UNICOS Kernel Internals Application Development

Nicholas P. Cardo, Sterling Software, Inc., Numerical Aerodynamic Simulation Facility , NASA Ames Research Center, M/S 258-6, Moffett Field, CA 94035-1000

ABSTRACT: *Having an understanding of UNICOS Kernel Internals is valuable information. However, having the knowledge is only half the value. The second half comes with knowing how to use this information and apply it to the development of tools. The kernel contains vasts amounts of useful information that can be utilized. This paper discusses the intricacies of developing utilities that utilize kernel information. In addition, algorithms, logic, and code will be discussed for accessing kernel information. Code segments will be provided that demonstrate how to locate and read kernel structures. Types of applications that can utilize kernel information will also be discussed.*

The UNICOS kernel is large, complicated and tracks a tremendous amount of information. Automated system monitoring tools can be developed to utilize kernel information for early detection of problems and make notifications or take appropriate actions.

The kernel is defined as *"The central controlling program that provides basic system facilities. The UNIX kernel creates and manages processes, provides functions to access the filesystem, and supplies communication facilities."*[1] Many people have a fear about accessing the kernel directly due to its complexities, when in its simplest definition, the kernel is just a large program.

1 Approaches

There are a couple of methods to use for extracting information from the kernel.

- System Calls.
- Read Tables.
- Direct kernel access.

1.1 System Calls

System calls can be used to obtain a large amount of useful information. In particular, information about the executing process or system configuration information can easily be obtained with system calls. Using system calls for this type of information is actually more efficient than reading the kernel and parsing the information. Some useful system calls are:

```
sysconf(2)   getjtab(2)   pathconf(2)
getpid(2)    limit(2)
```

The kernel maintains pointers which speed access to relevant information about a process or job. Additionally, the system call interface is the simplest method of obtaining kernel information. However, using system calls provides only a limited amount of information.

1.2 Reading Tables

Two system calls, `tabinfo()` and `tabread()`, are designed to read kernel tables with a simple and easy to use interface. The only drawback to using these is that they can only access the table specified. Many table structures contain pointers to other kernel structures, which cannot be traversed using this method. The list of tables that can be accessed through this method are provided in `<sys/table.h>`.

This method involves two steps. The first step is to retrieve information about the table. This is accomplished with the `tabinfo()` system call. The second step is to actually read the table with `tabread()`. For better performance, the entire table should be read with one read operation when possible.

For example, suppose an application needs to scan the process table to produce a report or to search for a process. This can be done with the following:

```
tabinfo(PROCTAB,&tbl_info);
tbl_size = tbl_info.head + (tbl_info.ent
        *tbl_info.len);
tbl = (char *)malloc(tbl_size);
tabread(PROCTAB,tbl,tbl_size,0);
```

The `tabinfo()` call retrieves information about the process table, such as number of entries, length of each entry and the address of the table. Not only is the information used for `tabread()`, but it is useful for creating a buffer to hold the table. The total size of a buffer to hold the table is calculated by multiplying the number of table entries by the size of an entry, then adding in the size of the table header. The `tabread()` call will then read the entire process table into the buffer. At this point any processing can be done on the copy of the table in the buffer. If the process table was read one entry at a time, it is highly likely that it will have changed by the time processing is complete.

1.3 Direct Kernel Access

In UNIX, everything can be considered a file, and accessed with normal file I/O routines. The kernel is no different and can be accessed directly through the device inode for the kernel, */dev/kmem*. Since this permits access to the running system, caution should be exercised when accessing it. Not only can the running system be corrupted, but system performance can be impacted. Following a few rules when accessing */dev/kmem* can keep from having major system problems or performance degradation.

- Open the kernel readonly.
- Perform as few I/O operations as possible.
- Check return codes and pointers.

