CS143A
Principles on Operating Systems
Discussion 08:

Instructor: Prof. Anton Burtsev

TA: Saehanseul Yi (Hans)

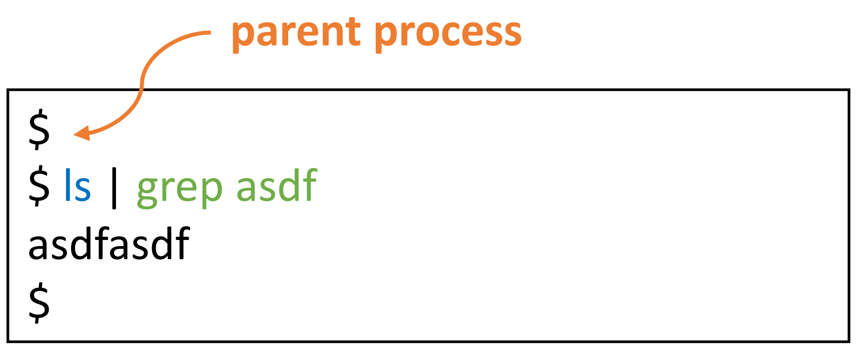
Nov 22, 2019 **Noon**

Agenda

- pipe() and fork(): visualization
- How to debug a user-program in xv6
- sh.c call structure

pipe() and fork()

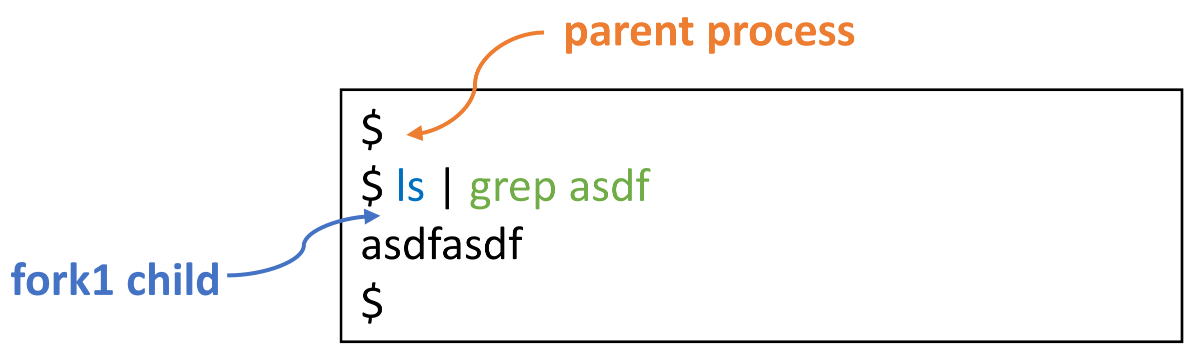
```
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
---------------------Point A--------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
---------------------Point B--------------------
   runcmd(pcmd>left);
}
if(fork1() == 0){
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   runcmd(pcmd>right);
}
close(p[0]);
close(p[1]);
---------------------Point C--------------------
wait();
wait();
break;
```



```
$
$ ls | grep asdf
asdfasdf
$
```

pipe() and fork()

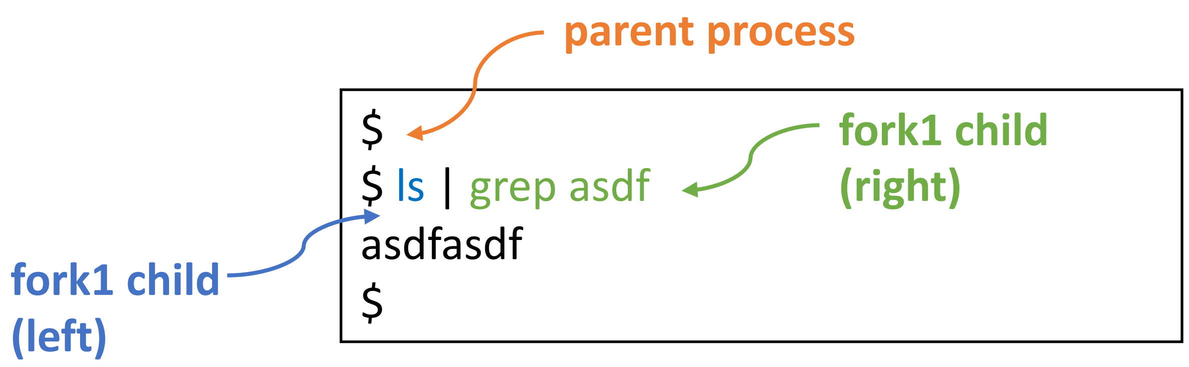
```
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
---------------------Point A--------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
---------------------Point B--------------------
   runcmd(pcmd>left);
}
if(fork1() == 0){
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   runcmd(pcmd>right);
}
close(p[0]);
close(p[1]);
---------------------Point C--------------------
wait();
wait();
break;
```



```
$
$ ls | grep asdf
asdfasdf
$
```

pipe() and fork()

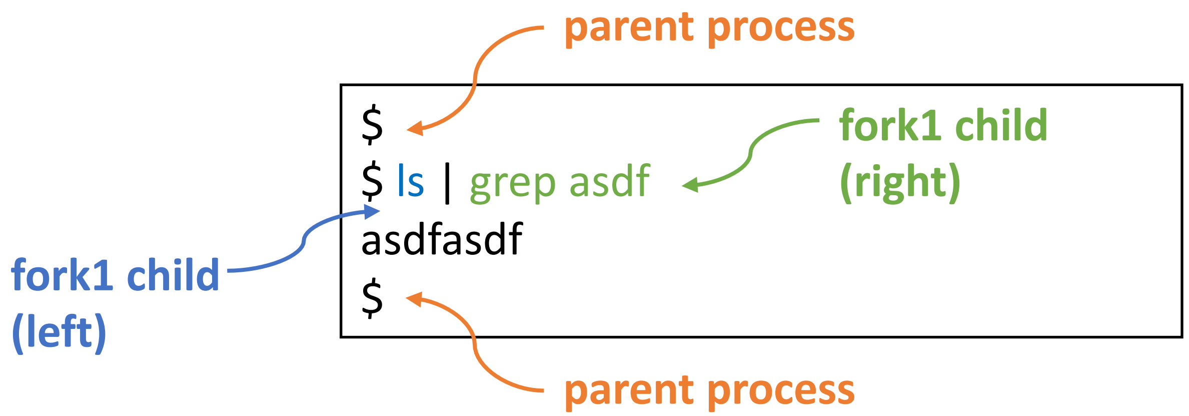
```
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
---------------------Point A--------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
---------------------Point B--------------------
   runcmd(pcmd>left);
}
if(fork1() == 0){
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   runcmd(pcmd>right);
}
close(p[0]);
close(p[1]);
---------------------Point C--------------------
wait();
wait();
break;
```



```
$
$ ls | grep asdf
asdfasdf
$
```

pipe() and fork()

```
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
---------------------Point A--------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
---------------------Point B--------------------
   runcmd(pcmd>left);
}
if(fork1() == 0){
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   runcmd(pcmd>right);
}
close(p[0]);
close(p[1]);
---------------------Point C--------------------
wait();
wait();
break;
```

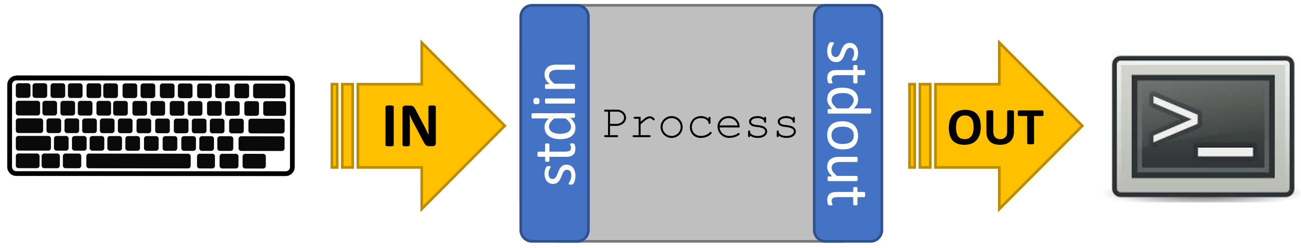


```
$
$ ls | grep asdf
asdfasdf
$
```

Wait… stdin? stdout?

(standard input, standard output)

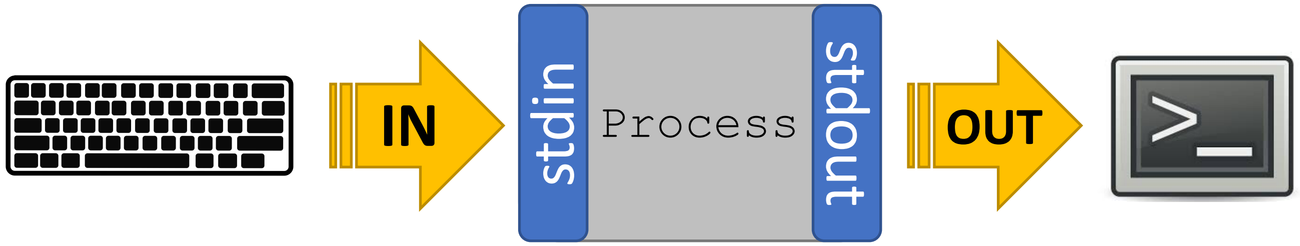
```
$
$ ls | grep asdf
asdfasdf
$
```

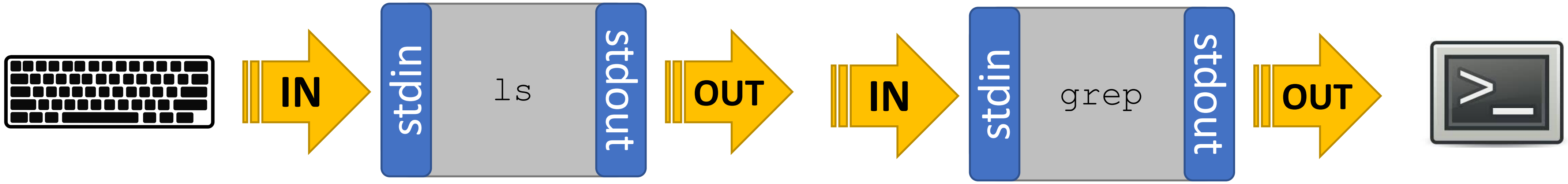


Wait… stdin? stdout?

(standard input, standard output)

```
$
$ ls | grep asdf
asdfasdf
$
```

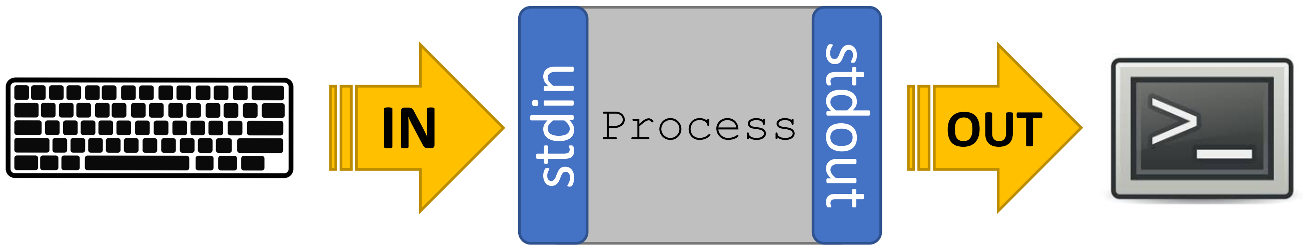


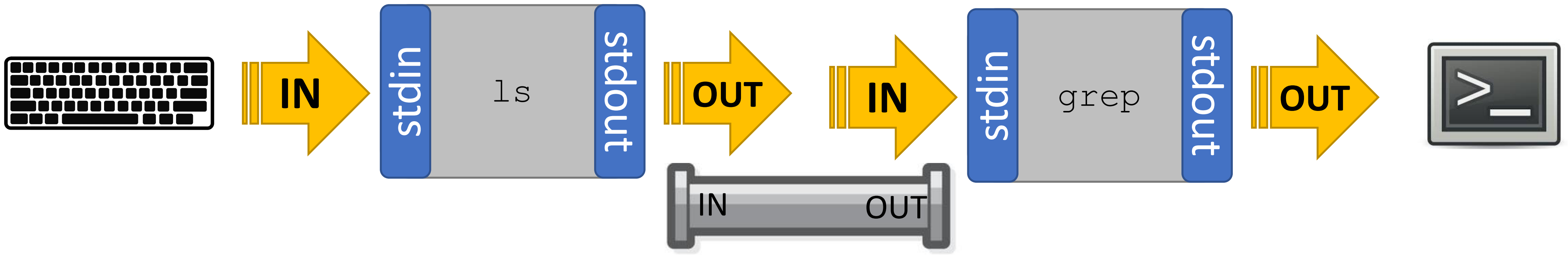


Wait… stdin? stdout?

