CS110 Lecture 06: Pipes, Signals, and Concurrency

Principles of Computer Systems

Autumn 2019

Stanford University

Computer Science Department

Lecturers: Chris Gregg and
Philip Levis

PDF of this presentation

Revisiting mysystem: fork, execvp, and waitpid

- Here's the implementation, with minimal error checking (the full version is right here, and a working version is on the next slide):

```c
static int mysystem(const char *command) {
  pid_t pid = fork();
  if (pid == 0) {
    char *arguments[] = {"/bin/sh", "-c", (char *) command, NULL};
    execvp(arguments[0], arguments);
    printf("Failed to invoke /bin/sh to execute the supplied command.");
    exit(0);
  }
  int status;
  waitpid(pid, &status, 0);
  return WIFEXITED(status) ? WEXITSTATUS(status) : -WTERMSIG(status);
}
```

- Instead of calling a subroutine to perform some task and waiting for it to complete, **`mysystem`** spawns a **child process** to perform some task and waits for it to complete.
- We don't bother checking the return value of **`execvp`**, because we know that if it returns at all, it returns a -1. If that happens, we need to handle the error and make sure the child process terminates, via an exposed **`exit(0)`** call.
- Why not call **`execvp`** inside parent and forgo the child process altogether? Because **`execvp`** would consume the calling process, and that's not what we want.

Next step: Assignment 4 (stsh)

- The `mysystem` function is the first example of `fork`, `execvp`, and `waitpid` all work together to do something genuinely useful.
  - The test harness we used to exercise `mysystem` is operationally a miniature terminal.
  - We need to continue implementing a few additional mini-terminals to fully demonstrate how `fork`, `waitpid`, and `execvp` work in practice.
  - All of this is paying it forward to your fourth assignment, where you'll implement your own shell—we call it `stsh` for Stanford shell—to imitate the functionality of the shell (`csh`, `bash`, `zsh`, `tcsh`, etc.) you've been using since you started using Unix.
- We'll introduce the notion of a pipe, the `pipe` system call, and how it creates a communication channels between two processes.
- We'll introduce `dup2` system call, and how it allows a process to manipulate its file descriptor table.

Shells do much more

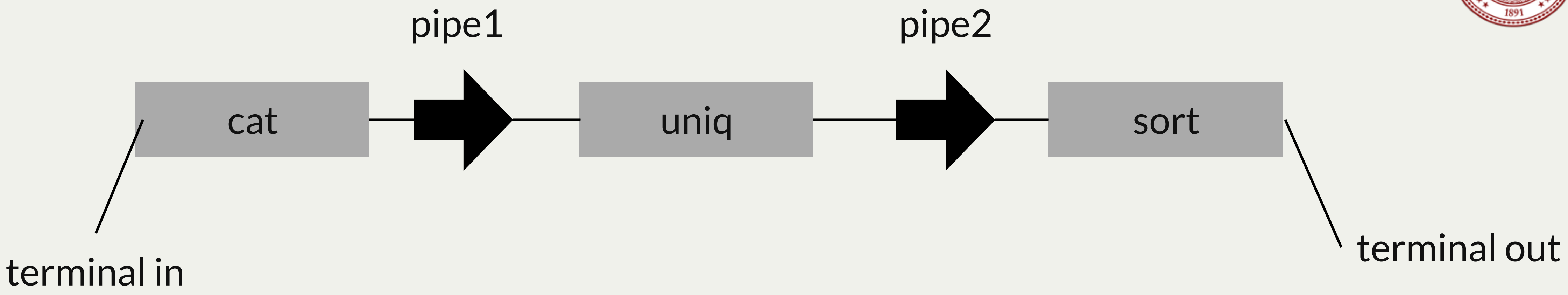
- **`mysystem`** is just a simple read-eval loop: it relies on a real shell (sh) to parse arguments and do all of the other things shells do
  - emacs & - create an emacs process and return control to the shell (backgrounding)
  - cat file.txt | uniq | sort - pipe the output of one command to the input of another
  - uniq < file.txt | sort > list.txt - make file.txt the input of uniq and output sort to list.txt
- Let's walk through the mechanisms of how shells do this so you can implement it for **`stsh`**

pipe(2)

- The **`pipe`** system call takes an uninitialized array of two integers and populates it with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**.
  - Here's the prototype: `int pipe(int fds[]);`
- **`pipe`** is particularly useful for allowing parent processes to communicate with spawned child processes.
  - Because they're file descriptors, there's no global name for the pipe (another process can't "connect" to the pipe)
  - The parent's table is replicated in the child, so the child automatically gets access to the pipe
- Example:cat file.txt | uniq | sort
  - Shell creates three child processes: cat, uniq and sort
  - Shell creates two pipes: one between cat and uniq, one between uniq and sort

pipe(2) example

- Example:cat file.txt | uniq | sort
  - Shell creates three child processes: cat, uniq and sort
  - Shell creates two pipes: one between cat and uniq, one between uniq and sort




```
int pipe1[2];
int pipe2[2];
pipe(pipe1);
pipe(pipe2);
```

Process	**stdin**	**stdout**
cat	terminal	pipe1[1]
uniq	pipe1[0]	pipe2[1]
sort	pipe2[0]	terminal

Using pipe(2)

- How does **`pipe`** work?
  - To illustrate how **`pipe`** works and how arbitrary data can be passed over from one process to a second, let's consider the following program (which you can find here, or run on the next slide):

```
int main(int argc, char *argv[]) {
  int fds[2];
  pipe(fds);
  pid_t pid = fork();
  if (pid == 0) {
    close(fds[1]);
    char buffer[6];
    read(fds[0], buffer, sizeof(buffer));
    printf("Read from pipe bridging processes: %s.\n", buffer);
    close(fds[0]);
    return 0;
  }
  close(fds[0]);
  write(fds[1], "hello", 6);
  waitpid(pid, NULL, 0);
  close(fds[1]);
  return 0;
}
```

pipe(2) code example

https://cplayground.com/?p=okapi-grasshopper-bear

Pipe file descriptors and file table entries

- How do **`pipe`** and **`fork`** work together in this example?
  - **`pipe`** allocates two descriptors, one for reading and one for writing
  - The **`fork`** call creates a child process, which has a shallow copy of the parent's **`fds`** array.
    - The reference counts in each of the two file table entries of the pipe are incremented from 1 to 2 to reflect the fact that two descriptors—one in the parent, and a second in the child—reference each of them.
    - Immediately after the **`fork`** call, anything printed to **`fds[1]`** is readable from the parent's **`fds[0]`** *and* the child's **`fds[0]`**.
    - Similarly, both the parent and child are capable of publishing text to the same resource via their copies of **`fds[1]`**.

Be clean and careful in your systems code

- The parent closes **`fds[0]`** before it writes to anything to **`fds[1]`** because it will never use it; close it doesn't linger around as long as the parent does
  - Similarly, the child closes **`fds[1]`** before it reads from **`fds[0]`**
- Further benefit: clearly shows someone reading the code that parent never uses **`fds[0]`** and child never uses **`fds[1]`**
- Explicitly clean up resources: close a file descriptor as soon as its use is over
  - Ensures that if the process runs for a long time it doesn't hold references to lots of dead files
  - In this code you could say "hey, I know the process will exit and clean up", but what if you start extending and building out this code, and put this logic in a loop?

Example: forking with fd manipulation

- Using **`pipe`**, **`fork`**, **`dup2`**, **`execvp`**, **`close`**, and **`waitpid`**, we can implement the **`subprocess`** function, which relies on the following record definition and is implemented to the following prototype (full implementation of everything is right here):

```c
typedef struct {
  pid_t pid;
  int supplyfd;
} subprocess_t;
subprocess_t subprocess(const char *command);
```

- The child process created by **`subprocess`** executes the provided **`command`** (assumed to be a **`'\0'`**-terminated C string) by calling **`"/bin/sh -c <command>"`** as we did in our **`mysystem`** implementation.
  - **`subprocess`** returns a **`subprocess_t`** with the process's **`pid`** and a descriptor called **`supplyfd`**.
- The child process reads writes to **`supplyfd`** from its standard input
  - We can pass arbitrary input to the child process

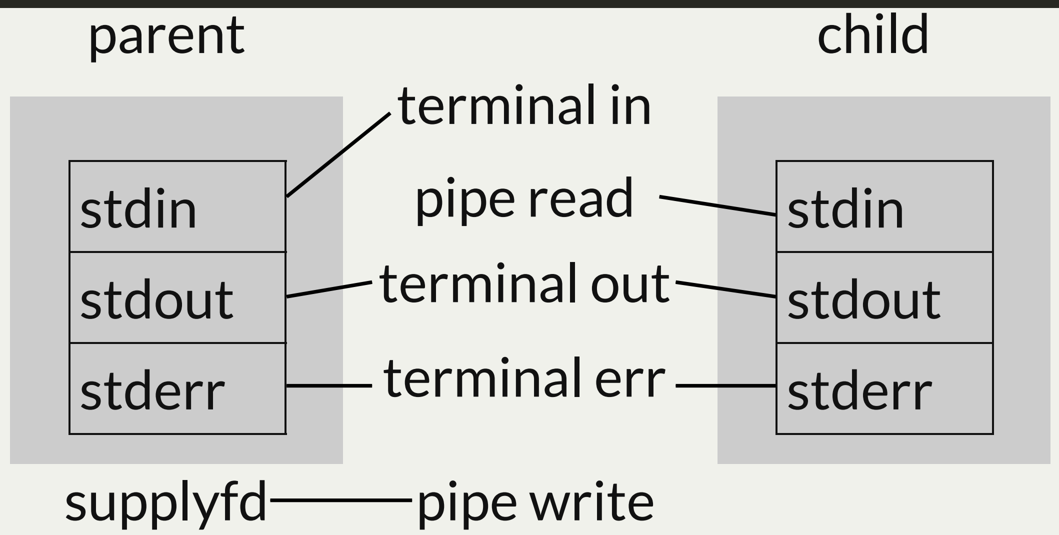
subprocess Test Harness

- This program spawns a child process that reads eight fancy words from **`stdin`**, sorts them, and writes the output to standard out:

```c
int main(int argc, char *argv[]) {
  subprocess_t sp = subprocess("/usr/bin/sort");
  const char *words[] = {
    "felicity", "umbrage", "susurration", "halcyon",
    "pulchritude", "ablution", "somnolent", "indefatigable"
  };
  for (size_t i = 0; i < sizeof(words)/sizeof(words[0]); i++) {
    dprintf(sp.supplyfd, "%s\n", words[i]);
  }
  close(sp.supplyfd);
  int status;
  pid_t pid = waitpid(sp.pid, &status, 0);
  return pid == sp.pid && WIFEXITED(status) ? WEXITSTATUS(status) : -127;
}
```

subprocess Test Harness File Descriptors

```c
int main(int argc, char *argv[]) {
  subprocess_t sp = subprocess("/usr/bin/sort");
  const char *words[] = {
    "felicity", "umbrage", "susurration", "halcyon",
    "pulchritude", "ablution", "somnolent", "indefatigable"
  };
  for (size_t i = 0; i < sizeof(words)/sizeof(words[0]); i++) {
    dprintf(sp.supplyfd, "%s\n", words[i]);
  }
  close(sp.supplyfd);
  int status;
  pid_t pid = waitpid(sp.pid, &status, 0);
  return pid == sp.pid && WIFEXITED(status) ? WEXITSTATUS(status) : -127;
}
```



Subtle requirement of semantics

- We need to produce a file descriptor that allows the caller to write data to the child's stdin
  - We know how to create such a file descriptor: a pipe
  - But how do we set up the read side of the pipe as the child's standard in?
- **`dup2(2)`**: copy a file descriptor into another file descriptor, closing the replaced file descriptor's entry if needed
  - **`dup2(int oldfd, int newfd)`**
  - Use **`dup2`** to copy the pipe read file descriptor into child's standard input

subprocess implementation

- Implementation of **`subprocess`** (error checking intentionally omitted for brevity):

```
subprocess_t subprocess(const char *command) {
  int fds[2];
  pipe(fds);
  subprocess_t process = { fork(), fds[1] };
  if (process.pid == 0) {
    close(fds[1]);
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    char *argv[] = {"/bin/sh", "-c", (char *) command, NULL};
    execvp(argv[0], argv);
  }
  close(fds[0]);
  return process;
}
```

  - The write end of the pipe is embedded into the **`subprocess_t`**. That way, the parent knows where to publish text so it flows to the read end of the pipe, across the parent process/child process boundary. This is bonafide interprocess communication.
  - The child process uses **`dup2`** to bind the read end of the pipe to its own standard input. Once the reassociation is complete, **`fds[0]`** can be closed.

Questions about pipes?

UNIX Signals

- A **signal** is a way to notify a process that an event occurred.
  - The kernel sends many signals (SIGSEGV, SIGBUS, SIGINT, ...)
    - Everyone who's programmed in C has unintentionally dereferenced a `NULL` pointer.
    - The kernel delivers a `SIGSEGV`, informally known as a segmentation fault (or a **SEG**mentation **V**iolation, or `SIGSEGV`, for short).
    - Unless you install a custom signal handler to manage the signal differently, a `SIGSEGV` terminates the program and generates a core dump.
  - Processes can send each other signals as well (SIGSTOP, SIGKILL)
- A **signal handler** is a function that executes when the signal arrives
  - Some signals have default handler(e.g., SIGSEGV terminates process and dumps core)
  - You can install custom handlers for most signals
- Each signal is represented internally by some number (e.g. `SIGSEGV` is 11).

Some Signals

- **`SIGFPE`**: whenever a process commits an integer-divide-by-zero (and, in some cases, a floating-point divide by zero on older architectures), the kernel hollers and issues a **`SIGFPE`** signal to the offending process. By default, the program handles the **`SIGFPE`** by printing an error message announcing the zero denominator and generating a core dump.
- **`SIGINT`**: when you type ctrl-c, the kernel sends a **`SIGINT`** to the foreground process group. The default handler terminates the process group.
- **`SIGTSTP`**: when you type ctrl-z, the kernel issues a **`SIGTSTP`** to the foreground process group. The foreground process group is halted until a **`SIGCONT`** signal.
- **`SIGPIPE`**: when a process attempts to write data to a pipe after the read end has closed, the kernel delivers a **`SIGPIPE`**. The default **`SIGPIPE`** handler prints a message identifying the pipe error and terminates the program.

A Systems Mystery

```
$ grep error file.txt > errors.txt &
[1] 4287
$
[1]+ Done                 grep error file.txt > errors.txt
```

- How does this work?
  - The shell returns control to the user after forking the child (not calling waitpid on the child)
  - But the shell still knows when the child completes
- There must be a way for the shell to learn about when things have happened to its children

SIGCHLD

- Whenever a child process **changes state**—that is, it exits, crashes, stops, or resumes from a stopped state, the kernel sends a `SIGCHLD` signal to the process's parent.
  - By default, the signal is ignored. We've ignored it until right now and gotten away with it.
- This particular signal type is instrumental to allowing forked child processes to run in the background while keeping the parent immediately aware of when something happens.
- Custom `SIGCHLD` handlers can call `waitpid`, which tells them the pids of child processes that gave changed state. If the child process terminated, either normally or abnormally, the `waitpid` also cleans up/frees the child.

Signals at Disneyland

- Here's an example of when you might want to use a **`SIGCHLD`** handler.
- The premise? Dad takes his five kids out to play. Each of the five children plays for a different length of time. When all five kids are done playing, the six of them all go home.
  - If Dad has stuff to do (rather than nap), this is a very simple analogy to many parallel data processing applications (if Dad only naps just call **`wait`**)
- The parent is dad, and subprocesses are children. (Full program is right here.)

```
static const size_t kNumChildren = 5;
static size_t numDone = 0;

int main(int argc, char *argv[]) {
  printf("Let my five children play while I take a nap.\n");
  signal(SIGCHLD, reapChild);
  for (size_t kid = 1; kid <= 5; kid++) {
    if (fork() == 0) {
      sleep(3 * kid); // sleep emulates "play" time
      printf("Child #%zu tired... returns to dad.\n", kid);
      return 0;
    }
  }
```

Signals at Disneyland

- Our first signal handler example: Disneyland
  - The program is crafted so each child process exits at three-second intervals. **`reapChild`**, handles each of the **`SIGCHLD`** signals delivered as each child process exits.
  - The **`signal`** prototype doesn't allow for state to be shared via parameters, so we have no choice but to use global variables.

```c
  // code below is a continuation of that presented on the previous slide
  while (numDone < kNumChildren) {
    printf("At least one child still playing, so dad nods off.\n");
    sleep(5);
    printf("Dad wakes up! ");
  }
  printf("All children accounted for.  Good job, dad!\n");
  return 0;
}

static void reapChild(int unused) {
  waitpid(-1, NULL, 0);
  numDone++;
}
```

Signals at Disneyland

- Here's the output of the above program.
  - Dad's wakeup times (at t = 5 sec, t = 10 sec, etc.) interleave the various finish times (3 sec, 6 sec, etc.) of the children, and the output published below reflects that.
  - Understand that the **`SIGCHLD`** handler is invoked 5 times, each in response to some child process finishing up.

```
cgregg@myth60$ ./five-children
Let my five children play while I take a nap.
At least one child still playing, so dad nods off.
Child #1 tired... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Child #2 tired... returns to dad.
Child #3 tired... returns to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Child #4 tired... returns to dad.
Child #5 tired... returns to dad.
Dad wakes up! All children accounted for.  Good job, dad!
cgregg@myth60$
```

Signal Handling Semantics

- A signal is **not** like a function call
  - Signals aren't handled immediately (there can be delays)
  - If a signal is delivered multiple times, the handler is only called once
  - There's a bitmask in the kernel
    - Delivering a signal sets the bit
    - Handling the signal clears the bit
    - If multiple instances of the signal are delivered before handling, handler executes **once**
- Signals execute asynchronously: the kernel can push a stack frame onto the process stack that causes it to execute a handler, then return back to what it was doing
  - This is very creepy
  - This makes signals sort-of-concurrent (technically, preemptive)
  - Keep your signal handlers simple or you will regret it
- This is much like how hardware behaves with interrupts -- POSIX brings that model to software

Example of Tricky Signal Semantics

- Consider the scenario where the five kids run about Disneyland for the same amount of time. Restated, **`sleep(3 * kid)`** is now **`sleep(3)`** so all five children flashmob dad when they're all done.
  - Dad never detects all five kids are present and accounted for, and the program runs forever because dad keeps going back to sleep. Why?

```
cgregg*@myth60$ ./broken-pentuplets
Let my five children play while I take a nap.
At least one child still playing, so dad nods off.
Kid #1 done playing... runs back to dad.
Kid #2 done playing... runs back to dad.
Kid #3 done playing... runs back to dad.
Kid #4 done playing... runs back to dad.
Kid #5 done playing... runs back to dad.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! At least one child still playing, so dad nods off.
Dad wakes up! At least one child still playing, so dad nods off.
^C # I needed to hit ctrl-c to kill the program that loops forever!
cgregg@myth60$
```

Waiting without blocking

- Calling **`waitpid`** repeatedly fixes the problem, but it changes the behavior of the program.
  - Calls to **`waitpid`** can prevent dad from returning to his nap. For real programs, this means they can't continue to do work (e.g., respond to shell commands.
- We need to instruct **`waitpid`** to only reap children that have exited but to return without blocking, even if there are more children still running. We use **`WNOHANG`** for this, as with:

```
static void reapChild(int unused) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, WNOHANG);
    if (pid <= 0) break; // note the < is now a <=
    numDone++;
  }
}
```

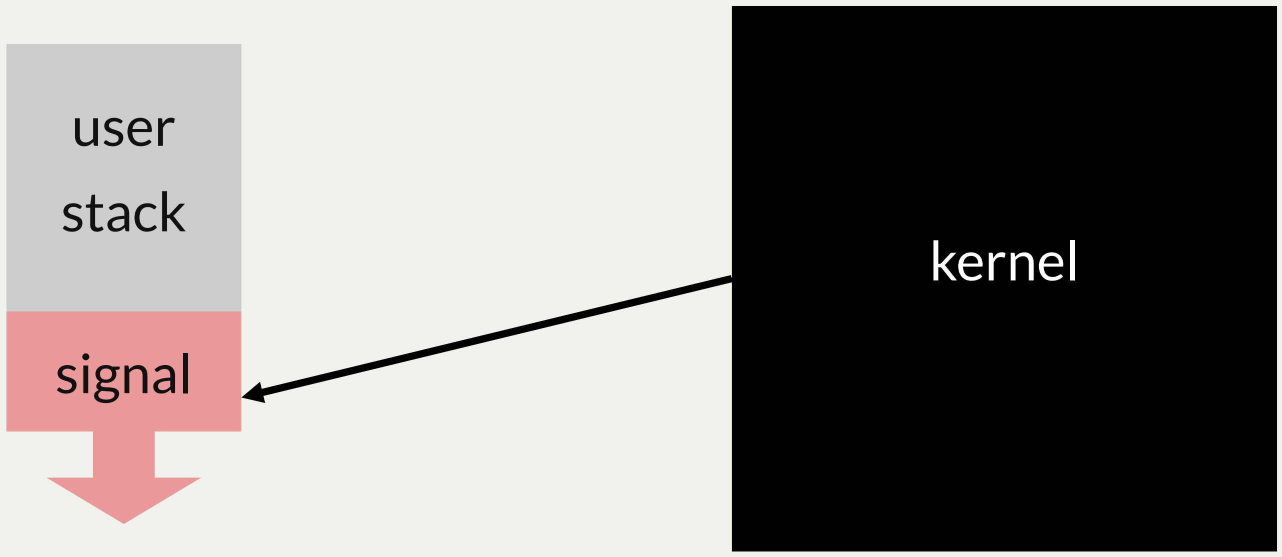
- Why not just call **`waitpid`** with **`WNOHANG`** in the main loop?
  - Mostly a style question: keeps main loop logic simpler.
  - Also means **`waitpid`** is called promptly, not determined by main loop.
  - Also, it means we learn about more than just termination (stopped, resumed, etc.).

How the Kernel Calls Userspace Handlers

- The kernel pushes a stack frame onto the process, so that the process calls the handler then returns to the kernel
- It does this when:
  - The process is running userspace code (i.e., your code)
  - The process returns from a system call
  - Can also interrupt some-long running system calls (e.g., read, which may never return)

How the Kernel Calls Userspace Handlers

- The kernel pushes a stack frame onto the process, so that the process calls the handler then returns to the kernel
- It does this when:
  - The process is running userspace code (i.e., your code)
  - The process returns from a system call
  - Can also interrupt some-long running system calls (e.g., read, which may never return)

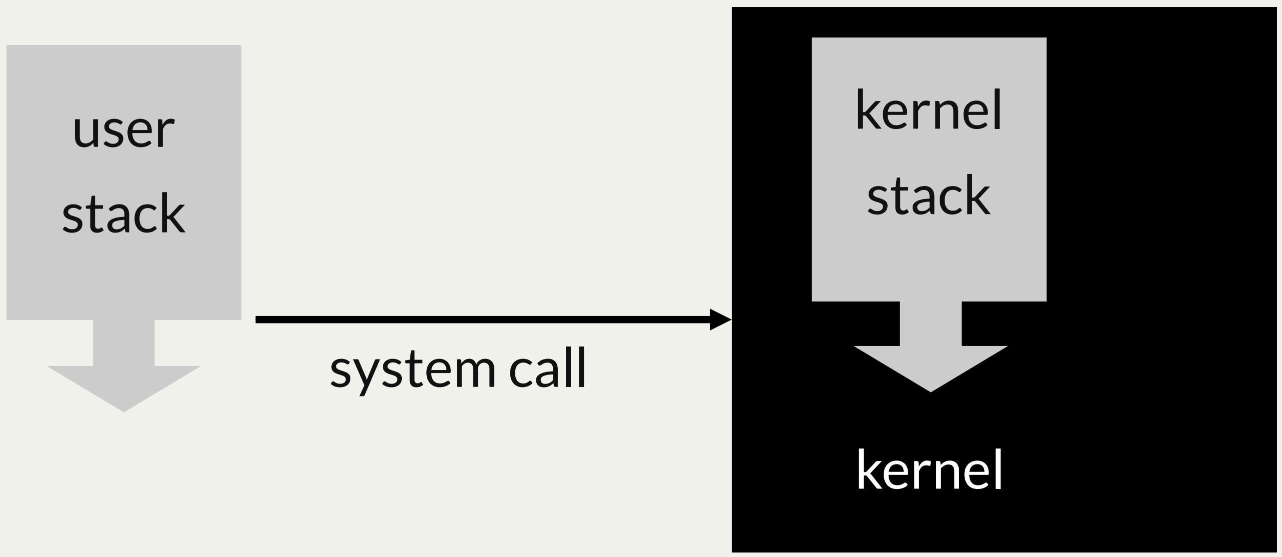


Two Stacks

- Every process actually has two stacks: a user stack and a kernel stack
- When a system call traps to the kernel, the kernel executes on its own stack
  - User program can't see or change kernel stack, otherwise it could control kernel, all security and isolation is thrown out the window
- When a system call traps to the kernel, the kernel saves the state of the user process on its own stack, so it can properly restore the user process

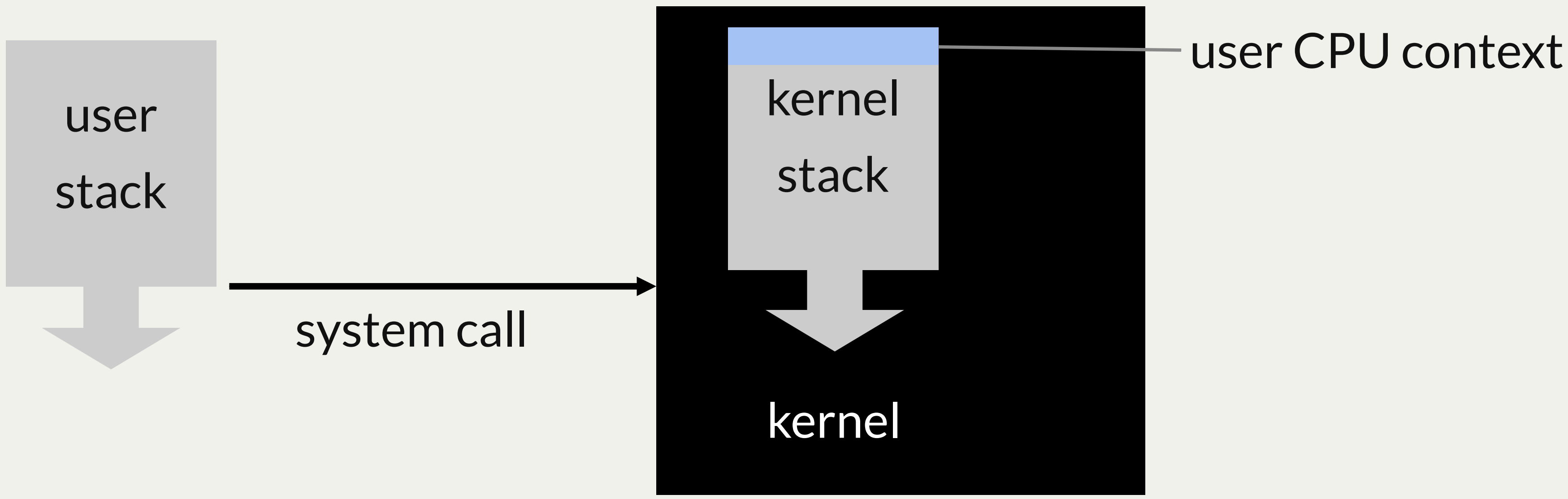
Two Stacks

- Every process actually has two stacks: a user stack and a kernel stack
- When a system call traps to the kernel, the kernel executes on its own stack
    - User program can't see or change kernel stack, otherwise it could control kernel, all security and isolation is thrown out the window
- When a system call traps to the kernel, the kernel saves the state of the user process on its own stack, so it can properly restore the user process



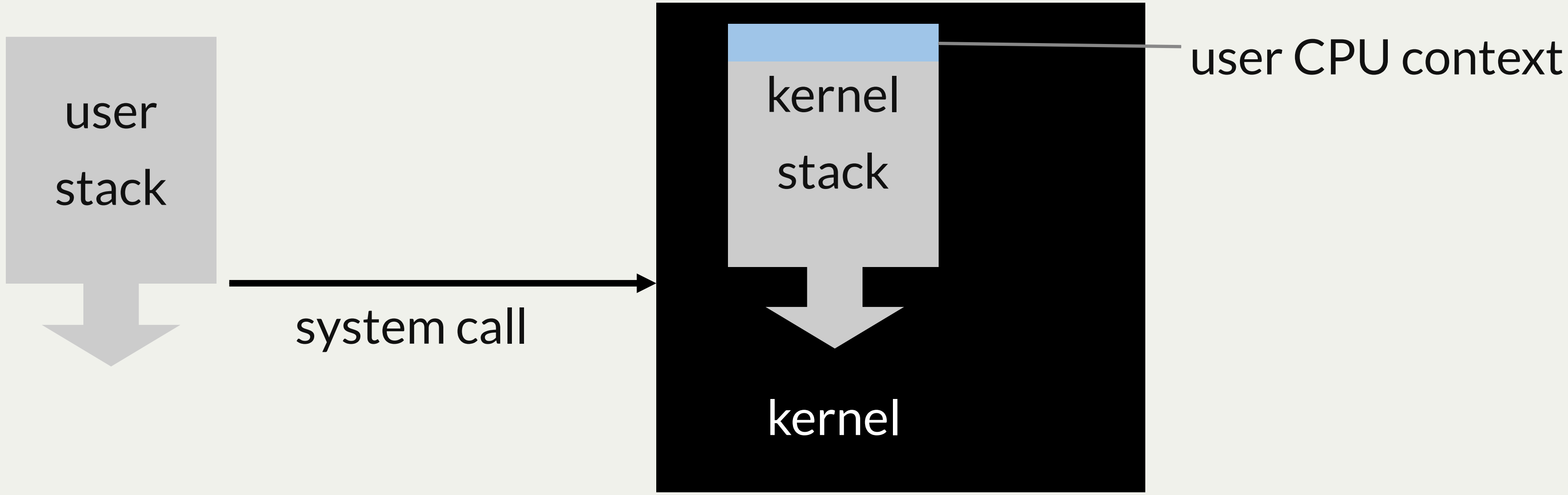
Two Stacks

- Every process actually has two stacks: a user stack and a kernel stack
- When a system call traps to the kernel, the kernel executes on its own stack
  - User program can't see or change kernel stack, otherwise it could control kernel, all security and isolation is thrown out the window
- When a system call traps to the kernel, the kernel saves the state of the user process on its own stack, so it can properly restore the user process
  - E.g., of the user registers, so they can be fully restored



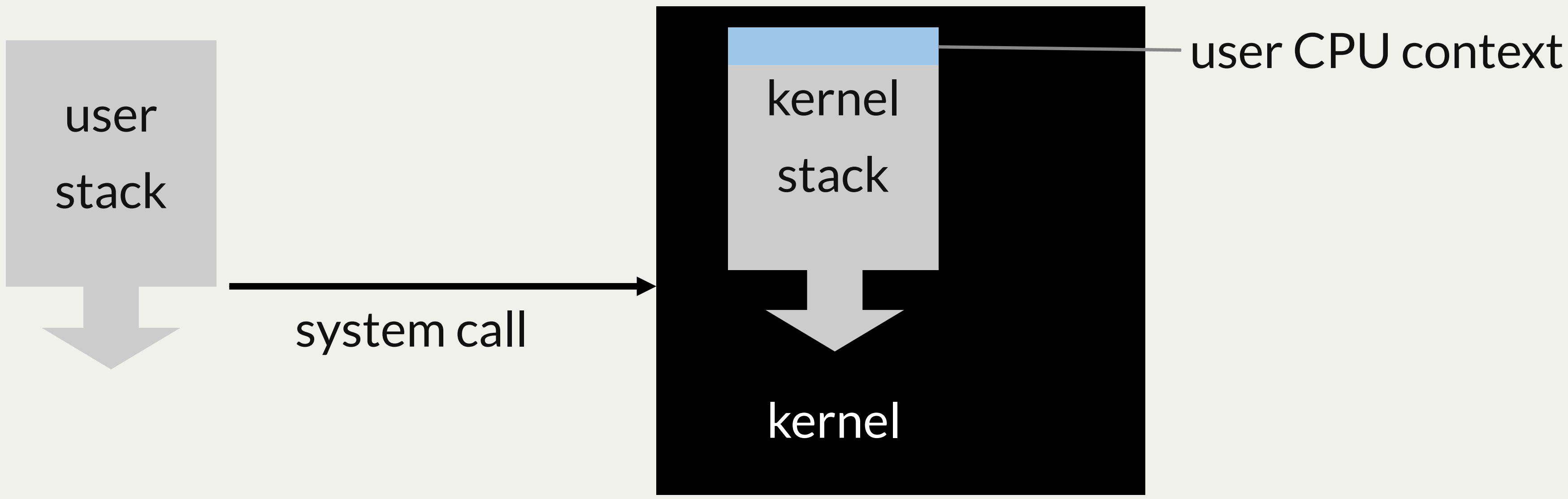
Two Stacks

- Every process actually has two stacks: a user stack and a kernel stack
- When a system call traps to the kernel, the kernel executes on its own stack
  - User program can't see or change kernel stack, otherwise it could control kernel, all security and isolation is thrown out the window
- When a system call traps to the kernel, the kernel saves the state of the user process on its own stack, so it can properly restore the user process
- With a signal, the kernel needs to complete and return to the user process, but this user process state is still needed, after the signal



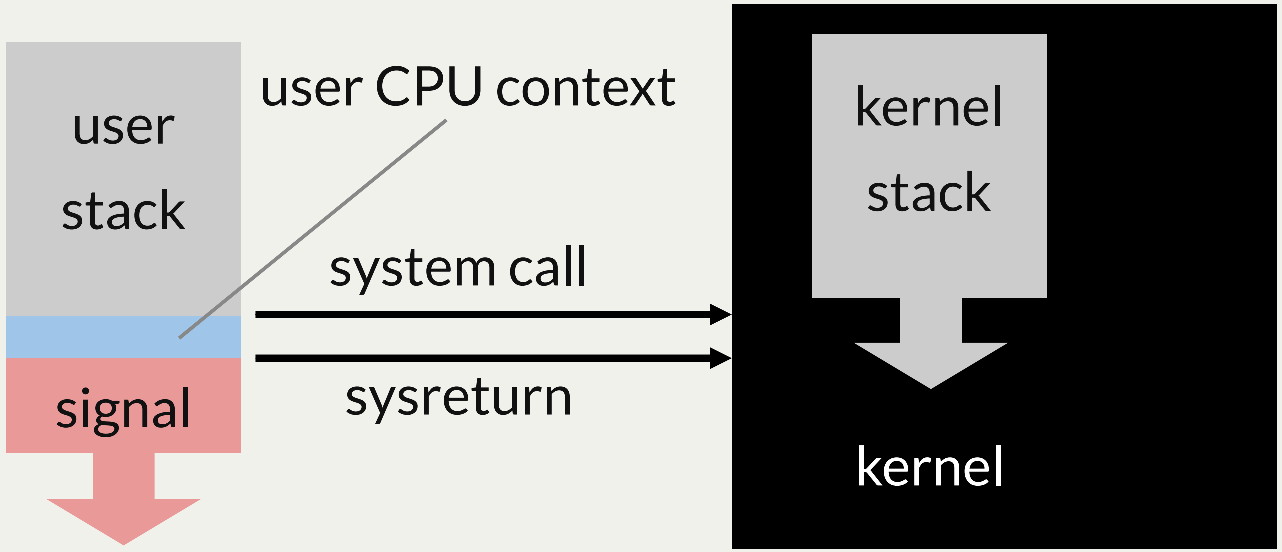
Properly Resuming a Process After a Signal

- When it pushes the stack frame for the signal, it pushes also all of the state needed to resume the user process properly



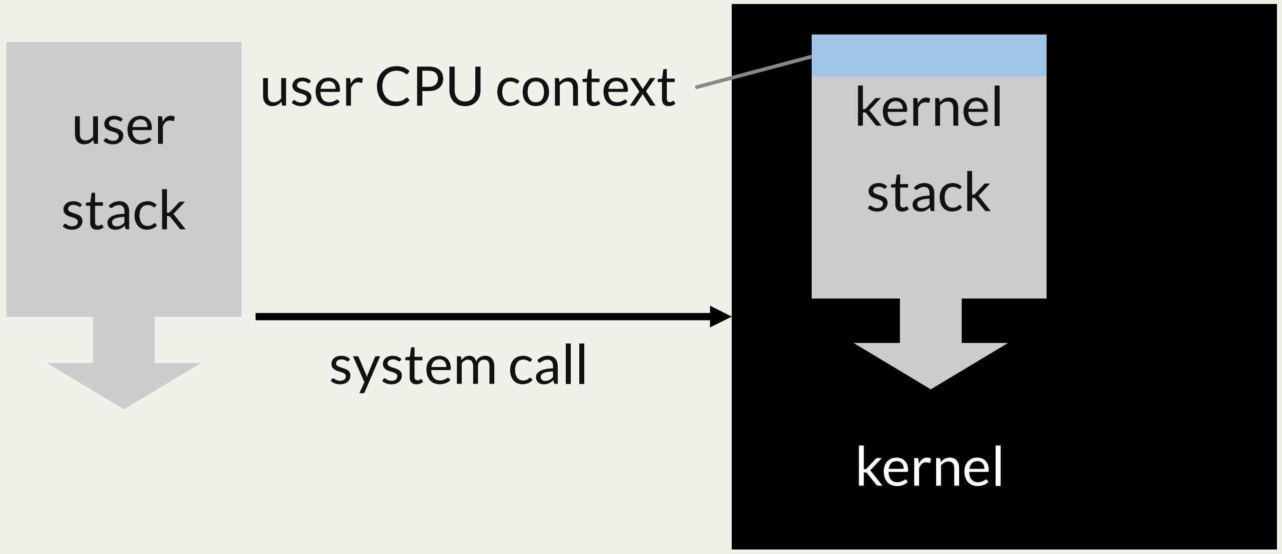
Properly Resuming a Process After a Signal

