Server Performance

Computer Systems

David Marchant

Based on slides by:

Randal E. Bryant and David R. O'Hallaron

Before we begin

- In previous years this lecture has covered IO multiplexing.
- This is interesting, but won't come up on the exam so we aren't going to cover it, but the slides are still here if you are interested.
- Instead we are going to focus on designing blocking systems, especially to avoid deadlock and races.
- This may be directly applicable to A4, and the exam (not a hint)

Some reminders…

- **Threads & Processes:** Both can maintain concurrent logical control through context switching. Threads are lighter weight and share more data.

- **Concurrency & Parrallel:** Parallel means running different processing at the literal same time. Concurrency can simulate this by interweaving

- **Semaphores & Mutex:** Synchronisation tools to ensure that we don’t encounter concurrency problems such as race conditions or deadlock

Approaches for Writing Concurrent Servers

Allow server to handle multiple clients concurrently

1. Process-based (Intro to Network Programming Lecture)

- Kernel automatically interleaves multiple logical flows
- Each flow has its own private address space

2. Event-based

- Programmer manually interleaves multiple logical flows
- All flows share the same address space
- Uses technique called *I/O multiplexing*.

3. Thread-based (Intro to Network Programming Lecture)

- Kernel automatically interleaves multiple logical flows
- Each flow shares the same address space
- Hybrid of of process-based and event-based.

Pros and Cons of Process-based Servers

- **+ Handle multiple connections concurrently**
- **+ Clean sharing model**
  - descriptors (no)
  - file tables (yes)
  - global variables (no)
- **+ Simple and straightforward**
- **– Additional overhead for process control**
- **– Nontrivial to share data between processes**
  - Requires IPC (interprocess communication) mechanisms
    - FIFO’s (named pipes), System V shared memory and semaphores

Pros and Cons of Thread-Based Designs

- **+ Easy to share data structures between threads**
  - e.g., logging information, file cache
- **+ Threads are more efficient than processes**

- **– Unintentional sharing can introduce subtle and hard-to-reproduce errors!**
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - Hard to know which data shared & which private
  - Hard to detect by testing
    - Probability of bad race outcome very low
    - But nonzero!
  - Future lectures

Approach #2: Event-based Servers

- **Server maintains set of active connections**
  - Array of connfd's
- **Repeat:**
  - Determine which descriptors (connfd's or listenfd) have pending inputs
    - e.g., using `select` or `epoll` functions
    - arrival of pending input is an *event*
  - If listenfd has input, then `accept` connection
    - and add new connfd to array
  - Service all connfd's with pending inputs

How does this help us?

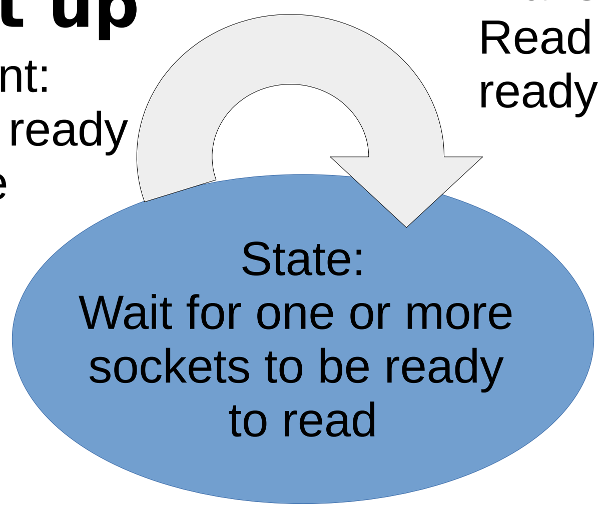
```
int main(int argc, char **argv)
{
    /* Boring declarations go here */

    listenfd = Open_listenfd(argv[1]);
    while (1) {
      clientlen = sizeof(struct sockaddr_storage);
      connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

      Rio_readinitb(&rio, connfd);
      while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", (int)n);
        Rio_writen(connfd, buf, n);
      }
    }
}
```

The key issue is that we want to wait on these two lines at the same time

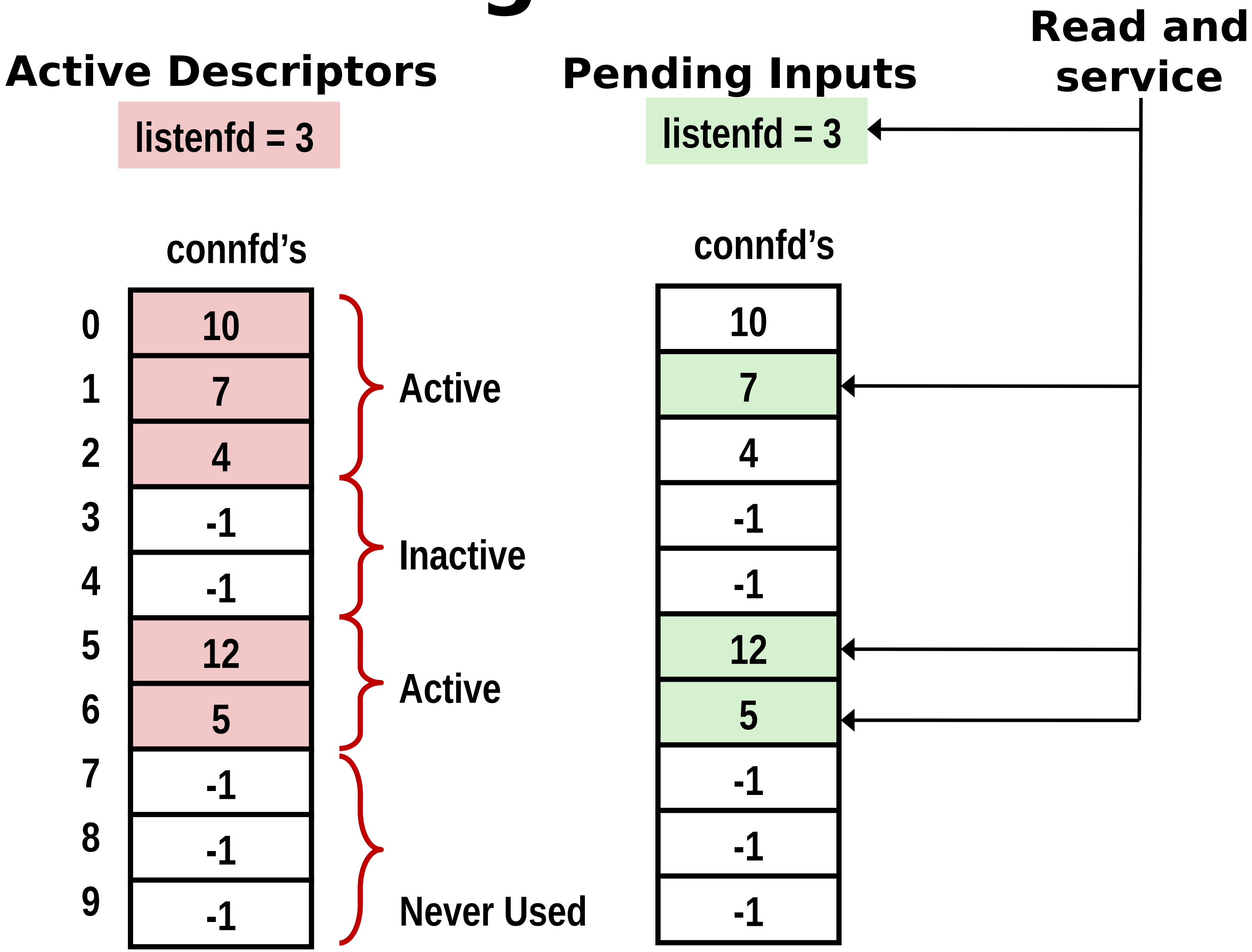
I/O Multiplexing

- **Use a** `select` **command to combine multiple events into a set, then wait for at least one of them to occur**

- **Our set of waitable events will be input through listenfd, plus any connfd’s that have been set up**



I/O Multiplexed Event Processing



Abridged Example (from book, pg 1015)

```
int main(int argc, char **argv) {
  // More boring declarations go here
  fd_set all_socks_set, ready_set;

  listenfd = Open_listenfd(PORT);

  FD_ZERO(&all_socks_set);                 // Clear all_socks_set
  FD_SET(STDIN_FILENO, &all_socks_set); // Add stdin to all_socks_set
  FD_SET(listenfd, &all_socks_set);     // +listenfd to all_socks_set

  while (1) {
    ready_set = all_socks_set;
    Select(listenfd+1, &ready_set, NULL, NULL, NULL);
    if (FD_ISSET(STDIN_FILENO, &ready_set))
      command(); // Read command line from stdin
    if (FD_ISSET(listenfd, &ready_set)) {
      clientlen = sizeof(struct sockaddr_storage);
      connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
      echo(connfd); /* Echo client input until EOF */
      Close(connfd);
}}}
```

Abridged Example (from book, pg 1015)

```
void command(void) {
    char buf[MAXLINE];
    if (!Fgets(buf, MAXLINE, stdin))
        exit(0); /* EOF */
    printf("%s", buf); /* Process the input command */
}
```

```
void echo(int connfd) {
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", (int)n);
    Rio_writen(connfd, buf, n);
    }
}
```

- **Do remember this code is heavily abridged so will not work as is, full code on pg 1015 of BOH**

Abridged Example (from book, pg 1015)

```
int main(int argc, char **argv) {
  // More boring declarations go here
  fd_set all_socks_set, ready_set;

  listenfd = Open_listenfd(PORT);

  FD_ZERO(&all_socks_set);              // Clear all_socks_set
  FD_SET(STDIN_FILENO, &all_socks_set); // Add stdin to all_socks_set
  FD_SET(listenfd, &all_socks_set);     // listenfd to all_socks_set

  while (1) {
    ready_set = all_socks_set;
    Select(listenfd+1, &ready_set, NULL, NULL, NUL
    if (FD_ISSET(STDIN_FILENO, &ready_set))
      command(); // Read command line from stdin
    if (FD_ISSET(listenfd, &ready_set)) {
      clientlen = sizeof(struct sockaddr_storage);
      connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
      echo(connfd); /* Echo client input until EOF */
      Close(connfd);
}}}
```

Our server blocks here but in such a way that we can wait for multiple things to happen and respond accordingly

I/O Multiplexing

- Thats all well and good, but it doesn’t show an example of how we’d actually run a server. . .
- Example on page 1018-20 does though
- Its going to be a lot of code, so we’re going to present it slightly differently . . . .

I/O Multiplexing example overview

```
#include "csapp.h"

typedef struct { /* Represents a pool of connected descriptors */
    int maxfd;        /* Largest descriptor in all_socks_set */
    fd_set all_socks_set;  /* Set of all active descriptors */
    fd_set ready_set; /* Subset of descriptors ready for reading  */
    int nready;       /* Number of ready descriptors from select */
    int maxi;         /* Highwater index into client array */
    int clientfd[FD_SETSIZE];    /* Set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* Set of active read buffers */
} pool;

void init_pool(int listenfd, pool *p);
void add_client(int connfd, pool *p);
void check_clients(pool *p);

int byte_cnt = 0; /* Counts total bytes received by server */
```

I/O Multiplexing example overview

```
int main(int argc, char **argv){
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    static pool pool;
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    init_pool(listenfd, &pool);
    while (1) {
        /* Wait for listening/connected descriptor(s) to be ready */
        pool.ready_set = pool.all_socks_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
        /* If listening descriptor ready, add new client to pool */
        if (FD_ISSET(listenfd, &pool.ready_set)) {
            clientlen = sizeof(struct sockaddr_storage);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool);
        }
        check_clients(&pool); /* Echo line from each ready descriptor */
    }
}
```

I/O Multiplexing example overview

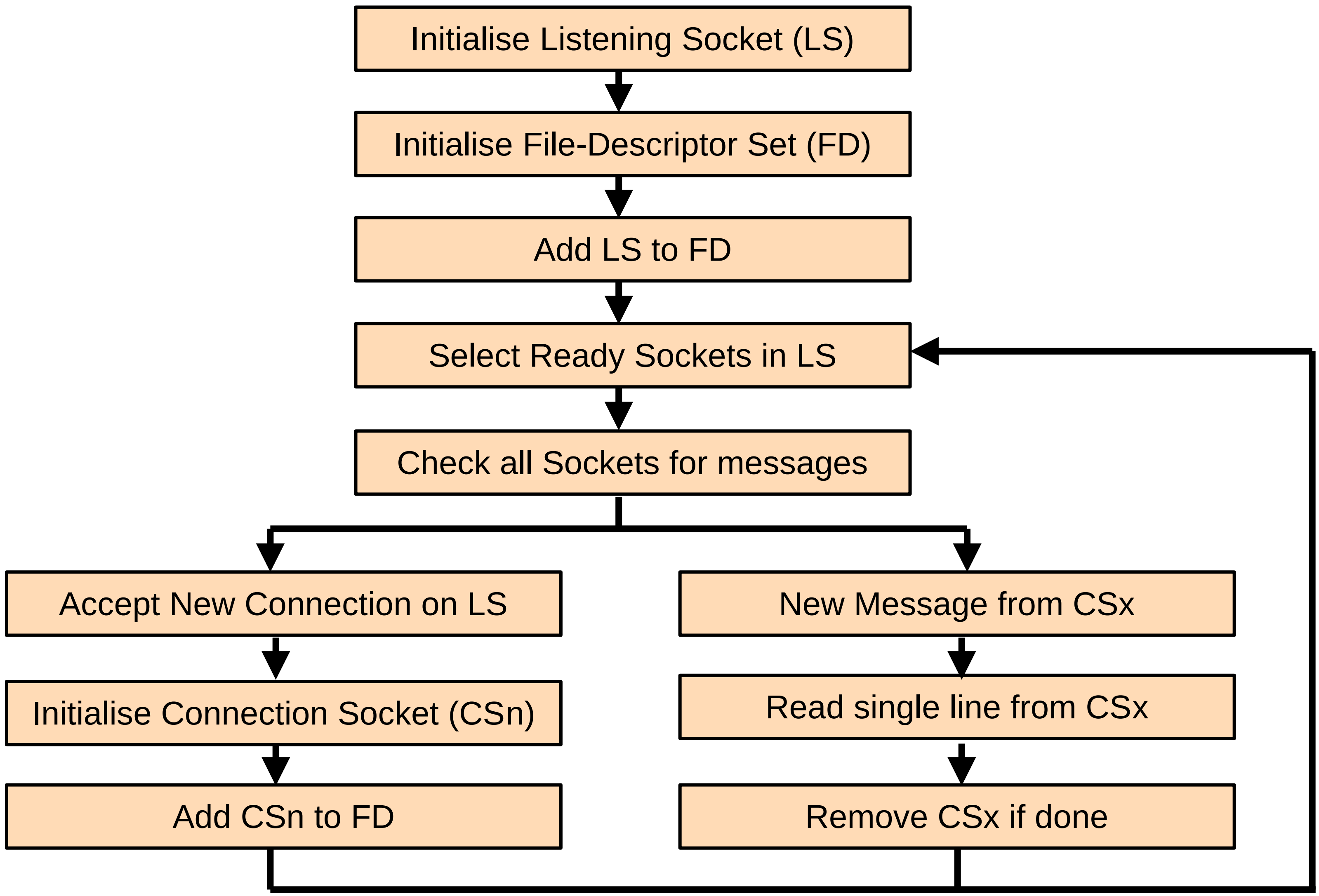
```
void init_pool(int listenfd, pool *p) {
    /* Initially, there are no connected descriptors */
    int i;
    p->maxi = -1;
    for (i=0; i< FD_SETSIZE; i++)
        p->clientfd[i] = -1;

    /* Initially, listenfd is only member of select read set */
    p->maxfd = listenfd;
    FD_ZERO(&p->all_socks_set);
    FD_SET(listenfd, &p->all_socks_set);
}
```

