Virtual Memory

- Background
- Demand Paging
- Process Creation
- Page Replacement
- Allocation of Frames
- Thrashing
- Demand Segmentation
- Operating System Examples
Background

- Virtual memory – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

[Diagram showing virtual memory and a memory map with a mapping to physical memory]
Virtual-address Space

Shared Library Using Virtual Memory
**Demand Paging**

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

**Transfer of a Paged Memory to Contiguous Disk Space**
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- Example of a page table snapshot:

<table>
<thead>
<tr>
<th>Frame #</th>
<th>valid-invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault

Page Table When Some Pages Are Not in Main Memory
Page Fault

- If there is ever a reference to a page, first reference will trap to OS ⇒ page fault
- OS looks at another table to decide:
  - Invalid reference ⇒ abort.
  - Just not in memory.
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = 1.
- Restart instruction: Least Recently Used
  - block move
  - auto increment/decrement location

Steps in Handling a Page Fault
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)
  \[
  EAT = (1 - p) \times \text{memory access} \\
  + p (\text{page fault overhead}) \\
  + [\text{swap page out}] \\
  + \text{swap page in} \\
  + \text{restart overhead}
  \]
Steps in Handling Page faults

- Trap to the OS
- Save the user registers and process state
- Determine that the interrupt was a page fault
- Check that the page reference was legal and determine the location of the page on the disk
- Issue a read from the disk to a free frame:
  - Wait in a queue for this device until the read request is serviced
  - Wait for the device to seek and/or latency time
  - Begin the transfer of the page to the selected free frame
- While waiting, allocate the CPU to some other user
- Receive an interrupt from the disk subsystem on I/O completion
- Save the registers and process state for the process running
- Determine that the interrupt was from the disk
- Correct the page table and other tables to show that the desired page is in the memory now
- Wait for the CPU to be allocated to this process again
- Restore the user registers, process state, and new page table, and resume the interrupted instruction

Demand Paging Example

- Memory access time = 200 ns
- Disk access time = 8 ms = 8,000,000 ns

- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out

- EAT = (1 – p) x 200 + p (8,000,000)
  = 200 + 7,999,800p

- What should p be for EAT to be <220 ns?
  220 > 200 + 7999800p
  or p < 0.000025
Process Creation

- Virtual memory allows other benefits during process creation:
  - Copy-on-Write
  - Memory-Mapped Files (later)

Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory.
  - If either process modifies a shared page, only then is the page copied.
- COW allows more efficient process creation as only modified pages are copied.
- Free pages are allocated from a pool of zeroed-out pages.
Before process 1 modifies page C

After Process 1 Modifies page C
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.

- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk.

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.

Need For Page Replacement
Basic Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process
Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Graph of Page Faults Versus The Number of Frames
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

1 1 4 5
2 2 1 3
3 3 2
9 page faults

4 frames
1 1 5 4
2 2 1 5
3 3 2
4 4 3
10 page faults

FIFO Replacement – Belady’s Anomaly
- more frames ⇒ more page faults

FIFO Page Replacement

<table>
<thead>
<tr>
<th>reference string</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 3 0 4 2 3 0 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td>7 7 7 2 2 4 4 4 0 0 0 7 7 7 1 1 1 0 0</td>
</tr>
<tr>
<td>0 0 0 3 3 3 2 2 2 1 1 1</td>
</tr>
<tr>
<td>1 1 3 2 2 2</td>
</tr>
</tbody>
</table>

page frames
**FIFO Illustrating Belady’s Anomaly**

![Graph showing number of page faults versus number of frames in FIFO algorithm.]

**Optimal Algorithm**

- Replace page that will not be used for longest period of time
- 4 frames example:
  
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- 6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs
Optimal Page Replacement

Reference string:

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| 7 | 7 | 7 | 2 | 2 | 2 | 2 | 2 | 2 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |

Counter implementation

Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.

When a page needs to be changed, look at the counters to determine which are to change.

Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Counter implementation

- Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
- When a page needs to be changed, look at the counters to determine which are to change.
LRU Page Replacement

<table>
<thead>
<tr>
<th>Reference String</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 2 3 0 1 2 0 1 7 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 7 7 2 2 4 4 4 0 1 1 1 3 0 0 3 2 2 2 2 2 2 2 7</td>
</tr>
</tbody>
</table>

LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement
Use Of A Stack to Record The Most Recent Page References

Reference string

\[
\begin{array}{cccccccc}
4 & 7 & 0 & 7 & 1 & 0 & 1 & 2 & 7 & 1 & 2 \\
\hline
2 & & & & & & 7 & & & \\
1 & & & & & & 2 & & & \\
0 & & & & & & 1 & & & \\
7 & & & & & & 0 & & & \\
4 & & & & & & 4 & & & \\
\end{array}
\]

Stack before

Stack after

a

b

LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists). We do not know the order, however.

- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules
### Second-Chance (clock) Page-Replacement Algorithm

- Reference bits
- Pages

The algorithm involves a circular queue of pages and a counter for each page to track the number of references. The next victim is determined by the counter value.

### Counting Algorithms

- Keep a counter of the number of references that have been made to each page.
- **LFU Algorithm**: replaces page with smallest count.
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages
- Proportional allocation – Allocate according to the size of process
  - \( S = \sum s_i \)
  - \( m = \text{total number of frames} \)
  - \( a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m \)

\[
\begin{align*}
  m &= 64 \\
  s_1 &= 10 \\
  s_2 &= 127 \\
  a_1 &= \frac{10}{137} \times 64 \approx 5 \\
  a_2 &= \frac{127}{137} \times 64 \approx 59
\end{align*}
\]
Priority Allocation

- Use a proportional allocation scheme using priorities rather than size.

- If process \( P_i \) generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another

- **Local replacement** – each process selects from only its own set of allocated frames
Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system

- **Thrashing** = a process is busy swapping pages in and out

---

Thrashing

- Why does paging work?
  - Locality model
    - Process migrates from one locality to another
    - Localities may overlap
- Why does thrashing occur?
  - $\Sigma$ size of locality $>$ total memory size
Locality In A Memory-Reference Pattern

\[ \Delta \equiv \text{working-set window} = \text{a fixed number of page references} \]

Example: 10,000 instruction

\[ WSS_i (\text{working set of Process } P_i) = \]

total number of pages referenced in the most recent \( \Delta \) (varies in time)

- if \( \Delta \) too small will not encompass entire locality
- if \( \Delta \) too large will encompass several localities
- if \( \Delta = \infty \Rightarrow \) will encompass entire program

\[ D = \sum WSS_i = \text{total demand frames} \]

- if \( D > m \) \( \Rightarrow \) Thrashing
- Policy if \( D > m \), then suspend one of the processes

Working-Set Model
**Working-set model**

- Working set: All the physical pages “owned” by a process
  - Essentially, all the pages the process can reference without incurring a page fault
- Working set limit: The maximum pages the process can own
  - When limit is reached, a page must be released for every page that’s brought in (“working set replacement”)
  - Default upper limit on size for each process
  - System-wide maximum calculated & stored in MmMaximumWorkingSetSize
    - approximately RAM minus 512 pages (2 MB on x86) minus min size of system working set (1.5 MB on x86)
    - Interesting to view (gives you an idea how much memory you’ve “lost” to the OS)
  - True upper limit: 2 GB minus 64 MB for 32-bit Windows

---

Operating System Concepts 9.47

---

Operating System Concepts 9.48
A process always starts with an empty working set
- It then incurs page faults when referencing a page that isn't in its working set
- Many page faults may be resolved from memory (to be described later)

Pages are brought into memory as a result of page faults
- Prior to XP, no pre-fetching at image startup
- But readahead is performed after a fault
  - See MmCodeClusterSize, MmDataClusterSize, MmReadClusterSize
- If the page is not in memory, the appropriate block in the associated file is read in
  - Physical page is allocated
  - Block is read into the physical page
  - Page table entry is filled in
  - Exception is dismissed
  - Processor re-executes the instruction that caused the page fault (and this time, it succeeds)
- The page has now been “faulted into” the process “working set”
**Prefetch Mechanism**

- First 10 seconds of file activity is traced and used to prefetch data the next time
  - Also done at boot time (described in Startup/Shutdown section)
- Prefetch “trace file” stored in \Windows\Prefetch
  - Name of .EXE-<hash of full path>.pf
- When application run again, system automatically
  - Reads in directories referenced
  - Reads in code and file data
    - Reads are asynchronous
    - But waits for all prefetch to complete

