CS35L Study Notes

wget

- Non-interactive network downloader

- Usage: wget <url>

diff

- Compare files line by line

- "+" and "-" indicates the different lines

Tr

-tr “set1” “set2” OR tr –cs “set1” “set2” < input.txt > output.txt, etc…

-c : Complement the set of characters in string1

-s : Replace each input sequence of repeated character listed in SET1 with single occurrence of that character

==SHELL AND THE SCRIPT===========================

A Unix shell is a command-line interpreter provides a traditional user interface for the Unix OS.

User runs programs by typing commands into shell.

Typical shell program in Unix:

- Bourne Shell /bin/sh

- C Shell csh (not shipped with Ubuntu)

- Bourne-Again Shell /bin/bash

Script Language

- Interpreted language

- Interpreter reads script, translates it into internal form, and execute programs

Script Languages vs Compiled Languages

- High level vs Low level,

- Compiled language is efficient but more complex to develop

Why Use a Shell Script?

- Simplicity, Portability, Ease of development

Typical Usage for Shell Scripts

- Automate frequent tasks, Simple text file manipulation, System admin tasks

--An Example of Shell Script---------------------

Objective: Count the number of files in current working directory

$ vim nfiles.sh write to file

ls | wc -l type and save quit

$ chmod +x nfiles.sh make it excutable

$ ./nfiles.sh run it!

Use the shell script interpretxer to run

$ sh nfiles.sh

--How does Shell Script work?-------------------

Shell Script read one -piece- of code each time:

- Built-in command

- Shell functions

- External command

Parent Shell ----wait child to finish---->Parent

| ^

----> Child Shell ---> New Program------|

Running in the background

- & at the end of the command/line of code

- Shell doesn't wait for the command to finish

if the program is running in the background.

== BASIC SHELL SCRIPTING ========================

--The #! First Line------------------------------

Shebang line tells the system to use which

interpreter to interpret and execute the code.

Examples:

#! /bin/sh

#! /usr/bin/python

Executable without shebang line

- System first try to run it as compiled

- "File is not runnable" error

- Retreat to /bin/sh

Why bother adding shebang line?

- Makes shell scripts more like real excutables

- 'ps' display real name instead of 'sh'

- Can be set-user-id

- Simpler to have alternative shells (e.g. csh)

--Variables---------------------------------------

Naming

- Start with letter/underscore

- Follow with letters/digits/underscores

Reference

first\_name = Chen

given\_name = $first\_name

last\_name = Liu

full\_name = "$first\_name $last\_name"

Variables hold string values

$ myvar=this\_is\_a\_long\_string\_that\_does\_not\_mean\_much *Assign a value*

$ echo $myvar *Print the value*

this\_is\_a\_long\_string\_that\_does\_not\_mean\_much

first=isaac middle=bashevis last=singer //*Multiple assignments allowed on one line*

fullname="isaac bashevis singer" //*Use quotes for whitespace in value*

oldname=$fullname //*Quotes not needed to preserve spaces in value*

fullname="$first $middle $last" //*Double quotes required here, for concatenating*

--Arguments----------------------------------------

Reference the arguments with $1, $2, ...

first\_name = $1

echo first\_name

--Print--------------------------------------------

echo

- print the string following 'echo'

- no extra options

- Ex:

$ echo Hello, world

Hello, world

printf

- printf format-string [args...]

- Ex:

$ printf "Hello, world\n"

Hello, world

$ printf "%s, %s\n" Hello world

Hello, world

--Execution Tracing---------------------------------

sh -x tells shell to print executing command, and

precede it with "+"

Example:

$ sh -x nfiles.sh

+ wc -l

+ ls

3

in the script

set -x turn on execution tracing

set +x turn off execution tracing

== TEXT HANDLING AND REGULAR EXPRESSION ============

--Text Handling-------------------------------------

- Shell command manipulate on text

- Variables are strings

- Input/Output are read as text

Recall the Unix tooling philosophy introduced in previous lecture.

----------------------------------------------------

--Search for text-----------------------------------

Print lines matching a pattern

- grep: Uses basic regular expressions (BRE)

- egrep: Use extended regular expressions (ERE)

same as grep -E

- fgrep: Fast grep matches fixed strings

same as grep -F

Grep with pipeline

- ls | grep py

- compare to find . -name \*py

- ls -l | grep 2010/09/

- how to make a equivalent command with find?

Grep with I/O redirection

- grep 'UCLA' < lakers\_roster.txt

--Regular Expressions-------------------------------

A regular expression provides a concise and flexible

means for "matching".

--Anchors--

\* Refer to the start and end of text units

\* ^ start of line

$ end of line

--Character Classes--

\s white space

\S NOT white space

\d digit

. any character except new line (\n)

--Special Characters--

\ Escape Character (\\* to match real '\*')

\n new line

\r carriage return

\t tab

--Quantifiers--

\* 0 or more

+ 1 or more

? 0 or 1

{3} exactly 3

{3,5} 3, 4 or 5

Non-greedy quantifier is not supported in POSIX/GNU

Use perl regexp instead by -P

greedy: grep 'e.\*o'

non-greedy: grep -P 'e.\*?o'

--POSIX Character Classes--

[:upper:] upper case letters

[:alpha:] all letters

[:punct:] punctuation

[:word:] digits, letters and underscore

To know more regexp, try search it on web

e.g. google 'regexp python newline'

--Shell Pattern vs RE-------------------------------

Shell use 'Shell Pattern' to match file names

\* match any characters

? match any single character

[abcd] match any characters within the bracket

find -name '?' correct

man -k '?' wrong

find -regexp '?' wrong

Most programs support regular expression, while some

programs support shell pattern. To find the exact

usage, please read the man page of the program.

--Other Text Processing Tools-----------------------

sort: sorts text

wc: output word count report (lines, words, chars)

head: output first lines of a file

tail: output last lines of a file

tr: transliterate files with a pattern

Read the man page of the above tools to know their

full functionality.

-- POSIX Regular Expression ---------------------

Regular Expressions are widely used in many

different tools. When you are using tools such as

grep, you should follow the POSIX standard.

BRE - Basic Regular Expression

ERE - Extended Regular Expression

POSIX Metacharacters Tables

Character BRE/ERE Meaning in Pattern

\ Both Turn off metacharacters, turn

on special usage.

Ex: \\* matches real '\*'

\{ indicates interval expr

. Both Any characters except '\n'

\* Both 0 or more occurrence of previous

chracter or subpattern

^ Both Anchor: start of line

$ Both Anchor: end of line

[ ] Both Match any chracters within the bracket

[^ ] Both Match any chracters but chracters

within the bracket

(n,m) ERE Define a subpattern.

\(n,m\) /BRE \n can be used to refer to n-th subpattern

\n is BRE-only

{n,m} ERE Match number of occurrences of preceded

\{n, m\} /BRE character or subpattern

? ERE Matches the preceded character or subpattern

zero or one time.

+ ERE Matches the preceded character or subpattern

one or more times

| ERE Match patterns from expression on either side

-- Regular Expressions Examples -------------------------------------

[[:alpha:]\_][[:alnum:]]\* variable name

[[:digit:]]\{3\}-[[:digit:]]\{3\}-[[:digit:]]\{4\} phone number

[[:alnum:]\_]\*\(\.[[:alnum:]]\*\)\{0,1\}@[[:alnum:]-\_]\*\.[[alnum:]]\*\(.[[alnum:]]\*\)\*

-- Basic Usage of sed ----------------------------------------------

Search a pattern and replace.

sed 's/pattern/replace/'

Examples:

- Print all the players numbers, sort by number

sed 's/\t.\*//' lakers\_roster.txt | sort -n

- Print all the players names only

sed 's/^[[:alnum:]-]\*\t//' lakers\_roster.txt \

| sed 's/\t.\*//' | sed 's/PLAYER/Names/' | less

- Create the subdirectory hierarchies as the given directory

find /home/ubuntu -type d -print |

sed 's;/home/ubuntu/;/home/ubuntu/test2/;' |

sed 's/^/mkdir /' |

sh -x

--------------------------------------------------------------------

-- Regular Expression in Vim ---------------------------------------

Similar grammar as ed and sed.

