# More UNIX I/O

Benjamin Brewster

# Open File Inheritance

- When you `exec…()` a process, you replace the current process with a new one, but the files are still open and accessible to the new process

- This may not be what you want!

- In this lecture, we discuss other IPC methods and UNIX I/O. We'll cover:
  - Close On Exec
  - File redirection in C
  - Pipes

# I/O redirection

- We saw I/O redirection in the shell:
  - `ls > file`
  - `stats < file1`
  - `cat longfile | more`
  - `find . -name "paper" -print 2> /dev/null`
  - `echo "an error occurred" 1>&2`

- I/O redirection is possible *because* open files are shared across `fork()` and `exec…()`

# I/O redirection

- I/O redirection is possible *because* open files are shared across `fork()` and `exec…()`

- Each child process has the same files open as its parent, as the file descriptors are the same file descriptors in both

- Note this important point: both parent and child read & write function calls move the *same* file pointer for the shared file descriptor!
    - To prevent this, have one of the processes close and re-open the file

# Sharing File Pointers - Example 1 of 2

```
$ cat forkFPsharing.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

void main()
{
  pid_t forkPID;
  int childExitMethod;
  int fileDescriptor;
  char *newFilePath = "./newFile.txt";
  char readBuffer[8];
  memset(readBuffer, '\0', sizeof(readBuffer));

  printf("PARENT: Opening file.\n");
  fileDescriptor = open(newFilePath, O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  if (fileDescriptor == -1) { printf("Hull breach - open() failed on \"%s\"\n", newFilePath); exit(1); }

  printf("PARENT: Writing 01234 to file.\n");
  write(fileDescriptor, "01234", 5);
  printf("PARENT: New FP position (all FP positions are zero-indexed): %d\n", lseek(fileDescriptor, 0, SEEK_CUR));
  fflush(stdout);

  printf("PARENT: Spawning child.\n");
  forkPID = fork();
```

The child process will have `fileDescriptor` open, with the SHARED file pointer!

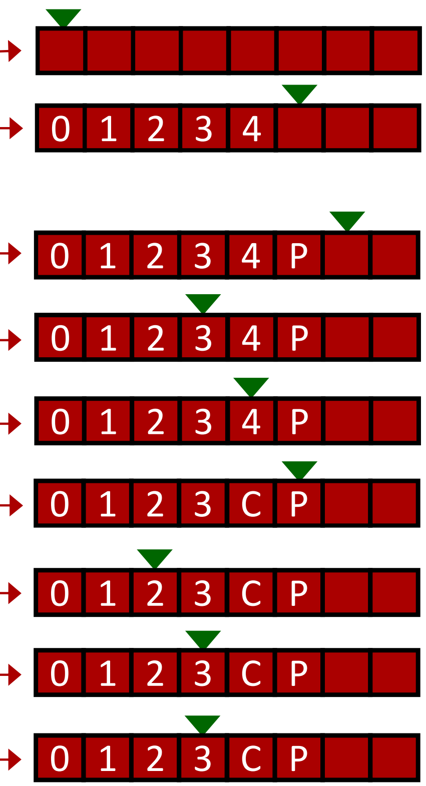
# Sharing File Pointers - Example 2 of 2

This code and trace statements just reads and writes to a file open and shared in both the parent and child processes

```
  switch (forkPID)
  {
  case -1: perror("Hull Breach!"); exit(1); break;
  case 0:
    printf("CHILD: Writing C to file.\n"); fflush(stdout);
    write(fileDescriptor, "C", 1);
    printf("CHILD: After write, new FP position: %d\n", lseek(fileDescriptor, 0, SEEK_CUR)); fflush(stdout);
    printf("CHILD: lseek back 3 chars.\n"); fflush(stdout);
    printf("CHILD: New FP position: %d\n", lseek(fileDescriptor, -3, SEEK_CUR)); fflush(stdout);
    printf("CHILD: Reading char.\n"); fflush(stdout);
    read(fileDescriptor, &readBuffer, 1);
    printf("CHILD: After read, new FP position: %d, char read was: %c\n", lseek(fileDescriptor, 0, SEEK_CUR), readBuffer[0]); fflush(stdout);
    break;
  default:
    printf("PARENT: Writing P to file.\n"); fflush(stdout);
    write(fileDescriptor, "P", 1);
    printf("PARENT: After write, new FP position: %d\n", lseek(fileDescriptor, 0, SEEK_CUR)); fflush(stdout);
    printf("PARENT: lseek back 3 chars.\n"); fflush(stdout);
    printf("PARENT: New FP position: %d\n", lseek(fileDescriptor, -3, SEEK_CUR)); fflush(stdout);
    printf("PARENT: Reading char.\n"); fflush(stdout);
    read(fileDescriptor, &readBuffer, 1);
    printf("PARENT: After read, new FP position: %d, char read was: %c\n", lseek(fileDescriptor, 0, SEEK_CUR), readBuffer[0]); fflush(stdout);
    waitpid(forkPID, &childExitMethod, 0);
    lseek(fileDescriptor, 0, SEEK_SET);
    read(fileDescriptor, &readBuffer, 7);
    printf("PARENT: child terminated; file contents: %s\n", readBuffer); fflush(stdout);
    break;
  }
}
```

# Sharing File Pointers - Results

```
$ gcc -o forkFPsharing forkFPsharing.c
$ forkFPsharing
PARENT: Opening file.
PARENT: Writing 01234 to file.
PARENT: New FP position (all FP positions are zero-indexed): 5
PARENT: Spawning child.
PARENT: Writing P to file.
PARENT: After write, new FP position: 6
PARENT: lseek back 3 chars.
PARENT: New FP position: 3
PARENT: Reading char.
PARENT: After read, new FP position: 4, char read was: 3
CHILD: Writing C to file.
CHILD: After write, new FP position: 5
CHILD: lseek back 3 chars.
CHILD: New FP position: 2
CHILD: Reading char.
CHILD: After read, new FP position: 3, char read was: 2
PARENT: child terminated; file contents: 0123CP
```



# Important Background Review

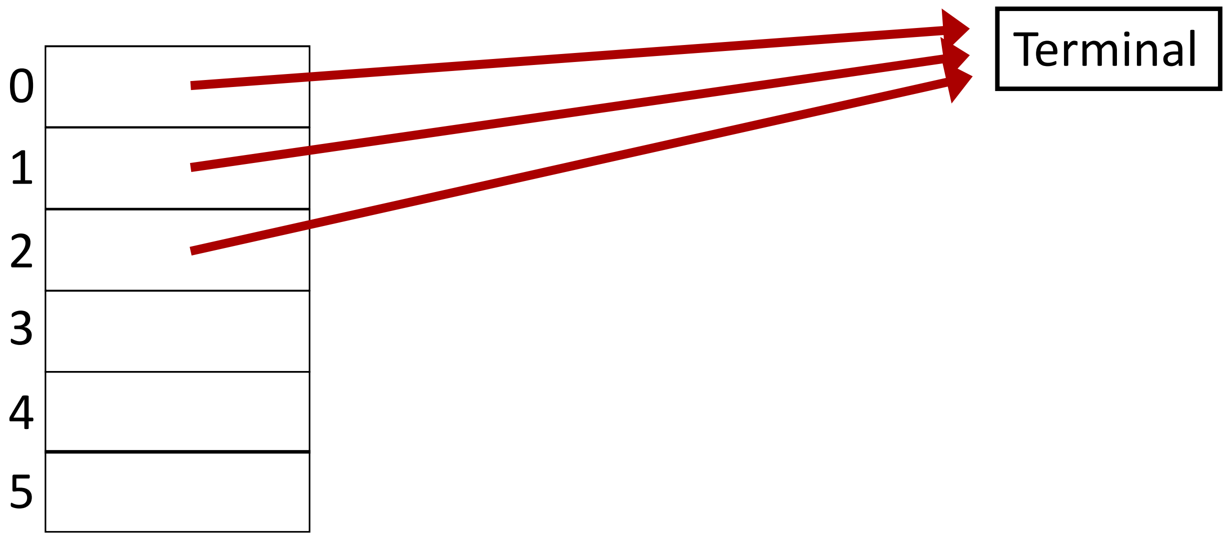
- The kernel opens stdin, stdout, and stderr automatically for every process created:
  - File descriptor 0 is stdin
  - File descriptor 1 is stdout
  - File descriptor 2 is stderr

- They default to reading and writing to the terminal

- The trick: you can change where the standard I/O streams are pointing to at any time before the `exec…()` call, including after the `fork()`

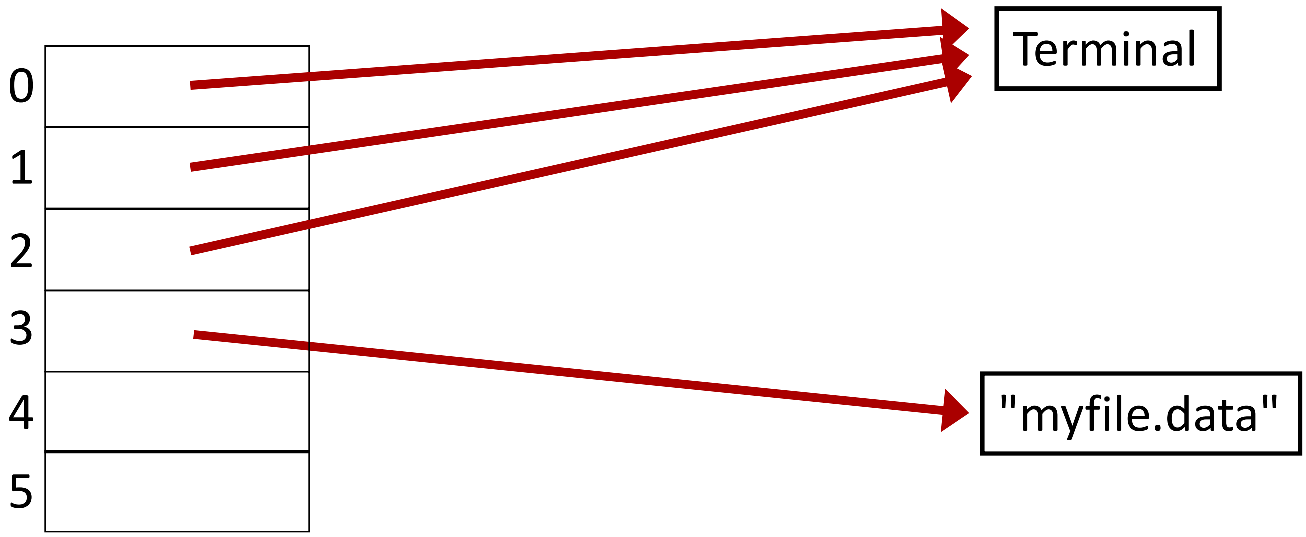
# Redirecting stdout

How all processes start:



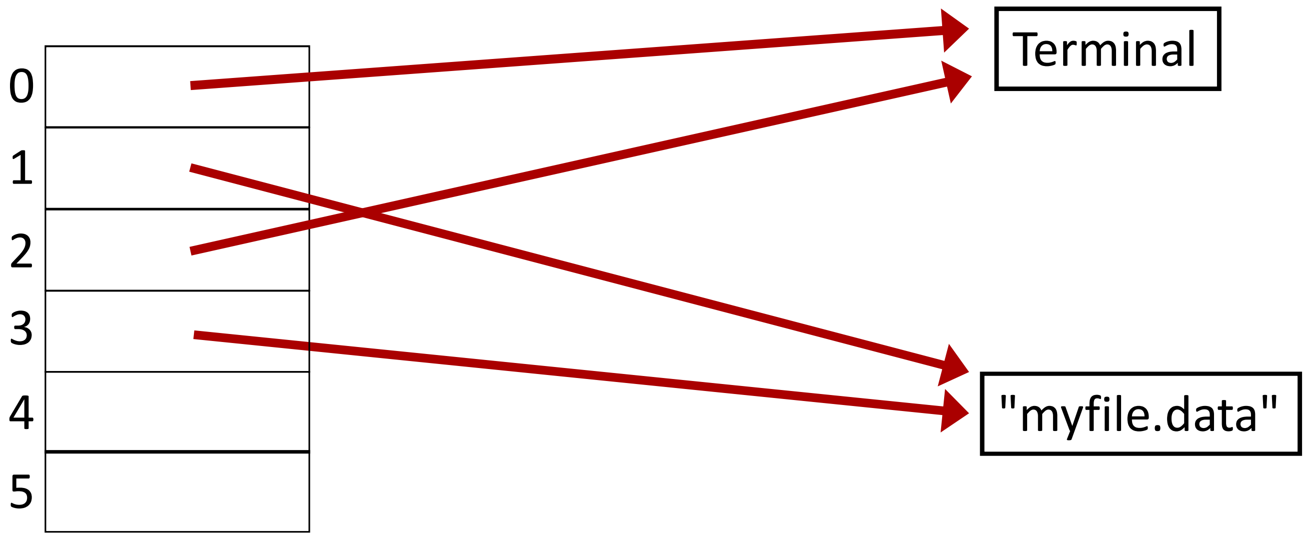
# Redirecting stdout

1. First open the new file



# Redirecting stdout

2. Call `dup2()` to change fd 1 to point where fd 3 points: `dup2(3, 1);`




stdout, merely another name for fd 1, can now be used to access “myfile.data”

# Redirecting stdout

```
$ cat redirectToFile.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char* argv[])
{
        if (argc == 1)
        {
                printf("Usage: redirectToFile <filename to redirect stdout to>\n");
                exit(1);
        }

        int targetFD = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
        if (targetFD == -1) { perror("open()"); exit(1); }
        printf("targetFD == %d\n", targetFD); // Written to terminal

        int result = dup2(targetFD, 1);
        if (result == -1) { perror("dup2"); exit(2); }
        printf("targetFD == %d, result == %d\n", targetFD, result); // Written to file

        return(0);
}
$ gcc -o redirectToFile redirectToFile.c
$ redirectToFile
Usage: redirectToFile <filename to redirect stdout to>
$ redirectToFile test.junk
targetFD == 3
$ cat test.junk
targetFD == 3, result == 1
```

Set FD `1` (stdout) to point to the same place that `targetFD` points.

So, anything written to stdout (like with `printf()`) will go to `targetFD`, which is the passed in filename

# Redirecting stdout & stdin with execlp()

```
$ cat sortViaFiles.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
        int sourceFD, targetFD, result;

        if (argc != 3)
        {
                printf("Usage: sortViaFiles <input filename> <output filename>\n");
                exit(1);
        }

        sourceFD = open(argv[1], O_RDONLY);
        if (sourceFD == -1) { perror("source open()"); exit(1); }
        printf("sourceFD == %d\n", sourceFD); // Written to terminal

        targetFD = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
        if (targetFD == -1) { perror("target open()"); exit(1); }
        printf("targetFD == %d\n", targetFD); // Written to terminal

        result = dup2(sourceFD, 0);
        if (result == -1) { perror("source dup2()"); exit(2); }
        result = dup2(targetFD, 1);
        if (result == -1) { perror("target dup2()"); exit(2); }

        execlp("sort", "sort", NULL);
        return(3);
}
```

Set FD `0` (stdin) to point to the same place that `sourceFD` points.

So, anything in stdin (like the contents of the input file) will be used by this program

Set FD `1` (stdout) to point to the same place that `targetFD` points.

So, anything written to stdout (like the result of sort) will go to `targetFD`, which is the passed in output filename

`execlp()` now starts with stdin and stdout pointing to files, which are used by `sort`

# Redirecting stdout & stdin with execlp()

```
$ cat sortViaFiles.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int r
{
ame> <output filename>\n");

exit(1); }
n to terminal

_TRUNC, 0644);
exit(1); }
n to terminal

it(2); }

it(2); }

    execlp("sort", "sort", NULL);
    return(3);
}
```

*Results:*

```
$ gcc -o sortViaFiles sortViaFiles.c
$ echo -e "3\n1\n2" > junkinput
$ cat junkinput
3
1
2
$ sortViaFiles junkinput junkoutput
sourceFD == 3
targetFD == 4
$ cat junkoutput
1
2
3
```

Set FD `0` (stdin) to point to the same place that `sourceFD` points.

So, anything in stdin (like the contents of the input file) will be used by this program

Set FD `1` (stdout) to point to the same place that `targetFD` points.

So, anything written to stdout (like the result of sort) will go to `targetFD`, which is the passed in output filename

`execlp()` now starts with stdin and stdout pointing to files, which are used by `sort`

# Close On Exec Details

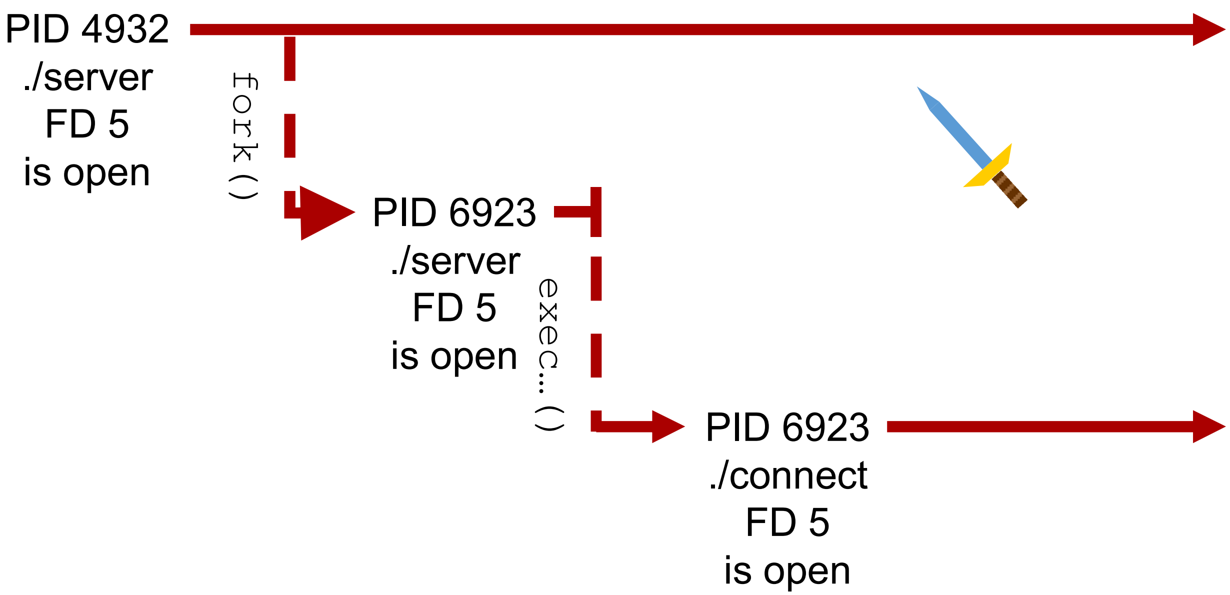
- A flag specified in an open() call that tells the kernel to close any open files when/if the process gets `exec..()`'d
- Why do we care? Because open files are inherited by child processes
  - Thus the file pointer is shared (where the file is currently being read and written to)
  - Secure and/or sensitive data is shared with the new process
- Specific to only that file you added the flag to
- This is inherited through fork, so if the parent specifies it, a child that later calls exec…() will trigger the close on exec flag

# Close on exec example

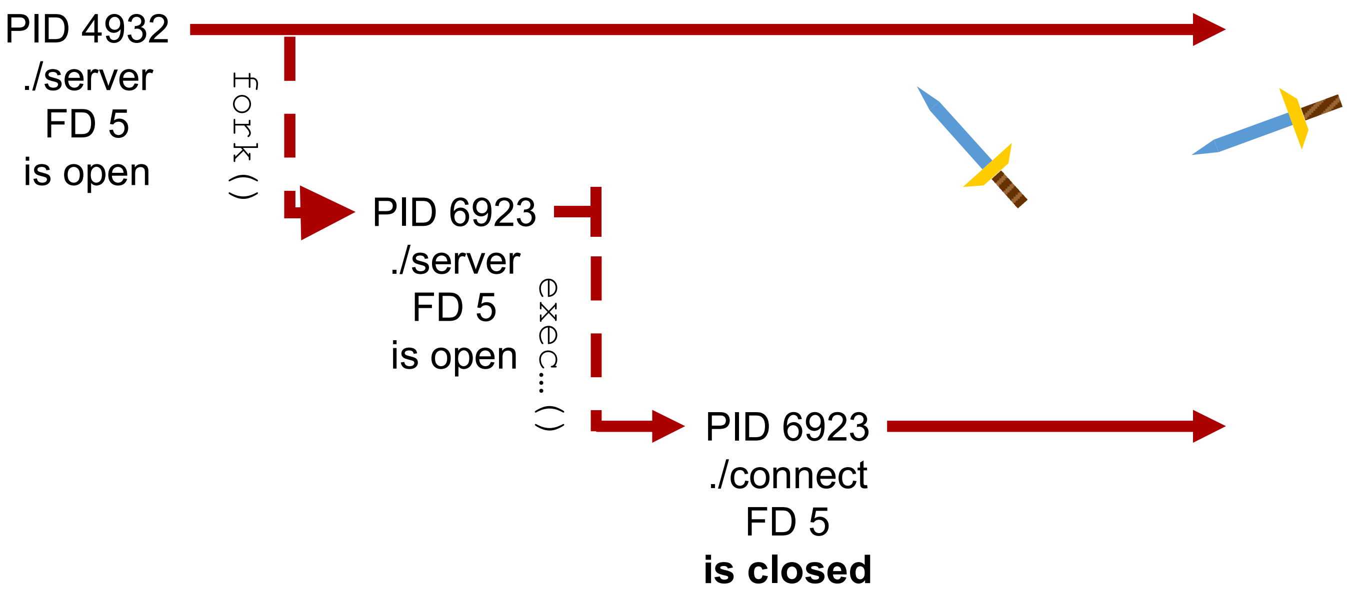
```
#include <fcntl.h>
...
int fd;
fd = open("file", O_RDONLY);
…
fcntl(fd, F_SETFD, FD_CLOEXEC);
…
// exec...
```

Close on Exec

# Normally…


PID 4932
./server
FD 5
is open
fork()
PID 6923
./server
FD 5
is open
exec…()
PID 6923
./connect
FD 5
is open

# With Close On Exec



# Real Inter-Process Communication (IPC)

- IPC methods in UNIX
  - Intermediate/temporary files :: Often used together with I/O redirection
  - Pipes :: IPC between two processes forked by a common ancestor process
  - FIFOs (named pipes) :: communication between any two processes on the same machine
  - Message queues :: communication between any two processes on the same machine
    - Not common; older System V libraries were replaced with POSIX libraries
    - Not a simple byte stream: In Linux, the POSIX queues can be mounted as a filesystem, with each message appearing as a file; can then use `ls`, `rm`, etc.
    - Supports message categories - often used for priorities
  - Sockets :: communication between any two processes, potentially separated by a network

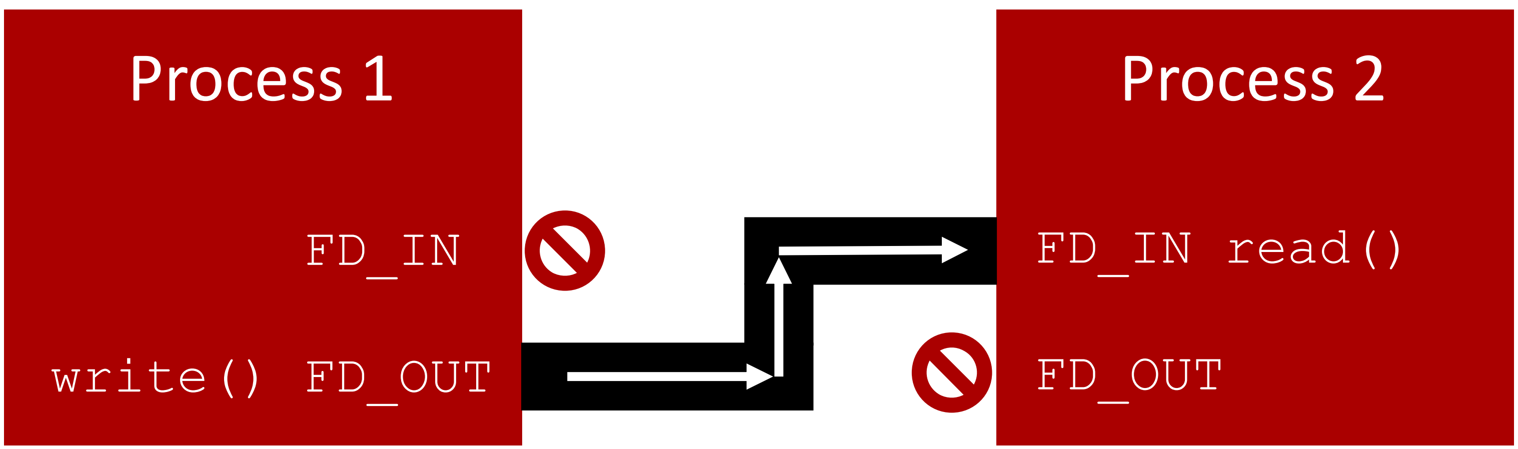
# Between-Process IPC - Intro to Pipes

- I/O redirection with `dup2()` allows you to redirect input and output between processes and files…

- But how do we *redirect input* between processes and other processes on the same machine?
  - We could use temporary/intermediate files, but:
    - Writes to disk are slow
    - No fast & efficient way to track when the other process is ready to receive or send new data other than some sort of semaphore library
  - Better answer: use pipes!

# Pipes

- Pipes provide a way to connect a write-only file descriptor in one process to an read-only file descriptor in another process




- `write()` puts bytes in the pipe, `read()` takes them out

# Creating a Pipe

- Pipes are possible because file descriptors are shared across `fork()` and `exec…()`
- A parent process creates a pipe
  - Results in two new open file descriptors, one for input and one for output
- The parent process calls `fork()` and possibly `exec…()`
  - Parent and child have the file descriptors created with the pipe
- The child process now reads from the input file descriptor, and the parent process writes to the output file descriptor
  - or vice-versa

# The `pipe()` Function

- You pass `pipe()` an array of two integers, where it stores the two new open file descriptors that it creates

- The first is the input file descriptor, and the second is the output file descriptor

- One of the descriptors should be used by the parent process and the other should be used by the child process

We'll talk about how to use a pipe to communicate between two non-decendent processes later

# Flow Control with `read()`

- `read()` succeeds if data is available
  - Receives the data and returns immediately
  - The return value of `read()` tells you how many bytes were read, which *may be less than you requested*

- If data is not available, read() will *block* waiting for data (your process execution is suspended until data arrives)
  - `read()` is a system call

# Flow Control with `write()`

- Similarly, write will not return until all the data has been written
  - `write()` is a system call

- Pipes have a certain size
  - Only so much data will fit in a pipe (typically 64K, but can be changed)
  - If the pipe fills up, and there is no more room, write() will *block* until space becomes available (ie somebody `read`s the data from the pipe)

# pipe() Example

```
$ cat pipeNfork.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
void main()
{
        int r, pipeFDs[2];
        char completeMessage[512], readBuffer[10];
        pid_t spawnpid;

        if (pipe(pipeFDs) == -1) { perror("Hull Breach!"); exit(1); } // Create the pipe with error check

        spawnpid = fork(); // Fork the child, which will write into the pipe
        switch (spawnpid)
        {
        case 0:  // Child
                close(pipeFDs[0]);  // close the input file descriptor
                write(pipeFDs[1], "CHILD: Hi parent!@@", 19); // Write the entire string into the pipe
                exit(0); break; // Terminate the child
        default:  // Parent
                close(pipeFDs[1]);  // close output file descriptor
                memset(completeMessage, '\0', sizeof(completeMessage)); // Clear the buffer

                while (strstr(completeMessage, "@@") == NULL) // As long as we haven't found the terminal...
                {
                        memset(readBuffer, '\0', sizeof(readBuffer)); // Clear the buffer
                        r = read(pipeFDs[0], readBuffer, sizeof(readBuffer) - 1); // Get the next chunk
                        strcat(completeMessage, readBuffer); // Add that chunk to what we have so far
                        printf("PARENT: Message received from child: \"%s\", total: \"%s\"\n", readBuffer, completeMessage);
                        if (r == -1) { printf("r == -1\n"); break; } // Check for errors
                        if (r == 0) { printf("r == 0\n"); break; }
                }
                int terminalLocation = strstr(completeMessage, "@@") - completeMessage; // Where is the terminal
                completeMessage[terminalLocation] = '\0'; // End the string early to wipe out the terminal
                printf("PARENT: Complete string: \"%s\"\n", completeMessage);
                break;
        }
}
$ pipeNfork
PARENT: Message received from child: "CHILD: Hi", total: "CHILD: Hi"
PARENT: Message received from child: " parent!@", total: "CHILD: Hi parent!@"
PARENT: Message received from child: "@", total: "CHILD: Hi parent!@@"
PARENT: Complete string: "CHILD: Hi parent!"
```

```
$ cat pipeNfork.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
void
{
```

## Pointer Arithmetic Example

`int array[3];`

| Element | Address Calculation | Address |
|---|---|---|
| array | 1100 | 1100 |
| array[0] | 1100 + (sizeof(int) * 0) | 1100 |
| array[1] | 1100 + (sizeof(int) * 1) | 1104 |
| array[2] | 1100 + (sizeof(int) * 2) | 1108 |

Therefore:

```
int* x = &array[0];                                          // x is int pointer
int* y = &array[2];                                          // y is int pointer
int z = y - x == 1108 - 1100 == 8 (bytes) => 2 (ints);       // z is int; z = 2
```

```
                        printf("PARENT: Message received from child:                   \n", readBuffer, completeMessage);
                        if (r == -1) { printf("r == -1\n"); break; }
                        if (r == 0) { printf("r == 0\n"); break; }
                }
                int terminalLocation = strstr(completeMessage, "@@") - completeMessage; // Where is the terminal
                completeMessage[terminalLocation] = '\0'; // End the string early to wipe out the terminal
                printf("PARENT: Complete string: \"%s\"\n", completeMessage);
                break;
        }
}
$ pipeNfork
PARENT: Message received from child: "CHILD: Hi", total: "CHILD: Hi"
PARENT: Message received from child: " parent!@", total: "CHILD: Hi parent!@"
PARENT: Message received from child: "@", total: "CHILD: Hi parent!@@"
PARENT: Complete string: "CHILD: Hi parent!"
```

(4068 - 4000) / sizeof(int) = 17

```
$ cat pipeNfork.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
void
{
```

**Could also have just done this:**

```
char* t = strstr(completeMessage, "@@"); // Where is the terminal
*t = '\0';
```

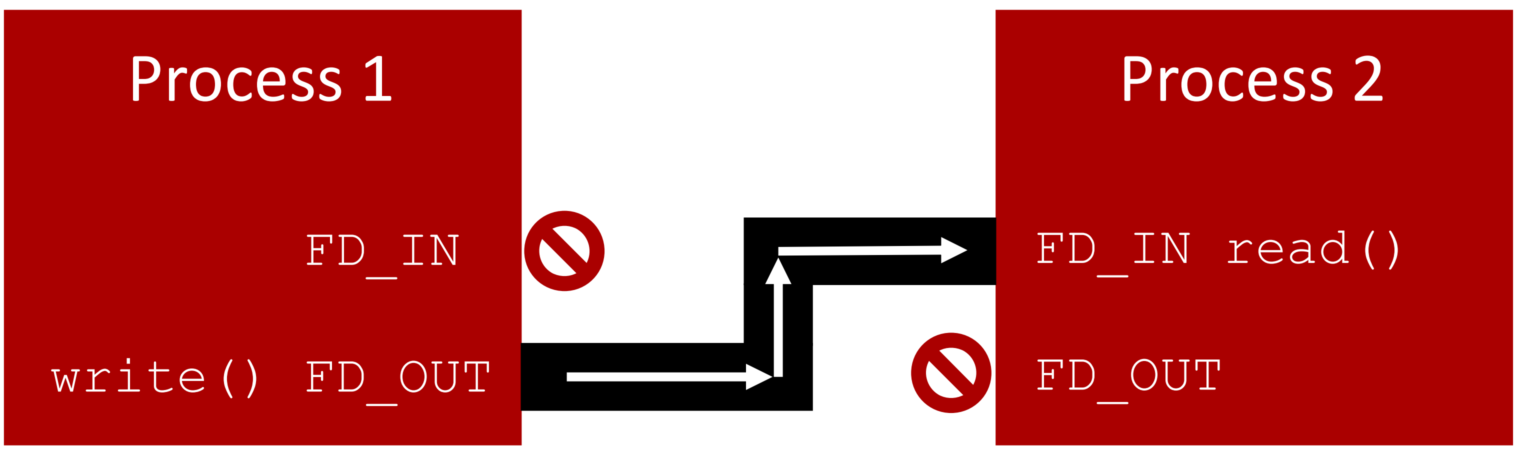
```
                        strcat(completeMessage, readBuffer); // Add                so far
                        printf("PARENT: Message received from child:                \n", readBuffer, completeMessage);
                        if (r == -1) { printf("r == -1\n"); break; }                errors
                        if (r == 0) { printf("r == 0\n"); break; }
                }
                int terminalLocation = strstr(completeMessage, "@@") - completeMessage; // Where is the terminal
                completeMessage[terminalLocation] = '\0'; // End the string early to wipe out the terminal
                printf("PARENT: Complete string: \"%s\"\n", completeMessage);
                break;
        }
}
$ pipeNfork
PARENT: Message received from child: "CHILD: Hi", total: "CHILD: Hi"
PARENT: Message received from child: " parent!@", total: "CHILD: Hi parent!@"
PARENT: Message received from child: "@", total: "CHILD: Hi parent!@@"
PARENT: Complete string: "CHILD: Hi parent!"
```

