Lecture 6: Processes

Dr. Naureen Nizam
nnizam@cs.toronto.edu

CSCB09H3: Software Tools and Systems Programming (SY 110)
Department of Computer and Mathematical Science
Feb 11/13, 2019

UTSC Tomorrow is created here.
UNIVERSITY OF TORONTO SCARBOROUGH
1265 Military Trail, Toronto, Ontario M1C 1A4

Administrivia

- **Assignments**
  - A2 Posted
  - Due: March 3, 2019
- **Labs**
  - Lab 4 solutions posted
  - Lab 5 posted – Memory Leaks & Makefile
- **Reading Week – Next Week (No Class and No Labs)**
- **Midterm**
  - Monday, February 25th (7 PM SW143, SW128, SW319)
  - Wednesday, February 27th (12 PM SW143) – Makeup (only those who are approved)

Last Week

- C Programming:
  - Tricks and Tips
    - Passing arguments to functions
    - Passing command line args to program
    - typedef
    - Organizing your program
    - Makefiles
    - Pre-processor directives
    - Error handling
  - File I/O

KING Chapters 13, 14, 15, 16, 17, 22, 24

I/O in C

- Have already seen library functions (from stdio.h) in sample programs:

```
int x = 10;
printf("The value of x is %d\n", x);
```

- Printing a character:

```
–int putchar(int c);
```

- Printing a string

```
–int puts(const char *s);
```

- These all print to stdout (standard output = screen)
  –What if we want to print to a file?

File I/O in C

- Have already seen library functions (from stdio.h) in sample programs:

```
int x = 10;
printf("The value of x is %d\n", x);
–int fprintf(FILE *stream, const char *format, ...);
```

- Printing a character:

```
–int putchar(int c);
–int fputc(int c, FILE *stream);
```

- Printing a string

```
–int puts(const char *s);
–int fputs(const char *s, FILE *stream);
```

- What is `FILE *stream`?

File interfaces in C/Unix

- Two main mechanisms for managing file access:
- File descriptors (low-level):
  - Each open file is identified by a small integer
  - Use for pipes, sockets (will see later what those are …)
- File pointers (regular files):
  - You use a pointer to a file structure (`FILE *`) as handle to a file.
  - The file struct contains a file descriptor and a buffer.
  - Use for regular files

Standard streams

- All programs automatically have three files open:
  - `FILE *stdin;`
  - `FILE *stdout;`
  - `FILE *stderr;`

	stdio name	File descriptor
Standard input	`stdin`	0
Standard output	`stdout`	1
Standard error	`stderr`	2

Comes by default from the keyboard.

Go by default to the screen.

- Those are ready to use:

```
fprintf(stdout, "Hello!\n"); // Identical to printf("Hello!\n");
fputs("Error!\n", stderr);
```


Operating on regular files

- A file first needs to be opened to obtain a `FILE *`

```
FILE *fopen(const char *filename,
  const char *mode);
```

  - `filename` identifies the file to open.
  - `mode` tells how to open the file:
    - "r" for reading, "w" for writing, "a" for appending
  - returns a pointer to a `FILE` struct which is the handle to the file. This pointer will be used in subsequent operations.
  - To close a file: `void fclose(FILE *stream);`

```
FILE *fp = fopen("my_file.txt", "w");
fprintf(fp, "Hello!\n");
fclose(fp);
```


Reading from files

- `char *fgets(char *s, int size, FILE *stream);`
  - Reads until a `\n` or `size-1` characters, whichever is first
  - Stores characters in buffer `s` points to
  - If a newline is read, it is stored into the buffer too.
  - Always null-terminates the string

- How would you use fgets to read from the keyboard?

- There is also another function to read from keyboard:
  `char *gets(char *s);`
  - Reads from keyboard until `\n` and stores results in buffer `s` points to
  - Why should you <u>never</u> use `gets`?

