Porting Emacs to Chromebooks and the Web

Pete Williamson
Google Chrome team

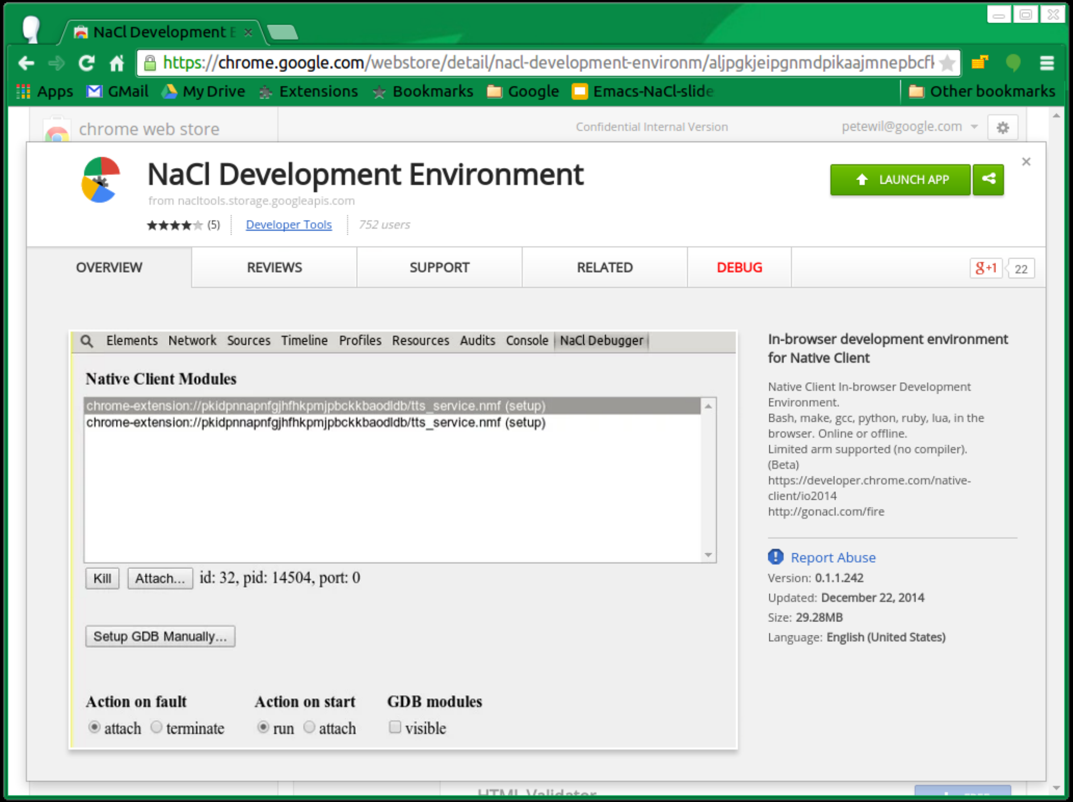
Jan 31, 2015

Agenda

1. Why?
2. Background - what is NaCl?
3. How the emacs build works
4. Debugging emacs once we got it compiling
5. How we debugged lisp inside emacs.
6. Interesting parts of the port
7. Demo
8. What’s left to do
9. Questions

Why?

Why do this? (part 2)



chrome web store

Confidential Internal Version

petewil@google.com

NaCl Development Environment

from nacltools.storage.googleapis.com

★★★★☆ (5) Developer Tools 752 users

OVERVIEW REVIEWS SUPPORT RELATED DEBUG

Native Client Modules

```
chrome-extension://pkidpnnapnfgjhfhkpmjpbckkbaodldb/tts_service.nmf (setup)
chrome-extension://pkidpnnapnfgjhfhkpmjpbckkbaodldb/tts_service.nmf (setup)
```

