CS3510

Threads

Bheemarjuna Reddy Tamma
IIT HYD

Outline

- Case for Threads
- Threads vs Processes
- Thread details
  - Pthread Library

Case for Parallelism

```
main()
  read_data();
  for(all data i ← 1 to N)
    compute(i);
    write_data(i);
  endfor
```

Blocking Write

```
main()
    read_data();
    for(all data i ← 1 to N)
        compute(i);
    CreateProcess(write_data(i));
    endfor
```

Does Writing in new process

Case for Parallelism

Consider the following code fragment:

```
for(k = 0; k < n; k++)
  a[k] = b[k] * c[k] + d[k] * e[k];
```

Instead:

```
fn(l, m) {
  for(k = l; k < m; k++)
      a[k] = b[k] * c[k] + d[k] * e[k];
}
CreateProcess(fn, 0, n/2);
CreateProcess(fn, n/2, n);
```

Case for Parallelism

Consider a Web server

create a number of processes, and for each process do

- get network message (HTTP REQ) from client
- get URL data from disk
- compose response
- send response (HTML Object)

- Server connections are fast, but client connections may not be
  - Takes server a loooong time to feed the response to client
  - While it's doing that it can't service any more requests

Parallel Programs

- To build parallel programs, such as:
  - Parallel execution on a multiprocessor
  - Web server to handle multiple simultaneous web requests
- We will need to:
  - Create several processes that can execute in parallel
  - Cause *each* to map to the *same* address space
    - because they're part of the same computation
  - Give each its starting address and initial parameters
  - The OS will then schedule these processes in parallel

Parallelism vs Concurrency

- Both are not same, but typically used interchangeably!
- Parallelism: Doing multiple tasks simultaneously
  - E.g., driving while talking!
  - E.g., Running Word App and Media player simultaneously, each one on separate Core/CPU
  - Not possible w/o multiple Cores
- Concurrency: Making progress for multiple tasks
  - Single CPU: time-sharing of CPU resource with time slices
  - Multiple CPU: same as parallelism
- So, Parallelism → Concurrency, but not vice versa.

Processes Overheads

- A full process includes numerous things:
  - an address space (defining all the code and data segments)
  - OS resources and accounting information
  - a "thread of control",
    - defines where the process is currently executing
    - That is the PC and registers
- Creating a new process is costly
  - all of the structures (e.g., page tables, address space) that must be allocated
  - Costly even after copy-on-write optimization
- Communicating between processes is costly
  - most communication goes through the OS to avoid synchronization issues with shared resources

Need "Lightweight" Processes

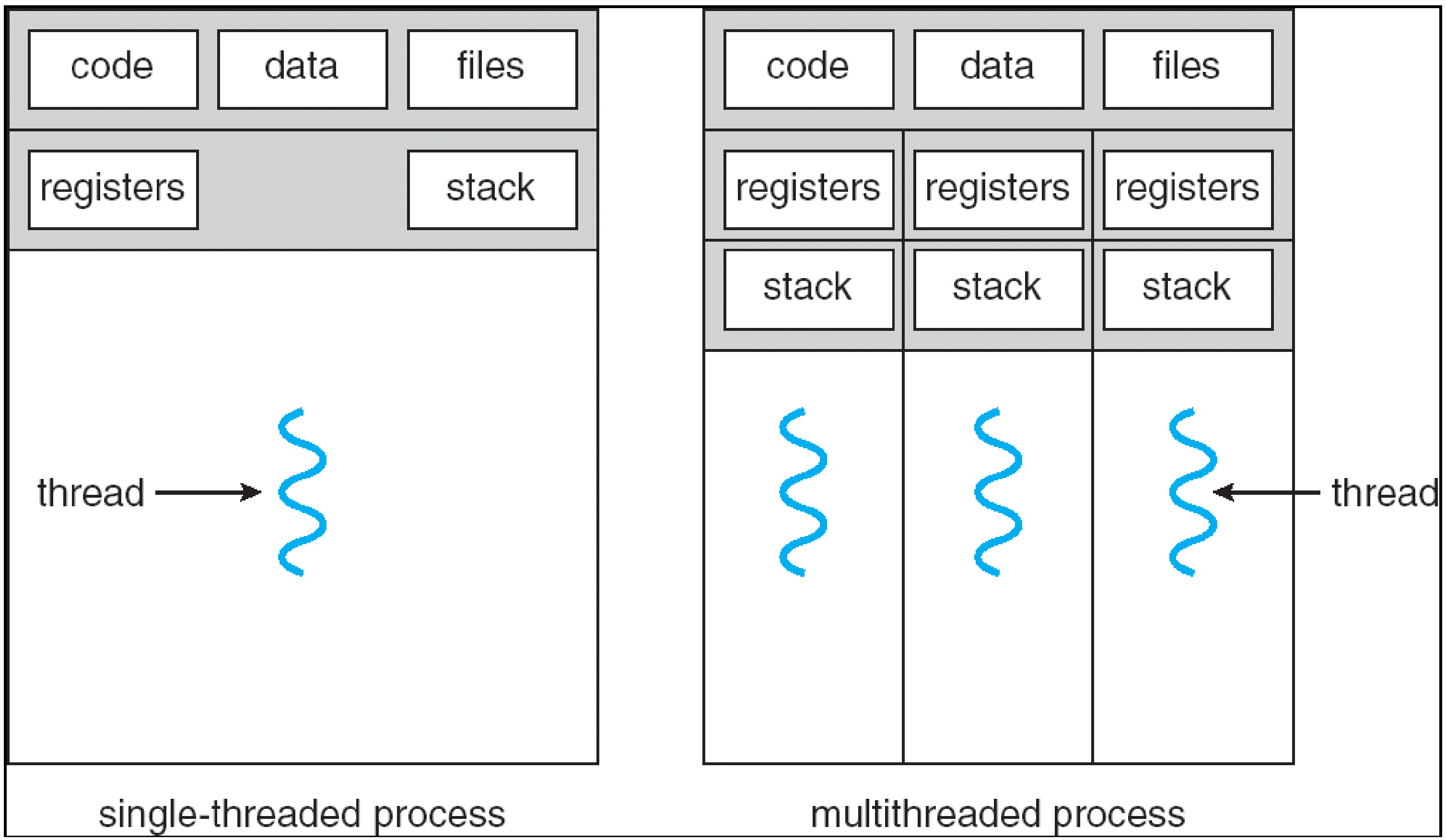
- What's similar in these processes?
  - They all share the same code, heap and data (most of the address space)
  - They all share the same privileges
  - They share almost everything in the process
- What don't they share?
  - Each has its own PC, register set, stack, and stack pointer

