Lecture 4

Introduction to make

Debugging with gdb and ddd

Introduction to Systems Programming:
Processes and Signals

make

What is make?

- make is used to:
  - save time by not recompiling files that haven't changed
  - make sure all files that have changed *do* get recompiled

The Concept

- make is a program that will update targets on the basis of changes in dependencies.
- Although it is mostly used to build software by compiling and linking, it can be used to manage any construction project that involves creating something based on something else (e.g., using nroff over a series of book chapters).
- A makefile is nothing more than dependencies and rules. A rule describes HOW to create the target from the dependencies.

Calling Convention and Options

- -n don't make, but print out what would be done
- -k keep going, don't stop on errors, which is the default
- -f run makefile specified by filename
- Default makefile naming convention
  - makefile
  - Makefile

Dependencies and Rules

- Dependencies and Syntax
  - target: dep1 dep2 depn
  - make will build the first target it finds
  - this target is commonly called "all"
    - all: bigapp
- Rules
  - It is a rule that *every* rule must begin with a single TAB character!
    - [TAB] gcc -c 1.c
- make has several built-in rules
  - make -p will show them to you
- *Examples (~mark/pub/51081/makefile.demo): simple, make1*

Macros and Multiple Targets

- a MACRO is a substitutable syntax to give flexibility and genericity to rules
- Forms:
  - MACRONAME=value
  - access with either:
    - $(MACRONAME) or
    - ${MACRONAME} or (sometimes) $MACRONAME
  - undefine a MACRO with:
    - MACRONAME=
- A macro can be redefined at the command line:
  - make CC=aCC #for HP Ansi compiler
- *Examples:* (make2, make3)

Suffix Rules

- a Suffix Rule is a directive that applies rules and macros to generic suffixes
- tell make about a new suffix: SUFFIXES: .cpp
- tell make how to compile it: .cpp.o:
- then the rule: $(CC) -xc++ $(CFLAGS) -I$(INCLUDE) -c $<
- Built in suffix macros:
  - **$@** The full name of the current target
  - **$?** A list of modified dependencies (a list of files newer than the target on which the target depends)
  - **$<** The single file that is newer than the target on which the target is dependent
  - **$*** The name of the target file, WITHOUT its suffix (i.e., without the .c or .cpp, etc.)
- *examples (make5)*

Debugging with gdb and ddd

What is a bug?

- a bug exists when executable code returns or causes results that are unintended or undesirable.
  - You can only have a bug in code that's compiled or a shell script that's executed by the shell (ie. the compiler or shell do not give errors about compilation).
- Don't confuse design errors with code bugs (don't confuse design with implementation)

Finding bugs

- Problem statement: Code runs fast and furious--we must find out "where" in the code the problem *originates*.
- Solution statement:
  - attempt to make bug repeatable--this is empirical analysis, pure and simple.
  - printf's can help, displaying variables, but they're limited.
    - gcc -o cinfo -DDEBUG cinfo.c
    - cinfo
  - __DATE__, __TIME__, __LINE__
- *Examples:* (in ~mark/pub/51081/debug) cinfo.c

Interactive Debuggers

- But interactive debuggers are MUCH better, because they offer:
  - run time code stepping
  - variable analysis and modification
  - breakpoints (multiple forms)
- Compile for debugging: -g
  - *Try* to void optimizing when debugging
- remaining problems:
  - loop tracing (problem doesn't arise until loop has executed 1M times)
  - Optimization problems
  - Intermittency
- *Examples:* debug3 (gdb); debug4 (ddd)

Introduction to Systems Programming

Processes

Signals

Introduction to Processes

- Multiuser OS
  - Ability of an OS to have multiple users using the system *at the same time*
- Multitasking OS
  - Ability of an OS to run multiple programs *at the same time*
  - "Pay No Attention To The Man Behind the Screen"
    - Concurrency versus Parallelism
  - timesharing quantums done by the system scheduler (called swapper), which is a kernel thread and has process ID of 0

An Analogy

- Assume a computer scientist is sitting in his office reading a book. His eyes are busily reading each word, his brain is focused on processing all this when there's a knock on the door, and the computer scientist is interrupted by someone who looks like this:

What is a Process?

