0.1 sec
```

FUNCTION_SCOPE encapsulates the data that is available to a function in the global stack.
LOCAL_SCOPE encapsulates the data that is available to a local scope (Example: variable declared in a while loop, is only available in that loop only.)
EMPTY is used when the user initializes a local block. When a function block is initialized with an initStackFrame call, the user must pass the number of parameters to that function. Since you cannot have parameters for a local block, like for a function block, the corresponding argument to the initStackFrame call must be 0, or EMPTY.
GC_VOLUNTARY denotes that the Garbage Collection run is issued by the user.
GC_AUTOMATIC denotes that the Garbage Collection run is issued by the gc_thread that runs it periodically.
LOG_STRUCTURE_UPDATES denotes whether or not updates in the data structures are to be logged to the console.
GC_AUTO_SLEEP_T is the time period for which the garbage collector thread sleeps.
GC_AUTO_COMPAC_GAP is the minimum time gap between two compactions.
GC_AUTO_MAS_GAP is the minimum time gap between two mark and sweep runs.

DEFINES SHADOWED FROM END-USER

```
#define GC_ENABLE               1
#define LEAST_HOLES_PERCENT     0.25        // 25%
#define EVALUATE                1
#define TRACK_WORD_ALIGN        0
```

GC_ENABLE indicates whether or not the garbage collector is to be used.
LEAST_HOLES_PERCENT is the minimum number of holes desired for compaction to be performed.
EVALUATE indicates whether the MMU performance is to be evaluated.

TRACK_WORD_ALIGN indicates whether or not to print whenever word alignment is performed.

```
typedef struct _stack_obj
{
   char _sym_name [31];
   uint32_t _local_add;
   uint32_t _array_idx;
   uint8_t _arrel_flag_param_count;
}
stack_obj_t;
```

The struct **stack_obj_t** data structure that defines an element of the global stack. This structure consists of the following data members:
_sym_name stores the name of the symbol. It can either be a C-identifier or a special symbol to indicate start(end) of new scopes (explained later). It is limited to 31 characters which ensures good coding practice.
_local_add stores the local address of the symbol.
_array_idx stores the index into the array. This field contains a meaningful value if and only if the symbol is part of an array.
_arrel_flag_param_count is a dual symbol, which has different meanings in different contexts.

In the context of arrays, it stores a flag whether or not the symbol is part of the array. It stores the number of parameters that are being passed into the new scope (function).

In the context of special symbols specifically a function scope, it stores the number of parameters being passed into the function scope.

```
class Stack {
    private :
        void error ( const char * type , const char * msg );
        uint32_t _max_size ;
        uint32_t _size ;
        stack_obj_t * _stack ;
    public :
        Stack ( );
        void init ( uint32_t max_size );
        void clear ( );
        uint32_t size ( ) const;
        stack_obj_t & back ( );
        void pop_back ( );
        void push_back ( const stack_obj_t * obj );
```

```
        stack_obj_t & operator[] ( uint32_t i );
        void destroy ( );
        ~Stack ( );
} ;
```

The Stack class is a data type, that enables smooth handling of the global stack.

Private:

It has 3 private data members, which are:
_max_size that restricts the size upto which the stack can grow.
_size that is the actual size of the stack.
_stack which is a pointer to the topmost record of the stack.

error() is a utility method that is internally used by the class to print the error type and message whenever an erroneous situation is encountered (Example: stack is empty but top of stack element is requested).

Public:

Stack() is the constructor method of the class.
init() method takes in one parameter called max_size and is used to create an array of stack of objects of that size and define the **_stack** pointer to point to that array.
clear() method is used to clear up the non-empty stack and set **_size** to 0.
size() returns the current size of the stack.
back() returns the top most element of the non-empty stack.
pop_back() pops the topmost element of the non-empty stack.
push_back() adds a new element to the top of the stack.
operator[]() method eases access of elements by allowing the user to access an element using the index of that element in the stack. The topmost element in the stack has the highest index.
destroy() method frees up the memory occupied by the stack and sets **_stack_** to NULL.
~Stack() is the destructor method of the class it calls the destroy method.

```
typedef struct _list_uint32
{
    uint32_t _frame_no ;
    struct _list_uint32 * _prev ;
    struct _list_uint32 * _next ;
}
list_uint32_t;
```

The struct **_list_uint32_t** is a data structure that defines the element of the list of frames. This structure consists of the following data members:
_frame_no is the frame number of the frame in memory.
_prev is the pointer to previous element of the list.

next is the pointer to the next element of the list.

