Lab 2: Windows Working Set

The instructions for Lab 2 are described below. There is no file hand out accompanying this lab. The base WRK source and the virtual machines provide everything you need. You have two weeks to complete this lab (due midnight October 20, 2010). Do not procrastinate. This is a difficult lab. Instructions for submitting your assignment via the submit system will be posted by the TA's in the forum.

1 Lab Overview

The working set is an important concept in memory management. The Windows® operating system often divides a working set into a process working set and a system working set, respectively, to trace the physical memory usage of each process and system. Operations on the working set in the Windows kernel are divided into the working set trimming algorithm of the working set manager (system level) and the page replacement algorithm (process level). The working set trimming algorithm scans the memory usage of the system periodically and trims the working set (that is, reduces the number of allocated pages) of some processes. For example, it can select a process with a low priority, and apply the least recently used (LRU) algorithm to select the page to delete. The page replacement algorithm is used per process. If the process requests an additional page after the process requisition page total exceeds a certain peak value, the working set size no longer increases and the existing pages are replaced using a page replacement policy.

This project analyzes the working set page replacement of Windows. There are two tasks:

1. Analyze the peak value of the process working set and the page replacement algorithm.

2. Modify the replacement algorithm and analyze the effects.

2 Data Structures Related to the Working Set

EPROCESS is the data structure that describes the process. Other data structures related to the working set can be found from EPROCESS. The central data structures related to the working set are MMSUPPORT, MMWSL, MMWSLE, MMWSLENTRY, MMPTE, and PMMWSLE_HASH. The relationship between these structures is shown in Figure 1.

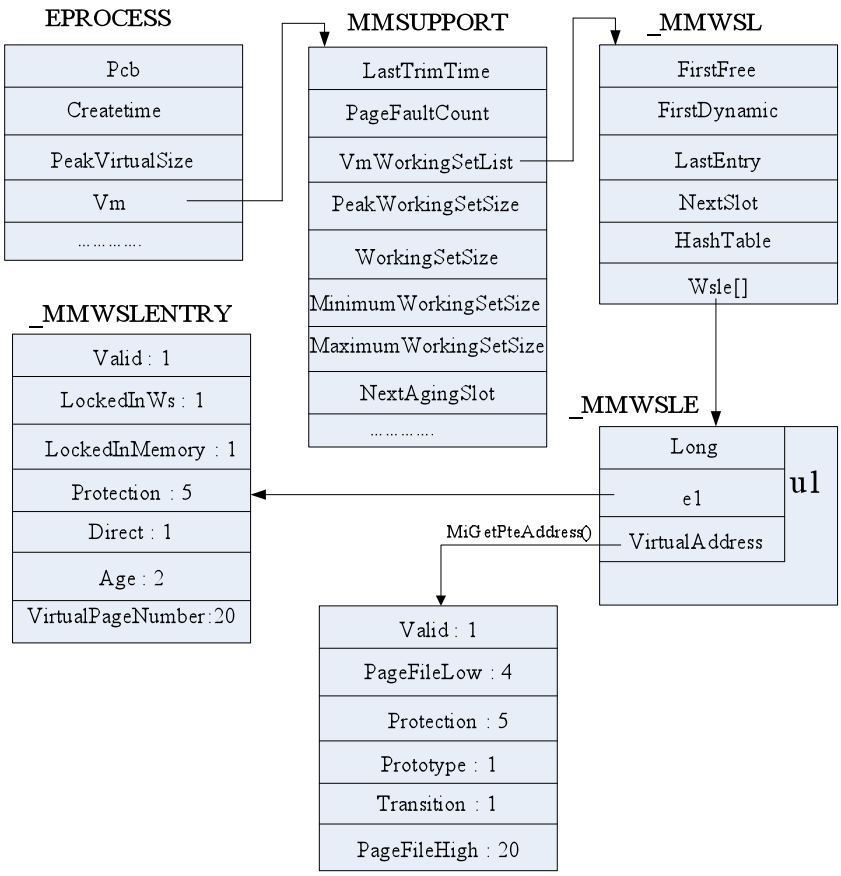

Figure 1. Working set data structures

Understanding the relationship between the working set structures is a great help in kernel debugging, algorithm modification, and kernel system call addition. This relationship may be described briefly as follows:

- The Age field of MMWSLENTRY serves as a basis for the trim operation.
- The Accessed bit of the hardware-supported page table entry (PTE) is used as a reference variable for aging. (PTE is not shown in Figure 1.)
- The HashTable field of MMWSL is a linear structure. It does not have a one-to-one correspondence with the array of working set list entries (represented by the Wsle pointer) and is just one of the subsets of the array. This can be seen, for example, in the code of the MiRemoveWsle() function.
- Three fields of MMSUPPORT are used in the aging algorithm:
  - NextAgingSlot: Index of the next aging page.
  - NextEstimationSlot: Index of the next estimation page.
  - EstimatedAvailable: The number of invalid pages of the working set of this process as estimated during the execution of the aging algorithm.