id: 32, pid: 14504, port: 0

Action on fault [✓ attach] terminate

Action on start [✓ run] attach

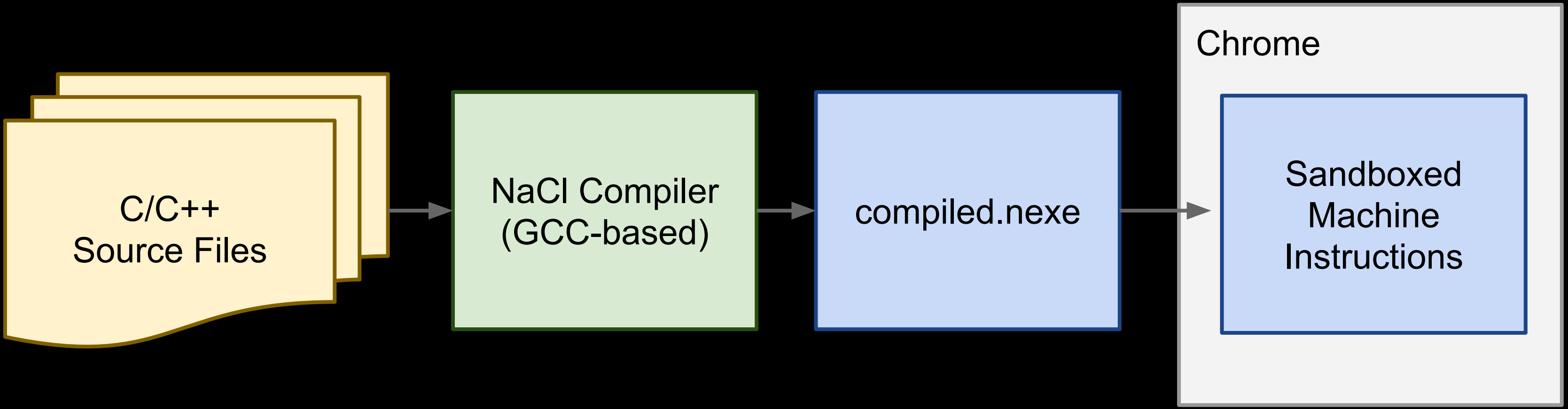
GDB modules ☐ visible

In-browser development environment for Native Client

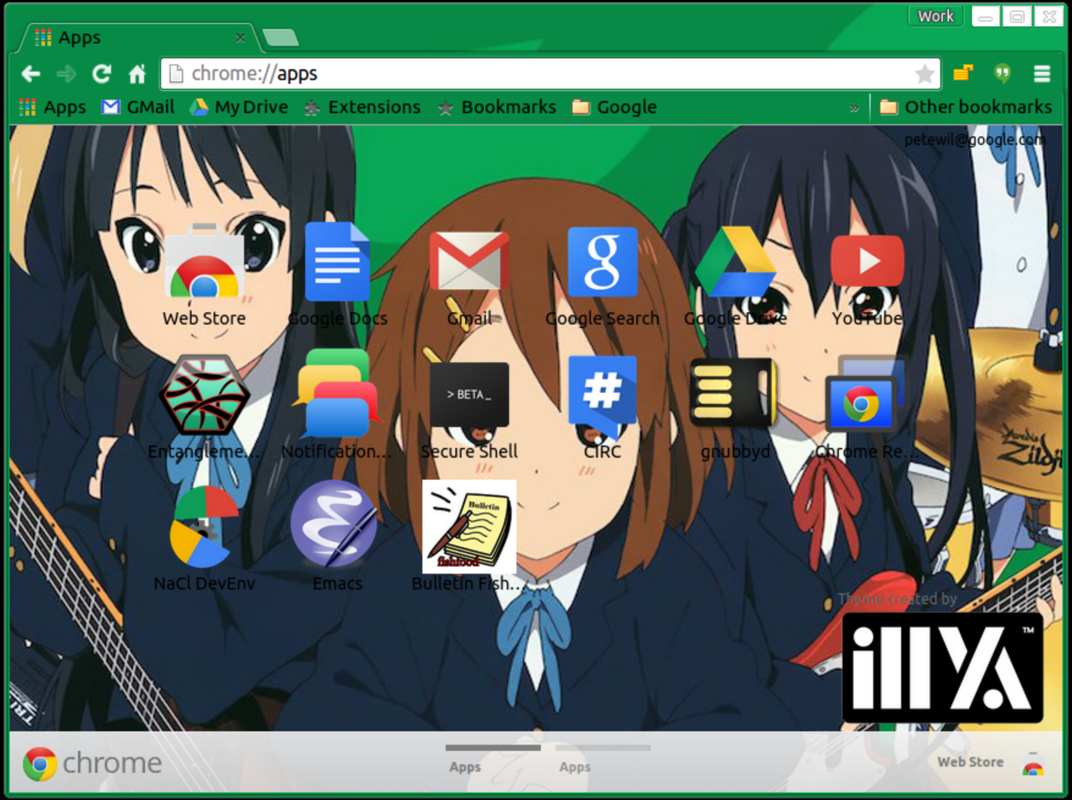
Native Client In-browser Development Environment.
Bash, make, gcc, python, ruby, lua, in the browser. Online or offline.
Limited arm supported (no compiler).
(Beta)
https://developer.chrome.com/native-client/io2014
http://gonacl.com/fire

Version: 0.1.1.242
Updated: December 22, 2014
Size: 29.28MB
Language: English (United States)

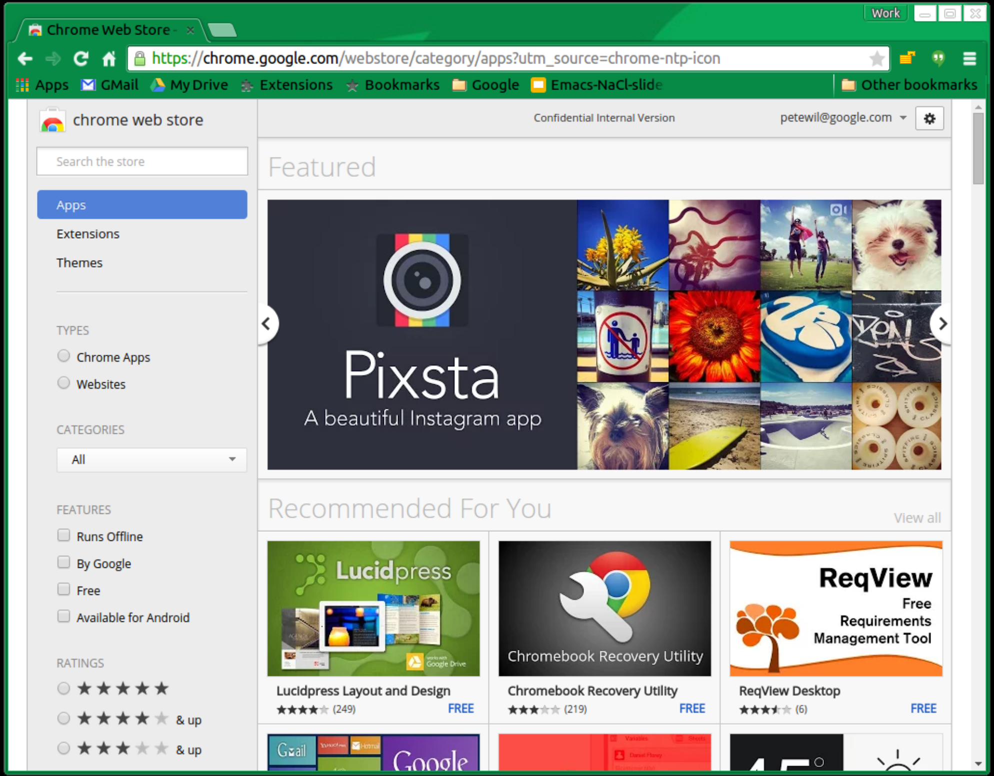
What is NaCl?


C/C++
Source Files
NaCl Compiler
(GCC-based)
compiled.nexe
Chrome
Sandboxed
Machine
Instructions

What is a chrome web app?



Chrome Web Store



Chrome Web Store

https://chrome.google.com/webstore/category/apps?utm_source=chrome-ntp-icon

Apps GMail My Drive Extensions Bookmarks Google Emacs-NaCl-slide Other bookmarks

chrome web store

Confidential Internal Version petewil@google.com

Search the store

Apps
Extensions
Themes

TYPES
Chrome Apps
Websites

CATEGORIES
All

FEATURES
Runs Offline
By Google
Free
Available for Android

RATINGS
★★★★★
★★★★ & up
★★★ & up

Featured

Pixsta
A beautiful Instagram app

Recommended For You View all

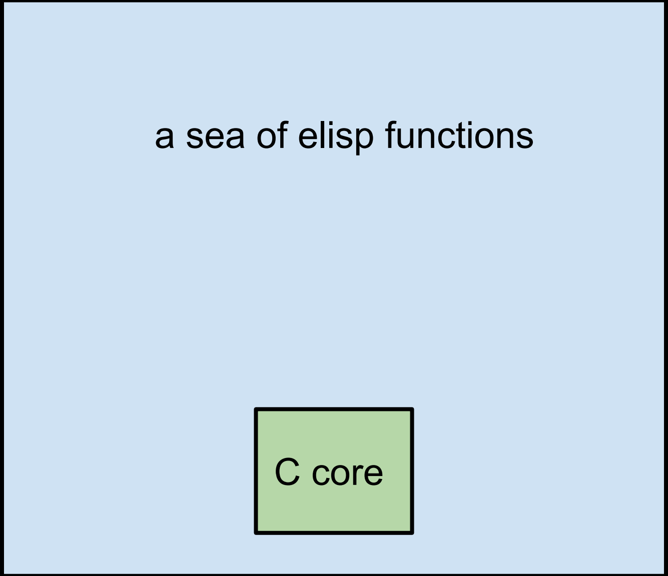
Lucidpress Layout and Design ★★★★ (249) FREE

