CS143A
Principles on Operating Systems
Discussion 02:
OS Interfaces

Instructor: Prof. Anton Burtsev

TA: Sae**hans**eul Yi (Hans)

Oct 16, 2019 **Noon**

Revision: 2

About me

- Link for all office hours/discussion: https://uci.zoom.us/j/93369206818

- Teaching staff office hours:

  Hari: *Mon* 12:00 PST
  Zhaofeng Li: Tue 12:00 PST
  Deep: *Wed* 9:00 AM PST
  Hans: *Thu* 12:00 PST
  Se-Min Lim: *Fri* 9:00 PST

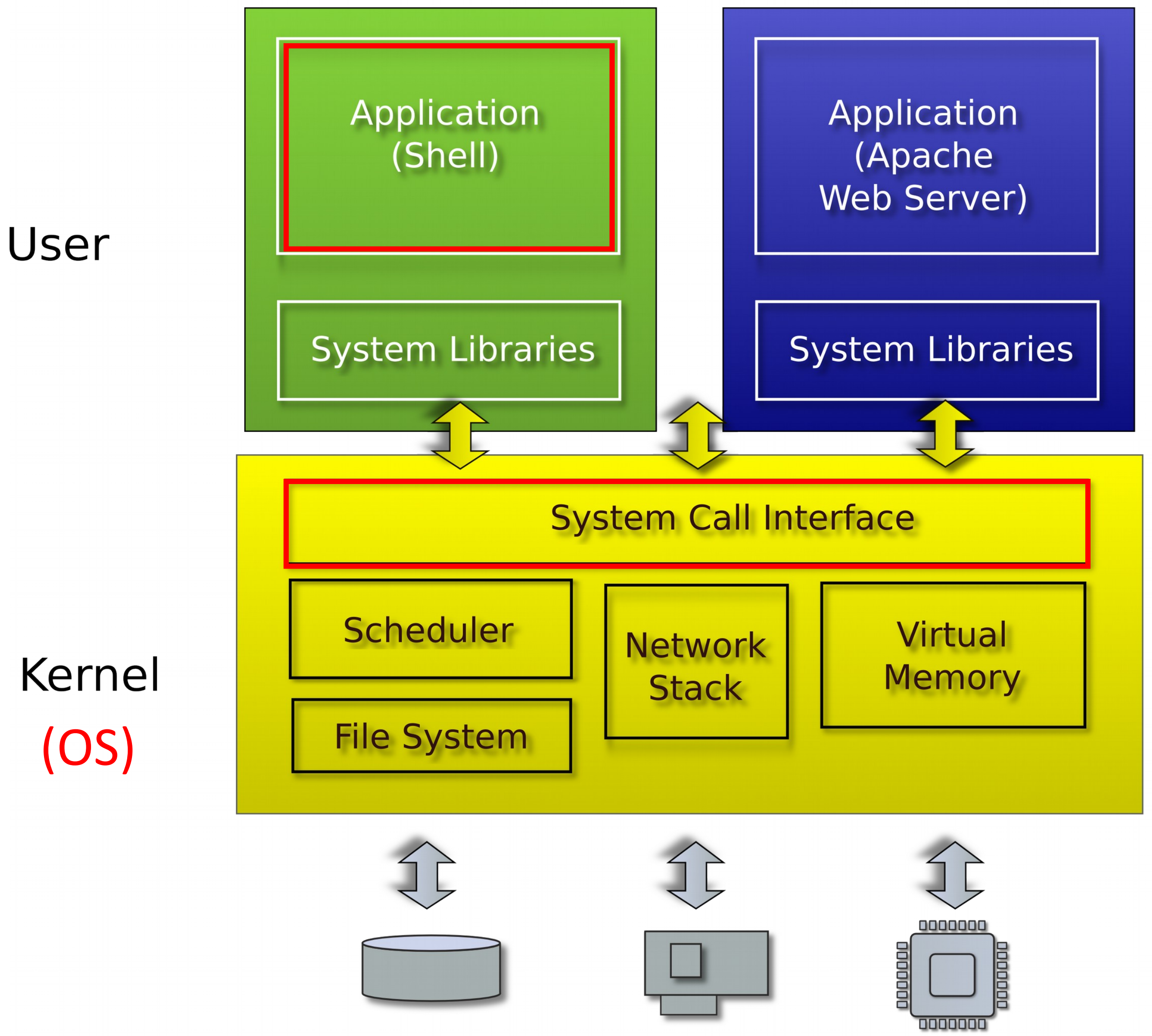
Motivating example: redirection

- Best example for explaining pipe(),fork(), and exec()
- Program output -> *stdout* (default: screen)
- | (pipe *operator*): send outputs to somewhere else

```
$
$ ls
a.out    b.out    asdfasdf
$
```

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

Typical UNIX OS



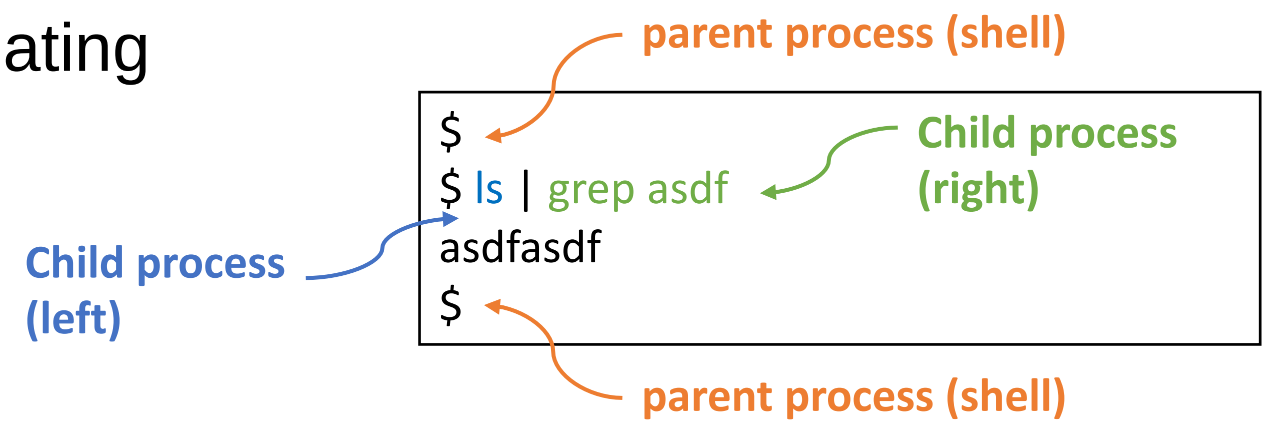
System calls are the interface of the OS

But what is shell?

- Normal process
  - Kernel starts it for each user that logs in into the system
  - In xv6 shell is created after the kernel boots
- Shell interacts with the kernel through system calls
  - E.g., starts other processes

System calls, interface for...

- Processes
  - Creating, exiting, waiting, terminating
- Memory
  - Allocation, deallocation
- Files and folders
  - Opening, reading, writing, closing
- Inter-process communication
  - Pipes

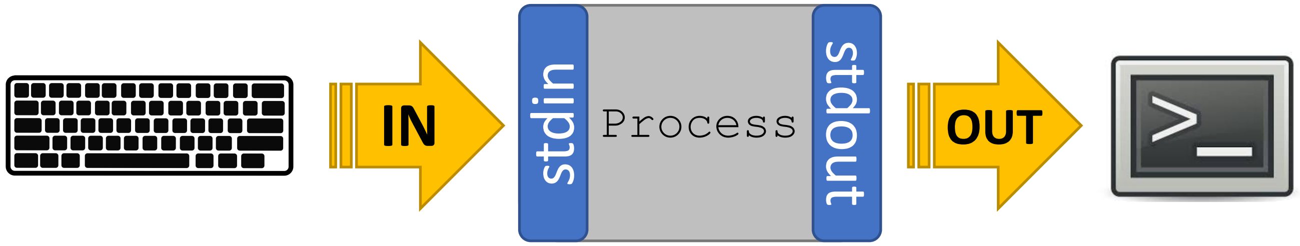


```
$
$ ls | grep asdf
asdfasdf
$
```

Wait… stdin? stdout?

(standard input, standard output)

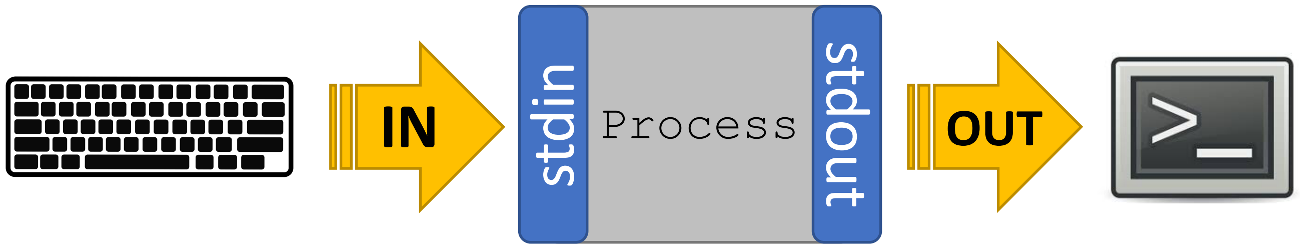
```
$
$ ls | grep asdf
asdfasdf
$
```

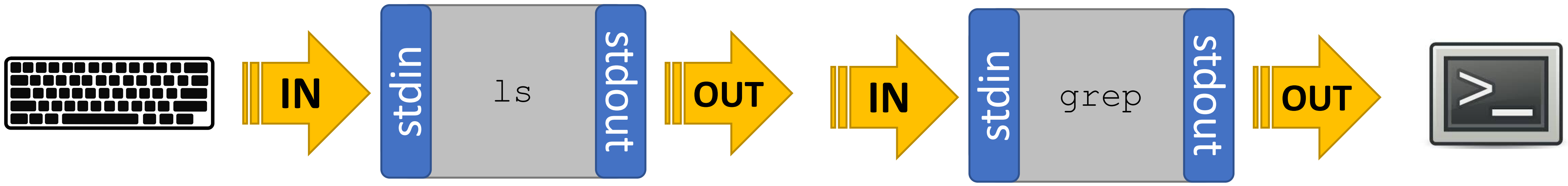


Wait… stdin? stdout?

(standard input, standard output)

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

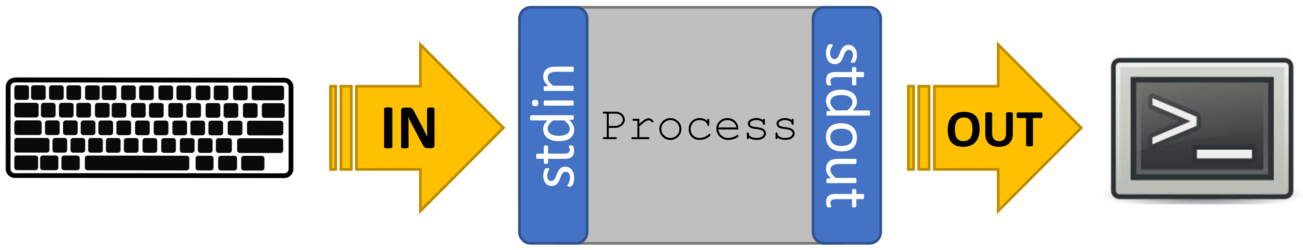


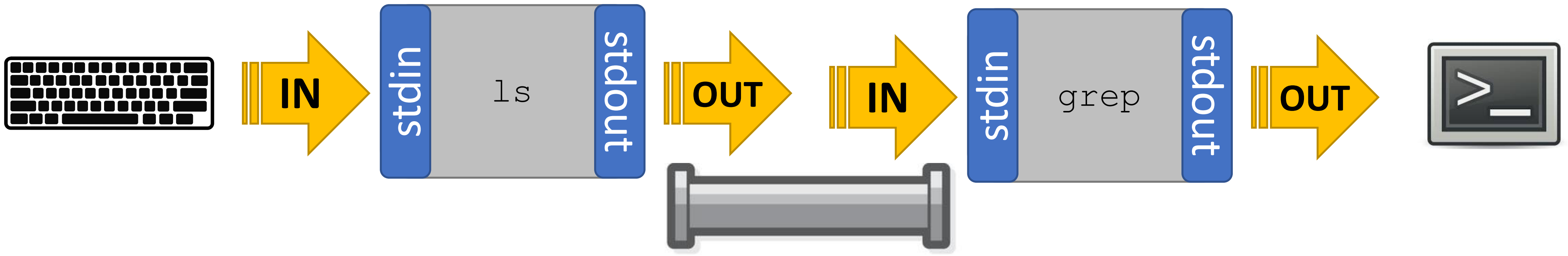


Wait… stdin? stdout?

