Avoiding Deadlock

Computer Systems

David Marchant

Based on slides by:

Randal E. Bryant and David R. O'Hallaron

Deadlock Avoidance

- **Deadlock can occur any time we have a blocking operation**
  - Network communications
  - Mutexes
- **Often time can be hidden in testing by buffers**
- **But if it CAN occur, we MUST assume it WILL**
- **Today we will look again at avoiding it locally, as well as across networks**

One worry: Races

- A *race* occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y

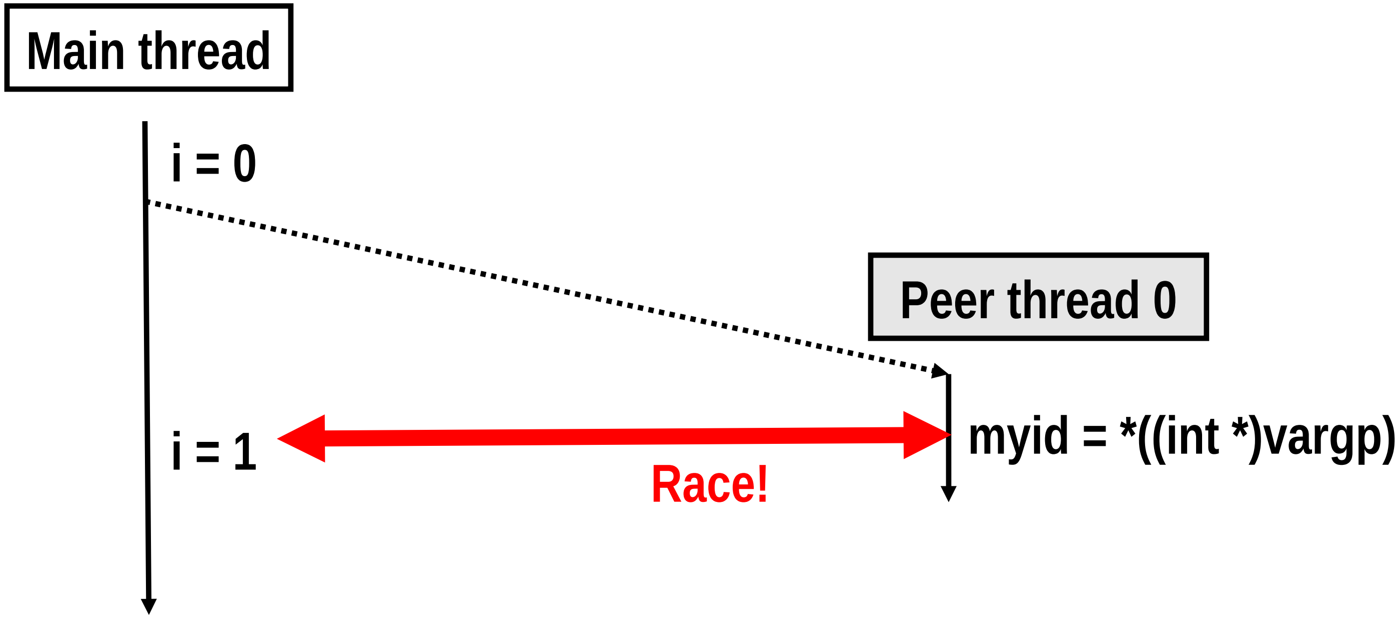
```
/* A threaded program with a race */
int main()
{
    pthread_t tid[N];
    int i;

    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

N threads are sharing i

race.c

Race Illustration

```
for (i = 0; i < N; i++)
    Pthread_create(&tid[i], NULL, thread, &i);
```



- **Race between increment of i in main thread and deref of vargp in peer thread:**
  - If deref happens while i = 0, then OK
  - Otherwise, peer thread gets wrong id value

Race Elimination

```c
/* Threaded program without the race */
int main()
{
    pthread_t tid[N];
    int i, *ptr;

    for (i = 0; i < N; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, thread, ptr);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

