Thread Management (join)

int pthread_join (pthread_t thread,
void **value_ptr**)** **[OUT] thread return value**

int pthread_detach (pthread_t thread**)**

int pthread_attr_setdetachstate (pthread_attr_t *attr, **[IN/OUT] attribute**
int detachstate**)** **[IN] state to be set**

int pthread_attr_getdetachstate (const pthread_attr_t *attr,
int *detachstate**)** **[OUT] detach state value**

Join blocks the calling thread until the target thread terminates, and returns it's thread_exit argument.

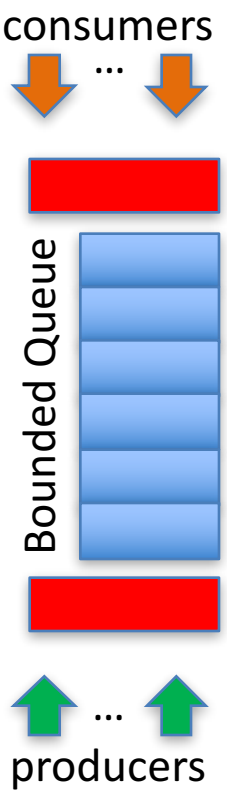
It is impossible to join a thread in a detached state. It is also impossible to reattach it

Thread Management (state)

int pthread_attr_getstacksize **(**const pthread_attr_t *restrict attr,
size_t *restrict stacksize**)**
int pthread_attr_setstacksize **(**pthread_attr_t *attr, size_t stacksize**)**
int pthread_attr_getstackaddr **(**const pthread_attr_t *restrict attr,
void **restrict stackaddr**)**
int pthread_attr_setstackaddr **(**pthread_attr_t *attr, void *stackaddr**)**
pthread_t pthread_self **(**void**)**
int pthread_equal **(**pthread_t t1, pthread_t t2**)**

The POSIX standard does not dictate the size of a thread's stack !

Example: A Producer – Consumer Queue



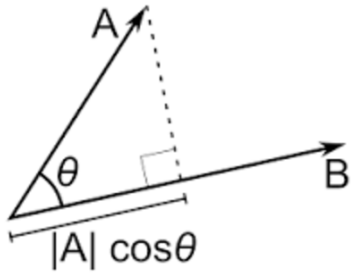
- We have a bounded queue where producers store their output and from where consumers take their input
- Protect the structure against intensive unnecessary accesses
  - Detect boundary conditions: queue empty and queue full

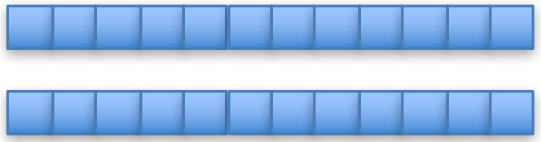
Example: dot-product

Scalar product, inner product

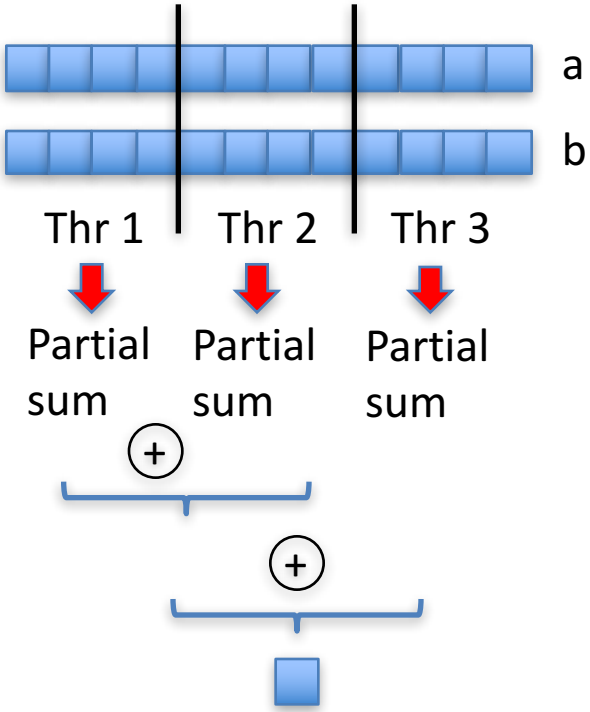
In mathematics, the **dot product** or scalar **product** (sometimes **inner product** in the context of Euclidean space, or rarely projection **product** for emphasizing the geometric significance), is an algebraic operation that takes two equal-length sequences of numbers (usually coordinate vectors) and returns a single number.

Dot product - Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/Dot_product Wikipedia





$$= \sum_{n=1}^{N} a_i . b_i$$



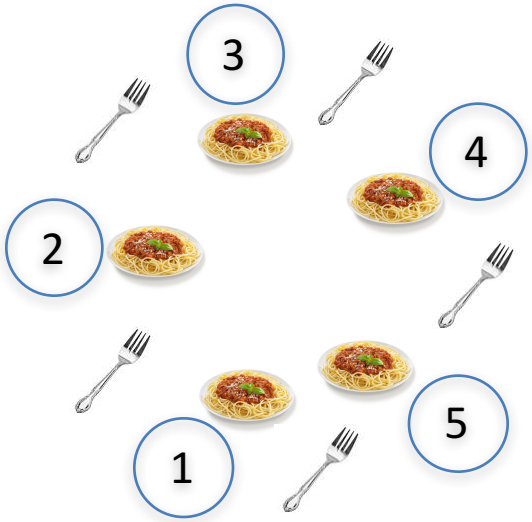
- Divide the arrays between participants to load-balance the work
  - Each will then compute a partial sum
- Add all the partial sums together for the final result (reduce operation)
- Technical details: cost of managing the threads vs. cost of the algorithm? How to minimize the management cost?

Homework: the dining philosophers problem

Dijkstra, 1965: Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to others. A philosopher can take the fork on his right or the one on his left as they become available, but cannot start eating before getting both of them.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.



Deliver in 2 weeks (Friday 09/09): A pdf document with the description of the problem as you understand it and with the solution you choose to implement. Implement a solution for an unbounded number of philosophers, where each philosopher is implemented as a thread, and the forks are the synchronizations needed between them. Provide the C source code and a makefile, allowing for smooth compilation and execution.