Chapter 4

Threads & Concurrency

Yunmin Go

School of CSEE

HANDONG GLOBAL UNIVERSITY

Agenda

- Overview
- Parallelism and Concurrency
- Multithreading Models
- Thread Creation
- Thread Cancellation & Some Issues
- Implicit Threading

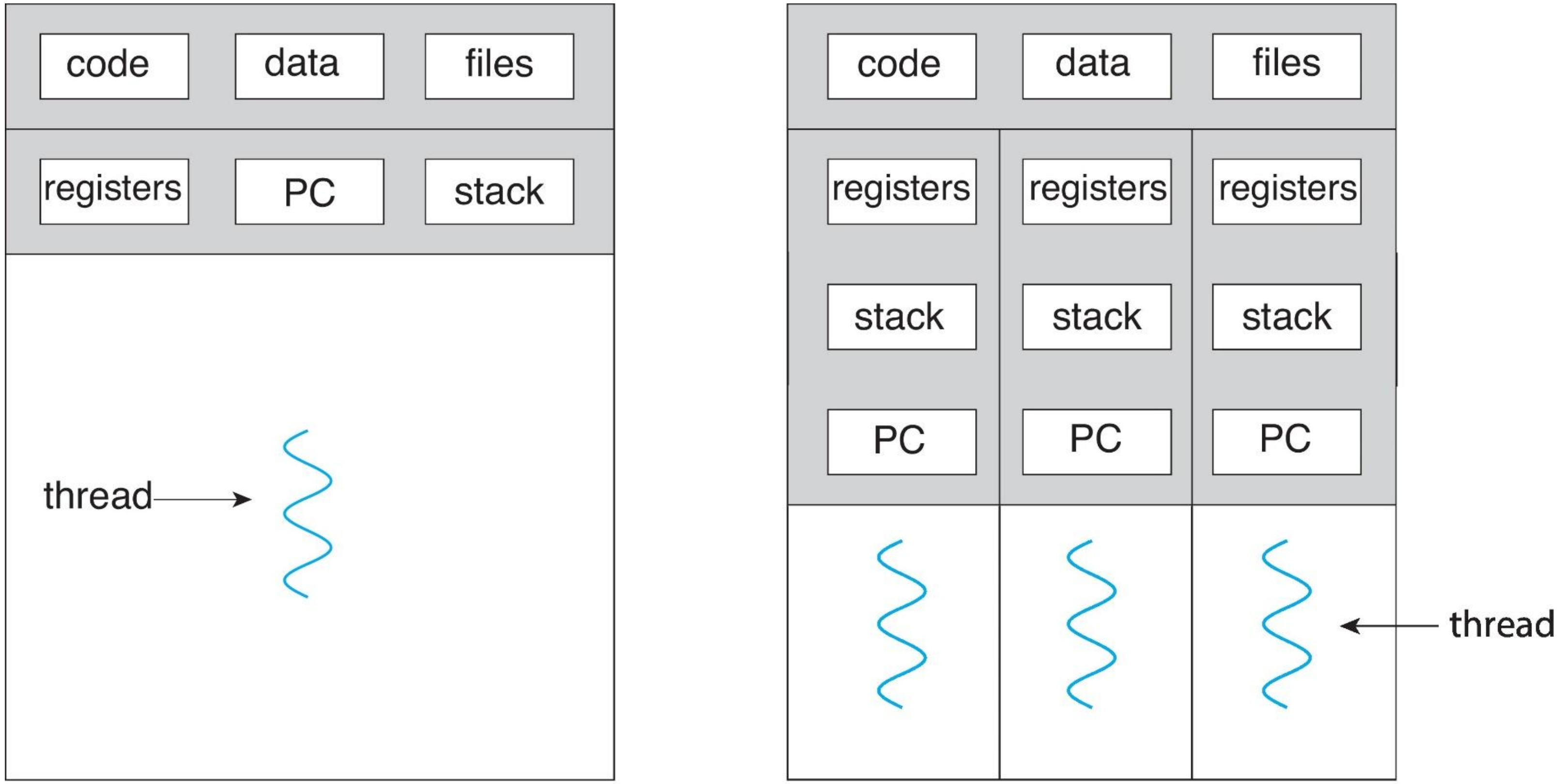
Overview

- **Process**: program in execution
  - Each process occupies resources required for execution

- **Thread**: a way for a program to split itself into two or more simultaneously running tasks
  - Basic unit of CPU utilization
  - Smaller unit than process
  - Threads in a process share resources

- A thread is comprised of
  - Thread ID, program counter, register set, stack, etc.

