

Operating Systems: Threads and Concurrency

Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

Acknowledgements: Material based on the textbook Operating Systems Concepts (Chapter 4)

Threads

- A traditional process has a single thread of control.
- **Multi-threaded applications** have multiple threads within a single process.
- **Thread** is a basic unit of CPU utilization.
- Process creation is many times heavy-weight while thread creation is light-weight.

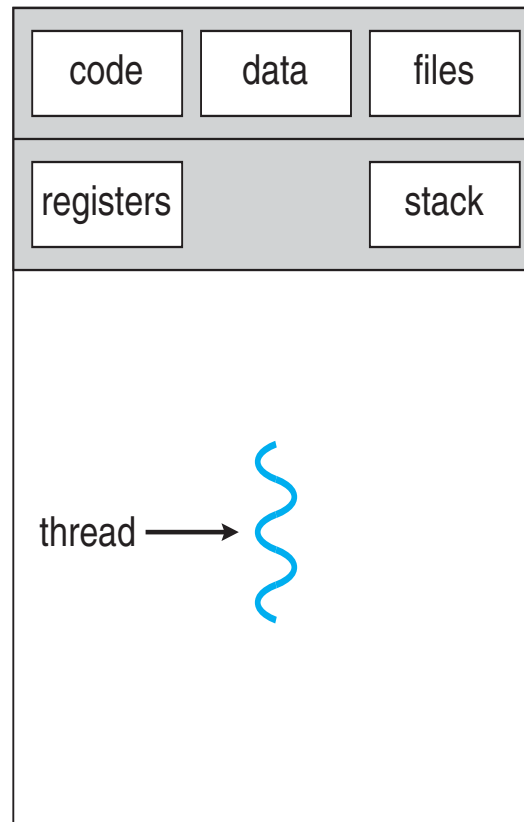
Threads in Single and Multithreaded Processes

- Each thread consists of:

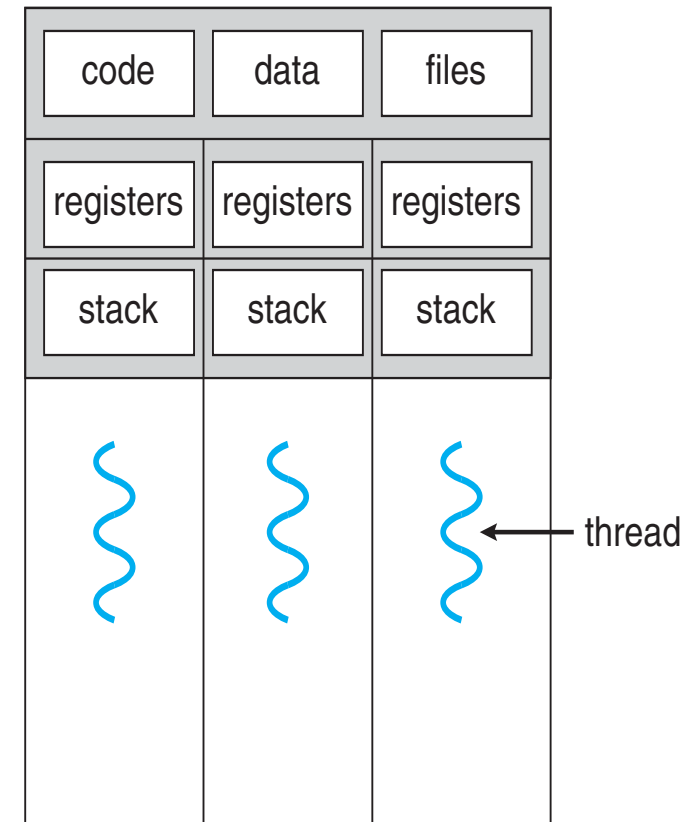
- Thread ID,
- Program Counter,
- Set of registers
- Stack

- However, they share

- code,
- data, and
- files



single-threaded process



multithreaded process

Multi-threading examples

- Editing word document
 - One thread can interpret the key strokes.
 - Second thread display images.
 - Third thread checks spelling and grammar.
 - Fourth thread does periodic automatic backups of the file being edited.
- Most operating system kernels are multi-threaded.
 - Many threads operate in the kernel process, where each thread performs a specific task.
 - Managing devices
 - Managing memory
 - Interrupt handling, etc.

