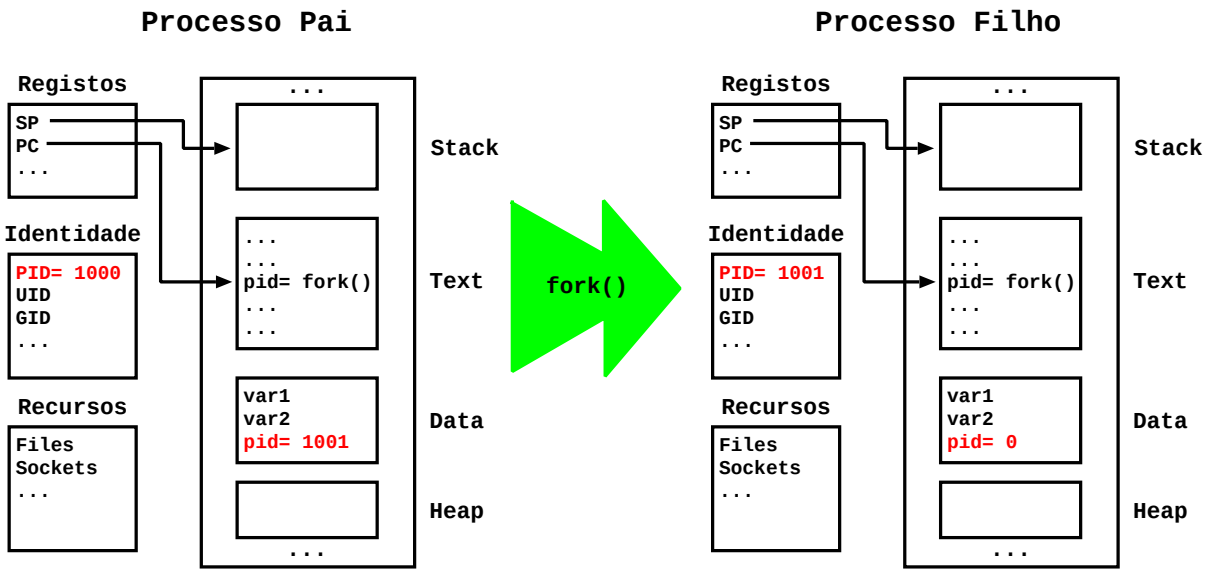
Parallel programming using threads

Extended and adapted by Eduardo R. B. Marques from original slides by Ricardo Rocha and Fernando Silva

Departamento de Ciência de Computadores
Faculdade de Ciências
Universidade do Porto

Computação Paralela 2018/2019

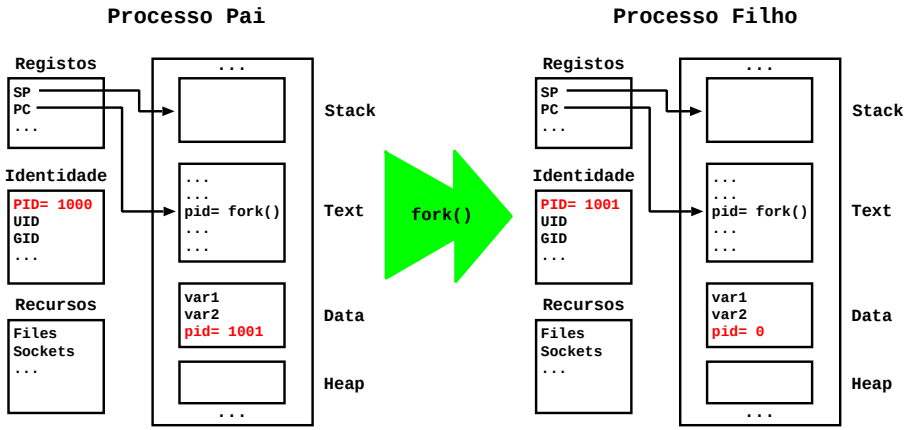
Revision: concurrent programming with processes



Revision questions:

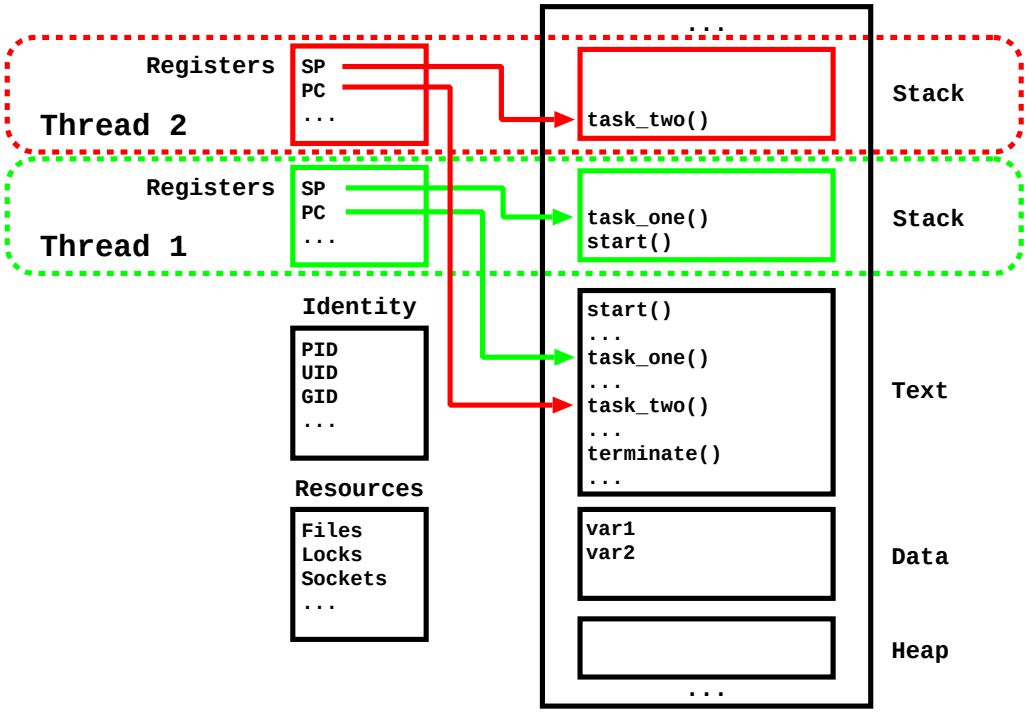
- How does `fork()` work?
- What is shared (and not shared) between parent and child processes?
- How may processes interact?

