Lecture 08: Race Conditions, Deadlock, and Data Integrity

Principles of Computer Systems

Autumn 2019

Stanford University

Computer Science Department

Lecturers: Chris Gregg and

Philip Levis

PDF of this presentation

Masking Signals and Deferring Handlers, Revisited

- Signals can execute at any time, preempting your main code
- Preemption can create *race conditions*, which is when the timing of execution can lead to incorrect output and corrupt data

https://cplayground.com/?p=aardvark-monkey-lobster

Masking Signals and Deferring Handlers

- One way to prevent race conditions is a *critical section*
- A critical section is a piece of code that executes *atomically*
  - The code is "indivisible" -- other code sees either before it executes or after
  - Other code never sees "during" the critical section/atomic code
- For signals, **`sigprocmask`** lets us define define critical sections
  - Recall, allows code to block signals
  - Code that executes while a signal is blocked is atomic with respect to that signal's handler: the handler executes either before the signal is blocked or after it is unblocked, but never during while it is blocked
- Another approach is to use race-free data structures
  - We won't go into great depth in this class
  - Simple ones are great (e.g., a circular queue), more complex ones are very tricky

Critical Section Example with sigprocmask

https://cplayground.com/?p=jay-kangaroo-mandrill

The invocation model of signals

Signal	SIGALRM	SIGCHLD	SIGUSR1	SIGUSR2	SIGINT
Pending	0	1	0	0	0
Enabled	1	0	1	0	1

- In practice, signals are implemented in a more complicated fashion than this, but this is their basic invocation model -- it's designed to resemble hardware interrupts
- When a signal arrives, set pending to 1
- Enabled is whether the signal is blocked (false) or not (true)
- Any time pending or enabled changes, if pending && enabled, deliver the signal
  - Atomically clear pending and enabled
  - When handler completes, restore enabled (unless it was blocked in handler)

Race Conditions and Concurrency

- Race conditions are a fundamental problem in concurrent code
  - Decades of research in how to detect and deal with them
- The can corrupt your data and violate its integrity, so it is no longer consistent
- Critical sections can prevent race conditions, but there are two major challenges
  - Figuring out exactly where to put all the critical sections
  - Structuring your code so critical sections don't limit performance

- Example of challenge 1: You have a tree data structure in your program. A signal handler prints out the tree. Your main code inserts and deletes from the tree. You need to make sure every update to the tree executes atomically, so a signal handler never sees a bad pointer.

- Example of challenge 2: if your code spends most of its time in long critical sections, then signals may be delayed for a long time (making your program less responsive).

Compilers and Visibility

- The assembly your compiler generates is not a linear, literal version of your program!
  - Variables are cached in registers
  - Statements can be re-ordered and dead code deleted
- The basic rule: the compiler only promises that you *see* something consistent with a linear, literal execution of your program
  - When might you see your program? When external functions are called (e.g., write(2))
  - In between these visibility points it can play lots of tricks

```
int x, y;
x = 5;
y = 7;
printf("%d %d\n", x, y)
```

Your compiler doesn't have to allocate space for x and y -- it can just pass constants to printf.

```
int x, y;
x = 5;
y = 7;
print_ints(&x, &y);
```

Your compiler has to allocate space for x and y -- it doesn't know what print_ints will do.

```
1 int x, y;
2 ...
3 x++;
4 y = x + 1;
5 print_ints(&x, &y);
```

Your compiler could have x in a register *r*, then store *r* + 1 in x and *r* + 2 in y. If x were modified between lines 3 and 4, y will still be stored as *r* + 2.

Detailed Example: Background Process Management and Cleanup

- Let's revisit the **`simplesh`** example from last week. The full program is right here.

```c
// simplesh.c
int main(int argc, char *argv[]) {
  while (true) {
    // code to initialize command, argv, and isbg omitted for brevity
    pid_t pid = fork();
    if (pid == 0) execvp(argv[0], argv);
    if (isbg) {
      printf("%d %s\n", pid, command);
    } else {
      waitpid(pid, NULL, 0);
    }
  }
  printf("\n");
  return 0;
}
```

- The problem to be addressed: Background processes are left as zombies for the lifetime of the shell. At the time we implemented **`simplesh`**, we had no choice, because we hadn't learned about signals or signal handlers yet.

