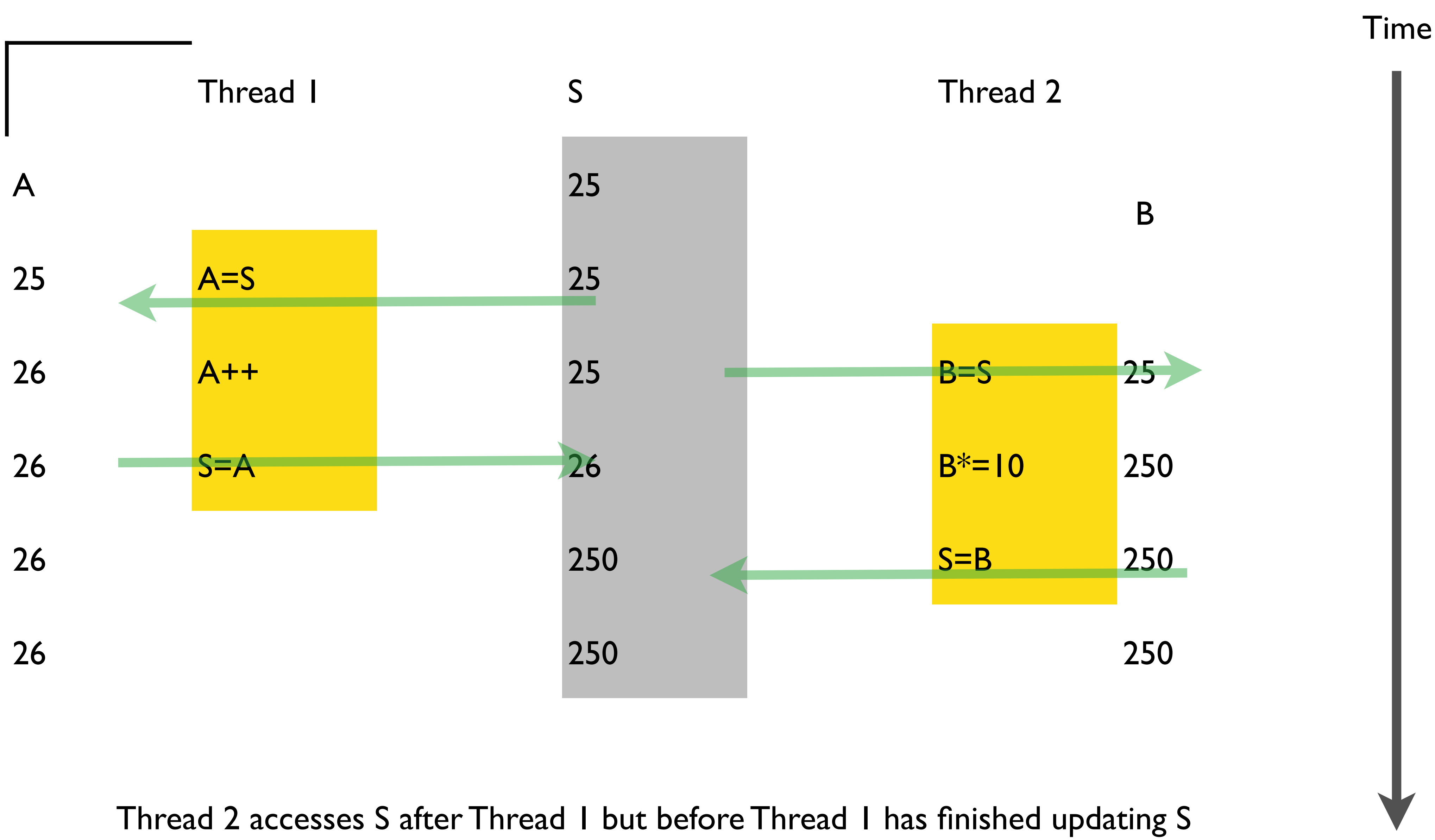
Synchronisation

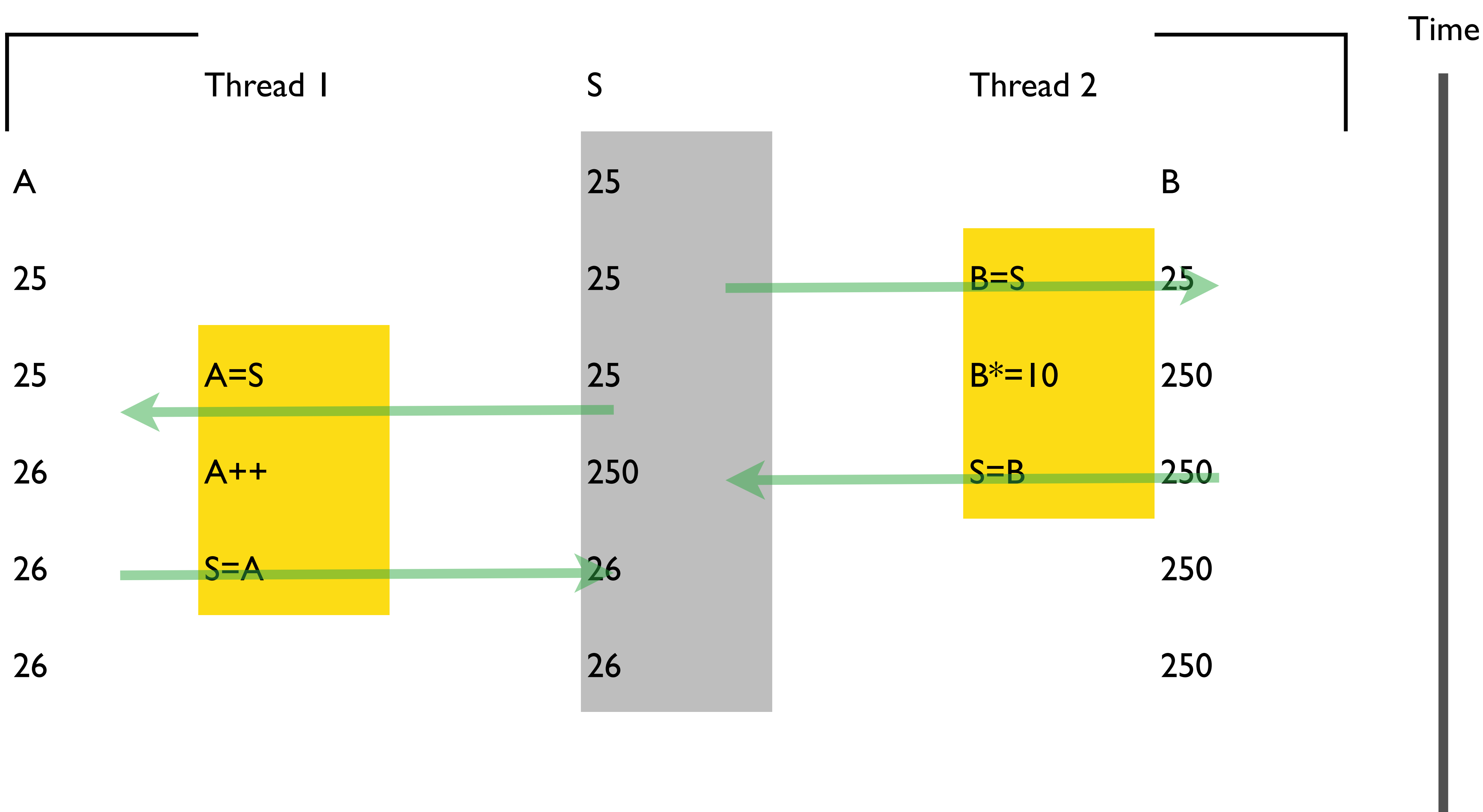
- With every thread we can associate a ‘write-set’:
  - the set of memory locations to which that thread writes.
- We’ve looked at situations where the threads can operate independently -- the ‘write sets’ of the threads don’t intersect.
- Where the write sets intersect, we must ensure that independent thread writes do not damage the data.

Shared Variable S: Thread 1 *before* Thread 2



Thread 2 accesses S after Thread 1 but before Thread 1 has finished updating S

Shared Variable S: Thread 1 *after* Thread 2



Thread 1 accesses S after Thread 2 but before Thread 2 has finished modifying S

Synchronisation

- Problem: Access to shared resources can be dangerous.
  - These are so-called ‘critical’ accesses.
- Solution. Critical accesses should be made exclusive. Thus, all critical accesses to a resource are *mutually exclusive*.
- In the example, both threads should have asked for exclusive access before making their updates.
  - Depending on timing, one or the other would get exclusive access first. The other would have to wait to get any kind of access.

