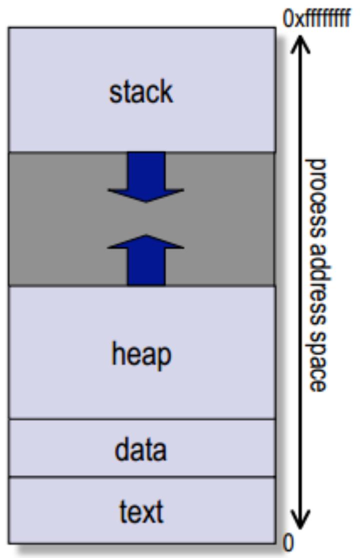
CS214-system programming

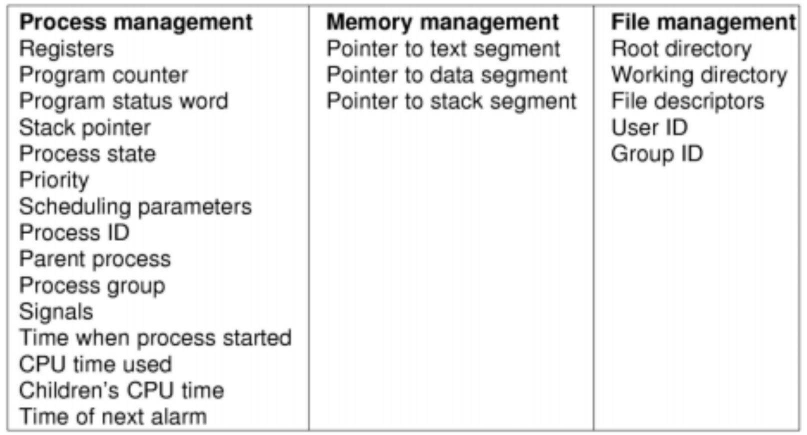
- Section 03/08 recitation 07

Yunhe Gao
yg397@scarletmail.rutgers.edu

Review: Processes

- A process is an instance of a computer program that is being executed, it contains the program code and its current activity
- Each process has a **Process Control Block (PCB)** to describe its current execution state and some resources it is now processing
  - Identification information (e.g. parent process, user, process ID)
  - Execution contexts such as separate threads of execution
  - Address space (virtual memory)
  - I/O state (file handles, network communication endpoints, etc.)
  - etc.
- PCBs are stored as entries in a process table



