Concurrency, Processes & Threads - Exercises

The first three exercises explore some issues relating to concurrent execution of processes, and the difference between a *process* and a *thread* in **Unix**. Threads are implemented slightly differently in different operating systems. These exercises assume you are working in Linux.

You might find these on-line references useful -

http://www.llnl.gov/computing/tutorials/pthreads/,
http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

Download the resource archive, ConcurProcThreadsSrcFiles.zip

Exercise 1.

Find file processdemo.c in the resource archive.

This program uses *`fork()`* to create a child process, which executes an infinite loop in a function called *`adjustX`* with parameters "child", 1. After creating the child process the parent process executes an infinite loop in its own instance of *`adjustX`* with parameters "parent", -1.

(a) Examine the source code and try to work out what the output from the program will be.

(b) Compile the program with e.g.

```
gcc processdemo.c -o processdemo
```

(c) Run the program. Is the output what you expected?

(d) While the program is running, use **`ps xl`** (in another terminal window) and identify the processes for *processdemo*. Which process is the parent and which is the child? How can you tell?

Hints:
(i) **`man ps`** will tell you what the output from **`ps`** means
(ii) look at the process id (PID) and parent process id (PPID) columns for your processes to work out the identifier for the parent and child versions of *processdemo*

```
UID  PID  PPID ………………… TTY
349  5842 2321 ………………… pts/2 ksh
349  9321 5842 ………………… pts/2 processdemo
349  9322 9321 ………………… pts/2 processdemo
349  8541 3467 ………………… pts/3 ksh
```

(e) Use the kill command to stop one of the clones. What happens to the other clone?

(f) Does it make any difference if the parent is killed before the child?

```
/* processdemo.c */
/* Process demonstration program. Note there are no shared variables */

#include <stdio.h>
#include <stdlib.h>

const clock_t MAXDELAY = 2000000;
int x = 50;   /* a global variable */

void delay(clock_t ticks) { /* a "busy" delay */
  clock_t start = clock();
  do
    ; while (clock() < start + ticks);
}

void adjustX(char * legend, int i) {
   while (1)  /* loop forever */
   {   printf("%s: %i\n", legend, x);
       x += i;
       delay(rand()%MAXDELAY);
   }
}

main()
{  int c;
   srand(time(NULL));
   printf("creating  new process:\n");
   c = fork();
   printf("process %i created\n", c);
   if (c==0)
      adjustX("child", 1);    /* child process */
   else
      adjustX("parent", -1);  /* parent process */
}
```

Exercise 2.
Extract the file threaddemo.c

This program shows how threads can be created in a C program, using the POSIX threads library. The program is superficially similar to processdemo, except that it uses threads instead of processes.

If you want more information on the routines available in the posix threads library on you can use the command `man pthread`.

For information on a specific routine use man <routine-name> (eg man pthread_create)

(a) Compile threaddemo.c

To compile this program you need to tell the compiler to use the threads library - use the command

```
gcc threaddemo.c –lpthread -o threaddemo
```

(b) Run the program. How does the output differ from processdemo. Can you explain the difference?

(c) How does the rate of output for threaddemo (measured in, say, lines per minute) compare to processdemo? Can you account for the difference?

(d) Use ps x (in another terminal window) to check you have only one threaddemo Unix process.

(e) Investigate the effect of removing the loop in main()

```
/* threaddemo.c */
/* Thread demonstration program. Note that this program uses a shared
variable in an unsafe manner (eg mutual exclusion is not attempted!) */

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

int x = 50;   /* a global (shared) variable */
const clock_t MAXDELAY = 2000000;
const int INC = 1, DEC = -1;

void delay(clock_t ticks) { /* a "busy" delay */
  clock_t start = clock();
  do
    ; while (clock() < start + ticks);
}

void * adjustX(void *n)
{  int *i = (int *)n;
   while (1)   /* loop forever */
   {    printf("adjustment = %2i; x = %i\n", *i, x);
        x += *i;
        delay(rand()%MAXDELAY);
   }
   return(n);
}

main()
{  int a;
   srand(time(NULL));
   pthread_t  up_thread, dn_thread;

   pthread_attr_t *attr;  /* thread attribute variable */
   attr=0;

   printf("creating threads:\n");
   pthread_create(&up_thread,attr, adjustX, (void *)&INC);
   pthread_create(&dn_thread,attr, adjustX, (void *)&DEC);

   while (1) /* loop forever */
   { ;}
}
```

Exercise 3.

What changes would you expect to the rate of output for processdemo and threaddemo if you repeat the comparison between them on a machine with (i) a larger number of users? (ii) no other users?