I/O Multiplexing example overview

```
void add_client(int connfd, pool *p) {
    int i;
    p->nready--;
    for (i = 0; i < FD_SETSIZE; i++)  /* Find an available slot */
        if (p->clientfd[i] < 0) {
            /* Add connected descriptor to the pool */
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);

            /* Add the descriptor to descriptor set */
            FD_SET(connfd, &p->all_socks_set);

            /* Update max descriptor and pool highwater mark */
            if (connfd > p->maxfd)
                p->maxfd = connfd;
            if (i > p->maxi)
                p->maxi = i;
            break;
        }
    if (i == FD_SETSIZE) /* Couldn't find an empty slot */
        app_error("add_client error: Too many clients");
}
```

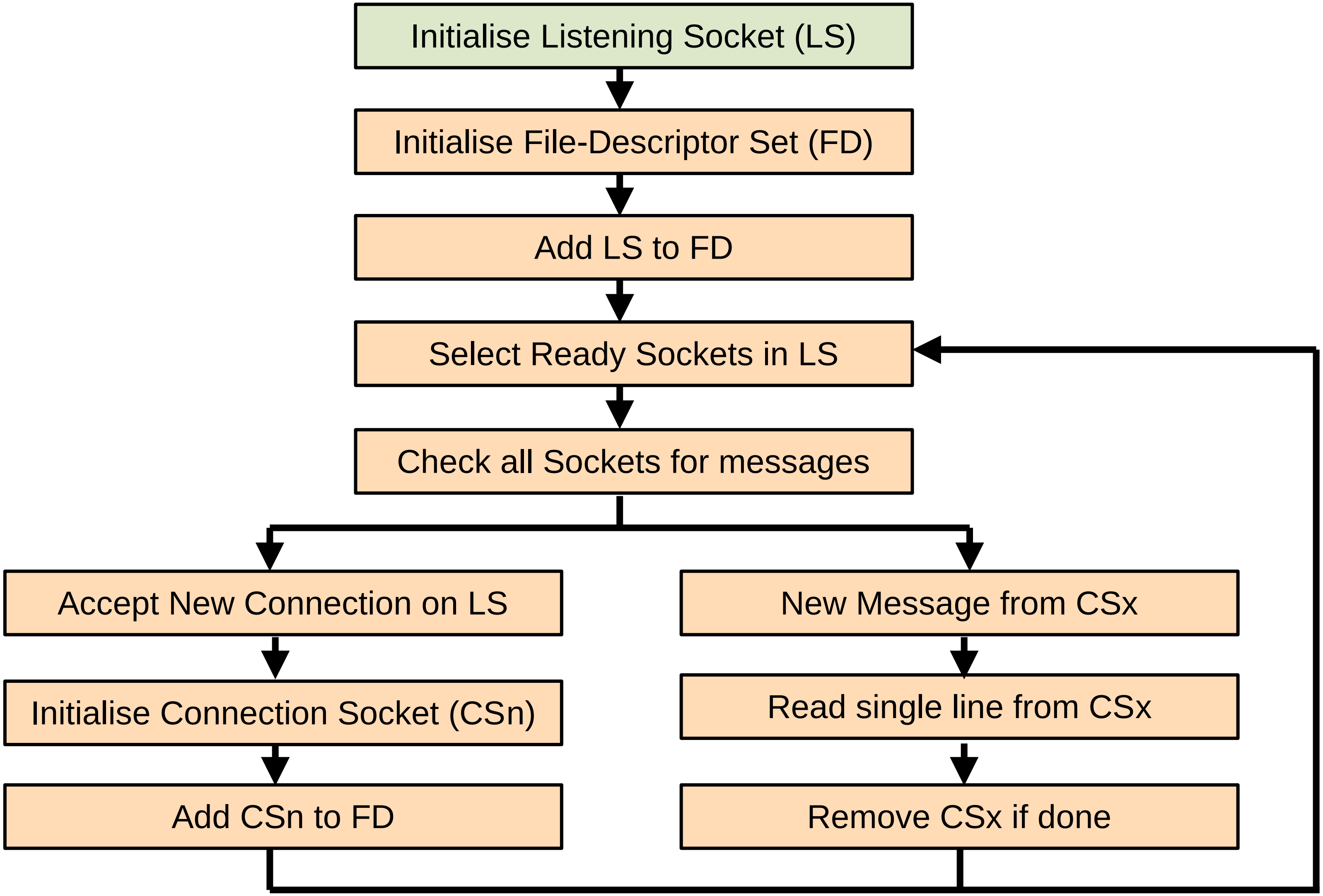
I/O Multiplexing example overview

```
void check_clients(pool *p) {
    int i, connfd, n;
    char buf[MAXLINE];
    rio_t rio;
    for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        rio = p->clientrio[i];
        /* If the descriptor is ready, echo a text line from it */
        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            p->nready--;
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
                byte_cnt += n;
                printf("Server received %d (%d total) bytes on fd %d\n",
                   n, byte_cnt, connfd);
                Rio_writen(connfd, buf, n);
            }
            /* EOF detected, remove descriptor from pool */
            else {
                Close(connfd);
                FD_CLR(connfd, &p->all_socks_set);
                p->clientfd[i] = -1;
            }}}}
```


Initialise Listening Socket (LS)
Initialise File-Descriptor Set (FD)
Add LS to FD
Select Ready Sockets in LS
Check all Sockets for messages
Accept New Connection on LS
Initialise Connection Socket (CSn)
Add CSn to FD
New Message from CSx
Read single line from CSx
Remove CSx if done

I/O Multiplexing example overview

```
#include "csapp.h"

typedef struct { /* Represents a pool of connected descriptors */
    int maxfd;        /* Largest descriptor in all_socks_set */
    fd_set all_socks_set;  /* Set of all active descriptors */
    fd_set ready_set; /* Subset of descriptors ready for reading  */
    int nready;       /* Number of ready descriptors from select */
    int maxi;         /* Highwater index into client array */
    int clientfd[FD_SETSIZE];    /* Set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* Set of active read buffers */
} pool;

void init_pool(int listenfd, pool *p);
void add_client(int connfd, pool *p);
void check_clients(pool *p);

int byte_cnt = 0; /* Counts total bytes received by server */
```

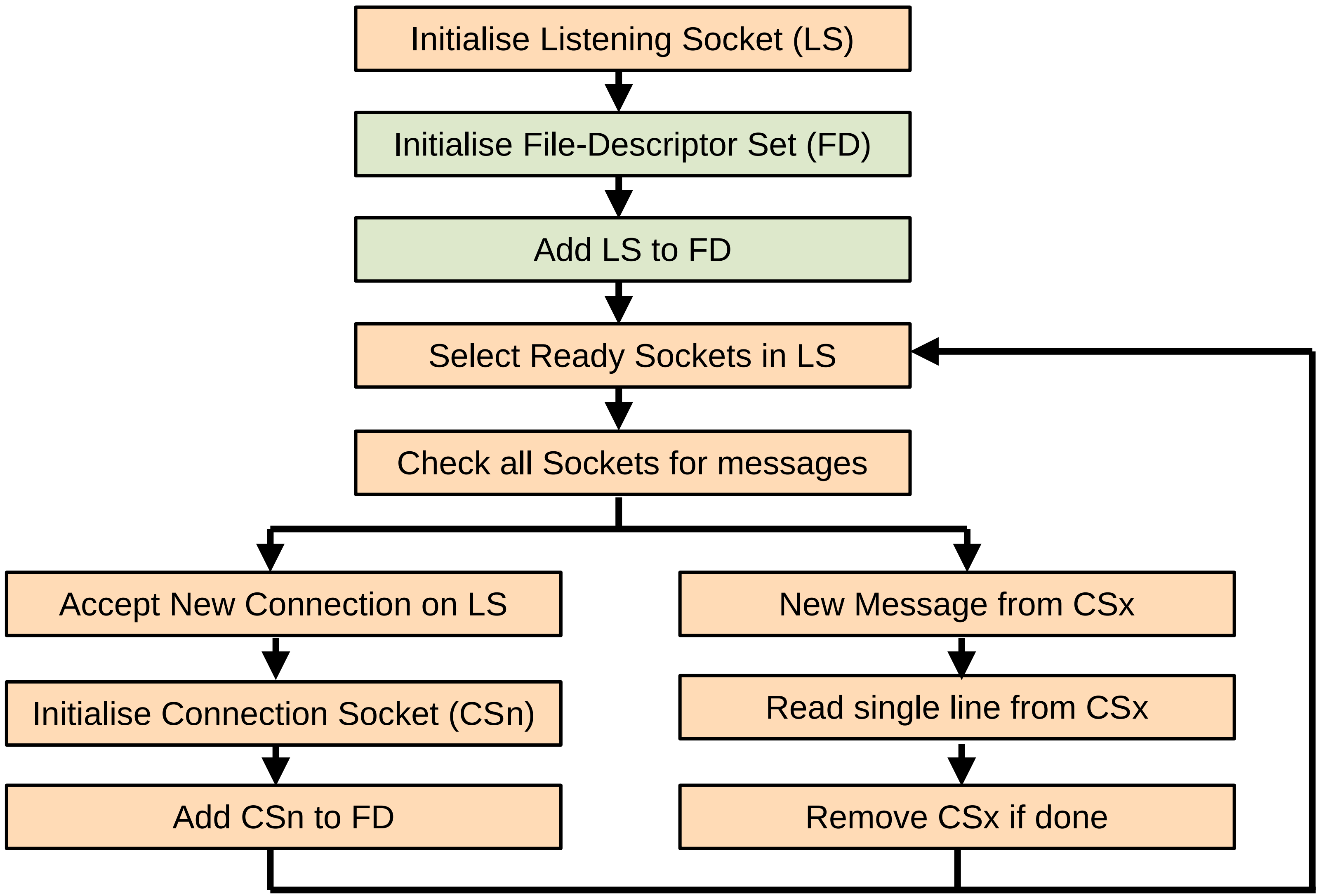


Initialise Listening Socket (LS)
Initialise File-Descriptor Set (FD)
Add LS to FD
Select Ready Sockets in LS
Check all Sockets for messages
Accept New Connection on LS
Initialise Connection Socket (CSn)
Add CSn to FD
New Message from CSx
Read single line from CSx
Remove CSx if done

I/O Multiplexing example overview

```
int main(int argc, char **argv){
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    static pool pool;
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    init_pool(listenfd, &pool);
    while (1) {
        /* Wait for listening/connected descriptor(s) to be ready */
        pool.ready_set = pool.all_socks_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
        /* If listening descriptor ready, add new client to pool */
        if (FD_ISSET(listenfd, &pool.ready_set)) {
            clientlen = sizeof(struct sockaddr_storage);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool);
        }
        check_clients(&pool); /* Echo line from each ready descriptor */
    }
}
```

IGNORE FOR NOW


Initialise Listening Socket (LS)
Initialise File-Descriptor Set (FD)
Add LS to FD
Select Ready Sockets in LS
Check all Sockets for messages
Accept New Connection on LS
Initialise Connection Socket (CSn)
Add CSn to FD
New Message from CSx
Read single line from CSx
Remove CSx if done

