USENIX Association

Proceedings of the 5th Annual Linux Showcase & Conference

Oakland, California, USA
November 5–10, 2001

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

A Study in Malloc: A Case of Excessive Minor Faults

Phillip Ezolt

Compaq Computer Corporation

Abstract

GNU libc's default setting for malloc can cause a significant performance penalty for applications that use it extensively, such as Compaq's high performance extended math library, CXML. The default malloc tuning can cause a significant number of minor page faults, and result in application performance of only half of the true potential. This paper describes how to remove the performance penalty using environmental variables and the method used to discover the cause of the malloc performance penalty.

1. Why?

When a performance problem is discovered the first question asked is usually "How can it be fixed?". Although the solution to the performance problem is valuable, the method used to diagnose and fix the problem is also valuable. An explanation can teach the inexperienced engineer the thought process of the experience engineer, and give the inexperienced engineer a method for finding and fixing future performance problems.

This paper describes how the performance problem of GNU libc's malloc was diagnosed and how a solution was discovered. The performance hunt is documented to demonstrate the methods used to find and fix a performance problem.

2. What?

A customer running a chemistry benchmark on an Alpha system reported a radically different application run time between Tru64 UNIX and Linux/Alpha on the same hardware. Since the hardware of the two test systems was identical, the runtime difference had to be caused by software. Fortunately, the customer used the Compaq Fortran compiler, the Compaq Portable Math Library (CPML), and the Compaq Extended Math Library (CXML) on Tru64 UNIX and Linux/Alpha. This meant that the compiler subsystem was also the same. The main difference was the operating system.

The program was run on both systems, and the "time" command showed the runtime of both. The customer reported that the user time was roughly the same on both operating systems, but system time on Linux/Alpha was much greater than on Tru64 UNIX.

	User	System	Elapsed	CPU
Linux	256.284u	209.641s	7:46.35	99.9%
Tru64 UNIX	257.027u	3.176s	4:29.85	96.4%

This difference pointed to a possible performance problem in the Linux operating system. To determine where in Linux the time was spent, DCPI[1] (an alpha profiling system) and was used to extract the following data[2]:

cycles[3]	dtbmiss	Image
8116120	1062	System Total
6979695	0	/vmlinux
6044706	0	cpu_idle
386675	0	do_anonymous_page
87264	0	__free_page
79269	0	__get_free_pages
60127	0	__copy_user
45412	0	EntMM
36035	0	do_page_fault
1110642	1052	Xvcc
226983	183	Dgemm_nt
213498	86	Dgemm_nn
187057	47	Icopy_
92613	153	Dgemm_tt

This profile showed that a large amount of the non idle kernel cycles was spent in the 'do_anonymous_page' kernel function. It also showed that a large number of dtbmisses occurred in xvcc, the customer's chemistry code.

The function of 'do_anonymous_page' was not immediately clear, but further investigation revealed that it was part of the Linux kernel's memory management routines (in /usr/src/linux/mm/memory.c), and that all calls to it ultimately began with the page fault handler

'handle_pte_fault'.   Therefore,   if
'do_anonymous_page' was called a large number of
times, the page fault handler was also being called a
large number of times.

In addition to DCPI, the "time" command was also
used to measure where time was spent. As a side effect,
it revealed that a large amount of minor page faults
occurred.

```
..
(168major+23099385minor)pagefaults
..
```

It was unclear at this point what a minor page fault
was, and whether a high number of them could cause a
performance problem. However, if Linux displayed a
high number of minor faults, and Tru64 UNIX did not,
it could have been an indication of the problem.

It was known that a minor fault was a type of
pagefault, and when a pagefault occurred a dtbmiss[4] or
itbmiss must also have occurred. The high number of
page faults that the "time" command reported
corresponded nicely with DCPI's report of a high
number of dtbmisses.

To determine if the number of minor faults was
different on the two Alpha operating systems, the
customer ran the following script on both Linux/Alpha
and Tru64 UNIX:

```
(findfault.sh)
#!/bin/sh

COMMAND=$1
#Print command with headers.

ps -a -o vsize,rss,minflt,majflt,cmd
| grep -e $COMMAND -e CMD | grep -v
grep | grep -v $0

while (true)
do
    sleep 1
    #Print command without headers.
    ps -a -o vsize, rss, minflt, \
    majflt, cmd | grep -e $COMMAND |\
    grep -v grep |grep -v $0
done
```
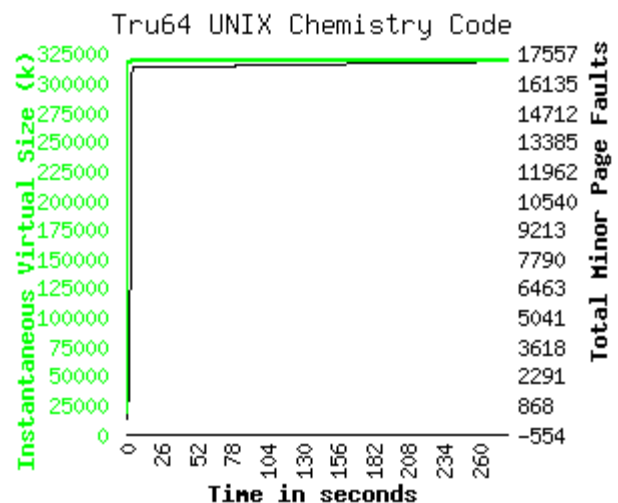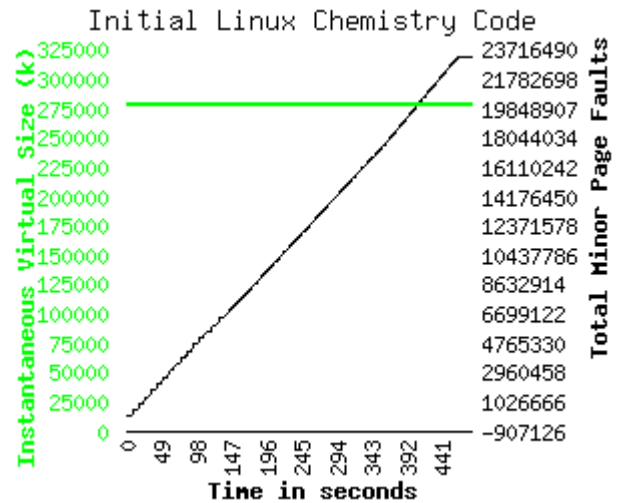
This script showed the size of the virtual and resident
set as well as the number of major and minor page
faults for a specified process.