Exercise 4 - explore joinEx.c

Remember that, working in Linux you compile including the pthread library like this –

```
gcc source.c –lpthread –o exec
```

(This names the executable file exec. Omitting the last option leaves you with an executable called a.out)

joinEx.c--

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

#define NTHRDS 10

//Worker thread data structure definition
typedef struct {
  int id;
  double rslt;
} threadData_t;

threadData_t threadData[NTHRDS];  // worker thread data structures
pthread_t thrd[NTHRDS];
      //data structure for managing several worker thread instances

void * theWork(void * n) { //main function of the "worker thread"
  threadData_t * pData = (threadData_t *) n;
  int i;

  pData->rslt = 0.0;
  for (i=0; i<1000000; i++) //do lots of work
    pData->rslt += (double)random();
  pData->rslt /= 1000000;
  printf("Result from thread %d : %e\n", pData->id, pData->rslt);
  pthread_exit(n);
      //Could exit((void*)0) if we wished not to return any data .
  // n actually points at the threadData struct, so can return the
  // result.
}

int main(int argc, char ** argv) {
  pthread_attr_t attr;
                  //data structure for managing thread attributes
  pthread_attr_init(&attr); //initialise thread attribute data ...
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
                           //...and set "joinable" attribute
  int t, e;
  void *status;

  for (t=0; t<NTHRDS; t++) { // create several worker thread
                                         instances
    threadData[t].id = t;
    printf("Creating thread %d\n", t);
41:     if ((e = pthread_create(&thrd[t],
                        &attr, theWork, (void*)(&threadData[t])))!= 0) {
42:       printf("Thread create error %d\n", e);
43:       exit(-1);
44:     }
45:   }
46:
47:   pthread_attr_destroy(&attr);
48:   //garbage-collect the thread attribute oject, no longer needed
49:
50:   for (t=0; t<NTHRDS; t++) { //wait for each worker thread to join
51:     if ((e = pthread_join(thrd[t], &status))!= 0) {
52:       printf("Thread %d join error %d\n", t, e);
53:       exit(-1);
54:     }
55:     printf("Thread %d has joined: status = %lx\n", t, (long)status);
56:   }
57:
58:   printf("All results:\n");
59:   for (t=0; t<NTHRDS; t++) {
60:     printf("Thread %2d: %e\n", t, threadData[t].rslt);
61:   }
62:   pthread_exit(NULL); //master-thread (program main) is done.
63: }
```

Notes

1. Lines 8-11 define a data record type with an int 'id' field and a double 'rslt' field. Line 13 declares an array of these records, one for each thread. These will be used to pass data into the thread functions and get data back from the thread functions. One record is associated with each thread. Lines 14-15 declare and array of pthread_t data structures for managing the threads.

2. Lines 17-28 define the 'thread function' -- each thread will run an instance. The void * parameter will actually point at a data record as defined above: the id field = the thread ID, the rslt field is where the thread accumulates its result. The function just averages 1000000 random numbers. Note 6 below explains how the results are passed back from the thread function.

3. Note (line 27) the call pthread_exit(n). This makes n available to main() via the pthread_join(thrd[t], &status) call on line 51. void *status is actually the value of n from the thread function; in this case it is actually the same as the pointer passed into the thread function.

4. Lines 31-4 set up a pthreads_attr data structures later used to ensure each thread has its joinable attribute set. This data structure is later disposed of in lines 47-8.

5. Lines 38-45 Create all the threads, first initialising the thread data structures so that the record associated with thread t has id=t. Then thread t is created, passing a pointer to threadData[t], cast to void *.

6. Lines 50-56 wait for all the threads to join. The status that will be printed out is actually the void * n from the thread function; this is actually the address of the relevant threadData[t]!
7. Lines 58-61 display all the results returned by the threads. The thread exit status could have been cast to threadData_t * and these rslt fields interrogated; but this is not necessary as the pointer returned by the exit status is the same as the pointer passed into the thread function in the first place.

This example is actually a good design pattern for getting user data into and out of a thread function via the void *s.

Exercise 5 -

Write your own examples of joining threads and passing data into and out of the thread function.

You might start by adapting threaddemo so that the thread function quits when x deviates by more than some threshold above or below the initial value. The main function should use join to wait for the two other threads, and print a suitable message. See threaddemo1.c for an example of how this might be done.

Further adapt threaddemo so that a loop iteration count is returned from each thread. See threaddemo2.c, threaddemo3.c for solutions: the former uses 'return n', the latter pthread_exit() as recommended by the pthead standard.

Try passing data into the threaddemo thread function using the threadData_t approach. See threaddemo4.c, for example.

Exercise 6 - explore simpleMutexEx.c

What happens when the lines to lock and unlock the mutex are commented out? Can you explain the output behaviour?

```
/*
 * Simple mutex demo using POSIX threads and mutexes (Linux version)
 * MJB Sep 07
 */
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

#define NTHRDS 5

int sharedData = 0;
pthread_mutex_t mutex;

void delay(int secs) { //utility function
  time_t beg = time(NULL), end = beg + secs;
  do ; while (time(NULL) < end);
}

//Try commenting out the calls to the lock and unlock functions.
void *add1000(void *n) {
  pthread_mutex_lock(&mutex); //Obtain exclusive mutex lock
  int j = sharedData;
  delay(rand() % 6);
  sharedData = j + 1000;
  pthread_mutex_unlock(&mutex); //Release mutex lock
  printf("1000 added!\n");
  return n;
}

int main() {
  pthread_attr_t *attr = NULL;
  pthread_t thrd[NTHRDS]; //Data strs form managing several threads
  int t;

  pthread_mutex_init(&mutex, NULL);
  srand(time(NULL)); // initialise random num generator

  for (t=0; t<NTHRDS; t++) //create & initialise several thread instances
    pthread_create(&thrd[t], attr, add1000, NULL);

  for (t=0; t<NTHRDS; t++) //wait for all threads to finish ...
    pthread_join(thrd[t], NULL);
  printf("Shared data = %d\n", sharedData); //...& display net result.
  return 0;
}
```

Exercise 5.

Explore prodconsUnsync.c

Exercise the program – does it behave correctly?
Add some code using pthread_join() and pthread_exit() as in joinEx.c so that the program exits cleanly. At the moment it doesn't and has to be killed with control-C.

See if you can improve its behaviour using mutexes.