```
class List_U32 {
    private :
        void error ( const char * type , const char * msg ) const;
        uint32_t _max_size ;
        uint32_t _size ;
        list_uint32_t * _first ;
        list_uint32_t * _last ;

    public :
        List_U32 ( );
        void init ( uint32_t max_size );
        void clear ( );
        uint32_t size ( ) const;
        const list_uint32_t * first ( ) const;
        uint32_t back ( ) const;
        void pop_back ( );
        void push_front ( uint32_t val );
        ~List_U32 ( );
} ;
```

The List_U32 class is a data type that enables smooth handling of the doubly linked list of frames. The interface provided with this class is limited and allows only pop at the back and push at the front, hence resembling the working of a queue data structure.

Private:

It has 4 private data members, which are:
_max_size that restricts the size upto which the list can grow.
_size that is the actual size of the list.
_first which is a pointer to the first frame in the list.
_last which is a pointer to the last frame in the list.

error() is a utility method that is internally used by the class to print the error type and message whenever an erroneous situation is encountered (Example: list is empty but list element is requested).

Public:

List_U32() is the constructor method of the class.
init() method takes in one parameter called max_size and is used to define the **_max_size**.
clear() method is used to clear all the elements in the list.
size() method returns the current size of the list.
first() returns the first element of the list.

back() returns the frame number of the last element of the non-empty list.
pop_back() pops the last element of the list and rearranges the pointers.
push_front() pushes a new element to the front of the list and rearranges the pointers.
~List_U32() is the destructor method of the class.

```
class Min_Heap_U32 {
    private :
        uint32_t max_size ;
        uint32_t size ;
        uint32_t * heap ;

        void error ( const char * type , const char * msg ) const;
        void MinHeapify ( uint32_t current_node );
        void BuildHeap ( );

    public :
        Min_Heap_U32 ( const List_U32 & vals );
        void push ( uint32_t element );
        void pop ( );
        uint32_t top ( ) const;
        uint8_t empty ( ) const;
        ~Min_Heap_U32 ( );
} ;
```

The **Min_Heap_U32** class is a data type that implements a min heap of 32 bit unsigned integers.

Private:

It has 3 private data members, which are:
_max_size that restricts the size upto which the heap can grow.
_size that is the actual size of the heap.
_heap which is a pointer to the root of the heap.

error() is a utility method that is internally used by the class to print the error type and message whenever an erroneous situation is encountered (Example: heap is empty but a heap element is requested).
MinHeapify() takes in the index of the node whose subtree needs to be heapified to restore the heap properties.
BuildHeap() builds the heap from scratch.

Public:

Min_Heap_U32() is the constructor method of the class. It takes a list of frames and based on the frame numbers builds the min heap using the **BuildHeap()** method. The top element of the heap points to the earliest frame.
push() method pushes a new element into the heap.
pop() method pops the least element off the heap.
top() method returns the top (least) element of the heap.
empty() method checks whether the heap is empty or not.
clear() method clears up all the elements of the heap.
~Min_Heap_U32() is the destructor method of the class and it calls the clear method of the heap.

```
class hashnode_uint32_t {
    private :
        uint32_t key ;
        uint8_t avail ;
    public :
        hashnode_uint32_t ( uint32_t v = 0 );
        friend class Unordered_Set_U32 ;
} ;
```

The class **hashnode_uint32_t** is a data type that implements a node in the unordered set. It is a friend of the class **Unordered_Set_U32**.

Private:

It has 2 private data members, which are:
key is the key of the node.
avail denotes whether or not the node is available.

Public:

hashnode_uint32_t() is the constructor method of the class which takes in one parameter that is the key of the node. It sets **avail** to 1 by default.

```
class Unordered_Set_U32 {
    private :
        uint32_t _size ;
        uint32_t _occ ;
        hashnode_uint32_t * _table ;

        void error ( const char * type , const char * msg );
        void ReHash ( hashnode_uint32_t * new_table );

    public :
        Unordered_Set_U32 ( uint32_t table_size );
        void insert ( uint32_t val ) ;
```

