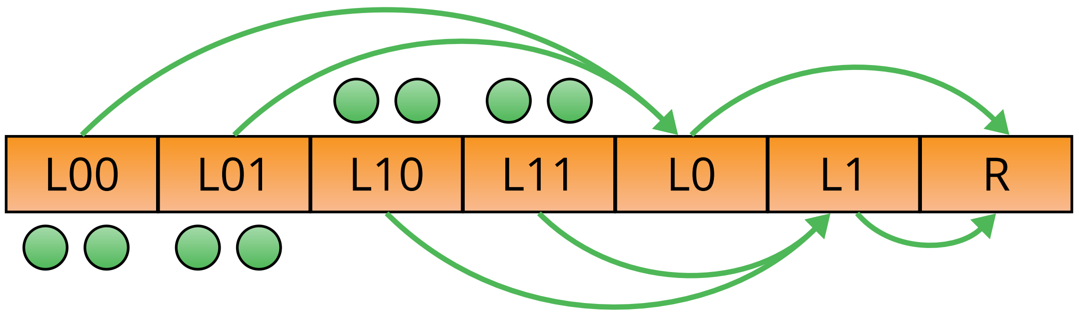
Operating Systems 202.1.3031

Spring 2025 Assignment 2

Synchronization and Processes



Ben-Gurion University of the Negev

Contents

1 Introduction

Welcome back to another exciting chapter in the world of operating systems! We hope that by now you are more familiar with xv6 and ready to delve deeper into the OS. In this assignment, you are going to implement the Peterson lock and tournament tree synchronization mechanisms in xv6. By developing this new system, you will learn about process synchronization as well as the sleep and wakeup mechanisms provided by the xv6 kernel.

An operating system, even a simple one like xv6, is a complex piece of software. Such low-level code is often challenging at first. This stuff takes time, but it can be fun and rewarding! We hope this class will serve to bring together a lot of what you've learned in your degree program here at BGU. **Take a deep breath and be patient with yourself.** When things don't work, keep calm and start debugging!

Good luck and have fun!

2 Submission Instructions

- Make sure your code compiles without errors and warnings and runs properly!
- We recommend that you comment your code and explain your choices, if needed. This would also be helpful for the discussion with the graders.
- Submit a **single copy** of xv6 containing all of the changes you made, in a single `.tar.gz` or `.zip` file.
- We advise you to work with git and commit your changes regularly. This will make it easier for you to track your progress, collaborate and maintain a working version of the code to compare to when things go wrong.
- Theoretical questions in this assignment are **not for submission**, but you are advised to understand the answers to these questions for the grading sessions.
- Submission is allowed **only in pairs** and **only via Moodle**. Email submissions will not be accepted.
- Before submitting, run the following command in your xv6 directory:

  ```
  $ make clean
  ```

  This will remove all compiled files as well as the `obj` directory.
- Help with the assignment and git will be given during office hours. Please email us in advance.

3 Task 1: Peterson Locks for Userspace

In this task, you will implement a new synchronization mechanism in xv6 using the Peterson lock we discussed in class. Our goal is to enforce mutual exclusion between two processes to protect a critical section. In a system with process memory isolation, such as xv6, any synchronization mechanism cannot be implemented without the help of the kernel, as processes cannot access each other's memory.

The only mechanisms xv6 provides for synchronization are the spinlock and the sleeplock. The spinlock is a kernel-mode mechanism that blocks an entire core by keeping it spinning until the lock clears (hence the name), and additionally requires interrupts to be disabled for the duration of the critical section. For these reasons, spinlocks can't be held for a long time, as they prevent other processes from running on the core. So we can't use spinlocks to protect a critical section in userspace.

The sleeplock, on the other hand, is a mechanism that allows a process to sleep while waiting for a lock to be released, allowing other processes to run on the core in the meantime. It requires some method to decide which process should be granted access to the critical section and which process should wait. For our lock, we do not want the process to busy-wait or sleep, but we do want to free up the CPU to run other processes and retry the lock acquisition later.

Unlike the algorithm discussed in class, this variant differs in two key ways:

1. Instead of busy-waiting, the kernel will yield the CPU to another process when the lock is held by the other process. Use the `yield()` function to give up the CPU. This conserves CPU resources and allows other processes to run on the core that would otherwise be spinning.
2. While theoretically Peterson does not require any atomic operations, in practice, modern compilers and CPU hardware can reorder instructions and optimize code in ways that can break the algorithm. In addition, memory caching can cause stale information to be read by a core, leading to incorrect locking behavior. We will use atomic operations and memory barriers to guarantee that the lock is held by only one process at a time, similar to the spinlock implementation.

How do we implement such a mechanism in xv6? We'll begin by describing the API for our Peterson lock. We will need four system calls:

1. `int peterson_create(void);`

   Creates a new Peterson lock and returns a unique identifier for the lock. In case the lock cannot be created, returns -1.

2. `int peterson_acquire(int lock_id, int role);`

   Takes a lock identifier and the role of the process (0 or 1) and attempts to acquire the lock. If the lock is already held by another process, the calling process should yield the CPU and try again later. In either case, when the function returns, it is guaranteed that no other process holds the lock and the calling process has acquired it. Returns 0 on success and -1 on error (e.g., the lock identifier or the role is invalid).

3. `int peterson_release(int lock_id, int role);`

   Takes a lock identifier and the role of the process (0 or 1) and releases the lock. The other role may not acquire the lock until it is released by this call. Returns 0 on success and -1 on error (e.g., the lock identifier or the role is invalid).

4. `int peterson_destroy(int lock_id);`

   Deletes the lock with the given identifier. Once this function returns, the lock identifier is invalid and neither role can acquire the lock. Either role can call this function. Returns 0 on success and -1 on error (e.g., the lock identifier is invalid).

Consider the following example usage of the Peterson lock:

```
int lock_id = peterson_create();
if (lock_id < 0) {
    printf("Failed to create lock\n");
    exit(1);
}

int fork_ret = fork();
int role = fork_ret > 0 ? 0 : 1;

for (int i = 0; i < 100; i++) {
    if (peterson_acquire(lock_id, role) < 0) {
        printf("Failed to acquire lock\n");
        exit(1);
    }

    // Critical section
    if (role == 0)
        printf("Parent process in critical section\n");
    else
        printf("Child process in critical section\n");

    if (peterson_release(lock_id, role) < 0) {
        printf("Failed to release lock\n");
        exit(1);
    }
}

if (fork_ret > 0) {
    wait(0);
    printf("Parent process destroying lock\n");
    if (peterson_destroy(lock_id) < 0) {
        printf("Failed to destroy lock\n");
        exit(1);
    }
}

exit(0);
```

The data structure for the Peterson lock will be maintained in the kernel. Design a structure that supports the fields required for the Peterson lock and the guarantees of the API, namely to keep track of the state of the lock and whether it is active (created) or not. Take inspiration from the `sleeplock` struct, and the implementation of the spinlock in the kernel.

A more sophisticated system would probably want to prevent a process that has exited or been killed from holding the lock indefinitely, or delete the lock when the process that created it exits, but for simplicity, we will ignore these cases in this assignment.

Next, create a global array of at least 15 Peterson locks (we will need 15 locks for the next task) and write code to initialize the array when the kernel starts. See the initialization of the `proc` array for an example.

Implement the four system calls as described by the API. Use the functions `__sync_lock_test_and_set`, `__sync_lock_release`, and `__sync_synchronize` to enforce atomicity and memory ordering where appropriate. These functions may be needed because the compiler and CPU hardware may reorder instructions to optimize the code. Furthermore, the kernel may run on multiple cores, and the memory cache may cause stale information to be read by a core, or new information might not propagate to other cores when we expect it to:

- `__sync_synchronize` is used to ensure all memory operations before the call are completed before any memory operations after the call are started and that the memory accessed is synchronized between cores. This does NOT mean that `__sync_synchronize` is a mutual exclusion mechanism, but rather deals only with the memory system. It should be used before reading state that is expected to be shared between cores, and after writing shared state.
- `__sync_lock_test_and_set` is an atomic exchange operation that sets a new value and returns the old value. It can be used to atomically set a value and check the old value in a single operation. However, this does not guarantee that other cores will see the new value immediately. To enforce that, we need to use `__sync_synchronize` before reading the value and after writing it.
- `__sync_lock_release` is used to set a value to 0 atomically. Other cores may not see the new value immediately, like `__sync_lock_test_and_set`.

Note that there are multiple ways to use the above synchronization primitives to implement the Peterson lock. **Any implementation that satisfies the requirements described in this assignment is acceptable.** You do not have to use all three. See the implementation of the spinlock for an example of how to use these functions, as well as its discussion in class.

Finally, test your implementation with the example code provided above.

In the next task, we will address the more general case of synchronization for multiple processes.

Task 1 — implement Peterson's lock

1. Add the Peterson lock data structure to the kernel and initialize an array of at least 15 locks.
2. Implement the four system calls:
   (a) `int peterson_create(void);`
   (b) `int peterson_acquire(int lock_id, int role);`
   (c) `int peterson_release(int lock_id, int role);`
   (d) `int peterson_destroy(int lock_id);`

   as described above.
3. Test your implementation with the example code provided above.

For submission: Changes to the xv6 kernel to implement the lock mechanism. You do not have to submit the test code.

Questions — not for submission

1. What is the purpose of the Peterson lock? How does it work? Why does it need to set the `turn` variable to the other process's role?
2. How do we keep track of the state of the Peterson lock? Can't we just use the spinlock to protect the critical section?
3. Why do we need a global array of locks? Can't we just create a lock when we need it?
4. How do we enforce that the lock is only held by one process at a time? How do we ensure the atomic modification of the lock state?
5. What does the `__sync_lock_test_and_set` function do? Why do we use it for?
6. What does the `__sync_synchronize` function do? Why do we use it for?
7. What are the fields required in the data structure for the Peterson lock? Which fields are related to the Peterson algorithm and which are related to our specific implementation?
8. What is the difference between a spinlock and a sleeplock?
9. How does `yield` work? How does it differ from `sleep`?
10. Does `yield` necessarily hand the CPU to the other process participating in the Peterson algorithm? Does it necessarily hand the CPU to another process at all?
11. What does `sched` do? How does it work?
12. How is the system call `sleep` different from the function `sleep`? How does the kernel know when the specified time has passed to wake up the process?
13. How can we make sure that calls on a lock that has been destroyed return -1 even if they were initiated before the call to `peterson_destroy`?

4 Task 2: Tournament Tree

In this task, you will implement a more complex synchronization mechanism using the Peterson lock from Task 1 as a building block. To generalize the Peterson lock to more than two processes, we will use a tournament tree as described in class. Unlike the previous task, using Peterson locks to construct a tournament tree can be done entirely in userspace.

The tournament tree is a binary tree where each node is a Peterson lock. Our system will only allow one process to hold the lock at the root of the tree at any given time. Furthermore, since only two processes can use one Peterson lock to negotiate access to a critical section, our system will assign locks to each process and a permanent role to each process in each tree level.

Unlike the theoretical discussion in class, which did not address any practical implementation details, to take the theory and turn it into a working system, we will need to address the following issues:

1. How should we represent the tournament tree in memory? We will use a binary tree where each node is a Peterson lock. The leaves of the tree will be the processes that are participating in the tournament and the internal nodes will be the locks that are used to synchronize access to the critical section.
2. How do we assign locks to processes?
3. How should the functionality of the tournament tree be exposed to the user? We will describe the API for the tournament tree in the next section.
4. Should we handle edge cases arising from the real-life representation of processes, such as processes that exit or are killed while holding a lock?

