Purpose:
To gain insight into modern day OS memory management techniques.

Introduction:
Almost every modern day OS provides virtual memory support. DOS memory management is not sophisticated enough to handle the memory requirements of modern day applications. As physical memory (DRAM) falls in price, many have predicted that virtual memory would no longer be necessary. However, this has not been the case and it appears that application and user memory requirements continue to surpass the common memory configurations found in many of today's systems.

Virtual memory gives the application the illusion of a large logical address space. However, this address space is only partially provided in physical memory. Applications, in fact, are typically limited to address spaces that are much smaller than the logical address space. In most systems, the logical address space is 2^{32} or 4 Gigabytes. Physical memory sizes (installed DRAM) range from 8 MegaBytes to 128 Megabytes on many personal computer systems.

In order to allow programs that are larger than physical DRAM to run and/or to allow multiprogramming, most OS's allow the physical address space to be extended using cheaper disk memory. A swap disk is a special disk partition (linux) or file (in Win95) that is used for this purpose. Swap disks range in size from 1 Megabyte to 1 Gigabyte and larger, although most systems provide a swap disk that roughly doubles the size of installed DRAM. For example, it is common for a personal computer to be shipped with 32 MBytes of DRAM. The Win95 Swap file size is typically set between 16 and 64 MBtyes. You should note that programs that require space in excess of physical DRAM + Swap size can NOT be run on these systems even though their logical address space is 4 Gigabytes in size.

Efficient memory allocation/deallocation and protection are the most important issues of a memory management system. It is NOT possible to provide protection, that is, the isolation of one process from another, without some form of hardware support (unless of course your OS traps every memory reference made by a process). Virtual memory complicates memory management software since a mechanism must be in place that will move processes or portions of processes between physical DRAM and the swap disk. This too requires help from the hardware in order to make it efficient and worthwhile.

Since fixed or dynamic variable-size block allocation strategies are usually too expensive to implement in today's systems because of overhead, most systems provide hardware support for paged implementations of virtual memory systems. For example, the Pentium provides hardware for a sophisticated logical to physical address translation scheme based on segmentation and a 2-level paging scheme that additionally supports protection. However, the simulation of such a system is somewhat complex. Therefore, in this project, you will implement a simulator for a system that supports only a 2-level paging system. In order to give you a solid understanding of virtual memory management, your implementation will support a logical address space that is larger than the physical address space. This system will allow you to evaluate the effectiveness of the Aging modification of the Not Frequently Used page replacement algorithm.

(Note: My original plan was to have you build a simulator for a segmentation system with paging. I gradually simplified the specification to a 2-level paging system as I coded the simulator over a two day period. You may have printed a version of the project specification in which I have used inaccurate terminology in the figures (e.g. references to segment numbers and segment tables, etc) which were part of the original specification. The specification given below corrects this terminology. If you have started the project, you will be pleased to know that the implementation details of the original specification have NOT changed.)