- Idea: why don't we separate the idea of process (address space, accounting, etc.) from that of the minimal "thread of control" (PC, SP, registers)?

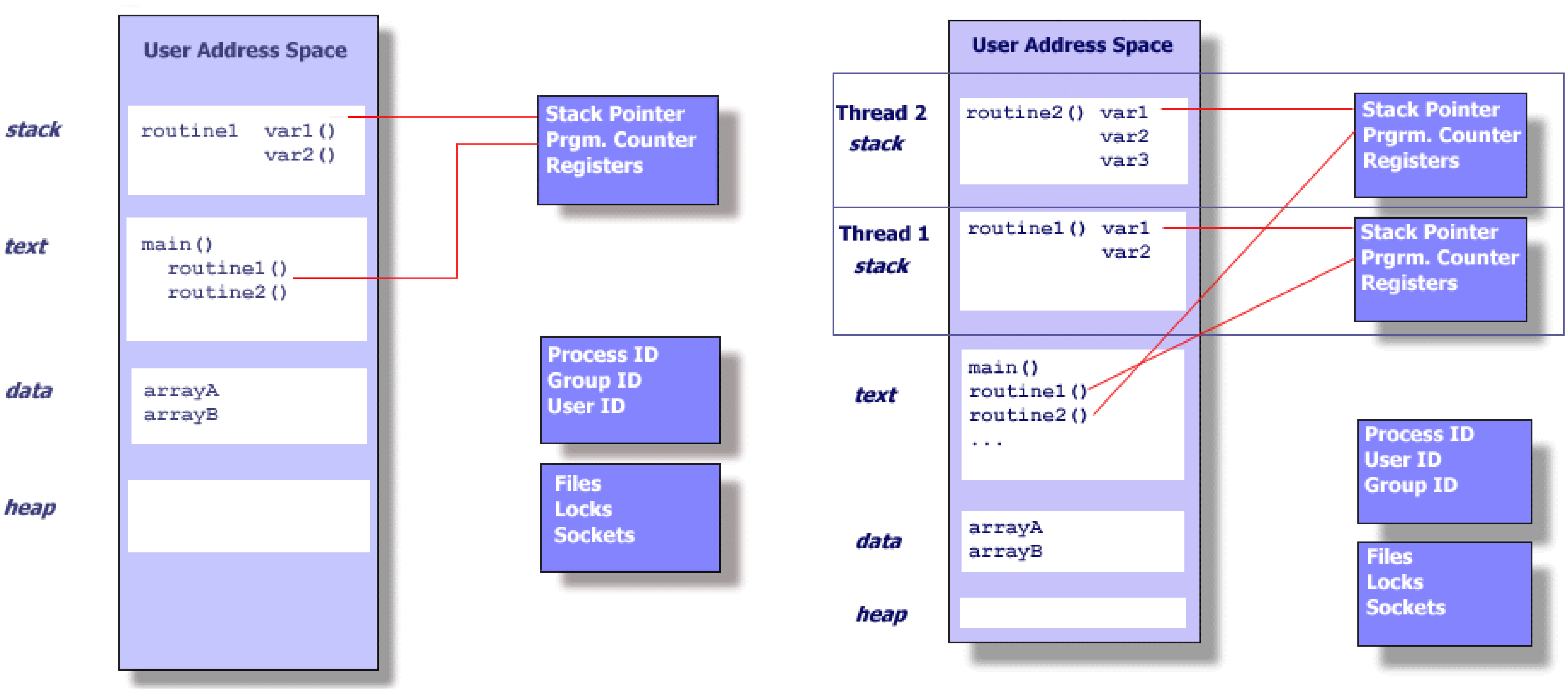
Threads and Processes

- Most operating systems therefore support two entities:
  - the <u>process</u>,
    - which defines the <u>address space</u> and general process attributes
  - the <u>thread</u>,
    - which defines a sequential execution stream within a process
- A thread is bound to a single process.
  - For each process, however, there may be many threads.
- Threads are the unit of scheduling
- Processes are *containers* in which threads execute