(standard input, standard output)

```
$
$ ls | grep asdf
asdfasdf
$
```



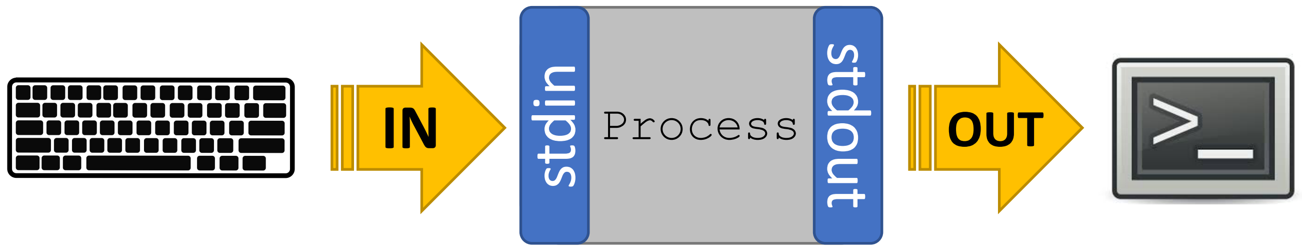


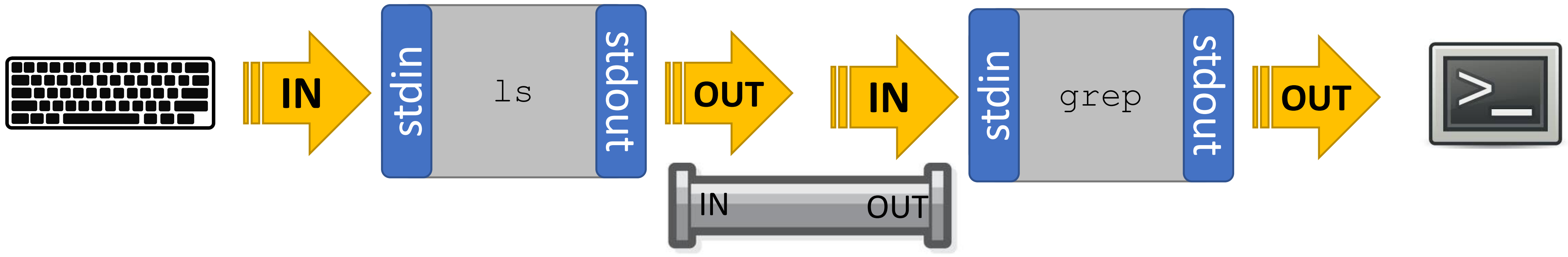
Wait… stdin? stdout?

(standard input, standard output)

```
$
$ ls | grep asdf
asdfasdf
$
```

- stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. **just an integer** for user-program)



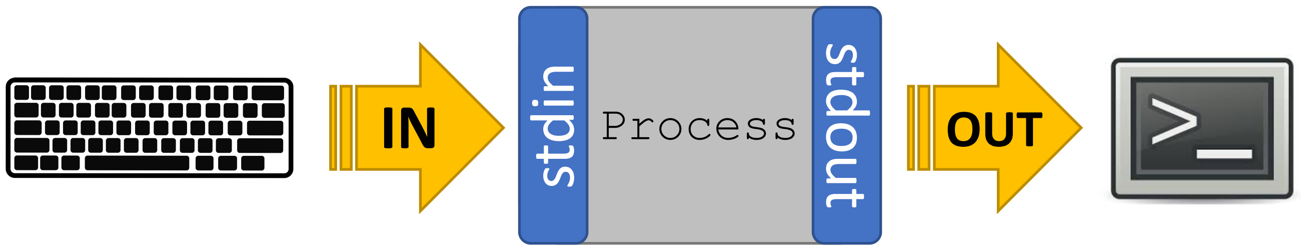


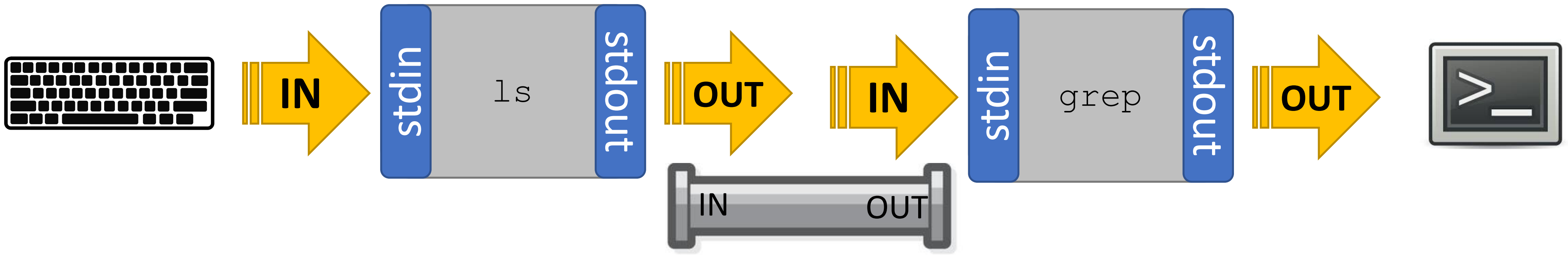
Wait… stdin? stdout?

(standard input, standard output)

```
$
$ ls | grep asdf
asdfasdf
$
```

- stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. **just an integer** for user-program)
- Each program has its own descriptor array(?) (e.g. A's stdin is 0 and B's stdin is 0 as well)



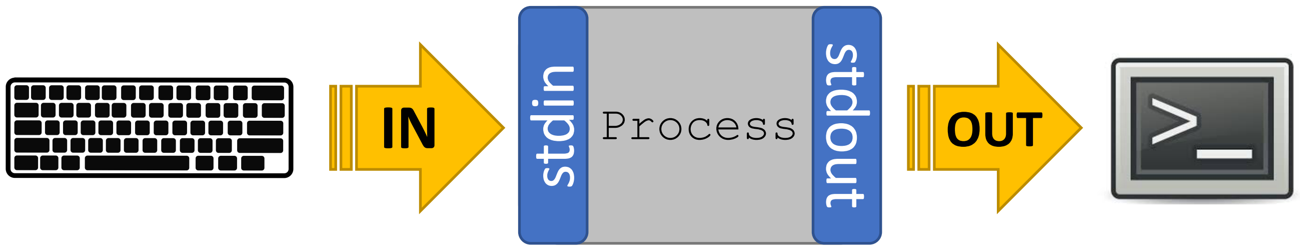


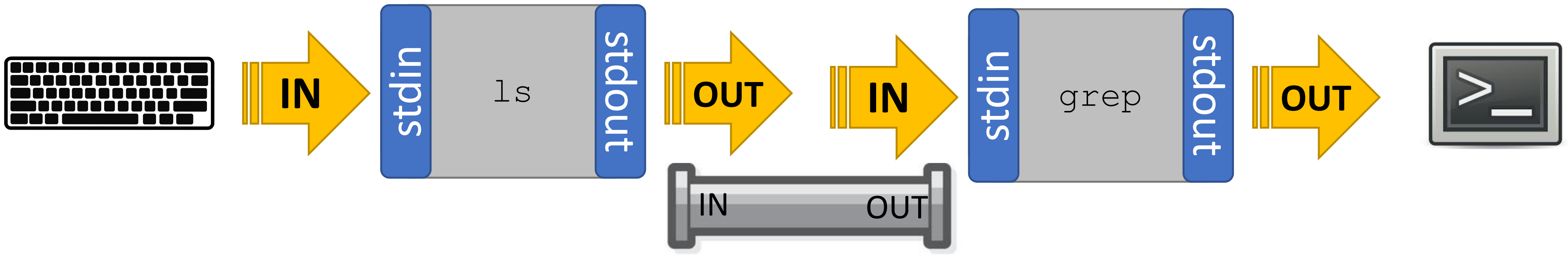
Wait… stdin? stdout?

(standard input, standard output)

```
$
$ ls | grep asdf
asdfasdf
$
```

- stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. **just an integer** for user-program)
- Each program has its own descriptor array(?) (e.g. A's stdin is 0 and B's stdin is 0 as well)
- How to modify process' file descriptors?



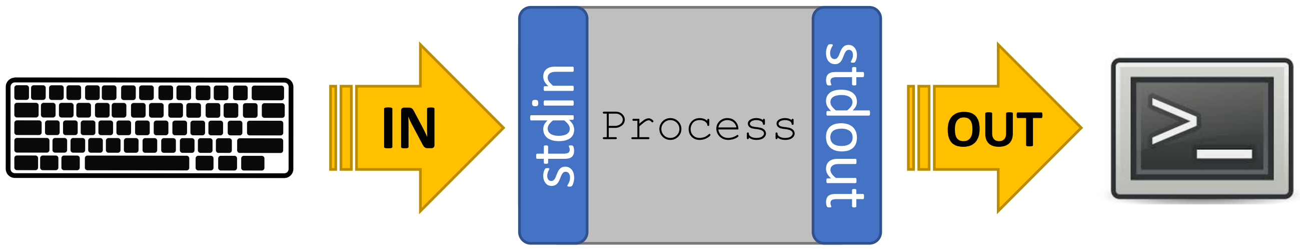


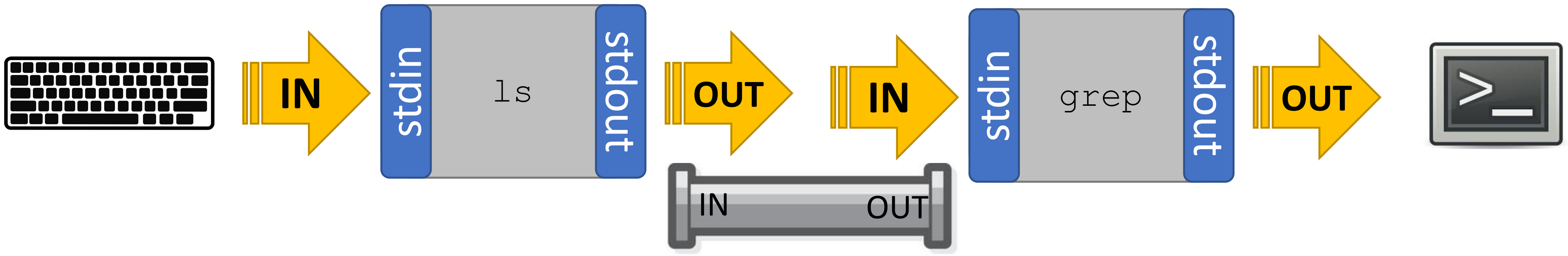
Wait… stdin? stdout?

(standard input, standard output)

```
$
$ ls | grep asdf
asdfasdf
$
```

- stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. **just an integer** for user-program)
- Each program has its own descriptor array(?) (e.g. A's stdin is 0 and B's stdin is 0 as well)
- How to modify process' file descriptors?
  - close, dup(or open)



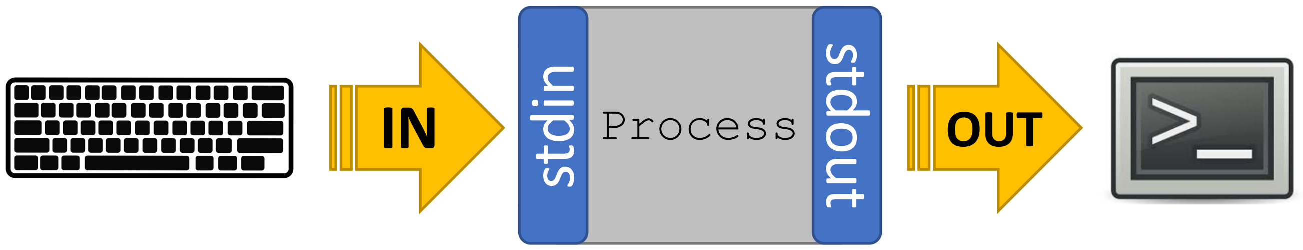


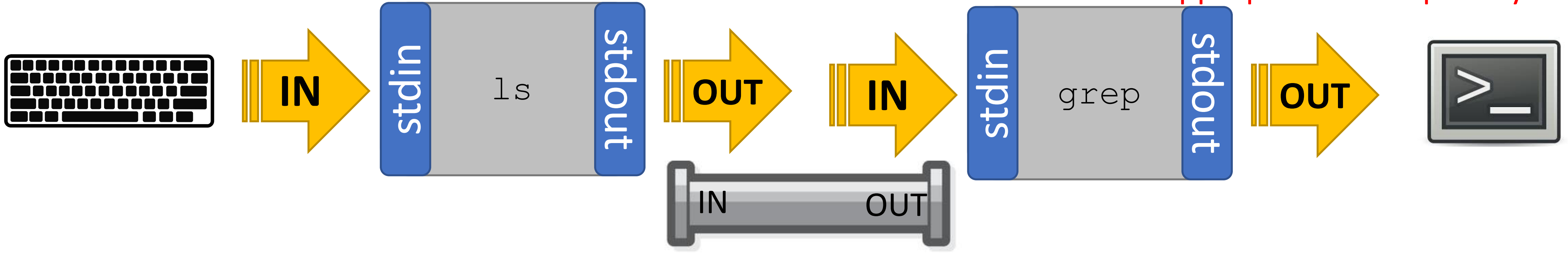
Wait… stdin? stdout?

