- Locks and locking strategies

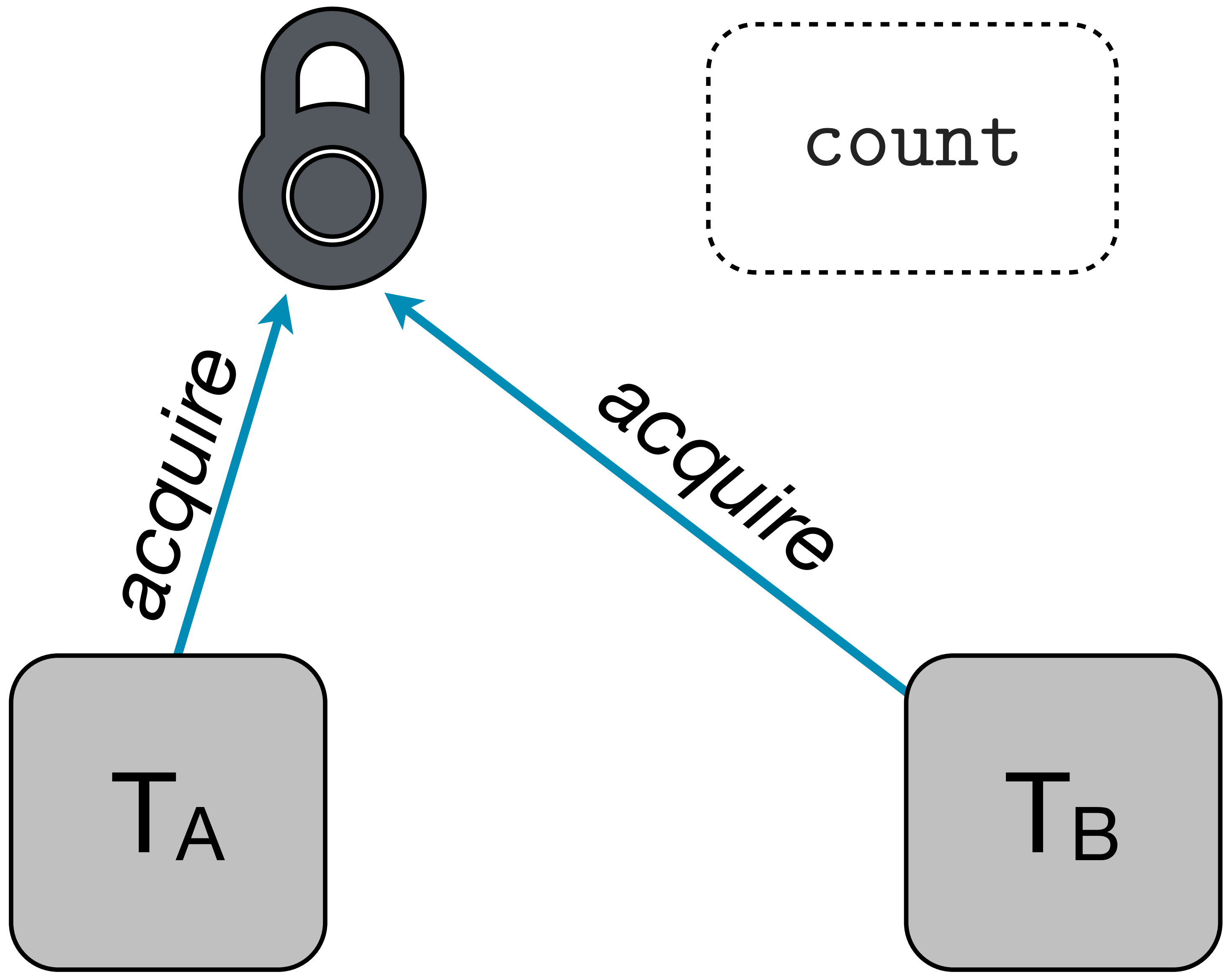
E.g., locking

Thread A

```
a1  count = count + 1
```

Thread B

```
b1  count = count + 1
```



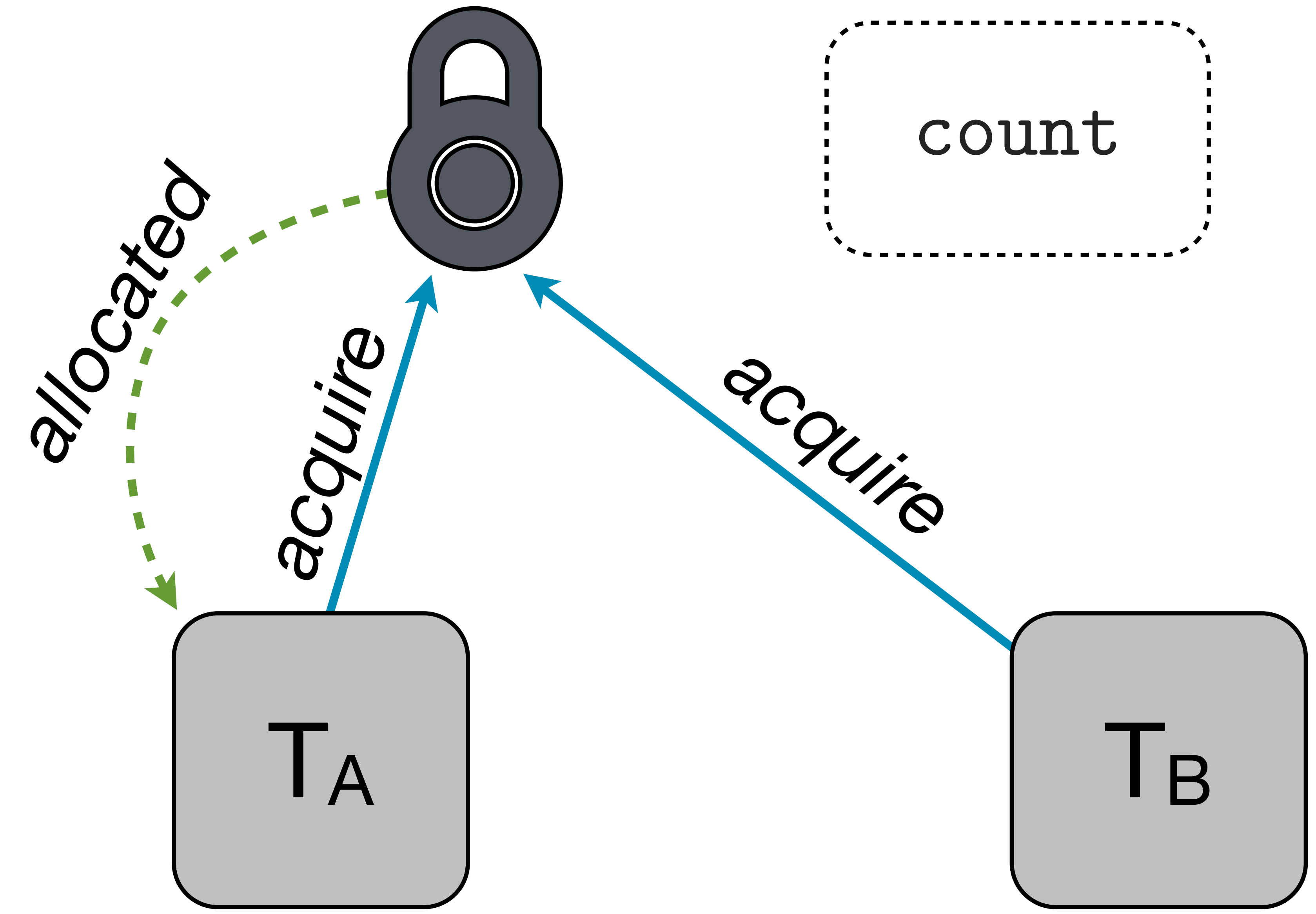
E.g., locking

Thread A

```
a1  count = count + 1
```

