Projects 3 & 4

CS-3013 Operating Systems
Hugh C. Lauer

(Slides include materials from
Modern Operating Systems, 3rd ed., by Andrew Tanenbaum
and from Operating System Concepts, 7th ed., by Silbershatz, Galvin, & Gagne)
Previously in CS-3013

Project 3 — a multithreaded simulation of a communal bathroom
  • Lots of threads and synchronization

Project 4 — a kernel message system and test program
  • Lots of synchronization inside Linux kernel
  • Big project
  • Multi-threaded test program

Not enough time in term for both

Too big/hard for some students

Testing gets short shrift
This Term — Choice

Kernel Project with a serious multi-threaded, multi-process test program
- Combines Projects 3 & 4
- Teams strongly recommended

Bathroom simulation plus research report on non-traditional operating systems
- Individual projects only
Choice B

Alt-Project 3, Bathroom Simulation
plus
Alt-Project 4, Research & Report
Bathroom Simulation —
Objective

• To learn to use threads and synchronization mechanisms in user space

• To implement a multi-threaded program that aggressively pounds on a shared resource
The Problem

- Model a communal bathroom used by both sexes

Possible states of the bathroom:–
- Vacant
- Occupied by women
- Occupied by men

Rules:–
- Anyone may enter if vacant
- A user may enter if occupied by same sex
  – Any number of same sex may be in bathroom at same time.
- A user must wait until vacant if occupied by opposite sex
  – Any number of users may by waiting at same time.
Implementation

- Control module in C or C++
  - Using `pthread` synchronization functions

- Multi-threaded Test Program
  - One thread per simulated person
  - Operate for simulated time
  - Random attempts to access and stay in bathroom

No Java. Too easy with SYNCHRONIZED classes
Implementation

• Control module:—
  • A .h file and a .c or .cpp file
• Interface:—
  enum gender = {male, female};
  void Enter(gender g);
  void Leave(void);
  void Initialize(...);
• Must correctly maintain:
  • State
  • # of users in bathroom
  • Gender of users in bathroom

Waits until *vacant* if occupied
Sets state to *vacant* if this is the last user to leave
Notifies waiting threads that they can proceed
Multiple threads trying to access or change at same time!
Multi-threaded Test Program

- **main()** function is *master thread*
  - Interprets `argc` and `argv`
  - Spawns $n$ user threads representing individuals
    - Randomly sets *gender, loop count* of each user thread
    - Specifies mean *arrival time* and mean *stay time*
  - Waits for user threads to finish
  - Prints summary
  - Exits cleanly

With no outstanding user threads!
User Thread

- Loops *loop count* times
  - A random number
- For each iteration:
  - Generate random *arrival* and *stay* times based on means
  - Sleep *arrival* units of time
  - Invoke `Enter(gender)`
    - May wait a long time if occupied by opposite gender
  - Sleep *stay* units of time
  - Invoke `Leave()`
  - Collect statistics
- Print statistics (min, max, average wait time to enter)
- Exit cleanly

So that master thread knows that user thread is done
As Simple as That!

With a few minor challenges!
**Synchronization Challenges**

- Shared data structure representing the state of bathroom, count, etc.
  - *Mutex* for protecting shared access
  - *Condition variable* for waiting
- Clean exit from user thread
  - `pthread_join` or barrier synchronization & detach
- Printing to *stdout* from multiple threads
  - Without getting mixed up.

Note that *mutex* is for managing data structure, not bathroom data structure, not bathroom.
Other Challenges

- Random number generation
  - Uniform around a mean
  - Normal distribution for stay times
  - Use “standard deviation” for width
- Better random number generation
  - Poisson distribution for arrivals?
  - May need to specify parameters on command line
- Simulating units of time
  - `#include <unistd.h>`
  - Suggest 1 or 10 milliseconds for one time unit.
Write-up

• Explain *invariant* of shared data structure
  • Show `Enter()` and `Leave()` preserve invariant
  • Or show states & state transition diagram

• Explain master thread and user thread relationship
  • Creation and termination

• Analysis of test cases
  • Three with different parameters
Due Date

• Project due on Tuesday, September 27, 2011

• Submit via Turnin
  • Alt-Project 3
  • Zip all files together!

