Systems Programming
Midterm Review

Tongping Liu

tongpingliu@cs.utsa.edu

Types of Questions

Selection (15%)

Answering Questions (30%)

Program Understanding (25%)

Programming (30%)

What we have covered?

Utility

Shell

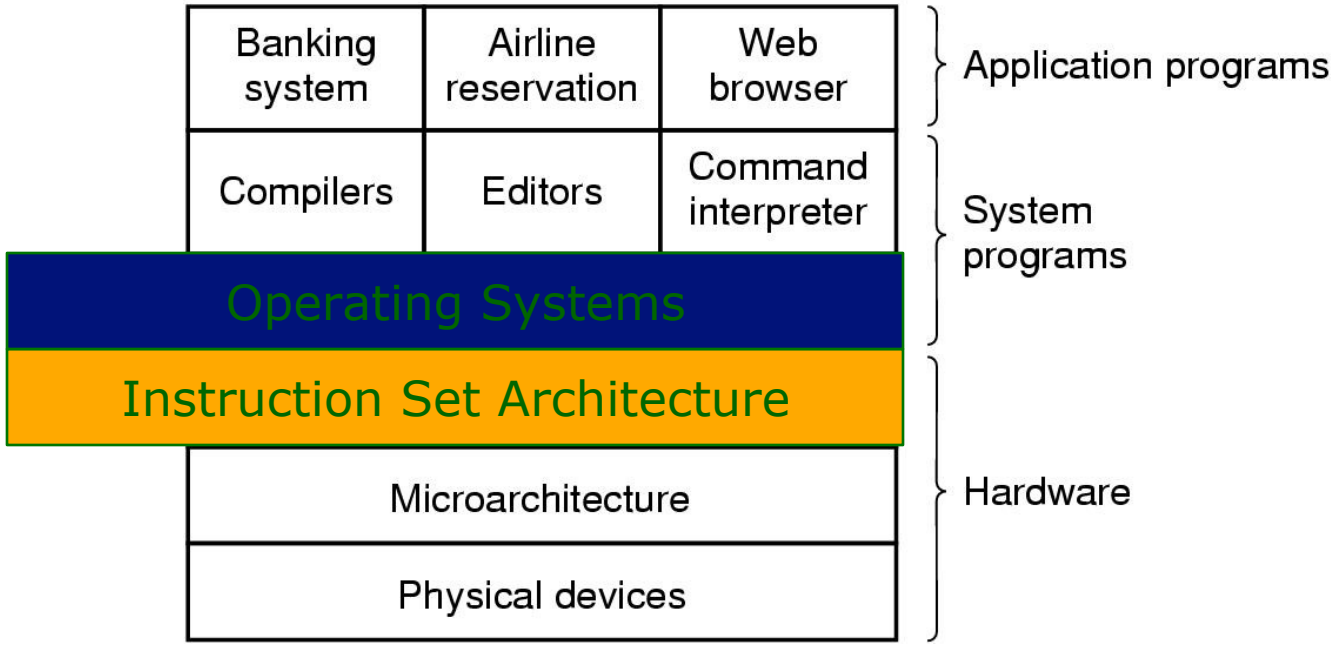
Awk

Sed

Perl

C Programming

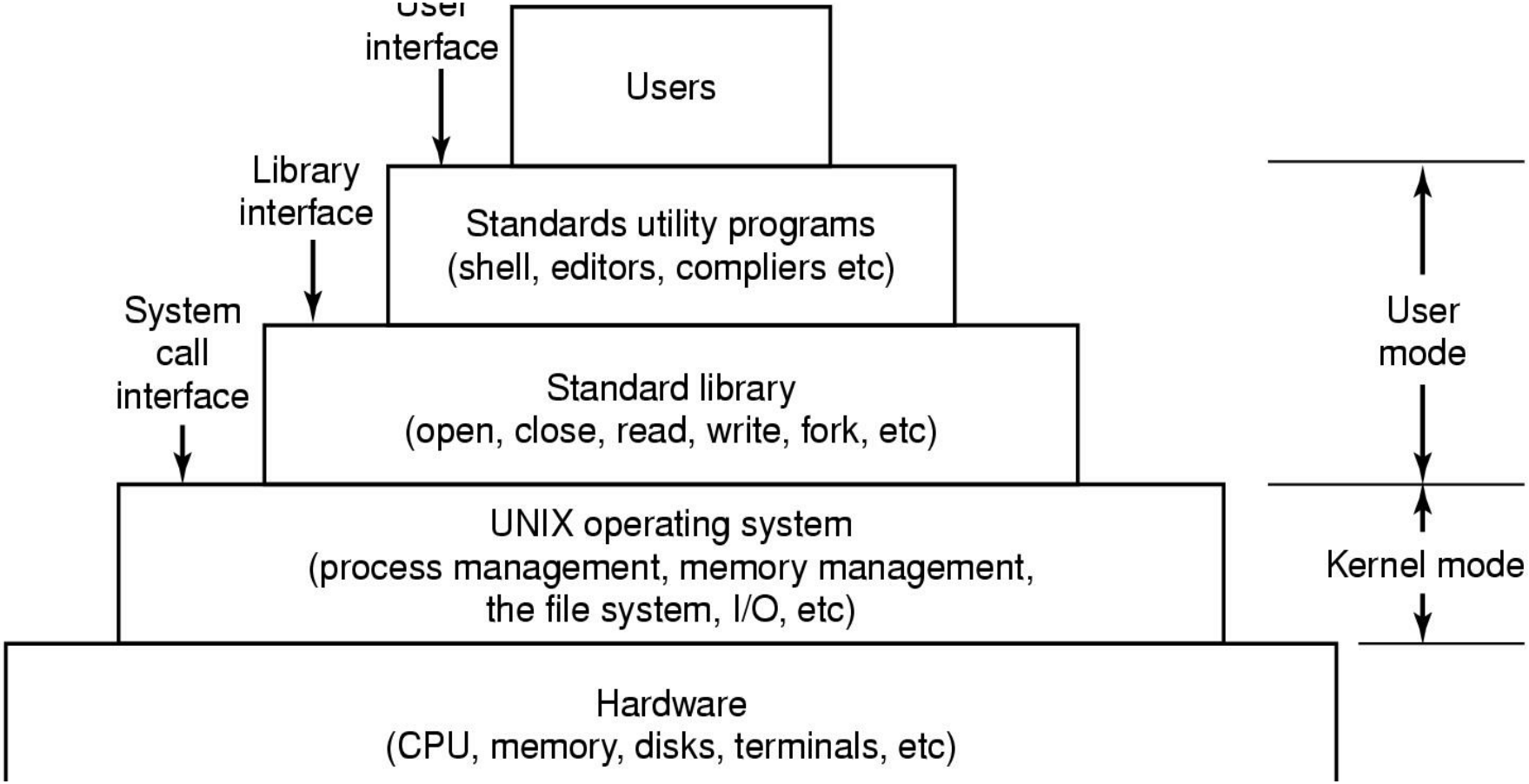
Basic Introduction



A computer system consists of

- hardware
- system programs
- application programs

Unix Layers


User interface
Users
Library interface
Standards utility programs
(shell, editors, compliers etc)
System call interface
Standard library
(open, close, read, write, fork, etc)
UNIX operating system
(process management, memory management,
the file system, I/O, etc)
Hardware
(CPU, memory, disks, terminals, etc)
User mode
Kernel mode

User Space vs Kernel Space

Safety Reason:

- This separation serves to protect data and functionality from faults (by improving fault tolerance) and malicious behaviour (by providing computer security).

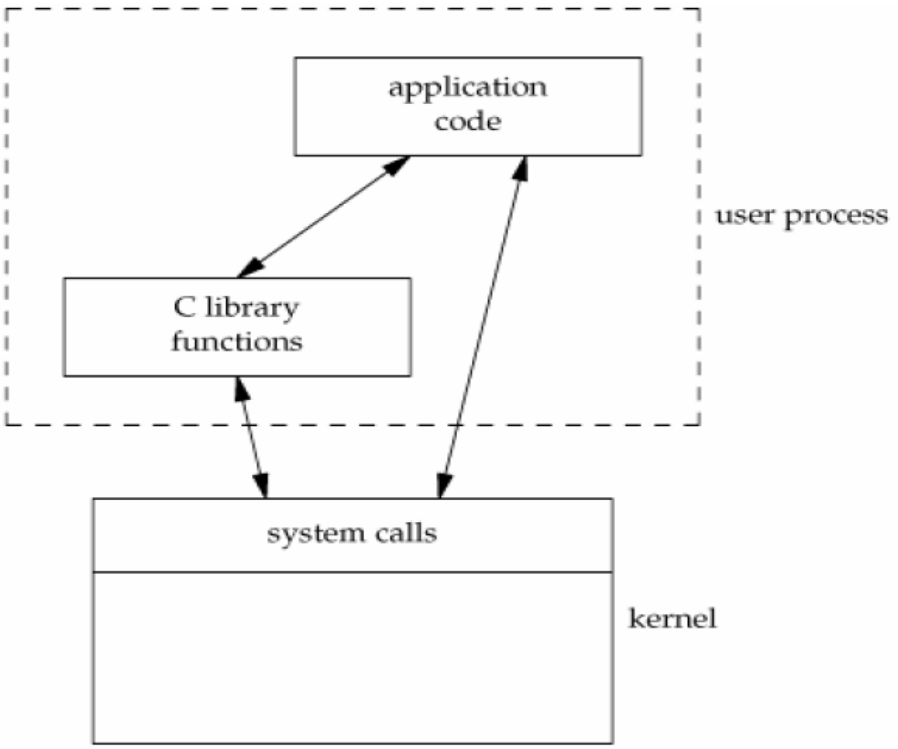
User space:

- Refers to all code which runs outside the operating system's kernel.
- User space usually refers to the various programs and libraries that interacting with the kernel.