In addition, every 3 days, system automatically defrags files involved in each application startup

Bottom line: Reduces disk head seeks

This was seen to be the major factor in slow application/system startup
9.53 Working Set Replacement

- When working set max reached (or working set trim occurs), must give up pages to make room for new pages
- Local page replacement policy (most Unix systems implement global replacement)
- Means that a single process cannot take over all of physical memory unless other processes aren't using it
- Page replacement algorithm is least recently accessed (pages are aged)
- On UP systems only in Windows 2000 – done on all systems in Windows XP/Server 2003
- New VirtualAlloc flag in XP/Server 2003: MEM_WRITE_WATCH

9.54 Soft vs. Hard Page Faults

- Types of 'soft' page faults:
  - Pages can be faulted back into a process from the standby and modified page lists
  - A shared page that's valid for one process can be faulted into other processes

- Some hard page faults unavoidable
  - Process startup (loading of EXE and DLLs)
  - Normal file I/O done via paging
    - Cached files are faulted into system working set

- To determine paging vs. normal file I/Os:
  - Monitor Memory->Page Reads/sec
    - Not Memory->Page Faults/sec, as that includes soft page faults
  - Subtract System->File Read Operations/sec from Page Reads/sec
  - Or, use Filemon to determine what file(s) are having paging I/O (asterisk next to I/O function)
  - Should not stay high for sustained period

Operating System Concepts 9.53

Operating System Concepts 9.54
**Working Set System Services**

- Min/Max set on a per-process basis
  - Can view with `!process` in Kernel Debugger
- System call below can adjust min/max, but has minimal effect prior to Server 2003
  - Limits are "soft" (many processes larger than max)
  - Memory Manager decides when to grow/shrink working sets
- New system call in Server 2003 (`SetProcessWorkingSetSizeEx`) allows setting hard min/max
  - Can also self-initiate working set trimming
    - Pass -1, -1 as min/max working set size (minimizing a window does this for you)

**Windows API:**

```c
SetProcessWorkingSetSize(
    HANDLE hProcess,
    DWORD dwMinimumWorkingSetSize,
    DWORD dwMaximumWorkingSetSize)
```

**Keeping Track of the Working Set**

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units
Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

Page Fault Rates over Time

Operating System Concepts 9.57

Operating System Concepts 9.58
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory.

- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.

- Simplifies file access by treating file I/O through memory rather than read() write() system calls.

- Also allows several processes to map the same file allowing the pages in memory to be shared.
Shared Memory in the WIN32 API

Allocating Kernel Memory
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class MemoryMapReadOnly
{
    // Assume the page size is 4 KB
    public static final int PAGE_SIZE = 4096;
    public static void main(String args[]) throws IOException
    {
        RandomAccessFile inFile = new RandomAccessFile(args[0],"r");
        FileChannel in = inFile.getChannel();
        MappedByteBuffer mappedBuffer =
        in.map(FileChannel.MapMode.READ_ONLY, 0, in.size());
        long numPages = in.size() / (long)PAGE_SIZE;
        if (in.size() % PAGE_SIZE > 0)
            ++numPages;
        int position = 0;
        for (long i = 0; i < numPages; i++)
        {
            byte item = mappedBuffer.get(position);
            position += PAGE_SIZE;
        }
        in.close();
        inFile.close();
    }
}

The API for the map() method is as follows:
map(mode, position, size)
Other Issues

- **Prepaging**
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume $s$ pages are prepaged and $\alpha$ of the pages is used
    - Is cost of $s \times \alpha$ save pages faults $>$ or $<$ than the cost of prepaging $s \times (1 - \alpha)$ unnecessary pages?
    - $\alpha$ near zero $\Rightarrow$ prepaging loses

- **Page size selection must take into consideration:**
  - fragmentation
  - table size
  - I/O overhead
  - locality

Other Issues (Cont.)

- **TLB Reach** - The amount of memory accessible from the TLB

- TLB Reach $= (\text{TLB Size}) \times (\text{Page Size})$

- Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.
Other Issues (Cont.)

- **Increase the Page Size.** This may lead to an increase in fragmentation as not all applications require a large page size.

