CS330: Operating Systems

Software locks, Semaphore

Recap: Spinlocks with hardware support

- Architectural support for atomic operations like atomic exchange, compare-and-swap, LL–SC and atomic add can be used to build spinlocks
- Ticket spinlocks provide fairness in locking, example implementation with atomic-add
- Outstanding issues: Blocking locks (will come back after semaphores)

Today's lecture: Software-only locks, Semaphores

Buggy #1

```
int flag[2] = {0,0};
void lock (int id)    /*id = 0 or 1 */
{
    while(flag[id ^ 1]); // ^ → XOR
    flag[id] = 1;
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Solution for two threads, T_0 and T_1 with id 0 and 1, respectively
- We have seen that this solution does not work, Why?

Buggy #1

```
int flag[2] = {0,0};
void lock (int id)    /*id = 0 or 1 */
{
    while(flag[id ^ 1]); // ^ → XOR
     flag[id] = 1;
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Solution for two threads, T_0 and T_1 with id 0 and 1, respectively
- We have seen that this solution does not work, Why?
- Both threads can acquire the lock as “while condition check” and “setting the flag” is non-atomic

Buggy #2

```
int flag[2] = {0,0};
void lock (int id)   /*id = 0 or 1 */
{
    flag[id] = 1;
    while(flag[id ^ 1]); // ^ → XOR
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Does this solution work?

Buggy #2

```
int flag[2] = {0,0};
void lock (int id)   /*id = 0 or 1 */
{
    flag[id] = 1;
    while(flag[id ^ 1]);  // ^ → XOR
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Does this solution work?
- No, as this can lead to a deadlock (flag[0] = flag[1] = 1) In other words the "progress" requirement is not met
- Progress: If no one has acquired the lock and there are contending threads, one of the threads must acquire the lock within a finite time

Buggy #3

```
int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   while(turn == id ^ 1));
}
void unlock (int id)
{
    turn = id ^ 1;
}
```

- Assuming T_0 invokes lock( ) first, does the solution provide mutual exclusion?

Buggy #3

```
int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   while(turn == id ^ 1));
}
void unlock (int id)
{
    turn = id ^ 1;
}
```

- Assuming T_0 invokes lock( ) first, does the solution provide mutual exclusion?
- Yes it does, but there is another issue with this solution - two threads must request the lock in an alternate manner
- Progress requirement is not met
  - Argument: one of the threads stuck in an infinite loop

Buggy #4

```
int flag[2] = {0,0};   int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   turn  = id ^1;
   flag[id] = 1;
   while(flag[id ^ 1]) && turn == (id ^1));
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Why this solution does not work?

Buggy #4

```
int flag[2] = {0,0};   int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   turn  = id ^1;
   flag[id] = 1;
   while(flag[id ^ 1]) && turn == (id ^1));
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Why this solution does not work?
- Mutual exclusion is not satisfied if T_0 context switched after setting the turn = 1 and T_1 acquires the lock (and sets turn = 0 in the process which allows T_0 to acquire the lock)

Attempt #5 (Peterson's solution)

```
int flag[2] = {0,0};   int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   flag[id] = 1;
   turn  = id ^1;
   while(flag[id ^ 1]) && turn == (id ^1));
}
void unlock (int id)
{
    flag[id] = 0;
}
```

- Homework: Prove that mutual exclusion is guaranteed
- What about fairness?

Attempt #5 (Peterson's solution)

```
int flag[2] = {0,0};   int turn = 0;
void lock (int id)    /*id = 0 or 1 */
{
   flag[id] = 1;
   turn  = id ^1;
   while(flag[id ^ 1]) && turn == (id ^1));
}
void unlock (int id)
{
     flag[id] = 0;
}
```

- Homework: Prove that mutual exclusion is guaranteed
- What about fairness?
- The lock is fair because if two threads are contending, they acquire the lock in an alternate manner
- Extending the solution to N threads is possible

Semaphores

- Mutual exclusion techniques allows exactly one thread to access the critical section which can be restrictive
- Consider a scenario when a finite array of size N is accessed from a set of producer and consumer threads. In this case,
    - At most N concurrent producers are allowed if array is empty
    - At most N concurrent consumers are allowed if array is full
    - If we use mutual exclusion techniques, only one producer or consumer is allowed at any point of time

Operations on semaphore

```
struct semaphore{
                 int value;
                 spinlock_t *lock;
                 queue *waitQ;
               }sem_t;

// Operations

sem_init(sem_t *sem, int init_value);
sem_wait(sem_t *sem);
sem_post(sem_t *sem);
```

- Semaphores can be initialized by passing an initial value
- *sem_wait* waits (if required) till the value becomes +ve and returns after decrementing the value
- *sem_post* increments the value and wakes up a waiting context
- Other notations: P-V, down-up, wait-signal

Unix semaphores

```
#include <semaphore.h>

main( ){
    sem_t  s;
    int K = 5;
    sem_init(&s, 0, K);
    sem_wait(&s);
    sem_post(&s);
}
```

- Can be used to in a multi-threaded process or across multiple processes
- If second argument is 0, the semaphore can be used from multiple threads
- Semaphores initialized with value = 1 (third argument) is called a binary semaphore and can be used to implement locks
- Initialize: sem_init(s, 0, 1)
  lock:sem_wait(s), unlock: sem_post(s)

Semaphore usage example: wait for child

```
child( ){
    ...
    sem_post(s);
    exit(0);
}
int main (void ){
        sem_init(s, 0);
        if(fork( ) == 0)
            child( );
        sem_wait(s);
}
```

- Assume that the semaphore is accessible from multiple processes, value initialized to zero
- If parent is scheduled after the child creation, it waits till child finishes
- If child is scheduled and exits before parent, parent does not wait for the semaphore

Semaphore usage example: ordering

```
A=0; B=0;

Thread-0 {
    A = 1;
    printf("B = %d\n", B);
}

Thread-1 {
        B=1;
        printf("A = %d\n", A);
}
```

- What are the possible outputs?

Semaphore usage example: ordering

```
A=0; B=0;

Thread-0 {
    A = 1;
    printf("B = %d\n", B);
}

Thread-1 {
        B=1;
        printf("A = %d\n", A);
}
```

- What are the possible outputs?
- (A = 1, B= 1), (A = 1, B = 0), (A = 0, B=1)
- How to guarantee A = 1, B= 1?

Semaphore usage example: ordering

```
sem_init(s1, 0);
A=0; B=0;
Thread - 0 {
    A = 1;
    sem_wait(s1);
    printf("B = %d\n", B);
}
Thread - 1 {
        B=1;
        sem_post(s1);
        printf("A = %d\n", A);
}
```

- What are the possible outputs?

Semaphore usage example: ordering

```
sem_init(s1, 0);
A=0; B=0;
Thread - 0 {
    A = 1;
    sem_wait(s1);
    printf("B = %d\n", B);
}
Thread - 1 {
        B=1;
        sem_post(s1);
        printf("A = %d\n", A);
}
```

- What are the possible outputs?
- (A = 1, B = 1), (A=0, B=1)
- How to guarantee A = 1, B= 1?

Ordering with two semaphores

```
sem_init(s1, 0);
sem_init(s2, 0)
A=0; B=0;

Thread - 0
{
    A = 1;
    sem_post(s1);
    sem_wait(s2);
    printf("%d\n", B);
}
```

- Waiting for each other guarantees desired output

```
Thread - 1
{
    B=1;
    sem_wait(s1);
    sem_post(s2);
    printf("%d\n", A);
}
```