CS-446/646

Threads

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Remember: Processes

A *Process* is a Heavyweight Abstraction

Includes many things

- A *Virtual Address Space* (defining all the Code and Data *Pages*)
- OS *Resources* (e.g. open *Files*) and *Accounting* information
- *Process State* (*Program Counter* (PC), *Stack Pointer* (SP), *Registers*, etc.)

Creating a new *Process* is costly

- A lot of data structures that must be allocated and initialized
  - *Remember:* **`struct proc_t`** (in Solaris)

Communicating between *Processes* is also costly

- Most methods of communication have to go through the OS
  - Overhead of making *System Calls* and copying message data

Concurrent Programs

Recall simple Web server example

- Using **`fork()`** to make copies of itself that each of them would handle some request
- Multiple requests can be handled simultaneously

To execute these Programs we need to:

- Create several *Processes* that execute concurrently
- Have each of them map to the same *Address Space* to share data
  - They can all be thought of as a group, part of the same "computational unit"
    - *Note:* Also, what if one in this group of *Processes* changes its *Virtual Address Space* mappings?
- Have the OS *Schedule* these *Processes* to run simultaneously (logically or physically)

This is inefficient:

- Space: PCB, *Page Tables* (more in later Lecture), etc.
- Time: Create data structures, **`fork()`** and copy *Virtual Address Space*, etc.

Remember: Processes

Rethinking *Processes*

- What would be similar in such "cooperating" *Processes* (e.g. ones created via `fork()`)
  - All should conceptually share the same Code and Data (*Address Space*)
  - All should conceptually share the same *Privileges*
  - All should conceptually share the same *Resources* (*Files*, *Sockets*, etc.)

- What is different in cooperating *Processes*
  - Each should have its own *State of execution*: PC, SP, and *Registers*

➡ Think separately about the concept of a *Process*, and a *State of execution*

- *Process* : *Address Space*, *Privileges*, *Resources*, etc.
- *State of execution*: PC, SP, *Registers*
  - Also called "*Thread of execution*" or just "*Thread*"

Threads & Processes

Modern OSs have the concepts of *Processes* and *Threads*

Thread

- Defines a sequential execution stream within a *Process* (PC, SP, *Registers*)
    - *Threads* are separate streams of executions that share an *Address Space*
    - This allows one *Process* to have multiple points of execution (will exist multiple associated *States* of execution)
        - can potentially use multiple CPUs

Process

- Defines an *Address Space* and general *Process Attributes*
- A *Thread* is bound to a single *Process*
    - A *Process* can however have multiple *Threads*

A *Thread* now becomes the unit of *Scheduling*

- *Processes* are now the containers in which *Threads* execute
- *Processes* become "static", *Threads* are the "dynamic" entities

Thread Control Block (TCB)

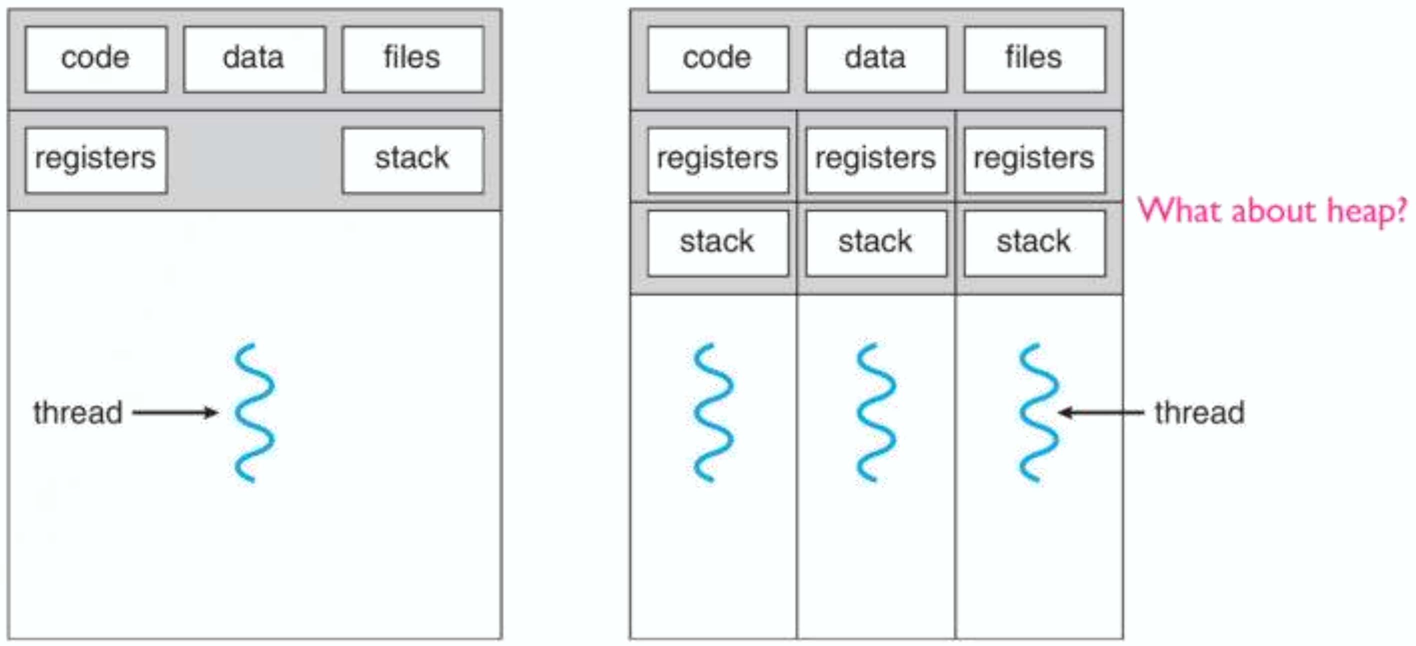
The required data structure (implementation specific)

- Lightweight & Fast

- *Thread Identifier:* Unique id (**`tid`**) assigned to every new *Thread*
- *Execution State* of *Thread* (e.g. *Running, Ready, Waiting, Started, Terminated*)
- *Stack* Pointer (**`ESP`** & **`EBP`** in x86): Points to *Thread's Stack* (in *Process' Address Space*)
- *Program Counter* (**`EIP`** on x86): Points to *Thread's* next *Instruction*
- Other *Register* values of the *Thread*
- Pointer to *Process Control Block* of the *Process* the *Thread* lives in
  - e.g. Pintos *Thread* struct

```c
struct thread {
  tid_t tid; /* Thread identifier. */
  enum thread_status status; /* Thread state. */
  char name[16]; /* Name (for debugging purposes). */
  uint8_t *stack; /* Saved stack pointer. */
  int priority; /* Priority. */
  struct list_elem allelem; /* List element for all threads list. */
  struct list_elem elem; /* List element. */
  unsigned magic; /* Detects stack overflow. */
};
```

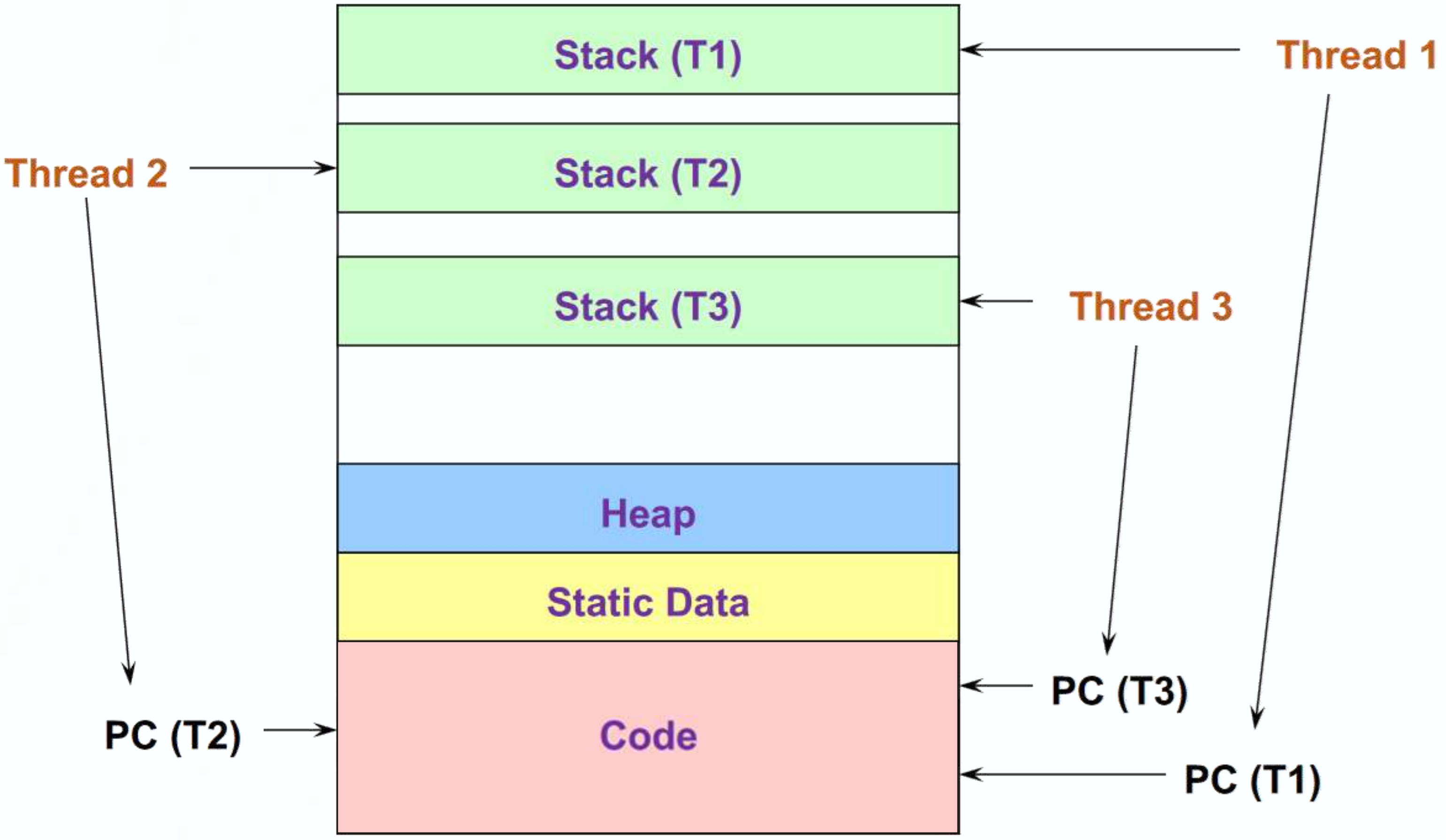
Threads in a *Process*



single-threaded process   multithreaded process

- *Threads* in the same *Process* share the same *Global Memory*
  - *Code* & *Data* and *Heap* Segments
    - I.e. share code, data, files, etc.

Threads in a *Process*



Threads & Processes

Why?

- *Concurrency* does not always require creation of an entirely new *Process*
- Easier to support *Multithreaded* applications

Multithreading can be very useful

- Can unlock "*Parallelism*", i.e. the potential to allocate different *Threads* to multiple cores/CPUs
  - *Note:* Not an easy task, adds in numerous optimization considerations, e.g. *Memory Pollution*
- Improving a Program's structure
- Handling simultaneous *Events* (e.g. web requests)
- Allowing a Program to overlap I/O and computation

So, *Multithreading* is even useful on a single-core Processor

- Although today we can have multicore power-efficient microprocessors in almost everything

- Required software engineering skillset:

Synchronization

Multithreaded Concurrent Server

Using `fork()` to create a new *Process* to handle requests is an overkill:

- *Remember:*

```
while (1) {
  int client_sock = accept(server_sock, addr, addrlen);
  if ((child_pid = fork()) == 0) { // Child
    handle_request( client_sock );
  } else { // Parent
    // Close server socket
  }
}
```

```
void handle_request(int sock) {
  // Process request
  close(sock);
}
```

Multithreaded Concurrent Server

Instead, create a new *Thread* for each request:

➢ *Example:*

```
void run_web_server() {
  while (1) {
    int client_sock = accept(server_sock, addr, addrlen);
    thread_create_interface(handle_request, client_sock);
  }
}
```

?

```
void handle_request(int sock) {
  // Process request
  close(sock);
}
```

Note: This will now execute in a new *Thread*, i.e. will have its own *State of execution* and can be separately *Scheduled*.

Thread API (Unix)

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

- Create a new *Thread* in the calling *Process*, run **`start_routine`** with arguments **`arg`**
  - Can be customized via **`attr`**

```
int pthread_join(pthread_t thread, void **retval);
```

- The **calling** *Thread* waits for the *Thread* specified by **`thread`** to terminate
  - If that *Thread* has already terminated, then returns immediately. The *Thread* specified by **`thread`** must be *Joinable*. If **`retval`** is not **`NULL`**, then it copies the exit status of the target *Thread* (i.e. the value that the target *Thread* supplied to **`pthread_exit()`**)

```
void pthread_exit(void *retval);
```

- Terminates the **calling** *Thread* and returns a value via **`retval`** that (if the *Thread* is *Joinable*) is available to another *Thread* in the same *Process* that calls **`pthread_join()`**

Thread API (Unix)

```c
void* thread_fn(void *arg) {
  int tid = (int)arg;

  printf("thread %d\n", tid);
  printf("is now\n");
  usleep(1);
  printf("running!\n");

  pthread_exit(NULL);
}

int main() {
  pthread_t t1, t2;

  pthread_create(&t1, NULL, thread_fn, (void*)1);
  pthread_create(&t2, NULL, thread_fn, (void*)2);

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

  return 0;
}
```

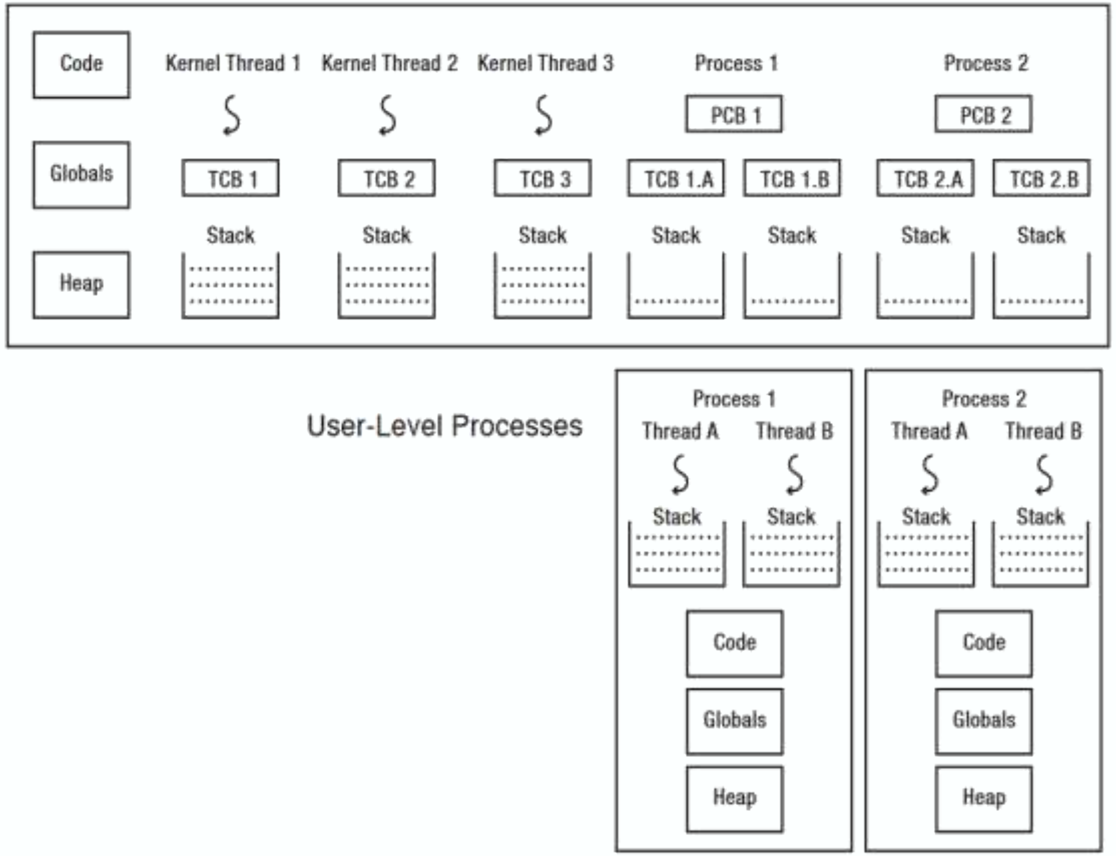
```
$ gcc -o threads threads.c -lpthread
$ ./threads
thread 1
is now
thread 2
is now
running!
running!
```

Thread Implementation

A general prototype:

`… thread_create( … , start_routine, args);`

- Allocate *Thread Control Block* (*TCB*)
- Allocate *Thread Stack*
- Build *Stack Frame* for *Base* of *Thread Stack*
- Put **`start_routine`**'s **`args`** on *Thread Stack*
- Put *Thread* on *Ready* Queue for *Scheduling*



- But where should *Threads* "live"?

Thread Implementation

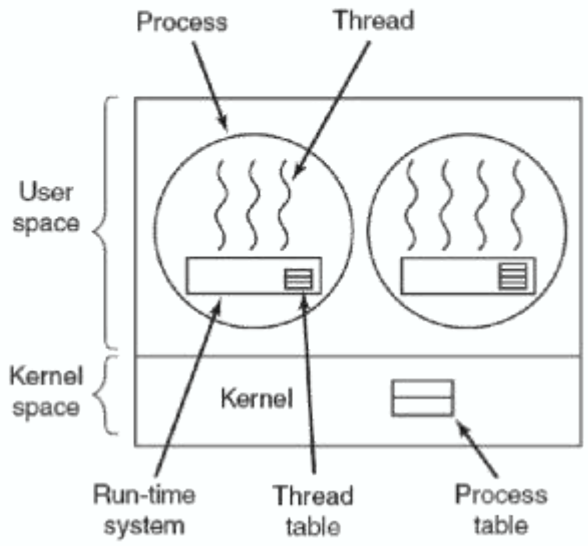
- *Multithreading* Models

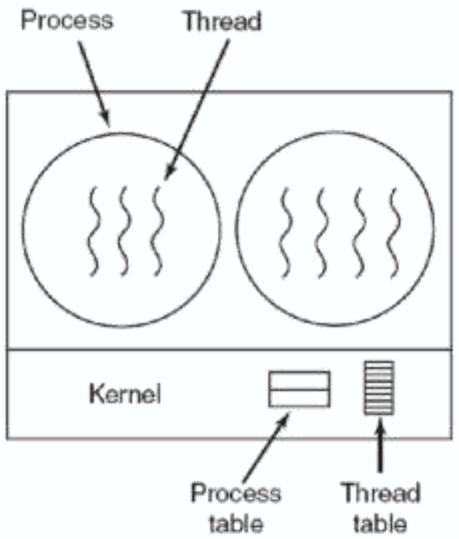
User-Level *Threads*:

- *Thread* management done by User-Level *Thread Library*, Kernel knows nothing

Kernel-Level *Threads*:

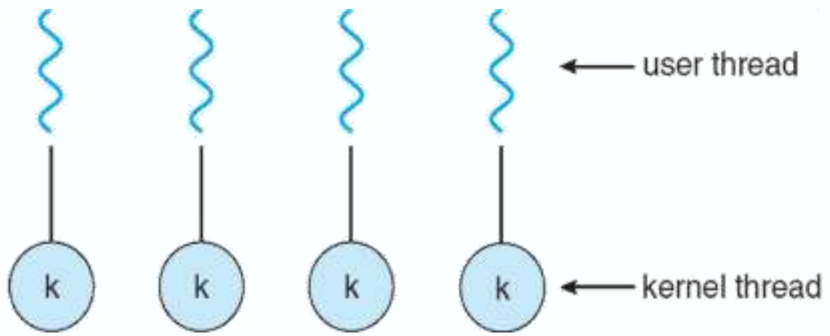
- *Threads* directly supported by the Kernel
  - Virtually all modern OS support Kernel *Threads*





Kernel-Level *Threads*

- All *Thread* operations are implemented in the Kernel by an OS-provided API



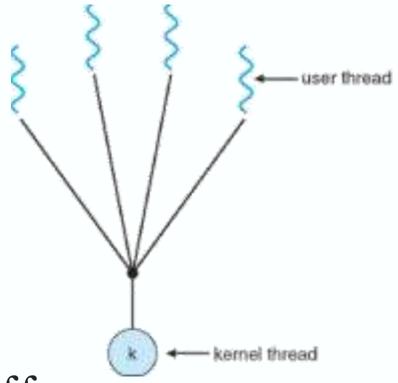
- The OS *Schedules* all the *Threads* in the system
- *Threads* initially called *Lightweight Processes*
  - Windows: *Threads*
  - Solaris: *Lightweight Processes* (**`LWP`**)
  - POSIX *Threads* (**`pthreads`**)
    - *See: Thread Contention Scope* attribute: **`PTHREAD_SCOPE_SYSTEM`**
      The *Thread* competes for resources with all other *Threads* in all *Processes* on the system that are in the same *Scheduling Allocation Domain* (a group of one or more Processors).

Kernel-Level *Thread* Limitations

- Every *Thread* operation must go through the Kernel (*System Call* Interface)
  - Kernel has to do Creation, Exiting, Joining, Synchronizing, Scheduling/Switching
    - On typical laptop: **`syscall`** might take 100 cycles, a **`fn`** call only takes 5 cycles
    - Result: Threads 10x-30x slower when implemented in Kernel
- One-size fits all *Thread* implementation
  - Kernel *Threads* must please every user of the OS
  - User may have to pay for certain features (priority, etc.) that aren't necessary
- General heavy-weight *Memory* requirements
  - E.g. requires its own fixed-size *Thread Stack* within the Kernel
    - *Remember:* In Linux, every *Thread* has its own *User Stack* and its own *Kernel Stack*
  - Other required data structures designed for heavier-weight *Processes*

User-Level *Threads*

- *Alternative:* Implement as a User-Level *Thread Library* (a.k.a. *Green Threads*)



- One Kernel *Thread* per-*Process*
  - Creating, Exiting, Joining, etc. are just functions of a User-Level *Thread Library*
  - Library does *Thread Context Switching*
- User-Level *Threads* are small and fast
  - Java: **`Thread`**
  - POSIX Threads (**`pthreads`**)
    - *Note:* NOT User-Level *Threads*, but see: *Thread Contention Scope* attribute: **`PTHREAD_SCOPE_PROCESS`**
      The *Thread* competes for resources with all other *Threads* in the same *Process* that were also created with the **`PTHREAD_SCOPE_PROCESS`** contention scope, and are scheduled relative to other *Threads* in the *Process* according to their *Scheduling* policy and priority.

User-Level *Thread* Limitations

- Can't take advantage of multiple CPUs or cores

- User-Level *Threads* are invisible to the OS
  - Not directly integrated with the OS

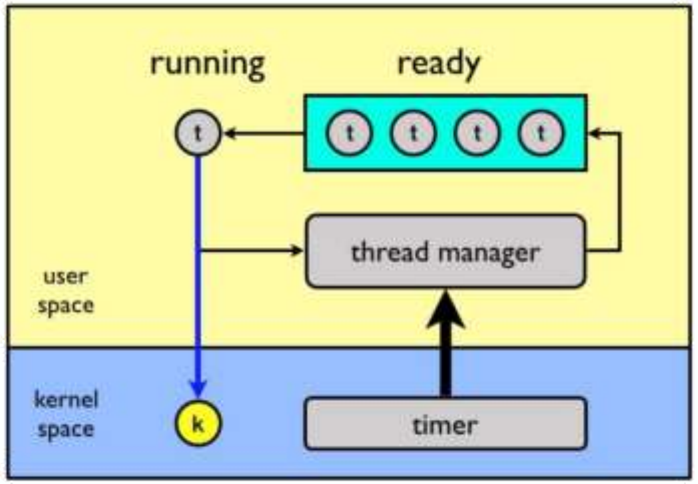
- As a result, the OS can make poor decisions
  - Scheduling a *Process* with ***Idle*** *Threads*
  - A "***Blocking***" *System Call* (e.g. Disk read) blocks all *Threads* of that *Process,* even if the *Process* has other *Threads* that can be executed
  - Unscheduling a *Process* with a *Thread* holding a ***Lock*** (more on this later…)

How to solve this?

- Communication between the Kernel and the User-Level *Thread* Manager (e.g. Windows 8)
  - *See:* Scheduler Activation

Implementing User-Level *Threads*

- Allocate a new *Thread Stack* for each **`thread_create`**
- Keep a *Queue* of *Runnable Threads*
- Schedule periodic Timer *Signal*
  Interval Timer: **`setitimer`**
  - Switch to another *Thread* on timer signals (*Preemption*)
- Replace "*Blocking*" *System Calls* (read/write/etc.) with "*Non-blocking*" versions
  - If operation **`WOULD_BLOCK`**, switch and run different *Thread*
- All these have to be performed/accounted-for at the *User Space*…



Implementing User-Level *Threads*

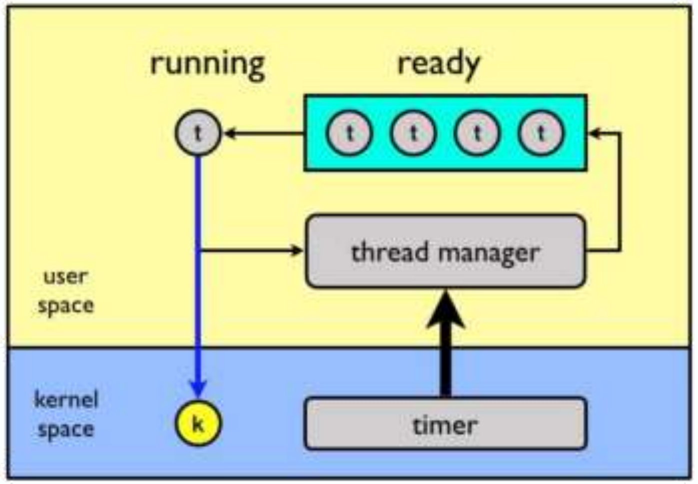
- The *Thread Scheduler* determines when a *Thread* runs

- It uses *Queues* to keep track of what *Threads* are doing
  - Just like the OS and *Processes*
  - But it is implemented at *User-Level* in a Library

- *Run Queue*: *Threads* currently running (usually one)

- *Ready Queue*: *Threads* ready to run

- *Pending Queue*: *Threads* waiting on a *Condition* until they can become *Ready*
  - e.g. *Thread* `sleep()`ing; placed there by *Scheduler* until a certain Interval Timer (OS-managed) expiration occurs, at which point it will move it back to *Ready Queue*



Kernel-Level vs User-Level *Threads*

Kernel-Level *Threads*

- *Good:* Integrated with OS (informed scheduling)
  - Kernel knowing means that when one *Thread* "*Blocks*", another one can be *Scheduled*
- *Bad:* Slower to create, manipulate, synchronize

User-level *Threads*

- *Good:* Faster to create, manipulate, synchronize
- *Bad:* Not integrated with OS (uninformed scheduling)
  - Kernel doesn't know means that when one *Thread* "*Blocks*", all *Threads* in the *Process* will "*Block*"

Understanding their differences is important

- Correct usage affects performance

Multiplexing User-Level *Threads*

Use both Kernel-Level and User-Level *Threads*

- Associate (& Multiplex) User *Threads* with a Kernel *Thread*

- A *Thread Library* is required to map User *Threads* to Kernel *Threads*

Big picture:

- **Kernel-Level *Thread*** : Notionally represent *Physical Concurrency* – How many cores?
- **User-Level *Thread*** : Notionally represent *Application Concurrency* – How many tasks?

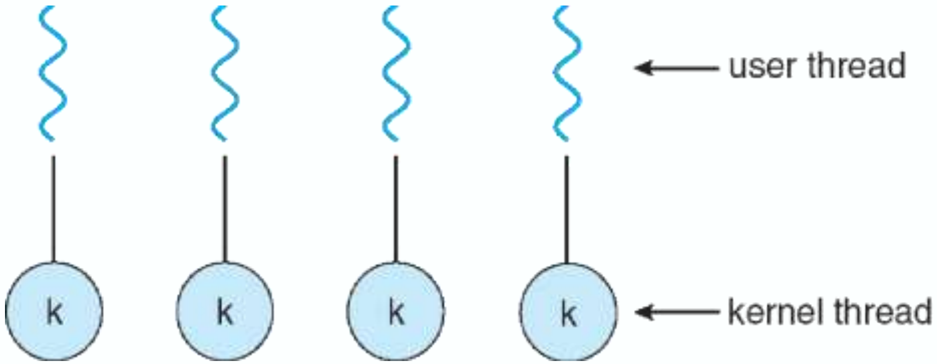
Different mappings exist, representing different tradeoffs

- *One-to-One*: One User *Thread* maps to one Kernel *Thread*
- *Many-to-One*: Many User *Threads* map to one Kernel *Thread*, i.e. Kernel sees a single *Process*
- *Many-to-Many*: Many User *Threads* map to many Kernel *Threads*

Multiplexing User-Level *Threads*

One-to-One (*1 : 1 Threading*)

- One User-Level *Thread* maps to one Kernel-Level *Thread*
- *Good:* More *Concurrency*
    - When one *Thread* "*Blocks*", others can run
    - Better multicore / multiprocessor performance
- *Bad*: Expensive
    - *Thread* operations involve Kernel
    - *Threads* need Kernel Resources
        - e.g. *Memory*



Note 1: The **`PTHREAD_SCOPE_SYSTEM`** contention scope typically indicates that a *User-Space Thread* is bound directly to a single *Kernel-Scheduling* entity. This is the case on Linux for the obsolete LinuxThreads implementation and the modern NPTL implementation, which are both 1:1 *Threading* implementations.
Linux supports **`PTHREAD_SCOPE_SYSTEM`**, but not **`PTHREAD_SCOPE_PROCESS`**.

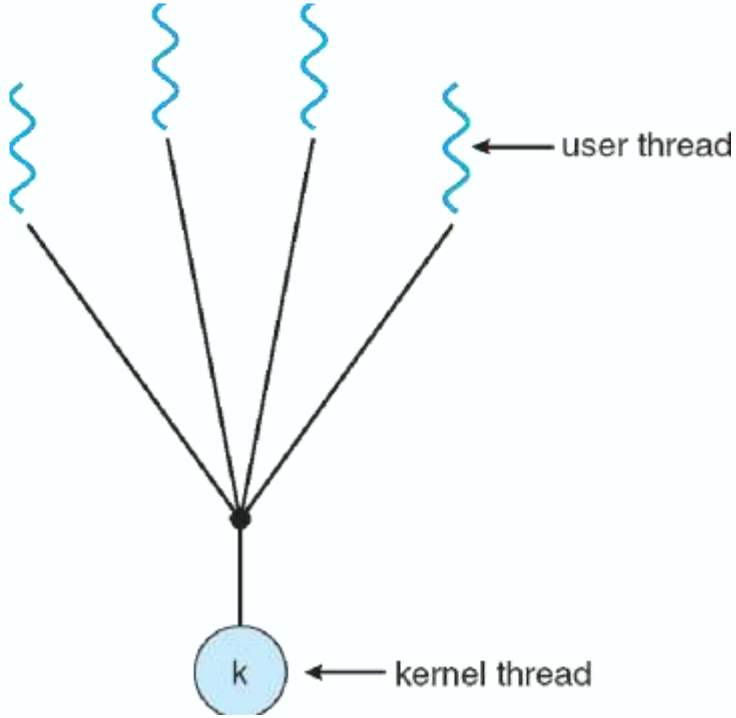
Note 2: There is no difference between *Thread* and *Process* in Linux. There only exists the concept of *Thread Group Identifier* (TGID) which allows to determine what is shared and what is not between created *Threads.*

Multiplexing User-Level *Threads*

Many-to-One (*n : 1 Threading*)

- Many User-Level *Threads* maps to one Kernel-Level *Thread*
- *Good:* Fast
  - No *System Calls* required

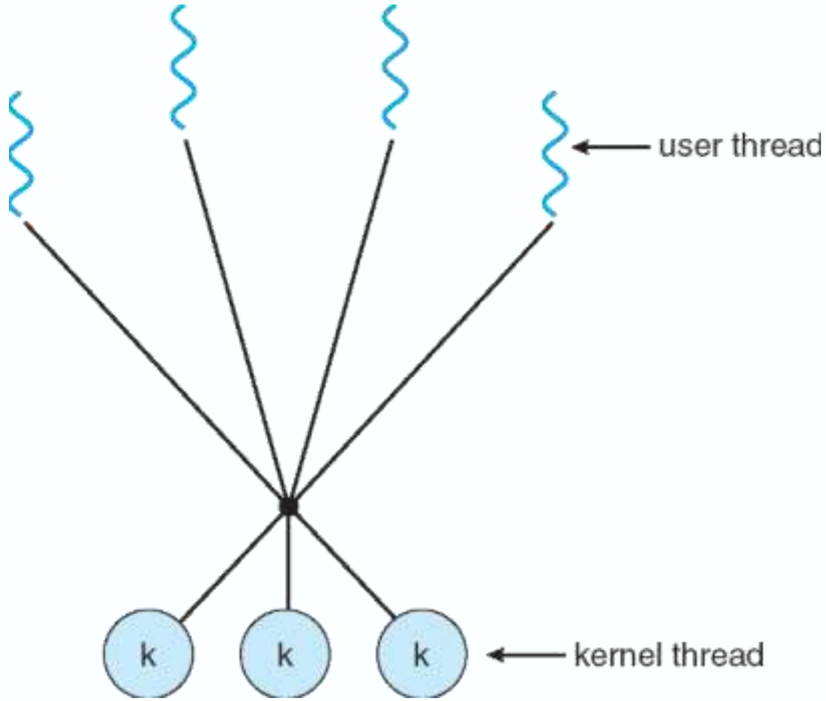
  Portable
  - Few system dependencies
- *Bad*: No *Parallel* execution of *Threads*
  - All *Threads "Block"* when e.g. one waits for I/O
- e.g. Java Virtual Machine (JVM) (also C#, others)
  - Java *Threads* are User-Level *Threads*
    - Can multiplex all Java *Threads* on one Kernel *Thread*



Multiplexing User-Level *Threads*

Many-to-Many* (*n : m Threading – n: User, m: Kernel **)**

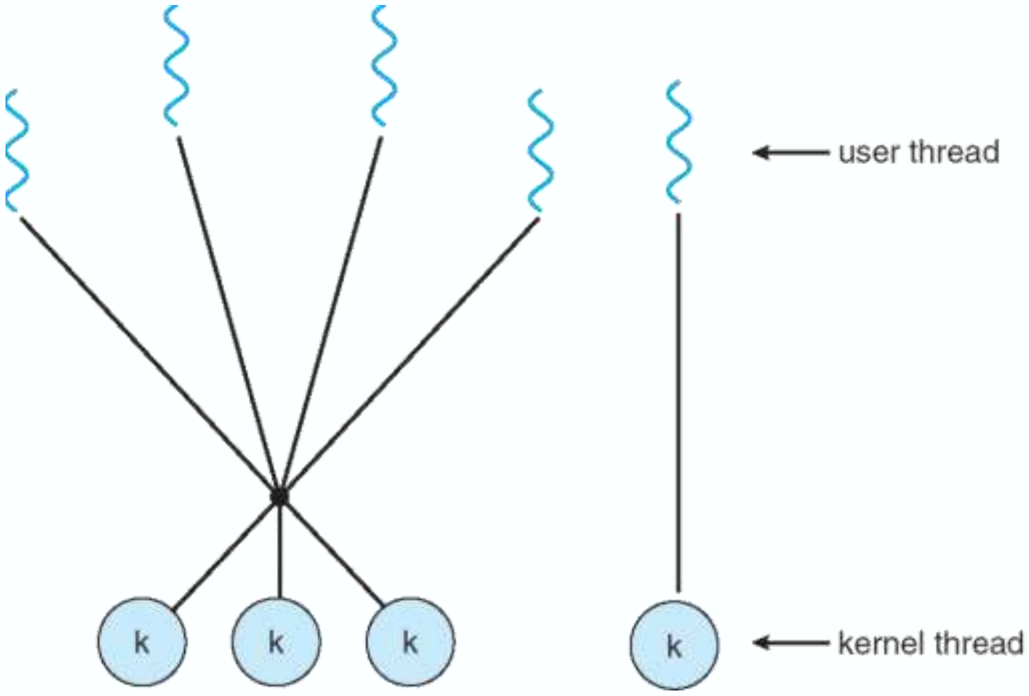
- Many User-Level *Threads* maps to many Kernel-Level *Threads*
  - n ≥ m
- **`thread_create`**, **`thread_exit`** still library functions as before
- *Good:* Flexible
  - OS creates Kernel *Threads* for *Physical Concurrency*
  - Applications create User *Threads* for *App Concurrency*
- *Bad*: Complex Implementation
  - Most programs use 1:1 *Threading* model anyway
- e.g. Java Virtual Machine (JVM) (also C#, others)
  - Java *Threads* are User-Level *Threads*
    - Can multiplex all Java *Threads* on one Kernel *Thread*



Multiplexing User-Level *Threads*

Two-Level

- Similar to *Many-to-Many*
  - Except that a User-Level *Thread* can be **bound** to a Kernel-Level *Thread*



Thread Pool

Creating a *Thread* for each request is costly

- Also, the created *Thread* will exit (and would normally be destroyed) after serving a request

- More User requests increase the number of required *Threads* → server overload

Thread Pool

- Pre-create/allocate a number of *Threads* waiting for work
- *Waking* up a *Thread* to serve User request
  - Much faster than from-scratch *Thread* creation
- When request is done, don't exit
  - *Return* to *Thread Pool*
- Imposes a limit to the max number of *Threads*
  - *Note:* Linux does not have a *Thread Pool*, so its number of Threads limitation (**`/proc/sys/kernel/threads-max`**) is not associated to this concept

- *Remember:* From a *Process*' point-of-view, there is no *Threads* limitation

Thread Miscellanea

- Semantics of **`fork()`** *System Call*
  - Should **`fork()`** duplicate only the calling *Thread* or all *Threads* of a *Process*?
    - Think about other active *Threads,* or other *Threads Trapped* in another *System Call*, etc.
  - POSIX **`fork()`** copies only the calling *Thread*

    `fork(2)`: A *Process* shall be **created with a single *Thread*.** If a Multi-Threaded *Process* calls **`fork()`**, the new *Process* shall contain **a replica of the calling *Thread* and its entire *Address Space***, possibly including the states of *Mutexes* and other *Resources*. Consequently, to avoid errors, the *Child Process* may only execute *Async-Signal-Safe* operations until such time as one of the **`exec`** functions is called.
    - Effectively, entire *Memory* is duplicated (including all the *Registers*), but the *Child Process* will only have one active *Thread,* i.e. only the **calling *Thread*** will be in a **non-suspended** State
      - i.e. other *Threads* are "there" but cannot run anymore as the system never assigns any CPU time to them; they are in fact missing in the Kernel *Thread Scheduler Table*

  Note: Potentially dangerous to call **`fork()`** from a Multi-Threaded process as any *Mutexes* (*Synchronization Mechanisms*, more on these later on…) held in non-calling *Threads* will be forever **locked** in the *Child Process*.

Thread Miscellanea

- *Signal* handling (Which *Thread* should *Signals* be delivered to?)
  - POSIX **`pthreads`** requires all *Threads* in a *Process* to:
    - Share some *Process*-wide attributes, including:
      - *Signal Dispositions* (how the *Process* behaves when it is delivered the *Signal*: ***`Term`***, ***`Ign`***, …)
    - Have some distinct attributes, e.g: *Signal Mask*, *Alternate Signal Stack*, etc.

  POSIX.1 distinguishes the notions of ***Signals* that are directed to the *Process* as a whole** and *Signals* that are directed to individual *Threads*. According to POSIX.1, a *Process*-directed *Signal* (sent using **`kill()`** for example) should be handled by a **single, arbitrarily selected Thread within the Process**.

  `signal(7)`: A *Signal* may be generated (and thus pending) for a *Process* as a whole (e.g. when sent using **`kill()`**) or for a specific *Thread* (e.g. certain *Signals*, such as **`SIGSEGV`** and **`SIGFPE`**, generated as a consequence of executing a specific machine-language instruction are *Thread*-directed, as are *Signals* targeted at a specific *Thread* using **`pthread_kill()`**). A *Process*-directed *Signal* may be delivered to any one of the *Threads* that does not currently have the *Signal* blocked. If more than one of the *Threads* has the *Signal* unblocked, then the Kernel chooses an **arbitrary** *Thread* to which to deliver the *Signal*.

  - Linux's implementation will by default try to deliver to the *Main Thread* first, unless another *Thread* has been nominated by the User

Non-Preemptive Thread Scheduling

- *Threads* voluntarily give up the CPU by `yield()`ing

```
int sched_yield (void);
```

Causes the calling *Thread* to relinquish the CPU. The *Thread* is moved to the end of the *Queue* for its *Static Priority* (more on that in later *Scheduling* Lecture) and a new *Thread* gets to run.

Ping *Thread*

```
while (1) {
  printf("ping\n");
  sched_yield();
}
```

Pong *Thread*

```
while (1) {
  printf("pong\n");
  sched_yield();
}
```

Note: Strategic calls to `sched_yield()` can improve performance by giving other *Threads* or *Processes* a chance to run … Avoid calling `sched_yield()` unnecessarily or inappropriately (e.g., when resources needed by other schedulable *Threads* are still held by the caller), since doing so will result in unnecessary *Context Switches*, which will degrade system performance.

Non-Preemptive Thread Scheduling

- *Threads* voluntarily give up the CPU by **`yield()`**ing

`int sched_yield (void);`

Causes the calling *Thread* to relinquish the CPU. The *Thread* is moved to the end of the *Queue* for its *Static Priority* (more on that in later *Scheduling* Lecture) and a new *Thread* gets to run.

- In other words, it *Context Switches* to another *Thread*

- So **`sched_yield()`** returns once we have *Context-Switched* back to the calling *Thread*
  - e.g. because another *Thread* called **`sched_yield()`**

Ping / Pong execution trace:

```
printf("ping\n");
sched_yield();
printf("pong\n");
sched_yield();
…
```

Preemptive Thread Scheduling

- *Non-Preemptive Threads* have to voluntarily give up CPU
  - A long-running *Thread* will take over the machine
  - Only voluntary calls to **`sched_yield`**, **`sleep`**, or finishing, will lead to a *Context Switch*

Preemptive Scheduling

- Causing involuntary *Context Switching*
  - Need to regain control of CPU *Asynchronously*
  - Use Timer *Interrupt*
  - Timer *Interrupt Handler* forces current *Thread* to "call" **`sched_yield`**

Thread Context Switching

Context Switching in the same *Process*, such that *Virtual Memory Address Space* is not switched

- Which would otherwise involve *Memory Address Mappings*, *Page Tables*, and Kernel *Resources*

Note: Without additional considerations (e.g. *Weak Affinity*) *Thread Context Switching* can lead to *Memory Pollution*

The *Context Switch* Routine does all of the magic

- Saves context of the *outgoing* (**`current`**) *Thread* (PC, CPU *Registers*)
  - Push all machine State onto the top of the *Kernel Stack* of the *outgoing Thread*
- Restores context of the *incoming* (**`new`**) *Thread*
  - Pop all machine State from the *Kernel Stack* of the *incoming Thread* and loads it into *CPU Registers*
- The *incoming Thread* becomes the **`current`** *Thread*
- Return control to caller of the *incoming Thread*

All this has to be done in *Assembly* language

- Works at the level of the Procedure *Calling Conventions*, so itself it cannot be implemented by using Procedure calls

Background: ***Calling Conventions***

A standard on how functions should be implemented and called by the *Machine*

- How a *Function Call* in C or C++ gets converted into *Assembly* language
  - How arguments are passed to a function, how return values are passed back out of a function, how the function is called, and how the function manages the *Stack* and its *Stack Frame*, etc.
- Compilers need to obey this standard when compiling code into Assembly
  - Set up the *Stack* and CPU *Registers* properly

Why ?

- A Program calls functions across many object files and libraries
  - To be able to interface all of these, we need a standardization for how function calls take place

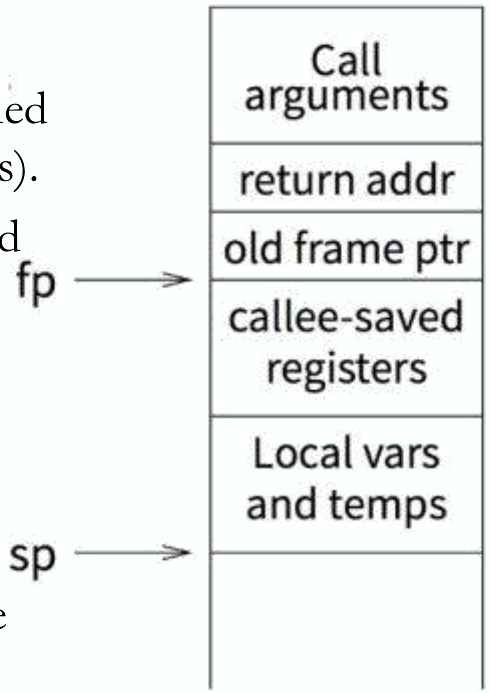
Background: ***Calling Conventions***

x86 *Calling Convention* – *Stack* setup

SP points to "bottom" of *Stack* (lower *Virtual Addresses*). As parameters are pushed on the *Stack*, the SP advances (downwards).

During a method call, variables are pushed on the *Stack* and the SP advances more. Parameters passed to the subroutine constantly change their *relative* offset w.r.t. the SP.

The FP points to the start (*Base*) of the *Stack Frame* and does not move during the subroutine call.
Parameters passed to the subroutine remain at a *constant* offset w.r.t. the FP.




```c
int compute(int a, int b)
{
  int i, result;
  result = 0;
  for (i = 0; i < a; i++)
    result = result + b - i;
  return result;
}
void foo()
{
  int x, y, z;
  x = 3;
  y = 5;
  z = compute(x, y);
  printf("compute(%d, %d)=%d\n", x, y, z);
}
```

Background: ***Calling Conventions***

Registers divided into 2 groups:

- *Caller-saved / Volatile Regs*: Hold temporary quantities that need not be preserved across calls, i.e. their values aren't needed after the next function call returns
  - Caller's responsibility to push these *Registers* onto the *Stack* or copy them somewhere else if it wants to restore these values after a procedure **`call`**
  - Considered normal for a **`call`** to "clobber" ("*Call-Clobbered Regs*") temporary values in these *Regs*, i.e. the Callee function is free to modify these
    - on x86, **`%eax`** [return val], **`%edx`**, **`%ecx`**
- *Callee-saved / Non-Volatile Regs*: Caller expects these to hold the same value after **`call`** (Callee returns)
  - Callee's responsibility to restore these to their original values before returning to the Caller, or to ensure it doesn't touch them ("*Call-Preserved Regs*")
    - on x86, **`%ebx`**, **`%esi`**, **`%edi`**, **`%ebp`**, **`%esp`**

Background: ***Calling Conventions***

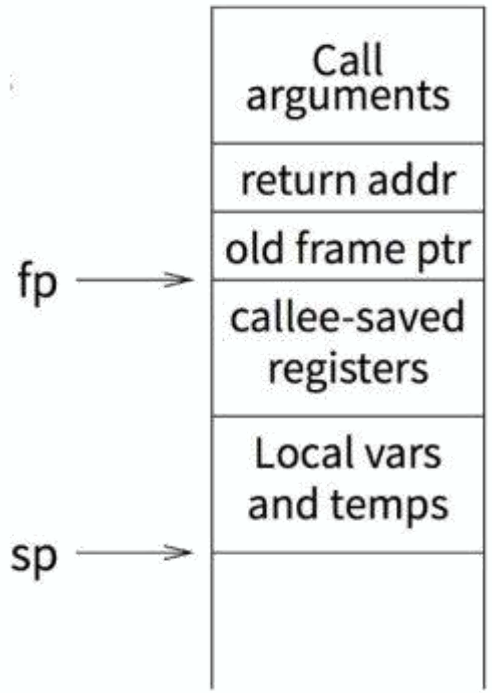
Registers divided into 2 groups:

- *Caller-saved / Volatile Regs*:
  - Caller's responsibility to push these *Registers* onto the *Stack* or copy them somewhere else if it wants to restore these values after a procedure **`call`**
- *Callee-saved / Non-Volatile Regs*:
  - Callee's responsibility to restore these to their original values before returning to the Caller (or to ensure it doesn't touch them)

Save active *Caller Registers*
`call foo` (pushes PC)

Save used *Callee Registers*
… do stuff …
Restore *Callee Saved Registers*
Return to Caller

Restore *Caller Registers*



Remember: **Pintos *Thread Implementation***

Pintos *Thread Control Block* (TCB) Structure:

```
struct thread {
  tid_t tid;
  enum thread_status status;
  char name[16];
  uint8_t *stack; /* Saved stack pointer. */
  int priority;
  struct list_elem allelem;
  struct list_elem elem;
  unsigned magic; /* Detects stack overflow. */
};
```

`stack` field is where the value of **`%esp`** (*Stack Pointer* (SP)) when a *Thread* is *Preempted* will be saved.

➢ *Note:* In x86 Assembly we can't write **`t->stack`**, we can instead take the *Address* of the **`struct thread`** in memory and the offset (in bytes) of the field:

```
uint32_t thread_stack_ofs = offsetof(struct thread, stack);
```

Remember: Threads have no owned *Heap*

/* Each thread structure is stored in its own 4 kB page. The thread structure itself sits at the very bottom of the page (at offset 0). The rest of the page is reserved for the thread's kernel stack, which grows downward from the top of the page (at offset 4 kB) */

```
                      4 kB +---------------------------------+
                           |          kernel stack           |
                           |                |                |
                           |                |                |
                           |                V                |
                           |         grows downward          |
                           |                                 |
                           |                                 |
                           |                                 |
                           |                                 |
                           |                                 |
                           |                                 |
                           |                                 |
                           |                                 |
 sizeof (struct thread)    +---------------------------------+
                           |              magic              |
                           |                :                |
                           |                :                |
                           |              status             |
                           |               tid               |
                      0 kB +---------------------------------+
```

Example: **Pintos *Thread Context Switching***

Pintos C declaration:

```
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

➢ Intuitively: To switch to another *Thread*, we just need to “switch the *Stacks*” (because every *Thread* is guaranteed to be running **`switch_threads`** at the point it was *Preempted* itself).
Do so by changing the value of **`%esp`**.

➢ **`switch_threads`** is implemented in *Assembly*

➢ “Bottom” of Stack after calling **`switch_threads`**:
Remember: Lower *Virtual Addresses*

0x0C00	next
0x0BFC	cur
0x0BF8	Return address

esp 0x0BF8

(*Remember:* Calling Convention specifies pushing **`args`** (i.e. **`next`**, **`cur`**) for called function (i.e. **`switch_threads`**), then return address before making a **`call`**)

Example: **Pintos** ***Thread Context Switching***

- **`switch_threads`** first saves *Callee-Saved Registers* on *Stack* of **`cur`**rent *Thread* (the one being *Preempted*)