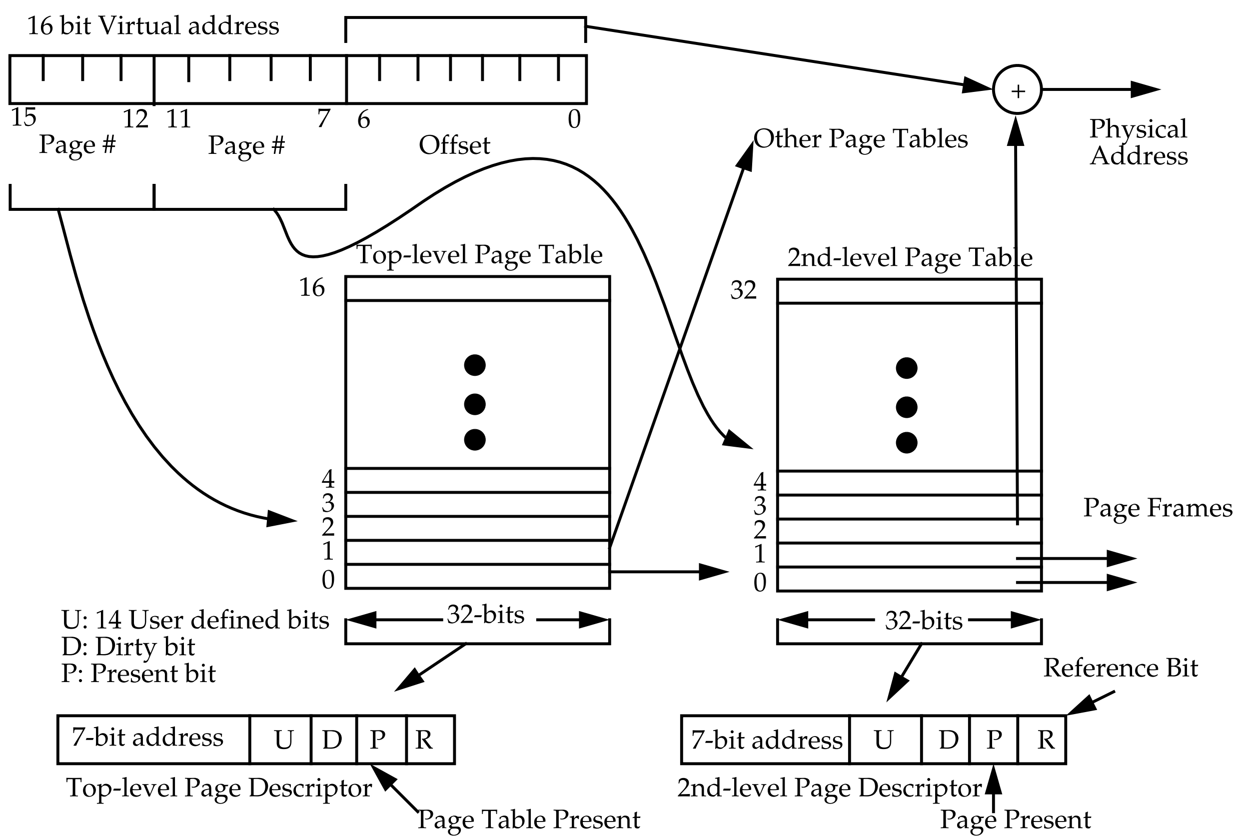
Description:
Since you have already implemented a process scheduling simulator, a natural extension of this project is to incorporate process memory references and system calls for allocation and deallocation of memory. You can reuse the code that you have already written which implements the Round Robin scheduling algorithm. You need to remove multilevel queueing and use a single queue (all processes have equal priority in this assignment). If you do not have 100% confidence that your code is correct, then you should use the process scheduling simulator that I have provided for you.

Your simulator will simulate a system in which virtual addresses are 16 bits wide. This yields a **64K** virtual memory space. However, physical memory will be limited to only **16KB** (1/4th of the virtual memory space). Page size is **128** bytes yielding **128** pages of physical memory (page frames).

The 4 most significant bits of a virtual address are an index into the Top-level page table. Each of the 16 page descriptors in this table are 4 bytes wide. Therefore, each Top-level page table occupies only 64 bytes of a 128 byte page. Since the Top-level page table will be placed in a page of its own, the rest of the space in the page is unused.

The next 5 bits of the virtual address are an index into a 2nd-level page table. Each of the 32 page descriptors in this page table are also 4 bytes wide. Therefore, a 2nd-level page table occupies a single page of physical memory.

The remaining 7 bits of the virtual address compose the offset which, when combined with the page frame given by the 2nd-level page table, form the physical address.



The Page descriptors for both tables are identical in structure. For the Top-level page descriptor, the 7-bit address is the page frame address of the page table when it is memory. For the 2nd-level page descriptor, the 7-bit address is the page frame address of the virtual page. Note that the low order 7 bits of these address are 0 allowing any page in the 16K of physical memory to be addressed.

