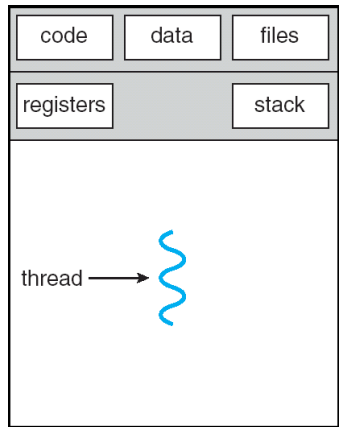


Threads

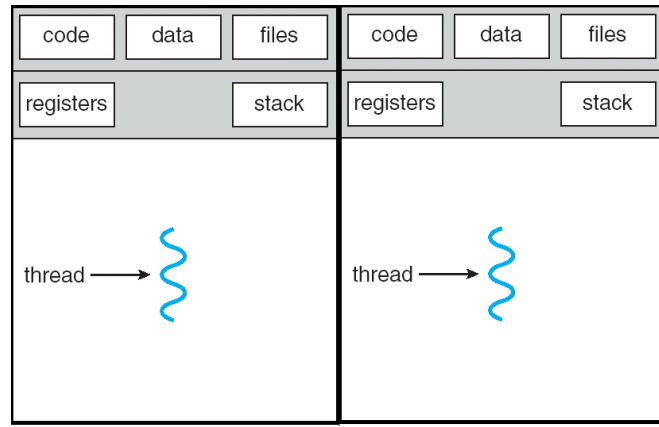
a new abstraction of program execution



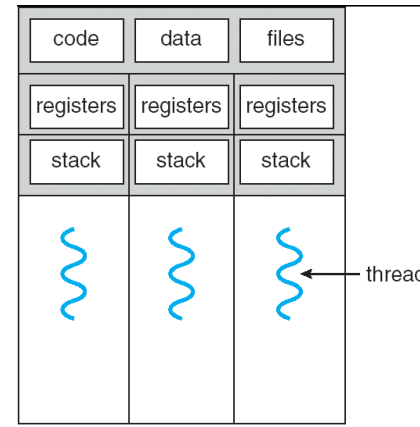
Memory Map of Multi-process vs Multithreading



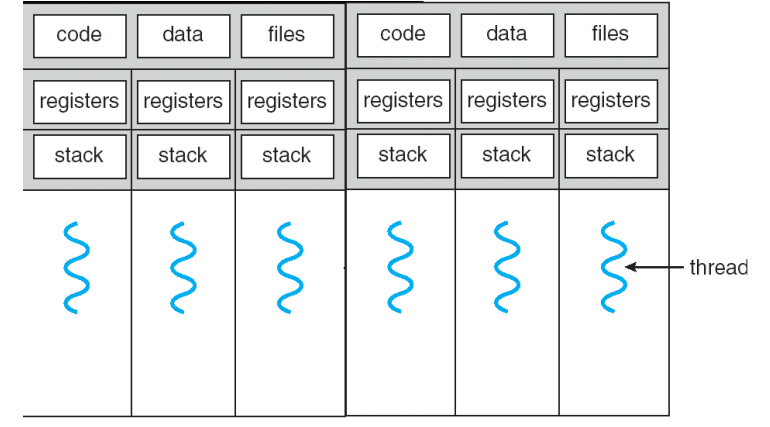
Single Process
Single Thread



Multi-Process Single Thread



Single Process
Multithreaded



Multi-Process Multithreaded

Multi-process Model vs Multithreaded Model

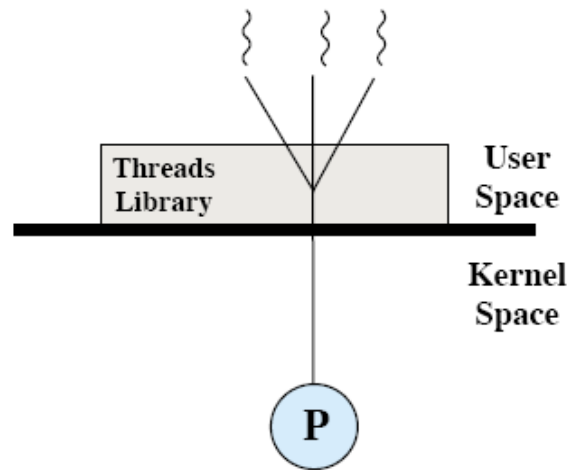
- Process Spawning
 - Process creation involve following actions:
 - Setting up process control block (PCB)
 - Allocation of memory address space
 - Loading program into allocated memory address space
 - Passing PCB to scheduler to queue up the process to run
- Thread Spawning
 - Threads are created within and belonging to processes
 - All the threads created within one process share the resources of the process including the address space
 - Scheduling is performed on a per-thread basis.
 - The thread model is a finer grain scheduling model than the process model
 - Threads have a similar lifecycle as the processes and will be managed mainly in the same way as processes are

Why use multi-threaded model over multi-process model?

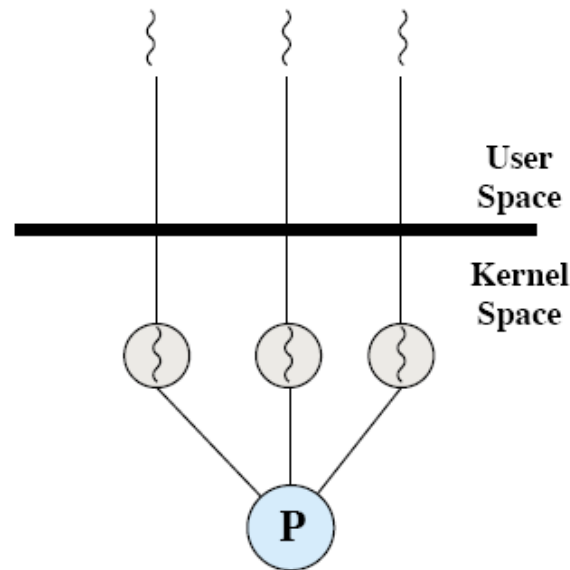
- A common terminology:
 - Heavyweight Process = Process
 - Lightweight Process = Thread
- Advantages (Thread vs. Process):
 - Much quicker to create a thread than a process
 - spawning a new thread only involves allocating a new stack and a new CPU state block
 - Much quicker to switch between threads than to switch between processes
 - Threads share data easily
- Disadvantages (Thread vs. Process):
 - Processes are more flexible
 - They don't have to run on the same processor
 - No security between threads: One thread can stomp on another thread's data
 - For threads which are supported by user thread package instead of the kernel:
 - If one thread blocks, all threads in task block.

Thread Implementation

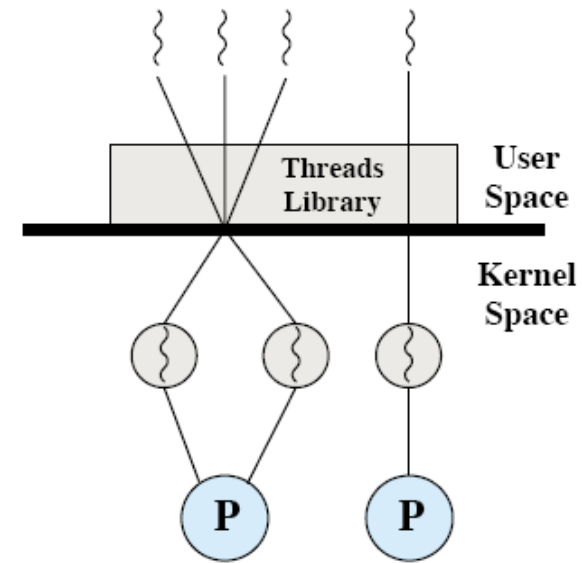
- Two broad categories of thread implementation
 - User-Level Threads (ULT)
 - Kernel-Level Threads (KLT)



Pure User-Level Thread (ULT)



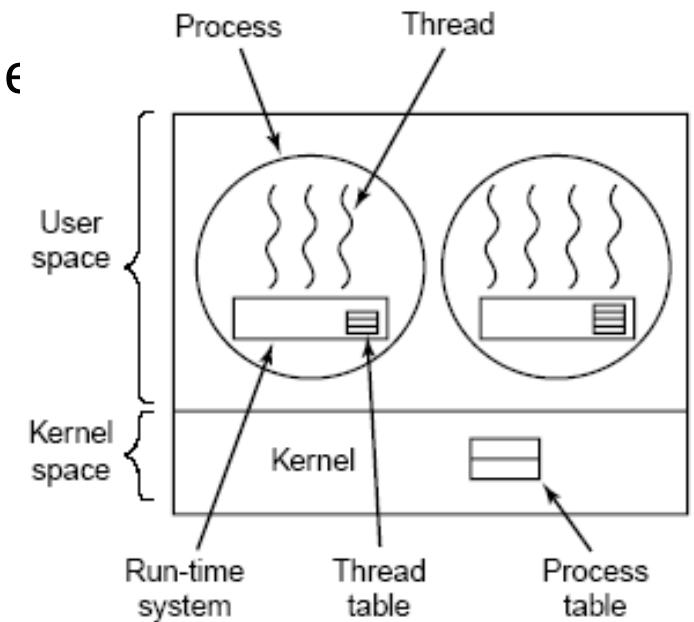
Pure Kernel-Level Thread (KLT)



Combined Level Threads (ULT/KLT)

Thread Implementation

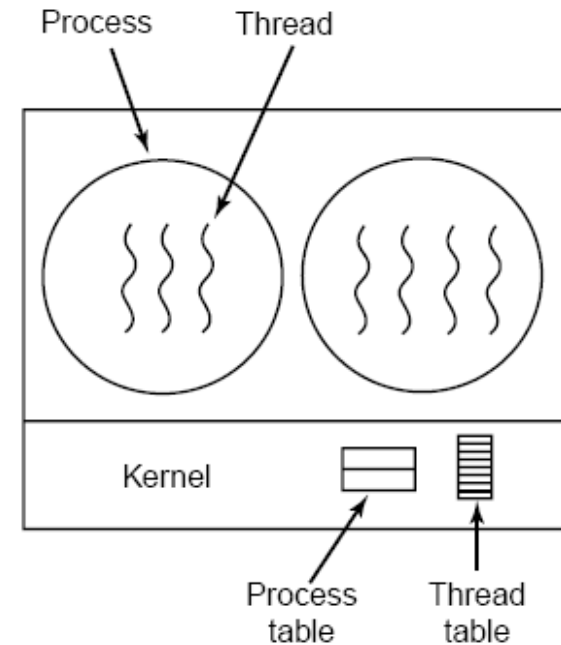
- User-Level Threads (ULT)
 - Kernel is not aware of existence of threads, it knows only processes with one thread of execution
 - Each user process manages its own private thread table
 - Advantages:
 - light thread switching: does not need kernel mode privileges
 - cross-platform: ULT can run on any underlying O/S
 - Disadvantage:
 - if a thread blocks, the entire process is blocked, including all other threads in it



User-Level Thread Package

Thread Implementation

- Kernel-Level Thread (KLT)
 - the kernel knows about and manages the threads: creating and destroying threads are system calls
 - Advantage:
 - fine-grain scheduling, done on a thread basis
 - if a thread blocks, another one can be scheduled without blocking the whole process
 - Disadvantage:
 - heavy thread switching involving mode switch



Kernel-Level Thread Package

POSIX Threads (pthreads) Library

- OS Support: FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, Android and Solaris. DR-DOS and Microsoft Windows implementation also exists
- Library : libpthread
- Header file : pthread.h
- More than 100 threads functions categorized in to 4 category
 - Thread Management
 - Mutexes
 - Condition Variables
 - Synchronization between Threads using read/write locks and barriers
- To compile with gcc you need to use `-pthread` option of gcc to link libpthread

Thread Identification

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

```
pthread_t pthread_self(void);
```

Returns thread ID of the calling thread

Note: Each process has at least single thread running by default even if you have not created thread explicitly

Thread Creation

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

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

Returns 0 on success and error number in case of failure

Parameters:

- tidp : pointer to thread ID variable which will be set with thread ID for newly created thread
- attr : specific attributes to control the thread (default value to pass is NULL)
- start_routine : pointer to function which will be executed upon creation of a new thread
- arg: pointer to argument(s) to be passed to start_routine function. **Note if more than one parameter need to be passed than you need to use structure instead of primitive type**

Thread termination

- 4 possible ways
 - **Implicit termination:** the thread routine returns; usually what we'll use
 - **Explicit termination:** the thread calls `pthread_exit()`
 - **Process exit:** any thread calls `exit()`, which terminates the process and all associated threads; maybe not what you really want
 - **Thread cancellation:** another thread calls `pthread_cancel()` to terminate a specific thread

Thread Termination

- If thread calls `exit()` system call, complete process will be terminated which is generally not desirable because the goal is to terminate a thread and not the process, hence we use `pthread_exit()` as below

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

```
void pthread_exit(void * retval);
```

Does not return anything

Thread termination functions

- **`void pthread_exit(void *val);`**
 - terminates current thread, with a return value equal to the pointer **`val`**
 - Note: Upon termination, a thread releases its runtime stack.
 - Therefore the pointer should point to:
 - a global variable, or
 - a dynamically allocated variable.

Thread Synchronization

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

```
int pthread_join(pthread_t tid, void **retval);
```

thread calling pthread_join waits for target thread with tid to terminate.
Target thread is terminated when it calls pthread_exit()

Returns 0 on success and error number in case of failure

Parameters:

tid : thread ID of target thread for which calling thread is waiting to terminate

retval : when target thread called pthread_exit(), exit status of the same is returned in retval

Thread reaping

- Threads are reaped by `pthread_join()`

`int pthread_join(pthread_t tid, void **val)`

- on success, `*val` is the return value of the terminated thread

Thread Synchronization

- One thread can request another thread belonging to the same process to terminate by calling `pthread_cancel()`

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

```
int pthread_cancel(pthread_t tid);
```

Returns 0 on success and error number in case of failure

Parameters:

tid : thread ID of target thread which is requested to be cancelled

Note: Behavior of the target thread will be the same as if it had called `pthread_exit()`

Process vs Thread System Calls

Process Primitive	Thread Primitive	Description
fork	pthread_create	Create a new flow of control
exit	pthread_exit	Exit from an existing flow of control
waitpid	pthread_join	Get exit status from flow of control
atexit	pthread_cancel_push	Register function to be called at exit from flow of control
getpid	pthread_self	Get ID of flow of control
abort	pthread_cancel	Request abnormal termination of flow of control

Example Program

```
#include "csapp.h"

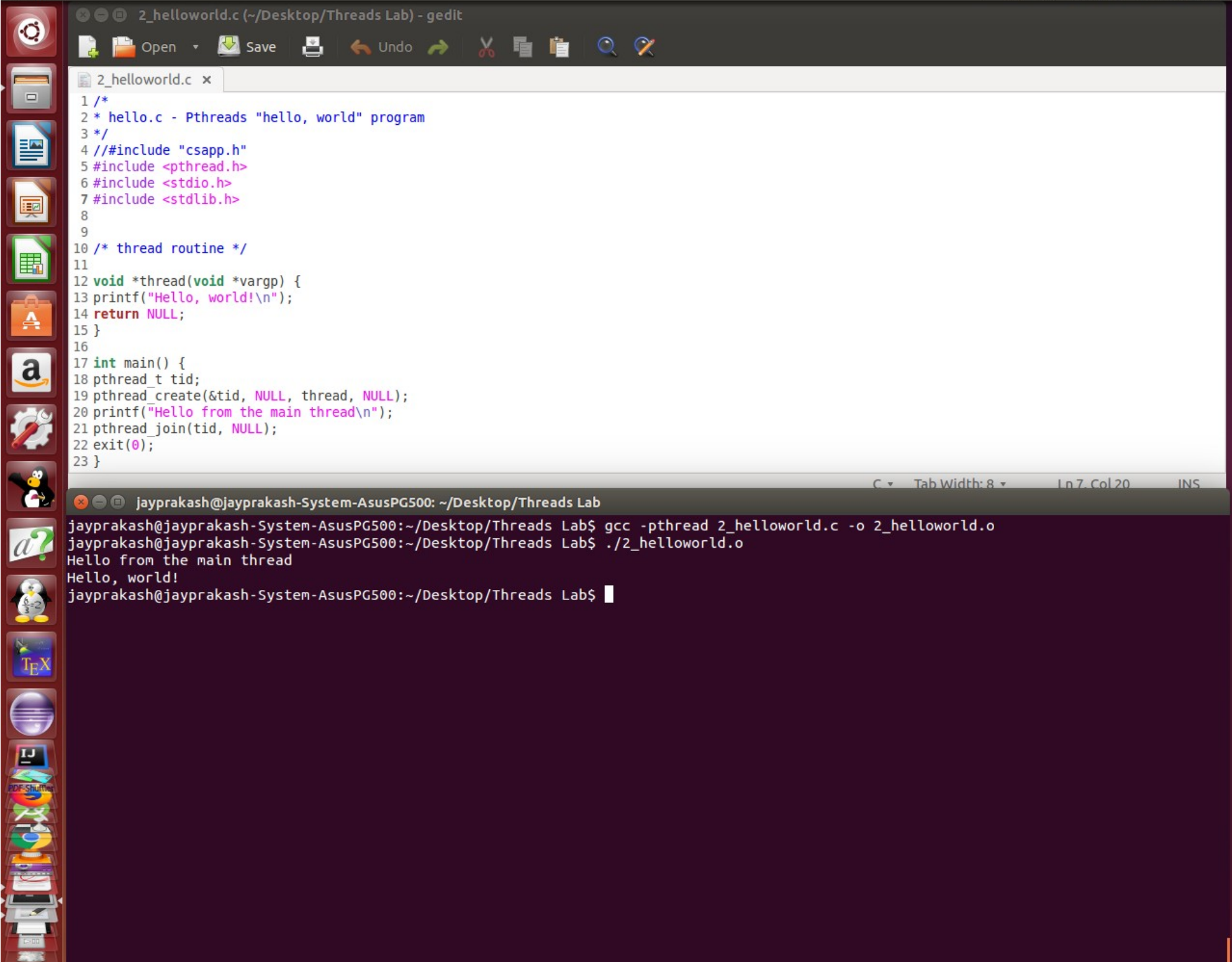
/* thread routine */
void *mythread(void *vargp) {
    printf("Hello from the other thread\n");
    return NULL;
}

int main() {
    pthread_t tid;

    Pthread_create(&tid, NULL, mythread, NULL);
    printf("Hello from the main thread\n");
    Pthread_join(tid, NULL);
    exit(0);
}
```

Our previous example

- No assumption about the statement execution order between threads is possible
- The output of the main thread might happen before the output of thread mythread
- `printf()` is **thread-safe**
 - Even if called by 2 threads simultaneously, it is guaranteed that outputs won't be mixed



The image shows a desktop environment with a terminal window and a gedit editor. The terminal window is titled "jayprakash@jayprakash-System-AsusPG500: ~/Desktop/Threads Lab" and shows the execution of a C program. The gedit editor is titled "2_helloworld.c (~/Desktop/Threads Lab) - gedit" and contains the source code for the program. The code is a C program that uses pthreads to create a thread and print "Hello, world!".

```
1 /*
2 * hello.c - Pthreads "hello, world" program
3 */
4 //include "csapp.h"
5 #include <pthread.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8
9
10 /* thread routine */
11
12 void *thread(void *vargp) {
13     printf("Hello, world!\n");
14     return NULL;
15 }
16
17 int main() {
18     pthread_t tid;
19     pthread_create(&tid, NULL, thread, NULL);
20     printf("Hello from the main thread\n");
21     pthread_join(tid, NULL);
22     exit(0);
23 }
```

The terminal output shows the compilation and execution of the program:

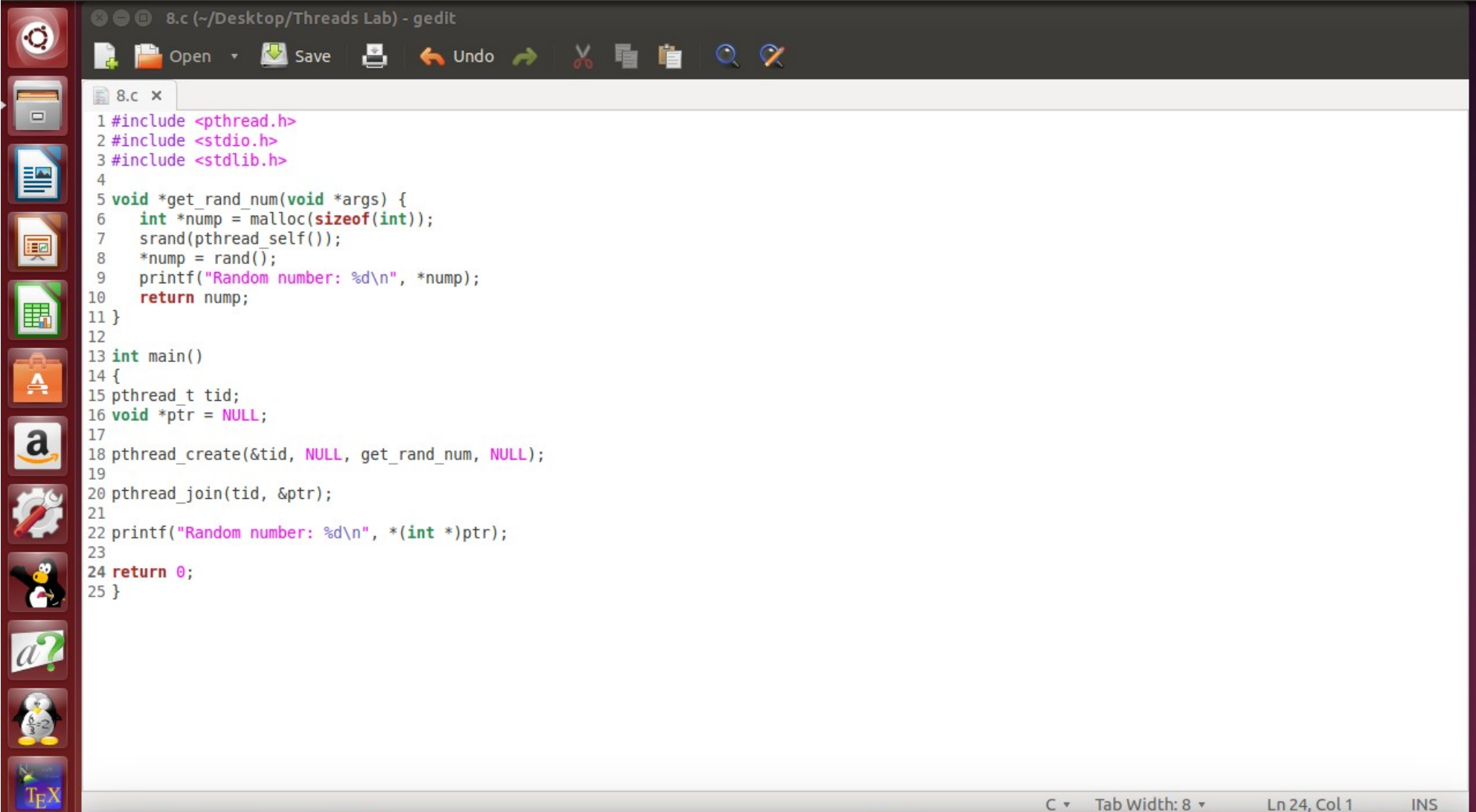
```
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ gcc -pthread 2_helloworld.c -o 2_helloworld.o
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./2_helloworld.o
Hello from the main thread
Hello, world!
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$
```

Return value example

```
void *get_rand_num(void *args) {
    int *nump = malloc(sizeof(int));
    srand(pthread_self());
    *nump = rand();
    printf("Random number: %d\n", *nump);
    return nump;
}

int main() {
    pthread_t tid;
    void *ptr = NULL;

    pthread_create(&tid, NULL, get_rand_num, NULL);
    pthread_join(tid, &ptr);
    printf("Random number: %d\n", *(int *)ptr);
    return 0;
}
```



8.c (~/.Desktop/Threads Lab) - gedit

Open Save Undo

8.c x

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 void *get_rand_num(void *args) {
6     int *nump = malloc(sizeof(int));
7     srand(pthread_self());
8     *nump = rand();
9     printf("Random number: %d\n", *nump);
10    return nump;
11 }
12
13 int main()
14 {
15     pthread_t tid;
16     void *ptr = NULL;
17
18     pthread_create(&tid, NULL, get_rand_num, NULL);
19
20     pthread_join(tid, &ptr);
21
22     printf("Random number: %d\n", *(int *)ptr);
23
24     return 0;
25 }
```

C Tab Width: 8 Ln 24, Col 1 INS

jayprakash@jayprakash-System-AsusPG500: ~/.Desktop/Threads Lab

```
jayprakash@jayprakash-System-AsusPG500:~/.Desktop/Threads Lab$ ./8.o
Random number: 1686197976
Random number: 1686197976
jayprakash@jayprakash-System-AsusPG500:~/.Desktop/Threads Lab$ ./8.o
Random number: 1846943549
Random number: 1846943549
jayprakash@jayprakash-System-AsusPG500:~/.Desktop/Threads Lab$ ./8.o
Random number: 1831534922
Random number: 1831534922
jayprakash@jayprakash-System-AsusPG500:~/.Desktop/Threads Lab$ ./8.o
Random number: 384451901
Random number: 384451901
jayprakash@jayprakash-System-AsusPG500:~/.Desktop/Threads Lab$
```

Notes

- Our previous example had the thread just return from function **thread**
 - In this case the call to **pthread_exit()** is implicit
 - The return value of the function serves as the argument to the (implicitly called) **pthread_exit()**

- What is the output?

```
void *thread (void *vargp) {  
    int arg = *((int*)vargp);  
    return &arg;  
}  
  
int main () {  
    pthread_t tid;  
    int thread_arg = 0xDEADBEEF;  
    int *ret_value;  
    pthread_create(&tid, NULL, thread, &thread_arg);  
    pthread_join(tid, (void **)(&ret_value));  
    printf("%X\n", *ret_value);  
    return 0;  
}
```


pthread_exit vs exit

- pthread_exit
 - Terminates the calling thread.
 - Does not terminate the entire process/other threads.
- exit
 - Terminates the entire process
 - All threads are gone
- The main thread is a little special
 - pthread_exit() waits for all threads in the process to terminate

Different between using exit(), return or pthread_exit()

- exit() – exit the complete process so if you call from one of the thread function, it will end the process without waiting for main or any other thread in the process to continue → **process ends**

- [thread_example1.c](#)

```
$ ./thread_example1.out
```

```
Main thread 521172736 is starting
```

```
child thread id 513017600 is starting
```

```
child thread id 513017600 is calling exit
```

```
$
```

Note: Message “Main thread 521172736 is finished” is never displayed because parent also have terminated when child called exit()

thread_example1.c (~/Desktop/Threads Lab) - gedit

Open Save Undo

thread_example1.c x

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 void * thr_fn(void *arg)
7 {
8     printf("child thread id %u is starting \n", (unsigned int)pthread_self());
9     printf("child thread id %u is calling exit\n", (unsigned int)pthread_self());
10    exit(0);
11 }
12 int main(void)
13 {
14     pthread_t tid;
15     printf("Main thread %u is starting\n", (unsigned int)pthread_self());
16     int err = pthread_create(&tid, NULL, thr_fn, NULL);
17     sleep(1); // ensure tid thread calls exit before main thread is completed
18
19     /* If thread executes exit(), process terminates and the following printf() will not get executed */
20
21     printf("Main thread %u is finished\n", (unsigned int)pthread_self());
22     exit(0);
23 }
```

C Tab Width: 8 Ln 8, Col 83 INS

jayprakash@jayprakash-System-AsusPG500: ~/Desktop/Threads Lab

```
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ gcc -pthread thread_example1.c -o thread_example1.o
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./thread_example1.o
Main thread 8804160 is starting
child thread id 501504 is starting
child thread id 501504 is calling exit
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$
```

Different between using exit(), return or pthread_exit()

- return() – when called from a thread then it will not wait for any of child thread to complete → **parent and all the child threads ends**

[thread_example2.c](#)

```
$ ./thread_example2.out
```


```
Main thread 1203812096 is starting
```

```
child thread id 1195656960 is starting
```

```
Main thread 1203812096 is finished
```

```
$
```

Note: Message “child thread id 1195656960 is calling exit” is never displayed because child is terminated by parent calling return



thread_example2.c (~/Desktop/Threads Lab) - gedit

Open Save Undo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 void * thr_fn(void *arg)
7 {
8     printf("child thread id %u is starting \n", (unsigned int)pthread_self());
9     sleep(1); // ensure main thread exit first
10    printf("child thread id %u is calling exit\n", (unsigned int)pthread_self());
11    exit(0);
12 }
13 int main(void)
14 {
15     pthread_t tid;
16     printf("Main thread %u is starting\n", (unsigned int)pthread_self());
17     int err = pthread_create(&tid, NULL, thr_fn, NULL);
18     sleep(1); // just enough wait for child thread to start
19     printf("Main thread %u is finished\n", (unsigned int)pthread_self());
20     return 0;
21 }
```

C Tab Width: 8 Ln 18, Col 16 INS

jayprakash@jayprakash-System-AsusPG500: ~/Desktop/Threads Lab

```
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./thread_example2.o
Main thread 1144674112 is starting
child thread id 1136371456 is starting
Main thread 1144674112 is finished
child thread id 1136371456 is calling exit
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$ ./thread_example2.o
Main thread 1277458240 is starting
child thread id 1269155584 is starting
Main thread 1277458240 is finished
jayprakash@jayprakash-System-AsusPG500:~/Desktop/Threads Lab$
```

Different between using exit(), return or pthread_exit()

- pthread_exit() – when called from a thread then it will only end that thread but all other threads in the same process or child threads can continue → **this thread ends but all the child threads or any other threads in the same process continues**

[thread_example3.c](#)

```
$ ./thread_example3.out
```

```
Main thread 1709131520 is starting
```

```
child thread id 1700976384 is starting
```

```
Main thread 1709131520 is finished but let the child thread continue
```

```
child thread id 1700976384 is calling exit
```

```
$
```

Note: pthread_exit() is called by parent/main thread hence it has terminated but child thread continues to execute until it calls exit()

thread_example3.c (~/.Desktop/Threads Lab) - gedit

Open Save Undo

thread_example3.c x

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 void * thr_fn(void *arg)
7 {
8     printf("child thread id %u is starting \n", (unsigned int)pthread_self());
9     sleep(1); // ensure main thread exit first
10    printf("child thread id %u is calling exit\n", (unsigned int)pthread_self());
11    exit(0);
12 }
13 int main(void)
14 {
15     pthread_t tid;
16     printf("Main thread %u is starting\n", (unsigned int)pthread_self());
17     int err = pthread_create(&tid, NULL, thr_fn, NULL);
18     sleep(1); // just enough wait for child thread to start
19     printf("Main thread %u is finished but let the child thread continue\n", (unsigned int)pthread_self());
20     pthread_exit(0);
21 }
```

C Tab Width: 8 Ln 18, Col 16 INS

jayprakash@jayprakash-System-AsusPG500: ~/.Desktop/Threads Lab

jayprakash@jayprakash-System-AsusPG500:~/.Desktop/Threads Lab\$ gcc -pthread thread_example3.c -o thread_example3.o

jayprakash@jayprakash-System-AsusPG500:~/.Desktop/Threads Lab\$./thread_example3.o

Main thread 3977480000 is starting

child thread id 3969177344 is starting

Main thread 3977480000 is finished but let the child thread continue

child thread id 3969177344 is calling exit

jayprakash@jayprakash-System-AsusPG500:~/.Desktop/Threads Lab\$