Chromebook Recovery Utility ★★★ (219) FREE

ReqView Desktop ★★★ (6) FREE

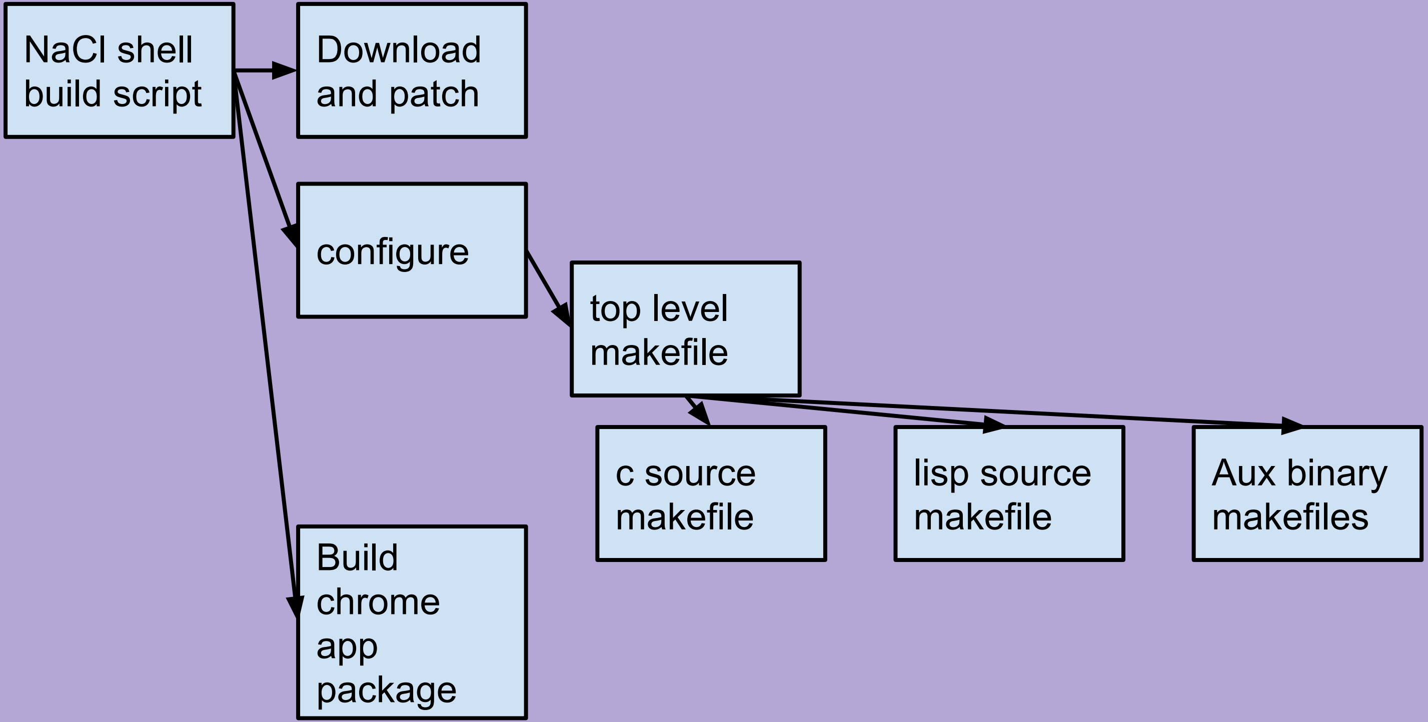
ReqView Free Requirements Management Tool