The 14 User defined bits, the Dirty bit, the Present bit and the Reference bit will be used in the page replacement algorithm to keep track of usage and to determine if the page needs to be saved to the swap device before being overwritten. For the Top-level page descriptor, the Present bit indicates whether or not the page containing the page table is in memory. For the 2nd-level page descriptor, P indicates whether or not the page is in memory.

User processes will submit memory requests as messages in your system. Whenever a process submits a message, it will suspend itself and, on the next clock cycle, a server task (the Memory Server Task (MST)) will run and carry out the request (process the message), if it is not sleeping on a previous request. There are three types of messages the MST should handle:

Create Segment:
This request is submitted when a process arrives. Each process will have 1 segment, a data segment, described by a Top-leve page table. The MST will attempt to allocate a page frame for the Top-level page table. If successful, the page frame allocated should be locked into memory (non-swappable) and the requesting process placed back on the FRONT of the readyQ. If all page frames are occupied, then the MST should run a page replacement algorithm (the **Aging** modification of the **Not Frequently Used**) and free up a page frame. The freed page frame is then allocated to the suspended process and the suspended process moved to the BlockedQ. Finally, the MST sleeps for one or two clocks depending on whether or not the page selected for replacement was dirty.

Destroy Segment:
When a process terminates, it will submit a request to destroy its segment. The MST should update the data structures and return. There is never any blocking that occurs on the part of the MST for deallocations. Therefore, MST will not have to sleep for this case.

Page Fault:
In a **demand paged** system, a process generates virtual addresses into its data segment and page faults its working set of pages into page frames. If the page table or page to which the reference is made is not mapped into a page frame, a page fault occurs. A message to the MST should then be generated and the faulting process should suspend itself. When the MST runs, it should determine if a free page frame exists, and if so, allocate it to the suspended process. The suspended process should then be put onto the FRONT of the ReadyQ. If no page frames are available, then a page frame must be selected for replacement using the **Aging** modification of the **Not Frequently Used** page replacement algorithm. Similar to the Create Segment request, the requesting process is then allocated the freed page frame and moved to the BlockedQ. Finally, the MST sleeps for one or two clocks depending on whether or not the page selected for replacement was dirty.

As noted above, if requests are pending and the MST is not sleeping then the schedular should run for the next clock cycle. In other words, the MST should have highest priority when requests for memory are pending.

The general format of the request is shown below:

PID	Request Type	Size	Top-level Page Index	2nd-level Page Index
0->n	0->3	0->b	0->15	0->31

PID: Process identifier.
Request Types: 0: Create segment, 1: Destroy segment, 2: Page fault.
Size: Given in bytes (valid for Request types 0)
Top-level Page Index: For page faults, this is the index of a page descriptor in the Top-level page table.
2nd-level Page Index: For page faults, this is the index of a page descriptor in the 2nd-level page table.

The following pseudocode should be used to design your simulator:

```
clock = 0;
add arrival time 0 PCBs to suspendQ, queue up data segment allocation request
clock++;
while( there are processes )
   {
   cur_job = GetCurrentPCB(...)/* Note: cur_job should be NULL if no PCBs to
   if ( cur_job == NULL )         run or if a memory request message is
      {                           present and MST is not sleeping. */
      If (message queue non-NULL && MST not sleeping) call MST(...)
      Else if ( sleep > 0 ) sleep--
      Add new jobs to suspendQ, queue up data segment allocation request
      Check for I/O completion of blocked processes.
      clock++
      continue
      }
   set status of cur_job to running
   while(1)
      {
      Add new jobs to suspendQ, queue up data segment allocation request
      Check for I/O completion of blocked processes.
      If (cur_job’s remaining time is 1)
         mark it complete, queue destroy segment request
      If (cur_job is running)
         {
         Number of refs = GetNumMemRefs()
         for ( i = 0; i < Number of refs; i++ )
            {
            if ( cur_job->page_fault == False )
               lastref = RandMemAccess(lastref, data_size, locality_counter++)
            else cur_job->page_fault = False; /* Restart last reference. */
            if (PagePresent(cur_job, lastref, &tl_pnum, &sl_pnum) == FALSE)
               { If ref outside segment (>= limit), kill process
                 Else
                    queue up PageFault message
                    cur_job->page_fault = True
                 break
               }
            }
         If (page fault)
            mark cur_job suspended
         }
      If (cur_job is running)
         If ( I/O call ) mark cur_job blocked
      If (sleep > 0) sleep--;
      if (cur_job is running)
         If (Time Slice expiration)mark cur_job ready
         If (message queue not empty && MST not sleeping)mark cur_job ready

      Update stats
      clock++;
      break inner loop if job is not running - Add job to the appropriate Q
      }
   }
```

