A Multi-Platform Pseudo Terminal API

Project Report
Submitted in Partial Fulfillment for
the Masters' Degree in Computer Science

By Qutaiba Mahmoud

Supervised By
Dr. Clinton Jeffery

ABSTRACT

This project is the construction of a pseudo-terminal API, which will provide a pseudo-terminal interface access to interactive programs. The API is aimed at developing an extension to the Unicon language to allow Unicon programs to easily utilize applications that require user interaction via a terminal. A pseudo-terminal is a pair of virtual devices that provide a bidirectional communication channel. This project was constructed to enable an enhancement to a collaborative virtual environment, because it will allow external tools such to be utilized within the same environment. In general the purpose of this API is to allow the UNICON runtime system to act as the user via the terminal which is provided by the API, the terminal is in turn connected to a client process such as a compiler, debugger, or an editor. It can also be viewed as a way for the UNICON environment to control and customize the input and output of external programs.

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003

Table of Contents:

Acknowledgments

I would like to thank my advisor, Dr. Clinton Jeffery, for his support, patience and understanding. Dr. Jeffery has always been prompt in delivering and sharing his knowledge and in providing his assistance.

1.0 Introduction

This work is part of a big project that is creating a virtual model of the NMSU computer science department, within the science hall building. Eventually a user will be able to log into their computer from any where and be able to interact with other users who are logged into the same space, and have access to all educational facilities provided by the department to the students, or a user can interact with other people physically present inside the building as if the user is attending a class or a meeting inside the building while its ongoing, or a user can even navigate inside the building and be able to see who is there in real time. One part of enhancing the CVE is to provide access to the educational tools available, so that the users can take advantage of the available programs. Therefore there was a need to design an API to enable the user to access and use those application programs. In this project, the API must be capable of controlling the input and the output for a single or multiple processes, with the help of pseudo terminals. For purposes of this project one of these processes is going to be the debugger GDB, this way a user will be able to access and interact with a GDB session in order to be able to debug a program and easily share the debugging session with other users, therefore enhancing the overall CVE by adding more capabilities.

1.1 Pseudo Terminals

Pseudo terminals were invented in 1983 for Berkeley software distribution (BSD) of UNIX. AT&T's System V included support for pseudo terminals as a driver in their STREAMS device model, along with the pseudo terminal multiplexer (Wikipedia). BSD is one of several branches of UNIX operating systems. Another one is evolved from UNIX system V developed by AT&T's Unix System Development Labs (Pate, 2003).

The term pseudo-terminal implies that it looks like a terminal to an application program, but it's not a real terminal (Stevens, 1992). Pseudo-terminals are pseudo-device pairs that provides a text terminal interface without associated virtual console (figure1), computer terminal or serial port hardware, instead a special interprocess communication channel acts like a terminal, one end of the channel is called the master side or master pseudo-terminal device, the other side is called the slave side, while replacing the role of the underlying hardware for the pseudo terminal session (“*The GNU C Library Reference Manual”*, 2001).

Figure 1: Simple breakdown of how pseudo terminals work

The way it works is that data written on the slave side of a pseudo terminal is supplied as input to a process reading from the master side. Data written on the master side is given to the slave as input. In this way, the process manipulating the master side of the pseudo terminal has control over the information read and written on the slave side, where information can be changed or formatted. Anything written on the master device is given to the slave device as input and anything written on the slave device is presented as input on the master device. This is different than the one way communication, where data flows in one direction, and the system offers no data interruption.

