I/O Management

Goals of this Lecture

Help you to learn about:

- The C/Unix **file** abstraction
- Standard C I/O
  - Data structures & functions
- Unix I/O
  - Data structures & functions
- The implementation of Standard C I/O using Unix I/O
- Programmatic redirection of stdin, stdout, and stderr
- Pipes

Agenda

The C/Unix file abstraction

Unix I/O system calls

C's Standard IO library (FILE *)

Implementing standard C I/O using Unix I/O

Redirecting standard files

Pipes

C/Unix File Abstraction

Problem:

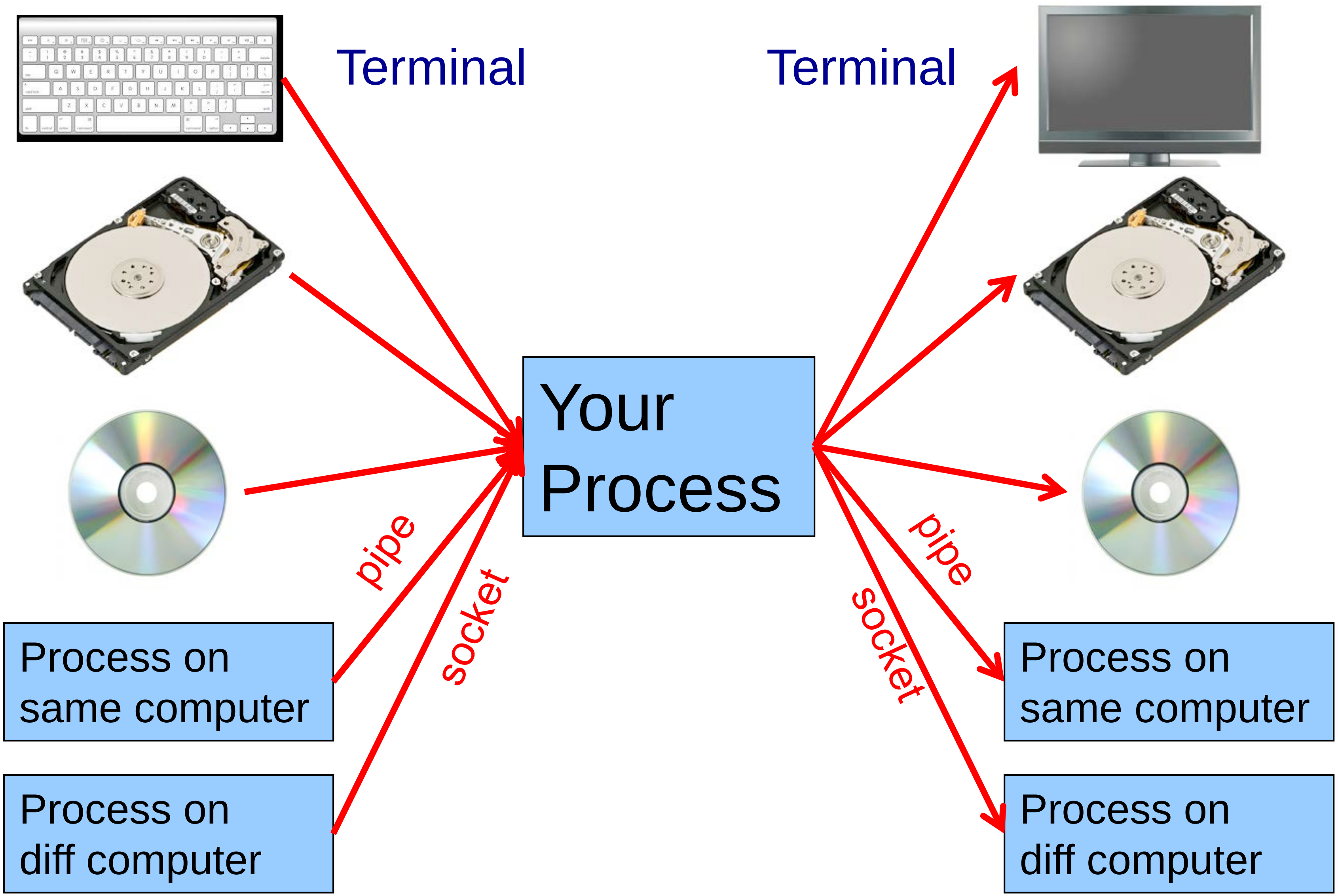
- At the physical level…
- Code that **reads** from **keyboard** is very different from code that reads from **disk**, etc.
- Code that **writes** to **video screen** is very different from code that writes to **disk**, etc.
- Would be nice if application programmer didn't need to worry about such details

Solution:

- **File**: a sequence of bytes
- C and Unix allow application program to treat any data source/destination as a **file**

Commentary: **Beautiful** abstraction!

Data Sources and Destinations

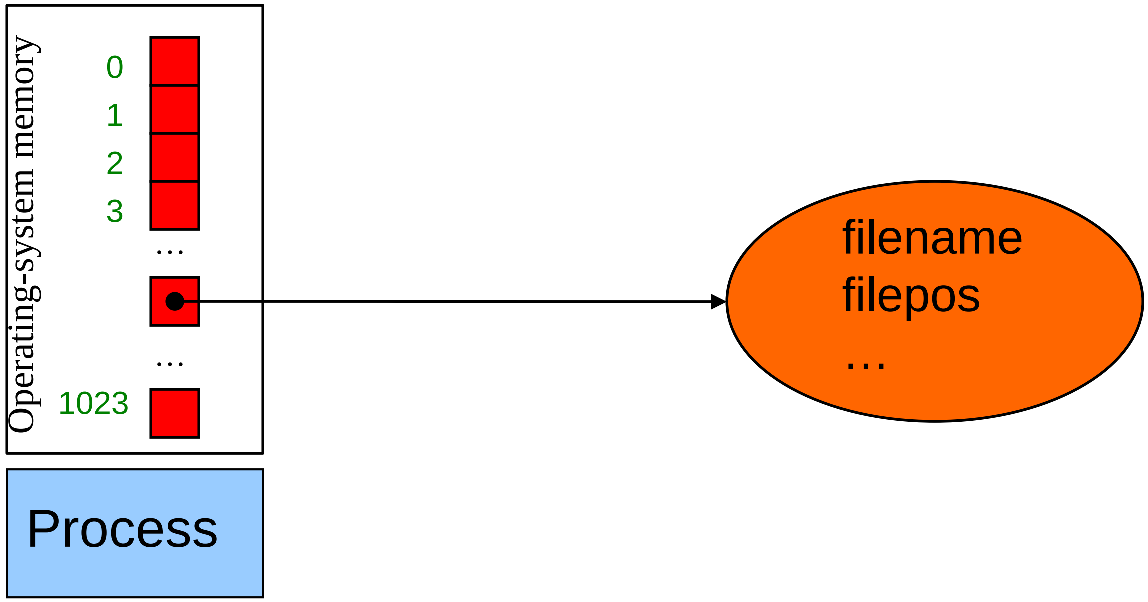


C/Unix File Abstraction

Each file has an associated **file position**

- Starts at beginning of file (if opened to read or write)
- Starts at end of file (if opened to append)

Unix I/O Data Structures



File descriptor: Integer that uniquely identifies an open file

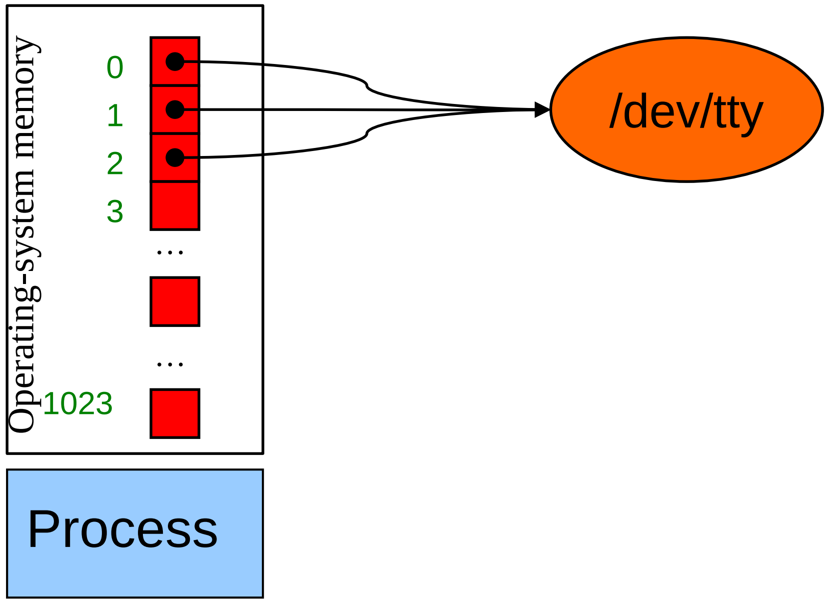
File descriptor table: an array

- Indices are file descriptors; elements are pointers to file tables
- One unique file descriptor table for each process

File table: a structure

- In-memory surrogate for an open file
- Created when process opens file; maintains file position

