

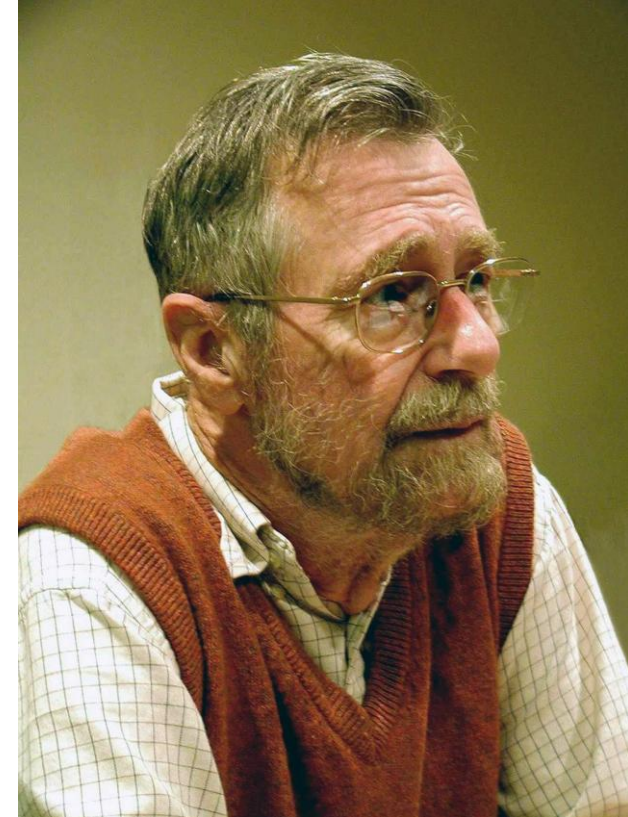
Semaphore

Operating System (OS)
안인규

강의자료는 KAIST 원유집 교수님 OS 강의 자료를 참고했습니다.

Semaphore:

- 동시성(concurrency) 문제들을 해결하기 위해 **lock** 과 **condition variable** 둘 다 필요
- Dijkstra와 그의 동료들이 이 둘을 통합할 수 있는 방법으로 Semaphore 제안 (Go to Statement Considered Harmful에서)
- One can use semaphores as both locks and condition variables



Edsger Wybe Dijkstra
(에츠허르 비버 데이크스트라)
1930.5.11~2002.8.6

Semaphore: A definition

- An object **with an integer value**
 - We can manipulate with two routines; `sem_wait()` and `sem_post()`.
 - Initialization

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

- Declare a semaphore `s` and initialize it to the value 1 (하나의 자원을 사용할 수 있다는 의미)
- The second argument, 0, indicates that the semaphore is shared between *threads in the same process*.
 - 0: Semaphore가 같은 프로세스 내의 thread들 간에만 공유됨
 - 0이 아닌 값: Semaphore가 여러 프로세스 간 공유됨 (예: `fork()`, Semaphore는 커널이 관리하는 `MAP_SHARED` 영역에...)

Semaphore: Interact with semaphore

- `sem_wait()`

```
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 }
```

- Semaphore 값을 1 감소, 그 값이 0보다 작으면 suspend execution (blocked state)
- When **positive**, the program operates the right way
- When **negative**, the value of the semaphore is equal to the number of waiting threads.

- `sem_post()`

```
1 int sem_post(sem_t *s) {  
2     increment the value of semaphore s by one  
3     if there are one or more threads waiting, wake one  
4 }
```

- Simply **increments** the value of the semaphore.
- If there is a thread waiting to be woken (Semaphore < 0), **wakes** one of them up.

Semaphore: Interact with semaphore (Cont.)

Time	T_1	T_2	T_3	T_4	Semaphore s
0					0
1	<code>sem_wait(s);</code>				-1
2		<code>sem_wait(s);</code>			-2
3			<code>sem_wait(s);</code>		-3
4				<code>sem_post(s);</code>	-2
5					...

