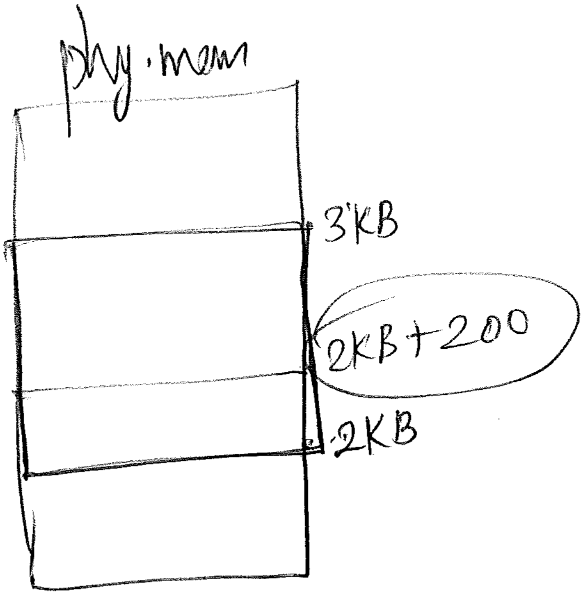
Midterm

Fri, Oct 20th

5:30 - 7:30 pm.

RockIt

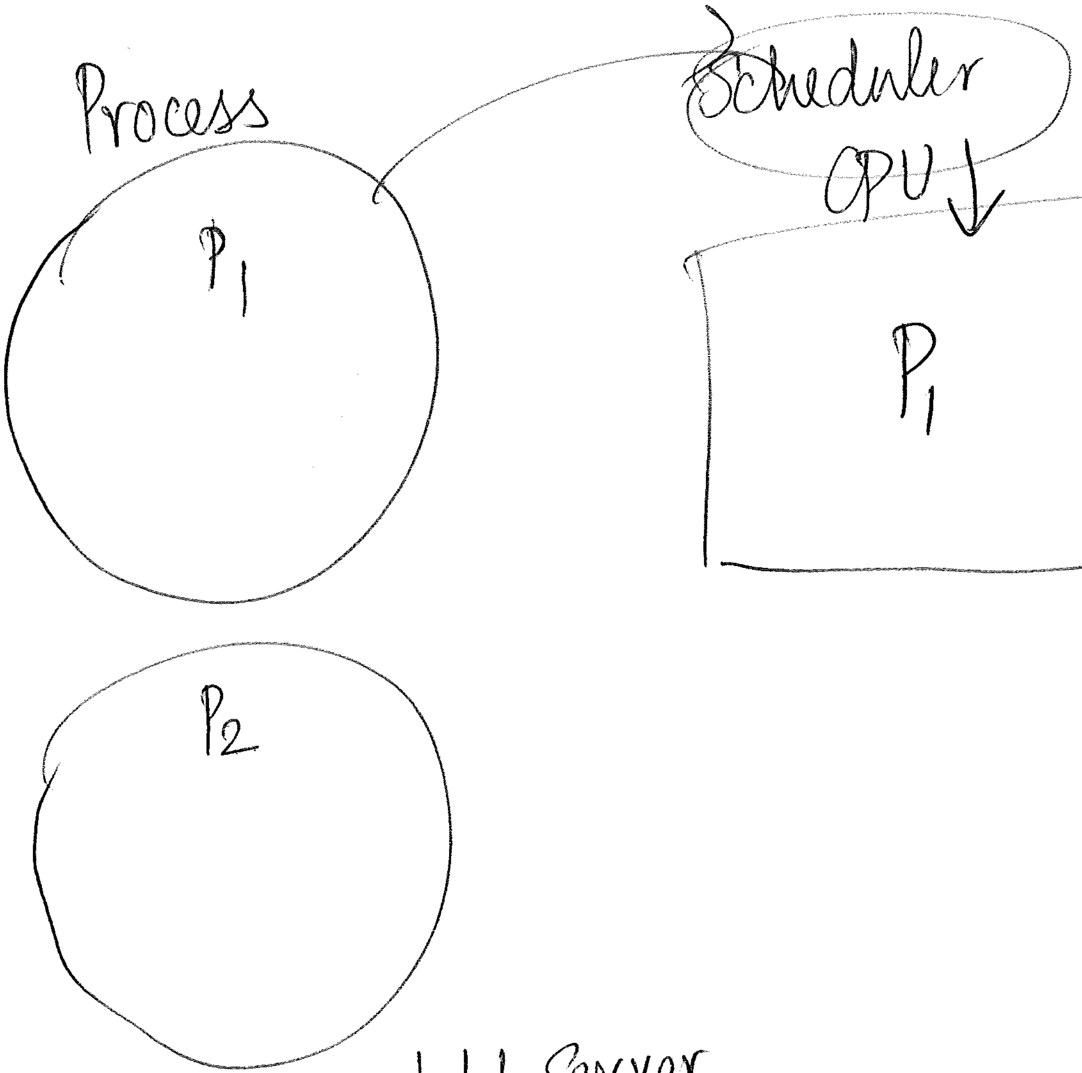
V.A: 200



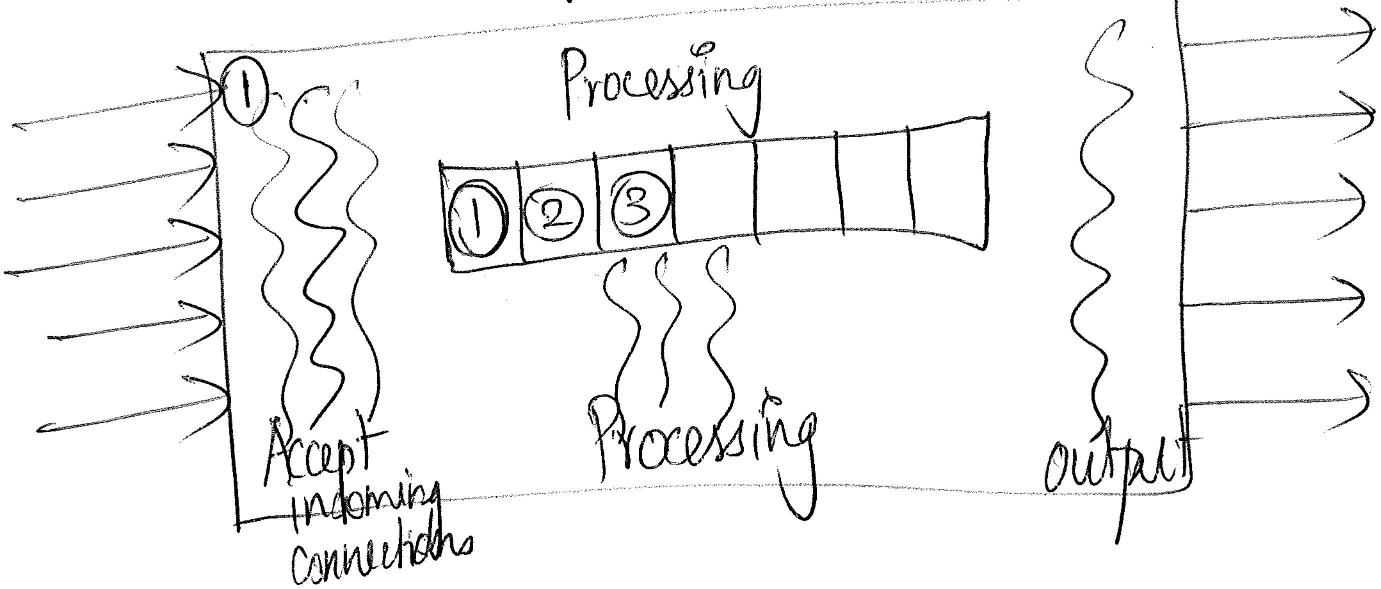
1. Virtualization < CPU ✓ / Memory ✓

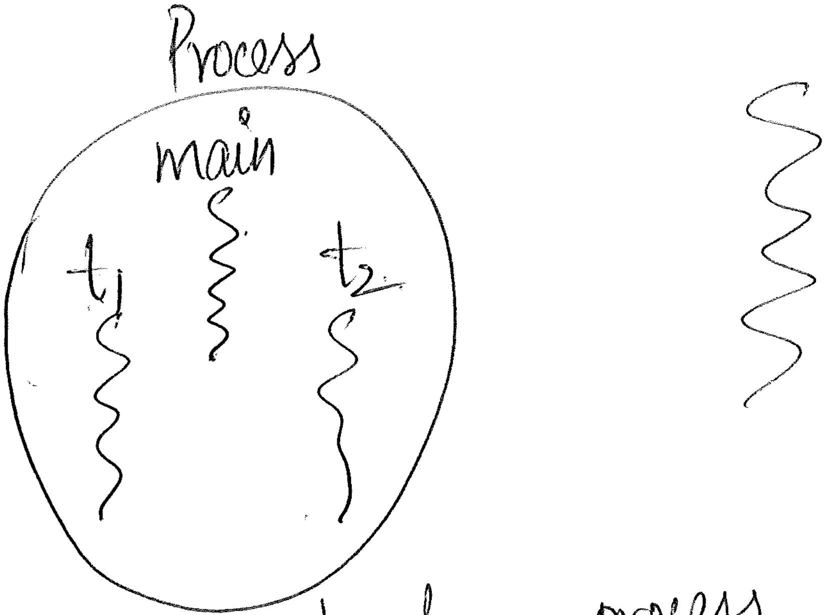
2. Concurrency *

3. Persistence.


Process
Scheduler
CPU
P1
P1
P2

Web Server


Processing
1
1
2
3
Accept incoming connections
Processing
output



thread - component of a process.

Threads

SHARE

- Heap.
- Static data.
- ~~Stack~~
- Code

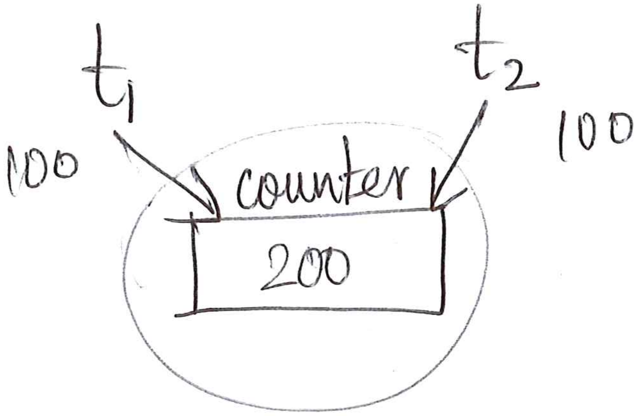
Address Space.

