

Lab 3

Process Management

ITSC205: Operating Systems Internals

NAME: \_\_\_\_Coleton Sanheim\_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

[Lab Outcome(s) 2](#_Toc472385055)

[Reading 2](#_Toc472385056)

[Introduction 2](#_Toc472385057)

[1.0 Trace Processes 3](#_Toc472385058)

[2.0 PROC – Pseudo File System 4](#_Toc472385059)

[3.0 Process System Calls 6](#_Toc472385060)

[4.0 Processes and Signals handling 8](#_Toc472385061)

[5.0 Windows Processes Analysis with Windbg 11](#_Toc472385061)

6.0 Inter-Process Communication (IPC)……………………...……………………………………….14

L*abs must be submitted by the due date for full credit. After due date late submissions will be accepted for a period of one week (seven days) and the grade will be reduced by ten percent (10%) per day after due day.* ***Assignments that are submitted more than seven days late will receive a grade of zero (0).***

I certify that the work submitted in this assignment is my own and that it has not been taken in whole or in part from any other source. I understand that the penalty for plagiarism will include a grade of zero (0) for this assignment plus disciplinary action in accordance with SAIT policies.

Signature: \_\_\_Coleton Sanheim\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**EVALUATION**:

|  |  |  |
| --- | --- | --- |
| Trace Processes | 10 |  |
| Explore PROC – Pseudo File system structure | 10 |  |
| Analyze process System calls | 15 |  |
| Run and analyze process and Signals Handling | 15 |  |
| Windows Process Analysis with debugger Windbg | 15 |  |
| Inter-Process Communication | 15 |  |
| TOTAL MARK | 80 |  |

Lab Outcome(s)

* Trace processes running on Linux systems
* Apply system calls to create and terminate processes
* Run and modify c programs to analyze processes system calls
* Send signals to processes using Linux utilities and system calls.

Reading

* Textbook chapter 3, sections 3.3.1 Process Creation and 3.3.2 Process Termination. Chapter 20 (Linux system ) section 20.4 Process Management

Introduction

A process is a program in execution. A Process requires system resources to complete its jobs such as: CPU time, memory space (memory areas to store image (executable) and to store global variables and stack for local variables and functions arguments), open files (file descriptor) and other handles.

In UNIX/Linux operating systems processes stablish parent /child relationship. Processes can be in different states such as: Running on CPU or waiting for resources in a specific queue. A signal can be sent to a process to suspend it, terminate it or change its behaviour.

To find information about signals use the command: **info signal**

To create or clone a process UNIX/Linux operating systems implements **fork( )** system call which generates an integer value to differentiate between a parent and child process (value=0). While the child implements **exec( )** system call to execute the program, the parent will invoke the system call wait() to wait for the child to complete its job –(**wait(&childPID)** waits on child PID). After child process completes its job the signal **(SIGCHLD)** is sent to the parent process which will invoke the wait( ) or waitpid() system call and releases resources used by child process. If the parent does not exist the child process may become an orphan (child of init process or a thread) or a zombie that will hold resources affecting system performance.

1. Trace Processes \_\_\_/10

Unix/Linux systems support many utilities/commands that can be used to manage processes. ***ps*** and ***top*** commandsarecommon commands to manage processes. Use Linux commands to manage processes and threads as follows:

1. Use the man command to learn different options that can be used with ***ps*** command. Write down the command and syntax that redirects to a file called **pro\_out** the Process Name (cmd), PID, and STATE of **ALL** processes on the system. Attach the created file called **pro\_out**

**ps -eo args,pid,stat > pro\_out**

1. Use the ***man ps*** command to learn about ***ps command***. At the last line of the manual type **/PROCESS STATE CODES** to search the manual and find information about process state. Read the topic and write down the different processes states code and the respective description.

