



Chapter 4: Threads

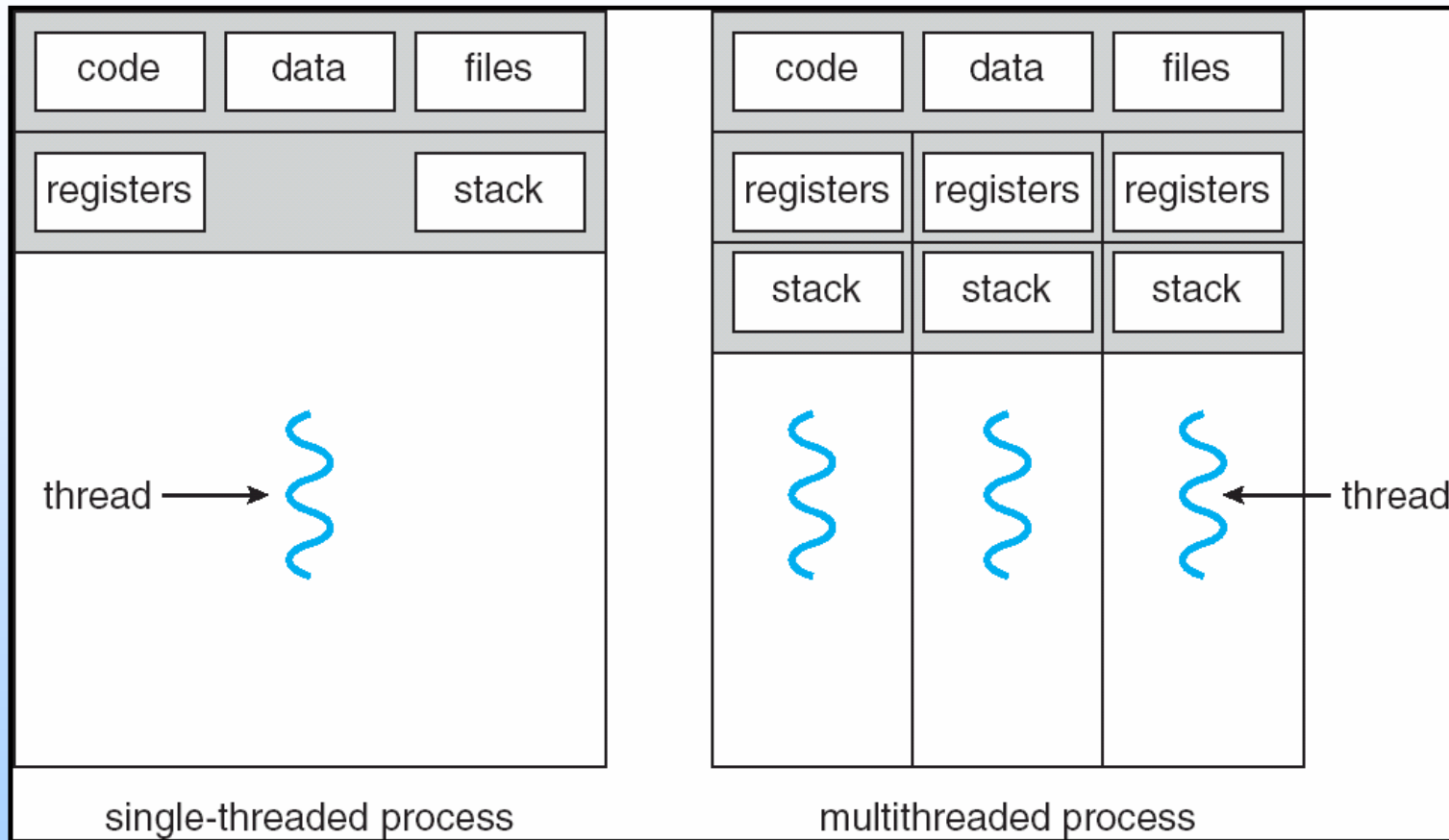


Silberschatz, Galvin and Gagne ©2005
Modified by Dimitris Margaritis, Spring 2007

Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Pthreads
- Windows XP Threads
- Linux Threads
- Java Threads

Single vs. Multithreaded Processes



Process: everything we've seen up to now

Thread: like a process, only shares memory, global variables, files, PID with other threads within the same process

Should you use threads or processes?

