Exam 1: Fork, Exec, etc.

So in this exam we want to build a version of gcc that compiles in parallel. The basic idea is that if way say something like this:

```
para_gcc a.c b.c c.c
```

It will compile a.c b.c and c.c to .o files IN PARALLEL and then when they are all complete link the .o files into the final executable (which in this rudimentary system will always be the default of a.out)

NOTE We have ordered the steps in what we think is the most convenient implementation order. If you have difficulty with one step though, some of them can be completed out of order so you can try the later parts too.

Table of Contents

Makefile (10 points)

So we're going to use one extra executable for this problem, named slow_gcc. Slow_gcc includes handy.h and uses a utility function in handy.c. If you want, you can compile it by hand like this:

```
gcc -o slow_gcc slow_gcc.c handy.c
```

But that is inefficient. Make a makefile that builds slow_gcc using the 2-step process (i.e. compile each c file to a .o individually, then link them into an executable). For max credit, you system should only rebuild what is necessary based on when files are updated.

You can feel free to further enhance your Makefile to build para_gcc (the file you'll be editing for this assignment). But only slow will be looked at in terms of grading the Makefile.

Running in parallel (20 points)

You can compile the given code like this (again, fell free to toss this in your Makefile if you want):

```
gcc para_gcc.c handy.c -o para_gcc
```

You can run the gcc with some example files like this:

```
./para_gcc a.c b.c c.c
```

And the output you see should look like this:

```
prompt> ./para_gcc a.c b.c c.c
I would like to run: ./slow_gcc -c a.c
done running: ./slow_gcc -c a.c
I would like to run: ./slow_gcc -c b.c
done running: ./slow_gcc -c b.c
I would like to run: ./slow_gcc -c c.c
done running: ./slow_gcc -c c.c
```

You should notice a few problems:

1. The code is just saying it would like to run gcc, not actually running gcc

2. The code is only doing the compilation stage of the process not the final link
3. The code is not running in parallel. It calls gcc one stage at a time.

To start with, use fork to solve problem #3. Your output should look like this:

```
prompt> ./para_gcc a.c b.c c.c
I would like to run: ./slow_gcc -c a.c
I would like to run: ./slow_gcc -c b.c
I would like to run: ./slow_gcc -c c.c
prompt> done running: ./slow_gcc -c a.c
done running: ./slow_gcc -c b.c
done running: ./slow_gcc -c c.c
```

In this example, the child processes are the ones printing "done running ./slow_gcc BLAH" messages.

Note that the order of a b and c could well be different (they are happening in parallel after all). The key is that the 3 "I would like to run" print before the 3 "done running".

NOTE: You also don't need to wait in the parent till your children complete before you exit (eventually, you will need to, but that's not required for this part). If you don't wait though, you might see the prompt print before all the "done running" messages (as it does in my example)...that's fine. If you did make the parent wait though, that's fine too.

Actually execing gcc (20 points)

Now use exec to make your forked children actually invoke the given gcc (e.g. slow_gcc). You'll need to pass two parameters to your gcc, "-c" and the name of the c file to compile. You can rely on gcc's default that when passed -c and filename like myfile.c, it will automatically output to a file named myfile.o.

You don't need to preserve the behavior of the previous step where there's a printout when gcc finishes. You also don't need to preserve the sleep - slow_gcc has a sleep built in.

Expected output (again prompt might print out early here):

```
SLOW GCC -c a.c
SLOW GCC -c b.c
SLOW GCC -c c.c
```

You should also see that the .o files are really created by the process.

Doing the final compile (15 points)

This requires previous step "Running in parallel" at minimum.

At this point we want to link all the produced .o files into an executable by execing gcc one last time. We'll need to wait till all our children finish before we do this.

The exec has a complication - it can take a different number of parameters (that is, it should take the same number of parameters as the original call to para_gcc.

