Parallel and Concurrent Programming CS3S666

Last Time

- Difference between concurrent and parallel.
- Fork()
  - Used to create a new process
  - Returns?

This Time

- Using execute
  - Used to specify a new program to execute
- Introduction to Pipes
  - Used to inter process communication

Processes and Programs

- A process is something that allows a program to be executed.
  - Fork() creates a new process with the same program as the parent process (and it’s state).
- Sometimes we want a process to be able to change the program it is executing.
- In Unix we do this using exec()*

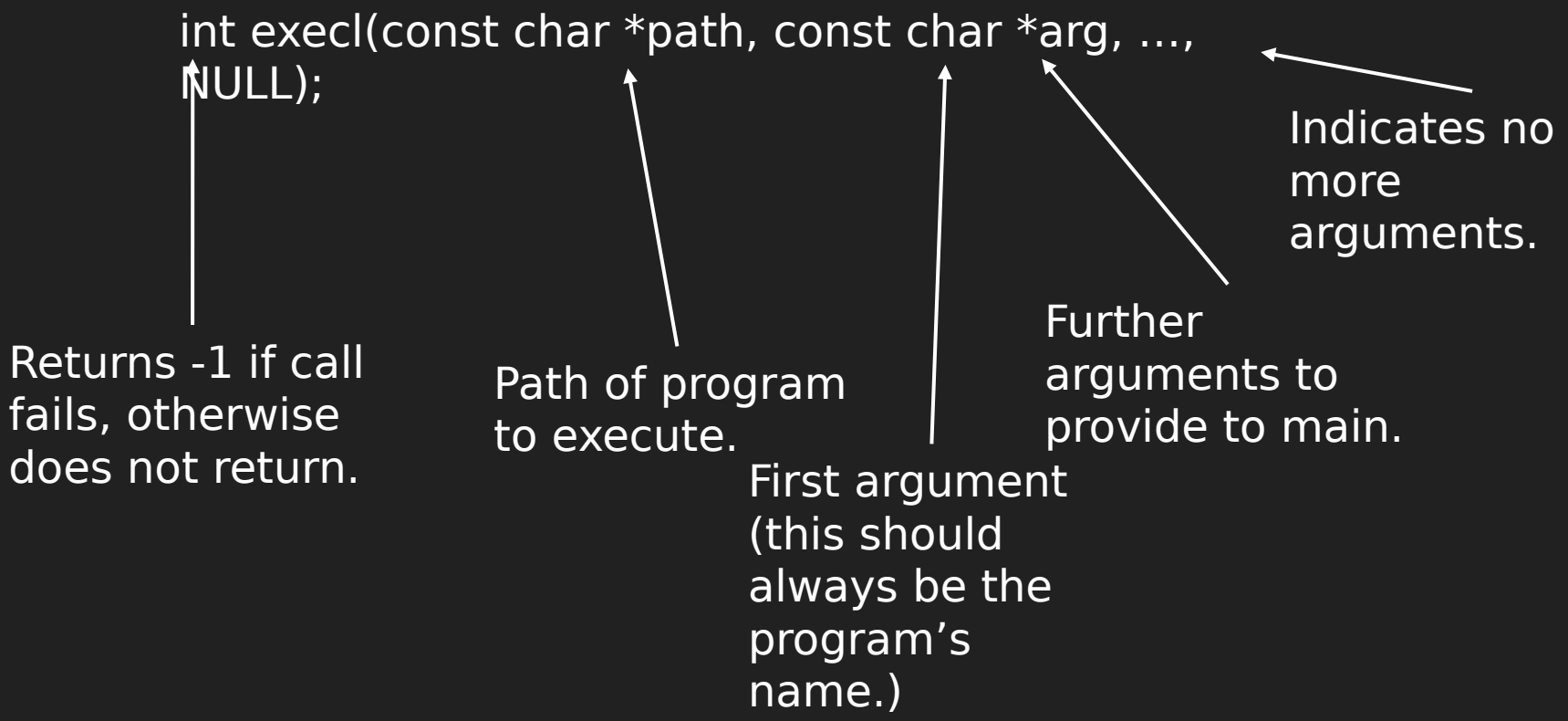
*There are many variants of this to use.

