ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΑΚΑΔΗΜΑΪΚΟ ΕΤΟΣ 2016-2017
ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ
ΜΑΘΗΜΑ: ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΕΡΓΑΣΤΗΡΙΟ 6 - ΣΗΜΕΙΩΣΕΙΣ

Νήματα (Threads)

Γενικά, ένα **νήμα (thread)** είναι μια ακολουθία ελέγχου μέσα σε μία διεργασία. Κάθε μια διεργασία έχει τουλάχιστον ένα νήμα εκτέλεσης. Όταν μια διεργασία εκτελεί ένα fork() ένα νέο αντίγραφο της διεργασίας δημιουργείται με τις δικές του μεταβλητές καιτο δικό του PID. Η νέα διεργασία ξεχωριστά από τις υπόλοιπες διαχειρίζεται από τους schedulers. Όταν δημιουργούμε ένα νέο νήμα, αυτό λαμβάνει από το Λειτουργικό Σύστημα μια νέα στοίβα εκτέλεσης αλλά μοιράζεται τις καθολικές μεταβλητές, του περιγραφείς αρχείων, τους χειριστές σημάτων και την τρέχουσα κατάσταση καταλόγου με τη διεργασία που το δημιούργησε.

Η βιβλιοθήκη που περιλαμβάνει την απαραίτητη λειτουργικότητα για τη διαχείριση νημάτων είναι η **pthread**. Όταν συμπεριλαμβάνουμε τη βιβλιοθήκη pthread.h στον κώδικά μας έχουμε τη δυνατότητα να χρησιμοποιήσουμε ένα σύνολο ορισμών και πρωτοτύπων που απαιτείται για να δημιουργήσουμε μια εφαρμογή πολλαπλών νημάτων (multi-thread application). Η δημιουργία ενός νέου νήματος γίνεται με την ακόλουθη συνάρτηση:

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr, void
*(*start_routine)(void *), void *arg);
```

Το πρώτο όρισμα αφορά στο αναγνωριστικό του νήματος μέσω του οποίου μπορούμε να το προσπελάσουμε. Το επόμενο όρισμα αποτελείται από τις ιδιότητες του νήματος. Αν δεν έχουμε κάποια ιδιαίτερη ιδιότητα να δώσουμε, τότε απλά μπορούμε να εισάγουμε το NULL. Σε επόμενα παραδείγματα, θα δείξουμε πως ακριβώς μπορούμε να ορίσουμε τις ιδιότητες ενός νήματος. Τα τελευταία δύο ορίσματα καθορίζουν τη συνάρτηση που θα 'εκτελέσει' το νήμα καθώς και τα ορίσματά της. Σε επιτυχή εκτέλεση της εντολής, θα λάβουμε το 0 ενώ σε περίπτωση λάθους θα λάβουμε ένα κωδικό λάθους.

Όταν ένα νήμα ολοκληρώσει την εκτέλεσή του, τότε θα πρέπει να καλέσουμε την ακόλουθη συνάρτηση:

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Η συνάρτηση τερματίζει το νήμα επιστρέφοντας ένα δείκτη σε ένα αντικείμενο. Αν το αντικείμενο αυτό είναι μια τοπική (στο νήμα) μεταβλητή, τότε θα προκύψει σοβαρό λάθος αφού με τον τερματισμό του νήματος διαγράφονται όλες οι τοπικές μεταβλητές.

Για να προκαλέσουμε την αναμονή τερματισμού ενός νήματος (όπως ακριβώς με την wait στις διεργασίες) υιοθετούμε τη συνάρτηση:

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

Η πρώτη παράμετρος της συνάρτησης είναι το νήμα που πρέπει να αναμείνουμε για τον τερματισμό του (βάζουμε το αναγνωριστικό που μας έδωσε η pthread_create) ενώ το δεύτερο όρισμα είναι ένας δείκτης σε δείκτη που μας δίνει την τιμή επιστροφής. Η συνάρτηση επιστρέφει το 0 σε επιτυχή εκτέλεση ενώ λαμβάνουμε κωδικό λάθους σε αντίθετη περίπτωση.

Code 1

```
//compile with: gcc -pthread -o test test.c
```

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

void *thread_function(void *arg);

char message[] = "Hello World";

int main() {
        int res;
        pthread_t a_thread;
        void *thread_result;
        res = pthread_create(&a_thread, NULL, thread_function, (void *)message);
        if (res != 0) {
                perror("Thread creation failed");
                exit(EXIT_FAILURE);
        }
        printf("Waiting for thread to finish...\n");
        res = pthread_join(a_thread, &thread_result);
        if (res != 0) {
                perror("Thread join failed");
                exit(EXIT_FAILURE);
        }
        printf("Thread joined, it returned %s\n", (char *)thread_result);
        printf("Message is now %s\n", message);
        exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
        printf("thread_function is running. Argument was %s\n", (char *)arg);
        sleep(3);
        strcpy(message, "Bye!");
        pthread_exit("Thank you for the CPU time");
}
```

Το ακόλουθο πρόγραμμα τροποποιεί το Code 1 και υλοποιεί την παράλληλη εκτέλεση μια διεργασίας και ενός νήματος.