(standard input, standard output)

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



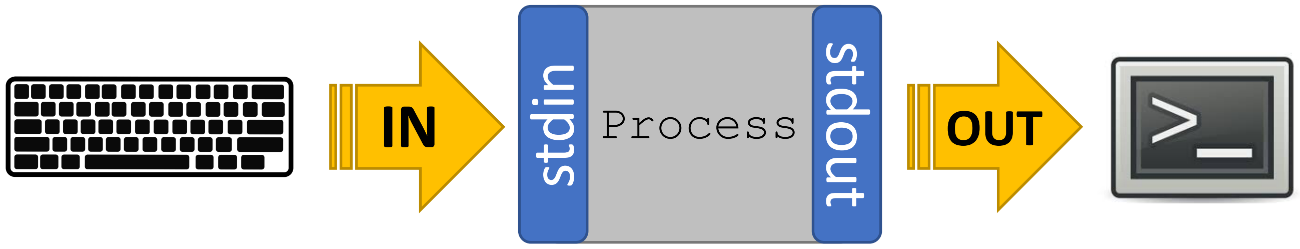


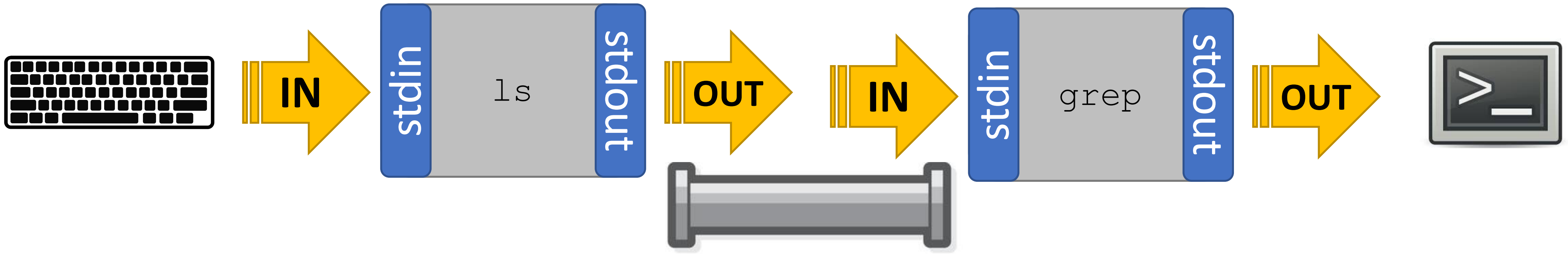
Wait… stdin? stdout?

(standard input, standard output)

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

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



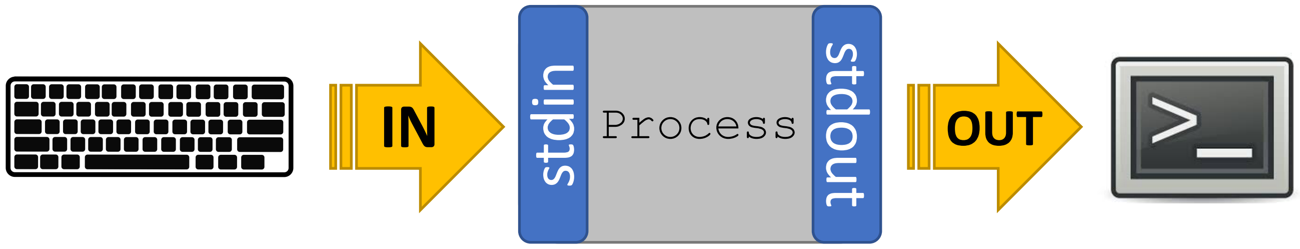


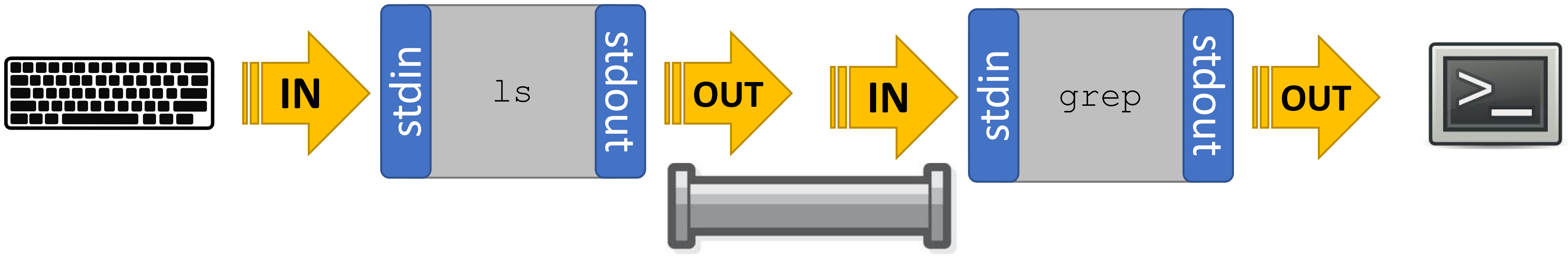
Wait… stdin? stdout?

(standard input, standard output)

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

- stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. **just an integer** in user-program)
- Each program has its own descriptor table



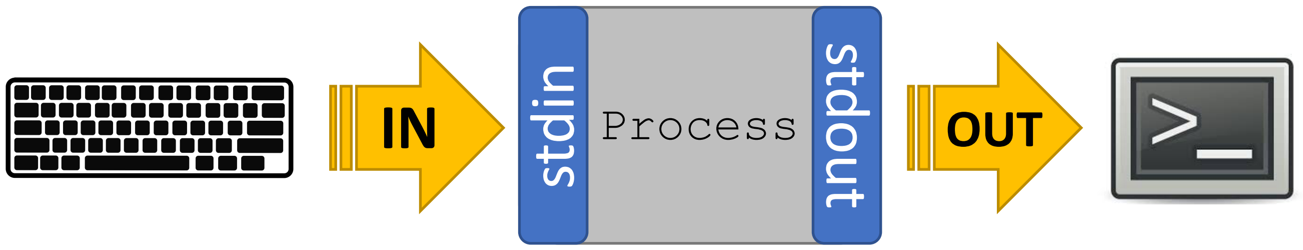


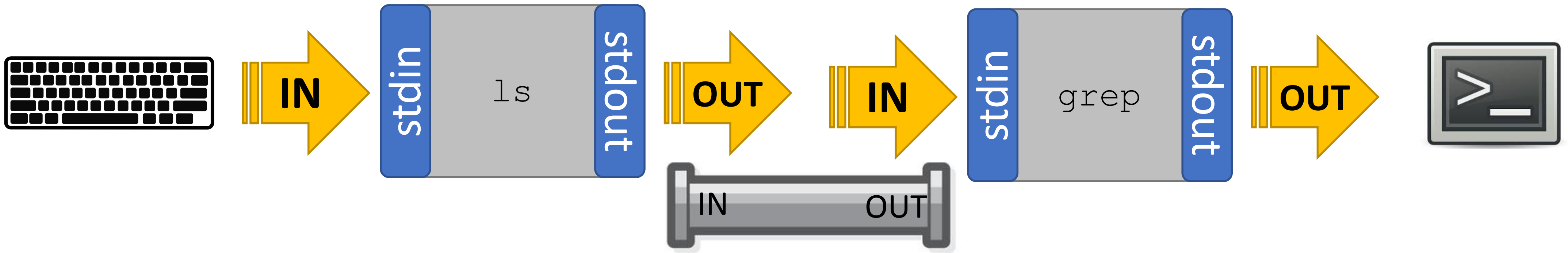
Wait… stdin? stdout?

(standard input, standard output)

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

- stdin(0), stdout(1), and stderr(2) are file descriptors(i.e. **just an integer** in user-program)
- Each program has its own descriptor table
- How to modify process' file descriptors?



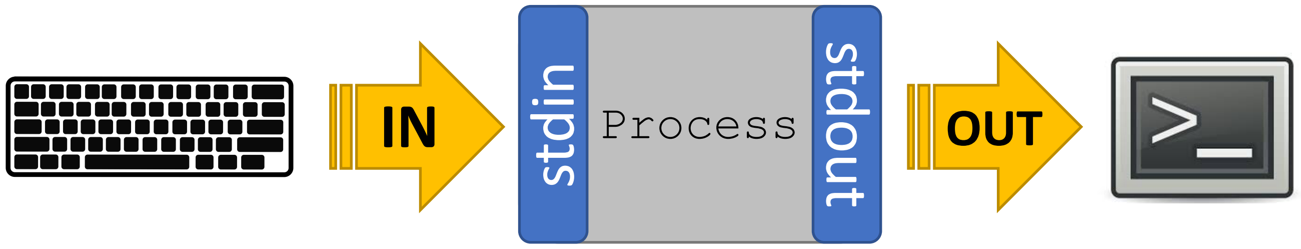


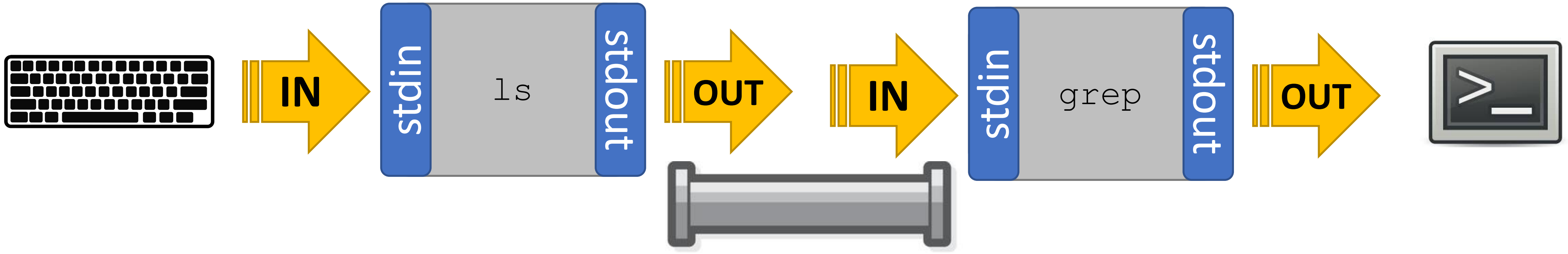
Wait… stdin? stdout?

(standard input, standard output)

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

- stdin(0), stdout(1), and stderr(2) are file descriptors(**just an integer** in user-program)
- Each program has its own descriptor table
- How to modify process’ file descriptors?
  - close, dup(or open)



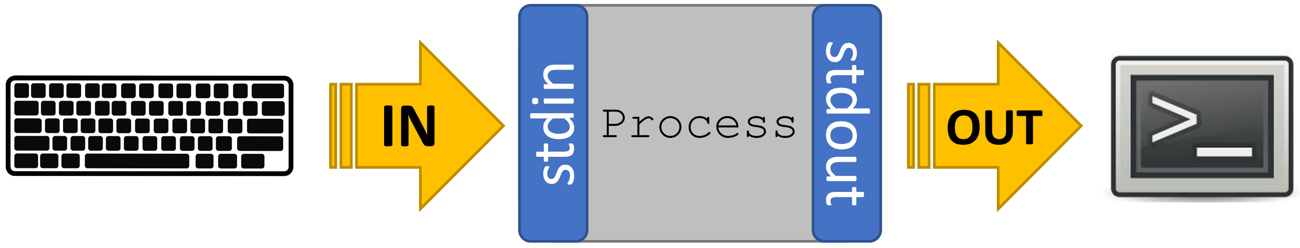


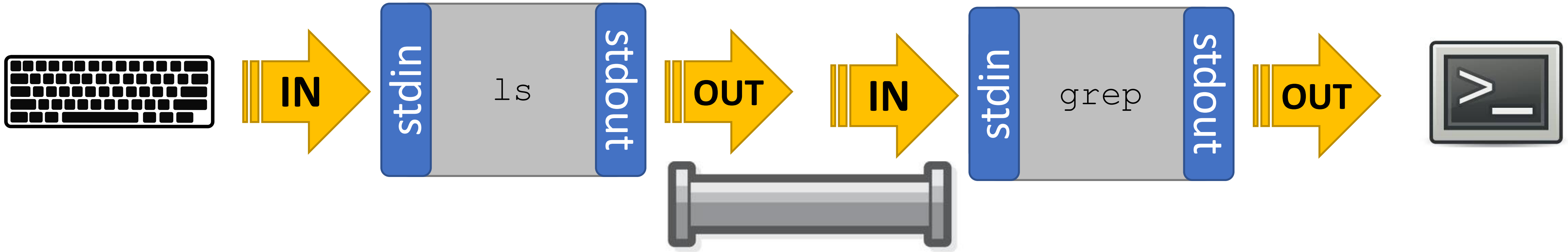
Wait… stdin? stdout?