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- Now we know about **`SIGCHLD`** signals and how to install **`SIGCHLD`** handlers to reap zombie processes. Let's upgrade our **`simplesh`** implementation to reap *all* process resources.

https://cplayground.com/?p=lapwing-rabbit-otter

Problem: Redundant Calls to waitpid

- Relies on a sketchy call to **`waitpid`** to halt the shell until its foreground process has exited.
  - When the user creates a foreground process, **`waitpid`** executes in the main loop
  - When the foreground process finishes, however, the **`SIGCHLD`** handler will run too, it calls **`waitpid`**
  - One of them will return the **`pid`**, one will return an error
- We can incorporate extra logic to handle the fact that some calls to **`waitpid`** expect to return an error (e.g., suppress error messages), but this is hacking around a poor design

Solution: One to `waitpid` to rule them all, one `waitpid` to find them

- We want the only place that calls to be in the SIGCHLD handler
- If we run a process in the foreground, go to sleep and have the SIGCHLD handler wake us up when the foreground process completes
  - This is a common pattern in concurrent code: you're waiting for something complete, go to sleep until another piece of code wakes you up
  - **`pause(2)`** allows us to sleep until a signal handler executes, but this is too coarse: we need to go back to sleep if it wasn't for the foreground process
- Basic algorithm:
  - Use **`pause()`** to sleep until a signal handler executes
  - The signal handler sets a variable to tell the main loop whether the foreground process exited
  - When the main loop wakes up, it checks the variable, goes back to sleep if needed

Updated code

https://cplayground.com/?p=grouse-shrew-owl

Houston, we have a problem!

```c
static void reapProcesses(int sig) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, WNOHANG);
    if (pid <= 0) {
        break;
    } else if (pid == fgpid) {
        fgpid = 0;
    }
  }
}

static void waitForForegroundProcess(pid_t pid) {
   fgpid = pid;
   while (fgpid == pid) {
       pause();
   }
}

    pid_t pid = forkProcess();
    if (pid == 0) {...}
    if (isbg) {
      printf("%d %s\n", pid, command);
    } else {
      waitForForegroundProcess(pid);
    }
```

Race condition on `fgpid`

- It's possible the foreground process finishes and **`reapProcesses`** is invoked on its behalf **`before`** normal execution flow updates **`fgpid`**. If that happens, the shell will spin forever and never advance up to the shell prompt.
- This is a race condition: we want to atomically fork the process and update **`fgpid`**, such that **`reapProcesses`** does not execute before we set **`fgpid`**
- Solution: use **`sigprocmask`** to block SIGCHLD before **`fork`**, then unblock in child and in parent after **`fgpid`** is set

Fixed race condition on `fgpid`

https://cplayground.com/?p=goosander-lobster-pheasant

Can you find the race condition here?

```
static void waitForForegroundProcess(pid_t pid) {
   fgpid = pid;
   unblockSIGCHLD();
   while (fgpid == pid) {
       pause();
   }
}
```

Different kind of race condition

```
1 static void waitForForegroundProcess(pid_t pid) {
2    fgpid = pid;
3    unblockSIGCHLD();
4    while (fgpid == pid) {
5        pause();
6    }
7 }
```

- This is a race condition, because we need to atomically unblock SIGCHLD and pause, or we might miss the SIGCHLD and never wake up.
  - Suppose the SIGCHLD handler executes between lines 4 and 5 -- pause will never return
- This is a different problem that what we've seen before: no data is corrupted, but we might *deadlock*
- **Deadlock:** program state in which no progress can be made, code is waiting for something that will never happen

`sigsuspend` to the rescue

- The problem with both versions of **`waitForForegroundProcess`** on the prior slide is that each lifts the block on **`SIGCHLD`** before going to sleep via **`pause`**.
- The one **`SIGCHLD`** you're relying on to notify the parent that the child has finished could very well arrive in the narrow space between lift and sleep. That would inspire deadlock.
- The solution is to rely on a more specialized version of **`pause`** called **`sigsuspend`**, which asks that the OS change the blocked set to the one provided, but only *after* the caller has been forced off the CPU. When some unblocked signal arrives, the process gets the CPU, the signal is handled, the original blocked set is restored, and **`sigsuspend`** returns.

```c
// simplesh-all-better.c
static void waitForForegroundProcess(pid_t pid) {
  fgpid = pid;
  sigset_t empty;
  sigemptyset(&empty);
  while (fgpid == pid) {
    sigsuspend(&empty);
  }
  unblockSIGCHLD();
}
```

- This is the model solution to our problem, and one you should emulate in your Assignment 3 **`farm`** and your Assignment 4 **`stsh`**.

