システム開発工学

スレッドプログラミング

Concurrent Programming with threads

山田泰司

taiji@aihara.co.jp

株式会社あいはら 研究開発チーム

並行プログラミングとは

- マルチプロセッサ・マルチコア性能を活用したい
- 並列処理により結果を素早く得たい

並行・並列プログラミングの手法：

- VIS(Sparc 系), VMX, AltiVec(PowerPC 系)(通称: Velocity Engine), MMX, SSE(Intel 系) 等、CPU の SIMD 機構
- pthreads 等による軽量プロセス
- fork(Unix のシステムコール) 等によるマルチプロセス
- クライアントサーバモデル (RPC, etc)、P2P モデルなどの分散コンピューティング
- OpenMP, Grid Engine, Xgrid, Open MPI, MS MPI などの並列コンピューティング環境

pthreads - POSIX スレッドとは

POSIX(Portable Operating System Interface) - **プログラミング言語** C **のシステムコールインターフェースやそれに関連する** OS **環境の差異を共通化することで、移植性の高いアプリケーション開発を容易にすることを目的とした** IEEE **が策定する規格**

- Solaris, Linux, BSD **等の** Unix **系オペレーティングシステム**
- Windows XP **における** Services for UNIX **による** POSIX **サブシステム**
- Windows NT/2000/2003 R2/Vista **における** POSIX **サブシステム**
- Cygwin **もしくは** pthreads-win32, etc
- Symbian OS, TRON, OS-9 **などへの** POSIX **対応**

POSIX **スレッド**: POSIX **で規格化されているスレッド関連** API

- **スレッド生成、操作**
- **相互排他** (Mutal Exclusion: mutex) **ロック、条件変数、による同期**
- **リーダー/ライターロック、スピンロック、による同期**
- **スケジューリング、シグナル処理、ほか**

逐次実行プログラム

ex00sequential.c

```
#include <stdio.h>      /* printf, etc */
#include <stdlib.h>     /* RAND_MAX, etc */

struct result {
  int id;
  double x;
};
void set_result(struct result *r)
{
  int i, d;

  srand(r->id);
  r->x = 0;
  for (i=0; i<10000; i++) {
    d = 1 + (int)((double)(6 - 1 + 1)*rand()/(RAND_MAX+1.0));
    printf("id: %d, d: %d\n", r->id, d);
    r->x += d;
  } /* 一万試行のサイコロの目の総和 */
}
        :
```

敢えて、乱数の種を固定し、再現性のある疑似乱数を使っていることに留意

逐次実行プログラム（つづき）

ex00sequential.c

```
          :
int main(void)
{
  int i;
  struct result rs[2] = { { /* .id = */ 1, }, { /* .id = */ 2, }, };

  for (i=0; i<2; i++)
    set_result(&rs[i]);

  for (i=0; i<2; i++) /* 2 つの独立した「一万試行のサイコロの目の総和」を表示 */
    printf("id: %d, x: %g\n", rs[i].id, rs[i].x);
  return 0;
}
```

```
$ make ex00sequential
$ ./ex00sequential
        :
id: 1, x: 35098
id: 2, x: 34977
```

（例題１）fork vs. pthreads

先の逐次実行プログラム `ex00sequential.c` **を** fork **を使って並列化せよ。また、**pthreads **を使って並列化せよ。**

但し、最終的に得られる結果が変化してはならない。

fork による並行プログラム

ex01fork.c

```
        :
#include <string.h>     /* memcpy, etc */
#include <sys/stat.h>   /* S_IRUSR, etc */
#include <sys/types.h>  /* IPC_*, etc */
#include <sys/ipc.h>    /* shmget, etc */
#include <sys/shm.h>    /* shmget, etc */
#include <unistd.h>     /* fork, etc */
        :
int main(void)
{
  int i;
  struct result rs[2] = { { /* .id = */ 1, }, { /* .id = */ 2, }, }, *shrs;
  int shrsid;
  pid_t pids[2];
  int status[2];

  if (!((shrsid=shmget(IPC_PRIVATE, sizeof(rs), IPC_CREAT|S_IRUSR|S_IWUSR)) &&
        (shrs=(struct result *)shmat(shrsid, NULL, 0)) &&
        shrs != (void *)-1)) {
    perror("shmget, shmat");
    exit(1);
  } /* shmget, shmat で rs と同サイズの System V IPC 共有メモリを shrs へ確保する */
  memcpy(shrs, rs, sizeof(rs));
```

fork による並行プログラム（つづき）

ex01fork.c

```
          :
  for (i=0; i<2; i++) /* 2 つのプロセスで独立した「一万試行のサイコロの目の総和」を得る */
    if ((pids[i]=fork()) < 0) {
      perror("fork");
      exit(1);
    }
    else if (pids[i] == 0) { /* 子プロセスでの処理 */
      set_result(&shrs[i]);
      exit(0);
    }
  for (i=0; i<2; i++) /* 2 つのプロセスが終了するのを待つ */
    waitpid(pids[i], &status[i], 0);
  memcpy(rs, shrs, sizeof(rs));
  shmctl(shrsid, IPC_RMID, 0); /* 共有メモリの廃棄 */

  for (i=0; i<2; i++)
    printf("id: %d, x: %g\n", rs[i].id, rs[i].x);
  return 0;
}
```

pthreads による並行プログラム

ex02pthreads.c

```
          :
#include <pthread.h>    /* pthread_create, etc */
          :
int main(void)
{
  int i;
  struct result rs[2] = { { /* .id = */ 1, }, { /* .id = */ 2, }, };
  pthread_t thr[2];

  for (i=0; i<2; i++) /* 2 つのスレッドで独立した「一万試行のサイコロの目の総和」を得る */
    pthread_create(&thr[i], NULL, (void *)&set_result, &rs[i]);
  for (i=0; i<2; i++)
    pthread_join(thr[i], NULL);

  for (i=0; i<2; i++)
    printf("id: %d, x: %g\n", rs[i].id, rs[i].x);
  return 0;
}
```