This is an important exercise in bridging the gap between theory and practice.

We will implement this system as a userspace library that provides the following functions:

1. `int tournament_create(int processes);`

Creates a new tournament tree with the given number of processes. This function accepts the number of processes that will participate in the tournament, which must be a power of 2 up to 16. It will create all locks needed for the tree, fork the processes, and assign each process a lock and a role in each tree level. Returns the tournament ID assigned to the process on success and -1 on error, such as an invalid number of processes or if the tree cannot be created. If a fork operation fails, the function should return -1, but does not attempt to clean up the tree.

2. `int tournament_acquire(void);`

   Attempts to complete the tournament to acquire the lock at the root of the tree for the calling process. This function uses the lock and role assigned to the calling process by `tournament_create`. Returns 0 on success and -1 on error.

3. `int tournament_release(void);`

   Releases all locks held by the calling process in the reverse order of acquisition. Returns 0 on success and -1 on error.

These functions should be implemented in a userspace library. Implement the functions in a file `user/libtournament.c` and add it to the `ULIB` variable in the `Makefile`, so it will be compiled and linked with the user programs. Add the declarations of the functions to `user/user.h` so that user programs can use them.

In order to maintain the tournament tree, we need to keep track of the locks and roles assigned to each process. To accomplish this, assign a running index from 0 to $N-1$ to each forked process and save it in a global variable in the library C file. The role of each process at each level $l \in [0, L-1]$ of the tree, where the bottom level is $l = L-1$ and the number of levels is $L = \text{log2}(N)$, can be calculated from the index of the process by extracting the bit at the corresponding position,

$$\text{role}_l = (\text{index}\,\&\,(1 << (L-l-1))) >> (L-l-1),$$

and the lock index at each level can be calculated by,

$$\text{lock}_l = \text{index} >> (L-l).$$

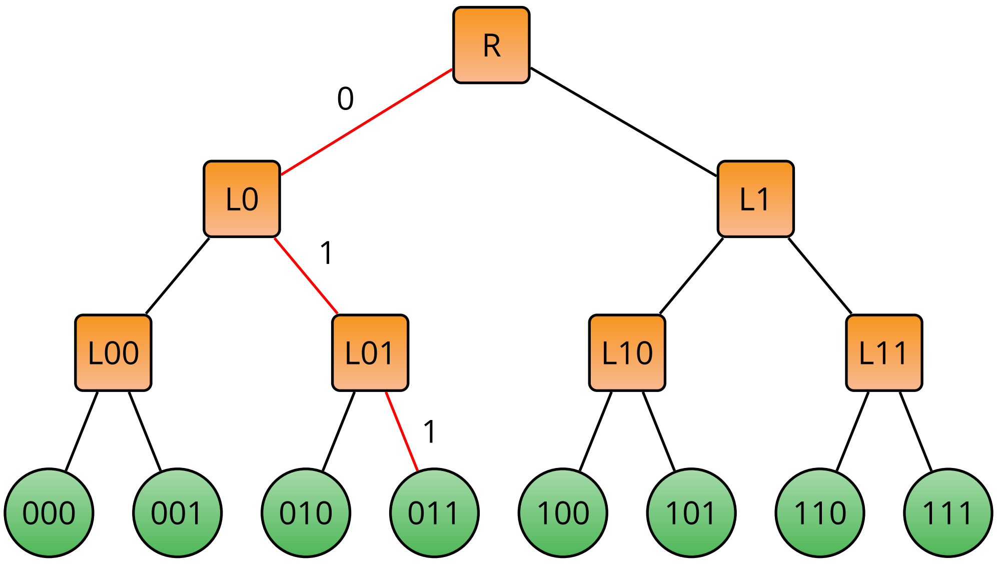
If all locks are stored in a BFS-order array (i.e., the lock for $l = 0$ are at the start of the array and so on), the lock at level l can be accessed by locks[i]

where

$$i = \text{lock}_l + \sum_{k=0}^{l-1} 2^k = \text{lock}_l + (1 << l) - 1.$$

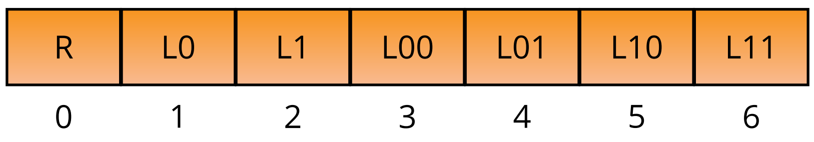
Finally, test your implementation by creating a userspace program called `tournament.c` that uses the library to create a tournament tree with N processes, where N is a power of 2 up to 16. Accept the number of processes as a command-line argument. Each process should acquire the lock at the root of the tree, print a message containing its PID and tournament ID, release the lock and exit. If the tournament system works correctly, only one process should be able to acquire the lock at the root of the tree at any given time and the prints should be synchronized. Note, however, that the order of the prints is not guaranteed, only that they will not overlap.

The tournament tree is shown below for an instance with 8 processes:



The binary indices of the processes are shown in the circles and the locks are shown in the rectangles with their binary indices within their level. The path from process 3 (011) to the root lock (R) is shown in red. The role bit of the process at each level is shown on the edge connecting the process to the lock. Note how the index of the lock at each level and the role is calculated from the index of the process.

The arrangement of the locks in the array is:



Verify that the correct array elements are hit by the calculations above for process 3 (011), including the root lock at index 0.

You may find it helpful to test that your system works using the above diagrams and calculations. However, for submission, the system must work with 16 processes and the locks must be created dynamically based on the number of processes. You may use `malloc` to allocate memory for the lock array. Store the array as a global variable within the library C file. You may assume that only one tournament tree will be created at a time and no cleanup of the tree is required to be implemented.

The representation described above and the correspnding formulas are not the only way to implement the tournament tree. **You may choose to represent the locks in a different way, as long as the API is respected and the implementation is correct.** If you choose a different representation, however, you may be asked to explain your choices and the correctness of your representation during the grading session.

Task 2 — implement the tournament tree

1. Implement the three functions:

   (a) `int tournament_create(int processes);`

   (b) `int tournament_acquire(void);`

   (c) `int tournament_release(void);`

   in the userspace library `user/libtournament.c`.

2. Add the library to the `ULIB` variable in the `Makefile` and add the function declarations to `user/user.h`.

3. Write a userspace program `tournament.c` that uses the library to create a tournament tree with 16 processes.

4. Test your implementation by running the program and verifying that the locks are acquired and released correctly.

For submission: The userspace library `user/libtournament.c` and the userspace program `tournament.c` and any other changes needed to the Makefile or other files.

Questions — not for submission

1. How does the tournament utilize the Peterson lock to guarantee that only one process can hold the lock at the root of the tree at any given time?
2. Why must the locks be released in the reverse order of acquisition?
3. How many locks are needed for a tournament tree with N processes? Why?
4. How many locks have to be taken by a process to acquire the lock at the root of the tree?
5. How do we set up the library so that all forked processes have access to the lock array?
6. Why do we need to create the locks dynamically?
7. Why did we wrap the tournament tree in a userspace library? Why not implement it directly in the userspace program?
8. How is the index assigned to each process? Where do we store it? How come each process can store its index in the same variable?
9. How do we calculate the lock and role for each process at each level of the tree?
10. How do we map a lock index to the correct element in the array?
11. Is the kernel aware of the tournament tree?
12. Are the locks actually stored in the array? If not, what is? Where are they? How do we access them?