


Operating system

Chapter 4: Threads



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Objectives

- To introduce the notion of a thread
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

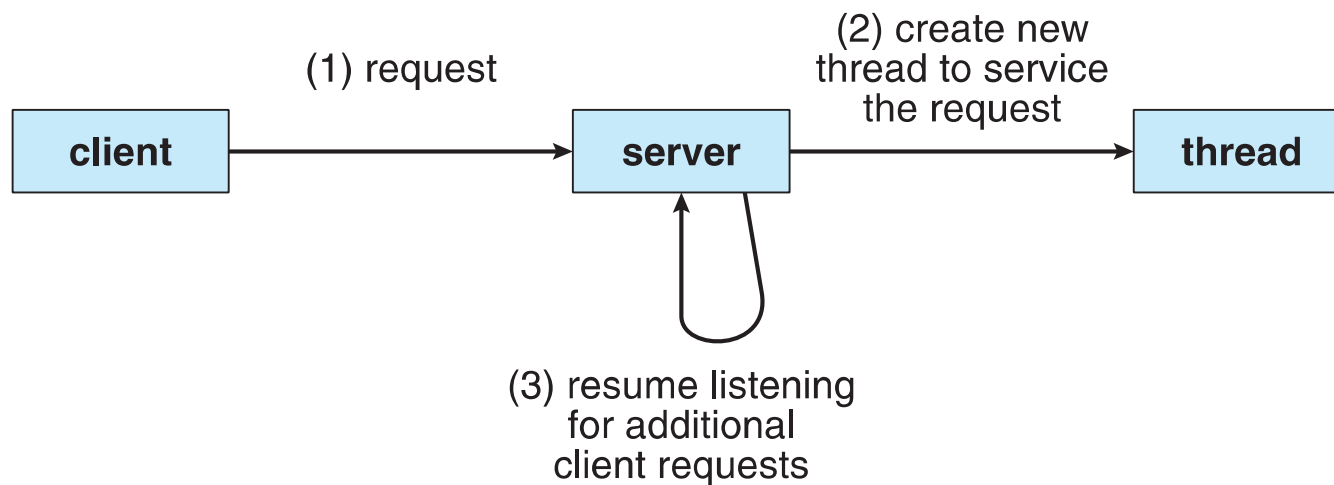
Overview



Motivation

- Most modern applications are multitask
- Multiple tasks with the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- V.V.I
- **Process** creation is **heavy-weight** while **thread** creation is **light-weight**
- Can simplify code, increase efficiency
- Threads run within application
- Kernels are generally multithreaded

Multithreaded Server Architecture



When a server receives a message, it services the message using a separate thread. This allows the server to service several concurrent requests.

Thread Definition

- A basic unit of CPU utilization

V.V.I

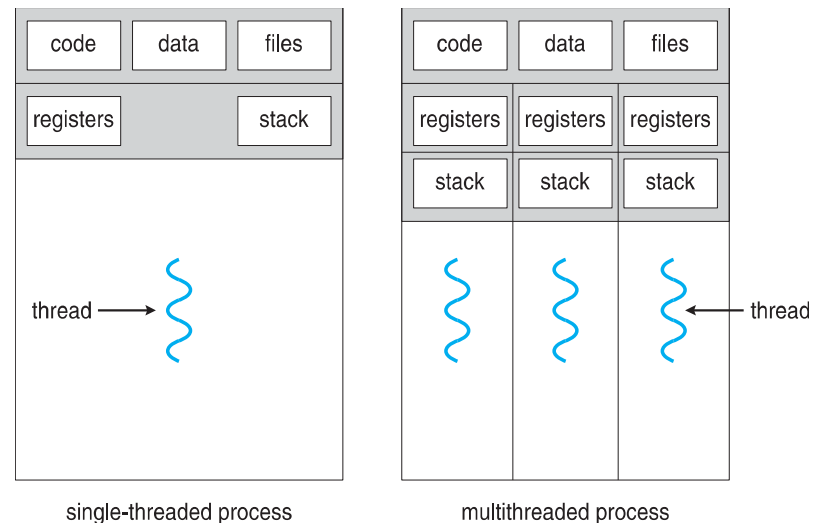
- Private: Thread ID, program counter, register set, stack
- Shared: code section, data section, OS resources (IO & file)

- Examples:

- Web browsers
- Word processors
- Database engines
- RPC!

- Versus Process:

- Time consuming
- Resource intensive

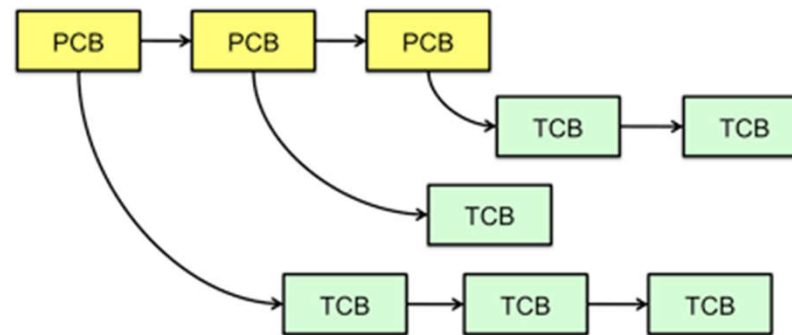


Per Thread Descriptor (Kernel Supported Threads)

- Each Thread has a *Thread Control Block (TCB)*
 - Execution State: CPU registers, program counter (PC), pointer to stack (SP)
 - Scheduling info: state, priority, CPU time
 - Various Pointers (for implementing scheduling queues)
 - Pointer to enclosing process (PCB) – user threads
 - ... (add stuff as you find a need)
- OS Keeps track of TCBs in “kernel memory”
 - In Array, or Linked List, or ...
 - I/O state (file descriptors, network connections, etc)

Multithreaded Processes

- Process Control Block (PCBs) points to multiple Thread Control Blocks (TCBs):

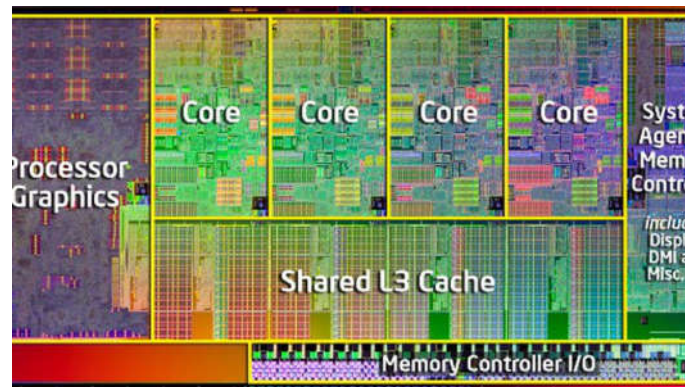


- Switching threads within a block is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables

Advantages of using threads

- Responsiveness
 - Allowing a program to continue running even part of it is blocked or lengthy
- Resource sharing
 - Memory, resources
- Economy
 - Fast
- Scalability
 - Threads may be running in parallel on processing cores

