Lecture 08: Race Conditions, Deadlock, and Data Integrity

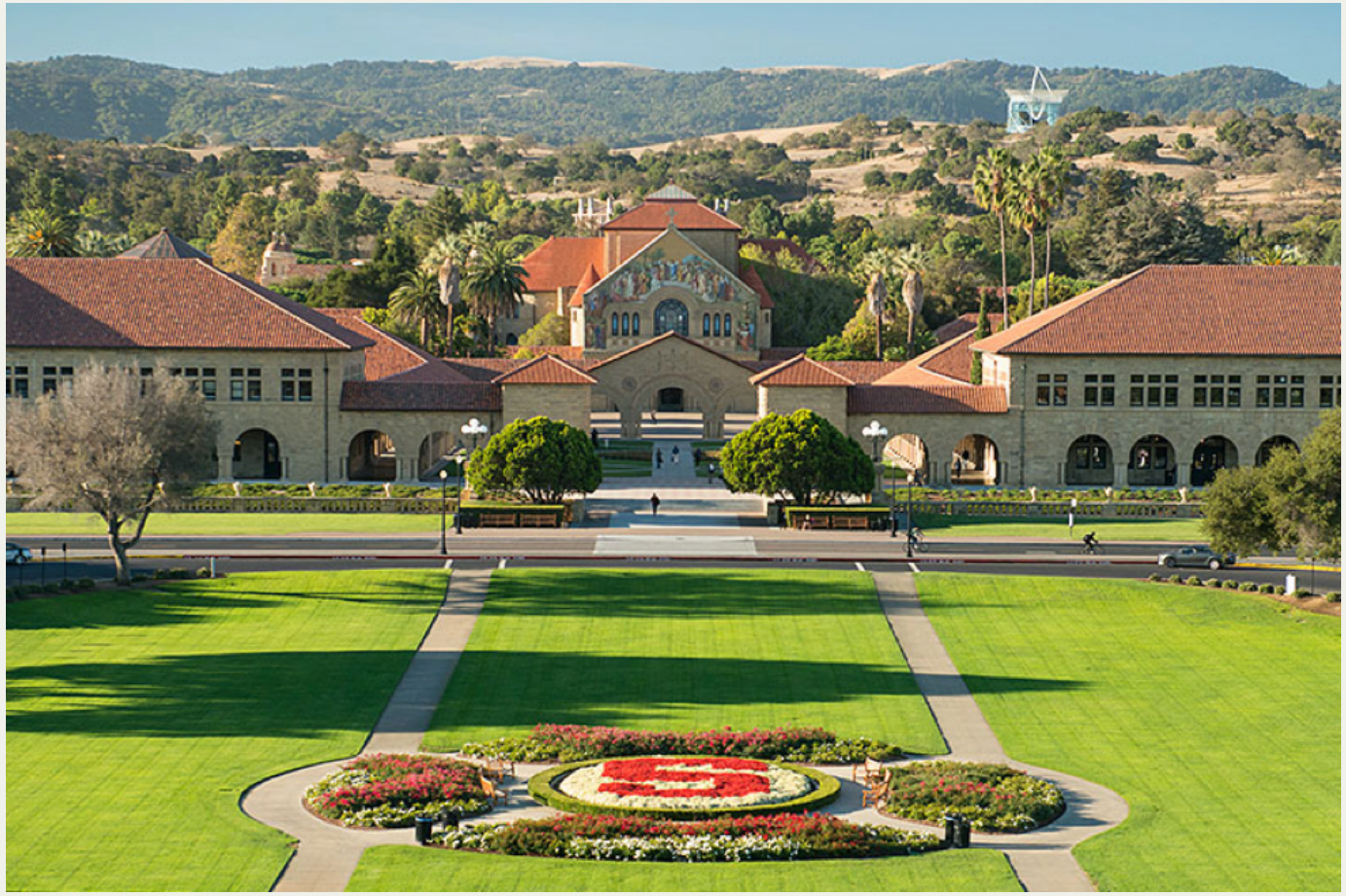
Principles of Computer Systems

Spring 2019

Stanford University

Computer Science Department

Lecturer: Chris Gregg



PDF of this presentation

Lecture 07 (review): Masking Signals and Deferring Handlers

- Comment on the end of last Wednesday's lecture:

```
// job-list-fixed.c
static void reapProcesses(int sig) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, WNOHANG);
    if (pid <= 0) break;
    printf("Job %d removed from job list.\n", pid);
  }
}

char * const kArguments[] = {"date", NULL};
int main(int argc, char *argv[]) {
  signal(SIGCHLD, reapProcesses);
  sigset_t set;
  sigemptyset(&set);
  sigaddset(&set, SIGCHLD);
  for (size_t i = 0; i < 3; i++) {
    sigprocmask(SIG_BLOCK, &set, NULL);
    pid_t pid = fork();
    if (pid == 0) {
      sigprocmask(SIG_UNBLOCK, &set, NULL);
      execvp(kArguments[0], kArguments);
    }
    sleep(1); // force parent off CPU
    printf("Job %d added to job list.\n", pid);
    sigprocmask(SIG_UNBLOCK, &set, NULL);
  }
  return 0;
}
```

- In discussing the job-list-fixed example (here), we discussed whether the *child's* signal handler could get called if the program that the child launched with **`execvp`** had a child of its own, and that child ended.
- I mistakenly discussed what might need to be done to avoid this, but it turns out that *nothing* needs to be done!
- Once the original child starts another program with **`execvp`** all of the original code is gone. Therefore, the signal handler cannot be called, because it doesn't exist any longer.
- This is distinct from the idea that *blocked* signals are still blocked across an **`execvp`** boundary.

Lecture 07 (review): Masking Signals and Deferring Handlers

- Signal extras: **`kill`** and **`raise`**
  - Processes can message other processes using signals via the **`kill`** system call. And processes can even send themselves signals using **`raise`**.

```
int kill(pid_t pid, int signum);
int raise(int signum); // equivalent to kill(getpid(), signum);
```

