OS Customization versus OS Code Modularity

ECE 344 – Fall 2006
Hans-Arno Jacobsen

Thanks to Michael Gong, Vinod Muthusamy, and Charles Zhang for helping to find interesting examples et al.
Possibly a Debugging Concern

```c
#ifdef SLOWER
#ifdef SLOW
#define SLOW
#endif
#endif
#ifdef SLOW
static void
checksubpage(struct pageref *pr){
// code removed
}
#endif
#endif
#endif
#endif

#else
#define checksubpage(pr)
((void)(pr))
#endif
#endif

#ifdef SLOWER
static void
checksubpages(void) {
// code removed
}
#endif
#endif
#endif

#ifdef SLOWER
static void
checksubpages(void) {
    // code removed
}
#endif
#endif
#endif
```
Possibly a Debugging Concern

/* SLOWER implies SLOW */
#ifdef SLOWER
    #ifndef SLOW
    #define SLOW
    #endif
#endif

#ifdef SLOW
    static void
    checksubpage(struct pageref *pr){
        // code removed
    }
    … (next column)
#endif // ifdef SLOW

#if defined SLOWER
    static void
    checksubpages(void) {
        // code removed
    }
#endif // ifdef SLOWER

#else
    #define checksubpage(pr) ((void)(pr))
#endif // ifdef SLOW

#if defined SLOWER
    static void
    checksubpages(void) {
        // code removed
    }
#endif // ifdef SLOWER

#endif // ifdef SLOWER
Observations

• Most likely the OS designers’ way of debugging memory allocation (guess)
• Multiple highly concentrated concerns to customize a part of OS for debugging
• Hard to read, understand, modify, test …
• FAST or NORMAL not even explicitly documented in code
Platform Support

...  
#define va_arg(__AP, __type)  
  ((__AP = (char *) ((__alignof__ (__type) > 4  
    ? ((__AP + 8 - 1) & -8  
      : ((__AP + 4 - 1) & -4)  
      + __va_rounded_size (__type)))),  
  *__type__) (void *)__AP - __va_rounded_size (__type))
#else  
/* For little-endian machines. */
...
#endif  
#endif  
#endif /* ! defined (__mips_eabi) */

- More of the above  
- Hardware platform specific customizations
static int __init readonly(char *str) {
    if (*str)
        return 0;
    root_mountflags |= MS_RDONLY;
    return 1;
}

static int __init readwrite(char *str) {
    if (*str)
        return 0;
    root_mountflags &= ~MS_RDONLY;
    return 1;
}

Linux Kernel 2.6: kernel initialization: do_mount.c
Lock & unlock I

int is_orphaned_pgrp(int pgrp) {
    int retval;
    read_lock(&tasklist_lock);
    retval = will_become_orphaned_pgrp(pgrp, NULL);
    read_unlock(&tasklist_lock);
    return retval;
}

• The same **scattering** and **crosscutting** of **synchronization concern** (see error checking)
• Similar pieces of code all over the place

Linux Kernel 2.6: kernel/exit.c
int session_of_pgrp(int pgrp) {
    struct task_struct *p;  int sid = -1;
    read_lock(&tasklist_lock);
    do_each_task_pid(pgrp, PIDTYPE_PGID, p) {
        if (p->signal->session > 0) {
            sid = p->signal->session;
            goto out;  }
    } while_each_task_pid(pgrp, PIDTYPE_PGID, p);
    p = find_task_by_pid(pgrp);
    if (p)
        sid = p->signal->session;
    out:
        read_unlock(&tasklist_lock);
    return sid;
}
static int
try_to_wake_up(task_t *p, unsigned int state, int sync) {
    int cpu, this_cpu, success = 0;
    unsigned long flags;
    long old_state;
    runqueue_t *rq;
    #ifdef CONFIG_SMP
    unsigned long load, this_load;
    struct sched_domain *sd, *this_sd = NULL;
    int new_cpu;
    #endif
    ... (next slide)
rq = task_rq_lock(p, &flags);
old_state = p->state;
if (!(old_state & state))
    goto out;
if (p->array)
    goto out_running;
cpu = task_cpu(p);
this_cpu = smp_processor_id();

#ifdef CONFIG_SMP
if (unlikely(task_running(rq, p)))
    goto out_activate;
new_cpu = cpu;
schedstat_inc(rq, ttwu_cnt);
...
#endif
Summary

- Certain concerns **crosscut** the principal or **core logic** (a.k.a. crosscutting concerns)
- **Similar concern** code **scatters** across the code base
- **Different pieces** of concern code **tangled** with core logic
- Scattering, tangling, and crosscutting apparently leads to code
  - that is hard to read and understand, let alone maintain
  - where the design intent is not cleanly represented in the code
  - where concerns are not well separated and modularized
  - removing a concern is error-prone
Need for Customization

• Customization of OS code is unavoidable
• OS code is often tailored to different hardware platforms
• … creating a whole family of OS versions
• Variety of hardware features (on different platforms) have far reaching implication for OS code
• Traditionally dealt with
  – At configuration time (various tools)
  – At compile time #ifdefs/#defines (driven through a make process or by a configuration tool)
  – Dynamically loadable kernel modules
Customization in OS/161

