Languages for Informatics

11 – Multi-Threading

Department of Computer Science
University of Pisa
Largo B. Pontecorvo 3
56127 Pisa

Topics

- Linux programming environment (2h)
- Introduction to C programming (12h)
- Basic system programming in Linux (10h)
  1. Signals and Error Handling
  2. Low-Level System Calls in C
  3. Multi-Tasking in C
  4. Multi-Threading in C
  5. Machine-To-Machine Communication in C

Overview

1. Shared Memory
2. PThread Management
    - Creating and Terminating Threads
    - Passing Arguments to Threads
    - Joining and Detaching Threads
3. Mutex Synchronization
    - Creating and Destroying Mutexes
    - Locking and Unlocking Mutexes
4. Semaphore Synchronization
5. Synchronization by Condition Variables

1. Shared Memory
2. PThread Management
   - Creating and Terminating Threads
   - Passing Arguments to Threads
   - Joining and Detaching Threads
3. Mutex Synchronization
   - Creating and Destroying Mutexes
   - Locking and Unlocking Mutexes
4. Semaphore Synchronization
5. Synchronization by Condition Variables

Processes and Threads

Suppose

- **sewing needles are processors**
- and **thread in a programs as thread fiber.**

- If you had two needles but only one thread, one needle is idle (waste of time)
- if you split the thread into two, one needle can continue sewing even if the other is busy with one button (blocking I/O)

Processes and Threads (cont'd)

- A computer program becomes a **process** when it is loaded from some store into the computer's memory and begins execution.
  - A process can be executed by a processor or a set of processors.
- A **thread** is a sequence of instructions within a program that can be **executed independently** of other code.
  - threads contain only necessary information, such as a **stack**, a **copy of the registers**, **program counter** and thread specific data to allow them to be **scheduled individually**.
  - Other data, like **address space, is shared within the process** among all threads.

Real Operating Systems

- One or many address spaces
- One or many threads per address space

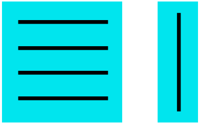
	1 address space	Many address spaces
1 thread per address space	MSDOS MacIntosh	Old UNIX (pre-1993)
Many threads per address space	Embedded OS Pilot	VMS, OS/2 MSWindows Solaris, HP-UX, Linux

- Multiple threads may run under multiple processes and communicate within the process.

An illustrative example

- Suppose we want to multiply a $M \times N$-dim. matrix with a N-dim. vector,

$$[\boldsymbol{x}]_m = \sum_{n=1}^{N} [\boldsymbol{A}]_{m,n}[\boldsymbol{b}]_n$$

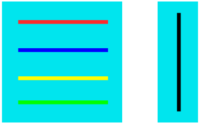


For, $M = 40$ and $N = 2e6$ on
Intel Celeron J4105 with 4 threads/core:
`Execution time:  0.660 s`

An illustrative example (cont'd)

- Suppose we want to multiply a $M \times N$-dim. matrix with a N-dim. vector,

$$[\boldsymbol{x}]_m = \sum_{n=1}^{N} [\boldsymbol{A}]_{m,n}[\boldsymbol{b}]_n$$



For, $M = 40$ and $N = 2e6$ on
Intel Celeron J4105 with 4 threads/core:
`Execution time:  0.183 s`

POSIX Threads

- Before the POSIX standard, each computer vendor would implement its own thread library and the resulting programs were not portable across different computer systems.
- POSIX Threads (**PThreads**) are a **standard for Unix-like** operating systems.
- A library that can be linked with C programs.
- Specifies an application programming interface (API) for multi-threaded programming

The PThread API

- The original Pthreads API was defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard. The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.
- Subroutines comprising the Pthreads API:
  1. **Thread management**: routines that create, detach, join threads. They also include functions to set/query thread attributes.
  2. **Mutexes**: routines for synchronization, i.e. ”mutual exclusion”, to create, destroy, lock and unlock mutexes.
  3. **Condition variables**: routines for Communications between threads that share a mutex.