The customer reported a significant number of minor
faults on Linux/Alpha, but nearly none on Tru64

UNIX.
The result of the test is reported in graphical form
below.

(Notice the difference in the scale of the minor faults)

Linux/Alpha had an ever-increasing number of minor
faults, while Tru64 UNIX's fault count stayed nearly
constant. Linux/Alpha's virtual set size fluctuated,
while Tru64 UNIX's stayed nearly constant.

3. Faults are at fault

The high minor fault count on Linux/Alpha pointed to
a significant difference between Tru64 UNIX and
Linux/Alpha. This was the first piece of the puzzle.
However, to understand what a high minor fault count
meant, it was necessary to understand what a minor
fault was.

A google[5] search of "minor fault" and "linux", revealed

the following information about the different types of page faults.

In Linux and Unix, page faults are either minor or major. A major fault requires an I/O operation to complete such as a page swap from disk. Minor faults can be handled without an I/O such as a Copy on Write (COW) request or a request for a zeroed page.

A linux kernel website[6]gave the following definitions:

Major fault

A major page fault occurs when an attempt to access a page not currently present in physical memory was made. The page must be swapped in to physical memory by the fault fix-up code.

Minor fault

A minor page fault occurs when an attempt to access a page present in physical memory, but without the correct permissions. An example is the first write to a second reference to a shared page, when the kernel must perform the copy-on-write and allow the task to update the copied page.

On Compaq's OpenVMS[7], a high number of minor faults usually indicated that a process's working set was larger than its allowed working set. Every attempt to use a new page would result in an old one being kicked out of its working set, and the program would spend a significant amount of time faulting in new pages.

It was assumed that this was what was happening on Linux.

The resident set of the customer's program hovered around 131 megabytes of memory, which seemed suspiciously close to a 128 megabytes limit. The Linux kernel code was searched for such a hard coded limit, but unfortunately, it was a dead end.

By running the following program, it was determined that a program could allocate 256 megabytes of memory, and touch every page without taking a minor fault:

```
#include <unistd.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
```

```
{
 int num_byte;
 char *buffer, *p;

 num_byte = atoi(argv[1])*1024*1024;
 buffer = malloc(num_byte);

 while (1){
    for (p = buffer;
         p < (buffer+num_byte);
         p += getpagesize())
         {*p= 0;}
    }
}
```

This lack of faults did not match the behavior of the customer's program.

4. Mmap Maker, Make Me a Map

The author of this paper would have been puzzled had he not remembered that a member of the Compaq Math library team reported a similar problem months ago. The problem of the Math Library team member and that of the customer appeared to be very similar. A message to the Math library team member revealed that he had found more information about the problem, but had not found a solution.

His message stated that:

"The problem involving minor page faults in DGEMM on Linux/Alpha is caused by the way Linux does heap management (i.e., malloc and free). Allocation of large buffers is done via mmap, and when they are freed, they are unmapped via munmap. The buffer allocated by DGEMM falls into this category. Thus, for each call to DGEMM, address space for the buffer is created, buffer pages are faulted into the resident set and then the buffer, and the address space, is deleted. "

This changed the focus of the search, and also allowed for the creation of a smaller test program which showed similar behavior to the original chemistry code: an ever increasing number of minor page faults on Linux, and a small number of page faults on Tru64 UNIX.

For those not fortunate enough to have a colleague who experienced a similar problem, the kernel's minor page fault handler could have been instrumented to print the address of instructions that cause more than 1000 minor page faults. Using this to find the guilty instruction, one could then use 'nm' and 'gdb' to

determine which function or line of code caused the minor faults. Although this would not be a general-purpose solution, the availability and modifiability of Linux kernel source makes this instrumentation possible.

5. Minor Fault in Allocation

Since it appeared that memory allocation was the cause of the problem, it could be tested independently of the customer's chemistry program. This was fortunate because the customer's chemistry program had many modules and a long compile time.

The following simple program could reproduce the high number of minor faults; it allocates a piece of memory, and then immediately frees it.

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

int main(int argc, char *argv[])
{
  int number_of_meg, num_byte;
  char *buffer;

  num_byte = atoi(argv[1])*1024*1024;

  while (1){
    buffer = malloc(num_byte);
    free(buffer);
    }
}
```

5.1 Linux/Alpha

An strace[8] of the test on Linux/Alpha confirms what the math library engineer had said. 'mmap' is used when mallocing large amounts of memory. Linux/Alpha had an ever-increasing amount of minor faults.