(standard input, standard output)

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

- stdin(0), stdout(1), and stderr(2) are file descriptors(**just an integer** in user-program)
- Each program has its own descriptor table
- 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



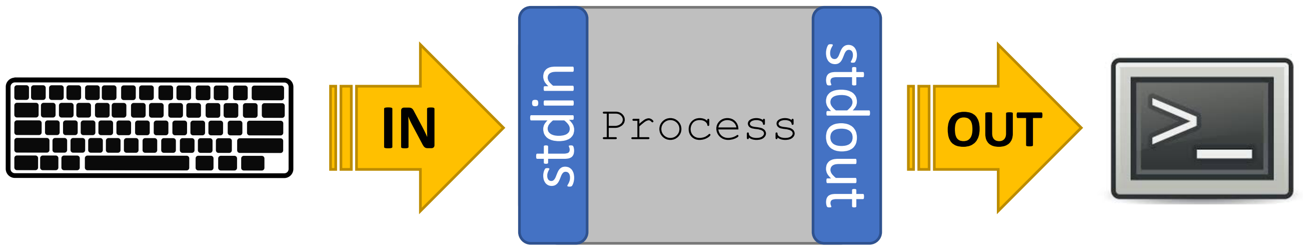


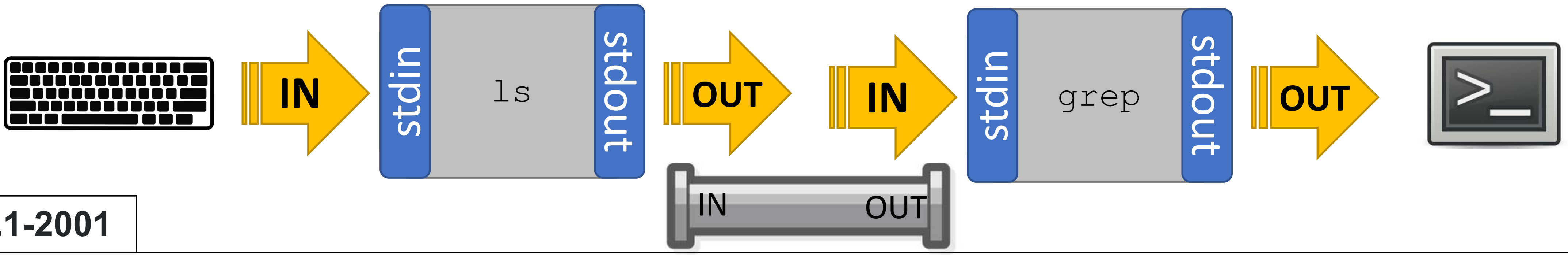
Wait... stdin? stdout?

(standard input, standard output)

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

- stdin(0), stdout(1), and stderr(2) are file descriptors(**just an integer** in user-program)
- Each program has its own descriptor table
- 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

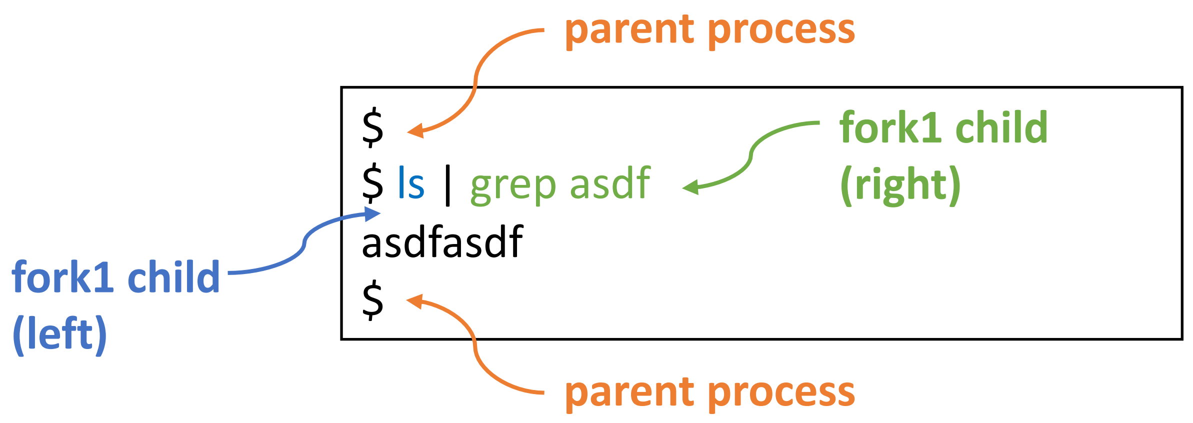
pipe() creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing. **pipe is uni-directional**

pipe() and fork()

int pipe(int *pipefd*[2]);
Create a pipe & assign each end to pipefd

pid_t fork(void);
Copy the current process (parent)
Returns the PID of the child (parent)
or 0 (child)

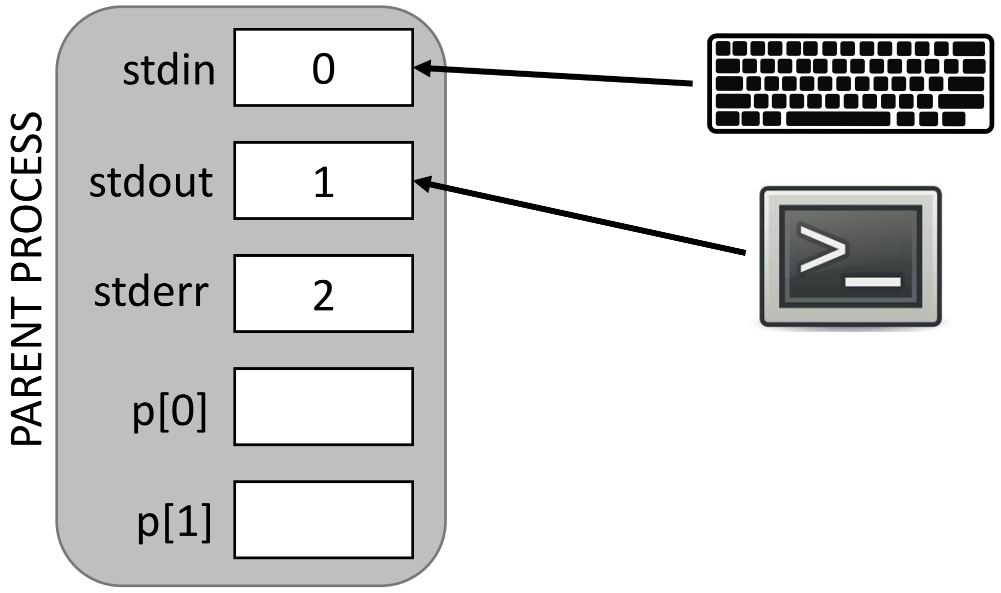
```
---------------------Point 0--------------------
case PIPE:
pcmd = (struct pipecmd*)cmd;
if(pipe(p) < 0)
   panic("pipe");
---------------------Point A--------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
---------------------Point B--------------------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

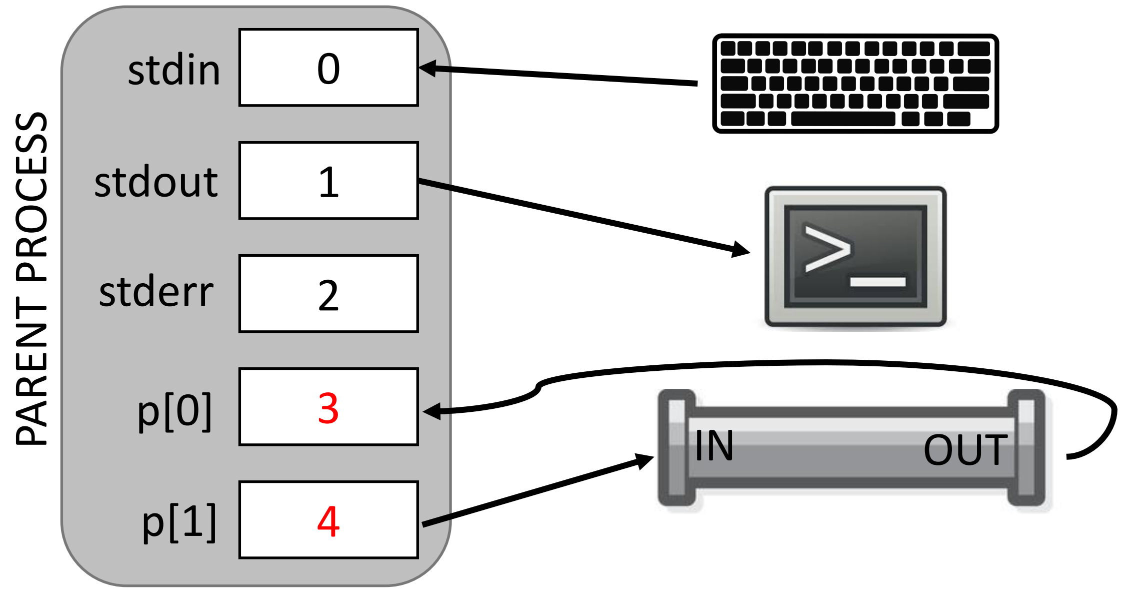
※ Throughout the example, stderr is always connected to the screen. Omitted for simplicity as well as p[0] and p[1] to the parent process

```
---------------------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)          int p[2]
   panic("pipe");
---------------------Point A-------------------
if(fork1() == 0){
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
---------------------Point B--------------------
   runcmd(pcmd->left);
}
```

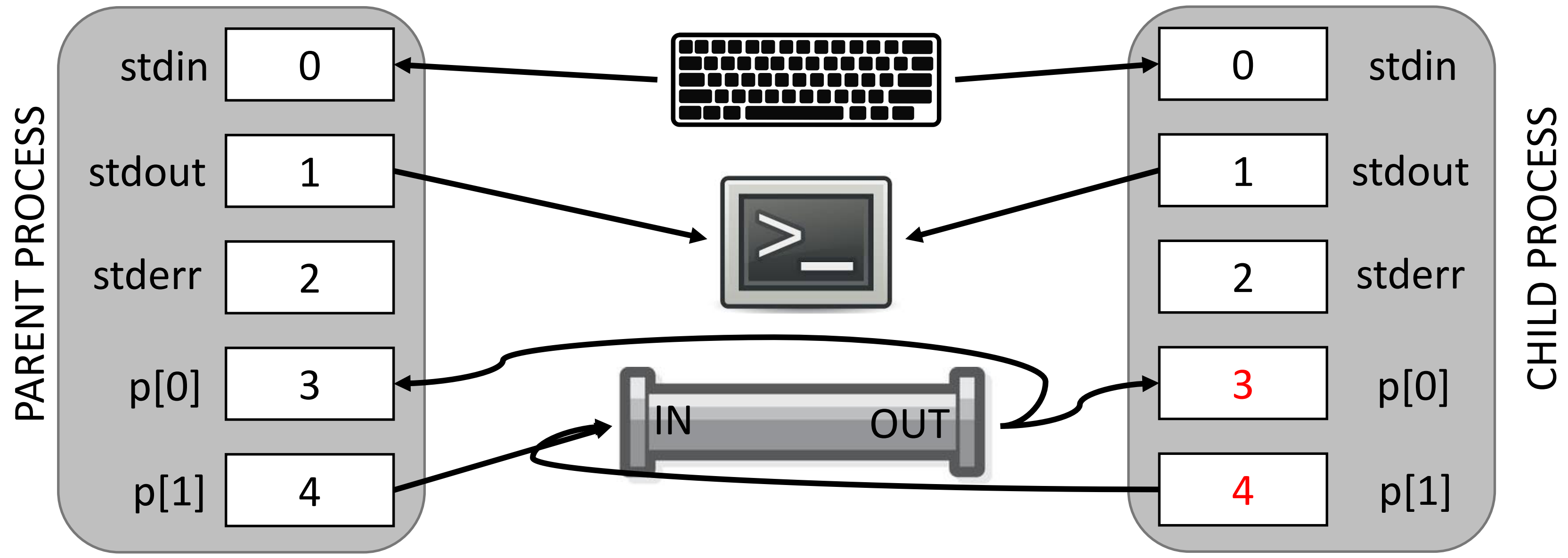


pipe() and fork()