- **Provide Multiple Page Sizes.** This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

Other Issues (Cont.)

- **Program structure**
  - int A[][] = new int[1024][1024];
  - Each row is stored in one page
  - Program 1
    - for (j = 0; j < A.length; j++)
      - for (i = 0; i < A.length; i++)
        - A[i,j] = 0;
    - 1024 x 1024 page faults
  - Program 2
    - for (i = 0; i < A.length; i++)
      - for (j = 0; j < A.length; j++)
        - A[i,j] = 0;
    - 1024 page faults
Other Considerations (Cont.)

- **I/O Interlock** – Pages must sometimes be locked into memory

- Consider I/O. Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

---

**Reason Why Frames Used For I/O Must Be In Memory**

Diagram showing a buffer connected to a disk drive.
Demand Segmentation

- Used when insufficient hardware to implement demand paging.
- OS/2 allocates memory in segments, which it keeps track of through segment descriptors
- Segment descriptor contains a valid bit to indicate whether the segment is currently in memory.
  - If segment is in main memory, access continues,
  - If not in memory, segment fault.

Operating System Examples

- Windows NT
- Solaris 2
Windows XP

- Uses demand paging with clustering. Clustering brings in pages surrounding the faulting page.
- Processes are assigned working set minimum and working set maximum
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, automatic working set trimming is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Working Set

- Working set: All the physical pages “owned” by a process
  - Essentially, all the pages the process can reference without incurring a page fault
- Working set limit: The maximum pages the process can own
  - When limit is reached, a page must be released for every page that’s brought in (“working set replacement”)
  - Default upper limit on size for each process
  - System-wide maximum calculated & stored in MmMaximumWorkingSetSize
    - approximately RAM minus 512 pages (2 MB on x86) minus min size of system working set (1.5 MB on x86)
    - Interesting to view (gives you an idea how much memory you’ve “lost” to the OS)
  - True upper limit: 2 GB minus 64 MB for 32-bit Windows
A process always starts with an empty working set
- It then incurs page faults when referencing a page that isn’t in its working set
- Many page faults may be resolved from memory (to be described later)

If the page is not in memory, the appropriate block in the associated file is read in
- Physical page is allocated
- Block is read into the physical page
- Page table entry is filled in
- Exception is dismissed
- Processor re-executes the instruction that caused the page fault (and this time, it succeeds)

The page has now been “faulted into” the process “working set”
Prefetch Mechanism

- First 10 seconds of file activity is traced and used to prefetch data the next time
  - Also done at boot time (described in Startup/Shutdown section)
- Prefetch “trace file” stored in \Windows\Prefetch
  - Name of .EXE-<hash of full path>.pf
- When application run again, system automatically
  - Reads in directories referenced
  - Reads in code and file data
    - Reads are asynchronous
    - But waits for all prefetch to complete

In addition, every 3 days, system automatically defrags files involved in each application startup

Bottom line: Reduces disk head seeks
- This was seen to be the major factor in slow application/system startup
Working Set Replacement

- When working set max reached (or working set trim occurs), must give up pages to make room for new pages
- Local page replacement policy (most Unix systems implement global replacement)
  - Means that a single process cannot take over all of physical memory unless other processes aren’t using it
- Page replacement algorithm is least recently accessed (pages are aged)
- On UP systems only in Windows 2000 – done on all systems in Windows XP/Server 2003
- New VirtualAlloc flag in XP/Server 2003: MEM_WRITE_WATCH

Soft vs. Hard Page Faults

- Types of "soft" page faults:
  - Pages can be faulted back into a process from the standby and modified page lists
  - A shared page that’s valid for one process can be faulted into other processes
- Some hard page faults unavoidable
  - Process startup (loading of EXE and DLLs)
  - Normal file I/O done via paging
    - Cached files are faulted into system working set
- To determine paging vs. normal file I/Os:
  - Monitor Memory->Page Reads/sec
    - Not Memory->Page Faults/sec, as that includes soft page faults
  - Subtract System->File Read Operations/sec from Page Reads/sec
  - Or, use Filemon to determine what file(s) are having paging I/O (asterisk next to I/O function)
  - Should not stay high for sustained period
Working Set System Services

