COMP3230 Principles of Operating Systems
Problem Set #2

Due date: 23:59 pm, October 27, 2023

Total 4 points

(version: 1.0)

Note: This assignment weights a total of 4 points in the final marks of this course. However, just like each process has its own private virtual address space, this assignment also has its own virtual score space of 100 points, which is mapped to 4 points of physical marks. The below points for each question are given in the said virtual score space.

Question 1 – **Synchronization** (50 points)

(This question is related to ILO 2c – "describe the principles and techniques used by OS to support concurrency and synchronization control".)

This is a solution (pseudocode) to the Readers-Writers problem given in the class.

```
sem_t semMutex, readMutex;
int readcount;

void init(void) {
      sem_init(&semMutex, 0, 1);
      sem_init(&readMutex,0, 1);
      readcount = 0;
}

void Writer(void) {
      sem_wait(&semMutex);
      do_writing();
      sem_post(&semMutex);
}
```

```
void Reader(void) {
      sem_wait(&readMutex);
      readcount++;
      if (readcount == 1)
            sem_wait(&semMutex);
      sem_post(&readMutex);

      do_reading();

      sem_wait(&readMutex);
      readcount--;
      if (readcount == 0)
            sem_post(&semMutex);
      sem_post(&readMutex);
}
```

The above solution is implemented by using semaphores. Design another solution of the Readers-Writers problem by using condition variables **and** mutex locks. Write down your pseudo code to implement the `Writer()` and `Reader()` functions. Please include declaration and initialization of necessary variables, if any.

```c
int readerCount, writerCount;
pthread_mutex_t mutex;
pthread_cond_t read;
pthread_cond_t write;

void init() {
    readerCount = 0;
    writerCount = 0;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&read, NULL);
    pthread_cond_init(&write, NULL);
}

void reader() {
    // Wait for writers to finish
    pthread_mutex_lock(&mutex);
    if (writerCount > 0) {
        pthread_cond_wait(&read, &mutex);
    }
    readerCount++;
    pthread_mutex_unlock(&mutex);

    do_reading();

    // Signal that all readers have finished
    pthread_mutex_lock(&mutex);
    readerCount--;
    if (readerCount == 0) {
        pthread_cond_signal(&write);
    }
    pthread_mutex_unlock(&mutex);
}

void writer() {
    // Wait for readers and writers to finish
    pthread_mutex_lock(&mutex);
    if (readerCount > 0 || writerCount > 0) {
        pthread_cond_wait(&write, &mutex);
    }
    writerCount++;
    pthread_mutex_unlock(&mutex);

    do_writing();

    // Signal that all writers have finished
    pthread_mutex_lock(&mutex);
    writerCount--;
    pthread_cond_signal(&write);
    pthread_cond_broadcast(&read);
    pthread_mutex_unlock(&mutex);
}
```

Question 2 – **Synchronization** (50 points)

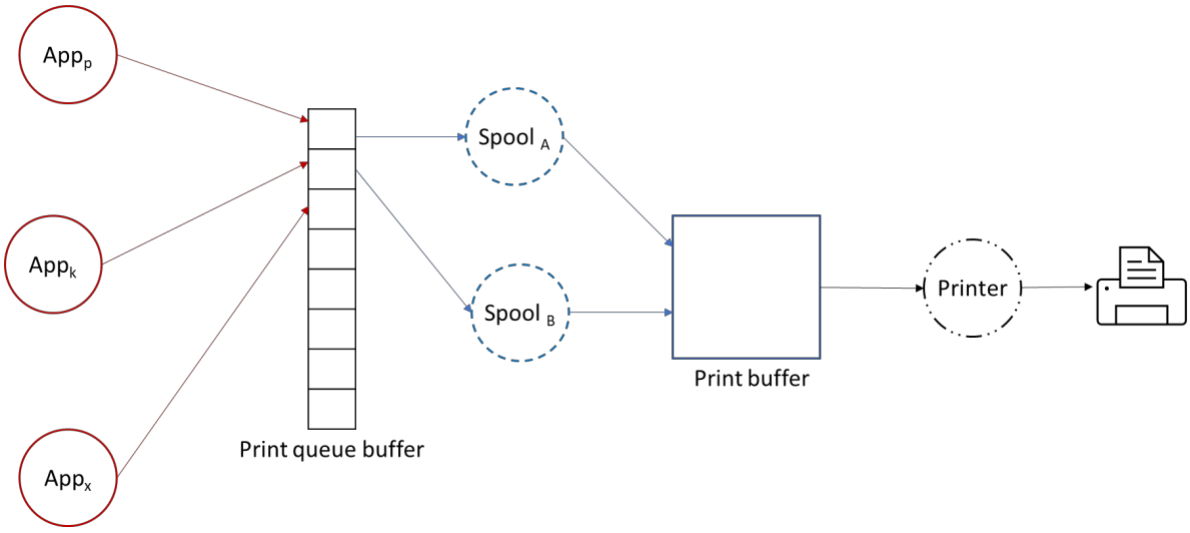
(This question is related to ILO 2c – "describe the principles and techniques used by OS to support concurrency and synchronization control".)

Consider the print spooler service provided by an Operating System.

When an application wants to print a file, it calls the printfile() system call, which results in placing a request in the next available slot of the print queue buffer. Assume the print queue buffer has **N buffer slots**; each can store an individual print request.

The kernel has **K spooler threads** running in the background. Each thread is responsible for processing a print request to generate an output file and placing the output file in the printer buffer. Assume the print buffer can only keep one output file at a time.

There is **a printer thread** running in the kernel space. When it detects that there is an output file in the print buffer, it will pass the file to the printer for printing and remove the file from the print buffer.



Write three functions that implement (1) the printfile() logic, (2) the spooler thread, and (3) the printer thread and use **semaphores** to coordinate the three threads in accessing the print queue buffer and the print buffer. **Present the logic in C-like pseudocode statements.**

Here are the data structures and utility functions relevant to the case.

/* this is the print queue buffer; each slot stores a request of type request_t */

```
request_t pqReqBuff[N];
```

/* this is for counting the number of requests in the print queue buffer */

```
int pqReqCount = 0;
```

/* this is the data type of the print buffer for temporary storing of the output file */
```
buffer_t pBuffer;
```

/* enqueue() - for adding a new request to the print queue buffer */

/* It updates the pqReqBuff array and pqReqCount variable, so MUST be called after getting the guard lock, i.e., should be called inside the critical section */

```
void enqueue(request_t req);
```

/* dequeue() - for getting (& removing) a request from the print queue buffer */

/* It updates the pqReqBuff array and pqReqCount variable, so MUST be called after getting the guard lock, i.e., should be called inside the critical section */

```
void dequeue(request_t *req);
```

/* process() - for processing the print request to generate the output file of type buffer_t; should be called outside the critical section */

```
void process(request_t req, buffer_t *output);
```

/* output() - for adding the output file to the printer buffer */

/* It updates the pBuffer variable, so MUST be called after getting the guard lock, i.e., should be called inside the critical section */

```
void output(buffer_t output);
```

/* retrieve() - for getting (& removing) the output file from the print buffer */

/* It updates the pBuffer variable, so MUST be called after getting the guard lock, i.e., should be called inside the critical section */

```
void retrieve(buffer_t *output);
```

/* printing() - for sending the output file to the hardware (printer); should be called outside the critical section */

```
void printing(buffer_t output);
```

(1) (10 points) Declare and initialize the semaphores (`sem_t`) and any (global) variables used in this application. Note: `sem_t` semaphores are general semaphores; use `sem_init()` to initialize the semaphores.

```
// Declare the semaphores here
sem_t pq_empty, pq_full, pq_mutex;
sem_t pBuff_empty, pBuff_full, pBuff_mutex;

// Initialize the semaphores here

void init() {
    // The Locks (Binary Semaphores)
    sem_init(&pq_mutex, 0, 1);
    sem_init(&pBuff_mutex, 0, 1);

    // The Counters
    sem_init(&pq_empty, 0, N);
    sem_init(&pq_full, 0, 0);

    sem_init(&pBuff_empty, 0, 1);       // Buffer Size = 1
    sem_init(&pBuff_full, 0, 0);
}
```

(2) (10 points) Implement the `printfile()` logic by completing the following function. Note: use `sem_wait()` and `sem_post()` to access the semaphores.

```
void printfile(request_t req) {
    sem_wait(&pq_empty);
    sem_wait(&pq_mutex);

    enqueue(req);

    sem_post(&pq_mutex);
    sem_post(&pq_full);
}
```

(3) (20 points) Implement the spooler thread logic by completing the following function. Note: use sem_wait() and sem_post() to access the semaphores.

```
void Spooler() {
    while(1) {
        // For retrieving the request from the print queue buffer
        request_t req;
        buffer_t output;        // For the output file

        // Consume the request from the print queue buffer
        sem_wait(&pq_full);
        sem_wait(&pq_mutex);

        dequeue(&req);

        sem_post(&pq_mutex);
        sem_post(&pq_empty);

        // Process the request
        process(req, &output);

        // Produce the output file
        sem_wait(&pBuff_empty);
        sem_wait(&pBuff_mutex);

        Output(output);

        sem_post(&pBuff_mutex);
        sem_post(&pBuff_full);
    }
}
```

(4) (10 points) Implement the printer thread logic by completing the following function. Note: use sem_wait() and sem_post() to access the semaphores.

```
void Printer() {
    while(1) {
        buffer_t output; // For the output file

        sem_wait(&pBuff_full);
        sem_wait(&pBuff_mutex);

        retrieve(&output);

        sem_post(&pBuff_mutex);
        sem_post(&pBuff_empty);

        // Print the output file
        printing(output);

    }
}
```