University of Wisconsin-Stout
CS 442, Operating Systems,
Saleh M. Alnaeli, Ph.D.
Fall, 2019

CS442, UW-Stout

System Calls and Programming in Unix/Linux

fork() , wait(), execvp, and more

1

Review

- Modes of Operation
- Processes
- System Calls
- Today
  - System Calls Examples
  - Inter-process Communications (IPC)

2

Modes of Operation

User Mode

- User program executes in user mode
- Certain areas of memory are protected from user access
- Certain instructions may not be executed

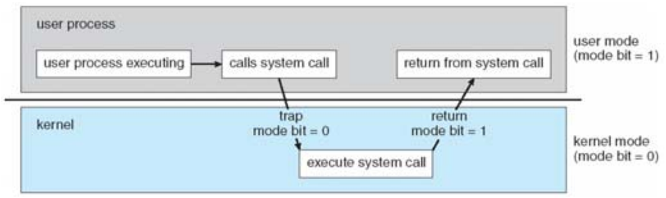
Kernel Mode

- Monitor executes in kernel mode
- Privileged instructions may be executed
- Protected areas of memory may be accessed

3

Dual-mode (user and kernel mode)

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user

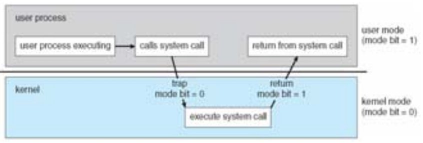


A **trap** (or exception) is a software generated interrupt caused either by an error (e.g. divide by 0 or invalid mem access) or by a specific request from a user program that an OS service be performed

4

Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
  - Timer is set to interrupt the computer after some time period
  - Keep a counter that is decremented by the physical clock.
  - Operating system set the counter (privileged instruction)
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time
- why the kernel will take control from a user?
- 3 Reasons
  - Interrupts
    - Triggered by timer and I/O devices
  - Exceptions
    - Triggered by unexpected program behavior
    - Or malicious behavior!
  - System calls (aka protected procedure call)
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points



5

System calls (aka protected procedure call)

- ✓ Request by program for kernel to do some operation on its behalf (**voluntarily**)
- ✓ A system call is any procedure provided by the kernel that can be called from user-level.
- ✓ Operating systems provide a substantial number of system calls.
  - o e.g., establish a connection to a web server,
  - o to send or receive packets over the network,
  - o to create or delete files, to read or write data into files, and
  - o to create a new user process.
  - o To the user program, these are called just like normal procedures, with parameters and return values.

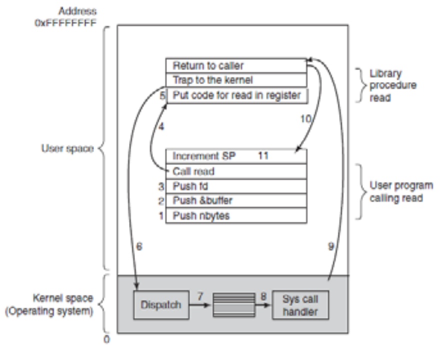


Figure. The 11 steps in making the system call *read(fd, buffer, nbytes).*

6

System Calls (2)

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Figure. Some of the major POSIX system calls. The return code *s* is −1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time.

7

System Calls (3)

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Figure. Some of the major POSIX system calls. The return code *s* is −1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time.

8

System Calls (4)

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Figure. Some of the major POSIX system calls. The return code *s* is −1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time.

9

System Calls (5)

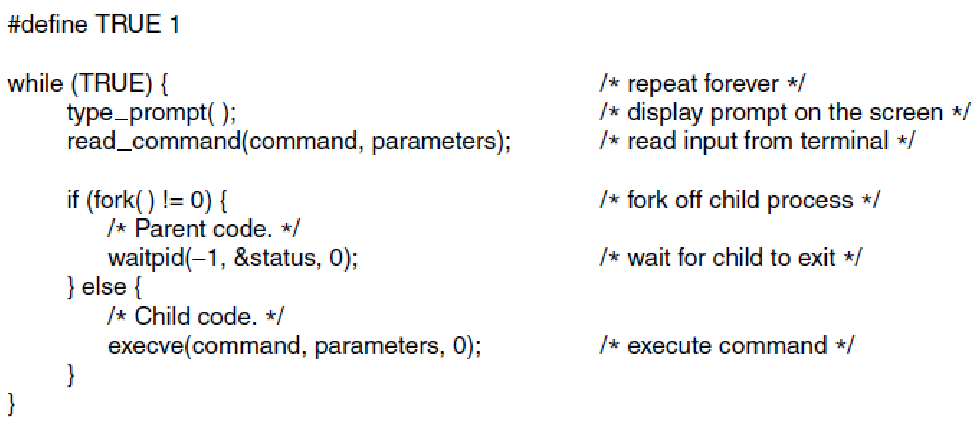
Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figure. Some of the major POSIX system calls. The return code *s* is −1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time.

10

System Calls for Process Management

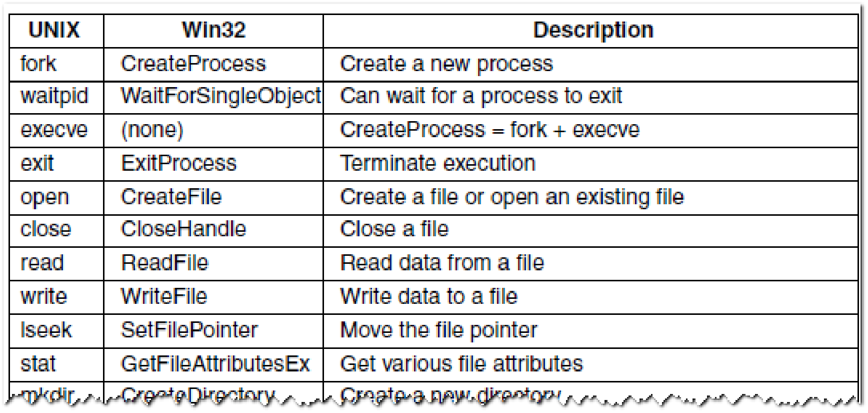


```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
     type_prompt( );                            /* display prompt on the screen */
     read_command(command, parameters);         /* read input from terminal */

     if (fork( ) != 0) {                        /* fork off child process */
         /* Parent code. */
         waitpid(−1, &status, 0);               /* wait for child to exit */
     } else {
         /* Child code. */
         execve(command, parameters, 0);        /* execute command */
     }
}
```

Figure. A stripped-down shell. Throughout this book, *TRUE* is assumed to be defined as 1.

11

The Windows Win32 API (1)

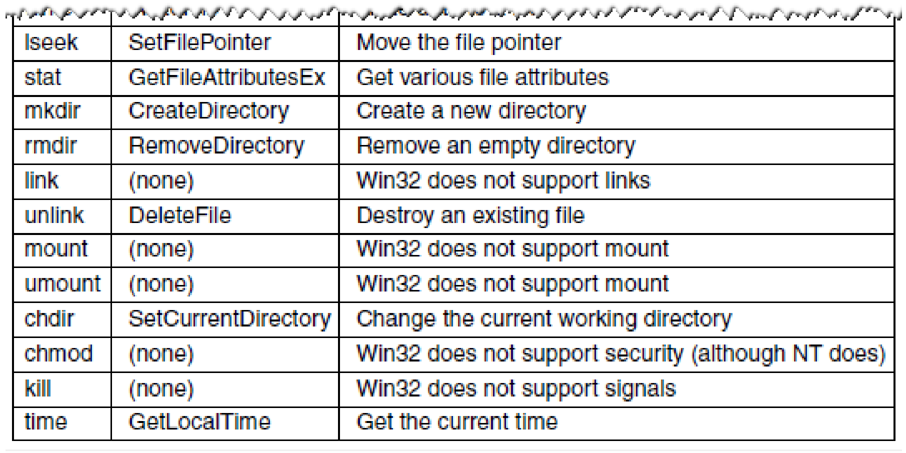


UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory

Figure. The Win32 API calls that roughly correspond to the UNIX calls of Fig. 1-18.

12

The Windows Win32 API (2)



lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Figure. The Win32 API calls that roughly correspond to the UNIX calls of Fig. 1-18.

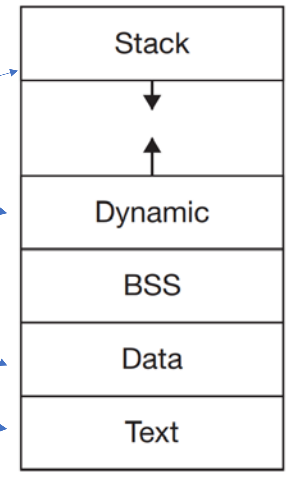
13

Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- I usually use the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

14

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
      int i;
      int current = 2;
      nprimes = atoi(argv[1]);
      prime = (int *)malloc(nprimes*sizeof(int));
      prime[0] = current;
      for (i=1; i<nprimes; i++) {
                                          int j;
            NewCandidate:
      current++;
      for (j=0; prime[j]*prime[j] <= current; j++) {
            if (current % prime[j] == 0)
                        goto NewCandidate;
            }
                        prime[i] = current;
            }
return(0);
}
```



Unix Address Space

15

15

fork() system calls

SYNOPSIS:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

DESCRIPTION:

fork() system call is used for creating a new process (child) by duplicating the calling process (parent).

RETURN VALUE:

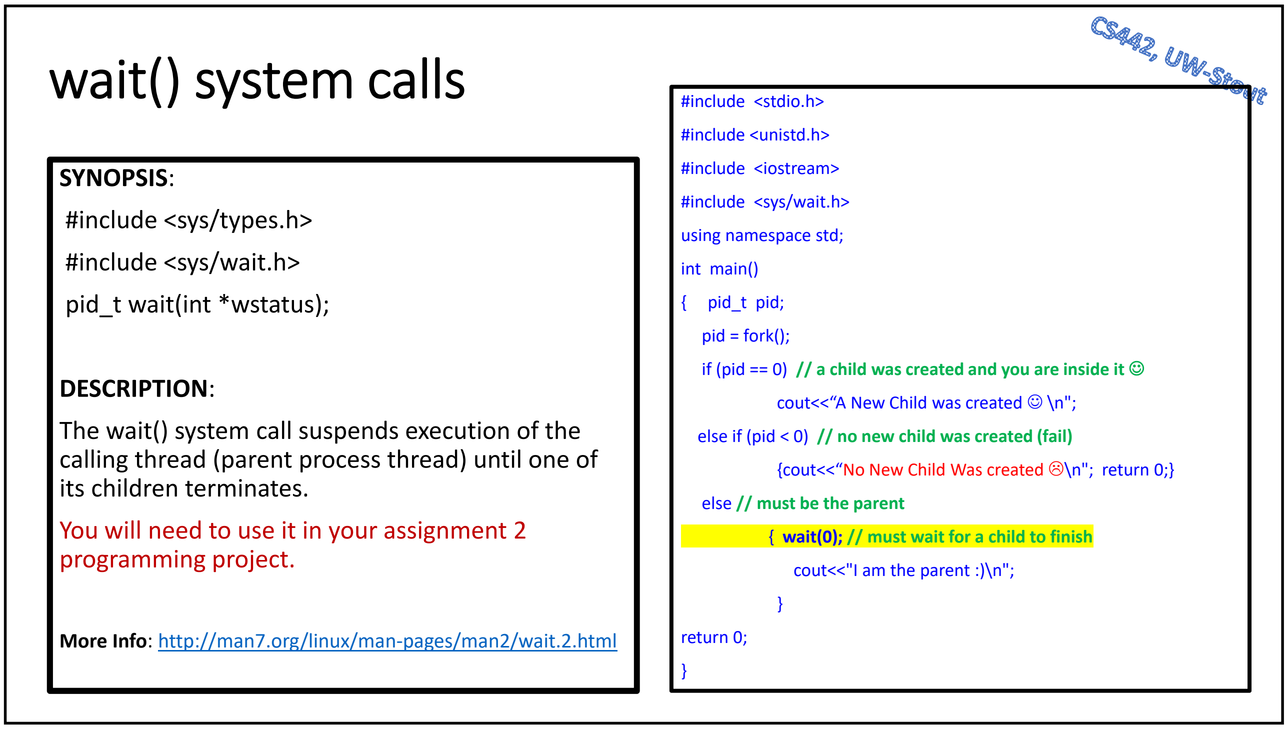
On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

You will need to use it in your assignment 2 programming project.

More Info: http://man7.org/linux/man-pages/man2/fork.2.html

```
#include  <stdio.h>
#include <unistd.h>
#include  <iostream>
using namespace std;
int  main()
{    pid_t  pid;
   pid = fork();
   if (pid == 0)  // means you are inside the child process
              cout<<"A New Child was created ☺ \n";
  else if (pid < 0)  // no new child was created (failed)
              {cout<<"No New Child Was created ☹\n";  return 0;}
   else // means you are inside the parent process
              cout<<"I am the parent :)\n";
return 0;
}
```

16



wait() system calls

SYNOPSIS:

#include <sys/types.h>

#include <sys/wait.h>

pid_t wait(int *wstatus);

DESCRIPTION:

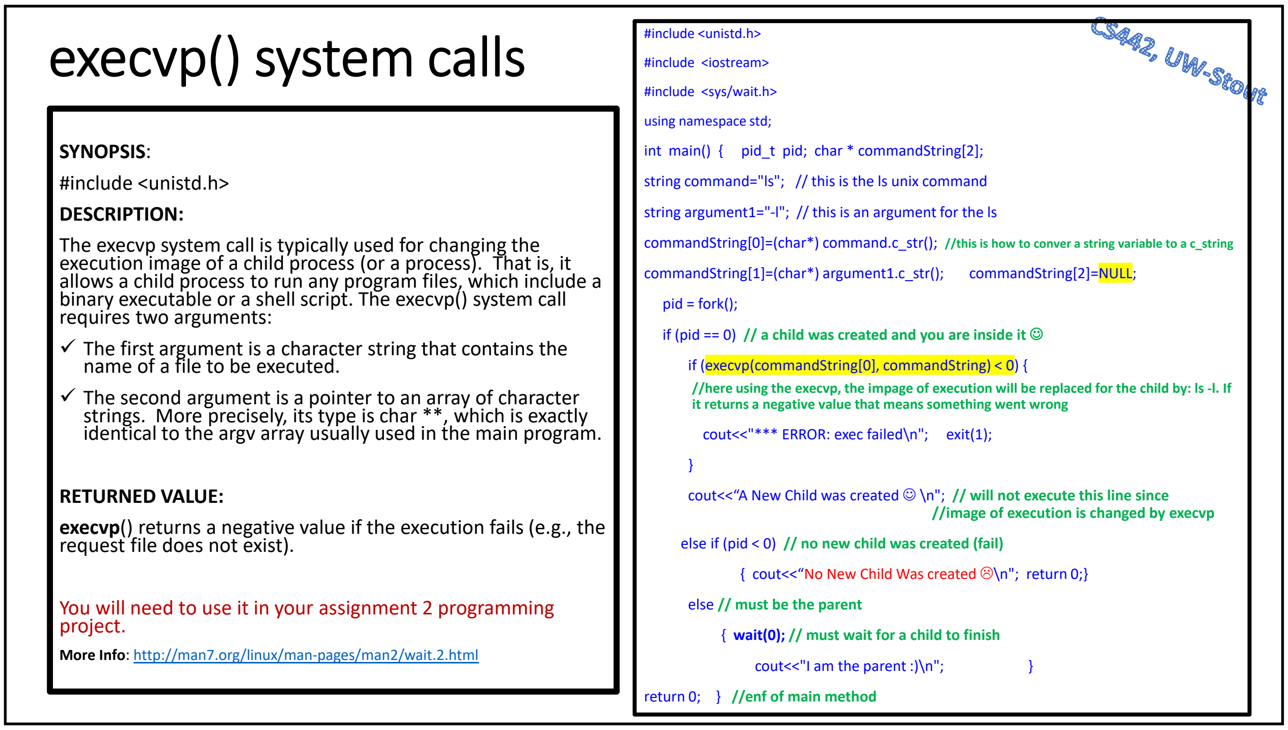
The wait() system call suspends execution of the calling thread (parent process thread) until one of its children terminates.

You will need to use it in your assignment 2 programming project.

More Info: http://man7.org/linux/man-pages/man2/wait.2.html

```
#include  <stdio.h>
#include <unistd.h>
#include  <iostream>
#include  <sys/wait.h>
using namespace std;
int  main()
{    pid_t  pid;
    pid = fork();
    if (pid == 0)  // a child was created and you are inside it ☺
                cout<<"A New Child was created ☺ \n";
   else if (pid < 0)  // no new child was created (fail)
                {cout<<"No New Child Was created ☹\n";  return 0;}
    else // must be the parent
                { wait(0); // must wait for a child to finish
                   cout<<"I am the parent :)\n";
                 }
return 0;
}
```

17



execvp() system calls

SYNOPSIS:

#include <unistd.h>

DESCRIPTION:

The execvp system call is typically used for changing the execution image of a child process (or a process). That is, it allows a child process to run any program files, which include a binary executable or a shell script. The execvp() system call requires two arguments:

- ✓ The first argument is a character string that contains the name of a file to be executed.
- ✓ The second argument is a pointer to an array of character strings. More precisely, its type is char **, which is exactly identical to the argv array usually used in the main program.

RETURNED VALUE:

execvp() returns a negative value if the execution fails (e.g., the request file does not exist).

You will need to use it in your assignment 2 programming project.

More Info: http://man7.org/linux/man-pages/man2/wait.2.html

```
#include <unistd.h>
#include  <iostream>
#include  <sys/wait.h>
using namespace std;
int  main()  {     pid_t  pid;  char * commandString[2];
string command="ls";   // this is the ls unix command
string argument1="-l";  // this is an argument for the ls
commandString[0]=(char*) command.c_str();  //this is how to conver a string variable to a c_string
commandString[1]=(char*) argument1.c_str();      commandString[2]=NULL;
   pid = fork();
   if (pid == 0)  // a child was created and you are inside it ☺
        if (execvp(commandString[0], commandString) < 0) {
         //here using the execvp, the impage of execution will be replaced for the child by: ls -l. If
         it returns a negative value that means something went wrong
           cout<<"*** ERROR: exec failed\n";    exit(1);
        }
        cout<<"A New Child was created ☺ \n";  // will not execute this line since
                                                 //image of execution is changed by execvp
      else if (pid < 0)  // no new child was created (fail)
               {  cout<<"No New Child Was created ☹\n";  return 0;}
        else // must be the parent
             {  wait(0); // must wait for a child to finish
                  cout<<"I am the parent :)\n";                  }
return 0;    }  //enf of main method
```

18

Cooperating processes

- Processes frequently need to communicate and cooperate with other processes to accomplish a task.
  - For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line.
- Cooperating processes can:
  - Improve performance by overlapping activities or performing work in parallel
  - Enable an application to achieve a better program structure as a set of cooperating processes, where each is smaller than a single monolithic program
  - Easily share information
- Issues:
  - How do the processes communicate and pass information to each other?
  - How do the processes safely share data
    - When dependencies are present (e.g., Consumer waits Producer)
    - Making sure two or more processes do not get in each others ways.

19

19

Inter-process Communications (IPC)

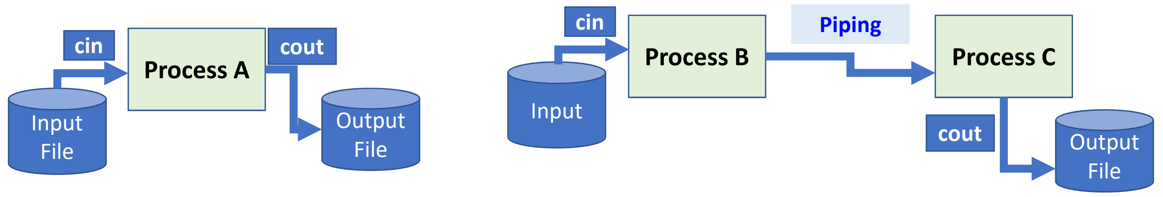
- Operating Systems provides techniques and mechanisms for facilitating communications and data sharing between software programs.
- The activities enabled by these techniques are called interprocess communications (**IPC**).
  - To facilitate the division of labor among several processes.
  - To facilitate the division of labor among computers on a network.
- A decision has to be made regarding which of the available IPC methods to use. (could be multiple techniques)
  - e.g., pipes, remote procedure calls, sockets for IPC, etc.

20

Using I/O Streams and Pipes for IPC

UW-Stout

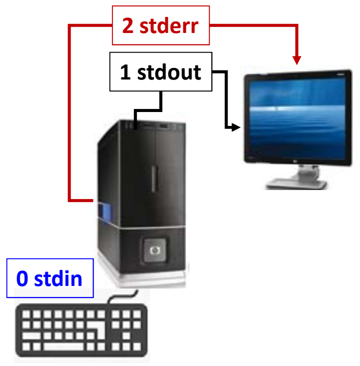
- Standard I/O streams are usually used to conduct IPC.
- In Linux, we can do:
  - **Redirection** which involves Reading or Writing data to files.
    - In Linux: **./myprogram < inputfile.txt > outputfile.txt** (try it!)
  - **Piping**: involves sending the data between two processes using anonymous pipes.
    - In Linux: **history | grep g++** (try it!)
  - A **pipe** is an I/O channel that a process can use to communicate with another process (in the same process or another process), or in some cases with itself. Data is written into one end of the pipe and read from the other.
  - There are two types of pipes for two-way communication: anonymous pipes (same machine) and named pipes (same or different like client server).



21

Standard Input, Output, and Streams

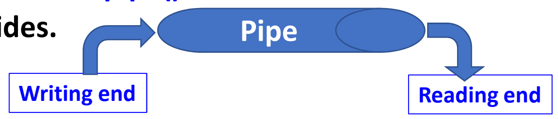
- A Linux shell receives input and sends output as sequences or streams of characters.
- Linux shells use three standard I/O streams, each of which is associated with a well-known file descriptor:
  - ✓**stdin** is the standard input stream, which provides input to commands. It has file descriptor **0**.
  - ✓**stdout** is the standard output stream, which displays output from commands. It has file descriptor **1**.
  - ✓stderr is the standard error stream, which displays error output from commands. It has file descriptor **2**.



22

IPC using Anonymous Pipes

- Anonymous pipes:
  - enable interconnection between processes such that the output (write) of one process is piped to the input (read) of another.
  - That is, they provide an efficient way to redirect standard input or output to child processes on the same computer.
  - OS uses a fixed-size buffer.
- Pipes can help designing programs with better performance and more complex nature.
- In Shell implementation assignment (programming project 2) you will need to use piping to implement the redirection of the in/out.
  - Will need to use the system call **pipe().**
  - **Explained in the next slides.**



```
#include<unistd.h>
int main() {
  int fds[2];
  pipe(fds);
  ……..
 }
```

23

Creating Pipes in C/C++

- To create a simple pipe with C, we make use of the system call pipe()
- It takes a single argument (an array of two integers), and if successful, the array will contain two new file descriptors to be used for the pipeline.

PROTOTYPE: int pipe(int fd[2]);

RETURNS: 0 on success or -1 on error

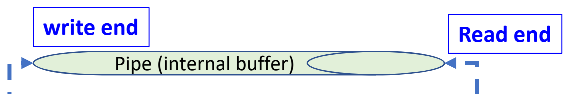
errno = EMFILE (no free descriptors), EMFILE (system file table is full), EFAULT (fd array is not valid)

✓fd[0] is set up for reading (read end),

✓fd[1] is set up for writing (write end)

24

Example: simple pipe in C/C++ Parent and Child



```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include<sys/wait.h>
#include <stdlib.h>    /* exit, EXIT_FAILURE */
#include <iostream>
using namespace std;
int main()
{ size_t Maxsize =80;       int    fd[2], nbytes;
  pid_t   childpid; char    readbuffer[80];
  char    sharedStr[] = "Hello, CS442!\n";
     if (pipe(fd)<0)  //create a pipe
              exit(1);  // error. No pipe was created.
 if((childpid = fork()) == -1)
      {      perror("fork did not work; Sorry!");
             exit(1);
      }
```

```
  if(childpid == 0)
  {
       /* Child process closes up input side of pipe */
       close(fd[0]);
       /* Send "string" through the output side of pipe */
       write(fd[1], sharedStr, (strnlen(sharedStr, Maxsize)) );
       exit(0);
  }
  else
  { wait(NULL);
       /* Parent process closes up output side of pipe */
       close(fd[1]);
       /* Read in a string from the pipe */
       nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
       cout<<"From Parent: Received string: "<< readbuffer<<endl;
  }
  return(0);
}
```

25

In-Class Exercise

- Please write a C/C++ program that forks (creates) a new child process that sends a message to its parent via a pipe created by the parent process (your main program).
- The message should include (3 lines):
  - ✓Your full name
  - ✓Department
  - ✓And email address
- Hints; You need to use the following system calls:
  - ✓pipe()
  - ✓fork()
  - ✓wait()
  - ✓Use the program on the previous slide as a model ☺

26

dup (C System Call)

- The dup() system call creates a copy of the file descriptor oldfd, using the lowest-numbered unused file descriptor for the new descriptor.
- Useful when implementing piping and redirection in Linux shell.
- **Synopsis**
  #include <unistd.h>
  int dup(int oldfd);

```
Example:

    int oldfd = open("duptest.txt", O_WRONLY | O_APPEND);
    int newCopy_fd = dup(oldfd);

    //Now newCopy_fd  can be used to refer to the open file, "duptest.txt".
```

- See the full example in the next slide >

27

dup Example

```
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <iostream>
#include <string>
using namespace std;
int main()
{   // system call open() returns
    //a file descriptor fd to a the file
    //"duptest.txt" stores in the same location
  int fd = open("duptest.txt",
         O_WRONLY | O_APPEND);
if (fd < 0) {
        cout << "Can not open the file" << endl;
        exit(1);
    } //end if
```

```
    // dup() will create the copy of fd as the
    //copy_fd then both can be used interchangeably
    //as needed.
    int copy_fd = dup(fd);

    //write() will write msg1 and msg2 to the file
    //using  the file descriptors
    string msg1 = "I am Dr. Alnaeli\n";
    string msg2 = "I live in Menomonie\n";

    //writing via the copy file descriptors
    write(copy_fd, (char *)msg1.c_str(), msg1.size());

    //writing via the original file descriptors
    write(fd, (char *)msg2.c_str(), msg2.size());
    return 0;

} //end of main
```

28

dup2 (C System Call)

- The dup2() system call performs the same task as dup(), but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in newfd.
- **SYNOPSIS**

  #include <unistd.h>

  int **dup2**(int oldfd, int newfd);

```
Example:

    int oldfd = open("dup2test.txt", O_WRONLY | O_APPEND);
    dup2(fd, 1);
    //Now all outputs will be sent to the file referred by the fd, "dup2test.txt".
    String msg="whatever \n";
    cout << msg<<endl;
    write(fd, (char *)msg.c_str(), msg.size());
```

- See the full example in the next slide >

29

Dup2 Example

```
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <iostream>
#include <string>
using namespace std;
int main()
{
    // system call open() returns a file
    descriptor fd to a the file "duptest.txt"
    stores in the same location
    int fd = open("dup2test.txt", O_WRONLY | O_APPEND);
    if (fd < 0) {
    cout << "Can not open the file" << endl;
    exit(1);
    }
```

All output will be sent to the file referenced by fd

```
    // here the newfd is the file descriptor of
    //stdout (i.e. 1)
    dup2(fd, 1);

    // All the output statements will be written
    in the file
    // "dup2test.txt"
    cout << "I will be written to the file
    dup2test.txt" << endl;

    // write() will write msg1 and msg2 to the
    //file using  the file descriptor
    string msg1 = "I am Dr. Alnaeli\n";
    string msg2 = "I live in Menomonie\n";
    cout << msg1;

    write(fd, (char *)msg2.c_str(), msg2.size());
    close(fd);
    return 0;
} //end of main
```

30

References and useful resources

- https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.bpxbd00/rtpip.htm
- http://www.gnu.org/software/libc/manual/html_node/Standard-Streams.html
- https://docs.microsoft.com/en-us/windows/desktop/ipc/interprocess-communications
- https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm
- **Textbooks (see syllabus)**

31