- Why use threads instead of processes?
 - Faster context switch (why?)
 - Easier to share data
 - Uses less memory and resources
 - Thread creation faster than process creation (30x faster in Solaris)
 - ▶ Less things to set-up
- Why use processes instead of threads?
 - Different code
 - Running on different machines
 - Different owners
 - Little communication
 - Sharing memory can lead to obscure bugs

User-level threads

- Threads can be provided at the **user** or **kernel level**
- User level: kernel knows nothing about threads
- Implemented in a library by somebody without touching the kernel
- User library handles
 - Thread creation
 - Thread deletion
 - Thread scheduling
- Benefits:
 - Faster creation and scheduling (why?)
- Drawbacks:
 - One thread blocking during I/O blocks **all** threads in process (even ready-to-run threads)

User-level threads (cont'd)

- Three primary thread libraries:
 - POSIX Pthreads
 - Win32 threads
 - Java threads

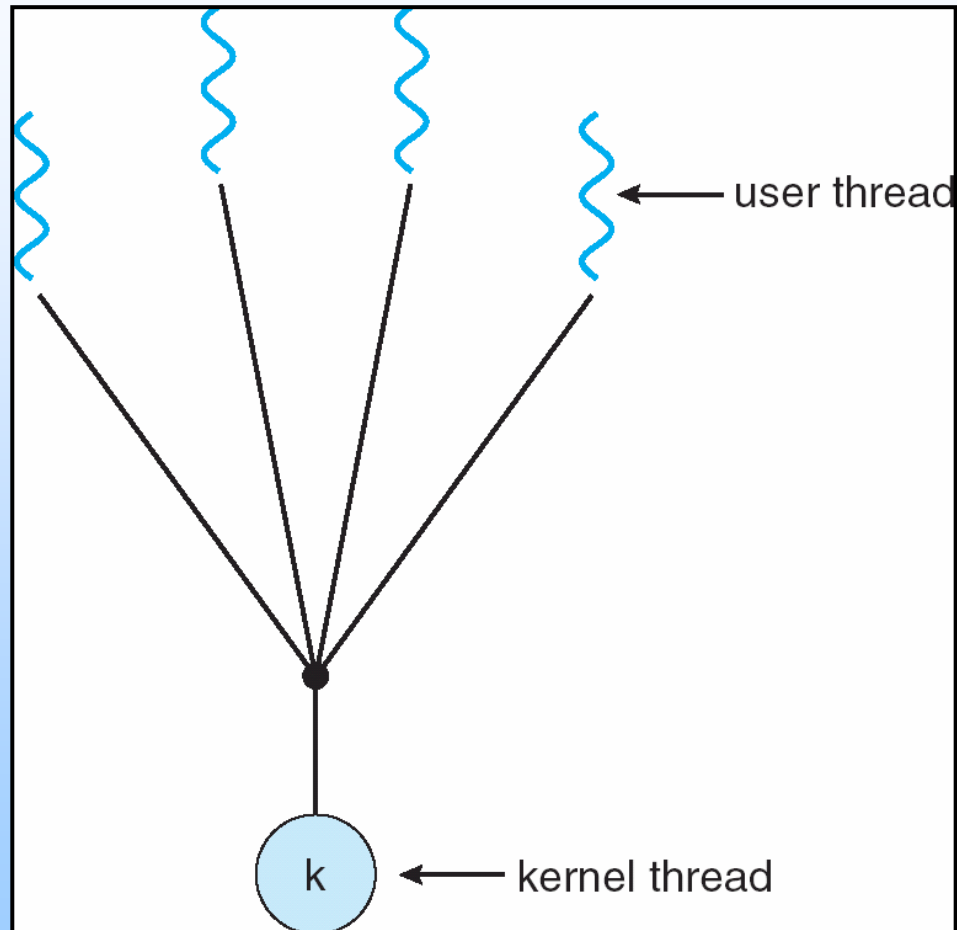
Kernel-level threads

- Kernel knows about threads
- Kernel handles thread creation, deletion, scheduling
- Benefits:
 - Kernel can schedule another thread if current one does blocking I/O
 - Kernel can schedule multiple threads on different CPUs on SMP multiprocessor
- Drawbacks:
 - Slower to schedule, create, delete than user-level
- Most modern OSes support kernel-level threads
 - Windows XP/2000
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X

Multithreading Models

- How to map kernel-level threads to user-level threads?
 - Many-to-One
 - One-to-One
 - Many-to-Many

