Unix: programowanie z użyciem wątków

Witold Paluszyński
witold.paluszynski@pwr.wroc.pl
http://kcir.pwr.wroc.pl/~witold/

Niniejszy dokument zawiera materiały do wykładu na temat programowania z użyciem wątków normy POSIX w systemie Unix. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

Wątki: wprowadzenie

- Wątek (ang. *thread*) jest sekwencją obliczeń realizowaną w ramach jednego procesu. Zestaw wątków należący do jednego procesu wykonuje się równolegle w ramach jego przestrzeni adresowej.

- Ponadto wątki współdzielą następujące cechy:
  - identyfikator użytkownika i grupy
  - bieżącą kartotekę dyskową
  - otwarte pliki
  - handlery sygnałów i ich deklaracje

- Natomiast każdy wątek ma swój unikalny identyfikator, indywidualny zestaw atrybutów, maskę sygnałów, priorytet i tzw. konteks obliczeniowy, tzn. stos wywołań procedur, zestaw rejestrów, itp.

- Wątek tworzy się funkcją `pthread_create()` jako wywołanie jakiejś funkcji użytkownika.

serwer wieloprocesowy:

```
void serve_client(int sock);

void main(void) {
  int sockS, sockC;
  struct sockaddr_un add;

  sockS=socket(PF_INET,
               SOCK_STREAM,0);
  add.sun_family=AF_INET;/*...*/
  bind(sockS,
       (struct sockaddr *)&add,
       sizeof(add));
  listen(sockS, 5);

  while (1) {
    sockC = accept(sockS,0,0);
    if (fork() == 0){/*potomek*/
      close(sockS);
      serve_client(sockC);
      exit(0);
    }
    close(sockC); /*rodzic*/
  }
}
```

serwer wielowątkowy:

```
void *serve_client(void *);

void main(void) {
  int sockS, sockC;
  struct sockaddr_un add;
  pthread_t t;

  sockS=socket(PF_INET,
               SOCK_STREAM,0);
  add.sun_family=AF_INET;/*...*/
  bind(sockS,
       (struct sockaddr *)&add,
       sizeof(add));
  listen(sockS, 5);

  while (1) {
    sockC = accept(sockS,0,0);
    pthread_create(&t, NULL,
                   serve_client,
                   (void *)&sockC);
    /*watki nie zamykaja gniazdek*/
    /*- sa to wspolne deskryptory*/
  }
}
```

- Jednak dla umożliwienia „zwykłym" funkcjom sygnalizacji przyczyn błędów, każdy z wątków MOŻE otrzymać prywatną kopię zmiennej globalnej `errno` (wymaga to kompilacji programu z definicją `-D_REENTRANT`).

Kończenie pracy wątków

- Wątek kończy się gdy:
  - zakończy się funkcja realizująca go
  - wątek wywoła funkcję `pthread_exit()`
  - wątek zostanie anulowany funkcją `pthread_cancel()`
  - jeden z wątków wykona funkcję `exec...`
  - zakończy się proces macierzysty wątku
- W chwili zakończenia wątku zwalniane są jego indywidualne struktury danych (np. stos), ale nie globalne struktury procesu.
- W chwili zakończenia wątki generują status, który może być odczytany przez wątek macierzysty, podobnie jak to jest z procesami.
- Jeśli zakończy się funkcja główna wątku to skutek jest taki jak gdyby wątek wywołał funkcję `pthread_exit()` ze zwróconą wartością jako argumentem. W ten sposób wszystkie jawnie tworzone wątki różnią się od głównego wątku procesu, którego zakończenie funkcji głównej (main) jest równoważne wywołaniu funkcji `exit()`.
- funkcje związane z wątkami nie sygnalizują błędów przez zmienną `errno` (która jest globalna w procesie), lecz przez niezerowe wartości funkcji.