```
--------------------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);
}
```

Executed by child process

fork() copies the descriptors too!

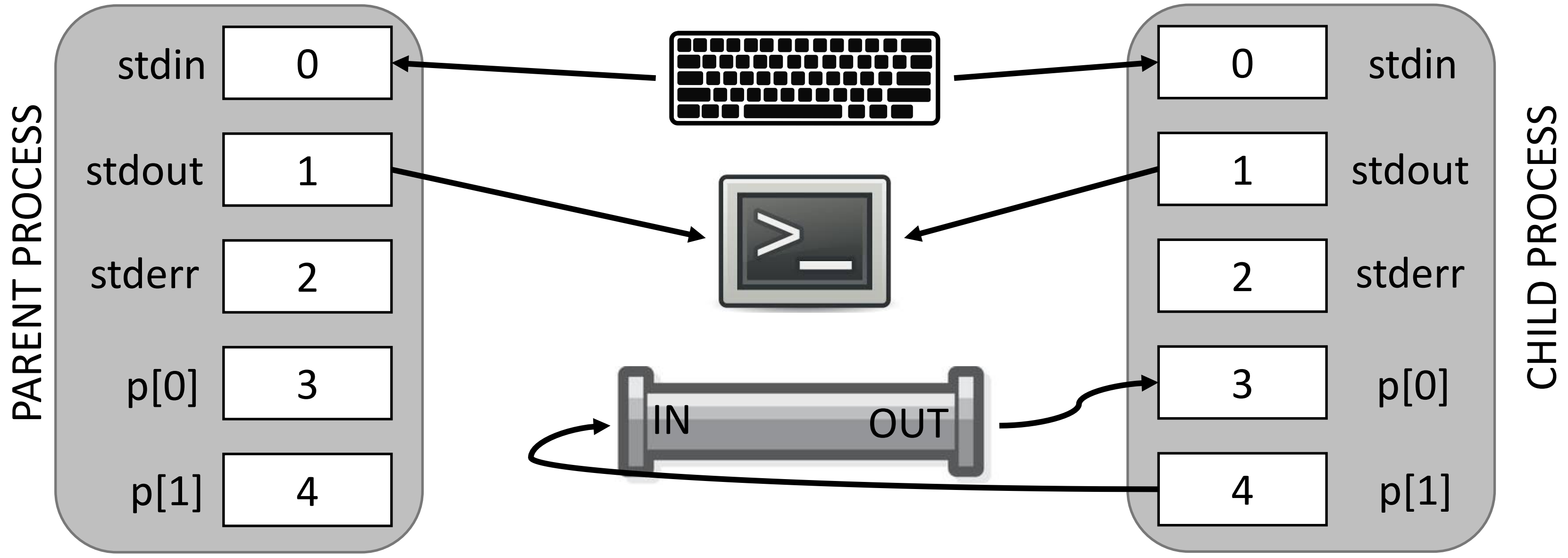


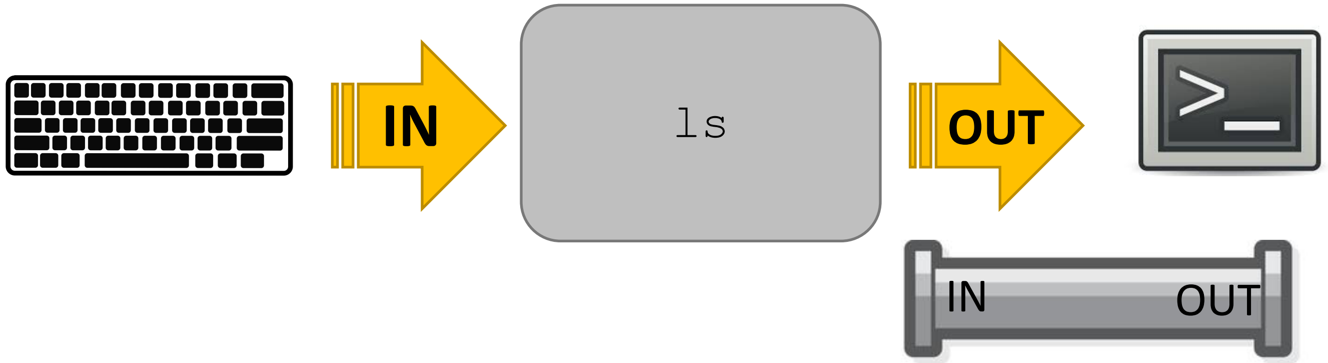
pipe() and fork()

```
--------------------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);
}
```

Executed by child process

fork() copies the descriptors too!



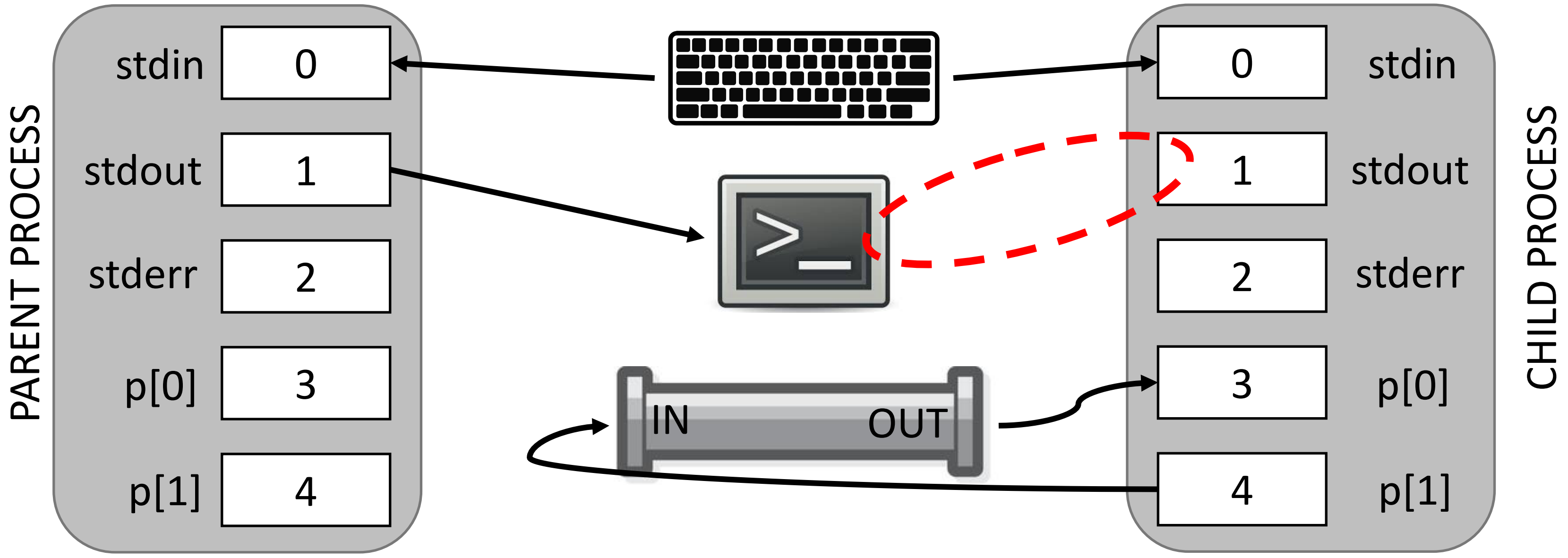


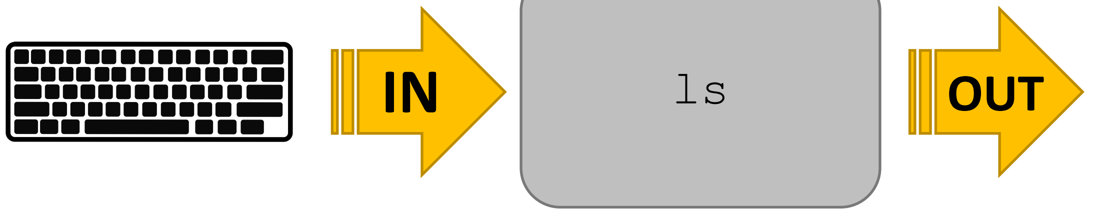
pipe() and fork()

fork() copies the descriptors too!

```
--------------------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);
}
```

Executed by child process



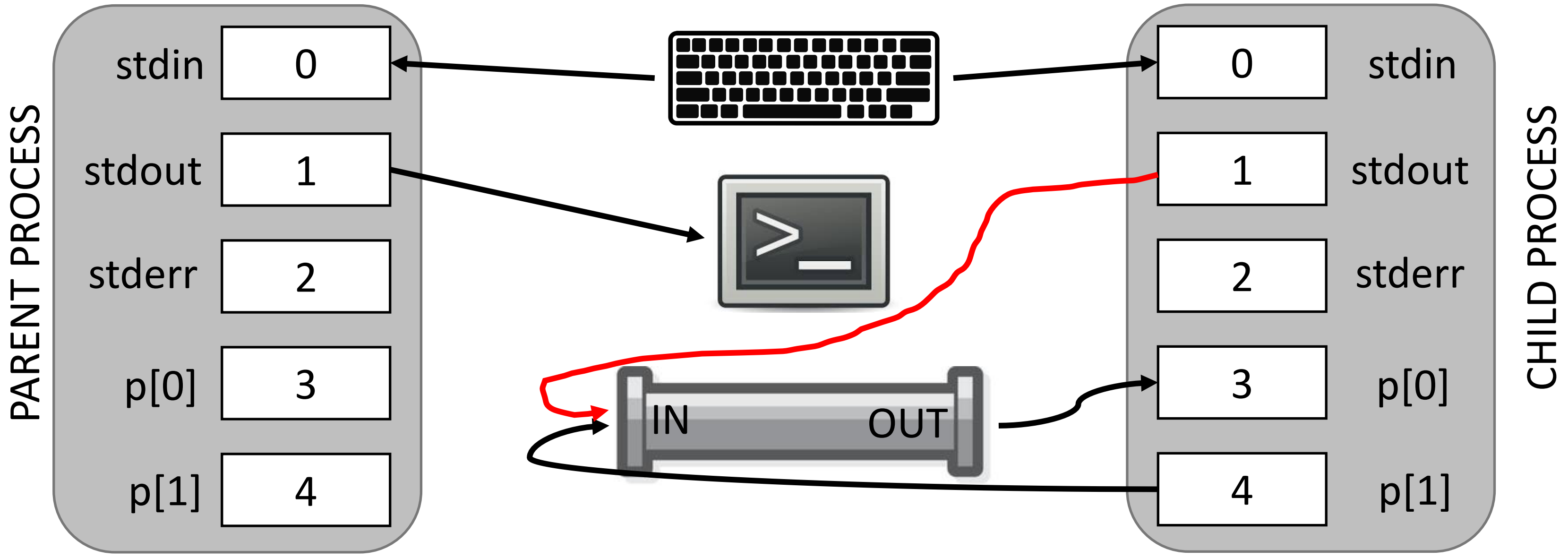


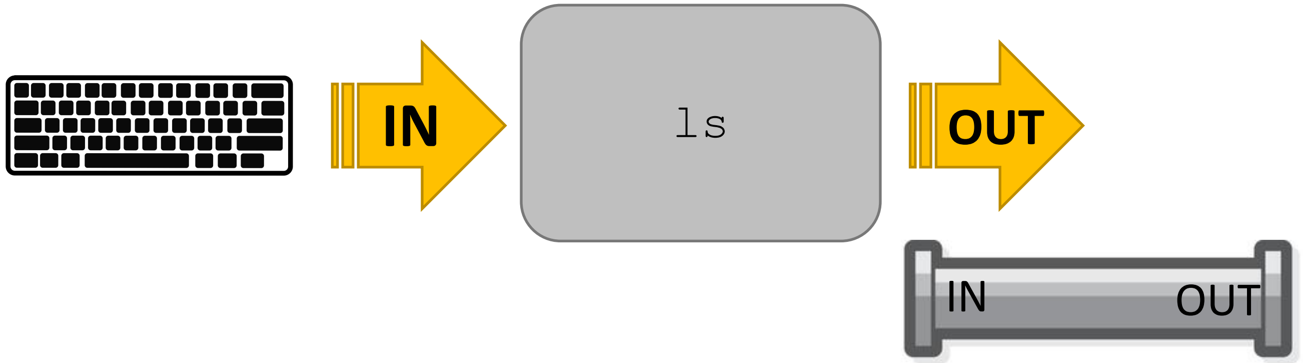
pipe() and fork()

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);
}
```

