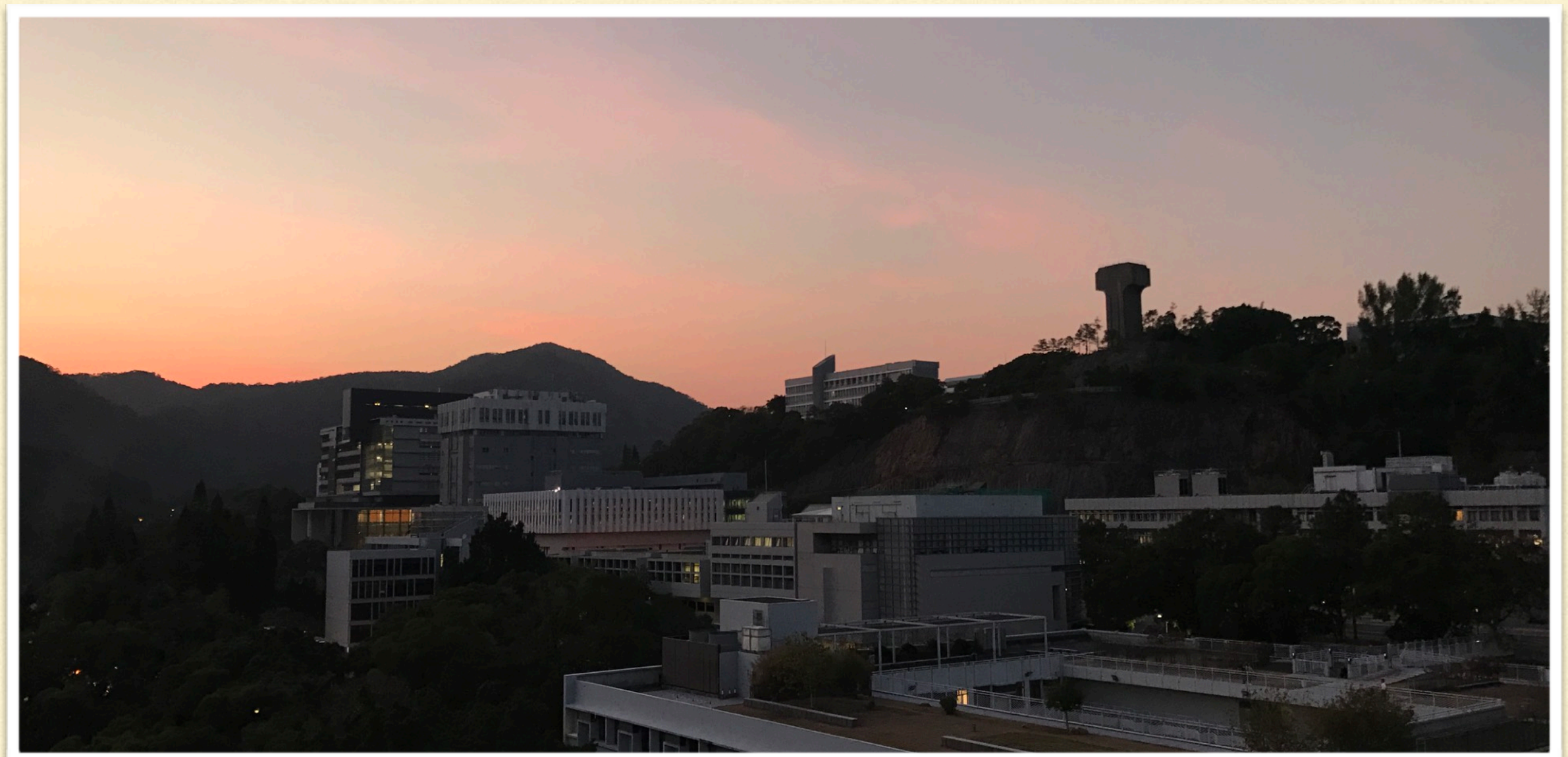


CSCI3150 TUTORIAL- IPC

Calvin Kam (hckam@cse)

