Introduction into Multicore Programming

Károly Bósa
(Karoly.Bosa@jku.at)

Research Institute for Symbolic Computation
(RISC)

Topic

Processes:

- What is a process?
- Why Processes and Not Threads?
- Process Control Block
- The address space/image of a process
- Process States
- Process Scheduling
- Context Switch
- Monitoring Processes with the ps Utility
- Creating Processes
- Calls Regarding Processes
- Introduction into Process Tracing

Setting and Getting Process Dynamic Priorities

- Only superuser and kernel processes can raise static priority levels (from the kernel 2.6.12 it is changed).
- A process inherits the priority of the process that created it.
- Each process with 0th static priority has a nice value that is used to calculate a dynamic priority level of the calling process.
- But the nice value of a process can be lowered by raising its nice value.

```
#include <unistd.h>

int nice(int incr);
```

- The *incr* parameter is the value added to the current nice value of the calling process.

- setpriority()/ getpriority() sets/returns the nice value for a process, process group,

```
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int value);
```

- The range of nice value in the Linux environment is - 20 to 19.

Example for setpriority()


```
#include <sys/resource.h>

//...
id_t pid = 0;
int which  = PRIO_PROCESS;
int value = 10;
int nice_value;
int ret;

nice_value = getpriority(which,pid);
if(nice_value < value){
   ret = setpriority(which,pid,value);
}
//...
```

- The *which* parameter can specify a process, process group, or a user. It can have the following values:
  - `PRIO_PROCESS`: Indicates a process
  - `PRIO_PGRP`: Indicates a process group
  - `PRIO_USER`: Indicates a user

- A 0 value for *who* indicates the current process, process group, or user.

Envirnoment Variables and Processes

Karoly.Bosa@jku.at

- Environment variables can be used to transmit any useful user – defined information between the parent and the child processes.

- Functions which can be query, add or modify environment variables:

```
#include <stdlib.h>

char *getenv(const char *name);
int setenv(const char *name, const char *value, int overwrite);
void unsetenv(const char *name);
```

- getenv() is used to determine whether a specific variable has been set.

- setenv() is used to change or add an environment variable.

- unsetenv() removes the environment variable specified by name .

The exit(), and abort() Calls

- There are two functions a process can call for self - termination, exit() and abort().

```
#include <stdlib.h>

void exit(int status);
void abort(void);
```

- The exit() function causes a normal termination of the calling process.

- The abort() function causes an abnormal termination of the calling process.

The kill() Function

- The kill() system call can be used to send any signal to any process group or process.

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

- The calling process can send the signal to one or several processes under these conditions:
  - **pid > 0**: The signal is sent to the process whose id is equal to the pid.
  - **pid = 0:** The signal is sent to all the processes whose process group id is the same as the calling process.
  - **pid = – 1:** The signal is sent to all processes to which the calling process has permission to send that signal.
  - **pid < – 1:** The signal is sent to all processes whose process group id is equal to the absolute value of pid and for which the calling process has permission to send that signal.
- To kill/terminate a process, sig has the value **SIGKILL**.

Resource Limits

- POSIX defines functions that restrict a process's ability to use certain resources.

- The operating system sets limitations on a process's ability to utilize system resources. These resource limits affect the following:
    - Size of the process's stack
    - Size of file and core file creation
    - Amount of CPU usage (size of time slice)
    - Amount of memory usage
    - Number of open file descriptors

- The operating system sets a hard limit on resource usage by a process.

- The process can set or change the soft limit of its resources. Its value should not exceed the hard limit set by the operating system

A process can lower its hard limit, but it is irreversible.

Only processes with special privileges can raise their hard limit.

POSIX Functions to Set Resource Limits I.

Karoly.Bosa@jku.at

```
#include <sys/resource.h>

int setrlimit(int resource, const struct rlimit *rlp);
int getrlimit(int resource, struct rlimit *rlp);
int getrusage(int who, struct rusage *r_usage);
```

- The **setrlimit()** function is used to set limits on the consumption of specified resources. This function can set both hard and soft limits.