I/O Multiplexing example overview

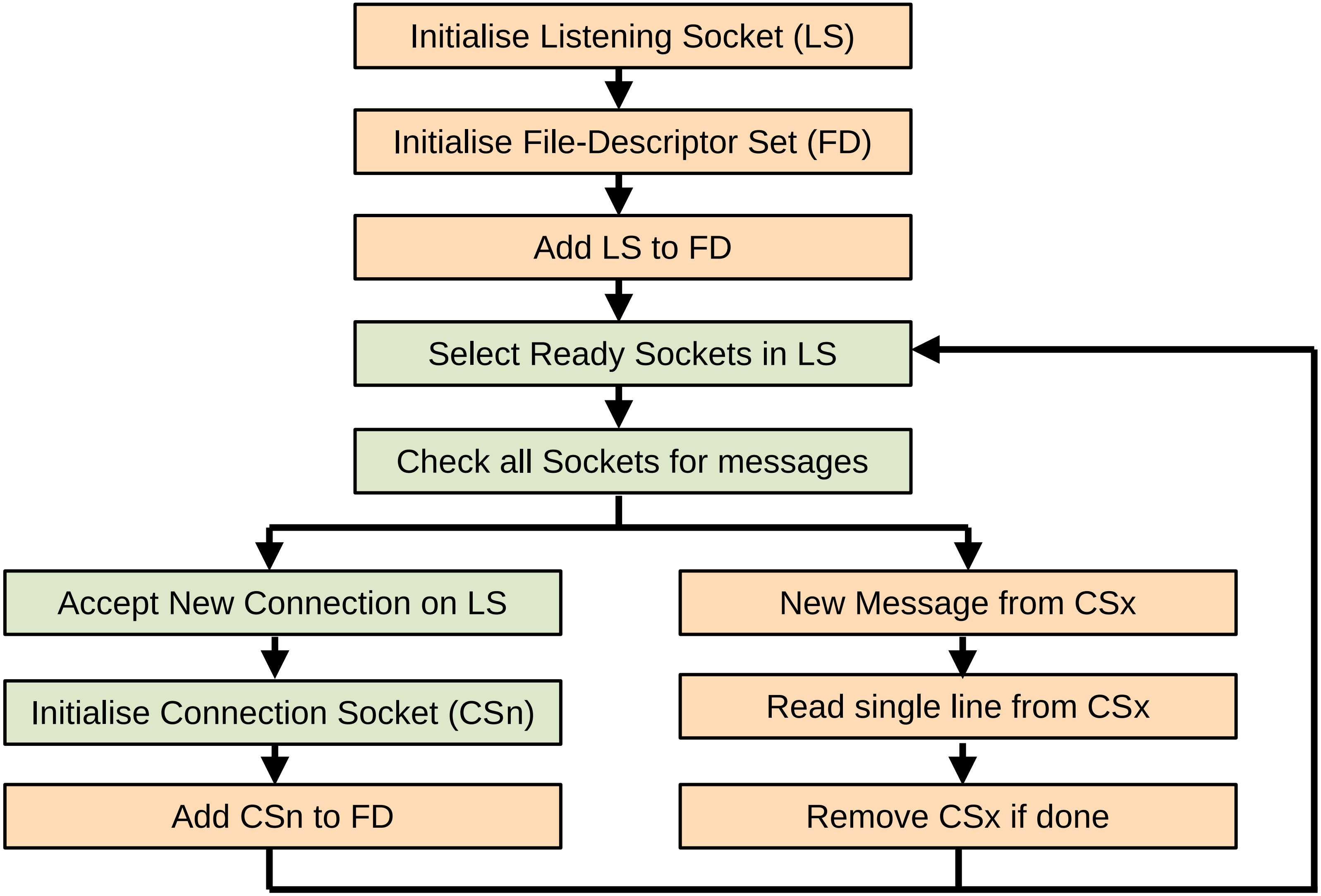
ALREADY COVERED

KEEP IGNORING ME

```
int main(int argc, char **argv){
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    static pool pool;
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    init_pool(listenfd, &pool);
    while (1) {
        /* Wait for listening/connected descriptor(s) to be ready */
        pool.ready_set = pool.all_socks_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
        /* If listening descriptor ready, add new client to pool */
        if (FD_ISSET(listenfd, &pool.ready_set)) {
            clientlen = sizeof(struct sockaddr_storage);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool);
        }
        check_clients(&pool); /* Echo line from each ready descriptor */
    }
}
```

I/O Multiplexing example overview

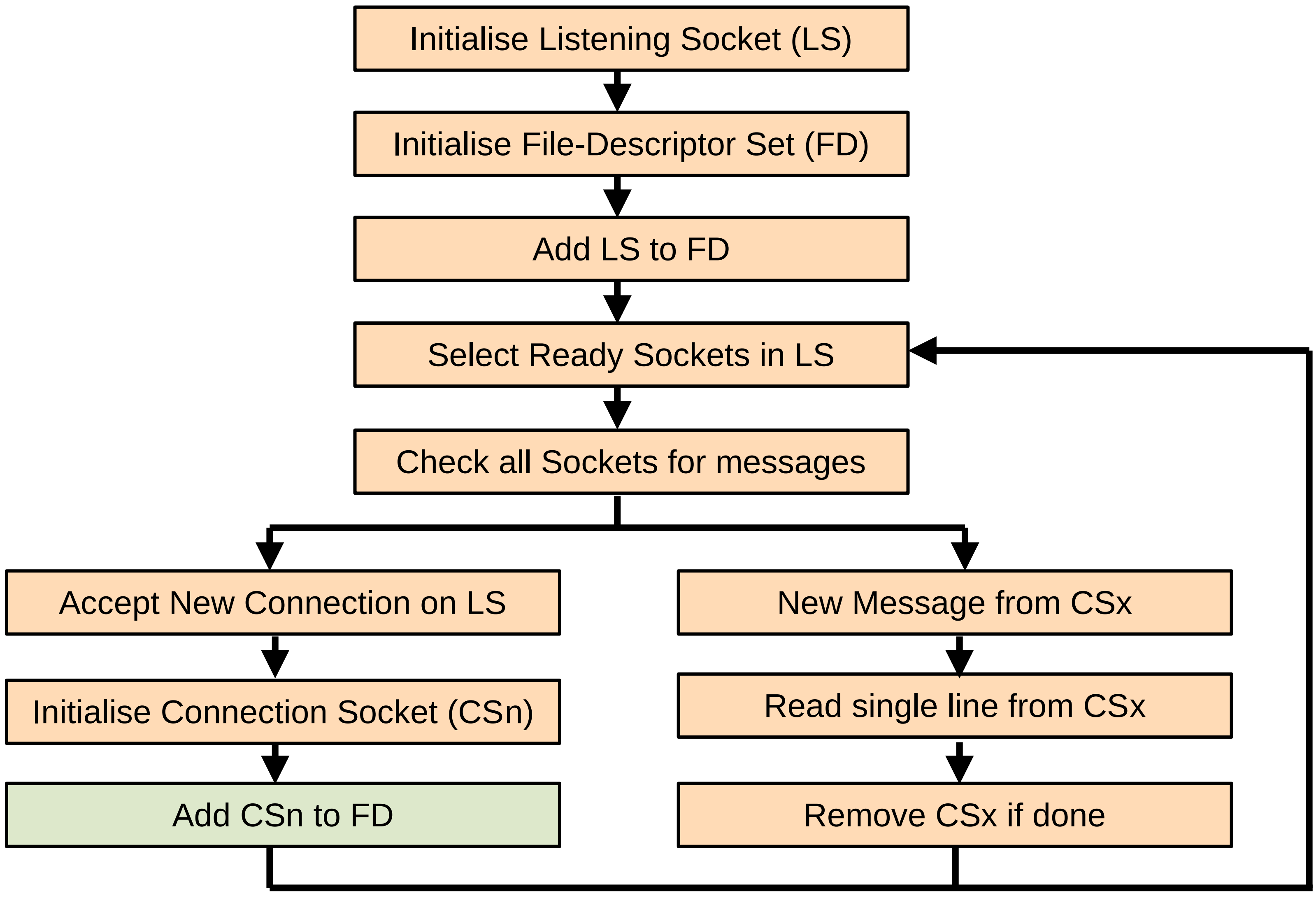
```
void init_pool(int listenfd, pool *p) {
    /* Initially, there are no connected descriptors */
    int i;
    p->maxi = -1;
    for (i=0; i< FD_SETSIZE; i++)
        p->clientfd[i] = -1;

    /* Initially, listenfd is only member of select read set */
    p->maxfd = listenfd;
    FD_ZERO(&p->all_socks_set);
    FD_SET(listenfd, &p->all_socks_set);
}
```


Initialise Listening Socket (LS)
Initialise File-Descriptor Set (FD)
Add LS to FD
Select Ready Sockets in LS
Check all Sockets for messages
Accept New Connection on LS
Initialise Connection Socket (CSn)
Add CSn to FD
New Message from CSx
Read single line from CSx
Remove CSx if done

I/O Multiplexing example overview

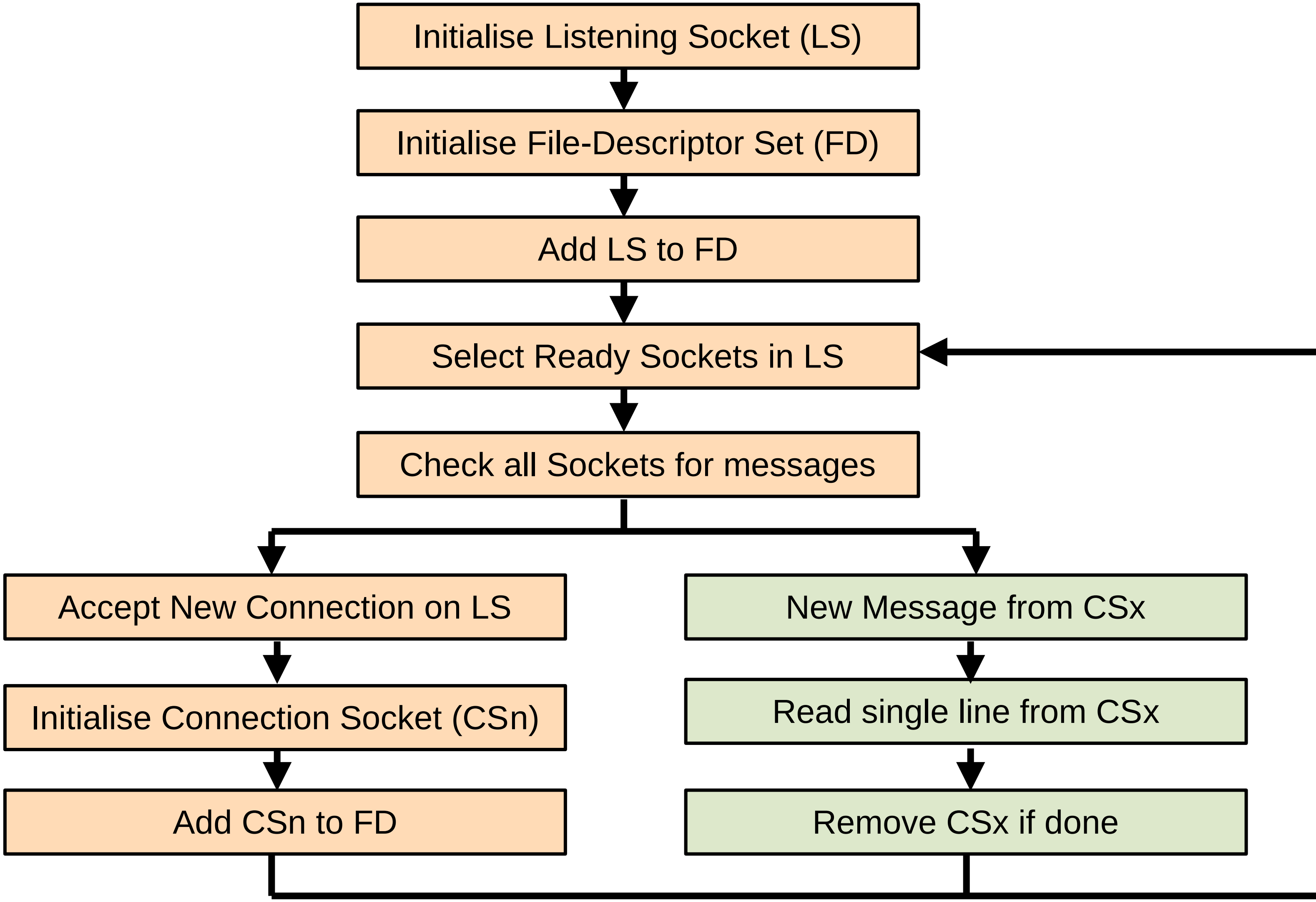
```
int main(int argc, char **argv){
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    static pool pool;
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    init_pool(listenfd, &pool);
    while (1) {
        /* Wait for listening/connected descriptor(s) to be ready */
        pool.ready_set = pool.all_socks_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
        /* If listening descriptor ready, add new client to pool */
        if (FD_ISSET(listenfd, &pool.ready_set)) {
            clientlen = sizeof(struct sockaddr_storage);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool);
        }
        check_clients(&pool); /* Echo line from each ready descriptor */
    }
}
```

NOT RELEVANT HERE


Initialise Listening Socket (LS)
Initialise File-Descriptor Set (FD)
Add LS to FD
Select Ready Sockets in LS
Check all Sockets for messages
Accept New Connection on LS
Initialise Connection Socket (CSn)
Add CSn to FD
New Message from CSx
Read single line from CSx
Remove CSx if done

I/O Multiplexing example overview

```
void add_client(int connfd, pool *p) {
    int i;
    p->nready--;
    for (i = 0; i < FD_SETSIZE; i++)  /* Find an available slot */
        if (p->clientfd[i] < 0) {
            /* Add connected descriptor to the pool */
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);

            /* Add the descriptor to descriptor set */
            FD_SET(connfd, &p->all_socks_set);

            /* Update max descriptor and pool highwater mark */
            if (connfd > p->maxfd)
                p->maxfd = connfd;
            if (i > p->maxi)
                p->maxi = i;
            break;
        }
    if (i == FD_SETSIZE) /* Couldn't find an empty slot */
        app_error("add_client error: Too many clients");
}
```


Initialise Listening Socket (LS)
Initialise File-Descriptor Set (FD)
Add LS to FD
Select Ready Sockets in LS
Check all Sockets for messages
Accept New Connection on LS
Initialise Connection Socket (CSn)
Add CSn to FD
New Message from CSx
Read single line from CSx
Remove CSx if done

I/O Multiplexing example overview

```
void check_clients(pool *p) {
    int i, connfd, n;
    char buf[MAXLINE];
    rio_t rio;
    for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        rio = p->clientrio[i];
        /* If the descriptor is ready, echo a text line from it */
        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            p->nready--;
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
                byte_cnt += n;
                printf("Server received %d (%d total) bytes on fd %d\n",
                   n, byte_cnt, connfd);
                Rio_writen(connfd, buf, n);
            }
            /* EOF detected, remove descriptor from pool */
            else {
                Close(connfd);
                FD_CLR(connfd, &p->all_socks_set);
                p->clientfd[i] = -1;
            }}}}
```

Pros and Cons of Event-based Servers

- **+ One logical control flow and address space.**
- **+ Can single-step with a debugger.**
- **+ No process or thread control overhead.**
  - Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado

- **– Significantly more complex to code than process- or thread-based designs.**
- **– Hard to provide fine-grained concurrency**
  - E.g., how to deal with partial HTTP request headers
- **– Cannot take advantage of multi-core**
  - Single thread of control

Summary: Approaches to Concurrency

- **Process-based**
  - Hard to share resources: Easy to avoid unintended sharing
  - High overhead in adding/removing clients
- **Event-based**
  - Tedious and low level
  - Total control over scheduling
  - Very low overhead
  - Cannot create as fine grained a level of concurrency
  - Does not make use of multi-core
- **Thread-based**
  - Easy to share resources: Perhaps too easy
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug
    - Event orderings not repeatable

