Synchronization!

When you fork off child processes, you generally want to make sure that your parent process waits until the child processes are finished. It is possible to just ignore them, but often enough you'll want to know that your child processes have finished, and you can do that with the `wait` command. This command waits until children are in a zombie state, and reacts!

`wait` sleeps until a child process enters the zombie state, and either returns -1 if there are no other child processes, OR `wait` selects a zombie child and frees that child's process table slot for use for another program, and then stores that child's termination status in `*t_status` and returns the child's PID.

`parent.c`

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


int main(int argc, char** argv) {
  pid_t childPid = fork();  // This is where the child process splits from the parent

  if (childPid == 0) {
    printf("I am a child! My parent's PID is %d, and my PID is %d\n", getppid(), getpid());

    char* args[] = {"./child", "Hello", "there", "exec", "is", "neat", 0};
      execv(args[0], args);

  } else {
    printf("I'm a parent! My pid is %d, and my child's pid is %d \n", getpid(), childPid);

  }

    int status;

  printf("waiting for my children\n");
  wait(&status);
    printf("Parent waited until the child ended with %d\n", WEXITSTATUS(status));
```

```
26
27    printf("Parent is now ending.\n");
28    return EXIT_SUCCESS;
29  }
```

Above, we're using the `wait` command to wait for our child! We've passed in the address of status so that we can call it with the `WEXITSTATUS` function in order to get the exit status of our child process!

Now, if we run this:

```
1  gcc parent.c
2  ./a.out
```

> I'm a parent! My pid is 16909, and my child's pid is 16910
>
> waiting for my children
>
> I am a child! My parent's PID is 16909, and my PID is 16910
>
> Hello from Child.c!
>
> I got 6 arguments:
>
> ./child Hello there exec is neat
>
> Child is now ending
>
> Parent waited until the child ended with 0
>
> Parent is now ending.

Our parent sat and waited until the child process finished, and then finished ourselves.

Terminating a Process

Many, if not most of our programs so far have all ended once the main function ended with a `return EXIT_SUCCESS` or possibly a `return 0`. This works fine and well, but it is also possible for the program to exit with the command `exit(status)` where `status` is an integer value, or by receiving an interrupt or a signal!

Signals and Interrupts

We've worked with interrupts before! Events that come from other programs or user input can act as signals for one of our programs to execute a series of commands. These commands are not unlike our signal capturing with bash! Processes typically have a default actions with signals, but we can overwrite those!

Signal Sending

We can send signals from within our programs to achieve some goal. Suppose we had a child process that was taking far too long. We could send a signal to our child process and make sure it ended! To do this, we need to make sure that we include the header `#include<signal.h>` before we try to use the system calls though!

`child.c`

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

int main(int argc, char** argv) {
  printf("Hello from Child.c!\n");
  printf(" I got %d arguments: \n", argc);

  int i;
  for (i =0; i < argc; i++){
    printf("%s ", argv[i]);
    sleep(2);
  }

  printf("\nChild is now ending.\n");

  return EXIT_SUCCESS;
}
```

`parent.c`

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

int main(int argc, char** argv) {
  pid_t childPid = fork();  // This is where the child process splits from the parent

```

```
10    if (childPid == 0) {
11      printf("I am a child! My parent's PID is %d, and my PID is %d\n",
      getppid(), getpid());
12
13      char* args[] = {"./child", "Hello", "there", "exec", "is", "neat", 0};
14        execv(args[0], args);
15
16    } else {
17      printf("I'm a parent! My pid is %d, and my child's pid is %d \n",
      getpid(), childPid);
18
19    }
20
21    sleep(5);
22    kill(childPid, SIGINT);
23
24    return EXIT_SUCCESS;
25  }
```

In the above code, all we changed was adding a sleep to our child's loop, and then including our signal library in `parent.c`, and then adding a `sleep(5)` and a `kill` command at lines 21 and 22!

If we run this code:

```
1  gcc child.c -o child
2  gcc parent.c
3  ./a.out
```

> I'm a parent! My pid is 18519, and my child's pid is 18520
>
> I am a child! My parent's PID is 18519, and my PID is 18520
>
> Hello from Child.c!
>
> I got 6 arguments:

Our parent got tired of waiting and killed the child process before it was finished!

Trapping a Signal

It's all fine and dandy to throw a sigint at a program we want to end, however, sometimes you may wish to do something more, like print one last bit of logs, deallocate memory, closing a file pointer, etc. There are any number of things you might want to do before exiting a program. Let's change our child program to capture sigint:

`child.c`

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

```c
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>

void signal_trap(int signal)
{
  if (signal == SIGINT){
    printf("received SIGINT\n");
    printf("if I had arrays, I could deallocate that memory\n");
    printf("for now, I'll just say goodbye cruel world");
    exit(0);
  }
}


int main(int argc, char** argv) {
  if (signal(SIGINT, signal_trap) == SIG_ERR)
    printf("\ncan't catch SIGINT\n");

  printf("Hello from Child.c!\n");
  printf(" I got %d arguments: \n", argc);

  int i;
  for (i =0; i < argc; i++){
    printf("%s ", argv[i]);
    sleep(2);
  }

  printf("\nChild is now ending.\n");

  return EXIT_SUCCESS;
}
```

Now, if we recompile our child program, and then run the parent, let's see what we get!

```
gcc child.c -o child
./a.out.  # << calling the previously compiled parent
```

> I'm a parent! My pid is 20123, and my child's pid is 20124
>
> I am a child! My parent's PID is 20123, and my PID is 20124
>
> Hello from Child.c!
>
> I got 6 arguments:

> ./child Hello there received SIGINT
>
> if I had arrays, I could deallocate that memory
>
> for now, I'll just say goodbye cruel world