Thread
Create, Terminate

Thread Creation
The pthread_create function starts a new threads in the calling process.

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

- First argument, thread_id is used to store thread id.
- Second argument, attr is used to customize thread attributes. If attr is NULL, create thread with default attributes
- Third, the thread starts running at the address of start_routine function. This function take a single argument.
- Finally, arg is argument used to pass to start_routine function.

Note:
When two threads are created, no guarantee that which will run first.

Return Value
On success, return 0
On error, return failure number.

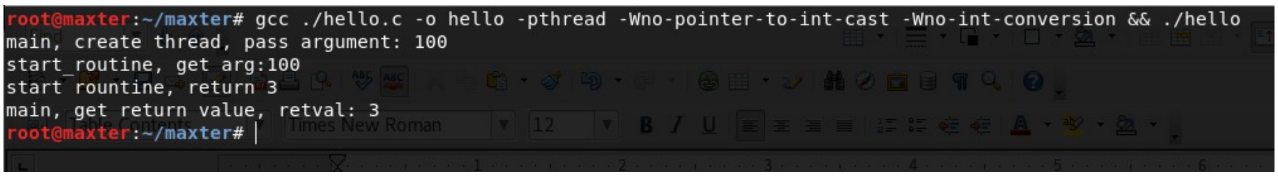
Thread Termination
A thread can exit on below conditions,
- It calls pthread_exit(retval), retval is the exit status value that is available to another thread in the same process that calls pthread_join.
- It returns from start routine function. This is equivalent to calling pthread_exit with the value supplied in the return statement.
- It is canceled. The exit code is set to PTHREAD_CANCELED.
- Its process terminate.

Example
Create thread, pass argument, get return value
```
#include <pthread.h>
#include <stdio.h>

void* start_routine(void* arg) {
    int n = (int)arg;
    printf("start_routine, get arg:%d\n", n);

    printf("start rountine, return 3\n");

    //return value
    pthread_exit((void*)3);
    // return (void*)3;
}

int main() {
    pthread_t thread_id;
    void* retval;//used to get thread return value

    printf("main, create thread, pass argument: 100\n");

    pthread_create(&thread_id, NULL, start_routine, 100);
    pthread_join(thread_id, &retval);

    int n = (int)retval;
    printf("main, get return value, retval: %d\n", retval);

    return 0;
}
```

Compile

```
gcc ./hello.c -o hello -pthread -Wno-pointer-to-int-cast -Wno-int-conversion
```

Result



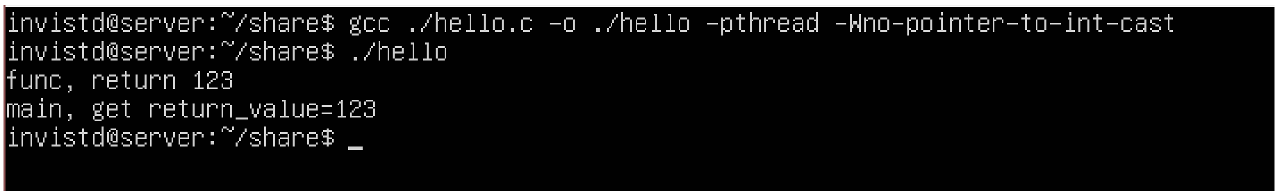
```
root@maxter:~/maxter# gcc ./hello.c -o hello -pthread -Wno-pointer-to-int-cast -Wno-int-conversion && ./hello
main, create thread, pass argument: 100
start_routine, get arg:100
start rountine, return 3
main, get return value, retval: 3
root@maxter:~/maxter# |
```

Example

Fetch thread return value (start routine function exit on return statement)

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

void* func(void* arg) {
    printf("func, return 123\n");
    return (void*)123;
}

int main(){
    pthread_t thread_1;
    void* return_value;

    pthread_create(&thread_1, NULL, func, NULL);
    pthread_join(thread_1, (void*)&return_value);

    printf("main, get return_value=%d\n", (int)return_value);

    return 1;
}
```

Result

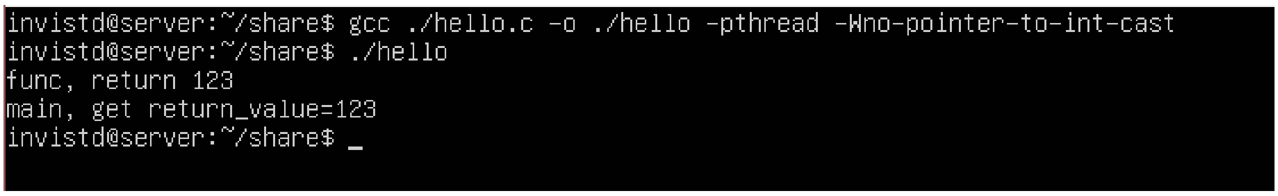


```
invistd@server:~/share$ gcc ./hello.c -o ./hello -pthread -Wno-pointer-to-int-cast
invistd@server:~/share$ ./hello
func, return 123
main, get return_value=123
invistd@server:~/share$ _
```

Example
Fetch thread return value ((start routine function exit on pthread_exit)

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

void* func(void* arg) {
    printf("func, return 123\n");
    // return (void*)123;
    pthread_exit((void*)123);
}

int main(){
    pthread_t thread_1;
    void* return_value;

    pthread_create(&thread_1, NULL, func, NULL);
    pthread_join(thread_1, (void*)&return_value);

    printf("main, get return_value=%d\n", (int)return_value);

    return 1;
}
```