Kernel space:

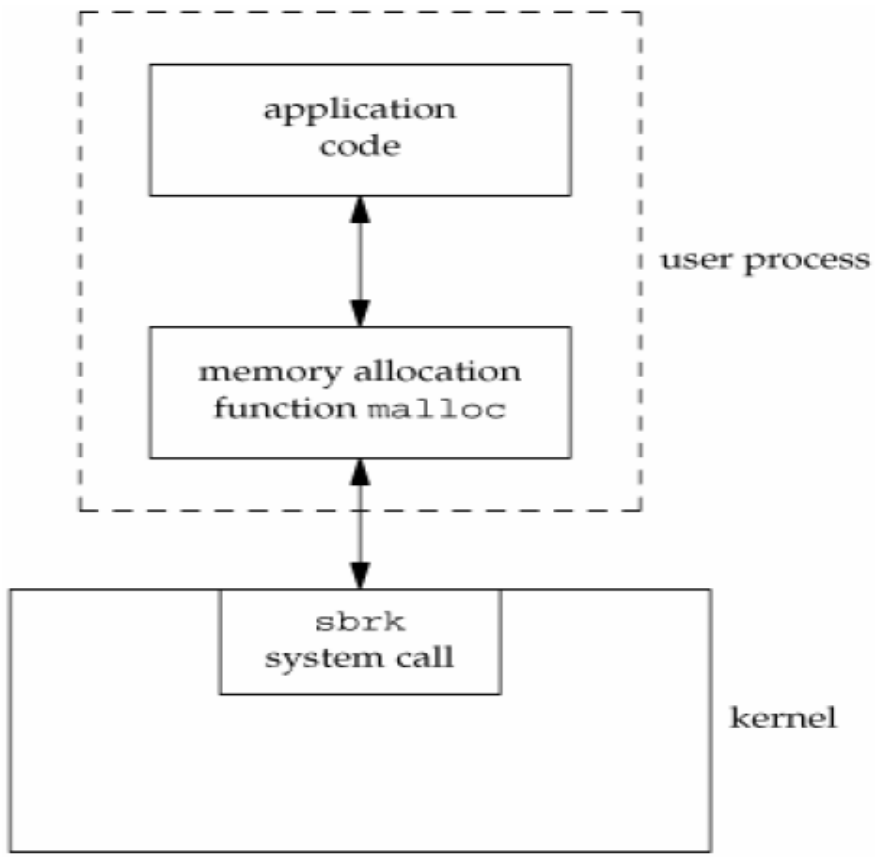
- Reserved for running privileged kernel, kernel extensions, and most device drivers.

System Calls



- A system call is how a program requests a service from an operating system's kernel.
- System calls provide an essential interface between a process and the operating system.

An System Call Example



Utility

Definition:

- Some software tools that are in the system and help analyze, manage, configure, optimize or maintain a computer

Reason:

- DON'T need to write a program to do simple task. For example, we want to know the content of a file.

Check the command of a utility:

- man CMD # Check the manual of a command CMD

Directory

Home directory: A home directory is a directory on a multi-user system, containing files for a given user of the system.

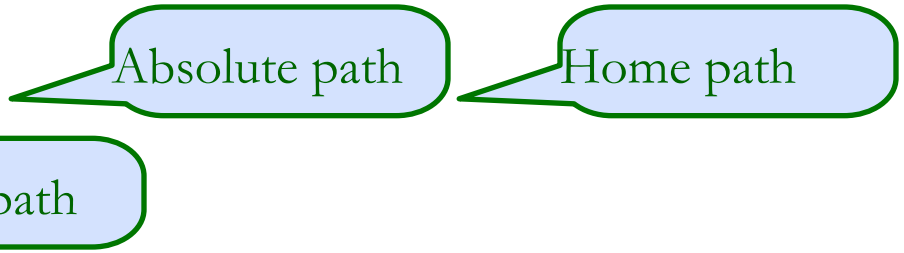
Root directory: the first or top-most directory in a hierarchy(/).

Working directory: the directory associated with a particular process.

Absolute path: An absolute path is defined as the specifying the location of a file or directory from the root directory

Relative path: Relative path is defined as path related to the present working directory(pwd).

- **/home/tongpingliu**
- **../mydir**



Managing files

Commands for basic file management:

- cp <file/dir> <dest> # Copy <file/dir> to a destined file/dir
- cp –r <dir> <dest> # Recursive copy a directory
- mv <source> <dest> # Move a file/directory from <src> to <dest>

- cat <file> # Display the full contents of <file> to the screen

- head <file> # Print out the top of file. "–n X" to print specific lines
- tail <file> # Print out the bottom of file. "–n X" to print specific lines

- clear # clear the screen

Output to a file

"Redirect" output of a command to a file

- Useful for commands that produce many lines of output
- Save results for later, or to use with another command

Syntax: [command] > [filename]

Warning! This will REPLACE any file with the same name

To APPEND to a file, use >>

- [command2] >> [filename]

```
> sort days.txt > sorted.txt

> head -n 3 sorted.txt
Friday
Monday
Saturday
```

Pipes – combining multiple commands

Pipes allow you to combine multiple commands

Syntax: [command 1] | [command 2]

Example:

- **Sort a file, then print the top 3 entries**

```
> sort days.txt | head -n 3
Friday
Monday
Saturday
```

Output from the first command is used as input to the second command

Utilities review

Utilities	Description
`cp, mv, rm`	copy, move, and delete files
`cat`	print files to console
`find`	print lines matching a pattern
`sort`	sort files
`tr`	translate or delete characters
`less, more`	view long files
`grep`	print lines matching a pattern
`>, >>`	Write command output to a file
`|`	Send command output to another command

More utilities: http://en.wikipedia.org/wiki/List_of_Unix_utilities

What we have covered?

Utility

Shell

Awk

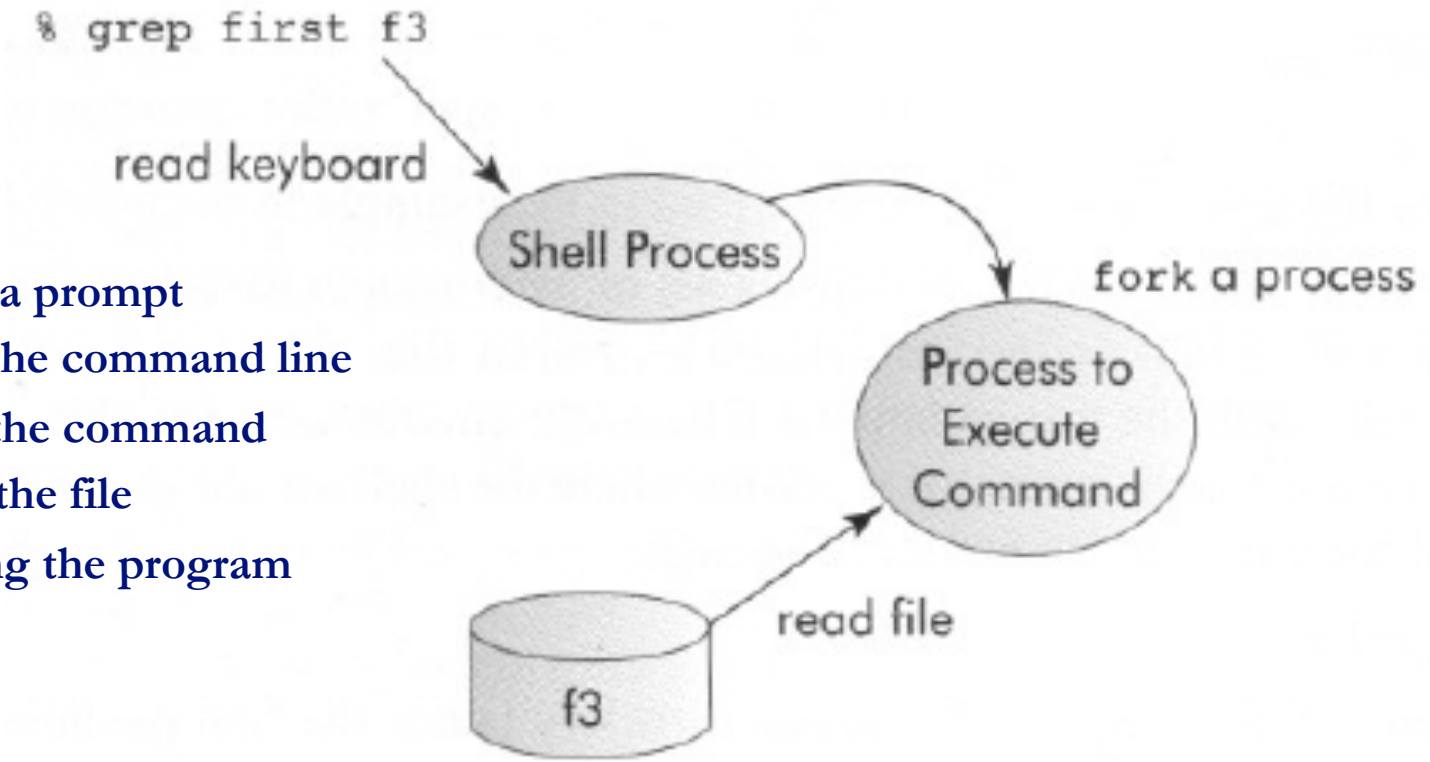
Sed

Perl

C Programming

Unix Shell

The shell isolates itself from program failures by creating a child process to execute each command/program