(standard input, standard output)

```
$
$ ls | grep asdf
asdfasdf
$
```

- stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. **just an integer** for user-program)
- Each program has its own descriptor array(?) (e.g. A's stdin is 0 and B's stdin is 0 as well)
- How to modify process' file descriptors?
  - close, dup(or open)
- What we need to do:
  close appropriate descriptors for each process and set the appropriate descriptor by copying



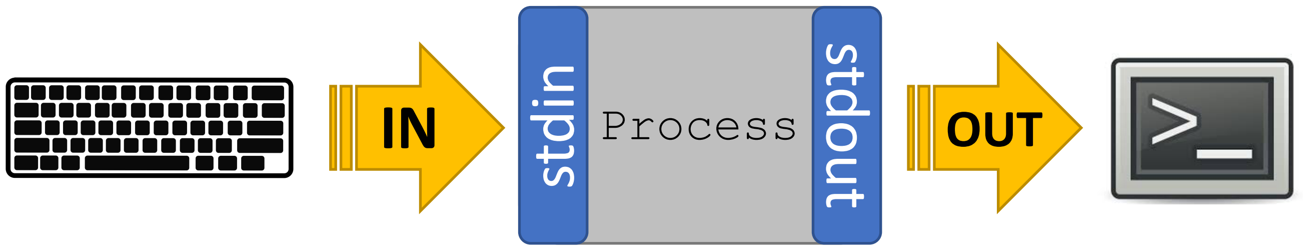


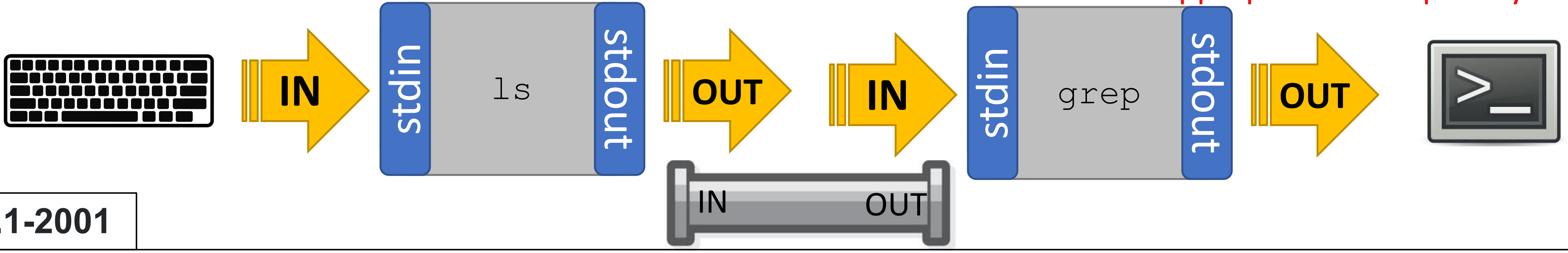
Wait… stdin? stdout?

(standard input, standard output)

```
$
$ ls | grep asdf
asdfasdf
$
```

- stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. **just an integer** for user-program)
- Each program has its own descriptor array(?) (e.g. A's stdin is 0 and B's stdin is 0 as well)
- How to modify process' file descriptors?
  - close, dup(or open)
- What we need to do: close appropriate descriptors for each process and set the appropriate descriptor by copying





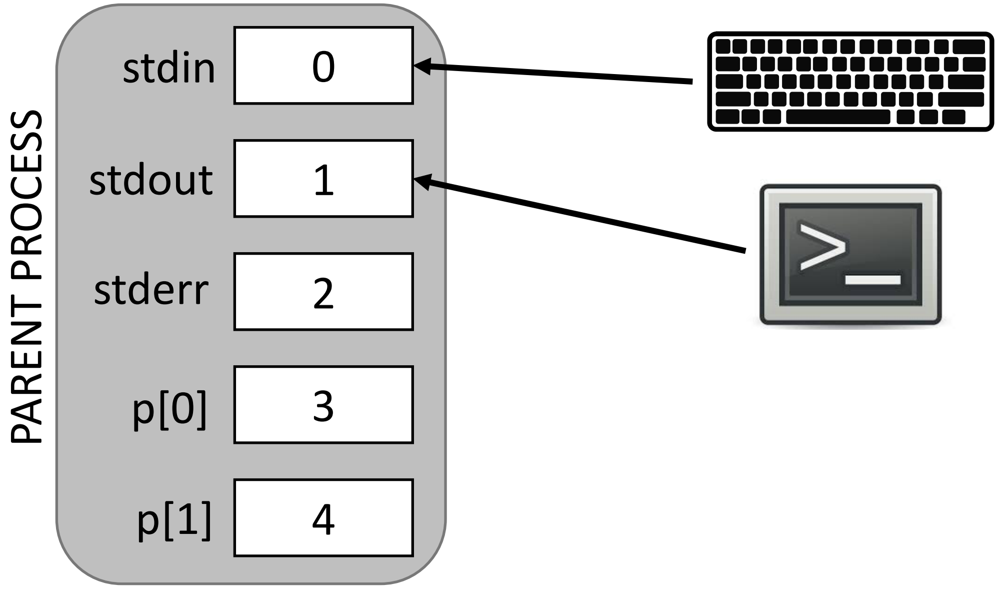
POSIX.1-2001

pipe() creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing. **pipe is uni-directional**

pipe() and fork()

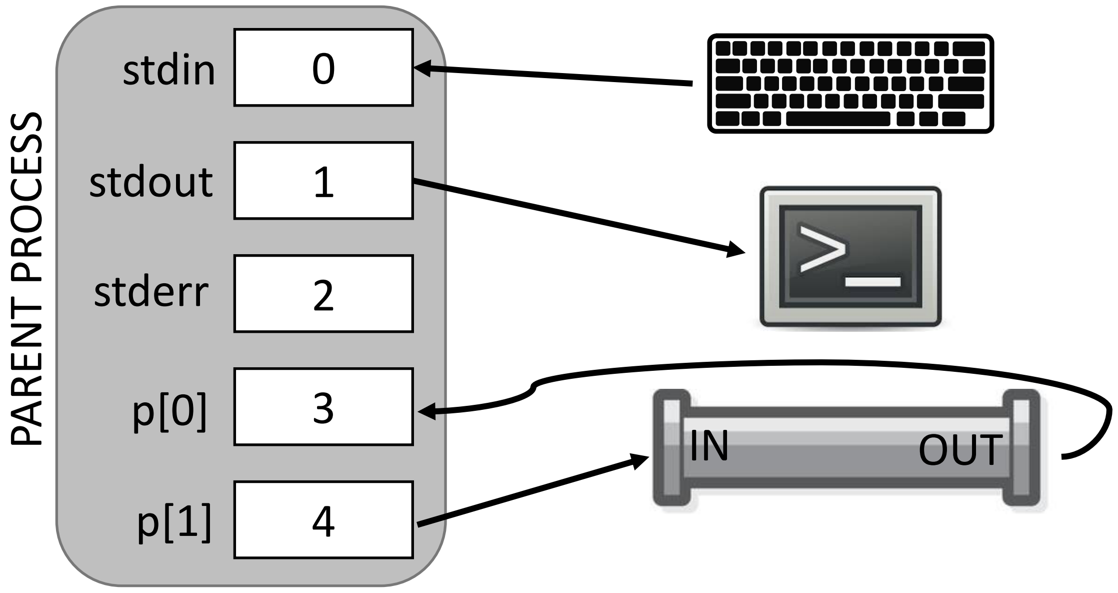
※ Throughout the example, stderr is always connected to the screen. Omitted for simplicity as well as p[0] and p[1] to the parent process

```
---------------------Point 0--------------------
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
---------------------Point A--------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
---------------------Point B--------------------
   runcmd(pcmd>left);
}
```



pipe() and fork()

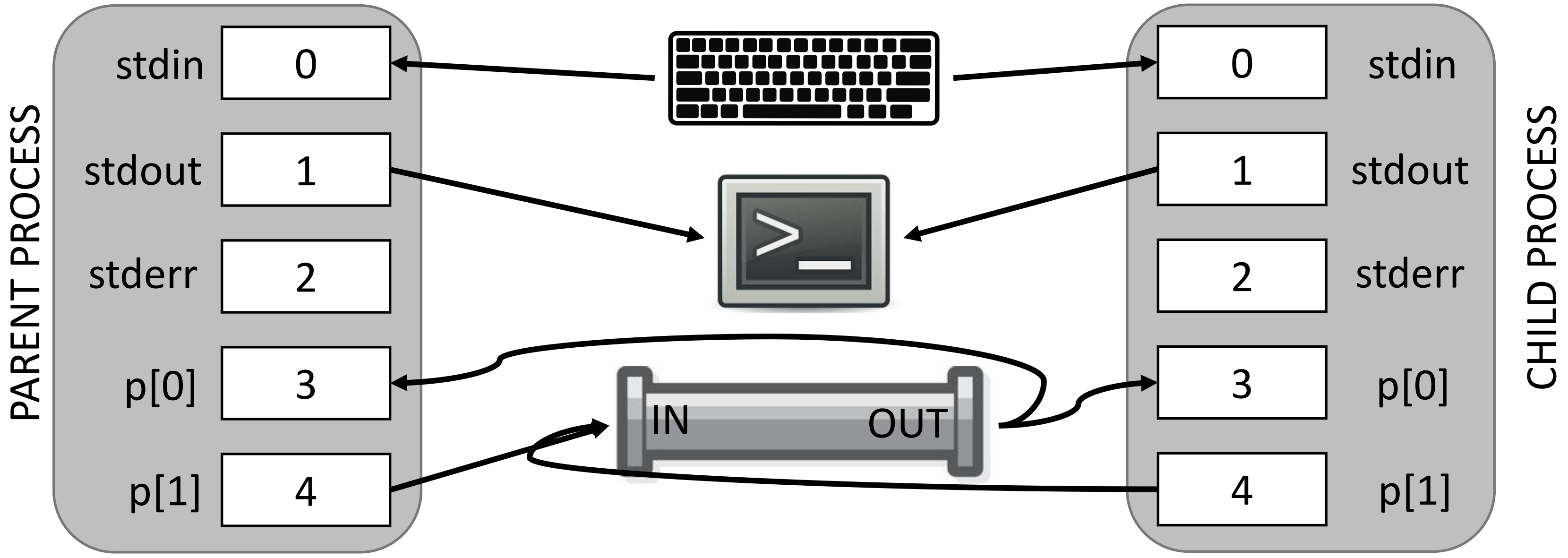
```
---------------------Point 0--------------------
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
---------------------Point A-------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
---------------------Point B--------------------
   runcmd(pcmd>left);
}
```



pipe() and fork()

```
--------------------Point 0-------------------
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
--------------------Point A------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
--------------------Point B-------------------
   runcmd(pcmd>left);
}
```

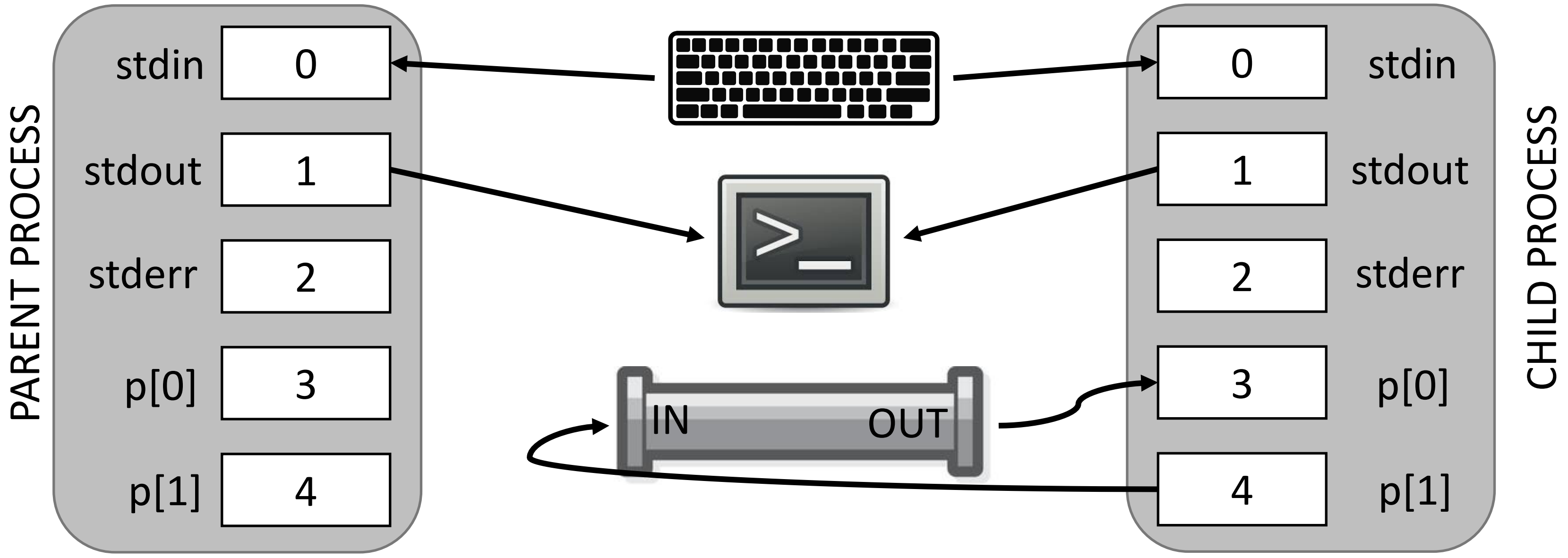
fork() copies the descriptors too!

