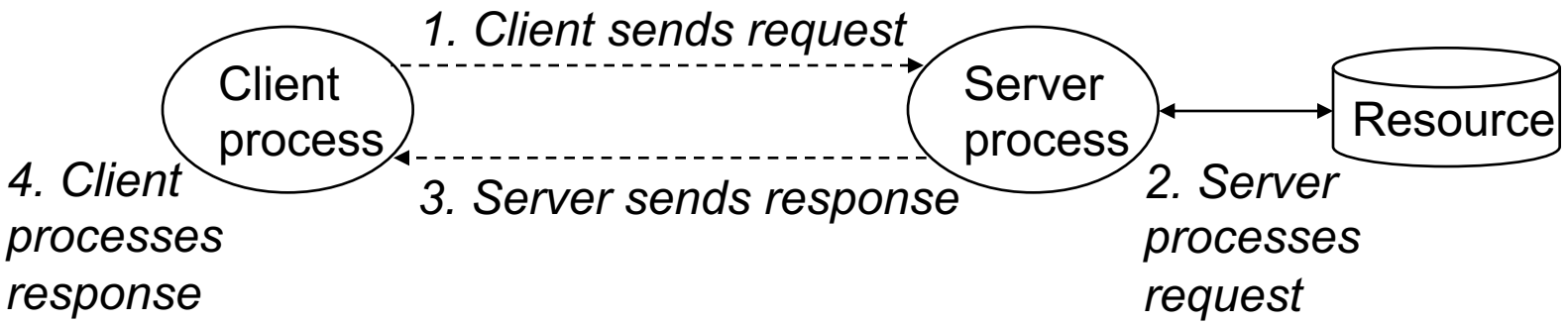
456: Network-Centric Programming

Yingying (Jennifer) Chen

Web: https://www.winlab.rutgers.edu/~yychen/
Email: yingche@scarletmail.rutgers.edu

Department of Electrical and Computer Engineering
Rutgers University

The Client–Server Model



Autotools

- De-facto standard for creating portable, complete, and self-contained program distributions
  - Standard build commands

    ./configure; make
    sudo make install
- Configure to automatically adjust the makefile
- A set of tools in the autotools package
  - *automake*
    - Examine how sources files depend on each other and generate a Makefile to compile the files in the correct order
  - *autoconf*
    - Permit automatic configuration of software installation, handling a large number of system features to increase portability

Autotools Example: code/lecture_3_code/autotools

- hello.c

Using autotools to create and build the HelloWorld project.

```
#include <stdio.h>

int main(int argc, char** argv)
{
  printf("%s", "Hello, Linux World!\n");
  return 0;
}
```

Autotools Example – configuration files

- Need to provide two files
  - Makefile.am

    An input file to *automake* that specifies a projects build requirements: what needs to be built, and where it goes when installed.
  - configure.ac (or configure.in)

    An input file to *autoconf* that provides the macro invocations and shell code fragments *autoconf* uses to build a configure script.

Autotools Example – configuration files

- Makefile.am

  bin_PROGRAMS = hello

  hello_SOURCES = hello.c

- configure.ac

macros

AC_INIT(hello.c)

AM_INIT_AUTOMAKE(hello,0.1)

AC_PROG_CC

AC_PROG_INSTALL

AC_OUTPUT(Makefile)

Any source file

AC_PROG_CXX

Macros in configure.ac

- The AC_INIT
- The AM_INIT_AUTOMAKE
  - Does all the standard initialization required by *automake* and takes two arguments, the package name and version number.
- The AC_PROG_CXX
  - Checks for the C++ compiler and sets the variables CXX
- AC_OUTPUT
  - Must be called at the end of configure.in to create the Makefiles.

