1. What is your ultimate goal?

Your goal is to implement **Context Switching**.

Normally, the Operating System handles switching between programs. Here, you are making the `uthread` program switch between `thread_a`, `thread_b`, and `thread_c` manually. You need to make the CPU "stop" what it's doing in one function, save its state (registers), jump to another function, and later come back exactly where it left off.

The "Magic" Moment: When you call `thread_switch(old_context, new_context)`, the CPU enters the function as Thread A but exits the function as Thread B.

---

2. Where do I start?

Don't write code yet. First, understand the **Thread Context**.

A thread's "life" is stored in its registers. If you change the **Stack Pointer (sp)** and the **Program Counter/Return Address (ra)**, you have effectively changed which thread is running.

---

3. What files do I need to change?

You only need to modify three specific areas across two files:

File 1: `user/uthread.c`

- **Step 1: Define the Context Struct.** Find the place to define `struct thread_context`. Based on your instructions, it must hold: `ra`, `sp`, `s0-s11`. These are all `uint64`.
- When we switch threads, we must:
  1. Save current thread's registers
  2. Load another thread's registers
- Where do we store those registers?
  - 👉 Inside the thread struct.
  - ```
    - This struct is literally:
    ```

    > A container to store CPU register values.
    > -That's why we create:
    ```

```
struct thread_context {
  uint64 ra;
  uint64 sp;
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};
```

Why these registers?

Register	Name	Role in Context Switching
ra	Return Address	Stores the memory address where the thread will resume execution.
sp	Stack Pointer	Points to the thread's private stack, preserving its local variables and function calls.
s0 / fp	Frame Pointer	Used to track the current stack frame; helps in navigating local function data.
s1 - s11	Saved Registers	Callee-saved registers that store long-term variables and intermediate calculation results.

- Because of RISC-V Calling convention.
  - RISC-V Calle saved registers are:
    - ra
      - ra = return address
      - when a function runs:
        - jal function
      - CPU stores return location in ra
      - so if we switch threads and don't save ra:
        - when we resume, it returns to the wrong place and it will crash.
    - sp
      - the most important
        - each thread has its own stack
          - `char stack[STACK_SIZE]`
        - Stack pointer tells the CPU:

            - where is the top of the stack?
        - if you don't restore the stack, the thread will use someone else's stack and everything breaks
    - s0-s11
- These must be preserved across function calls.
    - If we don’t save them:
        - When we switch back,
            - the thread’s local variables and return address are destroyed.
            - So we save exactly those.

- **Step 2: Update `struct thread`.** Add a `struct thread_context context;` member to the existing thread structure so each thread has a place to save its "soul."

```
struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct thread_context context; /* Add this line to the existing struct
thread */
};
```

Why does struct thread contain context?

- look at the code
    - think of it like each thread has its own CPU snapshot
        - each thread objects coantins:
            - its own stack
            - its own state (RUNNING, RUNNABLE, ETC)
            - Its own saved CPU registers
    - When we switch:

```
        Save current CPU → current_thread->context
        Load next_thread->context → CPU
```

```
    - now Cpu becomes that thread which is the point
```

- **Step 3: Implement `thread_create`.** When a thread is born, you must "fake" its first context.
  - Set `t->context.ra` to the function pointer (`func`).
  - Set `t->context.sp` to the **top** of that thread's stack (since stacks grow down, this is `t->stack + STACK_SIZE`).

What `thread_create()` Is Actually Doing
It does 3 things:

1. Finds a free thread slot
2. Marks it RUNNABLE
3. **Fakes its first CPU context**
   That's the key idea.

Step1, find a free thread:
This just scans the thread array and finds an unused slot.

```
  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
```

Step 2, mark it runnable:

```
  t->state = RUNNABLE;
```

This tells the scheduler:

- This thread is ready to run.
- But it still has never executed.
- Now comes the magic.

Step3, fake the context:
Remember:
When `thread_switch()` runs, it will:

```
ld ra, 0(a1)
ld sp, 8(a1)
ret
```

So when switching to a thread, CPU will:

1. Load `ra`

2. Load `sp`
3. Execute `ret`
4. Jump to whatever `ra` contains
   So we must prepare those values.

Why Set `ra = func`?

```
t->context.ra = (uint64)func;
```

This means:
When the thread is switched to for the first time:

- `ra` will contain the address of `func`
- CPU executes `ret`
- `ret` jumps to `ra`
- CPU starts executing `func()`

So the thread begins at that function.

We are tricking the CPU.

- It thinks it is "returning",
- but actually it is starting a new thread.

That's why we say we are **faking the first context**.

Why Set `sp` to Top of Stack?