- The parameter *resource* represents the resource type:

Resource Definitions	Descriptions
RLIMIT_CORE	Maximum size of a core file in bytes that may be created by a process
RLIMIT_CPU	Maximum amount of CPU time in seconds that may be used by a process
RLIMIT_DATA	Maximum size of a process's data segment in bytes
RLIMIT_FSIZE	Maximum size of a file in bytes that may be created by a process
RLIMIT_NOFILE	The maximum number of files that the process can open.
RLIMIT_STACK	Maximum size of a process's stack in bytes
RLIMIT_AS	Maximum size of a process's total available memory in bytes

POSIX Functions to Set Resource Limits II.

Karoly.Bosa@jku.at

```
#include <sys/resource.h>

int setrlimit(int resource, const struct rlimit *rlp);
int getrlimit(int resource, struct rlimit *rlp);
int getrusage(int who, struct rusage *r_usage);
```

- The soft and hard limits of the specified resource are represented by the *rlp* paramater. The *rlp* parameter points to a struct *rlimit:*

```
struct rlimit
{
        rlim_t rlim_cur;
        rlim_t rlim_max;
}
```

- rlim_cur and rlim_max can be assigned any value. They can also be assigned these symbolic constants defined in the header <sys/resource.h> :
  - RLIM_INFINITY : Indicates no limit
  - RLIM_SAVED_MAX : Indicates an unrepresentable saved hard limit
  - RLIM_SAVED_CUR : Indicates an unrepresentable saved soft limit

POSIX Functions to Set Resource Limits III.


```
#include <sys/resource.h>

int setrlimit(int resource, const struct rlimit *rlp);
int getrlimit(int resource, struct rlimit *rlp);
int getrusage(int who, struct rusage *r_usage);
```

- The **getrlimit()** returns the soft and hard limit of the specified resource in the rlp object.

- Both the getrlimit() and setrlimit() functions return 0 if successful and - 1 if unsuccessful.

POSIX Functions to Set Resource Limits IV.

Karoly.Bosa@jku.at

```
#include <sys/resource.h>

int setrlimit(int resource, const struct rlimit *rlp);
int getrlimit(int resource, struct rlimit *rlp);
int getrusage(int who, struct rusage *r_usage);
```

- The **getrusage()** returns information about the measures of resources used by the calling process or its child.

- The parameter *who* can have these values:
  - RUSAGE_SELF
  - RUSAGE_CHILDREN

- The information is returned in *r_usage* which points to a struct *rusage*:

```
struct rusage {
struct timeval ru_utime;    /* user time used */
struct timeval ru_stime;    /* system time used */
long ru_maxrss;             /*maximum resident set size */
long ru_ixrss;              /* integral shared memory size */
long ru_idrss;              /* integral unshared data size */
long ru_isrss;              /* integral unshared stack size */
long ru_minflt;             /* page reclaims */
long ru_majflt;             /* page faults */
long ru_nswap;              /* swaps */
long ru_inblock;            /* block input operations */
long ru_oublock;            /* block output operations */
long ru_msgsnd;             /* messages sent */
long ru_msgrcv;             /* messages received */
long ru_nsignals;           /* signals received */
long ru_nvcsw;              /* voluntary context switches */
long ru_nivcsw;             /* involuntary context switches */
};
```

POSIX Functions to Set Resource Limits IV.

Karoly.Bosa@jku.at

```
#include <sys/resource.h>

int setrlimit(int resource, const struct rlimit *rlp);
int getrlimit(int resource, struct rlimit *rlp);
int getrusage(int who, struct rusage *r_usage);
```

- The **getrusage()** returns information about the measures of resources used by the calling process or its child.