```
strace ./malloc_test 1
....
mmap(0, 1056768, PROT_NONE,
     0 /* MAP_??? */, 0, 0) =
0x20000456000
munmap(0x20000456000, 1056768) = 0
mmap(0, 1056768, PROT_NONE,
     0 /* MAP_??? */, 0, 0) =
0x20000456000
munmap(0x20000456000, 1056768) = 0
mmap(0, 1056768, PROT_NONE,
     0 /* MAP_??? */, 0, 0) =
0x20000456000
munmap(0x20000456000, 1056768) = 0
mmap(0, 1056768, PROT_NONE,
```

```
     0 /* MAP_??? */, 0, 0) =
0x20000456000
munmap(0x20000456000, 1056768) = 0
....
```


Linux/Alpha 1024k malloc
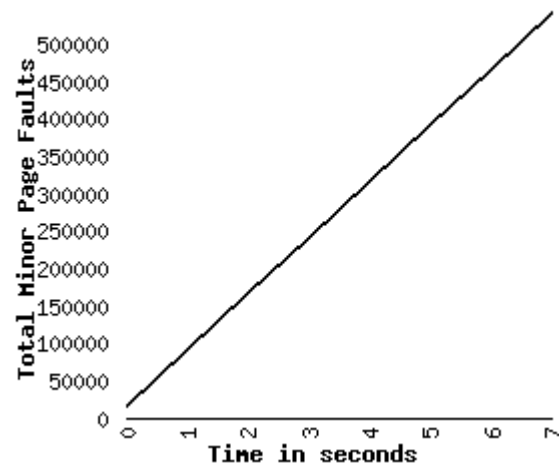
5.2 Tru64 UNIX

Tracing the same program on Tru64 UNIX yielded two interesting facts:

> Few minor page faults occurred on Tru64 UNIX, and

Tru64 UNIX used obreak() (a system call which increases the processes heap size) instead of mmap() (a system call which allocates system wide resources to a program) to malloc memory.

```
...
obreak (0x140108000) = 0
obreak (0x14020a000) = 0
obreak (0x140108000) = 0
obreak (0x14020a000) = 0
....
```


Tru64 UNIX 1024k malloc
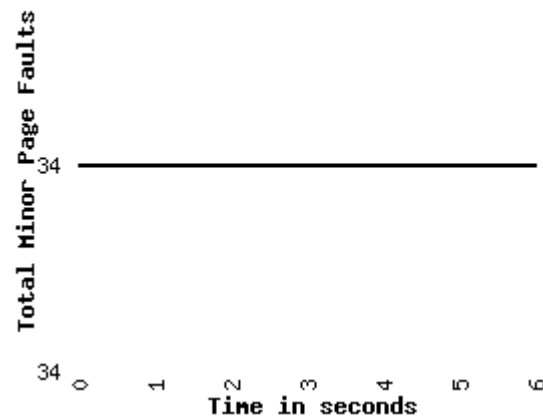
5.3 Intel/Linux

An Intel/Linux system behaved the same as an Linux/Alpha system, with a fluctuating mmap() value and an increasing number of faults.

```
strace ./malloc_test_i386 1
....
old_mmap(NULL, 1052672,
        PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS,
        -1, 0) = 0x4024f000
munmap(0x4024f000, 1052672) = 0
old_mmap(NULL, 1052672,
        PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS,
        -1, 0) = 0x4024f000
munmap(0x4024f000, 1052672) = 0
old_mmap(NULL, 1052672,
        PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS,
        -1, 0) = 0x4024f000
munmap(0x4024f000, 1052672) = 0
.....
```


Linux/x86 1024k malloc
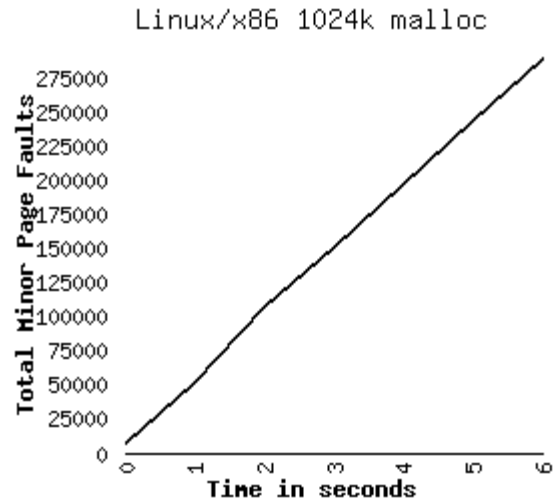
5.4 FreeBSD

To test this malloc issue on another operating system, FreeBSD was installed on VMware[9], a very fast i386 virtual machine.

FreeBSD did not display the increasing number of page faults. FreeBSD used break() to set memory limits, much like Tru64 UNIX.

