Effects of copy-on-write Memory Management on the Response Time of UNIX fork Operations

Jonathan M. Smith and
Gerald Q. Maguire, Jr.
Columbia University

ABSTRACT: We present techniques for analyzing the effect of "copy-on-write" page management strategies on the performance of UNIX *fork()* operations. The analysis techniques are applied on two workstations, the AT&T 3B2/310 and the Hewlett-Packard HP9000/350 in order to determine the relationships between the amount of memory in the parent's data segment, the fraction of this memory which is written by the child, and the improvement in execution time due to "copy-on-write." Since the implementation of "copy-on-write" is straightforward with modern MMUs, our results for these workstations are readily generalized to other workstations.

The results show that the size of the parent's allocated memory has little direct effect on performance, due to the fact that only page table entries are copied during the *fork()* operations. The execution time is most influenced by the amount of

This work was supported in part by equipment grants from AT&T and the Hewlett-Packard Corporation, and in part by NSF grant CDR-84-21402.

memory that must be copied, which can be determined from the product of memory allocated and the fraction of memory written. Thus, the worst case occurs when large address space programs update much of their memory.

In order to observe what occurs in practice, we measured two programs that have what are currently considered large address spaces. These programs, which we believe to be representative of the sorts of programs which use large amounts of system resources, updated less than half of the memory in their data segments.

---

1. Introduction

"... *the sole test of the validity of any idea is experiment.*"
[Feynman et al. 1963]

The UNIX *fork()* operation creates a copy of the calling process which is differentiated from its creator by the return value of *fork()*. The two processes have separate address spaces. Traditionally, UNIX systems actually copied the contents of the caller's address space to create the new process. Since the portion of the address space containing executable code was read-only, copying was not needed and an incremented reference count and text table entry sufficed [Ritchie & Thompson 1978]. Clearly, the *fork()* operation can be expensive in system resources. Thus, some attempts were made to take advantage of special cases. An example is the 4.2BSD [Joy 1982] *vfork()* call, which does not make a copy of the address space for the new process but instead allows it to share the address space with its creator. The creator is not runnable until the new process has replaced its image via an *exec()* operation. The *exec()* operation replaces the caller's image with an image derived from the contents of the named executable file.

It is common for the operation which immediately follows a *fork()* operation (after some descriptor manipulation) to be an *exec()* operation. In particular, this happens very frequently in the shell [Bourne 1978], which is the main user interface to UNIX. Thus, *vfork()*, in not copying, avoids unneeded work. However, the shared, rather than copied, address spaces force the programmer to be very aware of the differences between *fork()* and *vfork()*.

1.1 Copy-on-write

Another approach is to transparently alter the implementation of *fork()* to take advantage of favorable circumstances such as the shell's usage. This is done with a so-called "copy-on-write" *fork()*, where portions of addressable memory are shared until such time as they are changed. Similar memory management is done in TENEX [Bobrow et al. 1972] and more recently, Mach [Young et al. 1987]. Each process has a page table which maps its virtual addresses to physical addresses; when the *fork()* operation is performed, the new process has a new page table created in which each entry is marked with a "copy-on-write" flag; this is also done for the caller's address space. When the contents of memory are to be updated, the flag is checked. If it is set, a new page is allocated, the data from the old page copied, the update is made on the new page, and the "copy-on-write" flag is cleared for the new page. Thus, unexpected changes to shared state do not occur, as independent copies are created "on demand." This is very effective in the special case of the shell, where almost no copying has to be done before an *exec()* replaces the address space. A thorough description of the mechanism as implemented in UNIX is given by Bach [Bach 1986].

1.2 Motivation

In another paper [Smith & Ioannidis 1988] we discuss an implementation of a mechanism to *fork()* a process on a remote workstation; the major cost in execution time is incurred by data copying. Thus, we were interested in reducing the amount of copying, especially that which takes place over a communications channel. One strategy which we devised (assuming either homogeneous

software configurations on the workstations or NFS-available [Sandberg et al. 1985] binaries) was to have program images available on the remote system and send only the changes [Maguire 1988] which have been made to the address space, i.e., those which would be copied by a "copy-on-write" scheme. In order to understand the engineering tradeoffs, we examined the local case in some detail.

The arguments presented for "copy-on-write" have so far been qualitative; we felt that detailed quantitative data were necessary. The methodology and process of gathering these data are discussed in Section 2. Section 3 provides an analysis of the data. Section 4 examines the memory-update characteristics of some programs having desirable properties as described in Section 3, and Section 5 concludes with a discussion of our results.

