CSCI3150 LAB 5

Calvin Kam (hckam@cse) - 12 Oct 2017

AGENDA

- Inter-process communication
- Signals
- Useful functions for Assignment 1

WHAT IS IPC

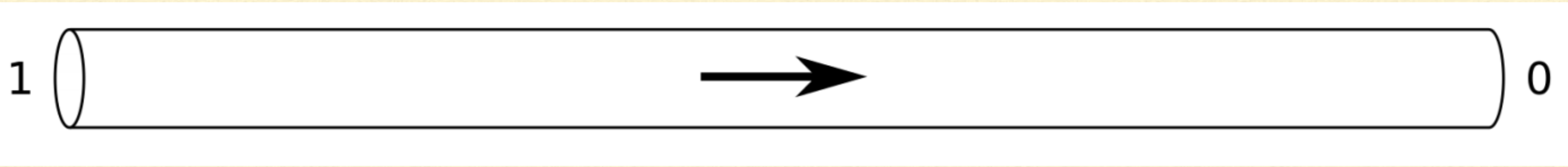
- **Inter-process communication**
- Allows different processes to communicate with each other.
- by using “pipe”.

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);
}
```

pipe2.c

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);
  }
```

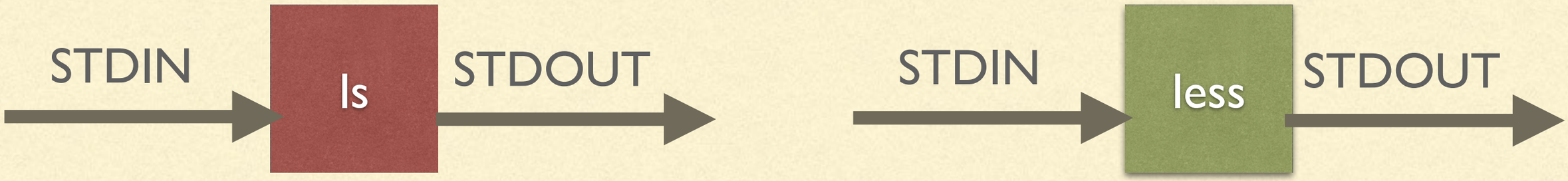
pipe3.c

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.

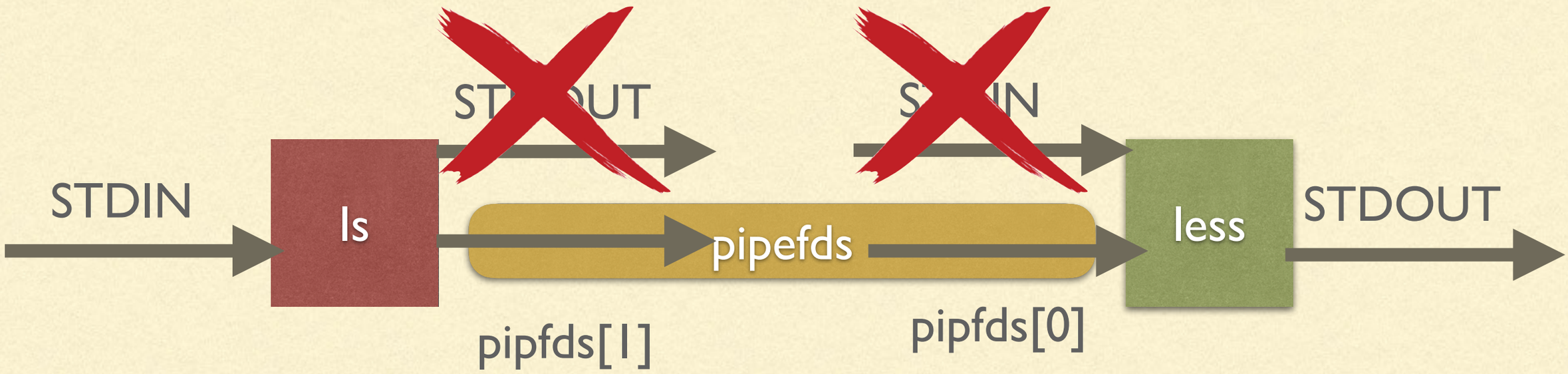
HOW TO IMPLEMENT "LS | LESS"?