• Report to instructor or TAs any difficulties
Questions?
Alternate Project 4

- Research and report on non-traditional operating systems
  - At least one smart phone operating system
  - At least one embedded operating system
  - At least one real-time operating system
- (one operating system may satisfy more than one requirement)
  - Must do three total
Alternate Project 4 (continued)

• May not be Linux, Unix, Windows, or Macintosh
  • Desktop, laptop, server
• May not be one of the lookalikes or variants
  • GNU/Linux, FreeBSD, Solaris, etc.
Two Steps

1. List of five such operating systems
   - One paragraph plus reference for each
   - Submit by Thursday, September 29
   - Professor will select three

2. Ten page report on three OS’s in depth
   - Compare and contrast
   - Discuss major abstractions, security, etc.
   - Explain differences, reasons why, focus, target market or applications, etc.
   - Due Sunday, October 9
Questions?

Alternate Project 3
Alternate Project 4
Combined Project 3 & 4

Kernel Message System
Multi-threaded Test Program
**Project**

- To build and test a message-passing system for communication among separate address processes
  - Implement API for message-passing functions
  - Kernel system calls to handle messages
  - Aggressive test program to exercise the message passing system
Objectives

- To gain experience programming and testing synchronization and IPC operations
- To gain experience working inside the Linux kernel on a non-trivial problem
- To integrate many of the lessons of this course
Integrates Many Lessons of This Course

- Processes and threads
  - In test program
- Synchronization and race conditions
  - In kernel
- Adding system calls
- Memory allocation & slab allocation
  - In kernel
- Random testing
- Reading kernel code
  - And modifying it
- Initialization subtleties
  - In kernel
- ...

Projects 3 & 4

CS-3013, A-Term 2011
Overview

• Add a mailbox to each process
  – Abstract object capable of holding *messages*
  – Messages are of bounded length, undefined structure
  – Mailboxes may be of bounded size
• All threads of a process share a mailbox
• Any Linux task can send a message to any mailbox
  – Including own mailbox
  – Addressed by `pid_t`
Overview (continued)

• Tasks may receive messages only from “own” mailbox
  – Any thread of task may receive!
• Mailbox created during *fork*
• Mailbox deleted during process *termination*
• Mailbox may be “stopped”
  – I.e., no more message are accepted
This Project

• Predefined API (Application Program Interface) at
  – `mailbox.h` in this directory

• All students must implement same API
  – User space interface program
  – Kernel implementation
  – Test program
API

int SendMsg(pid_t dest, void *msg, int len, bool block)

• Sends a message body at *msg to process dest
• Length = len; not more than MAX_MSG_SIZE
• Blocks if mailbox full and block = TRUE
• Returns zero if successful, error code if not

int RcvMsg(pid_t *sender, void *msg, int *len, bool block)

• Gets a message from own mailbox, puts in *msg
• Sender process ID returned in *sender
• Blocks if mailbox empty and block = TRUE
• Returns zero if successful, error code if not

• Messages in FIFO order
int ManageMailbox(bool stop, int *count)

• Gets number of messages currently queued in mailbox
  
  If stop = TRUE, prevents mailbox from receiving more messages
  
  – Unblocks all waiting SendMsg and RcvMsg calls
  
  – Future RcvMsg calls can still retrieve remaining queued messages

• Returns zero if successful, error code if not
API — Documented Error Codes

- **MAILBOX_FULL**
  - Non-blocking send
- **MAILBOX_EMPTY**
  - Non-blocking receive
- **MAILBOX_STOPPED**
  - On any send
  - Also after blocked send or receive
- **MAILBOX_INVALID**
  - On any call
- **MSG_TOO_LONG**
  - On send
- **MSG_ARG_ERROR**
  - Invalid argument or pointer
  - `copy_to_user` or `copy_from_user` fails
- **MAILBOX_ERROR**
  - Any other kind of error
- You may add other error codes as needed
Kernel Implementation

- Start with prePatch-Project4
  - i.e., so we all have same task_struct and header files
- Three system calls
  .long sys_mailbox_send
  .long sys_mailbox_rcv
  .long sys_mailbox_manage
- Create and delete mailboxes
- Memory allocation within kernel
  - All messages of fixed size
- Synchronization within kernel
  - Suggest simulating a monitor & producer-consumer per mailbox
Kernel Implementation

- Start with `prePatch-Project4`
  - i.e., so we all have same `task_struct`
- Three system calls
  - `long sys_mailbox_send`
  - `long sys_mailbox_rcv`
  - `long sys_mailbox_manage`
- Create and delete mailboxes
- Memory allocation within kernel
  - All messages of fixed size
- Synchronization within kernel
Pre-Patch your Kernel

- Apply `prePatch-Project4` to a clean kernel tree
- Adds to `task_struct`
  - `struct CS3013_mailbox *mailbox`
  - `spinlock_t mailbox_lock`
- Defines syscalls in `syscall.S` and `unistd.h`
  
  ```
  .long sys_mailbox_send    /* 341 */
  .long sys_mailbox_rcv     /* 342 */
  .long sys_mailbox_manage  /* 343 */
  ```
- Reason
  - Graders’ time in grading
  - *Don’t* modify other commonly used header files!
Note on “patch”

- Semantics changed slightly since 2009
- “patch” refuses to patch read-only files

Either

- Move the three files to be patched by hand
- Install an older version from patch-2.5.9.tar.gz
  - See documentation in this package
  - Installs in /usr/local/bin/patch

Already installed in your virtual machine
Kernel Implementation

- Start with prePatch-Project4
  - i.e., so we all have same task_struct
- Three system calls
  - .long sys_mailbox_send
  - .long sys_mailbox_rcv
  - .long sys_mailbox_manage
- Create and delete mailboxes
- Memory allocation within kernel
  - All messages of fixed size
- Synchronization within kernel
Creating a Mailbox

- During `do_fork()`
  - In `kernel/fork.c`

- *If this is a new process*
  - Use `kmalloc()` to allocate a new data structure for a new mailbox
  - Initialize
  - Set pointer in `task_struct->mailbox`
  - Initialize `mailbox_lock`

- *If this is just a new thread, keep existing mailbox*
  - Indicated by `CLONE_THREAD` argument to `do_fork()`
  - Copy `mailbox` pointer
  - Initialize `mailbox_lock`
Creating a Mailbox (continued)

• No mailbox for kernel threads, etc.
  • Indicated by null task_struct->mm

  • Set task_struct->mailbox to null
Kernel Implementation of Mailbox Operations

- Kernel functions (in own C module)
  - Three system calls for user-visible functions
    
    ```c
    sys_mailbox_send
    sys_mailbox_rcv
    sys_mailbox_manage
    ```
  - Two other functions for kernel support
    ```c
    mailbox_create
    mailbox_destroy
    ```

- Combination of *monitor* and *producer-consumer*
  - `spinlock_t` initialized to `SPIN_LOCK_UNLOCKED`
    - A mutex to protect the mailbox data structures
  - Two semaphores
    - `empty` initialized to maximum # of messages
    - `full` initialized to zero
  - Linked list of messages
  - Flag for stopped mailbox; counters and other fields as needed
Kernel Implementation (continued)

- **RcvMsg**
  - Wait on semaphore — `down_interruptible(&full)`
  - Grab `spin_lock`
  - Unlink first message from mailbox linked list
  - Release `spin_lock`
  - Post semaphore — `up(&empty)`
  - `copy_to_user` to copy message body and other information to caller
  - Free kernel space for unlinked message (see below)

- **Non-blocking receive**
  - Use `down_trylock(&full)`
Kernel Implementation (continued)

- **RcvMsg**
  - Wait on semaphore — `down_interruptible(&full)`
  - Grab `spin_lock`
  - Unlink first message from mailbox linked list
  - Release `spin_lock`
  - Post semaphore — `up(&empty)`
  - `copy_to_user` to copy message body and other information to caller
  - Free kernel space for unlinked message (see below)

- **Non-blocking receive**
  - Use `down_trylock(&full)`

Need to check if mailbox is empty. Or if task is interrupted!
Kernel Implementation (continued)

• RcvMsg
  • Wait on semaphore — \texttt{down\_interruptible}(&\texttt{full})
  • Grab \texttt{spin\_lock}
  • Unlink first message from mailbox linked list
  • Release \texttt{spin\_lock}
  • Post semaphore — \texttt{up}(&\texttt{empty})
  • \texttt{copy\_to\_user} to copy message body and other information to caller
  • Free kernel space for unlinked message (see below)

• Non-blocking receive
  • Use \texttt{down\_trylock}(&\texttt{full})
Kernel Implementation (continued)

- **SendMsg**
  - Allocate kernel space for new message (see below)
  - `copy_from_user` to copy message body into kernel
  - Fill in other details (sender, length, etc.)
  - Wait on semaphore — `down_interruptible(&empty)`
  - Grab `spin_lock`
  - Link new message to end of mailbox linked list
  - Release `spin_lock`
  - Post semaphore — `up(&full)`

- **Non-blocking send**
  - Use `down_trylock(&empty)`
Kernel Implementation (continued)

- ManageMailbox
  - Grab spin_lock
  - Count number of messages
  - If stop = TRUE
    - Set flag to prevent any more tasks from trying to send
    - Determine if any tasks waiting
  - Release spin_lock
  - Trick waiting tasks into “unblocking,” returning error
    - Use counters to keep track of number of waiting tasks
    - Do `up(&full)` or `up(&empty)` to unblock
Deleting a Mailbox

- In `do_exit()`
- *Only* if the “task group” is dead!
- Stop the mailbox to be sure that blocked send operations can complete!
  - Flush messages (to free their space)
  - Make sure all blocked operations are done
- Free the mailbox data structure
  - `kfree()`
- Zero out `task_struct->mailbox`

A subtle race condition can occur here (see below)
Memory Allocation Resources

- **kmalloc()**, **kfree()**
  - `linux/slab.h`
  - Similar to `malloc()` & `free()`, but with flags

- **Slab allocator**
  - `kmem_cache_t* kmem_cache_create()`
  - `void* kmem_cache_alloc()`
  - `void kmem_cache_free()`
  - `int kmem_cache_destroy()`
Memory Allocation Resources
(continued)

• Use slab allocator for messages
  – All message bodies the same size in kernel
  – Highly optimized for rapid allocation and free
  – Low fragmentation

• Initialization
  – Establish cache and any static variables
during fork_init() (in kernel/fork.c)
  – All mailboxes share same cache
Locking Tools in the Kernel

- **Mailbox**
  - Needs to be locked to link and unlink messages
  - Also to change state (**STOP**)
- **Remember** – the Linux Kernel is fully preemptive!
  - System call may be preempted before completion
  - Interrupt may schedule another process at any time
  - Interrupt may manage shared resources
  - But *not* while holding a spinlock

  – Due to support for symmetric multi-processing
Robert Love says ...

- It is a major bug if ...
  - An interrupt occurs to access a resource while kernel code is also manipulating that resource
  - Kernel code is preempted while accessing a shared resource
  - Kernel code sleeps while in the middle of a critical section
  - Two processors access same data at same time
- Implementing locking is *not* hard
- Tricky part is identifying *what* to lock.
Deleting mailbox

- Must stop mailbox first
  - Use `ManageMailbox` to unblock waiting tasks
- Be sure every waiting task has had time to complete operation
  - Potential race condition
- Flush remaining messages
- Delete mailbox data structure
Subtle Race Condition

• Process A may be sending msg to Process B
  • SendMsg gets copy of pointer B->mailbox
  • Gets pre-empted while in kernel!

• Process B tries to exit!
  • Deletes mailbox structure
  • Zeros out B->mailbox
  • Exits cleanly

• Process A gets to top of ready queue
  • Is dispatched
  • Continues to send to B->mailbox. Big trouble!
Subtle Race Condition (continued)

• **mailbox_lock** in `task_struct` provided for your convenience
  • You may use it however you wish
  • You may ignore it and do something else

• No single “correct” solution

• Suggestion:—`mailbox_lock` protects the **mailbox** pointer, not the `mailbox` itself
  • Perhaps an atomic integer is useful (p.132)
  • Counts number of tasks holding pointer to mailbox
Testing

• Make a multi-threaded program like Alt-Project 3, but …
  • Multiple processes
  • Each process with multiple threads
• Fork multiple processes
  • Create mailboxes, exchange mailbox IDs
• Randomly send messages to each other
  • Payload must be self identifying
• Acknowledge received messages
• Test extreme conditions
  • E.g., fill up mailbox
Dates and Milestones

- Project due at October 9
  - By September 23:–
    - Creation & deletion of mailboxes in `do_fork()`, `do_exit()`
    - Design of mailbox data structure for producer-consumer or simulated monitor
    - Test program for the same
  - By September 30:–
    - Sending and Receiving
    - Stopping mailbox with blocked calls on `SendMsg` or `RcvMsg`
    - Test program for all of this
  - By October 7:–
    - Flushing messages, race conditions,
    - Very aggressive test program
Final Submission

• Submit using *Turnin*

• Include
  – One **patch** file for kernel implementation
    • Difference from **prePatch-Project4** and your implementation
  – User space implementation of **mailbox.h** interface
  – Test program(s) and results
  – **Makefile**
  – Write-up must explain synchronization & programming invariants
    • Sending and receiving messages
    • Shutting down mailboxes and race condition
    • Test program

*Put your name on all documents & at top of every edited file!*
Teams

• Strongly advise 2- or 3-person teams
  • Teams strongly advised

• Please notify instructor of team membership by September 23
Project Points

- Kernel Message System – 45 points
- Aggressive Test Program – 30 points
Questions?