- The NextSlot field of MMSUPPORT is used in the page replacement algorithm.

2.1 MMPFN and Related Structures

This section describes the data structures of the page frame database involved when performing the removal operations MiFreeWsle() and MiFreeWsleList() on the working set page.

```
typedef struct _MMPFN {
    union {
        PFN_NUMBER Flink;          //Point to the predecessor page frame
in the backup/modified link list
        WSLE_NUMBER WsIndex;      //Indicate the working set index in the
working set page
        PKEVENT Event;             //The event address where the I/O
operation is being performed
        NTSTATUS ReadStatus;
        SINGLE_LIST_ENTRY NextStackPfn;
    } u1;
    PMMPTE PteAddress;             //Point to the corresponding page
table address
    union {
        PFN_NUMBER Blink;         //Point to the successor page frame
        ULONG_PTR ShareCount;
    } u2;
    union {
        struct {
            USHORT ReferenceCount;
            MMPFNENTRY e1;
        };
        struct {
            USHORT ReferenceCount;  //Access count
            USHORT ShortFlags;      //ID field, mainly indicating the
state of this page table, namely modified, prototype, reading, and writing
        } e2;
    } u3;
    union {
        MMPTE OriginalPte;
        LONG AweReferenceCount;
```

```
    };
    union {
        ULONG_PTR EntireFrame;
        struct {
            ULONG_PTR PteFrame: 25;  //Index of the page frame in the page
frame number database;
            ULONG_PTR InPageError : 1;
            ULONG_PTR VerifierAllocation : 1;
            ULONG_PTR AweAllocation : 1;
            ULONG_PTR Priority : MI_PFN_PRIORITY_BITS;
            ULONG_PTR MustBeCached : 1;
        };
    } u4;
} MMPFN, *PMMPFN;
```

Generally, operations using the structures shown above are performed as follows:

- Obtain the pointer to the respective page table entry (PTE) from the Key field of the indexed entry of MMWSLE_HASH or from the VirtualAddress field of MMWSLE using the macro definition MiGetPteAddress:

```
PointerPte =MiGetPteAddress
(WsInfo->VmWorkingSetList->Table[Index].Key)
PointerPte =MiGetPteAddress (WsInfo->VmWorkingSetList->Wsle
[index].u1.e)
```

- Obtain the pointer to the page frame number (PFN) descriptor from the PTE using the macro definition MI_PFN_ELEMENT:

```
Pfn1 = MI_PFN_ELEMENT (PointerPte->u.Hard.PageFrameNumber);
//Useful during deletion
```

- Obtain the location (index) of this page frame in the page frame database from PEN PFN using the macro definition MI_PFN_ELEMENT_TO_INDEX:

```
Pfn1->u4.PteFrame = (ULONG_PTR)MI_PFN_ELEMENT_TO_INDEX (Pfn1)
```

  Having the page frame location is useful when you add a page frame to (or insert a page frame into) the working set.

```
typedef struct _MMPFNLIST {
    PFN_NUMBER Total; //Number of page frames in this type of
                      // linked list
```

```
    MMLISTS ListName; //Type of linked list: idle linked list,
                      // modified linked list, backup linked list, etc.
    PFN_NUMBER Flink;         //Successor index
    PFN_NUMBER Blink;         //Predecessor index
} MMPFNLIST;
```

The pages in the page frame database are organized into six linked lists:

MmZeroedPageListHead	MmFreePageListHead
MmStandbyPageListHead	MmModifiedNoWritePageListHead
MmModifiedPageListHead	MmBadPageListHead.

Each is a pointer to the head node of its respective linked list. Given a PFN structure, you can obtain the next or the previous PFN structure from the Flink or Blink field (respectively) using the macro definition MI_PFN_ELEMENT.

3 Working Set Management Code Analysis

3.1 Kernel Code Related to the Working Set

The distribution of the kernel code related to the working set is shown in Table 1.

Table 1. Kernel code related to the working set

File name	Functions of module
ps.h	Contains data structures and interfaces related to the process, including part of structures of the working set
mi.h	Contains data structures and interfaces related to the memory management subsystem, including part of structures of the working set
wslist.c	Includes a series of functions for operating the working set structure
wstree.c	Implements some auxiliary functions in working set management, and implements such operations as add, delete and update on the working set or working set entries
wsmanage.c	Includes the functions for operating the working set of the process in the active state, and implements the working set management thread

3.2 Kernel Code Analysis for the Working Set

This section describes several representative functions that use the structures of the working set described previously, so that you may gain a more in-depth understanding of the working set and lay a foundation for reading and modifying the page replacement algorithm in the experiments described in this paper.