Executed by child process



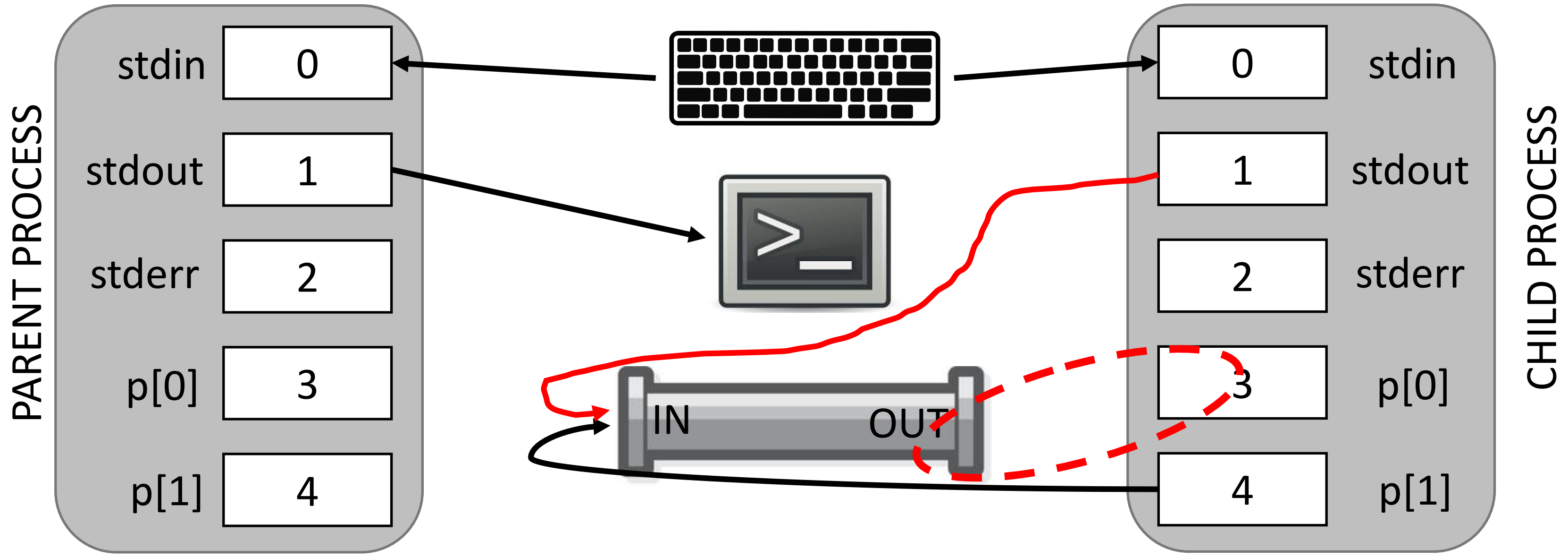


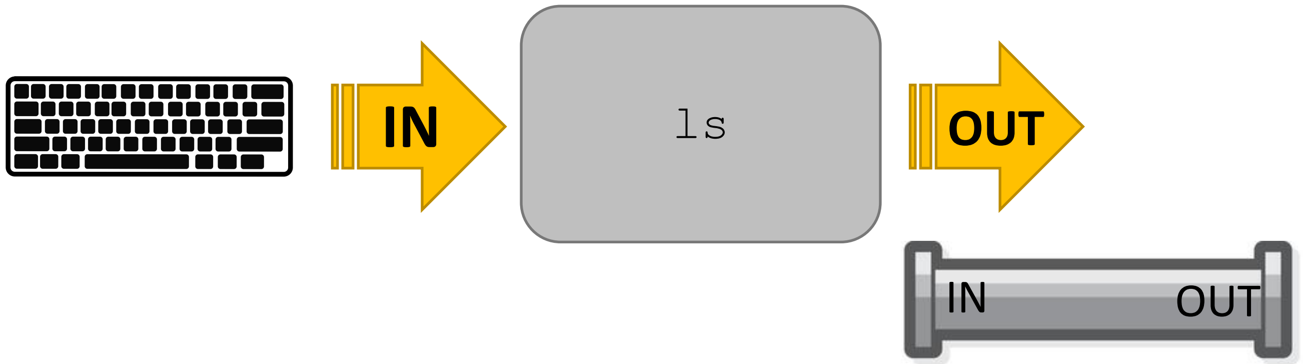
pipe() and fork()

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);
}
```

Executed by child process



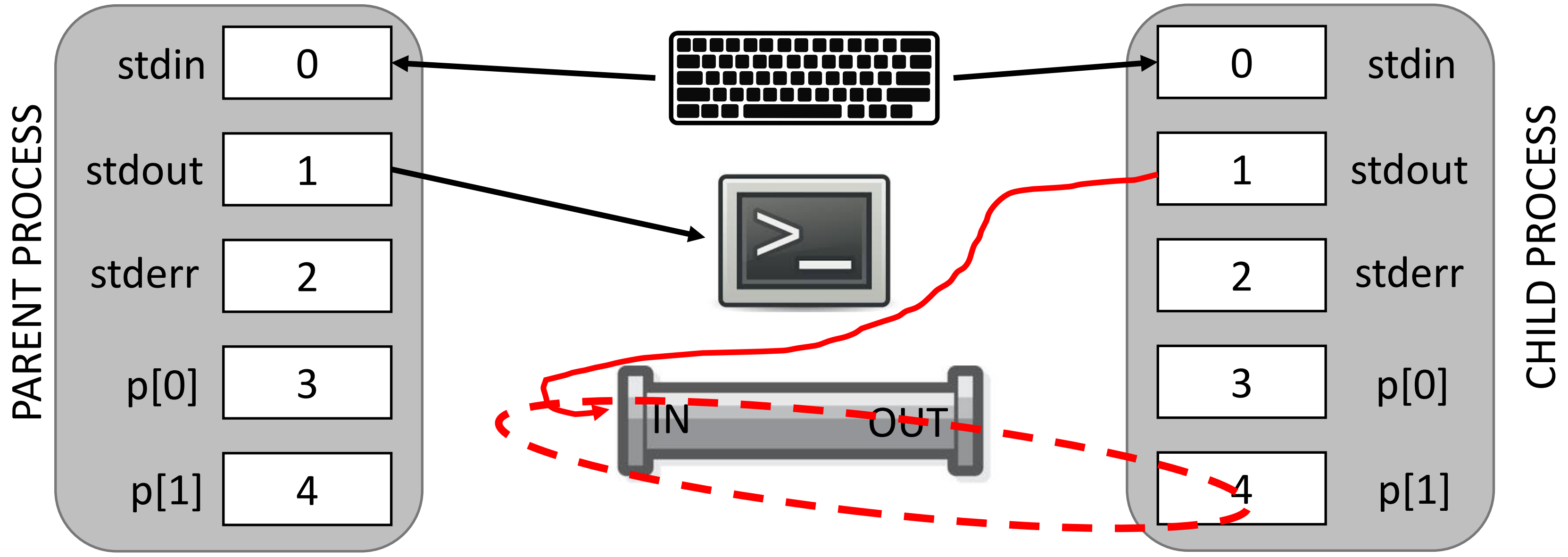


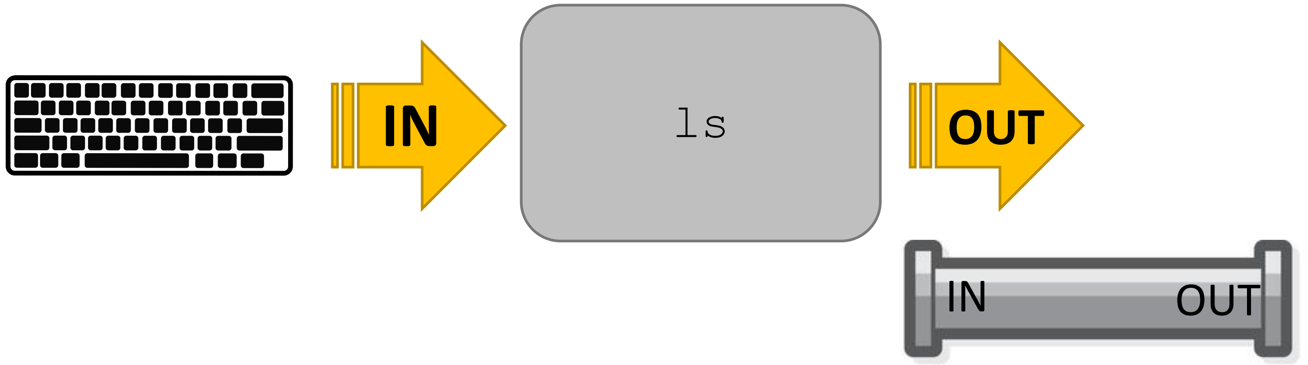
pipe() and fork()

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);
}
```

Executed by child process



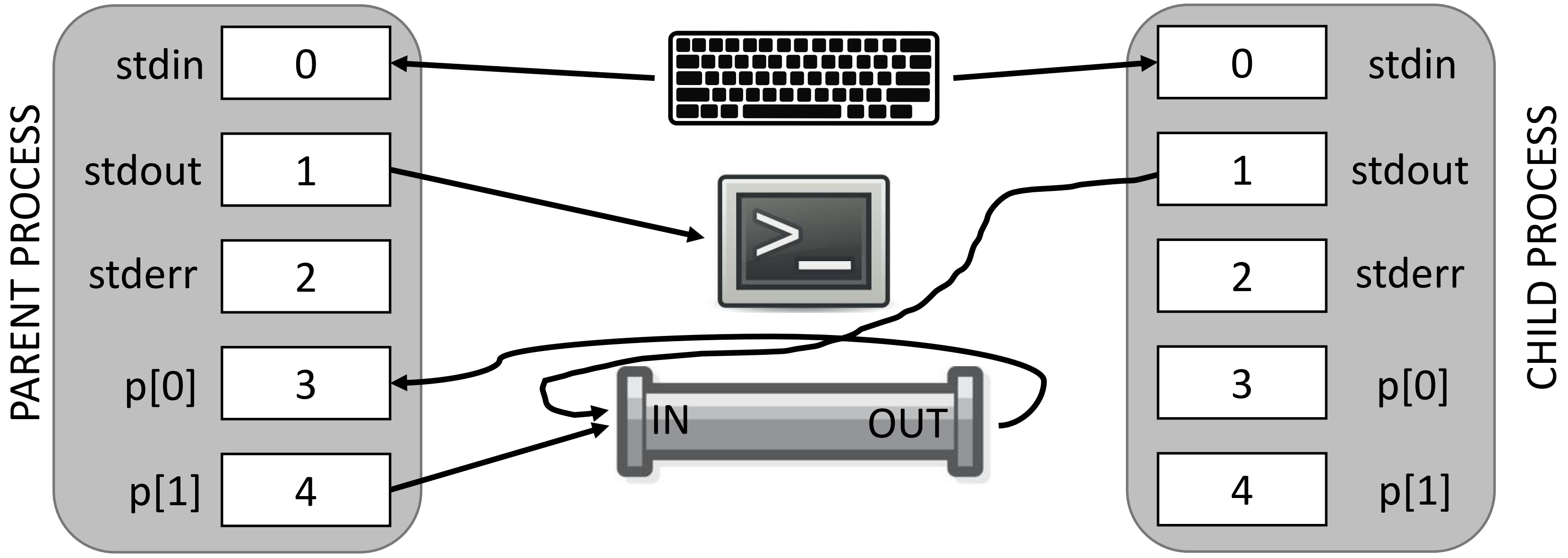


pipe() and fork()

