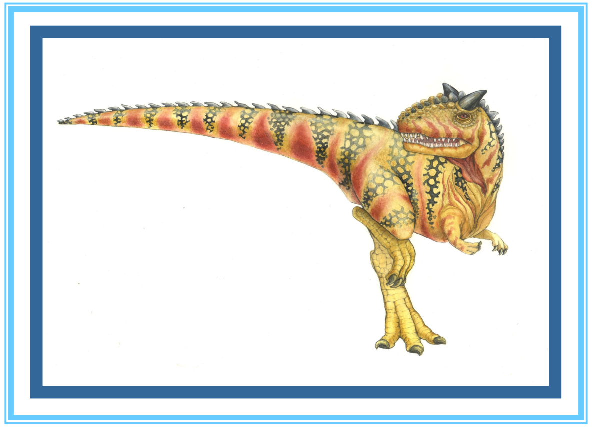
Chapter 4: Multithreaded Programming



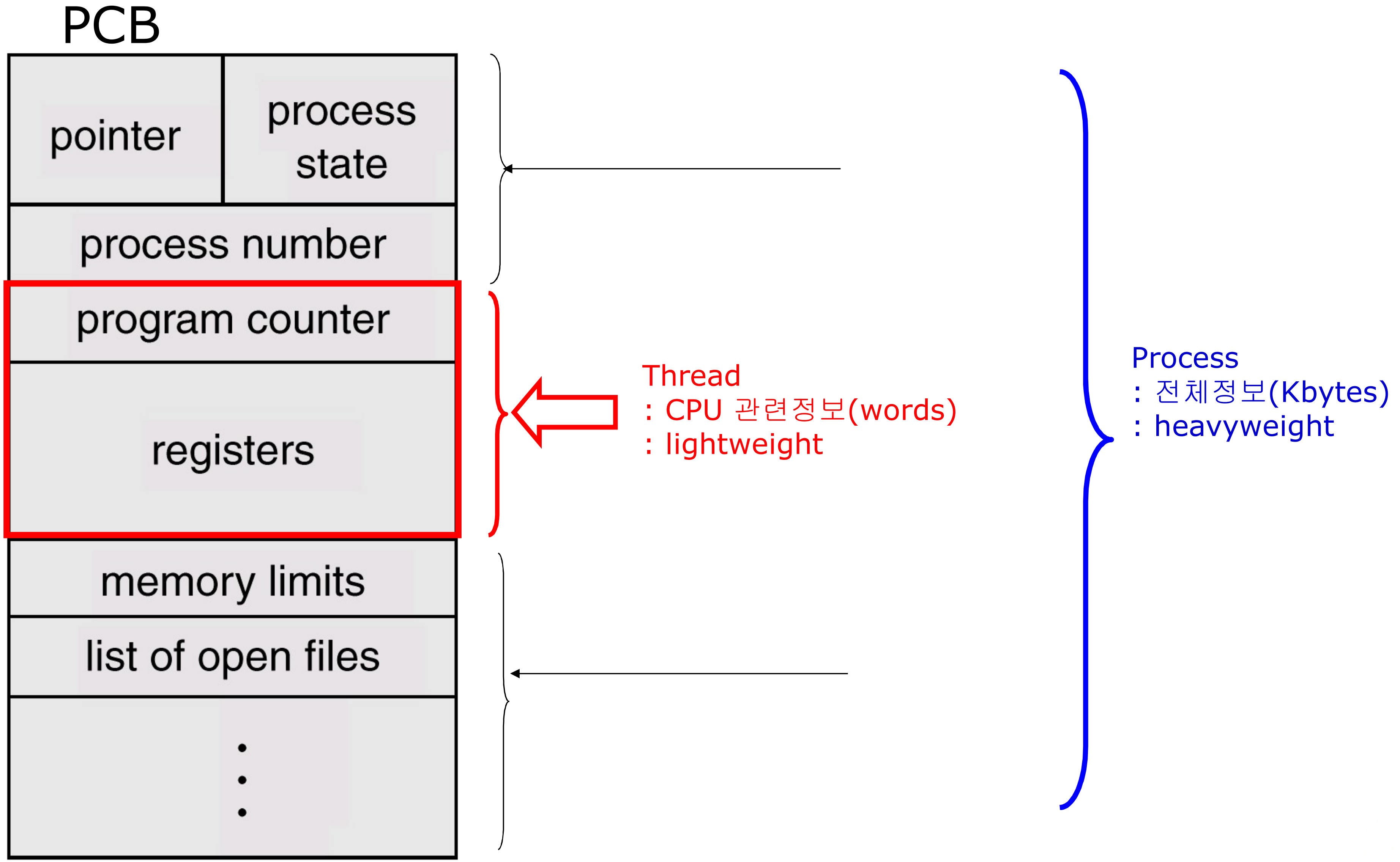
Objectives

- Identify the basic components of a thread, and contrast **threads and processes**
  - **Thread**— **a fundamental unit of CPU utilization** that forms the basis of multithreaded computer systems
- To discuss the APIs for the **Pthreads, Windows**, and **Java thread libraries**

Threads

- So far, process has a single thread of execution
  - single **program counters(PC or IP)**
- Consider having multiple **program counters(PC or IP)** per **process**
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB

Thread Overview



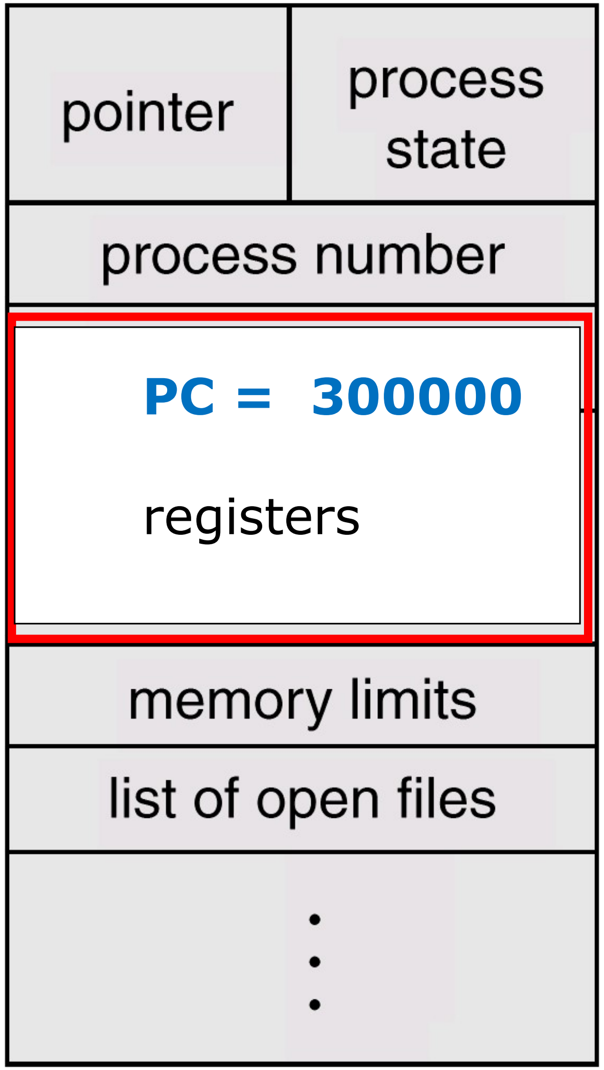
Thread
: CPU 관련정보(words)
: lightweight

Process
: 전체정보(Kbytes)
: heavyweight

One Process, One Thread

PCB

(Stored in main memory)



pointer

process state

process number

PC = 300000

registers

memory limits

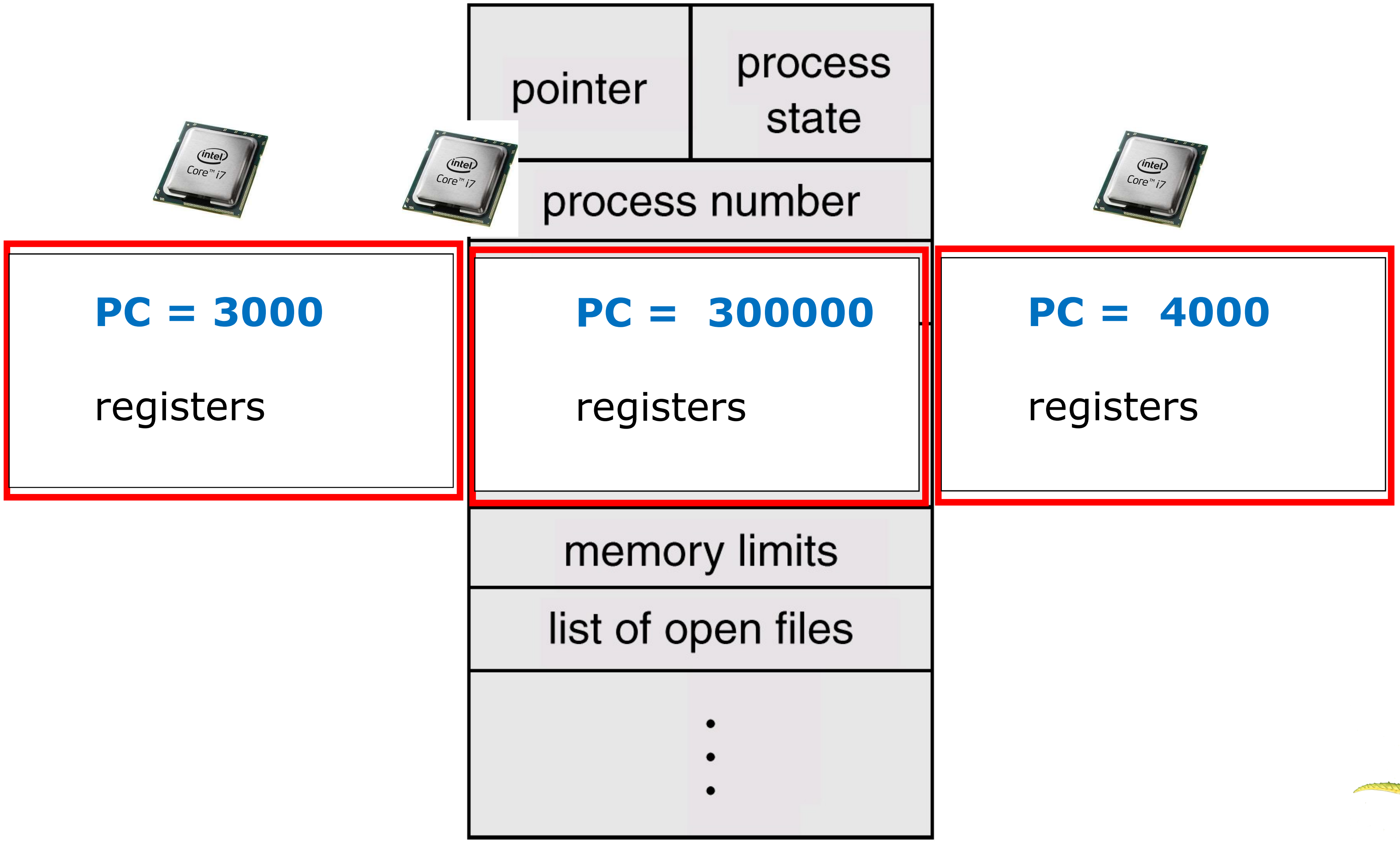
list of open files

•
•
•

One Process, Three Threads

PCB

(Stored in main memory)



pointer

process state

process number

PC = 3000

registers

PC = 300000

registers

PC = 4000

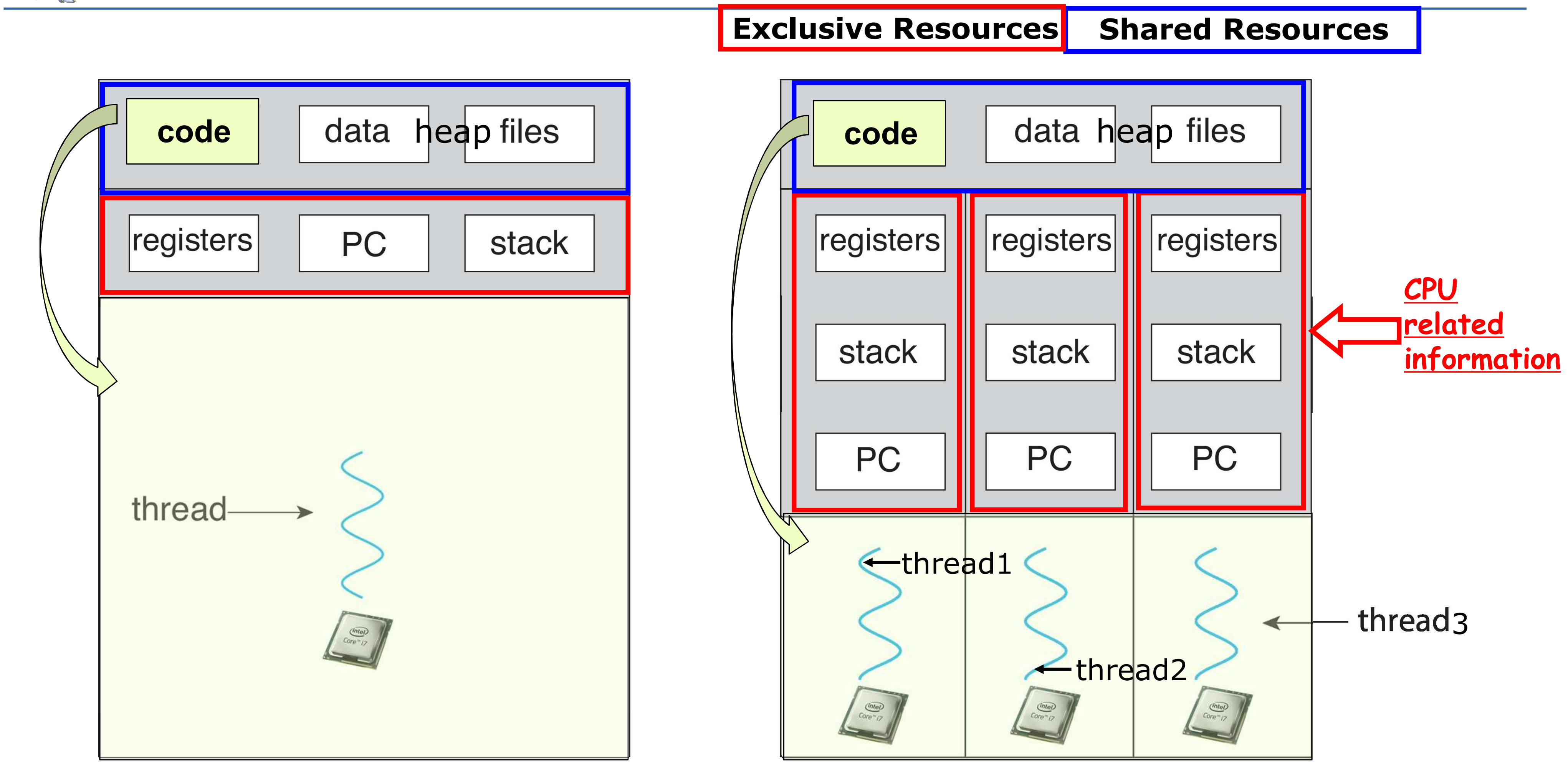
registers

memory limits

list of open files

•
•
•

Single and Multithreaded Processes



Exclusive Resources

Shared Resources

code

data

heap

files

registers

PC

stack

thread

registers

stack

PC

thread1

thread2

thread3

CPU related information

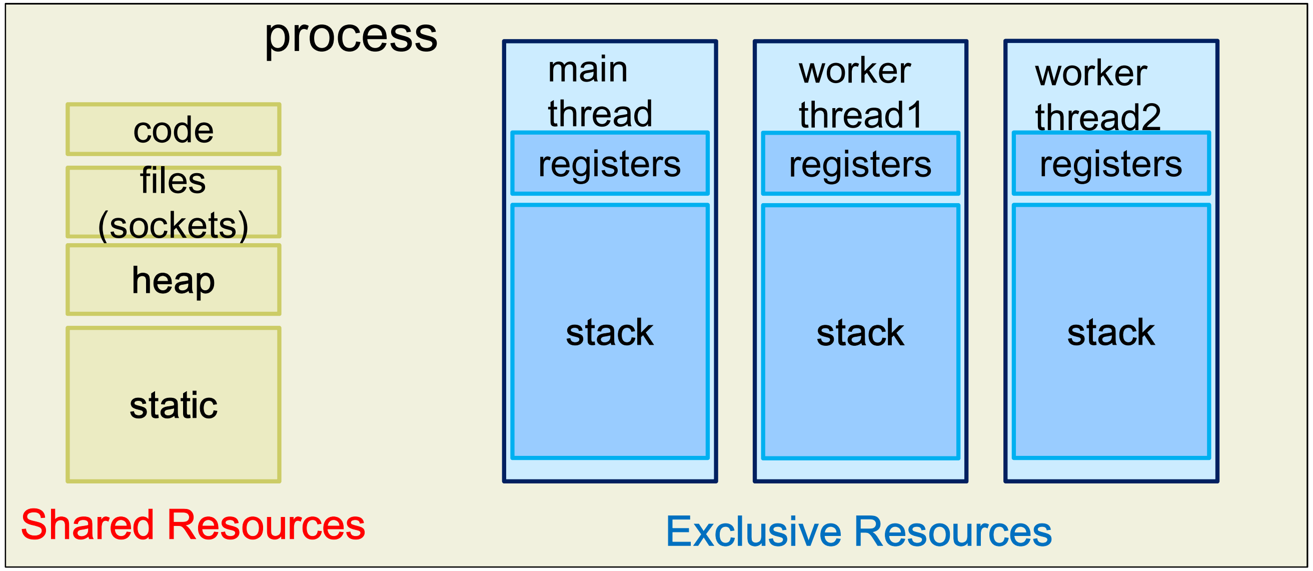
single-threaded process

Memory

multithreaded process

Main thread vs. Worker threads(Secondary threads)

Shared Resources vs. Exclusive Resources



- **Shared Resource among threads**
  - **Code(or Text)** : binary program code
  - **Open Files(Sockets)**
  - **Heap : dynamically allocated memory** during run time
  - **Static(or Data)** : **global variables**, non-local variables, **static local variables**
- **Exclusive Resources of each thread**
  - **Registers in CPU**
  - Stack containing temporary data
    - **automatic local variables, function parameters**, return addresses, return value

Scope and Storage Class of an Identifier

- **Scope(or visibility) of an identifier**
  - Defines **the part of the program** where **you can refer to it**
  - Can be limited to
    - A single block or A single function : **local**
    - The functions in a given file : **non-local**
    - The whole program : **global**
  - It is good programming practice to make identifiers ***as local as possible***

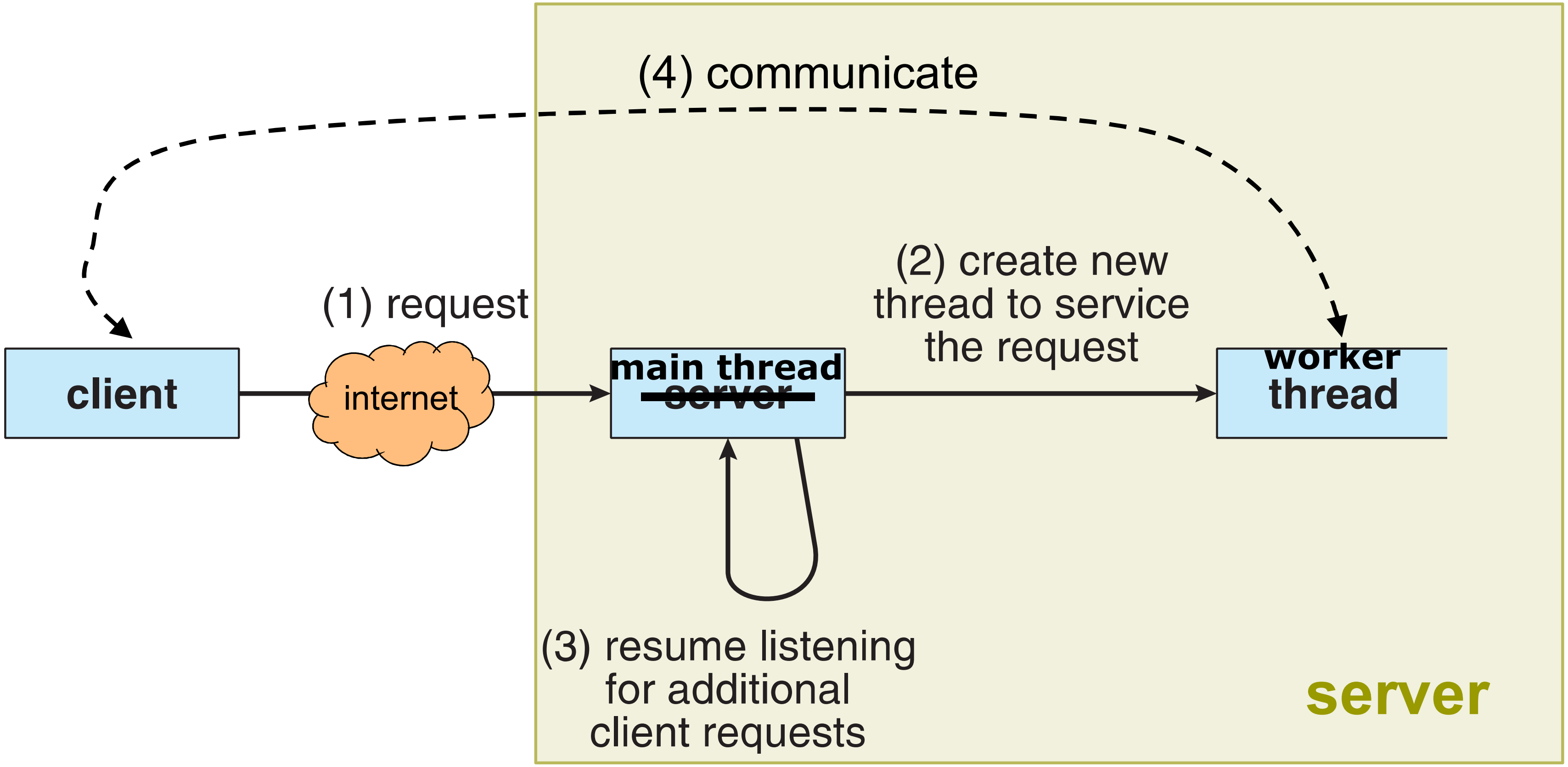
Scope and Storage Class of an Identifier

- **Storage Class**
  - **The location where the variable will be stored. It determines the life-time of variables.**
  - **Automatic** variable : the class of variable that is **stored on the runtime stack**. **Default** storage class
  - **Register** variable : To store the variable **in CPU registers** **rather memory location**, if possible, for quick access. It just give the compiler a **hint** that it should allocate date to a register
  - **Static** variables are **persistent**: they **exist for full duration of the program execution**; they are **not destroyed on completion of their block**; they used for both global and local variable.
  - **Extern**
    - used to give a reference of a global variable that is visible to ALL the program files
    - most commonly used when there are **two or more files sharing the same global variables**
    - used to **declare a global variable in another file**

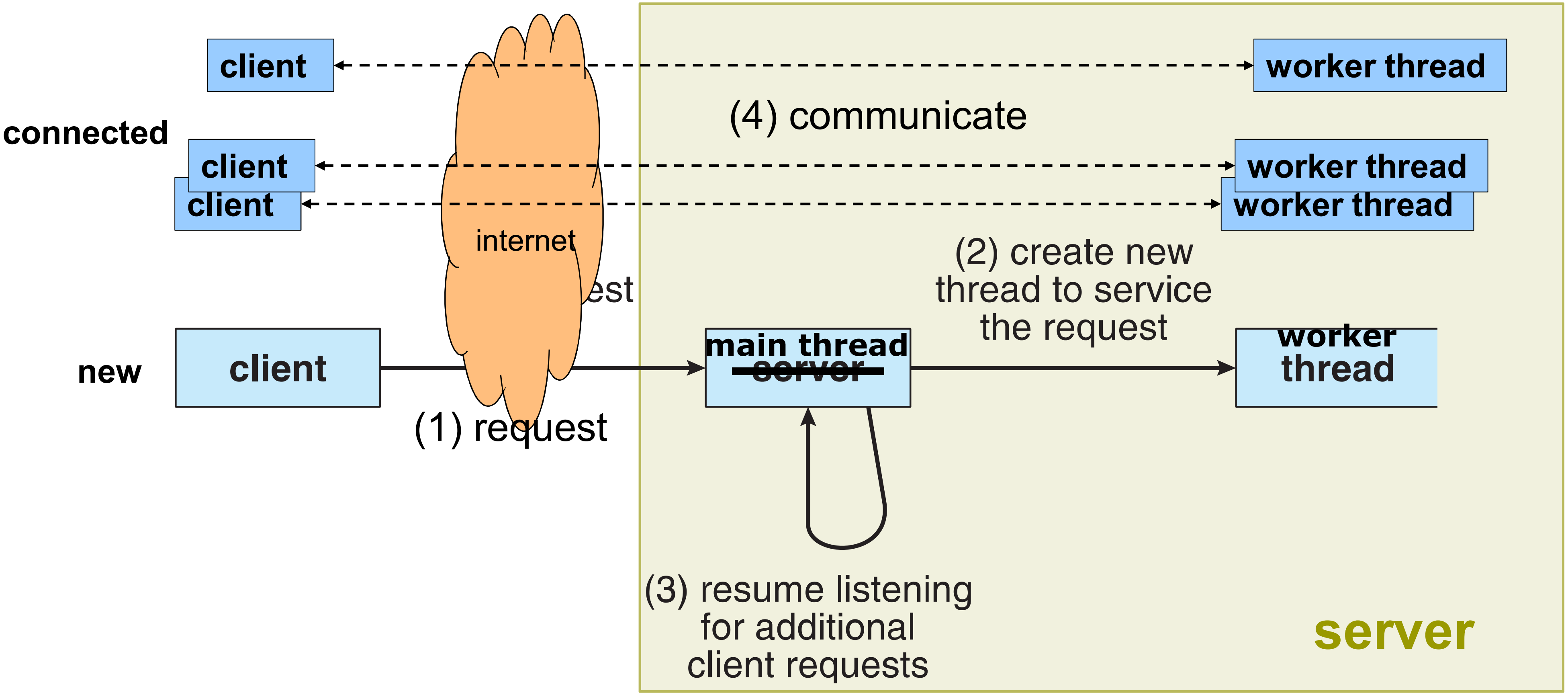
Motivation

- **Most modern applications are multithreaded**
- **Threads run within application**
- **Multiple tasks** with the application can be **implemented by separate threads**
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is **heavy-weight** while thread creation is **light-weight**
- Can simplify code, **increase efficiency**
- Kernels are generally multithreaded

Multithreaded Server Architecture


(4) communicate
(1) request
(2) create new thread to service the request
client
internet
main thread
~~server~~
worker thread
(3) resume listening for additional client requests
server

Multithreaded Server Architecture



Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
  - eg) multi-threaded Web - if one thread is blocked (eg network)
another thread continues (eg display)
- **Resource Sharing –** threads share resources of process, **easier than shared memory or message passing between processes**
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, **challenges** include:
  - **Dividing activities** – examining applications to find areas that can be divided into separate, concurrent tasks
  - **Balance** – programmer also ensure that the tasks perform equal work of equal value
  - **Data splitting** – the data accessed and manipulated by the tasks must be divided to run on separate cores
  - **Data dependency** – the data accessed by the tasks must be examined for dependencies between two or more tasks
  - **Testing and debugging** – when a program is running in parallel on multiple cores, many execution paths are possible

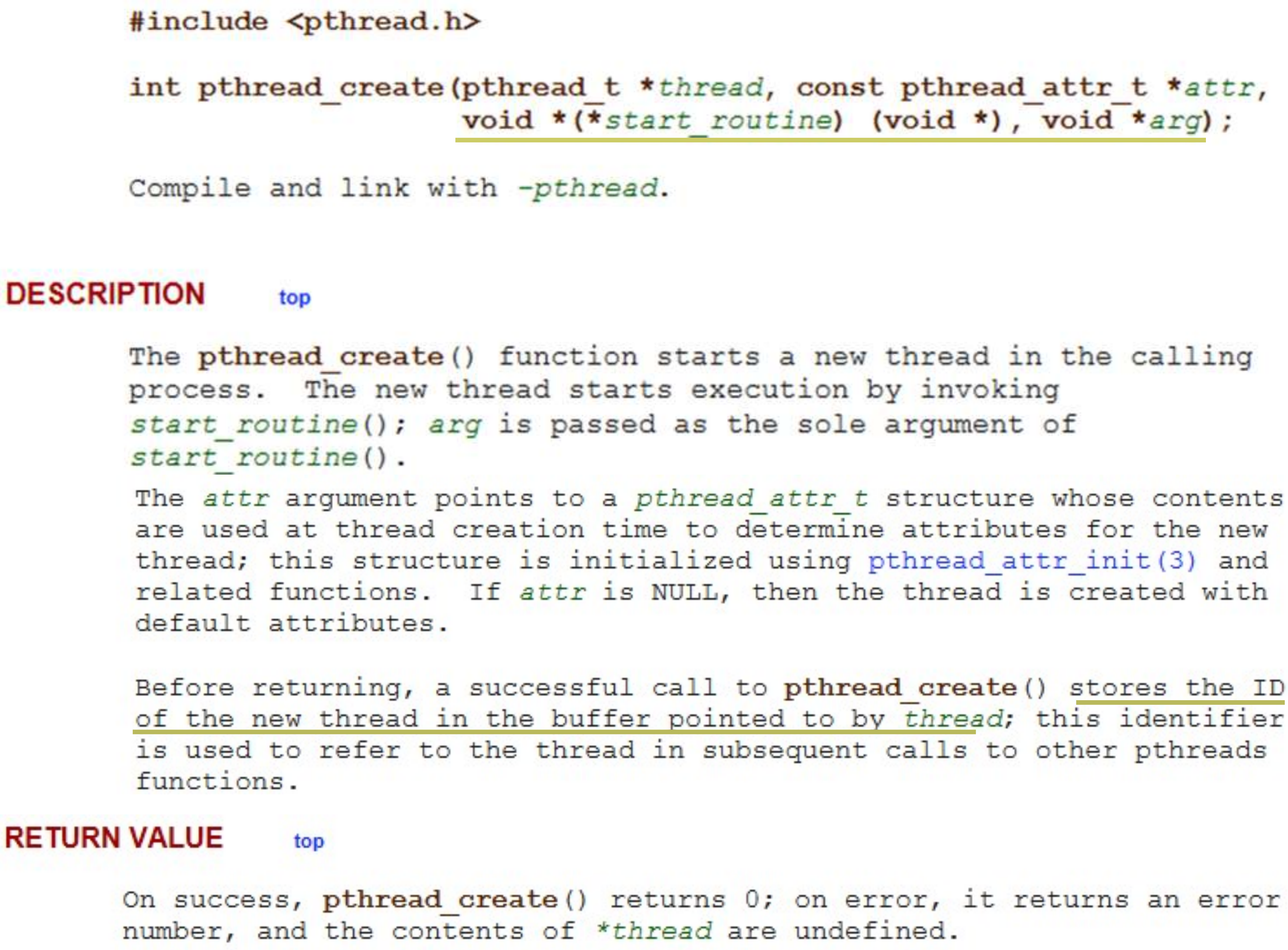
Thread Libraries

- **Thread library** provides programmer with **API for creating and managing threads**
- Two primary ways of **implementing a thread library**
  - Library entirely **in user space** with **no kernel support**
  - Kernel-level library **supported by the OS**
- Three primary thread libraries:
  - **POSIX Pthreads** – **user-level** or **kernel-level** library
  - **Windows threads** – **kernel-level** library
  - **Java threads**
    - Java thread API allows threads to be created and managed directly in Java program
    - JVM is running on top of a host OS
    - Java thread API is implemented **using a thread library available on the host system**

Pthreads

- May be provided either as **user-level** or **kernel-level**
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

- The C program on the next slides
  - *Calculates the **summation of a non-negative integer in a separate worker thread***



```
#include <pthread.h>

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

Compile and link with -pthread.
```

DESCRIPTION top

The **pthread_create**() function starts a new thread in the calling process. The new thread starts execution by invoking *start_routine*(); *arg* is passed as the sole argument of *start_routine*().

The *attr* argument points to a *pthread_attr_t* structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using pthread_attr_init(3) and related functions. If *attr* is NULL, then the thread is created with default attributes.

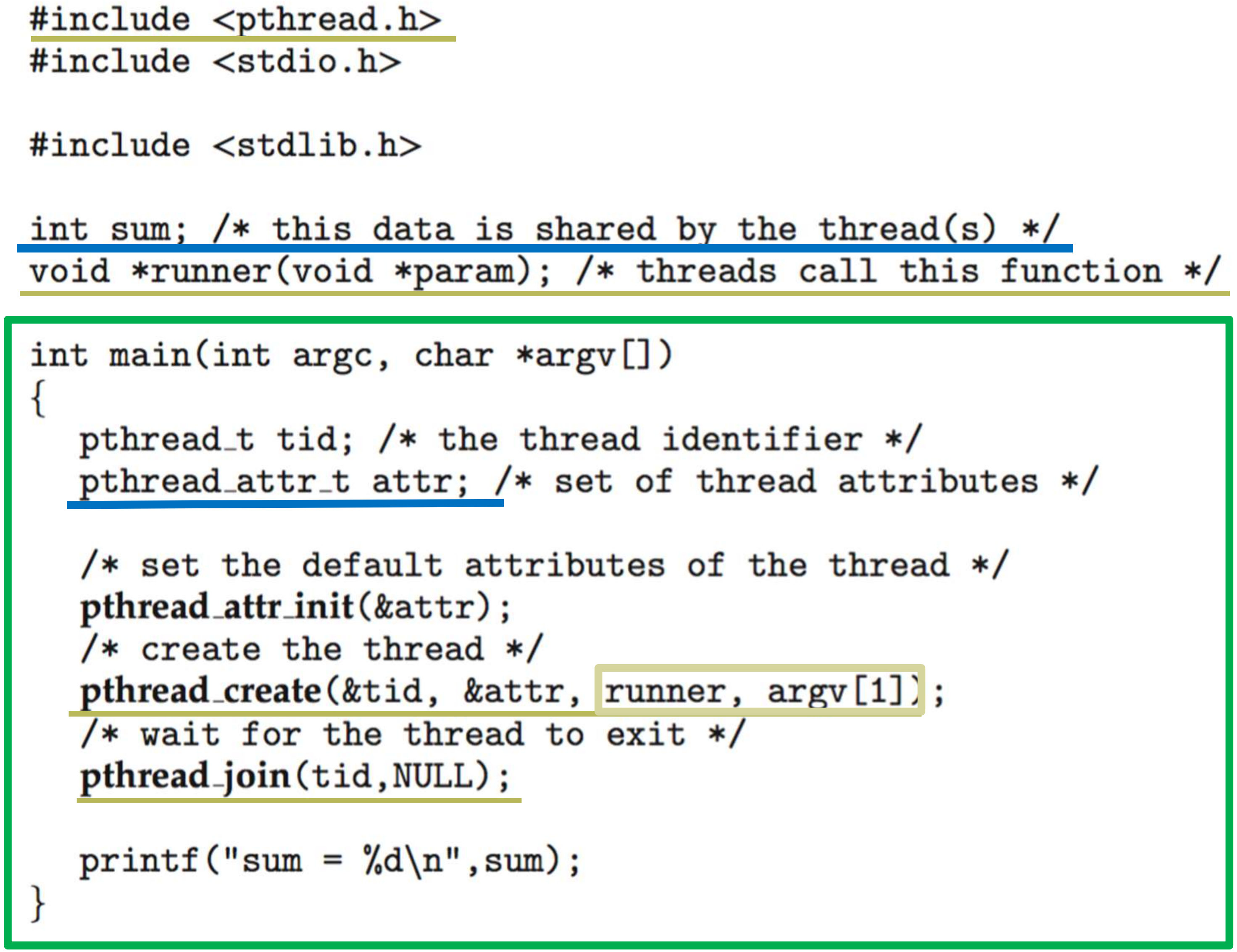
Before returning, a successful call to **pthread_create**() stores the ID of the new thread in the buffer pointed to by *thread*; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.

RETURN VALUE top

On success, **pthread_create**() returns 0; on error, it returns an error number, and the contents of **thread* are undefined.

Pthreads Example

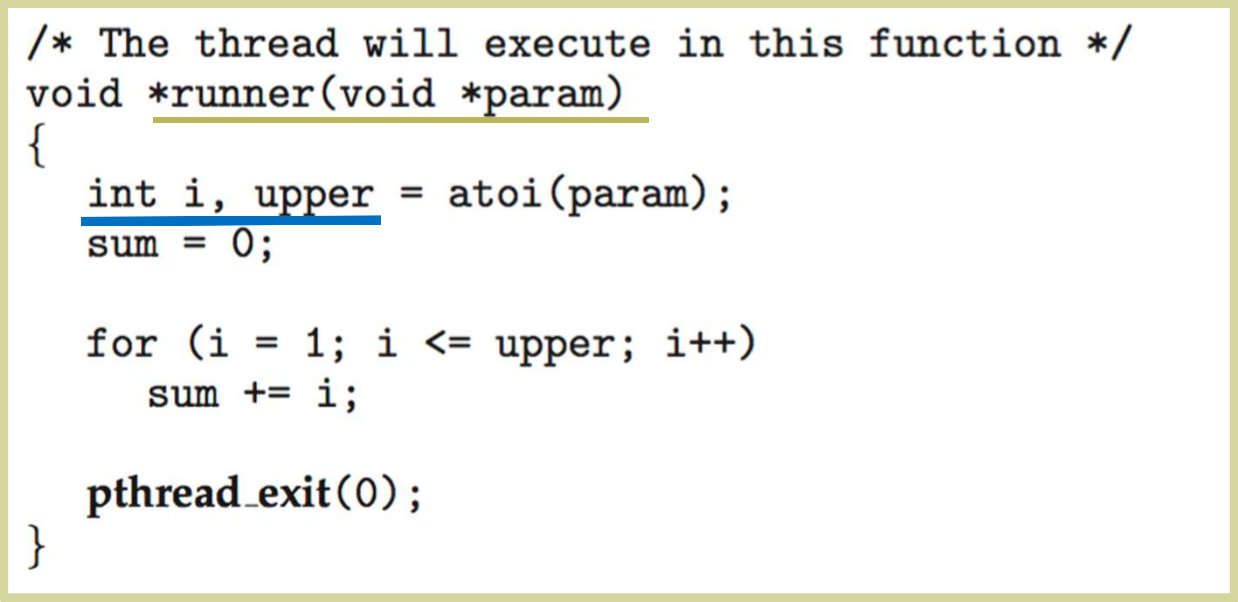
main thread



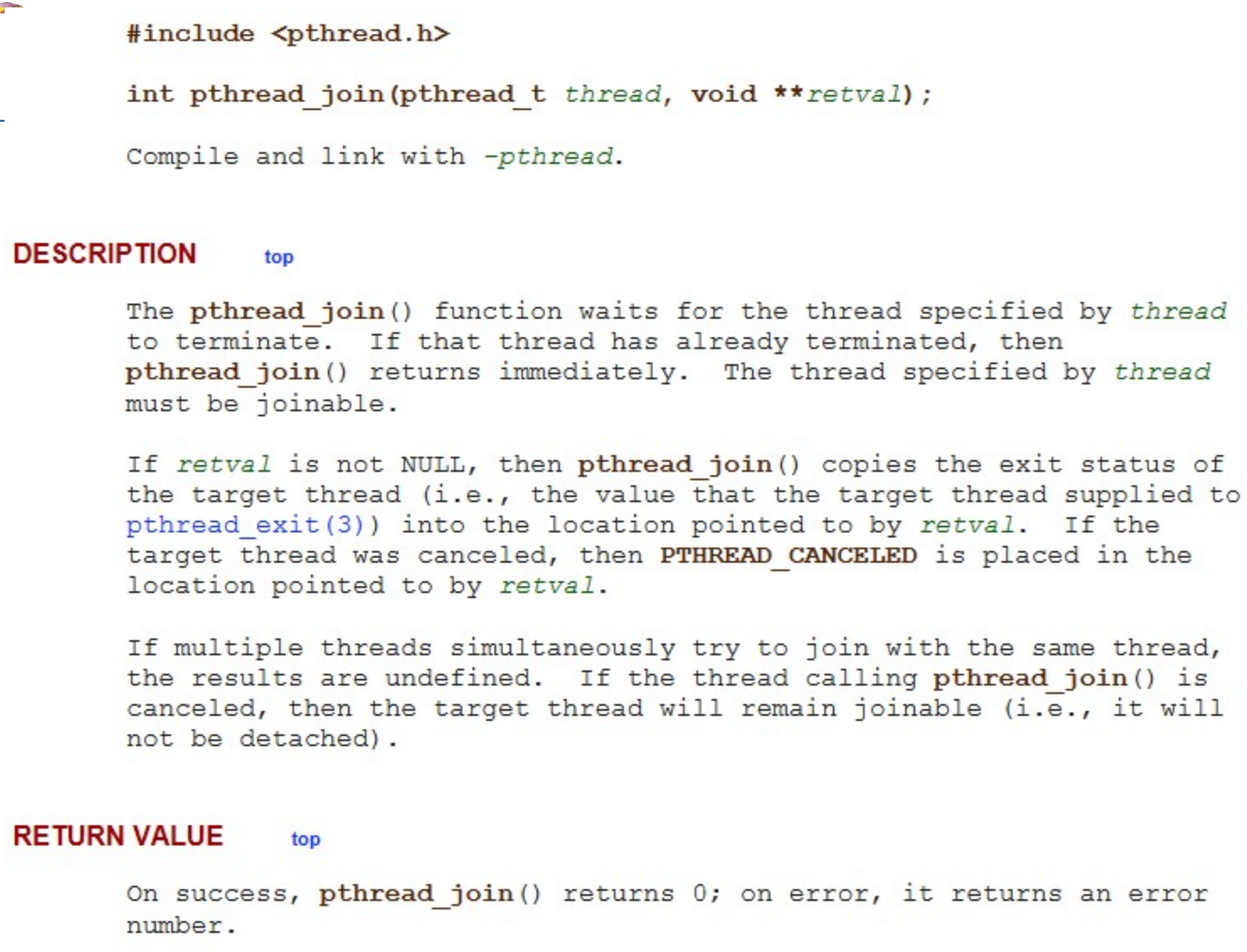
```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

Pthreads Example (cont)

worker thread
(secondary thread)



```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```



```
#include <pthread.h>

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

Compile and link with -pthread.
```

DESCRIPTION top

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.

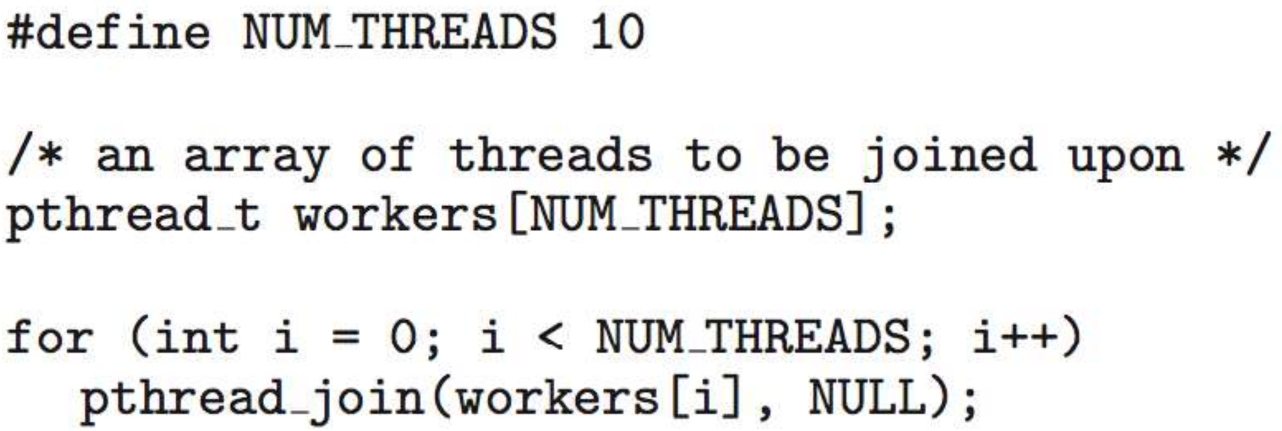
If *retval* is not NULL, then **pthread_join**() copies the exit status of the target thread (i.e., the value that the target thread supplied to pthread_exit(3)) into the location pointed to by *retval*. If the target thread was canceled, then **PTHREAD_CANCELED** is placed in the location pointed to by *retval*.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling **pthread_join**() is canceled, then the target thread will remain joinable (i.e., it will not be detached).

RETURN VALUE top

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

Pthreads Code for Joining 10 Threads



```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.10 Pthread code for joining ten threads.

thread1.c – create worker threads

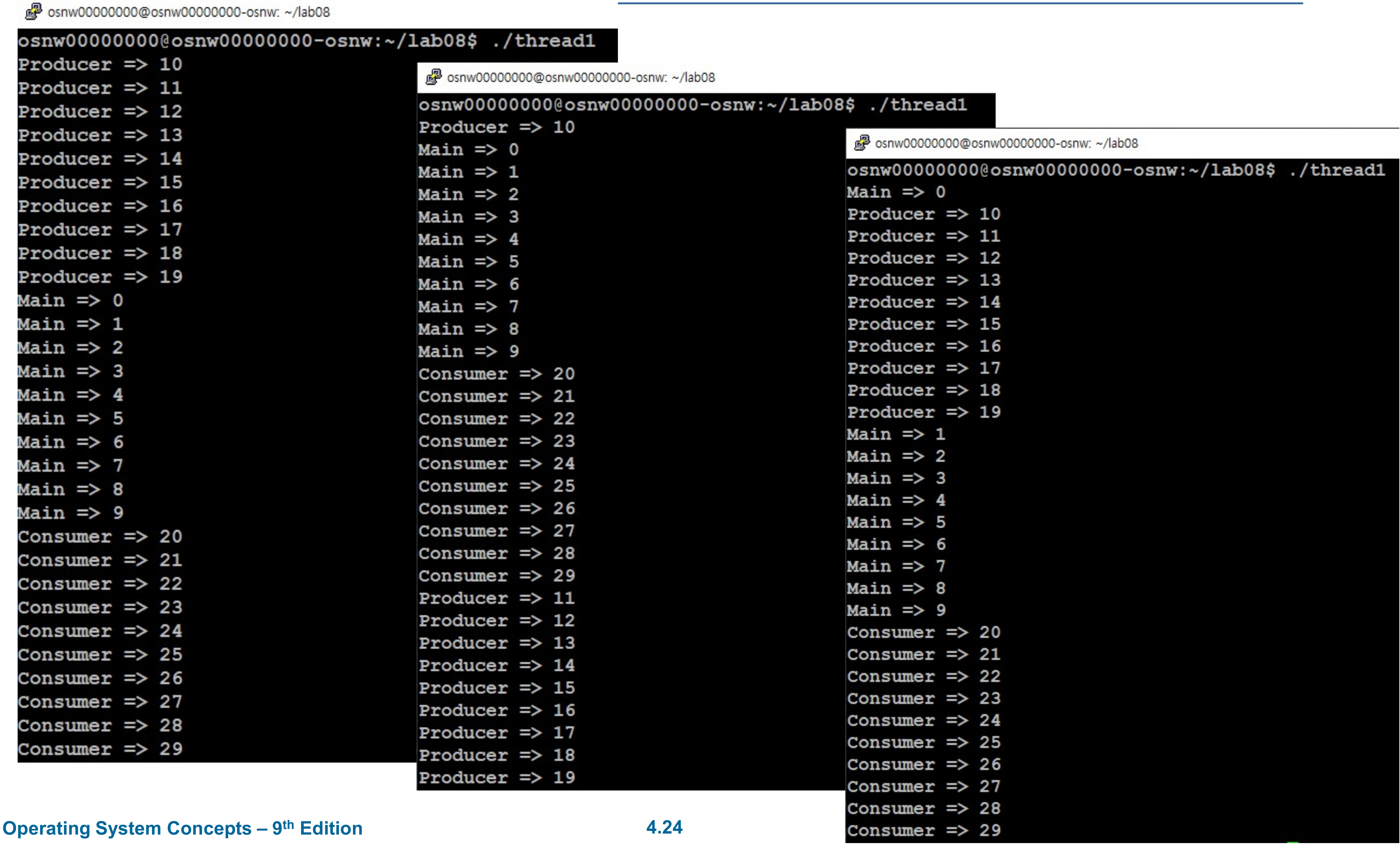
```
void *Producer(void *arg)
{
   int i;

   for(i=10; i<20; i++)
      printf("Producer => %d\n", i);
}

void *Consumer(void *arg)
{
   int i;

   for(i=20; i<30; i++)
      printf("Consumer => %d\n", i);
}
```

```
void main()
{
   int i;
   pthread_t ThreadVector[2];

   pthread_create(&ThreadVector[0], NULL, Producer, NULL);
   pthread_create(&ThreadVector[1], NULL, Consumer, NULL);

   for(i=0; i<10; i++)
      printf("Main => %d\n", i);

   pthread_join(ThreadVector[0], NULL);
   pthread_join(ThreadVector[1], NULL);
}
```

```
$gcc –o thread1 thread1.c -lpthread
$ps –eLf
```

thread1.c – create worker threads

Lab



osnw00000000@osnw00000000-osnw: ~/lab08

```
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread1
Producer => 10
Producer => 11
Producer => 12
Producer => 13
Producer => 14
Producer => 15
Producer => 16
Producer => 17
Producer => 18
Producer => 19
Main => 0
Main => 1
Main => 2
Main => 3
Main => 4
Main => 5
Main => 6
Main => 7
Main => 8
Main => 9
Consumer => 20
Consumer => 21
Consumer => 22
Consumer => 23
Consumer => 24
Consumer => 25
Consumer => 26
Consumer => 27
Consumer => 28
Consumer => 29
```

osnw00000000@osnw00000000-osnw: ~/lab08

```
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread1
Producer => 10
Main => 0
Main => 1
Main => 2
Main => 3
Main => 4
Main => 5
Main => 6
Main => 7
Main => 8
Main => 9
Consumer => 20
Consumer => 21
Consumer => 22
Consumer => 23
Consumer => 24
Consumer => 25
Consumer => 26
Consumer => 27
Consumer => 28
Consumer => 29
Producer => 11
Producer => 12
Producer => 13
Producer => 14
Producer => 15
Producer => 16
Producer => 17
Producer => 18
Producer => 19
```

osnw00000000@osnw00000000-osnw: ~/lab08

```
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread1
Main => 0
Producer => 10
Producer => 11
Producer => 12
Producer => 13
Producer => 14
Producer => 15
Producer => 16
Producer => 17
Producer => 18
Producer => 19
Main => 1
Main => 2
Main => 3
Main => 4
Main => 5
Main => 6
Main => 7
Main => 8
Main => 9
Consumer => 20
Consumer => 21
Consumer => 22
Consumer => 23
Consumer => 24
Consumer => 25
Consumer => 26
Consumer => 27
Consumer => 28
Consumer => 29
```

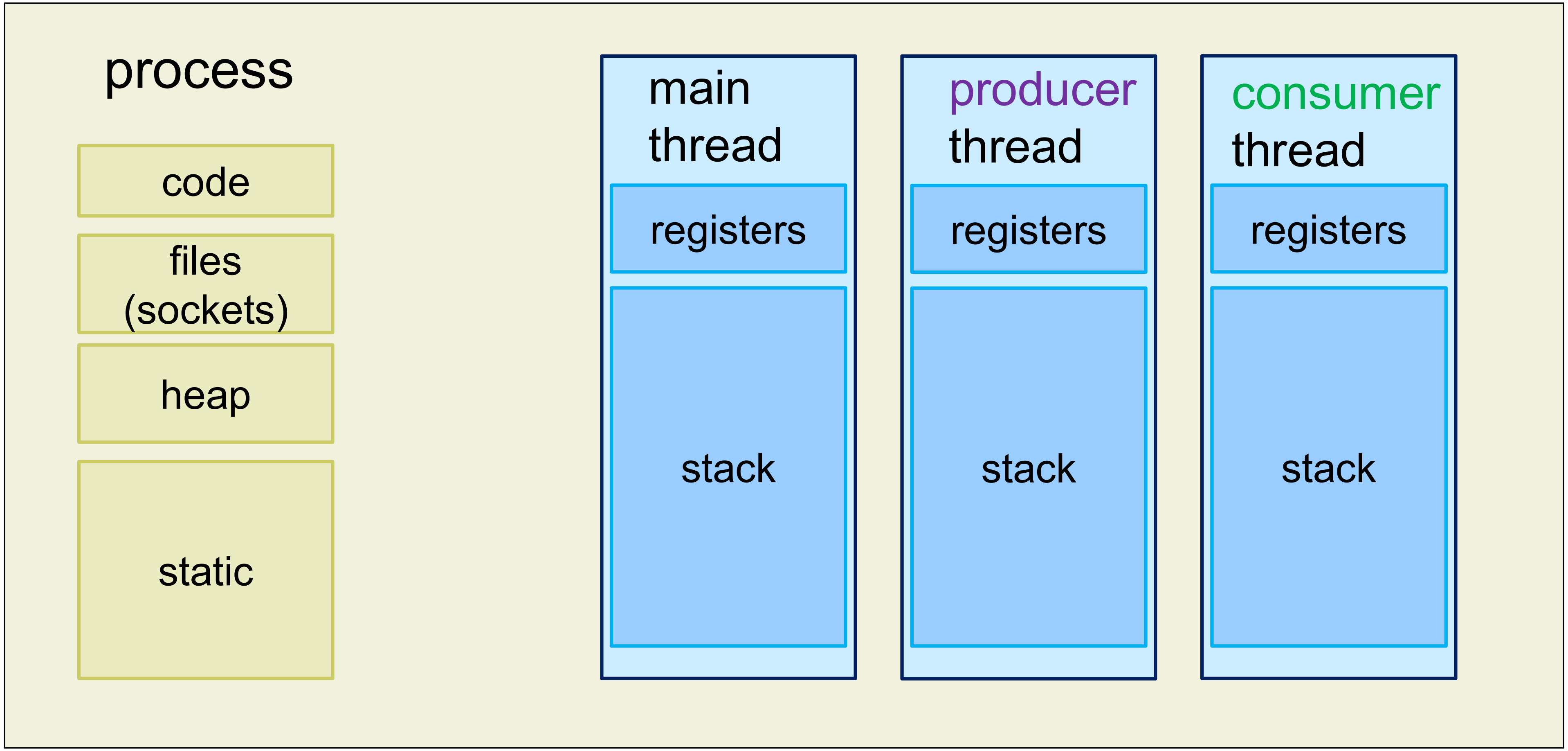
thread2.c – transfer pointer of struct as a parameter to thread function

```
typedef struct
{
   char field1[10];
   char field2[10];
   int  field3;
} PARAMS;

void *Producer(void *arg)
{
   PARAMS *pProducer = (PARAMS *) arg;
   sleep(1);
   printf("Producer => %s %d\n", pProducer->field1, pProducer->field3);
}
void *Consumer(void *arg)
{
   PARAMS *pConsumer = (PARAMS *) arg;
   sleep(2);
   printf("Consumer => %s %d\n", pConsumer->field2, pConsumer->field3);
}
```

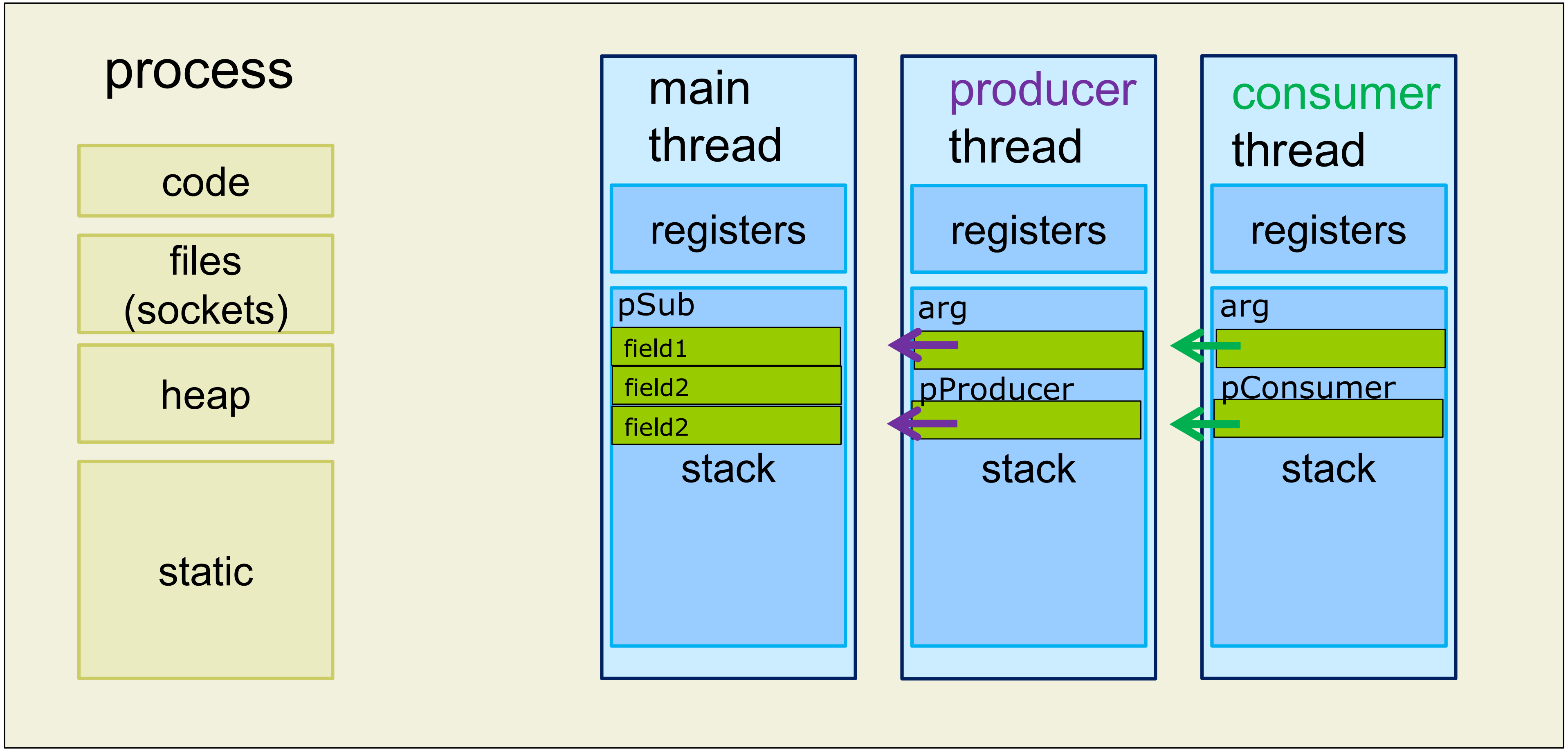
thread2.c – transfer pointer of struct as a parameter to thread function

```
pthread_t ThreadVector[2]; // non-local variables

void main()
{
   PARAMS pSub;

   strcpy(pSub.field1, "hello");
   strcpy(pSub.field2, "world");
   pSub.field3 = 2023;

   pthread_create(&ThreadVector[0], NULL, Producer, (void *) &pSub);
   pthread_create(&ThreadVector[1], NULL, Consumer, (void *) &pSub);

   pthread_join(ThreadVector[0], NULL);
   pthread_join(ThreadVector[1], NULL);
}
```

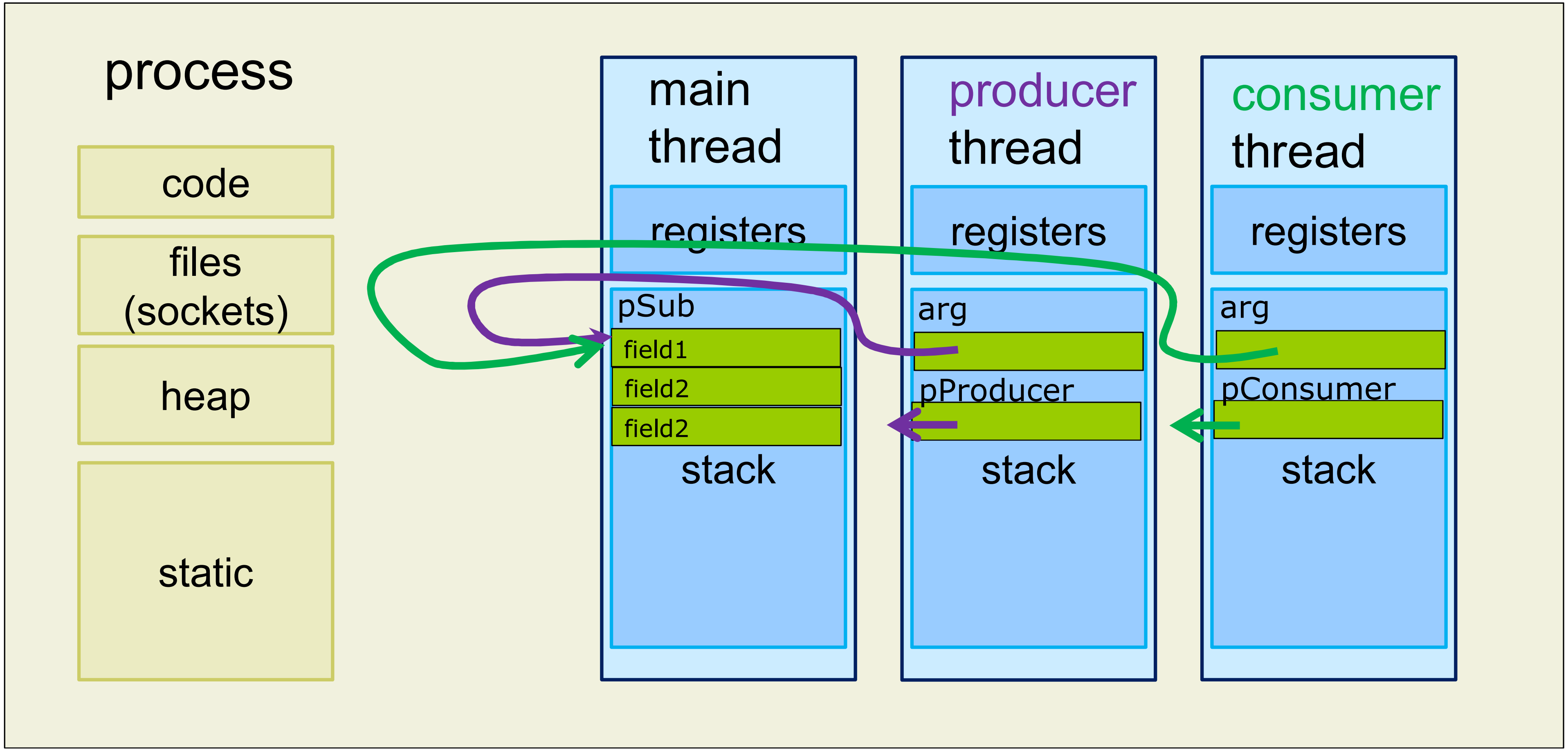
thread2.c – transfer pointer of struct as a parameter to thread function



thread2.c – transfer pointer of struct as a parameter to thread function

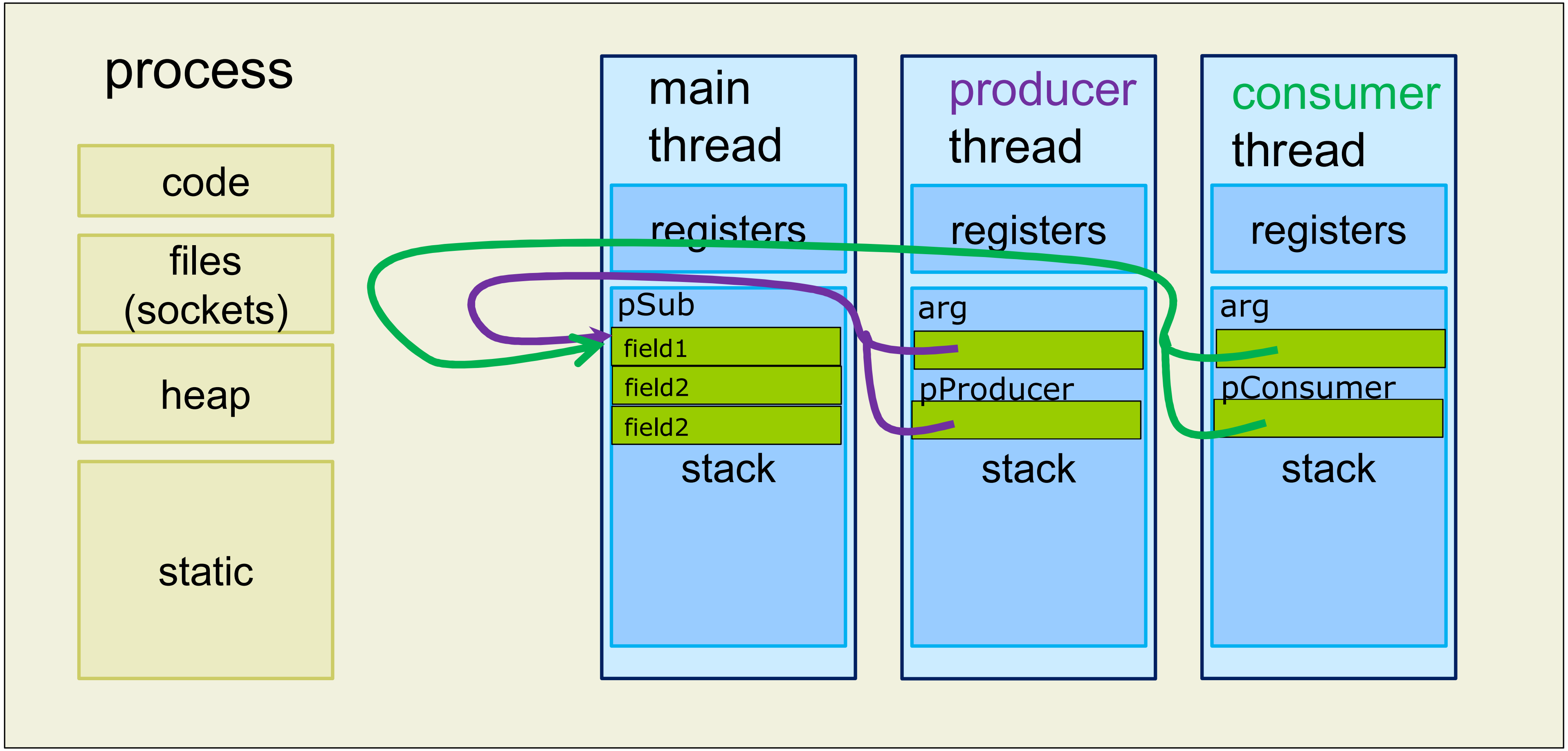


thread2.c – transfer pointer of struct as a parameter to thread function



After executing pthread_create(…, Producer, (void *) &pSub);
void *Producer(void *arg){}
pthread_create(…, Consumer, (void *) &pSub);
void *Consumer(void *arg){}

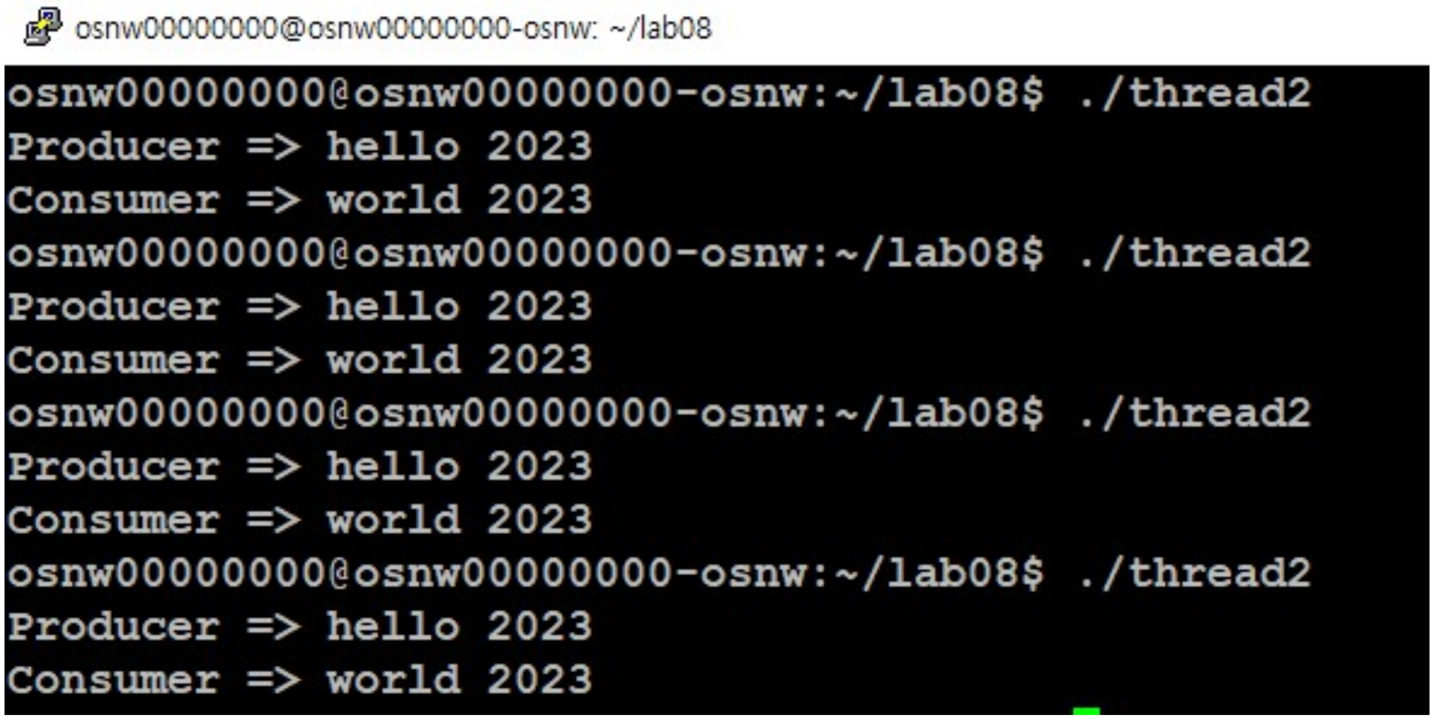
thread2.c – transfer pointer of struct as a parameter to thread function



After executing PARAMS *pProducer = (PARAMS *) arg;
PARAMS *pConsumer = (PARAMS *) arg;

thread2.c – transfer pointer of struct as a parameter to thread function

Lab



osnw00000000@osnw00000000-osnw: ~/lab08

```
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread2
Producer => hello 2023
Consumer => world 2023
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread2
Producer => hello 2023
Consumer => world 2023
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread2
Producer => hello 2023
Consumer => world 2023
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread2
Producer => hello 2023
Consumer => world 2023
```

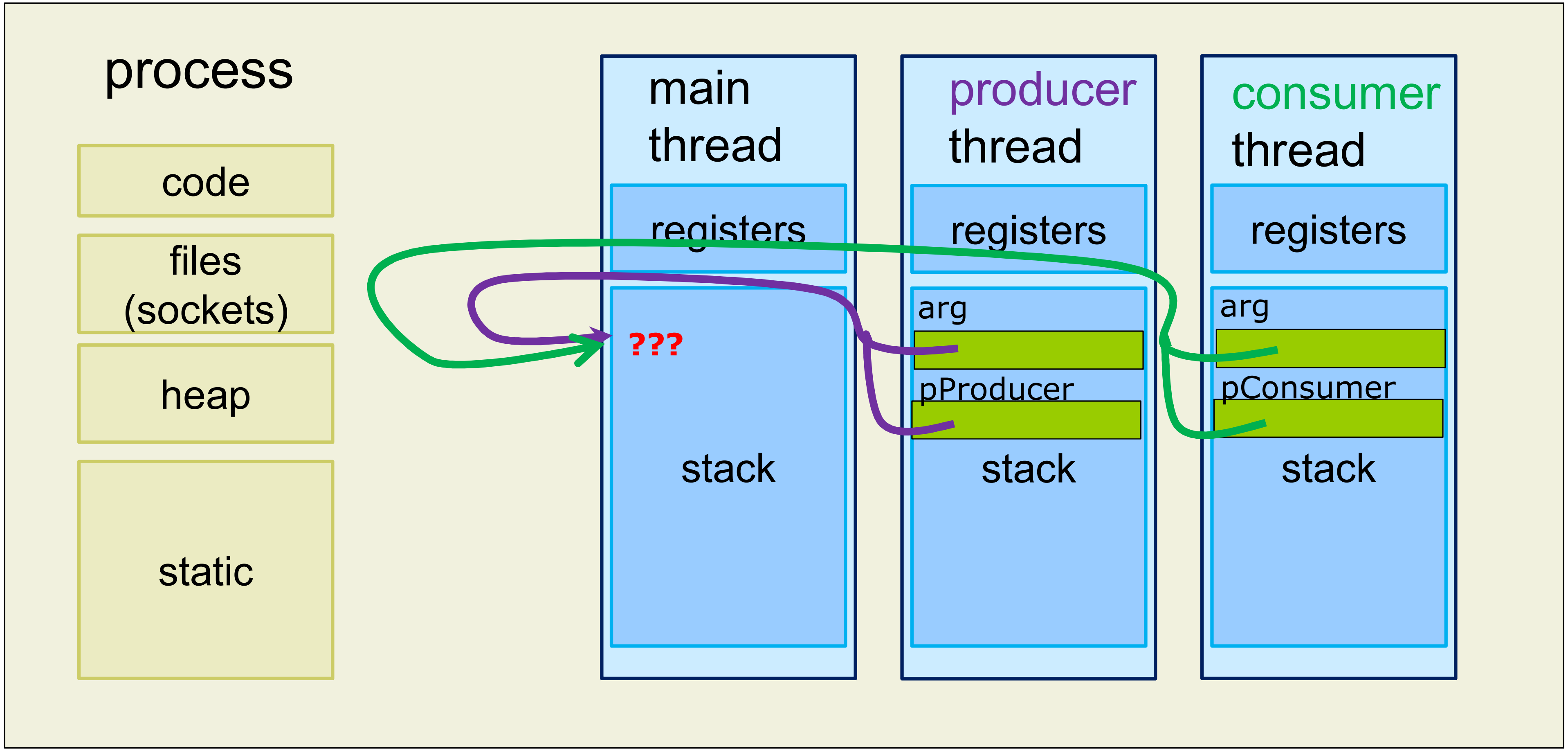
thread3.c – 포인터가 가르키는 데이터의 유효성 문제 1 : automatic variables

```
void sub()
{
   PARAMS pSub;

   strcpy(pSub.field1, "hello");
   strcpy(pSub.field2, "world");
   pSub.field3 = 2024;

   pthread_create(&ThreadVector[0], NULL, Producer, (void *) &pSub);
   pthread_create(&ThreadVector[1], NULL, Consumer, (void *) &pSub);

}
void main()
{
   sub();
   pthread_join(ThreadVector[0], NULL);
   pthread_join(ThreadVector[1], NULL);
}
```

thread2.c

```
void main()
{
   PARAMS pSub;

   strcpy(pSub.field1, "hello");
   strcpy(pSub.field2, "world");
   pSub.field3 = 2024;

   pthread_create(&ThreadVector[0], NULL, Producer, (void *) &pSub);
   pthread_create(&ThreadVector[1], NULL, Consumer, (void *) &pSub);

   pthread_join(ThreadVector[0], NULL);
   pthread_join(ThreadVector[1], NULL);
}
```

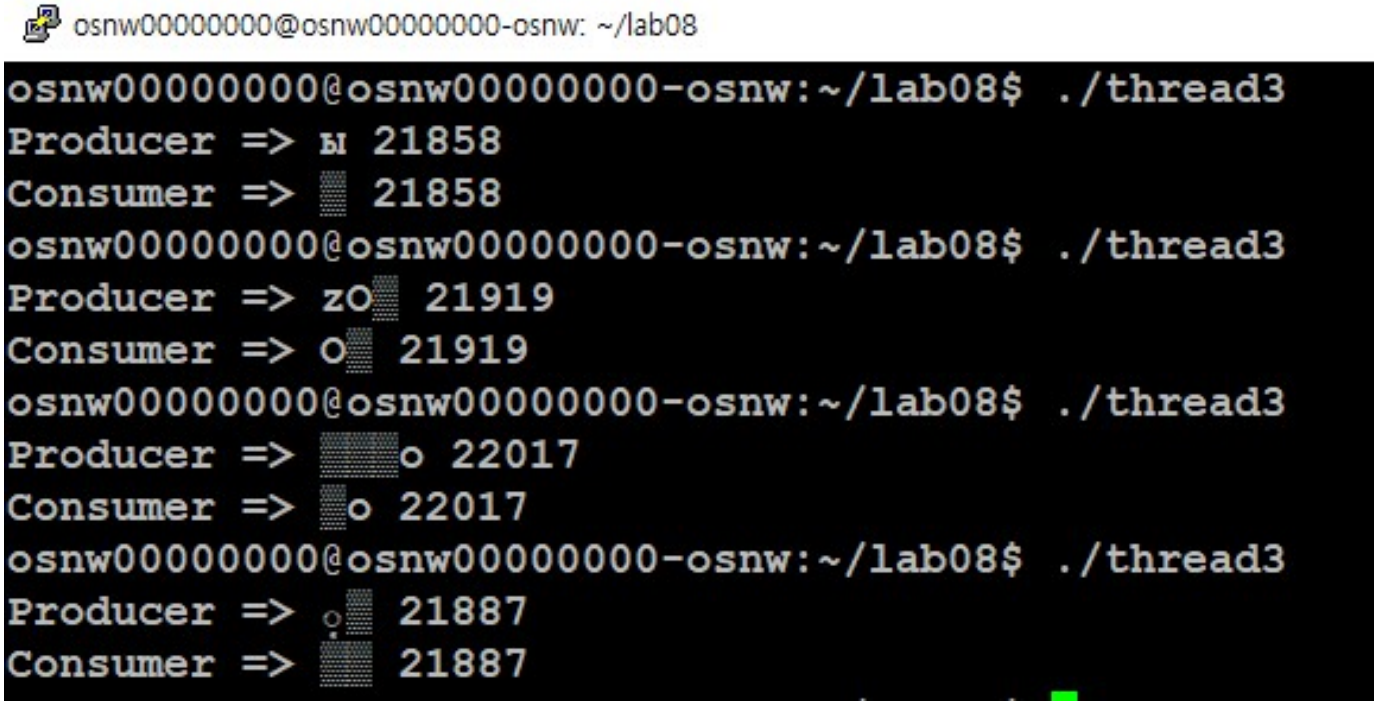
thread3.c – 포인터가 가르키는 데이터의 유효성 문제 1 : automatic variables



After executing sub();

thread3.c – 포인터가 가르키는 데이터의 유효성 문제 1 : automatic variables

Lab



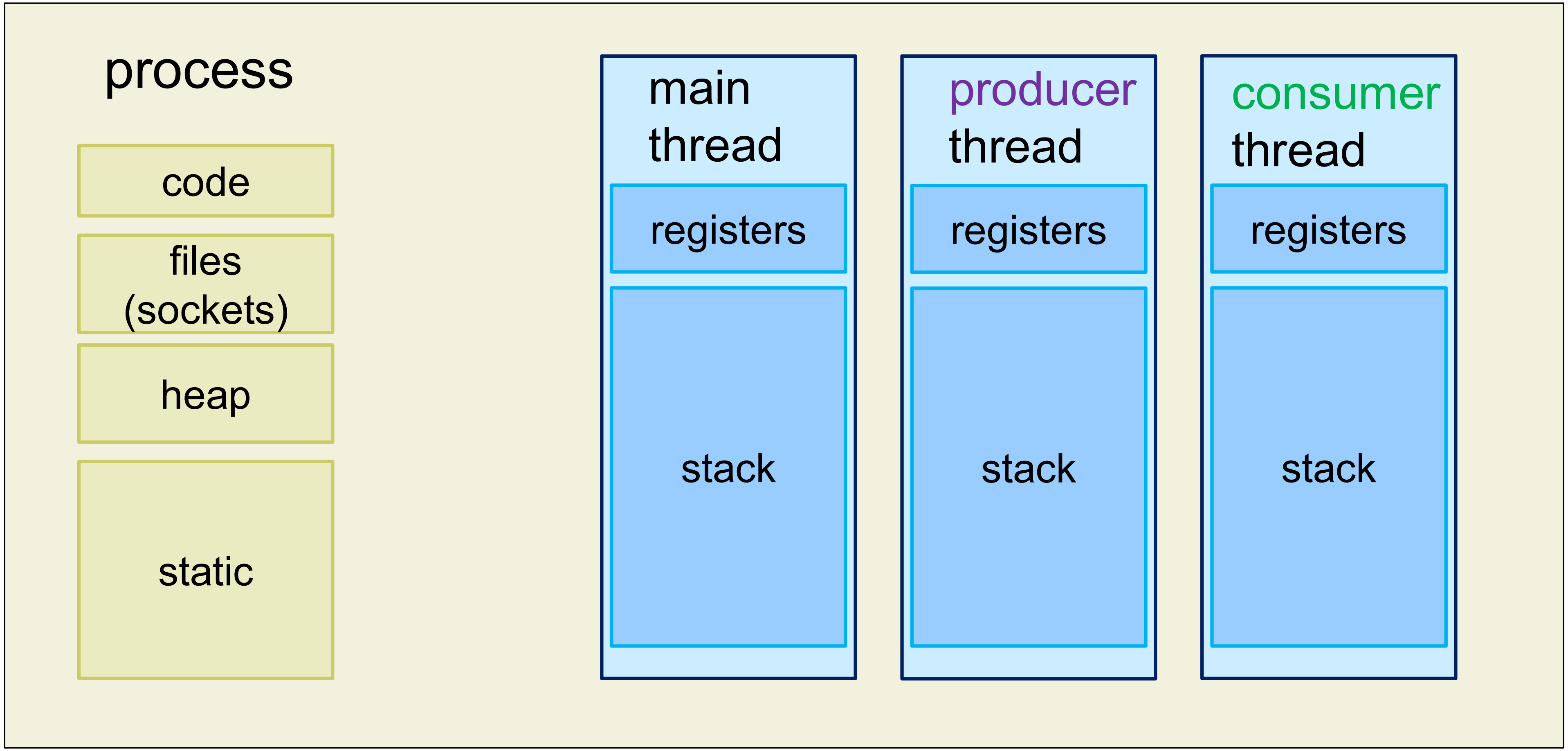
osnw00000000@osnw00000000-osnw: ~/lab08

```
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread3
Producer => ы 21858
Consumer => ▒ 21858
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread3
Producer => zO▒ 21919
Consumer => O▒ 21919
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread3
Producer => ▒▒▒o 22017
Consumer => ▒o 22017
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread3
Producer => ǫ▒ 21887
Consumer => ▒▒ 21887
```

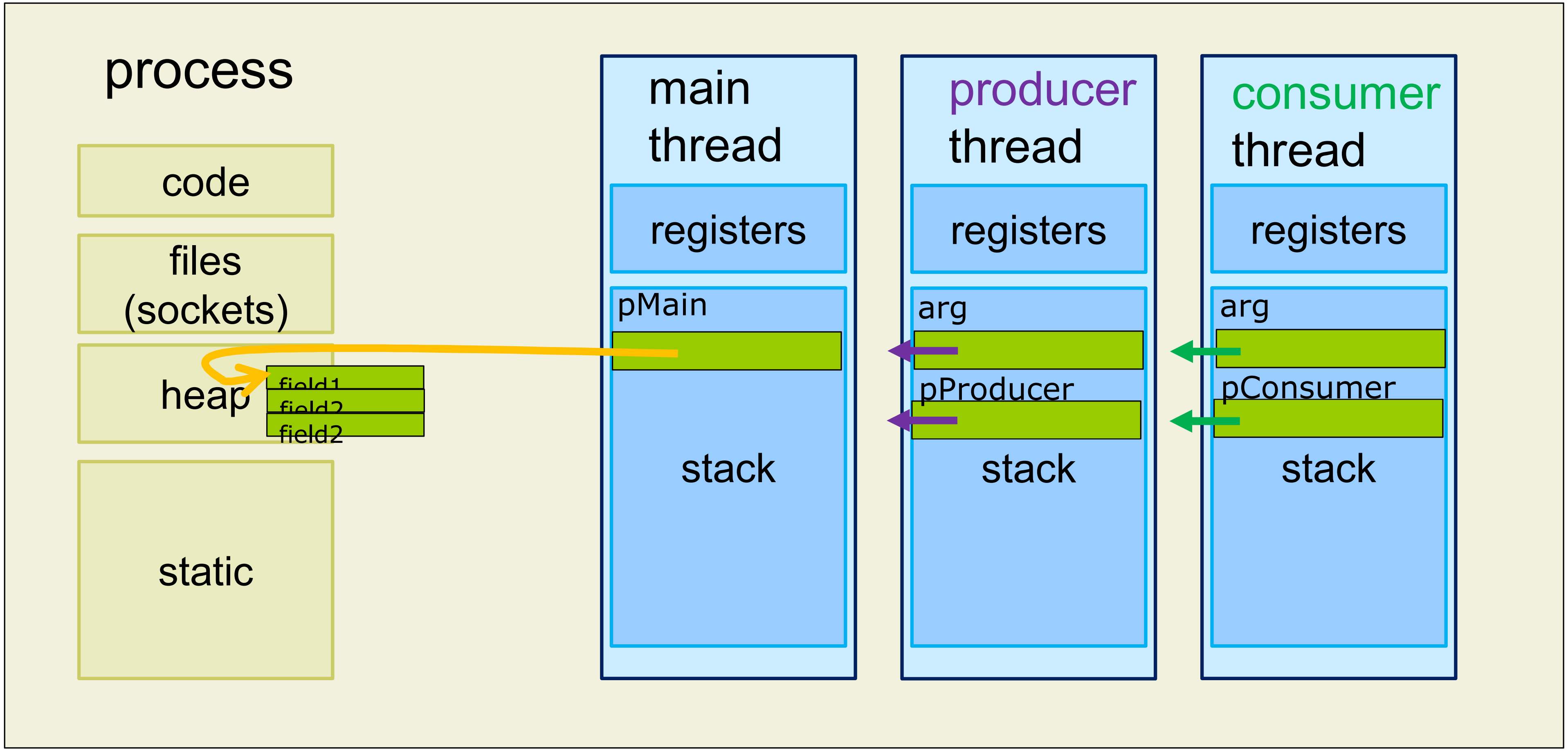
thread4.c – 포인터가 가르키는 데이터의 유효성 문제 2 : heap memory

```
void main()
{
   PARAMS *pMain;
   pthread_t ThreadVector[2];

   pMain = (PARAMS *) malloc( sizeof(PARAMS) );
   strcpy(pMain->field1, "hello");
   strcpy(pMain->field2, "world");
   pMain->field3 = 2021;

   pthread_create(&ThreadVector[0], NULL, Producer, (void *) pMain);
   pthread_create(&ThreadVector[1], NULL, Consumer, (void *) pMain);

   free( pMain );          // no problem ????
   pthread_join(ThreadVector[0], NULL);
   pthread_join(ThreadVector[1], NULL);
}
```

thread4.c – 포인터가 가르키는 데이터의 유효성 문제 2 : heap memory

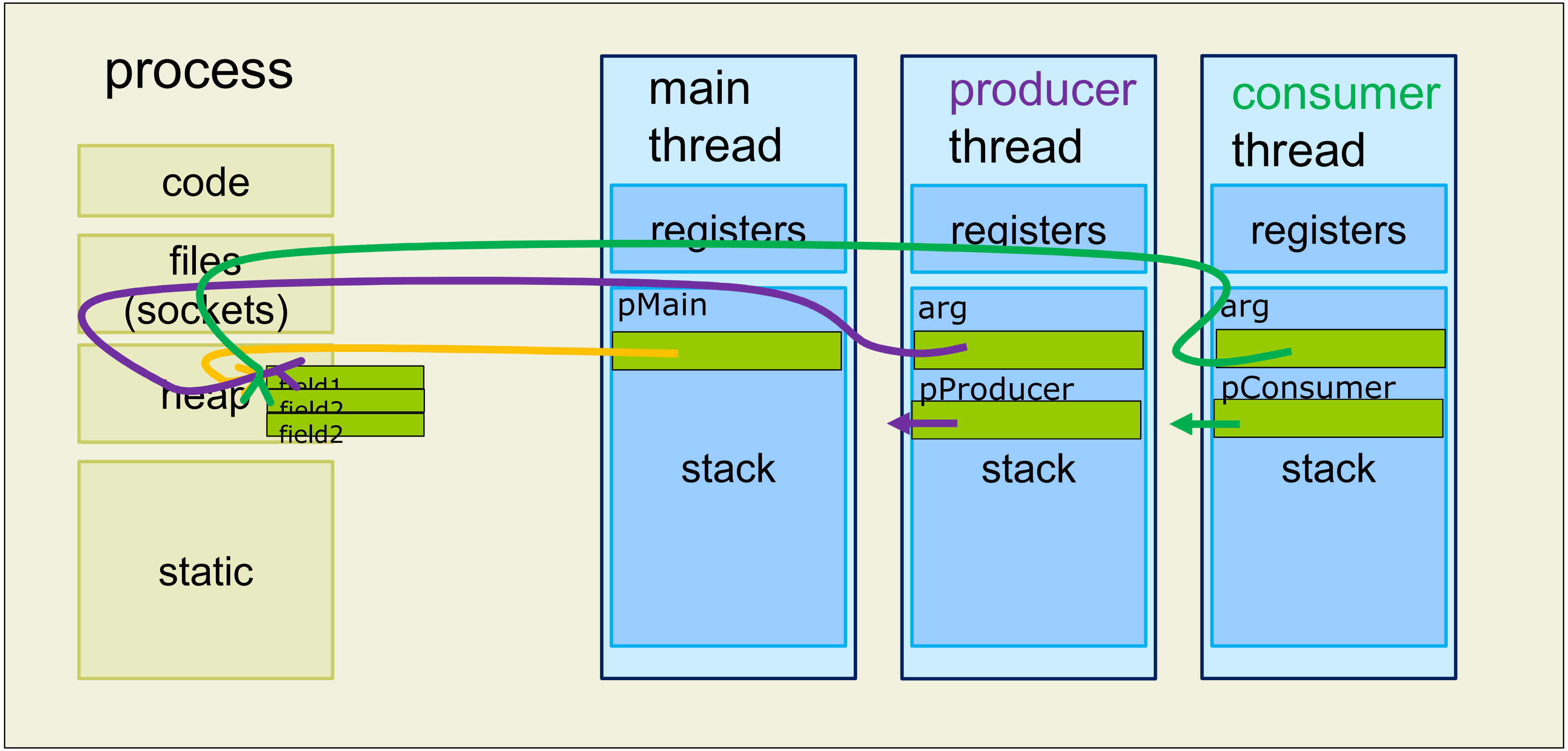


thread4.c – 포인터가 가르키는 데이터의 유효성 문제 2 : heap memory



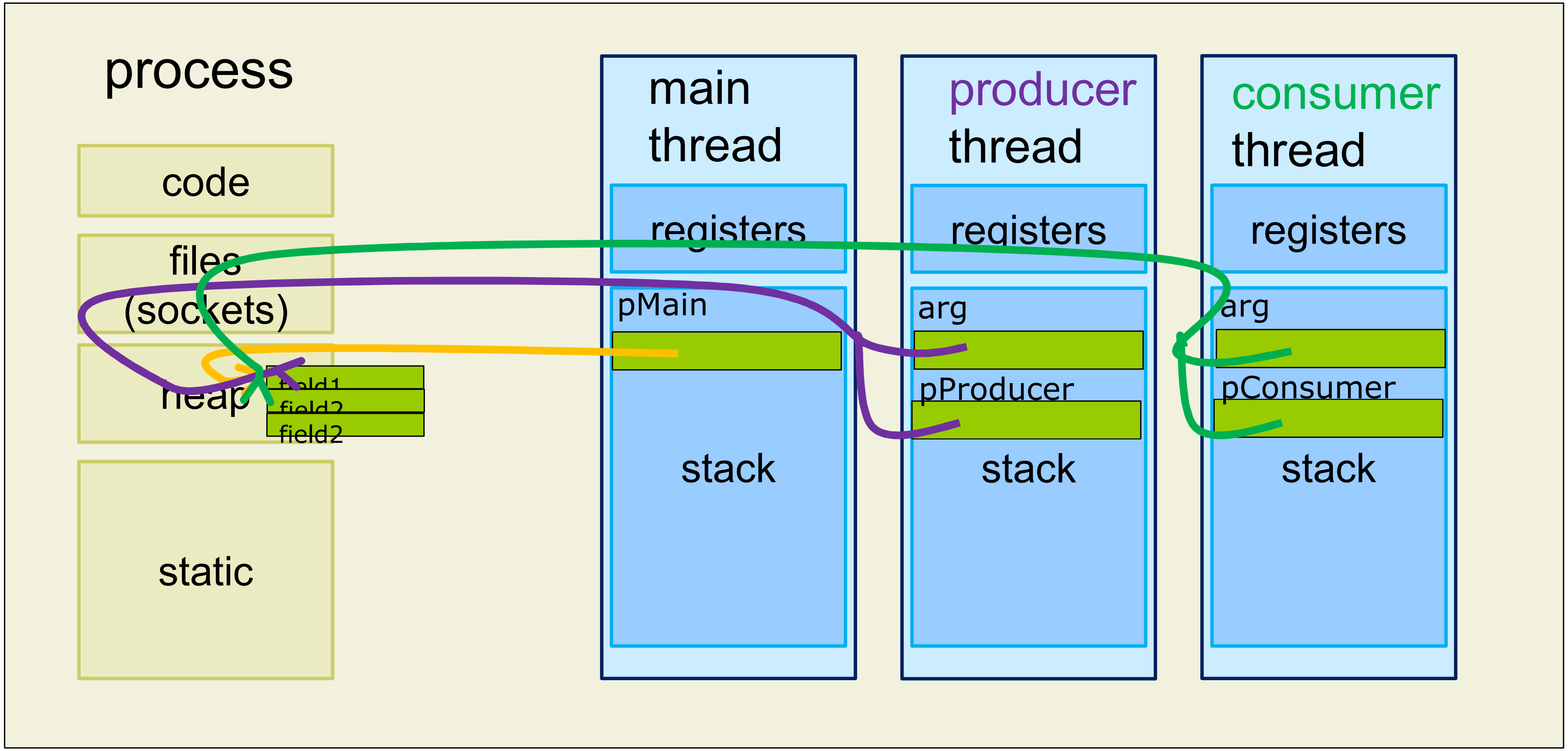
After executing pMain = (PARAMS *) malloc(sizeof(PARAMS));

thread4.c – 포인터가 가르키는 데이터의 유효성 문제 2 : heap memory



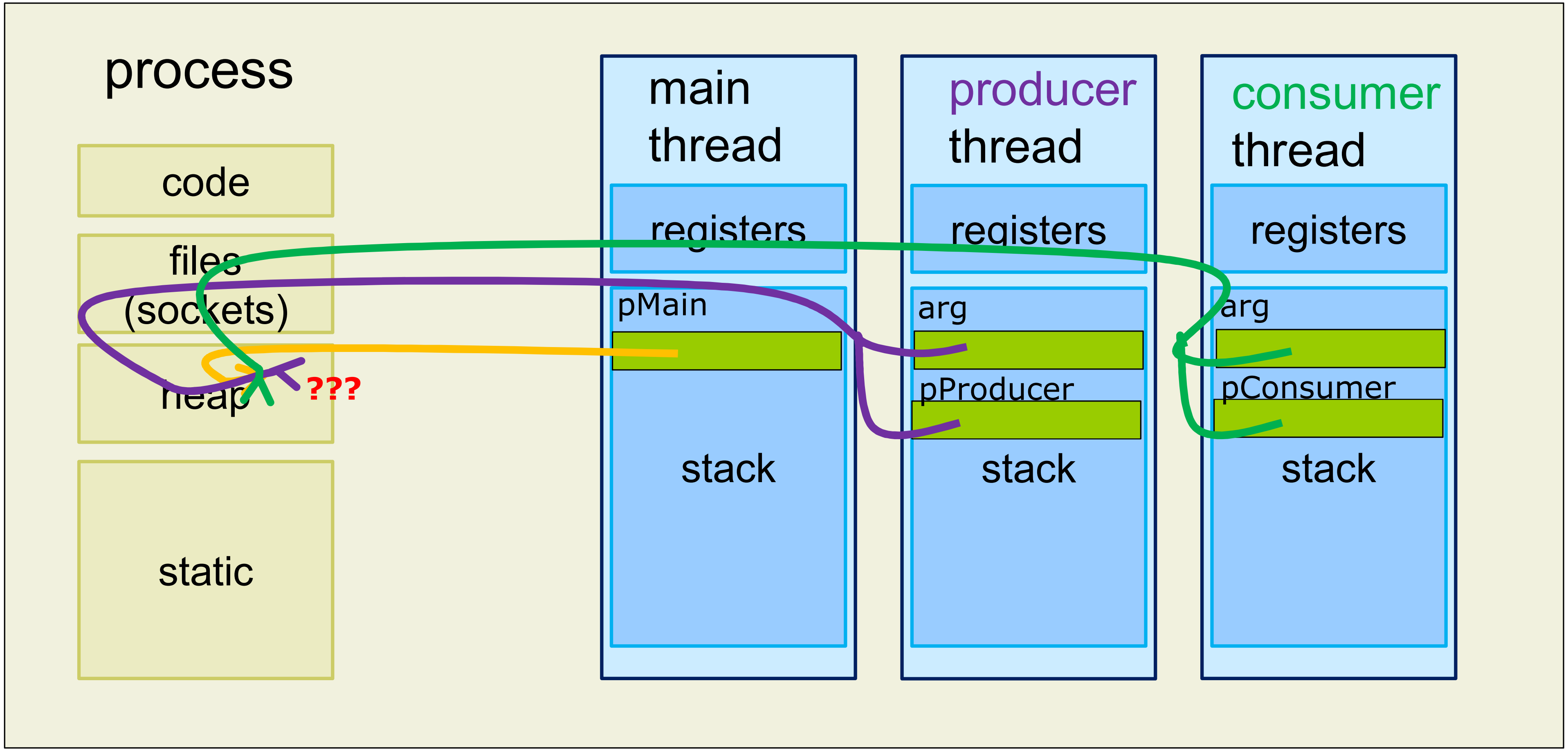
After executing thread_create(.., Producer, (void *) pMain);
void *Producer(void *arg){}
thread_create(.., Consumer, (void *) pMain);
void *Consumer(void *arg){}

thread4.c – 포인터가 가르키는 데이터의 유효성 문제 2 : heap memory



After executing PARAMS *pProducer = (PARAMS *) arg;
PARAMS *pConsumer = (PARAMS *) arg;

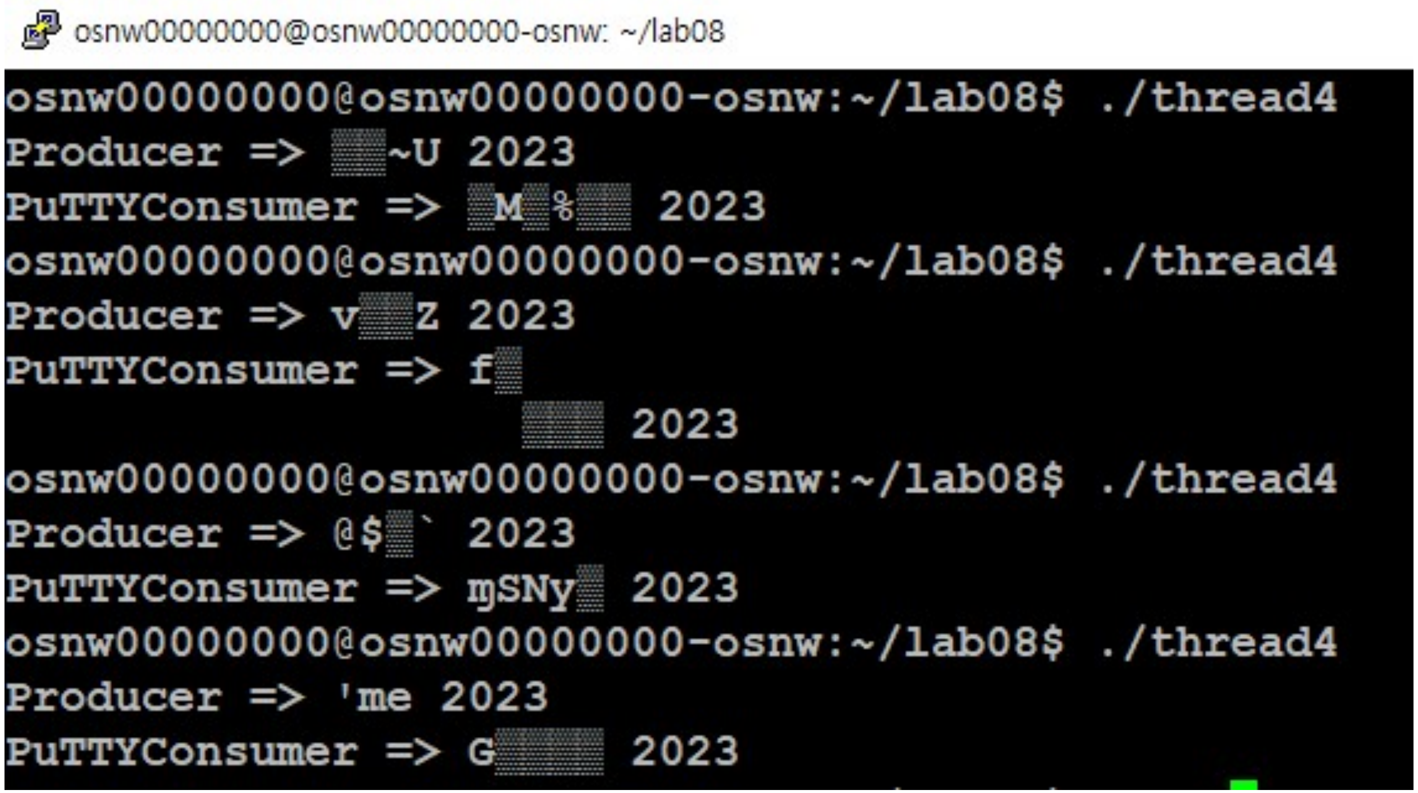
thread4.c – 포인터가 가르키는 데이터의 유효성 문제 2 : heap memory



After executing free(pMain);

thread4.c - 포인터가 가르키는 데이터의 유효성 문제 2 : heap memory

Lab



osnw00000000@osnw00000000-osnw: ~/lab08

```
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread4
Producer => ▒▒~U 2023
PuTTYConsumer => ▒M▒%▒▒ 2023
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread4
Producer => v▒▒Z 2023
PuTTYConsumer => f▒
                   ▒▒▒ 2023
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread4
Producer => @$▒` 2023
PuTTYConsumer => mjSNy▒ 2023
osnw00000000@osnw00000000-osnw:~/lab08$ ./thread4
Producer => 'me 2023
PuTTYConsumer => G▒▒▒▒ 2023
```