You will need at least the following data structures.

Page Frame Table

	Free/ Allocated	PID	Logical Page Num (high order 9 bits)	Page Table pointer (SPDescriptor *)
0	T/F	0->n	0->512	
		⋮		
127				

This table will be used by the MST to determine which pages of physical memory are available.
It will also be used by processes to map logical to physical pages.
It will also come in handy for debugging.

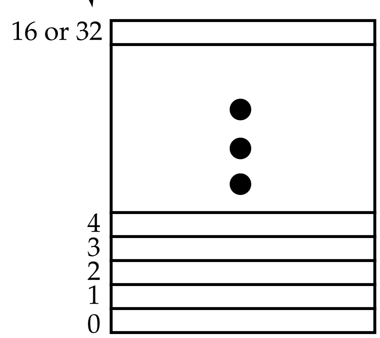
The Free/Allocated field indicates whether or not the page frame is in use.

The PID field is the process ID of the process that is allocated the page frame.

The logical page number is the logical page (high order 9 bits) of the process that uses this page frame. Use a special value for Top-level and 2nd-level page tables.

Segment Base Table

	PID	Data Base	Data Limit
0	0->n	0->127	0->64K
	⋮		
200			



Top-level or 2nd-level page table is an array of SPDescriptors defined in the typedefs that I have given you below.

Page Table pointer will point to the contents of the page frame when the page frame contains a Top-level or 2nd-level page table.

This table will be used to look up the page frame addresses of the Top-level page tables. These page frames are locked into memory. The limit field should be interpreted as the size of the segment in bytes.

The Page Frame Table is 128 elements in size. You can assume that the Segment Base Table is 200 elements long. Therefore, both of these tables can be allocated at program start-up using array definitions.

The data types that you use for the segment and page descriptors can be standard C integer data types such as 'int' and 'long'. I have provided typedefs below to help you with this part. For example, I have used 'int' as the type of all fields in the Page descriptors. However, it is up to you to enforce valid values, e.g. the 7-bit page frame address must have values in the range of 0 to 127 only. The Dirty and Present fields should have true/false values, etc.

You will need to do some bit operations, such as shifting the U bits for the aging algorithm. This is easily accomplished using the operators >> and << in C.

In order to extract the Top-level page number, the 2nd-level page number and offset from a process generated logical address, you can use the & bit operator. For example, to extract the page number, you might use:

```
logical_address = 0x438C;
sl_pnum = (logical_address & 0x0F80)>>7;
printf("%x\n", page_num);
```

This will remove the Top-level page number and offset and return the 2nd-level page number. For example, if logical_address = 0x438C, the sl_pnum extracted is 0x7 using the above expression. In order to print hexadecimal numbers, use the %x specifier in printf.

