HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 3
Synchronization
Course: Operating Systems

February 18, 2025

Goal This lab helps the student practice synchronization in the operating system, and figure out why we need the synchronization.

Contents In detail, this lab equips the students practicing experiments using synchronization techniques to solve the problem called **race condition**. The synchronization is performed on multi-thread contextualization using the following mechanisms:

- mutual exclusion (mutex)
- semaphore
- conditional variable

Besides, the practices also introduce and include some locking variants, i.e. spinlock, read/write spinlock, sequence lock.

Result After doing this lab, student can understand the definition of synchronization and implement a program without the race condition using the above techniques.

Prerequisites Students need to review the theory of synchronization.

Contents

1 Background

1.1 Race condition

Race condition is the condition of software where the systems' substain behaviors are dependent on the sequences of uncontrollable events. Therefore, we need mechanisms that provide synchronization ability.

1.1.1 Critical section problem

Consider the system includes n tasks, represented by threads or processes with shared memory, denoted by $\{P_0,P_1,...,P_{n-1}\}$

Each process has a critical section (i.e, segment of code) which has the following properties:

- data is **shared** among task
- the performed task **modifying** the shared data
- system employs the preemptive scheduling approach

We investigate the case of multi-thread processes having the shared data segment which is naively encountered the problem of race condition. The structure of multi-thread process as in Figure 1.

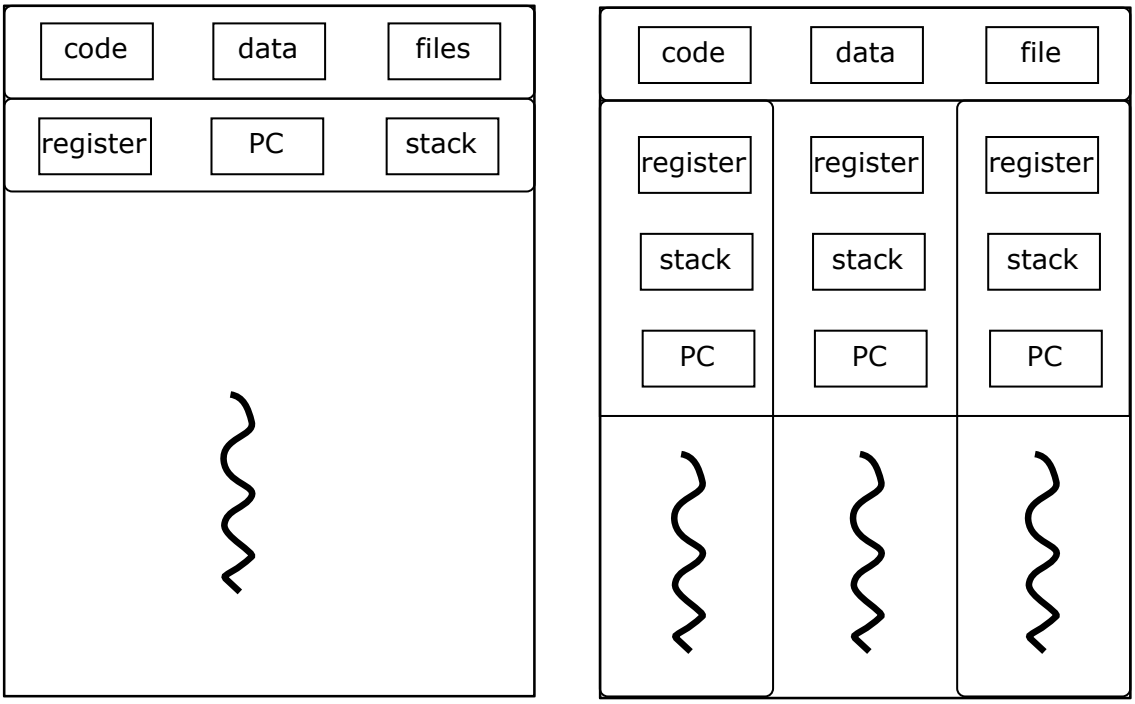

Figure 1: The structure of multi-thread process

Critical-section of the problem needs a mechanism to solve the raised problem of race condition. To propose the solution, we need first able to figure out, or identify, critical section, then we employs the synchronization mechanisms which are introduced/practiced in this work.

