Thread Synchronization

Outline

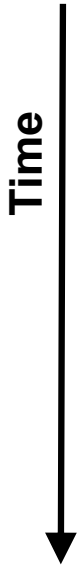
- Interleavings of Threads
  - Race conditions
- Lock-based Thread Synchronization
  - Mutexes
  - Semaphores
- Potential problems
  - Deadlocks
    - Lock contention & lock granularity
  - Livelocks
  - Starvation
- Examples
- More Lock-based Thread Synchronization
  - Barriers
  - Condition variables

Example: Bank account

```
void withdraw_money(int withdrawal) {
    int total = GetTotalFromAccount(); /* total funds in account */

    /* check whether the user has enough funds in account */
    if (total < withdrawal)
        error("You do not have that much money!");

    /* OK, the user has enough money,
       deduct the withdrawal amount from the total */
    total -= withdrawal;
    update_total_funds(total);
}
```

Bank account

```
void withdraw_money(int withdrawal) {
    int total = GetTotalFromAccount(); /* total funds in account */

    /* check whether the user has enough funds in account */
    if (total < withdrawal)
        error("You do not have that much money!");

    /* OK, the user has enough money,
           deduct the withdrawal amount from the total */
    total -= withdrawal;
    update_total_funds(total);
}
```

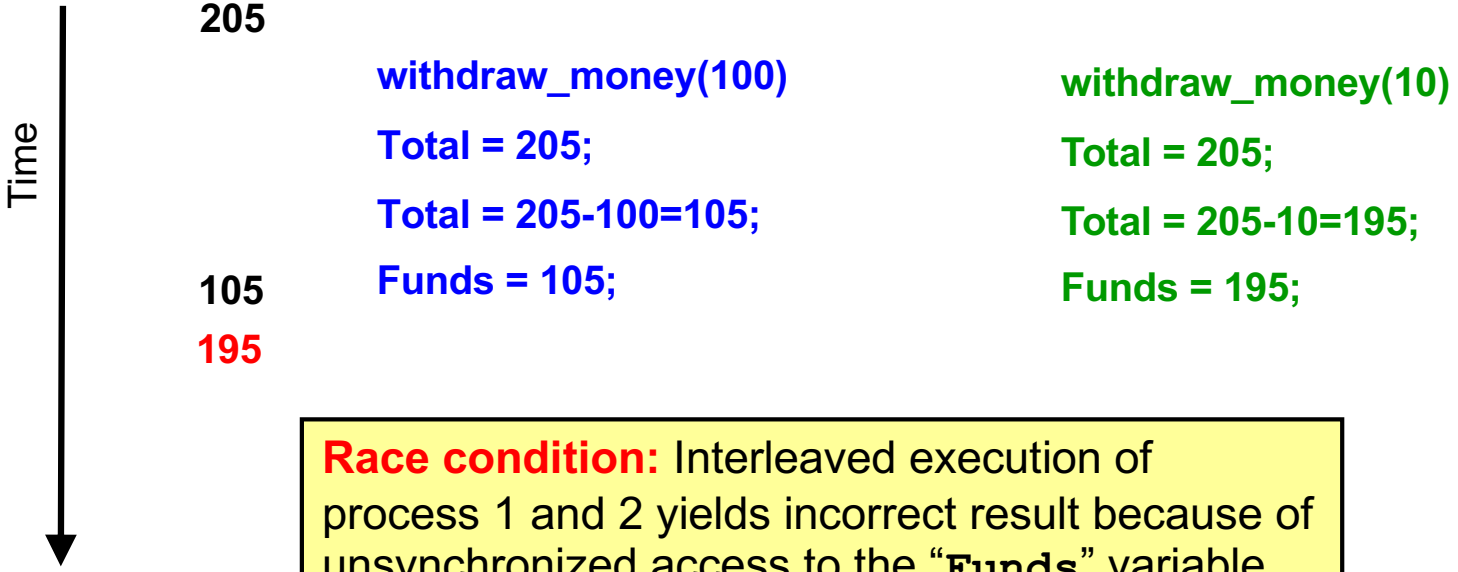


Funds	Thread 1	Thread 2
205		
	withdraw_money(100)	
	Total = 205;	
	Total = 205-100=105;	
105	Funds = 105;	
		withdraw_money(10)
		Total = 105;
		Total = 105-10=95;
95		Funds = 95;

Serialized execution of thread 1 and 2 yields correct result.

Bank account (cont.)

```
void withdraw_money(int withdrawal) {
    int total = GetTotalFromAccount(); /* total funds in account */

    /* check whether the user has enough funds in account */
    if (total < withdrawal)
        error("You do not have that much money!");

    /* OK, the user has enough money,
           deduct the withdrawal amount from the total */
    total -= withdrawal;
    update_total_funds(total);
}
```



Race condition: Interleaved execution of process 1 and 2 yields incorrect result because of unsynchronized access to the "**`Funds`**" variable.

Mutual Exclusion

Only one thread must be in the **critical section** at any time.

- Once a thread is in the critical section, no other thread can enter that critical section until the first thread has left that critical section.
- Serializes access to critical section.

Race Conditions

- **Race condition:** "a *situation in which multiple threads read and write a shared data item and the final result depends on the relative timing of their execution".*

critical section

```
void withdraw_money(int withdrawal) {
    int total = GetTotalFromAccount(); /* total funds in account */

    /* check whether the user has enough funds in account */
    if (total < withdrawal)
        error("You do not have that much money!");

    /* OK, the user has enough money,
           deduct the withdrawal amount from the total */
    total -= withdrawal;
    update_total_funds(total);
}
```

- **Critical section**: two or more code-parts that access and manipulate shared data (aka a shared resource).
- To prevent races, it must **not** happen that 2 threads concurrently execute within a critical section (mutual exclusion between threads).

The Smallest Shared Resource

- Modifying a shared variable such as a single bit or an integer is already a critical section.
- Example: **i++;**
  The compiler translates this statement to machine code, resembling the following pseudo code:

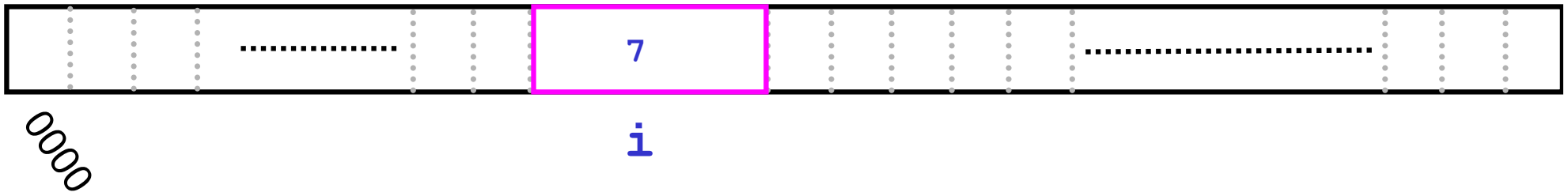
  1) **read the current value of i from memory and copy it into a register**
  2) **add 1 to the value stored in the register**
  3) **write back the new value of i from the register to memory**

- Executing the critical section concurrently leads to a race condition (read-modify-write).
  - See example on next slides.

- Note: processors like the Intel x86 cannot directly modify data in memory. Instead, they have to load data from memory to a register, modify it, and write it back to memory.
- Called "load/store architectures"

The Smallest Shared Resource (cont.)

- Serialized execution yields the correct result (9):



Thread 1: **i++**

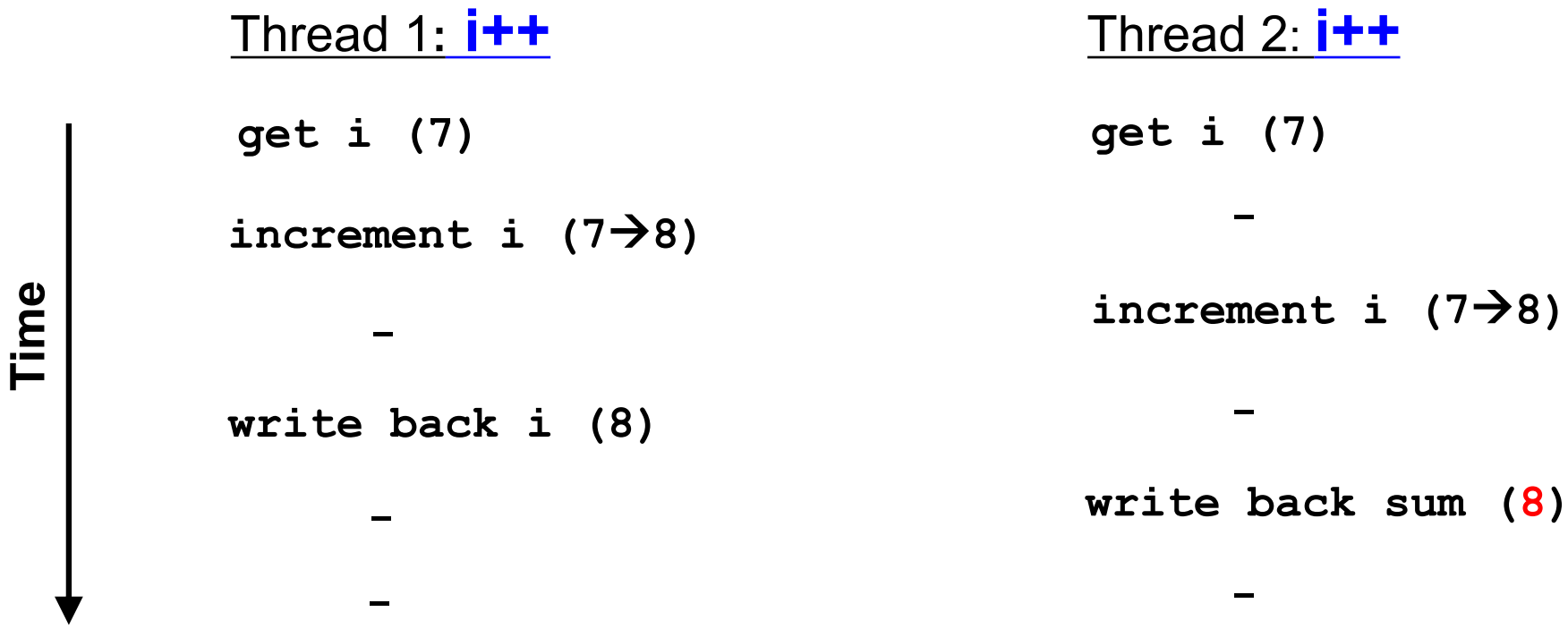
Thread 2: **i++**

Time

```
get i (7)                 -
increment i (7→8)         -
write back i (8)          -

-                         get i (8)
-                         increment i (8→9)
-                         write back i (9)
```

The Smallest Shared Resource (cont.)

- Interleaved execution (both threads simultaneously executing the critical section) results in i=8 instead of i=9 because of race condition:



- Race conditions are hard to find, because they are hard to reproduce!

Another example...

```
#define MAX 2
#define MAX_ITER 1000000000

long counter = 0;

void * thread_function(void * arg) {
    long l;
    for (l = 0; l < MAX_ITER; l++)
       counter = counter + 1; // critical section
}

int main(void) {
  pthread_t mythread[MAX]; int t = 0;

  for (t = 0; t < MAX; t++)
    pthread_create( &mythread[t], NULL, thread_function, NULL);

  for (t = 0; t < MAX; t++)
    pthread_join ( mythread[t], NULL);

  printf("Expected counter value: %lld, observed counter value: %lld\n",
         MAX_ITER * MAX, counter);
  printf("You experienced %lld race conditions,
          or %d%% of all accesses to the shared variable.\n",
          MAX_ITER * MAX - counter, 100 - 100 * counter / (MAX_ITER * MAX));
}
```

- Here the critical section is the statement where variable counter is read and written.
- If both threads read the value of counter at the same time, they will both add 1 to this value.
- Writing back the value to counter, one thread overwrites the result of the other thread.
- Thereby one counter increment gets lost.
- This is again a race condition.

Another example... (cont.)

- If you run the code from the previous slide you can observe race conditions.

