THREADS

REFERENCES:

1. "OPERATING SYSTEM CONCEPTS" 9TH EDITION BY ABRAHAM SILBERSCHATZ, PETER BAER GALVIN AND GREG GAGNE
2. "OPERATING SYSTEMS: INTERNALS AND DESIGN PRINCIPLES", 7TH EDITION BY WILLIAM STALLINGS
3. "INTRODUCTION TO OPERATING SYSTEMS ", BY PROF. CHESTER REBEIRO (HTTPS://NPTEL.AC.IN/COURSES/106/106/106106144/)

CONSIDER THE FOLLOWING SCENARIO:

- There are 4 CPUs in the system
- The given program is adding numbers upto 10 million using

an addall() and executing the process in one CPU.

Problem: Other processors are not utilized and single process

takes long time to complete execution

```c
#include <stdio.h>


unsigned long addall(){
  int i=0;
  unsigned long sum=0;

  while (i< 10000000){
     sum += i;
     i++;
  }
  return sum;
}


int main()
{
   unsigned long sum;
   srandom(time(NULL));
   sum = addall();
   printf("%lu\n", sum);
```

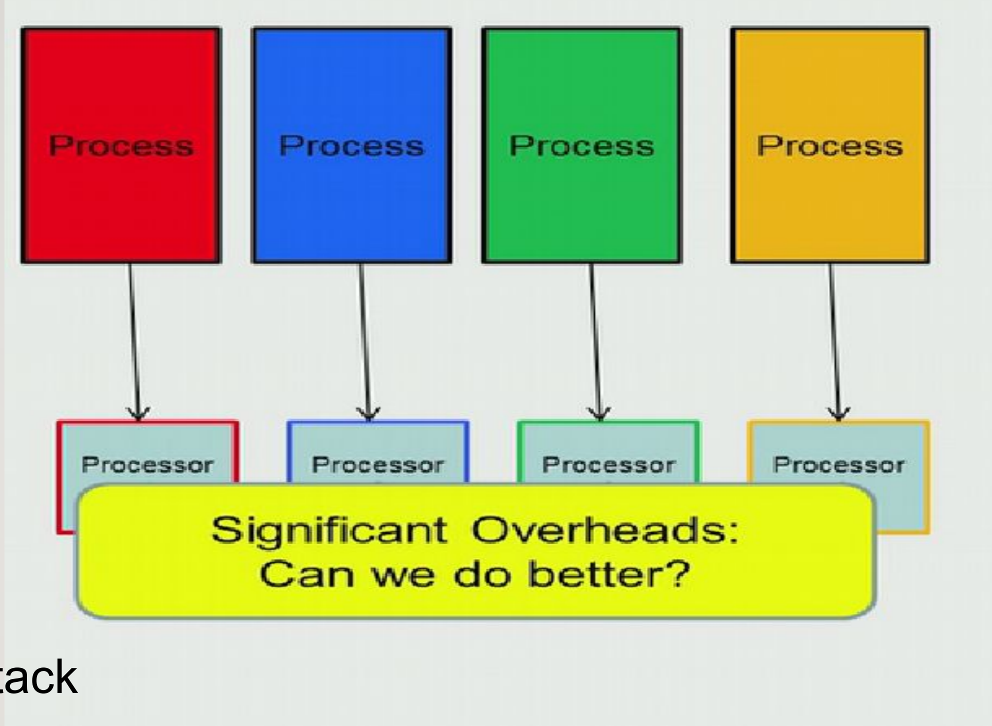
Ref. 3

What can be the better method to perform this summation?

- Create 4 processes such that each process adds 2.5 million no.s
- We need 4 fork() calls to create 4 processes
- Each process can execute in one processor
- Reduces the computation time,

But

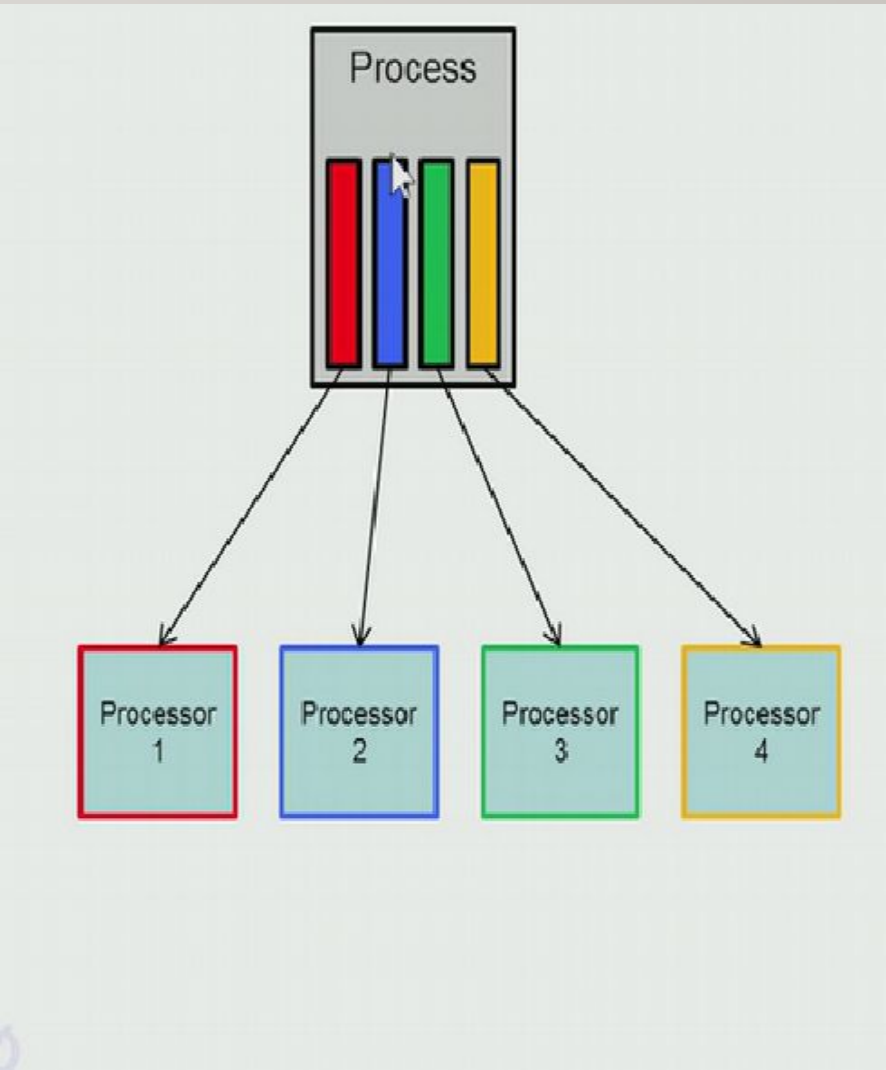
- Each process has its own set of instructions, data, heap and the stack
- A large portion of these 4 processes are similar.
- A lot of duplication of instructions and data
- Process Management and IPC required



Ref. 3

Any better method to reduce this overhead and achieve parallelization?