```
t->context.sp = (uint64)&t->stack[STACK_SIZE - 1];
```

Usually better to use:

```
t->context.sp = (uint64)(t->stack + STACK_SIZE);
```

Because:

- *`STACK_SIZE - 1` points to last byte*
- *But stack pointer should point just past the array*
- *That's cleaner and safer*

Each thread has its own stack:

```
char stack[STACK_SIZE];
```

Stacks grow downward in memory.
So the top of stack is the highest address.
When the thread starts running:

- It needs a valid stack
- Local variables will be pushed there
- Function calls will use it

If `sp` is wrong:
💥 Stack corruption
💥 Crash

So we give it a fresh stack.

```
void
thread_create(void (*func)())
{
  struct thread *t;
  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
  t->state = RUNNABLE;
  // YOUR CODE HERE
  // 1. Set the Return Address (ra) to the function the thread should run

  t->context.ra = (uint64)func;

  // 2. Set the Stack Pointer (sp) to the top of the allocated stack
  // Remember: stacks grow DOWN, so point to the end of the array

  t->context.sp = (uint64)&t->stack[STACK_SIZE - 1];
}
```

- **Step 4: Update `thread_schedule`.** Find the comment where it says to switch. You will call `thread_switch(&t->context, &next_thread->context)`.
  When scheduler does:

```
thread_switch(&current->context, &next->context);
```

Inside assembly:

1. Save current registers
2. Load next thread's registers
3. `ret`

If this is the first time this thread runs:

- `ra = func`
- `sp = top of its stack`
- `ret` jumps into `func`

And boom.
The thread starts.

```
void
thread_schedule(void)
{
  struct thread *t, *next_thread;
  /* Find another runnable thread. */
  next_thread = 0;
  t = current_thread + 1;
  for(int i = 0; i < MAX_THREAD; i++){
    if(t >= all_thread + MAX_THREAD)
      t = all_thread;
    if(t->state == RUNNABLE) {
      next_thread = t;
      break;
    }
    t = t + 1;
  }

  if (next_thread == 0) {
    printf("thread_schedule: no runnable threads\n");
    exit(-1);
  }

  if (current_thread != next_thread) {         /* switch threads?  */
    next_thread->state = RUNNING;
    t = current_thread;
    current_thread = next_thread;

    /* YOUR CODE HERE
     * Invoke thread_switch to switch from t to next_thread:
     * thread_switch(??, ??);
```

```
    */
    /* Invoke thread_switch to switch from t (old) to next_thread (new) */

   thread_switch((uint64)&t->context, (uint64)&next_thread->context);

  } else
   next_thread = 0;
}
```

File 2: `user/uthread_switch.S`

- **Step 5: Write the Assembly.** This is the "brain transplant" surgery.
  - **Save:** Use `sd` (store doubleword) to move current registers (`ra` , `sp` , `s0` , etc.) into the memory pointed to by the first argument (`a0`).
  - **Restore:** Use `ld` (load doubleword) to move values from the memory pointed to by the second argument (`a1`) into the physical registers.
  - **Finish:** Use `ret` . Since you just loaded a new `ra` into the register, `ret` will jump to the new thread's code.

```
    .text

    /*
         * save the old thread's registers,
         * restore the new thread's registers.
         */

    .globl thread_switch
thread_switch:
    /* YOUR CODE HERE */
    /* Save registers of the old thread (into address in a0) */
    sd ra, 0(a0)
    sd sp, 8(a0)
    sd s0, 16(a0)
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)
```

```
    /* Restore registers of the new thread (from address in a1) */
    ld ra, 0(a1)
    ld sp, 8(a1)
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)
    ret    /* return to ra */
```

Why Does the Struct Have to Match the Assembly?
In your assembly, you will see something like:

```
sd ra, 0(a0)
sd sp, 8(a0)
sd s0, 16(a0)
```

That means:
Memory layout MUST be:

```
offset 0  → ra
offset 8  → sp
offset 16 → s0
```

Because each `uint64` = 8 bytes.

If you change order in C struct:

💥 Offsets break
💥 Assembly loads wrong values
💥 Context switch fails

So the struct layout is designed to match the assembly offsets.

That's why it looks so “manual”.

1️⃣ What this assignment is about

You already did **Task 1** (uthread) on xv6, learning about manual context switching.

Task 2 moves to a **real Unix system** (Linux or macOS) with **pthreads**, which is the standard threading library in Unix.

The goal is:

- Implement a **shared hash table** that multiple threads can safely access at the same time.
- Learn about **race conditions** (what happens if threads access shared memory at the same time without protection).
- Learn about **mutex locks** to prevent race conditions.
- Implement **per-bucket locking** for efficiency (so threads only block each other if they access the same bucket, not the whole table).

2️⃣ Files involved

From the assignment description, the main files are in the `notxv6/` folder:

1. **ph-without-locks.c**
   - Already exists, demonstrates a **hash table accessed by multiple threads without locks**.
   - You **don't touch this file**. It will show race conditions when multiple threads run.
2. **ph-with-mutex-locks.c**
   - This is the file you will **modify**.
   - Copy `ph-without-locks.c` to this file if it's empty.
   - Your task is to **add mutex locks** to make the hash table thread-safe.
3. **Makefile**
   - Already configured to compile these files using gcc and pthreads.
   - Usually you don't need to touch it unless you need to check compilation.

3️⃣ What the hash table looks like

- **5 buckets**: `NBUCKET = 5`
- **100,000 keys**: `NKEYS = 100000`
- Each bucket has a **linked list** to store collisions.
- `bucket = key % NBUCKET` determines which bucket a key goes to.

4️⃣ What goes wrong without locks

If two threads do:
put(key)
at the same time:

- They might both try to update the same bucket's linked list.
- One thread might overwrite the `next` pointer another thread is writing.
- Result: keys are **lost**, i.e., "missing keys".

✅ This is called a **race condition**.

1️⃣ Hash Table Structure

