


```

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  }

```

| | | |
|------|------|------|
| 31.2 | Wait | Post |
|------|------|------|

```

1
sem_wait()
sem_wait()
sem_post()
sem_wait()
sem_wait()
post
sem_wait()

```

[D68b]

0#Z\$

| | | |
|------|---|-----------------------|
| 31.3 | | sem_wait()/sem_post() |
| | m | X X |

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

31.3

| | | | | |
|------------|------------|---|---|------------|
| sem_wait() | sem_post() | | 1 | |
| | | | 0 | sem_wait() |
| 0 | 0 | | 0 | |
| 0 | 0 | | | |
| sem_post() | | 1 | | |

31.1

##

| | 0 | 1 |
|---|------------|---|
| 1 | | |
| 1 | sem_wait() | |
| 0 | sem_wait() | |
| 0 | | |
| 0 | sem_post() | |
| 1 | sem_post() | |

0 sem_wait() sem_post() 1
 1 sem_wait() 1
 1 0
 sem_post() 0 1 1
 1 1
 31.2
 scheduler state 1
 0 1

##

| | 0 | | 1 | |
|---|------------|--|------------|--|
| 1 | | | | |
| 1 | sem_wait() | | | |
| 0 | sem_wait() | | | |
| 0 | | | | |
| 0 | T1 | | | |
| 0 | | | sem_wait() | |
| 1 | | | sem 1 | |
| 1 | | | (sem<0) | |
| 1 | | | T0 | |
| 1 | | | | |
| 1 | sem_post() | | | |
| 0 | sem | | | |
| 0 | T1 | | | |
| 0 | sem_post() | | | |
| 0 | T1 | | | |
| 0 | | | sem_wait() | |
| 0 | | | | |
| 0 | | | sem_post() | |
| 1 | | | sem_post() | |

binary semaphore

0/40/0
0/2/0

condition

condition variable

31.4

```

1  sem_t s;
2
3  void *
4  child(void *arg) {
5      printf("child\n");
6      sem_post(&s); // signal here: child is done
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     sem_init(&s, 0, X); // what should X be?
13     printf("parent: begin\n");
14     pthread_t c;
15     Pthread_create(c, NULL, child, NULL);
16     sem_wait(&s); // wait here for child
17     printf("parent: end\n");
18     return 0;
19 }
```

31.4

```

parent: begin
child
parent: end
```

sem_wait()

sem_post()

0

31.3 sem_wait()

sem_post() 1

0 0 1

sem_post() 0 sem_wait()

%#Z%

#

| | | | | |
|---|------------|--|------------------------------|--|
| | | | | |
| 0 | create(| | () | |
| 0 | sem_wait() | | | |
| 1 | sem 1 | | | |
| 1 | (sem<0) | | | |
| 1 | | | | |
| 1 | | | sem_post() | |
| 0 | | | sem 1 | |
| 0 | | | wake() | |
| 0 | | | sem_post() | |
| 0 | | | | |
| 0 | sem_wait() | | | |

sem_wait()

31.4

sem_post()

0

1

sem_wait()

1

1

0

sem_wait()

%#Z&

\$

| | | | | |
|---|------------|--|------------------------------|--|
| | | | | |
| 0 | create | | () | |
| 0 | | | | |
| 0 | | | sem_post() | |
| 1 | | | sem 1 | |
| 1 | | | wake() | |
| 1 | | | sem_post() | |
| 1 | | | | |
| 1 | sem_wait() | | | |
| 0 | sem 1 | | | |
| 0 | (sem>=0) | | | |
| 0 | sem_wait() | | | |

~~0/728~~

!