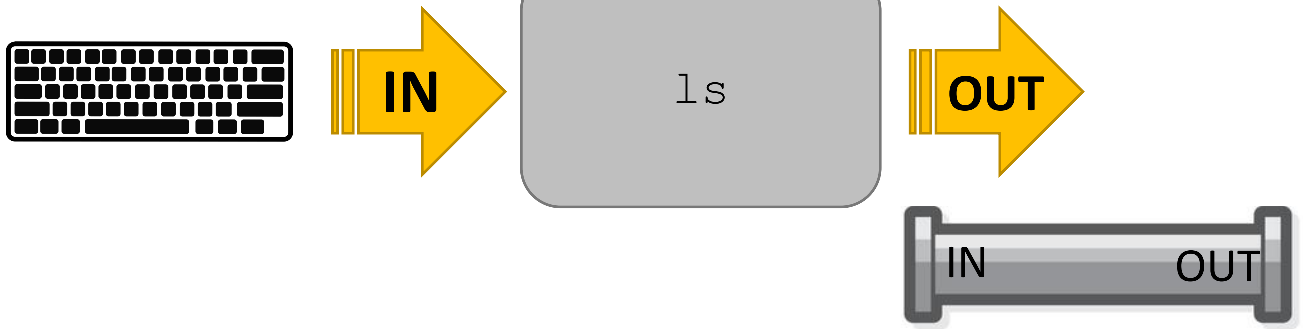


pipe() and fork()

```
--------------------Point 0-------------------
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
--------------------Point A------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
--------------------Point B-------------------
   runcmd(pcmd>left);
}
```

fork() copies the descriptors too!

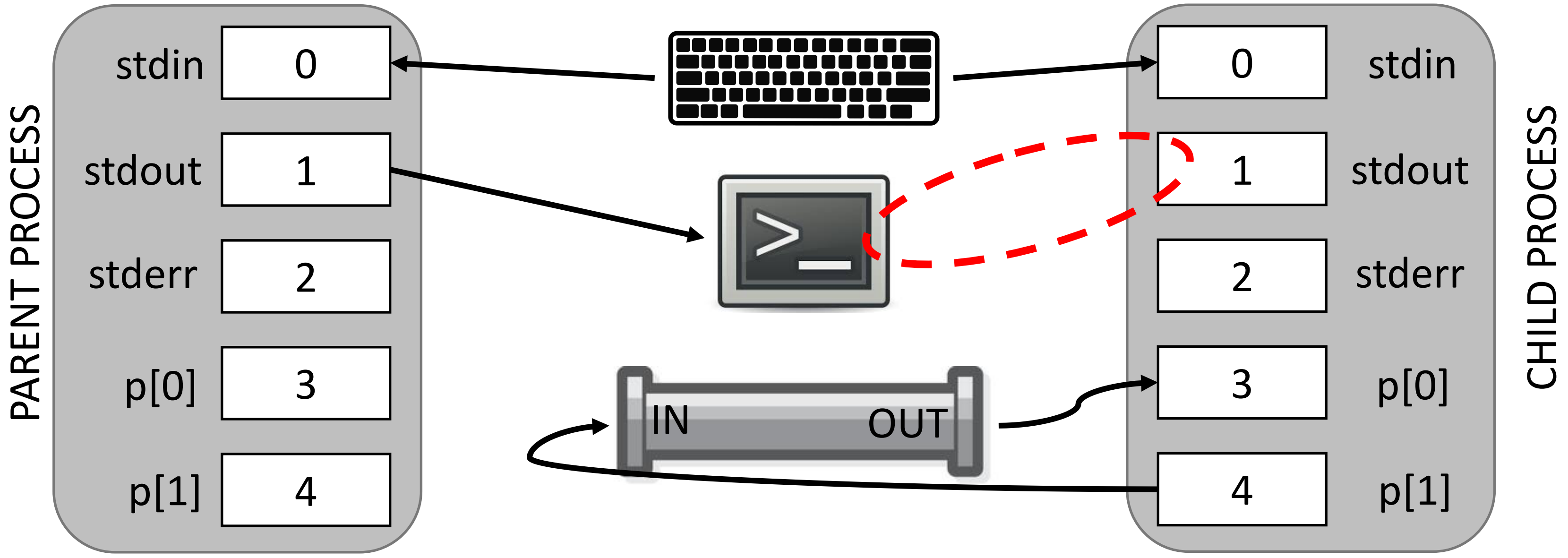


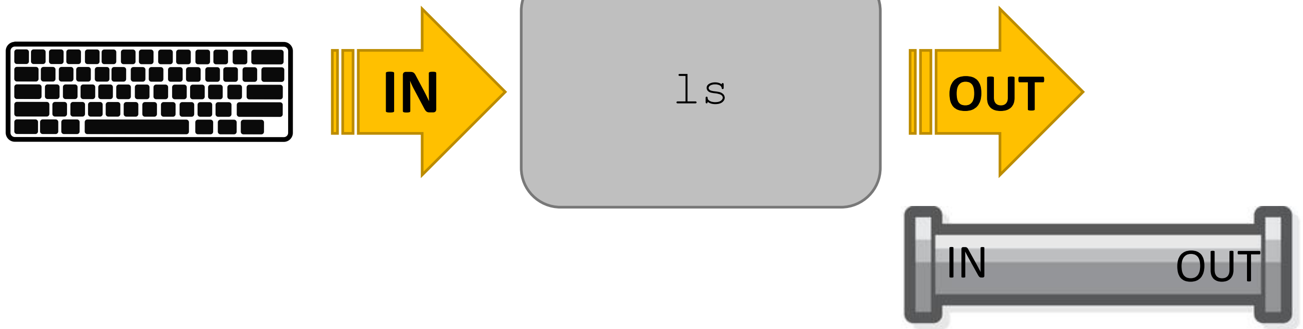


pipe() and fork()

```
--------------------Point 0-------------------
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
--------------------Point A------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
--------------------Point B-------------------
   runcmd(pcmd>left);
}
```

fork() copies the descriptors too!



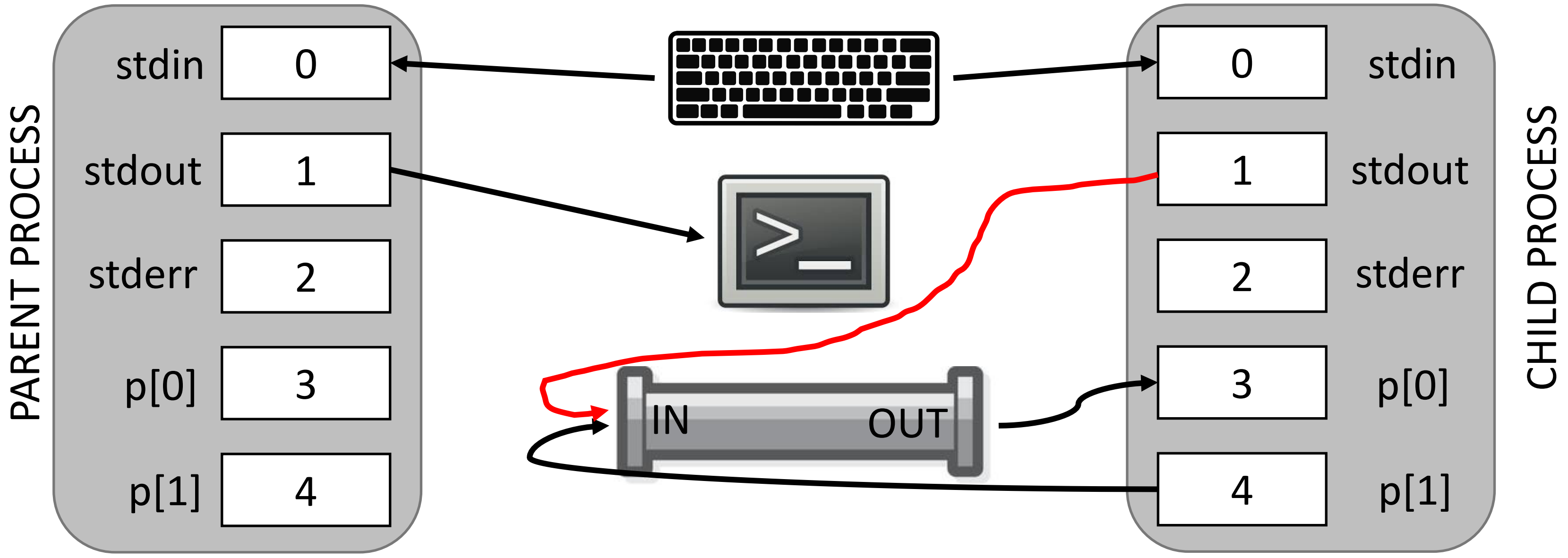


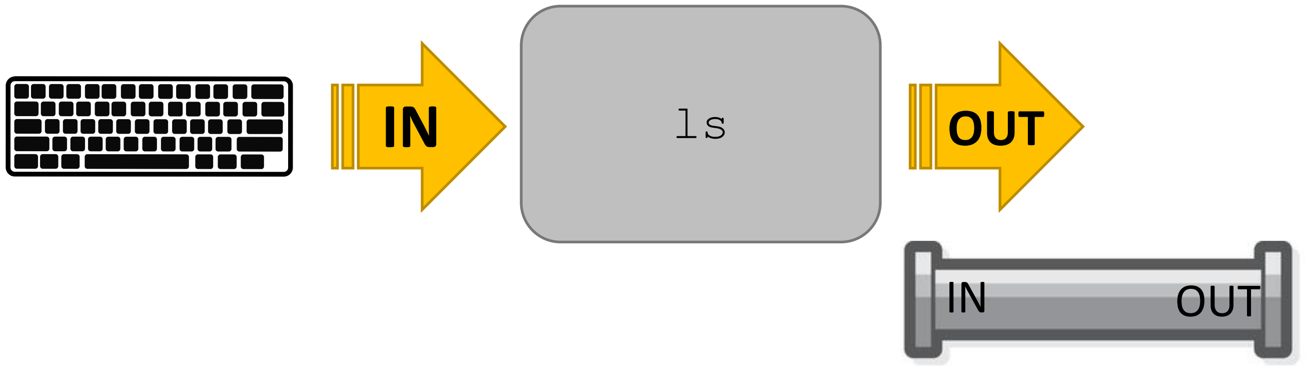
pipe() and fork()

```
--------------------Point 0-------------------
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
--------------------Point A------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
--------------------Point B-------------------
   runcmd(pcmd>left);
}
```

fork() copies the descriptors too!

dup()'s destination is the lowest & unused file descriptor!

