Kernel Mucking in Top

William LeFebvre – Argonne National Laboratory[1]

ABSTRACT

For many years, the popular program *top* has aided system administrations in examination of process resource usage on their machines. Yet few are familiar with the techniques involved in obtaining this information. Most of what is displayed by *top* is available only in the dark recesses of kernel memory. Extracting this information requires familiarity not only with how bytes are read from the kernel, but also what data needs to be read. The wide variety of systems and variants of the Unix opeating system in today's marketplace makes writing such a program very challenging. This paper explores the tremendous diversity in kernel information across the many platforms and the solutions employed by *top* to achieve and maintain ease of portability in the presence of such divergent systems.

Motivation

Any system administrator knows the litany. A line of users start forming outside of the office, the phone starts ringing off the hook, and everyone has the same thing to say: ''this lousy computer is taking minutes to do anything, even a simple `ls` command.'' Most experienced administrators will look for the same thing: a cpu-intensive process that is tying up most of the computers cycles. Perhaps they get their information from the standard Unix *ps* command, or perhaps they will start with *uptime* or *w*. Many administrators ''in the know'' will use the freely available software package *top*. In any case, the system administrator is seeking information that only the kernel has: what is the status of the computer's resources and which processes are using them?

For all but the most modern versions of Unix this information is only attainable by reading it directly out of kernel data structures. Imagine the task of writing a program which extracts this information and designing it to maximize portability. When you consider the incredible variety of hardware platforms and Unix variants in the marketplace, the job seems almost insurmountable. Kernel designers don't often consider ease of access to information by outside processes when designing internal data structures. Consequently, even minor operating system revisions may make changes which have a major impact on kernel-dependent programs. Although not completely successful, the design employed by *top* has achieved a reasonable medium between the needs of extracting useful information and maintaining ease of portability.

A Top Process Display for Unix

The software package *top* presents a full-screen display of the top cpu-using processes on the system. It also presents some essential system information about cpu cycles (system versus user), memory usage, load averages, process categories, and other tidbits. This information is updated regularly (usually every 5 seconds). Refer to Figure 1 for a sample display from top.

The display varies depending on the particulars of the underlying operating system. The sample shown was taken from a system running Sun's Solaris 2.3. In general, the top four lines show information about the overall health of the process environment. The first line shows the 1, 5, and 15 minute load averages. The second line shows the total number of processes and how they break down in to separate categories (such as sleeping, running, and stopped). The third line shows percentages spent in each cpu state: this is the line that will show when a cpu is spending a disproportionate amount of time in the kernel. The last line shows information about memory usage.

The remainder of the display consists of information about each individual process. Again, this information will vary depending on the operating system, but in general it will show the process id, username of the owner, internal priority, *nice* setting, total virtual memory size, amount of virtual address space currently in physical memory (the ''resident set'' size), process state, cpu time, cpu usage percentages, and command name. The display is sorted by one of the cpu percentages so that the top percentage using processes are shown first.

This information is updated regularly, usually every five seconds by default. The user can set the update time to any number of seconds (including

[1]This work was not funded through Argonne National Laboratory. The submitted manuscript has been authored by a contractor of the U. S. Government under contract No. W-31-109-ENG-38. Accordingly, the U. S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U. S. Government purposes.

zero, in which case updates happen continuously). Other options are also available to regulate a variety of items.

Usually there is only one way for *top* to obtain all the information it displays. It has to dig around in the kernel for it. This is the same way that Unix commands like *ps*, *netstat*, *vmstat*, and other status-displaying utilities obtain their information. The author has affectionately coined the phrase *kernel mucking* for this procedure. Despite any sugar coating that may be available in the libraries, in the final analysis there is usually only one way to get data out of the kernel: by using *open*, *lseek*, and *read* on the device */dev/kmem*. This device is a very special character device. The kernel maps accesses to this device so that they exactly correspond with kernel memory itself. Any byte in the kernel can be read from somewhere in */dev/kmem* (and if the device is opened for writing, any byte can be written as well).

Some Unix System V release 4 systems provide an easier way to get per-process information: the pseudo file system */proc*. In fact, the current trend in Unix releases is to eliminate the need for mucking around in the kernel as much as possible. The proc file system is one step in that direction. Some SVR4 versions now also have other hooks by which programs can get non-process related information about the system, such as Sun's *kstat* device.