/ producer/consumer

[D72] 30

empty full

31.5 put() get() 31.6 /

```

1  int buffer[MAX];
2  int fill = 0;
3  int use = 0;
4
5  void put(int value) {
6      buffer[fill] = value;    // line f1
7      fill = (fill + 1) % MAX; // line f2
8  }
9
10 int get() {
11     int tmp = buffer[use];    // line g1
12     use = (use + 1) % MAX;    // line g2
13     return tmp;
14 }
```

31.5 put() get()

```

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  }

```

31.6 full empty

MAX=1

| | | | | | | |
|---|-------|-----------------|------------------|-----------------|----|-------|
| | | | | CPU | | |
| | C1 | sem_wait(&full) | full | 0 wait | | full |
| 1 | | | sem_post(&full) | | | |
| | | P1 | sem_wait(&empty) | | | |
| | empty | MAX | 1 | empty | 0 | |
| | | P3 | sem_post(&full) | full | 1 | 0 |
| | | | | | | |
| | | | | P1 | | empty |
| 0 | | | | sem_wait(&full) | c1 | |

| | | | | | |
|----|-----|--------|--------|-------|------|
| | MAX | 1 | MAX=10 | | |
| | | put() | get() | | |
| | | | | Pa | Pb |
| f1 | | fill=0 | | Pa | fill |
| | f1 | 0 | | put() | Pa |
| | | | | 1 | Pb |

31.7

```

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 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);    // ... and 0 are full
32     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
33     // ...
34 }

```

31.7

put()/get()

NEW LINE

c0

full

sem_wait() c1

CPU

sem_wait() p0

full

full

31.8

full empty

```

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 }
27
28 int main(int argc, char *argv[]) {
29     // ...
30     sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31     sem_init(&full, 0, 0);    // ... and 0 are full
32     sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33     // ...
34 }

```

31.8

%#Z

reader-writer lock

[CHP71] 31.9

| | |
|----------------------------|-----------------------|
| | rwlock_acquire_lock() |
| rwlock_release_writelock() | writelock |

```

1  typedef struct _rwlock_t {
2      sem_t lock;           // binary semaphore (basic lock)
3      sem_t writelock; // used to allow ONE writer or MANY readers
4      int    readers; // count of readers reading in critical section
5  } rwlock_t;
6
7  void rwlock_init(rwlock_t *rw) {
8      rw->readers = 0;
9      sem_init(&rw->lock, 0, 1);
10     sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->lock);
15     rw->readers++;
16     if (rw->readers == 1)
17         sem_wait(&rw->writelock); // first reader acquires writelock
18     sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0)
25         sem_post(&rw->writelock); // last reader releases writelock
26     sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30     sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34     sem_post(&rw->writelock);
35 }

```

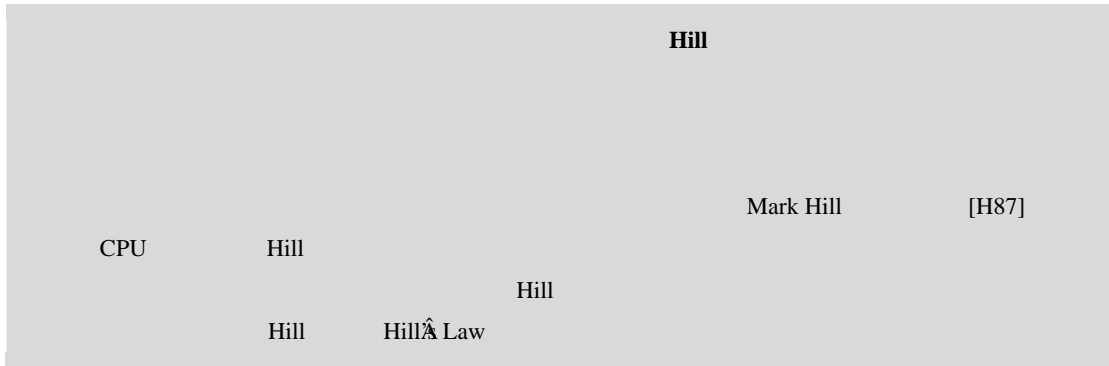
31.9 !

| | | |
|--|---------------------------|-----------|
| | lock | reader |
| | rwlock_acquire_readlock() | |
| | | writelock |

| | | |
|------------|------------|------|
| sem_wait() | sem_post() | lock |
|------------|------------|------|

writelock

sem_post()