norace.c

- **Avoid unintended sharing of state**

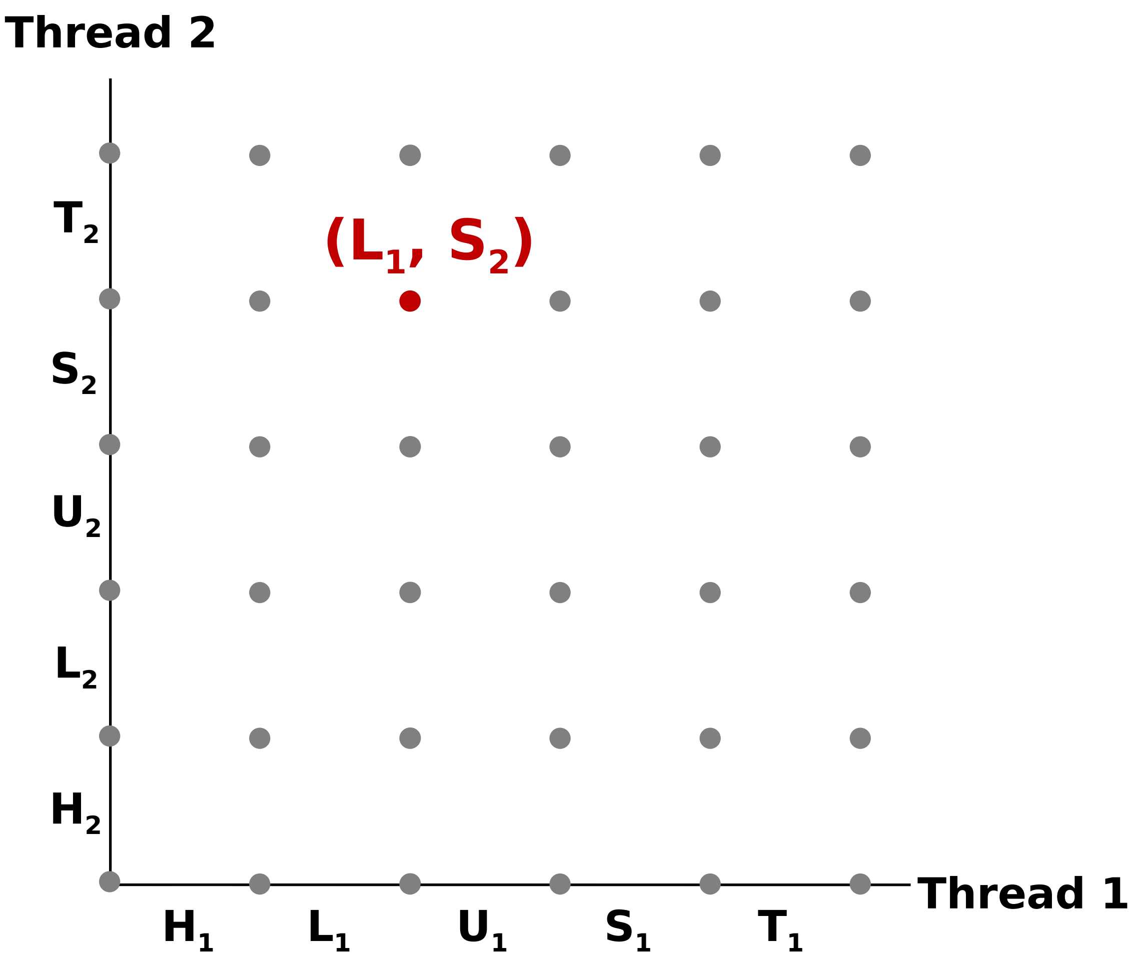
Deadlocking With Semaphores

```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 0 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
      P(&mutex[id]); P(&mutex[1-id]);
      cnt++;
      V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:	Tid[1]:
$P(s_0)$;	$P(s_1)$;
$P(s_1)$;	$P(s_0)$;
cnt++;	cnt++;
$V(s_0)$;	$V(s_1)$;
$V(s_1)$;	$V(s_0)$;

Progress Graphs



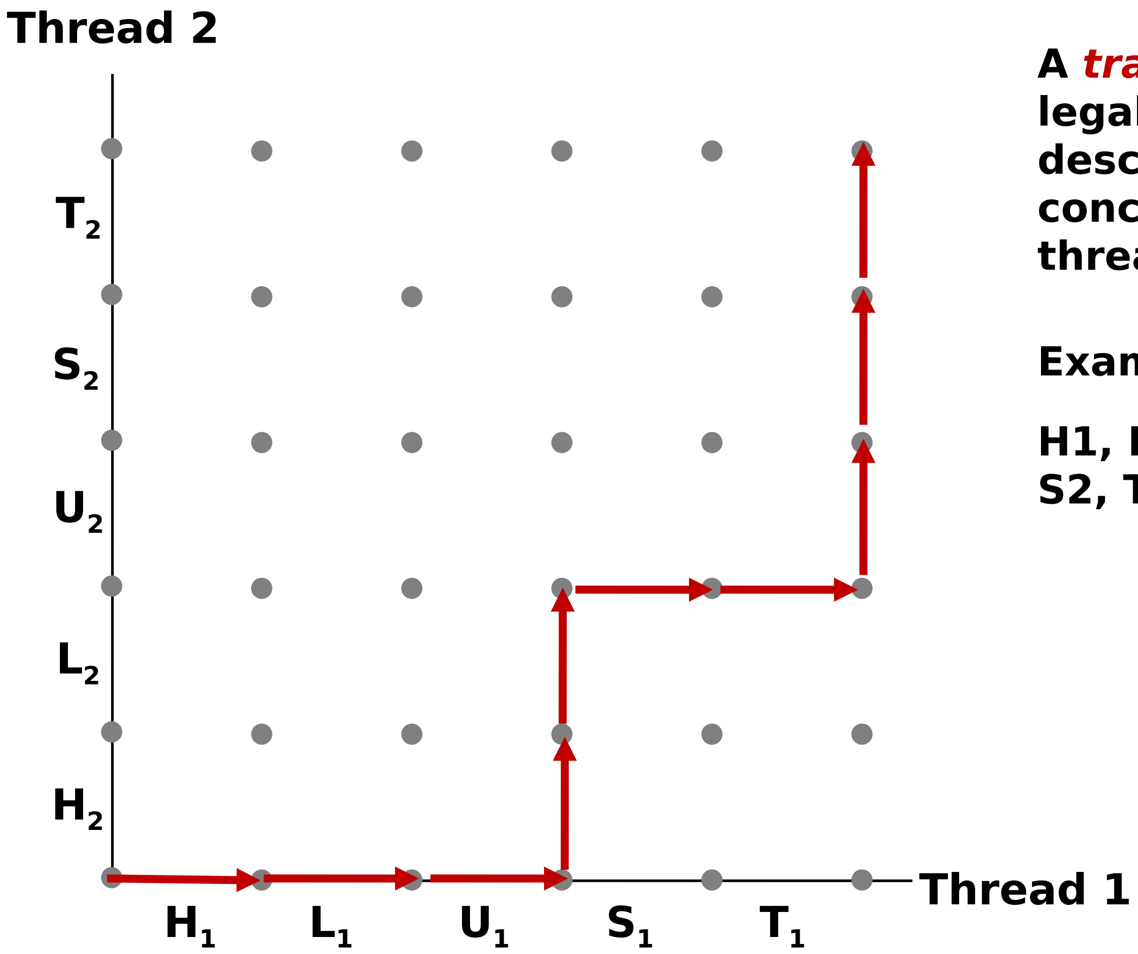
A *progress graph* depicts the discrete *execution state space* of concurrent threads.

Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible *execution state* ($Inst_1$, $Inst_2$).

E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.

Trajectories in Progress Graphs



A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Enforcing Mutual Exclusion

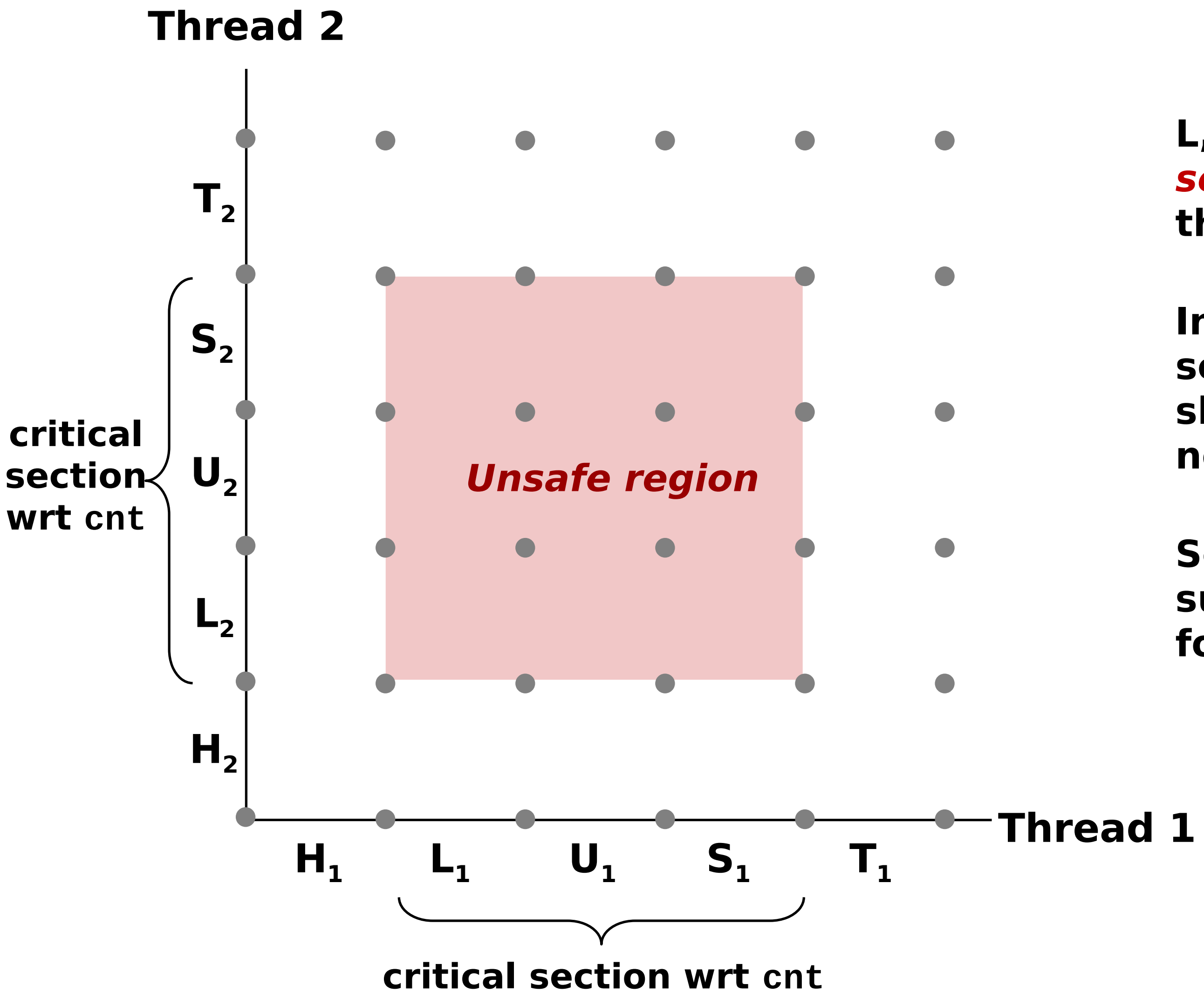
- ***Question:*** **How can we guarantee a safe trajectory?**

- **Answer: We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.**
  - i.e., need to guarantee ***mutually exclusive access*** for each critical section.

- **Classic solution:**
  - Semaphores (Edsger Dijkstra)

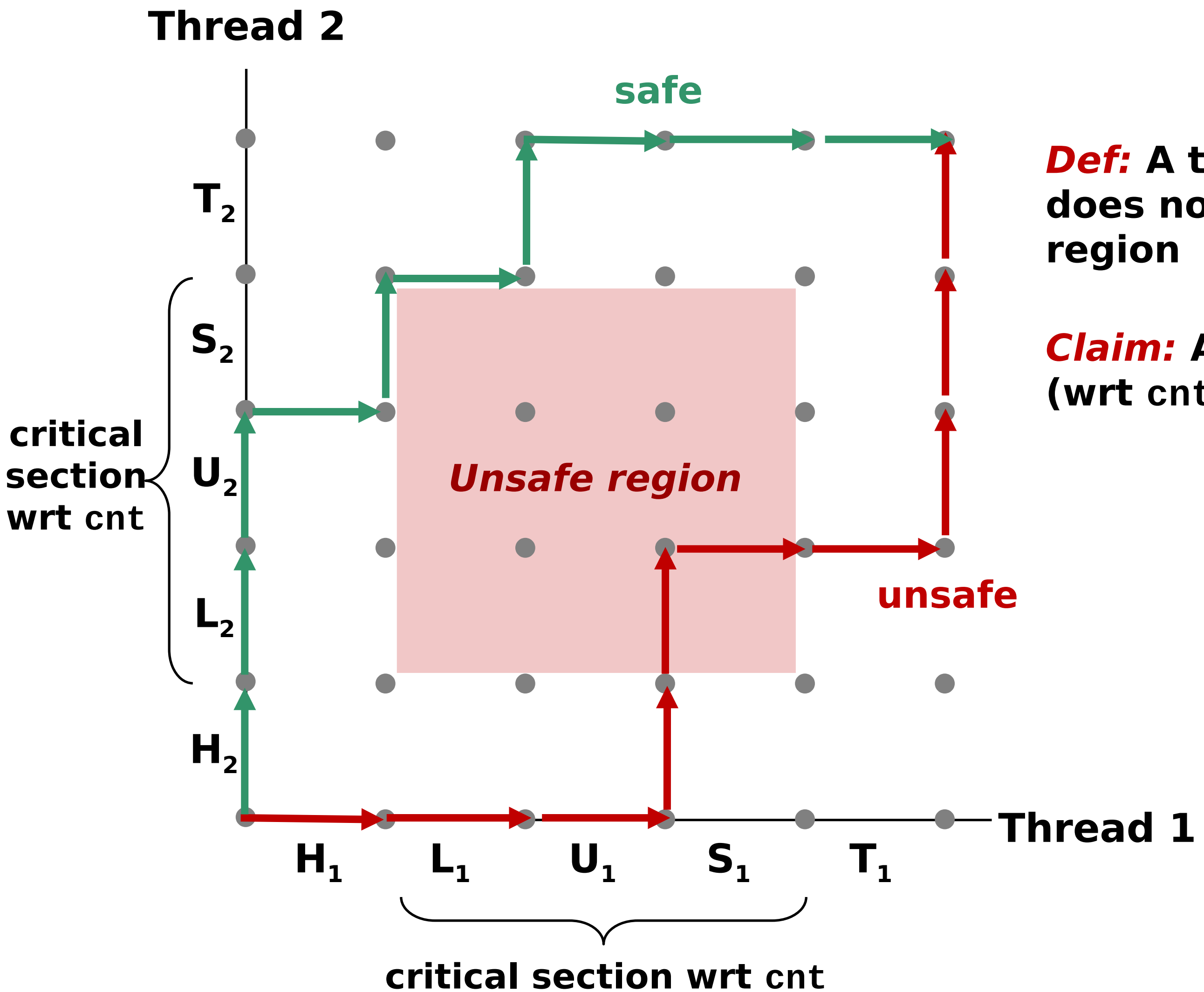
- **Other approaches**
  - Mutexes and condition variables from Pthreads
  - Monitors (Java) (boring languages are outside our scope)

Critical Sections and Unsafe Regions



- L, U, and S form a ***critical section*** with respect to the shared variable `cnt`