Search/Replace

:%s/pattern/string/cgiI

% - file scope, you can choose other scopes

Some differences:

Character classes

Quantifiers

\{-} - non-greedy matches more or more patterns

http://vimregex.com/

----------------------------------------------------------------------

-- Regular Expression in Emacs ---------------------------------------

Grammar is similar to vim, with exception e.g. syntax group.

Some Emacs Commands that use regexp

C-M-s incremental forward search matching

C-M-r incremental backward search matching

C-x query-replace-regexp query and replace

C-x grep call grep and put result in a buffer

== More Shell Scripting =============================

-- Variables Revisited ------------------------------

Arguments

- $n n-th argument, ${n} for n >= 10

- $\* all argument as one string, "arg1 arg2 arg3"

- $@ all argument as separated strings, "arg1" "arg2" "arg3"

Traverse the arguments

for i in $\*

> do echo i is $i

> done

Arithmetic

- Variables are stored as string, but it will expand

to numeric with arithmetic operators within $((...))

i=1

echo $i++

echo $((i++)) $i

1 2

echo $((++i)) $i

3 3

--Environment--

export $variable

- Put the variable into the environment

env $variable

- Temporarily change the environment variable

unset $variable

- Remove variable from current shell

-- Making Decisions-----------------------------------

-- Exit Code --

exit [exit-value]

The default exit status used if none is supplied is the

exit status of last command executed.

0 Command exited successfully

>0 Command failed

1-125 Command exited unsuccessfully

126 Command found, but file was not executable

127 Command not found

>128 Command died due to receiving a signal (e.g. ^C)

-- Make decisions based on exit code --

if pipeline1

then

statements-if-true-1

elif pipeline

statements-if-true-2

else

statements-if-all-fails

fi

-- The test command --

if test "$str1" = "$str2"

then

...

fi

if test $1 -eq $2

then

printf "%d equals to %d\n" $1 $2

fi

numeric test is integer only

-- Case statement -------------------------------------

case $1 in

-f)

...

-d | --directory)

...

\*)

echo $1: unknown option

exit 1

esac

-- Loops ----------------------------------------------

-- for loop --

Make a copy of every \*.txt file, and modify the original

version by replacing 'A' with '-A-'.

for i in \*.txt

do

echo $i

mv $i $i.old

sed 's/A/-A-/g' < $i.old > $i

done

-- while loop --

while condition

do

statements

done

until condition

do

statements

done

break / continue works the same as in c+++

-- Function ---------------------------------------

define a function:

wait\_for\_user() {

until who | grep "$1" > /dev/null

do

sleep ${2:-30}

done

return 0

}

What will the following commands do?

wait\_for\_user chenliu 20

wait\_for\_user chenliu

Example: word count for all the files:

mywc() {

echo "Number of words in $1"

wc -w < $1

}

for f in "$@"

do

mywc "$f"

done

-- String Manipulation --------------------------------

-- Pattern Matching Operators --

${variable#pattern}

- delete the shortest match of pattern from beginning

and return the rest

${variable##pattern}

- delete the longest match of pattern from beginning

and return the rest

${variable%pattern}

- delete the shortest match of pattern from the end

and return the rest

${variable%%pattern}

- delete the longest match of pattern from the end

and return the rest

NOTE: use the shell pattern instead of regexp

-- Substring Operators --

${string:position}

- returns the substring from $string at $position

${string:position:length}

- returns the substring from $string at $position with

length $length

-- Length of String --

${#string}

- returns the length of string $string

-- Quoting and Evaluation ------------------------------------

Backslash escaping

- escape the metachracters

Single quotes

- treat everything inside the single quotes as literature

Double quotes

- group enclosed text as a single string

eval

- run the arguments through a command-line processing steps

all over again

- Example:

listpage="ls | less"

$listpage #wrong! | and less are passed as arguments

eval $listpage #correct! equivalent to "ls | less"

---------------------------------------------------------------

-- More About the Loops -------------------------

--For loop--

--Loop through a sequence of numbers--

for ((i=0; i<10; i++))

do

echo $i

done

--Loop through a result of previous command--

lscommand=`ls`

for i in $lscommand

do

echo $i

done

--Loop through the content of a file--

for i in `cat`

do

echo "i is $i"

done < ls.txt

-- Read input to Shell --------------------------

--read command--

echo "Please input your words"

read words

echo $words

--Read the content of file using read--

while read var

do

echo $var

done < file.txt

---- Test command and [] ------------------------

-- Test the return value --

if grep 'abc' > /dev/null

then

echo 'yes!'

fi < abc.txt

grep 'abc' > /dev/null

if test $? -eq 0

then

echo 'yes!'

fi < abc.txt

-- Test a file --

File exists is built-in in every file test.

Use && to group multiple creteria.

test -r test2.txt && test -x test2.txt

-- [ ] as a shorthand

[ -f test.txt ]

\* leave spaces before and after [ ]

\* don't write test again, [] = test

\* use &&, || to do boolean arithmetics

---- Quotes -------------------------------------

-- Backslash \ --

\ escapes a single character

- turn a special character into its literal

e.g. \\*

- form special characters

e.g. \n

-- Single Quotes '' --

' ' tells shell to treat all the content enclosed

as their literature. (Strong quote)

-- Double Quotes "" --

" " tells shell to treat all the content enclosed

as a single string, but perform variable expansion

and special character handling first. (Weak quote)

-- Back Quotes `` --

` ` tells shell to execute the command enclosed in

a subshell and replaced it with the results of the

command.

---- Evaluation Order ---------------------------

Variables are expanded before the whole command

is executed.

e.g. ls $dir

The variable $dir will be expanded before the

command ls running.

Variable expansion is only performed once.

e.g. lsd = 'ls $dir'

$lsd #incorrect!!

Redirections can not be enclosed in a variable

e.g. readtext='cat < $textfile'

$readtext

'cat: '<', no such file or directory

Use eval command to start a new evaluation.

==== BUILD A COMMAND LINE TOOL USING PYTHON =====

---- Command Line Tools Revisited ---------------

-- What is a command line tool? --

A command line tool performs a task given input

from command line environment and output the

results after finishing the task.

-- Pros and Cons --

+ Modularity and reusability

+ Easy to automate

+ Easy to be evoked from different PLs

- Low usability

- Rely heavily on documentation

- Cannot do too complicated jobs (MIMO)

-- Philosophy --

\* Do one thing well

\* Error handling on input / output

\* Make output adaptable to other tools

-- Essentials of Command Line Tools --

\* Command

\* Short Option

\* Long Option

\* Option argument

\* Arguments

Example:

make -f myMakeFile -s --ignore-errors\

--include-dir="/var/lib/abc" myprj

---- optparse Module in Python -------------------

-- optparse --

optparse module is used for parsing command-line

options for command-line tools

Documentation: http://docs.python.org/library/optparse.html

-- OptionParser class --

>>> from optparse import OptionParser

>>> help(OptionParser)

parser = OptionParser()

-- Parse an option with option argument --

parser.add\_option("-n", "--numlines",

action="store", dest="numlines", default=1,

help="output NUMLINES lines (default 1)")

Store action tells optparse to take next argument into an

instance variable of OptionParser object.

-- Parse a bloolean option / flag --

parser.add\_option("-v", action="store\_true", dest="verbose")

-- Reference the value of option argument --

parser.numlines

---- Add your tools to Path ------------------------

-- Put your file to a folder --

e.g. /home/yourname/tools/

-- Make your scripts excutable --

chmod 755 yourscripts

-- Add the folder to PATH --

# Go to your home folder

cd ~

vim .bashrc

# Add the following line

export PATH="$PATH:/home/yourname/tools/"

Adding your tools to Path enables you call your

script without specifying the full path.

==== INTRODUCE MAKEFILE =============================

---- Make utility -----------------------------------

Run make command

- The program will look for a file named "Makefile"

in your directory and execute it.

make -f MyMakeFile

- Execute the specified make file

It is makefile actually defines the way we build a

software. Make is the command to interpret Makefile

and execute it.

