# POSIX Threads

# Contents

Creating new threads within a process

Synchronizing data access between threads in a single process

Modifying the attributes of a thread

Controlling one thread from another in the same process

# Advantages and Drawbacks of Threads

Advantages

- Sometimes it is very useful to make a program appear to do two things at once.
- The performance of an application that mixes input, calculation, and output may be improved by running these as three separate threads.
- Now that multi-cored CPUs are common, multiple threads inside a process enable a single process to better utilize the hardware resources available.
- Switching between threads requires the OS to do much less work than switching between processes.

Drawbacks

- Writing multithreaded programs requires very careful design.
- Debugging a multithreaded program is much, much harder that debugging a single-threaded one, because the interactions between the threads are very hard to control.
- A program that splits a large calculation into two and runs the two parts as different threads will not necessarily run more quickly on a single processor machine.

# The First Threads Program

Re-entrant code

- Macro `_REENTRANT` before any `#include` lines in your program
  - Some functions get prototypes for a re-entrant safe equivalent.
  - Some `stdio.h` functions that are normally implemented as macros become proper re-entrant safe fucnctions
  - The variable `errno`, from `errno.h`, is changed to call a function, which can determine the real `errno` value in a multithread safe way.

```
#include <pthread.h>

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

void pthread_exit(void *retval);

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

# The First Threads Program

```
#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);
    }
```

# The First Threads Program

```
    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");
}
```

```
TARGET = thread1
SRC = thread1.c
OBJ = thread1.o
CC = gcc
LDFLAGS = -pthread -lpthread
CFLAGS = -D_REENTRANT

all: $(OBJ)
    $(CC) -o $(TARGET) $(LDFLAGS) $(OBJ)

.c.o:
    $(CC) -c $(CFLAGS) $<

clean:
    rm -rf *.o $(TARGET)

~
~
"Makefile" 17L, 222C written                                  4,15          All
```

# The First Threads Program

```
$ ./thread1
Waiting for thread to finish...
thread_function is running. Argument was Hello World
Thread joined, it returned Thank you for the CPU time
Message is now Bye!
```

# Simultaneous Execution

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

#define LOOP_COUNTER 100000

void *thread_function1(void *arg);
void *thread_function2(void *arg);

int counter;

int main()
{
    int res1, res2;
    pthread_t a_thread1, a_thread2;
    void *thread_result;

    res1 = pthread_create(&a_thread1, NULL, thread_function1, NULL);
    res2 = pthread_create(&a_thread2, NULL, thread_function2, NULL);
    if (res1 != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    if (res2 != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    res1 = pthread_join(a_thread1, &thread_result);
    res2 = pthread_join(a_thread2, &thread_result);

    printf("main: Final counter: %d\n", counter);

    exit(EXIT_SUCCESS);
}
```

```c
void *thread_function1(void *arg)
{
    int i = 0;

    for (i= 0; i < LOOP_COUNTER; i++) {
        counter = counter + 1;
        //printf("Thread1: Counter = %d\n", counter);
    }
    pthread_exit("Thank you for the CPU time");
}

void *thread_function2(void *arg)
{
    int i = 0;

    for (i= 0; i < LOOP_COUNTER; i++) {
        counter = counter - 1;
        //printf("Thread2: Counter = %d\n", counter);
    }
```

# Simultaneous Execution

```
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./thread2
main: Final counter: 100000
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./thread2
main: Final counter: 0
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./thread2
main: Final counter: -98907
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./thread2
main: Final counter: -99192
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./thread2
main: Final counter: -98345
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./thread2
main: Final counter: -99086
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./thread2
main: Final counter: -27999
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./thread2
main: Final counter: -99826
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./thread2
main: Final counter: -99697
```

What is the problem?