```
/*
 * Producer/Consumer demo using POSIX threads without synchronization
 * Linux version
 * MJB Apr'05
 */
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

/* buffer using a shared integer variable */
typedef struct {
  int writeable; /*true/false*/
  int sharedData;
} buffer;

buffer theBuffer; /* global */

int store(int item, buffer *pb) {
  pb->sharedData = item;      /*put data in buffer... */
  pb->writeable = !pb->writeable;
  return 0;
}

int retrieve(buffer *pb) {
  int data;
  data = pb->sharedData;       /*get data from buffer...*/
  pb->writeable = !pb->writeable;
  return data;
}

void delay(int secs) { /*utility function*/
  time_t beg = time(NULL), end = beg + secs;
  do {
    ;
  } while (time(NULL) < end);
}

/* core routine for the producer thread */
void *producer(void *n) {
  int j=1;
  while (j<=10) {
    store(j, &theBuffer);
    printf("Stored: %d\n", j);
     j++;
    delay(rand() % 6);  /* up to 5 sec */
  }
  return n;
}

/* core routine for the consumer thread */
void *consumer(void *n) {
  int j=0, tot=0;
  while (j < 10) {
    j = retrieve(&theBuffer);
    tot += j;
    printf("Retrieved: %d\n", j);
    delay(rand() % 6);  /* up to 5 sec */
  }
  printf("Retrieved total = %d\n", tot);
  return(n);
}

int main() {
  pthread_attr_t *attr = NULL;
  pthread_t prodThread, consThread;

  theBuffer.writeable = 1;
  srand(time(NULL)); /* initialise the rng */

  pthread_create(&prodThread, attr, producer, NULL);
  pthread_create(&consThread, attr, consumer, NULL);

  while (1)
    ;        /*loop forever*/
  return 0;  /*!!!*/
}
```

Exercise 6.

Explore the use of condition variables in prodcons.c.

As in exercise 5, add code using pathread_join() to make main() exit cleanly.

Check that the threads alternate accesses of the buffer correctly. How do you know this?

Sometimes the console window show a number retrieved *before* it is stored! Huh? Is the computer time-reversed??? Explain what is going on here.

```
/* prodcons.c
 * Producer/Consumer demo using POSIX threads
 * synchronized with condition variables and mutexes
 * Linux version
 * MJB Apr 05 -- Sep 07
 */
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

/* buffer using a shared integer variable */
typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t notFull;   /*condition vbl*/
  pthread_cond_t notEmpty;
  int writeable; /*true/false*/
  int sharedData;
} buffer;

buffer theBuffer; /* global */
```

```
int store(int item, buffer *pb) {
  pthread_mutex_lock(&pb->mutex);
  while (!pb->writeable) /*buffer full*/
    pthread_cond_wait(&pb->notFull, &pb->mutex);
  pb->sharedData = item;      /*put data in buffer... */
  pb->writeable = !pb->writeable;
  pthread_cond_signal(&pb->notEmpty);
  pthread_mutex_unlock(&pb->mutex);
  return 0;
}

int retrieve(buffer *pb) {
  int data;
  pthread_mutex_lock(&pb->mutex);
  while (pb->writeable) /*buffer "empty" -- no new data*/
    pthread_cond_wait(&pb->notEmpty, &pb->mutex);
  data = pb->sharedData;       /*get data from buffer...*/
  pb->writeable = !pb->writeable;
  pthread_cond_signal(&pb->notFull);
  pthread_mutex_unlock(&pb->mutex);
  return data;
}

void delay(int secs) { /*utility function*/
  time_t beg = time(NULL), end = beg + secs;
  do {
    ;
  } while (time(NULL) < end);
}

/* core routine for the producer thread */
void *producer(void *n) {
  int j=1;
  while (j<=10) {
    store(j, &theBuffer);
    printf("Stored: %d\n", j);
     j++;
    delay(rand() % 6);  /* up to 5 sec */
  }
  return n;
}

/* core routine for the consumer thread */
void *consumer(void *n) {
  int j=0, tot=0;
  while (j < 10) {
    j = retrieve(&theBuffer);
    tot += j;
    printf("Retrieved: %d\n", j);
    delay(rand() % 6);  /* up to 5 sec */
  }
  printf("Retrieved total = %d\n", tot);
  return(n);
}

int main() {
  pthread_attr_t *attr = NULL;
  pthread_t prodThread, consThread;

  theBuffer.writeable = 1;
  pthread_cond_init(&theBuffer.notFull, NULL);
  pthread_cond_init(&theBuffer.notEmpty, NULL);

  pthread_mutex_init(&theBuffer.mutex, NULL);
  srand(time(NULL)); /* initialise the rng */

  pthread_create(&prodThread, attr, producer, NULL);
  pthread_create(&consThread, attr, consumer, NULL);

  pthread_join(prodThread, NULL); //wait for the two threads
  pthread_join(consThread, NULL); //to join, then finish
  return 0;  /*!!!*/
}
```

Can you arrange for the producer and consumer console displays to be time-stamped? (See prodconsTmd.c in the source file bundle.)

Exercise 7.

Explore circbuff.c. Does the size of a buffer make a difference to the smoothness of the program's operation?