```
 ...
2814 pagefault CALL break(0x4000)
2814 pagefault RET break 0
2814 pagefault CALL break(0x104000)
2814 pagefault RET break 0
2814 pagefault CALL break(0x14000)
2814 pagefault RET break 0
2814 pagefault CALL break(0x114000)
2814 pagefault RET break 0
```

```
2814 pagefault CALL break(0x14000)
2814 pagefault RET break 0
2814 pagefault CALL break(0x114000)
2814 pagefault RET break 0
2814 pagefault CALL break(0x14000)
2814 pagefault RET break 0
....
```


FreeBSD 1024k malloc
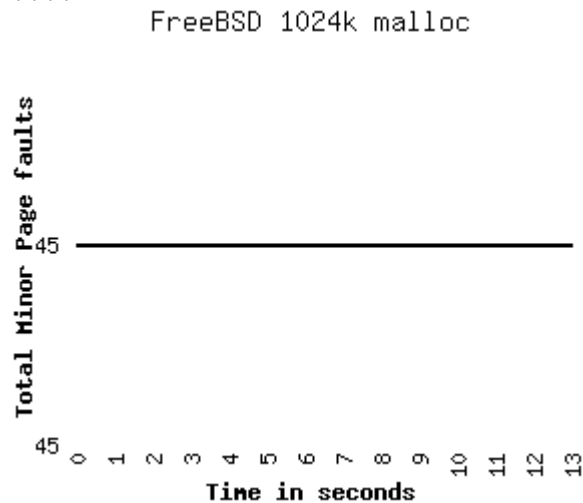
	Linux	Tru64 UNIX	Linux	FreeBSD
Architecture	Alpha	Alpha	Intel	Intel
Allocation	mmap	obreak	mmap	Break
Changing Allocation Amount	Yes	Yes	Yes	Yes
Large # of Minor Faults	Yes	No	Yes	No

It appeared as if this problem was unique to Linux, and possibly mmap.

6. Memory Management in Linux/Unix

To understand why different system calls were used (mmap() & break()) when the same lib memory allocation routine (malloc()) was called, it is necessary to understand how memory management in Linux/Unix works.

A Linux/Unix process can have three types of memory allocated on its behalf: stack, heap and mmaped memory.

Stack memory is managed by the operating system, and

is not generally managed by individual processes. Stack memory (or "the stack") usually contains local variables, and information saved during a function call.

Stack memory is automatically allocated by the operating system, when a process needs more. Stack memory is a temporary storage space, which is not guaranteed to remain allocated for the life of a process. Heap and mmaped memory are more permanent areas of memory and remain allocated for the life of a process. Normally, heap and mmapped memory are managed through malloc, but they can also be managed independently. (A process can call the memory allocation system calls directly to bypass malloc.)

Heap memory (or "the heap") is managed by the brk() system call. The brk() system call takes one argument which sets the "end of heap" for a process. If brk() is passed a value greater than the process's current brk() value, the size of a process's heap grows to the new value, and the operating system reserves more memory for the process. If the value passed to brk() is less than the current brk() value, the size of a process's heap shrinks to the new value, and the operating system will free memory from the process. (break(), brk() and obrk() are different names for the same system call 'brk()')

Mmaped memory is managed by the mmap() and munmap() system calls. When a piece of mmapped memory is to be allocated, mmap() is called the with size of the requested memory. A pointer to the memory is returned, which is used by the process. When the memory is to be deallocated, the pointer is passed to the munmap() system call, and the operating system deallocates the memory.

Use of mmap()/munmap() is more flexible than brk(), but it has more size restrictions and a higher overhead per allocation. If a piece of memory allocated with brk() is not at the end of the heap when it is freed, it can not be released back to the system as free memory, because the brk() interface only allows the end of heap memory to be specified. mmap() & munmap do not suffer this problem.

Some mallocs, GNU libc's in particular, use both heap memory and stack memory to fulfill allocation. Which type of memory is used depends on the size of the allocation request.

7. Focusing on the Linux problem

It was reasoned that at some point below one megabyte allocations, malloc would start to behave more like a traditional malloc(), using brk() instead of mmap(). As a result, a test program was rewritten to allow kilobytes to be specified as an allocation amount instead of megabytes.

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]){
  char *buffer;
  int num_byte;

  num_byte = atoi(argv[1])*1024;

  while(1)
    {buffer=malloc(num_byte);
     free(buffer);}
}
```

After further investigation under linux, it appeared that 128k was an important malloc threshold. Three memory allocations close to 128k in size (126k, 127k and 128k) yielded very different results.

7.1 126k allocation

When malloc was called with an allocation request of 126k, brk() was used to allocate the memory. free() did not release the memory; once the end of the heap was set to "0x8069000", it did not change. Minor page faults did not occur.

```
strace ./pagefault 126
....
brk(0) = 0x804965c
brk(0x8068e74) = 0x8068e74
brk(0x8069000) = 0x8069000
(Nothing further)
```
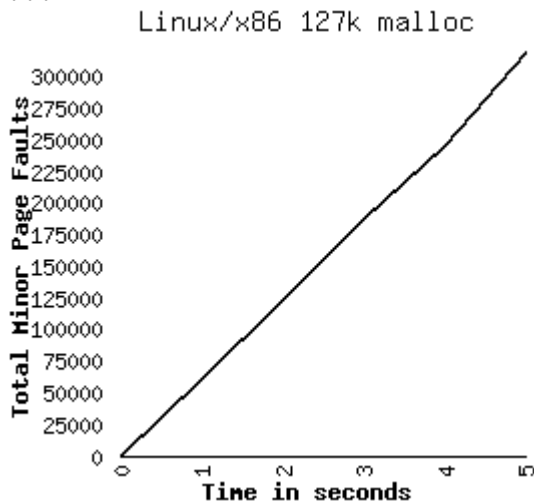
Linux/x86 126k malloc
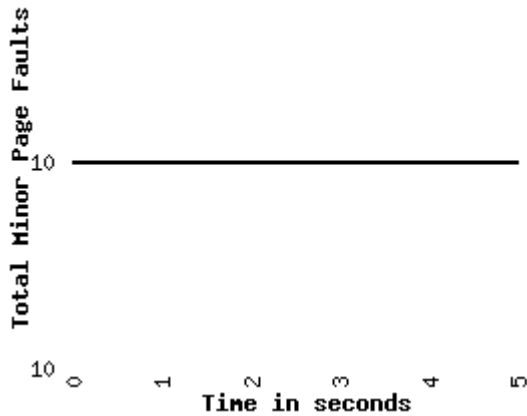
7.2 127k allocation