Example for reading from files

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

#define MAX 100

int main() {
   char buffer[MAX];
   FILE *fp = fopen("my_file.txt", "r");

   if( fp == NULL )
   {
      perror("Error while opening the file.\n");
      exit(1);
   }

    while ((fgets(buffer, MAX, fp)) != NULL){
         printf("%s", buffer);
    }
    fclose(fp);
  ………
```


Block I/O (aka binary I/O)

- Suppose a program wants to store a large set of numbers to file for later re-use (not human consumption).
- Could use fprintf for that. Why shouldn't we?

- Suppose you store the number 1,999,999 as follows:
  `fprintf(fp, "1999999");`

- How many bytes does this take up in the file?
  00110001 00111001 00111001 00111001 00111001 00111001 00111001
- How many bytes should it really take to store this number?
  00011110 10000100 01111111
- Block I/O allows you to read and write binary data, i.e. you read and write byte-for-byte rather than lines of characters.

Binary I/O

- Recall that `fgets` reads characters.
- By contrast, `fread` and `fwrite` operate on bytes.

```
size_t  fread(void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

  – read `nmemb` * `size` bytes into memory at `ptr`
  – returns number of items read

```
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

  – write `nmemb` * `size` bytes from `ptr` to the file pointer `stream`
  – returns number of items written

Example using
```
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

```
/* write an integer to the file */
int num = 1999999;
n = fwrite(&num, sizeof(num), 1, fp);


/* write a struct to the file */
struct rec {
  char name[20];
  int num;
} r;
r.num = 42;
strncpy(r.name, "koala", 20);
n = fwrite(&r, sizeof(r), 1, fp);
```


Example using
```
size_t  fread(void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

```
/* read an integer from the file */
int num;
n = fread(&num, sizeof(num), 1, fp);

/* read a struct from the file */
struct rec r;
n = fread(&r, sizeof(r), 1, fp);

/* display the contents of the variables */
printf("%d %s %d\n", num, r.name, r.num);
```

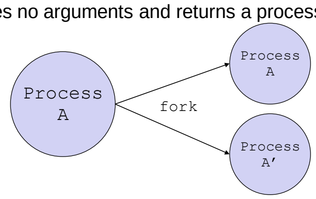

The plan for today

- Processes – Unix System Calls
  - fork()
  - wait()
  - exec()

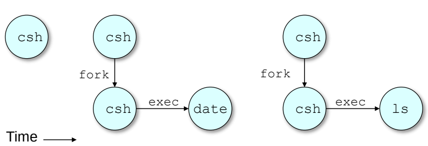
Haviland Chapters 5

Processes can create other processes

- The fork system call creates a **child** process:
  ```
  #include <unistd.h>
  pid_t fork(void);
  ```
- The fork system call creates a **duplicate** of the currently running program.
- Both processes run concurrently and independently
- It takes no arguments and returns a process ID.



Example: A shell



- When a command is typed, shell forks and then execs the typed command.

What is different between parent and forked child

- The fork system call creates a child process:
  ```
  #include <unistd.h>
  pid_t fork(void);
  ```

- The child gets a new PID (process ID) and PPID
- The return value from the fork call is different:
  - On success
    - fork() returns 0 to the child
    - fork() returns the child's PID to the parent.
  - On failure (no child created) fork returns -1 to parent

Fork example

```
int main ()
{
   pid_t pid;
   pid = fork();
   if (pid < 0) {
      perror("fork()");
   } else if (pid > 0) {
      printf("parent\n");
   } else {  /* pid == 0 */
      printf("child\n");
   }
   return 0;
}
```


Another fork example

Original process (parent)

```
int i; pid_t pid;
i = 5;
printf("%d\n", i);
/* prints 5 */
pid = fork();
/* pid == 677 */
if (pid > 0)
    i = 6;
else (pid == 0)
    i = 4;
printf("%d\n", i);
```

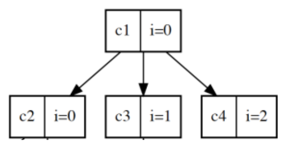
Child process

```
int i; pid_t pid;
i = 5;
printf("%d\n", i);

pid = fork();
/* pid == 0 */
if (pid > 0)
    i = 6;
else if (pid == 0)
    i = 4;
printf("%d\n", i);
```

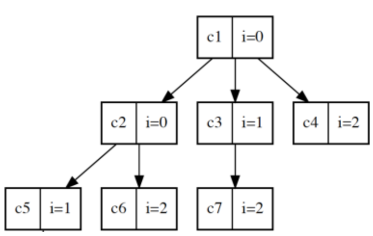

Yet another fork example

