Methods for Identifying Memory Leaks in AIX Systems

By Barry J. Saad and Harold R. Lee

Application memory leaks can be difficult to detect and on modern multi-tasking systems where many applications are running, it can be even more difficult to identify the offending process(es).

This paper will explore the reasons for memory leaks, and using the AIX system performance tools, provide a methodology for identifying them to the process level. This paper will consist of four major parts: (1) definition and causes for a memory leak on a system, (2) examining a process heap using svmon and ps, (3) a method for detecting memory leaks on a system, (4) a method for identifying individual processes which have memory leaks.

The Causes for Memory Leaks

The Jargon File version 4.2.0 defines a memory leak as:

memory leak

n. An error in a program's dynamic-store allocation logic that causes it to fail to reclaim discarded memory, leading to eventual collapse due to memory exhaustion. These problems were severe on older machines with small, fixed-size address spaces, and special "leak detection" tools were commonly written to root them out. With the advent of virtual memory, it is unfortunately easier to be sloppy about wasting a bit of memory (although when you run out of memory on a VM machine, it means you've got a _real_ leak!).

Memory is allocated to a processes heap by using the malloc() C library call or in C++ when a class is instantiated by calling a "constructor" which will "set up"
the class by allocating the required memory for the class and initializing the members. When the memory is no longer required by the process, it is released for reuse by the free() C library system call or in C++ when the class is “torn down” by calling a “destructor” which will release the memory for reuse. When a process terminates, all of the program’s dynamic-store which is also called the processes heap and stack are returned to operating system for reallocation to other processes.

Memory leaks occur when memory that is allocated to the heap is not released by the process and the heap will continue to grow. Over time, the process will either allocate all of the memory and fill its’ address space which may cause abnormal termination of the process or it will use up the virtual memory and cause the kernel to take protective measures to preserve the integrity of the operating system by killing processes to relieve the virtual memory demands. Unfortunately, memory leaks are usually caused by long running processes and the algorithm which the kernel uses to reduce virtual memory demands is to start killing processes based on the “youngest” process (most recently created) and working toward the “oldest”. This can have the effect of many processes being “killed” before the offending process is dealt with.

Due to the complexity of programs and the levels of memory allocation and deallocation abstraction in object oriented programs, memory leaks are sometimes very difficult if not impossible to completely eliminate. One approach to this problem is to limit the lifespan of a program where the program is terminated before any potential memory leaks are allowed to grow enough to become a problem. The Apache web server is an example of this approach. When the Apache server is started, a parent process started which allocates very little memory and does very little work. The parent’s job is to initially spawn some number of child server process set by the StartServers directive in the configuration file, and then to try to maintain the number of child server process between two levels either by creating or killing additional processes. This
window is set by the MaxSpareServers and MinSpareServers directives in the startup configuration file.

Each child server processes will keep track of the number of requests which it will handle and after this number has been reached, the process will die and return all of the virtual memory to the kernel heap for reallocation. The following snippet is from the configuration file instruction page. Notice the reason given for limiting the lifetime of the child:

<table>
<thead>
<tr>
<th>MaxRequestsPerChild directive</th>
</tr>
</thead>
<tbody>
<tr>
<td>The MaxRequestsPerChild directive sets the limit on the number of requests that an individual child server process will handle. After MaxRequestsPerChild requests, the child process will die. If MaxRequestsPerChild is 0, then the process will never expire.</td>
</tr>
<tr>
<td>Setting MaxRequestsPerChild to a non-zero limit has two beneficial effects:</td>
</tr>
<tr>
<td>• it limits the amount of memory that process can consume by (accidental) memory leakage;</td>
</tr>
<tr>
<td>• by giving processes a finite lifetime, it helps reduce the number of processes when the server load reduces.</td>
</tr>
</tbody>
</table>

Not all applications have the flexibility of Apache to deal with potential leaks. Another strategy is for companies to schedule regular intervals for the applications to be recycled. This involves shutting down the applications and restarting them. While some reboot the entire system, this is not necessary unless there are mitigating circumstances such as a maintenance patch or a reboot specific tuning change (e.g. changing the asynchronous subsystem parameters).
Another form of process memory leakage is called “heap fragmentation”. Heap fragmentation, which is the bane of operating system design, is something developers have had to contend with since the beginning. A google search on “heap fragmentation” will return over 64,000 hits illustrating the magnitude of this issue. Heap fragmentation is when available memory is broken into small, non-contiguous blocks. When this occurs, memory allocation can fail even though there is enough memory in the heap to satisfy the request, because no one block of memory is large enough to satisfy the allocation request.