1.1.2 Race condition caused by atomicity

In high level programming language, we can note that a statement is always implemented in machine language (on a typical machine) by many instructions as the following example:

```
instruction1:= register load
instruction2:= arithmetic operation
instruction3:= register store
```

If there are two or more tasks (threads/or shared-memory processes) accessing a single storage, the data manipulation instructions which are split-able may result in an incorrect final state. Such tools are provided by POSIX pthread as follows:

Mutex Lock The problem with mutex is that the task is put into a **sleep** state and later is woken to process the task. This sleeping and waking action are expensive operations.

Spin lock In a short description, spin lock provides a **(CPU) poll** waiting until it has got privilege.

If the mutex sleeps in a very short time, then it wastes the cost of expensive operations of sleeping and waking up. But if the long time is quite long, CPU polling is a big waste of computation.

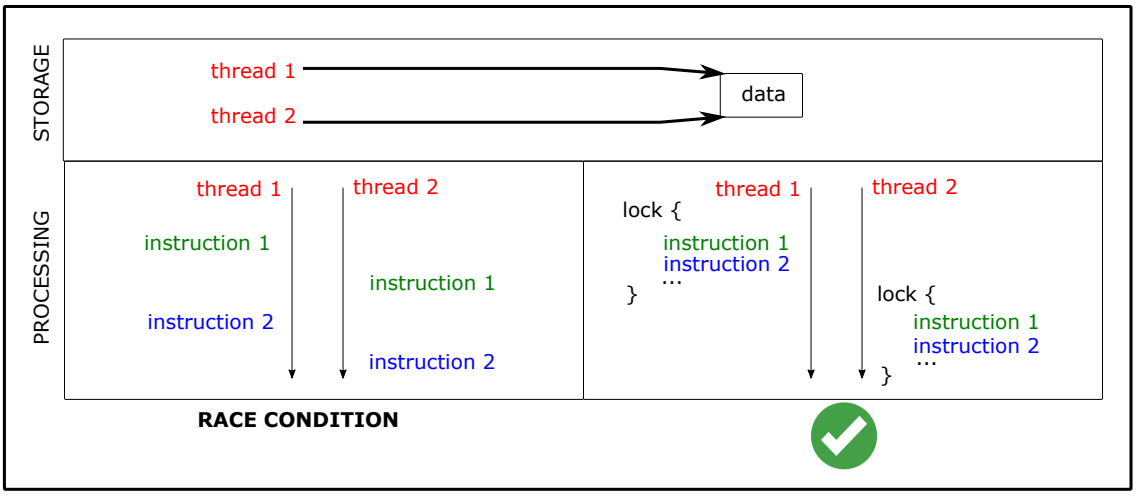

Figure 2: The race condition caused by atomicity

1.1.3 Race condition caused by in-correct ordering

In the previous chapter of process memory sharing, we introduced the bounded buffer problem. Even with mutex setting, the race conditional may still happen when a producer fills in a buffer an item before a consumer empties it causing an overwriting which results a loosing data. Another example of system behavior happens when a consumer retrieves a garbage (uninitialized) value if an item is retrieved before a producer fills in a meaningful value. These wrong system behaviors are caused by the in-correct ordering of the sequence of events.

Semaphore is suited cleanly to a producer-consumer model. A semaphore is basically an object that includes an internal counter, a waiting list of threads, and supports two different operations, i.e., wait and signal. The internal counter with a proper initialization provides a good mechamism to manage the limited number of storage slot in bounded buffer.