```
struct entry {
  int key;
  int value;
  struct entry *next;
};
struct entry *table[NBUCKET];
```

- `table` is an **array of 5 buckets** (`NBUCKET = 5`).
- Each bucket is a **linked list** (`struct entry *next`) to handle collisions.
- Each `entry` stores a key and value.

✅ This is the "shared data" multiple threads will access concurrently.

2️⃣ Mutex Locks

`static pthread_mutex_t locks[NBUCKET];`

- There is **one mutex per bucket**.
- Threads must **lock a bucket before modifying it** (`put()`), so two threads don't change the same linked list at the same time.

3️⃣ `put()` Function (Thread-Safe Insertion)

`int i = key % NBUCKET; // Determine the bucket`
`pthread_mutex_lock(&locks[i]); // LOCK bucket`

```
// check if key exists; if not, insert
...
pthread_mutex_unlock(&locks[i]); // UNLOCK bucket
```

- Only locks the **specific bucket** (per-bucket locking).
- This prevents data corruption **while still allowing other threads to insert into other buckets simultaneously**.
- This is the critical section: any memory update that could race must be inside the lock.

4️⃣ `get()` Function (Lookup / Read-Only)

```
struct entry* get(int key) {
int i = key % NBUCKET;
...
}
```

- **No lock needed** because:
  1. This happens **after all puts are finished**.
  2. Reading a linked list does not modify memory.

✅ This is safe without locks.

5️⃣ Threads Creation

PUT Threads

```
for(int i = 0; i < nthread; i++) {
pthread_create(&tha[i], NULL, put_thread, (void *)(long)i);
}
```

- Creates multiple threads (`nthread`) that each insert a **portion of the keys**.
- Each thread calls `put_thread()` , which calls `put()` for its range of keys.

GET Threads

```
for(int i = 0; i < nthread; i++) {
pthread_create(&tha[i], NULL, get_thread, (void *)(long)i);
}
```

- After insertion, threads are created to **check that all keys are present**.
- Each thread goes through **all keys** and counts missing keys.

6️⃣ Timing / Performance Measurement

```
double t0 = now(); // start
...
double t1 = now(); // end
printf("%d puts, %.3f seconds, %.0f puts/second\n", ...);
```

- Measures **throughput** (how many puts/gets per second).
- Shows the **speedup** if multiple threads are used.

7️⃣ Mutex Initialization and Cleanup

```
for(int i = 0; i < NBUCKET; i++)
pthread_mutex_init(&locks[i], NULL);

...
for(int i = 0; i < NBUCKET; i++)
pthread_mutex_destroy(&locks[i]);
```

- Initialize locks **before threads run**.
- Destroy locks **after all threads finish**.
  ✅ This is exactly what the assignment asked.\

```
#include <stdlib.h>      // Standard library for malloc, free, and random

#include <unistd.h>      // Standard symbolic constants (like NULL)

#include <stdio.h>       // Standard I/O for printf

#include <assert.h>      // For assert() to verify program assumptions

#include <pthread.h>     // POSIX threads library for multi-threading

#include <sys/time.h>    // For gettimeofday high-resolution timing



#define NBUCKET 5        // The hash table has 5 buckets (slots)

#define NKEYS 100000     // Total number of keys to insert in the test
```

```
struct entry {           // Define a node for the linked list in each bucket

  int key;               // The lookup key

  int value;             // The stored value

  struct entry *next;    // Pointer to the next entry in the chain (collision
handling)

};

struct entry *table[NBUCKET]; // The actual hash table (array of pointers)

int keys[NKEYS];              // Global array to store random keys for testing

int nthread = 1;              // Global variable for the number of threads



// Global array of mutex locks: one unique lock for every bucket in the table

static pthread_mutex_t locks[NBUCKET];



double now()             // Helper function to get current time in seconds

{

 struct timeval tv;

 gettimeofday(&tv, 0);   // Get system clock with microsecond precision

 return tv.tv_sec + tv.tv_usec / 1000000.0; // Convert to total seconds

}



static void

insert(int key, int value, struct entry **p, struct entry *n)

