2015/12/7

システムプログラミング演習5

学籍番号：201420694
氏名：星 遼平

1 mutex

mutex を利用して，銀行口座に関する入金，引き出し，残高確認のライブラリを作成し，動作の確認を行いなさい．動作確認にあたり，入金するスレッドを 3 つ，引き出しを行うスレッドを 3 つ作成し，それらのスレッドが同時に実行されても預金額が正しいことの確認を行いなさい．

プログラム

```
#include <stdio.h>
#include <pthread.h>
#include <time.h>

int account_balance = 1000;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

// NOTE: 現在残高を出力
void *check_balance() {
  printf("-> あなたの現在の残高 ￥%d \n", account_balance);
}

// NOTE: 引数の分預金する
void *deposit(void *arg) {
  int *money = (int *)arg;

  pthread_mutex_lock(&lock);

  account_balance += *money;
  printf("預金 ￥%d \n", *money);
  check_balance();

  pthread_mutex_unlock(&lock);
}
```

プログラム (続き)

```
// NOTE: 引数の分払戻する . もし預金残高が足りない場合は全額払戻する
void *withdraw(void *arg) {
  int *money = (int *)arg;
  int returning_money;

  pthread_mutex_lock(&lock);
  if (*money <= account_balance) {
    account_balance -= *money;
  } else {
    returning_money = account_balance;
    account_balance = 0;
  }
  printf("払戻 ￥%d \n", returning_money);
  check_balance();
  pthread_mutex_unlock(&lock);
}

int main() {
  pthread_t tid1, tid2, tid3, tid4, tid5, tid6;
  int arg1, arg2, arg3, arg4, arg5, arg6;

  srand((unsigned)time(NULL));

  arg1 = (rand() % 10) * 1000;
  arg2 = (rand() % 10) * 1000;
  arg3 = (rand() % 10) * 1000;
  arg4 = (rand() % 10) * 1000;
  arg5 = (rand() % 10) * 1000;
  arg6 = (rand() % 10) * 1000;

  pthread_create(&tid1, NULL, deposit, (void *)(&arg1));
  pthread_create(&tid2, NULL, withdraw, (void *)(&arg2));
  pthread_create(&tid3, NULL, deposit, (void *)(&arg3));
  pthread_create(&tid4, NULL, withdraw, (void *)(&arg4));
  pthread_create(&tid5, NULL, deposit, (void *)(&arg5));
  pthread_create(&tid6, NULL, withdraw, (void *)(&arg6));
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  pthread_join(tid3, NULL);
  pthread_join(tid4, NULL);
  pthread_join(tid5, NULL);
  pthread_join(tid6, NULL);
}
```

実行結果

```
$ ./a.out
払戻 ￥1000
-> あなたの現在の残高 ￥0
預金 ￥6000
-> あなたの現在の残高 ￥6000
払戻 ￥0
-> あなたの現在の残高 ￥1000
払戻 ￥1000
-> あなたの現在の残高 ￥0
預金 ￥7000
-> あなたの現在の残高 ￥7000
預金 ￥1000
-> あなたの現在の残高 ￥8000
```

2　スレッド版巡回バッファ

スレッド版巡回バッファを複数スレッドで利用し，動作の確認を行いなさい．動作確認にあたり，例えば 30 エントリを追加するスレッドを 2 つ，10 エントリを取り出すスレッドを 6 つ作成し，それらのスレッドが同時に実行されても正しく動作することの確認を行いなさい．適宜 printf(3) などで状態を出力させるようにするとよい．

プログラム

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

#define QSIZE 10

int put_num = 1;
int get_num = 1;

typedef struct {
  pthread_mutex_t buf_lock;
  int start;
  int num_full;
  pthread_cond_t notfull;
  pthread_cond_t notempty;
  void *data[QSIZE];
} circ_buf_t;

void print_data(circ_buf_t *cbp) {
  int i;

  for (i = 0; i < QSIZE; i++) {
    printf("data[%d] -> %s\n", i, (char *)(cbp->data[i]));
  }
}
```

プログラム (続き 1)