3.2.1 Page Aging and Trimming

Start to look for the cause for aging from the function KeBalanceSetManager, and determine the trimming and aging criteria. When there is a shortage of idle pages, more idle pages can be obtained by trimming the working set. KeBalanceSetManager, which is running on a separate thread and which is executed cyclically once per second, calls the working set management function MmWorkingSetManager. The working set management function first calls the MiComputeSystemTrimCriteria() function to determine the operation to be performed: trim the working set, age the working set, or do nothing. The triggering conditions for these three cases are:

- Trim the working set: This is based on the following three conditions:
  - The current number of available pages (Available) is smaller than the number of pages required.

- A record of the replaced page is already available in this working set: MiReplacing == TRUE.
- More than 1/4 of the available pages are recycled as the backup pages.

As soon as one of the above conditions is met, the trimming operation is performed immediately, and the corresponding value is assigned to the TrimCriteria variable.

- Age the working set: When none of the trimming conditions is true, and the number of currently available pages (Available) is smaller than the limit of 20,000, the aging operation is performed.
- No operation: When neither of the above conditions is true, it indicates that currently there is plenty of memory available in the system, so MiComputeSystemTrimCriteria() returns 0 and no operation is performed.

So far, the criteria for the working set processing has been determined and stored in WorkingSetRequestFlags and TrimCriteria variables. If the WorkingSetRequestFlags is not zero—meaning that the trimming or aging operation should be performed—then MiProcessWorkingSets(WorkingSetRequestFlags, &TrimCriteria) is called. If WorkingSetRequestFlags is zero, no operation is performed, and the MmModifiedPageListHead.Total counter of the modified page link list is checked to see whether it reaches or exceeds the MmModifiedPageMaximum limit. If it does, the modified page writer operation is activated.

The function call chain throughout the above process is shown below:

```
KeBalanceSetManager()
 →MmWorkingSetManager()
     → MiComputeSystemTrimCriteria()// MI_AGE_ALL_WORKING_SETS
     ←MiComputeSystemTrimCriteria()// MI_AGE_ALL_WORKING_SETS
     →MiProcessWorkingSets()
         → MiTrimWorkingSet()    //Determine whether to make trimming
according to MI_TRIM_ALL_WORKING_SETS
         ← MiTrimWorkingSet()
         → MiAgeWorkingSet()//Judge whether to perform aging according
to MI_AGE_ALL_WORKING_SETS
         ← MiAgeWorkingSet()
     ←MiProcessWorkingSets()
 ←MmWorkingSetManager()
KeBalanceSetManager()
```

3.2.1.1 MiTrimWorkingSets Algorithm Analysis

MiProcessWorkingSets calls MiTrimWorkingSet() to implement the working set trimming operation. Here is a detailed analysis of MiProcessWorkingSets**.**

Function prototype

```
MiTrimWorkingSet (WSLE_NUMBER Reduction, PMMSUPPORT WsInfo, ULONG
TrimAge)
```

Parameter description

IN WSLE_NUMBER Reduction: Number of pages to be trimmed.

IN PMMSUPPORT WsInfo: Pointer of the working set of the process.

IN ULONG TrimAge: Age limit of the pages to be trimmed.

Function description

Variable description:

WsleFlushList: Store the indexes and number of pages to be removed.

TryToFree, StartEntry, LastEntry: These three integer variables loop through the Wsle array to determine the pages to be trimmed.

WorkingSetList: working set link list pointer.

Wsle: Working set page pointer.

PointerPte: The PTE structure pointer of the current page.

NumberLeftToRemove: Number of pages not removed at the end of this process.

NumberNotFlushed: Number of pages not removed after MiFreeWsleList is called.

Description:

The starting location StartEntry and the ending location LastEntry of the search in Wsle are computed.

The TrimMore program segment searches and removes pages in a cyclic action from the working set page array Wsle, starting with TryToFree (initially set to StartEntry) using the least recently used (LRU) algorithm, and calls function MiFreeWsleList to release relevant pages. When NumberLeftToRemove (initially set to Reduction) reaches zero or when the search reaches its initial point (StartEntry), it stops.

If WsleFlushList.Count is not zero, MiFreeWsleList function is called to release it, returning the number of entries not flushed. This number is added to NumberLeftToRemove, and if the sum is not zero, further trimming should be performed. Further trimming, however, is impossible if the search has reached its initial point. If this is not the case, the algorithm returns to the TrimMore label to repeat the trimming (step 2 above).

If no further trimming is necessary or possible, the function MiTrimWorkingSet checks whether the working set can be contracted. If it can, the MiRemoveWorkingSetPages function is called. This function compresses the working set list entries into the front of the working set and frees the pages for unneeded working set entries.