-----------------------------------------------------

---- Build Process of C/C++ Programs ----------------

1 Compile the source file and output object files

2 Link the object files to assemble an executable

If we use compiler and linker to do this file by

file, it may requires hundreds lines of code.

Even worse, if we write them into a script, a minor

change such as rename a component may require

rewriting of the whole script.

Makefile is the most efficient way to automate the

build process.

-----------------------------------------------------

---- An Anatonomy of a Basic Makefile ---------------

-- Syntax --

target: dependencies

[tab] system command

Ex: Makefile-1

-- Dependencies --

For a complex software, usually changing one part

of the software will not affect the other parts.

We can specify the dependencies in Makefile by grouping

different sources into different targets.

Ex: Makefile-2

--------------

all: hello

hello: main.o factorial.o hello.o

g++ main.o factorial.o hello.o -o hello

main.o: main.cpp

g++ -c main.cpp

factorial.o: factorial.cpp

g++ -c factorial.cpp

hello.o: hello.cpp

g++ -c hello.cpp

clean:

rm -rf \*.o hello

--------------

If we change the file factorial.cpp, when execute make:

(1) factorial.cpp has changed, so factorial.cpp will be

recompiled and factorial.o will be regenerated.

(2) After factorial.o regenerated, the target hello will

be triggered and new hello will be relinked and a

new excutable will be generated.

During this process, hello.o, main.o are not changed.

-- Variables and comments --

Sometimes, we would like to set some global parameters

(e.g. the compiler, the options for the compiler, etc.)

This allows users to customize the building behavior

with minimum efforts.

-----------------------------------------------------

==== Autoconf and automake ==========================

-- The classic installation process of GNU tools --

./configure

make

make install

-- Why not simply use make? --

C/C++ code are not always portable

\* Different libraries

\* Different system specs (32bit/64bit)

\* Not all users are capable to modify Makefile

to fit their system.

-- How to use autotools to make code portable--

\* autoconf - generate the "configure" script

\* automake - generate Makefile

\* Libtool - create platform-independently

shared libraries

-- Install autotools --

\* autoconf, automake

sudo apt-get install autoconf

\* libtool

sudo apt-get install libtool

-- Use auto tools on our example --

1 ) run "autoscan" in your project folder

autoscan is the tool to prepare the input

for autoconf

2 ) mv configure.scan configrue.ac

Review configure.scan and change the content

if necessary, then save it with name configure.ac

3 ) create Makefile.in

mv Makefile Makefile.in

This Makefile will be used as a template to add

portable features into it.

4 ) run configuration file

./configure

5 ) Now you can make your code.

=== Diff and Patch ==================================================

----- Introduction to Diff ------------------------------------------

-- What is diff? --

diff - compare files line by line

diff helps you to find the difference between

two files.

- Source code versioning

- Compare similar files

- Extract useful information

-- Basic Output Format --

diff <from-file> <to-file>

The normal output format consists of one or more hunks of differences;

each hunk shows one area where the files differ. Normal format hunks

look like this:

CHANGE-COMMAND

< FROM-FILE-LINE

< FROM-FILE-LINE

---

> TO-FILE-LINE

> TO-FILE-LINE

There are 3 different change-commands:

\* FaT

Add the lines in range T of to-file after line F of from-file

\* FcT

Replace the lines in range F of from-file with lines in range T of to-file

\* FdT

Delete the lines in range F, and T is the where they would have

appeared had they not been deleted.

-- Unified Output Format --

diff -u <from-file> <to-file>

~~ Header ~~

The unified output format starts with a two-line header, which

looks like this:

--- FROM-FILE FROM-FILE-MOD-TIME

+++ TO-FILE TO-FILE-MOD-TIME

~~ Hunks ~~

Each hunk of unified output format shows one area where the files

differ. Unified format hunks look like this:

@@ FROM-FILE-RANGE TO-FILE-RANGE @@

LINE-FROM-BOTH-FILE

LINE-FROM-BOTH-FILE

The lines is preceeded by following prefixes:

' ' indicates the line is common to both files

'+' indicates the line is added here to the from-file

'-' indicates the line is removed here from the from-file

Unified output format is widely used in GNU patches.

-- Diff Multiple files --

diff <from-folder> <to-folder>

diff will autodetect the input is a file or a directory. And diff

each files in the directory

If you want to diff into subdirectories recursively, you may want to

use:

diff -r <from-folder> <to-folder>

---------------------------------------------------------------------

----- Introduction to Patch -----------------------------------------

-- What is patch? --

patch - apply a diff file to an original

-- Run patch --

patch [options] [originalfile [patchfile]]

We often use this line:

patch -pNUM < patchfile

-- patch is error tolerant --

\* Detect the diff file content intelligently

\* Remove leading junk information

\* Remove tailing junk information

\* Correct the line numbers

\* Automatic detect the filenames given input

-- -p option --

-p option give patch the clue to figure out what is the filename to

patch on by indicating the number of "slashes" to ignore in the

filenames

e.g. a patch may include the following filename:

/home/tom/software/abctool/src/abc.c

And Jim actually want to apply this patch to his own folder at:

/home/jim/tools/abctool/src/abc.c

patch -p4 will match the following filenames from working directory:

abctool/src/abc.c

Please use -p option in your patch, since there may be a lot of

ambiguity in code files management.

-- Reverse an applied patch --

patch pNUM -R < patchfile

---------------------------------------------------------------------

=====================================================================

=== Lab 4 Getting Started ===========================================

1) Grab You package

wget ftp://alpha.gnu.org/gnu/coreutils/coreutils-8.0.tar.gz

tar -xzvf coreutils-8.0.tar.gz

Here you go with a folder called coreutils-8.0

2) Create Patch

Copy and paste the patch from email archive to your own patch file

3) Try the patch

Try the patch using the command we introduced in today's lecture

and analysis the errors. (Log them!)

4) Figure out the correct patch

You need a "correct" patch, please note the difference between

version 8.0 to the version of the original patch. Edit the patch using

your favourite editors

5) Patch and fix

6) Generate your own patch using diff

=====================================================================

=== Version Control System ==========================================

--- Introduction to Version Control ---------------------------------

-- What is version control? --

Versioning is the process of assigning either unique version names

or unique version numbers to unique states of computer software.

-- Version Control --

In software development, usually a team of people may change the

same files. It is important to keep track of every changes made by

team members and to guarantee that each individual change will not

break or badly affect the entire system.

-- Version Control System --

A system enables users to manage the different states of a software

throughout the development cycle. Usually, it provides the following

functionality:

\* Check out source code. Sync changes

\* Check in changes

\* Diff, merge and confliction handling

\* Branching

-- Some terminology --

Project - A collection of files saved in the version control system

Repository - The location where the project files is stored

Revision - A certain version of a file

Branch - A fork of the project

---------------------------------------------------------------------

--- Operations on Version Control Systems ---------------------------

Sync, Update - update local files to the latest version

Commit - Submit local edits to the project

Diff - compare local files to the latest (or any) version in the

repository

Conflict - The situation that local files are not compatible to the

version in repository, which may prevent sync/commit. A typical

reason is that the local version is modified content edited by

other users simultaneously.

Merge - The process of merging local changes with changes done by

others.

Revert - Abandon local changes and revert to latest version in

repository

Add - To add a file to the project

Remove - To remove a file from the project

---------------------------------------------------------------------

--- Typical Version Control Systems ---------------------------------

-- Subversion --

Subversion is a open-source source control system and widely used in

many other open-source projects. A repository is create on an SVN

"server" and you can use SVN clients to connect to it and make

editions.

-- Git --

Git is an open-source source control system and get popular among

individual developers but also large corporations. Git is a

decentralized source control system, which provides much more

flexibility and increases the efficiency of version controlling.

-- Microsoft Team Foundation Server --

Offering source control with data collection, reporting and project

tracking. Three-tier architecture: client tier, the application tier

and the data tier. It is part of Visual Studio Team System.