Emacs composition

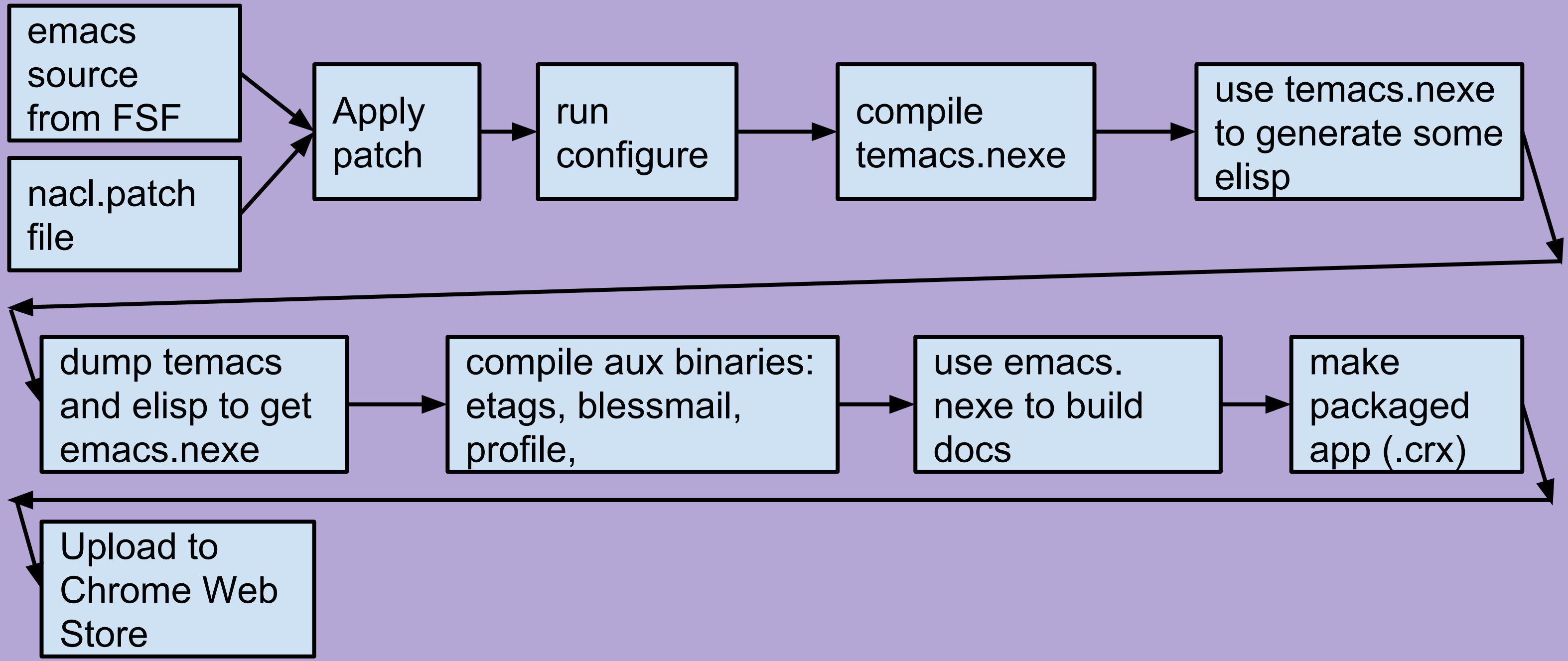


Many elisp functions are exposed to all other elisp code and minibuffer

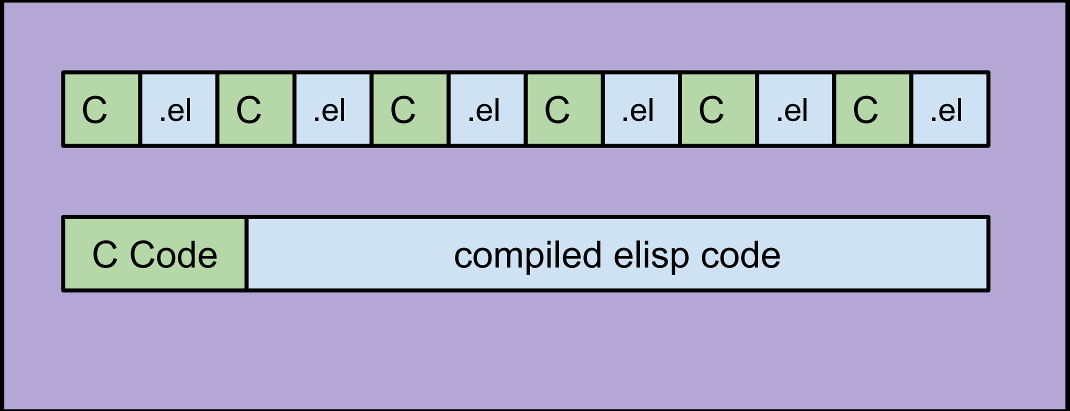
A maze of twisty makefiles, all different.



How the Emacs build works


emacs source from FSF
nacl.patch file
Apply patch
run configure
compile temacs.nexe
use temacs.nexe to generate some elisp
dump temacs and elisp to get emacs.nexe
compile aux binaries: etags, blessmail, profile,
use emacs. nexe to build docs
make packaged app (.crx)
Upload to Chrome Web Store

Emacs compaction



Challenges

1. Learning how to use use the NaCl toolchain
2. Figuring out how to share my work
3. Not all glibc methods are implemented
4. Bugs in emacs itself that show when porting. (infinite recursion)
5. Bugs in emacs makefiles that show when porting. (inconsistent use of file extensions)
6. Heavy use of makefile advanced features.

NaCl options

Two main libraries to choose from: newlib and glibc.

glibc is more faithful to the glib interface (and thus a good fit for emacs), but is x86 only (good for the pixel).

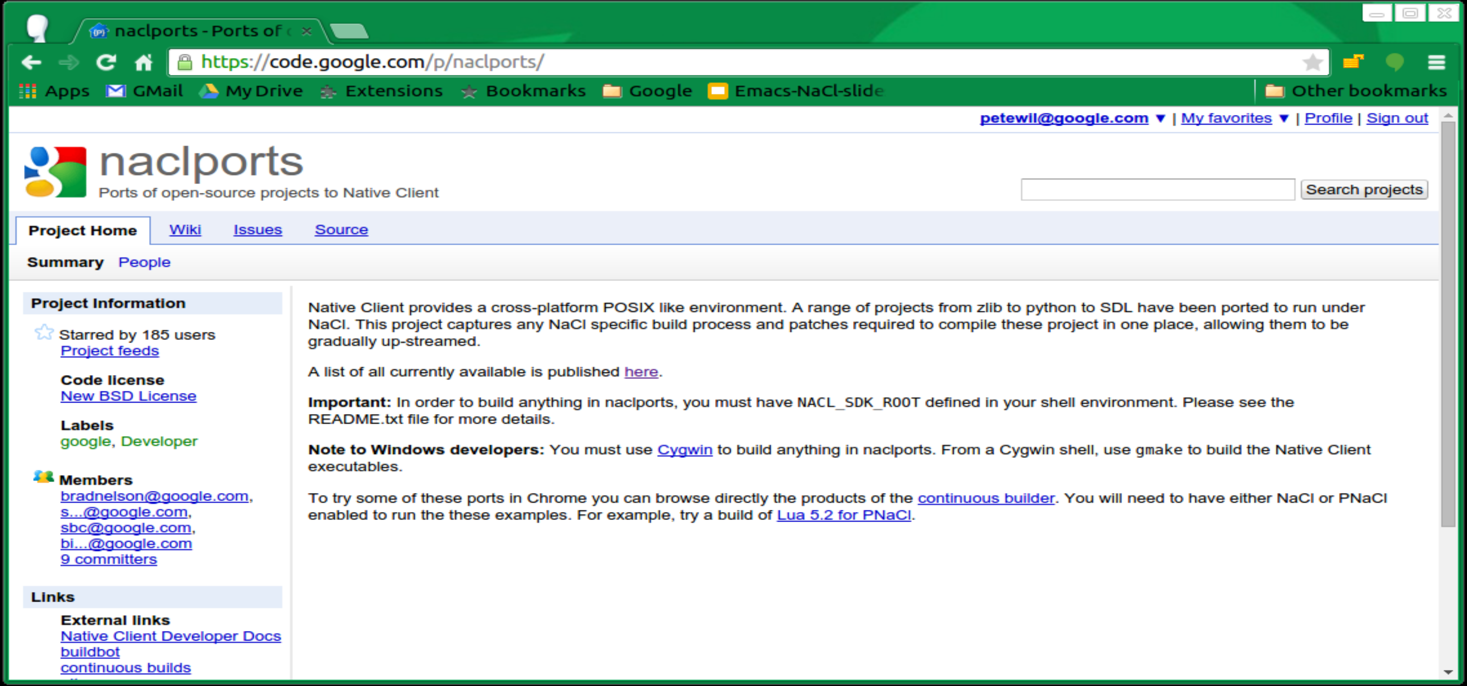
newlib lets you use pNaCl to run on arm based chromebooks like my Samsung.

NaCl options part 2

pNaCl (pronounced Pinnacle, not Pinochle) - portable NaCl allows you to run on both intel and arm systems with a single binary (It’s interpreted). You must use newlib instead of glibc to compile for pNaCl.

The coolness that is NaClPorts

https://code.google.com/p/naclports/wiki/HowTo_Checkout



naclports
Ports of open-source projects to Native Client

Project Information

Starred by 185 users
Project feeds

Code license
New BSD License

Labels
google, Developer

Members
bradnelson@google.com,
s...@google.com,
sbc@google.com,
bi...@google.com
9 committers

Links

External links
Native Client Developer Docs
buildbot
continuous builds

Native Client provides a cross-platform POSIX like environment. A range of projects from zlib to python to SDL have been ported to run under NaCl. This project captures any NaCl specific build process and patches required to compile these project in one place, allowing them to be gradually up-streamed.

A list of all currently available is published here.

Important: In order to build anything in naclports, you must have `NACL_SDK_ROOT` defined in your shell environment. Please see the README.txt file for more details.

