What we've learned from "HW: adding our system calls to kernel"

- System call definitions are part of the kernel!
  - they are executed in kernel mode
  - they have access to kernel-space
  - It can taint the security/protection mechanism of the kernel

- Parameters(data) are passed from user-space to kernel-space or vice versa

- Pointer parameters (char *buf)
  - Checking issues related to **pointers** are important
  - Never allow pointers to kernel-space
  - Check for invalid pointers
  - use copy_to_user and copy_from_user for data transfers

HW: Adding a Simple Module to the Kernel

System calls are easily to implement but difficult to install, test, and debug

- You compiled kernel from scratch.
-

Kernel modules

- can be installed on a running kernel
- they can be stopped/restarted/reinstalled on running kernel.

They can also taint the security/protection of the kernel

The code of kernel modules is also executed in kernel-space in the unrestricted mode.

https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.htm

Chapter 2. Managing kernel modules | Red Hat Product Documentation l

HW2: Adding a Module to a Linux Kernel

listing installed kernel modules

$ grubby --info=ALL | grep title

listing loaded kernel modules

$ lsmod

```
$ lsmod
```

```
Module                  Size  Used
by
fuse                  126976  3
uinput                 20480  1
xt_CHECKSUM            16384  1
ipt_MASQUERADE         16384  1
xt_conntrack           16384  1
ipt_REJECT             16384  1
nft_counter            16384  16
nf_nat_tftp            16384  0
nf_conntrack_tftp      16384  1
nf_nat_tftp
tun                    49152  1
bridge                192512  0
stp                    16384  1
bridge
llc                    16384  2
bridge,stp
nf_tables_set          32768  5
nft_fib_inet           16384  1
...
```

https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.htm
Chapter 2. Managing kernel modules | Red Hat Product Documentation |

HW2: Adding a Module to a Linux Kernel

kernel module info

`modprobe <MODULE_NAME>`

`$ lsmod | grep <MODULE_NAME>`

loading a kernel module

- select a directory
  - The modules are located in the `/lib/modules/$(uname -r)/kernel/<SUBSYSTEM>/` directory.

```
$ modprobe <MODULE_NAME>
```

- or

```
$ insmod <module_name>
```

Unloading a kernel module

```
$ modprobe -r <MODULE_NAME>
```

- or

```
$ rmmod <module_name>
```

https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.htm
Chapter 2. Managing kernel modules | Red Hat Product Documentation |

HW2: Adding a Module to a Linux Kernel

```c
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>

MODULE_DESCRIPTION("My kernel module");
MODULE_AUTHOR("Me");
MODULE_LICENSE("GPL");

static int dummy_init(void)
{
        pr_debug("Hi\n");
        return 0;
}

static void dummy_exit(void)
{
        pr_debug("Bye\n");
}

module_init(dummy_init);
module_exit(dummy_exit);
```

- **dummy**
  - name of the module
- module_init(...)
  - linux/module.h de tanimli
  - init_module()
  - executed when we install the module

    ```
    $ insmod dummy
    ```

- module_exit(...)
  - linux/module.
  - executed when we remove module

    ```
    $ rmmod dummy
    ```

https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.htm

Chapter 4: Threads & Concurrency

these slides are edited and some of the contents are taken from https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf and http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Main two point:

1) threads vs. process

2) kernel-level vs. user-level threads

Review of process

Concurrency and threads

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

Review: Processes

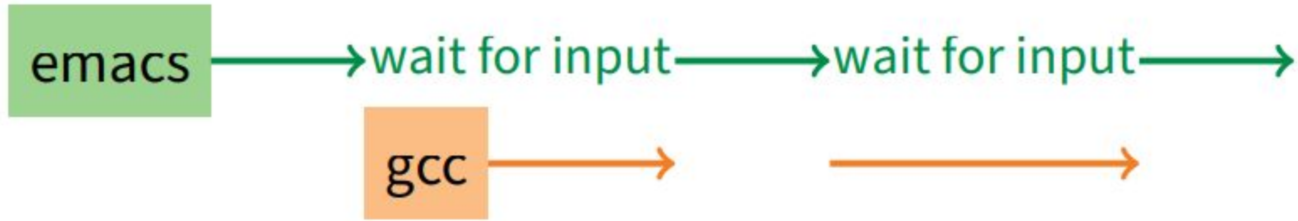
- A process is an instance of a program running

- Why processes?
  - Simplicity of programming
  - Speed: Higher throughput, lower latency

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Speed

Multiple processes can increase **CPU utilization**



Multiple processes can reduce **latency**

Running A then B requires 100 sec for B to complete

A is slower than if it had whole machine to itself,
but still <= 100 sec unless both A and B completely CPU-bound

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Multitasking in real world

1 worker 10 months to make 1 widget

hire 100 workers to make 100 widgets

- Latency for first widget >> 1/10 month
- Throughput may be < 10 widgets per month
    - if can't perfectly parallelize task
- Or 100 workers making 10,000 widgets may achieve > 10 widgets/month
        - if workers never idly wait

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

A process's view of the world

Each process has own view of machine

- Its own address space – *(char *)0xc000 different in P1 & P2
- Its own open files
- Its own virtual CPU (through preemptive
- multitasking)

Simplifies programming model

- gcc does not care that firefox is running

Sometimes want interaction between processes

- Simplest is through files: emacs edits file, gcc compiles it
- More complicated: Shell/command, Window manager/app.

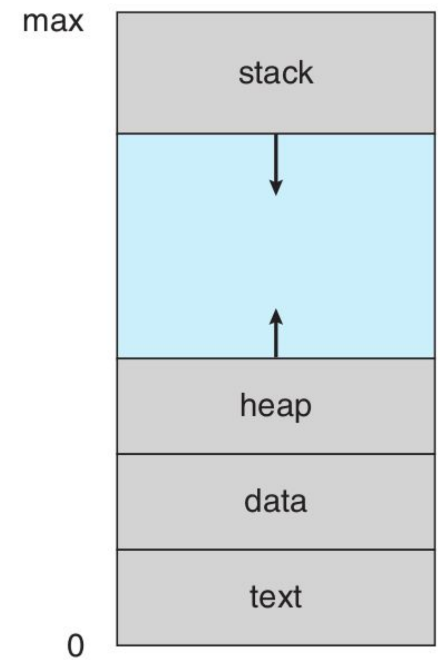
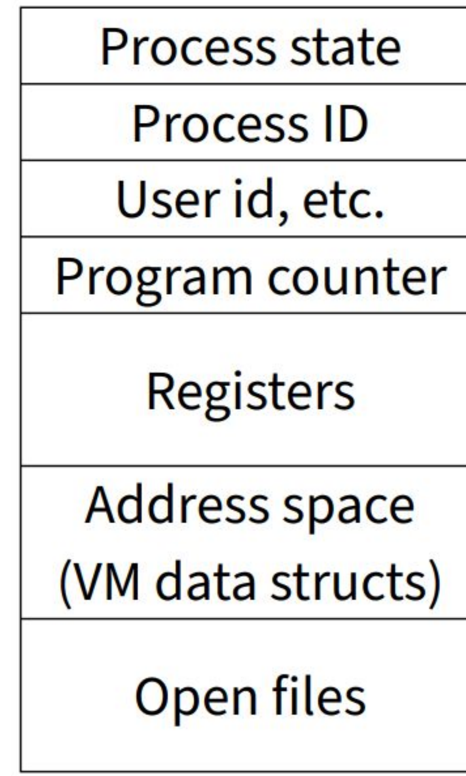


Figure 3.1 Layout of a process in memory.

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Kernel’s view of processes: implementing a process

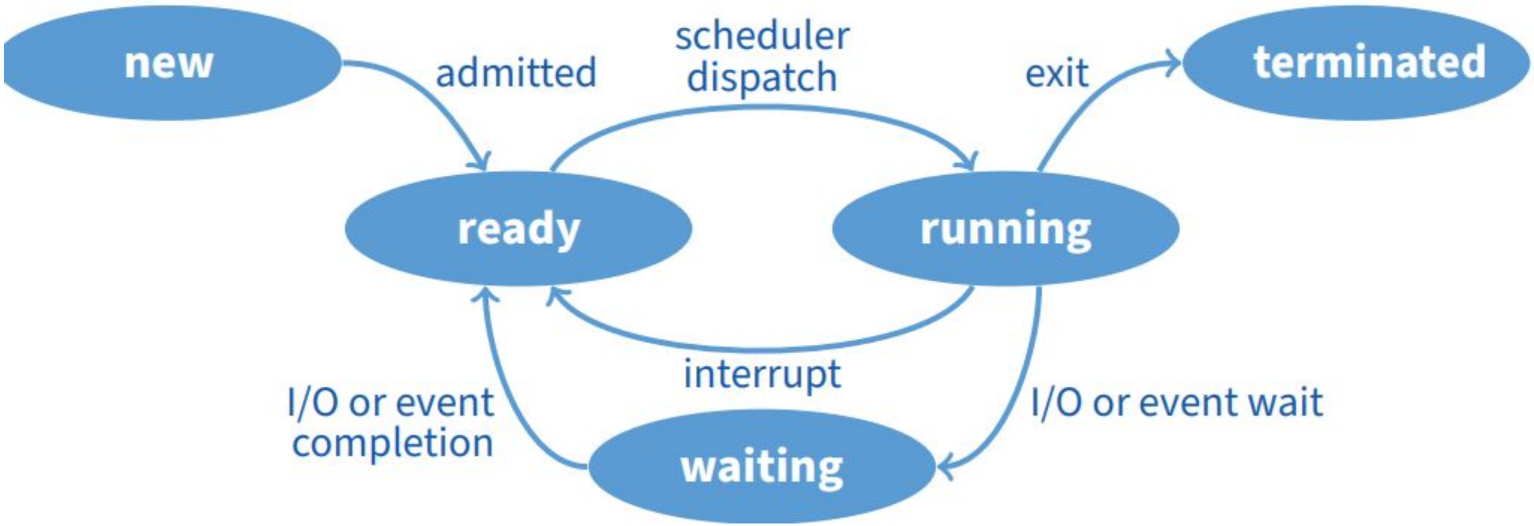
- Keep a data structure for each process
  - Process Control Block (PCB)
  - Called proc in Unix, task_struct in Linux,
- Tracks state of the process
  - Running, ready (runnable), waiting, etc.
- Includes information necessary to run
  - Registers, virtual memory mappings, etc.
  - Open files (including memory mapped files)
- Various other data about the process
  - Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether being debugged, which system call binary emulation in use, . . .



PCB

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Process states



Process can be in one of several states

- new & terminated at beginning & end of life
- running – currently executing (or will execute on kernel return)
- ready – can run, but kernel has chosen different process to run
- waiting – needs async event (e.g., disk operation) to proceed

Which process should kernel run?

- if 0 runnable, run idle loop (or halt CPU), if 1 runnable, run it
- if >1 runnable, must make scheduling decision

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Preemption

Can preempt a process when kernel gets control

- Running process can vector control to kernel
  - System call, page fault, illegal instruction, etc.
  - May put current process to sleep
    - e.g., read from disk
  - May make other process runnable
    - e.g., fork, write to pipe
- Periodic timer interrupt
  - If running process used up quantum, schedule another
- Device interrupt
  - Disk request completed, or packet arrived on network
  - Previously waiting process becomes runnable
  - Schedule if higher priority than current running proc.

Changing running process is called **a context switch**

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

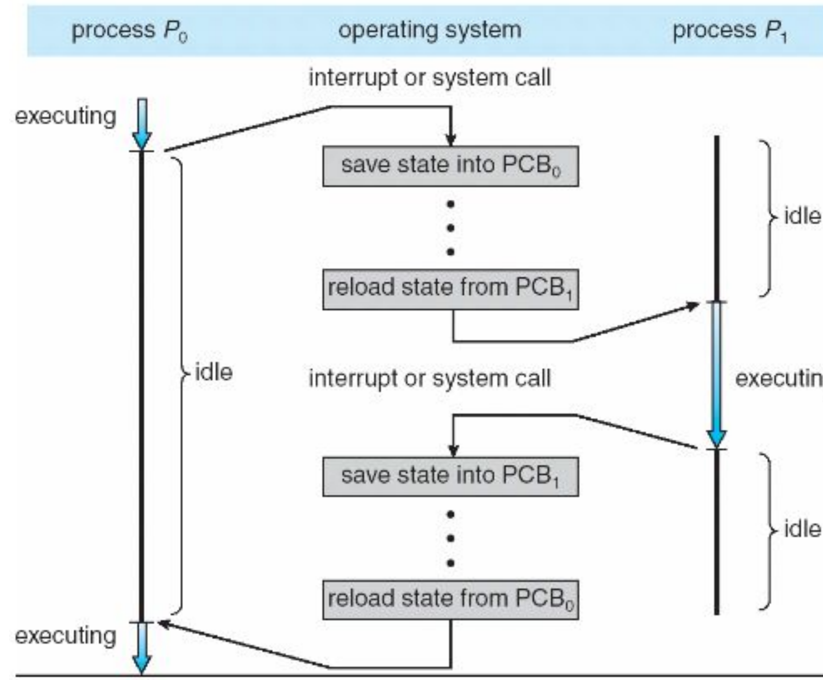
Context switch

Very machine dependent. Typical things include:

- Save program counter and integer registers (always)
- Save floating point or other special registers
- Save condition codes
- Change virtual address translations

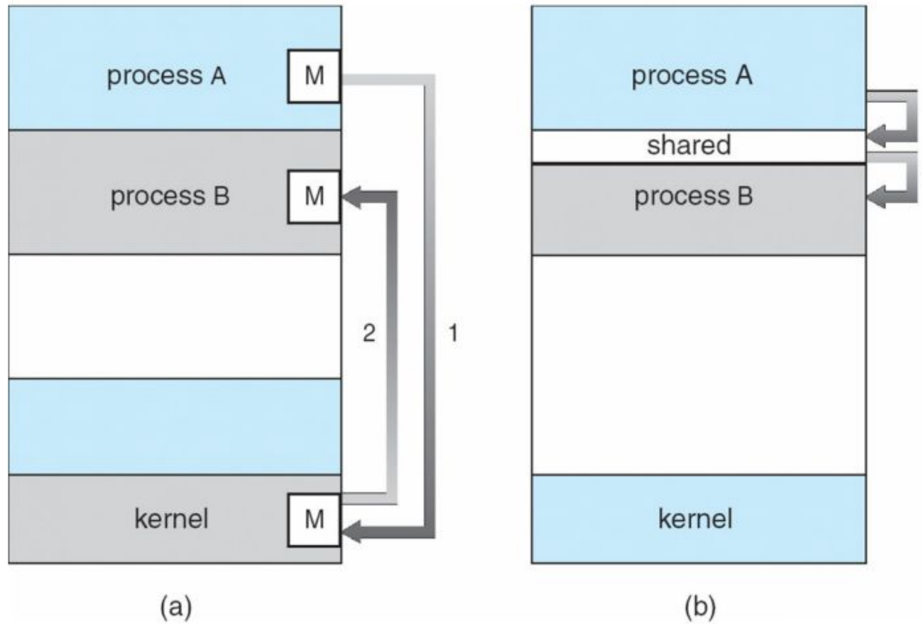
Non-negligible cost

- Save/restore floating point registers expensive
    - Optimization: only save if process used floating point
- May require flushing TLB (memory translation hardware)
- Usually causes more cache misses (switch working sets)



https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Inter-process communication



- How can processes interact in real time?
  - (a) By passing messages through the kernel
  - (b) By sharing a region of physical memory
  - (c) Through asynchronous signals or alerts

Creating/deleting processes in Unix

- `int fork (void);`
  - Create new process that is exact copy of current one
  - Returns *process ID* of new process in “parent”
  - Returns 0 in “child”
- `int waitpid (int pid, int *stat, int opt);`
  - `pid` – process to wait for, or -1 for any
  - `stat` – will contain exit value, or signal
  - `opt` – usually 0 or `WNOHANG`
  - Returns process ID or -1 on error
- `void exit (int status);`
  - Current process ceases to exist
  - `status` shows up in `waitpid` (shifted)
  - By convention, `status` of 0 is success, non-zero error
- `int kill (int pid, int sig);`
  - Sends signal `sig` to process `pid`
  - `SIGTERM` most common value, kills process by default (but application can catch it for “cleanup”)
  - `SIGKILL` stronger, kills process always

Other examples

E.g. windows system call CreateProcessA function (processthreadsapi.h) - Win32 apps | Microsoft Learn

CreateProcessAsUserA function (processthreadsapi.h) - Win32 apps | Microsoft Learn

```
BOOL CreateProcessAsUserA(
  [in, optional]      HANDLE                hToken,
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

Running programs

- `int execve (char *prog, char **argv, char **envp);`
  - `prog` – full pathname of program to run
  - `argv` – argument vector that gets passed to `main`
  - `envp` – environment variables, e.g., `PATH`, `HOME`
- **Generally called through a wrapper functions**
  - `int execvp (char *prog, char **argv);`
    Search `PATH` for prog, use current environment
  - `int execlp (char *prog, char *arg, ...);`
    List arguments one at a time, finish with `NULL`

```
pid_t pid; char **av;
void doexec () {
  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}

  /* ... main loop: */
  for (;;) {
    parse_next_line_of_input (&av, stdin);
    switch (pid = fork ()) {
    case -1:
      perror ("fork"); break;
    case 0:
      doexec ();
    default:
      waitpid (pid, NULL, 0); break;
    }
  }
```

- `int dup2 (int oldfd, int newfd);`
  - Closes `newfd`, if it was a valid descriptor
  - Makes `newfd` an exact copy of `oldfd`
  - Two file descriptors will share same offset (`lseek` on one will affect both)
- `int fcntl (int fd, int cmd, ...)` – **misc fd configuration**
  - `fcntl (fd, F_SETFD, val)` – sets close-on-exec flag
    When `val == 0`, `fd` not inherited by spawned programs
  - `fcntl (fd, F_GETFL)` – get misc fd flags
  - `fcntl (fd, F_SETFL, val)` – set misc fd flags

Loop that reads a command and executes it

Recognizes

`command < input > output 2> errlog`

```
void doexec (void) {
  int fd;
  if (infile) {      /* non-NULL for "command < infile" */
    if ((fd = open (infile, O_RDONLY)) < 0) {
      perror (infile);
      exit (1);
    }
    if (fd != 0) {
      dup2 (fd, 0);
      close (fd);
    }
  }

  /* ... do same for outfile→fd 1, errfile→fd 2 ... */

  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}
```

Example IPC: Pipes-message passing through kernel

- `int pipe (int fds[2]);`
  - Returns two file descriptors in `fds[0]` and `fds[1]`
  - Data written to `fds[1]` will be returned by `read` on `fds[0]`
  - When last copy of `fds[1]` closed, `fds[0]` will return EOF
  - Returns 0 on success, -1 on error
- **Operations on pipes**
  - `read`/`write`/`close` – as with files
  - When `fds[1]` closed, `read(fds[0])` returns 0 bytes
  - When `fds[0]` closed, `write(fds[1])`:
    - Kills process with `SIGPIPE`
    - Or if signal ignored, fails with EPIPE

```
void doexec (void) {
  while (outcmd) {
    int pipefds[2]; pipe (pipefds);
    switch (fork ()) {
    case -1:
      perror ("fork"); exit (1);
    case 0:
      dup2 (pipefds[1], 1);
      close (pipefds[0]); close (pipefds[1]);
      outcmd = NULL;
      break;
    default:
      dup2 (pipefds[0], 0);
      close (pipefds[0]); close (pipefds[1]);
      parse_command_line (&av, &outcmd, outcmd);
      break;
    }
  }
```

Overview of multicore programming

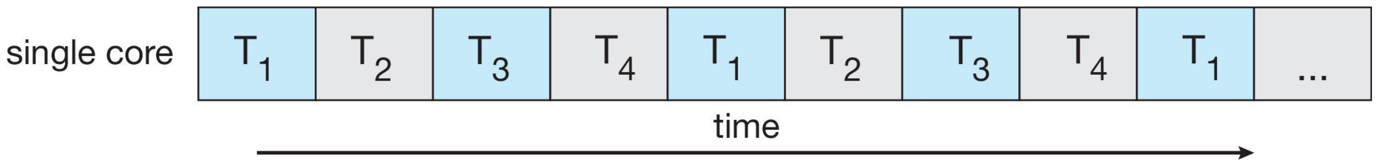
- Overview
- Multicore Programming
- Multithreading Models

Multicore Programming

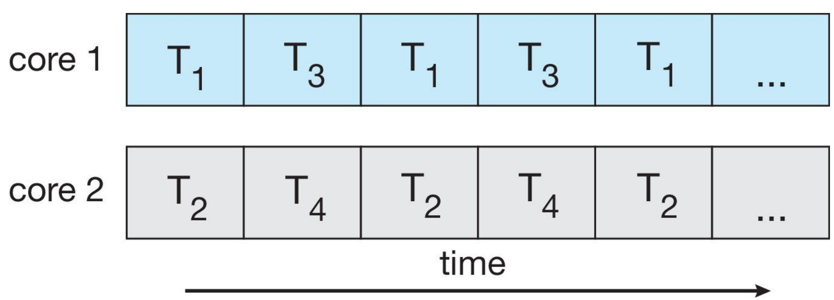
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously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

Concurrency (think as logical) vs. Parallelism (actual)

- **Concurrent execution on single-core system:**



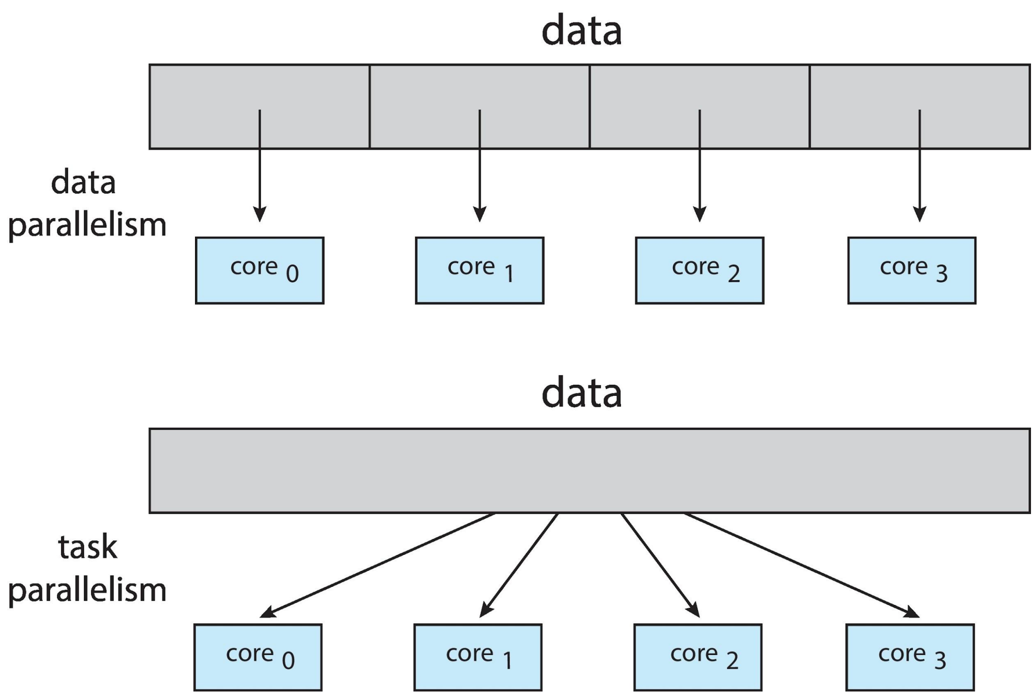
- **Parallelism on a multi-core system:**



Multicore Programming

- Types of parallelism
  - **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

Data and Task Parallelism



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components

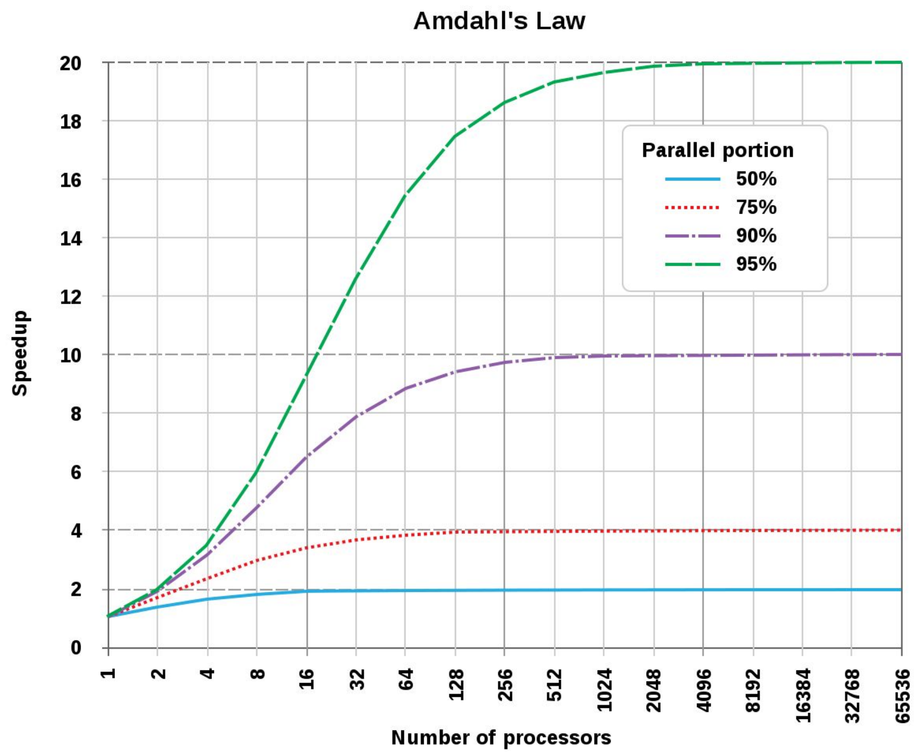
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

  - *S* is fraction of task that is necessarily serial (the rest is parallel)
  - *N* processing cores

- What is the speedup, if application is 75% parallel and 25% serial, moving from 1 to 2 cores?

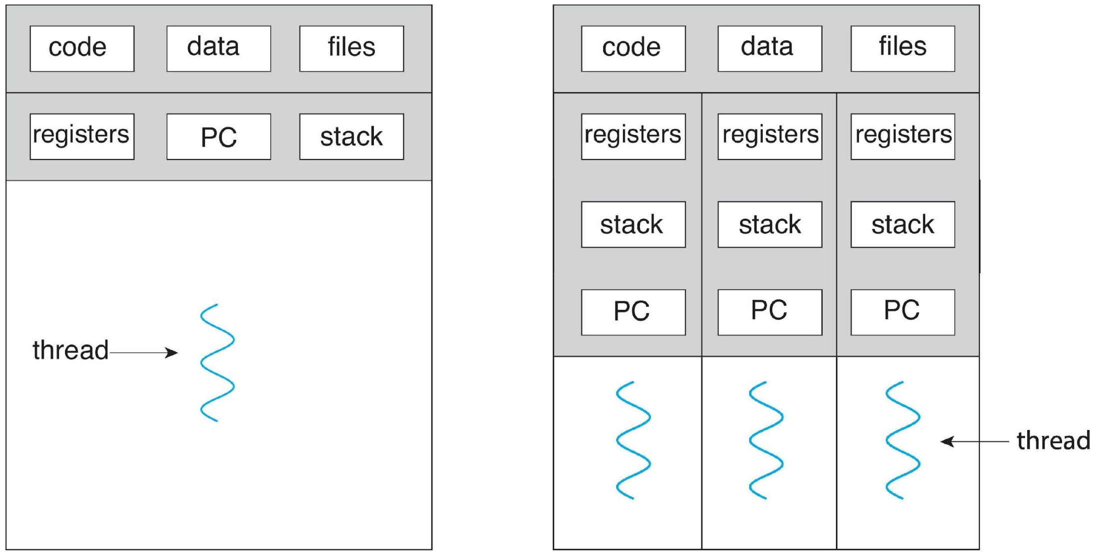
- What happens
  - as S approaches 0?
  - as S approaches 1?
  - as N approaches infinity?

Amdahl’s Law



https://en.wikipedia.org/wiki/Amdahl%27s_law#/media/File:AmdahlsLaw.svg

Single and Multithreaded Processes



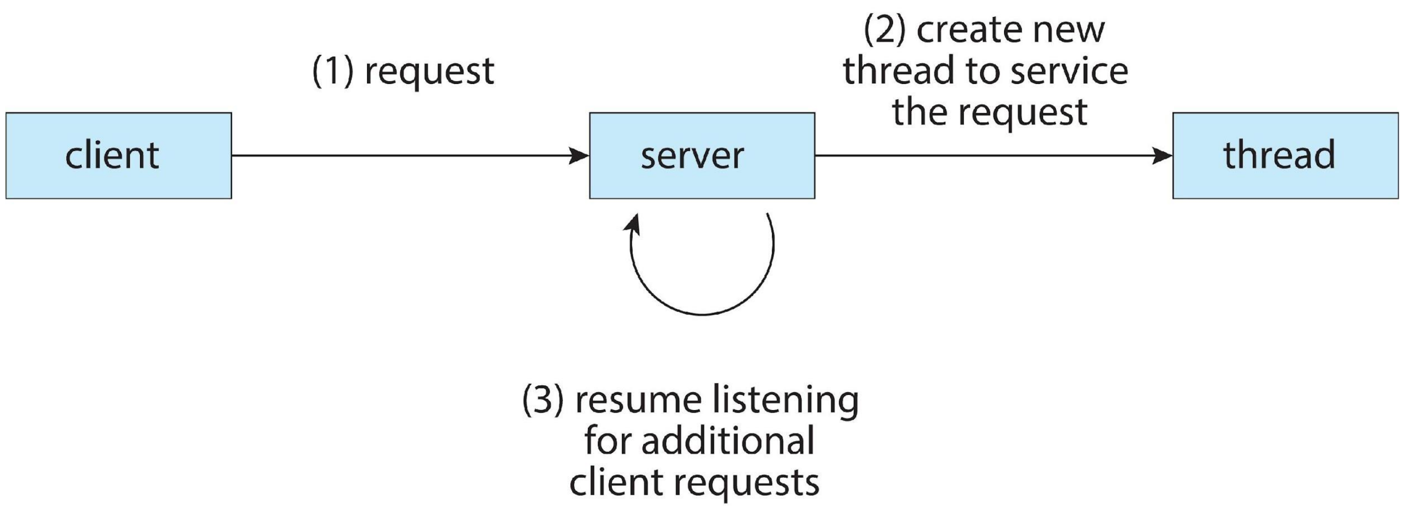
single-threaded process multithreaded process

- A thread is a schedulable execution context
  - Program counter, stack, registers, . . .
- Simple programs use one thread per process
- But can also have multi-threaded programs
  - Multiple threads running in same process's address space

Motivation for threads

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
    - Allows one process to use multiple CPUs or cores
    - Allows program to overlap I/O and computation
- Process creation is heavy-weight while thread creation is light-weight
    - All threads in one process share memory, file descriptors, etc.
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Multithreaded Server Architecture




```
for (;;) {
        fd = accept_client ();
        thread_create (service_client, &fd);
}
```

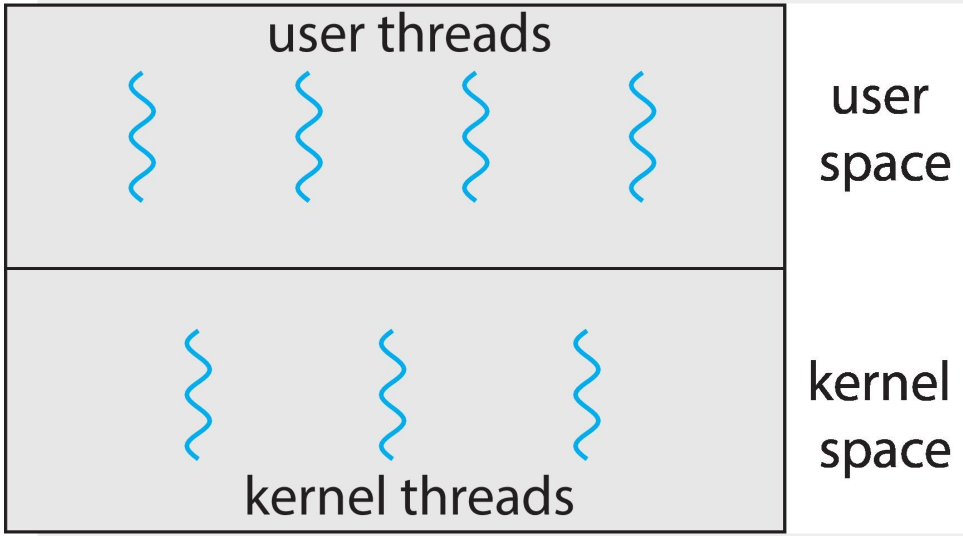
Benefits

- **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 multicore architectures

User and Kernel Threads

CPU-Scheduler?

- User level
- Kernel level



Thread reminders

The execution of multiple threads is **interleaved!**

- **there may be race condition**
- **may need synchronization**

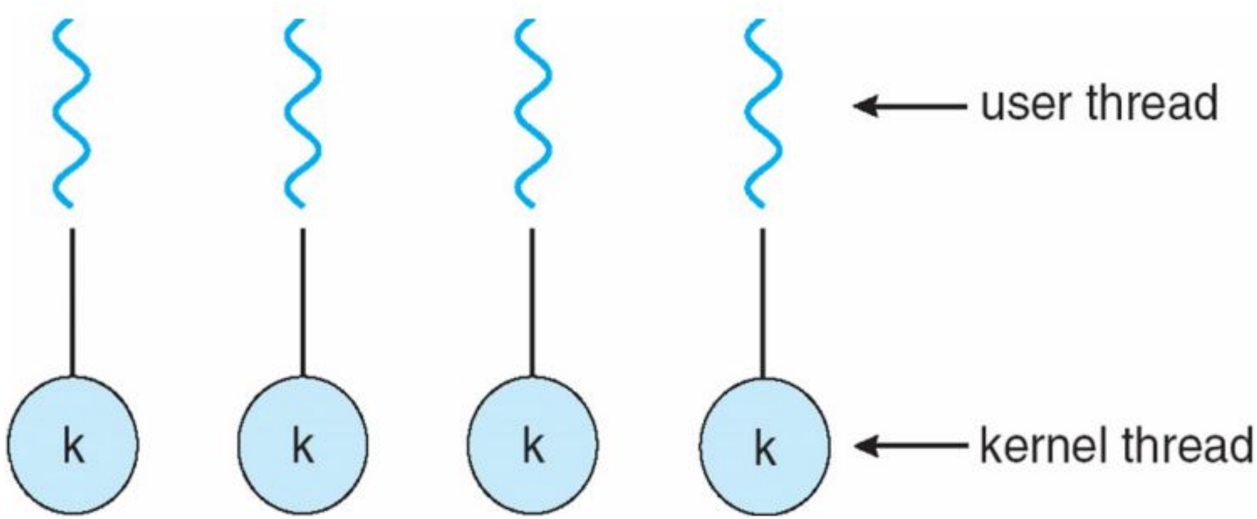
Can have **non-preemptive threads**

- One thread executes exclusively until it makes a blocking call

Or **preemptive threads** (what we usually mean in this class)

- May switch to another thread between any two instructions.

How to implement Kernel Threads?



Can implement **thread_create** as **a system call**

- Start with process abstraction in kernel
- **thread_create** like process creation with features stripped out
  - Keep same address space, file table, etc., in new process
  - rfork/clone syscalls actually allow individual control

Faster than a process, but still very heavy weight

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Limitations of kernel-level threads

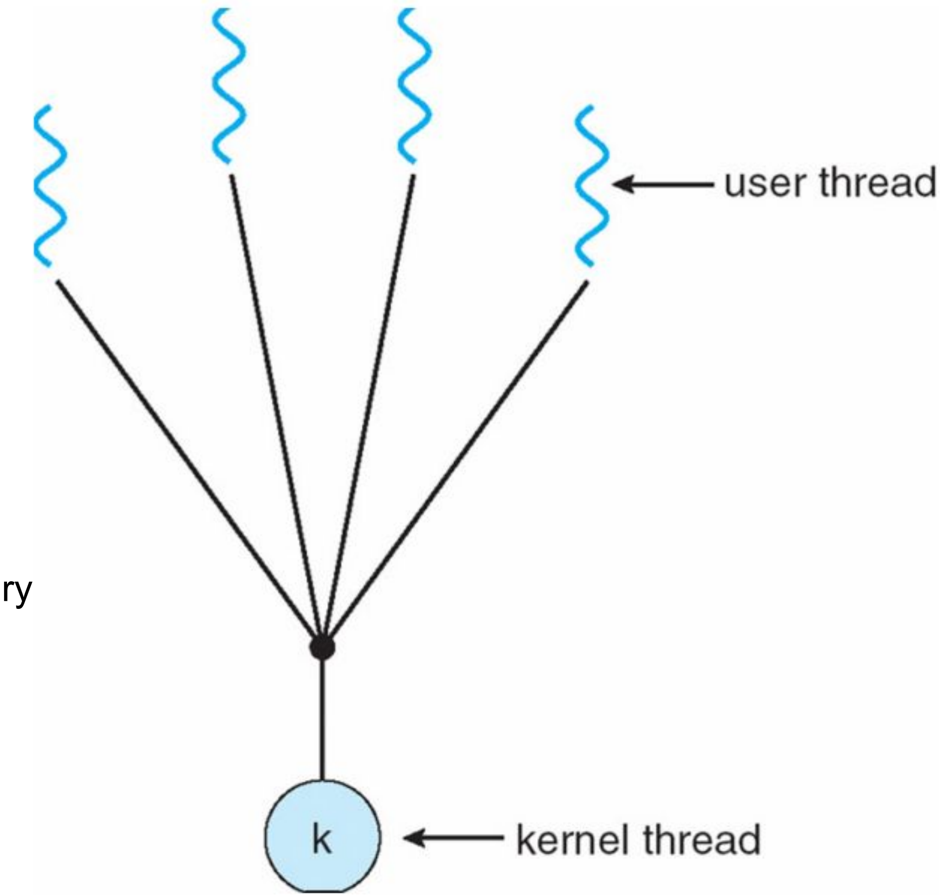
- Every thread operation must go through kernel
  - create, exit, join, synchronize, or switch for any reason
    - syscall takes 100 cycles, fn call 5 cycles
    - Result: threads 10x-30x slower when implemented in kernel

- One-size fits all thread implementation
  - Kernel threads must please all people
  - Maybe pay for fancy features (priority, etc.) you don't need

- General heavyweight memory requirements
  - E.g., requires a fixed-size stack within kernel
  - Other data structures designed for heavier-weight processes

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Alternative: User threads



Implement as **user-level library** (a.k.a. **green threads or fibers**)

- One kernel thread per process
- thread_create, thread_exit, etc., just library functions

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Implementing user-level threads

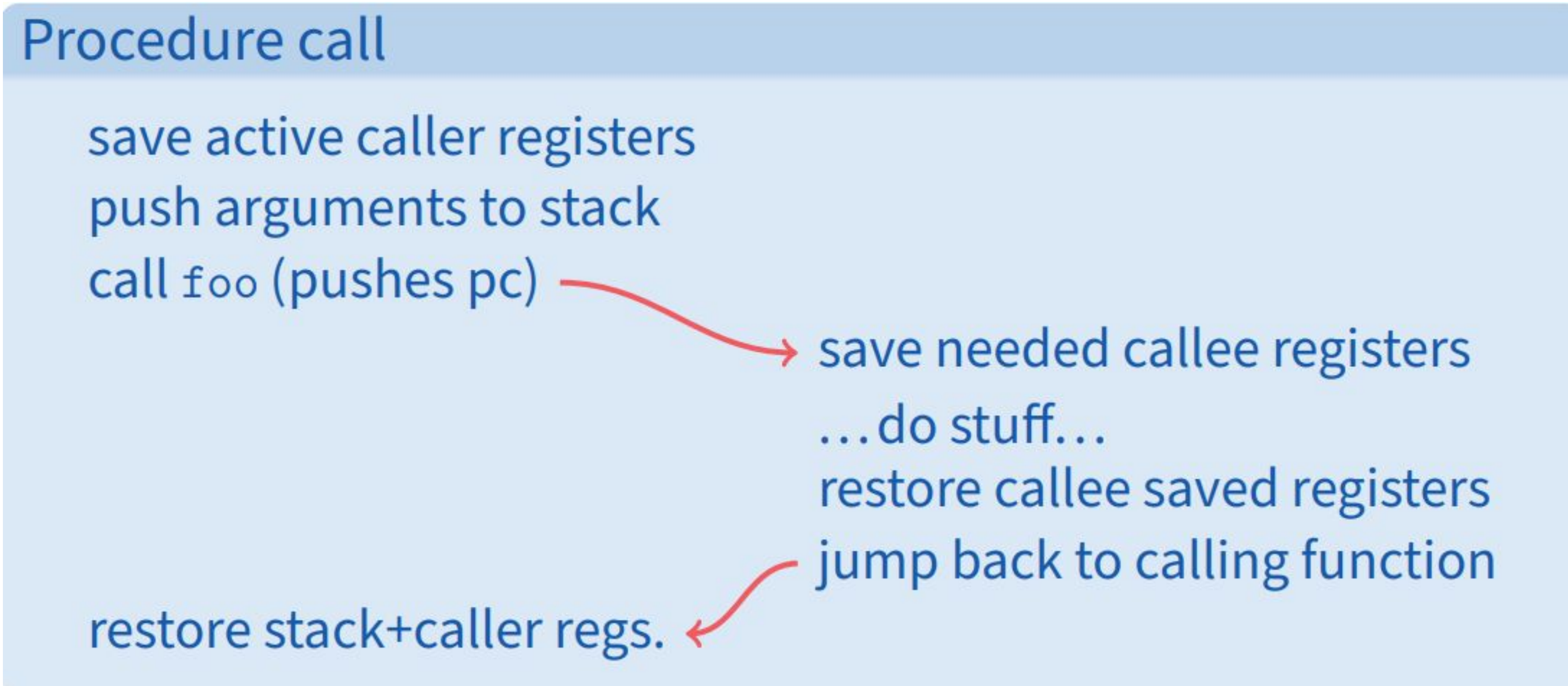
- Allocate a new stack for each thread_create
- Keep a queue of runnable threads
- Replace networking system calls (read/write/etc.)
  - If operation would block, switch and run different thread
- Schedule periodic timer signal (setitimer)
  - Switch to another thread on timer signals (preemption)

Example: Multi-threaded web server

- Thread calls read to get data from remote web browser
- “Fake” read function makes read syscall in non-blocking mode
- No data? schedule another thread
- On timer or when idle check which connections have new data

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Thread implementation details



- Caller must save some state across function call
  - Return address, caller-saved registers
- Other state does not need to be saved
  - Callee-saved regs, global variables, stack pointer

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Thread implementation details

Background: calling conventions

Registers: divided into 2 groups

- **caller-saved regs:** %eax [return val], %edx, & %ecx on x86
- **callee-saved regs** on x86, %ebx, %esi, %edi, plus %ebp and %esp

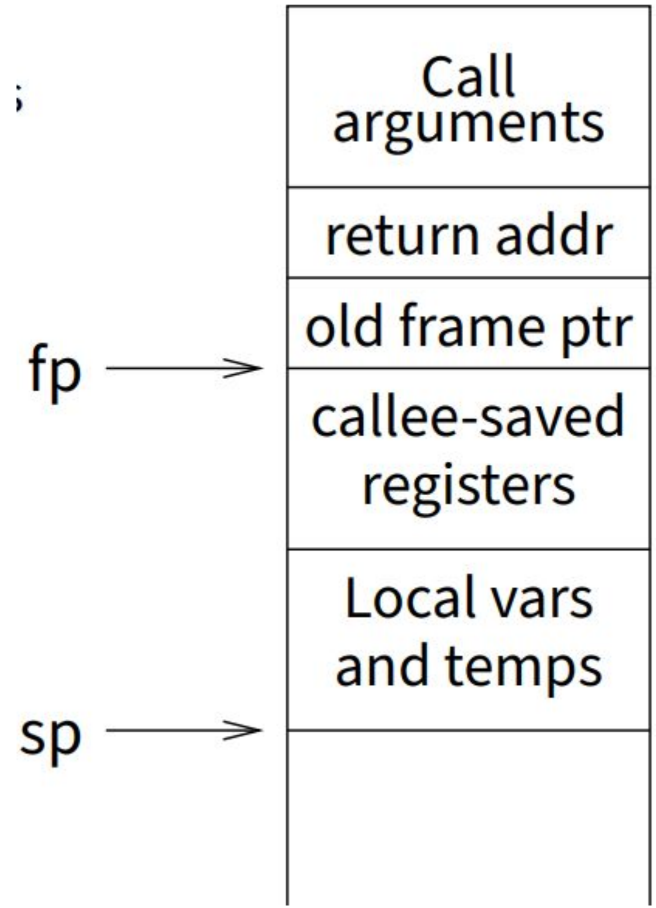
Local variables

- stored in registers and on stack

Function arguments

- go in caller-saved regs and on stack

Functions restore values of calle-saved regs upon return



https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

example from pintos

```
pushl %ebx; pushl %ebp                # Save callee-saved regs
pushl %esi; pushl %edi
mov thread_stack_ofs, %edx            # %edx = offset of stack field
                                      #       in thread struct
movl 20(%esp), %eax                   # %eax = cur
movl %esp, (%eax,%edx,1)              # cur->stack = %esp
movl 24(%esp), %ecx                   # %ecx = next
movl (%ecx,%edx,1), %esp              # %esp = next->stack
popl %edi; popl %esi                  # Restore calle-saved regs
popl %ebp; popl %ebx
ret                                   # Resume execution
```

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Limitations of user-level threads

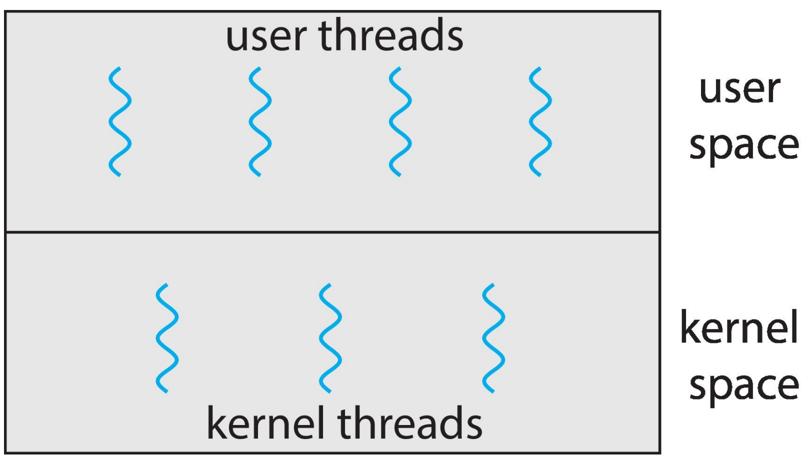
- A user-level thread library can do the same thing as Pintos
- Can't take advantage of multiple CPUs or cores
- A blocking system call blocks all threads
  - Can use O_NONBLOCK to avoid blocking on network connections
  - But doesn't work for disk (e.g., even aio doesn't work for metadata)
  - So one uncached disk read/synchronous write blocks all threads
- A page fault blocks all threads
- Possible deadlock if one thread blocks on another
  - May block entire process and make no progress
  - [More on deadlock in future lectures.]

Benefit:

fast: context switching between user threads within the same process is extremely efficient

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

Multithreading Models: User threads on kernel threads



- Many-to-One
- One-to-One
- Many-to-Many
- two-level

All models maps user-level threads to kernel-level threads.

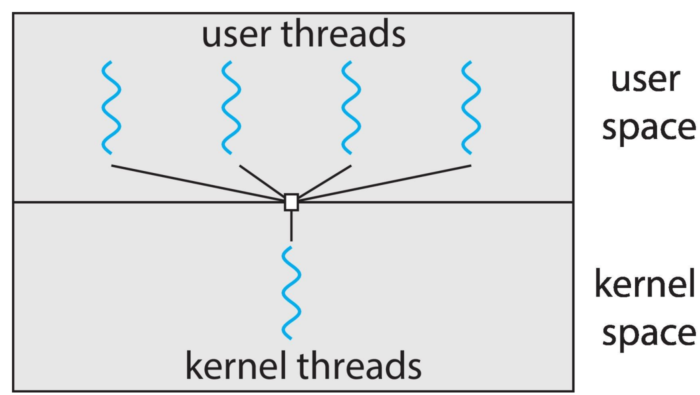
A kernel thread is similar to a process in a non-threaded (single-threaded) system.

The kernel thread is the unit of execution that is scheduled by the kernel to execute on the CPU.

The term **virtual processor** is often used instead of kernel thread.

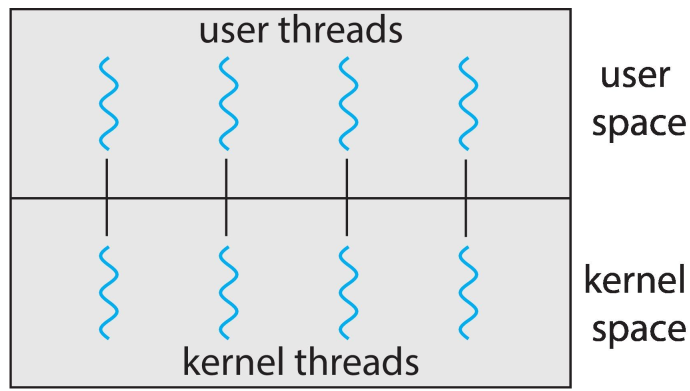
Many-to-One

- 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
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads (aka virtual threads)**
  - **GNU Portable Threads**



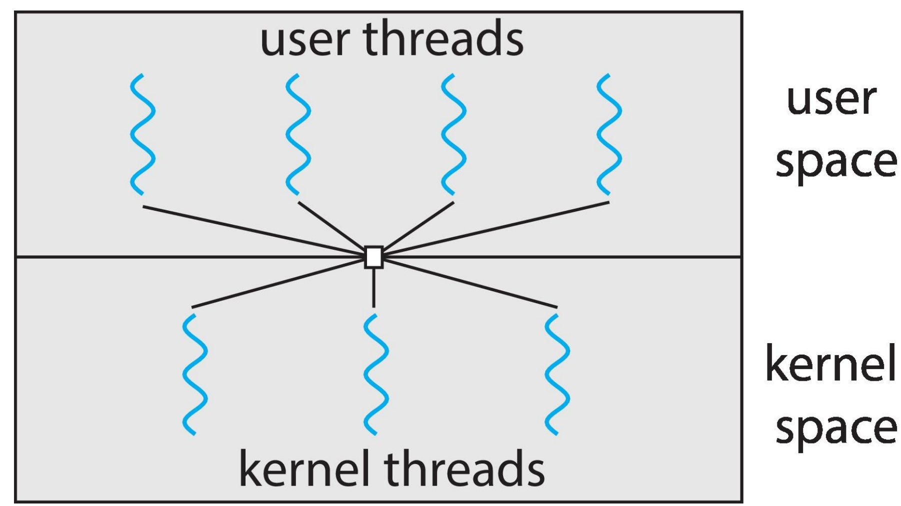
One-to-One

- 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
  - macOS
  - iOS
  - FreeBSD
  - Solaris



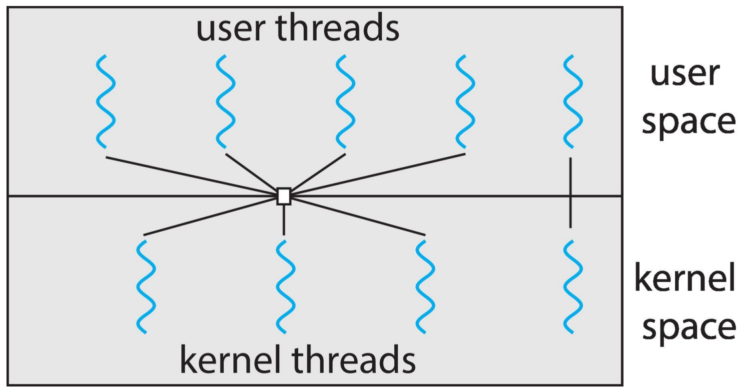
Many-to-Many Model

- 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, **fibers**
    - **scheduling happens at the user level**
- Otherwise not very common



Two-level Model

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



Limitations of n:m threading

- Many of same problems as n : 1 threads
  - Blocked threads, deadlock, . . .
- Hard to keep same # kthreads as available CPUs
  - Kernel knows how many CPUs available
  - Kernel knows which kernel-level threads are blocked
  - But tries to hide these things from applications for transparency
  - So user-level thread scheduler might think a thread is running while underlying kernel thread is blocked
- Kernel doesn't know relative importance of threads
  - Might preempt kthread in which library holds important lock

https://www.scs.stanford.edu/24wi-cs212/notes/processes.pdf

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

Alternative kernel interfaces for threads

User-level library

- Management in application's address space
- High performance and very flexible
- Lack functionality
  - processor blocked during system services

Operating system kernel

- Poor performance (when compared to user-level threads)
- Poor flexibility
- High functionality

Scheduler Activations

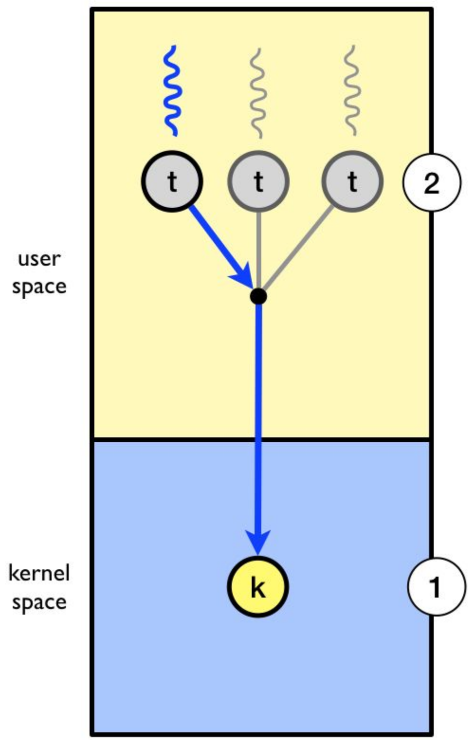
Goal: kernel interface combined with user-level thread package

- Same functionality as kernel threads
- Performance and flexibility of user-level threads

Scheduler activations

- Allow user level threads to act like kernel level threads/virtual processors
- **Notify** user level scheduler of relevant kernel events
  - Like what?
- Provide space in kernel to save context of user thread when kernel stops it
  - E.g., for I/O or to run another application

Scheduler activations: example



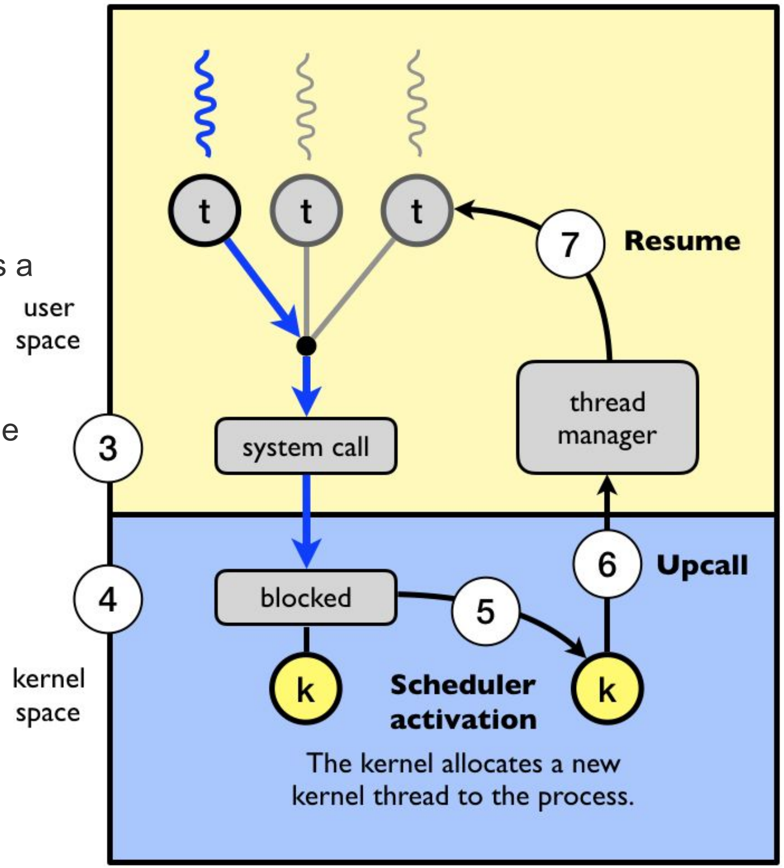
(2). The three user level threads take turn executing on the single kernel-level thread.

(1) to a process with three user-level threads

https://www2.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/#fn:scheduler-activations

3) The executing thread makes a blocking system call

4) the the kernel blocks the calling user-level thread and the kernel-level thread used to execute the user-level thread

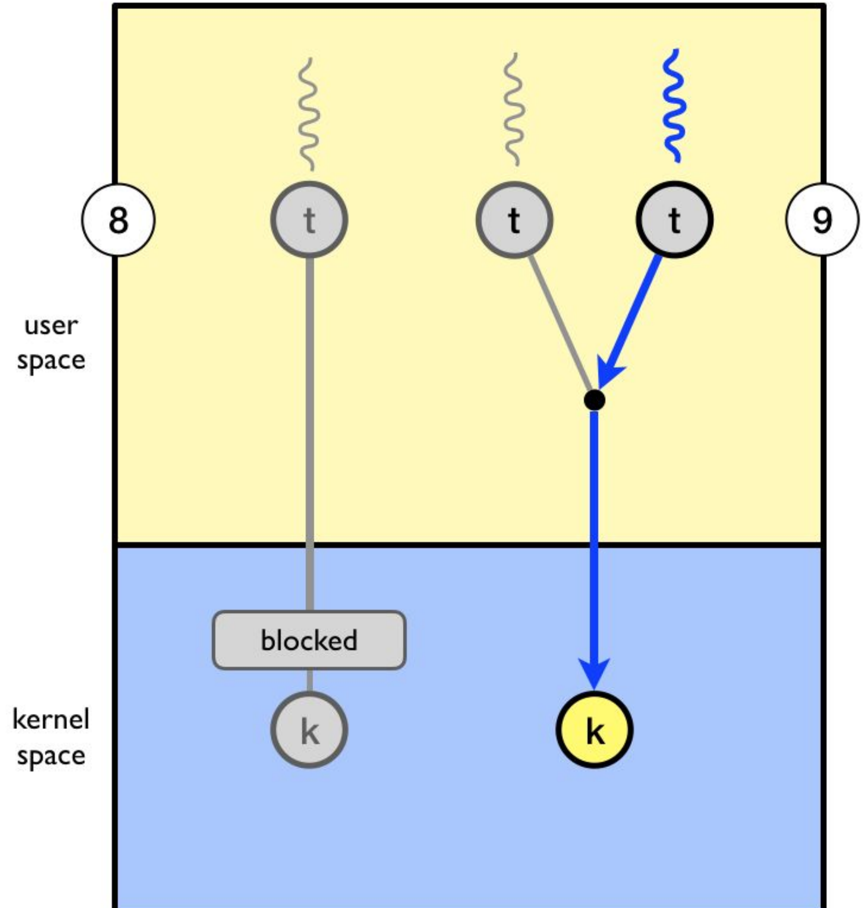


7) The user-level thread manager move the other threads to the new kernel thread and resumes one of the ready threads.

6) **Upcall**: the kernel notifies the user-level thread manager which user-level thread that is now blocked and that a new kernel-level thread is available

5)**Scheduler activation**: the kernel decides to allocate a new kernel-level thread to the process

https://www2.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/#fn:scheduler-activations



8) While one user-level thread is blocked
9) the other threads can take turn executing on the new kernel thread.

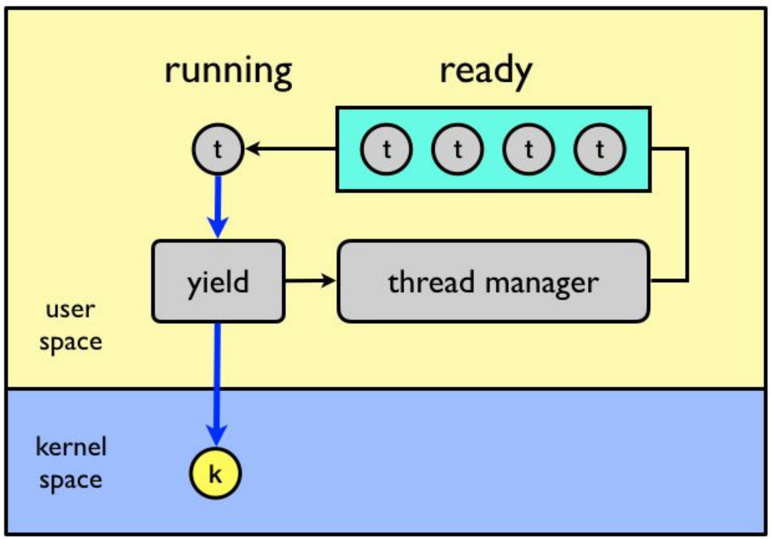
Main Limitation of scheduler activations

- Upcall performance (5x slowdown)

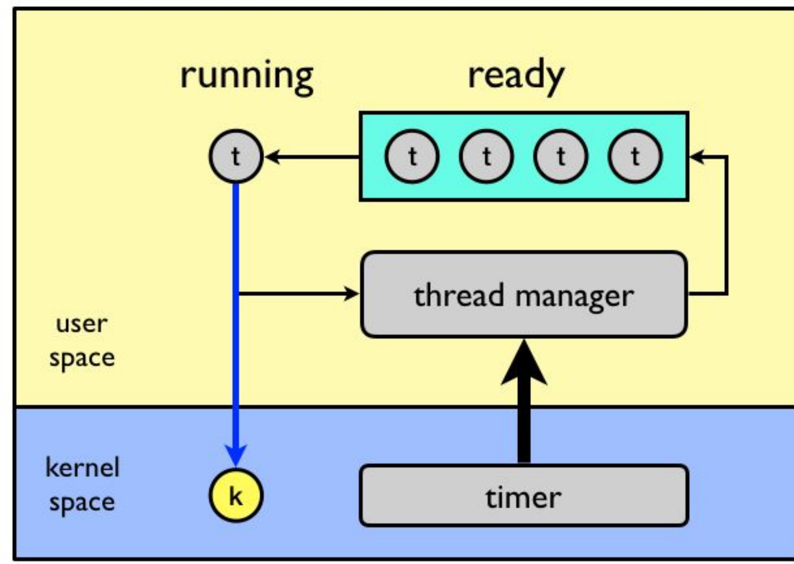
https://www2.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/#fn:scheduler-activations

Scheduling at the user-level

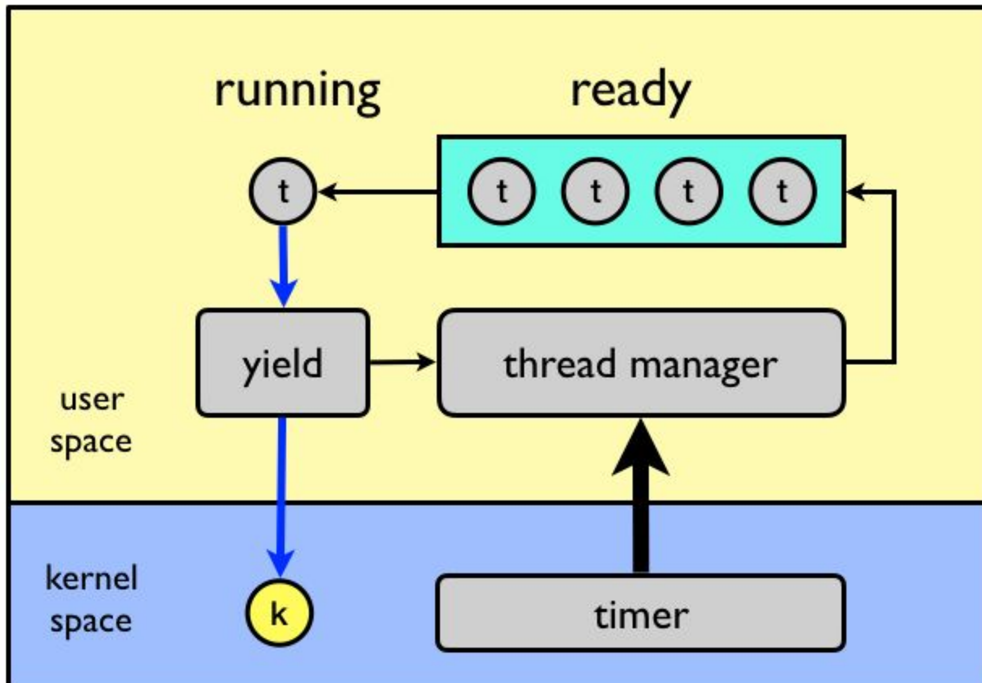
Cooperative (yield)



Preemptive



Or both


running
ready
t
t
t
t
t
yield
thread manager
user
space
kernel
space
k
timer

User-level thread scheduling

Note: User mode cooperatively scheduled threads, fibers or **stackful-coroutines**, are mostly abandoned for various reasons but used in **Go-Goroutines, C++ fibers etc.**

- **Distinguishing coroutines and fibers in C++**
- P1520R0 Response to response to "Fibers under the magnifying glass" (Gor Nishanov)
- Reference: P0866R0 Response to "Fibers under the magnifying glass" (Nat Goodspeed, Oliver Kowalke) #120
- Reference: P1364R0 Fibers under the magnifying glass (Gor Nishanov) #82
- Reference: P0876R5 fiber_context - fibers without scheduler (Oliver Kowalke, Nat Goodspeed)

Operating System Examples

- Windows Threads
- Linux Threads

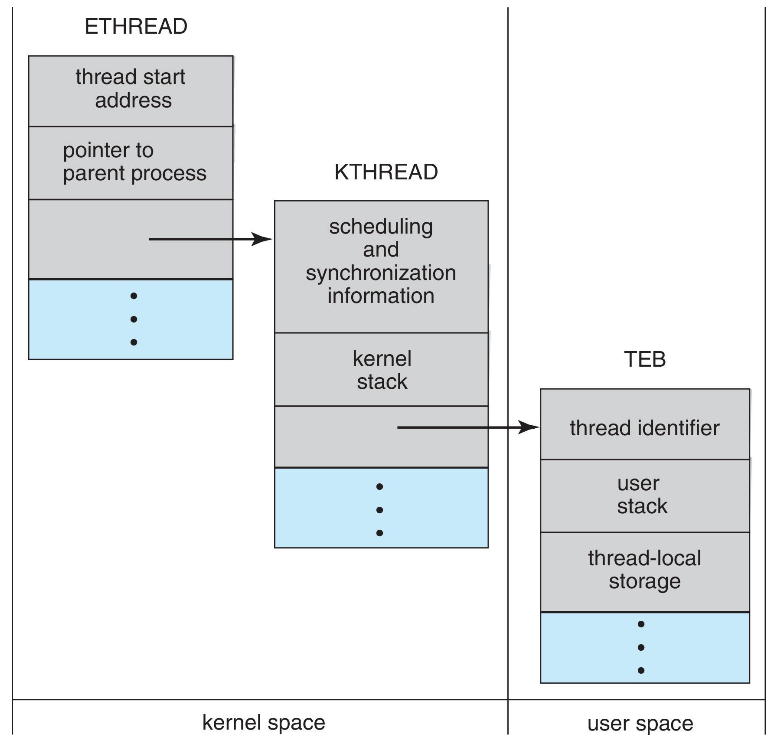
Windows Threads

- Windows API – primary API for Windows applications
- 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

Windows Threads (Cont.)

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows Threads Data Structures



Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone(), clone3()`** 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)

clone(2) - Linux manual page

```
clone, __clone2, clone3 - create a child process
```

#define _GNU_SOURCE /* See **feature_test_macros**(7) */

#include <sched.h>

int clone(int (**fn***)(void *), void ****child_stack***,**

int *flags*, **void ****arg*, **...**

/* pid_t ****ptid*, **struct user_desc ****tls*, **pid_t ****ctid* */);**

clone() creates a new process, in a manner similar to **fork**(2). It is actually a library function layered on top of the underlying **clone**() system call, hereinafter referred to as **sys_clone**. A description of **sys_clone** is given toward the end of this page.

When the child process is created with **clone**(), it executes the function *fn*(*arg*). (This differs from **fork**(2), where execution continues in the child from the point of the **fork**(2) call.) The *fn* argument is a pointer to a function that is called by the child process at the beginning of its execution. The *arg* argument is passed to the *fn* function.

When the *fn*(*arg*) function application returns, the child process terminates. The integer returned by *fn* is the exit code for the child process. The child process may also terminate explicitly by calling **exit**(2) or after receiving a fatal signal.

The *child_stack* argument specifies the location of the stack used by the child process. Since the child and calling process may share memory, it is not possible for the child process to execute in the same stack as the calling process. The calling process must therefore set up memory space for the child stack and pass a pointer to this space to **clone**(). Stacks grow downward on all processors that run Linux (except the HP PA processors), so *child_stack* usually points to the topmost address of the memory space set up for the child stack.

User Level Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Kernel-level library supported by the OS
    - explicit threading
  - Library entirely in user space
    - implicit threading
    - concurrent parts of the program indicated
    - compiler manages threading

Pthreads(you have seen in BIL222)

- 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 (Linux & Mac OS X)

Pthreads Example

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

#include <stdlib.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 */

  /* set the default attributes of the thread */
  pthread_attr_init(&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);
}
```

Pthreads Example (Cont.)

```
/* The thread will execute 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);
}
```

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_join(workers[i], NULL);
```

Windows Multithreaded C Program

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

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

Windows Multithreaded C Program (Cont.)

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

  Param = atoi(argv[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 */

   /* now wait for the thread to finish */
  WaitForSingleObject(ThreadHandle,INFINITE);

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

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

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

Java Threads

Implementing Runnable interface:

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

Waiting on a thread:

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

Java Executor Framework

- Rather than explicitly creating threads, Java also allows thread creation around the Executor interface:

```
public interface Executor
{
   void execute(Runnable command);
}
```

- The Executor is used as follows:

```
Executor service = new Executor;
service.execute(new Task());
```

Java Executor Framework

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
  private int upper;
  public Summation(int upper) {
    this.upper = upper;
  }

  /* The thread will execute in this method */
  public Integer call() {
    int sum = 0;
    for (int i = 1; i <= upper; i++)
      sum += i;

    return new Integer(sum);
  }
}
```

Java Executor Framework (Cont.)

```
public class Driver
{
 public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);

    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));

    try {
        System.out.println("sum = " + result.get());
    } catch (InterruptedException | ExecutionException ie) { }
  }
}
```

Implicit Threading

- 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 Pools

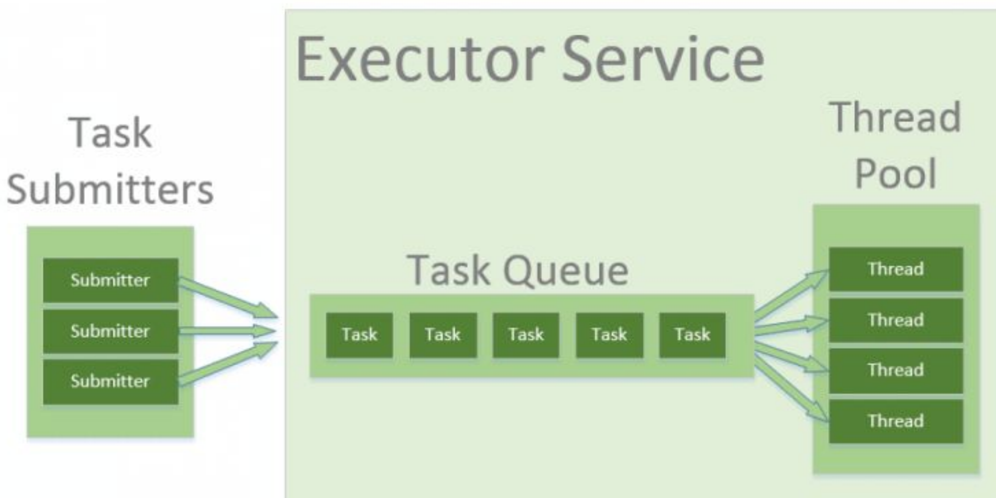
- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from 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.
    */
}
```

see [Using the Thread Pool Functions - Win32 apps | Microsoft Learn]

Java Thread Pools

- Three factory methods for creating thread pools in Executors class:
    - `static ExecutorService newSingleThreadExecutor()`
    - `static ExecutorService newFixedThreadPool(int size)`
    - `static ExecutorService newCachedThreadPool()`



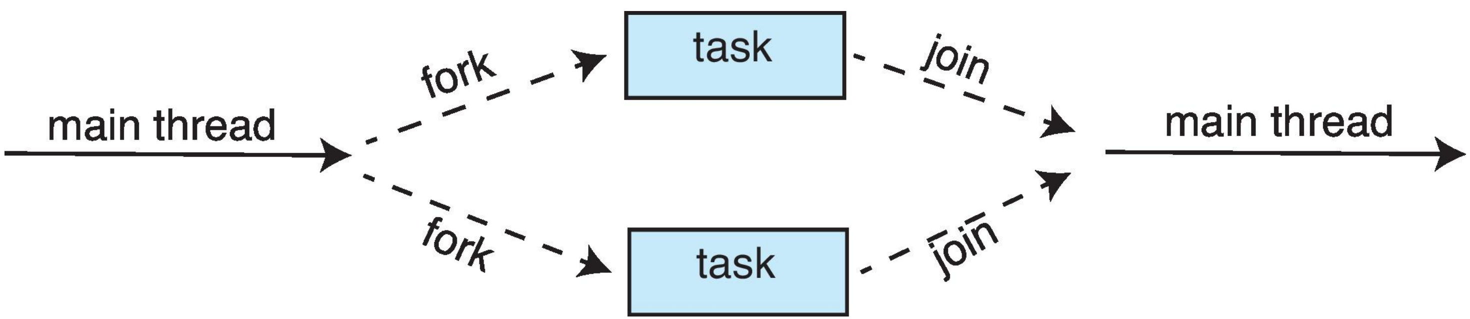
https://www.baeldung.com/thread-pool-java-and-guava

Java Thread Pools (Cont.)

```
import java.util.concurrent.*;

public class ThreadPoolExample
{
public static void main(String[] args) {
   int numTasks = Integer.parseInt(args[0].trim());

   /* Create the thread pool */
   ExecutorService pool = Executors.newCachedThreadPool();

   /* Run each task using a thread in the pool */
   for (int i = 0; i < numTasks; i++)
      pool.execute(new Task());

   /* Shut down the pool once all threads have completed */
   pool.shutdown();
}
```

Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.

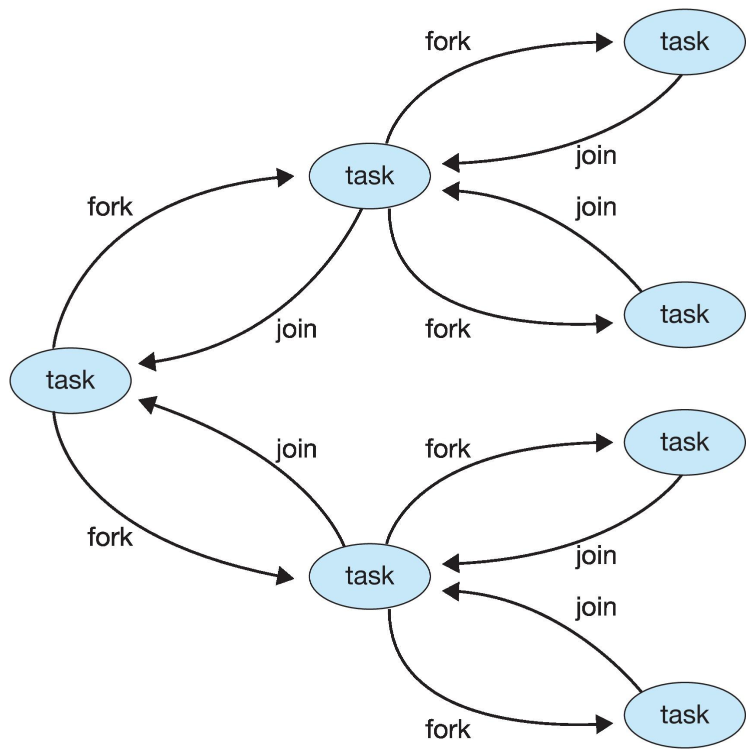


Fork-Join Parallelism

- General algorithm for fork-join strategy:

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

Fork-Join Parallelism



Fork-Join Parallelism in Java

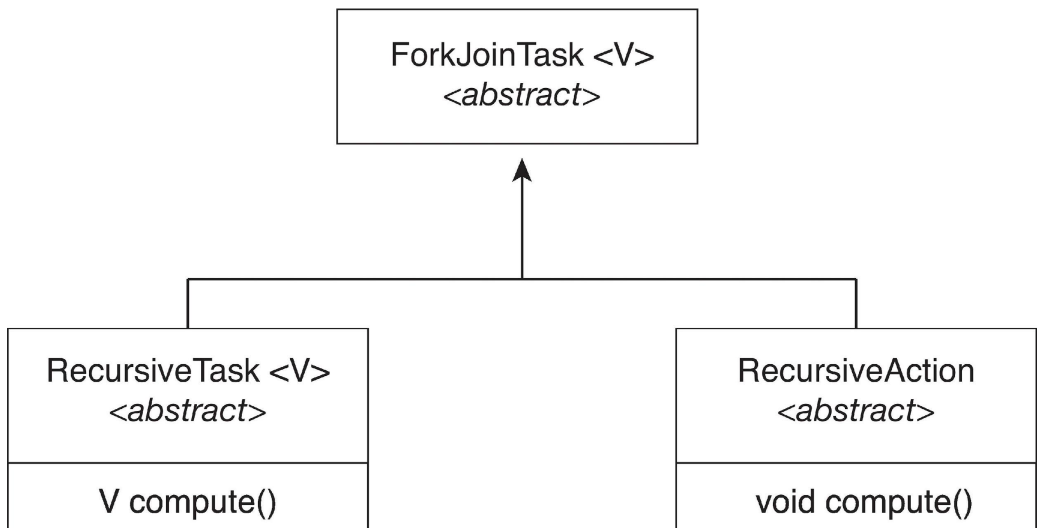
```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

Fork-Join Parallelism in Jav

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
   static final int THRESHOLD = 1000;

   private int begin;
   private int end;
   private int[] array;

   public SumTask(int begin, int end, int[] array) {
      this.begin = begin;
      this.end = end;
      this.array = array;
   }

   protected Integer compute() {
      if (end - begin < THRESHOLD) {
         int sum = 0;
         for (int i = begin; i <= end; i++)
            sum += array[i];

         return sum;
      }
      else {
         int mid = (begin + end) / 2;

         SumTask leftTask = new SumTask(begin, mid, array);
         SumTask rightTask = new SumTask(mid + 1, end, array);

         leftTask.fork();
         rightTask.fork();

         return rightTask.join() + leftTask.join();
      }
   }
}
```

Fork-Join Parallelism in Java

- The **`ForkJoinTask`** is an abstract base class
- **`RecursiveTask`** and **`RecursiveAction`** classes extend **`ForkJoinTask`**
- **`RecursiveTask`** returns a result (via the return value from the **`compute()`** method)
- **`RecursiveAction`** does not return a result



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

```
#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;
}
```

https://www.openmp.org/resources/tutorials-articles/

- Run the for loop in parallel

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

Grand Central Dispatch

- Apple technology for macOS and iOS operating systems
  - Extensions to C, C++ and Objective-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

from wikipedia: It is an implementation of task parallelism based on the thread pool pattern.The fundamental idea is to move the management of the thread pool out of the hands of the developer, and closer to the operating system.

Grand Central Dispatch

- Two types of dispatch queues:
  - **serial** – blocks removed in FIFO order, queue is per process, called **main queue**
    - Programmers can create additional serial queues within program
  - **concurrent** – removed in FIFO order but several may be removed at a time
    - Four system wide queues divided by quality of service:
    - `QOS_CLASS_USER_INTERACTIVE`
    - `QOS_CLASS_USER_INITIATED`
    - `QOS_CLASS_USER_UTILITY`
    - `QOS_CLASS_USER_BACKGROUND`

Grand Central Dispatch

- For the Swift language a task is defined as a closure – similar to a block, minus the caret
- Closures are submitted to the queue using the `dispatch_async()` function:

```
let queue = dispatch_get_global_queue
   (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue,{ print("I am a closure.") })
```

Intel Threading Building Blocks (oneTBB)

https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-onetbb/top.html

Intel® oneAPI Threading Building Blocks (oneTBB) is a runtime-based parallel programming model for C++ code that uses threads.

```cpp
int sum = oneapi::tbb::parallel_reduce(oneapi::tbb::blocked_range<int>(1,101), 0,
   [](oneapi::tbb::blocked_range<int> const& r, int init) -> int {
      for (int v = r.begin(); v != r.end(); v++  ) {
         init += v;
      }
      return init;
   },
   [](int lhs, int rhs) -> int {
      return lhs + rhs;
   }
);
```

Compile a program using pkg-config

To compile a test program test.cpp with oneTBB on Linux* OS and macOS*, provide the full path to search for include files and libraries, or provide a simple line like this:

```
g++ -o test test.cpp $(pkg-config --libs --cflags tbb)
```

Where:

`--cflags` provides oneTBB library include path:

```
$ pkg-config --cflags tbb``
-I<path-to>/tbb/latest/lib/pkgconfig/../..//include
```

`--libs` provides the Intel(R) oneTBB library name and the search path to find it:

```
$ pkg-config —libs tbb
    -L<path to>tbb/latest/lib/pkgconfig/../..//lib/intel64/gcc4.8 -ltbb
```

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- **`exec()`** usually works as normal – replace the running process including all threads

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:
     1. default
     2. 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

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - 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 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);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

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

Thread Cancellation in Java

- Deferred cancellation uses the **`interrupt()`** method, which sets the interrupted status of a thread.

```
Thread worker;

 . . .

/* set the interruption status of the thread */
worker.interrupt()
```

- A thread can then check to see if it has been interrupted:

```
while (!Thread.currentThread().isInterrupted()) {
    . . .
}
```

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