1. Shared Memory
2. PThread Management
   - Creating and Terminating Threads
   - Passing Arguments to Threads
   - Joining and Detaching Threads
3. Mutex Synchronization
   - Creating and Destroying Mutexes
   - Locking and Unlocking Mutexes
4. Semaphore Synchronization
5. Synchronization by Condition Variables

1. Shared Memory
2. PThread Management
   - Creating and Terminating Threads
   - Passing Arguments to Threads
   - Joining and Detaching Threads
3. Mutex Synchronization
   - Creating and Destroying Mutexes
   - Locking and Unlocking Mutexes
4. Semaphore Synchronization
5. Synchronization by Condition Variables

Creating and Terminating Threads

Routines

pthread_create `(&thread,&attr,start_routine,arg)`
pthread_exit `(status)`
pthread_cancel `(thread)`
pthread_attr_init `(attr)`
pthread_attr_destroy `(attr)`

Creating Threads

- Initially, your `main()` program comprises a single, default thread.
  - More threads can be created by the programmer
- **`pthread_create()`** creates a new thread and makes it executable
  - **can be called any number of times** from anywhere within your code.
  - Once created, **threads are peers**, and may create other threads.
  - The *maximum number* of threads that may be created by a process is implementation dependent.

Creating Threads
Arguments

- **`pthread_create()`** arguments
  - `thread`: A unique identifier for the new thread returned by the subroutine.
  - `attr`: An opaque attribute object to specify a thread attributes object, or `NULL` for the default values.
  - `start_routine`: the C function that the thread will execute once it is created.
  - `arg`: A single argument that may be passed to `start_routine`, passed by reference as a pointer cast of type void or `NULL`

Creating Threads
Attributes

- Set attributes for a newly created thread through special bit-variable of the type `pthread_attr_t`.
- Define variable
  `pthread_attr_t` **`attr;`**
- See also
  `pthread_attr_init(&attr);`
- Default values available at
  `https://man7.org/linux/man-pages`, Section 3.

Creating Threads
Attributes - Default Values

pthread_attr	Default POSIX	Comment
_getscope	PTHREAD_SCOPE_SYSTEM	Thread will compete for resources with all other threads in all processes.
_getdetachstate	PTHREAD_CREATE_JOINABLE	Thread is joinable by other threads.
_getstackaddr	NULL (turned-off)	Stack used by the thread is allocated by the OS.
_getstacksize	PTHREAD_STACK_MIN	Sets e.g. 8 MB (8192 kB) stack size for a new thread on Linux 64-bit.
_getschedparam	0	Max. priority of the thread.
_getschedpolicy	SCHED_OTHER	The scheduling policy is given by OS.
_getinheritsched	PTHREAD_INHERIT_SCHED	Scheduling policy and parameters are inherited from the creating thread.
_getguardsize	PAGESIZE (4096 B)	Size of guard area for a thread's created stack equal system page size.

There is **no need** to change MOST of the default values.

Terminating Threads

- `void` **`pthread_exit()`** causes the current thread to exit and free any thread-specific resources it is taking.
- Thread can terminate in several ways :
    - The thread returns normally from its starting routine. Its work is done.
    - The thread makes a call to the **`pthread_exit`** subroutine - whether its work is done or not.
    - The thread is canceled by another thread via the **`pthread_cancel`** routine.
    - If **`main()`** finishes first, without calling **`pthread_exit`** explicitly itself

Pthread Creation and Termination

Example

```
#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);
       }
     }
   pthread_exit(NULL);  /* finally, just exit w/o return value */
}
```

Pthread Creation and Termination

Example (cont'd)

Compile and Run

bash~$ **gcc helloworld5.c -Wall -lpthread**
bash~$ **./a.out**

Trace 1

```
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
In main:  creating thread 3
Hello World!  It's me, thread #0!
In main:  creating thread 4
Hello World!  It's me, thread #4!
Hello World!  It's me, thread #1!
Hello World!  It's me, thread #2!
Hello World!  It's me, thread #3!
```

Trace 2

```
In main:  creating thread 0
In main:  creating thread 1
Hello World!  It's me, thread #0!
In main:  creating thread 2
In main:  creating thread 3
Hello World!  It's me, thread #1!
In main:  creating thread 4
Hello World!  It's me, thread #3!
Hello World!  It's me, thread #2!
Hello World!  It's me, thread #4!
```