Thread B

```
b1  count = count + 1
```



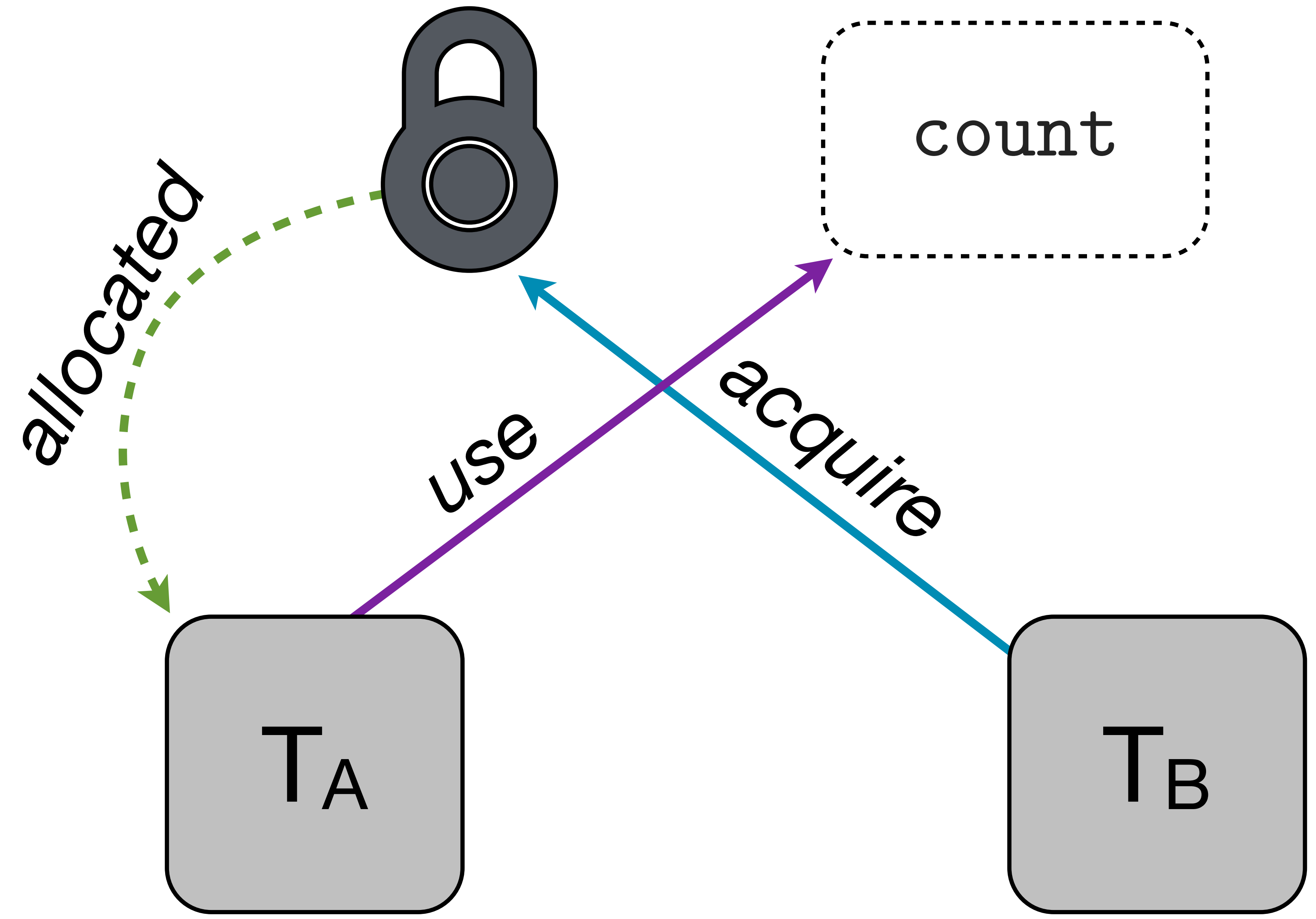
E.g., locking

Thread A

```
a1  count = count + 1
```

Thread B

```
b1  count = count + 1
```



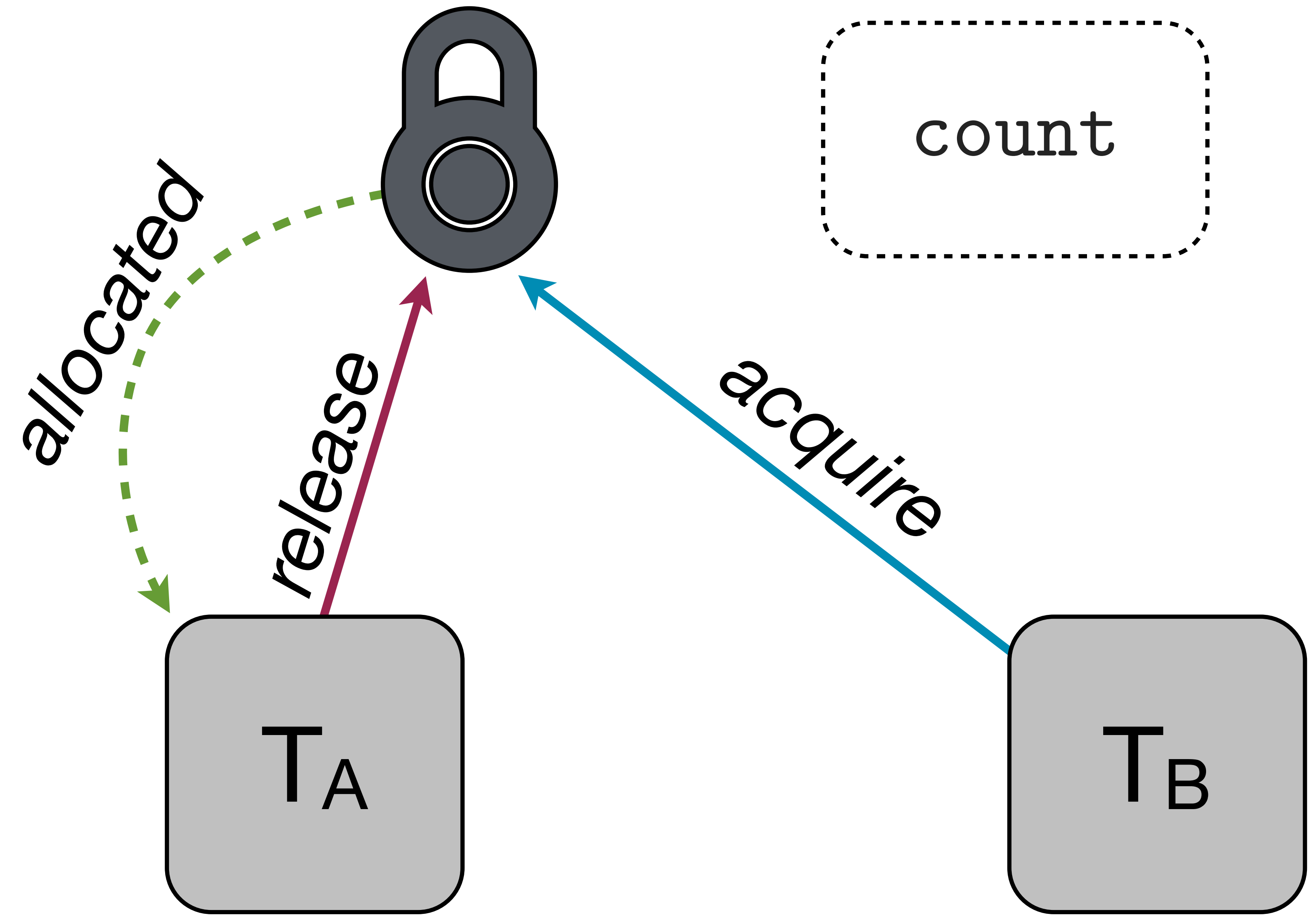
E.g., locking

Thread A

```
a1  count = count + 1
```

