❮ Prev (https://www.ops-class.org/slides/2017-02-08-forkandfilehandles/) 🎞 (deck.html) Next ❯ (https://www.ops-class.org/slides/2017-02-13-synchprimitives/)

fork() and pipe()

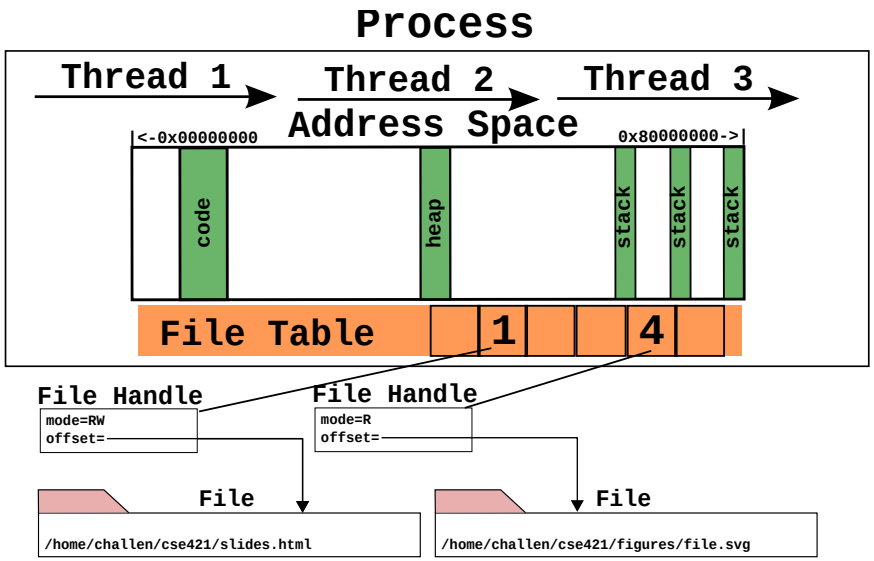
Process Creation

Where do processes come from?

fork() # create a new process

fork() is the UNIX system call that creates a new process.

- fork() creates a new process that is a **copy** of the calling process.
- After fork() we refer to the caller as the **parent** and the newly-created process as the **child.** This relationship enables certain capabilities.



fork() Semantics

- Generally fork() tries to make an **exact** copy of the calling process.
  - Recent version of UNIX have relaxed this requirement and there are now many flavors of fork() that copy different amounts of state and are suitable for different purposes.
  - For the purposes of this class, ignore them.
- Threads are a notable exception!

fork() Against Threads

- Single-threaded fork() has reliable semantics because the **only thread** the processes had is the one that called fork().
  - So nothing else is happening while we complete the system call.

- Multi-threaded fork() creates a host of problems that many systems choose to ignore.
    - Linux will only copy state for the thread that called fork().

Multi-Threaded fork()

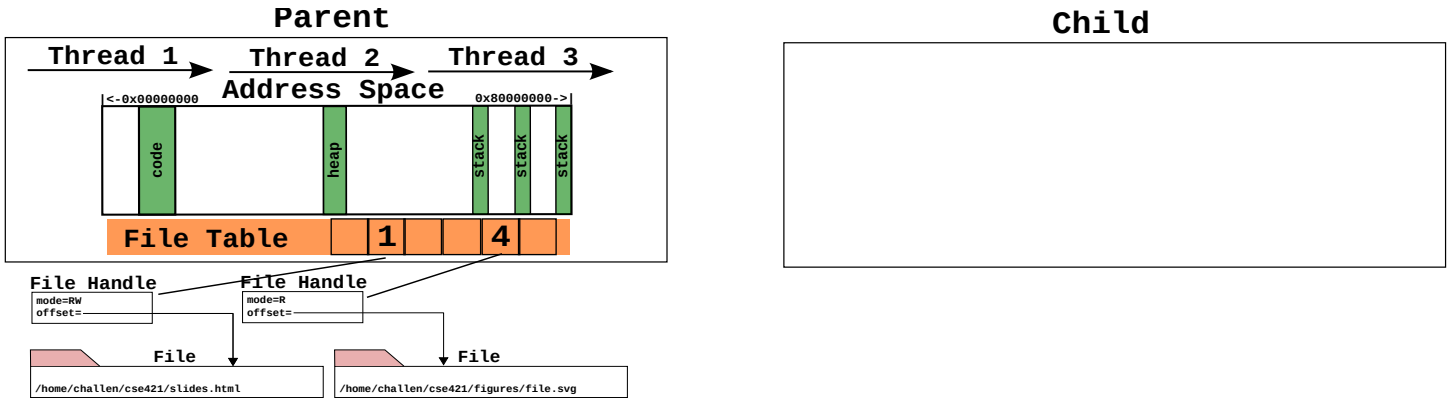
There are two major problems with multi-threaded fork()

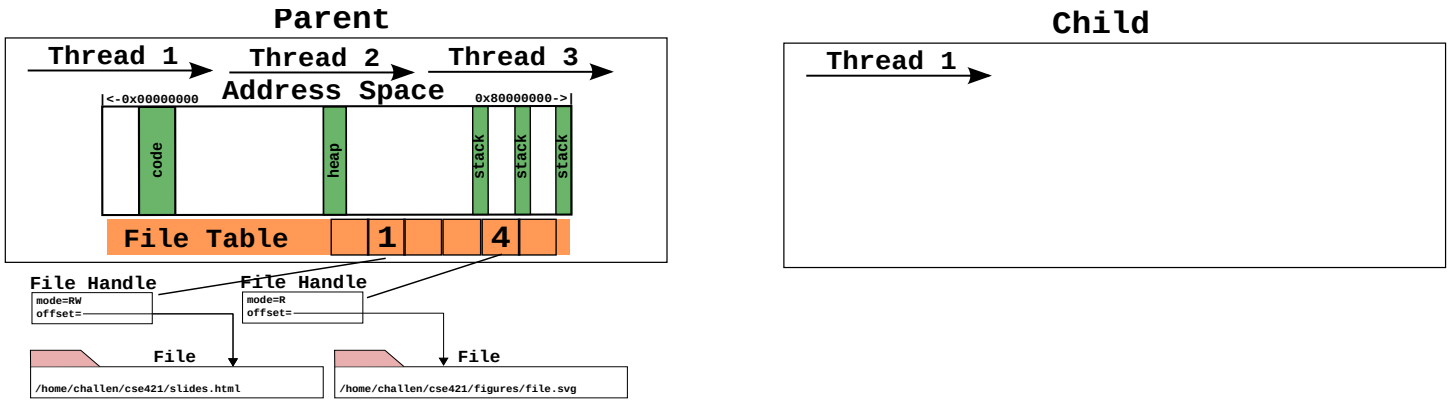
1. Another thread could be blocked in the middle of doing something (uniprocessor systems), or
2. another thread could be **actually** doing something (multiprocessor systems).

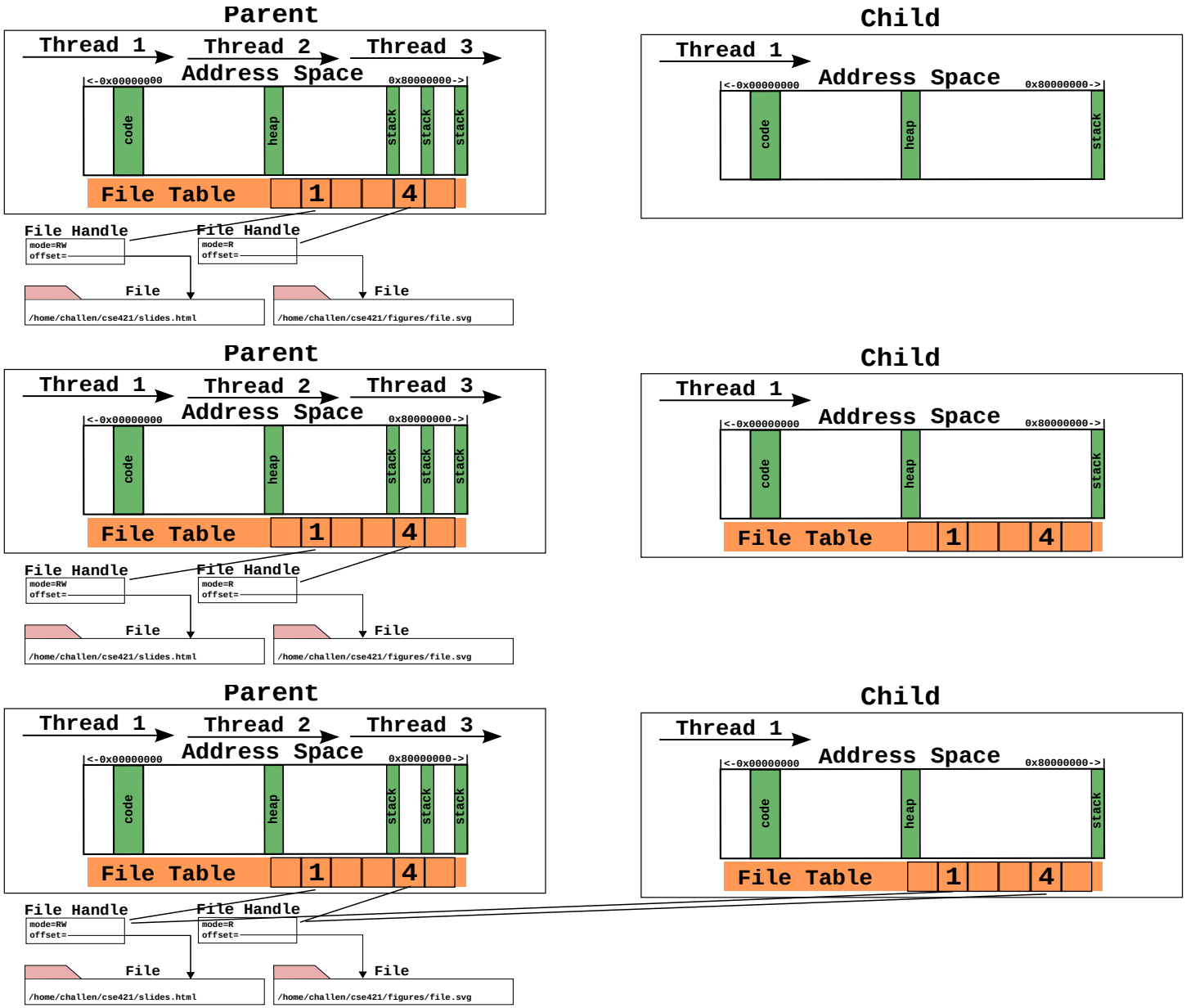
This ends up being a big mess. Let's just copy the calling thread.

fork()

1. fork() copies one thread–the caller.
2. fork() copies the address space.
3. fork() copies the process file table.






Parent
Thread 1
Thread 2
Thread 3
Address Space
<-0x00000000
0x80000000->
code
heap
stack
stack
stack
File Table
1
4
File Handle
mode=RW
offset=
File Handle
mode=R
offset=
File
/home/challen/cse421/slides.html
File
/home/challen/cse421/figures/file.svg
Child
Thread 1
Address Space
<-0x00000000
0x80000000->
code
heap
stack
File Table
1
4

After fork()

```
returnCode = fork();
if (returnCode == 0) {
  # I am the child.
} else {
  # I am the parent.
}
```

- The child thread returns executing at the exact same point that its parent called fork().
  - With one exception: fork() returns **twice**, the PID to the parent and 0 to the child.
- All contents of memory in the parent and child are identical.
- Both child and parent have the same files open at the same position.
  - **But, since they are sharing file handles changes to the file offset made by the parent/child will be reflected in the child/parent!**

Calm Like A fork()bomb

What does this code do?

```
while (1) {
  fork();
}
```

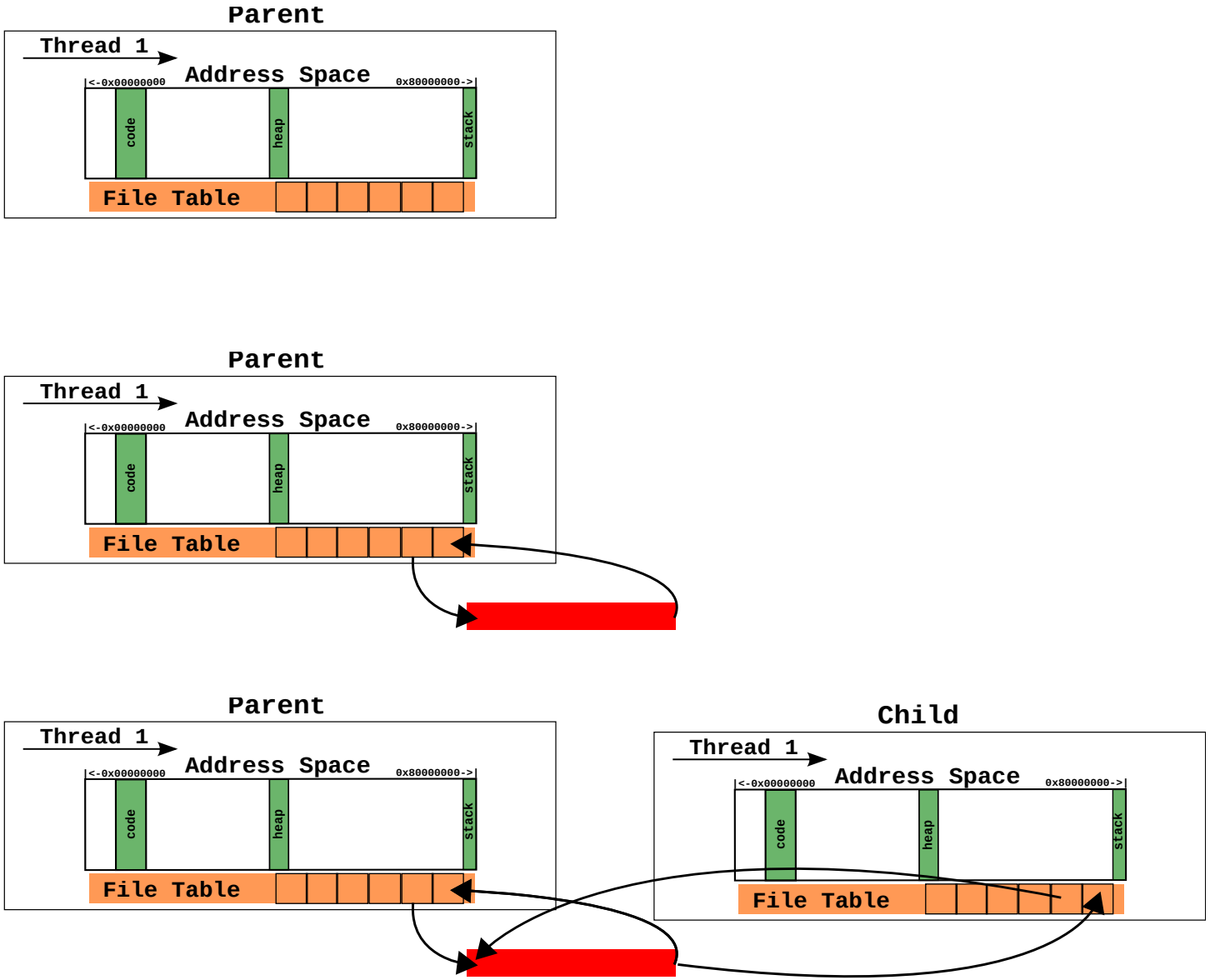
Pipes

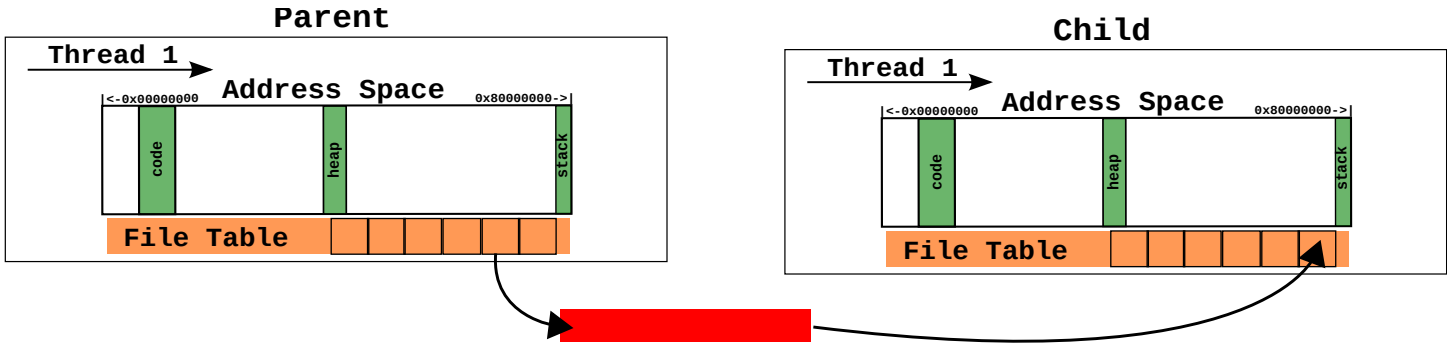
Chains of communicating processes can be created by exploiting the pipe() system call.

- pipe() creates an anonymous pipe object and returns a two file descriptors: one for the read-only end, and the other for the write-only end.
- Anything written to the write-only end of the pipe is immediately available at the read-only end of the pipe.
- Pipe contents are buffered in memory.
- **Why is this useful?**

IPC Using fork() and pipe()

1. Before calling fork() the parent creates a pipe object by calling pipe().
2. Next, it calls fork().
3. After fork() the parent closes its copy of the read-only end and the child closes its copy of the write-only end.
4. Now the parent can pass information to the child.


Parent
Thread 1
Address Space
<-0x00000000
0x80000000->
code
heap
stack
File Table
Parent
Thread 1
Address Space
<-0x00000000
0x80000000->
code
heap
stack
File Table
Parent
Thread 1
Address Space
<-0x00000000
0x80000000->
code
heap
stack
File Table
Child
Thread 1
Address Space
<-0x00000000
0x80000000->
code
heap
stack
File Table




```
# pipeEnds[0] gets the read end; pipeEnds[1] gets the write end.
int pipeEnds[2];

pipe(pipeEnds);

int returnCode = fork();

if (returnCode == 0) {

  # Don't need a loopback.
  close(pipeEnds[1]);

  # Read some data from the pipe.
  char data[14];
  read(pipeEnds[0], data, 14);
} else {

  # Don't need a loopback.
  close(pipeEnds[0]);

  # Write some data to the pipe.
  write(pipeEnds[1], "Hello, sweet child!\n", 14);
}
```

Issues with fork()

Copying all that state is expensive!

- Especially when the next thing that a process frequently does is start load a new binary which destroys most of the state fork() has carefully copied!

Several solutions to this problem:

- **Optimize existing semantics:** through copy-on-write, a clever memory-management optimization we will discuss in several weeks.
- **Change the semantics:** vfork(), which will fail if the child does anything other than immediately load a new executable.

- Does not copy the address space!

What if I don't want to copy all of my process state?

- fork() is now replaced by clone(), a more flexible primitive that enables more control:
  - over sharing, including sharing memory, and signal handlers,
  - and over child execution, which begins at a function pointer passed to the system call instead of resuming at the point where fork() was called.
- Try man clone in your CSE421 VM.

The Tree of Life

- fork() establishes a parent-child relationship between two process at the point when each one is created.
- The pstree utility allows you to visualize these relationships.

```
challen@cse421:~$ pstree
init─┬─apache2─┬─apache2
     │         ├─2*[apache2───17*[{apache2}]]
     │         └─2*[apache2───26*[{apache2}]]
     ├─atd
     ├─cron
     ├─dbus-daemon
     ├─dhclient3
     ├─exim4
     ├─6*[getty]
     ├─git-daemon
     ├─mailmanctl───8*[python]
     ├─rsyslogd───3*[{rsyslogd}]
     ├─sshd───sshd───sshd───bash───pstree
     ├─udevd───2*[udevd]
     ├─upstart-socket-
     └─upstart-udev-br
challen@cse421:~$
```