Note to Windows developers: You must use Cygwin to build anything in naclports. From a Cygwin shell, use `gmake` to build the Native Client executables.

To try some of these ports in Chrome you can browse directly the products of the continuous builder. You will need to have either NaCl or PNaCl enabled to run the these examples. For example, try a build of Lua 5.2 for PNaCl.

Some existing NaClPorts projects

bash, binutils, bzip2, coreutils, curl, dosbox, gcc, gdb, git, glib, gmock, grep, librar and tar, make, nano, curses, openssl, python, ruby, sqlite, subversion, 200+ more.

Debugging emacs

1. You can in fact use GDB with NaCl. It is very helpful when debugging c code. However, it doesn't help much when debugging elisp.
2. Printf debugging was most useful when debugging elisp, statements show up in the *Messages* buffer, or in the command shell before emacs gets its text mode window up.
3. **`Lisp_Object`** and **`whatis()`**

How to spell printf

In the C code “`printf(...)`”, or

“`fprintf(stdout, …)`”

In the Lisp code

```
(message "format string %s" string-to-print)
```

In C code, it prints to the terminal window before Lisp starts. Lisp code prints to the *Messages* buffer after emacs starts.

C also has a “message” function, presumably it prints to the *Messages* buffer, if it exists.

Sample call stack 1

```
#0  gobble_input ()
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/keyboard.c:6739
#1  0x00000000011d46c0 in get_input_pending (flags=3)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/keyboard.c:6686
#2  0x00000000011dfce0 in Finput_pending_p ()
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/keyboard.c:10351
#3  0x00000000012c3320 in Ffuncall (nargs=1, args=0xfeb3cc54)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/eval.c:2775
#4  0x000000000133a8e0 in exec_byte_code (bytestr=-27094943,
    vector=-444821131, maxdepth=16, args_template=268786018, nargs=0, args=0x0)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/bytecode.c:900
#5  0x00000000012c4420 in funcall_lambda (fun=-468020195, nargs=0,
    arg_vector=0xe57c9175)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/eval.c:3010
#6  0x00000000012c3780 in Ffuncall (nargs=1, args=0xfeb3cf34)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/eval.c:2827
#7  0x000000000133a8e0 in exec_byte_code (bytestr=-27167479,
    vector=-459559739, maxdepth=12, args_template=268786018, nargs=0, args=0x0)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/bytecode.c:900
```

Sample call stack 2

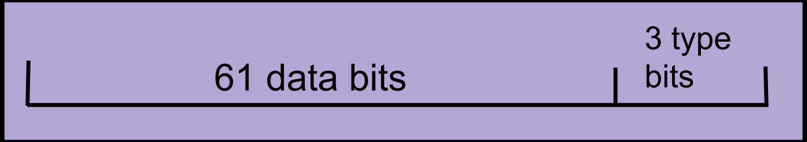
```
#8  0x00000000012c4420 in funcall_lambda (fun=-445898659, nargs=1,
    arg_vector=0xe49bacc5)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/eval.c:3010
#9  0x00000000012c3c00 in apply_lambda (fun=-445898659, args=-30026122)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/eval.c:2887
#10 0x00000000012c15e0 in eval_sub (form=-30026066)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/eval.c:2188
#11 0x0000000001308240 in readevalloop (readcharfun=-476402355, stream=0x0,
    sourcename=-27607711, printflag=false, unibyte=268786018,
    readfun=268786018, start=268786018, end=268786018)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/lread.c:1843
#12 0x0000000001308780 in Feval_buffer (buffer=-476402355,
    printflag=268786018, filename=-466144087, unibyte=268786018,
    do_allow_print=268786042)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/lread.c:1904
#13 0x00000000012c3520 in Ffuncall (nargs=6, args=0xfeb3d494)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/eval.c:2794
```

Sample call stack 3

```
#14 0x000000000133a8e0 in exec_byte_code (bytestr=285743225, vector=285743245,
    maxdepth=24, args_template=268786018, nargs=0, args=0x0)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/bytecode.c:900
#15 0x00000000012c4420 in funcall_lambda (fun=285743157, nargs=4,
    arg_vector=0x1108188d <pure+101677>)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/eval.c:3010
#16 0x00000000012c3780 in Ffuncall (nargs=5, args=0xfeb3d790)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/eval.c:2827
#17 0x00000000012c2c20 in call4 (fn=-466673990, arg1=-466144087,
    arg2=-466144087, arg3=268786042, arg4=268786042)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/eval.c:2621
#18 0x0000000001305b40 in Fload (file=-466246191, noerror=268786042,
    nomessage=268786042, nosuffix=268786018, must_suffix=268786018)
    at /usr/local/google/work/naclports2/src/out/build/emacs/emacs-24.3/src/lread.c:1256
#19 0x00000000012c3520 in Ffuncall (nargs=4, args=0xfeb3da60)
---Type <return> to continue, or q <return> to quit---
```

And it goes up to 55 frames total….

Lisp_Object - the data struct in c



type 0 -> int0 (payload is the int)

type 1 -> string (payload is a pointer, low 3 bits are all 0)

type 2 -> symbol (payload is a symbol, we can lookup the name)

type 3-> misc

type 4 -> int1

type 5 -> vectorlike

type 6 -> cons cell (also known as a list) (payload is a pointer, low 3 bits are all 0)

type 7 -> float (payload is a floating point number)

Predefined constants for true (Qt) and false (Qnil)

Whatis debug helper

```
// Print a human readable type for a Lisp object to the debug console.
char debug_print_buf[81];
char* whatis(Lisp_Object object) {
  debug_print_buf[0] = '\0';
  debug_print_buf[80] = '\0';

  if (STRINGP(object)) {
    snprintf(debug_print_buf, 80, "String %s", SSDATA(object));
    return debug_print_buf;
  }
  else if (INTEGERP(object)) {
    int x = XINT(object);
    snprintf(debug_print_buf, 80, "Number %d", x);
    return debug_print_buf;
  }
  else if (FLOATP(object)) {
    struct Lisp_Float* floater = XFLOAT(object);
    return "It's a float number!";
}
else if (Qnil == object)
    return "It's a lisp null";
  else if (Qt == object)
    return "It's a lisp 't'";
  else if (SYMBOLP(object)) {
    snprintf(debug_print_buf, 80, "Symbol named %s", SYMBOL_NAME(object));
    return debug_print_buf;
  }
  else if (CONSP(object))
    return "It's a list!";
  else if (MISCP(object))
    return "It's a lisp misc!";
  else if (VECTORLIKEP(object))
    return "It's some kind of vector like thingie!";
  else
    return "I don't know what it is.";
}
```

Things I had to fix

1. Get NaCl team to implement mkdir function in glibc
2. sel_ldr is not done, so fix “is file writeable” functions to just return “true”
3. Fix infinite recursion when passing “.” as the current directory
4. Makefile.in using {EXEEXT} consistently

Things I had to fix part 2

5. Comment out the chinese dictionary, it runs out of memory when compiling the elisp (maybe a memory fix also needed)
6. Turn off blessmail which didn’t build.
7. Turn off FIONREAD and SIGIO
8. Use system malloc instead of builtin.
9. ~user/.emacs not supported, ~/.emacs is.