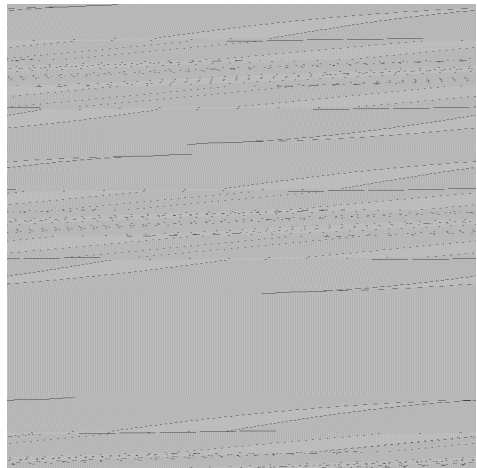
!

[CB08]

%#Z

dining philosopher's problem
[DHO71]

Dijkstra



31.10

5

5

31.10

```
while (1) {
    think();
```



```

1  void getforks() {
2      if (p == 4) {
3          sem_wait(forks[right(p)]);
4          sem_wait(forks[left(p)]);
5      } else {
6          sem_wait(forks[left(p)]);
7          sem_wait(forks[right(p)]);
8      }
9  }

```

cigarette smoker[^]A problem

sleeping barber problem

[D08]

%#Z)

Zemaphore

31.12

```

1  typedef struct  _Zem_t {
2      int value;
3      pthread_cond_t cond;
4      pthread_mutex_t lock;
5  } Zem_t;
6
7  // only one thread can call this
8  void Zem_init(Zem_t *s, int value) {
9      s->value = value;
10     Cond_init(&s->cond);
11     Mutex_init(&s->lock);
12 }
13
14 void Zem_wait(Zem_t *s) {
15     Mutex_lock(&s->lock);
16     while (s->value <= 0)
17         Cond_wait(&s->cond, &s->lock);
18     s->value--;
19     Mutex_unlock(&s->lock);
20 }
21
22 void Zem_post(Zem_t *s) {

```

```

23     Mutex_lock(&s->lock);
24     s->value++;
25     Cond_signal(&s->cond);
26     Mutex_unlock(&s->lock);
27 }

```

31.12

Zemahpore

Zemaphore Dijkstra

0

Linux

Lampson

[L83]

Windows

[B04]

$$\% \# \tilde{Z}^*$$

Allen Downey

[D08]

[B04] Implementing Condition Variables with Semaphores Andrew Birrell
December 2004

Birrell

[CB08] Real-world Concurrency Bryan Cantrill and Jeff Bonwick

ACM Queue. Volume 6, No. 5. September 2008

Sun

[CHP71] Concurrent Control with Readers and Writers

P.J. Courtois, F. Heymans, D.L. Parnas Communications of the ACM, 14:10, October 1971

[D59] A Note on Two Problems in Connexion with Graphs

E. W. Dijkstra

Numerische Mathematik 1, 269271, 1959

1959

[D68a] Go-to Statement Considered Harmful

E.W. Dijkstra

Communications of the ACM, volume 11(3): pages 147148, March 1968

[D68b] The Structure of the THE Multiprogramming System

E.W. Dijkstra

Communications of the ACM, volume 11(5), pages 341346, 1968

[D72] Information Streams Sharing a Finite Buffer

E.W. Dijkstra

Information Processing Letters 1: 179180, 1972

Dijkstra

Dijkstra

[D08] The Little Book of Semaphores

A.B. Downey

[DHO71] Hierarchical ordering of sequential processes

E.W. Dijkstra

[GR92] Transaction Processing: Concepts and Techniques Jim Gray and Andreas Reuter

Morgan Kaufmann, September 1992

| | | | | |
|---|---------|---------|----------|------|
| | 485 | 8.8 | | 1960 |
| OS | | | Dijkstra | |
| [H87] Aspects of Cache Memory and Instruction Buffer Performance Mark D. Hill | | | | |
| Ph.D. Dissertation, U.C. Berkeley, 1987 | | | | |
| Hill | | | | |
| [L83] Hints for Computer Systems Design Butler Lampson | | | | |
| ACM Operating Systems Review, 15:5, October 1983 | | | | |
| | Lampson | | | |
| signal() | | | | |
| | | Lampson | | |