  - The **`kill`** system call is analogous to the **`/bin/kill`** shell command.
    - Unfortunately named, since **`kill`** implies **`SIGKILL`** implies death.
    - So named, because the default action of most signals in early UNIX implementations was to just terminate the target process.
  - We generally ignore the return value of **`kill`** and **`raise`**. Just make sure you call it properly.
  - The **`pid`** parameter is overloaded to provide more flexible signaling.
    - When **`pid`** is a positive number, the target is the process with that pid.
    - When **`pid`** is a negative number less than -1, the targets are all processes within the process group **`abs(pid)`**. We'll rely on this in Assignment 4.
    - **`pid`** can also be 0 or -1, but we don't need to worry about those. See the man page for **`kill`** if you're curious.

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- The `job-list-broken` and `job-list-fixed` examples from the prior slide deck highlight a key issue that comes with the introduction of signals and signal handling.
  - Neither `job-list-broken` nor `job-list-fixed` can anticipate when a child process will finish up. That means it has no control over when `SIGCHLD` signals arrive.
  - Processes do, however, have some control over how they respond to `SIGCHLD` signals.
    - They install custom `SIGCHLD` handlers to surface information about what process exited. We've seen a lot of that already.
    - When a process **elects** to use signal handling, it shouldn't be penalized by having to live with the concurrency issue that come with it. That would only encourage programmers to avoid signals and signal handling, even when it's the best thing to do.
    - That's why the kernel provides the option to defer a signal handler to run only when it can't cause problems. That's what our `job-list-fixed` program does.
    - It's true that the program could abuse the power to block signals for longer than necessary, but we have no choice but to assume the program wants to use signal handlers properly, else they wouldn't be installing them in the first place.