Autotools Example – calling the autotools programs

- % aclocal
  - aclocal program will automatically generate aclocal.m4 files based on the contents of configure.in.
- % autoconf
  - autoconf produce the configuration script.
- % touch NEWS README AUTHORS ChangeLog
  - Build a couple of necessary files (e.g., NEWS, README, etc.)

```
chen@chen-VirtualBox:~/autotools$ ls
configure.ac  hello.c  Makefile.am
chen@chen-VirtualBox:~/autotools$ aclocal
chen@chen-VirtualBox:~/autotools$ ls
aclocal.m4  autom4te.cache  configure.ac  hello.c  Makefile.am
chen@chen-VirtualBox:~/autotools$ autoconf
chen@chen-VirtualBox:~/autotools$ ls
aclocal.m4  autom4te.cache  configure  configure.ac  hello.c  Makefile.am
chen@chen-VirtualBox:~/autotools$ touch NEWS README AUTHORS ChangeLog
chen@chen-VirtualBox:~/autotools$ ls
aclocal.m4  autom4te.cache  configure     hello.c      NEWS
AUTHORS     ChangeLog       configure.ac  Makefile.am  README
```

Three files at the beginning

Autotools Example – generating Makefile

- % automake --add-missing
  - automake produces Makefile.in based on makefile.am

```
chen@chen-VirtualBox:~/autotools$ automake --add-missing
configure.ac:2: warning: AM_INIT_AUTOMAKE: two- and three-arguments forms are deprecated.  For mor
e info, see:
configure.ac:2: http://www.gnu.org/software/automake/manual/automake.html#Modernize-AM_005fINIT_00
5fAUTOMAKE-invocation
configure.ac:3: installing './compile'
configure.ac:2: installing './install-sh'
configure.ac:2: installing './missing'
Makefile.am: installing './INSTALL'
Makefile.am: installing './COPYING' using GNU General Public License v3 file
Makefile.am:     Consider adding the COPYING file to the version control system
Makefile.am:     for your code, to avoid questions about which license your project uses
Makefile.am: installing './depcomp'
chen@chen-VirtualBox:~/autotools$ ls
aclocal.m4      ChangeLog  configure.ac  hello.c     Makefile.am  NEWS
AUTHORS         compile    COPYING       INSTALL     Makefile.in  README
autom4te.cache  configure  depcomp       install-sh  missing
```

Autotools Example – generating Makefile

- % ./configure
  - ./configure produces Makefile

```
chen@chen-VirtualBox:~/autotools$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
```

```
chen@chen-VirtualBox:~/autotools$ ls
aclocal.m4      ChangeLog   config.status  COPYING  INSTALL     Makefile.am  NEWS
AUTHORS         compile     configure      depcomp  install-sh  Makefile.in  README
autom4te.cache  config.log  configure.ac   hello.c  Makefile    missing
```

Autotools Example – generating Makefile

- % make
- % ./hello

```
chen@chen-VirtualBox:~/autotools$ make
gcc -DPACKAGE_NAME=\"\" -DPACKAGE_TARNAME=\"\" -DPACKAGE_VERSI
ON=\"\" -DPACKAGE_STRING=\"\" -DPACKAGE_BUGREPORT=\"\" -DPACKA
GE_URL=\"\" -DPACKAGE=\"hello\" -DVERSION=\"0.1\" -I.     -g -
O2 -MT hello.o -MD -MP -MF .deps/hello.Tpo -c -o hello.o hello
.c
mv -f .deps/hello.Tpo .deps/hello.Po
gcc  -g -O2   -o hello hello.o
chen@chen-VirtualBox:~/autotools$ ./hello
Hello, Linux World!
```