```
/*
 * Circular buffer demo using POSIX threads and mutexes
 * Linux version
 * MJB Apr'05
 */
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#define BUFSZ 6

/* circular buffer */
typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t notFull;   /*condition vbl*/
  pthread_cond_t notEmpty;
  int cnt /*num items in buf*/, frst, last;
  int buf[BUFSZ];
} buffer;

buffer theBuffer; /* global */

int store(int item, buffer *pb) {
  pthread_mutex_lock(&pb->mutex);
  while (pb->cnt == BUFSZ) /*buffer full*/
    pthread_cond_wait(&pb->notFull, &pb->mutex);
  pb->buf[pb->last] = item;      /*put data in buffer... */
  pb->last++; pb->last %= BUFSZ; /*... & update last ptr */
  pb->cnt++;
  pthread_cond_signal(&pb->notEmpty);
  pthread_mutex_unlock(&pb->mutex);
  return 0;
}

int retrieve(buffer *pb) {
  int data;
  pthread_mutex_lock(&pb->mutex);
  while (pb->cnt == 0) /*buffer empty*/
    pthread_cond_wait(&pb->notEmpty, &pb->mutex);
  data = pb->buf[pb->frst];       /*get data from buffer...*/
  pb->frst++; pb->frst %= BUFSZ; /*... & update first ptr */
  pb->cnt--;
  pthread_cond_signal(&pb->notFull);
  pthread_mutex_unlock(&pb->mutex);
  return data;
}

void display(buffer *pb) {
  int i;
  for (i = 0; i < BUFSZ; i++)
    printf("%d  ", pb->buf[i]);
  printf("; frst=%d; last=%d\n", pb->frst, pb->last);
}

void delay(int secs) { /*utility function*/
  time_t beg = time(NULL), end = beg + secs;
  do {
    ;
  } while (time(NULL) < end);
}

/* core routine for the producer thread */
void *producer(void *n) {
  int j=0;
  while (1) { /* loop forever */
    store(j, &theBuffer);
    printf("Stored: %d\n", j);
    display(&theBuffer);
    j = (j+1)%20;
    delay(rand() % 6);  /* up to 5 sec */
  }
  return n;
}

/* core routine for the consumer thread */
void *consumer(void *n) {
  int j;
  while (1) {  /* loop forever */
    j = retrieve(&theBuffer);
    printf("Retrieved: %d\n", j);
    display(&theBuffer);
    delay(rand() % 6);  /* up to 5 sec */
  }
  return(n);
}

int main() {
  pthread_attr_t *attr = NULL;
  pthread_t prodThread, consThread;

  theBuffer.cnt = 0;
  pthread_mutex_init(&theBuffer.mutex, NULL);
  srand(time(NULL)); /* initials the rng */

  pthread_create(&prodThread, attr, producer, NULL);
  pthread_create(&consThread, attr, consumer, NULL);

  getchar(); //Press ENTER to finish
  return 0;
}
```

Exercise 8.

Explore queue.c

This is a variation of circbuff.c with multiple producers, multiple consumers. Check they are working correctly? Does the program continue to work correctly as more of each are added? What happens if you try to do without the condition variables?