```
// NOTE: エントリを追加する関数
void put_cb_data(circ_buf_t *cbp, void *data) {
  pthread_mutex_lock(&cbp->buf_lock);

  if (put_num == 60)
    printf(" %2d 回目 追加 <- 追加終了 \n", put_num);
  else
    printf(" %2d 回目 追加 \n", put_num);
  put_num++;
  while (QSIZE <= cbp->num_full) {
    pthread_cond_wait(&cbp->notfull, &cbp->buf_lock);
    if ( cbp->num_full == QSIZE ) {
      printf("\n エントリがいっぱいのため取り出し待ち状態\n");
      printf("現在の巡回バッファ\n");
      printf("num_full -> %d\n", cbp->num_full);
      print_data(cbp);
      printf("\n");
    }
  }
  cbp->data[(cbp->start + cbp->num_full) % QSIZE] = data;
  cbp->num_full++;

  pthread_cond_signal(&cbp->notempty);
  pthread_mutex_unlock(&cbp->buf_lock);
}
```

プログラム (続き 2)

```
// NOTE: エントリを取り出す関数
void *get_cb_data(circ_buf_t *cbp) {
  void *data;

  if (get_num == 60)
    printf(" %2d 回目 取り出し <- 取り出し終了 \n", get_num);
  else
    printf(" %2d 回目 取り出し \n", get_num);
  get_num++;
  pthread_mutex_lock(&cbp->buf_lock);
  while (cbp->num_full <= 0) {
    pthread_cond_wait(&cbp->notempty, &cbp->buf_lock);
    if (cbp->num_full == 0) {
      printf("\n エントリが空のため追加待ち状態\n");
      printf("現在の巡回バッファ\n");
      printf("num_full -> %d\n", cbp->num_full);
      print_data(cbp);
      printf("\n");
    }
  }
  data = cbp->data[cbp->start];
  cbp->data[cbp->start] = NULL;
  cbp->start = (cbp->start + 1) % QSIZE;
  cbp->num_full--;

  pthread_cond_signal(&cbp->notfull);
  pthread_mutex_unlock(&cbp->buf_lock);
  return data;
}

// NOTE: 30 エントリ追加する関数
void *put_thirty_entries(void *arg) {
  char *entry = "Hello, World!!";
  int i;

  for (i = 0; i < 30; i++) {
    put_cb_data((circ_buf_t *)arg, (void *)entry);
  }
}
```

プログラム (続き 3)

```
// NOTE: 10 エントリ取り出す関数
void *get_ten_entries(void *arg) {
  char *entry;
  int i;

  for (i = 0; i < 10; i++) {
    entry = (char *)get_cb_data((circ_buf_t *)arg);
  }
}

int main() {
  circ_buf_t cbp = {
    PTHREAD_MUTEX_INITIALIZER, 0, 0,
    PTHREAD_COND_INITIALIZER, PTHREAD_COND_INITIALIZER, NULL
  };
  // NOTE: 30 エントリ追加するスレッド 2 つ
  pthread_t put_thirty_entries1, put_thirty_entries2;
  // NOTE: 10 エントリ取り出すスレッドを 6 つ
  pthread_t get_ten_entries1, get_ten_entries2, get_ten_entries3,
            get_ten_entries4, get_ten_entries5, get_ten_entries6;

  pthread_create(&get_ten_entries1, NULL, get_ten_entries, (void *)(&cbp));
  pthread_create(&put_thirty_entries1, NULL, put_thirty_entries, (void *)(&cbp));
  pthread_create(&get_ten_entries2, NULL, get_ten_entries, (void *)(&cbp));
  pthread_create(&put_thirty_entries2, NULL, put_thirty_entries, (void *)(&cbp));
  pthread_create(&get_ten_entries3, NULL, get_ten_entries, (void *)(&cbp));
  pthread_create(&get_ten_entries4, NULL, get_ten_entries, (void *)(&cbp));
  pthread_create(&get_ten_entries5, NULL, get_ten_entries, (void *)(&cbp));
  pthread_create(&get_ten_entries6, NULL, get_ten_entries, (void *)(&cbp));

  pthread_join(get_ten_entries1, NULL);
  pthread_join(put_thirty_entries1, NULL);
  pthread_join(get_ten_entries2, NULL);
  pthread_join(put_thirty_entries2, NULL);
  pthread_join(get_ten_entries3, NULL);
  pthread_join(get_ten_entries4, NULL);
  pthread_join(get_ten_entries5, NULL);
  pthread_join(get_ten_entries6, NULL);

  printf("\n 処理後の巡回バッファ\n");
  printf("num_full -> %d\n", cbp.num_full);
  print_data(&cbp);
}
```

実行結果