Single and Multithreaded Processes

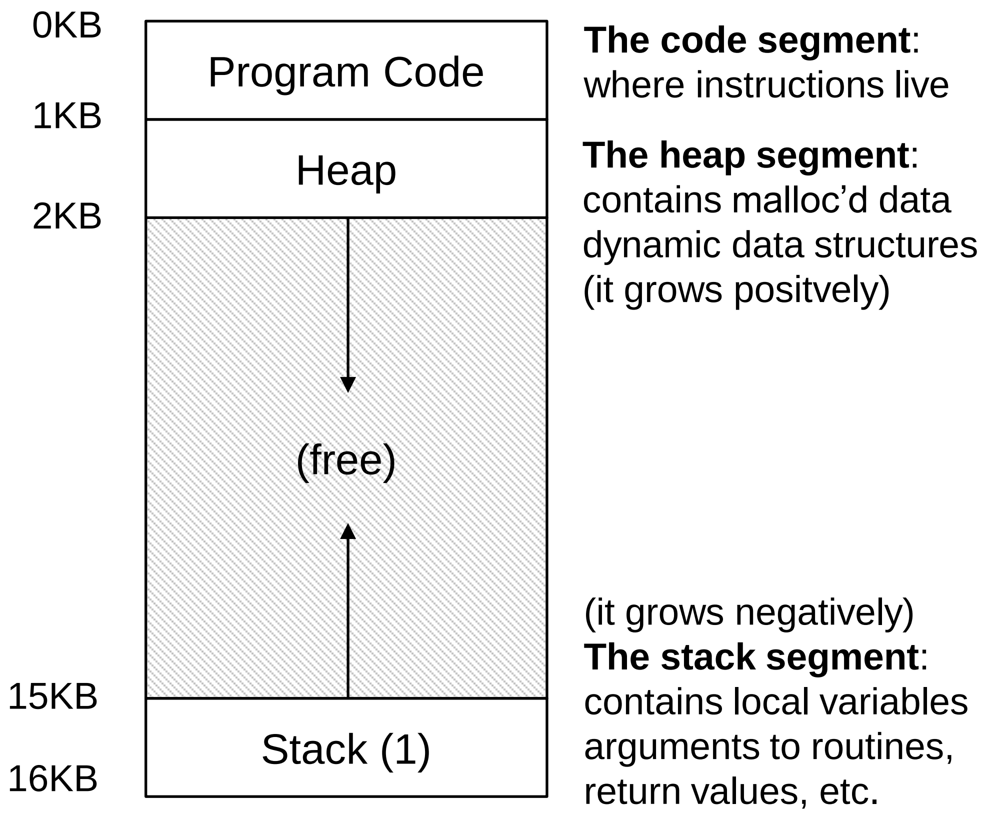


single-threaded process

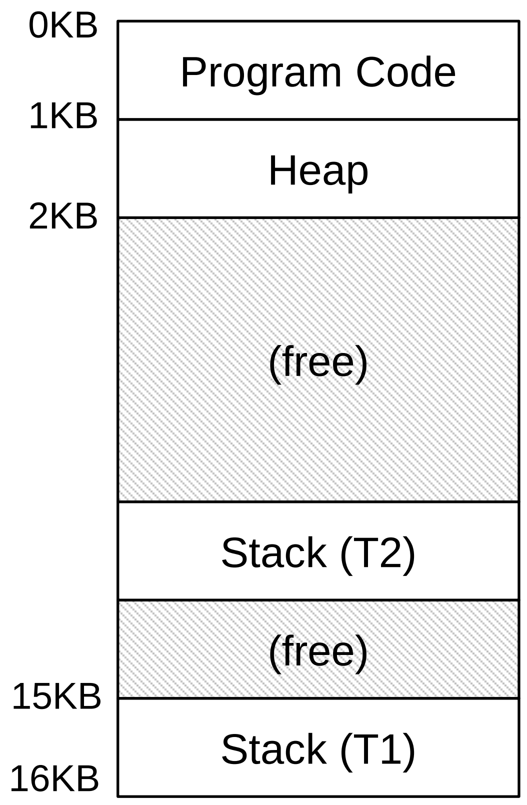
multithreaded process

Address Spaces

- There will be one stack per thread



<A Single-Threaded Address Space>



<Two threaded Address Space>

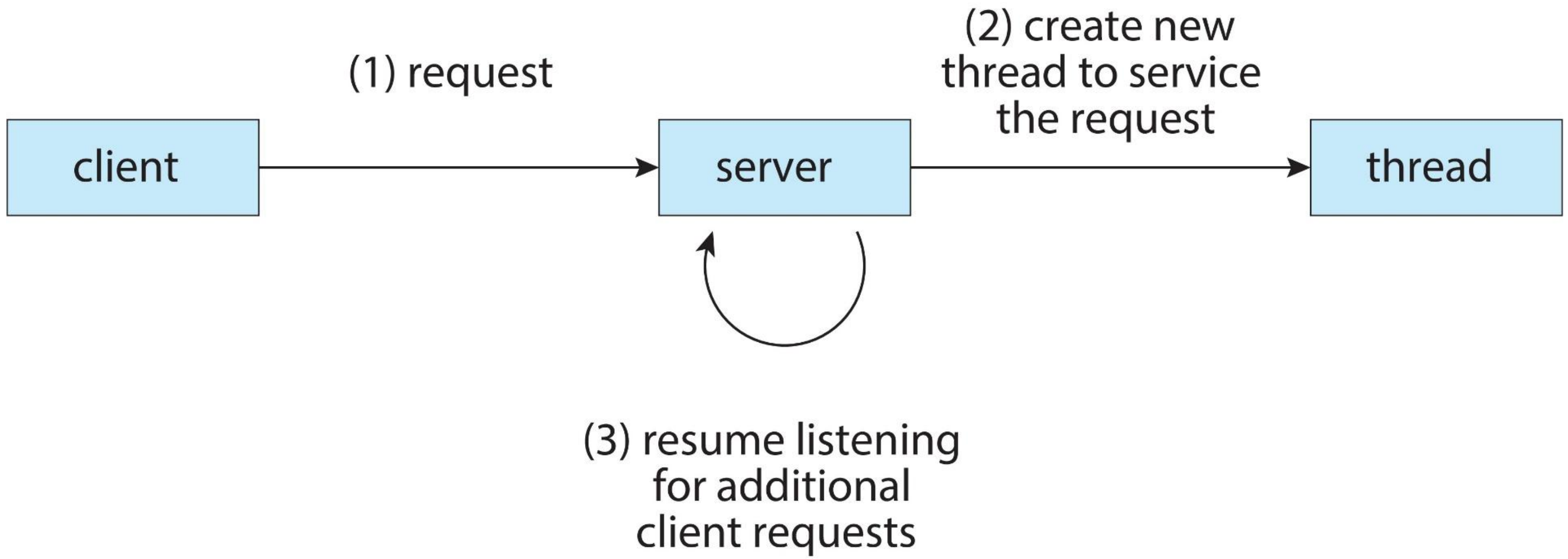
Thread Control Block (TCB)