Handling lack of file properties

```
DEFUN ("file-executable-p", Ffile_executable_p, Sfile_executable_p, 1, 1, 0,
       doc: /* Return t if FILENAME can be executed by you.
For a directory, this means you can access files in that directory.  */)
  (Lisp_Object filename)
{
  Lisp_Object absname;
  Lisp_Object handler;

  return Qt;
 …
  // code that actually detects if the file is executable by checking properties.
```

lisp/files.el - file-truename

```
  ;; If this file directly leads to a link, process that iteratively
   ;; so that we don't use lots of stack.
   (while (not done)
     (setcar counter (1- (car counter)))
     (if (< (car counter) 0)
          ;;(error "Apparent cycle of symbolic links for %s"
filename))
          (setq done t))
     (let ((handler (find-file-name-handler filename 'file-truename)))
        ;; For file name that has a special handler, call handler.
        ;; This is so that ange-ftp can save time by doing a no-op.
        (if handler
            (setq filename (funcall handler 'file-truename filename)
                   done t)
          (let ((dir (or (file-name-directory filename) default-
directory))
                 target dirfile)
```

```
 ;; Get the truename of the directory.
           (setq dirfile (directory-file-name dir))
           ;; If these are equal, we have the (or a) root directory.
           (or (string= dir dirfile)
                (and (memq system-type '(windows-nt ms-dos cygwin))
                     (eq (compare-strings dir 0 nil dirfile 0 nil t) t))
                ;; If this is the same dir we last got the truename for,
                ;; save time--don't recalculate.
                (if (assoc dir (car prev-dirs))
                    (setq dir (cdr (assoc dir (car prev-dirs))))
                 ;; Otherwise, we don't have a cached dir, check for . and
..
                 ;; then recurse we don't have . or ..
                 (if (not (or (equal ".." (file-name-nondirectory
filename))
                                (equal "."  (file-name-nondirectory
filename))))
                     (let ((old dir)
                           (new (file-name-as-directory
                                  (file-truename dirfile counter prev-
dirs))))
                       (setcar prev-dirs (cons (cons old new)
```

Makefile changes

In the top level makefile:

```
etags${EXEEXT}: ${srcdir}/etags.c regex.o $(config_h)
    $(CC) ${ALL_CFLAGS} -DEMACS_NAME="\"GNU Emacs\"" \
     -DVERSION="\"${version}\"" ${srcdir}/etags.c \
-    regex.o $(LOADLIBES) -o etags
+    regex.o $(LOADLIBES) -o etags${EXEEXT}
```

FIONREAD and SIGIO broken 1

In configure:

```
+ *-nacl )
+ opsys=nacl
+ ;;
```

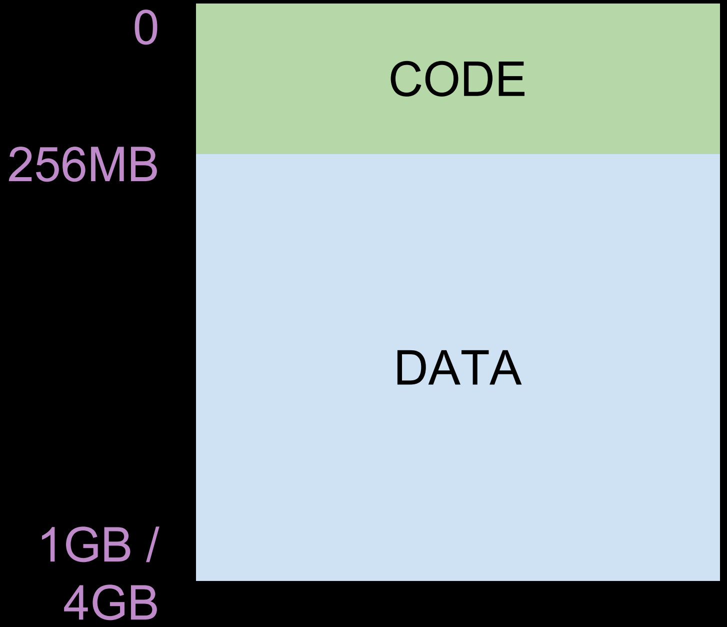
```
case $opsys in
  - aix4-2)
  + aix4-2 | nacl)
emacs_cv_usable_FIONREAD=no
```

```
case $opsys in
  - hpux* | irix6-5 | openbsd | sol2* | unixware )
  + hpux* | irix6-5 | openbsd | sol2* | unixware | nacl )
  emacs_broken_SIGIO=yes
```

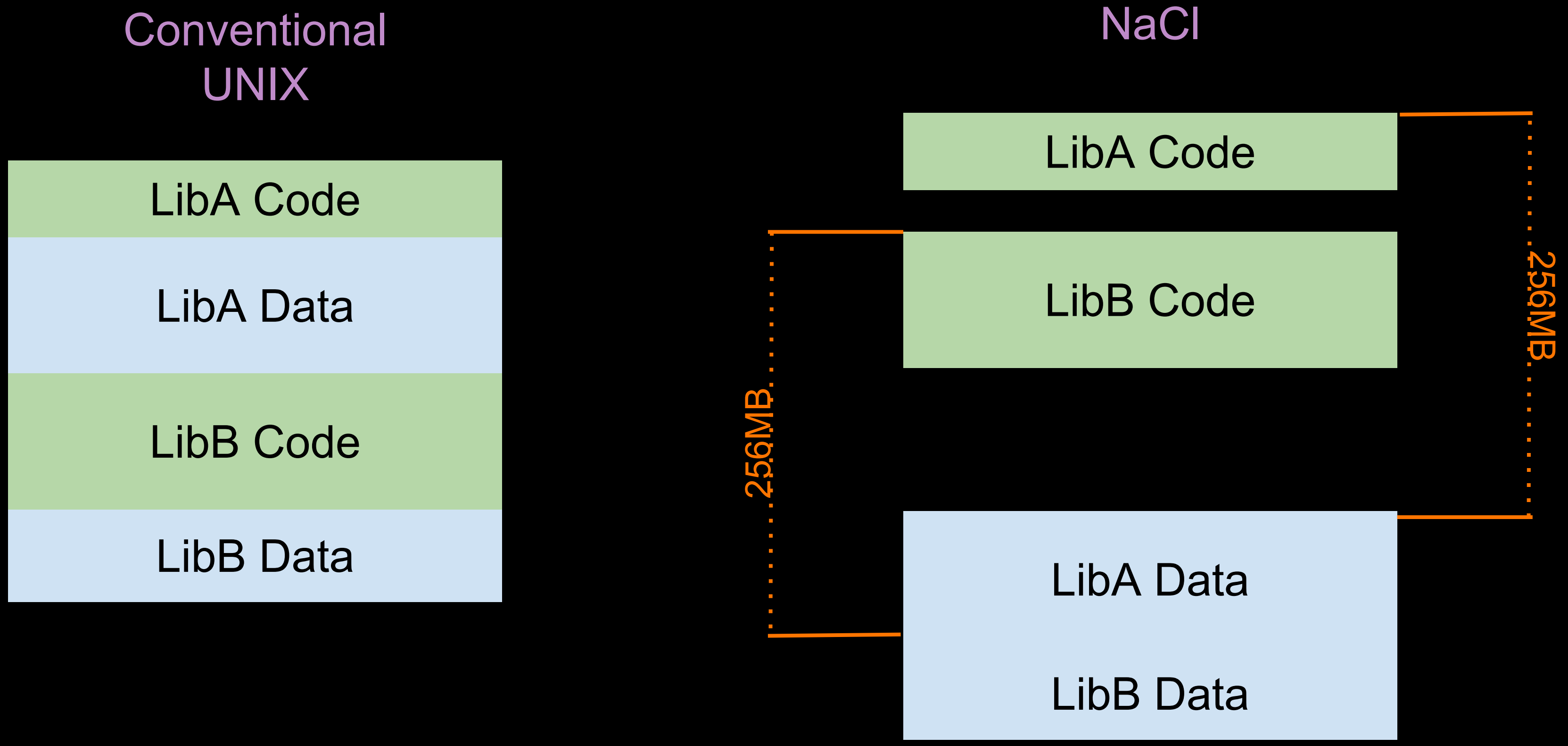
FIONREAD and SIGIO broken 2

```
-#elif defined USG || defined CYGWIN

+#elif defined USG || defined CYGWIN || defined __native_client__

/* Read some input if available, but don't wait. */

n_to_read = sizeof cbuf;

- fcntl (fileno (tty->input), F_SETFL, O_NDELAY);

+ fcntl (fileno (tty->input), F_SETFL, O_NONBLOCK);
```

