CS 360 Internet Programming
Client-Server Networking
*Serving Dynamic Content*

Daniel Zappala
Computer Science
Brigham Young University
1 Serving Dynamic Content
   - Types of Dynamic Content
   - Passing Data To A Script
   - Killing Long Running Scripts

2 Executing a GET Script
   - Linux System Calls
   - Example

3 Executing a POST Script
   - Linux System Calls
   - Example
Types of Dynamic Content

- server-side includes
  - resource includes macros or PHP code instructing the server to perform some operation
  - higher overhead than simply delivering a file
  - may require an interpreter

- server scripts
  - a separate program generates the resource
  - may have access to a local database
  - may be run by a separate process, a module inside the web server, or a persistent process
Serving Dynamic Content
Executing a GET Script
Executing a POST Script

Types of Dynamic Content
Passing Data To A Script
Killing Long Running Scripts

Passing Data To A Script

- environment variables: GET, POST
  - useful information such as remote host name, user agent
  - necessary information, such as request method, query string
- standard input: POST
- CGI standardizes how data is passed
  - http://www.w3.org/CGI/
  - http://hoohoo.ncsa.uiuc.edu/cgi/interface.html
CGI Environment Variables

- **general**
  - SERVER_SOFTWARE=name/version
  - SERVER_NAME=hostname
  - GATEWAY_INTERFACE=CGI/revision (CGI/1.1)

- **GET** (*GET /book.cgi?name=Rexford*)
  - REQUEST_METHOD=GET
  - QUERY_STRING=name=Rexford (everything after ? in URL)

- **POST** (*POST /book.cgi*)
  - REQUEST_METHOD=POST
  - pass arguments from entity body via standard input
Killing Long Running Scripts

- keep an array of running processes with start time
- set a timer that uses a signal
- when signal occurs, check the array and kill long running processes
Overview: GET

1. read and parse the HTTP headers
2. setup environment variable array and argument array
3. call fork()
4. child process
   - use dup2() to setup standard output to write to the socket
   - call execve() to execute new process that runs the script
5. parent process
   - wait for child to finish
fork

1. `#include <sys/types.h>`
2. `#include <unistd.h>`
3. `pid_t fork(void);`

- creates a new process by duplicating the calling process
- shares most resources but has a different process ID
- returns 0 to child, new process ID to parent
wait

1  #include <sys/types.h>
2  #include <sys/wait.h>
3
4  pid_t wait(int *status);

- waits for any child process to terminate
- returns process ID of child that termintes
- if not NULL, status is a code that indicates what happened to the child
execve

```c
#include <unistd.h>

int execve(const char *filename, char *const argv[], char *const envp[]);```

- executes a program pointed to by `filename`
- `argv`: array of argument strings
- `envp`: array of environment variables in format of key=value
- on success, does not return
- on error, returns -1 and sets errno
### dup2

1. `#include <unistd.h>`
2.
3. `int dup2(int oldfd, int newfd);`

- copies old file descriptor to the new file descriptor
- useful for child process to write to standard output and have it go to another descriptor (eg a socket, file, or pipe)
- on success, returns new file descriptor
- on error, returns -1 and sets errno
Executing a GET script

```c
// read and parse the HTTP headers
// setup environment variable array

pid = fork();
if (pid == 0) {
  // child process
  // make standard output write to the socket
  dup2(socket, 1);
  execve(....);
  perror("execve");
}
wpid = wait(NULL);
```
Overview: POST

1. read and parse the HTTP headers and body
2. setup environment variable array and argument array
3. create a pipe
4. call fork()
5. child process
   - setup standard input to read from the pipe
   - use dup2 to setup standard output to write to the socket
   - call execve() to execute new process that runs the script
6. parent process
   - read the entity body
   - write the entity body to the pipe
   - wait for child to finish


```c
#include <unistd.h>

int pipe(int filedes[2]);
```

- creates a pipe for sending information between two processes
- `filedes[0]` is for reading
- `filedes[1]` is for writing
- on success returns zero
- on error returns -1 and sets errno
Executing a POST script (child)

```c
int pipefd[2];
pipe(pipefd);

pid = fork();
if (pid == 0) {
    // child process
    // close the write side of the pipe
    close(pipefd[1]);
    // make standard input read from the pipe
    dup2(pipefd[0], 0);
    dup2(socket, 1);

    execve(....);
    perror("execve");
}
```
Executing a POST script (parent)

```c
1 // parent process
2 // close the read side of the pipe
3 close(pipefd[0]);
4 // write body of HTTP request to pipefd[1]
5 writeMessage(pipefd[1], body);
6 close(pipefd[1]);
7 wpid = wait(NULL);
```