Inter Process Communication

Instructor: William Zheng
Email:
william.zheng@digipen.edu
PHONE EXT: 1745

Why the need for inter-process communication?

- Program reuse
- Both Cooperation/Independence
- Parallelism
- OS needs (Microkernel)

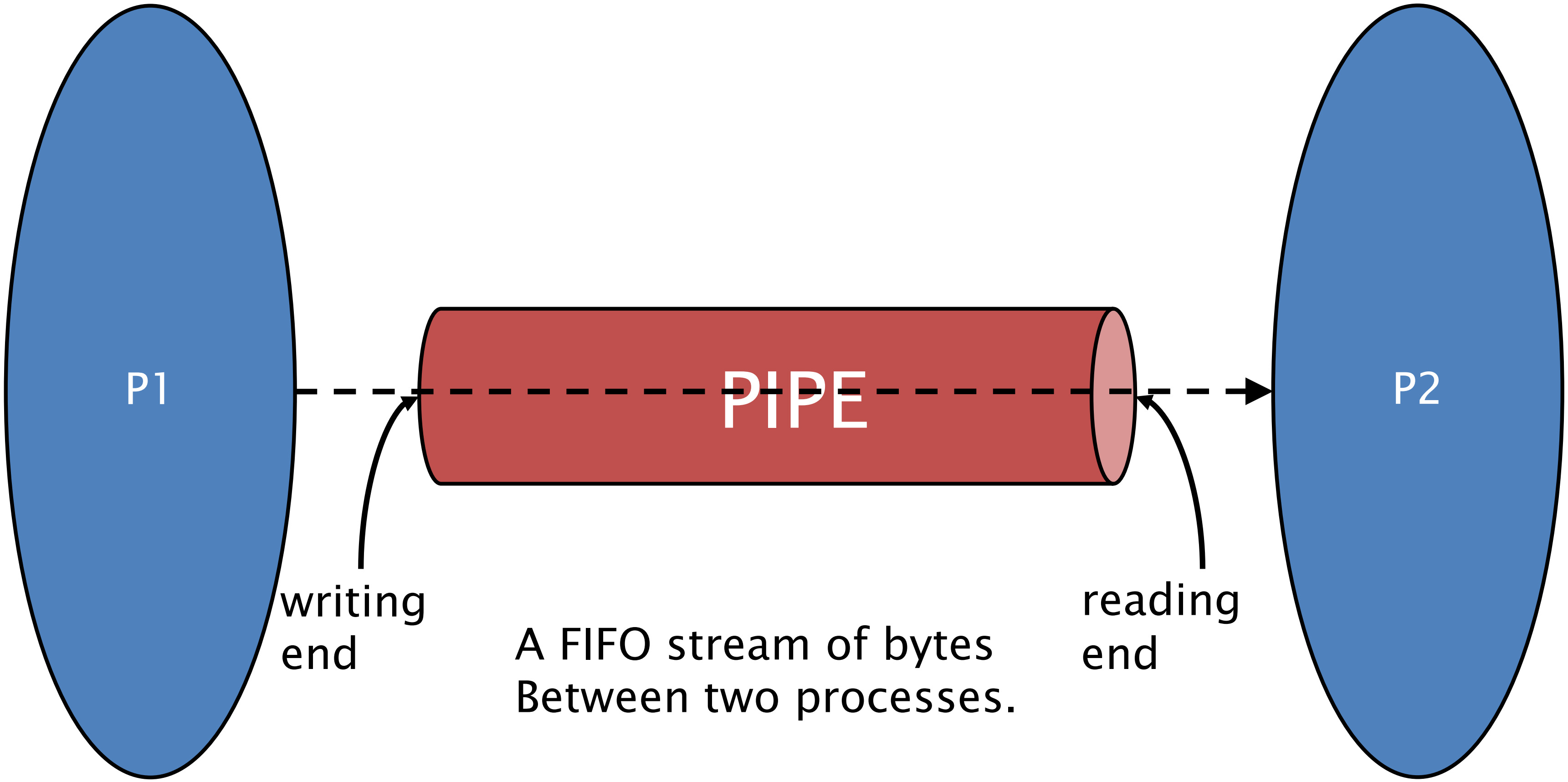
3 IPC mechanisms

- Pipes
- Shared memory
- Message Passing

Motivations for Program Reuse

- What would you do if you need to go through a file containing 10,000 names to count the number of names containing the “sam”?