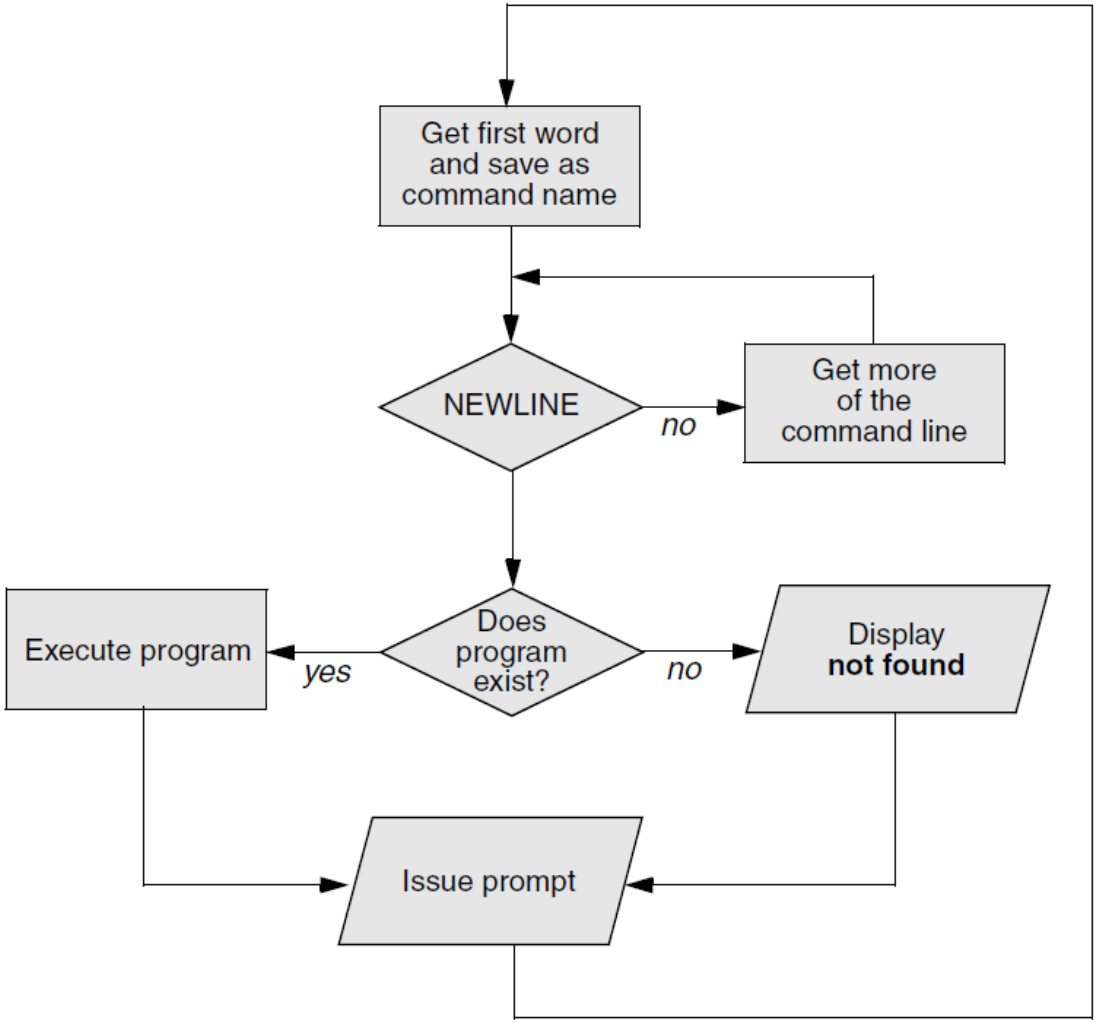


- Printing a prompt
- Getting the command line
- Parsing the command
- Finding the file
- Executing the program

Why creating a child process to execute a command, instead of by itself?

Protect itself from any fatal errors that might arise during execution

Processing the Command Line



Where does the Shell check for the existence of the given command ?

Depends on whether Absolute or Relative Path is provided. (PATH) environment

ls → /bin/ls

./myprogram

Variable Creation and Local Variables

In a directory with "a", "b" and "c" file.

Two commands:

echo ls

echo $(ls)

Result is "ls" since ls is a string here.

Result is "a b c" since we are echoing variables of executing "ls" command

```
#!/bin/bash
HELLO=Hello
function hello {
        local HELLO=World
        echo $HELLO
}
echo $HELLO
hello
echo $HELLO
```

$HELLO is "World" here.
Using LOCAL to mark local variables

$HELLO is "Hello" here.

Input parameters

The number of arguments:

$1: First parameter

$2: Second parameter

$#: the number of input parameters

String Comparison	Description
Str1 = Str2	Returns true if the strings are equal
Str1 != Str2	Returns true if the strings are not equal
-n Str1	Returns true if the string is not null
-z Str1	Returns true if the string is null
Numeric Comparison	**Description**
expr1 -eq expr2	Returns true if the expressions are equal
expr1 -ne expr2	Returns true if the expressions are not equal
expr1 -gt expr2	Returns true if expr1 is greater than expr2
expr1 -ge expr2	Returns true if expr1 is greater than or equal to expr2
expr1 -lt expr2	Returns true if expr1 is less than expr2
expr1 -le expr2	Returns true if expr1 is less than or equal to expr2
! expr1	Negates the result of the expression
File Conditionals	**Description**
-d file	True if the file is a directory
-e file	True if the file exists (note that this is not particularly portable, thus -f is generally used)
-f file	True if the provided string is a file
-g file	True if the group id is set on a file
-r file	True if the file is readable
-s file	True if the file has a non-zero size
-u	True if the user id is set on a file
-w	True if the file is writable
-x	True if the file is an executable

What we have covered?

Utility

Shell

Awk

Sed

Perl

C Programming

Awk Introduction

Awk is a utility for processing structured data

- Anything that has multiple entries in the same fields
- Example:
  - 'Contacts' file where each contact has a name & phone number
  - Awk could be used to print just the phone numbers

Splits a file into fields (columns) and operates on each row (line)

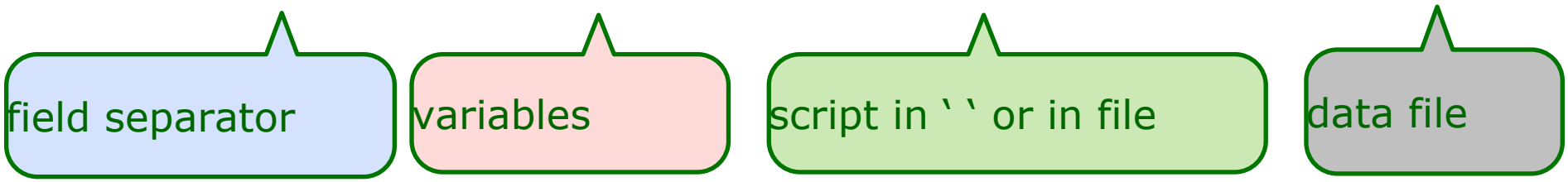
- Useful for processing log files, experimental data, etc

More information:

http://www.gnu.org/software/gawk/manual/gawk.html

awk - processing columns of data

awk [-F fs] [-v var=value] ['prog' | -f progfile] [file ...]



awk -F “,” '{print $2}' data.txt

awk -F “,” -f script.awk data.txt

If the field separator is SPACE or TAB, there is no need to spec

Awk examples

Value of a column = $1, $2, $3 for column 1, 2, 3, etc

To print out the second and fourth column:

data.txt

```
1 10 xyz 100
2 20 abc 200
3 30 def 300
4 40 ade 400
5 50 f2d 500
```

```
awk   '{print $2 "," $4}'   data.txt
```

single quotes around program

double quotes for text

Output:

```
10, 100
20, 200
30, 300
40, 400
50, 500
```

Variables

User variables

- string, numeric

Program variables

Variable	Meaning
$0	Current record
$1-$n	Fields in the current record
FILENAME	Current input file name (null for standard input)
FS	Input field separator (default: SPACE or TAB)
NF	Number of fields in the current record
NR	Record number of the current record
OFS	Output field separator (default: SPACE)
ORS	Output record separator (default: NEWLINE)
RS	Input record separator (default: NEWLINE)

What we have covered?

Utility

Shell

Awk

Sed and Regular expression

Perl

C Programming

Regular Expression

- A regular expression (regex or re for short) is a special text string for describing a search pattern.
- Some utilities/programs that use them:
  - –vi, ed, sed, and emacs
  - –awk, Perl and Python
  - –grep, egrep, fgrep
  - –compilers

Definition

- R is a regular expression if it is:
  1. a for some *a* in the alphabet $\sum$, standing for the language {a}
  2. ε, standing for the language {ε}
  3. Ø, standing for the empty language
  4. R_1+R_2 where R_1 and R_2 are regular expressions, and + signifies union (sometimes | is used)
  5. R_1R_2 where R_1 and R_2 are regular expressions and this signifies concatenation
  6. R* where R is a regular expression and signifies closure
  7. (R) where R is a regular expression, then a parenthesized R is also a regular expression

Regular Expression Quick Guide

```
^        Matches the beginning of a line
$        Matches the end of the line
.        Matches any character
*        Repeats a character 0 or more times
*?       Repeats a character 0 or more times (non-greedy)
+        Repeats a character 1 or more times
+?       Repeats a character 1 or more times (non-greedy)
[aeiou]  Matches a single character in the listed set
[^XYZ]   Matches a single character not in the listed set
[a-z0-9] The set of characters can include a range
(        Indicates where string extraction is to start
)        Indicates where string extraction is to end
```

Introduction

- sed: **s**tream **ed**itor
- Used for editing files automatically.
- Non-interactive editor: won't modify the file, all the output is just printed out.

```
sed [options] 'command' file(s)

sed [options] -f scriptfile file(s)
```

Deleting lines

```
$ cat a.txt
line 1
line 2
line 3
line 4
line 5
line 6
```

Deleting the third line:

```
$ sed '3d' a.txt
line 1
line 2
line 4
line 5
line 6
```

Deleting all the lines with ‘2’ in it:

```
$ sed '/2/'d a.txt
line 1
line 3
line 4
line 5
line 6
```

Deleting from the third line to the end:

```
$ sed '3,$d' a.txt
line 1
line 2
```

Deleting the last line:

```
$ sed '$d' a.txt
line 1
line 2
line 3
line 4
line 5
```

’s’ command

```
$ cat test.txt
#!/bin/bash
 function hello{
 echo "hello";
}
hello;
$ sed 's/hello/hi/g' test.txt
#!/bin/bash
 function hi{
 echo "hi";
}
hi;
$ cat test.txt
#!/bin/bash
 function hello{
 echo "hello";
}
hello;
```

```
$ sed -n 's/hello/hi/p' test.txt
 function hi{
 echo "hi";
hi;
```

```
$ sed -n 's/hello/&hi/p' test.txt
 function hellohi{
 echo "hellohi";
hellohi;
```

```
$ sed -n 's/\(he\)llo/\1lp/p' test.txt
 function help{
 echo "help";
help;
```

sed script

- # is followed by the comments. e.g., #!/bin/sed -f

```
$ cat lines
Line one.
The second line.
The third.
This is line four.
Five.
This is the sixth sentence.
This is line seven.
Eighth and last.
```

```
$ cat subs_demo
s/line/sentence/p
```

```
$ sed -n -f subs_demo lines
The second sentence.
This is sentence four.
This is sentence seven.
```

sed -n 's/line/sentence/p' lines

Use sed in a shell

script a.txt:

```
#!/bin/bash
echo -n 'what is the value? '
read value
sed  's/XXX/'$value'/' <<EOF
The value is XXX
EOF
```

```
$ ./a.txt
what is the value? 1234
The value is 1234
```

What we have covered?