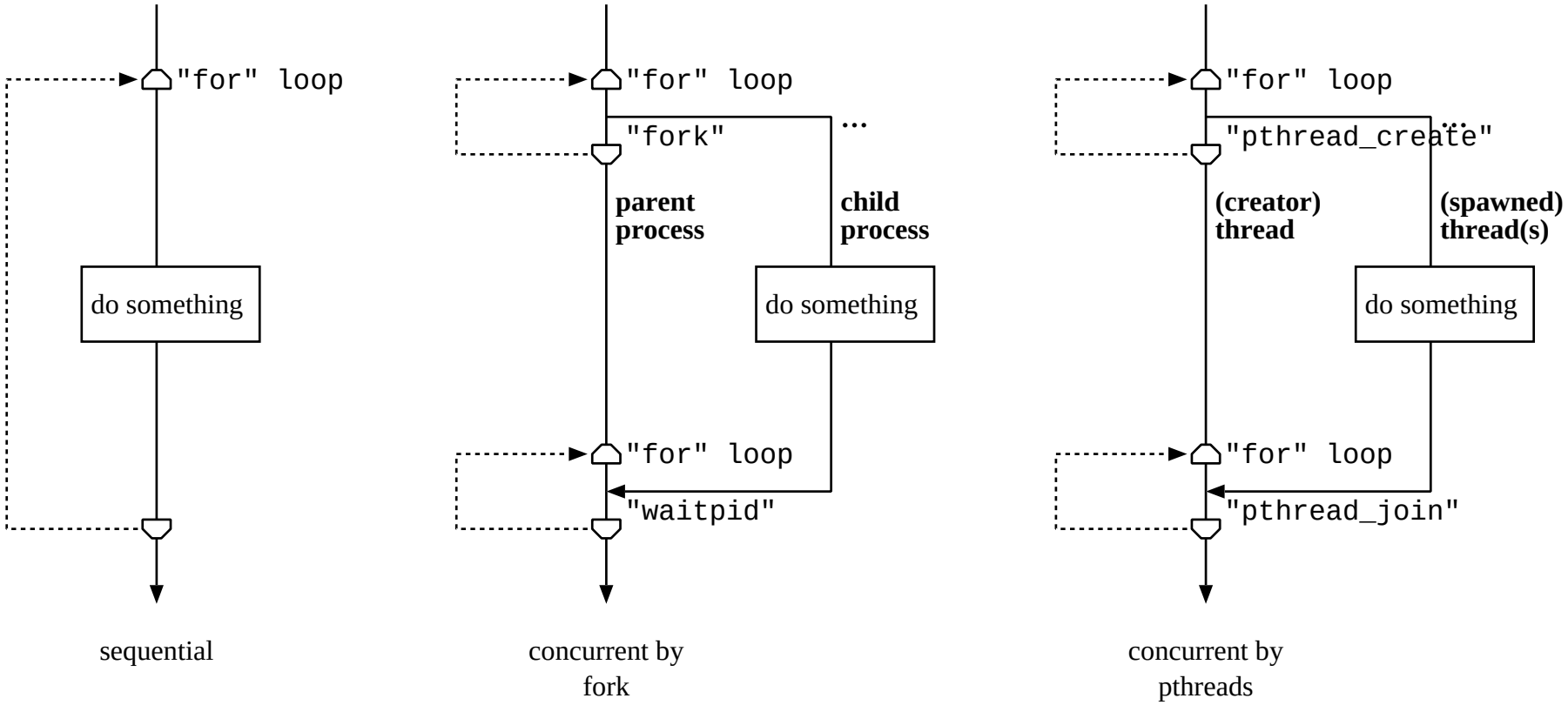
pthreads による並行プログラム（修正）

ex03pthreads.c

```
          :
void set_result(struct result *r)
{
  int i, d;
  unsigned seed = r->id;

  r->x = 0;
  for (i=0; i<10000; i++) {
    d = 1 + (int)((double)(6 - 1 + 1)*rand_r(&seed)/(RAND_MAX+1.0));
    printf("id: %d, d: %d\n", r->id, d);
    r->x += d;
  }
}
          :
```

2 つの新しいスレッドは呼び出しスレッドと並行して動作、すべてのスレッドはこのプロセスの資源 (大域変数、静的変数) を共有する (つまり、rand 内の静的変数も共有)。
よって、rand の代わりに、その再入可能版である rand_r を使うことで、悪影響を及ぼす資源の共有を防ぐ事になる。

逐次, fork, pthread



sequential

concurrent by fork

concurrent by pthreads

pthreads 事始め

pthreads **事始めの要点**

- **新しいスレッドを生成** - `pthread_create`
- **スレッドを合流** - `pthread_join`

スレッド内で `rand` **等の** MT-Unsafe **な関数を使ってはならない！**

```
$ find /usr/share/man -iname '*_r.*'
        :
/usr/share/man/man3/rand_r.3
/usr/share/man/man3/readdir_r.3
/usr/share/man/man3/strtok_r.3
        :
```

必ずマニュアルを見て再入可能 (reentrant) **つまり** MT-Safe **な関数を使う。**

```
$ man gethostbyname
```

そして、再入可能ではない関数を自分で書いてはならない (関数内で変更する独立した静的なデータはスレッドルーチンに渡される引数に閉じ込める事)。

再入可能ではない関数の例

```
char *strtok(char *str, const char *delim)
{
  static char *p;
  if (!delim) p = str;
        :
  return p;
}
```

```
struct tm *localtime(const time_t *clock)
{
  static struct tm;
        :
  return &tm;
}
```

```
char *inet_ntoa(struct in_addr in)
{
  static char a[16];
        :
  return a;
}
```

（例題２）行列の積の並列化

行列の積 `c[M][L]=a[M][N]*b[N][L]` **を求める以下のようなルーチンを並列化せよ。**
matn00.c

```
        :
double **mat(int M, int N)
{
  int i;
  double **m;

  m = (double **)malloc(sizeof(double *)*M);
  for (i=0; i<M; i++)
    m[i] = (double *)malloc(sizeof(double)*N);
  return m;
}
void mat_mult_mat(int M, int N, int L, double **a, double **b, double **c)
{
  int i, j, k;

  for (i=0; i<M; i++)
    for (j=0; j<L; j++)
      for (c[i][j] = 0, k=0; k<N; k++)
        c[i][j] += a[i][k]*b[k][j];
}
        :
  double **matx = mat(N, N), **maty = mat(N, N), **matz = mat(N, N);

  unit_mat(N, N, matx);
  unit_mat(N, N, maty);
  mat_mult_mat(N, N, N, matx, maty, matz);
        :
```