# Synchronization

Mutual exclusion with Semaphores

```
#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);
```

# Synchronization

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

#define LOOP_COUNTER 100000

void *thread_function1(void *arg);
void *thread_function2(void *arg);

int counter;
sem_t bin_sem;

int main()
{
    int res1, res2;
    pthread_t a_thread1, a_thread2;
    void *thread_result;

    res1 = sem_init(&bin_sem, 0, 1);
    if (res1 != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }

    res1 = pthread_create(&a_thread1, NULL, thread_function1, NULL);
    res2 = pthread_create(&a_thread2, NULL, thread_function2, NULL);
    if (res1 != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    if (res2 != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    res1 = pthread_join(a_thread1, &thread_result);
    res2 = pthread_join(a_thread2, &thread_result);

    printf("main: Final counter: %d\n", counter);

    sem_destroy(&bin_sem);

    exit(EXIT_SUCCESS);
}

void *thread_function1(void *arg)
{
    int i = 0;

    for (i= 0; i < LOOP_COUNTER; i++) {
        sem_wait(&bin_sem);
        counter = counter + 1;
        sem_post(&bin_sem);
        //printf("Thread1: Counter = %d\n", counter);
    }
    pthread_exit("Thank you for the CPU time");
}

void *thread_function2(void *arg)
{
    int i = 0;

    for (i= 0; i < LOOP_COUNTER; i++) {
        sem_wait(&bin_sem);
        counter = counter - 1;
        sem_post(&bin_sem);
        //printf("Thread2: Counter = %d\n", counter);
    }
    pthread_exit("Thank you for the CPU time");
}
```

# Synchronization

```
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./test_sem
main: Final counter: 0
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./test_sem
main: Final counter: 0
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./test_sem
main: Final counter: 0
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./test_sem
main: Final counter: 0
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$ ./test_sem
main: Final counter: 0
jcho@ACES-WS:~/Projects/programming/linux_sys_prog/ex02_sem$
```

# Synchronization

Mutual exclusion with Mutex

```
#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);
```

# Synchronization

## Synchronization with Semaphores

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

#define LOOP_COUNTER 1000

void *thread_function1(void *arg);
void *thread_function2(void *arg);

int counter;
sem_t bin_sem;
sem_t bin_sem_th1;
sem_t bin_sem_th2;

int main()
{
    int res1, res2;
    pthread_t a_thread1, a_thread2;
    void *thread_result;

    res1 = sem_init(&bin_sem, 0, 1);
    if (res1 != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }

    res1 = sem_init(&bin_sem_th1, 0, 1);
    if (res1 != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }

    res1 = sem_init(&bin_sem_th2, 0, 0);
    if (res1 != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }

    res1 = pthread_join(a_thread1, &thread_result);
    res2 = pthread_join(a_thread2, &thread_result);

    printf("main: Final counter: %d\n", counter);

    sem_destroy(&bin_sem);

    exit(EXIT_SUCCESS);
}

void *thread_function1(void *arg)
{
    int i = 0;

    for (i= 0; i < LOOP_COUNTER; i++) {
        sem_wait(&bin_sem_th1);
        sem_wait(&bin_sem);
        counter = counter + 1;
        sem_post(&bin_sem);
        printf("Thread1: Counter = %d\n", counter);
        sem_post(&bin_sem_th2);
    }
    pthread_exit("Thank you for the CPU time");
}

void *thread_function2(void *arg)
{
    int i = 0;

    for (i= 0; i < LOOP_COUNTER; i++) {
        sem_wait(&bin_sem_th2);
        sem_wait(&bin_sem);
        counter = counter - 1;
        sem_post(&bin_sem);
        printf("Thread2: Counter = %d\n", counter);
        sem_post(&bin_sem_th1);
    }
    pthread_exit("Thank you for the CPU time");
}
```

# Synchronization

```
Thread1: Counter = 1
Thread2: Counter = 0
Thread1: Counter = 1
Thread2: Counter = 0
Thread1: Counter = 1
Thread2: Counter = 0
Thread1: Counter = 1
Thread2: Counter = 0
Thread1: Counter = 1
Thread2: Counter = 0
Thread1: Counter = 1
Thread2: Counter = 0
Thread1: Counter = 1
Thread2: Counter = 0
Thread1: Counter = 1
Thread2: Counter = 0
Thread1: Counter = 1
Thread2: Counter = 0
Thread1: Counter = 1
Thread2: Counter = 0
Thread1: Counter = 1
Thread2: Counter = 0
main: Final counter: 0
```

# Thread Attributes

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
```

- This attribute allows you to avoid the need for threads to rejoin.

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
```

- This controls how threads are scheduled. The options are SCHED_OTHER, SCHED_RR, and SCHED_FIFO.

```
int pthread_attr_getschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr,
                               struct sched_param *param);
```

- This is a partner to schedpolicy and allow control over the scheduling of threads running with schedule policy SCHED_OTHER.