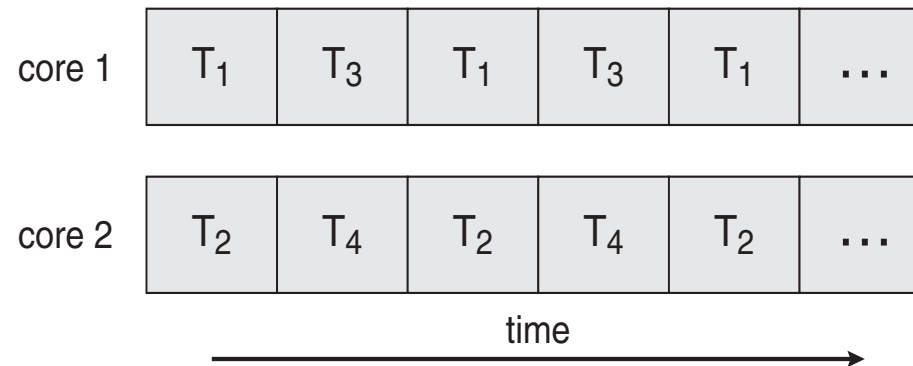
Advantages

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources by default.
 - Resource sharing is easier between threads than processes
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – a single process can take advantage of multiprocessor architectures
- Threads enable concurrent programming and, on multiple processor systems, true parallelism.

Parallelism

Parallelism implies a system can perform more than one task simultaneously

Parallelism on a multi core system

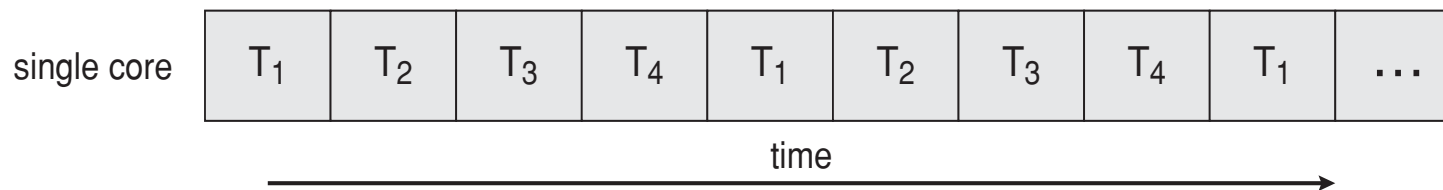


Concurrency

Concurrency supports more than one task making progress

- Single processor / core, scheduler provides concurrency

Concurrent execution on single-core system:



It is possible to have concurrency without parallelism?