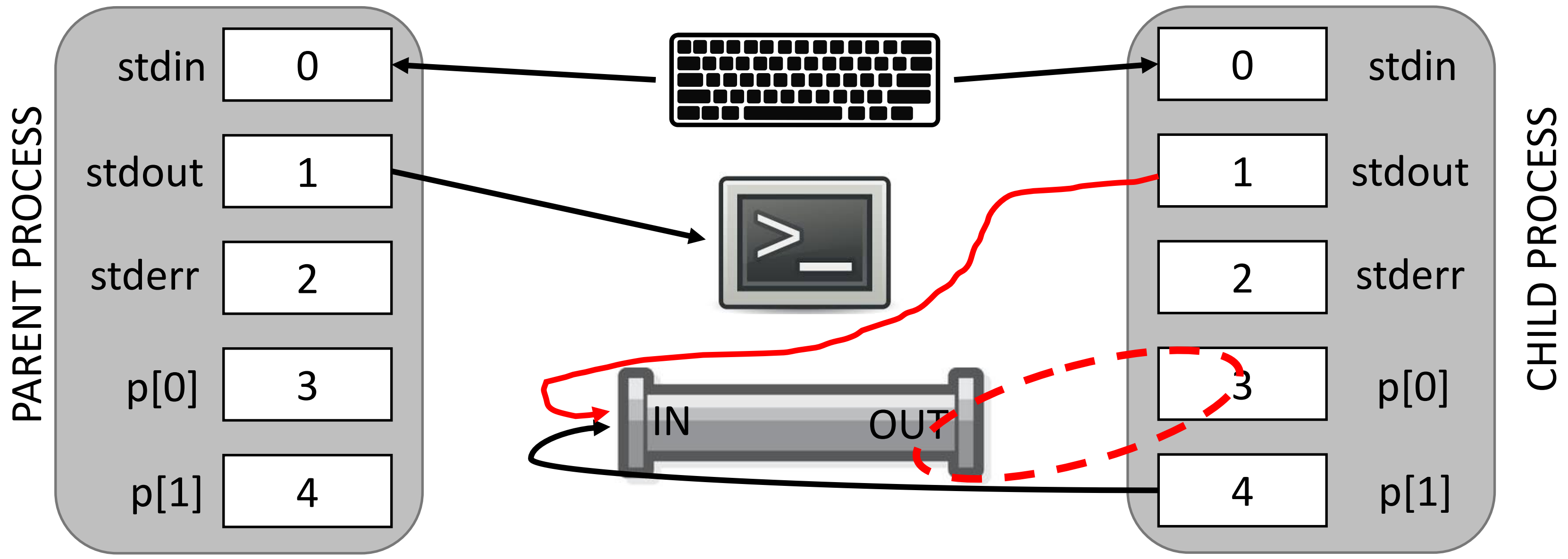


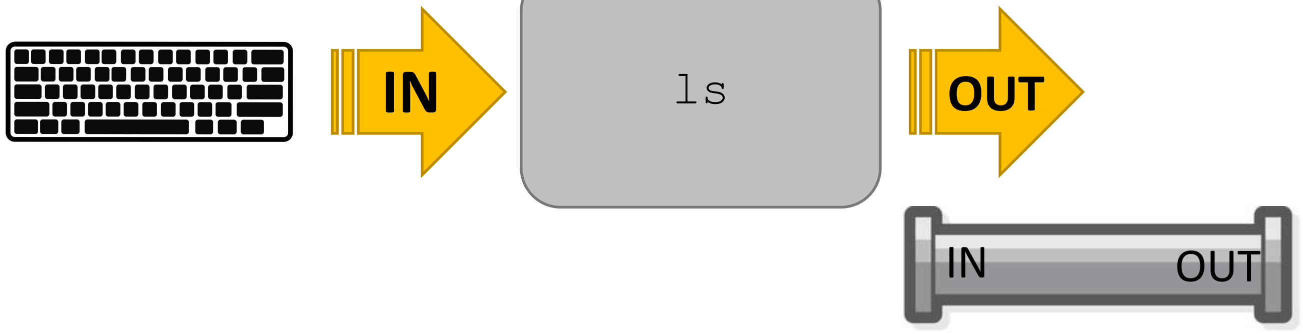


pipe() and fork()

```
--------------------Point 0-------------------
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
--------------------Point A------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
--------------------Point B-------------------
   runcmd(pcmd>left);
}
```

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

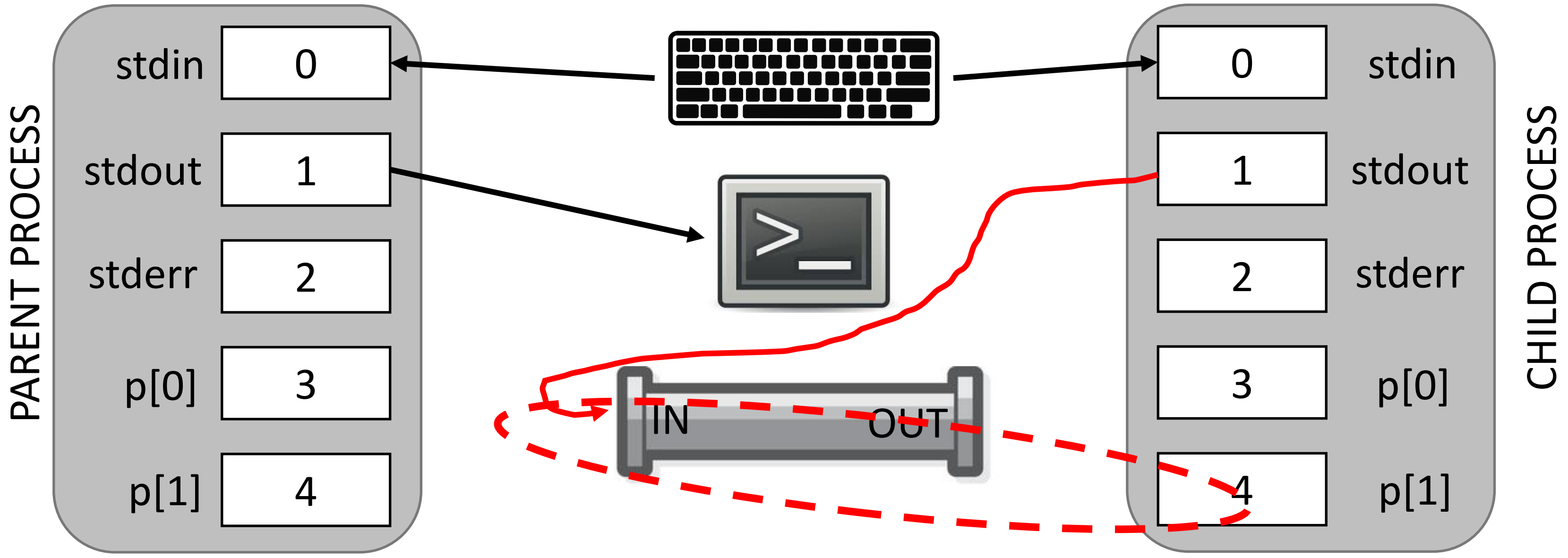


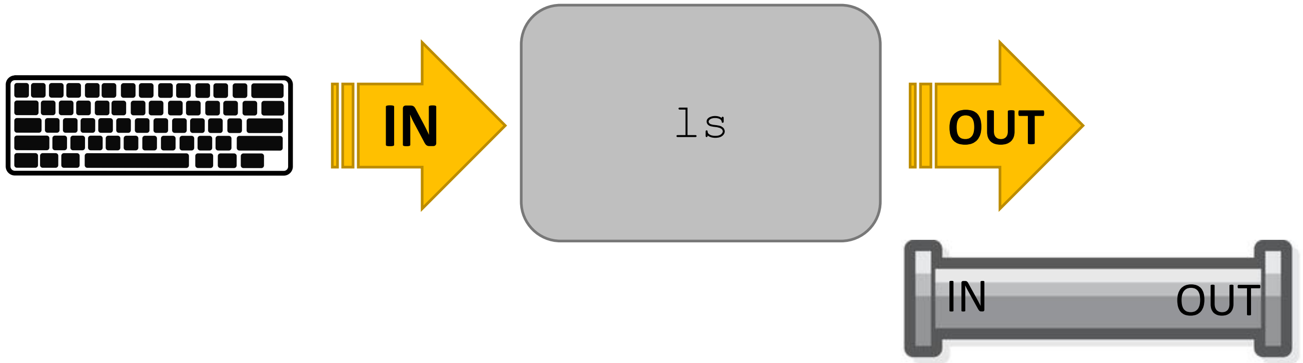


pipe() and fork()

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

```
--------------------Point 0-------------------
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
--------------------Point A------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
--------------------Point B-------------------
   runcmd(pcmd>left);
}
```

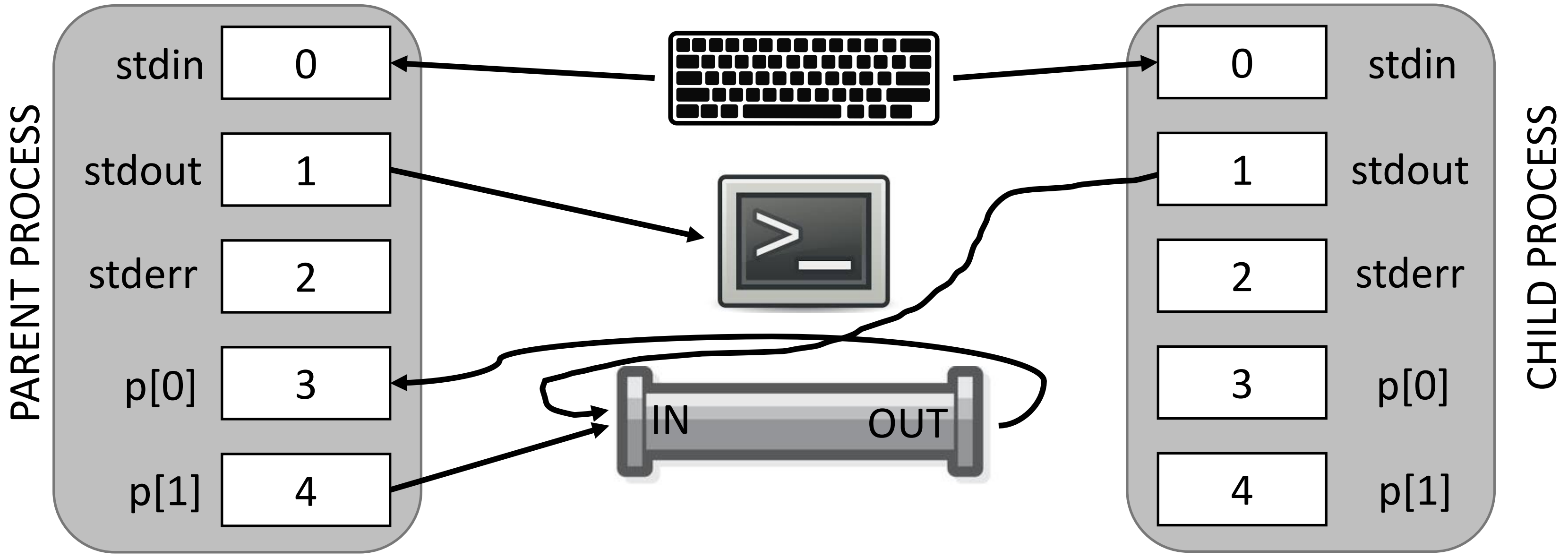




pipe() and fork()

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

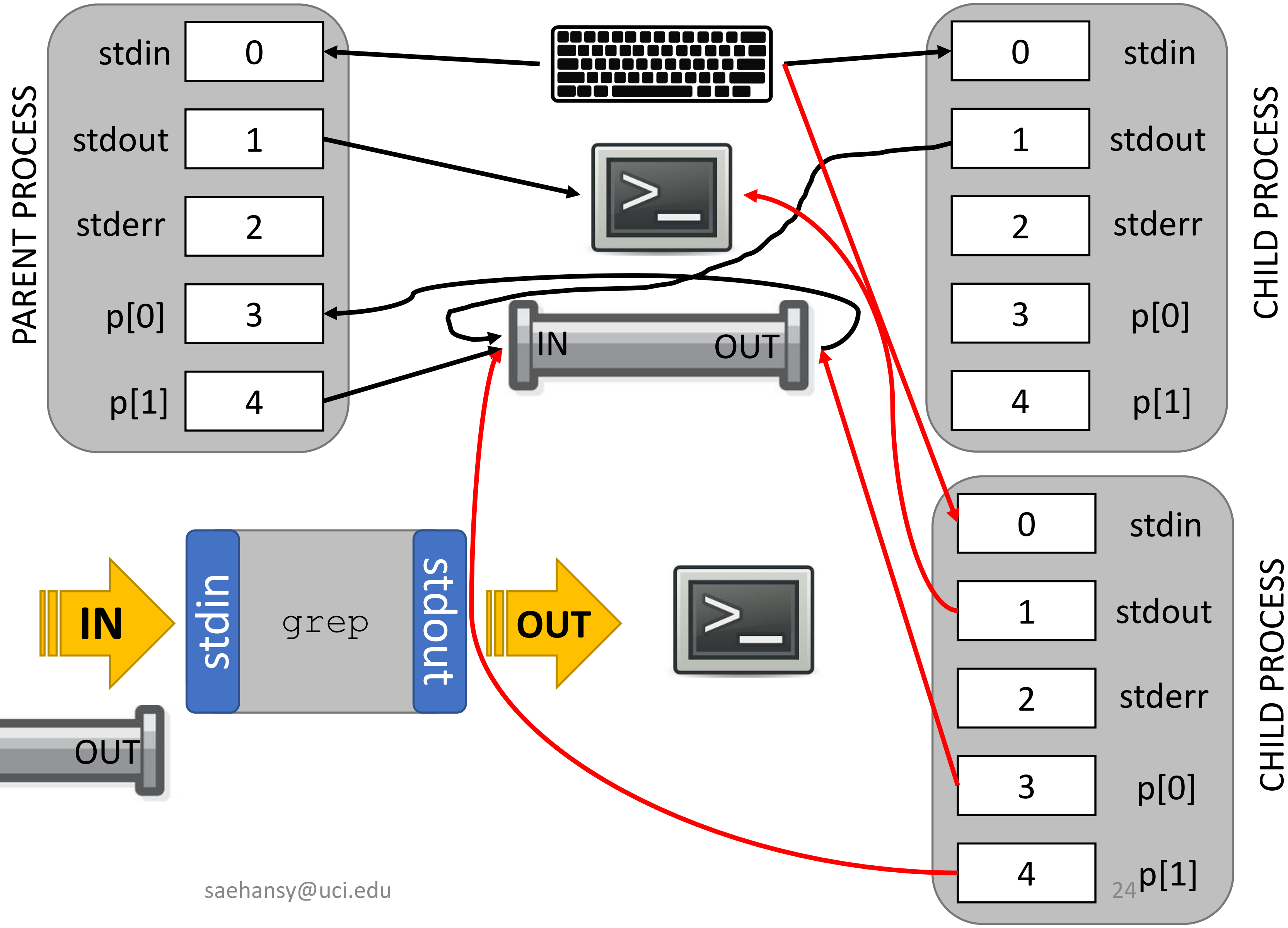
```
---------------------Point B--------------------
   runcmd(pcmd>left);
}
if(fork1() == 0){
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   runcmd(pcmd>right);
}
close(p[0]);
close(p[1]);
---------------------Point C--------------------
wait();
wait();
break;
```



pipe() and fork()