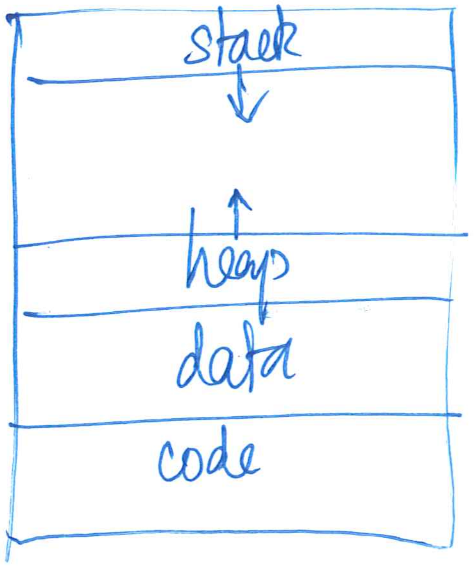
NOT SHARED

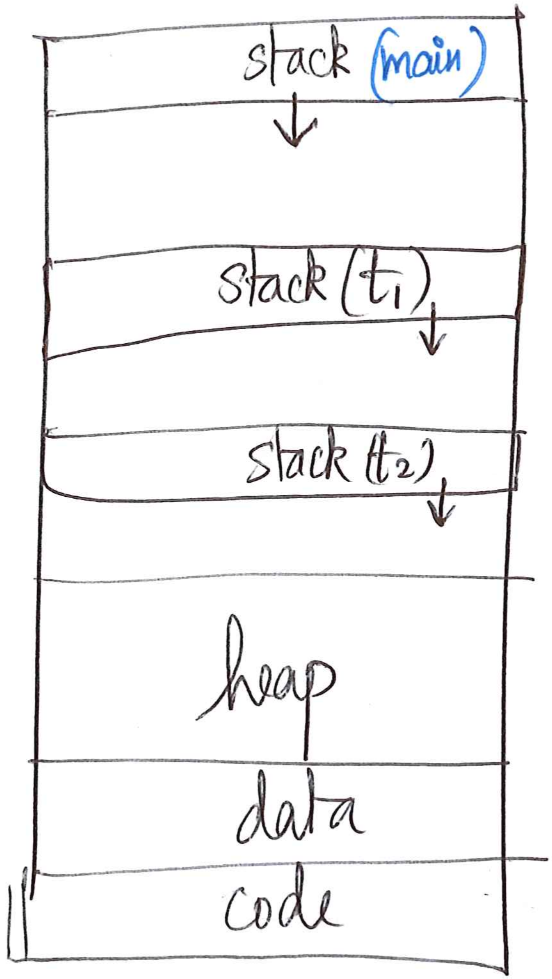
- Stack

Registers

- PC (IP)
- %eax, ... %edx.
- SP / BP