- **Thread Control Block (TCB)** is a data structure in the operating system kernel which contains thread-specific information needed to manage it.

- Examples of information in TCB (`thread_struct` in Linux)
  - Thread id
  - State of the thread (running, ready, waiting, start, done)
  - Stack pointer
  - Program counter
  - Thread's register values
  - Pointer to the process control block (PCB)

Why Use Threads?

- Process creation is expensive in time and resource
  - ex) Web server accepting thousands of requests



Why Use Threads?

- Parallelism
  - Single-threaded program: the task is straightforward, but slow.
  - Multi-threaded program: natural and typical way to make programs run faster on modern hardware.
  - **Parallelization**: The task of transforming standard single-threaded program into a program that does this sort of work on multiple CPUs.

- To avoid blocking program progress due to slow I/O.
  - Threading enables overlap of I/O with other activities within a single program.
  - It is much like multiprogramming did for processes across programs.

Benefits

- **Responsiveness**: may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing:** threads share resources of process, easier than shared memory or message passing
- **Economy:** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability:** process can take advantage of multiprocessor architectures

Agenda

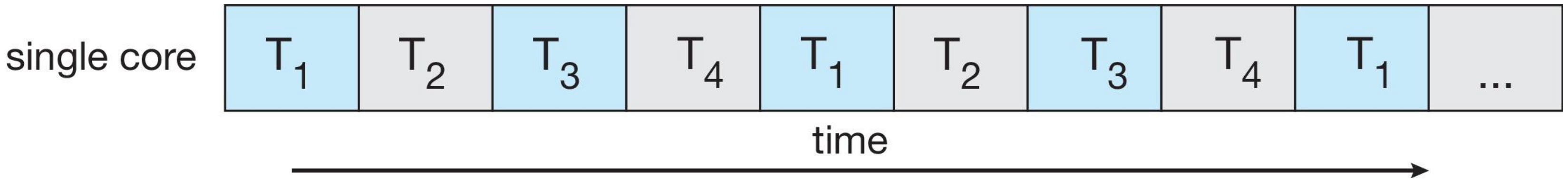
- Overview
- Parallelism and Concurrency
- Multithreading Models
- Thread Creation
- Thread Cancellation & Some Issues
- Implicit Threading

Parallelism and Concurrency

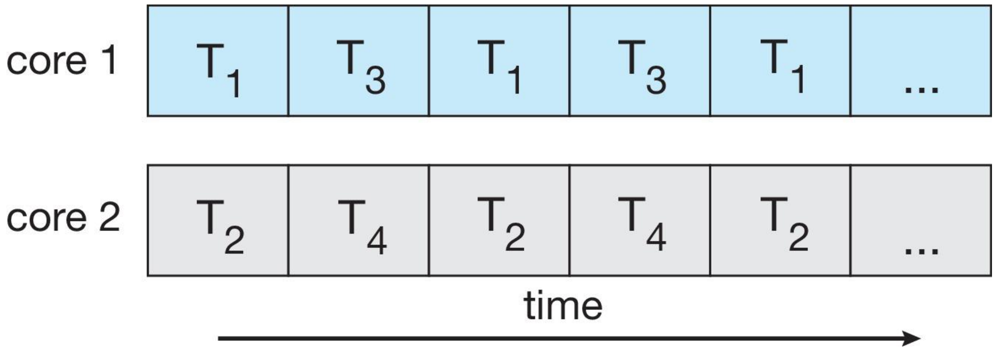
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Parallelism and Concurrency

- Concurrency vs. Parallelism
  - Concurrent execution on single-core system:

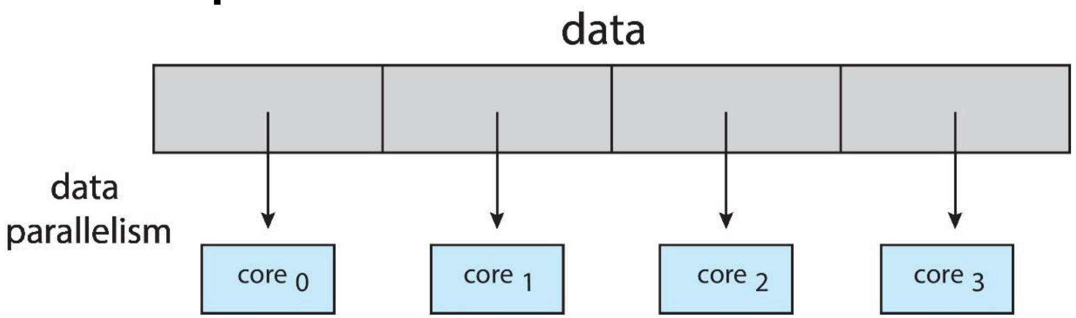


  - Parallelism on a multi-core system:

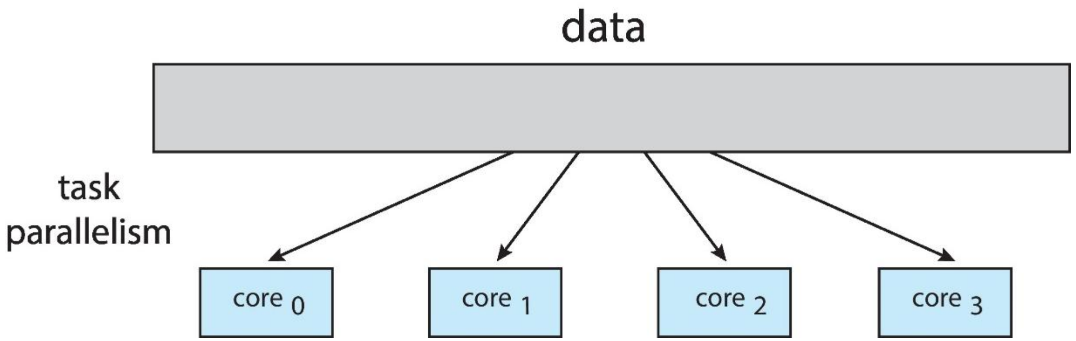


Types of Parallelism

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



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

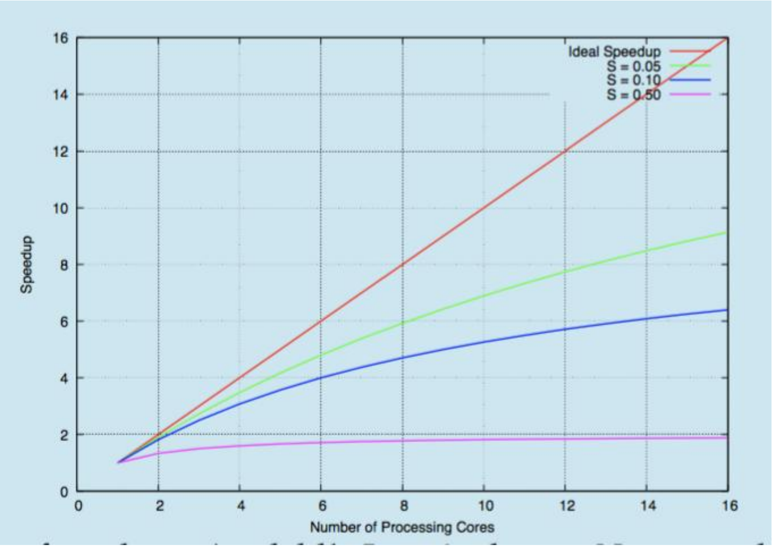


Amdahl's Law

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

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

    - ex) Application is 75% parallel (25% serial), moving from 1 to 2 cores results in speedup of 1.6 times



  - As N approaches infinity, speedup approaches 1 / S
  - Serial portion of an application has disproportionate effect on performance gained by adding additional cores.

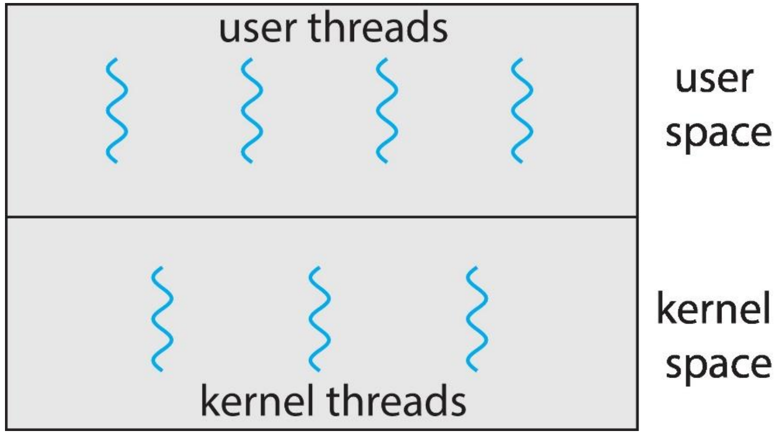
But does the law take into account contemporary multicore systems?

