A PERFORMANCE STUDY OF THE *UNIX*® SYSTEM V FORK SYSTEM CALL USING CASPER

Ronald E. Barkley and Curt F. Schimmel

In this paper, we describe a general-purpose timing and tracing package, called *CASPER*, for UNIX® System V. We also present the results of a study of the behavior of the UNIX System V `fork` system call using this package. CASPER is designed to support high-resolution timing and tracing of software under UNIX System V. With CASPER, we were able to obtain detailed and accurate information on the behavior of the `fork` system call, which led to minor algorithmic changes that reduced the time to execute small forks by about 15 percent.

Ronald E. Barkley *is a member of technical staff in the UNIX Systems Development Department at AT&T Bell Laboratories in Summit, New Jersey, and* ***Curt F. Schimmel*** *was a member of technical staff in that department. Mr. Barkley is responsible for measuring and analyzing the performance of new releases of UNIX System V. He joined the company in 1984 and has an M.S. in mathematics from Clemson University. While at AT&T, Mr. Schimmel worked on several releases of UNIX System V and related areas, including real-time enhancements, multiprocessor systems, virtual memory, and* `streams`*. He is now employed by Key Computer Laboratories, where he leads a project to enhance the UNIX system for a large-scale supercomputer. Mr. Schimmel has a bachelor's degree in systems*
(continued on page 109)

Introduction

There seems to be a dearth of useful tools available to kernel level designers to instrument and understand the behavior of the UNIX® system kernel. (Kernel refers to the software that provides core operating system services.) Hardware monitors are costly in terms of both dollars and setup time. UNIX System V does support kernel level profiling, which gives breakdowns of system CPU usage by kernel functions. However, the measurement process has some intrinsic limitations that detract from its usefulness.

For example, data for the UNIX system's kernel profiler is collected by periodically sampling the value of the program counter. Because the clock interrupt handler does this sampling, the profiler most likely will not have samples of any work performed by the handler. Another problem with the profiler is the rate at which it samples the program counter. This rate, which typically has a period of 10 ms (milliseconds), is too coarse to measure actual CPU usage accurately. (Panel 1 defines acronyms, terms, and UNIX system tools mentioned in this paper.) As a result, we must take a statistical approach involving many samples. (For a discussion of the intrinsic problems of sampling based profilers, see reference 1.) Finally, because of its maturity and monolithic structure, the UNIX System V kernel tends to spend little time in any single routine, which makes function-level breakdowns less useful.

In this paper, we will first describe a new general-purpose

package, called *CASPER*, that supports timing and tracing of UNIX System V software at both the user and kernel levels. We will then describe how we were able to use CASPER to gather timing data and information on other resource requirements of the `fork` system call. With this information, we were able to improve the performance of small forks by about 15 percent.

To begin, we present an overview of CASPER's design and the functions this package provides. For a more detailed discussion of its architecture and implementation, see reference 2.

Panel 1. Acronyms, Terms, and UNIX System Tools

`attach`	add a region to a process's address space
`bcopy`	copy a block
CASPER	tool for tracing UNIX System V user and kernel software
`casrecord`	generate a CASPER record
`contmemall`	allocate contiguous memory
CPU	central processing unit
DBD	disk block descriptor
`exec`	execute a new program
`exit`	terminate and leave a process
`fork`	create another UNIX system process
`getcpages`	get physically contiguous pages
`insert`	CASPER tool to instrument C programs
MMU	memory management unit
page	basic unit of physical or virtual memory management
PID	process identification (number)
`procdup`	duplicate a process
`ptalloc`	allocate a page table
`r_list`	array of pointers to page tables
region	contiguous portion of virtual address space
`setuctxt`	set user context
`shell`	command intepreter program
`u-area`	user area
`uballoc`	allocate a user area

Overview of CASPER Architecture

Historically, UNIX system kernel designers have not had adequate tools to measure and understand the behavior of the systems that they build. In the absence of such tools, kernel developers have had to fashion their own tools or forgo instrumentation and use intuition and instinct.