The first step to opening the kernel is to verify access to it. This can be done with the `access()` system call. The following code verifies read access to `/dev/kmem`.

```
if(access("/dev/kmem",R_OK)) {
 fprintf(stderr,"%s\n",
    sys_errlist[errno]);
 exit(1);
}
```

Provided access is permitted, the next step is to actually open `/dev/kmem`. This can be done with the following code.

```
if((fd=open("/dev/kmem",O_RDONLY)) == -1) {
 fprintf(stderr,"%s\n",
    sys_errlist[errno]);
 exit(1);
}
```

A function can be written to perform both operations and return either a -1 for a failure or the file descriptor for a successful open.

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>

int opn_kernel()
{
int     fd;

 /*
  *  Check for read access on /dev/kmem
  */
 if(access("/dev/kmem",R_OK))
    return(-1);

 /*
  *  Open the kernel readonly
  */
 fd = open("/dev/kmem",O_RDONLY);
 if(fd == -1)
    return(-1);

 return(fd);
}
```

Both `access()` and `open()` will set `errno` to indicate the appropriate reason for the failure. Opening the kernel can now be simplified as:

```
if((fd=opn_kernel()) == -1) {
 fprintf(stderr,"%s\n",
        sys_errlist[errno]);
 exit(1);
}
```

At this point, the integer file descriptor can be used to seek to various locations within kernel memory to retrieve tables.

2 Locating Symbols

It may be necessary at times to locate symbols within the kernel to extract their values. System tables can also be located by their symbol name. Locating symbols in the kernel is accomplished by utilizing `nlist()`.

Symbol tables are a part of any `a.out` structured file and are stored in the global name table (GNT) of the `a.out` file. The kernel is an executable similar to `a.out` and contains symbols in the GNT. To accurately identify all the symbols, a program would be necessary to display all the symbols in the GNT for the kernel. The command `/bin/nm` can be used to display all the valid symbols for the kernel. For example:

```
$ nm /unicos
```

will display all symbols and their byte addresses for the kernel.

However if only a few symbols are needed, the symbol names can be extracted from header files. Contained within header files is a section which identifies kernel specific declarations. This section of the header file is identifiable by `#ifdef KERNEL`. External declarations within this section are actually identifying symbol names.

For example, suppose there is interest in the `var` table. The first step is to look at the header file `<sys/var.h>` and locate the kernel specific declarations.

```
#ifdef  KERNEL
extern  struct  var     v;
#endif
```

This identifies the symbol `v` as the `var` table in the kernel. The `/etc/crash` command can be used with it's `nm` command to display symbols and their word addresses. To verify that `v` is a valid symbol:

```
# /etc/crash
> nm v
v              06373435 common memory
```

Using `/bin/nm` yields:

```
$/bin/nm /unicos
...
```

```
  63734350 T v
  ...
```

which indicates byte address 63734350 octal where `crash` displays word address 06373435 octal.

Once the symbol name is known, it is possible to convert that symbol to its address. The `nlist()` function will fill in an `nlist` structure which contains the address of the symbol. The address returned in the `nlist` structure is the byte address of the symbol whereas the length of the space pointed to by the symbol is returned in words. The system macro `wtob()` can be used to convert words to bytes. The following code returns `0` upon success and a
`-1` upon error.

```
#include <stdio.h>
#include <sys/types.h>
#include <nlist.h>
#include <errno.h>
#include <sys/param.h>
#include <sys/sysmacros.h>

static  struct nlist nl[] = {
 { ““ },
 { ““ },
};

/*
 *  Return the address of an entry from
 *  the symbol table
 */
int getsym(symname,sz,ad)
char*symname;
int*sz;
int*ad;
{
intret;

 nl[0].n_name = symname;

 /*
 *  Locate the symbol in the kernel
 */
 if((ret=nlist(“/unicos”,nl)) == -1)
    return(-1);

 /*
 *  Did it return an address
 */
 if(nl[0].n_value == 0) {
    errno = EFAULT;
    return(-1);
 }