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;
```

Executed by child process

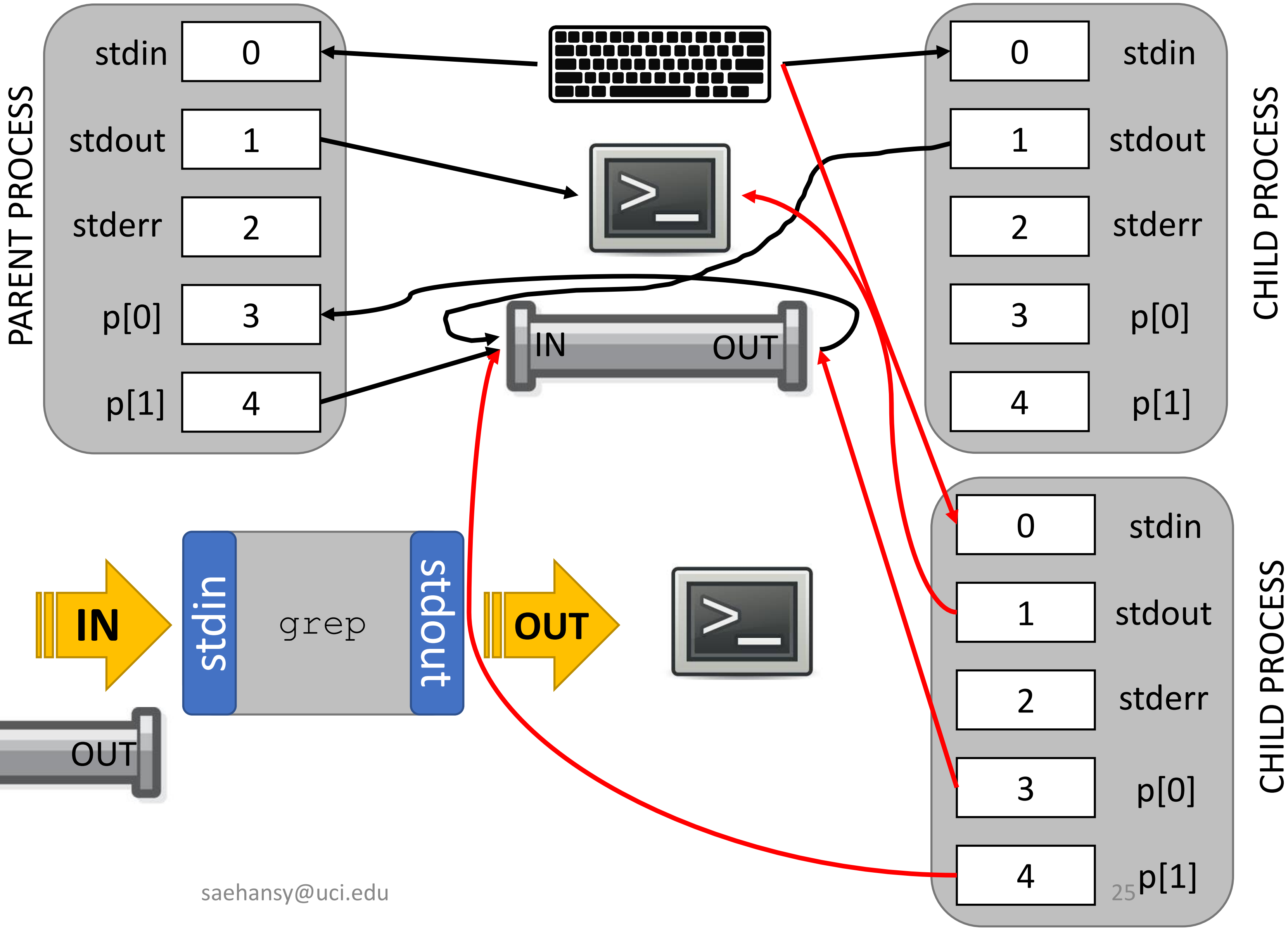


pipe() and fork()

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;
```

Executed by child process

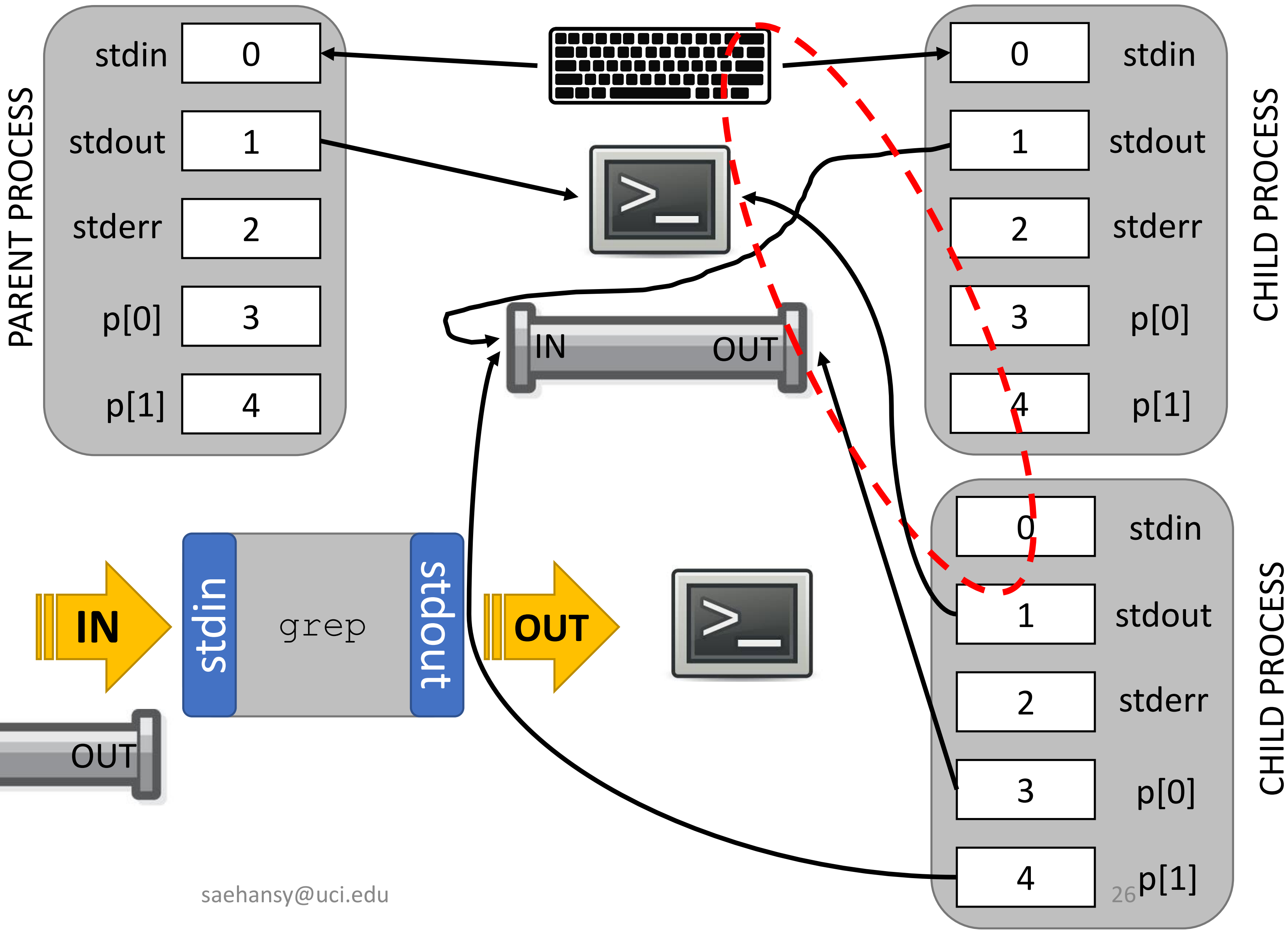


pipe() and fork()

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;
```

Executed by child process

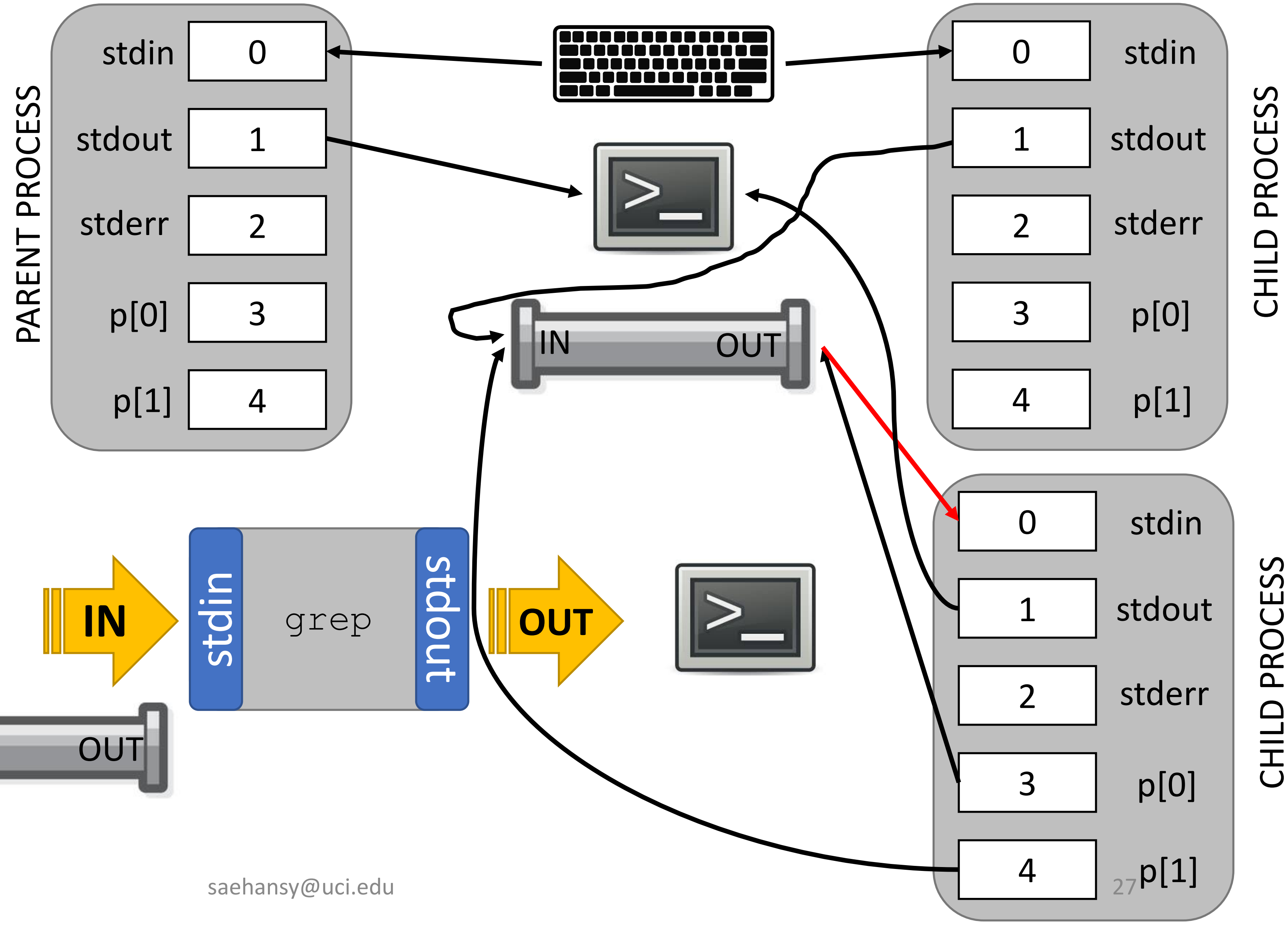


pipe() and fork()

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;
```

Executed by child process

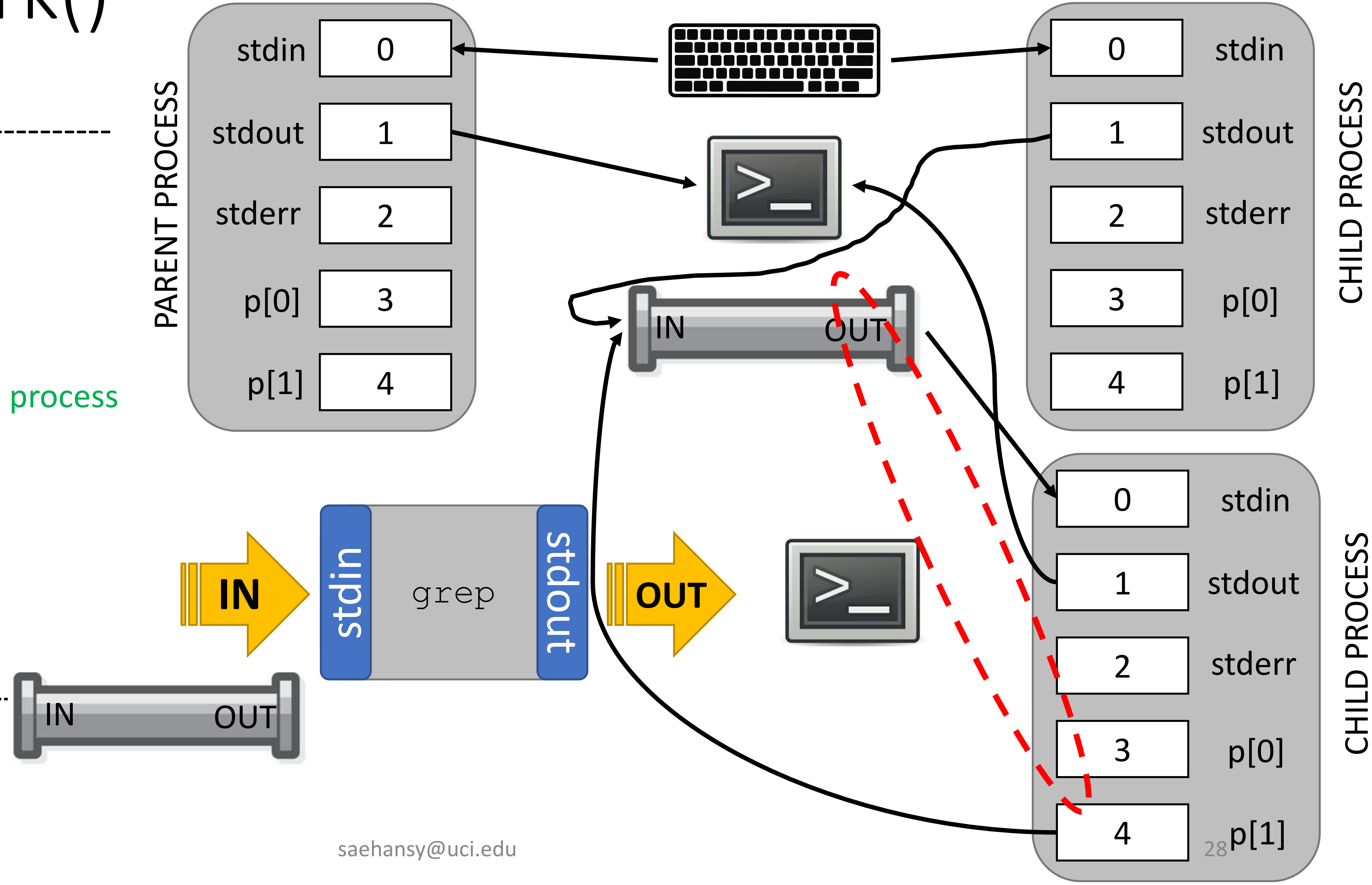


fork() copies the descriptors too!
dup()'s destination is the lowest & unused file descriptor!