```
/*
 * queue.c
 * Queue based on Circular buffer demo using POSIX threads and mutexes
 * Linux version
 * MJB Apr'05
 */
```

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define BUFSZ 6
#define LEGENDSZ 20

typedef struct {
  int seqNo;
  char legend[LEGENDSZ];
} item_t;

/* circular buffer */
typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t notFull;   /*condition vbl*/
  pthread_cond_t notEmpty;
  int cnt /*num items in buf*/, frst, last;
  item_t * buf[BUFSZ];
} buffer_t;

buffer_t theBuffer; /* global */

void store(item_t * pItem, buffer_t *pb) {
  pthread_mutex_lock(&pb->mutex);
  while (pb->cnt == BUFSZ) /*buffer full*/
    pthread_cond_wait(&pb->notFull, &pb->mutex);
  pb->buf[pb->last] = pItem;      /*put data item in buffer... */
  pb->last++; pb->last %= BUFSZ; /*... & update last ptr */
  pb->cnt++;
  pthread_cond_signal(&pb->notEmpty);
  pthread_mutex_unlock(&pb->mutex);
}

item_t * retrieve(buffer_t *pb) {
  item_t * pItem;
  pthread_mutex_lock(&pb->mutex);
  while (pb->cnt == 0) /*buffer empty*/
    pthread_cond_wait(&pb->notEmpty, &pb->mutex);
  pItem = pb->buf[pb->frst];       /*get data from buffer...*/
  pb->frst++; pb->frst %= BUFSZ; /*... & update first ptr */
  pb->cnt--;
  pthread_cond_signal(&pb->notFull);
  pthread_mutex_unlock(&pb->mutex);
  return pItem;
}

void delay(int secs) { /*utility function*/
  time_t beg = time(NULL), end = beg + secs;
  do {
    ;
  } while (time(NULL) < end);
}

/* core routine for the producer thread */
void *producer(void *n) {
  int j=0;
  char * prodID = (char *)n;
  while (1) { /* loop forever */
  item_t * pItem = (item_t *)malloc(sizeof(item_t));
  pItem->seqNo = j;
  strcpy(pItem->legend, prodID);
    store(pItem, &theBuffer);
    printf("%s stored: %d\n", pItem->legend, j);
    j++;
    delay(rand() % 6);  /* up to 5 sec */
  }
  return n;
}

/* core routine for the consumer thread */
void *consumer(void *n) {
  item_t * pItem;
  char * consID = (char *)n;
  while (1) {  /* loop forever */
    pItem = retrieve(&theBuffer);
    printf("%s retrieved: Seq no %d from %s\n",
          consID, pItem->seqNo, pItem->legend);
    free(pItem);
    delay(rand() % 6);  /* up to 5 sec */
  }
  return(n);
}

int main(int argc, char * argv[]) {
  pthread_attr_t *attr = NULL;
  pthread_t prodThread, consThread;
  char * prodID, * consID;

  int nProds = 1, nCons =  1, k;
  if (argc < 2) {
    printf("Using single prodoucer by default.\n");
  } else {
    nProds = atoi(argv[1]);
    printf("%d prodoucers.\n", nProds);
  }

  if (argc < 3) {
    printf("Using single consumer by default.\n");
  } else {
    nCons = atoi(argv[2]);
    printf("%d consumers.\n", nCons);
  }

  theBuffer.cnt = 0;
  pthread_mutex_init(&theBuffer.mutex, NULL);
  srand(time(NULL)); /* initials the rng */

  for (k=0; k < nProds; k++) {
    prodID = (char *)malloc(LEGENDSZ);
    sprintf(prodID, "PROD%d", k);
    pthread_create(&prodThread, attr, producer, (void *)prodID);
  }

  for (k=0; k < nCons; k++) {
    consID = (char *)malloc(LEGENDSZ);
    sprintf(consID, "CONS%d", k);
    pthread_create(&consThread, attr, consumer, (void *)consID);
  }

  getchar(); //Press ENTER to finish
  return 0;
}
```

Exercise 9.

Explore Diners.c

This is a simulation of the dining philosophers problem. N philosophers sit in a circle around a pot of noodles. Between the philosopher are N forks. A philosopher can think or eat but not at the same time. To eat, a philosopher needs BOTH forks – the forks on either side.

Can you see how this situation might deadlock.
Can you make the simulation Diners.c deadlock?
Look at DinersFixed.c – does this deadlock? What is it about the protocol in Diner.c that makes deadlock possible?

Try devising your own fix to Diners.c using mutexes and condition variables.