```
 $ ./a.out
 1 回目 取り出し
 2 回目 取り出し
 1 回目 追加
 3 回目 取り出し
 2 回目 追加
 4 回目 取り出し
       ・
(長いので省略)
       ・
19 回目 取り出し
20 回目 取り出し
13 回目 取り出し
18 回目 追加
22 回目 取り出し
22 回目 取り出し

エントリが空のため追加待ち状態
現在の巡回バッファ
num_full -> 0
data[0] -> (null)
data[1] -> (null)
data[2] -> (null)
data[3] -> (null)
data[4] -> (null)
data[5] -> (null)
data[6] -> (null)
data[7] -> (null)
data[8] -> (null)
data[9] -> (null)

 19 回目 追加
 20 回目 追加
       ・
(長いので省略)
       ・
 22 回目 追加
 24 回目 取り出し
 31 回目 追加
```

実行結果 (続き 1)

```
エントリがいっぱいのため取り出し待ち状態
現在の巡回バッファ
num_full -> 10
data[0] -> Hello, World!!
data[1] -> Hello, World!!
data[2] -> Hello, World!!
data[3] -> Hello, World!!
data[4] -> Hello, World!!
data[5] -> Hello, World!!
data[6] -> Hello, World!!
data[7] -> Hello, World!!
data[8] -> Hello, World!!
data[9] -> Hello, World!!

 26 回目 取り出し
 27 回目 取り出し
 28 回目 取り出し
 27 回目 取り出し
 26 回目 取り出し
        ・
(長いので省略)
        ・
 58 回目 追加
 52 回目 取り出し
 52 回目 取り出し
 59 回目 追加
 60 回目 追加 <- 追加終了
 54 回目 取り出し
 55 回目 取り出し
 54 回目 取り出し
 57 回目 取り出し
 58 回目 取り出し
 59 回目 取り出し
 60 回目 取り出し <- 取り出し終了
```

実行結果 (続き 2)

```
処理後の巡回バッファ
num_full -> 0
data[0] -> (null)
data[1] -> (null)
data[2] -> (null)
data[3] -> (null)
data[4] -> (null)
data[5] -> (null)
data[6] -> (null)
data[7] -> (null)
data[8] -> (null)
data[9] -> (null)
```

3 考察

pthread_mutex_unlock のタイミングによって標準出力順序が変わってくることがわかった．課題 1 にとりかかったときに，預金処理や払戻処理を実行した後に pthread_mutex_unlock をしたときに処理後にログ出力のため printf していたが，そうすると違うスレッドの処理も実行されてしまい，意図した出力がされてなかった．今回のログ出力に関しても pthread_mutex_unlock する前に記述するようにした．

pthread_create 関数の引数は void * 型である必要がある．課題 2 では 2 つの引数を受け取りたい put_cb_data があるが，これを pthread_create 関数の引数には渡せない．そこで，実装方法として考えたものが 2 つある．

1 つ目は引数に構造体とすることだ．今回の課題では circ_buf_t * 型と バッファに入れたいデータを渡したいので、この 2 つを保持することができる構造体を作成する．そうすることで実際に pthread_create の引数には構造体を void * 型にキャスト変換したものを渡すことができる．受け取り側では作成した構造体に再度キャスト変換することで実際に渡したい 2 つの引数を取り出すことができる．

2 つ目はスレッドとして実行したい関数を実行する関数を作成することだ．実行したい関数は put_cb_data と get_cb_data という 2 つの関数だ．pthread_create 関数に渡すためには返り値と引数が void * 型である必要がある．そこで，返り値と引数が void * 型の関数を作成し，その関数内で put_cb_data や get_cb_data を実行する．実際にスレッドを生成する際は新たに作成した関数を渡す．その中で実行したい関数が実行されるため結果的に実行したい関数がスレッドとして実行されるのと同じになる．

4 授業の感想

最近 Google が開発を行っている Go 言語の影響かわかりませんが，マルチスレッド (Go 言語だと Goroutine) を意識したプログラムを書くあるいは設計する機会が増えてきたと実感していました．また，PC の小型化が進むにつれて，マルチコア型のプロセッサの PC が主流になっています．私が使っているラップトップもマルチコア型のプロセッサを搭載していますが，これからのプログラマは今後よりマルチコアを活かすプログラムの設計および作成する能力が必要になると思います．今回の授業を通してスレッドに関する基礎的な知識を学ぶことが出来たと思います．

また，授業中実際に手元の PC でプログラムを書いて実行したことで，講義の内容の理解がより深まりました．今後の授業でもそのような機会を設けていただきたいと感じました．