- 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().

```
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"

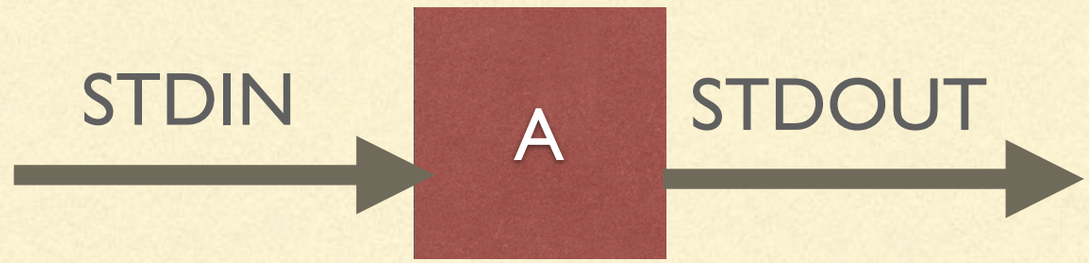
```
pid = fork();
    if(pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if(pid == 0) {//child
        close(pipefds[1]);
        dup2(pipefds[0],STDIN_FILENO);
        close(pipefds[0]);
        execlp("less","less",NULL);
    }
```

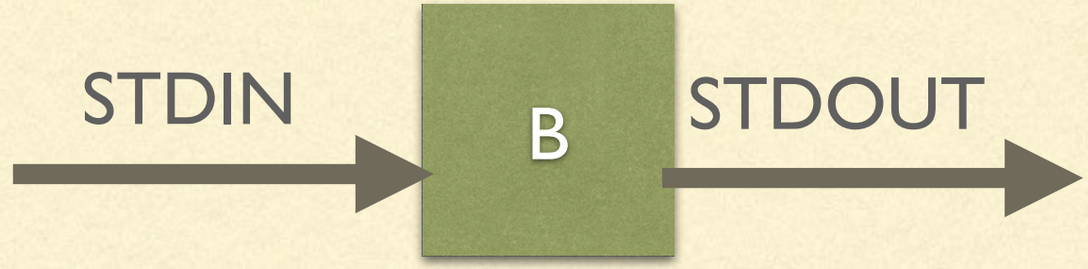
HOW TO IMPLEMENT "LS | LESS"?

- 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.

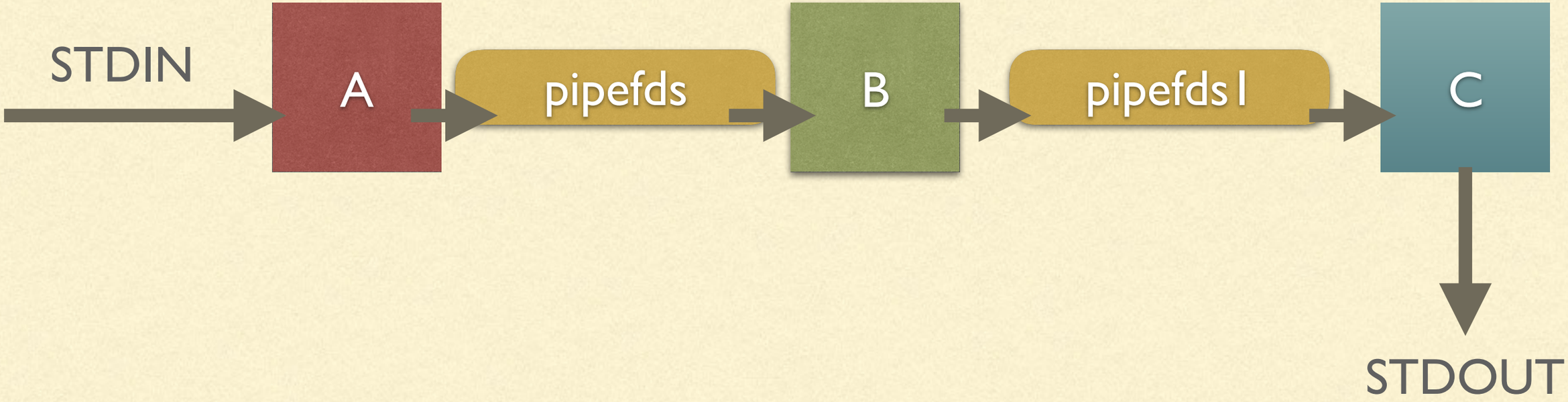
```
else { //parent
        pid1 = fork();
        if(pid1 == 0) { //child
            close(pipefds[0]);
            dup2(pipefds[1],STDOUT_FILENO);
            close(pipefds[1]);
            execlp("ls","ls",NULL);
        }
        close(pipefds[0]);
        close(pipefds[1]);
        waitpid(pid,&status,WUNTRACED);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

HOW ABOUT >=3 PROCESSES?