Multicore programming



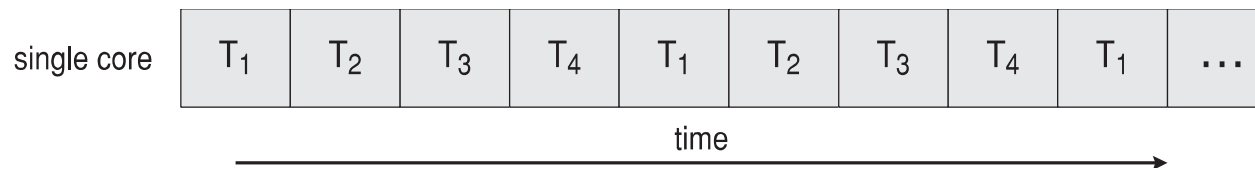
Multicore programming

Parallelism implies a system can perform more than one task simultaneously

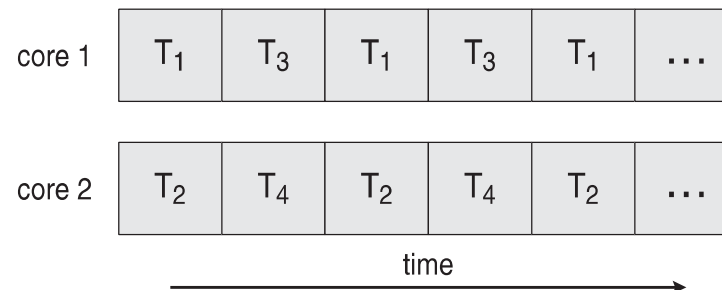
Concurrency supports more than one task making progress

✓ Single processor / core, scheduler providing concurrency

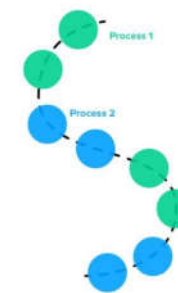
Concurrent execution on single-core system:



Parallelism on a multi-core system:

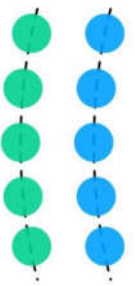


Concurrency



vs

Parallelism



Multicore programming

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

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

AMDAHL'S LAW

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

S : the portion of the application that must be performed serially

N : processing cores

Types of parallelism

- Types of parallelism
 - **Data parallelism**
 - distributing subsets of the same data across multiple computing cores and performing the same operation on each core.
 - **Task parallelism**
 - distributing not data but tasks (threads) across multiple computing cores.
- As # of threads grows, so does architectural support for threading
 - CPUs have cores as well as **hardware threads**
 - Consider **Oracle SPARC T4** with 8 cores, and 8 hardware threads per core



Multithreading Models

User threads and kernel threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
 - POSIX *Pthreads*
 - Windows threads
 - Java threads
- **Kernel threads** - Supported by the Kernel
 - ✓ Windows
 - ✓ Solaris
 - ✓ Linux
 - ✓ Tru64 UNIX
 - ✓ Mac OS X
- **Asynchronous** vs. **synchronous** threading
 - Parent & child threads

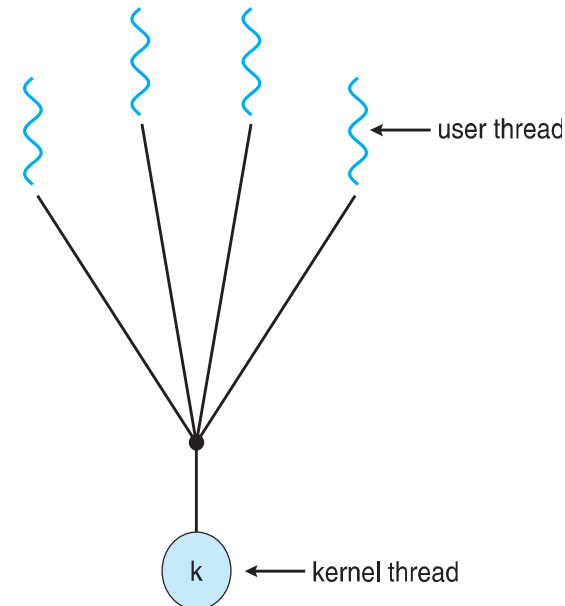
Multithreading Models

A relationship must exist between user threads and kernel threads.

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

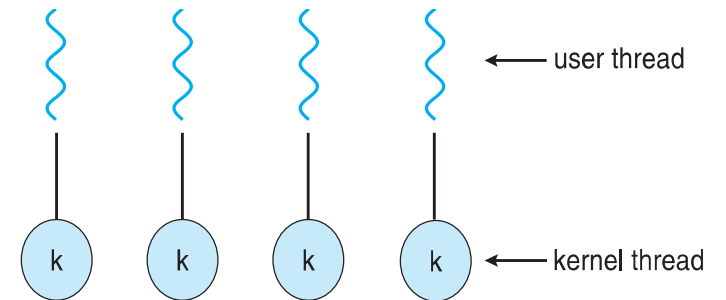
Many(user level)-to-one(kernel level)

- 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
- Few systems currently use this model
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads
- Used in very few systems.



One(user level)-to-one(kernel level)

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one

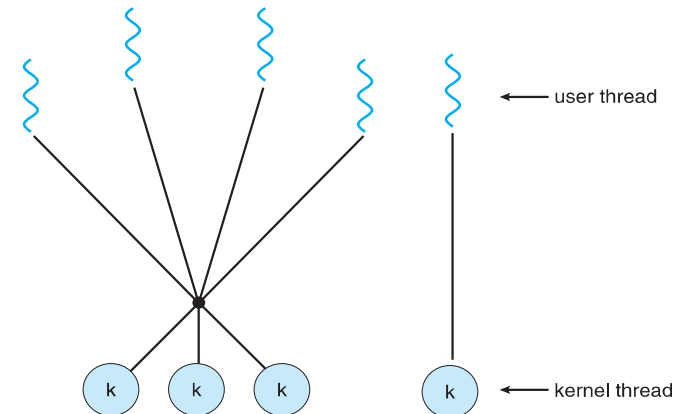
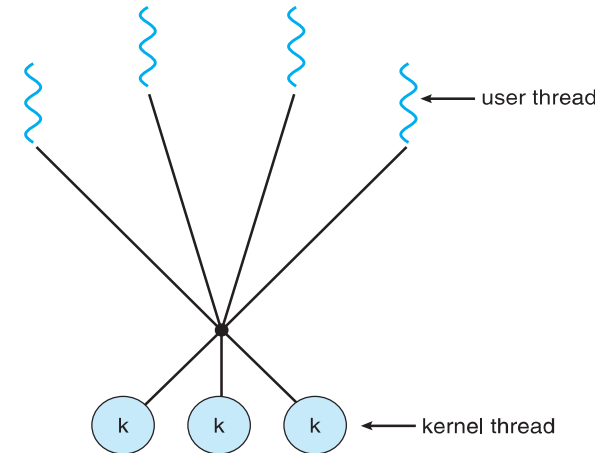


- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux
 - Solaris 9 and later

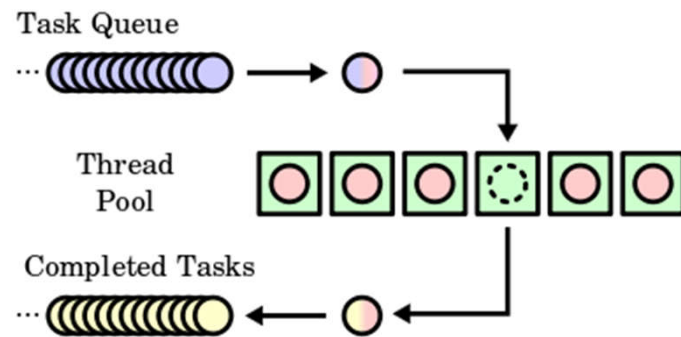
For linux:
`/proc/sys/kernel/threads-max`

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



Thread Operation

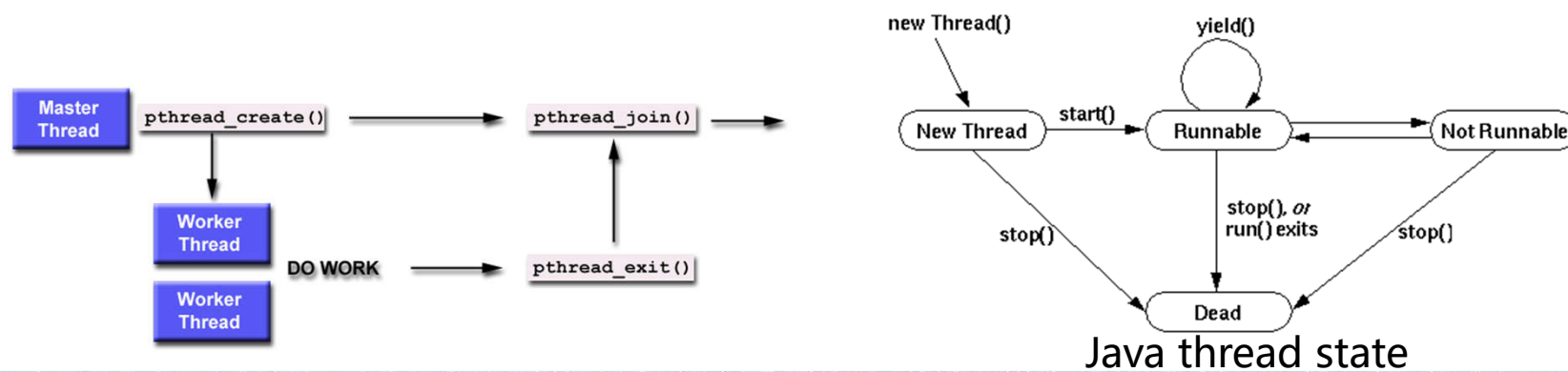


Thread Libraries

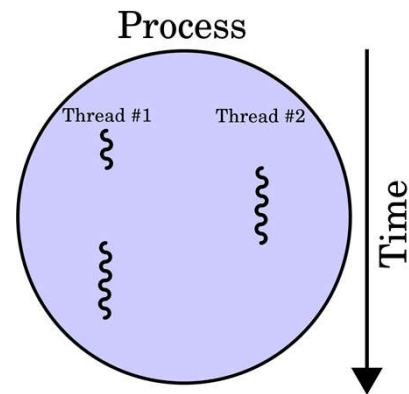
- **Thread library** provides programmer with API for creating and managing threads
 - POSIX **Pthreads**(**kernel-level lib, user-level lib**)
 - Win32 threads(**kernel-level lib**)
 - Java threads(**depend on host system**)
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS

Thread operations and states

- **Spawn**
 - When a new process is spawned, a thread for that process is also spawned
- **Running**
 - When the thread scheduler assign the thread with processor, it comes in RUNNING
- **Block**
 - When a thread needs to wait for an event, it will block
- **Unblock**
 - When the event for which a thread is blocked occurs, the thread is moved to the Ready queue
- **Finish**
 - When a thread completes, its register context and stacks are deallocate