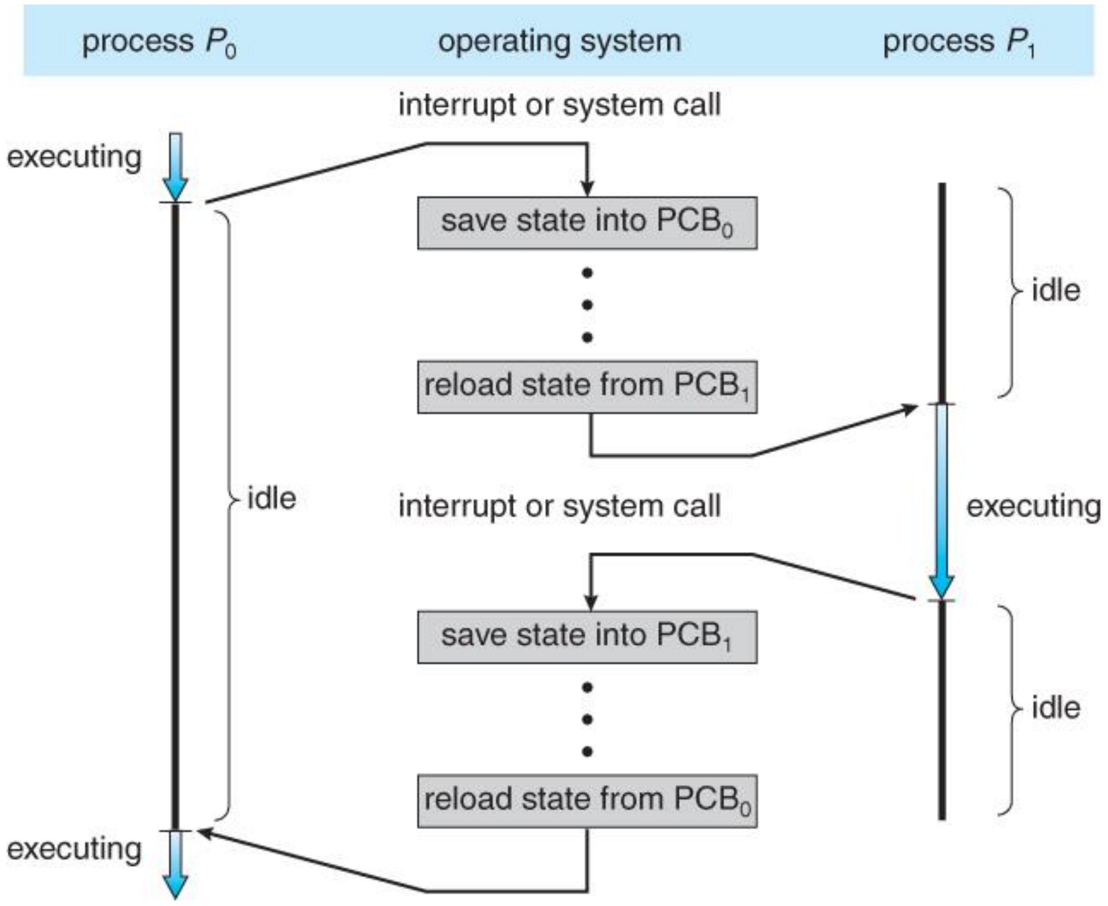


Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Fields of a Process Control Block