- Create 4 threads under 1 process, using Pthread.
- Each thread executes in separate processor
- Each thread shares common instructions, parameters and heap, etc.
- However, each thread has separate stack.
- Each thread will add 2.5 million no.s
- Threads are lighter than processes
- Very few or no system calls needed to create threads



Ref. 3

Threads

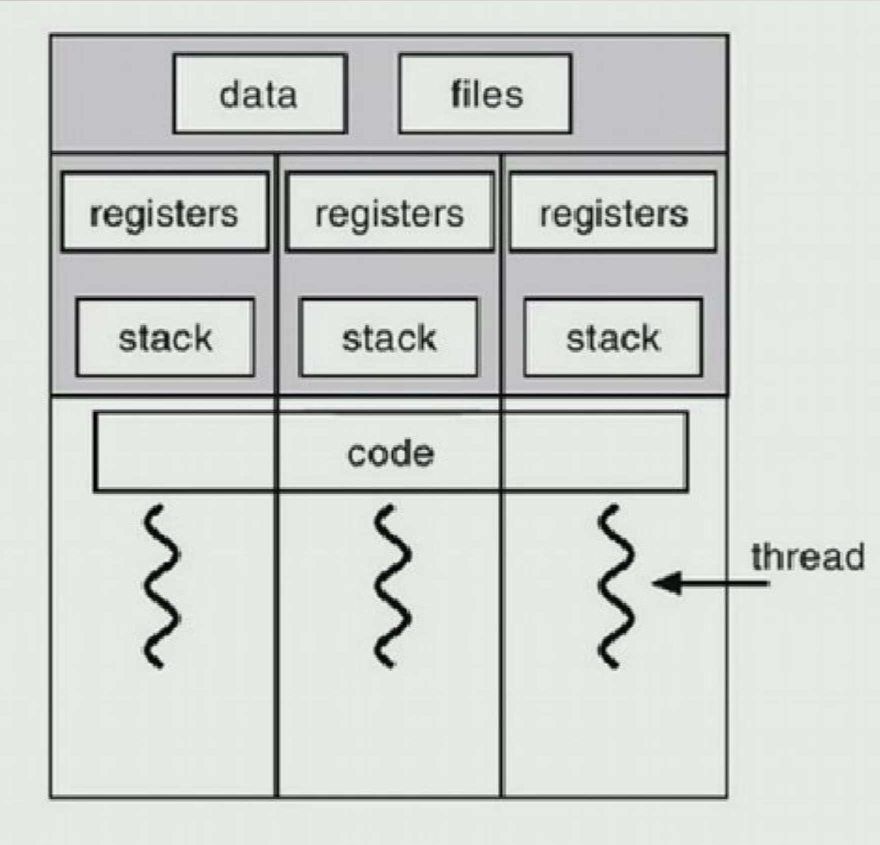
- Threads are separate streams of execution within a single process.
- They are not isolated from each other.
- The state of a thread is stored in Thread Control Block which contains registers and stack
- It provides mechanisms to perform multiple tasks concurrently.
- Each thread has got associated:

  Thread ID

  Program counter

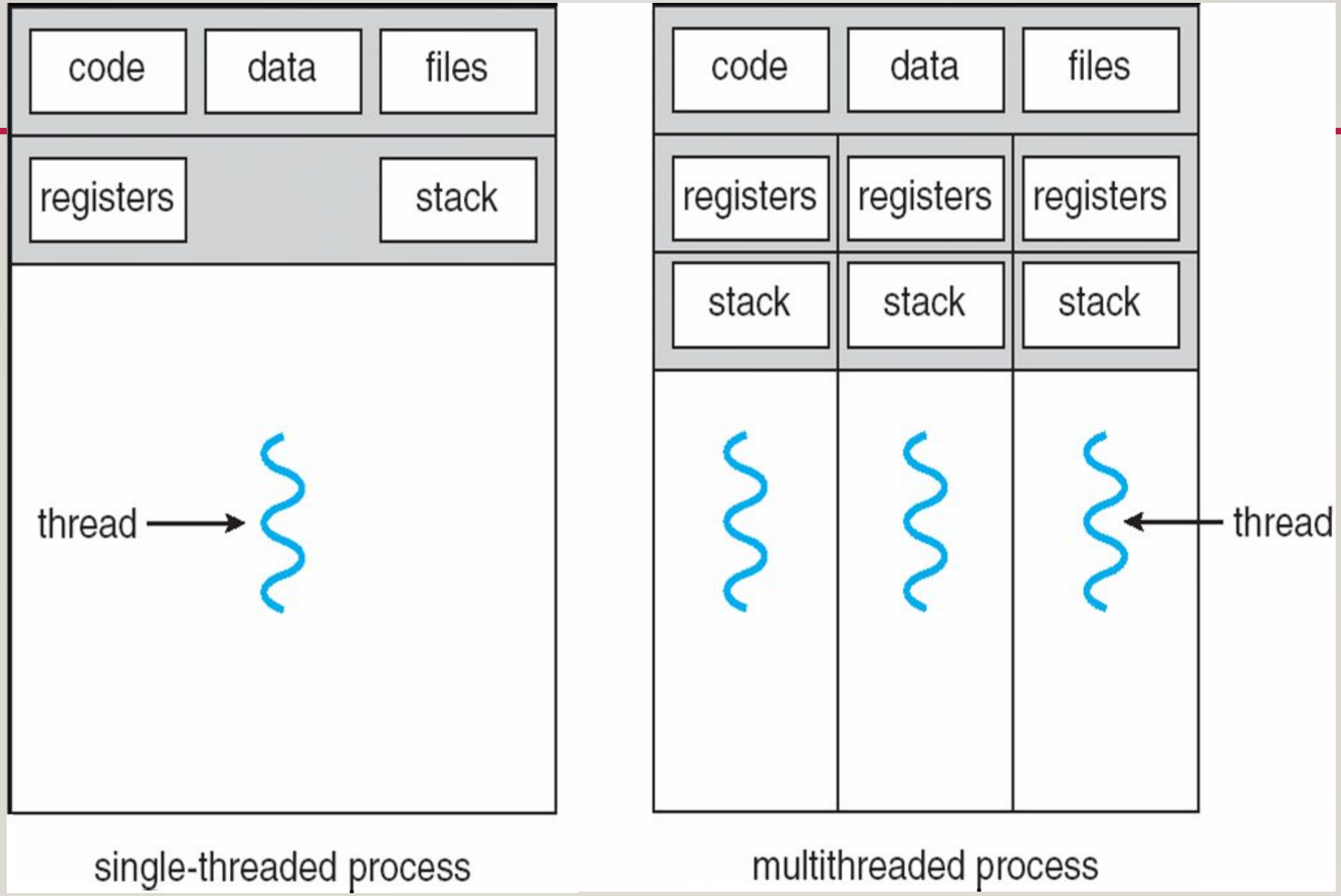
  Register set

  Stack



Ref. 1

SINGLE AND MULTITHREADED PROCESSES



heavyweight process **lightweight** process Ref. 1

Threads vs Processes

- A thread has no data segment or heap
- A thread cannot live on its own. It needs to be attached to a process
- There can be more than one thread in a process. Each thread has its own stack
- If a thread dies, its stack is reclaimed

- A process has code, heap, stack, other segments
- A process has at-least one thread.
- Threads within a process share the same code, files.
- If a process dies, all threads die.

Based on Junfeng Yang's lecture slides
http://www.cs.columbia.edu/~junfeng/13fa-w4118/lectures/l08-thread.pdf

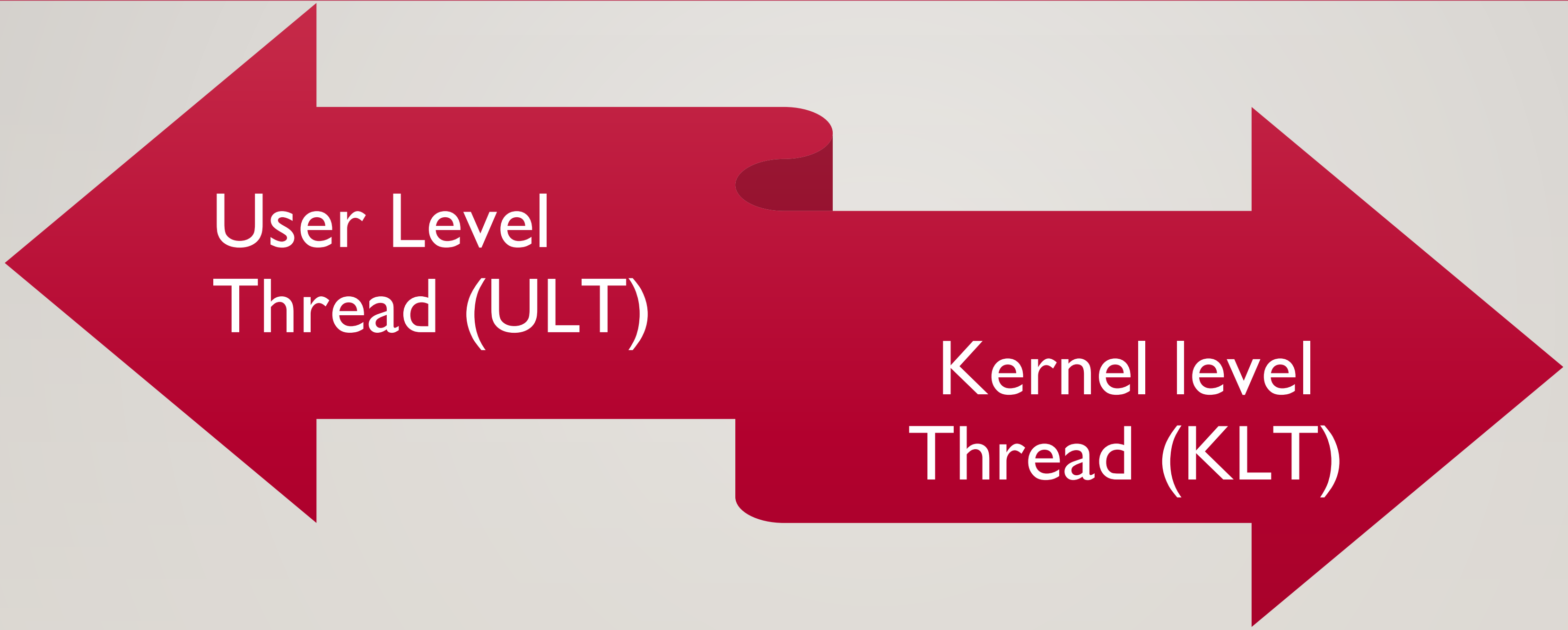
MERITS OF USING THREADS

- Threads can be created and destroyed quickly as compared to processes
- Applications can use threads to execute some functions in the background
- Threads can share the same address space.
- It takes less time to switch between threads due to smaller state record

THREADS SCHEDULING

- Threads are scheduled to execute in CPU independently
- State of each executing thread is maintained separately.
- If a process is suspended, then all its threads are suspended
- If a process is terminated, then all its threads are terminated
- A thread also has states like ready, running, waiting or blocked.

TYPES OF THREADS



NOTE: We are talking about threads for *user* processes. Both ULT & KLT execute in user mode. An OS may also have threads but that is not what we are discussing here.