Concurrent programming with processes



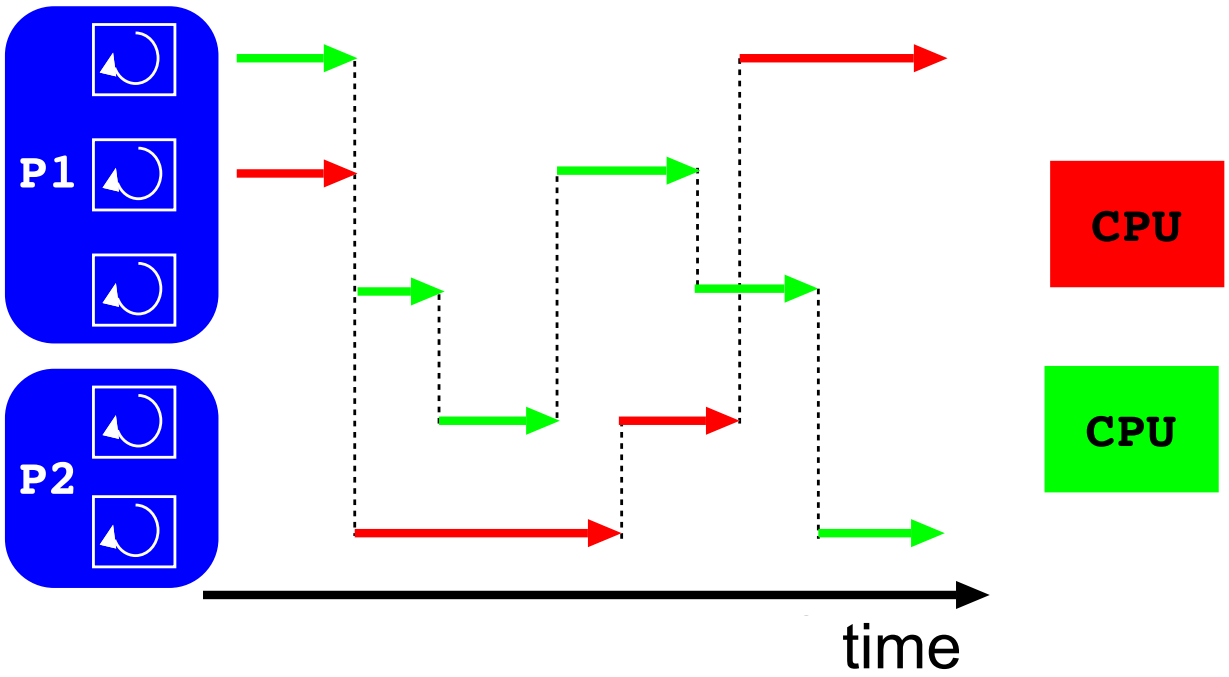
- Parent process invokes **`fork()`** to create a child process.
- The child process has a **separate memory address space**, initially a copy of the parent process' memory. Some OS resources like file descriptors and network sockets are shared between child and parent process though.
- Processes interact using OS-supported shared-memory, memory-mapped I/O, or other inter-process communication (IPC) primitives (message queues, semaphores, ...).

Multithreaded processes



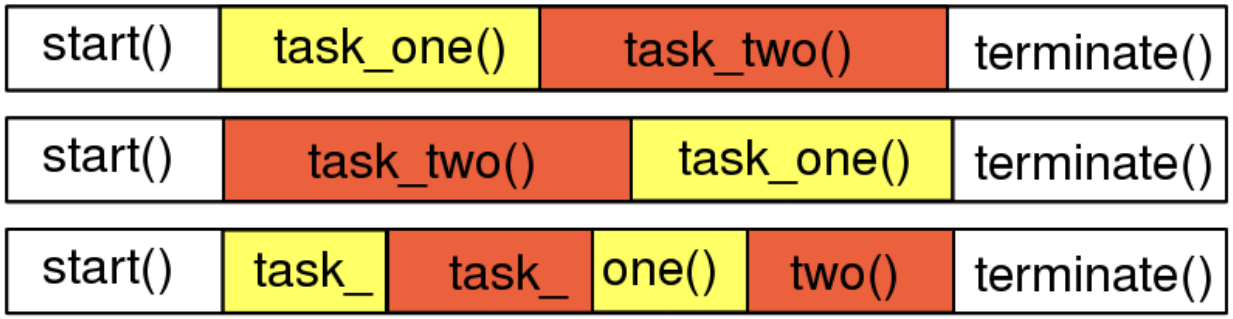
- **Multithreaded Process = { Threads } + { Shared resources }**
- A **thread** is a sequential execution flow within the process, with its own individual stack and program counter.
- **Threads transparently share the memory address space** and other process resources.

Multithreaded process execution



- All threads in the process execute concurrently, possibly on different processors / cores over time.
- Thread-level (as well as process-level) scheduling is typically preemptive and non-deterministic. Execution interleavings and processor / core allocation vary from execution to execution.

Using threads for parallel computing



- Parallel computing employs concurrency abstractions (message-passing, shared-memory, ...) with the aim of reducing the overall execution time of a computational workload.
- In the case of threads, we need to exploit the concurrency between actions in different threads (like computation or I/O) that can be executed independently and in any order.

Threads vs. Processes

Advantages of using threads:

- A more convenient programming model.
- The use of a single shared address space reduces the memory load on the system.
- Latencies for synchronization and context switch are typically lower.

Transparent resource sharing requires careful programming however, to ensure the correct operation of the program. Correct operation is usually termed **thread safety**. Particular care has to be taken to avoid race conditions, deadlocks, or memory corruption.

POSIX threads (pthreads)

- POSIX threads: a standardised C library interface for multithreading (IEEE 1003.1c-1995).
- Other libraries with similar intent are defined (e.g., Windows Threads library) and many languages provide built-in support for threads (e.g. Java). The thread model has the same core traits in any case.
- Getting started:
  - Source code needs to include **`pthread.h`**:

```
#include <pthread.h>

int main(int argc, char** argv) {
  // Ready? Set? Go!
  ...
}
```

  - Programs must link with the pthreads library using **`-lpthread`**, e.g.:

```
gcc myProgram.c -lpthread -o myProgram
```

Hello pthreads!

```
#include <pthread.h>
void* thread_main(void* arg) {
  long rank = (long) arg;
  printf("Hello from thread %ld\n", rank);
  return (void*) (rank + 1);
  // pthread_exit((void*) rank+1) could also be used equivalently
}
int main(int argc, char** argv) {
  long n_threads = atol(argv[1]);
  pthread_t* vth = (pthread_t*) malloc(sizeof(pthread_t) * (n_threads-1));
  for (long rank = 0; rank < n_threads - 1; rank++) {
    pthread_create(&vth[rank], NULL, &thread_main, (void*) rank);
  }
  printf("Hello from main thread\n");
  for (long rank = 0; rank < n_threads - 1; rank++) {
    long rval;
    pthread_join(vth[rank], (void**) &rval);
    printf("Thread %ld done, returned %ld\n", rank, rval);
  }
  free(th);
  printf("Done");
  return 0;
}
```

Hello pthreads! (2)

- Executing

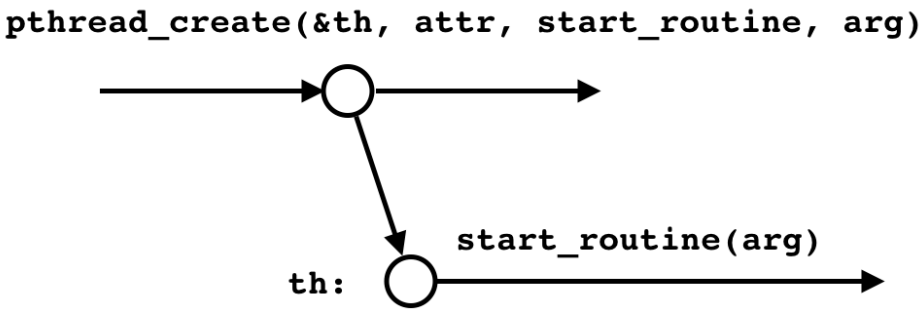
```
./hello_word.bin 4
```

- ... one may obtain (among several possible outputs):

```
Hello from thread 0
Hello from main thread
Hello from thread 1
Hello from thread 2
Thread 0 done, returned 1
Thread 1 done, returned 2
Thread 2 done, returned 3
Done
```

- Just print-outs, no use of shared data (which we will see in later examples).
- First, let us describe how the involved primitives work: `pthread_create` and `pthread_join`.