Lecture 08: Race Conditions, Deadlock, and Data Integrity

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

```
1  // simplesh.c
2  int main(int argc, char *argv[]) {
3    while (true) {
4      // code to initialize command, argv, and isbg omitted for brevity
5      pid_t pid = fork();
6      if (pid == 0) execvp(argv[0], argv);
7      if (isbg) {
8        printf("%d %s\n", pid, command);
9      } else {
10       waitpid(pid, NULL, 0);
11     }
12   }
13   printf("\n");
14   return 0;
15 }
```

- 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.

```
1  // simplesh-with-redundancy.c
2  static void reapProcesses(int sig) {
3    while (waitpid(-1, NULL, WNOHANG) > 0) {;} // nonblocking, iterate until retval is -1 or 0
4  }
5
6  int main(int argc, char *argv[]) {
7    signal(SIGCHLD, reapProcesses);
8    while (true) {
9      // code to initialize command, argv, and isbg omitted for brevity
10     pid_t pid = fork();
11     if (pid == 0) {
12       execvp(argv[0], argv);
13       printf("%s: Command not found\n", argv[0]);
14       exit(0);
15     }
16     if (isbg) {
17       printf("%d %s\n", pid, command);
18     } else {
19       waitpid(pid, NULL, 0);
20     }
21   }
22   printf("\n");
23   return 0;
24 }
```

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- The last version actually works, but it relies on a sketchy call to **`waitpid`** to halt the shell until its foreground process has exited.
  - When the user creates a foreground process, normal execution flow advances to an isolated **`waitpid`** call to block until that process has terminated.
  - When the foreground process finishes, however, the **`SIGCHLD`** handler is invoked, and its **`waitpid`** call is the one that culls the foreground process's resources.
  - When the **`SIGCHLD`** handler exits, normal execution resumes, and the original call to **`waitpid`** returns -1 to state that there is no trace of a process with the supplied **`pid`**.
  - The version on the last slide deck is effectively calling **`waitpid`** from **`main`** just to block until the foreground process vanishes.
  - Even if you're content with this unorthodox use of **`waitpid`**—i.e. invoking a system call when you know it will fail—the **`waitpid`** call is redundant and replicates functionality better managed in the **`SIGCHLD`** handler.
    - We should only be calling **`waitpid`** in one place: the **`SIGCHLD`** handler.
    - This will be all the more apparent when we implement shells (e.g. Assignment 4's **`stsh`**) where multiple processes are running in the foreground as part of a pipeline (e.g. **`more words.txt | tee copy.txt | sort | uniq`**)

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- Here's an updated version that's careful to call **`waitpid`** from only one place.

```
1  // simplesh-with-race-and-spin.c
2  static pid_t fgpid = 0; // global, intially 0, and 0 means no foreground process
3  static void reapProcesses(int sig) {
4    while (true) {
5      pid_t pid = waitpid(-1, NULL, WNOHANG);
6      if (pid <= 0) break;
7      if (pid == fgpid) fgpid = 0; // clear foreground process
8    }
9  }
10
11 static void waitForForegroundProcess(pid_t pid) {
12   fgpid = pid;
13   while (fgpid == pid) {;}
14 }
15
16 int main(int argc, char *argv[]) {
17   signal(SIGCHLD, reapProcesses);
18   while (true) {
19     // code to initialize command, argv, and isbg omitted for brevity
20     pid_t pid = fork();
21     if (pid == 0) execvp(argv[0], argv);
22     if (isbg) {
23       printf("%d %s\n", pid, command);
24     } else {
25       waitForForegroundProcess(pid);
26     }
27   }
28   printf("\n");
29   return 0;
30 }
```

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- The version on the last page introduces a global variable called **`fgpid`** to hold the process is of the foreground process. When there's no foreground process, **`fgpid`** is 0.
  - Because we don't control the signature of **`reapProcesses`**, we have to choice but to make **`fgpid`** a global.
  - Every time a new foreground process is created, **`fgpid`** is set to hold that process's pid. The shell then blocks by *spinning* in place until **`fgpid`** is cleared by **`reapProcesses`**.
- This version consolidates the **`waitpid`** code to reside in the handler and nowhere else.
- This version introduces two serious problems, so it's far from an A+ solution.
  - 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, and race conditions are no-nos.
  - The **`while (fgpid == pid) {;}`** is also a no-no. This allows the shell to spin on the CPU even when it can't do any meaningful work.
    - It would be substantially better for **`simplesh`** to yield the CPU and to only be considered for CPU time when there's a chance the foreground process has exited.