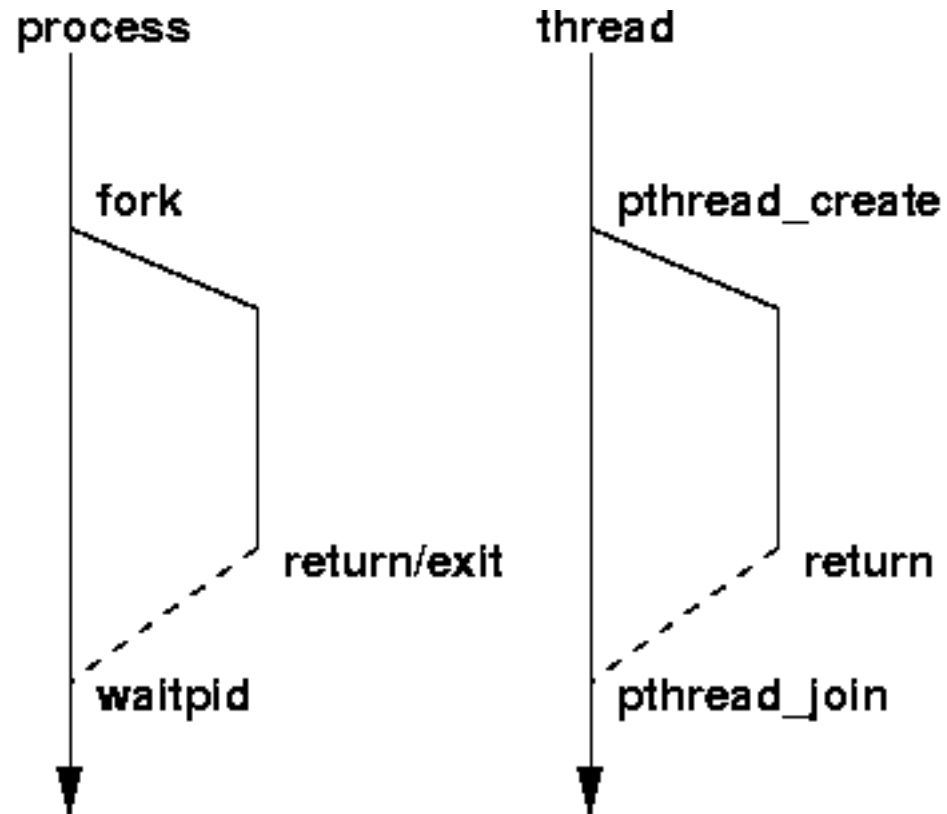
Parallel Programming using Pthread



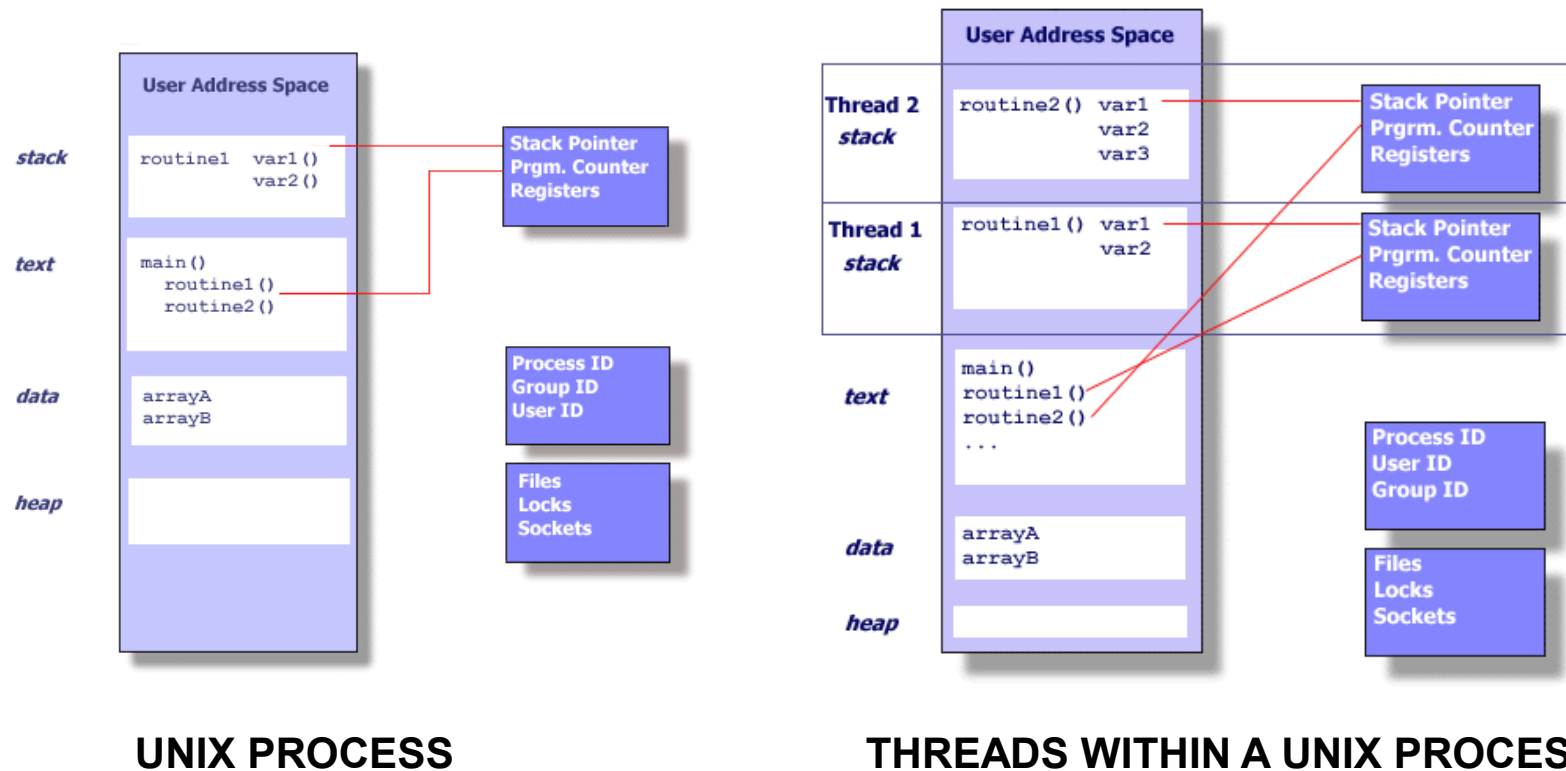
Overview of POSIX Threads

- *POSIX: Portable Operating System Interface for UNIX*
 - Interface to Operating System utilities
- *PThreads: The POSIX threading interface*
 - System calls to create and synchronize threads
 - compile a c program with `gcc -lpthread`
- *PThreads contain support for*
 - Creating parallelism and synchronization
 - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

Creation of Unix processes vs. Pthreads



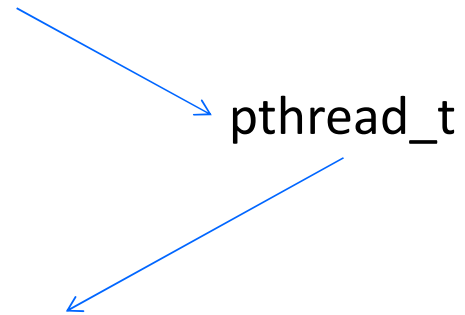
processes vs. Pthreads



THREADS WITHIN A UNIX PROCESS

C function for starting a thread

pthread.h



*One object for
each thread.*

```
int pthread_create (  
    pthread_t* thread_p    /* out */,  
    const pthread_attr_t* attr_p /* in */,  
    void* (*start_routine) ( void ) /* in */,  
    void* arg_p /* in */ );
```

A closer look (1)

```
int pthread_create (  
    pthread_t* thread_p /* out */,  
    const pthread_attr_t* attr_p /* in */,  
    void* (*start_routine) ( void ) /* in */,  
    void* arg_p /* in */ );
```

We won't be using, so we just pass NULL.

Allocate before calling.

A closer look (2)

```
int pthread_create (  
    pthread_t* thread_p /* out */,  
    const pthread_attr_t* attr_p /* in */,  
    void* (*start_routine) ( void ) /* in */,  
    void* arg_p /* in */ );
```

Pointer to the argument that should
be passed to the function *start_routine*.

The function that the thread is to run.

Function started by pthread_create

- Prototype:
`void* thread_function (void* args_p) ;`
- Void* can be cast to any pointer type in C.
- So args_p can point to a list containing one or more values needed by thread_function.
- Similarly, the return value of thread_function can point to a list of one or more values.

Wait for Completion of Threads

```
pthread_join(pthread_t *thread, void **result);
```

- Wait for specified thread to finish. Place exit value into *result.
- We call the function `pthread_join` once for each thread.
- A single call to `pthread_join` will wait for the thread associated with the `pthread_t` object to complete.