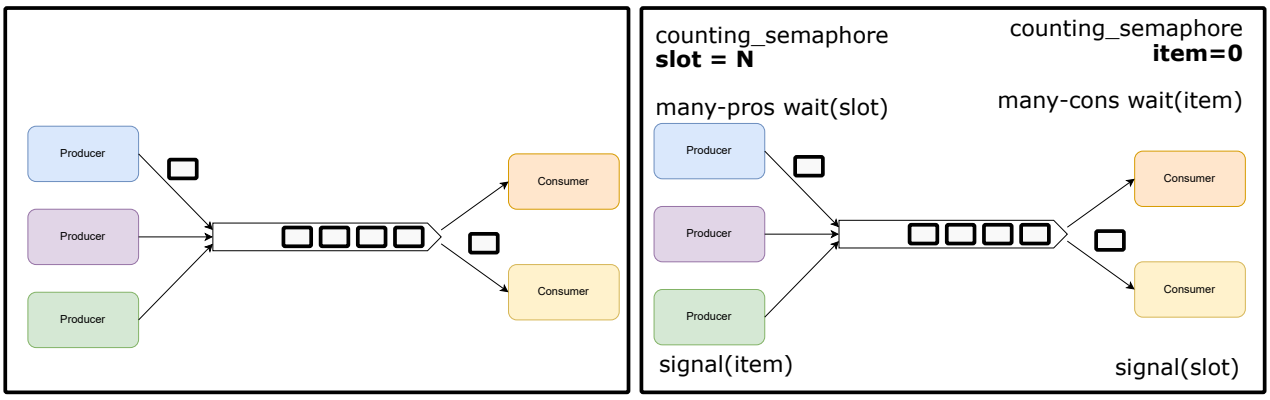

Figure 3: BoundedBuffer or Producer-Consumer problem

Although semaphore is perfect fit for bounded buffer problem, it constrains on the equal number of consumers(/readers) and producers(/writer). Some updated models such as few writers many readers is not a good application for semaphore. It relaxes the matching of internal counter with the number of slot then it helps in the unbalancing context between actors who access the buffer. These following tools provide the ordering mechanism with more relaxation on the number of system actors:

Conditional variable Conditional variable is a synchronization primitive that allows threads to wait until a particular condition occur. It just allows a thread to be signaled when something of interest to that thread occurs. It includes two operations: wait and signal. The conditional_variable can be used by a thread to block other threads until it notifies the conditional_variable.

Read-write spin lock and sequence lock In the previous section, we have seen some locking methods like mutex, spinlock, etc. In a high-speed manner, i.e., kernel driver, high-speed/fast communication, when you want to treat both the reader and the writer equally, then you have to use spin lock. The two following mechanisms provide a different priority policy between reader and writer. We introduce the main characteristics of them and then, we discuss the reader/writer conflict problem later as an exercise. (See more details in Section4)

Read-write spinlock: In some situations, we may have to give more access frequencies to the reader. The reader-writer spinlock is a suitable solution in this case.

Sequence lock: Reader-writer lock can cause writer starvation. Seqlock gives more permission to writer. Sequential lock is a reader-writer mechanism which is given high priority to the writer, so this avoids the writer starvation problem.

2 Programming Interfaces

POSIX Thread library provides a standard based thread API for C/C++. The functions and declarations of many common APIs in this library are conformed to POSIX.1 standards through many versions (2003, 2017 etc.). In Linux, the library is not integrated by default, so it requires an explicit linking declaration with **"-pthread"**. Semaphore is belong to POSIX (not pthread) standard. Sequence lock is not implemented in the both libraries but has been added to the kernel since Linux 2.5.x.