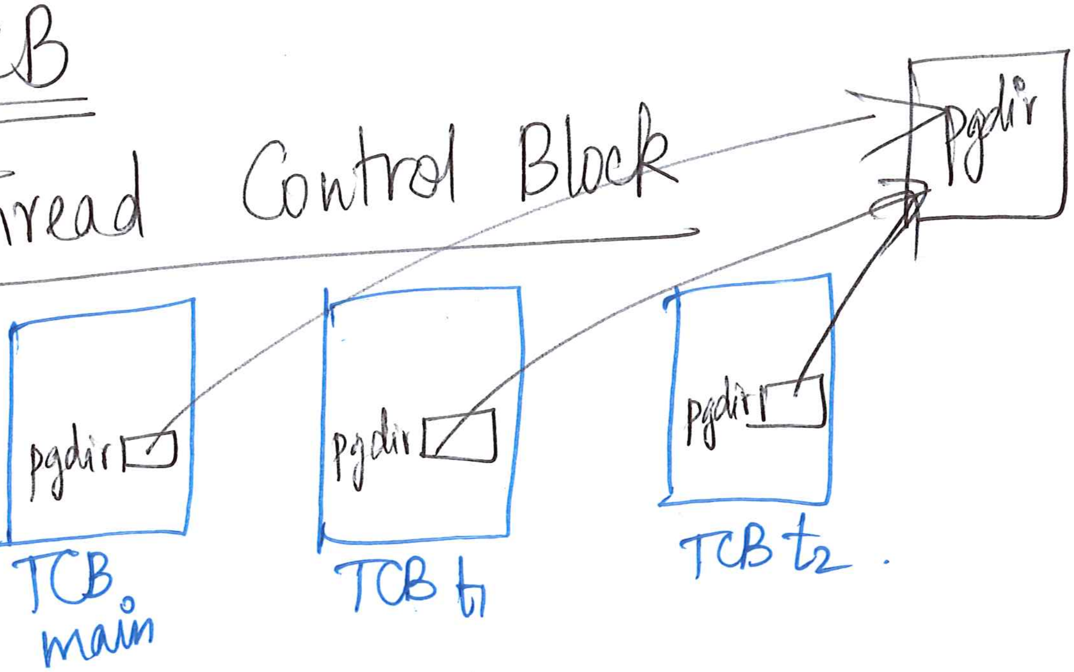

t1
t2
100
100
counter
200


stack
heap
data
code


stack (main)
stack (t1)
stack (t2)
heap
data
code