 *sz = wtob(nl[0].gse_blen);
 *ad = nl[0].n_value;

 return(0);
}
```

For example, to retrieve the address of the `var` table with this function would be accomplished as follows:

```
if((ret=getsym(“v”,&sz,&addr)) == -1) {
 fprintf(stderr,”%s\n”,
    sys_errlist[errno]);
 exit(1);
}
```

3 Reading /dev/kmem

Two pieces of information are needed to read kernel information; location and size. After opening the kernel, the next step is to go to the location desired and retrieve the information.

Positioning at a symbol for reading can be accomplished with `lseek()`. Note that for both `lseek()` and `nlist()`, the address is specified in bytes, so no conversion is necessary. If the address is a word address, use `wtob()` to convert the word address to a byte address.

Once positioned at the proper address, the information can be retrieved using `read()`.

The following code shows how lseek() and read() can be utilized to retrieve information from the kernel.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

/*
 *  Seek to an address and read
 */
int kread(buf,addr,sz,fd)
char *buf;
word addr;
word sz;
int  fd;
{
int  ret;

 /*
  *  Seek to the address
  */
 if((ret=lseek(fd,addr,SEEK_SET)) == -1)
    return(-1);

 /*
  *  Read sz bytes from addr
  */
 if((ret=read(fd,buf,sz)) != sz) {
    errno = EIO;
    return(-1);
 }

 return(0);
}
```

For example, to read the `var` table based on the information returned by `getsym()` would be accomplished with the following:

```
if((ret=kread(buf,addr,sz,fd)) == -1) {
   fprintf(stderr,"%s\n",
      sys_errlist[errno]);
   close(fd);
   exit(1);
  }
```

The function `kread()` can be utilized to retrieve entire tables, a single entry within a table or even just 1 word from kernel memory.

4 Putting the Pieces Together

The individual components for accessing kernel information have each been discussed separately. Applications can be developed utilizing these basic components.

For example: suppose that there is a need to extract the number of process slots from the running kernel. This count is stored in the `v_proc` field of the `var` table. So the application needs to read the `var` table from the kernel and display the field `v_proc`.

4.1 Reading Tables

This method utilizes `tabread()` and `tabinfo()` to read the `var` table. Reading is accomplished in two steps: obtain necessary information and actual reading. `tabinfo()` is used to obtain information about the `var` table including its address and size. This information is utilized by `tabread()` to actually read the table into a buffer for processing.

```
  #include <stdio.h>
  #include <stdlib.h>
  #include <sys/types.h>
  #include <sys/table.h>
  #include <sys/var.h>

  main()
  {
  structtbs tbl_info;
  char *tbl;
  long tbl_size;
  structvar *v;

   /*
    * Obtain information about the
    * var table
    */
   if(tabinfo(V,&tbl_info) == -1) {
      perror("tabinfo");
      exit(1);
   }

   /*
    * Calculate the size of the table
    * and allocate a buffer for the table
    */
   tbl_size = tbl_info.head +
             (tbl_info.ent*tbl_info.len);
   if((tbl=(char *)malloc(tbl_size))
      == NULL) {
      perror("malloc");
      exit(1);
   }
   /*
    * Read the table
    */
   if(tabread(V,tbl,tbl_size,0) == -1) {
      perror("tabread");
      free(tbl);
      exit(1);
   }

   /*
    * Print the desired field
    */
   v = (struct var *)tbl;
   printf("Process slot count = %d\n",
      v->v_proc);

   free(tbl);
   exit(0);
  }
```