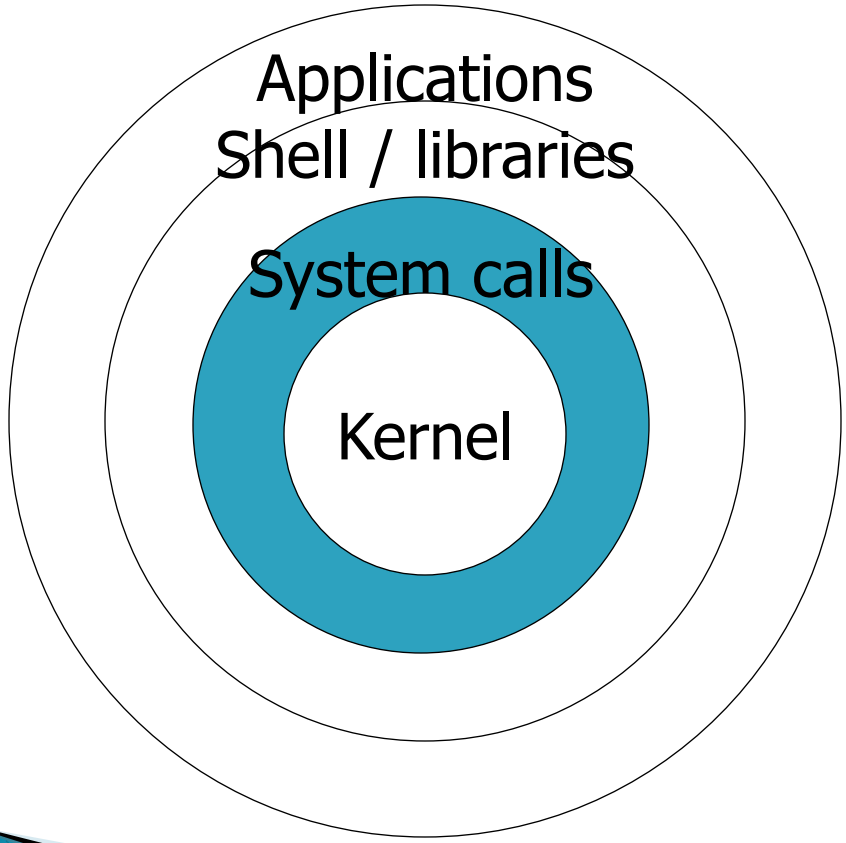
I/O is handled through System Calls

System Calls are functions/procedures provided by the operating system (rather than functions compiled into an application)

Operating System Functions

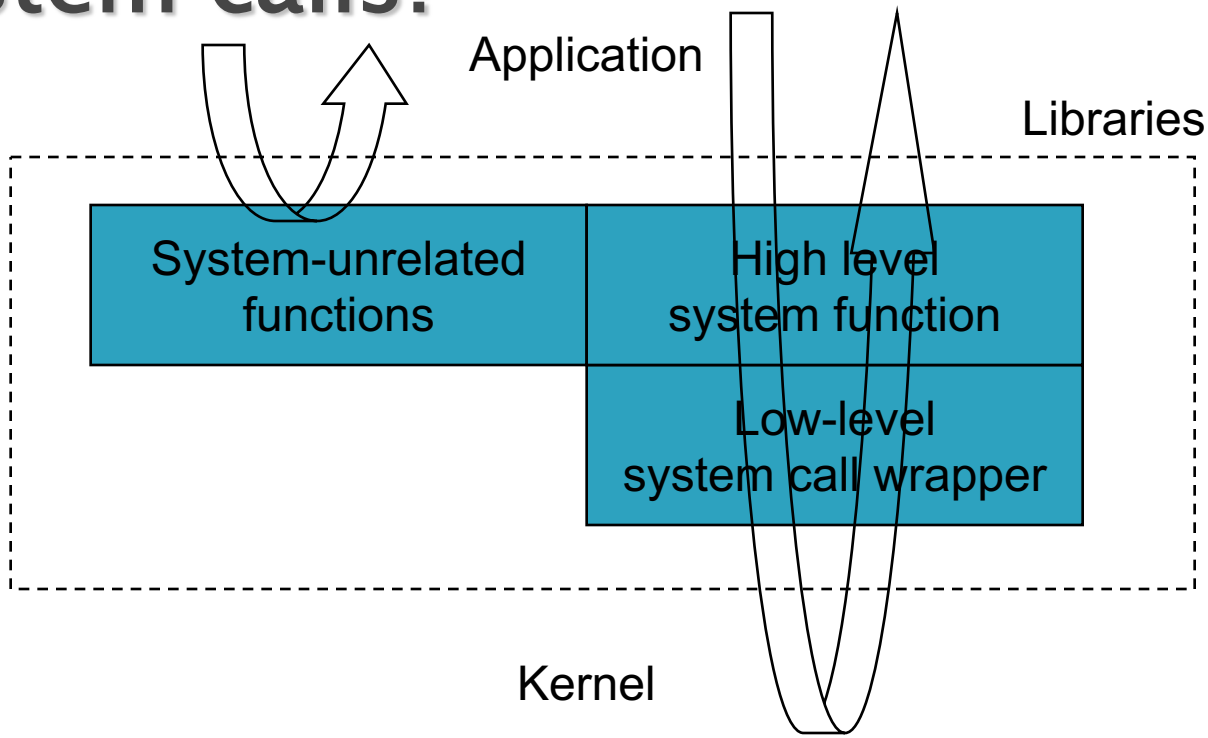
- Goals
  - Manage system resouces
  - Execute multiple applications simultaneously, without them being aware of each other
- Examples
  - CPU Sharing: Scheduled Processes
  - Memory Sharing: Virtual Address Space
  - Disk Sharing: Files
  - Network: Multiplexed Data streams