```
        uint8_t find ( uint32_t val ) const;
        uint32_t size ( ) const;
        ~Unordered_Set_U32 ( ) { clear() ; }
} ;
```

The class **Unordered_Set_U32** is a data type that implements the efficient unordered set of **hashnode_uint32_t** elements. The underlying implementation uses *hashing with linear-probing*.

Private:

It has 3 private data members, which are:
_size is the size of the hash table.
_occ is the current number of occupied nodes.
_table stores the pointer to **hashnode_uint32_t** element.

error() is a utility method that is internally used by the class to print the error type and message whenever an erroneous situation is encountered (Example: unordered set is empty but element is requested).

ReHash() method is used to rehash a smaller hash table into a large one.

Public:

hashnode_uint32_t() is the constructor method of the class and it takes the size of the hash table as the parameter and defines **_table** to point to the array of **hashnode_uint32_t** elements.
insert() inserts a new node element into the hash table.
find() returns whether or not an element is found in the unordered set.
clear() method clears up the entire hash table.
size() returns the current size of the hash table.
~Unordered_Set_U32() is the destructor method of the class and it calls the clear method.

```
void initializeBookKeepingStructures ( uint32_t y );
```

This function does exactly what it says, initializes all the book keeping structures that help us to efficiently manage the memory. It initializes the global stack and the list of free holes.

```
void setEnv ( uint32_t slp , uint32_t comp_gap , uint32_t mns_gap );
```

This function sets up the environment for the MMU by defining the signal handler for Ctrl+C (SIGINT). It initializes the last compaction and mark-n-sweep timestamps to null. It defines the time periods **GC_AUTO_SLEEP_T, GC_AUTO_COMPAC_GAP**, and

GC_AUTO_MAS_GAP using the user supplied parameters or the default ones. It also sets whether or not the garbage collector is to be run.

```
void evaluate ( );
```

The **evaluate** function calculates the average memory footprint and prints it to the console. It also calculates the average mark and sweep duration and the average compaction duration and prints them to the console.

The memory footprint is calculated by keeping a track of the number of frames occupied in the physical memory at different points of time and then taking an average of them.

$$Average\ Memory\ Footprint\ = \frac{Memory\ occupied}{Number\ of\ samples\ taken}$$

The average mark and sweep duration is calculated by dividing the total duration of the mark and sweep runs by the number of times they have been run.

$$Average\ Mark\ and\ Sweep\ Duration\ = \frac{Total\ Duration\ of\ ALL\ Mark\ and\ Sweep\ runs}{Number\ of\ runs}$$

The average compaction duration is calculated by dividing the total duration of the compaction runs by the number of times they have been run.

$$Average\ Compaction\ Duration\ = \frac{Total\ Duration\ of\ ALL\ compaction\ runs}{Number\ of\ runs}$$

```
void freeMem ( );
```

freeMem frees up all the book-keeping data structures namely the stack and the list of holes which were declared to efficiently manage the memory. It also frees up the entire physical memory itself which was acquired using malloc.

```
void stopMMU ( );
```

It is basically an terminate function which cancels the garbage collection thread, cleans up all the allocated memory by calling freeMem(), destroys the locks, evaluates the memory footprint(if required) and ends the process. This function may also be called voluntarily by the user program just before returning from the main function, or as a result of some error encountered by the MMU (via cleanAndExit), or on a keyboard interrupt (via handleInterrupt).

```
void cleanAndExit ( );
```

cleanAndExit function is called whenever the MMU encounters an error. The MMU prints the error and calls this function to do the necessary clean-up (by calling stopMMU) before exiting the process. There can be many reasons why the MMU may encounter an error, such as when trying to access a variable that is out of scope, trying to access an undeclared variable, going out of bounds in an array, re-declaration of variable, type errors, stack corruption, out of memory etc.

```
void handleInterrupt ( uint32_t sig );
```

This is a signal handler used to handle Ctrl+C (SIGINT) termination signals. It calls the stopMMU function and then calls the default signal handler for SIGINT.

```
void error ( const char * type , const char * msg );
```

This is an utility function that is used to print the error code and the error message to the console.

```
__suseconds_t usecsSince ( const struct timeval * ts );
```

This is an utility function which is used to return the time difference in microseconds that is between now and the time **ts** passed as parameter to this function.

```
uint8_t isValidCIdentifier ( const char * sym );
```

This is a validator which checks whether the symbol name is compliant with C standards and good coding practices. It returns 1 if yes else 0.
Checks performed:

- symbol name must have non-zero length
- symbol name must have length ≤ 30
- symbol name must start with alphabet or underscore and consist of only alphabets, numeric (0-9) or underscore

```
const char * dataType ( uint8_t data_type_num );
```

This is a utility function that returns the data type string corresponding to the data type number.