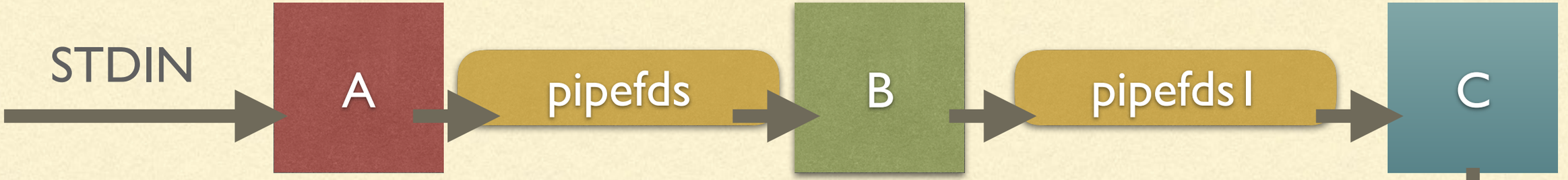




HOW ABOUT >=3 PROCESSES?



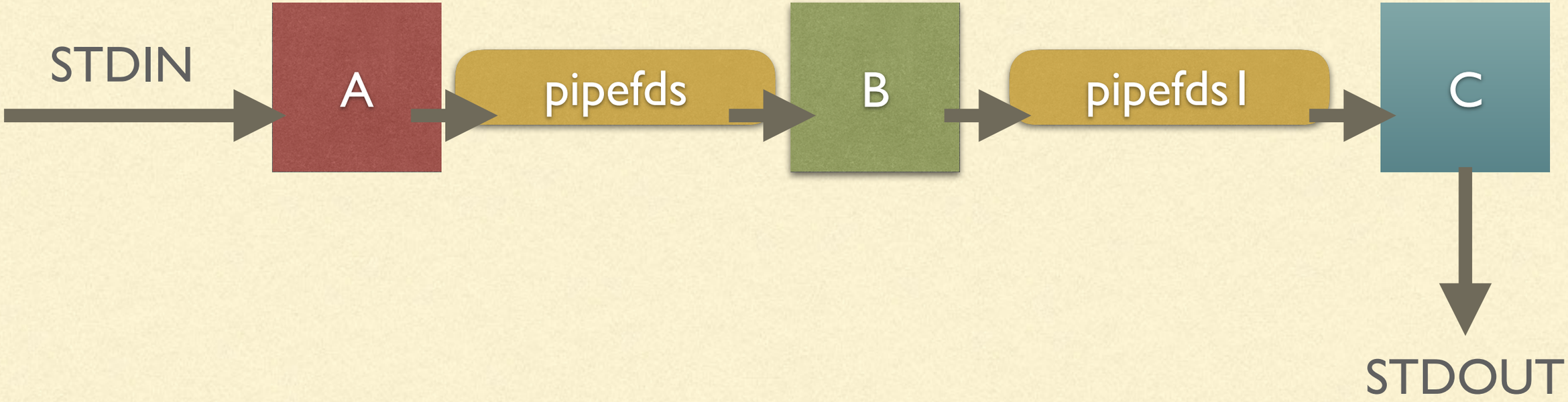
HOW ABOUT >=3 PROCESSES?



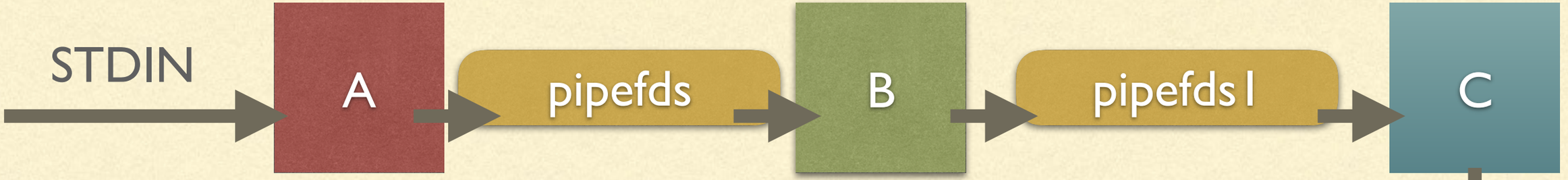

```
close(pipefds[0]);
dup2(pipefds[1],1);
close(pipefds[1]);
```

HOW ABOUT >=3 PROCESSES?