In fact, CASPER began as a set of specialized software designed to collect page-fault traces on UNIX System V. It has evolved into a general-purpose package that provides high-resolution timing and tracing services for software—at the user and kernel level—under UNIX System V. These services include: access to a high-resolution timer, buffering of event records, secondary storage of these time-stamped event records, and data retrieval. CASPER runs as a kernel-level daemon process. Access and control of CASPER from user level are provided through a special file, and the file-system permissions set for this special file govern who can use CASPER.

High-Resolution Timer. The standard UNIX System V clock is interrupt driven. At each clock interrupt, the value of a special clock variable is incremented and certain operating system services are performed. The clock's period is typically 10 ms or more, a resolution often too coarse to time many of the events that interest us.

Fortunately, many systems have access to a higher resolution clock that can be polled for its value. (For systems that do not have this hardware, we must rely on the standard UNIX system clock. The alternative is to build a plug-in board with such a timer on it—still less expensive than a hardware monitor.) On the AT&T 3B2 computer, where CASPER was initially implemented, this clock is the short interval timer that generates clock interrupts. CASPER examines this timer, which has a resolution of 10 μs (microseconds), to time stamp events of interest.

Event Records. An *event record* tells whether a particular event has occurred and contains a synopsis of the state of the system when that event occurred. For example, we might trace such events as entering and leaving a particular function, the arrival of a network message, or the completion of a disk task.

Panel 2. Sample Code to Record an Event of Interest

```
struct event_record {
  char       event_type;  /* the type of event that occurred */
  short      pid;         /* process id responsible for event */
  int        size;        /* size of data for event */
  };

#define   CHECKSUM   0x10

/*
 * Checksum(data, size, sum) finds the checksum for the <size> bytes of
 * the data starting at <data>. If the calculated checksum is different
 * from <sum>, return -1 to indicate an error; otherwise, return 0.
 */

checksum(data, size, sum)
  char       *data;
  int        size, sum;
{
  int        crcsum;
  struct     event_rec    rec;

  rec.event_type = CHECKSUM;     /* event is a CHECKSUM */
  rec.pid = u.u_procp->p_pid;    /* get process id from proc table */
  rec.size = size;               /* record size of checksum request */
  casrecord(&rec, sizeof(rec)); /* call CASPER to timestamp and save record */

  crcsum = crc(data, size);
  if (sum == crcsum)
          return 0;
  else
          return -1;
}
```


To support general-purpose tracing of arbitrary events, CASPER allows a developer to define the information content of event records. In the context of the C language, this simply means: The developer defines a *structure* whose fields contain a flag, which indicates the event has occurred, and other variables to record the values of various state variables at the time of the event. Examples of these state variables are: the process identification (PID), value of the program counter, or length of a job queue.

To use CASPER, a user identifies places in the source code that correspond to events of interest and includes code to generate an event record (see Panel 2). This record is then passed to the CASPER routine `casrecord`, which takes two arguments: a pointer to that record, and the size of the record. `Casrecord` then

accesses the high-resolution timer to generate a time stamp, and copies the record into a circular buffer maintained by CASPER. At some point, CASPER will arrange to have the circular buffer's contents copied to a user-specified file. Once the data is written to this file, a user can post-process it for analysis. CASPER provides high-level routines to give a user access to the data at the record level.

Function Level Profiling. Using CASPER, we can measure the elapsed time between any two events by inserting calls to `casrecord` before each event and noting the difference in time. If we choose these events as function entry and function exit, we can measure the elapsed time within a function. CASPER provides a utility program, called `insert`, that parses C programs, identifies functions and their exit points, assigns unique identifiers to them, and inserts calls to CASPER to save entry and exit information. This information can then be post-processed to give breakdowns of the amount of time spent in different functions.

The `Fork` System Call

We now present our study of fork behavior. To begin, we provide some background information on the internals of the `fork` system call and the memory management architecture of UNIX System V Release 3. For additional information about the architecture of the UNIX System V kernel, see reference 3.

