**OPERATING SYSTEM (PRACTICALS) – FALL 2012**

**EXPERIMENT 4 – CREATING PROCESSES**

|  |  |  |  |
| --- | --- | --- | --- |
| **DATE: 23/10/2012** | | **Students Names: MOHSIN KHAN/ M.AWAIS** | |
| **Marks Obtained: \_\_\_\_\_** | | **COURSE: BESE 16 A** | |
| **Deadline: 1400 hrs 23rd Oct 2012** | | **Instructor: Engr. Umar Mahmud** | |
|  | **Instructions**   * This lab is to be performed by a syndicate of at most **TWO** students. Write your remarks next to the space provided. * Plagiarism is strictly forbidden. * Submit hard copy of the report before deadline. Marks will be deducted for late submissions. | |  |
| 1. | **Objectives:**   1. Creating processes in Ubuntu. 2. Learning fork commands in Ubuntu | |  |
| 2. | **Time Required:** 3 hrs | |  |
| 3. | **Software Required:**   1. Ubuntu/Linux and gcc compiler (already available in Ubuntu) 2. Borland/Turbo C/Visual Studio for Windows Platform 3. VMWare if required 4. C language | |  |
| 4. | **How to compile C Program in Ubuntu/Linux**   1. Save your program with extension “.c” in any directory (working directory). 2. Open Terminal. 3. Navigate to the working directory 4. Suppose you saved the program with name “hello.c”. Now write following lines:   gcc -c hello.c  gcc -o hello hello.c  ./hello   1. The first line compiles your program. Object file hello.o is created 2. Second line makes an exe file with the name hello. 3. Third line runs the exe file and shows output of the program. | |  |
| 5. | **Using getpid():** This function returns the pid of the current program. Use the following code and write the output.  int main(){  int pid;  pid = getpid();  printf(“Process ID is %d\n”, pid);  return 0;  } | |  |
| 6. | What is the outcome of the following program?  int main(){  long i;  printf(“Process ID is %d\n”, getpid());  for(i=0; i<=400;i++)  {  printf(“i is %d\n”, i);  }  return 0;  } | |  |
| 7. | **Using getppid():** This function returns the pid of the parent process.  int main(){  int ppid;  ppid = getppid();  printf(“Parent Process ID is %d\n”, ppid);  return 0;  }  What is the outcome? | |  |
| 8. | **Using fork():** fork command in Linux creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent.  <http://manpages.ubuntu.com/manpages/lucid/man2/fork.2.html>  What is the outcome of the following program?  int main(){  fork();  printf(“The PID is %d\n”, getpid());  return 0;  }  **The PID is 2215** | |  |
| 9. | What is the outcome of the following program?  int main(){  int pid;  pid = fork();  if(pid==0){  printf(“I am the child, my process ID is %d\n”, getpid());  printf(“The child’s parent process ID is %d\n”, getppid());  }  else{  printf(“I am the parent, my process ID is %d\n”, getpid());  printf(“The parent process ID is %d\n”, getppid());  }  return 0;  }  **I am the parent, my process ID is 2454**  **The parent process ID is 2150**  **I am the child, my process ID is 2455**  **The child’s parent process ID is 1** | |  |
| 10. | To see if the pid is same as shown in the system, Open System Monitor. Check to see if the pid is same. Use the following code  int main(){  int pid,i;  pid = fork();  if(pid==0){  printf(“I am the child, my process ID is %d\n”, getpid());  printf(“The child’s parent process ID is %d\n”, getppid());  }  else{  printf(“I am the parent, my process ID is %d\n”, getpid());  printf(“The parent process ID is %d\n”, getppid());  }  scanf(“%d”,&i); //so that program halts for user input  return 0;  }  **I am the parent, my process ID is 2477**  **The parent process ID is 2150**  **I am the child, my process ID is 2478**  **The child’s parent process ID is 2477** | |  |
| 11. | What do you understand from points 8, 9 and 10?  **The fork() call can be used to generate child processes, process id of a process a be returned by getpid() call and its parent by getppid() call. In linux fork() can be used to generate new proccesses.** | | (3) |