Many-to-One Model



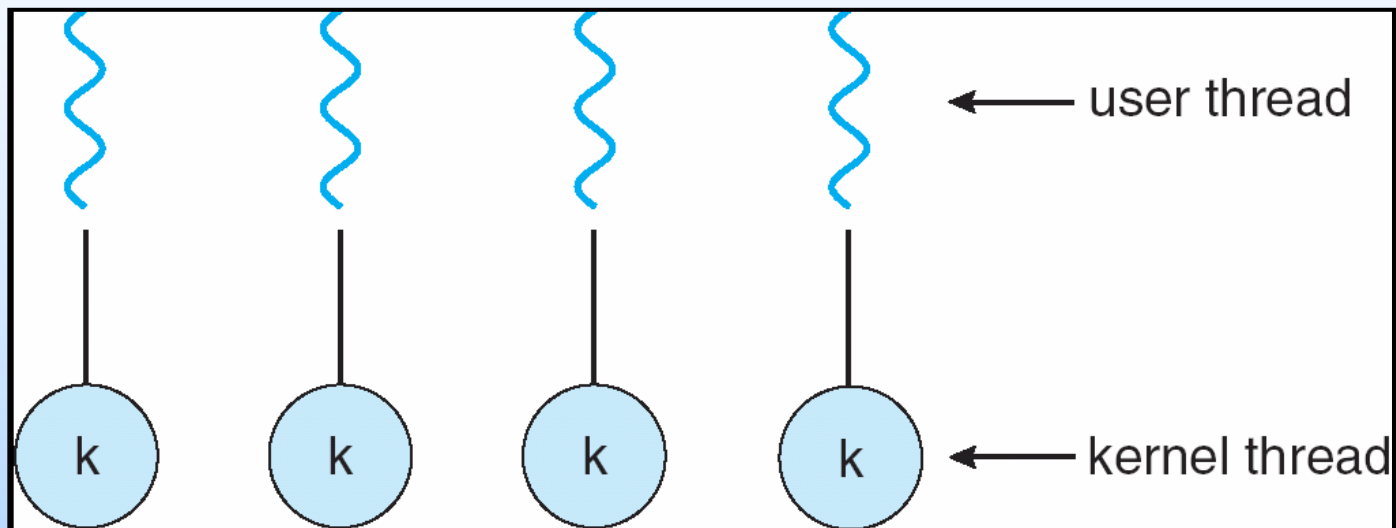
Many-to-One

- Many user-level threads mapped to single kernel thread

- Examples:
 - Solaris Green Threads
 - GNU Portable Threads

- Any disadvantages?
 - All block when one blocks
 - All run on 1 CPU

One-to-one Model



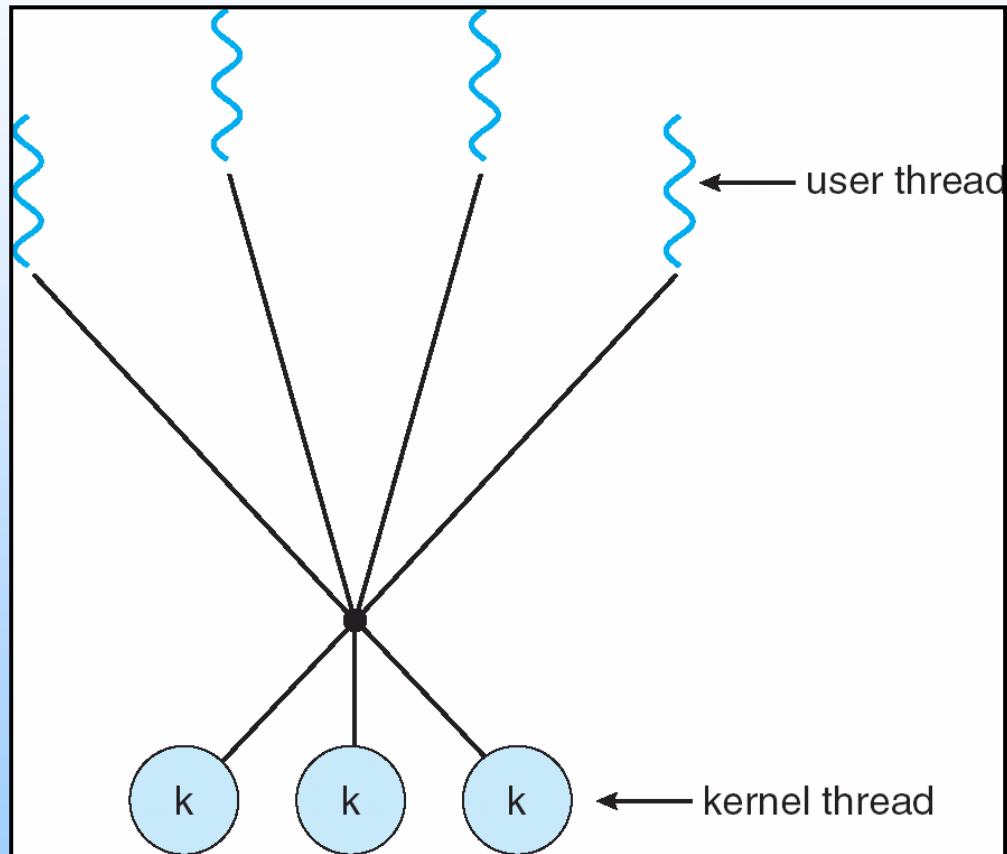
One-to-One

- Each user-level thread maps to a kernel thread

- Examples:
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 and later