To make this work, I recommend you use the execvp (or execv) variant that takes a array of parameters as it's second argument. To do this safely, you'll need to make a copy of the parameters to para_gcc terminated by a NULL. I put a function in handy.c that should help, here's how I used it:

```
char** argv_copy = malloc_a_copy_that_ends_in_null(argv, argc);
```

You'll also probably want to use replace_dotc_with_doto which I proved at the top of para_gcc itself.

Expected output:

```
SLOW GCC -c a.c
SLOW GCC -c b.c
SLOW GCC -c c.c
SLOW GCC a.o b.o c.o
```

The parent should now be waiting for the children to complete, so you shouldn't have any trouble with the prompt printing early. You should also see a.out produced which should be a runnable executable.

Dealing with failures (15 points)

This requires previous steps "Running in parallel" and "Actually running gcc".

So if one of the .o files does not build successfully, we should not attempt to link the overall executable and should instead print "child failed...aborting". To test this, just include a file that doesn't exist. Here's my output:

```
prompt> ./para_gcc a.c b.c c.c bad.c
SLOW GCC -c a.c
SLOW GCC -c c.c
SLOW GCC -c b.c
SLOW GCC -c bad.c
gcc: error: bad.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
child failed...aborting
```

You'll want to check to see if any of the children produced a nonzero status to see if a particular build failed.

Cleanup on Control-C (10 points)

This step requires "Doing the final compile" and "Actually running gcc" for full credit, but you can get some credit even with none of those.

So if you press ^C on the command line, it will send SIGINT to both the parent and child processes - so the child processes will stop without you needing to do anything. However, after the children stop we would like to clean up any .o files that might have been created:

```
SLOW GCC -c a.c
SLOW GCC -c c.c
SLOW GCC -c b.c
^Cgot sigint signal...quitting!
got sigint signal...quitting!
got sigint signal...quitting!
cleaning up files...
rm: cannot remove 'a.o': No such file or directory
rm: cannot remove 'c.o': No such file or directory
rm: cannot remove 'b.o': No such file or directory
```

Note in this case that I stopped the process early enough that no .os actually existed, but these errors are fine (you'll see some more successes after you complete the last part).

The "got sigint signal...quitting!" messages are already built into slow_gcc so you don't need to add that. However, your code should wait for the child gccs to complete before starting the removal process - i.e. "cleaning up files" should always print AFTER the "got sigint signal" messages.

HINT: while you could track the number of outstanding (i.e. unwaited) children to determine how many waits are needed, there's an easier way. Wait returns a (negative) error response if you wait with no children, so it's possible to just loop until it starts failing.

Use global variables to pass the handler the data it needs to pass to rm. It also might be wise to initially call the built in command "echo" which simply prints the given input rather than "rm" for your first tests. That way you don't accidently delete some .c files if you mess up your exec call.

Note that if you do your ^C too late, you'll interrupt the final call to slow_gcc rather than para_gcc and your handler won't run. That's unavoidable and fine.

Limiting the number of parallel builds (10 points)

This step requires "Doing the final compile". I recommend you backup your code before you start it, because it's easy to mess up the loop and get something pretty broken.

So it's not really a smart move to attempt to build every single .o in parallel at the same time. A build can have 100s of associated .c files, and we really don't want to spawn 100 of processes.

Instead, modify the code that it will only run max_gccs gcc instances in parallel. Max_gccs initially defaults to 2, so initially it should start 2 processes, then after a process finishes successfully it should start more, etc.

In this example I've modified the code to print when a compile finishes to make it obvious. You don't have to do this to get credit, but it makes it clearer if it's working.

BTW, be sure to use slow_gcc as your testcase here as otherwise your gccs will finish at almost exactly the same time.

```
SLOW GCC -c a.c
SLOW GCC -c b.c
gcc finished
SLOW GCC -c c.c
gcc finished
gcc finished
SLOW GCC a.o b.o c.o
```

Do test with a variety of different input file numbers and values for max_gccs. For testing with a lot of inputs just repeat files (e.g. para_gcc a.c b.c c.c c.c c.c c.c) - the code won't link in the end but it will do the right amount of compiling.

Submitting your solution

Submit your Makefile and para_gcc.c to the dropbox on Moodle.