- Access to variable **`counter`** is not synchronized.
  - The expected counter value with 2 threads is 2000000000.
  - The real counter value is always much smaller, because of race conditions between the two threads simultaneously updating the counter.

```
[comp2129@ucpu0 pthread1]$ gcc -lpthread parallucpu0count.c
[comp2129@ucpu0 pthread1]$ ./a.out
Expected counter value: 2000000000, observed counter value: 1010509326
You experienced 989490674 race conditions, or 50% of all accesses to the shared variable.
[comp2129@ucpu0 pthread1]$ ./a.out
Expected counter value: 2000000000, observed counter value: 1018887394
You experienced 981112606 race conditions, or 50% of all accesses to the shared variable.
[comp2129@ucpu0 pthread1]$ ./a.out
Expected counter value: 2000000000, observed counter value: 1025761570
You experienced 974238430 race conditions, or 49% of all accesses to the shared variable.
```

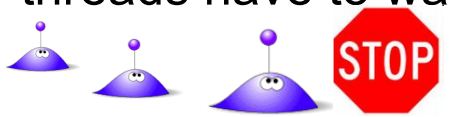
Outline

- Interleavings of Threads ✓
  - Race conditions ✓
- Lock-based Thread Synchronization
  - **Mutexes** ← next
  - Semaphores
- Potential problems
  - Lock contention & lock granularity
  - Deadlocks
  - Livelocks
  - Starvation
- Examples

Mutexes

```
1  #define MAX_ITER 1000000000
2  long counter = 0;
3  pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;
4
5  void * thread_function(void * arg) {
6      long l;
7      for (l=0; l < MAX_ITER; l++) {
8          pthread_mutex_lock(&mylock);
9          counter = counter + 1; //critical section
10         pthread_mutex_unlock(&mylock);
11     }
12 }
```

- The Pthread library provides **mutexes** to synchronize access to shared global variables (shared between threads).
  - (Variable **`counter`** in the above example is a shared global variable.)
- In line 3, a mutex named "**`mylock`**" is declared.
- Before a thread enters the critical section, it *locks* the mutex (Line 8).
- When a thread leaves the critical section, it *unlocks* the mutex (Line 10).
- The mutex ensures that at any time, only one thread will be allowed into the critical section. All other threads have to wait until the critical section becomes free again.



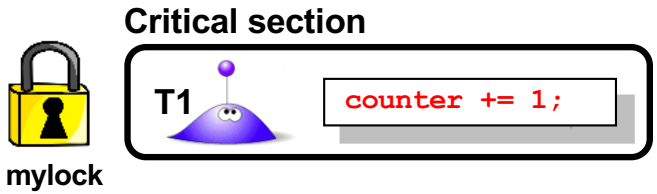
Mutexes (cont.)

```
1  pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;
2
3  void * thread_function(void * arg) {
4          ...
5          pthread_mutex_lock(&mylock);
6          counter = counter + 1; //critical section
7          pthread_mutex_unlock(&mylock);
8  }
```

- Line 1: Static declaration of a mutex.
- Mutexes have attributes.
- Statically declared mutexes can be initialized to default attributes using PTHREAD_MUTEX_INITIALIZER.

- A mutex protects shared data (by allowing only 1 thread into critical section)
- Initially the mutex is free (or unlocked):

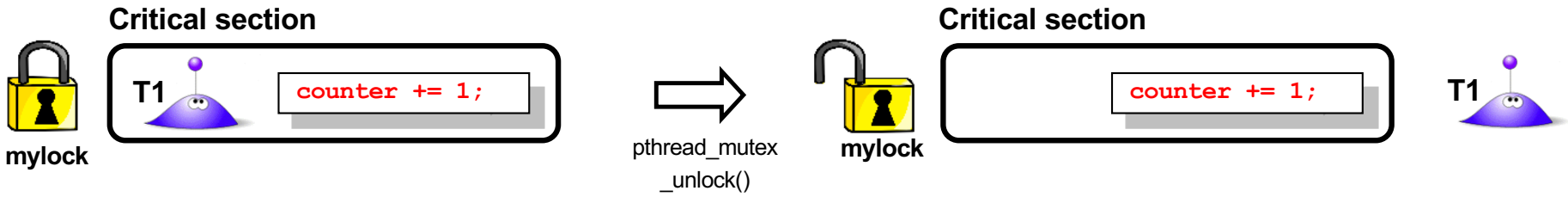
- By calling pthread_mutex_lock(), a thread can *acquire* the lock:



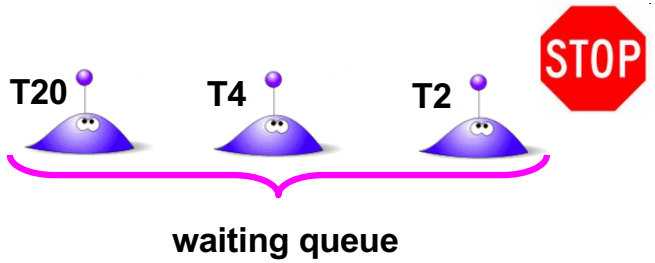
- mylock is locked.
- Thread T1 holds mylock.

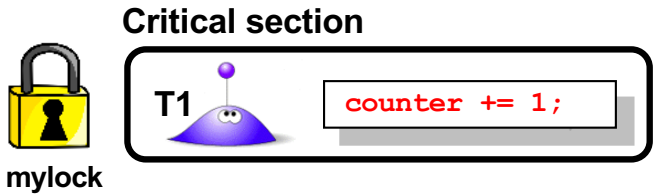
Mutexes (cont.)

- By calling pthread_mutex_unlock(), a thread can *relinquish* the mutex:



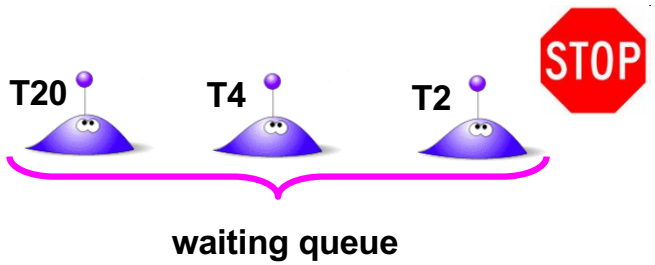
- While a mutex is locked, other threads that call pthread_mutex_lock() are **blocked**.
  - The call to pthread_mutex_lock() will not return before the calling thread was given the lock.
  - Blocked threads are kept in a waiting queue. They have to wait until the thread in the critical section relinquishes the lock.

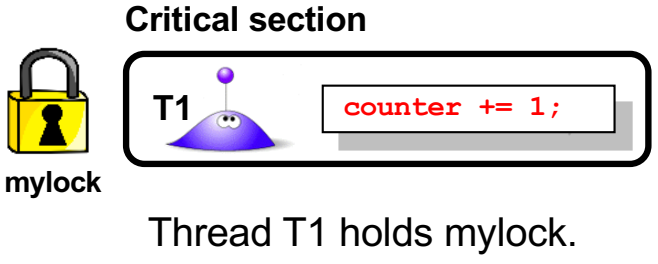




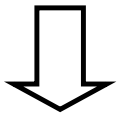
Mutexes (cont.)

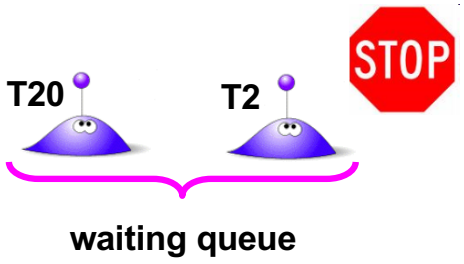
- When the thread in the critical section relinquishes the lock, a thread from the waiting queue will be allowed into the critical section.
  - The Pthread library does not guarantee *fairness*:
    - Not necessarily the thread being blocked for the longest time will be selected.

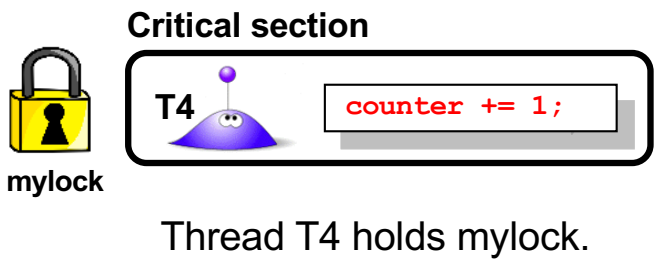




Thread T1 holds mylock.







Thread T4 holds mylock.

Mutexes (cont.)

```
1  pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;
2
3  void * thread_function(void * arg) {
4          ...
5          if ( 0 != pthread_mutex_trylock(&mylock) ) {
6              printf("Unsuccessful attempt to acquire lock\n");
7          } else {
8              // critical section goes here:
9              ...
10             pthread_mutex_unlock(&mylock);
11         }
12 }
```

Assume we do not want a thread to *block* at a mutex:

- **`pthread_mutex_trylock()`** attempts to acquire the lock.
  - If the lock is free, then the calling thread will acquire it.
  - If the lock is not free, then the call to pthread_mutex_trylock() will return immediately.
    - Instead of block in the waiting queue.
- The return value of **`pthread_mutex_trylock()`** is used to distinguish the above cases (see Line 5 in above example).

Mutexes (cont.)

```
1  void * thread_function(void * arg) {
2          ...
3          if ( 0 != pthread_mutex_trylock(&mylock) ) {
4              printf("Unsuccessful attempt to acquire lock\n");
5              pthread_mutex_unlock(&mylock); // Error: mutex not acquired!
6          } else {
7              // critical section goes here:
8              ...
9              pthread_mutex_unlock(&mylock);
10         }
11 }
```

Only the thread that owns a mutex should *unlock* it!

- If a thread unlocks a mutex that it does not own, the behavior is undefined (anything can happen, don't do it!).
  - The thread could unlock the mutex for the thread which is currently in the critical section! (a race condition may arise if another thread acquires the mutex).
  - Unlocking an unlocked mutex also results in undefined behavior.
- A frequent error is to unlock a mutex after an unsuccessful **`pthread_mutex_trylock()`** attempt (see Line 5 above).

Dynamic creation of mutexes

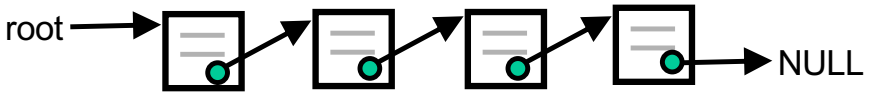
```
1  pthread_mutex_t  * mylock;
2
3 mylock = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
4 pthread_mutex_init(mylock, NULL);
5 ...
6  pthread_mutex_destroy(mylock);
7  free (mylock);
```

Mutexes can also be created at run-time.

- In Line 3, memory is allocated to hold a variable of type **`phtread_mutex_t`.**
- **`pthread_mutex_init()`** is used to initialize the mutex.
  - The second parameter is **`NULL`**, meaning that we use default attributes.
    - Sufficient for our use.
- When the mutex is no longer used:
  - **`pthread_mutex_destroy()`** destroys the mutex.
  - The memory allocated for the mutex must be free-ed by the programmer (Line 7)!

Dynamic creation of mutexes (cont.)

- Assume a linked list dynamic data structure:



- Assume a list of nodes where each node represents a counter, and multiple threads may access the same counter. We need to synchronize access to counters.
  - Assume the following list node struct and node allocation:

```
struct Node {
  pthread_mutex_t node_lock; // mutex to lock this list node
  int counter;               // data in this list node
  struct Node * next;        // pointer to next node in the list
};
```

```
struct Node * root = NULL;  // declare a pointer to a node
root = malloc (sizeof(struct Node)); // create the node
root->counter = 0;  // set data in node to 0
root->next = NULL;  // set the next pointer in node to NULL
pthread_mutex_init (&root->node_lock, NULL); // init the mutex
```

Programming with Mutexes

```
void * thread_function_1(void * arg) {
  ...
  pthread_mutex_lock(&mylock);
  counter += 1; //critical section
  pthread_mutex_unlock(&mylock);
}
```

```
void * thread_function_2(void * arg) {
  ...
  counter += 1; //critical section
  ...
}
```

Here the programmer forgot to protect the critical section by the corresponding mutex!