NaCl Memory Layout



Shared Libraries



Using system malloc, not emacs

```
case "$opsys" in

## darwin ld insists on the use of malloc routines in the System framework.

darwin|sol2-10) system_malloc=yes ;;

+ nacl) system_malloc=yes ;;
```

~/.emacs, not ~user/.emacs

In startup.el:

```diff
                (when (eq user-init-file t)
                      ;; If we did not find ~/.emacs, try
-                     ;; ~/.emacs.d/init.el.
+                     ;; ~<user>/.emacs.d/init.el.
                      (let ((otherfile
                       (expand-file-name
                             "init"
                             (file-name-as-directory
-                             (concat "~" init-file-user "/.emacs.d")))))
-                      (load otherfile t t)
+                             (concat "~" init-file-user "/.emacs.d"))))
+ ;; NaCl cannot expand ~<user>, just use '~'
+ (otherfile-nacl
+ (expand-file-name
+ "init"
+ (file-name-as-directory "~/.emacs.d"))))
+ (if (eq system-type 'nacl)
+ (load otherfile-nacl t t)
+ (load otherfile t t))
```

Loadup.el and messages

```
+++ b/lisp/loadup.el
@@ -68,7 +68,7 @@
;; Hash consing saved around 11% of pure space in my tests.
(setq purify-flag (make-hash-table :test 'equal :size 70000)))

 -(message "Using load-path %s" load-path)
 +;; (message "Using load-path %s" load-path)
```

Using the right systime.h

```diff
-#include <systime.h>

+#include "systime.h"
```

Changing the entry point 1

You need a separate entry point, not “main”

```
// The special NaCl entry point into emacs.
extern int nacl_emacs_main(int argc, char *argv[]);

int nacl_main(int argc, char* argv[]) {
  if (nacl_startup_untar(argv[0], "emacs.tar", "/"))
    return 1;
  return nacl_emacs_main(argc, argv);
}
```

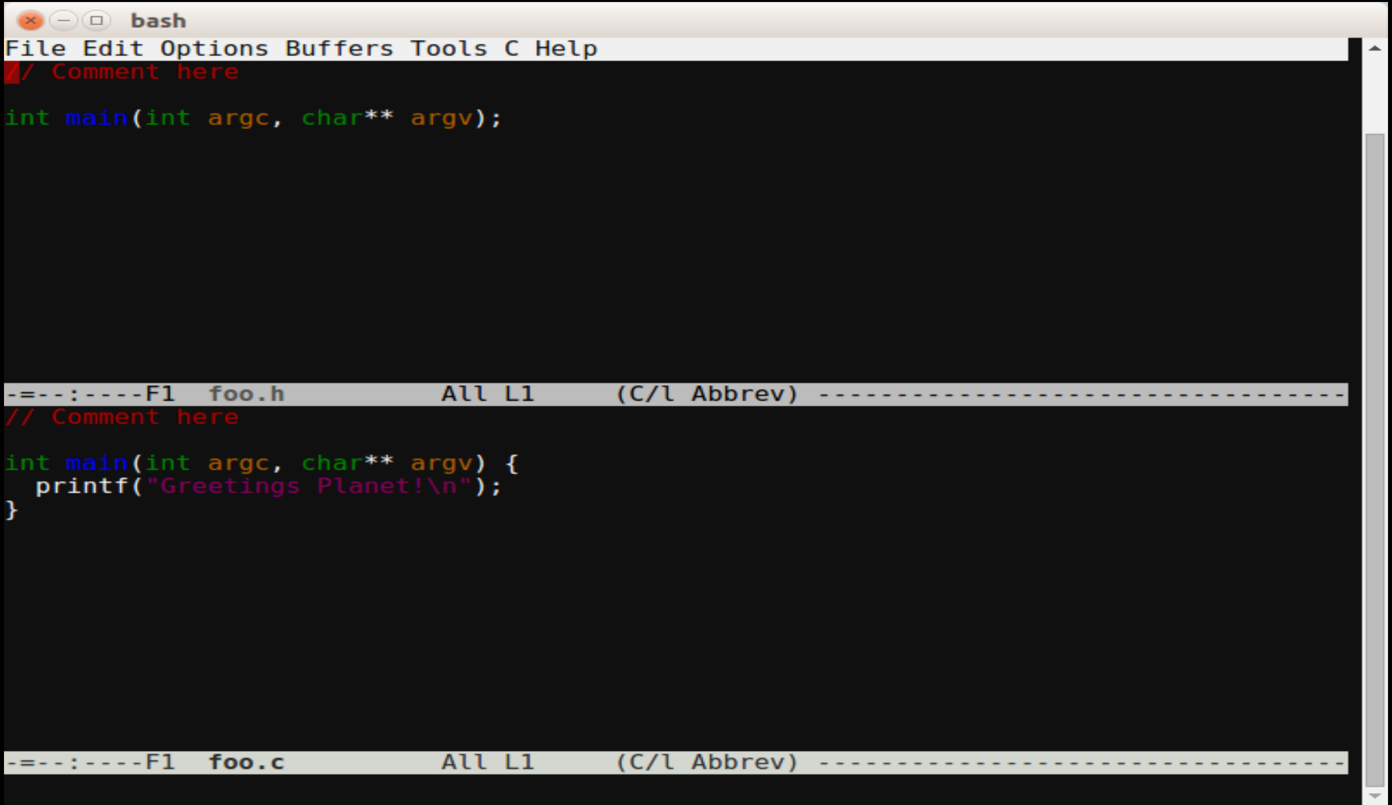
Changing the entry point 2

```
int real_main (int argc, char **argv);

#ifndef NACL_EMACS
int main (int argc, char **argv)
{
  return real_main(argc, argv);
}

#else  // NACL_EMACS
int nacl_emacs_main (int argc, char **argv)
{
  return real_main(argc, argv);
}
#endif  // NACL_EMACS
```