- Instructions in critical sections (wrt. some shared variable) should not be interleaved
- Sets of states where such interleaving occurs form ***unsafe regions***

Critical Sections and Unsafe Regions



Def: **A trajectory is *safe* iff it does not enter any unsafe region**

Claim: **A trajectory is correct (wrt `cnt`) iff it is safe**

Deadlocking With Semaphores

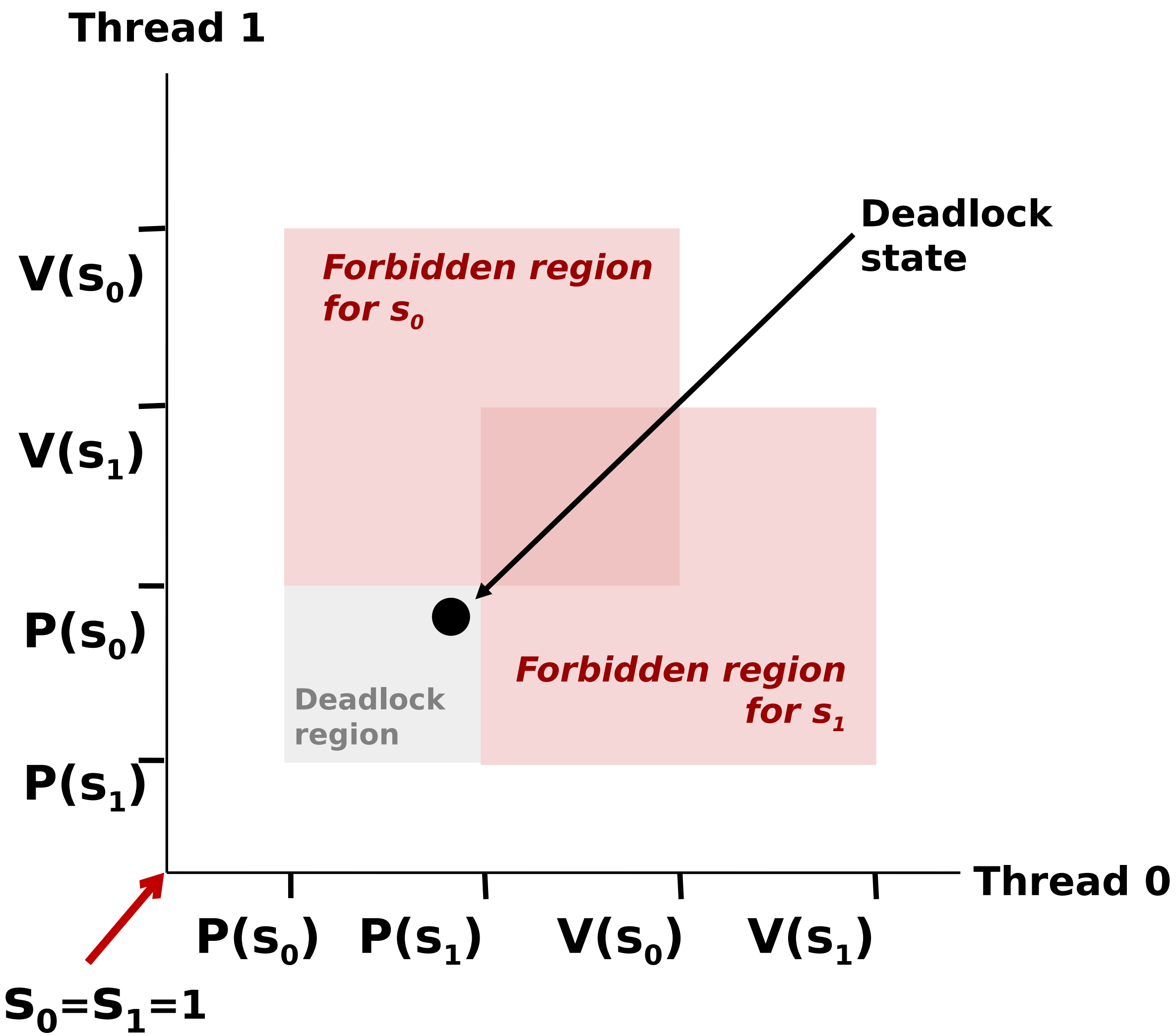
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 0 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
      P(&mutex[id]); P(&mutex[1-id]);
      cnt++;
      V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:
$P(s_0)$;
$P(s_1)$;
cnt++;
$V(s_0)$;
$V(s_1)$;

Tid[1]:
$P(s_1)$;
$P(s_0)$;
cnt++;
$V(s_1)$;
$V(s_0)$;

Deadlock Visualized in Progress Graph



Locking introduces the potential for ***deadlock:*** waiting for a condition that will never be true

Any trajectory that enters the ***deadlock region*** will eventually reach the ***deadlock state,*** waiting for either S_0 or S_1 to become nonzero

Other trajectories luck out and skirt the deadlock region

Unfortunate fact: deadlock is often nondeterministic (race)

Avoiding Deadlock

Acquire shared resources in same order

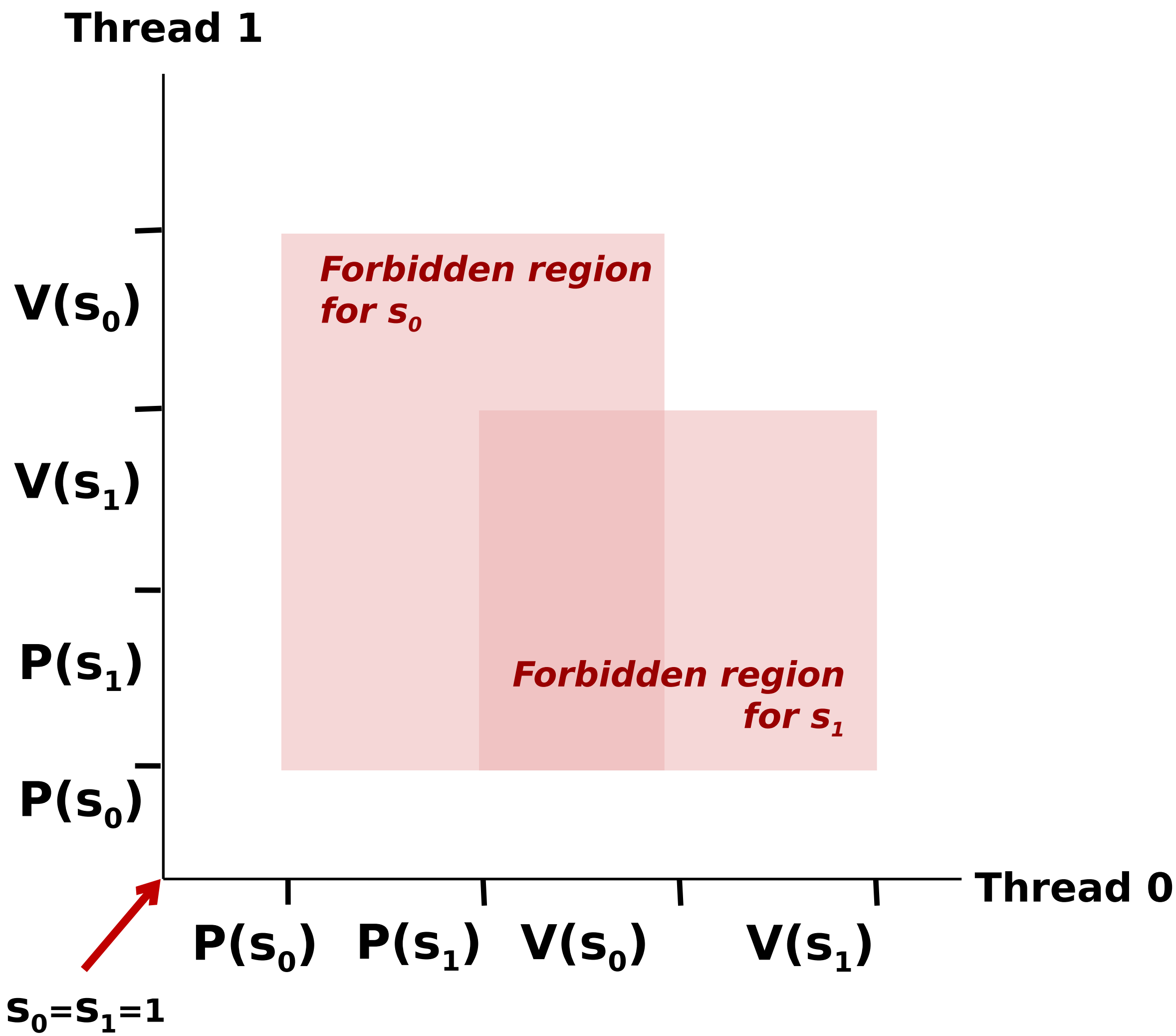
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 0 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 0 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
    cnt++;
    V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:
P(s0);
P(s1);
cnt++;
V(s0);
V(s1);

Tid[1]:
P(s0);
P(s1);
cnt++;
V(s1);
V(s0);

Avoided Deadlock in Progress Graph



No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial

Drawing a process graph

- **Easy to draw! (mostly).** Most applications are ‘symetrical’, and if they aren’t they probably should be.

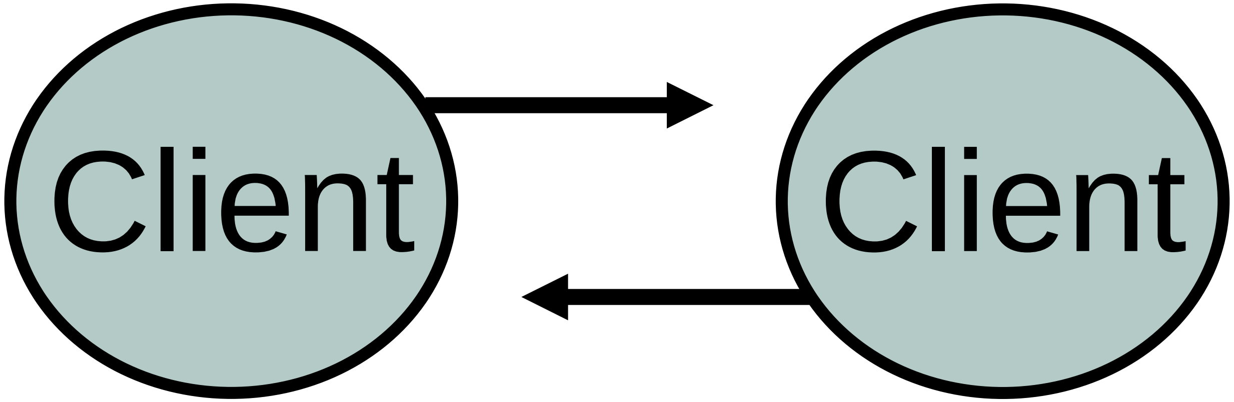
- **We can ignore any non-blocking code.** Anything that completes in a finite time can be ignored.

- **We only need to plot two axis’ (mostly).** It doesn’t matter if we have 2 processes or 2 million. The same logical dependency exists between them.

- **When in doubt, draw a diagram!** We often can’t prove that we have avoided deadlock, but a diagram can be a short-hand for showing how its impossible, *if our diagram reflect our code*.

Deadlock isn’t just local

- **You cannot mutex over a network**
- **But you can have two hosts reading/writing which will act in the same way**
- **The client-server model is used entirely to escape this problem**
- **Communication Sequential Processes (CSP)**



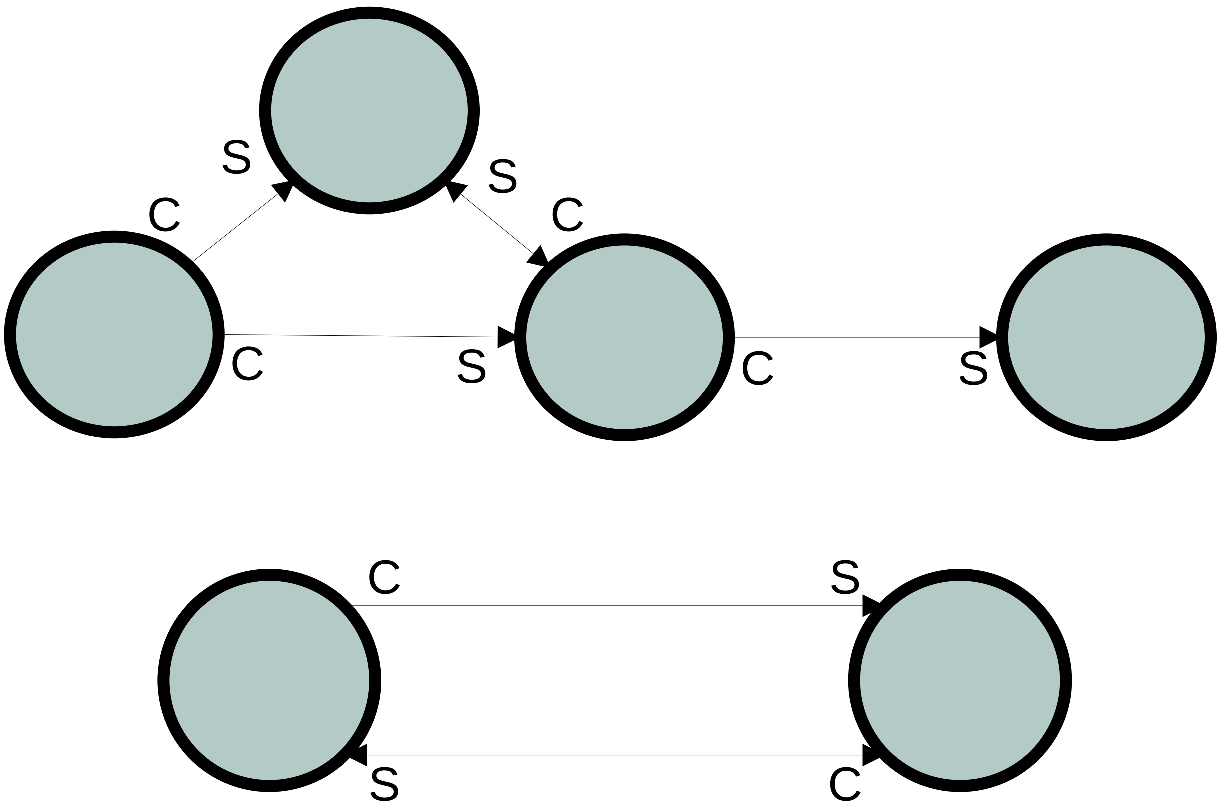
CSP

- **Communication Sequential Processes**
- **Proposed in 1978 by Tony Hoare**
- **Formal mathematical language for describing concurrent processes and their interactions**
  - Can *guarantee* no deadlock
  - Can *identify* livelock
- **Not a programming language, but principles are used in many contemporary languages such as Go**

Process Diagrams