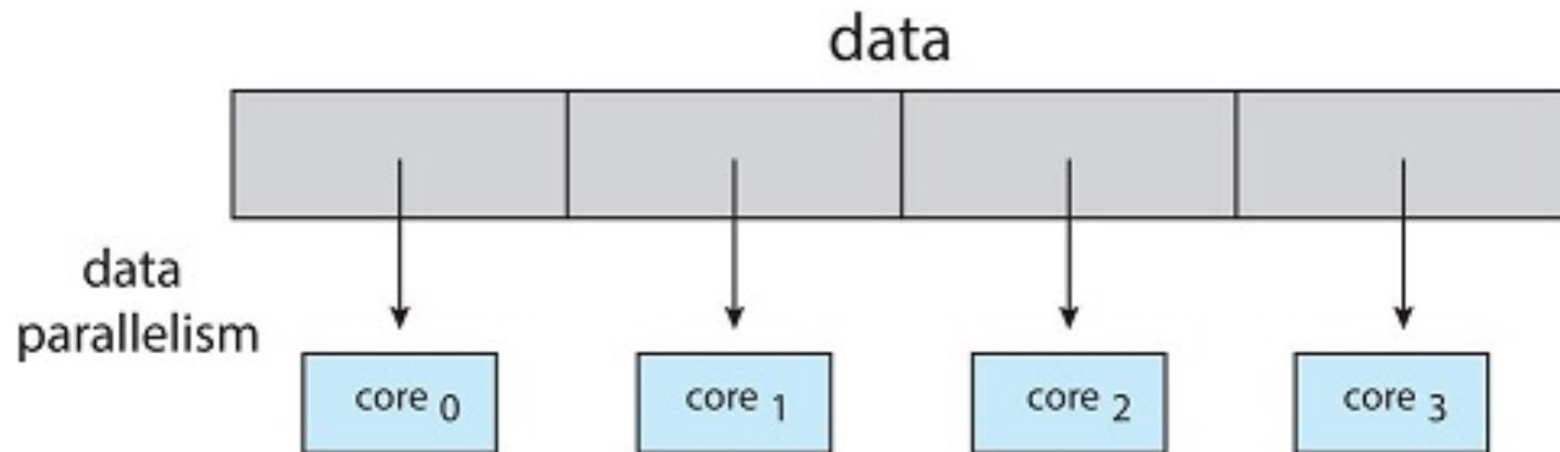
Multicore Programming

- **Multicore** or **multiprocessor** systems puts pressure on programmers, challenges include:
 - **Dividing activities:** Identifying tasks in the application that can be performed concurrently.
 - **Balance:** Evaluate if the tasks coded to run concurrently provide same or more value than the overhead of thread creation.
 - **Data splitting:** Preventing threads from interfering with each other.
 - **Data dependency:** Tasks need to be synchronized if data is shared.
 - **Testing and debugging:** Challenging as the race conditions become difficult to identify.
 - **Race condition** results when several threads try to **access and modify** the same data concurrently

Multicore Programming (Cont.)

Two types of parallelism: Data and Task parallelism

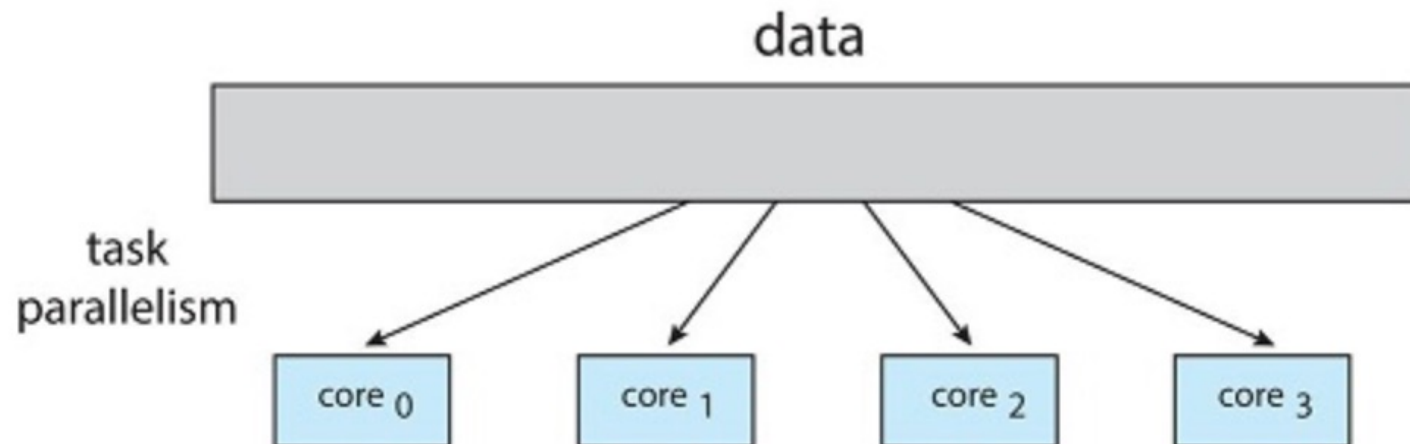
- **Data parallelism:** distributes subsets of the same data across multiple cores, same operation on each subsets