Agenda

- Overview
- Parallelism and Concurrency
- Multithreading Models
- Thread Creation
- Thread Cancellation & Some Issues
- Implicit Threading

User Threads and Kernel Threads

- **User thread**: thread supported by thread library in user level
  - Created by library function call (not system call)
  - Kernel is not concerned in user thread
  - Switching of user thread is faster than kernel thread.
- **Kernel thread**: thread supported by kernel
  - Created and managed by kernel
  - Scheduled by kernel
  - Cheaper than process
  - More expensive than user thread



Kernel Thread

- Most operating system kernels are also typically multithreaded
  - Each thread performs a specific task, such as managing devices, memory management, or interrupt handling

```
mcnl@mcnl:~$ ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0  2021 ?        00:06:14 /lib/systemd/systemd --system --deserialize 37
root           2       0  0  2021 ?        00:00:03 [kthreadd]
root           3       2  0  2021 ?        00:00:00 [rcu_gp]
root           4       2  0  2021 ?        00:00:00 [rcu_par_gp]
root           6       2  0  2021 ?        00:00:00 [kworker/0:0H-kblockd]
root           9       2  0  2021 ?        00:00:00 [mm_percpu_wq]
root          10       2  0  2021 ?        00:00:11 [ksoftirqd/0]
root          11       2  0  2021 ?        00:11:21 [rcu_sched]
root          12       2  0  2021 ?        00:00:23 [migration/0]
root          13       2  0  2021 ?        00:00:00 [idle_inject/0]
root          14       2  0  2021 ?        00:00:00 [cpuhp/0]
root          15       2  0  2021 ?        00:00:00 [cpuhp/1]
root          16       2  0  2021 ?        00:00:00 [idle_inject/1]
root          17       2  0  2021 ?        00:00:24 [migration/1]
root          18       2  0  2021 ?        00:00:02 [ksoftirqd/1]
root          20       2  0  2021 ?        00:00:00 [kworker/1:0H-kblockd]
root          21       2  0  2021 ?        00:00:00 [cpuhp/2]
root          22       2  0  2021 ?        00:00:00 [idle_inject/2]
root          23       2  0  2021 ?        00:00:26 [migration/2]
root          24       2  0  2021 ?        00:00:01 [ksoftirqd/2]
root          26       2  0  2021 ?        00:00:00 [kworker/2:0H-kblockd]
root          27       2  0  2021 ?        00:00:00 [cpuhp/3]
root          28       2  0  2021 ?        00:00:00 [idle_inject/3]
root          29       2  0  2021 ?        00:00:25 [migration/3]
root          30       2  0  2021 ?        00:00:01 [ksoftirqd/3]
```

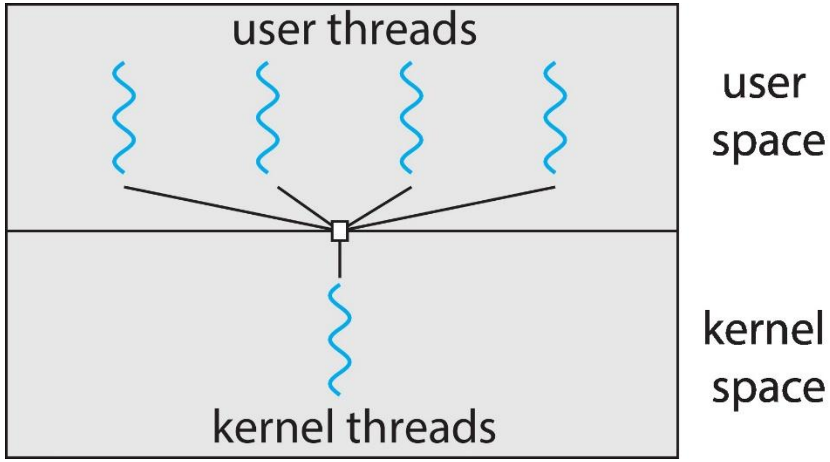
Multithreading Models

- Major issue: correspondence between user treads and kernel threads

- Multithreading Models
  - Many-to-One
  - One-to-One
  - Many-to-Many (Two-level)

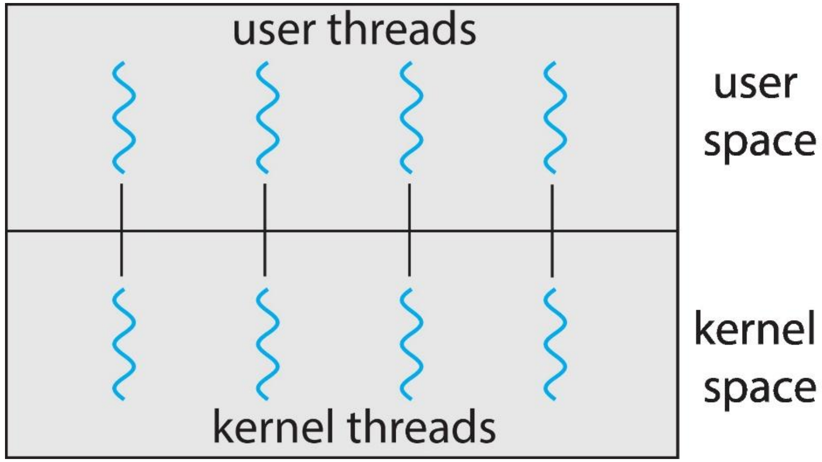
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - Early version of Java