```
data_type_num = 0 —-> "int"
data_type_num = 1 —-> "med_int"
data_type_num = 2 —-> "char"
data_type_num = 3 —-> "bool"
```

```
uint32_t stackReferenceInScope ( const char * sym );
```

stackReferenceInScope is used to check whether the variable name being used in the user program is valid. It first checks if the variable name is in the local scope or if local scope was not declared it checks until the function scope. This is used at time of **variable creation**, so that the name does not conflict with names within the nearest scope.

```
uint32_t stackReferenceVisible ( const char * sym );
```

stackReferenceVisible is used to check whether the variable name being used in the user program is valid. It first checks if the variable name is in the local scope, if not found then it checks until the function scope. This is used when a **value of a variable is needed**, and the value is chosen such that it is in the nearest scope.

```
void writePageTable( uint32_t k , uint32_t frame_no , uint8_t data_type );
```

writePageTable is used to write the kth page table entry. The 32-bit entry, as described already, consists of a 2-bit data type (least significant 2 bits of data_type) and a 30-bit frame number (least significant 2 bits of frame_no).

```
void writeKthWord ( uint32_t k , uint32_t val );
```

This function is used to write the value (unsigned) to the kth word location in the physical memory.

```
void writeKthWord ( uint32_t k , uint32_t val );
```

This function is used to write the value (signed) to the kth word location in the physical memory.

```
void readKthWord ( unsigned k , void * val );
```

This function is used to read the word from the kth word location in the physical memory.

```
void clearPageTableEntry ( uint32_t k );
```

This function is used to clear the kth page table entry. It is called during the sweep phase of the mark and sweep algorithm.

```
void initStackFrame(uint8_t param_count , uint8_t is_fn_scope);
```

initStackFrame function is used to add a new layer of scope in the global stack. It pushes a special stack record onto the global stack indicating the start of scope. Since

we are writing memory conscious programs, it is the responsibility of the user to initialize the stack frame to create the scopes. It takes two parameters, param_count and is_fn_scope. is_fn_scope can be either **LOCAL_SCOPE** or **FUNCTION_SCOPE**. param_count is used only if the scope is FUNCTION_SCOPE, as parameters can be passed only to a function. LOCAL_SCOPE indicates the start of a local scope while FUNCTION_SCOPE indicates the start of a function.

```
void deinitStackFrame ( );
```

deinitStackFrame function is used to pop the stack frame i.e., remove a scope layer from the global stack. It pops the special record that was pushed during initStackFrame. Since we are writing memory conscious programs, it is the responsibility of the user to eject the stack frame after using it.

```
void gc_run ( uint8_t flag );
```

This function performs the mark and sweep algorithm and then if required also performs compaction.

```
void* gc_thread (void * __);
```

This is the thread which periodically runs the gc_run() function. gc_initialize() is used to create this thread and let it run in the background.

```
void compaction ( uint8_t flag );
```

This function runs the compaction algorithm which compacts the memory, more about this in the next section.

```
void createParam ( const char * type , const char * sym );
```

This function is a wrapper around createVar. This is used to distinguish between variables that are going to be used in the program from the variables that are going to be passed as parameters to the function.

```
void assignParam ( const char * sym , const char * value );
```

This function is a wrapper around assignVar. This is used to distinguish between variables that are going to be used in the program from the variables that are going to be passed as parameters to the function.

```
void createArrParam(const char * type, const char * sym, uint32_t size);
```

This function is a wrapper around createArr. This is used to distinguish between arrays that are going to be used in the program from the arrays that are going to be passed as parameters to the function.

The following functions are defined so that arithmetic, logical and relational operations can be performed smoothly over the symbols defined in the program :

```
int32_t getInt ( const char * sym );
```

This function is used to get the value of an integer variable.

```
uint8_t getChar ( const char * sym );
```

This function is used to get the value of a char variable.

```
int32_t getMedInt ( const char * sym );
```

This function is used to get the value of a medium int variable.

```
uint8_t getBool ( const char * sym );
```

This function is used to get the value of a bool variable.

```
int32_t getArrInt ( const char * sym , uint32_t idx );
```

This function is used to get the value of an element of an int Array.

```
uint8_t getArrChar ( const char * sym , uint32_t idx );
```

This function is used to get the value of an element of an char Array.

```
int32_t getArrMedInt ( const char * sym , uint32_t idx );
```

This function is used to get the value of an element of an medium int Array.