# Thread Attributes

```
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inherit);
```

- This attribute takes two possible values: PTHREAD_EXPLICIT_SCHED and PTHREAD_INHERIT_SCHED.

```
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

- This attribute controls how the scheduling of a thread is calculated.

```
int pthread_attr_setstacksize(pthread_attr_t *attr, int stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr, int *stacksize);
```

- This attribute controls the thread creation stack size, set in bytes.

# Setting the Detached State Attribute

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

void *thread_function(void *arg);

char message[] = "Hello World";
int thread_finished = 0;

int main() {
    int res;
    pthread_t a_thread;

    pthread_attr_t thread_attr;

    res = pthread_attr_init(&thread_attr);
    if (res != 0) {
        perror("Attribute creation failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
    if (res != 0) {
        perror("Setting detached attribute failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, &thread_attr,
thread_function, (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
```

# Setting the Detached State Attribute

```
        (void)pthread_attr_destroy(&thread_attr);
        while(!thread_finished) {
            printf("Waiting for thread to say it's finished...\n");
            sleep(1);
        }
        printf("Other thread finished, bye!\n");
        exit(EXIT_SUCCESS);
    }

    void *thread_function(void *arg) {
        printf("thread_function is running. Argument was %s\n", (char *)arg);
        sleep(4);
        printf("Second thread setting finished flag, and exiting now\n");
        thread_finished = 1;
        pthread_exit(NULL);
    }
```

```
$ ./thread5
Waiting for thread to say it's finished...
thread_function is running. Argument was Hello World
Waiting for thread to say it's finished...
Waiting for thread to say it's finished...
Waiting for thread to say it's finished...
Second thread setting finished flag, and exiting now
Other thread finished, bye!
```

# Canceling a Thread

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);

int pthread_setcancelstate(int state, int *oldstate);

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

# Canceling a Thread

```
#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);
}
```

# Canceling a Thread

```
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);
}
```

```
$ ./thread7
thread_function is running
Thread is still running (0)...
Thread is still running (1)...
Thread is still running (2)...
Canceling thread...
Waiting for thread to finish...
$
```

# Threads in Abundance

```
#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");
```

# Threads in Abundance

```
    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);
}
```

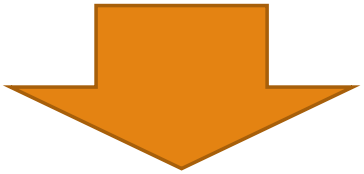
```
$ ./thread8
thread_function is running. Argument was 0
thread_function is running. Argument was 1
thread_function is running. Argument was 2
thread_function is running. Argument was 3
thread_function is running. Argument was 4
Bye from 1
thread_function is running. Argument was 5
Waiting for threads to finish...
Bye from 5
Picked up a thread
Bye from 0
Bye from 2
Bye from 3
Bye from 4
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
All done
```

# Threads in Abundance

```
thread_function is running. Argument was 0
thread_function is running. Argument was 2
thread_function is running. Argument was 2
thread_function is running. Argument was 4
thread_function is running. Argument was 4
thread_function is running. Argument was 5
Waiting for threads to finish...
Bye from 5
Picked up a thread
Bye from 2
Bye from 0
Bye from 2
Bye from 4
Bye from 4
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
All done
```

# Threads in Abundance

```
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);
```



```
res = pthread_create(&(a_thread[lots_of_threads]), NULL, thread_function, (void
*)lots_of_threads);
```

```
void *thread_function(void *arg) {
    int my_number = (int)arg;
```

# Summary

How to create several threads of execution inside a process

Semaphore and mutex: Two ways that threads can control access to critical code and data

How you could separate them from the main thread so that it no longer had to wait for threads that it had created to complete