4.2 Reading /dev/kmem

This method will use the routines previously discussed for opening and reading kernel memory. There are three steps to acommplishing this: get the address of the table, open the kernel and last to seek and read the table into a buffer.

```
  #include <stdio.h>
  #include <stdlib.h>
  #include <errno.h>
  #include <sys/types.h>
  #include <sys/var.h>

  main()
  {
  int  fd;
  int  addr;
  int  ret;
  int  sz;
  char *tbl;
  structvar *v;

   /*
    * Get the address and size of the
    * var table whose symbol is "v"
    */
   if((ret=getsym("v",&sz,&addr)) == -1) {
      fprintf(stderr,"%s\n",
         sys_errlist[errno]);
      exit(1);
   }

   /*
    * Allocate a buffer large enough
    * to hold the var table
    */
   tbl = malloc(sz);
   v = (struct var *)tbl;

   /*
```

```
     * Open the kernel for reading
     */
    if((fd=opn_kernel()) == -1) {
       fprintf(stderr,”%s\n”,
          sys_errlist[errno]);
       free(tbl);
       exit(1);
    }

    /*
     * Read the table based on information
     * obtained from getsym()
     */
    if((ret=kread(tbl,addr,sz,fd)) == -1) {
       fprintf(stderr,”%s\n”,
          sys_errlist[errno]);
       free(tbl);
       exit(1);
    }

    printf(“v_proc = %d\n”,v->v_proc);
    free(tbl);
    exit(0);
  }
```

5 Developing Applications

Most applications that utilize kernel internal information only require a small amount of information to make the application complete. There are very few applications, usually diagnostic or system monitoring, that are completely based upon kernel internal information.

To help illustrate the principles of developing successful applications, a utility will be developed. The task is to develop an application that can determine the current load on the system at that instance and that it is not necessary to take samples and average. Each step of the design and development process will be discussed to give an overall picture of the process.

5.1 Step #1: The Need

The first step to developing applications that access kernel information is to clearly identify what is needed. Unnecessary seeks and reads in the kernel can impact performance.

Once the target information is clearly identified, the next part is to determine if reading the kernel is necessary. Large amounts of information are available through other means. For example, suppose it was necessary to determine the configured maximum number of users for the running kernel. This information is contained in the field `v_users` in the `var` table. However, rather than reading the entire table for this one field, there is a much simpler method to obtain it, `sysconf(_SC_MAX_NUSERS)`. So it is possible to get some information out of the kernel without any complicated coding. By using `sysconf()`, not only is the code much simpler, but also easier to support and maintain.

Fully understanding the information needed is the key to developing well written and easily maintained programs. Also to be considered is the scope and quantity of information. It is very easy to turn a small application into a memory hog. Another important consideration is performance. Avoid large numbers of I/O operations when possible when accessing elements within a single table. Let’s suppose that an application needs to gather statistics from all the running processes in the system. There are two options; access the process slots 1 at a time or read the entire process table into a buffer then step through the entries. The better solution is to read the entire process table in 1 read operation. Not only for performance reasons but also for accuracy. Just because an application needs to see the information, doesn’t stop the kernel from updating its tables. Obtaining all the information in one read operation will minimize any discrepancies in the data.

5.1.1 Justifying The Need

A starting point would be the systems load average.

“A measure of the CPU load on the system. The load average in 4.3BSD is defined as an average of the number of processes ready to run or waiting for disk I/O to complete, as sampled over the previous 1 minute interval of system operation.”[2]

Since there is no requirement for sampling or averaging over a time delta, all that is needed is an instantaneous reading.

Applying this to UNICOS, the application needs to count how many processes exist on the the run queue not connected to a processor. In other words, how many processes have a state of `SRUN`. The process state is stored in the `p_stat` field of the `proc` structure. This information can be obtained by examining `<sys/proc.h>`. So, the application needs to read the process table and count all processes with `p_stat` equal to `SRUN`.

5.2 Step #2: The Commitment

Once the need has been determined, a desire to program and maintain the application is needed. The next major release of UNICOS can bring changes to kernel structures that affect locally developed applications. However, if the application is well written, there would be minimal to no application updates necessary. Most of the information an application would use is typically stored in a way that the need for changes are rare. Taking the necessary steps during development to ensure a long lived application can save time and eliminate unnecessary turmoil during system upgrades. Also, with the exception of architecture specific information, the kernel accesses are portable across UNICOS systems. A commitment to quality should always be a part of the development process. Poorly written code will only bring problems down the road.

