CSCI315 – Operating Systems Design

Department of Computer Science

Bucknell University

Introduction to Thread

Ch 4.1 – 4.3

This set of notes is based on notes from the textbook authors, as well as L. Felipe Perrone, Joshua Stough, and other instructors.
Xiannong Meng, Fall 2021.

A Different Model for Process Communication

- We discussed IPC with message passing, e.g., pipes
  - We will learn another form of message passing, sockets, later
- In message passing, the communicating processes are running in different context (as in different processes), thus passing information is slower;
- In shared memory, IPC is faster. However, we need to set up the shared memory.
  - We didn’t quite discuss this topic
- In this segment, we explore a different model for processes to communicate with shared memory, that is, using *threads.*

What Is A Thread?

A *thread* is a light-weight process.

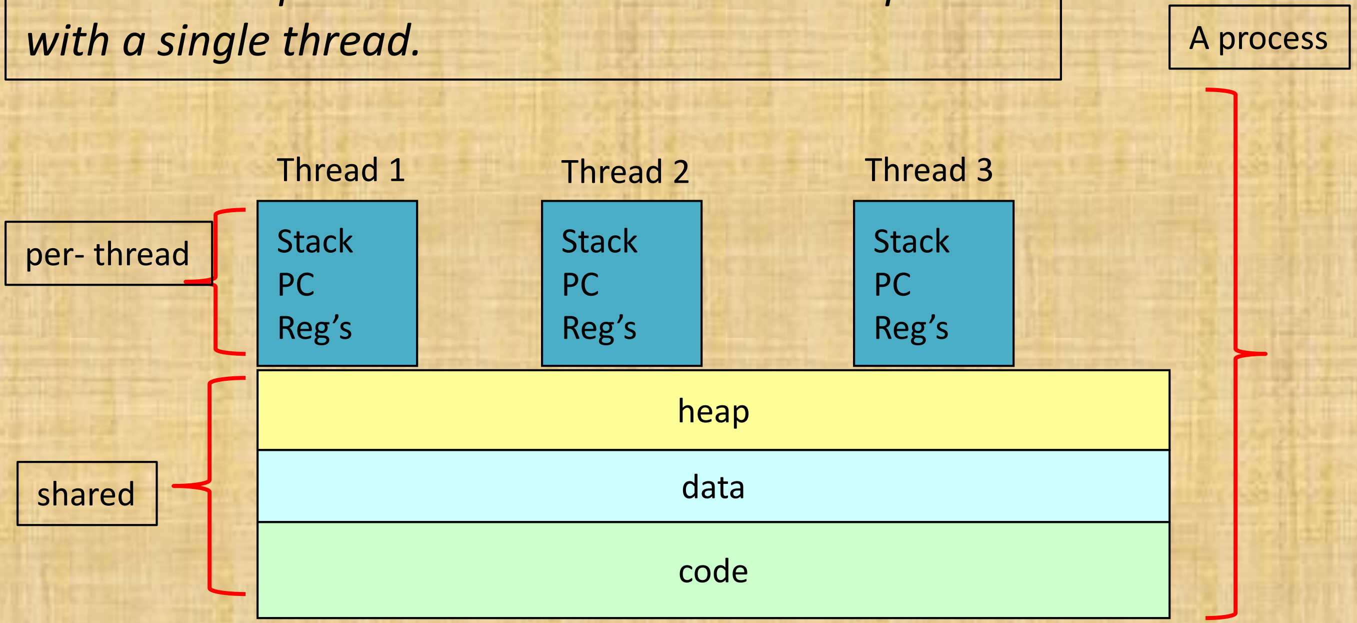
- Shared code
- Shared data
- Shared heap
- Independent PC
- Independent registers
- Independent stack

In memory, just like processes

Compared to: a *process* is a program in execution. Each process has its independent code, data, heap, PC, registers, and stack.

Process and Thread

Example: *A process that contains three threads.*
A traditional process can be considered as a process with a single thread.



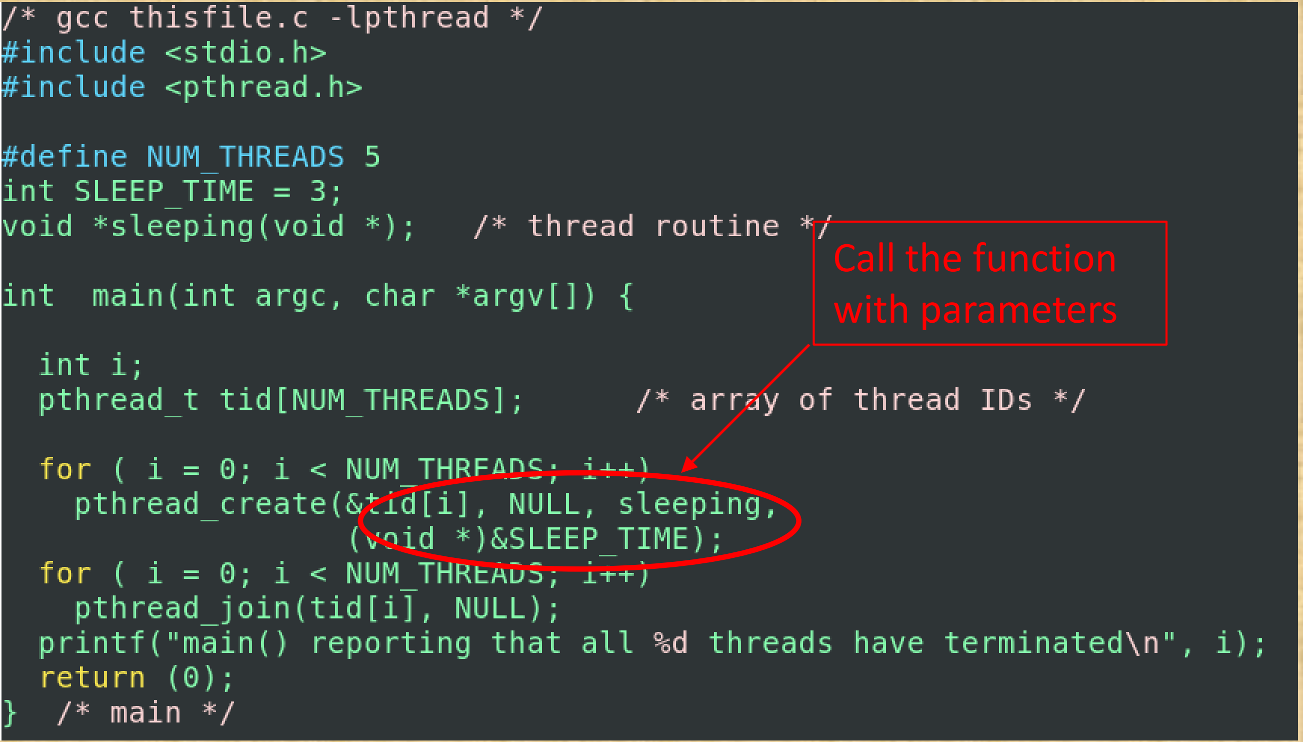
Why Threads?

- Consider the following two examples, operations can take place in parallel, we did them in CSCI 206
  - Matrix addition: A = B + C
  - Selection sort
- Advantages of using threads
  - **Responsiveness:** multiple threads can be executed in parallel, reducing the completion time needed for a problem
  - **Resource sharing:** multiple threads have access to the same data, sharing made easier
  - **Economy:** creating process (allocating memory and other resources) is costly. For the same number of execution units, threads are less expensive
  - **Scalability:** thread model can be easily scaled up

POSIX Threads

- While threads can be implemented in many different ways, the POSIX thread is a popular and effective implementation of threads on UNIX-like system
- POSIX: Potable Operating Systems Interface

A Simple, Complete Thread Example

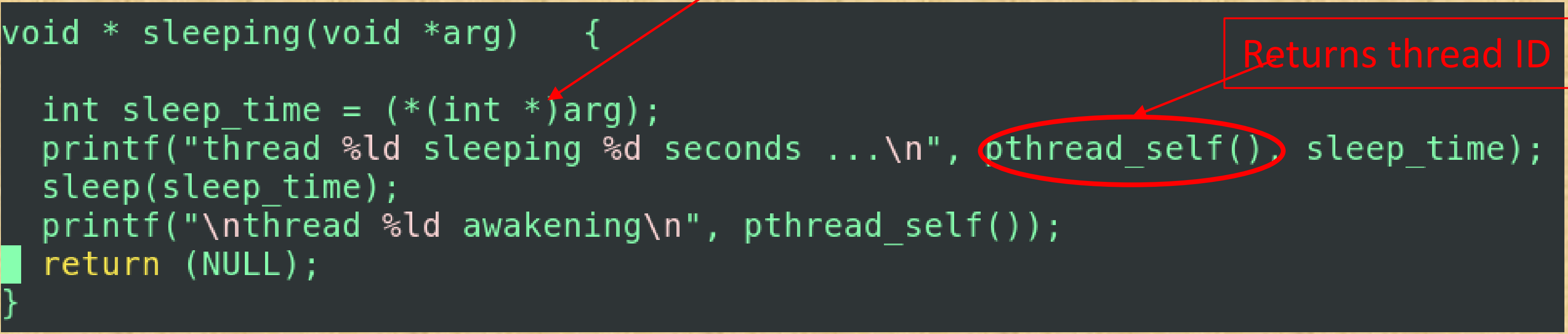


```c
/* gcc thisfile.c -lpthread */
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 5
int SLEEP_TIME = 3;
void *sleeping(void *);   /* thread routine */

int  main(int argc, char *argv[]) {

  int i;
  pthread_t tid[NUM_THREADS];      /* array of thread IDs */

  for ( i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], NULL, sleeping,
                   (void *)&SLEEP_TIME);
  for ( i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
  printf("main() reporting that all %d threads have terminated\n", i);
  return (0);
}  /* main */
```

Call the function with parameters

http://www.eg.bucknell.edu/~cs315/F2021/meng/code/thread/trd-sleep.c

The Thread Work: *sleeping()*



```
void * sleeping(void *arg)   {

  int sleep_time = (*(int *)arg);
  printf("thread %ld sleeping %d seconds ...\n", pthread_self(), sleep_time);
  sleep(sleep_time);
  printf("\nthread %ld awakening\n", pthread_self());
  return (NULL);
}
```

Cast param to proper type

Returns thread ID

Compile and Execute the Program

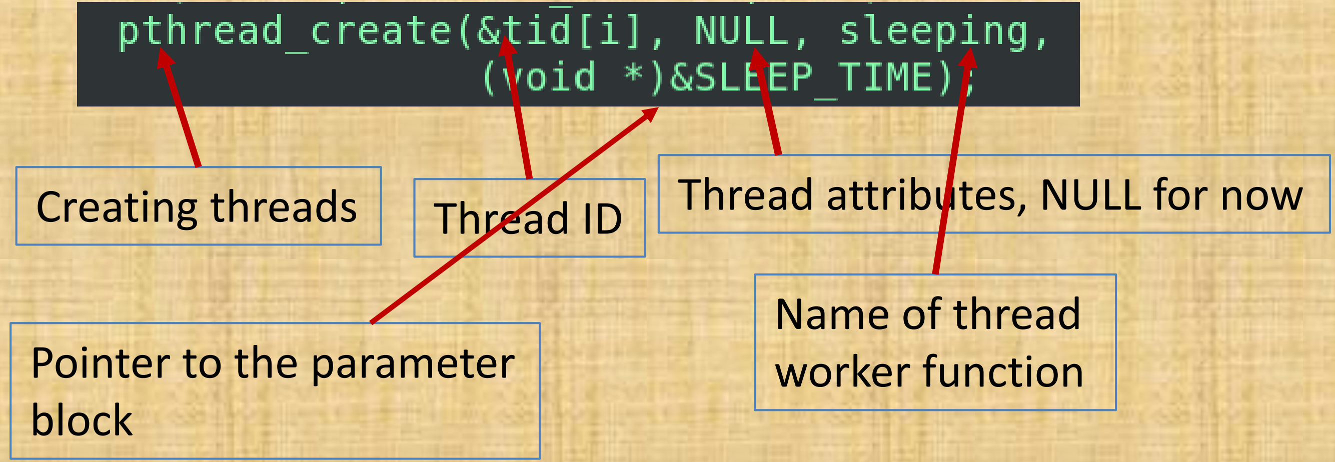
```
[xmeng@linuxremote]$ gcc -o thread-sleep trd-sleep.c –lpthread
[xmeng@linuxremote]$ ./thread-sleep
thread 140550497642240 sleeping 3 seconds ...
thread 140550518621952 sleeping 3 seconds ...
thread 140550508132096 sleeping 3 seconds ...
thread 140550476662528 sleeping 3 seconds ...
thread 140550487152384 sleeping 3 seconds ...
thread 140550497642240 awakening
thread 140550518621952 awakening
thread 140550508132096 awakening
thread 140550487152384 awakening
thread 140550476662528 awakening
main() reporting that all 5 threads have terminated
[xmeng@linuxremote]$
```

http://www.eg.bucknell.edu/~cs315/F2021/meng/code/thread/trd-sleep.c

Creating Threads

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

Including the pthread library headers



```
pthread_create(&tid[i], NULL, sleeping,
               (void *)&SLEEP_TIME);
```

Creating threads

Thread ID

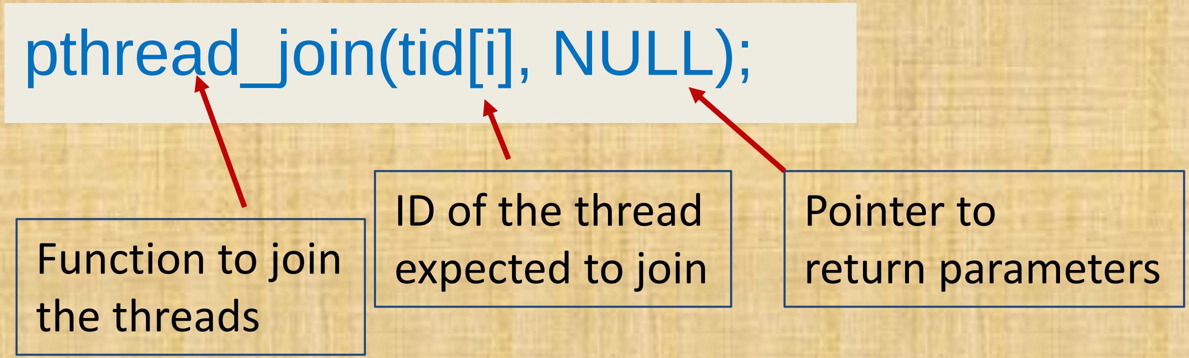
Thread attributes, NULL for now

Pointer to the parameter block

Name of thread worker function

As soon as threads are created, they start to execute the *worker* function

Joining Threads When Finishing



```
pthread_join(tid[i], NULL);
```

Function to join the threads

ID of the thread expected to join

Pointer to return parameters

The second parameter is of the type void **ptr, which is an address to a pointer (pointer to a pointer). If it is used, usually it returns the exit status of the thread.

Think about the “join” here vs “wait” in process.

Review of necessary C knowledge, pointers, function parameters, and others

Pointer Recap

```
NAME
       wait, waitpid, waitid - wait for process to change state

SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *wstatus);

       pid_t waitpid(pid_t pid, int *wstatus, int options);
```

In the parameter list, `int *wstatus`, the variable `wstatus` is a pointer to an integer variable. A **pointer** in C is basically a memory address. To access the content of the, the pointer has to be dereferenced, the following are valid.

```
int k = *wstatus; // value at mem address wstaus assigned to k
wstatus = &k;     // wstatus takes the mem address of k
```

Pointer Recap

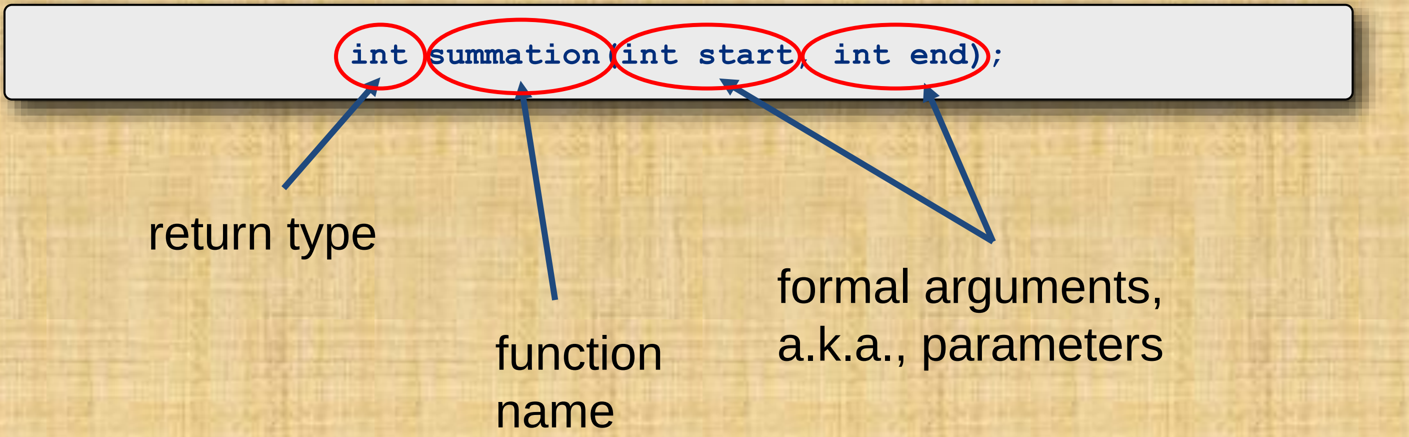
```
int ret_val;
int *status;
.
.
.
ret_val = wait(status);
.
.
.
```

```
int ret_val;
int status;
.
.
.
ret_val = wait(&status);
.
.
.
```

- Do both options compile correctly?
- Do both options run correctly?
- Can you explain what each one does?

Function Recap

Function Prototype



```
int summation(int start, int end);
```

return type

function name

formal arguments, a.k.a., parameters

Function As Parameter(s)

Function Prototypes

```
int add(int a, int b); // a + b
```

```
int sub(int a, int b); // a - b
```

Function Declaration

```
int f(int, int);
```

Function as Parameter(s)

Function prototype

```
int compute(int, int, int g(int, int));
```

Function body that uses function parameter(s)

```
int compute(int a, int b, int g(int, int))
{
   return g(a, b);
}
```

```
int x = compute(3, 4, add);  // 3 + 4 => 7
int y = compute(3, 4, sub); // 3 – 4 => -1
```