Linux Architecture – What are system calls?



- A call from a user mode process to a kernel function
- Expensive operation that requires mode switch
- Called via software interrupt but usually wrapped inside C library functions

Which library functions generate system calls?



- A system call typically performed by executing an interrupt.
- The interrupt causes the kernel to take over and perform the requested action, then hands control back to the application.
- This mode switching is the reason that system calls are slower to execute than an equivalent application-level function.

Strace tool can monitor system calls

- Try
  - strace touch x

touch command creates an empty (zero byte) new file called x

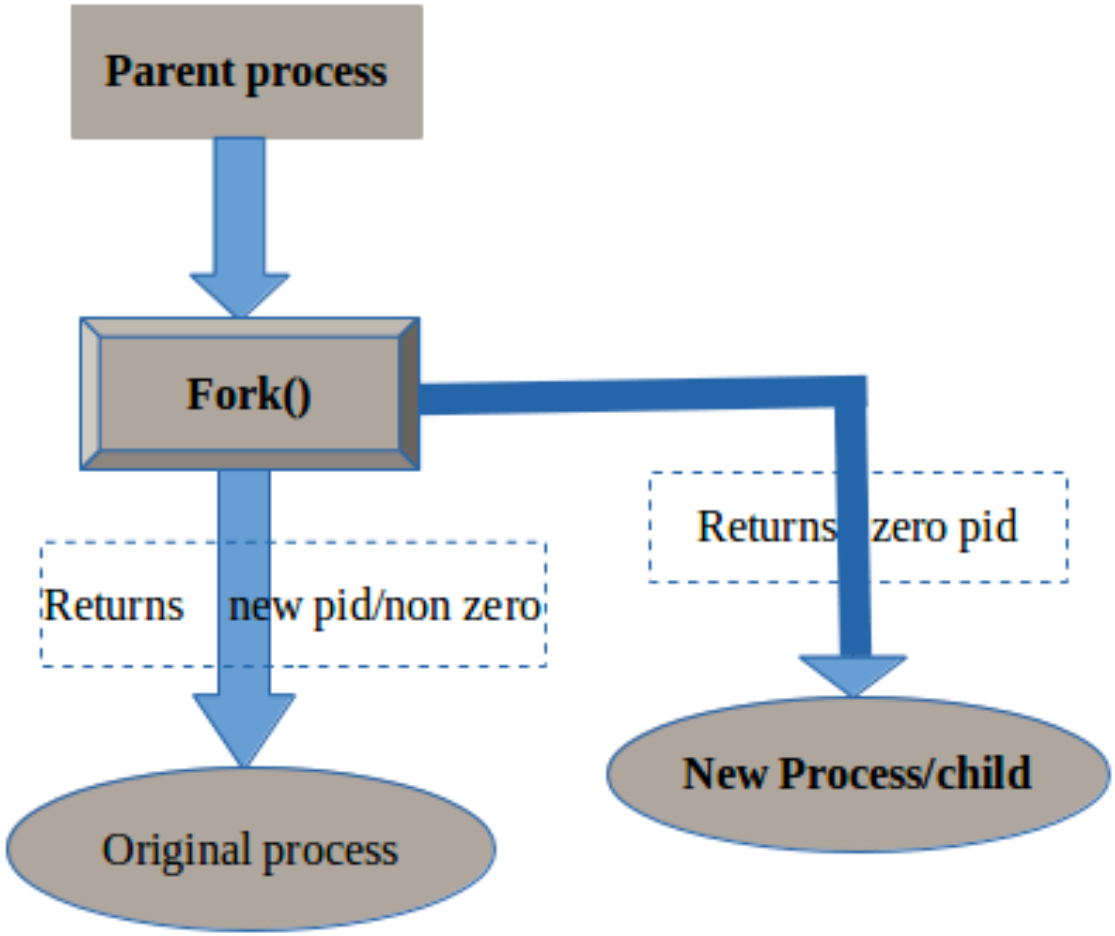
Strace touch x

```
-VirtualBox:~$ strace touch x
execve("/usr/bin/touch", ["touch", "x"], [/* 63 vars */]) = 0
brk(NULL)                               = 0xfab000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=89097, ...}) = 0
mmap(NULL, 89097, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fd78b048000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fd78b047000
mmap(NULL, 3971488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fd78aa6f000
mprotect(0x7fd78ac2f000, 2097152, PROT_NONE) = 0
mmap(0x7fd78ae2f000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c0000) = 0x7fd78ae2
f000
mmap(0x7fd78ae35000, 14752, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fd78ae35000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fd78b046000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fd78b045000
arch_prctl(ARCH_SET_FS, 0x7fd78b046700) = 0
mprotect(0x7fd78ae2f000, 16384, PROT_READ) = 0
mprotect(0x60e000, 4096, PROT_READ)     = 0
mprotect(0x7fd78b05e000, 4096, PROT_READ) = 0
munmap(0x7fd78b048000, 89097)           = 0
brk(NULL)                               = 0xfab000
brk(0xfcc000)                           = 0xfcc000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=2981280, ...}) = 0
mmap(NULL, 2981280, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fd78a797000
close(3)                                = 0
open("x", O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK, 0666) = 3
dup2(3, 0)                              = 0
close(3)                                = 0
utimensat(0, NULL, NULL, 0)             = 0
close(0)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Creating a Process

- Program is an executable file on disk
- Process represents an executing program