Utility

Shell

Awk

Sed and Regular expression

Perl

C Programming

Three basic types

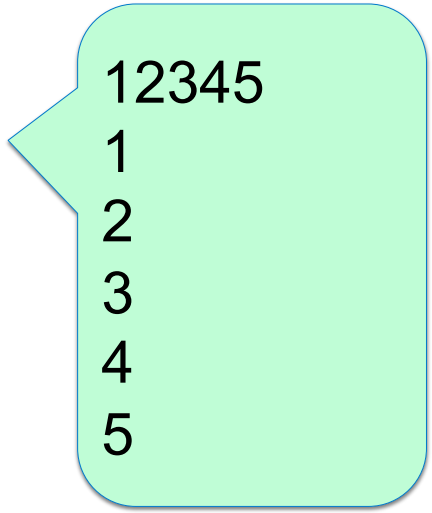
Variable Type	Description
Scalars ($)	Holds one number or string value at a time.
Arrays (@)	Holds a list of values. The values can be numbers, strings, or even another array. Array variable names always begin with a @.
Associative Arrays (%)	Uses any value as an index into an array.

Different beginning characters also provide a different namespace for each variable type.

Namespaces separate one set of names from another.

Access Array Elements

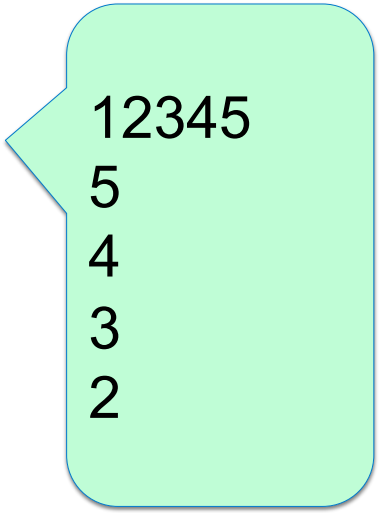
```
@array = (1..5);
print @array;     print "\n";
print $array[0];  print "\n";
print $array[1];  print "\n";
print $array[2];  print "\n";
print $array[3];  print "\n";
print $array[4];  print "\n";
```



A Perl program uses zero as the base array subscript.

Negative Subscript

```
@array = (1..5);
print @array;      print "\n";
print $array[-1];  print "\n";
print $array[-2];  print "\n";
print $array[-3];  print "\n";
print $array[-4;  print "\n”;
```



Use a negative script to get the value of the last elements.
-1 is the last one

Hash initialization

```
%assArray = ("Jack A.", "Dec 2", "Joe B.", "June 2", "Jane C.",
"Feb 13");

my %hash = ( foo => 42, bar => 43, baz => 44 );

my %hash = ( 'foo', 42, 'bar', 43, 'baz', 44 );
```

Creating hashes

```
my %wordhash;

$wordhash{'class'} = 3;

$wordhash{'course'} = 2;
```

```
$my %wordhash = ('class', 3, 'course', 2);

    OR

$my %wordhash = (`class' => 3, `course' => 2);
```

Accessing hash elements

```
my %wordhash = ('class', 3, 'course', 2);

my $count1 = $wordhash{'class'};

print $count1;   #will print 3
```

```
my %wordhash = ('class', 3, 'course', 2);

my $count1 = $wordhash{'class'};

my @array = @wordhash{'class', `course'};  #@array is (3, 2)
```

List

A list is a sequence of scalar values enclosed in parentheses. For example:

(1, 5.3, "hello", 2)

This list contains four elements, each of which is a scalar value: the numbers 1 and 5.3, the string hello, and the number 2.

Empty list: A list can have no elements
()

foreach

- go through each element in a structure:

```
foreach $item (@food)          # Visit each item in turn
{
    print "$item\n"; # Print the item
}
```

for

format:

```
for (initialise; test; inc)
{
        first_action;
        second_action;
        etc
}
```

```
for ($i = 0; $i < 10; ++$i)   # Start with $i = 1
{
     print "$i\n";
}
```

while

```
#!/usr/local/bin/perl
print “Password?\n ";          # Ask for input
$a = <STDIN>;                  # Get input
chop $a;                       # Remove the newline at end
while ($a ne "right")      # While input is wrong...
{
    print "sorry. Again? ";    # Ask again
    $a = <STDIN>;                # Get input again
    chop $a;                   # Chop off newline again
}
```

=~ s/…/…/(substitution)

- $sentence =~ s/london/London/

- Get patterns that have been matched:$1,…,$9.

```
$_ = "Today is Tuesday";
s/([A-Z])/:\1:/g;
print "$_\n";
```

:T:oday is :T:uesday

DEMO:1sub.pl

=~ tr/…/…/(translation)

tr: allows character-by-character translation.

```
$a =~ tr/abc/def/;

$count = ($a =~ tr/*/*/);

tr/a-z/A-Z/;
```

DEMO:2trans.pl

Subroutine

```
sub mysubroutine
{
        print "this is a routine\n";
}


&mysubroutine;              # Call the subroutine
&mysubroutine('a');         # Call it with a parameter
&mysubroutine(1, 2);        # Call it with two parameters
...
```

& is used to call a subroutine

Using “While” to traverse every line

```
#!/usr/bin/perl

$file='./mytest.txt';
open(FD, $file) or die "Can't open the file: $!";

while ($line = <FD>) {
    print $line;
}

close(FD);
```

DEMO:5open.pl

Using “While” to traverse every line (2)

```
#!/usr/bin/perl

$file='./mytest.txt';
open(FD, $file) or die "Can't open the file: $!";
@alllines=<FD>;
close(FD);
while (defined ($line = shift $alllines>)) {
    print $line;
}

close(FD);
```

shift() function is used to remove and return the first element from an array. Destroy the original array

DEMO:7while.pl

Using “While” to traverse every line (3)

```
#!/usr/bin/perl

$file='./mytest.txt';
open(FD, $file) or die "Can't open the file: $!";
@alllines=<FD>;
close(FD);
while ($alllines) {
        my $line
    print $line;
}

close(FD);
```

shift() function is used to remove and return the first element from an array. Destroy the original array

DEMO:10while.pl

What we have covered?

Utility

Shell

Awk

Sed and Regular expression

Perl

C Programming

Data types in C

Only really four basic types:

- char
- int (short, long, long long, unsigned)
- float
- double

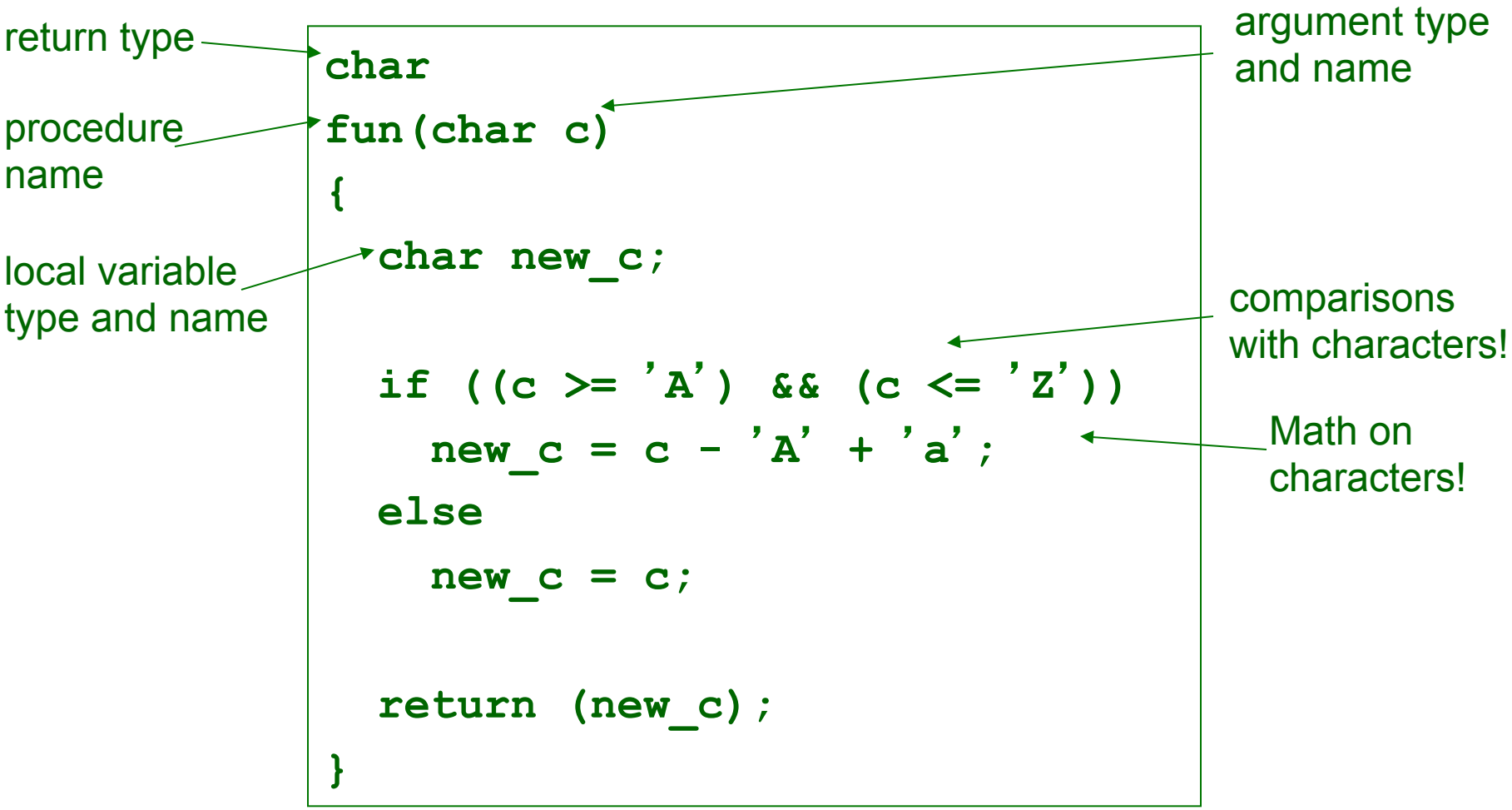
Size of these types on 64bit machines:

Type	Size (bytes)
char	1
int	4
short	2
long	8
long long	8
float	4
double	8

Sizes of these types vary from one machine to another!

Small Hello-World Program about Characters

What does this function do?



Booleans in C

```
#include <stdbool.h>

bool  bool1  = true;
bool  bool2  = false;
```

Important!
Compiler needs this or it won't know about "bool"!

