

# Lecture 29: Concurrency with processes

David Hovemeyer

November 18, 2020

601.229 Computer Systems Fundamentals



# Web server

Main web server loop:

```
while (1) {  
    int clientfd = Accept(serverfd, NULL, NULL);  
    if (clientfd < 0) { fatal("Error accepting client connection"); }  
    server_chat_with_client(clientfd, webroot);  
    close(clientfd);  
}
```

Do you see any limitations of this design?

# Web server

Main web server loop:

```
while (1) {  
    int clientfd = Accept(serverfd, NULL, NULL);  
    if (clientfd < 0) { fatal("Error accepting client connection"); }  
    server_chat_with_client(clientfd, webroot);  
    close(clientfd);  
}
```

Do you see any limitations of this design?

The server can only communicate with one client at a time

# Concurrency

In general, servers (including web servers) can receive requests from many clients, *simultaneously*

*Concurrency*: Processing involving multiple tasks that can execute *asynchronously* with respect to each other

- ▶ E.g., multiple server/client conversations could be ongoing at the same time

It would be good if our web server could serve multiple clients concurrently

# Concurrency vs. parallelism

*Concurrency* is distinct from *parallelism*

Consider two tasks, A and B, consisting of a sequence of instructions

# Concurrency vs. parallelism

*Concurrency* is distinct from *parallelism*

Consider two tasks, A and B, consisting of a sequence of instructions

A and B execute *concurrently* if relative ordering of instructions in A and B is not guaranteed

- ▶ I.e., an instruction in A could happen either “before” or “after” an instruction in B

# Concurrency vs. parallelism

*Concurrency* is distinct from *parallelism*

Consider two tasks, A and B, consisting of a sequence of instructions

A and B execute *concurrently* if relative ordering of instructions in A and B is not guaranteed

- ▶ I.e., an instruction in A could happen either “before” or “after” an instruction in B

A and B execute in *parallel* if instructions in A and B can execute *at the same time*

- ▶ Parallel execution requires multiple processors or cores

# Concurrency vs. parallelism

*Concurrency* is distinct from *parallelism*

Consider two tasks, A and B, consisting of a sequence of instructions

A and B execute *concurrently* if relative ordering of instructions in A and B is not guaranteed

- ▶ I.e., an instruction in A could happen either “before” or “after” an instruction in B

A and B execute in *parallel* if instructions in A and B can execute *at the same time*

- ▶ Parallel execution requires multiple processors or cores

Parallelism implies concurrency, but concurrency does not imply parallelism



# Concurrency with processes

# Multi-process web server

Code on web page: `mp_webserver.zip`

- ▶ Only the `main` function is different than original `webserver.zip`

We'll discuss some of the interesting implementation issues

# Processes

We've seen that the `fork` system call makes a new child process that is a duplicate of the parent process

- ▶ Including inheriting open files

# Processes

We've seen that the `fork` system call makes a new child process that is a duplicate of the parent process

- ▶ Including inheriting open files

Idea: each time the server accepts a connection, fork a child process to handle communication with that client

Multiple child processes can be executing concurrently

- ▶ OS kernel is responsible for allocating CPU time and handling I/O

Issue: we may want to limit the number of simultaneous child processes

- ▶ Processes are somewhat heavyweight in terms of system resources

Before starting a child process, the server loop will wait to make sure fewer than the maximum number of child processes are running

# wait, SIGCHLD

Several system calls exist to allow a parent process to receive a child process's exit status (wait, waitpid)

If a child terminates but the parent doesn't wait for it, it can become a zombie

A parent process can handle the SIGCHLD signal in order to be notified when a child process exits

# wait, SIGCHLD

Several system calls exist to allow a parent process to receive a child process's exit status (`wait`, `waitpid`)

If a child terminates but the parent doesn't wait for it, it can become a zombie

A parent process can handle the `SIGCHLD` signal in order to be notified when a child process exits

Idea: parent will keep a count of how many child processes are running: use `wait` system call and `SIGCHLD` signal handler to detect when child processes complete

# Signal handlers

The `signal` and `sigaction` system calls can be used to register a *signal handler* function for a particular signal

Signal handler for the `SIGCHLD` signal, so server is notified when a child process terminates:

```
/* current number of child processes running */
int g_num_procs;

void sigchld_handler(int signo) {
    int wstatus;
    wait(&wstatus);
    if (WIFEXITED(wstatus) || WIFSIGNALED(wstatus)) {
        g_num_procs--;
    }
}
```



# Registering a signal handler

Register the `sigchld_handler` function as a handler for the `SIGCHLD` signal:

```
struct sigaction sa;  
sigemptyset(&sa.sa_mask);  
sa.sa_flags = 0;  
sa.sa_handler = sigchld_handler;  
sigaction(SIGCHLD, &sa, NULL);
```

When a child process terminates, the OS kernel will deliver a `SIGCHLD` signal, and the `sigchld_handler` function will be called

# Preparing to fork

Before forking a child process, the server will wait until the number of processes is at least one less than the maximum:

```
while (g_num_procs >= MAX_PROCESSES) {  
    int wstatus;  
    wait(&wstatus);  
    if (WIFEXITED(wstatus) || WIFSIGNALED(wstatus))  
        g_num_procs--;  
}
```

```
int clientfd = Accept(serverfd, NULL, NULL);
```

```
g_num_procs++;  
pid_t pid = fork();
```

(Does this work?)