- No formal definition for these diagrams or how they look
- Two components, processes and channels
- A process can represent an OS process, OS thread, network host, or any other sequential code
- A channel is a connection between processes, and may be mono- or bi-directional
- Can be helpful to label client and server ends of a channel

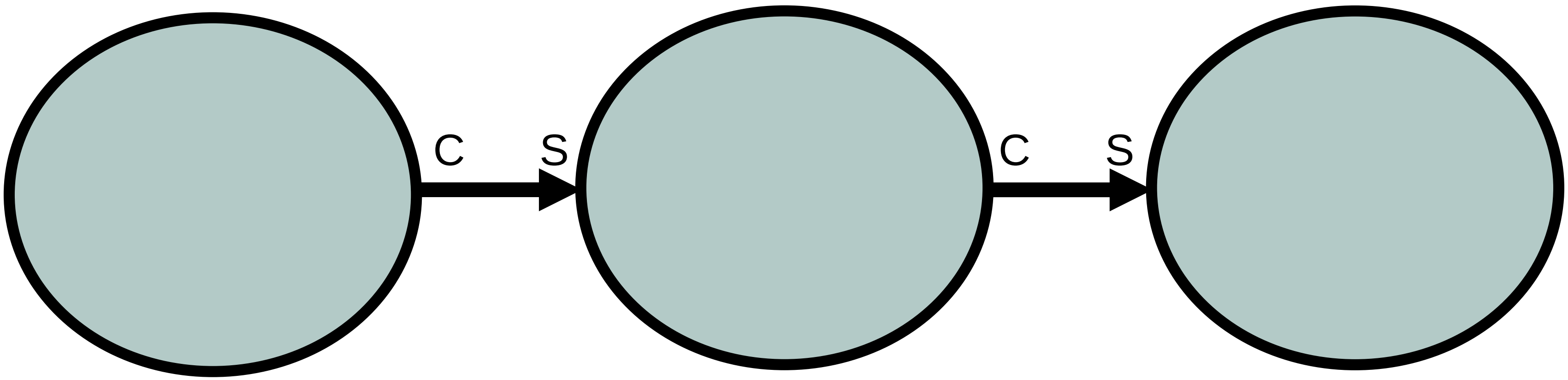
Process Diagrams



What does this get us

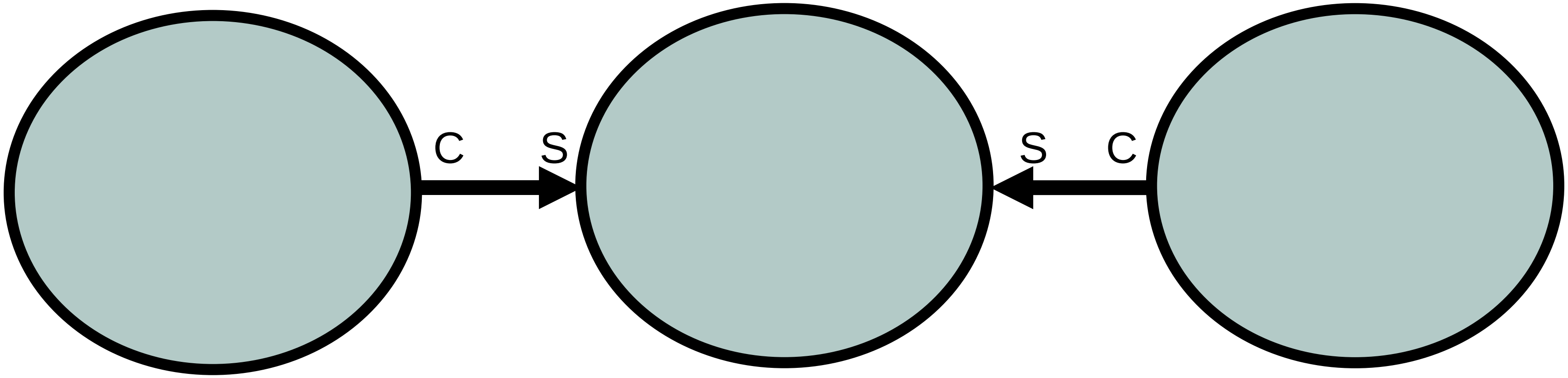
- **CSP is the source of the client-server model**
- **Semantically, in the client server model:**
  - Only clients initiate communications
  - If a client expects a response, a server will provide one in a finite amount of time
  - If a client expects a response, it will be immediately ready to receive it.
- **We have (hopefully) been keeping to this already**
- **If we can draw all channel interactions, we can understand all blocking points**
- **Any loop of client-server interactions has the potential to deadlock**

What about Deadlock between hosts?



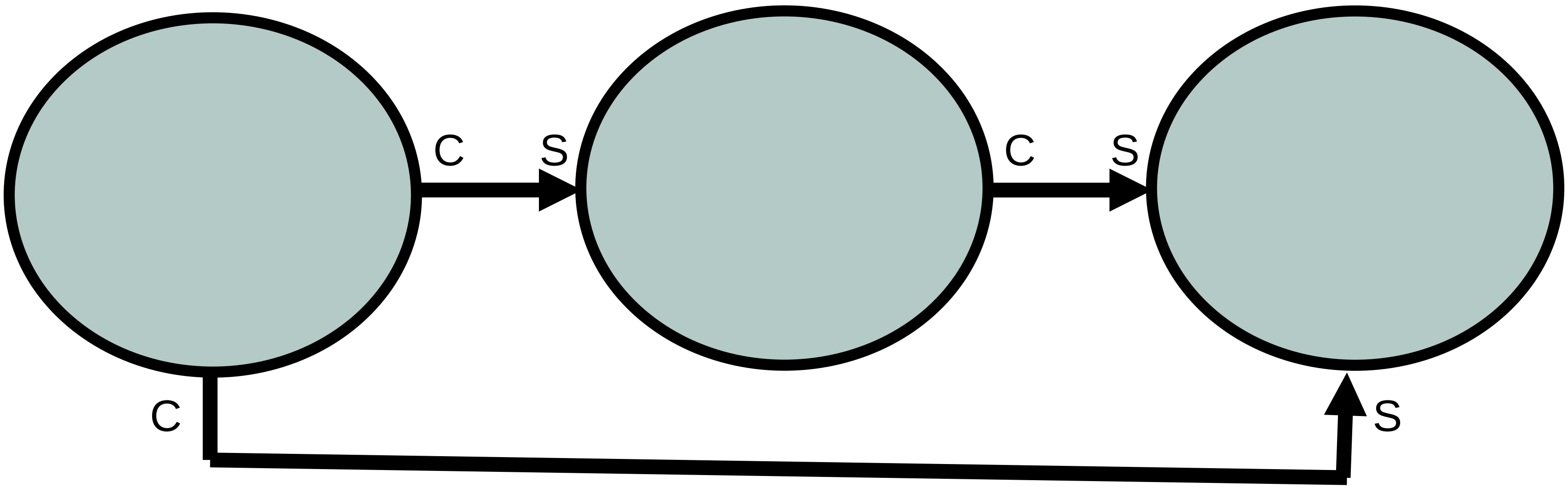
No deadlock

What about Deadlock between hosts?



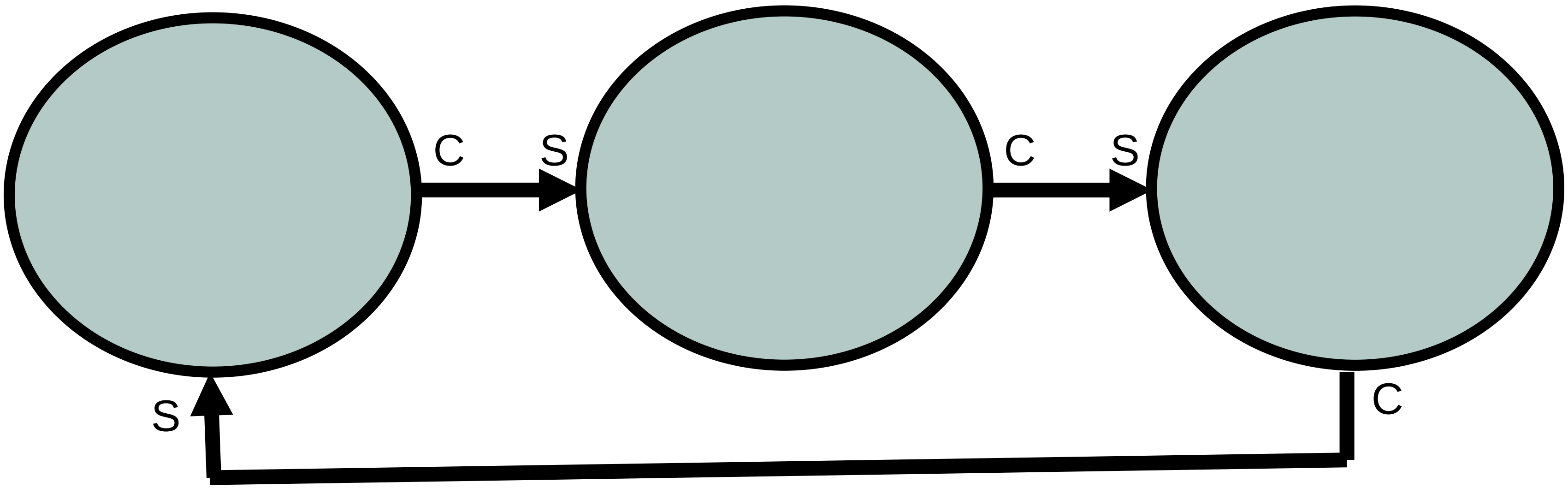
No deadlock

What about Deadlock between hosts?



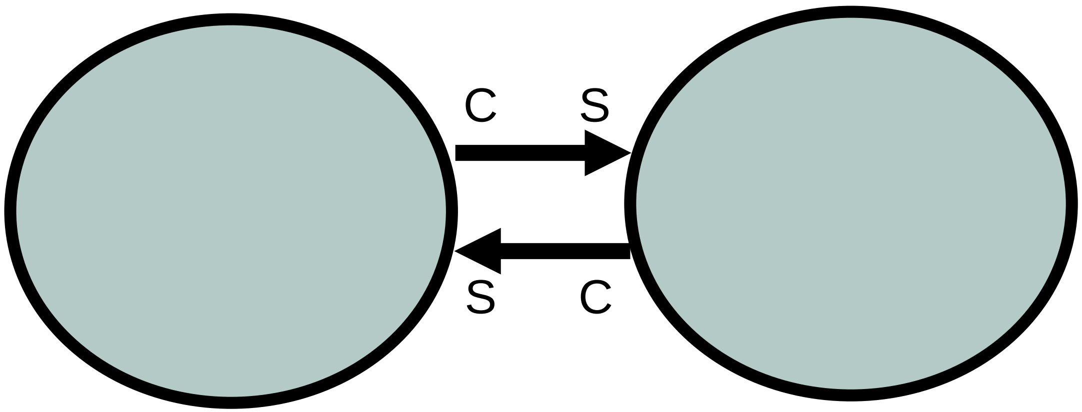
No deadlock

What about Deadlock between hosts?



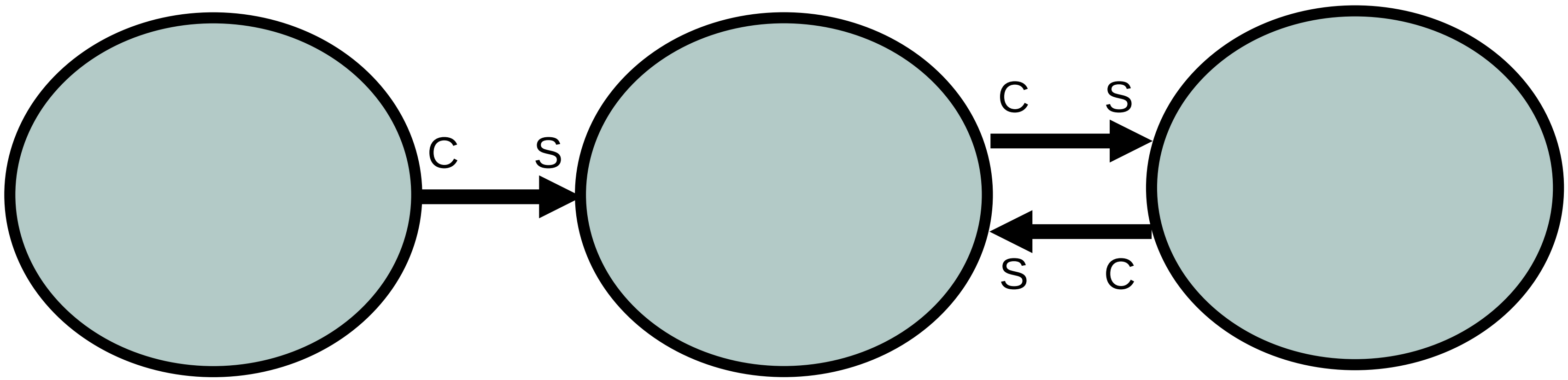
Deadlock

What about Deadlock between hosts?



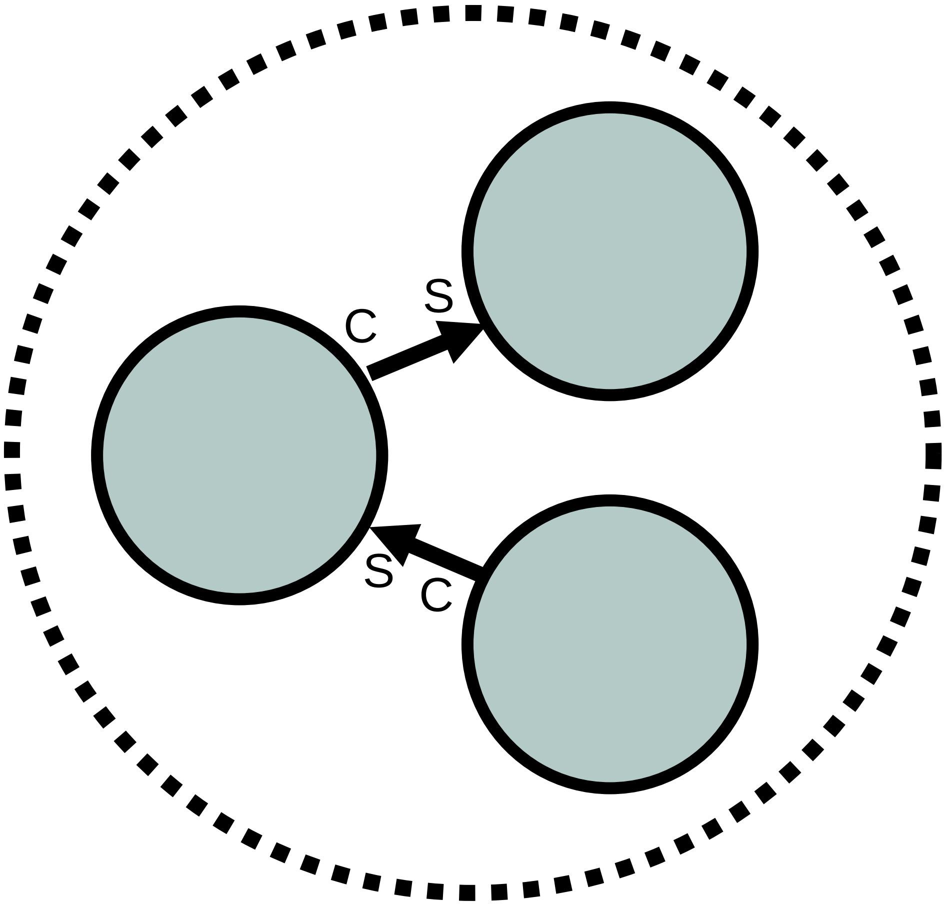
Deadlock

What about Deadlock between hosts?



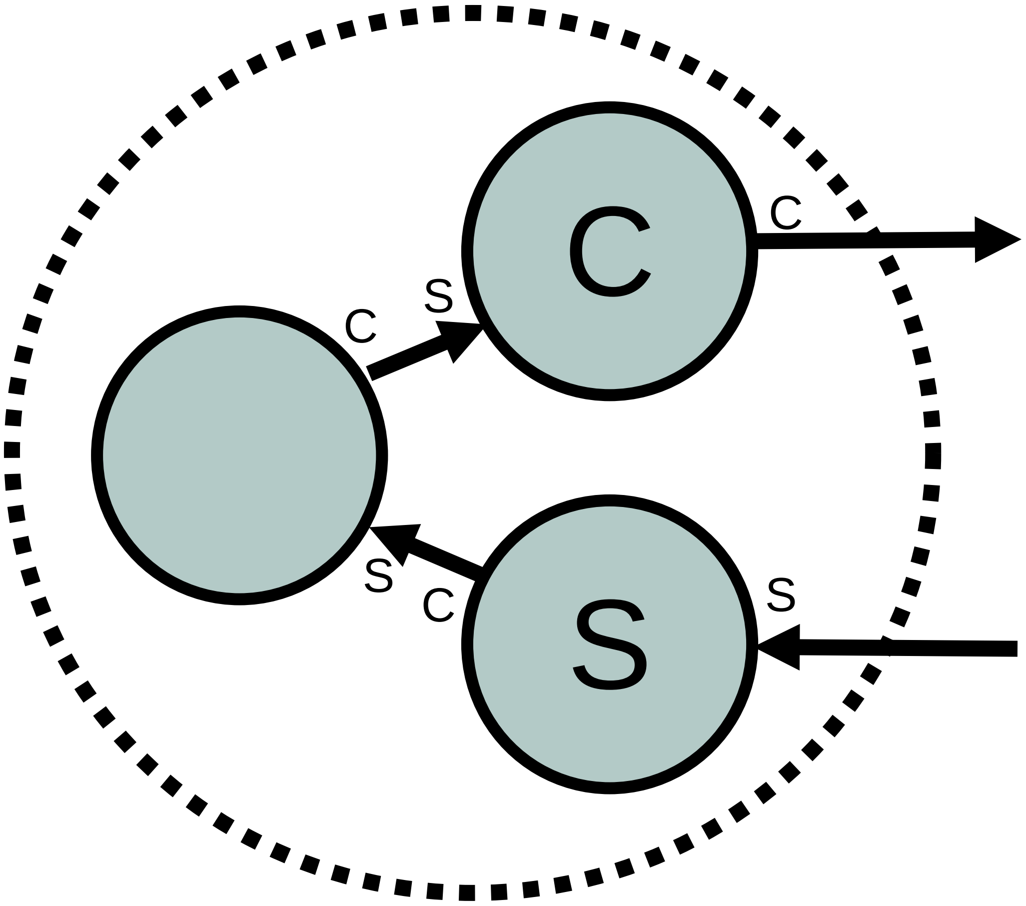
Deadlock

Didn’t we solve this already?



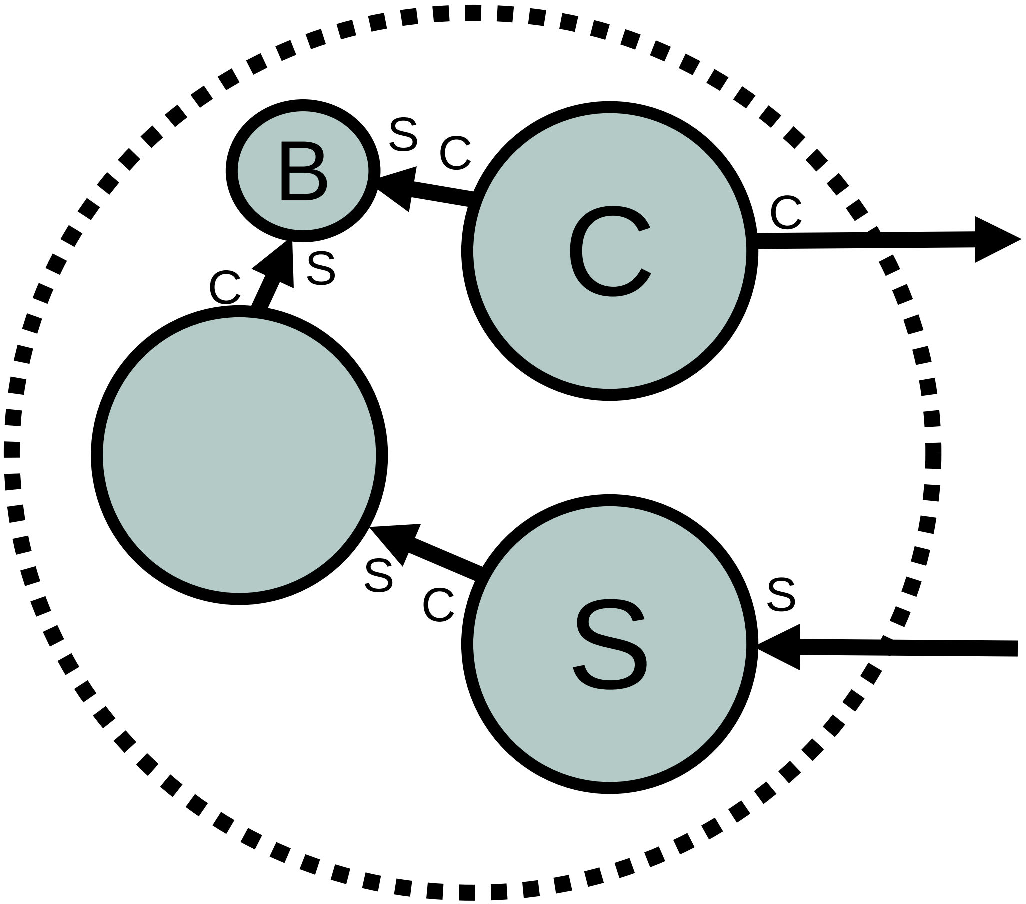
No deadlock

Didn't we solve this already?



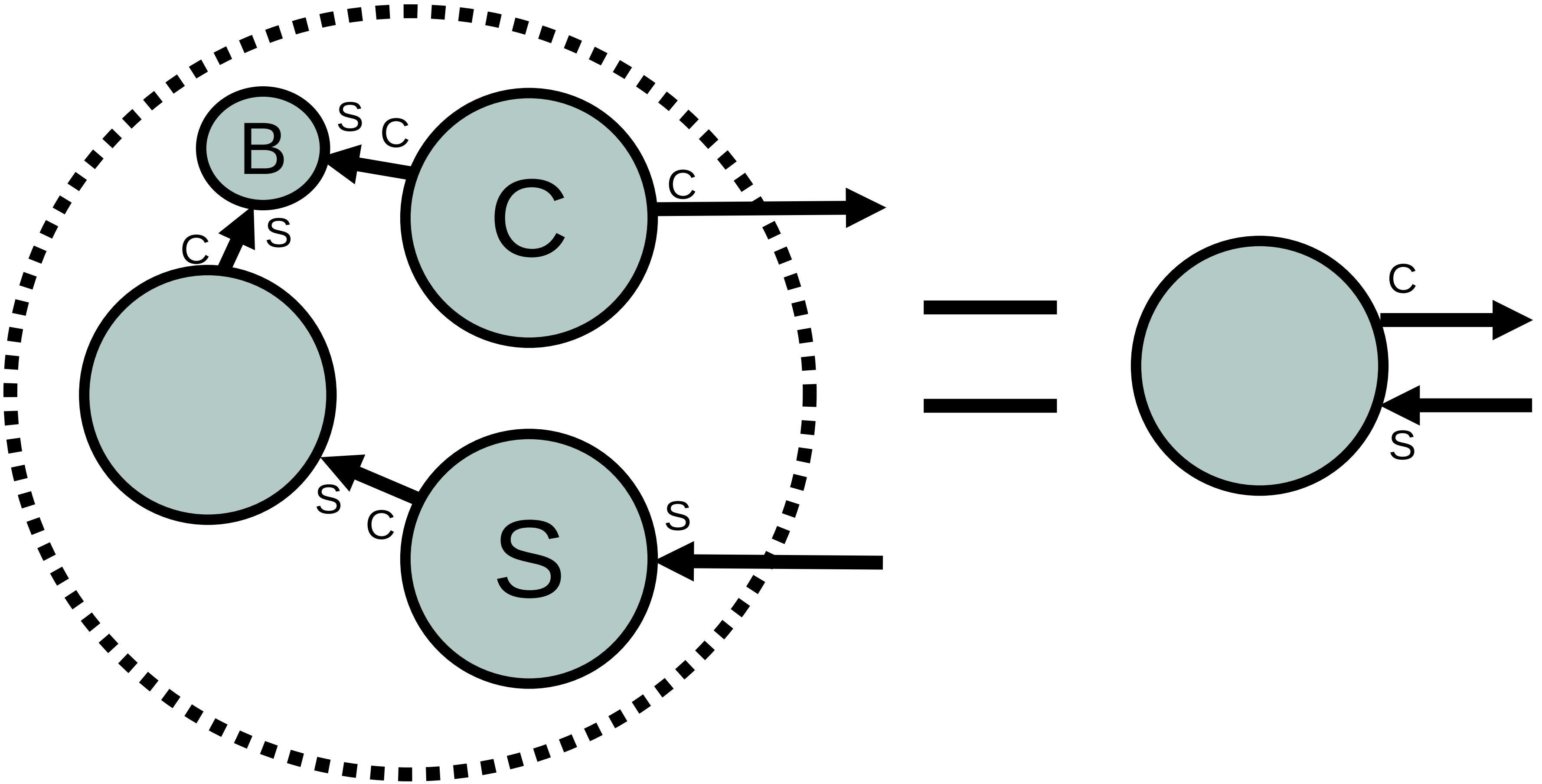
Maybe Deadlock deadlock?

Didn't we solve this already?



No deadlock

Why the dotted circle?



Some notes …

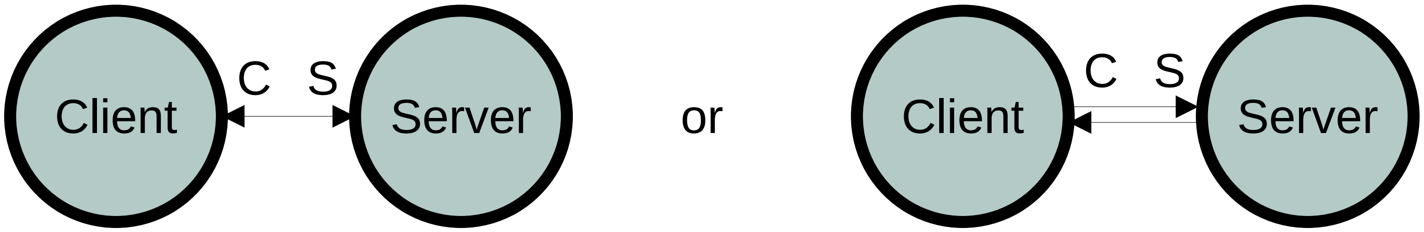
- A client/server loop doesn't actually mean deadlock
- But no client/server loop does guarantee no deadlock
- Depending on internal structure of a process, deadlock might not occur
- But the road to deadlock is paved by good intentions
- **Any client/server loop MUST be carefully examined and justified**

Translating code into diagrams

- These are more of a sketch, than a true reflection so some ambiguity is inevitable
- Only need to worry about blocking operations, external read and writes
- Concurrent connections should be shown as separate connections
- Sequential ones can be grouped
- Label channel ends (do as I say, not as I do)

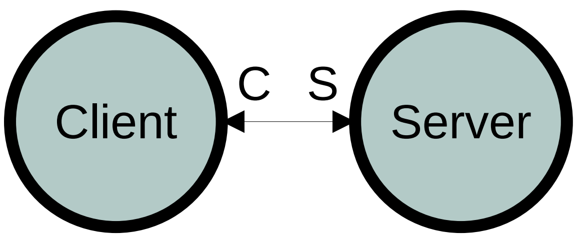
Translating code into diagrams

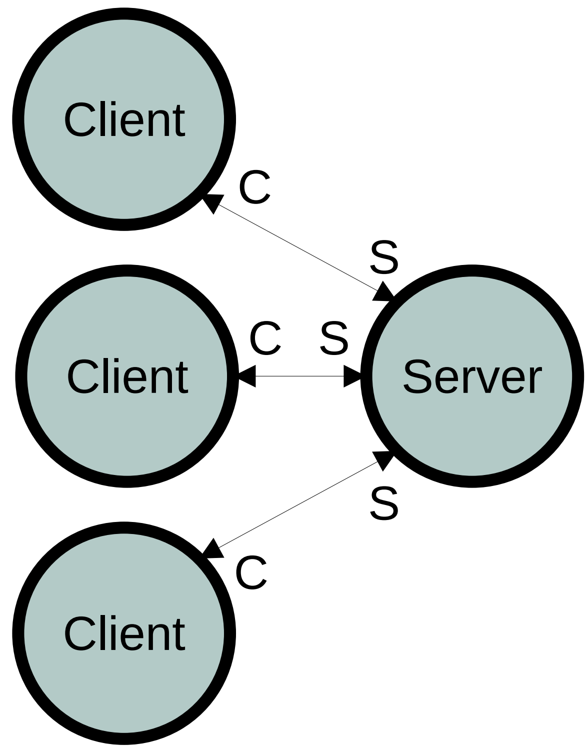
- **Lets draw A3**
- **A3 Server**
  - Listens for connections
  - Can handle parallel connections
  - Always responds
  - No additional comms from it
- **A3 Client**
  - Connects to the server
  - Sends two message types (register, get)
  - Each message is sequential



Translating code into diagrams

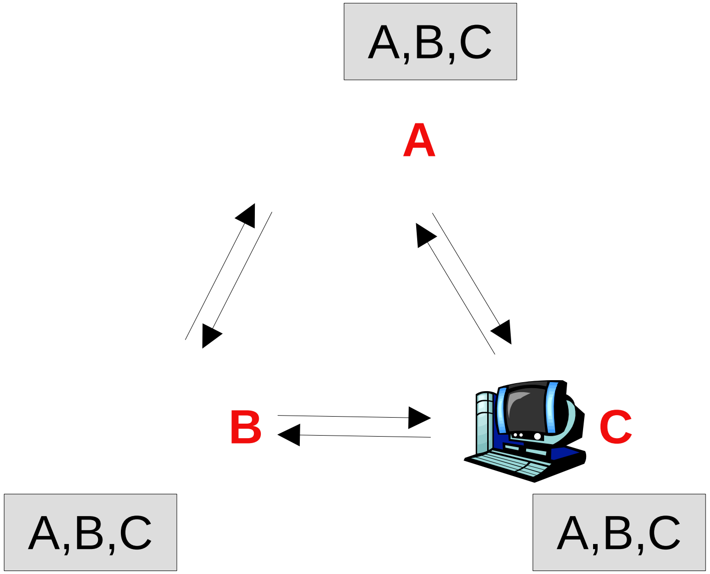
- Identical interactions can *often* be left out
- As each connection is served concurrently, we can effectively add infinite clients and nothing changes
- This might change if there are dependencies between connections





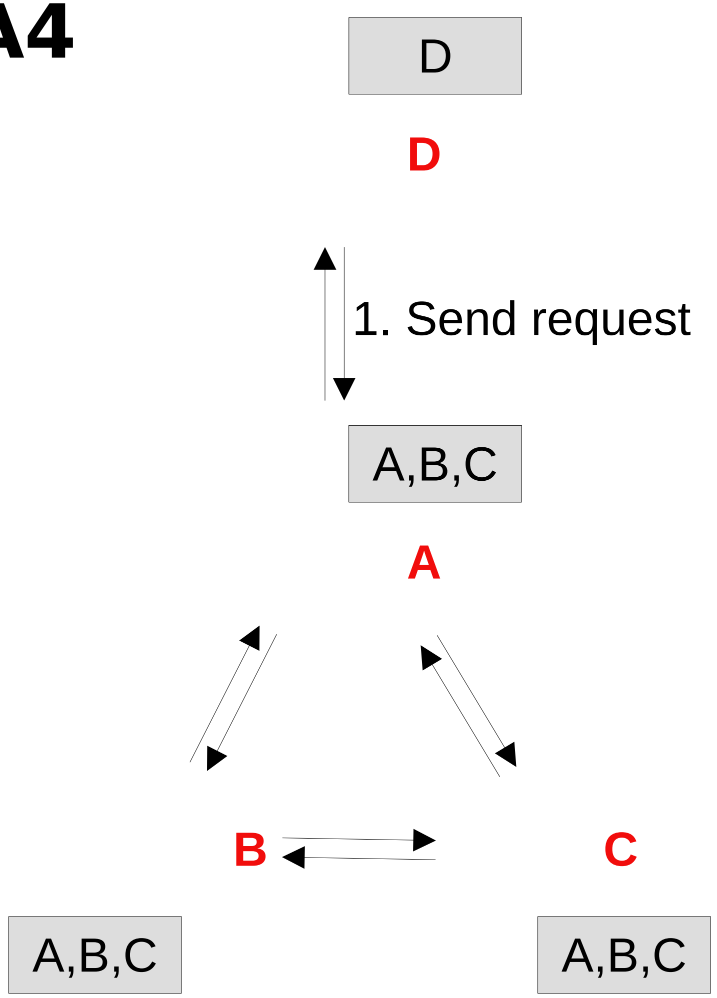
Peer to Peer Network in A4

- P2P is a way to share data files
- Peers connect to a network by registering with someone already on it
- Each Peer will attempt to maintain a list of everyone on the network
- If a peer gets a request to join, it will inform all the peers it knows about



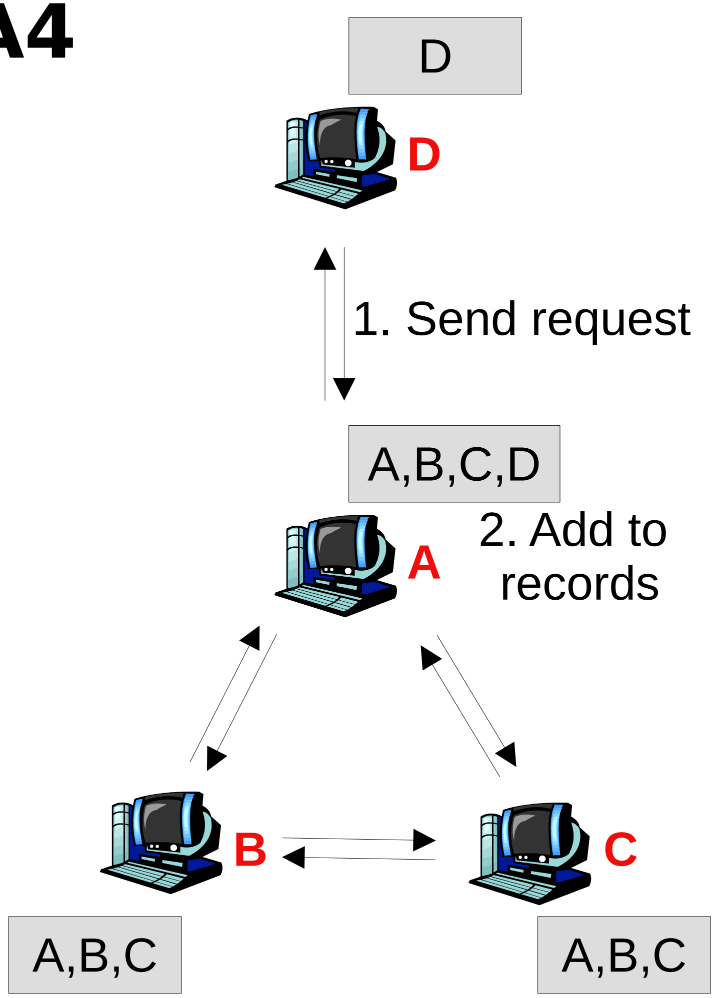
Peer to Peer Network in A4

- P2P is a way to share data files
- Peers connect to a network by registering with someone already on it
- Each Peer will attempt to maintain a list of everyone on the network
- If a peer gets a request to join, it will inform all the peers it knows about



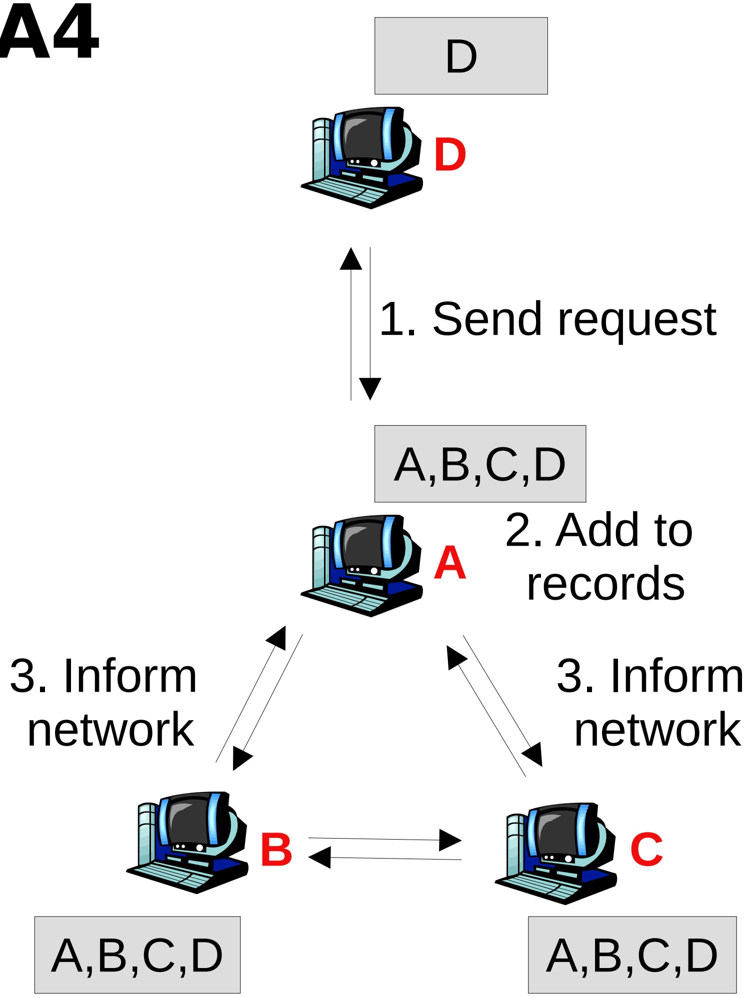
Peer to Peer Network in A4

- P2P is a way to share data files
- Peers connect to a network by registering with someone already on it
- Each Peer will attempt to maintain a list of everyone on the network
- If a peer gets a request to join, it will inform all the peers it knows about



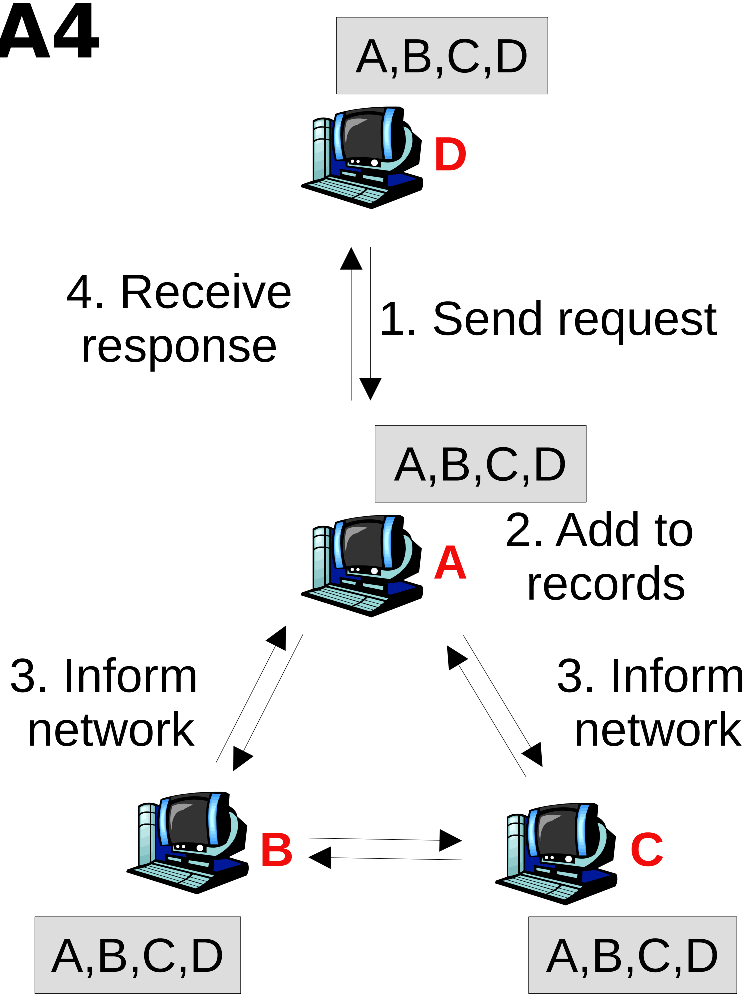
Peer to Peer Network in A4

- P2P is a way to share data files
- Peers connect to a network by registering with someone already on it
- Each Peer will attempt to maintain a list of everyone on the network
- If a peer gets a request to join, it will inform all the peers it knows about



