Other Synchronization Primitives

03/31/2021

Professor Amanda Bienz

Infinitely Many Synchronization Primitives

- **Java Monitors**
  - Monitors were originally in the Mesa language
  - Per-object locks and condition variables
  - Locks/unlocks on enter/leave a ‘synchronized’ method or block
  - Conditions occur with o.wait(), o.notify(), o.notifyAll()
- **Transactions - database-style**
  - “Begin transaction” and “end transaction” on data locations
  - Generally a set of database rows
  - Some processors support it on memory locations
- **Today : More about Semaphores and introducing Read-Write Locks**

Semaphore API in C

```c
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // initialize s to the value 1
```

- Include the semaphore header file
- Declare a semaphore s
- sem_init(&s, 0, 1) : initializes the semaphore to 1 and indicates that the semaphore is shared between **threads in the same process**
    - Alternatively sem_init(&s, 1, 1) indicates semaphore is shared between processes with no shared memory

Wait

```
int sem_wait(sem_t *s) {
  decrement the value of semaphore s by one
  wait if value of semaphore s is negative
}
```

- sem_wait()
  - If the value of the semaphore was one or higher when called sem_wait(), return right away
  - The caller will suspend execution waiting for a subsequent post
  - When negative, the value of the semaphore is equal to the number of waiting threads

Post (or Signal)

```
int sem_post(sem_t *s) {
  increment the value of semaphore s by one
  if there are one or more threads waiting, wake one
}
```

- sem_post()
  - Simply increment the value of the semaphore
  - If there is a thread waiting to be woken, **wake** one of them up

Binary Semaphores (Locks)

```
sem_t m;
sem_init(&m, 0, X); // initialize semaphore to X; what should X be?

sem_wait(&m);
//critical section here
sem_post(&m);
```

- What should X be?
    - The initial value should be 1

Thread Trace : Single Thread Using Semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	call sema_wait()	
0	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	
1	sem_post() returns	

Thread Trace : Two Threads Using a Semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() retruns	Running		Ready
0	(crit set: begin)	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem < 0)→sleep	sleeping
-1		Running	*Switch → T0*	sleeping
-1	(crit sect: end)	Running		sleeping
-1	call sem_post()	Running		sleeping
0	increment sem	Running		sleeping
0	wake(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	*Interrupt; Switch → T1*	Ready		Running
0		Ready	sem_wait() retruns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

Semaphores as Condition Variables

```c
sem_t s;

void *
child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int
main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

A Parent Waiting For Its Child

```
parent: begin
child
parent: end
```

The execution result

- What should X be?
  - The value of semaphore should be set to a 0

Thread Trace : Parent Waiting for Child (Case 1)

- The parent call sem_wait() before the child has called sem_post()

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	`call sem_wait()`	Running		Ready
-1	`decrement sem`	Running		Ready
-1	`(sem < 0)→sleep`	sleeping		Ready
-1	*Switch→Child*	sleeping	`child runs`	Running
-1		sleeping	`call sem_post()`	Running
0		sleeping	`increment sem`	Running
0		Ready	`wake(Parent)`	Running
0		Ready	`sem_post() returns`	Running
0		Ready	*Interrupt; Switch→Parent*	Ready
0	`sem_wait() retruns`	Running		Ready

Thread Trace : Parent Waiting for Child (Case 2)

- The child runs to completion before the parent call sem_wait()

Value	Parent	State	Child	State
0	`Create(Child)`	Running	*(Child exists; is runnable)*	Ready
0	*`Interrupt; switch→Child`*	Ready	`child runs`	Running
0		Ready	`call sem_post()`	Running
1		Ready	`increment sem`	Running
1		Ready	`wake(nobody)`	Running
1		Ready	`sem_post() returns`	Running
1	`parent runs`	Running	*`Interrupt; Switch→Parent`*	Ready
1	`call sem_wait()`	Running		Ready
0	`decrement sem`	Running		Ready
0	`(sem<0)→awake`	Running		Ready
0	`sem_wait() retruns`	Running		Ready

The Producer/Consumer (Bounded-Buffer) Problem

- Producer : put() interface
  - Wait for buffer to become empty in order to put data into it
- Consumer : get() interface
  - Wait for a buffer to become filled before using it

```c
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
    buffer[fill] = value;      // line f1
    fill = (fill + 1) % MAX;   // line f2
}

int get() {
    int tmp = buffer[use];     // line g1
    use = (use + 1) % MAX;     // line g2
    return tmp;
}
```

The Producer/Consumer (Bounded-Buffer) Problem

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);     // line P1
        put(i);               // line P2
        sem_post(&full);      // line P3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);      // line C1
        tmp = get();          // line C2
        sem_post(&empty);     // line C3
        printf("%d\n", tmp);
    }
}
...
```

First Attempt: Adding the Full and Empty Conditions

The Producer/Consumer (Bounded-Buffer) Problem

```
int main(int argc, char *argv[]) {
    // …
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with…
    sem_init(&full, 0, 0);    // … and 0 are full
    // …
}
```

First Attempt: Adding the Full and Empty Conditions (Cont.)