BSD Unix version 4.4 has the *sysctl* system call which provides a cleaner interface to kernel data. One of a number of structures can be requested. The kernel will fill in the appropriate data and return it to the requesting program. This is obviously much easier than kernel mucking, but is limited to only the information that the kernel writers saw fit to provide. If you want to access a value that is not provided by one of these structures, you have to resort to old fashioned means.

For most Unix variants, there's still only one way to get to all this information: kernel mucking. Obviously, any program which needs to do this is going to be very dependent on the specific organization of the kernel running on the machine. That means the task of writing such a program to be portable across different versions of Unix (and thus different architectures) is going to be extremly difficult. In fact, just making it portable across different minor revisions of the *same* version of Unix is difficult. One of the primary design goals for version 3 of *top* is to isolate all the machine-dependent code in one source file and to provide a clean and well defined interface as a set of functions. All the functions which handle options processing, screen management, display updates, internal commands, etc., are all part of the machine-independent portion of *top*. These functions make calls as needed on routines in the machine-dependent file library. Only the latter need concern itself with kernel mucking and other specifics of the operating system.

Version 2.7 of *top* (before all the machine dependent code was isolated) was difficult and time consuming to port: it only ran on a handful of different systems. The reorganization of version 3 made top significantly easier to port. As a consequence, ports now exist to allow top to run on these platforms:

```
last pid:  2980;  load averages:  0.08,  0.14,  0.11             11:12:21
58 processes:  56 sleeping, 1 stopped, 1 on cpu
Cpu states: 93.1% idle,  3.1% user,  3.9% kernel,  0.0% iowait,  0.0% swap
Memory: 21M real, 1424K free, 43M swap, 52M free swap

  PID USERNAME PRI NICE  SIZE   RES STATE   TIME   WCPU    CPU COMMAND
 2980 root       7    0 1692K 1412K cpu     0:01  0.77%  4.25% top
  709 lefebvre  28    0 9900K 2612K sleep  21:13  0.13%  1.93% Xsun
  741 lefebvre  28    0 3404K 1700K sleep   0:36  0.08%  0.77% cmdtool
  787 lefebvre  14    0   11M 2180K sleep   6:11  0.00%  0.00% maker4X.exe
   93 root      24    0 1772K  760K sleep   1:27  0.00%  0.00% automountd
    1 root      34    0  696K   96K sleep   1:23  0.00%  0.00% init
   70 root      34    0 2184K  600K sleep   0:39  0.03%  0.00% keyserv
   68 root      29    0 1608K  588K sleep   0:35  0.03%  0.00% rpcbind
  892 lefebvre -25    0 2376K  968K sleep   0:27  0.00%  0.00% xmh
 1213 lefebvre  14    0 5024K  324K sleep   0:14  0.00%  0.00% emacs
  716 lefebvre  24    0 1844K  684K sleep   0:11  0.00%  0.00% olwm
 2967 lefebvre  24    0 4960K 3836K sleep   0:07  0.00%  0.00% emacs
 2703 lefebvre  34    0 3388K 1332K sleep   0:04  0.00%  0.00% cmdtool
   86 root     -25    0 1560K  488K sleep   0:04  0.00%  0.00% inetd
 1255 lefebvre  34    0 3404K 1100K sleep   0:03  0.00%  0.00% cmdtool
```

Figure 1: Sample output from *top* running on Solaris 2.3

386BSD	Intel-based SVR4.2
AViiON w/DG/UX 5.4+	Mt. Xinu MORE/bsd (VAX)
BSD/386	NetBSD
Dynix 3.0.x	OS/MP 4.1A (Solbourne)
Dynix 3.2.x	SunOS 4.x
generic 4.3BSD	SunOS 5.x (Solaris 2.x)
generic 4.4BSD	Ultrix 4.2 or later
HPUX (most versions)	UMAX 4.3 (Encore)
Intel-based SVR4	UTek 4.1 (Tektronix)

Ports are in the works for the DEC Alpha machine and for IBM's AIX.

When describing detailed aspects of kernel mucking, this paper will try to remain as generic as possible, but this requires dealing in vague generalities. In some cases, specific examples are used to make a particular point. There is no guarantee that the example will work on any particular platform or Unix variant.

Accessing Kernel Information

Many programmers have never been exposed to the specific techniques involved in retrieving information from the kernel. Indeed, many programs have no such need. This section offers a brief overview on the technique.