1. Shared Memory
2. PThread Management
   - Creating and Terminating Threads
   - Passing Arguments to Threads
   - Joining and Detaching Threads
3. Mutex Synchronization
   - Creating and Destroying Mutexes
   - Locking and Unlocking Mutexes
4. Semaphore Synchronization
5. Synchronization by Condition Variables

Passing Arguments to Threads

- The `pthread_create()` routine permits the programmer to **pass one argument to the thread** start routine.
- For cases where **multiple arguments** must be passed, this limitation is easily overcome by creating a **structure** containing the arguments, and then passing a pointer to that structure in the `pthread_create()` routine.
- All arguments must be passed by reference and cast to `(void *)`.

Note

Make sure that all passed data is thread safe, i.e. can not be changed by other threads.

Passing Arguments to Threads

Example

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

struct thread_data {
   int thread_id;
   char * message;
 };

void *PrintHello(void *threadarg) {
   int tid;
   char *hello_msg;
   struct thread_data *my_data;
   my_data = (struct thread_data *) threadarg;
   tid = my_data->thread_id;
   hello_msg = my_data->message;
   printf("Thread %d: %s \n", tid, hello_msg);
   pthread_exit(NULL);
}
```

Passing Arguments to Threads

Example

```
int main(int argc, char *argv[]) {
  char *messages[NUM_THREADS];
  struct thread_data thread_data_array[NUM_THREADS]; //array of struct
  messages[0] = "English: Hello World!";
  ...
  pthread_t threads[NUM_THREADS];
  int rc; long t;

  for(t=0;t<NUM_THREADS;t++) {
    printf("In main: creating thread %ld\n", t);
    thread_data_array[t].thread_id = t;
    thread_data_array[t].message = messages[t];
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &thread_data_array[t]);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
      }
    }
  pthread_exit(NULL);
}
```

1. Shared Memory
2. PThread Management
   - Creating and Terminating Threads
   - Passing Arguments to Threads
   - Joining and Detaching Threads
3. Mutex Synchronization
   - Creating and Destroying Mutexes
   - Locking and Unlocking Mutexes
4. Semaphore Synchronization
5. Synchronization by Condition Variables

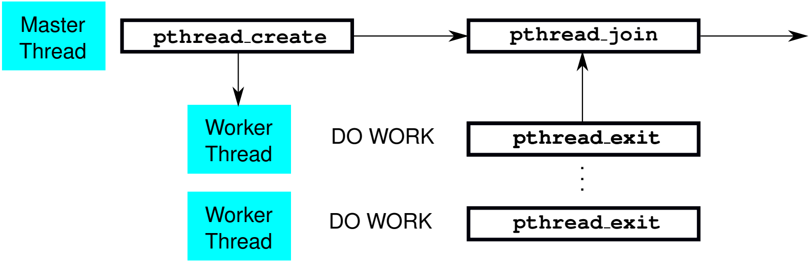
Joining and Detaching Threads

Routines

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
int pthread_detach(pthread_t thread);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int
    detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int
    *detachstate);
```

- **Joining** is one way to accomplish synchronization between threads.
- Two other synchronization methods, *mutexes and condition variables*, come later.

Joining Threads



- The **`pthread_join()`** subroutine blocks the calling thread until the specified `threadID` thread terminates.
- When the target is terminated by **`pthread_exit(`** `void` **`*rval_ptr)`**, the return value in the argument is accessible by **`pthread_join()`**.

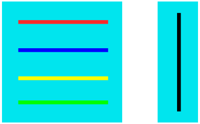
Joining Threads (cont'd)

- POSIX standard specifies that threads should be created as joinable.
- Consider **explicitly creating it as joinable.** This provides portability as not all implementations may create threads as joinable by default.
- **Procedure**:
  1. Declare a pthread attribute variable of the `pthread_attr_t` data type
  2. Initialize the attribute variable with **`pthread_attr_init()`**
  3. Set the attribute detached status with **`pthread_attr_setdetachstate()`**
  4. When done, free library resources used by the attribute with **`pthread_attr_destroy()`**

Example

Matrix-Vector Multiplication revisited

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>

/* Global variables */
int     MAX_THREAD;
int     M, N;
double** A;
double* b;
double* x;

void *matvec_mlt(void* junk) {  //assign rows to threads
   long my_junk = (long) junk;
   int i, j;
   int local_m = M/MAX_THREAD;
   int my_first_row = my_junk*local_m;
   int my_last_row = my_first_row + local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      x[i] = 0.0;
      for (j = 0; j < N; j++)
          x[i] += A[i][j]*b[j];
   }
   pthread_exit(NULL);
}
```



Example

Matrix-Vector Multiplication revisited

```
int main(int argc, char* argv[]) {
   if(argc != 4)    {
      fprintf(stderr, "Usage: %s <rows> <cols> <threads>\n", argv[0]); return 1;
    }
  M = atoi(argv[1]); N = atoi(argv[2]);
  int i,j,rc; long t;  //thread index
  void *status; // return status obtained by thread_join
  pthread_t* thread_handles;
  pthread_attr_t attr;
  MAX_THREAD = atoi(argv[3]);   //variable number of threads
  thread_handles = malloc(MAX_THREAD*sizeof(pthread_t));
  pthread_attr_init(&attr);   //reset to default.
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  ... /* allocate memory dynamically to A,b,x + assign values */
  for (t = 0; t < MAX_THREAD; t++) {
    rc = pthread_create(&thread_handles[t], &attr, matvec_mlt, (void *) t);
    if (rc) perror("thread_create");
  }
  pthread_attr_destroy(&attr);  /* Free attribute and wait for the other threads */
  for (t = 0; t < MAX_THREAD; t++) {
    rc = pthread_join(thread_handles[t], &status);
    if (rc) perror("thread_join");
    printf("Main: completed join with thread %ld having a status of %ld\n",t,(long)status);
  }
  ...  /*  print result */
  free(A); free(b); free(x);
  free(thread_handles); pthread_exit(NULL); return 0;
}
```

Example

Matrix-Vector Multiplication revisited

shell

```
bash~$ gcc mv_mlt_thread.c -Wall -lpthread
bash~$ ./a.out 2 2 2
Main: completed join with t 0 having a status of 0
Main: completed join with t 1 having a status of 0
A[0][0] = 33.00
A[0][1] = 36.00
A[1][0] = 27.00
A[1][1] = 15.00
b[0] = 43.00
b[1] = 35.00
x[0] = 2679.00
x[1] = 1686.00
```

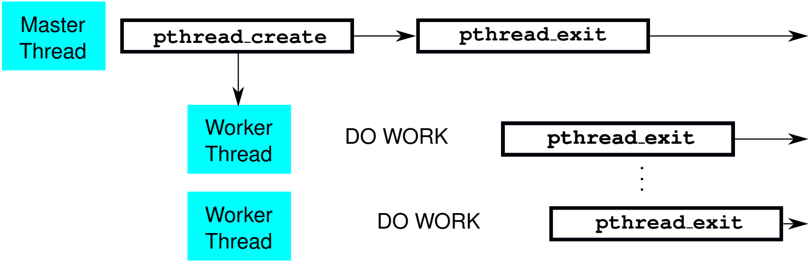
Example

Threads with return value

```
int ret[MAX_NUMBER_THREAD];

void *matvec_mlt(void* junk) {
   ...
   ret[my_junk] = <some_value_to_be_returned> ;
   pthread_exit(&ret[my_junk]);
   return NULL;
}

int main(int argc, char* argv[]) {
  int *ptr[MAX_THREAD];
  ...
  for (t = 0; t < MAX_THREAD; t++)
    pthread_join(thread_handles[t], (void**)&ptr[thread]);

  for (t = 0; t < MAX_THREAD; t++)
    printf("\n return value from thread = %d\n", *ptr[thread]);
}
  ...
```

Detaching Threads

- The **`pthread_detach()`** routine can be used to explicitly detach a thread **even though it was created as joinable.**



Example

Demo

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>  //sleep

void *func(void *data) {
    while (1) {
        printf("Speaking from the detached thread...\n");
        sleep(5); }
    pthread_exit(NULL); }

int main() {
    pthread_t handle;
    if (!pthread_create(&handle, NULL, func, NULL)) {
        printf("Thread create successfully !!!\n");
        if ( ! pthread_detach(handle) )
            printf("Thread detached successfully !!!\n");
    }
    printf("Main thread dying...\n");
    pthread_exit(NULL);
    return 0; }
```

1. Shared Memory
2. PThread Management
   - Creating and Terminating Threads
   - Passing Arguments to Threads
   - Joining and Detaching Threads
3. Mutex Synchronization
   - Creating and Destroying Mutexes
   - Locking and Unlocking Mutexes
4. Semaphore Synchronization
5. Synchronization by Condition Variables

An illustrative example

```
#include <stdio.h>
#include <pthread.h>
#define THREAD_MAX 2
volatile int counter = 0; //read from memory every time

void *testing(void *param) {
   int i;
   for(i = 0; i < 5; i++) {
     counter++;
     printf("thread %d counter = %d\n", (int)param,  counter);
   }
   pthread_exit(NULL);
}

int main() {
  int arr[] = {1,2};
  pthread_t thread[THREAD_MAX];

  for (int t=0;t<THREAD_MAX;t++)
    pthread_create(&thread[t], 0, testing, (void*) arr[t]);
  for (int t=0;t<THREAD_MAX;t++)
    pthread_join(thread[t], 0);
  pthread_exit(NULL);
  return 0;
}
```

An illustrative example

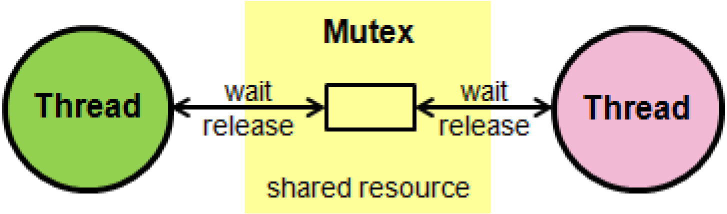
Compile and run

```
thread 1 counter = 1
thread 1 counter = 3
thread 1 counter = 4
thread 2 counter = 2
thread 2 counter = 6
thread 2 counter = 7
thread 2 counter = 8
thread 2 counter = 9
thread 1 counter = 5
thread 1 counter = 10
```

What has occured ?

- Any of the two jobs adds $+1$ to the same counter variable in memory.
- The job order depends on the (random) scheduler.
- Synchronization between the jobs is missing.

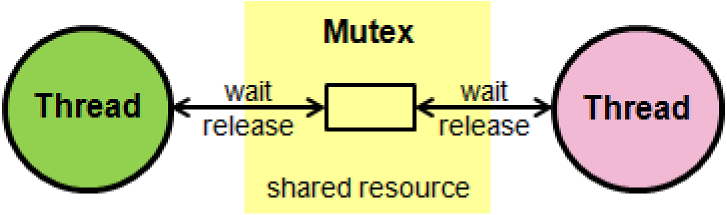
Mutex



Source: keil.com

- **Mutex** is a variable being **owned by one and only one thread.**
- **Principle**: When one thread owns the mutex variable, **any other thread is blocked** until this thread unlocks the mutex variable.

Mutex



Source: keil.com

Note

A **deadlock** occurs when one or more threads are blocked waiting for being unlocked that will never occur.

Mutex Variables

- A **typical sequence** in the use of a mutex is as follows:
  - **Create** and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - **Only one succeeds** and that thread owns the mutex
  - The owner thread performs some **set of actions**
  - The owner **unlocks the mutex**
  - **Another thread acquires** the mutex and repeats the process
  - Finally the mutex is **destroyed**

Note

- Make sure **every thread that needs to use a mutex does so**!
- For example, if 4 threads are updating the same data, but only one uses a mutex, the data can still be corrupted.

1. Shared Memory
2. PThread Management
   - Creating and Terminating Threads
   - Passing Arguments to Threads
   - Joining and Detaching Threads
3. Mutex Synchronization
   - Creating and Destroying Mutexes
   - Locking and Unlocking Mutexes
4. Semaphore Synchronization
5. Synchronization by Condition Variables

Creating and Destroying Mutexes

Routines

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
    pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

1. Mutex variables must be declared with type `pthread_mutex_t`, and initialized:
   - **Statically**, when it is declared. For example: `pthread_mutex_t` **`mymutex = PTHREAD_MUTEX_INITIALIZER;`**
   - **Dynamically**, with the **`pthread_mutex_init()`** routine. This method permits setting mutex object attributes, `attr`.
2. The mutex is **initially unlocked.**

Creating and Destroying Mutexes (cont'd)

- The `attr` object establishes properties for the mutex object (of type `pthread_mutexattr_t`)
- `pthread_mutexattr_settype`:
  - `PTHREAD_MUTEX_NORMAL`: This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock.
  - `PTHREAD_MUTEX_ERRORCHECK`: A thread attempting to relock this mutex without first unlocking it will return with an error.
  - `PTHREAD_MUTEX_RECURSIVE`: Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex, **to prevent deadlock scenario**.

Creating and Destroying Mutexes (cont'd)

Routines

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
    pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

3. The **`pthread_mutexattr_init()`** and **`pthread_mutexattr_destroy()`** routines are used to create and destroy mutex attribute objects respectively.
4. **`pthread_mutex_destroy()`** should be used to free a mutex object which is no longer needed.

1. Shared Memory
2. PThread Management
   - Creating and Terminating Threads
   - Passing Arguments to Threads
   - Joining and Detaching Threads
3. Mutex Synchronization
   - Creating and Destroying Mutexes
   - Locking and Unlocking Mutexes
4. Semaphore Synchronization
5. Synchronization by Condition Variables

Locking and Unlocking Mutex

Routines

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Usage:

1. **`pthread_mutex_lock()`** used by a thread to acquire a lock on the specified mutex variable according to above policy by `attr`.
2. **`pthread_mutex_trylock()`** will attempt to lock a mutex. If mutex already locked, routine returns `EBUSY` `errno` code.

Locking and Unlocking Mutex

Routines

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Usage:

3. **`pthread_mutex_unlock()`** will unlock a mutex if called by the owning thread. Returns a non-zero value
   - when the mutex was already unlocked
   - when the mutex is owned by another thread

The illustrative example revisted

```
#include <stdio.h>
#include <pthread.h>
volatile int counter = 0; //read from memory every time
#define THREAD_MAX 2
pthread_mutex_t myMutex;

void *testing(void *param) {
   for(int i = 0; i < 5; i++) {
     pthread_mutex_lock(&myMutex);     // any thread arriving here will be locked
     counter++;                        //increases counter
     printf("thread %lu counter = %d\n", (intptr_t) param,  counter);
     pthread_mutex_unlock(&myMutex);   //thread will be unlocked
   }
   pthread_exit(NULL);
   return 0;
}
int main() {
  int arr[] = 1,2;
  pthread_t thread[THREAD_MAX];
  pthread_mutex_init(&myMutex,0);
  for (int t=0;t<THREAD_MAX;t++)
    pthread_create(&thread[t], 0, testing, (void*) (intptr_t) arr[t]);
  for (int t=0;t<THREAD_MAX;t++)
    pthread_join(thread[t], 0);
  pthread_exit(NULL);
  pthread_mutex_destroy(&myMutex);
  return 0;
}
```

The illustrative example revisited

Compile and run

```
bash~$ gcc mutex.c -o mutex -Wall -lpthread
bash~$ ./mutex
thread 1 counter = 1
thread 1 counter = 2
thread 1 counter = 3
thread 2 counter = 4
thread 2 counter = 5
thread 2 counter = 6
thread 2 counter = 7
thread 2 counter = 8
thread 1 counter = 9
thread 1 counter = 10
```

Result

- The Mutex lock has synchronized the threads.
- The **counter is correctly updated among threads**.

1 Shared Memory

2 PThread Management
- Creating and Terminating Threads
- Passing Arguments to Threads
- Joining and Detaching Threads

3 Mutex Synchronization
- Creating and Destroying Mutexes
- Locking and Unlocking Mutexes

4 Semaphore Synchronization

5 Synchronization by Condition Variables

Semaphore Synchronization

- POSIX semaphores allow processes and threads to synchronize their actions.
  - Semaphore is a **signaling mechanism**
  - Mutex is a locking mechanism
- A semaphore is a **positive integer** variable s.
- Starting from $s = N$ (number of free resources), Dijkstra's[1] **`wait`** $P(s)$ and **`signal`** $V(s)$ operations are:
  - **`wait`**: *Decrements the value of semaphore variable by 1*. The process is blocked and may continue execution, when the new value of the semaphore variable is negative and positive, respectively.
  - **`signal`**: *Increments the value of semaphore variable by 1*. If the new value is zero, waiting process is awakened.

[1] The semaphore concept was invented by Dutch computer scientist Edsger Dijkstra in 1962/63

Semaphore Synchronization (cont'd)

Note

- OS guarantees that `wait()` and `signal()` are **atomic operations.**
  - Only one $P(s)$ or $V(s)$ operation at a time can modify s
  - When loop in $P(s)$ terminates, only that $P(s)$ can decrement s

Unnamed Semaphores

Procedure:

- **Declare** the semaphore global (outside of any funcion):

```
#include <semaphore.h>
sem_t s;
```

- **Initialize** the *unnamed* semaphore in the main function:

```
#include <semaphore.h>
int sem_init(sem_t *s, int pshared, unsigned int
    value);
```

s : address of the declared semaphore
pshared : should be 0 (not shared with threads in other processes)
value : the desired initial value of the semaphore

- On success, the return value is 0.

Unnamed Semaphores
Example

Thread 1	Thread 2	Data
`sem_wait(&s);`	—	0
—	`sem_wait(&s);`	0
`count++;`	/* blocked */	1
`sem_post(&s);`	/* blocked */	1
/* blocked */	`count++;`	2
/* blocked */	`sem_post(&s);`	2

- When you can't afford to wait for the lock, **`sem_trywait()`** locks immediately if $s > 0$ and sets `EAGAIN` error otherwise.
- **Destroy** the unnamed semaphore in the main function:

```
#include <semaphore.h>
int sem_destroy(sem_t *s);
```

The illustrative example revisted

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h> //sleep
volatile int counter = 0;
int THREAD_MAX=2;
sem_t mySem;
void *sem_testing(void *param) {
  int i;
  for(i = 0; i < 5; i++) {
    sem_wait(&mySem);      //any thread may lock the semaphore
    counter++;              //does its job
    printf("thread %lu counter = %d\n", (intptr_t) param,  counter);
    sem_post(&mySem);      //and unlock the semaphore again
  }
  pthread_exit(NULL);
  return NULL;
}
int main() {
  int arr[] = 1,2;
  pthread_t thread[THREAD_MAX];
  sem_init(&mySem,0,1);
  for (int t=0;t<THREAD_MAX;t++)
  pthread_create(&thread[t], 0, sem_testing, (void*) (intptr_t) arr[t]);
  for (int t=0;t<THREAD_MAX;t++)
    pthread_join(thread[t], 0);
  sem_destroy(&mySem);
  pthread_exit(NULL);
  return 0;
```

The illustrative example revisited

Compile and run

```
bash~$ gcc semaphore.c -o semaphore -Wall -lpthread
bash~$ ./semaphore
thread 1 counter = 1
thread 1 counter = 2
thread 1 counter = 3
thread 1 counter = 4
thread 2 counter = 5
thread 2 counter = 6
thread 2 counter = 7
thread 2 counter = 8
thread 2 counter = 9
thread 1 counter = 10
```

Result

- POSIX Mutex allows the **counter to be correctly updated among threads**.

1 Shared Memory

2 PThread Management
- Creating and Terminating Threads
- Passing Arguments to Threads
- Joining and Detaching Threads

3 Mutex Synchronization
- Creating and Destroying Mutexes
- Locking and Unlocking Mutexes

4 Semaphore Synchronization

5 Synchronization by Condition Variables

Condition Variables

- Condition variables[2] are like Mutexes ways for threads synchronization.
  - Condition variables allow particular threads to be notified once a particular **data value** occurs.
  - Mutex implements synchronization by controlling thread access to data
- A condition variable is **always used in conjunction with a mutex lock.**
- Birrel proposed the `condition` as condition variables abstraction as well as three operations `wait`, `signal` and `broadcast`.

[2] The concept of condition variables goes back to Birrel at Microsoft Research in 2003

Condition Variables (cont'd)

- The designated **variable type**