The final return value of MiTrimWorkingSet is the number of removed pages (Reduction – NumberLeftToRemove).

3.2.1.2 MiAgeWorkingSet Algorithm Analysis

MiProcessWorkingSets calls MiAgingWorkingSet() to implement the page aging operation. This function is used to estimate the number of invalid pages in the working set of this process. By using the DoAging parameter, it also determines whether to perform page aging. Here is a detailed analysis of the MiAgeWorkingSet() algorithm.

Function prototype

```
MiAgeWorkingSet (
 IN PMMSUPPORT VmSupport,
 IN LOGICAL DoAging,
 IN PWSLE_NUMBER WslesScanned,
 IN OUT PPFN_NUMBER TotalClaim,
 IN OUT PPFN_NUMBER TotalEstimatedAvailable)
```

Parameter description

IN PMMSUPPORT VmSupport: Pointer of the working set.

IN LOGICAL DoAging: Indicates whether to perform the aging operation.

IN PWSLE_NUMBER WslesScanned: Pointer to the total number of working set list entries scanned in one sweep, used as a control to prevent excessive aging on large systems that have many processes.

IN OUT PPFN_NUMBER TotalClaim: Pointer to system-wide claim to update.

IN OUT PPFN_NUMBER TotalEstimatedAvailable: Pointer to system-wide estimate to update.

Function description

Variable description:

DoAging: When DoAging=False, skip steps 1 through 3 outlined below.

Description:

1. Determine the number of aged pages. A single iteration of the Aging operation does not age all the pages of a process but only 1/ 2^MiAgingShift(4) pages starting from NextAgingSlot. All the pages can be aged once after 2^MI_AGE_AGING_SHIFT seconds.

The number of aged pages may be corrected in special cases:

- The calculated number of aged pages, NumberToExamine, is not allowed to exceed 8192.
- If the number of already scanned entries reaches or exceeds 262144 (0x40000, or number of pages in 1gigabyte), NumberToExamine is set to 64.

2. Determine the starting location of aging:

   Check whether **NextAgingSlot** is a valid page.

If it is not a valid page, use MI_NEXT_VALID_AGING_SLOT to determine the next aged page to be used as the starting page.

3. Repeat the following NumberToExamine times:

   Get the pointer to the respective page table entry (PTE);

If the page has been accessed since the last check (macro MI_GET_ACCESSED_IN_PTE returns non-zero), reset the Accessed bit of the PTE and reset the age of the current working set list entry;

Otherwise, increase Age for the page using macro MI_INC_WSLE_AGE; this macro increases the age by 1 up to the maximal value of 3.

Estimate the number of unused pages in the working set. Similar to aging, estimation is performed only on 1/ 2^MiEstimationShift(5) pages. Unlike aging, where pages are selected in sequence, here one page is selected every 2^MiEstimationShift pages, which is not a fixed value. For the specific algorithm, refer to the MI_NEXT_VALID_ESTIMATION_SLOT macro code. Pages are not checked continuously because the number of invalid pages in the entire working set is estimated at the end of the operation based on the computed probability. Various pages in the page set are fetched to ensure that the entire page set will be more accurate.

Determine the sample size. There are several different cases for determination of SampleSize:

- SampleSize = VmSupport -> WorkingSetSize - WorkingSetList -> FirstDynamic
- NumberToExamine = SampleSize >> MiEstimationShift.

This is the number of preferred pages. It depends on how the following conditions are met:

If the control parameter WslesScanned is larger than (1024 * 1024 * 1024) / PAGE_SIZE, the bigger computer may contain a lot of processes. To avoid performing the estimation operation too many times, the number of the pages to be estimated and the MiEstimationShift value are calculated again later.

If NumberToExamine is greater than 8192 (peak value), then NumberToExamine equals 8192, and the bigger computer has sufficient resources. In this case, the number of searches is reduced and MiEstimationShift is calculated again.

If NumberToExamine is greater than or equal to MI_MINIMUM_SAMPLE, the number of searches should not be smaller than the minimum value of 64.

Start to scan from FirstDynamic. If Wsle[CurrentEntry] is invalid, MI_NEXT_VALID_ESTIMATION__SLOT is called to recalculate a new starting estimation location, which is stored in CurrentEntry.

Estimate the number of invalid pages, starting from CurrentEntry, and store the value in the SampledAgeCounts array.

Calculate the number of invalid pages in a small range by using MI_CALCULATE_USAGE_ESTIMATE(SampledAgeCounts, CounterShift), and estimate the total number of invalid pages in this working set.

Store the estimation result in this working set:

- VmSupport ->Claim = Claim
- VmSupport -> EstimatedAvailable = Estimate

Use the estimation result to modify the TotalClaim and TotalEstimatedAvailable parameters of this function to make it return to the calling program.