- Mutexes are *advisory* and *voluntary*.
  - **Problem**: Nothing prevents a programmer to access a shared data-structure without acquiring the mutex!
  - Such bad practice will again result in race conditions and corruption of shared data.
    - Especially a problem with large programs like the Linux kernel
    - Many developers work on kernel code, e.g., drivers.
    - Easy to get locking wrong when accessing other people's data structures!
  - Example: in **`thread_function_2()`** above, the programmer forgot to protect the critical section by mutex **`mylock`**.
  - One way around this problem is to encapsulate access to shared data-structures in a so-called monitor (can be class in C++ or module in C, or a language-primitive).
    - (see next slide)

Programming with Mutexes (cont.)

```
static long counter = 0;
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

void increment_ctr(void) {
  pthread_mutex_lock(&mylock);
  counter += 1; //critical section
  pthread_mutex_unlock(&mylock);
}

long read_ctr(void) {
  long ret;
  pthread_mutex_lock(&mylock);
  ret = counter; //critical section
  pthread_mutex_unlock(&mylock);
  return ret;
}
```

Counter-component in C, aka MONITOR

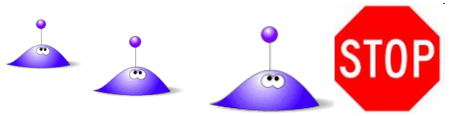
MONITOR = ADT + Synchronization

Concept of monitor:

- At any time, at most one thread can execute any monitor function (mutual exclusion!).
- Pthread library does not support monitors, so we made our own using mutexes.
- Modern programming languages support monitors: Java, C#, Ruby…

- This solution hides the counter variable from direct access by the programmer.
- The only way to access the counter is via the provided functions **`increment_ctr()`** and **`read_ctr()`**.
  - **Problem**: Now every access is via a function call, which is less efficient than direct access.
    - Optimizing compilers might be able to inline the above functions, thereby reducing the function call overhead.
- Locking-responsibility is shifted from the user of the counter to the maintainer (programmer) of the counter-component.

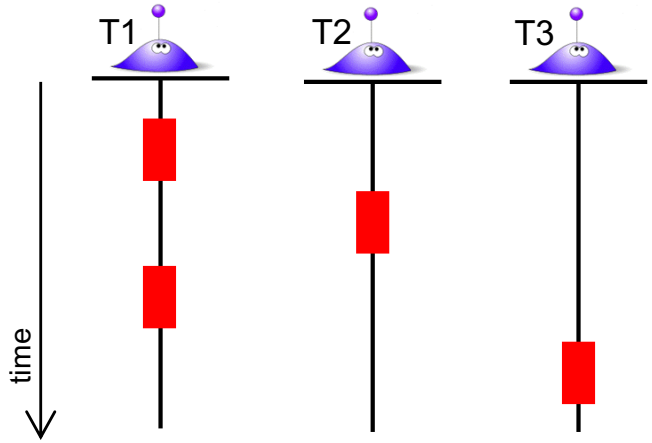
Serialization




```
1   unsigned long counter=0;
2
3   void * thread_function(void * arg) {
4       long l;
5       for (l=0; l < MAX_ITER; l++) {
6           pthread_mutex_lock(&mylock);
7           counter = counter + LongComputation();
8           pthread_mutex_unlock(&mylock);
9       }
10  }
```

```
counter = counter +
        LongComputation();
```

- At any one time, only one thread can be in critical section.
  - All other threads have to wait (block) until critical section becomes free.
  - Threads execute critical section one after the other, no parallelism!
  - This is called **serialization.**



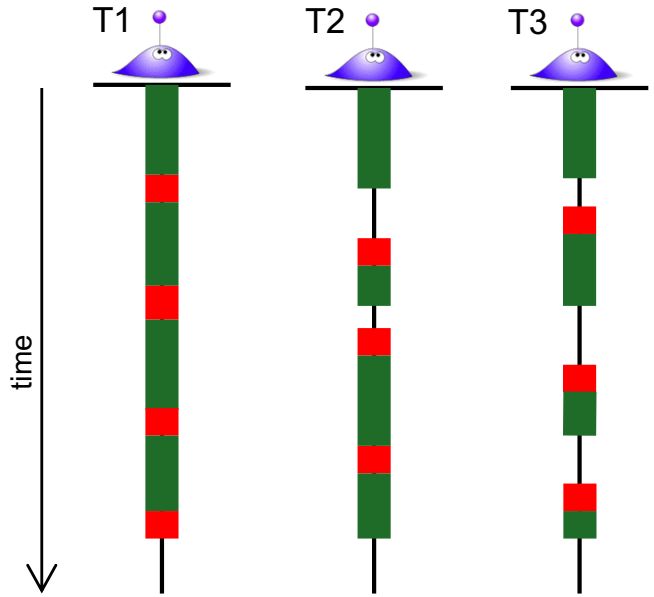
thread executing critical section

thread blocked

Serialization (cont.)

```
1   unsigned long counter=0;
2
3   void * thread_function(void * arg) {
4       long l, tmp;
5       for (l=0; l < MAX_ITER; l++) {
6           tmp = LongComputation();
6           pthread_mutex_lock(&mylock);
7           counter=counter+tmp;//crit.sect.
8           pthread_mutex_unlock(&mylock);
9       }
10  }
```

- We can place computations that do not access shared data outside of the critical section.
- Reduces serialization, increases the amount of parallel computation.
  - Example: **`tmp=LongComputation();`**
  - Note: this is not possible if **`LongComputation()`** accesses the shared counter!



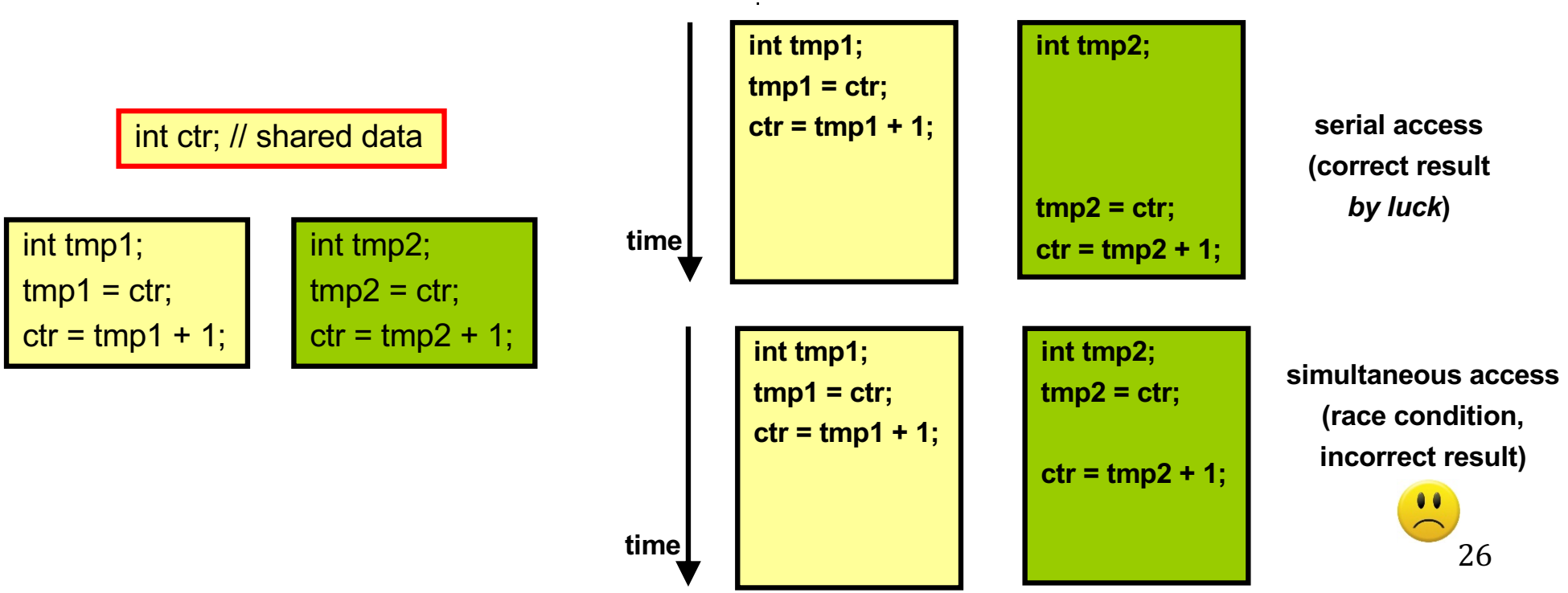
thread executing **`LongComputation()`**

thread executing critical section

thread blocked

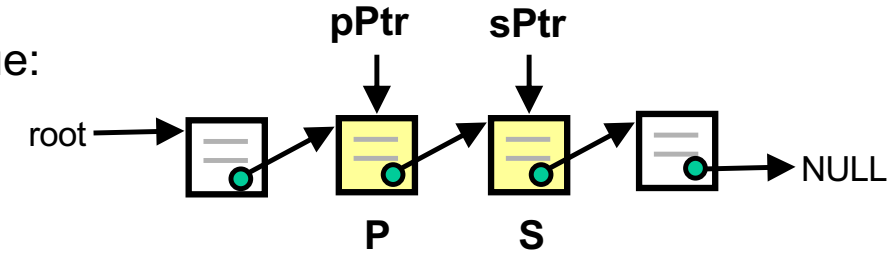
Programming with Mutexes

- What **data** needs protection? Ask those questions:
  - **Is data shared between several threads ?**
- A race condition happens when two threads access a variable simultaneously, and (at least) one access is a *write*.
  - Thus: simultaneous *reads* of a variable are not a problem!



Acquiring multiple mutexes

- For a given programming problem a thread might have to acquire more than one mutex.
- Example: splitting the counter value between a node S and his predecessor node P
  - Node S, plus Node P (the node before S) must be locked!
    - **Otherwise we have a race condition if a thread tries to update S and/or P in parallel!**
  - Pseudocode to split S's counter value:
    1) lock(pPtr->nlock);
    2) lock(sPtr->nlock);
    3) pPtr->ctr += sPtr->ctr/2;
    4) sPtr->ctr -= sPtr->ctr/2;
    5) unlock(pPtr->nlock);
    6) unlock(sPtr->nlock);




```
struct Node {
  pthread_mutex_t nlock;
  int ctr;
  struct Node * next;
};
```

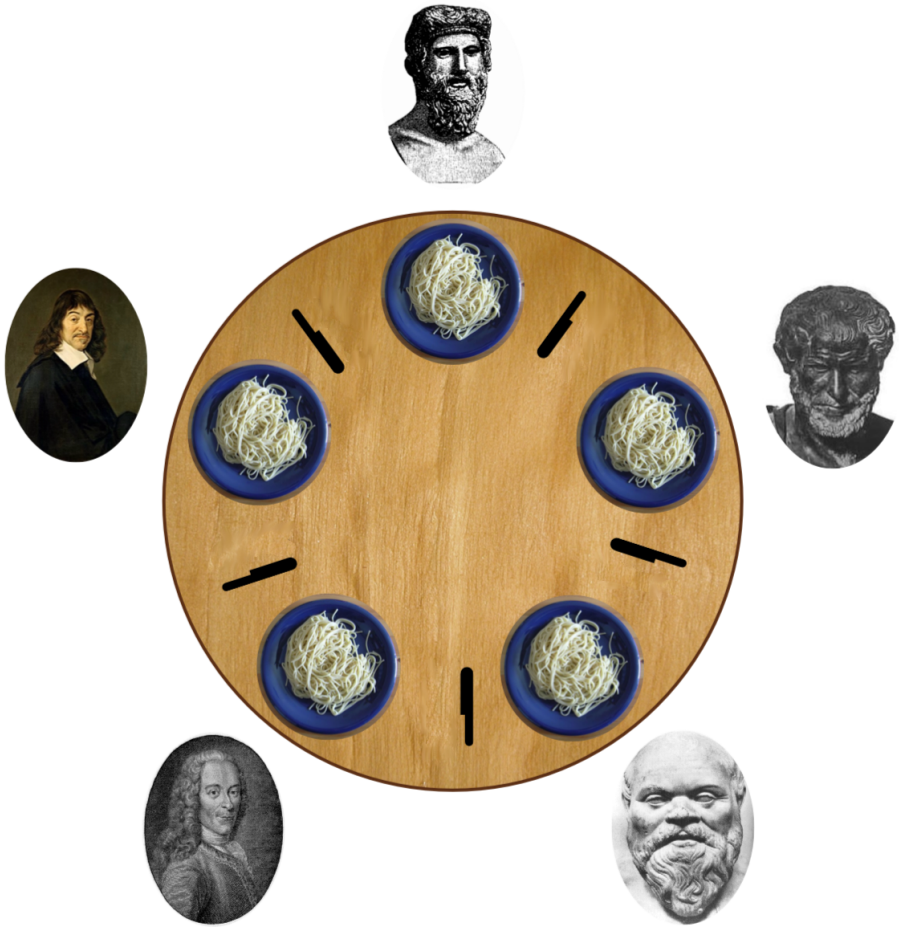
- Acquiring multiple mutexes can result in *deadlock*.
  - See example on next slide.