# A data race

Consider the loop to wait until `g_num_procs` is less than the maximum:

```
while (g_num_procs >= MAX_PROCESSES) {  
    int wstatus;  
    wait(&wstatus);  
}
```

The thing to understand about signals is that, in general, they can be delivered at *any* time

Imagine that SIGCHLD is received *after* checking `g_num_procs` but *before* calling `wait`

# A data race

Consider the loop to wait until `g_num_procs` is less than the maximum:

```
while (g_num_procs >= MAX_PROCESSES) {  
    int wstatus;  
    wait(&wstatus);
```

The thing to understand about signals is that, in general, they can be delivered at *any* time

Imagine that `SIGCHLD` is received *after* checking `g_num_procs` but *before* calling `wait`

Assuming that `sigchld_handler` detects that a child process has exited, the call to `wait` is unnecessary

# A data race

Consider the loop to wait until `g_num_procs` is less than the maximum:

```
while (g_num_procs >= MAX_PROCESSES) {  
    int wstatus;  
    wait(&wstatus);  
}
```

The thing to understand about signals is that, in general, they can be delivered at *any* time

Imagine that `SIGCHLD` is received *after* checking `g_num_procs` but *before* calling `wait`

Assuming that `sigchld_handler` detects that a child process has exited, the call to `wait` is unnecessary

- ▶ If `MAX_PROCESSES` is 1, server is deadlocked!

# Another data race

Consider the following seemingly innocuous statement:

```
g_num_procs--;
```

The code generated by the compiler is likely to be something similar to:

```
int tmp = g_num_procs;  
tmp = tmp - 1;  
g_num_procs = tmp;
```

Note that tmp would really be a register

# Another data race

Consider the following seemingly innocuous statement:

```
g_num_procs--;
```

The code generated by the compiler is likely to be something similar to:

```
int tmp = g_num_procs;  
tmp = tmp - 1;  
g_num_procs = tmp;
```

Note that tmp would really be a register

Consider what happens if a SIGCHLD signal is received *after* the initial value of g\_num\_procs is read, but *before* the updated value of tmp is stored back to g\_num\_procs

# Another data race

Consider the following seemingly innocuous statement:

```
g_num_procs--;
```

The code generated by the compiler is likely to be something similar to:

```
int tmp = g_num_procs;  
tmp = tmp - 1;  
g_num_procs = tmp;
```

Note that tmp would really be a register

Consider what happens if a SIGCHLD signal is received *after* the initial value of g\_num\_procs is read, but *before* the updated value of tmp is stored back to g\_num\_procs

- ▶ A decrement of g\_num\_procs (in sigchld\_handler) is lost, and the server no longer knows how many child processes are running!



# Data race explained

Consider code implementing `g_num_procs--`:

```
// Assume tmp is a register
```

```
int tmp = g_num_procs;
```

```
tmp = tmp - 1;
```

```
g_num_procs = tmp;
```

# Data race explained

Consider code implementing `g_num_procs--`:

*// Assume tmp is a register*

`int tmp = g_num_procs;`      *value of g\_num\_procs loaded to tmp*

`tmp = tmp - 1;`

`g_num_procs = tmp;`

# Data race explained

Consider code implementing `g_num_procs--`:

```
// Assume tmp is a register
```

```
int tmp = g_num_procs;
```

```
tmp = tmp - 1;
```

```
g_num_procs = tmp;
```

*SIGCHLD handled, g\_num\_procs decremented*

# Data race explained

Consider code implementing `g_num_procs--`:

```
// Assume tmp is a register
```

```
int tmp = g_num_procs;
```

```
tmp = tmp - 1;
```

```
g_num_procs = tmp;
```

*tmp (old value of g\_num\_procs) decremented*

# Data race explained

Consider code implementing `g_num_procs--`:

```
// Assume tmp is a register
```

```
int tmp = g_num_procs;
```

```
tmp = tmp - 1;
```

```
g_num_procs = tmp;
```

invalid count stored in `g_num_procs`

# Data race explained

Consider code implementing `g_num_procs--`:

```
// Assume tmp is a register
```

```
int tmp = g_num_procs;
```

```
tmp = tmp - 1;
```

```
g_num_procs = tmp;
```

Oops!