Thread B

```
b1  count = count + 1
```



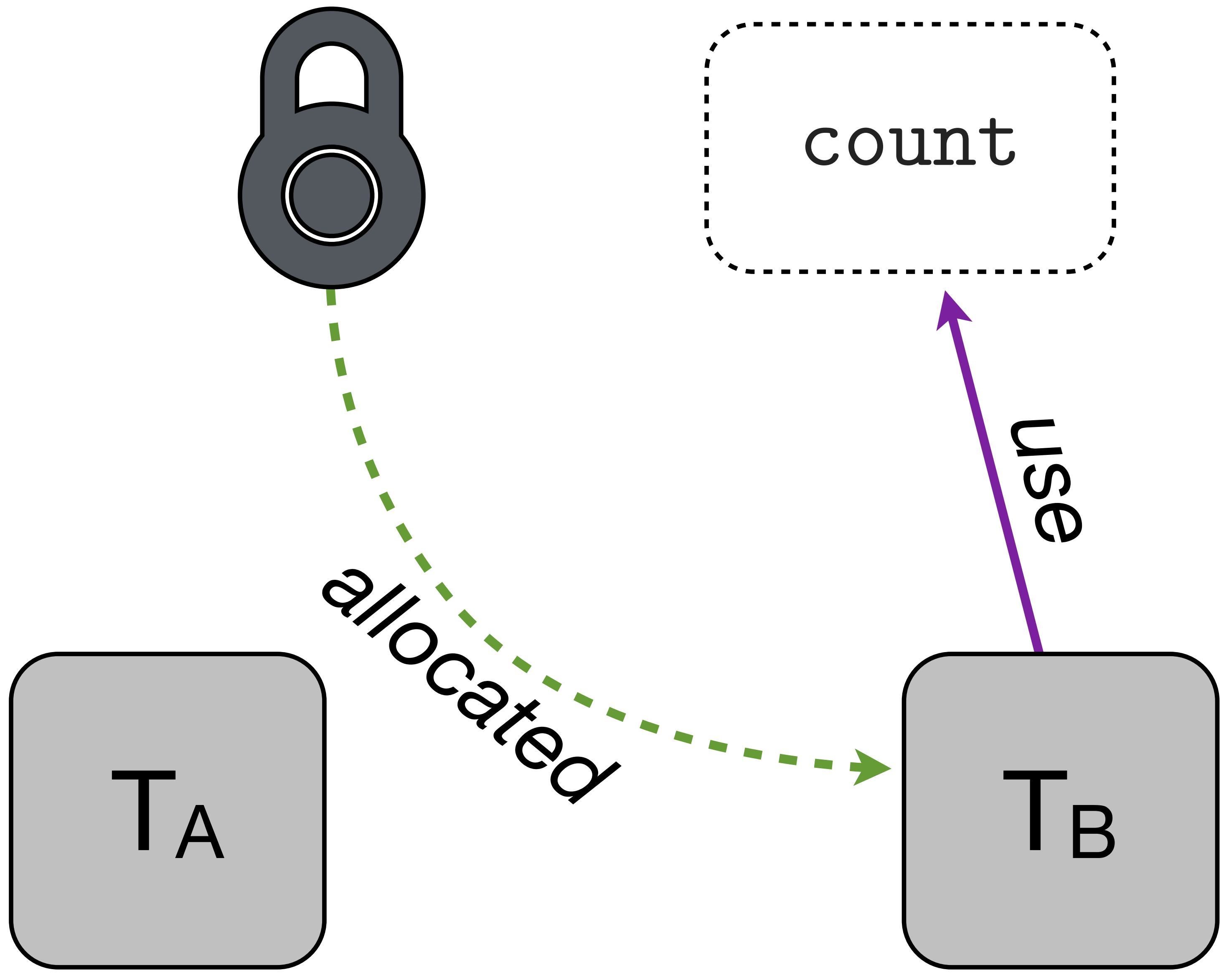
E.g., locking

Thread A

```
a1  count = count + 1
```

Thread B

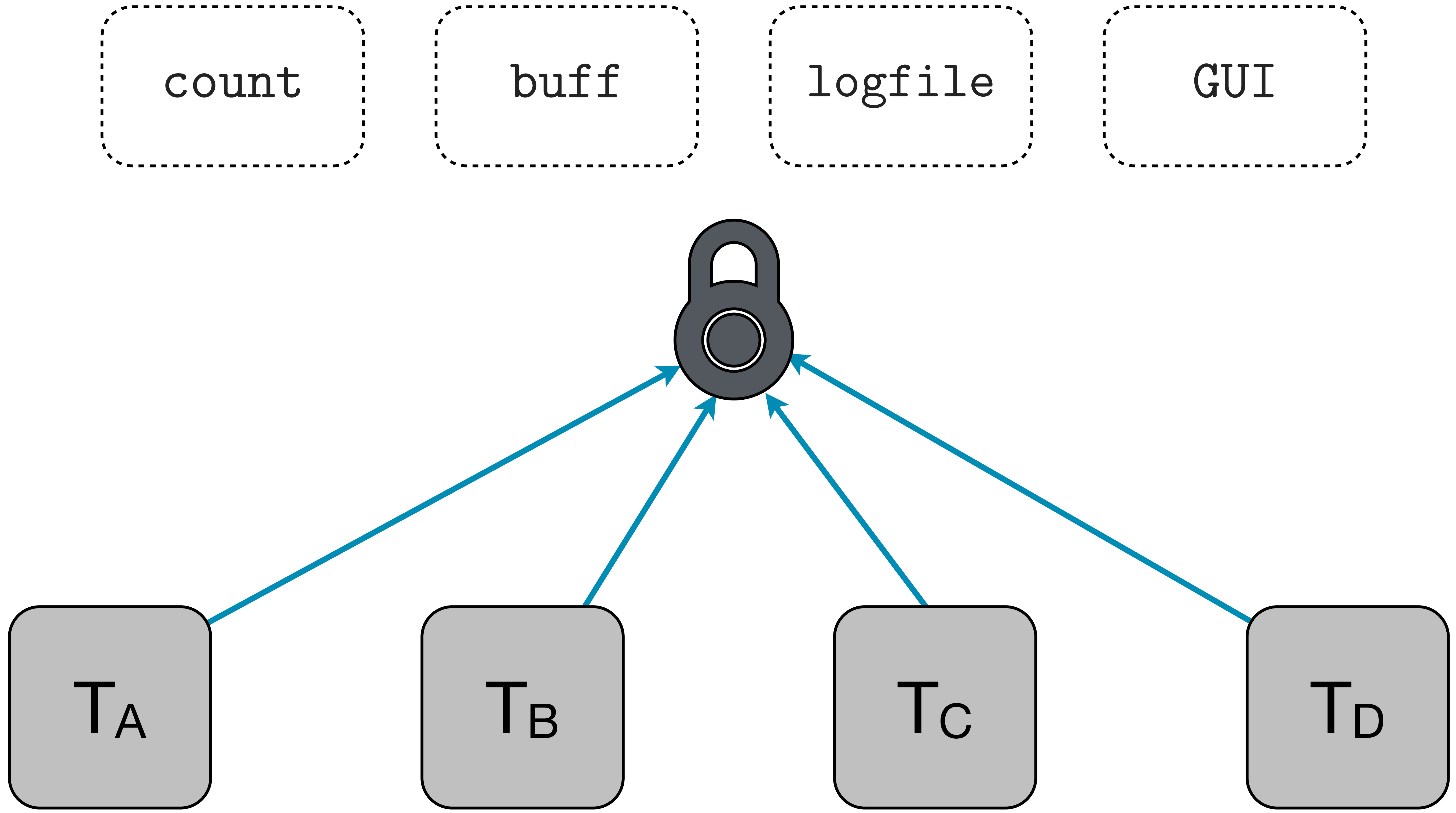
```
b1  count = count + 1
```



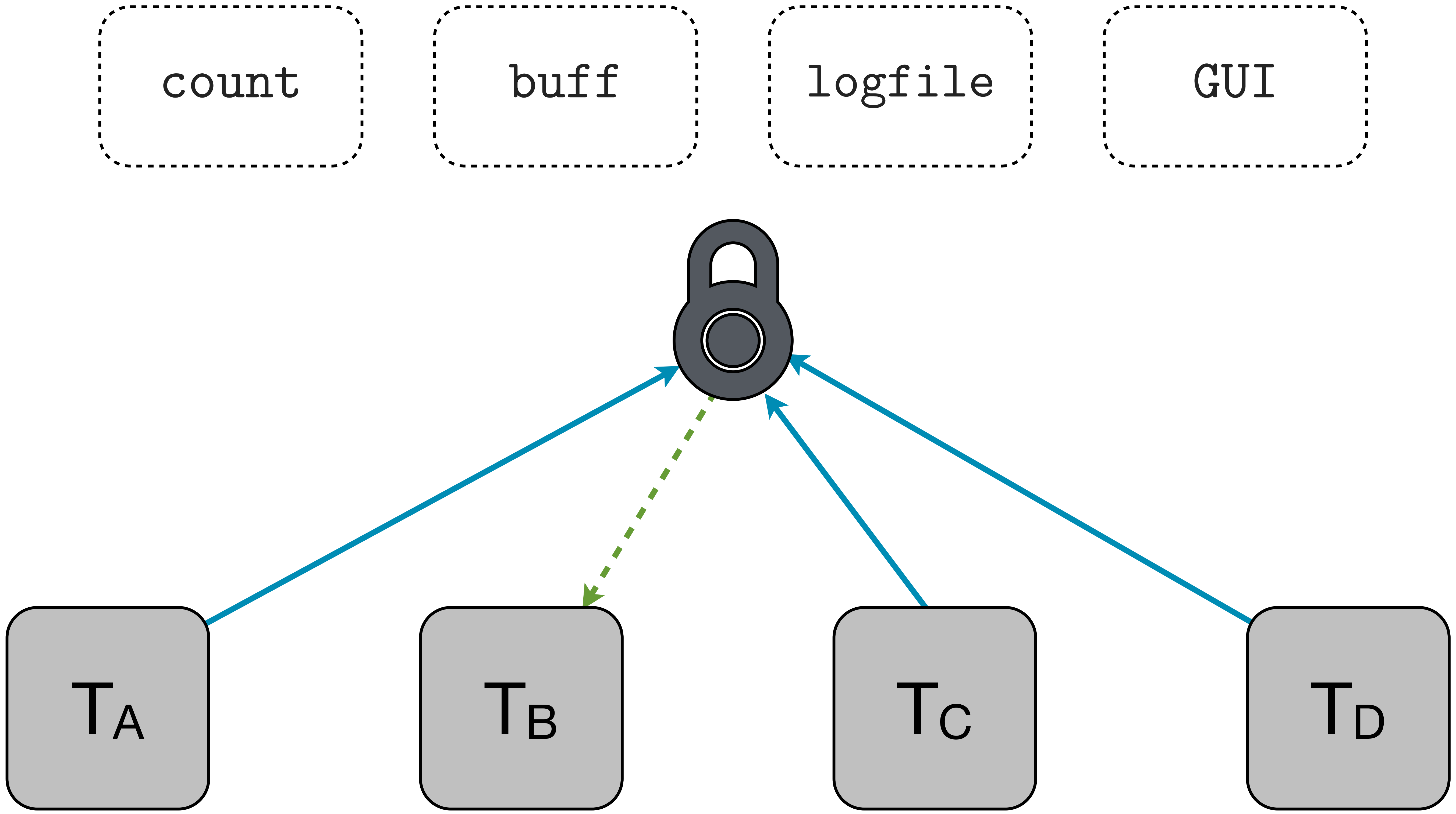
Locking strategies

- We may use a single lock to guard access to all shared resources
  - We call this a global or **coarse-grained** locking strategy
- Or we may assign locks to individual resources (or subsets of resources)
  - We call this a **fine-grained** locking strategy