When malloc was called with an allocation request of 127k, brk() was used to allocate the memory. free() released the memory; the end of the heap fluctuated between 0x806a000 and 0x804a000. A large number of minor page faults occurred.

```
strace ./pagefault 127
....
brk(0) = 0x804965c
brk(0x8069274) = 0x8069274
brk(0x806a000) = 0x806a000
brk(0x804a000) = 0x804a000
brk(0x806a000) = 0x806a000
brk(0x804a000) = 0x804a000
...
```

Linux/x86 127k malloc

7.3 128k allocation

When malloc was called with an allocation request of 128k, mmap was used to allocate the memory. free()

released the memory, as the repeated calls to mmap showed. A large number of minor page faults occurred.

```
strace ./pagefault 128
old_mmap(NULL, 135168,
        PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS,
        -1, 0) = 0x40116000
munmap(0x40116000, 135168) = 0
old_mmap(NULL, 135168,
        PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS,
        -1, 0) = 0x40116000
munmap(0x40116000, 135168) = 0
old_mmap(NULL, 135168,
        PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS,
        -1, 0) = 0x40116000
munmap(0x40116000, 135168) = 0
old_mmap(NULL, 135168,
        PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS,
        -1, 0) = 0x40116000
munmap(0x40116000, 135168) = 0
```

Linux/x86 128k malloc
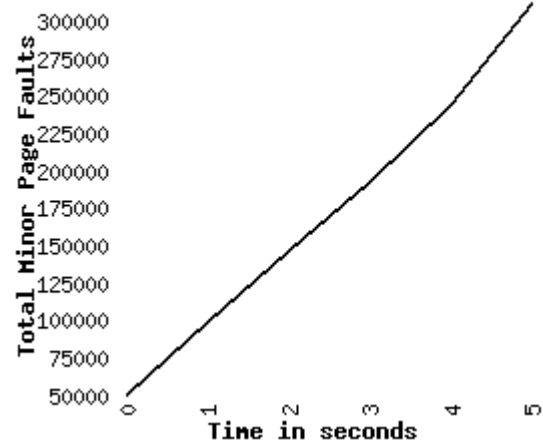
Allocation Size	126k	127k	128k
Linux Allocation Method	brk	brk	mmap
Changing Allocation Amount	No	Yes	Yes
Large # of Minor Faults	No	Yes	Yes

It appeared that mmap() was only part of the story, and that malloc and free worked differently depending on the amount of memory requested and freed.

8. Memory in slow motion

It was unclear at this point whether the malloc() function or the free() function was to blame for the high number of minor faults. To test why the faults were occurring, the author slowed down the loop by placing five second delays before both malloc and free. Fortunately, this would also put a five second break between image initialization (where faults legitimately occur) and the first malloc.

The following program was used:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]){
  char *buffer;
  int num_byte;
  num_byte = atoi(argv[1])*1024;
  while(1)
     { sleep(5);
      buffer=malloc(num_byte);
      sleep(5);
      free(buffer); }
}
```

While running the program with both a 127k and 128k call to malloc (brk & mmap() version), minor faults occurred only when the memory footprint of the image increased. This happened whenever a malloc occurred. Therefore malloc() was the cause of the page faults.

It is interesting to note that a single call to malloc caused a single page fault. The high minor fault count above was the result of malloc being called many, many times.
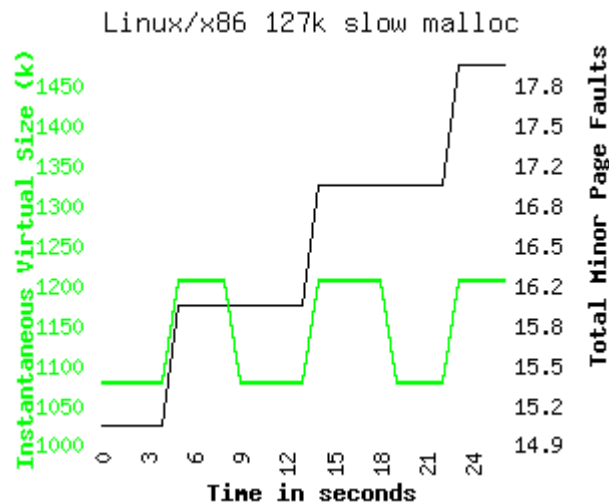
The number of minor faults increased when the process's virtual size increased. Memory allocation appears to cause the fault.
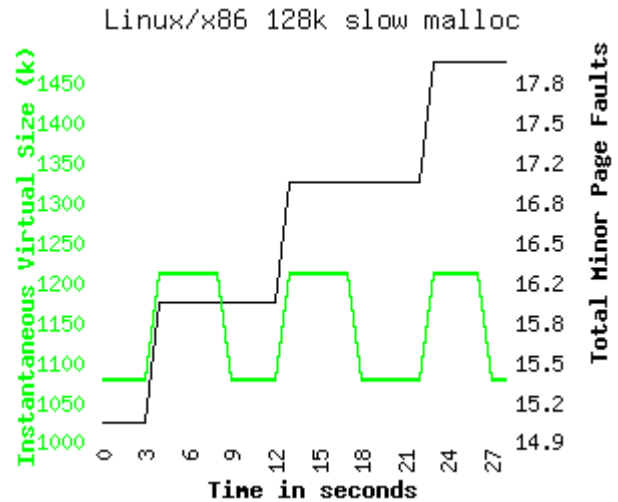
Similar results are seen for a 128k call to malloc. (when mmap is being used instead of brk())

Mallocing memory appeared to be the cause of the page faults. It was unclear whether any use of the brk() system call caused the single minor fault, or this was an oddity of GNU libc's malloc.

To determine where the blame lay, a simple program was written which used the brk() system call to change the amount of allocated heap in much the same way that malloc would call brk().

The following program is basically the same as the "malloc/free" program above, only it does its own memory management.