Whenever a process generates an logical address, you can translate it to a physical address using the following procedure (and described in detail in the PagePresent pseudocode below): Look up the Top-level page frame address in the Segment Base Table (see the figure on the previous page). Use this index to locate the Page Frame array element in the Page Frame Table. The contents of this page frame are given by the Page Table pointer field. This field points to a Top-level page table created by the MST in response to a Create request. The Top-level page tables can be created in MST by using calloc to create an array of SPDescriptors, e.g.,

```
  PageFrameTable[i].SP_table = (SPDescriptor *)calloc(16, sizeof(SPDescriptor)
```

The Top-level page number of the logical address can then be used to index this table as

```
    PageFrameTable[i].SP_table[tl_pnum]
```

to get you to the correct Top-level page descriptor. You can then use the page_frame field of the descriptor to obtain the page frame index back into the Page Frame Table for the 2nd-level page table. The contents of the 2nd-level page table are also given by the Page Table pointer field. In this case, the pointer points to an array of 32 SPDescriptors that are created by MST for a page fault request.

Use these as specified:

```
#define TIME_SLICE 3
#define CHANCE_OF_IO_REQUEST 10
#define CHANCE_OF_IO_COMPLETE 4
#define MAX_NUMBER_OF_REFERENCES 25
#define LOCALITY_JUMP 10
```

Same as previous project.

```
Boolean IORequest()
   {
   if ( rand() % CHANCE_OF_IO_REQUEST == 0 ) return True;
   else return False;
   }

Boolean IOComplete()
   {
   if ( rand() % CHANCE_OF_IO_COMPLETE == 0 ) return True;
   else return False;
   }
```

New functions:

```
Boolean SetDirtyBit()
   {
   if ( rand() % CHANCE_OF_WRITING == 0 ) return True;
   else return False;
   }

int GetNumMemRefs()
   { return rand()%MAX_NUMBER_OF_REFERENCES; }
```

Use this function by passing in the last reference this process has made, the size of the data segment and a counter value private to each process. Both the last reference and counter must be initialized to 0 when the PCB enters the simulation. All three pieces of information should be kept in the PCB of the process.

```
int RandMemAccess(last_ref, data_size, locality_counter)
   int last_ref, data_size, locality_counter;
   {
   static int sign = 1;
   sign = (sign == 1) ? -1 : 1;
   if ( locality_counter % LOCALITY_JUMP == 0 )
      last_ref += sign*(rand()%data_size);
   else
      last_ref += sign*(rand()%(data_size/50));
   if ( last_ref < 0 ) last_ref *=-1;
   if ( last_ref > data_size ) last_ref %= (data_size + 10);
   return last_ref;
   }
```

Similarly, MST might be specified as follows:

```
MST(...)
   {
   Get memory request from message queue;

   switch ( request type )
      {
      If ( Create )
         Find free page in Page Frame Table
         If (no free page frames exist)
            SwapPage(...) and try again
         Update Page Frame element, pid,
            special value for logical_page_num to lock page in memory
         Allocate the Top-level page table (calloc 16 SPDescriptors)
            in Page Frame element
         Find a free element in the Segment Base table and update
         break
      If ( Destroy )
         Free all Page Frame Table elements belonging to this process
         Free Segment Base table entry
         break
      If ( PageFault )
         {
         Find free page in Page Frame Table
         if (no free page frames exist)
            SwapPage(...) and try again
         Set allocated and pid values in Page Frame Table
         Get pointer to Top-level page table using Segment Base Table
         if ( message request indicates Page Table fault )
            set logical_page_num to special value (in Page Frame Table)
            calloc 2nd-level page table if necessary (32 SPDescriptors)
            set present bit and page_frame in Top-level page descriptor
         else /* Page not present */
            set logical_page_num to tl_pindex*32 + sl_pindex
            set present bit and page_frame in 2nd-level page descriptor
          break
         }
      }
   If (Request type is not Destroy) /* Note: completed processes are on
                                       finished queue. */
      If ( SwapPage() NOT called )
         Find PCB on suspendQ and move to FRONT of ReadyQ;
      else
         Find PCB on suspendQ and move to BlockedQ;

   }
```