Pipes – the basic idea



Pipes continued …

Named Pipes	Anonymous Pipes
Sharable btw any processes (subject to permissions)	Sharable btw processes of the same ancestry
Full Duplex in Windows Half Duplex in Linux	Unidirectional
`CreateNamedPipes` (Windows) `mkfifo` (Unix/Linux)	`CreatePipes` (Windows) `pipe` (Linux)
Obtain the pipe by the name of the pipe	Obtain the pipe from parent

Pipe Implementation

- Represented by two File Objects
  - Writing End/Reading End
- Practical Implementation
  - `CreatePipe` instantiates two File Handles
  - `pipe` returns two file descriptors

Parent – Child

How does parent process pass the pipe to the child process?

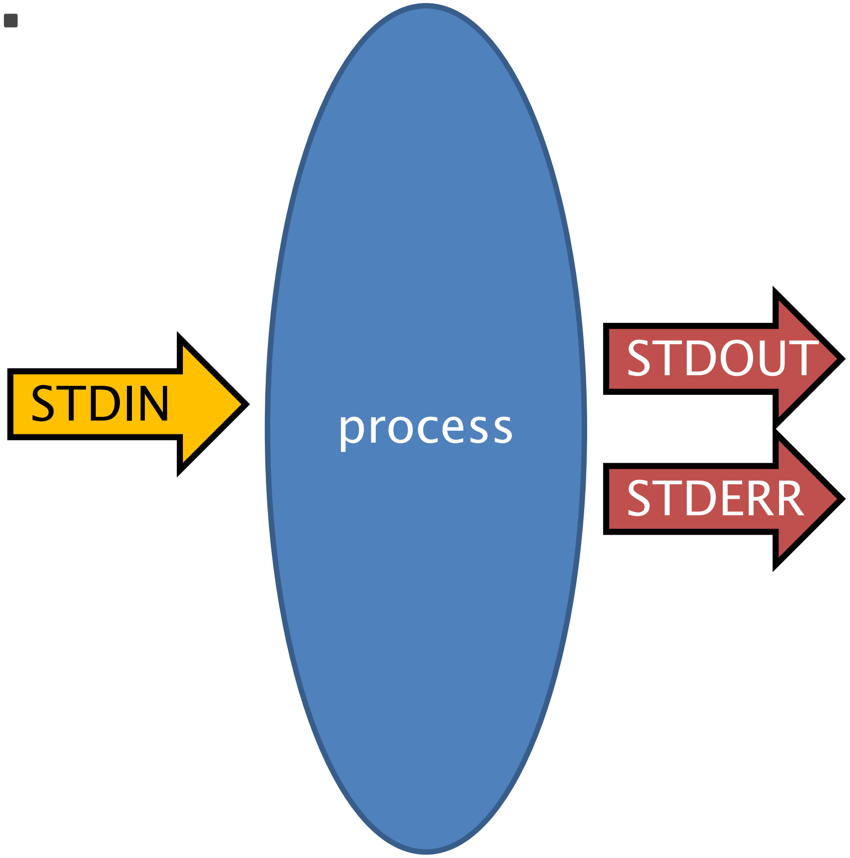
Key to passing anonymous pipes

- All child processes inherit the open files of the parent.