- fork() – system call to 'copy' a process (sole way to create processes in UNIX)
- The new process created by fork() is called the *child* process. This function is called once but returns twice.
  - The difference is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.

Creating a Process



fork() and getpid()

#include <unistd.h>
pid_t fork(void);

- the fork() command makes a complete copy of the running process and the only way to differentiate the two is by looking at the returned value:
- fork() returns the process identifier (pid) of the child process in the parent, and
- fork() returns 0 in the child.

#include <unistd.h>
pid_t getpid(void);

- getpid() returns the process ID (PID) of the calling process.

waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

- pid_t pid specifies the child processes the caller wants to wait for
- int *status points to a location where waitpid() can store a status value. The status pointer may also be NULL, in which case waitpid() ignores the child's return status.
- int options specifies additional information for waitpid(). (default 0)

Example: code/lecture_3_code/fork

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
//Using fork() to duplicate a program's process and show its pid

int main () {
    pid_t child_pid;
    printf ("the main program process ID is %d\n", (int) getpid ());
    child_pid = fork ();
    if (child_pid != 0) {      // parent process
        printf ("this is the parent process, with id %d\n", (int) getpid ());
        printf ("the child's process ID is %d\n", (int) child_pid);
    }else
        printf ("this is the child process, with id %d\n", (int) getpid ());
    return 0;
}
```

Book: Advanced Programming in the UNIX® Environment
Section 1.6 code/lecture_3_code/process_control

```
char   buf[MAXLINE];
pid_t  pid;
int                status;

printf("%% ");     /* print prompt (printf requires %% to print %) */
while (fgets(buf, MAXLINE, stdin) != NULL) {
        buf[strlen(buf) - 1] = 0;        /* replace newline with null */

        if ( (pid = fork()) < 0)
                err_sys("fork error");

        else if (pid == 0) {                /* child */
                execlp(buf, buf, (char *) 0); // execute commands in child process
                err_ret("couldn't execute: %s", buf);
                exit(127);
        }

        /* parent */
        if ( (pid = waitpid(pid, &status, 0)) < 0)
                err_sys("waitpid error");
        printf("%% ");
}
exit(0);
```