The following is pseudocode to help you with several functions.

```
SwapPage(...)
   static long called = 0;
   called++;
   if ( (called % NUM_MEM_FAULTS) == 0 )
      AgePages();
   Search all page tables in order BlockedQ, ReadyQ and SuspendQ.
      Start at end of ReadyQ and work backwards
   Consider all logical pages of each process before considering ANY page
      tables for replacement.
   Search Top-level page tables from 0 to 16 and 2nd-level page tables
      from 0 to 32.
   Define a 16-bit counter as the reference bit concatenated with
      the 14 user defined bits concatenated with the dirty bit.
   If 16 bit counter is 0, select page
   else
     Save pointer to lowest val and keep looking in queues(save 1st for ties)
   Update page descriptor as not present, free Page Frame Table element
   If ( dirty ) sleep++
   sleep++
   }
AgePages() /* Straight out of the text */
   {
   Scan Page Frame Table, if Page Table pointer field non-NULL, then
   Shift right 'user_bits' of SPDescriptors then OR 'referenced' into bit 13.
   Clear referenced bit
   }
Boolean PagePresent(pid, next_ref, seg_num, page_num)
   {
   Find the segment base array element in the Segment Base Table
   Check next_ref against the Data Limit field
   If greater, than segmentation violation - return and kill the process

   extract 4 bit Top-level page number and 5 bit 2nd-level page number

   Get pointer to the Top-level page table in Page Frame Table using the
      data_base index value in the Segment Base Table
   Update the referenced field in Top-level page descriptor

   Determine if 2nd-level page table is present by checking the Present Bit
   If not present, then generate a page fault - missing Page Table
      This action will then cause a MST message to be queued.

   Update referenced field in 2nd-level page descriptor
   Get pointer to the Page Table using the Top-level page descriptor
      page_frame value
   If not present then generate a page fault, missing logical page
      This action will also cause a MST message to be queued.

   Otherwise, reference does not cause a page fault
   Set the dirty bit in the Top-level page descriptor for this page table.
   Randomly set the dirty bit in the 2nd-level page descriptor using
     SetDirtyBit()
   }
```

My Declaration that you may or may not find useful:

```
#define CHANCE_OF_IO_REQUEST 10
#define CHANCE_OF_IO_COMPLETE 4
#define MAX_NUMBER_OF_REFERENCES 25
#define CHANCE_OF_WRITING 20
#define LOCALITY_JUMP 10
#define TIME_SLICE 3
#define NUM_PAGE_FRAMES 128
#define SEG_TABLE_SIZE 16
#define PAGE_TABLE_SIZE 32
#define PAGE_SIZE 128
#define MAX_NUM_PROCESSES 200
#define NUM_MEM_FAULTS 5
#define Free -1
#define Allocated 1
#define PageTableValue -1
#define SegTableValue -2

typedef enum { None, Running, Ready, Blocked, Suspended, Completed, Crashed
} Status;
typedef enum { False, True } Boolean;
typedef enum { Create, Destroy, Increase, Decrease, PageFault } RequestType;

struct Message
   { int pid;
     RequestType req_type;
     int size, tl_pindex, sl_pindex;
     struct Message *next; };
typedef struct Message *MQueue;

typedef struct
   { int page_frame, user_bits, dirty_bit, present_bit, referenced;
   } SPDescriptor;

typedef struct
   { Boolean free_alloc;
     int pid, logical_page_num;
     SPDescriptor *SP_table;
   } PageFrameEle;

typedef struct
   { int pid, data_base, data_limit;
   } SegmentBaseEle;

struct PCB
   { int pid, arr_time, ser_time, data_size, run_clocks, blocked_start_clock;
     int start_slice, blocked_clocks, wait_start_clock, wait_clocks;
     int suspend_clocks, suspend_start_clock, finish_clock, last_ref;
     int locality_counter, num_page_faults;
     Boolean page_fault;
     Status status;
     struct PCB *next; };
typedef struct PCB *PCBQueue;
```

The input data file is similar to that used on the previous project:

Process ID	Arrival Time	Service Time	Data Size
0->n	0->t	1->t	1->b

Note that the data segment sizes is in units of bytes.