Wątki: status, wcielanie, odłączanie

- Kończąc pracę, wątek normalnie pozostawia informacje o swoim stanie funkcją `pthread_exit()`. Jest to podobne do przekazania statusu wyjścia procesu.
- Jeden wątek może przejść w stan oczekiwania na zakończenie innego wątku tego samego procesu. Takie oczekiwanie, nazywane „wcielaniem" (ang. *join*), wywołuje się funkcją `pthread_join()`. Jest to podobne do oczekiwania na zakończenie podprocesu funkcją `waitpid()`.
- Jednak, w odróżnieniu od procesów, wątek może być wątkiem „odłączonym" (ang. *detached*), co oznacza, że nie może go wcielić inny wątek, a po zakończeniu tego wątku jest on ostatecznie usuwany, nie czekając na „wcielenie" i odebranie jego statusu przez inny wątek.
- Do dynamicznego odłączania wątków służy funkcja `pthread_detach()`.
- Można też od razu utworzyć wątek jako „odłączony", jednak domyślnie tak się nie dzieje. Stan odłączenia wątku jest jednym z atrybutów wątku, o domyślnej wartości odpowiadającej wątkowi nieodłączonemu.

Wątki: kasowanie

- Skasowanie wątku bywa przydatne, gdy program uzna, że nie ma potrzeby wykonywać dalej wątku, pomimo iż nie zakończył on jeszcze swej pracy. Wątek kasuje funkcja `pthread_cancel()`.
- Skasowanie wątku może naruszyć spójność modyfikowanych przezeń globalnych danych, zatem powinno być dokonywane w przemyślany sposób. Istnieją mechanizmy wspomagające „oczyszczanie" struktur danych (ang. *cleanup handlers*) po skasowaniu wątku. W związku z tym skasowany wątek nie kończy się od razu, lecz w najbliższym punkcie kasowania (ang. *cancellation point*). Jest to tzw. kasowanie synchroniczne.
- Punktami kasowania są wywołania niektórych funkcji, a program może również stworzyć dodatkowe punkty kasowania wywołując funkcję `pthread_testcancel()`.
- W przypadkach gdy wątek narusza struktury danych przed jednym z punktów kasowania, może on zadeklarować procedurę czyszczącą, która będzie wykonana automatycznie w przypadku skasowania wątku. Tę procedurę należy oddeklarować natychmiast po przywróceniu spójności struktur danych.

Atrybuty wątków

- Wątki posiadają zestaw atrybutów, które posiadają wartości domyślne, automatycznie przyjmowane w chwili tworzenia wątku.
- Możliwe jest utworzenie wątku z atrybutami różnymi od domyślnych. Wymaga to utworzenia obiektu artybutów typu `pthread_attr_t`.
- Wartości atrybutów wątek może również zmieniać w czasie pracy odpowiednimi funkcjami „set".

Wątki: obsługa sygnałów

- Każdy wątek ma własną maskę sygnałów. Wątek dziedziczy ją od wątku, który go zainicjował, lecz nie dziedziczy sygnałów czekających na odebranie.
- Sygnał wysłany do procesu jest doręczany do jednego z jego wątków.
- Sygnały synchroniczne, tzn. takie, które powstają w wyniku akcji danego wątku, np. SIGFPE, SIGSEGV, są doręczane do wątku, który je wywołał.
- Sygnał do określonego wątku można również skierować funkcją `pthread_kill`.
- Sygnały asynchroniczne, które powstają bez związku z akcjami, np. SIGHUP, SIGINT, są doręczane do jednego z tych wątków procesu, które nie mają tego sygnału zamaskowanego.
- Jedną ze stosowanych konfiguracji obsługi sygnałów w programach wielowątkowych jest oddelegowanie jednego z wątków do obsługi sygnałów, i maskowanie ich we wszystkich innych wątkach.

Przełączanie wątków

- Gdy proces tworzy kilka wątków, to mogą one być równolegle wykonywane o ile system operacyjny ma wystarczające zasoby; w przeciwnym wypadku stosowane jest przełączanie wątków; możliwe jest wybranie strategii przełączania i priorytetów wątków.
- W systemie Solaris (Sun Microsystems) wątki realizowane są przez mechanizm niższego poziomu, tak zwane LWP (*Light Weight Processes*); każdy proces ma do dyspozycji kilka LWP i każdy LWP może realizować jeden wątek naraz. Same LWP podlegają przełączaniu przez system operacyjny.

Synchronizacja wątków (wstęp)

- Ponieważ wątki są z definicji wykonywane współbieżnie i asynchronicznie, a także operują na współdzielonych globalnych strukturach danych, konieczne są mechanizmy dla zapewnienia wyłączności dostępu.

- Mechanizmy wzajemnej synchronizacji i blokowania ograniczają współbieżność, zatem należy stosować jak najdrobniejsze blokady.