2.1 Mutex lock

provided in POSIX Thread (pthread) library

```
#include <pthread.h>
pthread_mutex_t lock;

int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
-----------------------------------------------------------------------
int pthread_mutex_lock(pthread_mutex_t *mutex );
int pthread_mutex_unlock(pthread_mutex_t *mutex ) ;
```

An example:

pthread_mutex_t lock;
pthread_mutex_init(&lock,NULL);
...
pthread_mutex_lock(&lock);
$< CS >$
pthread_mutex_unlock(&lock);
$< RS >$

2.2 Spin lock

provided in POSIX Thread (pthread) library

```
#include <pthread.h>
pthread_spinlock_t lock;
int pshare; /* PTHREAD_PROCESS_SHARED
             * or PTHREAD_PROCESS_PRIVATE (creator only)
             */

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);
-----------------------------------------------------------------------
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

An example:

pthread_spinlock_t lock;

//we can use pshared=0 for NULL setting or pshared=PTHREAD_PROCESS_SHARED
pthread_spin_init(&lock,PTHREAD_PROCESS_SHARED);
...
pthread_spin_lock(&lock);
$< CS >$
pthread_spin_unlock(&lock);
$< RS >$

2.3 Semaphore

provided in POSIX semaphore (not PTHREAD)

```
#include <semaphore.h>
sem_t sem;

int sem_init(sem_t *sem, int pshared, unsigned int value)
int sem_destroy(sem_t *sem);
-----------------------------------------------------------------------
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

An example:
sem_t sem;
//we can use pshared=0 for NULL setting or pshared=PTHREAD_PROCESS_SHARED
//The 3rd argument is the default value of the semaphore
sem_init(&sem,0,5);
...
sem_wait(&sem);
$< CS >$
sem_post(&sem);
$< RS >$

2.4 Conditional Variable

provided in POSIX Thread (pthread) library

```
#include <pthread.h>
pthread_cond_t cv_count;

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
-----------------------------------------------------------------------
int pthread_cond_wait (pthread_cond_t *cond , pthread_mutex_t *mutex ) ;
int pthread_cond_signal (pthread_cond_t *cond ) ;
```

An example:
pthread_mutex_t mtx;
pthread_cond_t lock;

pthread_mutex_init(&mtx,NULL);
pthread_cond_init(&lock,NULL);
...
pthread_cond_wait(&lock,&mtx); /* May be locked if no signal is triggered */
< *CS* >
pthread_cond_signal(&lock);
< *RS* >

2.5 Reader-writer lock

provided in POSIX Thread (pthread) library

```
#include <pthread.h>

pthread_rwlock_t lock;

int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
------------------------------------------------------------------------
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

An example:

pthread_rwlock_init(&lock,NULL);
...
pthread_rwlock_rdlock(&lock);
< *CS* >
pthread_rwlock_unlock(&lock);
< *RS* >
...
pthread_rwlock_rdlock(&lock);
< *CS* >
pthread_rwlock_wrunlock(&lock);
< *RS* >

2.6 Sequence lock

has not provided in POSIX Thread library yet. Its implementation has existed in kernel (and hence, can be used only in kernel space) since 2.5.60.

```
is N/A Not available, implement as a flavor (exercise) as an API in userspace
```

Although it lacks a user-space implementation, it is widely used in the kernel to protect buffer data in modern programming patterns with many readers, many writers, i.e. SMP Linux kernel support.

3 Practices

In this section, we work on various "native" problems to recognize the "real" wrong behaviors. All these experiments are derived from theory slides with a minor modification. We practice the provided synchronization mechanisms and see how they work to correct the wrong things.

3.1 Shared buffer problem

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

int MAX_COUNT = 1e9;
static int count = 0;

void *f_count(void *sid) {
  int i;
  for (i = 0; i < MAX_COUNT; i++) {
    count = count + 1;
  }

  printf("Thread %s: holding %d \n", (char *) sid, count);
}

int main() {
  pthread_t thread1, thread2;

  /* Create independent threads each of which will execute function */
  pthread_create( &thread1, NULL, &f_count, "1");
  pthread_create( &thread2, NULL, &f_count, "2");


  // Wait for thread th1 finish
  pthread_join( thread1, NULL);

  // Wait for thread th1 finish
  pthread_join( thread2, NULL);

  return 0;
}
```

