Operating System

Multi-threading (Pthread library)

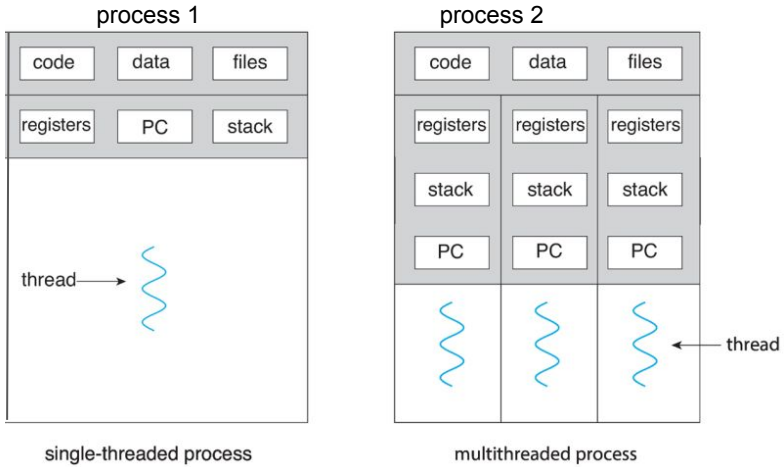
mojtaba nafez

Iran University of Science and Technology

Spring 2021

Thread:

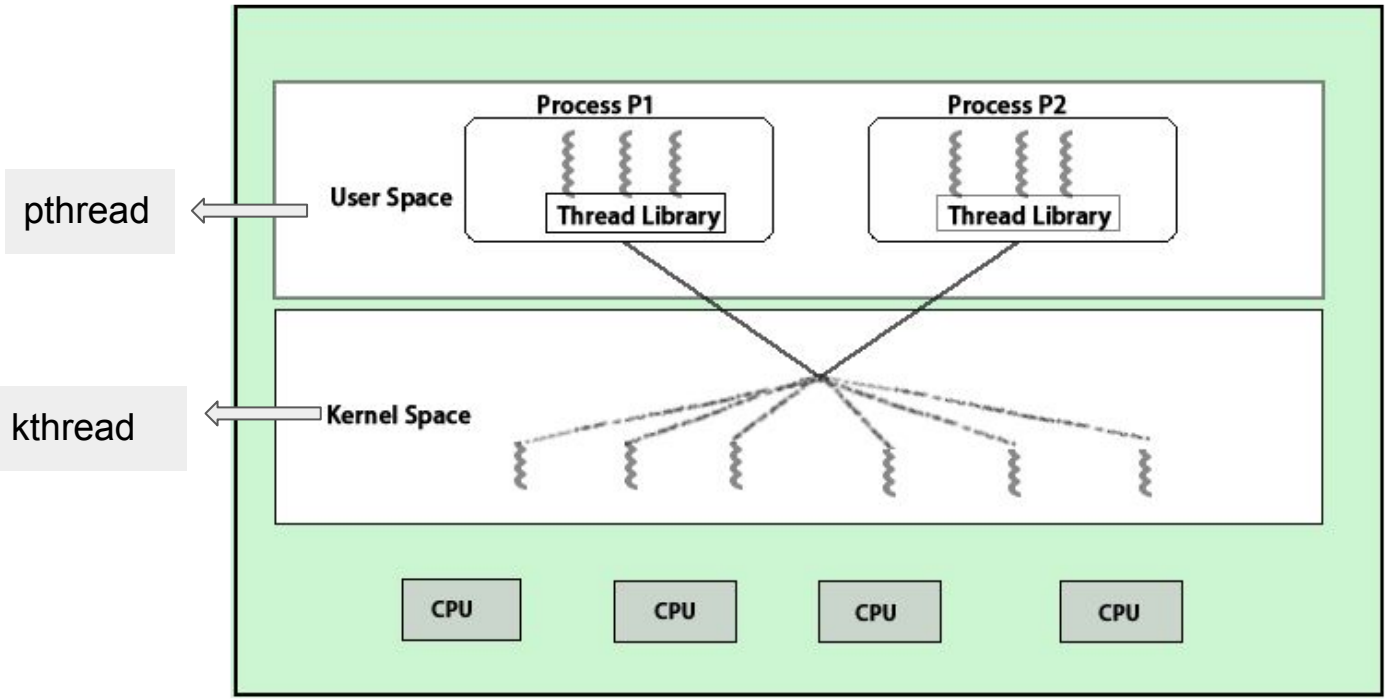
- A ***thread*** is a basic unit of CPU utilization. (basic than process)
- Threads used for **Parallelism.**
- It is most effective on multi-processor or multi-core systems



Thread vs Process:

- Thread is an execution unit that is part of a process. A process can have **multiple threads**.
- A Process is mostly isolated, whereas Threads **share memory.**
  (Process does not share data, and Threads share data with each other.)
- Process likely takes more time for **context switching** whereas as Threads takes less time for context Switching.
- A Process is not **Lightweight**, whereas Threads are Lightweight.