# Error Checking Reads and Writes

- Checking the return value of `read()` is very important
  - Not just if return value is -1 (an error)
  - The return value will tell you if the desired number of bytes was not read - this can tell you if the pipe didn't have the amount of data you expected it to
  - Our previous example used a terminator @@ instead of tracking byte counts because often you don't know how many bytes there will be
- Same goes for `write()`
  - If the number of bytes returned isn't what you expected (for example, if a signal handler interrupted), you'll need to loop over the write again, *writing only what got missed again* to it

# Closing Pipes

- Process 1 closes output pipe:
  - If process 2 is currently blocked on a `read()`, then process 2's `read()` will return 0
- Process 2 closes input pipe:
  - If process 1 tries to write to the pipe, `write()` will return -1, and errno (in process 1) will be set to EPIPE
  - Process 1 will be sent the SIGPIPE signal



# Named Pipe = FIFO

- FIFO = First-In, First-out
- Essentially, a persistent pipe, which is represented by a special file

- Create in C with `mkfifo()`, or with `mkfifo` in bash
- Once created, any process can open a FIFO with `open()`
- Once opened, it works just like a pipe (or really: just like any file)

# FIFO Use Cases

- You want to build a client-server architecture on a single machine, but you don't want to deal with the complexities of sockets
  - Can be used to transmit data between two non-related processes that didn’t use `pipe()` and then `fork()`
- You want to transmit data with a non-network aware program

# FIFO shell example

```
$ mkfifo my_fifo
$ ls -l *my*
prw-rw----. 1 brewsteb upg57541 0 Oct 31 14:46 my_fifo
```

- Since they are files, you can apply most of the common bash shell commands like:
  - read, sort, wc, cut, awk, etc.
- As well as all of the common file input/output system calls in C: `open()`, `read()`, `write()`, etc.

# Opening a FIFO

- When opening a FIFO, `open()` called by the first process will block; the first process will unblock once the second process calls open
- Example:
  1. Process A calls `open(…, O_RDONLY)` // Process A blocks
  2. Process B calls `open(…, O_WRONLY)` // Process A & B continue
     // execution

# FIFO Example

```
$ cat pipeNforkFIFO.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#include <fcntl.h>
void main()
{
	int r, newfifo, fd;
	char completeMessage[512], readBuffer[10];
	char stringToWrite[20] = "CHILD: Hi parent!@@";
	pid_t spawnpid;
	char* FIFOfilename = "myNewFifo";
	newfifo = mkfifo(FIFOfilename, 0644); // Create the FIFO
	spawnpid = fork(); // Fork the child, which will write into the pipe
	switch (spawnpid) {
	case 0:  // Child
		fd = open(FIFOfilename, O_WRONLY); // Open the FIFO for writing
		if (fd == -1) { perror("CHILD: open()"); exit(1); }
		write(fd, stringToWrite, strlen(stringToWrite)); // Write the entire string into the pipe
		exit(0); break; // Terminate the child
	default:  // Parent
		fd = open(FIFOfilename, O_RDONLY); // Open the FIFO for reading
		if (fd == -1) { perror("PARENT: open()"); exit(1); }
		memset(completeMessage, '\0', sizeof(completeMessage)); // Clear the buffer
		while (strstr(completeMessage, "@@") == NULL) { // As long as we haven't found the terminal...
			memset(readBuffer, '\0', sizeof(readBuffer)); // Clear the buffer
			r = read(fd, readBuffer, sizeof(readBuffer) - 1); // Get the next chunk
			strcat(completeMessage, readBuffer); // Add that chunk to what we have so far
			printf("PARENT: Message received from child: \"%s\", total: \"%s\"\n", readBuffer, completeMessage);
			if (r == -1) { printf("PARENT: r == -1, exiting\n"); break; } // Check for errors
			if (r == 0) { printf("PARENT: r == 0, exiting\n"); break; }
		}
		int terminalLocation = strstr(completeMessage, "@@") - completeMessage; // Where is the terminal
		completeMessage[terminalLocation] = '\0'; // End the string early to wipe out the terminal
		printf("PARENT: Complete string: \"%s\"\n", completeMessage);
		remove(FIFOfilename); // Delete the FIFO
		break;
	}
}
$ pipeNforkFIFO
PARENT: Message received from child: "CHILD: Hi", total: "CHILD: Hi"
PARENT: Message received from child: " parent!@", total: "CHILD: Hi parent!@"
PARENT: Message received from child: "@", total: "CHILD: Hi parent!@@"
PARENT: Complete string: "CHILD: Hi parent!"
```