Unix I/O Data Structures



At process start-up files with fd 0, 1, 2 are open automatically

(By default) each references file table for a file named /dev/tty

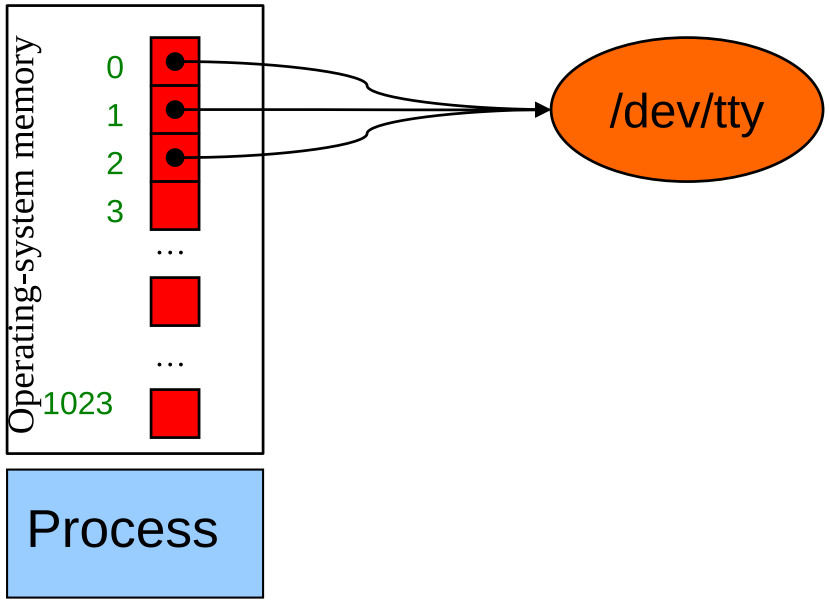
/dev/tty

In-memory surrogate for the terminal

Terminal

Combination keyboard/video screen

Unix I/O Data Structures



Read from stdin ⇒ read from fd 0
Write to stdout ⇒ write to fd 1
Write to stderr ⇒ write to fd 2

System-Level Functions

As noted in the ***Exceptions and Processes*** lecture…

Linux system-level functions for **I/O management**

Number	Function	Description
0	read()	Read data from file descriptor Called by getchar(), scanf(), etc.
1	write()	Write data to file descriptor Called by putchar(), printf(), etc.
2	open()	Open file or device Called by fopen(…, “r”)
3	close()	Close file descriptor Called by fclose()
85	creat()	Open file or device for writing Called by fopen(…, “w”)
8	lseek()	Change file position Called by fseek()

System-Level Functions

As noted in the ***Exceptions and Processes*** lecture..

Linux system-level functions for **I/O redirection** and **inter-process communication**

Number	Function	Description
32	dup()	Duplicate an open file descriptor
22	pipe()	Create a channel of communication between processes

Agenda

The C/Unix file abstraction

Unix I/O system calls

C's Standard IO library (FILE *)

Implementing standard C I/O using Unix I/O

Redirecting standard files

Pipes

Unix I/O Functions

```
int creat(char *filename, mode_t mode);
```

- Create a new empty file named **`filename`**
  - **`mode`** indicates permissions of new file
- Implementation:
  - Create new empty file on disk
  - Create file table
  - Set first unused file descriptor to point to file table
  - Return file descriptor used, -1 upon failure

Unix I/O Functions

```
int open(char *filename, int flags, …);
```

- Open the file whose name is **`filename`**
  - **`flags`** often is **`O_RDONLY`**
- Implementation (assuming **`O_RDONLY`**):
  - Find existing file on disk
  - Create file table
  - Set first unused file descriptor to point to file table
  - Return file descriptor used, -1 upon failure

Unix I/O Functions

```
int close(int fd);
```

- Close the file **`fd`**
- Implementation:
  - Destroy file table referenced by element **`fd`** of file descriptor table
    - As long as no other process is pointing to it!
  - Set element **`fd`** of file descriptor table to **`NULL`**

Unix I/O Functions

```
int read(int fd, void *buf, int count);
```

- Read into **buf** up to **count** bytes from file **fd**
- Return the number of bytes read; 0 indicates end-of-file

```
int write(int fd, void *buf, int count);
```

- Writes up to **count** bytes from **buf** to file **fd**
- Return the number of bytes written; -1 indicates error

```
int lseek(int fd, int offset, int whence);
```

- Set the file position of file **fd** to file position **offset**. **whence** indicates if the file position is measured from the beginning of the file (**SEEK_SET**), from the current file position (**SEEK_CUR**), or from the end of the file (**SEEK_END**)
- Return the file position from the beginning of the file

Unix I/O Functions

Note

- Only 6 system-level functions support all I/O from all kinds of devices!

Commentary: **Beautiful** interface!

Unix I/O Example 0

Proto-getchar()

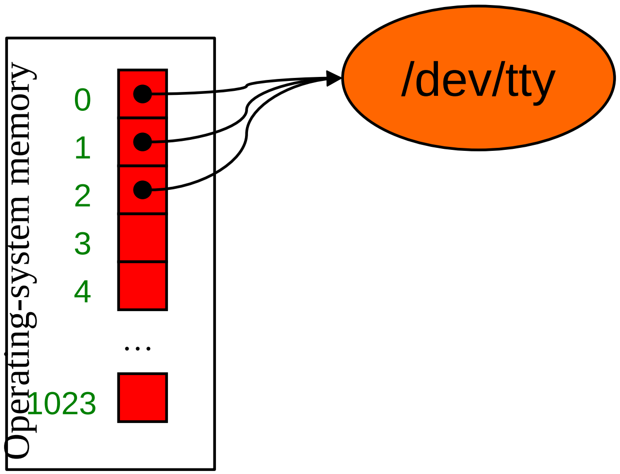
```
#include <string.h>
#include <unistd.h>

int proto_getchar(void)
{   char buf[1];
    int n;

    n = read(0, buf, 1);
    if (n==1)
        return buf[0];
    else return EOF;
}
```

# of bytes to (try to) read

0 is the file descriptor
of the standard input



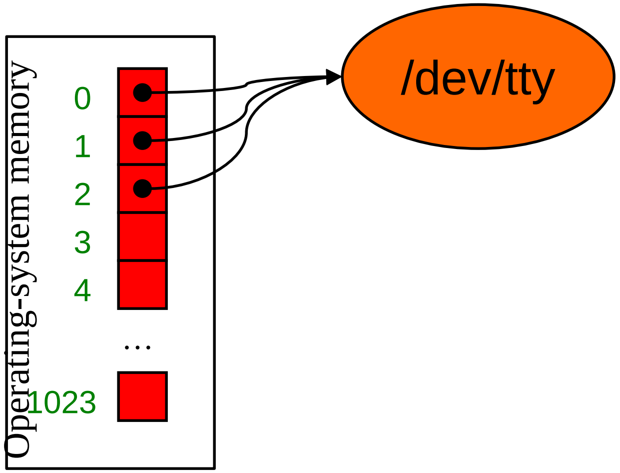
and the problem is . . . too slow.

Does a system call for every character.

Unix I/O Example 1

Write "hello, world\n" to /dev/tty

```
#include <string.h>
#include <unistd.h>
int main(void)
{  char hi[] = "hello, world\n";
   size_t countWritten = 0;
   size_t countToWrite = strlen(hi);
   while (countWritten < countToWrite)
      countWritten +=
         write(1, hi + countWritten,
               countToWrite - countWritten);
   return 0;
}
```

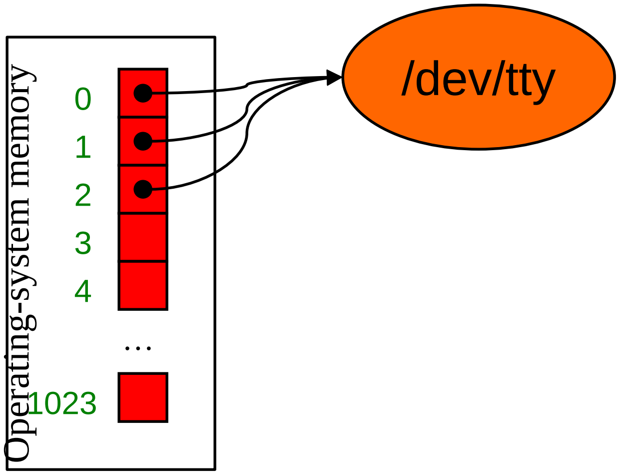


To save space, no error handling code is shown

Unix I/O Example 2

```
#include <fcntl.h>
#include <unistd.h>
int main(void)
{  enum {BUFFERSIZE = 10};
   int fdIn, fdOut;
   int countRead, countWritten;
   char buf[BUFFERSIZE];
   fdIn = open("infile", O_RDONLY);
   fdOut = creat("outfile", 0600);
   for (;;)
   {  countRead =
         read(fdIn, buf, BUFFERSIZE);
      if (countRead == 0) break;
      countWritten = 0;
      while (countWritten < countRead)
         countWritten +=
            write(fdOut,
               buf + countWritten,
               countRead - countWritten);
   }
   close(fdOut);
   close(fdIn);
   return 0;
}
```

Copy all bytes from infile to outfile

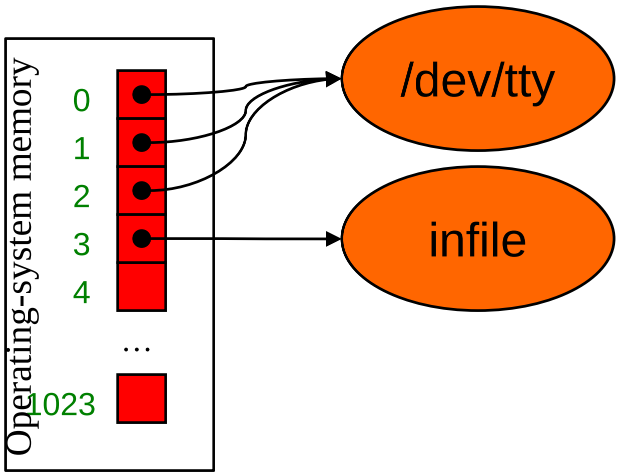


To save space, no error handling code is shown

Unix I/O Example 2

```
#include <fcntl.h>
#include <unistd.h>
int main(void)
{  enum {BUFFERSIZE = 10};
   int fdIn, fdOut;
   int countRead, countWritten;
   char buf[BUFFERSIZE];
   fdIn = open("infile", O_RDONLY);
   fdOut = creat("outfile", 0600);
   for (;;)
   {  countRead =
         read(fdIn, buf, BUFFERSIZE);
      if (countRead == 0) break;
      countWritten = 0;
      while (countWritten < countRead)
         countWritten +=
            write(fdOut,
               buf + countWritten,
               countRead - countWritten);
   }
   close(fdOut);
   close(fdIn);
   return 0;
}
```