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- The race condition can be cured by blocking **`SIGCHLD`** before forking, and only lifting that block after the global **`fgpid`** has been set.
  - Here's a version of the code that employs signal blocking to remove this race condition.

```
1  // simplesh-with-spin.c
2  // code for reapProcesses omitted, because it's the same as before
3
4  static void waitForForegroundProcess(pid_t pid) {
5    fgpid = pid;
6    unblockSIGCHLD(); // lift only after fgpid has been set
7    while (fgpid == pid) {;}
8  }
9
10 int main(int argc, char *argv[]) {
11   signal(SIGCHLD, reapProcesses);
12   while (true) {
13     // code to initialize command, argv, and isbg omitted for brevity
14     blockSIGCHLD();
15     pid_t pid = fork();
16     if (pid == 0) {
17       unblockSIGCHLD();
18       execvp(argv[0], argv);
19     }
20     if (isbg) {
21       printf("%d %s\n", pid, command);
22       unblockSIGCHLD();
23     } else {
24       waitForForegroundProcess(pid);
25     }
26   }
27 }
```

```
1  // simples-utils.c
2  // includes a collection of helper functions
3
4  static void toggleSIGCHLDBlock(int how) {
5    sigset_t mask;
6    sigemptyset(&mask);
7    sigaddset(&mask, SIGCHLD);
8    sigprocmask(how, &mask, NULL);
9  }
10
11 void blockSIGCHLD() {
12   toggleSIGCHLDBlock(SIG_BLOCK);
13 }
14
15 void unblockSIGCHLD() {
16   toggleSIGCHLDBlock(SIG_UNBLOCK);
17 }
```

Note that we call **`unblockSIGCHLD`** *in the child, before the* **`execvp`** *call. We do so, because the child will otherwise inherit the signal block.*

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- Race condition is now gone!
  - Note that we call **`blockSIGCHLD`** before **`fork`**, and we don't lift the block until **`fgpid`** has been set to the **`pid`** of the new foreground process.
  - We also call **`unblockSIGCHLD`** in the child right before the **`execvp`** call.
    - The child executable could very well depend on multiprocessing. If so, it would certainly call **`fork`** and rely on **`SIGCHLD`** signals and signal handling.
    - If we forget to call **`unblockSIGCHLD`**, the child process inherits the **`SIGCHLD`** block across the **`execvp`** boundary. That would compromise the child ability to work properly.
  - We also need to call **`unblockSIGCHLD`** for background processes. We do so after bookkeeping information is **`printf`**-ed to the screen, as we did for **`job-list-fixed`**.
  - We have not addressed the CPU spin issue, and we really need to.
    - We could change the while loop from **`while (fgpid == pid) {;}`** to **`while (fgpid == pid) {usleep(100000);}`**, as we have in this version.
    - **`usleep`** call will push the shell off the CPU every time it realizes it shouldn't have gotten it in the first place. But we'd really prefer to keep the shell off the CPU until the OS has some information suggesting the foreground process is done.

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- The C libraries provide a **`pause`** function, which forces the process to sleep until some unblocked signal arrives. This sounds promising, because we know **`fgpid`** can only be changed because a **`SIGCHLD`** signal comes in and **`reapProcesses`** is executed.
  - A version of **`simplesh`** whose **`waitForForegroundProcess`** implementation relies on **`pause`** is presented below on the left.
  - The problem here? **`SIGCHLD`** may arrive after **`fgpid == pid`** evaluates to **`true`** but before the call to **`pause`** it's committed to. That would be unfortunate, because it's possible **`simplesh`** isn't managing any other processes, which means that no other signals, much less **`SIGCHLD`**signals, will arrive to lift **`simplesh`** out of its **`pause`** call. That would leave **`simplesh`** in a state of **deadlock**.
  - You might think the second (lower right) version might help, but it has the same problem!

```
1 // simplesh-with-pause-1.c
2 static void waitForForegroundProcess(pid_t pid) {
3   fgpid = pid;
4   unblockSIGCHLD();
5   while (fgpid == pid) {
6     pause();
7   }
8 }
```

```
 1 // simplesh-with-pause-2.c
 2 static void waitForForegroundProcess(pid_t pid) {
 3   fgpid = pid;
 4   while (fgpid == pid) {
 5     unblockSIGCHLD();
 6     pause();
 7     blockSIGCHLD();
 8   }
 9   unblockSIGCHLD();
10 }
```