```
pthread_cond_t aCond;
```

- To **block the calling thread** on the condition variable `aCond`,

```
int pthread_cond_wait(pthread_cond_t *restrict aCond
    , pthread_mutex_t *restrict mutex);
```

  - The function takes two arguments, a condition variable and a mutex.
  - The calling thread must have acquired the mutex lock.
  - Note that before blocking the mutex lock is internally released. This allows other threads to also acquire the mutex lock and wait on this condition variable.
  - When this function returns, the lock is still held by this thread.

Condition Variables (cont'd)

- To **unblock at least one** of the threads that is blocked on the specified condition variable `aCond`,

```
int pthread_cond_signal(pthread_cond_t *cond);
```

  - This function has no effect if no threads are blocked on the condition variable `aCond`.
  - The unblocked thread re-acquires the associated mutex lock before returning from `pthread_cond_wait()`.

- Moreover,

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

unblocks all the threads that are blocked on the specified condition variable cond.

Condition Variables (cont'd)

- **Initialization** is pretty straight forward;

```
int pthread_cond_init(pthread_cond_t *restrict cond,
     const pthread_condattr_t *restrict attr);
```

  - This function initializes the condition variable `aCond` with attributes specified by `attr`.
  - When `attr` is `NULL`, the default condition variable attributes are used.

- Just like threads, condition variables **should be explicitly freed**,

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Ping-Pong Counter

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
pthread_mutex_t mux[2];
pthread_cond_t cond[2];
volatile int count = 0;
#define THREAD_MAX 2
void *playerX( void *param ) {
  long id = (intptr_t) param;
  if (id==0) {                              //THREAD 0
    for (int i = 0; i < 5; i++) {
    pthread_mutex_lock(&mux[0]);
    pthread_cond_wait(&cond[0], &mux[0]);
    count ++;
    printf("thread %lu  counter = %d\n", id,  count);
    pthread_cond_signal(&cond[1]);
    pthread_mutex_unlock(&mux[0]); } }
  else if(id==1)  {                          //THREAD 1
  for (int i = 0; i < 5; i++) {
    pthread_mutex_lock(&mux[1]);
    pthread_cond_wait(&cond[1], &mux[1]);
    count ++;
    printf("thread %lu  counter = %d\n", id,  count);
    pthread_cond_signal(&cond[0]);
    pthread_mutex_unlock(&mux[1]); } }
  pthread_exit(NULL); return NULL; }
```

Ping-Pong Counter
Example (cont'd)

```
int main() {
    pthread_t thread[THREAD_MAX];

    for (int t=0;t<THREAD_MAX;t++)   {
      pthread_mutex_init(&mux[0],0);    //init mutex dynamically
      pthread_cond_init(&cond[t],0);  //init cond. dynamically
      pthread_create(&thread[t], NULL, playerX, (void*) (intptr_t) t);
    }

    sleep(1); //give the first thread time, to get the lock
    pthread_cond_signal(&cond[0]);    // s=s+1

    for (int t=0;t<THREAD_MAX;t++)
      pthread_join(thread[t], 0);

    for (int t=0;t<THREAD_MAX;t++) {
      pthread_cond_destroy(&cond[t]);
      pthread_mutex_destroy(&mux[t]);
      }
    pthread_exit(NULL);
    return 0;
}
```

Ping-Pong Counter

Example (cont'd)

Compile and run

```
bash~$ gcc condition.c -o condition -Wall -lpthread
bash~$ ./condition
thread 0 counter = 1
thread 1 counter = 2
thread 0 counter = 3
thread 1 counter = 4
thread 0 counter = 5
thread 1 counter = 6
thread 0 counter = 7
thread 1 counter = 8
thread 0 counter = 9
thread 1 counter = 10
```

Result

- Condition Variables Signaling allows the **counter to be correctly updated among threads in a ping-pong fashion**.

Quiz

A thread life cycle consists of

1. 2
2. 3
3. 4
4. 5

states?