- Dostępne mechanizmy synchronizacji standardu POSIX
  - mutexy (ang. *mutual exclusion*)
  - zmienne warunkowe (ang. *conditional variables*)
  - blokady odczytu i zapisu (ang. *read-write locks*)
  - semafory

Mechanizmy synchronizacji

Mechanizmy synchronizacji zdefiniowane przez normę POSIX wykorzystują struktury pamięciowe tworzone jawnie w programach, np.:

```
static pthread_mutex_t mutex1;

pthread_mutex_t *mutex2;
mutex2 = (pthread_mutex_t *)
         calloc(1, sizeof(pthread_mutex_t));

pthread_mutex_t mutex3 = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t mutex4;
  pthread_mutexattr_t attr_ob;
  pthread_mutexattr_init(&attr_ob)
  pthread_mutex_init(&mutex4, &attr_ob)
```

Wyzerowany mutex jest otwarty.

Synchronizacja wątków (2)

```
#define _REENTRANT
#include <pthread.h>
#define NUM_THREADS 12

#define TRAND(limit,n) \
 {struct timeval t;\
  gettimeofday(&t,(void *)NULL);\
  (n)=rand_r((unsigned *) &t.tv_usec) % (limit)+1;}
void *thread_main(void *);
int  Global_data = 0;

main(int argc,char * argv[]) {
  int i;
  pthread_t thrds[NUM_THREADS];
  for (i=0; i< NUM_THREADS; i++)
    pthread_create(&thrds[i], NULL, thread_main, (void *) NULL);
  for (i=0; i< NUM_THREADS; i++)
    pthread_join(thrds[i], (void **) NULL);
}

void * thread_main(void *null) {
  static pthread_mutex_t Global_mutex; /* wyzerowany, tzn. otwarty */
  {int n; TRAND(NUM_THREADS,n); sleep(n);}
```

```
  pthread_mutex_lock(&Global_mutex);
  Global_data++;
  printf("Watek %3d, proces %d, dane globalne %d\n",
         pthread_self(), getpid(), Global_data);
  pthread_mutex_unlock(&Global_mutex);
  return NULL;
}
```

Synchronizacja wątków (3)

```
#define _REENTRANT
#include <sys/ipc.h>
#include <sys/shm.h>
#include <pthread.h>
#define NUM_THREADS 12


#define TRAND(limit,n) \
 {struct timeval t;\
  gettimeofday(&t,(void *)NULL);\
  (n)=rand_r((unsigned *) &t.tv_usec) % (limit)+1;}
void *thread_main(void *);


pthread_mutex_t *Global_mutexp;
int    g_shmid, *Global_datap;


main(int argc, char *argv[])
{
  pthread_mutexattr_t attr_obj;
  int i, s;
```

```
  /* utw.obszaru pamieci wspolnej mutexu i danych */
  g_shmid = shmget(IPC_PRIVATE,
                   sizeof(int)+sizeof(pthread_mutex_t),
                   IPC_CREAT|0666);
  Global_datap = (int *) shmat(g_shmid, 0, 0);
  Global_mutexp = (pthread_mutex_t *)
                  (Global_datap + sizeof(int));

  /* zainicj.mutexu w pam.wspolnej struk.atrybutowa */
  pthread_mutexattr_init(&attr_obj);
  pthread_mutexattr_setpshared(&attr_obj, PTHREAD_PROCESS_SHARED);
  pthread_mutex_init(Global_mutexp, &attr_obj);

  *Global_datap = 0;
  for (i=0; i< NUM_THREADS; i++) /* odpal.podproces.*/
    if ( fork() == 0 ){
      thread_main((void *) NULL);
      exit(2);
    }
  while((s = (int)wait(NULL)) && s != -1);

  shmdt((char *) Global_datap);
  shmctl(g_shmid, IPC_RMID, (struct shmid_ds *) 0);
  exit(0);
}
```

```
void * thread_main(void *null)   {
  {int n; TRAND(NUM_THREADS,n); sleep(n);}
  if (0!=pthread_mutex_trylock(Global_mutexp)) {
    printf("Watek %2d, proces %5d, czeka...\n",pthread_self(),getpid());
    pthread_mutex_lock(Global_mutexp);
  }
  (*Global_datap)++;
  printf("Watek %2d, proces %5d, dane globalne %d\n",
         pthread_self(), getpid(), *Global_datap);
  pthread_mutex_unlock(Global_mutexp);
  return NULL;
}
```

Przykład działa podobnie jak poprzedni, tylko nie tworzy wątków, a podprocesy, uzyskujące dostęp do wspólnego mutexu przez dostępny wszystkim obszar pamięci wspólnej, mieszczący dane globalne i mutex.

