Sistemas Operativos: Concurrency

Semaphores

Pedro F. Souto (pfs@fe.up.pt)

April 12, 2016

Roadmap

What is a semaphore?

Mutual Exclusion with Semaphores

"Joining" with Semaphores

Bounded Buffer with Semaphores

Introduction

- A semaphore is a powerful synchronization primitive. It can be used for:

  Mutual Exclusion like mutexes/locks
  Synchronization without busy waiting like condition variables

- A semaphore is a kind of a counter that supports two operations:

  up actually, `sem_post()` in `libpthreads`
  down actually, `sem_wait()` in `libpthreads`

  whose semantics are slightly unusual.

Definition and `libpthread` API (1/2)

```
1   int sem_wait(sem_t *s) {
2       decrement the value of semaphore s by one
3       wait if value of semaphore s is negative
4   }
5
6   int sem_post(sem_t *s) {
7       increment the value of semaphore s by one
8       if there are one or more threads waiting, wake one
9   }
```

- In addition, to incrementing/decrementing the semaphores value
  - When a thread calls `sem_wait()` it may

    either return immediately, if its value was positive upon calling

    or block, otherwise
  - When a thread calls `sem_post()` it:
    - unblocks one thread, if some is blocked on the semaphore;
- We can think that each each semaphore has a queue for waiting threads.
- When a semaphore value is negative, it gives the number of waiting threads

`libpthread` API (2/2)

- The `libpthread` API defines also an operation for initializing a semaphore

```
1   #include <semaphore.h>
2   sem_t s;
3   sem_init(&s, 0, 1);
```

- Indeed, depending on the problem to solve, we may need to initialize a semaphore with a proper value;
- Using semaphores for solving concurrency problems requires some ingenuity, so let's look at some examples

Roadmap

What is a semaphore?

Mutual Exclusion with Semaphores

"Joining" with Semaphores

Bounded Buffer with Semaphores

Application: Ensuring Mutual Exclusion

```
1   sem_t m;
2   sem_init(&m, 0, X); // initialize semaphore to X;
3
4   sem_wait(&m);
5   // critical section here
6   sem_post(&m);
```

Question What should the initial value of the semaphore be?

Application: Ensuring Mutual Exclusion

```
1   sem_t m;
2   sem_init(&m, 0, X); // initialize semaphore to X;
3
4   sem_wait(&m);
5   // critical section here
6   sem_post(&m);
```

Question What should the initial value of the semaphore be?

- Remember that we want the code to be executed in mutual exclusion.

Mutual Exclusion Execution

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: 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()` returns	Running
0		Ready	`(crit sect)`	Running
0		Ready	call `sem_post()`	Running
1		Ready	`sem_post()` returns	Running

Roadmap

What is a semaphore?

Mutual Exclusion with Semaphores

"Joining" with Semaphores

Bounded Buffer with Semaphores

Example: Simple Synchronization (Joining)

Problem A thread must wait for another one

Solution

Example: Simple Synchronization (Joining)

Problem A thread must wait for another one

Solution

```
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;
}
```

Question What should the initial value of the semaphore be?

Joining Execution 1

- The waiting thread calls `sem_wait()` before the other calls `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()` returns	Running		Ready

Joining Execution 2

- The signaling thread calls `sem_post()` before the other calls `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()` returns	Running		Ready

Roadmap

What is a semaphore?

Mutual Exclusion with Semaphores

"Joining" with Semaphores

Bounded Buffer with Semaphores

Bounded Buffer with Semaphores

```
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;
}
```

Bounded Buffer with Semaphores

```
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;
}
```

Question How to make this thread safe?

Bounded Buffer with Semaphores

```
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;
}
```

Question How to make this thread safe?

Idea Use 2 semaphores just like in the solution based on condition variables

`empty` which counts the number of empty positions in the BB

Question who should wait on this semaphore?

Bounded Buffer with Semaphores

```
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;
}
```

Question How to make this thread safe?