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



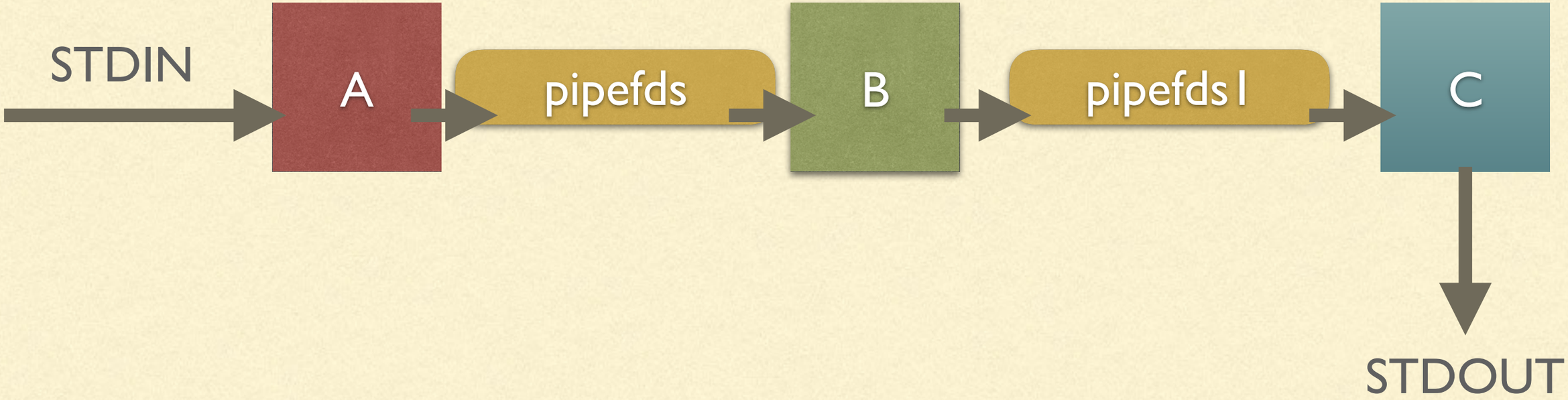
HOW ABOUT >=3 PROCESSES?