UWAGA: Linux nie obsługuje atrybutu PSHARED
`pthread_mutexattr_setpshared()`

Synchronizacja wątków (4)

```
#define _REENTRANT
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <pthread.h>

#define  TRAND(limit,n) {struct timeval t;\
           gettimeofday(&t,(void *)NULL);\
           (n)=rand_r((unsigned *) &t.tv_usec) % (limit)+1;}
void create_shared_file();
void *thread_main(void *);

#define GLOBAL_FILE "SHARED_FILE"
#define GLOBAL_SIZE sizeof(int)+sizeof(pthread_mutex_t)

pthread_mutex_t *Global_mutexp;
int             *Global_datap;
int gfd;

main(int argc, char * argv[])
{
```

```
  if ((argc>1) && (0==strncasecmp(argv[1],"init",4))) {
    create_shared_file();
    exit(0);
  }
  else if (0!=access(GLOBAL_FILE, F_OK)) {
    printf("Pierwsze wywolanie musi podac argv[1]=init\n");
    exit(-1);
  }

  while((gfd = open(GLOBAL_FILE, O_RDWR)) == -1)
    sleep(1);
  Global_datap = (int *) mmap(NULL, GLOBAL_SIZE,
                              PROT_READ|PROT_WRITE,
                              MAP_SHARED, gfd, 0);
  Global_mutexp = (pthread_mutex_t *)
                 (Global_datap + sizeof(int));
  thread_main((void *)NULL);
  close(gfd);
  exit(0);
} /* main */

void create_shared_file() {
  static char zeroed[GLOBAL_SIZE];
  pthread_mutexattr_t attr_obj;
```

```
  gfd = creat(GLOBAL_FILE, O_CREAT|O_RDWR|0644);
  write(gfd, zeroed, GLOBAL_SIZE);
  close(gfd);
  chmod(GLOBAL_FILE, S_IRWXU|S_IWGRP|S_IRGRP|S_IROTH);

  gfd = open(GLOBAL_FILE, O_RDWR);
  Global_datap = (int *) mmap(NULL, GLOBAL_SIZE,
                              PROT_READ|PROT_WRITE,
                              MAP_SHARED, gfd, 0);
  *Global_datap = 0;

  Global_mutexp = (pthread_mutex_t *)
                 (Global_datap + sizeof(int));
  pthread_mutexattr_init(&attr_obj);
  pthread_mutexattr_setpshared(&attr_obj,PTHREAD_PROCESS_SHARED);
  pthread_mutex_init(Global_mutexp, &attr_obj);
  close(gfd);
}

void * thread_main(void *null)   {
  {int n; TRAND(12,n); sleep(n);}
  if (0!=pthread_mutex_trylock(Global_mutexp)) {
    printf("Watek %2d, proces %5d, czeka...\n",
           pthread_self(), getpid());
    pthread_mutex_lock(Global_mutexp);
```

```
  }
  (*Global_datap)++;
  printf("Watek %2d, proces %5d, dane globalne %d\n",
         pthread_self(), getpid(), *Global_datap);
  pthread_mutex_unlock(Global_mutexp);
  return NULL;
}
```

Ten przykład również udostępnia mutexy wielu procesom, jednak odwzorowuje obszar pamięci mutexu do pliku, dzieki czemu możliwy jest globalny dostęp do mutexu z dowolnego procesu.

