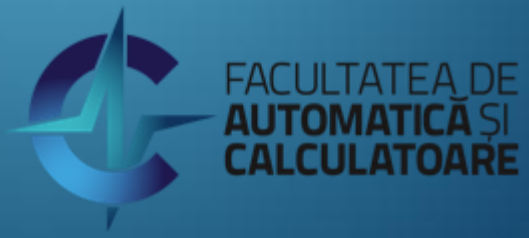
Arhitecturi Paralele
Pthread + Primitive sincronizare

Lect. Dr. Ing. Cristian Chilipirea
cristian.chilipirea@mta.ro

Curs susținut în parteneriat cu Prof. Florin Pop

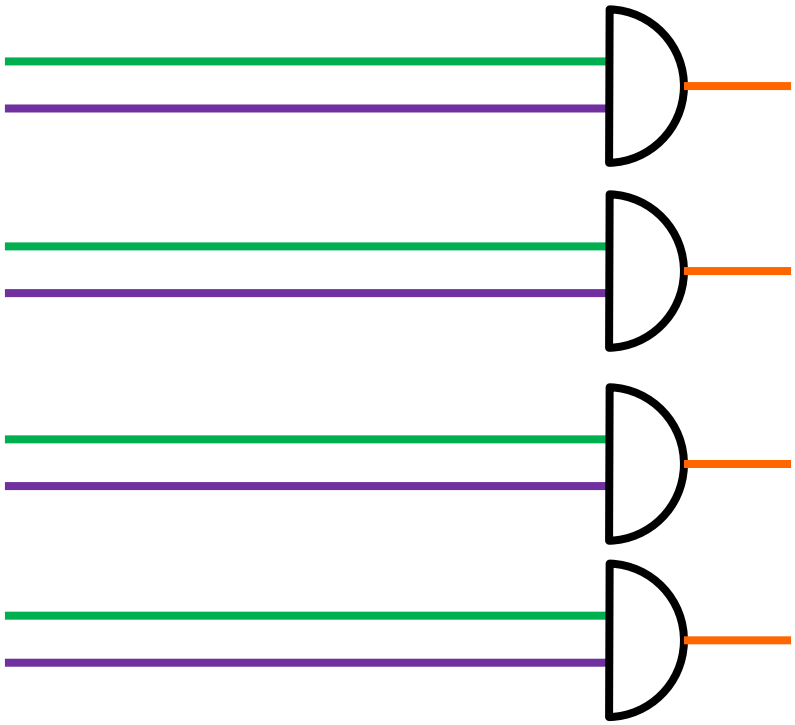


Tipuri de paralelism

- La nivel de bit (Bit level)
- La nivel de instrucțiune (Instruction level)
- La nivel de task (Task parallelism)

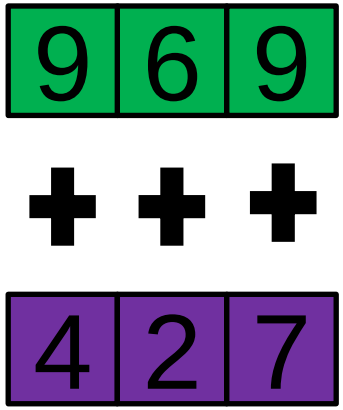
Tipuri de paralelism

- Bit level
  - C = A & B
- Instruction level
- Task parallelism



Tipuri de paralelism

- Bit level
- Instruction level
  - C[] = A[] + B[]
- Task parallelism



Adunarea a doi vectori

Tipuri de paralelism

- Bit level
- Instruction level
- Task parallelism

load Z		A = B + C
store H	.	store A
H=H*100	.	G[1] = G[2] + G[3]
H++	.	store G[1]
store H		G[4] = G[5] + G[6]

  - **Multi-Tasking (pot comunica și procesele)**
  - **Multi-Threading**

Tipuri de paralelism

- Bit level
- Instruction level
- Task parallelism
  - **Multi-Tasking (pot comunica și procesele)**
  - **Multi-Threading**

Cum pot comunica două procese?

Notații pseudocod

```
co S1 || S2 || ... || Sn oc
```

Ex.1:

```
x=0; y=0;
co x=x+1 || y=y+1 oc
z=x+y;
```

Notații pseudocod

```
co [cuantificator]{Sj}
```

Ex. 2:

```
    co [j=1 to n] {a[j]=0; b[j]=0;}
```

POSIX threads

POSIX threads

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{

}

pthread_join(thread, NULL);
```

Compilare pthread

gcc –o executabil cod.c –lpthread –lrt

#include<pthread.h>

#include<semaphore.h>

Pthread

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{

}

pthread_join(thread, NULL);
```

Acest element reprezintă thread-ul.
Este un thread handle

Pthread

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{
}

pthread_join(thread, NULL);
```

Prin acest parametru am putea să facem recomandări sistemului de operare.
Ex: să folosească anumite core-uri.

Pthread

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{
        Când se crează thread-ul va porni de
        la funcția dată ca parametru.
}

pthread_join(thread, NULL);
```

Pthread

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{
        Așa trimitem date thread-ului
}

pthread_join(thread, NULL);
```

Pthread

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{
        Astfel se pot extrage date din
        thread
}

pthread_join(thread, NULL);
```

Pthread

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
void* threadFunction(void *arg)
{
   a=a+1;
   return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
     pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
      pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Câte thread-uri avem în execuție?

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

Câte thread-uri avem în execuție?

3

main

threadFunction

threadFunction

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

Pthread

Spunem că thread-ul main **așteaptă** ca thread-ul cu handler-ul tid[i] să termine execuția

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
void* threadFunction(void *arg)
{
   a=a+1;
   return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
     pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
      pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
void* threadFunction(void *arg)
{
   a=a+1;
   return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
     pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
      pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Race condition

Thread 1	Thread 2
a = a + 2	**a = a + 2**

Care este valoare lui a?

Race condition

a = 0

Thread 1	Thread 2
a = a + 2	**a = a + 2**

Care este valoare lui a?

Race condition

a = 0

Thread 1	Thread 2
a = a + 2	**a = a + 2**

Care este valoare lui a?

4

Race condition

a = 0

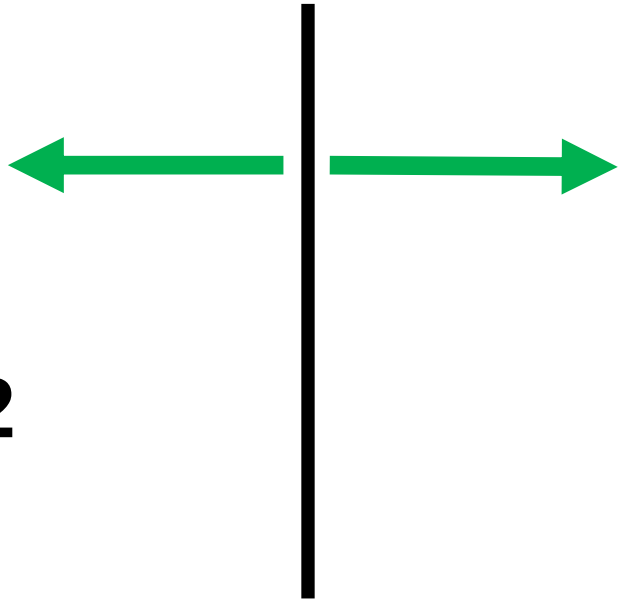
Thread 1	Thread 2
a = a + 2	**a = a + 2**

Care este valoare lui a?

4 **ȘI** 2

Race condition

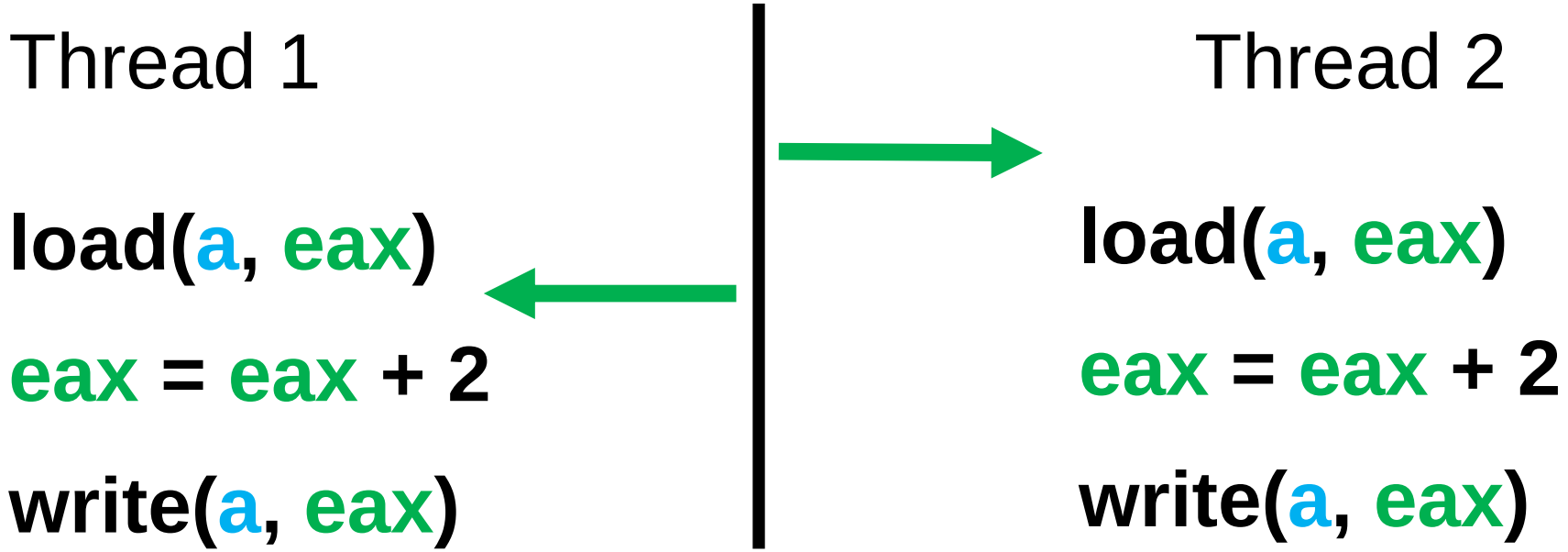
a = 0



Thread 1	Thread 2
load(a, eax)	load(a, eax)
eax = eax + 2	eax = eax + 2
write(a, eax)	write(a, eax)
eax =	eax =

