Test-and-Set (mutex)

```c
int TestAndSet(int *ptr, int new) {
    int old = *ptr; // fetch old value at ptr
    *ptr = new; // store 'new' into ptr
    return old; // return the old value
}
```

```c
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ;   // spin-wait
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

```
.var mutex
.var count

.main
.top

.acquire
mov  $1, %ax
xchg %ax, mutex     # atomic swap of 1 and mutex
test $0, %ax        # if we get 0 back: lock is free!
jne  .acquire       # if not, try again

# critical section
mov  count, %ax     # get the value at the address
add  $1, %ax        # increment it
mov  %ax, count     # store it back

# release lock
mov  $0, mutex

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

Anfangsinitialisierung.

% ax = dynamisch vergeben

% bx = Threadanzahl

Ticket Lock

```c
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}
void unlock(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```

Anfangsinitialisierung

% ax = ticket = 0

% bx = Anzahl Threads

% cx = turn = 0

```
.var ticket
.var turn
.var count

.main
.top

.acquire
mov $1, %ax
fetchadd %ax, ticket  # grab a ticket
.tryagain
mov turn, %cx         # check if it's your turn
test %cx, %ax
jne .tryagain

# critical section
mov  count, %ax       # get the value at the address
add  $1, %ax          # increment it
mov  %ax, count       # store it back

# release lock
mov $1, %ax
fetchadd %ax, turn

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

Test-and-Set (flag)

```
.var flag
.var count

.main
.top

.acquire
mov  flag, %ax      # get flag
test $0, %ax        # if we get 0 back: lock is free!
jne  .acquire       # if not, try again
mov  $1, flag       # store 1 into flag

# critical section
mov  count, %ax     # get the value at the address
add  $1, %ax        # increment it
mov  %ax, count     # store it back

# release lock
mov  $0, flag       # clear the flag now

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

Peterson's Algorithmus

```
.var flag  2          # Array 'flag' für die beiden Threads (zwei Speicherplätze)
.var turn             # Variable 'turn', um die Reihenfolge der Threads zu steuern
.var count            # Variable 'count', um eine Aktion (z. B. Eintritt in den kritischen Abschnitt) zu zählen

.main
lea flag, %fx         # Lade die Adresse des Arrays 'flag' in %fx

mov %bx, %cx          # Kopiere den Index des aktuellen Threads in %cx (Thread 0 oder 1)
neg %cx               # Negiere %cx, um den anderen Thread zu identifizieren
add $1, %cx           # Korrigiere den Wert von %cx, sodass %cx = 1 - %bx (der andere Thread)

.acquire              # Beginn des Beitritts in den kritischen Abschnitt
mov $1, 0(%fx,%bx,4)  # Setze flag[%bx] = 1 (signalisiere "möchte eintreten")
mov %cx, turn         # Setze turn auf den anderen Thread

.spin1                # Warte darauf, dass der andere Thread den kritischen Abschnitt verlässt
mov 0(%fx,%cx,4), %ax # Lade flag[1 - %bx] (Status des anderen Threads) in %ax
test $1, %ax          # Teste, ob flag[1 - %bx] == 1 (der andere Thread möchte eintreten)
jne .fini             # Wenn der andere Thread nicht eintreten will (flag[1 - %bx] == 0), weiter zu .fini

.spin2                # Zusätzliche Überprüfung der Reihenfolge
mov turn, %ax         # Lade den Wert von 'turn' in %ax
test %cx, %ax         # Teste, ob turn == 1 - %bx (der andere Thread ist an der Reihe)
je .spin1             # Wenn ja, gehe zurück zu .spin1 und warte

.fini                 # Kritischer Abschnitt betreten
mov count, %ax        # Lade den aktuellen Wert von 'count'
add $1, %ax           # Erhöhe 'count' um 1 (z. B. um einen Eintritt zu zählen)
mov %ax, count        # Speichere den neuen Wert von 'count'

.release              # Verlasse den kritischen Abschnitt
mov $0, 0(%fx,%bx,4)  # Setze flag[%bx] = 0 (signalisiere "möchte nicht mehr eintreten")
-
mov %cx, turn         # Setze turn auf den anderen Thread
halt                  # Beende den Ablauf
```

```
int turn = 0; // shared
Boolean flag[2] = {false, false};

Void acquire() {
    flag[tid] = true;
    turn = 1-tid;
    while (flag[1-tid] && turn == 1-tid) /* wait */ ;
}

Void release() {
    flag[tid] = false;
}
```

Anfangsinitialisierung:

%ax = Threadstatus = dynamisch

%bx = Index des aktuellen Threads = 1 oder 0