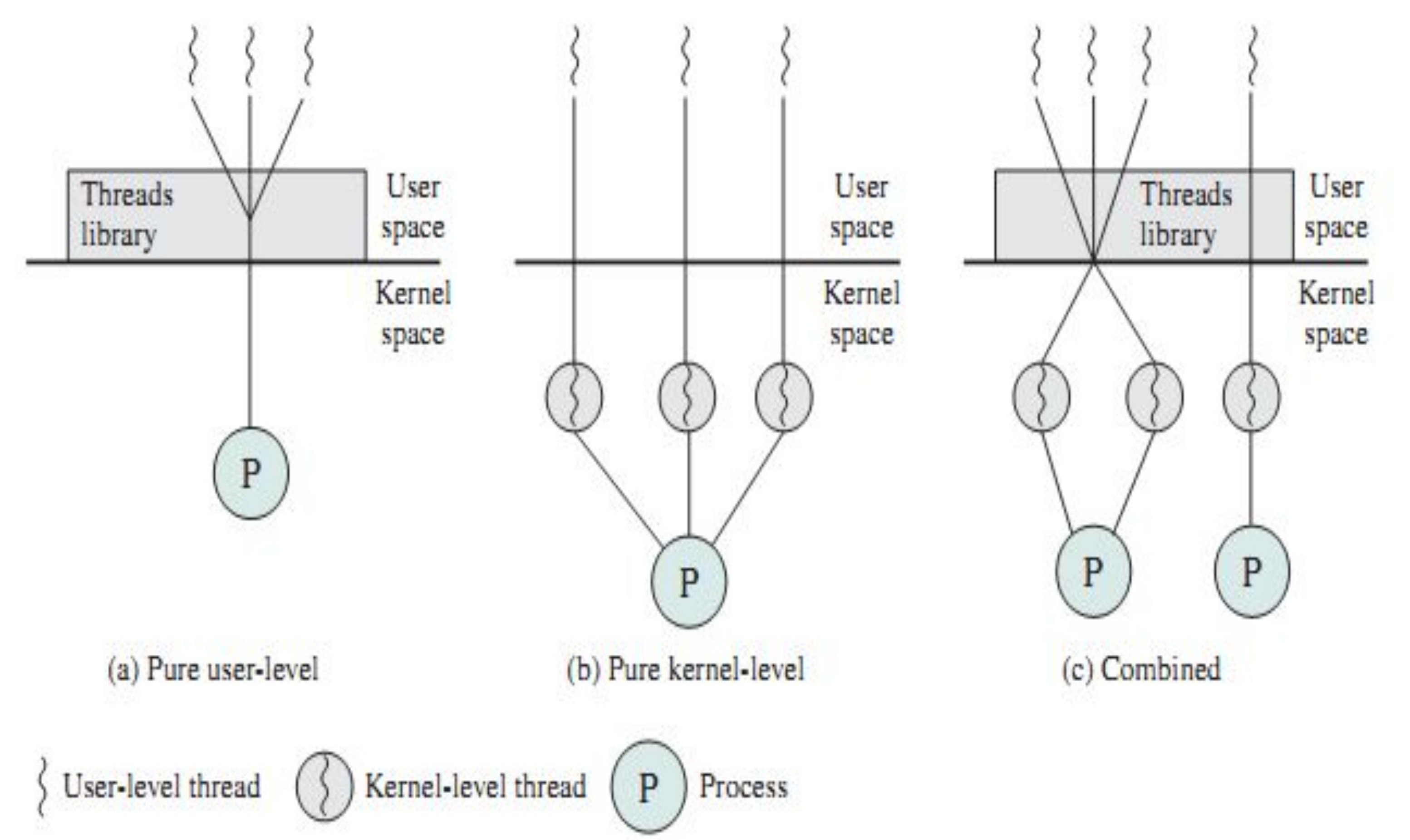
THREADS MANAGEMENT

User-Level Threads (ULTs)

- Managed by applications and user level thread library
- Kernel is not aware of these threads

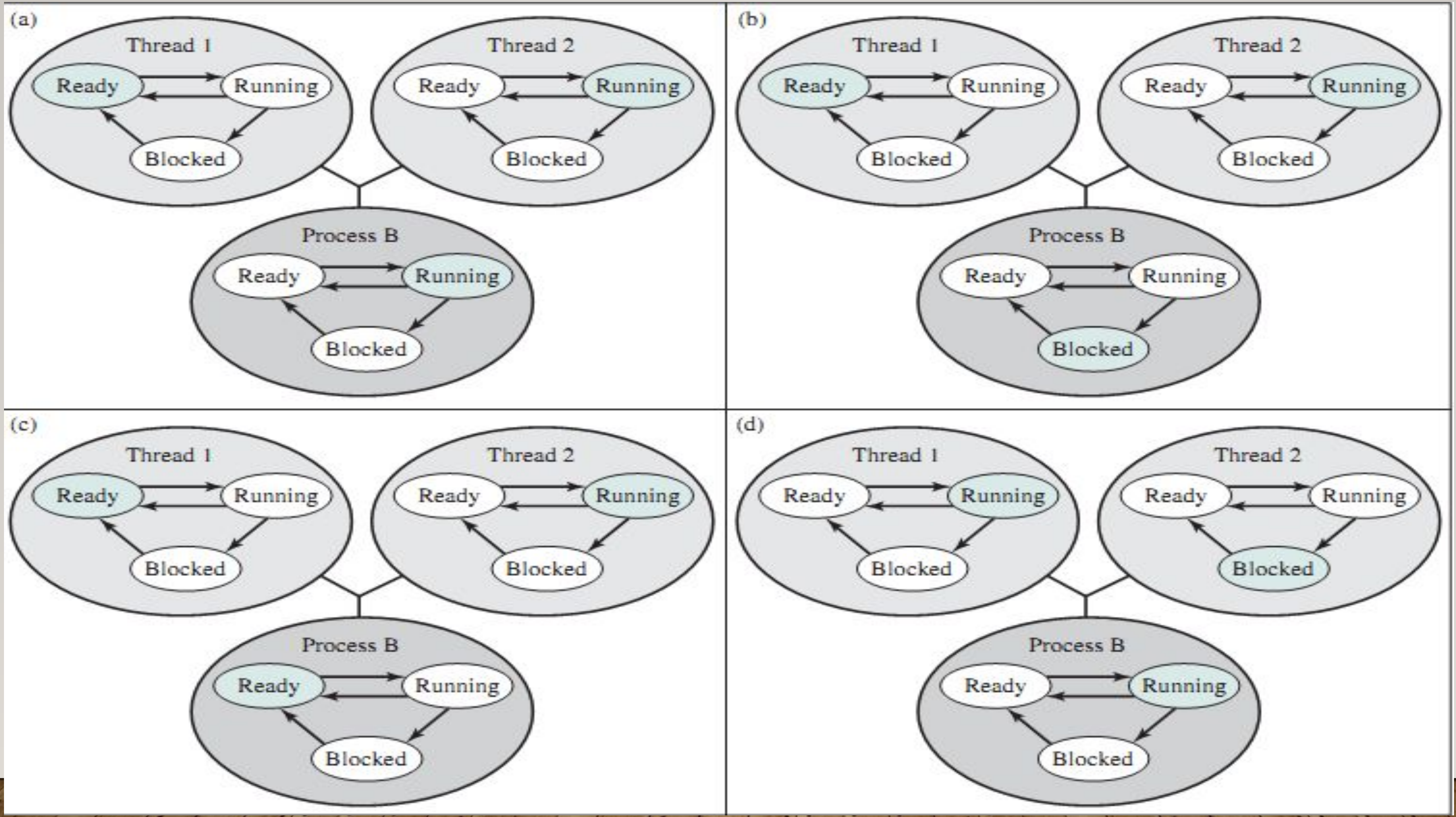
Kernel-Level Threads (KLTs)

