

# Thread

Pangfeng Liu  
National Taiwan University

April 4, 2015

# Process and Threads


- A process is a program in execution.
- A computer may have multiple processes running simultaneously.
- A thread is an execution instance of a process.
- A process may have multiple threads running simultaneously.

# UNIX Process

A UNIX process has the following information <sup>1</sup>.

- Process ID, process group ID, user ID, and group ID
- Environment variables
- Working directory
- Program instructions
- Registers
- File descriptors
- Signal actions

---

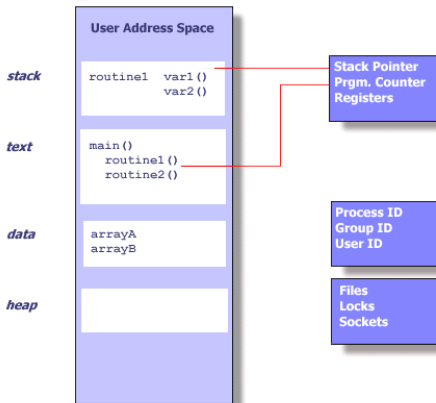
<sup>1</sup><https://computing.llnl.gov/tutorials/pthreads> 

# UNIX Process

A UNIX process may place the data in the following sections.

- Stack
  - Variables declared within functions.
- Heap
  - Storage allocated dynamically (e.g., malloc).
- Data
  - Variables declared globally.

# UNIX Process



2

<sup>2</sup>[https:](https://computing.llnl.gov/tutorials/pthreads/images/process.gif)

[//computing.llnl.gov/tutorials/pthreads/images/process.gif](https://computing.llnl.gov/tutorials/pthreads/images/process.gif)

# Discussion

- How to create a process in a UNIX environment?
- How to kill a process in a UNIX environment?

# Thread

- A thread executes within a process, and has its own resources – program text, program counter, stack, private variables, etc.
- A thread runs on a core, and multiple threads can run on multiple cores simultaneously.
- We use threads to explore parallelism.

# Parallelism

- We want to express parallelism with multi-threading on a multiprocessor because multiple threads can run on multiple cores simultaneously.
- We do *not* use processes since the communication among them require expensive inter-process-communication (IPC).
  - Threads within a process can easily share variables in a shared memory multiprocessor.
- We do *not* use processes since the process creation is expensive.
  - Thread creation is much cheaper.




# Thread

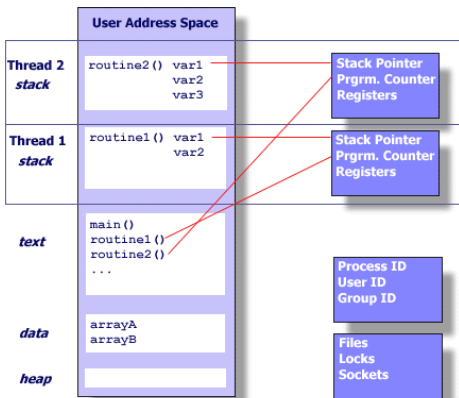
- Threads exist within a process and share some of the process resources.
- Threads have their own independent flows of control.
- Threads die if the process dies.
- Threads are "lightweight" because most of the overhead has already been accomplished through the creation of its process.

3

---

<sup>3</sup><https://computing.llnl.gov/tutorials/pthreads> 

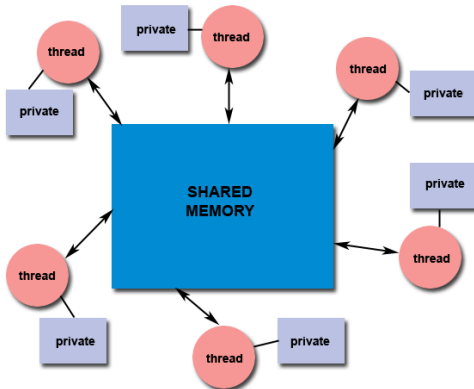
# Threads within a Process



# Memory Model

- All threads have access to the same global shared memory and programmers are responsible for synchronizing access (protecting) them.
- Threads also have their own private data for their own private usage.

# Shared Memory



5

<sup>5</sup><https://computing.llnl.gov/tutorials/pthreads/images/sharedmemoryModel.gif>


# Pthread

- Historically, hardware vendors have implemented their own proprietary versions of threads.
- These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

# Pthread

- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required<sup>6</sup>.
- For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
- Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
- Most hardware vendors now offer Pthreads in addition to their proprietary API's.