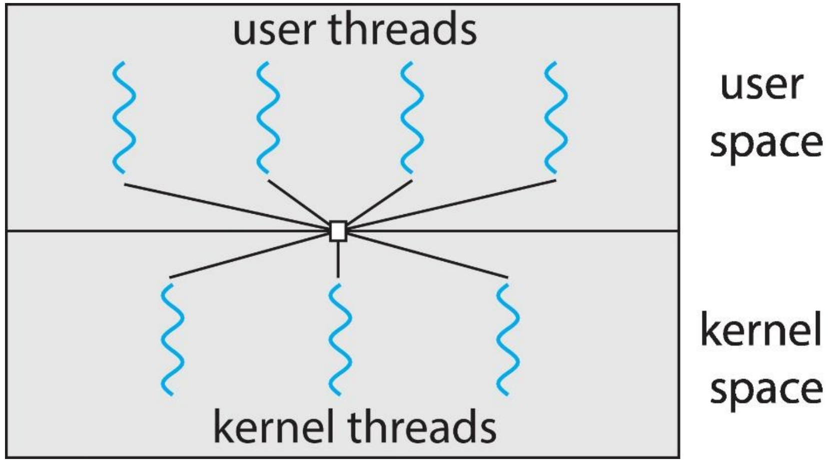
One-to-One

- 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

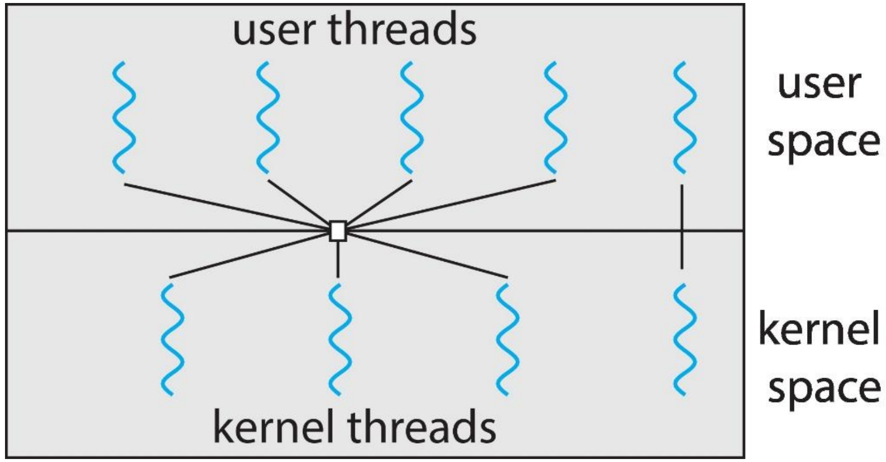


Many-to-Many Model

- 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



Many-to-Many Model



Two-level Model

Agenda

- Overview
- Parallelism and Concurrency
- Multithreading Models
- Thread Creation
- Thread Cancellation & Some Issues
- Implicit Threading

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - User level library (entirely in user space with no kernel support)
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) 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)

Pthread Create

- `pthread_create()`: Create a new thread

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

  - Starts a new thread in the calling process
  - *thread*: new thread ID on success
  - *attr*: to set thread attribute (stack size, scheduling priority…). NULL for default value.
  - *start_routine*: the routine that the thread will execute when it is created
  - *arg*: argument passed to *start_routine*
  - Return value
    - Success: 0
    - Error: error number

Pthread Join

- `pthread_join()`: Join with a terminated thread

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

  - It blocks the calling thread until the specified thread returns
  - When a parent thread returns, all the child thread are terminated
  - *thread*: specified thread to terminate
  - *retval*: return value from the terminated thread (a pointer returned by reference)
  - Return value
    - Success: 0
    - Error: error number

https://man7.org/linux/man-pages/man3/pthread_join.3.html

Compiling and Running

- To compile them, you must include the header `pthread.h`
  - Explicitly link with the pthreads library, by adding the –pthread flag.

```
$ gcc –o main main.c –Wall -pthread
```

  - The recent gcc automatically links with the pthread library.

Example: Thread Creation

thread_create.c

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

void *mythread (void *arg) {
    printf ("%s\n", (char *) arg);
    return NULL;
}

int main (int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

```
yunmin@yunmin:~/ch4$ ./thread_create
main: begin
A
B
main: end
```

* Trace #1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

* Trace #2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

* Trace #3

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Example: Passing Argument to Thread

thread_sum.c

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

void *thread_summation(void * arg);
int sum = 0;

void *thread_summation(void * arg) {
    int start = ((int*)arg)[0];
    int end = ((int*)arg)[1];
    while (start <= end)
    {
        sum+=start;
        start++;
    }
    return NULL;    // or pthread_exit(0);
}
```

```
int main(int argc, char *argv[]) {
    pthread_t id_t1, id_t2; // thread id
    int range1[] = {1, 5};
    int range2[] = {6, 10};