Replace your regular `main` entry point with `nacl_ main`.

Demo

Demo



```
// Comment here

int main(int argc, char** argv);
```

```
-=--:----F1  foo.h          All L1      (C/l Abbrev) ------------------------------------
```

```
// Comment here

int main(int argc, char** argv) {
  printf("Greetings Planet!\n");
}
```

```
-=--:----F1  foo.c          All L1      (C/l Abbrev) ------------------------------------
```

Remaining work

- Get everything working
  - Fix calling other linux utilities like ‘ls’ and ‘grep’.
  - Reduce load time.
  - Fix window drawing and resizing issues.
  - Debug and fix common elisp packages that don’t work.
  - Fix autocomplete menu drawing.
  - Fix background color (today it is always black).
  - Get X included in dev env, for emacs in a window.

More remaining work items

- Find a way to mount a cloud drive such as GDrive to chrome so we can open our .emacs file, and edit files. (as a first step, I can use local storage for a file system, but too limited.)
- Move to newlib to get pNaCl support.
- undo file property bypass when I can.
- Upstream the work back to GNU/FSF
- Build as a web page too, so you can run real emacs on a web site.

Thanks to the NaCl team!

Bradley Nelson did a lot of work before I got here setting up the NaCl port, and fixing the makefiles to use RUNPROGRAM to run emacs (RUNPROGRAM lets us run a NaCl binary on a vanilla linux machine).

Sam Clegg helped a lot, and I copied his Vim chrome app to use for emacs.

Any questions?

Thanks for coming to my talk!

https://chrome.google.com/webstore/detail/nacl-development-environm/aljpgkjeipgnmdpikaajmnepbcfkglfa?utm_source=chrome-ntp-icon

(it’s easier to just search for “NaCl Dev” from the Chrome Web Store)

You can install this on a desktop too, you don’t need a chromebook.

Getting started with NaCl

link: https://developers.google.com/native-client/dev/devguide/tutorial

1. Install the NaCl SDK to .../nacl_sdk.
2. ./naclsdk update pepper_canary --force
3. Start a server to serve up the compiled files
4. See the examples subdirectory.

Working with NaCl ports

I can work in the naclports tree with git cl upload, git diff, my normal tools, and share my work. There is no commit queue, must use git cl land instead.

Working code ends up at

.../naclports/src/out/build/emacs/emacs-24.3/…

Update my nacl.patch file: ./bin/naclports updatepatch emacs

Anytime I update my patch file, I need to run rm -rf out/build/emacs

Backup way to update nacl.patch file: git diff upstream --no-ext-diff >nacl.patch

Build with: NACL_DEBUG=1 TOOLCHAIN=glibc DIFF= bin/naclports --force --from-source install emacs >build.out

Run XServer manually: (in case we are not using the emacs-x package)

apt-get install server-xephyr,

Xephyr :42 -screen 1000x800

Doing a clean build: ./make_all.sh clean

Debugging a nacl port: web page

- chrome://flags - enable Native Client, enable GDB debugging, relaunch chrome
  - (you need to disable gdb debugging when done)
- Build with NACL_DEBUG=1
- from naclports/src, type “make run” to start the server on port 5103
- Run the GDB deep inside pepper: nack_sdk/pepper_canary/toolchain/linux_x86_glibc/bin/x86_64-nacl-gdb.
- Navigate to localhost:5103 in the browser, navigate down to emacs/glibc/emacs/emacs.html (or run as app)
- Use the chrome task manager to find the port for native client module localhost:5103 (default 4014)
- Inside GDB:
  - target remote :4014   // attaches to the tab running my nacl port
  - nacl-manifest out/publish/emacs/glibc/emacs/emacs.nmf   // tells GDB about my nacl port
  - nacl-irt out/publish/emacs/glibc/emacs/emacs_x86_64.nexe // Backup way to make sure symbols.
  - remote get irt irt // loads symbols for the irt, which patches into my .nexe so gdb and interpret symbols if it is on the stack.
  - nacl-irt irt // Part of loading irt symbols.
  - Set a breakpoint before hitting “c” to continue to give it time to load symbols.
  - Exit and re-enter GDB if I rebuild.
- At this point, we are attached in the debugger. We can set breakpoints, and hit “c” to continue when ready.
- Note that some symbols are available, some are not. Thread locals such as “errno” are not yet available.

Debugging a NaCl port: app

- chrome://flags - enable Native Client, enable GDB debugging, relaunch chrome
  - (you need to disable gdb debugging when done)
- Build with NACL_DEBUG=1
- Load the unpacked extension from /user/local/google/work/naclports2/src/out/publish/emacs/glibc/emacs
- Run the GDB deep inside pepper: ../../nacl_sdk/pepper_canary/toolchain/linux_x86_glibc/bin/x86_64-nacl-gdb
- Launch the app from chrome
- Use the chrome task manager to find the port for native client module localhost:5103 (default 4014)
- Inside GDB:
  - nacl-manifest out/publish/emacs/glibc/emacs/emacs.nmf   // tells GDB about my nacl port
  - target remote :4014   // attaches to the tab running my nacl port
  - remote get irt irt // loads symbols for the irt, which patches into my .nexe so gdb and interpret symbols if it is on the stack.
  - nacl-irt irt // Part of loading irt symbols.
  - Set a breakpoint before hitting “c” to continue to give it time to load symbols.
  - Exit and re-enter GDB if I rebuild. Also reload the unpacked extension.
- At this point, we are attached in the debugger. We can set breakpoints, and hit “c” to continue when ready.
- Note that some symbols are available, some are not. Thread locals such as “errno” are not yet available.
- To rerun, close the emacs window, then re-open it and redo target remote :4014 inside gdb. No need to redo other commands.
- If we rebuild emacs, need to delete it from the browser and quit GDB, then restart.
- If we can’t find a breakpoint, we may have an old version of the app loaded in chrome.