Inter Process Communication

IPC in Chrome (output of “`pstree -s Chrome`”)

```
ts/MacOS/Google Chrome
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ents/Versions/63.0.3239.132/Google Chrome Helper.app/Contents/MacOS/Google Chrome Helper --t
ts/Versions/63.0.3239.132/Google Chrome Framework.framework/Helpers/crashpad_handler --monit
ts/Versions/63.0.3239.132/Google Chrome Framework.framework/Versions/A/XPCServices/AlertNoti
```

Executables: **Chrome, Chrome Helper, Crashpad Handler, Alert Notification Service**

IPC in Chrome

- Chrome (*seems to be the manager of all the other processes below*)
- Chrome Helper runs as three different roles (depending on command line options)
  - GPU Process
  - NaCL Loader (Native Client)
  - Renderer (*multiple instances, possibly more of them as you open more tabs*)
- Crashpad Handler
- Alert Notification Service

IPC

- Communication among processes **on the same machine**
  - Shared-Memory
  - Direct Message Passing
  - Indirect Message Passing (via a Mailbox / Port)
  - Pipes
- Communication among processes **across different machines**
  - Sockets (CS457 Data Communication)

File Access: streams and descriptors

Stream (FILE *)	Java	C++	Descriptor (int)
stdin	System.in	cin	0
stdout	System.out	cout	1
stderr	System.err	cerr	2

- stdin, stdout, stderr are automatically open for any new process
- new descriptors are assigned an integer ≥ 3
- fileno(a_stream) => returns the int descriptor of a stream.
  Example: fileno(stdout) is 1, fileno(stderr) is 2

```
#include <stdio.h>
#include <fcntl.h>
// and more include
int main() {
  FILE *fs = fopen("abc.txt", "r");
  int fd = open("abc.txt", O_RDONLY);

  // Use the file(s) here
  close (fd);
  fclose(fs);
  return 0;
}
```


Output: I/O Library Functions vs. System Calls

```
// using I/O library functions
#include <stdio.h>

int main () {
  printf ("Hello\n");
  fprintf (stdout, "Hi");

  return 0;
}
```

```
// using system calls
// and file descriptors
#include <unistd.h>

int main () {
  // descriptor 1 => stdout
  write (1, "Hello\n", 7);
  write (STDOUT_FILENO, "Hi", 3);

  return 0;
}
```

similar principles for `scanf()`, `fscanf()`, and `read()`

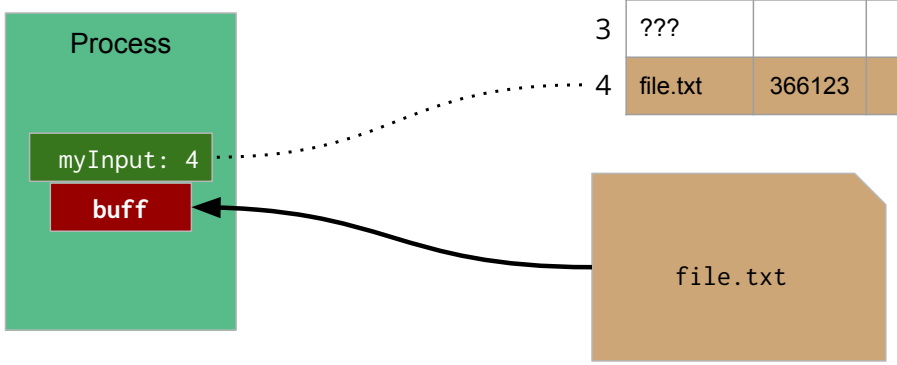
C Library Function	Java	Unix System Call
`fprintf()`	`System,out.print()`	`write()`
`fscanf()`	`var inp = new Scanner(System.in);` `inp.next___();`	`read()`
`fopen()`	`File data = new File("____")`	`open()`
`fclose()`	`data.close()`	`close()`

File Descriptors: Input

```
int myInput = open("/path/to/file.txt", O_RDONLY);
read(myInput, &buff, _____);
```

File Control Block

	Name	Size	Current offset	Disk Addr
0	<stdin>	???	???	???
1	<stdout>	???	???	???
2	<stderr>	???	???	???
3	???			
4	file.txt	366123	3510	2861

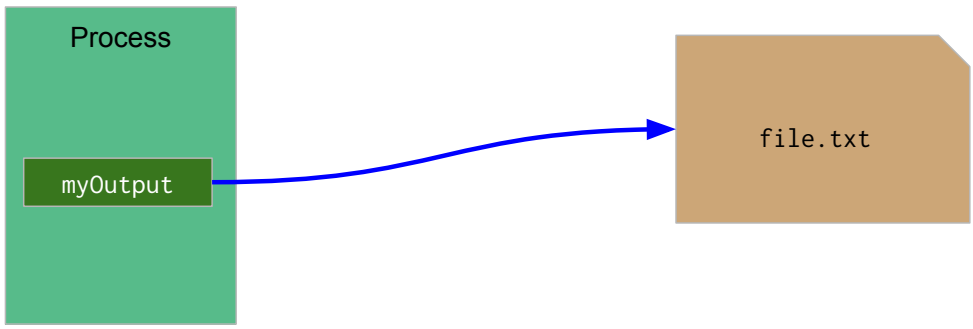


File Descriptors: Input (Simplified Diagram)

```
int myInput = open("/path/to/file.txt", O_RDONLY);
```

File Descriptors: Output (Simplified Diagram)

```
int myOutput = open("/path/to/file.txt", O_WRONLY);
```



dup2(fd, fda)

- Two actions performed by dup2()
  - Close the file (or resource) associated with **fda**
  - Make fda as an alias of fd (“Redirect” fda to fd)
    - *Copy file metadata from FTAB[fd] to FTAB[fda]*
- fd must be a **valid** descriptor (open file)
- Effect of alias:
  - A read from fda will actually read from fd
  - A write to fda will actually write to fd
- Practical use: input/output redirection

dup2(4,0): “copy” FCB[4] to FCB[0]

	Name	Size	Current offset	Disk Addr
0	<stdin>	???	???	???
1	<stdout>	???	???	???
2	<stderr>	???	???	???
3	???			
4	file.txt	366123	3510	2861

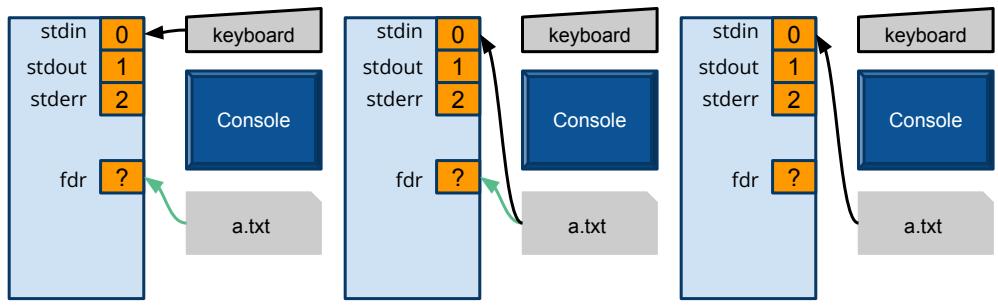
Before dup2(4,0)