（例題２）の解答例01

matn01.c

```
          :
#include <pthread.h>   /* pthread_create, etc */
#include <errno.h>     /* strerror, etc */
          :
struct thr_mat_mult_mat_arg {
  int N, i, j;
  double **a, **b, **c;
};
void *thr_mat_mult_mat(void *arg)
{
  int k;
  struct thr_mat_mult_mat_arg *a = (struct thr_mat_mult_mat_arg *)arg;

  for (a->c[a->i][a->j] = 0, k=0; k<a->N; k++)
    a->c[a->i][a->j] += a->a[a->i][k]*a->b[k][a->j];
  return NULL;
}
          :
```

（例題２）の解答例01（つづき）

matn01.c

```
          :
void mthr_mat_mult_mat(int M, int N, int L, double **a, double **b, double **c)
{
  int i, j, rv;
  pthread_t thrs[M*L];
  struct thr_mat_mult_mat_arg args[M*L];

  for (i=0; i<M; i++)
    for (j=0; j<L; j++) {
      args[L*i+j].N = N; args[L*i+j].i = i; args[L*i+j].j = j;
      args[L*i+j].a = a; args[L*i+j].b = b; args[L*i+j].c = c;
      if ((rv = pthread_create(&thrs[L*i+j], NULL, thr_mat_mult_mat, &args[L*i+j]))) {
        fprintf(stderr, "pthread_create: %s\n", strerror(rv)); exit(-1);
      }
    }
  for (i=0; i<M; i++)
    for (j=0; j<L; j++)
      pthread_join(thrs[L*i+j], NULL);
}
          :
  mthr_mat_mult_mat(N, N, N, matx, maty, matz);
          :
```

（例題２）の解答例02

matn02.c

```
        :
#define min(a, b) (((a)<(b))?(a):(b))
#define max(a, b) (((a)>(b))?(a):(b))
static int m_thr = 4;

void mthr_mat_mult_mat(int M, int N, int L, double **a, double **b, double **c)
{
  int i, j, k, n_thr = 0, rv;
  pthread_t thrs[min(M*L, m_thr)];
  struct thr_mat_mult_mat_arg args[min(M*L, m_thr)];

  for (i=0; i<M; i++)
    for (j=0; j<L; j++) {
      args[n_thr].N = N; args[n_thr].i = i; args[n_thr].j = j;
      args[n_thr].a = a; args[n_thr].b = b; args[n_thr].c = c;
      if ((rv = pthread_create(&thrs[n_thr], NULL, thr_mat_mult_mat, &args[n_thr]))) {
        fprintf(stderr, "pthread_create: %s\n", strerror(rv)); exit(-1);
      }
      n_thr++;
      if (n_thr == m_thr || (i==M-1 && j==L-1)) {
        for (k=0; k<n_thr; k++)
          pthread_join(thrs[k], NULL);
        n_thr = 0;
      }
    }
}
        :
```

（例題２）の解答例11

matn11.c

```
          :
#include <pthread.h>   /* pthread_create, etc */
#include <errno.h>     /* strerror, etc */
          :
struct thr_mat_mult_mat_arg {
  int N, L, i;
  double **a, **b, **c;
};
void *thr_mat_mult_mat(void *arg)
{
  int j, k;
  struct thr_mat_mult_mat_arg *a = (struct thr_mat_mult_mat_arg *)arg;

  for (j=0; j<a->L; j++)
    for (a->c[a->i][j] = 0, k=0; k<a->N; k++)
      a->c[a->i][j] += a->a[a->i][k]*a->b[k][j];
  return NULL;
}
          :
```

（例題２）の解答例11（つづき）

matn11.c

```
          :
void mthr_mat_mult_mat(int M, int N, int L, double **a, double **b, double **c)
{
  int i, rv;
  pthread_t thrs[M];
  struct thr_mat_mult_mat_arg args[M];

  for (i=0; i<M; i++) {
    args[i].N = N; args[i].L = L; args[i].i = i;
    args[i].a = a; args[i].b = b; args[i].c = c;
    if ((rv = pthread_create(&thrs[i], NULL, thr_mat_mult_mat, &args[i]))) {
      fprintf(stderr, "pthread_create: %s\n", strerror(rv)); exit(-1);
    }
  }
  for (i=0; i<M; i++)
    pthread_join(thrs[i], NULL);
}
          :
```

（例題２）の解答例12

matn12.c

```
        :
#define min(a, b) (((a)<(b))?(a):(b))
#define max(a, b) (((a)>(b))?(a):(b))
static int m_thr = 4;

void mthr_mat_mult_mat(int M, int N, int L, double **a, double **b, double **c)
{
  int i, k, n_thr = 0, rv;
  pthread_t thrs[min(M, m_thr)];
  struct thr_mat_mult_mat_arg args[min(M, m_thr)];

  for (i=0; i<M; i++) {
    args[n_thr].N = N; args[n_thr].L = L; args[n_thr].i = i;
    args[n_thr].a = a; args[n_thr].b = b; args[n_thr].c = c;
    if ((rv = pthread_create(&thrs[n_thr], NULL, thr_mat_mult_mat, &args[n_thr]))) {
     fprintf(stderr, "pthread_create: %s\n", strerror(rv)); exit(-1);
    }
    n_thr++;
    if (n_thr == m_thr || i==M-1) {
     for (k=0; k<n_thr; k++)
       pthread_join(thrs[k], NULL);
     n_thr = 0;
    }
  }
}
        :
```

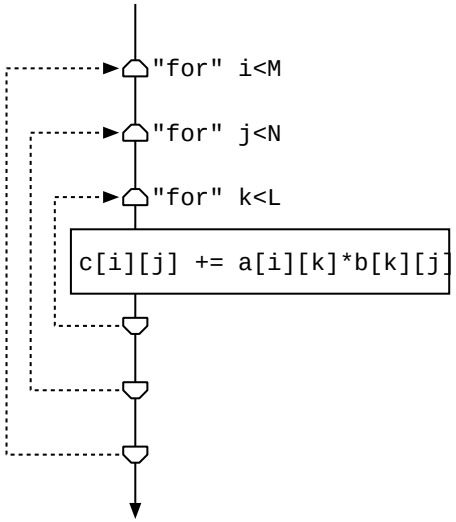
(考察) あまりスレッドの粒度が細か過ぎるとパフォーマンスの悪化に繋がる。スレッド使用の有無を選べるようにコーディングするのが吉。