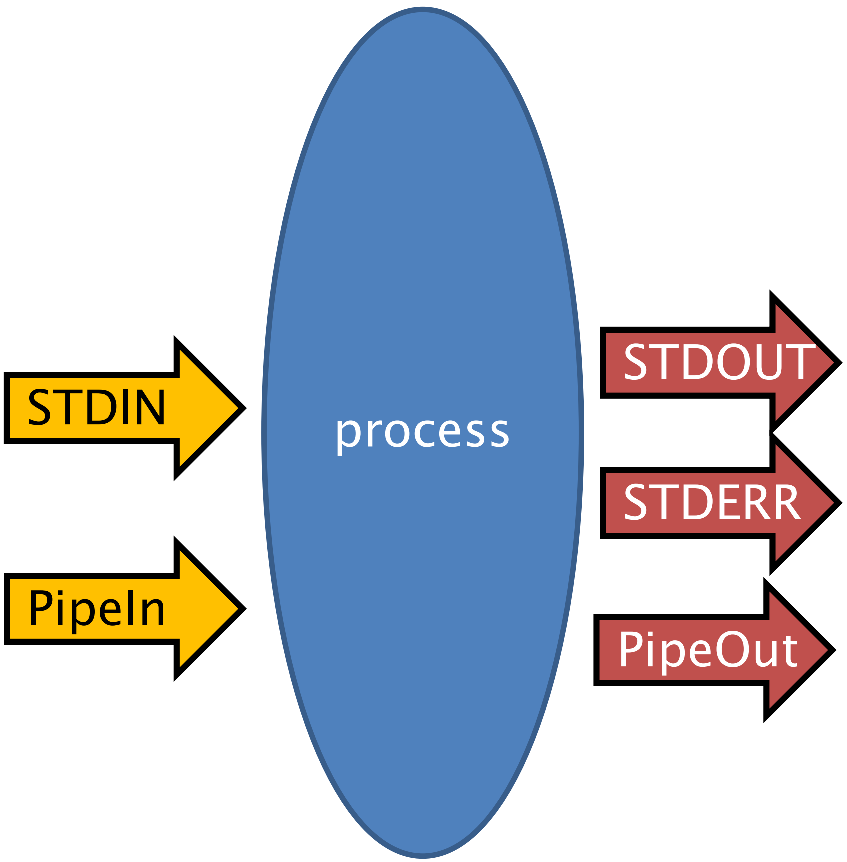
Piping

But how does the "grep … | wc" thingy work?