Przykład: producenci i konsumenci

przykład producentów i konsumenta (jednego)

```
#define MAXNITEMS       1000000
#define MAXNTHREADS         100

int     nitems; /* read-only by producer and consumer */
struct {
  pthread_mutex_t   mutex;
  int   buff[MAXNITEMS];
  int   nput;
  int   nval;
} shared = { PTHREAD_MUTEX_INITIALIZER };

void    *produce(void *), *consume(void *);
```

```
int
main(int argc, char **argv)
{
  int         i, nthreads, count[MAXNTHREADS];
  pthread_t   tid_produce[MAXNTHREADS], tid_consume;

  if (argc != 3)
    err_quit("usage: prodcons2 <#items> <#threads>");
  nitems = min(atoi(argv[1]), MAXNITEMS);
  nthreads = min(atoi(argv[2]), MAXNTHREADS);

  Set_concurrency(nthreads);
  /* start all the producer threads */
  for (i = 0; i < nthreads; i++) {
    count[i] = 0;
    Pthread_create(&tid_produce[i], NULL, produce,
                   &count[i]);
  }

  /* wait for all the producer threads */
  for (i = 0; i < nthreads; i++) {
    Pthread_join(tid_produce[i], NULL);
    printf("count[%d] = %d\n", i, count[i]);
  }
```

```
void *
produce(void *arg)
{
  for ( ; ; ) {
    Pthread_mutex_lock(&shared.mutex);
    if (shared.nput >= nitems) {
      Pthread_mutex_unlock(&shared.mutex);
      return(NULL); /* array is full, we're done */
    }
    shared.buff[shared.nput] = shared.nval;
    shared.nput++;
    shared.nval++;
    Pthread_mutex_unlock(&shared.mutex);
    *((int *) arg) += 1;
  }
}

void *
consume(void *arg)
{
  int     i;

  for (i = 0; i < nitems; i++) {
    if (shared.buff[i] != i)
      printf("buff[%d] = %d\n", i, shared.buff[i]);
```

```
  /* start, then wait for the consumer thread */
  Pthread_create(&tid_consume, NULL, consume, NULL);
  Pthread_join(tid_consume, NULL);

  exit(0);
}
```

```
  }
  return(NULL);
}
```

Przykład: producenci i konsumenci (2)

wersja z konsumentem pracującym równolegle

```
int
main(int argc, char **argv)
{
  int         i, nthreads, count[MAXNTHREADS];
  pthread_t   tid_produce[MAXNTHREADS], tid_consume;

  if (argc != 3)
    err_quit("usage: prodcons <#items> <#threads>");
  nitems = min(atoi(argv[1]), MAXNITEMS);
  nthreads = min(atoi(argv[2]), MAXNTHREADS);

  Set_concurrency(nthreads + 1);
  /* create all producers and one consumer */
  for (i = 0; i < nthreads; i++) {
    count[i] = 0;
    Pthread_create(&tid_produce[i], NULL, produce,
                   &count[i]);
  }
  Pthread_create(&tid_consume, NULL, consume, NULL);
```

```
  /* wait for all producers and the consumer */
  for (i = 0; i < nthreads; i++) {
    Pthread_join(tid_produce[i], NULL);
    printf("count[%d] = %d\n", i, count[i]);
  }
  Pthread_join(tid_consume, NULL);

  exit(0);
}
```

```
void * produce(void *arg) {
  for ( ; ; ) {
    Pthread_mutex_lock(&shared.mutex);
    if (shared.nput >= nitems) {
      Pthread_mutex_unlock(&shared.mutex);
      return(NULL); /* array is full, we're done */
    }
    shared.buff[shared.nput] = shared.nval;
    shared.nput++;
    shared.nval++;
    Pthread_mutex_unlock(&shared.mutex);
    *((int *) arg) += 1;
  }
}
void consume_wait(int i) {
  for ( ; ; ) {
    Pthread_mutex_lock(&shared.mutex);
    if (i < shared.nput) {
      Pthread_mutex_unlock(&shared.mutex);
      return;         /* an item is ready */
    }
    Pthread_mutex_unlock(&shared.mutex);
  }
}
void * consume(void *arg) {
```

```
  int     i;

  for (i = 0; i < nitems; i++) {
    consume_wait(i);
    if (shared.buff[i] != i)
      printf("buff[%d] = %d\n", i, shared.buff[i]);
  }
  return(NULL);
}
```

Synchronizacja wątków: zmienne warunkowe

- do czekania na, i sygnalizowania, spełnienia jakichś warunków służą zmienne warunkowe

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex,
                      const struct timespec *abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- algorytm działania funkcji `pthread_cond_wait`:
  1. odblokowanie mutexu podanego jako argument wywołania
  2. uśpienie wątku do czasu aż jakiś inny wątek wywoła funkcję `pthread_cond_signal` na danej zmiennej warunkowej
  3. ponowne zablokowanie mutexu

