## Operating Systems Principles Assignment 1 Report

Student ID: s3653030

Student Name: Qi Sun

Github Name & URL: OSP\_A1

https://github.com/jjsunshr/OSP\_A1

## The Producer-Consumer Problem For the Producer-Consumer problem, there would be five producer threads and five consumer threads created with specified produce and consume function as the working function.

## And pthread mutex and conditional variable are used for the above multi threads program to control thread synchronization and avoid deadlock.

## For the producer working function (producer\_worker): after the producer thread gets the mutex (global mutex), it would check the number of products in the buffer (buffer was used for the producer to store the produced product), if the buffer is full, the producer thread would wait conditionally until one consumer thread consume one product from the buffer, else if the buffer is not full, the producer thread would produce product and add it at the buffer, and then send the conditional variable (cond) to the consumer thread. At the end, the mutex would be release for other threads using.

## For the consumer worker function (consumer\_worker), when it gets the mutex (global mutex), it would first check whether the buffer is empty, if so it would be blocked until the conditional variable (cond) signaled the consumer for consuming. When the buffer is not empty, the consumer thread would consume one product from the buffer and then send the conditional variable (cond) to the producer threads. At the end, consumer thread would release the lock (global mutex).

## In the overall program design, sleep function would be used to keep the produce and consume process slowly so that the output would be clearly for us make the analysis. Also the producer threads in our program would sleep less time than the consumer thread so that there would be always products for the consumer threads.

## Real world application

**1. Online shopping order process:**

Users (consumer) would submit their orders to the online shopping system, and there would be a buffer to store different users’ orders, and many background threads would get order from the buffer one by one and process it. Such scenario can be thought as a Producer-Consumer application. There would be mutex used to control multi-threads synchronization, also conditional variable would be used between the order makers threads and order process threads to pass message between them.

**2. Highly concurrent web server processing user requests**:

For the highly concurrent web server, different requests types from the users would be classified and stored in the queues, which can be thought as the producer threads job. And when the specified queue is not empty, consumer threads would popped the request out of the queue and complete and response the user, such can be thought as the job of the producer threads. And mutex would be used so that one time just one thread can access the request queue for adding or popping. And also conditional variable would be used to remind the producer add new request or pop request from the queue by the consumer thread.

## The Dining Philosophers' Problem

## For the Dining Philosophers’ problem, there would be five Philosophers sitting in a circle with five forks around them. One philosopher can eat when taking two forks around he/she, or else he/she can’t eat dinner.

## For the overall design of the dining philosophers’ problem, there would be five fork mutexs created as a fork mutex array. Also five threads would be declared as a philosopher array. For the philosopher worker function, at first the philosopher index would be got from the parameter, and then left fork and right fork for this philosopher would be calculated, the specified philosopher would first sleep for a random time, and then acquire for the left fork (mutex), if acquires the left fork, the philosopher would then sleep a random time and acquire for the right fork, if the right fork is occupied, the left fork of the philosopher would be released, else the philosopher would sleep for random time and eating dinner, at the end the philosopher would release the left fork and the right fork.

## For the overall design, there would be just mutex used for the thread synchronization. When the specified philosopher gets the mutex for the left fork, then he/she will sleep with a random time and then try to get the right fork mutex, if he/she fails to get the right fork, to avoid deadlock, he/she would release the left fork already in hand. After eating within a random time, the philosopher would then release the left and right fork in hands. With the requirements of the assignment, the proposed multi-thread program should be fair for the philosopher to get nearly fairly time for eating, so an extra array would be used to record the current eat time so far, if the current philosopher eat time is less than the average eat time among the philosophers or is equal to 0, then the current philosopher would eat the dinner, else would not. From the running results, we can find the above proposed auxiliary array could make all philosophers got fairly eat time.

## Real world application

1. **Bank transaction**: two users make transaction to the other at the same time: two transactions can be thought as two transaction threads, when two thread hold resources and waiting for the other one thread release the resource, deadlock would occur. And there would be a user defined priority to determine which thread would hold all resources at first, after the thread completes, the other thread would hold the resources and make the transaction, which is familiar with the database Atom operation.

2. **Operating system process or thread scheduling strategies**: Multi-threads running in the system for the same target, limited resources available, the threads would be scheduled with priority such as waiting longest first or other mechanisms to ensure the threads won’t be allocated limited resources at the same time, and so multi-thread would run in sequence without deadlock.