3.2.2 Analysis of Page Release Process

MiTrimWorkingSet() determines the index numbers of the pages to be removed in the working set, and the number of pages to be released, based on the incoming trimming criteria, and encapsulates them in the WsleFlushList structure sent to the MiFreeWsleList function, which releases the listed pages. The page release steps are analyzed in “MiRemoveWsle Algorithm Analysis” below.

Windows is a multitasking operating system. Though the index number of the page to be released has been determined, this page may be shared by other processes during the period from determining the index number to calling MiRemoveWsle to actually release the page. If the page is shared, the MiDecrementShareCount() function is called to decrease the shared counter by 1.

Note: If this page is a prototype page table entry, looping should be performed for the first time to set FlushIndex[] in all these cases to zero.

MiRemoveWsle() is then called in the subsequent loops to remove the page.

3.2.2.1 MiRemoveWsle() Algorithm Analysis

Function prototype

```
MiRemoveWsle (WSLE_NUMBER Entry,PMMWSL WorkingSetList)
```

Parameter description

IN WSLE_NUMBER Entry: Index of the working set entry to be released.

IN PMMWSL WorkingSetList: Pointer to linked list of working set entries of the process.

Function description

Store the indexed working set entry (Wsle[Entry]) in the temporary variable WsleContents.

Check whether the current working set of the process is a working set of the system, and which category of the system working set it belongs to, and decrease the counter by 1.

Invalidate the working set entry and its local copy:
WsleContents.u1.e1.Valid = 0;
Wsle[Entry].u1.e1.Valid = 0;

If the working set entry has been hashed, delete it from the hash table. The lookup process is as follows:

- Determine the starting index of the search: Hash = MI_WSLE_HASH (WsleContents.u1.Long, WorkingSetList).
- Search from Hash to the end of the table, until the key value of the hash table entry matches the virtual address obtained from WsleContents.
- If no match is found, restart the search from the beginning of the hash table to the original starting index. This ensures that the entire hash table is searched once.
- After the location is found, set Table[Hash].Key to zero.

4 Viewing Working Set State Using WinDbg

4.1 Introduction to WinDbg Commands

WinDbg is a free source code–level debugging tool developed by Microsoft. WinDbg can be used for debugging in both Kernel mode and user mode. Robust debugging commands are available in both modes. To call the help file and view explanation of all the commands, type **.hh**.

This section describes the functions of the WinDbg commands for working set management. Table 2 summarizes these functions. (For detailed usage and other commands, refer to the help file.)

Table 2. Related debugging commands for working set management

Command	Function explanation
!process	Shows information about the current process, including the virtual address of PCB and the process image name.

!vm	Shows the memory configuration and usage of the current system.
!pte	Shows the page table entry and page directory entry for the specified virtual address.
dd	Shows the contents of the memory in specified address range.
dc	Shows the content of the specified virtual address using symbols.
dt	Shows information about local variables, global variables, and data types. To see the kernel data type information, enter **dt nt!***data_name*, where *data_name* is the name of the variable. Wildcards (*) can be used in names. For example, command **dt nt!***A** shows all variables whose names begin with A.
?(expression)	Calculates the value of an expression. The value can be displayed as a decimal value.

4.2 Viewing Working Set State

This section details how to locate a certain page of the working set and view its contents using the debugging commands together with the related data structures of the working set that are described in Section 1.1.

To view the start of the working set, we will first run the application and request 100-page memory. Each page can be written with other data. Then, we will set a breakpoint in the kernel to stop the kernel.

Note: WinDbg debugs the kernel in the same manner as other applications. However, to stop the kernel, we should know what code has walked through the kernel. Section 4.3.1 details how to add a system call. We set a breakpoint within the added system call, and this system call is executed in WinDbg. WinDbg stops when it reaches the system call code. In this way, we can debug the current kernel state.

View the state of the current process using the !process command.

kd> !process

```
PROCESS 81de1bf8  SessionId: 0  Cid: 07f4    Peb: 7ffdf000  ParentCid:
05fc
  DirBase: 1af1a000  ObjectTable: e1718598  HandleCount:   9.
  Image: Explore_WS.
  VadRoot 815f1d20 Vads 121 Clone 0 Private 193. Modified 0. Locked
0.
  DeviceMap e183e008
  Token                             e19169f8
……
```

```
THREAD 819aeba0  Cid 07f4.07e8  Teb: 7ffde000 Win32Thread: 00000000
RUNNING on processor 0
```

As you can see, the process starts at the address 0x81de1bf8. This is the address where the process control block (PCB) of this process resides. The process image of the current process is Explore_WS, which is the name of our own application.