Some More Pthread Functions

- `pthread_yield()` ;
 - Informs the scheduler that the thread is willing to yield
- `pthread_exit(void *value)` ;
 - Exit thread and pass value to joining thread (if exists)

Others:

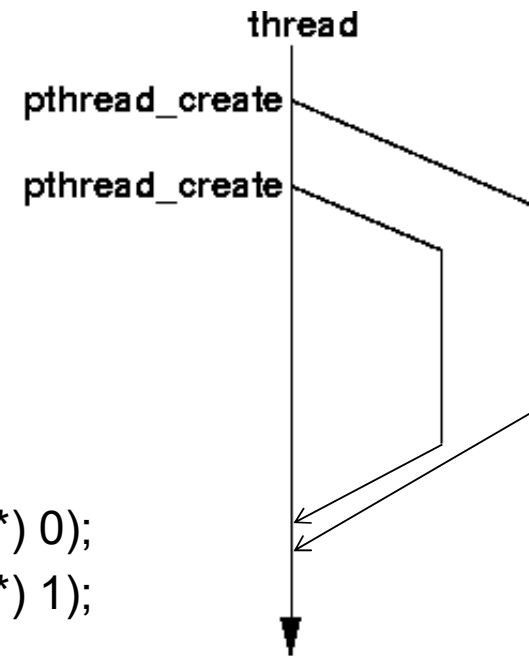
- `pthread_t me; me = pthread_self()` ;
 - Allows a pthread to obtain its own identifier pthread_t thread;
- Synchronizing access to shared variables
 - `pthread_mutex_init`, `pthread_mutex_[un]lock`
 - `pthread_cond_init`, `pthread_cond_[timed]wait`

Example of Pthreads with join

```
#include <pthread.h>
#include <stdio.h>
void *PrintHello(void * id){
    printf("Hello from thread %d\n", id);
}

void main (){
    pthread_t thread0, thread1;
    pthread_create(&thread0, NULL, PrintHello, (void *) 0);
    pthread_create(&thread1, NULL, PrintHello, (void *) 1);
    pthread_join(thread0, NULL);
    pthread_join(thread1, NULL);
}
```

```
gcc -g -o pth_hello pth_hello . c -lpthread
```



Pthread: **POSIX** thread

$$sum = \sum_{i=0}^N i$$

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

    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }
}
```

```
/* get the default attributes */
pthread_attr_t 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);
}

/* The thread will begin control 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);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

Windows multithread C program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    if (argc != 2) {
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr, "An integer >= 0 is required\n");
        return -1;
    }
}
```

```
/* create the thread */
ThreadHandle = CreateThread(
    NULL, /* default security attributes */
    0, /* default stack size */
    Summation, /* thread function */
    &Param, /* parameter to thread function */
    0, /* default creation flags */
    &ThreadId); /* returns the thread identifier */
```

```
if (ThreadHandle != NULL) {
    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}
```


Java thread programming

```
class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```

```
public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println
                        ("The sum of "+upper+" is "+sumObject.getSum());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Usage: Summation <integer value>");
    }
}
```

Implicit threading

- Three methods explored

- Thread Pools (Win)

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

- OpenMP (C lib)

```
#pragma omp parallel for
```

- Grand Central Dispatch (Mac OS, iOS)

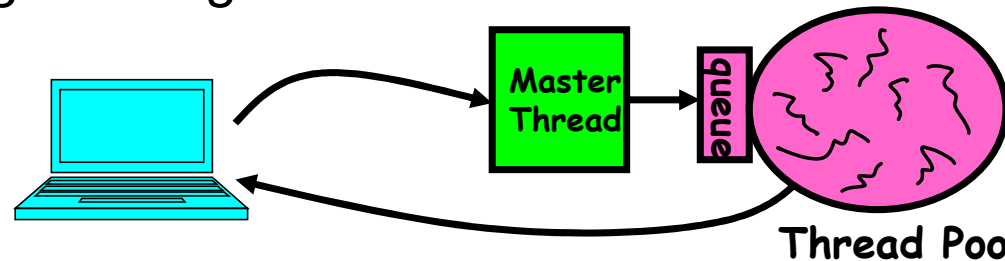
```
Block is in "^{" - ^{ printf("I am a block"); }
```

- Other Approaches

- Intel's TBB, Java's concurrent package

Thread Pools

- Problem with previous version: Unbounded Threads
 - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded “pool” of worker threads, representing the maximum level of multiprogramming

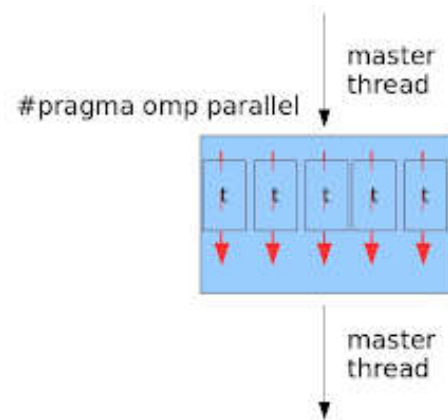


```
master() {  
    allocThreads(worker, queue);  
    while(TRUE) {  
        con=AcceptCon();  
        Enqueue(queue, con);  
        wakeUp(queue);  
    }  
}
```

```
worker(queue) {  
    while(TRUE) {  
        con=Dequeue(queue);  
        if (con==null)  
            sleepOn(queue);  
        else  
            ServiceWebPage(con);  
    }  
}
```


OpenMP

- OpenMP = **Open Multi-Parallelism**
- *OpenMP: An API for Writing Multithreaded*
- Compiler **directive** based



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

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

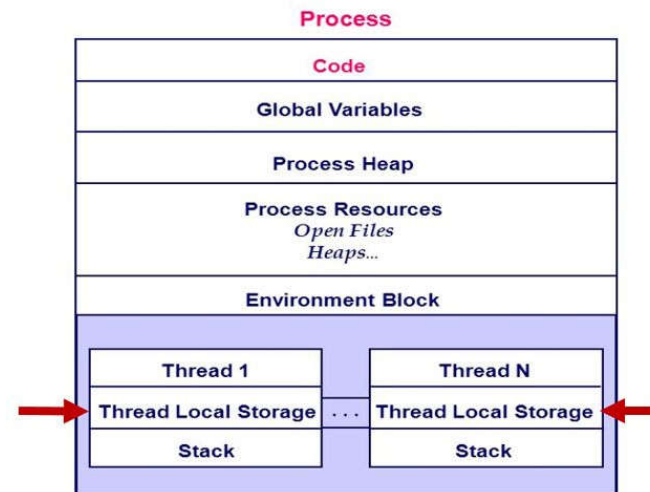
    return 0;
}
```

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

gcc -fopenmp -o test test.c

Thread-local storage

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
 - 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



Ex: Pthread Tls

```
3  #include <stdlib.h>
4  #include <pthread.h>
5
6  void foo(void); /* Functions that use the TLS data */
7
8  /*
9   * These variables are stored in thread local storage (
10  */
11  __thread int TLS_data1;
12  __thread int TLS_data2;
13
14  #define NUMTHREADS 2
15  typedef struct {
16      int data1;
17      int data2;
18  } threadparm_t;
19
20  void *thread_run(void *parm)
21  {
22      int rc;
23      threadparm_t *gData;
24      printf("Thread %.16lx: Entered\n", pthread_self());
25      gData = (threadparm_t *)parm;
26      /* Assign the value from global variable to thread s
27      TLS_data1 = gData->data1;
28      TLS_data2 = gData->data2;
29
30      foo();
31      return NULL;
32  }
```