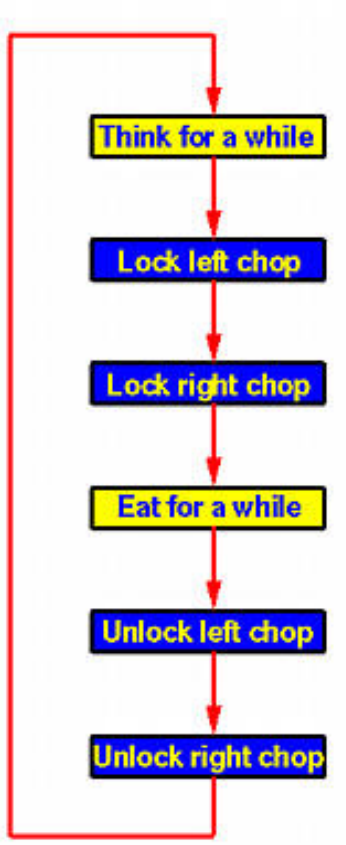
Outline

- Interleavings of Threads ✓
  - Race conditions ✓
- Lock-based Thread Synchronization
  - Mutexes ✓
  - Semaphores
- Potential problems
  - Lock contention & lock granularity
  - **Deadlocks**
  - Livelocks
  - Starvation ← **next**
- Examples

Deadlocks

- A *deadlock* is a condition involving one or more threads and one or more resources (e.g., mutexes), such that each thread is waiting for one of the resources, but all the resources are already held → system locks, no more progress possible.





Deadlocks

- **Self-Deadlock**: a thread attempts to acquire a mutex that it already holds. While waiting for the mutex, it does not release it → deadlock.

```
pthread_mutex_lock(&mylock); // Acquire mylock
pthread_mutex_lock(&mylock); // Try to acquire mylock again!
                             // Get blocked and wait for mylock
                             // to become available...
```

- ABBA-Deadlock: 2 threads attempting to acquire 2 mutexes A and B. One in the order A→B, the other in the order B→A.

Thread 0:

```
acquire mutex A
try to acquire mutex B
wait (block) for mutex B
```

Thread 1:

```
acquire mutex B
try to acquire mutex A
wait (block) for mutex A
```

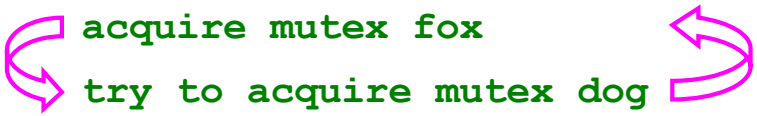
Multiple mutexes must always be obtained in the same order!

- Assume 3 mutexes: cat, dog, fox that secure three data-structures.
- Assume a function that needs to work on all three data-structures at the same time.
  - This function must acquire mutexes cat, dog and fox.
- If this function acquires the mutexes in the order cat → dog → fox, then every other function must obtain these locks (or a subset of them) **in the SAME ORDER.**
- Otherwise a deadlock may result:

Thread 0:

```
acquire mutex cat
acquire mutex dog
try to acquire mutex fox
wait for mutex fox
```

Thread 1:

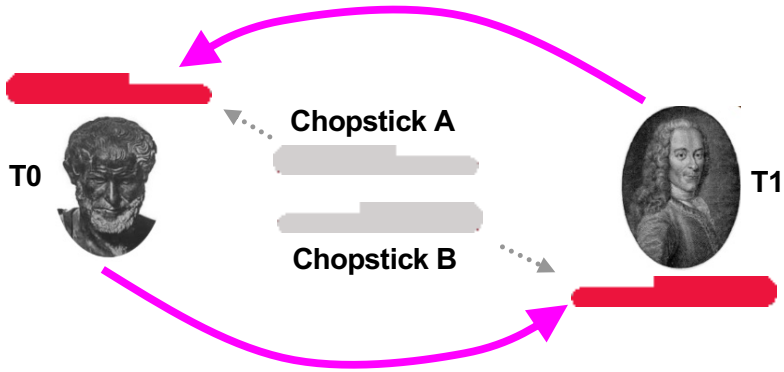


Necessary conditions for a deadlock

There are four *necessary* conditions for a deadlock:

1. **Mutual exclusion**: a resource can be assigned to at most one thread.
   - Example: a chopstick is a resource. It can be assigned to at most one philosopher at any time.
2. **Hold and wait**: threads both hold resources and request other resources.
   - Example: a philosopher holds the left chopstick and requests the right chopstick.
3. **No preemption**: a resource can only be released by the thread that holds it.
   - Example: it is not possible to forcibly take away a chopstick from a philosopher.
4. **Circular wait**: a cycle exists in which each thread waits for a resource that is assigned to another thread.
   - Example (case of 2 dining philosophers, ABBA-deadlock):

```
acquire chopstick A
try to acquire chopstick B
wait for chopstick B
```




```
acquire chopstick B
try to acquire chopstick A
wait for chopstick A
```

Deadlock Prevention

- One way to prevent deadlocks is to prevent circular waits (cycles).
  - prevent Condition 4 from the previous slide
- Cycles can prevented by a **locking hierarchy:**
  1) Impose an ordering on mutexes.
     1) Example: **A→B→C→D**
  2) Require that all threads acquire mutexes in the same order.

- Locking hierarchy requires programmers to know in advance what mutexes a thread will need.
- Example: It is not allowed to acquire mutexes B, C and D and only then decide to acquire A also.
  - Need to release B, C and D and reacquire A→B→C→D,
  - or use pthread_mutex_trylock(A) and proceed if A can be acquired.
    - if not, then release B, C and D and reacquire A→B→C→D.
- See examples on next slides.

Deadlock Prevention Example 1

- Ordering is vital: required mutexes must always be obtained in the same order, e.g., A->B→C.
  - **Example 1:** assume two threads need mutexes A, B and C.
    - Both threads acquire mutexes in the order A->B->C.
    - Only one thread will be able to acquire A, the other thread has to block...
    - No deadlock possible, because the blocking thread did not acquire B or C before.
    - The blocking thread can continue only after the mutexes have been released.

Thread 1:

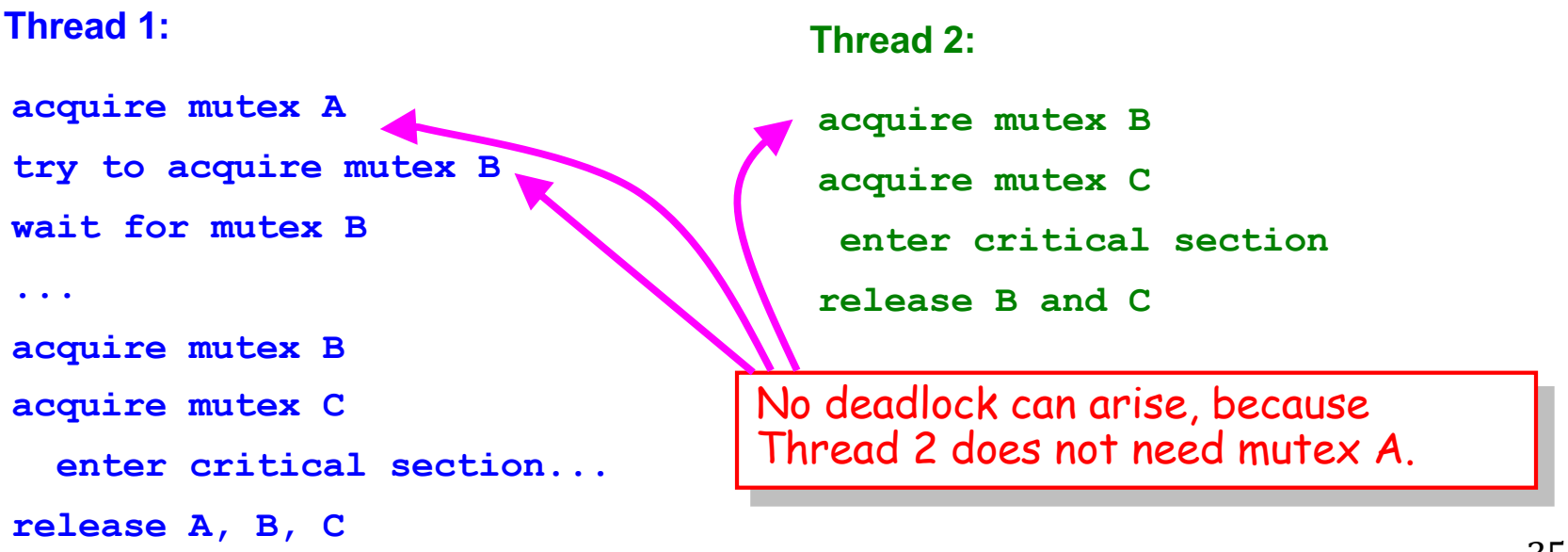
```
acquire mutex A
acquire mutex B
acquire mutex C
  enter critical section...
release A, B and C
```

Thread 2:

```
try to acquire mutex A
wait for mutex A
....
....
....
acquire mutex A
acquire mutex B
acquire mutex C
```

Deadlock Prevention Example 2

- Ordering is vital: required mutexes must always be obtained in the same order, e.g., A->B→C.
  - **Example 2:** assume Thread 1 need mutexes A, B and C, Thread 2 needs mutexes B and C.
    - Thread 1 acquires mutexes in the order A->B->C.
    - Thread 2 needs only a subset {B, C}
      - Acquires mutex-subset {B, C} according to the above order (B→C).



Deadlock Prevention Example 3

- Ordering is vital: required mutexes must always be obtained in the same order, e.g., A->B→C.
  - **Example 3:** assume Thread 1 need mutexes A, B and C, Thread 2 needs mutexes B and C. After acquiring B and C, Thread 2 finds out that it needs A as well. But A is already held by Thread 1...

Thread 1:

```
acquire mutex A
try to acquire mutex B
wait for mutex B
...
...
...
acquire mutex B
acquire mutex C
  enter critical section...
release A, B, C
```

Thread 2:

```
acquire mutex B
acquire mutex C
ups, I need mutex A as well!
try_lock(A)// A is already held
           // by Thread 0!
too bad! must release B, C
//start all over: acquire A→B→C
...
```

*

No deadlock can arise, because Thread 2 restarts mutex acquisition when it finds out that mutex A is already acquired by another thread.

Deadlock Prevention Example 4

- Ordering is vital: required mutexes must always be obtained in the same order, e.g., A->B→C.
  - **Example 4:** assume Thread 1 need mutexes A, B and C, Thread 2 needs mutexes B and C. After acquiring B and C, Thread 2 finds out that it needs A as well. Mutex A is not held by another thread...

Thread 1:

```
...
...
...
...
...
...
...
acquire mutex A
acquire mutex B
acquire mutex C
  enter critical section...
release A, B, C
```

Thread 2:

```
acquire mutex B
acquire mutex C
oops, I need mutex A as well!
try_lock(A)// A is avaiable
Great, acquired A as well!
 enter critical section...
release A, B, C
```

No deadlock can arise, because Thread 2 gets mutex A only if A is not held by another thread.

Deadlock Prevention

- Vital to the correct operation of a system.
  - Ordering of mutex acquisition is vital: mutexes must always be obtained in the same order, e.g., A→B→C→...→Z.
    - Called a locking hierarchy.
  - Preventing starvation: Ask: does this code always finish? If *foo* does not occur, will *bar* wait forever?
  - Do not double-acquire the same lock.
  - Complexity in a locking scheme invites deadlocks→design for simplicity.

- The order of *unlock* operations is immaterial! *

Deadlock Prevention

But even with a locking hierarchy things can go wrong:

Thread 0:

```
acquire mutex A
for(;;) // enter endless loop
        // never free mutex A
```

Thread 1:

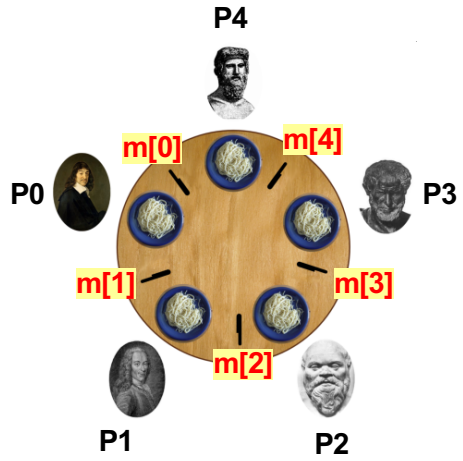
```
try to acquire mutex A
wait (block) forever for mutex A
```

- If a thread that holds a mutex *malfunctions* (e.g., *enters an endless loop)*, then other threads trying to acquire the mutex will block forever.
- Code in critical sections is therefore "even more critical" than other code!