---

<sup>6</sup><https://computing.llnl.gov/tutorials/pthreads> 

# Discussion

- Describe a “standard” in computer science, and why is it important.

# Hello

## Example 1: (hello.c) Main program

```
7 #define NUM_ID 8
8 int main(int argc, char *argv[])
9 {
10     for (int t = 0; t < NUM_ID; t++) {
11         printf("main: t = %d\n", t);
12         PrintHello(t);
13     }
14     return 0;
15 }
```



# Hello

## Example 2: (hello.c) Print

```
2 void PrintHello(int id)
3 {
4     printf("printHello: id = %d\n", id);
5 }
```

# Demonstration

- Run the hello.c program.

# Implementation

- Implement a program that prints 10 lines of “hello”.

# Discussion

- Describe the compiling environment you used.

# Pthread Concepts

- Initially there is only one *main thread*.
- The main thread can spawn other threads. For ease of explanation we will refer to them as the *spawned* threads of the main thread.
- For ease of explanation we will also use *child* threads to refer to those threads spawned by the main thread.
- You have been programming the main thread *only*.

# Pthread Steps

- Include the necessary header file `pthread.h`.
- Declare a variable of type `pthread_t` to store the identifier for the thread you will create.
- Call `pthread_create` creates a thread with the type `pthread_t` variable declared previously.
- Call `pthread_exit` when the main thread or the spawn thread wants to exit.

# Hello

## Example 3: (hello-pthread.c) Declaration

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #define NUM_THREADS 8
```

- Must include pthread.h.

# pthread\_create

## Example 4: pthread\_create

```
1 int pthread_create(pthread_t *thread,  
2   const pthread_attr_t *attr,  
3   void *(*start_routine)(void*), void *arg);
```



## pthread\_create Parameters

**thread** A pthread\_t pointer to the created thread.

**attr** A pthread\_type\_t pointer to the attribute of the thread.

We can set it to NULL for now.

## pthread\_create Parameters

**start\_routine** Thread starting routine. The routine must have `void *routine(void *)` prototype. i.e., it expects a pointer to void as the only parameter, and returns a pointer to void.

**arg** A pointer to an optional parameter to the spawned thread. We set it to the thread index `t` in our example.

**return value** Return 0 if success, and a non-zero error code on error.

# pthread\_exit

## Example 5: pthread\_exit

```
1 void pthread_exit(void *value);
```

**value** A thread can use this pointer to sent information back to its parent thread. We can set it to NULL for now because the threads will not send anything back to the main thread.

## Discussion

- Name the functions that create and destroy pthreads respectively.

# Hello with Pthread

## Example 6: (hello-pthread.c) Main program

```
13 int main(int argc, char *argv[])
14 {
15     pthread_t threads[NUM_THREADS];
16     for(int t = 0; t < NUM_THREADS; t++) {
17         printf("main: create thread %d\n", t);
18         int rc =
19             pthread_create(&threads[t], NULL,
20                           printHello, (void *)&t));
21         if (rc) {
22             printf("main: error code %d\n", rc);
23             exit(-1);
24         }
25     }
26     pthread_exit(NULL);
27     return 0;
28 }
```

# Main Program

- We can use the argument `arg` to pass information to the spawned threads.
- In our example we pass the index `t` to the spawned thread as thread index.
- Note that the type of `arg` is `void *` since we do not know what kind of parameter will be passed into a thread.

# Hello with Pthread

## Example 7: (hello-pthread.c) Print

```
6 void *printHello(void *thread_id)
7 {
8     int tid = *((int *)thread_id);
9     printf("printHello: tid = %d\n", tid);
10    pthread_exit(NULL);
11 }
```

# printHello

- This is the routine each spawned thread will run.
- Each thread receives a pointer to the thread index (`thread_id`) from the main thread, and prints it.
- We need to cast the type of the parameter back to `(int *)`, very much like the case of `qsort`.
- We do not have anything to send back to the main thread, so we use `NULL` in `pthread_exit(NULL)`.



# Compile and Link

- You need to link your program with the pthread library.
- You can use gcc to compile and link you program with pthread library. Note that the library must be at the end of the gcc command.
  - `gcc program.c -o program -lpthread`

# Demonstration

- Run the hello-pthread.c program.

## Discussion

- Is the answer correct? If not what is the reason?

# Hello with Pthread

## Example 8: (hello-pthread-correct.c) main