	Name	Size	Current offset	Disk Addr
0	*file.txt*	*366123*	*3510*	*2861*
1	<stdout>	???	???	???
2	<stderr>	???	???	???
3	???			
4	file.txt	366123	3510	2861

After dup2(4,0)

dup2(): duplicate a descriptor (input)

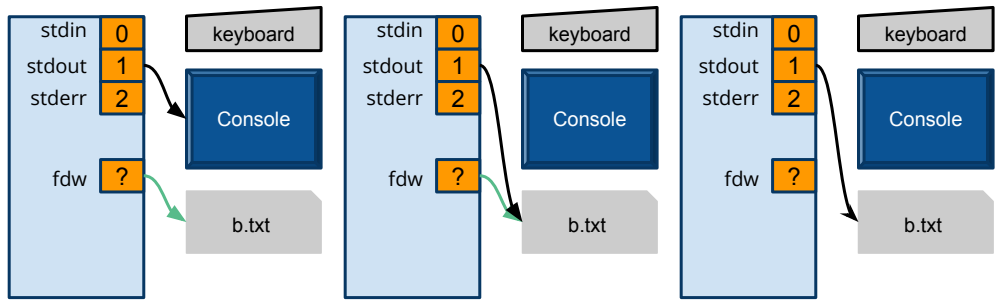
```
int main () {
  int fdr;

  fdr = open ("a.txt",
        O_RDONLY);

  // read from kbd
  fscanf(stdin, ...);

  dup2 (fdr, fileno(stdin));
  close (fdr);

  // from a.txt
  fscanf(stdin, ....);

  return 0;
}
```



Before dup2() After dup2() After close(fdr)

dup2(): duplicate a descriptor (output)

```
int main () {
  int fdw;

  fdw = open ("b.txt",
        O_WRONLY);

  // write to console
  fprintf(stdout, ...);

  dup2 (fdw, fileno(stdout));
  close (fdw);

  // to b.txt
  fprintf(stdout, ....);

  return 0;
}
```



Before dup2() After dup2() After close(fdw)

Can we reopen stdin, stdout, stderr which were closed due to dup2()?

StackOverflow Question

Unix Pipes

Copper Pipes

Law of physics: water flows from high to low pressure

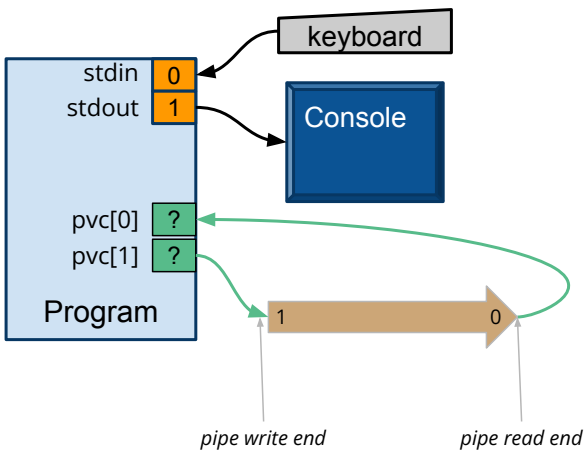
Unix Pipes

Inside a Unix pipe: data flows from [1] to [0]

Unix Pipes

- [Ordinary] Pipes
  - Uni-directional QUEUE (enqueue = write data into pipe, dequeue = read data from pipe)
  - Transient communication channel
  - Can be used for communication between PARENT and CHILD
  - Can be (ab)used for communication to SELF
  - Practical use: command pipes: `ls -l | grep -v "^d" | cat -n`
- Named Pipes
  - Bi-directional
  - Permanent channel (created as part of the Unix file system)
  - Can be shared by MANY processes (no parent-child relationship required)
- Both types can be manipulated using file-related system calls: open(), read(), write(), close()

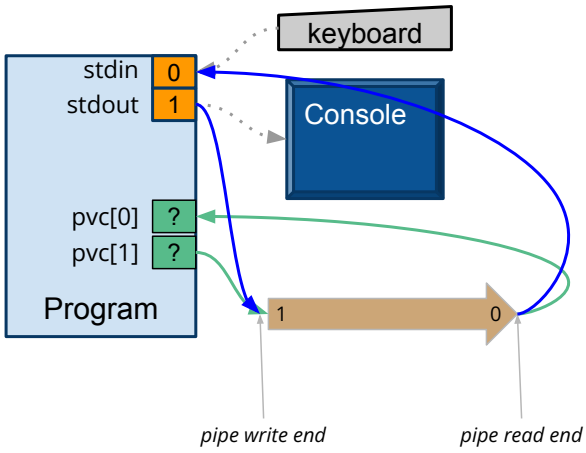
pipe(): transfer data to self (not useful)




```
int main () {
  int pvc[2];
  double num, val;
  char buff[50];

  pipe(pvc);

  write (pvc[1], &num, sizeof(double));
  write (pvc[1], "Help", 5); // 4 char + NULL

  read(pvc[0], &val, sizeof(double));
  read(pvc[0], buff, 5);
  return 0;
}
```

Be sure arrow directions represent INPUT/OUTPUT correctly

pipe() + dup2(): transfer data to self (not useful)

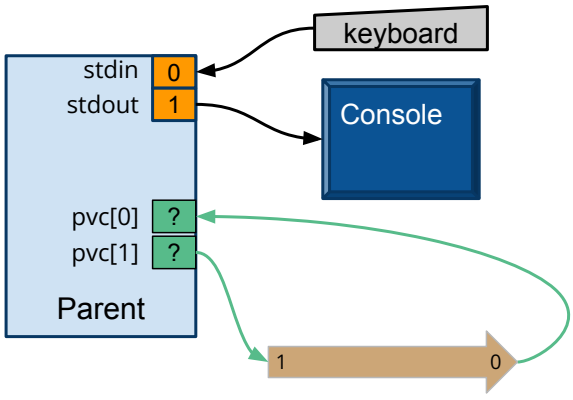



```
int main () {
  int pvc[2];
  double num, val;
  char buff[50];

  pipe(pvc);
  dup2(pvc[1], STDOUT_FILENO);
  dup2(pvc[0], STDIN_FILENO);
  // close(pvc[0]);
  // close(pvc[1]);
  write (STDOUT_FILENO, &num, sizeof(double));
  write (STDOUT_FILENO, "Help", 5); // 4 char + NULL

  read(STDIN_FILENO, &val, sizeof(double));
  read(STDIN_FILENO, buff, 5);
  return 0;
}
```