Race condition

a = 0

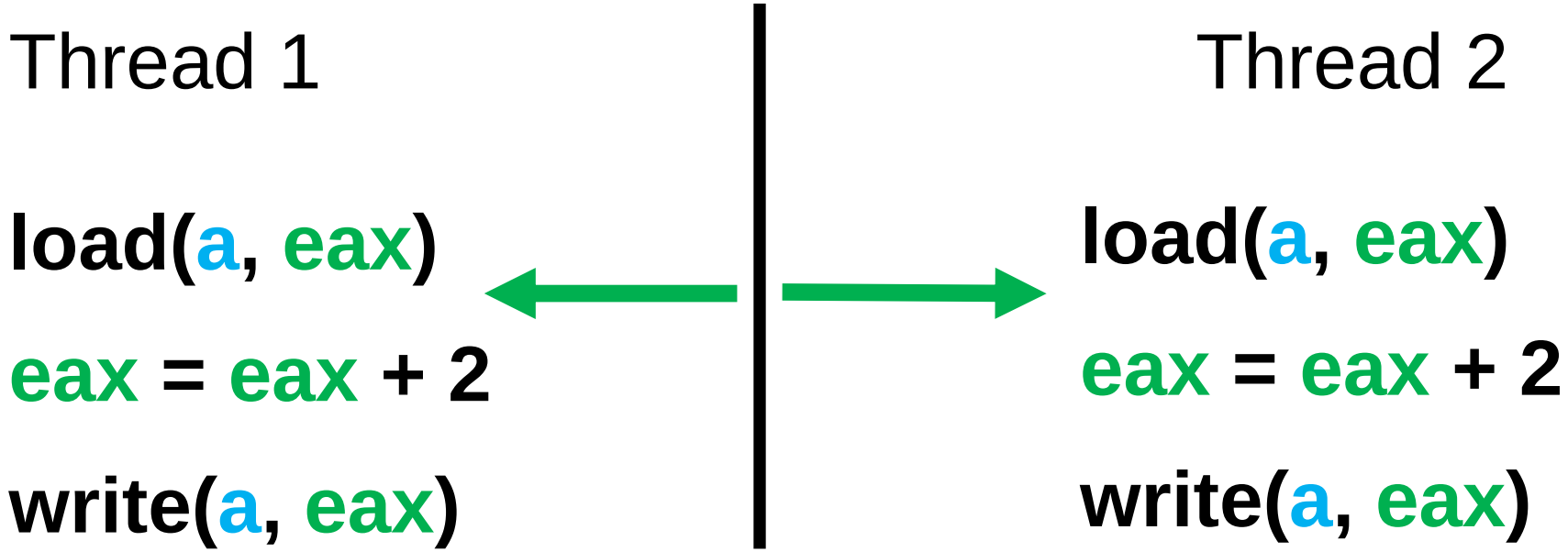


eax = 0

eax =

Race condition

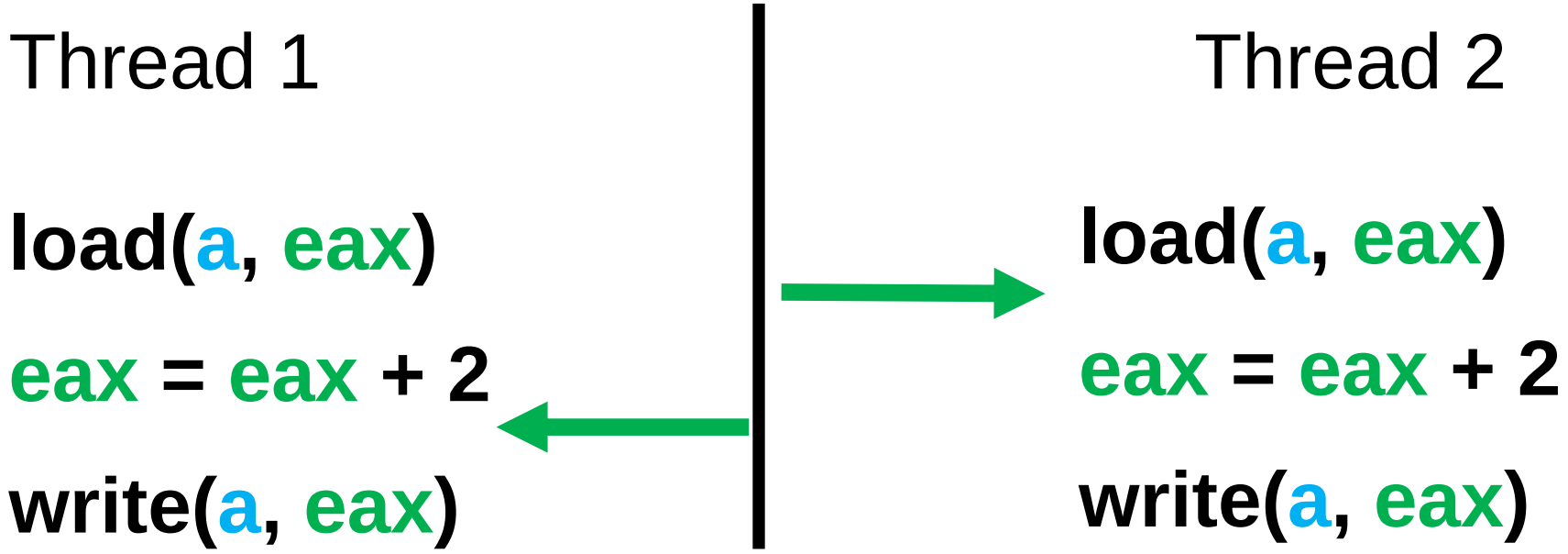
a = 0



eax = 0 **eax = 0**

Race condition

a = 0



eax = 2

eax = 0

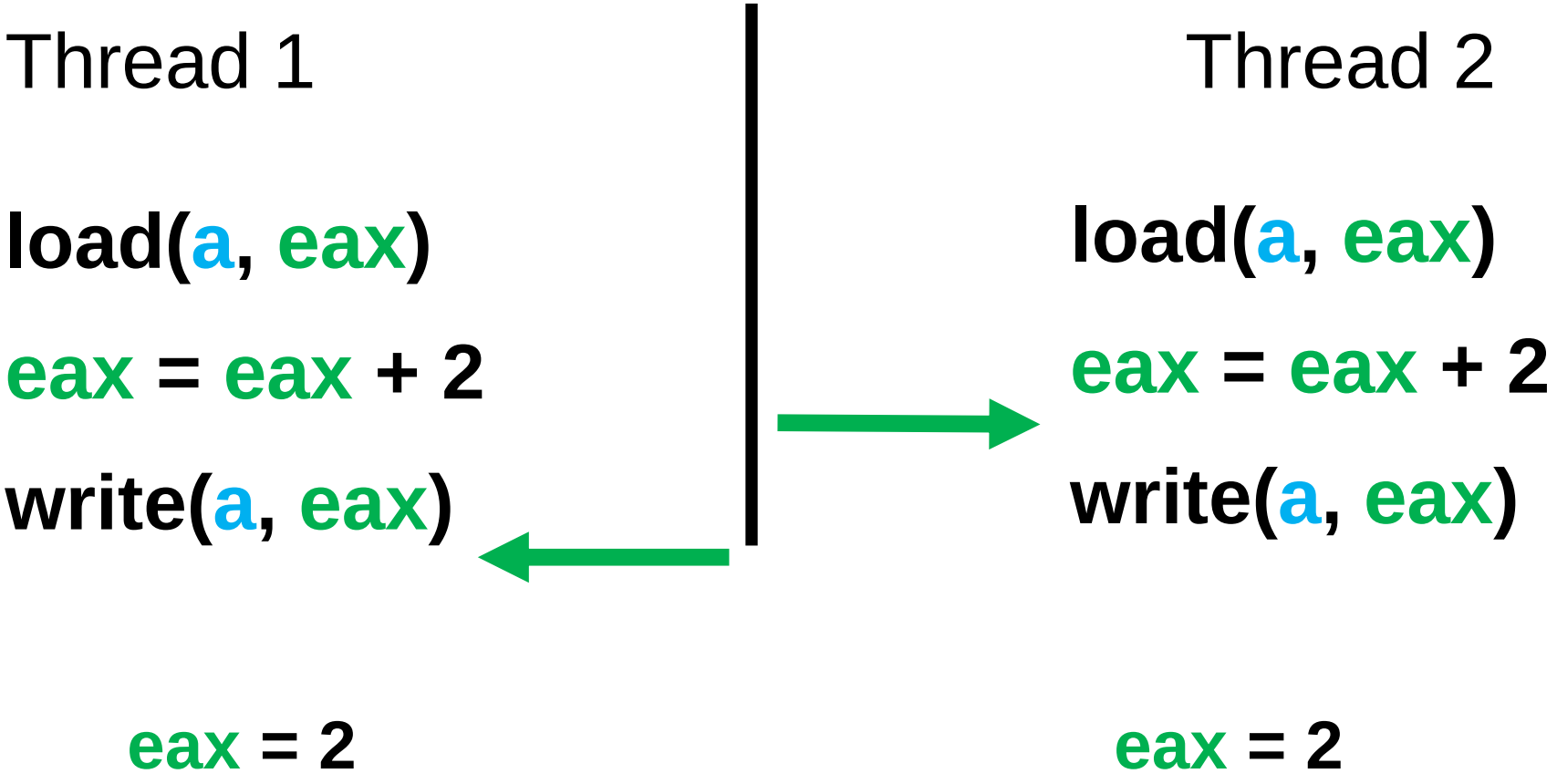
Race condition

a = 0

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax = 2	**eax = 2**

Race condition

a = 2



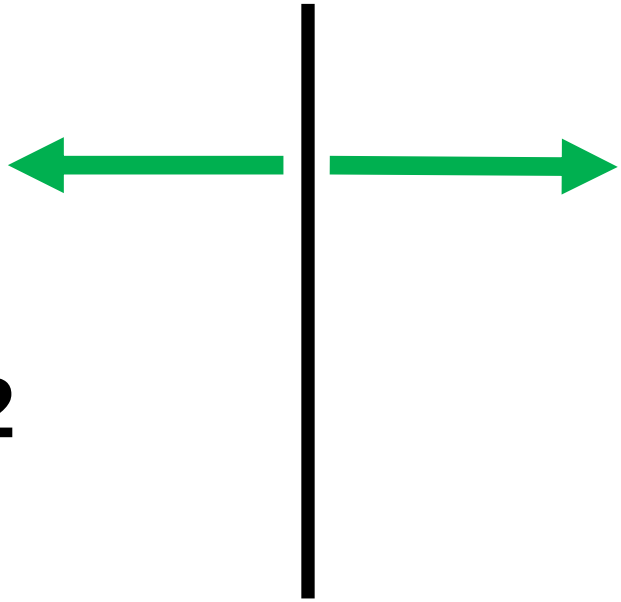
Race condition

a = 2

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax = 2	**eax = 2**

Race condition

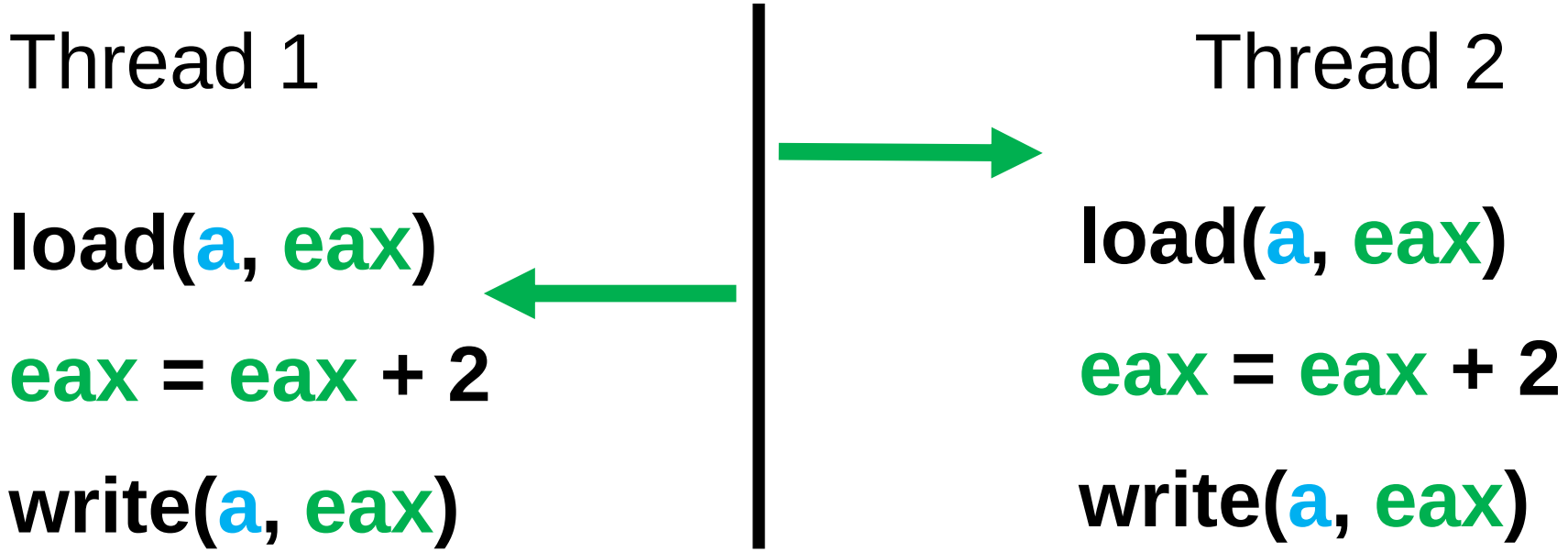
a = 0



Thread 1	Thread 2
load(a, eax)	load(a, eax)
eax = eax + 2	eax = eax + 2
write(a, eax)	write(a, eax)
eax =	eax =

Race condition

a = 0

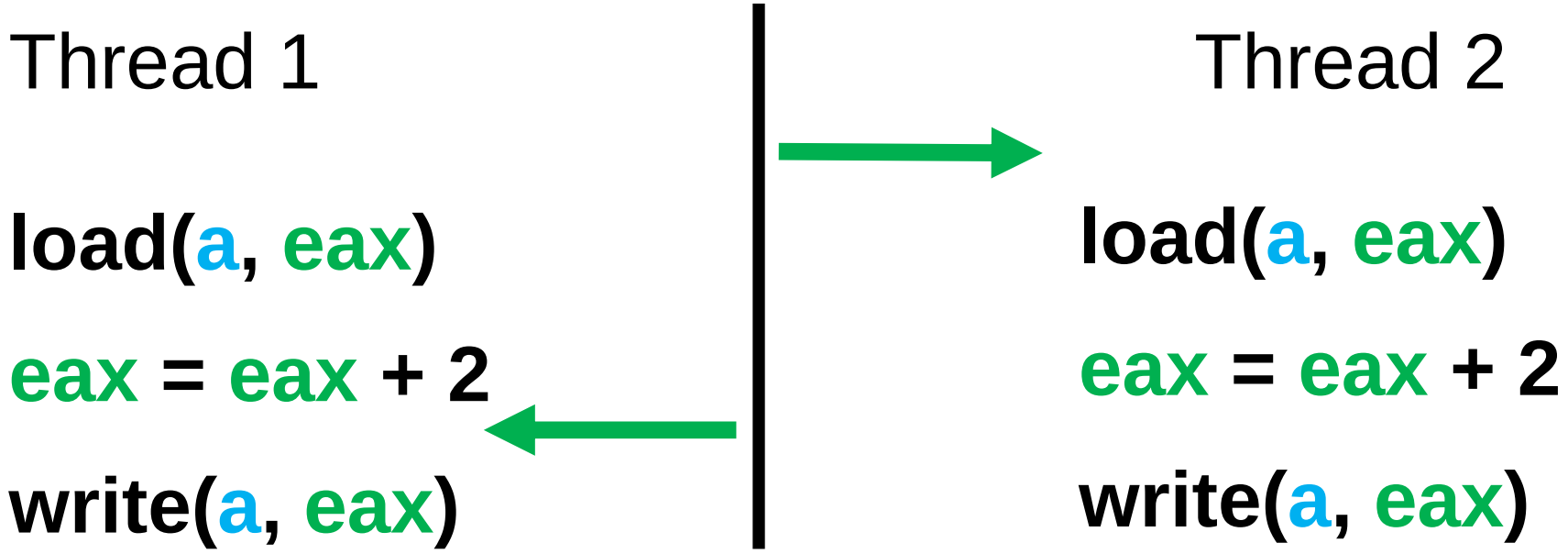


eax = 0

eax =

Race condition

a = 0

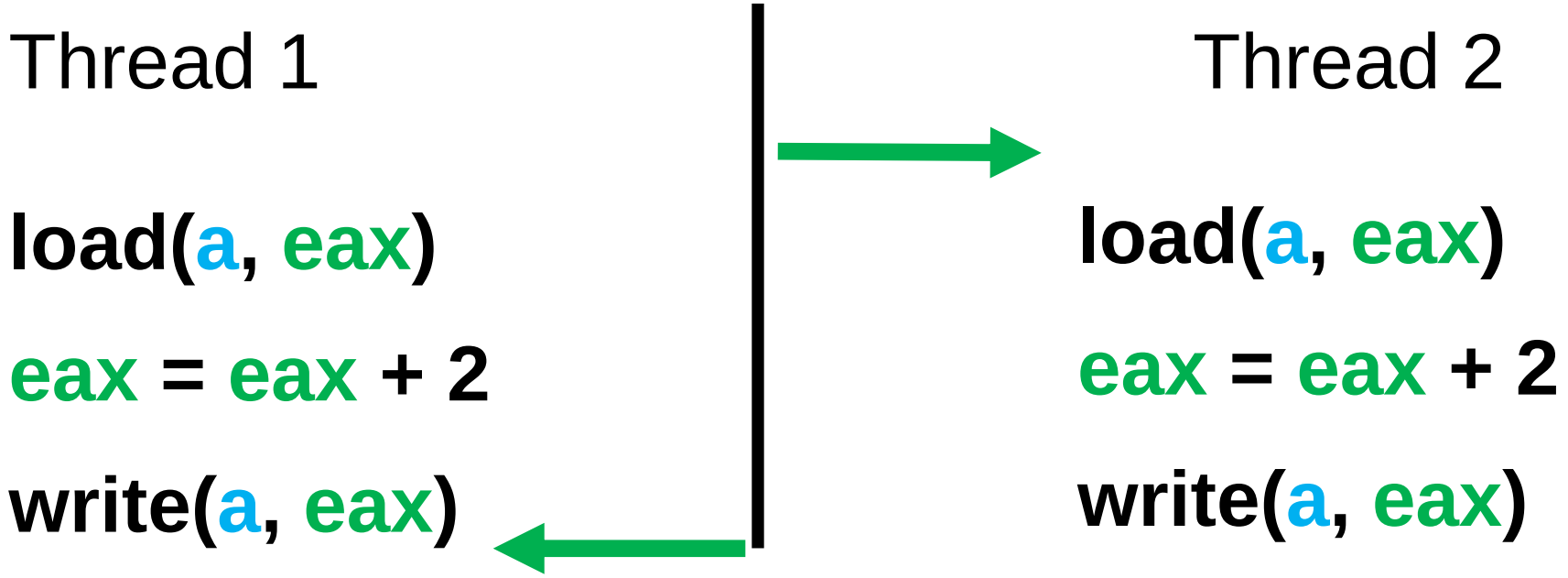


eax = 2

eax =

Race condition

a = 2

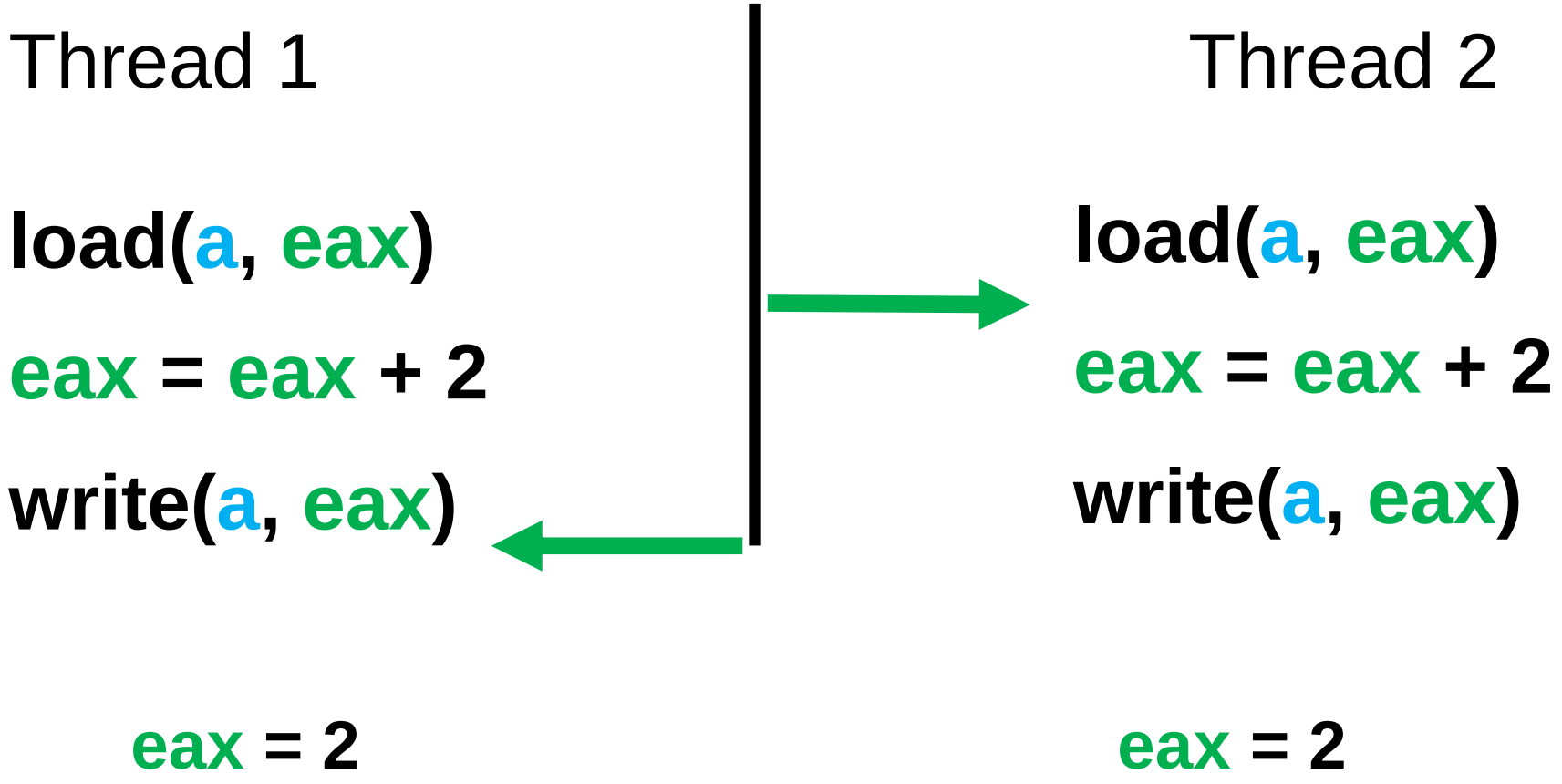


eax = 2

eax =

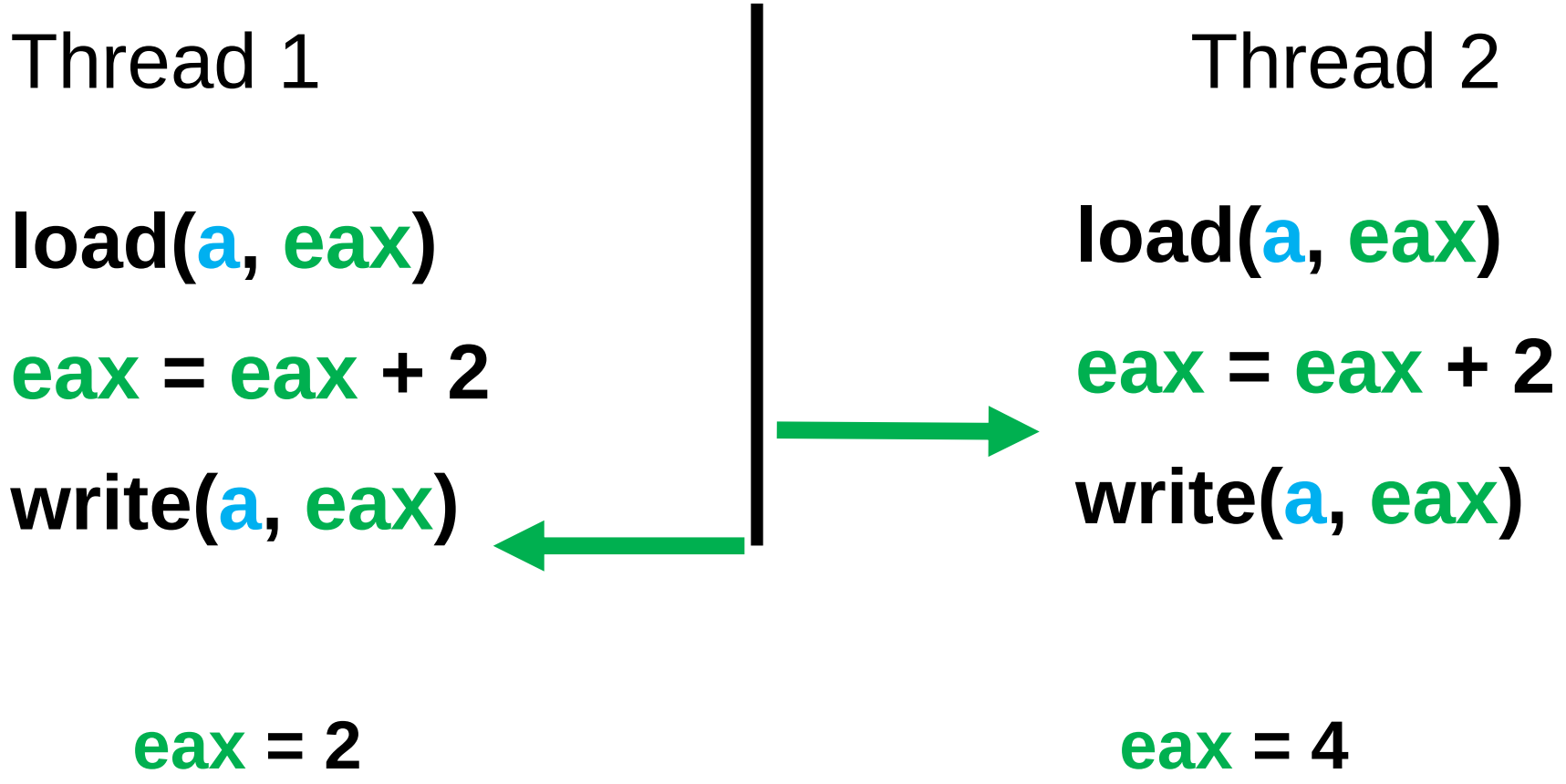
Race condition

a = 2



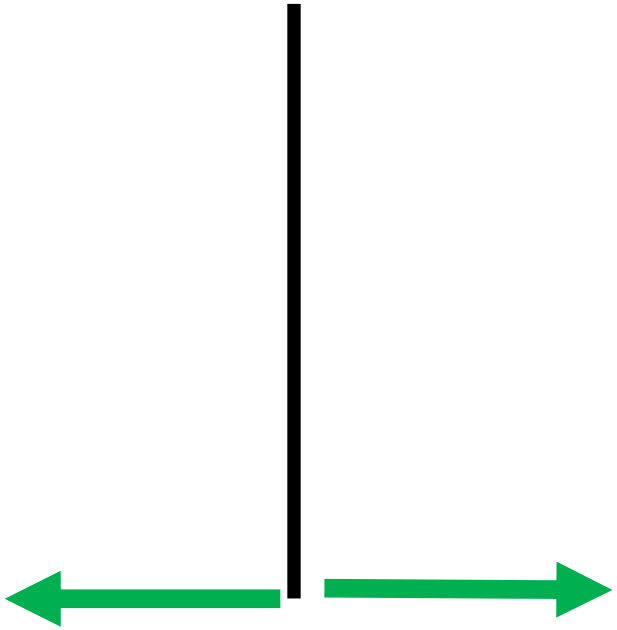
Race condition

a = 2



Race condition

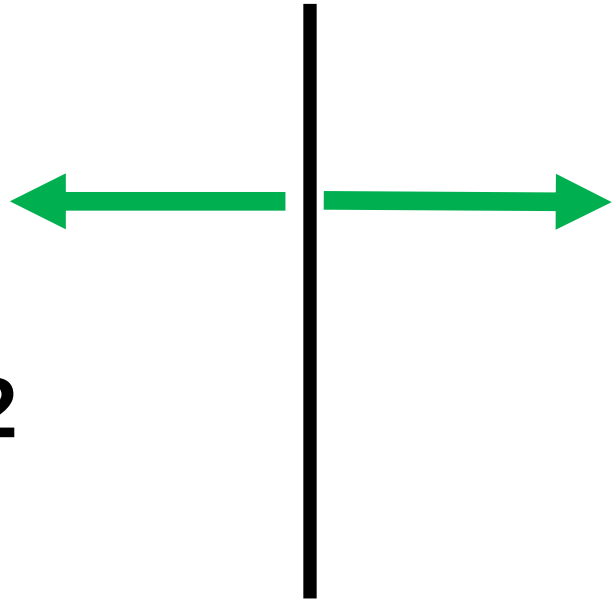
a = 4



Thread 1	Thread 2
load(a, eax)	load(a, eax)
eax = eax + 2	eax = eax + 2
write(a, eax)	write(a, eax)
eax = 2	eax = 4