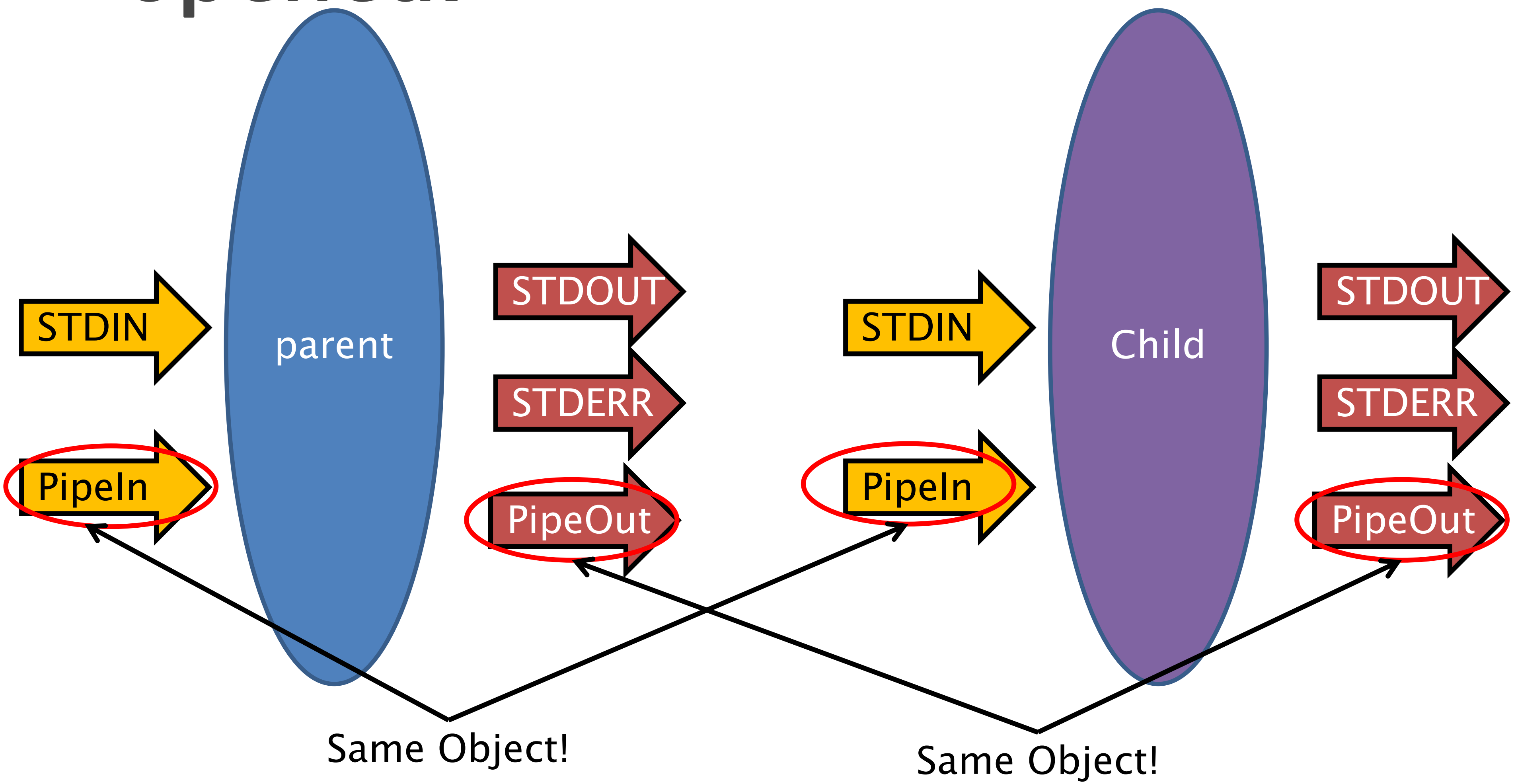
Every process has at least 3 Files…



If the process creates a pipe, it would have 2 more files opened.

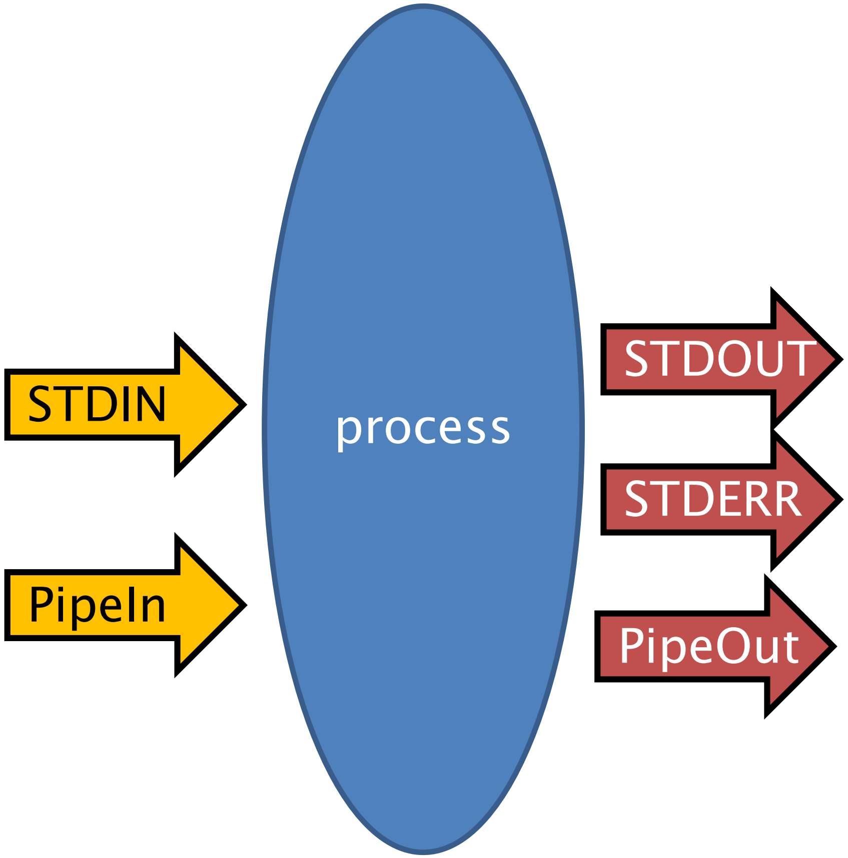


If I spawn a child process, it would have the same files opened.