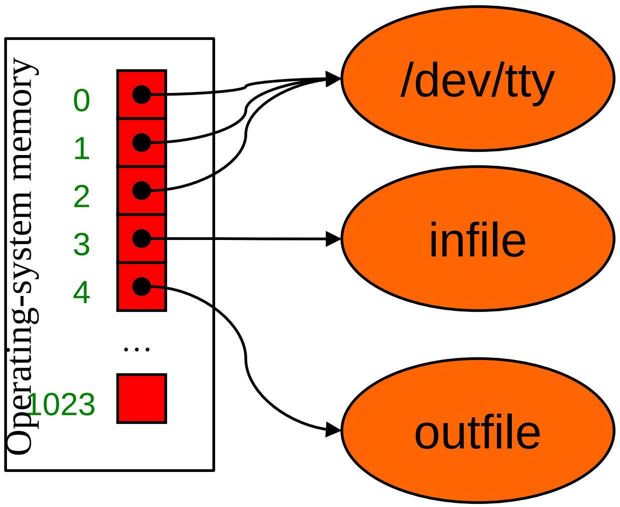
3



Unix I/O Example 2

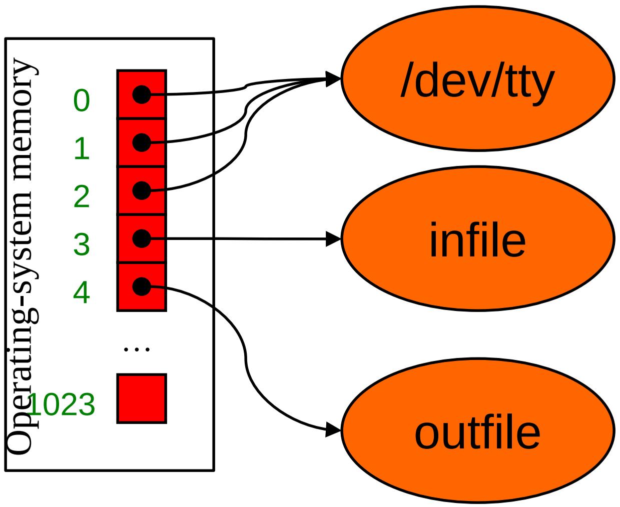
```
#include <fcntl.h>
#include <unistd.h>
int main(void)
{  enum {BUFFERSIZE = 10};
   int fdIn, fdOut;
   int countRead, countWritten;
   char buf[BUFFERSIZE];
   fdIn = open("infile", O_RDONLY);
   fdOut = creat("outfile", 0600);
   for (;;)
   {  countRead =
         read(fdIn, buf, BUFFERSIZE);
      if (countRead == 0) break;
      countWritten = 0;
      while (countWritten < countRead)
         countWritten +=
            write(fdOut,
               buf + countWritten,
               countRead - countWritten);
   }
   close(fdOut);
   close(fdIn);
   return 0;
}
```

3
4



Unix I/O Example 2

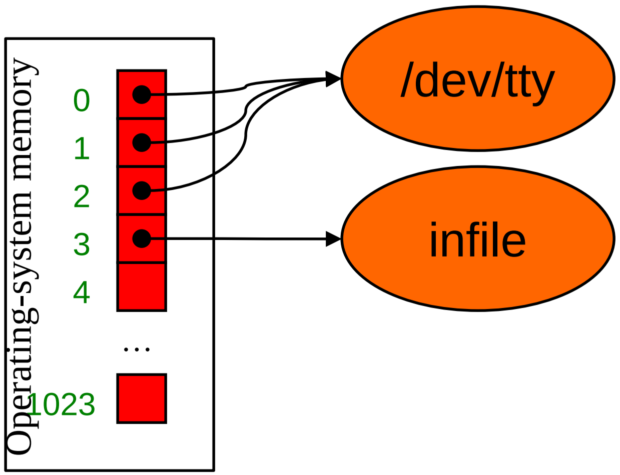
```
#include <fcntl.h>
#include <unistd.h>
int main(void)
{  enum {BUFFERSIZE = 10};
   int fdIn, fdOut;
   int countRead, countWritten;
   char buf[BUFFERSIZE];
   fdIn = open("infile", O_RDONLY);
   fdOut = creat("outfile", 0600);
   for (;;)
   {  countRead =
         read(fdIn, buf, BUFFERSIZE);
      if (countRead == 0) break;
      countWritten = 0;
      while (countWritten < countRead)
         countWritten +=
            write(fdOut,
               buf + countWritten,
               countRead - countWritten);
   }
   close(fdOut);
   close(fdIn);
   return 0;
}
```



Unix I/O Example 2

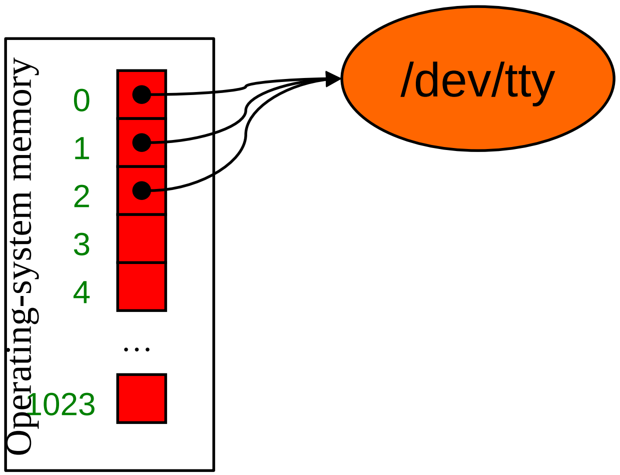
```
#include <fcntl.h>
#include <unistd.h>
int main(void)
{  enum {BUFFERSIZE = 10};
   int fdIn, fdOut;
   int countRead, countWritten;
   char buf[BUFFERSIZE];
   fdIn = open("infile", O_RDONLY);
   fdOut = creat("outfile", 0600);
   for (;;)
   {  countRead =
         read(fdIn, buf, BUFFERSIZE);
      if (countRead == 0) break;
      countWritten = 0;
      while (countWritten < countRead)
         countWritten +=
            write(fdOut,
               buf + countWritten,
               countRead - countWritten);
   }
   close(fdOut);
   close(fdIn);
   return 0;
}
```

3
4



Unix I/O Example 2

```c
#include <fcntl.h>
#include <unistd.h>
int main(void)
{  enum {BUFFERSIZE = 10};
   int fdIn, fdOut;
   int countRead, countWritten;
   char buf[BUFFERSIZE];
   fdIn = open("infile", O_RDONLY);
   fdOut = creat("outfile", 0600);
   for (;;)
   {  countRead =
         read(fdIn, buf, BUFFERSIZE);
      if (countRead == 0) break;
      countWritten = 0;
      while (countWritten < countRead)
         countWritten +=
            write(fdOut,
               buf + countWritten,
               countRead - countWritten);
   }
   close(fdOut);
   close(fdIn);
   return 0;
}
```



Agenda

The C/Unix file abstraction

Unix I/O system calls

C's Standard IO library (FILE *)

Implementing standard C I/O using Unix I/O

Redirecting standard files

Pipes

Standard C I/O Data Structure

We want 1-character-at-a-time I/O   (getc(), putc())

We want a-few-characters-at-a-time I/O (scanf, printf)

We *could* do this with read() and write() system calls,

BUT IT WOULD BE TOO SLOW to do 1 syscall per byte

Solution: Buffered input/output as an Abstract Data Type

The **FILE** ADT

- A **`FILE`** object is an in-memory surrogate for an opened file
  - Created by **`fopen()`**
  - Destroyed by **`fclose()`**
- Used by reading/writing functions

Standard C I/O Functions

Some of the most popular:

```
FILE *fopen(const char *filename, const char *mode);
```

- Open the file named **filename** for reading or writing
- **mode** indicates data flow direction
  - "r" means read; "w" means write, "a" means append)
- Creates **FILE** structure
- Returns address of **FILE** structure

```
int fclose(FILE *file);
```

- Close the file identified by **file**
- Destroys **FILE** structure whose address is **file**
- Returns 0 on success, EOF on failure

Standard C Input Functions

Some of the most popular:

`int fgetc(FILE *file);`

- Read a char from the file identified by **`file`**
- Return the char on success, **`EOF`** on failure

`int getchar(void);`

- Same as **`fgetc(stdin)`**

`char *fgets(char *s, int n, FILE *file);`

- Read at most **`n`** characters from **`file`** into array **`s`**
- Returns s on success, **`NULL`** on failure

`char *gets(char *s);`

- Essentially same as **`fgets(s, INT_MAX, stdin)`**
- *Buffer overflow waiting to happen*

Standard C Input Functions

Some of the most popular:

```
int fscanf(FILE *file, const char *format, ...);
```

- Read chars from the file identified by **`file`**
- Convert to values, as directed by **`format`**
- Copy values to memory
- Return count of values successfully scanned

```
int scanf(const char *format, ...);
```

- Same as **`fscanf(stdin, format, …)`**

Standard C Output Functions

Some of the most popular:

`int fputc(int c, FILE *file);`

- Write **c** (converted to a char) to file
- Return **c** on success, **EOF** on failure

`int putchar(int c);`

- Same as `fputc(c, stdout)`

`int fputs(const char *s, FILE *file);`

- Write string **s** to `file`
- Return non-negative on success, **EOF** on error

`int puts(const char *s);`

- Essentially same as `fputs(s, stdout)`

Standard C Output Functions

Some of the most popular:

```
int fprintf(FILE *file, const char *format, ...);
```

- Write chars to the file identified by **`file`**
- Convert values to chars, as directed by **`format`**
- Return count of chars successfully written
- Works by calling **`fputc()`** repeatedly

```
int printf(const char *format, ...);
```

- Same as **`fprintf(stdout, format, ...)`**

Standard C I/O Functions

Some of the most popular:

`int fflush(FILE *file);`

- On an output file: write any buffered chars to **`file`**
- On an input file: behavior undefined
- **`file`** == **`NULL`** ⇒ flush buffers of **all** open files

`int fseek(FILE *file, long offset, int origin);`

- Set the file position of **`file`**
- Subsequent read/write accesses data starting at that position
- Origin: **`SEEK_SET`**, **`SEEK_CUR`**, **`SEEK_END`**

`int ftell(FILE *file);`

- Return file position of **`file`** on success, -1 on error

Standard C I/O Example 1

Write "hello, world\n" to **stdout**

```
#include <stdio.h>
int main(void)
{  char hi[] = "hello world\n";
   size_t i = 0;
   while (hi[i] != '\0')
   {  putchar(hi[i]);
      i++;
   }
   return 0;
}
```

Simple
Portable
Efficient (via buffering)

```
#include <stdio.h>
int main(void)
{  puts("hello, world");
   return 0;
}
```

```
#include <stdio.h>
int main(void)
{  printf("hello, world\n");
   return 0;
}
```

Standard C I/O Example 2

Copy all bytes from infile to outfile

```
#include <stdio.h>
int main(void)
{   int c;
    FILE *inFile;
    FILE *outFile;
    inFile = fopen("infile", "r");
    outFile = fopen("outfile", "w");
    while ((c = fgetc(inFile)) != EOF)
       fputc(c, outFile);
    fclose(outFile);
    fclose(inFile);
    return 0;
}
```

Simple
Portable
Efficient (via buffering)

Standard C Buffering

Question: Exactly when are buffers flushed?