- These are created and managed by kernels
- Also called as light weight process



Ref. 1

Relationships Between ULT States And Process States



Ref.2

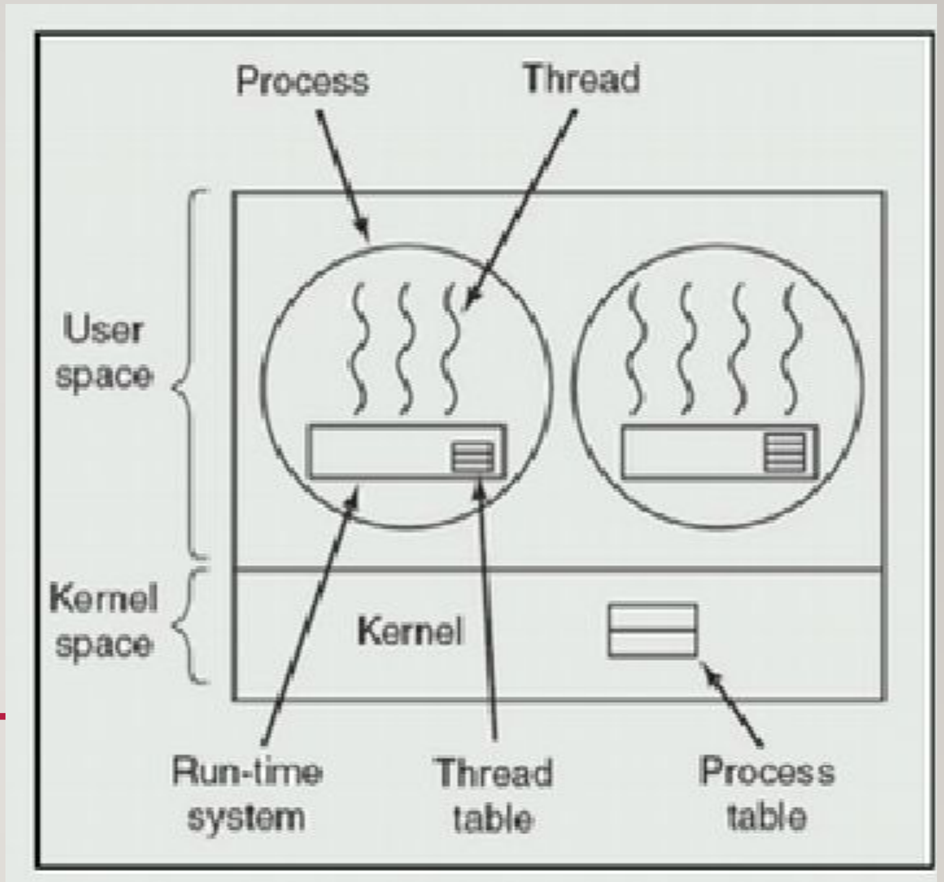
MERITS AND DEMERITS OF ULT

+ It can be implemented in OS that does not support threading
+ Fast creation and switching
+ Does not need System call

- Process with many threads also competes with a single threaded process
- Scheduling decisions cannot be made to favour processes with the larger number of threads
- If one thread makes system call then all others get blocked

Solution: JACKETING

converts a blocking system call into a non-blocking system call

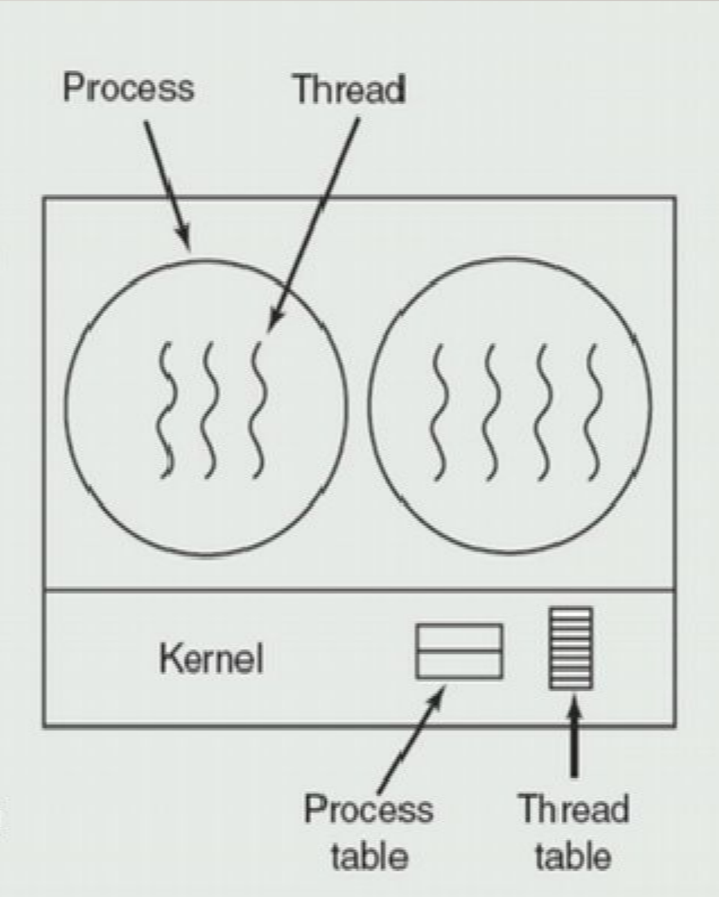


Ref.2

MERITS AND DEMERITS OF KLT

+ Thread table is stored in kernel space. Hence, kernel knows about no. of threads a process has
+ OS can provide more time quantum to a process with large no. of threads
+ Better to use for application that frequently blocks
+ One thread making system call does not block others