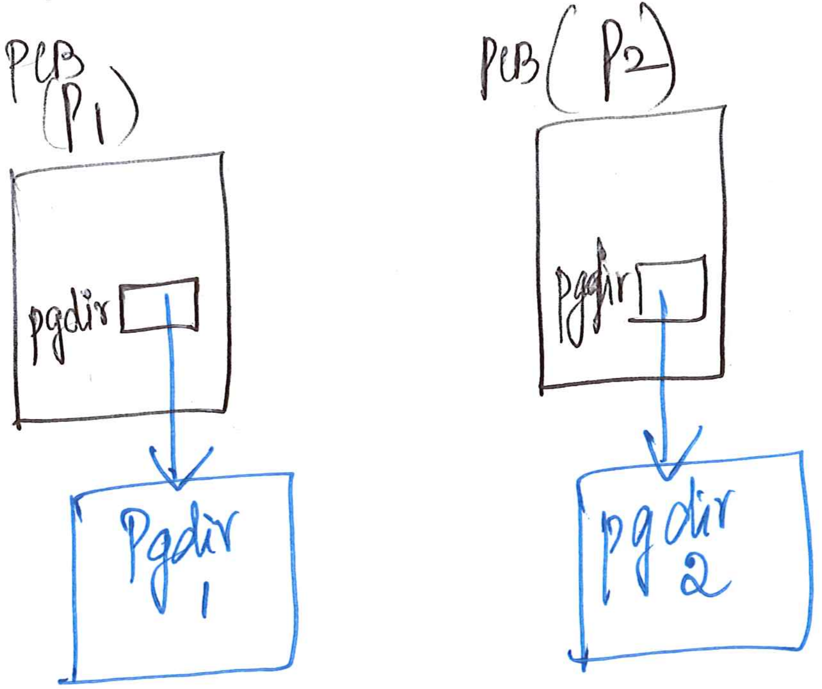
PCB

Thread Control Block

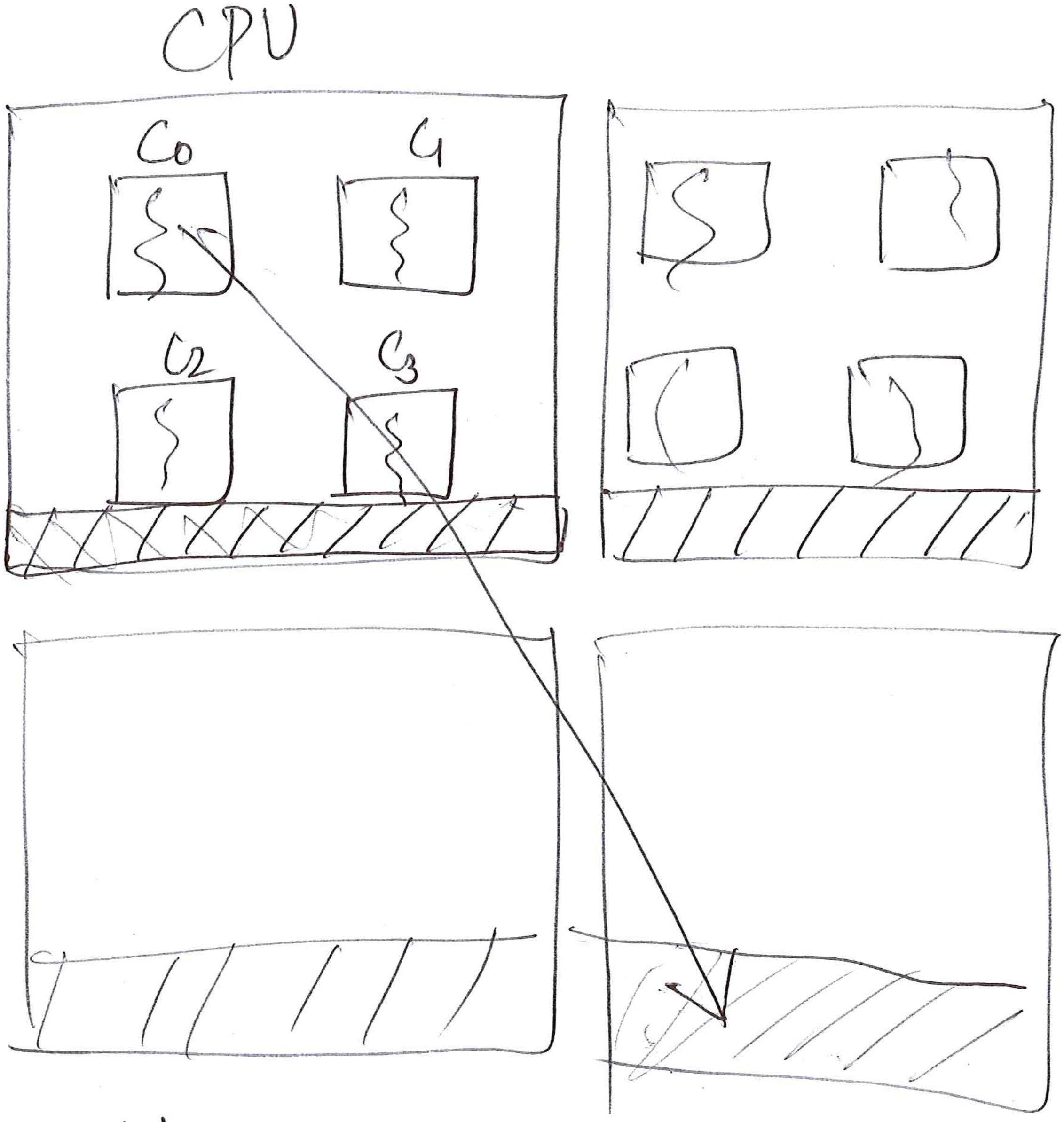


counter = counter + 1;

YOU ARE THE EVIL SCHEDULER!

