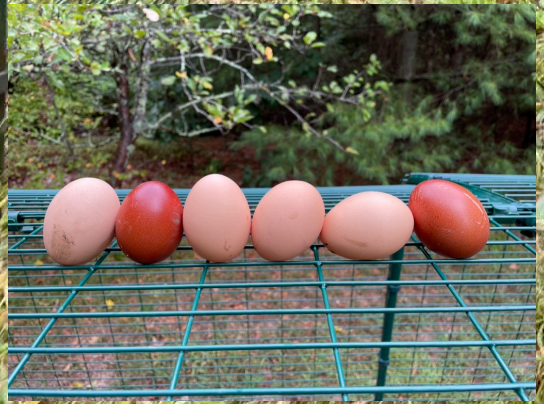
Join on Zoom if you are remote (see website for link).

377 Operating Systems

Process API

Which came first, the chicken or the egg?



CRUX: HOW TO CREATE AND CONTROL PROCESSES

What interfaces should the OS present for process creation and control? How should these interfaces be designed to enable ease of use as well as utility?

Creating New Processes

- UNIX presents one of the most intriguing ways to create new processes
- **fork():** create a new child process
- **exec():** replace the “image” of a process with another

C p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}

```

Here is an example program using **fork**.

When you run this program, you will see something like this…

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
```

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

```c
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

Here is an example program using **fork**.

When you run this program, you will see something like this…

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

First, we include a bunch of header files.

Aside, how do we know which header file to include anyway?

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}

```

First, we include a bunch of header files.

Aside, how do we know which header file to include anyway?

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

First, we include a bunch of header files.

Aside, how do we know which header file to include anyway?

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

First, we include a bunch of header files.

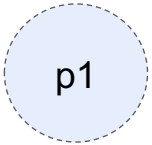
Aside, how do we know which header file to include anyway?

```
prompt> ./p1
```

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

The entry point for the process that we starts when we run a program from the command line.

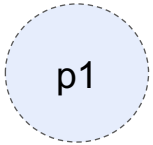


hello world (pid:29146)

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}

```

Here, we print the process id (pid) of the currently executing process.



hello world (pid:29146)

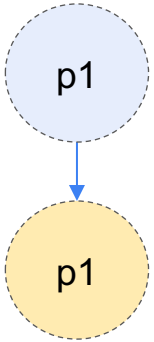
p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

A call to **fork**.

Invokes code in the OS to create a new process (PCB).

It creates a child.

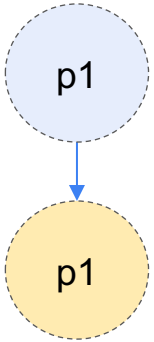


```
hello world (pid:29146)
```

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

Execution continues in both the parent and child process from the point of return from **fork**.

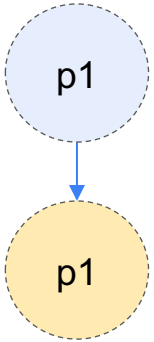


hello world (pid:29146)

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

It is non-deterministic as to which process will “have” the CPU next.



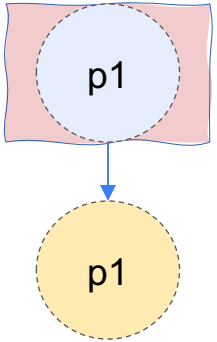
```
hello world (pid:29146)
```

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

Let us assume the parent continues to execute.

It checks the return value – always a good thing to do.



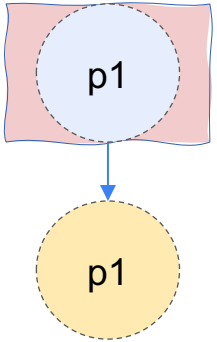
```
hello world (pid:29146)
```

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

We check the return code.
A non-zero value indicates that it is the parent.

It is the PID of the child.



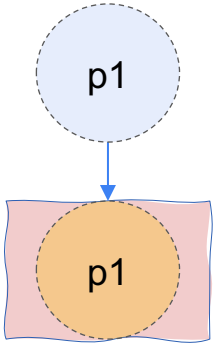
```
hello world (pid:29146)
```

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

At this point, there is a context switch.

Why?



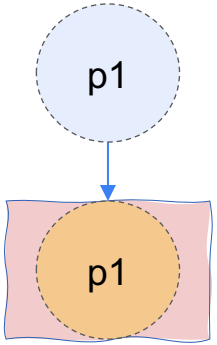
```
hello world (pid:29146)
```

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

The child starts to execute.
It checks the return value, but it never fails in the child.

We just won't reach this point.

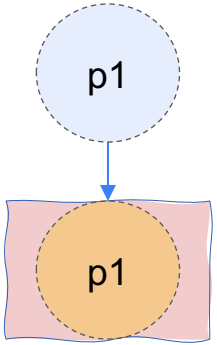