```
int main (void) {
  int i;
  pid_t child;

  for (i = 0; i < 3; i++) {
    child = fork();
    if (child == 0)
      printf ("Child %d saying hi\n", getpid());
  }
  return 0;
}
```

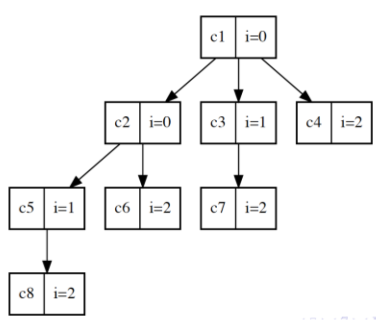

Yet another fork example

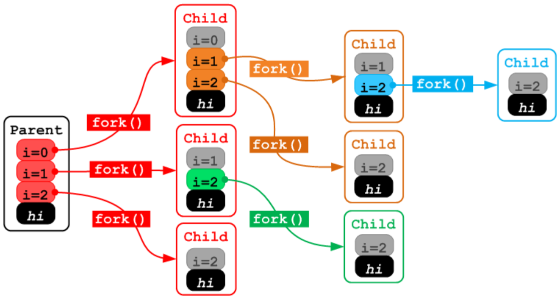


Yet another fork example



Yet another fork example





When does a child terminate?

- Like any other program:
  - The program's main function returns
  - The program calls `exit`
    - `void exit(int status);`
  - The program receives a signal that causes it to terminate
    - Will see next lecture what that means …
- Remember that programs have an exit status
  - Exit status of most recent command stored in `$?` in most shells
- Exit status is set by call to `exit` or main's `return`

`wait()`

- A parent might want to wait for a child to complete
- A parent might want to know the exit code of a child

- System call to wait for a child
  - `pid_t wait(int *status)`

- `wait` suspends execution of the calling process until one of its children terminates

`wait()`

- System call to wait for a child
  - `pid_t wait(int *status)`

- After calling wait() a process will:
  - block if all of its children are still running
  - return immediately with the PID of a terminated child, if there is a terminated child
  - return immediately with an error (-1) if it doesn't have any child processes.

Info returned by `wait()`

- System call to wait for a child
  - `pid_t wait(int *status)`

- Returns the pid of the terminated child or -1 on error

- `status` encodes the exit status of the child and how a child exited (normally or killed by signal)
- There are macros to process exit status:
  - `WIFEXITED` tells you if child terminated normally
  - `WEXITSTATUS` gives you the exit status

Example using `wait()`

```
int main (void) {
    pid_t child;
    int status, exit_status;

    if ((child = fork()) == 0) {
      sleep (5);
      exit (8);
    }
    wait (&status);
    if (WIFEXITED(status)) {
     exit_status = WEXITSTATUS(status);
     printf ("Child %d done: %d\n", child, exit_status);
    }
    return 0;
}
```

How does a child become a zombie?

- When a child terminates, but its parent process is not `wait`ing for it
- The child (its exit code) is kept around as a zombie until parent collects its exit code through `wait`
  - or until parent terminates
- Shows up as `Z` in `ps`

How does a child become an orphan?

- If the parent process terminates before the child
- Who is now the new parent?
  - Orphans get adopted by the `init` process
  - `init` is the first process started during booting
  - It's the root of the process hierarchy
  - `init` has a PID of 1
  - The PPID of orphans is 1

waitpid

- What if a process wants to wait for a particular child (rather than any child)
- What if a process does not want to block when no child has terminated?

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- First parameter specifies PID of child to wait for
- If options is 0, waitpid blocks (just like wait)
- If options is WNOHANG, it immediately returns 0 instead of blocking when no terminated child

Waitpid example

```
int main (void) {
  pid_t child;
  int status, exit_status;
  if ((child = fork()) == 0) {
      sleep (5);
      exit (8);
  }
  while (waitpid (child, &status, WNOHANG) == 0) {
     printf ("Waiting...\n");
     sleep (1);
  }
  if (WIFEXITED(status)) {
    exit_status = WEXITSTATUS(status);
    printf ("Child %d done: %d\n", child, exit_status);
  }
  ….
```

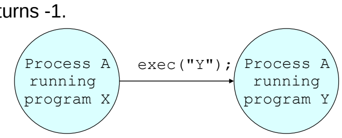

Wait a second ….

- Fork creates a duplicate of the current process
- How do we actually create a <u>new</u> process that runs a different program???

*When a program wants to have another program running in parallel, it will typically first use **fork**, then the child process will use **exec** to actually run the desired program.*

Exec

- The exec system call replaces the program being run by a process by a different one.
- The new program starts executing from the beginning.
- On success, exec never returns, on failure, exec returns -1.



Exec example

Program X