Step 3.1.1 Compile and execute the program

```
# The program name must match your own code,
# copycat may result error gcc: fatal error: no input files
$ gcc -pthread  -o shrdmem shrdmem.c
$ ./shrdmem
Thread 1: holding 1990079976
Thread 2: holding 1991664743
```

Step 3.1.2 Recognize the issue and propose a fix mechanism using the provided synchronization tool.
Hint: pthread_mutex_lock() and pthread_mutex_unlock() are useful to protect f_count() thread worker

```
# Implement by your self the fixed shrdmem_mutex program
# (it is not available yet)
# copycat may result error gcc: fatal error: no input files
$ gcc -pthread -o shrdmem_mutex shrdmem.c
$ ./shrdmem_mutex
Thread 2: holding 1001990720
Thread 1: holding 2000000000
```

3.2 Bounded buffer problem

```c
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>

#define BUF_SIZE 2
#define THREADS 1 // 1 producer and 1 consumer
#define LOOPS 3 * BUF_SIZE // variable

// Initiate shared buffer
int buffer[BUF_SIZE];
int fill = 0;
int use = 0;

/*TODO: Fill in the synchronization stuff */

void put(int value); // put data into buffer
int get();           // get data from buffer

void * producer(void * arg) {
  int i;
  int tid = (int) arg;
  for (i = 0; i < LOOPS; i++) {
    /*TODO: Fill in the synchronization stuff */
    put(i); // line P2
    printf("Producer-%d-put-data-%d\n", tid, i);
    sleep(1);
    /*TODO: Fill in the synchronization stuff */
  }
  pthread_exit(NULL);
}

void * consumer(void * arg) {
  int i, tmp = 0;
  int tid = (int) arg;
  while (tmp != -1) {
    /*TODO: Fill in the synchronization stuff */
    tmp = get(); // line C2
    printf("Consumer-%d-get-data-%d\n", tid, tmp);
    sleep(1);
    /*TODO: Fill in the synchronization stuff */
  }
  pthread_exit(NULL);
}

int main(int argc, char ** argv) {
  int i, j;
  int tid[THREADS];
  pthread_t producers[THREADS];
  pthread_t consumers[THREADS];

  /*TODO: Fill in the synchronization stuff */

  for (i = 0; i < THREADS; i++) {
    tid[i] = i;
    // Create producer thread
    pthread_create( & producers[i], NULL, producer, (void * ) tid[i]);

    // Create consumer thread
    pthread_create( & consumers[i], NULL, consumer, (void * ) tid[i]);
  }

  for (i = 0; i < THREADS; i++) {
    pthread_join(producers[i], NULL);
    pthread_join(consumers[i], NULL);
  }

  /*TODO: Fill in the synchronization stuff destroy (if needed) */

  return 0;
}

void put(int value) {
  buffer[fill] = value;           // line f1
  fill = (fill + 1) % BUF_SIZE; // line f2
}

int get() {
  int tmp = buffer[use];          // line g1
  buffer[use] = -1;               //clean the item
  use = (use + 1) % BUF_SIZE;   // line g2
  return tmp;
}
```

Step 3.2.1 Compile and execute the program

```
# The program name must match your own code
$ gcc -pthread -o pc pc.c
$ ./pc
Consumer 0 get data 0
Producer 0 put data 0
Consumer 0 get data 0
Producer 0 put data 1
Consumer 0 get data 1
Consumer 0 get data 1
Consumer 0 get data 1
...
```

Step 3.2.2 Recognize the issue and propose a fix mechanism using the provided synchronization tool.
Hint: sem_wait() and sem_signal() are useful to protect consumer() and producer() thread worker

```
# Implement by your self the fixed pc_sem program (it is not available yet)
# copycat may result error gcc: fatal error: no input files
$ ./pc_sem
Producer 0 put data 0
Consumer 0 get data 0
Producer 0 put data 1
Consumer 0 get data 1
Producer 0 put data 2
Consumer 0 get data 2
Producer 0 put data 3
Producer 0 put data 4
Consumer 0 get data 3
Producer 0 put data 5
Consumer 0 get data 4
Consumer 0 get data 5
...
```