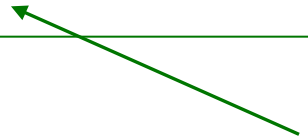
bool added to C in 1999

Many programmers had already defined their own Boolean type

- **To avoid conflict bool is disabled by default**

Enumerated Types in C

```
enum Color {RED, GREEN, BLACK, YELLOW};
enum Color  my_color = RED;
```



The new type name is
"**enum Color**"

Alternative style:

```
enum AColor {COLOR_RED, COLOR_WHITE,
             COLOR_BLACK, COLOR_YELLOW};
typedef  enum AColor  color_t;
color_t  my_color = COLOR_RED;
```

Structures

Compound data:

A date is

- an **`int month`** and
- an **`int day`** and
- an **`int year`**

```
struct ADate {
   int  month;
   int  day;
   int  year;
};

struct ADate date;

date.month = 9;
date.day = 1;
date.year = 2005;
```

Structure Representation & Size

`sizeof(struct …)` =

sum of **`sizeof`**(field)

\+ alignment padding

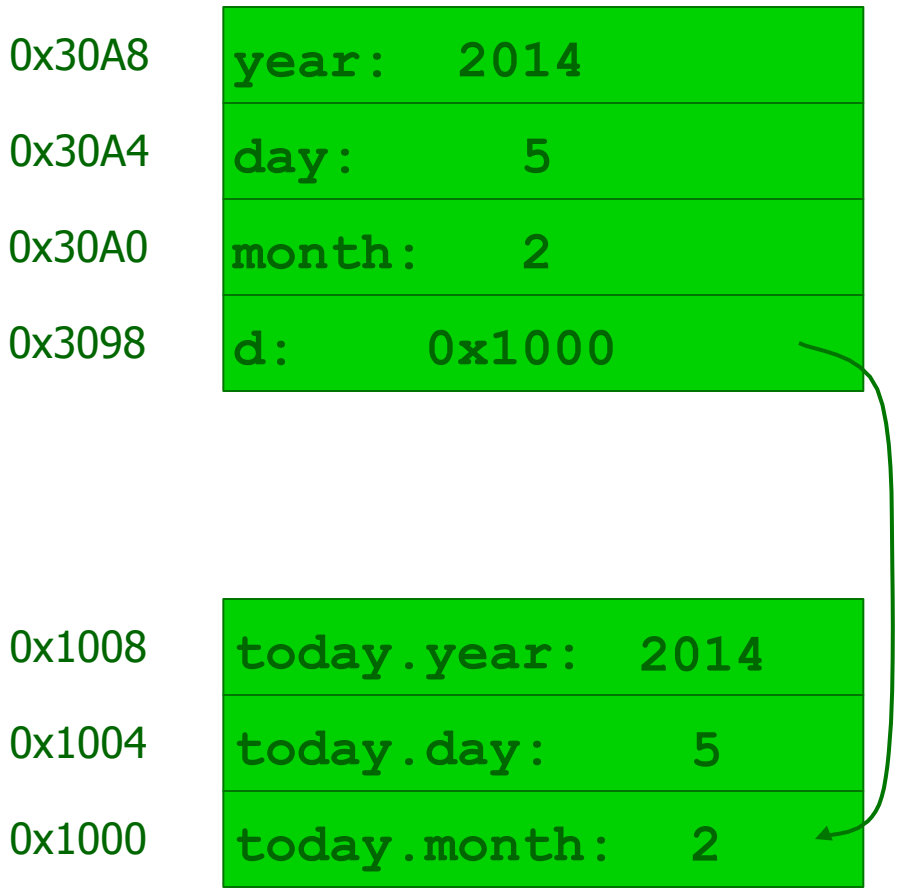
Processor- and compiler-specific

```
struct CharCharInt {
   char  c1;
   char  c2;
   int   i;
} foo;

foo.c1 = 'a';
foo.c2 = 'b';
foo.i  = 0xDEADBEEF;
```

x86 uses "little-endian" representation

Pointers to Structures (cont.)

```
void
create_date2(Date *d,
             int month,
             int day,
             int year)
{
  d->month = month;
  d->day   = day;
  d->year  = year;
}

void
fun_with_dates(void)
{
  Date today;
  create_date2(&today, 2, 5, 2014);
}
```



Unions

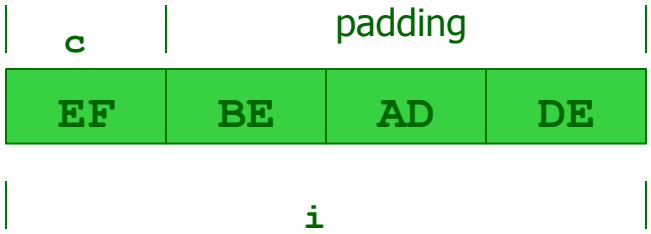
Choices:

An element is

- an `int i` or
- a `char c`

`sizeof(union …)` =
maximum of `sizeof`(field)

```
union AnElt {
   int   i;
   char  c;
} elt1, elt2;

elt1.i = 4;
elt2.c = 'a';
elt2.i = 0xDEADBEEF;
```



Interpreters & Compilers

Interpreter

- **A program that reads a source program and produces the results of executing that program**

Compiler

- **A program that translates a program from one language (the *source*) to another (the *target*)**

Compiler

Read and analyze entire program

Translate to semantically equivalent program in another language

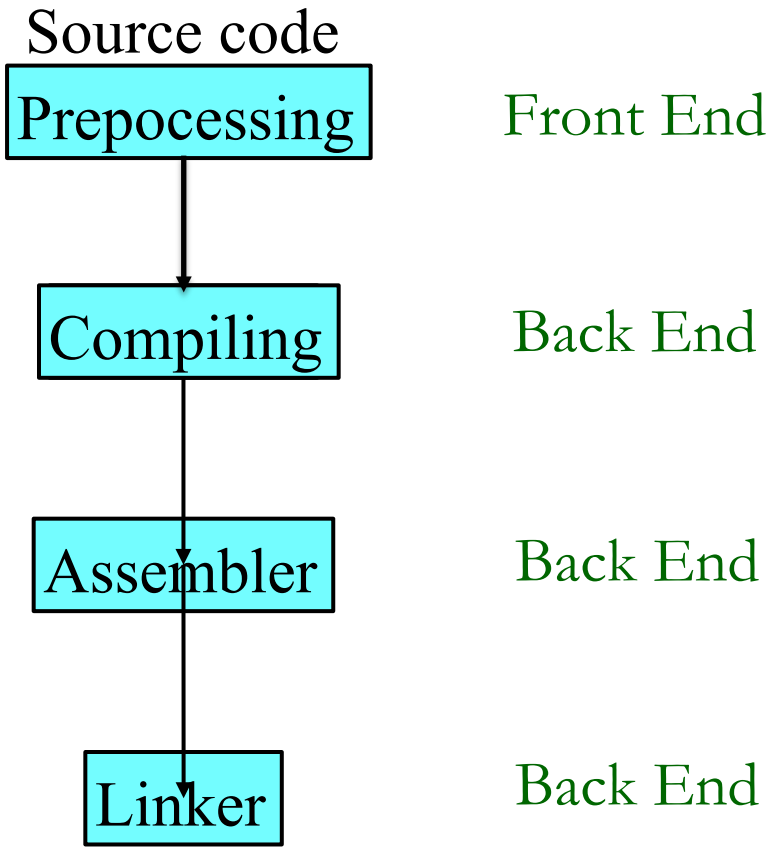
- Presumably easier to execute or more efficient
- Should “improve” the program in some fashion

Offline process

- Tradeoff: compile time overhead (preprocessing step) vs execution performance

Compilers: The Big picture

- **Front end: Read source program and understand its structure and meaning**
- **Back end: Generate equivalent target language program**



What does a linker do?

Merges object files

- merges multiple ***relocatable*** (.o) object files into a single ***executable*** object file that can loaded and executed by the loader.

Resolves external references

- as part of the merging process, resolves ***external references.***
  - *external reference*: reference to a symbol defined in another object file.

Relocates symbols

- relocates ***symbols*** from their relative locations in the .o files to new absolute positions in the executable.
- updates all references to these symbols to reflect their new positions.
  - references can be in either code or data
    - **code:** `a();  /* ref to symbol a */`
    - **data:** `int *xp=&x;  /* ref to symbol x */`
  - because of this modifying, linking is sometimes called *link editing.*

Executable and linkable format (ELF)

Standard binary format for object files

Derives from AT&T System V Unix

- later adopted by BSD Unix variants and Linux

One unified format for relocatable object files (.o), executable object files, and shared object files (.so)

- generic name: ELF binaries

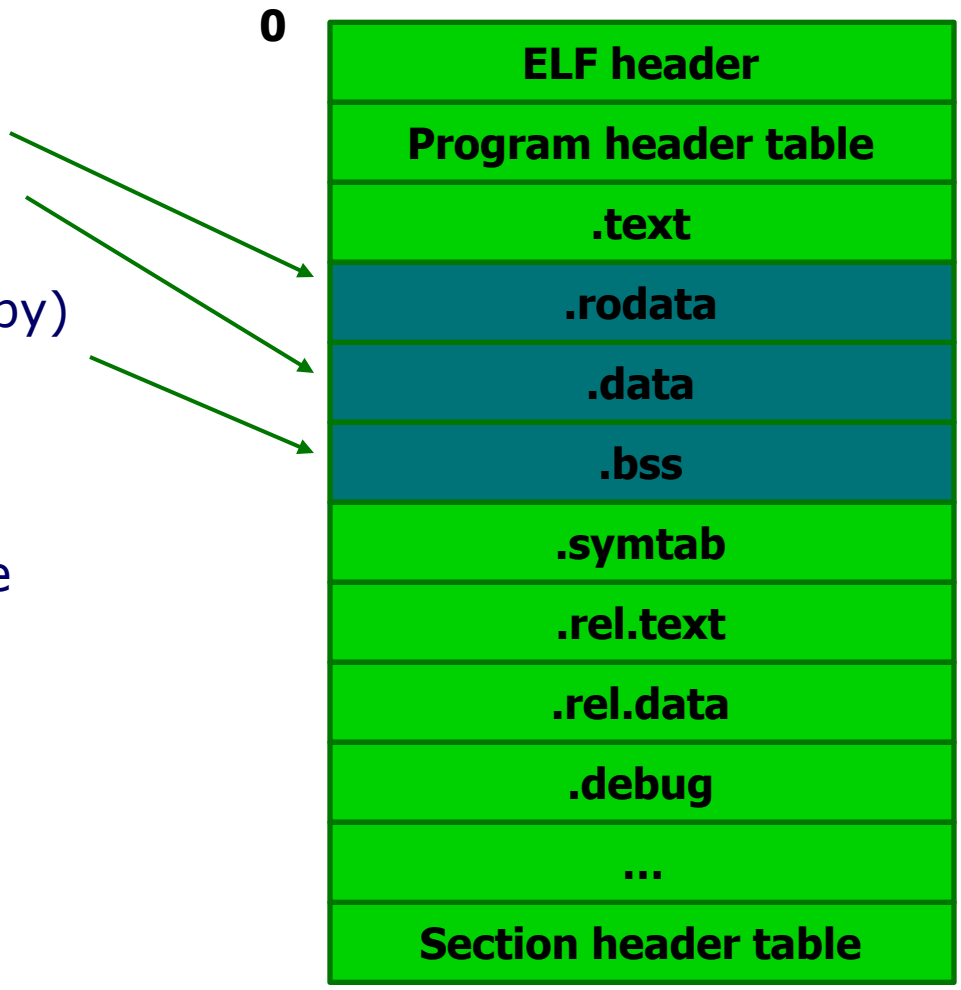
Better support for shared libraries than old a.out formats.