Thread vs Process:



Thread vs Process:

Process	Thread
Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Thread && Process in Linux:

- **Process in Linux:**

  ps - report a snapshot of the current processes

  Command: ps -A -l:

- **Thread in Linux:**

  ps -T -p “p_id” -l

  Command: ps -T -p 200 -l

  ps -A -T

  (The **SPID** column is the thread id)

- **Graphical System monitoring:**

- **System monitoring in terminal:**

  Command: top

POSIX Thread (pthread) libraries:

- **Historically**, hardware vendors have implemented their own proprietary versions of threads.
- A standardized programming interface was required.
- For UNIX systems, this interface has been specified by the **IEEE POSIX 1003.1c standard** (1995) (Portable Operating System Interface)
- Most hardware vendors now offer Pthreads in addition to their proprietary threads.
- On Linux pthread uses the **clone syscall** with a special flag CLONE_THREAD.
- Pthread is library **not syscall**.

POSIX Thread (pthread) libraries:

The subroutines which comprise the Pthreads API can be informally grouped into four major groups:

- **Thread management**: Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)
- **Mutexes**: Routines that deal with synchronization, called a “mutex”, which is an abbreviation for “mutual exclusion”. Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
- **Condition variables**: Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
- **Synchronization**: Routines that manage read/write locks and barriers.

POSIX Thread (pthread) libraries:

4.7.2 Linux Threads

- Linux does not distinguish between processes and threads - It uses the more generic term "tasks".
- The traditional fork( ) system call completely duplicates a process (task), as described earlier.
- An alternative system call, clone( ) allows for varying degrees of sharing between the parent and child tasks, controlled by flags such as those shown in the following table:

flag	Meaning
CLONE_FS	File-system information is shared
CLONE_VM	The same memory space is shared
CLONE_SIGHAND	Signal handlers are shared
CLONE_FILES	The set of open files is shared

- Calling clone( )with no flags set is equivalent to fork( ). Calling clone( ) with CLONE_FS, CLONE_VM, CLONE_SIGHAND, and CLONE_FILES is equivalent to creating a thread, as all of these data structures will be shared.
- Linux implements this using a structure **`task_struct`**, which essentially provides a level of indirection to task resources. When the flags are not set, then the resources pointed to by the structure are copied, but if the flags are set, then only the pointers to the resources are copied, and hence the resources are shared. (Think of a deep copy versus a shallow copy in OO programming.)
- **(Removed from 9th edition)** Several distributions of Linux now support the NPTL (Native POXIS Thread Library)
  - POSIX compliant.
  - Support for SMP (symmetric multiprocessing), NUMA (non-uniform memory access), and multicore processors.
  - Support for hundreds to thousands of threads.

➢ On Linux pthread uses the **clone syscall** with a special flag CLONE_THREAD.

POSIX Thread (pthread) libraries:

4.7.1 Windows XP Threads

- The Win32 API thread library supports the one-to-one thread model
- Win32 also provides the ***fiber*** library, which supports the many-to-many model.
- Win32 thread components include:
  - Thread ID
  - Registers
  - A user stack used in user mode, and a kernel stack used in kernel mode.
  - A private storage area used by various run-time libraries and dynamic link libraries (DLLs).
- The key data structures for Windows threads are the ETHREAD (executive thread block), KTHREAD (kernel thread block), and the TEB (thread environment block). The ETHREAD and KTHREAD structures exist entirely within kernel space, and hence are only accessible by the kernel, whereas the TEB lies within user space, as illustrated in Figure 4.10:

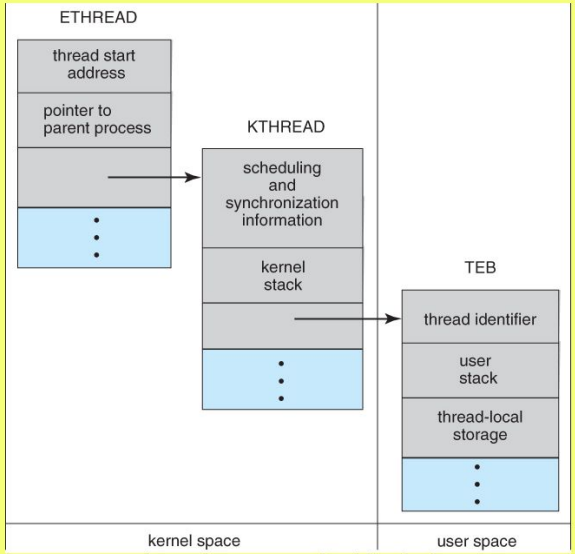
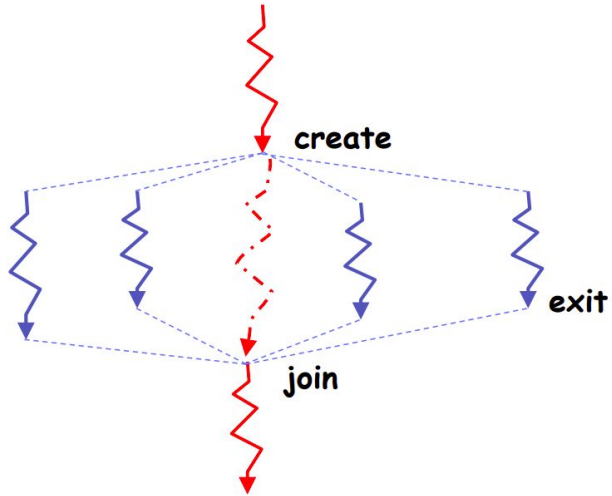