```
uint8_t getArrBool ( const char * sym , uint32_t idx );
```

This function is used to get the value of an element of an bool Array.

(C) Mark and Sweep Garbage Collection & Compaction

Memory Footprints

	DEMO 01	***DEMO 02***	***DEMO 03***
Without Garbage Collection	1.00006e+06 bytes	100010 bytes	430 bytes
With Garbage Collection	100007 bytes	100008 bytes	425.523 bytes

Running Times

	Average Time for Garbage Collection (secs)	*Average Time for Compaction (secs)*	*Total Average Time Overhead (secs)*
DEMO 01	0.142618	0.791481	0.4670495
DEMO 02	0.142115	0.62819	0.3851525
DEMO 03	0.0902563	0.65149	0.37087315

Memory Usage in words without GC

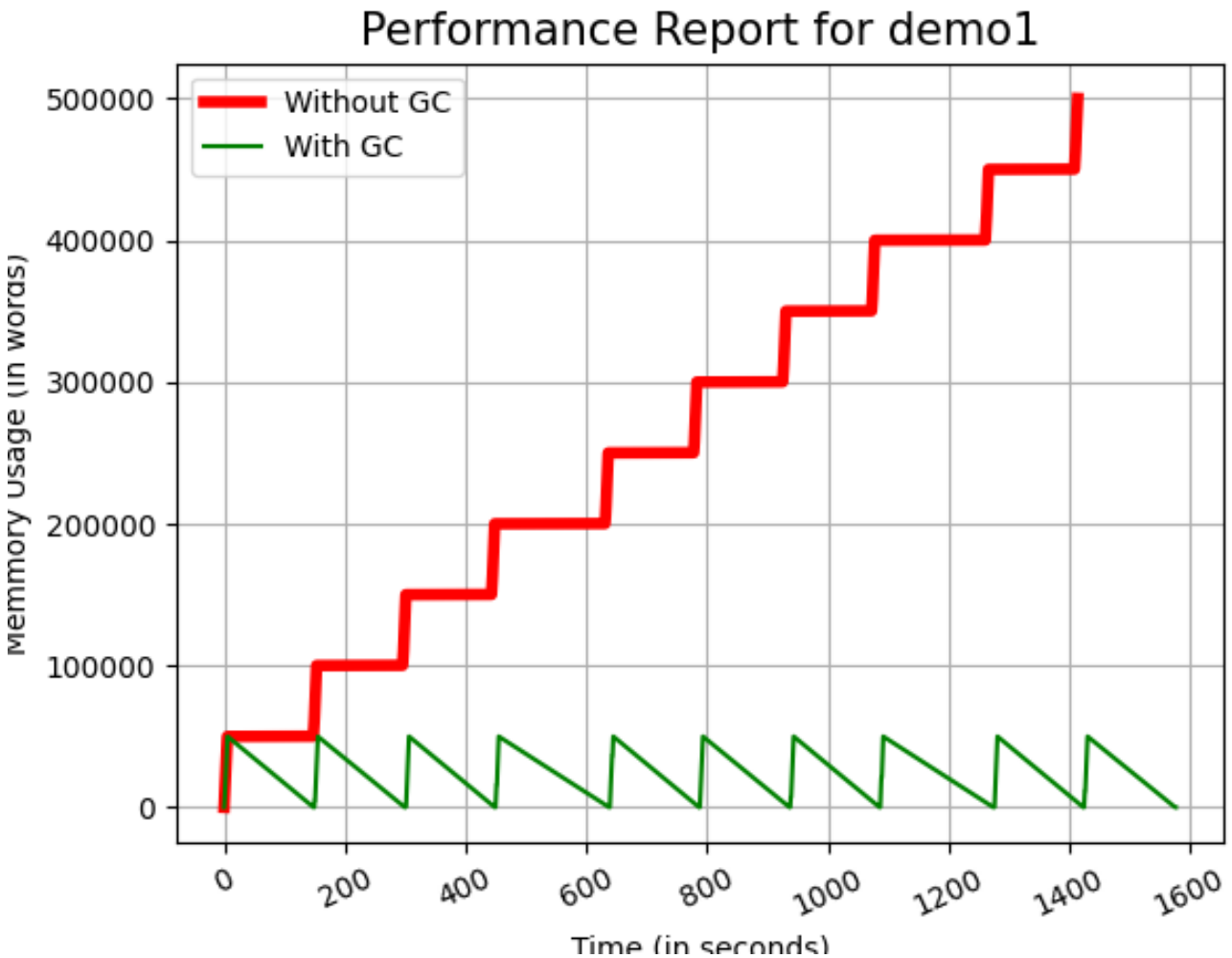
	Maximum	*Average*	*Standard Deviation*
DEMO 01	500030	250016	144345.938
DEMO 02	50005	25003.5	14434.911
DEMO 03	215	108.5	61.775

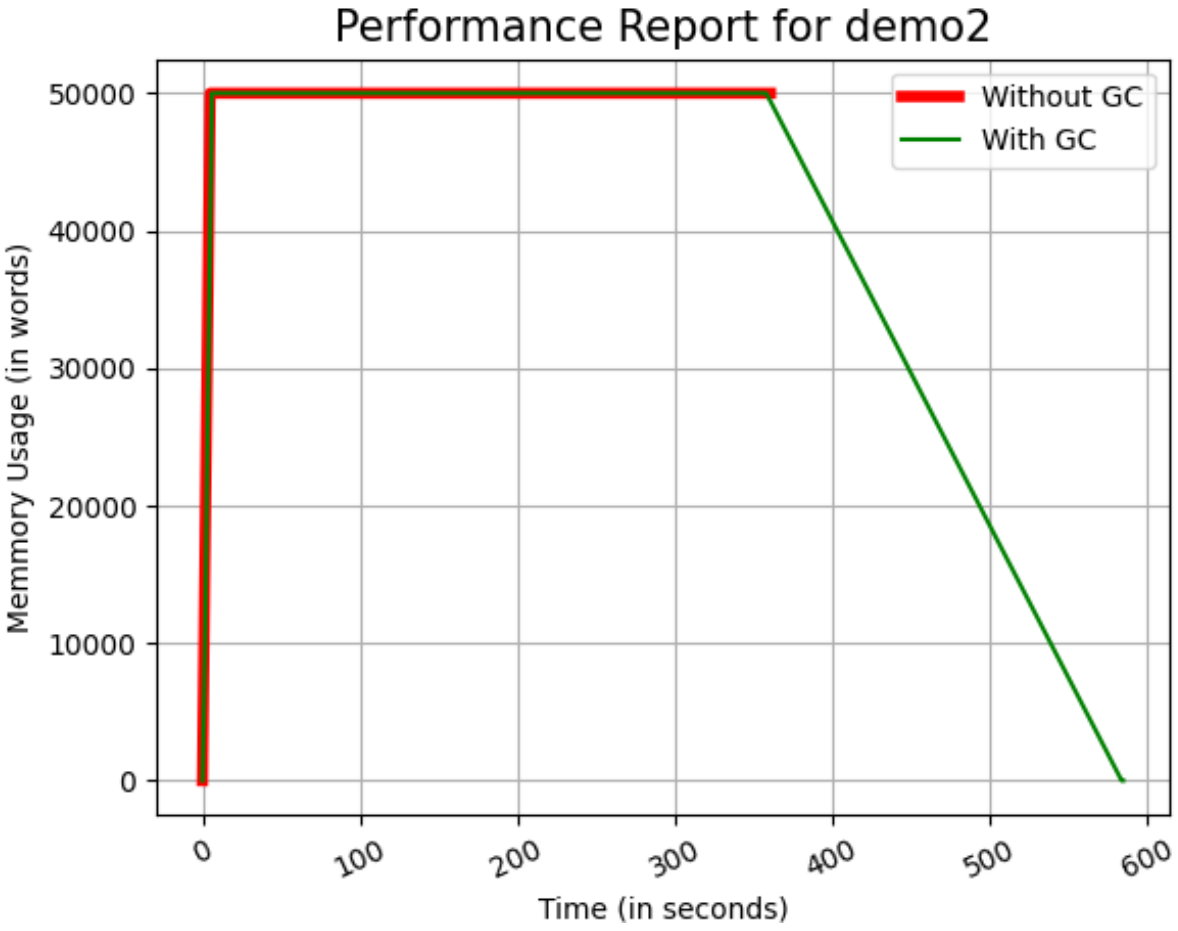
Memory Usage in words with GC

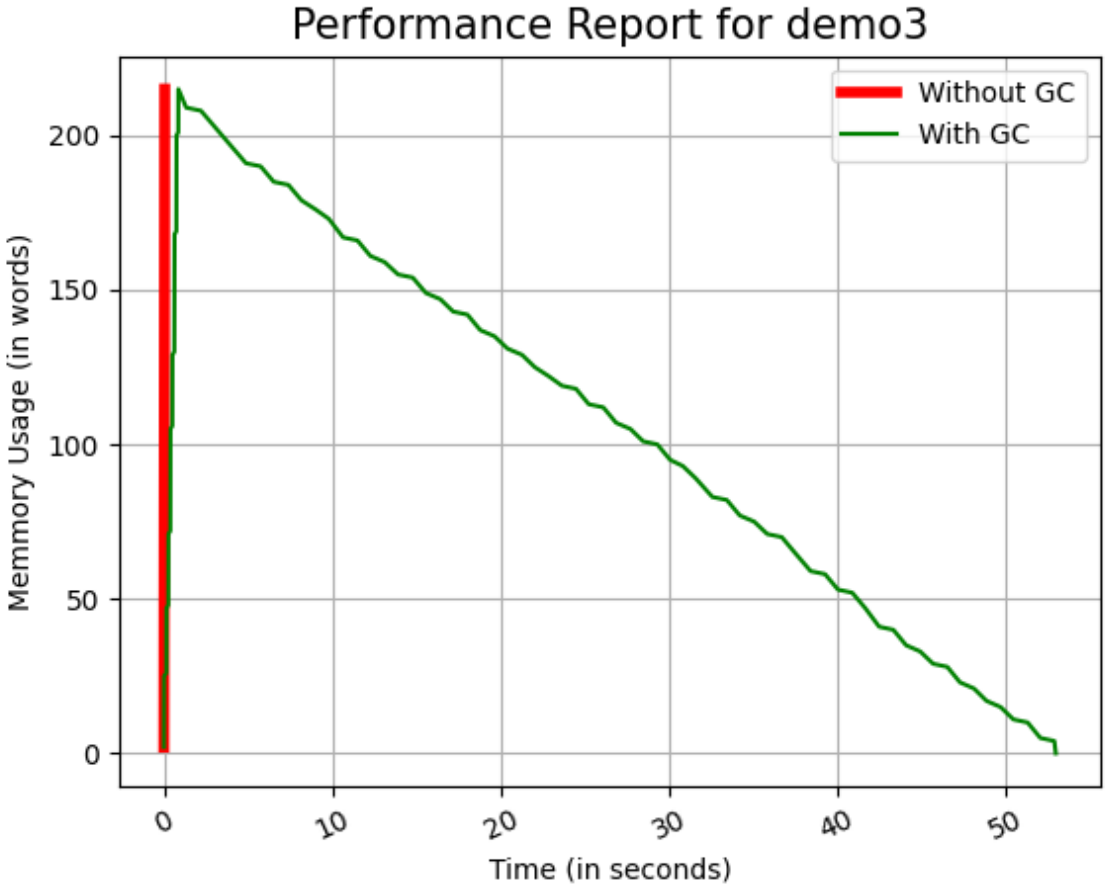
	Maximum	*Average*	*Standard Deviation*
DEMO 01	50005	25001.880	14434.671