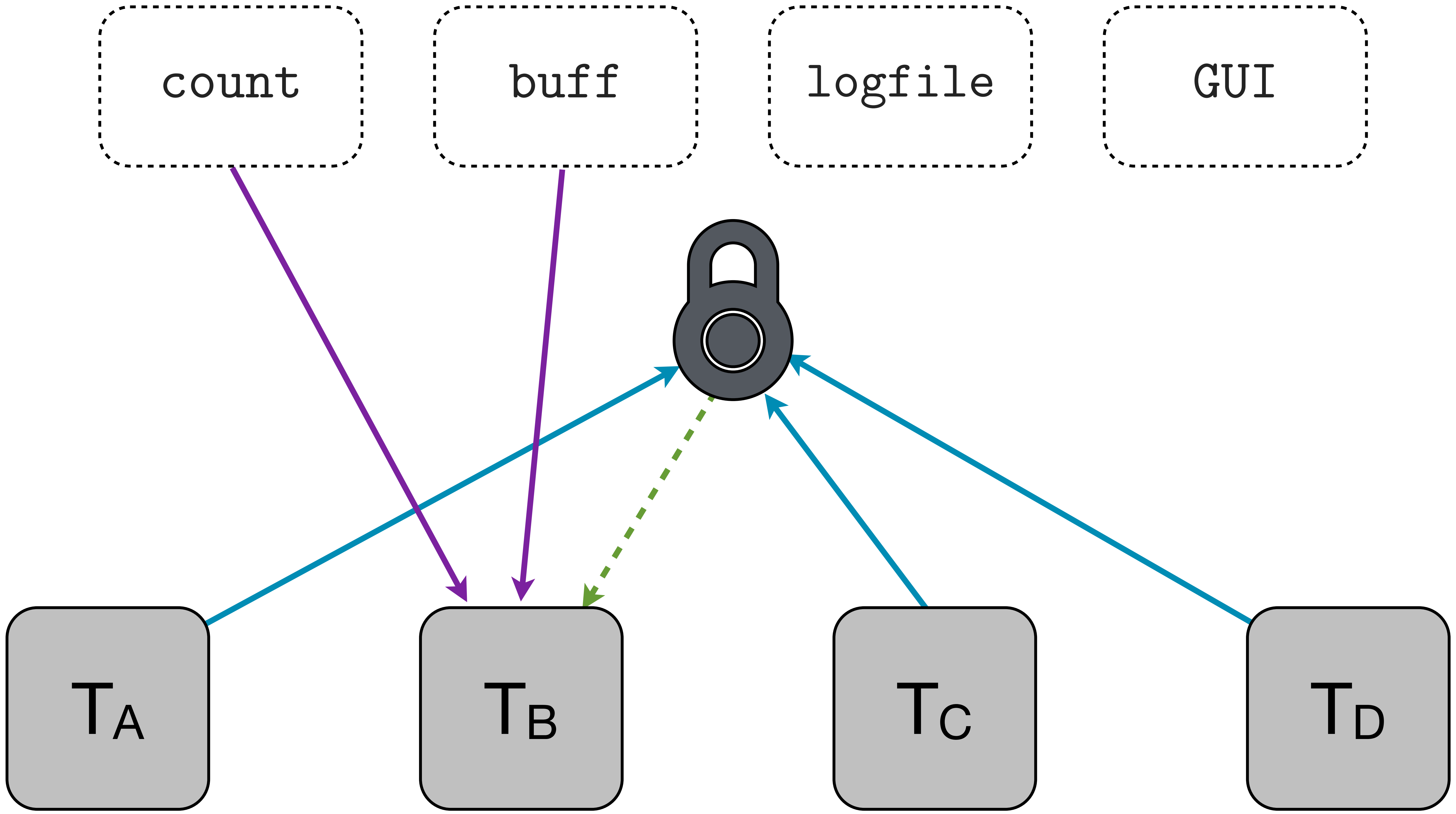
E.g., coarse-grained locking



E.g., coarse-grained locking



E.g., coarse-grained locking



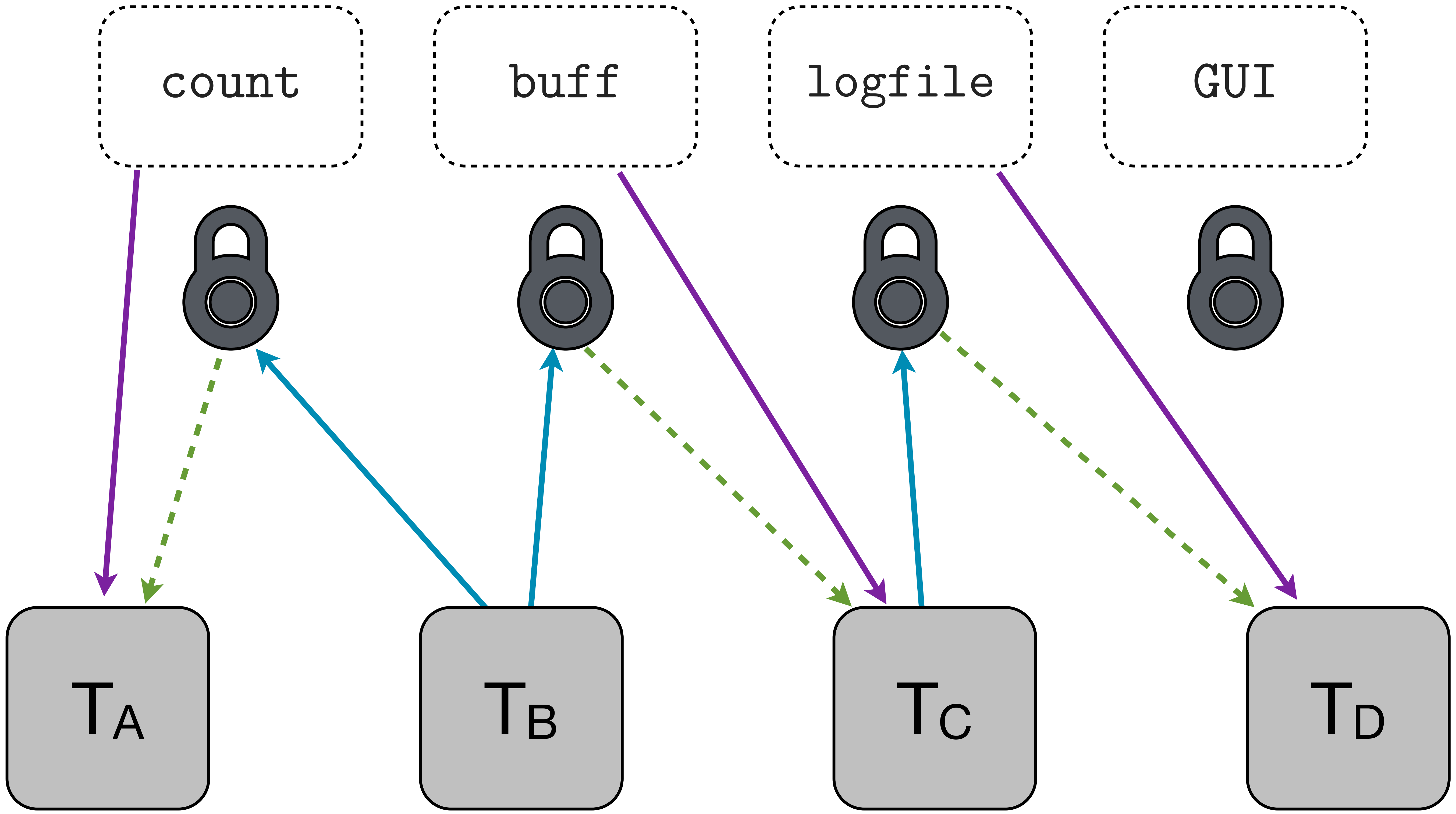
Coarse-grained locking …

... is (typically) easier to reason about

... but results in a lot of lock contention

... may result in poor resource utilization

E.g., fine-grained locking



Fine-grained locking ...

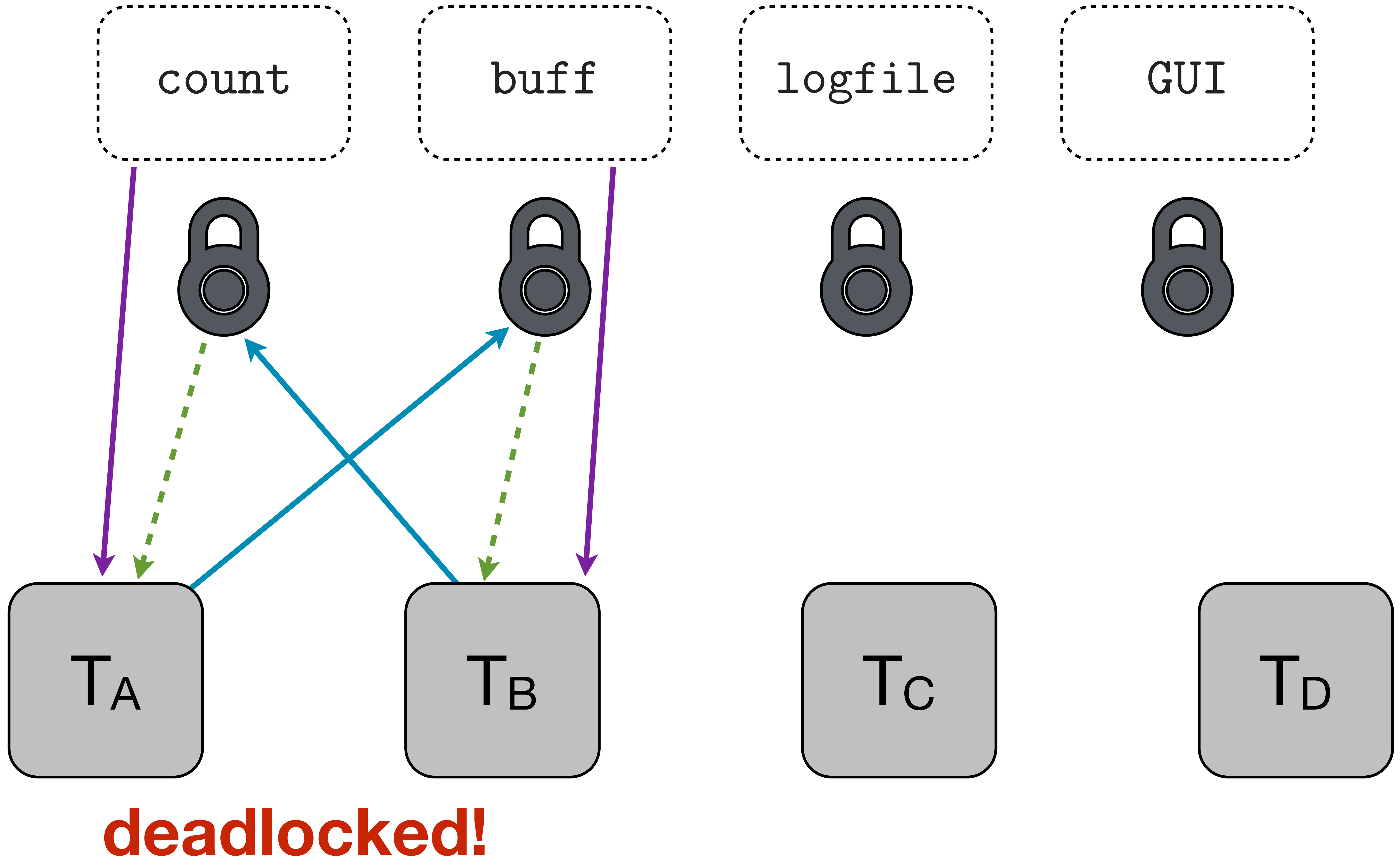
... may reduce (individual) lock contention

... may improve resource utilization

... can result in a lot of locking overhead

... but can be much harder to verify correctness!

E.g., fine-grained locking problem



E.g., lock API: pthreads “mutex”

```
// initialize mutex (can also use PTHREAD_MUTEX_INITIALIZER for defaults)
int pthread_mutex_init(pthread_mutex_t *mtx, pthread_mutexattr_t *attr);

// acquire lock on mutex (if mutex is already locked, block the calling thread)
int pthread_mutex_lock(pthread_mutex_t *mtx);

// release lock on mutex (a blocked thread may acquire it)
int pthread_mutex_unlock(pthread_mutex_t *mtx);

// destroy mutex (only safe on an unlocked mutex)
int pthread_mutex_destroy(pthread_mutex_t *mtx);
```

E.g., protecting counter increment

```c
int counter = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *inc(void *num) {
    for (int i=0; i<1000000; i++) {
        pthread_mutex_lock(&lock);
        counter += 1;
        pthread_mutex_unlock(&lock);
    }
    printf("Thread %ld counter = %d\n", pthread_self(), counter);
    pthread_exit(NULL);
}

int main() {
    pthread_t tid[5];
    for (int i=0; i<5; i++){
        pthread_create(&tid[i], NULL, inc, NULL);
        printf("Created thread %ld\n", tid[i]);
    }
    for (int i=0; i<5; i++) {
        pthread_join(tid[i], NULL); // wait for other threads
    }
    pthread_mutex_destroy(&lock);
    return 0;
}
```

```
Created thread 139882746513152
Created thread 139882738120448
Created thread 139882729727744
Created thread 139882721335040
Created thread 139882712942336
Thread 139882721335040 counter = 4782346
Thread 139882729727744 counter = 4904819
Thread 139882738120448 counter = 4976793
Thread 139882746513152 counter = 4986816
Thread 139882712942336 counter = 5000000
```

- Lots of lock contention!
- Note that counter values are still unpredictable until the end
    - Can we fix this?