Figure 4.14 - Data structures of a Windows thread

Multi-Thread:

- Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.
- Pthread_thread_t
- pthread_create()
- pthread_join()
- pthread_exit()
- Install Pthread manpages:

  sudo apt-get install manpages-posix manpages-posix-dev



Multi-Thread: pthread_create

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

```
Arguments:
   thread - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h) => thread_id written in pthread* thread
   attr - Set to NULL if default thread attributes are used. (else define members of the struct pthread_attr_t defined in bits/pthreadtypes.h) Attributes
          include:
                  detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
                  scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
                  scheduling parameter
                  inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
                  scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
                  guard size
                  stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
                  stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),
   void * (*start_routine) - pointer to the function to be threaded. Function has a single argument: pointer to void.
   *arg - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.
```

Multi-Thread: pthread_exit

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

```
Terminate calling thread.
Arguments:
   retval - Return value of thread.
```

```
int pthread_join(pthread_t thread, void **value_ptr);
```

```
wait for termination of another thread.
Arguments:       thread_t:     thread that we wait for it
                 **value_ptr:  return value of thread thread that we wait for it
```

example 1:

```
void *MyThreadFunc(void *args){
   printf("from thread:: thread is running.....\n from thread:: function args: %s\n",(char *)args);
   srand(time(NULL));
   int rnd=rand()%6;
   sleep(rnd);
   printf("from thread:: thread execution lasts %d second(s)\n",rnd);
   pthread_exit("goodby world");
}
void main(){
   pthread_t thread_id;
   void *thread_result;
   if(pthread_create(&thread_id,NULL,MyThreadFunc,"Hello World")==0){
        printf("from main:: waiting for thread %ld  .....\n\n",thread_id);
        if(pthread_join(thread_id,&thread_result)==0){
            printf("\n from main:: thread return value: %s\n",(char *)thread_result);
        }
    }
}
```

Example 2: Creating New Threads

```c
#include <pthread.h>
#define count_threads 10
typedef struct { int wid; } t_arg;
void *worker_func(void *_arg) { // implementation of the thread (in next slide) … }
int main(int argc, char *argv[]) {
  // defining some variables
  pthread_t threads[count_threads];
  t_arg args[count_threads];
  for (int i = 0; i < count_threads; i++) {
       args[i].wid = i;
       pthread_create(& threads[i], NULL,
                      worker_func, (void *)&args[i]);
  }
  for (int i = 0; i < count_threads; i++)
      pthread_join(threads[i], NULL);
  return 0;
}
```

Example 2: Critical Region

```c
#include <pthread.h>
#define count_threads 10
typedef struct { int wid; } t_arg;

int tail = 0;
int arr[count_threads];

void *worker_func(void *_arg) {
  t_arg* arg = (t_arg *)_arg;
  arr[tail] = arg->wid;
  printf("wid: %d\n", arg->wid);
  tail++;
  pthread_exit(NULL);
}
```

Example 2: Critical Region

```c
#include <pthread.h>
#define count_threads 10
typedef struct { int wid; } t_arg;

int tail = 0;
int arr[count_threads];

void *worker_func(void *_arg) {
  t_arg* arg = (t_arg *)_arg;
  arr[tail] = arg->wid;
  printf("wid: %d\n", arg->wid);
  tail++;
  pthread_exit(NULL);
}
```

Share resources

Critical Region:

Output of Our Example 2:

0: wid: 6	0: wid: 5	0: wid: 7	0: wid: 6
1: wid: 8	1: wid: 0	1: wid: 8	1: wid: 7
2: wid: 9	2: wid: 0	2: wid: 9	2: wid: 8
3: wid: 5	3: wid: 0	3: wid: 0	3: wid: 9
4: wid: 4	4: wid: 0	4: wid: 0	4: wid: 5
5: wid: 0	5: wid: 0	5: wid: 0	5: wid: 4
6: wid: 2	6: wid: 9	6: wid: 0	6: wid: 3
7: wid: 0	7: wid: 6	7: wid: 0	7: wid: 2
8: wid: 0	8: wid: 0	8: wid: 0	8: wid: 1
9: wid: 1	9: wid: 0	9: wid: 0	9: wid: 0

Result of running previous program four times

Questions?

?

END