- możliwe jest zjawisko „spontanicznego obudzenia", tzn. powrotu z funkcji `pthread_cond_wait` bez wywołania `pthread_cond_signal`; należy się z tym liczyć i sprawdzać poprawność stanu zmiennych po przebudzeniu

---

Synchronizacja wątków: zmienne warunkowe (cd.)

ogólnie schemat użycia zmiennej warunkowej ze stowarzyszonym mutexem jest następujacy:

```
struct {
  pthread_mutex_t mutex;
  pthread_cond_t  cond;
  { ... inne zmienne przechowujace warunek }
} globvar = { PTHREAD_MUTEX_INITIALIZER,
              PTHREAD_COND_INITIALIZER, {...} };
```

schemat sygnalizacji zmiennej warunkowej:

```
pthread_mutex_lock(&globvar.mutex);
{ kod ustawiajacy warunek oczekiwany przez CV }
pthread_cond_signal(&globvar.cond);
pthread_mutex_unlock(&globvar.mutex);
```

schemat sprawdzania warunku i oczekiwania:

```
pthread_mutex_lock(&globvar.mutex);
while ( { warunek niespelniony } )
  pthread_cond_wait(&globvar.cond, &globvar.mutex);
{ kod wykorzystujacy oczekiwany warunek }
pthread_mutex_unlock(&globvar.mutex);
```

Przykład: producenci i konsumenci (3)

```
int     buff[MAXNITEMS];
struct {
  pthread_mutex_t   mutex;
  pthread_cond_t    cond;
  int               nput;
  int               nval;
  int               nready;
} nready = { PTHREAD_MUTEX_INITIALIZER,
             PTHREAD_COND_INITIALIZER };
```

```
void *
produce(void *arg) {
  for ( ; ; ) {
    Pthread_mutex_lock(&nready.mutex);
    if (nready.nput >= nitems) {
      Pthread_mutex_unlock(&nready.mutex);
      return(NULL);       /* array is full, we're done */
    }
    buff[nready.nput] = nready.nval;
    nready.nput++;
    nready.nval++;
    nready.nready++;
    Pthread_cond_signal(&nready.cond);
    Pthread_mutex_unlock(&nready.mutex);
    *((int *) arg) += 1;
  }
}

void *
consume(void *arg) {
  int     i;

  for (i = 0; i < nitems; i++) {
    Pthread_mutex_lock(&nready.mutex);
    while (nready.nready == 0)
```

Przykład: producenci i konsumenci (4)

```
int     buff[MAXNITEMS];
struct {
  pthread_mutex_t   mutex;
  int               nput;   /* next index to store */
  int               nval;   /* next value to store */
} put = { PTHREAD_MUTEX_INITIALIZER };

struct {
  pthread_mutex_t   mutex;
  pthread_cond_t    cond;
  int               nready; /* number ready for consumer */
} nready = { PTHREAD_MUTEX_INITIALIZER,
             PTHREAD_COND_INITIALIZER };
```

```
      Pthread_cond_wait(&nready.cond, &nready.mutex);
    nready.nready--;
    Pthread_mutex_unlock(&nready.mutex);

    if (buff[i] != i)
      printf("buff[%d] = %d\n", i, buff[i]);
  }
  return(NULL);
}
```

```
void * produce(void *arg) {
  for ( ; ; ) {
    Pthread_mutex_lock(&put.mutex);
    if (put.nput >= nitems) {
      Pthread_mutex_unlock(&put.mutex);
      return(NULL);       /* array is full, we're done */
    }
    buff[put.nput] = put.nval;
    put.nput++;
    put.nval++;
    Pthread_mutex_unlock(&put.mutex);

    Pthread_mutex_lock(&nready.mutex);
    if (nready.nready == 0)
      Pthread_cond_signal(&nready.cond);
    nready.nready++;
    Pthread_mutex_unlock(&nready.mutex);
    *((int *) arg) += 1;
  }
}
void * consume(void *arg) {
  int     i;

  for (i = 0; i < nitems; i++) {
    Pthread_mutex_lock(&nready.mutex);
```

```
    while (nready.nready == 0)
      Pthread_cond_wait(&nready.cond, &nready.mutex);
    nready.nready--;
    Pthread_mutex_unlock(&nready.mutex);
    if (buff[i] != i)
      printf("buff[%d] = %d\n", i, buff[i]);
  }
  return(NULL);
}
```