• "options" declarations in the config file
  - options dumbvm defines OPT_DUMBVM in the code

• Definition of OPT_SYNCHROBS leads to conditional code in
  - kern/include/clock.h
  - kern/include/test.h
  - kern/main/menu.c
  - kern/test/tt3.c
  - kern/thread/thread.c

• This is an example for crosscutting conditional compilation in OS/161
Crosscutting

• Crosscutting phenomenon is often not due to bad design
• But tied to the characteristics of traditional development techniques
• … the decomposition mechanism of traditional development paradigms
  – Files, functions, structures
  – Classes, objects, interfaces, methods
Conventional Programming Paradigms

- Red shows lines pertaining to a given concern
- Not in just one place (i.e., file, function)
- Not even in a small number of places (files or functions)
- Example is a bit out of context for operating systems
- OS code would show very similar footprints
Is there a Solution?

• For separating crosscutting concerns from core code
• Pick and choose the concerns required (based on hardware platform etc.)
Yes ☺️!
Aspect-oriented Programming (AOP)

- AOP is a programming paradigm that aims to support the modularization of crosscutting concerns in software
- AOP is complementary to existing paradigms
- Emerged about 10 years ago from different research efforts studying the Separation of Concerns in software
- Supported in industry today by IBM, BEA,…
- AOP support is available for Java, C, C++ …
- AspectJ, AspectC, AspectC++
Key Idea

- Crosscutting concerns are represented by aspects in the program sources.
- Required aspects are woven into the program.
- The program is fully unaware of the aspect (i.e., in the sources, there is no aspect code inside the program).
  - Note, there are a few AOP approaches around today that do not fully follow this model (i.e., some code present in program).
- The program is often referred to as the base program or the core advised by the aspect code.
- Aspects specify when and what code to execute.
- This specification is declarative and outside the core.
- For AspectC weaving happens at compile time (other models are load time or run-time weaving.)
Example: Key Idea

OS/161 src/kern/aspect/trace.ac:

```c
before(): call ($ $bootstrap$(...)) {
  kprintf("> Entering %s \n", this->funcName);
}
```

Join point declaration

Pointcut declaration

advice
Join Points

• Well-defined points in the execution of a program
  – The point a function is called
  – The point a function is executed

• Examples for C
  – Function calls (before/after) (call site)
  – Function execution (before/after) (called site)
  – ...

• Examples for Java
  – Method calls & execution
  – Field reads & writes
  – Exceptions
  – ...

Pointcuts

- Declaratively define sets of join points
- **Call pointcut** (all join points associated with the call of a function)
- **Execution pointcut** (all join points associated with the execution of a function)
- Example
  
  ```
  call($ $bootstrap$(...))
  ```

  - All call join points involving functions that contain the word “bootstrap” in the function name
  - With any list of input parameter types
  - With any return value type
Advice

• The code executed when the associated pointcut matches a join point
Example: Memory Profiling 1

```c
size_t totalMemoryAllocated;
int totalAllocationFuncCalled;
int totalFreeFuncCalled;
void initProfiler(){
    totalMemoryAllocated = 0;
    totalAllocationFuncCalled = 0;
    totalFreeFuncCalled = 0;
}
void printProfiler(){
    printf("total memory allocated = %d bytes\n",
            totalMemoryAllocated);
    ...
    totalAllocationFuncCalled);
    ...
    totalFreeFuncCalled);
}
```
Example: Memory Profiling II

before(): execution(int main()) {
    initProfiler();
}

after(): execution(int main()) {
    printProfiler();
}

before(size_t s): call($ malloc(...)) && args(s) {
    totalMemoryAllocated += s;
    totalAllocationFuncCalled ++;
}

Example: Memory Profiling III

```c
before(size_t n, size_t s): call($ calloc(...)) && args(n, s) {
    totalMemoryAllocated += n * s;
    totalAllocationFuncCalled ++;
}
before(size_t s): call($ realloc(...)) && args(void *, s) {
    totalMemoryAllocated += s;
    totalAllocationFuncCalled ++;
}
before() : call(void free(void *)) {
    totalFreeFuncCalled++;
}
```
Example: Memory Profiling IV

• Is the code thread safe?

• Is thread-safety an aspect?

• Left as an exercise for the reader.
Use of AOP

• Build aspects into systems right from the start (i.e., design with aspects in mind)
• Use aspects to aid in debugging, analyzing, policy checking …
• Use aspects to refactor existing systems
  – Tailoring and customization
  – Adaptation
  – Extension
AspectC

- Developed by Michael Gong and myself
- Aspect-oriented extension to C
- ANSI-C compliant
- gcc source-compatibility
- Compiler and generated code is portable (mostly 😊)
- Seamless Linux, Solaris and Windows support (Mac OS X support in progress.)
- Integration in existing build processes possible
- Code transparency through source-to-source transformations
- Based on open source license and compiler
AspectC Resources

- Assignment 0 handout
- AspectC Tutorial
- AspectC Language Specification
- See the AspectC web site for submitting a bug report, if you think you found one
Resources

• Aspect-oriented Software Development Portal
  – http://www.aosd.net

• AspectJ
  – http://www.eclipse.org/aspectj/

• AspectC++
  – http://www.aspectc.org

• AspectC