Fork system call

Key I/O System Calls

- Transferring chunks of bytes from program to file descriptor
  - read
  - write
- Adding/Deleting file descriptors to files
  - open
  - close
- Here the abstraction breaks down – this is only for real files
  - lseek

- Open files are identified by *file descriptor* (nonnegative integer)
- Three default file descriptors (shells establish this convention)
  - 0 = STDIN_FILENO
  - 1 = STDOUT_FILENO
  - 2 = STDERR_FILENO

Bash redirection

- > redirect stdout to file
- < redirect stdin to file
- >> append stdout to file
- 2> redirect stderr to file
- 2>&1 redirect fd2 to fd1

Bash Examples:

- ls -l > a.txt
- ls -l > > a.txt
- cat nop.txt
- cat nop.txt 2> errorinfo.txt
- cat nop.txt > output.txt 2>&1

cat is a standard Unix utility that reads files sequentially, writing them to standard output.

Stdio Lib vs I/O System Calls

Unbuffered vs. Buffered I/O

- Unbuffered I/O
  - The term *unbuffered* means that each read or write invokes a system call in the kernel.
- Buffered I/O
  - The goal of the buffering is to use the minimum number of read and write calls (e.g., standard I/O library).

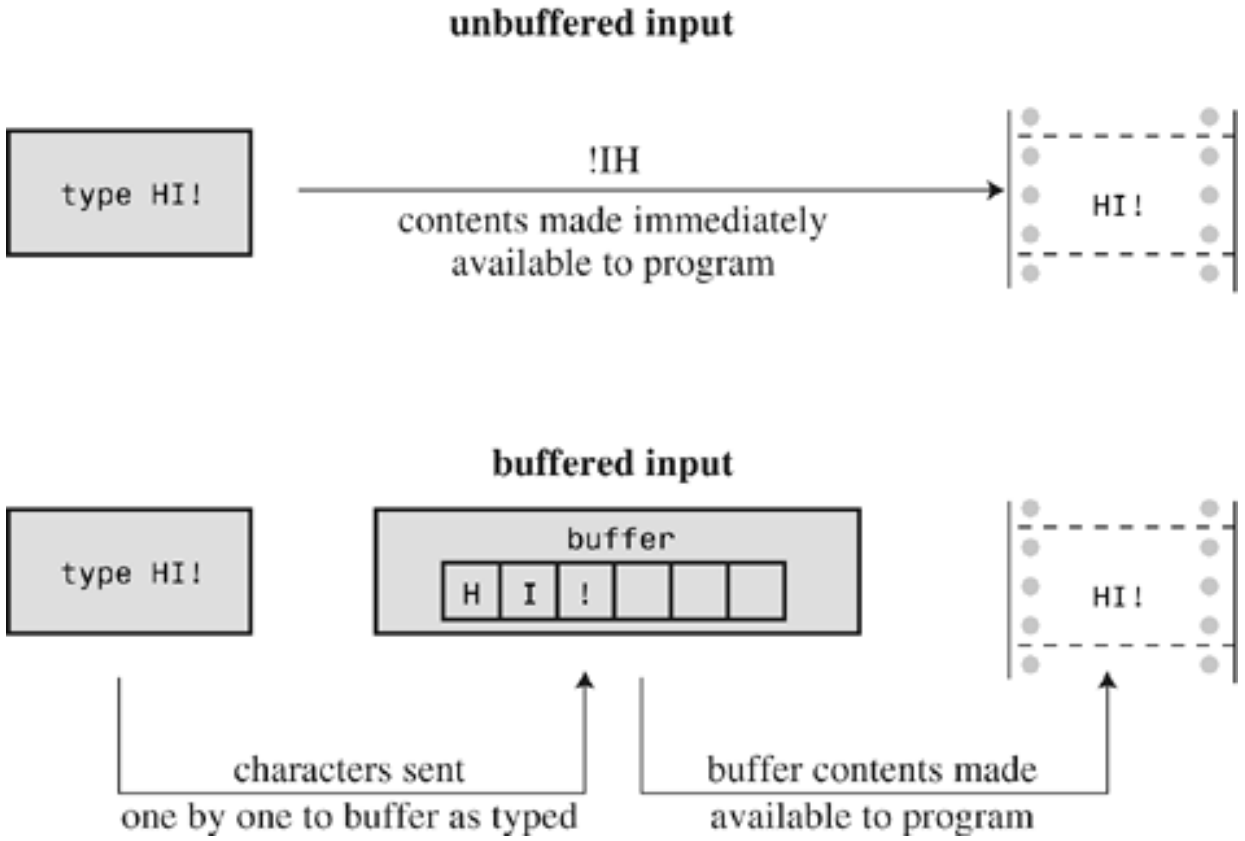
Standard I/O Library

- Portable C I/O library
  - first for Unix, now ANSI C – runs on Windows etc.
- Handles
  - **buffer allocation is automatic**: read into large buffer, dump to OS in fixed units
  - performs I/O in optimal-sized chunks

Key Functions

- fopen
  - returns pointer to FILE structure (*file pointer*)
  - pointer to buffer
- fclose
- fread
- fwrite
- fgetc / fputc
- fgets / fputs

Buffering


unbuffered input
type HI!
!IH
contents made immediately
available to program
HI!
buffered input
type HI!
buffer
H
I
!
HI!
characters sent
one by one to buffer as typed
buffer contents made
available to program

Buffering

- Minimizes number of read() and write() operations
- Three buffering modes:
  - fully buffered: I/O buffer is filled

```
void setbuf(FILE *fp, char *buf); // BUFSIZE
```

  - line-buffered: newline character

```
void setvbuf(FILE *fp, char *buf, int mode, size_t size);
```

  - unbuffered: as soon as possible

```
void setvbuf(FILE *fp, char *buf, int mode, size_t size);

int fflush(FILE *fp); //flush the buffer
```

Buffering

