

Vistula, IT Faculty, 2016

# **Operating Systems & Systems Programming**

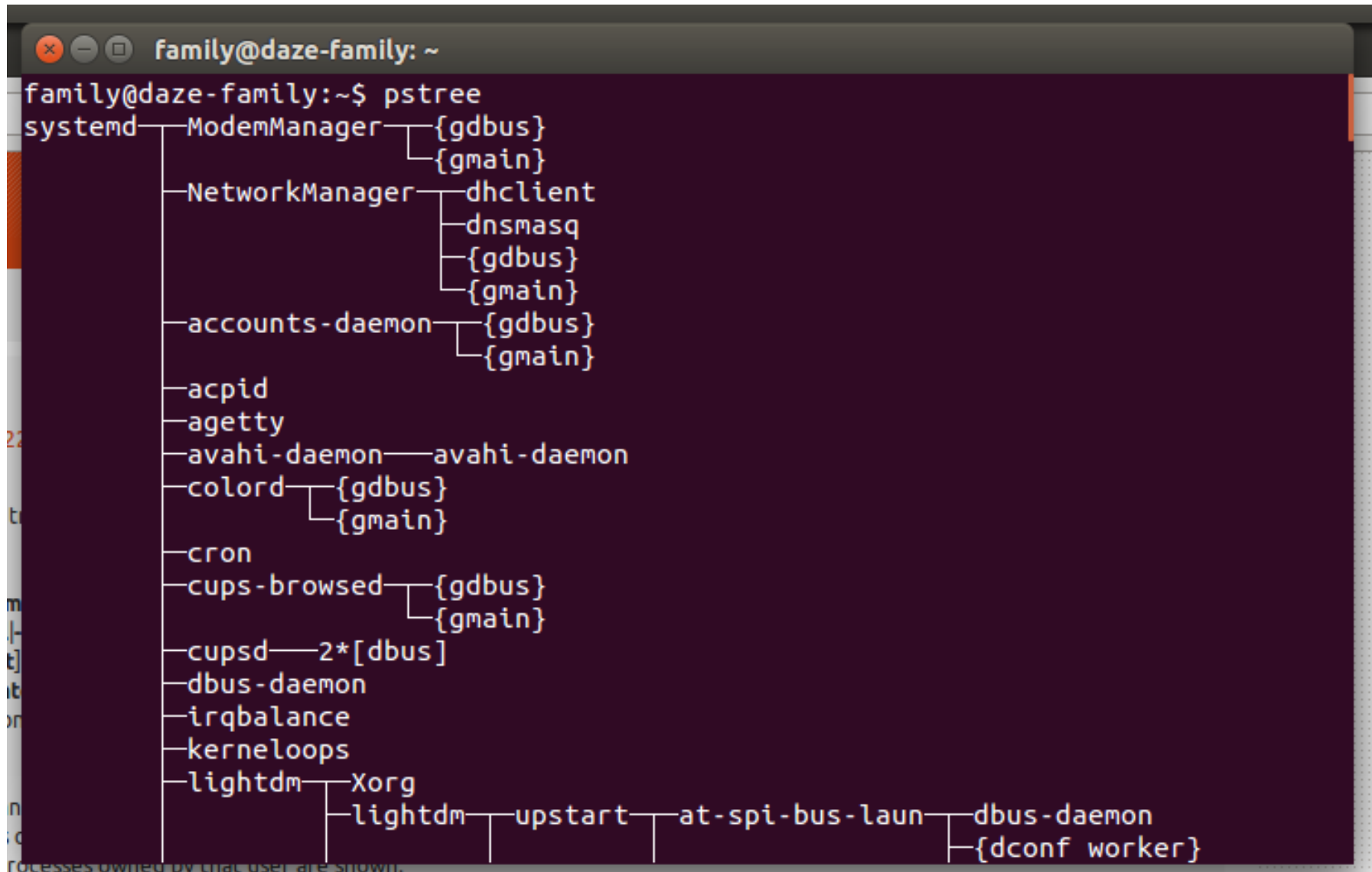
Dmitry A. Zaitsev

<http://daze.ho.ua>

## **Lecture 3:**

# **Creation of Processes in OS Linux. Inter Process Communication: Pipes**

# Tree of processes



# Create a child process

- `#include <unistd.h>`
- `pid_t fork(void);`
- Duplicates the calling process

```
pid_t pid;
```

```
pid=fork();
```

```
if(pid==0) { // Child }
```

```
else { // Parent }
```

# Execute a file

- `int execl(const char *path, const char *arg, ...  
/* (char *) NULL */);`
- `int execv(const char *path, char *const  
argv[]);`

# Wait for process to change state

- `#include <sys/types.h>`  
`#include <sys/wait.h>`
- `pid_t wait(int *wstatus);`
- `pid_t waitpid(pid_t pid, int *wstatus, int options);`
- `int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);`

# More process functions

- Send signal to a process

```
#include <sys/types.h>
```

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

```
int kill(pid_t pid, int sig);
```

- terminate the calling process

```
#include <unistd.h>
```

```
void _exit(int status);
```

# Fork & Execl example

```
pid_t pid;  
pid=fork();  
if(pid==0)  
{ // Child  
    execl("myprog", "myprog", 1, NULL);  
}  
// Parent  
wait();
```

# Threads

- `#include <pthread.h>`
- Link `-lpthread`
- create a new thread

```
int pthread_create(pthread_t *thread, const  
pthread_attr_t *attr, void *(*start_routine)  
(void *), void *arg);
```

- terminate calling thread

```
void pthread_exit(void *retval);
```



# Inter Process Communication

- Sharing resources
- Problem of critical section
- Demand for mutual exclusion facilities
- Solutions
- Inter Process Communication (IPC): flags of events, semaphores, pipes, messages, common memory segments etc

# Overview of pipes and FIFOs

- Pipes and FIFOs (also known as named pipes) provide a unidirectional interprocess communication channel.
- A pipe has a read end and a write end.
- Data written to the write end of a pipe can be read from the read end of the pipe.
- The communication channel provided by a pipe is a byte stream: there is no concept of message boundaries.

# Create pipe

- `#include <unistd.h>`
- `int fd[2];`
- `int pipe(int pipefd[2]);`
- `int pipe2(int pipefd[2], int flags);`
- Flags: `O_NONBLOCK`, `O_DIRECT`, `O_CLOEXEC`

# Working with pipes

- Write to a pipe

```
write(fd[1], message, sizeof(message));
```

- Read from a pipe

```
read (fd[0], message, sizeof(message));
```

```

#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#define SHELL "/bin/sh"
int my_system (const char *command)
{
    int status;
    pid_t pid;
    pid = fork ();
    if (pid == 0)
    {
        execl (SHELL, SHELL, "-c", command, NULL);
        _exit (EXIT_FAILURE);
    }
    else if (pid < 0)
        status = -1;
    else if (waitpid (pid, &status, 0) != pid)
        status = -1;
    return status;
}

```

## Running Shell Commands

```

int main()
{
    char cmd[256];
    while(1)
    {
        printf("input command: ");
        scanf("%s",cmd);
        my_system(cmd);
    }
}

```

# Pipes

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#define MSGLEN 64
int main(){
    int  fd[2];
    pid_t pid;
    int  result;
    result = pipe (fd);
    if (result < 0) {
        perror("pipe");
        exit (1);
    }
    pid = fork();
    if (pid < 0) {
        perror ("fork");
        exit(2);
    }
}
```

```
if (pid == 0) {
    char message[MSGLEN];
    while(1) {
        memset (message, 0, sizeof(message));
        printf ("Enter a message: ");
        scanf ("%s",message);
        write(fd[1], message, strlen(message));
    }
    exit (0);
}
else {
    char message[MSGLEN];
    while (1) {
        memset (message, 0, sizeof(message));
        read (fd[0], message, sizeof(message));
        printf("Message entered %s\n",message);
    }

    exit(0);
}
}
```