Parallel and Concurrent Programming CS3S666

Pipes and Redirection

Last Time

- Execl()
  - Used to replace the program in the current process with the targeted program.
- Pipe()
  - Used to create a pipe for reading and writing.

This Time…

- A deeper look into Pipes and File Descriptors
- Redirecting Pipes for Inter Process communication

What is a File Descriptor?

- A file descriptor is an integer that represents a particular I/O source/target.

- There are three file descriptors that every process creates:
  - 0 – stdin (cin in C++)
  - 1 – stdout (cout in C++)
  - 2 - stderr (cerr in C++)

File Descriptors

```cpp
#include<cstdlib>
#include<iostream>
#include<unistd.h>
#include <sys/wait.h>

using namespace std;

int main(void)
{
  int x;

  cout << getpid() << endl;

  cin >> x;

  return 0;
}
```

Compile and run and then open another terminal and type:
“cd /proc/your_process_number/fd
And then type in ls -lrt

```
simon@simon-VirtualBox:~$ cd /proc/7923/fd
simon@simon-VirtualBox:/proc/7923/fd$ ls -lrt
total 0
lrwx------ 1 simon simon 64 Jul 22 13:54 2 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 13:54 1 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 13:54 0 -> /dev/pts/0
simon@simon-VirtualBox:/proc/7923/fd$
```

Shows current file descriptors.

File Descriptors

```cpp
#include<cstdlib>
#include<iostream>
#include<unistd.h>
#include <sys/wait.h>

using namespace std;

int main(void)
{
  int x;

  int myPipes[2];

  if (pipe(myPipes) < 0) exit(1);

  cout << getpid() << endl;

  cin >> x;

  return 0;
}

```

```
simon@simon-VirtualBox:/proc/7923/fd$ cd /proc/7946/fd
simon@simon-VirtualBox:/proc/7946/fd$ ls -lrt
total 0
l-wx------ 1 simon simon 64 Jul 22 13:56 4 -> 'pipe:[48092]'
lr-x------ 1 simon simon 64 Jul 22 13:56 3 -> 'pipe:[48092]'
lrwx------ 1 simon simon 64 Jul 22 13:56 2 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 13:56 1 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 13:56 0 -> /dev/pts/0
simon@simon-VirtualBox:/proc/7946/fd$
```

The pipes are also shown.

File Descriptors

```cpp
#include<cstdlib>
#include<iostream>
#include<unistd.h>
#include <sys/wait.h>

using namespace std;

int main(void)
{
  int x;

  int myPipes[2];

  if (pipe(myPipes) < 0) exit(1);

  x = fork();

  cout << getpid() << endl;

  cin >> x;

  return 0;
}
```

- Does a child process inherit the file descriptors?

```
simon@simon-VirtualBox:~$ g++ -o Example Example.cpp
simon@simon-VirtualBox:~$ ./Example
7971
7972
```

```
simon@simon-VirtualBox:/proc/7972/fd$ cd /proc/7971/fd
simon@simon-VirtualBox:/proc/7971/fd$ ls -lrt
total 0
l-wx------ 1 simon simon 64 Jul 22 13:58 4 -> 'pipe:[48331]'
lr-x------ 1 simon simon 64 Jul 22 13:58 3 -> 'pipe:[48331]'
lrwx------ 1 simon simon 64 Jul 22 13:58 2 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 13:58 1 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 13:58 0 -> /dev/pts/0
simon@simon-VirtualBox:/proc/7971/fd$ cd /proc/7972/fd
simon@simon-VirtualBox:/proc/7972/fd$ ls -lrt
total 0
l-wx------ 1 simon simon 64 Jul 22 13:58 4 -> 'pipe:[48331]'
lr-x------ 1 simon simon 64 Jul 22 13:58 3 -> 'pipe:[48331]'
lrwx------ 1 simon simon 64 Jul 22 13:58 2 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 13:58 1 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 13:58 0 -> /dev/pts/0
simon@simon-VirtualBox:/proc/7972/fd$
```

Appear to be the same as expected.

What about if we use execl()?

```cpp
#include<cstdlib>
#include<iostream>
#include<unistd.h>
#include <sys/wait.h>

using namespace std;

int main(void)
{
  int x;

  int myPipes[2];

  if (pipe(myPipes) < 0) exit(1);

  x = fork();
  cout << getpid() << endl;

  if (x == 0)
  {
    if (execl("./LP", "LP", NULL) < 0)
      cout << "Error" << endl;
  }
  cin >> x;

  return 0;
}
```

```cpp
#include<iostream>
#include<unistd.h>
#include <stdlib.h>

using namespace std;

int main(int argc, const char * argv[])
{
  int x = 0;
  cin >> x;
  exit(0);
}
```

```
simon@simon-VirtualBox:~$ g++ -o Example Example.cpp
simon@simon-VirtualBox:~$ ./Example
8016
8017
```

```
simon@simon-VirtualBox:/proc/7972/fd$ cd /proc/8016/fd
simon@simon-VirtualBox:/proc/8016/fd$ ls -lrt
total 0
l-wx------ 1 simon simon 64 Jul 22 14:05 4 -> 'pipe:[48646]'
lr-x------ 1 simon simon 64 Jul 22 14:05 3 -> 'pipe:[48646]'
lrwx------ 1 simon simon 64 Jul 22 14:05 2 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 14:05 1 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 14:05 0 -> /dev/pts/0
simon@simon-VirtualBox:/proc/8016/fd$ cd /proc/8017/fd
simon@simon-VirtualBox:/proc/8017/fd$ ls -lrt
total 0
l-wx------ 1 simon simon 64 Jul 22 14:06 4 -> 'pipe:[48646]'
lr-x------ 1 simon simon 64 Jul 22 14:06 3 -> 'pipe:[48646]'
lrwx------ 1 simon simon 64 Jul 22 14:06 2 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 14:06 1 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 14:06 0 -> /dev/pts/0
simon@simon-VirtualBox:/proc/8017/fd$
```

```cpp
#include<cstdlib>
#include<iostream>
#include<unistd.h>
#include <sys/wait.h>

using namespace std;

int main(void)
{
  int x;

  int myPipes[2];

  if (pipe(myPipes) < 0) exit(1);

  x = fork();

  if (x == 0)
  {
    if (execl("./LP", "LP", NULL) < 0)
    cout << "Error" << endl;
  }

  string name;
  cout << "Enter something:";
  cin >> name;
  close(myPipes[0]);
  write(myPipes[1], name.c_str(), name.length());

  wait(NULL);

  return 0;
}
```

Using a pipe in the child Process:

```cpp
#include<iostream>
#include<unistd.h>
#include <stdlib.h>

using namespace std;

int main(int argc, const char * argv[])
{
int x = 3;

char buffer[256];

read(x, buffer, 256);
cout << "I got: " << buffer << endl;

exit(0);
}
```

```
[Simons-MacBook-Pro:Desktop$ atom LittleProg.cpp
[Simons-MacBook-Pro:Desktop$ g++ -o LP LittleProg.cpp
[Simons-MacBook-Pro:Desktop$ g++ -o Examples examples.cpp
[Simons-MacBook-Pro:Desktop$ ./Examples
Enter something:Hello
I got: Hello
Simons-MacBook-Pro:Desktop$
```

Using a Pipe in Child Process

- it is possible to use an existing pipe even after using execl()
- But getting the child to guess the number is a bit risky

- Better Way ?

dup()

- Creates a duplicate of the file descriptor.
- New file descriptor will be lowest available.

```
int dup(int oldfd)
```

```cpp
#include<cstdlib>
#include<iostream>
#include<unistd.h>
#include <sys/wait.h>

using namespace std;

int main(void)
{
  int pid1, pid2;

  int myPipes[2];

  if (pipe(myPipes) < 0) exit(1);

  pid1 = fork();

  int duplicateFd = dup(myPipes[0]);
  char buffer[6];
  buffer[5] = '\0';

  if (pid1 == 0)
  {
    read(myPipes[0], buffer, 5);
    cout << "Child 1:" << buffer << endl;
  }
  else
  {
    pid2 = fork();
    if (pid2 != 0)
    {
      write(myPipes[1], "HelloWorld", 11);
      waitpid(pid1, NULL, 0);
      waitpid(pid2, NULL, 0);
    }
    else
    {
      read(duplicateFd, buffer, 5);
      cout << "Child 2:" << buffer << endl;
    }
  }
  return 0;
}

```

```
[Simons-MacBook-Pro:Desktop$ g++ -o Examples examples.cpp
[Simons-MacBook-Pro:Desktop$ ./Examples
Child 1:Hello
Child 2:World
Simons-MacBook-Pro:Desktop$
```

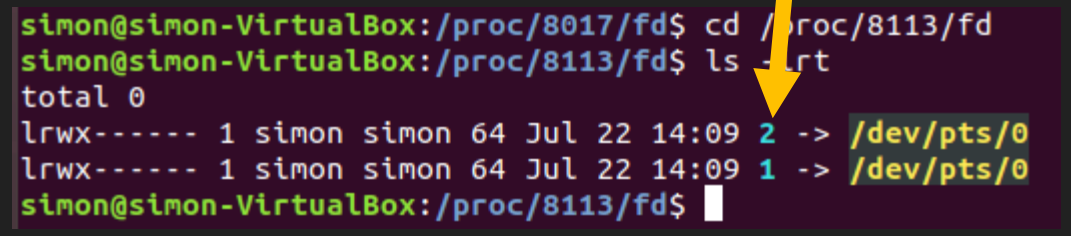
Why is dup() useful?

- To understand how this is useful we need to check what happens if we call close()?
- In particular if we call close on stdin/stdout?

A look at close()

```
1   #include<cstdlib>
2   #include<iostream>
3   #include<unistd.h>
4   #include <sys/wait.h>
5
6   using namespace std;
7
8   int main(void)
9   {
10    close(0);
11    cout << getpid() << endl;
12    sleep(60);
13    return(0);
14  }
15
```

Notice 0 has gone.



Using dup()

```
1  #include<cstdlib>
2  #include<iostream>
3  #include<unistd.h>
4  #include <sys/wait.h>
5
6  using namespace std;
7
8  int main(void)
9  {
10
11   close(0);
12   cout << getpid() << endl;
13   int myPipe[2];
14
15   if (pipe(myPipe) < 0) exit(-1);
16
17   dup(myPipe[0]);
18
19   sleep(60);
20   return(0);
21 }
22
```

Notice 0 has now been assigned to the pipe.

```
simon@simon-VirtualBox:/proc/8113/fd$ cd /proc/8124/fd
simon@simon-VirtualBox:/proc/8124/fd$ ls -lrt
total 0
lr-x------ 1 simon simon 64 Jul 22 14:11 4 -> 'pipe:[49072]'
l-wx------ 1 simon simon 64 Jul 22 14:11 3 -> 'pipe:[49072]'
lrwx------ 1 simon simon 64 Jul 22 14:11 2 -> /dev/pts/0
lrwx------ 1 simon simon 64 Jul 22 14:11 1 -> /dev/pts/0
lr-x------ 1 simon simon 64 Jul 22 14:11 0 -> 'pipe:[49072]'
simon@simon-VirtualBox:/proc/8124/fd$
```

Putting it all

```cpp
#include<cstdlib>
#include<iostream>
#include<unistd.h>
#include <sys/wait.h>

using namespace std;

int main(void)
{

    close(0);
    int myPipe[2];

    if(pipe(myPipe) < 0) exit(-1);

    dup(myPipe[0]);

    int x = fork();

    if(x == 0)
    {
       if(execl("./LP","LP",NULL) < 0);
           exit(-1)
    }
     else
     {
       close(0);
       write(myPipe[1], "HelloFriend\n",12);
       close(1);
       wait(NULL);
     }

  return 0;
}
```

```cpp
#include<iostream>
#include<unistd.h>
#include <stdlib.h>

using namespace std;

int main(int argc, const char * argv[])
{
string message;

cin >> message;
cout << "My message: " << message << endl;

exit(0);
}
```

This is called redirection.

```
Simons-MacBook-Pro:Desktop$ g++ -o Examples examples.cpp
Simons-MacBook-Pro:Desktop$ g++ -o LP LittleProg.cpp
Simons-MacBook-Pro:Desktop$ ./Examples
My message: HelloFriend
Simons-MacBook-Pro:Desktop$
```

\n is important, as it tells cin that the string has ended

Summary

- We’ve learnt more about file descriptors.
- We understand more about the mechanics behind standard input/output/errors.
- We’ve seen how to use dup() to redirect data so it can be accessed by a child process.

  Step 1: Close file descriptor you want to redirect to.

  Step2: Call dup with file descriptor of pipe you want to redirect.

Next Time

- Named pipes
- This weeks tutorial is about pipes and redirection