Programming Project 8

OS2025-SWU-软工中外 34 班

Spring 2025

Experimental topic

Programming Projects in Chapter 7

Page P-40 to P-44 of Operating System Concepts (10th) or this pdf

Experimental resources

/resources/code_for_project/code_for_project8

Experimental report naming format:

StudentID-name-Project-X (e.g., 2220183211-张三-Project-1)

Experimental objectives 1. (Course Objective 2) Master the basic concepts, design ideas and operation status of the process. Understand the basic principles and ideas of process synchronization and mutual exclusion, and identify the key links, steps and constraints of engineering problems in this field. Use the basic means of implementing process synchronization and mutual exclusion to analyze and solve the generator consumer problem in the field of process synchronization and mutual exclusion.

Experimental report submission

> **Attention**
>
> Please upload your report with naming format to the ftp server within **two weeks** (Deadline: 2025-05-27).

1 Producer–Consumer Problem

In Section 7.7.1, we presented a semaphore-based solution to the producer–consumer problem using a bounded buffer. In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 5.9 and 5.10. The solution presented in Section 7.7.1 uses three semaphores: `empty` and `full`, which count the number of `empty` and `full` slots in the buffer, and `mutex`, which is a binary (or mutual-exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, you will use standard counting semaphores

for `empty` and `full` and mutex lock, rather than a binary semaphore, to represent `mutex`. The producer and consumer—running as separate threads—will move items to and from a buffer that is synchronized with the `empty`, `full`, and `mutex` structures. You can solve this problem using either Pthreads or the Windows API.

2 The Buffer

Internally, the buffer will consist of a fixed-size array of type `buffer_item` (which will be defined using a typedef). The array of `buffer_item` objects will be manipulated as a circular queue. The definition of `buffer_item`, along with the size of the buffer, can be stored in a header file such as the following:

```
/* buffer.h */
typedef int buffe_ item;
#define BUFFER_SIZE  5
```

The buffer will be manipulated with two functions, `insert_item()` and remove_item(), which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears in the following code.

```
#include "buffer.h"
/* the buffer */
buffer_item buffer[BUFFER SIZE];
int insert_item(buffer_item item) {
        /* insert item into buffer
        return 0 if successful, otherwise
        return -1 indicating an error condition */
}

int remove_item(buffer_item *item) {
        /* remove an object from buffer placing it in item
        return 0 if successful, otherwise
        return -1 indicating an error condition */
}
```

The `insert_item()` and remove_item() functions will synchronize the producer and consumer using the algorithms outlined in Figures 7.1 and 7.2. The buffer will also require an initialization function that initializes the mutual-exclusion object mutex along with the `empty` and `full` semaphores.

The `main()` function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the `main()` function will sleep for a period of time and, upon awakening, will terminate the application. The `main()` function will be passed three parameters on the command line:

1. How long to sleep before terminating

2. The number of producer threads
3. The number of consumer threads

A skeleton for this function appears in following code.

```
#include "buffer.h"
int main(int argc, char *argv[]) {
        /* 1. Get command line arguments argv[1], argv[2], argv[3] */
        /* 2. Initialize buffer */
        /* 3. Create producer thread(s) */
        /* 4. Create consumer thread(s) */
        /* 5. Sleep */
        /* 6. Exit */
}
```

3 The Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the `rand()` function, which produces random integers between 0 and `RAND_MAX`. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer. An outline of the producer and consumer threads appears in the following code or.

As noted earlier, you can solve this problem using either Pthreads or the Windows API. In the following sections, we supply more information on each of these choices.

4 Pthreads Thread Creation and Synchronization

Creating threads using the Pthreads API is discussed in Section 4.4.1. Coverage of mutex locks and semaphores using Pthreads is provided in Section Section 7.3. Refer to those sections for specific instructions on Pthreads thread creation and synchronization.

```
#include <stdlib.h> /* required for rand() */
#include "buffer.h"

void *producer(void *param) {
   buffer_item item;

   while (true) {
      /* sleep for a random period of time */
      sleep(...);
      /* generate a random number */
      item = rand();
      if (insert_item(item))
         fprintf("report error condition");
      else
         printf("producer produced %d\n",item);
}

void *consumer(void *param) {
   buffer_item item;

   while (true) {
      /* sleep for a random period of time */
      sleep(...);
      if (remove_item(&item))
         fprintf("report error condition");
      else
         printf("consumer consumed %d\n",item);
}
```

Fig. 3.1: An outline of the producer and consumer threads