```
hello world (pid:29146)
```

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

We check the return code. If it is 0, which in this case it is, we are executing in the child process.



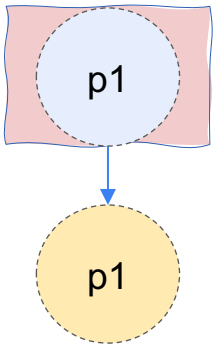
hello world (pid:29146)

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

Right before we make the call to **printf**, there is a context switch.

Why?

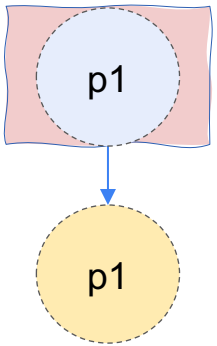


```
hello, I am parent of 29147 (pid:29146)
```

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

The parent invokes **printf**.



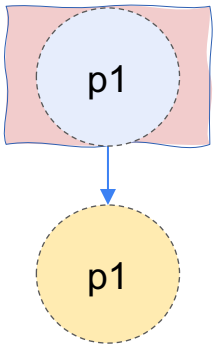
```
hello, I am parent of 29147 (pid:29146)
```

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

The parent returns from main.

This puts the parent process into a waiting status for the child automatically.



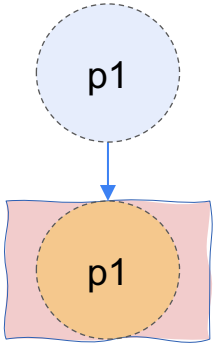
```
hello, I am parent of 29147 (pid:29146)
```

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

The child resumes execution at some point.

READY -> RUNNING

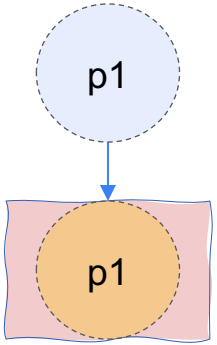


hello, I am child (pid:29147)

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

Calls printf.



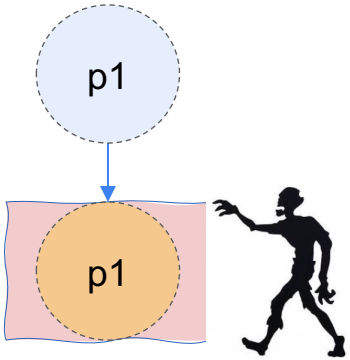
```
hello, I am child (pid:29147)
```

C p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

Returns from main.

This creates a **zombie** process.

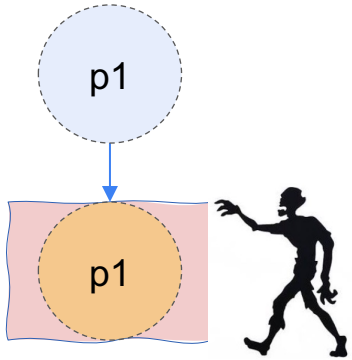


hello, I am child (pid:29147)

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}

```

The parent process has already completed executing, so the OS is going to reap the child.



hello, I am child (pid:29147)

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}

```

The parent process has already completed executing, so the OS is going to reap the child.

p1

hello, I am child (pid:29147)

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}

```

Although the parent has exited, it will not necessarily automatically be deallocated.

