KOÇ UNIVERSITY

COMP304
Assignment 1
Problem Session

Fareed Qararyah
fqararyah18@ku.edu.tr

Koç University, Istanbul, Turkey

Assignment 1

- Interprocess communication using **ordinary pipes**
- **Kernel module**

Ordinary pipes

Allow process communication in producer-consumer fashion

- Producer process writes to one end of pipe
- Consumer process reads from the other end

- Using pipes
  - Define integer array of size 2 to hold file descriptors

```
int fd[2];
```

  - Construct pipe using `pipe` function (defined in `unistd.h`)
    - returns -1 if unsuccessful

```
pipe(fd);
```

Ordinary pipes

  - fd[0] refers to read end, fd[1] to write end
  - If process will write to the pipe, close reading end and vice versa
  - Write to the pipe using `write` function and fd[1]
  - Read from the pipe using `read` function and fd[0]
- Ordinary pipe can be accessed by process who created it and its children processes
- Details of using pipes in Chapter 3 (9'th Edition)

Example

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1
int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings\n";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;
/* create the pipe */
if (pipe(fd) == -1) {
fprintf(stderr,"Pipe failed");
return 1;
}
pid = fork(); /* fork a child process */
if (pid < 0) { fprintf(stderr, "Fork Failed"); return 1;}/* error occurred */

if (pid > 0) { /* parent process */
close(fd[READ_END]); /* close the unused end of the pipe */
write(fd[WRITE_END], write_msg, strlen(write_msg)+1); /* write to the pipe */
close(fd[WRITE_END]);/* close the write end of the pipe */
}
else { /* child process */
close(fd[WRITE_END]); /* close the unused end of the pipe */
read(fd[READ_END], read_msg, BUFFER_SIZE);/* read from the pipe */
printf("read %s",read_msg);/* close the write end of the pipe */
close(fd[READ_END]);
}
return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: bash

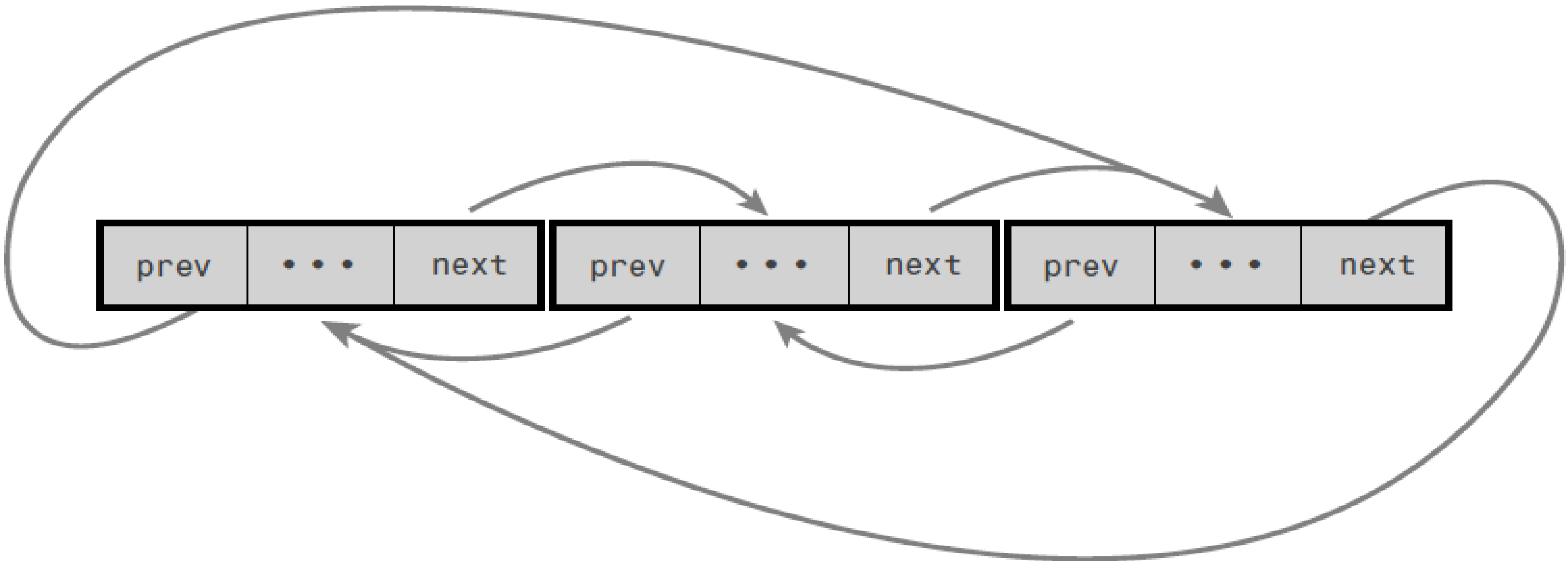
```
fareed@ubuntu:~/ass1/prob2$ gcc example_1.c -o example_1
fareed@ubuntu:~/ass1/prob2$ ./example_1
read Greetings
fareed@ubuntu:~/ass1/prob2$
```

Process Descriptor and the Task Struct

- Processes are more than just the executing program code (often called the text section in Unix). They also include a set of resources
  - Open files
  - Processor state
  - An address space
  - A data section containing global variables
  - …
- In Linux **process descriptor** holds these details in struct *task_struct* (<linux/sched.h>)

Task List

- In Linux
  - Task list is a Circular doubly linked list



- Each element in the task list is a process descriptor of the type struct *task_struct*

Process Family Tree

- All processes are descendants of the *init* process, whose PID is one
- Every process on the system has exactly one parent. Likewise, every process has zero or more children.
- Processes that are all direct children of the same parent are called *siblings*
- Each *task_struct* has a pointer to
  - the parent's *task_struct,* named parent
  - a list of children, named children.

Process Family Tree

- Given the current process, it is possible to obtain the process descriptor of its parent with the following code
  - struct task_struct *my_parent = current->parent;
- Similarly, it is possible to iterate over a process's children with
  - struct task_struct *task;
  - struct list_head *list;
  - list_for_each(list, &current->children) {

    task = list_entry(list, struct task_struct, sibling); /* task now points to one of current's children */

    }

THANK YOU