# Data race

A *data race* is a (potential) bug where two concurrently-executing paths access a shared variable, and at least one path writes to the variable

- ▶ Paths “race” to access shared data, outcome depends on which one “wins”

# Data race

A *data race* is a (potential) bug where two concurrently-executing paths access a shared variable, and at least one path writes to the variable

- ▶ Paths “race” to access shared data, outcome depends on which one “wins”

Data race is a special case of a *race condition*, a situation where an execution outcome depends on unpredictable event sequencing

A data race can cause data invariants to be violated (e.g., “g\_num\_procs accurately reflects the number of processes running”)



# Data race

A *data race* is a (potential) bug where two concurrently-executing paths access a shared variable, and at least one path writes to the variable

- ▶ Paths “race” to access shared data, outcome depends on which one “wins”

Data race is a special case of a *race condition*, a situation where an execution outcome depends on unpredictable event sequencing

A data race can cause data invariants to be violated (e.g., “g\_num\_procs accurately reflects the number of processes running”)

Solution: *synchronization*

- ▶ Implement a protocol to avoid uncontrolled access to shared data

# sigprocmask to the rescue

Signal handler functions are a potential cause of data races because they execute asynchronously with respect to normal program execution

- ▶ OS kernel could deliver a signal at any time

# sigprocmask to the rescue

Signal handler functions are a potential cause of data races because they execute asynchronously with respect to normal program execution

- ▶ OS kernel could deliver a signal at any time

`sigprocmask`: allows program to block and unblock a specific signal or signals

# sigprocmask to the rescue

Signal handler functions are a potential cause of data races because they execute asynchronously with respect to normal program execution

- ▶ OS kernel could deliver a signal at any time

`sigprocmask`: allows program to block and unblock a specific signal or signals

Idea: block `SIGCHLD` whenever `g_num_procs` is being accessed by program code

- ▶ Prevent `sigchld_handler` from unexpectedly modifying `g_num_procs`

# blocking/unblocking SIGCHLD

toggle\_sigchld function:

```
void toggle_sigchld(int how) {  
    sigset_t sigs;  
    sigemptyset(&sigs);  
    sigaddset(&sigs, SIGCHLD);  
    sigprocmask(how, &sigs, NULL);  
}
```

Use to protect accesses to g\_num\_procs:

```
toggle_sigchld(SIG_BLOCK);  
g_num_procs++;  
toggle_sigchld(SIG_UNBLOCK);
```

# Back to the web server!

Web server main loop:

```
while (1) {  
    wait_for_avail_proc();  
    int clientfd = accept connection from client  
    toggle_sigchld(SIG_BLOCK);  
    g_num_procs++;  
    toggle_sigchld(SIG_UNBLOCK);  
    pid_t pid = fork();  
    if (pid < 0) {  
        fatal("fork failed");  
    } else if (pid == 0) { /* in child */  
        server_chat_with_client(clientfd, webroot);  
        close(clientfd);  
        exit(0);  
    }  
    close(clientfd);  
}
```

# File descriptor sharing

When a subprocess is forked, the child process inherits the parent process's file descriptors

In the web server, the forked child process inherits `clientfd`, the socket connected to the client

- ▶ Convenient, since we want the child process to handle the client's request

# File descriptor sharing

When a subprocess is forked, the child process inherits the parent process's file descriptors

In the web server, the forked child process inherits `clientfd`, the socket connected to the client

- ▶ Convenient, since we want the child process to handle the client's request

Important: the *parent* process must close `clientfd`, otherwise the web server will have a file descriptor leak

- ▶ OS kernel imposes limit on number of open files per process
- ▶ Too many file descriptors open → can't open any more files or sockets



# Limiting number of processes

Before calling fork, web server calls wait\_for\_avail\_proc:

```
void wait_for_avail_proc(void) {
    toggle_sigchld(SIG_BLOCK);
    while (g_num_procs >= MAX_PROCESSES) {
        int wstatus;
        wait(&wstatus);
        if (WIFEXITED(wstatus) || WIFSIGNALED(wstatus)) {
            g_num_procs--;
        }
    }
    toggle_sigchld(SIG_UNBLOCK);
}
```

Calls wait if too many processes are currently running

# Interrupted system calls

When a program receives a signal, it can interrupt the currently-executing system call

Special handling is required for accept system call to wait for connection from client:

```
int clientfd;
do {
    clientfd = accept(serverfd, NULL, NULL);
} while (clientfd < 0 && errno == EINTR);
if (clientfd < 0) {
    fatal("Error accepting client connection");
}
```

When `errno` is `EINTR`, it indicates that the system call was interrupted

# Async-signal safety

While we're talking about signals...

Because of the potential of signal handlers to introduce data races into the program, some library functions aren't safe to call from a signal handler

Good idea to know these: `man signal-safety` on Linux

Standard I/O routines (printf, scanf, etc.) are not async-signal safe ☹

# Putting it together

In the `mp_webserver` directory:

```
$ gcc -o mp_webserver main.c webserver.c csapp.c -lpthread
```

```
$ ./mp_webserver 30000 ./site
```

# Result

Visiting URL `http://localhost:30000/index.html`:

