ICT1012

OPERATING SYSTEMS

LABORATORY MANUAL

Lab2-w5: Lab Assignments

AY 2025/26

BACHELOR OF ENGINEERING

IN

INFORMATION AND COMMUNICATIONS TECHNOLOGY

Contents

Lab Assignment

Prerequisite

- Download "**lab.zip**" from "xSiTe" folder "**Labs/xv6labs-w5: Thread**" to "**c:/ICT1012/xv6labs-w5**".
- Unzip and keep all folders and files directly under "**c:/ICT1012/xv6labs-w5**" without the "labs" sub-folder.

Submission deadline

You must submit your result to Gradescope before 20 February 2026 11:59pm

Assignment Task 1: uthread

1. This task is to familiarize with the design and implementation of a context switching mechanism for a user-level threading system.
2. To get started, "*xv6*" is provided with two files "*user/uthread.c*" and "*user/uthread_switch.S*", and a rule in the "*Makefile*" to build a "*uthread*" program. It is not required to add *"$U/_uthread*" under "*UPROGS*".
3. "*uthread.c*" contains most of a user-level threading package, and code for three simple test threads.
4. The threading package is missing some of the code to create a thread and to switch between threads.
   List of functions in "*uthread.c*" is provided below. Incomplete functions are highlighted in yellow.

Function	Description
int main(int argc, char *argv[])	The entry point that initializes the threading system, creates user threads, and starts the scheduler.
void thread_init(void)	Sets up the initial threading environment, establishing the current execution context as the first "main" thread.
void thread_create(void (*func)())	Allocates a new thread, assigns a private stack, and sets the initial return address (ra) to the provided function.
void thread_a(void)	A user-defined function representing the specific workload or execution loop for Thread A.
void thread_b(void)	A user-defined function representing the specific workload or execution loop for Thread B.
void thread_c(void)	A user-defined function representing the specific workload or execution loop for Thread C.
void thread_yield(void)	Voluntarily pauses the current thread and triggers the scheduler to give another runnable thread a chance.

void thread_schedule(void)	The core logic that searches the thread table for a RUNNABLE thread and executes the context switch.

The assembly language source code file "*uthread_switch.S*" contains incomplete function "*thread_switch*" that needs to be addressed.

5. The task is to come up with a plan to create threads and save/restore registers to switch between threads, and implement that plan.

Data structures:

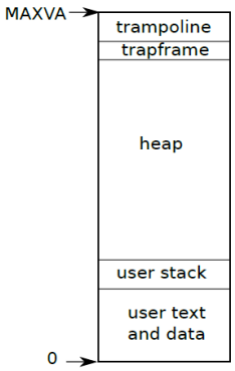
1. Create a structure "***thread_context***" to store thread context.
   Explanation for the registers used in the *"thread_context"* and the "*uthread_switch.S*" assembly.

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.

   These are the registers that must be saved and restored during a context switch to ensure a thread resumes exactly where it left off.
   The Stack Pointer (sp) is the most vital: it ensures that when the thread wakes up, it is looking at its own memory, not the stack of the thread that just ran.
   The Return Address (ra) ensures that when the ret instruction is called at the end of the switch, the CPU jumps back into the user thread function (like *thread_a*) rather than crashing.
2. Add the "***thread_context***" structure to "*struct thread*".

Thread function: thread_create

1. "*thread_create*" needs to set stack pointer address (*thread_context* structure) to point to the thread's own stack.



The stack grows from higher address to lower. Therefore, the stack pointer at the beginning must point to the end of the stack structure.

2. It also needs to set the return address to the function the thread should run. This is the pointer to the thread function that will be executed by the thread.

Thread function: thread_schedule

Add a call to "***thread_switch***" in "*thread_schedule*"; pass arguments needed to invoke a switch from thread "*t*" to "*next_thread*".

Thread switch function: thread_switch in uthread_switch.s

The goal is to ensure that "*thread_switch*" saves the **callee-saved** registers of the thread being switched away from, restores the **callee-saved** registers of the thread being switched to, and returns to the point in the latter thread's instructions where it last left off.

Refer to: "***kernel/swtch.S***" and xv6 manual for callee-saved registers.

Explanation for some of the assembly codes used:

In xv6 (specifically the RISC-V version), ld and sd are fundamental assembly instructions used to move 64-bit data between CPU registers and memory.

1. sd (Store Doubleword)

   The sd instruction takes a 64-bit value from a register and stores it into memory.

   ```
   sd ra, 0(a0)  /* Take the value in RA and write it to address a0 + 0 bytes */
   sd sp, 8(a0)  /* Take the value in SP and write it to address a0 + 8 bytes */
   ...
   ```

   - The code takes the values currently sitting in the physical CPU registers and copies them into the struct thread_context in RAM.
   - The offsets (0, 8, 16...) match the 8-byte size of uint64 variables in the C struct.
2. ld (Load Doubleword)

   The ld instruction reads a 64-bit value from memory and loads it into a register.

   ```
   ld ra, 0(a1)  /* Read the value from address a1 + 0 and put it into register RA */
   ld sp, 8(a1)  /* Read the value from address a1 + 8 and put it into register SP */
   ...
   ```

   - The code ignores the old thread. It reaches into the thread_context of the next thread (pointed to by a1) and pulls those values back into the physical CPU registers.
   - Once ld sp, 8(a1) executes, the CPU is now officially using the new thread's stack. Any local variables accessed after this point belong to the new thread.
3. The jump happen with 'ret'

   ```
   ret   /* Return to the address currently in 'ra' */
   ```

   - Remember that we just loaded a new value into ra from the next thread's context.
   - When ret executes, the CPU doesn't go back to the code that called "*thread_switch*". Instead, it jumps to whatever address was saved in the new thread's ra.

- If this is a brand-new thread, it jumps to the start of "*thread_a*", *thread_b*, or "*thread_c*".

Example

1. The following example illustrates "*uthread*" behavior:

```
:/mnt/c/ICT1012/xv6labs-w5 $ make clean
:/mnt/c/ICT1012/xv6labs-w5 $ make qemu
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
…
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$
```

2. Exit xv6 shell with keys "**Ctrl + a**" followed by "**x**".
3. Run all test cases for "*uthread*".

```
:/mnt/c/ICT1012/xv6labs-w5$ ./grade-lab-thread uthread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (2.4s)
:/mnt/c/ICT1012//xv6labs-w5$
```

Assignment Task 2: Multi-Threaded Hash Table & Synchronization using 'pthreads'

Task introduction

1. While the "*uthread*" portion (Assignment Task 1) uses **xv6** to learn low-level context switching, the hash table task is performed on a **Unix-like host (Linux/macOS)** using the "***pthreads*"** library.

   Environmental Context: Why Unix instead of xv6?

   - **Multicore Hardware:** xv6 is a simplified kernel often running on a single virtual CPU in educational settings. Real-world Unix environments allow threads to run on multiple physical CPU cores simultaneously, which is necessary to observe true race conditions and performance scaling.
   - **Standard Tooling:** Unix provides mature debugging and performance monitoring tools (like "*gprof*" or time) and the standard POSIX threads API ("**pthreads"**), which is the industry standard for parallel programming.

   This assignment is to explore parallel programming with threads and locks using a hash table. This assignment is to be performed on a real Linux or MacOS computer (**not xv6, not qemu**) that has multiple cores. Most recent laptops have multicore processors.

   Reference to UNIX "*pthread*" threading library:

   https://pubs.opengroup.org/cgi/kman.cgi

2. Hash table overview

   This lab explores the challenges of **concurrency** through a shared hash table (key, value). The program needs to manage a hash table with **5 buckets** (NBUCKET) and **100,000 keys** (NKEYS) using linked lists for collision handling.

   The Architecture: Hash Table & Buckets

   - **The Rule:** Any item (Key) you want to store always goes into a specific bucket based on its ID: bucket = Key % 5.
   - **Collisions:** Since we have 100,000 keys but only 5 buckets, each bucket contains a long chain (Linked List) of items.
   - **The Goal:** The workload consists of two distinct phases.
     - **100,000 Puts:** The total number of keys (100,000) is split equally among the threads. If 2 threads are used, each thread performs 50,000 insertions.
     - **200,000 Gets:** After insertions are complete, **every** thread iterates through the **entire** set of 100,000 keys to verify they exist. With 2 threads, this results in a total of 200,000 lookups (100,000 per thread).

Multi-Threaded Hash Table without Synchronization

- The file "***notxv6/ph-without-lock.c***" contains a simple hash table that is correct if used from a single thread, but incorrect when used from multiple threads.

  From the main xv6 directory (e.g., *"~/xv6labs-w5"*), type this:

```
ICT1012/xv6labs-w3$ cd notxv6/
ICT1012/xv6labs-w3/notxv6$ make ph-without-locks
cc    ph-without-locks.c  -o ph-without-locks
/ICT1012/xv6labs-w3/notxv6$ ./ph-without-locks 1
```

- Note that to build "*ph-without-locks*", the "*Makefile*" uses Ubuntu (UNIX) OS's "*gcc*", not the "*xv6*" tools.
- The argument to "*ph-without-locks*" specifies the number of threads that execute "*put*" and "get" operations on the hash table. After running for a little while, "*ph-without-locks 1*" will produce output similar to this:

```
100000 puts, 4.818 seconds, 20753 puts/second
0: 0 keys missing
100000 gets, 4.897 seconds, 20419 gets/second
```

  The numbers may differ from this sample output by a factor of two or more, depending on how fast your computer is, whether it has multiple cores, and whether it's busy doing other things.