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

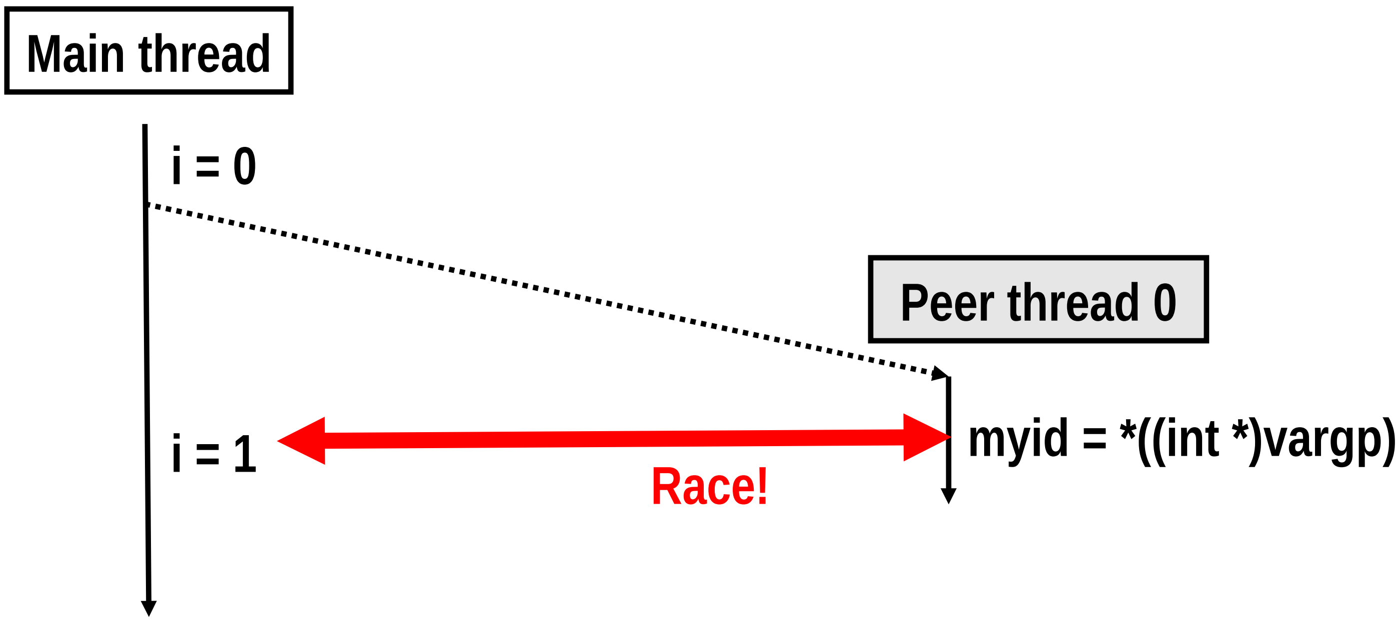
```
/* A threaded program with a race */
int main()
{
    pthread_t tid[N];
    int i;

    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

```
/* 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

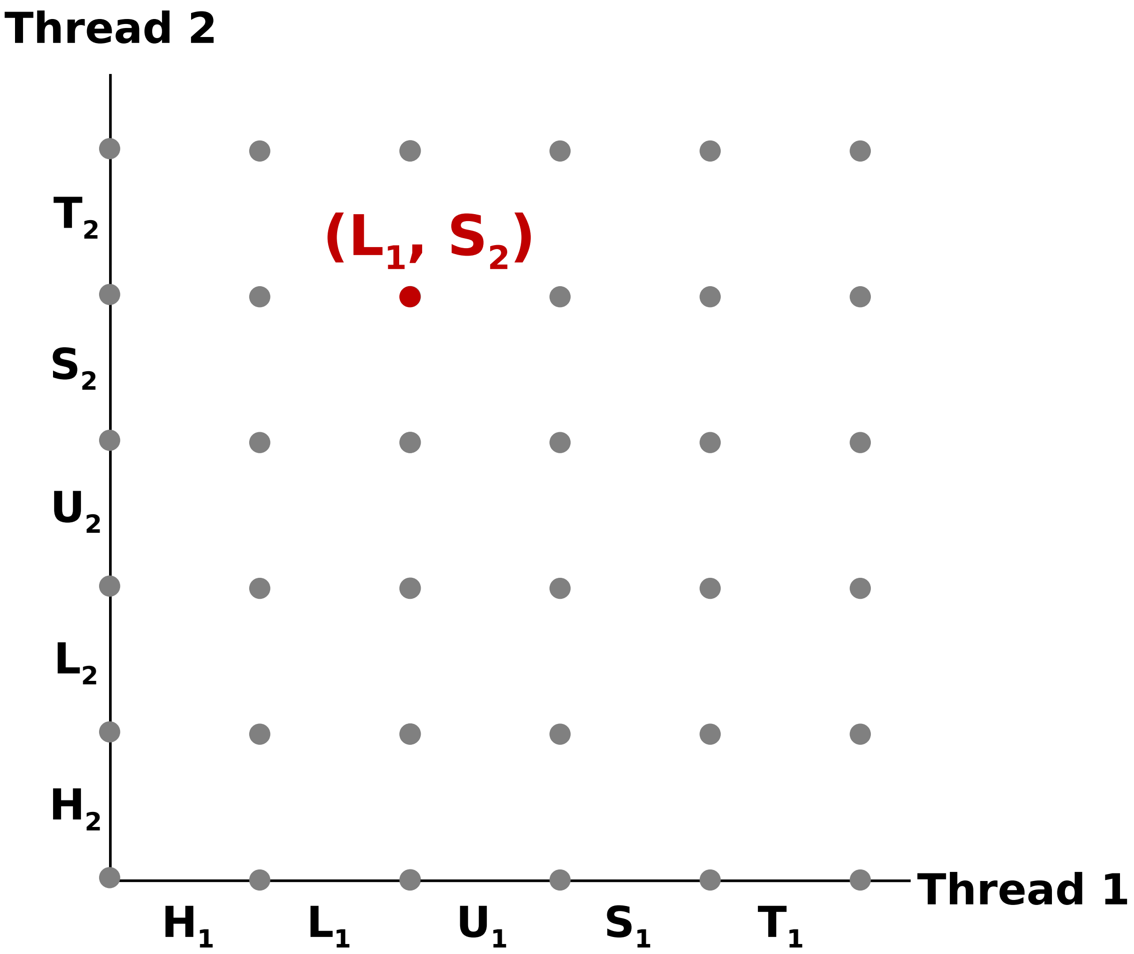
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 0 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], 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]:
$P(s_0)$;
$P(s_1)$;
cnt++;
$V(s_0)$;
$V(s_1)$;

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

Progress Graphs



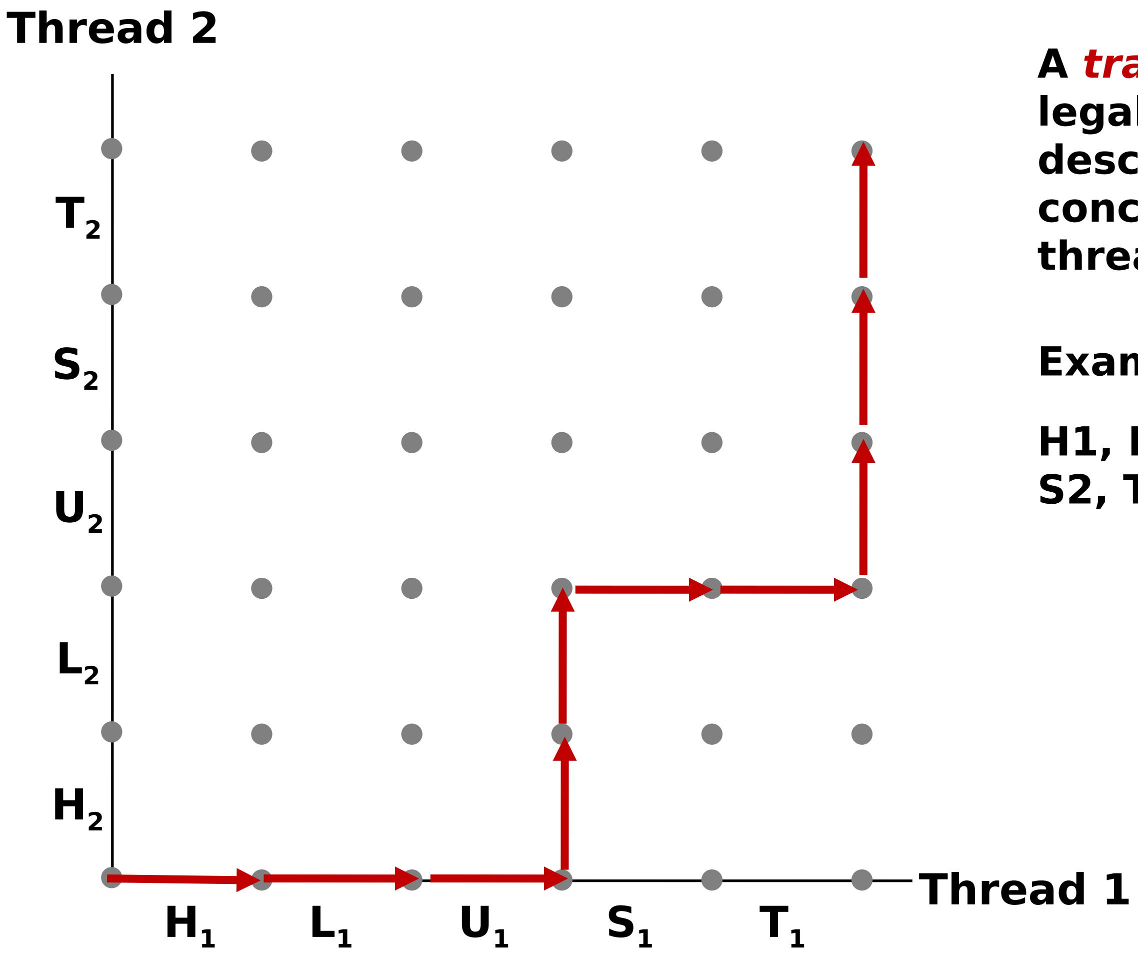
A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

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.

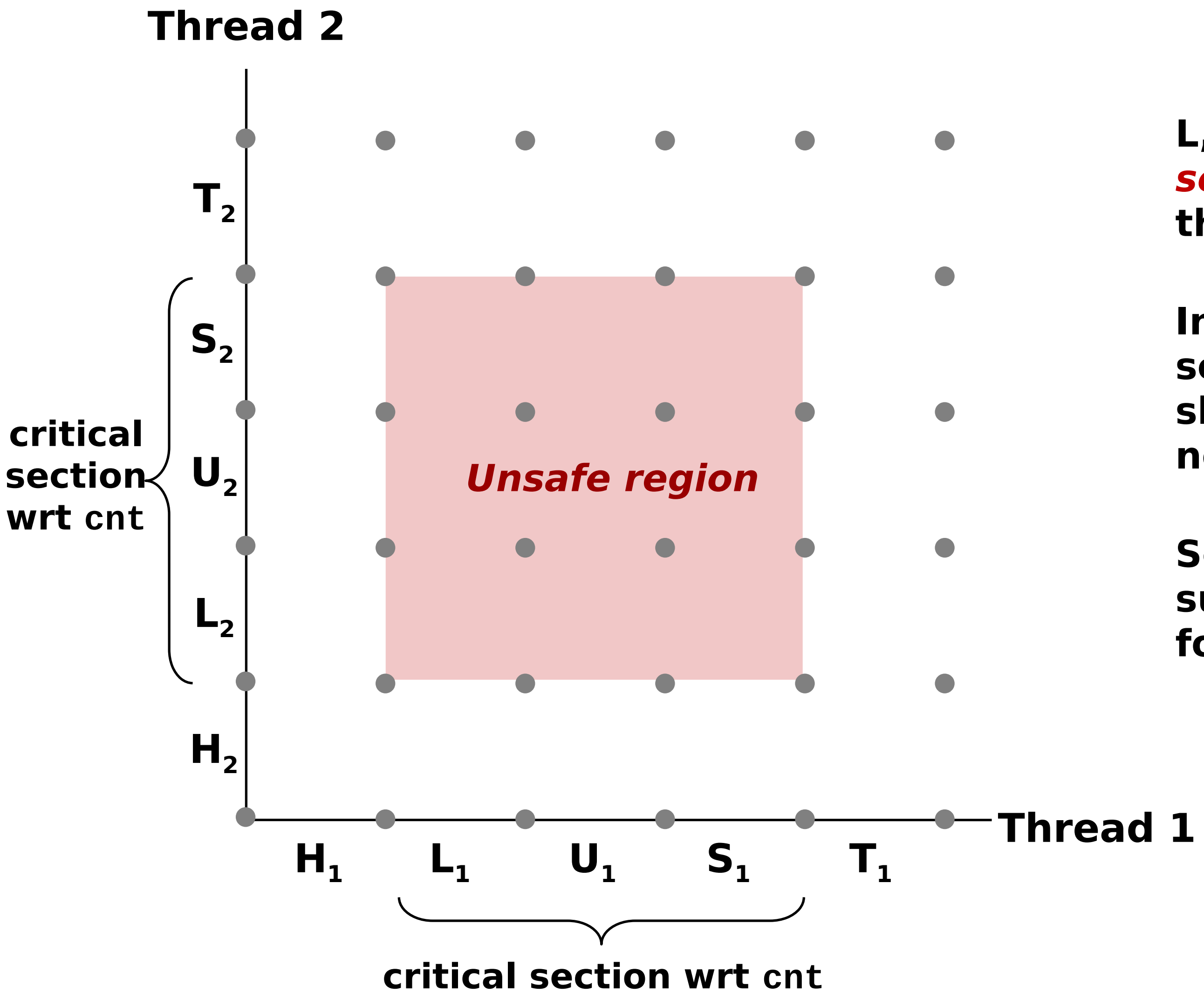
Example:

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

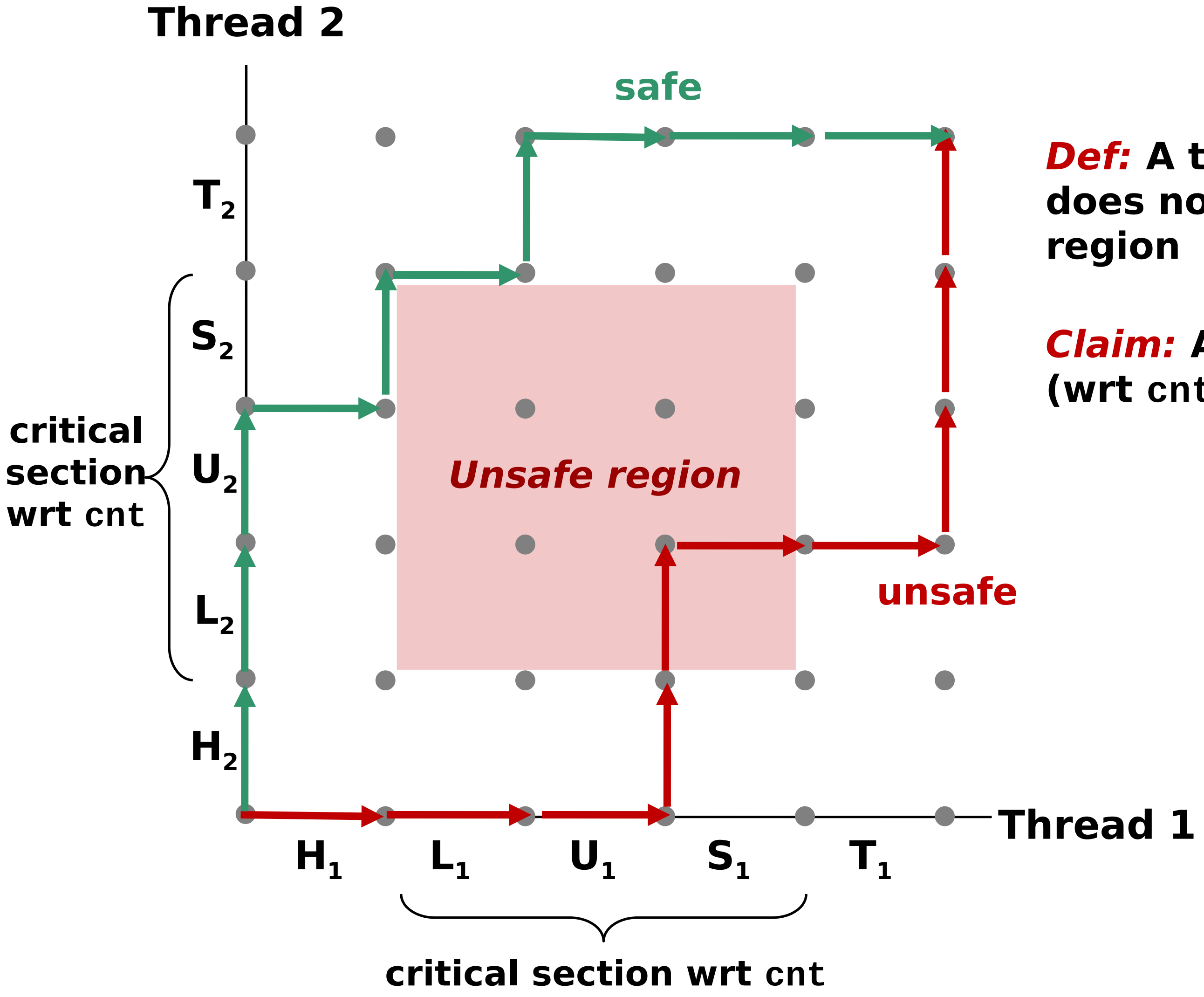


L, U, and S form a *critical section* with respect to the shared variable `cnt`

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

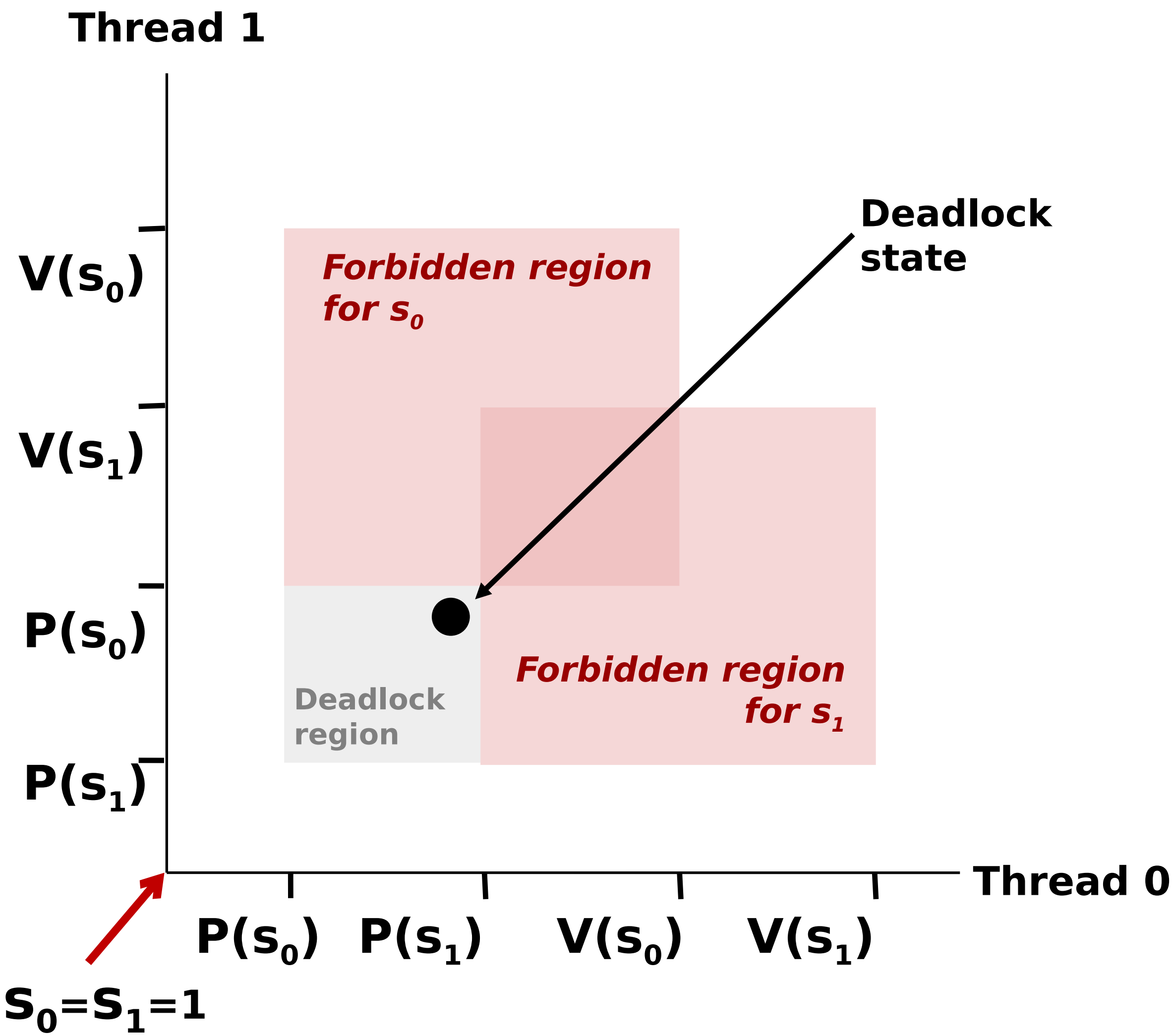
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 0 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], 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

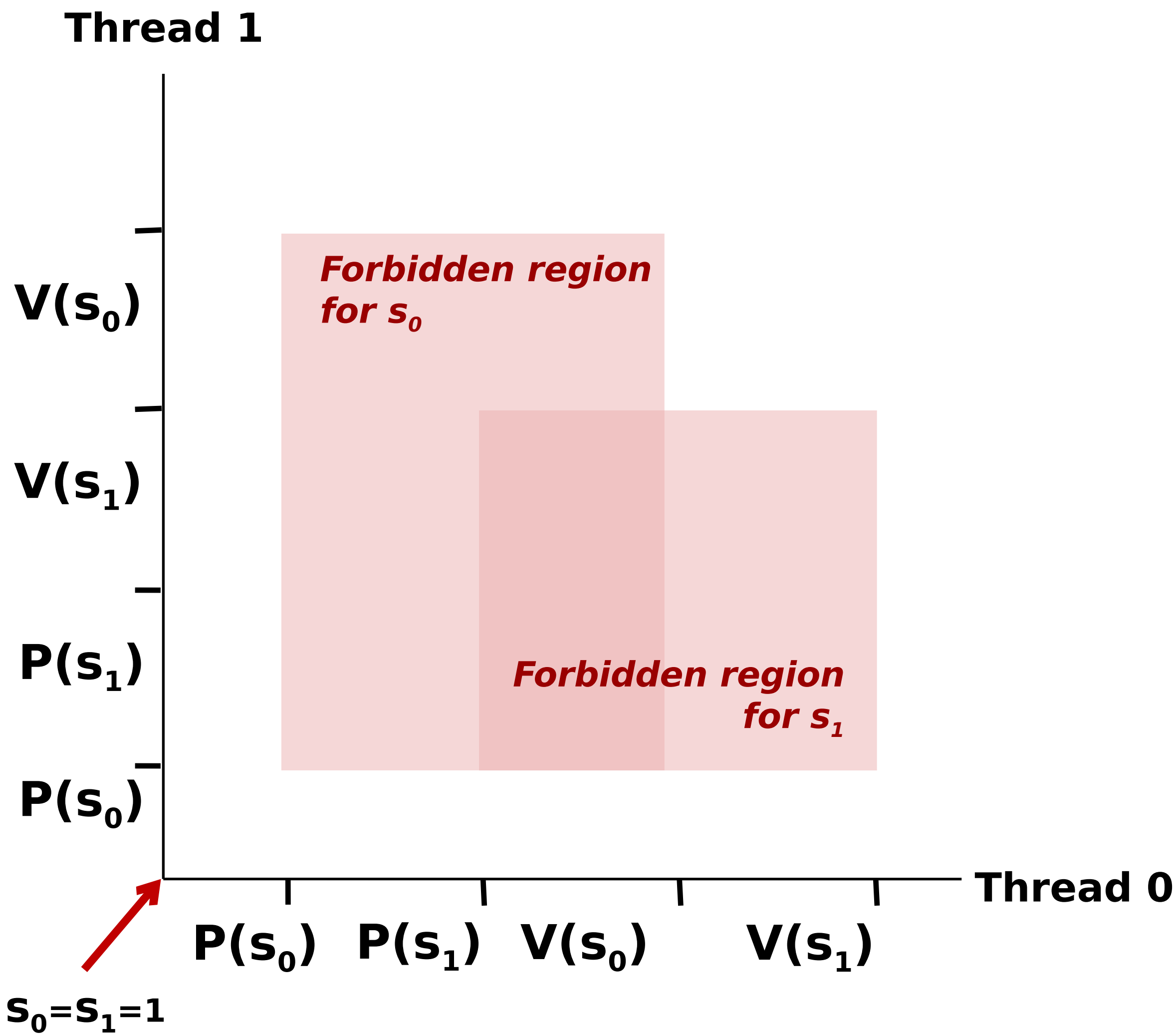
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 0 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 0 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], 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.

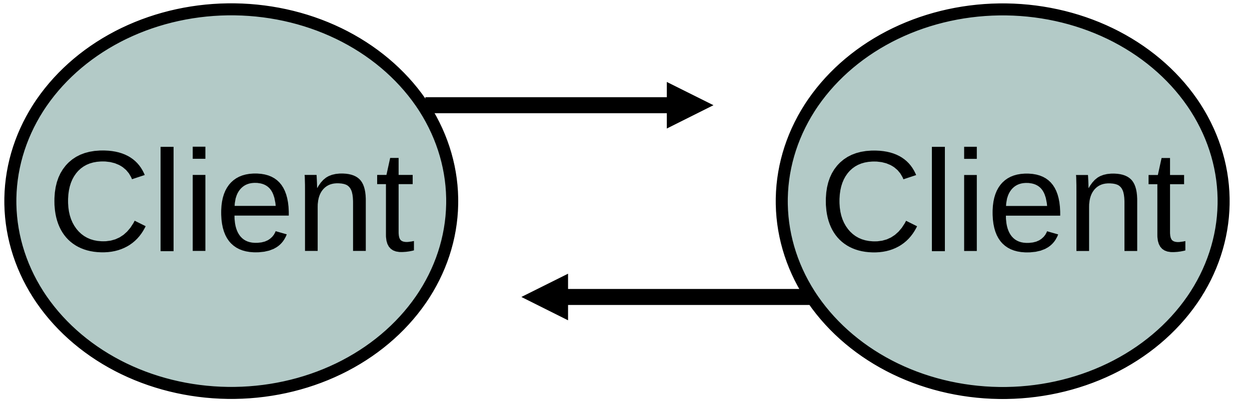
- **We can ignore any non-blocking code.** Anything that completes in a finite time can be ignored.