---------------------------------------------------------------------

== More About Diff File Format =======================

-- An Anatonomy of Unified File Format --

diff -p -u -r1.134 rm.c

--- src/rm.c 14 May 2005 07:58:37 -0000

+++ src/rm.c 29 Aug 2005 21:11:54 -0000

@@ -51,7 +51,9 @@

#include "system.h"

#include "dirname.h"

#include "error.h"

+#include "lstat.h"

#include "quote.h"

+#include "quotearg.h"

#include "remove.h"

#include "root-dev-ino.h"

@@ -95,6 +97,33 @@ static struct option const long\_opts[] =

Line 1: the diff command create this file

Line 2,3: file names and time stamps

Line 4: Range of file

\* -/+ indicates from/to file

\* first number indicates starting line of hunk

\* second number indicates the length of hunk

Starting from line 5 until next "@@":

the content of hunk

-- Hints when modifying diff file --

\* Don't change the file names

\* Make sure the line numbers are correct

\* Make minimal change as possible

======================================================

== Introduction to Git ===============================

-- Taxonomy of Version Control Systems --

\* Local Version Control Systems

Local database keeps tracks of history of files.

e.g. Revision history of Google docs

\* Centralized Version Control Systems

Put all the files on central server and clients

sync with the server.

e.g. Subversion, CVS

\* Distributed Version Control Systems

Clients don't just check out latest snapshot, but

fully mirror the repository.

e.g. Git, Mercurial, Bazaars etc.

-- History of Git --

\* Linux kernel development requires a DVCS

\* BitKeeper, a commercial software

\* Linus Torvalds decide to build a free version

\* Git is born in 2005

-- Three States of Git --

working | staging | repository

<----------Checkout--

---Stage-->

---Commit-->

-- Basic Workflow --

\* Change the files in working directory

\* Stage the changed files

\* Do a commit into repository

-- Git Cheat Sheet -----------------------------------

-- Install Git --

\* apt-get install git-core

-- Setup Git --

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

$ git config --global core.editor emacs

$ git config --global merge.tool vimdiff

$ git config --list

-- Initialize a repository --

# Go to the project's directory

$ git init

# Add files (stage them) --

$ git add \*.c

$ git add README

# Commit the changes

$ git commit -m 'Initial project checkin'

-- Clone an existing repository --

$ git clone <git-address>

-- Checking status --

$ git status

-- Committing all changed files (skip staging) --

$ git commit -a -m "first commit"

-- Revert your changes --

$ git reset --hard

-- View changes --

$ git diff <file>

-- Add a remote repository --

$ git remote add <remotename> <git-addr>

-- Push to a remote repository --

$ git push <remotename>

-- Update from a remote repository --

$ git pull

-------------------------------------------------------

-- Git Branch -----------------------------------------

-- Create a branch --

$ git branch iss53 # create a branch issue 53

-- Check out a branch --

$ git checkout -b iss53 # checkout a new branch issue 53

After commit, your branch will have a new snapshot

-- Merge branches --

$ git chekcout master

$ git merge iss53 # merge iss53 with master branch

-- See last commit of each branch --

$ git branch -v

-------------------------------------------------------

More information, see http://progit.org/book/

=======================================================

== INTRODUCTION TO GITHUB =============================

-- Setting up GitHub --

http://help.github.com/linux-set-up-git/

-- SSH Pub Keys --

In order to use GitHub on your machine, you need update

Your SSH key to GitHub.

-- Generate an SSH Key --

$ ssh-keygen -t rsa -C <your-email>

<Press Enter three times>

-- Upload to Git --

Account Settings > SSH Public Keys > Add another pub key

-- Test connection --

ssh -T git@github.com

-- Create new repository --

http://help.github.com/create-a-repo/

=======================================================

==== 1 C for C++ Programmers =========================

-- About the name "C++" --

You must know the self-increment operator "++", if

we have i = 1, then expression "i++" still evaluated

as 1. That's the idea for C++, C++ is compatible to

C programming language, but with some extension.

-- "C = C++--" --

Most language features of C++ is compatible with C,

except the following:

\* classes and object=oriented features (including

templates, inheritance, override etc.)

\* new and delete.

\* The stream operator >> and <<

\* standard libraries (e.g. <iostream>, <string>)

\* flexible variable definition

---- 1.1 I/O operations ----

In C, there is no I/O stream objects like istream

or ostream class in <iostream>. We use the I/O

functions provided in <stdio.h> to handle the inputs

and outputs.

#include <stdio.h>

-- FILE pointer --

FILE \* is a pointer to a file, <stdio.h> provides

three global pointers: stdout, stderr, stdin.

Recall that in Unix, input output streams is actually

a file.

We can open/close regular files with the following

functions:

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

int fclose(FILE \*stream);

We can read/write bytes from file using:

size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

In most situations, we write characters (string) to a

file, the following is the most frequently used I/O

functions handling characters / strings.

-- Write output to files (streams) --

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

int fprintf(FILE \*stream, const char \*format, ...);

int fputs(const char \*str, FILE \*stream);

int fputc(int char, FILE \*stream);

-- Read input from files (streams) --

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

int fscanf(FILE \*stream, const char \*format, ...);

char \*fgets(char \*str, int n, FILE \*stream);

int fgetc(FILE \*stream);

-- String format --

%s char\* string (character array)

%d int integers %ld long long integer

%f float float number %lf double double-accuracy floats

---- 1.2 Memory Allocation ----

Memory allocation in C is performed in a very low level,

the programmer need to specify the size of the memory block

they want to allocate, and the pointer pointing to the

block of memory.

#include<stdlib.h>

void \*malloc(size\_t size);

void free(void \*ptr);

void \*realloc(void \*ptr, size\_t size);

Example:

int i;

int \*arr = malloc(20 \* sizeof(int));

for (i = 0; i < 20; i++)

arr[i] = i;

free(arr);

Always free after malloc!!

---- 1.3 Struct ----

In C, there is no concept of class, but we can still

define struct for complex data types.

struct point {

int x;

int y;

};

-- typedef --

typedef creates an alias of one type.

Example: typedef struct point Point;

We usually integrate struct declaration with typedef:

typedef struct point {

int x;

int y;

} Point;

-- Use struct variables --

Point\* ptrX = malloc(sizeof(Point));

ptrX->x = 1;

ptrX->y = 2;

free(ptrX);

---- 1.4 Pointers ----

Pointer store the address of a block of memory.

It is associated with a type.

int \*ptrNum;

You can dereference the pointer with '\*', and []

\*ptrNum

ptrNum[0]

\*(ptrNum + 2)

\*ptrNum + 2

You can get a pointer to a variable using &

ptrNum = &iNum;

-- Pointers to pointers --

char c = 'A';

char \*ptrC = &c;

char \*\*ptrPtrC = &ptrC;

======================================================

== 2 Compile C Code ==================================

From source code to executables:

Libraries

Source ------------> Assembly-------------> Excutable

Code compiler Code Link Editor

Compiler

\* Preprocessing the code

- Remove comments

- Interpreting macros

\* Translate source code to assembly code.

Assembler

\* Creates object code

Link Editor

\* Combine symbols from multiple files, if the source

code references library functions or functions in

other source files.

======================================================

== 3 Introduction to GDB =============================

-- What is a debugger --

The purpose of a debugger such as gdb is to allow

you to see what is going on “inside” another program

while it executes—or what another program was doing at

the moment it crashed.

-- Example: What's wrong with divint --

- download the code

wget http://cs.ucla.edu/~chenliu/cs35l/lecture9/divint.c

- compile and run

gcc -g divint.c -o divint

./divint

Floating point exception

- debug!

gdb divint

(gdb) run # execute the code

... # the code hits the rock and hang

(gdb) where # which line causes the problem?

...

(gdb) list # list the code near the position

...

(gdb) print rhs

$1 = 0 # ah! Dividing by zero!

(gdb) up # check what had happened when calling the

# function from upper level

(gdb) list # oops! y is 0.

More power of GDB will be revealed in next lecture.

======================================================

==== BEFORE YOU DEBUG IT ... =========================

-- Why we can debug? --

Debugger relies on the debugging symbols to recognize

the place it should look into. NOT every executable binary

can be simply loaded into gdb for debuggin.

Q: I want to debug the whole Linux. Is it possible?