（参考）AltiVecによる行列の積

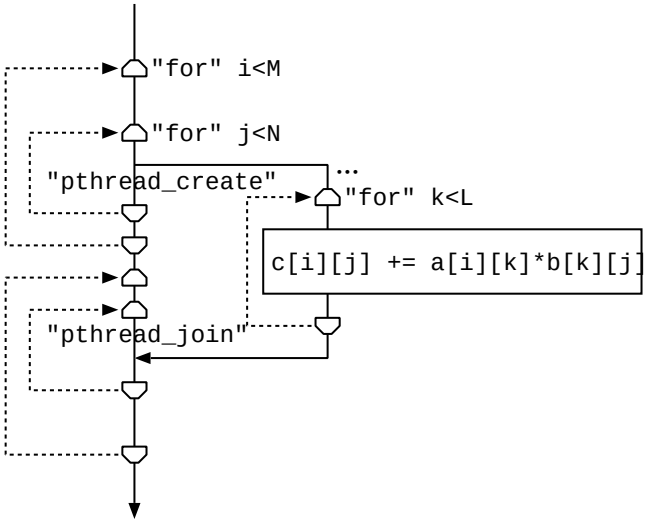
matn20.c

```
        :
#if defined(__ALTIVEC__)
#include <altivec.h>   /* __vector */
#endif
        :
void mat_mult_mat(int M, int N, int L, double **a, double **b, double **c)
{
  int i, j, k, l;

  for (i=0; i<M; i++)
    for (j=0; j<L; j++) {
      __vector float vc = (__vector float)(0, 0, 0, 0);

      for (c[i][j] = 0, k=l=0; l<N/4; k=(++l)*4) {
       __vector float
         va = (__vector float)(a[i][k+0], a[i][k+1], a[i][k+2], a[i][k+3]),
         vb = (__vector float)(b[k+0][j], b[k+1][j], b[k+2][j], b[k+3][j]);

       vc = vec_madd(va, vb, vc);
      }
      for (; k<N; k++)
       c[i][j] += a[i][k]*b[k][j];
      for (l=0; l<4; l++)
       c[i][j] += ((float *)&vc)[l];
    }
}
        :
```

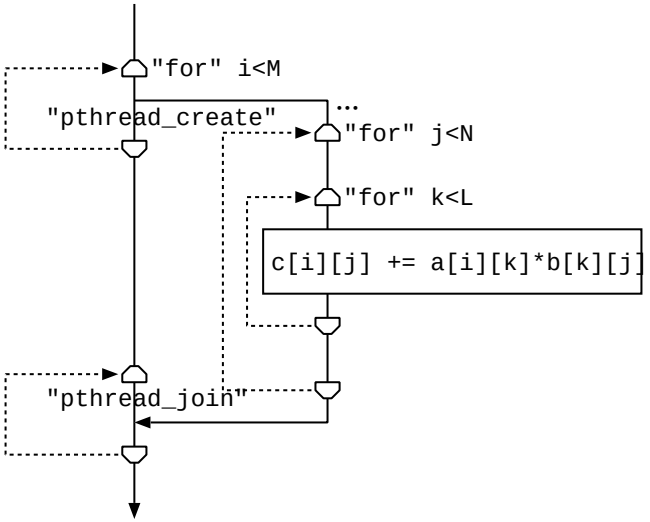
行列の積 with pthread



sequential
(mat00.c)



concurrent by
pthreads (mat01.c)



concurrent by
pthreads (mat10.c)

（例題３）複数のDNSBL問い合わせを並列化

IPv4 アドレス (例えば、127.0.0.2) において、そこが不正なアクセスを常習的に行なっているなどを、DNS サーバの機構を活用して情報提供してくれるサービスを DNSBL(DNS ベースのブラックリスト) という。例えば、cbl.abuseat.org, bl.spamcop.net, bl.spamcannibal.org, dnsbl-1.uceprotect.net, zen.spamhaus.org, などのサイトがある。利用方法は、IPv4 の 4 オクテットを逆順にドットで区切って文字列にし、DNSBL 提供元サイト名と結合したホスト名を、DNS キャッシュサーバに問い合わせ、見つかればそのホストはそこのブラックリストに登録されている事になる。つまり、コマンドラインで行なえば以下のようになる。

```
$ host 2.0.0.127.cbl.abuseat.org
2.0.0.127.cbl.abuseat.org has address 127.0.0.2
```

指定した IPv4 アドレスが、複数の DNSBL のうち少なくとも一つでも登録されているかを調べる C プログラムを書け。そして、それを pthreads で並列化せよ。

(注) 一度 DNS キャッシュサーバに問い合わせ結果が蓄積されると、そのキャッシュ期間が満了するまで、同じ問い合わせはすぐさま得られることに注意。

（例題３）の解答例、逐次実行

abmail/tools/abaddr.c

```
        :
const char *dnsbls[] = {
  NULL,
  "cbl.abuseat.org",
  "bl.spamcop.net",
        : /* 以下、他の DNSBL サイトを同様に列挙 */
};
int dnsbl_find(const char *ip)
{
  int rv = 0, i;
  unsigned int ip4[4];
  struct hostent *he;
  char name[2048];

  if (sscanf(ip, "%u.%u.%u.%u", &ip4[0], &ip4[1], &ip4[2], &ip4[3]) != 4)
    return rv; /* 不正な IPv4 アドレス表記 */
  for (i=1; i<sizeof(dnsbls)/sizeof(char *); i++) {
    snprintf(name, sizeof(name), "%u.%u.%u.%u.%s",
             ip4[3], ip4[2], ip4[1], ip4[0], dnsbls[i]);
    if ((he=gethostbyname(name))) {
      rv = i;
      break;
    }
  }
  return rv;
}
        :
```