**D uninterruptible sleep**

**I Idle kernel thread**

**R running or runnable**

**S interruptible sleep**

**T stopped by job control signal**

**t stopped by debugger during the tracing**

**W paging**

**X dead**

**Z defunct ("zombie") process, terminated but not reaped by its parent**

**For BSD formats and when the stat keyword is used, additional characters may be displayed:**

**< high-priority**

**N low-priority**

**L has pages locked into memory**

**s is a session leader**

**l is multi-threaded**

**+ is in the foreground process group**

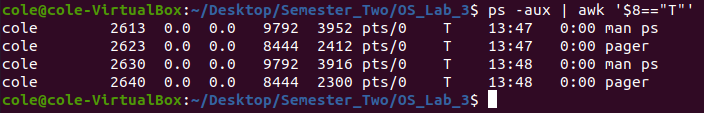
1. Run the **top command** and analyze top results
   1. How many processes at running at a time?

**1-2 processes**

* 1. What is the difference between top and ps commands?

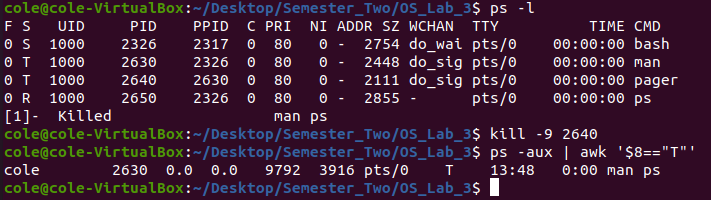
**Top constantly updates while ps only shows at the time the command was used**

1. Attach screen captures to demo the following:
   1. Create two processes. Suspend the processes (you can use Ctrl-z or &) and use the respective command(s) to display **ONLY** the properties or attributes of the suspended process.



1. Create a Zombie process as follows :
   1. Start, and keep running, man by typing: **man ps**
   2. Suspend the process by pressing ctrl-z
   3. Find the status of man process and its child processes: **ps -l**
   4. Terminate a parent process: **kill -9 *PID\_of\_pager***
   5. Find the status of man process and its child processes: **ps –l**

6. Attach a screen capture that displays the creation of a zombie process



2.0 /PROC pseudo file-system Structure \_\_\_\_/10

The **/proc pseudo file-system** is the root source of information about the processes within a GNU/Linux system. Within **/proc** directory, you'll find a set of directories with numbered filenames. These numbers represent the **process IDs** (pids) of active processes within the system.

Each ***pid*** directory presents a hierarchy of information about that process (process data structure) including the command line that started it, a symbolic link to the root file system, a symbolic link to the directory (current working directory) where the process was started, and other **process attributes** and configurations. Use **man proc** command for more information.

Under /proc directory the directories identify with numbers represent the PID of processes running in the system. These directories are the PCB (Process Control Block) of the processes

1. Use the respective command to identify the PID of **init** or **systemd** process
2. Use cd to change to /proc directory and use the respective PID of init or systemd to explore the attributes of the process
3. Edit the file called **status** for **init or systemd** process and answer the following questions:
4. What is the state of this process?

**State: S (sleeping)**

1. What is its PID and PPID?

**PID: 1**

**PPID: 0**

1. How many threads are running in this process?

**Threads: 1**

1. What is **context switch**?

**voluntary\_ctxt\_switches: 4252**

**nonvoluntary\_ctxt\_switches: 1160**

4. Use **man proc**. Explore **/proc/pid** information, demo and explain the purpose of the following:

* 1. Limits

**Used to define how much space/memory can be allocated to certain aspects of the process**

* 1. fd (File Descriptor)

**Used to contain all the file descriptors**

* 1. environ

**Used to contain the environment variables**

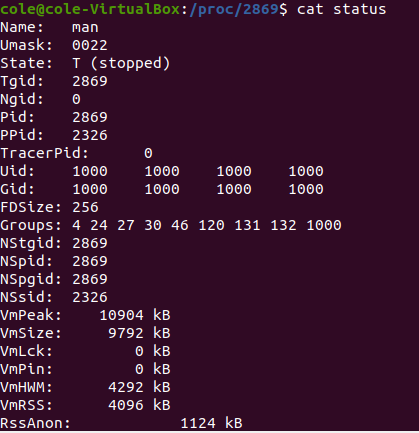
* 1. stack

**Describes what is on the stack**

5. Use **man proc** to find and briefly describe the purpose of pid\_max file located in **/proc/sys/kernel**

**This file specifies the value at which PIDs wrap around (i.e., the value in this file is one greater than the maximum PID). PIDs greater than this value are not allocated; thus, the value in this file also acts as a system-wide limit on the total number of processes and threads. The default value for this file is 32768**

