Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 13

- Concurrent Programming
- fork
- Pipes
- system, exec, popen

Concurrent Programming (1)

- Concurrent programming is a hot topic
- New chips have multiple CPU cores on one die
- Why? CPU clock frequency has hit physical limits
- Orchestrate many processors to work on single task
- Issues include
  - Speedup: how much faster will the program run on N CPUs?
  - Communication Overhead: how much time is spent on inter-process communication?
  - Fairness: resources are allocated in order they are requested
  - Deadlocks: two processes are waiting for the other to release a resource.

Concurrent Programming (2)

- UNIX is a multi-tasking operating system
- Many processes can run simultaneously (e.g. try `top`, `ps auxw`)
- Small number of CPUs (usually 1-2) execute processes in a time-shared manner
- Most processes sleep and wait for external events, such as user input

Here we briefly look at process generation in UNIX and process communication via pipes

fork

- UNIX library function `fork` creates a child process, which runs in parallel with the parent process
- Child process gets a copy of the parent memory (including file descriptors)

fork Example 1

```
int main()
{
  pid_t pid;      // process id
  pid = fork();   // spawn child process
                  // both processes resume execution here
                  // only difference: pid!

  if (pid < 0) { perror("fork"); exit(10); }

  if (pid == 0) { // child process
    cout << "child process running" << endl;
    for (int i=0; i < 100000000; ++i);
    cout << "child done" << endl;
    return 0;
  }

  // parent (original) process
  cout << "child process pid:" << pid << endl;

  // do something here
  waitpid(pid, 0, 0); // wait for child process to finish
  cout << "parent done" << endl;
  return 0;
}
```


fork Example 2

```
#include <cstdio>
#include <cstdlib>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
using namespace std;

int x;

void work(char c)
{
  x = c;
  for (int i=0; i < 20; ++i) {
    for (int j=0; j < 10000000; ++j);
    cout << c << flush;
  }
  cout <<  " " << c << " " << x
       << " DONE " << endl;
}
```

```
int main()
{
  pid_t pid; // process id

  // spawn child process
  pid = fork();

  if (pid < 0)
    return pid; // error

  if (pid == 0) { // child
    work('A');
    return 0;
  }

  // parent process
  work('B');

  // wait for child
  waitpid(pid, 0, 0);
  return 0;
}
```

ABBBBBBBABBABABABABABBAABABABBAB B 66 DONE
AAAAAAAA A 65 DONE

Pipes

- Pipes are half-duplex communication channels
- Can be used for process communication
  E.g. connecting stdout of one process with stdin of another
- Shell `cmd1|cmd2` is implemented using pipes: `cmd1` forks a second process (`cmd2`) and connects `cmd1` stdout to `cmd2` stdin.
- In above scheme, the pipe has to be created before calling `fork`

```
int fds[2];
if (pipe(fds) < 0) { // error ...
...write(fds[1], buf1, len);
...read(fds[0], buf2, len);
```


Pipe Example

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
using namespace std;

int main(int argc, char **argv)
{
  int fds[2];
  pid_t pid;

  if (pipe(fds) < 0) { // create pipe
    perror("pipe failed"); exit(1);
  }
  if ((pid = fork()) < 0) { // child
    perror("fork failed"); exit(2);
  }
```

```
  if (pid == 0) {     // child

    close(fds[1]);    // close output

    char c;
    while (read(fds[0], &c, 1) > 0) {
      cout << "read: " << c << endl;
    }
    close(fds[0]);    // close input
    cout << "child done" << endl;

  } else {            // parent

    close(fds[0]);    // close input

    char msg[] = "hello..world..";
    write(fds[1], msg, strlen(msg));
    close(fds[1]);  // -> EOF

    waitpid(pid, 0, 0);
    cout << "parent done" << endl;
  }
  exit(0);
}
```

system

Execute shell commands from within C programs

```
int system(const char *cmd_string);
```

- Executes command by calling `/bin/sh -c cmd_string`
- Returns -1 on error (fork failed) and exit value otherwise
- Drawback: no input/output

```
#include <cstdlib>

system("cp foo bar");
```


exec Family

Replace current process image with new one

```
int execl(const char *cmd_file, const char *arg, ...);
```

- `cmd_file` either executable or script file starting with `#! <interpreter>` (e.g. `#! /bin/bash`)
- Argument list must be 0-terminated
- = arg0, arg1, ... accessible by main
- Functions only return in case of error

```
#include <unistd.h>

execl("/bin/ls", "ls", "-l", 0);
```


Pipes + Exec

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <unistd.h>

int main(int argc, char **argv)     // ls | wc -l
{
  int fds[2];
  pid_t pid;
  if (pipe(fds) < 0)      { perror("pipe failed"); exit(1); }
  if ((pid = fork()) < 0) { perror("fork failed"); exit(2); }
  if (pid == 0) {               // child
    close(fds[1]);              // close output
    dup2(fds[0], 0);            // close fd 0, fd 0 = fds[0]
    execl("/usr/bin/wc", "wc", "-l", 0);
    perror("failed to run wc");
  } else {                      // parent
    close(fds[0]);              // close input
    dup2(fds[1], 1);            // close fd 1, fd 1 = fds[1]
    execl("/bin/ls", "ls", 0);
    perror("failed to run ls");
  }
  exit(0);
}
```


popen

Convenient wrapper for pipe, fork, and exec

```
FILE *popen(const char *command, const char *type);

int pclose(FILE *stream);
```

- Runs command in shell childprocess
- Returns 0 if fork or pipe failed or command is not found
- Gives access to stdin or stdout through FILE pointer
- `type`: `"r"`: read from cmd, `"w"`: write to cmd
- `pclose` waits for command to finish

popen Example

```
#include <cstdio>
#include <cstdlib>

int main()
{
  FILE *fp = popen("ls", "r");
  if (!fp) { perror("error"); exit(10); }

  while (1) {
    int c = fgetc(fp);
    if (c == EOF) break;
    fputc(c, stdout);
  }
  fclose(fp);
}
```