To allocate a pseudo-terminal, and for making this pseudo-terminal available for actual use many functions can be used, for instance you can use the function `getpt(),` the `getpt()` function returns a new file descriptor for the next available master pseudo-terminal, the normal return value from `getpt()` is a non-negative integer file descriptor, in the case of an error, a value of (-1) is returned instead. If a user wants to open both sides of a pseudo-terminal in a single operation, the function `openpty()` can be used, this function allocates and opens a pseudo-terminal pair, returning the file descriptor for the master in (*amaster), and the file descriptor for the slave in (*aslave)

Pseudo-terminals can be created when a process opens (/dev/ptmx), it gets a file descriptor for a pseudo-terminal master (PTM), and a pseudo-terminal slave (PTS) device is created in the (/dev/pts) directory. Each file descriptor obtained by opening (/dev/ptmx) is an independent PTM with its own associated PTS, once both the pseudo-terminal master and slave are open, the slave provides processes with an interface that is identical to that of a real terminal.

1.2 OTHER TERMINALS

A terminal is different from a computer, because it does not provide any processing of information. As in legacy computer systems where a terminal is connected to a computer, the function of the terminal is to send commands to the processing computer; therefore a terminal consists of just a keyboard and a monitor. The notion of a terminal in new computers is a place where a user can type commands in an interface designed for text entry, therefore the user should be familiar with some UNIX or DOS commands, because the interface is usually not graphical. This interface is a terminal program which is usually included with the operating system, and available in many computing platforms. For example, one can use a terminal program within the Macintosh OSX operating system to run UNIX commands or access other machines.

Since most computers have the basic components of a terminal, you can make a computer act or do what a text terminal do, in other words you can make the computer emulate a terminal. In emulation, one of the serial ports of the computer will be used to connect the emulated terminal to another computer, either with a direct cable connection from serial port to serial port, or via a modem. Emulation enables more than just a terminal since the PC doing the emulation can also do other tasks at the same time it's emulating a terminal. For example, a software may be run on the computer to enable transfer of files over the serial line to the other computer that you are connected to. Another type of terminal emulation is where you set up a real terminal to emulate another brand of terminal. To do this you can select the emulation you want from the terminal's set-up menu.

1.3 RELATION TO OTHER PSEUDO TERMINAL AP PLICATIONS

There are many applications of pseudo-terminals, such as Xterm ; a terminal emulator for the X Window System, in which the terminal emulator process is associated with the master device and the shell is associated with the slave. Other applications include remote log in handlers such as Telnet and SSH, where users can access remote computers as if they were on that remote machine itself. There are many other programs that use pseudo terminals like EXPECT (Libes, 1994). It utilizes a terminal interface for programming interactions with another program and enhances it with command line capability. It allows the user to write scripts that responds to the program the user is interacting with. But the user will have to know ahead of time what output the program might send, in order to write the proper script for it. EXPECT could be described as a programmable interaction that could be interrupted by the user without stopping the controlled program.
This report presents the design and implementation of an API that will extend the UNICON language by providing a pseudo-terminal feature. This feature will be utilized as an interface access to running processes by UNICON.

This work is going into a general-purpose UNICON Language feature which in turn will be used in the Virtual Environment. It builds a link between a program such as GDB or VI, and the virtual environment. It will also be used in many other applications.

2.0 Methodology

The project was developed on an OPTERON 150 processor 2.4 GHz, 1 GB RAM, running 64 bit Linux. The version of the C compiler used to build the project is GCC version 4.1.0.

In order to create the proper procedure for this API, in basic terms we will need two end points, one to access the server, and the other to access the client.
A test application utilizing this API was developed. It uses pseudo-terminals as an interface access to GDB, so that the user is interacting with GDB through another program that is controlling the GDB. The testing application will control the GDB program by writing strings to it that look like GDB command lines, read GDB's output, and display it within a graphical user interface. This will happen with the ability to keep the same output or change it, or even send it to a file. The program is basically running between the user and the GDB debugger. It will have the ability to replace the GDB prompt with some other prompt, and it can send additional commands to the GDB.
When utilizing the debugger GDB, users will have the ability to run and debug their programs, this is because GDB can provide information of what is taking place inside another program while it executes. This will save time in the software development cycle because it will be easy to find where the errors are and focus on fixing them. There are many advantages to GDB. The most obvious one is that it is free, and it is considered to be a well-known debugger in many Unix and Linux environments.