WHAT IS IPC

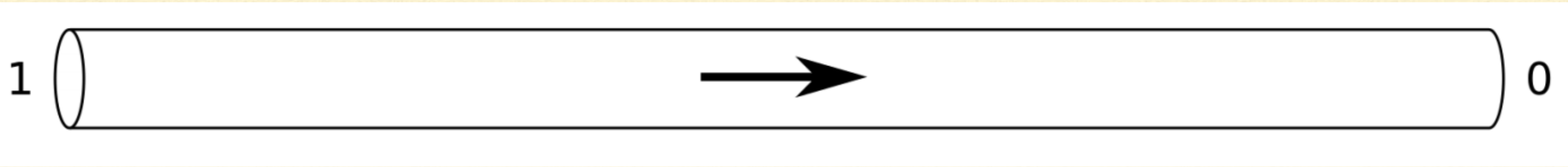
- **Inter-process communication**
- Allows different processes to communicate with each other.
- by using “pipe”.

DATA STREAM

- Every program will has one input stream and two output stream.
- STDIN default to the keyboards and STDOUT/STDERR default to terminal
- Piping is to connect them between programs.

PIPE CREATION

- Let's try some coding to test the piping.
- Pipe is **unidirectional**, i.e. only one direction of flow is allowed.
- One number for input, another for output.



PIPE CREATION

```c
/* pipe1.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char* argv[]) {
    int pipefds[2];
    char buf[30];
    //create pipe
    if (pipe(pipefds) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    //write to pipe
    printf("writing to file descriptor #%d\n", pipefds[1]);
    write(pipefds[1], "CSCI3150", 9);
    //read from pipe
    printf("reading from file descriptor #%d\n", pipefds[0]);
    read(pipefds[0], buf, 9);
    printf("read \"%s\"\n", buf);
    return 0;
}
```

To create pipe

PIPE CREATION

```
int pipe(int pipefd[2]);
```

To create pipe, return -1 when on error

```
ssize_t write(int fd, const void *buf, size_t count);
```

Write to a file descriptor

```
ssize_t read(int fd, void *buf, size_t count);
```

Read from a file descriptor

PIPE WITH FORK

- Let's create one more process and use a pipe to connect them.

PIPE WITH FORK

```c
int main(int argc, char* argv[]) {
    int pipefds[2];
    pid_t pid;
    char buf[30];
    //create pipe
    if(pipe(pipefds) == -1){
        perror("pipe");
        exit(EXIT_FAILURE);
}
  memset(buf,0,30);
  pid = fork();
if (pid > 0) {
  printf(" PARENT write in pipe\n");
//parent close the read end
      close(pipefds[0]);
//parent write in the pipe write end
  write(pipefds[1], "CSCI3150", 9);
//after finishing writing, parent close the write end
      close(pipefds[1]);
//parent wait for child
  wait(NULL);
}
```

PIPE WITH FORK

```
else {
      //child close the write end
      close(pipefds[1]);   //-----line *
      //child read from the pipe read end until the pipe is empty
      while(read(pipefds[0], buf, 1)==1)
        printf("CHILD read from pipe -- %s\n", buf);
      //after finishing reading, child close the read end
      close(pipefds[0]);
      printf("CHILD: EXITING!");
      exit(EXIT_SUCCESS);
    }
    return 0;
}
```

POINT TO NOTE ****

- Remember to close the pipes
  - when you finish writing/reading
  - that are NOT IN USE.