Thread creation

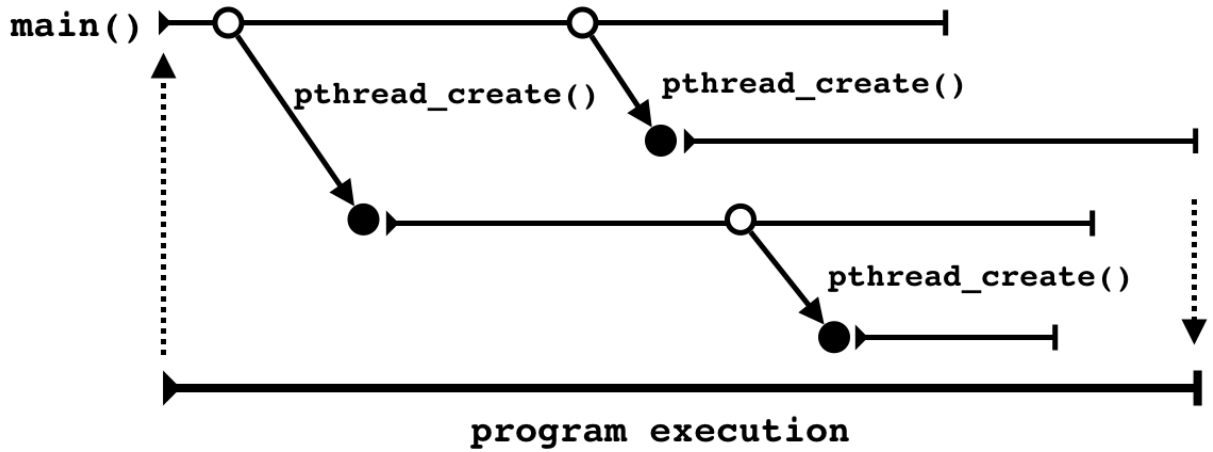



```
int pthread_create(pthread_t *th, pthread_attr_t *attr,
  void * (*start_routine)(void *), void *arg)
```

`pthread_create` creates a new thread:

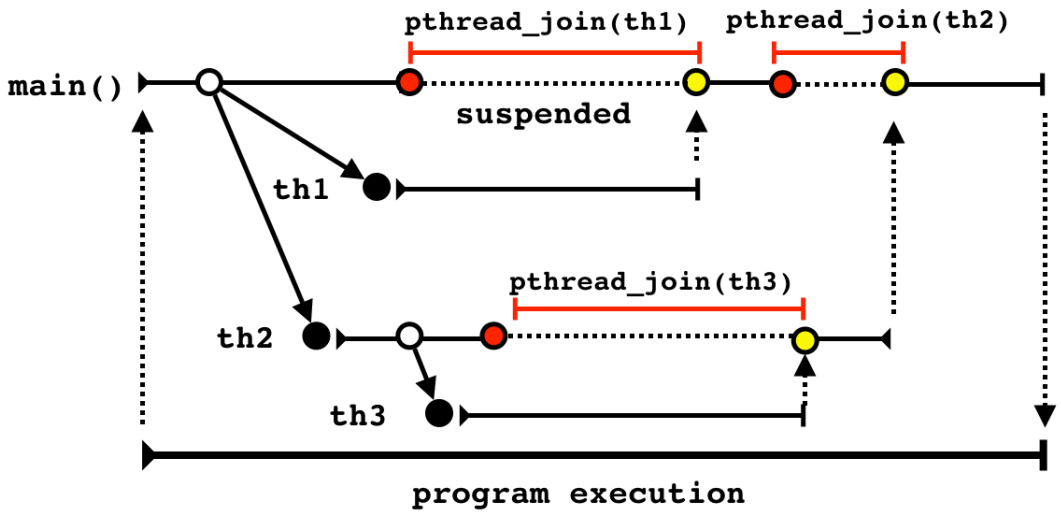
- `th` is the thread handle returned on exit;
- `attr` defines the thread's attributes (`NULL` for defaults);
- `start_routine` is a function defining the entry point for the thread;
- `arg` is the argument to pass to `start_routine`;
- 0 is returned on success, non-zero value indicates an error.

Multithreaded program lifecycle



- The C program starts in `main()` (as usual) that runs in its own thread, the "main" thread.
- New threads are dynamically created using `pthread_create()`.
- A thread ends execution when its starting procedure returns **OR** it calls `pthread_exit()`. It is also possible to use `pthread_cancel` to stop a thread from another thread (but we won't make use of it).
- The overall program execution ends when all threads are terminated **OR** one of the threads calls `exit()` causing all others to be abruptly terminated.

Joining threads




```
int pthread_join(pthread_t th, void **thread_return);
```

- **`pthread_join(th, thread_return)`** suspends the calling thread until **`th`** terminates. The return value of **`th`** through its start procedure or **`pthread_exit`** is given in **`thread_return`** (if set to **`NULL`**, **`th`**'s return value will be ignored).
- Note: **`th`** must be joinable, i.e., not be in a detached state set using **`pthread_detach`** (we won't make use of this feature).

Summary of other pthread lifecycle functions

```
pthread_t pthread_self(void);
void pthread_exit(void* val);
int pthread_tryjoin_np(pthread_t th, void **retval);
int pthread_timedjoin_np(pthread_t th, void **retval,
                         struct timespec* time);
int pthread_detach(pthread_t th);
int pthread_cancel(pthread_t th);
```

- `pthread_self()` returns the handle of calling thread.
- `pthread_exit(v)` terminates calling thread with a return value of `v`.
- `pthread_tryjoin_np(th, r)` join `th` or return immediately (does not block).
- `pthread_timedjoin_np(th, r, to)` join `th` with timeout `tp`[1]
- `pthread_detach(th)` detaches `th` (cannot be joined later).
- `pthread_cancel(th)` sends a cancellation request to `th`.

[1] Similarly to `join`, other primitives have "try" and time-out based variants.

Caution with stack-allocated data

- Threads SHOULD NOT share stack-allocated data through pointers to local function variables.
- In particular, be careful with the start routine argument for **`pthread_create`**, and the return value of threads:

```
void foo() {
  some_data_t localVar = ...;
  pthread_create(..., start, &local_var); // WRONG!
}
void* start_routine(void * arg) {
  some_data_t localVar = ...;
  return &local_var; // OR pthread_exit(&local_var); // WRONG!
}
```

- In these cases, you may use primitive values (disguised as **`void*`**, as as in the example). If pointers are used instead, they should refer to (valid) data in the global address space (heap or static-allocated).