Why?

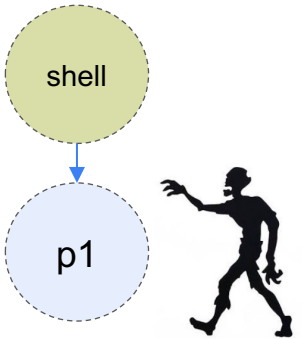
hello, I am child (pid:29147)

p1.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
  } else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

The shell will handle the reaping of the parent.

Unless it wasn't the shell that created it…



p2.c > ...

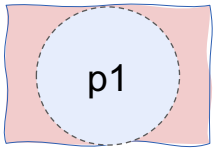
```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

Here is a slight variant to the previous program.

p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

We begin execution in main.



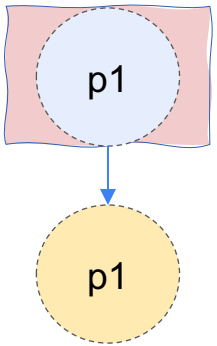
p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

We print.

Then call **fork**.

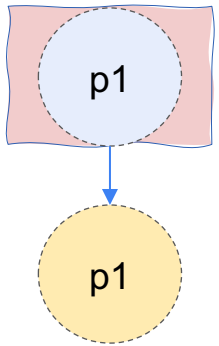
Creates a child.



p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

We check for errors.

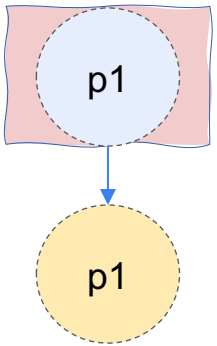


p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

We check if we are the child.

We are not.



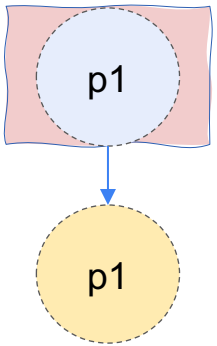
p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

We call **wait.**

This will cause the parent process to “sleep”.

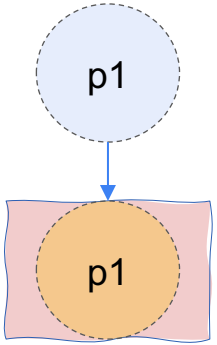
Waiting for the child to complete.



p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

We context switch to the child (at some point later in time).

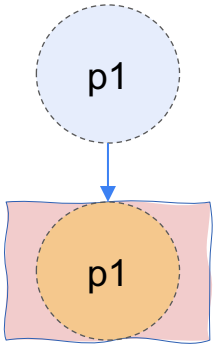


p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

We return from **fork** and check return codes.

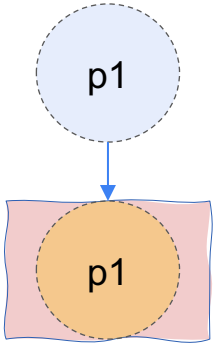
If we get to this point, there is no error in the child.



C p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

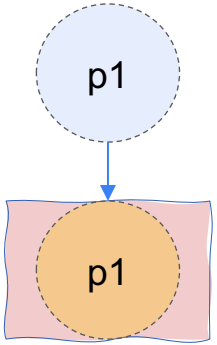
We are running the child process, so our return value from **fork** is 0.



p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

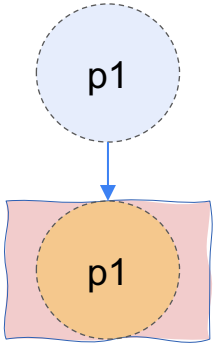
We print.



p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

We print.



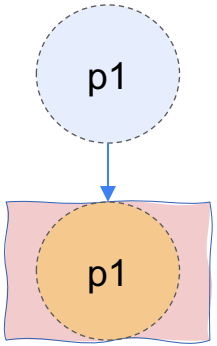
p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

Next, we put the process to sleep for 1 second.