    // create thread
    pthread_create(&id_t1, NULL, thread_summation, (void *)range1);
    pthread_create(&id_t2, NULL, thread_summation, (void *)range2);

    // wait for the thread to exit
    pthread_join(id_t1, NULL);
    pthread_join(id_t2, NULL);
    printf("result: %d \n", sum);
    return 0;
}
```

Result?

Example: Return Value from Thread

thread_ret.c

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

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}
```

```
int main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void **) &m);  // this thread has been
                                    // waiting inside of the
                                    // pthread_join() routine.
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

Result?

Dangerous Code

- Be careful with how values are returned from a thread

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

  - When the variable *r* returns, it is automatically de-allocated

Multiple Threads

- Pthreads code for joining 10 threads

```
#define NUM_THREADS  10

// an array of threads to be joined upon
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_create(&workers[i], NULL, thread_worker, (void *)arg);

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Agenda

- Overview
- Parallelism and Concurrency
- Multithreading Models
- Thread Creation
- Thread Cancellation & Some Issues
- Implicit Threading

Pthread Cancel

- `pthread_join()`: Send a cancellation request to a thread

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

- This function sends a cancellation request to the thread
- Whether and when the target thread
  reacts to the cancellation request depends on two attributes that are under the control of that thread: its cancelability state and type.
- Cancelability state is determined by `pthread_setcancelstate()`
- Cancellation type is determined by `pthread_setcanceltype()`
- *thread*: thread ID to cancel
- Return value
  - Success: 0
  - Error: nonzero erro number

Example: Pthread Cancel

thread_cancel.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>

int counter = 0;          // Counter
pthread_t tmp_thread;     // Thread ID for func2

void* func1(void* arg) {
    while (1) {
        printf("Thread #1 (counter=%d)\n", counter);
        if (counter == 5) {
            // for cancel thread_two
            pthread_cancel(tmp_thread);
            // for exit from thread_one
            pthread_exit(NULL);
        }
        sleep(1);   // sleep 1 second
    }
}

void* func2(void* arg) {
    // get thread ID
    tmp_thread = pthread_self();
    while (1) {
        printf("Thread #2 (counter=%d)\n", counter);
        counter++;
        sleep(1); // sleep 1 second
    }
}
```

```c
int main() {
     pthread_t thread_one, thread_two;

    // create thread_one and thread_two
    pthread_create(&thread_one, NULL, func1, NULL);
    pthread_create(&thread_two, NULL, func2, NULL);

    // waiting for when threads are completed
    pthread_join(thread_one, NULL);
    pthread_join(thread_two, NULL);

    return 0;
}
```

```
yunmin@yunmin:~/ch4$ gcc thread_cancel.c -o thread_cancel -lpthread
yunmin@yunmin:~/ch4$ ./thread_cancel
Thread #2 (counter=0)
Thread #1 (counter=1)
Thread #2 (counter=1)
Thread #1 (counter=2)
Thread #1 (counter=2)
Thread #2 (counter=2)
Thread #1 (counter=3)
Thread #2 (counter=3)
Thread #2 (counter=4)
Thread #1 (counter=5)
```

Thread Cancellation

- Problem with thread cancellation
  - A thread share the resource with other threads

    cf. A process has its own resource.

    → A thread can be cancelled while it updates data shared with other threads
- If thread has cancellation disabled, cancellation remains pending until thread enables it.
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

Thread Cancellation

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- Setting thread cancellation type
  - `pthread_t pthread_setcanceltype(int type, int *oldtype);`
    - *type*: `PTHREAD_CANCEL_ASYNCHRONOUS` (Asynchronous cancellation) or `PTHREAD_CANCEL_DEFERRED` (Deferred cancellation)
    - Default type is deferred
      - Cancellation only occurs when thread reaches **cancellation point** i.e. `pthread_testcancel()`, then cleanup handler is invoked
      - Safer than asynchronous cancellation

- Enabling/disabling thread cancelation
  - `pthread_t pthread_setcancelstate(int state, int *oldstate);`
    - *state*: `PTHREAD_CANCEL_DISABLE` or `PTHREAD_CANCEL_ENABLE`
    - `PTHREAD_CANCEL_DISABLE`: cancellation remains pending until thread enables it

Example: Thread Cancellation

thread_setcancel.c

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

unsigned int counter;

void* threadFunction(void* arg) {
    pthread_setcanceltype(*(int*)arg, NULL);
    counter = 0;
    while (1) {
        printf("Running... %u \n", counter);
        while (counter < 4000000000) {
            counter++;
            // if (counter % 500000000 == 0)
            //     pthread_testcancel();
        }
    }
    return NULL;
}
```

```
yunmin@yunmin:~/ch4$ ./thread_setcancel
Maximum Value of Unsigned Long=4294967295
Running... 0
Thread with asynchronous cancellation stopped. counter=2585900876
Running... 0
Running... 4000000000
Thread with deferred cancellation stopped. counter=4000000000
```

```c
int main() {
    pthread_t thread;
    int async = PTHREAD_CANCEL_ASYNCHRONOUS;
    int deferred = PTHREAD_CANCEL_DEFERRED;
    printf("Maximum Value of Unsigned Long=%u \n", UINT_MAX);

    // Asynchronous cancellation
    pthread_create(&thread, NULL, threadFunction, &async);
    sleep(3);
    pthread_cancel(thread);
    pthread_join(thread, NULL);
    printf("Thread with asynchronous cancellation stopped. "
           "counter=%u \n", counter);

    // Deferred cancellation
    pthread_create(&thread, NULL, threadFunction, &deferred);
    sleep(3);
    pthread_cancel(thread);
    pthread_join(thread, NULL);
    printf("Thread with deferred cancellation stopped. "
           "counter=%u \n", counter);
    return 0;
}
```