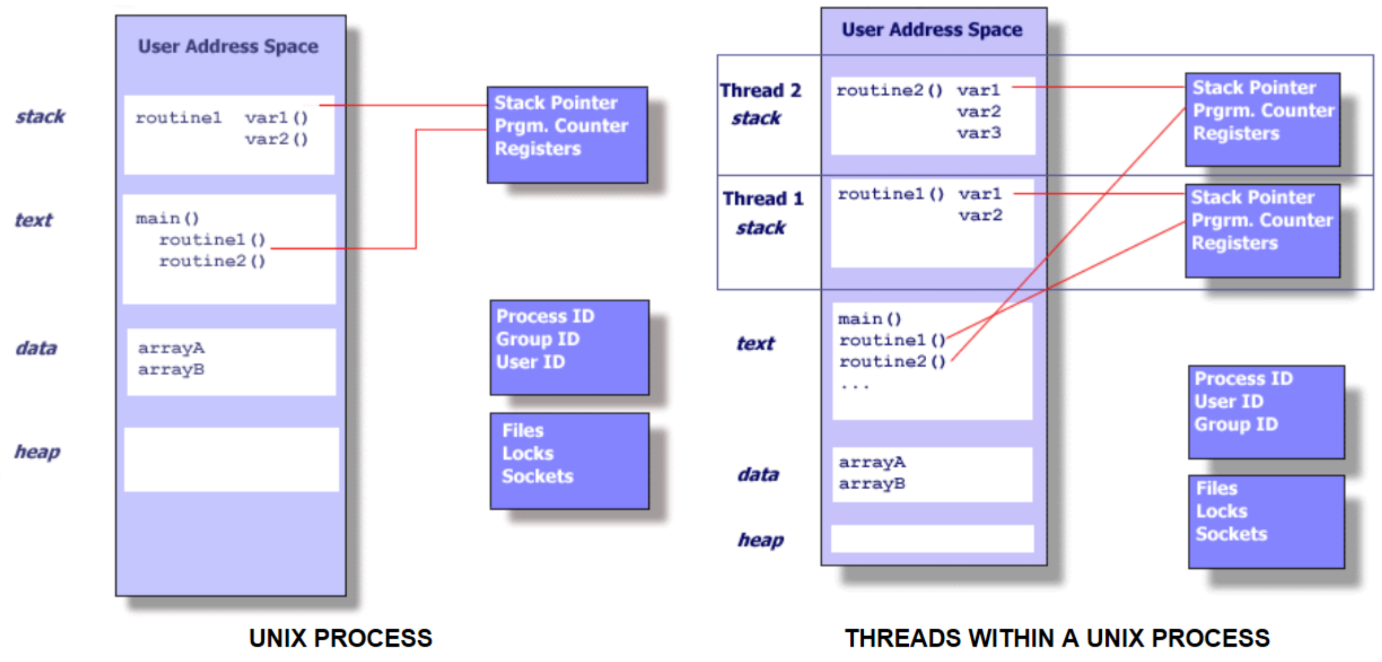
Context switch of processes

- The kernel implements multitasking through context switch
- Few CPU resources vs. a bunch of processes: CPU usage is splitted in form of time slices to all processes
- Context of a process: the state (PCB) of a process that the kernel needs to resume a preempted/interrupted process.
- Each process is scheduled to use CPU resources within the allocated time piece, after the time slice expires, the process stops running and its state would be stored inside PCB for next running.



Thread

- A thread is a flow of execution through the process code
- As a process is created, it has at least one running thread
  - Each process can have multiple threads
- A thread can be regarded as a light-weighted process: most of the overhead has already been accomplished through the creation of the corresponding process
- Each has its own program counter, system registers and a stack
  - local variables aren't shared between threads
- User address space and process control block is shared among all the threads for the same process
  - Threads of the same processes share code, global variables and heap
  - Memory allocated from the heap or global variables can be used for sharing data between threads


User Address Space
stack
text
data
heap
routine1 var1 () var2 ()
main () routine1 () routine2 ()
arrayA arrayB
Stack Pointer Prgm. Counter Registers
Process ID Group ID User ID
Files Locks Sockets
UNIX PROCESS
User Address Space
Thread 2 stack
Thread 1 stack
routine2 () var1 var2 var3
routine1 () var1 var2
main () routine1 () routine2 () ...
arrayA arrayB
Stack Pointer Prgrm. Counter Registers
Process ID User ID Group ID
Files Locks Sockets
THREADS WITHIN A UNIX PROCESS

An example of thread

- The dispatcher, reads incoming request for work from the network. After examining the request, it chooses an idle (i.e. blocked) or creates a worker thread and hands it the request.
- The dispatcher then wakes up the sleeping worker, moving it from blocked state to ready state, or create a new one.
- The worker checks to see if the request can be satisfied from the Web page cache. If not, it starts a read operation to get the page from the disk and blocks until the disk operation completes.
- When the worker is working, the dispatcher can listen for other incoming requests.

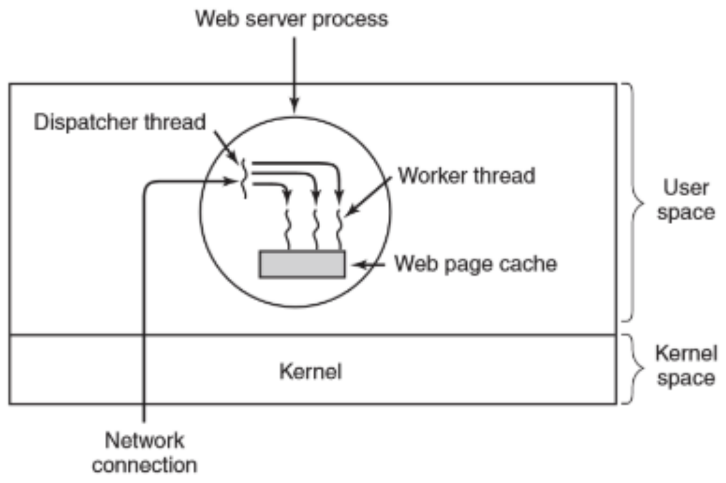
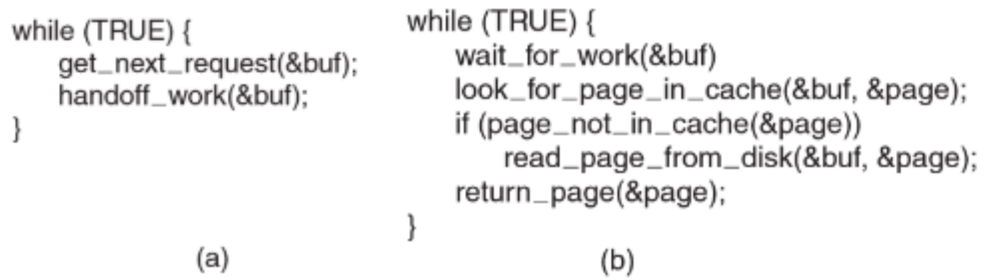


Figure 2-8. A multithreaded Web server.



```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread. (b) Worker thread.

Why threads

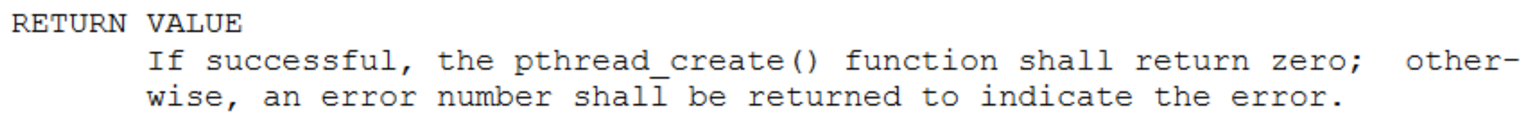
- By decomposing an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.
- The ability to communicate and share data among themselves.
- Since threads are more light-weighted than processes, they are easier (i.e., faster) to create and destroy than processes.
- Threads are useful on systems with multiple CPUs, where real parallelism is possible.

Two concrete examples where a thread is needed

- Blocking I/O. Without threads, blocking I/O halts the whole process. In a multithreaded process, individual threads may block, waiting on I/O, while other threads make forward progress.
- Memory savings. Threads provide an efficient way to share memory yet utilize multiple units of execution. In this manner they are an alternative to multiple processes.

pthread_create

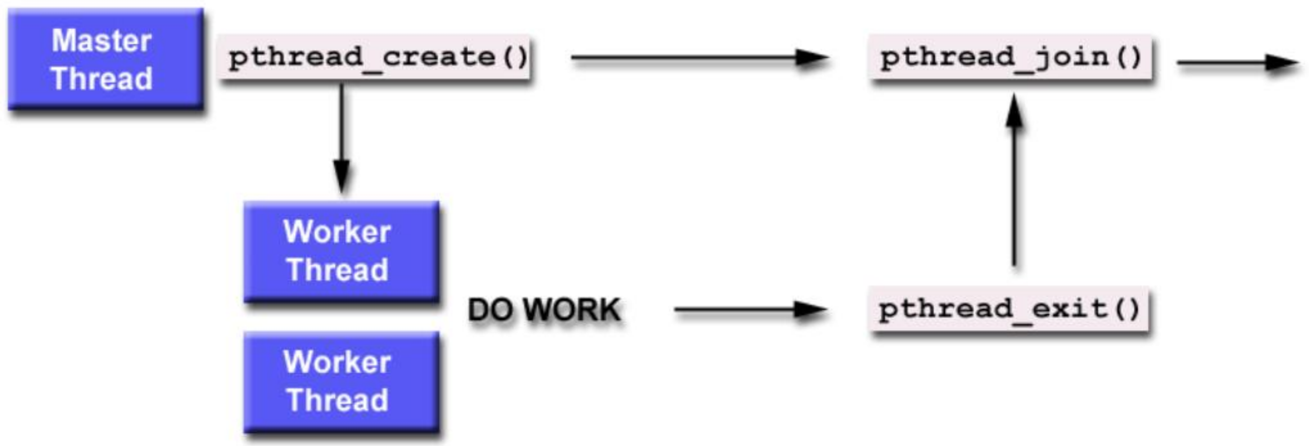
```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
              void *(*start_routine) (void *), void *arg);
```



```
RETURN VALUE
       If successful, the pthread_create() function shall return zero;  other-
       wise, an error number shall be returned to indicate the error.
```

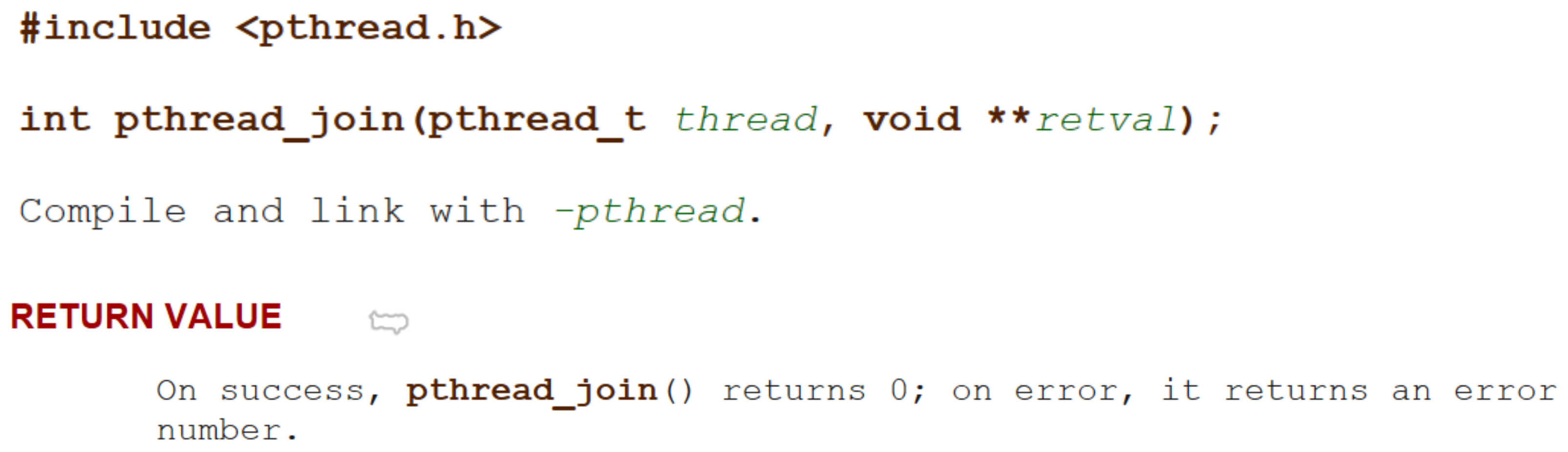
- thread: stores the ID of the new thread in the buffer pointed by thread. This identifier is used to refer to the thread in subsequent calls to other pthread functions.
- attr: an opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
- start_routine: the C routine that the thread will execute once it is created.
  - function to execute in the thread
- arg: a single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.
- Upon successful completion, pthread_create() shall store the ID of the created thread in the location referenced by thread.

Thread Joining



- The pthread_join() subroutine blocks the calling thread until the thread with the specific thread_id terminates.
- The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().
- A joining thread can match only one pthread_join() call. It is a logical error to attempt multiple joins on the same thread.
- Attribute: only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

pthread_join



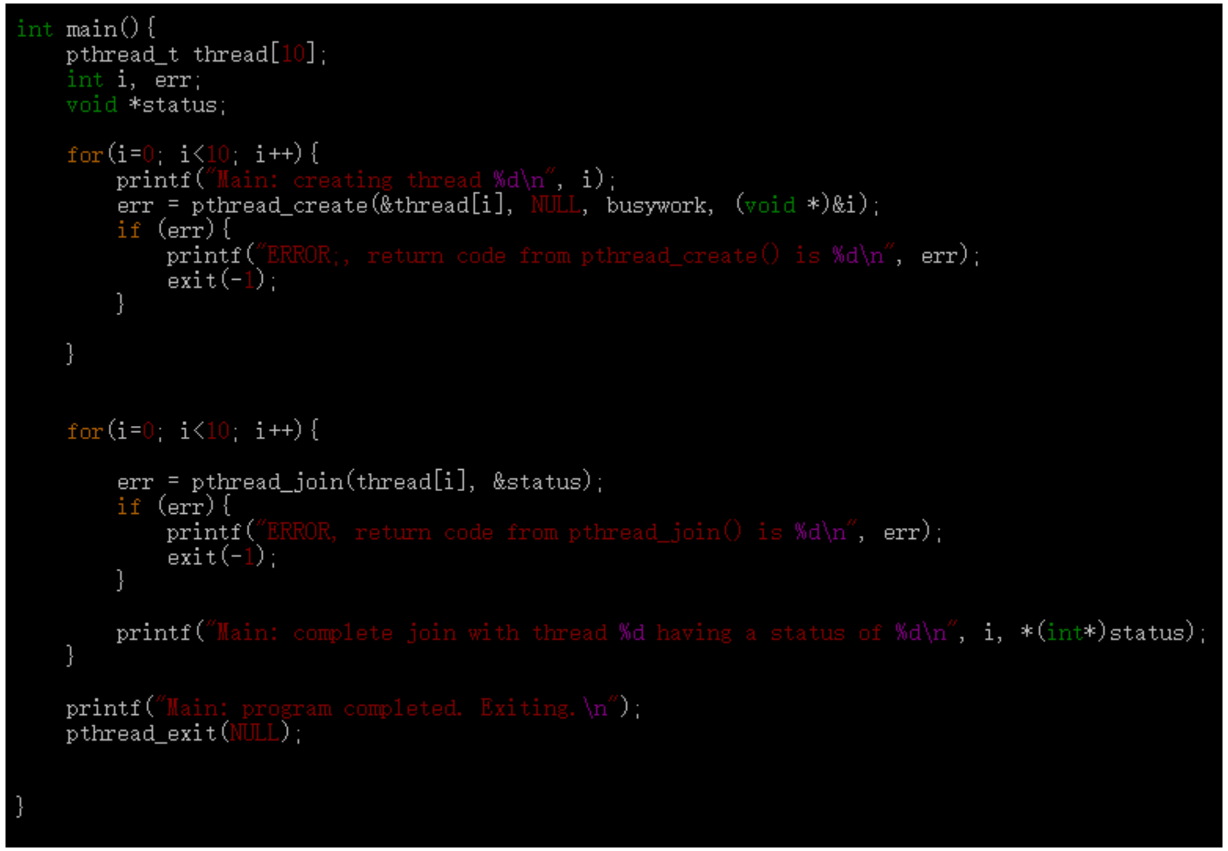
```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);