Dot product – definition

- The dot product of vectors $u = [u_0, \ldots, u_{n-1}]$ and $v = [v_0, \ldots, v_{n-1}]$ is given by

$$u \cdot v = \sum_{i=0}^{n-1} u_i \times v_i = (u_0 \times v_0) + \ldots + (u_{n-1} \times v_{n-1})$$

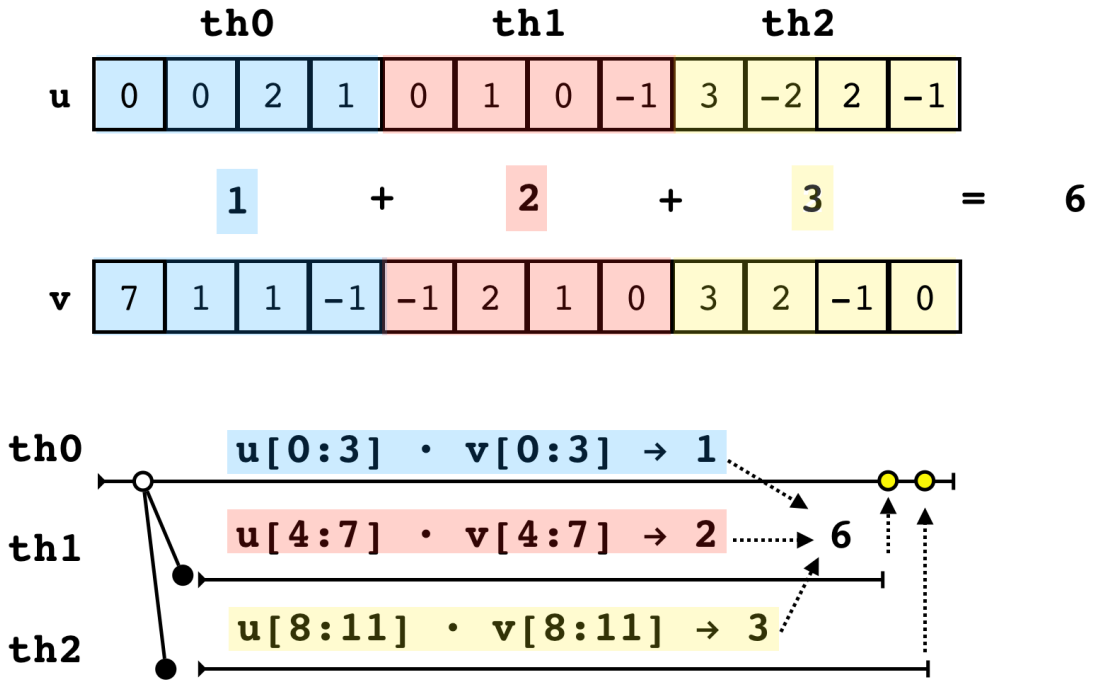
- Example: if $u = [0, 2, 1, 3]$ and $v = [1, 0, -1, 3]$ then
  $u \cdot v = (0 \times 1) + (2 \times 0) + (1 \times -1) + (3 \times 3) = 8$

Dot product – sequential code

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  for (int i = 0; i < n; i++) {
     r += u[i] * v[i];
  }
  return r;
}
```

- How can this code be parallelised using multiple threads?
- Note that there are no dependencies between iterations in respect to u[i] * v[i].

Dot product – parallelisation approach



- Let thread $i = 0, \ldots, N-1$ calculate a partial dot product:

$$r_i = u[i \times k : (i+1) \times k] \cdot v[i \times k : (i+1) \times k]$$

  where $k = n/N$.
- The partial dot products can be summed (reduced) to obtain the final result $r = r_0 + \ldots + r_{N-1}$.

Dot product – code for thread coordination

```
typedef struct {
  ... // next slide
} shared_data_t;
shared_data_t sd; // shared data global variable
void* do_work(void *arg) { ... } // next slide
int parallel_dot_product(int* u, int* v, int n, int n_threads) {
  pthread_t* vth = (pthread_t*) malloc((n_threads-1)*sizeof(pthread_t));
  ...
  for (long rank = 1; rank < n_threads; rank++)
    pthread_create(&vth[rank-1], NULL, &do_work, (void*) rank);
  do_work((void*) 0);
  for (long rank = 1; rank < n_threads; rank++)
    pthread_join(vth[rank-1], NULL);
  ...
  free(vth);
  return sd.result;
}
```

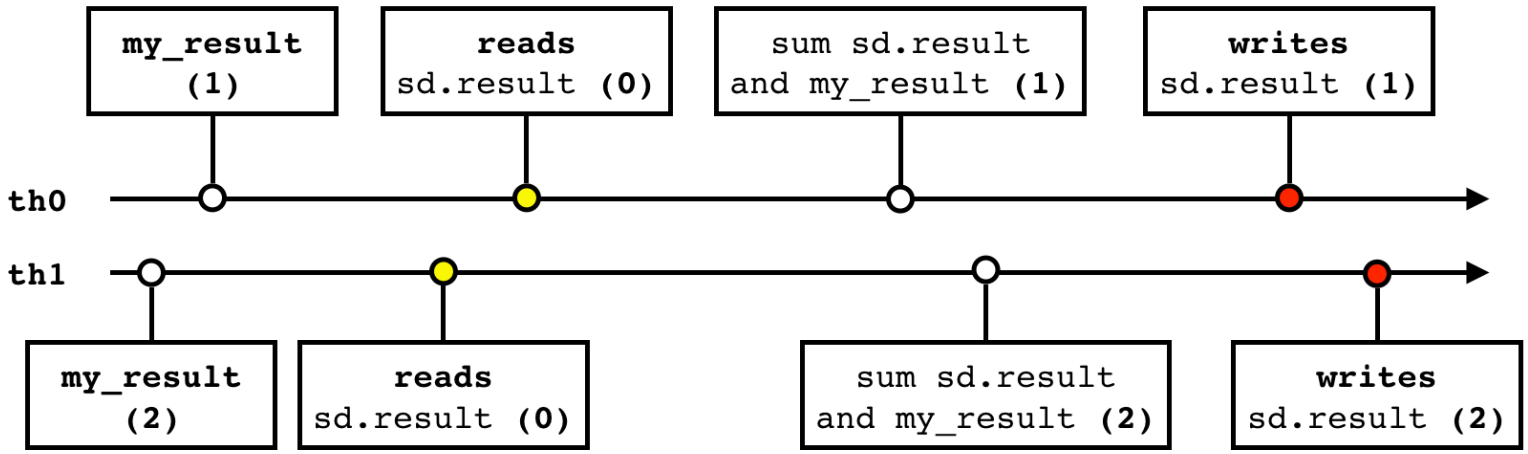
The code is similar to the “Hello pthreads” example.

Dot product – parallel computation per each thread

```
typedef struct {
  int n;                  // global size of vectors
  int *u, *v;             // pointers to the vectors
  int result;             // result
  int n_threads;          // number of threads
  ...
} shared_data_t;
shared_data_t sd; // shared data global variable
void* do_work(void* arg) {
  long rank = (long) arg;
  int  my_n = sd.n / sd.n_threads;
  int *my_u = sd.u + rank * my_n ,
      *my_v = sd.v + rank * my_n ;
  int my_result = dot_product(my_u, my_v, my_n); // sequential version
  ... // is there something missing?
  sd.result = sd.result + my_result; // update global result
  ... // is there something missing?
  return 0;
}
```

Is there something (important) missing?

Dot product – race conditions



- Suppose that two threads simultaneously execute

  $$\text{sd.result} = \text{sd.result} + \text{my\_result}$$ [2]

- A possible interleaving of actions (shown above) may lead to `sd.result` ending up with the value 2 **rather than** $3 = 1 + 2$.
- There is a **race condition** over `sd.result`!
- **Definition**: a race condition happens when (1) two or more threads simultaneously access the same data AND (2) at least one of them is a writer.

[2] or equivalently sd. result += my_result

Dot product – effect of race condition

- The result may be wrong and be different from time to time, e.g.:

```
> ./seq_dp.bin 1000 # sequential version
u . v = -1654
> ./par_dp.bin 1000 1000  # wrong result
u . v = -1741
Threads: 1000
> ./par_dp.bin 1000 1000  # this time same result as seq. version
u . v = -1654
Threads: 1000
> ./par_dp.bin 1000 1000 # another wrong result
u . v = -1730
Threads: 1000
```

- Note: the race is only frequently observable for a high number of threads. A programming mistake such as this may go unnoticed easily if only a few basic tests are conducted.
- Concurrency bugs are many times subtle and hard to detect and replicate.

Dot product – using a mutex

We need to use a **mutex**!

```
typedef struct {
  ...
  pthread_mutex_t mutex; // mutex for updating result
} shared_data_t;
shared_data_t sd; // shared data global variable
void* do_work(void* arg) {
  ...
  pthread_mutex_lock(&sd.mutex); // enter critical region
  sd.result = sd.result + my_result; // update global result
  pthread_mutex_unlock(&sd.mutex); // leave critical region
  ...
}
int parallel_dot_product(int* u, int* v, int n, int n_threads) {
  ...
  pthread_mutex_init(&sd.mutex, NULL); // initialization
  ...
  pthread_mutex_destroy(&sd.mutex); // tear-down
  ...
}
```

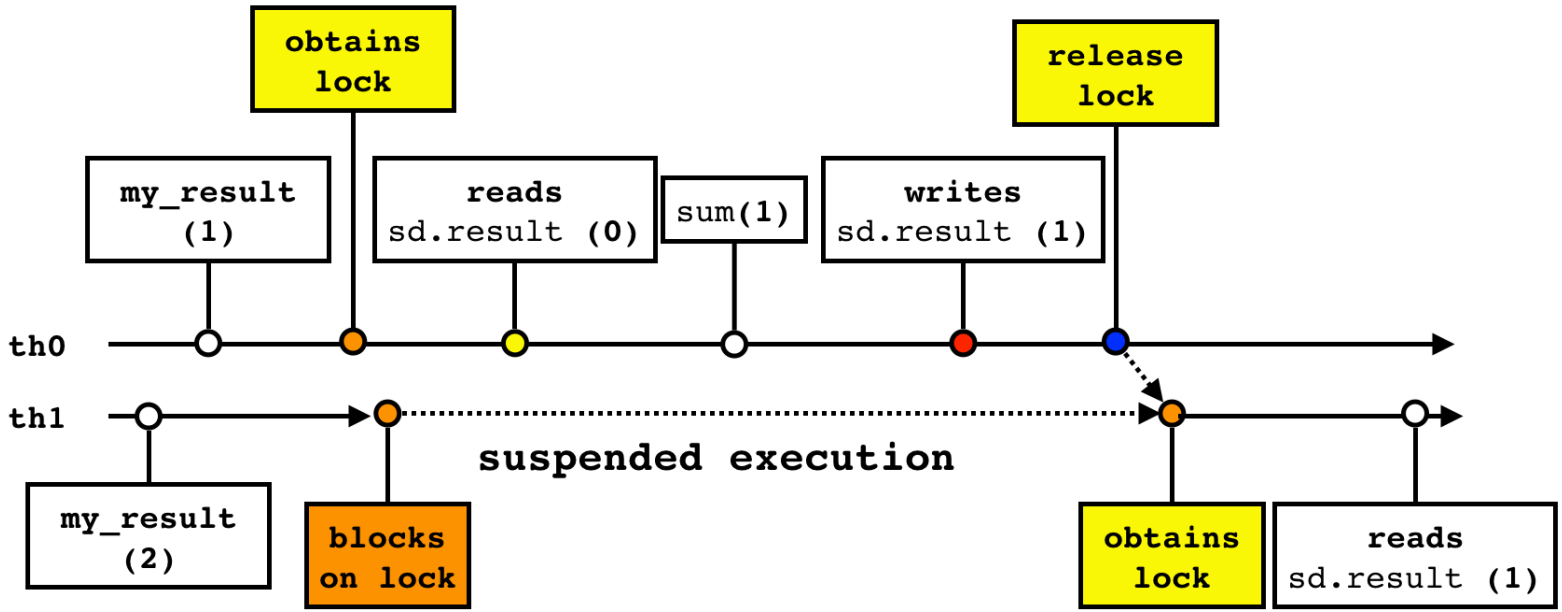
Pthread mutex functions

In the dot product example, we ensure absence of races by employing a a **mutex** (MUTual EXclusion) resource. A mutex can be owned, i.e. "locked", by at most one thread at any given time.

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- `pthread_mutex_init(mutex, attr)` initialises `Code` with given attributes `attr` (usually `attr=NULL` for defaults).
- `pthread_mutex_lock(mutex)` blocks execution of calling thread until it obtains the lock over the mutex.
- `pthread_mutex_unlock(mutex)` releases the lock over the mutex.
- `pthred_mutex_destroy(mutex)` tears-down the mutex.

Dot product – race condition eliminated



- Update over `sd.result` is now synchronised using mutual exclusion.
- After a thread obtains the lock, other competing threads will block until the lock is released.

Bad use of mutex objects – a few examples

Typical programming mistakes (among others):

- Lock is not released – other threads may block forever – **deadlock!**

```
pthread_mutex_lock(&sd.mutex);
sd.result = sd.result + my_result;
// pthread_mutex_unlock(&sd.mutex); Oops, I forgot!
```

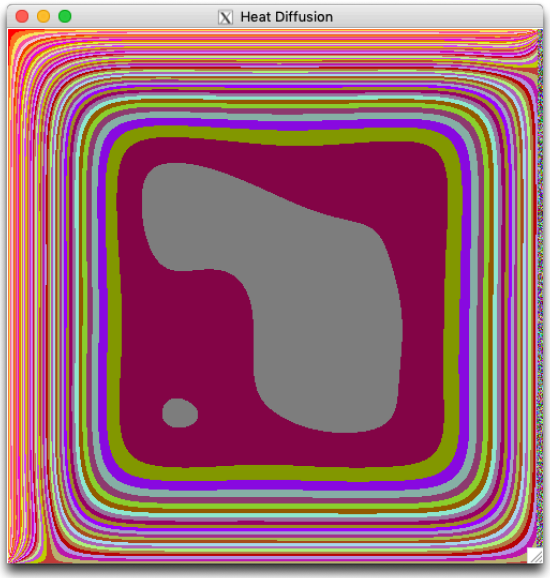
- Lock also covers code outside critical region – **inhibits parallelism!**.

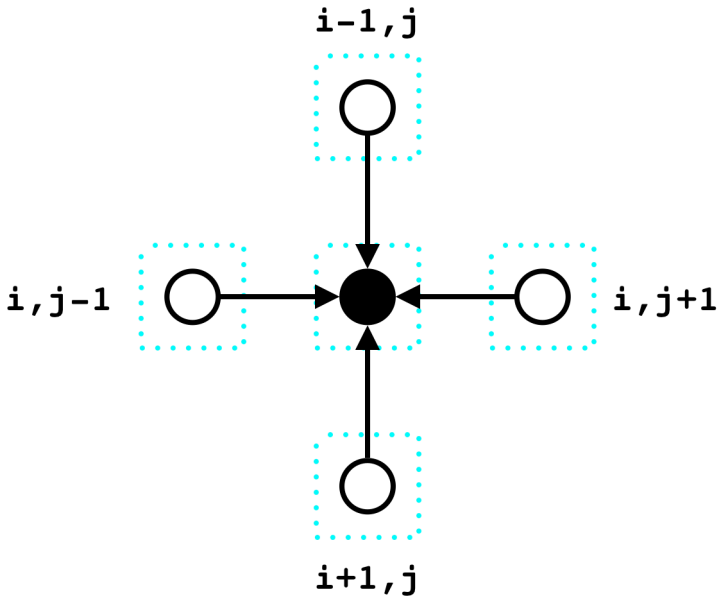
```
pthread_mutex_lock(&sd.mutex);
int my_result = dot_product(my_u, my_v, my_n);
sd.result = sd.result + my_result;
pthread_mutex_unlock(&sd.mutex);
```

- Lock does not cover entire critical region – **atomicity violation!** – code does not logically execute as an atomic transaction. The fragment below allows races.

```
int v = sd.result;
pthread_mutex_lock(&sd.mutex);
sd.result = v + my_result;
pthread_mutex_unlock(&sd.mutex);
```

Heat diffusion algorithm





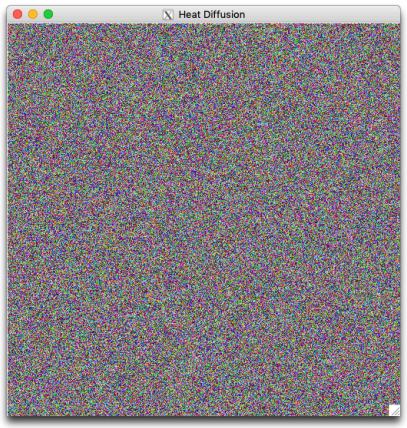
- To simulate heat diffusion in 2-D space, a finite difference scheme can be employed iteratively over a grid.
- For position (i, j) a new value $x'_{i,j}$ is calculated as:

$$x'_{i,j} = x_{i,j} + \alpha(x_{i,j-1} + x_{i,j+1} + x_{i+1,j} + x_{i-1,j} - 4x_{i,j})$$

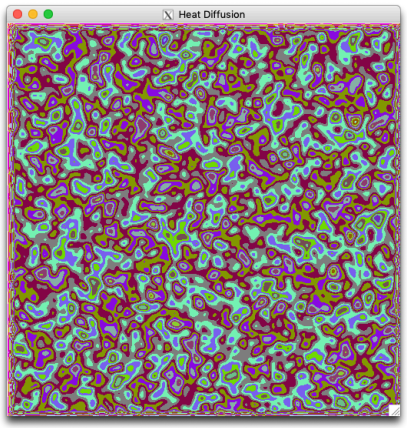
- $\alpha > 0$ is a constant parameter.
- The algorithm iterates until a steady-state solution is found.

Heat diffusion – evolution

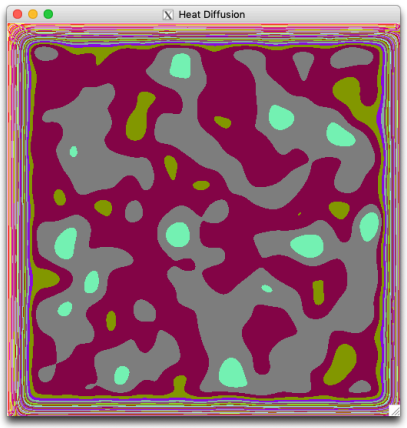
initial state



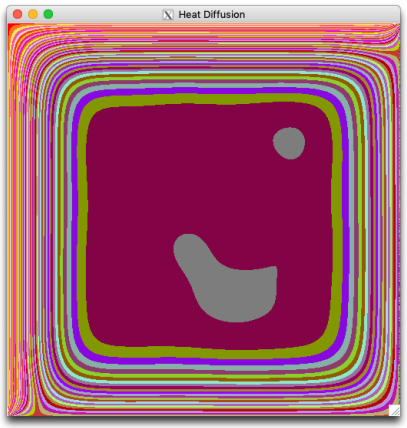
100 iterations



1000 iterations



10000 iterations



- The code for heat diffusion: **`seq_hd.c`** (sequential) and **`par_hd.c`** (parallel).
- A X11 visualisation window appears (use **`-h`** for available options; use **`make GUI=0`** to disable the GUI).

Heat diffusion – computation step

```
double heat_diffusion(
  int m, int n, double alpha,
  double** Xold, double** Xnew
) {
  double error = 0;
  for (int i=1; i <= m; i++) {
    for (int j=1; j <= n; j++) {
       Xnew[i][j] = Xold[i][j] + alpha *
                    (  Xold[i][j-1] + Xold[i][j+1] +
                       Xold[i-1][j] + Xold[i+1][j]
                       - 4.0 * Xold[i][j] );
       double diff = fabs(Xnew[i][j] - Xold[i][j]);
       if (diff > error)
         error = diff;
    }
  }
  return error;
}
```

This code can be parallelised using a row-wise or column-wise partition.

Heat diffusion – iteration

```
for (i = 0; i < cfg.maxIter && error >= cfg.errorThreshold; i++) {
  // Computation step.
  error = heat_diffusion(cfg.N, cfg.N, cfg.alpha, Xold, Xnew);
  // Swap buffers for next iteration (avoid costly copying).
  double** tmp = Xold;
  Xold = Xnew;
  Xnew = tmp;
}
```

For parallelisation we also need to:

- Reduce the error (take the maximum) calculated by each thread, as in the dot product example.
- Ensure that threads are always synchronised in the same iteration.

Heat diffusion – parallelisation (part 1)