```
#include <stdlib.h>
#include <stdio.h>

#define PAGE_SHIFT 12
#define PAGE_SIZE (1UL << PAGE_SHIFT)
#define PAGE_MASK (~(PAGE_SIZE-1))
#define PAGE_ALIGN(addr)
(((addr)+PAGE_SIZE-1)&PAGE_MASK)

int main(int argc, char *argv[]){
 char *buffer;

 /* Page-aligned start of heap */
 void *heap=PAGE_ALIGN(sbrk(0));

 int num_byte;
 num_byte = atoi(argv[1])*1024;

 while(1)
  { sleep(5);
    /* Increase the start address.*/
    brk(heap+num_byte);
    sleep(5);
```
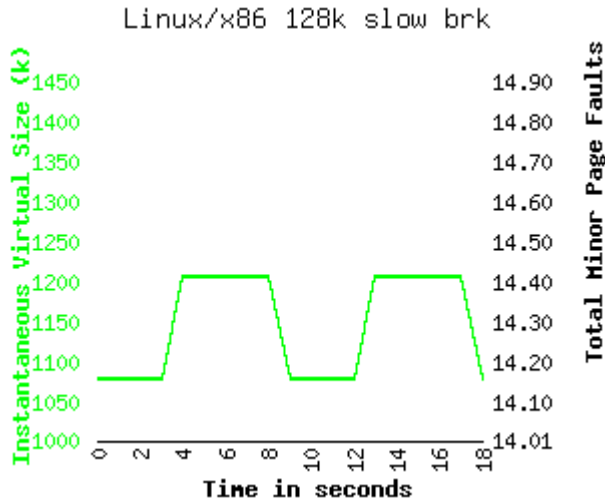
```
      /* Reset the start address. */
      brk(heap);}
}
```

Linux/x86 128k slow brk

When run, each brk() statement did not produce a minor fault. The linux kernel was not causing the minor faults.

It appeared that GNU libc's malloc was the cause of the faults. This would explain why Tru64 UNIX and FreeBSD did not exhibit the problem. Neither used GNU libc.

9. Malloc: How Can It Be Tuned?

The next step was to download the GNU libc, and investigate the malloc source.

Exploration of the malloc.c file revealed a function "mallopt" which could be used to tune the way that GNU libc's malloc performs. [10]

Two options looked interesting:

M_TRIM_THRESHOLD
This is the minimum size (in bytes) of the top-most, releasable chunk that will cause sbrk to be called with a negative argument in order to return memory to the system. [11]

M_MMAP_THRESHOLD
All chunks larger than this value are allocated outside the normal heap, using the mmap system call. This way it is guaranteed that the memory for these chunks can be returned to the system on free.

The page fault program was modified as shown below to turn off malloc trimming.

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

int main(int argc, char *argv[]){
  char *buffer;
  int num_byte;
  num_byte = atoi(argv[1])*1024;

  mallopt(M_TRIM_THRESHOLD,-1);

  while(1)
    { sleep(5);
      buffer=malloc(num_byte);
      sleep(5);
      free(buffer); }
}
```
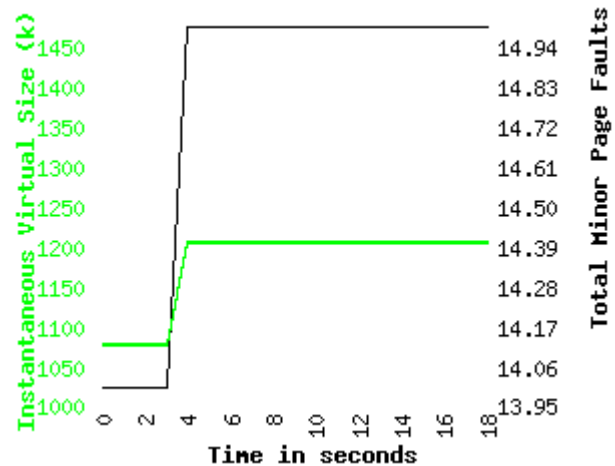
When malloc trimming was turned off and malloc was using brk(), free() did not return memory to the system.

```
 strace ./pagefault3 127
....
brk(0)                  = 0x8049690
brk(0x80692a8)          = 0x80692a8
brk(0x806a000)          = 0x806a000
```

Linux/x86 127k slow malloc w/TRIM

The minor page faults for the malloc of 127k (when malloc used brk()) with malloc trimming turned off.