```
int i = 5;
printf("%d\n", i);
exec("Y");
printf("%d\n", i);
```

Program Y

```
printf("hello\n");
```


exec properties

- New process inherits from calling process:
  - PID and PPID, UID, GID
  - controlling terminal
  - CWD, resource limits
  - pending signals

exec()

- exec() is not one specific function, but a family of functions:

```
execl(char *path, char *arg0, …, (char *)NULL);
execv(char *path, char *argv[]);
execlp(char *file, char *arg0, …,(char *)NULL);
execvp(char *file, char *argv[]);
```

- First parameter: name of executable; then commandline parameters for executable; these are passed as argv[0], argv[1], …, to the main program of the executable.
- `execl` and `execv` differ from each other only in how the arguments for the new program are passed
- `execlp` and `execvp` differ from `execl` and `execv` only in that you don't have to specify full path to new program

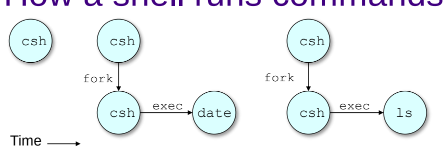
Example using execl

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

int main (void) {
   printf ("Before exec\n");
   execl ("/bin/ls", "ls", "-l", (char *)NULL);
   perror ("execl");
   exit (1);
}
```


Example using execv

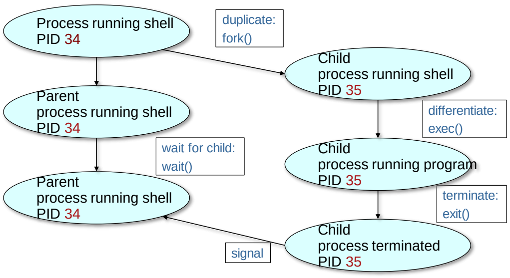
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main (void) {
  char *args[] = {"ls", "-l", NULL};
  printf ("Before exec\n");
  execv ("/bin/ls", args);
  perror ("execv");
  exit (1);
}
```


How a shell runs commands



- When a command is typed, shell forks and then execs the typed command.

How a shell runs commands



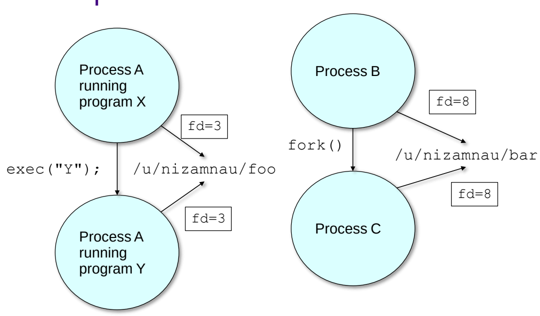
Shell skeleton

```
while (1) {     // infinite while loop
        print_prompt();
        read_command(command, parameters);
        if (fork()) {
            wait(&status);
        } else {
            execve(command, parameters, NULL);
        }
}
```

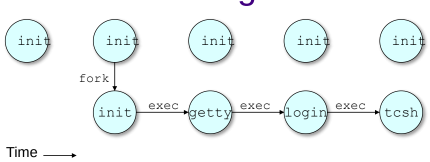

Processes and File Descriptors

- File descriptors are handles to open files.
- They belong to processes not programs.
- They are a process’s link to the outside world.

FDs preserved across fork and exec



Initializing Unix



- See “top”, “ps –aux” to see what’s running
- The **only** way to create a new process is to duplicate an existing process. Therefore the ancestor of **all** processes is `init` with pid = 1
- The **only** way to run a program is with `exec`

Shell Job control

- A job (or process) is program in execution
  - Use `ps` to view processes
- Foreground job: has control of the terminal
- Background job: runs concurrently with shell in the background
  - To run a program in the background append & to the name of the program
- At any point a program can be running or suspended
  - Hit `<ctrl> z` to suspend the current foreground job

Shell Job control

- `jobs` gives you a list of jobs; each is associated with a job number
- `fg [num]` puts job `num` in the foreground
- `bg [num]` puts job `num` in the background
- `kill %num` kills job `num`
  - You can kill the current foreground job with Ctrl-c
  - You can suspend (pause) current foreground job with Ctrl-z

That's it for today!