- Min/Max set on a per-process basis
  - Can view with `!process` in Kernel Debugger
- System call below can adjust min/max, but has minimal effect prior to Server 2003
  - Limits are “soft” (many processes larger than max)
  - Memory Manager decides when to grow/shrink working sets
- New system call in Server 2003 (SetProcessWorkingSetSizeEx) allows setting hard min/max
- Can also self-initiate working set trimming
  - Pass -1, -1 as min/max working set size (minimizing a window does this for you)

Windows API:
SetProcessWorkingSetSize(
    HANDLE hProcess,
    DWORD dwMinimumWorkingSetSize,
    DWORD dwMaximumWorkingSetSize)

Locking Pages

- Pages may be locked into the process working set
  - Pages are guaranteed in physical memory (“resident”) when any thread in process is executing

Windows API:
status = VirtualLock(baseAddress, size);
status = VirtualUnlock(baseAddress, size);

- Number of lockable pages is a fraction of the maximum working set size
  - Changed by SetProcessWorkingSetSize
- Pages can be locked into physical memory (by kernel mode code only)
  - Pages are then immune from “outswapping” as well as paging
    MmProbeAndLockPages
Balance Set Manager

- Nearest thing Windows has to a “swapper”
  - Balance set = sum of all in swapped working sets
- Balance Set Manager is a system thread
  - Wakes up every second. If paging activity high or memory needed:
    - trims working sets of processes
    - if thread in a long user-mode wait, marks kernel stack pages as pageable
    - if process has no nonpageable kernel stacks, “outswaps” process
    - triggers a separate thread to do the “outswap” by gradually reducing target process’s working set limit to zero
- Evidence: Look for threads in “Transition” state in PerfMon
  - Means that kernel stack has been paged out, and thread is waiting for memory to be allocated so it can be paged back in
- This thread also performs a scheduling-related function
  - CPU starvation avoidance - already covered

System Working Set

- Just as processes have working sets, Windows’ pageable system-space code and data lives in the “system working set”
- Made up of 4 components:
  - Paged pool
  - Pageable code and data in the exec
  - Pageable code and data in kernel-mode drivers, Win32K.Sys, graphics drivers, etc.
  - Global file system data cache
- To get physical (resident) size of these with PerfMon, look at:
  - Memory \ Pool Paged Resident Bytes
  - Memory \ System Code Resident Bytes
  - Memory \ System Driver Resident Bytes
  - Memory \ System Cache Resident Bytes
  - Memory \ Cache bytes counter is total of these four “resident” (physical) counters (not just the cache; in NT4, same as “File Cache” on Task Manager / Performance tab)
Session Working Set

- New memory management object to support Terminal Services in Windows 2000/XP/Server 2003
- Session = an interactive user
- Session working set = the memory used by a session
  - Instance of WinLogon and Win32 subsystem process
  - WIN32K.SYS remapped for each unique session
    - Win32 subsystem objects
    - Win32 subsystem paged pool
  - Process working sets page within session working set
- Revised system space layout

Managing Physical Memory

- System keeps unassigned physical pages on one of several lists
  - Free page list
  - Modified page list
  - Standby page list
  - Zero page list
  - Bad page list - pages that failed memory test at system startup
- Lists are implemented by entries in the "PFN database"
  - Maintained as FIFO lists or queues
Paging Dynamics

- New pages are allocated to working sets from the top of the free or zero page list
- Pages released from the working set due to working set replacement go to the bottom of:
  - The modified page list (if they were modified while in the working set)
  - The standby page list (if not modified)
    - Decision made based on "D" (dirty = modified) bit in page table entry
  - Association between the process and the physical page is still maintained while the page is on either of these lists

Standby and Modified Page Lists

- Modified pages go to modified (dirty) list
  - Avoids writing pages back to disk too soon
- Unmodified pages go to standby (clean) list
- They form a system-wide cache of "pages likely to be needed again"
  - Pages can be faulted back into a process from the standby and modified page list
  - These are counted as page faults, but not page reads
**Modified Page Writer**

- When modified list reaches certain size, modified page writer system thread is awoken to write pages out
  - See MmModifiedPageMaximum
  - Also triggered when memory is overcommitted (too few free pages)
  - Does not flush entire modified page list
- Two system threads
  - One for mapped files, one for the paging file
- Pages move from the modified list to the standby list
  - E.g. they can still be soft faulted into a working set

**Free and Zero Page Lists**

- Free Page List
  - Used for page reads
  - Private modified pages go here on process exit
  - Pages contain junk in them (e.g. not zeroed)
  - On most busy systems, this is empty
- Zero Page List
  - Used to satisfy demand zero page faults
    - References to private pages that have not been created yet
  - When free page list has 8 or more pages, a priority zero thread is awoken to zero them
  - On most busy systems, this is empty too
As processes incur page faults, pages are removed from the free, modified, or standby lists and made part of the process working set.

