Making Processes

- Topics
  - Process creation from the user level:
    - fork, execvp, wait, waitpid
- Learning Objectives:
  - Explain how processes are created
  - Create new processes, synchronize with them, and communicate exit codes back to the forking process.
  - Start building a simple shell.

Recall how we create processes: `fork`

```
#include <unistd.h>
pid_t   ret_pid;

ret_pid = fork();
switch (ret_pid){
        case 0:
                /* I am the child. */
                break;
        case -1:
                /* Something bad happened. */
                break;
        default:
                /*
                 * I am the parent and my child's
                 * pid is ret_pid.
                 */
                break;
 }
```

But what good are two identical processes?

- So `fork` let us create a new process, but it's identical to the original. That might not be very handy.
- Enter `exec` (and friends): The `exec` family of functions replaces the current process image with a new process image.
- `exec` is the original/traditional API
- `execve` is the modern day (more efficient) implementation.
- There are a pile of functions, all of which ultimately invoke `execve`; we will focus on `execvp` (which is recommended for Assignment 5).
- If `execvp` returns, then it was unsuccessful and will return -1 and set `errno`.
- **If successful, `execvp` does not return, because it is off and running as a new process.**
- Arguments:
  - file: Name of a file to be executed
  - args: Null-terminated argument vector; the first entry of which is (by convention) the file name associated with the file being executed.

Programming with Exec

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
pid_t    ret_pid;

ret_pid = fork();
switch (ret_pid){
         case 0:
                  /* I am the child. */
                  if (execvp(path, argv) == -1)
                           printf("Something bad happened: %s\n",
                               strerror(errno));
                  break;
         case -1:
                  /* Something bad happened. */
                  break;
         default:
                  /*
                   * I am the parent and my child's
                   * pid is ret_pid.
                   */
                  break;
}
```

Screen Capture

- Examine parent.c
- Examine child.c
- Predict what will happen if you run:
  - ./parent child
  - ./parent ./child
  - ./parent ./child a b c
  - ./parent ls .
- Examine the strace output when you run:
  - ./parent ./child
  - You might find –e handy:
    - `strace -e trace=process ./parent ./child`)

Coordinating with your child

- Sometimes it is useful for a parent to wait until a specific child, all children, or any child exits.

```
pid_t waitpid(pid_t pid, int *stat_loc, int options)
```

- Waits (puts parent in blocked state) until the child with `pid` exits.
  - If `pid = -1`, wait until any child exits.
  - If `pid` does not exist or is not a child of the waiting process, returns -1.
- Return value is the pid of the terminating process
- `stat_loc` is filled in with a status indicating how/why the child terminated.

Screen capture

- Examine `waiting-parent.c`
- Run waiting-parent.c with:
  - ./waiting-parent ./child a b c
  - ./waiting-parent child

Why did children exit?

- Sometimes, it is useful to know why a child exited. How do you find out?
- The status returned by `waitpid` provides additional information:
  - `WEXITSTATUS(status)`: If true, evaluates to the low-order 8-bits of the value the child passed to `exit`.
  - `WIFEXITED(status)`: This returns true if the child exited normally by calling `exit`.
  - `WIFSIGNALED(status)`: This returns true if the child terminated due to receipt of a signal.
  - `WIFSTOPPED(status)`: This returns true if the child has not terminated, but is stopped and can be restarted.

`waitpid` options

- What is that last argument to `waitpid`?
- A set of bits you can or together:
  - `WNOHANG`: return immediately if no child has exited or changed state; returns 0.
  - `WUNTRACED`: also return if a child has stopped (but can be restarted).
  - `WCONTINUED`: also return if a child has been resumed by delivery of the SIGCONT signal.

Screen Capture

- Let's look at `waiting-parent2.c` and `badchild.c`
  - `badchild.c` loops forever, but `waiting-parent2.c` specifies `WNOHANG`.
  - Run `./waiting-parent2 ./badchild &`
  - You'll see interleaved messages and will need to do `kill <pid>` (e.g., `kill 42308)` to kill the child.
- Now examine `waiting-parent3.c`
- This time, we still block, but we ask to be notified on stop/continue of child.
  - Run ./waiting-parent3 ./badchild &
  - Try stopping and starting the child with:
    - `kill –STOP pid`
    - `kill –CONT pid`

Compare and Contrast

Blocking waitpid:

```
p = fork();
waitpid(p, &status, 0);
```

Polling waitpid:

```
p = fork();
while (p != waitpid(p, &status, WNOHANG));
```

What is the difference between these two uses?

Measuring Efficiency: Utilization

- How do you measure how efficiently something is being done?
- We want to use utilization as the metric.
  - The fraction of the processor being used to do useful work.
- However, you will see utilization used in different ways:
  - "The processor was 100% utilized."
  - "We were able to do this with only 1% processor utilization."
- Are big numbers good or bad?

Screen Capture

- Take a look at util.c
- Now run it in the background: `util &`
- While that's run, watch the output of top.
- This program is achieving near 100% CPU utilization.
- Is that a good thing?

Tradeoffs between Polling and Blocking

- Blocking avoids wasted work.
- Blocking sometimes suffers from atomicity problems:

```
if (event hasn't happened)
    issue blocking call
```

- Polling can sometimes be more responsive:
  - If it takes you longer to block than the time it takes for the event on which you are waiting to complete, polling might be better.
- We often refer to polling as busy-waiting (we keep the processor busy while we wait). In general, you should avoid busy-waiting!


BEN&JERRY'S
Applied Math & Computer Science
Sophomore Advising Fair
Join us for some free ice cream and a chance to meet with faculty, advisors and current concentrators in Applied Mathematics and Computer Science. Whether you are ready to declare your concentration or just learn more about the courses offered, we encourage to to stop by and stay as long as you'd like.
When: Monday, Nov. 14th from 4-5:30pm
Where: Maxwell Dworkin Lobby
(33 Oxford Street)