Dining Philosopher's Deadlock

- The previous Dining Philosopher's Algorithm does **not** use a locking hierarchy.

```
#define MAX 5
pthread_t thr[MAX];
pthread_mutex_t m[MAX];

void * tfunc (void * arg) {
  long i = (long) arg; // thread id: 0..4
  for (;;) {
    pthread_mutex_lock( &m[i] );
    pthread_mutex_lock( &m[(i + 1) % MAX] );
    printf("Philosopher %d is eating...\n", i);
    pthread_mutex_unlock(&m[i]);
    pthread_mutex_unlock(&m[(i + 1) % MAX]);
  }
}
```

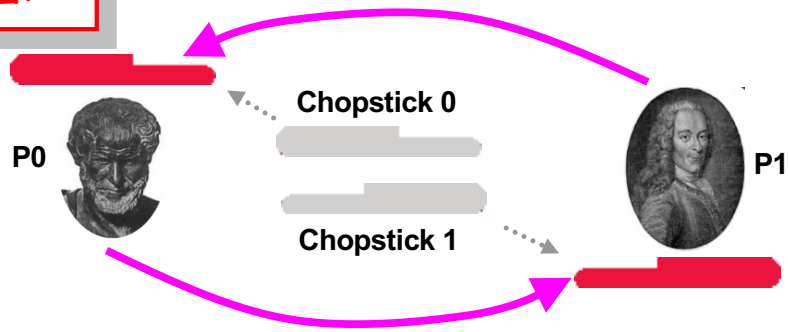


P0,..P3: m[0]→m[1]→m[2]→m[3]→m[4]

P4: m[4]→m[0]

Consider the case for MAX=2:

```
acquire chopstick 0
try to acquire chopstick 1
wait for chopstick 1
```

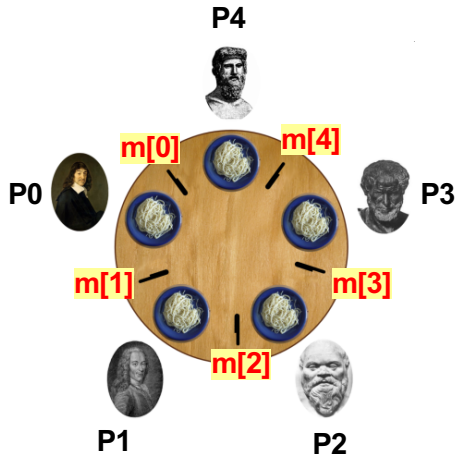



```
acquire chopstick 1
try to acquire chopstick 0
wait for chopstick 0
```

Dining Philosopher's (fixed)

- Introduce a locking hierarchy:
  1) pick up the chopstick with the smaller index.
  2) Pick up the chopstick with the higher index.

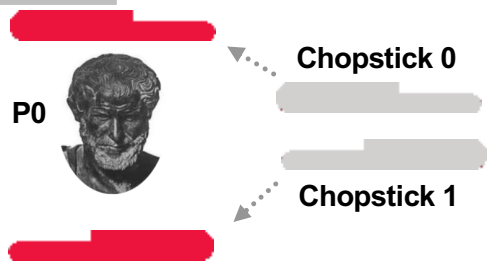
```
for (;;) {
    if ( i < ((i + 1) % MAX) ) {
      pthread_mutex_lock(&mtx[i]);
      pthread_mutex_lock(&mtx[(i + 1) % MAX]);
    } else {
      pthread_mutex_lock(&mtx[(i + 1) % MAX]);
      pthread_mutex_lock(&mtx[i]);
    }
    printf("Philosopher %d is eating...\n", i);
    pthread_mutex_unlock(&mtx[i]);
    pthread_mutex_unlock(&mtx[(i + 1) % MAX]);
}
```



locking hierarchy:
P0,..P4: m[0]→m[1]→m[2]→m[3]→m[4]

Consider the case for MAX=2:

```
acquire chopstick 0
acquire chopstick 1
eat
release chopstick 0
release chopstick 1
```




```
try to acquire chopstick 0
wait for chopstick 0
...
...
...
```

Outline

- Interleavings of Threads ✓
  - Race conditions ✓
- Lock-based Thread Synchronization
  - Mutexes ✓
  - Semaphores
- Potential problems
  - Deadlocks ✓
    - **Lock contention & lock granularity**
  - Livelocks
  - Starvation ← **next**
- Examples

Lock-based synchronization

- On the previous slides, we used mutexes from the pthread library to prevent multiple threads entering a critical section at the same time.
- Other synchronization primitives exist:
  - monitors (introduced in previous slides)
  - semaphores (to be discussed on the next slides)
- Mutexes, semaphores, monitors have one **commonality**:

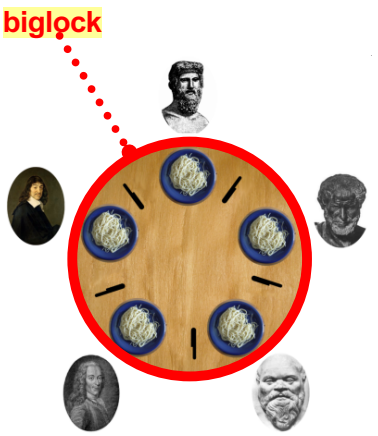
  They **protect shared data** such that

  1) Only one thread can be in the critical section at any time.
  2) Other threads attempting to enter the critical section have to wait (block).
- Mutexes, semaphores, monitors also are called **locks.**
  - Protect shared data by locking a critical section against multiple entry of threads.
  - Note: modification of shared data done in critical sections → no more race conditions possible.
- Synchronization via locks is called **lock-based synchronization**.
  - Related terms that we will discuss: lock contention, scalability of locks.

Lock Contention Example

- A less effective solution for the Dinning Philosophers' Problem:
  - Lock the whole table.

```
#define MAX 8
pthread_t thr[MAX];
pthread_mutex_t biglock;

void * tfunc (void * arg) {
  long i = (long) arg;
  for (;;) {
    pthread_mutex_lock(&biglock); //lock table
    printf("Philosopher %d is eating...\n", i);
    pthread_mutex_unlock(&biglock);
  }
}
```



- Now only one philosopher can eat at any time.
- All other philosophers have to block on the biglock.
- We say that there is **contention** for biglock.
- Many philosophers wait for biglock, biglock is a **highly contended lock**.
  - **A highly contented lock limits parallelism!**

Lock Contention and Scalability

- **Lock Contention**: lock is currently in use, but another thread tries to acquire it. A *highly contended* lock has many threads waiting to acquire it.
  - Reason: Lock frequently obtained, or held for a long time, or both.
- A lock **serializes** activities. A highly contended lock is a bottleneck, limiting parallelism.
- **Scalability**: measurement how well a system can be expanded.
  - Example: adding processors/cores to a server computer.
  - Ideal case: doubling the number of processors/cores doubles the system's performance.
- A highly contended lock limits scalability.
  - All threads queue up, trying to acquire the contended lock.
  - Little parallel activity as a result.

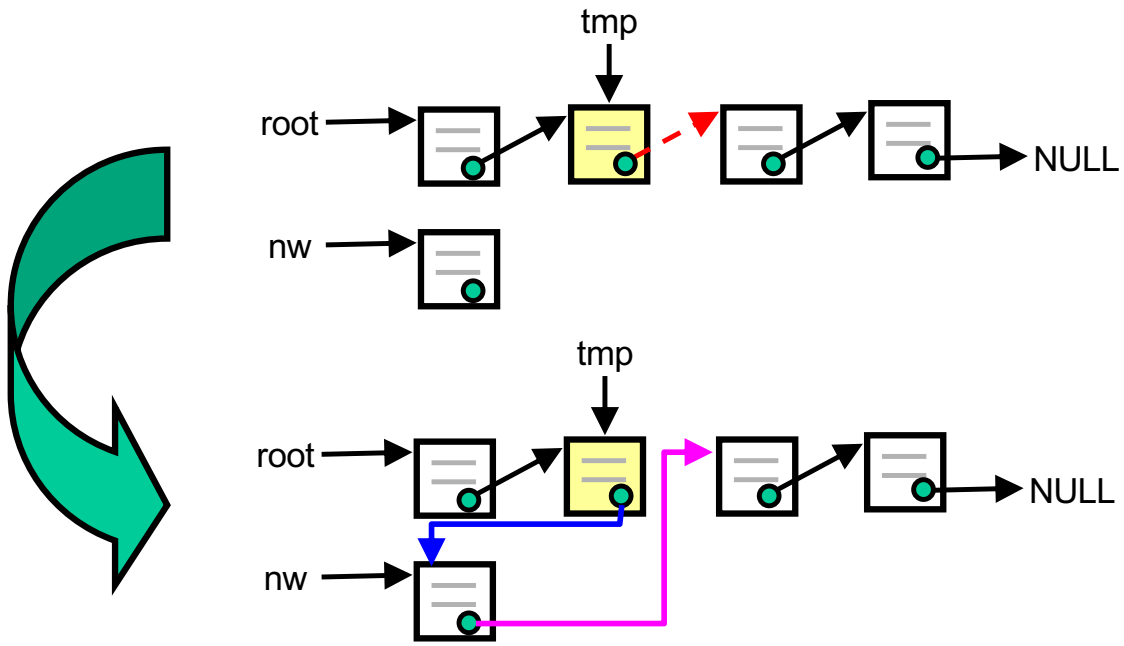
❖ We will discuss lock contention on the next slides.

❖ We will discuss scalability in our lecture on performance.

Example: Lock Contention

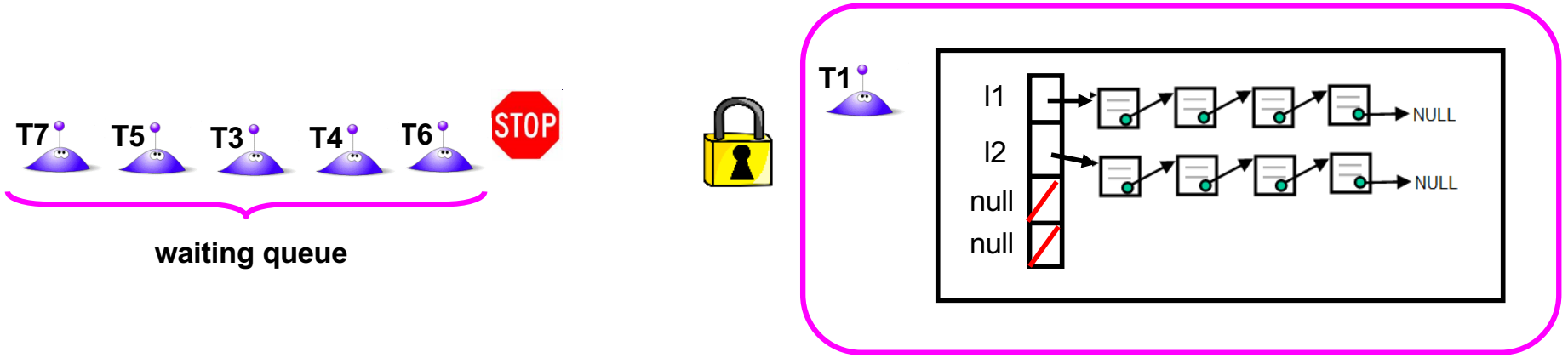
Recall: to insert a new node in a linked list,

1) we must create a new node nw
2) Create a temporary pointer to the node after which the new node is to be inserted
3) Update pointers: set nw.next = tmp.next, and tmp.next = nw:



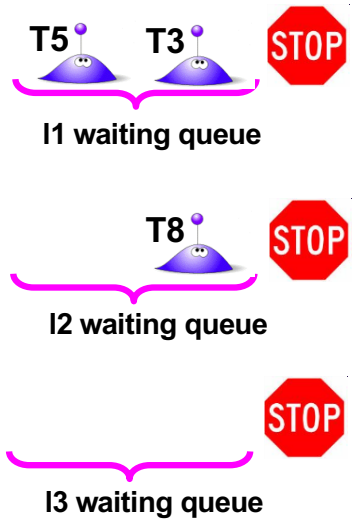
- Assume now that two threads attempt to insert a new node into the list simultaneously.
- Assume that both attempt to insert the node after the yellow node.
- →race condition!
- If multiple threads use our linked list, we need to synchronize access to lists.
- See next slides.