E.g., protecting counter increment

```c
int counter = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *inc(void *num) {
    pthread_mutex_lock(&lock);
    for (int i=0; i<1000000; i++) {
       counter += 1;
    }
    printf("Thread %ld counter = %d\n", pthread_self(), counter);
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}

int main() {
    pthread_t tid[5];
    for (int i=0; i<5; i++){
        pthread_create(&tid[i], NULL, inc, NULL);
        printf("Created thread %ld\n", tid[i]);
    }
    for (int i=0; i<5; i++) {
        pthread_join(tid[i], NULL); // wait for other threads
    }
    pthread_mutex_destroy(&lock);
    return 0;
}
```

```
Created thread 140077130561280
Created thread 140077122168576
Created thread 140077113775872
Created thread 140077105383168
Created thread 140077096990464
Thread 140077122168576 counter = 1000000
Thread 140077113775872 counter = 2000000
Thread 140077105383168 counter = 3000000
Thread 140077130561280 counter = 4000000
Thread 140077096990464 counter = 5000000
```

- Less locking overhead
- Predictable counter outputs
- But virtually no concurrency

E.g., protecting counter increment

```c
int counter = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *inc(void *num) {
    pthread_mutex_lock(&lock);
    for (int i=0; i<1000000; i++) {
       counter += 1;
    }
    printf("Thread %ld counter = %d\n", pthread_self(), counter);
    pthread_exit(NULL);
    pthread_mutex_unlock(&lock);
}

int main() {
    pthread_t tid[5];
    for (int i=0; i<5; i++){
        pthread_create(&tid[i], NULL, inc, NULL);
        printf("Created thread %ld\n", tid[i]);
    }
    for (int i=0; i<5; i++) {
        pthread_join(tid[i], NULL); // wait for other threads
    }
    pthread_mutex_destroy(&lock);
    return 0;
}
```

```
Created thread 139755903194880
Created thread 139755894802176
Created thread 139755886409472
Created thread 139755878016768
Created thread 139755869624064
Thread 139755903194880 counter = 1000000
                                        (hangs)
```

- Mutex isn't released before thread termination — remaining threads are blocked forever
- Must pay careful attention to lock usage!

Lock implementation

- Basic idea: need an “acquire” function that lets only one caller through while others block

```
typedef struct { int locked; } lock_t;

void acquire(lock_t *l) {
    while (1) {
        if (!l->locked) {
            l->locked = 1;
            break;
        }
    }
}
```

```
void release(lock_t *l) {
    l->locked = 0;
}
```

Lock implementation

```
void acquire(lock_t *l) {
    while (1) {
        if (!l->locked) {
            l->locked = 1;
            break;
        }
    }
}
```

problem: calling thread may be preempted between testing the value of the thread and setting its value

- Race condition may allow multiple threads to acquire the lock!
- Cannot easily fix this problem in software — rely on hardware support

“Test-and-Set” operation

- Many architectures support an atomic test-and-set operation
  - E.g., on x86 we have the “atomic exchange” instruction: `xchg`
    - Can use it to implement acquire:

```
# note: pseudo-assembly!
acquire:
    movl  $1, %eax           # set up "new" value in reg
    xchgl l->locked, %eax    # swap values in reg & lock
    testl %eax, %eax
    jne   acquire            # spin if old value ≠ 0
```

Spin lock

- This implementation ensures mutex, but is very expensive
  - Blocked threads are burning CPU time to repeatedly check the lock status
- “Starvation” issue: no guarantee if/when a thread stuck looping will acquire the lock!

```
acquire:
    movl  $1, %eax
    xchgl l->locked, %eax
    testl %eax, %eax
    jne   acquire
```

Ticket lock

- Clever starvation-free alternative to test-and-set based spinlock

```
typedef struct {
    int ticket;
    int turn;
} lock_t;

lock_t lock = { 0, 0 };
```

```
void acquire(lock *lock) {
    int tkt = lock->ticket++; // need atomic ++
    while (tkt != lock->turn)
        ; // spin
}

void release(lock *lock) {
    lock->turn = lock->turn + 1;
}
```

- Once a thread gets a “ticket”, it will eventually acquire the lock
- Requires an atomic increment instruction; e.g., `xadd` on x86

Eliminating “spin”

- Would like to minimize CPU usage of tasks blocking on a lock
  - Ideally: try to check/acquire lock again only when there’s good reason (e.g., it’s been released by another thread)
- Typically rely on OS support for distinct scheduler state and explicit unblocking mechanism
  - e.g., in xv6, processes may be “SLEEPING”, and sleep/wakeup functions allow processes to block on and wait for notifications on specific “channels”

E.g., xv6 sleep/wakeup

```c
// Put calling process to sleep on chan
void sleep(void *chan)
{
  proc->chan = chan;
  proc->state = SLEEPING;
  sched(); // context switch away from proc
  proc->chan = 0;
}
```

```c
// Wake up all processes sleeping on chan
void wakeup1(void *chan)
{
  struct proc *p;
  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}
```

- What happens if sleep and wakeup are called concurrently?
  - Race condition! Process calling `sleep` may either be continue to run or be put to sleep — latter scenario is termed a “lost wakeup”
  - Fix this with mutex around critical sections

E.g., xv6 sleep/wakeup

```
void sleep(void *chan, struct spinlock *lk)
{
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }

  proc->chan = chan;
  proc->state = SLEEPING;
  sched(); // note: scheduler releases lock
  proc->chan = 0;

  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
}
```

```
void wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}


void wakeup1(void *chan)
{
  struct proc *p;
  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}
```

- Note that acquire/release still make use of spinlocks
  - But they are held only for a fairly short period of time

E.g., sleep/wakeup in wait/exit

```c
// Wait for a child process to exit
int wait(void)
{
  struct proc *p;
  int havekids, pid;

  // this lock ensures we will not miss the wakeup
  acquire(&ptable.lock);
  for(;;){
    for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
      if(p->parent != proc)
        continue;
     if(p->state == ZOMBIE){
        pid = p->pid;
        release(&ptable.lock);
        return pid;
      }
    }

    // sleep on channel identified by parent proc
    sleep(proc, &ptable.lock);
  }
}
```