Binary Semaphores (Locks)

- What should **x** be?

```
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);
```

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	

Binary Semaphores (Locks)

- What should **x** be?
 - The initial value should be **1**.

```
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);
```

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() returns	Running		Ready
0	(crit set: begin)	Running		Ready
0	<i>Interrupt; Switch → T1</i>	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem < 0) → sleep	sleeping
-1		Running	<i>Switch → T0</i>	sleeping
-1	(crit sect: end)	Running		sleeping
-1	call sem_post()	Running		sleeping
0	increment sem	Running		sleeping
0	wait(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	<i>Interrupt; Switch → T1</i>	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

- Semaphores for ordering
 - **One thread (Parent)** is waiting for something to happen
 - **Another thread (Child)** makes that something happen and signals that it has happened → Waking the waiting thread
 - We are thus using the semaphore as an ordering primitive (similar to our use of condition variables earlier).

Semaphores As Condition Variables

```
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 }
```

A Parent Waiting For Its Child

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

The execution result

- What should **x** be?

Semaphores As Condition Variables

```
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 }
```

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 is **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	<i>(Child exists; is runnable)</i>	Ready
0	call <code>sem_wait()</code>	Running		Ready
-1	decrement sem	Running		Ready
-1	<code>(sem < 0) → sleep</code>	sleeping		Ready
-1	<i>Switch → Child</i>	sleeping	child runs	Running
-1		sleeping	call <code>sem_post()</code>	Running
0		sleeping	increment sem	Running
0		Ready	wake(Parent)	Running
0		Ready	<code>sem_post()</code> returns	Running
0		Ready	<i>Interrupt; Switch → Parent</i>	Ready
0	<code>sem_wait()</code> 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	<i>(Child exists; is runnable)</i>	Ready
0	<i>Interrupt; switch→Child</i>	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	<i>Interrupt; Switch→Parent</i>	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 a buffer to become *empty* in order to put data into it.
- **Consumer:** `get()` interface
 - Wait for a buffer to become *filled* before using it.

```
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 }
```

Bounded-Buffer) Problem

```
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 }
```

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

The Producer/Consumer (Bounded-Buffer) Problem

Time	P1 (producer)	P2 (producer)
0	<code>sem_wait(&empty);</code>	
1		<code>sem_wait(&empty);</code>
2		<code>Put(i);</code>
3		<code>-> buffer[fill]=value;</code>
4	<code>Put(i);</code> Race condition!	
5	<code>-> buffer[fill]=value;</code>	
6	<code>-> fill=fill+1;</code>	
7		<code>-> fill=fill+1;</code>

Solution (Cont.)

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);           // p1
8          while (count == MAX)                 // p2
9              Pthread_cond_wait(&empty, &mutex); // p3
10         put(i);                               // p4
11         Pthread_cond_signal(&fill);          // p5
12         Pthread_mutex_unlock(&mutex);       // p6
13     }
14 }
15
```

```
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);           // c1
20         while (count == 0)                   // c2
21             Pthread_cond_wait(&fill, &mutex); // c3
22         int tmp = get();                      // c4
23         Pthread_cond_signal(&empty);        // c5
24         Pthread_mutex_unlock(&mutex);       // c6
25         printf("%d\n", tmp);
26     }
27 }
```

- Conditional Variable을 이용한 producer/Consumer Solution
- 하나의 Mutex로 Lock/Unlock이 관리

Mutual Exclusion

```
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 }
```

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

A Solution: Adding Mutual Exclusion (Cont.)

Time	P (producer)	C (consumer)	Mutex	Empty
0			1	0
1	<code>sem_wait(&mutex);</code>		0	0
2	<code>sem_wait(&empty);</code>		0	-1
3	Sleep (Switch to "C")		0	-1
4		<code>sem_wait(&mutex);</code>	-1	-1
5		Sleep (Dealock!)	-1	-1

A Working 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
```

Adding Mutual Exclusion (Correctly)

A Working Solution

```
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 }
35     ...
```

Adding Mutual Exclusion (Correctly)

Reader-Writer Locks

- Given shared resources (Linked list, hash table, ...), imagine a number of concurrent list operations, including **inserts** and simple **lookups**.
 - **Insert (=Write):**
 - Change the state of the list
 - A traditional critical section makes sense.
 - **Lookup (=Read):**
 - Simply *read* the data structure.
 - As long as we can guarantee that no insert is on-going, we can allow many lookups to proceed **concurrently**.