Answers:

If reading from a file

- (1) When buffer is empty

Standard C Buffering

Question: Exactly when are buffers flushed?

Answers:

If writing to an ordinary file

(1) File's buffer becomes full

(2) Process calls **`fflush()`** on that file

(3) Process terminates normally

If writing to **`stdout`** (in addition to previous)

(4) **`stdout`** is bound to terminal and **`'\n'`** is appended to buffer

(5) **`stdin`** and **`stdout`** are bound to terminal
and read from **`stdin`** occurs

If writing to **`stderr`**

- Irrelevant; **`stderr`** is unbuffered

Standard C Buffering Example

```
#include <stdio.h>
int main(void)
{  int dividend, divisor, quotient;

   printf("Dividend: ");
   scanf("%d", &dividend);

   printf("Divisor: ");
   scanf("%d", &divisor);

   printf("The quotient is ");
   quotient = dividend / divisor;
   printf("%d\n", quotient);
   return 0;
}
```

Output buffered
Buffer flushed
Output buffered
Buffer flushed
Output buffered
Buffer flushed

```
$ pgm
Dividend: 6
Divisor: 2
The quotient is 3
$
```

```
$ pgm
Dividend: 6
Divisor: 0
Floating point exception
$
```

Agenda

The C/Unix file abstraction

Unix I/O system calls

C's Standard IO library (FILE *)

Implementing standard C I/O using Unix I/O

Redirecting standard files

Pipes

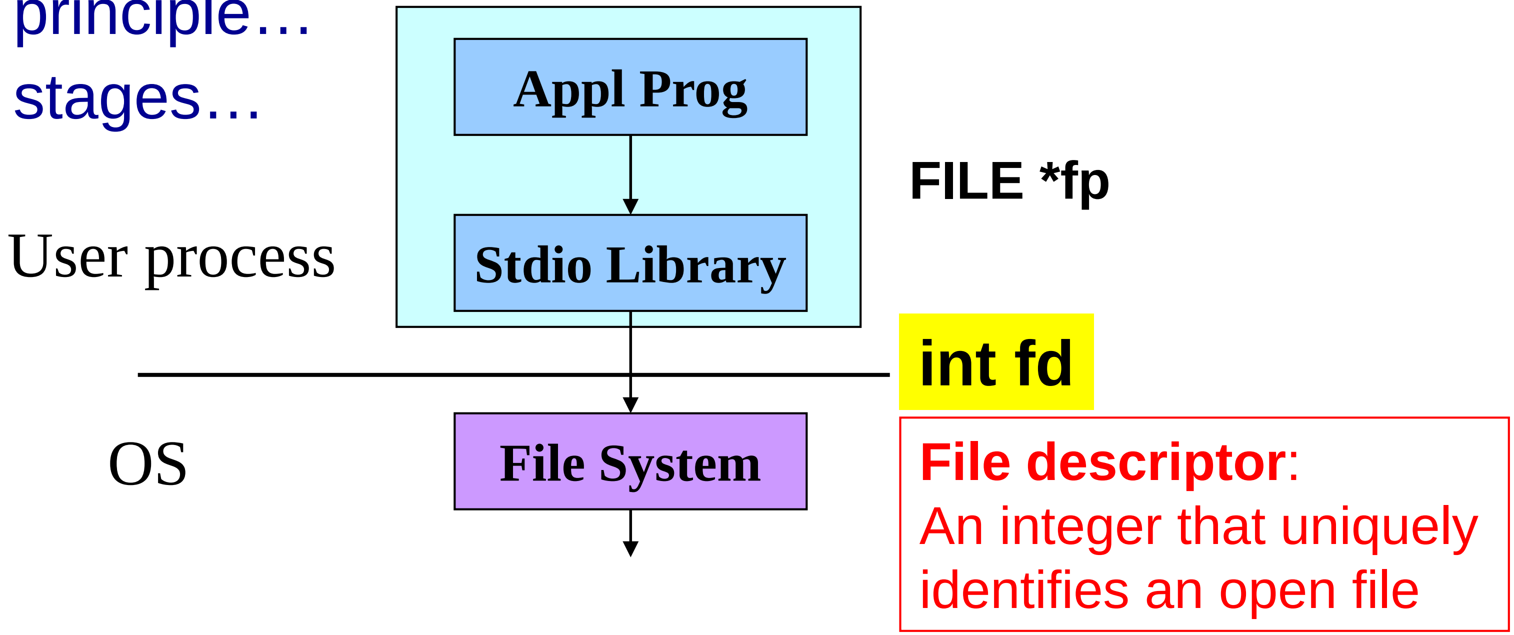
Standard C I/O

Question:

- How to implement standard C I/O data structure and functions using Unix I/O data structures and functions?

Answer:

- In principle…
- In stages…



Implementing `getchar` and `putchar`

`getchar()` calls **`read()`** to read one byte from fd 0

`putchar()` calls **`write()`** to write one byte to fd 1

```
int getchar(void)
{  unsigned char c;
   if (read(0, &c, 1) == 1)
      return (int)c;
   else
      return EOF;
}
```

```
int putchar(int c)
{  if (write(1, &c, 1) == 1)
      return c;
   else
      return EOF;
}
```

Implementing Buffering

Problem: poor performance

- **`read()`** and **`write()`** access a physical device (e.g., a disk)
- Reading/writing one char at a time can be time consuming
- Better to read and write in larger blocks
  - Recall ***Storage Management*** lecture

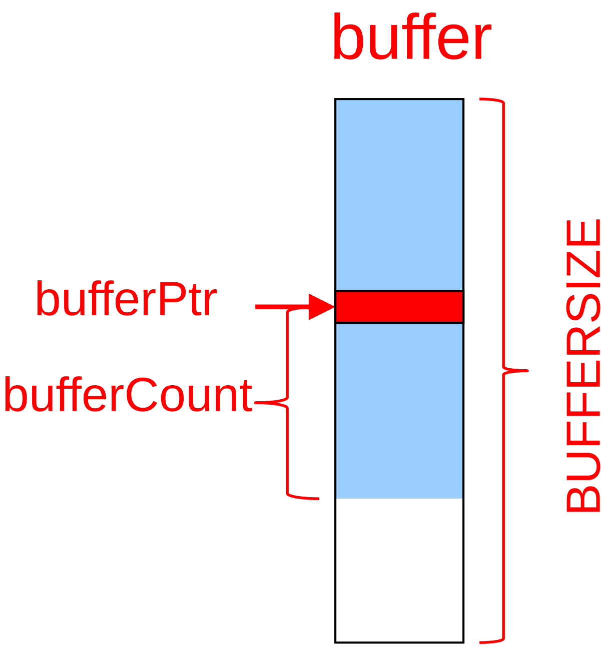
Solution: buffered I/O

- **Read** a large block of chars from source device into a buffer
  - Provide chars from buffer to the client as needed
- **Write** individual chars to a buffer
  - "Flush" buffer contents to destination device when buffer is full, or when file is closed, or upon client request

Implementing `getchar` Version 2

`getchar()` calls **`read()`** to read multiple chars from fd 0 into buffer

```
int getchar(void)
{  enum {BUFFERSIZE = 4096}; /*arbitrary*/
   static unsigned char buffer[BUFFERSIZE];
   static unsigned char *bufferPtr;
   static int bufferCount = 0;
   if (bufferCount == 0) /* must read */
   {  bufferCount =
         read(0, buffer, BUFFERSIZE);
      if (bufferCount <= 0) return EOF;
      bufferPtr = buffer;
   }
   bufferCount--;
   bufferPtr++;
   return (int)(*(bufferPtr-1));
}
```



Implementing `putchar` Version 2

`putchar()` calls **`write()`** to write multiple chars from buffer to fd 1

```
int putchar(int c)
{  enum {BUFFERSIZE = 4096};
   static char buffer[BUFFERSIZE];
   static int bufferCount = 0;
   if (bufferCount == BUFFERSIZE) /* must write */
   {  int countWritten = 0;
      while (countWritten < bufferCount)
      {  int count =
            write(1, buffer+countWritten, BUFFERSIZE-countWritten);
         if (count <= 0) return EOF;
         countWritten += count;
      }
      bufferCount = 0;
   }
   buffer[bufferCount] = (char)c;
   bufferCount++;
   return c;
}
```

Real implementation also flushes buffer at other times

Implementing the `FILE` ADT

Observation:

- **`getchar()`** reads from **`stdin`** (fd 0)
- **`putchar()`** writes to **`stdout`** (fd 1)

Problem:

- How to read/write from/to files other than **`stdin`** (fd 0) and **`stdout`** (fd 1)?
- Example: How to define **`fgetc()`** and **`fputc()`**?