However, when malloc trimming was turned off and malloc was using mmap, free() did return memory to system.
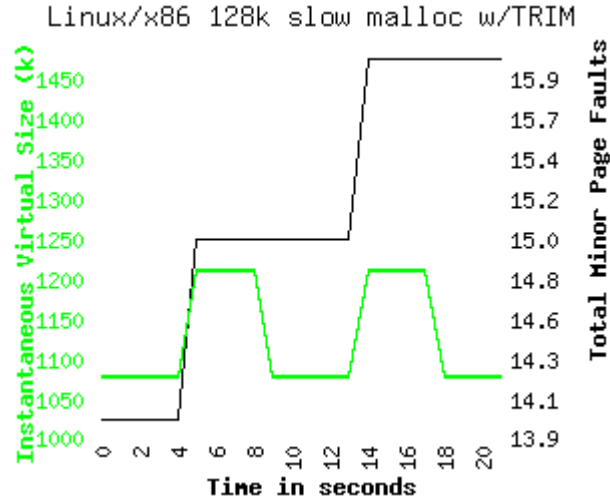
```
strace ./pagefault3 128
...
old_mmap(NULL, 135168,
        PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS,
        -1, 0) = 0x4014e000
```

```
munmap(0x4014e000, 135168) = 0
old_mmap(NULL, 135168,
        PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS,
        -1, 0) = 0x4014e000
munmap(0x4014e000, 135168) = 0
```


Linux/x86 128k slow malloc w/TRIM

Disabling malloc trimming did NOT remove the minor page faults for the malloc of 128k (when malloc used mmap())

Disabling Trimming was half of the solution to the puzzle. Minor faults stopped occurring when using brk() version of malloc, but not the mmap() version. Thankfully, GNU malloc allows us to turn off malloc's use of mmap() by setting "M_MMAP_MAX" option to 0, as shown in the following program.

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

int main(int argc, char *argv[]){
  char *buffer;
  int num_byte;
  num_byte = atoi(argv[1])*1024;

  /* Turn off malloc trimming. */
  mallopt(M_TRIM_THRESHOLD,-1);
  /* Turn off mmap usage. */
  mallopt(M_MMAP_MAX, 0);

  while(1)
    { buffer=malloc(num_byte);
      free(buffer);}
}
```
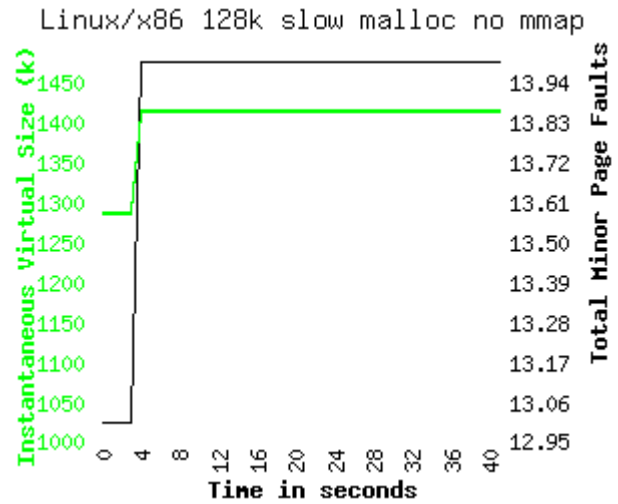
All mallocs used brk(), no memory was returned to the system and no page faults occurred. Success!

```
./pagefault 4 128
...
brk(0)          = 0x804974c
brk(0x8069764) = 0x8069764
```