- Three buffering modes:
  - fully buffered: I/O buffer is filled

```
void setbuf(FILE *fp, char *buf); // BUFSIZE
```

  - line-buffered: newline character

```
void setvbuf(FILE *fp, NULL, _IOLBF, 0); // equivalent
void setlinebuf(FILE *fp); // equivalent
```

  - unbuffered: as soon as possible

```
void setvbuf(FILE *fp, NULL, _IONBF, 0);
```

The **setvbuf**() function may be used on any open stream to change its buffer. The *mode* argument must be one of the following three macros:

_IONBF unbuffered

_IOLBF line buffered

_IOFBF fully buffered

Buffering Example code/lecture_3_code/setbuf

```
#include <stdio.h>
int main (){
    char buffer[100];
    FILE *pFile1, *pFile2;
    pFile1=fopen ("myfile1.txt","w");
    pFile2=fopen ("myfile2.txt","w");

    setbuf ( pFile1 , buffer );    //fully buffered
    fputs ("This is sent to a buffered stream", pFile1);
    fflush (pFile1);

    setbuf ( pFile2 , NULL );
    fputs ("This is sent to an unbuffered stream",pFile2);

    fclose (pFile1);
    fclose (pFile2);
    return 0;
}
```

I/O Efficiency Difference between Stdio and System calls?

Comparison of Buffer Size Using System calls code/lecture_3_code/copy

```
#include "ourhdr.h"

#define   BUFFSIZE   10000

Int main(void)
{
    int      n;
    char   buf[BUFFSIZE];

    while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                    err_sys("write error");

    if (n < 0)
            err_sys("read error");

    exit(0);
}
```

$ time ./main < data_1MB.txt > received.txt

Buffer effect

Size	real(s)	user(s)	sys(s)
1	2.943	0.812	2.100
8	0.418	0.112	0.290
64	0.110	0.020	0.064
512	0.024	0.004	0.011
64K	0.014	0.002	0.003

Buffer effect

- **Real** is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).
- **User** is the amount of CPU time spent in user-mode code (outside the kernel) *within* the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- **Sys** is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls *within the kernel,* as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process. See below for a brief description of kernel mode (also known as 'supervisor' mode) and the system call mechanism.

Reading Homework Assignment

- Reading
  - CSAPP: Section 10.1–10.3, 10.5–10.10
  - APUE: Section 1.6