- Any disadvantages?
 - Overhead for creating threads
 - Many operating systems limit number of threads

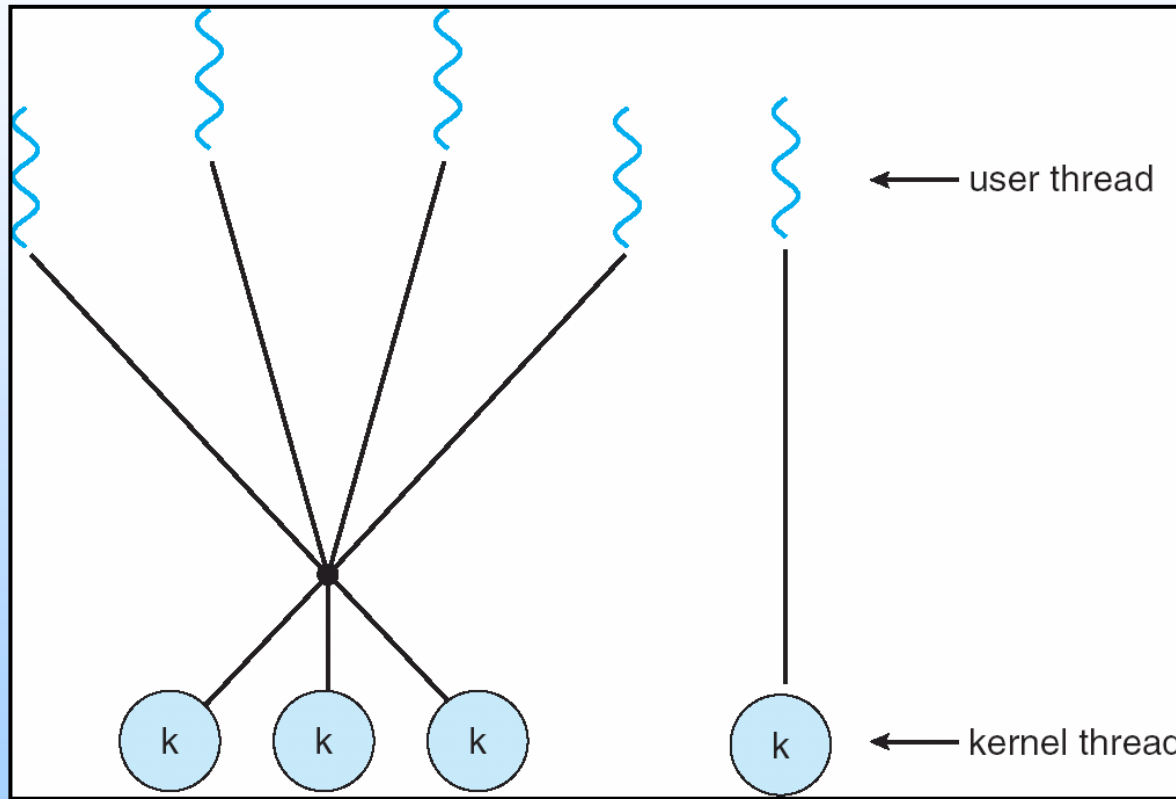
Many-to-Many Model



Many-to-Many Model

- Allows many user level threads to be mapped to smaller or equal number of kernel threads
- Allows the flexibility of choosing the number of kernel threads allocated to a process
- “Best of both worlds”
- Solaris prior to version 9