Solution:

- Use **`FILE`** structure

Implementing the FILE ADT

```
enum {BUFFERSIZE = 4096};

struct File
{  unsigned char  buffer[BUFFERSIZE];  /* buffer */
   int             bufferCount; /* num chars left in buffer */
   unsigned char *bufferPtr;   /* ptr to next char in buffer */
   int             flags;      /* open mode flags, etc. */
   int             fd;         /* file descriptor */
};

typedef struct File FILE;

/* Initialize standard files. */
FILE *stdin  = …
FILE *stdout = …
FILE *stderr = …
```

Derived from K&R Section 8.5

More complex on our system

Implementing `fopen` and `fclose`

`f = fopen(filename, "r")`

- Create new **`FILE`** structure; set **`f`** to point to it
- Initialize all fields
- **`f->fd = open(filename, …)`**
- Return **`f`**

`f = fopen(filename, "w")`

- Create new **`FILE`** structure; set **`f`** to point to it
- Initialize all fields
- **`f->fd = creat(filename, …)`**
- Return **`f`**

`fclose(f)`

- **`close(f->fd)`**
- Destroy **`FILE`** structure

Implementing `fgetc`

```
int fgetc(FILE *f)
{  if (f->bufferCount == 0) /* must read */
   {  f->bufferCount =
            read(f->fd, f->buffer, BUFFERSIZE);
      if (f->bufferCount <= 0) return EOF;
      f->bufferPtr = f->buffer;
   }
   f->bufferCount--;
   f->bufferPtr++;
   return (int)(*(f->bufferPtr-1));
}
```

- Accepts FILE pointer f as parameter
- Uses fields within f
- Reads from f->fd instead of 0

Implementing `fputc`

```
int fputc(int c, FILE *f)
{  if (f->bufferCount == BUFFERSIZE) /* must write */
   {  int countWritten = 0;
      while (countWritten < f->bufferCount)
      {  int count =
            write(f->fd, f->buffer+countWritten,
                  BUFFERSIZE-countWritten);
         if (count <= 0) return EOF;
         countWritten += count;
      }
      f->bufferCount = 0;
   }
   f->buffer[f->bufferCount] = (char)c;
   f->bufferCount++;
   return c;
}
```

Real implementation also flushes buffer at other times

- Accepts FILE pointer f as parameter
- Uses fields within f
- Writes to f->fd instead of 1

Implementing Standard C I/O Functions

Standard C Function	In Unix Implemented by Calling
fopen()	**open()** or **creat()**
fclose()	**close()**

Implementing Standard C I/O Functions

Standard C Function	In Unix Implemented by Calling
fgetc()	**read()**
getchar()	fgetc()
fgets()	fgetc()
gets()	fgets()
fscanf()	fgetc()
scanf()	fscanf()

Implementing Standard C I/O Functions

Standard C Function	In Unix Implemented by Calling
fputc()	**write()**
putchar()	fputc()
fputs()	fputc()
puts()	fputs()
fprintf()	fputc()
printf()	fprintf()

Implementing Standard C I/O Functions

Standard C Function	In Unix Implemented by Calling
fflush()	**write()**
fseek()	**lseek()**
ftell()	**lseek()**

Agenda

The C/Unix file abstraction

Unix I/O system calls

C's Standard IO library (FILE *)

Implementing standard C I/O using Unix I/O

Redirecting standard files

Pipes

Redirection

Unix allows programmatic redirection of **`stdin`**, **`stdout`**, or **`stderr`**

How?

- Use **`open()`**, **`creat()`**, and **`close()`** system-level functions
- Use **`dup()`** system-level function

`int dup(int oldfd);`

- Create a copy of file descriptor **`oldfd`**
- Old and new file descriptors may be used interchangeably; they refer to the same open file table and thus share file position and file status flags
- Uses the lowest-numbered unused descriptor for the new descriptor
- Returns the new descriptor, or -1 if an error occurred.

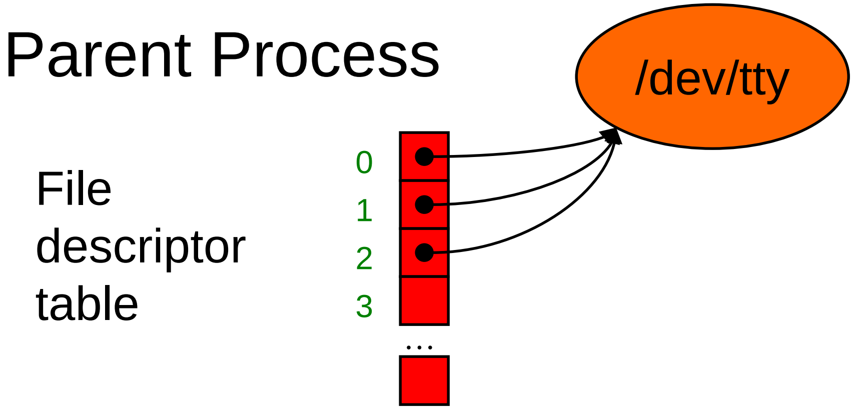
Paraphrasing man page

Redirection Example

How does shell implement **somepgm > somefile**?

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

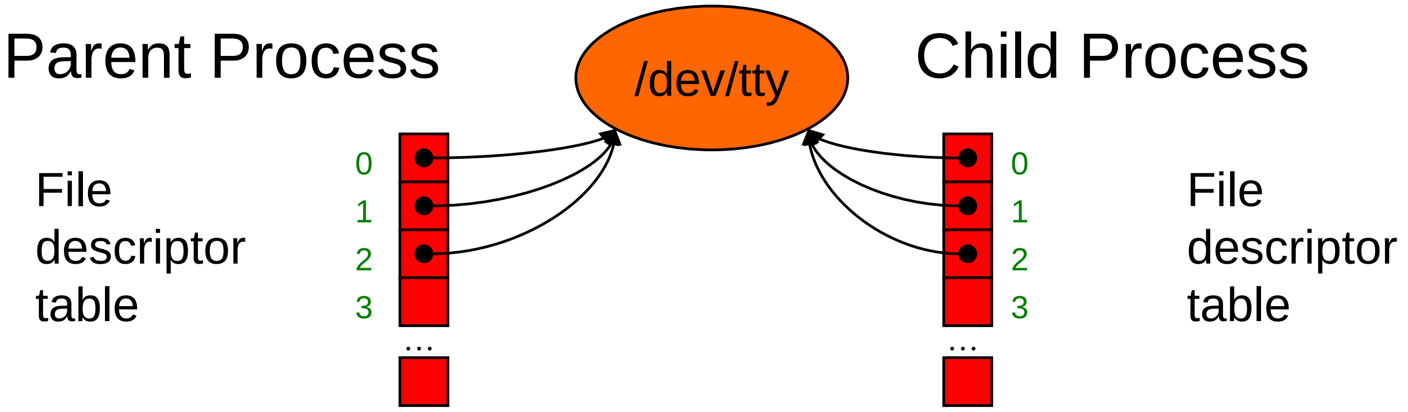
Redirection Example Trace (1)




```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

Parent has file descriptor table; first three point to "terminal"

Redirection Example Trace (2)

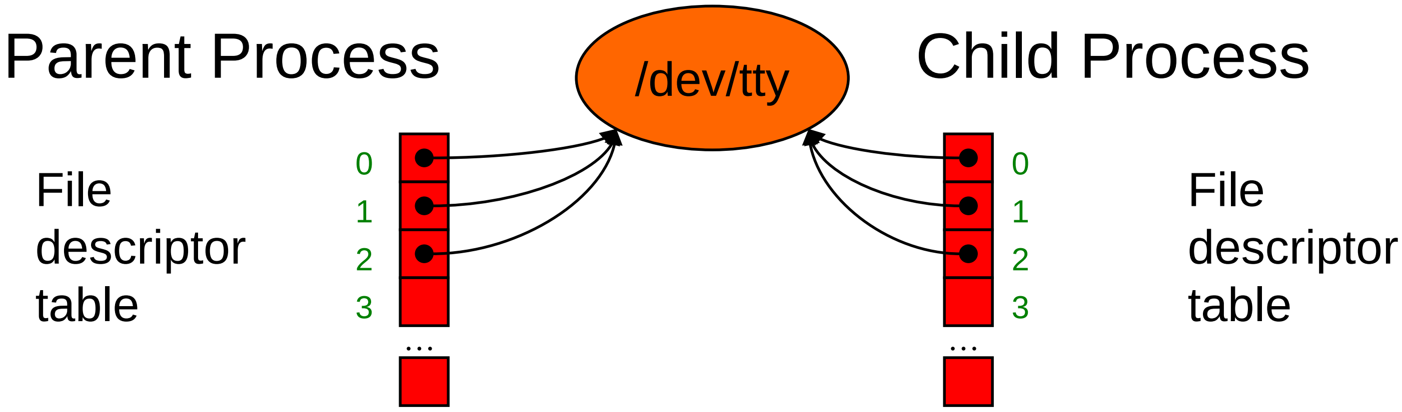



```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

Parent forks child; child has identical-but distinct file descriptor table

Redirection Example Trace (3)