- **Example:** Adding numbers 1 to N, where N is large. The set is divided into number of cores and the same computation is performed on each set.

Multicore Programming (Cont.)

- **Task parallelism** – distributing threads across cores, each thread performing unique operation.



- **Example:** Windows word document example
(Slide# 4)

Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- N processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- If an application is 75% parallel (25% serial), moving from 1 to 2 cores results in speedup of 1.6 times
- As $N \rightarrow \infty$, speedup approaches $1 / S$.

Amdahl's Law Contd..

- According to the law, adding more # of processes after a certain number has no effect on speedup!
- Some suggest formula does not account for hardware performance, therefore ceases to apply $N \rightarrow \infty$

User Threads and Kernel Threads

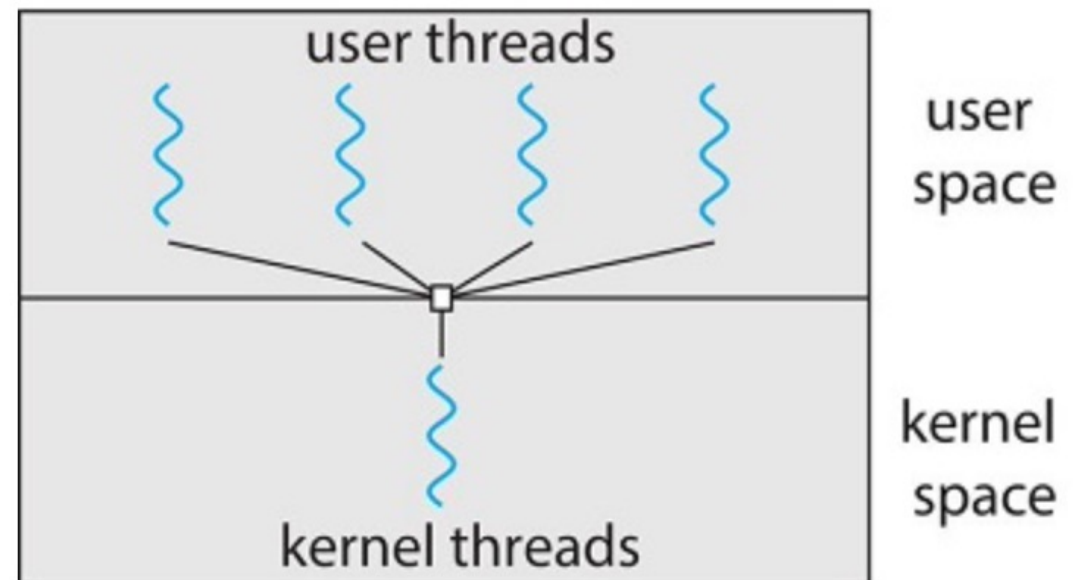
- **User level threads** - management done by user-level threads library in user space. These threads are put by programmers into their programs.
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Windows threads
 - Java threads
- **Kernel level threads** – managed by the Kernel in kernel space.
- Virtually all general purpose operating systems have thread support:
 - E.g., Windows , Solaris, Linux, Tru64 UNIX, Mac OS X

Multithreading Models

- A relationship must exist between user threads and kernel threads.
- This is achieved using one of the below three ways.
 - Many-to-One
 - One-to-One
 - Many-to-Many

