User-level synchronisation primitives

Curtis Millar – 8th November 2023

User-level synchronisation primitives

- Fundamentals of scheduling primitives
- Break
- Fast, efficient, and correct scheduling primitives
- Q & A

Fundamentals of scheduling primitives

Curtis Millar – 8th November 2023

Fundamentals of scheduling primitives

1. Review multitasking
2. A model of concurrency
3. The basic structure of a synchronisation primitive
4. Common synchronisation primitives & their use

Multitasking

Multitasking

- What is multitasking?
- Multiple threads of execution
- Shared execution resources (CPUs, threads of a lower-level scheduler)
- Policy for when to switch between threads
- Policy for which thread to switch to

Co-operative multitasking

- Without a central scheduler
  - Threads select another thread to run instead of themselves
  - Threads decide both *when* to switch and *which thread* to switch to
- With a central scheduler
  - Threads yield time and the scheduler selects a thread to execute
  - Threads decide *when* to switch but the scheduler decides *which* thread execute next
  - A thread can still inform the scheduler which thread it wants to execute next

Preemptive multitasking

- The scheduler programs a hardware timer to interrupt execution before running a thread
- The thread’s execution is interrupted when the specified time is reached and the scheduler can change which thread executes
- The scheduler decides both *when* to switch threads and *which* thread executes
- The running thread can still voluntarily allow a switch and even nominate a preferred thread to execute next

A model of multitasking

- One or more execution resources, e.g., a CPU
- A set of *threads* contending for *time* to exclusively use the execution resource(s)
- A scheduler with
  - The scheduling state of each thread in the system
  - The relationships between threads in the system
  - A policy for selecting a thread to which it will allocate an execution resource at any given point in time

Concurrency

Concurrent applications

- Multiple threads of execution
- Shared resources, e.g., data structures, threads, files, streams, devices
- Threads can change resources between *consistent states*
    - If a resource isn't in a consistent state, e.g., because a different thread is part-way through operating on it, it must *wait* until it is
- Threads may depend on resources being a particular state (or set of states)
    - If a resource isn't in the required state, the thread must *wait* until it is

An example concurrent application

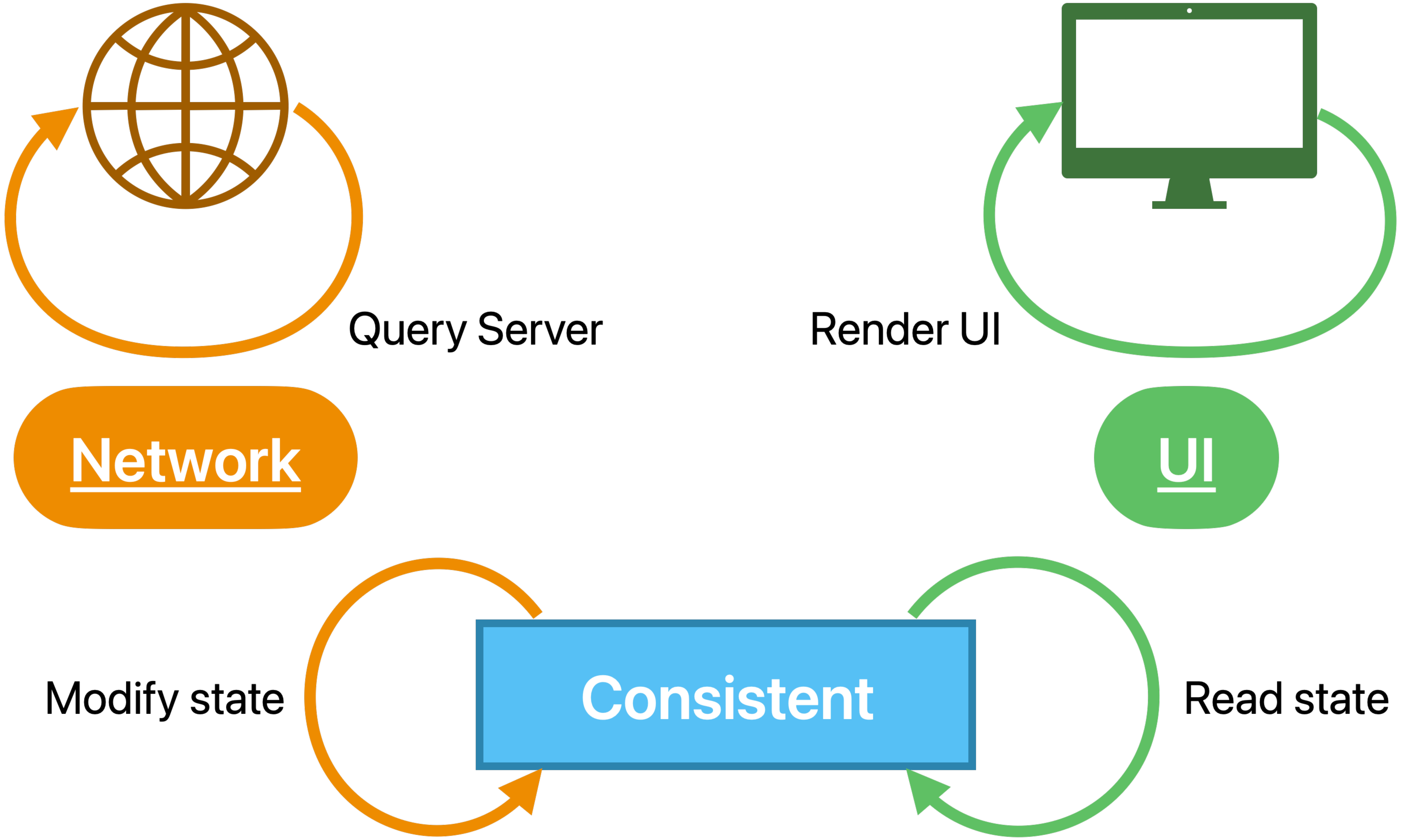
- Video streaming service on networked computer
  - Multiple processing cores with parallel execution across threads
  - Multiple threads dealing with hardware
  - Threads for client sessions and communication with remote servers
- Application on local machine
  - Dedicated UI thread
  - Thread for communicating with server
  - Thread for GPU and hardware video decode

Primitives of concurrency

- A thread must
  - determine the state of the resource, and
  - either wait for that resource to change to a required state or
  - start performing its operation on that resource
- Resources must change state atomically,
  - A thread must not observe another thread's partial operations on a resource
- A thread that is *waiting* can't usefully execute, so should not be scheduled
  - The scheduler needs to know whether each thread is waiting

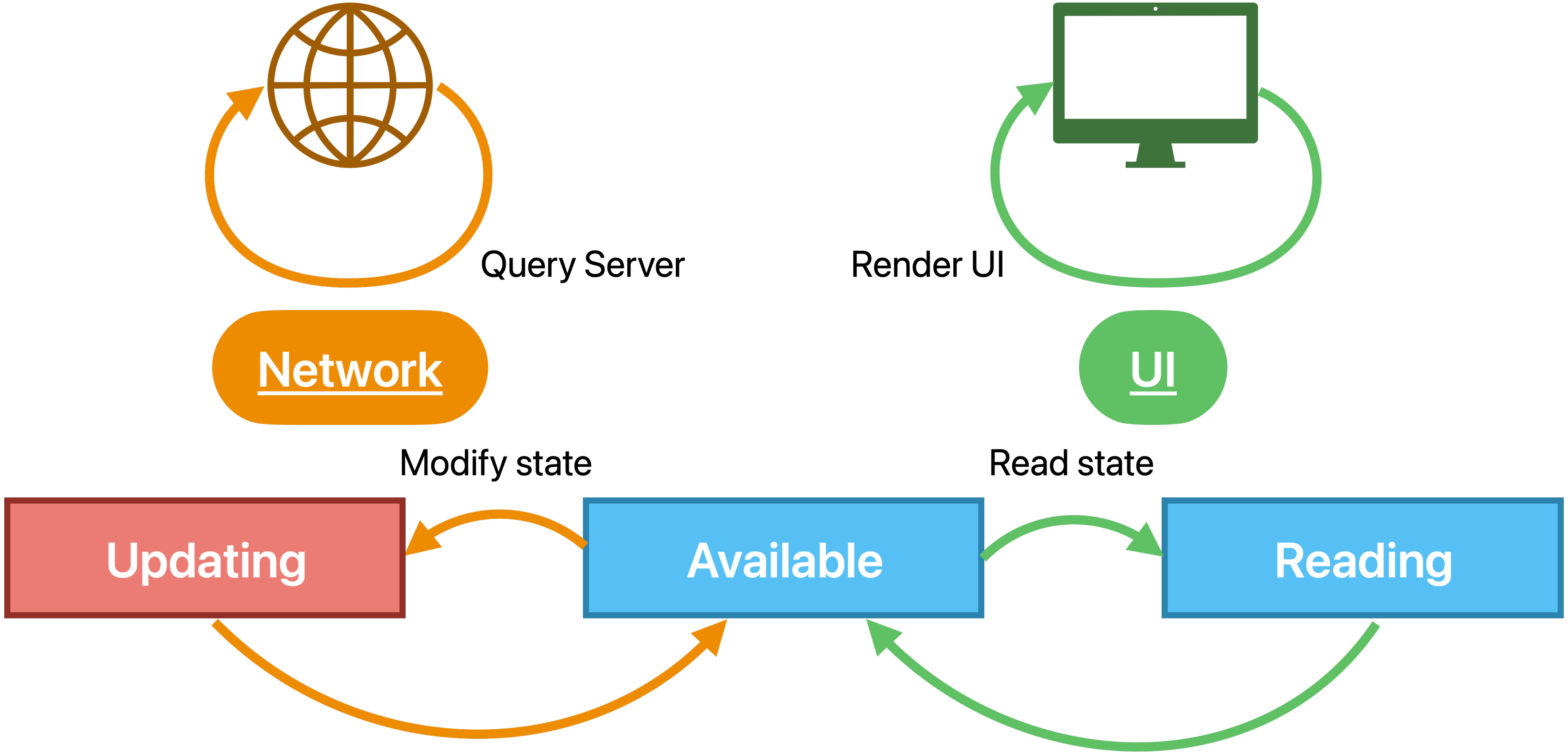
Concurrent operations on a shared resource

Atomic operations between consistent states



Concurrent operations on a shared resource

Atomic operations between consistent and inconsistent states



Synchronisation objects

Structure of a synchronisation primitive

- Object in memory with an encoding of the state of some resource
- Thread enters the *critical section* of the primitive
  - Only one thread may execute in the critical section of the primitive at a time
- During the critical section, the thread may
  - Read and modify the synchronisation object
  - Block itself *on* the synchronisation primitive
  - Unblock other threads that are blocked on the primitive
- Exits the critical section of the primitive

Waiting and waking

- Multiple threads may be blocked on a given primitive at any time
- Any time the state of a primitive changes, one or more of the blocked threads may no longer need to wait
- When a thread changes the state of a primitive, it must unblock the threads that no longer need to wait
- The synchronisation object records the set of threads blocked on the primitive

Overall structure

- The resource managed by the primitive
- The synchronisation object, with
  - an encoding of the state of the resource, and
  - a record of the threads blocked on the primitive
- A critical section to
  - observe and change the encoded state,
  - (potentially) block the running thread to wait for a change in the encoded state, and
  - (potentially) unblock threads that were waiting for a change in the encoded state

Common primitives

Mutual exclusion

- Permit only one thread to access a resource at a time
- When a thread *acquires* the mutex
  - No other threads are accessing the mutex
  - It can make arbitrary modifications
- When the thread *releases* the acquired mutex
  - The resource must be in a consistent state
  - One of the threads waiting can exclusively acquire
- Threads form a queue where each in turn gets exclusive access

Mutual exclusion

C11 mutex

```c
/* Add the given series to the list the user wants to watch */
void add_series_to_watchlist(user_t *user, tv_series_t *series) {
 /* Acquire access to the user's data exclusively */
 mtx_lock(&user->mutex);

 /* Insert the series into the watchlist */
 list_append(user->watchlist, series);

 /* Release exclusive access to the user's data */
 mtx_unlock(&user->mutex);
}
```

