Universidade Estadual de Campinas
Instituto de Computação

MC504 Sistemas Operacionais

T22 Semaphores

Referência principal
Ch. 31 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 15 de outubro de 2018

Arthur João Catto, PhD

2º semestre de 2018

Semaphores: A Definition

- A semaphore is a synchronization object with an integer value that, in POSIX, can be manipulated by two routines: `sem_wait()` and `sem_post()`.

- Before being used, a semaphore must be initialized and this initial value will determine its behavior, as we will see next.

```
#include "semaphore.h"

sem_t s;
sem_init(&s, 0, 1);
```

On linux, “semaphore.h” defaults to <semaphore.h>.
On macOS it loads a wrapper that simulates the POSIX API.

- Here, a semaphore `s` is created and initialized to `1` (the third argument of the call).

- The second argument is zero to indicate that the semaphore will be shared among threads in the same process.
  - A semaphore can also be shared among several processes, and this is what other values of this second argument would indicate.

Semantics of sem_wait()

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

- sem_wait() is implemented atomically.
- It decrements the value of the semaphore and returns immediately if the result is non-negative.
- Otherwise, it will cause the caller to sleep until awaken by a subsequent sem_post() call from another thread.

Semantics of sem_post()

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

- sem_post() is also implemented atomically.
- It increments the semaphore and returns immediately if its associated waiting list is empty.
- Otherwise, it will also awake a waiting thread before returning.

Binary Semaphores (Locks)

- A binary semaphore functions just like a lock

```
sem_t bs;
sem_init(&bs, 0, X) ; // initialize semaphore to X. What should X be?

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

- Look back at the definition of `sem_wait()` and `sem_post()` and choose the right value for `X`.

- To discuss the functioning of the binary semaphore, let us examine a scenario with two threads.

Thread trace: a single thread uses the semaphore

- In this case, there are two threads, but Thread 0 runs without interruption.

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait( )`	
0	`sem_wait( )` returns	
0	(critical section)	
0	call `sem_post( )`	
1	`sem_post( )` returns	

Thread trace: two threads use the semaphore

- In this case, there are two threads, which compete for the semaphore.
- Thread 0 is interrupted twice by the system.
- Experiment with scenarios of your choice to make sure you understand the model.

	Thread 0		Thread 1	
Value	state	action	state	action
1	Running		Ready	
1	Running	call sem_wait ()	Ready	
0	Running	sem_wait() returns	Ready	
0	Running	(crit sect: begin)	Ready	
0	Ready	*Interrupt: switch → T1*	Running	
0	Ready		Running	call sem_wait ()
-1	Ready		Running	decrement sem
-1	Ready		Sleeping	(sem <0) → asleep
-1	Running		Sleeping	*Switch → T0*
-1	Running	(crit sect: end)	Sleeping	
-1	Running	call sem_post ()	Sleeping	
0	Running	increment sem	Sleeping	
0	Running	wake (T1)	Ready	
0	Running	sem_post() returns	Ready	
0	Ready	*Interrupt: switch → T1*	Running	
0	Ready		Running	sem_wait() returns
0	Ready		Running	*(critical section)*
0	Ready		Running	call sem_post ()
1	Ready		Running	sem_post() returns

Semaphore as Condition Variable: Parent Waiting for Child

- We have done this before: a thread creates another thread and then waits until it finishes.
- What should the value of X in line 8 be to achieve the desired functionality?
- Again, let us examine two thread traces to get acquainted with the algorithm.

```
1.  sem_t s;

2.  void *child(void *arg) {
3.      printf("child\n");
4.      sem_post(&s); // child is done
5.      return NULL;
6.  }

7.  int main(int arge, char *argv[]) {
8.      sem_init(&s, 0, X); // what should X be?
9.      printf("parent: begin\n");
10.     pthread_t c;
11.     mythread_create(&c, NULL, child, NULL);
12.     sem_wait(&s);   // wait here for child
13.     printf("parent: end\n");
14.     return 0;
15. }
```

Thread trace: parent waiting for child (case 1)

- In this case, we assume that the parent thread is not interrupted after having created the child.
- Thus, the parent calls sem_wait() before the child 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) ⟶ asleep	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

Thread trace: parent waiting for child (case 2)

- In this case, we assume that the parent thread is interrupted just after having created the child.
- Thus, the child calls sem_post() before the parent 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

Solving the Producer / Consumer Problem

- In this case we will use what is called a ***counting semaphore***.
- We will introduce two semaphores, empty and full, that the threads will use to indicate that a buffer entry has been emptied or filled, respectively.

```
sem_t empty;
sem_t full;

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
    // ...
```

The put() and get() routines

- These are essentially the same routines we designed for our solution using condition variables.

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

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

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

Adding full and empty semaphores

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        sem_wait(&empty) ;          // p1
        put(i);                     // p2
        sem_post(&full);            // p3
    }
}

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

- Assume MAX = 1, one producer and one consumer.
  - Does it work?
- Assume MAX = 10, one producer and one consumer.
  - Does it work?
- Assume more than one producers and consumers.
  - Does it work?

Trying to implement mutual exclusion

- When there are multiple producers or consumers a race condition arises in put() and get().
- We can try to prevent it using a binary semaphore to implement mutual exclusion among those calls.
- We'll create it in main() and later use it in producer() and consumer().

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

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 mutex to producer() and consumer()