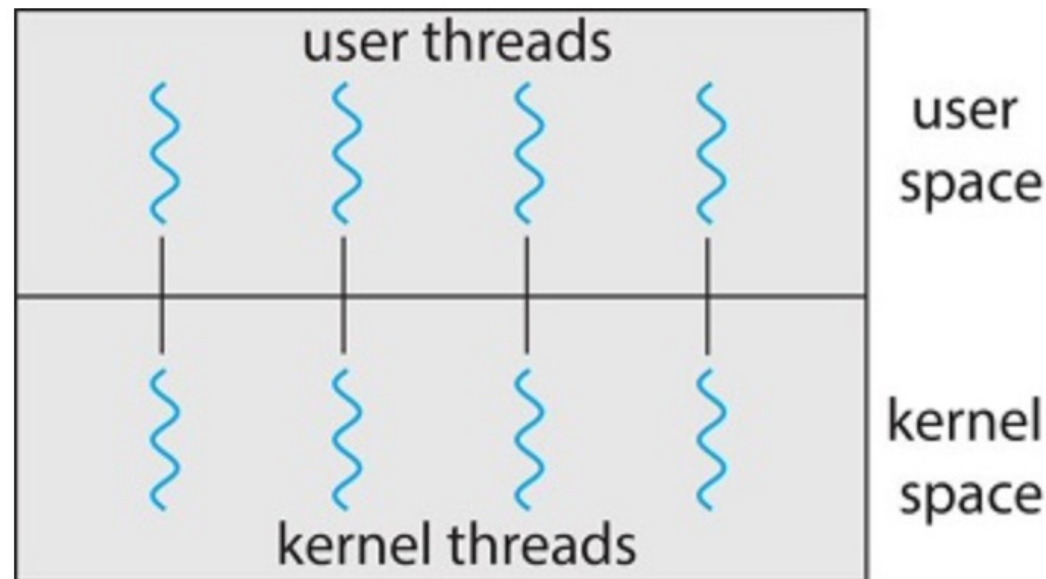
Many-to-One

- Many user-level threads mapped to single kernel thread
- Thread management handled by the **thread library in user space** - very efficient.
- One thread blocking causes all threads to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples OS (implemented it in past):
 - **Solaris Green Threads**
 - **GNU Portable Threads**



