『Linux による並行プログラミング入門』
正誤表

(1)
p.98，8.5，上から 11 行目
修正前：mutex にロックが掛かっていると
修正後：ロックが掛かっていない指定の mutex にロックが掛かると

(2)
p.109，List 9-3，下から 9 行目
修正前：pthread_cond_wait
修正後：pthread_cond_timedwait

(3)
p111，10.1，上から 2 行目
修正前：タスクの実行状態を考えると
修正後：シングルコアプロセッサ上でのタスクの実行状態を考えると

(4)
p111，10.2，上から 1 行目
修正前 : タスクをどのような順序で実行するか決定することを**タスクスケジューリング**という．タスクスケジューリングのために，…
修正後 : タスクをどのような順序で実行するか決定することを**タスクスケジューリング**という．本章では，簡単のため**シングルコアプロセッサ**上でのタスクスケジューリングについて扱う．タスクスケジューリングンのために，…

(5)
p.116，10.4，上から 2 行目
修正前：複数のスレッドの実行順を
修正後：シングルコアプロセッサ上で動作する複数のスレッドの実行順を

(6)
p.117，最後の段落の後（List 10-2 の前），文章の追加
　マルチコアプロセッサ上では複数のプロセス，スレッドが同時に実行されるため，優先度の高いスレッドと低いスレッドが同時に実行される．それではタスクスケジューリングの設定の効果を確認できない．List 10-2 のプログラムでは，sched_setaffinity()を呼ぶことによってプロセス内のスレッドを特定のコア上でのみ実行するように設定している．

(7)

p.117，List 10-2 のプログラムを以下のものに差替え

（黄色で示した個所が加筆，修正箇所です）

```
#define _GNU_SOURCE
#include <sched.h>

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <errno.h>
pthread_mutex_t mtx;
pthread_mutex_t msg_mtx;
pthread_cond_t    cnd;
int x=0;
char msg_buf[32];
char *msg_cur=msg_buf;
// スケジューリングパラメータを設定する関数
static int SetPriority( pthread_t pth, int policy, int priority )
{
   struct sched_param sched;
   int err;
   sched.sched_priority = priority;
   if ( (err=pthread_setschedparam( pth, policy, &sched )) != 0 )
   {
      switch(err)
      {
         case EINVAL: perror("EINVAL");break;
         case EPERM: perror("EPERM");break;
         case ESRCH: perror("ESRCH");break;
         case EFAULT: perror("EFALUT");break;
         default: break;
      }
   }
   pthread_getschedparam( pth, &policy, &sched );
   printf("sched:%d", sched.sched_priority );
```

```
    switch( policy )
    {
        case SCHED_FIFO    : printf("f¥n");break;
        case SCHED_RR      : printf("r¥n");break;
        case SCHED_OTHER : printf("o¥n");break;
        default : break;
    }
    return( 0 );
}
// CPU 時間を消費する関数
void spend_cputime(int second)
{
    struct timespec ts;
    while(1) //  指定の時間になるまでループ
    {
        //  スレッドごとの CPU 時間を基準にする
        clock_gettime(CLOCK_THREAD_CPUTIME_ID,&ts);
        if ( ts.tv_sec >= second )
            break;
    }
}

void Thread(void *ch) //  スレッドで実行される関数
{
    int i;
    char id = (char)ch; // msg_buf に書き込む文字

    pthread_mutex_lock(&mtx);
    pthread_mutex_unlock(&mtx);

    pthread_mutex_lock(&mtx);
    while (x<1)
    {
        pthread_cond_wait(&cnd,&mtx); //  合図を待つ
        //  四つのスレッドで同期を取る
    }
    pthread_mutex_unlock(&mtx);
```

```
    for (i=0;i<5;++i)
    {
        spend_cputime(i+1); //  スレッド時間で 1 秒間待つ

        pthread_mutex_lock(&msg_mtx);
        *msg_cur = id; //  配列に文字を書き込む
        msg_cur++;
        pthread_mutex_unlock(&msg_mtx);
    }
}

int main(int argc, char *argv[])
{
    pthread_t th[4];
    int y;

    //  特定のコアでのみタスクを実行できるように設定する
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(0, &mask);

    if (sched_setaffinity(0, sizeof(mask), &mask) == -1) {
        perror("Failed to set CPU affinity");
        exit(1);
    }

    // [1]  ロックの初期化
    pthread_mutex_init(&mtx,NULL);
    pthread_mutex_init(&msg_mtx,NULL);

    // [2]  条件変数の初期化
    pthread_cond_init(&cnd,NULL);

    //  スレッド生成中、スケジューリング設定中に処理が進まないようにロックを掛ける
    pthread_mutex_lock(&mtx);

    // [3]  スレッドの生成
    pthread_create( th,     NULL, (void *(*)(void *))Thread, (void*)'a' );
```

```
    pthread_create( th+1, NULL, (void *(*)(void *))Thread, (void*)'b' );
    pthread_create( th+2, NULL, (void *(*)(void *))Thread, (void*)'c' );
    pthread_create( th+3, NULL, (void *(*)(void *))Thread, (void*)'d' );

    // [4] スレッドのスケジューリングポリシーと優先度を設定する
    SetPriority( th[0], SCHED_RR, 1 );
    SetPriority( th[1], SCHED_RR, 1 );
    SetPriority( th[2], SCHED_OTHER, 0 );
    SetPriority( th[3], SCHED_FIFO, 2 );

    pthread_mutex_unlock(&mtx);

    // [5] スレッドの処理開始の同期をとるため、条件変数で合図を送る
    pthread_mutex_lock(&mtx);
    x=1;
    pthread_mutex_unlock(&mtx);
    pthread_cond_broadcast(&cnd);

    // [6] スレッドの終了を待つ
    pthread_join(th[0],(void **)&y);
    pthread_join(th[1],(void **)&y);
    pthread_join(th[2],(void **)&y);
    pthread_join(th[3],(void **)&y);

    // [7] スレッドに書き込まれた msg を表示する
    *msg_cur = '¥0';
    printf("msg:%s¥n",msg_buf);

    return EXIT_SUCCESS;
}
```

(8)

p.120，問題 10-2，1 行目

修正前：スケジューリングパラメータの設定を変更し

修正後：スレッドの生成順序とスケジューリングパラメータの設定を変更し

(9)

p.160，問題 10-2，2 行目

修正前：スレッド B,D,A,C の優先度を高い順になるように設定すればよい.