```
13 int main(int argc, char *argv[])
14 {
15     pthread_t threads[NUM_THREADS];
16     int threadIndex[NUM_THREADS];
17     for(int t = 0; t < NUM_THREADS; t++) {
18         printf("main: create thread %d\n", t);
19         threadIndex[t] = t;
20         int rc =
21             pthread_create(&threads[t], NULL,
22                           printHello,
23                           (void *)(&threadIndex + t));
24         if (rc) {
25             printf("main: error code %d\n", rc);
26             exit(-1);
27         }
28     }
29     pthread_exit(NULL);
30     return 0;
31 }
```

# Main Thread Variables

- We use an array `threadIndex` to keep the thread index for all threads.
- We then pass the address of the index to the spawned threads. This prevents the threads from accessing the same memory address in the previous example.
- From this example we know that the spawned threads can access the memory of the main thread.

# Demonstration

- Run the hello-pthread-correct.c program.
- Is the answer correct?
- Is there timing constraints between these threads?

# Implementation

- Implement a pthread program that prints 10 lines of “hello”. You should use `pthread_create` to create 10 threads to do this, where each thread prints an index from 0 to 9.

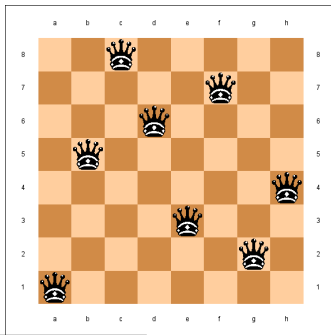
## Discussion

- Does your program function properly if you remove the `pthread_exit` from the main program? Explain the reason if it does not.



# Eight Queen

- Place eight queen on a chessboard so that no queen can attack any other queen.<sup>7</sup>
- We would like to know the number of such solutions.



<sup>7</sup><http://support.sas.com/documentation/cdl/en/orcpug/59630/HTML/default/images/queens.png>

# Solution

- Use an array `position` to store the positions of  $n$  queens.
- The  $i$ -th element of the array stores the row index of the queen at the  $i$ -column.
- Receive the size of the board  $n$  as the second command line argument.
- Use a recursive function `queen` to compute the number of solutions.

# Headers

## Example 9: (queen.c) Headers

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4 #define MAXN 20
```

# Headers

- `stdlib.h` for `atoi`.
- `MAXN` is the maximum size of the board.

# Eight Queen

## Example 10: (queen.c) Main program

```
29 int main(int argc, char *argv[])
30 {
31     int position[MAXN];
32     assert(argc == 2);
33     int n = atoi(argv[1]);
34     assert(n <= MAXN);
35     printf("%d\n", queen(position, 0, n));
36     return 0;
37 }
```

## main

- position for queen positions.
- n is the size of the board.
- queen will return the number of solutions.

# Eight Queen

## Example 11: (queen.c) queen

```
16 int queen(int position[], int next, int n)
17 {
18     if (next >= n)
19         return 1;
20     int sum = 0;
21     for (int test = 0; test < n; test++)
22         if (ok(position, next, test)) {
23             position[next] = test;
24             sum += queen(position, next + 1, n);
25         }
26     return sum;
27 }
```

# queen

- The `queen` function computes the number of solutions.
- The `queen` uses a parameter `next` to keep track of the column number it wishes to place a queen next.
- If `next` is already  $n$  then we have placed all  $n$  queens.
- Otherwise we try all rows and sum up the number of solutions from all possible row placements.



# Eight Queen

Example 12: (queen.c) ok

```
7  int ok(int position[], int next, int test)
8  {
9      for (int i = 0; i < next; i++)
10         if (position[i] == test ||
11             (abs(test - position[i]) == next - i))
12             return 0;
13     return 1;
14 }
```

ok

- The queen function determines if we could place a queen at the end of position.
- The newly added queen is at row position[next], column next.
- if any previously placed queen is at the same row or at the diagonal of the newly placed queen, the function returns 0. Otherwise it return 1.

# Demonstration

- Run and time the queen.c program.

# Implementation

- Implement a sequential program that solves the eight queen problem.