Race condition

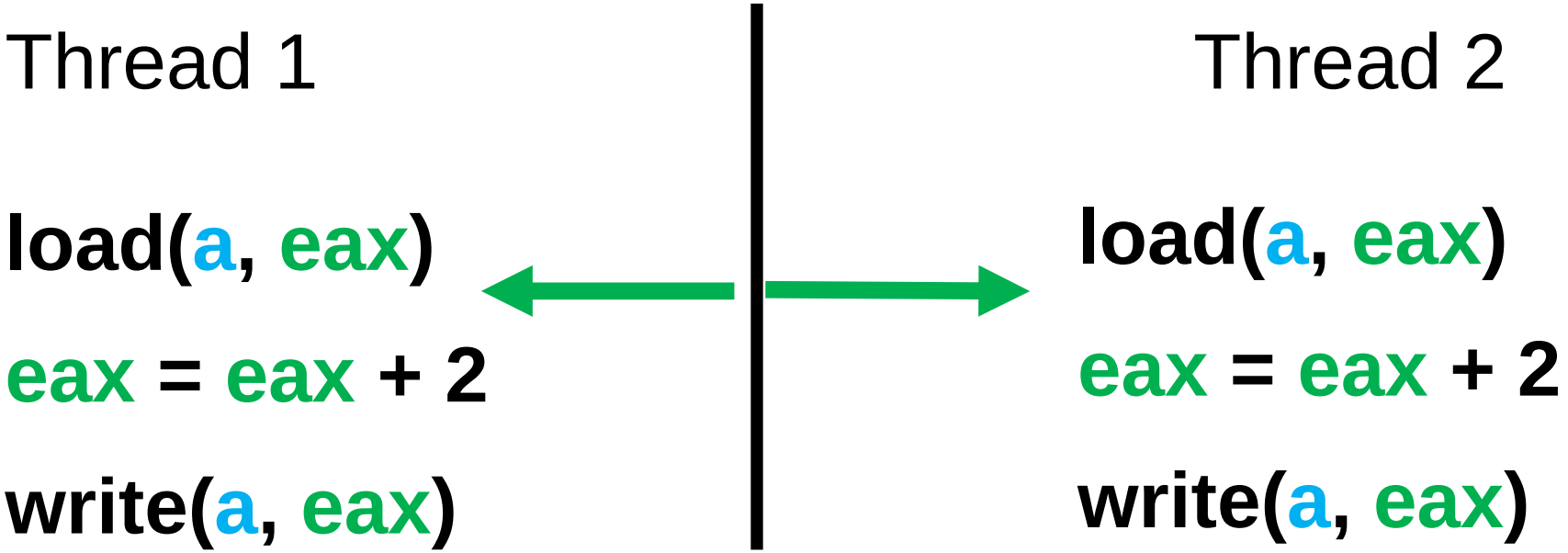
a = 0



Thread 1	Thread 2
load(a, eax)	load(a, eax)
eax = eax + 2	eax = eax + 2
write(a, eax)	write(a, eax)
eax =	eax =

Race condition

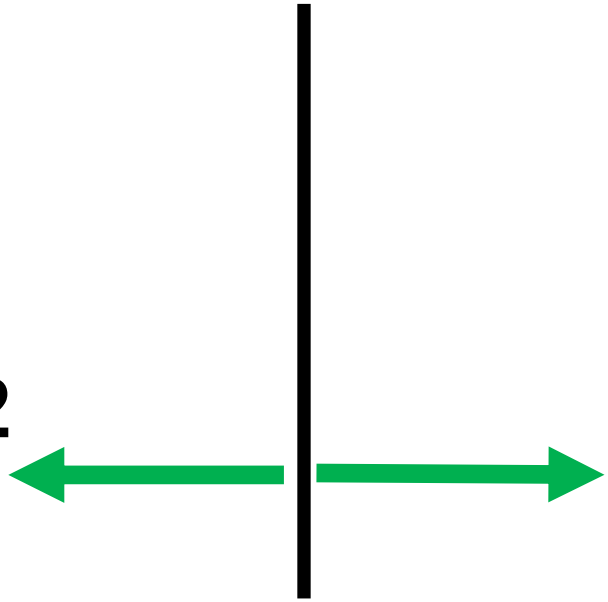
a = 0



eax = 0 **eax = 0**

Race condition

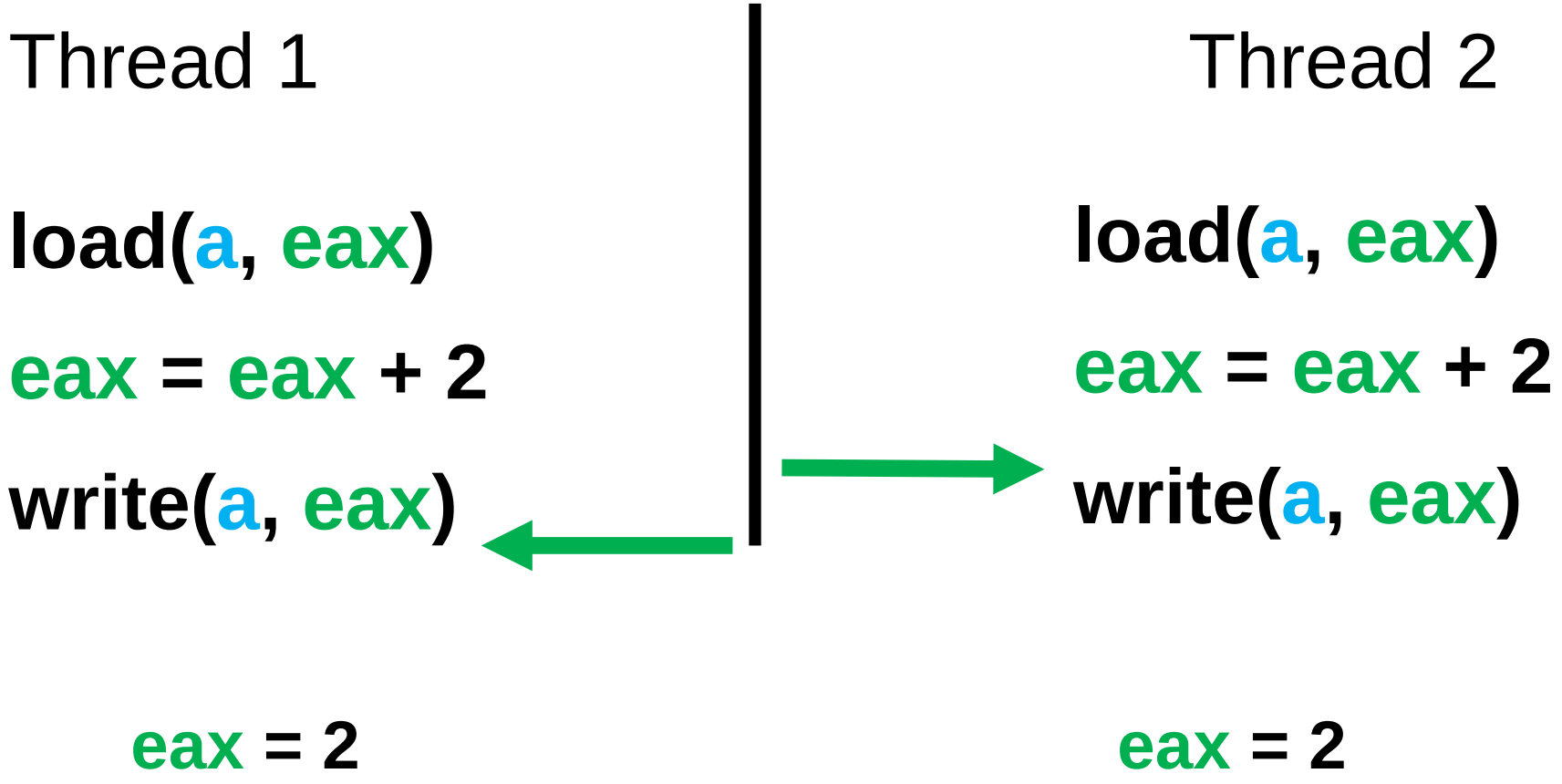
a = 0



Thread 1	Thread 2
load(a, eax)	load(a, eax)
eax = eax + 2	eax = eax + 2
write(a, eax)	write(a, eax)
eax = 2	eax = 2

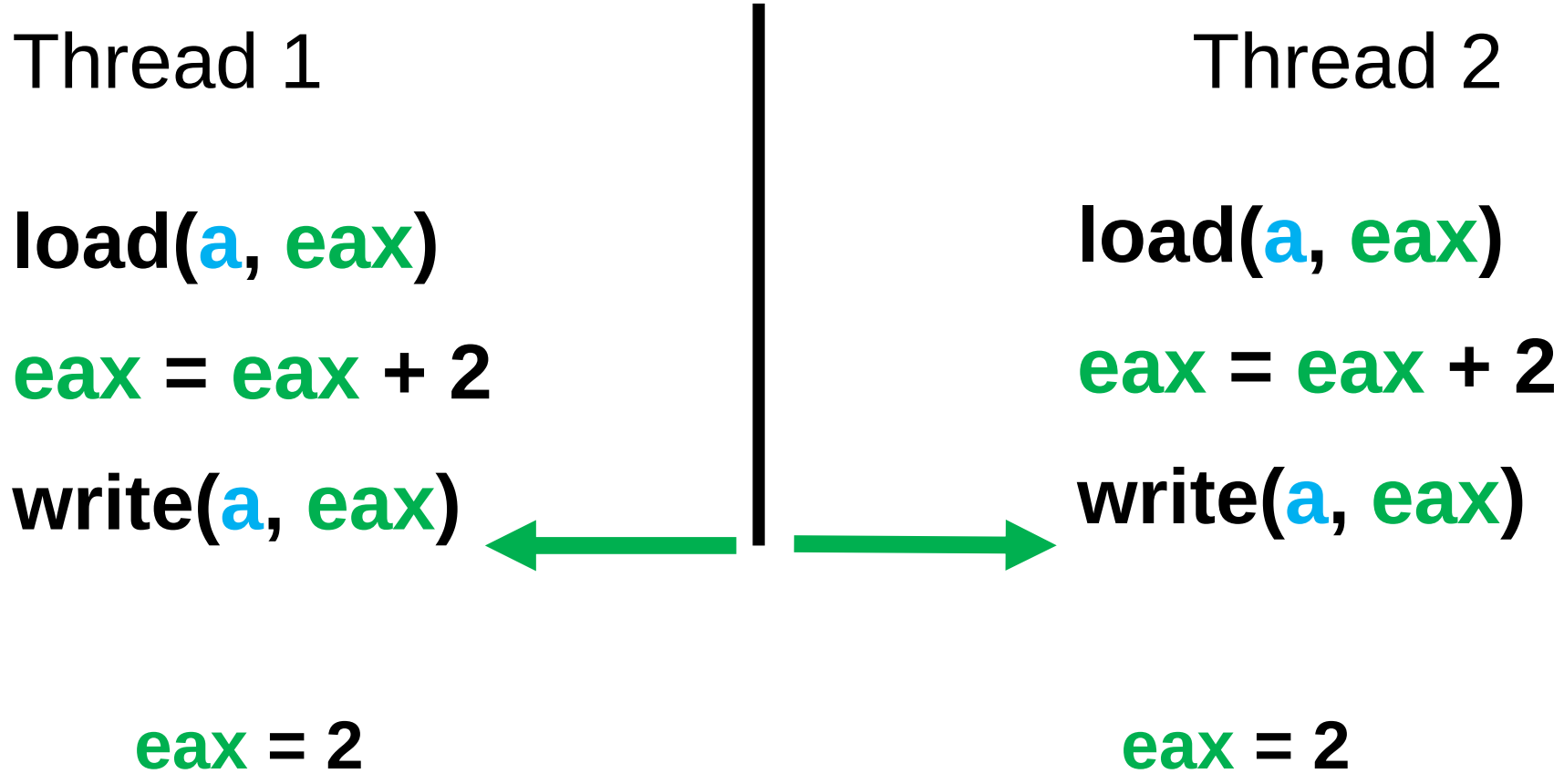
Race condition

a = 2



Race condition

a = 2



Race condition

a = 2

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**

eax = 2 **CREW** **eax = 2**

Primitive de sincronizare
Synchronization primitives

- Atomics — Atomice
- Semaphore — Semafoare
  - Binary semaphore (Mutex)
  - Critical section
- Barrier — Bariere

Atomics

- Fie variabile de 64 biți pe un procesor 64 biți **C = A + B**

```
load(A, eax)
load(B, ebx)
eax = eax + ebx
write(C, eax)
```

Atomics

- Fie variabile de 64 biți pe un procesor 32 biți **C = A + B**

load(A[0], eax)

load(B[0], ebx)

eax = eax + ebx

write(C[0], eax)

load(A[1], eax)

load(B[1], ebx)

eax = eax + ebx

write(C[1], eax)

Atomics

- Fie variabile de 64 biți pe un procesor 32 biți **C = A + B**

```
load(A[0], eax)
load(B[0], ebx)
eax = eax + ebx
write(C[0], eax)
load(A[1], eax)
load(B[1], ebx)
eax = eax + ebx
write(C[1], eax)
```


Putem avea doar jumătate de **C** modificat

Atomics

- Fie variabile de 64 biți pe un procesor 32 biți **C = A + B**

```
load(A[0], eax)
load(B[0], ebx)
eax = eax + ebx
write(C[0], eax)
load(A[1], eax)
load(B[1], ebx)
eax = eax + ebx
write(C[1], eax)
```

Atomicitatea asigură că **C** va fi vizibil doar complet modificat, sau complet nemodificat.

Excludere mutuală
Mutual exclusion - mutex

Mutual exclusion - Dekker's solution

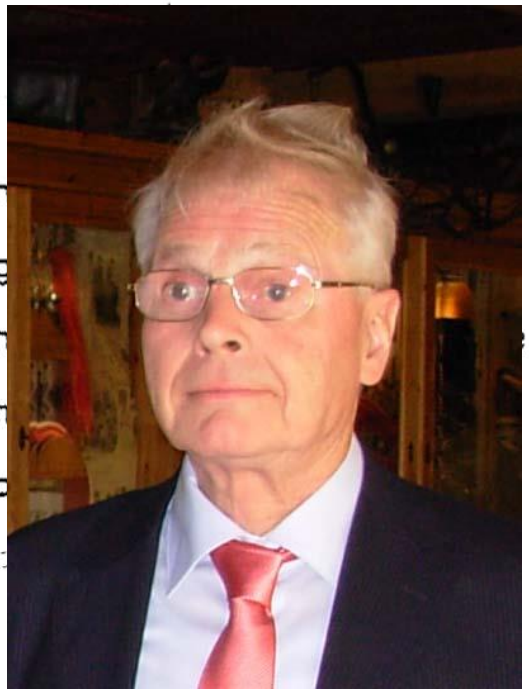
EWD35 - 1

EWD35.htm

About the sequentiality of process descriptions.

Over de sequentialiteit van procesbeschrijvingen.

Het is niet ongebruikelijk, wanneer een spreker zijn
een inleiding. Omdat ik mij hier mogelijk richt tot een g
minder vertrouwd is met de problematiek, die ik wil aansn
die ik zal moeten gebruiken, wilde ik in dit geval ter in
inleidingen houden, nl. een om de achtergrond van de prob
een tweede, om U een gevoel te geven voor de aard van de
wij tegen het lijf zullen lopen.



Theodorus (Dirk) J. Dekker

Mutual exclusion – Dijsktra's Solution

Solution of a Problem in Concurrent Programming Control

E. W. Dijkstra
Technological University, Eindhoven, The Netherlands

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

Introduction

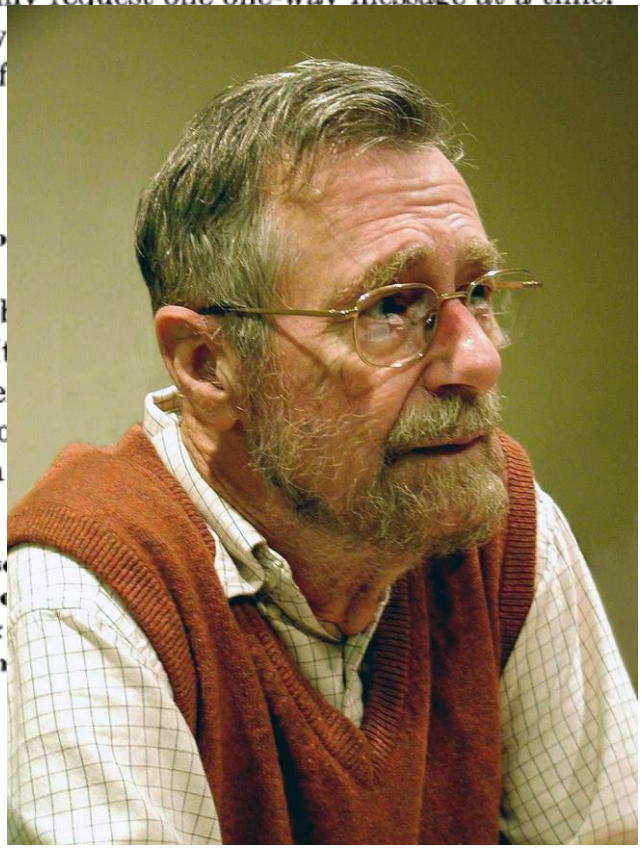
Given in this paper is a solution to a problem for which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. The paper consists of three parts: the problem, the solution, and the proof. Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in

computer can only request one one-way message at a time. And only this w
this problem is f

The Solution

The common
"Boo
The integer
will only be set
by the others. It
well outside the
mentioned set to
The program

```
"integer j;
Li0:  b[i] := fals
Li1:  if k ≠ i the
Li2:  begin c[i] :
Li3:  if b[k] then
      go to Li1
      end
        else
```



Dekker's Solution

```
wants_to_enter[0] = true
while wants_to_enter[1] {
        if turn != 0 {
                wants_to_enter[0] = false
                while turn != 0 { // busy wait }
                wants_to_enter[0] = true
        }
}
// critical section ...
turn = 1
wants_to_enter[0] = false
```

Dijsktra's Solution

```
b[i] = false
while(sw[i]) {
        sw[i] = false
        if (k!=i) {
                c[i] = true
                if(b[k])
                        k = i
                sw[i] = true
        } else {
                c[i] = false
                for(j=0;j<N;j++)
                        if(j!=i && !c[j])
                                sw[i] = true
        }
 }
//critical
b[i] = true
c[i] = true
```

lock() - P

unlock() - V

flag[i] =
;
flag[0] =
;

Peterson's Solution

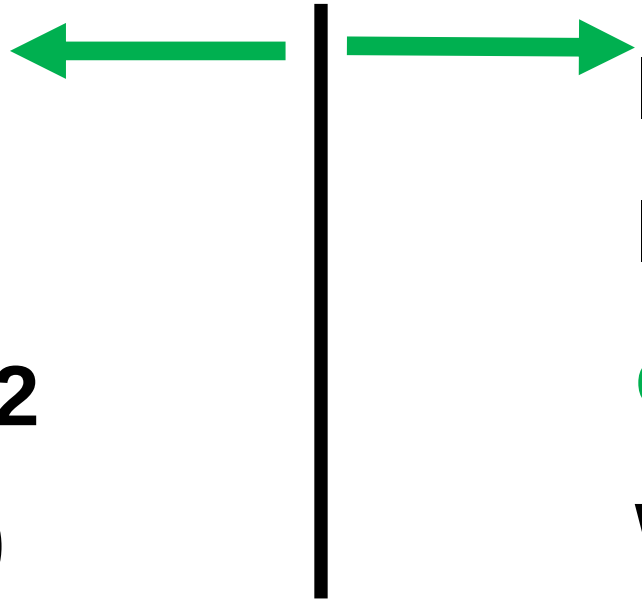
```
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1) { // busy wait }
// critical section ...
flag[0] = false;
```