Two-level Model



Two-level Model

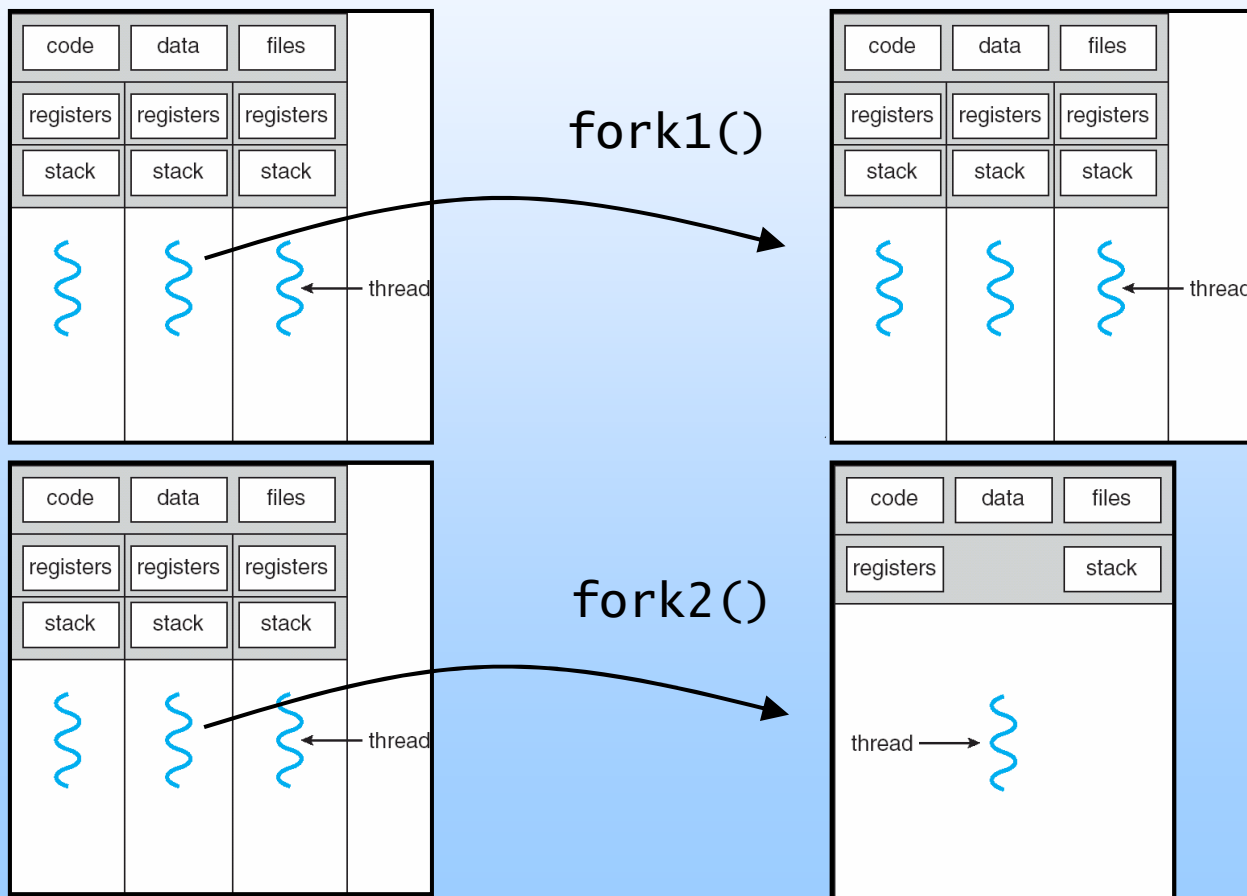
- Similar to many-to-many, but allows a user thread to be **bound** to kernel thread
- Examples
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 and earlier

Threading Issues

- Semantics of `fork()` and `exec()` system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread-specific data

Semantics of `fork()` and `exec()`

- Does `fork()` duplicate only the calling thread or all threads?
- Some UNIX systems have two versions of `fork()`
- `exec()` usually replaces all threads with new program