## Discussion

- Can you think of any simple optimization that will speed up your program?

## $n$ Queen with pthread

Now we want to parallelize the previous program with pthread.

- We create  $n$  threads, and use one thread to search each of the  $n$  subtrees when we place the first queen at all  $n$  possible cells in the first column.
- All threads can run in parallel.
- Right now we just let each thread report the number of answers it finds, and do not sum them.

# Issues

- How do all thread know the size of the board?
  - We declare a global `n`.
- How does a thread know which subtree it should solve?
  - We pass this information to the spawned threads by the `arg` parameter in `pthread_create`.
- Can these threads share a board?
  - No, they need their own position.
- Do these threads need to communicate with each other?
  - No, because right now we do not sum the numbers of solutions.

# Headers

## Example 13: (queen-pthread.c) Headers

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <assert.h>
5
6 #define MAXN 20
7 int n; /* a global n */
```



# Headers

- pthread.h for pthread functions.
- stdlib.h for atoi.
- MAXN is the maximum size of the board.
- A global n for all threads.

# Eight Queen with Pthread

## Example 14: (queen-pthread.c) Main program

```
39 int main (int argc, char *argv[])
40 {
41     assert(argc == 2);
42     n = atoi(argv[1]);
43     assert(n <= MAXN);
44     int *position;
45     pthread_t threads[MAXN];
46     for(int i = 0; i < n; i++) {
47         position = (int *)calloc(n, sizeof(int));
48         assert(position != NULL);
49         position[0] = i;
50         int error = pthread_create(&threads[i], NULL, goQueen,
51                                   (void *)position);
52         assert(error == 0);
53     }
54     pthread_exit(NULL);
55 }
```

## main

- We call `calloc` to allocate an array and keep the starting address in `position`.
- We set the first element of `position` according to the thread index, so that each thread searches its own subtree.
- We pass the `position` to `pthread_create` so that the threads can access their own arrays.
- From this example we know that all threads share the global variables and heap space.

## goQueen

## Example 15: (queen-pthread.c) goqueen

```
31 void *goQueen(void *pos)
32 {
33     int *position = (int *)pos;
34     printf("goQueen: thread %d, # of solution = %d\n",
35           position[0], queen(position, 1));
36     pthread_exit(NULL);
37 }
```

## goQueen

- goQueen is the starting function of spawned threads.
- goQueen uses its own pointer petition to point to the array in the heap. Some casting is required.
- goQueen calls queen with next set to 1.
- All threads can get the board size from the global variable n.

# Demonstration

- Run and time the queen-pthread.c program.

# Implementation

- Implement a parallel program that solves the  $n$  queen problem with pthread. Each thread only reports the number of solutions it finds and no summation is required.
- Measure the speedup and efficiency.

# Discussion

- Why do we call queen with 1?
- What are the speedup and efficiency you are getting?
- Is the workload evenly distributed?



# Total Number

- Now we want to compute the total number of solutions, instead of individual number of solution from each thread.
- We use a global variable `numSolution` to store the sum of the numbers of solutions from all threads. This variable is initialized to 0 automatically.

# Headers

## Example 16: (queen-pthread-sum.c) Headers

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <assert.h>
5
6 #define MAXN 20
7 int n;      /* a global n */
8 int numSolution;
```

# Headers

- `pthread.h` for pthread functions.
- `stdlib.h` for `atoi`.
- `MAXN` is the maximum size of the board.
- A global `n` for all threads.
- A global `numSolution` for the total number of solutions.

# Eight Queen with Pthread

## Example 17: (queen-pthread-sum.c) Main program

```
42 int main (int argc, char *argv[])
43 {
44     assert(argc == 2);
45     n = atoi(argv[1]);
46     assert(n <= MAXN);
47     int *position;
48     pthread_t threads[MAXN];
49     for(int i = 0; i < n; i++) {
50         position = (int *)calloc(n, sizeof(int));
51         assert(position != NULL);
52         position[0] = i;
53         int error = pthread_create(&threads[i], NULL, goQueen,
54                                   (void *)position);
55         assert(error == 0);
56     }
57     printf("total # of solution %d\n", numSolution);
58     pthread_exit(NULL);
59     return 0;
60 }
```

## main

- We print the number of solutions numSolution before we call pthread\_exit.
- If we place the printf after pthread\_exit, it will not be executed.