High-level takeaways: signals and concurrency

- Concurrency is powerful: it lets our code do many things at the same time
  - It can run faster (more cores!)
  - It can do more (run many programs in background)
  - It can respond faster (don't have to wait for current action to complete)
- Signals are a way for concurrent processes to interact
  - Send signals with kill and raise
  - Handle signals with signal
  - Control signal delivery with sigprocmask, sigsuspend
  - Preempt running code
  - Making sure code running in a signal handler works correctly is difficult
  - Specialized system calls (pause, sigsuspend) help, but there's still the compiler problem
- *Race conditions* occur when code can see data in an intermediate and invalid state (often KABOOM)
  - Prevent race conditions with *critical sections*
- *Deadlock* is when your program halts, waiting for something that will never happen

- Assignments 3 and 4 use signals, as a way to start easing into concurrency before we tackle multithreading
- Take CS149 if you want to learn how to write high concurrency code that runs 100x faster

Questions about signal handling

Example midterm question #1

- Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
static void bat(int unused) {
  printf("pirate\n");
  exit(0);
}

int main(int argc, char *argv[]) {
  signal(SIGUSR1, bat);
  pid_t pid = fork();
  if (pid == 0) {
    printf("ghost\n");
    return 0;
  }
  kill(pid, SIGUSR1);
  printf("ninja\n"); return 0;
}
```

- For each of the five columns, write a **yes** or **no** in the header line. Place a **yes** if the text below it represents a possible output, and place a **no** otherwise.

ghost ninja pirate	pirate ninja	ninja ghost	ninja pirate ninja	ninja pirate ghost

Example midterm question #1

- Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
static void bat(int unused) {
  printf("pirate\n");
  exit(0);
}

int main(int argc, char *argv[]) {
  signal(SIGUSR1, bat);
  pid_t pid = fork();
  if (pid == 0) {
    printf("ghost\n");
    return 0;
  }
  kill(pid, SIGUSR1);
  printf("ninja\n"); return 0;
}
```

- For each of the five columns, write a **yes** or **no** in the header line. Place a **yes** if the text below it represents a possible output, and place a **no** otherwise.

yes!	yes!	no!	no!	no!
ghost ninja pirate	pirate ninja	ninja ghost	ninja pirate ninja	ninja pirate ghost

Example midterm question #2

- Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs

Example midterm question #2

- Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs

- Possible Output 1: 112265
  Possible Output 2: 121265
  Possible Output 3: 122165

- If the **>** of the **`counter > 0`** test is changed to a **>=**, then **`counter`** values of zeroes would be included in each possible output. How many different outputs are now possible? (No need to list the outputs—just present the number.)

Example midterm question #2

- Consider this program and its execution. Assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

- List all possible outputs

- Possible Output 1: 112265
  Possible Output 2: 121265
  Possible Output 3: 122165

- If the **>** of the **`counter > 0`** test is changed to a **>=**, then **`counter`** values of zeroes would be included in each possible output. How many different outputs are now possible? (No need to list the outputs—just present the number.)
  - 18 outputs now (6 x the first number)

If we have time...

Playing with fire

https://cplayground.com/?p=aardvark-monkey-lobster

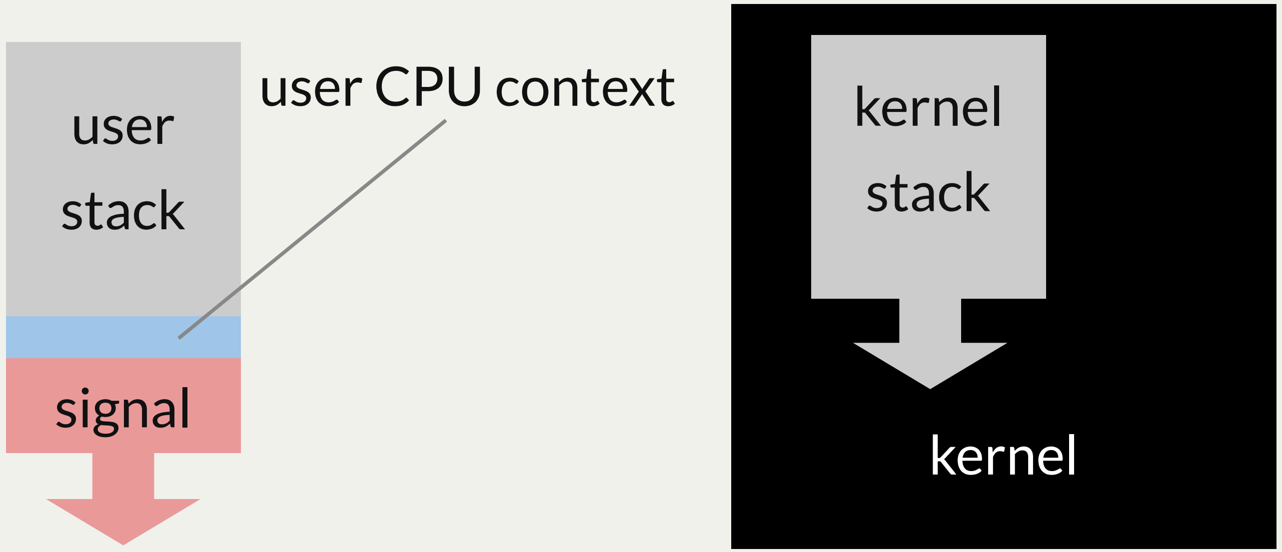
Playing with fire

```
signal
1570987083.612345 counter_1: 0, counter_2: 1
1570987083.612345 counter_1: 0, counter_2: 1
signal
```