Development of this project required many iterations. In the early stages of developing the API, a prototype looked like it was working, but all the GDB I/O was dealing directly with the real console and not going through the pseudo-terminal. The API was not in control of the I/O, instead it was using the read and write functions to interact with the terminal and to capture user input and to display the GDB output.

One attempt implemented some Unix functions that created pipes, which worked well for some applications, particularly for reading or writing from the user, however it could only do one or the other at a specific time, but not both. The problem with `pipe()` is that it buffers data and allows the processes to deadlock each other, which didn’t work for this application. The pipe function is also limited in its use because it doesn’t conform to POSIX standard which created a portability problem.

The function `fork()` was used many times at the beginning, which creates a child process that differs from the parent process. The reasoning behind it was to create a GDB master and slave processes, but the use of `fork()` was a problem. Uunder Linux, the function `fork()` is implemented using copy-on-write pages, so the only penalty incurred by fork is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child, and that made the program slower. A bigger problem was that `fork()` was not portable for different platforms.

Also the function `getpt()` was used to accomplish the task of opening the master pseudo terminal. This function worked well but it was not portable, because it is specific to the GNU C library. Therefore `getpt()` was replaced with the function `POSIX_openpt`, which was more specific to the Unix98 pseudo_terminal support.

To complete the pseudo-terminal creation process, the functions `grantpt(), unlockpt(),` and `ptsname_r()` are also used. The `grantpt()` function changes the file permission for the slave pty device to match that of the master pty device. The `unlockpt()` function simply unlocks the slave pty, enabling the master/slave pair. Finally, the `ptsname_r()` function returns the filename of the slave pty associated with the master/slave pty pair. However, there are two functions that perform this specific task. The `ptsname()` function returns a statically allocated string containing the slave filename. Since the API uses `fork()` it can be reasonably expected that static variables are not safe, since race conditions may occur between the parent process, the controller process, and the child process, which eventually becomes the image of an external program such as GDB. The second, is the aforementioned `ptsname_r()` function is reentrant and consequently creates a copy of the string containing the slave filename, as opposed to a static one, therefore eliminating race conditions.

In order to create a copy of the file descriptor `oldfd()`, the functions `dup()` and `dup2()` were used . Function `dup()` was implemented to get control of an interactive session. And function `dup2()`is used in place of `dup()` to automatically copy over the new file descriptor into the old file descriptor. If for any reason the dup function failed then the child process was exited with an error.

In pseudo-terminals, the function `select()` is used to poll the input or output device of the slave to see if it is ready for reading or writing, to capture real time input/output, or as close to real time as possible. Then the real time value for `select()` is set to zero. That way if I/O is not immediately available then the program can continue without blocking. However, if it is available the program does the following: if the program is reading, it repeats the process of polling the slave device and writing a single character until the entire input buffer has been read. If it is writing, then it polls the slave once and then attempts to write the entire buffer all at once. There should be a way to wait for the slave to become ready for output. This is because if you don’t wait, the slave end may not have enough time to write out all of the messages the master

end is waiting for. The API can be changed so it has inherently zero time out or you can specify a value for a time out. However, the test program designed for testing the functionality of this API waits for about one second. It could be possible to reduce the waiting time below the one second mark, but one second seemed to be enough time for the slave end to finish with its output. Using the sleep function in order to eliminate waiting at all resulted in the suspension of the test program long enough for the slave to deliver all of its output. However, that took about six to eight seconds. Using the function `sched_yield()` yielded the processor by skipping the process for the first time and going to the next process in line. Doing this didn't result in consistent reliable wait period, because it depended on the number of processes running at the time. Therefore the time out value in the `select()` function provided the best results for efficiency and reliability.

2.1 PSEUDO TERMINAL API FUNCTIONS DESCRIPTION

ptopen() `struct ptstruct *ptopen(const char *);`

It allocates memory for a new ptstruct, initializes all data structure element, it also forks and executes the slave process.