- "*ph-without-locks*" runs two benchmarks. First it adds lots of keys to the hash table by calling "*put()*", and prints the achieved rate in puts per second. Then it fetches keys from the hash table with "*get()*". It prints the number keys that should have been in the hash table as a result of the "*puts*" but are missing (zero in this case), and it prints the number of "*gets per second*" it achieved.

- "*ph-without-locks*" can use its hash table from multiple threads at the same time by giving it an argument greater than one. With 2 threads:

```
ICT1012/xv6labs-w3/notxv6$ ./ph-without-locks 2
100000 puts, 2.426 seconds, 41215 puts/second
0: 613 keys missing
1: 613 keys missing
200000 gets, 4.524 seconds, 44213 gets/second
ICT1012/xv6labs-w3/notxv6$
```

  The first line of this "*ph-without-locks* 2" output indicates that when two threads concurrently add entries to the hash table, they achieve a total rate of 41,215 inserts per second. That's about twice the rate of the single thread from running "*ph-without-locks* 1". That's an excellent "parallel speedup" of about 2x, as much as one could possibly hope for (i.e. twice as many cores yielding twice as much work per unit time).

However, the two lines saying 1226 keys missing indicate that a large number of keys that should have been in the hash table are not there. That is, the "puts" were supposed to add those keys to the hash table, but something went wrong.

- Refer to the code in "*notxv6/ph-withoutlocks.c*", particularly at "*put()*" and "*insert()*" to reason out what happened.

  When multiple threads call "*put()*" simultaneously, they often attempt to update the same linked list at once. Without synchronization, one thread can overwrite the next pointer of another, causing entries (keys) to be "dropped" from memory.

- The data loss happened because there was no synchronisation between the threads. Next, we will add **mutex** (**mut**ually **ex**clusive) locks to synchronise thread tasks.

Multi-Threaded Hash Table with Synchronization

- A **dummy** file "*ph-with-mutex-locks.c*" is provided to avoid compile error. Copy the code in "*ph-without-locks.c*" to "*ph-with-mutex-locks.c*". Keep the file "*ph-without-locks.c*" as it is. To accomplish this task, modify the code in "*ph-with-mutex-locks.c*".
- To avoid missing keys, insert lock and unlock statements in function "*put*" in "*ph-with-mutex-locks.c*", so that the number of keys missing is always 0 with two threads.

  The relevant "*pthread*" calls are:

```
pthread_mutex_t lock;              // declare a lock
pthread_mutex_init(&lock, NULL); // initialize the lock
pthread_mutex_lock(&lock);         // acquire lock
pthread_mutex_unlock(&lock);       // release lock
```

  Reference:

  "pthread_mutex_init":
  https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_mutex_init.html

  "pthread_mutex_lock" and "pthread_mutex_unlock":
  https://pubs.opengroup.org/onlinepubs/007908799/xsh/pthread_mutex_lock.html

  Note:

  For this lab task, "*get*" does not require lock. First reason is that "get" (second phase) is performed after all the "*put*" (first phase) is done. Second reason is that "get" will read the key and never writes, that is it doesn't change the data, so that there is no chance of missing keys when performing "*get*".

  - Experiment "*ph-with-mutex-locks*" with 1 and then 2 threads.
  - There are situations where concurrent "*put()*" have no overlap in the memory they read or write in the hash table, and thus don't need a lock to protect against each other.

TASK: Multi-Threaded Hash Table with Synchronization Per-Bucket

- To obtain parallel speedup **implement your solution to have a lock per hash bucket**. In "**Per-Bucket Locking**", a separate mutex (lock) is assigned to each of the 5 buckets.