2. *Data Acquisition*

There are two parameters of interest, i.e., the size of the storage to be "copied" in the new process and the fraction (between 0.0 and 1.0) of memory references which are writes. The number of times each parameter was exercised was also made variable, in order to remove various small-sample artifacts that can occur. Such artifacts are illustrated by the plots in Figures 4 and 5. The desired data were gathered with the C program presented as an Appendix, *do_fork.c*. A script was written in order to drive the *do_fork()* program with various values; the values used for the measurements described in this report were gathered with this shell script:

```
if [ ! -f do_fork ]
then
   echo "Making do_fork."
   make do_fork
fi
if [ ! -f do_fork ]
then
   echo "No do_fork. Exiting."
   exit 1
fi
```

```
echo "size do_fork:"
size do_fork
for forks in 0 1 3 10 32 100 316 1000
do
   for heap_size in \
      0 1000 3162 10000 31622 100000 316228
   do
      for write_frac in 0.0 0.1 0.3 0.5 0.7 0.9 1.0
      do
         echo "time do_fork $forks $heap_size \
       $write_frac"
         time do_fork $forks $heap_size $write_frac
      done
   done
done
```

The script first ensures that an executable *do_fork* binary is available, attempting to make one if not. Once *do_fork* is available, it is invoked in the innermost of three nested loops, which vary its parameters controlling the number of *fork()* operations to be executed, the size of the heap to allocate, and the fraction of the allocated heap which is to be written to. Prior to each invocation, a message is written with *echo*, stating what the invocation parameters of *do_fork* are.

Data sets for analysis by S [Becker & Chambers 1984] are then created using the shell script, by, e.g. for the 3B2,

```
script 2>&1 | \
   grep "^real" | \
   cut -f2 | \
   awk '{ i=index($0,m); m=substr($0,1,i-1); \
   s=substr($0,i+1,length($0)-i-1); s=60*m+s; \
   print s}' > real.3B2
```

and reading the list of numbers into an S vector. The following data sets were extracted from the script output:

number The number of times an invocation of *do_fork* was to create a child process. The values 0, 1, 3, 10, 32, 100, 316 and 1000 (0 plus powers of *sqrt(10)* were selected in order to make both order of magnitude induced effects (as we are changing by orders of magnitude) and implementation artifacts (because we start at small values, e.g., 0 and 1) visible.

mem The number of bytes allocated to the process's heap, via *malloc()*. The values 0, 1,000, 3,162, 10,000, 31,622, 100,000 and 316,228 were chosen for both artifact and order of magnitude visibility, as discussed previously; the extra factor of 1000 (over the values of *number*) is to compensate for the page size, since otherwise it would require (for a 2K page[1]) 8 values before we accessed a page other than the first one. Clearly, there is no practical difference between 316,228 and 310K; it is merely aesthetically appealing to use the correct digits.

frac The fraction of memory which is to be written (actually, we write one byte per page in order that the memory access loop not contribute to the response time beyond causing faults). The interesting boundary values of 0.0 and 1.0 were chosen, as well as the values 0.1, 0.3, 0.5, 0.7, and 0.9, which were chosen for their coverage of the input domain.

real The real time, in seconds, printed by an invocation of `time do_fork` with the parameters as set in the other vectors.

user Likewise for user time.

sys Likewise for system time.

3. Data Analysis

Given the data discussed in the previous section, we wish to analyze the data in order that we can qualitatively discuss the effects of "copy-on-write" page management on response time. One of the difficulties is that by our experimental design, the measured response time is a function of not one, but three quantities, *number*, *mem*, and *frac*. There are two obvious hypotheses which we can use our analysis to refute or verify. First, that the response time increases as the size of the data segment increases, for a fixed fraction of write references. Second, that the response

1. HP-UX on our HP9000/350s uses a 4K pagesize, but given the instrumentation (e.g., *do_fork.c*) the difference is rarely relevant; in fact, only when the offset of a particular byte in the last page accessed causes an extra 2K bytes of memory to be paged in.

time increases as the fraction of write references increases, for a fixed data segment size.