Reading from */dev/kmem*

It may seem strange to you that you access memory as if it were a disk, but that is essentially what is done. Think of kernel address space as if it were one large file, with the zeroth byte of the kernel corresponding to the beginning of the file. If you want to read the byte at address *x*, you would use a code fragment like this one:

```
int i;
unsigned char c;
i = open("/dev/kmem");
lseek(i, x, 0);
read(i, &c, sizeof(unsigned char));
```

Reading an entire longword is done similarly: the only difference being the use of an unsigned long and the argument to *sizeof*.

Finding Kernel Addresses

So where do the addresses come from? The kernel is stored on disk as an executable in the root directory. Depending on the particular Unix variant in use, it could be named */unix*, */vmunix*, */stand/unix*, */kernel/unix*, or something similar. This is the very same image that is loaded at bootstrap time. It is always stored in the same format as a regular executable, complete with symbol table. This table contains every global variable name along with its address. For an ordinary executable or object file (the format is the same) this information is used by the linker, *ld*, to match up uses of external variables to their definitions. Many installed executables have had this information removed or stripped (usually by the command *strip*), but the kernel image is intentionally not stripped. The C library contains a function that knows how to obtain the addressing information given a list of variable names: *nlist*.

The C library function *nlist* takes an array of *struct nlist*. Each structure element has room for a variable name (called a *symbol name*), its ''value'', type, and other tidbits of information. The *nlist* function expects this array to have the name fields already filled in with the names of the variables you want. It then opens the executable of your choice, finds all the values, and fills in the rest of the structure. This value is not the variable's value, but rather the addresses in the executable where the variable is actually stored. Don't confuse these two! The documentation (and the include file) for *nlist* refer to a ''symbol'' and a ''symbol's value.'' The symbol corresponds directly to the variable's name. The ''symbol value,'' however, is the address in memory where the variable's value is found.

So let's say that we want to find the process id number of the last process that was created. Now, not all Unix variants keep track of this information, but those that do usually store it in a kernel variable called *mpid*. On most Unix systems the compiler prepends an underbar to every external variable name before adding it to the symbol table, so we really want to ask for the variable *_mpid*. Here are the steps we would take:

- open */dev/kmem*
- use *nlist* to obtain address for *_mpid*
- *lseek* to that address
- *read* the value at that location
- display the result

This is fleshed out in Figure 2. For clarity, this example excludes error checking code. In addition to checking the value returned by *open*, the value returned by *lseek* should be checked to insure that it is 0, and the value returned by *read* should be checked to see if it is equial to the number of bytes requested in the read. If either of these kernel calls does not return what is expected, then it is usually an indication that the kernel address used in the *lseek* call is not valid.

```
#include <nlist.h>
int i, mpid;
struct nlist nlst[] =
          {"_mpid"}, {0};

i = open("/dev/kmem", 0);
nlist("/vmunix", nlst);
lseek(i, nlst[0].n_value, 0);
read(i, &mpid, sizeof(mpid));
```

Figure 2: Reading a kernel variable's value

The *kvm* Library

Many modern-day versions of Unix make this task less onerous by providing a kernel-access library called *kvm*. Users of this library initially call *kvm_open* to initiate use of a given kernel image. This function, rather than returning a file descriptor, will return a pointer to a *struct kvm*, much like the streams library returns a pointer to a *FILE* structure. All other functions in the *kvm* library take a *struct kvm* pointer as an argument and will perform their machinations on the corresponding kernel image. Most *kvm* libraries provide functions to carry out the followng operations: open, close, read, write, symbol list (*nlist*), walk through the process structures, obtain process information by process id, obtain a process's user structure. Those who are doing kernel mucking are well advised to use the *kvm* library on any system where it is available. It hides some of the really grubby details.

Process Structure

Although kernel internals vary widely between different Unix variants, there are some basic concepts shared by all (or nearly all). In the earliest versions of Unix there were two kernel structures employed to keep track of all the information about a process: the process structure and the user structure [4]. This design has been carried through to BSD Unix [3] and System V Unix [2] and is present in every Unix variant seen by this author.