Idea Use 2 semaphores just like in the solution based on condition variables

- `empty` which counts the number of empty positions in the BB
  - **Question** who should wait on this semaphore?
- `full` which counts the number of full positions in the BB

Bound Buffer with Semaphores: 1st Try (1/2)

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           sem_wait(&empty);           // line P1
8           put(i);                     // line P2
9           sem_post(&full);            // line P3
10      }
11  }
12
13  void *consumer(void *arg) {
14      int i, tmp = 0;
15      while (tmp != -1) {
16          sem_wait(&full);            // line C1
17          tmp = get();                // line C2
18          sem_post(&empty);           // line C3
19          printf("%d\n", tmp);
20      }
21  }
22
23  int main(int argc, char *argv[]) {
24      // ...
25      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26      sem_init(&full, 0, 0);    // ... and 0 are full
27      // ...
28  }
```

Bound Buffer with Semaphores: 1st Try (2/2)

Question What is wrong with this solution?

Bound Buffer with Semaphores: 1st Try (2/2)

Question What is wrong with this solution?

Answer It does work properly for `MAX=1`

Bound Buffer with Semaphores: 1st Try (2/2)

Question What is wrong with this solution?

Answer It does work properly for `MAX=1`

But what if `MAX>1`?

Bound Buffer with Semaphores: 1st Try (2/2)

Question What is wrong with this solution?

Answer It does work properly for `MAX=1`

But what if `MAX>1`?

What if multiple threads try to access the BB simultaneously?

Bound Buffer with Semaphores: 1st Try (2/2)

Question What is wrong with this solution?

Answer It does work properly for `MAX=1`

But what if `MAX>1`?

What if multiple threads try to access the BB simultaneously?

Solution?

Bound Buffer with Semaphores: 1st Try (2/2)

Question What is wrong with this solution?

Answer It does work properly for `MAX=1`

But what if `MAX>1`?

What if multiple threads try to access the BB simultaneously?

Solution?

- Let's add a semaphore to ensure mutual exclusion

Bounded Buffer with Semaphores: 2nd Try (1/2)

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           sem_wait(&mutex);           // line p0 (NEW LINE)
9           sem_wait(&empty);           // line p1
10          put(i);                     // line p2
11          sem_post(&full);            // line p3
12          sem_post(&mutex);           // line p4 (NEW LINE)
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&mutex);           // line c0 (NEW LINE)
20          sem_wait(&full);            // line c1
21          int tmp = get();            // line c2
22          sem_post(&empty);           // line c3
23          sem_post(&mutex);           // line c4 (NEW LINE)
24          printf("%d\n", tmp);
25      }
26  }
```

Bounded Buffer with Semaphores: 2nd Try (2/2)

Question What is the problem now?

Bounded Buffer with Semaphores: 2nd Try (2/2)

Question What is the problem now?

What if a consumer starts before any producer?

Bounded Buffer with Semaphores: 2nd Try (2/2)

Question What is the problem now?

What if a consumer starts before any producer?

Deadlock ensues:

- The consumer waits for a producer to add an item to the BB
- But the producer cannot do it, because it cannot enter the CS without the consumer releasing the mutex

Generally a deadlock is a race-condition in which each thread of a set (of threads) is waiting for an event that can be generated only by another thread in this set.

Bounded Buffer with Semaphores: 2nd Try (2/2)

Question What is the problem now?

What if a consumer starts before any producer?

Deadlock ensues:

- The consumer waits for a producer to add an item to the BB
- But the producer cannot do it, because it cannot enter the CS without the consumer releasing the mutex

Generally a deadlock is a race-condition in which each thread of a set (of threads) is waiting for an event that can be generated only by another thread in this set.

How to fix this?

Bounded Buffer with Semaphores: 2nd Try (2/2)

Question What is the problem now?

What if a consumer starts before any producer?

Deadlock ensues:

- The consumer waits for a producer to add an item to the BB
- But the producer cannot do it, because it cannot enter the CS without the consumer releasing the mutex

Generally a deadlock is a race-condition in which each thread of a set (of threads) is waiting for an event that can be generated only by another thread in this set.

How to fix this?

Answer Acquire the mutex just before accessing the BB

Bounded Buffers with Semaphores: A Solution

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           sem_wait(&empty);           // line p1
9           sem_wait(&mutex);           // line p1.5 (MOVED MUTEX HERE...)
10          put(i);                     // line p2
11          sem_post(&mutex);           // line p2.5 (... AND HERE)
12          sem_post(&full);            // line p3
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&full);            // line c1
20          sem_wait(&mutex);           // line c1.5 (MOVED MUTEX HERE...)
21          int tmp = get();            // line c2
22          sem_post(&mutex);           // line c2.5 (... AND HERE)
23          sem_post(&empty);           // line c3
24          printf("%d\n", tmp);
25      }
26  }
```