A shared page may be resident in several processes' working sets at one time (this case not illustrated here).
Page Frame Number-Database

- One entry (24 bytes) for each physical page
- Describes state of each page in physical memory

Entries for active/valid and transition pages contain:
- Original PTE value (to restore when paged out)
- Original PTE virtual address and container PFN
- Working set index hint (for the first process...)

Entries for other pages are linked in:
- Free, standby, modified, zeroed, bad lists (parity error will kill kernel)

Share count (active/valid pages):
- Number of PTEs which refer to that page; 1->0: candidate for free list

Reference count:
- Locking for I/O: INC when share count 1->0, DEC when unlocked
- Share count = 0 & reference count = 1 is possible
- Reference count 1->0: page is inserted in free, standby or modified lists

Page Frame Database – states of pages in physical memory

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active/valid</td>
<td>Page is part of working set (sys/proc), valid PTE points to it</td>
</tr>
<tr>
<td>Transition</td>
<td>Page not owned by a working set, not on any paging list I/O is in progress on this page</td>
</tr>
<tr>
<td>Standby</td>
<td>Page belonged to a working set but was removed; not modified</td>
</tr>
<tr>
<td>Modified</td>
<td>Removed from working set, modified, not yet written to disk</td>
</tr>
<tr>
<td>Modified no write</td>
<td>Modified page, will not be touched by modified page write, used by NTFS for pages containing log entries (explicit flushing)</td>
</tr>
<tr>
<td>Free</td>
<td>Page is free but has dirty data in it – cannot be given to user process – C2 security requirement</td>
</tr>
<tr>
<td>Zeroed</td>
<td>Page is free and has been initialized by zero page thread</td>
</tr>
<tr>
<td>Bad</td>
<td>Page has generated parity or other hardware errors</td>
</tr>
</tbody>
</table>
Page tables and page frame database

Process 1 page table
- valid
- Invalid: disk address
- Invalid: transition

Process 2 page table
- valid
- Invalid: disk address
- Valid

Process 3 page table
- valid
- Invalid: transition
- Invalid: disk address
- Prototype PTE
- Active
- Modified
- Bad
- Modified no write
- Standby
- Free
- Standby
- Zeroed

Why “Memory Optimizers” are Fraudware

Before:

<table>
<thead>
<tr>
<th>Notepad</th>
<th>Word</th>
<th>Explorer</th>
<th>System</th>
<th>Available</th>
</tr>
</thead>
</table>

During:

Available
RAM Optimizer

Notepad Word Explorer System

After:

Available

See Mark’s article on this topic at http://www.winnetmag.com/Windows/Article/ArticleID/41095/41095.html
Page Files

- What gets sent to the paging file?
  - Not code – only modified data (code can be re-read from image file anytime)
- When do pages getpaged out?
  - Only when necessary
  - Page file space is only reserved at the time pages are written out
  - Once a page is written to the paging file, the space is occupied until the memory is deleted (e.g. at process exit), even if the page is read back from disk
- Windows XP (& Embedded NT4) can run with no paging file
  - NT4/Win2K zero pagefile size actually creates a 20MB temporary page file (temp pf.sys)
  - WinPE never has a pagefile
- Page file maximums:
  - 16 page files per system
  - 32-bit x86: 4095MB
  - 32-bit PAE mode, 64-bit systems: 16 TB

Why Page File Usage on Systems with Ample Free Memory?

- Because memory manager doesn’t let process working sets grow arbitrarily
  - Processes are not allowed to expand to fill available memory (previously described)
    - Bias is to keep free pages for new or expanding processes
  - This will cause page file usage early in the system life even with ample memory free
- We talked about the standby list, but there is another list of modified pages recently removed from working sets
  - Modified private pages are held in memory in case the process asks for it back
  - When the list of modified pages reaches a certain threshold, the memory manager writes them to the paging file (or mapped file)
  - Pages are moved to the standby list, since they are still “valid” and could be requested again
Sizing the Page File

- Given understanding of page file usage, how big should the total paging file space be?
  - (Windows supports multiple paging files)
- Size should depend on total private virtual memory used by applications and drivers
  - Therefore, not related to RAM size (except for taking a full memory dump)
- Worst case: system has to page all private data out to make room for code pages
  - To handle, minimum size should be the maximum of VM usage ("Commit Charge Peak")
    - Hard disk space is cheap, so why not double this
  - Normally, make maximum size same as minimum
  - But, max size could be much larger if there will be infrequent demands for large amounts of page file space
    - Performance problem: page file extension will likely be very fragmented
    - Extension is deleted on reboot, thus returning to a contiguous page file

Contiguous Page Files

- A few fragments won’t hurt, but hundreds of fragments will
- Will be contiguous when created if contiguous space available at that time
- Can defrag with Pagedefrag tool (freeware - www.sysinternals.com)
  - Or buy a paid defrag product…
When Page Files are Full

- When page file space runs low
  - 1. "System running low on virtual memory"
    - First time: Before pagefile expansion
    - Second time: When committed bytes reaching commit limit
  - 2. "System out of virtual memory"
- Page files are full
  - Look for who is consuming pagefile space:
    - Process memory leak: Check Task Manager, Processes tab, VM Size column
      - or Perfmon "private bytes", same counter
    - Paged pool leak: Check paged pool size
      - Run poolmon to see what object(s) are filling pool
      - Could be a result of processes not closing handles - check process "handle count" in Task Manager

System Nonpaged Memory

- Nonpagedable components:
  - Nonpagedable parts of NtosKrn.exe, drivers
  - Nonpaged pool (see PerfMon, Memory object: Pool nonpaged bytes)
  - non-paged code
  - non-paged data
  - pageable code+data (virtual size)
  - output of "drivers.exe" is similar
    - Win32K.Sys is paged, even though it shows up as nonpaged
      - Other drivers might do this too, so total nonpaged size is not fully visible

---

52
Optimizing Applications
Minimizing Page Faults

- Some page faults are unavoidable
  - Code is brought into physical memory (from .EXEs and .DLLs) via page faults
  - The file system cache reads data from cached files in response to page faults
- First concern is to minimize number of “hard” page faults
  - I.e. page faults to disk
  - See Performance Monitor, “Memory” object, “page faults” vs. “page reads” (this is system-wide, not per process)
  - For an individual app, see Page Fault Monitor:

  ```
c:\> pfmon /?
c:\> pfmon program-to-be-monitored
  ``

Accounting for Physical Memory Usage

- Process working sets
  - Perfmon: Process / Working set
  - Note, shared resident pages are counted in the process working set of every process that’s faulted them in
  - Hence, the total of all of these may be greater than physical memory
- Resident system code (NTOSKRNL + drivers, including win32k.sys & graphics drivers)
  - See total displayed by !drivers 1 command in kernel debugger
- Non pageable pool
  - Perfmon: Memory / Pool nonpaged bytes
- Free, zero, and identity page lists
  - Perfmon: Memory / Available bytes
- Pageable, but currently-resident, system-space memory
  - Perfmon: Memory / Pool paged resident bytes
  - Perfmon: Memory / System cache resident bytes
  - Memory / Cache bytes counter is really total of these four “resident” (physical) counters
- Modified, Bad page lists
  - Can only see size of these with !memusage command in kernel debugger
**Solaris**

- Maintains a list of free pages to assign faulting processes
- `Lotsfree` – threshold parameter (amount of free memory) to begin paging
- `Desfree` – threshold parameter to increasing paging
- `Minfree` – threshold parameter to being swapping
- Paging is performed by `pageout` process
- Pageout scans pages using modified clock algorithm
- `Scanrate` is the rate at which pages are scanned. This ranges from `slowscan` to `fastscan`
- Pageout is called more frequently depending upon the amount of free memory available

---

**Solaris 2 Page Scanner**

![Solaris 2 Page Scanner Diagram](image)