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

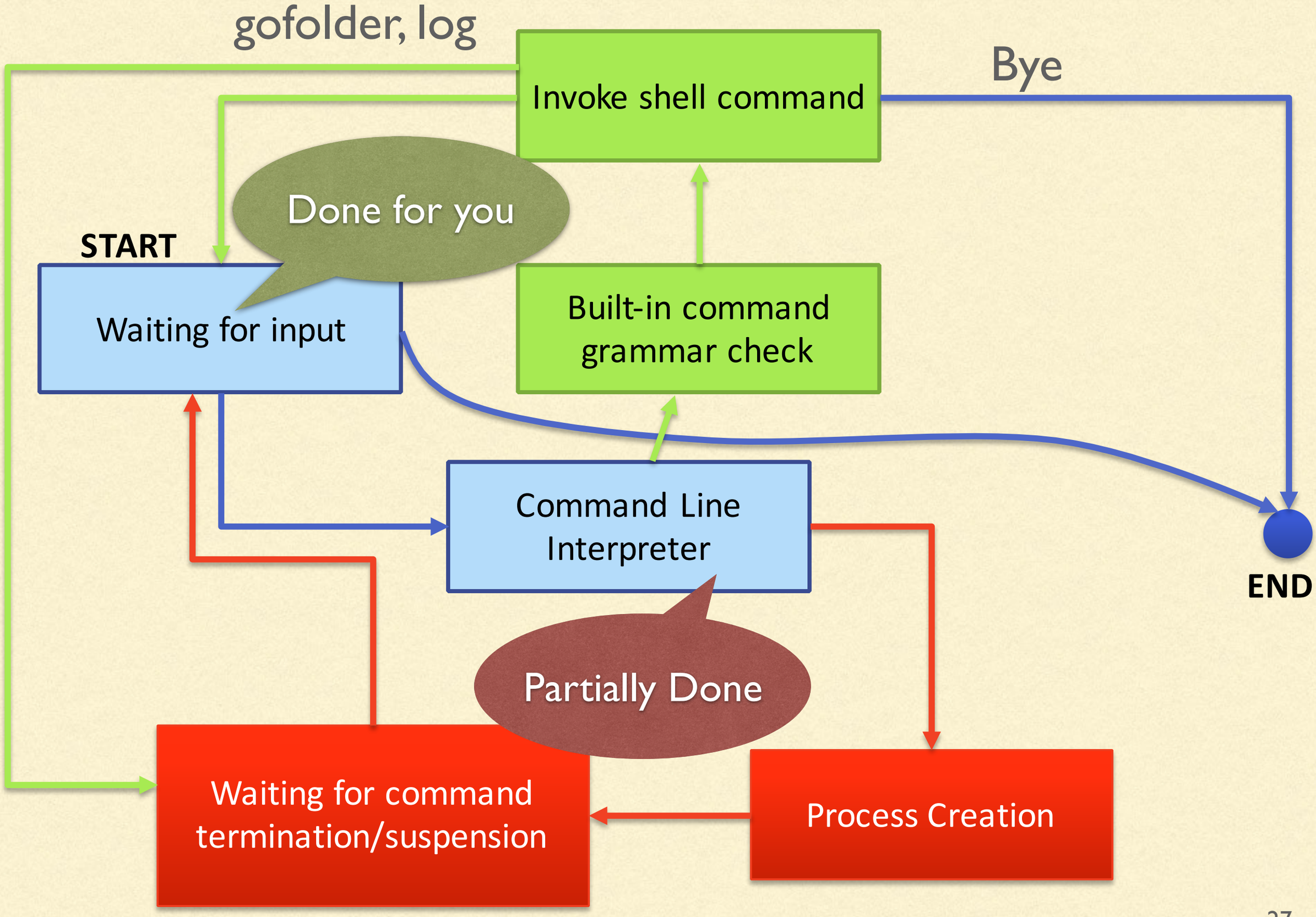
HOW ABOUT >=3 PROCESSES?

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



ASSIGNMENT 1

- Writing a Simple Shell
- Has following features:
  - Command Execution
  - Shell Commands
  - Signal Handling
  - Command Chaining using && and ||


gofolder, log
Invoke shell command
Bye
Done for you
START
Waiting for input
Built-in command
grammar check
Command Line
Interpreter
END
Partially Done
Waiting for command
termination/suspension
Process Creation

PROCESS CREATION: ERROR HANDLING

- If exec*() fails to invoke the program, it **WILL RETURN** to the original code and **CONTINUE** execution.
- Return Value: -1, errno is set.

PROCESS CREATION: ERROR HANDLING

```c
/* Exec/exec_error.c */
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
int main(int argc,char *argv[])
{
    printf("Try to execute lss\n");
    execl("/bin/lls","lls",NULL);
    printf("execl returned! errno is [%d]\n",errno);
    perror("The error message is :");
    return 0;
}
```

WAITPID()

- waitpid() is an advanced version of wait().
- It can tell you more about status of the children.

```
pid_t waitpid(pid_t pid, int *status, int options);
```

PID that you wants to wait

Pointer that stores the status of the child

Behaviour of waitpid()

WAITPID()