Multithreaded Processes



Single vs Multithreaded Processes



Source: https://computing.llnl.gov/tutorials/pthreads/

Threads vs. Processes

- *A thread has no separate code/data segment or heap*
- *A thread cannot live on its own, it must live within a process*
- *There can be more than one thread in a process, the first thread calls main & has the process's stack*
- Inexpensive creation
- Inexpensive context switching
- If a thread dies, its stack is reclaimed

- A process has code/data/heap & other segments
- There must be at least one thread in a process
- *Threads within a process share code/data/heap, share I/O, but each has its own stack & registers*
- Expensive creation
- Expensive context switching
- If a process dies, its resources are reclaimed & all threads die

Separating Threads and Processes

- Makes it easier to support multithreaded applications
  - Different from multiprocessing, multiprogramming
  - Multiprocessing ≡ Multiple CPUs, parallel job execution
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ Multiple threads per Process
- Concurrency (multithreading) is useful for:
  - improving program structure
  - handling concurrent events (e.g., web requests)
  - building parallel programs
  - Resource sharing
  - Multiprocessor utilization
- Is multithreading useful even on a single processor?

Benefits of multithreaded programs

- Responsiveness
  - Interactive apps like Web server
- Resource sharing
  - Implicit as threads share the same address space
- Economy
  - Creation of multithread process is cheaper
  - Solaris
    - Process creation is 30 times slower than that of thread creation
    - Context switch is about 5 times slower
- Scalability
  - Single-threaded process can only run on one CPU
  - Multithreading in multi-core m/cs increases parallelism

Examples of multithreaded programs

- Embedded systems
  - Elevators, Planes, Medical systems
  - Single Program, concurrent operations
- Most modern OS kernels
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - Threads for I/O devices, interrupt handling, managing amount of free memory, etc
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done

Single-Threaded Example

- Consider the following CHESS program:

```
main() {
    DisplayChessBoad();
    ComputeNextMove();
}
```

- What is the behavior here?
- Version of program with Threads:

```
main() {
    CreateThread(DisplayChessBoad());
    CreateThread(ComputeNextMove());
}
```

Single-Threaded Example

- Consider the following C program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("clist.text");
}
```

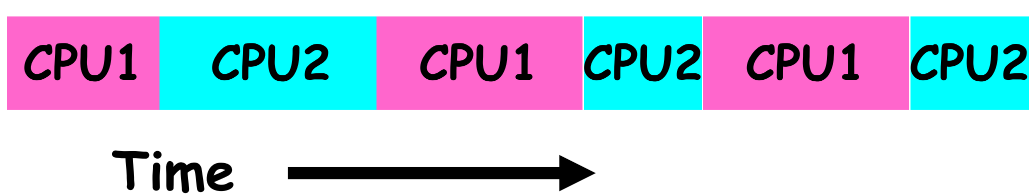
- What is the behavior here?
  - Program would never print out class list
  - Why?
    - ComputePI would never finish
  - Solution?
    - (w/o using threads?)

Use of Threads

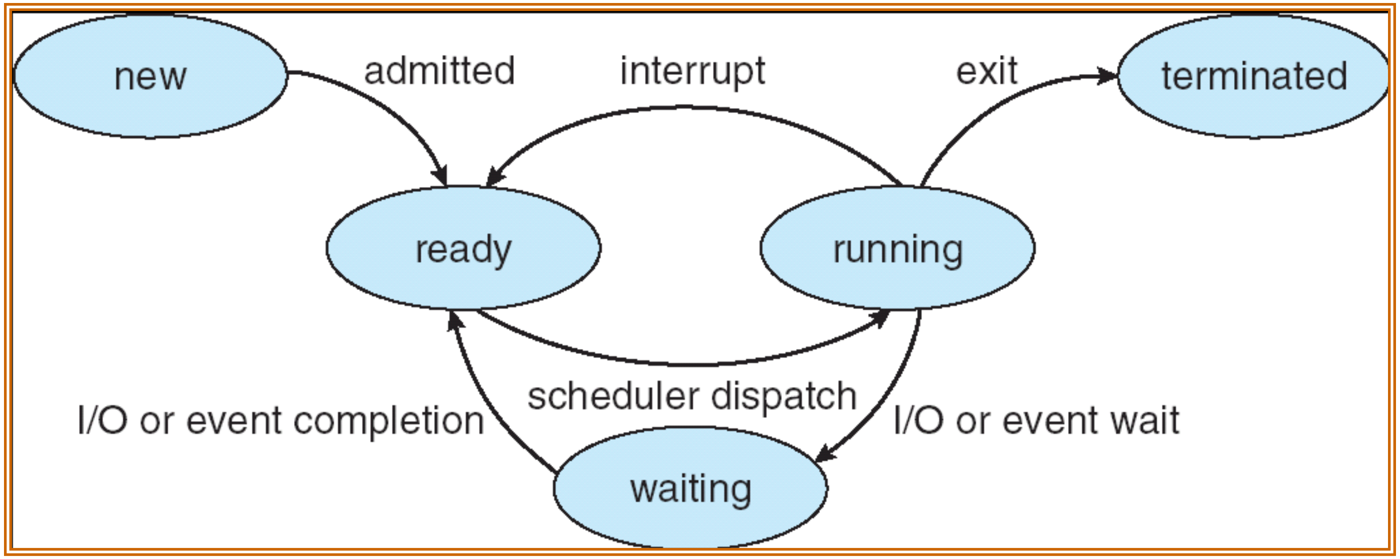
- Version of program with Threads:

```
main() {
    CreateThread(ComputePI(“pi.txt”));
    CreateThread(PrintClassList(“clist.text”));
}
```

- What does “CreateThread” do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs



Lifecycle of a Thread (or Process)



- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their state

Cooperative Threads

A *cooperative* thread runs until *it* decides to give up the CPU

```
main()
{
    tid t1 = CreateThread(fn, arg);
    …
    Yield(t1);
}
fn(int arg)
{
    …
    Yield(any);
}
```

Cooperative Threads

- Cooperative threads use non pre-emptive scheduling
- Advantages:
  - Simple
    - Scientific apps
- Disadvantages:
  - For badly written code
- Scheduler gets invoked only when Yield is called
- A thread could also yield the processor when it blocks for I/O

Non-Cooperative Threads

- No explicit control passing among threads
- Rely on a scheduler to decide which thread to run
- A thread can be pre-empted at any point
- Often called pre-emptive threads
- Most modern thread packages use this approach
  - Pthreads API
  - Win32 threads API
  - Java API

Classification of OS

# threads Per AS: / # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux Windows 10, Solaris, HP-UX, OS X

- Most operating systems have either
  - One or many address spaces
  - One or many threads per address space

Example User Thread Interface

t = thread_fork(initial context)

creates a new thread of control

thread_stop()

stops the calling thread, sometimes called thread_block

thread_start(t)

starts the named thread

thread_yield()

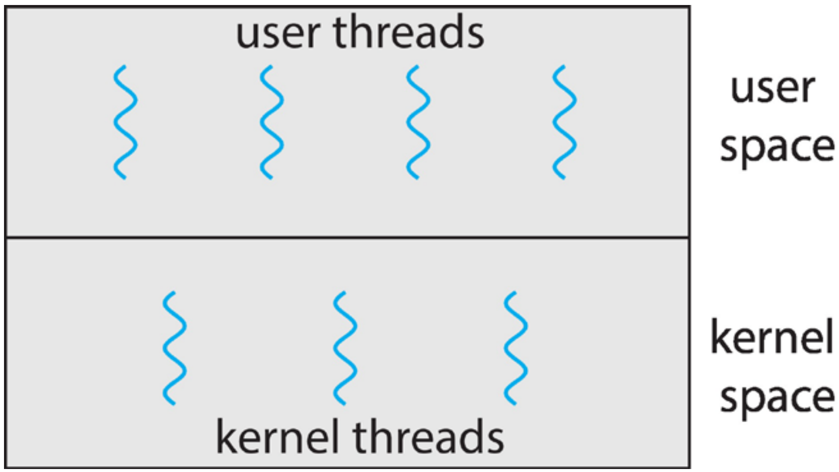
voluntarily *gives* up the processor

thread_exit()

terminates the calling thread, sometimes called thread_destroy

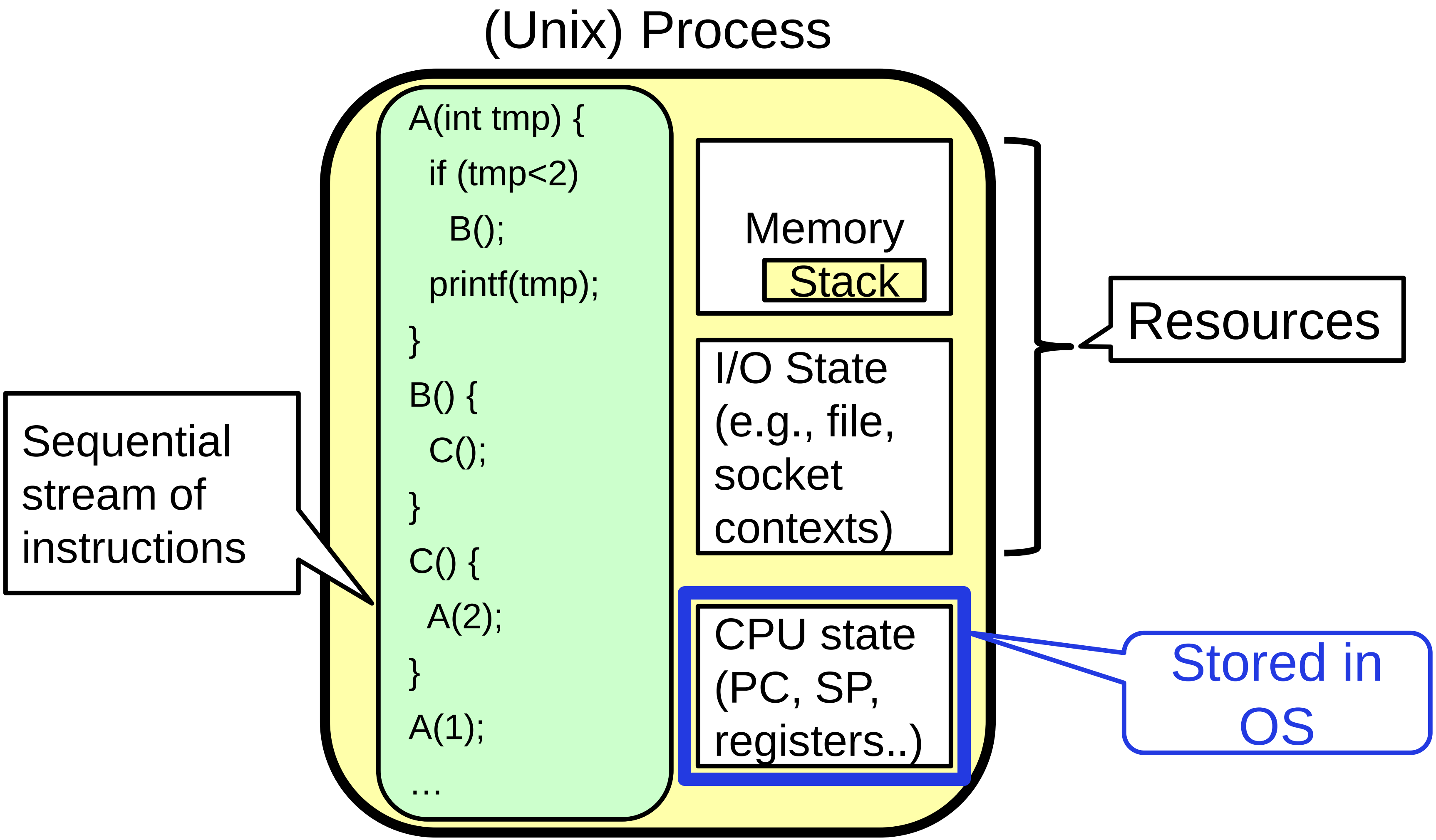
Multithreading models

- There are actually 2 level of threads:
- Kernel threads:
  - Supported and managed directly by the kernel.
  - Windows, Mac, Linux