View details about the process structure using the dt command.

kd> dt nt!_EPROCESS 81de1bf8

```
+0x000 Pcb            : _KPROCESS
+0x078 ProcessLock     : _EX_PUSH_LOCK
+0x080 CreateTime      : _LARGE_INTEGER 0x1c7b881`3b7cd070
+0x088 ExitTime        : _LARGE_INTEGER 0x0
+0x0d0 ExceptionPort    : 0xe125fad0
+0x0d4 ObjectTable      : 0xe1718598 _HANDLE_TABLE
+0x0d8 Token           : _EX_FAST_REF
+0x198 DefaultHardErrorProcessing : 1
+0x19c LastThreadExitStatus : 0
+0x1a0 Peb             : 0x7ffdf000 _PEB
+0x1a4 PrefetchTrace     : _EX_FAST_REF
+0x1e4 SeAuditProcessCreationInfo : _SE_AUDIT_PROCESS_CREATION_INFO
+0x1e8 Vm              : _MMSUPPORT
+0x230 MmProcessLinks   : _LIST_ENTRY [ 0x808a0338 – 0x819ab690 ]
+0x238 ModifiedPageCount : 0
+0x23c JobStatus        : 0
+0x240 Flags           : 0x450801
+0x240 CreateReported    : 0y1
```

The hexadecimal data preceded by the PLUS SIGN represent the offsets of data fields within the EPROCESS structure. For example, the offset of the PCB relative to the EPROCESS address is 0, which indicates that it is the first data member of EPROCESS. Referring to the relationship of data structures that is presented in Figure 1, we can see that the offset of the working set–related field Vm relative to EPROCESS is +0x1e8.

Calculate the address of the Vm structure using the ? command.

kd> ?(81de1bf8+1e8)

```
Evaluate expression: -2116149792 = 81de1de0
```

By default, we enter the hexadecimal data in the ? command. In the result, the content to the left of the equal sign is the decimal data, and the content to the right of the equal sign is the hexadecimal data. The address of the Vm structure is 0x81de1de0.

Output the details about Vm using the dt command.

kd> dt nt!_MMSUPPORT 81de1de0

```
+0x000 WorkingSetExpansionLinks : _LIST_ENTRY [ 0x808a4bb0 –
0x819ab648 ]
+0x008 LastTrimTime     : _LARGE_INTEGER 0x1c7b881`3b7cd070
+0x010 Flags           : _MMSUPPORT_FLAGS
+0x014 PageFaultCount   : 0x189
+0x018 PeakWorkingSetSize : 0x18d
+0x01c GrowthSinceLastEstimate : 0x189
+0x020 MinimumWorkingSetSize : 0x32
+0x024 MaximumWorkingSetSize : 0x159
+0x028 VmWorkingSetList : 0xc0502000 _MMWSL
+0x02c Claim           : 0
+0x030 NextEstimationSlot : 0
+0x034 NextAgingSlot    : 0
+0x038 EstimatedAvailable : 0
+0x03c WorkingSetSize   : 0x18d
+0x040 WorkingSetMutex  : _EX_PUSH_LOCK
```

The output shows that the size of the current working set is 0x18d pages, the page fault count is 0x189 pages, the maximum value of the working set is 0x159 pages, and the minimum value is 0x32 pages.

The _MMWSL structure represents a working set list. The offset of its address relative to _MMSUPPORT is 0x028. The address of _MMWSL is calculated below:

kd> ?(81de1de0+0x028)

```
Evaluate expression: -2116149752 = 81de1e08
kd> dd 81de1e08 l 1  //Show that the content stored in 81de1e08 is
c0502000, which is the real address of _MMWSL
81de1e08  c0502000 //This is different from the Vm address calculated
above
```

Output the details about the working set link list structure _MMWSL using the dt command.

kd> dt nt!_MMWSL c0502000

```
+0x000 FirstFree        : 0x18d    //The location where the working set
page is added next time
+0x004 FirstDynamic     : 4      //Subscript of the first available
page among the working set pages
+0x008 LastEntry        : 0x23b   // Subscript of the last available
page among the working set pages
+0x00c NextSlot         : 4      //The starting page of the search when
the page replacement algorithm is used
+0x010 Wsle             : 0xc0502698 _MMWSLE
//The virtual address where the page table entry structure of the
working set stores pages
+0x014 LastInitializedWsle : 0x259
+0x018 NonDirectCount   : 0xb0
+0x01c HashTable        : (null)
+0x020 HashTableSize    : 0
+0x024 NumberOfCommittedPageTables : 6
+0x028 HashTableStart   : 0xc0703000
+0x02c HighestPermittedHashAddress : 0xc0c00000
+0x038 UsedPageTableEntries : [768] 0x6b
+0x638 CommittedPageTables : [24] 7
```

View the details about the page table entry of the working set (all the virtual addresses of the page).