- Slow
- Larger overhead due to kernel level management
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel



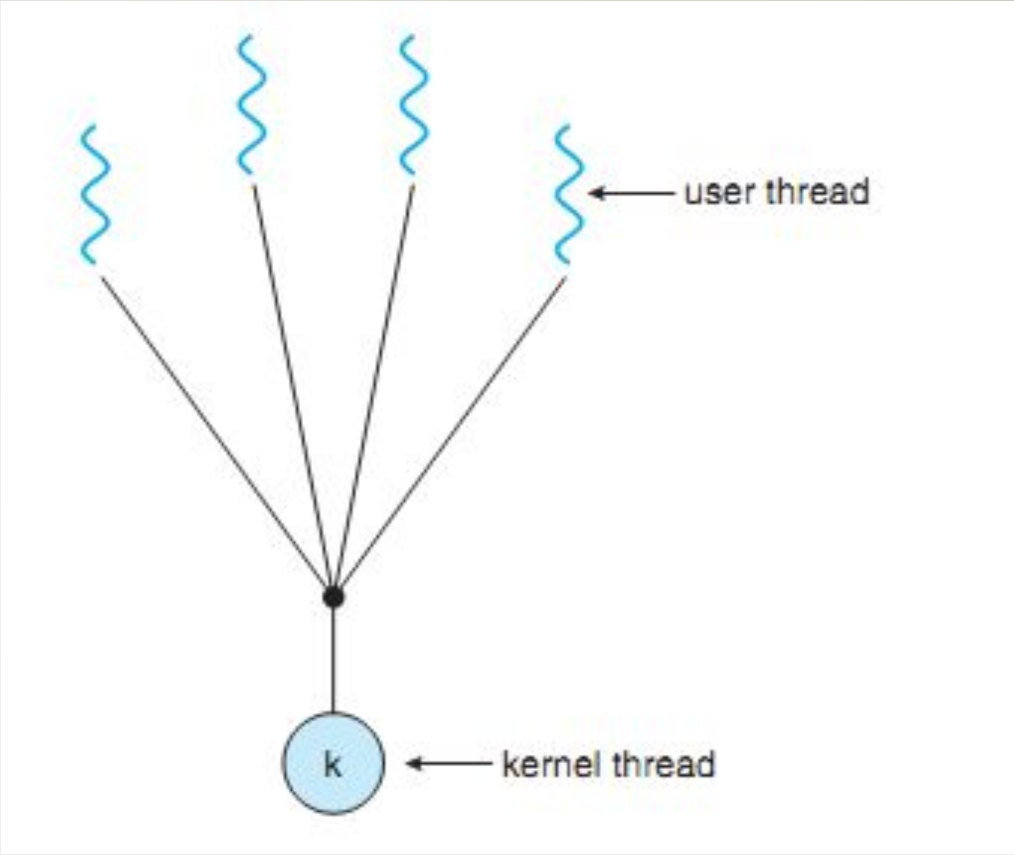
Ref.2

MULTITHREADING MODELS

- One-to-One
- Many-to-One
- Many-to-Many

MANY-TO-ONE

Many user-level threads mapped to single kernel thread

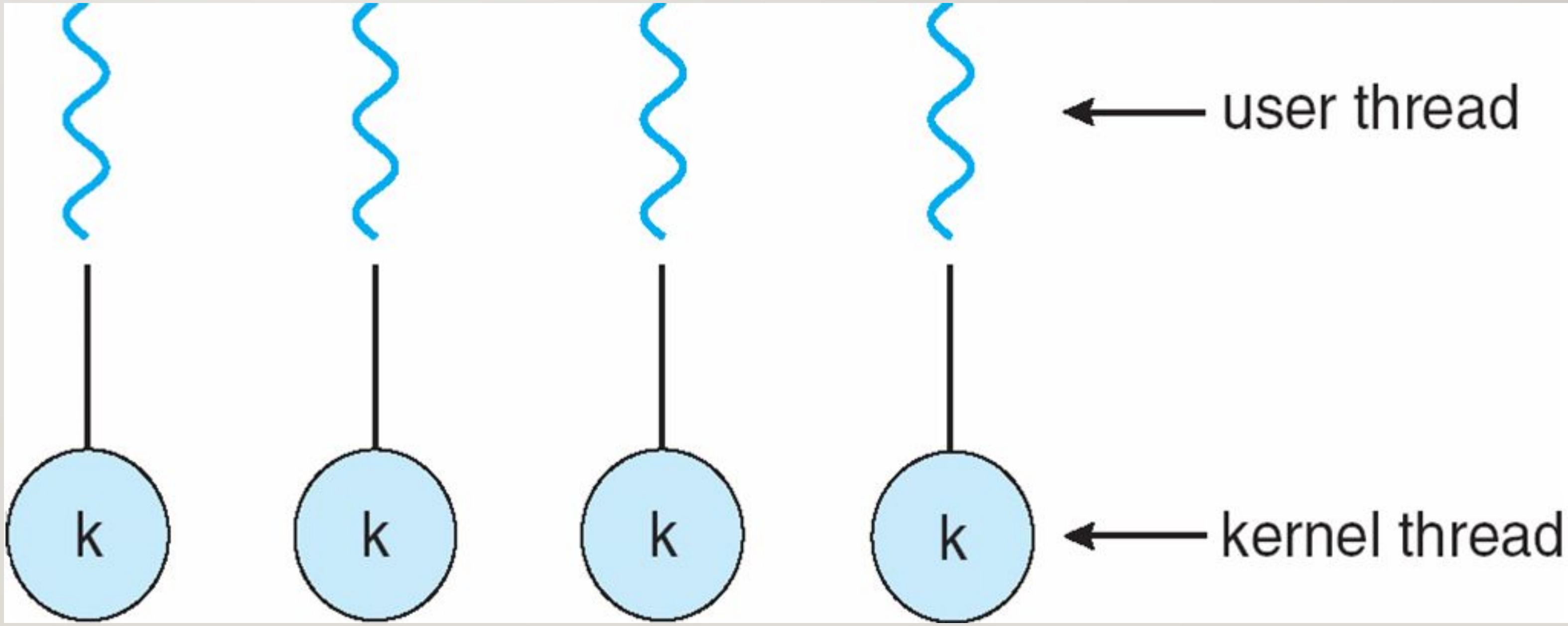


Ref.1

ONE-TO-ONE

Each user-level thread maps to kernel thread

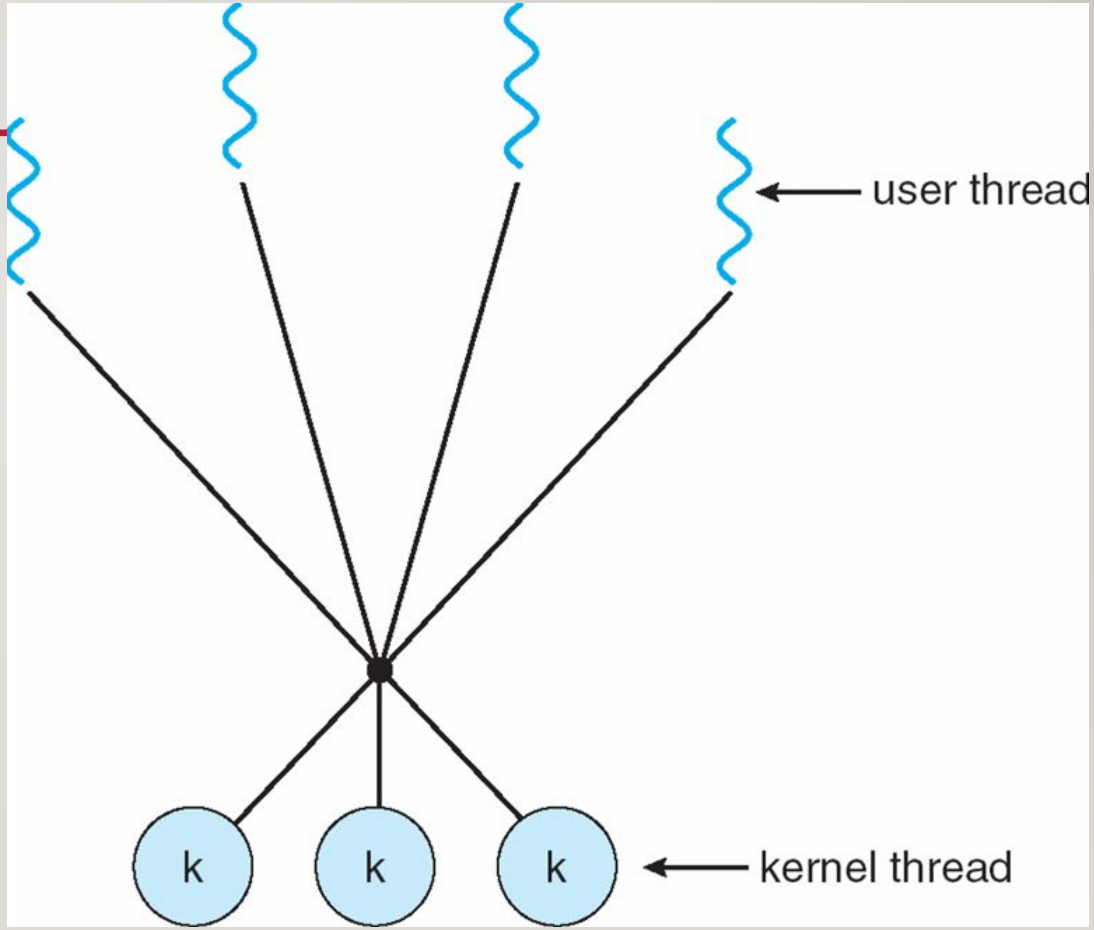
- Examples
  - Windows NT/XP/2000
  - Linux



Ref.1

MANY-TO-MANY MODEL

- Many user level threads are mapped to many kernel threads
- It allows the operating system to create a sufficient number of kernel threads
- Example
  - Windows NT/2000



Ref.1

THREAD LIBRARIES

- It provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

- Three main thread libraries in use today:
  - POSIC Pthreads
  - Win32
  - Java

POSIX

- It can be used over Linux systems
- Pthreads API must be compiled with ***–pthread*** or ***–lpthread***

```
#include <pthread.h>

pthread_t pthread_self()
```

returns : ID of current (this) thread

pthread library

- Create a thread in a process

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

Thread identifier (TID) much like

Pointer to a function, which starts execution in a different thread

Arguments to the function

- Destroying a thread

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

- Join : Wait for a specific thread to complete

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

TID of the thread to wait for

Exit status of the thread

Ref.3

Example

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

unsigned long sum[4];

void *thread_fn(void *arg){
  long id = (long) arg;
  int start = id * 2500000;
  int i=0;

  while(i < 2500000){
     sum[id] += (i + start);
     i++;
  }
  return NULL;
}

int main(){
  pthread_t t1, t2, t3, t4;

  pthread_create(&t1, NULL, thread_fn, (void *)0);
  pthread_create(&t2, NULL, thread_fn, (void *)1);
  pthread_create(&t3, NULL, thread_fn, (void *)2);
  pthread_create(&t4, NULL, thread_fn, (void *)3);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  pthread_join(t4, NULL);
  printf("%lu\n", sum[0] + sum[1] + sum[2] + sum[3]);
  return 0;
}
```

Note. You need to link the pthread library

```
$ gcc threads.c –lpthread
$ ./a.out
```

TERMINATING THREAD

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