DEMO 02	50004	25002.999	14435.777
DEMO 03	215	107.139	61.595

Parameters used to run the demos

	Memory Requested in MB	*GC Sleep Time Period*	*Min Compaction Time Gap*	*Min Mark and Sweep Time Gap*	*Array Size*	*k*	*a,b*
DEMO 01	250	5	0.01	0.01	50000	—	—
DEMO 02	250	5	0.01	0.01	—	50000	—
DEMO 03	250	5	0.00001	0.00001	—	—	9227465, 14930352




Performance Report for demo2
Without GC
With GC
Memmory Usage (in words)
50000
40000
30000
20000
10000
0
0
100
200
300
400
500
600
Time (in seconds)


Performance Report for demo3
Without GC
With GC
Memmory Usage (in words)
200
150
100
50
0
0
10
20
30
40
50
Time (in seconds)

(D) Compaction in Garbage Collection

Compaction brings the CPU to a standstill !!!

The efficiency of the system is decreased in the case of compaction due to the fact that all the free spaces will be transferred from several places to a single place. Huge amount of time (~30s) is invested for this procedure and the CPU will remain idle for all this time. Despite the fact that the compaction avoids external fragmentation, it makes the system inefficient.

Solution: Perform only when needed.

We suggest that instead of performing compaction every time the garbage collector is run, do it only when the percentage of holes (free frames) is above a certain limit. This means that there may be a lot of external fragmentation and compaction will make the system efficient thereafter.

Also do not let compaction occur too frequently, set a minimum gap of time between two compaction runs.

Better Solution: Let the user decide.

All is well and good if the user is satisfied. So why not let the user decide? We proposed a limit in the above solution in the form of the minimum time gap between any two runs of the compaction algorithm; this limit can be set by the user himself.

So an educated user depending on the program they write can decide the frequency of compaction, while they can always use the default values provided by the users.

Algorithm

The algorithm for performing compaction is given below as a pseudo code.