1. close the read side
2. write
3. close the write side

1. close the write side
2. read
3. close the read side

WHAT IF YOU FORGET TO CLOSE

Suppose the child forgets to close the write side…

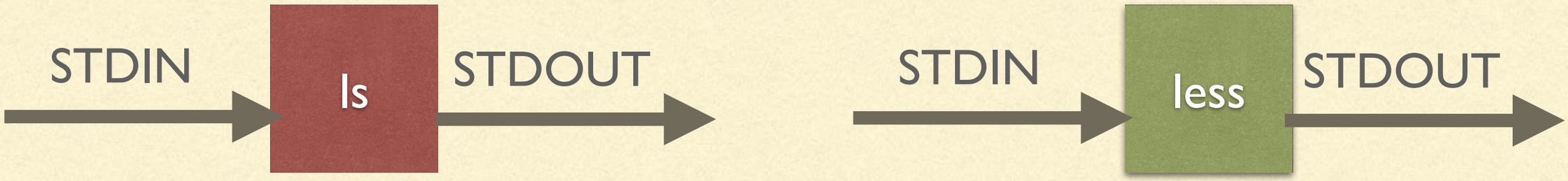
```
else {
      //child read from the pipe read end until the pipe is empty
    while(read(pipefds[0], buf, 1)==1)
      printf("CHILD read from pipe -- %s\n", buf);
    //after finishing reading, child close the read end
    close(pipefds[0]);
    printf("CHILD: EXITING!");
    exit(EXIT_SUCCESS);
  }
```

WHAT IF YOU FORGET TO CLOSE

- After fork(), the variables are cloned to the child.
- If the write side (pipefd[1]) is kept open, the OS assumes somebody will put data into the pipe at any time.
- It will block process reading from the read side (pipefd[0]) as the write is not complete.
- Then the process is blocked.

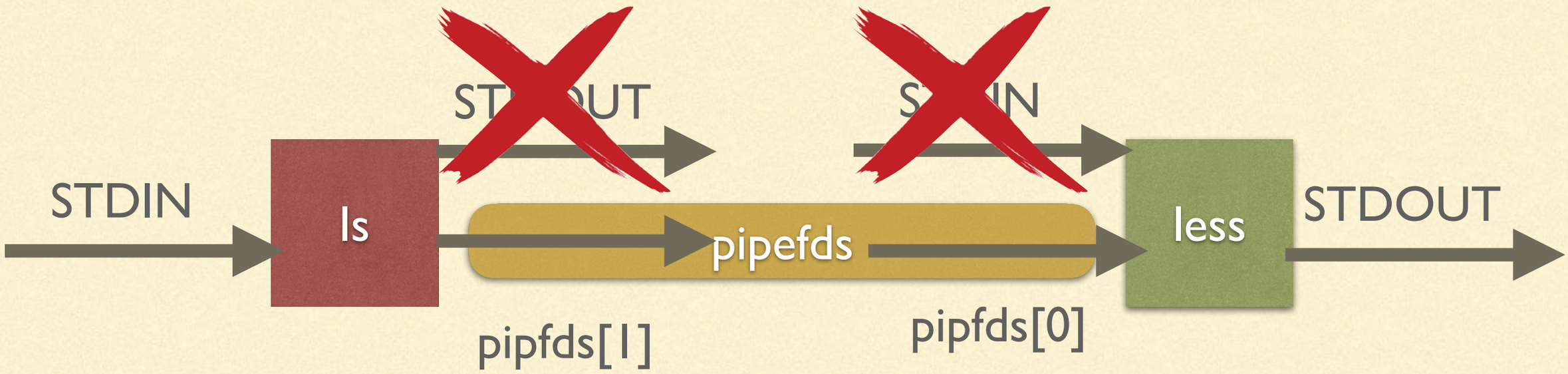
HOW TO IMPLEMENT "LS | LESS"?

- Let's try something advance. This time we redirect the input/ output of the processes.
- Suppose we have two processes running "ls" and "less".
- Normally they are connected by STDIN/STDOUT.



HOW TO IMPLEMENT "LS | LESS"?

- What we need to do is:
- Redirect the stdout of ls to pipefds[1], stdin of less to pipefd[0]



HOW TO IMPLEMENT "LS | LESS"?

- The function we need to redirect is "dup2"

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

- For example, `dup2(pipefds[1],STDOUT_FILENO);`
  - It is to redirect the STDOUT to write side of pipefds.
- Remember to CLOSE PIPES.

HOW TO IMPLEMENT "LS | LESS"?

- Initialize the pipe by pipe().

```c
int pipefds[2];
    pid_t pid,pid1;
int status;
if(pipe(pipefds) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
}
pid = fork();
if(pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
}
```

HOW TO IMPLEMENT "LS | LESS"?