man fork

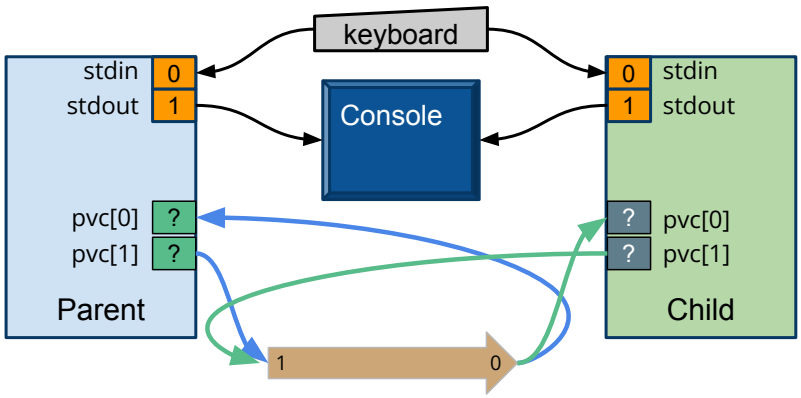
child process inherits OPEN FILES/DESCRIPTORS from parent process

pipe() + fork(): transfer data from parent to child




```
int main () {
  int pvc[2];
  char buff[50];

  pipe(pvc);
  pid_t who = fork();
  if (who > 0) {
    close(pvc[0]);
    write (pvc[1], “Help”, 5);
  } else {
    close(pvc[1]);
    read(pvc[0], buff, 5);
  }
  return 0;
}
```

pipe() + fork()

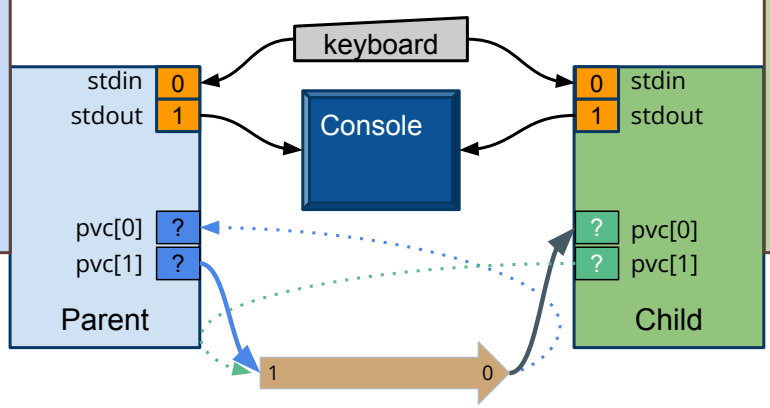



```
int main () {
  int pvc[2];
  char buff[50];

  pipe(pvc);
  pid_t who = fork();
  if (who > 0) {
    close(pvc[0]);
    write (pvc[1], “Help”, 5);
  } else {
    close(pvc[1]);
    read(pvc[0], buff, 5);
  }
  return 0;
}
```

After fork(), before close()

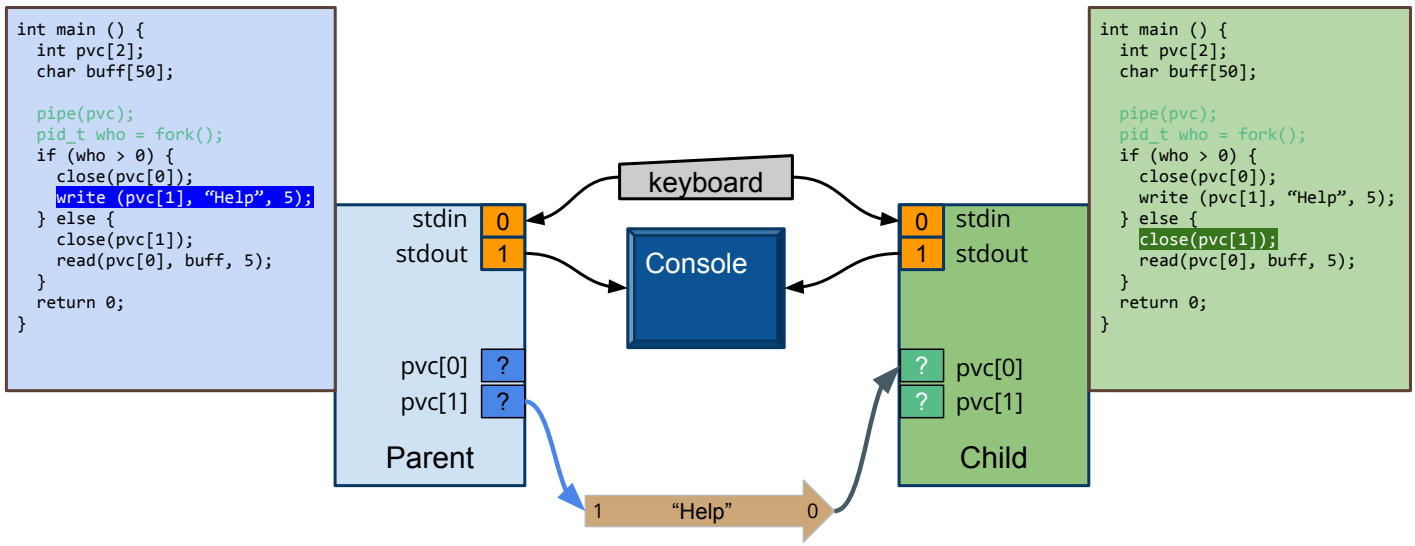
pipe() + fork()

```
int main () {
  int pvc[2];
  char buff[50];

  pipe(pvc);
  pid_t who = fork();
  if (who > 0) {
    close(pvc[0]);
    write (pvc[1], “Help”, 5);
  } else {
    close(pvc[1]);
    read(pvc[0], buff, 5);
  }
  return 0;
}
```