INPUTS:

char * = command for child to execute after forking

OUTPUT:

returns the newly allocated structure if no errors occurred, it returns NULL if any errors occurred

ptclose() `void ptclose(struct ptstruct *);`

It closes all pty file descriptors associated with the ptstruct and then frees the memory allocated for the structure

INPUTS:

struct ptstruct* = pointer to ptstruct to close

OUTPUT:

returns void

ptgetstr() `int ptgetstr(char *, const int, struct ptstruct*);`

It checks if input file descriptor is ready and than stores output into the buffer passed as an argument, it also stops reading input when a new line is read.

INPUTS:

char* = storage for output from input file descriptor

int = maximum size of the buffer

struct ptstruct* = pointer to a ptstruct to read from

OUTPUT:

returns num of bytes read if characters read from input file descriptor

return -1 if error occurs

return 0 timeout

ptlongread() `int ptlongread(char *, const int, struct ptstruct*);`

It checks if input file descriptor is ready and than stores output into the buffer passed as an argument. It attempts to read number bytes passed as an argument.

INPUTS:

char* = storage for output from input file descriptor

int = number of characters to read from input

struct ptstruct* = pointer to a ptstruct to read from

OUTPUT:

returns num of bytes read if characters read from input file descriptor

return -1 if error occurs

return 0 if timeout occurs

ptgetstrt() `int ptgetstrt(char *, const int, struct ptstruct*, unsigned long, int);`

It waits for an amount of milliseconds passed as an argument for input buffer to become ready ,and than stores output into the buffer passed as an argument.

INPUTS:

char * = pointer to buffer to write to output file descriptor

int = length of buffer

struct ptstruct* = pointer to a ptstruct to write to

unsigned long = number of microseconds to wait before timeout

int = 0 than read up to a new line, or 1 than read as many bytes as possible

OUTPUT:

returns num of bytes written if character written to output file descriptor

return -1 if error occurs
return 0 if timeout occurs

ptputstr() `int ptputstr(struct ptstruct *, const char *, const int);`

It checks if output file descriptor is ready, and then stores input from the buffer passed as an argument into the output file.

INPUTS:

struct ptstruct* = pointer to a ptstruct to write to
char * = pointer to buffer to write to output file descriptor
int = length of buffer

OUTPUT:

returns num of bytes written if character written to output file descriptor
return -1 if error occurs
return 0 if timeout occurs

ptputc() `int ptputc(const char, struct ptstruct *);`

It waits for an output file descriptor to become ready and then writes single character to output.

INPUTS:

struct ptstruct* = pointer to a ptstruct to write to
char = character to write to output file descriptor

OUTPUT:

returns 1 if success
returns -1 if error occurs
returns 0 timeout

printptstruct() `void printptstruct(struct ptstruct *);`

It prints values of the ptstruct for debugging purposes.

INPUTS:

struct ptstruct * = pointer to a ptstruct to print

OUTPUT:

returns void

3.0 Results

The API attaches the slave TTY to the new executable, such as GDB or VI in the UNIX implementation. The same idea is also followed in the Windows implementation by looking for a specific string and modifying it with the user's input. Any main differences in the API design will be discussed in the Windows section. All input and output from the new executable is sent to the slave terminal, instead of STDIN STDOUT and STDERR. This allows a master process to control the input and output of the new executable, it also allows the master process to have asynchronous input/output with the slave, so that the program will not block if the input and output from the slave are not available. Although the API does provide synchronous input and output via a time out parameter, the API gives the programmer fundamental read and write functions to communicate with the slave. It also handles the memory allocation deallocation of the PTY data structure.

3.1 UNIX Implementation.

In the UNIX implementation, The API is capable of correctly writing input to and reading output from a specified slave process, which is verified by a test program. The test took as input the name of some other executable, referred to henceforth as the argument, and opened up two slave sessions, one running GDB and the other VI. The test program successfully ran the argument within the GDB session and wrote the argument's consequent call stack into the VI session. It began by writing "file <argument>", "run", and "where" into the GDB session. Then, it wrote "i" into the VI session. It then read the output from the GDB session into a buffer and then wrote that buffer into the VI session to place it in insert mode. It finished by writing "<esc>", ":w! out.txt", and ":q!" into the VI session. The test program had complete control of all input and output from GDB and VI, therefore providing the desired pseudo terminal interface.