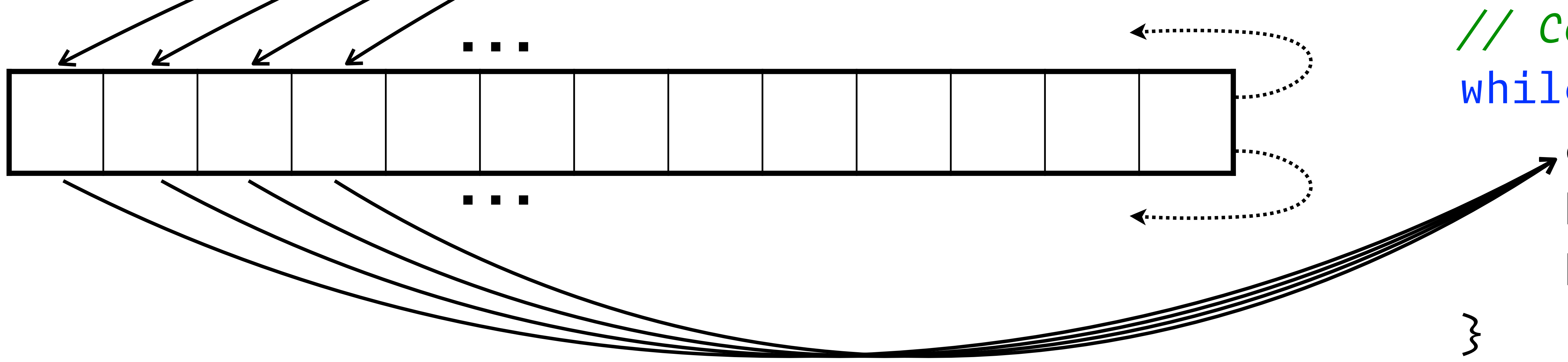
```c
// Exit the current process.
void exit(void)
{
  struct proc *p;
  acquire(&ptable.lock);

  // wake up parent process to reap this one
  wakeup1(proc->parent);

  // init adopts & reaps orphaned children
  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
    if(p->parent == proc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

Producer/Consumer problem

- One of many classical — i.e., paradigmatic — concurrent problems
- Setup: concurrent producer & consumer threads sharing a finite buffer

```c
typedef struct {
    int queue[BSIZE];
    int n_items;
    int head;
    int tail;
} buffer_t;
```

```c
// Producer (may be more than 1)
while (1) {
    buf->queue[buf->tail] = produce();
    buf->tail = (buf->tail + 1) % BSIZE;
    buf->n_items++;
}
```



```c
// Consumer (may be more than 1)
while (1) {
    consume(buf->queue[buf->head]);
    buf->head = (buf->head + 1) % BSIZE;
    buf->n_items--;
}
```

Producer/Consumer problem

```
// Producer
while (1) {
    buf->queue[buf->tail] = produce();
    buf->tail = (buf->tail + 1) % BSIZE;
    buf->n_items++;
}
```

```
// Consumer
while (1) {
    consume(buf->queue[buf->head]);
    buf->head = (buf->head + 1) % BSIZE;
    buf->n_items--;
}
```

- Must guard access to all shared data with a mutex
- But access to shared buffer must also be carefully *synchronized*
    - I.e., consumer may only consume from non-empty buffer, and producer may only produce into buffer with open slots

Producer/Consumer problem

```
// Producer
while (1) {
    while (buf->n_items == BSIZE)
        ; // spin barrier

    item = produce();
    pthread_mutex_lock(&lock);
    buf->queue[buf->tail] = item;
    buf->tail = (buf->tail + 1) % BSIZE;
    buf->n_items++;
    pthread_mutex_unlock(&lock);
}
```

```
// Consumer
while (1) {
    while (buf->n_items == 0)
        ; // spin barrier

    pthread_mutex_lock(&lock);
    item = buf->queue[buf->head];
    buf->head = (buf->head + 1) % BSIZE;
    buf->n_items--;
    pthread_mutex_unlock(&lock);
    consume(item);
}
```

- More subtle race condition: when consumer updates `n_items`, multiple producers may fall through spin barrier (and vice versa)
- Must check condition in mutex, but unlock to allow other thread to run

Producer/Consumer problem

```
// Producer
while (1) {
    pthread_mutex_lock(&lock);
    while (buf->n_items == BSIZE) {
        pthread_mutex_unlock(&lock);
        // hope consumer decrements n_items
        pthread_mutex_lock(&lock);
    }
    pthread_mutex_unlock(&lock);

    item = produce();

    pthread_mutex_lock(&lock);
    buf->queue[buf->tail] = item;
    buf->tail = (buf->tail + 1) % BSIZE;
    buf->n_items++;
    pthread_mutex_unlock(&lock);
}
```

- Ridiculous!
- Prefer a way to block producer until consumer makes space available
    - Similar to sleep/wakeup mechanism in kernel

Condition variable

- Gives us mechanism for:
  - Representing a condition used for thread synchronization
  - Where a thread might **wait** (block) until the condition changes
  - Where a thread might **signal** other blocked threads to wake up and re-check the condition

E.g., pthreads “cond”

```
// initialize condition variable (or use PTHREAD_COND_INITIALIZER for defaults)
int pthread_cond_init(pthread_cond_t *cv, pthread_condattr_t *attr);

// block on cv and release mtx (which must be held by calling thread)
// mtx is automatically re-acquired before returning
int pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mtx);

// unblock one thread that is blocked on cv
int pthread_cond_signal(pthread_cond_t *cv);

// unblock all threads that are blocked on cv
int pthread_cond_broadcast(pthread_cond_t *cv);
```

Producer/Consumer problem

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t has_space = PTHREAD_COND_INITIALIZER,
               has_items = PTHREAD_COND_INITIALIZER;

// Producer
while (1) {
    pthread_mutex_lock(&lock);
    while (buf->n_items == BSIZE)
        pthread_cond_wait(&has_space, &lock);
    pthread_mutex_unlock(&lock);

    item = produce();

    pthread_mutex_lock(&lock);
    buf->queue[buf->tail] = item;
    buf->tail = (buf->tail + 1) % BSIZE;
    buf->n_items++;
    pthread_mutex_unlock(&lock);

    pthread_cond_signal(&has_items);
}
```

```c
// Consumer
while (1) {
    pthread_mutex_lock(&lock);
    while (buf->n_items == 0)
        pthread_cond_wait(&has_items, &lock);

    item = buf->queue[buf->head];
    buf->head = (buf->head + 1) % BSIZE;
    buf->n_items--;
    pthread_mutex_unlock(&lock);

    pthread_cond_signal(&has_space);

    consume(item);
}
```

released while blocking