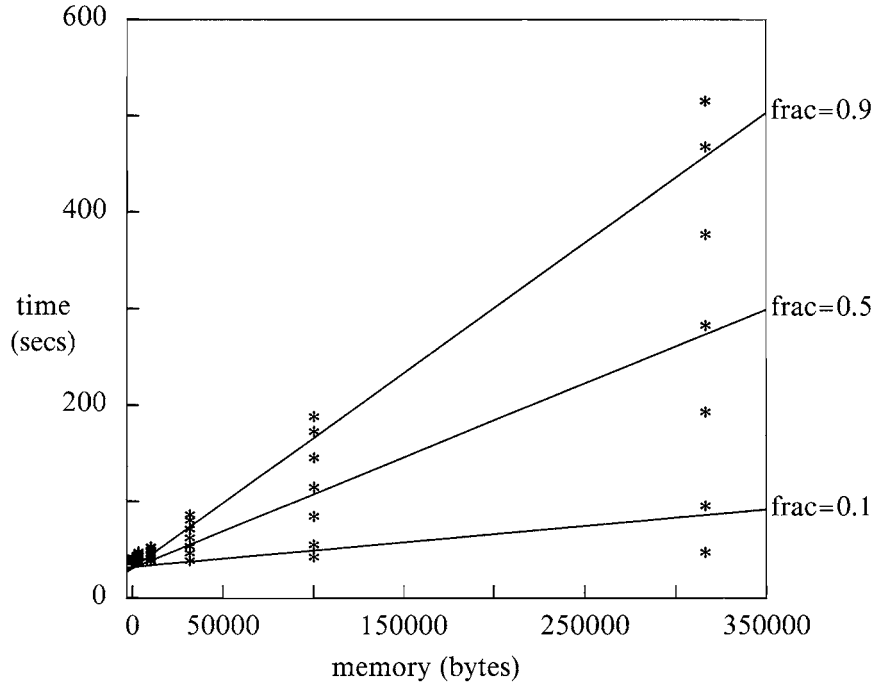

Figure 1: Effect of fraction of memory written

Figure 1 shows *mem* plotted on the x axis against *real* on the y axis for an AT&T 3B2/310 with 2 megabytes of memory (of which 1.2 megabytes are available to user processes), a 30 megabyte hard disk, and running UNIX System V, Release 3.0, Version 2. All times are given in units of seconds. We have fixed the value of *number* to be 1000 to remove artifacts. The dependent variable, plotted vertically, is the real time, in seconds. The independent variable, the size of the data memory in bytes, is on the horizontal axis. Regression lines are drawn through the plotted points corresponding to *frac* values of 0.1, 0.5, and 0.9. These regression lines have equations

```
y = 1.709e-4*x+31.4
y = 7.670e-4*x+30.5
y = 1.349e-3*x+30.7
```

for the respective *frac* values. Thus, with these equations, we

could estimate that a process with a 1 megabyte data segment which writes into half of that segment would take about 800 (797.5 = 7.67e−4*1.0e6+30.5) seconds of real time to perform 1000 *fork()* operations. It is obvious that the lines fit the plotted points quite well, indicating that the relationship is quite close to linear.

The same data are plotted for a Hewlett-Packard HP9000/350 with 8 megabytes of main memory and a 70 megabyte hard disk, running HP-UX 6.0 (same units, restrictions, and axis markings) in Figure 2.

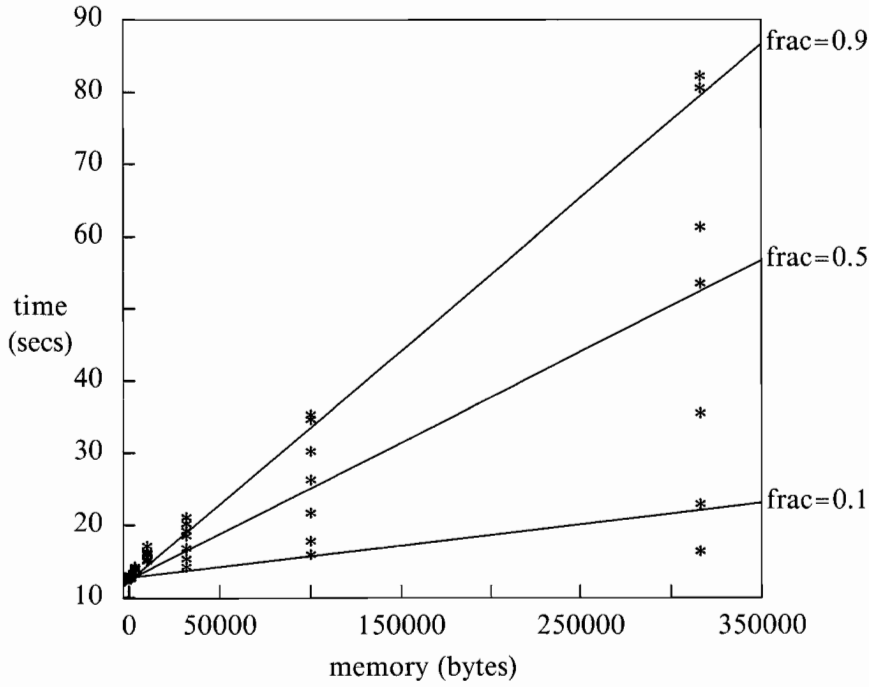

Figure 2: Effect of write fraction (HP-UX)

The equations for the lines with *frac* set to 0.1, 0.5, and 0.9 are

```
y=2.952e−5*x+12.7
y=1.264e−4*x+12.4
y=2.124e−4*x+12.2
```

respectively. The effect of the faster processor in the HP9000/350 is clear from the extent of the y axis in this figure versus that of the previous one. In fact, the important parameter in comparing processor speeds under this workload is memory-copying speed.

In order to measure this, we wrote a short C program which took the number of bytes to copy as an argument, the relevant fragment of which is:

```
p = malloc( size );
pre_page( p );
clock = times( &tb1 );
memcpy( p, p, size );
clock = times( &tb2 ) - clock;
```

For *size* set to 316,228 and a page size of 2K bytes (4K on the HP9000) we measured 0.40 seconds of real time, 0.39 seconds of user time, and 0.00 seconds of system time on the 3B2/310. The values were 0.06 seconds of real time, 0.06 seconds of user time, and 0.00 seconds of system time on the HP9000/350. These values held true through several trials, and show that for memory-copying the HP is about (to the limited accuracy of the measurements) 6.7 times faster than the 3B2. They also provide an upper bound on the memory copy rate which can be used to evaluate overhead incurred by page management operations. For the HP, we get 5M (5,270,467 = 316,228/0.06) bytes per second, or about 1,300 (1,286 = 5,270,467/4,096) $4K^2$ pages per second. For the 3B2, we get 0.8M (810,841 = 316,228/0.39) bytes per second or about 400 (396 = 810,841/2,048) 2K pages per second.