The process structure is also known as the proc structure (the name given to the structure is ‘‘proc,’’ as in *struct proc*). This structure is typically defined in the include file `<sys/proc.h>`. The information contained in this structure is what the kernel needs to have readily available in order to keep track

```
#include <nlist.h>
#include <sys/proc.h>

main()
{
    int i, fd, bytes, nproc;
    unsigned long proc;
    struct proc *pbase, *pp;
    static struct nlist nlst[] = {
        { "_proc" },
#define X_PROC 0
        { "_nproc" },
#define X_NPROC 1
        { 0 }
    };

    /* open kmem, call nlist, get variables' values */
    fd = open("/dev/kmem", 0);
    nlist("/vmunix", nlst);
    lseek(fd, nlst[X_PROC].n_value, 0);
    read(fd, &proc, sizeof(proc));
    lseek(fd, nlst[X_NPROC].n_value, 0);
    read(fd, &nproc, sizeof(nproc));

    /* allocate space for proc structure array */
    bytes = nproc * sizeof(struct proc);
    pbase = (struct proc *)malloc(bytes);

    /* read all the proc structures in one fell swoop */
    lseek(fd, proc, 0);
    read(fd, (caddr_t)pbase, bytes);

    /* iterate thru the result */
    for (pp = pbase, i = 0; i < nproc; pp++, i++) {
        /* display information for interesting processes*/
        if (pp->p_stat != 0) {
            printf("pid %d, uid %d\n", pp->p_pid, pp->p_uid);
        }
    }
}
```

Figure 3: Extracting and examining the entire proc array

of the process throughout its lifetime. When the process exits and when the process's parent has picked up the exit information (for example, via *wait*), the process structure is freed. Examples of information typically stored in the proc structure [3, page 73] are:

- process id, parent process id, pointers to child process structures
- real user id, effective user id
- scheduling: priority (including *nice*), recent cpu utilization, sleep time
- memory management: pointers to page tables and shared program text
- process size (text, data, and bss)
- some signal information

Sound familiar? It should. Just about every piece of process information displayed by *ps* or *top* is stored in the process structure.

The process structures are usually stored in a large array that is allocated at boot time. The size of this array will dictate the maximum number of processes that can be running on the system at any one time. The array elements are also sometimes referred to as process slots. When a process exits, its slot is marked as available. When a process is created, the kernel will hunt down a free slot to use for the new process.

The process array is stored in the variable *proc*. The number of elements (slots) in the array is stored in the variable *nproc*. On a system that has no *kvm* library, you have to use *read* directly to find a specific process or to iterate through the process slots. The *kvm* library will typically include several functions that find and return data in the proc array, making such access significantly easier (albeit rather inefficient). These functions would likely be named *kvm_nextproc*, *kvm_getproc*, and *kvm_setproc*.

As an example of reading arrays and structures from the kernel, Figure 3 contains a complete program that reads and iterates through the *proc* array. This program reads the entire array in at once, then steps through it. This is the method that *top* usually uses to read the *proc* array and to read any other large array, as it is far more efficient than performing one read per array element.

User Structure

The user structure contains all the stuff that the kernel needs when a process is running (or more specifically, when the process is swapped in). It is defined in the include file `<sys/user.h>`. Unlike the proc structure, this structure is not actually stored in fixed kernel memory. Instead it resides in the process's virtual address space. When the process is swapped out to disk, this structure goes with it. The user structure typically contains information [3, page 77] such as:

- execution states (register values and processor status structures)
- open files (file descriptors)
- creation mask (the umask)
- current directory inode
- resources usage information (*struct rusage*) and limits (*struct rlimit*)
- executable ''command'' name

For most mucking problems, this structure would not be needed, except for the fact that it is the only place where you can get the name of the executable currently running in this process (i.e., the command name). Complicating the matter is the fact that this structure is extremly difficult to find. Since it is stored in the process's address space, it is not readily available to someone who is only mucking around in */dev/kmem*. All that you find there are pointers to the virtual memory page information, which you can then use to track down the physical page addresses, then open another special file, */dev/mem*, and muck around in it for the user structure.

That is, of course, assuming that the process is actually in memory. If it has been swapped out, then an entirely different method must be used to hunt down the swapped out pages in a completely different device: */dev/drum*. The *kvm* library makes this trivial by providing the function *kvm_getu*. This function takes a pointer to a proc strcuture and does whatever is necessary to retrieve the user structure, passing it back to the caller. Having this function available is especially helpful since the code to obtain the user structure is particularly sensitive to different virtual memory management techniques, and can vary widely between different Unix vendors.