```
34 void foo() {
35     printf("Thread %.16lx: foo(), TLS data=%d %d\n",
36         pthread_self(), TLS_data1, TLS_data2);
37 }
38
39
40 int main(int argc, char **argv)
41 {
42     pthread_t thread[NUMTHREADS];
43     int rc=0;
44     int i;
45     threadparm_t gData[NUMTHREADS];
46     printf("Enter Testcase - %s\n", argv[0]);
47     printf("Create/start %d threads\n", NUMTHREADS);
48     for (i=0; i < NUMTHREADS; i++) {
49         /* Create per-thread TLS data and pass it to the thread */
50         gData[i].data1 = i;
51         gData[i].data2 = (i+1)*2;
52         rc = pthread_create(&thread[i], NULL, thread_run, &gData[i]);
53     }
54     printf("Wait for all threads to complete, and release their resources\n");
55     for (i=0; i < NUMTHREADS; i++) {
56         rc = pthread_join(thread[i], NULL);
57     }
58     printf("Main completed\n");
59     return 0;
60 }
```

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
 1. Signal is generated by particular event
 2. Signal is delivered to a process
 3. Signal is handled by one of two signal handlers:
 - default
 - user-defined
- Every signal has **default handler** that kernel runs when handling signal
 - ✓ **User-defined signal handler** can override default
 - ✓ For single-threaded, signal delivered to process

Handling signals in multithreaded program

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process

delivering a signal to specifies the process

```
kill(pid t pid, int signal)
```

delivering a signal to specifies the process

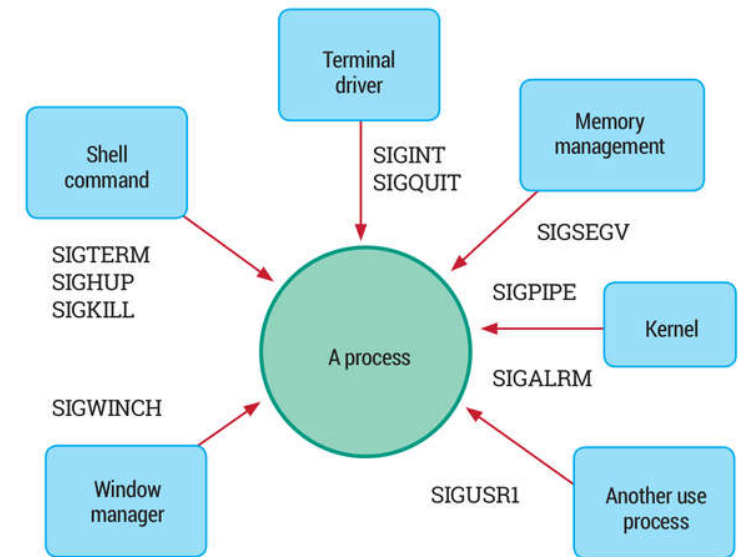
```
pthread kill(pthread t tid, int signal)
```


Linux signals

- A mechanism for the kernel and these processes to coordinate their activities.
- One way to do this is for a process to inform others when something important happens.

A **signal** is basically a one-way notification. A **signal** can be sent by the kernel to a process, by a process to another process, or a process to itself.

SIGINT	2	Interrupt a process (used by Ctrl-C)
SIGHUP	1	Hang up or shut down and restart process
SIGKILL	9	Kill the process (cannot be ignored or caught elsewhere)
SIGTERM	15	Terminate signal, (can be ignored or caught)
SIGTSTP	20	Stop the terminal (used by Ctrl-z)
SIGSTOP	19	Stop execution (cannot be caught or ignored)



EX 1.

```
int signal () (int signum, void (*func)(int))
```

The signal() will call the func function if the process receives a signal signum.

```
(base) [thzhao@admin1 process]$ ./signal
1 : Inside main function
2 : Inside main function
3 : Inside main function
4 : Inside main function
5 : Inside main function
6 : Inside main function
^C
Inside handler function
7 : Inside main function
8 : Inside main function
```

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int signum){

    //Return type of the handler function should be void
    printf("\nInside handler function\n");
}

int main(){
    signal(SIGINT,sig_handler); // Register signal handler
    for(int i=1;;i++){ //Infinite loop
        printf("%d : Inside main function\n",i);
        sleep(1); // Delay for 1 second
    }
    return 0;
}
```

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```


Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

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

If thread has cancellation disabled, cancellation remains pending until thread enables it
Default type is deferred

Cancellation only occurs when thread reaches **cancellation point**

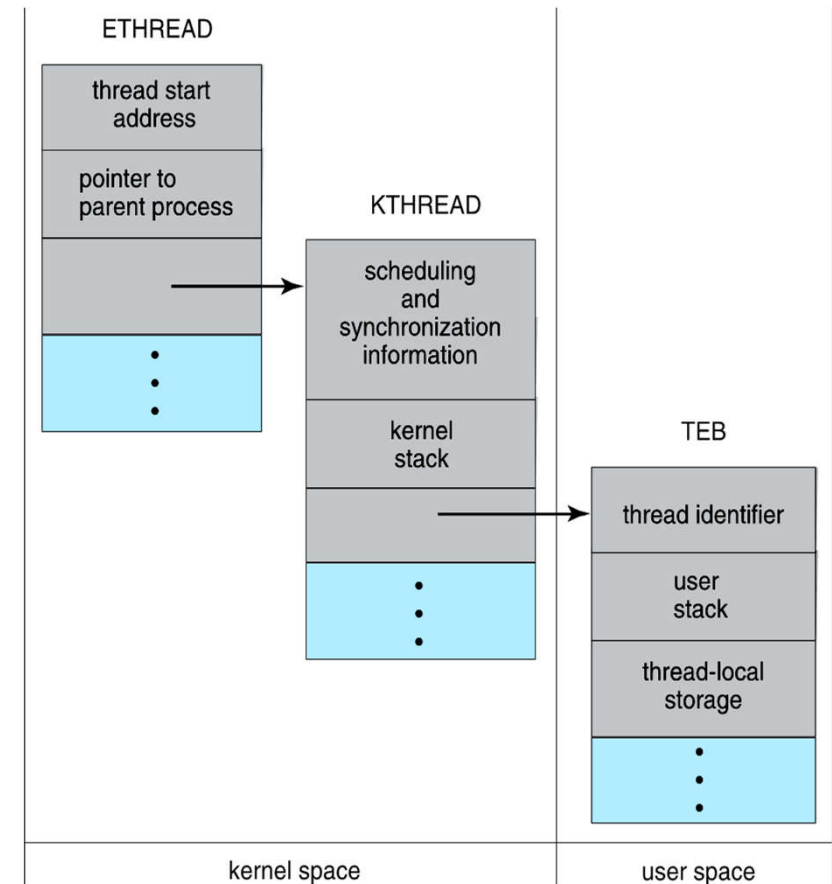
i.e. `pthread_testcancel()`

Then **cleanup handler** is invoked

On Linux systems, thread cancellation is handled through signals

Windows threads data structures

- Implements the **one-to-one** mapping, **kernel-level**
- Each thread contains
 - A **thread id**
 - **Register set** representing state of processor
 - Separate user and kernel **stacks** for when thread runs in user mode or kernel mode
 - **Private data** storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread
- Data structures:
 - **Execution thread block**, **kernel thread block** and **thread environment block**



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)