- The first fork() creates the first child.
- Redirect stdin to the pipe, then close all, and exec "less"

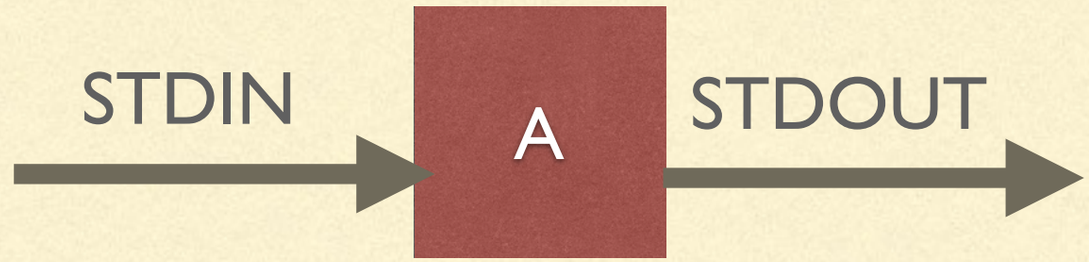
```
pid = fork();
    if(pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if(pid == 0) {//child
        close(pipefds[1]);
        dup2(pipefds[0],STDIN_FILENO);
        close(pipefds[0]);
        execlp("less","less",NULL);
    }
```

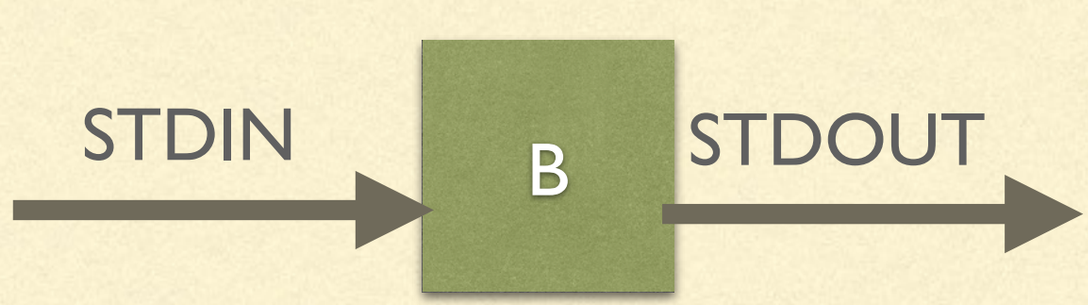
HOW TO IMPLEMENT "LS | LESS"?

- Then for the parent, fork again to create the second child.
- Redirect stdout to the pip, then close all and exec "ls"
- Finally wait for all children.

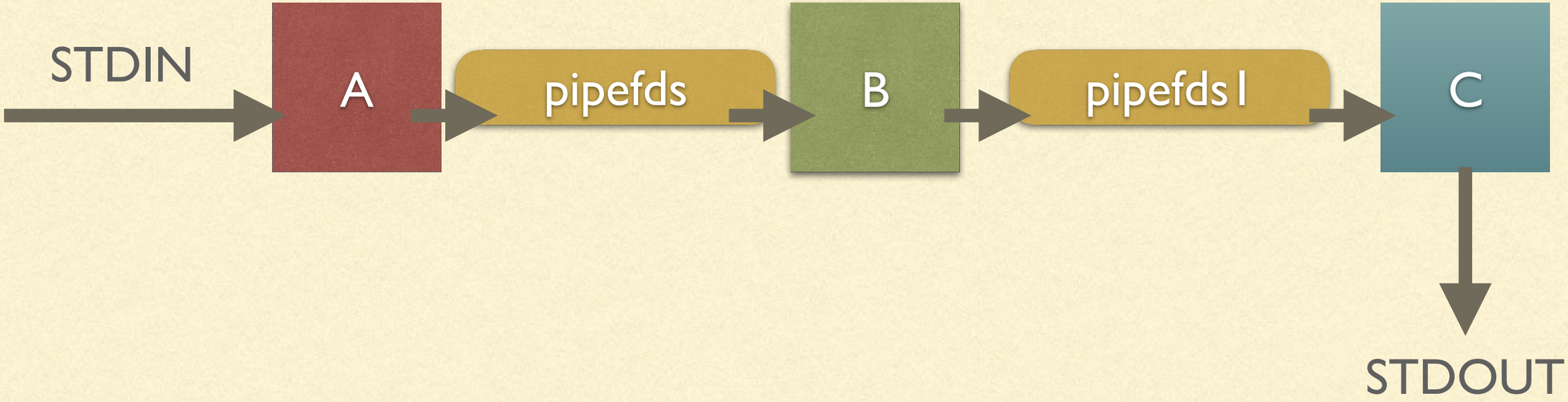
```
else { //parent
        pid1 = fork();
        if(pid1 == 0) { //child
            close(pipefds[0]);
            dup2(pipefds[1],STDOUT_FILENO);
            close(pipefds[1]);
            execlp("ls","ls",NULL);
        }
        close(pipefds[0]);
        close(pipefds[1]);
        waitpid(pid,&status,WUNTRACED);
        waitpid(pid1,&status,WUNTRACED);
    }
```

