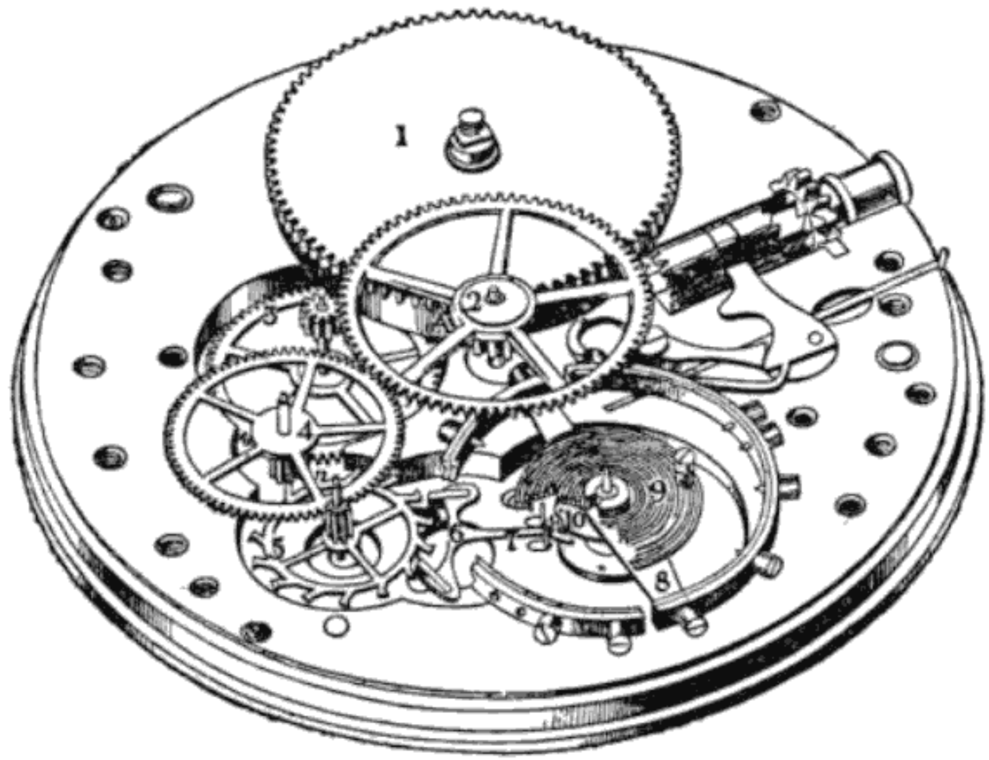
Achieving Synchronization or How to Build a Semaphore

CS 241

March 12, 2012



Announcements

MP5 due tomorrow

Jelly beans...

Today

- Building a Semaphore
- If time: A few midterm problems

Review: Semaphores

Problem: coordinating simultaneous access to shared data

```
int cnt = 0;

void * worker( void *ptr )
{
    int i;
    for (i = 0; i < ITERATIONS_PER_THREAD; i++)
        cnt++;
}
```

Shared data

Critical section
(just one line in this simple example)

Solution: mutually exclusive access to critical region

- Only one thread/process accesses shared data at a time

Semaphores for mutual exclusion

Basic idea

- Associate a unique semaphore *mutex*, initially 1, with each shared variable (or related set of shared variables)
- Surround corresponding critical sections with *wait(mutex)* and *post(mutex)* operations.

Terminology

- **Binary semaphore**: semaphore whose value is always 0 or 1
- **Mutex:** binary semaphore used for mutual exclusion
  - *wait* operation: "locking" the mutex
  - *post* operation: "unlocking" or "releasing" the mutex
  - "Holding" a mutex: locked and not yet unlocked
- **Counting semaphore**: used to count a set of available resources

goodcounter.c: good synchronization

```c
#include <semaphore.h>

...

int cnt = 0;
sem_t cnt_mutex;

int main(void)
{
    ...
    /* Initialize mutex */
    sem_init(&cnt_mutex, 0, 1);
    ...
}

void * worker( void *ptr )
{
    int i;
    for (i = 0; i < ITERATIONS_PER_THREAD; i++) {
        sem_wait(&cnt_mutex);
        cnt++;
        sem_post(&cnt_mutex);
    }
}
```

Necessary include

Declare mutex

Initialize to 1

Surround critical section

How do we build mutual exclusion?

```
lock();

critical_section();

unlock();
```

What goes here?

Assumption for remainder of lecture:
Above code is run simultaneously in multiple threads/processes

Mutual Exclusion Solutions

Software-only candidate solutions

- Lock variables
- “Turn”
- “Two flag and turn”

Hardware solutions

- Test-and-set / swap

Semaphores

Lock Variables

```
int lock = 0;
...
while (lock) {
   /* spin spin spin spin */
}
lock = 1;

critical_section();

lock = 0;
```

Lock Variables

```
int lock = 0;
...
while (lock) {
   /* spin spin spin spin */
}
lock = 1;

critical_section();

lock = 0;
```

```
lock = 1
lock = 0
lock = 1
lock = 1
```

No mutual exclusion!

Turn-based mutual exclusion

```
pthread_t turn = first_thread_id;
...
while (turn != my_thread_id) {
     /* wait your turn */
}
critical_section();
turn = other_thread_id;

…
```

Turn-based mutual exclusion

```
pthread_t turn = first_thread_id;
```

Process 0

Process 1

```
turn = 0
turn = 1
```

```
...
while (turn != my_thread_id) {
      /* wait your turn */
}
critical_section();
turn = other_thread_id;

...
```

No progress!

Other process
may not be executing
in this critical section.

Two Flag and Turn Mutual Exclusion

```
owner[0] = ~~false~~ true
owner[1] = ~~false~~ true
turn = ~~0~~ ~~1~~ 0
```

```
int owner[2]={false, false};
int turn;
…
owner[my_process_id] = true;
turn = other_process_id;
while (owner[other_process_id] &&
       turn == other_process_id) {
    /* wait your turn */
}

critical_section();

owner[my_process_id] = false;
…
```

Progress & mutual exclusion!

"Peterson's solution"

Are we done?

Peterson's algorithm works, but...

Problem: software solutions can be slow

- at just the moment we'd like to be fast: contention for shared resource
- Solution: hardware support

Atomic Test and Set Instruction

```
boolean test_and_set(boolean* lock) atomic {
    boolean initial = *lock;
    *lock = true;
    return initial;
}
```

atomic = executed without interruption

Test and Set for mutual exclusion

```
boolean lock = 0;
while (test_and_set(&lock))
    ;
critical_section();
lock = 0;
```

Understanding Test and Set

Original

```
boolean test_and_set(boolean* lock) atomic {
    boolean initial = *lock;
    *lock = true;
    return initial;
}
```

Functionally equivalent version

```
boolean test_and_set(boolean* lock) atomic {
    if (*lock == 1)
        return 1; // failure
    else {
        *lock = 1;
        return 0; // success
    }
}
```

Test and Set for mutual exclusion

```
boolean lock = 0;
while (test_and_set(&lock))
    ;
critical_section();
lock = 0;
```

Remaining problem: busy-waiting

Now are we done?

Hardware solutions are fast, but...

Problem: starvation

- No guarantee about which process "wins" the test-and-set race
- It'll eventually happen, but a process could wait indefinitely

Problem: deadlock

- Proc. 1 enters critical section, gets interrupted by higher priority Proc. 2
- P1 can't make progress: waiting to run until P2 is done
- P2 can't make progress: busy-waiting until P1 exits critical section

Problem: busy-waiting

- Critical section might be arbitrarily long
- Waiting processes all still spend CPU time!

These problems occur for software solutions too

Solution: Semaphores

Semaphores vs. Test and Set

Semaphore

```
semaphore s = 1;
...
sem_wait(&s);
critical_section();
sem_post(&s);
```

Test and Set

```
lock = 0;
...
while(test_and_set(&lock)
    ;
critical_section();
lock = 0;
```

The magic: avoid busy-waiting during sem_wait()

Inside a Semaphore

Add a waiting queue

Multiple process waiting on `s`

- Wake up one of the blocked processes upon getting a signal

Semaphore data structure

```
typedef struct {
    int      count;
    queue_t waiting;
} semaphore_t;
```

Binary Semaphores

```
typedef struct bsemaphore {
    enum {0,1} value;
    queue_t    queue;
} bsem_t;


void sem_wait_B (bsem* s) {
    if (s.value == 1)
        s.value = 0;
    else {
        place current process in s->queue;
        block current process;
    }
}
```

Binary Semaphores

```
typedef struct bsemaphore {
    enum {0,1} value;
    queue_t    queue;
} bsem_t;


void sem_post_B (bsem* s) {
    if (s->queue is empty())
        s->value = 1;
    else {
        remove process P from s->queue;
        place P on ready list;
    }
}
```

General Semaphore

```
typedef struct {
    int      count;
    queue_t queue;
} semaphore_t;
```

```
void sem_wait(semaphore_t* s) {
    s.count--;
    if (s.count < 0) {
        place P in s->queue;
        block P;
    }
}
```

```
void semSignal(semaphore_t* s) {
    s.count++;
    if (s.count ≤ 0) {
        remove P from s.queue;
        place P on ready list;
    }
}
```

Making the operations atomic

Isn't this exactly the problem semaphores were trying to solve?

- Are we stuck??!

Solution: resort to test and set:

```
typedef struct {
    boolean lock;
    int count;
    queueType queue;
} semaphore_t;
```

```
void sem_wait(semaphore_t* s) {
    while (test_and_set(lock)) { }
    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
    lock = 0;
}
```

Making the operations atomic

Busy-waiting *again*!

How are semaphores better than just test-and-set?

T&S: Busy-wait until ready to run

- Could be arbitrarily long!
- We're waiting for other processes which may be in long critical sections

Semaphores: Busy-wait just during sem_wait, sem_post

- Now we're waiting for other processes which are doing very short operations (sem_wait, sem_post)

Summary

Mutual exclusion possible in software

Mutual exclusion faster in hardware

- Common atomic instruction: test_and_set

Hardware operations used to bootstrap semaphores

- ...which block processes to avoid busy-waiting and can select which ones to un-block

Next time: Classic applications of mutual exclusion