CS 537 Part 2 Study Guide by Leping Li

Tuesday, July 19, 2022 1:54 AM

Thread

Instead of our classic view of a single point of execution within a program (i.e., a single PC where instructions are being fetched from and executed), a multi-threaded program has more than one point of execution (i.e., multiple PCs, each of which is being fetched and executed from). Perhaps another way to think of this is that each thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data.

The context switch between threads is quite similar to the context switch between processes, as the register state of T1 must be saved and the register state of T2 restored before running T2. With processes, we saved state to a process control block (PCB); now, we'll need one or more thread control blocks (TCBs) to store the state of each thread of a process. But address remain the same.

Threads share Process ID, Code(instructions), Most data(heap), Open File descriptors, Current Working Directory(environment variables), User and Group ID(permission, limits)

Each thread has its own Thread ID, set of registers including Program Counter and Stack pointer, Stack for local variables and return address(in same address space)

Many user-level threads map to one kernel-level thread. Use libraries. OS is not aware of user-level threads. For user-level threads, you can have your own schedule. Lower overhead thread operations since no system call. However, cannot use multiprocessors. Once one user-level thread is blocked, the whole user process is also blocked.

Kernel-level thread, one-to-one mapping. Each kernel thread scheduled by OS. Thread operations performed by OS. Each kernel-level thread run in parallel on a multiprocessor. When one thread is blocked, others keep going. Higher overhead of thread operations. OS must scale well with increasing numbers of threads.

Race Condition

A race condition occurs when two or more threads can access shared data and try to change it at the same time. A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread. An indeterminate program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not deterministic, something we usually expect from computer systems. To avoid these problems, threads should use some kind of mutual exclusion primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

Atomicity

when the instruction is executed atomically, it would perform the update as desired. Atomically, in this context, means “as a unit”, which sometimes we take as “all or none.”

Lock

- Provide mutual exclusion between threads. To evaluate whether a lock works, checks does the lock work, preventing multiple threads from entering a critical section(Mutual Exclusion). Does each thread contending for the lock get a fair shot at acquiring it once it is free(Fairness) ? And the time overheads added by using the lock(Performance).
- Earliest approach
- disable interrupts for critical sections, only works on uniprocessor
- this approach requires us to allow any calling thread to perform a privileged operation (turning interrupts on and off), and thus trust that this facility is not abused. A greedy program could call lock() at the beginning of its execution and thus monopolize the processor; worse, an errant or malicious program could call lock() and go into an endless loop. Does not work on multiprocessors, only disable one processor. Turning off interrupts for extended periods of time can lead to interrupts becoming lost, which can lead to serious systems problems. inefficient, code that masks or unmasks interrupts tends to be executed slowly by modern CPUs.

- Using Load and Store
- use a simple variable (flag) to indicate whether some thread has possession of a lock. After a thread acquire the lock, other threads will simply spin-wait in the while loop for that thread to call unlock() and clear the flag
- Both threads can set flag to 1, lock is incorrect

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Spin-waiting wastes time waiting for another thread to release a lock

- Solution

PETERSON'S ALGORITHM: INTUITION

Mutual exclusion: Enter critical section if and only if
- Other thread does not want to enter OR
- Other thread wants to enter, but your turn (only 1 turn)

Progress: Both threads cannot wait forever at while() loop
- Completes if other process does not want to enter
- Other process (matching turn) will eventually finish

Bounded waiting (not shown in examples)
- Each process waits at most one critical section (because turn given to other)

Problem: doesn't work on modern hardware
(hw doesn't provide sequential consistency due to caching)

- However, load and store is not atomic instruction
- Test and Set
-

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

this sequence of operations is performed atomically

-

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

- To work correctly on a single processor, it requires a preemptive scheduler. Without preemption, spin locks don't make much sense on a single CPU, as a thread spinning on a CPU will never relinquish it.

