In this exercise, we are going to implement a strace-inspired debugger, based on ptrace.

1. Create a file (e.g., snyfer.c) for our code:

   We need to include a number of headers to get access to ptrace and supplementary facilities like
I/O, system call symbols, and data structures for communicating with the kernel about saved
registers. We also need to define the GNU_SOURCE symbol because we use the GNU/Linux
specific function `strsignal(3)'

   #define GNU_SOURCE
   #include <stdio.h>
   #include <stdlib.h>
   #include <string.h> //memset, strncmp, strsignal
   #include <unistd.h>
   #include <sys/wait.h>
   #include <sys/ptrace.h>
   #include <sys/types.h>
   #include <sys/syscall.h>
   #include <sys/user.h>
   #include <asm/ptrace-abi.h>
   #include <asm/ptrace.h>

2. Now, add some more global state and symbols:

   #define LINELEN 81
   extern char* __progname; //a GNU symbol
   /** The process ID to trace. */
   long int tr_pid = 0;
   /** pointer to user-entered string */
   char* usercmd = NULL;

3. The next step is defining function signatures. These are usually defined in a header file. We use
the “static” keyword as there is no reason for them to be invoked outside this module/program.

   static void do_sniff();
   static void do_usage(void);
   static void handle_syscall(int*);
   static void init_attach(char*);
   static void create_interpreter(void);
   static void quit(void);
   static void kill_trace_session(void);
   static void handle_continue(int*, int*);

   Design note: the system starts up, calls init_attach(), calls create_interpreter() and then
invokes \texttt{do_sniff()}. This function loops around a call to \texttt{ptrace()} within \texttt{handle_continue()} and occasionally calls \texttt{handle_syscall()}.

4. The most(only) interesting point of \texttt{do_usage()} is that it shows how we can use the \texttt{\_\_progname} symbol instead of manipulating \texttt{argv[0]}, which is poor form.

\begin{verbatim}
static void
do_usage()
{
    fprintf(stderr,
        "Usage: %s -p [pid]\n",
        \_\_progname);
    return;
}
\end{verbatim}

5. The \texttt{create_interpreter()} function allocates memory to hold the input we will receive from the user on stdin.

\begin{verbatim}
static void
create_interpreter()
{
    usercmd = calloc(LINELEN, sizeof(char));
    if(NULL==usercmd)
    {
        fprintf(stderr,
            "failed to allocate memory for user command line\n");
        exit(-3);
    }
    return;
}
\end{verbatim}

6. The \texttt{init_attach()} function parses its parameter as a number (i.e., a process ID). It then asks the kernel (via ptrace) to register it as the target processes parent, which effectively means that this process (\texttt{snyfer}) wants to receive signals about what happens in the child/traced process.

\begin{verbatim}
static void
init_attach(char* tpid)
{
    int s = 0;
    long p_ret = 0;
    pid_t p = 0;
    int attach_status = 0;
    tr_pid = strtol(tpid, NULL, 10);
    fprintf(stdout,
        "[snyfer] tracing process %ld\n",
        tr_pid);
    p_ret = ptrace(PTRACE_ATTACH,
        tr_pid,
        NULL, //ignored
        NULL); //ignored
    if(-1==p_ret)
    {
        
CPSC 457 – Tutorial 16 March 20, 2012 2/8
fprintf(stderr, 
  "[snyfer] failed to attach to child, exiting...
")
  exit(-1);
}
p = waitpid(tr_pid, 
  &attach_status, 
  WUNTRACED | WCONTINUED);
if(WIFSTOPPED(attach_status))
{
  s = WSTOPSIG(attach_status);
  fprintf(stdout, 
    "Successful attach. Child stopped by signal %d: %s\n", 
    s, 
    strsignal(s));
} else{
  fprintf(stdout, 
    "failed to stop / attach target process\n");
  exit(-2);
}
return;

Note the use of the
  waitpid() system call so that the new parent (i.e., us) can synchronize 
  execution with the child's state. Also, we use
  fprintf() for printing messages to standard output 
  or standard error output, as necessary.

7. Now, the next function will demonstrate the use of different ptrace requests to extract data from 
   the user meta-data memory area in ptrace and the request to read the register set.

/** This function assumes everything else is set up properly. */
static void
handle_syscall(int* syscall_started)
{
  long int oeax = 0;
  long int eax = 0;
  struct user_regs_struct regfile;
  oeax = ptrace(PTRACE_PEEKUSER, 
                tr_pid, 
                4 * ORIG_EAX, 
                NULL);
  if(SYS_write==oeax)
  {
    if(0==*syscall_started)
    {
      *syscall_started = 1;
      //print registers
      ptrace(PTRACE_GETREGS, 
             tr_pid, 
             NULL, 
             &regfile);
      fprintf(stdout, 
        "process issued syscall: write(%ld, %ld, %ld)\n", 
        regfile.ebx, 
        regfile.ecx,
Note: In this code, we only use ptrace to read the other program's memory and registers. There are also ptrace requests to write to both (memory and registers). Naturally, this means you can change both data (for example, data used in making control flow decisions) and registers (for example, used in storing stack state or system call arguments, as well as pretty much everything else). Also notice the use of the SYS_write symbol referring to the write(2) system call.