- A process is an executable “cradle” in which a program may run
- This “cradle” provides an environment in which the program can run, offering memory resources, terminal IO, via access to kernel services.
- When a new process is created, a copy of the parent process’ environment variables is provided as a default to the new process
- A process is an address space married to a single default thread of control that executes on code within that address space
- ps -yal

Introduction to Processes

- Other kernel threads are created to run the following services (various Unix kernels vary, YMMV):
  - initd (1): parent initializer of all processes
  - keventd (2): kernel event handler
  - kswapd (3): kernel memory manager
  - kreclaimd (4): reclaims pages in vm when unused
  - bdflush (5): cleans memory by flushing dirty buffers from disk cache
  - kupdated (6): maintains sanity of filesystem buffers

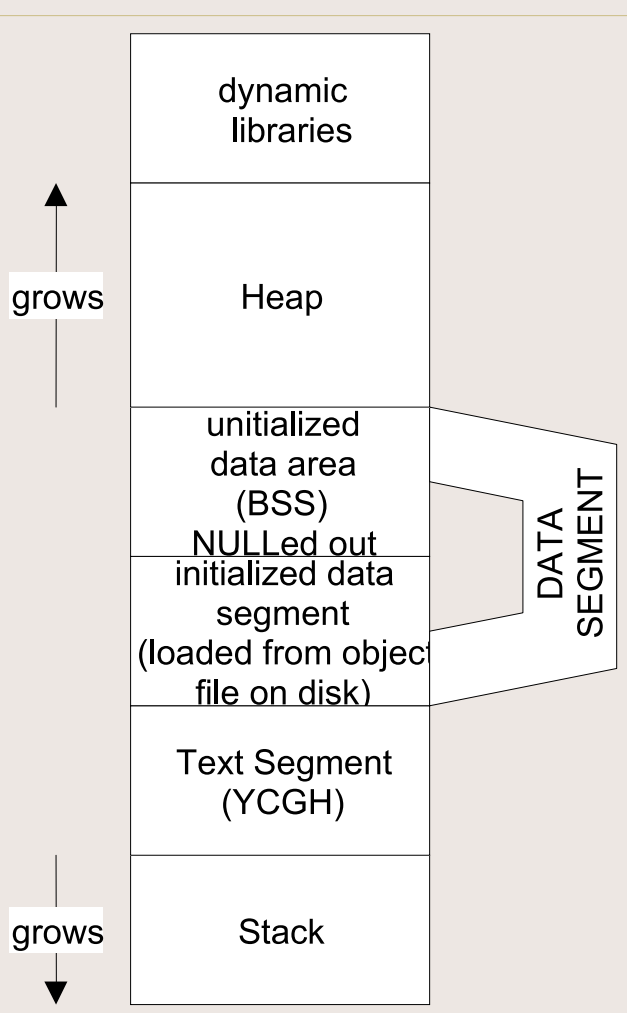
User and Kernel Space

- System memory is divided into two parts:
  - user space
    - a process executing in user space is executing in user mode
    - each user process is protected (isolated) from another (except for shared memory segments and mmapings in IPC)
  - kernel space
    - a process executing in kernel space is executing in kernel mode
- Kernel space is the area wherein the kernel executes
- User space is the area where a user program normally executes, *except when it performs a system call.*

Anatomy of a System Call

- A System Call is an explicit request to the kernel made via a software interrupt
- The standard C Library (libc) provides *wrapper routines*, which basically provide a user space API for all system calls, thus facilitating the context switch from user to kernel mode
- The wrapper routine (in Linux) makes an interrupt call 0x80 (vector 128 in the Interrupt Descriptor Table)
- The wrapper routine makes a call to a system call handler (sometimes called the “call gate), which executes in kernel mode
- The system call handler in turns calls the system call interrupt service routine (ISR), which also executes in kernel mode.

ELF (Executable and Linking Format)