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

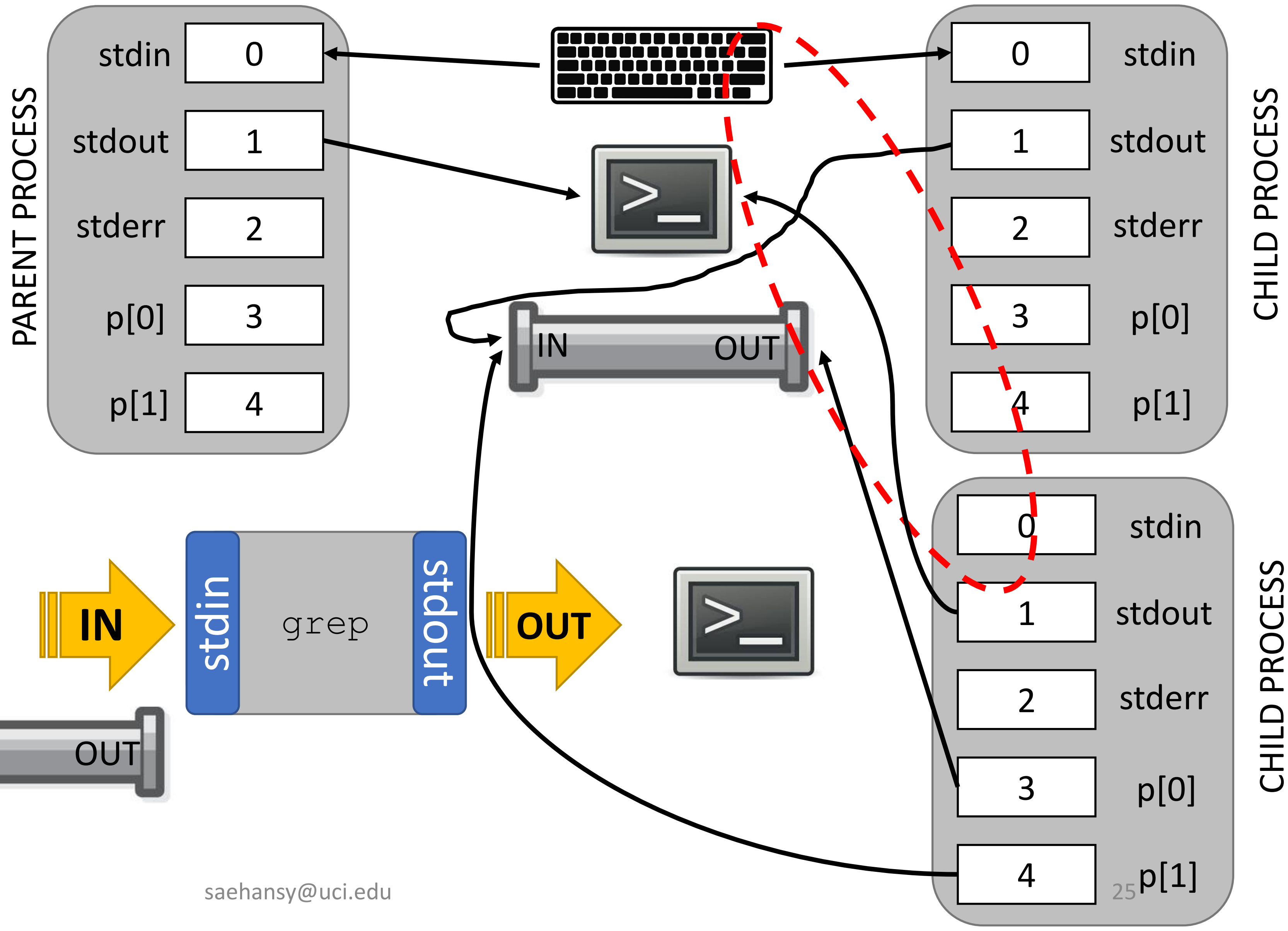
```
--------------------Point B--------------------
    runcmd(pcmd>left);
}
if(fork1() == 0){
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd>right);
}
close(p[0]);
close(p[1]);
--------------------Point C-----------
wait();
wait();
break;
```



fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

pipe() and fork()

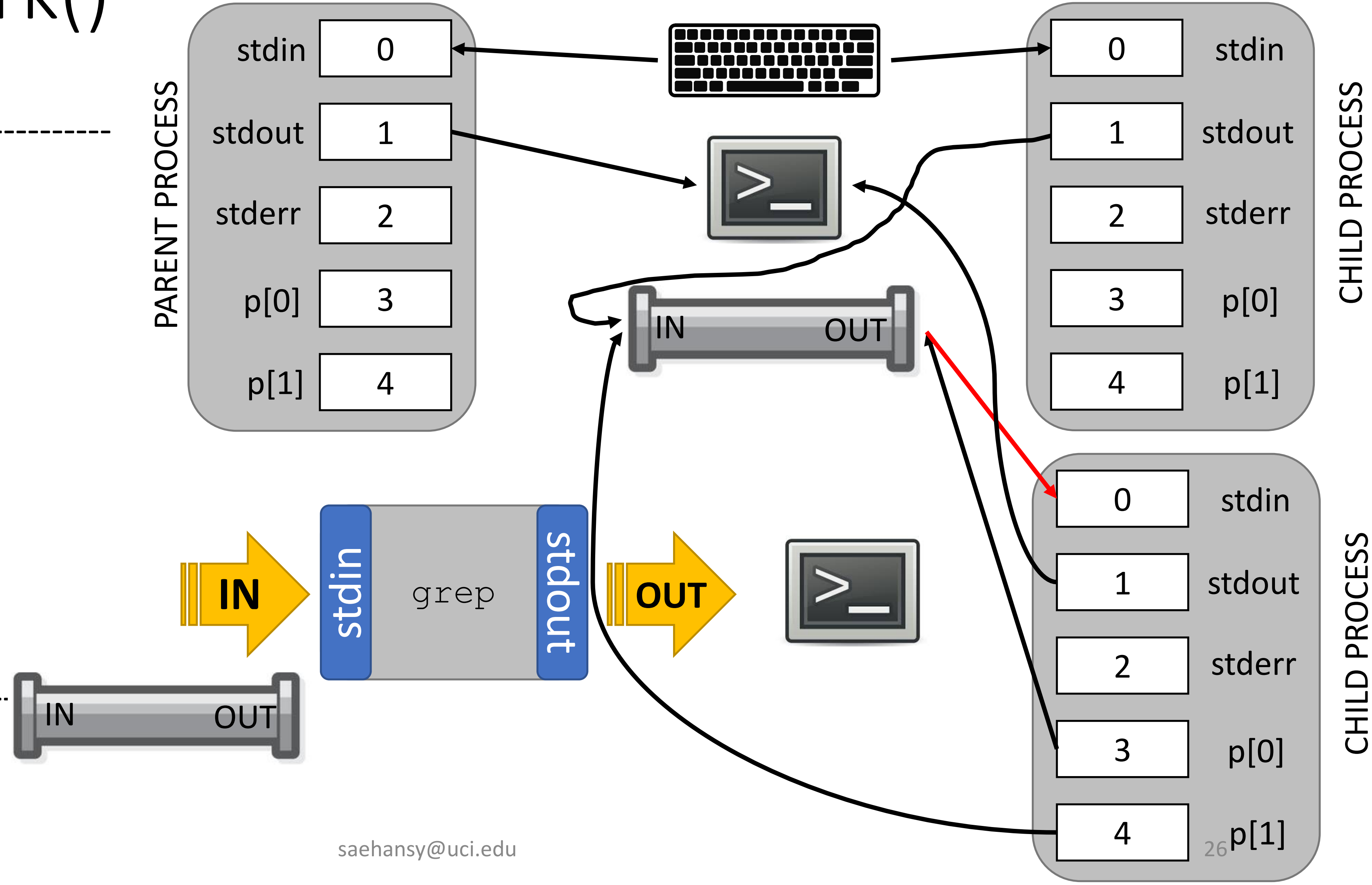
```
--------------------Point B--------------------
   runcmd(pcmd>left);
}
if(fork1() == 0){
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   runcmd(pcmd>right);
}
close(p[0]);
close(p[1]);
--------------------Point C-----------
wait();
wait();
break;
```



fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

pipe() and fork()

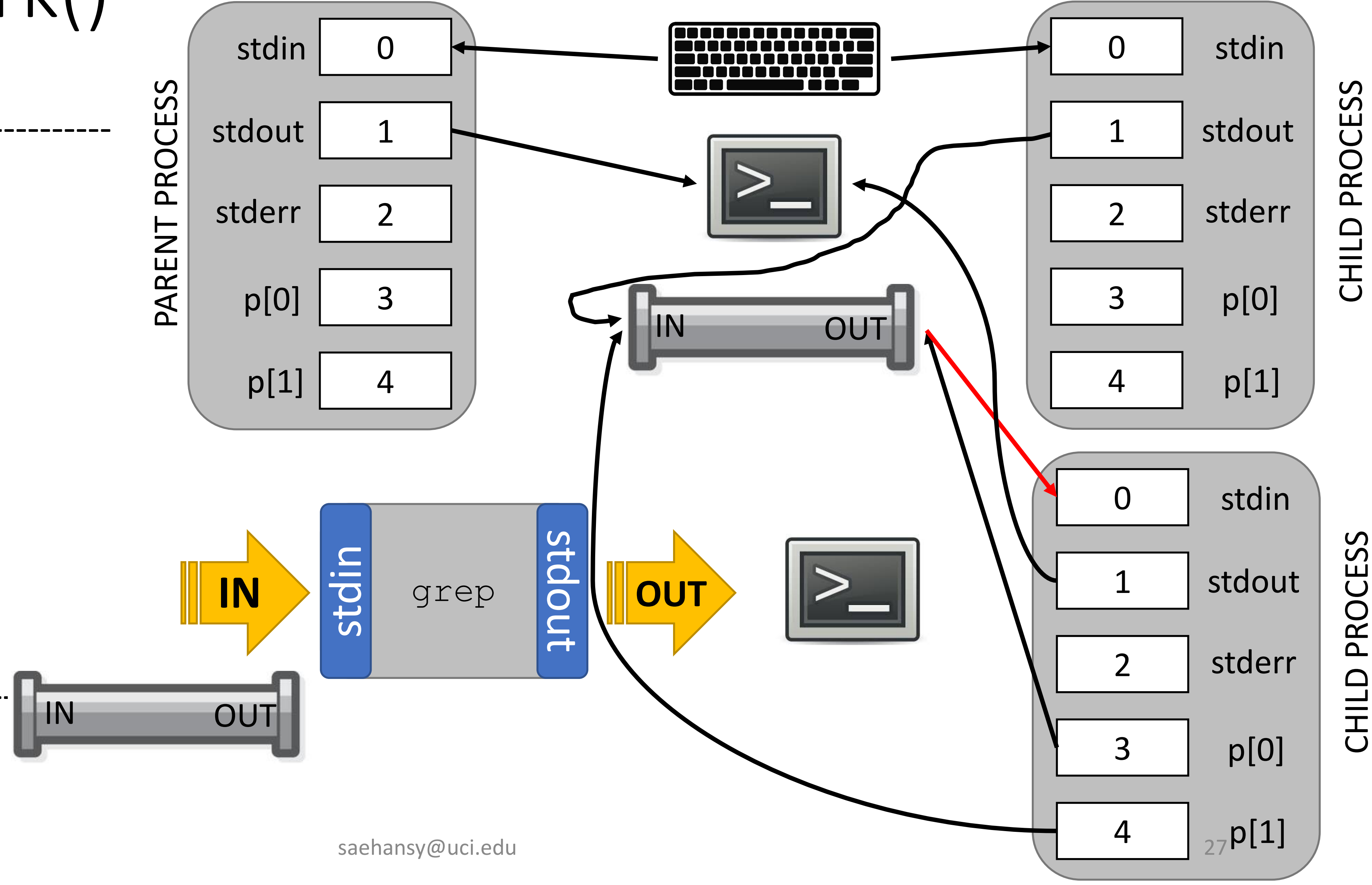
```
--------------------Point B-------------------
   runcmd(pcmd>left);
}
if(fork1() == 0){
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   runcmd(pcmd>right);
}
close(p[0]);
close(p[1]);
--------------------Point C-----------
wait();
wait();
break;
```