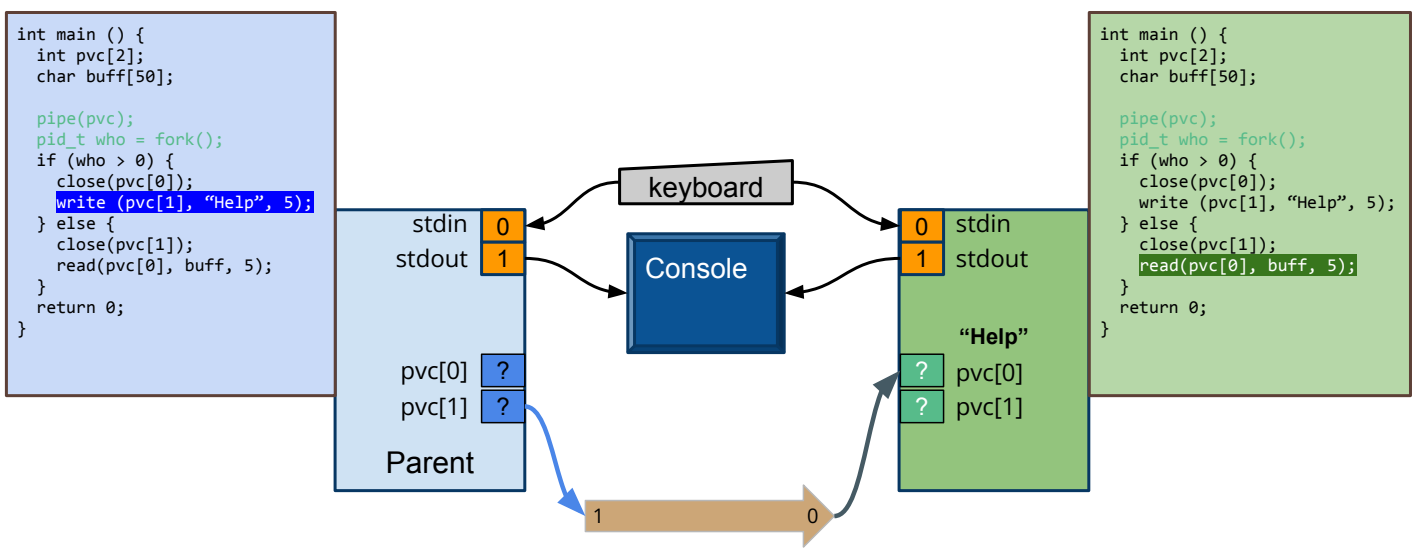



```
int main () {
  int pvc[2];
  char buff[50];

  pipe(pvc);
  pid_t who = fork();
  if (who > 0) {
    close(pvc[0]);
    write (pvc[1], “Help”, 5);
  } else {
    close(pvc[1]);
    read(pvc[0], buff, 5);
  }
  return 0;
}
```

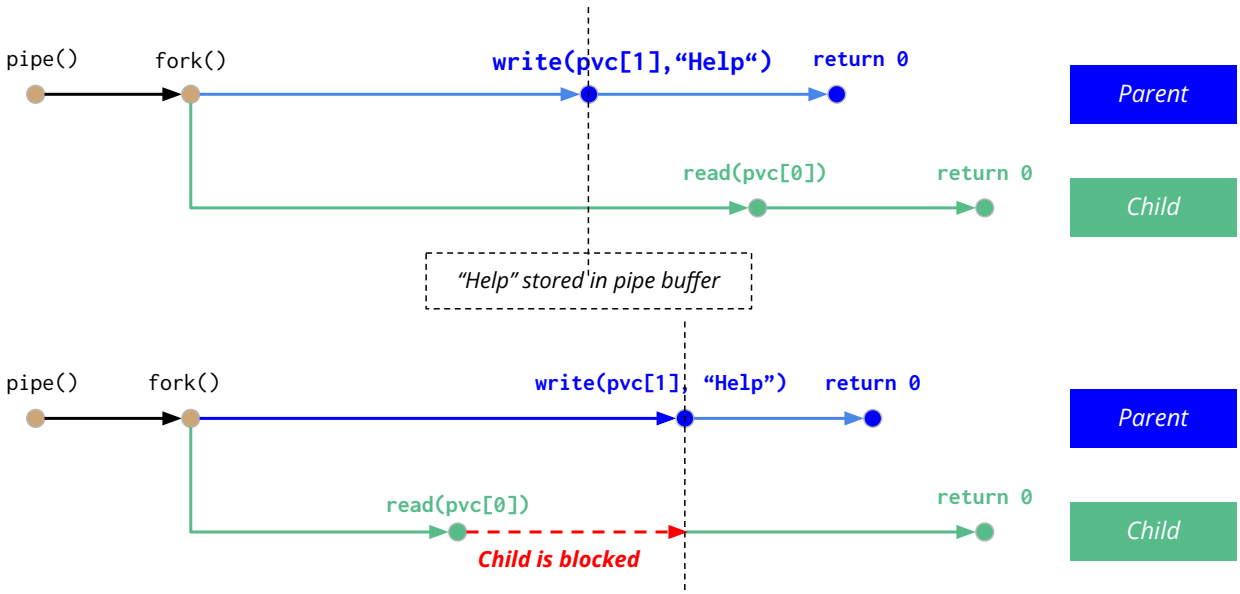
pipe() + fork()



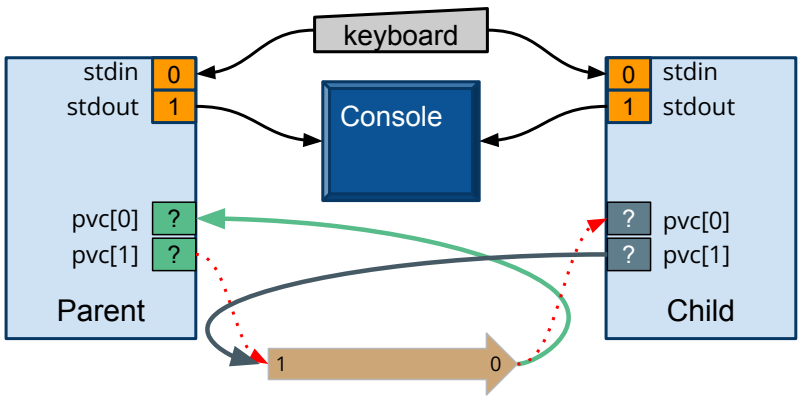
pipe() + fork(): transfer from parent to child



Parent Child Relative Speed



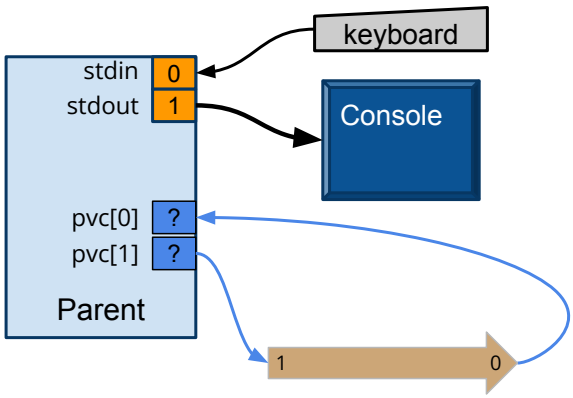
pipe() + fork(): transfer data from child to parent



After close()

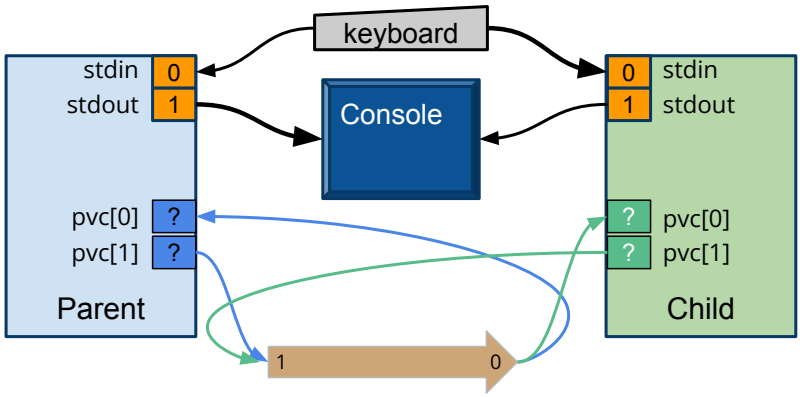
```
int main () {
  int pvc[2];
  char buff[50];

  pipe(pvc);
  pid_t who = fork();
  if (who > 0) {
    close(pvc[1]);
    read(pvc[0], buff, 5);
  } else {
    close(pvc[0]);
    write (pvc[1], "Help", 5);
  }
  return 0;
}
```

pipe() + fork() + dup2()



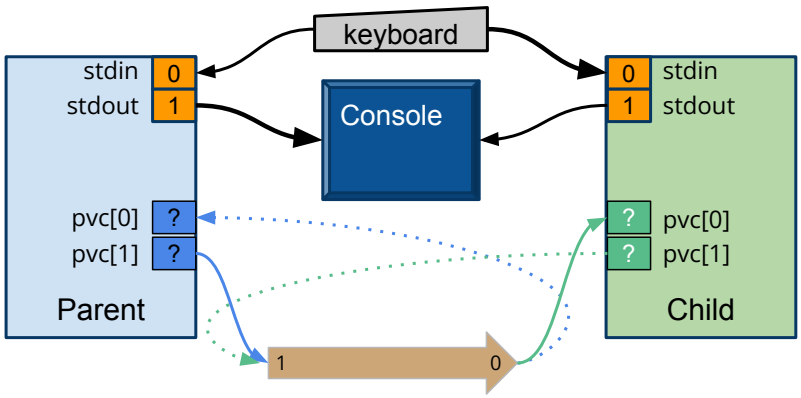

```
int main () {
  int pvc[2];
  char buff[50];

  pipe(pvc);
  pid_t who = fork();
  if (who > 0) { // parent
    close(pvc[0]);
    dup2(pvc[1], STDOUT_FILENO);
    close(pvc[1]);
    write (STDOUT_FILENO, “Help”, 5);
  } else {  // child
    close(pvc[1]);
    dup2(pvc[0], STDIN_FILENO);
    close(pvc[0]);
    read(STDIN_FILENO, buff, 5);
  }
  return 0;
}
```


pipe() + fork() + dup2()



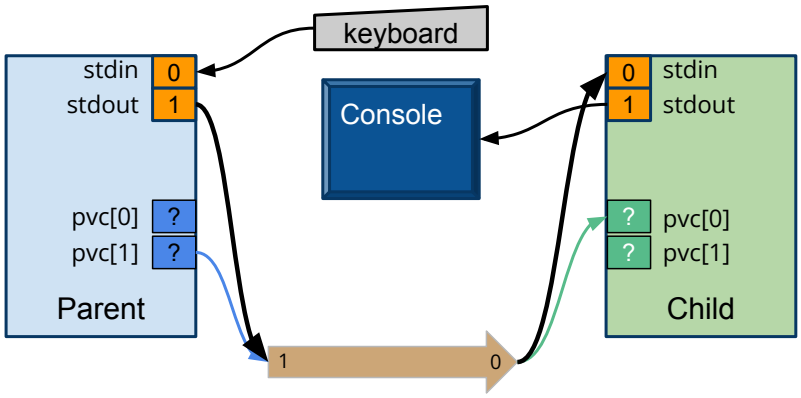

```
int main () {
  int pvc[2];
  char buff[50];

  pipe(pvc);
  pid_t who = fork();
  if (who > 0) { // parent
    close(pvc[0]);
    dup2(pvc[1], STDOUT_FILENO);
    close(pvc[1]);
    write (STDOUT_FILENO, “Help”, 5);
  } else {  // child
    close(pvc[1]);
    dup2(pvc[0], STDIN_FILENO);
    close(pvc[0]);
    read(STDIN_FILENO, buff, 5);
  }
  return 0;
}
```

pipe() + fork() + dup2()



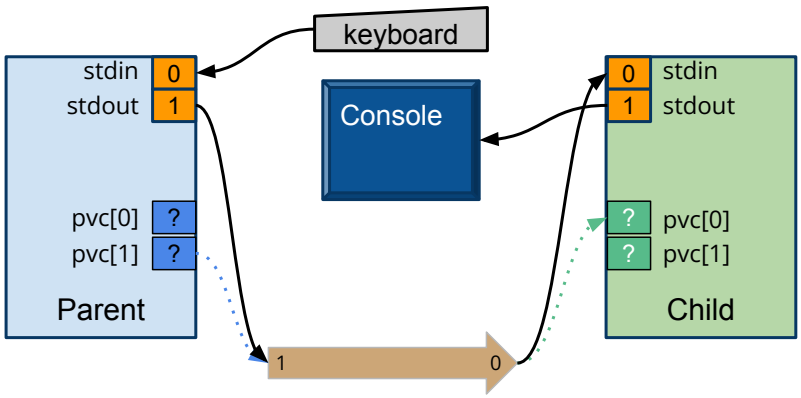

```
int main () {
  int pvc[2];
  char buff[50];

  pipe(pvc);
  pid_t who = fork();
  if (who > 0) { // parent
    close(pvc[0]);
    dup2(pvc[1], STDOUT_FILENO);
    close(pvc[1]);
    write (STDOUT_FILENO, “Help”, 5);
  } else {  // child
    close(pvc[1]);
    dup2(pvc[0], STDIN_FILENO);
    close(pvc[0]);
    read(STDIN_FILENO, buff, 5);
  }
  return 0;
}
```


pipe() + fork() + dup2()



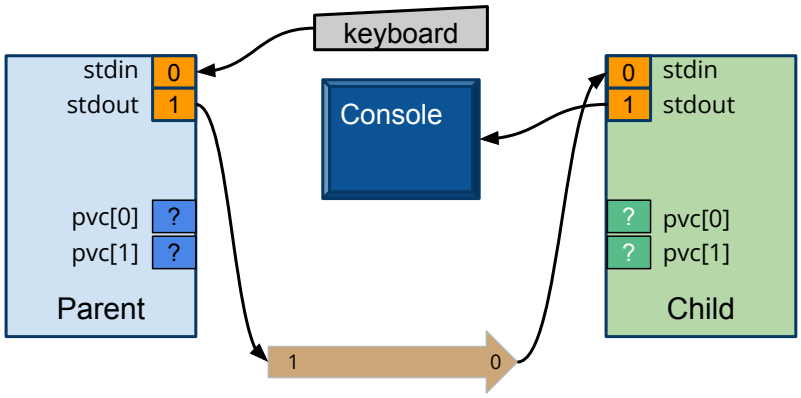

```
int main () {
  int pvc[2];
  char buff[50];

  pipe(pvc);
  pid_t who = fork();
  if (who > 0) { // parent
    close(pvc[0]);
    dup2(pvc[1], STDOUT_FILENO);
    close(pvc[1]);
    write (STDOUT_FILENO, “Help”, 5);
  } else {  // child
    close(pvc[1]);
    dup2(pvc[0], STDIN_FILENO);
    close(pvc[0]);
    read(STDIN_FILENO, buff, 5);
  }
  return 0;
}
```

pipe() + fork() + dup2()




```
int main () {
  int pvc[2];
  char buff[50];

  pipe(pvc);
  pid_t who = fork();
  if (who > 0) { // parent
    close(pvc[0]);
    dup2(pvc[1], STDOUT_FILENO);
    close(pvc[1]);
    write (STDOUT_FILENO, “Help”, 5);
  } else {  // child
    close(pvc[1]);
    dup2(pvc[0], STDIN_FILENO);
    close(pvc[0]);
    read(STDIN_FILENO, buff, 5);
  }
  return 0;
}
```


pipe() + fork() + dup2()




```
int main () {
  int pvc[2];
  char buff[50];

  pipe(pvc);
  pid_t who = fork();
  if (who > 0) { // parent
    close(pvc[0]);
    dup2(pvc[1], STDOUT_FILENO);
    close(pvc[1]);
    write (STDOUT_FILENO, “Help”, 5);
  } else {  // child
    close(pvc[1]);
    dup2(pvc[0], STDIN_FILENO);
    close(pvc[0]);
    read(STDIN_FILENO, buff, 5);
  }
  return 0;
}
```

pipe() can work without dup2()

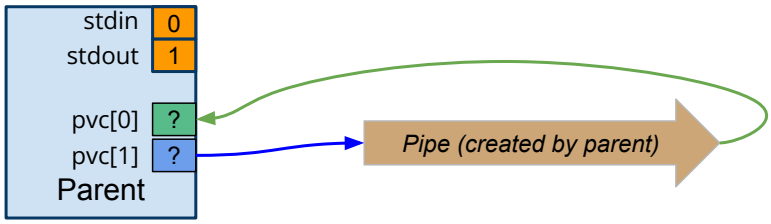
But, it is "*required*" for
pipe() + fork() + exec*()

Shell Pipe

```
ls -R | grep pdf
```

2 child processes and 1 pipe

Running: `ls -r | grep pdf`

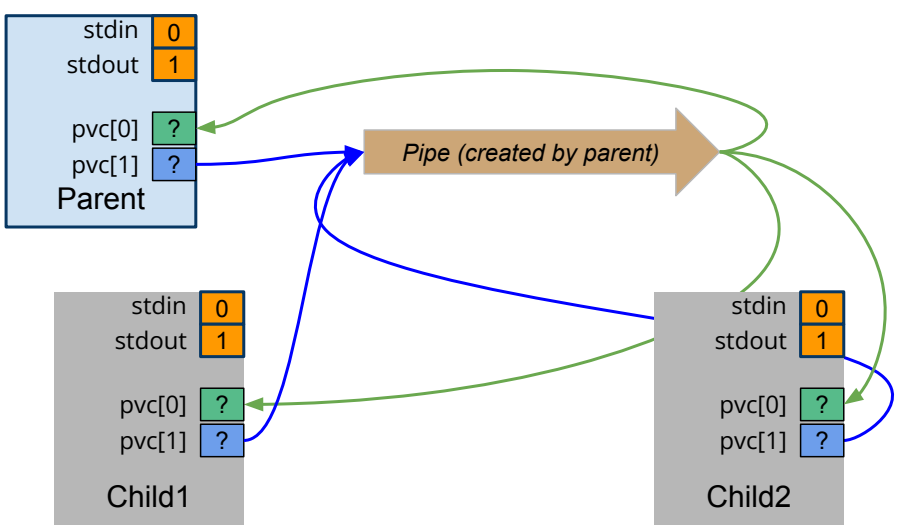


Parent:
- **Create pipe**
- fork() 2x
- close pvc

Child1
- dup(pvc[1], 1)
- close pvc
- execlp ("ls")

Child2
- dup(pvc[0], 0);
- close pvc
- execlp("grep")

Running: `ls -r | grep pdf`

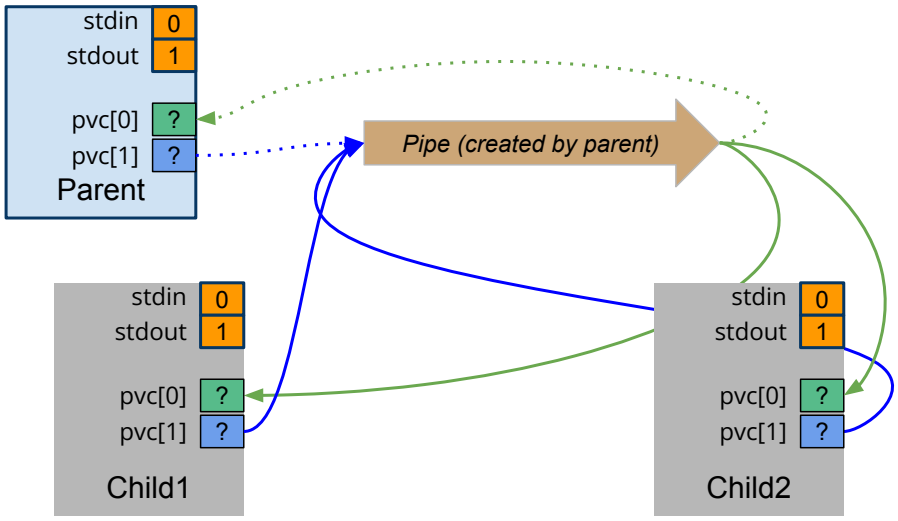


Parent:
- Create pipe
- **fork() 2x**
- close pvc

Child1
- dup(pvc[1], 1)
- close pvc
- execlp("ls")

Child2
- dup(pvc[0], 0);
- close pvc
- execlp("grep")

Running: `ls -r | grep pdf`

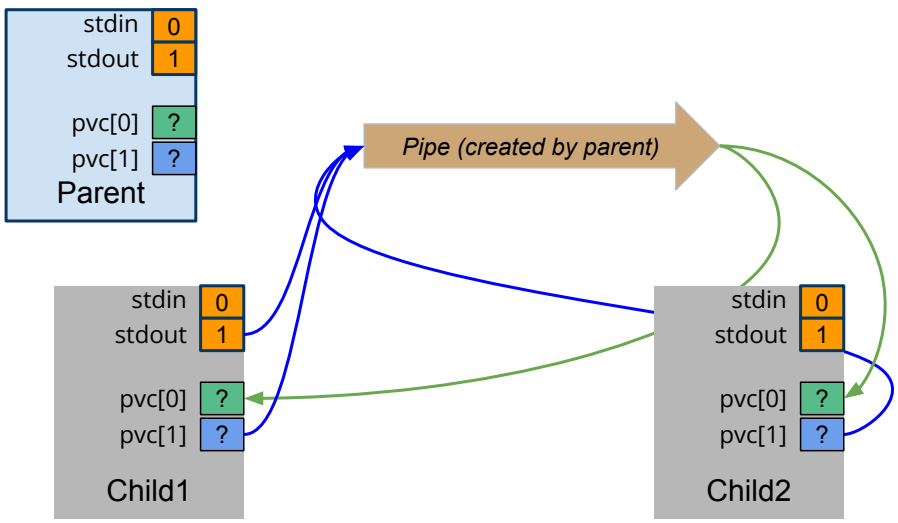


Parent:
- Create pipe
- fork() 2x
- **close pvc**

Child1
- dup(pvc[1], 1)
- close pvc
- execlp("ls")

Child2
- dup(pvc[0], 0);
- close pvc
- execlp("grep")

Running: `ls -r | grep pdf`

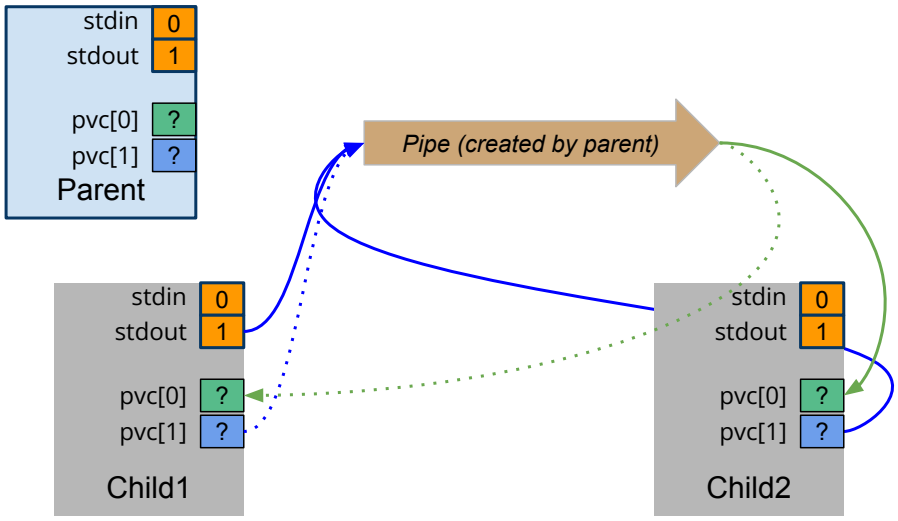


Parent:
- Create pipe
- fork() 2x
- close pvc

Child1
- **dup(pvc[1], 1)**
- close pvc
- execlp("ls")

Child2
- dup(pvc[0], 0);
- close pvc
- execlp("grep")

Running: `ls -r | grep pdf`

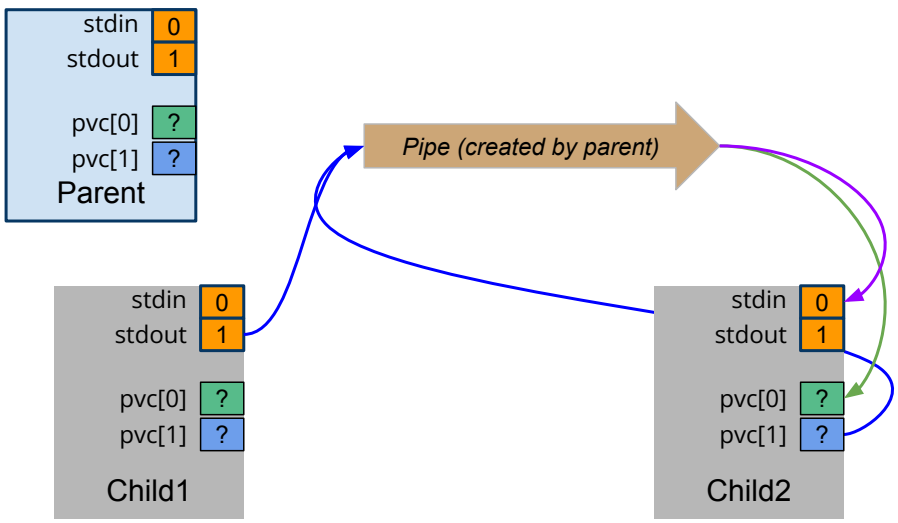


Parent:
- Create pipe
- fork() 2x
- close pvc

Child1
- dup(pvc[1], 1)
- **close pvc**
- execlp("ls")

Child2
- dup(pvc[0], 0);
- close pvc
- execlp("grep")

Running: `ls -r | grep pdf`

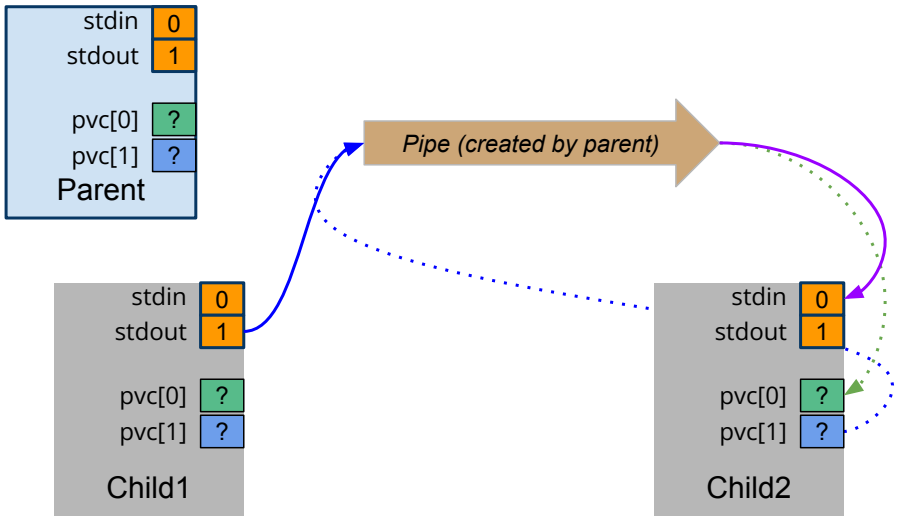


Parent:
- Create pipe
- fork() 2x
- close pvc

Child1
- dup(pvc[1], 1)
- close pvc
- execlp("ls")

Child2
- **dup(pvc[0], 0);**
- close pvc
- execlp("grep")

Running: `ls -r | grep pdf`

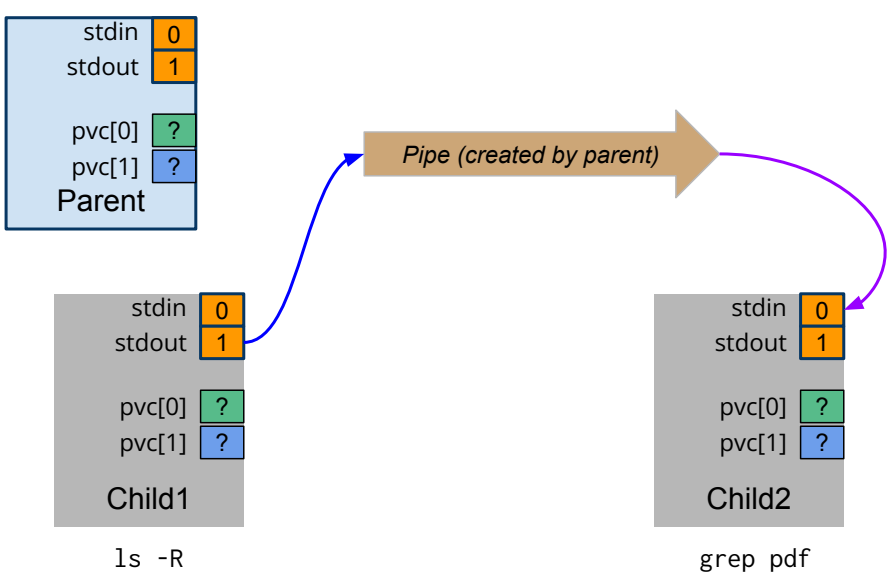


Parent:
- Create pipe
- fork() 2x
- close pvc

Child1
- dup(pvc[1], 1)
- close pvc
- execlp("ls")

Child2
- dup(pvc[0], 0);
- **close pvc**
- execlp("grep")

Shell pipe: `ls -R | grep pdf`

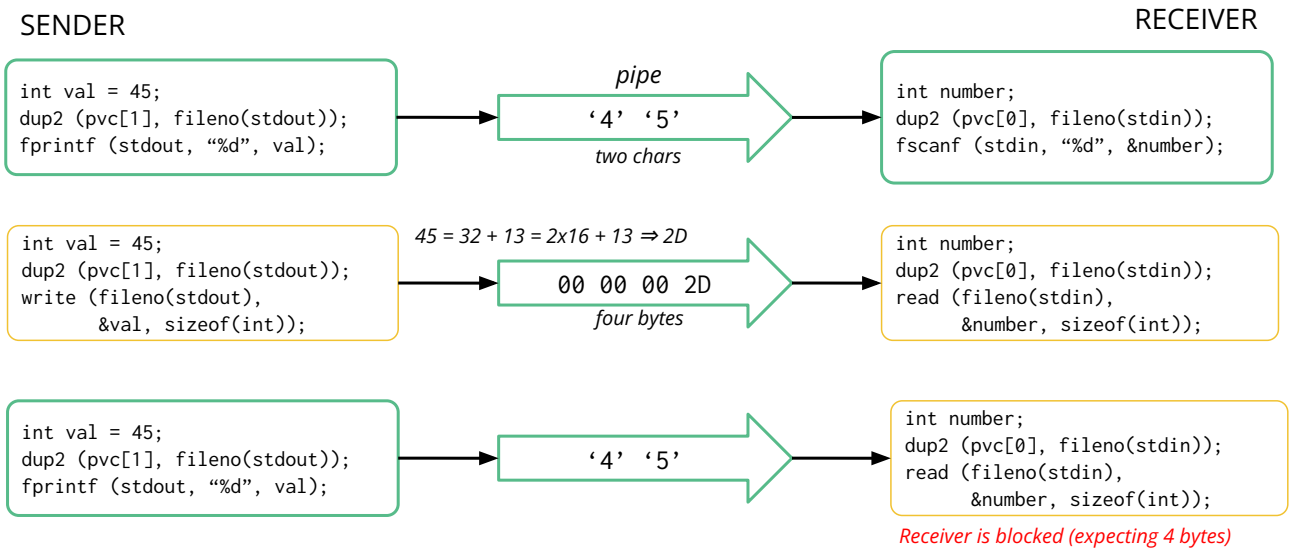


Parent:
- Create pipe
- fork() 2x
- close pvc

Child1
- dup(pvc[1], 1)
- close pvc
- **execlp("ls")**

Child2
- dup(pvc[0], 0);
- close pvc
- **execlp("grep")**

Text vs. Binary



Two-Way Communication with Pipes?

Pipe Buffers

- Each pipe is associated with a *limited size* buffer
  - OSX: 16K bytes
  - Since Linux 2.6.11: 64K bytes
- An attempt to read() with *insufficient amount of data* in the pipe **blocks** the caller
- An attempt to write() to a “full” pipe **blocks or fails** the caller
  - It fails when the descriptor was open with O_NONBLOCK flag
  - It blocks otherwise

Unix Pipes: one-directional ***synchronous*** communication (with payload) between parent/child

Unix Signals: one-directional ***asynchronous*** *dataless* communication between any processes

Unix Pipes vs. Unix Signals

- One directional form of **parent-child** communication
- Pipe buffers allow data transfer (either binary or text) between parent-child
  - Sender invokes `write()`
  - Recipient invokes `read()`
- Data is received **synchronously** by the recipient

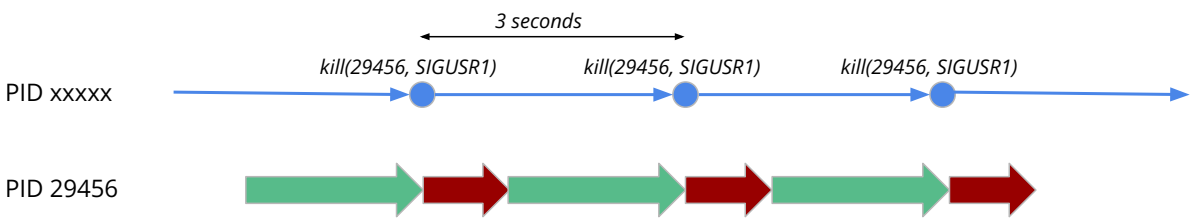
- One directional form of communication between processes (**not limited to parent-child**)
- **Dataless** communication
- Software equivalent of hardware interrupt
  - Sender invokes `kill()`
  - Recipient invokes `signal()` to setup signal handler
- Signal is received **asynchronously** by the recipient

PID xxxxx

```
// Signal sender
#include <signal.h>

int main() {
  while (1) {
    sleep(3);
    kill (29456, SIGUSR1);
  }
  return 0;
}
```

PID 29456

```
#include <signal.h>
void u1_handler() {
  printf ("Received SIGUSR1 signal");
}

void main() {
  signal (SIGUSR1, u1_handler);
  while(1) {
    // do something
  }
  return 0;
}
```



Signal and System Calls

- When a signal handler is invoked while a system call is **blocked**, the system call may
  - Restart automatically after the signal handler returns OR
  - Terminate with an error code EINTR
- Read the details under the ERRORS section of the syscall man page
  - Example: scanf()

System V signal()

- Linux provides System V (“five”) `signal()` behavior
  - After a signal is delivered and received by its handler function, the signal disposition is restored to its default behavior
- Refer to `man 7 signal` for details of signal disposition