Within the test program a while loop was implemented with two different conditions, each used in separate trial run, the first one if there was no time out and the other case if there was a time out. For both cases, the test program tests the return value of the API read function to return with either an error or a not ready value

The first condition was made to achieve strictly asynchronous input and output, using a read and write with a timeout parameter of zero, as the following code shows. That was not realized because not enough time was given to the slave process and to write all of its output to the slave TTY. The following code will produce this output.

Code:

```
while((gdb_bytes_read=ptgetstrt(vi_buffer,sizeof(vi_buffer)
,pty1,0,0))
```

Output:

```
file hello
run
where
```

The second condition implemented an alternative asynchronous input output, when using synchronous input and output by using the time out as the following code shows, a more acceptable out come was achieved.

Code:

```
while((gdb_bytes_read=ptgetstrt(vi_buffer,sizeof(vi_buffer)
,pty1,1e6,0)) >= 0 )
```

Output:

```
file hello
run
where
GNU gdb 6.4
Copyright 2005 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-suse-linux".
(gdb) file hello
Reading symbols from /home/ugrad12/qmahmoud/cs598/hello...done.
Using host libthread_db library "/lib64/libthread_db.so.1".
(gdb) run
Starting program: /home/ugrad12/qmahmoud/cs598/hello
[tcsetpgrp failed in terminal_inferior: Inappropriate ioctl for device]
[tcsetpgrp failed in terminal_inferior: Inappropriate ioctl for device]
[tcsetpgrp failed in terminal_inferior: Inappropriate ioctl for device]
Hello

Program exited normally.
(gdb) where
No stack.
(gdb)
```

It is only possible to tell that no data from the slave is available at a specific moment in time, therefore a decision has to be made whither to go on or keep waiting. If the API was implemented to go on, then there is no way to know when exactly the output from slave is ready. If the API

implemented did not go on, then a time out parameter will need to be specified, in this case a time out of on second was used, which lead to a more complete output from the slave TTY. This is based upon the limitation of using the system call `select()` because `select()` will either timeout or return with an error. According to the IEEE and POSIX standard 1003.1, the `select()` function returns with an error if one of its parameters was not valid , except for the case of an interrupt. Therefore if everything in PTY data structure is setup correctly, `select()` should only return an undesirable value if there is an interrupt error or if it times out. We don't have any other information about the state of the slave TTY during a read. This is really is not a true limitation of `select()` because in order to know about the state of a slave TTY will involve guessing.

Since the API allocates memory for the data structure using free store, some checks have to be in place to insure the `create()` and `close()` functions are not misused. One check the API implements is to hide the definition of the data structure itself from the programmer. This prohibits the user from reallocating any elements in the data structure. The API also performs the obligatory NULL checks for pointers to PTY data structures passed as arguments to the API functions. However, this does not prohibit a user from declaring a PTY data structure and allocating it outside of an API function, and then passing that misallocated pointer into an API function.

3.2 Windows Implementation.

In the Windows part, the API attaches the pipes to a child process, in this case, the executable child.exe. All input and output is captured by the pipes instead of standard I/O, which allowed the master process to have synchronous I/O with the slave. The second test program was designed to fit the Windows platform environment. The test program reads 4 lines from an external program child.exe, searches for different words output by that program, and replaces them with different words in order to show that the I/O was fully captured by the pseudo terminal structure.
The windows version of the API and the test and the external program child.exe were all developed using Visual studio from Microsoft. The solution was made up of three different projects which are the ptstruct library it self, the test program, and the child.exe program. The ptstruct liberary was created as a static windows library with the extension .lib. The Windows code was separated from the UNIX code using the preprocessor directive #ifdef WINDOWS.
To provide windows functionality, the API uses `createPipe(), WriteFile(),` and `readFile()` to simulate the pseudo terminal. However the `WriteFile()` and `ReadFile()` functions block therefore making the API not asynchronous. This poses a limitation for the Windows API and makes it inferior to the LINUX version of the API. The API attempts to provide asynchronous behavior by implementing the function `WaitForSingleObject(),` but this function only works for certain objects such as Events, Mutexes, Semaphores, threads, or processes. Although it is different than the function `Select(),` the `WaitForSingleObject()` still offers functionality by waiting for the file handle to be in a signaled state or for a timeout value to expire before returning. Before each read and write the