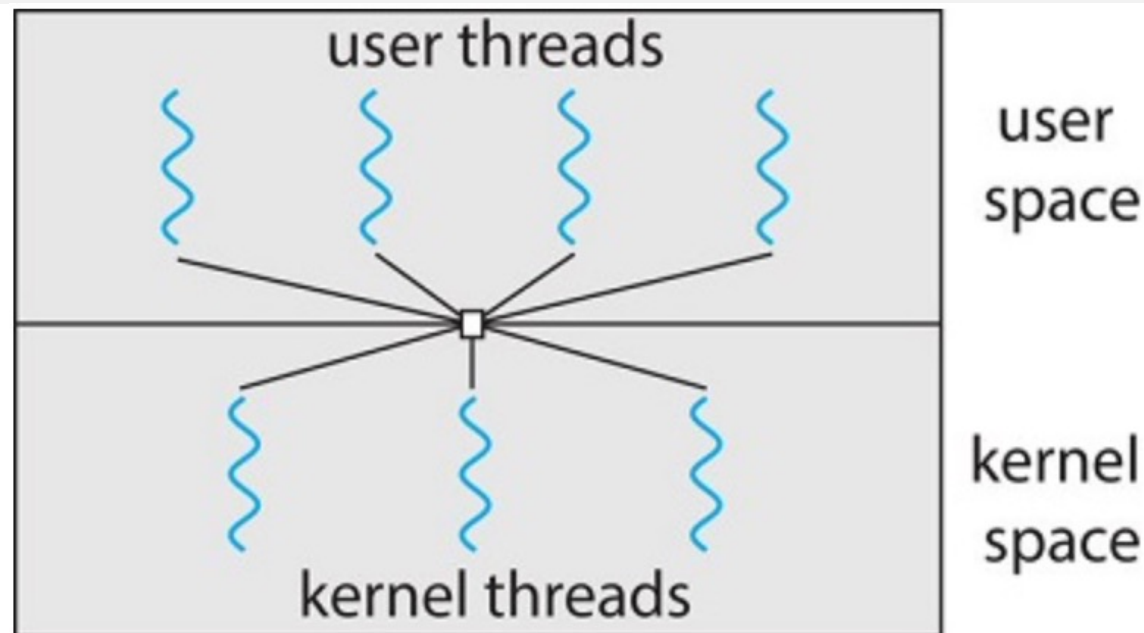
One-to-One

- Each user-level thread maps to a kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one, but has max. overhead.
- Number of threads per process sometimes restricted due to overhead
- Examples: Windows, Linux, Solaris 9 and later



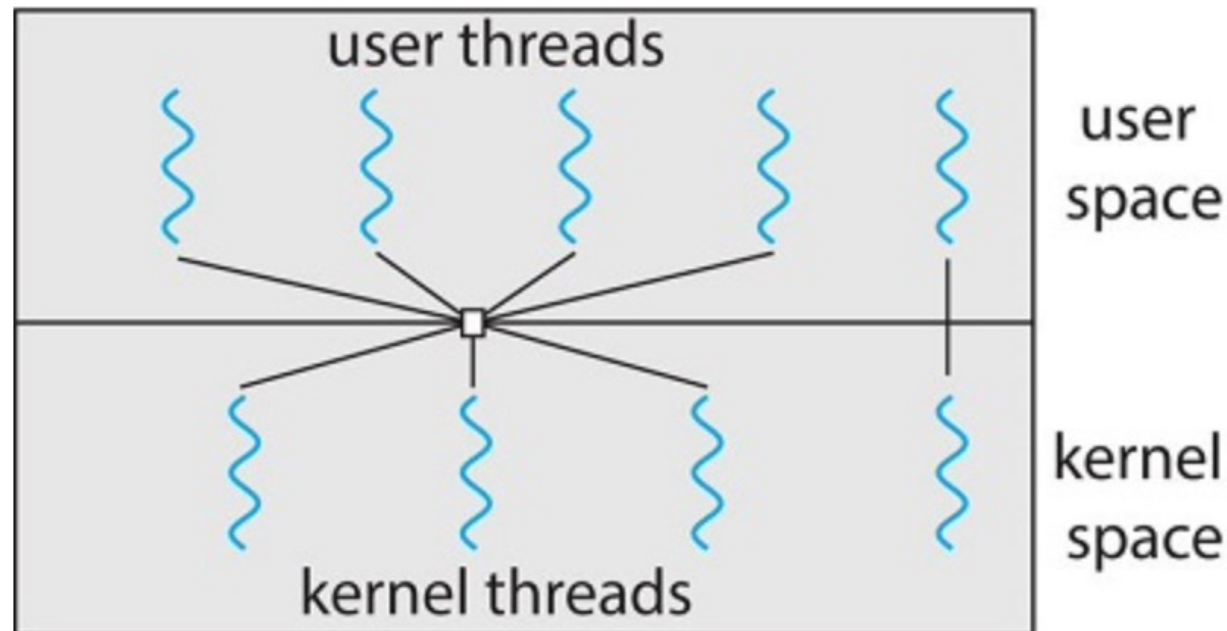
Many-to-Many Model

- Allows many user level threads to be mapped to an equal or smaller no. of kernel threads.
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls does not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.
- Examples: Solaris prior to version 9



Two-level Model (variation of Many-to-many)

- Similar to many to many model, except that it allows a user thread to be **bound** to kernel thread as well
 - Solaris 8 and earlier



Pthreads

- May be provided either as user-level or kernel-level
- Pthreads, is a POSIX standard 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)
- On Linux, pthread library implements the 1:1 model
- Function names start with “pthread_”

Pthreads

- To be able to create threads in your C program you need to include the `pthread.h` header file.
- Each thread has a unique thread ID. To create thread IDs for your threads in your program you should use the `pthread_t` data type.
- Thread attributes should be created/modified using `pthread_attr_t` data structure.
- Declare and code the function in which the thread begins control. For an example see the `runner()` function on the next slide.
- To create threads use `pthread_create()` function and pass in the necessary parameters.
- For the parent thread to output the sum after all summing threads have exited, it is important that you use the `pthread_join()` function.

Pthreads Example

```
#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.)

```
/* 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);  
}
```

Question

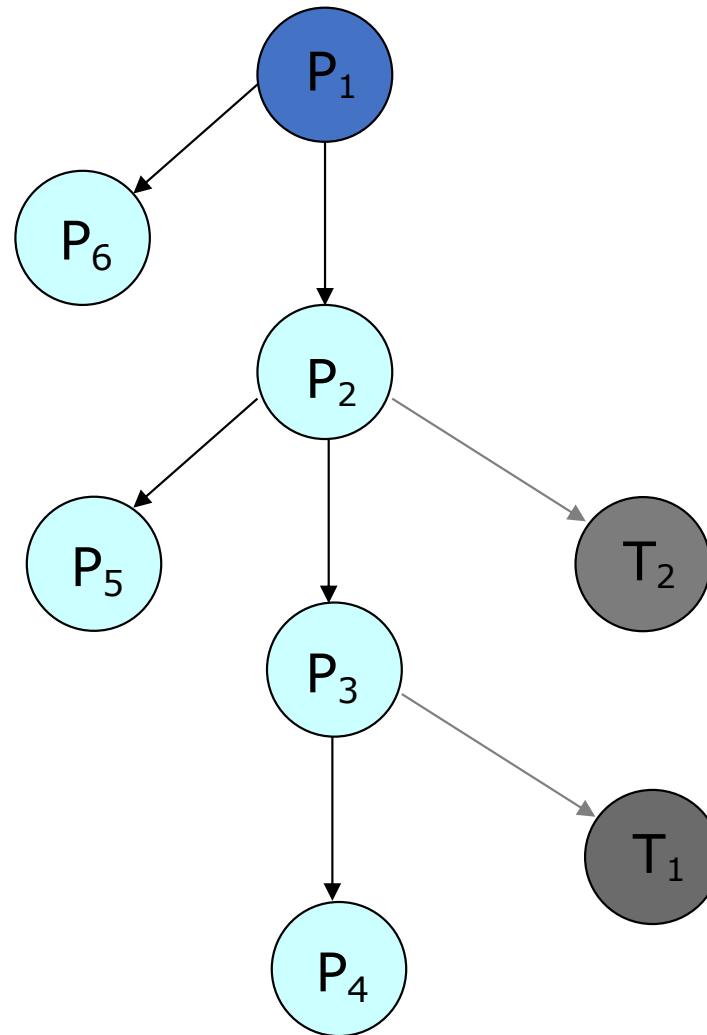
```
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) { /* child process */
        fork();
        thread create( . . . );
    }
    fork();
    return 0;
}
```

How many unique processes are created?

How many unique threads are created?

Draw the process and thread tree.

Process and thread tree for question in slide 4.24



Linux Threads

- Linux treats processes and threads the same.
- Linux refers to them as *tasks* rather than *threads*
- Thread creation as discussed here can be achieved through `clone()` system call by setting appropriate flags (enables behaviors) in the system call.

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.

- `clone()` is better than `fork()` as offer fine grained control in creating the child process and is allows a child task to share the address space of the parent task (process)

Implicit Threading

- In **Implicit Threading**, creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
 - Thread Pools
 - OpenMP
 - Grand Central Dispatch