Compile and link with -pthread.
```

RETURN VALUE

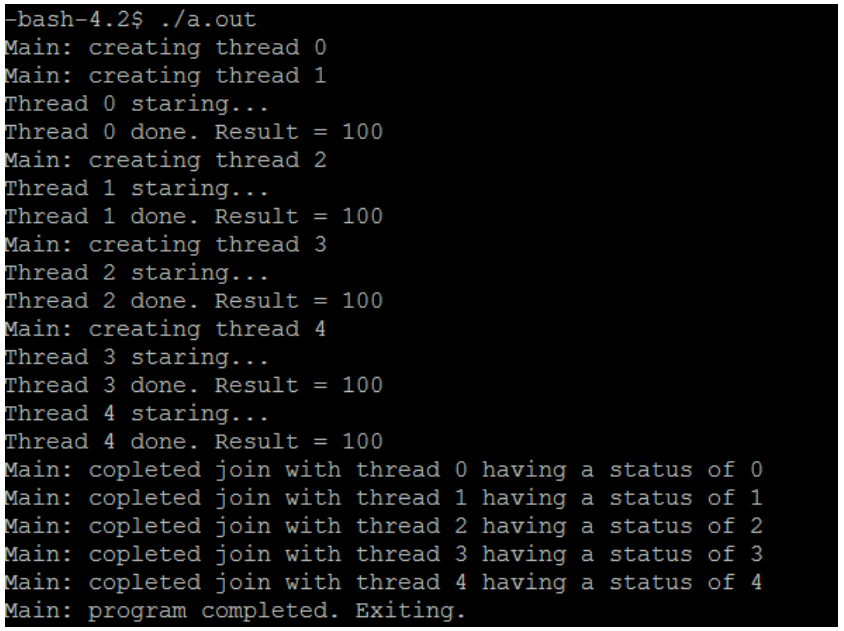
```
On success, pthread_join() returns 0; on error, it returns an error
number.
```

- The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.
- If retval is not NULL, then pthread_join() copies the exit status of the target thread into the location pointed to by retval. If the target thread was canceled, then THREAD_CANCELED is placed in the location pointed to by retval.
- If multiple threads simultaneously try to join with the same thread, the results are undefined.

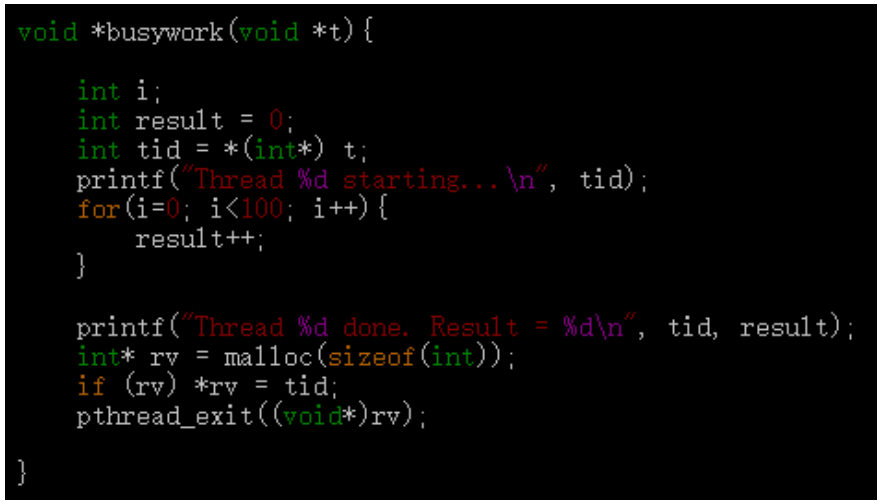
Example



```c
int main(){
    pthread_t thread[10];
    int i, err;
    void *status;

    for(i=0; i<10; i++){
        printf("Main: creating thread %d\n", i);
        err = pthread_create(&thread[i], NULL, busywork, (void *)&i);
        if (err){
            printf("ERROR;, return code from pthread_create() is %d\n", err);
            exit(-1);
        }

    }


    for(i=0; i<10; i++){

        err = pthread_join(thread[i], &status);
        if (err){
            printf("ERROR, return code from pthread_join() is %d\n", err);
            exit(-1);
        }

        printf("Main: complete join with thread %d having a status of %d\n", i, *(int*)status);
    }

    printf("Main: program completed. Exiting.\n");
    pthread_exit(NULL);


}
```

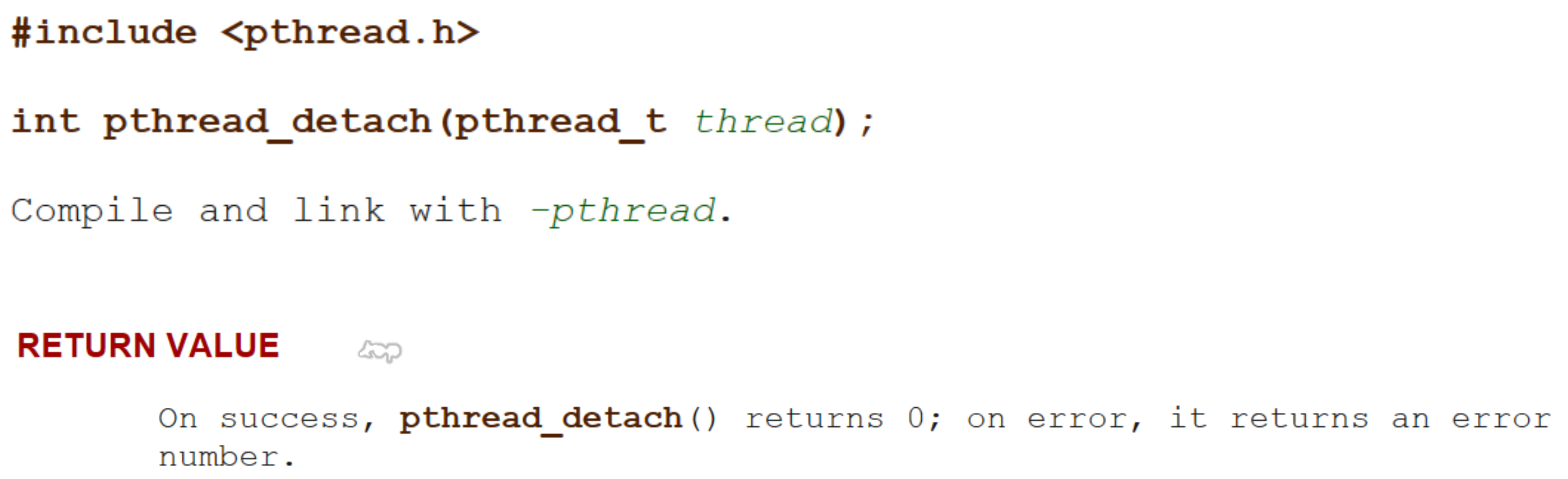


```
-bash-4.2$ ./a.out
Main: creating thread 0
Main: creating thread 1
Thread 0 staring...
Thread 0 done. Result = 100
Main: creating thread 2
Thread 1 staring...
Thread 1 done. Result = 100
Main: creating thread 3
Thread 2 staring...
Thread 2 done. Result = 100
Main: creating thread 4
Thread 3 staring...
Thread 3 done. Result = 100
Thread 4 staring...
Thread 4 done. Result = 100
Main: copleted join with thread 0 having a status of 0
Main: copleted join with thread 1 having a status of 1
Main: copleted join with thread 2 having a status of 2
Main: copleted join with thread 3 having a status of 3
Main: copleted join with thread 4 having a status of 4
Main: program completed. Exiting.
```



```c
void *busywork(void *t){

    int i;
    int result = 0;
    int tid = *(int*) t;
    printf("Thread %d starting...\n", tid);
    for(i=0; i<100; i++){
        result++;
    }

    printf("Thread %d done. Result = %d\n", tid, result);
    int* rv = malloc(sizeof(int));
    if (rv) *rv = tid;
    pthread_exit((void*)rv);

}
```

pthread_detach



```
#include <pthread.h>

int pthread_detach(pthread_t thread);

Compile and link with -pthread.
```

RETURN VALUE

```
On success, pthread_detach() returns 0; on error, it returns an error
number.
```

- The pthread_detach() function marks the thread identified by thread as detached.
- When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.
- Attempting to detach an already detached thread results in unspecified behavior.

Thanks