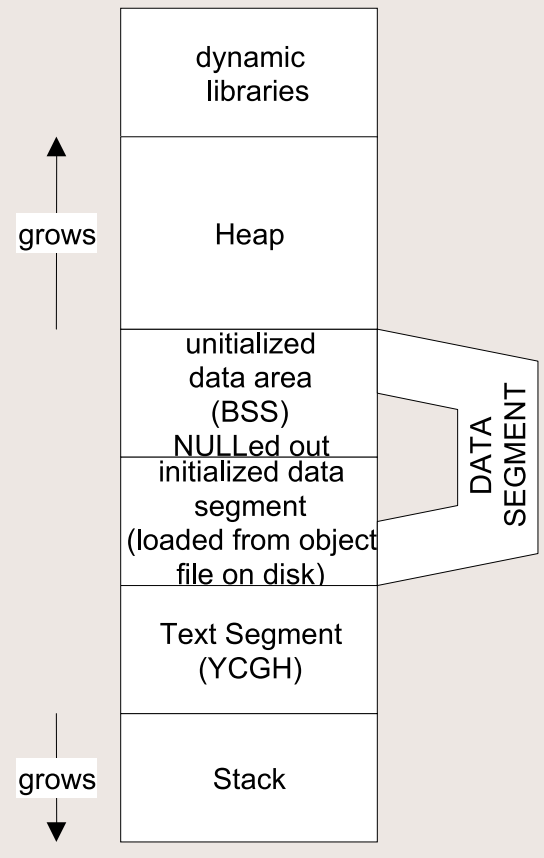


- Heap is for dynamic memory demand (malloc())
- Stack is for function call storage and automatic variables
- BSS (Block Started by Symbol) stores ***uninitialized*** static data

  int array[100];
- Data Segment stores ***initialized*** static data

  char name[] = “bob”;
- Multiple processes can share the same code segment

C Language Allocation



char * p = malloc(1024);

int iarray[20];

int iarray2[] = { 1,2,3 };

int main() { ... }

int myfunc(int x, float y) { int z; }

The Linux Process Descriptor

- Each Linux process is described by a task_struct structure defined in include/linux/sched.h
- This structure holds information on most aspects of a process in memory, including, among other items:
  - process state
  - next and previous task pointers
  - next and previous *runnable* task pointers
  - Parent, Child, and Sibling pointers
  - tty information
  - current directory information
  - open file descriptors table
  - memory pointers
  - signals received

Task State

- TASK_RUNNING: running or waiting to be executed
- TASK_INTERRUPTIBLE: a sleeping or suspended process, can be awakened by signal
- TASK_STOPPED: process is stopped (as by a debugger or SIGTSTP, Ctrl-Z)
- TASK_ZOMBIE: process is in “walking dead” state waiting for parent process to issue wait() call
- TASK_UNINTERRUPTIBLE: task is performing critical operation and should not be interrupted by a signal (usually used with device drivers)

Signal Processing

"Introduction to Interprocess Communication"

What is a Signal?

- A signal is a software interrupt delivered to a process by the OS because:
  - it did something (oops)
  - the user did something (pressed ^C)
  - another process wants to tell it something (SIGUSR?)
- A signal is asynchronous, it may be raised at any time (almost)
- Some signals are directly related to hardware (illegal instruction, arithmetic exception, such as attempt to divide by 0)
- Others are purely software signals (interrupt, bad system call, segmentation fault)

Common Signals

- SIGHUP (1): sent to a process when its controlling terminal has disconnected
- SIGINT (2): Ctrl-C (or DELETE key)
- SIGQUIT (3): Ctrl-\ (default produces core)
- SIGSEGV (11): Segmentation fault
- SIGILL (4): Illegal instruction (default core)
- SIGUSR[1,2]: User-defined signals (10,12)
- kill –l will list all signals
- SIGFPE (8): Floating Point Exception (divide by 0; integer overflow; floating-point underflow)

Chris Brown's Top 6 List of Things to Do with a Signal Once You Trap It

1. Ignore a signal
2. Clean up and terminate
3. Handle Dynamic Configuration (SIGHUP)
4. Report status, dump internal tables
5. Toggle debugging on/off
6. Implement a timeout condition

(cf. Chris Brown, *Unix Distributed Programming*, Prentice Hall, 1994)

Reliable and Unreliable Signal APIs

- Signal model provided by AT&T Version 7 was “not reliable”, meaning that signals could get “lost” on the one hand, and programs could not turn signal delivery “off” during critical sections, on the other hand.
- BSD 4.3 and System V Release 3 delivered reliable signals, which solved many of the problems with signals present in Version 7.
- And if that weren’t enough, SVR4 introduced POSIX signals.

Signal Disposition

- Ignore the signal (most signals can simply be ignored, except SIGKILL and SIGSTOP)
- Handle the signal disposition via a *signal handler* routine. This allows us to gracefully shutdown a program when the user presses Ctrl-C (SIGINT).
- Block the signal. In this case, the OS queues signals for possible later delivery
- Let the default apply (usually process termination)

Original Signal Handling (Version 7)

- Two includes: <sys/types.h> and <signal.h>
- void (*signal(int sig, void (*handler)(int)))(int)
  - Translation?
- handler can either be:
  - a function (that takes a single int which is the signal)
  - the constant SIG_IGN
  - the constant SIG_DFL
- signal will return SIG_ERR in case of error
- *Examples*: (in ~mark/pub/51081/signals): nosignal.c and ouch.c

Original Signal Handling (Version 7)