- **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)**



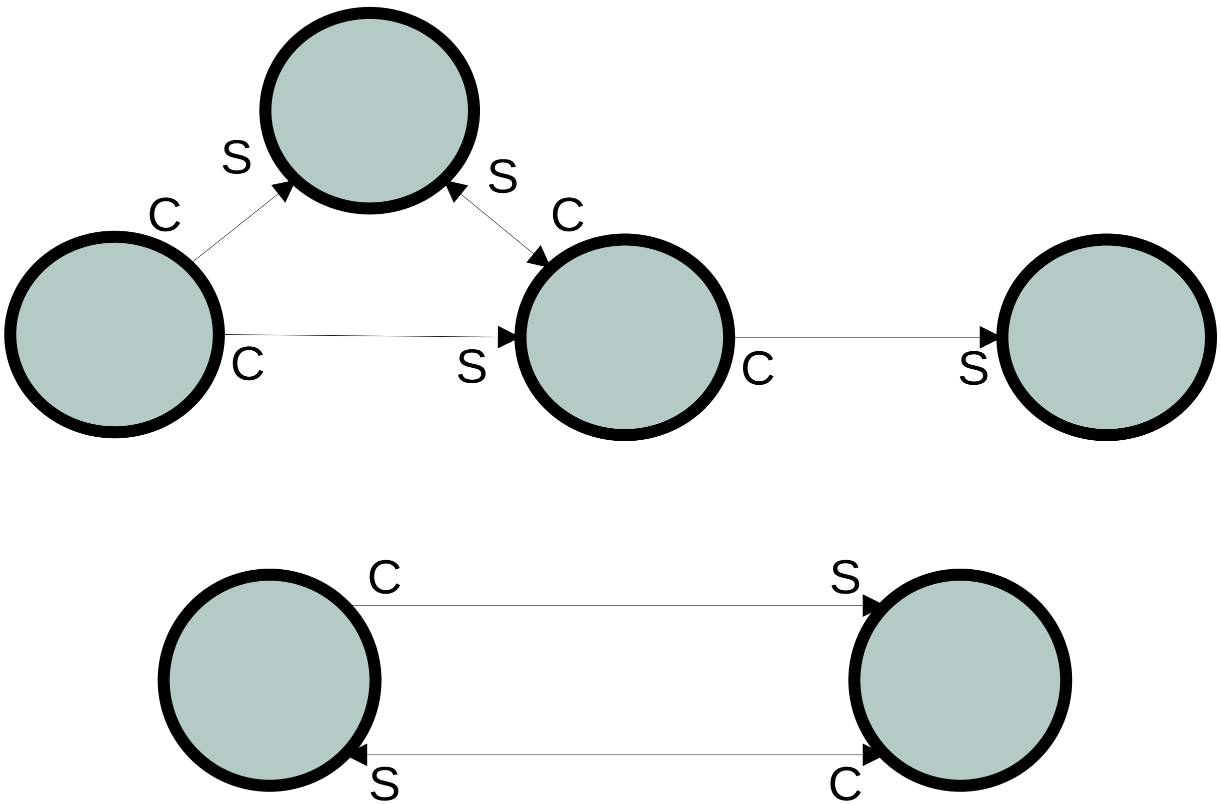
CSP

- **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**

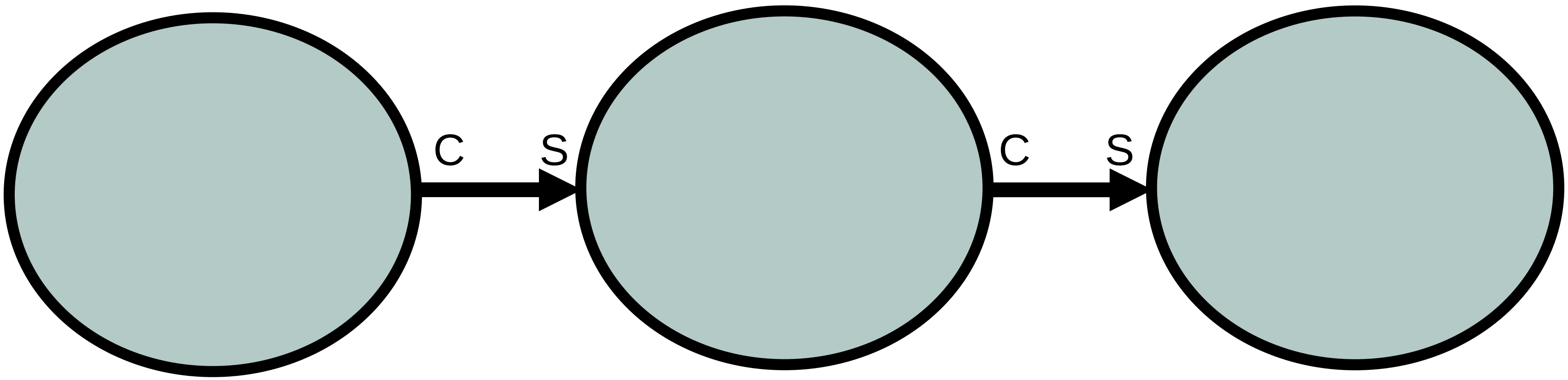
Process Diagrams



What does this get us

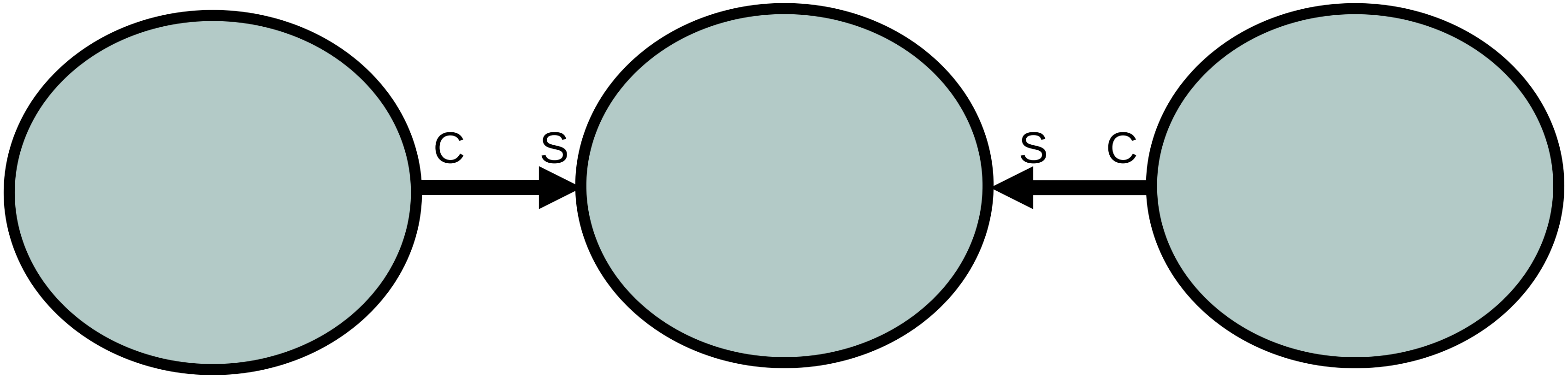
- **CSP is the source of the client-server model**
- **Semantically, in the client server model:**
  - Only clients initiate communications
  - If a client expects a response, a server will provide one in a finite amount of time
  - If a client expects a response, it will be immediately ready to receive 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?



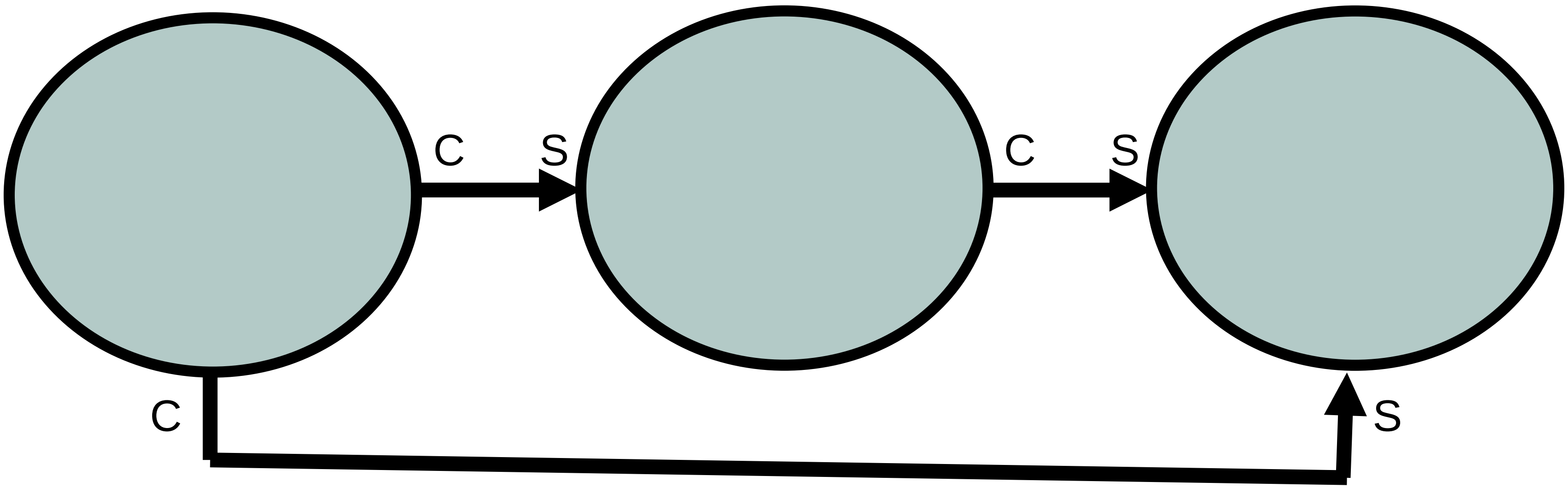
No deadlock

What about Deadlock between hosts?



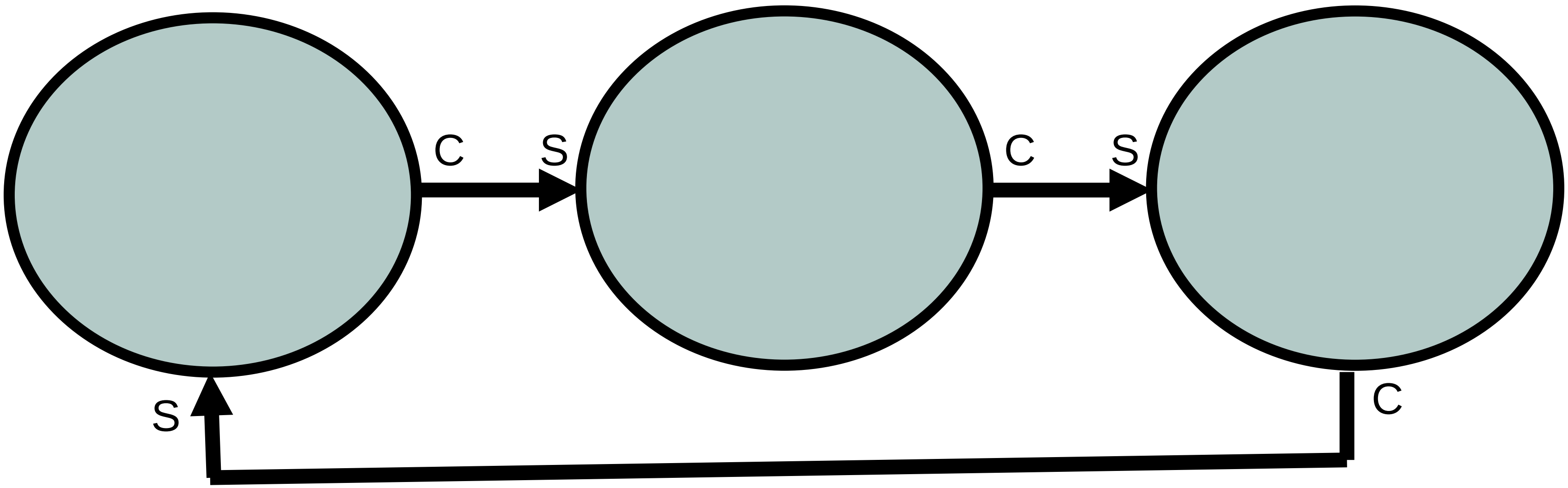
No deadlock

What about Deadlock between hosts?



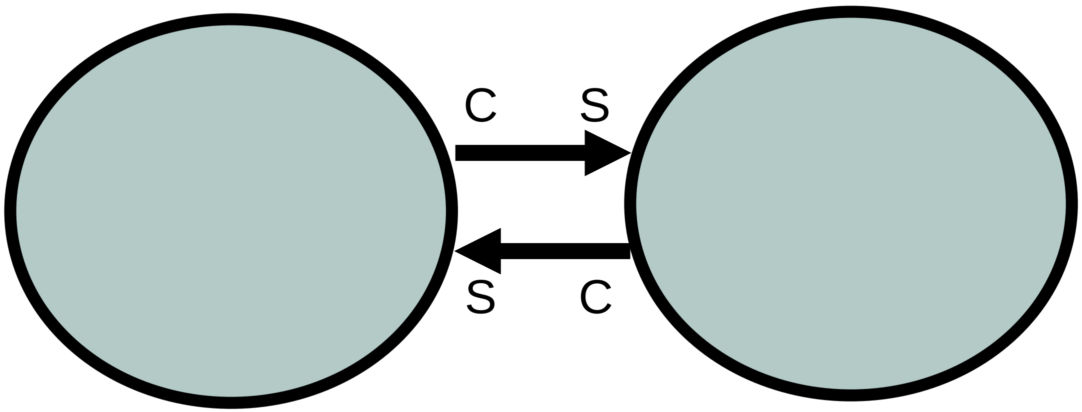
No deadlock

What about Deadlock between hosts?



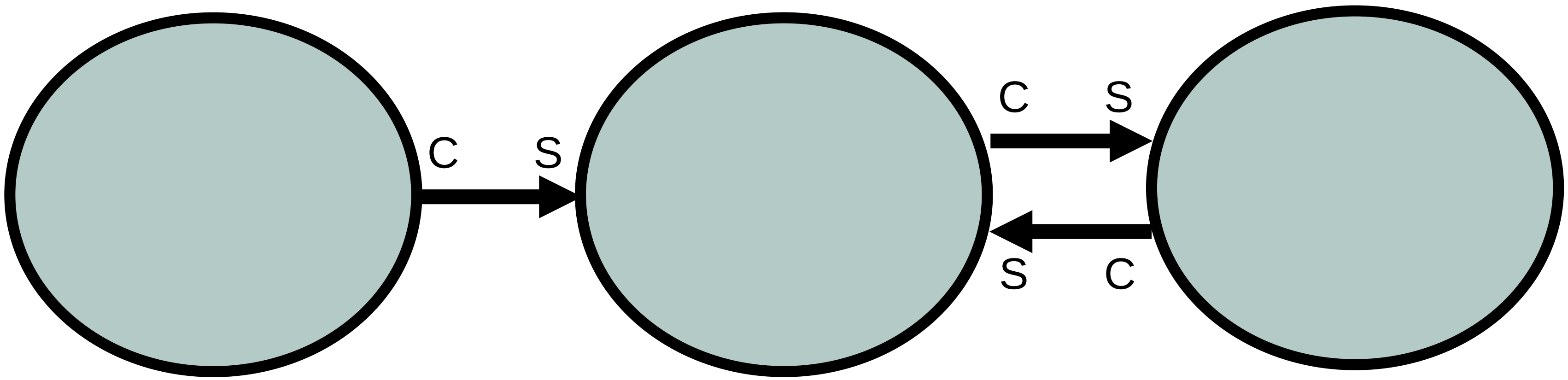
Deadlock

What about Deadlock between hosts?