kd> dd C0502698 l 0x18d

```
c0502698  c0300203 c0301203 c0501203 c0502203
c05026a8  c01ff201 7ffc2009 7ffa6009 7ffa5009
c05026b8  7ffa4009 7ffa3009 7ffd0009 7ffa1009
c05026c8  7c94d001 0012f201 c01f2201 7c9b7221
……
c0502c98  00a40201 00a50201 00a60201 00a70201
c0502ca8  00a80201 00a90201 00aa0201 00ab0201
c0502cb8  00ac0201 00ad0201 00ae0201 00af0201
c0502cc8  00b00201
```

The first column in italic indicates the virtual address. After that, each row contains four entries of 4-byte data. Each entry represents the virtual address of a page and page attributes. Since the page size is 4 kilobytes (0x1000), all virtual addresses are aligned on 0x1000: 7c94d000, 0012f000, etc. The lower 12 bits (or three rightmost hexadecimal digits) of each entry are the attribute bits of respective page.

The structure of the entry is as follows:

```
typedef struct _MMWSLENTRY {
  ULONG_PTR Valid : 1;
  ULONG_PTR LockedInWs : 1;
  ULONG_PTR LockedInMemory : 1;
  ULONG_PTR Protection : 5;
  ULONG_PTR Hashed : 1;
  ULONG_PTR Direct : 1;
  ULONG_PTR Age : 2;       // Age bits indicate how long since page
was accessed
  ULONG_PTR VirtualPageNumber : MM_VIRTUAL_PAGE_SIZE;
// The high 20 bits indicate the page number; the physical page address
is calculated from this
} MMWSLENTRY;
```

View information about the page directory and the page table entry of a specific page using the !pte command.

kd> !pte 00b00201

```
               VA 00b00201
PDE at   C0300008       PTE at C0002C00    //The virtual address
where the page table and the page directory are located
contains 17E1C867     contains 1895D867      //Content of the
virtual address (physical address)
pfn 17e1c ---DA–UWEV    pfn 1895d ---DA–UWEV
```

In this output, C0300008 is the virtual address of the page directory entry where the page table is located, 17E1C867 is the content of the page entry, and the high 20 bits indicate the location of the corresponding page table in the memory.

C0002C00 is the virtual address of the page table where the page is located, 1895D867 is the content of the page table entry, the first 20 bits indicate the physical page number, and the low 12 bits indicate a symbol.

Show content of a specific page using the dc command.

Check whether this content is the content that was written by our application Explore_WS by taking the last page as a sample. From step 6, you can see that the virtual address of the last page is 0x00b00201. On the command line, set the last 12 bits to zero to obtain 0x00b00000.

kd> dc 0x00b00000

```
00b00000  74697257 61702065 63206567 65746e6f  Write page conte
00b00010  703a2078 20656761 21303031 00000021  x :page 100!!...
00b00020  00000000 00000000 00000000 00000000  ................
```

From here, we can see that virtual address 0x00b00000 is really the last page we request in application, with the content "Write page conte x :page 100!!".

4.3 Viewing Working Sets of Multiple Processes with WinDbg

In this section, the WinDbg commands that are explained in Section 3.2 are used in an experiment that views the state of multiple processes, analyzes the locations of working sets of each process in the virtual address space, and verifies that the virtual address is valid for the current process only.

In this experiment, two processes need to be started. It is best that the two selected programs are similar, differing only in the content that is written to the page when requesting the page. Here, two programs Other-Alloc.exe and alloc.exe are executed. Write the content "VirtualAlloc—Alloc Page mum ….." in the page for the first process, and write the content "Other VirtualAlloc—Alloc Page mum….." in the page for the second process. Do not perform any other operation. The purpose of this experiment is to be able to see the difference between the content of the two pages. In the experiment, the breakpoint is set in Row 194 in the kernel file wslist.c, and then two processes are run at the same time. The records are shown in the Table 3.

Table 3. Comparing content elements of two processes

Program Content Elements	**alloc.exe**	**Other-alloc.exe**
View program structure address	**kd> !process→PROCESS 81dce220**	**kd> !process→PROCESS 81e41a10**
View working set structure address	**kd> ? (81dce220+1e8)** **= 81dce408**	**kd> ? (81e41a10+1e8)** **= 81e41bf8**
View content of working set link list	**kd> dt nt!_MMSUPPORT 81dce408** [The working set size 0xa5a is obtained.]	**kd> dt nt!_MMSUPPORT 81e41bf8** [The working set size 0x65a is obtained.]

