Making assertions about your code

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

—Brian Kernighan

A very useful feature supported by the C preprocessor permits you to make assertions about the correctness of your code.

Each assertion is checked as your program runs (at run-time):
- if the assertion is correct, execution continues without any noticeable difference (a tiny, tiny, bit slower),
- if the assertion fails, execution is halted, an error describing the failed assertion is printed, and the program terminates.

Consider:

```
#include <stdio.h>
#include <assert.h>

....
for(int i=0 ; i <= 5 ; i = i+1) {
  assert(i != 5);
  printf("i = %d\n", i);
  assert(i*i < 25);
  printf("i squared = %d\n", i*i);
}
```

When compiled and run, produces:

```
  i = 0, i squared = 0
  i = 1, i squared = 1
  i = 2, i squared = 4
  i = 3, i squared = 9
  i = 4, i squared = 16
```

Creating A New Unix/Linux Process

We shall first create a new process by calling the `fork()` system call.

Firstly, we note that each process is uniquely identified by an integer value termed its process identifier, process-id, or pid.

A process can get its own process-id with the system call ` getpid()`, and get its parent’s process-id with ` getppid()`. 
C code to fork a new process

`fork()` is very unusual because it returns different values in the (existing) parent process, and the (new) child process:

- the value returned by `fork()` in the parent process will be the process-id of the child process;
- the value returned by `fork()` in the child process will be 0, indicating that it is the child, because 0 is not a valid process-id.

Each successful invocation of `fork()` returns a new monotonically increasing process-id (the kernel 'wraps' the value back to the first unused positive value when it reaches 100,000).

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

void function(void)
{
    int pid; // some systems define a pid_t
    switch (pid = fork()) {
        case -1:
            perror("fork()"); // process creation failed
            exit(1);
            break;
        case 0:
            // new child process
            printf("c: value of pid=%\n", pid);
            printf("c: child's pid=%\n", getpid());
            printf("c: child's parent pid=%\n", getpid());
            break;
        default:
            // original parent process
            sleep(1);
            printf("p: value of pid=%\n", pid);
            printf("p: parent's pid=%\n", getpid());
            printf("p: parent's parent pid=%\n", getpid());
            break;
    }
    fflush(stdout);
}
```

produces:

```
c: child's value of pid=0
c: child's pid=5642
c: child's parent pid=5641
p: parent's value of pid=5642
p: parent's pid=5641
p: parent's parent pid=3244
```

Of note, calling `sleep(1)` may help to separate the outputs, and `fflush()` in each process to force its output to appear.
Memory in Parent and Child Processes

The (existing) parent process and the (new) child process continue their own execution.

Importantly, both the parent and child have their own copy of their program's memory (variables, stack, heap).

The parent naturally uses the memory that it had before it called fork(); the child receives its own copy of the same memory. The copy is made at the time of the fork() function. As execution proceeds, each process may update its own memory without affecting the other process.

[ OK, I lied - on contemporary operating systems, the child process does not receive a full copy of its parent's memory at the time of the fork():

- the child can share any read-only memory with its parent, as neither process can modify it.
- the child's memory is only copied from the parent's memory if either the parent modifies its (original) copy, or if the child attempts to write to its copy (that it hasn't yet received!)
- this sequence is termed copy-on-write.
]
**Waiting for a Process to Terminate**

If a single program has two distinct execution paths/sequences, then the parent and child may run different parts of the same program. Typically the parent will want to know when the child terminates.

The typical sequence of events is:

- the parent waits for the child's termination, calling the blocking function `wait(&status)`.
- [optionally] the child process replaces details of its program (code) and data (variables) by calling the `exec()` function.
- the child calls `exit(value)`, with an integer value to represent its success or failure. By convention, zero (= EXIT_SUCCESS) indicates successful execution, non-zero otherwise.
- the child's value given to `exit()` is written by the operating system to the parent's `status`. 

![Diagram of process execution and termination](image_url)
Running a New Program

Of course, we do not expect a single program to meet all our computing requirements, or for both parent and child to conveniently execute different paths through the same code, and so we need the ability to commence the execution of new programs after a fork()..

Under Unix/Linux, a new program may replace the currently running program. The new program runs as the same process (it has the same pid, confusing!), by overwriting the current process’s memory (instructions and data) with the instructions and data of the new program.

The single system call `execve()` requests the execution of a new program as the current process:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
char *newargs[] = {
    "ls",
    "-
",
    NULL
};
....
execve("/bin/ls", newargs, environ);
exit(EXIT_FAILURE);
```

On success, `execve()` does not return (to where would it return?)
On error, -1 is returned, and errno is set appropriately (EACCES, ENOENT, ENOEXEC, ENOMEM, ....).

The single system call is supported by a number of library functions (see man `exec`) which simplify the calling sequence.

Typically, the call to `execve()` (via one of its library interfaces) will be made in a child process, while the parent process continues its execution, and eventually waits for the child to terminate.
Why the exit status of a program is important

To date, we've always used `exit(EXIT_FAILURE)` when a problem has been detected, or `exit(EXIT_SUCCESS)` when all has gone well. Why?

The operating system is able to use the `exit status` of a program to determine if it was successful.

Consider the following program which exits with the integer status provided as a command-line argument:

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

int main(int argc, char *argv[])
{
    int status = 0;  // DEFAULT STATUS IS SUCCESS
    if(argc > 1)
        status = atoi(argv[1]);
    printf("exiting(%d)\n", status);
    exit(status);
}
```
Why the exit status of a program is important, *continued*

Most operating system *shells* are, themselves, programming languages, and they may use a program's `exit status` to direct control-flow within the shells - thus, the programming language that is the shell, is treating your programs as if they are external functions.

Shells are typically programmed using files of commands named *shellscripts* or *command files* and these will often have conditional constructs, such as `if` and `while`, just like C. It's thus important for our programs to work with the shells that invoke them.

We now compile our program, and invoke it with combinations of zero, and non-zero arguments:

```
prompt> mycc -o status status.c
prompt> ./status 0 && ./status 1
exiting(0)
prompt> ./status 1 && ./status 0
exiting(1)
prompt> ./status 0 || ./status 1
exiting(1)
prompt> ./status 1 || ./status 0
exiting(0)
```

- Example1 - consider the sequence  
  ```
prompt> cd mydirectory && rm -f *
  ```
- Example2 - consider the actions in a *Makefile* (discussed in a later lecture).
  If a `target` has more than one action, then the `make` program executes each until one of them fails (or until all succeed).

---