```c
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

```c
void *consumer(void *arg) {
    for (int i =0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get( );
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

- Here we added mutex to producer() and consumer() to prevent the race condition on the buffer.
- Is our solution correct now?

How to avoid deadlock

- Our proposed scheme does not work because **producer()** and **consumer()** may get trapped, each one waiting for a condition that only the other could provide.
- This is a classical problem known as **deadlock**, which we will study in detail later.
- In this case, the problem can be solved by reducing the scope of the mutual exclusion between **producer()** and **consumer()**, as shown in the next slide.
- The result is a simple and working bounded buffer, a commonly-used pattern in multithreaded programs.

Avoiding deadlock

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);
        put(i);
        sem_post(&mutex);
        sem_post(&full);
    }
}
```

```
void *consumer(void *arg) {
    for (int i =0; i < loops; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        int tmp = get( );
        sem_post(&mutex);
        sem_post(&empty);
        printf("%d\n", tmp);
    }
}
```

- Here we reduce the scope of mutex in producer() and consumer() to prevent the deadlock.
- Is our solution correct now?
- Why do producer and consumer wait and signal different semaphores?
- Is order of waits important?
- Is order of signals important?
- What if we have 2 producers or 2 consumers? Do we need to change anything?
- Can we use semaphores for FIFO ordering?

Reader-Writer Locks

- In some cases we need a more flexible primitive that takes into account that different accesses to a data structure might require different kinds of locking.
- For example, imagine a number of concurrent list operations, including inserts and simple lookups.
  - Inserts change the state of the list and thus must be protected by e.g. a critical section.
  - On the other hand, lookups simply read the list and, as long as there is no simultaneous insert, many of them may proceed concurrently.
- The special type of lock that will support this type of operation is known as a **reader-writer lock**.

A simple reader-writer lock

```
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);
}
```

A simple reader-writer lock

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

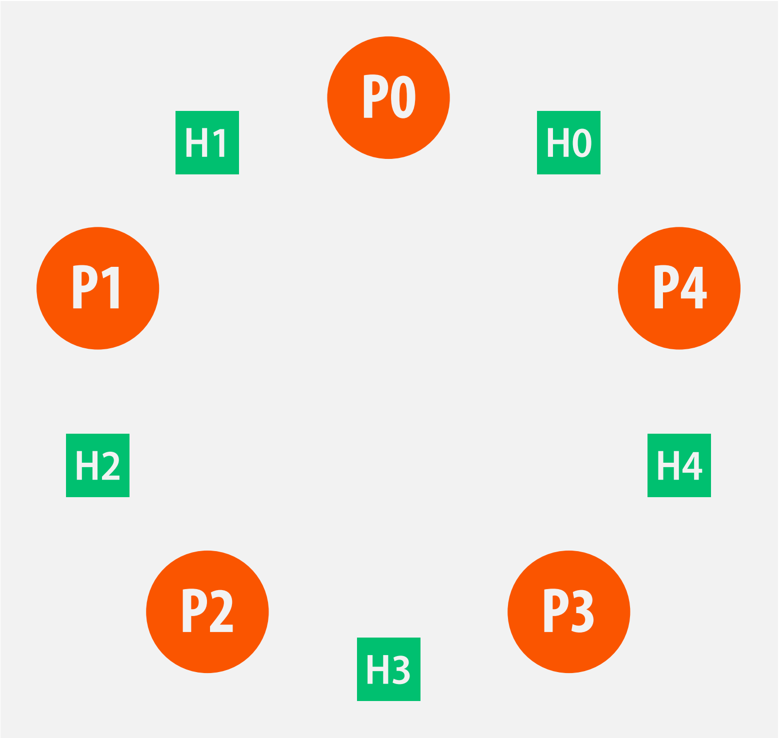
A simple reader-writer lock

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

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

- Examine the assumptions and the design of this solution.
  - Is it correct? Is it fair? Is it efficient?
- Can readers harm writers somehow? Can writers harm readers?

The Dining Philosophers



- Assume there are five "philosophers" sitting around a table.
- Between each pair of philosophers is a single hashi (and thus, five total).
- The philosophers each have times where they think, and don't need any hashis, and times when they eat.
- In order to eat, a philosopher needs two hashis, both the one on their left and the one on their right.
- The contention for these hashis, and the synchronization problems that ensue, are what makes this problem worth of being studied in concurrent programming.

Designing a solution

- From the problem description we can derive the basic loop of each philosopher:

```
while (1) {
      think();
      gethashis();
      eat();
      puthashis();
}
```

- The hashis are the shared resources that must be protected.
- Our solution must also satisfy the following requirements:
  - No deadlock
  - No starvation
  - High concurrency

Designing a solution

- To make it easier to reference the hashis, let us create two helper functions which, for a given philosopher p, calculate the indices of the hashis to his left and to his right.

```c
int left(int p)  { return p; }
int right(int p) { return (p + 1) % 5; }
```

- We will also associate a binary semaphore with each hashi

```c
sem_t hashis[5];

int main(int argc, char *argv[]) {
    // ...
    for (int i = 0; i < 5; i++)
        sem_init(&hashis[i], 0, 1);
    // ...
}
```

Designing gethashis() and puthashis() (version 1)

- This enables us to write a first version of the functions that implement the model

```
void gethashis(int p) {
    sem_wait(hashis[left(p)]);
    sem_wait(hashis[right(p)]);
}

void puthashis(int p) {
    sem_post(hashis[left(p)]);
    sem_post(hashis[right(p)]);
}
```

- How do you like it? Is it correct? Does it satisfy the additional requirements?

Breaking the dependency in gethashis()

- We can break the circular wait in gethashis() by imposing a different policy for one of the philosophers (say, philosopher number 4)

```
void gethashis(int p) {
    if (p == 4) {
        sem_wait(hashis[right(p)]);
        sem_wait(hashis[left(p)]);
    } else {
        sem_wait(hashis[left(p)]);
        sem_wait(hashis[right(p)]);
    }
}
```

- Is that enough? Is it correct? Does it satisfy the additional requirements?