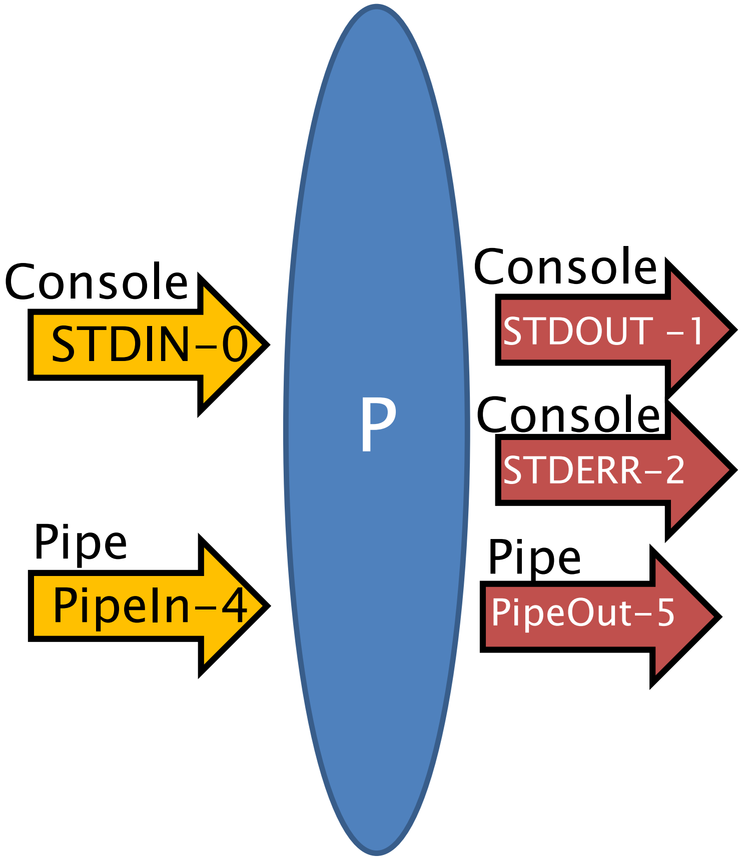
File redirection is required…

File redirection (Linux/Unix) – 1



- File Descriptors
  - Just a Number
- In Linux/Unix,
  - 0 for stdin
  - 1 for stdout
  - 2 for stderr
- Any number for the PipeIn or PipeOut

File redirection (Linux/Unix) – 2



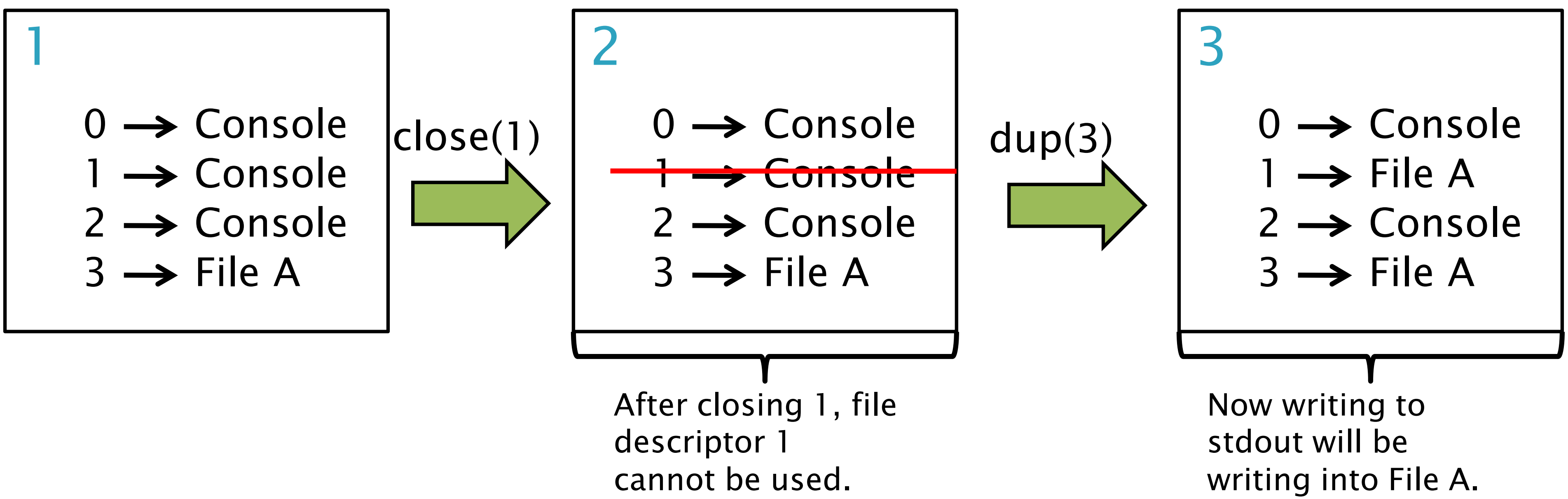
- Aim: get 1 to be the same object as pipeout.
- Use `dup` system call.