```
typedef struct {
  double **Xold, **Xnew;
  double global_error;
  pthread_mutex_t mutex;
  ...
} shared_data_t;
shared_data_t sd;
void* do_work(void * arg) {
  ...
  for (int iter = 0;  iter < cfg.maxIter
                && sd.global_error >= cfg.errorThreshold; iter++) {
    double my_error = heat_diffusion(my_N, cfg.N, cfg.alpha,
                                     my_Xold, my_Xnew);
    pthread_mutex_lock(& (sd.mutex));
    if (my_error > sd.global_error) sd.global_error = my_error;
    pthread_mutex_unlock(& (sd.mutex));
    ...
  }
```

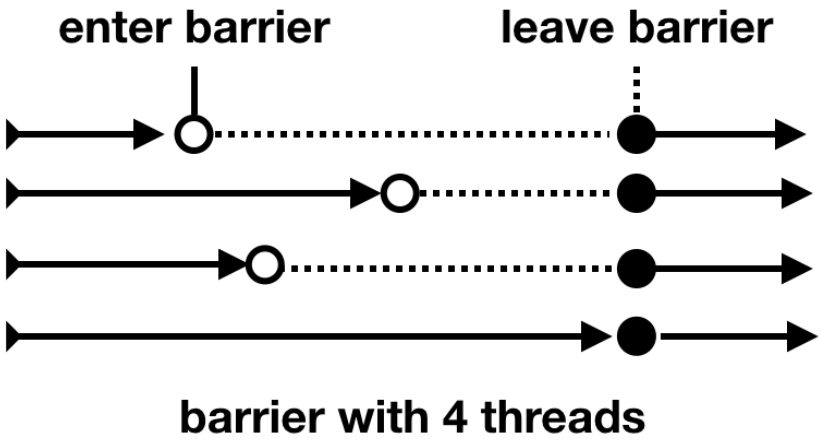
Similar strategy to the dot product algorithm.

Heat diffusion – parallelisation (part 2)

```
typedef struct {
  ...
  pthread_barrier_t barrier;
} shared_data_t;
shared_data_t sd;
void* do_work(void * arg) {
  ...
  for (int iter = 0;  iter < cfg.maxIter
                && sd.global_error >= cfg.errorThreshold; iter++) {
    ...
    pthread_barrier_wait( & (sd.barrier) );
    double** tmp = my_Xold;
    my_Xold = my_Xnew;
    my_Xnew = tmp;
  }
```

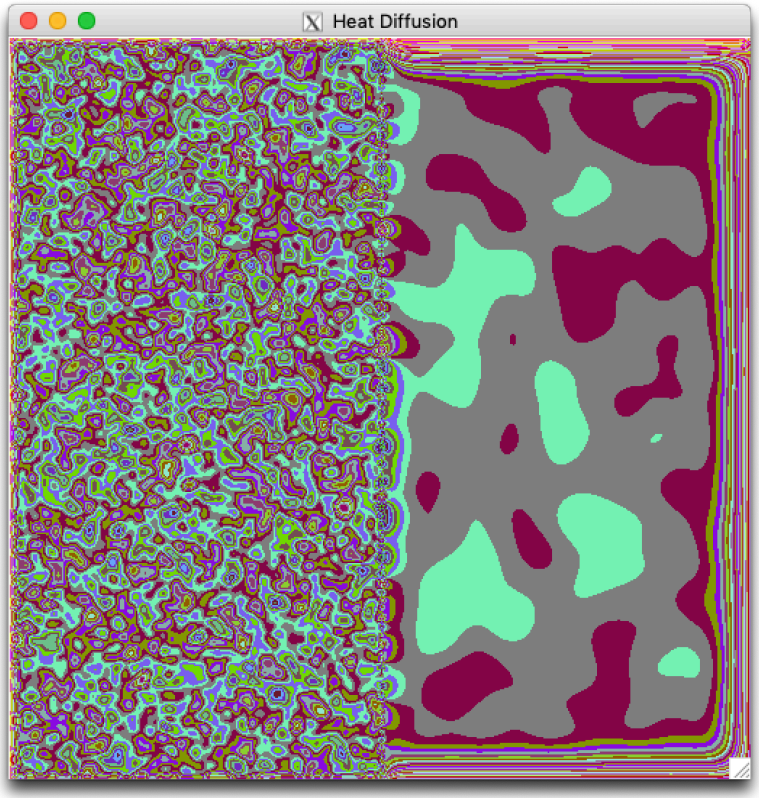
To ensure threads are synchronised in the same iteration, we employ a **barrier**. (Do you remember `MPI_Barrier` ?)

Barriers



A **barrier** is a primitive that blocks all participants at a synchronization point until all participants reach that point.

Heat diffusion



If we remove the call to **`pthread_barrier_when`**, then the threads become out-of-sync, as illustrated by the above screenshot for 2 threads.

Pthreads support for barriers

Barrier operations[3]:

```
int pthread_barrier_init(pthread_barrier_t *barrier,
   pthread_barrierattr_t *restrict attr, unsigned count);
int pthread_wait(pthread_barrier_t *barrier);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- **`pthread_barrier_init(b,attr,count)`** initialises **`b`** for **`count`** participants.
- **`pthread_barrier_wait(b)`** blocks the calling thread on barrier **`b`**, until all participants reach the barrier. The barrier is reset for re-use on return.
- **`pthread_barrier_destroy(b)`** tears-down **`b`**.

[3] Barriers are implemented only in Pthreads implementations that enable the "Barriers option", e.g., they are not defined in standard Linux or MacOs distributions. We'll use an alternative implementation of barriers using **condition variables** (discussed next).

Waiting on conditions

```
int do_wait = 1;
while (do_wait) {
  pthread_mutex_lock(mutex);
  if (some_condition())
    do_wait = 0;
  pthread_mutex_unlock(mutex);
}
```

- In many a cases a thread has to wait for a certain synchronisation condition to hold. Using a mutex we can repeatedly test for the condition, as illustrated above. This is a costly "busy-wait" scheme and may also generate high contention between threads, due to repeated lock acquisition and release.
- Ideally, we would like that the thread suspends execution and relinquishes the lock until the condition holds. This type of synchronisation is provided by **condition variables**, also known as **monitors**.