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

Reader-Writer Locks

- Only a **single writer** can acquire the lock.
- Once a reader has acquired a **read lock**,
 - **More readers** will be allowed to acquire the read lock too.
 - A writer will have to wait until all readers are finished.

```
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;
```

```
1. void rwlock_init(rwlock_t *rw) {
2.     rw->readers = 0;
3.     sem_init(&rw->lock, 0, 1);
4.     sem_init(&rw->writelock, 0, 1);
5. }
```

Reader-Writer Locks

```
1. void rwlock_acquire_readlock(rwlock_t *rw) {
2.     sem_wait(&rw->lock);
3.     rw->readers++;
4.     if (rw->readers == 1)
5.         sem_wait(&rw->writelock); // first reader acquires writelock
6.     sem_post(&rw->lock);
7. }
```

```
1. void rwlock_release_readlock(rwlock_t *rw) {
2.     sem_wait(&rw->lock);
3.     rw->readers--;
4.     if (rw->readers == 0)
5.         sem_post(&rw->writelock); // last reader releases writelock
6.     sem_post(&rw->lock);
7. }
```

```
1. void rwlock_acquire_writelock(rwlock_t *rw) {
2.     sem_wait(&rw->writelock);
3. }
```

```
1. void rwlock_release_writelock(rwlock_t *rw) {
2.     sem_post(&rw->writelock);
3. }
```

A Reader-Writer Locks (Cont.)

Read_count	Write_lock	lock	T_1 (Reader)	T_2 (Reader)	T_3 (Writer)
		0	<code>sem_wait(&rw->lock);</code>		
1			<code>rw->readers++;</code>		
	0		<code>sem_wait(&rw->writelock);</code>		
		1	<code>sem_post(&rw->lock);</code>		
				<code>sem_wait(&rw->lock);</code>	
2				<code>rw->readers++;</code>	
				<code>sem_post(&rw->lock);</code>	
	-1				<code>sem_wait(&rw->writelock);</code>
					???

A Reader-Writer Locks (Cont.)

`rwlock_acquire_readlock(rwlock_t *rw)`

Read_count	Write_lock	lock	T_1 (Reader)	T_2 (Reader)	T_3 (Writer)
		0	<code>sem_wait(&rw->lock);</code>		
1			<code>rw->readers++;</code>		
	0		<code>sem_wait(&rw->writelock);</code>		
		1	<code>sem_post(&rw->lock);</code>		
2				<code>sem_wait(&rw->lock);</code>	
				<code>rw->readers++;</code>	
				<code>sem_post(&rw->lock);</code>	
	-1		Read a shared resource	Read a shared resource	<code>rwlock_acquire_writelock</code>
					<code>sem_wait(&rw->writelock);</code>
					??? → Sleep!

A Reader-Writer Locks (Cont.)

`rwlock_acquire_readlock(rwlock_t *rw)`

Read_count	Write_lock	lock	T ₁ (Reader)	T ₂ (Reader)	T ₃ (Writer)
		0	<code>sem_wait(&rw->lock);</code>		
1			<code>rw->readers++;</code>		
	0		<code>sem_wait(&rw->writelock);</code>		
		1	<code>sem_post(&rw->lock);</code>		
				<code>sem_wait(&rw->lock);</code>	
2				<code>rw->readers++;</code>	
				<code>sem_post(&rw->lock);</code>	
	-1				<code>rwlock_acquire_writelock</code>
					<code>sem_wait(&rw->writelock);</code>
					??? → Sleep!

Read a shared resource

Read a shared resource

What if there comes **new readers** again and again ...

A Reader-Writer Locks (Cont.)

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

Reader-Writer Locks (Sol.)

```
1. void rwlock_acquire_readlock(rwlock_t *rw) {
2.     sem_wait(&rw->lock);
3.     rw->readers++;
4.     if (rw->readers == 1)
5.         sem_wait(&rw->writelock); // first reader acquires writelock
6.     sem_post(&rw->lock);
7. }
```

```
1. void rwlock_release_readlock(rwlock_t *rw) {
2.     sem_wait(&rw->lock);
3.     rw->readers--;
4.     if (rw->readers == 0)
5.         sem_post(&rw->writelock); // last reader releases writelock
6.     sem_post(&rw->lock);
7. }
```