Hardware assisted Solution

```
while (test_and_set(lock));
// critical section
lock = 0;
```

Race condition - solution

a = 0



Thread 1	Thread 2
lock(locka)	lock(locka)
load(a, eax)	load(a, eax)
eax = eax + 2	eax = eax + 2
write(a, eax)	write(a, eax)
unlock(locka)	unlock(locka)
eax = 0	eax = 0

Race condition - solution

?

a = 0

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 0	**eax = 2**

Race condition - solution

OK

a = 0

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 0	**eax = 2**

Race condition - solution

?

a = 2

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 0	**eax = 2**

Race condition - solution

OK

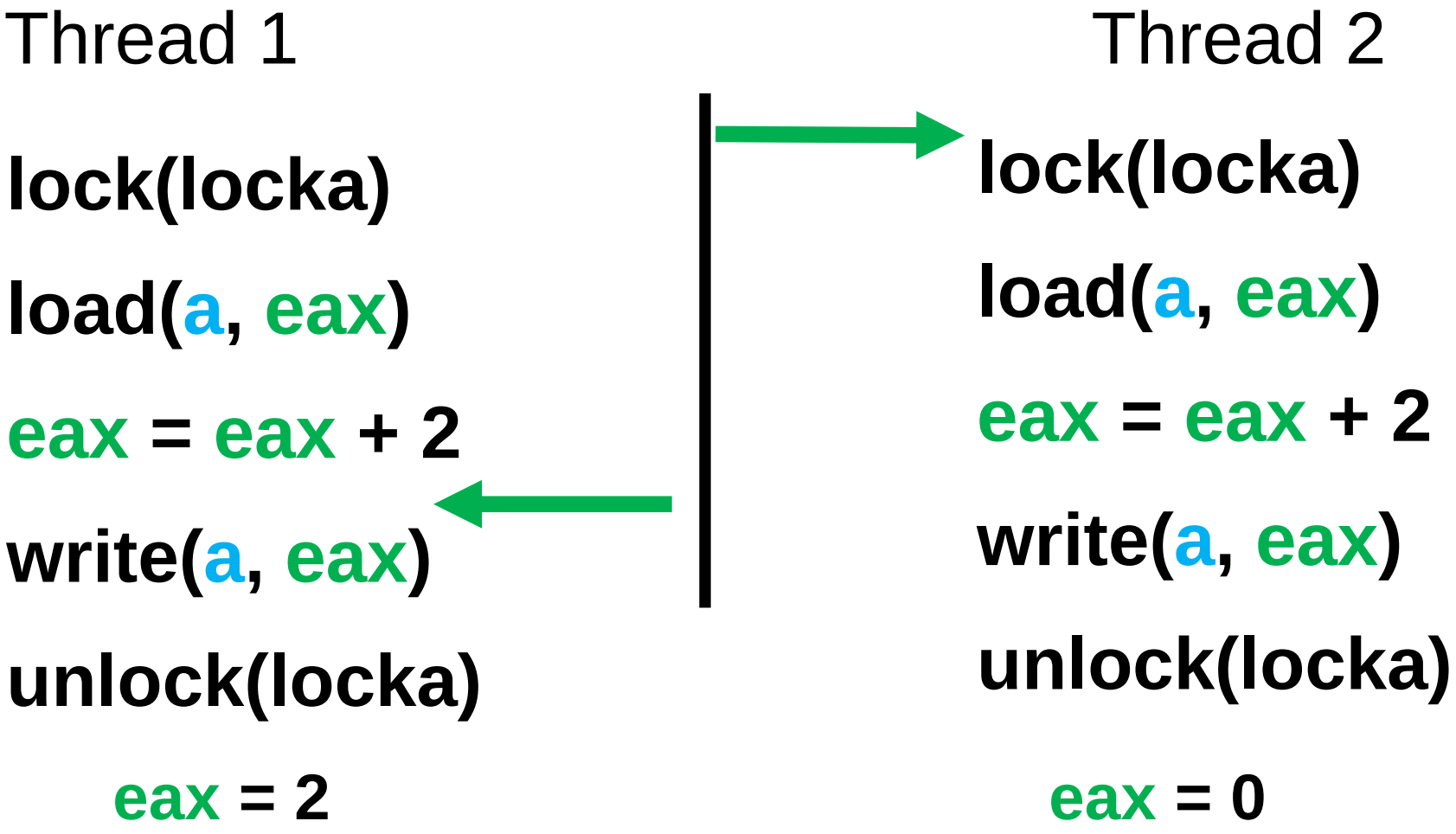
a = 2

Thread 1	Thread 2
lock(locka)	lock(locka)
load(a, eax)	load(a, eax)
eax = eax + 2	eax = eax + 2
write(a, eax)	write(a, eax)
unlock(locka)	unlock(locka)
eax = 0	eax = 2

Race condition - solution

?

a = 0



Race condition - solution

OK

a = 0

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 2	**eax = 0**

Race condition - solution

?

a = 2

Thread 1

lock(locka)

load(a, eax)

eax = eax + 2

write(a, eax)

unlock(locka)

eax = 2

Thread 2

lock(locka)

load(a, eax)

eax = eax + 2

write(a, eax)

unlock(locka)

eax = 0

Race condition - solution

OK

a = 2

Thread 1

lock(locka)

load(a, eax)

eax = eax + 2

write(a, eax)

unlock(locka)

eax = 2

Thread 2

lock(locka)

load(a, eax)

eax = eax + 2

write(a, eax)

unlock(locka)

eax = 0

Race condition - solution

?

a = 4

Thread 1	Thread 2
lock(locka)	lock(locka)
load(a, eax)	load(a, eax)
eax = eax + 2	eax = eax + 2
write(a, eax)	write(a, eax)
unlock(locka)	unlock(locka)
eax = 2	eax = 2

Race condition - solution

OK

a = 4

Thread 1	Thread 2
lock(locka)	lock(locka)
load(a, eax)	load(a, eax)
eax = eax + 2	eax = eax + 2
write(a, eax)	write(a, eax)
unlock(locka)	unlock(locka)
eax = 2	eax = 2

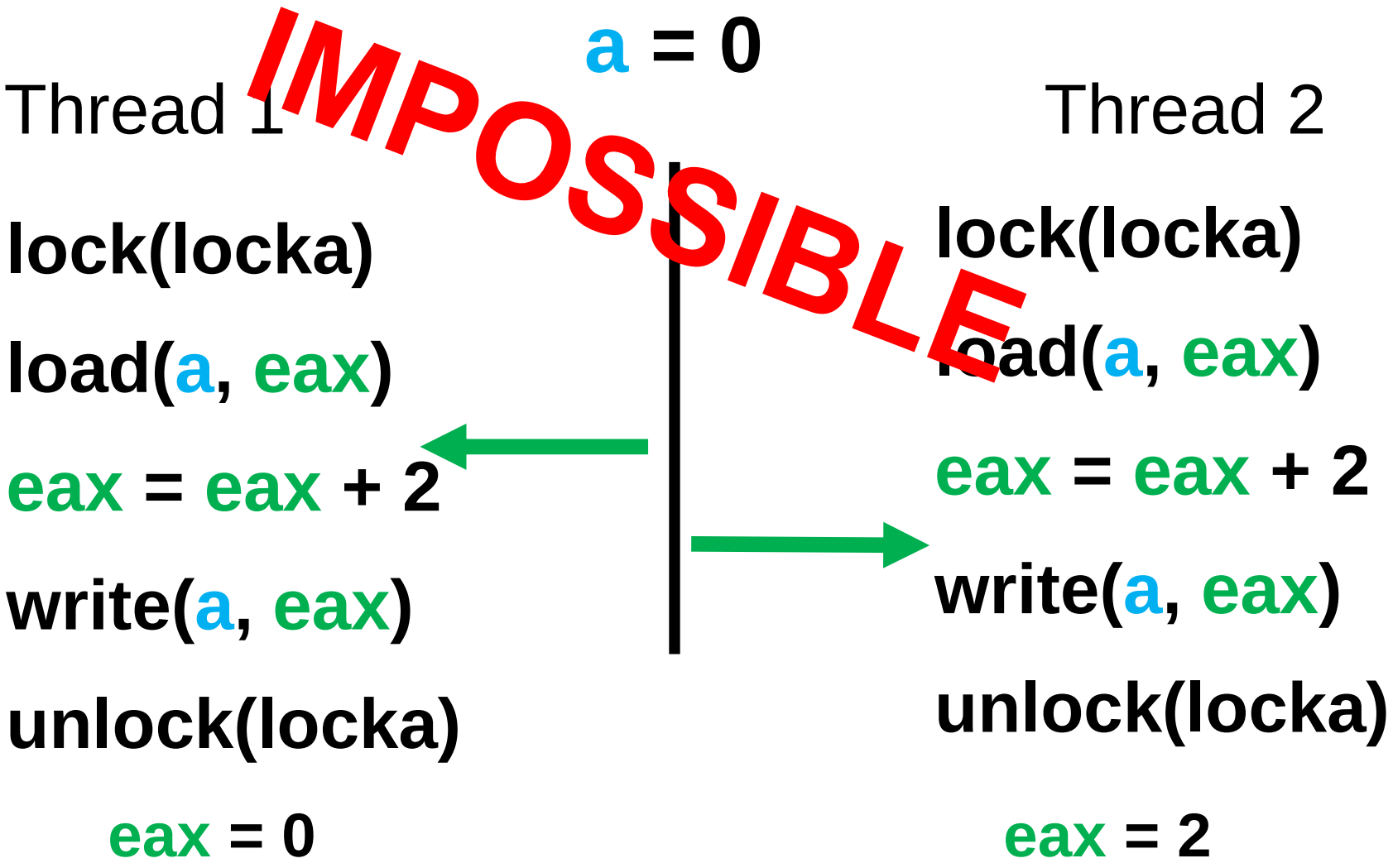
Race condition - solution

?

a = 0

Thread 1	Thread 2
lock(locka)	lock(locka)
load(a, eax)	load(a, eax)
eax = eax + 2	eax = eax + 2
write(a, eax)	write(a, eax)
unlock(locka)	unlock(locka)
eax = 0	eax = 2

Race condition - solution



Mutex Pthread

ÎN MAIN

Înainte de a porni thread-urile

pthread_mutex_t **mutex**;

pthread_mutex_init(&**mutex**, NULL);

Mutex Pthread

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

Poate fi folosit să anunțe că acest mutex e împărțit mai multor procese

Mutex Pthread

pthread_mutex_lock(&**mutex**);
load(a, eax)

eax = eax + 2

write(a, eax)

pthread_mutex_unlock(&**mutex**);

Mutex Pthread

ÎN MAIN

După ce au terminat thread-urile

pthread_mutex_destroy(&**mutex**);

Semaphore

ÎN MAIN

Înainte de a porni thread-urile

gcc –o executabil cod.c –lpthread –lrt

#include<pthread.h>

#include<semaphore.h>

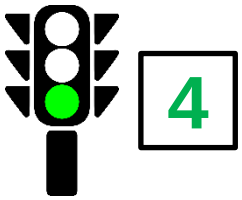
sem_t **semaphore**;

int semaphore_value= **4**;

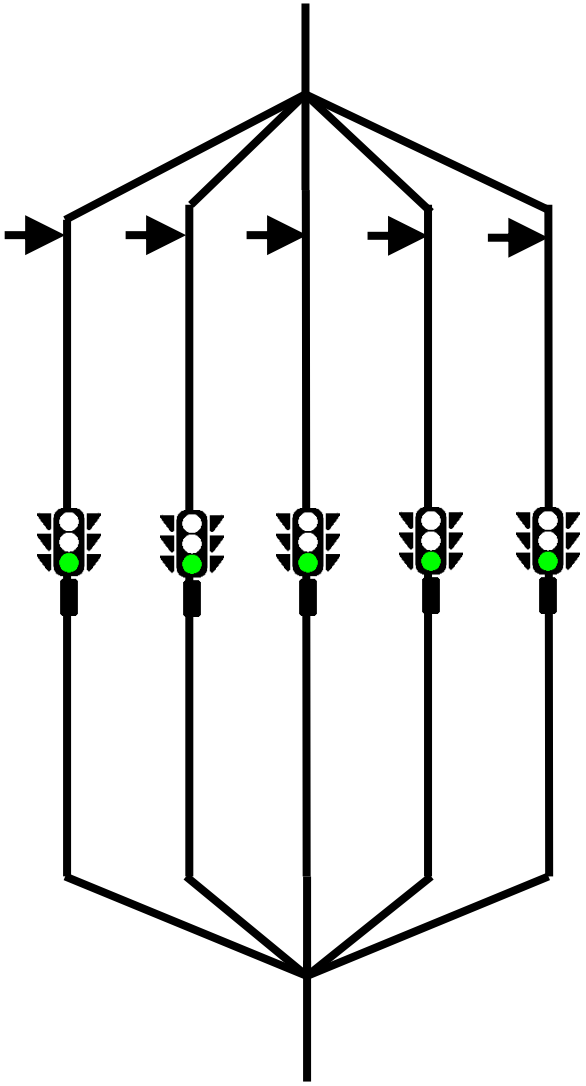
sem_init(& **semaphore**, 0, semaphore_value);

Semaphore

```
sem_wait(&semaphore);
```

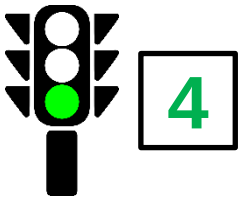


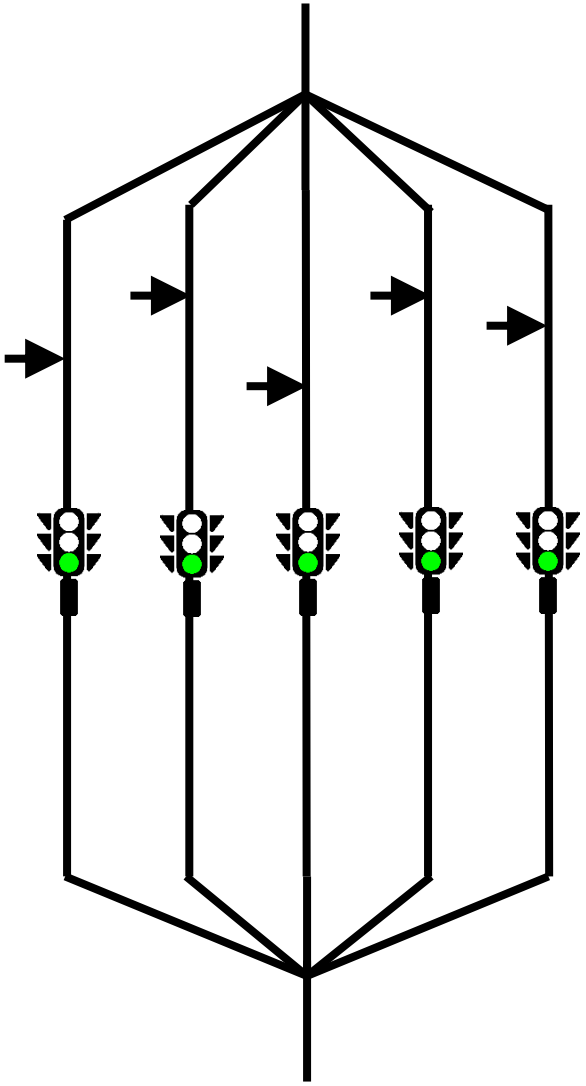
P() sau Proberen
 - Dijsktra



Semaphore

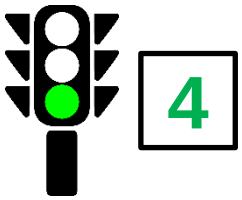
```
sem_wait(&semaphore);
```

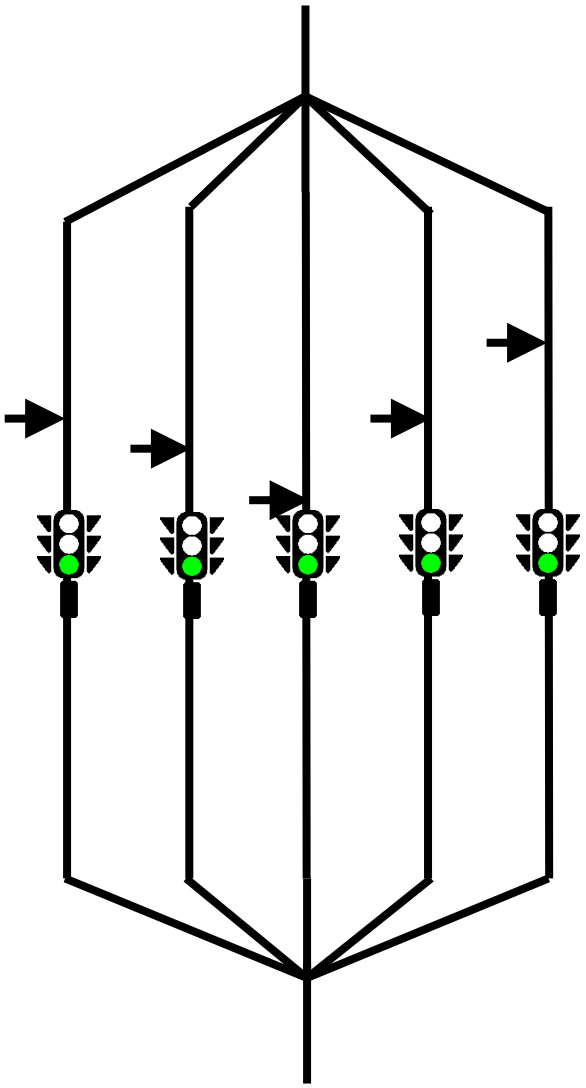




Semaphore

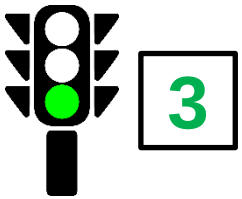
```
sem_wait(&semaphore);
```

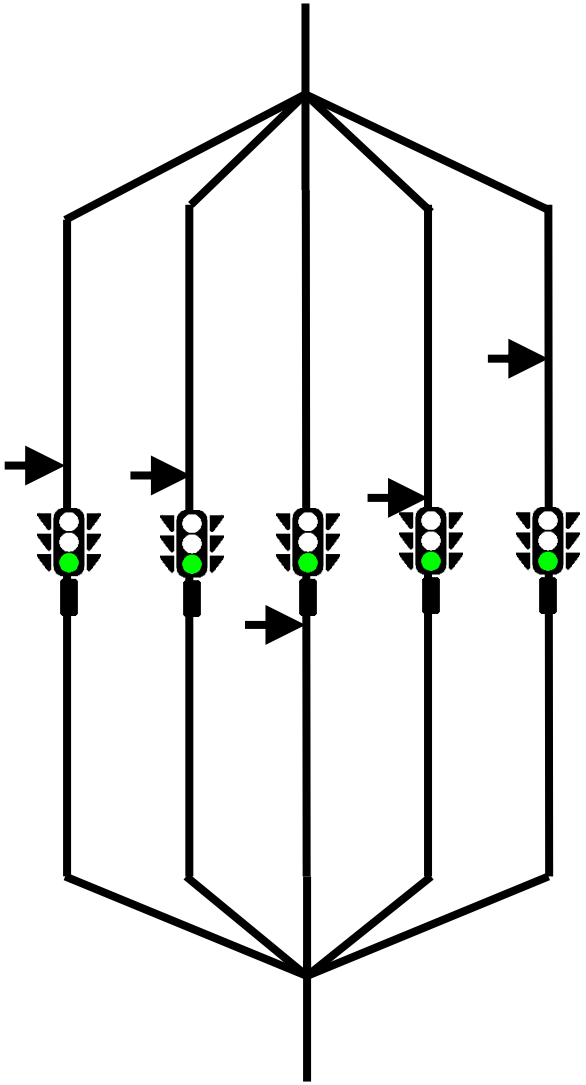




Semaphore

```
sem_wait(&semaphore);
```

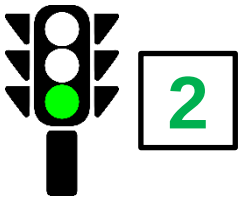


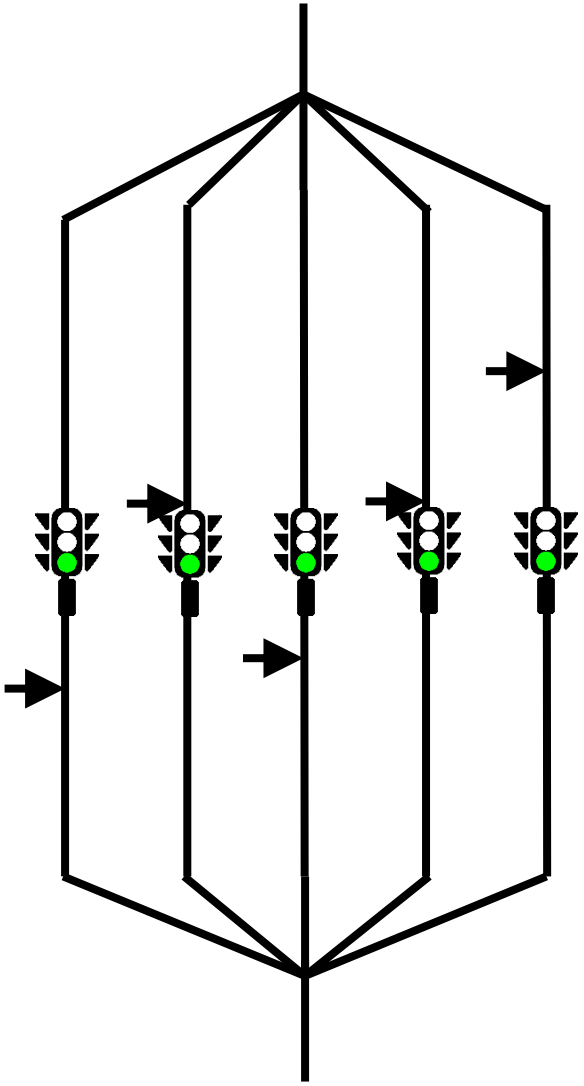


Semaphore

Nu contează că doua thread-uri au ajuns simultan la semafor, acesta este protejat, la fel ca un mutex.