（例題３）の解答例、並列化

abmail/tools/abaddr.c

```
        :
struct thr_dnsbl_find_arg {
  char name[2048];
  int i, *rv, *n;          /* スレッドのインデックス、返り値 (0 もしくは i)、スレッド数 (unused) */
  pthread_t *selves;       /* スレッド配列 (unused) */
  pthread_mutex_t *mutex; /* 相互排他変数へのポインタ */
};
        :
void *thr_dnsbl_find(void *arg)
{
  int rv = 0;
  struct thr_dnsbl_find_arg *a = (struct thr_dnsbl_find_arg *)arg;
  struct addrinfo hints, *res;

  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_INET; hints.ai_socktype = SOCK_STREAM;
  if ((rv = (getaddrinfo(a->name, NULL, &hints, &res) == 0)?a->i:0)) {
    freeaddrinfo(res);
    if (*a->rv == 0) {
      pthread_mutex_lock(a->mutex);   /* mutex をロック */
      *a->rv = rv;                    /* 返り値を格納 */
      pthread_mutex_unlock(a->mutex); /* mutex をロック解除 */
    }
  }
  return NULL;
}
        :
```

（例題３）の解答例、並列化（つづき）

abmail/tools/abaddr.c

```
        :
int dnsbl_find(const char *ip)
{
  int rv = 0 /* 共有したい返り値 */, i, n = 1;
  unsigned int ip4[4];
  pthread_t threads[sizeof(dnsbls)/sizeof(char *)];
  pthread_mutex_t mutex; /* 相互排他 (mutex) 変数 */
  struct thr_dnsbl_find_arg args[sizeof(dnsbls)/sizeof(char *)];

  if (sscanf(ip, "%u.%u.%u.%u", &ip4[0], &ip4[1], &ip4[2], &ip4[3]) != 4)
    return rv; /* 不正な IPv4 アドレス表記 */
  pthread_mutex_init(&mutex, NULL); /* 相互排他変数の初期化 */
  for (i=1; i<sizeof(dnsbls)/sizeof(char *) && rv == 0; i++) {
    snprintf(args[i].name, sizeof(args[i].name), "%u.%u.%u.%u.%s",
             ip4[3], ip4[2], ip4[1], ip4[0], dnsbls[i]);
    args[i].i = i; args[i].rv = &rv; args[i].n = &n;
    args[i].selves = threads; args[i].mutex = &mutex;
    pthread_create(&threads[i], NULL, thr_dnsbl_find, (void *)&args[i]);
    n++;
  } /* すべてのスレッドが生成されないうちに結果が得られる場合もあることに注意 */
  if (rv)
    pthread_join(threads[rv], NULL); /* 返り値 rv が正なら、rv 番目スレッドのみと合流 */
  else                               /* さもなくば、生成したすべてのスレッドと合流 */
    for (i=1; i<n; i++) pthread_join(threads[i], NULL);
  return rv;
}
        :
```

（例題３）の解答例、並列化（修正版）

abmail/tools/abaddr.c

```
        :
int dnsbl_find(const char *ip)
{
  int rv = 0 /* 共有したい返り値 */, i, n = 1;
  unsigned int ip4[4];
  pthread_t threads[sizeof(dnsbls)/sizeof(char *)];
  pthread_mutex_t mutex; /* 相互排他 (mutex) 変数 */
  struct thr_dnsbl_find_arg args[sizeof(dnsbls)/sizeof(char *)];

  if (sscanf(ip, "%u.%u.%u.%u", &ip4[0], &ip4[1], &ip4[2], &ip4[3]) != 4)
    return rv; /* 不正な IPv4 アドレス表記 */
  pthread_mutex_init(&mutex, NULL); /* 相互排他変数の初期化 */
  for (i=1; i<sizeof(dnsbls)/sizeof(char *) && rv == 0; i++) {
    snprintf(args[i].name, sizeof(args[i].name), "%u.%u.%u.%u.%s",
             ip4[3], ip4[2], ip4[1], ip4[0], dnsbls[i]);
    args[i].i = i; args[i].rv = &rv; args[i].n = &n;
    args[i].selves = threads; args[i].mutex = &mutex;
    pthread_create(&threads[i], NULL, thr_dnsbl_find, (void *)&args[i]);
    n++;
  } /* すべてのスレッドが生成されないうちに結果が得られる場合もあることに注意 */
  if (rv) {
    pthread_join(threads[rv], NULL); /* 返り値 rv が正なら、rv 番目スレッドのみと合流 */
    for (i=1; i<n; i++)              /* 合流しないスレッドはすべてデタッチ */
      if (i != rv) pthread_detach(threads[i]);
  }
  else                               /* さもなくば、生成したすべてのスレッドと合流 */
    for (i=1; i<n; i++) pthread_join(threads[i], NULL);
  return rv;
}
        :
```