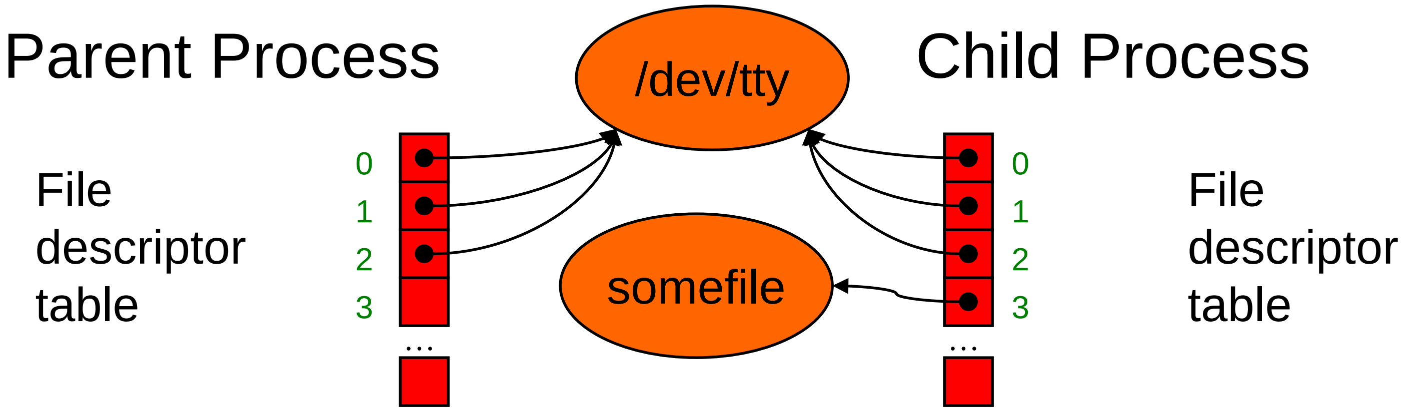



```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 060);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

Let’s say OS gives CPU to parent; parent waits

Redirection Example Trace (4)



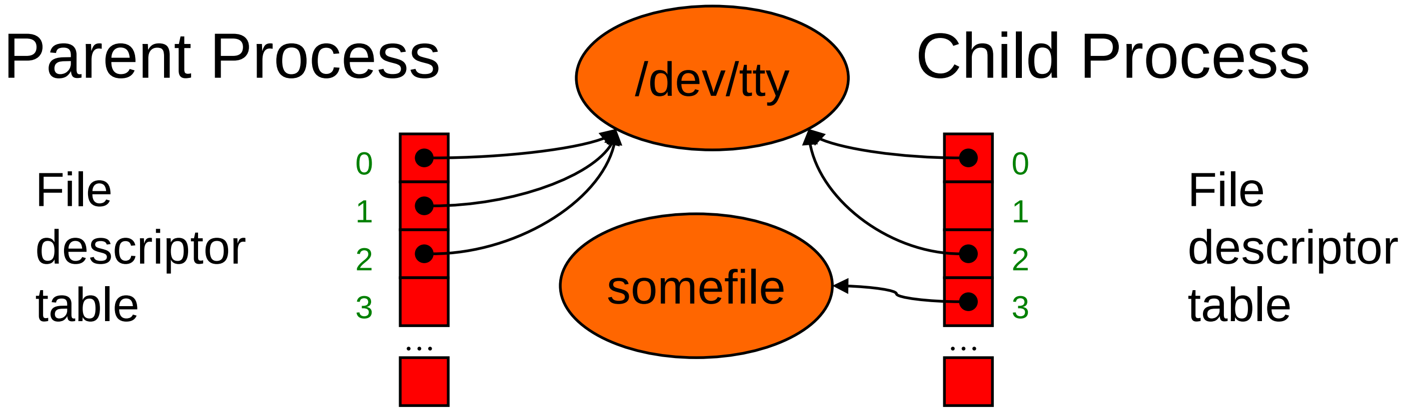

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

3

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

OS gives CPU to child; child creates somefile

Redirection Example Trace (5)



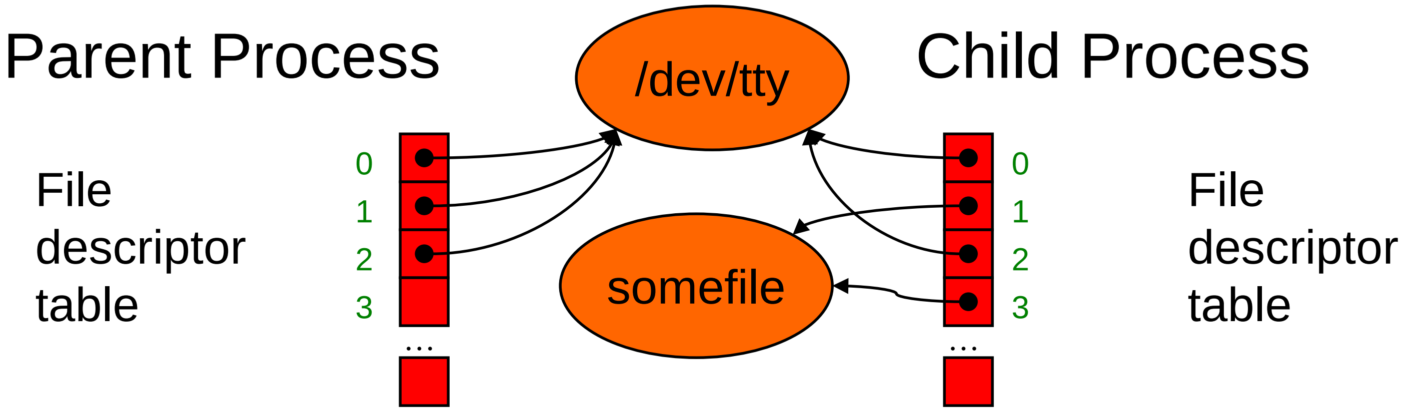

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

3

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

Child closes file descriptor 1 (stdout)

Redirection Example Trace (6)



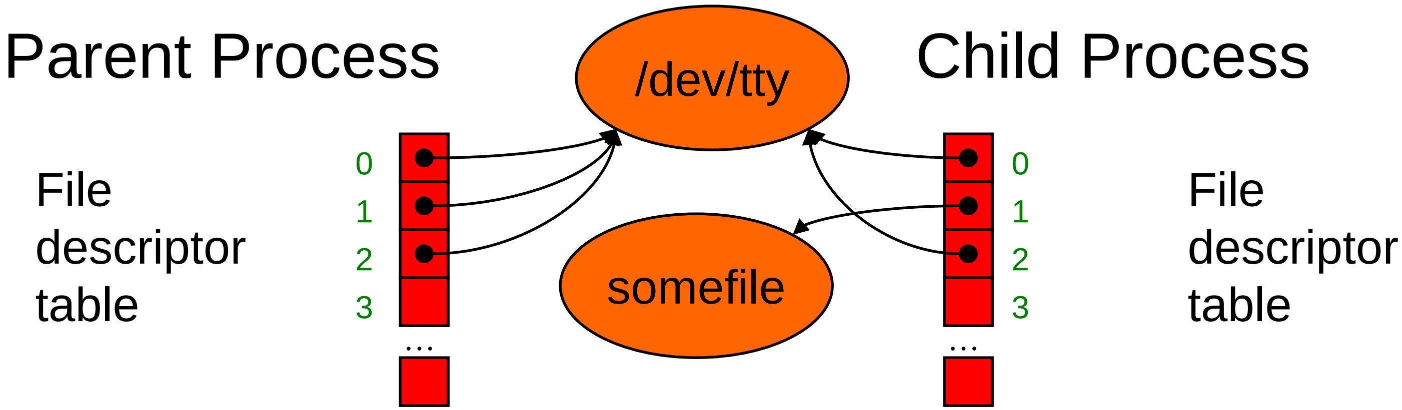

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

3

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

Child duplicates file descriptor 3 into first unused spot

Redirection Example Trace (7)



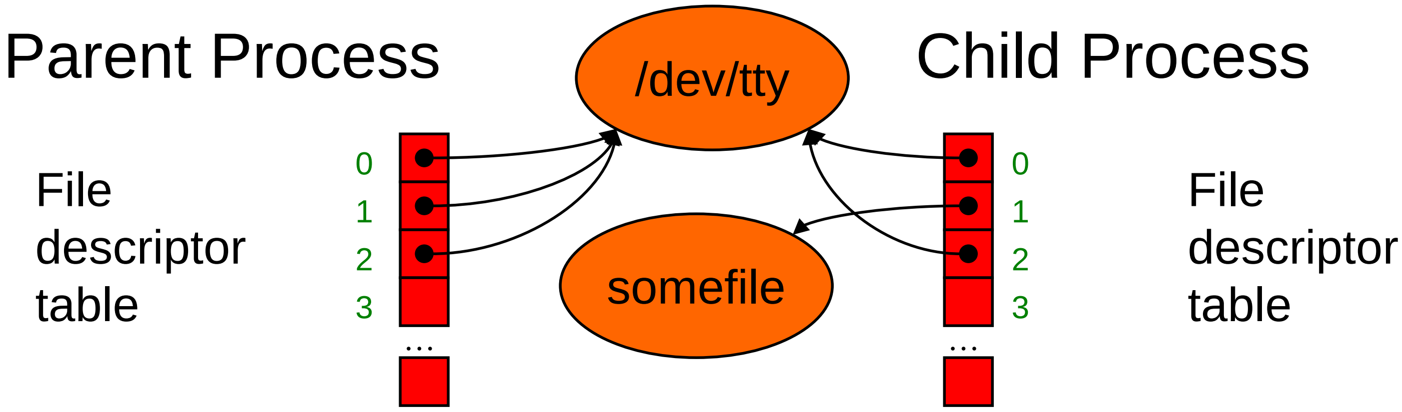

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

3

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

Child closes file descriptor 3

Redirection Example Trace (8)



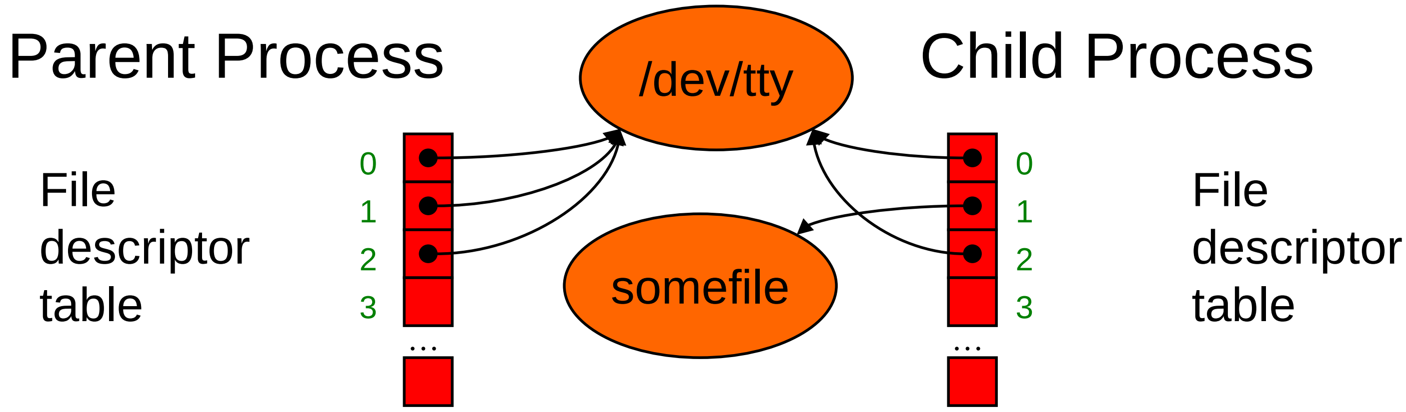

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

3

