Analysis of semaphore implementations

```
void P(sema *s){
  pthread_mutex_lock(&s->lock);
  s->value--;
  while (s->value < 0){
    if (s->waiters < -1 * s->value){
      s->waiters++;
      pthread_cond_wait(&s->wait,&s->lock);
      s->waiters--;
    }else{
      break;
    }
  }
  pthread_mutex_unlock(&(s->lock));
  return;
}

void V(sema *s){
  pthread_mutex_lock(&s->lock);
  s->value++;
  if (s->value <= 0){
    pthread_cond_signal(&s->wait);
  }
  pthread_mutex_unlock(&(s->lock));
}
```

s->waiters is initialized to 0 and is non-negative because for every decrement operation there is a preceding increment operation. Thus, for any *s->value* >= 0 *s->waiters* < *-1 * s->value* is false. The following simplified implementation of P will be considered.

```
void P(sema *s){
  pthread_mutex_lock(&s->lock);
  s->value--;
  while (s->waiters < -1 * s->value){
    s->waiters++;
    pthread_cond_wait(&s->wait,&s->lock);
    s->waiters--;
  }
  }
  pthread_mutex_unlock(&(s->lock));
}
```

Wlog, let *s->value* == -2 and *s->waiters* == 2. Thread A call V, increments *s->value* to -1, and signals. Thread A continues running, calls P, decrements *s->value* to -2, and is not pushed onto the queue of waiting threads. Thread B is awakened by the signal from thread A, reacquires mutex and decrements *s->waiters* to 1. Because *s->value* == -2 and *s->waiters* == 1, thread B increments *s->waiters* to 2 and is pushed back onto the queue of waiting threads. Thus thread A “received” its own signal, avoided waiting, and the thread B was pushed back onto the waiting queue of threads.

The above behavior reduces the role of the scheduler and can lead to thread starvation. The behavior is avoided by guaranteeing the queuing of each thread that calls P (sem_wait), as provided in The Little Book of Semaphores by Allen B. Downey (Version 2.2.1):

```
void sem_wait(Semaphore *semaphore){
  mutex_lock(semaphore->mutex);
  semaphore->value--;
  if (semaphore->value < 0){
    do{
      cond_wait(semaphore->cond, semaphore->mutex);
    }while (semaphore->wakeups < 1);
    semaphore->wakeups--;
  }
  mutex_unlock(semaphore->mutex);
}

void sem_signal(Semaphore *semaphore){
  mutex_lock(semaphore->mutex);
  semaphore->value++;
  if (semaphore->value <= 0){
    semaphore->wakeups++;
    cond_signal(semaphore->cond);
  }
  mutex_unlock(semaphore->mutex);
}
```

The do...while loop guarantees the queuing of each thread that calls sem_wait. *semaphore->wakeups* provides the accounting to ensure that every signal does not awaken more than one thread due to the specification of phtread.