Thread Cancellation

- Terminating a thread before it has finished
- E.g., two cooperating threads, one discovers an error
- Two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Why is this an issue?
 - What if a thread is in the middle of
 - ▶ Allocating resources
 - ▶ Performing I/O
 - ▶ Updating a shared data structure

Thread Cancellation (cont'd)

- Essentially, deferred cancellation = “target, please cancel yourself”
 - Occurs when target checks for cancellation signal
- Allows cancellation at “safe” points
 - Called **cancellation points** in Pthreads

Signal Handling

- Signals are used in UNIX to notify a process that a particular event has occurred
- A **signal handler** is used to process signals
 1. Signal is generated by a particular event
 2. Signal is delivered to a process
 3. Signal is handled (or ignored/blocked)
- Options:
 - Deliver the signal to the thread to which the signal applies
 - ▶ Applicable with synchronous signals e.g., illegal memory access
 - Deliver the signal to **every** thread in the process
 - Deliver the signal to **certain** threads in the process
 - Assign a specific thread to receive **all signals** for the process

Thread Pools

- Motivating example: a web server running on an SMP machine
- To handle each connection:
 1. Create a new process to handle it
 - ▶ too slow, inefficient
 2. Create a new thread to handle it
- Option 2 better but still has some problems:
 - Some overhead for thread creation/deletion
 - Thread will only be used for this connection
 - Unbounded number of threads might crash web server
- Better solution: use a thread pool of (usually) fixed size

Thread Pools (cont'd)

- Threads in pool sit idle
- Request comes in:
 - Wake up a thread in pool
 - Assign it the request
 - When it completes, return it to pool
 - If no threads in pool available, wait
- Advantages:
 - Usually slightly faster to wake up an existing thread than create a new one
 - Allows the number of threads in the application to be limited by the size of the pool
 - More sophisticated systems dynamically adjust pool size

Thread-specific Data

- Allows each thread to have its own copy of data
- Useful for implementing protection
 - For example, user connects to bank's database server
 - Server process responds, has access to all accounts
 - Multiple people may be accessing their accounts at the same time
 - Thread assigned to manipulate or view user's bank account
 - Using thread-specific data limits (unintentional or erroneous) access by other threads

Pthreads

- A POSIX **standard** (IEEE 1003.1c) API for thread creation and synchronization
 - NOT an implementation
 - Implementation in different OSes may be using user or kernel threads
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- Fairly portable
- Available on pyrite
- Has man pages on Linux (check “man pthread_create”)
- To use:
 - `#include <pthread.h>`
 - Link with pthread library: `g++ prog.cc -lpthread`

Pthreads: creating threads

- Thread creation

```
pthread_create(pthread_t *th,  
              pthread_attr_t *attr,  
              void *(*start_routine)(void *),  
              void *arg);
```

- On success:

- Returns 0
- Thread identifier placed in “th”
- A new thread (with attributes “attr”, NULL means default) starts executing function “start_routine” with argument “arg”

- On failure:

- Returns non-zero error code

- Thread runs until it returns from `start_routine()` or if it calls `pthread_exit()`

Pthreads creating example

```
void *my_thread(void *arg)
{
    char *msg = (char *) arg;
    cout << "Thread says " << msg << "\n";
    return NULL;
}

...

int main()
{
    pthread_t t1, t2;
    char *msg1 = "Hello";
    char *msg2 = "world";
    pthread_create(&t1, NULL, my_thread, msg1);
    pthread_create(&t2, NULL, my_thread, msg2);

    return 0;
}
```

Pthreads: waiting

- To wait for a thread to complete:

```
pthread_join(pthread_t th, void **thread_return)
```

- `thread_return` is either NULL or address of a pointer to the buffer that holds the result of the thread
- Returns: 0 on success, -1 on failure
- Notes:
 - The resources used by thread (thread ID and stack) are deallocated AFTER another joins with it
 - Only one thread may wait for the termination of another thread (should get an error if another tries to join with the same thread)

Pthreads waiting example

```
int main()
{
    N = 0;
    pthread_t t1, t2, t3;
    cout << "Parent creating threads\n";
    pthread_create(&t1, NULL, thread, new int(1));
    pthread_create(&t2, NULL, thread, new int(2));
    pthread_create(&t3, NULL, thread, new int(3));

    cout << "Threads created\n";
    N = 3;

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    cout << "Threads are done\n";
    return 0;
}
```