HOW ABOUT >=3 PROCESSES?

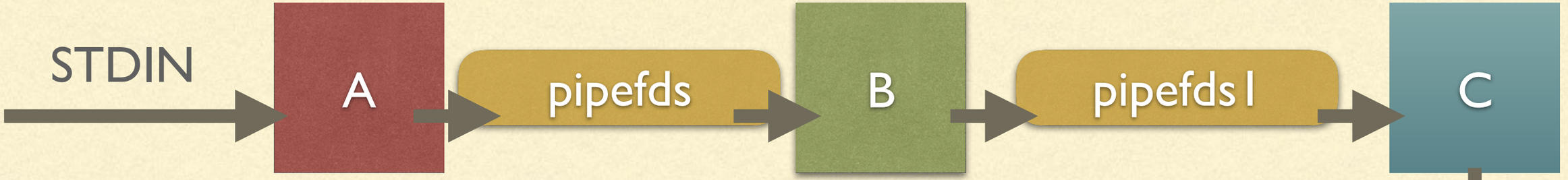




HOW ABOUT >=3 PROCESSES?



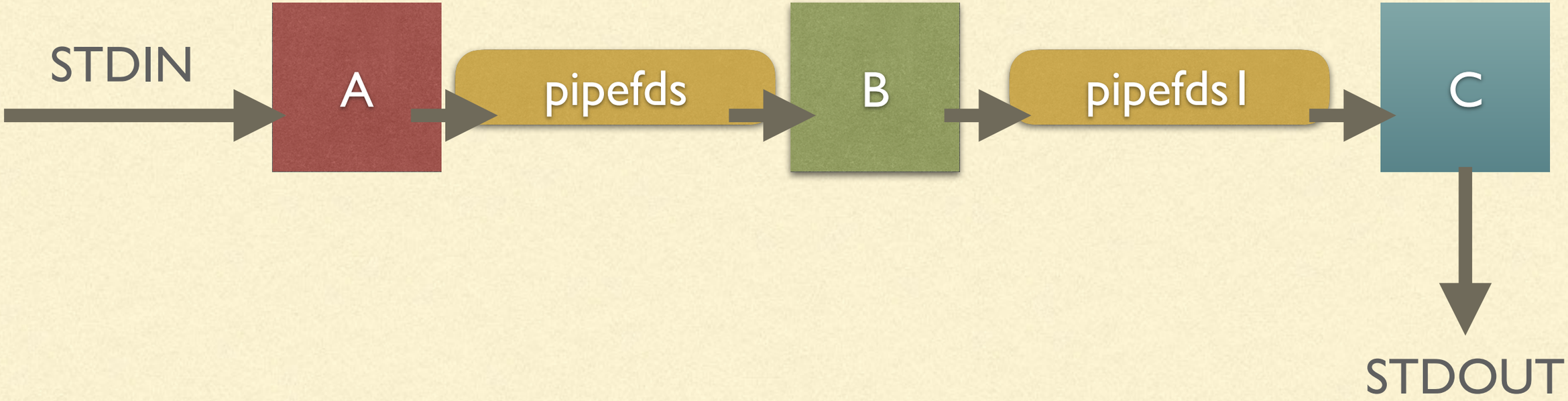
HOW ABOUT >=3 PROCESSES?



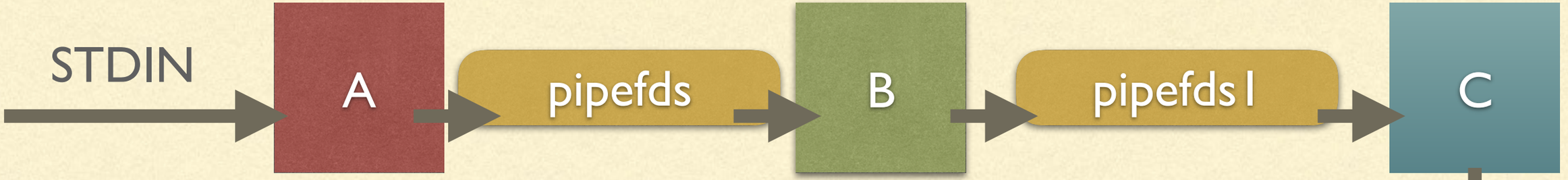

```
close(pipefds[0]);
dup2(pipefds[1],1);
close(pipefds[1]);
```

HOW ABOUT >=3 PROCESSES?

```
close(pipefds1[1]);
dup2(pipefds1[0],0);
close(pipefds1[0]);
```



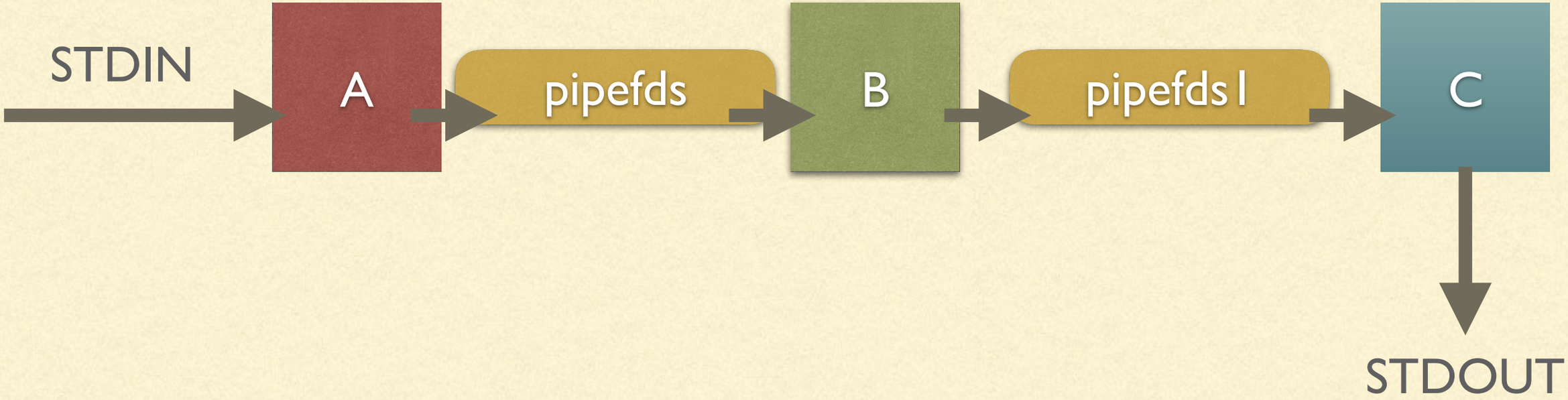
HOW ABOUT >=3 PROCESSES?




```
close(pipefds[1]);
close(pipefds1[0]);
dup2(pipefds[0],0);
dup2(pipefds1[1],1);
close(pipefds[0]);
close(pipefds1[1]);
```

HOW ABOUT >=3 PROCESSES?

Try to code it in C to implement ls | cat | cat



SUMMARY

- Pipe is a unidirectional data stream.
- Using pipe() to initialise a pipe.
- Read()/Write() to get from/put to the pipe.
- dup2() is for redirection of STDIN/STDOUT to pipe.