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- 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.

```
1  // simplesh-all-better.c
2  static void waitForForegroundProcess(pid_t pid) {
3    fgpid = pid;
4    sigset_t empty;
5    sigemptyset(&empty);
6    while (fgpid == pid) {
7      sigsuspend(&empty);
8    }
9    unblockSIGCHLD();
10 }
```

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

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- Let's go through an example that is the kind of signals problem you may see on the midterm exam.
- Indeed, the problem is from a past midterm in CS 110:
  - 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.

```
1  static void bat(int unused) {
2    printf("pirate\n");
3    exit(0);
4  }
5
6  int main(int argc, char *argv[]) {
7    signal(SIGUSR1, bat);
8    pid_t pid = fork();
9    if (pid == 0) {
10     printf("ghost\n");
11     return 0;
12   }
13   kill(pid, SIGUSR1);
14   printf("ninja\n"); return 0;
15 }
```

- 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**

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- Let's go through an example that is the kind of signals problem you may see on the midterm exam.
- Indeed, the problem is from a past midterm in CS 110:
  - 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.

```
1  static void bat(int unused) {
2    printf("pirate\n");
3    exit(0);
4  }
5
6  int main(int argc, char *argv[]) {
7    signal(SIGUSR1, bat);
8    pid_t pid = fork();
9    if (pid == 0) {
10     printf("ghost\n");
11     return 0;
12   }
13   kill(pid, SIGUSR1);
14   printf("ninja\n"); return 0;
15 }
```

- 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**

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- Let's go through another example that is the kind of signals problem you may see on the midterm exam.
  - 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.

```
1  int main(int argc, char *argv[]) {
2      pid_t pid;
3      int counter = 0;
4      while (counter < 2) {
5          pid = fork();
6          if (pid > 0) break;
7          counter++;
8          printf("%d", counter);
9      }
10     if (counter > 0) printf("%d", counter);
11     if (pid > 0) {
12         waitpid(pid, NULL, 0);
13         counter += 5;
14         printf("%d", counter);
15     }
16     return 0;
17 }
```

- List all possible outputs

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- Let's go through another example that is the kind of signals problem you may see on the midterm exam.
  - 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.

```
1  int main(int argc, char *argv[]) {
2      pid_t pid;
3      int counter = 0;
4      while (counter < 2) {
5          pid = fork();
6          if (pid > 0) break;
7          counter++;
8          printf("%d", counter);
9      }
10     if (counter > 0) printf("%d", counter);
11     if (pid > 0) {
12         waitpid(pid, NULL, 0);
13         counter += 5;
14         printf("%d", counter);
15     }
16     return 0;
17 }
```

- 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.)

Lecture 08: Race Conditions, Deadlock, and Data Integrity

- Let's go through another example that is the kind of signals problem you may see on the midterm exam.
  - 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.

```
1  int main(int argc, char *argv[]) {
2      pid_t pid;
3      int counter = 0;
4      while (counter < 2) {
5          pid = fork();
6          if (pid > 0) break;
7          counter++;
8          printf("%d", counter);
9      }
10     if (counter > 0) printf("%d", counter);
11     if (pid > 0) {
12         waitpid(pid, NULL, 0);
13         counter += 5;
14         printf("%d", counter);
15     }
16     return 0;
17 }
```

- 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)