- The parameter *who* can have these values:
  - RUSAGE_SELF
  - RUSAGE_CHILDREN

- The information is returned in *r_usage* which points to a struct *rusage*:

```
struct rusage {
struct timeval ru_utime;   /* user time used */
struct timeval ru_stime;   /* system time used */
long ru_maxrss;            /*maximum resident set size */
long ru_ixrss;             /* integral shared memory size */
long ru_idrss;             /* integral unshared data size */
long ru_isrss;             /* integral unshared stack size */
long ru_minflt;            /* page reclaims */
long ru_majflt;            /* page faults */
long ru_nswap;             /* swaps */
long ru_inblock;           /* block input operations */
long ru_oublock;           /* block output operations */
long ru_msgsnd;            /* messages sent */
long ru_msgrcv;            /* messages received */
long ru_nsignals;          /* signals received */
long ru_nvcsw;             /* voluntary context switches */
long ru_nivcsw;            /* involuntary context switches */
};
```

Supported in Linux 2.4 Supported from Linux 2.6

Example for Setting Resource Limits


```
#include <sys/resource.h>

//...
struct rlimit R_limit;
struct rlimit R_limit_values;

//...

R_limit.rlim_cur = 2000;
R_limit.rlim_max = RLIM_SAVED_MAX;
setrlimit(RLIMIT_FSIZE,&R_limit);
getrlimit(RLIMIT_FSIZE,&R_limit_values);
cout << "file size soft limit: " << R_limit_values.rlim_cur << endl;

//...
```

Example for getrusages()


```
#include<sys/time.h>
#include<sys/resource.h>

//…

double getcputime(void) {
        struct timeval tim;
        struct rusage ru;
        getrusage(RUSAGE_SELF, &ru);
        tim = ru.ru_stime;
        double t = (double)tim.tv_sec + (double)tim.tv_usec / 1000000.0;
        tim = ru.ru_stime;
        t += (double)tim.tv_sec + (double)tim.tv_usec / 1000000.0;
        return t;
}
int main(void) {
        //…
        double t = getcputime();
        printf("The total CPU time comsumed by the current process is: %.6lf (sec)\n", t);
        return 0;
}
```


Remark: struct *timeval* is defined in <sys/time.h>.

The wait() Function Call

- Processes can suspend execution until a child process terminates by executing wait() system call.

```
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- The wait() function obtains the exit status from the process table.
- If the parent process has more than one terminated child process, the wait() function retrieves the exit status for only one child.
- Both the wait() and waitpid() functions return the PID of the child process whose exit status was obtained.

The waitpid() Function Call

- The waitpid() function is the same as wait() , except that it takes some additional parameters

```
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- The *pid* parameter specifies a set of child processes for which the exit status is retrieved:
  - **pid > 0:** A single child process,
  - **pid = 0:** Any child process whose group id is the same as the calling process,
  - **pid < – 1**: Any child processes whose group id is equal to the absolute value of pid,
  - **pid = – 1**: Any child processes.

- The options parameter determines how the wait should behave:
  - **WNOHANG :** Return immediately if no child has exited.
  - **WUNTRACED :** also return if specified child has stopped.
  - **WCONTINUED :** (since Linux 2.6.10) return and reports the exit status of specified child is continued(resumed).

The *Status* Parameter of wait() Calls

- If the value in status is 0, then the child process has terminated successfully.
- Macros for Evaluating status:
  - **WIFEXITED(*status*)** returns true if the child terminated normally, that is, by calling ***exit****(3)* or **_exit**(2), or by returning from main().
  - **WEXITSTATUS(*status*)** If WIFEXITED is nonzero, this evaluates to the low-order 8 bits of the status argument the terminated child process passed to _exit(), exit(), or the value returned from main().
  - **WIFSIGNALED(*status*)** returns true if the child process was terminated by a signal.
  - **WTERMSIG(*status*)** returns the number of the signal that caused the child process to terminate. This macro should only be employed if **WIFSIGNALED** returned true.
  - **WCOREDUMP(*status*)** returns true if the child produced a core dump. This macro should only be employed if **WIFSIGNALED** returned true.
  - **WIFSTOPPED(*status*)** returns true if the child process was stopped by delivery of a signal; this is only possible if the call was done using **WUNTRACED** or when the child is being traced (see ***ptrace****(2)*).
  - **WSTOPSIG(*status*)** returns the number of the signal which caused the child to stop. This macro should only be employed if **WIFSTOPPED** returned true.
  - **WIFCONTINUED(*status*)** (Since Linux 2.6.10) returns true if the child process was resumed by delivery of **SIGCONT**.