This will allow other processes to execute.

But not the parent. Why?



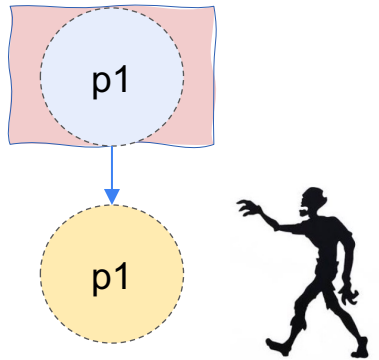
p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

We return from sleep.

Then return from main.

We now have a zombie child.



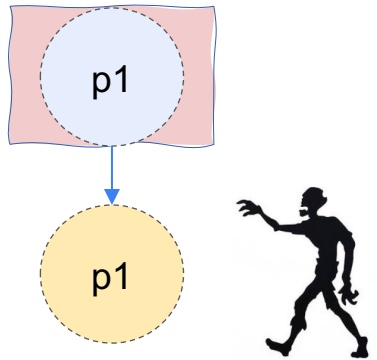
p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

Eventually, the parent executes again.

We return from wait since the parent has a child that has exited.

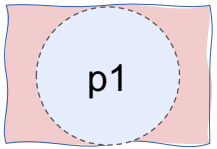
This kills the zombie.



p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

The parent prints.

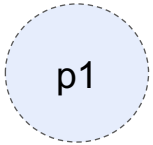


p2.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

The parent prints.

And finally returns from main.

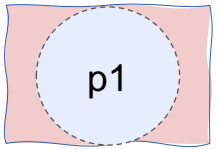


p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

Now, let's do something cool.

We once again start in main.

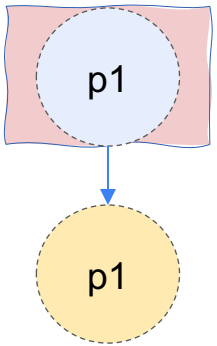


p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

The parent process prints.

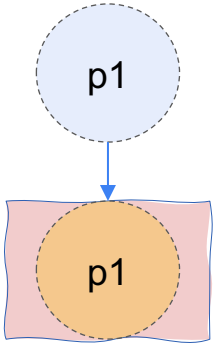
It then calls **fork**.



p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

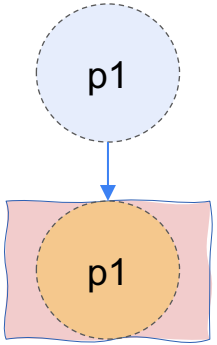
Imagine we context switch to the child.



p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

We print.

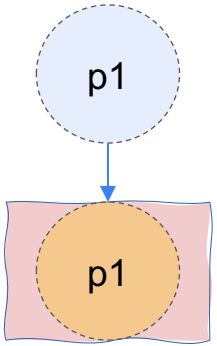


p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

Next, we create an array of pointers to characters (aka strings).

We duplicate string literals and assign them to particular positions in the array.



p3.c > ...

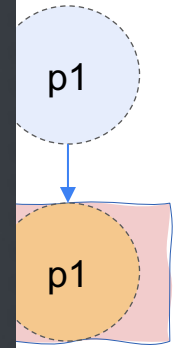
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait
#include <unistd.h>

int main(int argc,
  printf("hello wo
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr
    exit(1);
  } else if (rc ==
    // child (new
    printf("hello,
    char *myargs[3
    myargs[0] = st
    myargs[1] = st
    myargs[2] = NU
    execvp(myargs[
    printf("this s
  } else {
    // parent goes
    int wc = wait(
    printf("hello,
        (int)ge
  }
  return 0;
}
```