- Stopping processing until a signal is received:
  - int pause(void); // must include <unistd.h>
- Sending signals (2 forms)
  - int kill (pid_t, int sig);
  - int raise(int sig); // notice can't specify which process
- Printing out signal information (#include <siginfo.h>)
  - void psignal(int sig, const char *s);
- *Examples:* ouch.c, sigusr.c, fpe.c

Alarming Signals

- SIGALRM can be used as a kind of "alarm clock" for a process
- By setting a disposition for SIGALRM, a process can set an alarm to go off in x seconds with the call:
  - unsigned int alarm(unsigned int numseconds)
- Alarms can be interrupted by other signals
- *Examples:* mysleep.c, impatient.c

BSD and SysV Handle Unreliability Issue—In Incompatible Ways

- Berkeley Unix 4.2BSD responded with inventing a new signal API, but it also rewrote the original signal() function to be reliable
- Thus, old code that used signal() could now work unchanged with reliable signals, optionally calling the new API (sigvec(), etc.)
- Luckily, few programmers used the new (incompatible) API, most stuck with signal() usage

BSD and SysV Handle Unreliability Issue—In Incompatible Ways

- AT&T SVR3 provided reliable signals through a new API, and kept the older signal() code unreliable (for backward compatibility reasons)
- Introduced a new primary function:
  - void (*sigset(int sig, void (*handler)(int)))(int)
  - Since sigset accepted the same parameters as before:
    - #define signal sigset /* would port older or BSD4.2 code */
- Introduced a new default for disposition: SIG_HOLD (in addition to SIG_DFL, SIG_IGN)

BSD and SysV Handle Unreliability Issue—In Incompatible Ways

- SVR3 added its own set of new functions for reliable signals:
  - int sighold(int sig); /*adds sig to the signal mask disposition */
  - int sigrelse(int sig); /* removes sig from the signal mask disposition, and waits for signal to arrive (suspends)*/
  - int sigignore(int sig); /* sets disposition of sig to SIG_IGN */
  - int sigpause(int sig); /* combination of sigrelse and pause(), but safe */
- *examples (sigset.c)*

Enter POSIX Signals

- Uses the concept of signal sets from 4.2BSD
- A signal set is a bitmask of signals that you want to *block*, i.e., signals that you specifically *don't* want to handle
- Each bit in the bitmask (an array of 2 unsigned longs) corresponds to a given signal (i.e., bit 10 == SIGUSR1)
- All signals not masked (not blocked) will be delivered to your process
- In POSIX signals, a blocked signal is not thrown away, but buffered as *pending*, should it become unmasked by the process at some later time

Central POSIX Functions

- int sigaddset(sigset_t * set, int signo);
  - adds a particular signal to the set
- int sigemptyset(sigset_t * set);
  - Zeros out the bitmask (program wants *all* signals)
- int sigfillset(sigset_t * set);
  - Masks all signals (blocks all signals)
- int sigdelset(sigset_t * set, int signo);
  - unmasks signo from the set (program wants the signal)
- int sigsend(idtype_t idtype, id_t id, int sig);
- int sigsuspend(const sigset_t * set);

POSIX sigaction

int sigaction (int sig, const struct sigaction *iact, struct sigaction *oact);

```
struct sigaction {
        __sighandler_t sa_handler;
        void (*sa_sigaction)(int, siginfo_t *, void *);
        unsigned long sa_flags
        …
        sigset_t sa_mask;  //set of signals to be BLOCKED
};
```

- sa_flags
  - * SA_RESTART flag to automatically restart interrupted system calls
  - * SA_NOCLDSTOP flag to turn off SIGCHLD signaling when children die.
  - * SA_RESETHAND clears the handler (ie. resets the default) when the signal is delivered (recidivist).
  - * SA_NOCLDWAIT flag on SIGCHLD to inhibit zombies.
  - * SA_SIGINFO flag indicates use value in sa_sigaction over sa_handler

POSIX Reentrant Functions

- Reentrant functions are those functions which are safe for reentrance:
  - Scenario: a signal SIGUSR1 is received in the middle of myfunc().
  - The handler for SIGUSR1 is called, which makes a call to myfunc()
  - myfunc() has just been “reentered”
- A function “safe” for reentrance is one that:
  - defines no static data
  - calls only reentrant functions or functions that do not raise signals

POSIX Reentrant-Safe Functions

- alarm, sleep, pause
- fork, execle, execve
- stat, fstat
- open, close, creat, lseek, read, write, fcntl, fstat
- sigaction, sigaddset, sigdelset, sig* etc.
- chdir, shmod, chown, umask, uname