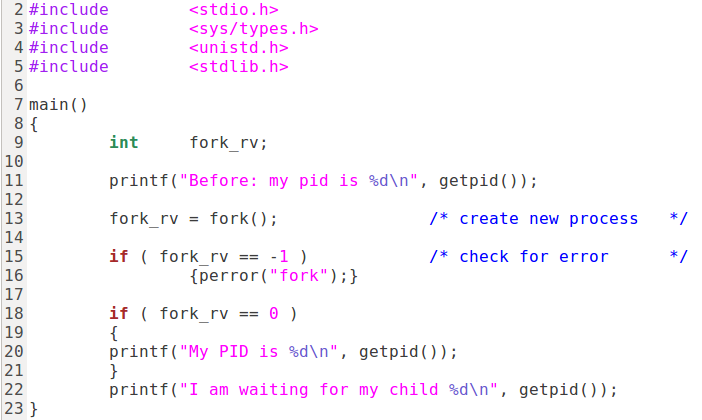
1. Create a new process (e.g. man, bash , gedit, or browser)
2. Identify process PID
3. Access /PROC pseudo file system and attach a screen captures that demo some of the attributes of the process you created



**3.0 Process system calls \_\_\_\_\_\_/15**

**fork( ), execvp( ) and wait( ) system calls**

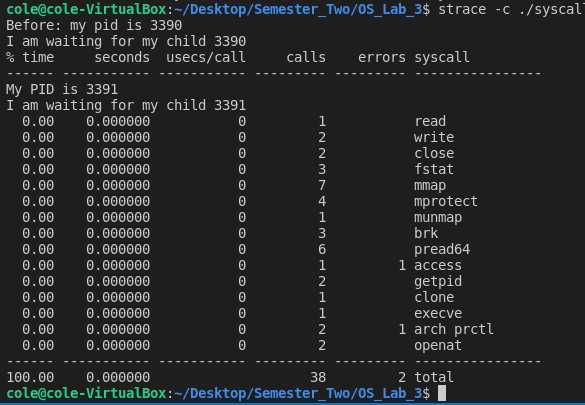
1. Compile and run the following C code



1. What is the purpose of this code?

**To clone the process of the program and print out the pid of parent and child**

1. Use **strace –c** to identify the system calls used by this program. Attach the screen capture that displays the system calls. Explain the purpose of the different **fork( ) system call output values:**



**0**

**Returns 0 if successfully complete**

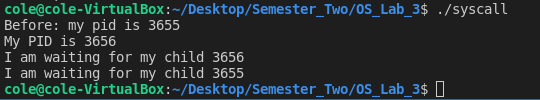
**-1**

**Returns -1 if not successfully completed**

**>0**

**Does not return >0**

1. Identify child process and parent process code and modify the program using **sleep()** and **wait()** system calls so that the child process sleeps for 5 seconds and the parent waits for the child to complete its task.
2. Attach modified code and results



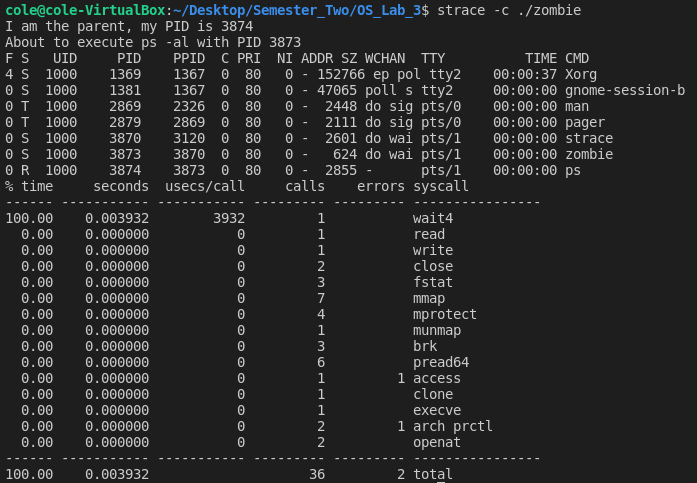
1. Compile and run the following C code
   1. What is the purpose of this code?

**To clone a new process of itself**

* 1. How many processes were cloned in this program?

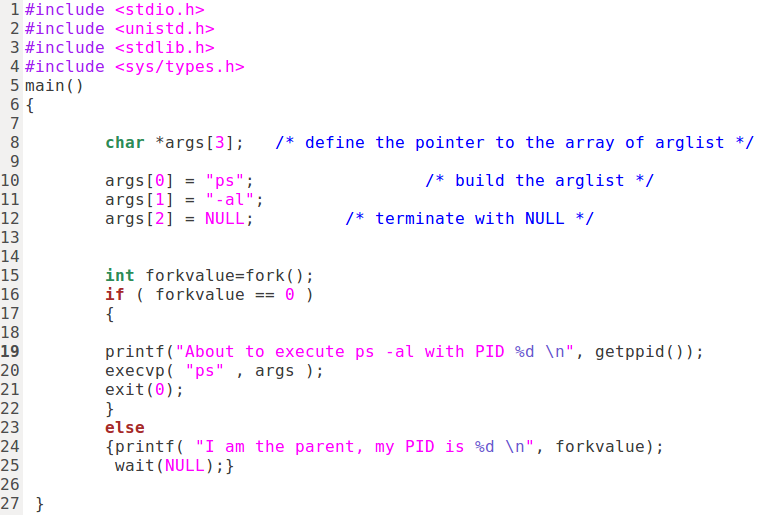
**1, the program its self was cloned**

* 1. Identify the system calls in this program

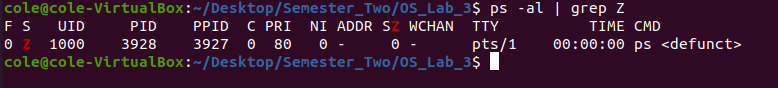


* 1. Which process executes ps command (child or parent process)?

**Child process**



* 1. Modify the program to create a zombie by making the parent process to **sleep(120).**
  2. Compile and run the program. While the program is running open a second terminal and use the command: **ps – Al | grep Z** to identify the zombie.
  3. Attach modified code and the results



4.0 Processes and Signal handles \_\_\_\_/15

Signals are messages sent to a process. Requests for signals come from users, kernel or processes. Signals generated by programs are synchronous signals ( trap) . Signals generated by user (hardware) are asynchronous signals. Signals numbers and their symbolic names are in /usr/src/include/signal.h .

1. The command **kill -l** displays signals supported by Linux . How many signals are supported by Linux? **\_\_\_\_64\_\_\_\_\_**

2. Use the command ***kill –l*** to identify the code for the following signals:

a. SIGKILL **\_\_\_9\_\_\_**

b. SIGTERM **\_\_15\_\_\_**

c. SIGSTOP **\_\_\_19\_\_\_**

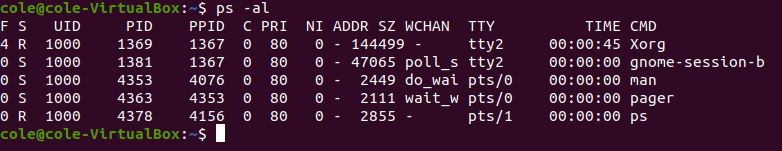
d. SIGHUP **\_\_\_1\_\_\_**

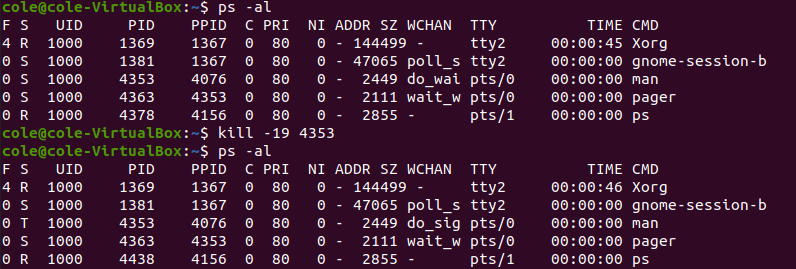
e. SIGCHLD **\_\_\_17\_\_\_**

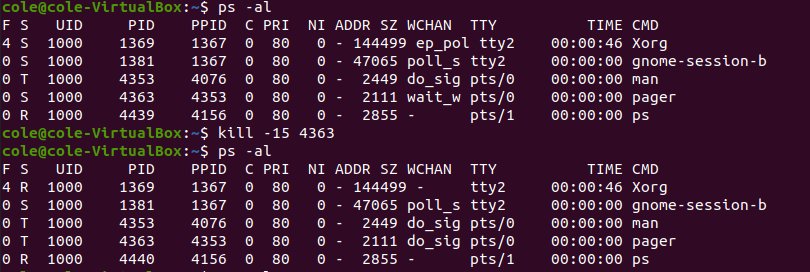
1. What is the difference among SIGKILL, SIGTERM and SIGSTOP?

**SIGKILL causes a process to terminate immediately, SIGTERM does the same but it can be caught, blocked or ignored. SIGSTOP will stop a process for later resumption.**

1. Type ***man ls*** to start a process
2. Open a second terminal and use ***ps*** command with respective options to find the PID of ***man***  process
3. Use the ***kill command*** with respective code to send the SIGSTOP to **man** process
4. Use the kill command with the respective code to send SIGTERM signal to a process. Use ***ps*** to verify the results.
5. Attach the screen captures that demo the results







**Signals handling**

Compile and run the following Signals program:

/\*\* sigsample.c demonstrates how signals can terminate a process,

\* be ignored by a process, or be caught by a process \*\*/

#include <stdio.h>

#include <signal.h>

void catcher() /\* a function to run when a signal is caught \*/

{

printf(" Ouch! \n");

system("who");

}

main()

{

int i;

printf("Type ^C during either Case 1, Case 2, or Case 3");

printf("Case 1: no special arrangements..");

for (i=0;i<10;i++){

putchar('\*');fflush(stdout);

sleep(1);

}

putchar('\n');

**signal**(SIGINT, SIG\_IGN); /\* ignore INTerrupts \*/

printf("Case 2: ignoring interrupts..");

for (i=0;i<10;i++){

putchar('\*');fflush(stdout);

sleep(1);

}

putchar('\n');

signal(SIGINT, catcher); /\* handle interruptions \*/

printf("Case 3: catching interrupts..");

for (i=0;i<10;i++){

putchar('\*');fflush(stdout);

sleep(1);

}

putchar('\n');

}

1. Research and briefly explain the main system calls of the program: putchar(), fflush(), and signal().

**putchar() outputs a char to the terminal**

**fflush() clears the buffer of the stream**

**signal() sets a function to a handle**

1. Specifically, what does signal(SIGINT, SIG\_IGN) do?

**Tells the program to ignore the SIGINT signal**

1. Specifically, what does signal(SIGINT, catcher) do?

**Tells the program to catch the SIGINT signal**

1. Use **man kill** command and differentiate signals default actions. What is the difference among **exit**, **stop** and **core** actions?