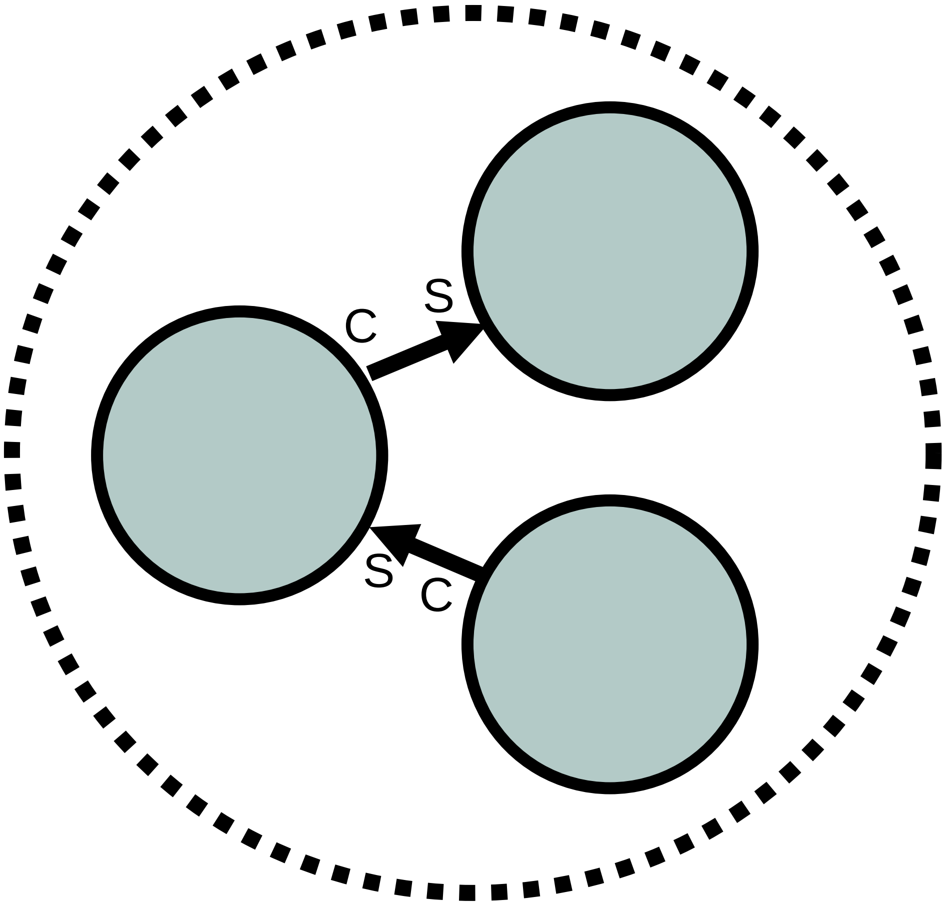
Deadlock

What about Deadlock between hosts?



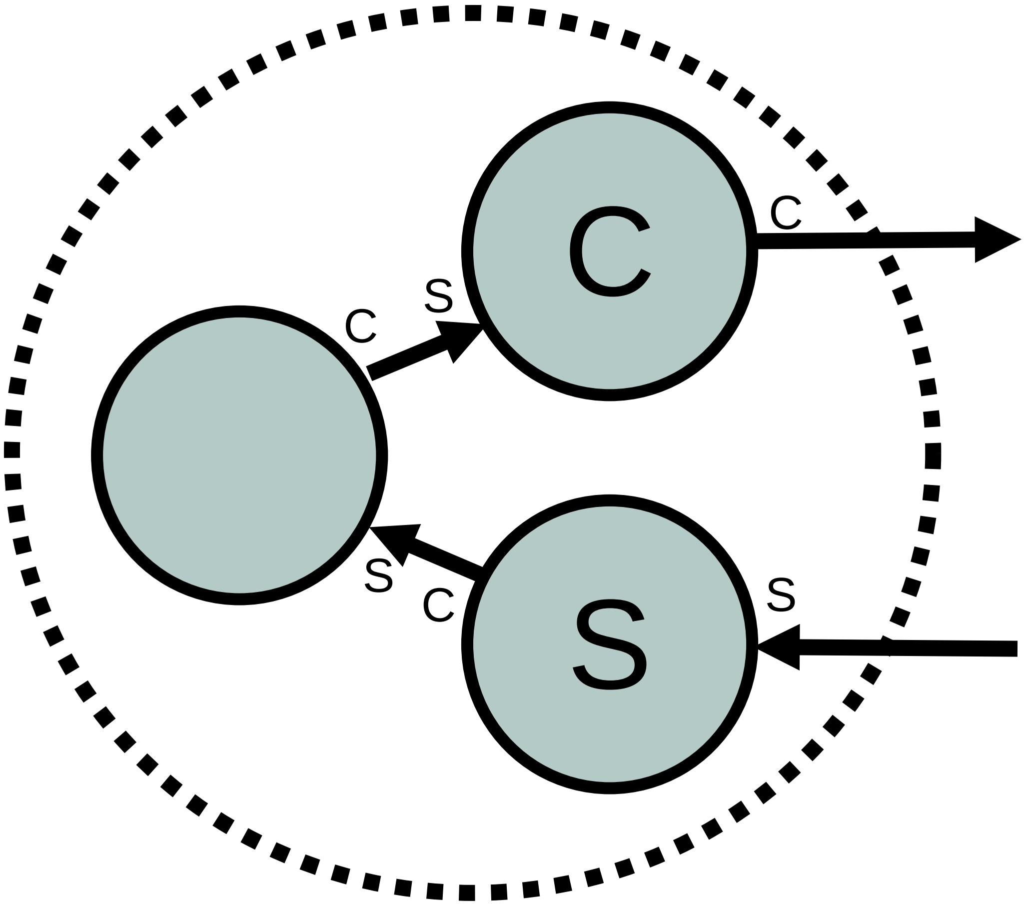
Deadlock

Didn’t we solve this already?



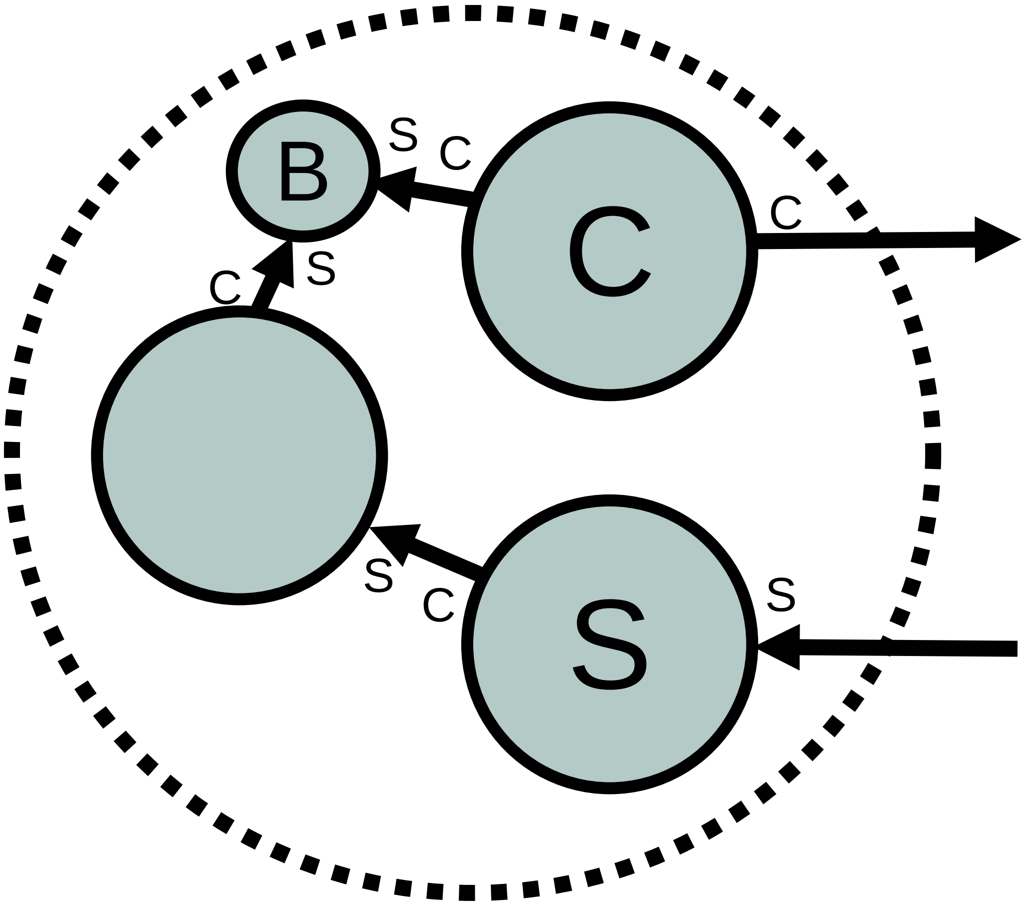
No deadlock

Didn't we solve this already?



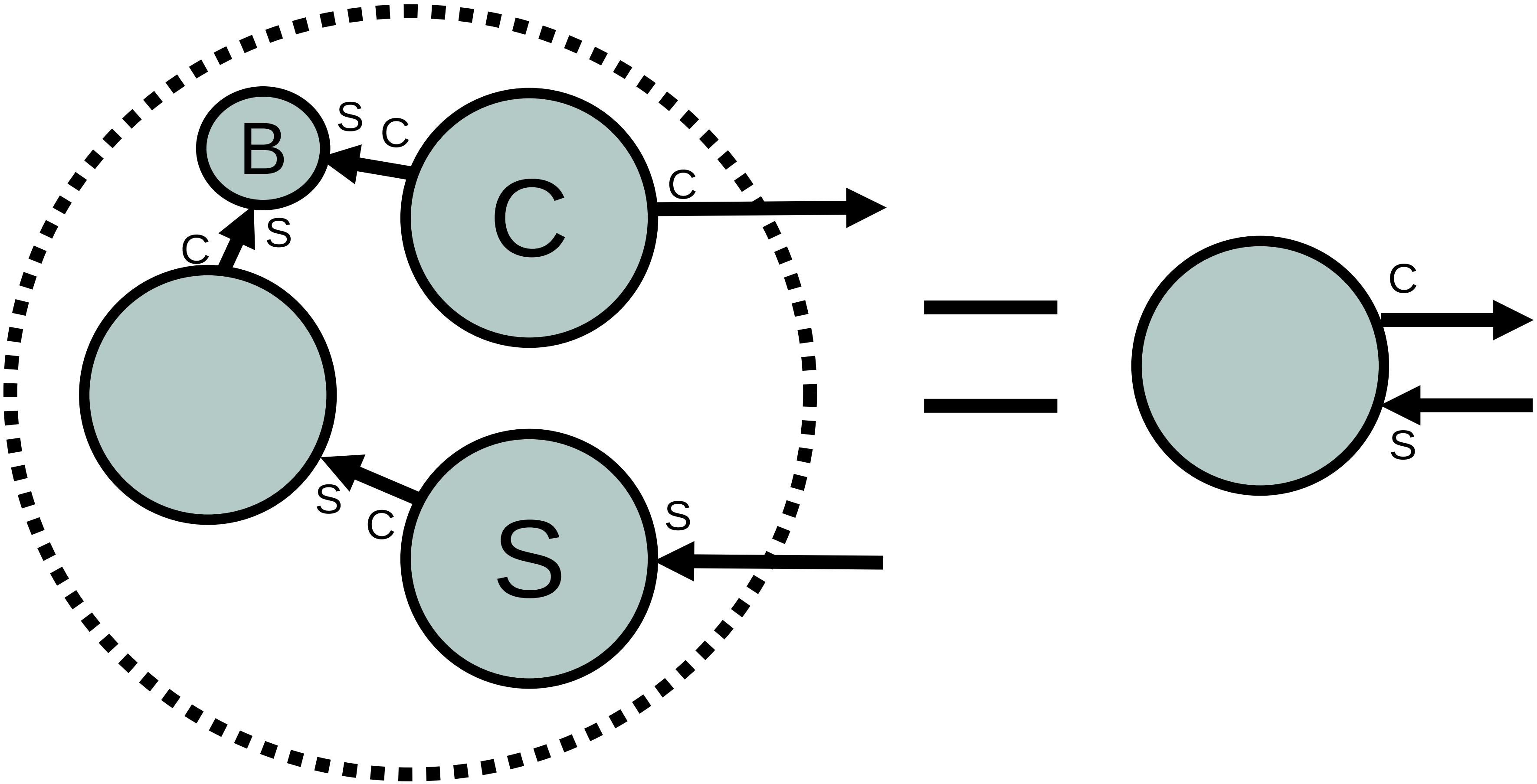
Maybe
Deadlock
deadlock?

Didn't we solve this already?



No deadlock

Why the dotted circle?



Some notes …

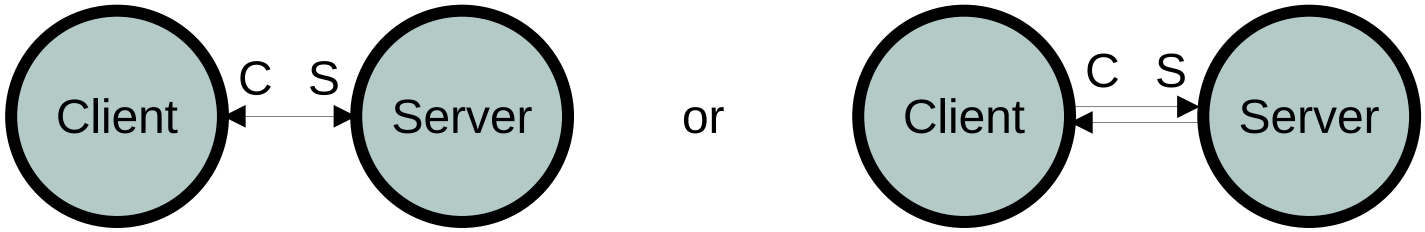
- A client/server loop doesn’t actually mean deadlock
- But no client/server loop does guarantee no deadlock
- Depending on internal structure of a process, deadlock might not occur
- But the road to deadlock is paved by good intentions
- **Any client/server loop MUST be carefully examined and justified**

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)

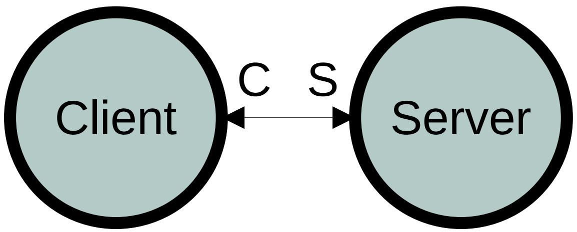
Translating code into diagrams

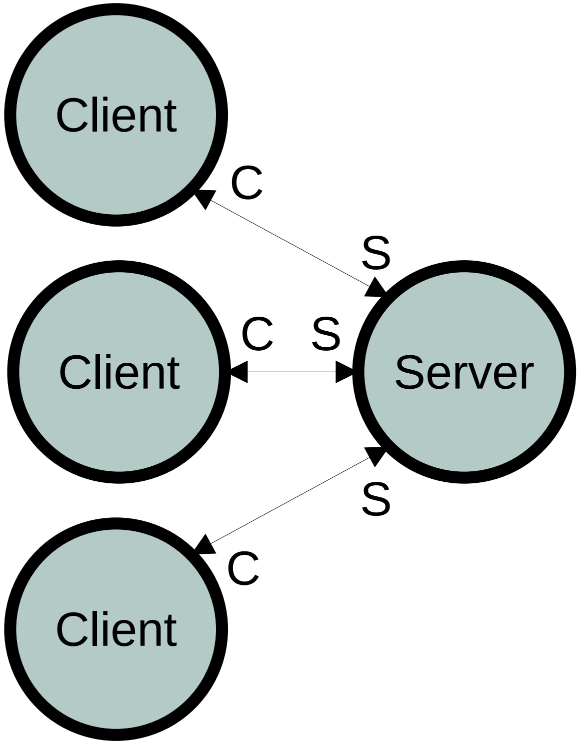
- **Lets draw A3**
- **A3 Server**
  - Listens for connections
  - Can handle parallel connections
  - Always responds
  - No additional comms from it
- **A3 Client**
  - Connects to the server
  - Sends two message types (register, get)
  - Each message is sequential



Translating code into diagrams

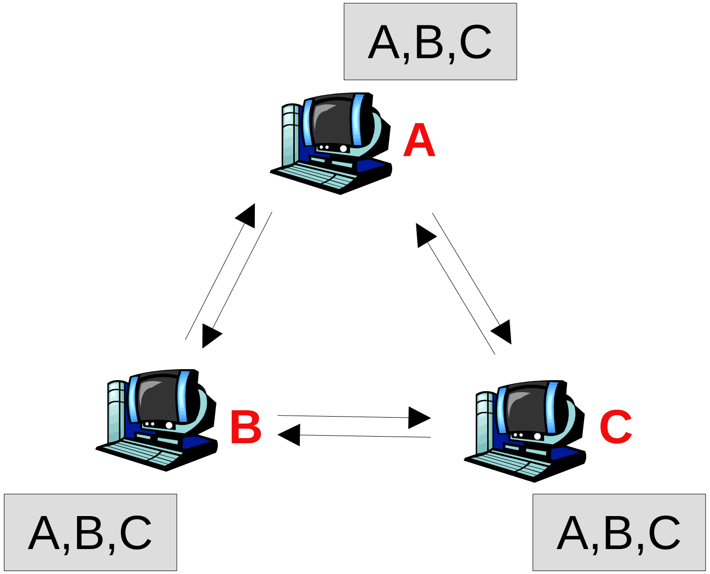
- Identical interactions can *often* be left out
- As each connection is served concurrently, we can effectively add infinite clients and nothing changes
- This might change if there are dependencies between connections





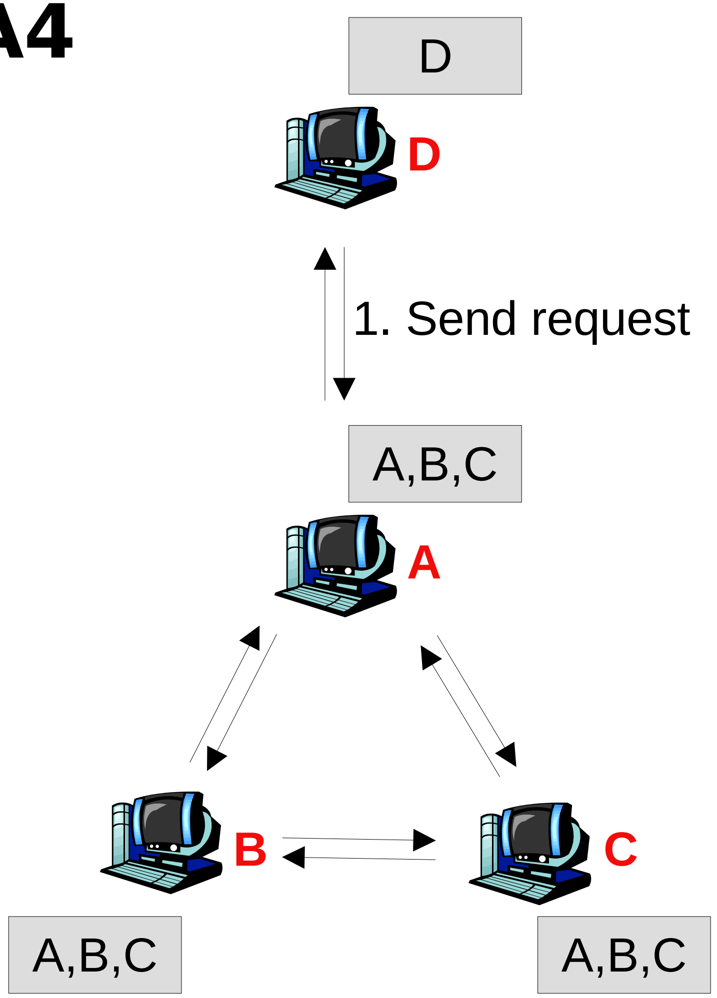
Peer to Peer Network in A4

- 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



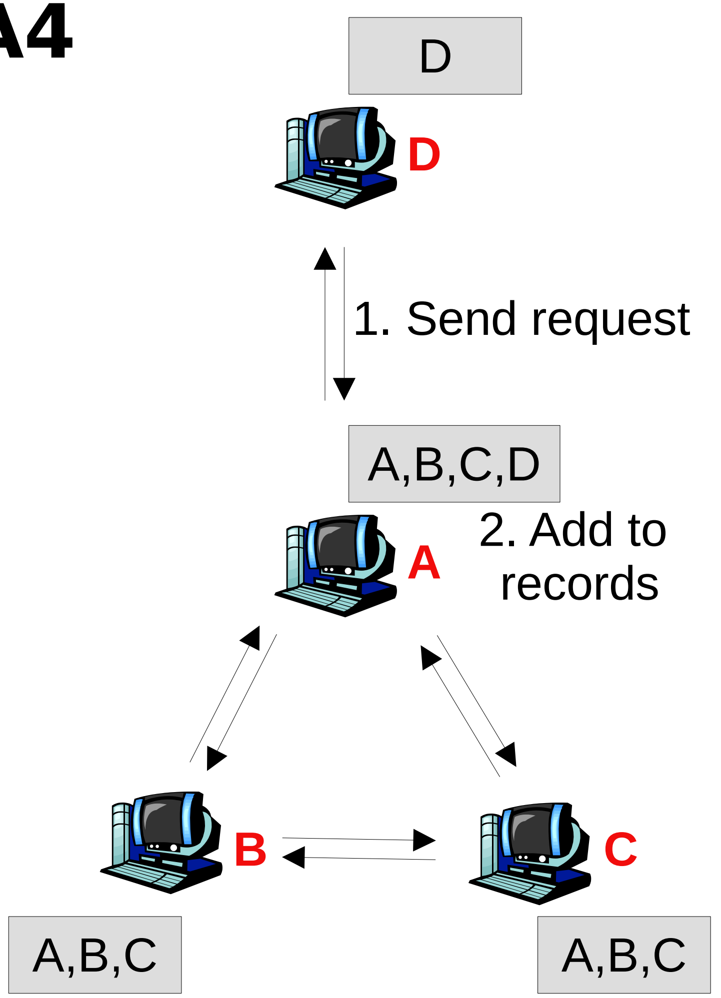
Peer to Peer Network in A4

- 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



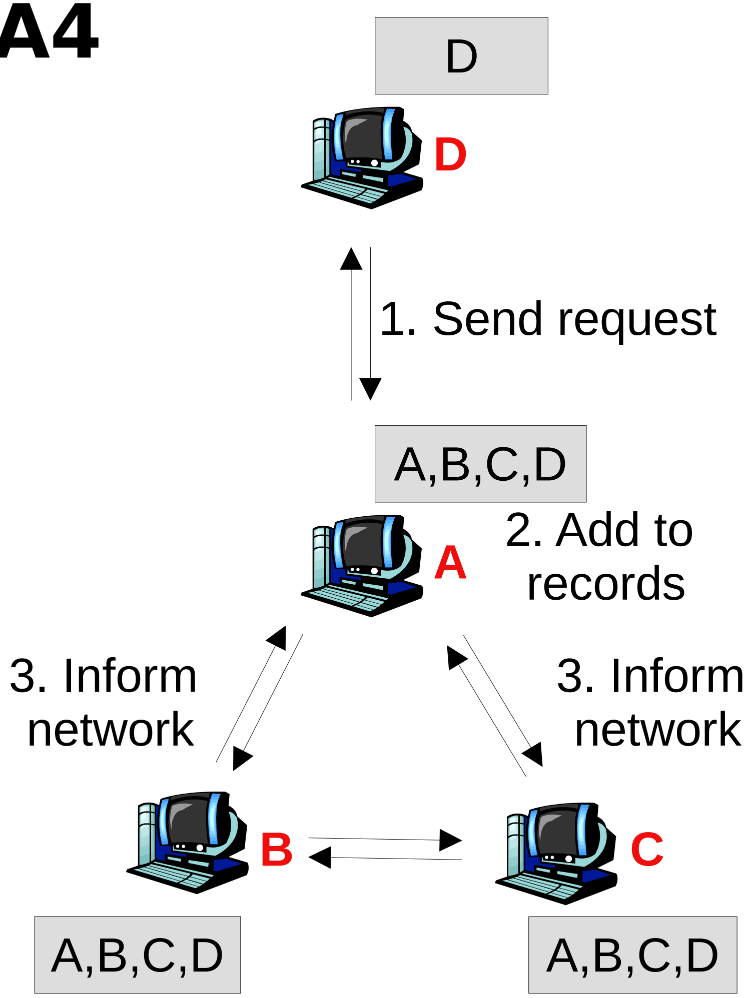
Peer to Peer Network in A4

- 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



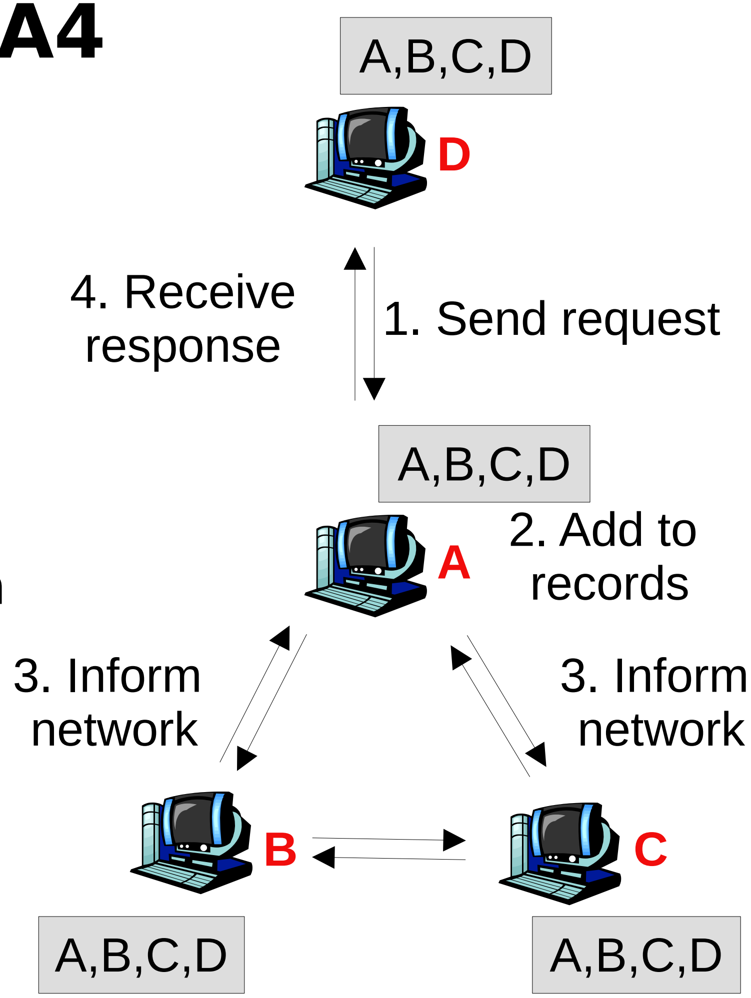
Peer to Peer Network in A4

- 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



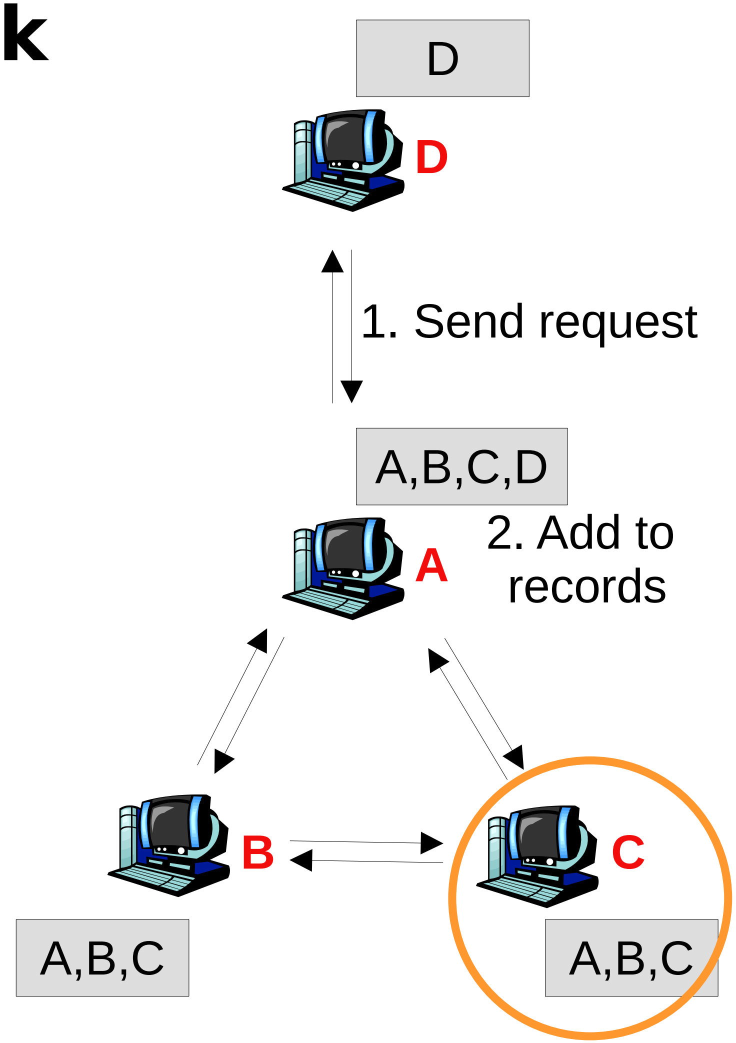
Races, across the network

- 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 slooooow
- 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

Bye Bye

- This is the last lecture from me
  - I’ll still be at cafes until A4 is handed in
  - And still reachable on the Discord
- Please provide feedback
  - First time giving the first section of CompSys
  - Teaching is my career focus, I do read feedback
- Bachelors project supervision
  - CSP! Networking, or in hardware with FPGAs
  - GPU optimisation and HPC
  - Scientific Workflows