Initialisation

- Ensures an operation only happens once
- A resource may not be in a consistent state when an application starts (can't be done statically or too expensive to do up-front)
- Resource must be initialised before any thread operates on it
- Before operating on the resource, each thread
  - checks if it has been initialised; if it hasn't
    - if its the first to check, it initialises it, otherwise
    - it waits until its initialised
- Threads wait in a group and are woken at the same time

Initialisation

C11 once flag

```
once_flag remote_connected = ONCE_FLAG_INIT;
connection_t *remote_server;

void connect_to_remote(void) {
  /* Set up the connection to the server */
  remote_server = create_connection(REMOTE_ADDRESS);
}

local_video_t *copy_from_remote(remote_video_t *video) {
  /* Ensure there is a connection to the remote server */
  call_once(&remote_connected, connect_to_remote);

  return create_video(copy_from_remote(remote_server, video->path));
}
```

Condition variables

- Used in combination with a mutex
- Allows for distinct sets of threads waiting for a resource
- Thread with an acquired mutex can check for a *condition* to be true for the resource and, if it isn't, wait with a specific group of threads
- When another thread changes the resource such that the condition (may) be true, wakes one or all of the threads from the specific group
- Threads woken from the condition variable re-acquire the mutex they held

Condition variables

Server handling storage requests

```
void storage_server(storage_t *storage) {
  mtx_lock(&storage->lock);
  while (storage->connected) {
    /* Wait until there's a new request or event */
    while (!(storage->new_request || storage->filesystem_event)) {
      cnd_wait(&storage->server, &storage->lock);
    }

    if (storage->new_request) handle_request(storage);
    if (storage->filesystem_event) handle_event(storage);

    /* Clear the conditions before checking again */
    storage->new_requests = storage->filesystem_event = false;
  };
  mtx_unlock(&storage->lock);
}
```

Condition variables

Client requesting a read from storage

```
media_t *read_from_storage(storage_t *storage, media_ref_t *ref) {
  mtx_lock(&storage->lock);

  request_t *request = create_read_request(storage, ref);

  /* Wake the storage controller to fetch the media */
  storage->new_request = true;
  cnd_signal(&storage->server);

  /* Wait until the request has been handled */
  while (!request->complete) {
    cnd_wait(&request->waiter, &storage->lock);
  }

  mtx_unlock(&storage->lock);

  return request->media;
}
```

Reader/writer locks

- Permit any number of threads to access the resource without any modifying it
- Permits exclusive access to the resource and the ability to modify it
- Readers can acquire if there isn’t a writer holding the lock
- Writers can acquire if there are no threads holding the lock
- How can you guarantee progress for writers?
    - Should readers be able to acquire the lock if there is a writer waiting?

Reader/writer lock

POSIX reader/writer locks

```
void series_add_episode(series_t *series, episode_t *episode) {
  /* Get exclusive access to the series to modify */
  pthread_rwlock_wrlock(&series->rwlock);

  list_append(series->episodes, episode);

  /* Release exclusive access */
  pthread_rwlock_unlock(&series->rwlock);
}

epsiode_t *series_latest_episode(series_t *series) {
  /* Get shared read-only access */
  pthread_rwlock_rdlock(&series->rwlock);

  episode_t *episode = list_tail(series->episodes);

  /* Release acces */
  pthread_rwlock_unlock(&series->rwlock);
}
```

Thread join

- Prevent a thread from executing until some other thread has completed
- Allows one thread to spawn several other threads to perform work in parallel and wait until all threads have completed work

Thread join

C11 threads

```
/* Encode a chunk of video */
int encode_chunk(chunk_t *chunk);

video_t *parallel_encode(video_t *video, encode_settings_t *settings) {
  /* Create a chunk and per CPU */
  size_t cpus = cpu_count();
  chunk_t *chunks = split_video(video, cpus);

  /* Create a thread per chunk */
  thrd_t threads[cpus];
  for (size_t c = 0; c < cpus; c++) thrd_create(&threads[c], encode_chunk, &chunks[c]);

  /* Wait for all threads to finish */
  int result;
  for (size_t c = 0; c < cpus; c++) thrd_join(&threads[c], &result);

  return join_chunks(chunks);
}
```

Summary

Fundamentals of scheduling primitives

1. Review multitasking
2. A model of concurrency
3. The basic structure of a synchronisation primitive
4. Common synchronisation primitives & their use

Break

10 mins

Fast, efficient, and correct scheduling primitives

Curtis Millar – 8th November 2023

Fast, efficient, and correct scheduling primitives

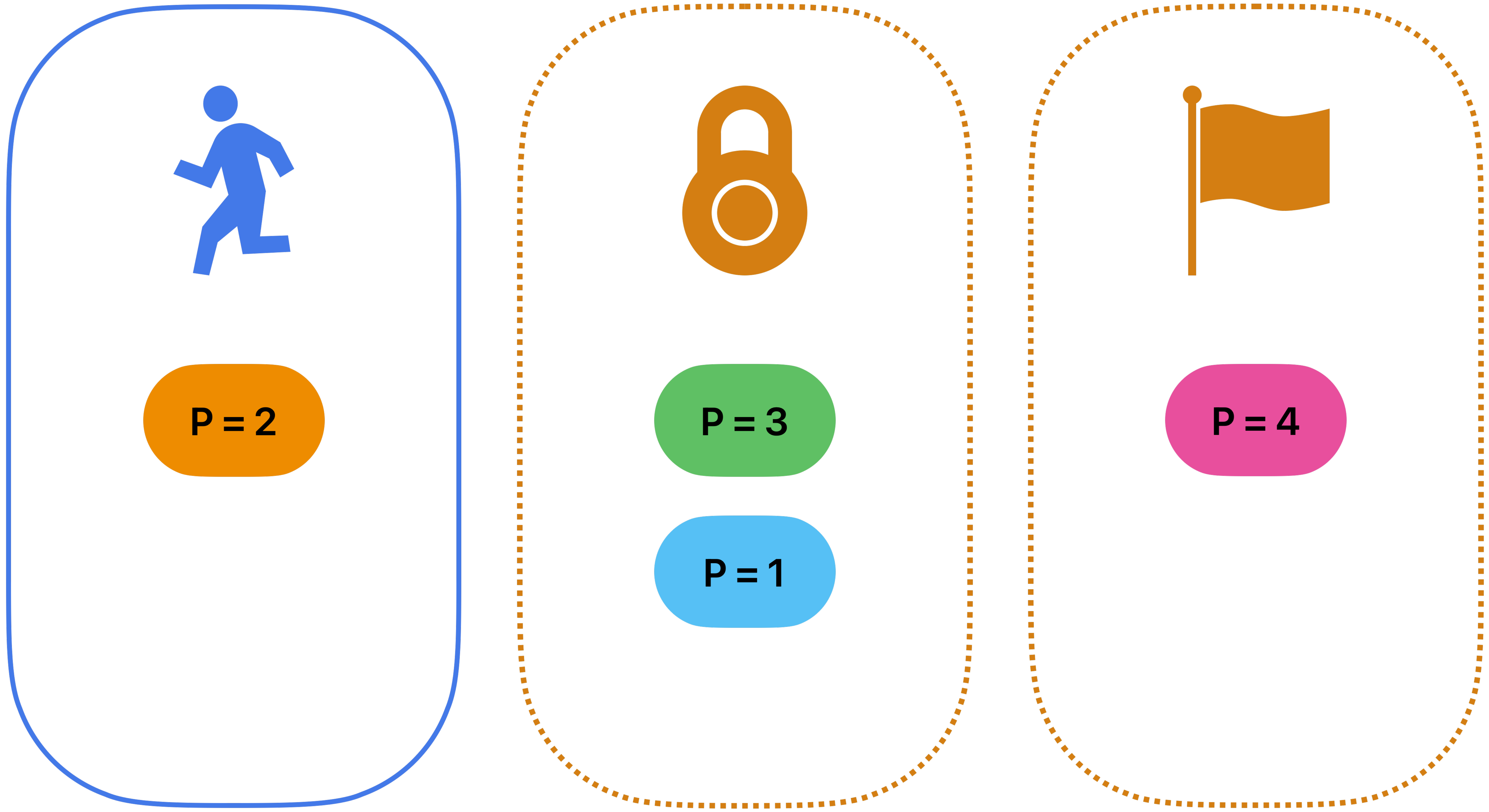
1. How synchronisation primitives inform scheduling policy
2. Sharing a scheduler between applications
3. Efficiently managing the allocations of synchronisation objects
4. Avoiding context switches to a shared scheduler

Scheduling policy

Scheduling state of synchronisation primitives

- Operations on synchronisation primitives change scheduling state
- Affect whether threads are blocked waiting or are runnable
- Determine how many of the woken threads can actually make progress
  - Only one thread waiting for a mutex can actually acquire it
- The scheduler uses *priority* to choose between runnable threads
- The scheduler should use the same priority to choose between *wakeable* threads

Scheduling state of synchronisation primitives

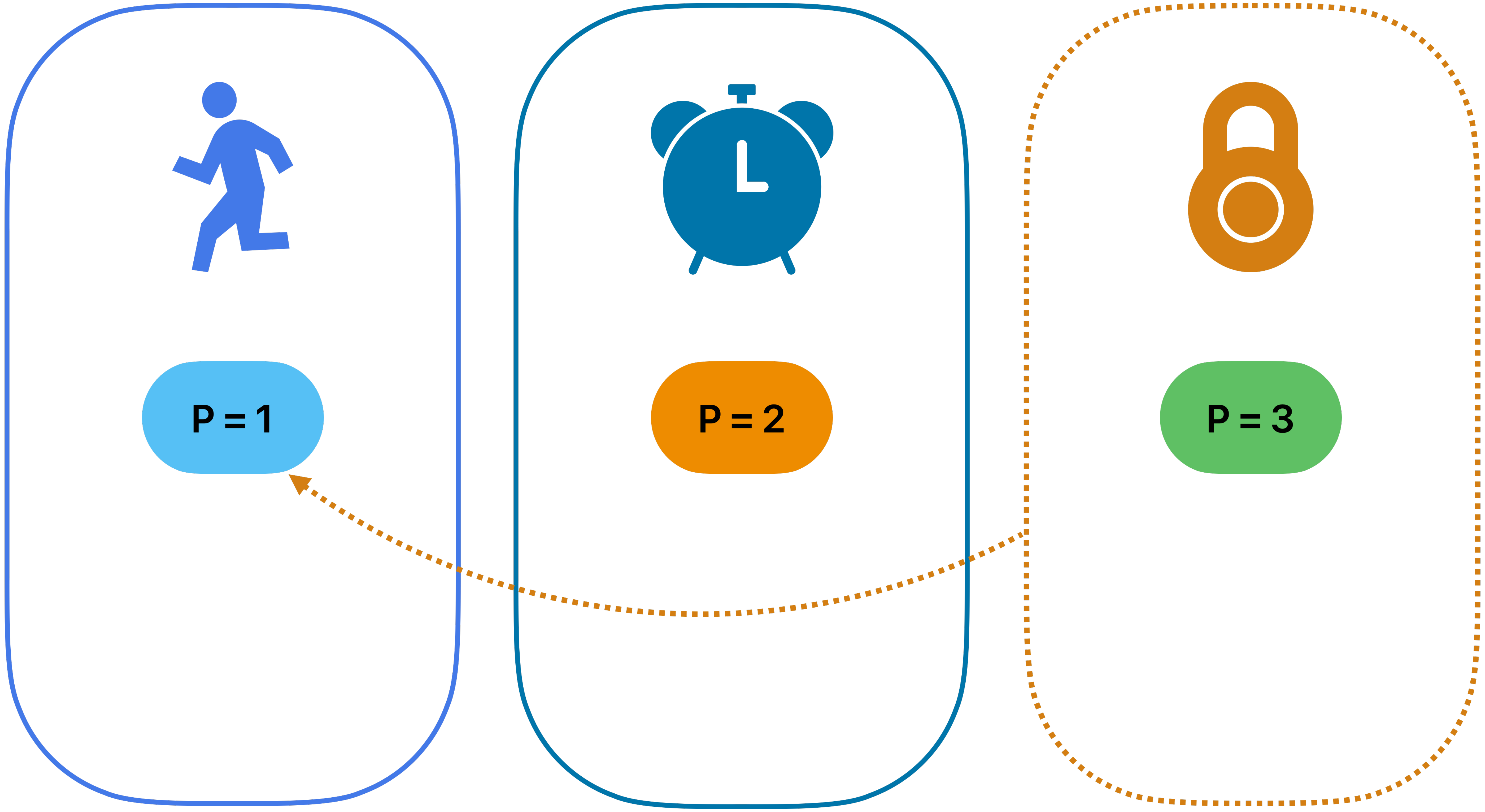


Priority inversion

- A thread is blocked waiting on a synchronisation primitive
- The thread makes progress when threads that are able to change the state of the primitive execute
- The thread(s) that will change the state to end the wait runs with a lower priority than the configured priority of the blocked thread
- The blocked thread has its priority *inverted*, in that it is now progressing at a relatively lower priority than configured
- Scheduling policy needs to mitigate priority inversion