??????

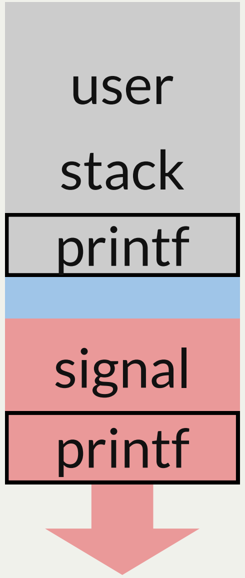
Reentrant code

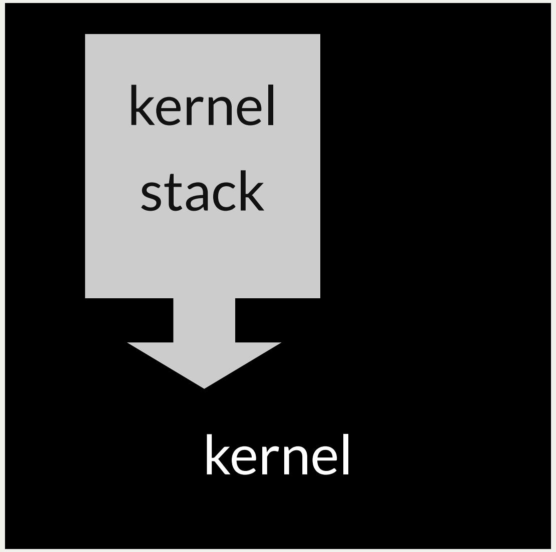
- Recall that the kernel has user processes execute signal handlers by pushing stack frames
- What happens if it does this while a function is executing, and the handler calls the same function?
  - Standard example: printf, stdio
  - In the middle of a printf, your signal handler runs and calls printf
  - This is called *reentrancy*
  - Code is *reentrant* if it will execute correctly when re-entered mid-execution
  - printf() is not reentrant



Reentrant code

- Recall that the kernel has user processes execute signal handlers by pushing stack frames
- What happens if it does this while a function is executing, and the handler calls the same function?
  - Standard example: printf, stdio
  - In the middle of a printf, your signal handler runs and calls printf
  - This is called *reentrancy*
  - Code is *reentrant* if it will execute correctly when re-entered mid-execution
  - printf() is not reentrant (remember those weird double-prints...)





Async-signal-safe

- POSIX defines which functions are *async-signal-safe*, that is, asynchronous signals can call safely
  - These functions are reentrant
  - All of your system call friends so far: read, write, signal, open, dup2, pipe, execve
  - Lots of string functions: strcmp, strcpy, etc.
  - `$ man signal-safety`
- The basic issue is static buffers: recall what happened to our buffer when the SIGALRM handler cleared it

The root of the problem

- Signals are the first appearance of concurrent/reentrant code in UNIX
- It turns out that correctly handling concurrency and reentrancy in a clean way that's not hair-pullingly difficult requires a bit of support and atomicity in APIs (e.g., sigsuspend)
  - Using the simple signal APIs is rife with problems: it seems to work, but then fails in ways you did not expect or anticipate
- We now understand concurrency and reentrancy much better, and subsequent APIs (e.g., pthreads, which we'll start covering on Wednesday) are much cleaner
  - You can't safely call printf from a signal handler (it's not async-signal-safe), but you can call it in a multithreaded program (it is thread-safe)