（例題３）の解答例、並列化とキャンセル

abmail/tools/abaddr.c

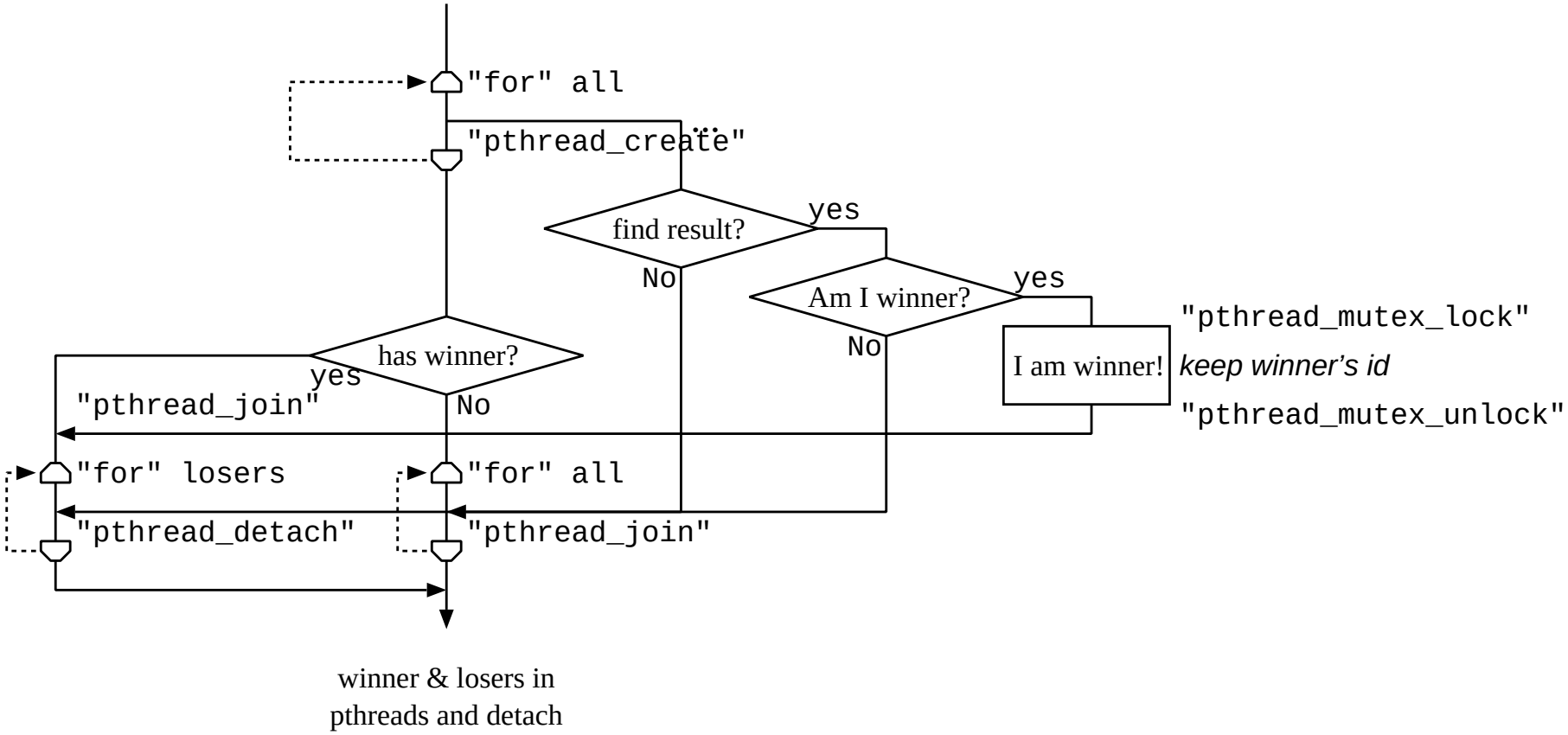
```
          :
void *thr_dnsbl_find(void *arg)
{
  int rv = 0, i;
  struct thr_dnsbl_find_arg *a = (struct thr_dnsbl_find_arg *)arg;
  struct addrinfo hints, *res;

  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL); /* 非同期キャンセレーション */
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);      /* キャンセレーションを許可 */
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_INET; hints.ai_socktype = SOCK_STREAM;
  if ((rv = (getaddrinfo(a->name, NULL, &hints, &res) == 0)?a->i:0)) {
    freeaddrinfo(res);
    if (*a->rv == 0) {
      pthread_mutex_lock(a->mutex);   /* mutex をロック */
      *a->rv = rv;                    /* 返り値を格納 */
      for (i=1; i<*a->n; i++)         /* 他のすべてのスレッドをキャンセル */
        if (i != a->i)
          pthread_cancel(a->selves[i]);
      pthread_mutex_unlock(a->mutex); /* mutex をロック解除 */
    }
  }
  return NULL;
}
          :
```