man strdup

```
STRDUP(3)                  Linux Programmer's Manual                 STRDUP(3)

NAME
       strdup, strndup, strdupa, strndupa - duplicate a string

SYNOPSIS
       #include <string.h>

       char *strdup(const char *s);

       char *strndup(const char *s, size_t n);
       char *strdupa(const char *s);
       char *strndupa(const char *s, size_t n);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       strdup():
           _XOPEN_SOURCE >= 500
               || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
               || /* Glibc versions <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
       strndup():
           Since glibc 2.10:
               _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
               _GNU_SOURCE
       strdupa(), strndupa(): _GNU_SOURCE

DESCRIPTION
       The  strdup() function returns a pointer to a new string which is a du-
       plicate of the string s.  Memory for the new string  is  obtained  with
       malloc(3), and can be freed with free(3).

Manual page strdup(3) line 1 (press h for help or q to quit)
```

gs?

what is



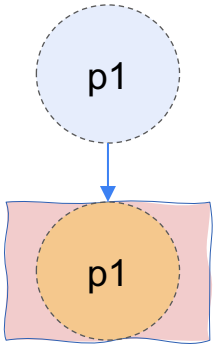
p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

What are these strings?

These strings are the program we want to execute **in place** of the currently executing process.

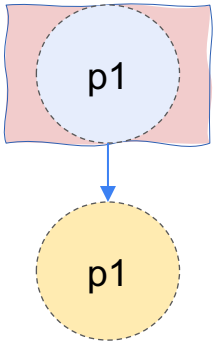
- wc: the “word count” program
- p3.c: the argument to wc
- NULL: the end of the args



p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

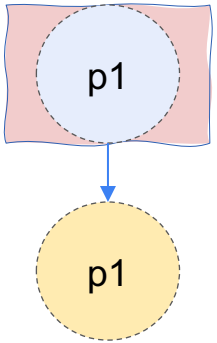
We are just about to execute the next line in the child, but we have a context switch.



p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

This process is not the child, so we continue.

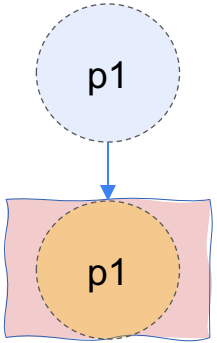


p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

We call wait and, well, wait.

We then context switch back to the child.



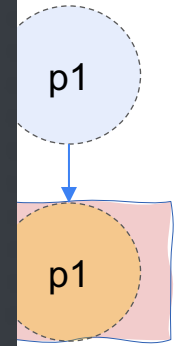
p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait
#include <unistd.h>

int main(int argc,
  printf("hello wo
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr
    exit(1);
  } else if (rc ==
    // child (new
    printf("hello,
    char *myargs[3
    myargs[0] = st
    myargs[1] = st
    myargs[2] = NU
    execvp(myargs[
    printf("this s
  } else {
    // parent goes
    int wc = wait(
    printf("hello,
           (int)ge
  }
  return 0;
}
```

man execvp

```
EXEC(3)                  Linux Programmer's Manual                 EXEC(3)

NAME
       execl, execlp, execle, execv, execvp, execvpe - execute a file

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       int execl(const char *pathname, const char *arg, ...
                       /* (char  *) NULL */);
       int execlp(const char *file, const char *arg, ...
                       /* (char  *) NULL */);
       int execle(const char *pathname, const char *arg, ...
                       /*, (char *) NULL, char *const envp[] */);
       int execv(const char *pathname, char *const argv[]);
       int execvp(const char *file, char *const argv[]);
       int execvpe(const char *file, char *const argv[],
                       char *const envp[]);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       execvpe(): _GNU_SOURCE

DESCRIPTION
       The  exec() family of functions replaces the current process image with
       a new process image.  The functions described in this manual  page  are
       front-ends  for execve(2).  (See the manual page for execve(2) for fur-
       ther details about the replacement of the current process image.)