We can use the regression lines we have presented for further analysis. The y-intercept (about 31 seconds for the 3B2/310 and 12 seconds for the HP9000/350) should represent the time required for 1000 forks which allocate 0 bytes of memory; examination of the script output confirms that this figure is accurate. Since *do_fork* is written to be compact (no standard I/O, etc.) this should in fact accurately indicate the cost of performing a *fork* when divided by the number of operations performed. Thus, using the computed intercepts we have given for *number* set to 1000, the average 3B2/310 *fork* requires about 31 milliseconds of real time (=(31.4+30.5+30.7)/(3*1000)). For a fixed number of *fork()* operations the y-intercept is not nearly as interesting as the slope of the line. We should note that in reality, the function is not a line, as the quantization of bytes into page size quantities forces a

2. For comparison purposes, this would be about 2,600 (2,573 = 5,270,467/2,048) 2K pages per second.

staircase function. However, for purposes of analysis we can assume that a linear function exists. The slope of the line for some known value of *frac* gives the relationship between changes in *real* caused by changes in *mem*. Hence, we can use the slope of the regression line to estimate the rate at which page faults are serviced. Mem · frac gives a fixed amount of memory, with which we use the equation of the regression line to compute a real time estimate. Then, the observed page fault service rate can be computed with the simple formula $\frac{\text{mem} \cdot \text{frac}}{\textit{real time}}$. In fact, the slope of the line can be used to compute the service rate directly, for a known value of *frac* and *number*; this rate is given by $\frac{\text{number} \cdot \text{frac}}{\textit{slope}}$, which calculates a value in units of bytes per second. For the 3B2, these values are 585,138, 651,890, and 667,161 (286, 319, and 326 2K pages per second, respectively) for the three values of *frac* plotted. The corresponding values for the HP9000 are 3,387,534, 3,955,696, and 4,237,288 bytes per second (827, 965, and 1,034 4K[3] pages per second, respectively). Using the best observed page fault service rates for each processor, we calculate the ratio of the page fault service rate and the time for memory-copying, which is 0.823 (=667,161/810,841) for the 3B2, and 0.804 (=4,237,288/5,270,466) for the HP. Values for the ratio can range between 0 and 1; the best case is a value near 1, as this indicates that the virtual memory management incurs very little overhead. We can in fact estimate this overhead using the information we have. Using $\tau()$ to measure time, we know that

$$\tau(\textit{fork}) = \tau(\textit{copy one page}) \cdot \text{frac} \cdot \#\,\textit{pages} +$$
$$\tau(\textit{overhead for page table entry}) \cdot \#\,\textit{pages} +$$
$$\tau(\textit{overhead to create new process})$$

Now, $\tau(\textit{copy one page})$ is really a function of the hardware components (e.g., bus, processor, memory) comprising a system, and we've shown how it can be gathered with a small auxiliary program. But from our numbers and analysis we can get $\tau(\textit{overhead for page table entry})$ and $\tau(\textit{overhead to create new process})$. Thus, for

3. For comparison, 1,655, 1,932, and 2,069 2K pages per second.

any given *fork* operation, the time required is completely parameterized by τ(*copy one page*), τ(*overhead for page table entry*), τ(*overhead to create new process*), *frac*, and *mem* (# pages). The key is that the first three are determined by the system characteristics, and they can thus be precomputed; the application-dependent influences are completely encapsulated in the latter two parameters. Thus, an application can be characterized on a given system by its size in pages and the fraction of those pages which are written to.

3.1 Relationships

The shapes of the plots we generated are quite similar for both processors, with the HP9000 plots time values scaled because it's significantly faster than the 3B2. We'll use the 3B2 to illustrate the analysis in the remaining figures.

One of the failings of the x-y plots is that there are two independent variables. The perspective plot shows that the real