8. We can define a subroutine for issuing a SIGKILL via ptrace. Note that doing so doesn't kill our program, merely the child. Of course, we have nothing else to do but twiddle our thumbs now, so we call exit(0) and quickly terminate as well.

```c
static void
kill_trace_session()
{
    //send child PTRACE_KILL
    fprintf(stdout,
            "killing child and exiting...\n\n");
    ptrace(PTRACE_KILL,
            tr_pid,
            NULL,
            NULL);
    free(usercmd);
    usercmd = NULL;
    exit(0);
}
```

9. Calling quit() allows us to terminate while letting the child continue because we requested PTRACE_DETACH via ptrace(2).

```c
static void
quit()
{
    //cleanup and quit
    fprintf(stdout,
            "Tracer will quit. Traced process %ld will continue running.\n",
            tr_pid);
    ```
Notice both here and above that we are returning the usercmd memory chunk to glibc before exiting. Nice of us, but not strictly necessary because the entire process will be cleaned up very soon -- but still a good habit to form. Keeping track of allocation and de-allocation sites is tricky, particularly across multiple components, scopes, and layers of abstraction. A very nice source of bugs.

10. The next function asks the child program to continue its execution, until it encounters a system call. When the child process stops, we do some checkings on the signal, to see if the program was not killed and if it did not finished.

```c
static void handle_continue(int* cont,
    int* syscall_started)
{
    long p_ret = 0;
    pid_t p = 0;
    int attach_status = 0;
    int s = 0;
    //continue to the next system call (or end of this one)
    p_ret = ptrace(PTRACE_SYSCALL,
        tr_pid,
        NULL,
        NULL);
    p = waitpid(tr_pid,
        &attach_status,
        WUNTRACED | WCONTINUED);
    if(-1==p)
    {
        perror("waitpid");
        exit(-5);
    }
    if(WIFEXITED(attach_status))
    {
        fprintf(stdout, "exited, status=%d\n", WEXITSTATUS(attach_status));
    }else if (WIFSIGNALED(attach_status)){
        s = WTERMSIG(attach_status);
        fprintf(stdout,"killed by signal %d : %s\n",
            s,
            strsignal(s));
    }else if (WIFSTOPPED(attach_status)){
        s = WSTOPSIG(attach_status);
        fprintf(stdout,"stopped by signal %d : %s\n",
            s,
            strsignal(s));
        if(SIGTRAP==WSTOPSIG(attach_status) ||
```
```c
SIGSTOP==WSTOPSIG(attach_status))
{
    handle_syscall(syscall_started);
} else{
    fprintf(stderr, "unrecognized signal state for stopped @ syscall\n");
}
else if (WIFCONTINUED(attach_status)) {
    fprintf(stdout,"continued\n");
} else{
    fprintf(stdout, "unrecognized stop condition\n");
}
if(!WIFEXITED(attach_status) && !WIFSIGNALED(attach_status))
{
    *cont = 1;
} else{
    fprintf(stdout, "setting stop flag\n");
    *cont = 0;
}
return;
}

11. The next function is responsible for user interaction It also kicks off `handle_continue()` and the system call handler in turn.

static void
do_sniff()
{
    int should_continue = 1; // TRUE
    int syscall_started = 0; // FALSE
    do
    {
        fprintf(stdout,
            "snyfer> ");
        usercmd = fgets(usercmd, LINELEN, stdin);
        if(NULL == usercmd)
        {
            fprintf(stderr,
                "problem reading input\n");
            exit(-4);
        }
        if(0==strncmp(usercmd, "quit", 4) ||
            0==strncmp(usercmd, "exit", 4)){
            quit();
        } else if(0==strncmp(usercmd, "run", 3)){
            fprintf(stdout, "not implemented\n");
        } else if(0==strncmp(usercmd, "kill", 4)){
            kill_trace_session();
        } else if(0==strncmp(usercmd, "cont", 4)){
            handle_continue(&should_continue,
            &syscall_started);
        } else{
            fprintf(stdout, "snyfer did not recognize your command\n");
        }
        memset(usercmd, '\0', LINELEN);
    } while(1==should_continue);
```
return;
}

12. And finally a simple main, which sets things up and then hands off control to `do_sniff()`.

```c
/**
 * ./snyfer -p [PID]
 * opens an interactive session where you can re-write arguments to
 * system calls of the traced process. Its default mode is simply to
 * print the syscall number and arg values.
 * *
 */
int main(int argc,
    char* argv[])
{
    if(3==argc)
    {
        init_attach(argv[2]);
        create_interpreter();
        do_sniff();
    }else{
        do_usage();
        return -1;
    }
    return 0;
}
```

13. Now, it's your turn. This program can be easily compiled directly by using gcc (there is not many files, for example). But, for good practice, write the `Makefile` to compile it. Don't forget to clean up after yourself.

14. Let's improve this code. Modify the program to handle other system calls, e.g., `sys_read`, `sys_open`, `sys_close`. 
SAMPLE OUTPUT

Our program snyfer has the following sample output:

```
[user@host ptrace]$ ./snyfer -p 1234
[snyfer] tracing process 1234
Successful attach. Child stopped by signal 19 : Stopped (signal)
snyfer> run
not implemented
snyfer> cont
stopped by signal 5 : Trace/breakpoint trap
process issued syscall: write(1, -1216974841, 23)
snyfer> cont
stopped by signal 5 : Trace/breakpoint trap
system call returned 23
snyfer> cont
stopped by signal 5 : Trace/breakpoint trap
process issued syscall: write(1, -1216974848, 30)
snyfer> cont
stopped by signal 5 : Trace/breakpoint trap
system call returned 30
snyfer> kill
killing child and exiting...
[user@host ptrace]$
```