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;
```

Executed by child process

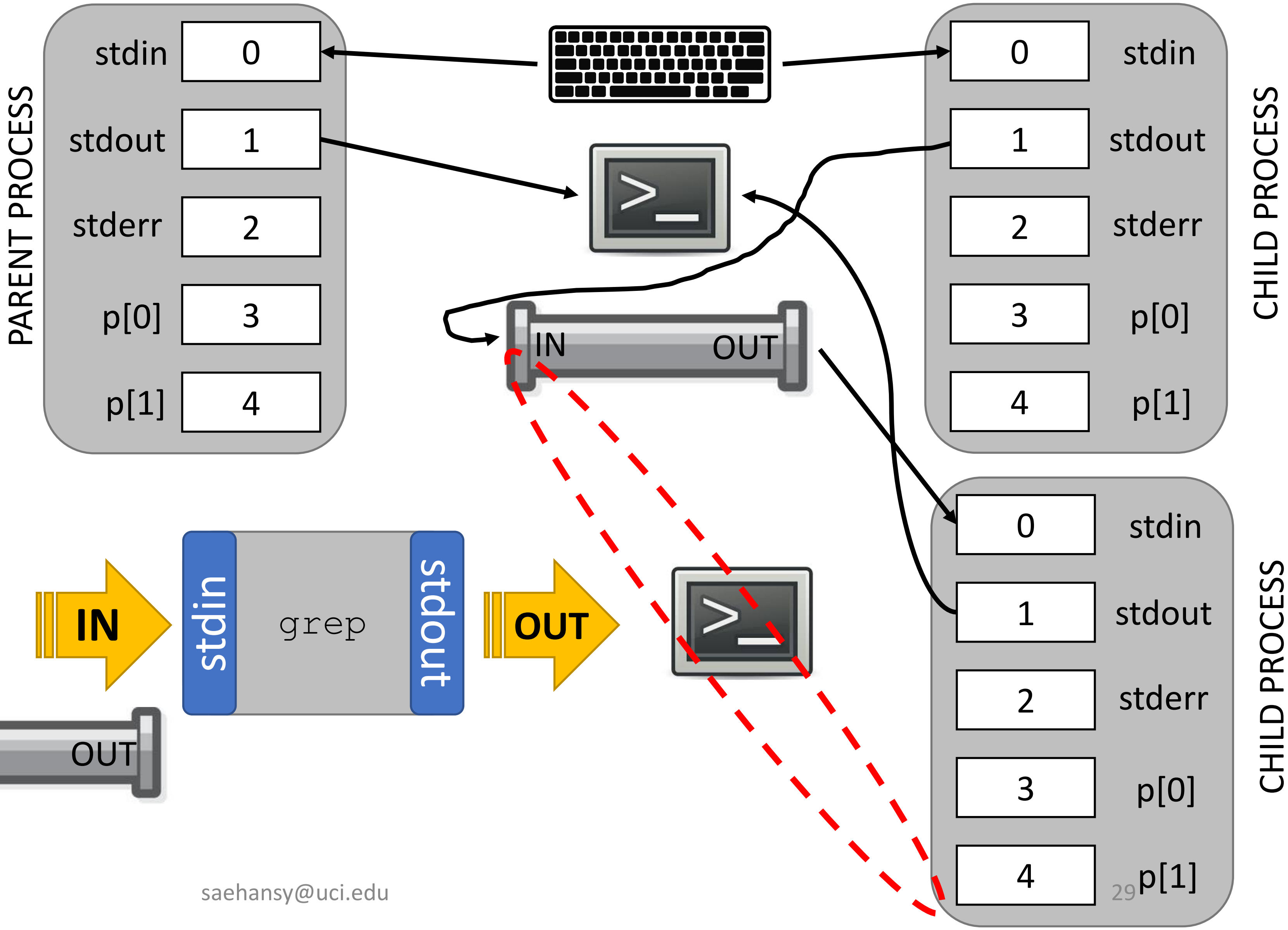


pipe() and fork()

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;
```

Executed by child process

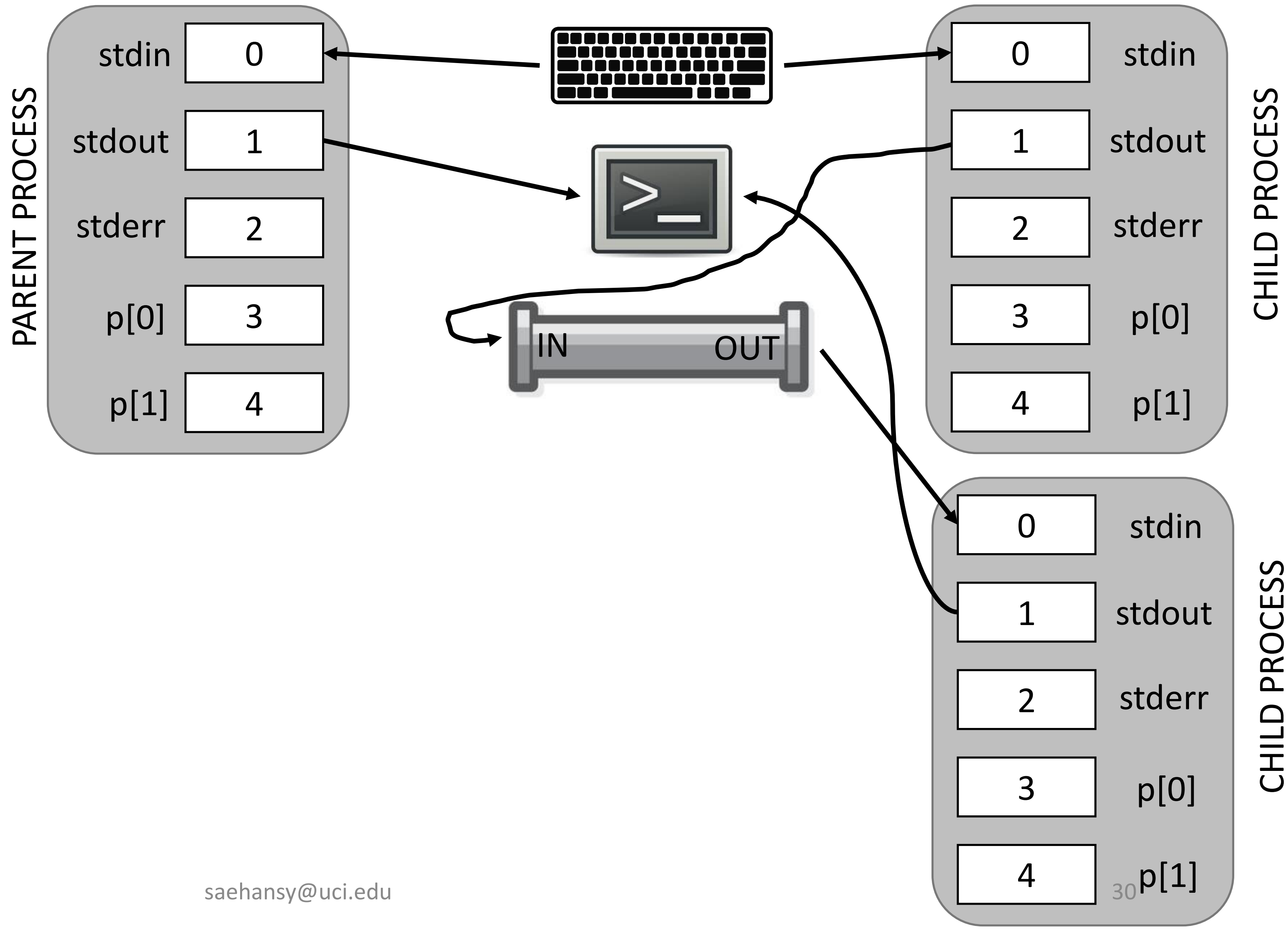


pipe() and fork()

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;
```

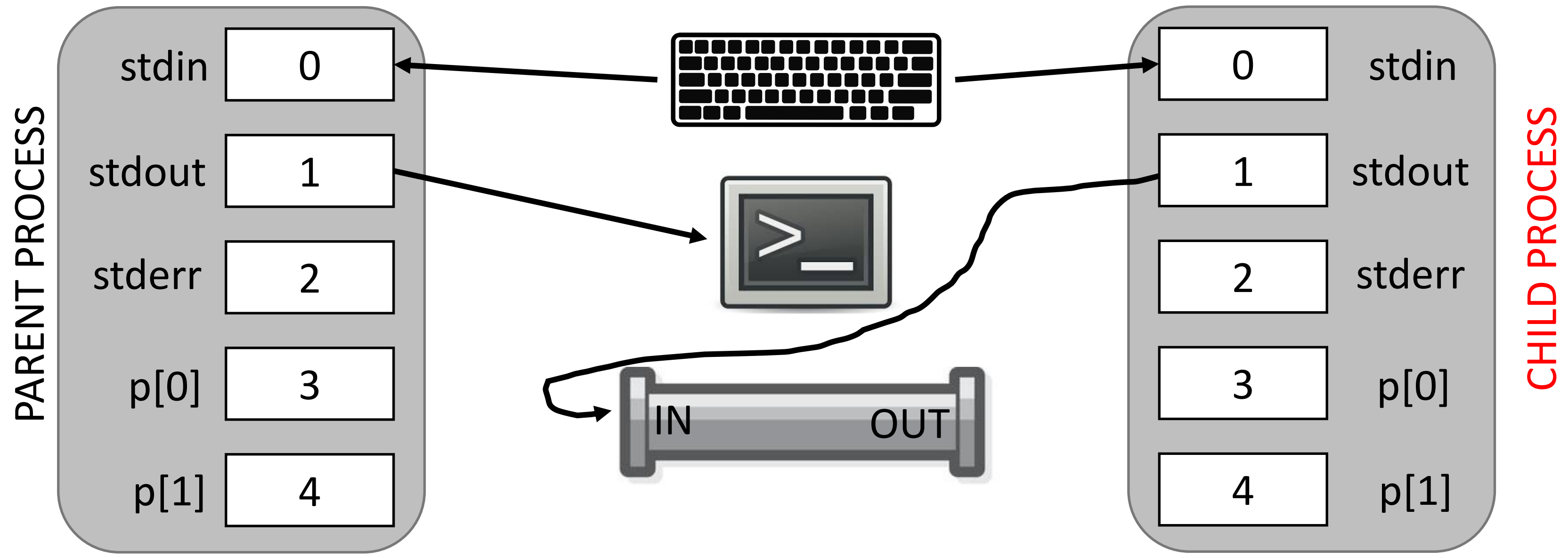
Parent waits child processes



pipe() and fork() and **exec()**