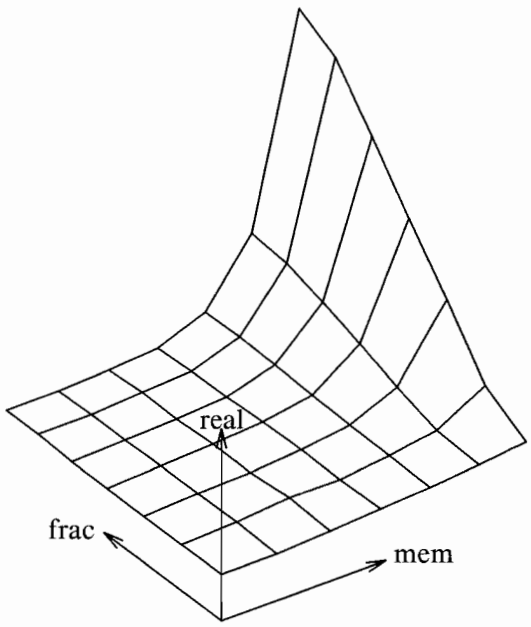

Figure 3: Perspective plot, *mem* versus *frac* versus *real*

time increases as a *product* of *mem* and */frac*; the maximum value is 505.82 seconds, for *number* at 1,000, *mem* at 316,228, and */frac* at 1.0. Figure 3 shows a 3-D perspective plot of *real* (z-axis), *mem* (x-axis), and */frac* (y-axis). The point (316,228, 1.0, 505.82) is the furthest, highest point on the graph. Thus *mem* is increasing from our left to our right (it's exponential as the data values are chosen to increase exponentially), and */frac* is increasing from our right to our left, moving away from us. Figure 1 would be then be overlaid cross-sections taken from the perspective plot by intersecting a series of y-planes with it. Of course, not all the data of Figure 1 are available due to hidden line elimination. We've limited the data shown in Figure 3 to that gathered with *number* set to 1000. This was done after analysis of the raw data showed two things which limited the value of the data gathered for a small number of */fork* operations. First, there was very little opportunity for the data to become evident against the overhead of executing the parent program. This could, of course, have been removed by calling *times()* from inside *do_fork*, but given our strong preference

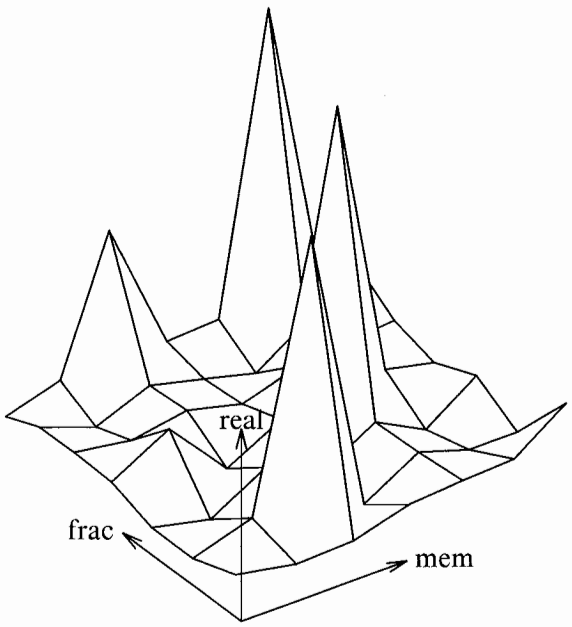

Figure 4: Perspective plot, *mem* versus */frac* versus *real*

for the shell as a measurement apparatus, this was not done. Second, the timing data were apparently overwhelmed by other sampling noise, such as that caused by various background processes and network daemons (although the processors were otherwise idle). These were not shut down due to the deleterious effect on our working environment.

If we plot a 3-D perspective plot with parameters as before, except that *number* is 1, we get Figure 4, which demonstrates what sort of artifacts, or "noise" can arise due to inadequate sample size. In fact, the question might be raised as to why *real* time is used, rather than *sys*. Philosophically, the *real* time is what is most relevant to an observer. Scientifically, analysis shows that for *number* large, *real* is less than 20 percent greater than *sys*, and that they are closely correlated. This is illustrated in Figure 5, where the x axis has values of *number*, and the y axis is the value $\frac{real - sys}{sys}$. The plot shows that the relationship between *real* and

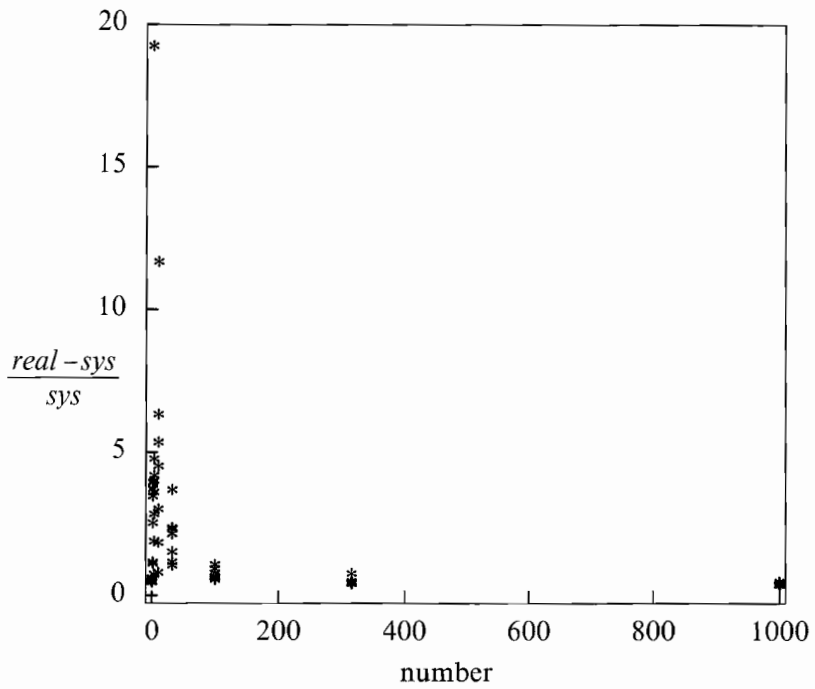

Figure 5: Relation between *real* and *sys*

sys is not very good for small values of *number*; they differ by almost a factor of 20. However, things improve as *number* gets larger; a detailed graph is provided in Figure 6 by restricting *number* to values of 100 or more. It's clear from this illustration that for *number* at 1000, *sys* and *real* are reasonably good approximations of each other. Incidentally, we should note that for small values of *number* (e.g., 1), *sys* is subject to the same noise problem that *real* suffers from; this is easily observed with a perspective plot, which we will not present for space considerations.

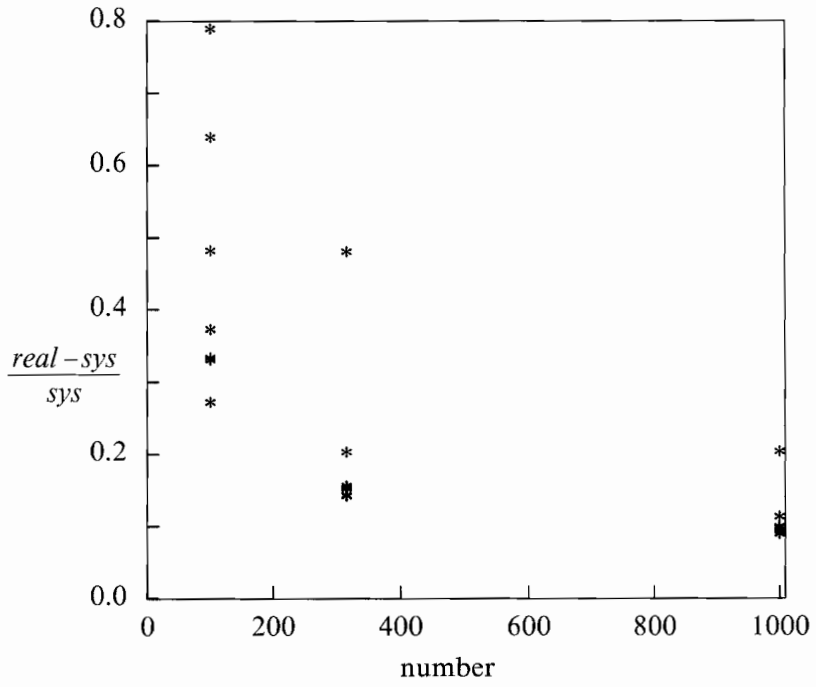

Figure 6: Relation between *real* and *sys*

4. *Write Fraction for Real Programs*

In the last section, we saw that the factors *mem* and *frac* influenced the real time The biggest savings for the "copy-on-write" scheme would come from programs with large address spaces which updated a small fraction of their data before exiting or *exec()*ing a new binary. As discussed before, this works well for

the shell, but the shell typically uses very little of its data segment. While it may expand the address space as necessary to store new variables or metacharacter expansions, this does not account for many pages. We thus sought programs with large address spaces, in order to see what effect the "copy-on-write" scheme would have.

As they had the largest address spaces (of programs in common use in our department), we set out to take some measurements of the memory utilization of two symbolic interpreters. We chose 4.2BSD's [BSD 1982] Franz Lisp (Opus 38.92), as it is widely available. Another less detailed set of measurements was taken using the GNU Emacs [Stallman 1986] LISP interpreter, which is also widely available. These measurements were taken on a DEC VAX-11/750, because both pieces of software were available there (Franz Lisp is not available on our HPs and 3B2s, although GNU Emacs is). Since we are measuring data segment utilization, and the machines discussed in this paper all have 32 bit architectures, the measurement results should be portable. This is particularly true because we use relative measures, such as the fraction of the data segment which has changed. While a particular architecture may have a less efficient representation of the data, this should not change the fraction of the data altered by the program significantly.

4.1 Franz Lisp

Our first exercise was choosing a computationally intensive process so that we could gather some statistics on the sort of processes which one would want to improve the performance of [Leland & Ott 1986]; that is, those that consume a lot of resources. Experience with an ABSTRIPS [Sacerdoti 1974] implementation led us to use this system to gather statistics. ABSTRIPS is a "planning" system which works by constructing increasingly detailed series of actions at decreasing levels ("criticality levels") of abstraction. There are primitives defined (in predicate logic) for each level of abstraction; as the levels are traversed, we gradually "flesh out" the details of a plan for achieving the goal. ABSTRIPS relies heavily on the use of a theorem prover; hence, it

is representative of much current AI computation. An example of its output is given in Figure 7.

```
Franz Lisp, Opus 38.92
-> [load abstrips.lsp]
t
-> criticality level: 4
skeleton plan : ((goal c))
criticality level: 3
skeleton plan : ((get-slippers d)
(give-slippers DOG ME) (goal c))
criticality level: 2
skeleton plan : ((gothrudoor c b DOG)
(gothrudoor b a DOG)
(gothrudoor a d DOG)
(get-slippers d)
(gothrudoor d a DOG)
(gothrudoor a b DOG)
(gothrudoor b c DOG)
(give-slippers DOG ME) (goal c))
criticality level: 1
skeleton plan : ((gothrudoor c b DOG)
(gothrudoor b a DOG)
(pushopen a d)
(gothrudoor a d DOG)
(get-slippers d)
(gothrudoor d a DOG)
(gothrudoor a b DOG)
(gothrudoor b c DOG)
(give-slippers DOG ME)
(goal c))
((gothrudoor c b DOG)
(gothrudoor b a DOG)
(pushopen a d)
(gothrudoor a d DOG)
(get-slippers d)
(gothrudoor d a DOG)
(gothrudoor a b DOG)
(gothrudoor b c DOG)
(give-slippers DOG ME) (goal c))
->
```

Figure 7: ABSTRIPS Output

The problem in our example was to have a dog fetch your slippers from another room. This problem takes about 15 minutes to plan on a VAX-11/750; the implementation makes heavy use of recursion and maintains several large lists.

The size of the Franz executable (from the UNIX *size* command) is 139,264 (text) + 511,488 (data). The data on memory usage was obtained by using the UNIX system's ability to create a *core dump* of a process's address space; since the text segment is read-only, only the data and stack segments are dumped. Sending the SIGQUIT signal to a process causes a core dump; this was done at the following points in the execution of the ABSTRIPS planner.

1. When the LISP interpreter was started. This gives us a baseline value, with no program loaded and no code executed. The core dump occupied 528,384 bytes.
2. Immediately after ABSTRIPS was loaded. This tells us how much of the address space change is due to storage of the ABSTRIPS program. The core dump occupied 556,032 bytes; a bytewise comparison with the previous dump showed that 56,937 bytes had changed.
3. Immediately after ABSTRIPS execution is terminated. This tells us how much of the address space has changed during execution. The core dump occupied 613,376 bytes, and differed from the previous dump at 77,910 bytes. The difference between this dump and the first dump was a total of 123,942 bytes changed. No garbage collection was announced.

An important issue is the locality of reference; our measurement programs for the "copy-on-write" fork performance showed that we could write every page by writing one byte on each page. The byte comparison routine delivers addresses where it found differences between two files; the difference in bytes could then be measured by piping the output to `wc -l`; if we divide each address by the pagesize (512 on the VAX) and pass the results to `uniq | wc -l`, we can find the number of pages that have changed; in this case 270 of the 1,198 (=613,376/512) pages changed, for a write fraction of 0.23.

4.2 GNU Emacs

GNU Emacs provides a facility to dump the currently executing image into an executable file. When this file is executed, the state of the Emacs interpreter is restored to the state it had when the (`dump-emacs`) was invoked. We took the following measurements on the VAX/11-750.[4] The size of the GNU Emacs editor we measured was 437,248 (text) + 208,896 (data), determined with *size*. We sought an example program which had the sort of behavior (fairly computation-oriented) that we desired. We based our desire for computationally-intensive examples on the observation that as heavy resource users, these programs would demonstrate the greatest effects from an optimization. GNU Emacs provides a library of LISP code; one of the routines provides a graphic solution of the classic "Towers of Hanoi" problem. We ran the GNU LISP interpreter on the following input:

```
(dump-emacs "pre-hanoi" "/usr/local/emacs" )
(hanoi 10)
(dump-emacs "post-hanoi" "/usr/local/emacs" )
```

(As might be expected, this requires patience at 9600 baud!) The interpreter emitted several messages to the effect that it was performing garbage collection.

At the completion of the computation, we performed a bytewise comparison on the two dump files:

```
$ ls -l post-hanoi pre-hanoi
-rwxr-xr-x  1 jms   phd   851968 Oct 27 08:25 post-hanoi
-rwxr-xr-x  1 jms   phd   737280 Oct 26 16:01 pre-hanoi
```

which showed that 183,312 bytes had changed, which for the computed data segment size of 414,720 (=851,968-437,248) is slightly less than thirty-five percent of the dump; that is, almost the same percentage we had observed with Franz Lisp and ABSTRIPS. Several times during the computation and in the `dump-emacs` function the garbage collector was run. Thus the amount which

4. The results for GNU Emacs were checked on the workstations, and they are fairly consistent. For the "Towers of Hanoi" problem discussed below, the fraction of the data altered by the program was 0.30 on the 3B2 and 0.48 on the HP9000. Much of the difference is due to what features the runnable Emacs is pre-loaded with; the VAX executable has large amounts of pre-loaded information, which is read-only.

appears to have changed may include parts which did *not* change but were relocated and thus appear to have changed. It also compacted storage which appeared to be changed (since newly-allocated storage is considered changed from the previous non-allocated storage). The important point is that these changes would be seen by a page-management mechanism in either case.

5. Conclusions

"Copy-on-write" paging strategies for address space inheritance have been shown to be effective in reducing the real time required to perform UNIX *fork()* operations. This qualitative assessment is based on the quantitative data we gathered and analyzed. For large processes, the time required is proportional to the fraction of write references, so that a child process which updates half (0.5) of its address space will spend half the time doing copying that a child process which updates all (1.0) of its address space will. For a pair of interpreters with large address spaces, we showed that the portion of the address space changed from process startup until process termination was small, typically less than 0.5. These measurements concur with those of Zayas [Zayas 1987], who measured program behavior in an Accent environment, and confirm the desirable properties observed of a similar scheme for fast state transfers to remote systems in the V [Theimer et al. 1985] system.

Thus, if these interpreters or programs which behave similarly were to *fork()* child processes which executed tasks similar to those described, a reduction of 50 percent or more of the system time devoted to copying data might be achieved. This confirms that the scheme for remote *fork()* described in the introduction has considerable merit.

This reduction in copying also reduces the amount of swap space required, reduces the amount of time spent swapping, increases the number of processes which can be run without paging, and decreases the *cost* of context switches (where the cost of paging out the written pages and the paging in of pages which are only read and have not been modified is included). Thus the advantages of the text table are extended to unmodified pages (or

viewed another way UNIX gains via "copy-on-write" the ability to *eliminate* the text table **and** improved *fork()* performance). With respect to these page management strategies, it should be noted that the earlier TENEX had these advantages ten years earlier and needed neither a distinguished text table nor the confusion of two varieties of *fork()*.

Notes

The paper was improved by thoughtful comments from John Ioannidis and Nathaniel Polish.

Appendix A: do_fork.c

```
#include <errno.h>
#include <sys/param.h>

#ifndef NBPC
#define PAGE_SIZE 2048
#else
#define PAGE_SIZE NBPC
#endif

#ifndef NULL
#define NULL 0
#endif

main( argc, argv )
int argc;
char *argv[];
{
   int count = 0, heap_size = 0, pid, status;
   double atof(), write_fraction = 0.0,
      write_count = 0.0, write_size = 0.0;
   register char *ptr;
   char *malloc();
   extern int errno;

   if( argc > 1 )
   {
      count = atoi( argv[1] );
      if( argc > 2 )
      {
         heap_size = atoi( argv[2] );
         if( ( ptr = malloc( heap_size ) )
            == (char *) NULL )
            error( "Insufficient memory available. Exiting.\n" );
         if( argc > 3 )
         {
            write_fraction = atof( argv[3] );
            if( write_fraction < 0.0 || write_fraction > 1.0 )
               error( "0.0 <= writes <= 1.0; Exiting.\n" );
            write_size = write_fraction * (double) heap_size;
         }
      }
   }

   while( count > 0 )
   {
```

```
      switch( (pid = fork()) )
      {
      case -1:   /* failed. If EAGAIN, wait. */
         if( errno == EAGAIN )
            wait( &status );
         break;
      case 0:      /* child. make refs if needed, and exit */
         while( write_count < write_size )
         {
            *ptr = '\0';
            ptr = &ptr[PAGE_SIZE];
            write_count += (double) PAGE_SIZE;
         }
         exit( 0 );
      default:
         count -= 1;
      }
   }
   exit( 0 );
}

error( string )
char *string;
{
   write( 2, string, strlen( string ) );
   exit( 1 );
}
```

References

M. J. Bach, *The Design of the UNIX Operating System,* Prentice-Hall (1986).

Richard A. Becker and John M. Chambers, *S – An Interactive Environment for Data Analysis and Graphics*, Wadsworth, 1984.

D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, TENEX, a Paged Time Sharing System for the PDP-10, *Communications of the ACM* **15**(3) pages 135-143 (March 1972).

S. R. Bourne, The UNIX Shell, *The Bell System Technical Journal* **57**(6, Part 2) pages 1971-1990 (July-August 1978).

BSD, *UNIX User's Manual, 4.2 BSD,* University of California, Berkeley (1982).

Richard P. Feynman, Robert B. Leighton, and Matthew Sands, *The Feynman Lectures on Physics,* Addison-Wesley, Reading, MA (1963).

W. Joy, *4.2BSD System Manual*, 1982.

G. Q. Maguire Jr., PACS for those interested in image processing: an expert configuration system, in *Les Entretiens de Lyon – Computer Science and Life: Medical Imaging and Experts Systems Applied to Medicine* (1988). S.E.E., C.E.R.F. and S.F.B.M.N.

Will E. Leland and Teunis J. Ott, Load-balancing Heuristics and Process Behavior, in *Proceedings, ACM SigMetrics Performance 1986 Conference* (1986).

D. M. Ritchie and K. L. Thompson, The UNIX Time-Sharing System, *Bell System Technical Journal* **57**(6) pages 1905-1930 (July-August 1978).

E. D. Sacerdoti, Planning in a Hierarchy of Abstraction Spaces, *Artificial Intelligence* **5**, (1974).

R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and R. Lyon, The Design and Implementation of the Sun Network File System, pages 119-130 in *USENIX Proceedings* (June 1985).

Jonathan M. Smith and John Ioannidis, Implementing remote *fork()* with checkpoint/restart, Technical Report CUCS-365-88 (supersedes CUCS-275-87), Columbia University Computer Science Department (1988).

Richard Stallman, *GNU Emacs Manual, Fourth Edition, Version 17,* Free Software Foundation, Inc., 100 Mass Ave., Cambridge, MA 02138 (February 1986).

Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton, Preemptable Remote Execution Facilities for the V-System, pages 2-12 in *Proceedings, 10th ACM Symposium on Operating Systems Principles* (1985).

M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63-76 (8-11 November 1987). In *ACM Operating Systems Review 21*:5

E. Zayas, Attacking the Process Migration Bottleneck, *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 13-24 (8-11 November 1987). In *ACM Operating Systems Review 21*:5

[*submitted Sept. 12, 1988; accepted Oct. 10, 1988*]