The Answer is "maybe", the debugging symbols are created

during the compiling process. If you compile your code

with:

gcc -g # enable standard debugging symbols

gcc -ggdb # enable gdb specific debugging symbols

-- Start and quit gdb --

You can start gdb using:

$ gdb [<binary>]

You can quit gdb using:

(gdb) quit

-- Getting Help --

You can always get help using:

(gdb) help <gdb-command>

-- Shorthand on commands --

You can always be "lazy" in gdb, by just typing part

of your command. E.g. "l" equals to "list".

======================================================

==== BASIC GDB COMMANDS ==============================

-- Listing Source --

\* list your current location

(gdb) list

\* list a function

(gdb) list divint

-- Run the Program --

\* run your program until exiting

(gdb) run

-- Run with the args --

\* (gdb) run <args>

# You don't need to specify your program name here

e.g. if you are debugging ls

you can use run -t to debug with the option t set.

\* gdb --args <binary> <args>

If you envoke gdb in this way, when you execute run

command, your gdb will start the program with the args

you have specified.

-- Printing Variables and Expressions --

\* print the value of a variable

(gdb) print guest\_name

$1 = 0x4005e4 "Ziggy"

\* print the value of an expression

(gdb) print strlen(guest\_name)

$2 = 5

\* print a temporary variable

print $2

$3 = 5

\* print data given format

(gdb) print [/format] <expression>

-d: decimal int

-u: decimal unsigned

-x: hexadecimal

-o: octal

-t: binary

-c: character

-- Run gdb in Emacs --

M-x gdb

======================================================

==== NAVIGATE IN YOUR PROGRAM ========================

-- Breakpoints --

Breakpoints are stop signs for the program. After

setting a break point, gdb will return control to user

when the code hits the breakpoint. Use breakpoints to stop

your program whenever necessary.

-- Set Breakpoints --

\* Set break point to a given line number

(gdb) break 12

Breakpoint 1 at 0x4004c1: file hello.c, line 12.

\* Set break point to a given function

(gdb) break main

Breakpoint 2 at 0x4004c1: file hello.c, line 10.

-- Toggle Breakpoints --

\* Temporarily deactivate a breakpoint

(gdb) disable #BP

\* Reactivate a disabled breakpoint

(gdb) enable #BP

-- Set Conditions to Break Points --

(gdb) condition #BP <cond>

-- Remove Breakpoints --

\* Delete a specific breakpoint

(gdb) delete #BP

\* Delte all break points

(gdb) delete

-- Set Conditions

-- Navigate the program --

\* Step forward (into)

(gdb) step

\* Step forward (over)

(gdb) next

======================================================

==== KNOW ABOUT YOUR PROGRAM =========================

-- Watch a variable / expression --

'print' command can give us information about the data

in the program, but we may not necessarily know when

exactly do we need to print.

\* Stop the program if the value of expression changed

(gdb) watch <expression>

\* Stop the program whenever the value of the program reads

the value of any object involved in the evaluation of

expression

(gdb) rwatch <expression>

\* Stop the program for either case above

(gdb) awatch <expression>

-- Display a variable / expression --

(gdb) display <expression>

print the value of expression whenever the program

stops. Format settings can be used.

(gdb) undisplay #DISP

cancel the display #DISP.

-- Find out the backtrace --

(gdb) bt

#0 0x000000000040050d in divint (lhs=3, rhs=0) at divint.c:16

#1 0x00000000004004e3 in main () at divint.c:10

The numbers followed by '#' is the frame stack number, 0

is your current frame.

-- Navigate to a given frame --

(gdb) frame #FRM

Bring you to the frame #FRM, you can also use up and

down to navigate between neighbor frames.

-- Get the info about the stack --

\* info about current frame

(gdb) info frame

Stack level 0, frame at 0x7fffffffe830:

rip = 0x40050d in divint (divint.c:16); saved rip 0x4004e3

called by frame at 0x7fffffffe850

source language c.

Arglist at 0x7fffffffe820, args: lhs=3, rhs=0

Locals at 0x7fffffffe820, Previous frame's sp is 0x7fffffffe830

Saved registers:

rbp at 0x7fffffffe820, rip at 0x7fffffffe828

\* list local variables

(gdb) info locals

\* list arguments

(gdb) info args

== SYSTEM CALLS vs LIBRARY CALLS =====================

-- System Calls --

System calls are requests of service from an operating

system's kernel.

A system call looks like a regular function call, but

it's different -- it requests a service from an operating

system's kernel. This may include hardware services(e.g.

status of disk usage), creating and executing processes

and communicating with integral kernel services (e.g.

scheduling).

-- CPU modes --

There are two operating modes in CPU architecture.

\* Kernel mode

Unrestricted mode. Full instruction availability. I/O

operation, area of memory accessed are unlimited.

\* User mode

Limited instructions can be executed. I/O are not

permitted. Limited access to memory.

-- Privilege Levels --

The design of modern systems design a security model to

specify multiple previliege levels under which a software

may be executed.

Applications are usually prevented to access other

running applications or operating system itself. They are

confined in its own address space.

When an application requires access to higher

privilledge level, it envokes a system call.

-- Library Calls --

Systems provide a library to offer API to system calls.

\* Wrapper of system calls

\* Exposes a calling convention

\* Make use of system calls more modular

\* Increase portability

\* Typical implementation: glibc

======================================================

== HOW LIBRARY CALL WORKS? EXAMPLE: printf ===========

-- Userspace printf() --

\* Construct the string

\* Cannot actual print, not in kernel mode

\* Call write()

-- Library call write() --

\* Place the string in register, place int "1" in another

\* calls an interrupt with the system call handler

-- Enter the Kernel --

\* Read values from register specified

\* Write to stdout

\* Return control to the caller

-- Done --

\* write() returns to printf()

\* printf() returns

Note: this is not a complete walk-through of printf

function, it is just for the purpose of illustrating how

system calls are made from library calls.

======================================================

== syscall() and "System Call Functions" =============

1 #include <unistd.h>

2 #include <stdio.h>

3 #include <sys/types.h>

4 #include <sys/syscall.h>

5 int main(void) {

6 long ID1, ID2;

7 /\*-----------------------------\*/

8 /\* direct system call \*/

9 /\* SYS\_getpid (func no. is 20) \*/

10 /\*-----------------------------\*/

11 ID1 = syscall(SYS\_getpid);

12 printf ("syscall(SYS\_getpid)=%ld\n", ID1);

13 /\*-----------------------------\*/

14 /\* "libc" wrapped system call \*/

15 /\* SYS\_getpid (Func No. is 20) \*/

16 /\*-----------------------------\*/

17 ID2 = getpid();

18 printf ("getpid()=%ld\n", ID2);

19 return(0);

20 }

In many situations, people call the wrapper functions such

as getpid() as "System Call" as well.

======================================================

== EXAMPLE SYSTEM CALLS ==============================

#include <unistd.h>

-- Process --

void exit(int);

pid\_t fork(void);

-- I/O --

ssize\_t read(int, void \*, size\_t);

ssize\_t write(int, const void \*, size\_t);

-- File Status --

int fstat(int filedes, struct stat \*buf);

======================================================

=== PROCESS MEMORY REGION ============================

/------------------\ lower

Code | | memory

Read-only data | Text | addresses

| |

|------------------|

static vars | (Initialized) |

| Data |

| (Uninitialized) |

|------------------|

| |

| Stack | higher

| | memory

\------------------/ addresses

Fig. 1 Process Memory Regions

-- What's a Stack? --

Stack - LIFO data structure

High level programs are built upon the concept of

stack.

\* Procedure Calls

\* Dynamically allocate local variables

\* Pass parameters

\* Return values

-- The Stack Region --

\* Contiguous block of memory containing data

\* The stack pointer (SP) points to top of stack

\* Bottom of stack is at a fixed address

\* Size is dynamically adjusted by the kernel

\* CPU implements PUSH/POP instructions

=== STACK FRAMES =====================================

-- Stack Frame --

\* A logical frame of data

\* Pushed/Popped when calling/returning a function

\* Parameters, local vars, data necessary to recover

previous stack frame

-- Frame Pointer (FP) --

FP points to fixed location within a frame and

varaibles are referenced by offset to the FP.