Unix File Descriptors

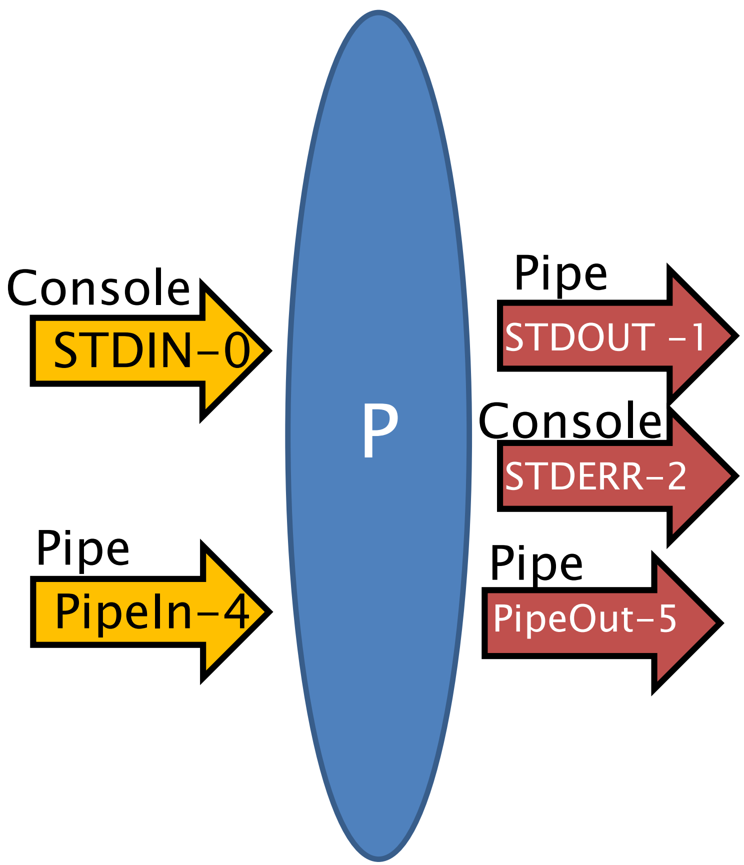
- Unix/Linux files are numbered
  - 0, 1, 2 … etc
  - 0 is stdin, 1 is stdout, 2 is stderr
  - By default,
    - scanf and std::cin gets input from stdin.
    - printf and std::cout will print to stdout.
  - The default “file” opened for stdin, stdout and stderr will be the console. (where you can observe the printing)
  - However, using redirection, we can actually get 0, 1 or 2 to be “files” other than the console.