5.2.1 Applying the Commitment

“*System programmers never turn down a challenge, afterall, it’s only software*.”[3] And thus the commitment to the product is established.

5.3 Step #3: The Method

It’s now time to take the plunge. As previously discussed, there are three methods to access kernel information. It is necessary to select the method for extracting information, keeping in mind the best solution might be a combination of the three methods. All methods have advantages and disadvantages.

5.3.1 System Calls

Although using system calls has the simplest interface, they are limited to what that can access. Typically they can be used effectively to obtain information about the executing process or the system itself.

In the case of the example program, this information is not directly obtainable.

5.3.2 tabinfo()/tabread()

Here simplicity is the main advantage. All the work of locating the table and accessing it have been integrated into these two functions.

However, the disadvantage is lack of versatility and control. The use of this method is limited to accessing only the tables identified in `<sys/table.h>`. Another limiting factor is that there is no movement permitted within the kernel. This method is unable to traverse a pointer in a table to another structure. Most useful information is typically contained within the major tables and traversing pointers is usually not necessary.

The example program requires that all process table entries be read. Having this simple requirement affords the ability to use a less complicated method of obtaining the information. The example program fits the profile perfectly of an application to best utilize this method.

5.3.3 /dev/kmem

By adding a little complexity, great versatility is achieved. By opening and reading `/dev/kmem`, there are no limitations to accessing the data. Pointers are easily followed and tables are easily located. The down side to this is that to achieve this capability, additional code is necessary to open the kernel, locate tables in the kernel and read the kernel.

Complexity is not part of the example program. Therefore, the added advantages of using this method are not needed. This would not be the best method to choose for the example program.

5.3.4 Which to Choose

It can be a difficult decision to choose the proper access method. Generally speaking, if the amount of information needed from the kernel is small and contained in a major table, then choose the `tabinfo()`/`tabread()` method. However, if vasts amount of information are needed from various kernel tables or if pointers need to be traversed, then choose the `/dev/kmem` method. When accessing large amounts of information from various tables, some performance improvements, such as caching, can be built into the `/dev/kmem` method.

Keeping applications simple and easy to maintain only makes for a better application. In the case of the example program, there is no complexity to the application. The information needed is clear and can easily be retrieved. Accessing `/dev/kmem` directly for this application would not be the best choice. And since the information is not readily available from a system call, the only choice is to utilize `tabinfo()` and `tabread()`.

5.4 Step #4: The Integration

Building the kernel access routines into an application does not have to be an epic event. Here are a few tips on making this a simple and easy to do task.

- Build the application modular. This way the kernel access components can “plug-n-play”.
- Use a stub program to develop the kernel access routines. This way application development won’t have to stop and wait while the kernel routines are debugged.
- Use the “In-n-Out” approach to accessing the kernel. In other words, go in, get the information then get out.

As with any development effort, avoid last minute changes. Accessing kernel information should be well planned out. Short cuts taken are potential system problems.

5.4.1 Integrating the Components

Now that the requirements and method of choice have been layed out, putting it all together is the next step. Putting the pieces together to build the application yields the following program:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/table.h>
#include <sys/aoutdata.h>
#include <sys/cred.h>
#include <sys/proc.h>
#include <errno.h>

main()
{
struct tbs tbl_info;
char  *tbl;
long  tbl_size;
struct proc    *p;
int   x,srun=0;

  /*
   * Read the process table information
   */
  if(tabinfo(PROCTAB,&tbl_info) == -1) {
     perror(“tabinfo”);
     exit(1);
  }

  /*
   *  Allocate a buffer
   */
  tbl_size = tbl_info.head +
     (tbl_info.ent * tbl_info.len);
  if((tbl=(char *)malloc(tbl_size))
     == NULL) {
     perror(“malloc”);
```