Result



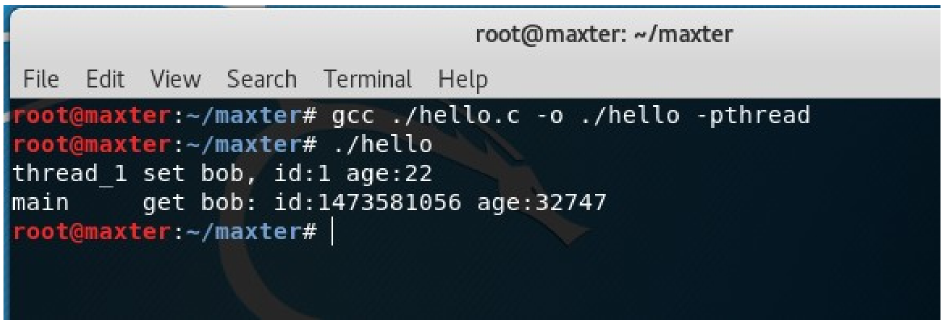
```
invistd@server:~/share$ gcc ./hello.c -o ./hello -pthread -Wno-pointer-to-int-cast
invistd@server:~/share$ ./hello
func, return 123
main, get return_value=123
invistd@server:~/share$ _
```

Ensure that memory of exit status is still valid after thread is terminated
When a thread exits by calling pthread_exit or by simply returning from the start routine, the exit status can be obtained by another thread by calling pthread_join.
If the memory of exit status is allocated on thread stack. It will released when thread stack is destroyed. Then, if another thread fetch the exit status, it will get an invalid memory.
So, be careful that the memory of exit status is still valid after thread terminated.

Example

```
//bad code
#include <pthread.h>
#include <stdio.h>

struct employee{
    int id;
    int age;
};

void* func(void* arg) {
    struct employee bob = {1, 22};
    printf("thread_1 set bob, id:%d age:%d\n", bob.id, bob.age);

    pthread_exit((void*)&bob);//bob will be destroyed after thread_1
terminated
}
```

```
int main(){
    pthread_t thread_1;
    struct employee *bob;

    pthread_create(&thread_1, NULL, func, NULL);
    pthread_join(thread_1, (void*)&bob);

    //main thread fetch bob. But bob memory just released
    printf("main     get bob: id:%d age:%d\n", bob->id, bob->age);

    return 0;
}
```



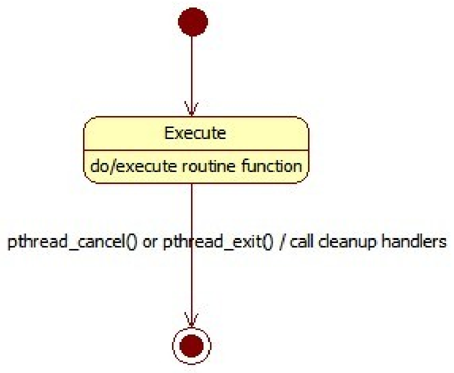
```
root@maxter:~/maxter# gcc ./hello.c -o ./hello -pthread
root@maxter:~/maxter# ./hello
thread_1 set bob, id:1 age:22
main     get bob: id:1473581056 age:32747
root@maxter:~/maxter# 
```

Ensure normally terminating thread does not return value matches PTHEAD_CANCEL
PTHREAD_CANCEL is the value returned when a thread is canceled.
If start routine function return PTHREAD_CANCEL, another thread that waiting for it, will confuse that it is canceled.

Thread Cleanup Handler
Cleanup-handler is a function that automatically executed when a thread is canceled or terminated. It is used to prevent any trouble caused by the suddenly cancellation, such as: memory leak, mutex can not release, deadlock.
pthread_cleanup_push pushes routine onto the top of the cleanup handler stack. When routine is invoked later, it will be given arg as its argument.

```
void pthread_cleanup_push(void (*routine)(void *), void *arg)
```

A cleanup handler is popped from the stack and executed in the following circumstances:
- Thread is canceled by calling pthread_cancel, all cleanup handlers in stack are popped and executed.
- Thread is exited by calling pthread_exit, all cleanup handlers in stack are popped and executed.
- Thread calls pthread_cleanup_pop with a nonzero argument, the top-most cleanup handler is popped and executed.


Execute
do/execute routine function
pthread_cancel() or pthread_exit() / call cleanup handlers

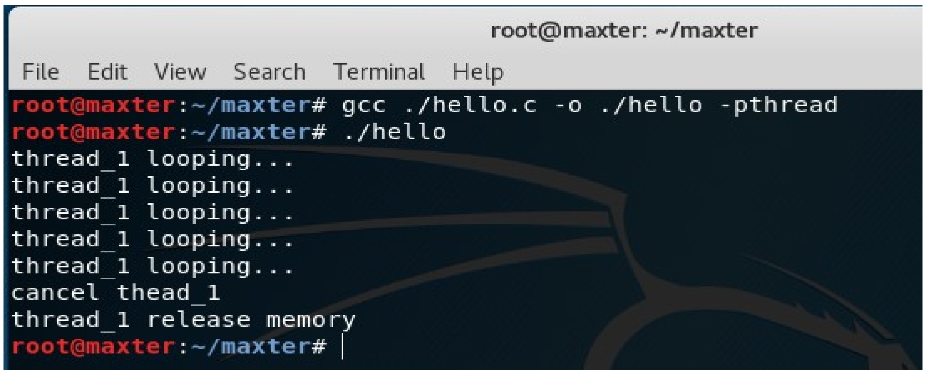
```
void pthread_cleanup_pop(int arg)
```

if arg is non-zero, pthread_cleanup_pop function pop and execute the top-most cleanup handler in the stack.

Example

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

pthread_t thread_1;

void release_memory(void* arg){
    printf("thread_1 release memory\n");
    free(arg);
}

void* func_1(void* arg){
    char* buff = (char*)malloc(16);
    strcpy(buff, "looping...");

    pthread_cleanup_push(release_memory, buff);

    while(1){
        printf("thread_1 %s\n", buff);
        sleep(1);
    }

    free(buff);

    pthread_cleanup_pop(0);
}

int main(){
    pthread_create(&thread_1, NULL, func_1, NULL);

    sleep(5);
    printf("cancel thead_1\n");
    pthread_cancel(thread_1);

    pthread_join(thread_1, NULL);
    return 0;
}
```

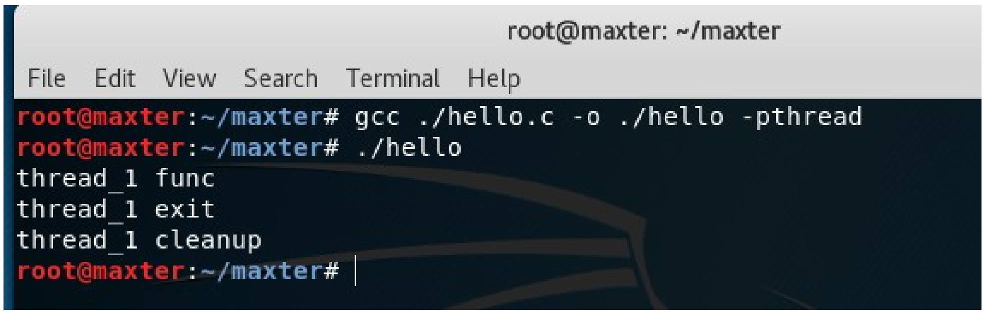
In above program,
- thread_1 has a infinite loop. It print “thread_1 looping...” each loop.
- Push release_memory function to cleanup handlers
- Main thread starts thread_1 and cancel it after 5 seconds.
- thread_1 will call release_memory before it is canceled.



```
root@maxter:~/maxter# gcc ./hello.c -o ./hello -pthread
root@maxter:~/maxter# ./hello
thread_1 looping...
thread_1 looping...
thread_1 looping...
thread_1 looping...
thread_1 looping...
cancel thead_1
thread_1 release memory
root@maxter:~/maxter# 
```

Example

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

void cleanup(void* arg) {
    printf("thread_1 cleanup\n");
}

void* func(void* arg) {
    printf("thread_1 func\n");

    pthread_cleanup_push(cleanup, NULL);

    printf("thread_1 exit\n");

    pthread_cleanup_pop(0);//put this function to avoid compile error
}

int main(){
    pthread_t thread_1;
    pthread_create(&thread_1, NULL, func, NULL);
    pthread_join(thread_1, NULL);

    return 0;
}
    pthread_exit((void*)0);
```



```
root@maxter:~/maxter# gcc ./hello.c -o ./hello -pthread
root@maxter:~/maxter# ./hello
thread_1 func
thread_1 exit
thread_1 cleanup
root@maxter:~/maxter# 
```

Exit start routine function by return statement vs pthread_exit
Cleanup-handler is execute if a thead terminated by pthread_exit. This does not happen if thread terminate by performing return statement in start routine function.

	Cleanup-handlers is executed
Exit by pthread_exit	**YES**
Exit by performing return statement	**NO**