**Exit – terminate the process**

**Core – produce a core dumb then terminate the process**

**Stop – suspend the process**

1. What information is in a "core dump"?

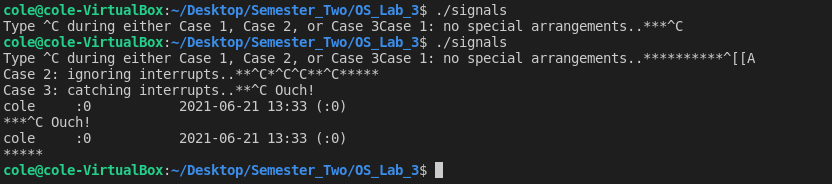
**System memory and the part of the program that aborted, it also may contain:**

* **The processor’s state**
* **The processor register’s contents**
* **Memory management information**
* **The program’s counter and stack pointer**
* **Operating system and processor information and flags**

1. Why SIGKILL and SIGSTOP cannot be blocked?

**So there is always some way to stop an unresponsive program**

1. Attach screen captures that demo the code results ( signal() system call implementation). Explain the results for each case.



**Case 1: it handles interrupts the default way, by interrupting the program**

**Case 2: it ignores all interrupts and continues running**

**Case 3: it catches the interrupt, prints out ouch and the current user, then continues running**

5.0 Process Analysis with Windbg \_\_\_/15

Kernel debugging can be used to investigate user mode processes as well as internal kernel structure. Debugging tools for Windows support remote and local kernel debugging. To perform kernel debugging the following files are required:

* 1. Symbols file that contain the name of functions and variables and the format of data structure. This file is generated by the linker. Symbol table files must match version of the image. The following command loads required symbols from the internet symbol server and keeps a local copy under C:\symbols

srv\*c:symbols\*http://msdl.microsoft.com/download/symbols

* 1. Debugging Tools for Windows located at <http://msdn.microsoft.com> these tools can be used to debug user and kernel mode processes/threads
  2. Windows Software Development Kit (SDK) located at <http://developer.microsoft.com/en-US/windows/downloads/windows-10-sdk>

It contains C header and libraries necessary to compile and link Windows applications.

Instructions:

* + 1. Start Windows 10 virtual machine provided by instructor and login. Password: OperatingSystems
    2. Verify if Secure Boot is disabled.
    3. To enable kernel debugger, the debugger needs to be on.

Use bcdedit /debug on

* + 1. Access C:\Program Files (x86) 🡪 Windows Kits 🡪 10 🡪Debuggers🡪 x64
    2. Run windbg as Administrator
    3. Click on File🡪 Kernel Debugging and select Local ( We will debug locally ).
    4. Once in local kernel –debugging mode, many commands that start with ( ! ) known as bang can be used to display the contents of internal processes/threads and memory data structure.
    5. Use Lived kernel debugger lkd > to explore processes and threads data structure by typing the following commands:

a. ldk> dt nt!\_eprocess displays the data structure of executive process including KPROCESS ( which is the PCB of each process in Windows). KPROCESS is within EPROCESS and stored in pcb ( Process Control Block)

b. ldk> dt nt!\_kprocess displays processes data structure (PCB)

c. if you want to display the data type and structure of only one field you can use

ldk> dt nt!\_eprocess UniqueProcessID

The !process extension command displays metadata corresponding to a particular process or all processes. The syntax is:

!process Process ID Flags or

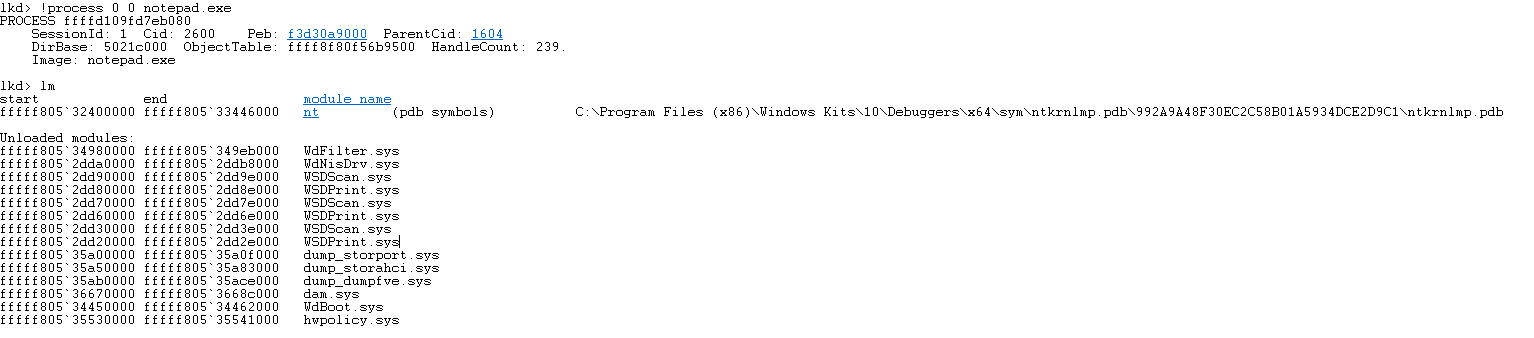
!process Process base address Flags

If Flags is 0 only minimum amount of information is displayed

If you do not know the process ID or base address you can use

!process 0 0 as follows:

1. ldk> !process 0 0 will display summary information about each process and its threads currently running .
2. ldk> !process 0 0 service.exe will display all instances of this process. Write down the ID of one of the instances of explorer.exe process and find the details of that instance using the command !process ID (replace ID with respective process ID – Cid ) or !process memory-address (replace memory address with respective process memory address)
3. ldk> dt nt!\_eprocess ffffca83e7281080 Will display the data structure and including address to pcb (kprocess) for the process with specified address
4. (5 marks) Kernel Mode- Use Windows debugger (lkd) kernel debugger to explore windows protected process and its threads data structure
   * Display and analyze information of a process such as notepad.exe as follows: start notepad.exe or any other process you want to analyze !process 0 0 notepad.exe
   * Use ldk> lm to display the user mode modules loaded by this process
   * Provide screen captures that displays
     1. basic information of notepad.exe process or the process you decide to analyze
     2. the process control block of this process,
     3. Display the peb and token structure for the process. What is the purpose of peb and token structure?
     4. Details of one thread of this process



Windows System Calls. There are many system calls in Windows. Windbg can be used to display the address, assembly and other functions calls by the system calls. The System Service Descriptor Table (SSDT) table contains pointers (In Unix is the syscall Table) to system calls. Each address in the table is the entry point of a routine in kernel space (system call). System calls that start with Nt are routines used by user-mode components of the OS to interact with kernel mode. Drivers cannot called these routines because drivers are kernel mode.

In the kernel SSDT table is called KeServiceDescriptorTable and is exported by ntoskrnl.exe.

* + - 1. Use the following command to display the address of the KeServiceDescriptorTable and record the address

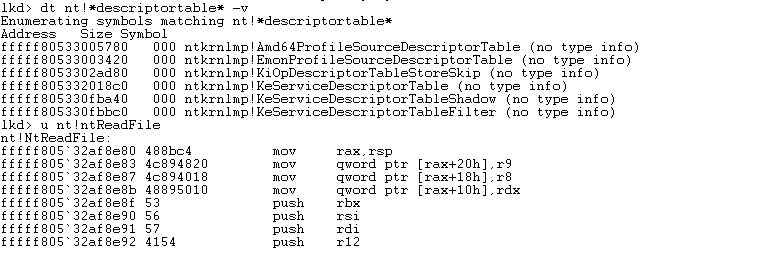
ldk> dt nt!\*descriptortable\* -v

* + - 1. NTReadFile is a system call. Use the implementation of this system call by displaying the assembly code of this routine as follows:

ldk> u nt!NTReadFile

The mov instructions are moving pointers into specified register and the push instruction (operation) is pushing the arguments of the system call into the stack. This demonstrates how arguments of the system calls are moved to registers directly and others are pushed into the stack (memory)

* + - 1. Attach screen captures that demo results or output of address of SSDT and the assembly of the system call NTReadFile



6.0 Inter-Process Communication (IPC)-Sockets \_\_\_\_/15

A system that runs several processes requires techniques to communicate them. The processes running in the same machine are called related processes and can exchange information via kernel communication techniques such as pipes or sharing memory (message passing). Processes running in different machines such as client /server systems (unrelated processes) requires more complex techniques to communicate such as sockets. A socket creates an endpoint for communication and returns the identifier for the socket. Sockets modes are: STREAM for TCP communication or DATAGRAM for UDP communication.

The following are the system calls required by a server to communicate with clients:

1. Socket(). Create a socket. A socket requires a domain such as Internet, type of socket (STREAM or DATAGRAM) and a protocol within the network code in the kernel.
2. Bind(). Bind the address to the socket. Assign address (combination of host and port) to socket.
3. Listen(). Listen for connection on the socket. Listen to incoming calls in a queue
4. Accept().Accept a connection on a socket. It requires pointer to structure of address of caller and length of address of caller
5. Transfer Data. In TCP will be read( )and write( )system calls. In UDP will be sendto( ) and recvfrom( ) system calls
6. Close(). Close connection. The file descriptor returned by accept() may be closed with close() system call

Client systems will require the following system calls:

1. Socket(). Socket to connect to the network.
2. Connect(). Connect to a socket. Attempts to connect the socket specified by pointer to server address. Only STREAM sockets (TCP) require connection. DATAGRAM sockets do not required connection
3. Transfer data by reading (read()) and written(write() to file descriptors for TCP communication. DATAGRAM sockets will send and receive messages
4. Closed connection
5. Download from D2L the server code gdrecv.c , the client code dgsend.c and dgram.c ( it contains sockets functions and address struct)
6. Use Linux man to learn about the following system calls: read the system call definition, synopsis and identify the header and the arguments required in each of system call:
   1. man socket

**#include <sys/types.h>**

**#include <sys/socket.h>**

**int socket(int domain, int type, int protocol);**

* 1. man gethostname

**#include <unistd.h>**

**int gethostname(char \*name, size\_t len);**

**int sethostname(const char \*name, size\_t len);**

* 1. man bind

**#include <sys/types.h>**

**#include <sys/socket.h>**

**int bind(int sockfd, const struct sockaddr \*addr,**

**socklen\_t addrlen);**

* 1. man recvfrom

**#include <sys/types.h>**

**#include <sys/socket.h>**

**ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);**

**ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags,**

**struct sockaddr \*src\_addr, socklen\_t \*addrlen);**

**ssize\_t recvmsg(int sockfd, struct msghdr \*msg, int flags);**

* 1. man sendto

**#include <sys/types.h>**

**#include <sys/socket.h>**

**ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);**

**ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags,**

**const struct sockaddr \*dest\_addr, socklen\_t addrlen);**

**ssize\_t sendmsg(int sockfd, const struct msghdr \*msg, int flags);**

1. Edit dgram.c , dgrecv.c and dgsend.c code and identify the system calls and compare the system calls arguments with the ones you learnt in Linux manual. List the system calls implemented by the server and by the client

Server Client

**recvfrom(sock,buf,BUFSIZ,0, sendto(sock, msg, strlen(msg), 0,**

**(struct sockaddr \*) &saddr,&saddrlen))>0 ) (struct sockaddr \*)&saddr,sizeof(saddr))**

1. Compile the code as follows: gcc dgrecv.c dgram.c -o receiver
2. Run the server process listening to port 4444 as follows: ./receiver 4444
3. Open a second terminal and use ps command to find and record PID of this server process
4. Open another terminal to compile sender (client) process as follows: gcc dgsend.c dgram.c -o sender
5. Send a message to the server as follows: ./sender hostname 4444 “ Hello Server “ **NOTE:** replace hostname with the respective hostname
6. After sending the message to server. Verify on the terminal running server process if the message was transferred. Attach the screen capture with the results



1. Analyze the results. What type of communication are we implementing TCP or UDP ( Socket STREAM or DATAGRAM)? why

**It utilizes UDP, as TCP uses STREAM so there would be a connect() systemcall in dgram but there is not so it is UDP which is DATAGRAM**

1. Create screen captures that demo client and server running and the results