```
      exit(1);
   }

   /*
    *  Read the entire process table
    */
   if(tabread(PROCTAB,tbl,tbl_size,0)
      == -1) {
      perror("tabread");
      free(tbl);
      exit(1);
   }

   /*
    *  Loop through the buffer looking
    *  for SRUN processes
    */
   p = (struct proc *)tbl;
   for(x=0;x<tbl_info.ent;x++) {
      if(p->p_stat == SRUN)
         srun++;
      p++;
   }
   free(tbl);
   printf("The current load is %d\n",srun);
   exit(0);
  }
```

5.4.2 Contrasting Approach

To illustrate the extra steps required for reading kernel information from `/dev/kmem` directly, the example program has been rewritten utilizing that approach.

```
  #include <stdio.h>
  #include <stdlib.h>
  #include <errno.h>
  #include <sys/types.h>
  #include <sys/param.h>
  #include <sys/aoutdata.h>
  #include <sys/sema.h>
  #include <sys/cred.h>
  #include <sys/proc.h>

  main()
  {
  int     fd;
  int     addr;
  int     ret;
  int     sz;
  int     entries;
  int     srun = 0;
  int     x;
  struct  proc *pptr;
  char    *buf;

   /*
    *  Locate the proc table
    */
   if((ret=getsym("proc",&sz,&addr))== -1){
      fprintf(stderr,"%s\n",
         sys_errlist[errno]);
      exit(1);
   }

   /*
    *  Open the kernel
    */
   if((fd=opn_kernel()) == -1) {
      fprintf(stderr,"%s\n",
         sys_errlist[errno]);
      exit(1);
   }

   /*
    *  Allocate a buffer for the proc table
    */
   buf = malloc(sz);

   /*
   *  Read the proc table
   */
   if((ret=kread(buf,addr,sz,fd)) == -1) {
      fprintf(stderr,"%s\n",
         sys_errlist[errno]);
      free(buf);
      exit(1);
   }

   /*
    *  Important to close /dev/kmem
    */
   close(fd);

   /*
    *  Loop through the buffer looking for
    *  SRUN processes
    */
   entries = sz / sizeof(struct proc);
   pptr = (struct proc *)buf;
   for(x=0;x<entries;x++) {
      if(pptr->p_stat == SRUN)
         srun++;
      pptr++;
   }