```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

Child calls **execvp()**

Redirection Example Trace (9)

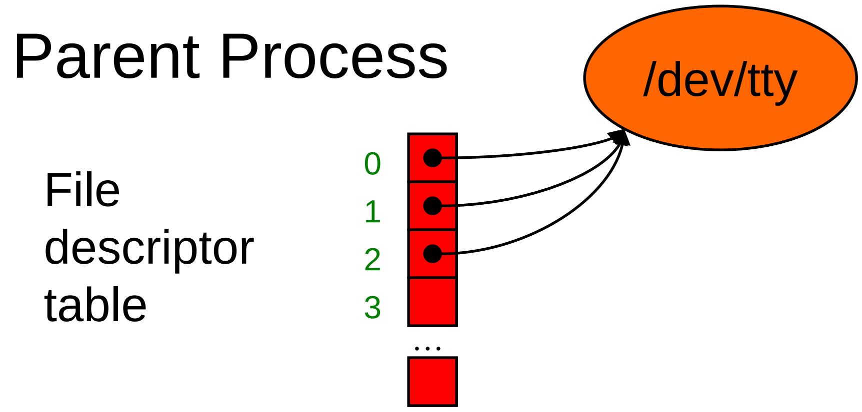



```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepfm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

somepgm

Somepgm executes with stdout redirected to somefile

Redirection Example Trace (10)




```
pid = fork();
if (pid == 0)
{  /* in child */
   fd = creat("somefile", 0600);
   close(1);
   dup(fd);
   close(fd);
   execvp(somefile, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
```

Somepgm exits; parent returns from **`wait()`** and proceeds

Agenda

The C/Unix file abstraction

Unix I/O system calls

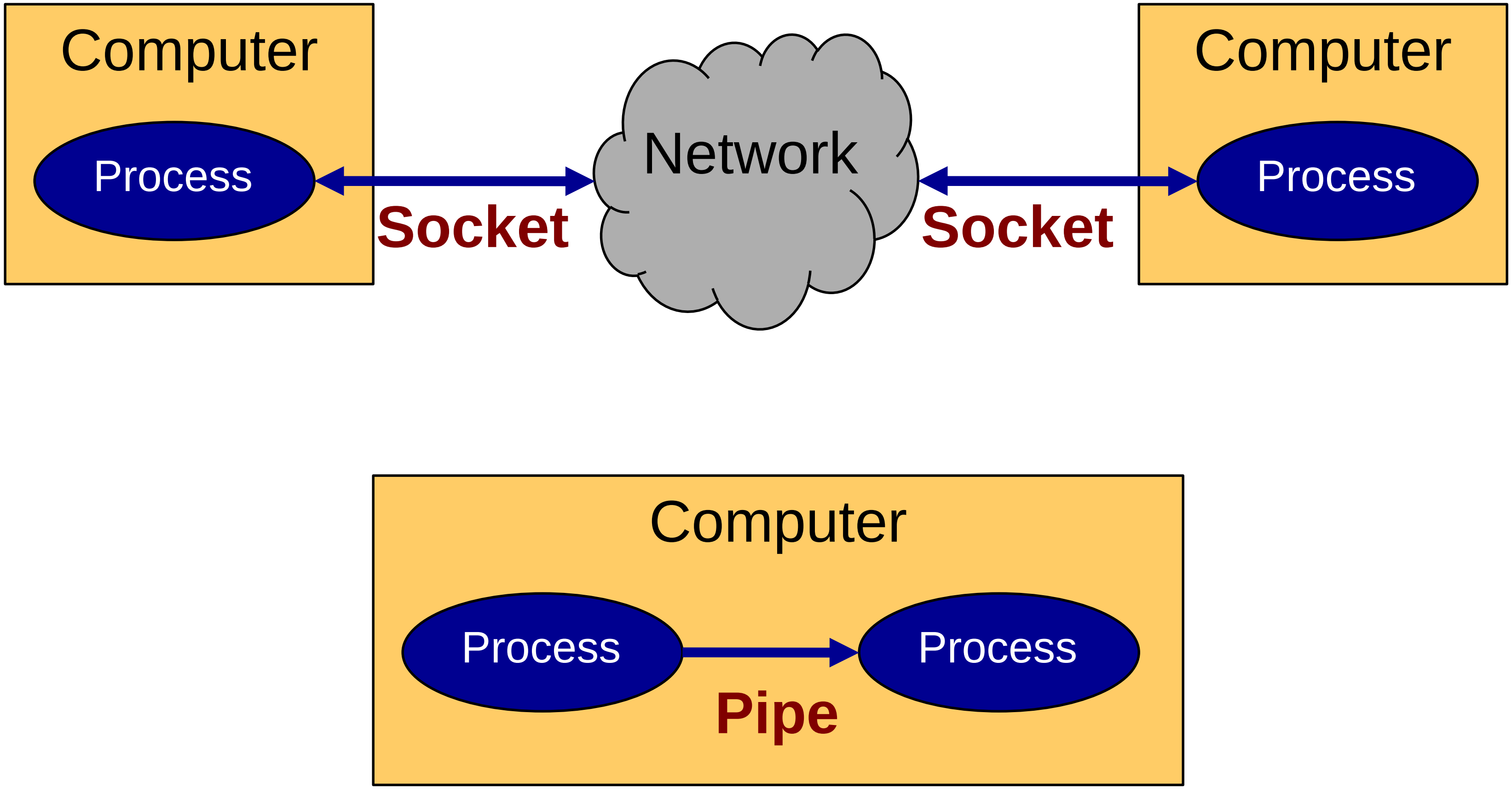
C's Standard IO library (FILE *)

Implementing standard C I/O using Unix I/O

Redirecting standard files

Pipes

Inter-Process Communication (IPC)



IPC Mechanisms

Socket

- Mechanism for **two-way** communication between processes on **any computers** on same network
- Processes created independently
- Used for client/server communication (e.g., Web)

Pipe

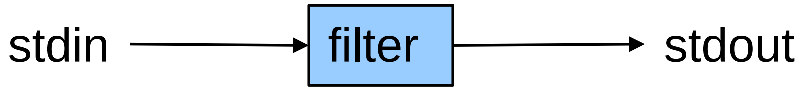
- Mechanism for **one-way** communication between processes on the **same computer**
- Allows parent process to communicate with child process
- Allows two “sibling” processes to communicate
- Used mostly for a **pipeline** of **filters**

Both support **file** abstraction

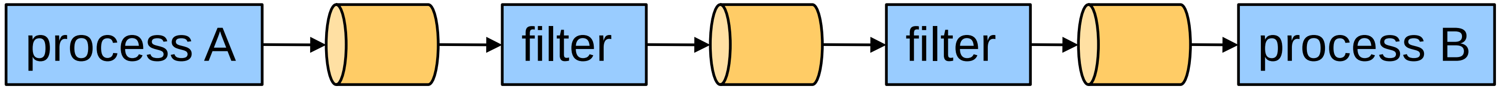
Pipes, Filters, and Pipelines

Pipe

Filter: Program that reads from stdin and writes to stdout



Pipeline: Combination of pipes and filters



Pipeline Examples

When debugging your shell program…

`grep alloc *.c`

- In all of the .c files in the working directory, display all lines that contain "alloc"

`cat *.c | decomment | grep alloc`

- In all of the .c files in the working directory, display all non-comment lines that contain "alloc"

`cat *.c | decomment | grep alloc | more`

- In all of the .c files in the working directory, display all non-comment lines that contain "alloc", one screen at a time

Creating a Pipe

`int pipe(int pipefd[2])`

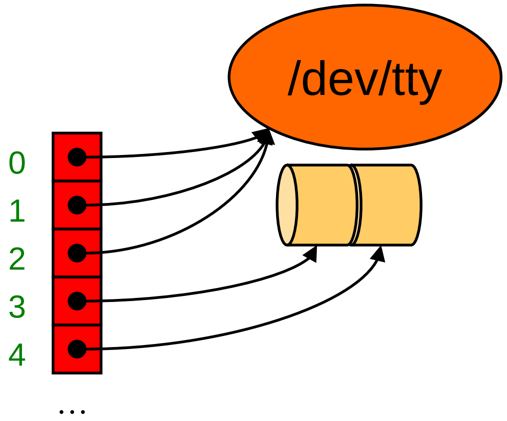
- **`pipe()`** creates a pipe, a unidirectional data channel that can be used for interprocess communication
- The array **`pipefd`** is used to return two file descriptors referring to the ends of the pipe
- **`pipefd[0]`** refers to the read end of the pipe
- **`pipefd[1]`** refers to the write end of the pipe
- Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe

  - Quoting **`man –s2 pipe`**

Pipe Example 1 (1)

Parent process sends data to child process

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(p[1]);
   /* Read from fd p[0] */
   exit(0);
}
/* in parent */
close(p{0});
/* Write to fd p[1] */
wait(NULL);
```