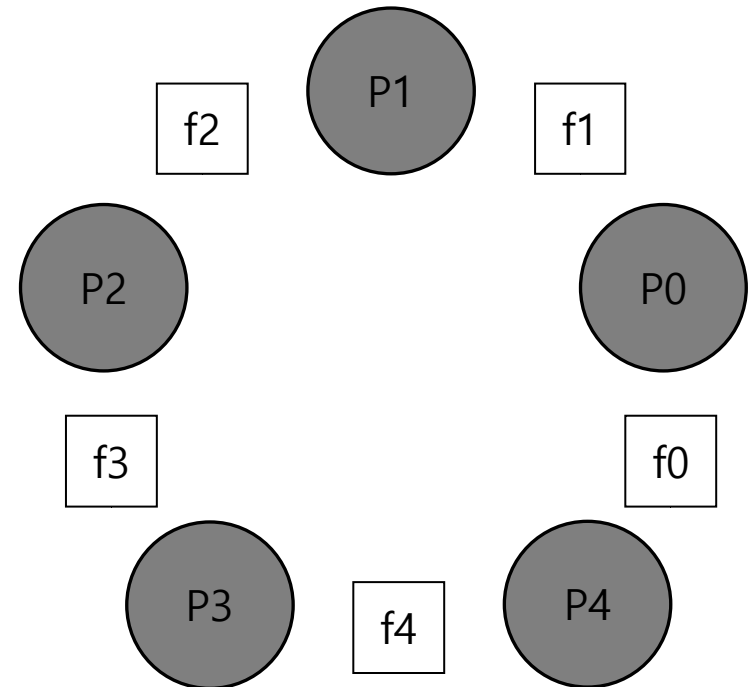
```
1. void rwlock_acquire_writelock(rwlock_t *rw) {
2.     sem_wait(&rw->writelock);
3. }
```

```
1. void rwlock_release_writelock(rwlock_t *rw) {
2.     sem_post(&rw->writelock);
3. }
```

The Dining Philosophers

- Assume there are five “**philosophers**” sitting around a table.
 - Between each pair of philosophers is a single fork (five total).
 - The philosophers each have times where they **think**, and don’t need any forks, and times where they **eat**.
 - In order to *eat*, a philosopher needs **two forks**, both the one on their *left* and the one on their *right*.
 - **The contention for these forks.**

```
while (1) {  
    think();  
    getforks();  
    eat();  
    putforks();  
}
```



The Dining Philosophers (Cont.)

- Key challenge
 - There is **no deadlock**.
 - **No** philosopher **starves** and never gets to eat.
 - **Concurrency** is high.

```
while (1) {  
    think();  
    getforks();  
    eat();  
    putforks();  
}
```

Basic loop of each philosopher

```
// helper functions  
int left(int p) { return p; }  
  
int right(int p) {  
    return (p + 1) % 5;  
}
```

Helper functions (Downey's solutions)

- Philosopher p wishes to refer to the fork on their left \rightarrow call `left(p)`.
- Philosopher p wishes to refer to the fork on their right \rightarrow call `right(p)`.

The Dining Philosophers (Cont.)

- We need some **semaphore**, one for each fork: `sem_t forks[5]`.

```
1  void getforks() {
2      sem_wait(forks[left(p)]);
3      sem_wait(forks[right(p)]);
4  }
5
6  void putforks() {
7      sem_post(forks[left(p)]);
8      sem_post(forks[right(p)]);
9  }
```

The `getforks()` and `putforks()` Routines (Broken Solution)

- **Deadlock** occur!
 - If each philosopher happens to **grab the fork on their left** before any philosopher can grab the fork on their right.
 - Each will be stuck *holding one fork* and waiting for another, *forever*.

A Solution: Breaking The Dependency

- Change **how forks are acquired**.
 - Let's assume that philosopher 4 acquire the forks in a *different order*.

```
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  }
```

- There is no situation where each philosopher grabs one fork and is stuck waiting for another. **The cycle of waiting is broken.**
- How to break the cycle of waiting in A Reader-Writer Lock?

Semaphore using condition variable: Zemaphores

```
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 }
```

Semaphore using condition variable: Zemaphores

```
1 void Zem_wait(Zem_t *s) {
2     Mutex_lock(&s->lock);
3     while (s->value <= 0)
4         Cond_wait(&s->cond, &s->lock);
5     s->value--;
6     Mutex_unlock(&s->lock);
7 }
```

How To Implement Semaphores (Cont.)

```
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 }
```

- Semaphore don't maintain the invariant that *the value of the semaphore*.
 - The value never be lower than zero.
 - This behavior is **easier** to implement and **matches** the current Linux implementation.