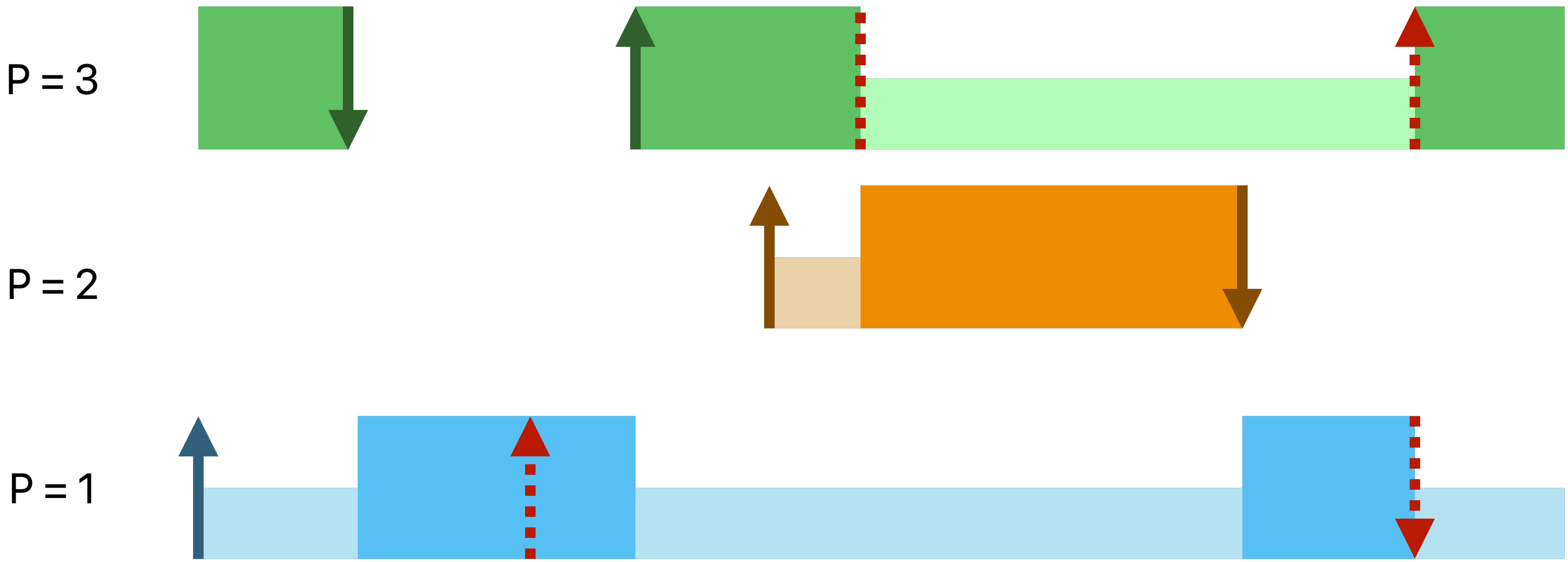
Priority inversion

Scheduling primitive state



Priority inversion

Scheduling timeline

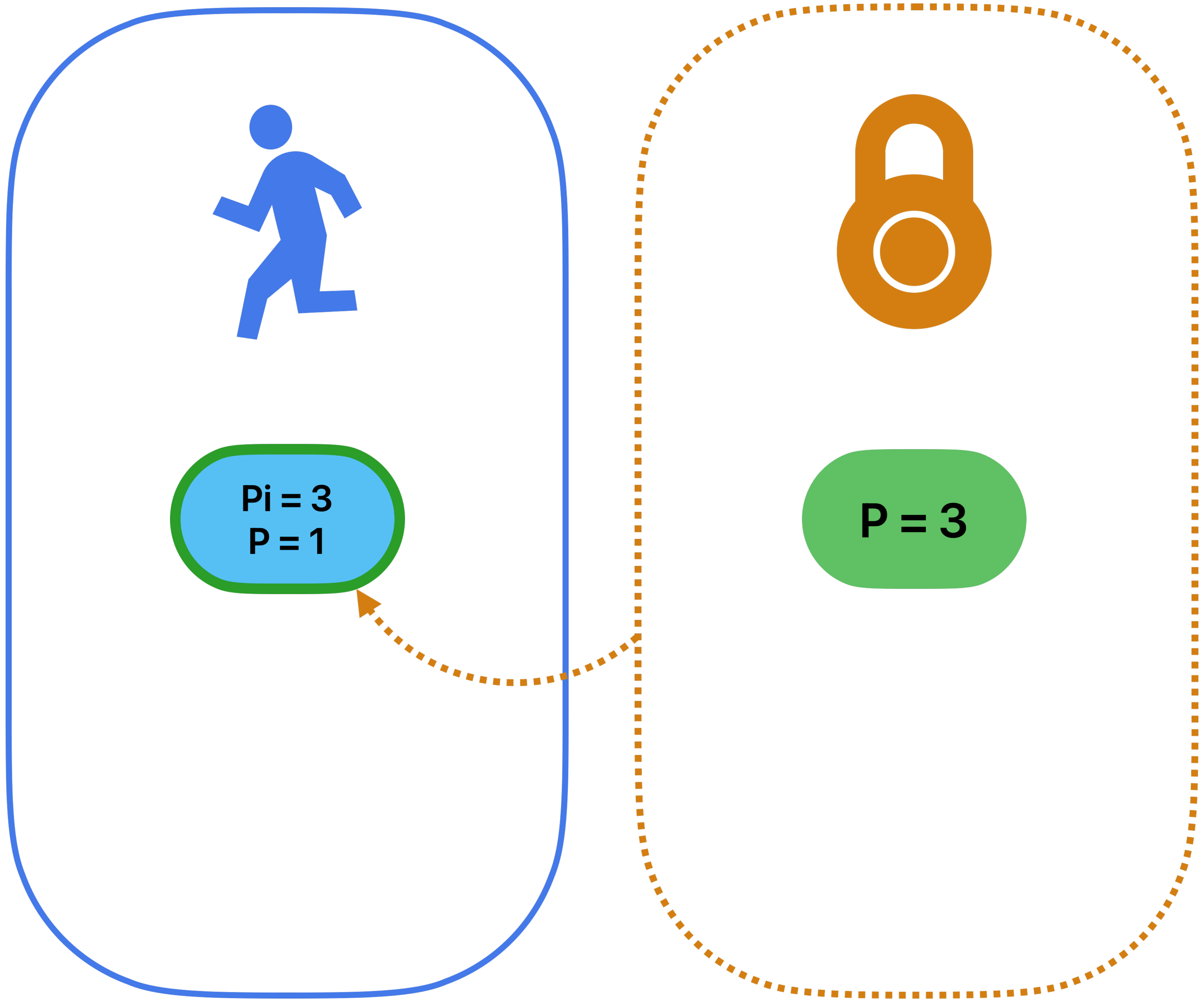


Priority inheritance

- The set of threads and resource contention relationships between them is complex and dynamic
- Threads that are waiting can donate their priority to a runnable thread that can progress towards ending the wait
- Very dependent on the model of priority used by the scheduler
- Scheduler must know the which runnable threads can unblock waiting threads
- Scheduler tracks the set of synchronisation primitives, the their waiting threads, and the threads in control of the resources

