

# Lecture 8

## Programming Shared Memory II

Synchronization Primitives; Mutex

Ceng471 *Parallel Computing* at December 16, 2010

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

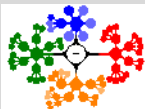
Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

Dr. Cem Özdoğan  
Computer Engineering Department  
Çankaya University



## 1 Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to Threads

Thread Cancellation

Joining and Detaching Threads

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

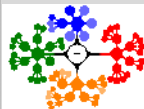
## 2 Synchronization Primitives in Pthreads

Mutual Exclusion for Shared Variables

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

- **Passing Arguments to Threads**



Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Passing Arguments to Threads**
- The *pthread\_create()* function allows the programmer to pass one argument to the thread function.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

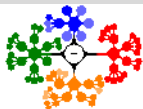
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Passing Arguments to Threads**
- The `pthread_create()` function allows the programmer to pass one argument to the thread function.
- For cases where multiple arguments must be passed, this limitation is easily overcome by creating a **structure**.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

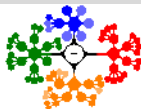
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Passing Arguments to Threads**
- The `pthread_create()` function allows the programmer to pass one argument to the thread function.
- For cases where multiple arguments must be passed, this limitation is easily overcome by creating a **structure**.
- This structure contains all of the arguments, and then a pointer is passed to that structure in the `pthread_create()` routine.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

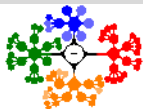
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Passing Arguments to Threads**
- The `pthread_create()` function allows the programmer to pass one argument to the thread function.
- For cases where multiple arguments must be passed, this limitation is easily overcome by creating a **structure**.
- This structure contains all of the arguments, and then a pointer is passed to that structure in the `pthread_create()` routine.
- All arguments must be passed by reference and cast to `(void *)`.

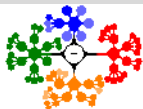
Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation  
Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Passing Arguments to Threads**
- The `pthread_create()` function allows the programmer to pass one argument to the thread function.
- For cases where multiple arguments must be passed, this limitation is easily overcome by creating a **structure**.
- This structure contains all of the arguments, and then a pointer is passed to that structure in the `pthread_create()` routine.
- All arguments must be passed by reference and cast to `(void *)`.
- **Threads have non-deterministic start-up and scheduling.**

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

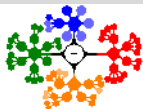
Passing Arguments to  
Threads

Thread Cancellation  
Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables





- **Passing Arguments to Threads**
- The `pthread_create()` function allows the programmer to pass one argument to the thread function.
- For cases where multiple arguments must be passed, this limitation is easily overcome by creating a **structure**.
- This structure contains all of the arguments, and then a pointer is passed to that structure in the `pthread_create()` routine.
- All arguments must be passed by reference and cast to `(void *)`.
- Threads have non-deterministic start-up and scheduling.
- **How can you safely pass data to newly created threads?**

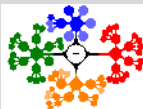
Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation  
Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Example:** Demonstrates how to pass a simple integer to each thread.

```
long *taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
    taskids[t] = (long *) malloc(sizeof(long));
    *taskids[t] = t;
    printf("Creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
    ...
}
```

**Figure:** Passing single argument to thread function.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

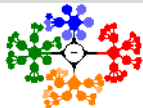
Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Thread Basics: Passing Arguments III

- **Example:** Demonstrates how to pass/setup multiple arguments to thread function via a structure.



Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

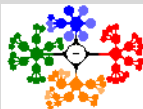
Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Thread Basics: Passing Arguments III

- **Example:** Demonstrates how to pass/setup multiple arguments to thread function via a structure.



Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Thread Basics: Passing Arguments III

- **Example:** Demonstrates how to pass/setup multiple arguments to thread function via a structure.

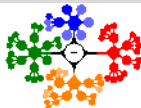
```
struct thread_data{
    int thread_id;
    int sum;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    ...
}

int main (int argc, char *argv[])
{
    ...
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
    ...
}
```

**Figure:** Passing multiple arguments to thread function via a structure.



## Thread Basics: Passing Arguments III

- **Example:** Demonstrates how to pass/setup multiple arguments to thread function via a structure.

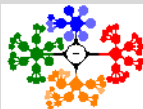
```
struct thread_data{
    int thread_id;
    int sum;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    ...
}

int main (int argc, char *argv[])
{
    ...
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
    ...
}
```

Each thread  
receives a *unique*  
instance of the  
structure.



**Figure:** Passing multiple arguments to thread function via a structure.

- **Cancellation.**



## Thread Basics: Passing Arguments, Cancellation and Joining

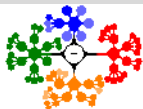
Passing Arguments to  
Threads

**Thread Cancellation**

Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Cancellation.**
- Consider a simple program to evaluate a set of positions in a chess game.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

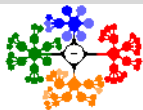
**Thread Cancellation**

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables





- **Cancellation.**
- Consider a simple program to evaluate a set of positions in a chess game.
- Assume that there are  $k$  moves, each being evaluated by an independent thread.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Cancellation.**
- Consider a simple program to evaluate a set of positions in a chess game.
- Assume that there are  $k$  moves, each being evaluated by an independent thread.
- If at any point of time, a position is established to be of a certain quality, the other positions that are known to be of worse quality must stop being evaluated.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Cancellation.**
- Consider a simple program to evaluate a set of positions in a chess game.
- Assume that there are  $k$  moves, each being evaluated by an independent thread.
- If at any point of time, a position is established to be of a certain quality, the other positions that are known to be of worse quality must stop being evaluated.
- In other words, the threads evaluating the corresponding board positions must be canceled.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Cancellation.**
- Consider a simple program to evaluate a set of positions in a chess game.
- Assume that there are  $k$  moves, each being evaluated by an independent thread.
- If at any point of time, a position is established to be of a certain quality, the other positions that are known to be of worse quality must stop being evaluated.
- In other words, the threads evaluating the corresponding board positions must be canceled.
- **Posix threads provide this cancellation feature.**

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Cancellation.**
- Consider a simple program to evaluate a set of positions in a chess game.
- Assume that there are  $k$  moves, each being evaluated by an independent thread.
- If at any point of time, a position is established to be of a certain quality, the other positions that are known to be of worse quality must stop being evaluated.
- In other words, the threads evaluating the corresponding board positions must be canceled.
- Posix threads provide this cancellation feature.
- **A thread may cancel itself or cancel other threads.**

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

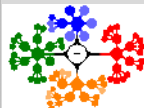
Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Thread Basics: Cancellation II

- **pthread\_cancel.**

```
1  int
2  pthread_cancel (
3      pthread_t  thread);
```



## Thread Basics: Passing Arguments, Cancellation and Joining

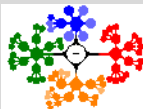
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **pthread\_cancel.**

```
1  int
2  pthread_cancel (
3      pthread_t  thread);
```

- Here, *thread* is the handle to the thread to be canceled. When a call to this function is made, a cancellation is sent to the specified thread.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

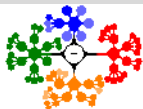
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **pthread\_cancel.**

```
1  int
2  pthread_cancel (
3      pthread_t  thread);
```

- Here, *thread* is the handle to the thread to be canceled. When a call to this function is made, a cancellation is sent to the specified thread.
- It is not guaranteed that the specified thread will receive or act on the cancellation. Threads can protect themselves against cancellation.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

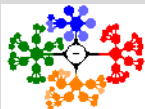
Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables





- **pthread\_cancel.**

```
1  int
2  pthread_cancel (
3      pthread_t  thread);
```

- Here, *thread* is the handle to the thread to be canceled. When a call to this function is made, a cancellation is sent to the specified thread.
- It is not guaranteed that the specified thread will receive or act on the cancellation. Threads can protect themselves against cancellation.
- When a cancellation is actually performed, cleanup functions are invoked for reclaiming the thread data structures.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **pthread\_cancel.**

```
1  int
2  pthread_cancel (
3      pthread_t  thread);
```

- Here, *thread* is the handle to the thread to be canceled. When a call to this function is made, a cancellation is sent to the specified thread.
- It is not guaranteed that the specified thread will receive or act on the cancellation. Threads can protect themselves against cancellation.
- When a cancellation is actually performed, cleanup functions are invoked for reclaiming the thread data structures.
- The **pthread\_cancel** function returns after a cancellation has been sent. The cancellation may itself be performed later.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

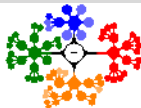
Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

- **Joining and Detaching Threads.**



Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Joining and Detaching Threads.**
- The main program must wait for the threads to run to completion.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

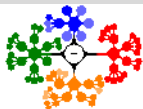
Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Thread Basics: Joining and Detaching I



- **Joining and Detaching Threads.**
- The main program must wait for the threads to run to completion.
- “Joining“ is one way to accomplish synchronization between threads.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

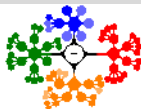
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Joining and Detaching Threads.**
- The main program must wait for the threads to run to completion.
- “Joining“ is one way to accomplish synchronization between threads.
- Function **pthread\_join** which suspends execution of the calling thread until the specified thread terminates.

```
1  int
2  pthread_join (
3      pthread_t thread,
4      void **ptr);
```

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

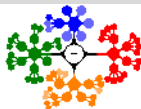
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Joining and Detaching Threads.**
- The main program must wait for the threads to run to completion.
- “Joining“ is one way to accomplish synchronization between threads.
- Function **pthread\_join** which suspends execution of the calling thread until the specified thread terminates.

```
1  int
2  pthread_join (
3      pthread_t thread,
4      void **ptr);
```

- A call to this function waits for the termination of the thread whose id is given by thread.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

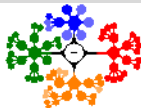
Thread Cancellation

Joining and Detaching  
Threads

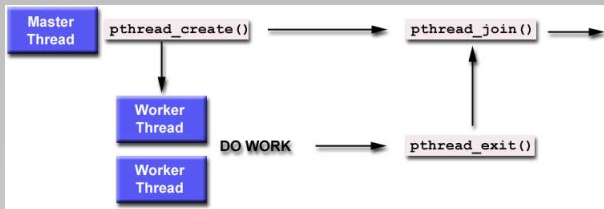
Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Thread Basics: Joining and Detaching II



- A call to this function waits for the termination of the thread whose id is given by thread.



**Figure:** Threads joining.

### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

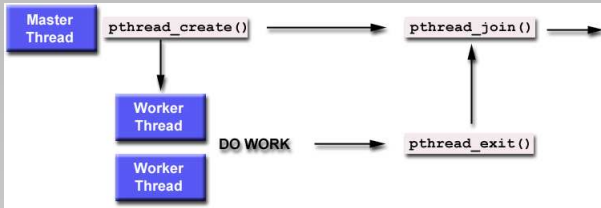
Mutual Exclusion for Shared  
Variables



## Thread Basics: Joining and Detaching II



- A call to this function waits for the termination of the thread whose id is given by `thread`.



**Figure:** Threads joining.

- On a successful call to **pthread\_join**, the value passed to **pthread\_exit** is returned in the location pointed to by *ptr*.

### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

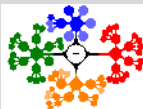
Thread Cancellation

Joining and Detaching  
Threads

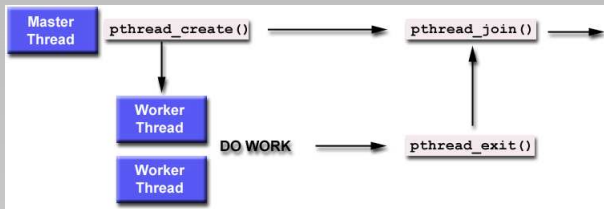
### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Thread Basics: Joining and Detaching II



- A call to this function waits for the termination of the thread whose id is given by thread.



**Figure:** Threads joining.

- On a successful call to **pthread\_join**, the value passed to **pthread\_exit** is returned in the location pointed to by *ptr*.
- On successful completion, **pthread\_join** returns 0, else it returns an error-code.

### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

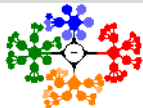
Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Thread Basics: Joining and Detaching III

- When a thread is created, one of its attributes defines whether it is **joinable** or **detached**.



## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

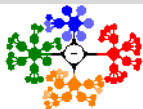
Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Thread Basics: Joining and Detaching III

- When a thread is created, one of its attributes defines whether it is **joinable** or **detached**.
- Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.



## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

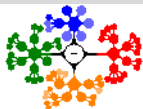
Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Thread Basics: Joining and Detaching III

- When a thread is created, one of its attributes defines whether it is **joinable or detached**.
- Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable.



## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

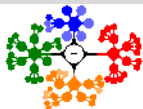
Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

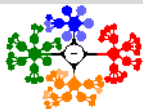
## Thread Basics: Joining and Detaching III

- When a thread is created, one of its attributes defines whether it is **joinable or detached**.
- Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable.
- To explicitly create a thread as joinable or detached, the **attr** argument in the *pthread\_create()* routine is used.



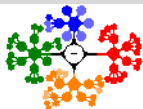
## Thread Basics: Joining and Detaching III

- When a thread is created, one of its attributes defines whether it is **joinable or detached**.
- Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable.
- To explicitly create a thread as joinable or detached, the **attr** argument in the `pthread_create()` routine is used.
- **Detaching:**



## Thread Basics: Joining and Detaching III

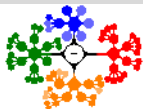
- When a thread is created, one of its attributes defines whether it is **joinable or detached**.
- Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable.
- To explicitly create a thread as joinable or detached, the **attr** argument in the *pthread\_create()* routine is used.
- **Detaching:**
- The **pthread\_detach()** routine can be used to explicitly detach a thread even though it was created as joinable.





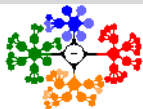
## Thread Basics: Joining and Detaching III

- When a thread is created, one of its attributes defines whether it is **joinable or detached**.
- Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable.
- To explicitly create a thread as joinable or detached, the **attr** argument in the `pthread_create()` routine is used.
- **Detaching:**
- The **pthread\_detach()** routine can be used to explicitly detach a thread even though it was created as joinable.
- If a thread requires joining, consider explicitly creating it as joinable (portability).



## Thread Basics: Joining and Detaching III

- When a thread is created, one of its attributes defines whether it is **joinable or detached**.
- Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable.
- To explicitly create a thread as joinable or detached, the **attr** argument in the `pthread_create()` routine is used.
- **Detaching:**
- The **pthread\_detach()** routine can be used to explicitly detach a thread even though it was created as joinable.
- If a thread requires joining, consider explicitly creating it as joinable (portability).
- If you know in advance that a thread will never need to join with another thread, consider creating it in a detached state (resources).





- **Reentrant functions** are those that can be safely called when another instance has been suspended in the middle of its invocation.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Reentrant functions** are those that can be safely called when another instance has been suspended in the middle of its invocation.
- All thread functions must be reentrant because a thread can be preempted in the middle of its execution.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- **Reentrant functions** are those that can be safely called when another instance has been suspended in the middle of its invocation.
- All thread functions must be reentrant because a thread can be preempted in the middle of its execution.
- If another thread starts executing the same function at this point, a non-reentrant function might not work as desired.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Synchronization Primitives: Mutex I

- While communication is implicit in shared-address-space programming,



## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Synchronization Primitives: Mutex I

- While communication is implicit in shared-address-space programming,
- much of the effort associated with writing correct threaded programs is spent on **synchronizing concurrent threads** with respect to their data accesses or scheduling.



## Thread Basics: Passing Arguments, Cancellation and Joining

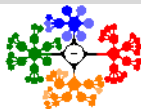
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- While communication is implicit in shared-address-space programming,
- much of the effort associated with writing correct threaded programs is spent on **synchronizing concurrent threads** with respect to their data accesses or scheduling.
- Using **pthread\_create** and **pthread\_join** calls, we can create concurrent tasks.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

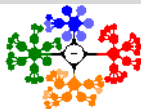
Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables





- While communication is implicit in shared-address-space programming,
- much of the effort associated with writing correct threaded programs is spent on **synchronizing concurrent threads** with respect to their data accesses or scheduling.
- Using **pthread\_create** and **pthread\_join** calls, we can create concurrent tasks.
- These tasks work together to manipulate data and accomplish a given task.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

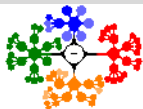
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- While communication is implicit in shared-address-space programming,
- much of the effort associated with writing correct threaded programs is spent on **synchronizing concurrent threads** with respect to their data accesses or scheduling.
- Using **pthread\_create** and **pthread\_join** calls, we can create concurrent tasks.
- These tasks work together to manipulate data and accomplish a given task.
- When multiple threads attempt to manipulate the same data item,

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- While communication is implicit in shared-address-space programming,
- much of the effort associated with writing correct threaded programs is spent on **synchronizing concurrent threads** with respect to their data accesses or scheduling.
- Using **pthread\_create** and **pthread\_join** calls, we can create concurrent tasks.
- These tasks work together to manipulate data and accomplish a given task.
- When multiple threads attempt to manipulate the same data item,
- the results can often be **incoherent** if proper care is not taken to synchronize them.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex II



- Consider the following code fragment being executed by multiple threads.

```
1  /* each thread tries to update variable best_cost
2                                     as follows */
3  if (my_cost < best_cost)
4      best_cost = my_cost;
```

### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex II



- Consider the following code fragment being executed by multiple threads.

```
1  /* each thread tries to update variable best_cost
2                                     as follows */
3  if (my_cost < best_cost)
4      best_cost = my_cost;
```

- The variable *my\_cost* is thread-local and *best\_cost* is a global variable shared by all threads.

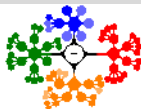
### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads  
Thread Cancellation  
Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex II



- Consider the following code fragment being executed by multiple threads.

```
1  /* each thread tries to update variable best_cost
2                                     as follows */
3  if (my_cost < best_cost)
4      best_cost = my_cost;
```

- The variable *my\_cost* is thread-local and *best\_cost* is a global variable shared by all threads.
- This is an undesirable situation, sometimes also referred to as a **race condition**.

### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

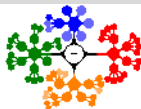
Thread Cancellation

Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex II



- Consider the following code fragment being executed by multiple threads.

```
1  /* each thread tries to update variable best_cost
2                                     as follows */
3  if (my_cost < best_cost)
4      best_cost = my_cost;
```

- The variable *my\_cost* is thread-local and *best\_cost* is a global variable shared by all threads.
- This is an undesirable situation, sometimes also referred to as a **race condition**.
- So called because the result of the computation depends on the race between competing threads.

### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

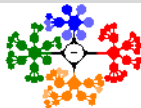
Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex III

- To understand the problem with shared data access, let us examine one execution instance of the above code fragment.



### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

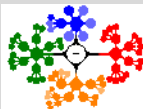
### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



## Synchronization Primitives: Mutex III

- To understand the problem with shared data access, let us examine one execution instance of the above code fragment.
- Assume that there are two threads,



### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex III

- To understand the problem with shared data access, let us examine one execution instance of the above code fragment.
- Assume that there are two threads,
- The initial value of *best\_cost* is 100,



### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

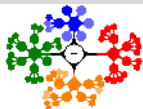
Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex III

- To understand the problem with shared data access, let us examine one execution instance of the above code fragment.
- Assume that there are two threads,
- The initial value of *best\_cost* is 100,
- The values of *my\_cost* are 50 and 75 at threads t1 and t2, respectively.



### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

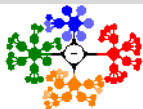
Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

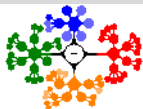
## Synchronization Primitives: Mutex III

- To understand the problem with shared data access, let us examine one execution instance of the above code fragment.
- Assume that there are two threads,
- The initial value of *best\_cost* is 100,
- The values of *my\_cost* are 50 and 75 at threads t1 and t2, respectively.
- If both threads execute the condition inside the if statement concurrently, then both threads enter the then part of the statement.



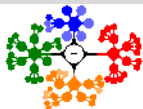
## Synchronization Primitives: Mutex III

- To understand the problem with shared data access, let us examine one execution instance of the above code fragment.
- Assume that there are two threads,
- The initial value of *best\_cost* is 100,
- The values of *my\_cost* are 50 and 75 at threads t1 and t2, respectively.
- If both threads execute the condition inside the if statement concurrently, then both threads enter the then part of the statement.
- Depending on which thread executes first, the value of *best\_cost* at the end could be either 50 or 75.



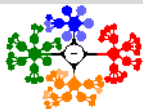
## Synchronization Primitives: Mutex III

- To understand the problem with shared data access, let us examine one execution instance of the above code fragment.
- Assume that there are two threads,
- The initial value of *best\_cost* is 100,
- The values of *my\_cost* are 50 and 75 at threads t1 and t2, respectively.
- If both threads execute the condition inside the if statement concurrently, then both threads enter the then part of the statement.
- Depending on which thread executes first, the value of *best\_cost* at the end could be either 50 or 75.
- **There are two problems here:**



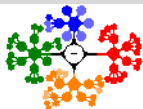
## Synchronization Primitives: Mutex III

- To understand the problem with shared data access, let us examine one execution instance of the above code fragment.
- Assume that there are two threads,
- The initial value of *best\_cost* is 100,
- The values of *my\_cost* are 50 and 75 at threads t1 and t2, respectively.
- If both threads execute the condition inside the if statement concurrently, then both threads enter the then part of the statement.
- Depending on which thread executes first, the value of *best\_cost* at the end could be either 50 or 75.
- There are two problems here:
  - ❶ non-deterministic nature of the result;



## Synchronization Primitives: Mutex III

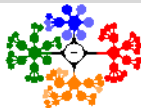
- To understand the problem with shared data access, let us examine one execution instance of the above code fragment.
- Assume that there are two threads,
- The initial value of *best\_cost* is 100,
- The values of *my\_cost* are 50 and 75 at threads t1 and t2, respectively.
- If both threads execute the condition inside the if statement concurrently, then both threads enter the then part of the statement.
- Depending on which thread executes first, the value of *best\_cost* at the end could be either 50 or 75.
- There are two problems here:
  - ① non-deterministic nature of the result;
  - ② more importantly, the value 75 of *best\_cost* is inconsistent in the sense that no serialization of the two threads can possibly yield this result.





## Synchronization Primitives: Mutex IV

- Race condition occurred because the test-and-update operation is an **atomic operation**;



### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex IV

- Race condition occurred because the test-and-update operation is an **atomic operation**;
  - i.e., the operation should not be broken into sub-operations.



### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex IV

- Race condition occurred because the test-and-update operation is an **atomic operation**;
  - i.e., the operation should not be broken into sub-operations.
- Furthermore, the code corresponds to a **critical segment**;



### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex IV

- Race condition occurred because the test-and-update operation is an **atomic operation**;
  - i.e., the operation should not be broken into sub-operations.
- Furthermore, the code corresponds to a **critical segment**;
  - i.e., a segment that must be executed by only one thread at any time.



### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex IV

- Race condition occurred because the test-and-update operation is an **atomic operation**;
  - i.e., the operation should not be broken into sub-operations.
- Furthermore, the code corresponds to a **critical segment**;
  - i.e., a segment that must be executed by only one thread at any time.
- Many statements that seem atomic in higher level languages such as C may in fact be non-atomic.



## Synchronization Primitives: Mutex IV

- Race condition occurred because the test-and-update operation is an **atomic operation**;
  - i.e., the operation should not be broken into sub-operations.
- Furthermore, the code corresponds to a **critical segment**;
  - i.e., a segment that must be executed by only one thread at any time.
- Many statements that seem atomic in higher level languages such as C may in fact be non-atomic.
  - i.e., a statement of the form *global\_count*+ = 5 may comprise several assembler instructions and therefore must be handled carefully.



## Synchronization Primitives: Mutex IV

- Race condition occurred because the test-and-update operation is an **atomic operation**;
  - i.e., the operation should not be broken into sub-operations.
- Furthermore, the code corresponds to a **critical segment**;
  - i.e., a segment that must be executed by only one thread at any time.
- Many statements that seem atomic in higher level languages such as C may in fact be non-atomic.
  - i.e., a statement of the form  $global\_count+ = 5$  may comprise several assembler instructions and therefore must be handled carefully.
- Threaded APIs provide support for implementing critical sections and atomic operations using **mutex**-locks (mutual exclusion locks).



# Synchronization Primitives: Mutex V

- Mutex-locks have two states: locked and unlocked.



## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

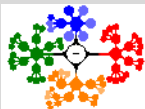
## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



# Synchronization Primitives: Mutex V

- Mutex-locks have two states: locked and unlocked.
- At any point of time, **only one thread can lock a mutex lock**.



## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

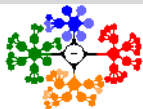
Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Synchronization Primitives: Mutex V

- Mutex-locks have two states: locked and unlocked.
- At any point of time, **only one thread can lock a mutex lock**.
- A lock is an atomic operation.



## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

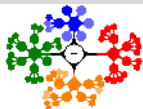
Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Synchronization Primitives: Mutex V

- Mutex-locks have two states: locked and unlocked.
- At any point of time, **only one thread can lock a mutex lock**.
- A lock is an atomic operation.
  - To access the shared data, a thread must first try to acquire a mutex-lock.





- Mutex-locks have two states: locked and unlocked.
- At any point of time, **only one thread can lock a mutex lock**.
- A lock is an atomic operation.
  - To access the shared data, a thread must first try to acquire a mutex-lock.
  - If the mutex-lock is already locked, the process trying to acquire the lock is **blocked**.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- Mutex-locks have two states: locked and unlocked.
- At any point of time, **only one thread can lock a mutex lock**.
- A lock is an atomic operation.
  - To access the shared data, a thread must first try to acquire a mutex-lock.
  - If the mutex-lock is already locked, the process trying to acquire the lock is **blocked**.
  - This is because a locked mutex-lock implies that there is another thread currently in the critical section and that no other thread must be allowed in.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

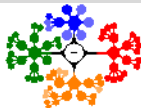
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- Mutex-locks have two states: locked and unlocked.
- At any point of time, **only one thread can lock a mutex lock**.
- A lock is an atomic operation.
  - To access the shared data, a thread must first try to acquire a mutex-lock.
  - If the mutex-lock is already locked, the process trying to acquire the lock is **blocked**.
  - This is because a locked mutex-lock implies that there is another thread currently in the critical section and that no other thread must be allowed in.
  - **When a thread leaves a critical section, it must unlock the mutex-lock so that other threads can enter the critical section.**

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

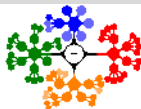
Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Synchronization Primitives: Mutex V



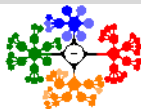
- Mutex-locks have two states: locked and unlocked.
- At any point of time, **only one thread can lock a mutex lock**.
- A lock is an atomic operation.
  - To access the shared data, a thread must first try to acquire a mutex-lock.
  - If the mutex-lock is already locked, the process trying to acquire the lock is **blocked**.
  - This is because a locked mutex-lock implies that there is another thread currently in the critical section and that no other thread must be allowed in.
  - When a thread leaves a critical section, it must unlock the mutex-lock so that other threads can enter the critical section.
- All mutex-locks must be initialized to the unlocked state at the beginning of the program.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads  
Thread Cancellation  
Joining and Detaching  
Threads

Synchronization  
Primitives in PThreads

Mutual Exclusion for Shared  
Variables



- The function **pthread\_mutex\_lock**;

```
1  int
2  pthread_mutex_lock (
3      pthread_mutex_t *mutex_lock);
```

## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

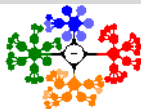
Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



# Synchronization Primitives: Mutex VI



- The function **pthread\_mutex\_lock**;

```
1  int
2  pthread_mutex_lock (
3      pthread_mutex_t *mutex_lock);
```

- A call to this function attempts a lock on the mutex-lock *mutex\_lock*.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Synchronization Primitives: Mutex VI



- The function **pthread\_mutex\_lock**;

```
1  int
2  pthread_mutex_lock (
3      pthread_mutex_t *mutex_lock);
```

- A call to this function attempts a lock on the mutex-lock *mutex\_lock*.
- The data type of a *mutex\_lock* is predefined to be *pthread\_mutex\_t*.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function **pthread\_mutex\_lock**;

```
1  int
2  pthread_mutex_lock (
3      pthread_mutex_t *mutex_lock);
```

- A call to this function attempts a lock on the mutex-lock *mutex\_lock*.
- The data type of a *mutex\_lock* is predefined to be *pthread\_mutex\_t*.
- If the mutex-lock is already locked, the calling thread blocks; otherwise the mutex-lock is locked and the calling thread returns.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function **pthread\_mutex\_lock**;

```
1  int
2  pthread_mutex_lock (
3      pthread_mutex_t *mutex_lock);
```

- A call to this function attempts a lock on the mutex-lock *mutex\_lock*.
- The data type of a *mutex\_lock* is predefined to be *pthread\_mutex\_t*.
- If the mutex-lock is already locked, the calling thread blocks; otherwise the mutex-lock is locked and the calling thread returns.
- A successful return from the function returns a value 0. Other values indicate error conditions such as deadlocks.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function **pthread\_mutex\_unlock**;

```
1  int
2  pthread_mutex_unlock (
3      pthread_mutex_t *mutex_lock);
```

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function **pthread\_mutex\_unlock**;

```
1  int  
2  pthread_mutex_unlock (  
3      pthread_mutex_t *mutex_lock);
```

- On leaving a critical section, a thread must **unlock the mutex-lock** associated with the section.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function **pthread\_mutex\_unlock**;

```
1  int
2  pthread_mutex_unlock (
3      pthread_mutex_t *mutex_lock);
```

- On leaving a critical section, a thread must **unlock the mutex-lock** associated with the section.
- If it does not do so, no other thread will be able to enter this section, typically resulting in a deadlock.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

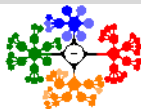
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function **pthread\_mutex\_unlock**;

```
1  int
2  pthread_mutex_unlock (
3      pthread_mutex_t *mutex_lock);
```

- On leaving a critical section, a thread must **unlock the mutex-lock** associated with the section.
- If it does not do so, no other thread will be able to enter this section, typically resulting in a deadlock.
- On calling **pthread\_mutex\_unlock** function, the lock is relinquished and one of the blocked threads is **scheduled** to enter the critical section.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



# Synchronization Primitives: Mutex VIII

- The specific thread is determined by the **scheduling policy**.



## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

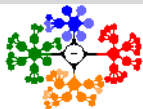
Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex VIII

- The specific thread is determined by the **scheduling policy**.
- if the thread priority scheduling is not implied, the assignment will be left to the native system scheduler and may appear to be more or less **random**.



### Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

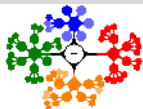
Joining and Detaching  
Threads

### Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

## Synchronization Primitives: Mutex VIII

- The specific thread is determined by the **scheduling policy**.
- if the thread priority scheduling is not implied, the assignment will be left to the native system scheduler and may appear to be more or less **random**.
- **Mutex variables** must be declared with type *pthread\_mutex\_t*, and must be initialized before they can be used.



Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

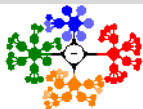
Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

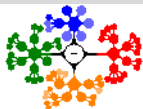
## Synchronization Primitives: Mutex VIII

- The specific thread is determined by the **scheduling policy**.
- if the thread priority scheduling is not implied, the assignment will be left to the native system scheduler and may appear to be more or less **random**.
- **Mutex variables** must be declared with type `pthread_mutex_t`, and must be initialized before they can be used.
- There are two ways to initialize a mutex variable:



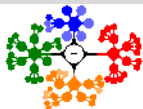
## Synchronization Primitives: Mutex VIII

- The specific thread is determined by the **scheduling policy**.
- if the thread priority scheduling is not implied, the assignment will be left to the native system scheduler and may appear to be more or less **random**.
- **Mutex variables** must be declared with type `pthread_mutex_t`, and must be initialized before they can be used.
- There are two ways to initialize a mutex variable:
  - 1 **Statically, when it is declared. For example:**  
`pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`



## Synchronization Primitives: Mutex VIII

- The specific thread is determined by the **scheduling policy**.
- if the thread priority scheduling is not implied, the assignment will be left to the native system scheduler and may appear to be more or less **random**.
- **Mutex variables** must be declared with type `pthread_mutex_t`, and must be initialized before they can be used.
- There are two ways to initialize a mutex variable:
  - 1 Statically, when it is declared. For example:  
`pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`
  - 2 Dynamically, with the **`pthread_mutex_init()`** routine. This method permits setting mutex object attributes, *attr*.

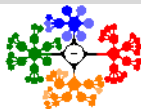


## Synchronization Primitives: Mutex VIII

- The specific thread is determined by the **scheduling policy**.
- if the thread priority scheduling is not implied, the assignment will be left to the native system scheduler and may appear to be more or less **random**.
- **Mutex variables** must be declared with type `pthread_mutex_t`, and must be initialized before they can be used.
- There are two ways to initialize a mutex variable:
  - 1 Statically, when it is declared. For example:

```
pthread_mutex_t mymutex =  
PTHREAD_MUTEX_INITIALIZER;
```
  - 2 Dynamically, with the **pthread\_mutex\_init()** routine. This method permits setting mutex object attributes, *attr*.
- If a programmer attempts a **pthread\_mutex\_unlock** on a previously unlocked mutex or one that is locked by another thread, the effect is undefined.





- The function `pthread_mutex_init`;

```
1  int
2  pthread_mutex_init (
3      pthread_mutex_t *mutex_lock,
4      const pthread_mutexattr_t *lock_attr);
```

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

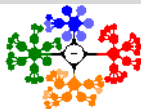
Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables





- The function `pthread_mutex_init`;

```
1  int
2  pthread_mutex_init (
3      pthread_mutex_t *mutex_lock,
4      const pthread_mutexattr_t *lock_attr);
```

- We need one more function before we can start using mutex-locks, namely, a function to initialize a mutex-lock to its unlocked state.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function `pthread_mutex_init`;

```
1  int
2  pthread_mutex_init (
3      pthread_mutex_t *mutex_lock,
4      const pthread_mutexattr_t *lock_attr);
```

- We need one more function before we can start using mutex-locks, namely, a function to initialize a mutex-lock to its unlocked state.
- The mutex is initially unlocked.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function `pthread_mutex_init`;

```
1  int
2  pthread_mutex_init (
3      pthread_mutex_t *mutex_lock,
4      const pthread_mutexattr_t *lock_attr);
```

- We need one more function before we can start using mutex-locks, namely, a function to initialize a mutex-lock to its unlocked state.
- The mutex is initially unlocked.
- The **attributes** of the mutex-lock are specified by *lock\_attr*.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function `pthread_mutex_init`;

```
1  int
2  pthread_mutex_init (
3      pthread_mutex_t *mutex_lock,
4      const pthread_mutexattr_t *lock_attr);
```

- We need one more function before we can start using mutex-locks, namely, a function to initialize a mutex-lock to its unlocked state.
- The mutex is initially unlocked.
- The attributes of the mutex-lock are specified by `lock_attr`.
- If this argument is set to `NULL`, the default mutex-lock attributes are used (normal mutex-lock).

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

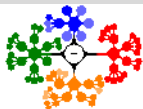
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- Locks represent serialization points since critical sections must be executed by threads one after the other.

## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Synchronization Primitives: Overheads of Locking I

- Locks represent serialization points since critical sections must be executed by threads one after the other.
- Encapsulating large segments of the program within locks can, therefore, lead to **significant performance degradation**.



## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

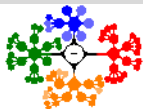
Thread Cancellation

Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Synchronization Primitives: Overheads of Locking I



- Locks represent serialization points since critical sections must be executed by threads one after the other.
- Encapsulating large segments of the program within locks can, therefore, lead to **significant performance degradation**.
- It is therefore important to minimize the size of critical sections and to handle critical sections and shared data structures with extreme care.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

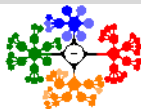
Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables

# Synchronization Primitives: Overheads of Locking I



- Locks represent serialization points since critical sections must be executed by threads one after the other.
- Encapsulating large segments of the program within locks can, therefore, lead to **significant performance degradation**.
- It is therefore important to minimize the size of critical sections and to handle critical sections and shared data structures with extreme care.
- It is often possible to reduce the idling overhead associated with locks using an alternate function, *pthread\_mutex\_trylock*.

## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

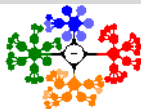
Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



# Synchronization Primitives: Overheads of Locking I



- Locks represent serialization points since critical sections must be executed by threads one after the other.
- Encapsulating large segments of the program within locks can, therefore, lead to **significant performance degradation**.
- It is therefore important to minimize the size of critical sections and to handle critical sections and shared data structures with extreme care.
- It is often possible to reduce the idling overhead associated with locks using an alternate function, *pthread\_mutex\_trylock*.
- It does not have to deal with queues associated with locks for multiple threads waiting on the lock.

## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function `pthread_mutex_trylock`;

```
1  int
2  pthread_mutex_trylock (
3      pthread_mutex_t *mutex_lock);
```

## Thread Basics: Passing Arguments, Cancellation and Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

## Synchronization Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function `pthread_mutex_trylock`;

```
1  int
2  pthread_mutex_trylock (
3      pthread_mutex_t *mutex_lock);
```

- This function attempts a lock on *mutex\_lock*.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function **pthread\_mutex\_trylock**;

```
1  int
2  pthread_mutex_trylock (
3      pthread_mutex_t *mutex_lock);
```

- This function attempts a lock on *mutex\_lock*.
  - If the lock is successful, the function returns a zero.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function `pthread_mutex_trylock`;

```
1  int
2  pthread_mutex_trylock (
3      pthread_mutex_t *mutex_lock);
```

- This function attempts a lock on *mutex\_lock*.
  - If the lock is successful, the function returns a zero.
  - If it is already locked by another thread, **instead of blocking** the thread execution, it returns a value *EBUSY*.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

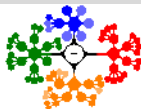
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function `pthread_mutex_trylock`;

```
1  int
2  pthread_mutex_trylock (
3      pthread_mutex_t *mutex_lock);
```

- This function attempts a lock on *mutex\_lock*.
  - If the lock is successful, the function returns a zero.
  - If it is already locked by another thread, **instead of blocking** the thread execution, it returns a value *EBUSY*.
  - **This allows the thread to do other work and to poll the mutex for a lock.**

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

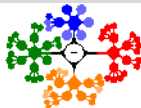
Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables



- The function `pthread_mutex_trylock`;

```
1  int
2  pthread_mutex_trylock (
3      pthread_mutex_t *mutex_lock);
```

- This function attempts a lock on *mutex\_lock*.
  - If the lock is successful, the function returns a zero.
  - If it is already locked by another thread, **instead of blocking** the thread execution, it returns a value *EBUSY*.
  - This allows the thread to **do other work** and to poll the mutex for a lock.
- Furthermore, `pthread_mutex_trylock` is typically much faster than `pthread_mutex_lock` on typical systems.

Thread Basics:  
Passing Arguments,  
Cancellation and  
Joining

Passing Arguments to  
Threads

Thread Cancellation

Joining and Detaching  
Threads

Synchronization  
Primitives in Pthreads

Mutual Exclusion for Shared  
Variables