Thread-Local Storage

- In a process, all threads share global variables
- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data

```
__thread int tls;        // on pthread
```

  - Each thread has its own 'int tls' variable
  - Different from local variables
    - Local variables visible only during single function invocation
    - TLS visible across function invocations
  - Similar to static data
    - TLS is unique to each thread
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Thread-Local Storage in pthread

thread_tls.c

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

#define THREADS 3

__thread int tls;
int global;

void *func(void *arg)
{
    int num = *((int*)arg);
    tls = num;
    global = num;
    sleep(1);
    printf("Thread = %d tls = %d global = %d\n",
           num, tls, global);
}
```

```c
int main() {
    int ret;
    pthread_t thread[THREADS];
    int num;

    for (num = 0; num < THREADS; num++) {
        ret = pthread_create(&thread[num], NULL,
                             &func, (void*)&num);
        if (ret) {
            printf("error pthread_create\n");
            exit(1);
        }
    }

    for (num = 0; num < THREADS; num++) {
        ret = pthread_join(thread[num], NULL);
        if (ret) {
            printf("error pthread_join\n");
            exit(1);
        }
    }
    return 0;
}
```

fork() and exec()

- `fork()` on multithreaded process
  - Duplicates all threads in the process?
  - Duplicates only corresponding thread?

  → UNIX supports two versions of fork
    - fork(), fork1()

- `exec()` on multithreaded process
  - Replace entire process including all threads

Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)
  - Flags control behavior

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.

- `struct task_struct` points to process data structures (shared or unique)

Agenda

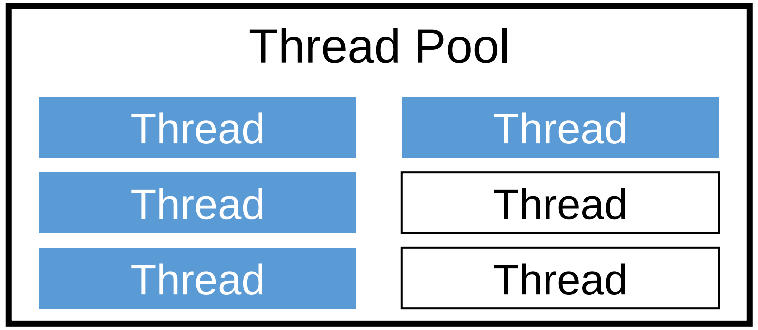
- Overview
- Parallelism and Concurrency
- Multithreading Models
- Thread Creation
- Thread Cancellation & Some Issues
- Implicit Threading

Implicit Threading

- **Implicit Threading**: Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread pools
  - Fork Join model
  - OpenMP
- Other methods include Grand Central Dispatch and Microsoft Threading Building Blocks (TBB), etc

Thread Pools

- 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 execute after a time delay or to execute periodically



Java Thread Pools

- Supported by Java executor framework in `java.util.concurrent` package
- Three factory methods for creating thread pools in Executors class:
  - `static ExecutorService newSingleThreadExecutor()`
    - Creates a pool of size 1
  - `static ExecutorService newFixedThreadPool(int size)`
    - Creates a thread pool with a specified number of threads
  - `static ExecutorService newCachedThreadPool()`
    - Creates an unbounded thread pool, reusing in many instances

OpenMP

- Provides support for parallel programming in shared-memory environments
  - Set of compiler directives and an API for C, C++, FORTRAN
  - Identifies parallel regions: blocks of code that can run in parallel
- Create as many threads as there are cores

```
#pragma omp parallel          // each runs the statement
     printf("Hello, World!\n");
```

- Run for loop in parallel

```
#pragma omp parallel for      // unroll loop over cores
     for(i=0; i<N; i++) {
         c[i] = a[i] + b[i];
     }
```

Example: OpenMP

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

int main(int argc, char *argv[])
{
    /* sequential code */
    #pragma omp parallel
    {
        printf("I am a parallel region (thread=%d)\n", omp_get_thread_num());
    }

    /* sequential code */
    return 0;
}
```

Results (depends on your machine)

```
yunmin@mcnl-server:~/workspace/os/ch4$ gcc openmp.c -o openmp -fopenmp
yunmin@mcnl-server:~/workspace/os/ch4$ ./openmp
I am a parallel region (thread=81)
I am a parallel region (thread=18)
I am a parallel region (thread=85)
I am a parallel region (thread=54)
I am a parallel region (thread=15)
I am a parallel region (thread=41)
I am a parallel region (thread=11)
I am a parallel region (thread=94)
I am a parallel region (thread=50)
I am a parallel region (thread=58)
```