```
brk(0x806a000) = 0x806a000
```


Linux/x86 128k slow malloc no mmap

Allocation Size	127k	127k	128k	128k
Linux Allocation Method	brk	brk	mmap	brk
Malloc Trimming	No	Yes	Yes	Yes
Increasing # of page faults	Yes	No	Yes	No

To stop the increasing number of page faults, it was necessary to turn off both malloc trimming, and the use of memory mapping. As a result, once allocated, memory was never returned to the system.

10. Fine Tuning

Although calling mallopt allowed a program to perform better, recompiling and changing source code to tune for a particular version of malloc was not a clean solution to the performance problem.

Fortunately, GNU libc's malloc could also be tuned through environmental variables, which were nearly identical to mallopt options. By setting the environmental variables **MALLOC_MMAP_MAX_** to "**0**" and **MALLOC_TRIM_THRESHOLD_** to "**-1**", malloc behaved as if **mallopt(M_MMAP_MAX,0)** and **mallopt(M_TRIM_THRESHOLD,-1)** were called.

Setting these variables showed dramatic speedup in the user's chemistry code, and a significant reduction in the amount of time spent in the system. (This is without a change of a single line of code!)

Malloc	User (sec)	System (sec)	Elapsed	Major Faults	Minor Faults
Normal	216.0	166.7	6:29.20	170	23099385
Tuned	196.7	14.3	3:41.71	168	16820

Initial Linux Chemistry Code

Tuned Linux Chemistry Code
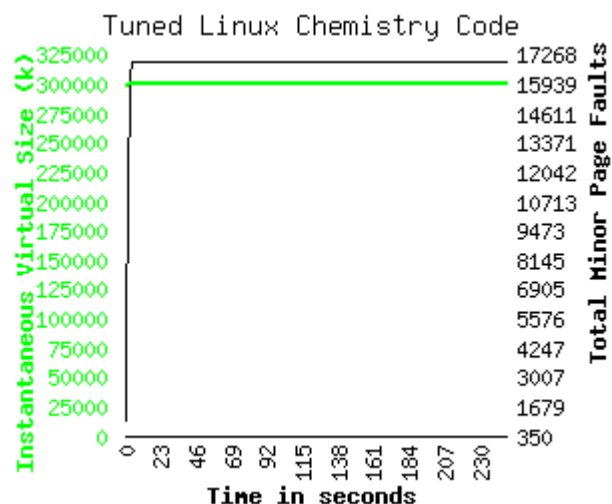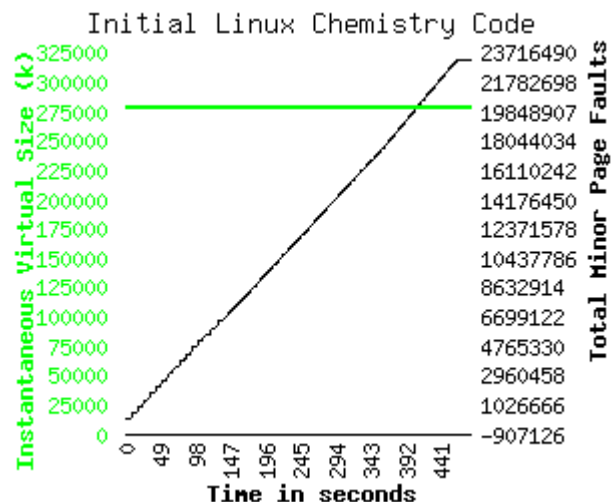
(Notice the difference in the scale of the minor faults)

It is interesting to note that the tuned Linux/Alpha code now behaved much like the high performing Tru64 UNIX code.

11. Why memory map at all?

If the performance of malloc when using mmap() was worse than using brk(), why did GNU libc designers decided to use it at all?

The info pages of GNU libc give a explanation[12]:

"Very large blocks (much larger than a page) are allocated with mmap (anonymous or via /dev/zero) by this implementation. This has the great advantage that these chunks are returned to the system immediately when they are freed. Therefore, it cannot happen that a large chunk becomes "locked" in between smaller ones and even after calling free wastes memory. The size threshold for mmap to be used can be adjusted with mallopt. The use of mmap can also be disabled completely."

It appears that GNU libc is tuned for system wide efficiency in memory usage, instead of raw performance. Using brk() instead of mmap() could cause memory that has been freed to be locked in place, becoming unused. This fits with the preceding experiments. Notice that the tuned Linux/Alpha code has a virtual set size that is about 10% bigger than the non-tuned code.

MicroQuill, makers of SmartHeap, describe[13] the differences between brk and mmap as follows:
"mmap_vs._sbrk"
"The sbrk() approach grows the heap and the process address space in page increments as the sum of allocations and unallocated, fragmented heap space increases. Most tasks eventually reach some typical maximum heap footprint, which remains constant with time. This technique is most efficient and is the default.

The mmap() approach grows the heap and the process address space as required to contain all of the current allocations. Large unallocated blocks of the heap are returned to the OS for use elsewhere in the system. Of course, some of the heap will remain unallocated and fragmented. This technique is less efficient, but is well suited to a few situations in which the sbrk() technique runs out of heap space prematurely. Our recommendation is to adopt the sbrk() approach for maximum flexibility and performance. If a problem is observed in your environment with excessive process address space, then you should consider trying the mmap() build to see if it helps.
....
be aware that mmap is significantly slower than sbrk."

12. Summary

When allocating and deallocating large (>128k) amounts of memory on Linux, the default memory management tunings have a high performance penalty. By using the brk() with no malloc trimming to allocate memory instead of malloc trimming and mmap(), the number of minor page faults decreases, and the performance of malloc increases.

Before running the performance sensitive program, to improve malloc performance, turn off mmap usage and malloc trimming, by either:

1) Adding the following code to a program before heavily using malloc:

```
mallopt(M_MAP_MAX,0);
mallopt(M_TRIM_THRESHOLD,-1)
```

2) Setting the following environmental variables: (Note the trailing underscores)

For sh compatible shells:
```
export MALLOC_MMAP_MAX_=0
export MALLOC_TRIM_THRESHOLD_=-1
```

For csh compatible shells:
```
setenv MALLOC_MMAP_MAX_ 0
setenv MALLOC_TRIM_THRESHOLD_ -1
```

13. Thank You

Bill Carr, for his help thinking through a piece of the puzzle, and for his review of the paper.

Jeff Arnold, for providing a clue to the problem, which changed the plan of attack.

John Henning, for his review of the paper, and many suggestions for improvements.

T. Daniel Crawford, for his patience, meticulous problem reports, quick turn around, and his many test runs.

Sarah Ezolt (Wifezilla), for her last minute editing, help and understanding.

14. Copyright Information

[1] http://www.tru64.unix.compaq.com/dcpi

[2] DCPI counts were sampled at the frequency of 126976, and are therefore approximately equal to 1/126976 the number of events that actually occurred.

[3] Cycles are roughly equivalent to the number of cycles spent in an image or function. Cycles can be used to approximate the amount of time spent in a function or image.

[4] Dtbmiss is caused by instructions that require a virtual to physical page mapping which is not found in the data translation buffer. An itbmiss is similar, but the miss occurs in the instruction translation buffer.

[5] http://www.google.com/

[6] http://www.kernelnewbies.org/glossary/

[7] http://www.openvms.compaq.com/

[8] strace is a linux tool that displays all calls, parameters and return values for kernel system calls

[9] http://www.vmware.com/

[10] http://www.gnu.org/manual/glibc-2.0.6/html_node/libc_29.html#SEC29

[11] sbrk() called with a negative argument is the same operation as a brk() being called with a value less than the current value.

[12] info:/libc/Efficiency and Malloc

[13] http://www.microquill.com/kb/faq_ans.htm