   free(buf);
   printf("The current load is %d\n",srun);
   exit(0);
  }
```

Taking into consideration the coding required for accessing and reading the kernel, this method is larger and more complicated to maintain. Not only does this code need to be maintained, but also the code for `getsym`, `opn_kernel` and `kread`.

5.5 *Step #5: The Product*

Completing steps 1-4 will result in step five, the final product. Exercising caution during design and development will lead to a stable and safe final product.

5.5.1 Applying The Product

The final product has been produced. The application will count successfully and safely the number of processes on the run queue not connected to a processor for a single instance in time. The application is layed out well and will be easy to maintain during its service.

6 Can't Get There From Here

Kernel memory has a complex structured organization with multiple entry points. If it is stored in kernel memory, there's a way to get to it.

The starting point for getting to any piece of kernel information would be a kernel table. From here, following pointers to other structures will eventually lead to the piece of information desired.

The possiblility exists that only a single entry from an associated table is needed. Typically when an entry in a kernel table is associated with an entry in another kernel table, a pointer in the structure is usually supplied. This eliminates the necessity to read a second table and search for the appropriate entry.

6.1 Example

An example application of traversing kernel pointers could the necessity to access the session table entry for a particular process. Once the process is located, the session table entry for that process needs to be located.

Each process table entry contains a pointer to its appropriate session table entry. This pointer is actually contained within the `pcomm` structure in the process table entry. The address of the session table entry can be found in the field `p->p_pcomm.pc_sess`.

The following program demonstrates accessing the session table entry for a particular process. In this example, the goal is to display the total cpu time used by the session associated with a specified process.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/aoutdata.h>
#include <sys/sema.h>
#include <sys/cred.h>
#include <sys/proc.h>
#include <sys/session.h>
#include <sys/sysmacros.h>
#include <sys/machd.h>

main(argc,argv)
intargc;
char**argv;
{
int fd;
int addr;
int ret;
int sz;
int entries;
int         srun = 0;
int         x;
struct      proc *pptr;
char        *buf;
int         found = 0;
struct      sess se;

 /*
  * Was the target process specified
  */
 if(argc == 1) {
    fprintf(stderr,
       "process name required\n");
    exit(1);
 }

 /*
  *  Locate the proc table
  */
 if((ret=getsym("proc",&sz,&addr))== -1){
    fprintf(stderr,"%s\n",
       sys_errlist[errno]);
    exit(1);
 }

 /*
  *  Open the kernel
  */
 if((fd=opn_kernel()) == -1) {
    fprintf(stderr,"%s\n",
       sys_errlist[errno]);
    exit(1);
 }

 /*
  *  Allocate a buffer for the proc table
  */
 buf = malloc(sz);

 /*
  *  Read the proc table
  */
 if((ret=kread(buf,addr,sz,fd)) == -1) {
    fprintf(stderr,"%s\n",
       sys_errlist[errno]);
    free(buf);
    exit(1);
 }

 /*
  *  Look for the process
  */
 entries = sz / sizeof(struct proc);
 pptr = (struct proc *)buf;
 for(x=0;x<entries;x++) {
    if(!strcmp(pptr->p_comm,argv[1])) {
       found++;
       break;
    }
```

```
        pptr++;
    }

    /*
     *  Make sure process was found
     */
    if(!found) {
        close(fd);
        free(buf);
        fprintf(stderr,
            ”process %s was not found\n”,
            argv[1]);
        exit(1);
    }

    /*
     *  Read the session table entry
     */
    addr = wtob(pptr->p_pcomm.pc_sess);
    if((ret=kread(&se,addr,sizeof(se),fd))
        == -1){
        fprintf(stderr,”%s\n”,
            sys_errlist[errno]);
        free(buf);
        exit(1);
    }

    /*
     *  Important to close /dev/kmem
     */
    close(fd);

    /*
     *  Print out the results
     */
    printf(“Process %s (%d) is in “
        “session %d which used %d seconds\n”,
        argv[1],pptr->p_pid,se.s_sid,
        ((se.s_ucputime +
        se.s_scputime)/HZ));

    /*
     *  Release allocated memory
     */
    free(buf);
    exit(0);
  }
```

7 Application Types

There are two distinct classes of applications requiring kernel information; user and system.

7.1 User

User applications can be broken down into two subclasses; user applications and system level user applications. The difference being a users program versus a system program for users use.

7.1.1 User Application

The chances that an appliction of this type would need to access the kernel is very rare. Typically any information required can be obtained through system calls. An example of the type of information that might be needed would be information about resources which can be obtained from the session table. In this case using getjtab() would be sufficient.

7.1.2 System Level User Application

These types of applications are more likely to have a need to access kernel information. However, in most cases, the information can be obtained from system calls. An example of this type of application could be an anonymous ftp server that won’t permit usage if the systems load gets too high.

7.2 System

The majority of applications utilizing kernel information would fall into this category. Typical applications would be for system monitoring or performing diagnostics.

An example of this application at NAS would be a utility called `top`. It provides detailed information about the current processes in the system. Figure 1 shows a display from `top` that refreshes at specified time intervals. As can be seen, it provides very detailed information about the existing processes in the system. It is also capable of displaying session information.

8 Where To Go From Here

Applications are only limited to what the mind can think of. All information about the system can be obtained and utilized to create system monitoring and debugging tools. Some useful applications might be:

- process/session monitoring utility
- reset disk error counts without rebooting
- monitor for down, wdwn or sdwn tape drives
- monitor disk errors
- monitor multitasking jobs

9 Summary

This paper has shown safe and easy to maintain methods of extracting information from the kernel. Additionally, it has shown how to utilize this information into application development.