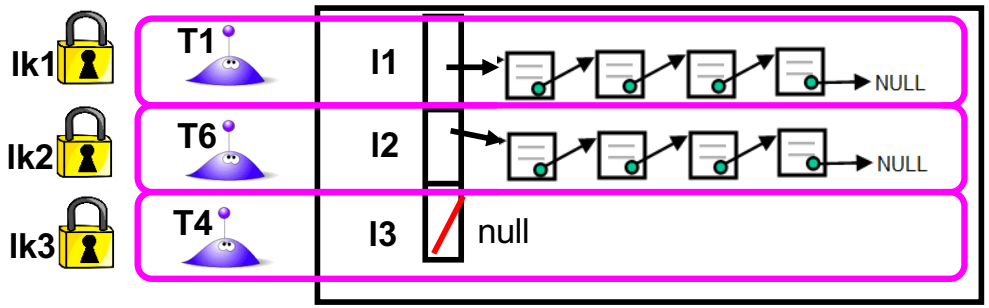
Example: Lock Contention, Try1



- The situation is even more sever for our ADT that implements linked lists.
  - Several threads may want to access the same or different lists simultaneously!
- Without synchronization, race conditions will result.
- We could use one mutex to protect the list ADT.
  - Now all threads have to block at a single mutex to wait for list access.
    - even if they want to access different lists!
  - Reduces parallelism in the application
  - Serializes the program!
  - **Coarse-grained lock**.

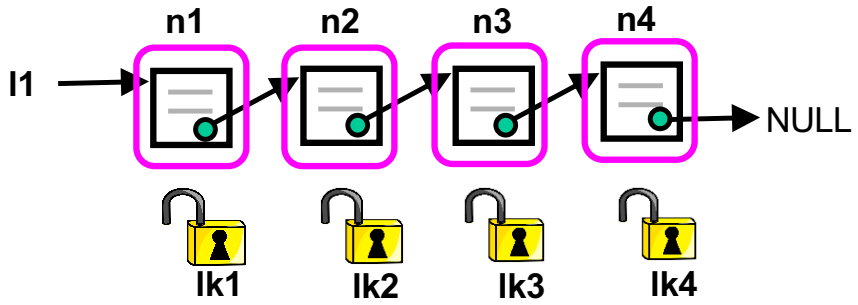
Example: Lock Contention, Try2





- We could use one mutex for *each linked list*.
  - Now the threads have to block at the list that they want to access.
  - Access to different lists can happen simultaneously.
  - More parallelism than with Try1.
  - Still program serialization!
  - **Medium-grained lock.** 
  - **Question: what if a thread wants to move a list node from list l1 to list l2?**

Example: Lock Contention, Try3



- We could use one mutex for *each list node*.
  - See also Slide #22.
- Now threads can lock individual list nodes.
  - Access to different list nodes can happen simultaneously.
  - More parallelism than with Try2.
  - **Fine-grained lock.**
- For some operations, it might be necessary to lock more than one node!
  - Important to obey locking hierarchy (to prevent deadlock!)

Locking Granularity

- **Locking granularity** describes the amount of data that a lock protects.
- A **coarse-grained lock** protects a large amount of data.
  - Example: a lock protecting a whole subsystem, e.g., the whole filesystem-code in the Linux kernel, or the complete linked list ADT.
- A fine-grained lock protects only a small amount of data.
  - Example: one lock for each node of a linked list.
- Coarse-grained locks are likely to be highly contended.
- Locks usually start coarse-grained.
  - Code evolves to more fine-grained locking if lock contention becomes a problem.
- However, locking and unlocking add overhead to a program!

Outline

- Interleavings of Threads ✓
  - Race conditions ✓
- Lock-based Thread Synchronization
  - Mutexes ✓
  - **Semaphores** ← next
  - Condition variables
- Potential problems
  - Deadlocks ✓
    - Lock contention & lock granularity ✓
  - Livelocks
  - Starvation
- Examples

Semaphores

Semaphores are non-negative integer synchronization variables.

- Two basic operations on a semaphore **s**:
  - **P(s):** **[while (s == 0) wait();**

    s--;]

    Meaning: a thread has to wait until the value of s is greater than 0; then s is decremented by 1 and the thread is allowed to continue execution.
  - **V(s):** **[s++;]**

    Meaning: increment s.
- It is guaranteed that the statements between brackets [ ] are executed indivisibly.
- The statements between brackets [ ] are therefore an atomic operation.
  - At any time, only one P() or V() operation can modify s.
- Semaphore invariant: **(s >= 0)**

Semaphores (cont.)

```
#include <semaphore.h>

#define MAX 4
#define MAX_ITER 50000000
pthread_t thr[MAX];
sem_t s;

long counter = 0;

void * tfunc (void * arg) {
  int i;
  for (i=0; i<MAX_ITER; i++) {
    sem_wait(&s);
    counter++; //critical section
    sem_post(&s);
  }
}

int main() {
  int i, j;
  sem_init(&s, 0, 1);
  ...
}
```

Binary Semaphore:

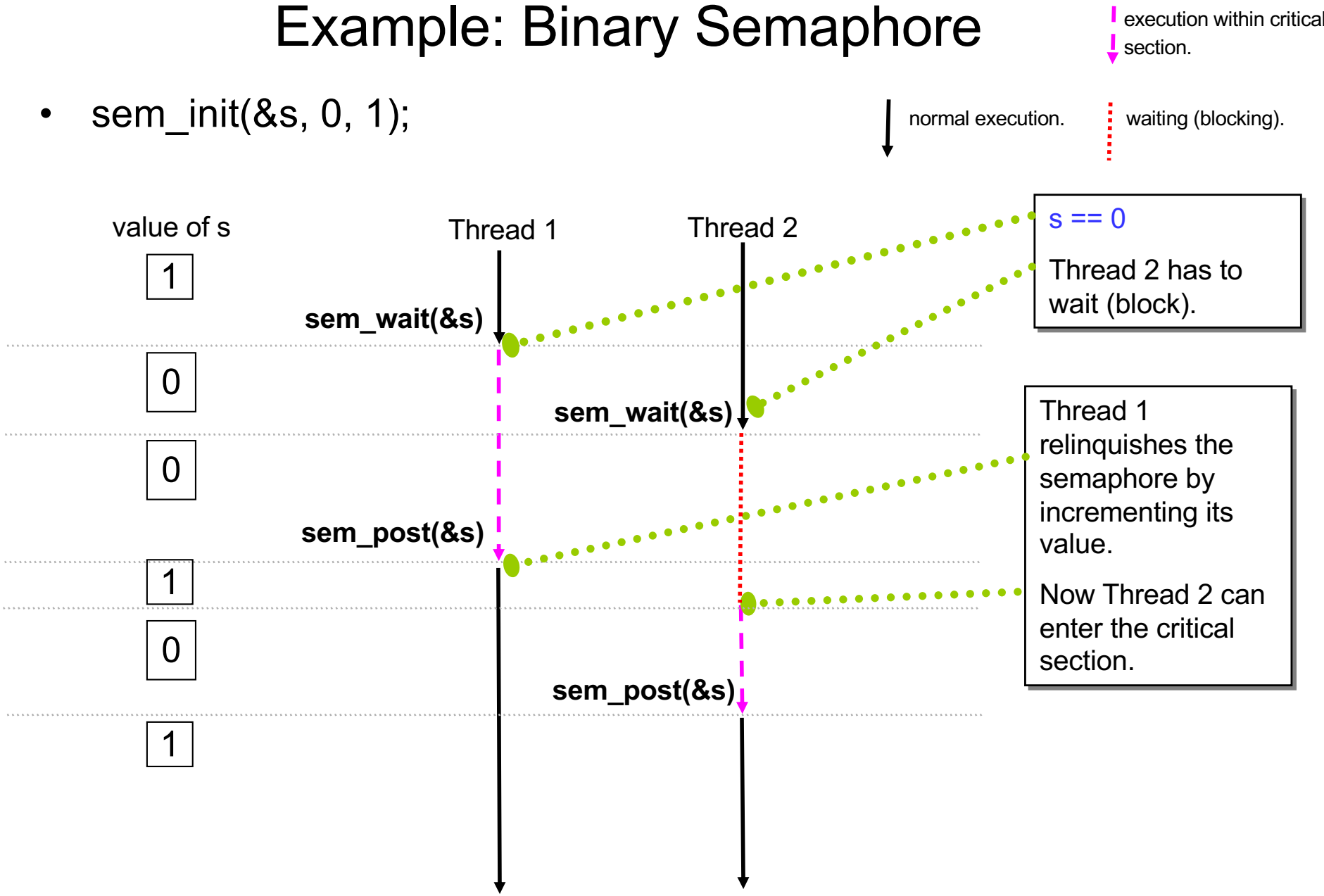
- Initialized to 1 initially.
- Achieves mutual exclusion between threads
  - Behaves like a mutex.

Counting Semaphore:

- Initialized to N > 1.
- Allows N threads at the same time in the critical section. This is a special case, e.g., for bounded buffers.
- See examples on next slides.

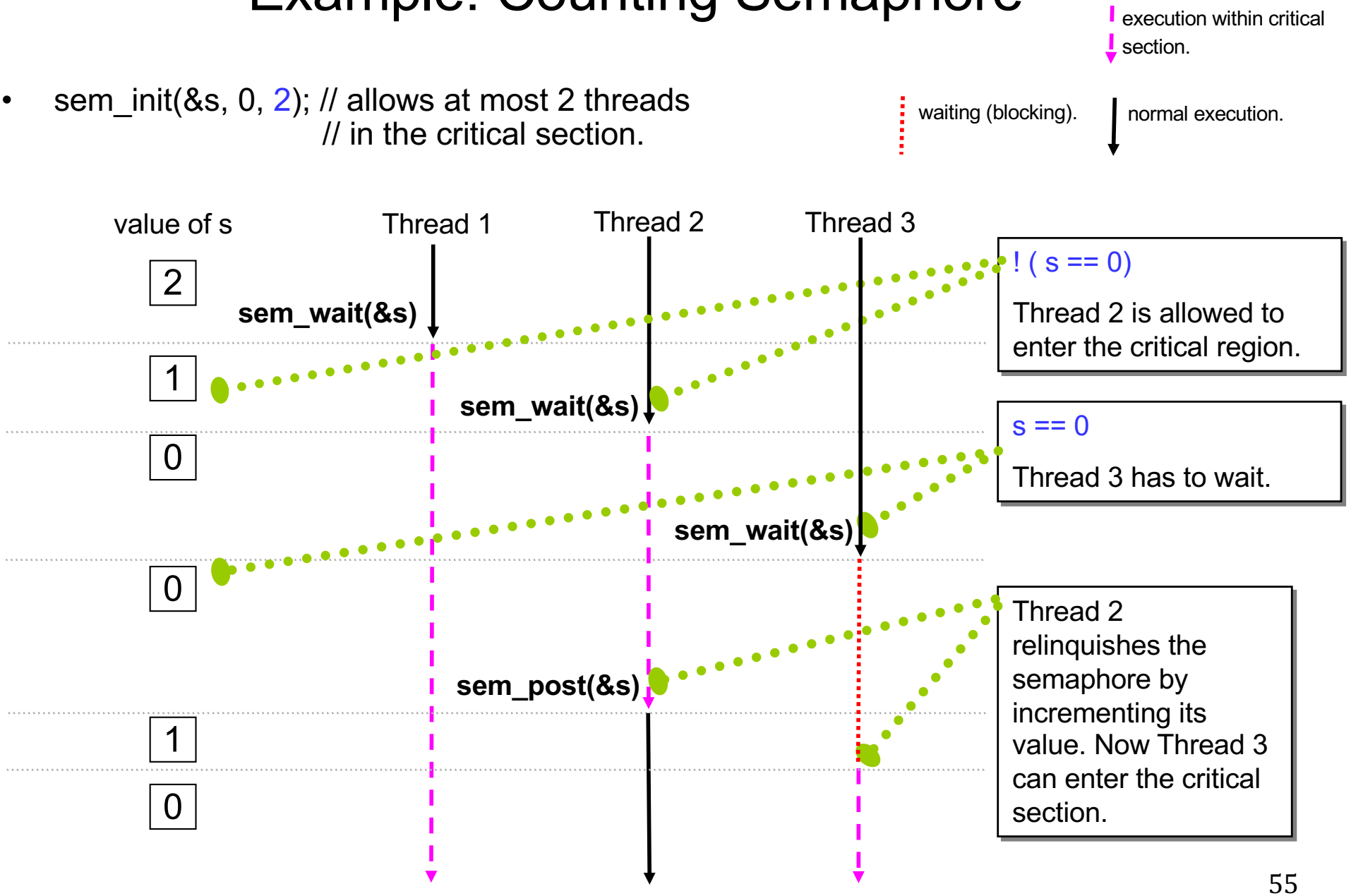
Example: Binary Semaphore

- sem_init(&s, 0, 1);



Example: Counting Semaphore

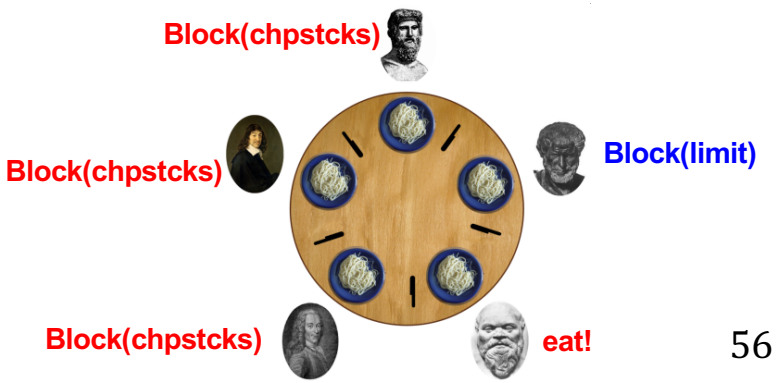
- sem_init(&s, 0, 2); // allows at most 2 threads
  // in the critical section.