- Imagine the MAX is greater than 1
  - If there are multiple producers, race condition can happen at line f1
  - It means that old data is overwritten
- We've forgotten **mutual exclusion**
  - The filling of a buffer and incrementing of the index into the buffer is a **critical section**

A Solution : Adding Mutual Exclusion

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);       // line p0 (NEW LINE)
        sem_wait(&empty);       // line p1
        put(i);                 // line p2
        sem_post(&full);        // line p3
        sem_post(&mutex);       // line p4 (NEW LINE)
    }
}

```

(Cont.)

Adding Mutual Exclusion (Incorrectly)

A Solution : Adding Mutual Exclusion

```
(Cont.)
void *consumer(void *arg) {
        int i;
        for (i = 0; i < loops; i++) {
                sem_wait(&mutex);       // line c0 (NEW LINE)
                sem_wait(&full);        // line c1
                int tmp = get();        // line c2
                sem_post(&empty);       // line c3
                sem_post(&mutex);       // line c4 (NEW LINE)
                printf("%d\n", tmp);
        }
}
```

Adding Mutual Exclusion (Incorrectly)

A Solution : Adding Mutual Exclusion (Cont.)

- Imagine two threads : one producer and one consumer
  - The consumer acquires the mutex (line c0)
  - The consumer calls sem_wait() on the full semaphore (line c1)
  - The consumer is blocked and yields the CPU
    - The consumer still holds the mutex!
  - The producer calls sem_wait() on the binary mutex semaphore (p0)
  - The producer is now stuck waiting too (**deadlock**)

Finally, A Working Solution

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);      // line p1
        sem_wait(&mutex);      // line p1.5 (MOVED MUTEX HERE...)
        put(i);                // line p2
        sem_post(&mutex);      // line p2.5 (... AND HERE)
        sem_post(&full);       // line p3
    }
}

(Cont.)
```

Adding Mutual Exclusion (Correctly)

Finally, A Working Solution

```
(Cont.)
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);        // line c1
        sem_wait(&mutex);       // line c1.5 (MOVED MUTEX HERE…)
        int tmp = get();        // line c2
        sem_post(&mutex);       // line c2.5 (… AND HERE)
    sem_post(&empty);       // line c3
    printf(“%d\n”, tmp);
}
}

int main(int argc, char *argv[]) {
// …
sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with …
sem_init(&full, 0, 0);    // ... and 0 are full
sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
// …
}
```

Adding Mutual Exclusion (Correctly)

Semaphore vs Condition Variable

- Semaphores can act as condition variables
- What are the differences?
- Example : if buffer has 10 free spaces
    - Condition variable : wake up all consumers and if a consumer is in the first thread, it calls get()
    - Semaphore : Value of semaphore is 10; will be greater than 0 for first 10 threads

Reader-Writer Locks

- Imagine a number of concurrent list operations, include inserts and simple lookups
  - **Insert :**
    - Change state of list
    - Traditional critical section makes sense
  - **Lookup:**
    - Simply read the data structure
    - As long as we can guarantee no insert, we can allow many lookups to proceed concurrently

This special type of lock is known as a reader-write lock.

Reader-Writer Locks

- Only single writer can acquire the lock
- Once reader has a read lock:
  - More readers will be allowed to acquire read lock, also
  - Writer will have to wait until all readers are finished

```
typedef struct _rwlock_t {
    sem_t lock;  // binary semaphore (basic lock)
    sem_t writelock; // used to allow ONE writer or MANY readers
    int readers;     // count of readers reading in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
…
```

Reader-Writer Locks (Cont.)

```c
        rw->readers++;
        if (rw->readers == 1)
            sem_wait(&rw->writelock); // first reader acquires writelock
        sem_post(&rw->lock);
    }

    void rwlock_release_readlock(rwlock_t *rw) {
        sem_wait(&rw->lock);
        rw->readers--;
        if (rw->readers == 0)
            sem_post(&rw->writelock); // last reader releases writelock
        sem_post(&rw->lock);
    }

    void rwlock_acquire_writelock(rwlock_t *rw) {
        sem_wait(&rw->writelock);
    }

    void rwlock_release_writelock(rwlock_t *rw) {
        sem_post(&rw->writelock);
    }
```

A Reader-Writer Lock (Cont.)

- The reader-write locks have fairness problem
  - It would be relatively easy for the reader to start the writer
  - How to prevent more readers from entering the lock once a writer is waiting?