PURDUE
UNIVERSITY®

CS 25200: Systems Programming

Lecture 21: Semaphore Implementation, Thread Safety and Race Conditions

Prof. Turkstra

Midterm Exam

- Thursday, October 17 8:00pm – 10:00pm
  - WALC 1055
  - Covers material through and including lecture 19 (scheduling)
  - Lectures and labs
  - Homework out tonight/tomorrow
    - Due before exam
    - Print and hand in
    - Not a comprehensive indicator

Announcements

- Lab 3 Checkpoint grades available
- Quiz/attendance too
- Remember: one week regrade request window!

Lecture 21

- Semaphore review
- Semaphore implementation
- Spinlocks
- Threads and fork()
- Thread safety
- Races

Terminology

- Critical section: section of code or collection of operations in which only one thread may be executing at a time
- Mutual exclusion: the property that exactly one thread is doing a certain thing at one time

Count example

```
#include <stdio.h>
#include <pthread.h>

int count;

void inc(long n) {
  for (int i = 0; i < n; i++) {
    count++;
  }
}

void dec(long n) {
  for (int i = 0; i < n; i++) {
    count--;
  }
}
```

Count example

```
int main(int argc, char **argv) {
  long n = 10000000;
  pthread_t t1;
  pthread_t t2;

  pthread_create(&t1, NULL, (void * (*)(void *)) inc, (void *) n);
  pthread_create(&t2, NULL, (void * (*)(void *)) dec, (void *) n);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  printf("%d\n", count);
  return 0;
}
```

Race condition

- count++/-- statement is not atomic
  - load C
  - add one to C
  - store C
- Interleaving of thread statements impacts the result
  - Non-deterministic

Non-determinism

- Instruction execution order among separate threads depends on many things
  - Threading implementation
  - Operating system (scheduling, preempting)
  - Hardware interrupts
- This occurs even on single CPU, single core systems

Synchronization

- It is the programmer's job to anticipate all possible orderings and protect against errors
- Concurrency and synchronization is hard
  - Sometimes the overhead of implementing and debugging a concurrent program is not worth it
    - Therac-25
  - Safety critical systems should always have hardware interlocks

Criteria

- Musts
  - Processes not in critical section should not block others
  - No one waits forever
  - Multi-processor friendly
- Desirable
  - Fairness – everyone eventually gets into the critical section
  - Efficient – don't waste resources (no busy waiting)
  - Simple – symmetric code, easy to use
    - Like bracketing

Processes and mutual exclusion

- Always lock before manipulating shared memory
- Always unlock after manipulating shared memory
- Do not lock again if already locked
- Do not unlock if not locked by you
- Do not spend large amounts of time in critical section

Atomic

- Appears to the entire system as occurring all at once without interruption
  - No interrupts
  - No signals
  - No concurrent processes or threads

Semaphore

- Synchronization variable that takes on positive integer values
  - Dijkstra, 60s
- Two operations:

  P(semaphore): atomic operation waits for semaphore > 0, then decrements by one
  - "Proberen" in Dutch
- V(sempahore): atomic operation increments by one
  - "Verhogen"

Pseudo code

```
P(S)
wait(S) {
  while (S < = 0);
  S--;
}
```

```
V(S)
signal(S) {
  S++;
}
```

- Done atomically
- Not usually in hardware – implementation later

Binary semaphores

- Often called a mutex or lock
- Semaphore that takes on values of 0 and 1 only
- Too much milk?

```
P(milkSemaphore)
if (!milk)
  buy milk;
V(milkSemaphore)
```

Properties

- Machine-independent
- Simple
- Works with many processes
- Can have different critical sections with different semaphores
- Can acquire many resources simultaneously (multiple P's)
- Can permit multiple processes to enter critical section at once

Usage

- Mutual exclusion
  - One process is accessing critical section at a time
  - What about separate groups of data that need to be accessed independently
- Condition synchronization
  - Permit processes to wait for something
  - What if disparate groups of processes want to wait for unrelated events?

Fixing inc.c

Implementation

- Uniprocessor solution: disable interrupts!

```
typedef struct {
  int count;
  queue q;
} semaphore;
```

P

```
void P(semaphore s) {
  disable interrupts;
  if (s->count > 0) {
    s->count--;
    enable interrupts;
    return;
  }
  add(s->q, current_thread);
  sleep(); // re-dispatch
  enable interrupts;
  return;
}
```

V

```
void V(semaphore s) {
  disable interrupts;
  if (isEmpty(s->q)) {
    s->count++;
  } else {
    thread = removeFirst(s->q);
    wakeup(thread); // put thread on ready q
  }
  enable interrupts;
  return;
}
```

Multiprocessor?

- Cannot just turn off interrupts
  - Doesn't prevent other processors from accessing shared memory
- Turn off other processors?
  - Bad :-(
- Use atomic read and write?
  - Needs to be atomic across all processors!
- Big research area for a long time

Test-and-set (IBM)

- Atomic read-modify-write instruction
- TAS – on most CISC architectures
- Semantics:
  - Set value to k, but return old value
  - k = 1 → binary semaphore

```
int lock;
while (TAS(&lock, 1) != 0);
<critical section>
lock = 0;
```

TAS

- Implemented by memory hardware or CPU refusing to relinquish bus access
- Still have to disable interrupts on current core
- Why?

RISC Mechanism

- Load-linked
  - ldl – loads a word from memory and sets per-processor flag associated with that word (in cache)
  - Store operation to same location (by any processor) resets all processors' flags for that word
- Store-conditionally
  - stc – stores word iff flag still set, indicates success or failure

```
int lock;
while ((ldl(&lock) != 0) || !stc(&lock, 1));
<critical section>
lock = 0;
```

Purdue Trivia

- We are on the seventh iteration of the Boilermaker Special, Purdue's official mascot.
  - World's largest, fastest, heaviest, and loudest collegiate mascot
  - Dedicated 9/3/2011
- Boilermaker Xtra Special is a smaller version designed for use indoors
  - Eighth iteration
- Both are entrusted to the Purdue Reamer Club

Spinlocks

- The two previous solutions rely on a spinlock
  - Busy waiting
- Still used for multicore implementations
  - Optimization: if thread holding lock is sleeping, also sleep

Mutexes

- Spinlocks are not guaranteed to be fair
- Can build semaphores on top of spinlocks
  - Similar to single-processor implementation
  - Replace enable/disable interrupts with spin_lock() and spin_unlock()

Threads and fork()

- fork() behaves differently on some platforms
- For most *nix variants, calling fork() duplicates only the calling thread
  - Even if that thread is the parent of other threads
- Solaris' thr_create() will also duplicate the children
  - fork1() if you wish to avoid that

Thread safety

- Thread-safe code manipulates shared data structures properly
- Thread safe: guaranteed to be free of race conditions when accessed or used by multiple threads simultaneously
- Not thread safe: should not be used simultaneously by multiple threads
- Often designated in man pages

man page

- Function may be marked MT-Unsafe
  - strtok()
- Or MT-Safe
  - strtok_r()
- Often times MT-Safe code is re-entrant – state is saved e.g. on a stack

Linked list race

```
typedef struct linked_list {
  struct node *head;
} linked_list;

typedef struct node {
  int value;
  struct node *next;
} node;

linked_list *list = malloc(sizeof(list));
```

```
int insert(linked_list *list, int val) {
  if (!list->head) {
    list->head = malloc(sizeof(node));
    list->head->value = val;
    return;
  }

  node *new_node = malloc(sizeof(node));
  new_node->next = list->head;
  list->head = new_node;
}
```

Race condition

- Output is dependent on the sequence or timing of uncontrollable events

Thread safe

```
typedef struct linked_list {
  pthread_mutex_t mutex;
  struct node *head;
} linked_list;

typedef struct node {
  int value;
  struct node *next;
} node;

linked_list *list = malloc(sizeof(list));
pthread_mutex_init(&list->mutex, NULL);
```

```c
int insert(linked_list *list, int val) {
  pthread_mutex_lock(&list->mutex);
  if (!list->head) {
    list->head = malloc(sizeof(node));
    list->head->value = val;
    pthread_mutex_unlock(&list->mutex);
    return;
  }

  node *new_node = malloc(sizeof(node));
  new_node->next = list->head;
  list->head = new_node;
  pthread_mutex_unlock(&list->mutex);
}
```

Remove

```
int remove_head(linked_list *list) {
  node *tmp = NULL;
  int val;

  tmp = list->head;
  if (tmp == NULL) return -1;
  list->head = list->head->next;
  val = tmp->value;
  free(tmp);
  tmp = NULL;

  return val;
}
```

Remove MT-Safe

```
int remove_head(linked_list *list) {
  node *tmp = NULL;
  int val;

  pthread_mutex_lock(&list->mutex);
  tmp = list->head;
  if (tmp == NULL) {
    pthread_mutex_unlock(&list->mutex);
    return -1;
  }
  list->head = list->head->next;
  pthread_mutex_unlock(&list->mutex);
  val = tmp->value;
  free(tmp);
  tmp = NULL;

  return val;
}
```

What if?

- What if we forgot the first call to pthread_mutex_unlock()?
- What if we move pthread_mutex_unlock() to right above the return statement?
- What if we had a mutex for every node?

Suggested homework

- Analyze singly linked list, doubly linked list, and binary tree algorithms
  - Where are the possible synchronization bugs
  - How can you prevent them?
  - What might the trade-offs be?

Questions?