修正後：スレッド B,D,A,C の順に生成し，同じ優先度に設定すればよい.

(10)

p.160，問題 10-2，3 行目

修正前：例えば，スレッド B の優先度を 4，D を 3，A を 2，C を 1 に設定する.

修正後：例えば，スレッド B,D,A,C の優先度をすべて 1 に設定する.

(11)

p.160，問題 10-2，4 行目

削除：List 10-2 のプログラムで書き換える箇所は，…箇所のみである.

(12)

p.160，問題 10-2，プログラムを以下に差替え

（黄色で示した個所が加筆，修正箇所です）

```
#define _GNU_SOURCE
#include <sched.h>

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <errno.h>
pthread_mutex_t mtx;
pthread_mutex_t msg_mtx;
pthread_cond_t    cnd;
int x=0;
char msg_buf[32];
char *msg_cur=msg_buf;
// スケジューリングパラメータを設定する関数
static int SetPriority( pthread_t pth, int policy, int priority )
{
   struct sched_param sched;
   int err;
```

```
    sched.sched_priority = priority;
    if ( (err=pthread_setschedparam( pth, policy, &sched )) != 0 )
    {
        switch(err)
        {
            case EINVAL: perror("EINVAL");break;
            case EPERM: perror("EPERM");break;
            case ESRCH: perror("ESRCH");break;
            case EFAULT: perror("EFALUT");break;
            default: break;
        }
    }
    pthread_getschedparam( pth, &policy, &sched );
    printf("sched:%d", sched.sched_priority );
    switch( policy )
    {
        case SCHED_FIFO    : printf("f¥n");break;
        case SCHED_RR       : printf("r¥n");break;
        case SCHED_OTHER : printf("o¥n");break;
        default : break;
    }
    return( 0 );
}
// CPU 時間を消費する関数
void spend_cputime(int second)
{
    struct timespec ts;
    while(1) //  指定の時間になるまでループ
    {
        //  スレッドごとの CPU 時間を基準にする
        clock_gettime(CLOCK_THREAD_CPUTIME_ID,&ts);
        if ( ts.tv_sec >= second )
            break;
    }
}

void Thread(void *ch) //  スレッドで実行される関数
{
```

```
    int i;
    char id = (char)ch; // msg_buf に書き込む文字

    pthread_mutex_lock(&mtx);
    pthread_mutex_unlock(&mtx);

    pthread_mutex_lock(&mtx);
    while (x<1)
    {
        pthread_cond_wait(&cnd,&mtx); //  合図を待つ
        //  四つのスレッドで同期を取る
    }
    pthread_mutex_unlock(&mtx);

    for (i=0;i<5;++i)
    {
        spend_cputime(i+1); //  スレッド時間で 1 秒間待つ

        pthread_mutex_lock(&msg_mtx);
        *msg_cur = id; //  配列に文字を書き込む
        msg_cur++;
        pthread_mutex_unlock(&msg_mtx);
    }
}

int main(int argc, char *argv[])
{
    pthread_t th[4];
    int y;

    //  特定のコアでのみタスクを実行できるように設定する
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(0, &mask);

    if (sched_setaffinity(0, sizeof(mask), &mask) == -1) {
        perror("Failed to set CPU affinity");
        exit(1);
```

```
}

// [1] ロックの初期化
pthread_mutex_init(&mtx,NULL);
pthread_mutex_init(&msg_mtx,NULL);

// [2] 条件変数の初期化
pthread_cond_init(&cnd,NULL);

// スレッド生成中、スケジューリング設定中に処理が進まないようにロックを掛ける
pthread_mutex_lock(&mtx);

// [3] スレッドの生成
pthread_create( th,     NULL, (void *(*)(void *))Thread, (void*)'b' );
pthread_create( th+1, NULL, (void *(*)(void *))Thread, (void*)'d' );
pthread_create( th+2, NULL, (void *(*)(void *))Thread, (void*)'a' );
pthread_create( th+3, NULL, (void *(*)(void *))Thread, (void*)'c' );

// [4] スレッドのスケジューリングポリシーと優先度を設定する
SetPriority( th[0], SCHED_RR, 1 );
SetPriority( th[1], SCHED_RR, 1 );
SetPriority( th[2], SCHED_RR, 1 );
SetPriority( th[3], SCHED_RR, 1 );

pthread_mutex_unlock(&mtx);

// [5] スレッドの処理開始の同期をとるため、条件変数で合図を送る
pthread_mutex_lock(&mtx);
x=1;
pthread_mutex_unlock(&mtx);
pthread_cond_broadcast(&cnd);

// [6] スレッドの終了を待つ
pthread_join(th[0],(void **)&y);
pthread_join(th[1],(void **)&y);
pthread_join(th[2],(void **)&y);
pthread_join(th[3],(void **)&y);
```

```
    // [7] スレッドに書き込まれた msg を表示する
    *msg_cur = '¥0';
    printf("msg:%s¥n",msg_buf);

    return EXIT_SUCCESS;
}
```