Priority inheritance

Scheduling primitive state



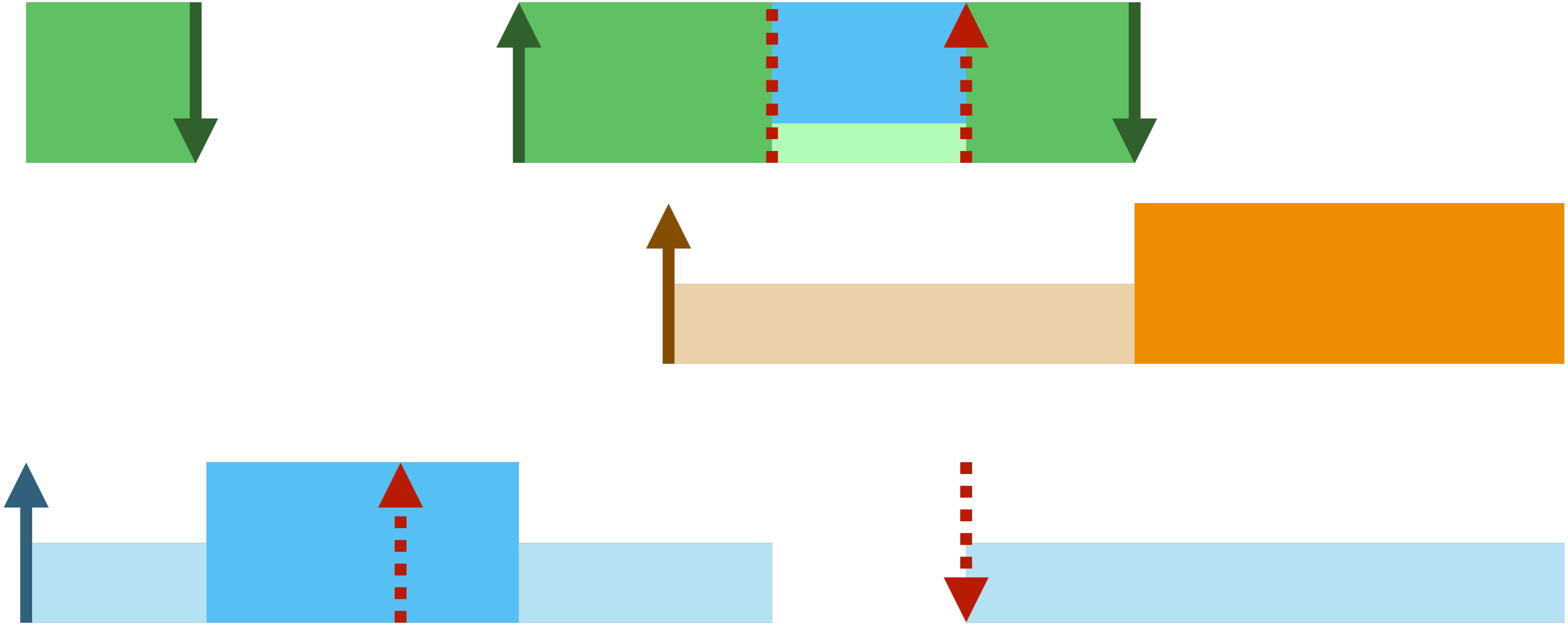
Priority inheritance

Scheduling timeline

P = 3

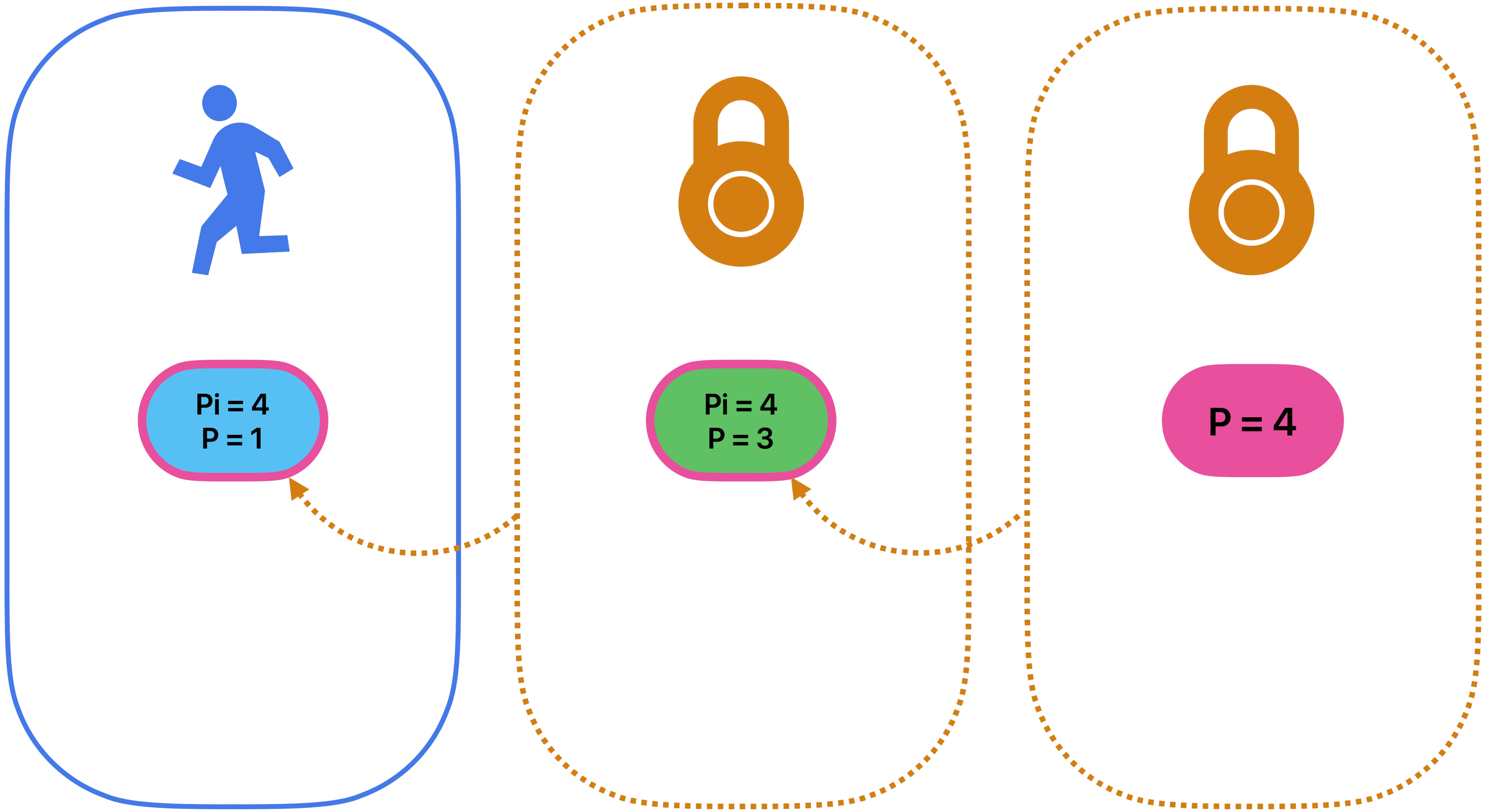
P = 2

P = 1



Priority inheritance

Transitive inheritance

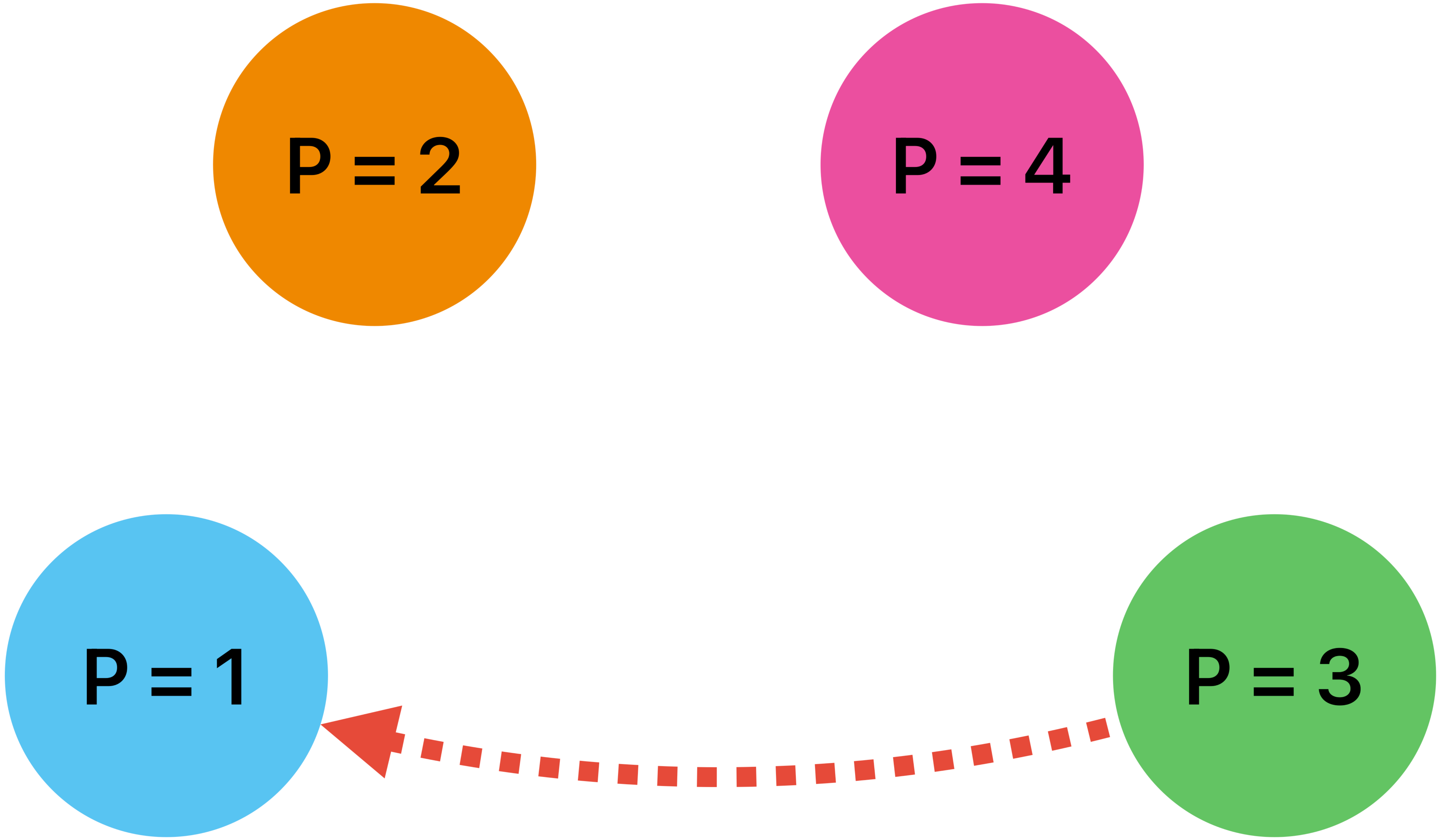


Blocking thread graph

- Blocked threads depend on the threads that unblock them
  - Track the threads blocked on a primitive and the threads that can change the state of the primitive
- Those threads may themselves be blocked
- Priority inheritance is transitive
- Progress is guaranteed if *some* runnable thread inherits the priority
- A cycle in the graph indicates deadlock

