CSC222: Systems Programming
C System Libraries

1 The Exec Family

See program: `modDate.c`
Like `system()` there are a class of functions that can be used to execute system commands.

- In the `system()` function, the program executes the command, waits for the command to execute, and then continues.
- The `exec()` family of functions replaces the current process image with a new process image.
- In the `execl()` command (execute and leave), the command executes in place of the current process.
- The definition is

```
int execl(const char *path, const char *arg, ...);
     // or const char *arg0, ..., char *argn, 0;
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,
           ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- The initial argument for these functions is the pathname of a file which is to be executed.
- The `execl(), exclp(), excle()` functions are variadic functions, with a list of arguments, so the argument list is terminated with a NULL value (`(char *)NULL`).
- By convention, the first argument sent (`arg0`) should point to the filename associated with the file being executed.
- The `execle()` function adds an additional array of strings that is also NULL-terminated. This corresponds to the environment for this executable. The other functions use the current process's external variable environment for the new process image.
- The `execv()` and `execvp()` functions provide an array of pointers to null-terminated strings for the argument list. This array is also NULL-teriminated.
- The `execlp(), execvp()` functions use the PATH variable for searching.
- ***Return Value***:
  If any of the `exec()` functions returns, an error will have occurred. The return value is -1, and the global variable `errno` will be set to indicate the error.
- All of these functions are actually frontends for the `execve()` function.
  `int execve(const char *filename, char *const argv[], char *const envp[]);`

2 Forking A Process

See program: `splitFork.c`

As we have seen, modern computers and modern operating systems function by multitasking. Even with one processor, the computer appears to be able to run as if it has several processes running simultaneously. We saw how to manage multiple processes from the shell. Now we shall learn a little how this is accomplished with C.

- Fork - create a child process
  `pid_t fork(void);`
- Fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an ***exact duplicate*** of the calling process, referred to as the parent, except for the following points:[1]

[1] We only highlight key differences here. See man pages for full details.

  - The child has its own unique process ID, and this PID does not match the ID of any existing process group.
  - The child's parent process ID is the same as the parent's process ID.
  - Process resource utilizations and CPU time counters are reset to zero in the child.
  - The child's set of pending signals is initially empty (sigpending).
- The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description as the corresponding file descriptor in the parent.
- On success, the PID of the child process is returned ***in the parent***, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created and errno is set.
- System works by running one process for a certain time then swapping out and running other processes. This switching back and forth is known as ***process scheduling***, and can get quite complex - but is handled for us by the OS. Run the previous program with the value 100000 to see when things are swapped out and the effect it can have with shared resources (like the output stream!) Would be even messier if they shared an input stream in common!
- Note the order in which each process is executed is ***non-deterministic***. This can make debugging a little trickier since errors might not always be recreatable (deadlock situations are a classic example.)
- You can cause a process to stop at a given time (if you wish) by putting it to ***sleep***. This is done with the command: `sleep(unsigned int seconds)`. Set the `ENABLE_SLEEPING` macro to 1 to see the difference.
- ***See program:*** `forkSharing.c`
  To see what is *not* shared. This program also illustrates the effect of the parent finishing before the child process terminates. You can use the `wait()` function to have the parent wait on a child process to terminate. In addition we can get the exit status (among other things) of the child process.
- One of the most common uses for a fork is for a process to spawn off a child process which immediately executes another command via `execl()` (or similar function).

3 Pipes

See program `forkPipe.c`.

- Pipes allow for communication between two forked processes.[2]
- Pipes are file descriptors not associated with an actual file (but an interprocess channel).
- General format:
  ```
  int pipe(int filedes[2]);
  ```
- The `pipe()` function creates a pipe and places two file descriptors one each into the arguments fildes[0] and fildes[1], that refer to the open file descriptions for the read and write ends of the pipe.
- Data can be written to filedes[1] and read from filedes[0].
- For the same process, this generally would be meaningless. But with forking, since the descriptors remain the same on each child, one process uses filedes[1] and the other filedes[0]. Since these cannot be shared, for two-way communication, two separate pairs must be created!
- You can avoid using the read and write and use fscanf, fprintf instead by doing an `fdopen` command. ***See program***`forkPipeTwo`.