windows version of the API calls `ZeroMemory(),` to clear the buffer by filling in the buffer with the null character. If this is not done, then the buffer reads in all the garbage characters still left from memory. The two figures show that the output of the child.exe when it runs on its own, and the output of the test program that controls the input and output of the child.exe program.

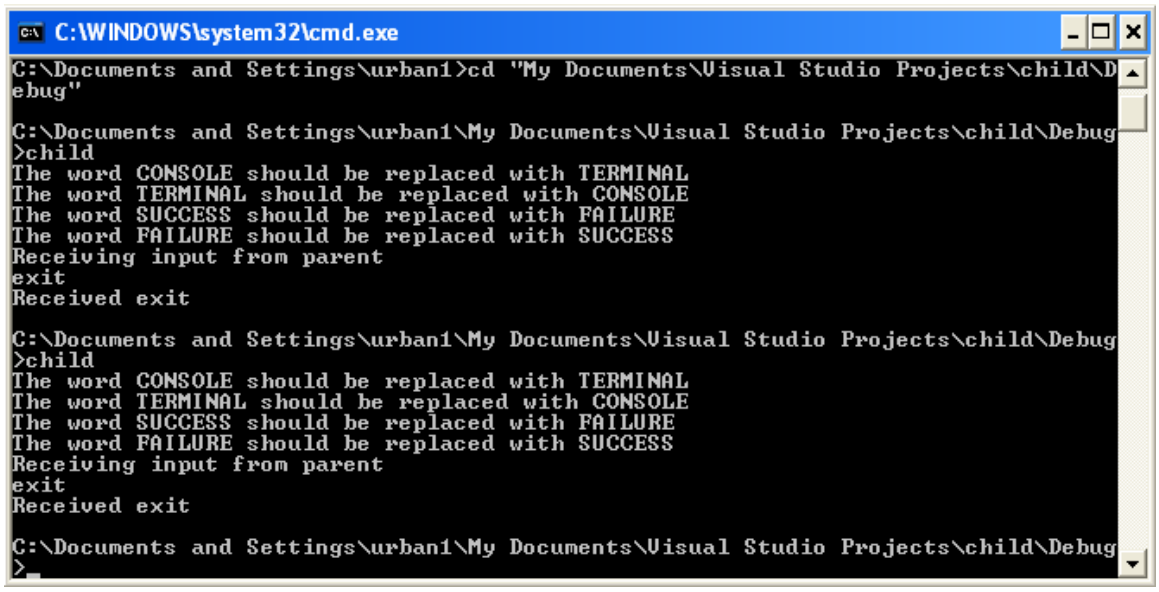

Figure 4: The external program "child.exe"