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
int main(int argc,char *argv[]) {
    int pid;
    int status;
    if(!(pid = fork())) {
        printf("My PID: %d\n",getpid());
        exit(0);
    }
    waitpid(pid,&status,WUNTRACED);
    if(WIFEXITED(status))  {
        printf("Exit Normally\n");
        printf("Exit status: %d\n",WEXITSTATUS(status));
    }
    else  {
        printf("Exit NOT Normal\n");
    }
    return 0;}
```

WAITPID()

- waitpid() has several macros for showing children's status.
- WIFEXITED(status) is to check whether the child exits normally (by exit() or return) .
- WEXITSTATUS(status) is to get the exit status (return value) of the child.

SIGNALS

- Signals are interrupts sent to the process.
- If custom signal handlers are not defined or not changed to ignore, default signal handler will be used.
- Eg: SIGSEGV(SEG Fault), SIGINT (Ctrl-C), **SIGTSTP** (Ctrl-Z), **SIGCHLD**, **SIGTERM**, **SIGKILL**, and etc.
- We use kill() to send out signals (not just kill the process!).
- BTW, **EOF** (Ctrl-D) is **NOT** signal

CUSTOM SIGNAL ROUTINES

- We can let the process behaves differently upon different signals.
- Can set them to ignore or even custom user-level handler.
- Of course, we cannot do anything on **SIGKILL** (unstoppable).

CUSTOM SIGNAL ROUTINES

```
#include <stdio.h>
#include <signal.h>

int main(int argc,char *argv[])
{
    signal(SIGINT,SIG_IGN);
    printf("Put into while 1 loop..\n");
    while(1) { }
    printf("OK!\n");
    return 0;
}
```

SIG_IGN: Ignore

CUSTOM SIGNAL ROUTINES

```c
/* Signals/custom.c */
#include <stdio.h>
#include <signal.h>

void handler(int signal)
{
    printf("Signal %d Received.Kill me if you can\n",signal);
}

int main(int argc,char *argv[])
{
    signal(SIGINT,handler);
    printf("Put into while 1 loop..\n");
    while(1) { }
    printf("OK!\n");
    return 0;
}
```

GET CURRENT PATH

- In the prompt, it shows the current working directory.
- We can use a getcwd() to get it easily.

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

GETCWD()

```
#include <stdio.h>
#include <limits.h> // Needed by PATH_MAX
#include <unistd.h> // Needed by getcwd()
int main(int argc,char *argv[]){
    char cwd[PATH_MAX+1];
     if(getcwd(cwd,PATH_MAX+1) != NULL){
         printf("Current Working Dir: %s\n",cwd);
}
    else{
        printf("Error Occured!\n");
    }
 return 0;
}
```

CHANGE DIRECTORY

- To change the working directory, we can use the following function.
- “gofolder” in your assignment.

```
#include <unistd.h>
int chdir(const char *path);
```

CHDIR()

```c
/* Shell/chdir.c */
#include <stdio.h>
#include <unistd.h>
#include <limits.h>
#include <errno.h>
#include <string.h>
int main(int argc,char *argv[]) {
    char buf[PATH_MAX+1];
    char input[255];
    if(getcwd(buf,PATH_MAX+1) != NULL) {
        printf("Now it is %s\n",buf);
        printf("Where do you want to go?:");
        fgets(input,255,stdin);
        input[strlen(input)-1] = '\0';
        if(chdir(input) != -1) {
            getcwd(buf,PATH_MAX+1);
            printf("Now it is %s\n",buf);       }
        else {
            printf("Cannot Change Directory\n"); }
    }
return 0;
}
```

Error Checking

CHANGE ENVIRONMENT VARIABLE

- In order to setup a searching sequence for shell, we need to change the $PATH variable.
- You can either provide a new set of environment variables in some of exec*() family members or using this function:

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite);
```