{
```

```
  struct entry *e = malloc(sizeof(struct entry)); // Allocate memory for a new
node

  e->key = key;          // Set the key

  e->value = value;      // Set the value

  e->next = n;           // Point new node's next to the current head (n)

  *p = e;                // Update the bucket pointer to point to this new
head

}



static void put(int key, int value)

{

  int i = key % NBUCKET; // Determine which bucket the key belongs to

  pthread_mutex_lock(&locks[i]); // LOCK: Start of critical section for bucket
'i'



  struct entry *e = 0;

  for (e = table[i]; e != 0; e = e->next) { // Traverse the list in bucket 'i'

    if (e->key == key)   // If the key already exists...

      break;             // Stop searching

  }

  if(e){                 // If key was found...

    e->value = value;    // Update the existing key's value

  } else {               // If key is not in the table...

    insert(key, value, &table[i], table[i]); // Add it to the front of the
bucket
```

```
  }

  pthread_mutex_unlock(&locks[i]); // UNLOCK: End of critical section
}

static struct entry* get(int key)
{
  int i = key % NBUCKET; // Determine which bucket the key should be in
  struct entry *e = 0;
  // Searching through the bucket's linked list
  for (e = table[i]; e != 0; e = e->next) {
    if (e->key == key) break; // Found the key
  }
  return e;              // Return the entry (or 0 if not found)
}

static void * put_thread(void *xa)
{
  int n = (int) (long) xa; // Convert thread argument back to integer ID
  int b = NKEYS/nthread;   // Divide keys equally among threads

  for (int i = 0; i < b; i++) {
    put(keys[b*n + i], n); // Each thread puts its specific range of keys
```

```
  }

  return NULL;

}



static void * get_thread(void *xa)

{

  int n = (int) (long) xa; // Thread ID

  int missing = 0;         // Counter for keys that weren't found



  for (int i = 0; i < NKEYS; i++) { // Every thread checks for ALL keys

    struct entry *e = get(keys[i]);

    if (e == 0) missing++; // If key is missing, increment counter

  }

  printf("%d: %d keys missing\n", n, missing); // Report total missing keys

  return NULL;

}



int main(int argc, char *argv[])

{

  pthread_t *tha;        // Array of thread identifiers

  void *value;           // Return value from thread joins

  double t1, t0;         // Timing variables

```

```
  if (argc < 2) {        // Check if user provided thread count

    fprintf(stderr, "Usage: %s nthreads\n", argv[0]);

    exit(-1);

  }

  nthread = atoi(argv[1]); // Parse thread count from command line

  tha = malloc(sizeof(pthread_t) * nthread); // Allocate thread array

  srandom(0);            // Seed random number generator for reproducibility

  assert(NKEYS % nthread == 0); // Ensure keys can be divided evenly by
threads

  for (int i = 0; i < NKEYS; i++) {

    keys[i] = random();  // Fill keys array with random data

  }



  for(int i = 0; i < NBUCKET; i++){

    pthread_mutex_init(&locks[i], NULL); // Initialize one mutex for each
bucket

  }



  // PHASE 1: Concurrent Insertions (PUTS)

  t0 = now();            // Start clock

  for(int i = 0; i < nthread; i++) {

    // Create threads to execute put_thread

    assert(pthread_create(&tha[i], NULL, put_thread, (void *) (long) i) == 0);

  }
```

```
  for(int i = 0; i < nthread; i++) {

    assert(pthread_join(tha[i], &value) == 0); // Wait for all threads to
finish

  }

  t1 = now();            // End clock

  printf("%d puts, %.3f seconds, %.0f puts/second\n",

          NKEYS, t1 - t0, NKEYS / (t1 - t0));



  // PHASE 2: Concurrent Lookups (GETS)

  t0 = now();            // Reset clock

  for(int i = 0; i < nthread; i++) {

    // Create threads to execute get_thread

    assert(pthread_create(&tha[i], NULL, get_thread, (void *) (long) i) == 0);

  }

  for(int i = 0; i < nthread; i++) {

    assert(pthread_join(tha[i], &value) == 0); // Wait for all threads to
finish

  }

  t1 = now();            // End clock

  printf("%d gets, %.3f seconds, %.0f gets/second\n",

          NKEYS*nthread, t1 - t0, (NKEYS*nthread) / (t1 - t0));



  for(int i = 0; i < NBUCKET; i++){

    pthread_mutex_destroy(&locks[i]); // Clean up/destroy the mutexes
```

```
  }



  return 0;                 // Exit successfully

}
```