4 Exercise

PROBLEM 1 Design and implement sequence lock API.

```
#include "seqlock.h" /* TODO: implement this header file */

/*
 * TODO: Implement these following APIs
 */
pthread_seqlock_t lock;

/* Init with default NULL attribute attr=NULL */
int pthread_seqlock_init(pthread_seqlock_t *seqlock);
int pthread_rwlock_destroy(pthread_seqlock_t *seqlock);
------------------------------------------------------------------------
int pthread_seqlock_rdlock(pthread_seqlock_t *seqlock);
int pthread_seqlock_rdunlock(pthread_seqlock_t *seqlock);
int pthread_seqlock_wrlock(pthread_seqlock_t *seqlock);
int pthread_seqlock_wrunlock(pthread_seqlock_t *seqlock);
```

The reader/writer conflict resolution following the description:
Reader-writer lock conflict resolution

- When there is no thread in the critical section, any reader or writer can enter into a critical section. But only one thread can enter.
- If the reader is in critical section, the new reader thread can enter ocasionally, but the writer cannot enter.
- If the writer is in critical section, no other reader or writer can enter.
- If there are some readers in the critical section by taking the lock, and there is a writer want to enter. That writer has to wait if another reader is coming until all of readers have finish. That why this mechanism is reader prefer.

Sequence lock conflict resolution the conflict resolution mechanism in teader-writer lock can cause writer starvation. The following policy is implemented by the sequence lock:

- When no one is in the critical section, one writer can enter the critical section and takes the lock, increasing the sequence number by one to an odd value. When the sequence number is an odd value, the writing is happening. When the writing has been done, the sequence is back to even value. Only one writer is allow into critical section.
- When the reader wants to read data, it checks the sequence number which is an odd value, then it has to wait until the writer finish.
- When the value is even, many readers can enter the critical section to read the value.
- When there are only a reader and no writer in the critical section, if a writer want to enter the critical section it can take the lock without blocking.

PROBLEM 2 (*Aggregated Sum*)

Implement the thread-safe program to calculate the sum of a given integer array using $< tnum >$ number of threads. The size of the given array $< arrsz >$ and the $< tnum >$ value is provided in the program arguments. You are provided a pre-processed argument program with the usage as the following description.

```
aggsum, version 0.01

 usage:  aggsum arrsz tnum [seednum]

 Generate randomly integer array size <arrsz> and calculate sum parallelly
 using <tnum> threads. The optional <seednum> value use to control the
 randomization of the generated array.

   Arguments:

     arrsz     specifies the size of array.
     tnum      number of parallel threads.
     seednum   initialize the state of the randomized generator.
```

The last argument $< seednum >$ is an already implemented mechanism. This value is used to generate the integer values in the array and we don't touch it to keep it for later validated testcase generation. The data generation mechanism is also provided. Call the following routine to fill in the "buf" shared memory buffer.

```
int generate_array_data (int* buf, int arraysize, int seednum);
```

TODO: You have to implement the following thread routine.

```
struct _range {
  int start;
  int end;
};

void* sum_worker (struct _range idx_range) {
  //printf("In worker from %d to %d\n", idx_range.start, idx_range.end);

  /*
   * TODO implement a thread safe sum operator works in the range
   *       i.e. for (i=idx_range.start; i<= idx_range.end; i++){}
   *      and write the sum to sumbuff  (in program global data)
   */
}

int main()
{
   pthread_t tid; /* Sample code is only single thread */
   struct _range thread_idx_range;
```

```
    ...
    /* Sample code use full range */
    thread_idx_range.start = 0;
    thread_idx_range.end = arrsz - 1;

    ...
    /* TODO: implement multi-thread mechanism */
    pthread_create(&tid, NULL, sum_worker, thread_idx_range));
}
```

PROBLEM 3 (*Interruptable system logger*)

Design and implement a logger support the two operations *wrlog*() and *flushlog*() to manipulate the log data buffer "logbuf"

```
#define MAX_LOG_LENGTH 10
#define MAX_BUFFER_SLOT 5
char ** logbuf;

int wrlog(char** logbuf, char* new_data);
int flushlog(char** logbuf);
```

In this problem, there are many programs or actors that call wrlog() to append the log data to a shared buffer (i.e., log-file cache) which can be flushed to disk eventually. For simplicity, we assume the buffer contains 5 (= MAX_BUFFER_SLOT) data slots and the flush event occurs periodically at time out. We also assume a LOG is a fixed length string, i.e. *char new_log*[MAX_LOG_LENGTH].