Platform Independent Design

In an attempt to isolate as much of the machine dependencies as possible, all of the kernel mucking in *top* is contained in a single collection of functions all residing within one file. Ideally, this is the only file that needs to change when compiling a version of *top* for a different platform. This file, along with some ancillary documentation, forms a machine *module*. At configuration time, a module name is chosen that is appropriate to the platform. All these modules are collected together in one directory. It is intersting to note that these modules comprise about 80% of the total source code for *top*.

The Challenge

The crux of the the design is the collection of functions used to obtain the information and their exact definition. The author sincerely wishes that there already existed an OS-independent definition for such a library, but the truth of the matter is that systems vary too much to make such a definition workable. Even if such a definition existed its adoption by just the major vendors would take years. As development of this interface definition progressed, it became clear that the information needs varied too much between systems and much of the decisions about which statistics to display had to be left to the modules themselves.

An excellent example of this dilemma is the memory status line. In older versions of Unix (BSD 4.2 and SunOS 3), this line displayed: amount of real memory in use, amount of virtual memory allocated, amount of real memory still free. The first two figures were supplemented with the amount of memory used recently (or ‘‘active’’). These exactly corresponded to fields in the *vmtotal* structure named *total*. But different virtual memory implementations maintain different types of statistics. In fact, SunOS version 4.0 still used the *struct vmtotal* to track some of the virtual memory statistics, but did not fill in the ‘‘active’’ portions of the structure. The original layout of the line appeared as:

```
Memory: 2408K (2560K) real,
    6700K (3202K) virtual, 992K free
```

Using such a layout for SunOS 4.0 was not appropriate, since there was no number available which would make sense when placed inside the parentheses. The current SunOS 4 port displays: available (real) memory, in use, free, locked. Clearly, the machine module needs some control over the labelling of the information. Therefore, not only does it need to pass back the data itself, but also strings describing the data.

The Design

Some of the decisions made about the module interface were dictated by earlier design decisions pertaining to output disply handling. The display engine in *top* does not use *curses* [1] or anything similar to it. As early as version 2, it was decided that *top* could do a better job of optimizing the number of characters output to the screen than any sort of screen or window management software: *top* compares the numerical data before converting it to ASCII and displaying it on the screen. When the module interface was developed for *top* version 3, the raw numbers were still needed so that the display interface could work pretty much the same way.

But when it came to the process lines, the original version 2 design decision was that comparing the individual numbers did not yield enough of a benefit. For those, the text line was formatted first, then a character-by-character comparison was carried out between the new and old lines, and overstrikes and cursor movement used as necessary to update the screen using the fewest characters possible. This display handler design still exists in version 3. Consequently, the module interface requires that information about an individual process be returned as a preformatted string of text.

The Function Definitions

Putting it all together, a machine module is expected to have the following functions:

`machine_init(struct statics *statics)`
Carries out any necessary machine-specific initialization. This includes calling *kvm_init* or similar operations, retrieving values from the kernel that are not expected to change (such as *nproc* and the pointer to the *proc* table), allocating any permanent arrays (such as an array to hold the proc table), and doing any other calculations for values that will not change over time. This function also fills in a *struct statics* array with static information: currently arrays of string labels for the first few lines of the display. The structure is documented in Figure 5.

`char *format_header(char *uname_field)`
Returns the header line for the process display area. The argument, *uname_field*, is used as the label for the username/uid column. A command line

```
struct statics
{
    char **procstate_names;         process state names
    char **cpustate_names;          cpu state names
    char **memory_names;            memory statistics names
};

struct system_info
{
    int    last_pid;                last process id issued
    double load_avg[NUM_AVERAGES];  load averages
    int    p_total;                 total number of processes
    int    p_active;                number of processes active (displayable)
    int    *procstates;             array of process states data
    int    *cpustates;              array of cpu states data
    int    *memory;                 array of memory statistics data
};
```

Figure 4: Structures filled in by machine module functions

argument allows the user to choose between usernames and user ids in the display. The machine-independent portion of *top* processes this and decides how the column should be labeled (either `USERNAME` or `UID`) and passes this as the argument to this function. The function embeds it in an appropriate place in the line that is returned. This function is necessary because the machine module has complete control over the formatting of the individual process status lines. Therefore, it must also have control over the column headings.