- User threads:
  - Supported above the kernel, and without kernel knowledge by user-level threads library.
  - E.g., POSIX Pthreads API

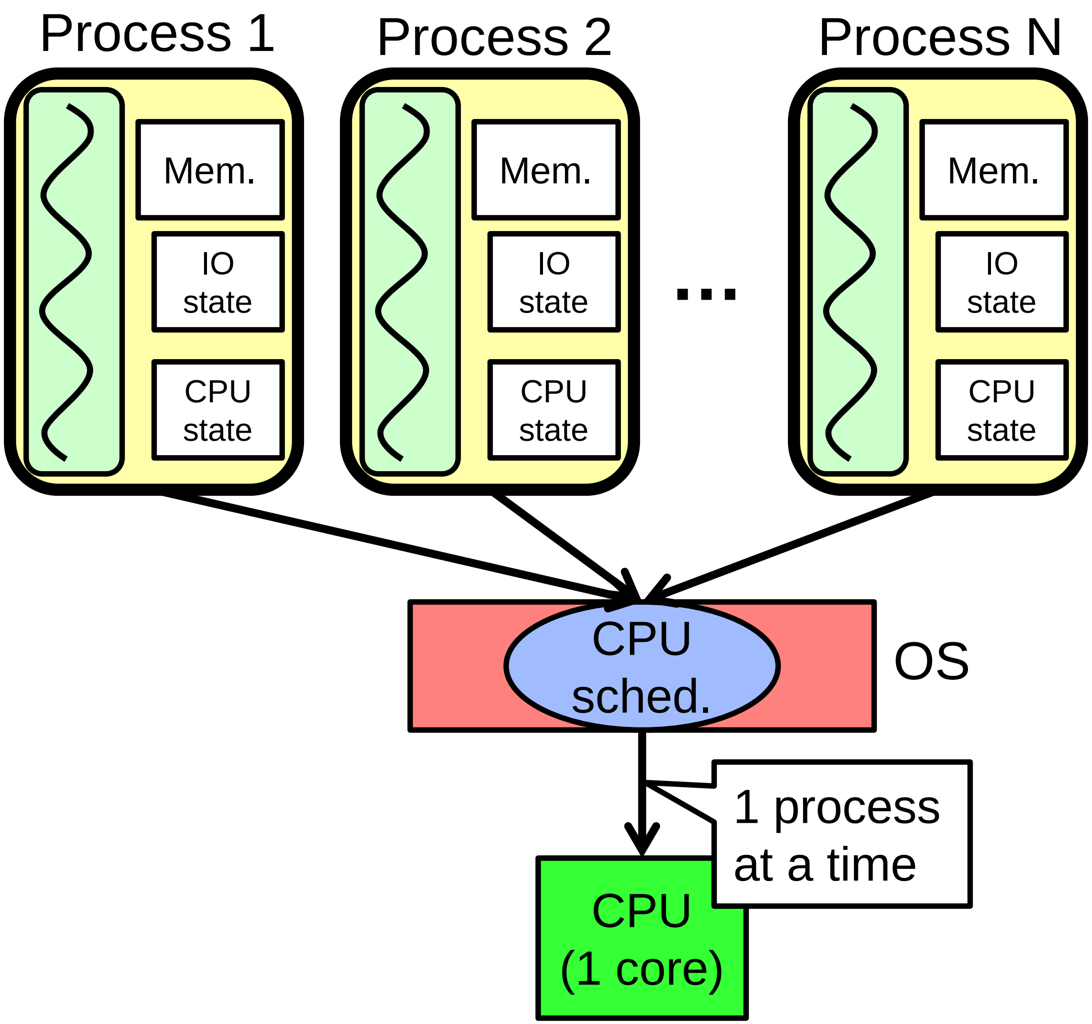


1:1 mapping b/w user and kernel threads in Windows & Linux

Putting it Together: Process

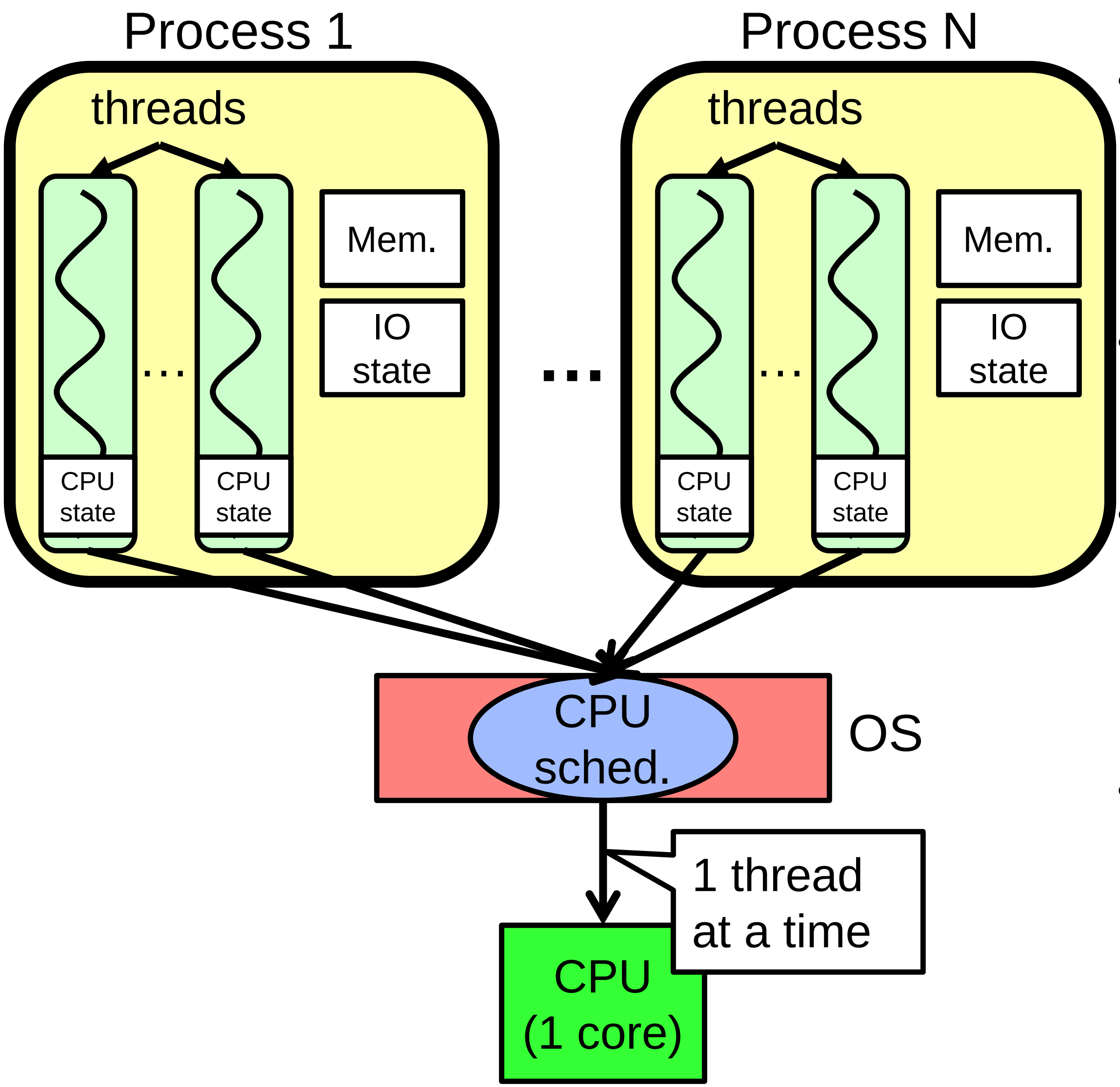


Putting it Together: Processes



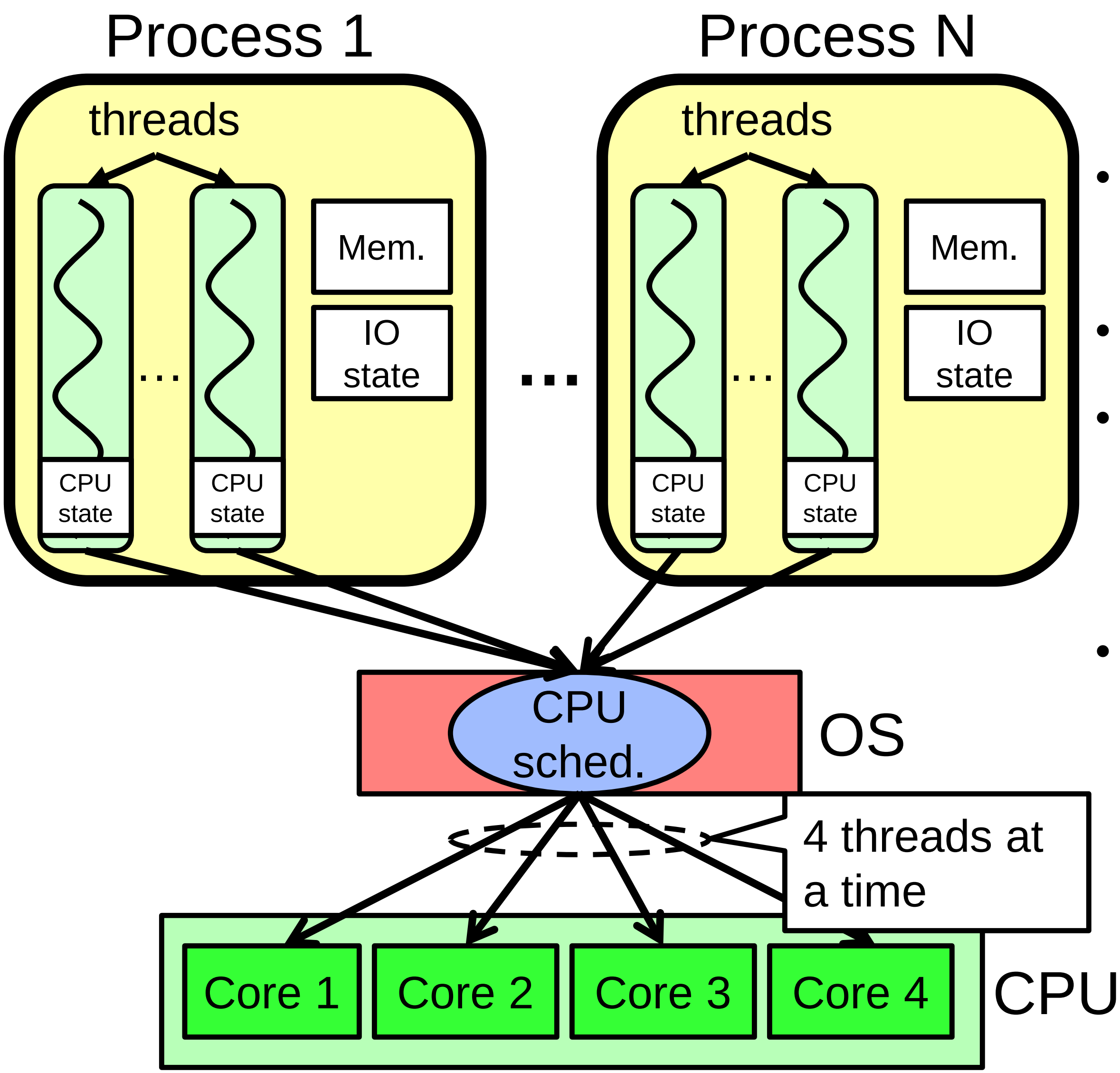
- Switch overhead: high
  - CPU state: *low*
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: *yes*
  - Memory/IO: *yes*
- Sharing overhead: high (involves at least a context switch)

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!)

Pthreads

- 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
- May be provided either as user-level or kernel-level threads
- Common in UNIX operating systems (Linux & Mac OS X)
  - Implemented by glibc

Pthreads API

- **pthread_create** (thread id, attr, start_routine, arguments_start_routine)
- **pthread_exit** (status)
- **pthread_cancel** (thread id)
- **pthread_attr_init** (attr)
- **pthread_attr_destroy** (attr)
- **pthread_join** (thread id, status)
- **pthread_detach** (thread id)
- **pthread_attr_setdetachstate** (attr, detachstate)
- **pthread_attr_getdetachstate** (attr, detachstate)
- **pthread_self** (), **pthread_equal** (TID1, TID2)

Pthreads: Creating Threads

- main() program comprises a single, default thread
  - All other threads must be explicitly created using pthread_create, anywhere in the program
- By default, thread is created with certain attributes (configurable)
  - pthread_attr_init and pthread_attr_destroy are used to initialize/destroy thread attribute object
  - Other routines are then used to query/set specific attributes in the thread attribute object
  - Detached or joinable state
  - Scheduling policy, Scheduling parameters
  - Stack address, Stack size, etc

Pthread API: Creation and Termination