Snchronisation

- A wide variety of algorithms have been devised that ensure synchronisation of competing threads:
  - “Semaphores”, “Monitors”, …
- All are very tricky to get right
- A helpful way to get simpler guarantees of correct behaviour is to have hardware support.
  - An “atomic” test-and-set instruction
    (read a memory location, check its value, and update the value if needed)
  - An “atomic” operation is one that either performs ALL its steps (without interruption)
    or does NONE of them.
    It never does just SOME of them.

Ensuring Mutual Exclusion.

- Mutual Exclusion is a form of thread synchronisation
  - Used to manage access to shared resources (data, IO hardware, shared code, etc.)
- Key idea:
  - There is a "lock" associated with the resource
  - It is locked by a thread when it is using the resource, and unlocked by that thread when done
  - Any other thread must wait for it to be unlocked before proceeding to use that resource.
- accomplished using special "*mutex* variables".

Accessing a protected resource

- To access a mutex-protected resource, an agent must acquire a lock on the mutex variable.
  - If the mutex variable is unlocked, it is immediately locked and the agent has acquired it. When finished, the agent must unlock it.
  - If the mutex variable is already locked, the agent has failed to acquire the lock -- the protected resource is in exclusive use by someone else.
    - The agent is usually blocked until lock is acquired.
    - A non-blocking version of lock acquisition is available.
- A mutex variable is a complex data object
  - It needs to record if it is locked or unlocked (obviously).
  - It needs to record which thread has acquired the lock.
  - It also needs to keep track of which threads are waiting for it to be unlocked.

Shared Variable S Protected by Mutex M

Time

A	Thread 1	S	Thread 2	B	M
	Lock(M)	25			locked
25	A=S	25	Lock(M)		locked
26	A++	25	[thread blocked]		locked
26	S=A	26			locked
26	Unlock(M)	26			locked
26		26	B=S	26	locked
		26	B*=10	260	locked
		260	S=B	260	locked
		260	Unlock(M)		locked
		260			unlocked

Create pthread mutex variable

- Static:
  - pthread_mutex_t m = PTHREAD_MUTEX_INITIALISER;
    - Initially unlocked
- Dynamic
  - pthread_mutex_init(<ref to mutex variable>,attributes)

Lock and Unlock Mutex

- pthread_mutex_lock(<mutex variable reference>);
  - acquire lock or block while waiting
- pthread_mutex_trylock(<mutex variable reference>);
  - non-blocking; check returned code
- pthread_mutex_unlock(<mutex variable reference>);
  - Should only ever be called by the thread that has the lock!

Problems

- Voluntary
  - Mutexes ‘protect’ *code*, and only if all thread code uses them properly
  - The use of a mutex lock/unlock is a protocol that threads should follow so that code behaves well.
    - It cannot enforce good behaviour
  - Other programmers don’t have to use them to get access to the protected resource
    - This is part of the tradeoff. Use processes rather than threads if you want better protection.
- Unfair
  - If multiple threads are blocked on a mutex, the order in which they waken up is not guaranteed to be any particular order.

$$\int_0^4 (16 - x^2)dx = 42.6666\ldots$$

- We break the interval [0..4] into a thousand vertical slices (`H`=0.004),
  - use the "trapezoidal rule" to calculate the `area` of each slice
  - sum all of these `areas` into an `answer` variable.
- Sequential solution in `seq-integrate.c` gives an answer of 42.666656
- Concurrent solution in `nom-integrate.c` usually gives the same answer
  - but occasionally gives results like 42.539102 or 42.602656 or 42.602853 or ….
- The problem arise when two threads try to perform

```
answer = answer + area ;
```

  at almost exactly the same time.

Integration with mutex

- We finally decide to use a mutex (mutex-integrate.c)
- We get the same results as the sequential code.

mutex-integrate (mutex & thread)

```
pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

void *IntegratePart(void *i) {

  double a,b,area;
  int rc;

  a = (int)i * H ;
  b = a + H;
  area = trapezoid(a,b);

  // critical section with mutex
  rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

  answer=answer+area;

  rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_lock()\n",rc);

    pthread_exit(NULL);
}
```

Body is same as for `nom-integrate`

integration example wrap-up

- We can parallelise integration, but mutexes are required
- If we omit them, then errors may occur
- The worst case is when such errors are rare
  - such errors may not be revealed by testing
  - e.g. Mars Rover flash memory bug.
- What about speed?
  - Mutex usage statistics: http://0pointer.de/blog/projects/mutrace.html