In practical point of view, the behavior of the system can be illustrated as a sequence of writing log data (**wrlog()**) and the flush (**flushlog()**) will be periodically triggered when a timeout is occured.

```
int main()
{
  wrlog(data1);
  wrlog(data2);
  wrlog(data3);
  ...
  wrlog(dataN);
}
```

In summary, the draft operations of the logger are:

wrlog() append data to shared buffer but not exceed the buffer size. If it reaches the limits, it needs to wait until the buffer is flushed.

flushlog() clean buffer aka. move all the stored items to somewhere (in here is printing to screen) and then delete all of them. This action runs eventually; for simplicy we just make a periodical call here.

Further development: The interruptable mechanism of flush log can (further) support more unpredict events, i.e., it can handle a signall $SIGUSR1$, $SIGUSR2$ which are introduced in the previous lab.

You are provided a referenced code with non-protected buffer by default, the program's output is:

```
$./logbuf
flushlog()
wrlog(): 0
wrlog(): 1
wrlog(): 2
wrlog(): 3
wrlog(): 4
wrlog(): 7
wrlog(): 12
wrlog(): 13
wrlog(): 16
wrlog(): 17
wrlog(): 21
wrlog(): 24
wrlog(): 11
wrlog(): 26
wrlog(): 14
wrlog(): 6
wrlog(): 27
wrlog(): 28
wrlog(): 8
wrlog(): 20
wrlog(): 9
wrlog(): 22
wrlog(): 10
wrlog(): 19
wrlog(): 25
wrlog(): 29
wrlog(): 18
wrlog(): 23
wrlog(): 5
wrlog(): 15
flushlog()
Slot  0: 0
Slot  1: 1
Slot  2: 2
Slot  3: 3
Slot  4: 4
Slot  5: 7
Slot  6: 12
flushlog()
flushlog()
flushlog()
```

TODO: Implement the protection mechanism for wrlog() and flushlog() routines to make it a safe data accessing (to buffer). If it has a proper configuration then the program behavior is somehow like this illustration (**comment out** the print function name in wrlog() and flushlog() to make **this clean output**).

```
$ ./logbuf
Slot   0: 0
Slot   1: 1
Slot   2: 2
Slot   3: 3
Slot   4: 4
Slot   5: 5
Slot   0: 12
Slot   1: 6
Slot   2: 7
Slot   3: 10
Slot   4: 8
Slot   5: 11
Slot   0: 17
Slot   1: 13
Slot   2: 16
Slot   3: 15
Slot   4: 14
Slot   5: 18
Slot   0: 25
Slot   1: 20
Slot   2: 19
Slot   3: 21
Slot   4: 22
Slot   5: 23
Slot   0: 27
Slot   1: 28
Slot   2: 26
Slot   3: 24
Slot   4: 29
Slot   5: 9
```

PROBLEM 4 (***Wait and notify***) Use condition variables to manage waiting processes, allowing processes to notify others when resources become available.

```
int buffer[BUFFER_SIZE];
int count = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;

int write(...)
```

```
{

        pthread_mutex_lock(&mutex);
        while (count == BUFFER_SIZE) {// Buffer is full
            pthread_cond_wait(&cond_var, &mutex);
        }
        // Add item to buffer
        buffer[count++] =  ...

        // Notify writer that an item is available
        // TODO: enhance the mechanism to ensure this happens
        //       when condition vars are locked
        pthread_cond_signal(&cond_var);
        pthread_mutex_unlock(&mutex);
}

int read(...)
{
        ...
        while (count == 0) {// Buffer is empty
            pthread_cond_wait(&cond_var, &mutex);
        }
        // Retrieve and get_item from buffer
        get_ite(buffer[--count]);

        // Notify reader that there is space in the buffer
        // TODO: enhance the mechanism to ensure this happens
        //       when condition vars are locked
        pthread_cond_signal(&cond_var);
        pthread_mutex_unlock(&mutex);
        ...
}
```