Diners.c

```
/* The Dining Philospohers Problem:
N philosophers sit around a circular table. Each philosopher
spends her/his life alternatively thinking and eating.
In the centre of the table is a large plate of spaghetti.
A philosopher needs two forks to eat a helping of spaghetti.
Unfortunately, as philosophy is not as well paid as computing,
the philosophers can only afford N forks. One fork is placed
between each pair of philosophers and they agree that each will
only use the fork to his immediate right and left.

This situation can be simulated using a single Philosopher process
of which N instances run each in their own thread. Each fork is
represented by a data structure which includes a mutex to govern
access to the fork: a philosopher must obtain a mutex lock before
accessing the fork, and release the mutex after.

The problem is that this protocol can deadlock: How? And how can
improve the protocol so that is is deadlock free?
*/

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

/************* Time functions **************/
clock_t simBaseTime = 5000;

clock_t sleepTime() {
  return (clock_t)(rand()%1000)*simBaseTime*2;
}

clock_t eatTime() {
  return (clock_t)(rand()%1000)*simBaseTime;
}

void delay(clock_t drn) { //duration in clock ticks
  clock_t start = clock();
  do
    ; while (clock() < start + drn);
}



/***** Data structure and type definitions ******/
typedef struct {
  pthread_mutex_t mtx;
  int taken, id;
} Fork;

typedef Fork * pFork;

typedef struct {
  int id, stpRqustd, state;
  pFork left, right;
} Philosopher;

typedef Philosopher * pPhil;

/************** Fork Methods **************/
void ForkInit(pFork f, int i) {
  f->taken = 0; f->id = i;
  pthread_mutex_init(&f->mtx, NULL);
}

void put(pFork f) {
  f->taken = 0;
  pthread_mutex_unlock(&f->mtx);
}

void get(pFork f) {
  pthread_mutex_lock(&f->mtx);
  f->taken = 1;
}

/************ Philosopher constants, methods ************/
const int THINKING = 0, HUNGRY = 1, GOTRIGHT = 2,
          EATING = 3, GOTLEFT = 4;

const char * stateDescr[] = { "thinking", "hungry", "got right",
                              "eating", "got left" };

void setState(pPhil ph, int st) {
  static char pad[81];
  int i;
  for (i=0; i<(ph->id)*8; i++)
    pad[i] = ' ';
  pad[i] = '\0';
  printf("%sPh%d: %s\n", pad, ph->id, stateDescr[st]);
  ph->state = st;
}

void PhilosopherInit(pPhil ph, int i, pFork l, pFork r) {
  ph->id = i; ph->left = l; ph->right = r; ph->state = THINKING;
}

void * philRun(void * vp) { /* the philosopher thread function */
  pPhil ph = (pPhil)vp;
  while (!ph->stpRqustd) {
    setState(ph, THINKING);
    delay(sleepTime());
    setState(ph, HUNGRY);
    get(ph->right);
    setState(ph, GOTRIGHT);
    delay(500000);
    get(ph->left);
    setState(ph, EATING);
    delay(eatTime());
    put(ph->right);
    put(ph->left);
  }
  return vp;
}

void stopRequest(pPhil ph) {
  ph->stpRqustd = 1;
}


/*************** Global Data ****************/
int numPhils;
pPhil * phils;
pFork * forks;
```

```
pthread_attr_t * attr;
pthread_t * threads;

/************* Global Functions **************/
void startSim() {
  int i;
  pFork newFork;
  pPhil newPhil;

  time_t seed;
  time(&seed);
  srand(seed);

  attr = NULL;
  phils = (pPhil *)calloc(numPhils, sizeof(pPhil));
  forks = (pFork *)calloc(numPhils, sizeof(pFork));
  threads = (pthread_t *)calloc(numPhils, sizeof(pthread_t));

  for (i =0; i<numPhils; i++) {
    newFork = (pFork)malloc(sizeof(Fork));
    ForkInit(newFork, i);
    forks[i] = newFork;
  }
  for (i =0; i<numPhils; i++) {
    newPhil = (pPhil)malloc(sizeof(Philosopher));
    phils[i] = newPhil;
    PhilosopherInit(newPhil, i, forks[(i-1+numPhils) % numPhils], forks[i]);
    pthread_create(&threads[i], attr, philRun, (void *)newPhil);
  }
}

void stopSim() {
  int i;
  for (i=0; i<numPhils; i++)
    stopRequest(phils[i]);
}

void cleanup() {
  int i;
  for (i=0; i<numPhils; i++) {
    free(forks[i]);
    free(phils[i]);
  }
  free(forks);
  free(phils);
  free(threads);
}

int main(int argc, char ** argv) {
  int i;
  time_t now, beg = time(NULL);

  if (argc < 3) {
    printf("Usage: Diners <num phils> <sim speed 1--100>\n");
    return 0;
  }
  sscanf(argv[1], "%d", &numPhils);
  if (numPhils < 4 || numPhils > 9)
    numPhils = 7;

  sscanf(argv[2], "%d", &i);
  if (i<1 || i>100) i = 1;
  simBaseTime = 5000/i;

  printf("Type ctrl-Z to quit.\n%d philosophers:\n", numPhils);
  startSim();

  while(!feof(stdin)) {
    delay(1000000);
    now = time(NULL);
    printf("%ld seconds elapsed\n", now - beg);
  }
  cleanup();
  printf("Goodbye from the philosphers!\n");
  return 0;
}
```