## goQueen

## Example 18: (queen-pthread-sum.c) goqueen

```
32 void *goQueen(void *pos)
33 {
34     int *position = (int *)pos;
35     int num = queen(position, 1);
36     printf("goQueen: thread %d, # of solution = %d\n",
37           position[0], num);
38     numSolution += num;
39     pthread_exit(NULL);
40 }
```

# goQueen

- goQueen reports the number of solution queen finds.
- goQueen also adds the number of solutions computed by queen to the global variable numSolution.

# Demonstration

- Run and time the queen-pthread-sum.c program.



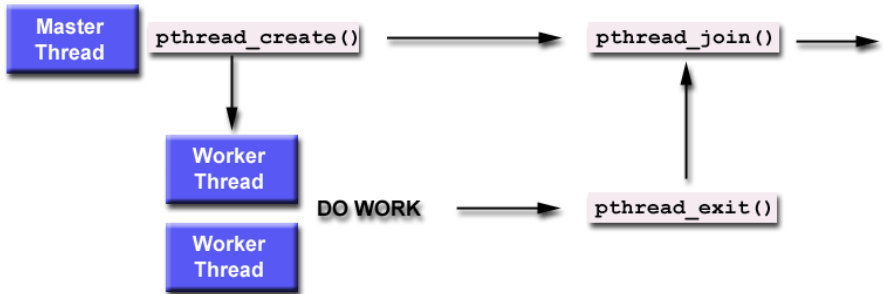
# Discussion

- Does the program produce correct output?

# Synchronization

- The problem of the previous program is that the main thread did *not* wait for all spawned threads to complete.
- We need a *barrier synchronization*, and after that the main thread can get the correct total number of solutions in `numSolution`.
- The main thread can wait for its spawned threads by calling `pthread_join`.

# Join a Pthread




# Pthread Attribute

Steps to set threads as joinable.<sup>8</sup>

- Declare a pthread attribute variable of type `pthread_attr_t`.
- Initialize the attribute variable with `pthread_attr_init()`.
- Set the attribute detached status with `pthread_attr_setdetachstate()`.
- When done, free library resources used by the attribute with `pthread_attr_destroy()`.

---

<sup>8</sup><https://computing.llnl.gov/tutorials/pthreads/> 

# pthread\_attr\_init

Example 19: (pthread-attr-init.c) Initialize/destroy attributes

```
1 int pthread_attr_init(pthread_attr_t *attr);  
2 int pthread_attr_destroy(pthread_attr_t *attr);
```

- Initialize and destroy the attribute of a thread.

# pthread\_attr\_setdetachstate

Example 20: (pthread-attr-setdetachstate.c) Set the attribute

```
1 int pthread_attr_setdetachstate(pthread_attr_t *attr,  
2                               int detachstate);
```

- Set the attribute of a thread.
- In our example we want to set the threads so that they can join with the main thread, so we will set `PTHREAD_CREATE_JOINABLE`.

# $n$ Queen with Pthread

Example 21: (queen-pthread-sum-join.c) Main program

```
44 int main (int argc, char *argv[])
45 {
46     assert(argc == 2);
47     n = atoi(argv[1]);
48     assert(n <= MAXN);
49     int *position;
50     pthread_t threads[MAXN];
51     pthread_attr_t attr;
52     pthread_attr_init(&attr);
53     pthread_attr_setdetachstate(&attr,
54         PTHREAD_CREATE_JOINABLE);
```

# Steps

- Declare a variable `attr` of type `pthread_attr_t` and initialize it with `pthread_attr_init()`.
- Set the attribute to `PTHREAD_CREATE_JOINABLE` with `pthread_attr_setdetachstate()`.
- Use the address of `attr` as the second argument in `pthread_create` while creating threads.
- Destroy `attr`.



# $n$ Queen with Pthread

## Example 22: (queen-pthread-sum-join.c) Main program

```
56 for (int i = 0; i < n; i++) {  
57     position = (int *)calloc(n, sizeof(int));  
58     assert(position != NULL);  
59     position[0] = i;  
60     int error = pthread_create(&threads[i], &attr, goQueen,  
61                               (void *)position);  
62     assert(error == 0);  
63 }  
64 pthread_attr_destroy(&attr);
```

# $n$ Queen with Pthread

## Example 23: (queen-pthread-sum-join.c) Main program

```
66     for (int i = 0; i < n; i++) {  
67         pthread_join(threads[i], NULL);  
68 #ifdef VERBOSE  
69     printf("main: thread %d done\n", i);  
70 #endif  
71     }  
72     printf("total # of solution %d\n", numSolution);  
73     pthread_exit(NULL);  
74     return 0;  
75 }
```

# Join a Pthread

## Example 24: pthread\_join

```
1 int pthread_join(pthread_t thread,  
2                  void **value_ptr);
```

**thread** The thread you want to wait for. This is the variable you used to call pthread\_create.

**value\_ptr** A pointer to a pointer to the return value from the spawned thread. We do not return anything from the spawned thread so we set it to NULL.

# Demonstration

- Run and time the queen-pthread-sum-join.c program.

# Discussion

- Does this program produce correct results?
- Does this program *always* produce correct results?

# A Global Counter

- The previous program uses a recursive function `queen` to compute the number of solutions.
- We will change the program so that `queen` does not return the number of solutions it finds – instead whenever it finds a solution it adds 1 to the global counter `numSolution`.

## Example 25: (queen-pthread-sum-join-race.c) queen

```
19 void queen(int position[], int next)
20 {
21     if (next >= n)
22         numSolution++;
23     else
24         for (int test = 0; test < n; test++)
25             if (ok(position, next, test)) {
26                 position[next] = test;
27                 queen(position, next + 1);
28             }
29 }
```

# Changes

- The return type of `queen` is changed to `void` since it does not return the number of solutions anymore.
- The program structure is simplified since we do not keep track of the number of solutions found.



## Example 26: (queen-pthread-sum-join-race.c) goQueen

```
31 void *goQueen(void *pos)
32 {
33     int *position = (int *)pos;
34     queen(position, 1);
35     pthread_exit(NULL);
36 }
```

# Demonstration

- Run the queen-pthread-sum-join-race.c program.

# Discussion

- Is the answer correct? If not what is the reason?
- Does the previous program queen-pthread-sum-join.c have the same problem? If yes why it did not show up?

# Race

- In fact the race condition exists in queen-pthread-sum-join.c. It does not show up because there are only a few additions to `numSolution`.
- When we increase the number of additions to `numSolution`, it becomes much easier for the race condition to occur.
- The solution is to use the return value of `pthread_exit` to return the number of solutions found in a thread.

# pthread\_exit

## Example 27: pthread\_exit

```
1 void pthread_exit(void *value);
```

## Example 28: pthread\_join

```
1 int pthread_join(pthread_t thread,  
2                  void **value_ptr);
```

## Return value

- We will find a place for the spawned thread to store the number of solutions it finds, and return the address by `pthread_exit` to the main thread.
- The main thread will use the second parameter of `pthread_join` to retrieve the number of solutions a thread found.

# No Global

## Example 29: (queen-pthread-sum-join-correct.c) Headers

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <assert.h>
5
6 #define MAXN 20
7 int n; /* a global n */
```

# No Global

- We remove the global variable `numSolution`, which causes the race condition.



## goQueen

## Example 30: (queen-pthread-sum-join-correct.c) goQueen

```
31 void *goQueen(void *pos)
32 {
33     int *position = (int *)pos;
34     int *num = (int *)malloc(sizeof(int));
35     *num = queen(position, 1);
36 #ifdef VERBOSE
37     printf("goQueen: thread %d, # of solution = %d\n",
38           position[0], *num);
39 #endif
40     pthread_exit(num);
41 }
```

# Heap

- We decide to put the number of solutions a spawned thread found in the heap, with the address stored in `num`.
- Recall that all threads share the heap.
- This address, i.e., the value of `num`, is passed back with `pthread_exit`.

## main

## Example 31: (queen-pthread-sum-join-correct.c) main