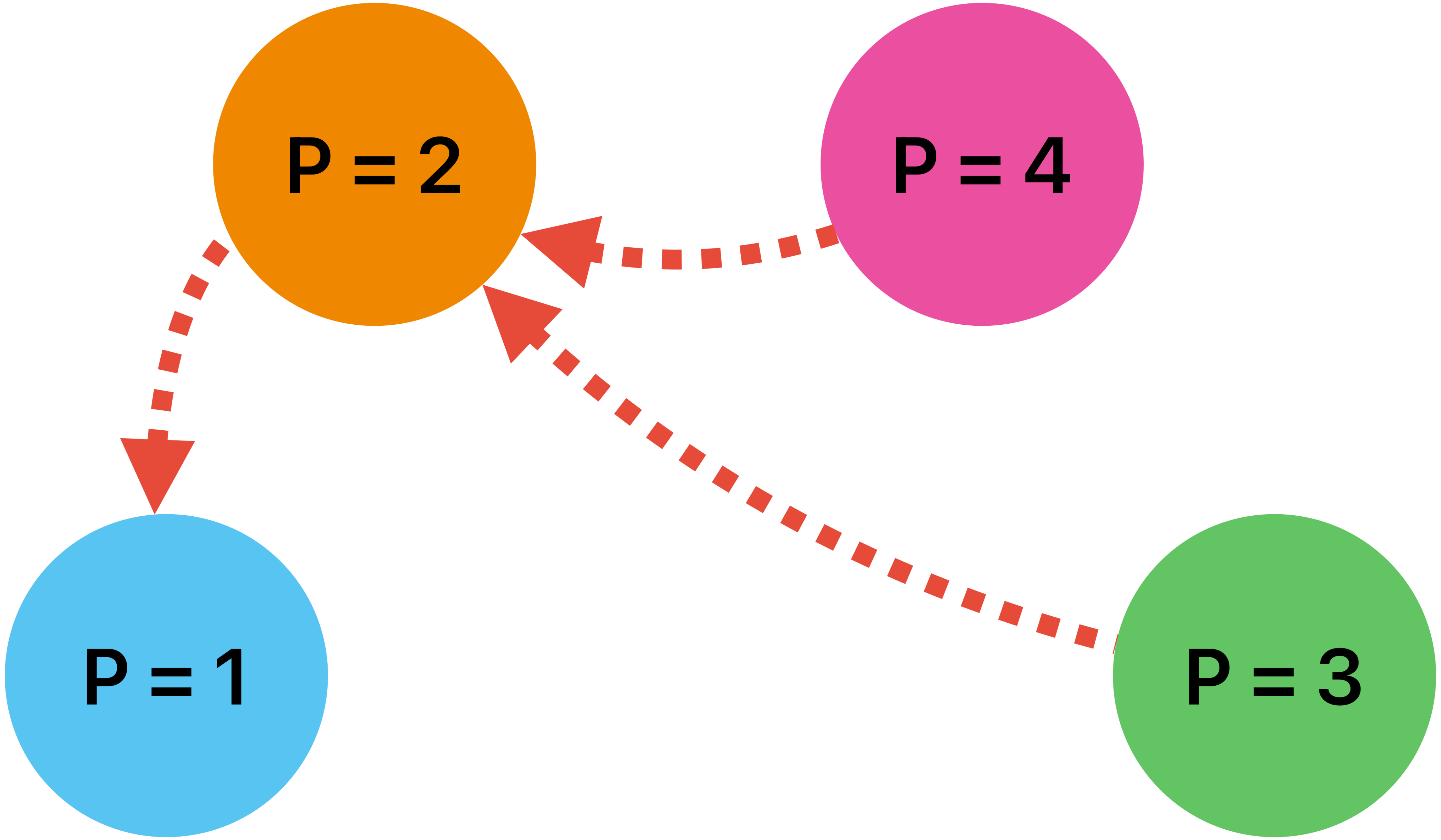
Blocking thread graph

Waiting on a held mutex



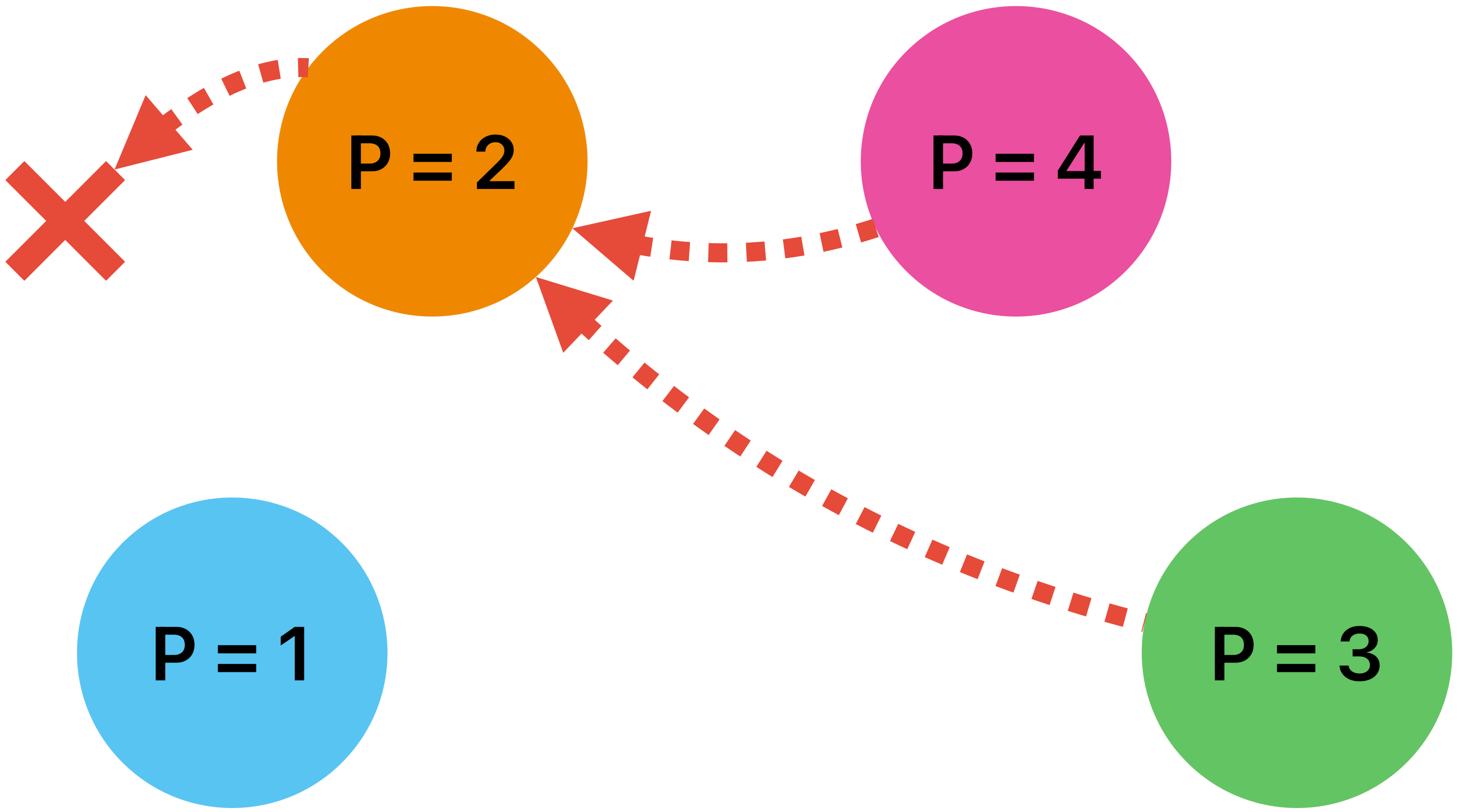
Blocking thread graph

Transitive waiting



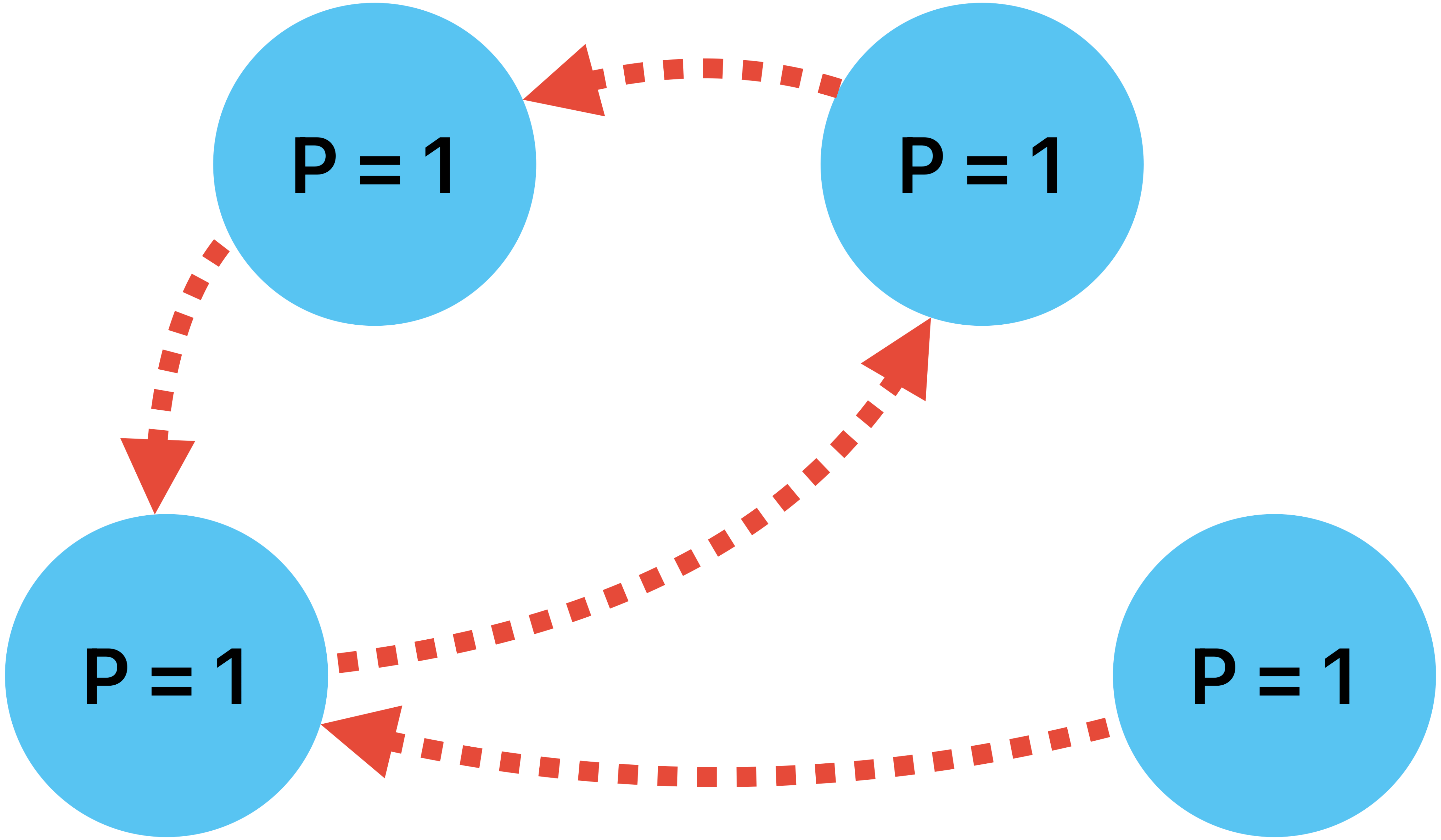
Blocking thread graph

Waiting without an inheritor



Blocking thread graph

Deadlock



Sharing a scheduler

Sharing a scheduler between applications

- Each address space has threads sharing resources managed with synchronisation primitives
- CPUs are shared across multiple applications
- Scheduler for these CPUs exists outside of these applications
- Modifying scheduling state must be done in the scheduler
- Scheduler can easily ensure critical sections execute atomically

Implementation in a shared scheduler

In each address space:

- execution of threads
- shared resources
- reference to sync primitive

In scheduler:

- thread state, e.g., priority, blocked
- sync primitive state, e.g., blocked threads, waker threads
- sync primitive critical section

Implementation in a shared scheduler

User-level mutex implementation

```
/* Struct used to identify a mutex by address */
typedef struct {
  uint8_t _unused;
} mutex_t;

void mutex_lock(mutex_t *mutex) {
  /* Call to the scheduler to lock the mutex */
  scheduler_call(SCHEDULER_MUTEX_LOCK, (uintptr_t) mutex);
}

void mutex_unlock(mutex_t *mutex) {
  /* Call to the scheduler to unlock the mutex */
  scheduler_call(SCHEDULER_MUTEX_LOCK, (uintptr_t) mutex);
}
```

Implementation in a shared scheduler

Mutex lock in the scheduler

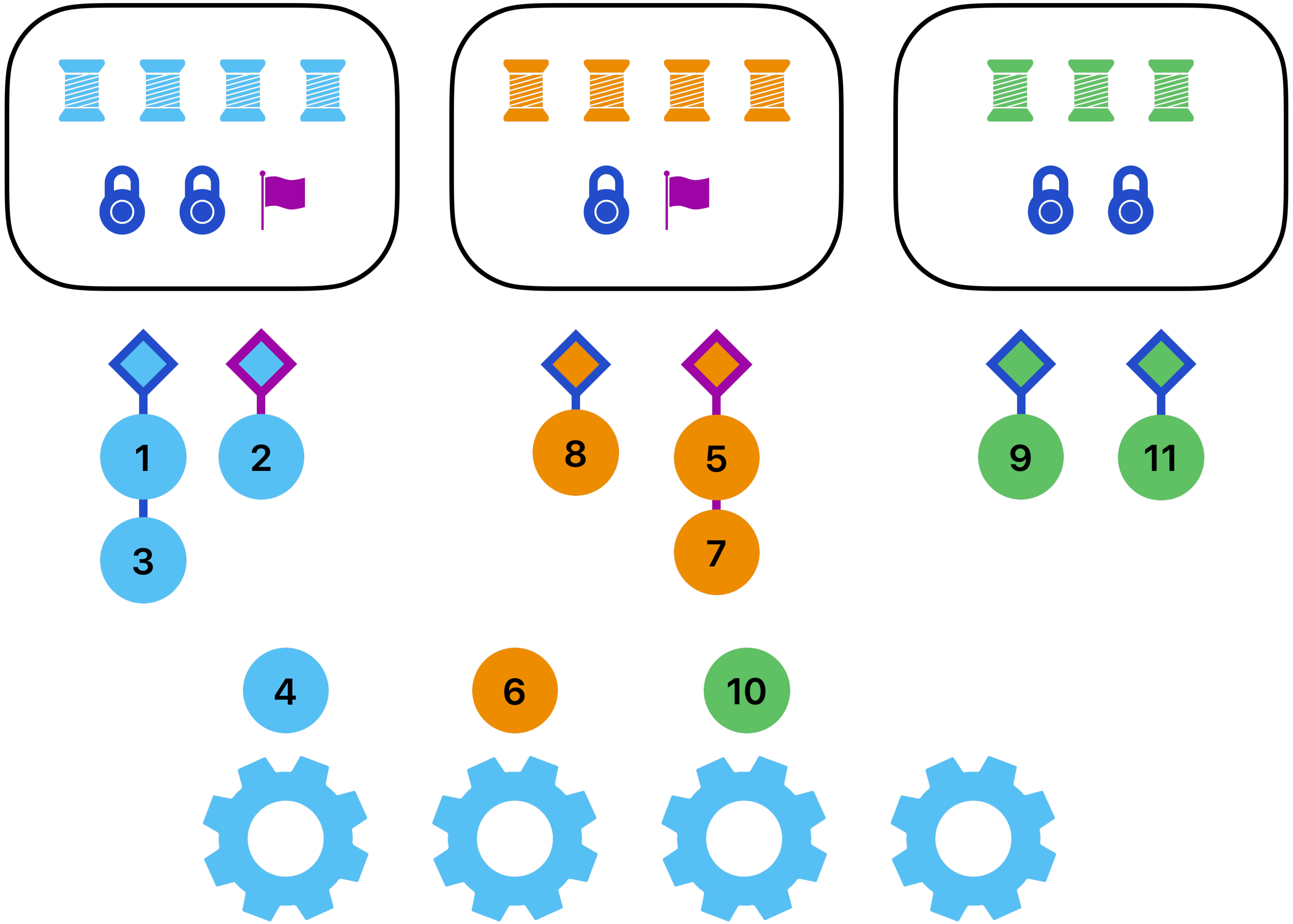
```
/* Lock the mutex at the address in the address space for the thread */
void scheduler_mutex_lock(thread_t *thread, space_t *space, uintptr_t address) {
  primitive_t *object = space_primitive(space, address);

  section_enter(&object->section);                /* Enter critical section of primitive */

  /* Check if already locked */
  if (object->state != MUTEX_UNLOCKED) {
    object->state = MUTEX_LOCKED;                 /* Thread can acquire the mutex exclusively */
    object->inheritor = thread;                   /* Thread inherits from waiters */
  } else {
    thread_suspend(thread);                       /* Stop the thread running */
    heap_insert(object->waiters, thread);         /* Add thread to priority heap */
  }

  section_exit(&object->section);                 /* Exit critical section of primitive */
}
```