Condition variables in Pthreads

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
```

- `pthread_cond_init` / `pthread_cond_destroy` are used for initialization / tear-down (as usual).
- `pthread_cond_wait(cond,mutex)`: waits on `cond`, releasing the lock on `mutex` while blocked, and re-acquiring it when unblocked.
- `pthread_cond_signal(cond)` unblocks **one** thread waiting on `cond`.
- `pthread_cond_broadcast(cond)` unblocks **all** threads waiting on `cond`.

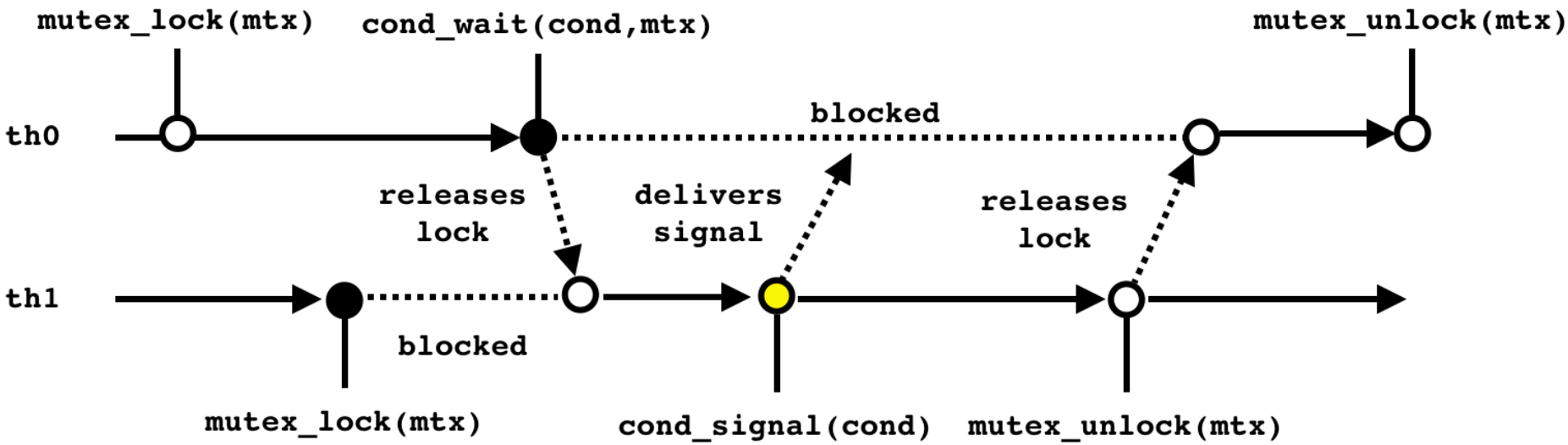
Condition variables – use

```
// Waiting thread
pthread_mutex_lock(&mutex);
while (! some_condition())
  pthread_cond_wait(&cond, &mutex);
pthread_mutex_unlock(&mutex)

// Signalling thread
pthread_mutex_lock(&mutex);
if (some_condition() )
  pthread_cond_signal(&cond); // or broadcast
pthread_mutex_unlock(&mutex);
```

- A loop is required in combination with `pthread_cond_wait` due to (rare but) possible **spurious wakeups**.
- A condition variable is used associates to a single mutex. A thread that calls `pthread_cond_wait / signal / broadcast` must have a lock on the mutex.
- Multiple condition variables may be associated to the same mutex.

Condition variables – illustration



A call to `pthread_cond_wait` proceeds in the following steps:

1. releases the mutex and suspends execution
2. waits for a signal
3. waits to re-acquire the mutex before returning

Using condition variables – blocking queue example

```
typedef struct {
  unsigned capacity, size, head;
  void** data;
  pthread_mutex_t mutex;
  pthread_cond_t not_empty, not_full;
} bqueue_t;
void bqueue_init(bqueue_t* q, unsigned capacity);
void bqueue_add(bqueue_t* q, void* v);
void* bqueue_remove(bqueue_t* q);
void bqueue_destroy(bqueue_t* q, unsigned capacity);
```

A fixed-capacity blocking queue is a data structure commonly used in multithreaded programs, e.g., may be used to implement task pools. The intended semantics are:

- **`bqueue_add(q,v)`** adds `v` to the end of `q`, blocking if `q` is full.
- **`bqueue_remove(q)`** removes head of `q`, blocking if `q` is empty.
- We will use a circular buffer for elements (**`data`**). The use of condition variables would be similar for other representations.

Implementing a blocking FIFO queue (2)

```
void bqueue_add(bqueue_t* q, void* item) {
  pthread_mutex_lock( & (q -> mutex));
  while (q -> size == q -> capacity) // block if full
    pthread_cond_wait( & (q -> not_full), & (q->mutex) );
  q -> data[ (q -> head + q -> size) % q -> capacity ] = item;
  q -> size ++;
  pthread_cond_signal(& (q -> not_empty)); // signal not_empty
  pthread_mutex_unlock(& (q -> mutex));
}
```

Implementing a blocking FIFO queue (3)

```
void* bqueue_remove(bqueue_t* q) {
  void* item;
  pthread_mutex_lock( & (q -> mutex));
  while (q -> size == 0) // block if empty
    pthread_cond_wait( & (q -> not_empty), & (q->mutex) );
  item = q -> data [q -> head];
  q -> head = (q -> head + 1) % q -> capacity;
  q -> size --;
  pthread_cond_signal(& (q -> not_full)); // signal not_full
  pthread_mutex_unlock( & (q -> mutex));
  return item;
}
```

Implementing pthread_barrier_t

```
typedef struct {
  unsigned size, in, out;
  pthread_mutex_t mutex;
  pthread_cond_t all_in, all_out;
} pthread_barrier_t;
int pthread_barrier_init(pthread_barrier_t* b, unsigned count) {
  b -> count = count;
  b -> in = 0;
  b -> out = 0;
  pthread_mutex_init(&(barrier -> mutex), NULL);
  pthread_cond_init(&(barrier -> all_in), NULL);
  pthread_cond_init(&(barrier -> all_out), NULL);
  return 0;
}
```

Given that pthreads barriers are not defined in standard Linux or MacOs distributions, we may define our own (relatively simple) implementation of `pthread_barrier_t` using condition variables.

Implementing pthread_barrier_t (2)

```
int pthread_barrier_wait(pthread_barrier_t* b) {
  pthread_mutex_lock(& (b -> mutex) );
  b -> in ++;
  if (b -> in == b -> count)
    pthread_cond_broadcast( & (b -> all_in) ); // all in
  else while (b -> in != b -> count)
    pthread_cond_wait( & (b -> all_in), & (b -> mutex) );
  b -> out ++;
  ...
  return 0;
}
```

- b −> in == b −> count ⟶ all threads reached the barrier.
- b −> out == b −> count ⟶ all threads are ready to leave the barrier (not blocked on all_in).

Implementing pthread_barrier_t (3)

```
int pthread_barrier_wait(pthread_barrier_t* b) {
  pthread_mutex_lock(& (b -> mutex) );
  b -> in ++;
  if (b -> in == b -> count)
    pthread_cond_broadcast( & (b -> all_in) ); // all in
  else while (b -> in != b -> count)
    pthread_cond_wait( & (b -> all_in), & (b -> mutex) );
  b -> out ++;
  if (b -> out == b -> count) {
    b -> in = 0; // reinitialise barrier for next round
    b -> out = 0;
    pthread_cond_broadcast( & (b -> all_out) ); // all out
  } else while (b -> out != 0)
      pthread_cond_wait( & (b -> all_out), & (b -> mutex) );
  pthread_mutex_unlock(& ( b -> mutex ));
  return 0;
}
```

- out/all_out are used to guarantee threads exit the barrier orderly. Note that otherwise a thread could leave and re-enter the barrier while others are still blocked on all_in .