Example: Dining Philosophers

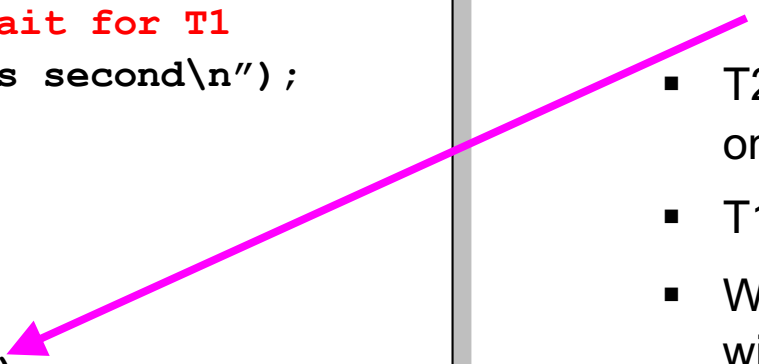
```
sem_t chpstcks[N], limit;

void * Philosopher(void * arg) {
  long id = (long) arg;
  for(;;) {
    think();
    sem_wait(&limit);
    sem_wait(&chpstcks[id]);
    sem_wait(&chpstcks[(id+1)%N];
    eat();
    sem_post(&chpstcks[id]);
    sem_post(&chpstcks[(id+1)%N]);
    sem_post(&limit);
  }
}

int main() {
  int i;
  for(i=0; i<N; i++)
    sem_init(&chpstcks[i], 0, 1);
  sem_init(&limit, 0, N-1);
  ...
}
```

- A counting semaphore can prevent the Dining Philosophers from dead-locking:
  - Assume N philosophers sitting at the table.
  - Use a counting semaphore with an initial count of N-1.
    - → At most N-1 philosophers can pick up the left chopstick at once.
    - → At least one of those philosophers will have access to two chopsticks. This philosopher can eat.



Synchronizing Threads using Semaphores

```
sem_t s;

void * T1(void * arg) {
  ...
  printf("this comes first\n");
  sem_post(&s);
  ...
}

void * T2(void * arg) {
  ...
  sem_wait(&s); //wait for T1
  printf("this comes second\n");

  ...
}

int main() {
  sem_init(&s, 0, 0);
  ...
}
```



- Besides mutual exclusion, semaphores can also be used to synchronize threads.
- Example:
  - Assume 2 threads executing the thread routines T1 and T2.
  - Assume a semaphore s.
    - s is initiallized to 0.
  - T2 has a sem_wait() operation on s.
  - T1 has a sem_post() operation.
  - When T2 reaches sem_wait(), it will block until T1 has executed sem_post().
  - Question: what happens if T1 executes sem_post() before T2 executes sem_wait() ?

Synchronizing Threads using Semaphores

```
1  sem_t ____________________;
2
3  void * T1(void * arg) {
4    for (;;) {
5      ________________________
6      printf ("ping\n");
7      ________________________
8    }
9  }
10
11 void * T2(void * arg) {
12   for (;;) {
13     ________________________
14     printf ("pong\n");
15     ________________________
17   }
16 }
17
18 int main() {
19   ______________________
20   ______________________
21   ...
22 }
```

- Example:
  - Assume 2 threads, executing the thread routines T1 and T2, respectively.
  - Thread T1 outputs "ping" in an endless loop.
  - Thread T2 outputs "pong" in an endless loop.
  - How can we synchronize T1 and T2 using semaphores, such that the output will be

    ping
    pong
    ping
    pong
    ...

?

Semaphore History

- Edsger Wybe Dijkstra (1920—2002) formalized semaphores as a generalized locking mechanism (back in 1968).
  - the Dining Philosopher's Problem is also due to E. Dijkstra.
- The original lock and unlock operations of semaphores where P() and V().
  - from the Dutch words *Passeren* (to pass) and *Vrijgeven* (to release).
  - Later on, those operations where renamed to sem_wait() and sem_post().
  - Some semaphore implementations use different names like down() and up().
- In 1972, Edsger W. Dijkstra received the Turing Award for his contributions to algorithm theory, operating system design and programming language formalization (ALGOL).
  - Famous words:
    - "*Goto Statement Considered Harmful*"
    - ***"Program testing can be used to show the presence of bugs, but never to show their absence!"***

Comparison of Semaphores and Mutexes

A mutex provides mutual exclusion similar to a binary semaphore (a semaphore initialized to 1).

- Allows only one thread at a time to execute a critical section.

Mutexes are less flexible than semaphores:

- A mutex that is locked has an owner, a semaphore doesn't!
  - The thread that acquires a mutex must also release it!
- A mutex is initially unlocked.
  - Semaphores can be initialized to any value, **including zero**.
  - A semaphore initialized to zero corresponds to an initial "locked" state.
- A mutex can only assume two states, "unlocked" and "locked".
  - Can be represented by a single bit.
- A semaphore is a counter.
  - A thread executing sem_wait() will wait until the counter value is > 0 and then decrease the counter value by one. A thread executing sem_post() will increase the counter value by one. Counter values are in the range 0 .. N.

Outline

- Interleavings of Threads ✓
  - Race conditions ✓
- Lock-based Thread Synchronization ✓
  - Mutexes ✓
  - Semaphores ✓
- Potential problems
  - Deadlocks ✓
    - Lock contention & lock granularity ✓
  - **Livelocks** ← next
  - **Starvation** ← next
- Amdahl's Law

Monitors vs. (Semaphores and Mutexes)

Main problem with mutexes and semaphores is that their use is **voluntarily**.

- A programmer who forgets to use the semaphore/mutex associated with a shared data item will introduce a race condition.
  - Limited tool support to check such a programming error.
- A programmer who violates a locking hierarchy will introduce a potential deadlock.
  - Limited tool support to check such a programming error at compile-time.
  - Some operating systems can detect a dead-locked program at run-time and "somehow" resolve the deadlock.

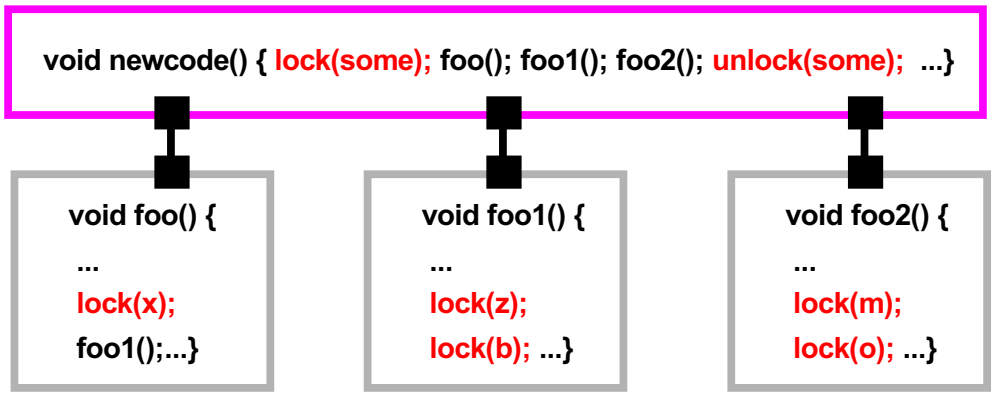
Monitor = shared data + mutual exclusion/synchronization mechanism

- Monitors are safer than mutexes and semaphores.
  - Locking responsibility is moved from the user of the shared data item to the implementer of the monitor component.
- More flexible: the exact monitor functionality is up to its implementer.
  - However, the "assumed" behavior of a monitor is that only one thread can access the monitor's shared data at any time (mutual exclusion).
- Monitors are perhaps slightly more inefficient due to additional procedure calls to the monitor component (when compared to direct access of shared data).

Critique of Lock-based Synchronization

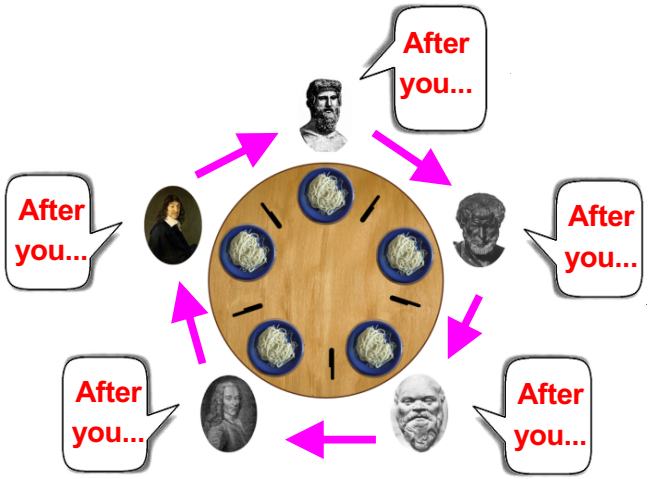
“Locks do not compose” !

- We build large sw-systems from parts.
- While holding locks, it is problematic to make calls to “unknown” code, because the called code might acquire locks as well.
  - Could violate the locking hierarchy and result in deadlock.
- Consequence: we cannot compose sw from parts unless we know the exact locking behavior of the parts.
  - This is meant by “locks do not compose”.
  - Tedious for large code bases! (Code calls code which calls other code aso!).



Potential problems with thread synchronization

- Deadlocks
- Livelocks
  - Two or more threads are busy synchronizing and do not make progress in what they actually want to compute:



- Starvation
  - One thread is never allowed into the critical section.
  - Need some fairness-property to prevent starvation and ensure that every thread is able to make progress.

Outline

- Interleavings of Threads ✓
  - Race conditions ✓
- Lock-based Thread Synchronization ✓
  - Mutexes ✓
  - Semaphores ✓
- Potential problems
  - Deadlocks ✓
    - Lock contention & lock granularity ✓
  - **Livelocks** ✓
  - **Starvation** ✓
- Amdahl's Law

Limits to Performance Scalability

- Not all programs are “embarrassingly” parallel.
- Programs have sequential parts and parallel parts.

Sequential part: cannot be parallelized because of data *dependencies*.

Parallel part: no data dependence, the different loop iterations (Line 5) can be executed in parallel.

```
1   a = b + c;
2   d = a + 1;
3   e = d + a;
4   for (k = 0; k < e; k++)
5      M[k] = 1;
```

Limits to Performance Scalability

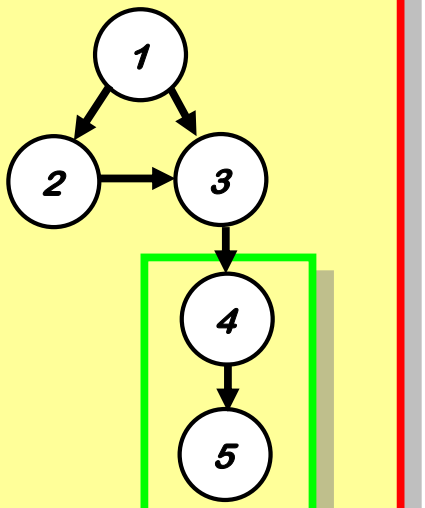
- Not all programs are “embarrassingly” parallel.
- Programs have sequential parts and parallel parts.

Sequential part: cannot be parallelized because of data *dependencies*.