Each process running on a UNIX system has a context or state that consists of the following:

- A unique PID number
- An entry in the system-wide process table
- A `u-area` (user area)
- A virtual address space.

The PID number provides a way to refer uniquely to each process in the system. The process-table entry and the `u-area` contain state information that the operating system needs to schedule and execute the process. This information includes: a register save area, a separate stack for use while executing a system call for the process, the process priority, and a pointer to the virtual-address-space information.

The system uses virtual addresses so that several programs can coexist in memory at the same time. Furthermore, programs are not limited by the amount of available physical memory. The *virtual address space* is the range of virtual addresses that the system recognizes. Each process has its own virtual address space, and the memory management subsystem maps pages in the virtual address space to actual physical resources. As a minimum, the virtual address space includes the process's *text* (the instructions that the process is executing), its *data*, and its *stack* (a last-in, first-out temporary storage area).

New processes are created under the UNIX operating system with the `fork` system call. The new process that a fork creates is referred to as the *child*, and the process that invoked the fork is called the *parent*. Although fork creates the child as an exact duplicate of the parent, the child is given a different PID number. Thus, the steps involved in executing a `fork` system call are:

1. Allocate a new process-table entry and initialize it with the contents of the parent's entry.
2. Allocate a unique PID number.
3. Allocate a new `u-area` and initialize it with the contents of the parent's `u-area`.
4. Duplicate the address space of the parent process for the child.

The memory management unit (MMU) on the AT&T 3B2 divides the 32-bit, 4-Gbyte (gigabyte) virtual-address space into four equal-sized sections (Figure 1). Sections 0 and 1 are reserved for the kernel; the remaining 2 Gbytes, in sections 2 and 3, are available for user processes. Associated with each section is a variable-sized segment table. Each entry in the segment table points to a page table, and each page-table entry points to a 2-kbyte page of physical memory. (A "k" is 1024.)

The virtual address space of each process is composed of several *regions*, each a contiguous portion of virtual address space. A region may be shared or private and may be read-only or read/write. Each process minimally has three regions, one each for text, data, and stack.

Internally, each region is represented by a kernel data structure (Figure 2). This structure contains information such as the region's size, its type (text, data, stack, etc.), the number of processes that share the region, and a pointer to an array of pointers to MMU-page tables that