```
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi
```

Save *Callee-Saved Regs* of the **`cur`**rent *Thread*

(esp + 24) 0x0C00	next
(esp + 20) 0x0BFC	cur
(esp + 16) 0x0BF8	Return address
(esp + 12) 0x0BF4	ebx
(esp + 8) 0x0BF0	ebp
(esp + 4) 0x0BEC	esi
(esp + 0) 0x0BE8	edi

esp 0x0BE8

Example: **Pintos *Thread Context Switching***

- **`switch_threads`** loads the value of the **`cur`**rent *Thread*'s (the one being *Preempted*) *Stack Pointer* via accessing the **`thread_stack_ofs`** into the **`%edx`** *Register*, as we will need it to perform switching

```
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi
mov thread_stack_ofs, %edx
```

`thread_stack_ofs` is the offset of **`stack`** field in **`thread`** struct

Example: **Pintos *Thread Context Switching***

- **`switch_threads`** saves the *Stack Pointer* of the **`cur`**rent *Thread*, and sets **`%esp`** to point to the (previously saved) *Stack Pointer* of the **`next`** *Thread* to run

```
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

mov thread_stack_ofs, %edx

movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp
```

```
%eax = cur
cur -> stack = %esp
%ecx = next
%esp = next -> stack
```

Save *Stack Pointer* of the **`cur`**rent *Thread*

Load *Stack Pointer* of the **`next`** *Thread*

`SWITCH_CUR(%esp)` is equivalent to **`20(%esp)`**, or address **`esp+20`**, which holds the call arg **`cur`**

`SWITCH_NEXT(%esp)` is equivalent to **`24(%esp)`**, or address **`esp+24`** , which holds the call arg **`next`**

Example: Pintos *Thread Context Switching*

- **`switch_threads`** has effectively now switched *Threads* (**`%esp`**). It finally restores *Callee-Saved Regs* it had pushed onto the **`next`** *Thread*'s *Stack* at the time that one was *Preempted*, and **`return`**s from the call to **`switch_threads`** into the **`next`** *Thread*'s frame (at the time that itself previously had **`call`**ed **`switch_threads`**)

```
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

mov thread_stack_ofs, %edx

movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

popl %edi
popl %esi
popl %ebp
popl %ebx

ret
```

Restore *Callee-Saved Regs* of the **`next`** *Thread*

return from **`switch_threads()`** in the **`next`** *Thread* (happened when it was *Context-Switched*)

Example: Pintos *Thread Context Switching*

- `switch_threads` in overview:

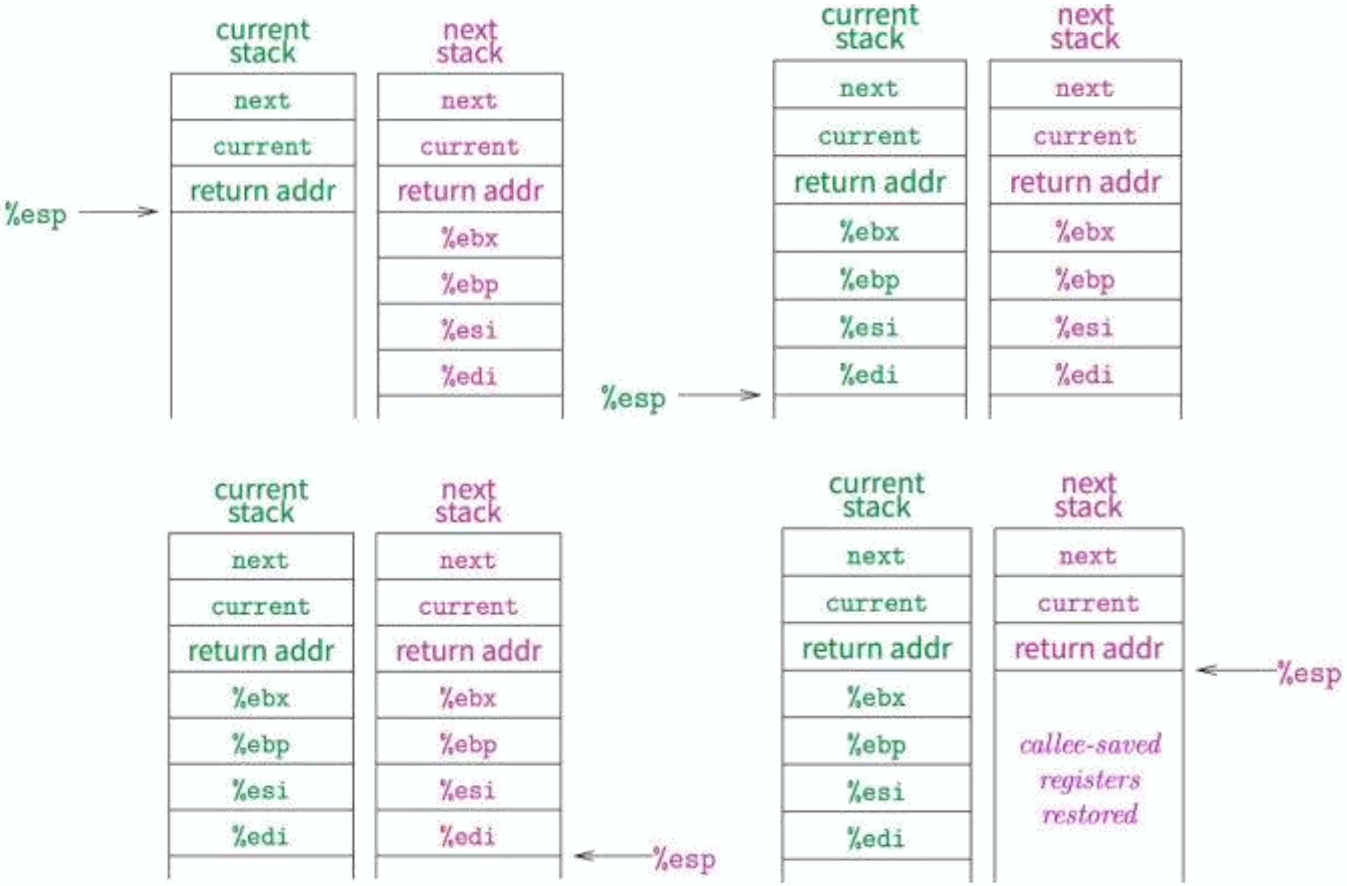
```
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

mov thread_stack_ofs, %edx

movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

popl %edi
popl %esi
popl %ebp
popl %ebx

ret
```



Example: **Pintos *Process Context Switching***

Process Switching: A combination of *Thread Switching* & *Kernel* / *User Space* switching

- When a *Process* is running (in *User Space*), a Timer *Interrupt* will yield control of the CPU back to the Kernel, which will result in an *Interrupt Handler*
  - If *Time Slice* has been used up, also sets flag for *Interrupt Handler* to know it should perform *Context Switching*
- The *Interrupt Handler* will call a **`thread_yield()`** function which calls **`schedule()`** to find the **`next`** *Thread* to run. This will now be ready to call **`switch_threads(cur, next)`**.
- If the **`next`** Kernel-Level *Thread* is associated with a *Process*, switching back to *User Space* takes place
  - Pintos also follows the *1:1 Threading* model
- This is done by **`process_activate()`**
  - Takes place after **`switch_threads()`** but before returning from the *Interrupt Handler*
- In x86, sets *Control Register 3* (CR3 – holds *Physical Address* of *Page Directory Table* for specific *Process*) for the *Process* that is now running, and saves **`%esp`** (*Thread Stack Pointer*) of (the newly (re)run) Kernel-Level *Thread* to *Task State Segment* (*TSS*)
  - *Remember:* Kernel *Thread Stack* ≠ User *Thread Stack* of *Process*
    - i.e. we need to remember the *Thread*'s Kernel State separately

CS-446/646

Time for Questions !