pthread_mutex_t → type

- init
- lock
- ! trylock ← don't use unless you know what you're doing
- unlock
- destroy

```
lock(&m)
// critical section
unlock(&m)
```

pthread_RWlock_t

- init
- Rdlock
- Wrlock
- unlock
- destroy

readers	writers
rdlock(&l)	wrlock(&l)
// consult	// change
// critical	// critical
// variables	// variables
unlock(&l)	unlock(&l)

pthread_cond_t

- init
- wait
- signal
- broadcast
- destroy

```
waiting                          | modifier
lock(&m)                         | lock(&m)
while(!condition)                | if(condition)
    wait(&c,&m)                  |     signal(&c)
// do stuff                      | // do stuff
unlock(&m)                       | unlock(&m)
```

inside wait:
unlock(&m)
wait to wakeup
lock(&m)

pthread_barrier_t

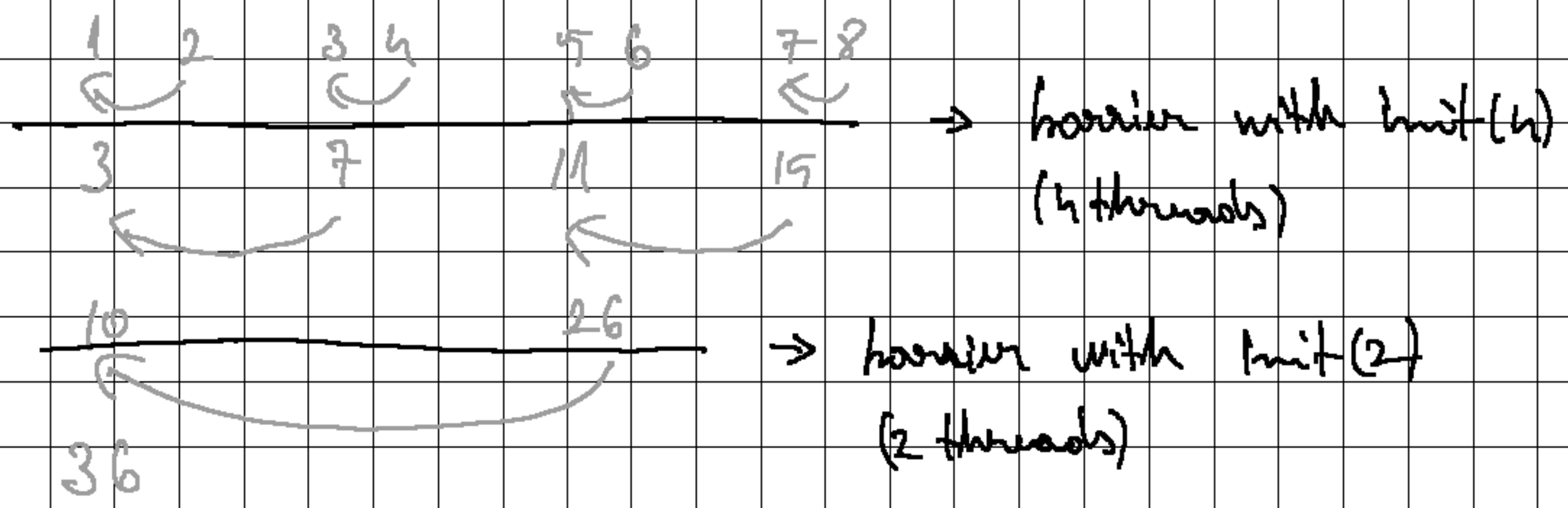
init

wait

destroy

init(4)

Threads will call: wait(&b)



(A barrier makes the threads wait for everyone then continue)

Sem_t

init

wait

post

destroy

Semaphore limits the threads inside the critical section

Ex init(1) - only 1 thread will call wait, the rest sit until it finishes

1- binary semaphore

init(3)

Threads: wait(&s)

// critical section

post(&s)

```
#include <stdio.h>
#include <pthread.h>
int turn=0;
pthread_mutex_t m;

void *a(void *a){
int i=0;
while(i<100){
    pthread_mutex_lock(&m);
    if(turn==0){
        printf("a\n"); i++; turn=1;
    }
    pthread_mutex_unlock(&m)
}
return NULL
}
void *b(void *b){
int i=0
while(i<100){
    pthread_mutex_lock(&m);
    if(turn==1){
        printf("b\n"); i++; turn=0;
    }
    pthread_mutex_unlock(&m)

}
return NULL
}
```

```
int main(int argc, char ** argv){
    pthread_t ta,tb;
    pthread_mutex_init(&m,NULL);
    pthread_create(&ta, NULL, fa, NULL);
    pthread_create(&tb, NULL, fb, NULL);
    pthread_join(ta, NULL);
    pthread_join(tb, NULL);
    pthread_mutex_destroy(&m);
    return 0;
}
```

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t ma, mb;

void fa(void *a){
    int i;
    for(i=0; i<100; i++){
        pthread_mutex_lock(&ma);
        printf("a\n");
        pthread_mutex_unlock(&mb);
    }
    return NULL;
}

void fb(void *b){
    int i;
    for(i=0; i<100; i++){
        pthread_mutex_lock(&mb);
        printf("b\n");
        pthread_mutex_unlock(&ma);
    }
    return NULL;
}
```

```
int main(int argc, char ** argv){
    pthread_t ta, tb;
    pthread_mutex_init(&ma, NULL);
    pthread_mutex_init(&mb, NULL);
    pthread_mutex_lock(&mb);
    pthread_create(&ta, NULL, fa, NULL);
    pthread_create(&tb, NULL, fb, NULL);
    pthread_join(ta, NULL);
    pthread_join(tb, NULL);
    pthread_mutex_destroy(&m);
    return 0;
}
```

```
i = rand() %. 1000;
while(1){
    j = rand() %. 1000;
    if (i != j) {
        break
    }
}
lock( &m[i]);
lock( &m[j]);
// swap
unlock( &m[i]);
unlock( &m[j]);
```

```
if (j < i){
    aux = j;
    j = i;
    i = aux;
}
```

	i	j
A	3	5
B	5	2
C	2	3
X	1	2
Y	2	1

Solution

We need to lock in the same order

⇒ it will lock up (dead lock)

lock (&m[1])	lock(&m[2])
lock(&m[2])	lock(&m[1])