- Spin lock provides mutual exclusion, does not provide any fairness guarantees, performance overheads can be quite painful because of spin.
- Compare and Swap
-
```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```
-
```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```
- if just build a simple spin lock with CompareAndSwap, its behavior is identical to the spin lock with TestAndSet. But it is powerful as lock-free synchronization.
- Fetch and Add
-
```
1  int FetchAndAdd(int *ptr) {
2      int old = *ptr;
3      *ptr = old + 1;
4      return old;
5  }
```
-
```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn   = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15  }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }
```
- Ticket lock ensures progress for all threads
- Too much spinning

- Too much spinning wastes CPU. The problem gets worse with N threads contending for a lock; N − 1 time slices may be wasted in a similar manner, simply spinning and waiting for a single thread to release the lock.
- when you are going to spin, instead give up the CPU to another thread. As Al Davis might say, “just yield, baby!”
-

```
void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}
```

Condition Variable

- A condition variable is an explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition); some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition).
-

```
1  int done  = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

- Always use loop when checking condition
- Without state variable **done**, the child runs immediately and calls thr_exit() immediately; in this

case, the child will signal, but there is no thread asleep on the condition. When the parent runs, it will simply call wait and be stuck; no thread will ever wake it.

- Without mutex lock, if the parent calls thr_join() and then checks the value of done, it will see that it is 0 and thus try to go to sleep. But just before it calls wait to go to sleep, the parent is interrupted, and the child runs. The child changes the state variable done to 1 and signals, but no thread is waiting and thus no thread is woken. When the parent runs again, it sleeps forever, which is sad.
- Producer/Consumer
- Imagine one or more producer threads and one or more consumer threads. Producers generate data items and place them in a buffer; consumers grab said items from the buffer and consume them in some way.
- only put data into the buffer when count is zero (i.e., when the buffer is empty), and only get data from the buffer when count is one (i.e., when the buffer is full).
-

```
1  int buffer[MAX];
2  int fill_ptr = 0;
3  int use_ptr  = 0;
4  int count    = 0;
5
6  void put(int value) {
7      buffer[fill_ptr] = value;
8      fill_ptr = (fill_ptr + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use_ptr];
14     use_ptr = (use_ptr + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

```
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);
8          while (count == MAX)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

- If just one condition, every thread might go to sleep. Two consumers run first and both wait. Producer fills the buffer, signal consumer 1 and goes to sleep. Consumer 1 wakes up, empty the buffer, signal consumer 2 and goes to sleep. Consumer 2 find that buffer is empty, goes to sleep.
- If use if statement to check count, consumer might get no data. Consumer 1 run first and wait. Producer fills the buffer, signal consumer 1 and goes to sleep. Then consumer 2 sneaks in, take the data and signal producer. Then consumer 1 runs, however, there is no data to read. Needs to check condition again after woke up

- Semaphores
- A semaphore is an object with an integer value that we can manipulate with two routines; in the POSIX standard, these routines are sem_wait() and sem_post().Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value.
-

```
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  }
```

- Binary semaphore, behaves like lock, provides mutual exclusion. Semaphore is initialized with value 1. sem_post() does not wait for some particular condition to hold like sem_wait() does. Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up. The value of the semaphore, when negative, is equal to the number of waiting threads.
-

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

- For ordering, behaves like condition variable. Semaphore is initialized with value 0. If parent runs first, sem_wait() will cause parent goes to sleep.
-

```
1  void *producer(void *arg) {
2      int i;
3      for (i = 0; i < loops; i++) {
4          sem_wait(&empty);        // Line P1
5          sem_wait(&mutex);        // Line P1.5 (MUTEX HERE)
6          put(i);                  // Line P2
7          sem_post(&mutex);        // Line P2.5 (AND HERE)
8          sem_post(&full);         // Line P3
9      }
10 }
11
12 void *consumer(void *arg) {
13     int i;
14     for (i = 0; i < loops; i++) {
15         sem_wait(&full);         // Line C1
16         sem_wait(&mutex);        // Line C1.5 (MUTEX HERE)
17         int tmp = get();         // Line C2
18         sem_post(&mutex);        // Line C2.5 (AND HERE)
19         sem_post(&empty);        // Line C3
20         printf("%d\n", tmp);
21     }
22 }
```

- Producer/Consumer. Empty is initialized with number of buffer, full is initialized with value 0. The reason to add mutual exclusion is when there are multiple producers, there can be a race condition, multiple producers write to the buffer at the same time. Besides, wait mutex after wait empty/full to avoid deadlock. If wait mutex before, one consumer can acquire the mutex, since the buffer is empty now, it will go to sleep and wait but does not release the mutex. Producer run now and try to acquire mutex, deadlock occurs.
-