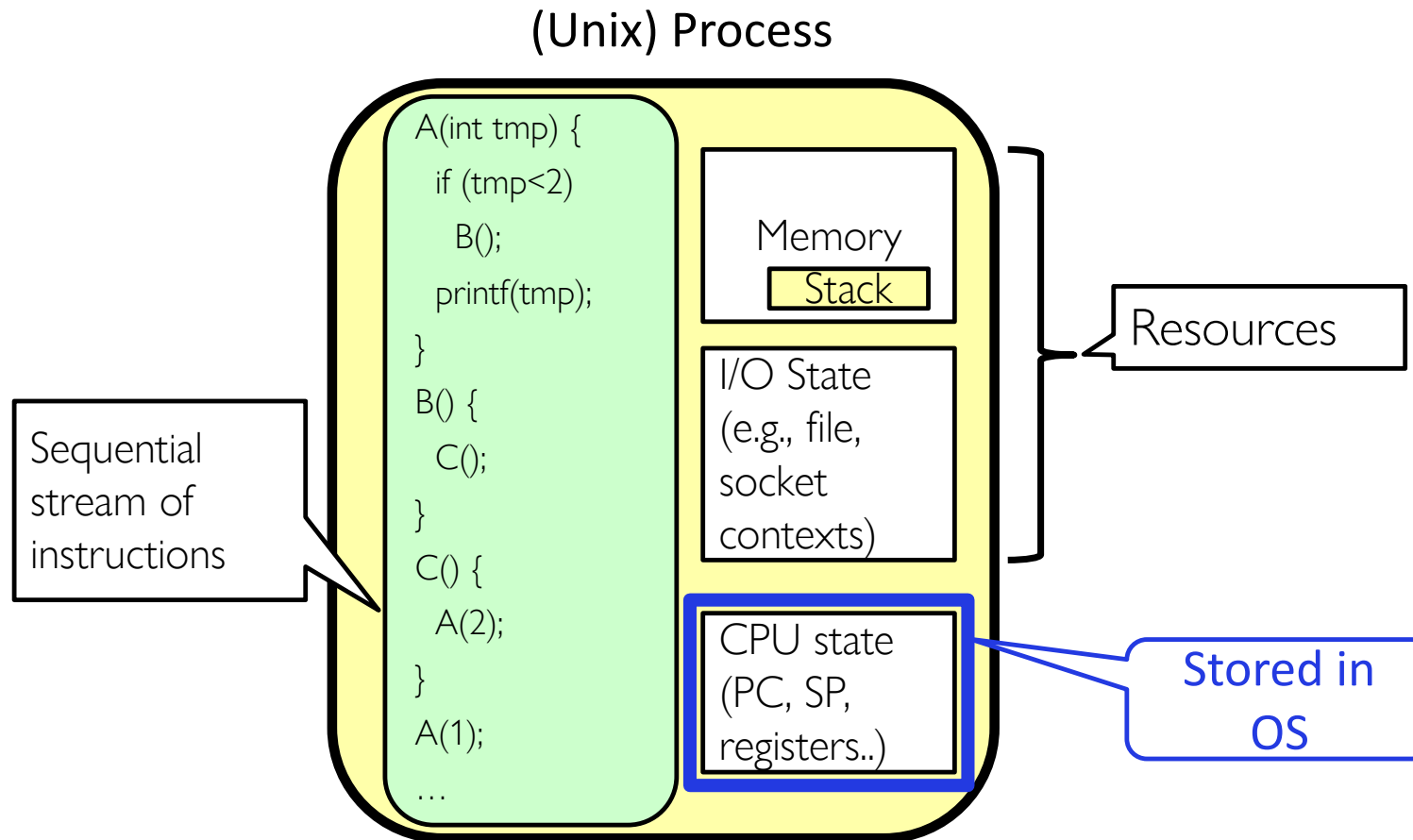
- Flags control behavior

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)