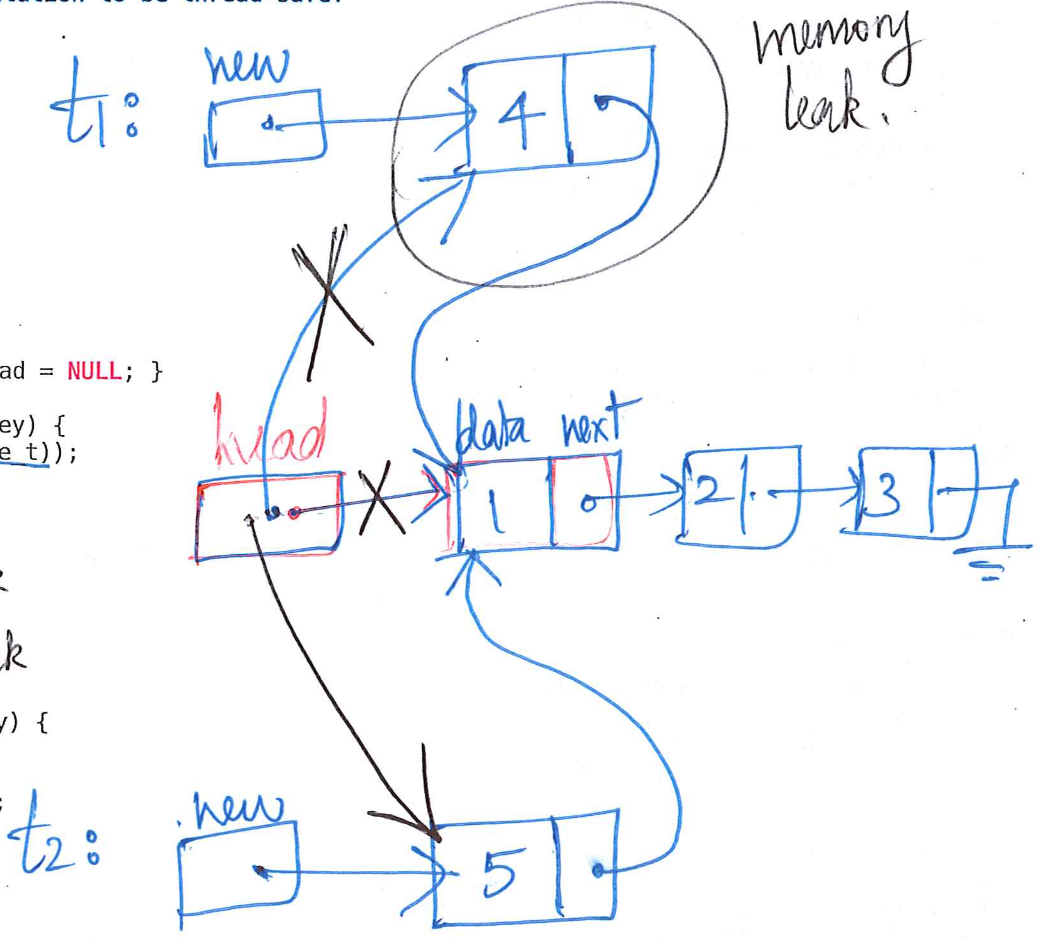


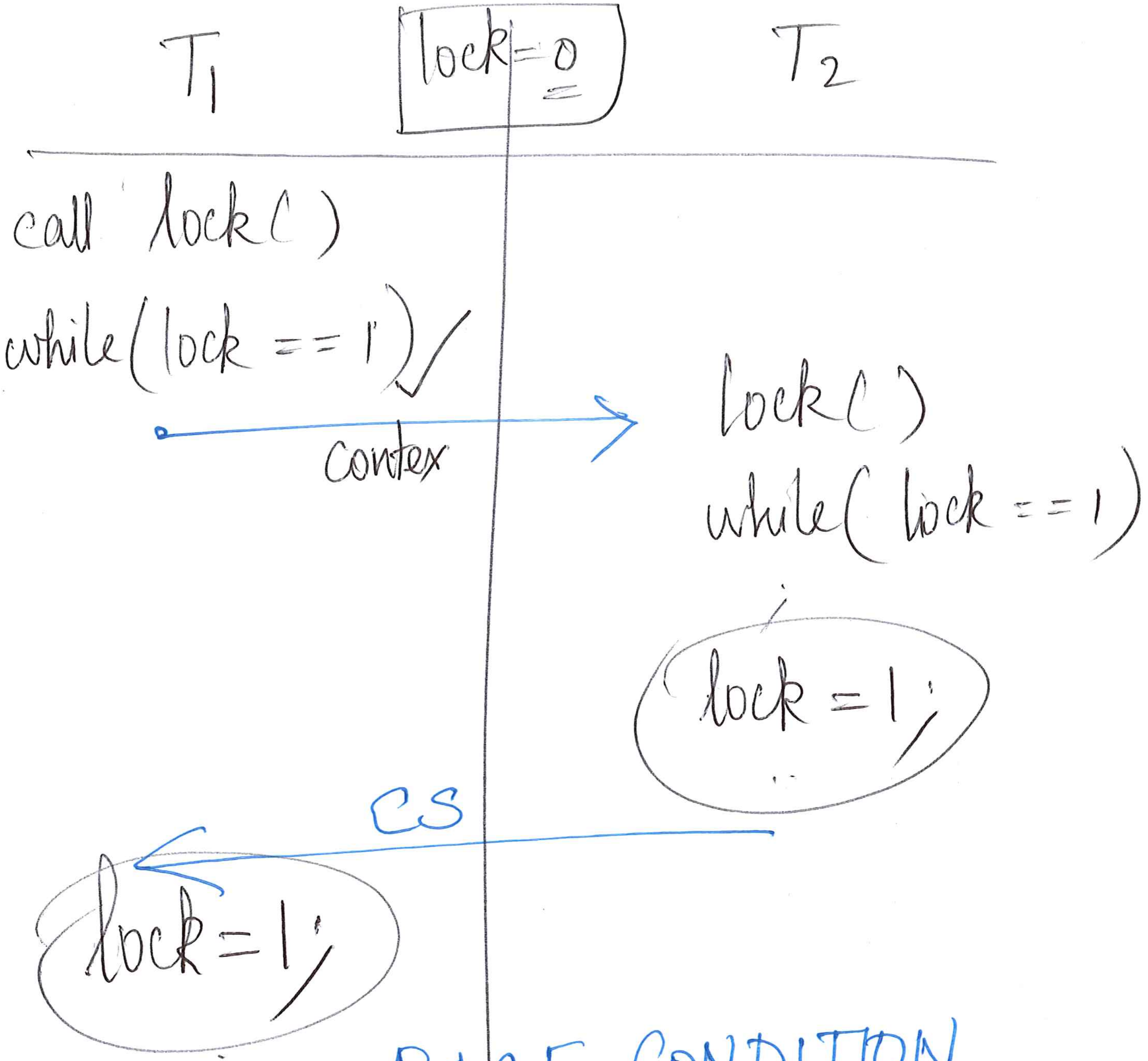
pid:1	1	1	pid=2	pid=2	
tid:1	2	3	tid=1	tid=2	