（例題３）の解答例、並列化とキャンセル（つづき）

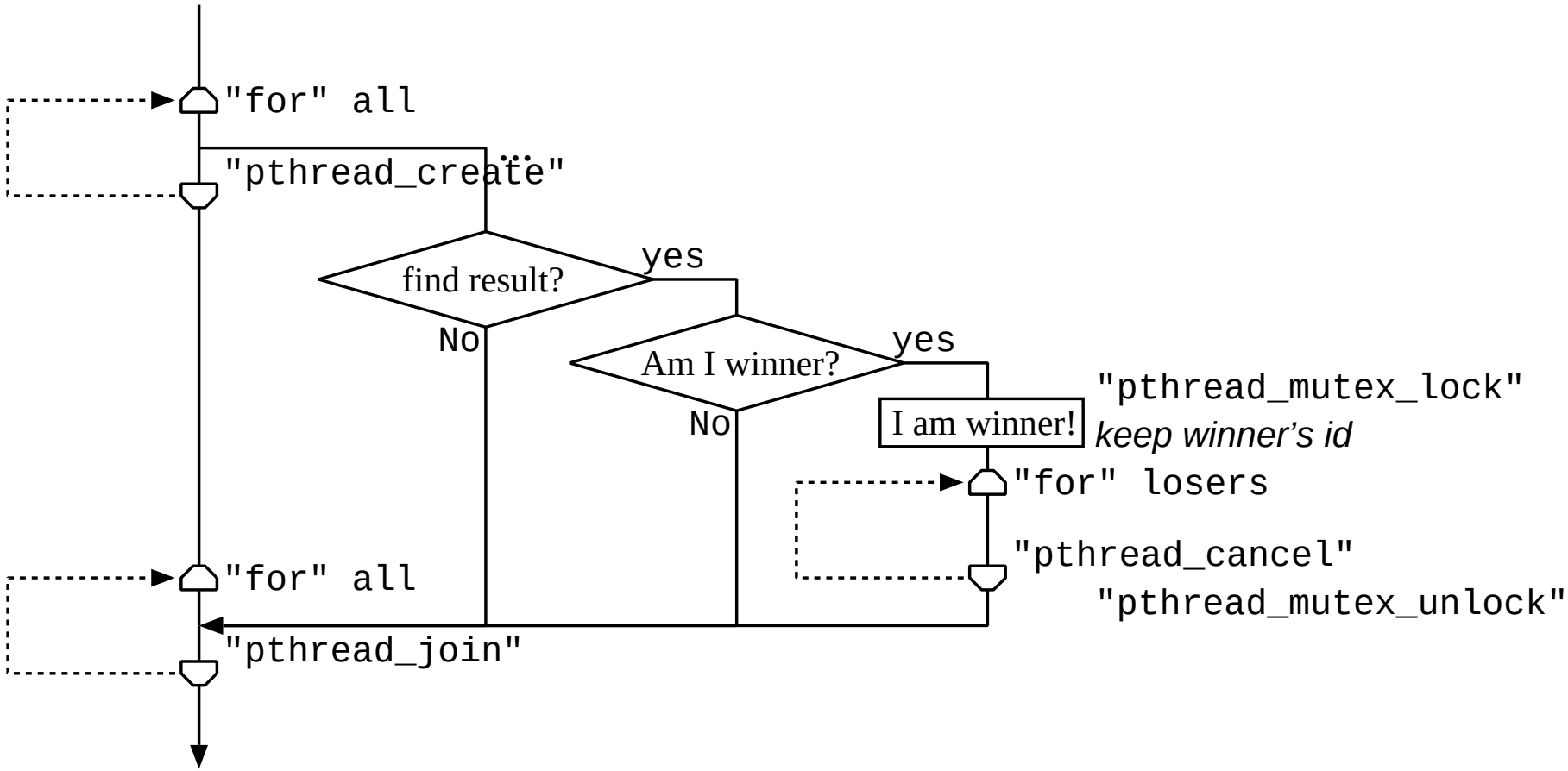
abmail/tools/abaddr.c

```
          :
int dnsbl_find(const char *ip)
{
  int rv = 0 /* 共有したい返り値 */, i, n = 1;
  unsigned int ip4[4];
  pthread_t threads[sizeof(dnsbls)/sizeof(char *)];
  pthread_mutex_t mutex; /* 相互排他 (mutex) 変数 */
  struct thr_dnsbl_find_arg args[sizeof(dnsbls)/sizeof(char *)];

  if (sscanf(ip, "%u.%u.%u.%u", &ip4[0], &ip4[1], &ip4[2], &ip4[3]) != 4)
    return rv; /* 不正な IPv4 アドレス表記 */
  pthread_mutex_init(&mutex, NULL); /* 相互排他変数の初期化 */
  for (i=1; i<sizeof(dnsbls)/sizeof(char *) && rv == 0; i++) {
    snprintf(args[i].name, sizeof(args[i].name), "%u.%u.%u.%u.%s",
             ip4[3], ip4[2], ip4[1], ip4[0], dnsbls[i]);
    args[i].i = i; args[i].rv = &rv; args[i].n = &n;
    args[i].selves = threads; args[i].mutex = &mutex;
    pthread_create(&threads[i], NULL, thr_dnsbl_find, (void *)&args[i]);
    n++;
  } /* すべてのスレッドが生成されないうちに結果が得られる場合もあることに注意 */
  for (i=1; i<n; i++) /* 生成したすべてのスレッドと合流 */
    pthread_join(threads[i], NULL);
  return rv;
}
          : /* 果して正常に動くか、どうか */
```

pthread with detach and cancel(1)



pthread with detach and cancel(2)



winner & losers in
pthreads and cancel

（例題４）イベントプログラミングとスレッド

glx-pthreads/v_rossler00.c と v_rossler01.c は、常微分方程式の求解し、OpenGL を X11 上で扱う glx によってその数値計算結果を可視化するプログラムの典型例である。

v_rossler00.c と比較して v_rossler01.c は、微小に異なる初期値からの解軌道を数本 $(= m)$ まとめて求めている。

v_rossler01.c を pthreads で並列化せよ。

(注) 答えはひとつではない。試行錯誤してみて (v_rossler01〜04.c)、その中で最も優れていると思われるものを v_rossler.c にあげておく。

```
$ make CC=gcc LDLIBS='-lsocket -lnsl' v_rossler00 v_rossler01
$ ./v_rossler00
$ ./v_rossler01
$ make CC=gcc LDLIBS='-lsocket -lnsl' v_rossler02 v_rossler03 v_rossler04 v_rossler
$ ./v_rossler02
$ ./v_rossler03
$ ./v_rossler04
```

(補足) もし動かない場合「GLX_RGBA」がある行をコメントアウトしてください (手抜きで恐縮です)。

（例題４）の解答例、並列化と状態変数

v_rossler.c の主要部分の抜粋１

```
        :
void view_loop(struct view *v, void *o)
{
  struct odeset *odes = (struct odeset *)o;
  int j, k = 0, m = 0;
  pthread_t thrs[odes->m];
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  struct thr_odes arg = { &mutex, &cond, odes, &m, }, *a;
        :
  pthread_mutex_init(&mutex, NULL); /* mutex を初期化 */
  pthread_cond_init(&cond, NULL);   /* 状態変数 cond を初期化 */
  for (k = 0; k < odes->m; k++) { /* 求解したい軌道分のスレッドを生成 */
    arg.k = k; a = malloc(sizeof(arg)); memcpy(a, &arg, sizeof(arg));
    pthread_create(&thrs[k], NULL, thr_odes_routine, a);
  }
  while (!0) {
    usleep(v->wait); /* イベントプログラミングでは CPU を占有してはならない、のでスリープ */
    pthread_mutex_lock(&mutex); /* 状態変数 cond に関連付ける mutex をロック */
    while (m < odes->m) { /* すべての軌道について求解が済んでるか否かを確認 */
      struct timeval tv;
      struct timespec ts = { 0, 1 }; /* 済んでなければ、ちょっと待つ */

      gettimeofday(&tv, NULL);
      ts.tv_sec += tv.tv_sec + (tv.tv_usec*1000 + ts.tv_nsec)/1000000000L;
      ts.tv_nsec += (tv.tv_usec*1000 + ts.tv_nsec)%1000000000L;
      pthread_cond_timedwait(a->cond, a->mutex, &ts); /* 指定するのは絶対時間であることに注意！ */
    }
    m = 0;
        : /* イベント処理 */
    pthread_cond_broadcast(&cond); /* すべてのスレッドに求解再開を指令！ */
    pthread_mutex_unlock(&mutex); /* 状態変数 cond に関連付ける mutex をロック解除 */
  }
}
        :
```