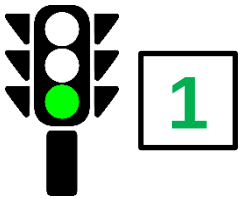
```
sem_wait(&semaphore);
```

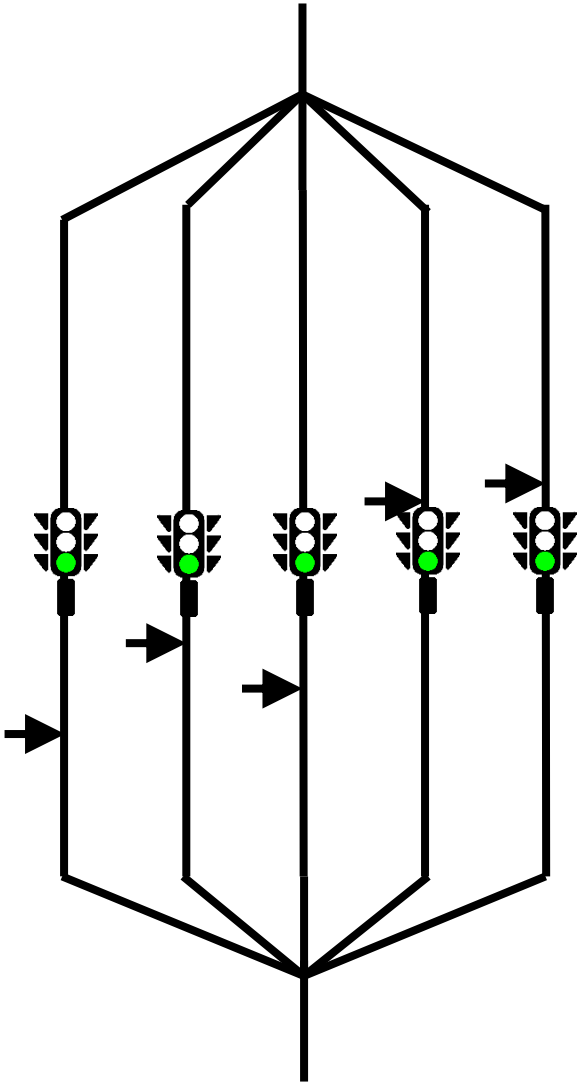




Semaphore

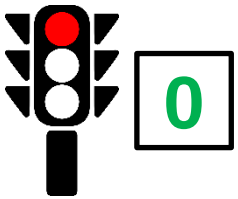
```
sem_wait(&semaphore);
```

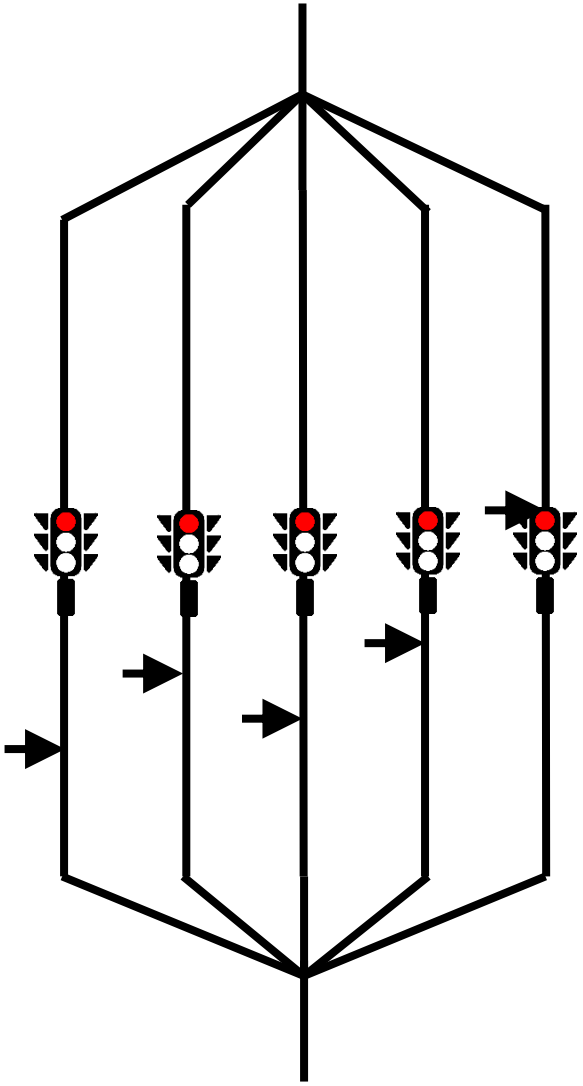




Semaphore

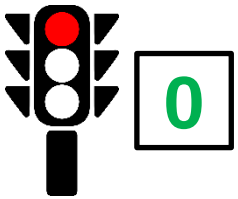
```
sem_wait(&semaphore);
```

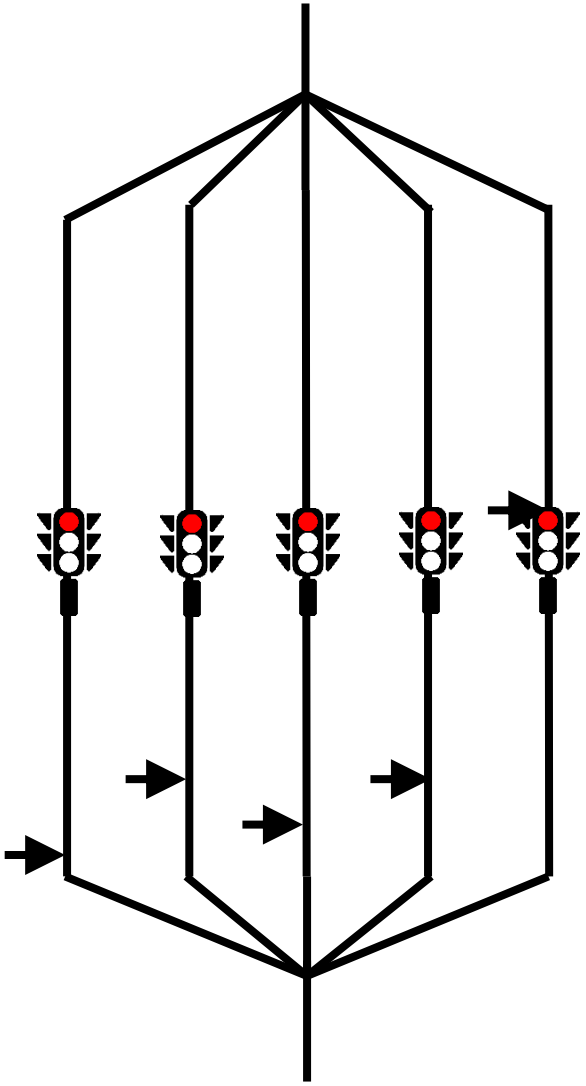




Semaphore

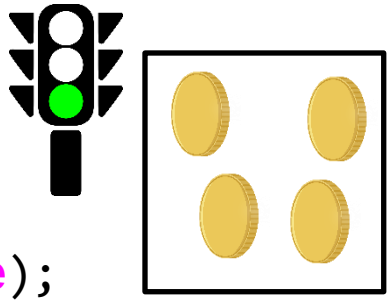
```
sem_wait(&semaphore);
```



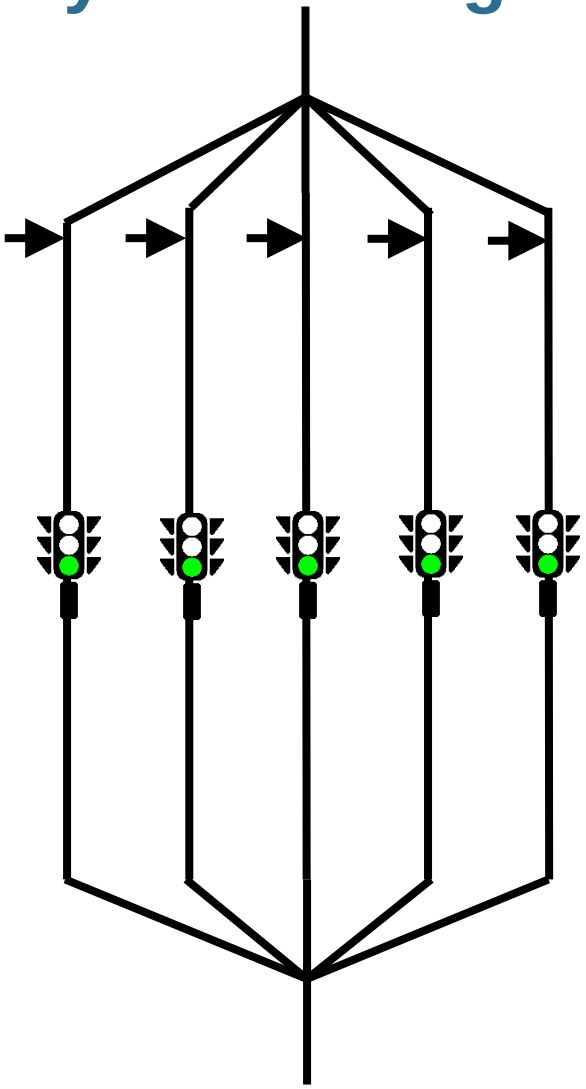


Semaphore – A different way of thinking

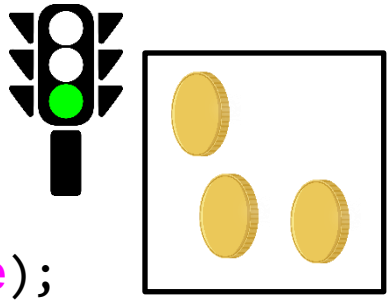
Un semafor are un set de token-uri.