For applications that have a low memory usage, the standard heap is adequate; allocations will not fail due to heap fragmentation. However, if the application allocates memory frequently and uses a variety of allocation sizes, memory allocation can fail due to heap fragmentation.

Heap fragmentation is caused by numerous subsequent malloc()s and free()s on the process heap with widely varying sizes. This cause “holes” to form in the contiguous heap and unless a malloc() is small enough to use the space within a “hole” the process heap must grow to satisfy the request.

One strategy for dealing with heap fragmentation is to employ a technique called “Garbage Collection”. When a request cannot be satisfied from the heap due to fragmentation, the garbage collection routine is called which moves all of the “holes” to the end of the heap, making all of the space available at the end of the heap. This garbage collection is analogous to defragmenting a hard drive, which moves all of the unused space to the end.

Some of the early operating systems employed garbage collection techniques however this causes very erratic performance from the system since all activity had to stop while the memory was re arranged. The Java virtual machine (JVM), which is a “guest” operating system, currently uses this method with similar results. Garbage collection is not used on most operating systems today due in part to the erratic performance it produces.
Another strategy which some programs use to eliminate this is to manage the heap internally through the use of pointers. When the program starts there is a single memory allocation call to obtain the heap and then the program itself will allocate portions out of it to satisfy the internal requests. Most data base programs like Oracle and DB2 use this method of internal memory management.

Operating system developers continue to work on more efficient memory allocation algorithms and incorporate them into the kernels to reduce this issue to one on minimum impact. The Windows™ developers have introduced a heap manager called the “Low Fragmentation Heap” (LFH) which places the holes in a Cartesian btree structure similar to the AIX 4.2 – 5.2 malloc routine known as the “Yorktown malloc” since it was developed by the Yorktown research facility.

A new malloc called the “watson malloc” is available on AIX 5.3. The Watson malloc which is cache based and uses a simplified rbtree (red-black tree) structure increases malloc performance as well as reducing heap fragmentation. The new malloc() is optimized for large multi-threaded applications and time will tell how much more efficient it is. For more information on this malloc refer to the following link:


Refer to the section - Understanding the Watson Allocation Policy

From a system administration perspective, heap fragmentation is dealt with the same manner as a memory leak. The applications must be quiesced and restarted.
Examining A Process Heap Using svmon and ps

For an overview of the virtual memory management at a process level, refer to the following articles and info pages:

http://w3-03.ibm.com/support/americas/pseries/performance.html

Article – Understanding Virtual Memory Management

Also refer to the man or info page on the svmon and ps commands

The processes virtual memory footprint consists of three separate items. The actual (1) heap and the (2) process stack which is addressed by (esid) register “2”, and the (3) shared library data which is addressed by register “f”. The loader segment “d” is shared by all processes and is ignored.

```
# svmon -nrP 262188

-------------------------------------------------------
<table>
<thead>
<tr>
<th>Pid</th>
<th>Command</th>
<th>Inuse</th>
<th>Pin</th>
<th>Pgsp</th>
<th>Virtual</th>
<th>64-bit</th>
<th>Mthrd</th>
<th>LPage</th>
</tr>
</thead>
<tbody>
<tr>
<td>262188</td>
<td>mamain</td>
<td>65980</td>
<td>4784</td>
<td>0</td>
<td>65977</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vsid</th>
<th>Esid</th>
<th>Type</th>
<th>Description</th>
<th>LPage</th>
<th>Inuse</th>
<th>Pin</th>
<th>Pgsp</th>
<th>Virtual</th>
</tr>
</thead>
<tbody>
<tr>
<td>23e9</td>
<td>2</td>
<td>work</td>
<td>process private</td>
<td>-</td>
<td>53263</td>
<td>3</td>
<td>0</td>
<td>53263</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Addr Range:</td>
<td></td>
<td></td>
<td>0..53247 : 65314..65535</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1f09d</td>
<td></td>
<td>work</td>
<td>loader segment</td>
<td>-</td>
<td>3508</td>
<td>0</td>
<td>0</td>
<td>3508</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Addr Range:</td>
<td></td>
<td></td>
<td>0..8684</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63ad</td>
<td>f</td>
<td>work</td>
<td>shared library data</td>
<td>-</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Addr Range:</td>
<td></td>
<td></td>
<td>0..582</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63ed</td>
<td></td>
<td>clnt code,</td>
<td>/dev/hd1:29598</td>
<td>-</td>
<td>2</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Addr Range:</td>
<td></td>
<td></td>
<td>0..7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c3a7</td>
<td></td>
<td>clnt</td>
<td>/dev/hd2:4183</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Addr Range:</td>
<td></td>
<td></td>
<td>0..7# ps gv 262188</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The virtual memory footprint is the sum of registers 2 and f which produces the following:

53263 + 12 = 53275 4K pages

The same process using the ps command with the “gv” flags shows the following:

```
# ps gv 262188

PID  TTY STAT TIME PGIN SIZE RSS LIM TSIZ TRS %CPU %MEM COMMAND
262188 pts/0 A 0:00 1 213100 213108 xx 1 8 0.0 40.0 ./mamain
```

The info page states that the “SIZE” column is:
SIZE (v flag) The virtual size of the data section of the process (in 1KB units).

The SIZE is reported as 213100 1K units. In order to convert to 4K pages we will multiply the SIZE by 1024 and divide by 4096

213100*1024 = 21214400 bytes or 212,144 Megabytes

21214400/4096 = 53275 4K pages

Notice that the virtual SIZE column with the output of the ps v command exactly correlates to the two private segments ( 2 and f) reported by the svmon command. The correct method of determining the virtual size of a process is to use the SIZE column produced with the “v” flag.

NOTE: The SIZE is not the same as the SZ column which is produced by the –l (lowercase ell) flag. The SZ column is the core image of the process and may be different if part of the process is swapped out. It also includes the program binary image.

Method For Detecting Memory Leaks On A System

The total virtual memory requirements of a system are reported using the “vmstat” command. The “avm” column ( Avtive Virtual Memory) displays the total virtual requirements of the system at that point in time. By using the vmstat command to monitor the virtual memory usage, a memory leak would be detected by a continual increase in the “avm” number over time. The best way to view the usage is to graph it since trends will be much easier to spot. A system that does not have a memory leak will show a utilization that will vary but will have no continuous slope to the graph. The following graph is of a system that does not exhibit any memory leak issues.
The following three samples were taken from a system that did exhibit a memory leak. Notice the “slope” shows a continual increase in virtual memory requirements.
Method For Identifying Individual Processes Which Have Memory Leaks

After a suspected memory leak has been established using the previous method, the next step is to identify the offending process.

The methodology is to capture the output of a “ps gv” command and after some period of time has elapsed, capture a second set of data. The SIZE columns from the two sets of data are compared to see which program(s) heaps have grown.

Appendix A contains an awk script which will process the two files and report the differences. Appendix B contains the source to a memory leak simulator which was compiled and run to simulate a memory leaking program.

The following ps listing shows the output of the ps gv command of the program:

```
$ ps gv | grep memhog
 307398 pts/0 A 0:00 0 5236 5244 xx 2 8 0.0 2.0 memhog 1
(... some time has elapsed...)
$ ps gv | grep memhog
 307398 pts/0 A 0:00 0 61560 56448 xx 2 8 0.0 14.0 memhog 1
```

The following output from the script shows the same information:

```
# ./post_vg.sh two

<table>
<thead>
<tr>
<th>pid</th>
<th>Before Size</th>
<th>After Size</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>122962</td>
<td>1964</td>
<td>1964</td>
<td>0</td>
</tr>
<tr>
<td>204980</td>
<td>332</td>
<td>332</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>135400</td>
<td>48</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>225454</td>
<td>204</td>
<td>204</td>
<td>0</td>
</tr>
<tr>
<td>237698</td>
<td>508</td>
<td>508</td>
<td>0</td>
</tr>
<tr>
<td>307398</td>
<td>5236</td>
<td>61560</td>
<td>56324</td>
</tr>
<tr>
<td>295058</td>
<td>1888</td>
<td>1888</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110836</td>
<td>644</td>
<td>644</td>
<td>0</td>
</tr>
</tbody>
</table>

*** Total Delta 56324
```
APPENDIX A - Awk script used to process output of ps vg command taken at two different time periods.

#!/bin/ksh
#
# Correlate ps.before and ps.after data ..
# command output from ps vg
# ONE_FILE-temp_ps_vg

print_help() {
    print "Usage: post_vg.sh [single_file|before_ps after_ps]"
    print " Post process ps vg output "
    print " where, "
    print " single_file contains a before and after snapshot"
    print " No files specified - assume"
    print " ==> ps_vg.before "
    print " ==> ps_vg.after "
    exit -1
}

main() {
    if [[ $1 == "-?" ]]
    then
        print_help
        exit -1
    fi
    if [[ $# == 2 ]]
    then
        cat $1 $2 > $ONE_FILE
    elif [[ $# == 1 ]]
    then
        cat $1 > $ONE_FILE
    else
        cat ps_vg.before ps_vg.after > $ONE_FILE
    fi
    post_vg
    rm $ONE_FILE
}

post_vg() {
    cat $ONE_FILE | awk 'BEGIN {
            list_label = "None"
        }
        /PID/ { if( list_label == "None" )

            if( list_label == "None" )

            list_label = $1
        }'
list_label = "Before"
else
    list_label = "After"
next

{ pid_list[$1]
   pid_size[$1, list_label ] = $6
}
END {

printf("pid\tBefore Size\tAfter Size\t Delta\n")
printf("--------\t----------\t----------\t--------\n")

for( pid in pid_list ) {
    if( (pid,"Before") in pid_size && (pid,"After") in pid_size ) {
        delta = pid_size[pid, "After"] - pid_size[pid, "Before"]
        d_total += delta
        printf("%s\t%11d\t%10d\t%10d\n", pid, \
               pid_size[pid, "Before"], \
               pid_size[pid, "After"], \
               delta )
    }
}

printf("*** Total Delta %d\n", d_total)
}

main $@
Appendix B - Memory leak simulation program source code

```c
#include <stdio.h>
#include <malloc.h>

#define MAXPTR 1024
#define MEG 1024*1024

int main(int argc, char *argv[]) {
    char   *buf[MAXPTR];
    char   *buf1;
    char   c;
    unsigned long   memmax;
    unsigned long   step;
    unsigned long   i,j,k;

    if(argc != 3) {
        printf("USAGE: %s  <Max Memory(meg)>  <stepi(meg)>

", argv[0]);
        exit(-1);
    }

    for(i=0;i!=MAXPTR;i++)
        buf[i]=NULL;
    i=j=k=0;
    memmax=(atoi(argv[1])*MEG);
    step  =(atoi(argv[2])*MEG);

    printf("Running Memhog with maxmem = %u  Step = %u

", memmax, step);

    sleep(5);
    for(i=step;i < memmax; i+=step,k++) {
        if((buf[k]=(char*)malloc(step)) == NULL) {
            perror("ERROR in malloc");
            exit(-1);
        }
        buf1=buf[k];
        printf("Mallocing and Writing %u bytes
", i);
        for(j=0;j<step;j+=30) {
            strcpy(buf1+j, "The Camp Town Ladies......");
        }
        printf("\nMalloced and Wrote %u bytes
", i);
        c=getchar();
    }

    for(i=0;buf[i]!=NULL; i++){
        free((char*)buf[i]);
    }

    exit(0);
}
```