Data Sections

Static data
- initialized, read-only
- initialized, read/write
- uninitialized, read/write
- (BSS = "Block Started by)

Initialized
- Initial values in ELF file

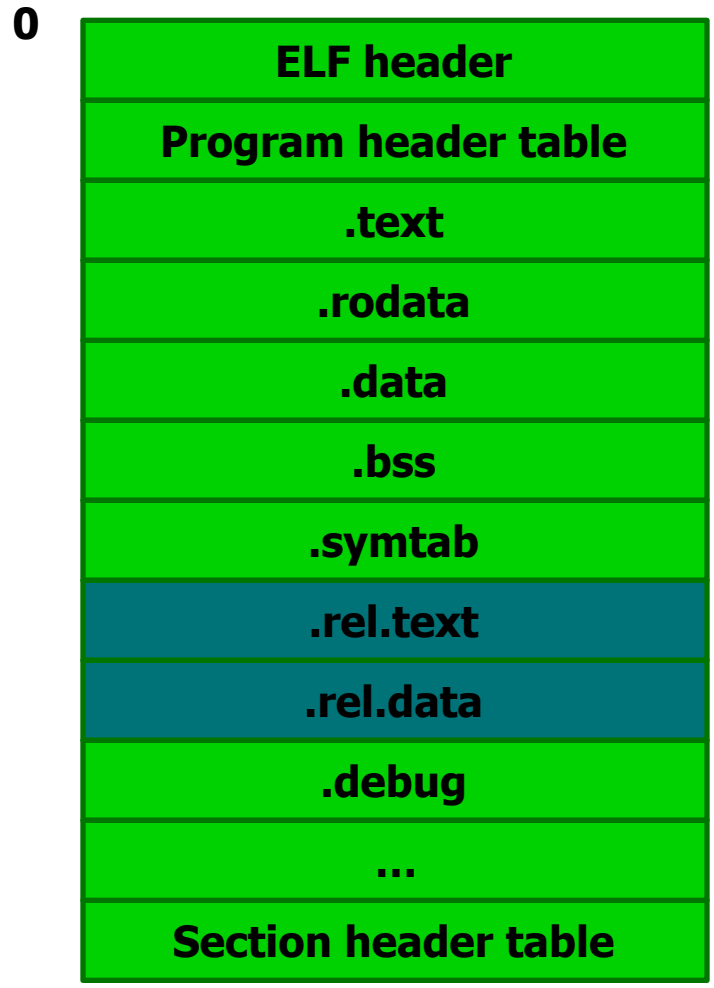
Uninitialized
- Only total size in ELF file



Relocation Information

Describes where and how symbols are used

- A list of locations in the .text section that will need to be modified when the linker combines this object file with others
- Relocation information for any global variables that are referenced or defined by the module
- Allows object files to be easily relocated



Linker Symbol Classification

Global symbols

- Symbols defined by module ***m*** that can be referenced by other modules
- C: non-`static` functions & global variables

External symbols

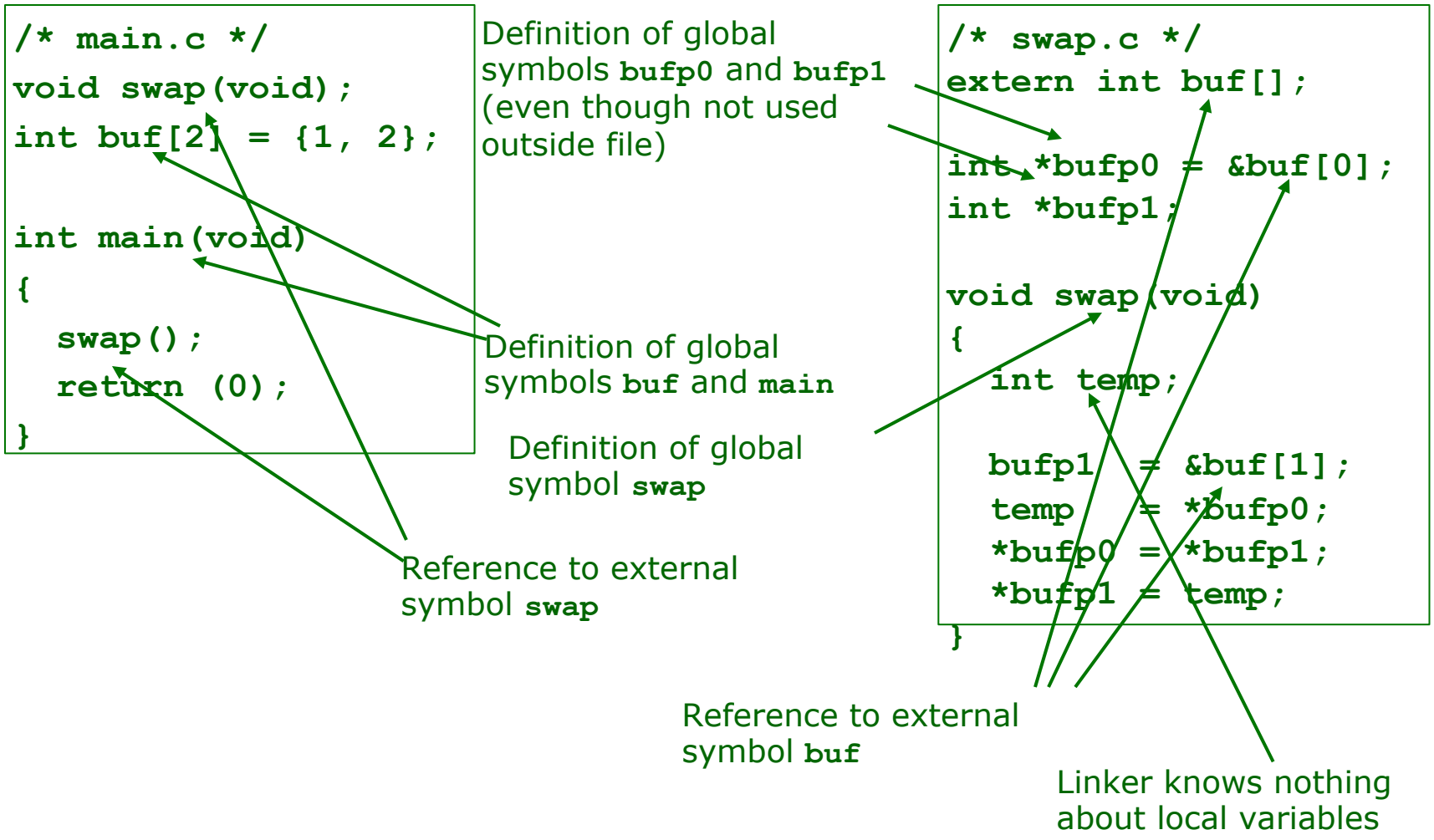
- Symbols referenced by module ***m*** but defined by some other module
- C: `extern` functions & variables

Local symbols

- Symbols that are defined and referenced exclusively by module ***m***
- C: `static` functions & variables

Local linker symbols ≠ local function variables!

Linker Symbols



Linker Symbols

```
/* main.c */
void swap(void);
int buf[2] = {1, 2};

int main(void)
{
  swap();
  return (0);
}
```

What's missing?

- swap – where is it?

main is a 19-byte function located at offset 0 of section 1 (.text)

swap is referenced in this file, but is undefined (UND)

buf is an 8-byte object located at offset 0 of section 3 (.data)

use **`readelf -S`** to see sections

```
UNIX% gcc -O -c main.c
UNIX% readelf -s main.o

Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
…
     8: 0000000000000000    19 FUNC    GLOBAL DEFAULT    1 main
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND swap
    10: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 buf
```

Linker Symbols

```
/* swap.c */
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

void swap(void)
{
  int temp;

  bufp1  = &buf[1];
  temp   = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

bufp0 is an 8-byte object located at offset 0 of section 3 (.data)

swap is a 38-byte function located at offset 0 of section 1 (.text)

buf is referenced in this file, but is undefined (UND)

bufp1 is an 8-byte uninitialized (COMMON) object with an 8-byte alignment requirement