Code 2

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

void *thread_function(void *arg);
```

```
char message[] = "Hello World";
int run_now = 1;

int main() {
	int res;
	pthread_t a_thread;
	void *thread_result;
	res = pthread_create(&a_thread, NULL, thread_function, (void *)message);
	if (res != 0) {
		perror("Thread creation failed");
		exit(EXIT_FAILURE);
	}

	int print_count1 = 0;
	while (print_count1++ < 20) {
		if (run_now == 1) {
			printf("1");
			run_now = 2;
		}
		else {
			sleep(1);
		}
	}

	printf("Waiting for thread to finish...\n");
	res = pthread_join(a_thread, &thread_result);
	if (res != 0) {
		perror("Thread join failed");
		exit(EXIT_FAILURE);
	}
	printf("Thread joined, it returned %s\n", (char *)thread_result);
	printf("Message is now %s\n", message);
	exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
	int print_count2 = 0;
	printf("thread_function is running. Argument was %s\n", (char *)arg);
	while(print_count2++ < 20) {
		if (run_now == 2) {
			printf("2");
			run_now = 1;
		}
		else {
			sleep(1);
		}
	}
```

```
        strcpy(message, "Bye!");
        pthread_exit("Thank you for the CPU time");
}
```

Συγχρονισμός Νημάτων

Για να επιτύχουμε το συγχρονισμό διαφόρων νημάτων, θα πρέπει να υιοθετήσουμε τους εξής τρόπους: σημαφόροι και mutexes (mutual exclusion). Ένας mutex είναι στην ουσία ένας δυαδικός σημαφόρος απλά η διαφορά τους είναι στον τρόπο χειρισμού. Μόνο το νήμα που έχει 'κλειδώσει' ένα mutex μπορεί να τον 'ξεκλειδώσει'. Στα ακόλουθα παραδείγματα, υιοθετούμε το συγχρονισμό νημάτων με τη βοήθεια δυαδικών σημαφόρων που παίρνουν τις τιμές 0 ή 1. Στο προηγούμενο εργαστήριο είδαμε τις συναρτήσεις διαχείρισης σημαφόρων που υιοθετούνται με βάση το System V πρότυπο ενώ στο παρόν εργαστήριο θα υιοθετήσουμε συναρτήσεις που ορίζονται στο πρότυπο POSIX. Το ακόλουθο παράδειγμα υλοποιεί συγχρονισμό δύο νημάτων με βάση τις συναρτήσεις κατά System V.

Code 3

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

//create user defined semun for initializing the semaphores

void *Thread1(void *arg)
{
 int semid;
 semid = (int)arg;

 //in order to perform the operations on semaphore
 // first need to define the sembuf object
 struct sembuf op1,op2;

 //operation for 0th semaphore
 op1.sem_num = 0; //signifies 0th semaphore
 op1.sem_op = -1; //reduce the semaphore count to lock
 op1.sem_flg = 0; //wait till we get lock on semaphore

 //operation for 1th semaphore
 op2.sem_num = 1; //signifies 0th semaphore
 op2.sem_op = -1; //reduce the semaphore count to lock
 op2.sem_flg = 0; //wait till we get lock on semaphore

 //locking the 0th semaphore
 if (semop(semid,&op1,1) == -1)
  {
```

```
    pthread_exit("Thread1:semop failure Reason:");
   }
  else
   fprintf(stderr,"Thread1:Successfully locked 0th semaphore\n");
  //lock the 1th semaphore
  if (semop(semid,&op2,1) == -1){

    pthread_exit("Thread1:semop failure Reason:");
   }
  else
   fprintf(stderr,"Thread1:Successfully locked 1th semaphore\n");

  //release the 0th semaphore
  op1.sem_num = 0; //signifies 0th semaphore
  op1.sem_op = 1; //reduce the semaphore count to lock
  op1.sem_flg = 0; //wait till we get lock on semaphore

  if (semop(semid,&op1,1) == -1)
   {
    pthread_exit("Thread1:semop failure Reason:");
   }
  else
   fprintf(stderr,"Thread1:Successfully unlocked 0th semaphore\n");

  //release the 1th semaphore