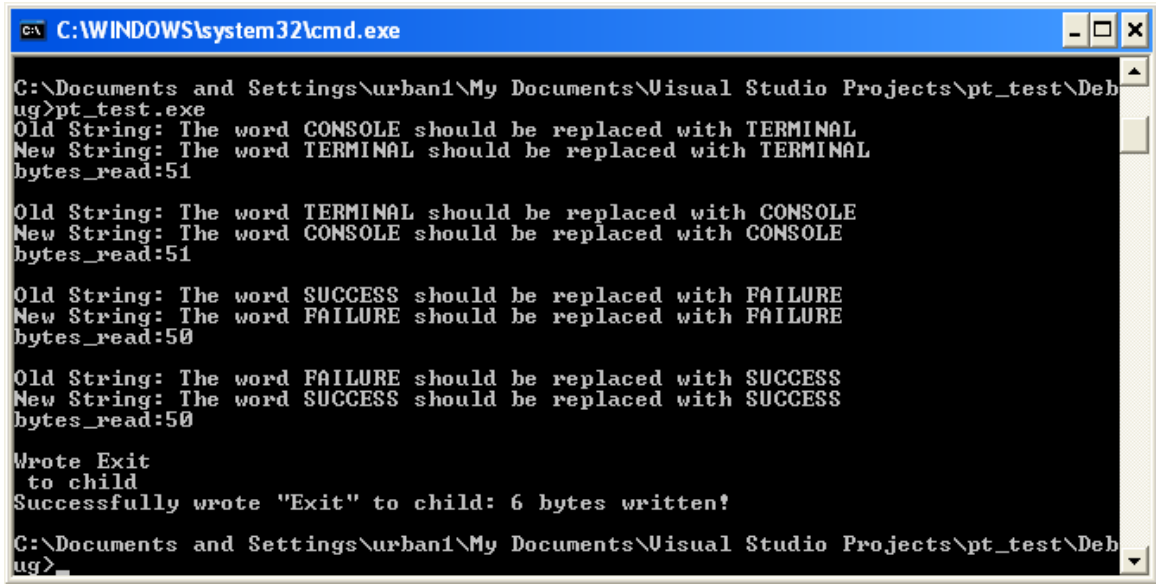

Figure 5: The output of the test program "pt_test.exe" controlling the I/O for the "child.exe"

4.0 Conclusion

The project implemented a pseudo-terminal functionality for use by the Unicon virtual machine. It can be used to capture the input and output of any external text-based program. This can effectively merge an external program into any other program that uses this API.
The API provides only a fundamental read and write to a pseudo terminal interface. It encapsulates the system calls required to bind a slave pseudo terminal device with a slave process.

5.0 Recommendations

What would be beneficial would be methods for buffered input and output. By buffering reads and writes, the program could implement the synchronous input and output, but wait to actually read from or write to a slave pseudo terminal device until the buffer is full.
The API could also benefit from a set of methods that could generate and parse user generated commands or macros, much in the same as a UNIX shell. These new methods would provide a means to parse user input for the definitions and the issuance of these new commands, and then disseminate the commands to the slave processes according to the user definitions.
As mentioned earlier, this project is one part of the CVE (Collaborative Virtual Environment) project ultimately designed for the CS department at NMSU. The pseudo terminal capabilities of this project are not limited in their scope. For the developmental and testing purposes, this API was designed to work with GDB/VI only. Nevertheless, it can be expanded to include almost any process or service provided by the educational institute. For instance, you would be able to use this API to connect to, access, and use programs such as those used in the CVE.

References:

Jeffery, C., Dabholkar, A., Tachtevrenidis, K., Kim, Y. (2005). A Framework for Prototyping Collaborative Virtual Environments. *CRIWG,* Retrieved January 21, 2006 from http://www.cs.nmsu.edu/~jeffery/vcsc/vcsc.pdf

Lawyer D. S. (August, 2006). Text-Terminal-HOWTO. Retrieved October 1st, 2006, from http://www.tldp.org/HOWTO/Text-Terminal-HOWTO.html#toc1

Stevens, W. R. (1992). Advanced Programming in the Unix Environment. Boston, Addison-Wesley Professional.

Pate S. D. (2003). UNIX Filesystems: Evolution, Design, and Implementation. Indianapolis, Wiley Publishing Inc.

The GNU C Library Reference Manual (2003). Retrieved November 17th, 2006, from http://www.gnu.org/software/libc/manual/html_node/Pseudo_002dTerminals.html

Libes, D. (1994). Exploring Expect. Sebastopol, CA. O'Reilly Media