Systems Software HS15

Lab Exercises

Claudio Mura
claudio@ifi.uzh.ch

Practical notes on UNIX Processes

Processes and Identifiers

- a process is a program in execution
  - the "dynamic version" of a program
- the OS keeps some information for each process
  - process id (*pid*), state, parent pid, children, open files, ...
- process id is a fundamental piece of information
  - represented with type `pid_t`
  - `getpid()` function to retrieve it

```
pid_t my_pid = getpid();
std::cout << "Pid of current process:" << my_pid << "\n";

pid_t parent_pid = getppid();
std::cout << "Pid of parent process:" << parent_pid << "\n";
```

Creating a Process: `fork()`

- creates a new process (*child*) from an existing process (*parent*)
  - the child is a copy of the parent
    - content of text, stack and data segments is copied
  - some major differences (eg: pid, parent pid)
- returns two times
  - once in the child, once in the parent
- returns a `pid_t` variable containing:
  - the pid of the child, in the parent process
  - the value 0, in the child process
  - the value -1, if the process could not be created

Waiting for Child Process

- after `fork()` returns, the parent and the child process execute concurrently
- when a child completes its execution, it remains in a *zombie* state
  - finished execution, inactive, but its resources can not be deallocated until the parent *waits* for him
- the parent process should call the `wait()` function
  - suspend its execution until a child terminates
    - can wait for a specific child using `waitpid()`
  - returns immediately if a child has already terminated

Checking Child Termination

- both wait and waitpid take an `int* status` argument
  - pointer to a previously declared integer variable
  - information about status of terminated process
- a set of macros are available to check how the process has terminated
  - `WIFEXITED`, `WIFSTOPPED`, `WIFSIGNALED`, ...
  - read the man pages for other useful macros

Example of `wait()`

```
pid_t pid = fork();

if( pid == -1 ) {
  std::cerr << "Error on fork()" << std::endl;
  exit( 1 );
}
if( pid == 0 ) {
  pid_t my_pid = getpid();
  std::cout << "Child process, pid = " << my_pid << std::endl;
  exit( EXIT_SUCCESS );
}
else {
  int status;
  wait( &status );

  if( WIFEXITED( status ) ) {
    std::cout << "Child terminated normally" << std::endl;
    return 0;
  }
  else {
    // handle error
  }
}
```

Changing Process Image

- a process is a program in execution
- using the `exec()` family of functions it is possible to change the program run by a process
  - common practice: use `fork`, then call `exec` in the child
- the text, stack and data segment of the process are replaced
- you must provide to `exec()`:
  - fullpath of executable of the new program
  - input arguments for the new program
    - by convention, the first is the program name
- Note: if successful, `exec` does not return!

Example of `exec()`

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


int main ( int argc, char* argv[] )
{
  execl( "/bin/echo",
         "echo",
         "Message to be displayed",
         NULL );

  std::cerr << "Error: exec has returned!" << std::endl;

  return 1;
}
```

Exercise 2

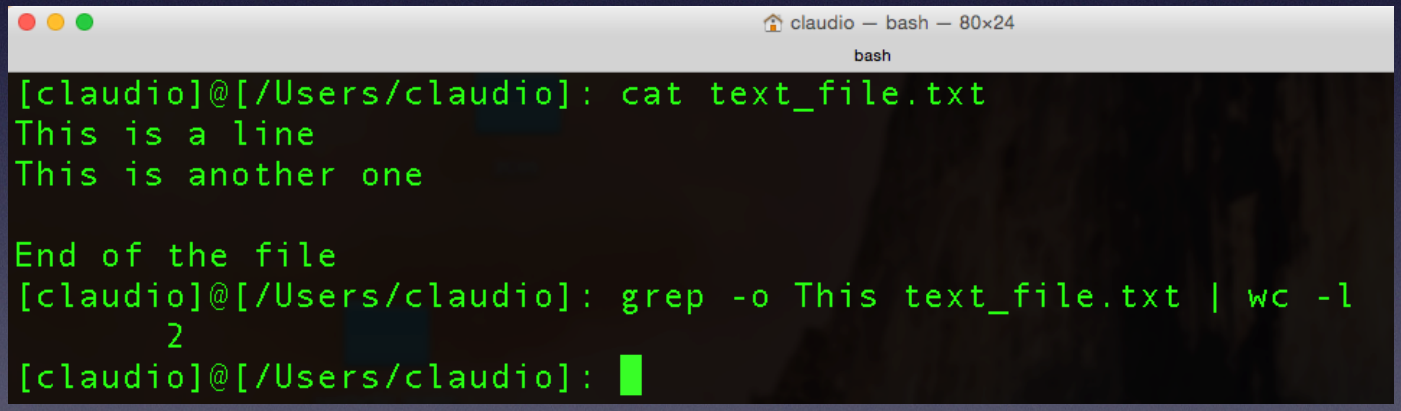
Process creation and management

Parallel string search in text files

- Write a multi-process program that, given a string (*pattern*) and a list of text files, counts the occurrences of the pattern inside the text files
  - pattern and filenames passed as command line arguments
- You should create multiple processes:
  - one child process for every text file
  - each child performs the counting on one file, then outputs the result to a text file
  - use `fork()` from the main process to create the children
  - main process reads output files, computes overall number of occurrences and prints it to standard output

Counting occurrences in a file

- Don't code it yourself! You must use `exec*` to execute an instance of a shell that does it for you
  - using the following command: `grep -o PATTERN FILENAME | wc -l`



  - add `> OUTPUT_FILENAME` to write the result to a file
  - the output file with the result should be called `result-PID.txt`, with `PID` being the process id of the child process

Counting occurrences in a file

- With **exec** (example):

```
execl( "/bin/sh", "/bin/sh",
       "-c",
       "grep -o This text_file.txt | wc -l > out.txt",
       ( char* )0 );
```

Assembling the results

- Task of the main process
  - open the output files generated by the child processes
    - all children must have generated their result file - so they must have terminated, and successfully!
    - wait for the processes and check termination status before reading the files
  - from each output file, read the *partial* occurrences count
    - i.e., the number of occurrences found by the child process that generated that output file
  - sum up the partial occurrences and print the result to standard output

Additional comments

- Handle the following abnormal situations appropriately
  - insufficient number of command-line arguments
  - `fork()` or `exec()` fail
  - a child returns abnormally
  - errors while opening file
- A draft of the solution is provided together with the task description
  - you may use it at your convenience