SETENV()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
int main(int argc,char *argv[])
{
char *command1[] = {"shutdown",NULL};
printf("Running shutdown.. it is in /sbin :P \n\n");
setenv("PATH","/bin:/usr/bin:.",1);
execvp(*command1,command1);
 if(errno == ENOENT)
printf("No Command found...\n\n");
else     printf("I dont know...\n");
 return 0; }
```

Need To Overwrite

COMMAND CHAINING

- In this assignment, you are required to chain commands by AND (&&) and OR (||).
- These are very useful in shell script programming.
- A (Operator) B.
  - Execution of B depends on the exit status of A.
  - Runs successfully = Exit Normally with Exit Status 0.
- In assignment we won't test you for chaining many commands, but of course try take the challenges ;P

AND &&

- A && B
- If A runs successfully, then B will run.
- If A fails, then B will NOT run.
- Usage:
  - Doing a series of jobs of which the subsequent job only runs if the previous one runs successfully.
  - mkdir abc && cd abc

OR (||)

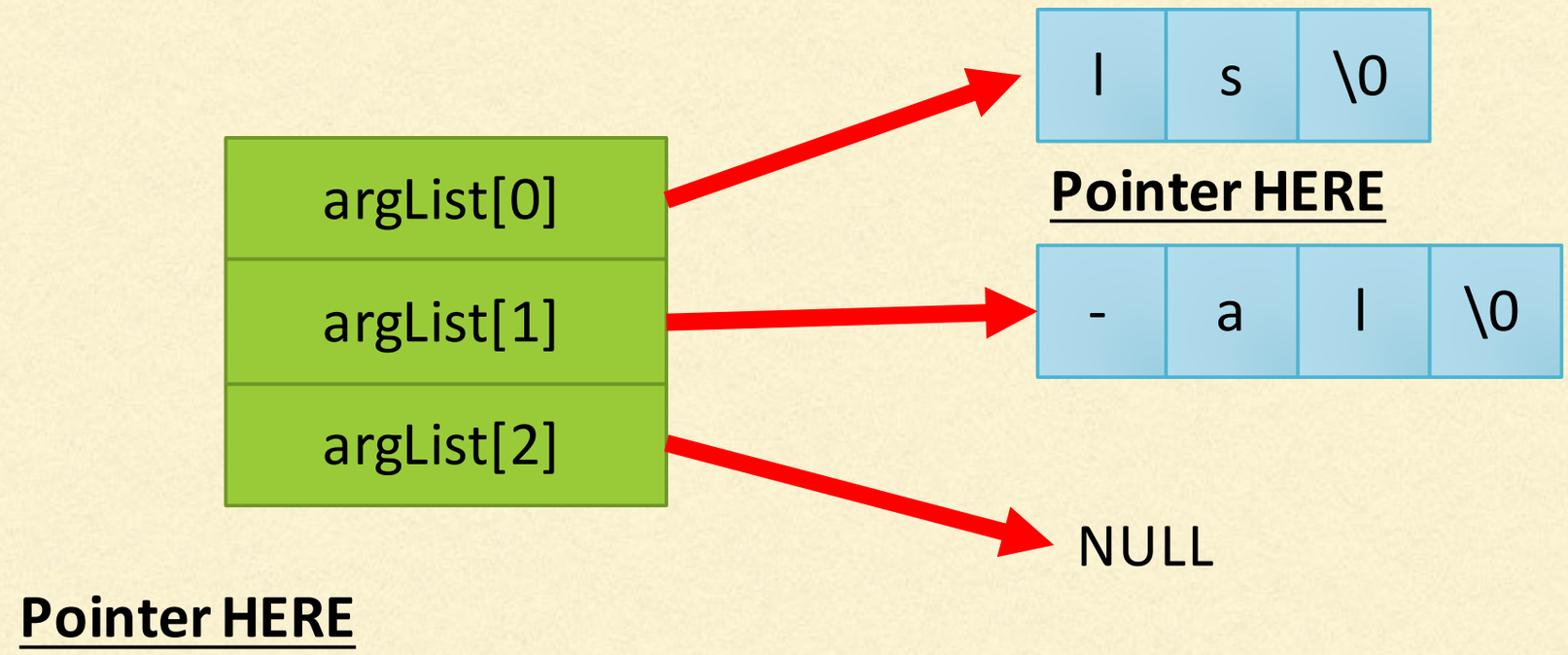
- A || B
- If A runs successfully, B will NOT run.
- If A fails, then B will run.
- Usage:
  - Error Reporting.
  - rm abc && echo “Error”.

DATA STRUCTURE

- Remember your data structure class :P
- It is good to store your commands in a manageable data structure.
- Multi-dimensional arrays
- Linked List, vector
- and etc….

ARGUMENT ARRAY

- In some exec*() members, you need to provide an argument array.
- Actually it is an array of pointers.



ARGUMENT ARRAY

- As we have to access two pointers in order to get to the string, dereferencing twice (malloc() two times) are needed.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc,char *argv[])
{
    char **argList = (char**) malloc(sizeof(char*) * 3);
    argList[0] = (char*)malloc(sizeof(char) * 10);
    strcpy(argList[0],"ls");
    argList[1] = (char*)malloc(sizeof(char) * 10);
    strcpy(argList[1],"-al");
    argList[2] = NULL;

    execvp(*argList,argList);
   return 0;
}
```

SUMMARY

- Inter-process communication
- Signals
- Useful functions for Assignment 1