  op2.sem_num = 1; //signifies 0th semaphore
  op2.sem_op = 1; //reduce the semaphore count to lock
  op2.sem_flg = 0; //wait till we get lock on semaphore

  if (semop(semid,&op2,1) == -1)
   {
    pthread_exit("Thread1:semop failure Reason:");
   }
  else
   fprintf(stderr,"Thread1:Successfully unlocked 1th semaphore\n");
}

void *Thread2(void *arg)
{
  int semid;
  semid = (int)arg;

  //in order to perform the operations on semaphore
  // first need to define the sembuf object
  struct sembuf op1,op2;

  //operation for 0th semaphore
  op1.sem_num = 0; //signifies 0th semaphore
```

```
op1.sem_op = -1; //reduce the semaphore count to lock
op1.sem_flg = 0; //wait till we get lock on semaphore

//operation for 1th semaphore
op2.sem_num = 1; //signifies 0th semaphore
op2.sem_op = -1; //reduce the semaphore count to lock
op2.sem_flg = 0; //wait till we get lock on semaphore

//lock the 0th semaphore
if (semop(semid,&op1,1) == -1)
 {
   pthread_exit("Thread2:semop failure Reason:");
 }
else
 fprintf(stderr,"Thread2:Successfully locked 0th semaphore\n");

//lock the 1th semaphore
if (semop(semid,&op2,1) == -1)
 {
        pthread_exit("Thread2:semop failure Reason:");
 }
else
 fprintf(stderr,"Thread2:Successfully locked 1th semaphore\n");

//release 0th semaphore

op1.sem_num = 0; //signifies 0th semaphore
op1.sem_op = 1; //reduce the semaphore count to lock
op1.sem_flg = 0; //wait till we get lock on semaphore

if (semop(semid,&op1,1) == -1)
 {

   pthread_exit("Thread2:semop failure Reason:");
 }
else
 fprintf(stderr,"Thread2:Successfully unlocked 0th semaphore\n");

//release the 1th semaphore
op2.sem_num = 1; //signifies 0th semaphore
op2.sem_op = 1; //reduce the semaphore count to lock
op2.sem_flg = 0; //wait till we get lock on semaphore

if (semop(semid,&op2,1) == -1)
 {
   pthread_exit("Thread2:semop failure Reason:");
 }
else
```

```
   fprintf(stderr,"Thread2:Successfully unlocked 1th semaphore\n");

}

int main()
{
 pthread_t tid1,tid2;
 int semid;

 //create user defined semun for initializing the semaphores

 typedef union semun
 {
  int val;
  struct semid_ds *buf;
  ushort * array;
 }semun_t;

 semun_t arg;
 semun_t arg1;

 //creating semaphore object with two semaphore in a set
 //viz 0th & 1th semaphore
 semid = semget(IPC_PRIVATE,2,0666|IPC_CREAT);
 if(semid<0)
  {
   perror("semget failed Reason:");
   exit(EXIT_FAILURE);
  }

 //initialize 0th semaphore in the set to value 1
 arg.val = 1;
 if ( semctl(semid,0,SETVAL,arg)<0 )
  {
   perror("semctl failure Reason:");
   exit(EXIT_FAILURE);
  }
 //initialize 1th semaphore in the set to value 1
 arg1.val = 1;
 if( semctl(semid,1,SETVAL,arg1)<0 )
  {
   perror("semctl failure Reason: ");
   exit(EXIT_FAILURE);
  }

 //create two threads to work on these semaphores
 if(pthread_create(&tid1, NULL,Thread1, semid))
  {
```

```
    printf("\n ERROR creating thread 1");
    exit(EXIT_FAILURE);
   }
 if(pthread_create(&tid2, NULL,Thread2, semid) )
  {
    printf("\n ERROR creating thread 2");
    exit(EXIT_FAILURE);
   }
 //waiting on these threads to complete
 pthread_join(tid1, NULL);
 pthread_join(tid2, NULL);

 //once done clear the semaphore set
 if (semctl(semid, 1, IPC_RMID ) == -1 )
  {
    perror("semctl failure while clearing Reason:");
    exit(EXIT_FAILURE);
   }
 //exit the main threads
 pthread_exit(NULL);
 return 0;
}
```

Σχετικά με το πρότυπο POSIX διατίθενται τέσσερις συναρτήσεις:

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t * sem);

int sem_post(sem_t * sem);
int sem_destroy(sem_t * sem);
```

Η sem_init χρησιμοποιείται για τη δημιουργία των σημαφόρων αποδίδοντας τους ένα ακέραιο αριθμό. Αν το pshared είναι 0 τότε ο σημαφόρος είναι τοπικός στην τρέχουσα διεργασία ενώ σε διαφορετική περίπτωση ο σημαφόρος διαμοιράζεται σε πολλές διεργασίες.

Οι sem_wait και sem_post παίρνουν ένα δείκτη προς το σημαφόρο ώστε να εκτελέσουν κάποιες ενέργειες πάνω στους σημαφόρους. Η sem_post αυξάνει την τιμή του σημαφόρου κατά 1. Η sem_wait μειώνει την τιμή του σημαφόρου κατά 1 αλλά αναμένει μέχρι ο σημαφόρος να έχει τιμή μεγαλύτερη του 0. Τέλος, η sem_destroy παίρνει σαν όρισμα το δείκτη προς ένα σημαφόρο και απελευθερώνει τους πόρους που σχετίζονται με το σημαφόρο. Αν κάποιο νήμα αναμένει για το σημαφόρο, η συγκεκριμένη συνάρτηση θα επιστρέψει σφάλμα.

Code 4

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
```

```
void *thread_function(void *arg);
sem_t bin_sem;
#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
        int res;
        pthread_t a_thread;
        void *thread_result;
        res = sem_init(&bin_sem, 0, 0);
        if (res != 0) {
                perror("Semaphore initialization failed");
                exit(EXIT_FAILURE);
        }
        res = pthread_create(&a_thread, NULL, thread_function, NULL);
        if (res != 0) {
                perror("Thread creation failed");
                exit(EXIT_FAILURE);
        }
        printf("Input some text. Enter 'end' to finish\n");
        while(strncmp("end", work_area, 3) != 0) {
                        fgets(work_area, WORK_SIZE, stdin);
                        sem_post(&bin_sem);
        }
        printf("\nWaiting for thread to finish...\n");
        res = pthread_join(a_thread, &thread_result);
        if (res != 0) {
                perror("Thread join failed");
                exit(EXIT_FAILURE);
        }
        printf("Thread joined\n");
        sem_destroy(&bin_sem);
        exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
        sem_wait(&bin_sem);
        while(strncmp("end", work_area, 3) != 0) {
                printf("You input %d characters\n", strlen(work_area) -1);
                sem_wait(&bin_sem);
        }
        pthread_exit(NULL);
}
```

Συγχρονισμός με Mutexes

Η ιδέα του συγχρονισμού με mutexes είναι να κλειδώσουμε ένα πόρο έτσι ώστε μόνο ένα νήμα να είναι σε θέση να χρησιμοποιεί τον πόρο αυτό. Οι συναρτήσεις που μπορούμε να χρησιμοποιήσουμε για το συγχρονισμό με mutexes είναι οι ακόλουθες:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*mutexattr);

int pthread_mutex_lock(pthread_mutex_t *mutex));

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Η λειτουργία των συναρτήσεων είναι παραπλήσια με τις προηγούμενες συναρτήσεις που είδαμε για τους σημαφόρους.

Code 5

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
pthread_mutex_t work_mutex; /* protects both work_area and time_to_exit */
#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;

int main() {
        int res;
        pthread_t a_thread;
        void *thread_result;
        res = pthread_mutex_init(&work_mutex, NULL);
        if (res != 0) {
                perror("Mutex initialization failed");
                exit(EXIT_FAILURE);
        }
        res = pthread_create(&a_thread, NULL, thread_function, NULL);
        if (res != 0) {
                perror("Thread creation failed");
                exit(EXIT_FAILURE);
        }
        pthread_mutex_lock(&work_mutex);
        printf("Input some text. Enter 'end' to finish\n");
        while(!time_to_exit) {
                fgets(work_area, WORK_SIZE, stdin);
                pthread_mutex_unlock(&work_mutex);
                while(1) {
                        pthread_mutex_lock(&work_mutex);
                        if (work_area[0] != '\0') {
```

```
                        pthread_mutex_unlock(&work_mutex);
                        sleep(1);
                }
                else {
                        break;
                }
        }
    }
    pthread_mutex_unlock(&work_mutex);

    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    pthread_mutex_destroy(&work_mutex);
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    sleep(1);
    pthread_mutex_lock(&work_mutex);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        work_area[0] = '\0';
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
        pthread_mutex_lock(&work_mutex);
        while (work_area[0] == '\0' ) {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
            pthread_mutex_lock(&work_mutex);
        }
    }
    time_to_exit = 1;
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    pthread_exit(0);
}
```

Ορισμένες φορές θέλουμε να σταματήσουμε/τερματίσουμε ένα νήμα πριν το πέρας της εκτέλεσής του. Η συνάρτηση που μπορούμε να υιοθετήσουμε είναι η:

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

η οποία παίρνει το ID του νήματος σαν παράμετρο. Επιπρόσθετα, μπορούμε να θέσουμε την κατάσταση ενός νήματος σε ακύρωση με την ακόλουθη συνάρτηση:

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

Η πρώτη παράμετρος είναι είτε PTHREAD_CANCEL_ENABLE που επιτρέπει το νήμα να ακυρωθεί ή PTHREAD_CANCEL_DISABLE που δηλώνει πως αιτήσεις ακύρωσης θα αγνοηθούν. Η oldstate pointer επιτρέπει την προηγούμενη κατάσταση να ανακτηθεί. Αν δεν μας ενδιαφέρει η προηγούμενη κατάσταση, απλά θέτουμε ως δεύτερη παράμετρο το NULL. Αν η αίτηση ακύρωσης γίνει αποδεκτή τότε μπαίνουμε σε ένα δεύτερο επίπεδο ελέγχου του νήματος, τον τύπο ακύρωσης που τίθεται με τη βοήθεια της ακόλουθης συνάρτησης:

```
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);
```

Η πρώτη παράμετρος μπορεί να πάρει τις ακόλουθες τιμές: PTHREAD_CANCEL_ASYNCHRONOUS που άμεσα εκτελεί τις αιτήσεις ακύρωσης και την την PTHREAD_CANCEL_DEFERRED που κάνει τις αιτήσεις ακύρωσης να αναμείνουν μέχρι να εκτελεστεί μια από τις pthread_join, pthread_cond_wait, pthread_cond_timedwait, pthread_testcancel, sem_wait, ή sigwait.

Code 6

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

int main() {
        int res;
        pthread_t a_thread;
        void *thread_result;
        res = pthread_create(&a_thread, NULL, thread_function, NULL);
        if (res != 0) {
                perror("Thread creation failed");
                exit(EXIT_FAILURE);
        }
        sleep(3);
        printf("Canceling thread...\n");
        res = pthread_cancel(a_thread);
        if (res != 0) {
                perror("Thread cancelation failed");
                exit(EXIT_FAILURE);
        }
        printf("Waiting for thread to finish...\n");
        res = pthread_join(a_thread, &thread_result);
        if (res != 0) {
                perror("Thread join failed");
                exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}
```

```
void *thread_function(void *arg) {
	int i, res;
	res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
	if (res != 0) {
		perror("Thread pthread_setcancelstate failed");
		exit(EXIT_FAILURE);
	}
	res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
	if (res != 0) {
		perror("Thread pthread_setcanceltype failed");
		exit(EXIT_FAILURE);
	}
	printf("thread_function is running\n");
	for(i = 0; i < 10; i++) {
		printf("Thread is still running (%d)...\n", i);
		sleep(1);
	}
	pthread_exit(0);
}
```

Ακολουθεί ένα παράδειγνμα δημιουργίας και διαχείρισης πολλαπλών νημάτων.

Code 7

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 6

void *thread_function(void *arg);

int main() {
	int res;
	pthread_t a_thread[NUM_THREADS];
	void *thread_result;
	int lots_of_threads;
	for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++) {
		res = pthread_create(&(a_thread[lots_of_threads]),
		NULL, thread_function, (void *)&lots_of_threads);
		if (res != 0) {
			perror("Thread creation failed");
			exit(EXIT_FAILURE);
		}
		sleep(1);
	}
	printf("Waiting for threads to finish...\n");
	for(lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0; lots_of_threads--) {
		res = pthread_join(a_thread[lots_of_threads], &thread_result);
```

```
                if (res == 0) {
                        printf("Picked up a thread\n");
                }
                else {
                        perror("pthread_join failed");
                }
        }
        printf("All done\n");
        exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
        int my_number = *(int *)arg;
        int rand_num;
        printf("thread_function is running. Argument was %d\n", my_number);
        rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
        sleep(rand_num);
        printf("Bye from %d\n", my_number);
        pthread_exit(NULL);
}
```