Program starts here



```
int N; /* Global variable. */

void thread(void *x)
{
    int *id = (int *) x;
    while (N != *id);
    cout << "Thread " << *id << endl;
    N--;
    delete id;
    pthread_exit(0);
}
```

Program
continues



Windows XP Threads

- Implements the one-to-one mapping
- Each thread contains
 - A thread id
 - Register set
 - Separate user and kernel stacks
 - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads

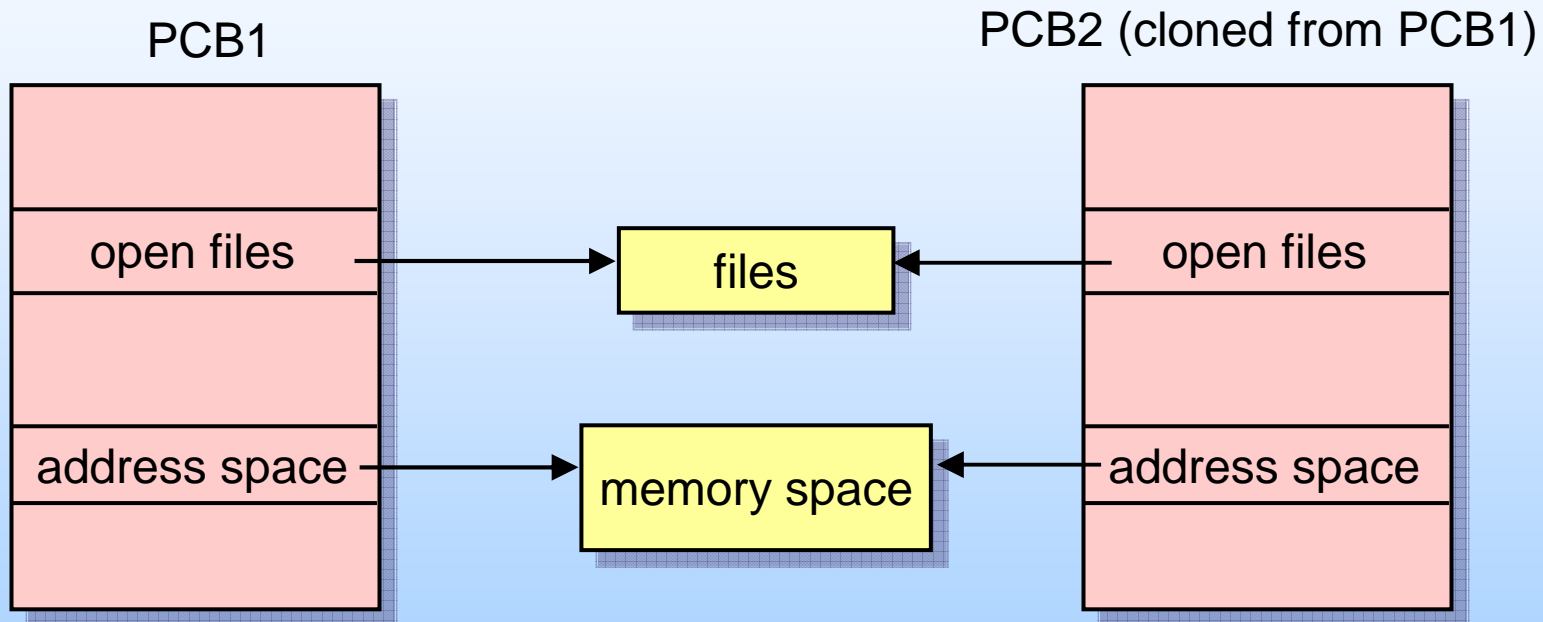
Linux Threads

- Linux refers to them as **tasks** rather than threads
- Thread creation is done through the `clone()` system call
- `clone()` allows a child task to share different things with the parent task such as:
 - the address space
 - the table of open files
 - the signal handlers
- If nothing is shared: same as `fork()`, creates essentially a process
- If everything is shared: creates a thread as we know it

Linux Threads (cont'd)

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

`clone()` is used to create kernel-level threads



To the user: PCB1 and PCB2 are
“kernel-level threads within the same process”

To the kernel: PCB1 and PCB 2 are processes (tasks)

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
 - Extending Thread class
 - Implementing the Runnable interface
- Details in textbook



End of Chapter 4



Silberschatz, Galvin and Gagne ©2005
Modified by Dimitris Margaritis, Spring 2007