```
Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     8: 0000000000000000    38 FUNC    GLOBAL DEFAULT    1 swap
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND buf
    10: 0000000000000008     8 OBJECT  GLOBAL DEFAULT  COM bufp1
    11: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 bufp0
```

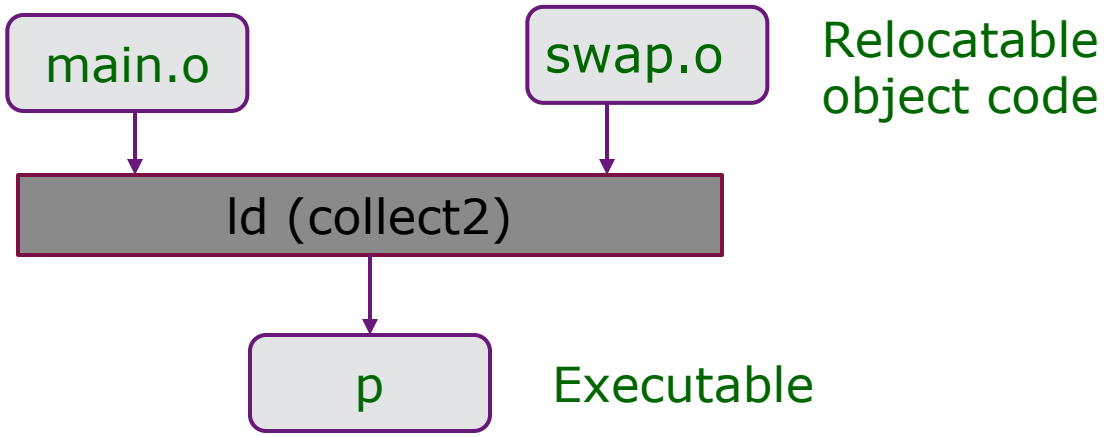
Linking Steps

Symbol Resolution

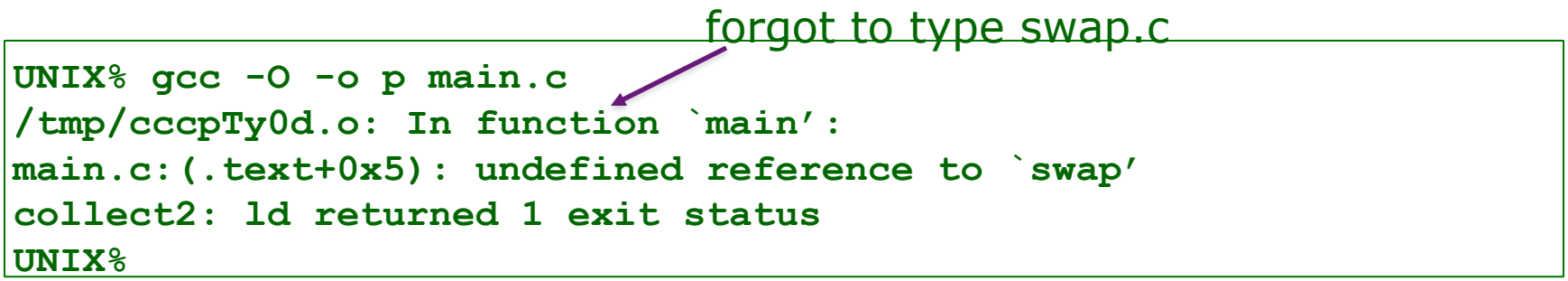
- Determine where symbols are located and what size data/ code they refer to

Relocation

- Combine modules, relocate code/data, and fix symbol references based on new locations



Problem: Undefined Symbols



Missing symbols are not compiler errors

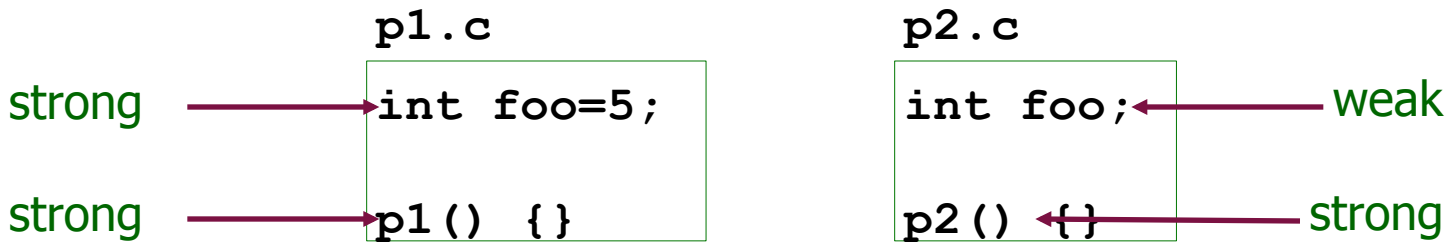
- May be defined in another file
- Compiler just inserts an undefined entry in the symbol table

During linking, any undefined symbols that cannot be resolved cause an error

Strong & Weak Symbols

Program symbols are either strong or weak

strong	procedures & initialized globals
weak	uninitialized globals



Strong & Weak Symbols

A strong symbol can only appear once

A weak symbol can be overridden by a strong symbol of the same name

- References to the weak symbol resolve to the strong symbol

If there are multiple weak symbols, the linker can pick an arbitrary one

Linker Puzzles: What Happens?

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols **p1**

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** might overwrite **y**! Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to **x** in **p2** will overwrite **y**! Nasty!

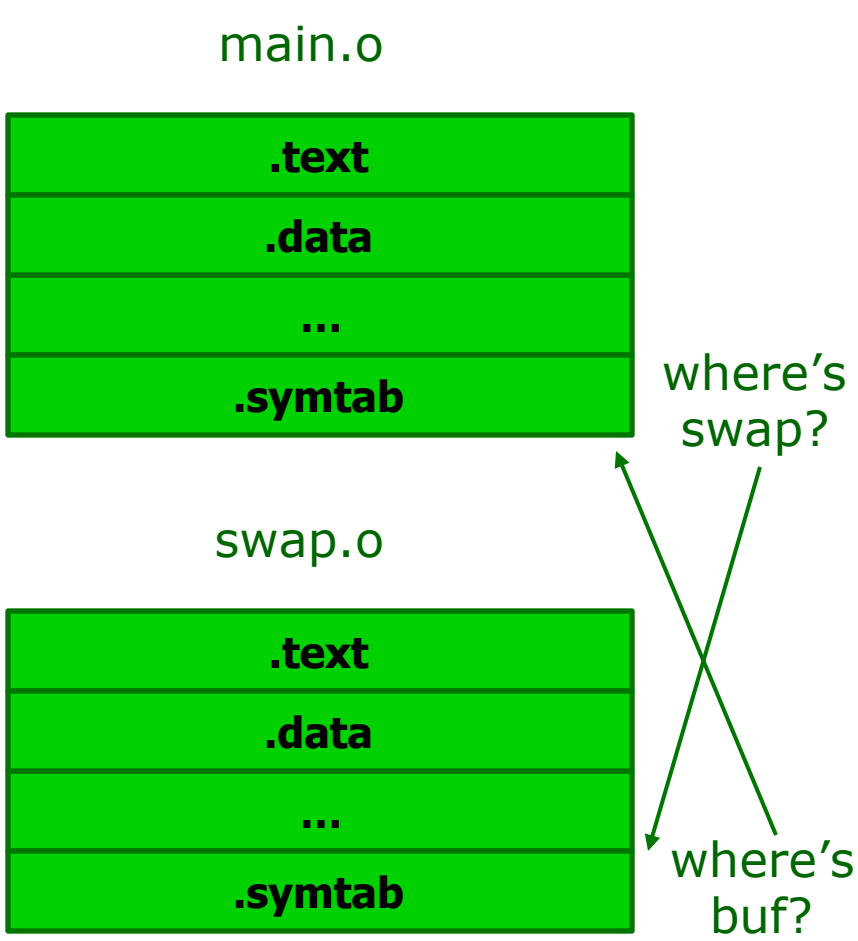
```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to **x** will refer to the same initialized variable

Nightmare scenario: replace r.h.s. **int** with a **struct** type, each file then compiled with different alignment rules

Symbol Resolution



Undefined symbols must be resolved

- Where are they located
- What size are they?

Linker looks in the symbol tables of all relocatable object files

- Assuming every unknown symbol is defined once and only once, this works well

Relocation

Once all symbols are resolved, must combine the input files

- Total code size is known
- Total data size is known
- All symbols must be assigned run-time addresses

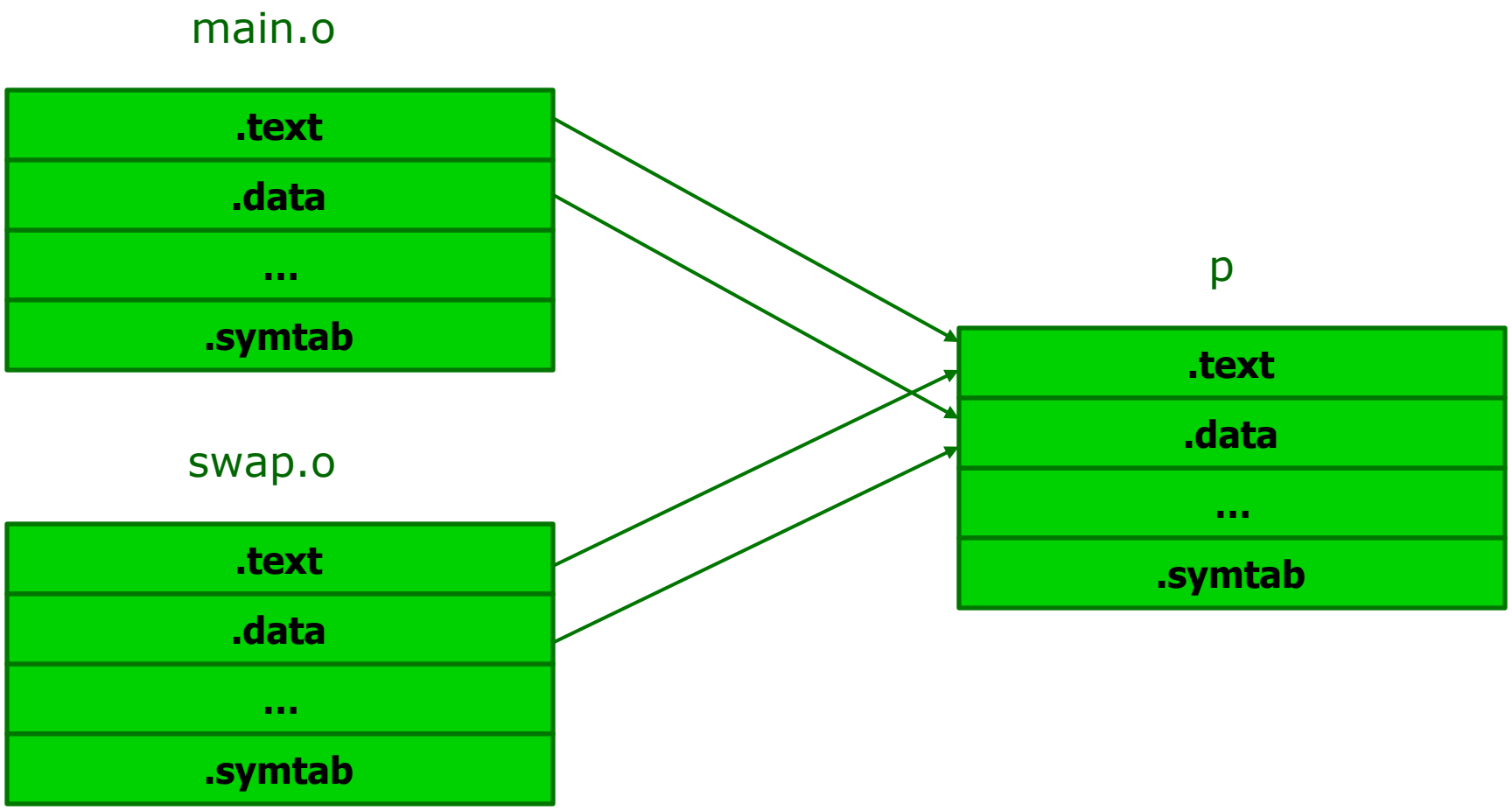
Sections must be merged

- Only one text, data, etc. section in final executable
- Final run-time addresses of all symbols are defined

Symbol references must be corrected

- All symbol references must now refer to their actual locations

Relocation: Merging Files



Static Libraries (.a archive files)

- concatenate related relocatable object files into a single file with an index (called an archive).
- enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
- If an archive member file resolves reference, link into executable.