Implementation in a shared scheduler

Mutex unlock in the scheduler

```
/* Unlock the mutex at the address in the space for the thread */
void scheduler_mutex_unlock(thread_t *thread, space_t *space, uintptr_t address) {
  primitive_t *object = space_primitive(space, address);

  section_enter(&object->section);                /* Enter critical section of primitive */

  thread_t *waiter = heap_dequeue(object->waiters);
  if (waiter != NULL) {
    thread_resume(waiter);                        /* Resume the waiting thread */
    object->inheritor = waiter;                   /* Unblocked a waiting thread */
  } else {
    object->state = MUTEX_UNLOCKED;               /* Nothing holds the mutex */
  }

  section_exit(&object->section);                 /* Exit critical section of primitive */
}
```

Operations in a shared scheduler



Operations in a shared scheduler

Thread creation

1. Create thread stack & context in address space
2. Enter scheduler
3. Create thread scheduling state with initial register state (pc, sp, thread ID)
4. Add thread to running set
5. Exit scheduler

Operations in a shared scheduler

Thread termination

1. Wait for thread to stop executing
2. Enter scheduler
3. Remove thread from running set
4. Destroy thread scheduling state
5. Exit scheduler
6. Destroy stack & context in address space

Operations in a shared scheduler

Blocking on a primitive

1. Find sync primitive at user-level
2. Enter critical section in scheduler
3. Get running thread’s state
4. Look up scheduling object
5. Set priority inheritor
6. Remove running thread from runnable set and add to blocked queue
7. Exit critical section

Operations in a shared scheduler

Unblocking a single thread blocked on a primitive

1. Find sync primitive at user-level
2. Enter critical section in scheduler
3. Look up scheduling object
4. Remove element from queue
5. Set inheritor to dequeued thread
6. Add dequeued thread to runnable set
7. Exit critical section

Operations in a shared scheduler

Unblocking all threads blocked on a primitive

1. Find sync primitive at user-level
2. Enter critical section in scheduler
3. Look up scheduling object
4. Remove all elements from queue
5. Add all dequeued threads to runnable set
6. Exit critical section

Memory allocation and usage

Data structures for scheduler state

- What data structures does the scheduler need to maintain its state?
  - An object for each thread
  - An object for each scheduling primitive
  - A collection for the blocked and waking threads for a given primitive
  - Mapping from user-level addresses to synchronisation primitives
- When are these data structures allocated and freed?
- Does insertion and removal of collections require allocation?

Allocating in the scheduler

- Allocations can require blocking until memory is available
- If you need to allocate in order to block, you can’t block to wait for the allocation
- The scheduler shouldn’t allocate when modifying scheduler state

Useful properties of blocked threads

- Threads can only be blocked on one primitive at a time
- There can never be more primitives with blocked threads than the number of threads
- The scheduler only needs to maintain state for primitives with a non-empty set of blocked threads

Turnstiles

- First appeared in Solaris 2.0 (SunOS 5.0) in 1992
- The scheduler tracks the state of a synchronisation primitive in a *turnstile* data structure
- Intrusive data structure in the thread state for being in a blocking set
- Intrusive data structure in the turnstile for being in a map of virtual address turnstile
- There is always the same number of turnstiles and threads
- The first thread to block on a primitive takes an unused turnstile
- The last thread to finish waiting on a primitive releases the turnstile

Turnstiles

User-level mutex lock

```
typedef struct {
  atomic(thread_id_t) owner;
} mutex_t;

void mutex_lock(mutex_t *mutex) {
  thread_id_t current = THREAD_NONE;
  /* Set the owner to self if it's currently none */
  while (!atomic_compare_exchange_strong(&mutex->owner, &current, thread_id_self())) {
    thread_id_t owner = current | THREAD_WAITERS;
    /* Mark the mutex as having waiters */
    if (atomic_compare_exchange_strong(&mutex->owner, &current, owner)) {
      /* Call to the scheduler to block on the mutex */
      scheduler_call(SCHEDULER_MUTEX_BLOCK, (uintptr_t)mutex, owner);
      break;
    }
  }
}
```

Turnstiles

Mutex block in the scheduler

```
/* Block on the mutex at the address in the address space for the thread */
void scheduler_mutex_block(
  thread_t *thread, space_t *space, uintptr_t address, thread_id_t owner
) {
  section_enter(&space->section);                     /* Enter critical section of space */

  turnstile_t *object = space_turnstile(space, address);
  if (object == NULL) {
    object = turnstile_from_pool();                  /* Allocate object for first waiter */
    space_turnstile_insert(space, address, object);
  }

  object->inheritor = thread_from_id(owner);   /* Current owner inherits from waiters */
  thread_suspend(thread);                             /* Stop the thread running */
  heap_insert(object->waiters, thread);             /* Add thread to priority heap */

  section_exit(&space->section);                      /* Exit critical section of space */
}
```

Turnstiles

User-level mutex unlock

```
void mutex_unlock(mutex_t *mutex) {
  thread_id_t current = thread_id_self();
  thread_id_t next_owner = THREAD_NONE;
  /* Release the lock initially assuming no waiter */
  while (
    !atomic_compare_exchange_strong(&mutex->owner, &current, next_owner)
  ) {
    /* Keep waking until another thread is unblocked */
    while (next_owner == THREAD_NONE) {
      next_owner = scheduler_call(SCHEDULER_MUTEX_UNBLOCK, (uintptr_t)mutex);
    }
  }
}
```

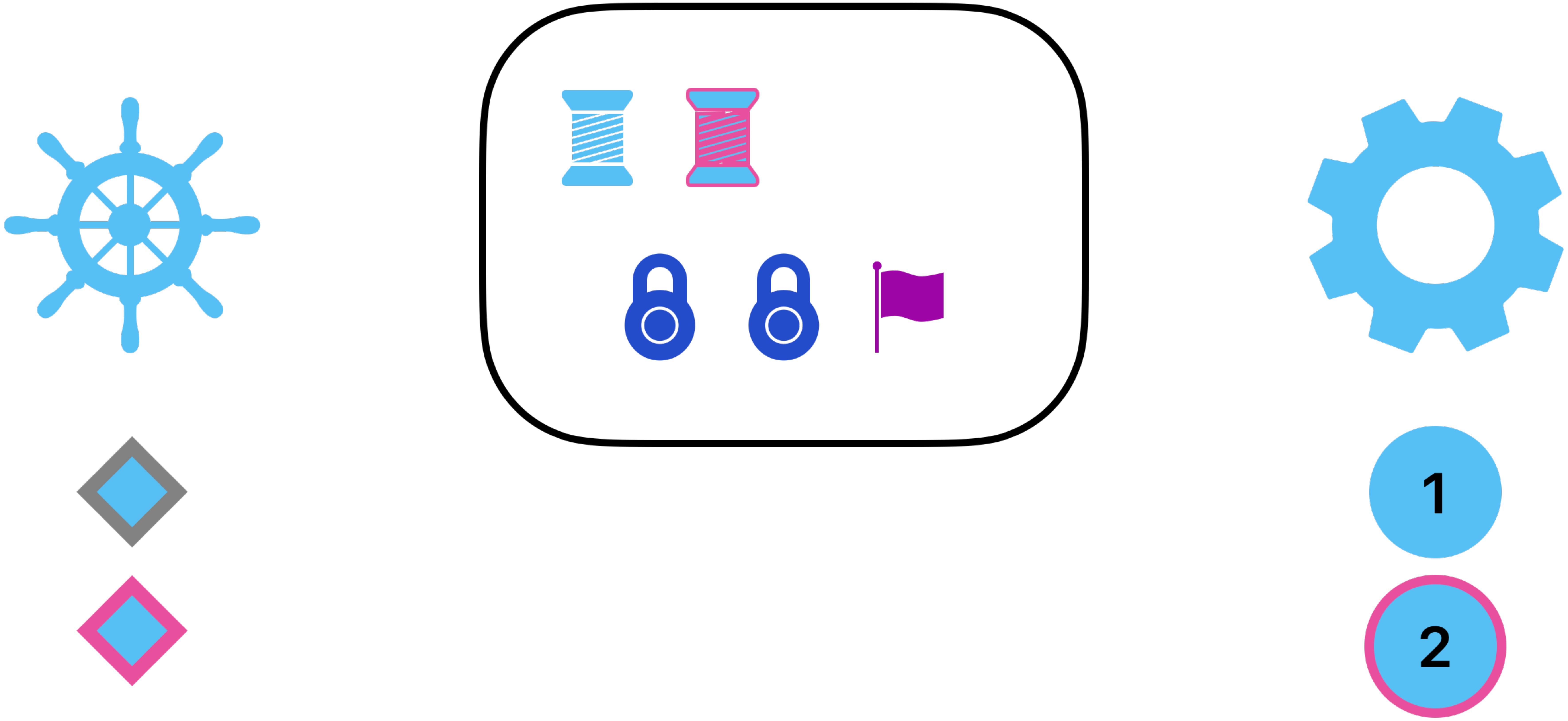
Turnstiles

Mutex unblock in the scheduler

```
/* Unblock a waiter on the mutex at the address in the space */
thread_id_t scheduler_mutex_unblock(thread_t *thread, space_t *space, uintptr_t address) {
   section_enter(&space->section);            /* Enter critical section of space */

   thread_id_t new_owner = THREAD_NONE;
   turnstile_t *object = space_turnstile(space, address);
   if (object != NULL) {
      thread_t *waiter = heap_dequeue(object->waiters);
      thread_resume(waiter);                   /* Resume the waiting thread */
      object->inheritor = waiter;              /* Unblocked a waiting thread */
      new_owner = waiter->id;

      if (heap_empty(object->waiters)) {
         space_turnstile_remove(space, address);
         turnstile_into_pool(object);          /* Remove object back into pool */
      } else {
         new_owner |= THREAD_WAITERS
      }
   }

   section_exit(&space->section);             /* Exit critical section of space */
   return new_owner;                          /* Indicate a thread was unblocked */
}
```