```
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 }
```

Figure 31.17: **Implementing Zemaphores With Locks And CVs**

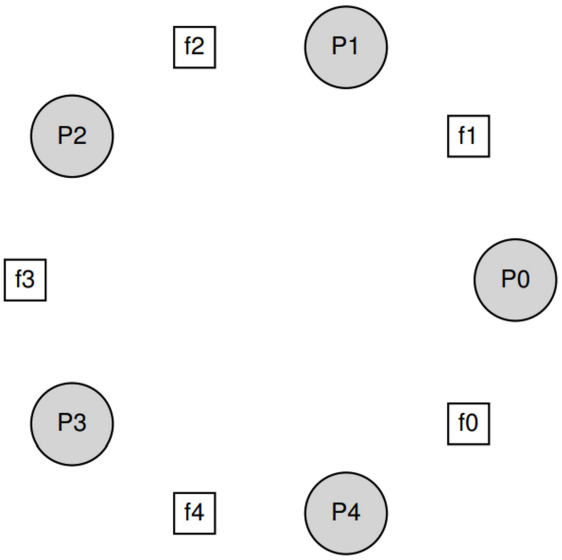
Read-Writer Lock

- Commonly used in database.
- when the first reader acquires the lock; in that case, the reader also acquires the write lock by calling sem_wait() on the writelock semaphore, and then releasing the lock by calling sem_post(). Thus, once a reader has acquired a read lock, more readers will be allowed to acquire the read lock too; however, any thread that wishes to acquire the write lock will have to wait until all readers are finished; the last one to exit the critical section calls sem_post() on “writelock” and thus enables a waiting writer to acquire the lock. May starve the writer.
-

```
1  typedef struct _rwlock_t {
2    sem_t lock;      // binary semaphore (basic lock)
3    sem_t writelock; // allow ONE writer/MANY readers
4    int   readers;   // #readers 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) // first reader gets writelock
17     sem_wait(&rw->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) // last reader lets it go
25     sem_post(&rw->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 }
```

Dinning Philosophers

- there are five “philosophers” sitting around a table. Between each pair of philosophers is a single fork (and thus, 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, and the synchronization problems that ensue, are what makes this a problem we study in concurrent programming.
-



- Basic loop

```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

- Helper function

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

- Wrong solution

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

- 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. Deadlock occurs.

- Breaking dependency

```
1  void get_forks(int p) {
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  }
```

- philosopher 4 (the highest numbered one) gets the forks in a different order than the others
- Better Solution
-

```
sem_t mayEat[5];
sem_t mutex;    // Initialized to 1
int state[5] = {THINKING};
take_chopsticks(int i) {
   wait(&mutex); // enter critical section
   state[i] = HUNGRY;
   testSafetyAndLiveness(i); // check if i can run
   signal(&mutex); // exit critical section
   wait(&mayEat[i]);
}
put_chopsticks(int i) {
   wait(&mutex); // enter critical section
   state[i] = THINKING;
   testSafetyAndLiveness((i+1) % 5); // check if neighbor can run now
   testSafetyAndLiveness((i+4) % 5);
   signal(&mutex); // exit critical section
}
testSafetyAndLiveness(int i) {
   if(state[i]==HUNGRY && state[(i+4) % 5] != EATING && state[(i+1) % 5] != EATING) {
      state[i] = EATING;
      signal(&mayEat[i]);
   }
}
```

Atomicity-Violation Bugs

- The desired serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution).

```
1  Thread 1::
2  if (thd->proc_info) {
3    fputs(thd->proc_info, ...);
4  }
5
6  Thread 2::
7  thd->proc_info = NULL;
```

- if the first thread performs the check but then is interrupted before the call to fputs, the second thread could run in-between, thus setting the pointer to NULL; when the first thread resumes, it will crash, as a NULL pointer will be dereferenced by fputs. Use mutual exclusion to fix it

Order-Violation Bugs

- The desired order between two (groups of) memory accesses is flipped, A should always be executed before B, but the order is not enforced during execution)

```
1  Thread 1::
2  void init() {
3     mThread = PR_CreateThread(mMain, ...);
4  }
5
6  Thread 2::
7  void mMain(...) {
8      mState = mThread->State;
9  }
```

- if Thread 2 runs immediately once created, the value of mThread will not be set when it is accessed within mMain() in Thread 2, and will likely crash with a NULL-pointer dereference. Note that we assume the value of mThread is initially NULL; if not, even stranger things could happen as arbitrary memory locations are accessed through the dereference in Thread 2.