DinersFixed.c

```
/* The Dining Philospohers Problem:
N philosophers sit around a circular table. Each philosopher
spends her/his life alternatively thinking and eating.
In the centre of the table is a large plate of spaghetti.
A philosopher needs two forks to eat a helping of spaghetti.
Unfortunately, as philosophy is not as well paid as computing,
the philosophers can only afford N forks. One fork is placed
between each pair of philosophers and they agree that each will
only use the fork to his immediate right and left.

This situation can be simulated using a single Philosopher process
of which N instances run each in their own thread. Each fork is
represented by a data structure which includes a mutex to govern
access to the fork: a philosopher must obtain a mutex lock before
accessing the fork, and release the mutex after.

Version of philosopher thread which avoids deadlock: Even-numbered
philosophers take the right then the left fork, off-numbered ones
take the forks in the opposite order.
*/

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

/************* Time functions **************/
clock_t simBaseTime = 5000;

clock_t sleepTime() {
  return (clock_t)(rand()%1000)*simBaseTime*2;
}

clock_t eatTime() {
  return (clock_t)(rand()%1000)*simBaseTime;
}

void delay(clock_t drn) { //duration in clock ticks
  clock_t start = clock();
  do
    ; while (clock() < start + drn);
}



/***** Data structure and type definitions ******/
typedef struct {
  pthread_mutex_t mtx;
  int taken, id;
} Fork;

typedef Fork * pFork;

typedef struct {
  int id, stpRqustd, state;
  pFork left, right;
} Philosopher;
```

```
typedef Philosopher * pPhil;

/************** Fork Methods **************/
void ForkInit(pFork f, int i) {
  f->taken = 0; f->id = i;
  pthread_mutex_init(&f->mtx, NULL);
}

void put(pFork f) {
  f->taken = 0;
  pthread_mutex_unlock(&f->mtx);
}

void get(pFork f) {
  pthread_mutex_lock(&f->mtx);
  f->taken = 1;
}

/************ Philosopher constants, methods ************/
const int THINKING = 0, HUNGRY = 1, GOTRIGHT = 2,
          EATING = 3, GOTLEFT = 4;

const char * stateDescr[] = { "thinking", "hungry", "got right",
                              "eating", "got left" };

void setState(pPhil ph, int st) {
  static char pad[81];
  int i;
  for (i=0; i<(ph->id)*8; i++)
    pad[i] = ' ';
  pad[i] = '\0';
  printf("%sPh%d: %s\n", pad, ph->id, stateDescr[st]);
  ph->state = st;
}

void PhilosopherInit(pPhil ph, int i, pFork l, pFork r) {
  ph->id = i; ph->left = l; ph->right = r; ph->state = THINKING;
}

void * philRun(void * vp) { /* the philosopher thread function */
  pPhil ph = (pPhil)vp;
  int parity = ph->id % 2;  /* Fixed to avoid deadlock: even-numbered  */
  while (!ph->stpRqustd) {  /* philosophers take the right fork first, */
    setState(ph, THINKING); /* odd-numbered ones take the right first. */
    delay(sleepTime());
    setState(ph, HUNGRY);
    if (parity == 0) {
      get(ph->right);
      setState(ph, GOTRIGHT);
    } else {
      get(ph->left);
      setState(ph, GOTLEFT);
    }
    delay(500000);
    if (parity == 0) {
      get(ph->left);
    } else {
      get(ph->right);
    }
    setState(ph, EATING);
    delay(eatTime());
    put(ph->right);
    put(ph->left);
  }
  return vp;
}

void stopRequest(pPhil ph) {
  ph->stpRqustd = 1;
}


/*************** Global Data *****************/
int numPhils;
pPhil * phils;
pFork * forks;
pthread_attr_t * attr;
pthread_t * threads;

/************* Global Functions **************/
void startSim() {
  int i;
  pFork newFork;
  pPhil newPhil;

  time_t seed;
  time(&seed);
  srand(seed);

  attr = NULL;
  phils = (pPhil *)calloc(numPhils, sizeof(pPhil));
  forks = (pFork *)calloc(numPhils, sizeof(pFork));
  threads = (pthread_t *)calloc(numPhils, sizeof(pthread_t));

  for (i =0; i<numPhils; i++) {
    newFork = (pFork)malloc(sizeof(Fork));
    ForkInit(newFork, i);
    forks[i] = newFork;
  }
  for (i =0; i<numPhils; i++) {
    newPhil = (pPhil)malloc(sizeof(Philosopher));
    phils[i] = newPhil;
    PhilosopherInit(newPhil, i, forks[(i-1+numPhils) % numPhils], forks[i]);
    pthread_create(&threads[i], attr, philRun, (void *)newPhil);
  }
}

void stopSim() {
  int i;
  for (i=0; i<numPhils; i++)
    stopRequest(phils[i]);
}

void cleanup() {
  int i;
  for (i=0; i<numPhils; i++) {
    free(forks[i]);
    free(phils[i]);
  }
  free(forks);
  free(phils);
  free(threads);
}

int main(int argc, char ** argv) {
  int i;
  time_t now, beg = time(NULL);

  if (argc < 3) {
    printf("Usage: Diners <num phils> <sim speed 1--100>\n");
    return 0;
  }
  sscanf(argv[1], "%d", &numPhils);
  if (numPhils < 4 || numPhils > 9)
    numPhils = 7;
  sscanf(argv[2], "%d", &i);
  if (i<1 || i>100) i = 1;
  simBaseTime = 5000/i;

  printf("Type ctrl-Z to quit.\n%d philosophers:\n", numPhils);
  startSim();

  while(!feof(stdin)) {
    delay(1000000);
    now = time(NULL);
    printf("%ld seconds elapsed\n", now - beg);
  }
  cleanup();
  printf("Goodbye from the philosphers!\n");
  return 0;
}
```

Exercise 10: - A deadlock-free dining philosophers simulation using condition variables.

The dining philosophers simulation above was made deadlock free in a slightly contrived way, by reversing the order in wich alternate philosophers pick their forks.

The following gives an example of an alternative approach using condition variables. The forks are not modelled in this simulation, just the philosophers, and a philosopher does not attempt to eat unless both neighbours are NOT eating. This is implemented by means of a condition variable for each philosopher.

The eatIfOk function checks the status of both neighbours (with the global mutex locked) and signals the OkToEat condition variable if the above conditions are met. Function dpPickup() (fork) calls eatIfOk() then forces the philosopher to wait of its okToEat condition repeatedly while not EATING. Note that dpPutDown() (fork) calls eatIfOk() on each of the neighbours.

Investigate for various numbers of philosophers.

Try commenting out all the condition wait and signal calls -- can it deadlock then? Give an example of how.

Build the simulation using 'make' at a command prompt in its directory.

```
/* dpmonitor.h
 *
 * The dining philosophers using a monitor
 * implemented with Pthreads
 *
 */
enum {N_PHIL = 25};

void dpInit(void);        // Initialise the monitor
void dpPrintstate(int i); // Print the monitor state
void dpPickup(int i);     // Pick up the resource
void dpPutdown(int i);    // Put down the resource

          ------------------------------------------------------
```

```
/*
 * dpmonitor.c
 * Dining philosophers monitor
 */
#include <assert.h>
#include <sys/types.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "dpmonitor.h"

/************ Type declarations **********************/
typedef enum {THINKING, HUNGRY, EATING} state_t;


/************ Local function prototypes **************/
static void eatIfOk(int i);
static int leftNghbr(int i);
static int rightNghbr(int i);

/*********** Monitor variables **********************/
static pthread_mutex_t dpMutex;
static pthread_cond_t okToEat[N_PHIL];
static state_t state[N_PHIL];
extern int nPhil;



/*********** Monitor function definitions ***********/
void dpPickup(int i) {
  pthread_mutex_lock(&dpMutex);
  state[i] = HUNGRY;
  eatIfOk(i);
  while (state[i] != EATING) {
    pthread_cond_wait(&okToEat[i], &dpMutex);
  }
  pthread_mutex_unlock(&dpMutex);
}

void dpPutdown(int i) {
  pthread_mutex_lock(&dpMutex);
  state[i] = THINKING;
  eatIfOk(rightNghbr(i));
  eatIfOk(leftNghbr(i));
  pthread_mutex_unlock(&dpMutex);
}

/* Prints out state of philosophers as, say, TEHHE, meaning
 * that philosopher 0 is THINKING, philosophers 1 and 4 are
 * EATING, and philosophers 2 and 3 are HUNGRY.
 */
void dpPrintstate(int i){
  static char stat[] = "THE";

  pthread_mutex_lock(&dpMutex);
  printf("%02d:", i);
  for (int i=0; i<nPhil; i++) {
    printf("%c", stat[(int)(state[i])]);
  }
  printf("\n");
  pthread_mutex_unlock(&dpMutex);
}

/*
 * Initialise the monitor mutex, state and condition variables
 * N.B. No mutex to lock and unlock in this function
 */
```

```
void dpInit(void) {
  int rc;

  // Initialise the monitor mutex
  rc = pthread_mutex_init(&dpMutex, NULL);
  assert(rc == 0);

  // Initialise the state and the condition variables
  for (int i=0; i<nPhil; i+=1) {
    state[i] = THINKING;
    rc = pthread_cond_init(&okToEat[i], NULL);
    assert(rc == 0);
  }
}

/***************** Local function definitions ********************/
static void  eatIfOk(int i) {
  if ((state[i] == HUNGRY) &&
      (state[rightNghbr(i)] != EATING) &&
      (state[leftNghbr(i)] != EATING)) {
    state[i] = EATING;
    pthread_cond_signal(&okToEat[i]);
  }
}

static int leftNghbr(int i) {
  return ((i+(nPhil-1)) % nPhil);
}

static int rightNghbr(int i) {
  return ((i+1) % nPhil);
}
          -------------------------------------------------------
/*
 * main.c
 * Dining philosophers
 */

#include <assert.h>
#include <unistd.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <pthread.h>
#include <string.h>
#include <time.h>
#include "dpmonitor.h"

enum {N_ITERATIONS = 1000};
int nItns = N_ITERATIONS;
int nPhil = N_PHIL;
int spd = 10;

static void *philosopher(void *arg);
static void beginThreads(void);
static void endThreads(void);
static void randomDelay(int);

static pthread_t thread[N_PHIL];

//Usage dp <num philosophers> <num iterations> <speed (1-100)>
//Missing or out of range run-time arguments will default to
//  N_PHIL = 25, N_ITERATIONS = 1000, speed 10int main(int argc, char
*argv[]) {
  if (argc >= 2) {
    sscanf(argv[1], "%d", &nPhil);
    if (nPhil < 4 || nPhil > N_PHIL) {
      printf("Num philosophers out of range: using default\n");
      nPhil = N_PHIL;
    }
  }
  if (argc >= 3) {
    sscanf(argv[2], "%d", &nItns);
    if (nItns < 4 || nItns > N_ITERATIONS) {
      printf("Number of iterations out of range: using default\n");
      nItns = N_ITERATIONS;
    }
  }
  if (argc >= 4) {
    sscanf(argv[3], "%d", &spd);
    if (spd < 1 || spd > 100) {
      printf("Speed out of range: using default\n");
      spd = 10;
    }
  }
  dpInit();
  beginThreads();
  endThreads();
  printf("\nFINISHED\n");
  return 0;
}

/*
 * The function implementing the philosopher behaviour.
 * The parameter 'arg' identifies the philosopher: 0, 1, 2, ...
 */
static void *philosopher(void *arg) {
 long id = (long)(arg);

  for (int n = 0; n < nItns; n+=1) {
    randomDelay(100/spd);
    dpPickup(id);
    dpPrintstate(id);
    randomDelay(100/spd);
    dpPutdown(id);
  }
  pthread_exit(NULL);
}

static void beginThreads(void) {
  int rc;
 for (long i=0; i<nPhil; i+=1) {
    rc = pthread_create(&thread[i], NULL,
                    philosopher, (void *)i);
    assert(rc == 0);
  }
}

static void endThreads(void) {
  int rc;

  for (int i=0; i<nPhil; i+=1) {
    rc = pthread_join(thread[i], NULL);
    assert(rc == 0);
  }
}


static void randomDelay(int t) {
  struct timespec delay = {0, (rand() % 49 + 1) * 2000000 * t};
 nanosleep(&delay, NULL);
}
```

Make file :

```
all: dp

dp: dpmonitor.o main.o
	gcc -Wall -pthread -std=c99 -D_POSIX_C_SOURCE=199309L -o dp
dpmonitor.o main.o

dpmonitor.o: dpmonitor.h dpmonitor.c
	gcc -Wall -pthread -std=c99 -D_POSIX_C_SOURCE=199309L -c dpmonitor.c

main.o: main.c dpmonitor.h
	gcc -Wall -pthread -std=c99 -D_POSIX_C_SOURCE=199309L -c main.c

clean:
	-rm dp *.o *~
```