Example Multifork


```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int ac, char * av[])
{
   if(ac < 3)
   {
       printf("usage : multifork command arg1
arg2 ...n");
       exit(EXIT_FAILURE);
   }
   else{
       int i;
       char * command = av[1], * arg;
       for(i = 2; i < ac; i++)
       {
          arg = av[i];
          if(!fork())
             execl(command, "", arg, NULL);
       }
       for(i = 2; i < ac; i++)
       {
           int status;
            wait(&status);
        }
         exit(EXIT_SUCCESS);
    }
}
```

Output Example :

$ ./multifork /bin/echo aaaa bbbbb ccccc
aaaa
ccccc
bbbbb
./multifork /bin/echo aaaa bbbbb ccccc
aaaa
bbbbb
ccccc
./multifork /bin/echo aaaa bbbbb ccccc
bbbbb
ccccc
aaaa

Process Trace Introduction I.

Karoly.Bosa@jku.at

- Have you ever wondered how system calls can be intercepted?
- Have you ever tried fooling the kernel by changing system call arguments?
- Have you ever wondered how debuggers stop a running process and let you take control of the process?
- **ptrace** is a system call found in several Unix and Unix-like operating systems.
- By using ptrace one process can control another, enabling the controller to inspect and manipulate the internal state of its target.
- ptrace is used by debuggers and other code-analysis tools, mostly as aids to software development.

Process Trace Introduction II.

Karoly.Bosa@jku.at

- Operating systems offer services through a standard mechanism called system calls.
- When a process wants to invoke a system call, it puts the arguments to system calls in registers and calls interrupt **0x80**.
- On the **i386 architecture**, the system call number is put in the register **%eax**.
- The arguments to this system call are put into registers **%ebx, %ecx, %edx, %esi** and **%edi**, in that order, e.g.:

```
write(2, "Hello", 5)
```

roughly would translate into:

```
movl $4, %eax
movl $2, %ebx
movl $hello, %ecx
movl $5, %edx
int $0x80
```


where $hello points to a literal string “Hello”.

Ptrace Example I.


```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h> /* For constants RIG_EAX
                   etc., from kernel 2.6.25, use sys/reg.h */
int main()
{   pid_t child;
    long orig_eax;
    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "ls", NULL);
    }
else {
        wait(NULL);
        orig_eax = ptrace(PTRACE_PEEKUSER,
                    child, 4 * ORIG_EAX,
                    NULL);
        printf("The child made a "
               "system call %ld\n", orig_eax);
        ptrace(PTRACE_CONT, child, NULL, NULL);
    }
    return 0;
}
```

When run, this program prints:

The child made a system call 11

along with the output of ls (System call number 11 is execve).

Remark1: System call numbers can be found in /usr/include/asm/unistd.h
Remark2: /usr/include/linux/user.h is gone since 2.6.25. Use /usr/include/sys/reg.h

By Pradeep Padala, Linux Journal, 2002 dec.

Ptrace Example I.


```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h> /* For constants ORIG_EAX
                   etc., from kernel 2.6.25, use sys/reg.h */
int main()
{   pid_t child;
    long orig_eax;
    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL,
NULL);
        execl("/bin/ls", "ls", NULL);
    }
else {
        wait(NULL);
        orig_eax = ptrace(PTRACE_PEEKUSER,
                    child, 4 * ORIG_EAX,
                    NULL);
        printf("The child made a "
               "system call %ld\n", orig_eax);
        ptrace(PTRACE_CONT, child, NULL, NULL);
    }
    return 0;
}
```

- PTRACE_TRACEME. tells the kernel that the process is being traced, and when the child executes the execve system call, it hands over control to its parent.
- The eax register contains the system call number. We can read its value from child's USER segment by calling ptrace with PTRACE_PEEKUSER.

By Pradeep Padala, Linux Journal, 2002 dec.

Ptrace Syntax

- long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);

- The first argument determines the behaviour of ptrace and how other arguments are used. The value of request should be one of

  PTRACE_TRACEME, PTRACE_PEEKTEXT, PTRACE_PEEKDATA, PTRACE_PEEKUSER, PTRACE_POKETEXT, PTRACE_POKEDATA, PTRACE_POKEUSER, PTRACE_GETREGS, PTRACE_GETFPREGS, PTRACE_SETREGS, PTRACE_SETFPREGS, PTRACE_CONT, PTRACE_SYSCALL, PTRACE_SINGLESTEP, PTRACE_DETACH.

ptrace Example II.

Karoly.Bosa@jku.at

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h> /*from kernel 2.6.25, use
                                sys/reg.h */
#include <sys/syscall.h>   /* For SYS_write etc */
int main()
{   pid_t child;
    long orig_eax, eax;
    long params[3];
    int status;
    int insyscall = 0;
    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL,
NULL);
        execl("/bin/ls", "ls", NULL);
    }
    else {
      while(1) {
        wait(&status);
        if(WIFEXITED(status)) break;
        orig_eax = ptrace(PTRACE_PEEKUSER,
                child, 4 * ORIG_EAX, NULL);
        if(orig_eax == SYS_write) {
          if(insyscall == 0) {
            /* Syscall entry */
            insyscall = 1;
            params[0] = ptrace(PTRACE_PEEKUSER, child, 4 * EBX, NULL);
            params[1] = ptrace(PTRACE_PEEKUSER, child, 4 * ECX, NULL);
            params[2] = ptrace(PTRACE_PEEKUSER, child, 4 * EDX, NULL);
            printf("Write called with %ld, %ld, %ld\n", params[0], params[1],
                  params[2]);
          }
          else { /* Syscall exit */
            eax = ptrace(PTRACE_PEEKUSER,
                      child, 4 * EAX, NULL);
            printf("Write returned "
                    "with %ld\n\n", eax);
            insyscall = 0;
          } //else
        } //if
        ptrace(PTRACE_SYSCALL, child, NULL, NULL);
      } //while
    } //else
    return 0;
}
```

PTRACE_SYSCALL, makes the kernel stop the child process whenever a system call entry or exit is made

By Pradeep Padala, Linux Journal, 2002 dec.

ptrace Example II. Output


```
@linux:~/ptrace > ls
a.out        dummy.s      ptrace.txt
libgpm.html  registers.c  syscallparams.c
dummy        ptrace.html  simple.c

@linux:~/ptrace > ./a.out
Write called with 1, 1075154944, 48
a.out        dummy.s      ptrace.txt
Write returned with 48

Write called with 1, 1075154944, 59
libgpm.html  registers.c  syscallparams.c
Write returned with 59

Write called with 1, 1075154944, 30
dummy        ptrace.html  simple.c
Write returned with 30
```

ptrace Example III.


```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h> /*from kernel 2.6.25, use
                                                sys/reg.h and
sys/user.h */
#include <sys/syscall.h>
int main()
{   pid_t child;
    long orig_eax, eax;
    long params[3];
    int status;
    int insyscall = 0;
    struct user_regs_struct regs;
    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "ls", NULL);
    }
    else {
      while(1) {
        wait(&status);
        if(WIFEXITED(status))
            break;
        orig_eax = ptrace(PTRACE_PEEKUSER,
                    child, 4 * ORIG_EAX,  NULL);
        if(orig_eax == SYS_write) {
            if(insyscall == 0) {
              /* Syscall entry */
              insyscall = 1;
              ptrace(PTRACE_GETREGS, child, NULL, &regs);
              printf("Write called with %ld, %ld, %ld\n",
                    regs.ebx, regs.ecx, regs.edx);
            }
            else { /* Syscall exit */
              eax = ptrace(PTRACE_PEEKUSER, child, 4 * EAX, NULL);
              printf("Write returned with %ld\n", eax);
              insyscall = 0;
            }
          }
          ptrace(PTRACE_SYSCALL, child,  NULL, NULL);
        }
      }
      return 0;
}
```

- Calling ptrace with a PTRACE_GETREGS will place all the registers in a single call.
- user_regs_struct defined in <linux/user.h>

By Pradeep Padala, Linux Journal, 2002 dec.

ptrace Example IV. – 1st Part

Karoly.Bosa@jku.at

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h> /*from kernel 2.6.25, use  sys/reg.h */
#include <sys/syscall.h>
const int long_size = sizeof(long);
void reverse(char *str)
{   int i, j;
    char temp;
    for(i = 0, j = strlen(str) - 2;
        i <= j; ++i, --j) {
        temp = str[i]; str[i] = str[j]; str[j] = temp;
    }
}
void getdata(pid_t child, long addr, char *str, int len) {//… }
void putdata(pid_t child, long addr, char *str, int len) {//… }

int main()
{
  pid_t child;
  child = fork();
  if(child == 0) {
     ptrace(PTRACE_TRACEME, 0, NULL, NULL);
     execl("/bin/ls", "ls", NULL);
  }
else {
     long orig_eax;
     long params[3];
     int status;
     char *str, *laddr;
     int toggle = 0;
     while(1) {
       wait(&status);
       if(WIFEXITED(status))
          break;
       orig_eax = ptrace(PTRACE_PEEKUSER, child, 4 * ORIG_EAX, NULL);
       if(orig_eax == SYS_write) {
         if(toggle == 0) {
           toggle = 1;
           params[0] = ptrace(PTRACE_PEEKUSER, child, 4 * EBX, NULL);
           params[1] = ptrace(PTRACE_PEEKUSER, child, 4 * ECX,  NULL);
           params[2] = ptrace(PTRACE_PEEKUSER, child, 4 * EDX, NULL);
           str = (char *)malloc((params[2]+1)* sizeof(char));
           getdata(child, params[1], str, params[2]);
           reverse(str);
           putdata(child, params[1], str, params[2]);
         }
         else { toggle = 0; }
       }
     ptrace(PTRACE_SYSCALL, child, NULL, NULL);
     }} return 0; }
```


By Pradeep Padala, Linux Journal, 2002 dec.

ptrace Example IV. – 2nd Part

Karoly.Bosa@jku.at

```
void getdata(pid_t child, long addr, char *str, int len)
{  char *laddr;
   int i, j;
   union u {
          long val;
          char chars[long_size]; }data;
   i = 0;
   j = len / long_size;
   laddr = str;
   while(i < j) {
      data.val = ptrace(PTRACE_PEEKDATA, child,
                    addr + i * 4, NULL);
      memcpy(laddr, data.chars, long_size);
      ++i;
      laddr += long_size;
   }
   j = len % long_size;
   if(j != 0) {
      data.val = ptrace(PTRACE_PEEKDATA,
                    child, addr + i * 4, NULL);
      memcpy(laddr, data.chars, j);
   }
   str[len] = '\0';
}
```

```
void putdata(pid_t child, long addr, char *str, int len)
{   char *laddr;
   int i, j;
   union u {
          long val;
          char chars[long_size]; }data;
   i = 0;
   j = len / long_size;
   laddr = str;
   while(i < j) {
      memcpy(data.chars, laddr, long_size);
      ptrace(PTRACE_POKEDATA, child, addr + i * 4, data.val);
      ++i;
      laddr += long_size;
   }
   j = len % long_size;
   if(j != 0) {
      memcpy(data.chars, laddr, j);
      ptrace(PTRACE_POKEDATA, child, addr + i * 4, data.val);
   }
}
```


By Pradeep Padala, Linux Journal, 2002 dec.

Ptrace Example IV. Output


```
@linux:~/ptrace > ls
a.out        dummy.s      ptrace.txt
libgpm.html  registers.c  syscallparams.c
dummy        ptrace.html  simple.c

@linux:~/ptrace > ./a.out
txt.ecartp      s.ymmud      tuo.a
c.llacys_egnahc  c.sretsiger  lmth.mpgbil
c.elpmis        lmth.ecartp  ymmud
```

ptrace: Attaching to a Running Process I.

Karoly.Bosa@jku.at

- The following is the code for a small example tracing program:

```
int main()
{   int i;
    for(i = 0;i < 10; ++i) {
        printf("My counter: %d\n", i);
        sleep(5);
    }
    return 0;
}
```

- Save the program as dummy2.c. Compile and run it:

```
gcc -o dummy2 dummy2.c ./dummy2 &
```

- If you want to trace or debug a process already running, then ptrace(PTRACE_ATTACH, ..) should be used.

- After we are done with modifications or tracing, we can let the traced process continue on its own by calling ptrace(PTRACE_DETACH, ..).

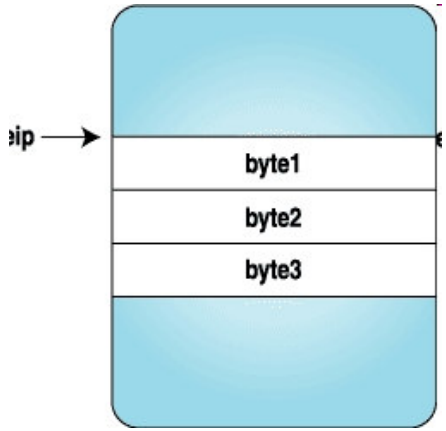
ptrace: Attaching to a Running Process II.

Karoly.Bosa@jku.at

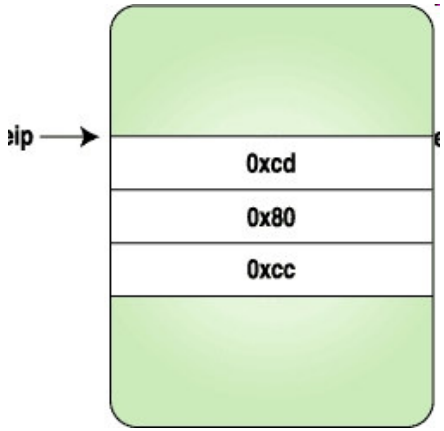
```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>   /* For user_regs_struct etc.
                             from kernel 2.6.25, use sys/user.h */
int main(int argc, char *argv[])
{   pid_t traced_process;
    struct user_regs_struct regs;
    long ins;
    if(argc != 2) {
        printf("Usage: %s <pid to be traced>\n",
               argv[0], argv[1]);
        exit(1);
    }
    traced_process = atoi(argv[1]);
    ptrace(PTRACE_ATTACH, traced_process,
          NULL, NULL);
    wait(NULL);
    ptrace(PTRACE_GETREGS, traced_process,
           NULL, &regs);
    ins = ptrace(PTRACE_PEEKTEXT, traced_process,
                 regs.eip, NULL);
    printf("EIP: %lx Instruction executed: %lx\n",
           regs.eip, ins);
    ptrace(PTRACE_DETACH, traced_process,
           NULL, NULL);
    return 0;
}
```


By Pradeep Padala, Linux Journal, 2002 dec.

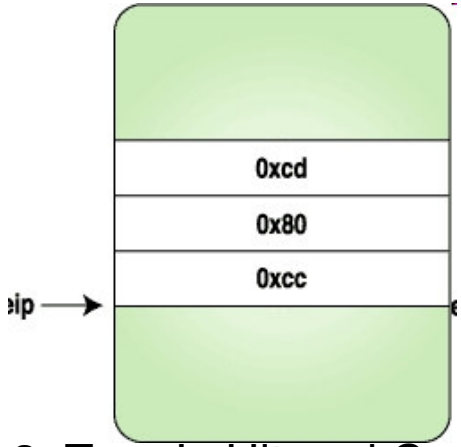
ptrace: Setting Breakpoints I.



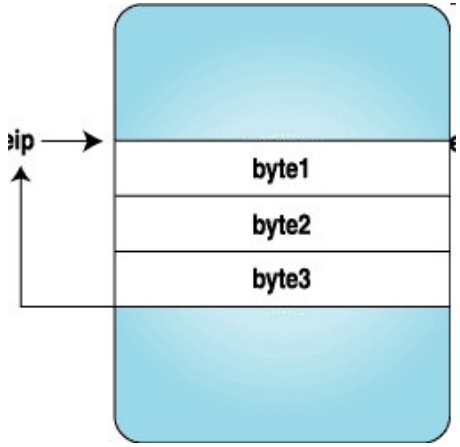
Step 1. After the Process Is Stopped



Step 2. After the Trap Instruction Bytes Are Set



Step 3. Trap Is Hit and Control Is Given to the Tracing Program



Step 4. After the Original Instructions Are Replaced and eip Is Reset to the Original Location

ptrace: Setting Breakpoints II.

Karoly.Bosa@jku.at

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h> /*from kernel 2.6.25, use  sys/user.h */
const int long_size = sizeof(long);
void getdata(pid_t child, long addr, char *str, int len) {//… }
void putdata(pid_t child, long addr, char *str, int len) {//… }

int main(int argc, char *argv[])
{   pid_t traced_process;
    struct user_regs_struct regs, newregs;
    long ins;
    /* int 0x80, int3 */
    char code[] = {0xcd,0x80,0xcc,0};
    char backup[5]; //4+1 for '\0'
    if(argc != 2) {
        printf("Usage: %s  <pid to be traced>\n", argv[0]);
        exit(1);
    }
    traced_process = atoi(argv[1]);
    ptrace(PTRACE_ATTACH, traced_process,  NULL, NULL);
    wait(NULL);
    ptrace(PTRACE_GETREGS, traced_process, NULL, &regs);
/* Copy instructions into a backup variable */
    getdata(traced_process, regs.eip, backup, 4);

    /* Put the breakpoint */
    putdata(traced_process, regs.eip, code, 4);

    /* Let the process continue and execute the int 3 instruction */
    ptrace(PTRACE_CONT, traced_process, NULL, NULL);

    wait(NULL);
    printf("The process stopped, putting back the original
instructions\n");
    printf("Press <enter> to continue\n");
    getchar();
    putdata(traced_process, regs.eip, backup, 4);

    /* Setting the eip back to the original instruction to let the
process continue */
    ptrace(PTRACE_SETREGS, traced_process,  NULL, &regs);
    ptrace(PTRACE_DETACH, traced_process,  NULL, NULL);
    return 0;
}
```


By Pradeep Padala, Linux Journal, 2002 dec.

ptrace: Setting Breakpoints III.


```
void getdata(pid_t child, long addr, char *str, int len)
{   char *laddr;
    int i, j;
    union u {
            long val;
            char chars[long_size];
    }data;
    i = 0;
    j = len / long_size;
    laddr = str;
    while(i < j) {
        data.val = ptrace(PTRACE_PEEKDATA, child,
                    addr + i * 4, NULL);
        memcpy(laddr, data.chars, long_size);
        ++i;
        laddr += long_size;
    }
    j = len % long_size;
    if(j != 0) {
        data.val = ptrace(PTRACE_PEEKDATA, child,
                    addr + i * 4, NULL);
        memcpy(laddr, data.chars, j);
    }
    str[len] = '\0';
}
```

```
void putdata(pid_t child, long addr, char *str, int len)
{   char *laddr;
    int i, j;
    union u {
            long val;
            char chars[long_size];
    }data;
    i = 0;
    j = len / long_size;
    laddr = str;
    while(i < j) {
        memcpy(data.chars, laddr, long_size);
        ptrace(PTRACE_POKEDATA, child,
                addr + i * 4, data.val);
        ++i;
        laddr += long_size;
    }
    j = len % long_size;
    if(j != 0) {
        memcpy(data.chars, laddr, j);
        ptrace(PTRACE_POKEDATA, child,
                addr + i * 4, data.val);
    }
}
```


By Pradeep Padala, Linux Journal, 2002 dec.