```
65  int numSolution = 0;
66  for (int i = 0; i < n; i++) {
67      int *num;
68      pthread_join(threads[i], (void **)&num);
69      numSolution += *num;
70  #ifdef VERBOSE
71      printf("main: thread %d done\n", i);
72  #endif
73  }
74  printf("total # of solution %d\n", numSolution);
75  pthread_exit(NULL);
76  return 0;
77 }
```

## Return Value

- Note that the API requires that we pass the address of a pointer into `pthread_join`.
- The value returned by `pthread_join` is then added into `numSolution`, which is now a local variable.

# Demonstration

- Run and time the queen-pthread-sum-join-correct.c program.

# Implementation

- Implement a parallel program that solves the  $n$  queen problem with pthread. You need to sum up the number of solutions from all threads correctly.
- Measure the speedup and efficiency.

# Discussion

- Does this program produce correct results?
- Compare the timing from queen-pthread-sum-join-correct.c and queen-pthread-sum-join-race.c

# Mutex

- We can prevent the race condition by allowing only thread to access the shared variable `numSolution`.
- Pthread library provide a *mutex* (mutual exclusion) mechanism that allows only one thread to proceed in execution. This effectively provide a way to implement a critical section.



# Steps

- Declare a mutex variable of type `pthread_mutex_t`. This variable must be global because every thread will use it.
- Initialize the mutex with `pthread_mutex_init`.
- Lock the mutex with `pthread_mutex_lock`.
- Unlock the mutex with `pthread_mutex_unlock`.

# Declaration

## Example 32: (queen-pthread-sum-join-race-mutex.c) Declaration

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <assert.h>
5
6 #define MAXN 20
7 int n;                                /* a global n */
8 int numSolution;
9 pthread_mutex_t numSolutionLock;
```

# Mutex Variable

- Declare a global mutex variable `numSolutionLock`.
- We will use this mutex variable to synchronize the access to the global counter `numSolution`.

## main

## Example 33: (queen-pthread-sum-join-race-mutex.c) queen

```
41 int main (int argc, char *argv[])
42 {
43     assert(argc == 2);
44     n = atoi(argv[1]);
45     assert(n <= MAXN);
46     int *position;
47     pthread_t threads[MAXN];
48     pthread_attr_t attr;
49     pthread_attr_init(&attr);
50     pthread_attr_setdetachstate(&attr,
51     PTHREAD_CREATE_JOINABLE);
52     pthread_mutex_init(&numSolutionLock, NULL);
```

# pthread\_mutex\_init

## Example 34: (pthread-mutex-init.c) Initialize a mutex

```
1 int pthread_mutex_init(pthread_mutex_t *mutex,  
2                          const pthread_mutexattr_t *attr);
```

**mutex** The mutex to initialize.

**attr** The attribute of the mutex to set. Set to NULL for the default.

## main

- Initialize numSolutionLock with `pthread_mutex_init`.
- Use NULL for the default mutex attribute value.

# pthread\_mutex\_lock

Example 35: (pthread-mutex-lock.c) Lock/Unlock a mutex

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex);  
2 int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

`mutex` The mutex to lock/unlock.

## queen

Example 36: (queen-pthread-sum-join-race-mutex.c) queen

```
20 void queen(int position[], int next)
21 {
22     if (next >= n) {
23         pthread_mutex_lock(&numSolutionLock);
24         numSolution++;
25         pthread_mutex_unlock(&numSolutionLock);
26     } else
27         for (int test = 0; test < n; test++)
28             if (ok(position, next, test)) {
29                 position[next] = test;
30                 queen(position, next + 1);
31             }
32 }
```



# queen

- Whenever the function `queen` finds a solution, it needs to lock `numSolution` with mutex `numSolutionLock` by calling `pthread_mutex_lock`.
- If it cannot acquire the lock it will have to wait.
- After adding 1 to `numSolution`, `queen` releases the lock by calling `pthread_mutex_unlock`.

## main

## Example 37: (queen-pthread-sum-join-race-mutex.c) queen

```
64  for (int i = 0; i < n; i++) {  
65      pthread_join(threads[i], NULL);  
66  #ifdef VERBOSE  
67      printf("main: thread %d done\n", i);  
68  #endif  
69  }  
70  printf("total # of solution %d\n", numSolution);  
71  pthread_mutex_destroy(&numSolutionLock);  
72  pthread_exit(NULL);  
73  return 0;  
74 }
```

# pthread\_mutex\_destroy

Example 38: (pthread-mutex-destroy.c) pthread\_destroy

```
1 int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

## main

- After all spawned threads finish, we release the mutex by calling `pthread_mutex_destroy`.

# Demonstration

- Run and time the queen-pthread-sum-join-race-mutex.c program.

# Discussion

- Does this program produce correct results?
- Compare the timing from queen-pthread-sum-join-race-mutex.c with that from queen-pthread-sum-join-correct.c.