Using static libraries

Linker's algorithm for resolving external references:

- Scan .o files and .a files in the command line order.
- During the scan, keep a list of the current unresolved references.
- As each new .o or .a file ***obj*** is encountered, try to resolve each unresolved reference in the list against the symbols in ***obj***.
- If any entries in the unresolved list at end of scan, then error.

Problem:

- command line order matters!
- Moral: put libraries at the end of the command line.

```
UNIX% gcc -O -c main.c
UNIX% gcc -static -o program main.o ./libmine.a
```

Shared libraries

Static libraries have the following disadvantages:

- Duplicating lots of common code in the executable files on a filesystem.
  - e.g., every C program needs the standard C library
- Duplicating lots of code in the virtual memory space of many processes.
- Minor bug fixes of system libraries require each application to explicitly relink

Solution:

- ***shared libraries*** (dynamic link libraries, DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.
  - dynamic linking can occur when executable is first loaded and run.
    - common case for Linux, handled automatically by ld-linux.so.
  - dynamic linking can also occur after program has begun.
    - in Linux, this is done explicitly by user with dlopen().
  - shared library routines can be shared by multiple processes.

Advantages of dynamic linking

The executable is smaller (it not include the library information explicitly)

When the library is changed, the code that references it does not usually need to be recompiled.

The executable accesses the .DLL at run time; therefore, multiple codes can access the same .DLL at the same time (saves memory)

Shortcomings of dynamic linking

Performance degrade about ~10%

- Need to load shared objects (once)
- Need to resolve addresses (once or every time)

What if the necessary dynamic library is missing?

Could have the library, but wrong version

Dynamic Libraries

Static	Dynamic
Linked at compile-time	Linked at run-time
UNIX: foo.a	UNIX: foo.so
Relocatable ELF File	Shared ELF File

What are the differences?

Static & Dynamic Libraries

Static

- **Library code added to executable file**
- **Larger executables**
- **Must recompile to use newer libraries**
- **Executable is self-contained**
- **Some time to load libraries at compile-time**
- **Library code shared only among copies of same program**

Dynamic

- Library code not added to executable file
- Smaller executables
- Uses newest (or smallest, fastest, …) library without recompiling
- Depends on libraries at run-time
- Some time to load libraries at run-time
- Library code shared among all uses of library

Static & Dynamic Libraries

Static

Creation

```
ar rcs libfoo.a bar.o baz.o
ranlib libfoo.a
```

Use

```
gcc -o zap zap.o -lfoo
```

Adds library's code, data, symbol table, relocation info, …

Dynamic

Creation

```
gcc -shared -fPIC
 -o libfoo.so bar.o baz.o
```

Use

```
gcc -o zap zap.o -lfoo
```

Adds library's symbol table, relocation info

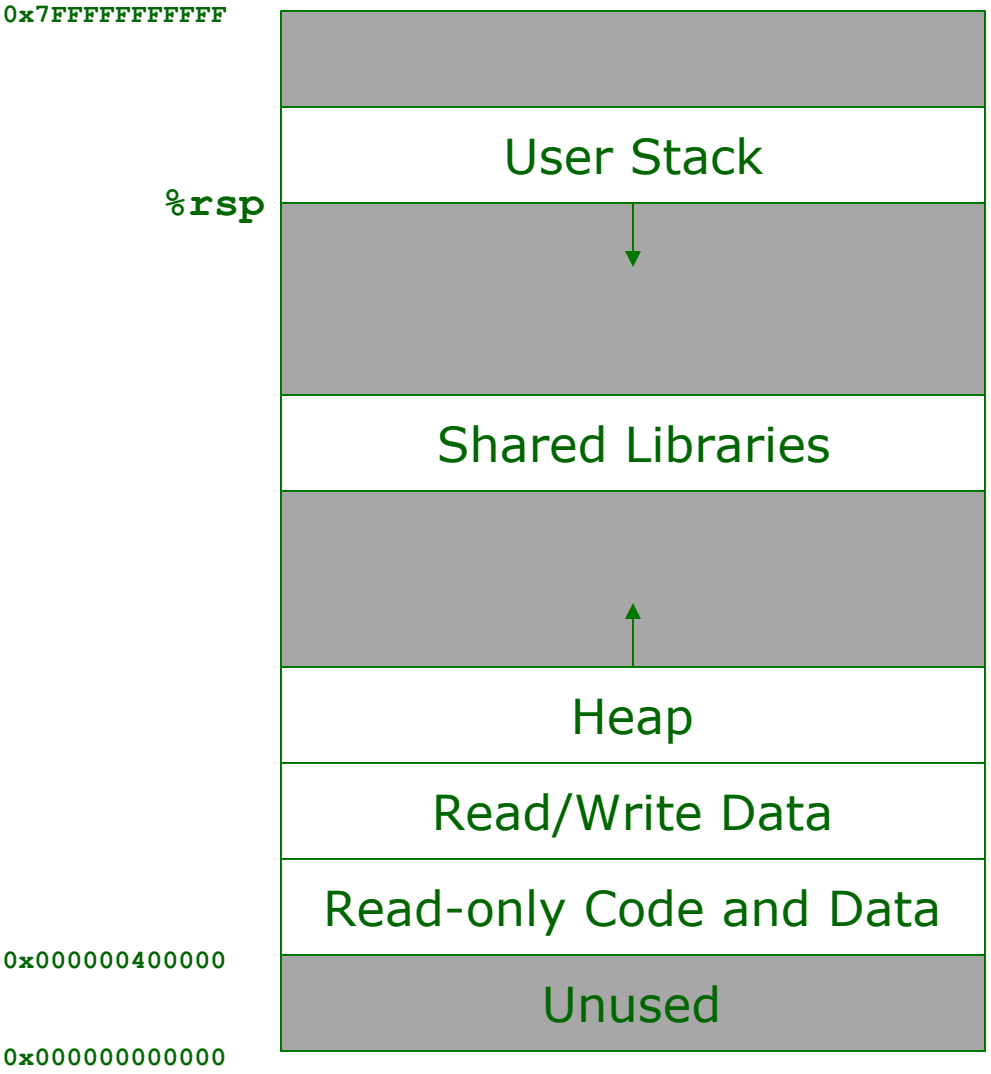
Loading

Linking yields an executable that can actually be run

Running a program

- `unix% ./program`
- **Shell does not recognize "`program`" as a shell command, so assumes it is an executable**
- **Invokes the *loader* to load the executable into memory (any unix program can invoke the loader with the `execve` function – more later)**

Creating the Memory Image (sort of...)



Create code and data segments

- **Copy code and data from executable into these segments**

Create initial heap segment

- **Grows up from read/ write data**

Create stack

- **Starts near the top and grows downward**

Call dynamic linker to load shared libraries and relocate references

Example of different segments

```
int array[10];
```

Globals (.bss segment)

```
int getMax(int a, int b) {
  return (a > b ? a : b);
}
```

a,b at stack

```
int main() {
  int *ptr;
  int maximum;
  maximum = array[0];
  ptr = &array[1];
  for( ; ptr < &array[ARRAY_SIZE]; ptr++)
  {
    maximum = getMax(maximum, *ptr);
  }
}
```

ptr, maximum at stack

ptr pointing to the globals

Demo: test.c
/proc/PID/maps

Examining /proc/PID/maps file

```
tongpingliu@elk02:~$ ps -a
  PID TTY          TIME CMD
23975 pts/4    00:00:00 sftp
23976 pts/4    00:00:00 ssh
25772 pts/0    00:03:15 test
25952 pts/2    00:00:00 ps
29478 pts/3    00:00:00 gdb
tongpingliu@elk02:~$ cat /proc/25772/maps
08048000-08049000 r-xp 00000000 00:1a 42732827   /home/tongpingliu/demo/linkingandloader/test
08049000-0804a000 r--p 00000000 00:1a 42732827   /home/tongpingliu/demo/linkingandloader/test
0804a000-0804b000 rw-p 00001000 00:1a 42732827   /home/tongpingliu/demo/linkingandloader/test
b7e7a000-b7e7c000 rw-p b7e7a000 00:00 0
b7e7c000-b7fd8000 r-xp 00000000 08:05 32899      /lib/tls/i686/cmov/libc-2.9.so
b7fd8000-b7fd9000 ---p 0015c000 08:05 32899      /lib/tls/i686/cmov/libc-2.9.so
b7fd9000-b7fdb000 r--p 0015c000 08:05 32899      /lib/tls/i686/cmov/libc-2.9.so
b7fdb000-b7fdc000 rw-p 0015e000 08:05 32899      /lib/tls/i686/cmov/libc-2.9.so
b7fdc000-b7fe1000 rw-p b7fdc000 00:00 0
b7fe1000-b7fe2000 r-xp b7fe1000 00:00 0          [vdso]
b7fe2000-b7ffe000 r-xp 00000000 08:05 33783      /lib/ld-2.9.so
b7ffe000-b7fff000 r--p 0001b000 08:05 33783      /lib/ld-2.9.so
b7fff000-b8000000 rw-p 0001c000 08:05 33783      /lib/ld-2.9.so
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
```

Finally, maximum is 9 but at 0xbffff65c
ptr: pointing to 0x804a068 but at 0xbffff660
Array is at 0x804a040

LD_PRELOAD mechanism

Build shared libraries memlib.so

- `gcc -fpic -shared -o memlib.so memlib.c`

Set LD_PRELOAD

- `export LD_PRELOAD=/path/to/memlib.so ./memtest`

Make sure the heap is initialized

Demo: memtest.c
/proc/PID/maps

Sample Examinations (Part 1 and Part 2)

Selection:

- **Which description about system call is correct? (multiple answers)**

Program understanding:

What is the printing of the following bash script? (5 points)

```
HELLO=Hello;
function funcHello {
   local HELLO = world;
   echo "$HELLO";
 }
 echo "hello";
 echo "$HELLO";
 funcHello;
 echo "$HELLO";
```

Sample Examinations (part 3)

Program writing:

1. Writing a perl program to print out each line of input.txt?

Sample Examinations (part 4)

Answering the questions:

1. What is the big difference between using static libraries and using dynamic libraries.

2. What shell will do after reading a command "find ./ –type f"?