The main goal of this paper was to de-mystify the process of accessing the kernel. Detailed examples have been used to illustrate the ease of writing such applications. Once the application has been thought of, the rest is only software.

United States sites are welcome to visit NAS on the web at URL `http://www.nas.nasa.gov`. Contained within it is a software archive full of useful applications. The author can be reached at `cardo@nas.nasa.gov` for additional information.

10 References

[1] Samuel J. Leffler, Marshall Kirk McKusik, Michael J. Karels, John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, p 428, 1989.

[2] Samuel J. Leffler, Marshall Kirk McKusik, Michael J. Karels, John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, p 429, 1989.

[3] Nicholas P. Cardo.

[4] Samuel J. Leffler, Marshall Kirk McKusik, Michael J. Karels, John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.

[5] Maurice J. Bach. *The Design of the UNIX Operatng System*, Prentice-Hall, 1986.

[6] *UNICOS C Library Reference Manual*, SR-2080 8.0, Cray Research Inc.

[7] *UNICOS System Calls Reference Manual*, SR-2012 8.0, Cray Research Inc.

```
Swap (MW): 2.678 in, 0.000 out 226 users vn                14:23:16 PDT
1018.0 MW Mem: 45.3 Free 972.7 Used 87.4 Swap          Up 7+17:07:16 p 1
796 procs: 726 sleep, 48 run, 4 zomb, 2 susp, 4 big, 598 swap, 44 low-p
16 cpus: 0.0% idle 88.0% user 12.0% sys

PID   USERNAME TT/Q  PRI NICE  SIZE STATE   CPU  LIMIT   %CPU MT COMMAND
89627 username batch 563  -2 151070 runF   4899  5500  50.18%    command
78300 username batch 868  -2 148178 run    6644 14400  49.42%    command
40890 username batch 452  -2  98910 runB  13996 19000  77.61%    command
90942 username batch 776  -2  90328 run    4776 14400  71.95%    command
20450 username batch 569  -2  63212 runE    614  5990  47.88%    command
33311 username dfr   997  19  47470 run    5972  7200   0.00%    command
54138 username dfr   997  19  40280 run    4239  7200   0.00%    command
95528 username batch 765  -2  38852 runH   3736  7200  81.98%    command
86070 username dfr   997  19  35824 run    1786  7200   0.00%  4 command
86073 username dfr   997  19  35824 run    1473  7200   0.00%  4 command
86071 username dfr   997  19  35824 run    1473  7200   0.00%  4 command
86072 username dfr   997  19  35824 run    1464  7200   0.00%  4 command
54332 username dfr   997  19  32258 run    4893  7200   0.00%    command
80995 username dfr   997  19  28726 run     612  7000   0.00%    command
97580 username batch 556  -2  26724 runP   3805 28800  44.51%  4 command
97581 username batch  39  -2  26724 slp       0 28800   0.00%  4 command
97582 username batch  39  -2  26724 slp       0 28800   0.00%  4 command
97583 username batch  39  -2  26724 slp       0 28800   0.00%  4 command
19086 username dfr   997  19  25498 run    2320  7200   0.00%    command
20409 username batch 859  -2  23642 run     736 14400  65.42%    command
35260 username dfr   997  19  23054 run    6063  7200   0.00%  4 command
35261 username dfr    39  19  23054 slp       0  7200   0.00%  4 command
35262 username dfr    39  19  23054 slp       0  7200   0.00%  4 command
35263 username dfr    39  19  23054 slp       0  7200   0.00%  4 command
85205 username dfr   997  19  20418 run    3126  7196   0.00%    command
47735 username batch 530  -2  15710 runK   9189 14000  44.24%    command
```

Figure 1: `top` **display.**