pipe() and fork()

fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

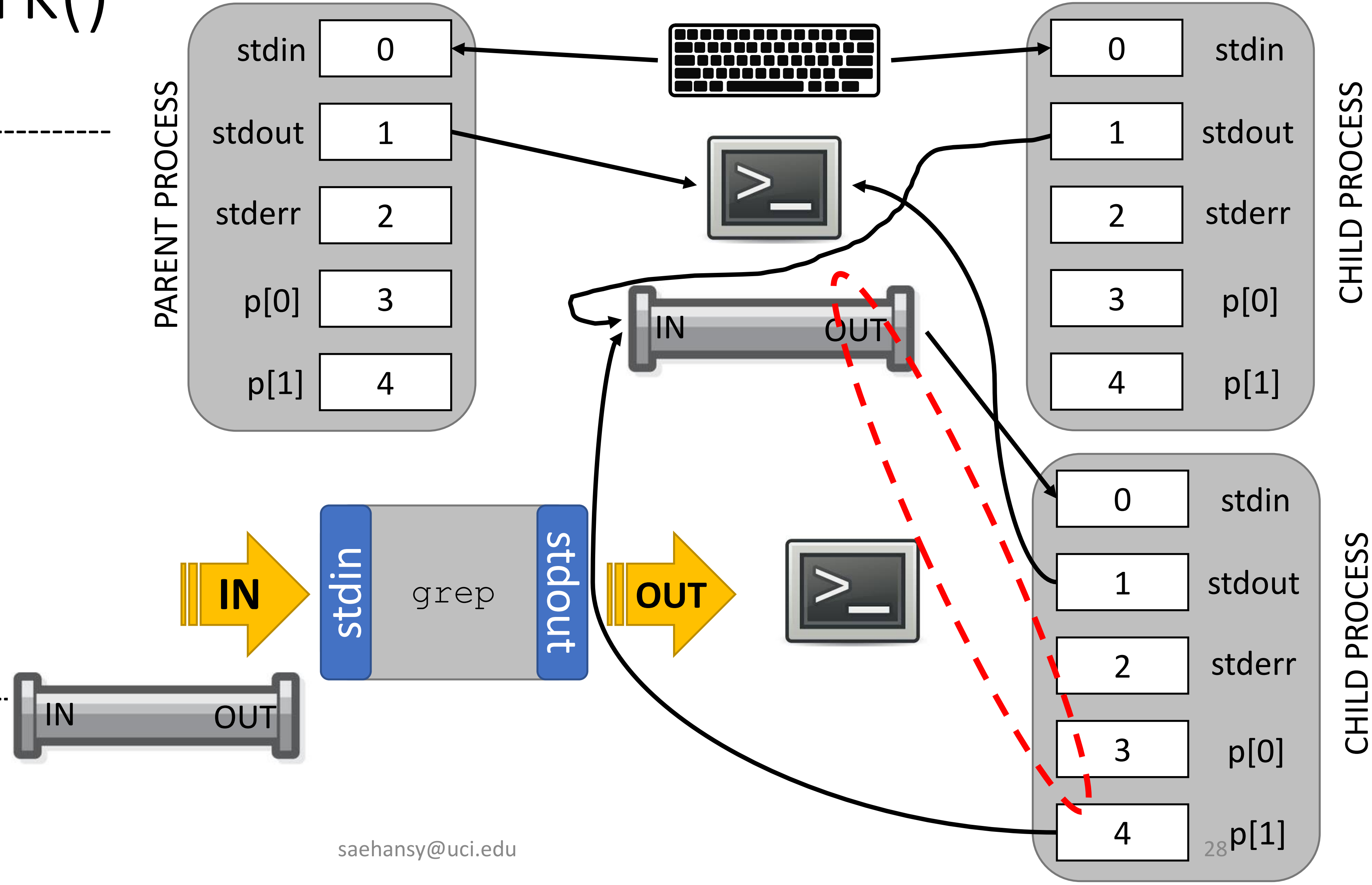
```
--------------------Point B--------------------
   runcmd(pcmd>left);
}
if(fork1() == 0){
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   runcmd(pcmd>right);
}
close(p[0]);
close(p[1]);
--------------------Point C-----------
wait();
wait();
break;
```



fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

pipe() and fork()

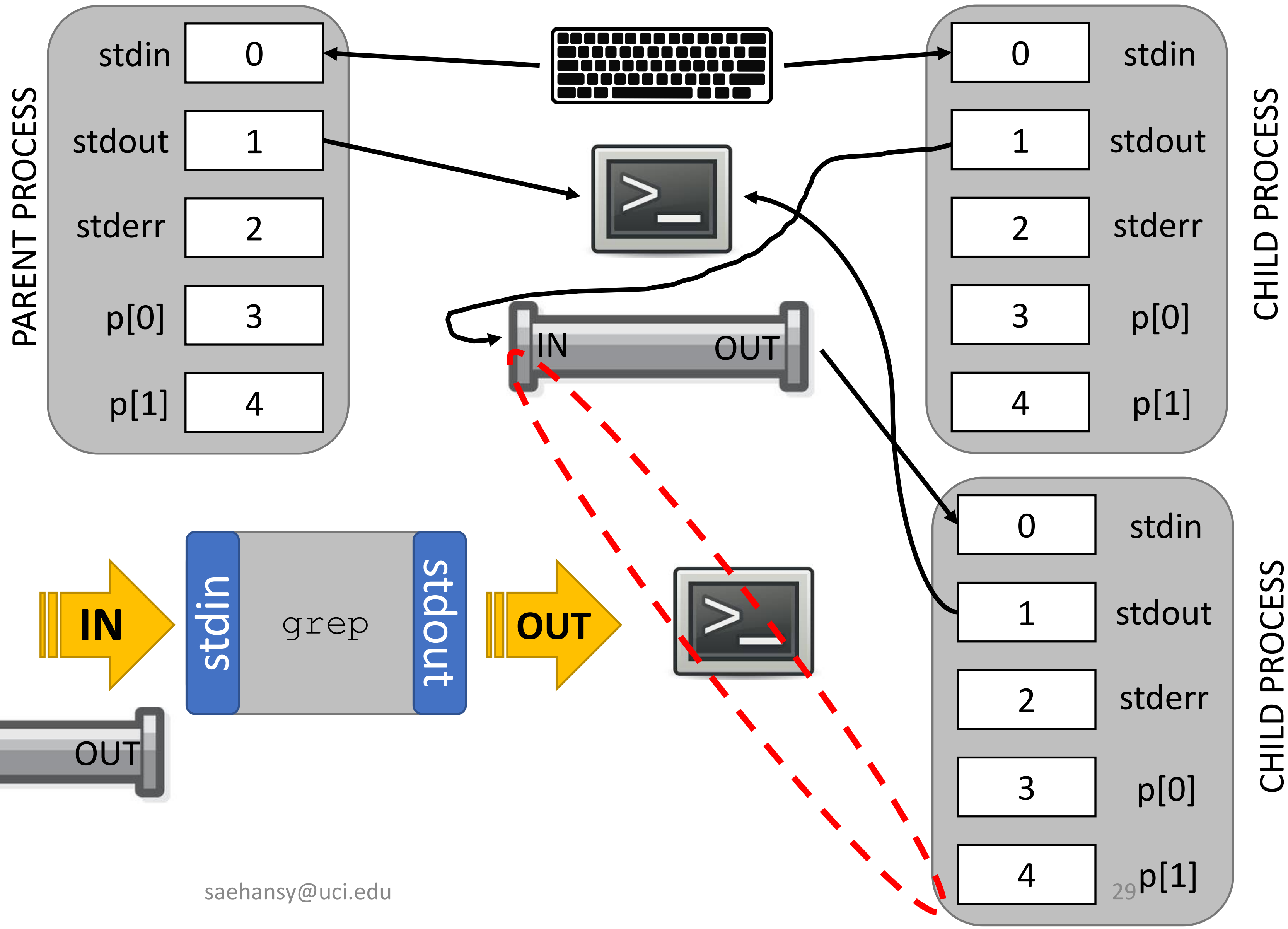
```
--------------------Point B-------------------
   runcmd(pcmd>left);
}
if(fork1() == 0){
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   runcmd(pcmd>right);
}
close(p[0]);
close(p[1]);
--------------------Point C-----------
wait();
wait();
break;
```



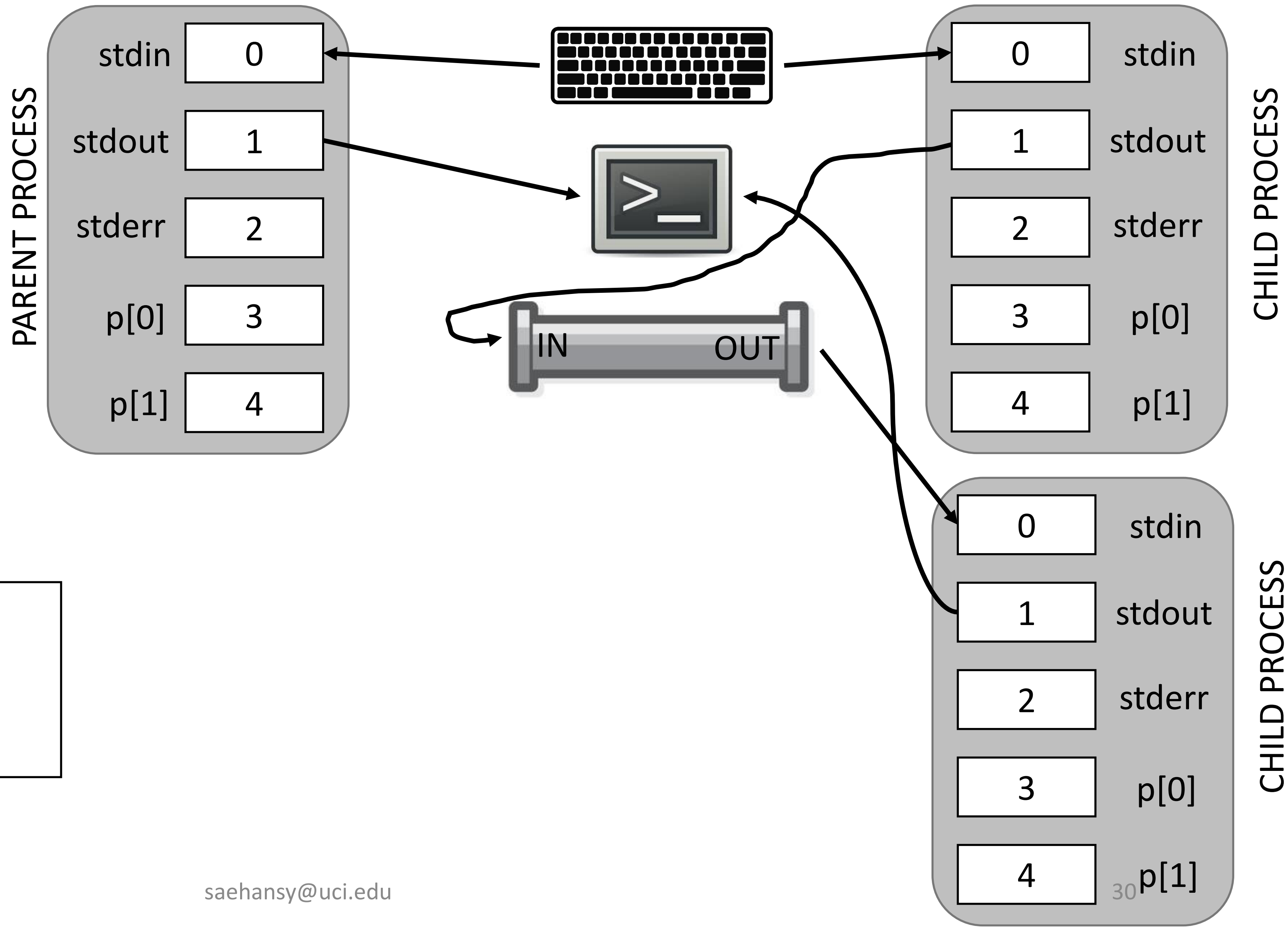
fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

pipe() and fork()

```
--------------------Point B--------------------
    runcmd(pcmd>left);
}
if(fork1() == 0){
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd>right);
}
close(p[0]);
close(p[1]);
--------------------Point C-----------
wait();
wait();
break;
```