When a procedure call happens, FP is stored.

-- example1.c --

void function(int a, int b, int c) {

int buffer1[2];

int buffer2[4];

buffer1[0] = 0;

buffer1[1] = a;

buffer2[0] = b;

buffer2[3] = c;

}

void main() {

function(1,2,3);

}

-- Compile the code into assmeblers --

gcc -fno-stack-protector -S -o example1.s example1.c

-- Essential of assembler code --

~~ Inside main() ~~

pushl %ebp

movl %esp, %ebp

subl $12, %esp

movl $3, 8(%esp)

movl $2, 4(%esp)

movl $1, (%esp)

call function

~~ Inside function() ~~

pushl %ebp

movl %esp, %ebp

subl $32, %esp

movl $0, -8(%ebp)

movl 8(%ebp), %eax

movl %eax, -4(%ebp)

movl 12(%ebp), %eax

movl %eax, -24(%ebp)

movl 16(%ebp), %eax

movl %eax, -12(%ebp)

leave

~~ Instructions ~~

pushl - push a value into stack

movl - move data from one reg/mem to another

subl - substract the value

call - call function

leave - leave the function

~~ Registers ~~

%rsp: stack pointer

%rbp: frame pointer

%eax: general data registers

-- Stack status when "function" is called --

[ buffer2 ] [ buffer1 ] [fp(main)] [ret] [a] [b] [c]

=== BUFFER OVERFLOW ==================================

A buffer overflow is the result of stuffing more data

into a buffer than it can handle.

-- example2.c --

void function(char \*str) {

char buffer[16];

strcpy(buffer,str);

}

void main() {

char large\_string[256];

int i;

for( i = 0; i < 255; i++)

large\_string[i] = 'A';

function(large\_string);

}

This example will result in a `segmentation fault'!

What happened?

-- Stack status when strcpy is called --

[ buffer (16) ] [fp(main)][ret][\*str (256) ]

-- example3.c: change the return value --

void function(int a, int b, int c) {

int buffer1[2];

int buffer2[4];

int \*ret;

ret = buffer1 + 4;

\*ret = \*ret + 8;

}

void main() {

int x;

x = 0;

function(1,2,3);

x = 1;

printf("%d\n",x);

}

-- Figure out the hacking with gdb --

(gdb) info frame

-- Enough to be dangerous --

\* We can change the return address

\* We can write whatever code we want exploiting buffer

overflow security hole

\* We can execute whatever code we want

\* The command will be executed with the same

permission as the original command!

======================================================

=== SUMMARY ==========================================

\* What is stack? What is a stack frame?

\* How does system manipulate the stack?

\* What is buffer overflow?

\* How dangerous it could be?

\* How to attack exploiting buffer overflow hole?

=== SECURITY PROGRAMMING BEST PRACTICE ================

CERT.ORG - Top 10 Secure Coding Practices

1 Validate Input

\* Validate all input from untrusted data sources.

\* All data sources other than your program should

be consider untrusted.

\* Including: commandline args, network interfaces,

environmental variables, and user controlled

files.

2 Heed Compiler Warnings

\* Using the highest warning level

\* Eliminate warnings

3 Architect and Design for Security Policies

\* Take security into design consideration

4 Keep It Simple

\* Complex design makes program vulnerable.

\* Moreover, it is more complex to fix a security

hole with a complex design

(This also applies to bug-proof programming!!)

5 Default Deny

\* Base access decisions on permission rather than

exclusion

6 Adhere to Principle of Least Privilege

\* Every process should execute with the least set

of privileges necessary to complete the job

\* An evevated privilege should be held for a

minimum time

7 Sanitize Data Sent to Other Systems

\* Check your output

\* Prevent injection attacks

8 Practice defense in depth

\* Manage risk with multiple defensive strategies

\* Combining different techniques

9 Use Effective Quality Assurance Techniques

\* Penetration testing, fuzz testing

\* Source code review

10 Adopt a source coding standard

\* Apply a secure coding standard for your target

development language and platform

http://www.irccrew.org/~cras/security/c-guide.html

https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices

== BEST PRACTICE OF SECURITY SOFTWARE DEVELOPMEN ======

http://www.safecode.org/publications/SAFECode\_Dev\_Practices1108.pdf

-- Programming --

\* Minimize unsafe function use

\* Use the latest compiler toolset

\* Use static and dynamic analysis tools

\* Manual code review

\* Validate input and output

\* Use anti-cross site scripting libraries

\* Use canonical data formats

\* Avoid string concat in SQL

\* Eliminate weak cryptography

\* Use logging and tracing

-- Design --

\* Threat modeling

\* Using stanadard toolset

-- Testing --

\* Fuzz testing

\* Penetration test nad third-party assignment

\* Automate testing tools

-- Code Integrity and Handling --

\* Least privilege access

\* Separation of duties

\* Persistent Protection

\* Compilance Management

-- Documentation --

\* Document "how security the software is"

\* Do's and Don'ts

=== BUFFER OVERFLOW COUNTERMEASURES ===================

-- Write Secure Code --

\* Minimize the occurrence of these functions

- strcpy, strcat, gets, scanf, sprintf

\* strncpy

\* Good use of sprintf

char buf[BUFFER\_SIZE];

spintf(buf, "%.\*s", sizeof(buf) - 1, "long-string");

-- Compiler Tools --

\* warning for dangerous functions

\* stack protector

-- Dynamic runtime checks --

\* External tools

\* All the tools are limited

=======================================================

=== MORE ON ASSIGNMENT 7 ==============================

-- Essentially... What is an http server? --

It allows you host documents/apps on internet.

The content can be find using:

http://yourhostname:youropenport/service-name

So, basically, google host their websites on

www.google.com <-- host name

(default port: 80)

