作业 7-8

褚晓敏

2020 年 11 月 3 日

1 作业 7

1. 资源 0,1 已经没有空闲，但是 A-C 仍然需要，因此不可能先结束；需要先把剩余资源分配给 D 让 D 结束，这就要求 x 至少是 1. D 结束之后剩余资源分别是 1,1,x+1,2,1 个. 资源 1 仍然不可能满足 B，资源 4 仍然不可能满足 A，因此必须先满足 C，此时要求 x 至少是 2. C 结束之后剩余资源分别是 2,2,x+1,3,1；注意到 A 对资源 4 的要求不可能满足，因此先将资源分配给 B，让 B 退出，A 进程独自无限等待，x 为 2 时已经可以做到. 因此 x 最小为 2.

2. 3 个资源是对称的，不妨设 A 请求的顺序是 1,2,3. 如果 B 最先提出的请求是 1，那么无论谁抢到了 1，另一个都必须等待它结束释放资源才可以继续执行，不会发生死锁；否则，若两个进程并发，则 A 先获得了 1 而 B 先获得了 2 或者 3，会出现死锁. 保证不会出现死锁的情况占了 1/3.

2 作业 8

1. 可以认为 x 是 P2 独占的，y 的写操作是 P1 独占的. P2 对 y 的读操作一定发生在 P1 执行 signal(S1) 之后，wait(S2) 返回之前，因此读到的 y 的值一定是 3，从而 x 最后的值是 5；P1 对 z 的写操作没有确定的顺序，z 最终可能是 4（P1 在后）或者 9（P2 在后）；z 为 4 的情况是 P1 执行 z=y+1 时 P2 已经退出，最终 y 为 7；z 为 9 的情况还有两种可能，y=z+y 发生在 z=x+z 之前或者之后。最终 (x,y,z) 的可能取值为 (5, 7, 4), (5, 7, 9), (5, 12, 9).

2. 这是单生产者多消费者的缓冲区问题. 设有信号量 S，初值为 0；m 是 mutex. 顾客等叫号或者等取号都是等，因此可以直接实现为无界的缓冲区，也可以在顾客数量到一定程度的时候直接关闭取号服务。

```

int total, current;

customer_service() {
    lock(m);
    total += 1;
    signal(S);
    release(m);
    // 等待叫号应该实现为用户注册callback
}

teller_service() {
```

```
    wait(S);
    lock(m);
    current += 1;
    release(m);
    叫号(self, current);
    // 柜台服务
}
```

3.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include <unistd.h>

struct Monitor {
    int buffer[16];
    int result, index;
    pthread_mutex_t lock;
};
struct Monitor monitor;

unsigned long long diff(struct timespec start, struct timespec end) {
    return (end.tv_sec - start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec);
}

int monitor_get_task(struct Monitor *self, int *a, int *b) {
    pthread_mutex_lock(&(self->lock));
    if (self->index >= 16) {
        pthread_mutex_unlock(&(self->lock));
        return -1;
    }
    *a = self->buffer[self->index];
    *b = self->buffer[self->index + 1];
    printf("get task: at %d numbers %d, %d\n", self->index, *a, *b);
    self->index += 2;
    pthread_mutex_unlock(&(self->lock));
    return 0;
}

void monitor_put_result(struct Monitor *self, int result) {
    pthread_mutex_lock(&(self->lock));
    printf("put result: %d\n", result);
    struct timespec wait_time;
    wait_time.tv_sec = 0;
    wait_time.tv_nsec = 1000000L * (rand() % 10 + 1);
    nanosleep(&wait_time, NULL);
    self->result += result;
    pthread_mutex_unlock(&(self->lock));
}

void monitor_init(struct Monitor *self) {
    self->index = 0;
    self->result = 0;
```

```
    pthread_mutex_init(&(self->lock), NULL);
    for (int i = 0; i < 16; i++) {
        self->buffer[i] = rand() % 20;
    }
}

void *thread_main(void *place_holder) {
    int a, b;
    while (1) {
        if (monitor_get_task(&monitor, &a, &b) != 0) {
            break;
        }
        struct timespec wait_time;
        wait_time.tv_sec = 0;
        wait_time.tv_nsec = 1000000L * (rand() % 10 + 1);
        nanosleep(&wait_time, NULL);
        monitor_put_result(&monitor, a + b);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t workers[8];
    monitor_init(&monitor);
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    for (int i = 0; i < 8; i++) {
        pthread_create(&(workers[i]), NULL, thread_main, NULL);
    }
    for (int i = 0; i < 8; i++) {
        pthread_join(workers[i], NULL);
    }
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("result: %d, in (ns)%llu\n", monitor.result, diff(start_time, end_time));
    return 0;
}
```

结果：

```
get task: at 0 numbers 7, 9
get task: at 2 numbers 13, 18
get task: at 4 numbers 10, 12
get task: at 6 numbers 4, 18
get task: at 8 numbers 3, 9
get task: at 10 numbers 0, 5
get task: at 12 numbers 12, 2
get task: at 14 numbers 7, 3
put result: 22
put result: 22
put result: 12
put result: 16
put result: 31
put result: 5
put result: 10
put result: 14
result: 132, in (ns)61999000
```