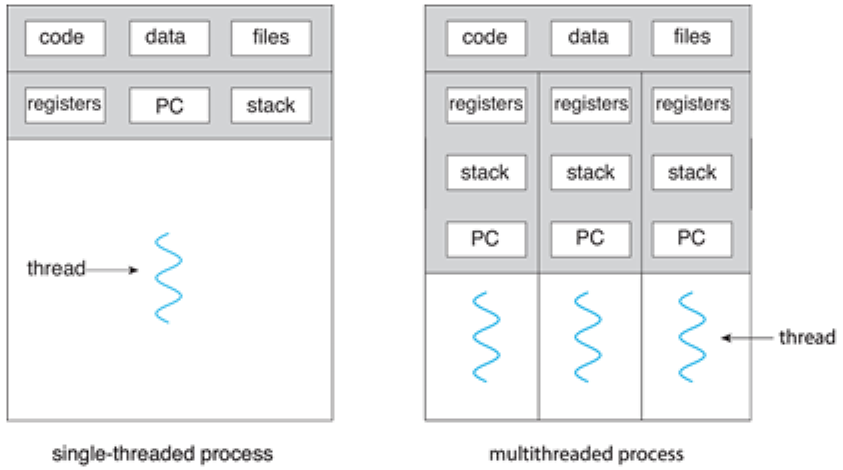
OPERATING SYSTEMS

Threads

FBA

Thread

➢ A thread is a path of execution within a process.



- A thread contains -
    - Thread ID
    - Program Counter
    - Register Set
    - Stack
- Shares with other threads belonging to the same process -
    - Code Section
    - Data Section
    - OS resources

➢ A traditional process has a single thread of control (Single Threaded Process)

➢ Process with multiple threads of control, can perform more than one task at a time (Multi Threaded Process)

Benefits

There are four major categories of benefits to multi-threading:

1. **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.

1. **Resource sharing** - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.

1. **Economy** - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.

1. **Scalability, i.e. Utilization of multiprocessor architectures** - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. (Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)

Multicore Programming

Multicore or **multiprocessor** systems putting pressure on programmers, challenges include:

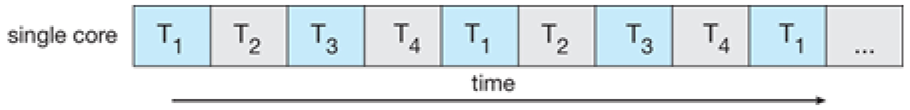
- ➔ Dividing activities
- ➔ Balance
- ➔ Data splitting
- ➔ Data dependency
- ➔ Testing and debugging

Parallelism implies a system can perform more than one task simultaneously

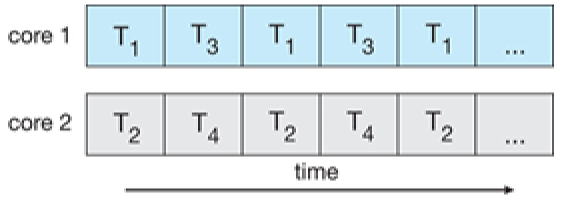
Concurrency supports more than one task making progress

- Single processor / core, scheduler providing concurrency

Concurrent execution on single-core system:



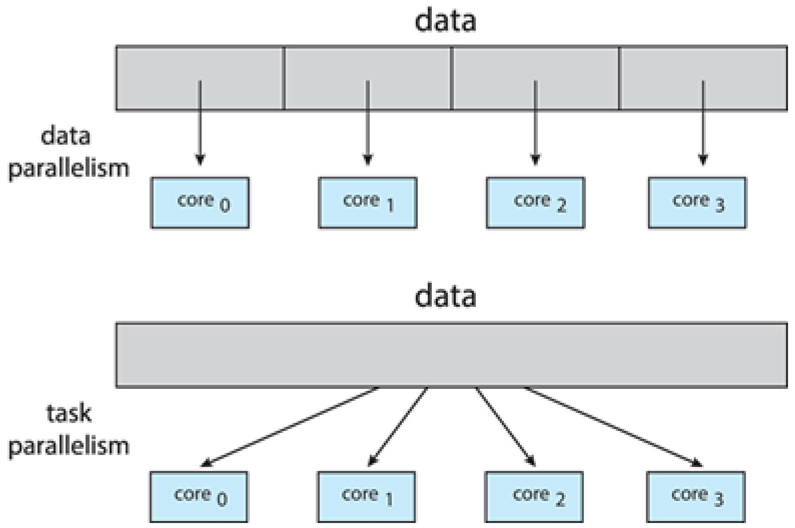
Parallelism on a multi-core system:



Multicore Programming

Data parallelism – distributes subsets of the same data across multiple cores, same operation on each

Task parallelism – distributes threads across cores, each thread performing unique operation



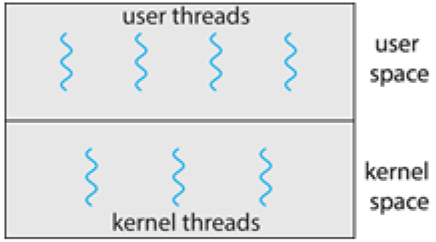
OPERATING SYSTEMS

Multithreading Models

FBA

User and Kernel Threads

- There are two types of threads to be managed in a modern system: User threads and kernel threads.
- In a specific implementation, the user threads must be mapped to kernel threads.



- **User threads** are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.

  Examples- POSIX Pthreads, Windows threads, Java threads

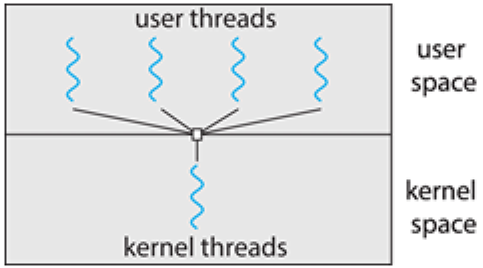
- **Kernel threads** are supported within the kernel of the OS itself. All modern OS support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

  Examples- virtually all general-purpose operating systems, including: Windows, Linux, Mac OS X, iOS, Android

Multithreading Models

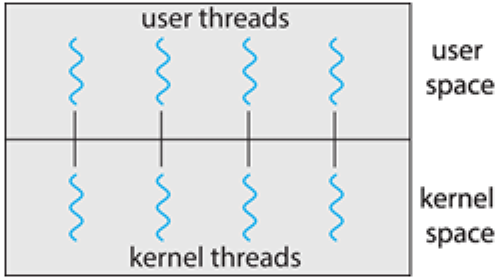
Many-to-One:

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



One to One:

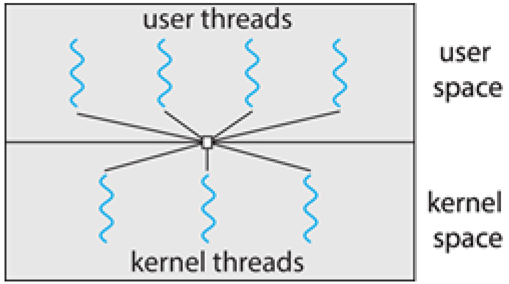
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



Multithreading Models

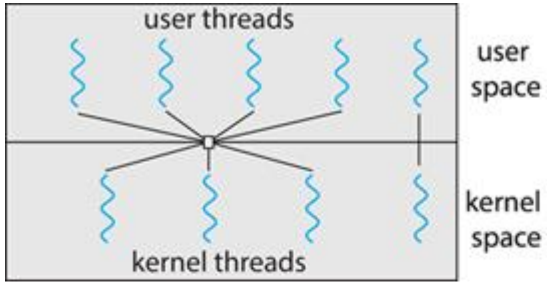
Many-to-Many:

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the ThreadFiber package
- Otherwise not very common



Two-level Model:

- Similar to M:M, except that it allows a user thread to be bound to kernel thread



OPERATING SYSTEMS

Thread Libraries

FBA

Thread library

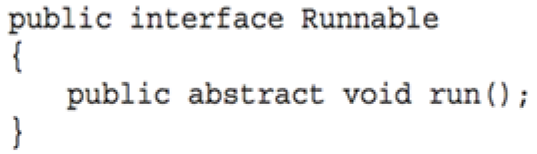
- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls, and requires a kernel with thread library support.
- There are three main thread libraries in use today:
  - POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
  - Win32 threads - provided as a kernel-level library on Windows systems.
  - Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

Pthreads

- The POSIX standard (IEEE 1003.1c) defines the specification for pThreads, not the implementation.
- pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
- Global variables are shared amongst all threads.
- One thread can wait for the others to rejoin before continuing.
- pThreads begin execution in a specified function.
- Common in UNIX operating systems (Linux & Mac OS X)

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface



```
public interface Runnable
{
    public abstract void run();
}
```

➔ Standard practice is to implement Runnable interface

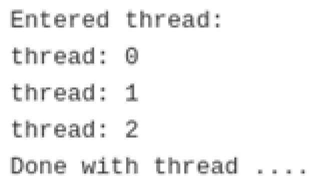
Create Thread using pthread in C:

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

void *funcThread(void *arg);
int main(){
            pthread_t t1;
            pthread_create(&t1,NULL,funcThread,NULL);
            pthread_join(t1,NULL);

            return 0;
}
void *funcThread(void *arg){
            printf("Entered thread:\n");
            for(int i=0;i<3;i++){
                        printf("thread: %d\n",i);
            }
            printf("Done with thread ....\n");
}
```

Output:



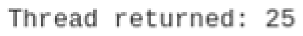
```
Entered thread:
thread: 0
thread: 1
thread: 2
Done with thread ....
```

Return values from thread function using pthread in C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
int *func_thread(int *v);
int *t_ret;
int main(){
          pthread_t t1;
          int n=5;
          pthread_create(&t1,NULL,func_thread,&n);
          pthread_join(t1,&t_ret);
          printf("Thread returned: %d\n",t_ret);

          return 0;
}
int *func_thread(int *v){
    *v=*v*5;
          return *v;
}
```

Output



```
Thread returned: 25
```

Return values from thread function by cancelling the thread using pthread in C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
void *func_thread(int *n);
void *t_ret;
int num=5;
int main(){
            pthread_t t1;
            pthread_create(&t1,NULL,(void *)func_thread,&num);
            pthread_join(t1,&t_ret);
            printf("Thread returned: %d\n",(int *)t_ret);

            return 0;
}

void *func_thread(int *n){
            printf("Entered in Thread:\n");
            if(*n % 2==0){
                        pthread_exit(*n * *n);
                        printf("Operation completed\n");
            }
            else{
                        pthread_exit(*n * *n * *n);
                        printf("Operation completed\n");
            }
}
```

Output

```
Entered in Thread:
Thread returned: 125
```

Implicit Threading

Motivation:
Shifts the burden of addressing the programming challenges outlined earlier (Dividing activities,Balance,Data splitting,Data dependency,Testing and debugging) , from the application programmer to the compiler and run-time libraries.

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Five methods explored:
    - Thread Pools
    - Fork-Join
    - OpenMP
    - Grand Central Dispatch
    - Intel Threading Building Blocks

Thread Pool

- ➔ Create a number of threads in a pool where they await work.
- ➔ Advantages:
  - ◆ Usually slightly faster to service a request with an existing thread than create a new thread.
  - ◆ Allows the number of threads in the application(s) to be bound to the size of the pool.
  - ◆ Separating task to be performed from mechanics of creating task allows different strategies for running task.
    - ● i.e,Tasks could be scheduled to run periodically.
- ➔ Windows API supports thread pools:

PVOID

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
     * this function runs as a separate thread.
     */
}
```

Threading Issues

- **fork() and exec() System Calls:** Duplicate all the threads or not?
- **Thread cancellation:** Thread cancellation is the task of terminating a thread before it has completed.
- **Signal Handling:** Where should a signal be delivered?
- **Thread Pool:** Create a number of threads at the process start-up.
- **Thread Specific data:** Each thread might need it's own copy of certain data.

The fork() and exec() system calls

- fork()
  - The fork() system call is used to create a separate, duplicate process
- exec()
  - When a exec() system call is invoked, the program specified in the parameter to exec() will replace the entire process –including all threads
- **Issue:** If one thread in a program calls fork(), does the new process duplicate all threads or the new process single-threaded?
- **Solution:** Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call
- Which version of fork() will to be used depends on the application
  - If exec() is called immediately after forking then duplicating all threads is unnecessary.
  - If the separate process does not call exec() after forking, the separate process should duplicate all threads

Thread Cancellation

- Thread cancellation is the task of Terminating a thread before it has completed.
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation:** one thread terminates the target thread immediately.
    - There is an issue if a thread is cancelled while in the midst of updating data it is sharing with other threads.
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - Cancellation occurs only after the target thread has checked a flag to determine if it should be cancelled or not. So it should be canceled at a point when it can be cancelled safely.

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- Synchronous and asynchronous received signal.
- A **signal handler** is used to process signals
    1. A signal is generated by the occurrence of a particular event
    2. A generated signal is delivered to a process
    3. Once delivered, the signal must be handled
- Every Signal may be handled by one of two possible handlers:
    1. A default signal handler
    2. A user-defined signal handler

Signal Handling

- Signals are always delivered to a process but delivering signals is more complicated in multi-threaded program. In that case following options exist
  1. Deliver the signal to the thread to which the signal applies (Synchronous cancellation)
  2. Deliver the signal to every thread in the process (Asynchronous cancellation)
  3. Deliver the signal to certain threads in the process
  4. Assign a specific thread to receive all signals for the process