The condition variables provide a synchronization primitive that allows threads to wait for certain conditions to occur. They are typically used in conjunction with mutexes to protect shared resources.

Using condition variables correctly can help avoid deadlocks, but it's important to follow best practices. Students need to define a mechanism release the lock before waiting for a condition to be met.

PROBLEM 5 (*Periodic detection with recovery*) It performs periodic checks allowing the system to remain responsive then it can incorporate various recovery strategies.

```
int is_safe() {
    //TODO: perform polling checks
    ....
    if (!found) {
        return -1; // Unsafety detected
```

```
    }
}

int periodical_detector(void *arg) {
    while (1) {
        sleep(5); // Periodic check every 5 seconds
        pthread_mutex_lock(&lock);
        if (!is_safe()) {
            // TODO: Abnormal detected! Taking corrective action...
                if (!finished) {
                    break; // Break after terminating
                }
        }
    }
    return 0;
}

int main()
{
     //TODO: pthread_create to activate periodical_detector
}
```

PROBLEM 6 (***Asynchronous resource requests***) Processes can request resources without needing to block if those resources are not immediately available. Instead, they can register a callback or a notification to be informed when the resource becomes available.

Resource set: handle multiple processes and ensure that resources are allocated fairly.

```
typedef struct {
    int id; // Process ID
    int requested_resources; // Resources the process requests
    void (*callback)(int); // Callback function for resource allocation
} process_request_t;
int available_resources = NUM_RESOURCES;
pthread_mutex_t resource_lock;
pthread_cond_t resource_cond;
```

Call back mechanism: the registration of a callback function is used to be informed when the resource becomes available.

```
void resource_callback(int process_id) {
    // TODO: perform action when resource available
}
```

Non-blocking request management: when a resource becomes available, the resource set management can invoke a callback function or signal the requesting process.

```
int resource_man(void* arg)
```

```
{
    process_request_t* request = (process_request_t*)arg;

    pthread_mutex_lock(&resource_lock);

    // Non-blocking allocation
    while (request->requested_resources > available_resources) {
        printf("Process %d waiting for resources\n", request->id);
        pthread_cond_wait(&resource_cond, &resource_lock);
    }

    // Allocate resources
    available_resources -= request->requested_resources;
    request->callback(request->id);

    //TODO: perform process
    // ...

    // Release resource
    available_resources += request->requested_resources;
    printf("Process %d released resources\n", request->id);

    pthread_cond_broadcast(&resource_cond);
    pthread_mutex_unlock(&resource_lock);
}
```

PROBLEM 7 (*Lock-Free data structures*). Lock-free data structures rely on atomic operations provided by modern CPU which ensures that updates to shared data are performed safely

```
#include <stdatomic.h>
//TODO: implementation of a lock-free stack
typedef struct LockFreeStack {
    atomic_node* head; // Pointer to the top node
} LockFreeStack;

// Push an item onto the stack
bool push(LockFreeStack* stack, int value) {
   // TODO: Set new node's next to current head
}

// Pop an item from the stack
bool pop(LockFreeStack* stack, int* value) {
    //TODO // Get the value to return and free the popped node
}
```

Revision History

Revision	Date	Author(s)	Description
...	...	...	
2.0	10.2022	PD Nguyen	Update lab content, practices and exercises
3.0	10.2023	PD Nguyen	Update practices and exercises
3.1	10.2024	PD Nguyen	Update resource allocation exercises