```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS       5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS       5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS      5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}


#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS      4

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
```

```
  printf("Thread %ld starting...\n",tid);
  for (i=0; i<1000000; i++)
  {
    result = result + sin(i) * tan(i);
  }
  printf("Thread %ld done. Result = %e\n",tid, result);
  pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int rc;
  long t;
  void *status;

  /* Initialize and set thread detached attribute */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  for(t=0; t<NUM_THREADS; t++) {
    printf("Main: creating thread %ld\n", t);
    rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
      }
    }

  /* Free attribute and wait for the other threads */
  pthread_attr_destroy(&attr);
  for(t=0; t<NUM_THREADS; t++) {
    rc = pthread_join(thread[t], &status);
    if (rc) {
      printf("ERROR; return code from pthread_join() is %d\n", rc);
      exit(-1);
      }
    printf("Main: completed join with thread %ld having a status of %ld\n",t,(long)status);
    }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS      4

void *BusyWork(void *t)
{
  int i;
  long tid;
  double result=0.0;
  tid = (long)t;
  printf("Thread %ld starting...\n",tid);
  for (i=0; i<1000000; i++)
  {
    result = result + sin(i) * tan(i);
  }
  printf("Thread %ld done. Result = %e\n",tid, result);
  pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int rc;
  long t;
  void *status;

  /* Initialize and set thread detached attribute */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  for(t=0; t<NUM_THREADS; t++) {
    printf("Main: creating thread %ld\n", t);
    rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
      }
    }

  /* Free attribute and wait for the other threads */
  pthread_attr_destroy(&attr);
```

```
  for(t=0; t<NUM_THREADS; t++) {
    rc = pthread_join(thread[t], &status);
    if (rc) {
      printf("ERROR; return code from pthread_join() is %d\n", rc);
      exit(-1);
      }
    printf("Main: completed join with thread %ld having a status of %ld\n",t,(long)status);
    }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}



/******************************************************************************
* FILE: detached.c
* DESCRIPTION:
*   This example demonstrates how to explicitly create a thread in a
*   detached state. This might be done to conserve some system resources
*   if the thread never needs to join later. Compare with the join.c program
*   where the threads are created joinable.
* AUTHOR: 01/30/08 Blaise Barney
* LAST REVISED: 01/29/09
******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS      4

void *BusyWork(void *t)
{
  long i, tid;
  double result=0.0;
  tid = (long)t;
  printf("Thread %ld starting...\n",tid);
  for (i=0; i<1000000; i++) {
    result = result + sin(i) * tan(i);
    }
  printf("Thread %ld done. Result = %e\n",tid, result);
}

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

```
{
pthread_t thread[NUM_THREADS];
pthread_attr_t attr;
int rc;
long t;

/* Initialize and set thread detached attribute */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

for(t=0;t<NUM_THREADS;t++) {
   printf("Main: creating thread %ld\n", t);
   rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
   if (rc) {
     printf("ERROR; return code from pthread_create() is %d\n", rc);
     exit(-1);
     }
 }

/* We're done with the attribute object, so we can destroy it */
pthread_attr_destroy(&attr);

/* The main thread is done, so we need to call pthread_exit explicitly to
*  permit the working threads to continue even after main completes.
*/
printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);

/******************************************************************************
* FILE: condvar.c
* DESCRIPTION:
*   Example code for using Pthreads condition variables.  The main thread
*   creates three threads.  Two of those threads increment a "count" variable,
*   while the third thread watches the value of "count".  When "count"
*   reaches a predefined limit, the waiting thread is signaled by one of the
*   incrementing threads. The waiting thread "awakens" and then modifies
*   count. The program continues until the incrementing threads reach
*   TCOUNT. The main program prints the final value of count.
* SOURCE: Adapted from example code in "Pthreads Programming", B. Nichols
*   et al. O'Reilly and Associates.
* LAST REVISED: 03/07/17  Blaise Barney
******************************************************************************/
#include <pthread.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when condition is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      printf("inc_count(): thread %ld, count = %d  Threshold reached. ",my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal.\n");
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
    sleep(1);
    }
  pthread_exit(NULL);
}

void *watch_count(void *t)
{
  long my_id = (long)t;

  printf("Starting watch_count(): thread %ld\n", my_id);
```

```
  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait routine
  will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    printf("watch_count(): thread %ld Count= %d. Going into wait...\n", my_id,count);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received. Count= %d\n", my_id,count);
    }
  printf("watch_count(): thread %ld Updating the value of count...\n", my_id);
  count += 125;
  printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
  printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
```

```
  printf ("Main(): Waited and joined with %d threads. Final value of count = %d. Done.\n",
NUM_THREADS, count);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit (NULL);

}
```

Td2
Exo1

```
/* A compiler avec gcc -o E12 -pthread E12.c */
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<sys/types.h>
#include<unistd.h>

void * helloworld(void *arg) {
printf("Hello world! pid=%d pthread_self=%p\n", getpid(), (void *) pthread_self());
return NULL;
 }

int main(int argc, char** argv) {
int i, nb;
pthread_t * threads;
nb = atoi(argv[1]);
//threads = malloc(nb * sizeof(pthread_t));
threads=(pthread_t*)malloc(sizeof(pthread_t)*nb);

for (i = 0; i < nb; i++) {
pthread_create(&threads[i], NULL, helloworld, NULL);
 }

 printf("Début de l'attente\n");

 for (i = 0; i < nb; i++) {
 pthread_join(threads[i], NULL);
 }
```

```
 printf("Fin de l'attente\n");
return 0;
}
```

Exo2

```
/* A compiler avec gcc -o E12 -pthread E12.c */
 #include <stdio.h>
 #include <stdlib.h>
 #include <pthread.h>
 #include <sys/types.h>
 #include <unistd.h>


 void * helloworld(void * arg) {
 printf("Hello world! i=%d\n",* (int*)arg);
 return NULL;
 }

 int main(int argc, char* argv[]) {
 int i, nb;
 int * args;
 pthread_t * threads;
 nb = atoi(argv[1]);
 threads = malloc(nb * sizeof(pthread_t));
 args = malloc(nb * sizeof(int));
 for (i = 0; i < nb; i++) {
 args[i] = i;
 pthread_create( &threads[i], NULL, helloworld, &args[i]);
 //pthread_create( &threads[i], NULL, helloworld, &i);
 }
 printf("Start waiting\n");
 for (i = 0; i < nb; i++) {
 pthread_join(threads[i], NULL);
 }
 printf("End Waiting\n");
 return 0;
 }
```

Exo 3-4

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

```
#include <sys/types.h>
#include <unistd.h>

struct threadargs {
   int i;
   int * somme;
   pthread_mutex_t * mutex;
};

void * helloworld(void * arg) {
   struct threadargs * ta = (struct threadargs *) arg;
   int i;
   for (i = 0; i < 1000000; i++) {
      pthread_mutex_lock(ta->mutex);
      *ta->somme += ta->i;
      pthread_mutex_unlock(ta->mutex);
   }
   return NULL;
}

int main(int argc, char** argv) {
   int i, nb;
   struct threadargs * args;
   int somme = 0;
   pthread_mutex_t mutex;
   pthread_t * threads;
   nb = atoi(argv[1]);
   threads = malloc(nb * sizeof(pthread_t));
   args = malloc(nb * sizeof(struct threadargs));
   pthread_mutex_init(&mutex, NULL);
   for (i = 0; i < nb; i++) {
      args[i].i = i;
      args[i].somme = &somme;
      args[i].mutex = &mutex;
      pthread_create(  &threads[i], NULL, helloworld, &args[i]);
   }
   for (i = 0; i < nb; i++) {
      pthread_join(threads[i], NULL);
   }
   printf("somme=%d\n", somme);
   return 0;
}
```

Passing argument

```
#include <pthread.h>
#include <stdio.h>
void ChildThread(void *argument){

    printf('childthread');
int i,count;
count=(int)argument;
for(i=1;i<=count;++i){
printf('child cound - %d\n',i);
}
pthread_exit(0);
}
int main (void){
pthread_t hThread;
int ret,count=20;
//ret=pthread_create(&hThread,NULL,(void *)ChildThread,NULL);
//7ot argument count
ret=pthread_create(&hThread,NULL,(void *)ChildThread,(void*)count);
 printf("ret = %d" , ret);
if(ret<0){
printf("Thread creation failed \n");
exit(-1);

}
// iza 2emet el join ma byontoron
pthread_join(hThread,NULL);
printf("parent is continuing ...\n");
return 0;
}
```

Race condition

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#define NITERS 1000
void *count(void *arg);
volatile unsigned int cnt=0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; //
int main(){
pthread_t tid1,tid2;
pthread_create(&tid1,NULL,count,NULL);
pthread_create(&tid2,NULL,count,NULL);
pthread_join(tid1,NULL);
pthread_join(tid2,NULL);
printf("cnt: %d\n",cnt);
 // pthread_mutex_destroy(&mutex);(bl statique ma mn7t destroy)
exit(0);
}

 void *count(void*arg){
 volatile int i=0;
 for(;i<NITERS;i++){
        pthread_mutex_lock(&mutex);
    cnt++;
   pthread_mutex_unlock(&mutex);
 }
 return NULL;
 }
 /*
 kel we7de bada te3mol count 10000
 bas error eno 2awal chi tole3 11442990
 problem  Race condition 3amm yetseba2o ta ye5do el count
 solution mutex
 */
```

Exo2

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
void *Allen(void *arg);
```

```
void *Bob(void *arg);
pthread_mutex_t book1=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t book2=PTHREAD_MUTEX_INITIALIZER;
int main() {
 pthread_t tid1, tid2;
 pthread_create(&tid1, NULL, Allen, NULL);
 pthread_create(&tid2, NULL, Bob, NULL);
 pthread_join(tid1, NULL);
 pthread_join(tid2, NULL);
 return 0;
}
//alin ma3o book 1 be7aje la boo2 w bob bel 3akes // problem deadlock
// how we can solve // mutex
void *Allen(void *arg) {
 pthread_mutex_lock(&book1);
 sleep(10);
 pthread_mutex_lock(&book2);
 printf("Allen has collected all books he need, he is going to do homework!");
 pthread_mutex_unlock(&book2);
 pthread_mutex_unlock(&book1);
}
void *Bob(void *arg) {
    // Bob locks in the same order: book1 -> book2
    pthread_mutex_lock(&book1);
    sleep(1);
    pthread_mutex_lock(&book2);
    printf("Bob has collected all books he needs, he is going to do homework!\n");
    pthread_mutex_unlock(&book2);
    pthread_mutex_unlock(&book1);
    return NULL;
}
```