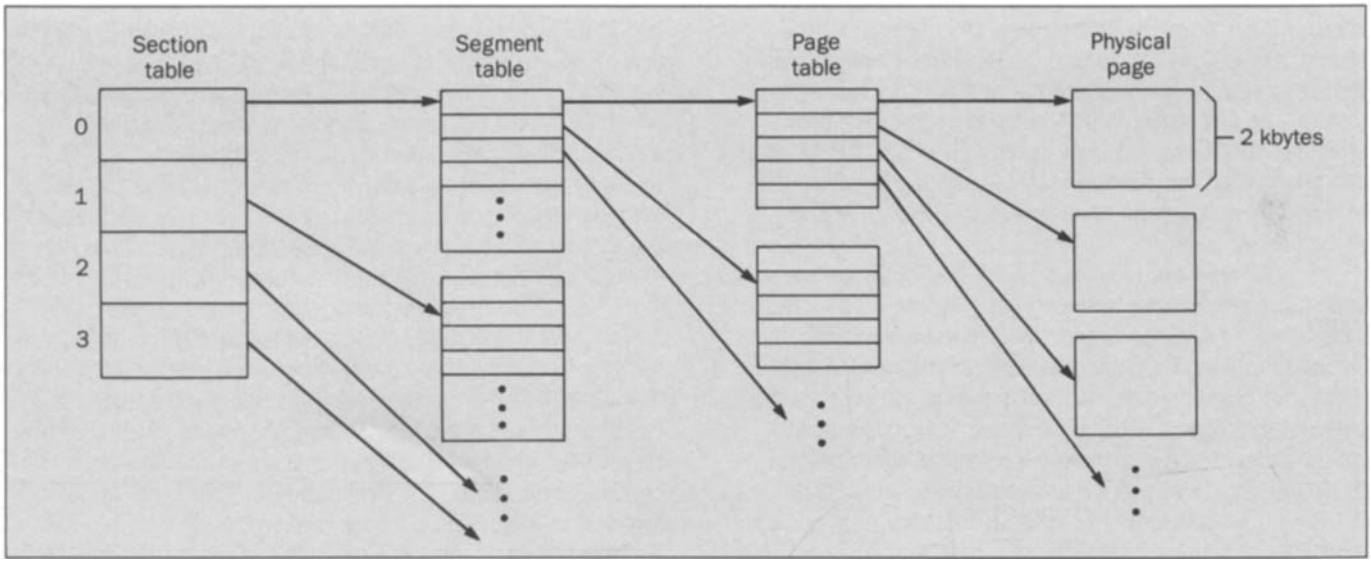
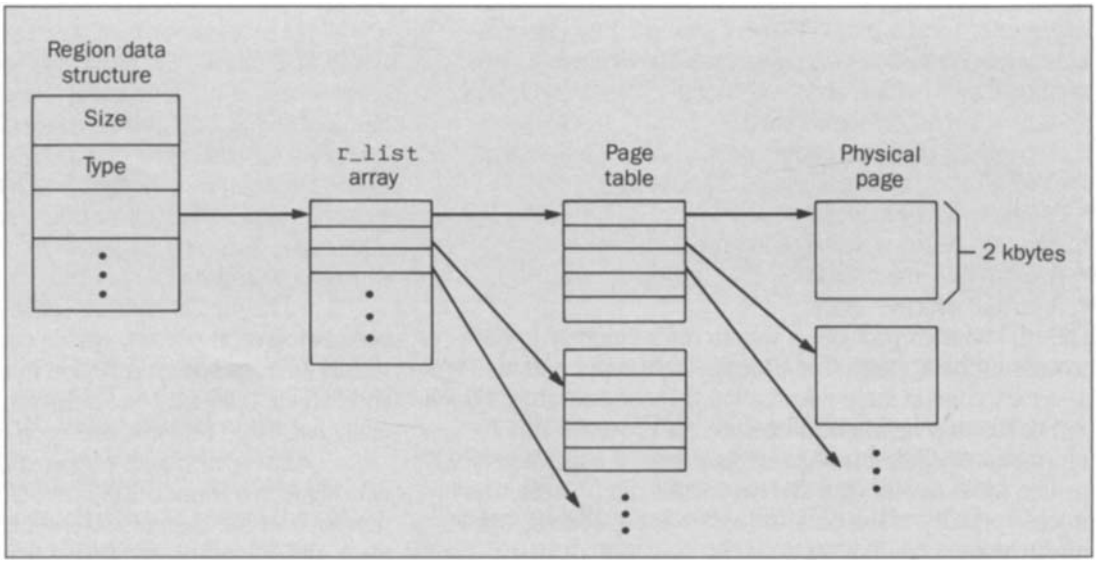


Figure 1. Table organization of the memory management unit. The section table is a 32-bit, 4-Gbyte virtual-address space. Sections 0 and 1 are reserved for the kernel; sections 2 and 3 are for user processes.

Figure 2. Organization of the region data structure. Each process has at least three regions (one each for its text, data, and stack).



represent the region's pages. This *array of pointers* is referred to as the `r_list`. On the 3B2 computer, a page table contains 64 one-word entries that each refer to a 2-kbyte page. Thus, a single page table can address 128 kbytes of memory. Each page table is followed by a second kernel table of *disk block descriptors* or DBDs. Every page-table entry has a corresponding entry in the DBD table that indicates where a copy of the page can be found on the disk, if the page is not now in memory.

System V uses the *copy-on-write* technique to duplicate the address space during a fork. To defer copying the entire address space, both the parent and the child receive a read-only copy of it. When either process attempts to write to a page in the address space, a protection fault occurs; the kernel copies that page and makes it writable for the writing process. Then, only this page needs to be copied to the other process. This approach saves the processing overhead of copying the entire address space when the child process may reference only a portion of it.

For writable regions such as the data and stack, copy-on-write is implemented by making a copy of the region data structure—including the page tables and DBDs—and assigning it to the child process. Write permissions on all pages in the region are turned off in both the parent and child processes. The actual pages that form the region are not copied during the fork.

For read-only regions such as text, the region is connected to the new child process through an `attach` operation that causes the region to be visible in the child's address space. Because the region is read-only, we do not need to make a copy of the region data structure or page tables. The region can simply be shared.

Analysis of `Fork`

We separated the actions of the `fork` system call into four major sections and instrumented them to find their individual cost. These sections are:

- Process-table search—Finds a free entry and unique PID.
- `uballoc` (user block allocator) routine—Initializes the segment tables and allocates the pages for the child's `u-area`. (User block is synonymous with `u-area`.)
- `setuctxt` (set user context) routine—Copies the parent's `u-area` and kernel stack to the child.
- Main loop in the `procdup` (process duplicator) routine—Duplicates the parent's address space and assigns it to the child.

We ran a test on an AT&T 3B2/310 computer that did 100 forks of the smallest possible process (one text page, one stack page, one data page) and averaged the results (Table I). There was no other activity on the system while the test was run. The line for "Miscellaneous" in Table I accounts for the rest of the time spent in `fork`, after we subtract the time spent in the major sections listed. Miscellaneous includes time for copying the process-table fields, duplicating file descriptors for any open files, and incrementing the reference count on the current directory.

Most of the time in `fork` is spent duplicating the address space. This time will increase as the size of the process increases, while the other sections remain constant.

Performance Improvements to `Fork`

We examined the code for the three largest sections—`uballoc`, `setuctxt`, and `procdup`—for possible algorithmic or data structure changes to improve performance.

Stack Copy. The job of `setuctxt` is to copy the parent's `u-area` and kernel stack to the child process. This function is dominated by a call to `bcopy` (block copy) to copy the entire 6-kbyte `u-area`. This block copy is unnecessary because most of the space in `u-area` is room provided for the kernel stack to grow and contains no useful information during a `fork` system call (Figure 3).

The `u-area` structure occupies only 1764 bytes, followed by a few words for the user-file descriptors, followed by the kernel stack that is only a few hundred bytes deep during a fork. So only about 2 kbytes of data really needs to be copied, a third of the total `u-area`. We implemented this improvement and reran the above test with the results in Table II.

As expected, the time spent in `setuctxt`

dropped from 3.17 ms to 1.31 ms, or to about a third of what it had been because now only a third of the data is being copied. This trivial code change improves performance about 13 percent for small forks.

Page Table Allocation. Because less time is now spent in copying the `u-area`, the relative percentage of time spent in the `procdup` code increased. Our examination of the code showed that `ptalloc` (page table allocator) is called frequently to allocate memory for new segment tables, page tables, and the `r_list` array for each region for the child process. In particular, the child needs:

- Two segment tables, one for section 2 and one for section 3
- One page table for the data region and one for the stack region
- An `r_list` array for both the data and stack regions.

Because the text is shared, no additional allocations are needed for it. This requires a total of six calls to `ptalloc` for a fork of a small process. Larger processes will require more page tables and may require more calls to allocate additional space.

We instrumented `ptalloc` to determine how much time was spent in it during a fork. The results showed 600 calls to `ptalloc` during the test program that executed 100 forks, or six calls per fork, as was expected above. Each call consumed an average of 450 μs, or 2.7 ms per fork (450 μs × 6). A fork is now about 13 ms, so about 21 percent of the time during a fork is spent in `ptalloc`. Because this is about half of the total `procdup` time, it seemed worthwhile to examine `ptalloc` further to see if we could improve it.

The code in `ptalloc` maintains a list of free page tables using a linked list and a bitmap. The bitmap makes it easier to allocate the physically contiguous page tables that are needed for segment tables, page table and DBD pairs, and `r_list` arrays. A single page table is 64 words, so there are eight per page. The linked list links together pages that contain free page tables, and the eight bits in the bitmap for each page show the locations of the free page tables. If more than eight physically contiguous page tables are needed, new physically contiguous pages are allocated from free memory.

Table I. Averaged Results for the Smallest Process

Section	Mean time (ms)	Variance	Total time (%)
Search	0.67	0.09	5
`uballoc`	3.86	0.44	26
`setuctxt`	3.27	0.05	22
`procdup` loop	5.84	0.48	40
Miscellaneous	1.07	—	7
Total fork	14.71	3.67	100

Only very large processes may need more than a few physically contiguous page tables (which would be used as segment tables). For example, a single page table for the `r_list` array can manage a region of 8 Mbytes (megabytes). Page tables and DBDs are only allocated in pairs. Segment tables typically use only three contiguous page tables. Thus, an allocation algorithm that optimizes for the most frequent cases of one, two, and three contiguous page tables might help.

However, each newly allocated page table must be cleared before it is used. The memory traffic to clear the page tables could be dominating the total `ptalloc` time, so we added an extra CASPER instrumentation point to find out how much of the `ptalloc` time was spent in clearing the page tables. This confirmed that 60 percent of the time in `ptalloc` was spent clearing memory. There is no optimization to be done here, because the page tables must be cleared to ensure that all the page table valid bits are off before they are used.

What about the remaining 40 percent of the time in `ptalloc`? An allocation algorithm that optimizes for small page table sizes might cut this time in half at best. If we save 20 percent of the time in `ptalloc`, we would save 20 percent of the total 21 percent that `fork` devotes to `ptalloc`, or only about 4 percent for the entire fork. Such a small savings did not seem to warrant the added complexity in the code, especially when there was no guarantee of improvement. So, we decided not to alter `ptalloc`.

`Getcpages`. The `ptalloc` routine calls

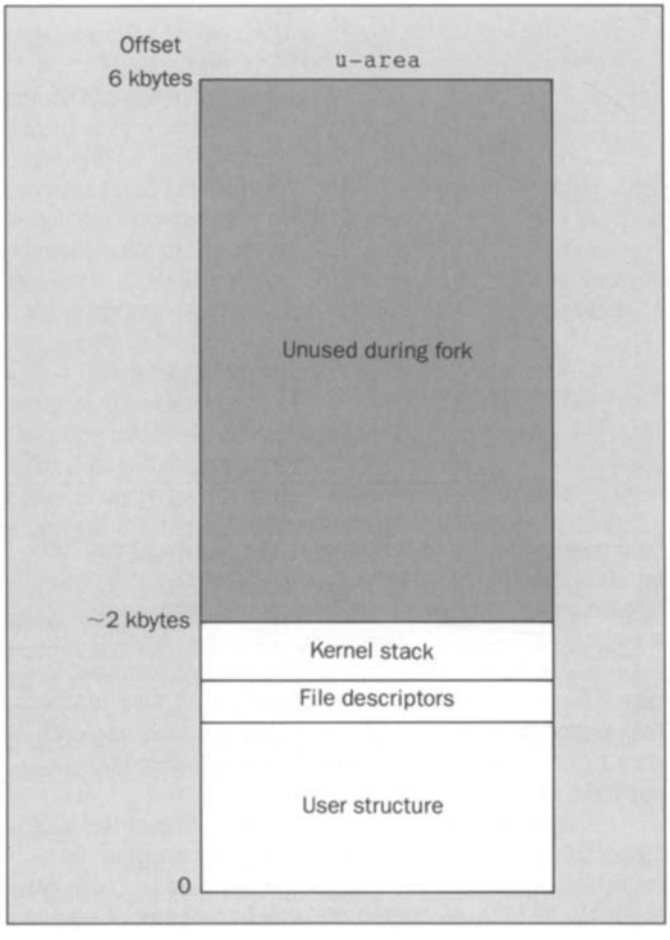


Figure 3. Layout of the user area `u-area`. On a fork, only the area in use by the parent process really needs to be copied to the child.

`getcpages` (get contiguous pages) to allocate more physically contiguous memory when it cannot get enough space from its own free list. `Getcpages` in turn calls `contmemall` (contiguous memory allocator) that does the real work of finding contiguous pages. Our examination of the common cases showed the code for `contmemall` was inefficient.

The current algorithm for `contmemall` starts its search for contiguous pages at the first page of memory after the fixed portion of memory that the kernel occupies after booting. The algorithm then sequentially examines each page of memory in the system to see if the page is free. If it is (and if `contmemall` was asked to find n contiguous pages), the algorithm then checks the next sequential n–1 pages to see if they are also free. If they are, the search ends and the memory is allocated. If not, then the sequential search continues until all memory has been examined.

Table II. Averaged Results After Stack Copy Improvement

Section	Mean time (ms)	Variance	Total time (%)
Search	0.66	0.07	5
`uballoc`	3.85	0.37	30
`setuctxt`	1.31	0.15	10
`procdup` loop	5.78	0.36	46
Miscellaneous	1.12	—	9
Total fork	12.72	2.83	100

The overwhelming majority of cases, though, request only *one* page. We can infer this as follows:

- DBDs and page tables are only allocated in pairs of 128 words.
- A one-page segment table would be enough for 32 Mbytes of address space.
- A one-page `r_list` is enough for a 64-Mbyte region.

So, it should be rare that `contmemall` ever allocates more than one page.

To confirm this, we instrumented the code to count how many calls were made to `contmemall` and how many calls asked for more than a page. The system was then used for a normal day's work, including building new kernels, editing, and so on. Of thousands of calls to `contmemall`, only two requested more than a page.

If only a single page is needed, then it can simply be removed from the free list of memory and we can avoid the linear search of main memory. For cases that require allocating more than one contiguous page, we can improve the algorithm further by noticing that a contiguous section

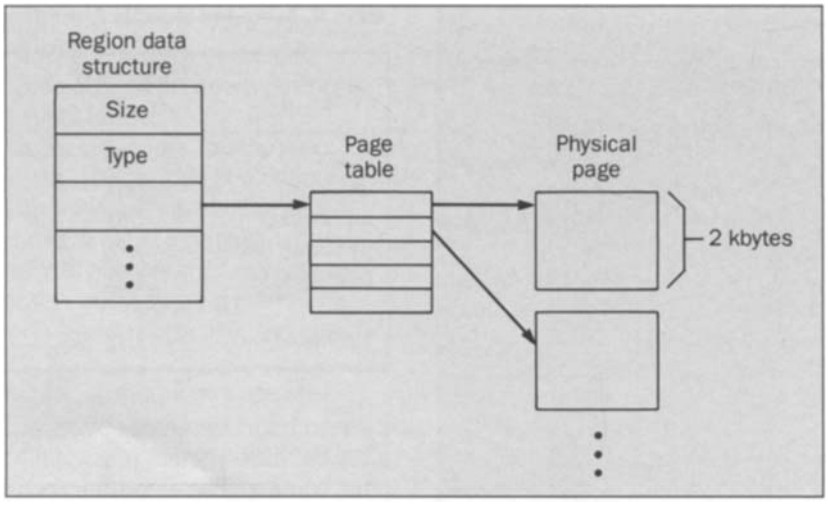


Figure 4. Region data structure with direct page-table pointer. An `r_list` array is not required when a region has only one page table.

of memory that is potentially free must start with a free page. Because a free list of memory already exists, there is no need to perform a linear search to find a free page. The algorithm can simply run down the free list and then test contiguous pages that follow each entry to see if they are also free. This saves the time of looking at memory pages that are already in use. On heavily loaded systems where memory is tight, the savings can be great because the free-list search is short compared to a linear search of nearly all main memory.

The `r_list` Array. The `r_list` array is always allocated in multiples of page tables. Even the smallest region has a full 64-word `r_list` array. While memory space is not an issue here, allocation time is.

Each entry in the array points to a page table that contains up to 64 pages of 2 kbytes each. Thus, one page table can serve regions up to 128 kbytes in size. Almost every standard UNIX system command fits into text and data regions of this size. Stacks rarely grow larger than one or two pages. Therefore, the most frequent cases only need one word in the `r_list`. This can be easily implemented by changing the region structure's `r_list` field, which currently is a pointer to an array of pointers to page tables. We can make this field a union of this pointer and a pointer to a page table. So, in the common case where there is only one page table in the region, the `r_list` field points directly to this page table (Figure 4) and thus avoids a level of indirection, the allocation of the largely unused array space, and the deallocation time during `exit`.

We expect this optimization to save two calls to `ptalloc` per fork (one to allocate the `r_list` for the data region, and one for the stack region). Because each `ptalloc` takes about 450 μs, this would save 900 μs per fork, or 6 percent.

Extending Copy-on-Write. A good portion of the time (about 20 percent) during the fork is spent copying the page-table entries to the child and turning on the copy-on-write bits in both the parent and child processes. Copy-on-write is a great savings when the child process does an `exec` soon after the fork. To save time during the fork, we can defer copying the user data until a write is attempted to it. Because the write may never occur, much time can potentially be saved.

We can extend this concept of lazy evaluation further to the region level. If we can save time by deferring the copying of actual data, why not also defer copying the page-table entries to save more time? To implement this, we can attach the regions in a read-only mode (using the segment permissions) until either the parent or child process attempts to write to the region. The faulting process now copies the region, turns on the copy-on-write bits,

and continues.

Suppose the child issues an `exec` or `exits` before the parent writes to its regions. Then, when the parent does a write and gets a fault, the region's use count will be back to one, and the region can simply be reattached in read/write mode. This eliminates the needless copy-on-write faults that occur in the parent when it faults on write-protected pages whose count is already at one. The savings are potentially large for long-lived programs, like the UNIX system `shell`, that fork frequently.

A further extension to this is to apply copy-on-write on a per-segment basis. When a region is duplicated, the page tables would be shared. Segment permissions for shared-page tables would be set to read-only to implement copy-on-write for the page tables. This technique requires a new page-table data structure that contains a use count.

If a write fault occurs in a page table that is shared, the faulting process makes a copy of the page table, turning on both the copy-on-write bits and the write permission for that segment. If the use count on the page table is one, then the segment permissions can simply be changed to read/write. This approach again has the advantage that it eliminates needless copy-on-write faults after sharing has ended.

Summary

Using the timing and tracing services of CASPER, we have been able to obtain detailed information on the resource requirements of the UNIX System V `fork` system call. The existence of a package such as CASPER has allowed us to concentrate on data analysis rather than on the data-collection process. The entire `fork` analysis described here was accomplished in a single day, which would not have been possible without CASPER.

We concentrated on those sections of code with the greatest potential gain and redesigned the algorithms so they efficiently handle the most common cases. As a result, we have demonstrated that we can improve (reduce) CPU time for small forks by about 15 percent. With another small change in the way the `r_list` array is implemented and managed, we can improve the CPU time for small forks by another 6 percent.

Acknowledgments

Danielle Bendet, Danny Chen, Gerry Gordon, Jeff Lankford, T. Paul Lee, and Hira Nirang have all contributed to the package that is now CASPER. Thanks also go to Danny Chen and Jerry Feder for their comments on this paper.

References

1. C. Ponder and R. Fateman, "Inaccuracies in Program Profilers," *Software—Practice and Experience*, Vol. 18, May 1988, pp. 459-468.
2. R. E. Barkley and D. Chen, "CASPER the Friendly Daemon," *Proceedings of 1988 Summer USENIX Conference*, USENIX Association, San Francisco, California, June 1988, pp. 251-260.
3. M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1986.

Biographies (continued)

software science from the Rochester Institute of Technology and a master's degree in computer science from Rensselaer Polytechnic Institute. The work reported was done while Mr. Schimmel was with AT&T.

(Manuscript received August 19, 1988)