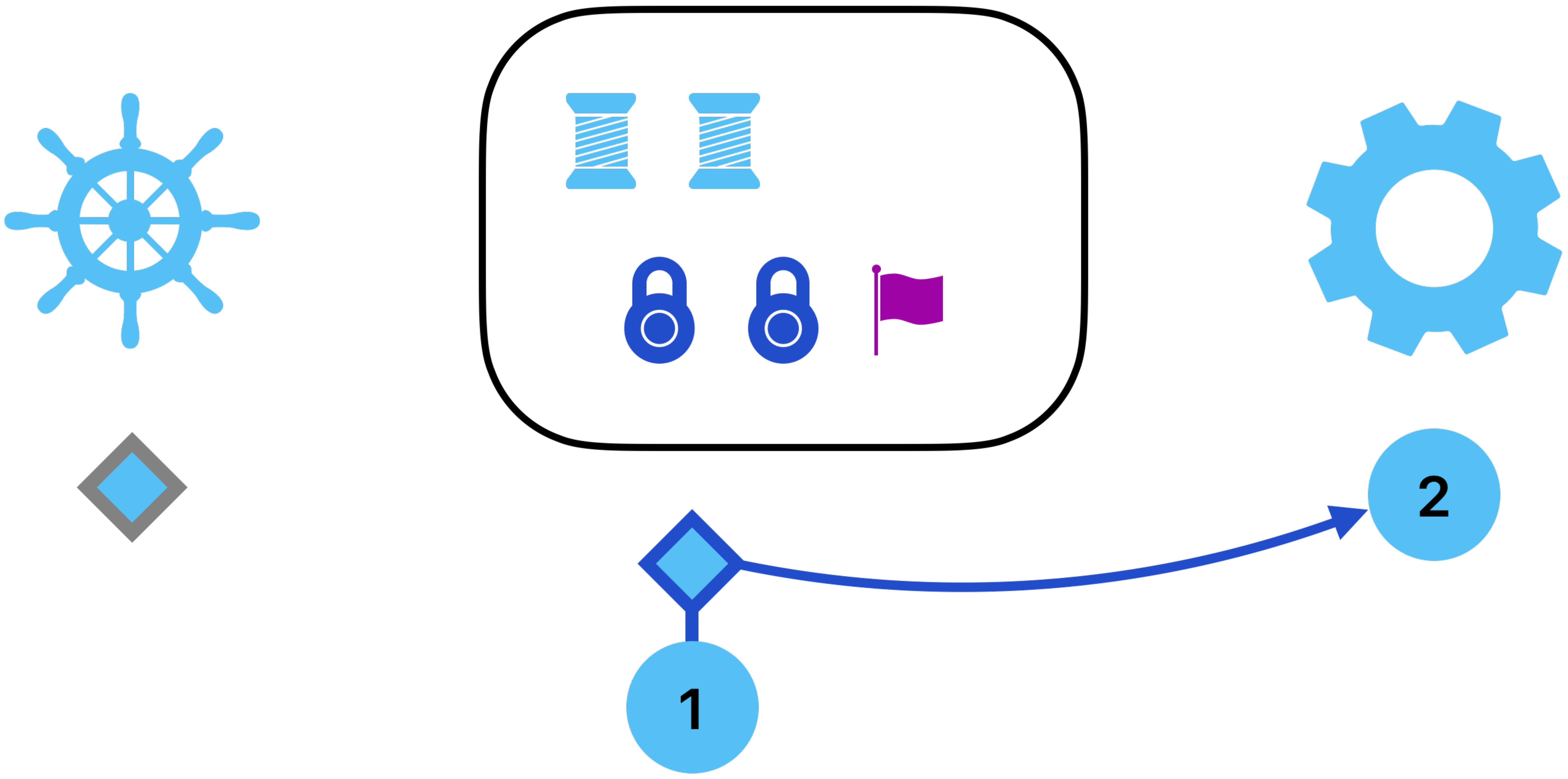
Turnstiles

Allocating a spawned thread



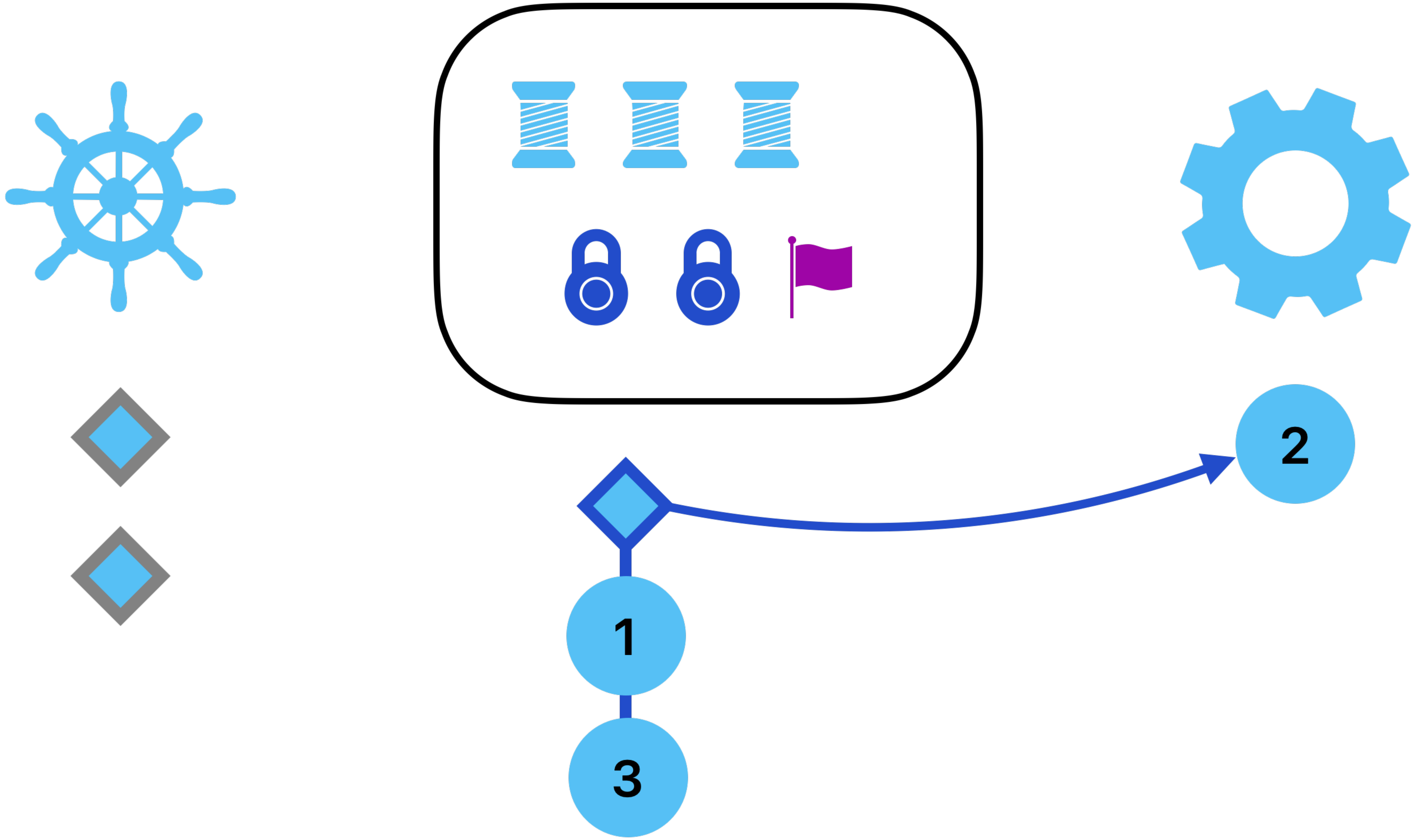
Turnstiles

First thread to block on a primitive



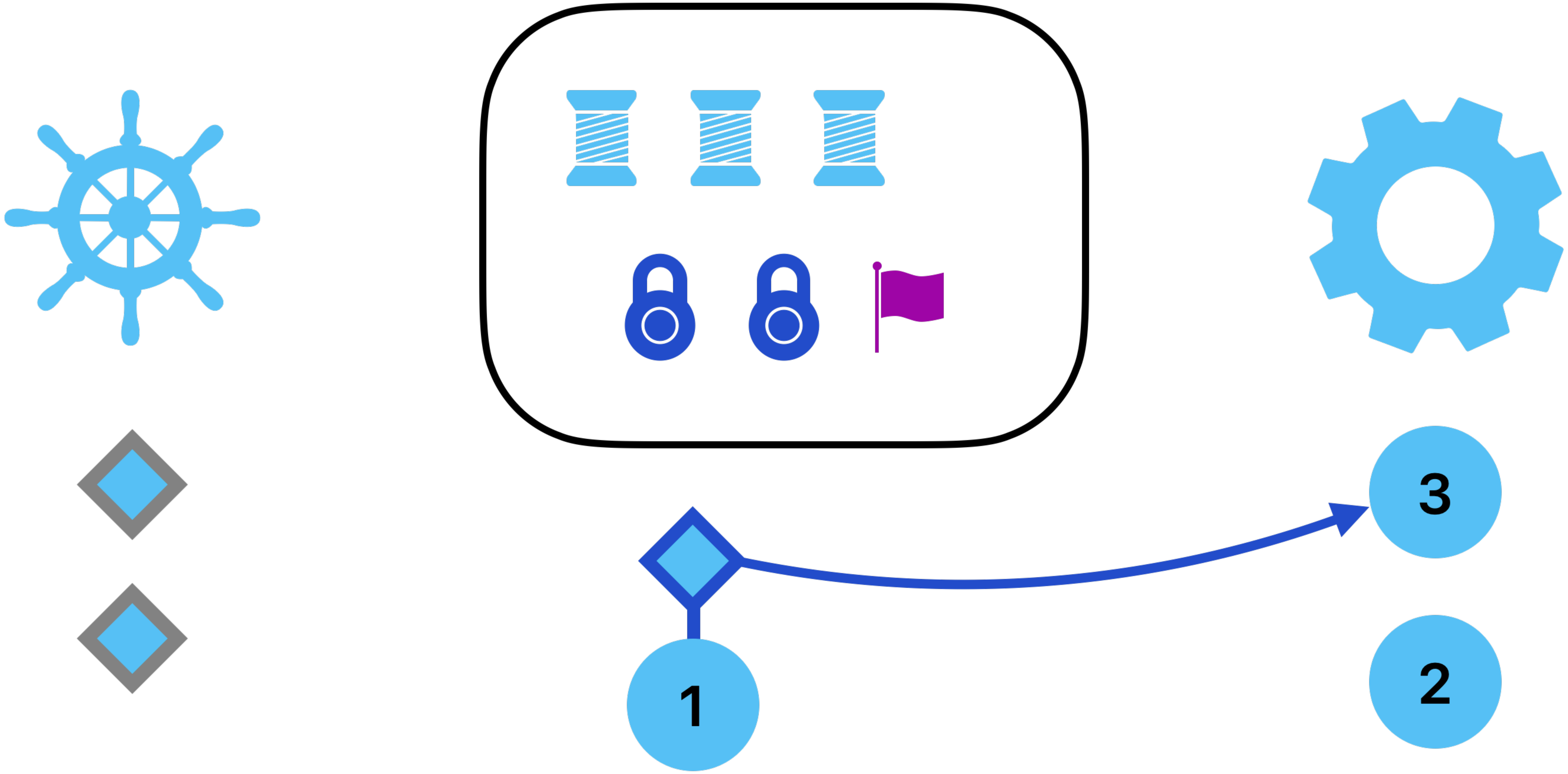
Turnstile

Additional thread blocking on a primitive



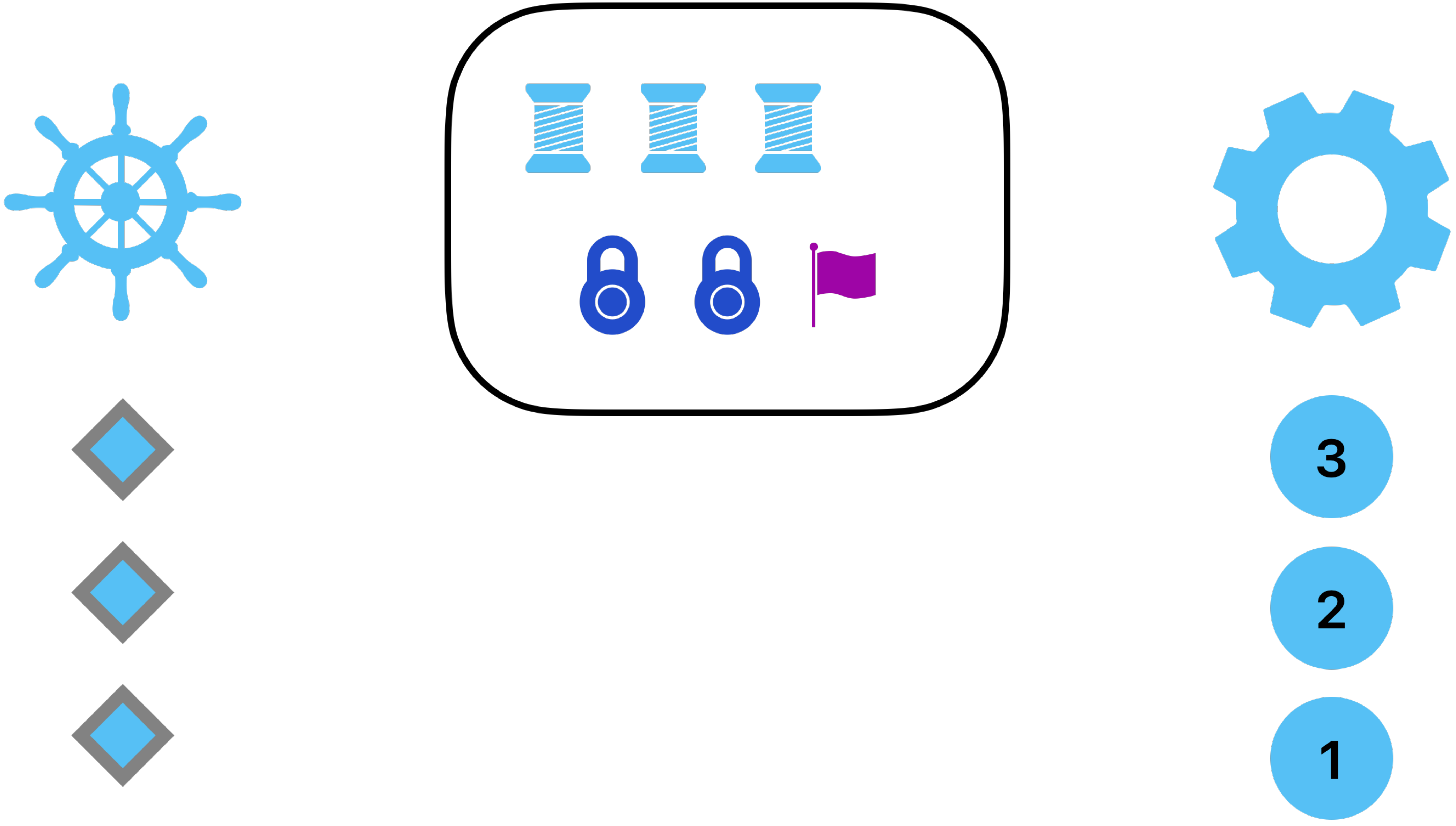
Turnstile

Thread unblocked with other waiters



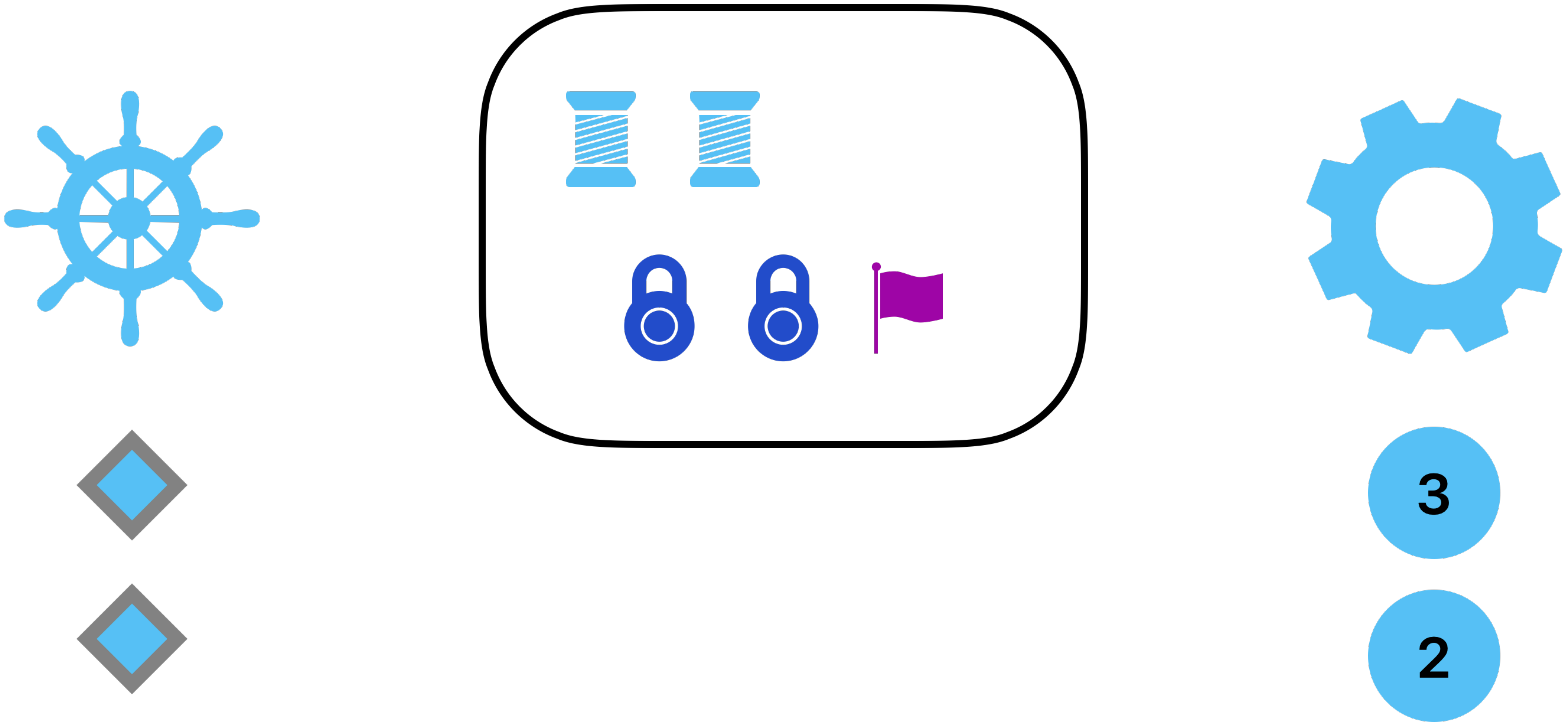
Turnstile

Last thread unblocked from a primitive



Turnstiles

Deallocating a terminated thread



Avoiding context switches

Costs of context switching

- Switching to the scheduler is expensive
  - At least a *mode switch* but potentially an *address space switch*
- Executing in a critical section prevents other threads from making progress
  - Prevents other scheduling operations from occurring

When to switch to the scheduler

- Only need to enter the scheduler to change scheduling state
  - If a thread blocks
  - If some threads are unblocked
- Don’t enter the scheduler when not required
  - If the primitive is already in the required state for a thread
  - If a thread changes the state of a primitive but there are no blocked threads

Atomics instead of critical sections

- Store the state of a synchronisation in an atomic variable
  - The current state of the managed resource
  - Whether there are any waiters
  - The identity of *some* thread that can inherit priority to guarantee progress

Atomic operation sequence

- Compare and exchange with value of required state to updated state
  - Success indicates no need to block and wait
- On failure, compare exchange with a value indicating waiters
  - Success indicates that the value hadn't changed state and now tracks there being waiters
- On failure, retry as the primitive has changed state and may now be in the required state
- If the state is changed and the previous value indicated waiters, then enter the scheduler to wake on the primitive

Mutex with atomics

Acquire:

1. Compare exchange (no holder, no waiters) -> (current thread, no waiters)
2. On failure:
   - if not already marked with waiters compare exchange from current to current with waiters (retry on failure)
   - block on the primitive in the scheduler
3. On success: lock is held

Release:

1. Exchange with (no holder, no waiters)
2. If previous value indicated waiters, wake on the address

Ordering of requests to the scheduler

- The order of operations on the atomic is inconsistent with the scheduler operations
- A thread may be preempted between updating the atomic and entering the critical section in the scheduler
- A thread releasing a resource can perform an unblock operation before the blocked thread becomes blocked
- If we only maintain synchronisation state while there are blocked threads, there is nowhere to track the 'pre-posted' block operation

Handling delayed blocking operations

- The scheduler needs to verify the validity of a blocking operation during the critical section
- If the blocking operation is no longer valid once the critical section is reached, it is ignored and the thread can retry its operation
- An invalid blocking operation must never result in a blocked thread, but a valid blocking operation can be safely ignored
- If the state of the atomic has not changed since blocking thread last read it, then blocking operation must still be valid
- If the scheduler can read the memory of the atomic directly, it can check that it is the same as the value seen by the blocking thread

The fast user-level mutex (futex)

- First appeared in Linux 2.6.0 in 2002
- Implement the state of the primitive at user-level using atomics
- Only call into the scheduler when blocking or un-blocking is required
- Pass the observed value of the atomic along with waits
- Check the value of the atomic during the critical section
- Ignore a blocking operation if the value in the atomic has changed

The fast user-level mutex

User-level mutex lock

```
typedef struct {
  atomic(thread_id_t) owner;
} mutex_t;

void mutex_lock(mutex_t *mutex) {
  thread_id_t current = THREAD_NONE;
  thread_id_t self = thread_id_self();
  thread_id_t waiters = 0;
  /* Set the owner to self if it's currently none */