```
function do_compaction ( )
{
  last_page_table_entry_idx <- 0
  while ( frameNum(last_page_table_entry_idx) != 0 )
     last_page_table_entry_idx += 1

  if ( ! last_page_table_entry_idx )  return

  holes_min_heap <- to_min_heap(LIST_OF_HOLES)
  compacted_phys_seg_end_frame_no <- PAGE_TABLE_SIZE + last_page_table_entry_idx - 1

  for ( virtual_add : from 0 to last_page_table_entry_idx - 1 )
 {
     page_table_entry <- readKthWord(virtual_add)
     old_frame_no <- extractFrameNum(page_table_entry)

     if ( old_frame_no < compacted_phys_seg_end_frame_no ) continue
```

```
    new_frame_no <- holes_min_heap.get_min()

    data_type <- extractDataType(page_table_entry)
    frame_content <- readKthWord(old_frame_no)

    writePageTable(virtual_add, new_frame_no, data_type)
    writeKthWord(new_frame_no, frame_content)

    holes_min_heap.pop_min()
    holes_min_heap.insert(old_frame_no)
  }
  LIST_OF_HOLES <- to_list(holes_min_heap)
}
```

Compaction should re-distribute the frames such that the block of allocated frames is contiguous and also condensed at one end of the whole memory. Clearly, if any page table entry points to a frame that lies inside that block of frames, then we need not change anything. But if it lies outside the block, then we can allocate the free frame with the smallest index (word-wise in the physical memory) as the new frame to which that entry points to. So, the page table entry should be changed to point to the new frame, the content of the old frame should be copied to the new frame, and the old and the new frames should be inserted and removed respectively from the collection of free frames.

So what are the functionalities that we need from this *collection* of free frames? We should be able to extract the smallest frame number. We should be able to delete the smallest frame number. We should be able to insert any frame number into the collection. Hence, an obvious choice of the data structure to store the collection of free frames in the compaction algorithm is a *min-heap*.

Let $p = number\ of\ (valid)\ entries\ in\ the\ page\ table$
$h = number\ of\ free\ frames\ in\ the\ physical\ memory\ at\ the\ time\ of\ compaction$
$t = time\ to\ read\ from\ or\ write\ to\ the\ physical\ memory$
then the worst-case time complexity of the algorithm is
$T_{compaction} = O(h + p.t + p.log(h))$

Assume that t is a small constant. Also, since the size of the physical memory is not more than $4GB$ or $10^9 words$, $h \leq 10^9$ and $log_{10}(h) \leq 9$.

Hence, $T_{compaction} = O(h + p)$.

As already explained, the physical memory is divided into a block of actual physical frames that store the data content and a page table that acts as an index to the actual frames. Therefore, the sum $h + p$ is upper bounded by the size of the physical memory (in words), let that be M.

So, in the worst case $T_{compaction}$ can be $O(M)$.

(E) Locks

```
pthread_mutex_t __compaction_lock__ ;
```

```
pthread_mutex_t __marknsweep_lock__ ;
```

A lock __compaction_lock__ has been defined, to lock down the user process during compaction. This lock comes into play only during the compaction part of the garbage collection phase. This is because compaction accesses overwrites the memory into a compact form, and adjusts the page table pointers. In fact it redistributes the frames in the memory so that all the occupied frames cover a contiguous block of memory in the start. The user processes cannot be allowed to interfere with this *re-shuffling* procedure, by trying to make a new page table entry or writing to (reading from) a page in the memory. Thus, the CPU is brought to a standstill, and when compaction is over, the user processes are resumed.

No memory-access related functionality should be available during the compaction downtime and hence __compaction_lock__ is used. But during the mark-and-sweep garbage collection, it is easy to understand that some functions like *assigning* value to a variable and *getting* value of a variable should be allowed, and others like *creating* variables (scalars or arrays) and *freeing* variables should not be allowed. Therefore, we need another lock, __marknsweep_lock__, that is able to discriminately provide mutual exclusion between garbage collection and only some specific functionalities, in contrast to compaction, that should be mutually excluded from absolutely all the functions.
__marknsweep_lock__ mutually excludes the mark-and-sweep garbage collection task with the following tasks.
(1.) compaction
(2.) creating a(n) scalar/array variable
(3.) freeing a variable and removing its corresponding page-table entry