- When it pushes the stack frame for the signal, it pushes also all of the state needed to resume the user process properly
- When the signal handler returns, it calls the system call **`sysreturn`**



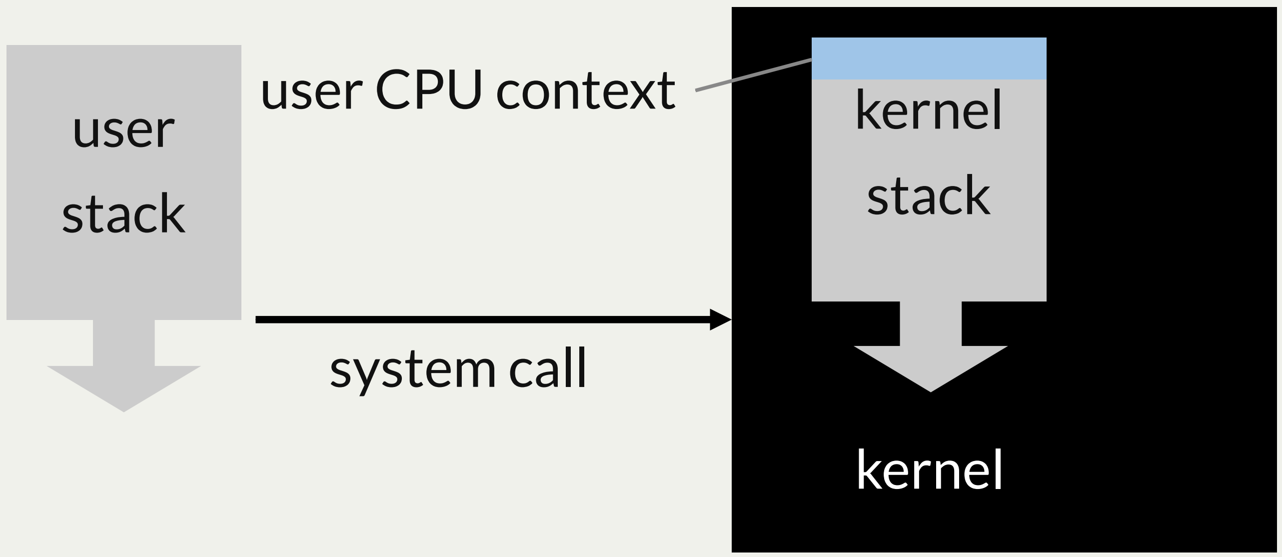
Properly Resuming a Process After a Signal

- When it pushes the stack frame for the signal, it pushes also all of the state needed to resume the user process properly
- When the signal handler returns, it calls the system call **`sysreturn`**
  - This passes the user CPU context back to the kernel, allowing it to restore the process



Concurrency

- A signal handler can be called at any point in the process execution
- You can be in the middle of executing some code and have your signal handler called



Playing With Fire

https://cplayground.com/?p=magpie-chinchilla-penguin

What Happened: Concurrency

- Recall, a signal handler can be called at any point in the process execution
- What if it executes right when we return from the system call:

```
counter_1++;
gettimeofday(&now, NULL);
counter_2++;
```

- counter_1 will be set to 0, counter_2 will be set to 0, then counter_2 will be incremented
- Because the signal handler can *preempt* your code, and run seemingly at any time, you need to be careful about any state they share
- This is a limited form of *concurrency*, but raises many of the same issues as when two pieces of code run at the same time (and share memory)

Concurrency

- One of the seven key systems principles we'll be covering this quarter
- Concurrency: performing multiple actions at the same time

- Concurrency is extremely powerful: it can make your systems faster, more responsive, and more efficient. It's fundamental to all modern software.
- But it's also very tricky to program -- we will spend a good deal of the quarter showing you all of the challenges and the mechanisms we use to tackle them (starting next lecture)
  - It boils down to shared data, and making sure code always sees that data in a consistent state, e.g., doesn't see counter_1 and counter_2 be different
  - Data analytics frameworks make it possible to massively parallelize computations by defining a data model where there is almost no shared data: the data is split into many independent chunks that are processed in parallel

This Lecture

- Pipes for interprocess communication
- Managing file descriptors
- subprocess example
- Signals and their semantics
- Using SIGCHLD to manage subprocesses
- Edge cases caused by signal semantics
- Signal execution model
- Concurrency