- There are limits to the number of pipes you can create.
  See program `matrixMultInParallel` for an example.
- If you have multiple processors, you can get speed up by using parallelism (otherwise, you won't find much speed increase).
  See programs `matrixMultInParallelTwo` ***and*** `matrixMultInParallelImproved` for examples.

[2] There are other ways too.

4 Duplicating File Descriptors

See programs `dup*Example.`
It often becomes necessary to make a duplicate of a file descriptor. This is similar to the use of IO redirects in Bash. The easiest way to do that is to use the commands `dup` and `dup2`. See the man pages and sample code for details.

5 Signals

When an abnormal event happens, we can use ***signals*** to notify the processes of an exceptional condition. An example is when the `ctrl-c` key is pressed to create an interrupt.

- In C we use the `signal()` function (from `<signal.h>`).
- Prototype is:
  ```
  typedef void (*sighandler_t)(int);
  sighandler_t signal(int signum, sighandler_t handler);
  ```
  That is, the signal requires a signal handler to associate with a given signal number. The signal handler is a (pointer to a) function that takes an `int` and returns nothing.
- According to the man pages, the behavior of this function varies across Unix (and even Linux) versions and the use of `sigaction` is recommended instead. We shall thus explore `sigaction` but for legacy reasons you should at least familiarize yourself with the `signal()` function as well, as older code might still use this function!
- Defined in the same header, the `sigaction()` function has the following prototype:
  ```
  int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
  ```
- The method is used to change the action taken by a process on receipt of a specific signal (specified by `signum`).
- If `act` is non-null, the new action for signal `signum` is installed from `act`.
- If `oldact` is non-null, the previous action is saved in `oldact`.
- The structure of `sigaction` is roughly:
  ```
  struct sigaction {
      void     (*sa_handler)(int);
      void     (*sa_sigaction)(int, siginfo_t *, void *);
      sigset_t   sa_mask;
      int        sa_flags;
      void     (*sa_restorer)(void);
  };
  ```
- ***Warning***: On some architectures, this uses a union so that `sa_handler` and `sa_sigaction` should *not* be used at the same time.
- ***See program:*** `dontInterruptMe.c`

6 Deadlock

Although we shall not address the issue of resolving deadlock, other than a few suggestions. We shall discuss briefly the Dining Philosopher's problem. To summarize, the problem happens when two (or more) processes are both waiting to share resources while blocking the other. For example, process A is holding resource 1 and waiting on resource 2, while process B is holding resource 2 and waiting on resource 1.... ***deadlock.***

Resolving deadlocks is one of the key issues in multitasking and operating systems (imagine if the OS locked up!)
See Program: `diningPhilosophers.C`

7 Threads

See Program: `threadingTheSum.c`
Besides a program creating multiple processes, which act independently of each other but can communicate via pipe streams, a *single process* can also have multiple ***threads*** which all execute the same program.

- In UNIX, the main way to use threads is to use POSIX threads, or ***Pthreads***.
- Unlike processes, these threads all share the same global memory (data and heap segments), but each thread has its own stack (automatic, local, variables).
- (POSIX) Threads also share (among others):
  - Process ID
  - Parent process ID
  - User and group IDs
  - Signal dispositions (how signals are handled, see previous section)
  - Open file descriptors (like forked processes do)
  - Current directory and root directory
- Besides automatic variables, (POSIX) threads do not share (among others):
  - Thread ID
  - Signal mask (for signals)
  - Errno variable (otherwise, this would make errors very difficult to catch)
- Compiling with pthreads requires the `-pthread` (or `-pthreads`) flag to be set.
- There are several advantages to threads versus non-threads and multiple processors
  - Threads are (typically) much more efficient than forked processes, but note that threads would share the same processor so you could not take full advantage of say a multi-core computer.
  - There is less communication overhead between threads than between forked processes. You don't need to use pipes for example since they can use shared memory space.
  - Overlapping regular processing with input and ouput.
    Typically, when waiting for an input (or output) to complete the thread must freeze (be idle). With multi-threading (or multi-processing), the other threads (or processes) can be busy doing meaningful tasks.
  - You can do priority scheduling. For example, if a program needs to do calculations as well as real-time displays, the real-time displays can have priority to ensure that there is no flickering or lag while the calculations might be done in the background. The same can go for user interaction. One thread can be busy constantly (consistently) checking for user input and reacting to it while other threads can be performing important calculations.
- You can find out more by reading the man pages. In Ubuntu you may have to add the packages:
  `manpages-posix manpages-posix-dev`
- You can find out more by reading various pthread tutorials such as:
  https://computing.llnl.gov/tutorials/pthreads/ (from Lincoln Labs).