| 12. | What is the outcome of this program?  /\*\*  \* This program forks a separate process using the fork()/exec() system calls.  \*  \* Figure 3.10  \*  \* @author Gagne, Galvin, Silberschatz  \* Operating System Concepts - Seventh Edition  \* Copyright John Wiley & Sons - 2005.  \*/  #include <stdio.h>  #include <unistd.h>  #include <sys/types.h>  int main(){  pid\_t pid;  /\* fork a child process \*/  pid = fork();  if (pid < 0) { /\* error occurred \*/  fprintf(stderr, "Fork Failed\n");  exit(-1);  }  else if (pid == 0) { /\* child process \*/  printf("I am the child %d\n",pid);  execlp("/bin/ls","ls",NULL);  }  else { /\* parent process \*/  /\* parent will wait for the child to complete \*/  printf("I am the parent %d\n",pid);  wait(NULL);    printf("Child Complete\n");  exit(0);  }  }  **I am the parent 2796**  **I am the child 0**  **a.out \_Chapter\_3\_c\_.pptx**  **\_Chapter\_1\_a\_.pptx Experiment\_03\_-\_Process.docx**  **\_Chapter\_3\_a\_.pptx Experiment\_04\_-\_Creating\_Processes.docx**  **\_Chapter\_3\_b\_.pptx x.cpp**  **Child Complete** | |  |
| 13. | Explain how was the execution carried out in program in point 11? You may use a flow model to describe it.  **Assumption: point 12 execution**  **once the fork() call is encountered, the child process is created, the parent is suspended in the else scope by the wait() call to wait for the termination of the child process,the child process is run and I.e the else if scope, and executes the ls program which displays the directory listing and the child execution is completed, the control is returned to the parent and the parent finish the execution.**  A description... | | (3) |
| 14. | **Using Exec:** The fork system call creates a new process but that process contains, and is executing, exactly the same code that the parent process has. More often than not, we'd like to run a new program. | |  |
| 15. | **Example of Execve():**The *execve* system call replaces the current process with a new program.Type the following command in Terminal and show the output.  ls -aF /  Now execute the following code  /\* execve: run a program \*/  #include <stdlib.h> /\* needed to define exit() \*/  #include <unistd.h> /\* needed to define getpid() \*/  #include <stdio.h> /\* needed for printf() \*/  int  main(int argc, char \*\*argv) {  char \*args[] = {"ls", "-aF", "/", 0}; /\* each element represents a command line argument \*/  char \*env[] = { 0 }; /\* leave the environment list null \*/  printf("About to run /bin/ls\n");  execve("/bin/ls", args, env);  perror("execve"); /\* if we get here, execve failed \*/  exit(1);  }  What is the outcome?  **Output of ls -aF /:**  **./ boot/ etc/ lib/ mnt/ root/ srv/ usr/**  **../ cdrom/ home/ lost+found/ opt/ sbin/ sys/ var/**  **bin/ dev/ initrd.img@ media/ proc/ selinux/ tmp/ vmlinuz@**  **Outcome of code:**  **About to run /bin/ls**  **./ boot/ etc/ lib/ mnt/ root/ srv/ usr/**  **../ cdrom/ home/ lost+found/ opt/ sbin/ sys/ var/**  **bin/ dev/ initrd.img@ media/ proc/ selinux/ tmp/ vmlinuz@** | |  |
| 16. | **Using Execlp**  execlp, which allows you to specify all the arguments as parameters to the function. Note that the first parameter is the command. The second parameter is the first argument in the argument list that is passed to the program (argv[0]). These are often the same but don't have to be. The last parameter must be a null pointer.  /\* execlp: run a program using execlp \*/  #include <stdlib.h> /\* needed to define exit() \*/  #include <unistd.h> /\* needed to define getpid() \*/  #include <stdio.h> /\* needed for printf() \*/  int main(int argc, char \*\*argv) {  printf("About to run ls\n");  execlp("ls", "ls", "-aF", "/", (char\*)0);  perror("execlp"); /\* if we get here, execlp failed \*/  exit(1);  }  What is the output?  **About to run ls**  **./ boot/ etc/ lib/ mnt/ root/ srv/ usr/**  **../ cdrom/ home/ lost+found/ opt/ sbin/ sys/ var/**  **bin/ dev/ initrd.img@ media/ proc/ selinux/ tmp/ vmlinuz@** | |  |