Parallel part: no data dependence, the different loop iterations (Line 5) can be executed in parallel.

```
1   a = b + c;
2   d = a + 1;
3   e = d + a;
4   for (k = 0; k < e; k++)
5      M[k] = 1;
```

Dependencies:



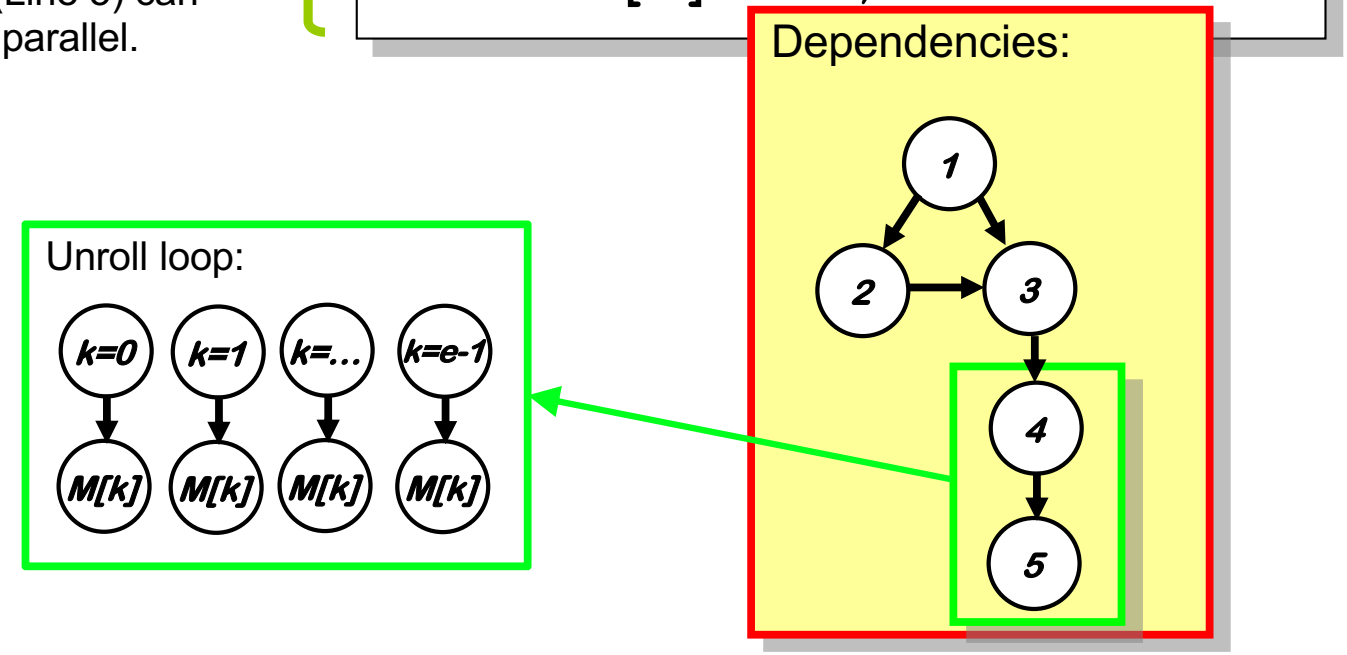
Limits to Performance Scalability

- Not all programs are “embarrassingly” parallel.
- Programs have sequential parts and parallel parts.

Sequential part: cannot be parallelized because of data *dependencies*.

Parallel part: no data dependence, the different loop iterations (Line 5) can be executed in parallel.

```
1   a = b + c;
2   d = a + 1;
3   e = d + a;
4   for (k = 0; k < e; k++)
5       M[k] = 1;
```

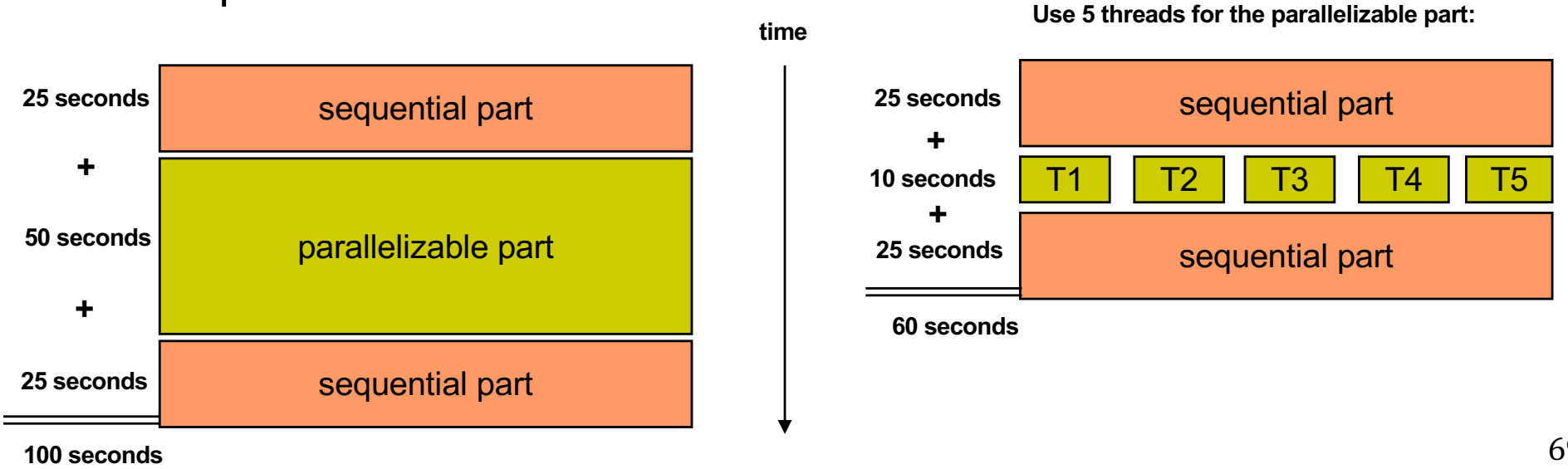


Amdahl's Law

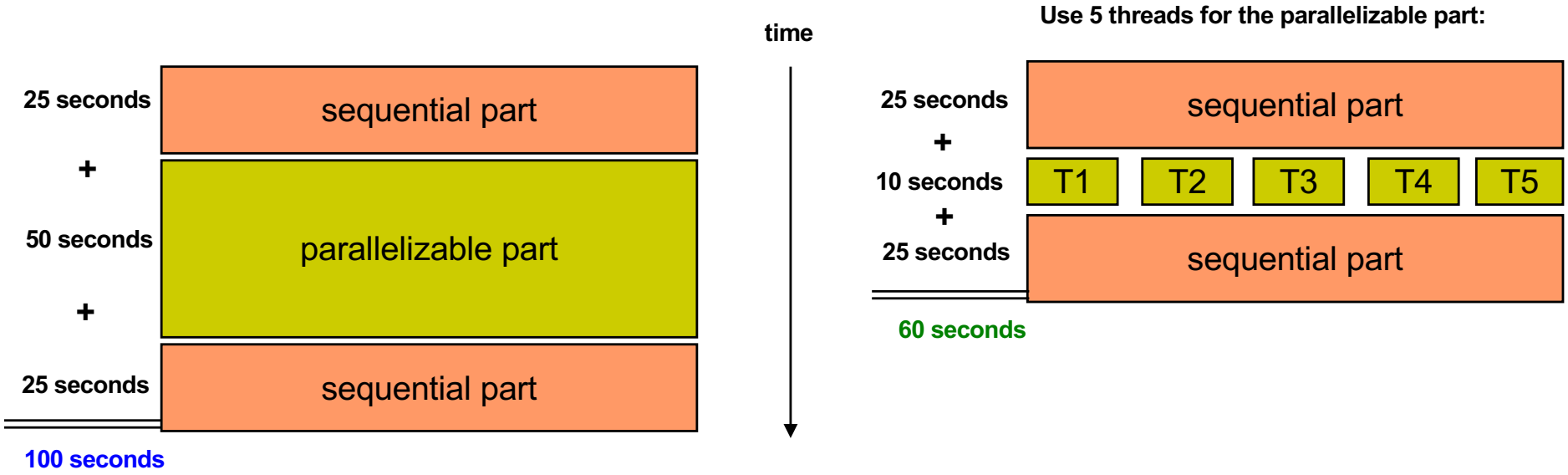
In 1967, Gene Amdahl, then IBM computer mainframe architect, stated that

> *"The performance improvement to be gained from some faster mode of execution is limited by the fraction of the time that the faster mode can be used."*

- *"Faster mode of execution"* here means program parallelization.
- The potential speedup is defined by the fraction of the code that can be parallelized.

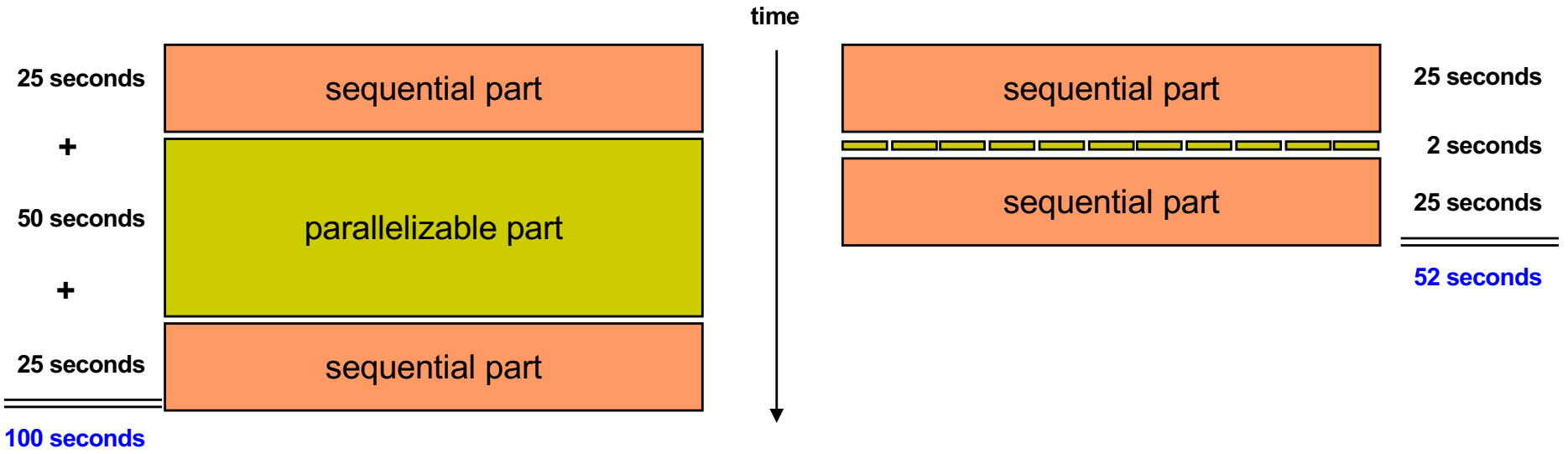


Amdahl's Law (cont.)



- Speedup = old running time / new running time
  = 100 seconds / 60 seconds
  = 1.67
- The Parallel version is 1.67 times faster than the sequential version.

Amdahl's Law (cont.)



- We may use more threads executing in parallel for the **parallelizable part**, but the **sequential part** will remain the same.
  - **The sequential part of a program limits the speedup that we can achieve!**
- Even if we *theoretically* could reduce the parallelizable part to 0 seconds, the best possible speedup in this example would be

  speedup = 100 seconds / 50 seconds = 2.

Amdahl's Law (cont.)

- *p* = fraction of work that can be parallelized.
- *n* = the number of threads executing in parallel.

$$new\_running\_time = (1\text{-}p) * old\_running\_time + \frac{p * old\_running\_time}{n}$$

$$Speedup = \frac{old\_running\_time}{new\_running\_time} = \frac{1}{(1 - p) + \frac{p}{n}}$$

- Observation: if the number of threads goes to infinity ($n \to \infty$), the speedup becomes $\frac{1}{1-p}$.

- Parallel programming pays off for programs which have a **large parallelizable part**.

Amdahl's Law (Examples)

- p = fraction of work that can be parallelized.
- n = the number of threads executing in parallel.

$$Speedup = \frac{old\_running\_time}{new\_running\_time} = \frac{1}{(1 - p) + \frac{p}{n}}$$

- Example 1: p=0, an embarrassingly sequential program.

$$\text{speedup} = \frac{1}{1 + \frac{0}{n}} = 1 \text{ (no speedup!)}$$

- Example 2: p=1, an embarrassingly parallel program.

$$\text{speedup} = \frac{1}{0 + \frac{1}{n}} = n \text{ (the number of processors gives the speedup!)}$$

- Example 3: p=0.75, n = 8

$$\text{speedup} = \frac{1}{0.25 + \frac{0.75}{10}} = 2.91$$

- Example 4: p=0.75, n = ∞

$$\text{speedup} = \frac{1}{0.25 + \frac{0.75}{\infty}} = 4 \text{ (theoretical upper bound if 25\% of the code cannot be parallelized)}$$