Peer to Peer Network in A4

- P2P is a way to share data files
- Peers connect to a network by registering with someone already on it
- Each Peer will attempt to maintain a list of everyone on the network
- If a peer gets a request to join, it will inform all the peers it knows about



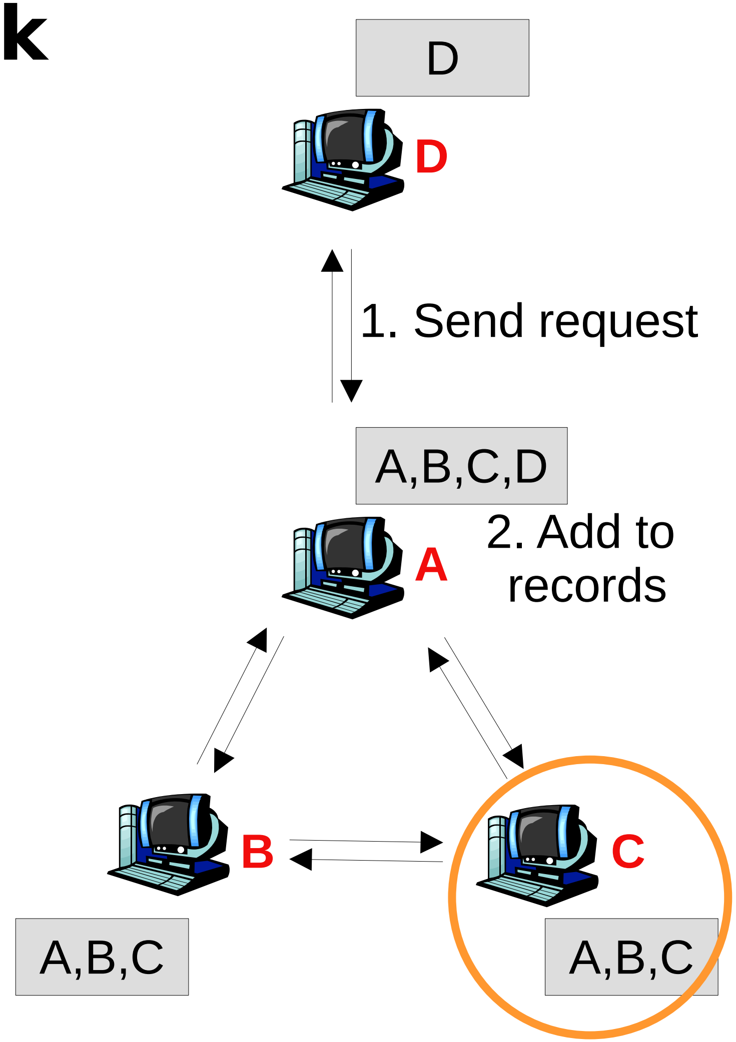
Races, across the network

- Recall that races occur when the outcome depends on the arbitrary ordering of interactions
- Fixed locally with locks, or by not sharing in the first place
- Global variable races don’t occur as no global memory
- Locks do not exist at a network level (mostly)
  - Could centralise vital info, but this is sloooow
- Many (not all) races can be coped with
- Up to applications to avoid/cope with races as they occur

Races, across the network

- Consider if at this point, C wants to get a file, it only sees A and B as peers
- Race, as D *should* be included but is not yet
- Does this really matter though?
- For selecting a peer, maybe not
- If we needed a report of the complete network, maybe

- Solving this problem is out of scope, and can lead to lots of fun solutions (Santa problem)



Some conclusions

- Deadlock and races are as bad in networking as they are in multiprocessing/threading
- Races tend not to occur as no global memory
- Deadlock can very much occur both locally and remotely
- Use diagrams to debug the structure of your code
- A diagram is only useful if it reflects your implementation