```
if(fork1() == 0){
    …
    runcmd(pcmd->right);
}
```

runcmd() contains exec functions



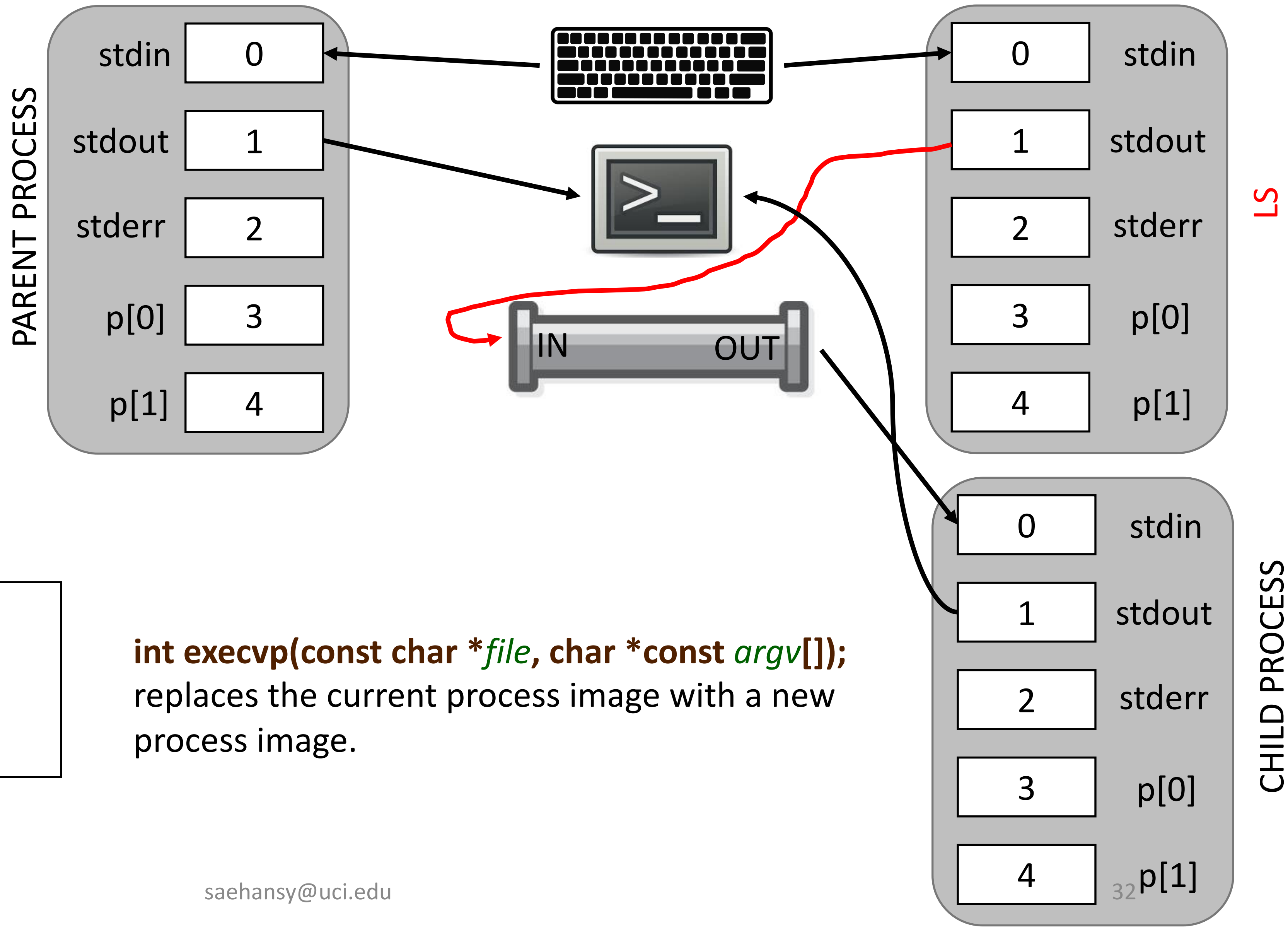
```
$
$ ls | grep asdf
asdfasdf
$
```

int execvp(const char ****file*, char *const** ***argv*****[]);**
replaces the current process image with a new process image.

pipe() and fork() and **exec()**

```
if(fork1() == 0){
    …
    runcmd(pcmd->right);
}
```

runcmd() contains exec functions



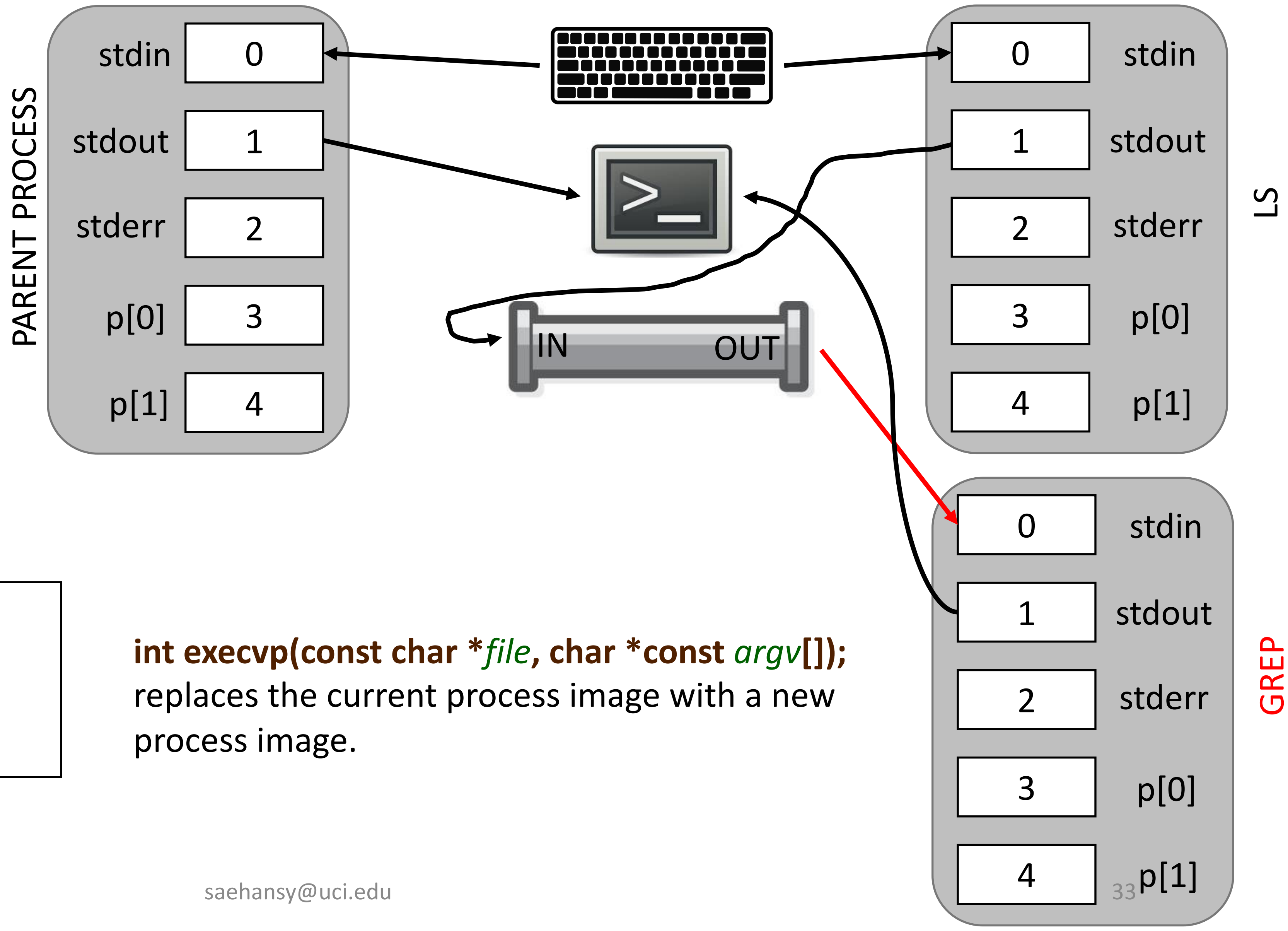
```
$
$ ls | grep asdf
asdfasdf
$
```

int execvp(const char ****file*, **char *const *argv*[]);
replaces the current process image with a new process image.

pipe() and fork() and **exec()**

```
if(fork1() == 0){
    …
    runcmd(pcmd->right);
}
```

runcmd() contains exec functions



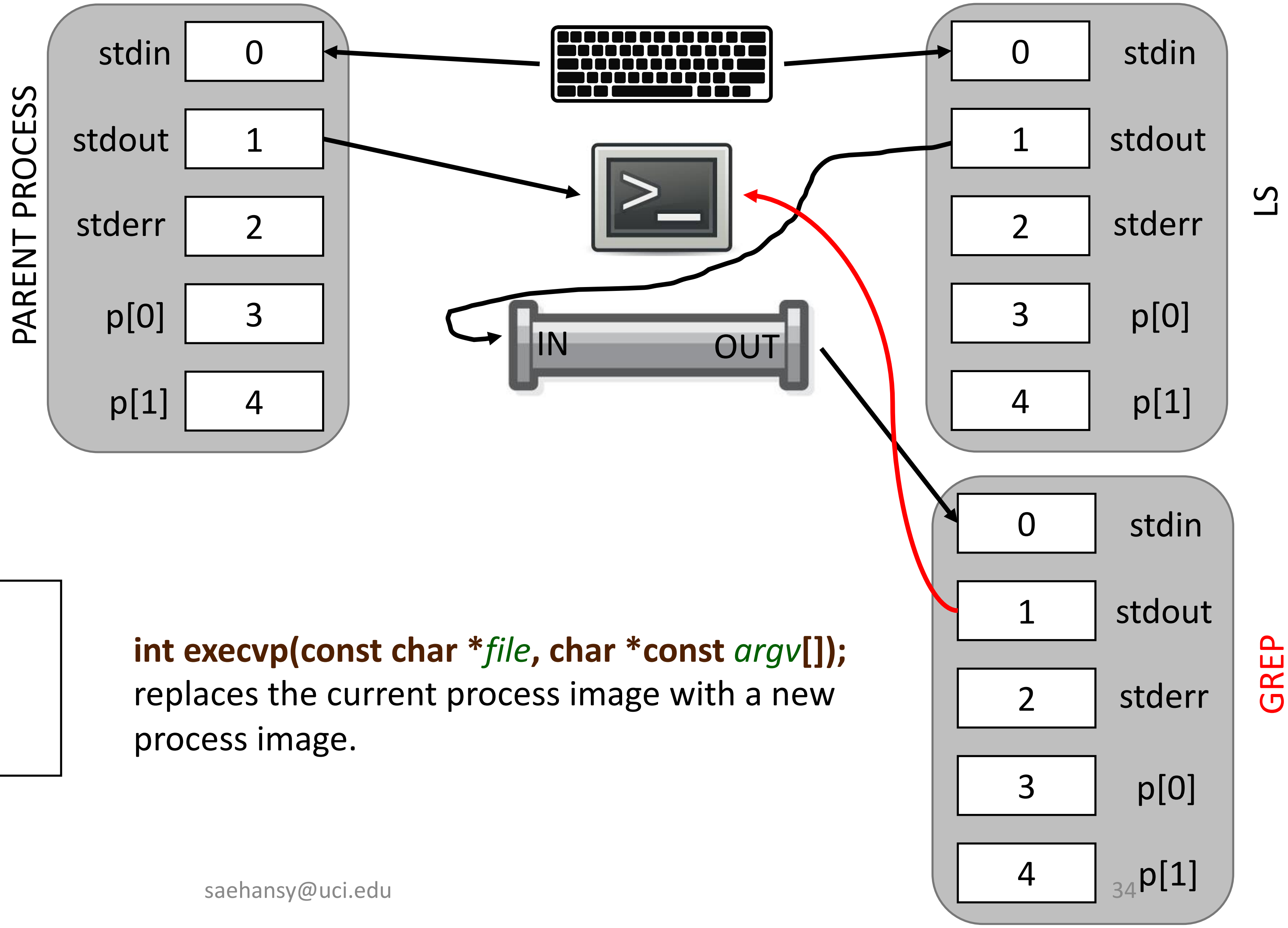
```
$
$ ls | grep asdf
asdfasdf
$
```

int execvp(const char ****file*, char *const** ***argv*****[]);**
replaces the current process image with a new process image.

pipe() and fork() and **exec()**

```
if(fork1() == 0){
    …
    runcmd(pcmd->left);
}
```

runcmd() contains exec functions



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

int execvp(const char *****file*, char *const ***argv***[]);**
replaces the current process image with a new process image.