`get_system_info(struct system_info *si)`
Fills in a *system_info* structure with current information about the status of the system. This is called once per display iteration. The structure is documented in Figure 4.

```
caddr_t get_process_info(
    struct system_info *si,
    struct process_select *sel,
    int (*compare)() )
```
Retrieves current process information, paying attention only to those processes which meet the selection criteria in *sel*. The information is then sorted by *qsort* (3) using *compare* as the comparison function. It returns an arbitrary value used as a handle for *format_next_process*.

```
char *format_next_process(
    caddr_t handle,
    char *(*get_userid)())
```
Format and return a string that describes the next process in the sorted list. The first argument is the handle returned by *get_process_info*. The second argument is a function that, given a uid returns either a username or a uid (used for formatting the username column).

`int proc_compare(caddr_t p1, caddr_t p2)`
A *qsort* comparison function suitable for use as the *compare* argument for *get_process_info*. It was originally intended that different comparison functions would be made available by the machine module to provide for sorting on different columns of the output. Since the machine module is the only part of *top* that knows how to look in the proc structure and that knows how to format a process status line, it would be necessary for the module to provide such comparison functions. Although this flexibility exists in the design, it has not yet been exploited.

`int proc_owner(int pid)`
Returns the uid of the owner of the process *pid*. This is used in the machine-independent part of top to validate the use of the internal *kill* and *renice* commands.

```
int setpriority(int dummy,
    int who,
    int niceval)
```
On those systems which do not have a *setpriority* (2) system call, this function needs to be provided in the machine module. It is intended to be compatible in a limited mannter with the BSD *setpriority* system call. In *top*, the first argument will always be *PRIO_PROCESS* and can safely be ignored, the second argument is the process id, and the third is the new desired priority. The machine independent part of *top* guarantees that this function will never get called unless the user who ran *top* is either the superuser or the owner of the process whose pid is the second argument. *However*, the machine independent part of *top* has no easy way to see if the new priority is less than the old priority. It is up to this function to perform that security check (just as

```
main()
{
    process_options;
    machine_init(&statics);
    display_init(&statics);
    initialize_signals_and_miscellany;
    while (more_to_display)
    {
        get_system_info(&system_info);
        processes = get_process_info(&system_info, &ps, proc_compare);
        display_load_averages;
        display_time;
        display_procstates;
        display_cpustates;
        display_memory;
        for (i = 0; i < system_info.p_active; i++)
            display_process(format_next_process(processes, get_userid));
    }
}
```

Figure 5: Sketch of main algorithm for *top*

the real *setpriority* call in BSD makes such a check, so must this one). In general, only modules for System V Unix variants will need to supply this function.

The functions defined above enable the machine-independent side of *top* to have a clean structure unemcumbered by details about differences in machine specifics. The overall algorithm is given in Figure 5. This hides many of the details, but highlights the use of the machine module functions.

Labelling Information

The function *machine_init* fills in a structure with ''static'' information: data that will not change during execution. This data consists solely of text strings suitable for labelling information returned by *get_system_info*. The *statics* structure contains (in the current design) three pointers, each pointing to a NULL-terminated array of strings. In the *system_info* structure are three corresponding pointers to integer arrays: *procstates*, *cpustates*, *memory*. For each of these array pairs (i.e.: *procstate_names* and *procstates*), the machine-independent code will display the label after the number. Refer to Figure 4 for the actual structure definitions. As part of its output optimization, the display engine knows that it only needs to write the string labels to the screen once. On subsequent updates it only changes the numerical data, and then only if the data has actually changed.[2] Both process states and memory statistics are handled the same way. If a string in the *_names* array of the *statics* structure is zero length, then the corresponding element in the array of numbers (from the *system_info* structure) is skipped. If the statistic is zero, then neither the label nor the number are displayed. The display engine also takes care of pluralization and trailing commas. The statistics for cpu states are handled differently. The numbers in the cpustates array are assumed to be in tenths of a percentage point. For example, the integer 105 is displayed as `10.5`. Each of these numbers is displayed (with a trailing percent sign) even if it is zero, and they are always formatted to take up 5 columns.

Important Kernel Data

Although each port deals with differences in kernel structure, methods, and variable names, there are some pieces of information which are commonly essential across most of the platforms.

proc The beginning of the proc array (an array of *struct proc*).

nproc The size of the proc array.

avenrun The array of load averages. There are almost always three elements: one minute average, 5 minute average, 15 minute average. These are the same numbers shown as ''load average'' by *uptime* (1). Some systems store this as a float or double while others store it as an integer with an implied binal point.

mpid The process id assigned to the last process. Not all platforms assign process id's sequentially: those that don't do not have such a variable.

cp_time An array of counters indicating time spent in each of several different cpu states, typically idle, user mode, niced user mode, kernel (or system) mode. At every clock interrupt one of these counters (depending on the current kernel state) is incremented [3, page 51]. By comparing the counters over time, *top* can calculate percentage time spent in each kernel state. Some System V kernels keep this information in the structure *sysinfo*. The size and significance of the individual members of the array vary as well.

ncpus On multiprocessor systems, the number of cpu's installed.

Another important set of information is the statistics on memory usage. But the widely different methods of memory management means that the kernel data structures involved are hardly ever the same between two different platforms. As a case in point, consider the difference between BSD 4.3 for the VAX and SunOS 4. Both were derived from BSD 4.2. On the VAX, all memory usage statistics are collected in *struct vmtotal* as described earlier (see section ''The Challenge''). This information can be easily retrieved from the kernel using technquies previously described. Earlier versions of SunOS 4 made a half-hearted attempt to maintain the data in this structure, but the memory management techniques employed under SunOS 4 did not lend themselves well to the gathering of such statistics. As a consequence, the only way to collect meaningful information about physical memory usage under SunOS 4 is by walking the array of page descriptors (*struct page*) in the kernel. This is similar to walking the array of process structures and can be done in one of two ways: either a call to *read* inside a loop (one read per structure) or one large read for the entire array followed by a loop iterating through the returned data. Either way, this is obviously much more involved than retrieving just one structure.

Conclusions

It is a tremendous challenge to write a program which is so heavily dependent on Unix internals in a way that it is still easy to port. There are many aspects of the machine module design which the author is still not particularly fond of. This design

[2] This isn't strictly true. If the number of digits required changes (i.e., the number to display changes from `9` to `10`) then the remainder of the line needs to be rewritten since the text labels no longer appear in the same columns.

works quite well, and those who have engaged in ports to other platforms indicate that it is reasonably easy to use. Some improvements could be made, but in general the design seems to work quite well.

Over the years, *top* itself has become a very popular tool for system administration. The information it provides is invaluable in troubleshooting process-related problems on the system. Several Unix vendors now provide ports of *top* as part of their standard operating system distribution. By any measure *top* can be deemed a success.

New versions of *top* are routinely distributed through the normal free software distribution channels: the newsgroup *comp.sources.unix* and all the sites which archive postings to that group. The latest release can always be found via anonymous FTP at the site *eecs.nwu.edu* in the directory */pub/top*. The most recent beta test version is usually placed in the same directory. Despite the author's recent change in job status, he hopes that Northwestern University will continue to provide storage for the package on its anonymous FTP server.

Acknowledgments

The author would like to thank all the people who have provided suggestions and additional code for *top*, and he would especially like to thank those who took the time to write additional machine modules as well as those who beta tested new versions. Thanks are also extended to Rice University, Northwestern University, and now Argonne National Laboratory for providing facilities and motivation.

Author Information

William LeFebvre is a Computer Systems Engineer in the Decision and Information Systems Division of Argonne National Laboratory. He received a Bachelor of Arts degree (with a major in Computer Science) in 1983 and a Master of Science degree in 1987, both from Rice University in Houston, Texas. William can be reached at Argonne National Laboratory, 9700 South Cass Avenue, DIS/900/MS-12, Argonne IL 60439-4812. His electronic mail address is: lefebvre@dis.anl.gov .

Bibliography

[1] Arnold, Kenneth, ''Screen Updating and Cursor Movement Optimization: A Library Package,'' *UNIX Programmer's Supplementary Documents*, **(PS1)**, 4.3 Berkeley Software Distribution, April, 1986.

[2] Bach, Maurice, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.

[3] Leffler, Samuel, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman, *4.3BSD UNIX Operating System*, Addison-Wesley Publishing, Reading, Massachusetts, 1989.

[4] Thompson, K., ''UNIX Implementation,'' *The Bell System Technical Journal*, **57** (6), July 1978.