```
#include <pthread.h> //Implemented by glibc
#include <stdio.h>
#define NUM_THREADS     5
void *PrintHello(void *threadNo)
{
    long tid;
    tid = (long*)threadNo;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(void *threadNo); //can pass status to other threads
}
```

→ Use gcc & g++ with -pthread flag

→ getrlimit: get user limits of system resources like memory, no of processes, timeslice, etc

→ $ulimit: get and set user limits of system resources

→ $cat /proc/[PID]/limits

→ $sudo cat /proc/[PID]/sched

Pthread API: Creation and Termination

```
int main ( )
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t; void * status;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n",
rc);
         exit(-1);
      }
   }
   /* main() should wait for thread(s) to finish */
   for(t=0; t<NUM_THREADS; t++)
   pthread_join(threads[t],&status); //collect status
pthread_exit(NULL); //makes
}
```

Pthread API: Thread Argument Passing (safeway)

```
long *taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
   taskids[t] = (long *) malloc(sizeof(long));
   *taskids[t] = t;
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *)
taskids[t]);
   ...
}
```

Full Source Code:

https://computing.llnl.gov/tutorials/pthreads/samples/hello_arg1.c

Pthread API: Thread Argument Passing (Unsafe)

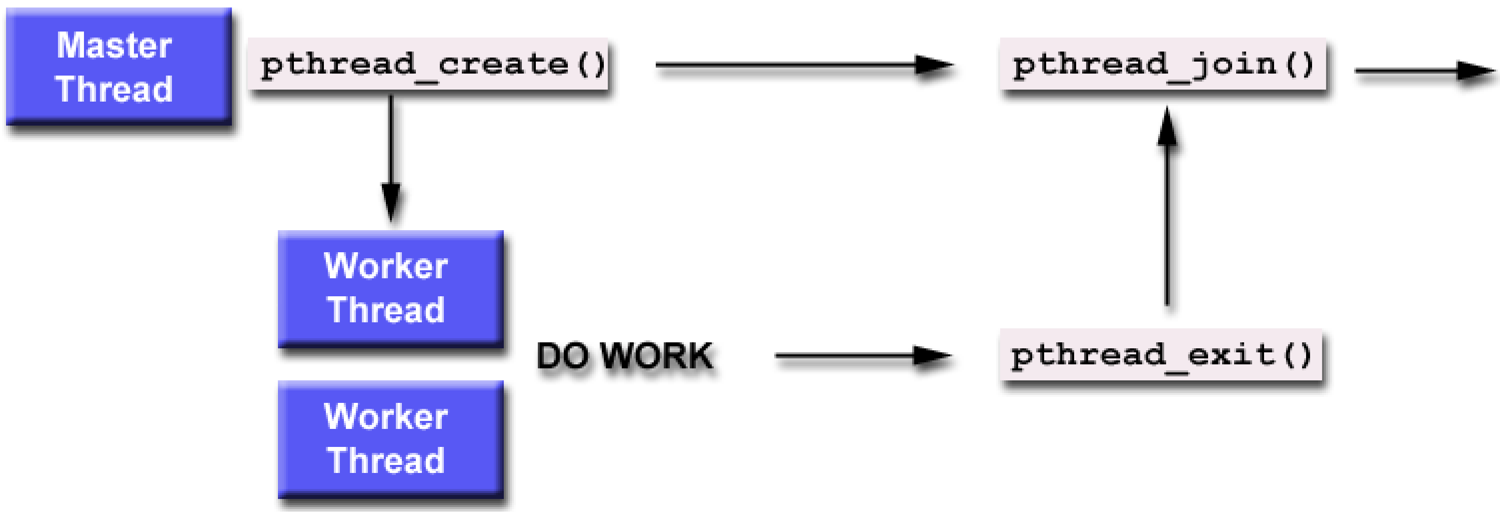
```
int rc;
long t;

for(t=0; t<NUM_THREADS; t++)
{
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
   ...
}
```

- Param **t** is changed by the main thread as it creates new threads

Full Src Code:
https://computing.llnl.gov/tutorials/pthreads/samples/hello_arg3.c

Pthread API: Thread Join/Detach



- “joining" is one of the ways to accomplish synchronization between threads
- Calling pthread_exit() at last in main( ) blocks the process till all its threads are done!
- Example:

https://computing.llnl.gov/tutorials/pthreads/samples/join.c

Pthread: Issues

After a thread has been created, how do you know

a) when it will be scheduled to run by the OS

b) which processor/core it will run on?

Ans:

1. Depends on underlying thread scheduling algo (FIFO/RO/etc for pthreds) or
2. Implementation specific

Robust programs should not depend on threads running order or core on which a thread runs on

Linux Threads

- Linux does not distinguish between processes and threads
  - Uses term task (struct task_struct)
  - clone ( ) for creating threads
    - Flags passed as args determine level of sharing b/w parent and child tasks
    - CLONE_FS, CLONE_VM, CLONDE_FILES
    - Sharing → threads
    - No sharing → processes
  - fork( ) for creating duplicate tasks (processes)

Multithreading Issues

- Semantics of fork() and exec() system calls
  - Child process duplicates all threads of parent?
  - Two versions of fork()!!
  - exec() inside a thread will replace the entire process (inc all threads) with prg specified as arg for exec()
- Thread cancellation
  - Asynchronous vs. Deferred Cancellation
  - pthread_cancel (thread id) supports both, but deferred is recommended as it's safe
- Signal handling
  - Which thread to deliver it to?
  - kill(pid,signal)
  - pthread_kill(tid,signal)

Thread Hazards

```
int a = 1, b = 2, w = 2;

main( ) {
   CreateThread(fn, 4);
   CreateThread(fn, 4);
   while(w)   ;
}
fn( ) {
    int v = a + b;
    w--;
}
```

Concurrency Problems

A statement like w-- in C (or C++) is implemented by several machine instructions:

```
ld      r4, #w
add     r4, r4, -1
st      r4, #w
```

Now, imagine the following sequence, what is the value of w?

ld r4, #w	
______________	______________
______________	ld r4, #w
______________	add r4, r4, -1
add r4, r4, -1	st r4, #w
st r4, #w	

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Implicit Threading: Creation and management of threads done by compilers and run-time libraries rather than programmers!
- Three methods
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), `java.util.concurrent` package

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 run periodically

- Windows API supports thread pools:

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

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies parallel regions – blocks of code that can run in parallel

```
#pragma omp parallel
```

Create as many threads as there are cores!

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

Run for loop in parallel

```
#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;
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in "^{ }" - `^{ printf("I am a block"); }`
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue
- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process, called main queue
  - concurrent – removed in FIFO order but several may be removed at a time
    - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
   (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

Summary

- Threads increase concurrency/parallelism
- Threads may cause synchronization issues
  - Need to employ synchronization primitives to avoid thread hazards

Reading Assignment

- Chapter 4 from OSC by Galvin et al
- Chapter 2 from MOS by Tanenbaum et al
- http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- https://computing.llnl.gov/tutorials/pthreads/