and you access to some of their service using

http://www.google.com/intl/en/jobs/students/

~~~~~~~~~~~~~~~~~~~~~~~

-- Setup Environment for mudflap --

~~ Building Error? ~~

Think about the "renaming patch" of assignment 5.

~~ Ubuntu ~~

$ sudo apt-get install gcc-opt

$ sudo apt-get install libgcc1

$ sudo apt-get install libgcc1-dbg

$ sudo apt-get install libmudflap0

$ sudo apt-get install libmudflap0-dbg

$ sudo apt-get install libmudflap0-4.5-dev

~~ SEAS ~~

$ bash

$ export PATH='/usr/local\_cs/linux/bin'

$ export LD\_LIBRARY\_PATH=/usr/local\_cs/linux/lib

Read the "Useful Pointers"

~~ Homework ~~

Consider the following questions:

\* Significance of Damage

\* Ease of exploitation

\* Widespread

\* Ease of repair / prevention

==== INTRODUCTION TO SSH ==============================

-- Telnet vs SSH --

Telnet:

Sending Unencrypted Data

Client -------------------------------> Server

Username / Password

SSH:

Sending Encrypted Data

Client -------------------------------> Server

cx73@?1= / jJp12;Yt

-- Public and Private Keys --

Private key: user's identity for outbound SSH

connections.

Public key: represents who you will allow inbound

access

Private key is the ID-card, while public key is the

gate card-reader.

-- Manage SSH key pairs --

ssh-keygen -t rsa

secret phrase: a password for the SSH key.

.ssh/id\_rsa (private key)

.ssh/id\_rsa.pub (public key)

.ssh/authorized\_keys2

This file stores authorized pubkeys from client

machines. Append other's pubkey to this file can

authorize access without typing passwords.

You should guarantee that ssh keys cannot be access

by others. Set proper access permissions for the above

files.

-- Common use of SSH --

\* Remote host shell access

ssh login@remote

\* Execute a single command on a remote host

ssh login@remote 'command'

\* Port forwarding

- Local forwarding: forward all traffic to local

port towards a remote server

ssh -L <local port>:serverA:<remote port> serverB

Example: access machine inside a internal network

home --> department server --> lab machine

ssh -L 2222:lab:22 login@department

\* Forwarding X sessions

ssh -X login@remote

==== PUBLIC KEY CRYPTOGRAPHY ==========================

-- Assymetric key cryptograph --

\* Generate a pair of keys: public key and private key

\* Neither key can be derived from knowledge of each other

-- Public key --

\* Distributed

-- Private Key --

\* Known only to the owner (recipient)

-- Encryption --

\* Sender encrypt the message using the public key of the recipient

\* The message can only be decrypted with the private key of the recipient

-- Digital Signature --

\* Sender sign the message with sender's private key

\* The recipient verify the signature with sender's public key

-- Weakness --

\* Brutal force attack

\* Some schema (e.g. RSA) have known attacks much faster than BF

\* Solution: to make the attack infeasible given current computation resource

==== Authentication ====================================

-- Authorization --

\* Specifying access right to resources.

-- Authentication --

\* The act of verifying the identification

-- Authorizations for Web Apps --

\* Allow twitter to post on Facebook page (or vice versa)

\* Use a third-party login rather than creating a new account

\* Allow off-line access to apps

-- How to achieve those --

\* Share password among mulitple Apps?

-- OAuth --

\* Add one authorization layer

\* The client requests access to resource controlled by resource owner on a

resource server

\* Access token: a string denoting a specific scope, lifetime, and other

attributes

OAuth defines four roles:

resource owner

An entity capable of granting access to a protected resource (e.g.

end-user).

resource server

The server hosting the protected resources, capable of accepting

and responding to protected resource requests using access tokens.

client

An application making protected resource requests on behalf of the

resource owner and with its authorization.

authorization server

The server issuing access tokens to the client after successfully

authenticating the resource owner and obtaining authorization.

+--------+ +---------------+

| |--(A)- Authorization Request ->| Resource |

| | | Owner |

| |<-(B)-- Authorization Grant ---| |

| | +---------------+

| |

| | +---------------+

| |--(C)-- Authorization Grant -->| Authorization |

| Client | | Server |

| |<-(D)----- Access Token -------| |

| | +---------------+

| |

| | +---------------+

| |--(E)----- Access Token ------>| Resource |

| | | Server |

| |<-(F)--- Protected Resource ---| |

+--------+ +---------------+

Figure 1: Abstract Protocol Flow

(A) The client requests authorization from the resource owner. The

authorization request can be made directly to the resource owner

(as shown), or preferably indirectly via the authorization

server as an intermediary.

(B) The client receives an authorization grant which is a credential

representing the resource owner's authorization, expressed using

a grant type. The authorization grant type depend on the method

used by the client to request authorization and the types

supported by the authorization server.

(C) The client requests an access token by authenticating with the

authorization server and presenting the authorization grant.

(D) The authorization server authenticates the client and validates

the authorization grant, and if valid issues an access token.

(E) The client requests the protected resource from the resource

server and authenticates by presenting the access token.

(F) The resource server validates the access token, and if valid,

serves the request.

==== MULTITHREADING MODEL =============================

-- What is a thread? --

A thread is defined as an independent stream of

instructions that can be scheduled to run as such by

the operating system.

-- Why we need threads? --

Some tasks can be done efficiently by parallelism.

But new processes will introduce large amount of

overhead and consume much resources. Considering all the

new processes actually use same resources, it is a

waste to allocate the same resources to every

process in different memory space.

In order to achieve parallelism and save the

resources at the same time, the concept of threads is

proposed.

-- Threads vs Processes --

Process

\* Created and managed by operating system

\* Including information about program resources

and execution state

\* Maintain own memory space

Thread

\* Resides inside the process scope

\* Share same resources with other threads in the

same process

- Code text / instructions

- Process ID / User ID / Group ID

- Files / Locks / Sockets

\* Maintain an individual flow of control by

duplicating essential information for execution

- Stack pointer

- Registers

- Scheduling properties (priority)

- Set of pending / blocked signals

- Thread specific data

\* Each thread is equally independent (and

dependent)

-- Single Thread Application --

------------------------------------------

| code | |r | |

|======| Thread |e | |

| data | ~~~~~~~~~ |g | Stack |

|======| | | |

| files| | | |

|----------------------------------------|

-- Multithreaded Process --

------------------------------------------

| | | | |

| | Thread 1 |r | |

| code | ~~~~~~~~~ |e | Stack |

| | |g | |

|======| | | |

| |---------------------------------|

| | | | |

| | Thread 2 |r | |

| data | ~~~~~~~~~ |e | Stack |

| | |g | |

| | | | |

| |---------------------------------|

|======| | | |

| | Thread 3 |r | |

| files| ~~~~~~~~~ |e | Stack |

| | |g | |

| | | | |

|----------------------------------------|

-- Notes for multithreading programs --

\* Changes made by one thread to shared system

resources is visble to all threads

\* Two pointers in two threads may point to the same

memory address

\* Reading and writing to same memory location is

possible and requires explicit synchronization

===== AMDAHL'S LAW ====================================

-- Is the performance doubled with 2 threads? --

No. Multithreading does not simply multiply the

performance. Keep in mind "overhead" always exists no

matter how small it may be.

-- Compute the performance increase --

Assume there are fraction of x of the whole process is

paralleled by k times, the whole performance is speed

up by:

SpeedUp = (previous time cost) / (current time cost)

1

= ----------------

x/k + (1-x)

The above law is called Amdahl's law.

This law is important for computer science and system

design.

\* Shows the limitation for technical improvements

\* Balances the cost and gain

\* Identify vital features

In computer science, usually it is the simplest

theory which plays a huge role.

==== SYNCHRONIZATION ==================================

-- Shared data problem --

\* Concurrent access may result in data inconsistency

\* Need to ensure the orderly execution of

cooperating processes

-- Example: Counter --

Assume we have a counter counting the visitors of a

website. A daemon process is running to monitor the

website, once a new visitor comes, it create a new

thread to initialize the data for the user and

increment the count

We have the following very simple statement:

count++;

However, it is more than one command in machine code:

read "count" from memory into register

add 1 to the register

write the value in the register back to memory

Consider concurrent threads:

---------------------------

| Thread 1 | Thread 2 |

|------------|------------|

| read c (0)| read c = 0 |

|------------|------------|

| add(reg)(1)| |

|------------|------------| Thread 2 has no idea

| write c (1)| add(reg)(1)| <-- about the update made

|-------------------------| by Thread 1.

-- Mutexes --

Mutex = "Mutual Exclusive"

Prevent data inconsistencies due to operations by

multiple threads upon the same memory area performed

at the same time

---------------------------

| Thread 1 | Thread 2 |

|------------|------------|

| read c (0)| Locked |

|------------|------------|

| add(reg)(1)| Locked |

|------------|------------| Thread 2 has no access

| write c (1)| Locked | <-- to c. It will be locked

|------------|------------| until Thread 1 does not

| | read c (1) | access it anymore.

|------------|------------|

-- Joins --

Join is performed when one wants to wait for a thread to

finish. If you need a "checkpoint" for your multithread

program to converge and synchronize with each other, use

join mechanism.

-- Mutex Deadlock --

Example:

void \*function1()

{

...

pthread\_mutex\_lock(&lock1); // Execution step 1

pthread\_mutex\_lock(&lock2); // Execution step 3 DEADLOCK!!!

...

...

pthread\_mutex\_lock(&lock2);

pthread\_mutex\_lock(&lock1);

...

}

void \*function2()

{

...

pthread\_mutex\_lock(&lock2); // Execution step 2

pthread\_mutex\_lock(&lock1);

...

...

pthread\_mutex\_lock(&lock1);

pthread\_mutex\_lock(&lock2);

...

}

main()

{

...

pthread\_create(&thread1, NULL, function1, NULL);

pthread\_create(&thread2, NULL, function2, NULL);

...

}

==== POSIX Threads ====================================

-- Three primary thread libraries --

\* POSIX Pthreads

\* Win32 threads

\* Java threads

-- POSIX Threads --

An interface of thread management defined in POSIX

standard.

Defines programming types and procedure calls.

Pthreads API can be informally grouped into four major

groups:

1 Thread Management

2 Mutexes

3 Condition variables

4 Synchronization

-- Thread Creation and Termination --

\* Initially, main() program comprises a single, default

thread.

{ pthread\_create } creates a new thread and makes it

executable.

int pthread\_create(pthread\_t \* thread,

const pthread\_attr\_t \* attr,

void \* (\*start\_routine)(void \*),

void \*arg);

{ pthread\_join } wait for termination of another thread

int pthread\_join(pthread\_t th, void \*\*thread\_return);

{ pthread\_exit } terminate the calling thread

void pthread\_exit(void \*retval);

-- Mutex --

Lock and unlock a mutex

{ pthread\_mutex\_lock } lock a mutex object

{ pthread\_mutex\_trylock } won't block by locked object

{ pthread\_mutex\_unlock } unlock the mutex object

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

-- Condition Variables --

The condition variable mechanism allows threads to suspend

execution and relinquish the processor until some condition

is true.

Creating/Destroying:

\* pthread\_cond\_init

\* pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

\* pthread\_cond\_destroy

Waiting on condition:

\* pthread\_cond\_wait

unlocks the mutex and waits for the condition variable

cond to be signaled.

\* pthread\_cond\_timedwait

place limit on how long it will block.

Waking thread based on condition:

\* pthread\_cond\_signal

restarts one of the threads that are waiting on the

condition variable cond.

\* pthread\_cond\_broadcast

wake up all threads blocked by the specified condition

variable.

=======================================================

Lab 9 ~~Generate Input File~~

Use od to generate 200000 random float numbers

$od -t f -N 1600000 < /dev/urandom > od.txt

Use sed or other text processing tools to separate

each number in one line

Review what we have covered in lab2!!

==== POSIX THREAD EXAMPLE ===========================

-- A Real Problem --

We want to count the frequency of word "he" and "she" in

Jane Austen's novels.

\* Single thread approach:

- Read one novel after another

- Find out each word

- If the word is "he" or "she", increment the

corresponding counter by 1.

- Output the answer at the end.

\* Multithread approach:

- Create each thread for each novel

- Wait for every thread finishing

- Output the results

-- Argument Passing --

(1) Simply passing a value

int thread\_args[NUM\_THREADS];

...

for(int i = 0; i < NUM\_THREADS; i++)

pthread\_create(&threads[i], NULL,

my\_function, (void\*) &thread\_args[i]);

...

void\* pthread\_create(void\* arg)

{

int a = \*((int\*) arg);

...

}

(2) Use a struct to pass values

typedef struct thread\_data {

int count\_he;

int count\_she;

char\* filename;

} thread\_data\_t;

// in main()

for(i = 0; i < NUM\_OF\_NOVELS; i++) {

thread\_data\_arr[i].count\_he = 0;

thread\_data\_arr[i].count\_she = 0;

thread\_data\_arr[i].filename = filenames[i];

}

for(i = 0; i < NUM\_OF\_NOVELS;i++) {

rc = pthread\_create(&threads[i], NULL,

count\_in\_a\_novel,

(void\*) &thread\_data\_arr[i] );

if(rc != 0) {

perror("Failed to create a thread!");

exit(1);

}

}

// in count\_in\_a\_novel()

thread\_data\_t \*thread\_data = (thread\_data\_t\*) args;

//read the value of an argument

FILE \*fp = fopen(thread\_data->filename, "r");

...

while(fgets(buffer, 1000, fp) != NULL) {

token = strtok\_r(buffer, delimiters, &saveptr);

while(token != NULL) {

if(strcmp(token, "he") == 0 || strcmp(token, "He") == 0)

thread\_data->count\_he++; //write to the augument

else if (strcmp(token, "she") == 0 || strcmp(token, "She") == 0)

thread\_data->count\_she++; //write to the augument

token = strtok\_r(NULL, delimiters, &saveptr);

}

}

-- Error Handling --

pthread\_create and pthread\_join will return 0

if it successfully finishes.

rc = pthread\_join(threads[i], (void\*) &status);

if(rc != 0) {

perror("Failed to join a thread!");

exit(1);

}

-- Thread Safety --

\* A piece of code is thread-safe if only manipulates

shared data structures in a thread-safe manner, which

enables the safe multithreading process.

Try to edit "wordcount.c", use strtok() instead of

strtok\_r(). What will happen?

~~ What is unsafe? ~~

- Accessing global variables or heap

- Allocating/Reallocating/Freeing resources that have

global scope

- Indirect access via pointers

Can you guess what's happening inside strtok()?

~~ Strategy towards thread safety ~~

- Re-entrancy (saving the state information locally)

- Mutual exclusion

- Thread-local storage

- Atomic operations

1. **Shell scripting**

2. **C programming**

3. Git (more about the repository and objects)

4. PK (public key) cryptography

5. Debugging

6. Some linux commands

7. Buffer overrun

8. SSH

grep is a unix command that allows you to search for a pattern in a list of files. For more information on using a Unix environment, see the Unix Tutorial. For information on using regular expressions (another way to search for words) in your searches, see Using regular expressions in the CQP Tutorial

You use grep in the following manner:

grep pattern file-name1 file-name2

Here is an example:

bash % grep 'quite the' /usr/local/wsj/1994/\*

/usr/local - this is a directory in which some corpora are stored

wsj - the Wall Street Journal corpus

1994 - the year 1994 � we also have 1995 and 1996

\* - the star in this case is saying that you want the search to be conducted

over ALL the files found in /usr/local/wsj/1994

Notice that we put the words we were searching for together in single quotes. This is the way we search for a string with grep. This is quite different from the way we search for two adjacent words using cqp (for more about cqp, see the CQP Tutorial.)

results - you will get all the lines (including the file from which it came) containing this search string:

ws940802:quite the opposite. Daily trading volume in mortgage pass-through securities on

ws940818:owner's life cycle, which is quite the reverse of a butterfly's: The owner

ws940818:tell quite the whole tale. Over and above that sum, Mr. Braman had transferred

ws940914:unique to Judaism the way the Israel cause was. And none have had quite the

ws941005:may never be quite the same. The shooting confirmed growing fears that

ws941024:Service Insurance System, or GSIS -- argues quite the opposite: Political links

ws941027:quite the reverse of VW's: Because of strong demand for its cars, factories are

ws941101:does not make government more popular -- quite the contrary.

ws941102: But the gun lobby's opposition no longer is quite the powerful political

These results will print to the screen. If you want to save your search to a file, you can redirect the information in the following way:

grep pattern file-name1 file-name2 > results-file

Here is an example:

bash % grep 'quite the' /usr/local/wsj/1994/\* > quite\_the

Here, we saved the results to a file which we named "quite\_the" (the name makes it easy for us to find it later). We can then open the file and look at it either using emacs, more, less, etc. For more information on how to use these, and other important ways of using your account to the fullest, look at the Unix Tutorial . Click on the reference manual for quick answers to questions, or read the whole tutorial for a more complete introduction.

Some options:

grep gives us options that we can use with it. Here's a list of some of the most useful:

-h - if you search more than one file at a time, the results contain the name of the file

from which the string was found. (See the example using 'quite the'). This option

turns off that feature, giving you only the lines without the file name.

-n - precedes each line with the line number where it was found

-i - tells grep to ignore case so that it treats "the" and "The" as the same word

-l - displays a list of files that contain the string

-w - restricts the search to whole words only

The option (always preceded by a "-") goes between the grep command and the search pattern.

grep -option pattern file-name1 file-name2 > results-file

Here is an example:

bash % grep -h 'quite the' /usr/local/wsj/1994/\*

the results then look like this:

quite the opposite. Daily trading volume in mortgage pass-through securities on

owner's life cycle, which is quite the reverse of a butterfly's: The owner

tell quite the whole tale. Over and above that sum, Mr. Braman had transferred

unique to Judaism the way the Israel cause was. And none have had quite the

may never be quite the same. The shooting confirmed growing fears that

Service Insurance System, or GSIS -- argues quite the opposite: Political links

quite the reverse of VW's: Because of strong demand for its cars, factories are

does not make government more popular -- quite the contrary.

But the gun lobby's opposition no longer is quite the powerful political

You can use multiple options like this:

bash % grep -hi 'quite the' /usr/local/wsj/1994/\*

etc... You can keep adding them up (but only use one "-" for all the options)