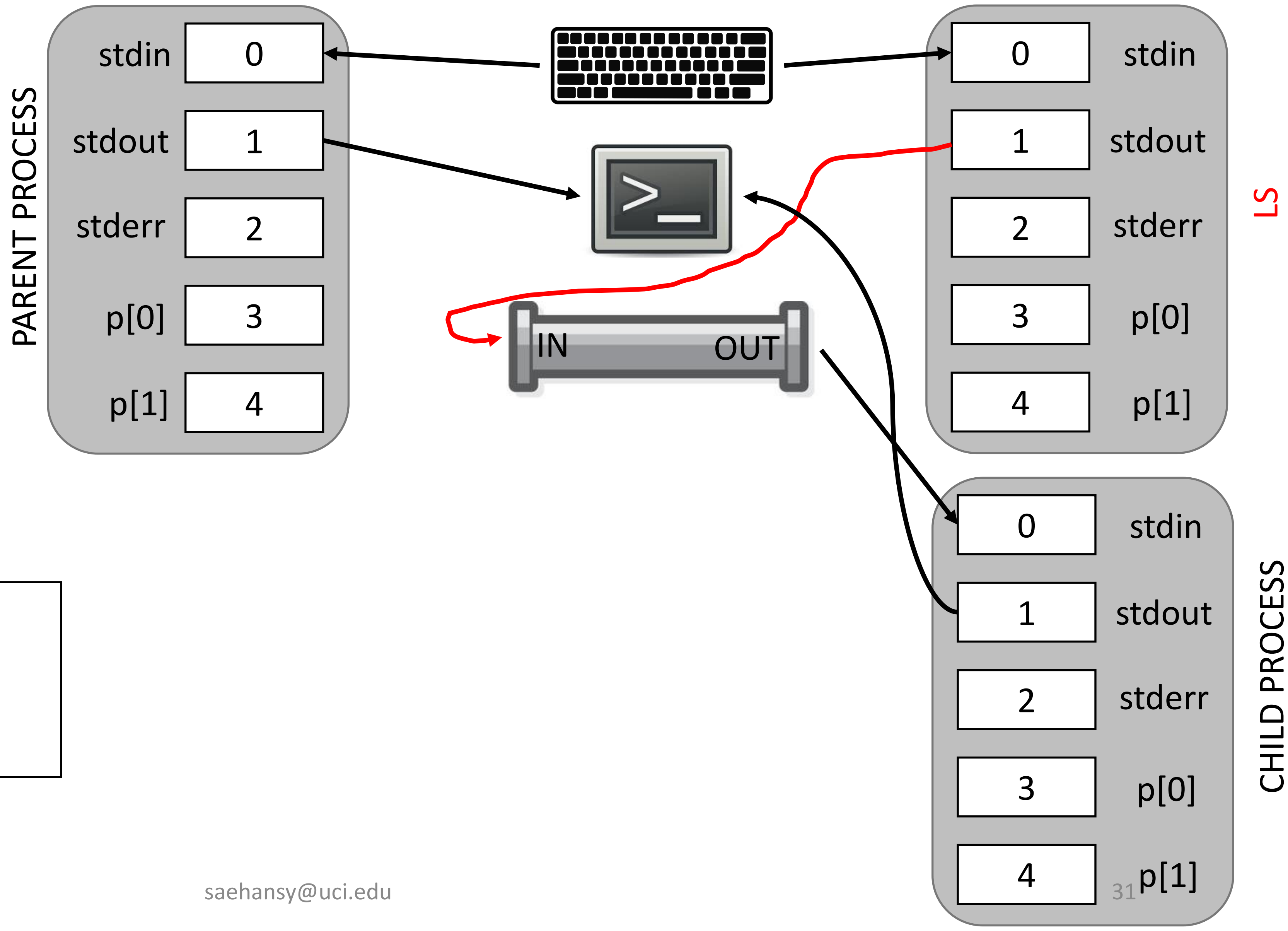


pipe() and fork()



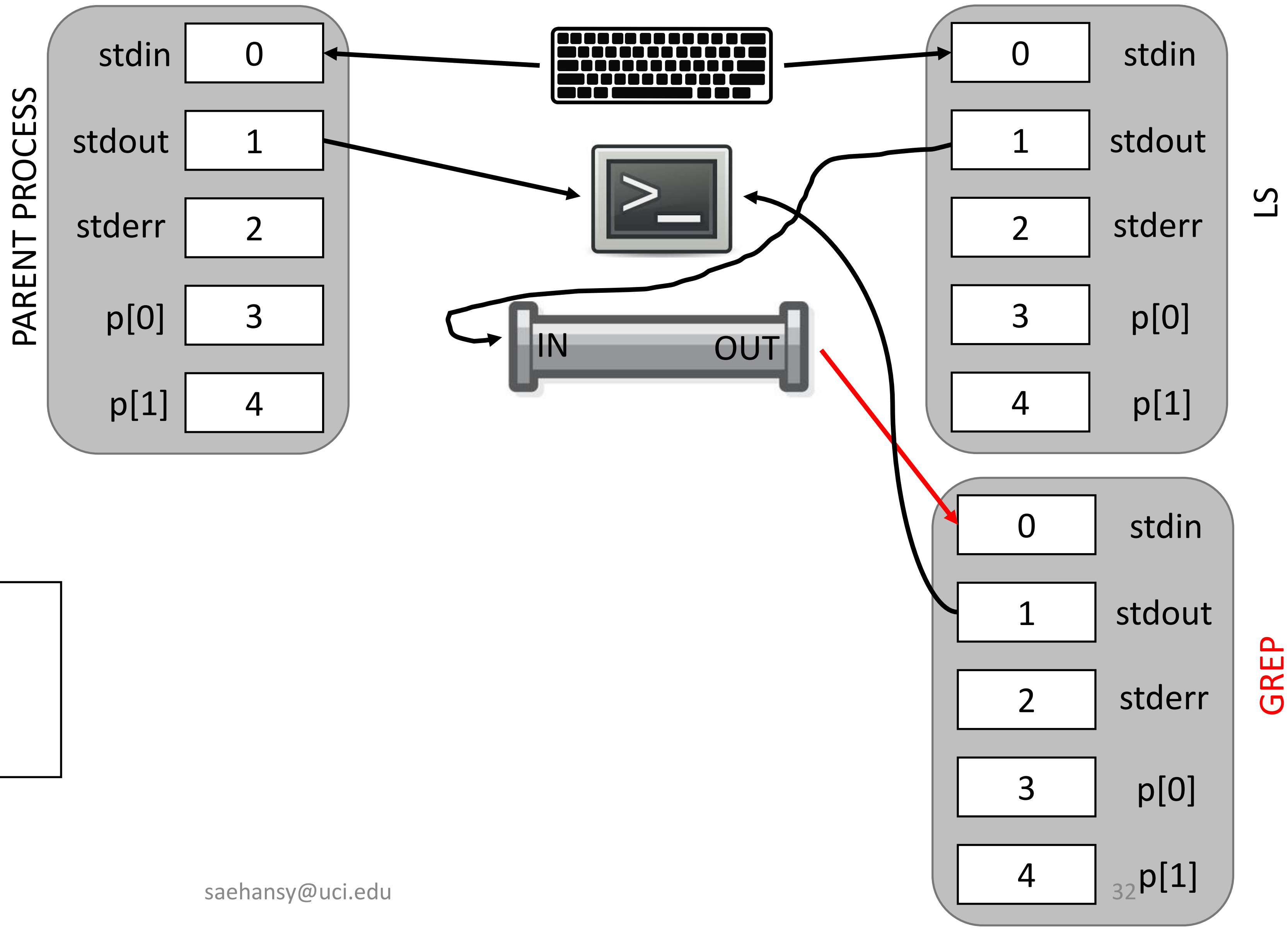
```
$
$ ls | grep asdf
asdfasdf
$
```

pipe() and fork()



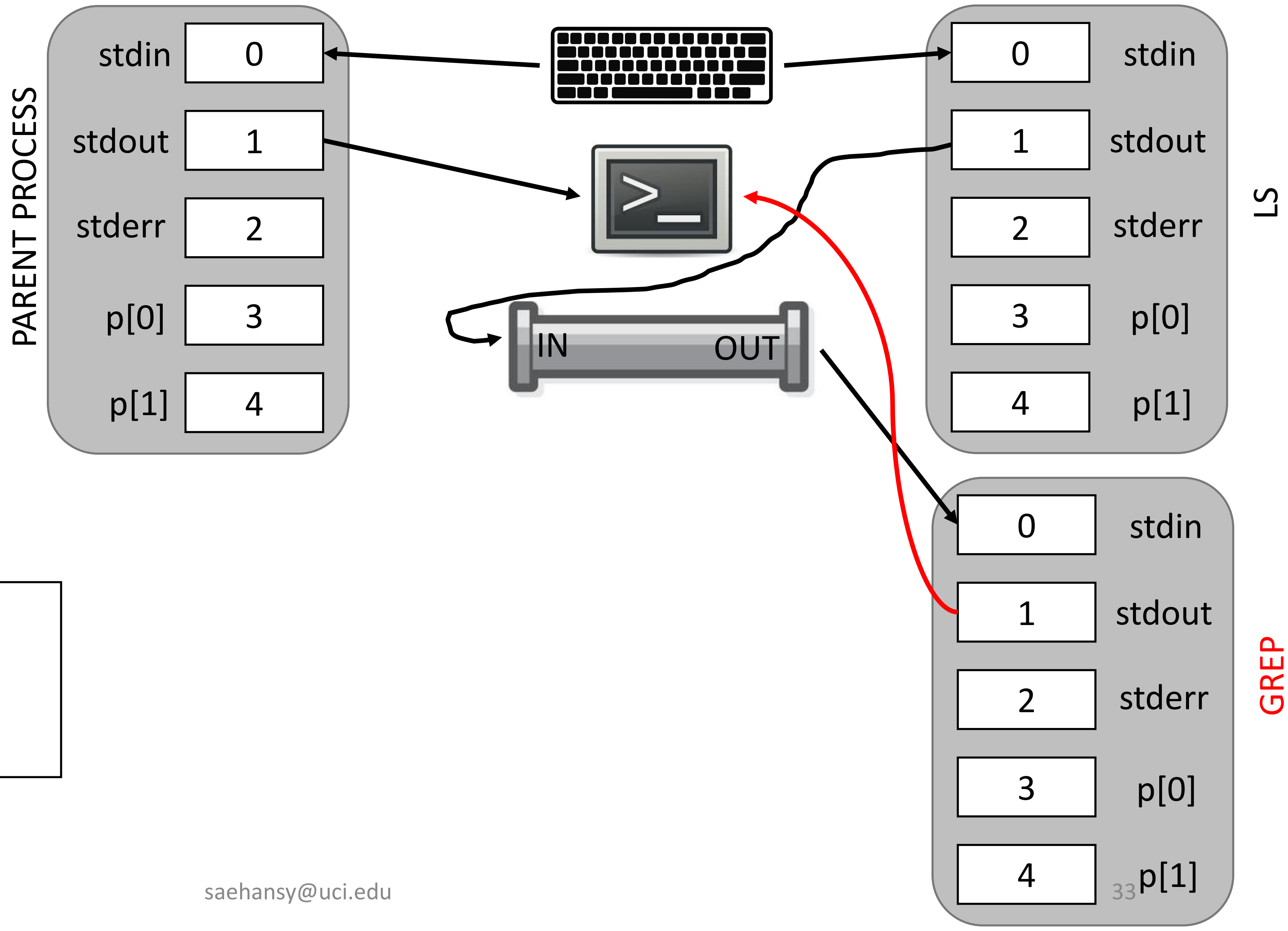
```
$
$ ls | grep asdf
asdfasdf
$
```

pipe() and fork()



```
$
$ ls | grep asdf
asdfasdf
$
```

pipe() and fork()



```
$
$ ls | grep asdf
asdfasdf
$
```

Debugging xv6 user-programs

- If you start gdb with make ‘qemu-nox-gdb’ only kernel symbols are loaded
- The symbols of user programs(UPROGS in Makefile)—including sh, grep, ls—must be loaded for debugging
- *file <binary>* followed by *break main*
- UPROGS binary names start with _ (e.g. _sh)

```
(gdb) file _nsh
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Load new symbol table from "/home/saehansy/Workspace/ics143a/FQ19/qemu/xv6-public/_nsh"? (y or n)y
Reading symbols from /home/saehansy/Workspace/ics143a/FQ19/qemu/xv6-public/_nsh...done.
```

Debugging xv6 user-programs

- We are dealing with shell which has fork() and exec()
- Tell GDB what to follow (parent? children? or new process? old one?)
    - set follow-fork-mode (parent|**children**)
    - set follow-exec-mode (**new**|old)
    - make sure set the breakpoint inside child's code!
- if you having trouble booting xv6 after setting breakpoints, set them just before sh is executed
    - break exec
    - continue
    - 1st break
    - continue
    - 2nd break
    - if you type continue here, it will execute the shell. Type necessary things before typing continue including *del br 1*

it's a little buggy.. gdb is not always correct

Understanding sh.c

- Try out various commands, and use gdb to follow the call stack(graph)
- Make a note on each function
- Drawing a call graph for each scenario helps understanding the structure