Deadlock

```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

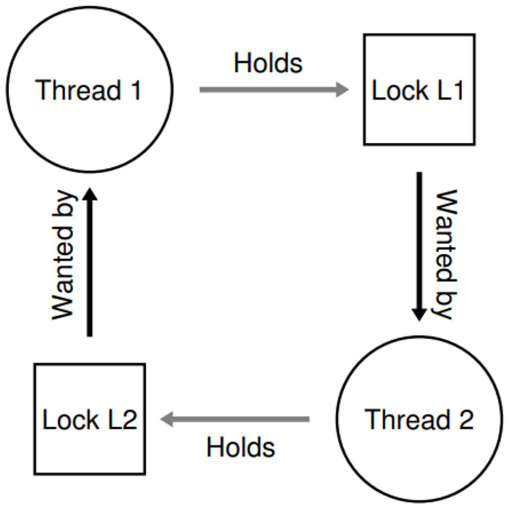


Figure 32.7: **The Deadlock Dependency Graph**

- in large code bases, complex dependencies arise between components. Deadlock may occur. Another reason is due to the nature of encapsulation.

- ```
  Vector v1, v2;
  v1.AddAll(v2);
  ```

- Internally, because the method needs to be multi-thread safe, locks for both the vector being added to (v1) and the parameter (v2) need to be acquired. The routine acquires said locks in some arbitrary order (say v1 then v2) in order to add the contents of v2 to v1. If some other thread calls v2.AddAll(v1) at nearly the same time, we have the potential for deadlock, all in a way that is quite hidden from the calling application.
- Conditions for Deadlock
  - Mutual exclusion: Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
  - Hold-and-wait: Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
  - No preemption: Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
  - Circular wait: There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.
  - If any of these four conditions are not met, deadlock cannot occur.
  - To prevent mutual exclusion, use lock free algorithm, try to replace lock with atomic primitive like CompareAndSwap. However, livelock is possible.
  - To prevent hold-and-wait, try acquire all locks at once, atomically. Must know ahead of time which locks will be needed, must be conservative(acquire all possible locks), reduces concurrency which is bad.
  ```

```
pthread_mutex_lock(prevention);   // begin acquisition
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

- To prevent no preemption, use try lock. If thread cannot get what it wants, release what it holds. Livelock is possible, threads changing states but not making any progress. Possible solution for livelock is adding random backoff, add a random delay before looping back and trying the entire thing over again, thus decreasing the odds of repeated interference among competing threads.

```
top:
  pthread_mutex_lock(L1);
  if (pthread_mutex_trylock(L2) != 0) {
    pthread_mutex_unlock(L1);
    goto top;
  }
```

- To prevent circular wait: provide a total ordering on lock acquisition. For example, there are two locks in the system(L1 and L2), always acquire L1 before L2. Or partial ordering.

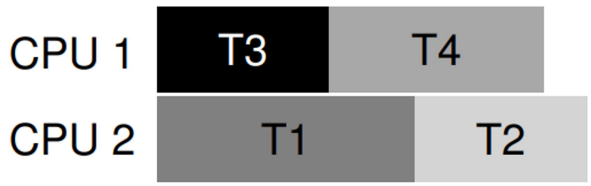
Lock Ordering in Xv6

Creating a file requires simultaneously holding:

- a lock on the directory,
- a lock on the new file's inode,
- a lock on a disk block buffer,
- idelock,
- ptable.lock

- Avoid deadlock via scheduling. As long as threads that acquire the same locks do not run at the same time. No deadlock could ever arise.

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no



However, the cost is performance. The total time to complete the jobs is lengthened considerably. Besides, the system must have full knowledge of the entire set of tasks that must be run and the locks that they need. Further, such approaches can limit concurrency.