```
void pthread_exit (return_value)
```

Threads terminate in one of the following conditions:

- Completes function execution and return value
- pthread_cancel() request received by thread
- Thread initiates termination
- The process of the threads terminates

THREAD CANCELLATION

- *pthread_cancel()* : Terminates a thread before it has completed its execution

Whether thread cancel or not depends in its state and type

States

  - PTHREAD_CANCEL_DISABLE: Thread can not be cancelled.
  - PTHREAD_CANCEL_ENABLE: This is default state. Thread can be cancelled

THREAD CANCELLATION

- Two types of thread cancellation
  - **Asynchronous cancellation:** terminates the target thread immediately
    - PTHREAD_CANCEL_ASYNCHRONOUS
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - PTHREAD_CANCEL_DEFERRED: Cancel when thread reaches ‘cancellation point’

THREADING ISSUES

- Use of **fork()** and **exec()** system calls
- Signal handling
- Thread pools
- Thread safety
- Thread-specific data

USE OF *FORK() ,EXEC(), EXIT()*

- **Does fork() duplicate only the calling thread or all threads?**
- Few unix OS keep two version of fork to have both the options.
- E***xec():*** the program specified in the parameter to exec() will replace the entire process—including all threads
- **Recommendation: In a process of multiple threads use *fork() only* after *exec()***

SIGNAL HANDLING

- Signals are used to notify about events to a process.
- A signal handler is used to process signals in the following way:
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Signal delivery options:
  - To the intended thread
  - To every thread in the intended process
  - To certain threads in the process
  - Assign a specific thread to receive all signals for the process

THREAD POOLS

- Create and maintain a number of threads in a pool
- Assign work to the threads as per the need
- Faster method to handle a request using an existing thread instead of creating a new one
- It bounds the number of threads in the application(s) to the size of the pool

THREAD SAFETY

A function is called *thread-safe* when it can be called by multiple threads at the same time without creating any disruptions.

Example of a function i.e. not safe:

```
static int glob = 0;
static void Incr (int  loops)
{ int loc, j;
  for (j = 0; j<loops; j++ {
       loc = glob;
       loc++;
       glob = loc;}
}
```

Employs global or static values that are shared by all threads

HOW TO ENSURE THREAD SAFETY?

- Serialize the function: Keep the critical section of the code locked so that only one thread access it at a time other threads out
- Use only thread safe system functions
- Avoid use of global and static variables

THREAD SPECIFIC DATA

- Makes existing functions thread-safe .
  - May be slightly less efficient than being reentrant
- Allows each thread to have its own copy of data
  - Provides per-thread storage for a function
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

THREADS: Pros and Cons

- Advantages of multithreading
  - Easy to share resources and faster to create
- Disadvantages of multithreading
  - Compete for acquiring memory
  - Ensure threads-safety
  - Error in one can disrupt the execution of other threads due to sharing of resources
- Considerations for future design
  - Handling signals is tricky
  - All threads must run the same program

OPERATING SYSTEM EXAMPLES

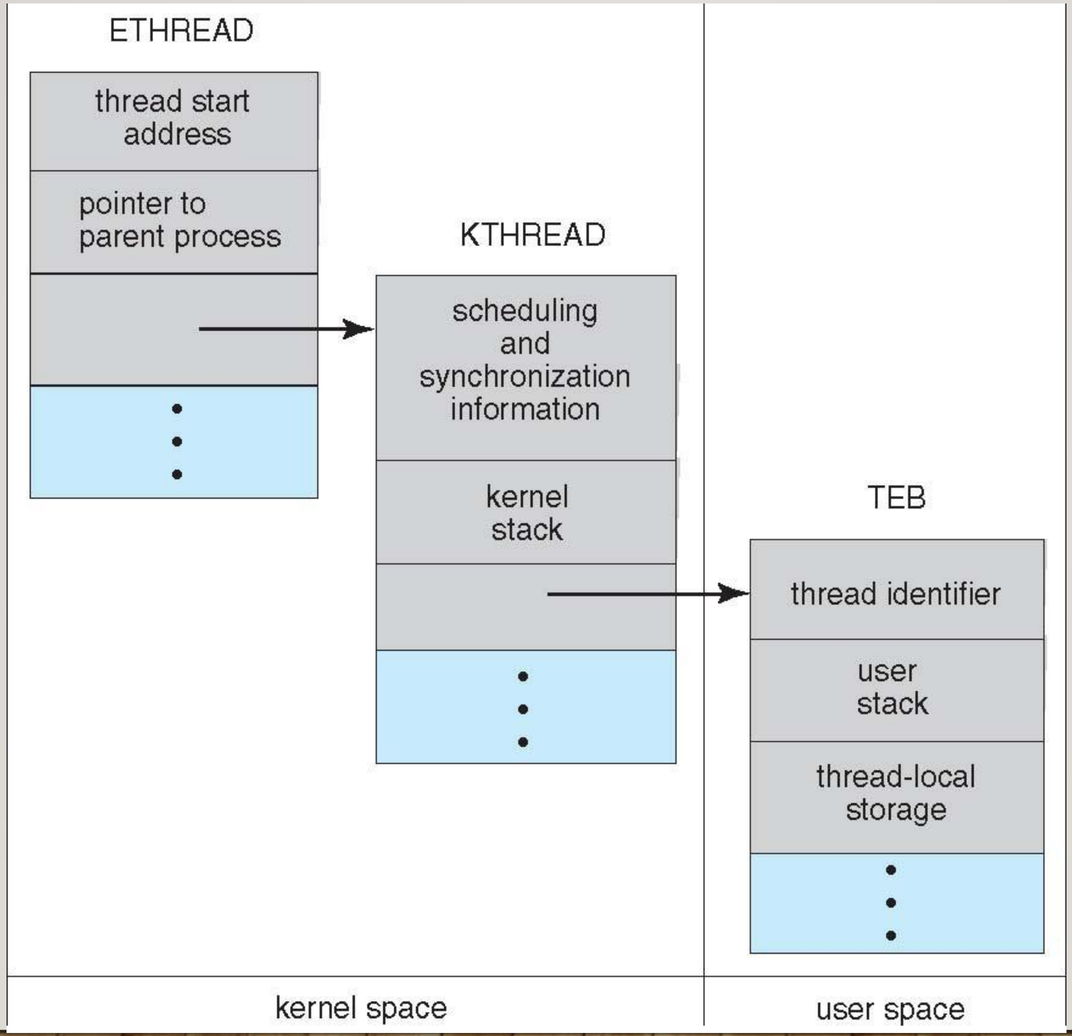
- Windows XP Threads

- Linux Thread

WINDOWS XP THREADS

- It implements one-to-one mapping of threads with kernel-level
- Each thread contains
  - A unique thread id
  - Set of Registers
  - Separate user and kernel stacks
  - Private data storage area
- These are called context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

WINDOWS XP THREADS



LINUX THREADS

- Threads are referred as Tasks in Linux
- Tasks are created using **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

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.