Using dup

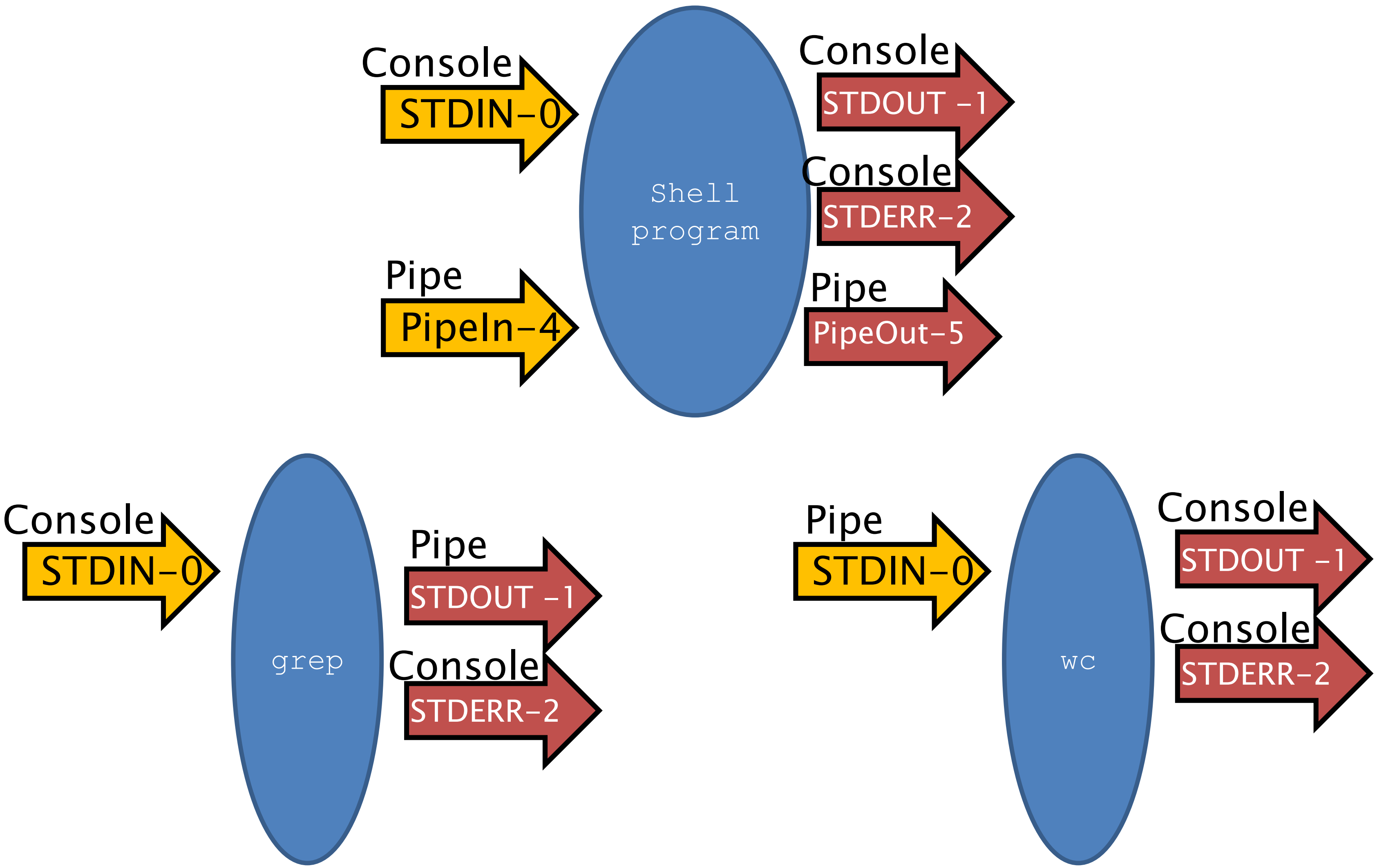
- `close()` system call closes a file. (Renders the file descriptor invalid)
- The `dup()` system call duplicates the given file descriptor using the lowest available file descriptor number.



What does `dup` do?



“grep abc | wc”



Console
STDIN–0
Pipe
PipeIn–4
Shell program
Console
STDOUT –1
Console
STDERR–2
Pipe
PipeOut–5
Console
STDIN–0
grep
Pipe
STDOUT –1
Console
STDERR–2
Pipe
STDIN–0
wc
Console
STDOUT –1
Console
STDERR–2

POSIX Pipes

- A pipe is a stream of communication between two processes
- You can think of it as a virtual file stream shared between two processes
- A process can read and/or write to a pipe
- The pipe function gets two descriptors (integer labels)
  - Read descriptor – read from the pipe
  - Write descriptor – write to the pipe
- Both processes must know the descriptors
- read and write are used with the pipe