Threads only block each other if they hit the same bucket, allowing for parallel insertions in different buckets while preventing data loss.
The behaviour of "*ph-with-mutex-locks*" implemented with "Per-Bucket Locking" is:

```
ICT1012/xv6labs-w5/notxv6$ ./ph-with-mutex-locks 1
100000 puts, 4.397 seconds, 22742 puts/second
0: 0 keys missing
100000 gets, 5.101 seconds, 19606 gets/second
ICT1012/xv6labs-w5/notxv6$ ./ph-with-mutex-locks 2
100000 puts, 3.001 seconds, 33318 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 4.799 seconds, 41678 gets/second
```

Task

- Modify the code to declare, create, lock, unlock, and destroy locks to control the access to the buckets.
    1. Declare the mutex for each bucket to be accessible to all the threads in a structure that allows you to lock only the bucket being accessed at a given time.
    2. ***Create the locks for all the buckets*** in a place that allow all the threads to access them.
        a. You are working with threads, where is the most reasonable point to create locks that will be used by all threads?
    3. Modify function put to lock and unlock the mutex controlling the bucket being accessed.
    4. ***Destroy the locks for all the buckets*** in a place that allow all the threads to access them.
        a. Like in the initialization, where is the most reasonable point to destroy locks that have been used by all threads
- Use the grading script to evaluate your results (see Evaluaton below)
- Upload it to gradescope (see Assignment Submission below)

Comparative Analysis of Results

Configuration	Threads	Missing Keys	Throughput (Puts/Sec)	Performance Note
No Lock	1	0	~19,738	Correct but slow (Sequential).
No Lock	2	**1,442 total**	**~42,341**	Fastest, but **corrupted** data.
With Lock	1	0	~22,742	Correct; negligible lock overhead.
With Lock	2	**0**	**~33,318**	**Correct & Fast** (1.46x speedup).

Note: Throughput and keys missing may vary.

Analysis: *The "No Lock" 2-thread version is the fastest because it ignores safety. However, the "With Lock" version achieves a significant speedup (exceeding the 1.25x goal) while ensuring 0 missing keys. The identical missing key count in both threads (721 each) confirms that once a key is lost during insertion, it is gone for everyone.*

Key Learning Takeaways

- **Race Conditions:** These are non-deterministic bugs where the output depends on the timing of thread execution.
- **Lock Granularity:** Using one lock for the whole table would be safe but slow. Using one lock per bucket (granular locking) allows threads to work in parallel, maximizing hardware utilization.
- **Coordination:** Synchronization is not just about protection (mutexes) but also about waiting for rounds to complete (barriers), ensuring threads stay in sync during complex multi-step tasks.

Evaluation

```
ICT1012/Refer/xv6labs-w5$ ./grade-lab-thread ph
== Test ph_no_locks_single == gcc -o ph-without-locks -g -O2 -DSOL_THREAD -DLAB_THREAD
notxv6/ph-without-locks.c -pthread
ph_no_locks_single: OK (10.6s)
== Test ph_no_locks_multi == make: 'ph-without-locks' is up to date.
Test passed: Detected 32940 missing keys due to race conditions.
ph_no_locks_multi: OK (7.4s)
== Test ph_safe == gcc -o ph-with-mutex-locks -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph-
with-mutex-locks.c -pthread
ph_safe: OK (7.9s)
== Test ph_fast == make: 'ph-with-mutex-locks' is up to date.
ph_fast: OK (17.8s)
```

Assignment Submission

1. Execute "**make grade**" to test "uthread", "ph-without-locks" and "ph-with-mutexlocks" tasks.

```
ICT1012//xv6labs-w3$ make grade
== Test uthread ==
$ make qemu-gdb
uthread: OK (8.5s)
== Test ph_no_locks_single == make[1]: Entering directory 'ICT1012/xv6labs-w5'
gcc -o ph-without-locks -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph-without-locks.c -
pthread
make[1]: Leaving directory 'ICT1012//xv6labs-w5'
ph_no_locks_single: OK (10.3s)
== Test ph_no_locks_multi == make[1]: Entering directory '/ICT1012/ /xv6labs-w5'
make[1]: 'ph-without-locks' is up to date.
make[1]: Leaving directory '/ICT1012/Refer/xv6labs-w3'
Test passed: Detected 33784 missing keys due to race conditions.
ph_no_locks_multi: OK (7.5s)
== Test ph_safe == make[1]: Entering directory 'ICT1012/xv6labs-w3'
gcc -o ph-with-mutex-locks -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph-with-mutex-
```

```
locks.c -pthread
make[1]: Leaving directory '/ICT1012 /xv6labs-w5
ph_safe: OK (7.9s)
== Test ph_fast == make[1]: Entering directory '/ICT1012//xv6labs-w3'
make[1]: 'ph-with-mutex-locks' is up to date.
make[1]: Leaving directory '/mnt/c/ICT1012/Refer/xv6labs-2023-thread_instructor'
ph_fast: OK (17.5s)
Score: 60/60
ICT1012/xv6labs-w5$
```

2. Execute "**make zipball**" to create "**lab.zip**".

3. Upload the "**lab.zip**" file to Gradescope assignment "**xv6labs-w5: Thread**" to get auto graded.