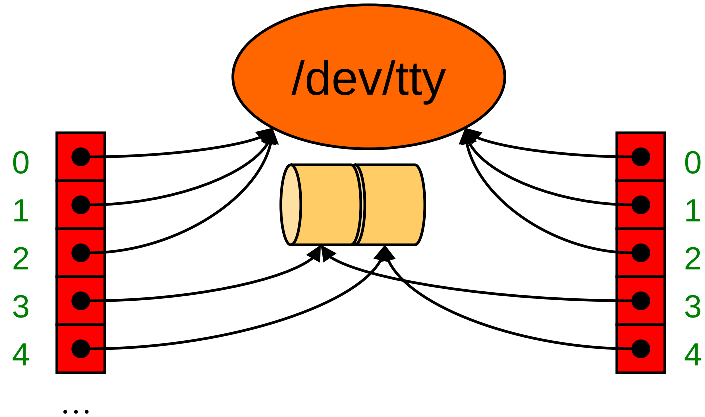


p[0] = 4
p[1] = 3

Pipe Example 1 (2)

Parent process sends data to child process

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(p[1]);
   /* Read from fd p[0] */
   exit(0);
}
/* in parent */
close(p{0});
/* Write to fd p[1] */
wait(NULL);
```



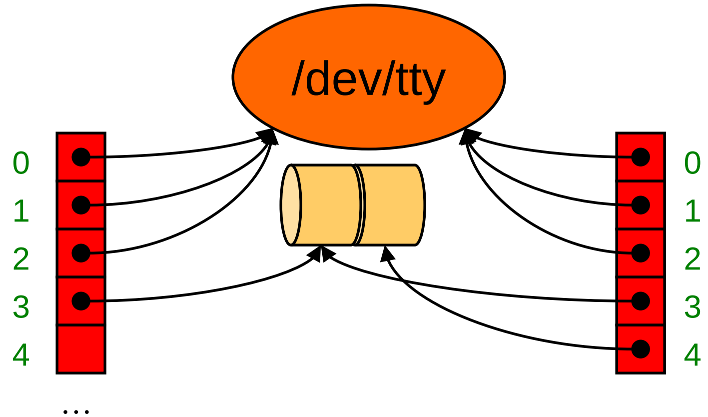
p[0] = 4
p[1] = 3


```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(p[1]);
   /* Read from fd p[0] */
   exit(0);
}
/* in parent */
close(p{0});
/* Write to fd p[1] */
wait(NULL);
```

Pipe Example 1 (3)

Parent process sends data to child process

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(p[1]);
   /* Read from fd p[0] */
   exit(0);
}
/* in parent */
close(p{0});
/* Write to fd p[1] */
wait(NULL);
```



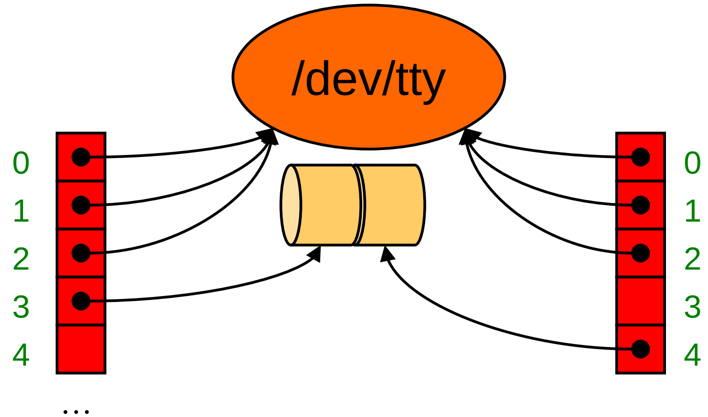

```
p[0] = 4
p[1] = 3
```

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(p[1]);
   /* Read from fd p[0] */
   exit(0);
}
/* in parent */
close(p{0});
/* Write to fd p[1] */
wait(NULL);
```

Pipe Example 1 (4)

Parent process sends data to child process

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(p[1]);
   /* Read from fd p[0] */
   exit(0);
}
/* in parent */
close(p{0});
/* Write to fd p[1] */
wait(NULL);
```



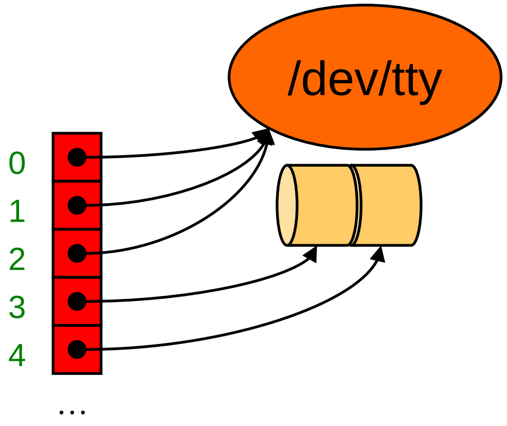
p[0] = 4
p[1] = 3

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(p[1]);
   /* Read from fd p[0] */
   exit(0);
}
/* in parent */
close(p{0});
/* Write to fd p[1] */
wait(NULL);
```

Pipe Example 2 (1)

Parent sends data to child through stdin/stdout

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   /* Read from stdin */
   exit(0);
}
/* in parent */
close(1);
dup(p[1])
close(p[1]);
close(p[0]);
/* write to stdout */
wait(NULL);
```

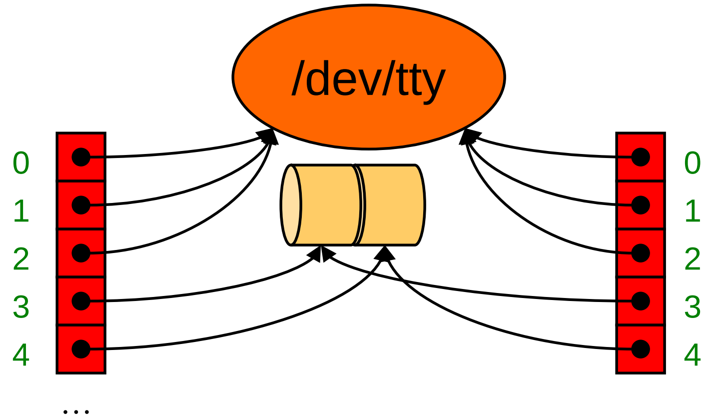



```
p[0] = 4
p[1] = 3
```

Pipe Example 2 (2)

Parent sends data to child through stdin/stdout

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   /* Read from stdin */
   exit(0);
}
/* in parent */
close(1);
dup(p[1])
close(p[1]);
close(p[0]);
/* write to stdout */
wait(NULL);
```



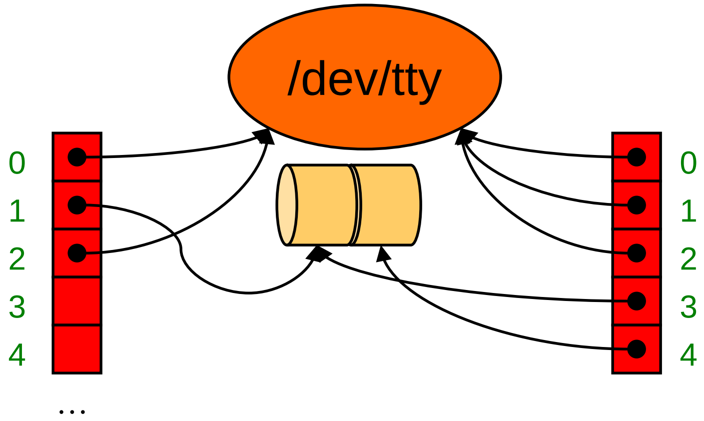

```
p[0] = 4
p[1] = 3
```

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   /* Read from stdin */
   exit(0);
}
/* in parent */
close(1);
dup(p[1])
close(p[1]);
close(p[0]);
/* write to stdout */
wait(NULL);
```

Pipe Example 2 (3)

Parent sends data to child through stdin/stdout

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   /* Read from stdin */
   exit(0);
}
/* in parent */
close(1);
dup(p[1])
close(p[1]);
close(p[0]);
/* write to stdout */
wait(NULL);
```



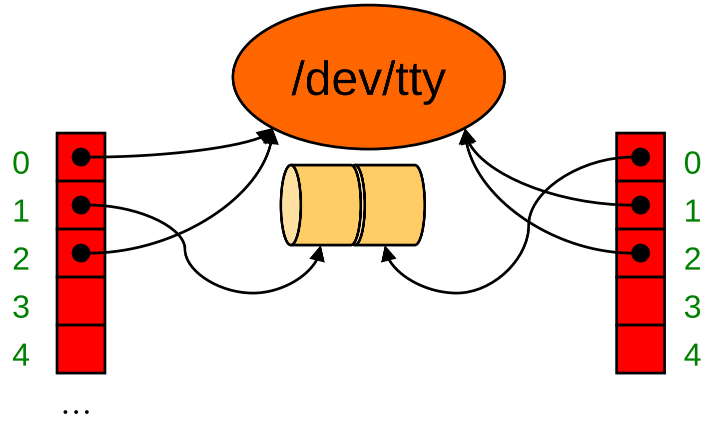

```
p[0] = 4
p[1] = 3
```

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   /* Read from stdin*/
   exit(0);
}
/* in parent */
close(1);
dup(p[1])
close(p[1]);
close(p[0]);
/* write to stdout */
wait(NULL);
```

Pipe Example 2 (4)

Parent sends data to child through stdin/stdout

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   /* Read from stdin */
   exit(0);
}
/* in parent */
close(1);
dup(p[1])
close(p[1]);
close(p[0]);
/* write to stdout */
wait(NULL);
```



p[0] = 4
p[1] = 3

Now add in execs, and you get the shell's implementation of pipes!

```
int p[2];
…
pipe(p)
pid = fork();
if (pid == 0)
{  /* in child */
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   /* Read from stdin */
   exit(0);
}
/* in parent */
close(1);
dup(p[1])
close(p[1]);
close(p[0]);
/* write to stdout */
wait(NULL);
```

Summary

The C/Unix file abstraction

Unix I/O

- File descriptors, file descriptor tables, file tables
- **creat()**, **open()**, **close()**, **read()**, **write()**, **lseek()**

C's Standard I/O

- **FILE** structure
- **fopen()**, **fclose(),** **fgetc()**, **fputc()**, …

Implementing standard C I/O using Unix I/O

- Buffering

Redirecting standard files

- **dup()**

Pipes

- **pipe()**