       The initial argument for these functions is the name of a file that  is
Manual page execvp(3) line 1 (press h for help or q to quit)
```

ls the **execvp**
t the string
array
er to the start
st and second
vely.



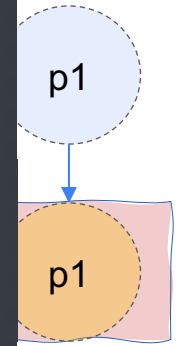
p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait
#include <unistd.h>

int main(int argc,
  printf("hello wo
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr
    exit(1);
  } else if (rc ==
    // child (new
    printf("hello,
    char *myargs[3
    myargs[0] = st
    myargs[1] = st
    myargs[2] = NU
    execvp(myargs[
    printf("this s
  } else {
    // parent goes
    int wc = wait(
    printf("hello,
          (int)ge
  }
  return 0;
}
```

DESCRIPTION

The **exec**() family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for **execve**(2). (See the manual page for **execve**(2) for further details about the replacement of the current process image.)

ls the **execvp**
t the string
array
er to the start
st and second
vely.

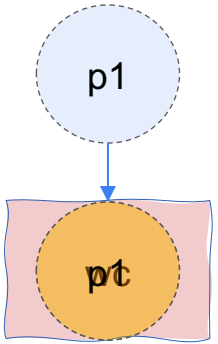


p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

execvp is going to replace the current process image with a new process image.

The image of the **wc** program.



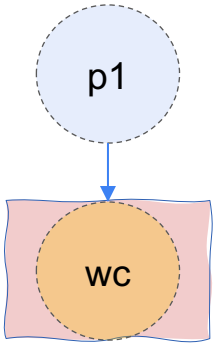
p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

At this point, the child process has a new face and begins execution from main in the **wc** program.

It ceased to be the process you know of as the **p1** program…

It has been replaced by the formidable **wc** program.

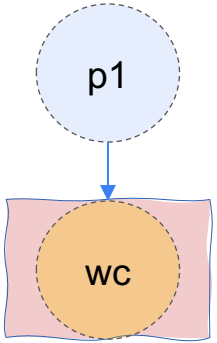


p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

The **execvp** system call never returns.

Unless, of course there is an error.

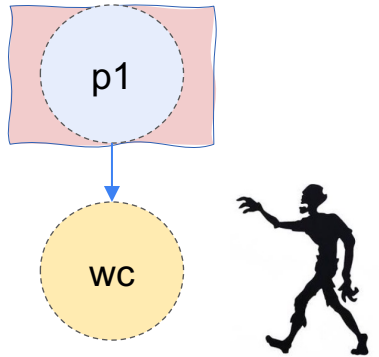


p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

At some point in the future, we context switch back to the parent.

We assume the **wc** program runs to completion and eventually exits.

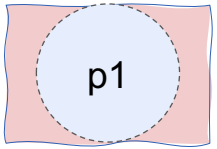


p3.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p3.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
           (int)getpid());
  }
  return 0;
}
```

The **wait** returns.

Eventually, the parent returns.



p4.c > ...

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);

    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p4.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    assert(wc >= 0);
  }
  return 0;
}
```

Ok, last example. Phew!

This is very much like the last example, but with a twist!

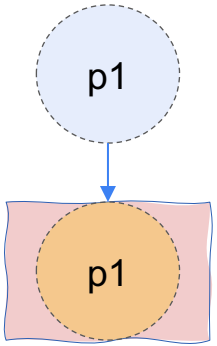
p4.c > ...

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);

    // now exec "wc" ...
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p4.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    assert(wc >= 0);
  }
  return 0;
}
```

Let's fast forward to this point in time.

The parent is waiting.

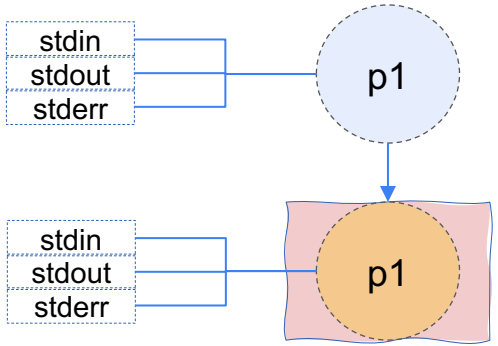
The child is executing.



p4.c > ...

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);

    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p4.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    assert(wc >= 0);
  }
  return 0;
}
```

In this example, it is important to mention that all processes are associated with the standard I/O streams: **stdin**, **stdout**, **stderr**.