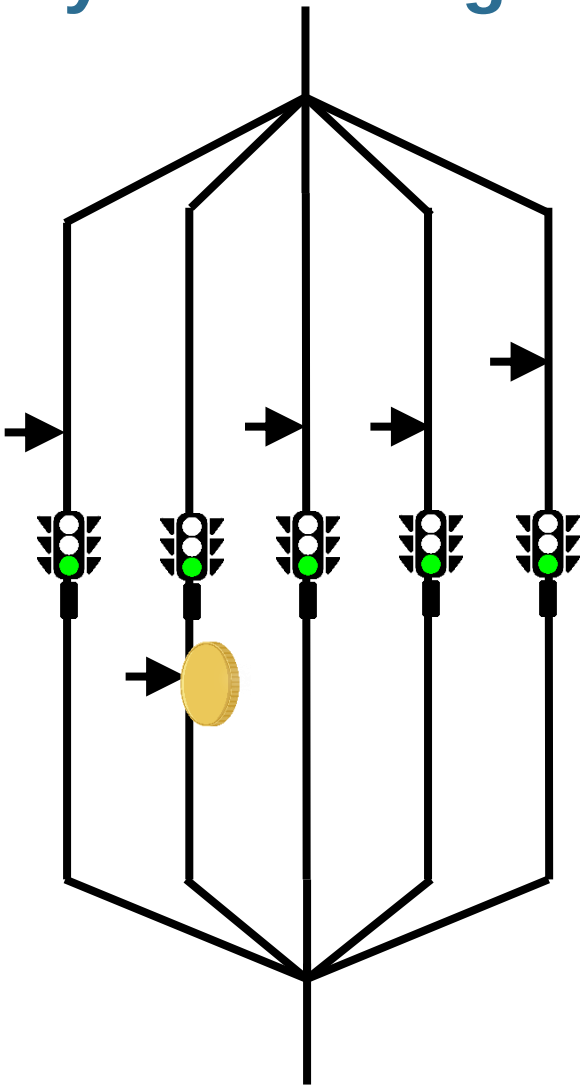
```
sem_wait(&semaphore);
```



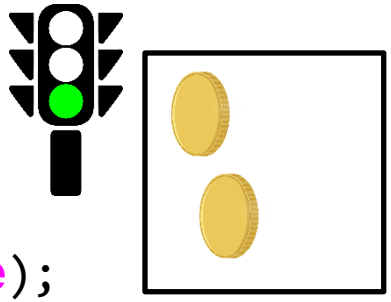
Semaphore – A different way of thinking



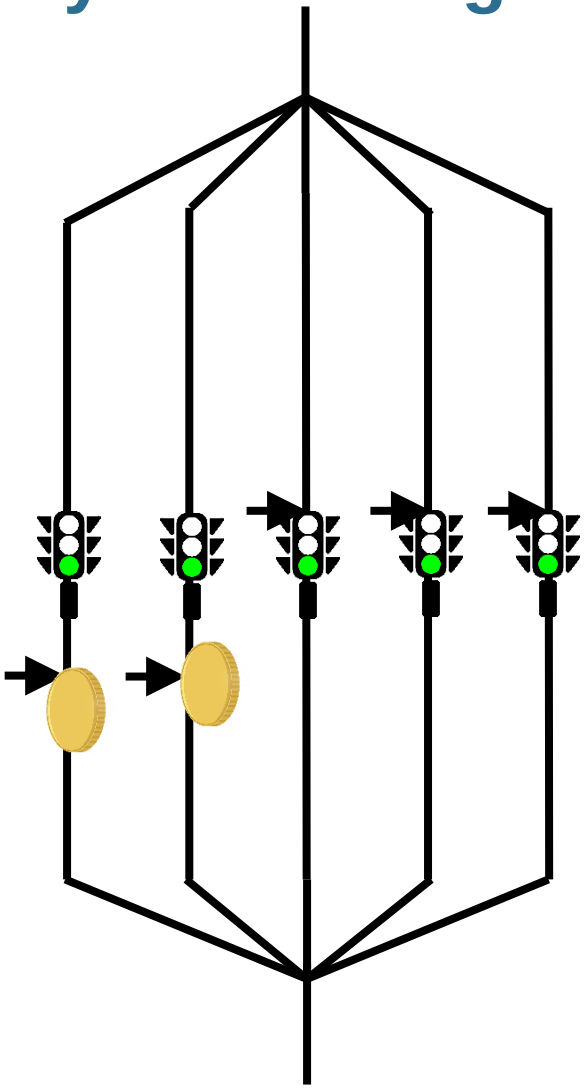
```
sem_wait(&semaphore);
```



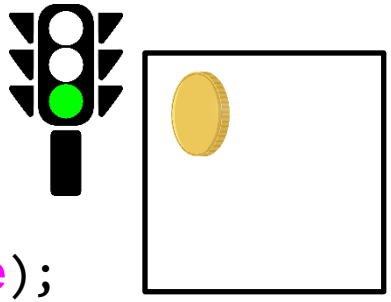
Semaphore – A different way of thinking



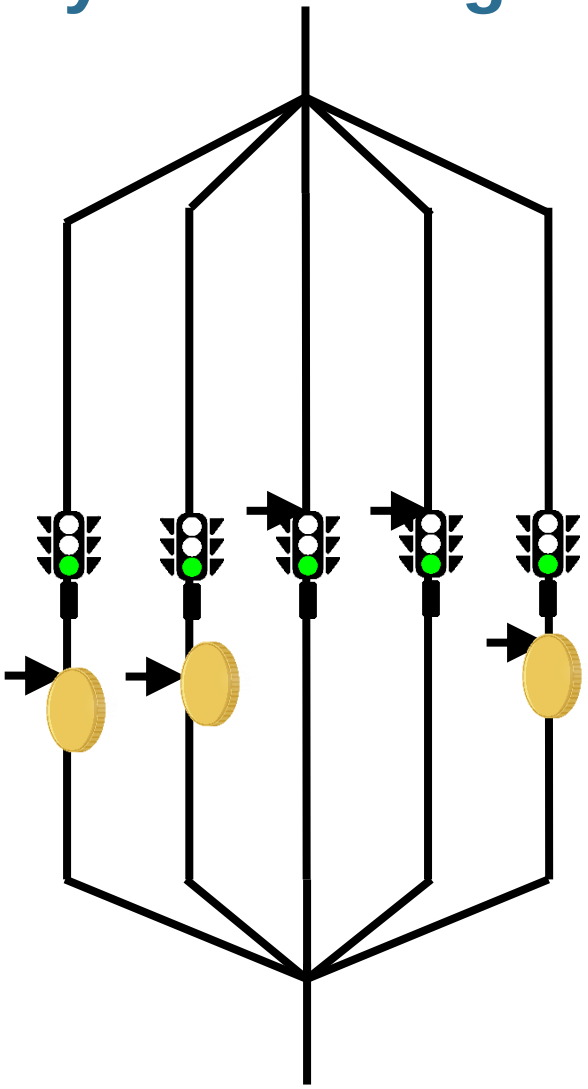
```
sem_wait(&semaphore);
```



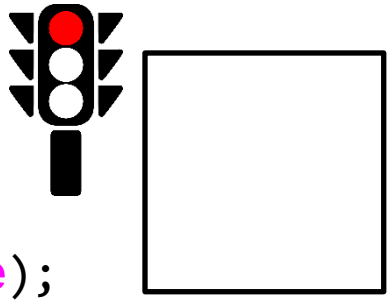
Semaphore – A different way of thinking



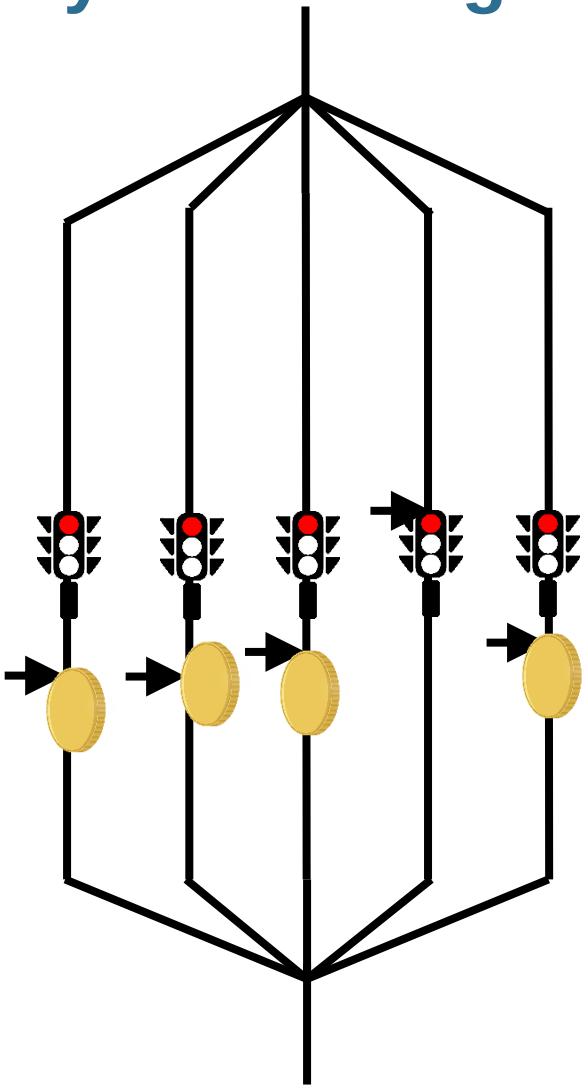
```
sem_wait(&semaphore);
```



Semaphore – A different way of thinking

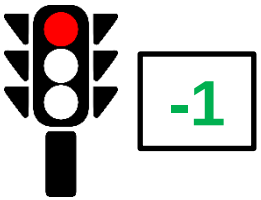


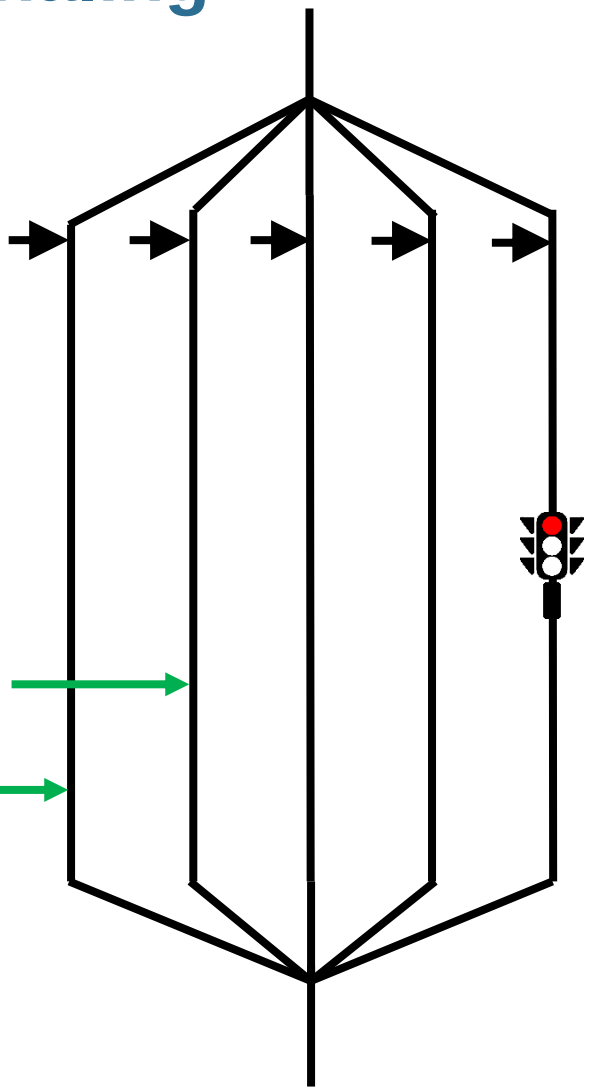
```
sem_wait(&semaphore);
```



Semaphore – Signaling

```
sem_wait(&semaphore);
```





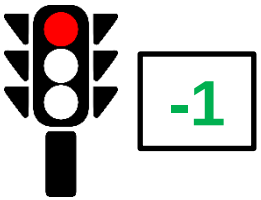
```
sem_post(&semaphore);
```

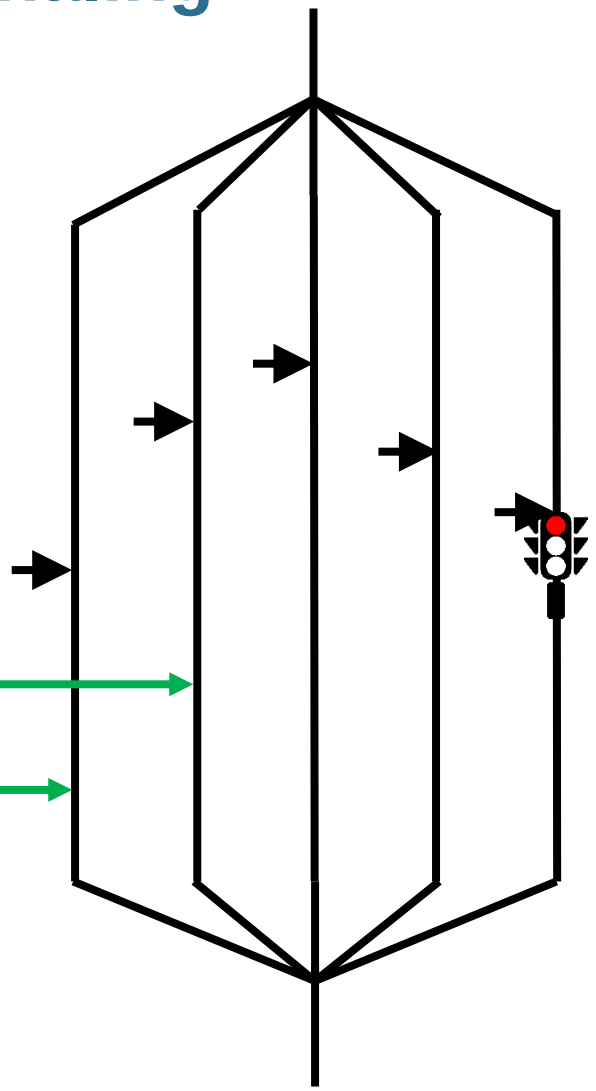
```
sem_post(&semaphore);
```

V() sau Verogen
 - Dijsktra

Semaphore – Signaling

```
sem_wait(&semaphore);
```



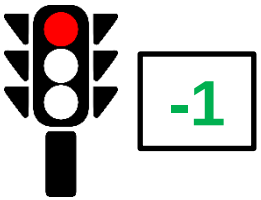


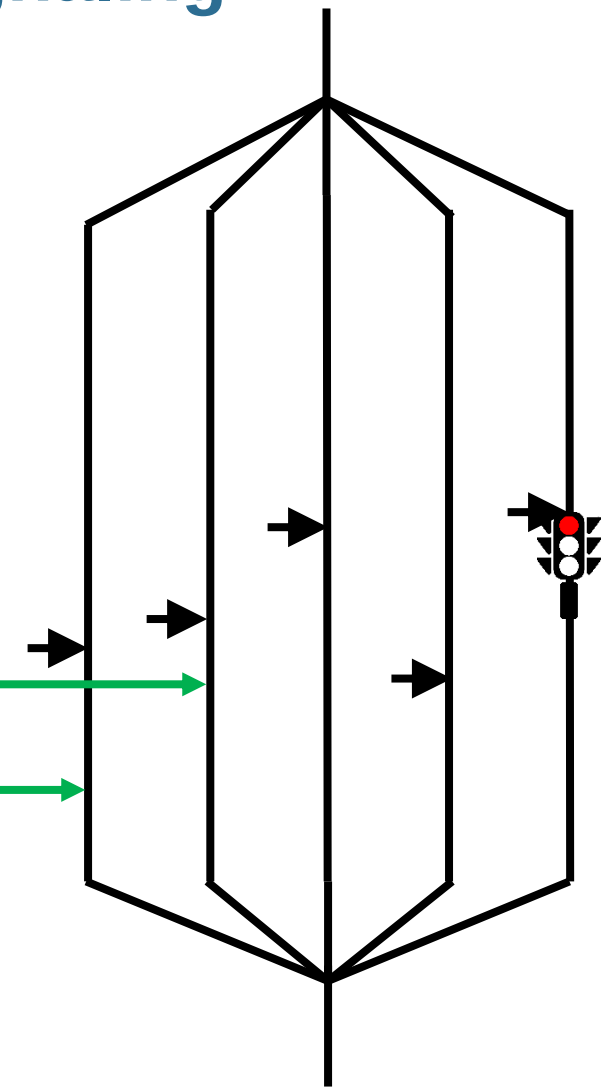
```
sem_post(&semaphore);
```

```
sem_post(&semaphore);
```

Semaphore – Signaling

```
sem_wait(&semaphore);
```



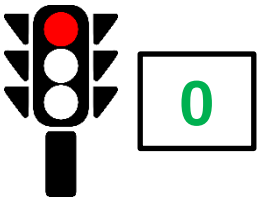


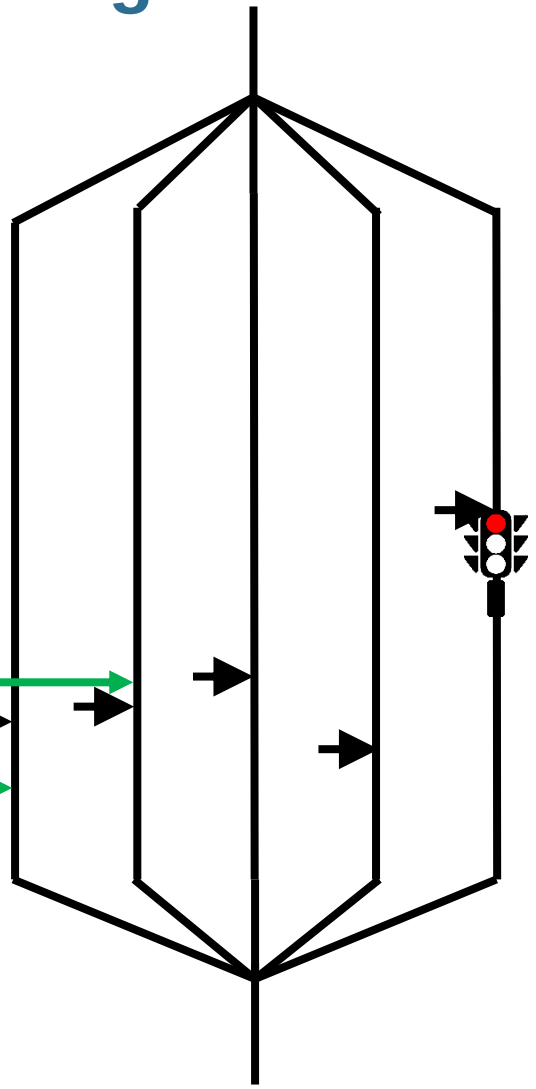
```
sem_post(&semaphore);
```

```
sem_post(&semaphore);
```

Semaphore – Signaling

```
sem_wait(&semaphore);
```

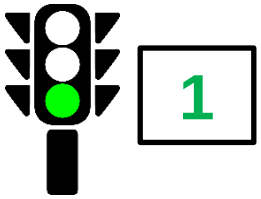




```
sem_post(&semaphore);
sem_post(&semaphore);
```

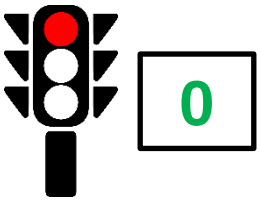
Semaphore – Signaling

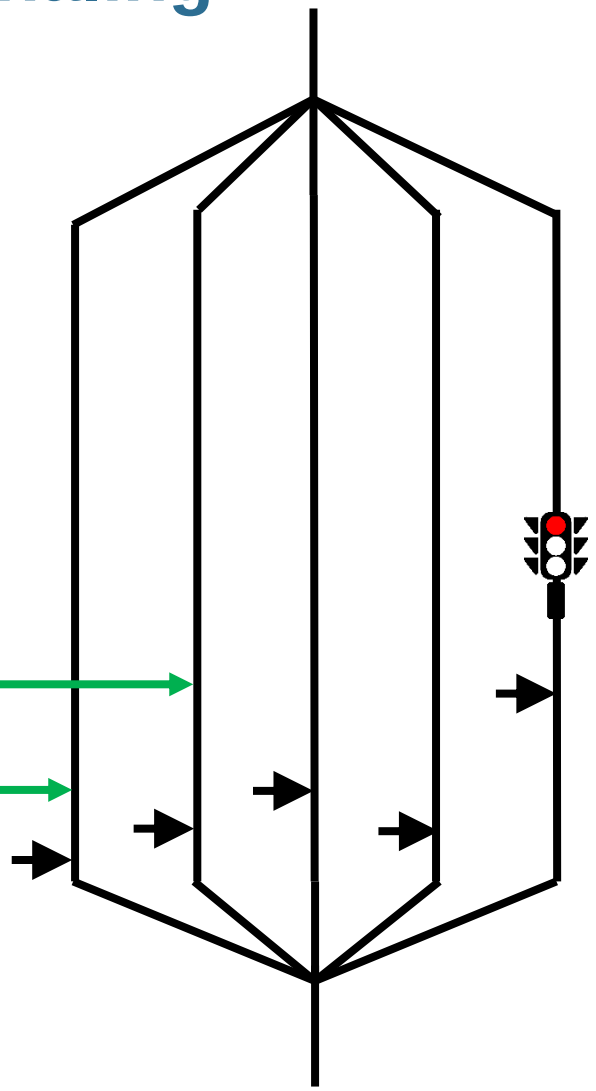
```
sem_wait(&semaphore);
```



Semaphore – Signaling

```
sem_wait(&semaphore);
```





Semaphore

ÎN MAIN

După ce au terminat thread-urile

sem_destroy(& **semaphore**);

Barrier

ÎN MAIN

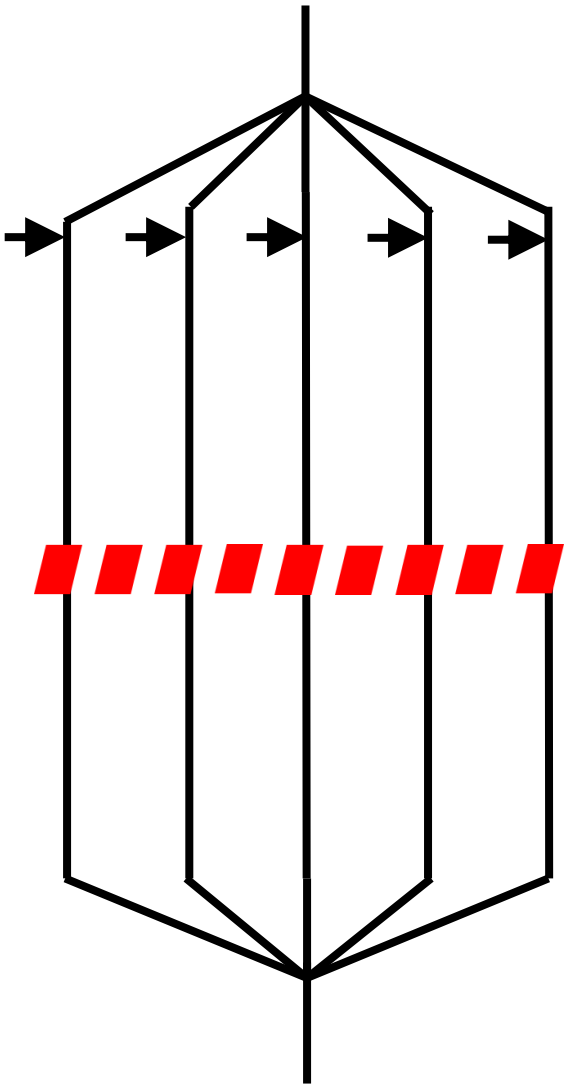
Înainte de a porni thread-urile

```
pthread_barrier_t barrier;
int num_threads = 5;
pthread_barrier_init(&barrier, NULL, num_threads);
```

Barrier

pthread_barrier_wait(&**barrier**);

5

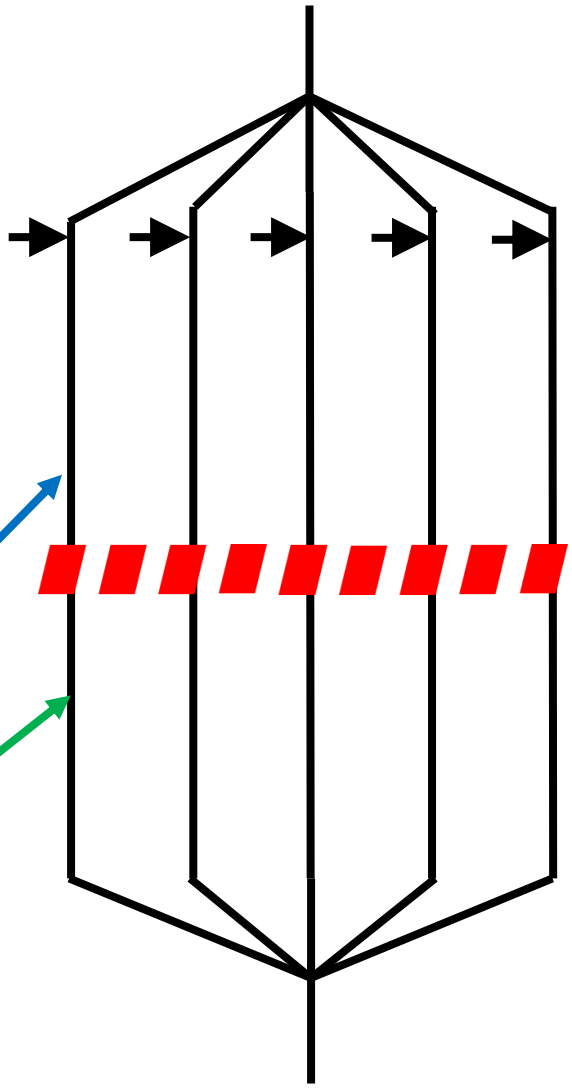


Barrier

pthread_barrier_wait(&**barrier**);

5

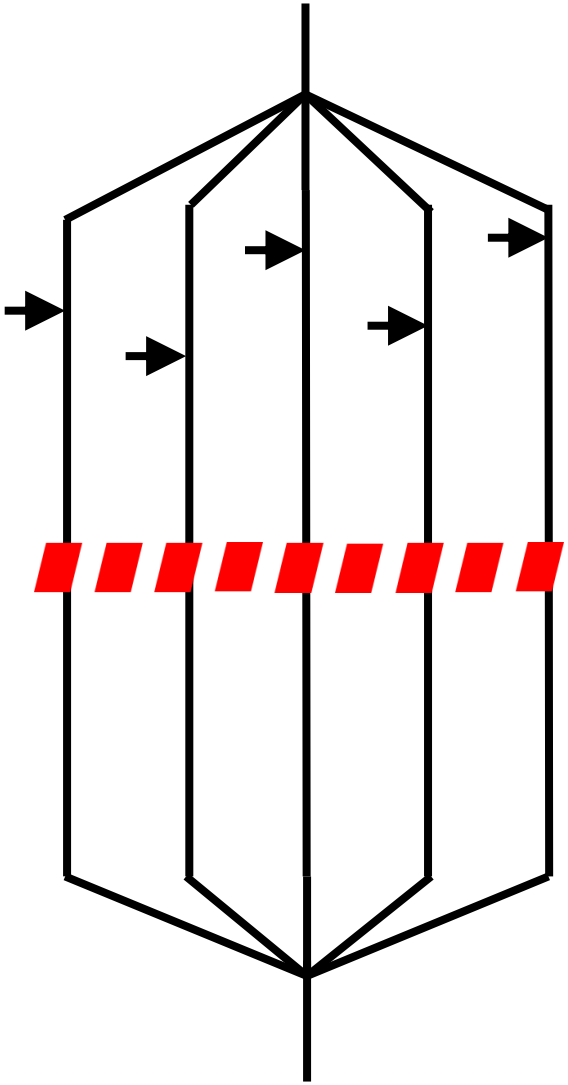
Pentru toate thread-urile,
Tot codul de aici
este executat înainte de orice
bucata de cod de aici



Barrier

pthread_barrier_wait(&**barrier**);

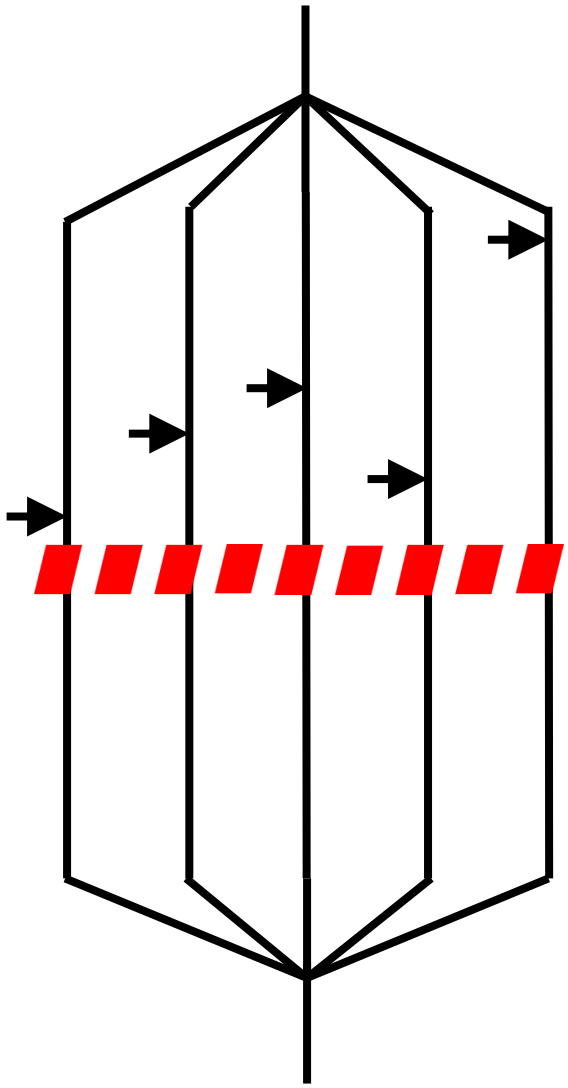
5



Barrier

pthread_barrier_wait(&**barrier**);

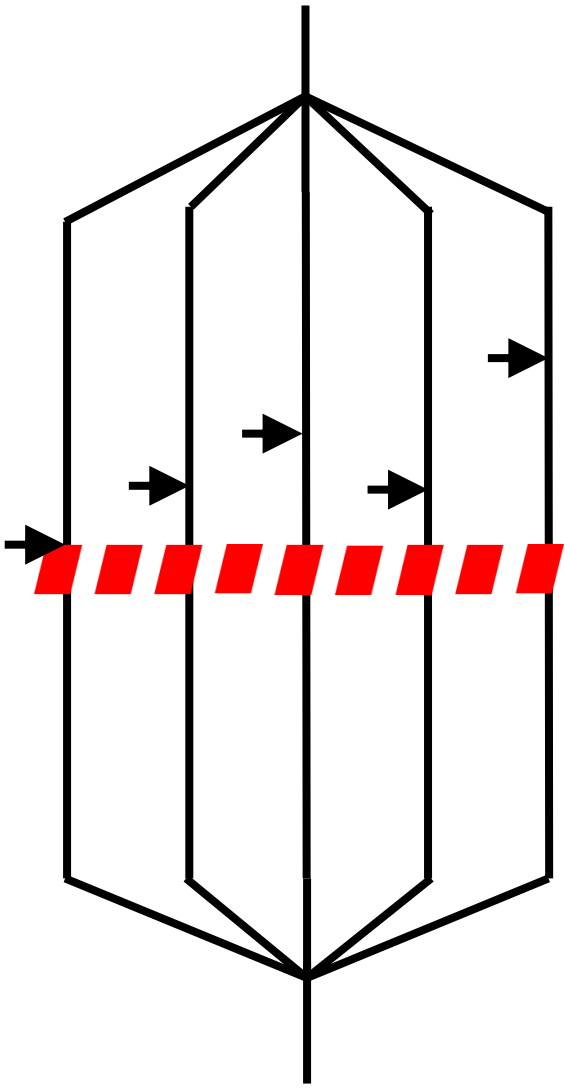
5



Barrier

pthread_barrier_wait(&**barrier**);

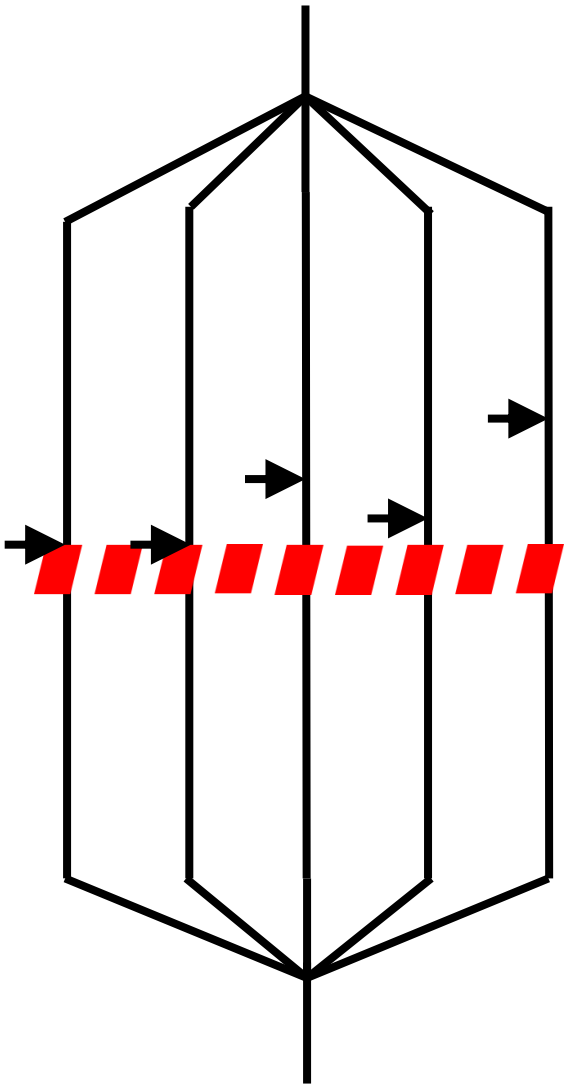
5



Barrier

pthread_barrier_wait(&**barrier**);

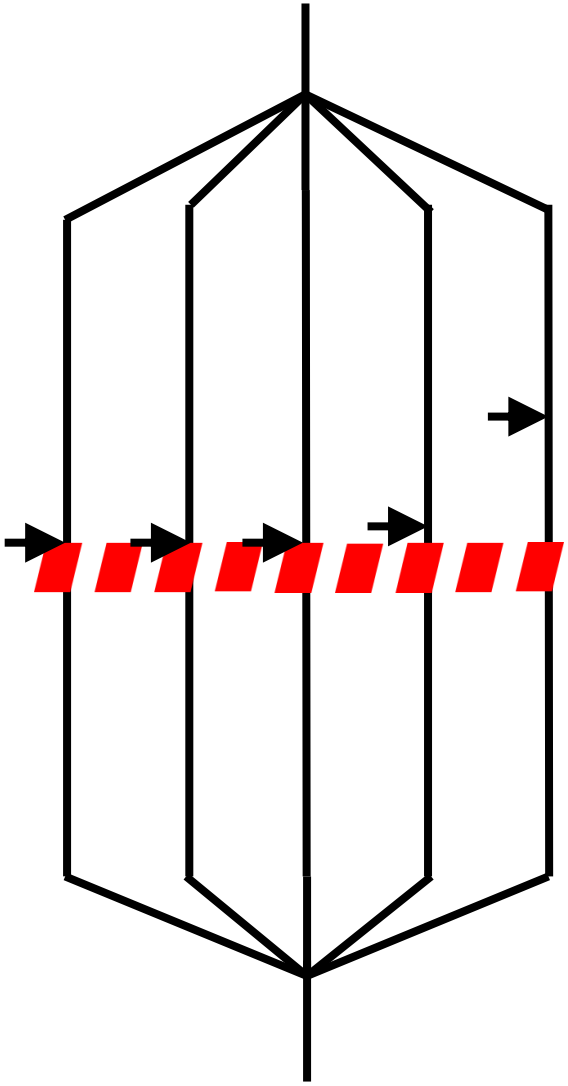
3



Barrier

pthread_barrier_wait(&**barrier**);

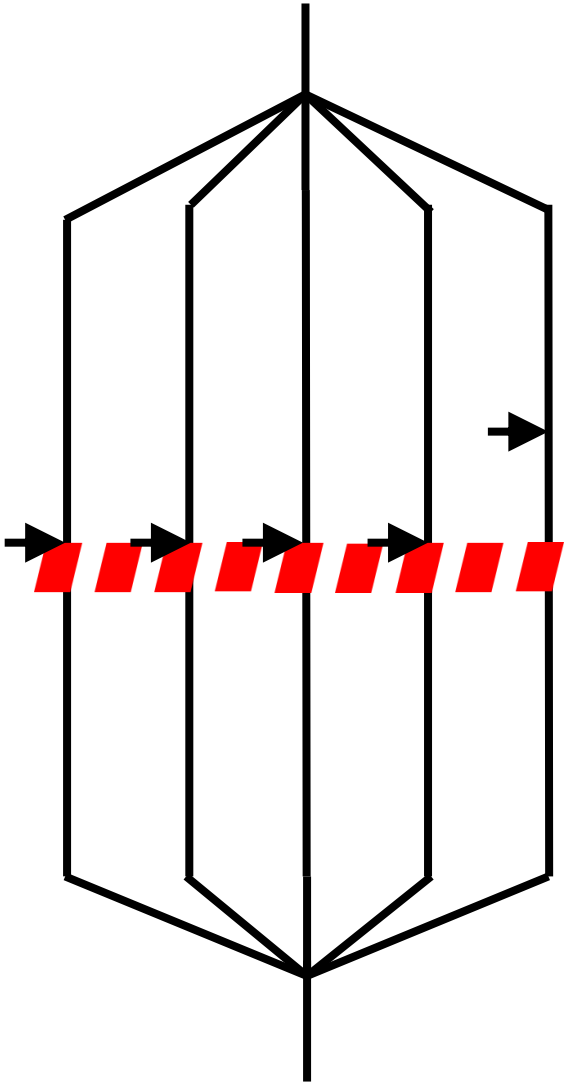
2



Barrier

pthread_barrier_wait(&**barrier**);

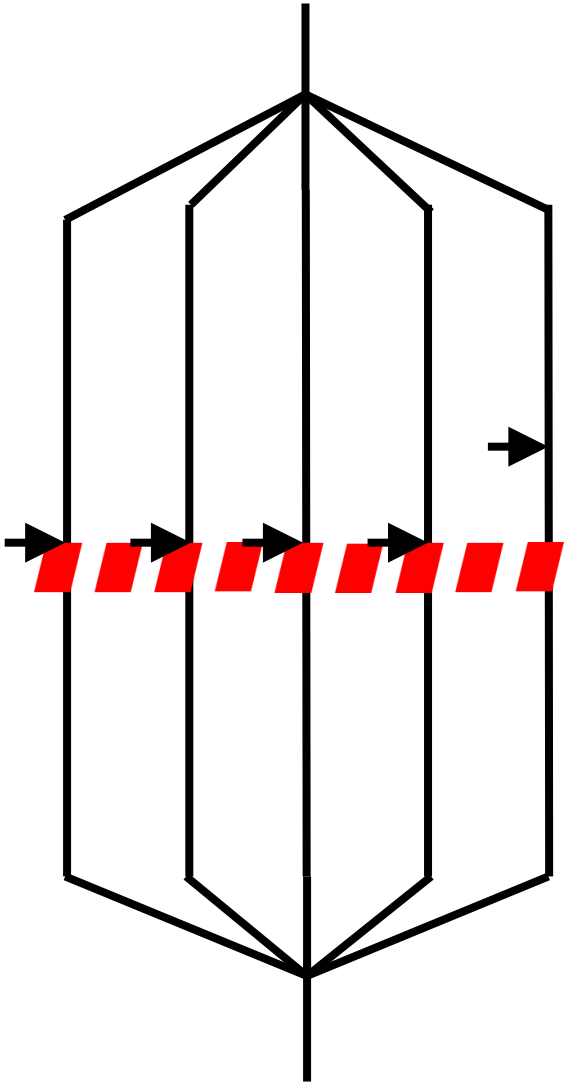
1



Barrier

pthread_barrier_wait(&**barrier**);

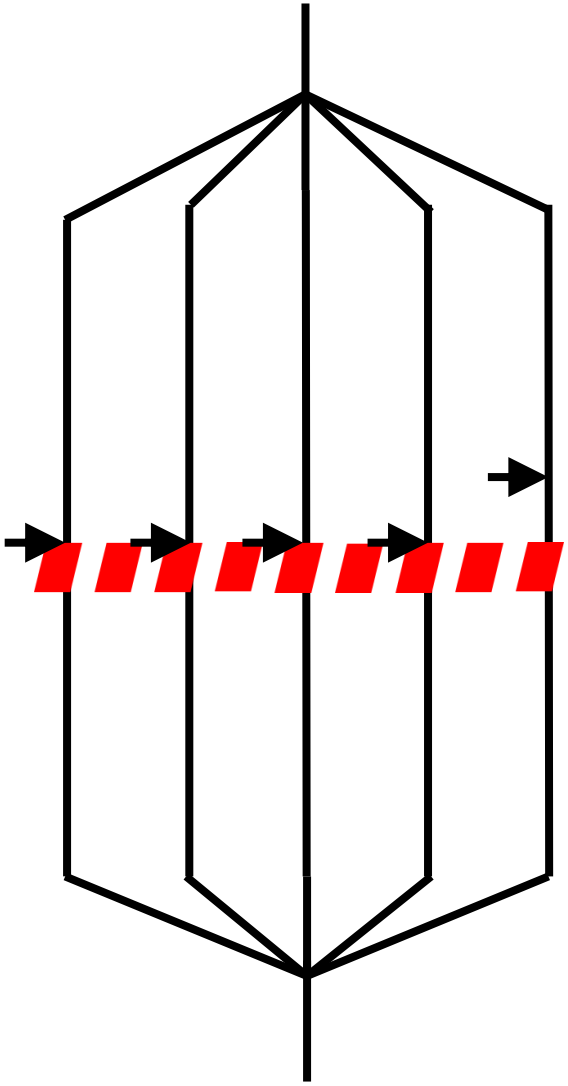
1



Barrier

pthread_barrier_wait(&**barrier**);

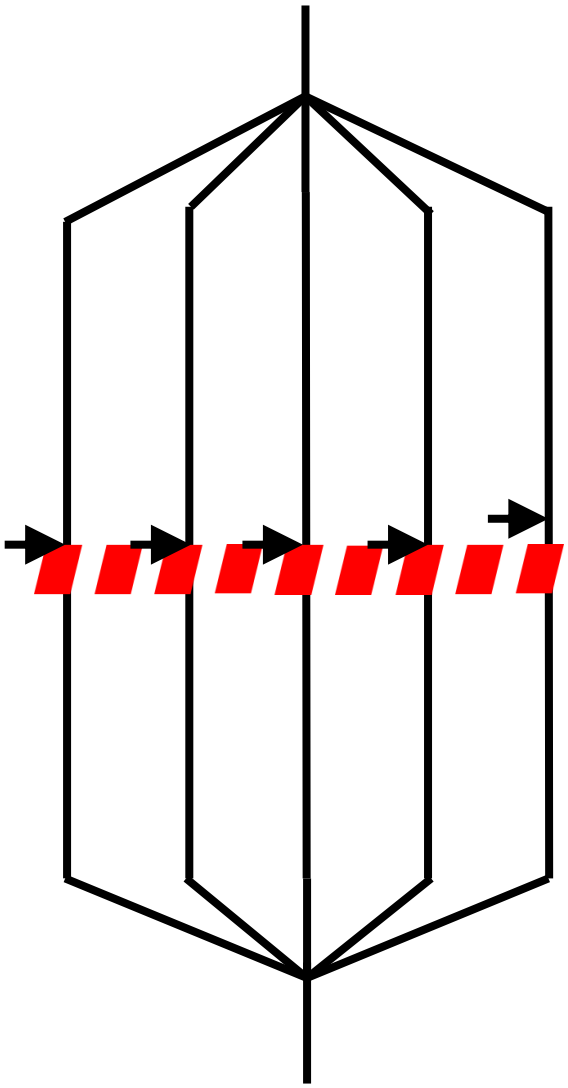
1



Barrier

pthread_barrier_wait(&**barrier**);

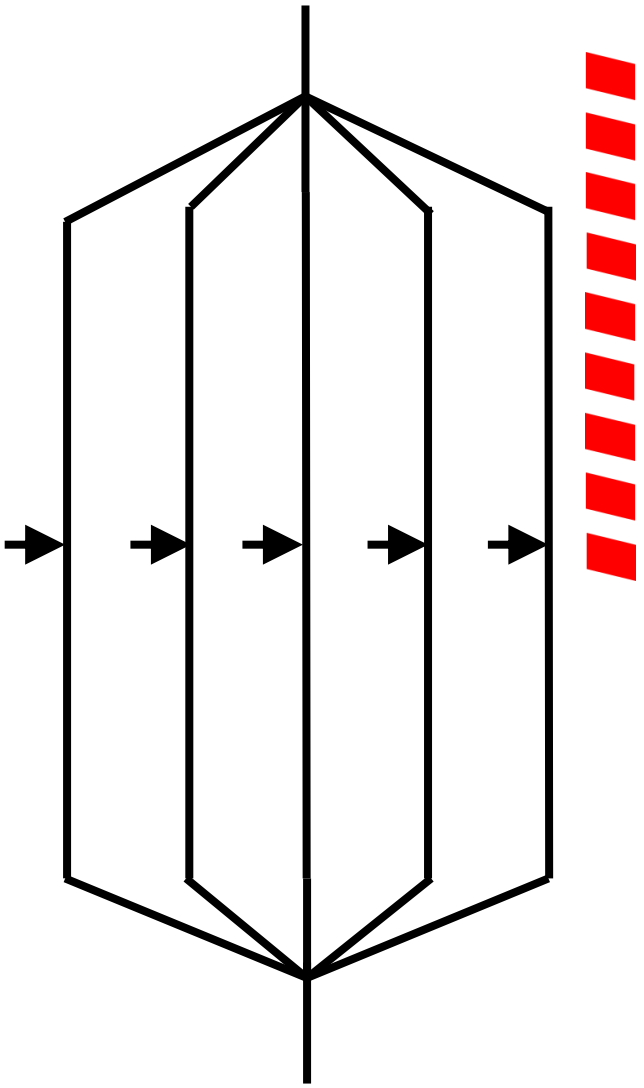
1



Barrier

pthread_barrier_wait(&**barrier**);