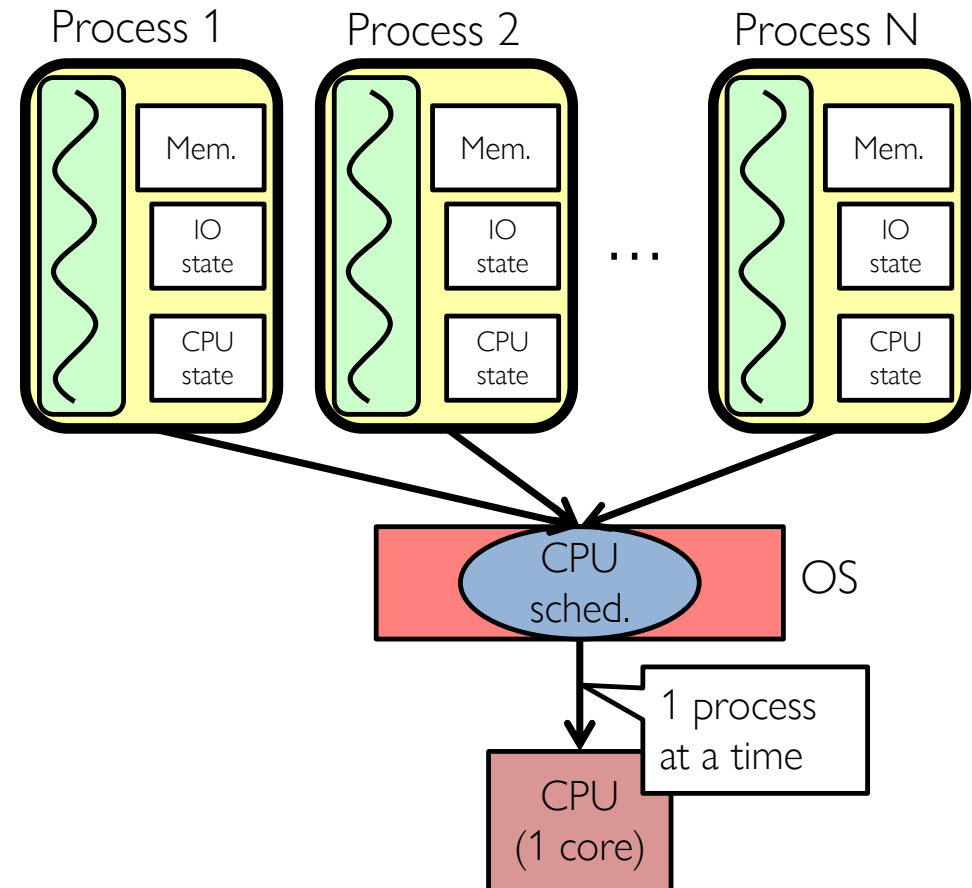
Summary

Putting it Together: Process



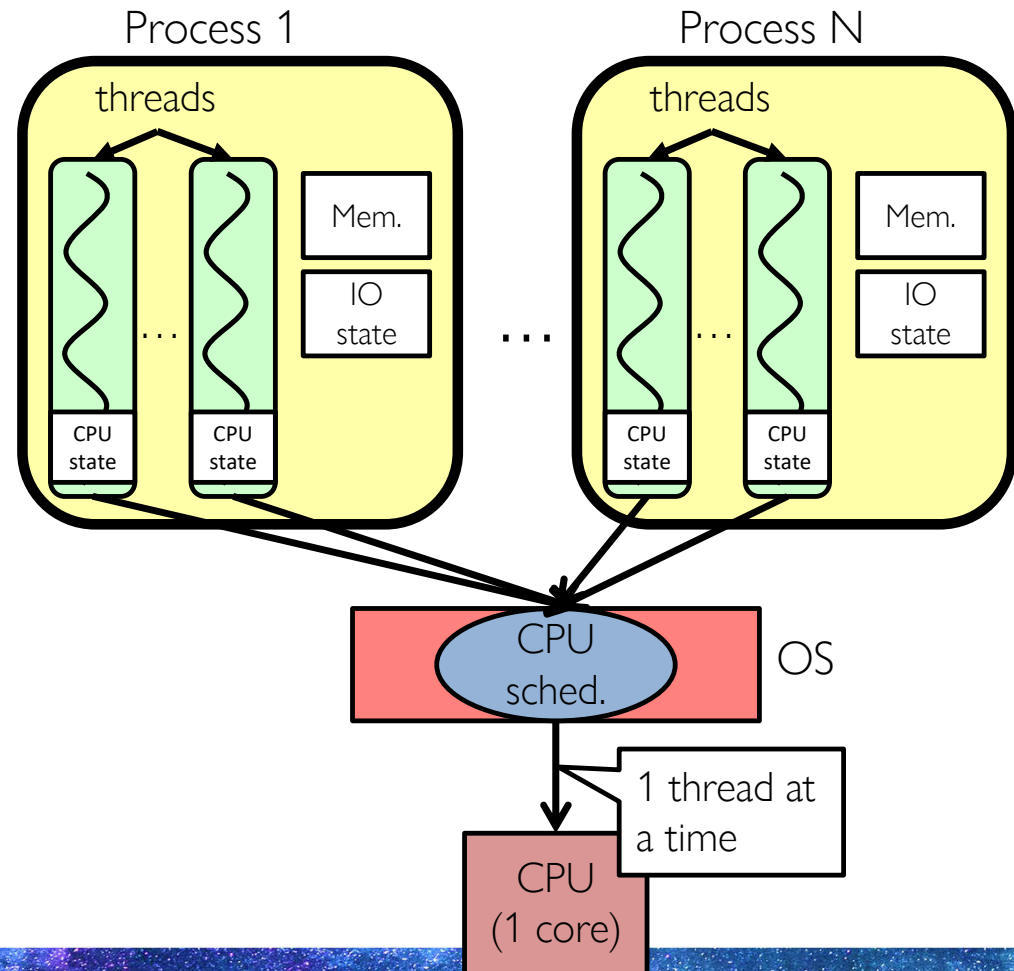
Putting it Together: Processes

- Switch overhead: **high**
 - CPU state: *low*
 - Memory/IO state: **high**
- Process creation: **high**
- Protection
 - CPU: *yes*
 - Memory/IO: *yes*
- Sharing overhead: **high** (involves at least a context switch)



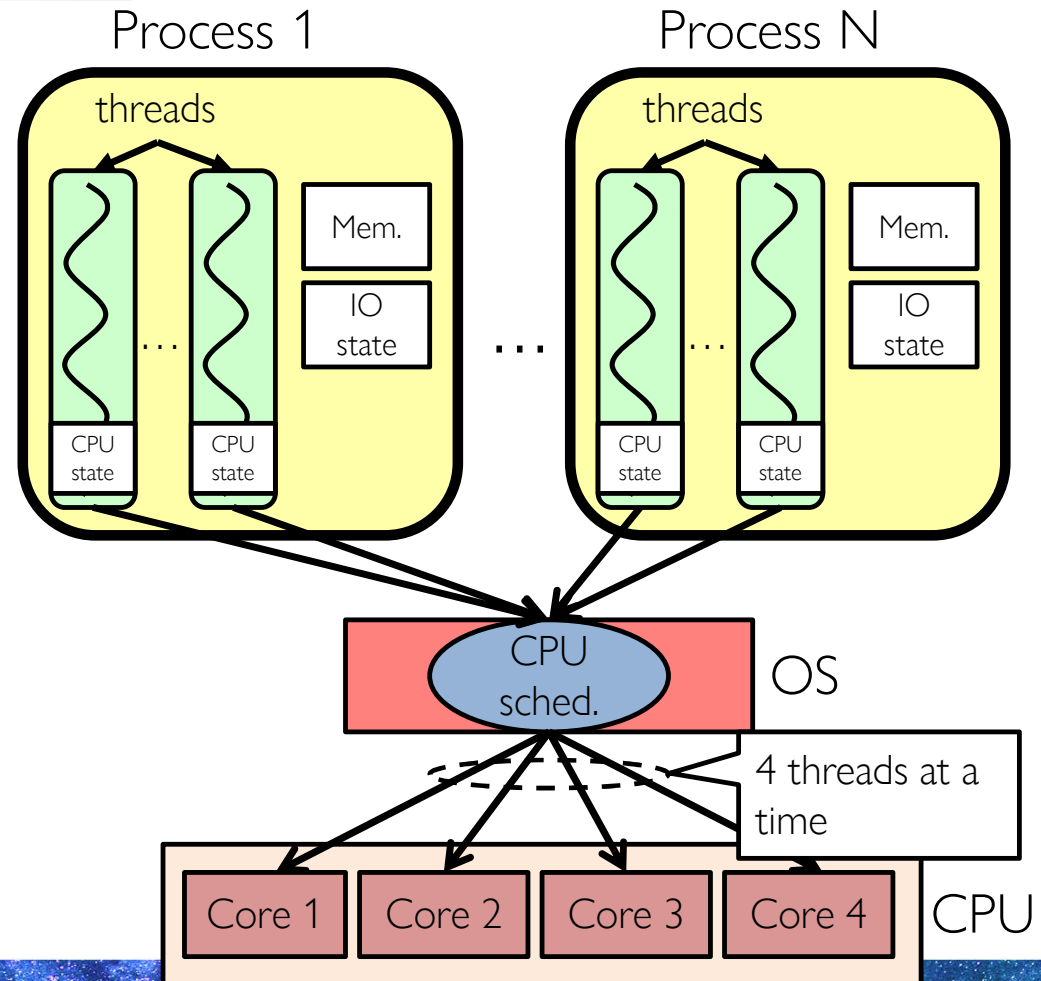
Putting it Together: Threads

- Switch overhead: **medium**
 - CPU state: *low*
- Thread creation: **medium**
- Protection
 - CPU: *yes*
 - Memory/IO: *no*
- Sharing overhead: *low(ish)*
(thread switch overhead low)



Putting it Together: Multi-Cores

- Switch overhead: *low*
(only CPU state)
- Thread creation: *low*
- Protection
 - CPU: *yes*
 - Memory/IO: *No*
- Sharing overhead: *low*
(thread switch overhead low, may not need to switch at all!)



End of Chapter 4

