OS_Assignment8

Assign	
tag	homework
姓名	周鹏宇
学号	2019K8009929039

1. 设有两个优先级相同的进程 `T1`，`T2` 如下。令信号 `S1`，`S2` 的初值为 `0`，已知 `z = 2`，试问 `T1`，`T2` 并发运行结束后 `x`、`y`、`z` 的值？

```
线程 T1                 线程 T2
1   y := 1;             x := 1;
2   y := y + 2;         x := x + 1;
3   V(S1);              P(S1);
4   z := y + 1;         x := x + y;
5   P(S2);              V(S2);
6   y := z + y;         z := x + z;
```

- 由于T1在第三行对信号量S1做V操作，而T2在第三行对信号量S1做P操作，故T2第三行必然发生在T1第三行之后，同理，T1第5行必然发生在T2第五行之后
- 首先关注以S1结点的前半部分
  - 一种情况是在T1进行V(S1)操作后继续进行直至P(S2)，之后T2运行，那么此时T1完成1、2、4行，随后T2完成1、2、4行
  - 另一种情况是T1进行V(S1)操作后由T(2)运行，在运行第四行之前又切回T1，T1运行至第5行，随后T2运行到第五行或第六行，此阶段完成顺序为T1 1 2、T2 1 2 、T1 4 、T2 4
  - 但实际上可以发现，两者间的1、2、4行没有相关性，执行顺序的改变并不影响最后的结果，换言之信号量S1所引起的不确定性是可以忽略的
- 然后关注后半部分
  - 可能是T2执行4、6，T1执行4、6
  - 可能是T2执行4、T1执行4、T2执行6、T1执行6
  - 可能是T1执行4、T2执行4、6，T1执行6
  - T2执行4、可能是T1执行4，6；T2执行6

- 此时可以注意到，T1的6相对于T2的4，关系是绝对滞后的，故T1的6对T2的4的影响可忽略，则第二项和第三项可以合并

则结果有：

- `x = 5`、`y = 7`、`z = 4`
- `x = 5`、`y = 12`、`z = 9`
- `x = 5`、`y = 7`、`z = 9`

2. 银行有 `n` 个柜员，每个顾客进入银行后先取一个号，并且等着叫号，当一个柜员空闲后，就叫下一个号

   请用 `PV` 操作分别实现：

- 顾客取号操作 `Customer_Service`
- 柜员服务操作 `Teller_Service`

简单分析一下以后发现此题似乎不需要互斥信号量，不过两个进程之间通信还是需要一个信号量来同步资源的，则给出伪代码如下：

```
semaphore waiting = n;              // 信号量，等待的顾客数

int Customer_Service() {
    // 增加等待的顾客数
    V(waiting);
}

int Teller_Service() {
    while (True) {
        // 等待有顾客取号
        P(waiting);
        // 服务
        do_service();
    }
}
```

3. 多个线程的规约（Reduce）操作是把每个线程的结果按照某种运算（符合交换律和结合律）两两合并直到得到最终结果的过程。

   试设计管程 `monitor` 实现一个 `8` 线程规约的过程，随机初始化 `16` 个整数，每个线程通过调用 `monitor.getTask` 获得 `2` 个数，相加后，返回一个数 `monitor.putResult`，然后再 `getTask()` 直到全部完成退出，最后打印归约过程和结果。

   要求：为了模拟不均衡性，每个加法操作要加上随机的时间扰动，变动区间 `1~10 ms`。

提示：使用 `pthread_` 系列的 `cond_wait`、`cond_signal`、`mutex` 实现管程，使用 `rand()` 函数产生随机数，和随机执行时间。

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

#define N_TASK 16
#define ONE_MS 6000000
#define N_THREAD 8

typedef struct {
  int a, b;
} data;

typedef struct node {
  int num;
  struct node *next;
} node;

typedef struct {
  int cnt;
  node *head;
  node *nail;
} queue;

void queue_init(queue * que)
{
  que->cnt = 0;
  que->head = NULL;
  que->nail = NULL;
}

void queue_push(queue * que, int num)
{
  node *n = malloc(sizeof(node));
  n->num = num;

  if (que->head == NULL) {
    que->head = n;
    que->nail = n;
    n->next = NULL;
  } else {
    que->nail->next = n;
    que->nail = n;
  }
  que->cnt += 1;
}

int queue_pop(queue * que)
{
  node *n = que->head;
  int result = n->num;
  if (n->next == NULL) {
    que->head = NULL;
```

```
    que->nail = NULL;
  } else {
    que->head = n->next;
  }
  que->cnt -= 1;
  free(n);
  return result;
}

int queue_print(queue * que)
{
  for (node * p = que->head; p != NULL; p = p->next) {
    printf("%d ", p->num);
  }
  printf("\n");
}

typedef struct {
  queue que;
  pthread_mutex_t lock;
  pthread_cond_t reduce;
  int task;
} monitor;

void monitor_init(monitor * p)
{
  p->task = N_TASK - 1;
  queue_init(&p->que);
  for (int i = 0; i < N_TASK; i++) {
    queue_push(&p->que, rand() % 100);
  }

  pthread_mutex_init(&p->lock, NULL);
}

bool monitor_getTask(monitor * p, data * result)
{
  pthread_mutex_lock(&p->lock);

  if (p->task <= 0) {
    pthread_mutex_unlock(&p->lock);
    return false;
  }

  p->task -= 1;

  while (p->que.cnt < 2) {
    pthread_cond_wait(&p->reduce, &p->lock);
  }

  result->a = queue_pop(&p->que);
  result->b = queue_pop(&p->que);

  printf("getTask: %d, %d\n", result->a, result->b);

  pthread_mutex_unlock(&p->lock);
  return true;
}
```

```
void monitor_putResult(monitor * p, int res)
{
  pthread_mutex_lock(&p->lock);

  queue_push(&p->que, res);
  printf("putResult %d\n", res);

  pthread_cond_signal(&p->reduce);

  pthread_mutex_unlock(&p->lock);
}

volatile void delay(unsigned time)
{
  // struct timeval t_start, t_end;
  // gettimeofday(&t_start, NULL);
  unsigned cnt = 0;
  for (unsigned i = 0; i < time; i++) {
    unsigned j = ONE_MS;
    while (j--) {
      cnt += i;
    }
  }
  // gettimeofday(&t_end, NULL);
  // printf("time: %ld", t_end.tv_usec - t_start.tv_usec);
}

monitor task_monitor;

void *func_thread_reduce(void *arg)
{
  data task;
  while (1) {
    if (!monitor_getTask(&task_monitor, &task)) {
      printf("End\n");
      return NULL;
    }
    delay(rand() % 10);
    int res = task.a + task.b;
    monitor_putResult(&task_monitor, res);
  }
  return NULL;
}

int main()
{
  monitor_init(&task_monitor);
  printf("Initial Queue:\n");
  queue_print(&task_monitor.que);

  pthread_t thread[N_THREAD];
  for (int i = 0; i < N_THREAD; i++) {
    pthread_create(thread + i, NULL, func_thread_reduce, NULL);
  }
  for (int i = 0; i < N_THREAD; i++) {
    pthread_join(thread[i], NULL);
  }
```

```
  printf("Result:\n");
  queue_print(&task_monitor.que);

  return 0;
}
```