| 17. | More help can be found here <http://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/exec.html> | |  |
| 18. | **Using fork() and exec():** The fork system call creates a new process. The execve system call overwrites a process with a new program. A process forks itself and the child process execs a new program, which overlays the one in the current process.  /\* forkexec: create a new process. \*/  /\* The child runs "ls -aF /". The parent wakes up after 5 seconds \*/  #include <stdlib.h> /\* needed to define exit() \*/  #include <unistd.h> /\* needed for fork() and getpid() \*/  #include <stdio.h> /\* needed for printf() \*/  Int main(int argc, char \*\*argv) {  void runit(void);  int pid; /\* process ID \*/  switch (pid = fork()) {  case 0: /\* a fork returns 0 to the child \*/  runit();  break;  default: /\* a fork returns a pid to the parent \*/  sleep(5); /\* sleep for 5 seconds \*/  printf("I'm still here!\n");  break;  case -1: /\* something went wrong \*/  perror("fork");  exit(1);  }  exit(0);  }  void runit(void) {  printf("About to run ls\n");  execlp("ls", "ls", "-aF", "/", (char\*)0);  perror("execlp"); /\* if we get here, execlp failed \*/  exit(1);  }  What is the outcome of the program?  **About to run ls**  **./ boot/ etc/ lib/ mnt/ root/ srv/ usr/**  **../ cdrom/ home/ lost+found/ opt/ sbin/ sys/ var/**  **bin/ dev/ initrd.img@ media/ proc/ selinux/ tmp/ vmlinuz@**  **I'm still here!** | |  |
| 19. | What do you understand from point 18?  **when a child is created the parent still stays there till the child is not terminated. In the above case first the child is created and runit is run in which execlp is used to execute ls command, after the child exits, control is returned to the parent and since now pid is not 0 the default case is run and after 5 seconds of sleep the program is terminated.** | | (3) |
| 20. | More help can be found here  <http://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/forkexec.html>  <http://www.advancedlinuxprogramming.com/alp-folder/alp-ch03-processes.pdf> | |  |
| 21. | **Creating a Process in Windows:**  /\*\*  \* This program creates a separate process using the CreateProcess() system call.  \*  \* Figure 3.12  \*  \* @author Gagne, Galvin, Silberschatz  \* Operating System Concepts - Seventh Edition  \* Copyright John Wiley & Sons - 2005.  \*/  #include <windows.h>  #include <stdio.h>  int main( VOID ){  STARTUPINFO si;  PROCESS\_INFORMATION pi;  ZeroMemory( &si, sizeof(si) );  si.cb = sizeof(si);  ZeroMemory( &pi, sizeof(pi) );  // Start the child process.  if( !CreateProcess( NULL, // No module name (use command line).  "C:\\WINDOWS\\system32\\mspaint.exe", // Command line.  NULL, // Process handle not inheritable.  NULL, // Thread handle not inheritable.  FALSE, // Set handle inheritance to FALSE.  0, // No creation flags.  NULL, // Use parent's environment block.  NULL, // Use parent's starting directory.  &si, // Pointer to STARTUPINFO structure.  &pi ) // Pointer to PROCESS\_INFORMATION structure.  )  {  printf( "CreateProcess failed (%d).\n", GetLastError() );  return -1;  }  // Wait until child process exits.  WaitForSingleObject( pi.hProcess, INFINITE );  // Close process and thread handles.  CloseHandle( pi.hProcess );  CloseHandle( pi.hThread );  } | |  |
| 22. | Which OS gives you a better interface for creating and executing child programs and why?  **Linux provides a better interface for generating and executing child processes because its code and commands are quite small,simple and straight forward as compared to windows.** | | (1) |