View virtual address of working set page	**kd> dd c0502698 l 0xa5a**	**kd> dd c0502698 l 0x65a**

View contents in the pages with the same virtual address	**kd> dc 04b90000**
	[The content is displayed as follows:]
	`VirtualAlloc–Alloc Page`
	`mum 1148.at0x 4b90000..`
		kd> dc 04b90000
		[The content is displayed as follows:]
		`Other VirtualAlloc–Alloc Page`
		`mum 1148.at0x 4b90000..`

View the physical page addresses where PDEs and PTEs of pages with the same virtual address are located	**kd> !pte 04b90000**
	VA 04b90000
	`PDE at C0300048`
	`PTE at C0012E40`
	`contains 16E44867`
	`contains 161FE867`
	`pfn 16e44 ---DA–UWEV`
	`pfn 161fe ---DA–UWEV`
		kd> !pte 04b90000
		VA 04b90000
		`PDE at C0300048`
		`PTE at C0012E40`
		`contains 166B8867`
		`contains 16552867`
		`pfn 166b8 ---DA–UWEV`
		`pfn 16552 ---DA–UWEV`

cr3 register	**kd> r cr3** `cr3=17eca000`	**kd> r cr3** `cr3=17b82000`

This experiment indicates that two virtual addresses are valid for the current process only, the page directories and the page table entries of all the processes reside in the space with the address range 0xC0000000-0xC03FFFFF4M, the page directories are mapped within the 0xC0300000-0xC0303FFF range (4K), and the cr3 register is used between processes to find a physical memory page that is suitable for the processes.

5 Project Setting for Working Set Management Experiment

Three experiment projects are described in this section: an analysis experiment on the page replacement algorithm of working set, an algorithm modification experiment, and an experiment on writing an application verification algorithm. These three experiments are related to each other and depend heavily on each other. Consider performing the experiments together, as presented in this section, based on the requirements of the three experiments. For example, determining the peak value for the working set (experiment 1) is the foundation for the subsequent experiments, and modification of the page replacement algorithm (experiment 2) may require multiple tests and improvements to the test application.

5.1 Part 1: Analyze the Page Replacement Algorithm

5.1.1 Background Knowledge

The page replacement algorithm is one of the key operating system algorithms. It involves process management and memory management. If additional pages are requested after the

number of pages assigned to a process exceeds the maximal allowed value for the working set, the system replaces old pages using a replacement algorithm. This experiment can be conducted by using WinDbg together with the Windows kernel source code.

5.1.2 Experiment Requirements

This experiment has three requirements:

- Find the peak value of the working set page of the process.
- Analyze the kernel code path (function-calling relationship) from page fault to execution of the replacement algorithm using the call stack information obtained from WinDbg during debugging, and analyze the basic components of each function.
- Conduct a detailed analysis of the page replacement algorithm procedure, because this is the foundation for later modifying the page replacement algorithm (experiment 2).

5.1.3 Submission

To meet the experiment requirements, the experiment you must submit documents in PDF format with the following contents:

- Describe how you found the peak value of the process' working set, and how you determined it.
- Show the function calling relationship for page faults and page replacements.

5.1.4 Experiment Steps and Guide

Perform the following steps to conduct this experiment:

Conduct static analysis of the code, find the execution conditions for the page replacement algorithm, and appropriately modify these conditions to determine the peak value of the working set. (See the source code in Table 1 to find the page replacement algorithm.)

- Set a breakpoint in the page replacement algorithm using WinDbg. Let the kernel attempt to execute the page replacement algorithm. Here, you may need to write a user application that would require the kernel's page replacement mechanism to be engaged.
- Verify correctness of your understanding of the algorithm, and record each step in detail; this step will be helpful in the subsequent experiments.
- When the kernel executes the page replacement algorithm, view the call path from the page default processing function to the page replacement algorithm in the call stack window of the WinDbg.
- Analyze the functionality of each called function, analyze the page replacement algorithm in detail, and write a detailed document.

5.2 Part 2: Modify the Page Replacement Algorithm

5.2.1 Background Knowledge

In this experiment, you modify the page replacement algorithm of the kernel. Before the experiment, you should gain a thorough understanding of this algorithm, such as the structure inside the working set and the storage location of pages.

5.2.2 Experiment Requirements

This experiment has four requirements:

- Modify the page replacement algorithm to use an algorithm of your choice and recompile the kernel.
- Start the system using the new kernel image.
- Set a breakpoint in the new algorithm and verify that the kernel correctly executes the modified page replacement algorithm.

5.2.3 Submission

To meet the experiment requirements, you must submit the following materials:

- A document in PDF format with the following contents:
    - A description of how your new algorithm works.
    - A code listing of your implementation, with detailed comments.
- The source file containing your kernel modifications and your WRKX86.exe.

5.2.4 Experiment Guide

Perform the following steps to conduct this experiment:

- Modify the original replacement algorithm:
- Conduct a thorough analysis of breakpoint setting.
- Start the system.
- Write your own algorithm; replace the page replacement algorithm in the system with the new algorithm.
- Recompile the kernel and start the system using the new kernel.
- Set a breakpoint in the new algorithm, verify correctness of the algorithm by performing single-step debugging, and record the verification result.