Process Control Part 1

|  |  |
| --- | --- |
| **Processes**  To manage executables, operating systems treat each executable as an instance of a process. With modern operating systems, many processes can be executed at the same time.  You can invoke the same program like vim multiple times (via separate terminal windows) while using a workstation in our lab. Each invocation of vim would be a different process instance (each having a unique process ID). Each of those terminal windows will also be a process.  If the hardware has only one CPU, how does the operating system manage execution of multiple processes at the same time? It switches between processes giving each some execution time. When does it switch?   * Some steps like I/O require blocking to wait until the request completes. If the process is waiting for user input, the blocking may last a long time. * Errors (illegal instruction, seg fault) * Interrupts   + User types data on a keyboard   + Completion of an I/O request from/to disk   + A timer interrupt allowing the OS to gain control * System calls (e.g., sleep)   To get the files containing information for interprocess communication, cp –r /usr/local/courses/rslavin/cs3423/IPC . | Process State Diagram    Basically they are put in the queue in the ready state |
| **Processes and Threads**  Processes and threads are both independent sequences of execution. Processes can create threads for execution. Threads share memory with the processes that create them. Processes can also create other processes which don't share memory with the parent process.  In Microsoft Windows, processes can create threads. Workstation GUI applications typically have one or more threads for managing user interaction.  In Linux, processes can **fork** to create additional parallel processes. | Threads share resources  Processes do not share memory |
| **Why should we create parallel Processes?**  **GUI**:  For GUI applications, the user may request a large quantity of data. Instead of blocking his/her use of the GUI application until all of the data is gathered, we could read the data (or a subset) in a separate process/thread, but allow the user to continue working with the application.  **Execution Concurrency**:  Some algorithms may work more effectively by dividing the problem into multiple parallel processes. (This might not be effective if running on a single CPU.)  **Optimizing Resource Usage**:  Some processes may be I/O intensive. While performing I/O a process blocks, freeing the CPU for use by other processes.  **Terminating an Unresponsive Request**  A request for some resource (database, server) might take too long. Instead of blocking "forever" (and annoying a consumer), another process may have a timer to decide that the process making the request is taking too long and should be terminated. | **Example 1: Some example programs which use parallel processes**  **Text Editor**: while user is editing text, a parallel process looks for spelling/grammar errors and marks them.  **Interactive Games**: multiple users (each executing on separate processes) manipulate their game characters simultaneously.  **Spreadsheet**: after data is entered in a spreadsheet cell, a parallel process propagates changes to affected cells.  **Database**: database server satisfies multiple requests in parallel. Since the request cause I/O, it is more efficient to run multiple requests instead of single threading requests. |
| **Process Identification Functions**  pid\_t **getpid**() returns the process ID (pid) for this process  pid\_t **getppid**() returns the process ID of the parent of this process  Note that pid\_t is an unsigned integer. When printing it, either first assign it to a long or typecast it to a long.  These functions require <unistd.h>. | **Example 2: getting pid and ppid**  $ vi prtPids.c  #include <stdio.h>  #include <unistd.h>  int main ()  {  long lPid;  long lPPid;  lPid = getpid();  lPPid = getppid();  printf("PID=%ld, PPID=%ld\n", lPid, lPPid);  return 0;  } |
| **User Identification Functions**  int **getuid**() returns the user ID (uid) for this process. This isn't the login name.  int **getgid**() returns the group ID for this process  Linux also allows users to dynamically change their UID or GID to access files associated with those effective IDs. The process can then create files with owner and group based on those effective IDs.  int **geteuid**() returns the effective user ID for this process. This isn't the login name.  int **getegid**() returns the effective group ID for this process.  Usually, the real and effective IDs are the same.  These functions require <unistd.h>. | **Example 3: getting UID, GID, and the corresponding effective IDs**  $ vi prtUser.c  #include <stdio.h>  #include <unistd.h>  int main ()  {  printf("UID=%d, GID=%d\n", getuid(), getgid());  printf("Eff UID=%d, Eff GID=%d\n", getuid(), getgid());  return 0;  } |
| **Forking a Child Process**  A process can create a child process by calling **fork**. The calling process becomes the parent.  pid\_t **fork**() creates a child process. Returns:  0 child process receives this  child PID parent process receives this  -1 parent process receives this if the creation failed.  The child process receives a copy of the address space of the parent. Both processes continue with the instruction after the **fork** call.  In the code on the right, both the parent process and the child process increment lMyVal. Why does it print 15 in both processes instead of 15 and 20? ?? | **Example 4: creating a child process**  $ vi example4.c  #include <stdio.h>  #include <unistd.h>  #include <errno.h>  #include <string.h>  int main ()  {  long lForkPid;  long lMyVal = 10; // a variable to demonstrate  // that each process has their  // own copy  // create a child process  lForkPid = fork(); //fork return child process id  // Both the parent and child continue here  switch(lForkPid)  {  case -1:  errExit("fork failed: %s", strerror(errno));  break;  //if it is the child process, run the code in this case 0  case 0: // child process  printf("Child Process: PID=%ld, PPID=%ld\n"  , (long) getpid(), (long) getppid());  break;  default: // parent process  printf("Parent Process: PID=%ld, PPID=%ld\n"  , (long) getpid(), (long) getppid());  printf("Parent Process: my child's PID=%ld\n"  , lForkPid);  }  lMyVal += 5;  printf("My PID=%ld, lMyVal=%ld\n"  , (long) getpid(), lMyVal);  return 0;  }  $ gcc -o ex4 example4.c errExit.o  $ ./ex4  **Parent Process: PID=28952, PPID=24693**  **Parent Process: my child's PID=28953**  **My PID=28952, lMyVal=15**  **Child Process: PID=28953, PPID=28952**  **My PID=28953, lMyVal=15** |
| **Sharing of File Resources**  Child processes will initially share the same System File Table entries. This is important to prevent data from being lost when writing to a file.  Note that the parent and child processes are not sharing the FILE structure associated with stdio buffering. |  |
| **Impact of Sharing File Resources and File Buffering**  The sleep (*seconds*) function causes the process to sleep for approximately that number of seconds.  When we run example5 with output directed to the monitor, notice that the output is ordered differently from example4. It prints the first line from the parent process, but the sleep system call caused Linux to let the other process run. The second printf for the parent process didn't happen until after the sleep(2) completed.  Now execute example5 again, but redirect the output to a file. What happened? ??  Why?   * When the output is directed to a file, it ?? * The sleep(2) allowed Linux to switch execution to a different process. * The child process then wrote to its FILE structure's buffer. When the child process terminated, it wrote to the file. * The parent process then continued writing to its FILE structure's buffer. When the parent process terminated, the output was written to the file. | **Example5: sleep and file buffering**  $ cp example4.c example5.c  Insert the following line of code between the two printf statements within the switch **default** (i.e., the code only for the parent process)  sleep(2);  Compile and run the code again with stdout directed to the monitor.  $ ./ex5 This sleep 2 sec in between  **Parent Process: PID=29061, PPID=24693**  **Child Process: PID=29062, PPID=29061**  **My PID=29062, lMyVal=15**  **Parent Process: my child's PID=29062**  **My PID=29061, lMyVal=15**  $ ./ex5 > ex5.out This sleep 2 sec before writing to buffer and then flush 1 time at the end  $ cat ex5.out  **Child Process: PID=29066, PPID=29065**  **My PID=29066, lMyVal=15**  **Parent Process: PID=29065, PPID=24693**  **Parent Process: my child's PID=29066**  **My PID=29065, lMyVal=15** |
| **Waiting for a Child Process**  Frequently, parent processes need to wait (i.e., block) for child processes to complete before the parent process continues.  pid\_t **wait**(int \*piExitStatus)  causes the parent process to wait until one of its children exits (normally or abnormally).  **piExitStatus** – if non-NULL, this should be the address of an integer variable which will receive the exit status of the child process.  pid\_t **waitpid**(pid\_t childpid  , int \*piExitStatus, int iOptions)  causes the parent process to wait until the specified child exits (normally or abnormally).  **childpid** – the PID of the child that the parent wishes to wait on  **piExitStatus** – if non-NULL, this should be the address of an integer variable which will receive the exit status of the child process.  **iOptions** – options  The returned value from the wait functions is usually the pid of the child process that exited. They return -1 if an error occurred; therefore, examine errno for the reason.  Please see the waitpid manual page for more information on iOptions. Always initialize it to 0.  If we fork many children which ever finish first will cause the wait to stop | **Example6: waiting on a child**  #include <stdio.h>  #include <unistd.h>  #include <errno.h>  #include <string.h>  int main ()  {  long lForkPid;  long lWaitPid;  int iExitStatus;  int iExit;  char szInput[20];  // create a child process  lForkPid = fork();  // Both the parent and child continue here  switch(lForkPid)  {  case -1:  errExit("fork failed: %s", strerror(errno));  break;  case 0: // child process  printf("Child Process: PID=%ld, PPID=%ld\n"  , (long) getpid(), (long) getppid());  printf("Enter an exit value to continue child:");  scanf("%d", &iExit);  printf("Child is exiting with a %d\n", iExit);  return iExit;  default: // parent process  lWaitPid = wait (&iExitStatus); // wait for child to complete  if (lWaitPid == -1)  errExit("wait error: %s", strerror(errno));  printf("Parent Process: PID=%ld, PPID=%ld\n"  , (long) getpid(), (long) getppid());  printf("Parent Process: my child's PID=%ld\n"  , lForkPid);  printf("Parent Process: wait pid=%ld\n"  , lWaitPid); |
| **Waiting for a Child Process continued – Examine Exit Status**  If piExitStatus is not NULL, an exit status is returned through the parameter. It can be examined using some macros which are found in #include <sys/wait.h>. See the code for more information.  Notice that the **child continued** since the **parent is waiting**. We enter a value of 2 so that the child will exit returning 2. We must use the macros to properly examine the values. As the output shows, the child terminated normally, but returned 2.  Sizenote fo fun: cmd: ex6 & 🡸 this cmd send ex6 to the back ground  To bring to the foreground cmd: fg | **Example 6 continued: Waiting and Checking the exit status from the child**  printf("Parent Process: exit status=%d\n"  , iExitStatus);  if (WIFEXITED(iExitStatus) != 0)  printf("Child terminated normally: %d\n"  , WEXITSTATUS(iExitStatus));  if (WIFSIGNALED(iExitStatus) != 0)  printf("Child terminated due to uncaught signal: %d\n"  , WTERMSIG(iExitStatus));  }  printf("My PID=%ld\n"  , (long) getpid());  return 0;  }  $ gcc -o ex6 example6.c errExit.o  $ ./ex6  **Child Process: PID=26192, PPID=26179**  **Enter an exit value to continue child:2**  **Child is exiting with a 2**  **Parent Process: PID=26179, PPID=12775**  **Parent Process: my child's PID=26192**  **Parent Process: wait pid=26192**  **Parent Process: exit status=512**  **Child terminated normally: 2**  **My PID=26179** |
| **How can we have the child terminate abnormally?**  **kill -9 <pid>**  We tell ./ex6 to execute in the background. Since it has a child process, we see the output from the child process; however, stdin (the terminal) is associated with the foreground shell. Any commands we enter are not the stdin for ./ex6 until we make it the foreground process.  When the child was abnormally terminated, the parent continued executing; however, since it is a background process, we don't see the output. We entered the **fg** command to return the background process to the foreground. Linux then tells us that ./ex6 is being shown by simply stating: "./ex6".  The remainder of the output comes from ./ex6 | **Example 6 continued: Abnormal termination**  $ ./ex6 &  **[1] 26233**  $ (this is the foreground prompt)  **Child Process: PID=26234, PPID=26233**  **Enter an exit value to continue child:??**  **[1] + Suspended (tty input) ./ex6**  $ **fg**  **./ex6**  **Parent Process: PID=26233, PPID=12775**  **Parent Process: my child's PID=26234**  **Parent Process: wait pid=26234**  **Parent Process: exit status=9**  **Child terminated due to uncaught signal: 9**  **My PID=26233** |
| **Zombies and Orphans**  Linux wants parents to check on their children via wait or waitpid to get its exit status.  **What happens when a child terminates, but its parent wasn't waiting on it?** The child process becomes a zombie and still exists until the parent dies. They state around in case the parent subsequently does a wait. Its exit status is stored in the Linux Process Table, but they take up no other space.  **What happens when a parent terminates before its children?** The children become **orphans**. Linux requires that all processes have a parent, so they are adopted by a special process named init. | **Example7: creating zombies or orphans**  Without a command argument, this program has the parent sleep while the child terminates causing the child to be a zombie.  With a command argument, the child sleeps while the parent terminates causing the child to be an orphan.  #include <stdio.h>  #include <unistd.h>  #include <errno.h>  #include <string.h>  int main (int argc, char \*argv[])  {  long lForkPid;  long lMyVal = 10; // a variable to demonstrate  // that each process has their  // own copy  // create a child process  lForkPid = fork();  // Both the parent and child continue here  switch(lForkPid)  {  case -1:  errExit("fork failed: %s", strerror(errno));  break;  case 0: // child process  printf("Child Process: PID=%ld, PPID=%ld\n"  , (long) getpid(), (long) getppid());  if (argc > 1)  { // child sleeps while parent termiantes  sleep(8);  printf("Child Done after sleep\n");  }  break;  default: // parent process  printf("Parent Process: PID=%ld, PPID=%ld\n"  , (long) getpid(), (long) getppid());  printf("Parent Process: my child's PID=%ld\n"  , lForkPid);  if (argc <= 1)  { // parent sleeps while child terminates  sleep(8);  printf("Parent Done after sleep\n");  }  }  lMyVal += 5;  printf("My PID=%ld, lMyVal=%ld\n"  , (long) getpid(), lMyVal);  return 0;  }  $ gcc -o ex7 example7.c errExit.o |
| **Showing Zombie and Orphan**  **ps** will show a zombie as "<defunct>".  orphans will show a ppid of 1.  In the first invocation of ex7, we have the parent sleep while the child terminates. The child becomes a zombie. ps shows the child as <defunct>.  In the second invocation of ex7, we have the child sleep while the parent terminates. The child becomes an orphan. Its parent is now process 1.  You need to kill the orphan processes. | **Example 7 continued: showing a zombie and an orphan**  **# Child terminates before parent, causing child to become a zombie**  $ ./ex7 &; sleep 1; ps –fu clark  **[1] 17592**  **Parent Process: PID=17592, PPID=12775**  **Parent Process: my child's PID=17594**  **Child Process: PID=17594, PPID=17592**  **My PID=17594, lMyVal=15**  **UID PID PPID C STIME TTY TIME CMD**  **clark 12768 12697 0 Oct01 ? 00:00:00 sshd: clark@pts/1**  **clark 12769 12768 0 Oct01 ? 00:00:00 tcsh -c /usr/lib/openssh/sftp-server**  **clark 12772 12769 0 Oct01 ? 00:00:00 /usr/lib/openssh/sftp-server**  **clark 12775 12768 0 Oct01 pts/1 00:00:00 -tcsh**  **clark 17592 12775 0 08:18 pts/1 00:00:00 ./ex7**  **clark 17594 17592 0 08:18 pts/1 00:00:00 [ex7] <defunct>**  **clark 17595 12775 0 08:19 pts/1 00:00:00 ps -fu clark**  **~/cs3423/ProcessControl>Parent Done after sleep**  **My PID=17592, lMyVal=15**  **# Parent terminates before child, process 1 adopts the child**  $ ./ex7 Child &; sleep 1; ps –fu clark  **[1] 17608**  **Parent Process: PID=17608, PPID=12775**  **Parent Process: my child's PID=17610**  **My PID=17608, lMyVal=15**  **Child Process: PID=17610, PPID=17608**  **UID PID PPID C STIME TTY TIME CMD**  **clark 12768 12697 0 Oct01 ? 00:00:00 sshd: clark@pts/1**  **clark 12769 12768 0 Oct01 ? 00:00:00 tcsh -c /usr/lib/openssh/sftp-server**  **clark 12772 12769 0 Oct01 ? 00:00:00 /usr/lib/openssh/sftp-server**  **clark 12775 12768 0 Oct01 pts/1 00:00:00 -tcsh**  **clark 17610 1 0 08:23 pts/1 00:00:00 ./ex7 Child**  **clark 17611 12775 0 08:23 pts/1 00:00:00 ps -fu clark**  **[1] + Done ./ex7 Child**  **~/cs3423/ProcessControl>Child Done after sleep**  **My PID=17610, lMyVal=15** |
| **Execing**  It is often necessary to create another process which uses a different program from the parent process. This is accomplished by **first forking to a child** and have the **child invoke** one of the **exec** family of functions which causes the process to be overlaid with a different executable. The parent can wait for the child even though the child is a different executable.  int **execl**(char \*szCommand  , const char \*szArg0, ...)  executes the specified fully qualified command using the command arguments. Since this function is passed a variable number of arguments, the last argument must be NULL to enable execl to determine the number of arguments. szArg0 should be the name of the command.  int **execvp**(char \*szCommand, char \*argv[])  executes the specified command using the command argument array. The last element of the array must be NULL to enable execvp to determine the number of arguments. argv[0] should be the name of the command. If the command begins with a slash, it assumes the command is at the specified location. Otherwise, it follows the **path** to find it.  There are several other forms of **exec** functions which you may want to explore. | **Example8: forking a child which uses execl**  #include <stdio.h>  #include <unistd.h>  #include <errno.h>  #include <string.h>  #include <sys/wait.h>  int main ()  {  long lForkPid;  long lWaitPid;  int iExitStatus = 0;  char szInput[20];  // create a child process  lForkPid = fork();  // Both the parent and child continue here  switch(lForkPid)  {  case -1:  errExit("fork failed: %s", strerror(errno));  break;  case 0: // child process  printf("Child Process: PID=%ld, PPID=%ld\n"  , (long) getpid(), (long) getppid());  // invoke a different executable for the child  execl("/bin/ls", "ls", "-l", NULL);  errExit("Child process failed to exec: %s", strerror(errno));  default: // parent process  lWaitPid = wait (&iExitStatus);  if (lWaitPid == -1)  errExit("wait error: %s", strerror(errno));  printf("Parent Process: PID=%ld, PPID=%ld\n"  , (long) getpid(), (long) getppid());  printf("Parent Process: my child's PID=%ld\n"  , lForkPid);  printf("Parent Process: wait pid=%ld\n"  , lWaitPid);  printf("Parent Process: exit status=%d\n"  , iExitStatus);  }  printf("My PID=%ld\n"  , (long) getpid());  return 0;  } |
| **Executing ls as a separate executable.**  Compile and execute it. | $ gcc -o ex8 example8.c errExit.o  $ ./ex8  **Child Process: PID=3066, PPID=3065**  **total 148**  **-rw------- 1 clark faculty 1784 Sep 29 18:08 errExit.c**  **-rw------- 1 clark faculty 1832 Sep 29 18:08 errExit.o**  **-rwx------ 1 clark faculty 9017 Sep 29 18:15 ex4**  **-rwx------ 1 clark faculty 13281 Oct 2 17:08 ex6**  **-rwx------ 1 clark faculty 9118 Oct 4 08:19 ex7**  **-rwx------ 1 clark faculty 9178 Oct 6 09:14 ex8**  **-rw------- 1 clark faculty 999 Sep 29 18:15 example4.c**  **-rw------- 1 clark faculty 1356 Oct 2 17:00 example6.c**  **-rw------- 1 clark faculty 1370 Oct 4 08:18 example7.c**  **-rw------- 1 clark faculty 1379 Oct 6 09:14 example8.c**  **-rw------- 1 clark faculty 1021 Sep 29 18:52 exampleX.c**  **-rwx------ 1 clark faculty 8850 Sep 29 18:11 prtPids**  **-rw------- 1 clark faculty 192 Sep 29 16:12 prtPids.c**  **-rwx------ 1 clark faculty 8849 Sep 29 18:10 prtUser**  **-rw------- 1 clark faculty 181 Sep 29 16:12 prtUser.c**  **Parent Process: PID=3065, PPID=12775**  **Parent Process: my child's PID=3066**  **Parent Process: wait pid=3066**  **Parent Process: exit status=0**  **My PID=3065** |
| **Using execvp to pass a list of arguments** | **Example9: forking a child which uses execvp**  **#include <stdio.h>**  **#include <unistd.h>**  **#include <errno.h>**  **#include <string.h>**  **#include <sys/wait.h>**  **int main ()**  **{**  **long lForkPid;**  **long lWaitPid;**  **int iExitStatus = 0;**  **char szInput[20];**  **char \*execArgv[20];**  **// create a child process**  **lForkPid = fork();**  **// Both the parent and child continue here**  **switch(lForkPid)**  **{**  **case -1:**  **errExit("fork failed: %s", strerror(errno));**  **break;**  **case 0: // child process**  **printf("Child Process: PID=%ld, PPID=%ld\n"**  **, (long) getpid(), (long) getppid());**  **// invoke a different executable for the child**  **execArgv[0] = "ls";**  **execArgv[1] = "-l";**  **execArgv[2] = NULL; <HAVE TO HAVE A NULL AT THE END**  **execvp("ls", execArgv);**  **errExit("Child process failed to exec: %s", strerror(errno));**  **default: // parent process**  **lWaitPid = wait (&iExitStatus);**  **if (lWaitPid == -1)**  **errExit("wait error: %s", strerror(errno));**  **printf("Parent Process: PID=%ld, PPID=%ld\n"**  **, (long) getpid(), (long) getppid());**  **printf("Parent Process: my child's PID=%ld\n"**  **, lForkPid);**  **printf("Parent Process: wait pid=%ld\n"**  **, lWaitPid);**  **printf("Parent Process: exit status=%d\n"**  **, iExitStatus);**  **}**  **printf("My PID=%ld\n"**  **, (long) getpid());**  **return 0;**  **}** |

**©2017 Larry W. Clark and Rocky Slavin,** UTSA CS students may make copies for their personal use