Exec()

- Executes a program from the entry point (this differs to fork).
- Provided it is successful it overwrites the existing program.
- There are many different flavours.

Exec()

- We will use execl:



Last Time

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

int main()
{
    pid_t x = fork();

    if (x == 0)
    {
        std::cout << "I am a child" << std::endl;
    }
    else
    {
        std::cout << "I am a parent" << std::endl;
    }
    return 0;
}
```

```
[eduroamtf-4039:Desktop$ g++ -o Examples examples.cpp
[eduroamtf-4039:Desktop$ ./Examples
i am a parent
I am a child
eduroamtf-4039:Desktop$
```

Using Execl()

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

int main()
{
   pid_t x = fork();

   if (x == 0)
   {
      execl("./LittleProg", "LittleProg", NULL);
      std::cout << "I'm a child" << std::endl;
   }
   else
   {
      std::cout << "I'm a parent." << std::endl;
   }

   return 0;
}
```

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

using namespace std;

int main()
{
   cout << "I'm a little program" <<
endl;
   return 0;
}
```

Compiled to LittleProg.

```
[eduroamtf-4039:Desktop$ g++ -o Examples examples.cpp
[eduroamtf-4039:Desktop$ g++ -o LittleProg LittleProg.cpp
[eduroamtf-4039:Desktop$ ./Examples
 I'm a parent
 I'm a little program
 eduroamtf-4039:Desktop$
```

Note “I’m a child” **not** printed. Why?

Passing Arguments

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

int main()
{
  pid_t x = fork();

  if (x == 0)
  {
    execl("./LP", "LP", "I was the child", "Now I am not",
NULL);
    std::cout << "I’m a child" << std::endl;
  }
  else
  {
    std::cout << "I’m a parent." << std::endl;
  }

  return 0;
}
```

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

using namespace std;

int main(int argc, const char *argv[])
{
  cout << argv[1] << endl;
  cout << argv[2] << endl;
  return 0;
}
```

Compiled to LP.

```
[eduroamtf-4039:Desktop$ g++ -o LP LittleProg.cpp
[eduroamtf-4039:Desktop$ g++ -o Examples examples.cpp
[eduroamtf-4039:Desktop$ ./Examples
 I'm a parent
 I was the child
 Now I am not
 eduroamtf-4039:Desktop$
```

Zombie Processes

- When a child process completes it can pass a small amount of data back to the parent.
- This means the process still exists until that information has been collected.
- The function call wait() can be used to ensure the parent waits for the child to finish and allow the OS to clean up the child process.
- If parent fails to wait, small amounts of data is left.

Zombie Processes

- On Unix and Unix-like computer operating systems, a zombie process or defunct process is a process that has completed execution (via the exit system call) but still has an entry in the process table: it is a process in the "Terminated state".

- The process is dead, but still has an entry in the process table
  - They don’t use any resources, but do retain PID.
  - The parent process needs to wait() the child processes.
  - A Zombie process can not be killed using kill command

Pid_t wait(int *status)

- Called by parent process.
- Will block parent until ***a*** child completes.
- Returns pid of child process.
- Accepts status that has two parts, the exit parameter and the status.

Terminating a Process

- To terminate a process, use exit(int status)
- The status will be passed back to calling process.

Using Wait

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

int main()
{
  pid_t x = fork();

  if (x == 0)
  {
    execl("./LP", "LP", "I was the child", "Now I am not",
NULL);
    std::cout << "I’m a child" << std::endl;
  }
  else
  {
    std::cout << "I’m a parent." << std::endl;
  }

  int status;
  pid_t child = wait(&status);

  if (child == x)
  {
    std::cout << "My child has terminated" << std::endl;
  }

  return 0;
}
```

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

using namespace std;

int main(int argc, const char *argv[])
{
  cout << argv[1] << endl;
  cout << argv[2] << endl;
  sleep(5);
  return 0;
}
```

```
eduroamtf-4039:Desktop$ g++ -o Examples examples.cpp
eduroamtf-4039:Desktop$ ./Examples
I'm a parent
I was the child
Now I am not
My child has terminated
eduroamtf-4039:Desktop$
```

Passing a value back

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

int main()
{
  pid_t x = fork();

  if (x == 0)
  {
   execl("./LP", "LP", "I was the child", "Now I am    not", NULL);
   std::cout << "I'm child" << std::endl;
  }
  else
  {
    std::cout << "I'm parent." << std::endl;
  }

  int status;
  pid_t child = wait(&status);

  if (child == x)
  {
    std::cout << "My child has terminated" << std::endl;
    std::cout << "Val returned = " << WEXITSTATUS(status) << std::endl;
  }

  return 0;
}
```

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

using namespace std;

int main(int argc, const char *argv[])
{
  cout << argv[1] << endl;
  cout << argv[2] << endl;
  sleep(1);
  exit(5);
}
```

```
[eduroamtf-4039:Desktop$ g++ -o LP LittleProg.cpp
[eduroamtf-4039:Desktop$ g++ -o Examples examples.cpp
[eduroamtf-4039:Desktop$ ./Examples
I'm a parent
I was the child
Now I am not
My child has terminated
Val returned = 5
eduroamtf-4039:Desktop$
```

Pipes

- Pipes are used to pass data between processes.

```
int p[2];

pipe(p) // returns -1 if fails
```

Accepts integer array of size 2. To populate with file descriptor.

- p[0] is read end of pipe.

- P[1] is the write end of the pipe.

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

using namespace std;

int main()
{
     const char *msg1 = "Hello";
     const char *msg2 = "It's me";

     int p[2];

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

     pid_t x = fork();

     if (x == 0)
     {
          close(p[1]);
          char inbuff[8];
          read(p[0], inbuff, 6);
          std::cout << inbuff << endl;
          read(p[0], inbuff, 8);
          std::cout << inbuff << endl;
     }
     else
     {
          close(p[0]);
          write(p[1], msg1, 6);
          write(p[1], msg2, 8);
          wait(0);
     }

     return 0;
}
```

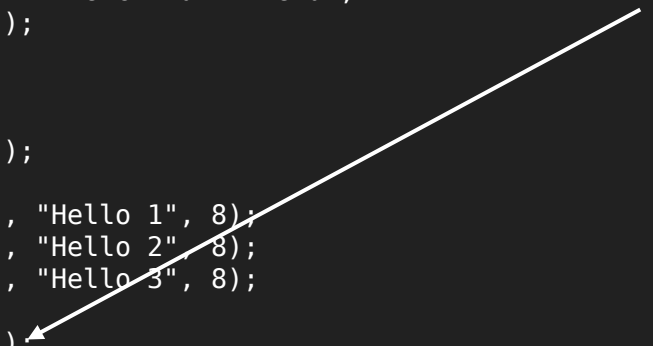
```
[eduroamtf-166:Desktop$ g++ -o Examples examples.cpp
[eduroamtf-166:Desktop$ ./Examples
Hello
It's me
eduroamtf-166:Desktop$
```

Always a good idea to close pipes we are not using.

How do we use pipes across different programs? That’s for next time!

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

using namespace std;

int main(void)
{

  int p[2];

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

  pid_t x = fork();

  if (x == 0)
  {
    close(p[1]);
    char buffer[8];
    int n;

    while ((n = read(p[0], buffer, 8)) > 0)
    {
      cout << "You said: " << buffer << endl;
    }
    cout << "I'm the child" << endl;
    close(p[0]);
  }
  else
  {
    close(p[0]);

    write(p[1], "Hello 1", 8);
    write(p[1], "Hello 2", 8);
    write(p[1], "Hello 3", 8);

    close(p[1]);
    cout << "I'm the parent" << endl;
    wait(NULL);
  }

  return 0;
}
```

```
eduroamtf-166:Desktop$ g++ -o Examples examples.cpp
eduroamtf-166:Desktop$ ./Examples
I'm the parent
You said: Hello 1
You said: Hello 2
You said: Hello 3
I'm the child
eduroamtf-166:Desktop$
```



What happens if we forget to close p[1]?

Summary

- Typical method to create a new process:
  - Fork() followed by Exec(…).

- Pipes used to communicate between processes.

```
int p[2];

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

read(p[0], inbuff, num_bytes); //Read data from
pipe[0].

write(p[1], outbuff, num_bytes); //Send data into
pipe[1]
```

Next Time

- Redirecting Pipes.
  - Currently pipe can only be used in the current program.
- Named Pipes.
- This weeks tutorial is about using Pipes and Exec()