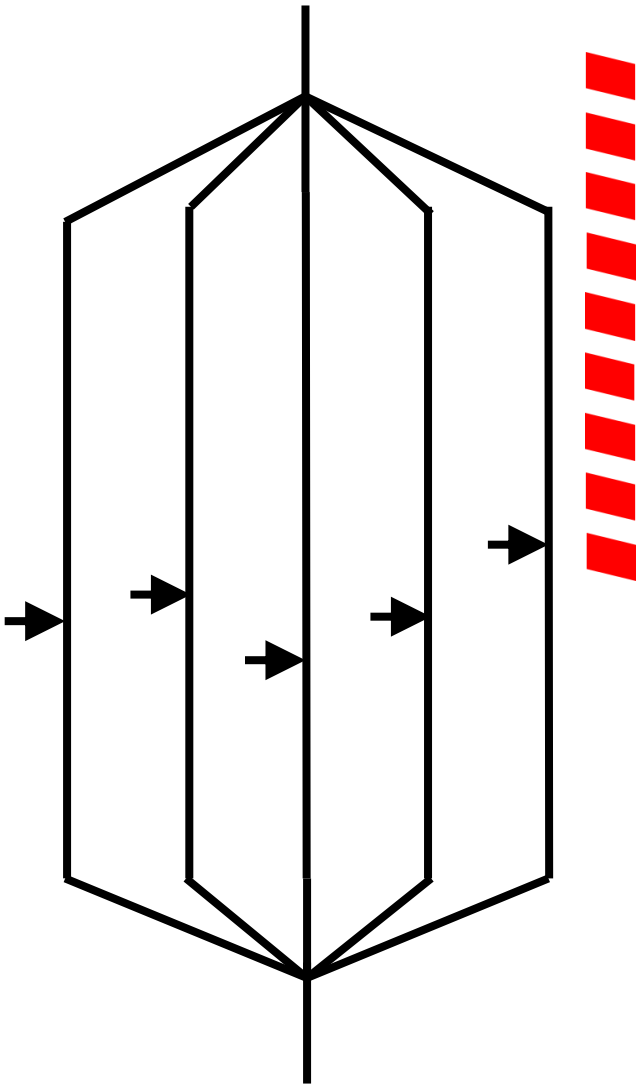
0



Barrier

pthread_barrier_wait(&**barrier**);

0



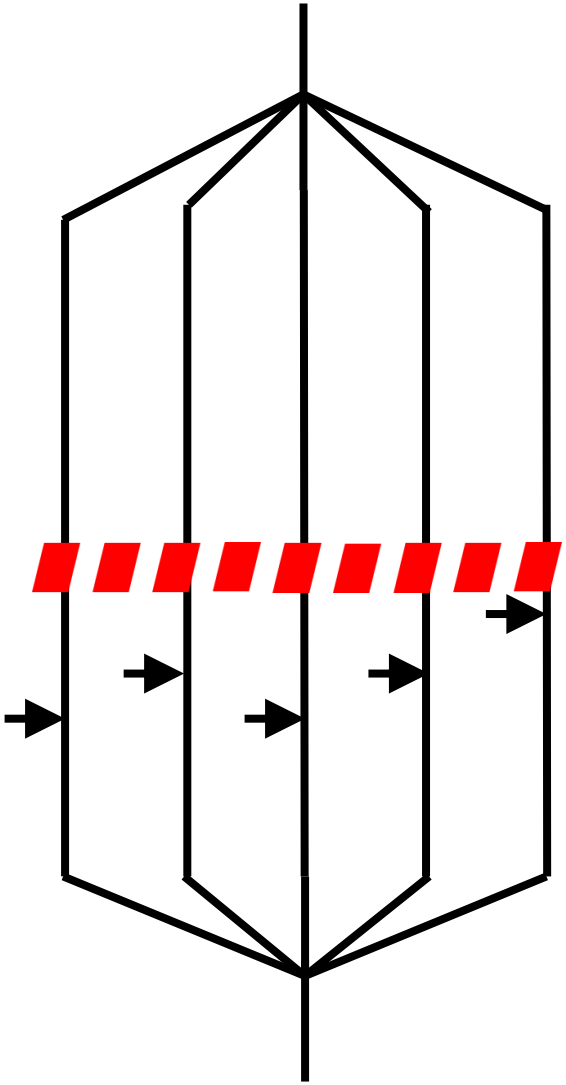
Barrier

Cum știe o barieră când să se reseteze?

O soluție ar fi:
Reusable Barrier in
The Little Book of Semaphores
By Allen B. Downey

pthread_barrier_wait(&**barrier**);

5



ÎN MAIN

După ce au terminat thread-urile

```
pthread_barrier_destroy(&barrier);
```

Unele probleme nu pot fi paralelizate

Calculating the hash of a hash of a hash …

of a string.

Deep First Search

Huffman decoding

Outer loops of most simulations

P complete problems

Paralelizare prin împărțirea problemei

Sunt o serie de probleme care sunt extrem de ușor paralelizabile.

Embarrassingly parallel

Embarrassingly parallel problems

Multiplicare unui vector cu un scalar

9	6	9	4	2	7	6	5	6	1

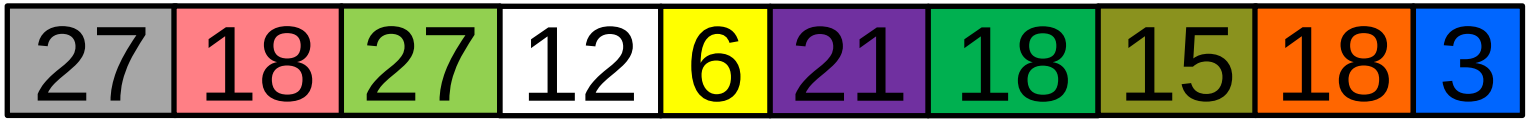
* 3

27	18	27	12	6	21	18	15	18	3

Embarrassingly parallel problems

Toate calculele pot fi efectuate în același timp

* 3



Embarrassingly parallel problems

Câte elemente sunt?

Embarrassingly parallel problems

Câte elemente sunt?

Embarrassingly parallel problems

Câte elemente sunt? **N**

Embarrassingly parallel problems

Dar câte elemente de procesare?

Embarrassingly parallel problems

Dar câte elemente de procesare? **P**

Embarrassingly parallel problems

Dar câte thread-uri?

Embarrassingly parallel problems

Dar câte thread-uri? **P**

În majoritatea cazurilor obținem performanță maximă când numărul de thread-uri este egal cu numărul de elemente de procesare, sau core-uri.

Embarrassingly parallel problems

Cum este P față de N?

Embarrassingly parallel problems

P << N

Embarrassingly parallel problems

Caz concret: P = 2

Cum împărțim?

Embarrassingly parallel problems

Caz concret: P = 2

Cum împărțim?

Thread 1

Thread 2

Embarrassingly parallel problems

Caz concret: P = 2

Cum împărțim?

Thread 1

Thread 2

Embarrassingly parallel problems

Caz concret: P = 2

Cum împărțim?

Thread 1

Thread 2

Embarrassingly parallel problems

Caz concret: P = 2

Cum împărțim? Putem și random

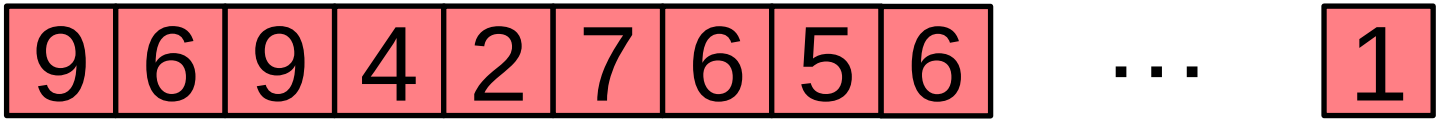
Thread 1

Thread 2

Embarrassingly parallel problems

Caz concret: P = 2

Cum împărțim?



Thread 1

Thread 2

Embarrassingly parallel problems

Caz concret: P = 2

Cum împărțim?

Este utilă?

Thread 1

Thread 2

Embarrassingly parallel problems

Caz concret: P = 2

Cum împărțim?

Ce ne dorim?

Thread 1

Thread 2

Embarrassingly parallel problems

Caz concret: P = 2

Cum împărțim?

Ce ne dorim?

Thread 1

Thread 2

Aproximativ același număr elemente

Embarrassingly parallel problems

Aproximativ N/P elemente

Thread 1

Thread 2

Embarrassingly parallel problems

Aproximativ N/P elemente

Dacă N nu se divide perfect la P?

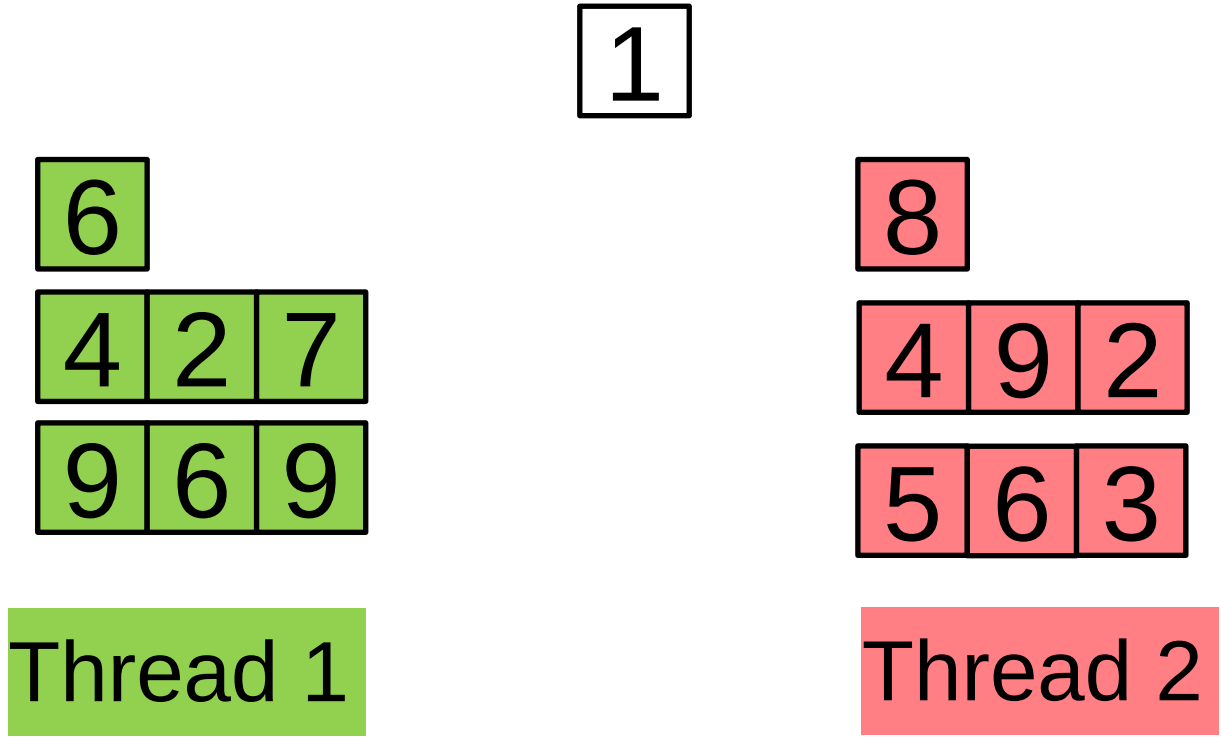
Thread 1

Thread 2

Embarrassingly parallel problems

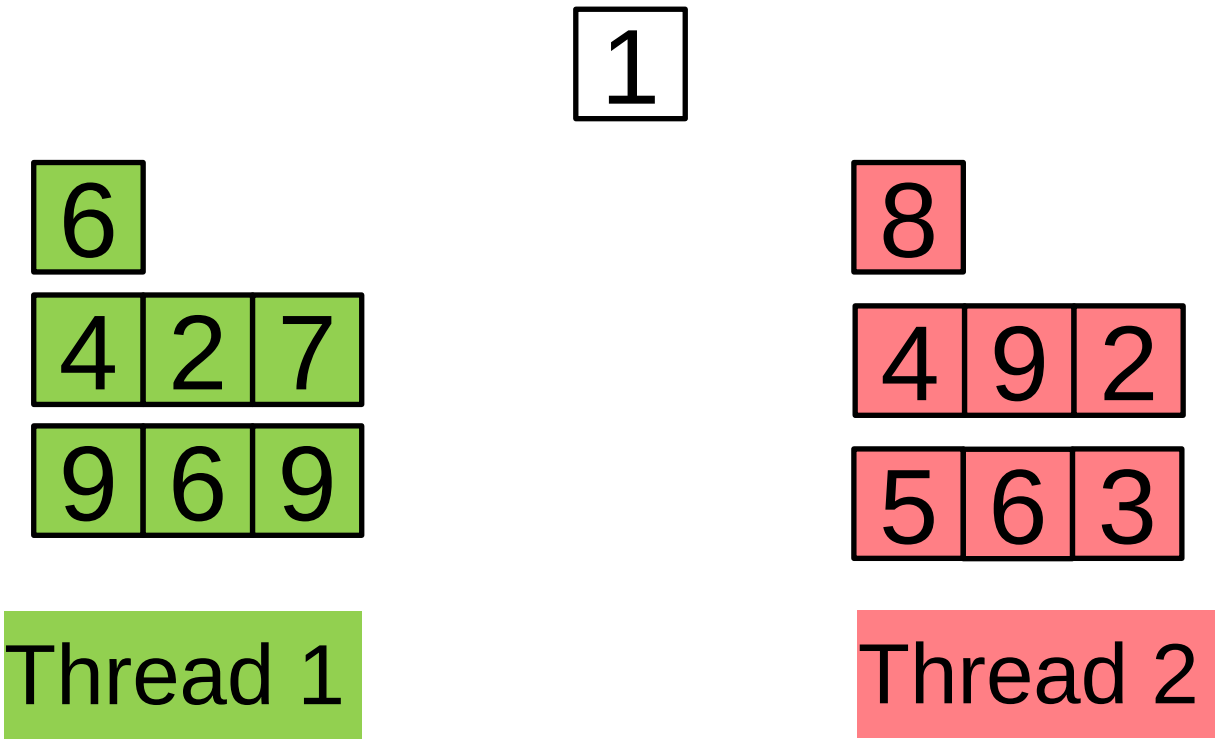
Aproximativ N/P elemente

Dacă N nu se divide perfect la P?



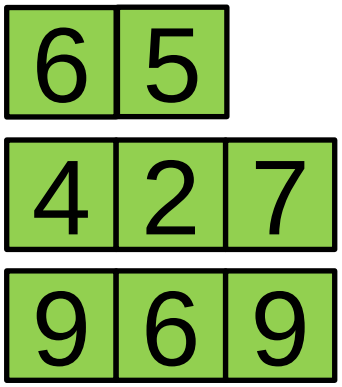
Embarrassingly parallel problems

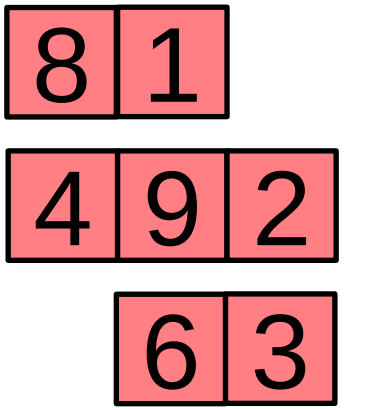
floor(N/P) elemente **floor(15/2) = 7**



Embarrassingly parallel problems

ceil(N/P) elemente **ceil(15/2) = 8**





Thread 1

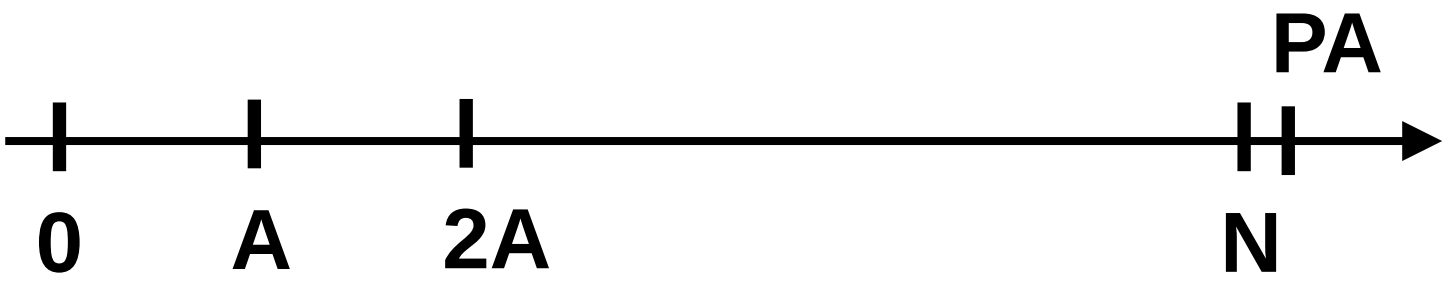
Thread 2

Embarrassingly parallel problems

A = floor(N/P)

Embarrassingly parallel problems

A = ceil(N/P)



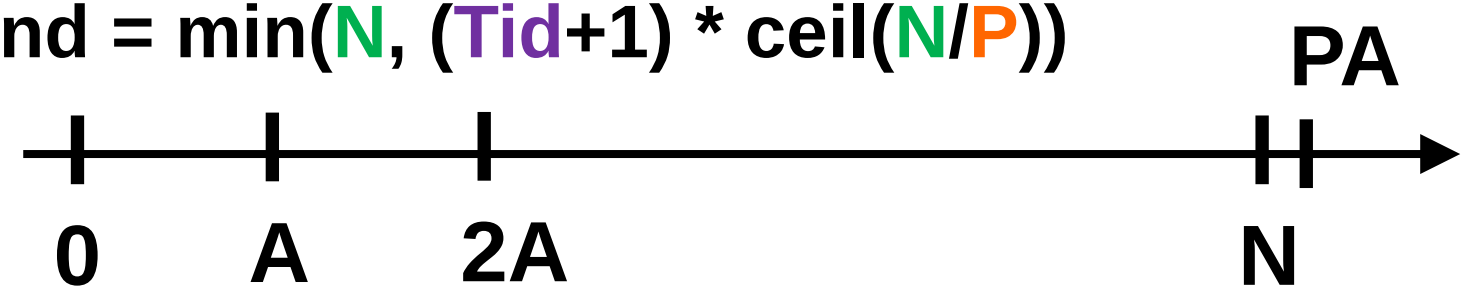
Embarrassingly parallel problems

Formule elegante:

Tid este identificator de thread, are valori de la 0 la P

start = Tid * ceil(N/P)

end = min(N, (Tid+1) * ceil(N/P))



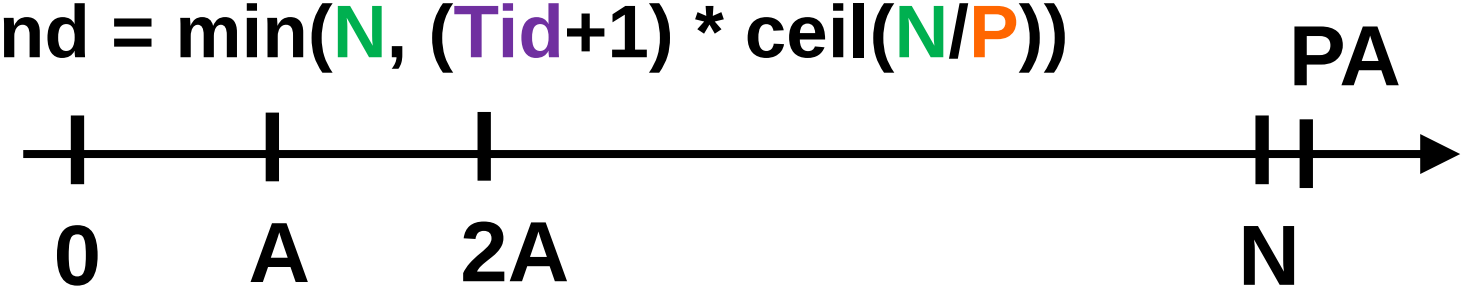
Embarrassingly parallel problems

Formule elegante:

Tid este identificator de thread, are valori de la 0 la P

start = Tid * ceil(N/P)

end = min(N, (Tid+1) * ceil(N/P))



Funcționează și:

start = round(Tid * N/P)

end = round((Tid+1) * N/P) **De ce?**