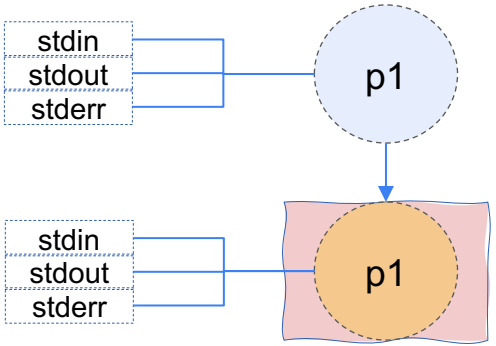
p4.c > ...

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);

    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p4.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    assert(wc >= 0);
  }
  return 0;
}
```

The child then closes the standard output stream.

Yes, you can do this.

And for good reason.



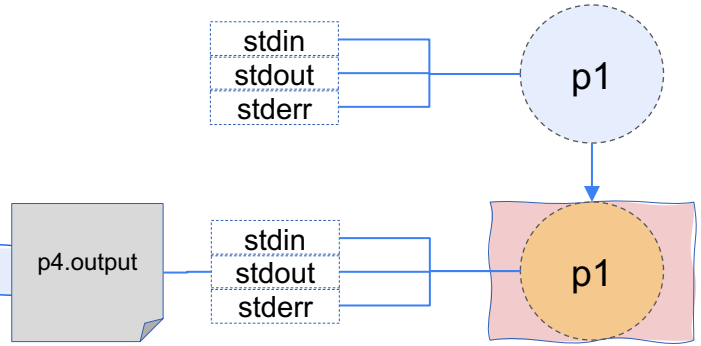
p4.c > ...

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);

    // now exec "wc" ...
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p4.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    assert(wc >= 0);
  }
  return 0;
}
```

The child then opens a new file “./p4.output”.

This new file will be assigned the standard output file descriptor effectively making it the new standard output.

How?



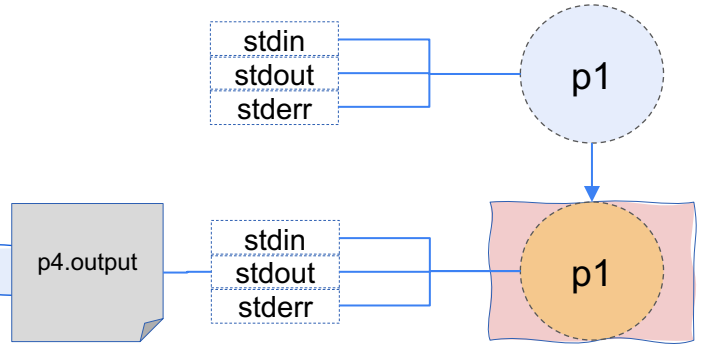
p4.c > ...

```c
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);

    // now exec "wc" ...
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p4.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    assert(wc >= 0);
  }
  return 0;
}
```

When **execvp** is called, the file descriptors are passed on to the new process image.

Thus, all standard output is *redirected* to the file.

This is how shells implement I/O redirection.



Why? Motivating the API

Of course, one big question you might have:

Why would we build such an odd interface to what should be the simple act of creating a new process?

Why? Motivating the API

Of course, one big question you might have:

Why would we build such an odd interface to what should be the simple act of creating a new process?

Well, as it turns out, the separation of `fork()` and `exec()` is essential in building a UNIX shell.

Why? Motivating the API

Of course, one big question you might have:

Why would we build such an odd interface to what should be the simple act of creating a new process?

Well, as it turns out, the separation of `fork()` and `exec()` is essential in building a UNIX shell.

It lets the shell run code after the call to `fork()` but before the call to `exec()`
this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.


The End