  while (!atomic_compare_exchange_strong(&mutex->owner, &current, self | waiters)) {
    thread_id_t owner = current | THREAD_WAITERS;
    if (atomic_compare_exchange_strong(&mutex->owner, &current, owner)) {
    /* Mark the mutex as having waiters */
      switch (scheduler_call(SCHEDULER_MUTEX_BLOCK, (uintptr_t)mutex, owner)) {
      case WAITED_NO_WAITERS: waiters = 0;              break;
      case WAITED_WAITERS:    waiters = THREAD_WAITERS; break;
      default: break;
      }
    }
    current = THREAD_NONE;
  }
}
```

The fast user-level mutex

Mutex block in the scheduler

```c
/* Block on the mutex at the address in the address space for the thread */
int scheduler_mutex_block(thread_t *thread, space_t *space, uintptr_t address, thread_id_t owner) {
  section_enter(&space->section);              /* Enter critical section of space */

  if (read_from_space(space, address) != owner) {
    section_exit(&space->section);             /* Exit critical section of space */
    return COMPARE_FAIL;
  }

  turnstile_t *object = space_turnstile(space, address);
  if (object == NULL) {
    object = turnstile_from_pool();            /* Allocate object for first waiter */
    space_turnstile_insert(space, address, object);
  }

  object->inheritor = thread_from_id(owner);  /* Current owner inherits from waiters */
  thread_suspend(thread);                      /* Stop the thread running */
  heap_insert(object->waiters, thread);        /* Add thread to priority heap */

  section_exit(&space->section);               /* Exit critical section of space */
  return resume_response(thread);
}
```

The fast user-level mutex

User-level mutex unlock

```
void mutex_unlock(mutex_t *mutex) {
  thread_id_t current = atomic_exchange(
    &mutex->owner,
    THREAD_NONE
  );
  /* Wake from the queue in the scheduler if there are waiters  */
  if ((current & THREAD_WAITERS) != 0) {
    scheduler_call(
      SCHEDULER_MUTEX_UNBLOCK,
      (uintptr_t)mutex
    );
  }
}
```

The fast user-level mutex

Mutex unblock in the scheduler

```
/* Unblock a waiter on the mutex at the address in the space */
thread_id_t scheduler_mutex_unblock(thread_t *thread, space_t *space, uintptr_t address) {
    section_enter(&space->section);                /* Enter critical section of space */

    turnstile_t *object = space_turnstile(space, address);
    if (object != NULL) {
        thread_t *waiter = heap_dequeue(object->waiters);
        object->inheritor = waiter;                /* Unblocked a waiting thread */

        if (heap_empty(object->waiters)) {
            space_turnstile_remove(space, address);
            turnstile_into_pool(object);           /* Remove object back into pool */
            thread_resume(waiter, WAITED_WAITERS);
        } else {
            thread_resume(waiter, WAITED_NO_WAITERS);
        }
    }

    section_exit(&space->section);                 /* Exit critical section of space */
    return new_owner;                              /* Indicate a thread was unblocked */
}
```

A challenge

How would you implement this on seL4?

- An implementation of the futex API
- Checking the value of the atomic primitive from within the critical section
- Without requiring anything outside the address space to read arbitrarily within the address space
- Using turnstiles to only maintain state for primitives with blocked threads
- Allowing primitives with no blocked threads to move in memory or be created spontaneously
- Without any modification to the seL4 microkernel

Summary

Fast, efficient, and correct scheduling primitives

1. How synchronisation primitives inform scheduling policy
2. Sharing a scheduler between applications
3. Efficiently managing the state of synchronisation primitives
4. Avoiding context switches to a shared scheduler

Bibliography

- Sandeep Khanna, Michael Sebrée, and John Zolnowsky. Realtime Scheduling in SunOS 5.0. In *USENIX Winter Technical Conference*, 1992.
- J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond Multiprocessing... Multithreading the SunOS Kernel. In *USENIX Summer Technical Conference*, 1992.
- Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Ottawa Linux Symposium*, 2002.