NUMA

```
// Make this Linked List implementation to be thread-safe!

#include <stdio.h>
#include <stdlib.h>

typedef struct __node_t {
  int key;
  struct __node_t *next;
} node_t;

typedef struct __list_t {
  node_t *head;
} list_t;

void List_Init(list_t *L) { L->head = NULL; }

void List_Insert(list_t *L, int key) {
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    return;
  }
  new->key = key;        lock
  new->next = L->head;
  L->head = new;         unlock
}

int List_Lookup(list_t *L, int key) {
  node_t *tmp = L->head;
  while (tmp) {
    if (tmp->key == key) return 1;
    tmp = tmp->next;
  }
  return 0;
}

void List_Print(list_t *L) {
  node_t *tmp = L->head;
  while (tmp) {
    printf("%d ", tmp->key);
    tmp = tmp->next;
  }
  printf("\n");
}

int main(int argc, char *argv[]) {
  list_t mylist;
  List_Init(&mylist);
  List_Insert(&mylist, 10);
  List_Insert(&mylist, 30);
  List_Insert(&mylist, 5);
  List_Print(&mylist);
  printf("In List: 10? %d 20? %d\n", List_Lookup(&mylist, 10),
         List_Lookup(&mylist, 20));
  return 0;
}
```





T1 | lock = 0 | T2

call lock( )

while(lock == 1') ✓

contex

lock( )

while(lock == 1)

lock = 1;

CS

lock = 1;

RACE CONDITION

TEMPLATE: FILL THIS IN TO MAKE YOUR OWN LOCK

```
typedef struct __lock_t {
   // whatever data structs you need goes here
} lock_t;

void init(lock_t *lock) {
   // init code goes here
}

void acquire(lock_t *lock) {
   // lock acquire code goes here
}

void release(lock_t *lock) {
   // lock release code goes here
}
```

FIRST PRIMITIVE: TEST-AND-SET (or ATOMIC EXCHANGE)

```
// given ptr, sets *ptr to new value; returns the old value at *ptr
int Exchange(int *lock, int new) {
   int old = *lock;
   *lock = new;
   return old;
}
```

SECOND PRIMITIVE: COMPARE-AND-SWAP

```
int CompareAndSwap(int *ptr, int expected, int new) {
   int actual = *ptr;
   if (actual == expected)
           *ptr = new;
   return actual;
}
```

THIRD PRIMITIVE(S): LOAD-LINKED, STORE-CONDITIONAL

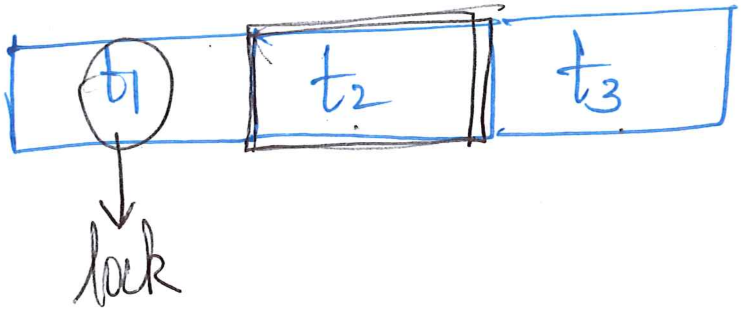
```
int LoadLinked(int *ptr) {
   return *ptr;
}

int StoreConditional(int *ptr, int value) {
   if (no one has updated *ptr since LoadLinked to this address) {
          *ptr = value;
          return 1; // success
   } else {
          return 0; // fail (does not do the store)
   }
}
```

lock

~~0~~ 1

```
Exchange(&lock, 1)
return = 0.

SpinLock(int * lock). {
  while (CAS(lock, 0, 1)
            == 1)
     ;
}

SpinUnlock(int *lock) {
  //CAS(lock, 1, 0);
    *lock = 0; //
}
```

	xchg(test_and_set)	CAS
1. Correctness	✓	✓
2. Fairness	✗	✗
3. Performance	✗	✗


t1
t2
t3
lock