Process Management

Azzam Mourad

www.encs.concordia.ca/~mourad

mourad@encs.concordia.ca

COEN 346

Agenda

- How to create and terminate a process
- Relation between a parent and child process
- The use of fork() and exec() family of functions
- Assignment 2 (part 2)

Assignment 2 (part 2 – a)

- Write a C/C++ program, called *Asg2iia.cpp or Asg2iia.c* that does the following:
  - Executes as a parent process, which occurs naturally.
  - The parent process must output the following statement: “*Parent process is running and about to fork to a child process*”.
  - The parent process must then create a child process (using *fork()*).
  - The child will simply print out to the standard output the following statement: *”I am the child process*”.
  - You are NOT allowed to use the *exec* calls in this part.
  - That is, you must make sure that the child will still run the proper code to perform what it needs to do without the executions of any of the “*exec*” calls.

Assignment 2 (part 2 – a)

- Once the child starts, the parent must wait for the child to die before it continues.
- Output:

```
Parent process is running and about to fork to a child process
I am the child process
Parent acknowledges child termination Parent will terminate now
```

Assignment 2 (part 2 – b)

- Write a C/C++ program, called "*outsider.cpp*" or "*outsider .c*" that outputs the following statement: "Outsider program is running.
- Write a C/C++ program called *Asg2iib.cpp or Asg2iib.c*, which is similar to the one you created in Part II-A above, with the following exceptions:
  - The child process must execute the code of the *Outsider* program using the *exec* system call
- Output:

```
Parent process is running and about to fork to a child process
Outsider program is running. Time now is Mon Jan 29 01:16:26 EST 2007
Parent acknowledges child termination Parent will terminate now
```

Process Management

- A process is created for you program when you run it from a shell
- This is the parent process
- You can create child processes inside the program using the fork() command

Process Creation

- The fork() system call will spawn a new child process which is an identical process to the parent except that has a new system process ID.
- The process is copied in memory from the parent and a new process structure is assigned by the kernel.
- The return value of the function is which discriminates the two threads of execution. A zero is returned by the fork function in the child's process.
- The environment, resource limits, controlling terminal, current working directory, root directory and other process resources are also duplicated from the parent in the forked child process.

Process Creation (vfork)

- The vfork() function is the same as fork() except that it does not make a copy of the address space.
- The memory is shared reducing the overhead of spawning a new process with a unique copy of all the memory.
- The vfork() function also executes the child process first and resumes the parent process when the child terminates.

Process Creation using fork()

```
#include <sys/types.h>
#include <unistd.h>
using namespace std;

main() {
pid_t pID = fork();
if (pID == 0) // child
{ // Code only executed by child process}
else if (pID < 0) // failed to fork
{ cerr << "Failed to fork" << endl; exit(1);}
else // parent
{ // Code only executed by parent process}
// Code executed by both parent and child
}
```

Process Termination

- The C library function exit() calls the kernel system call _exit() internally.
- The kernel system call _exit() will cause the kernel to close descriptors, free memory, and perform the kernel terminating process clean-up.
- The C library function exit() call will flush I/O buffers and perform additional clean-up before calling _exit() internally.
- The function exit(*status*) causes the executable to return "status" .
- The parent process can examine the terminating status of the child.
- The parent process will often want to wait until all child processes have been completed using the wait() function call

exec family of functions

- The exec() family of functions will initiate a program from within a program.

- The functions return an integer error code.

  (0=Ok / -1=Fail)

execl

- The function call "execl()" initiates a new program in the same environment in which it is operating.
- An executable (with fully qualified path. i.e. /bin/ls) and arguments are passed to the function.
- int execl(const char *path, const char *arg1, const char *arg2, ... const char *argn, (char *) 0);
- #include <unistd.h>
- main() { execl("/bin/ls", "-r", "-t", "-l", (char *) 0); }
- All function arguments are null terminated strings. The list of arguments is terminated by NULL.

execlp

- The routine execlp() will perform the same as execl except that it will use environment variable PATH to determine which executable to process.
- Thus a fully qualified path name would not have to be used.
- The first argument to the function could instead be "ls".

execv

- This is the same as execl() except that the arguments are passed as null terminated array of pointers to char.
- int execv(const char *path, char *const argv[]);
- #include <unistd.h>
- main() {
- char *args[] = {"-r", "-t", "-l", (char *) 0 }
- execv("/bin/ls", args);}

execvp

- The routine execvp() will perform the same execv except that it will use environment variable PATH to determine which executable to process.
- Thus a fully qualified path name would not have to be used.
- The first argument to the function could instead be "ls".

execve

- The function call "execve()" executes a process in an environment which it assigns.

- **Set the environment variables:**

  Assignment:

  - char *env[] = { "USER=*user1*", "PATH=/usr/bin:/bin:/opt/bin", (char *) 0 };

- char *Env_argv[] = { "/bin/ls", "-l", "-a", (char *) 0 };

- execve (Env_argv[] , Env_argv, Env_envp);