POSIX pipe example

```
int main(void) {
  int pid;
  char buffer[1024];
  int descriptor[2];
  pipe(descriptor);
  pid = fork();
  if (pid == 0) {
    fprintf(stdout,"input: ");
    fgets(buffer,sizeof(buffer),stdin);
    write(descriptor[1],buffer,strlen(buffer)+1);
    exit(0); }
  else {
    wait(NULL);
    read(descriptor[0],buffer,sizeof(buffer));
    fprintf(stdout,"message: %s",buffer); }
  return 0;
}
```

POSIX pipe example notes

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

- The unistd.h (POSIX) header file needs to be included for the pipe, read, and write functions
- The pipe functions fills in the descriptor array
  - descriptor[0] is for reading from the pipe
  - descriptor[1] is for writing to the pipe

POSIX shared memory

- Allows for a block of memory to be shared between multiple processes
- shmget is used to create (or retrieve if already created) a block of memory
  - Returns an identifier for the block
- shmat attaches the memory to the process
  - Returns the address of memory block
- shmdt detaches the memory from the process
- shmctl is used to delete the memory block

Shared memory example

```
int main(void) {
  int pid, shmid;
  char *buffer;
  shmid = shmget((key_t)123,1024,0666|IPC_CREAT);
  buffer = (char *)shmat(shmid,NULL,0);
  strcpy(buffer,"");
  pid = fork();
  if (pid == 0) {
    strcpy(buffer,"step on no pets");
    shmdt(buffer);
    exit(0); }
  else {
    wait(NULL);
    fprintf(stdout,"message: %s\n",buffer);
    shmdt(buffer);
    shmctl(shmid,IPC_RMID,0); }
  return 0;
}
```

Shared memory example notes

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

- Header file sys/shm.h needed for shared memory functions
- Shared memory has file-style permissions
  - user/group/other, 3 bits each (9 bit total)
  - 6 = 0x110 = rw-
- Each process must know the key value of the shared memory block

POSIX message queues

- Allows for passing messages between processes
- msgget creates (or retrieves an existing) message queue
  - Returns an identifier for the queue
- msgsnd posts a message to the queue
- msgrcv retrieves (and removes) a message from the queue
- msgctl is used to delete the message queue

Message queue example

```
int main(void) {
  int pid, queue_id;
  msg_struct msg;
  queue_id = msgget((key_t)100,0666|IPC_CREAT);
  pid = fork();
  if (pid == 0) {
    msg.type = MSG_STRUCT;
    strcpy(msg.buffer,"Was it a rat I saw?");
    msgsnd(queue_id,&msg,1024,0);
    strcpy(msg.buffer,"No lemons, no melon.");
    msgsnd(queue_id,&msg,1024,0);
    exit(0); }
  else {
    wait(NULL);
    while (msgrcv(queue_id,&msg,1024,0,IPC_NOWAIT) != -1) {
      if (msg.type == MSG_STRUCT)
        fprintf(stdout,"message: %s\n",msg.buffer);
      else
        fprintf(stdout,"unknown message\n"); }
    msgctl(queue_id,IPC_RMID,NULL); }
  return 0;
}
```

Queue example notes (1)

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

- Header file sys/msg.h must be included
- Each process must know the key value of the queue
- Queue has file-style permissions
- msgrcv has two modes
  - IPC_NOWAIT: returns -1 if there are no messages on the queue (nonblocking)
  - not IPC_NOWAIT: blocks until there is a message on the queue

Queue example notes (2)

```
typedef struct {
  long int type;
  char buffer[1024];
} msg_struct;
#define MSG_STRUCT 1
```

- Messages can have different structures
- A message structure must have a long int as its first field (used for the message type identifier)
- Message size (specified in msgsnd and msgrcv) is the message structure size *without* the long int identifier

IPC mechanisms

- Shared memory
  - Processes have access to a common block of memory
  - Processes read and write to the shared memory
  - Synchronization issues
- Message passing
  - Information passed between processes via the kernel
  - Requires a message protocol
  - Safe, but slow

Communications Models

(a) Message passing. (b) shared memory.