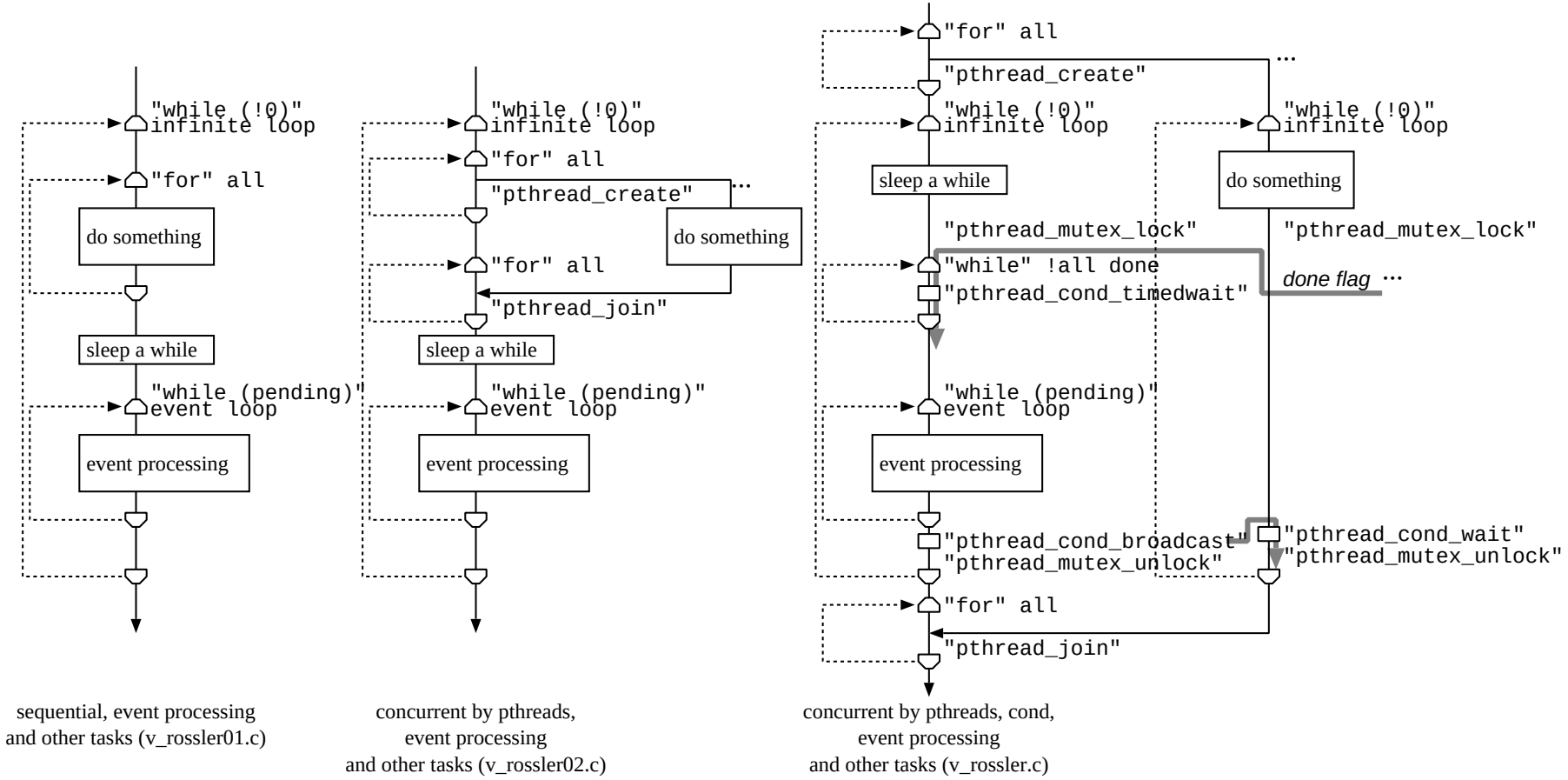
（例題４）の解答例、並列化と状態変数（つづき１）

v_rossler.c の主要部分の抜粋２

```
        :
struct thr_odes {
  pthread_mutex_t *mutex;
  pthread_cond_t *cond;
  struct odeset *odes;
  int *m, k;
};
void *thr_odes_routine(void *arg)
{
  struct thr_odes *a = (struct thr_odes *)arg;
  struct odeset *odes = a->odes;
  double t = odes->t;
  int i, c = odes->c, k = a->k;

  while (!0) {
    for (i=0; i<odes->s; i++) {
      double *v0 = odes->v[k][c], *v1 = odes->v[k][(c+1)%odes->n];

      ode_rk4(odes->p, odes->delta_t, odes->ode->N, odes->ode->dvdt,
      t+=odes->delta_t, v0, v1);
      c = (c + 1)%odes->n;
    }
    pthread_mutex_lock(a->mutex); /* 状態変数 cond に関連付ける mutex をロック */
    (*a->m)++;
    pthread_cond_wait(a->cond, a->mutex); /* mutex に関連付けられた状態変数 cond について待機 */
    c = odes->c;
    pthread_mutex_unlock(a->mutex); /* 状態変数 cond に関連付ける mutex をロック解除 */
  }
  free(arg);
  return NULL;
}
        :
```

(考察) pthread_cond_wait, pthread_cond_timedwait は mutex のロック解除に始まり、再ロックに終ることを理解の助けとせよ。

pthread with events



sequential, event processing
and other tasks (v_rossler01.c)

concurrent by pthreads,
event processing
and other tasks (v_rossler02.c)

concurrent by pthreads, cond,
event processing
and other tasks (v_rossler.c)

スレッドプログラミング：まとめ

POSIX スレッドの利用を通して、

- スレッドの生成、操作
- MT-Safe な関数をスレッド内で使用すること
- 並列可能性の設計、スレッドの粒度とパフォーマンス（行列の積）
- 同期 (mutex ロック)、キャンセレーション（複数の DNSBL 問い合わせを並列化）
- 非同期キャンセレーション-Safe な関数か否か、を検討
- 同期と状態変数（イベントプログラミングとスレッド）
- より詳しくは、「Pthreads プログラミング（榊 正憲 訳、オライリー)」他

その他のリソース：

`abmail/{tools/abaddr.c,abmail.c} - http://www.aihara.co.jp/~taiji/abmail/`

主題ではないが `AltiVec - http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf`