Your call will be answered by one of our dispatchers. Information obtained will then be used to open up a service call on your behalf and will be forwarded to the appropriate engineer. The engineer will then respond within the standard 1 hour call back time. Email response is much the same, however an email will be returned to you by the Dispatcher letting you know that the call has been logged and giving you the Call # for your information.

OPCOM Registration

Lisa Thompson / Deborah Hummel

How is support determined and how do I get there from here?

Customers who qualify for support services is determined by Solaris 2.x Migration Managers.Your SUN Sales Representative will correspond with management staff and forward the necessary information required for registration.This registration is then sent to OPCOM for verification purposes when customers call for support.

When there are revisions, deletions or additions to be made to registrations which are currently in the data base you must contact the dispatchers at OPCOM or send email with your changes to:

solaris2-register@sun.com

Please keep in mind this includes stand-ins for vacation coverage.

Future SunOpsis Articles

Porting NIT to DLPI
Driver Debugging
NIS+ Programming Interface
User Level Threads - an example
POSIX compliance
Performance Tuning
MT-safe Xview
and still more Hints & Tips

SunOpsis is published monthly by
The Solaris 2.x Migration Support Center

To subscribe or contribute articles please contact:

Solaris 2.x Migration Support Center
100 Renfrew Drive,
Markham, Ontario,Canada
L3R 9R6

1-800-363-6200 or email us at
solaris2@Sun.Com

Publisher: Dean Kemp
Editors: Brian A. Onn / Gil Hauer
Design & Layout: Lisa Thompson / Deborah Hummel
Contributing Writers:
Enrico Giuditti
David Guerra
Gil Hauer
Deborah Hummel
Dean Kemp
Richard Marejka
Georg Nikodym
Brian A. Onn
G. Roderick Singleton
Lisa Thompson
Larry Tsui
Srinath Venkat
Ron Winacott

```
                    /* is now being deleted */
```

The common pitfall is allocating (with **malloc(3c)**) storage for `swt_ent[]` but neglecting storage for `ste_path`. Don't fall into this trap. You must allocate space for each `ste_path` in all entries of `swt_ent`. The following code fragment illustrates a correct way to do this:

```
swaptbl_t *swtbl;
char *mypath;
struct swapent *myswapent;
int i, n_swap;

.  .  .

swtbl = (swaptbl_t *)malloc(sizeof(int) +
n_swap * sizeof(struct swapent));
swtbl->swt_n = n_swap;
mypath = (char *)malloc(n_swap * MAXPATHLEN);
myswapent = swtbl->swt_ent;
for (i=0; i<n_swap; i++, myswapent++) {
    myswapent->ste_path = mypath;
    mypath += MAXPATHLEN;
}
```

Problems with some HSFS CDROMs

Brian.Onn@Canada.Sun.Com

With some vendor's HSFS (High Sierra File System) CDROMs, a directory listing may show files that can not be found. This is especially evident with a long listing (ls -l) on the mounted CDROM.

This is caused by non-conformance of the vendor's CDROM format to the ISO-9660 Standard, which specifies that all filenames on an HSFS CDROM should be in upper-case only. No mixed or lower-case file names are permitted.

All versions of SunOS after 4.1.1 map upper-case characters to lower-case when returning directory listings of HSFS directories, and map lower-case to upper-case when doing directory searches. Thus the mixed or lower-case filenames that are stored in the CDROM directory are not found in a subsequent search, producing the "file not found" errors from **ls(1)**.

Unfortunately, there is no real fix for this since the vendor's CDROM is non-conforming and should be using the Rock Ridge extensions if they truly desire mixed case filenames. Your vendor should be contacted for a replacement CDROM, or a tape version of their software.

As a stopgap solution, a version of the hsfs loadable kernel module that does not do filename case conversion has been created and is available from the OPCOM FTP server (see page 13) Note that this module should only be used to read these non-conforming disks, and should not be used for all HSFS disks. The fact that it does not do filename mapping makes the driver itself non-conforming. It is also an unsupported solution, provided only to assist you in getting over a hurdle.

Early access MT Hints

Ron.Winacott@Canada.Sun.Com

1) When compiling MT code, you must `#define MTSAFE`, before any `#include` lines, either in the source or on the compiler command line as `-DMTSAFE`. Note that `MTSAFE` will change to `_REENTRANT` in future releases.

2) There is a bug in `/usr/include/synch.h`. You must #include `<sys/time.h>` after #include `<lock-types.h>` in this file.

The OPCOM FTP Server

The OPCOM FTP server is available 24 hours a day, and contains programs that have been ported to Solaris 2.x, documentation, selected white papers, RFC's, etc. Many of the example programs presented in the text of this newsletter can be downloaded from our FTP server.

The server is located internally to Sun Microsystems of Canada, protected behind Sun's Internet firewall, and is mirrored to our Internet host **opcom.sun.ca**. If you are not directly connected to the Internet, or otherwise cannot FTP to our server, you can still access the files we provide. We have set up an automated email based file server for this purpose. To find out how you can get files from this server, send email to **ftp@opcom.canada.sun.com** (or ...!uunet!sun!suncan!opcom!ftp), with the subject **help**.

If you are internal to Sun Microsystems, Inc., you can access the server via FTP to opcom.canada. If you are outside Sun, please use the server at opcom.sun.ca. Note that the opcom.sun.ca link is noticeably slow.

Calling Solaris 2.x Support Dispatch

Lisa Thompson / Deborah Hummel

We can be reached at **1-800-363-6200** or email:

solaris2@sun.com - for Solaris 2.x issues
threads@sun.com - for User Level Threads issues

SOLARIS 2.x HINTS AND TIPS

The entries in this section are a series of interesting items discovered by users of Solaris 2.x.

To submit an entry, send email to opcom@sun.com with an appropriate subject line.

BT and Complex Command LInes

Georg.Nikodym@Canada.Sun.Com

When using the BT scheduling class, it's sometimes desirable to run complex commands.

If you simply run:

```
bt command1 | command2  file
```

then command2 (as well as the writing to file) will run in timeshare.

To run the entire command in BT, change it to:

```
echo "command1 | command2  file" | bt sh
```

Hanging NIS Clients

Brian.Onn@Canada.Sun.Com

There is a potential for Solaris 1.x NIS clients of a Solaris 2.x NIS+ server (running in YP compatibility mode) to hang. This is caused by timing variances between the Solaris 1.x client code and the 2.x server code. This incompatibility will not be fixed on the older client code (the newer server code is correct), but there is a simple work-around. The problem lies in the startup code `/etc/rc.local` on the Solaris 1.x client. In that file, you will find a line that has

```
ifconfig -a netmask + broadcast + > /dev/null.
```

During boot, this is the first command that uses the network, and at this point the machine is not bound to any server. The client may hang trying to bind to the server while trying to find its netmask.

The work-around is to add a line

```
sleep 5; ypwhich > /dev/null 2>&1; sleep 5
```

just below the if statement that starts ypbind. This ensures that ypbind is bound to a server before the first client request is received.

DES Key Incompatibility

Richard.Marejka@Canada.Sun.Com

The DES keys used in Solaris 1.x are not compatible with the those used in Solaris 2.x. The Solaris 1.x keys are commonly kept in the `/etc/publickey` file (and NIS map), while the Solaris 2.x keys are kept in the credentials table. This difference means that the source of the keys must be the same for the client and the server. For example, consider:

a) A 4.x server with NIS that exports a filesystem with secure NFS (that is, using secure RPC), and

b) A Solaris 2.x client of this server using NIS+ with DES level security.

In this case the client cannot mount the exported filesystem since the client will send it's Solaris 2.x/NIS+ credentials with the mount request and the server will attempt to verify using the Solaris 1.x/publickey credentials

A Common Pitfall For swapctl(2)

Larry.Tsui@Canada.Sun.Com

Recently there have been questions on the proper use of the swapctl(2) system call. Some users of swapctl(2) have been seeing it return simply EFAULT (bad address), which is not what they wanted.

The swapctl(2) system call is used as follows:

```
swapctl(SC_LIST, (void *)swtbl);
```

According to the man page, *swtbl* is a pointer to a `swaptable` structure containing:

```
int swt_n;       /* number of swapents following */
struct swapent swt_ent[];/* array of swt_n swapents */
```

The swapent structure contains:

```
char *ste_path;    /* name of the swap file */
off_t ste_start;   /* starting block for swapping */
off_t ste_length;  /* length of swap area */
long ste_pages;    /* number of pages for swapping */
long ste_free;     /* number of ste_pages free */
long ste_flags;    /* ST_INDEL bit set if swap file */
```

```
    local.sin_family = AF_INET;
    local.sin_port = htons(UDP_PORT);
    local.sin_addr.s_addr = htonl(INADDR_ANY);

    /* bind the socket */
    if (bind(sockfd, (struct sockaddr *) & local,
                                sizeof local) < 0) {
        char s[100];
        sprintf(s, "bind: errno = %d", errno);
        perror(s);
        exit(1);
    }

    /* read and print what we receive */
    while ((retcode = recvfrom(sockfd, buf, sizeof buf,
        0, (struct sockaddr *) & remote, &fromlen)) > 0) {
        printf("%s: read from %s:\n%s\n",
             argv[0],
             inet_ntoa(remote.sin_addr),
             buf);
        fflush(stdout);
    }

    if (retcode < 0) {
        perror("read");
        exit(1);
    }

    close(sockfd);
    exit(0);
}


/* UDP talker - client program for UDP listener */
/* (c) 1992 SMCC, a division of SMI */

#ifdef __STDC__
#include <stddef.h>
#endif

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#ifdef MULTICAST
#define MCAST_ADDR "224.9.9.2"
#define INTERFACE "le0"
#endif

#define UDP_PORT 2112

char *text[] = {
    "My other car is a SPARC!\n",
    "SPARC on board!\n",
    "Solaris 2.x!\n",
    "Operation Commitment\n"
};

int lines = 4;
/*
* Usage: talker hostname
* where hostname is the name of the host running listener
*/
#ifdef __STDC__
void main(int argc, char *argv[])
#else
main(argc, argv)
    int argc;
    char *argv[];
#endif
{

    int sockfd;             /* socket to "plug" into
                                the socket */
    struct sockaddr_in sock; /* socket structure for
                                client */
    struct hostent *host;  /* remote host data */
    extern int lines;
    extern char *text[];
    char buf[BUFSIZ];
    int i;
    int flags;

    /* make a UDP socket */
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror(argv[0]);
        exit(1);
    }

    /* assign the values */
    sock.sin_family = AF_INET; /* Address Family -
                                  Internet */
    sock.sin_port = htons(UDP_PORT);

#ifdef MULTICAST
    /* convert dotted decimal to network format. */
    sock.sin_addr.s_addr = inet_addr(MCAST_ADDR);
#else
    if ((host = gethostbyname(argv[1]))
                  == (struct hostent *) NULL) {
        fprintf(stderr, "unknown host: %s\n", argv[1]);
        exit(1);
    }
    (void) memcpy((char *) &sock.sin_addr,
            (char *) host->h_addr, host->h_length);
#endif

    printf("sending ... \n");

    flags = 0;

    /* send each line in a separate datagram */
    for (i = 0; i < lines; i++) {
        sleep(1);
        if (sendto(sockfd, text[i], strlen(text[i]) + 1, flags,
            (struct sockaddr *) & sock, sizeof sock) < 0) {
            perror("sendto");
            exit(1);
        }
    }

    close(sockfd);
    exit(0);
}
```

There are two commands that can be used to look at network statistics, or watch packets enter on the ethernet interface. The **netstat(1m)** command is used to view multicast groups 'netstat -g' or you can view statistics using 'netstat -s'. To view live packets as they arrive use the command 'snoop multicast'. Refer to the **snoop(1m)** manual pages for other options.

IP multicasting has several advantages over network broadcasting. The main advantage is the ability to address a group of hosts by a single multicast address. Another advantage is the reduction in packet examination that a given host must perform to assert whether the packet is destined for the host or should be discarded. These two features alone should encourage users to investigate the possibilities of implementing IP Multicasting on your network.

and is generally the same as ICMP. IGMP does several tasks, however its main job is to keep hosts and gateways informed on the status and configuration of multicast groups. This is accomplished by querying hosts and waiting for a response to be sent by the one of the hosts in the group. An IP datagram is the mechanism used to transport the IGMP message between systems.

To join a multicast group on Solaris 2.x a **setsockopt(3N)** must be used with the appropriate options. eg.

```
setsockopt (sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                           (char *)&mreq, sizeof mreq);
```

where

sockfd	File descriptor of the socket
IPPROTO_IP	Protocol number of the associated level
IP_ADD_MEMBERSHIP	Add membership to the group
(char *)&mreq	IP multicast address and interface addrr
sizeof mreq	size of the structure

See the manual pages **ip(7)** for other appropriate options including IP_DROP_MEMBERSHIP.

Below is an example of a client/server program that implements multicasting. The server program(listener) is User Datagram Protocol(UDP) based which listens on the socket for packets destined for the multicast address. The structure `mreq` has two member `in_addr` structures both being cast to a `u_long` type. Talker is also UDP based program and merely sends out a line of text to the intended receiver(listener). The talker program is not a member of the multicast group showing that a client not be a member of the group to send packets to it.

To compile these programs enter the following commands:

```
% cc -DMULTICAST -o listener listener.c -lsocket -lnsl
% cc -DMULTICAST -o talker talker.c -lsocket -lnsl
```

Execute the listener first, then run talker.

```
/* UDP listener - server program for UDP talker */
/* (c) 1992 SMCC, a division of SMI */

 #ifdef __STDC__
 #include <stddef.h>
 #endif
 #include <stdio.h>
 #include <sys/types.h>
 #include <sys/socket.h>
 #include <sys/errno.h>
#include <netinet/in.h>
#include <fcntl.h>

#ifdef MULTICAST
#include <net/if.h>
#include <sys/sockio.h>

#define MCAST_ADDR "224.9.9.2"
#define INTERFACE "le0"

#endif

#define UDP_PORT 2112

/*
* Usage: listen
*/
#ifdef __STDC__
void main(int argc, char *argv[])
#else
main(argc, argv)
   int argc;
   char *argv[];
#endif
{
   extern int errno;

   int sockfd;        /* fd for the socket */
   struct sockaddr_in local;  /* local socket
                              structure */
   struct sockaddr_in remote; /* remote socket
                              structure */
   int fromlen = sizeof(struct sockaddr_in);
   int retcode;    /* return code */
   char buf[BUFSIZ];

#ifdef MULTICAST
   struct ifreq ifr;  /* interface structure */
   struct ip_mreq mreq; /* multicast request */
   struct sockaddr_in *sa; /* internet specific
                              sockaddr */
#endif

   /* open a UDP socket */
   if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
       perror("socket");
       exit(1);
   }

#ifdef MULTICAST
   /* copy the interface name */
   strcpy(ifr.ifr_name, INTERFACE);

   /* Retrieve interface address of socket */
   ioctl(sockfd, SIOCGIFADDR, &ifr);

   /* convert dotted decimal format to suitable
    * internet format
    */
   mreq.imr_multiaddr.s_addr
                   = inet_addr(MCAST_ADDR);

   /* assign interface address */
   sa = (struct sockaddr_in *) & ifr.ifr_addr;
   mreq.imr_interface.s_addr = sa->sin_addr.s_addr;

   /* join the multicast group */
   if (setsockopt(sockfd, IPPROTO_IP,
       IP_ADD_MEMBERSHIP, (char *) &mreq,
                                 sizeof mreq) < 0) {
       perror("setsockopt");
       exit(1);
   }

#endif
   /* register the socket */
```

on each process running on the system at this time. This **ioctl(2)** call will return a `prpsinfo_t` structure from which the **rpc.btd** can determine if the process is running in BT mode, and its current priority.

If a process is found running at the requested priority and true batch mode is on, then the request is moved to a queue for that priority slot by the **rpc.btd**. Each second, a timer is triggered in the **rpc.btd** and the same `/proc` check is done until the requested slot is free.

If the queued process gets scheduled to run, the **rpc.btd** will fork and a **priocntl(2)** system call is used in the child to put it into the BT scheduling class. This action sets the priority of the process correctly. At the same time, the time quantum value is set to **NOCHANGE** so that the dispatch table values are used. **Exec(2)** is then called which overlays the child **rpc.btd**, thus inheriting the current (BT) scheduling class

The **bt_tool** continues to monitor the **rpc.btd** for the status of any running and queued processes that are under its control. Any output from the running processes is piped back to the **bt_tool** to be displayed in a log window. If the process is queued to run later, the output is sent to a log file for the **bt_tool** to look at when the process finally gets scheduled. When the log window is released from the **bt_tool**, the log file is removed from the disk.

All of the *batch intelligence* is in the **rpc.btd** and **not** in the BT class. **Bt_tool** maintains a list of running processes and logs of all the processes that have run. If the tool is exited before any queued processes have run the **rpc.btd** will schedule them and the output will go to a log file. This file will be mailed to the user and removed from the disk if the **bt_tool** does not request the log after one hour of its creation.

THE BT COMMAND

The **bt** command also uses the **rpc.btd** to control batch processes. When the user wishes to start a process in batch mode using the **bt** command, the request is sent to the **rpc.btd** and the process is either started or queued to start later when the requested slot is free (depending on the command line arguments.)

The output from a non-queued process is sent to the stdout of the shell it is running from, or it is mailed back to the user if the process is queued in true batch mode. The **bt** command can also send signals to processes running locally or remotely in batch mode and can display the status of running processes and queues.

A beta version of the **bt_tool**, **rpc.btd** and the **bt** command with the latest version of the BT scheduling class is available from the OPCOM FTP server (see page 13).

IP Multicasting

Enrico.Giuditti@Canada.Sun.Com

Internet protocol (IP) multicasting is the ability to send an IP datagram to a finite number of hosts using a single IP address. This IP address is called a multicast group, and may consist of zero or more hosts. A single host may belong to many groups, joining or leaving a group at any time during the life of that group. A host need not be a member of a group to send a message to that group.

There are two types of multicast groups based upon the IP address of the group (not on membership to the group).

a) Permanent - can have zero or more hosts with a well known IP address. (eg 224.0.0.1)

b) Transient - all multicast groups except permanent ones

As with unicast and broadcast datagrams, multicast datagrams are delivered on a best effort basis.

Multicasting is accomplished through the use of IP class D addressing. The four high order bits are set to "1110" with the remaining 28 bits used to identify the specific multicast group. This allows for a permissible range of 224.0.0.0 to 239.255.255.255. Address 224.0.0.0 will never be assigned by the NIC which issues all formal internet addresses. Address 224.0.0.1 is a permanent group to address all multicast hosts on a directly connected network.

Multicast IP packets must ultimately resolve down to an ethernet destination. To create a unique ethernet destination address, the lower order 23 bits of the IP multicast address are mapped onto the ethernet address. Coupled with the original upper 24 bits of the ethernet address (01-00-5E) hex, this new hardware address forms the multicast address at the ethernet level. Note that there is a potential for conflict here, as there are 28 significant bits in a Class D IP address, but only the lower 23 are being used.

A host can reside in three states when participating on a multicasting network. It can either:

a) participate fully by belonging to a multicast group thereby receiving and sending multicast datagrams, or

b) be configured to send but not receive multicast datagrams, or

c) not participate in multicasting at all.

The administrative tasks of informing hosts and gateways about multicasting operations is handled by the Internet Group Management Protocol(IGMP). This protocol resides in the IP layer

```
 * At this point we could print out some results.
 */
        print_usage( stdout, &pr );
        break;
   }       /* end of switch

   return( 0 );
}

/*
 * PRINT_USAGE - print (most of) the usage structure.
 *
 */

   void
print_usage( FILE *fp, prusage_t *p ) {
   double  user= ttodouble( &p->pr_utime );
   double  sys = ttodouble( &p->pr_stime );
   double  et  = ttodouble( &p->pr_term ) -
                 ttodouble( &p->pr_create );

   fprintf( fp, "%2.1fu %2.1fs %4.1fet %3.1fl%% %dmin+%dmaj
%di+%dob %di+oc %dv+%dinv\n",
        user,
        sys,
        et,
        100.0 * ( ( user + sys ) / et ),
        p->pr_minf,
        p->pr_majf,
        p->pr_inblk,
        p->pr_oublk,
        p->pr_ioch,
        p->pr_vctx,
        p->pr_ictx
   );

   return;
}

/*
 * TTODOUBLE - convert a timestruc_t to a double.
 *
 */

   double
ttodouble( timestruc_t *t ) {
   return( (double) t->tv_sec + ((double) t->tv_nsec)
                        / ((double) NANOSEC) );
}
```

What Next?

Subsequent articles will further explore `/proc` and many of the other ioctl's that it supports.

The getrusage system call and prusage program can be obtained from OPCOM via the anonymous FTP server (see page 13).

Batch Processing on Solaris 2.x

Ron.Winacott@Canada.Sun.Com

In the September issue of SunOpsis, you were introduced to the batch scheduling class, BT. This month's article will describe an environment that allows users to run commands in batch mode. This environment is composed of:

- **bt_tool**, a user interface
- a command line environment, **bt**, and
- **rpc.btd**, an rpc daemon that services local and remote requests

This environment arose from the need to run commands sequentially in what could be called *true batch mode*, as well as the requirement to borrow idle cpu time from a remote system that has BT loaded.

In this context, *true batch mode* is used to mean that if the user starts a number of processes (for example, P1, P2 and P3) at the same priority in BT mode, they will each execute sequentially.

There is an inherent problem, however, since BT (the scheduling class) has no way to prevent a process from joining a dispatch queue that may have something else running on it. As a result, this needed functionality has been designed into the **rpc.btd** daemon.

Both **bt_tool**, and the **bt** command use rpc calls to control processes on the local and remote systems running BT.

Some of the other features are:

- Start a process in one of the four priority slots
- Monitor any processes running on your system in batch mode
- Send STOP, CONTinue, or KILL signals to any process running in BT that you own on the local or remote host
- Run a command in BT mode on a remote system
- True batch mode queuing of processes when the requested slot is busy
- Automatic scheduling of a queued process when the slot becomes free
- Command history for fast recall of a previously run process in bt_tool
- Process output is saved to a log window in bt_tool, and displayed back to you when using the **bt** command
- Filter the status list in the **bt_tool**. Any process that is running in BT mode can be seen by any **bt_tool**. You may want to see just the processes that are under the control of your **bt_tool**

BT_TOOL

When a command is entered into **bt_tool**, all the information about the state of the tool is collected and saved. This information includes the true batch mode, the local or remote host name, and the current working directory of the tool at this time, the environment, and the command to execute. A request to start the command is made to the **rpc.btd**.

If true batch mode is enabled, then, using the `/proc` file system, the **rpc.btd** checks for any other processes running in the required priority slot. The **PIOCPSINFO ioctl(2)** call is used

```
* (asynchronous)
*  % cc -DUSE_SIGCHLD -o prusage prusage.c
*
* written: RWMarejka; 1992.10.08
* (c) 1992 SMCC, a division of SMI
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/signal.h>
#include <sys/fault.h>
#include <sys/syscall.h>
#include <sys/procfs.h>
#include <unistd.h>
#include <limits.h>

#if defined(USE_SIGCHLD)
#  include <signal.h>
#endif

/*
* External References
*/

extern int     getprusage( pid_t, prusage_t * );
extern void    print_usage( FILE *, prusage_t * );
extern double  ttodouble( timestruc_t * );

/*
* External Declarations
*/

int        status;     /* child's exit status */
prusage_t  pr;         /* child's usage structure */

/*
* SIGCHILD - catch the SIGCHLD event.
*
*/

#if defined(USE_SIGCHLD)

   void
sigchild() {
   pid_t   pid;

   if ( ( pid = waitpid( 0, &status, WNOWAIT ) ) == -1 ) {
       perror( "sigchild:waitpid:WNOWAIT" );
       exit( 1 );
   }

   if ( getprusage( pid, &pr ) == -1 )
       perror( "sigchild:getprusage" );

   if ( ( pid = waitpid( pid, &status, 0 ) ) == -1 ) {
       perror( "sigchild:waitpid:0" );
       exit( 1 );
   }

   return;
}

#endif /* defined(USE_SIGCHLD) */

/*
* GETPRUSAGE - get process usage
*
*/

   int
getprusage( pid_t pid, prusage_t *pup ) {
   int     s = 0;
   int     fd;
   char    buf[PATH_MAX];

   sprintf( buf, "/proc/%05d", pid );

   if ( ( fd = open( buf, O_RDONLY ) ) != -1 ) {
       if ( ioctl( fd, PIOCUSAGE, pup ) == -1 )
           s   = -1;

       if ( close( fd ) == -1 )
           s   = -1;
   }
   else
       s   = -1;

   return( s );
}

/*
* Main - Affirmation
*
*/

   int
main( int argc, char *argv[] ) {
   pid_t      pid;/* process id of child */
#if defined(USE_SIGCHLD)
   struct  sigaction   act;
   sigset_tset;

   act.sa_handler=sigchild;
   act.sa_flags=SA_NOCLDSTOP;
   sigemptyset( &act.sa_mask );

   sigfillset( &set );
   sigdelset( &set, SIGCHLD );

   if ( sigaction( SIGCHLD, &act, NULL ) == -1 ) {
       perror( "main:sigaction" );
       exit( 1 );
   }
#endif
   switch ( pid = vfork() ) {
    case -1/*: Error*/
       perror( "fork" );
       exit( 1 );

    case 0/*: Child*/
/*
* Go off and run the program.
*/
       execvp( argv[1], &argv[1] );
       perror( "execve" );
       exit( 1 );

    default/*: Parent*/
/*
* Wait for the child to finish but keep it in a "wait-able" s
* so that we may "procfs" it.
*/
#if defined(USE_SIGCHLD)
       sigsuspend( &set );
#else
       if ( waitpid( pid, &status, WNOWAIT ) == -1 ) {
           perror( "waitpid:WNOWAIT" );
           exit( 1 );
       }

       if ( getprusage( pid, &pr ) == -1 )
           perror( "getprusage" );

       if ( waitpid( pid, &status, 0 ) == -1 ) {
           perror( "waitpid:0" );
           exit( 1 );
       }
#endif /* defined(USE_SIGCHLD) */
/*
```

that maintains the "running" usage information. It is a simple matter to traverse these structures and compute a resource usage structure.

The loadable system call is complete and has been tested. However, it is not a good solution since:

a) `RUSAGE_CHILDREN` is not supported,

b) loadable system calls are undocumented,

c) it requires a small stub (the user interface to getrusage) to be compiled with the application.

The loadable system call is available from the OPCOM FTP server (see page 13).

Beyond getrusage(2)

Fortunately, Solaris 2.x offers much more powerful instrumentation and usage collection than Solaris 1.x. All of this is available using the `/proc` filesystem. The procfs filesystem is documented in the **proc(4)** manual page, a seventeen page document that fully describes the interface and it's usage. The core idea of procfs is to provide a filesystem type interface to the address space of every process currently executing on a system. Using a filesystem interface:

- can be cleanly implemented using the existing VFS/VNODE layer,
- provides a programming interface familiar to programmers.

The programming interface is a simple as:

```
int fd;
charbuf[PATH_MAX];

sprintf( buf, "/proc/%05d", getpid() );
fd = open( buf, O_RDONLY );
...
read(2)/lseek(2)/ioctl(2)
...
close( fd );
```

The only place where the filesystem does not work is if the opened process exits. In this case the file descriptor is no longer valid and will generate errors when used. The only thing that you can do at that point is to close the descriptor.

The above code fragment, while trivial, does illustrate the basic concept. The procfs is most useful when examining other processes and using the large set of **ioctl(2)** options. One ioctl will return the resource usage of the opened process (`PIOCUSAGE`) and this can be used as a replacement to getrusage. There are two possible implementation paths: self and child. The self path is as simple as:

```
int         fd;
char        buf[PATH_MAX];
prusage_t   pr;

sprintf( buf, "/proc/%05d", getpid() );
fd = open( buf, O_RDONLY );
ioctl( fd, PIOCUSAGE, &pr );
close( fd );
```

You now have the current resource usage of the executing process, this includes all LWPs (active and exited).

To obtain resource usage for children there are two possible methods:

- synchronously using **waitpid(2)**
- asynchronously using a signal handler for `SIGCHLD`.

The synchronous scheme is:

```
    Parent          Child

1)  fork            fork
2)                  child exec's program (optional)
3) waitpid
4)                  child exits
5) waitpid returns
6) open /proc for child
7) ioctl(PIOCUSAGE)
8) close
9) waitpid
```

Why the two calls to waitpid? The first uses the `WNOWAIT` option that will return when the child exits but leaves the child in the ZOMBIE state. This keeps the `/proc` entry active and allows the open, ioctl, close sequence to correctly capture the results. The second waitpid is called with no options and will release the ZOMBIE child process from the system.

The asynchronous method is virtually identical, except of course a signal handler executes the steps 3-9. As a note, remember to set the `SA_NOCLDSTOP` option in the flags member of the `sigaction` structure. You probably are not interested when children stop, only when they exit.

The program below implements (an approximate) equivalent of the **csh(1)** built-in **time** command.

```
/*
 * cc -g -o prusage prusage.c
 *
 * PRUSAGE - get process usage (the Solaris way).
 *
 * A simple program that demonstrates how to use
 * the procfs to retreive process usage statistics for
 * children.
 *
 * The program can be compiled to operate synchronously
 * with the child or asynchronously. The former scheme
 * uses waitpid while the latter uses a SIGCHLD handler.
 * To compile use:
 *
 * (synchronous)
 *  % cc -o prusage prusage.c
 * or
```

```
    mutex_lock( &prtmutex );
    printf( "server:server:tid:%2d,fd:%2d\n", tid, fd );
    mutex_unlock( &prtmutex );

    while ( fgets( buf, BUFSIZ, fp ) ) {
        mutex_lock( &prtmutex );
        printf( "server:tid:%2d: %s", tid, buf );
        mutex_unlock( &prtmutex );

        if ( strcmp( buf, "END" ) == 0 )
            break;
    }

    fclose( fp );

    mutex_lock( &prtmutex );
    printf( "server:thr_exit:tid:%2d\n", tid );
    mutex_unlock( &prtmutex );

    thr_exit( fd );
/* NOTREACHED */
}
```

The getrusage(2) Syscall

Richard.Marejka@Canada.Sun.Com

Introduction

If the number of customer calls is any indication, the favourite system call from Solaris 1.x is getrusage(2).

Being able to instrument a process is probably one of the key issues in understanding it's efficiency. Programmers use resource usage statistics for various purposes such as capacity planning, fine-grain accounting and system utilization. The familiar getrusage interface from Solaris 1.x is still present in Solaris 2.x. However, the amount of useful information has been reduced to two time fields. The good news is that getrusage has a replacement that is much better. In this article we'll explore the old interface, the new interface, the replacement, and how it can be used.

Solaris 1.x Behaviour

In Solaris 1.x, the operating system keeps track of both process and child resource usage in the process structure. The process usage is current to the last system call or context switch and the child usage is the total over all exited children (of the process).

The resource usage structure looks like:

```
struct rusage {
    struct timeval ru_utime;  /* user time used */
    struct timeval ru_stime;  /* system time used */
    int ru_maxrss;            /* maximum resident set size */
    int ru_ixrss;             /* currently 0 */
    int ru_idrss;             /* integral resident set size */
    int ru_isrss;             /* currently 0 */
    int ru_minflt;            /* page faults not requiring
                                 physical I/O */
    int ru_majflt;            /* page faults requiring
                                 physical I/O */
    int ru_nswap;             /* swaps */
    int ru_inblock;           /* block input operations */
    int ru_oublock;           /* block output operations */
    int ru_msgsnd;            /* messages sent */
    int ru_msgrcv;            /* messages received */
    int ru_nsignals;          /* signals received */
    int ru_nvcsw;             /* voluntary context switches */
    int ru_nivcsw;            /* involuntary context switches */
};
```

Some of these fields are zero, some are of little interest (signals and context switches) and some you cannot live without, for example: `ru_minflt`, `ru_majflt`, `ru_inblock`.

Solaris 2.x Implementation

In Solaris 2.x, numerous aspects of the system have changed. The kernel scheduling unit is now the LWP (light-weight process) and not simply the process. This change allows a process to have many concurrent threads of execution.

Two of the changes with regard to the process structure are:

1) the structure only has process summary information for LWP's that have exited

2) the structure does not accumulate statistics for child processes.

Item #1 leads to the conclusion that the old Solaris 1.x code:

```
struct  rusage  ru;
...
getrusage( RUSAGE_SELF, &ru );
```

cannot return any usable information. Since the LWP usage is added into the process usage only after it exits and most programs only have a single thread of execution (and hence a single LWP), the getrusage information will be zero (actually the accumulated system and user time are correct).

A direct result of item #2 is that the old Solaris 1.x system call

```
struct  rusage  ru;
...
getrusage( RUSAGE_CHILDREN, &ru );
```

that used to return child resource usage information cannot return any usable data (it actually is able to compute child user and system time usage). The remainder of the structure is zero.

Where does this leave us, since neither form of getrusage returns much useable data?

There is one possible solution for `RUSAGE_SELF`, but there is no solution for `RUSAGE_CHILDREN` (the data is just not available). The `RUSAGE_SELF` solution involves a loadable system call to traverse three structures in the kernel: process, thread and light-weight process. Each process has a list of threads within the process and each thread is linked to an LWP. It is each LWP

```
    }

    if ( !( hp = gethostbyname( argv[1] ) ) ) {
        perror( "gethostbyname" );
        exit( 1 );
    }
    else
        server_addr = *(u_long *) hp->h_addr;

    if ( ( sock = socket( AF_INET, SOCK_STREAM, 0 ) ) == -1 ) {
        perror( "socket" );
        exit( 1 );
    }

    saddr.sin_port = htons( atoi( argv[2] ) );
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = htonl( server_addr );

    if ( connect( sock, &saddr,
            sizeof( struct sockaddr_in ) ) == -1 ) {
        perror( "connect" );
        exit( 1 );
    }

    if ( !( fp = fdopen( sock, "w" ) ) ) {
        perror( "client:fdopen" );
        exit( 1 );
    }

    while ( gets( buf ) ) {
        fprintf( fp, "%s ", buf );
        fflush( fp );

        if ( strcmp( "END", buf ) == 0 )
            break;
    }

    fclose( fp );
    return( 0 );
}
```

The Server

```
/*
 % cc -L $LD_RUN_PATH -o server server.c -lsocket -lnsl -lthread
 *
 * SERVER - an internet TCP server.
 * (c) 1992 SMCC, a division of SMI
 *
 */

/*
 * Include Files
 */

#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <thread.h>
#include <synch.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

/*
 * Constants & Macros
 */

#define QMAX    5

/*
 * External References
 */

extern int server( int );

/*
 * External Declarations
 */

mutex_t    prtmutex;

/*
 * Main - The Summer Side of Life
 *
 */

main( int argc, char *argv[] ) {
    struct sockaddr_in      saddr;
    struct sockaddr_in      caddr;
    int         addrlen = sizeof( struct sockaddr_in );
    int         sock;
    int         conn;

    if ( argc != 2 ) {
        fprintf( stderr, "usage: server port\n" );
        exit( 1 );
    }

    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = htonl( INADDR_ANY );
    saddr.sin_port = htons( atoi( argv[1] ) );

    if ( ( sock = socket( AF_INET, SOCK_STREAM, 0 ) ) == -1 )
        perror( "socket" );
        exit( 1 );
    }

    if ( bind( sock, &saddr, sizeof( saddr ) ) ) {
        perror( "bind" );
        exit( 1 );
    }

    if ( listen( sock, QMAX ) == -1 ) {
        perror( "listen" );
        exit( 1 );
    }

    while ( ( conn = accept( sock, &caddr, &addrlen ) ) != -1
        thread_t tid;

        if ( thr_create( NULL, 0, server, conn,
                        THR_DETACHED, &tid ) )
            perror( "server:thr_create" );

        mutex_lock( &prtmutex );
        printf( "server: thr_create:%2d:conn:%d\n",
                                        tid, conn);
        mutex_unlock( &prtmutex );

        addrlen = sizeof( struct sockaddr_in );
    }

    return( 0 );
}

/*
 * SERVER - the server thread function
 *
 */
int
server( int fd ) {
    char        buf[BUFSIZ];
    FILE        *fp = fdopen( fd, "r" );
    thread_t    tid = thr_self();

    if ( !fp )
        fprintf(stderr, "server:tid:%2d: fdopen failed\n", tid );
```

testing your changes. Beware of new synchronization techniques and any new critical sections you may have introduced.

Multithreading

In Solaris 2.x all device drivers must be MT safe. All entry points of your drivers (except loading, unloading, attaching and detaching) of your driver must be re-entrant.

Any critical sections of these entry points must be protected by mutex locks. The calls to `mutex_enter` and `mutex_exit` must always appear in pairs. You should note that the old method of using `spl()` and `splx()` no longer protect your data structures.

The amount of parallelism present in your driver determines whether it is considered MT-cold or MT-hot. All drivers should strive for the latter. This quality of drivers is usually proportional to the amount of effort expended in the design process.

More Information

A good source of information is available in the *Writing Device Drivers* manual.

Using Threads And Sockets

Richard.Marejka@Canada.Sun.Com

User Level Threads can be used in networking programs as an alternative to the old paradigm of:

Server:
socket → bind → listen → accept → fork → close

Child:
fork return → read/write → close → exit

where the server is free to accept further connection after the fork, and the child is left to service the current connection for the life of that connection. User Level Threads simplifies this model considerably. The new (threaded) paradigm is:

Server (main thread)
socket → bind → listen → accept → thr_create → close

Server (service thread):
→ read/write → close → thr_exit

where the service thread runs in the same address space as the main thread (that monitors the rendezvous socket and accepts network connections). This allows multiple connections to the same server to operate in parallel, saving the expense and overhead of a fork while retaining most of the features of the old paradigm.

Some of the features that cannot be carried into the new paradigm are:

- setuid, since the entire process is affected
- chdir, since the entire process is affected
- execve, since the server would disappear

From the above list, any operation that would modify the operating environment of one thread must be examined. Fortunately this is not usually a problem since most server children simply take over the management of one conversation with a client.

The two sample programs below implement a simple server-client pair. It should be noted that the client has no knowledge that its server is multi-threaded. The server required only minor changes to operate in a threaded environment, in particular, the accept loop (as demonstrated from the algorithm above) is the only point of modification. This source code is available electronically from the OPCOM FTP server (see page 13).

The Client Code

```
/*
 * cc -o client client.c -lsocket -lnsl
 *
 * CLIENT - an internet TCP client.
 * (c) 1992 SMCC, a division of SMI
 */

 /*
 * Include Files
 */

 #include <stdio.h>
 #include <stdlib.h>
 #include <signal.h>
 #include <errno.h>
 #include <unistd.h>
 #include <string.h>
 #include <sys/types.h>
 #include <sys/socket.h>
 #include <sys/sockio.h>
 #include <sys/ioctl.h>
 #include <netinet/in.h>
 #include <netdb.h>


 /*
 * Main -
 *
 */

 main( int argc, char *argv[] ) {
    int                 sock;
    long                server_addr;
    struct sockaddr_in  saddr;
    struct hostent      *hp;
    char                buf[BUFSIZ];
    FILE                *fp;

    if ( argc != 3 ) {
        fprintf( stderr, "usage: client server port" );
        exit( 1 );
```

1) use **sigaction(2)**. sigaction behaves the same way as Solaris 1.x **signal(2)** , with the extra capability to mask out signals while the signal handler is executed. Although sigaction is part of the SVID and POSIX.1,it is not defined in the ANSI-C standard;

2) use **sigset(2)**. The signal is masked after it is received and, reset when the signal handler returns. If `sigset(sig, SIG_HOLD)` is executed inside the signal handler, the signal handler does not get reset. sigset is defined for the SVID but it is not part of POSIX.1 nor ANSI-C standard

I chose to use sigaction in my port.

One other tricky part with signal handling was with SIGIO delivery. In Solaris 1.x SIGIO is delivered to the process that opened the device (e.g. `/dev/kbd`). However, under Solaris 2.x, one has to notify the stream head of the intention to receive SIGIO (now called SIGPOLL for SVID compliance) with the following ioctl:

```
(void) ioctl(kbdFd, I_SETSIG, S_INPUT);
```

Client applications that form part of X11R5 core distribution have also been ported. The most difficult was `xload`, which is a cpu perfmeter. The code uses a lot of kernel structures, most of which have changed in Solaris 2.x; I ended up rewriting much of the code.

How To Get It

The binary copy of X11R5 for Solaris 2.x can be FTP'ed from The OPCOM FTP server (see page 13), in `/pub/x11r5`, or in `/pub/tars/x11r5.tar.Z`. I have done minimal testing on a 4/75GX (SS2), 4/25 (ELC) and 4/630-GX. I would appreciate any feedback you may have via direct email.

Porting A Device Driver to Solaris 2.x

Gil.Hauer@Canada.Sun.Com

With the introduction of loadable device drivers in SunOS 4.0.3, the task of developing and debugging a device driver was made somewhat easier. Since the developer didn't need to recompile and reboot the operating system, more time could be spent engineering and qualifying the driver.

With Solaris 2.x introducing several new concepts in the realm of device drivers, the ability to load and unload drivers is again a time-saver.

This article tries to provide a simplified guide to porting device drivers from a Solaris 1.x (SunOS 4.x) environment to the new Solaris 2.x environment.

Compiling And Loading

The easiest way to tackle the port is one step at a time. Hence, the first step should be to simply get the driver to compile and load under Solaris 2.x.

One way to do this is to comment (or #ifdef) out all of the code and data structures that are not required for loading and unloading. If your driver was not loadable under Solaris 1.x, you will need to add the required routines. All drivers are loadable in Solaris 2.x.

You should be left with the `_init`, `_info` and `_fini` routines, the `xx_attach`, `xx_detach`, `xx_identify`, `xx_probe` and `xx_getinfo` functions as well as their related data structures. All of these routines need to be functional and correct before loading and unloading can work.

A few more changes are required in your source file. Specifically, you may need to change the header files #include-ed in your driver. All of the required header files are now located in `/usr/include/sys` and there are some new ones that are required in order to support the DDI/DKI interfaces, such as `<sys/ddi.h> and <sys/sunddi.h>`.

The makefile that you use also requires a small change. Drivers should be compiled with the _KERNEL flag defined. You should do this by adding the -D_KERNEL option to your compile command. Also, you need to add a link step, using **ld(1)** since .o files are not loadable in Solaris 2.x.

It is a good idea at this point to review your source and provide ANSI C function prototypes. This allows the compiler to perform more rigorous type-checking and to help you trap any errors *before* a kernel panic. As well, you should use the ANSI volatile and const keywords where appropriate.

Try compiling the driver and fixing any problems. Once the driver compiles "cleanly" (there are no syntax errors), you may copy the executable into `/usr/kernel/drv` and introduce it to the system using the **add_drv(1m)** command. This command will automatically update files in `/etc` and try to load the executable. If it fails, you will usually need to use the **rem_drv(1m)** command to remove the driver, so you can fix the problem and start again.

Remaining Entry Points

Now that the driver loads and unloads successfully you are ready to port the remaining entry points of the driver. All of the entry point definitions must be checked carefully since there are many new parameters and their functionality has changed somewhat. Proper use of ANSI C prototypes are very helpful at this stage.

It is best to tackle this stage bit by bit, removing the comments (or #ifdefs) for one or two entry points at a time and carefully

SUN MICROSYSTEMS COMPUTER CORPORATION

SunOpsis

The Solaris 2.x Migration Support Center Newsletter

1-800-363-6200

VOLUME 1, NUMBER 3, October 1992

The Publisher's Corner

Dean.Kemp@Canada.Sun.Com
Mgr Solaris 2.x Migration Support Center

The Beta version of Solaris 2.1 is now available, which features improved performance at all levels, and window performance that meets that of Solaris 1.x. For more detailed information on Solaris 2.1, please refer to our preview in last month's issue. If you don't have last month's issue, you can get it from the OPCOM FTP server (see page 13), from the OPCOM support channels, or from your local Sales Account Manager or Systems Engineer. To get access to the Solaris 2.1 Beta, please contact either your local Sales Account Manager or Systems Engineer.

The Threads Programmers Guide is now available as well. For those of you who are on the threads early access program, please contact us, and we will be happy to send one out to you.

The OpCom T-shirt contest is still on. The complete details are in the September issue, but I'll mention them again briefly; Write an article for publication in a future issue of SunOpsis, and you could win an OpCom T-shirt. Each article must have something to do with Solaris 2.x, and the article will become the property of Sun Microsystems, Inc. It can be any length, but 2 - 2 1/2 pages is preferred.

And finally, we value your suggestions. Please do not hesitate to contact me if I can be of assistance.

INDEX

X11R5 Porting Experience

Larry.Tsui@Canada.Sun.Com

This article summarizes my porting effort of the generic MIT X11R5. This was a straight port of X11R5 with no enhancements.

The first hurdle was creating suitable configuration files. The MIT distribution comes with configurations for SunOS4.1.x and generic SVR4 machines. I merged these two sets of configuration files and made minor changes to suit the needs of Solaris 2.x; for an example, the variable SharedLibLoadFlags (in `mit/config/sv4Lib.tmpl`) was changed from "-G -z text" to "-G".

Porting the actual code was quite straightforward in most parts. Most of the library calls that are different in Solaris 2.x have been documented in the System Transition Guide for Application Developers (which comes with Solaris 2.x AnswerBook and the Solaris Migration Kit).

Signal handling in the device dependent layer (`mit/server/ddx/sun`) and the os layer (`mit/server/os`) was a little more tricky. The traditional **signal(2)** call behaves differently in Solaris 2.x. After a signal has been caught and before it is delivered, the signal handler is reset back to its default handler. In Solaris 1.x, the signal handler remains unchanged after signal disposition, but it does not allow signal masking during signal delivery. There are two ways to address this problem: