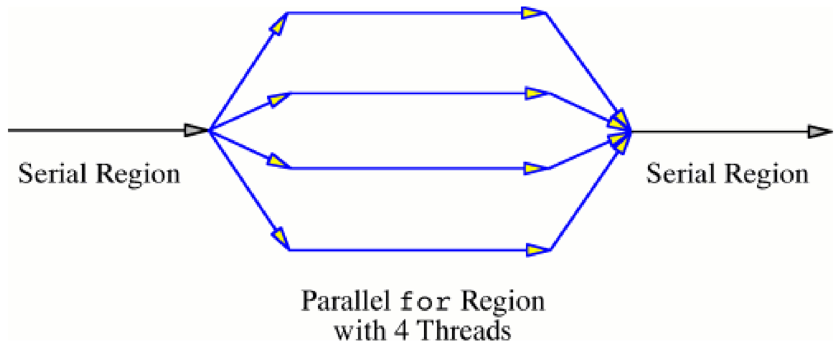
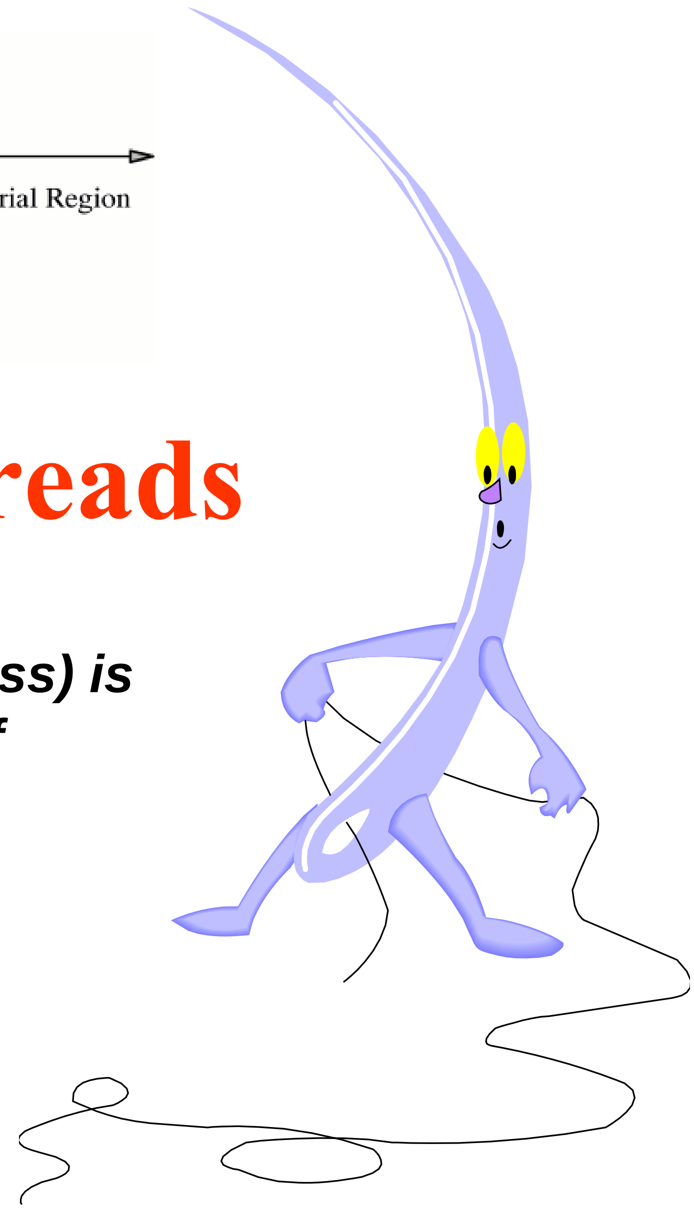
exploring the living universe...

COMP 3230

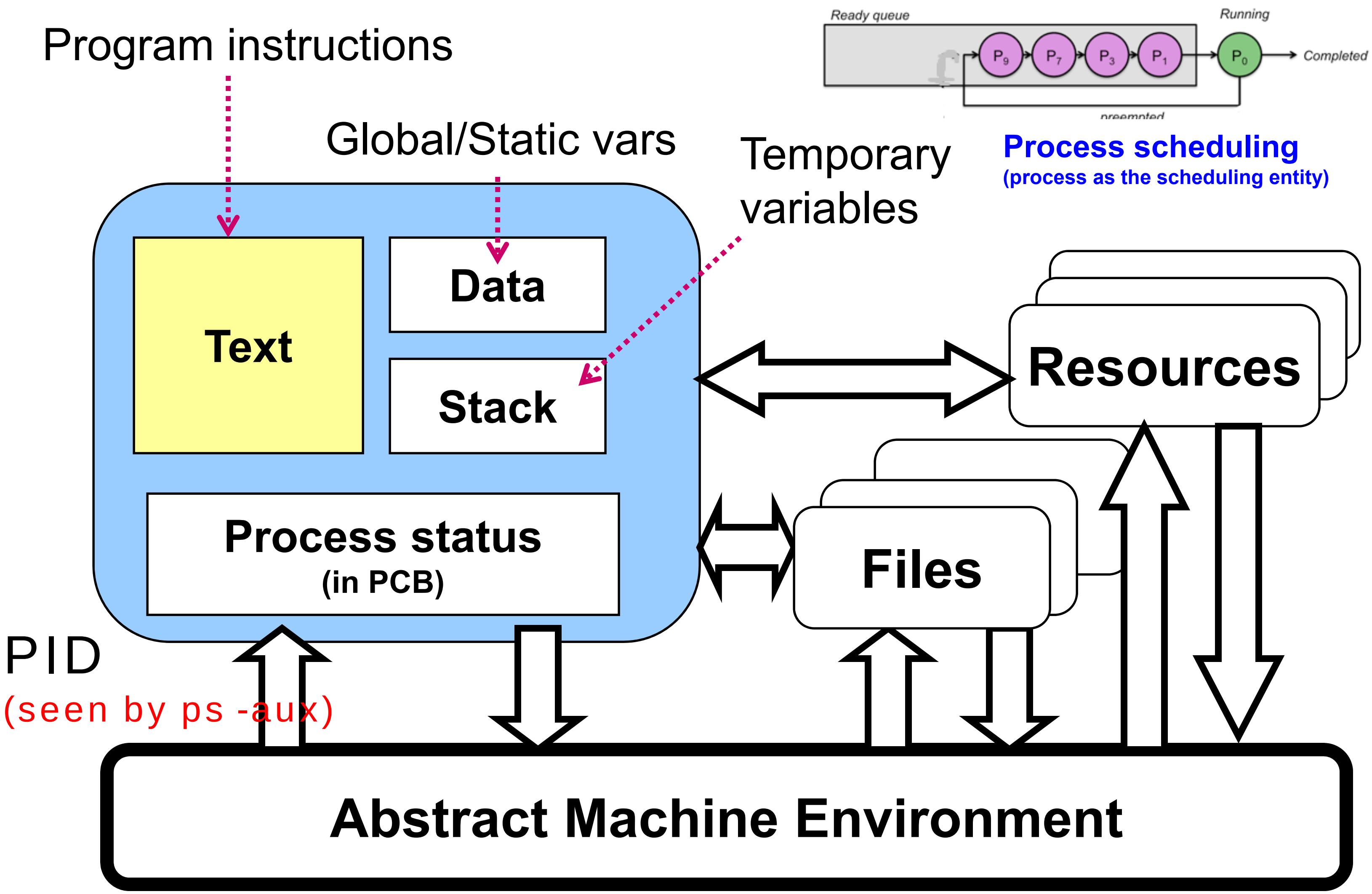


Lecture 5 Threads

A thread (or light-weight process) is an encapsulation of the flow of control in a program.



Remember What is a Process



A Process = *text* + *data* + *stack* + *heap* + other OS resources.

What are threads?

- Threads within a process <u>share</u>
  - Text segment (instructions)
  - Data segment (global and static variables)
  - BSS segment (uninitialized data)
  - Heap (dynamically allocated data)
  - File descriptors (if file is open , all threads can read/write to it)
  - Signals mask
  - Current working directory
  - User ID and group ID

What are threads? (2)

What are NOT shared!

- Each thread has its own ***stack***
  - used to keep its own local variables, temporary variables, return addresses **when they execute a function call**.
- Each thread has its own set of register values (e.g., ***program counter (PC), stack pointer* (SP)**) that are loaded when the thread is active and saved when it becomes inactive. We call these "**CPU context**".
- See figures in next few slides!

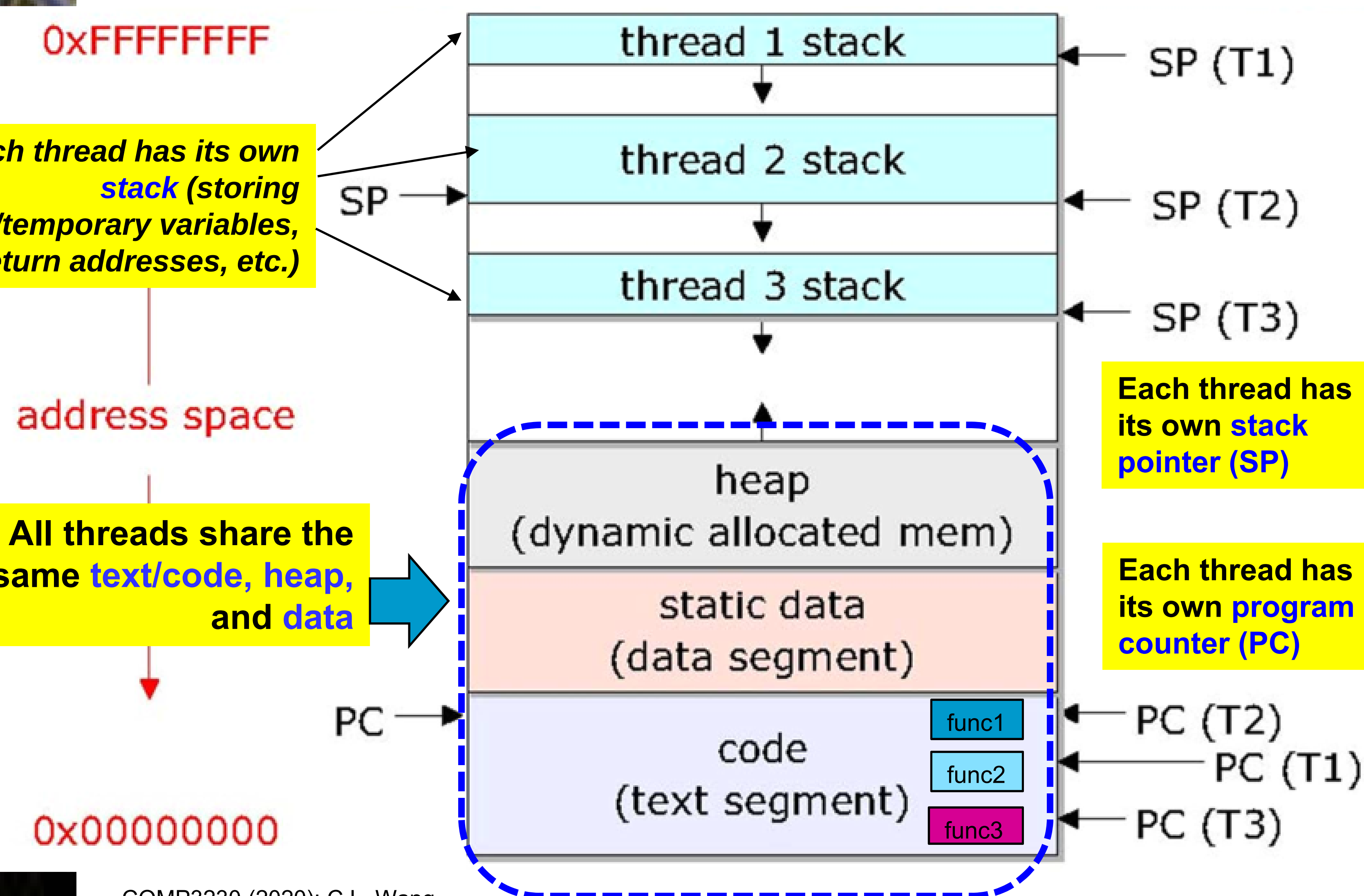
Address Space of a Multi-threaded Program (1)

Each thread has its own stack (storing local/temporary variables, return addresses, etc.)

All threads share the same text/code, heap, and data

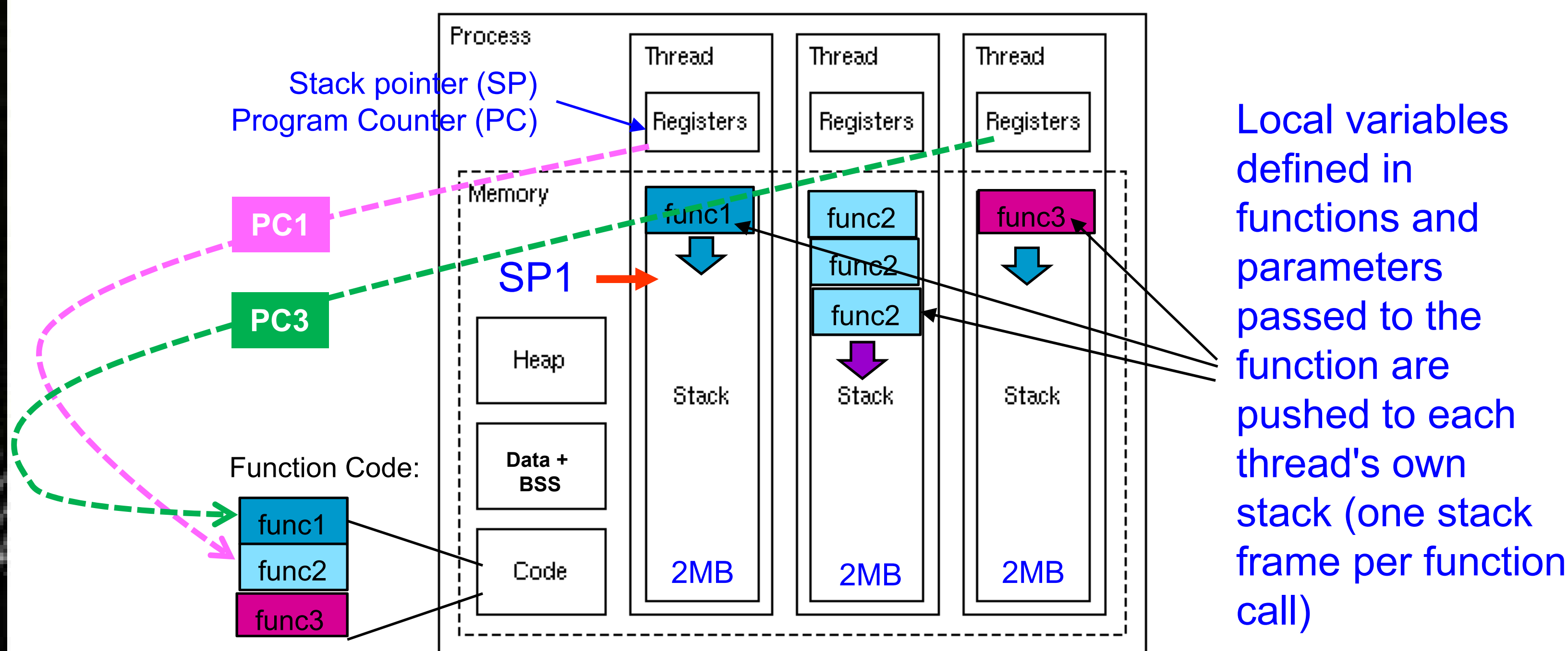
Each thread has its own stack pointer (SP)

Each thread has its own program counter (PC)



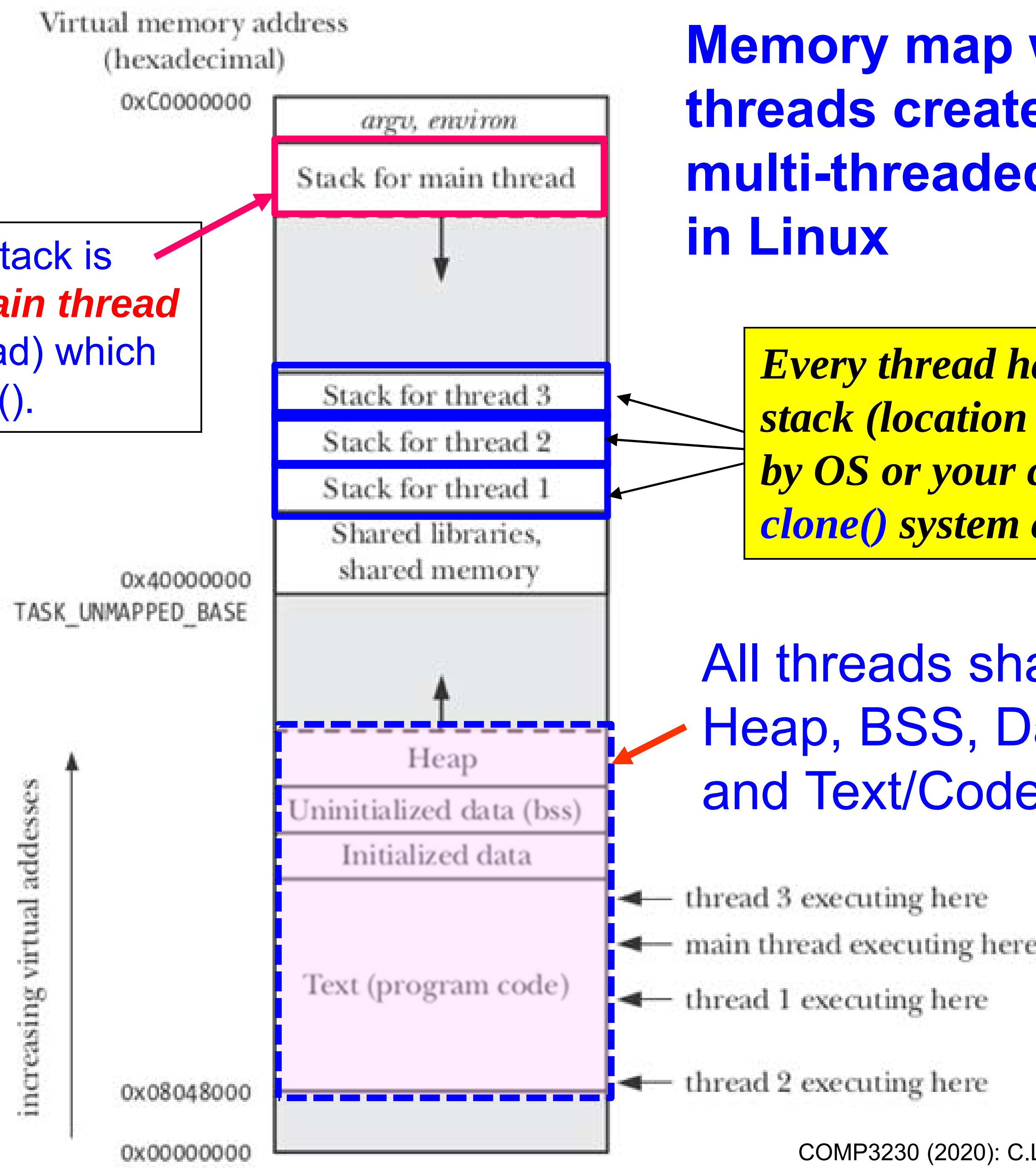
Address Space of a Multi-threaded Program (2)

- *Each thread has its own register set (e.g., PC, SP).*
- *Each thread has its own stack to keep local variables, return value etc. Each thread can call different functions with their code located at different part of code segment* → PCs are different



Memory map with threads created by a multi-threaded program in Linux

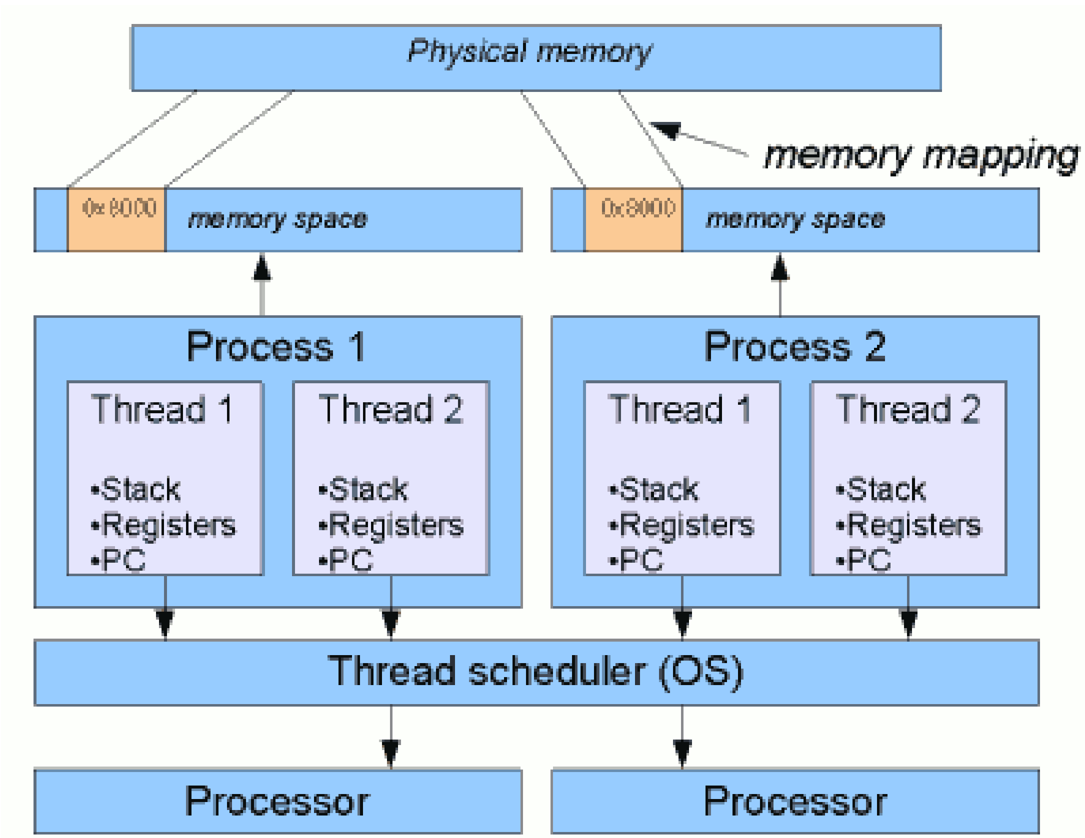
The main stack is used by ***main thread*** (initial thread) which starts main().



Every thread has it's own stack (location is determined by OS or your code via clone() system call)

All threads share Heap, BSS, Data, and Text/Code

Address Space viewed from Process & Threads

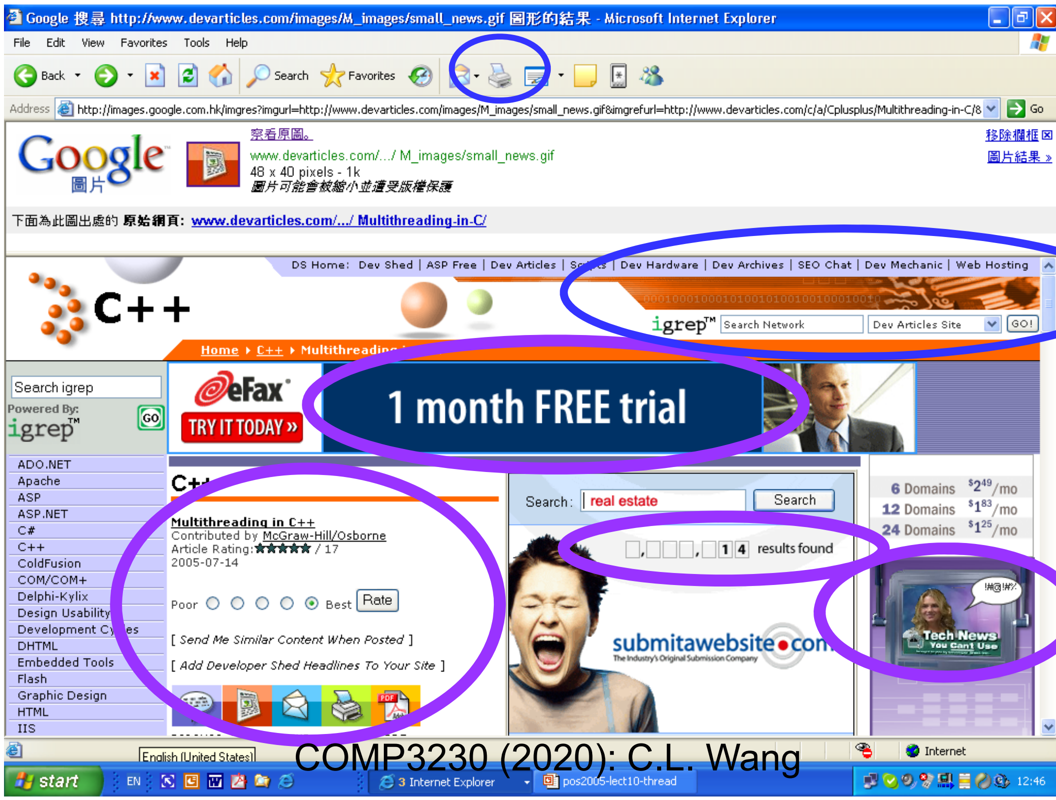


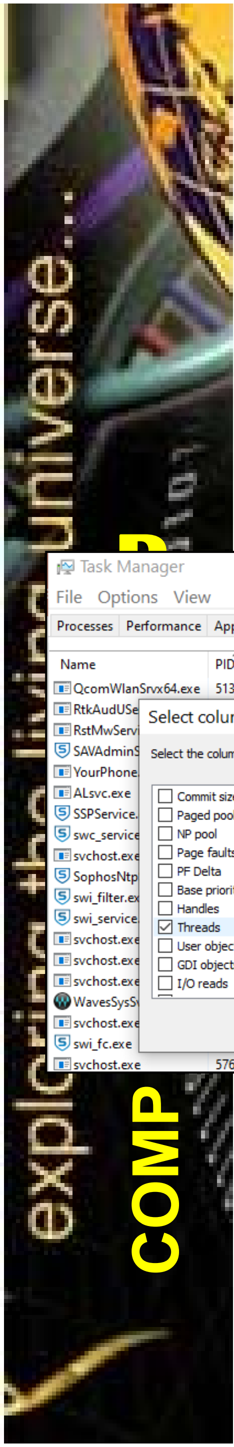
Each process has its own memory space (virtual address 0x8000 from P1 and P2 will be mapped to different physical address)

Threads in the same process shared the same memory space (virtual address 0x8000 from Thread 1 or Thread 2 of Process 1 is mapped to the same physical address)

Example: Multithreaded Web Browser

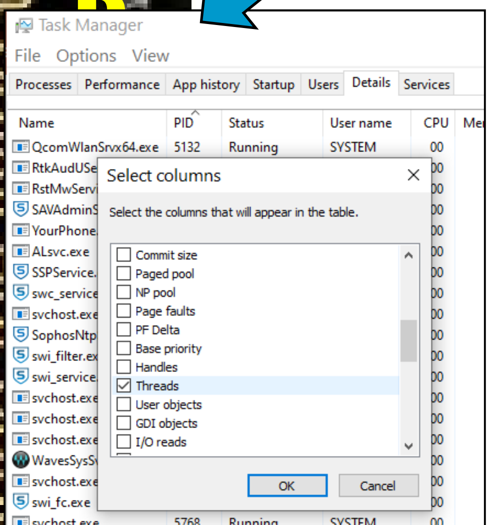
- Web browser (IE or Chrome) : you can scroll a page while it's downloading an applet or an image, play animation and sound **concurrently**, print a page in the background, while you download a new page.





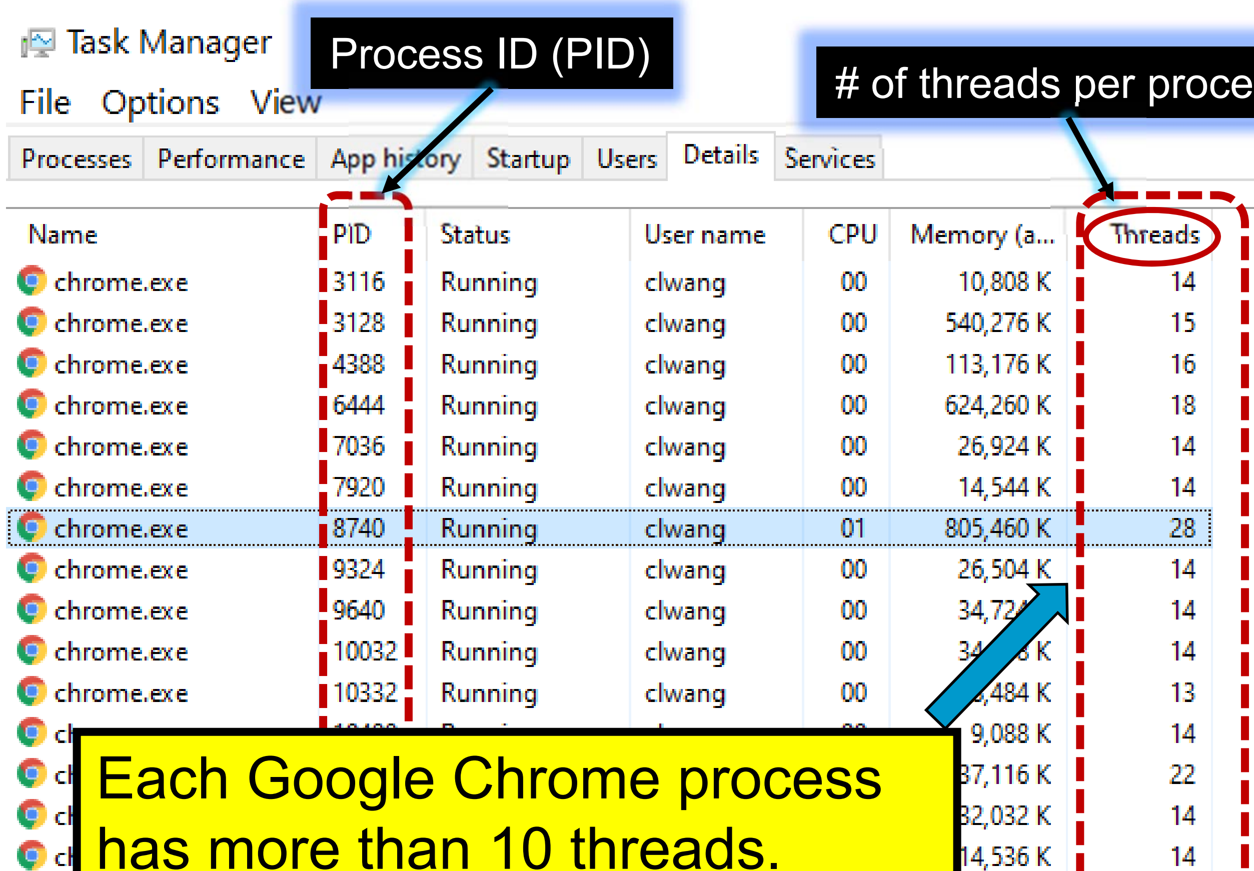
View "Threads" in Multithreaded Web Browser

Open **Task Manager** → select **View** > right click on the PID bar → **Select Columns**... and enable Threads



Process ID (PID)

# of threads per process



Name	PID	Status	User name	CPU	Memory (a...	Threads
chrome.exe	3116	Running	clwang	00	10,808 K	14
chrome.exe	3128	Running	clwang	00	540,276 K	15
chrome.exe	4388	Running	clwang	00	113,176 K	16
chrome.exe	6444	Running	clwang	00	624,260 K	18
chrome.exe	7036	Running	clwang	00	26,924 K	14
chrome.exe	7920	Running	clwang	00	14,544 K	14
chrome.exe	8740	Running	clwang	01	805,460 K	28
chrome.exe	9324	Running	clwang	00	26,504 K	14
chrome.exe	9640	Running	clwang	00	34,72[illegible]	14
chrome.exe	10032	Running	clwang	00	[illegible] K	14
chrome.exe	10332	Running	clwang	00	[illegible],484 K	13
ch[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	9,088 K	14
ch[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]7,116 K	22
ch[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]2,032 K	14
ch[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]4,536 K	14

Each Google Chrome process has more than 10 threads.

View Threads in Linux

ps -eLf will give you a list of all the threads and processes currently running on the system (“**L**” tells ps to show individual threads)

LWP: light weight process (thread) ID (alias SPID, TID).
NLWP = number of threads for the underlying process.

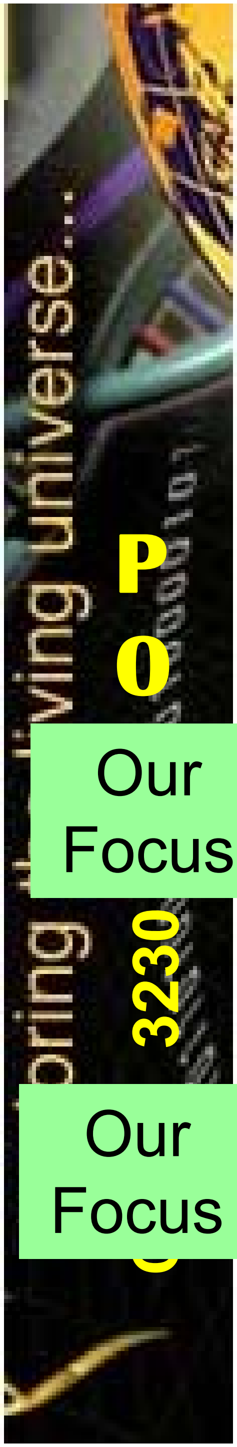
Multiple threads with the same PID (e.g., 4 threads with PID=528)

```
root@server:~
[root@server ~]# ps -eLf
UID          PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
root           1     0     1  0    1 Jun12 ?        00:00:08 init
root           2     1     2  0    1 Jun12 ?        00:00:00 [kthreadd/599]
root           3     2     3  0    1 Jun12 ?        00:00:00 [khelper/599]
root         147     1   147  0    1 Jun12 ?        00:00:00 /sbin/udevd -d
root         528     1   528  0    4 Jun12 ?        00:00:00 /sbin/rsyslogd -i /var/run/syslogd.
root         528     1   529  0    4 Jun12 ?        00:00:33 /sbin/rsyslogd -i /var/run/syslogd.
root         528     1   530  0    4 Jun12 ?        00:00:28 /sbin/rsyslogd -i /var/run/syslogd.
root         528     1   531  0    4 Jun12 ?        00:00:00 /sbin/rsyslogd -i /var/run/syslogd.
dbus         541     1   541  0    1 Jun12 ?        00:00:00 dbus-daemon --system
root         552     1   552  0    1 Jun12 ?        00:00:00 NetworkManager --pid-file=/var/run/
root         556     1   556  0    1 Jun12 ?        00:00:00 /usr/sbin/modem-manager
named        575     1   575  0   27 Jun12 ?        00:00:00 /usr/sbin/named -u named
named        575     1   576  0   27 Jun12 ?        00:00:01 /usr/sbin/named -u named
named        575     1   577  0   27 Jun12 ?        00:00:01 /usr/sbin/named -u named
```

A single-threaded process has PID = Thread ID (LWP)

Benefits of Threads

- **Fast Context Switch: (Thread 1→ Thread 2 is fast)**
  - All the threads created in a program **run in the same address space** (shared code, data, heap) → no "process context" switch.
- **Background processing**
  - A thread can execute a sequence of code that does not require user interaction and run in the foreground.
- **Speed up execution:**
  - The threads can run in parallel on several processors/cores (each has its own program counter)
- **Asynchronous processing :**
  - Threads allow overlapping I/O and computations in a simple way (even on uniprocessor machines).



Special Note: "Thread Abstraction" at Different Levels.

Not Our Focus

- ~~(1) Programming Languages level:~~
  - ~~**Threads = "programming abstractions"**~~
  - ~~Examples: **Java, C#, Ruby, Python, Objective-C, Go**~~
  - ~~JAVA multi-threading can be implemented by (1) Green Thread (user-level), (2) Native OS (kernel level)~~

Our Focus

- **(2) Library level:**
  - **Examples: POSIX Threads** (***Pthreads***).
  - **Threads = "programming abstractions"**

Our Focus

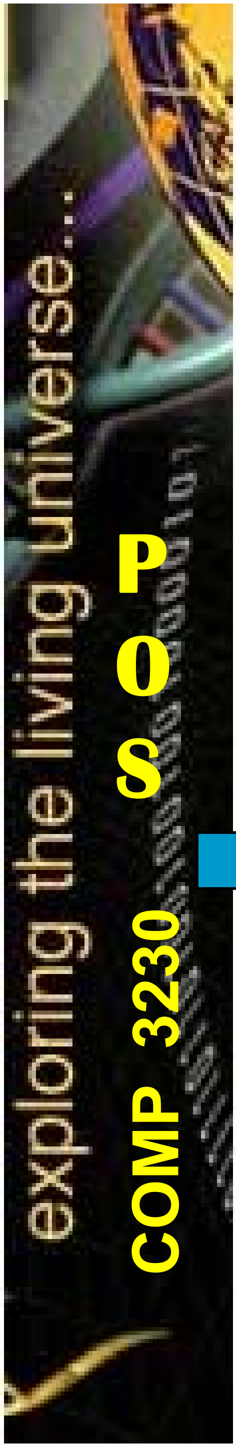
- **(3) OS level:**
  - **Threads = "scheduling entities" in OS kernel**
  - **Examples: Linux, Windows NT/2000/XP/Vista and later, Mac OS X, Solaris 9 and later.**

Thread Abstraction at Programming Language Level

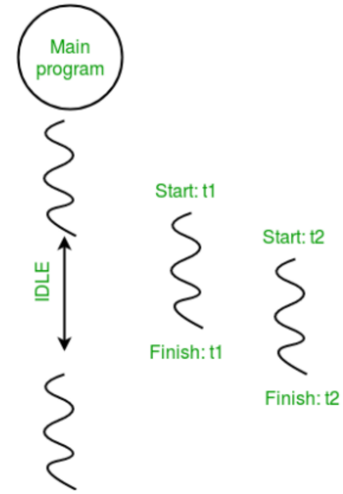
Not Our Focus

- **Java: Thread Class**
  - Thread MyThread = new Thread();
  - MyThread.start();
- **Python**
  - t1 = threading.Thread(target=*print_square*, args=(10,))
  - t1.start() // starting thread 1
  - t1.join() // wait until thread 1 is completely executed
- **GO: goroutine**
  - The **go** statement runs a function in a separate thread of execution
    - **go** list.Sort()
    - **go** f(x, y, z)

For your own interest!




```python
# importing the threading module
import threading

def print_cube(num):
    """
    function to print cube of given num
    """
    print("Cube: {}".format(num * num * num))

def print_square(num):
    """
    function to print square of given num
    """
    print("Square: {}".format(num * num))

if __name__ == "__main__":
    # creating thread
    t1 = threading.Thread(target=print_square, args=(10,))
    t2 = threading.Thread(target=print_cube, args=(10,))

    # starting thread 1
    t1.start()
    # starting thread 2
    t2.start()

    # wait until thread 1 is completely executed
    t1.join()
    # wait until thread 2 is completely executed
    t2.join()

    # both threads completely executed
    print("Done!")
```

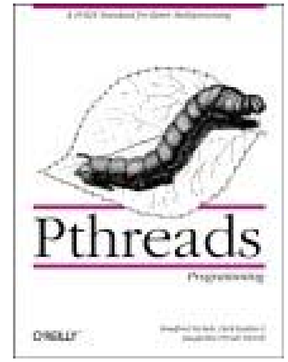


- **target**: the function to be executed by thread
- **args**: the arguments to be passed to the target function

Square: 100
Cube: 1000
Done!

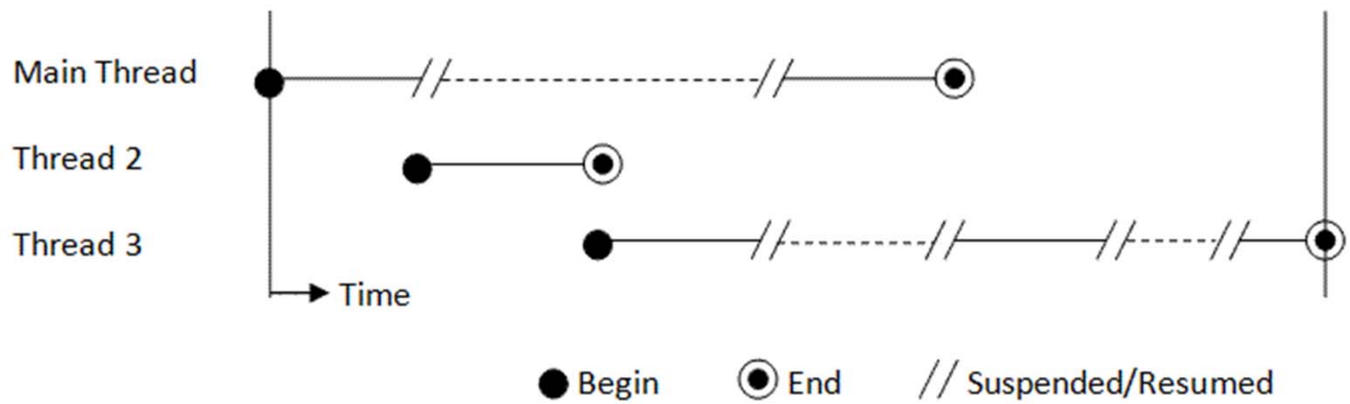
Library-level Thread Abstraction: Pthreads (POSIX threads)

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *PrintHello(void *threadid) {
   int tid;
   tid = (int) threadid;
   printf("Hello World! It's me, thread #%d!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc, t;
for(t=0; t<NUM_THREADS; t++){
   printf("In main: creating thread %d\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); }
}
pthread_exit(NULL);
}
```



Pthreads are defined as a set of C language programming types and procedure calls, implemented with a **pthread.h** header file and a thread library.

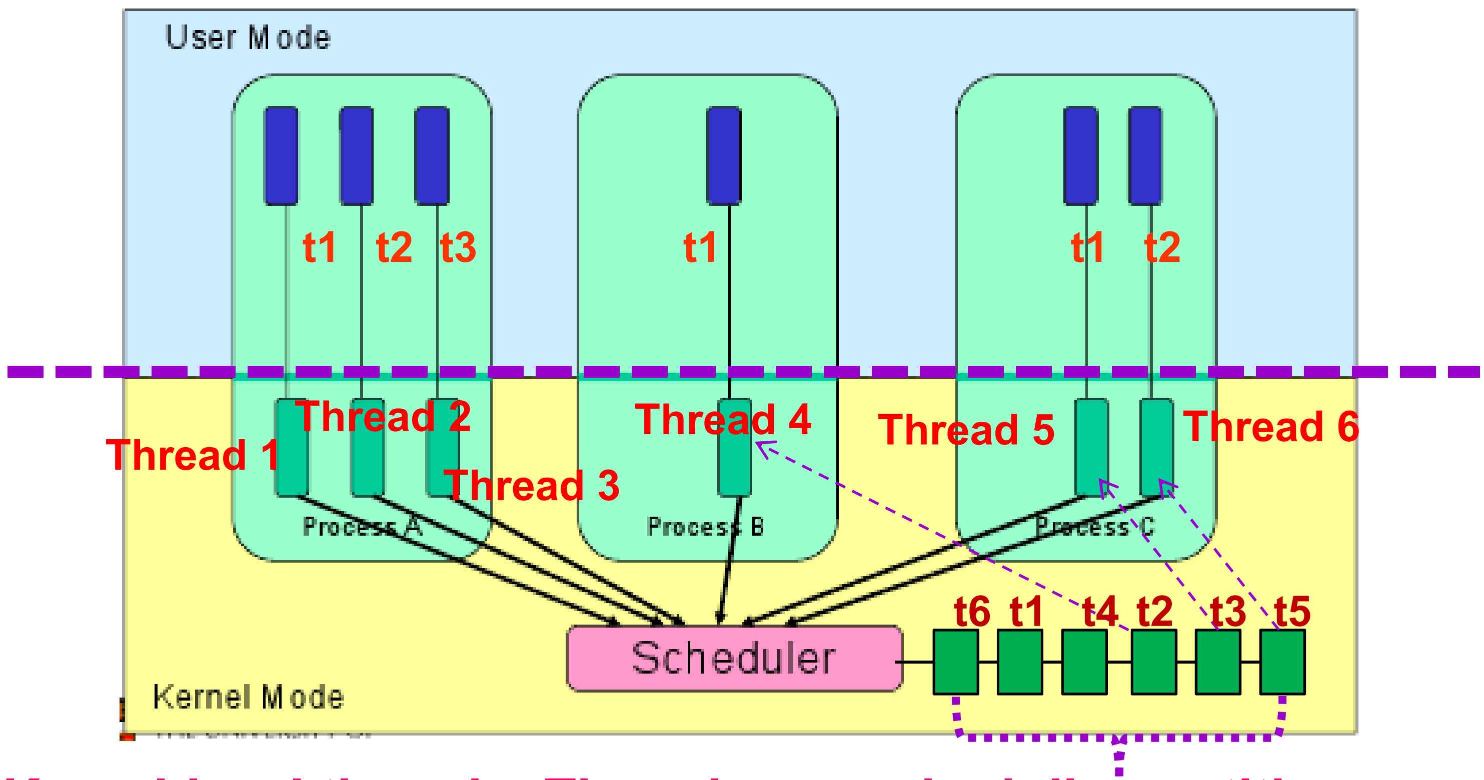
(More to be discussed)



Kernel-level threads (Linux Threads)

Kernel-level threads

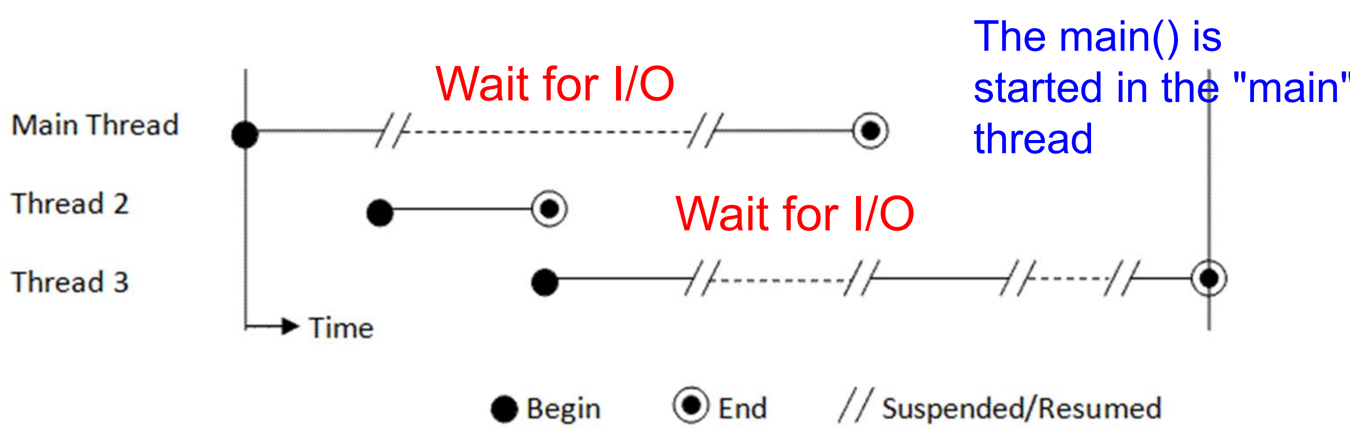
- One-to-one thread mapping: Every user-level thread has its own kernel thread.
- Kernel directly schedules the threads.



Kernel-level threads: Threads are scheduling entities

Kernel-level threads

- **Examples: Linux, Windows, Mac OS X.**
  - must call system call to create **thread,** e.g., **clone() in Linux**
- **Advantages**:
  - ***Increased scalability and throughput***: each kernel thread is a kernel-schedulable entity. **Multiple threads can run concurrently on different cores**.
  - ***Increased interactivity***: When one thread blocks (doing I/O), the other threads created by the same process can continue to execute.



Threads in Linux

- In Linux, **threads**, also called **Lightweight Processes (LWP)**
- Each thread has its own **thread ID (TID)**
  - appears as **LWP** (in ps –eLf) or **SPID** (in ps –T)
  - **gettid()** (or syscall(SYS_gettid)) returns the caller's thread ID (TID).
    - **In a single-threaded process: TID = PID**
    - **In a multithreaded process**: **all threads have the same PID**, but each one has a unique **TID**.

To the Linux kernel's scheduler, ***threads are nothing but the standard processes*** which happen to share certain resources (e.g., code/data/heap, etc.).

Threads in Linux: ps -eLf

ps -eLf shows all the threads and processes currently running on the system.

LWP: **light weight process (thread) ID (alias SPID, TID).**
NLWP: **number of threads of the underlying process.**

All threads of the same process will share same PID: ***syslogd, named***

```
root@server:~
[root@server ~]# ps -eLf
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
root         1     0     1  0    1 Jun12 ?        00:00:08 init
root         2     1     2  0    1 Jun12 ?        00:00:00 [kthreadd/599]
root         3     2     3  0    1 Jun12 ?        00:00:00 [khelper/599]
root       147     1   147  0    1 Jun12 ?        00:00:00 /sbin/udevd -d
root       528     1   528  0    4 Jun12 ?        00:00:00 /sbin/rsyslogd -i /var/run/syslogd.
root       528     1   529  0    4 Jun12 ?        00:00:33 /sbin/rsyslogd -i /var/run/syslogd.
root       528     1   530  0    4 Jun12 ?        00:00:28 /sbin/rsyslogd -i /var/run/syslogd.
root       528     1   531  0    4 Jun12 ?        00:00:00 /sbin/rsyslogd -i /var/run/syslogd.
dbus       541     1   541  0    1 Jun12 ?        00:00:00 dbus-daemon --system
root       552     1   552  0    1 Jun12 ?        00:00:00 NetworkManager --pid-file=/var/run/
root       556     1   556  0    1 Jun12 ?        00:00:00 /usr/sbin/modem-manager
named      575     1   575  0   27 Jun12 ?        00:00:00 /usr/sbin/named -u named
named      575     1   576  0   27 Jun12 ?        00:00:01 /usr/sbin/named -u named
named      575     1   577  0   27 Jun12 ?        00:00:01 /usr/sbin/named -u named
```

A single-threaded process has PID = LWP

"ps –T –p <pid>" (-T: lists all threads)

- **"SPID" column represents thread IDs**
- **"CMD" column shows thread names.**
- The following command list all threads created by a process with <pid> = **55499**.

```
[[redacted] ~]$ ps -T -p 55499
  PID  SPID TTY          TIME CMD
55499 55499 pts/1    00:00:00 Suricata-Main
55499 55500 pts/1    00:00:00 RxPcapeth51
55499 55501 pts/1    00:00:02 FlowManagerThre
55499 55502 pts/1    00:00:00 SCPerfWakeupThr
55499 55503 pts/1    00:00:00 SCPerfMgmtThrea
```

Main thread: SPID = PID (TID)

```
# ps -T -p 3692
  PID  SPID TTY          TIME CMD
 3692  3692 ?        00:00:27 mysqld
 3692  3707 ?        00:02:26 mysqld
 3692  3708 ?        00:02:30 mysqld
 3692  3709 ?        00:02:31 mysqld
 3692  3710 ?        00:02:36 mysqld
 3692  3711 ?        00:02:36 mysqld
 3692  3712 ?        00:02:31 mysqld
 3692  3713 ?        00:02:31 mysqld
 3692  3714 ?        00:02:33 mysqld
```

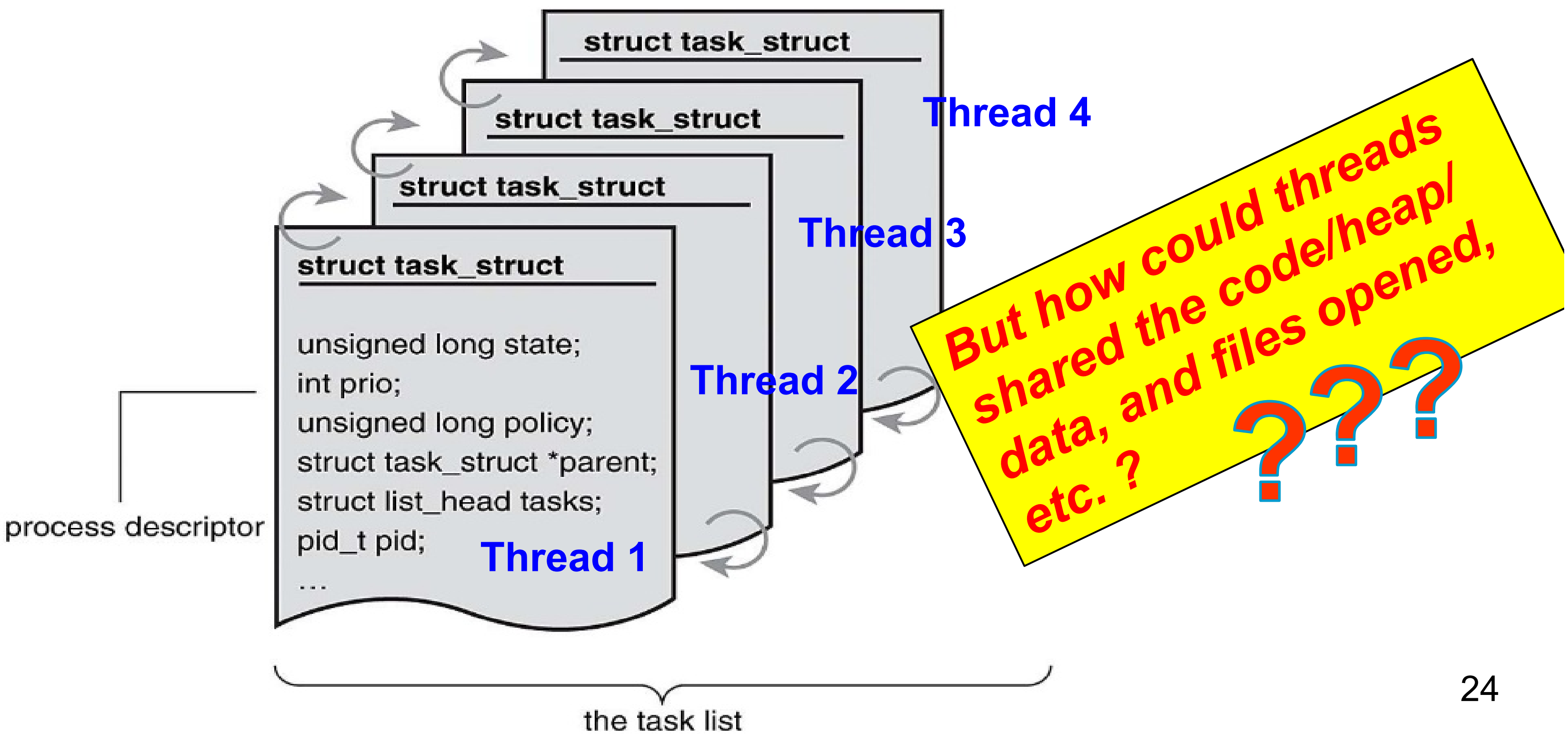
The command displays the threads of the *mysqld* process (MySQL server daemon).

Thread Implementation in Linux

- Processes are created with **fork()**.
- Threads are created with **clone()**.
- **Implementation Trick:**
  - **Linux implements threads as "processes".**
  - Linux Kernel does ***not*** distinguish between a **process** and a **thread**, but call them all **"task".**
  - *Each thread is represented with the same data structure "**task_struct"** **(same as process)** and the scheduling of these is **the same as that of processes.***

Thread Implementation in Linux

- "Linux implements all *threads* as standard *processes*." → Each thread has a unique *task_struct* and appears to the kernel as a normal process



Recall: task_struct

```
struct task_struct {
/* these are hardcoded - don't touch */
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  long counter;
  long priority;
  struct task_struct *next_task, *prev_task;
  struct task_struct *next_run,  *prev_run;
  pid_t pid;
  pid_t tgid;
/* pointers to (original) parent process, youngest child, younger sibling,
* older sibling, respectively.  */
  struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
  long utime, stime, cutime, cstime, start_time;
/* open file information */
  struct files_struct *files;
/* memory management info */
  struct mm_struct *mm;

.....
struct thread_struct thread;
/* store CPU-specific state of this task, e.g.,
  program counter (PC), SP,  … */
}
```

Ordinary single-threaded process: PID displayed in ps = tgid in task_struct

= process ID (= TID shown in ps/top)

= thread group ID (= PID shown in ps/top)

No TID in task_struct?

mm_struct: a process' address space

Keep hardware state

```
struct mm_struct {
  int count;
  pgd_t * pgd; (point to the process's 1st-level page table)
  unsigned long context;
  unsigned long start_code, end_code, start_data, end_data;
  unsigned long start_brk, brk, start_stack, start_mmap;
  unsigned long arg_start, arg_end, env_start, env_end;
  unsigned long rss, total_vm, locked_vm;
  unsigned long def_flags;
  struct vm_area_struct * mmap;
  struct vm_area_struct * mmap_avl;
  struct semaphore mmap_sem;
};
```

For Your Own Interest

Various IDs (User View)

- ***PID***: process identifier
  - = thread group identifier (tgid) in task_struct
  - getpid() return task_tgid_vnr(current); /* returns current->tgid */
- ***TID***: thread identifier
  - = *pid* in task_struct; unique system-wide
  - **gettid()** return task_**pid**_vnr(current);

- ***TGID***: thread group identifier
  - = *pid* of the main thread that started the whole process.
- ***pthread_id***: pthread identifier (pthread_t type)
  - assigned and maintained by the Pthreads implementation

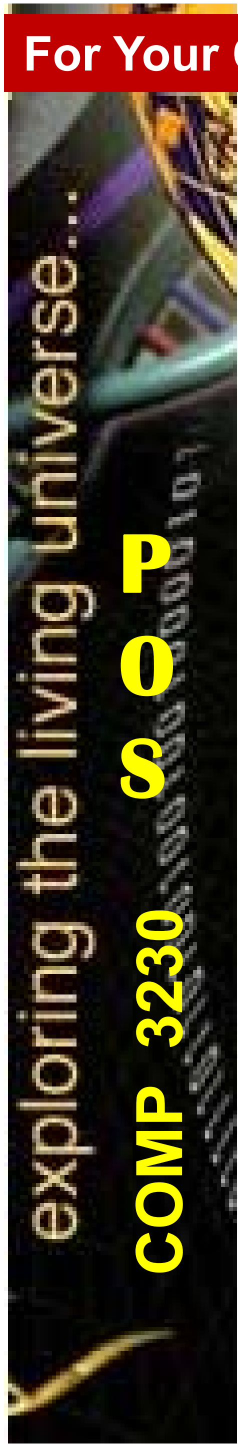
The *pid* in the task_struct, on the surface, corresponds to the process ID, but in fact, it corresponds to the **thread ID! The PID you see in** ps/top **is actually tgid in task_struct**

getpid() (sys_getpid)

- Return the *thread group id* of the current process.
- Despite the name, this returns the **tgid** not the **pid**.
- The tgid and the pid are identical unless **CLONE_THREAD** was specified on **clone()** in which case the **tgid** is the same in all threads of the same group.

```
SYSCALL_DEFINE0(getpid)
{
        return task_tgid_vnr(current);
}

/* Thread ID - the internal kernel "pid" */
SYSCALL_DEFINE0(gettid)
{
        return task_pid_vnr(current);
}
```

https://elixir.bootlin.com/linux/latest/source/kernel/sys.c



For Your Own Interest

thread_struct

Used to store CPU-specific state of this task (needed during the thread switching)

x86

```
struct thread_struct {
    /* Cached TLS descriptors: */
    struct desc_struct    tls_array[GI
    unsigned long         sp0;
    unsigned long         sp;
#ifdef CONFIG_X86_32
    unsigned long         sysenter_cs;
#else
    unsigned long         usersp;    /*
    unsigned short         es;
    unsigned short         ds;
    unsigned short         fsindex;
    unsigned short         gsindex;
#endif
#ifdef CONFIG_X86_32
    unsigned long         ip;
#endif
```

Instruction pointer (= program counter)

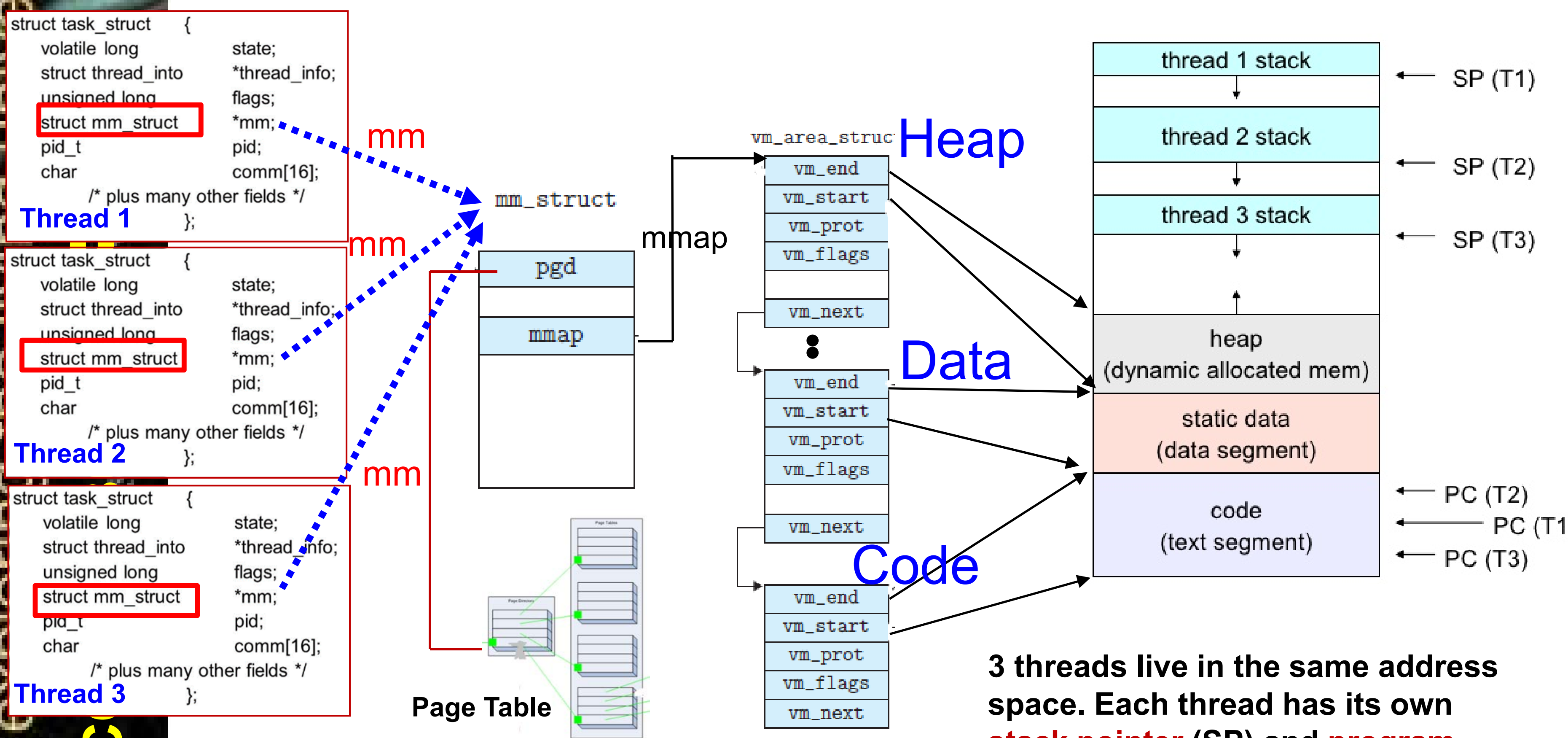
arm64

```
struct cpu_context {
    unsigned long x19;
    unsigned long x20;
    unsigned long x21;
    unsigned long x22;
    unsigned long x23;
    unsigned long x24;
    unsigned long x25;
    unsigned long x26;
    unsigned long x27;
    unsigned long x28;
    unsigned long fp;
    unsigned long sp;
    unsigned long pc;
};
```

```
struct thread_struct {
    struct cpu_context     cpu_context;    /* cpu context */
    unsigned long          tp_value;
    struct fpsimd_state     fpsimd_state;
    unsigned long          fault_address;    /* fault info */
    struct debug_info      debug;         /* debugging */
};
```

How threads in the same process share heap, data, and code?

- **Simple!** All threads share the same **mm_struct**.



All threads shared the same page table

3 threads live in the same address space. Each thread has its own stack pointer (SP) and program counter (PC) saved in thread_struct

Thread Creation in Linux

- **Use clone() to create a new thread:**
  - **clone**(… CLONE_**VM** | CLONE_**FS** | CLONE_**FILES** | CLONE_**SIGHAND, 0**)

> **CLONE_VM** – **virtual memory space shared** (use the same struct **mm_struct**)
> **CLONE_FS** – **file system info shared**
> **CLONE_FILES** – **all open files are shared** (use the same **files_struct**)
> **CLONE_SIGHAND** – **all signal handlers shared** (use the same **sighand_struct**)

- **fork** actually calls **clone** to create a child process: **fork() = "clone(SIGCHLD, 0);"**
  - The SIGCHLD flag tells the kernel to send the SIGCHLD to the parent when the child terminates.
- **Note: clone() is used to implement the pthread_create() – Read Pthreads part**

clone() system call

int clone(int flags);

- **CLONE_VM**
  - **If set**, **the calling process and the <u>child process</u> run in the same address space** → memory writes visible in the other process. (same effect as the shared memory)
- **CLONE_FS**
  - **If set**, the caller and the **<u>child process</u>** share the same file system information (e.g., root directory, current working directory, etc.).
- **CLONE_FILES**
  - **If set,** any file descriptor created by the calling process or by the child process is also valid in the other process.
- **CLONE_SIGHAND**
  - **If set,** the calling process and the **child process** share the same table of signal handlers.

See we still use the terms "**calling process**" & "**child process",** but not "threads", since <u>Linux treats process and threads the same internally</u>.

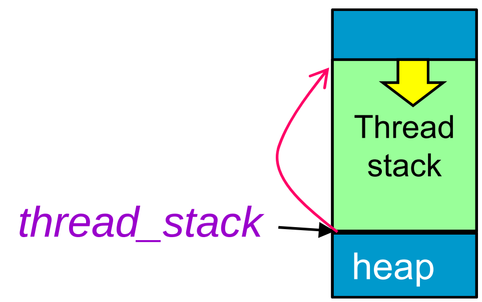
clone() Flags

- **CLONE_VM: Parent and child share address space.**
- **CLONE_SIGHAND: Parent and child share signal handlers and blocked signals.**
- **CLONE_FILES: Parent and child share open files.**
- **CLONE_FS: Parent and child share filesystem information.**
- **CLONE_PARENT**: Child is to have same parent as its parent.
- CLONE_IO: the new process shares an I/O context with the calling process
- CLONE_NEWNS: Create a new namespace for the child.
- CLONE_SYSVSEM: Parent and child share System V SEM_UNDO semantics.
- CLONE_VFORK: vfork() was used, parent will sleep until the child wakes it.
- CLONE_STOP: Start process in the TASK_STOPPED state.
- CLONE_SETTLS: Create a new TLS (thread-local storage) for the child.
- CLONE_CHILD_SETTID: Set the TID in the child.
- CLONE_PARENT_SETTID: Set the TID in the parent.
- **CLONE_THREAD**: the child is placed in the same thread group as the calling process.
- CLONE_INTO_CGROUP (since Linux 5.7)
- …

For Your Own Interest

Creating a thread using Clone()

- **Provide more precise control than fork()**
- **int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);**
  - **The child_stack argument specifies the location of the stack used by the child process (i.e., your new thread).**
  - ***fn* : function to be started!**
- **Example: creating a thread**
  - **void *thread_stack = malloc(STACK_SIZE);**
  - **thread_pid = clone(&func, thread_stack+STACK_SIZE, CLONE_SIGHAND|CLONE_FS|CLONE_VM|CLONE_FILES, NULL);**

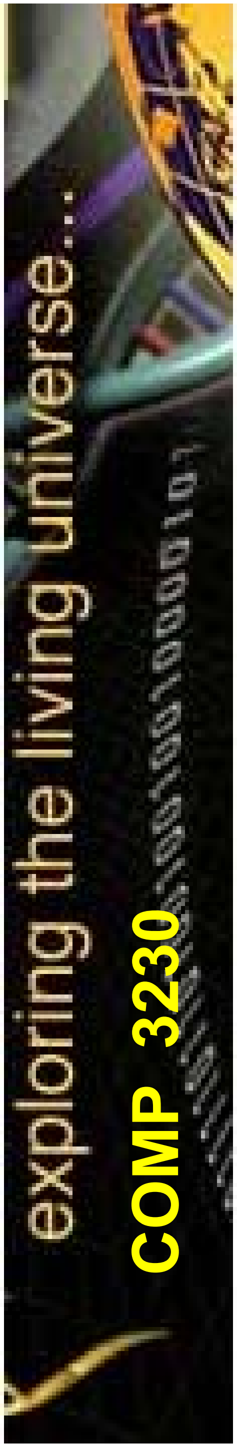


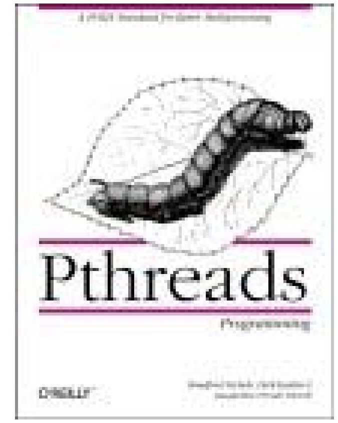
Sample code: https://www.opensourceforu.com/2011/08/light-weight-processes-dissecting-linux-threads/

Fork() vs Clone()

- **task_struct** has pointers to other structs such as **mm_struct** (address space), **files_struct** (open files), **sighand_struct** (signal handlers), ….
- **Wht thread creation is faster than process creation?**
  - When you **fork** a new **"process"** (**clone(SIGCHLD, 0))",** all of these structs will be duplicated (*duplication takes time*) + **page table is duplicated** (*this takes time, too*).
  - When you **clone** a new **"thread",** these structs will be shared between the new and old task_structs (both point to the same **mm_struct**, the same **files_struct**, ...), **page table is shared.**

CLONE_SIGHAND|CLONE_FS|CLONE_VM|CLONE_FILES





POSIX Pthreads

POSIX = "**P**ortable **O**perating **S**ystem **I**nterface [for Uni**X**]" is the name of a family of related standards specified by the IEEE to define the application programming interface (API)"

(More to be discussed in Tutorial 3 and Lecture 6 Part II)

Pthreads Implementations

For your own interest!

- **Linux:**
  - **LinuxThreads (1996, obsolete): clone()**
  - **Native POSIX Thread Library (NPTL): 2002**
    - **pthread_create()** calls **clone()** to create new threads → kernel-level thread (one-to-one thread mapping).
  - Others: PCthreads, Nthreads, Clthreads, …
- **Windows**
  - **pthreads-w32: uses Win32 threads API**
  - **MinGW-w64 Winpthreads (x86-64)**
  - **Pthread.dll library**

Pthreads Win32

```
/* 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 */
```

Multi-threading Programming Languages with Pthreads

- **Java JVM** implementation using Pthreads
  - **HotSpot JVM** (i.e. Oracle JDK and OpenJDK) was implemented on top of Pthreads.
- The **Python** interpreter maps **Python thread** requests to Pthreads.
- **PHP** with Pthreads for Web development
- **go-pthreads**: Lightweight binding of Pthreads to **Google Go**
- Intel **OpenMP** implementation for Linux is based on Pthreads

For your own interest!

Pthreads functions

- **Must add “#include <pthread.h>”**
- **Compile and link with -pthread.**
- **Some functions to be discussed:**
  - pthread_attr_init()
  - pthread_attr_(get/set)stack
  - **pthread_create()**
  - **pthread_join()**
  - pthread_self()
  - pthread_mutex_lock/unlock
  - (More in Tutorial 3)

pthread_attr_init()

- int **pthread_attr_init**(pthread_attr_t ***attr**);
- The function is used to initialize object attributes to their default values.
- The default values for attributes (attr) are:
  - **scope:** The *contention scope* of a user thread defines how it is mapped to a kernel thread. Default **PTHREAD_SCOPE_SYSTEM** in Linux
  - **stackaddr** = **NULL** → New thread has system-allocated stack address.
  - **stacksize**: default minimum PTHREAD_STACK_SIZE set in pthread.h
- **…(Don't worry, we use Default attribute!)**

pthread_attr_init(): scope

- **PTHREAD_SCOPE_SYSTEM**
  - **1:1 thread model:** the thread is **directly mapped to one kernel thread and** will be scheduled against **all other threads in the system.**
    - Example: if there is one process P1 with 10 threads with scope **PTHREAD_SCOPE_SYSTEM** and a single threaded process P2, P2 will get one timeslice out of 11 and every thread in P1 will get one timeslice out of 11. I.e. P1 will get 10 time more timeslices than P2.
- Linux only supports **PTHREAD_SCOPE_SYSTEM**.
- **Pthreads-w32** only supports PTHREAD_SCOPE_SYSTEM

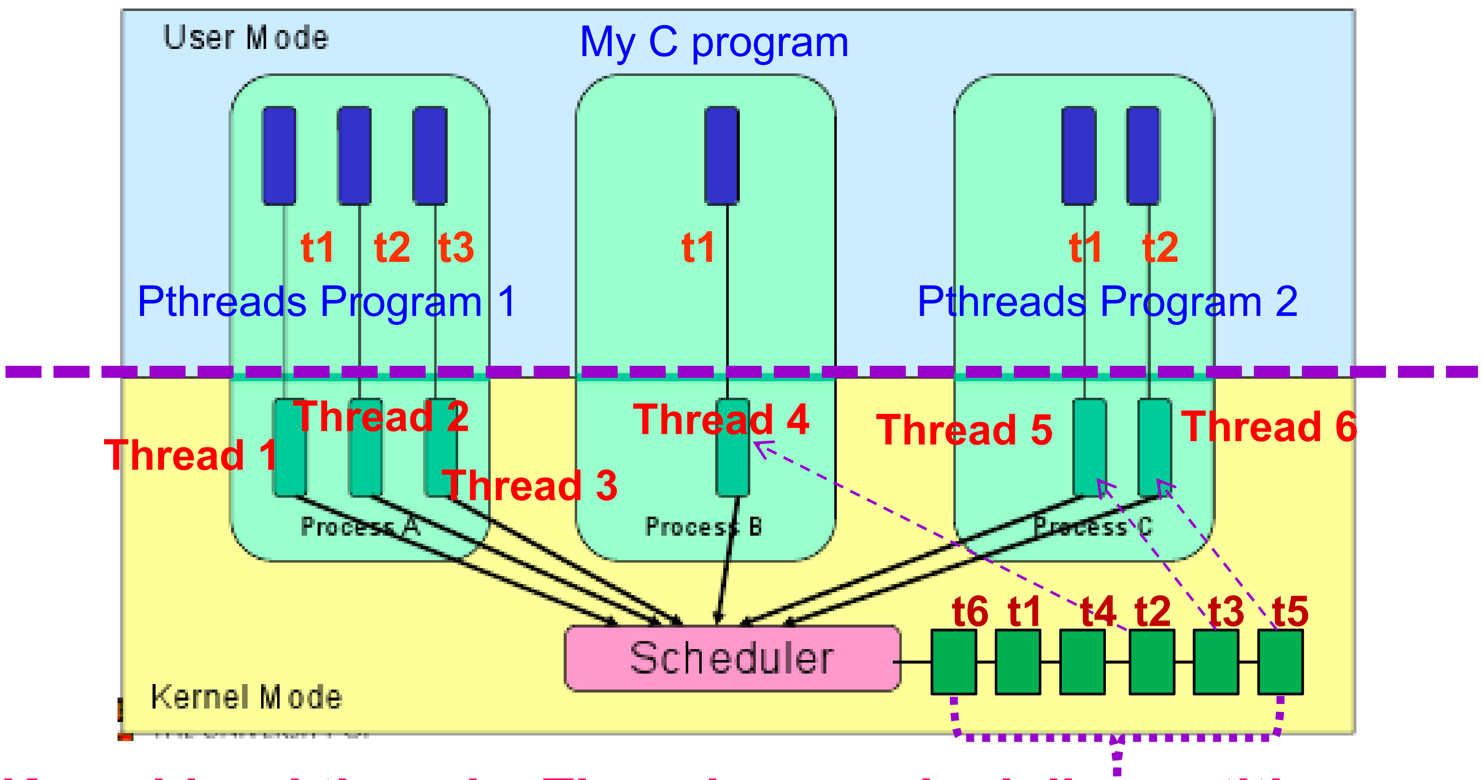
Try $man pthread_attr_setscope()

```
POSIX.1 requires that an implementation support at least one  of  these  contention  scopes.   Linux  supports
PTHREAD_SCOPE_SYSTEM, but not PTHREAD_SCOPE_PROCESS.
```

pthread_attr_init(): scope

PTHREAD_SCOPE_SYSTEM

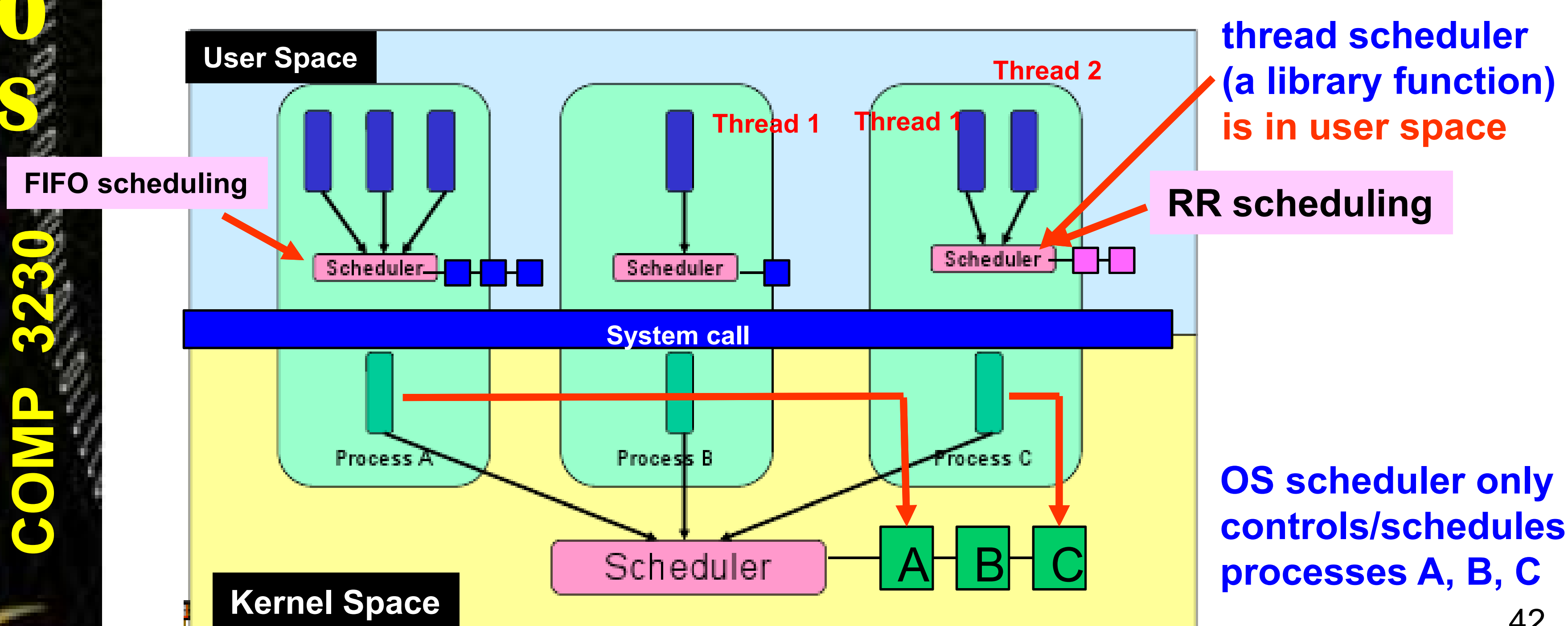
- One-to-one thread mapping: Every user-level thread has its own kernel thread.
- Kernel directly schedules the threads.



Kernel-level threads: Threads are scheduling entities

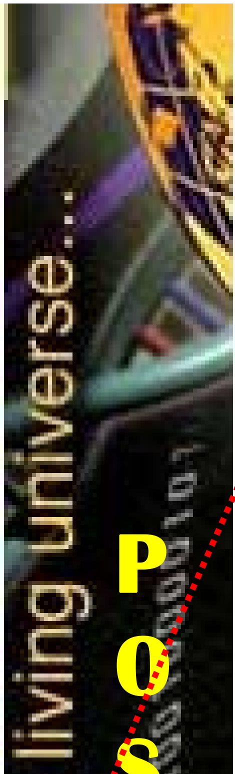
pthread_attr_init(): scope

- **PTHREAD_SCOPE_PROCESS**
  - The thread competes for resources (e.g., CPU) with all other threads created by the same process.
  - **M:1 thread model.** Threads are scheduled relative to other threads in the process according to their scheduling policy and priority (by user-space scheduler)



pthread_attr_(get/set)stack

- **int pthread_attr_setstack**(pthread_attr_t ***attr**, void ***stackaddr**, size_t **stacksize**);
  - set the user-defined **base address** (***stackaddr***) of the thread stack and **stack size**
  - Also pthread_attr_**set**stacksize(pthread_attr_t *attr, size_t **size**): set the stack size. If size is zero, a default size is used.
- **int pthread_attr_getstack**(const pthread_attr_t ***attr**, void ****stackaddr**, size_t ***stacksize**);
  - returns the **base address (lowest addressable byte) of the thread stack** and **stack size**.
  - Also **pthread_attr_getstacksize**(&**attr**, &**stacksize**);



Sample code from Tutorial 3: show stack sizes

```c
void show_stack(pthread_attr_t *attr, pthread_t thread, char *prefix) {
    size_t stack_size, guard_size;
    void *stack_addr;
    int rc;

    rc = pthread_attr_getguardsize(attr, &guard_size);
    assert(rc == 0);

    rc = pthread_attr_getstack(attr, &stack_addr, &stack_size);
    assert(rc == 0);

    printf("Thread %s (id=%lu) stack:\n", prefix, thread);
    printf("\tstart address\t= %p\n", stack_addr);
    printf("\tend address\t= %p\n", stack_addr + stack_size);
    printf("\tstack size\t= %.2f MB\n", stack_size/1024.0/1024.0);
    printf("\tguard size\t= %lu B\n", guard_size);
}
```

Get the stack size and stack address

Print the size and stack address

```c
void *entry_point(void *arg) {
    pthread_t thread = pthread_self();

    int rc;
    pthread_attr_t attr;
    rc = pthread_getattr_np(thread, &attr);
    assert(rc == 0);

    pthread_mutex_lock(&lock);
    show_stack(&attr, thread, (char *)arg);
    pthread_mutex_unlock(&lock);

    return NULL;
}
```

Pthread ID

```c
int main(int argc, char* argv[]) {
    pthread_t p1, p2;
    int rc;

    rc = pthread_create(&p1, NULL, entry_point, "1");
    assert(rc == 0);
    rc = pthread_create(&p2, NULL, entry_point, "2");
    assert(rc == 0);

    entry_point("main");

    rc = pthread_join(p1, NULL);
    assert(rc == 0);
    rc = pthread_join(p2, NULL);
    assert(rc == 0);

    return 0;
}
```

Main thread creates two threads, each will call “entry_point”

Output: Each thread has its own stack, 8MB each

```
yczhong@workbench:~/repos/pthread-tutorial$ ./show_stack
Thread main (id=140328868591424) stack:
        start address    = 0x7fff0d7ca000
        end address      = 0x7fff0dfc9000
        stack size       = 8.00 MB
        guard size       = 0 Bytes
Thread 2 (id=140328851629824) stack:
        start address    = 0x7fa0dada7000
        end address      = 0x7fa0db5a7000
        stack size       = 8.00 MB
        guard size       = 4096 Bytes
Thread 1 (id=140328860022528) stack:
        start address    = 0x7fa0db5a8000
        end address      = 0x7fa0dbda8000
        stack size       = 8.00 MB
        guard size       = 4096 Bytes
```

stack size = 8MB

stack size = 8MB

stack size = 8MB

Set Thread Stack Size pthread_attr_setstack()

pthread_attr_setstack(&attr, **sp**, **stack_size**);

Running the program on **Linux/x86-32** with the NPTL threading implementation

Default stack size is 2MB

```
$ ulimit -s       # No stack limit ==> default stack size is 2 MB
unlimited
$ ./a.out
Thread attributes:
        Detach state        = PTHREAD_CREATE_JOINABLE
        Scope               = PTHREAD_SCOPE_SYSTEM
        Inherit scheduler   = PTHREAD_INHERIT_SCHED
        Scheduling policy   = SCHED_OTHER
        Scheduling priority = 0
        Guard size          = 4096 bytes
        Stack address       = 0x40196000
        Stack size          = 0x201000 bytes
```

```
s = pthread_attr_setstack(&attr, sp, stack_size);
if (s != 0)
    handle_error_en(s, "pthread_attr_setstack");
```

New stack size set as 0x3000000

```
s = pthread_attr_getstack(attr, &stkaddr, &v);
if (s != 0)
    handle_error_en(s, "pthread_attr_getstack");
printf("%sStack address       = %p\n", prefix, stkaddr);
printf("%sStack size          = 0x%zx bytes\n", prefix, v);
```

```
$ ./a.out 0x3000000
posix_memalign() allocated at 0x40197000
Thread attributes:
        Detach state        = PTHREAD_CREATE_DETACHED
        Scope               = PTHREAD_SCOPE_SYSTEM
        Inherit scheduler   = PTHREAD_EXPLICIT_SCHED
        Scheduling policy   = SCHED_OTHER
        Scheduling priority = 0
        Guard size          = 0 bytes
        Stack address       = 0x40197000
        Stack size          = 0x3000000 bytes
```

You can read the complete source code by **$man pthread_attr_init (@workbench)**

Example: Set Thread Stack Size pthread_attr_setstacksize()

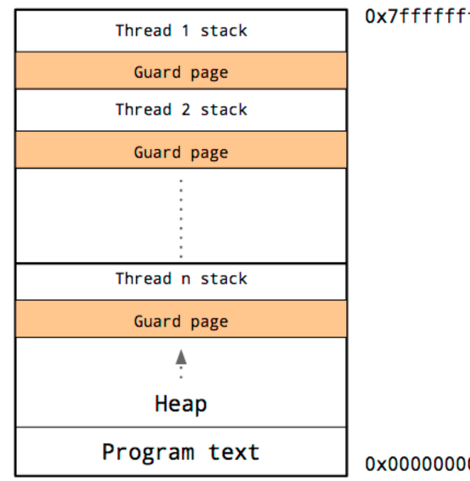
The **stacksize** attribute defines the minimum stack size (in bytes) allocated for the created threads stack.

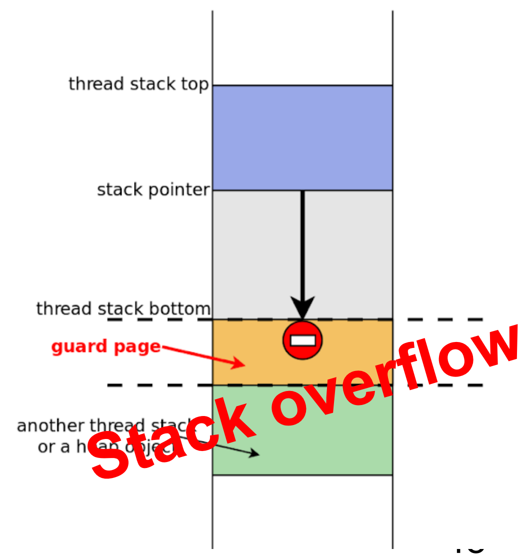
```
#include <pthread.h>
#include <limits.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;

size_t size = PTHREAD_STACK_MIN + 0x4000;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);

/* only size specified in tattr*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

Thread Stack Overflows

Guard Page

- A **guard page** is a locked-down page put at the end of the stack.
- When the application tries to access it, the guard page will trigger a segmentation fault (**SIGSEGV** signal).
- **Guard page** helps protect against **stack overflows** (which may trash other thread's stack or main thread's heap).





More about the thread stack

- **The ulimit command shows the stack size for the main thread (whole process)**, **not the thread stack size**.
- **The main stack** (used by main thread) is created by kernel, allocated at the higher memory addresses and theoretically it can even grow up to the end of the heap (default is 8MB in Linux)
- **The thread stack** is allocated by **application itself**, and could be allocated in the **heap**, though typically created in between heap and main stack.
  - Use the **pthread_attr_setstacksize()** to set the size explicitly in your application.
  - The default stack size for a new thread created by pthread_create() is **2MB in Linux/x86-32** and **8 MB in Linux x86-64.**

pthread_create()

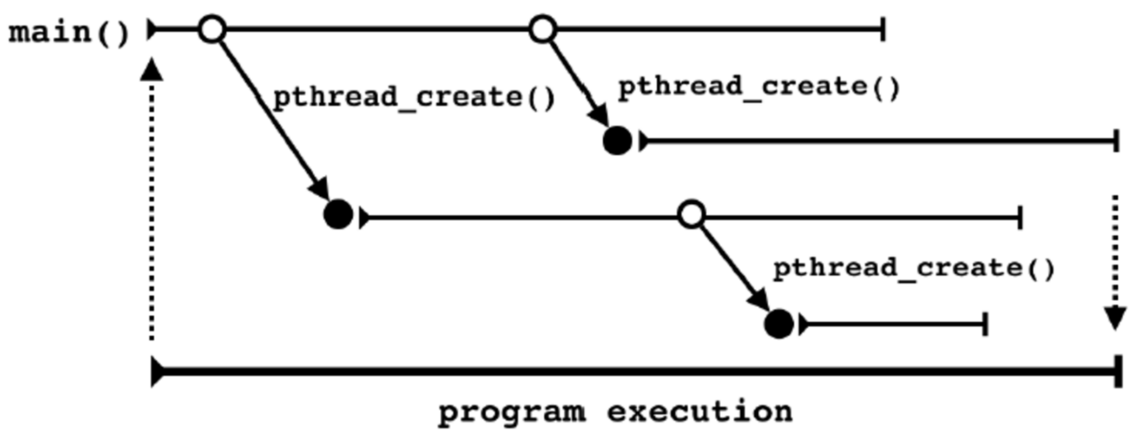
#include <pthread.h>

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

- ***&thread***: stores the *ID of the new thread* in the buffer pointed to by thread; which is used to refer to the thread in subsequent calls
- ****attr*** : thread attribute (e.g., **stack size**)
  - **NULL**: use default attributes. If attributes are modified later, the thread's attributes are not affected.
- ****start_routine*** : a pointer to the function to be threaded.
- ****arg*** : pointer to the argument of ***start_routine***
  - running ***start_routine*** with ***arg*** as the only argument.

pthread_create()

- The C program starts in **main()** that runs in its own thread, the "main" thread.
- New threads are dynamically created using **pthread_create().**
- A thread **ends execution** when its starting procedure **returns** OR it calls **pthread_exit().**
- pthread_create() returns **zero on success**. On error, it returns an **error number**



Example: pthread_create()

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *PrintHello(void *threadid) {
   int tid;
   tid = (int) threadid;
   printf("Hello World! It's me, thread #%d!\n", tid);
   pthread_exit(NULL); /* terminate the thread */
}

int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc, t;
  for(t=0; t<NUM_THREADS; t++){
     printf("In main: creating thread %d\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){  /* if rc >0 → fail to create thread */
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1); }
  }
pthread_exit(NULL);
}
```

Compile the code :
gcc thread.c -o thread **-lpthread**

pthread_create() returns a non-zero value on failure

The second argument to pthread_create() is **NULL** indicating to create a thread with default attributes.

For Your Own Interest

pthread_create by clone()

```
const int clone_flags = (CLONE_VM | CLONE_FS
                        | CLONE_FILES | CLONE_SYSVSEM
                        | CLONE_SIGHAND | CLONE_THREAD
                        | CLONE_SETTLS | CLONE_PARENT_SETTID
                        | CLONE_CHILD_CLEARTID
                        | 0);
```

It is asked to share the virtual memory, file system, open files, shared memory and signal handlers with the parent thread/process

```
TLS_DEFINE_INIT_TP (tp, pd);

if (__glibc_unlikely (ARCH_CLONE (&start_thread,
                              STACK_VARIABLES_ARGS,
                              clone_flags, pd, &pd->tid,
                              tp, &pd->tid)
                  == -1))
```

ARCH_FORK is an inline call to clone()

pthread_create by clone()
CLONE_THREAD

- **CLONE_THREAD**: the child is placed in the **same thread group** as the calling process.
- A new thread created with CLONE_THREAD has the same parent process as **the caller of clone()**

fork()

clone(CLONE_THREAD)

Grand parent
(PID=000)

Parent
(**PID=001**
PPID=000)

Child
(**PID=001**
PPID=000)

threads are all peers

Track your threads by ps -eLf

```
// th_name.c
#include <stdio.h>
#include <pthread.h>

void * f1() {
    printf("f1 : Starting sleep\n");
    sleep(30);
    printf("f1 : Done sleep\n");
}

int main() {

    pthread_t  f1_thread;
    pthread_create(&f1_thread, NULL, f1, NULL);
    pthread_setname_np(f1_thread, "f1_thread");

    printf("Main : Starting sleep\n");
    sleep(40);
    printf("Main : Done sleep\n");
    return 0;

}
```

The two threads created in the pthread program (th_name.c) have the same PID (**31088**), but different thread IDs (LWP=**31088** & **31089**)

Main thread: PID=LWP=**31088**
Both threads have **PPID=29342**

```
$ /tmp/th_name > /dev/null &
[3] 2055
$ ps -eLf | egrep "th_name|UID"
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
         31088 29342 31088  0    2 10:01 pts/4    00:00:00 /tmp/th_name
         31088 29342 31089  0    2 10:01 pts/4    00:00:00 /tmp/th_name
         31095 29342 31095  0    1 10:01 pts/4    00:00:00 egrep th_name|UID
```

View threads in workbench (syslog process in workbench)

```
USER         PID  SPID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1     1  0.0  0.0 226156  7296 ?        Ss   Oct06   7:06 /sbin/init
root          61    61
systemd+     169   169
systemd+     171   171
daemon       190   190
root         191   191  0.0  0.0 105980  4488 ?        Ss   Oct06   0:01 /usr/sbin/cron -f
root         193   193  0.0  0.0  62656  4508 ?        Ss   Oct06   0:27 /lib/systemd/systemd-logind
syslog       194   194  0.0  0.0 362640  7728 ?        Ssl  Oct06   0:00 /usr/sbin/rsyslogd -n
syslog       194   199  0.0  0.0 362640  7728 ?        Ssl  Oct06   0:13 /usr/sbin/rsyslogd -n
syslog       194   200  0.0  0.0 362640  7728 ?        Ssl  Oct06   0:00 /usr/sbin/rsyslogd -n
syslog       194   201  0.0  0.0 362640  7728 ?        Ssl  Oct06   0:16 /usr/sbin/rsyslogd -n
```

Syslog was designed to monitor network devices and systems to send out notification messages

```
clwang@workbench:~$ ps -L -p 194
  PID   LWP TTY          TIME CMD
  194   194 ?        00:00:00 rsyslogd
  194   199 ?        00:00:13 in:imuxsock
  194   200 ?        00:00:00 in:imklog
  194   201 ?        00:00:16 rs:main Q:Reg
clwang@workbench:~$
```

```
clwang@workbench:~$ pstree -p 194
rsyslogd(194)─┬─{rsyslogd}(199)
              ├─{rsyslogd}(200)
              └─{rsyslogd}(201)
clwang@workbench:~$
```

4 threads are created in ***syslog*** process (**PID=194**)

View PID, TID, LWP, TGID using ps

Total 3 threads: main thread + 2 sub-threads created by **pthread_create()**

```
# ps -eo pid,tid,lwp,tgid,pgrp,sid,tpgid,args -L | awk
  PID   TID   LWP  TGID  PGRP   SID TPGID COMMAND
20992 20992 20992 20992 20992 30481 20992 ./threadTest
20992 20993 20993 20992 20992 30481 20992 ./threadTest
20992 20994 20994 20992 20992 30481 20992 ./threadTest
```

- **Total 3 threads: TID=LWP= 20992, 20993 or 20994**
- **All threads shared the same PID and TGID (=20992)**

Which one is the thread group leader (main thread)? Answer: the thread with **TID 20992**

pthread_t pthread_self(void);

- pthread_self() **returns the ID of the calling thread**, which is the same value that is returned in ***thread** in the **pthread_create().**
- The returned thread IDs has nothing to do with the TID in Linux, only guaranteed to be unique **within a process**.
  - **POSIX thread IDs are not the same as the thread IDs returned by the Linux specific gettid() system call.**

Pthread ID vs PID and TID

- **int pthread_create(pthread_t *tid, …**
  - **pthread_t tid;** // unsigned long int; platform dependent, no meaning at the system level, guaranteed to be unique only within a process!
  - **int x = pthread_create (&tid,…**
- **pid_t fork(void);**
  - **pid_t** pid; // pid_t is a signed integer type, e.g, int
  - pid = getpid(); // the returned value is obtained from the Linux kernel (= TGID).
- **pid_t gettid(void);**
  - The returned value is obtained from the Linux kernel, a system-wide unique id.

pthread_self()

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void* func(void* p) {
   printf("From the function, the thread id = %d\n", pthread_self());
   pthread_exit(NULL);
}
main() {
   pthread_t thread; // declare thread
   pthread_create(&thread, NULL, func, NULL);
   printf("From the main function, the thread id = %d\n", thread);
   pthread_join(thread, NULL); //join with main thread
}
```

Print the same value

Output

```
From the main function, the thread id = 1
From the function, the thread id = 1
```

Note: Pthread ID is the ID provided by the Pthread library and has no meaning at the system level.

Tutorial 3 Sample Code

```c
/ * COMP3230 T3: show_tid.c */
#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>
#include <pthread.h>
#define gettidv1() syscall(__NR_gettid)
#define gettidv2() syscall(SYS_gettid)
```

```c
void *ThreadFunc1()
{
  printf("the pthread_1 id is %ld\n", pthread_self());
  printf(" thread_1's PID is %d\n", getpid());
  printf(" LWP (TID) of thread_1 is: %ld\n", (long int) gettidv1());
  pause();
  return 0;
}

```

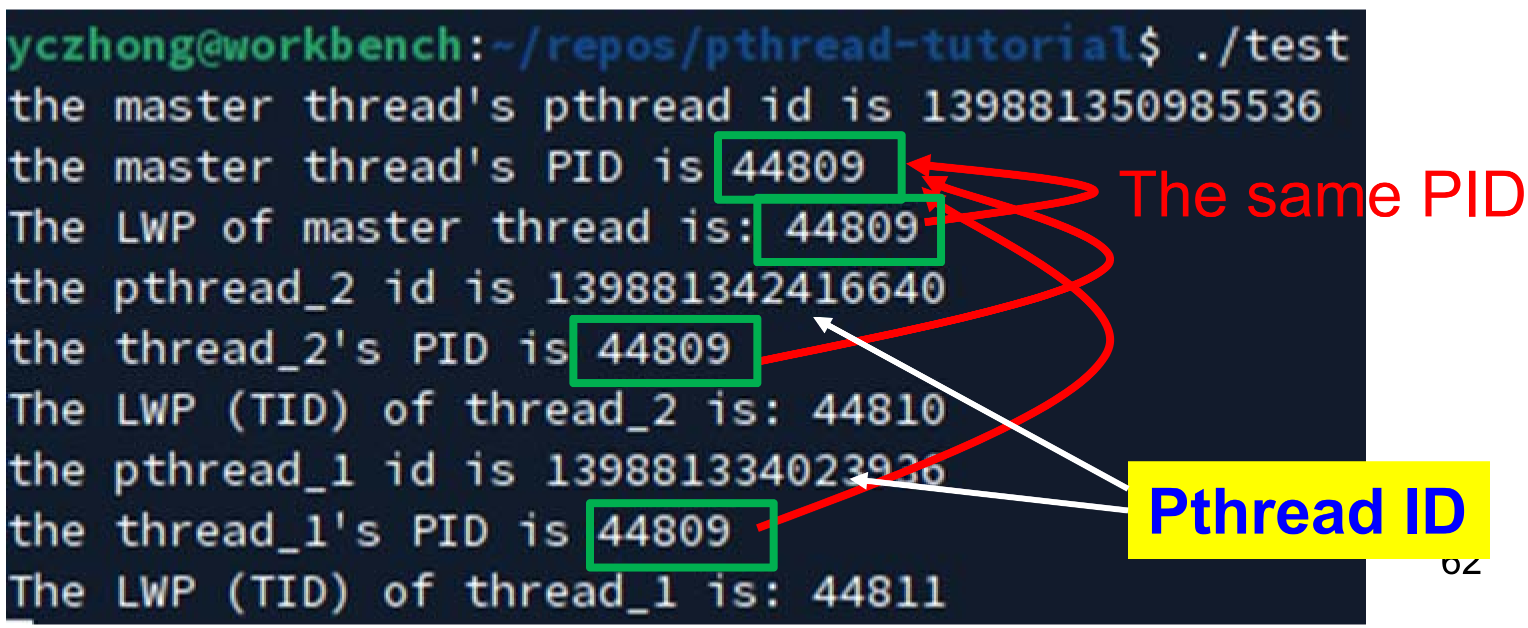
```c
void *ThreadFunc2()
{
  printf(" pthread_2 id is %ld\n", pthread_self());
  printf(" thread_2's PID is %d\n", getpid());
  printf(" LWP (TID) of thread_2 is: %ld\n",
            (long int) gettidv1());
     pause(); return 0;
}
```

LWP/TID vs pthread ID

```c
int main(int argc, char *argv[])
{
     pid_t tid; pthread_t pthread_id;
     printf("the master thread's pthread id is %ld\n", pthread_self());
     printf("the master thread's PID is %d\n", getpid());
     printf(" LWP of master thread is: %ld\n", (long int) gettidv1());
      // create two threads
      pthread_create(&pthread_id, NULL, ThreadFunc2, NULL);
      pthread_create(&pthread_id, NULL, ThreadFunc1, NULL);
         pause(); return 0;
}
```

LWP/TID vs pthread ID

- The LWP (TID) is the thread ID, which is **44811** and **44810** for thread_1 and thread_2, respectively.
- The PID is the process ID of the thread group leader, which is **44809** = thread1/2's PID



void pthread_exit(void *retval)

- **terminate the calling thread** and return a value via *retval* (used for inspection by other threads).
  - **pthread_exit()** provides an interface similar to **exit()** **but on a per-thread basis**
  - **pthread_exit()** will exit the thread that calls it, while the remaining threads can continue execution!
  - **pthread_exit() routine never returns! (thread terminated)**
  - **An implicit call to pthread_exit() occurs** when any thread **returns from its start routine**.
  - **pthread_exit()** will release any thread-specific data (e.g., **thread stack**). Any data allocated on the stack becomes invalid, because the stack is gone.

Note on pthread_exit()

- When a thread terminates, **process-shared resources** (e.g., file descriptors, mutexes, semaphores) **are not released**. It has to wait until **the last thread** in a process terminates.
    - For example, files are not closed, because they may be still used by other threads
- **Note: If any of the threads in your Pthreads program calls exit() → causing all others to be abruptly terminated(!) → exit()** should only be used only when the entire process needs to be terminated!

exit() vs pthread_exit()

- What happens to the entire process (and to other threads) when pthread_exit() is called from the main() thread?
  - The main thread will stop executing and will remain in zombie (defunct) status until all other threads exit → *zombie thread (slide 69)*!

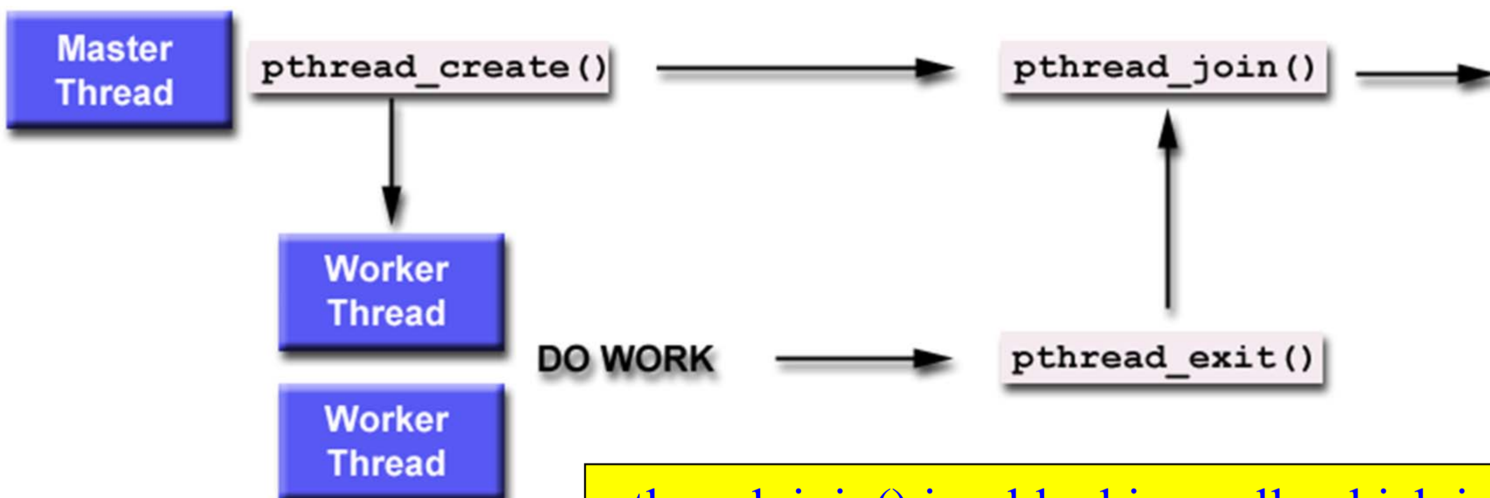
```
int main(int argc, char* argv[]){
  int rc;
  pthread_t thread_id;
  thread_t tid;

  tid = pthread_self();
  printf("\nmain thread(%d) ", tid

  rc = pthread_create(&thread_id, 
  if(rc){
    printf("\n ERROR: return code 
    exit(1);
  }
  sleep(1);
  printf("\n Created new thread (%
  pthread_exit(NULL);
}
```

the main thread is terminated

pthread_join()

int pthread_join(pthread_t thread, void **value_ptr);

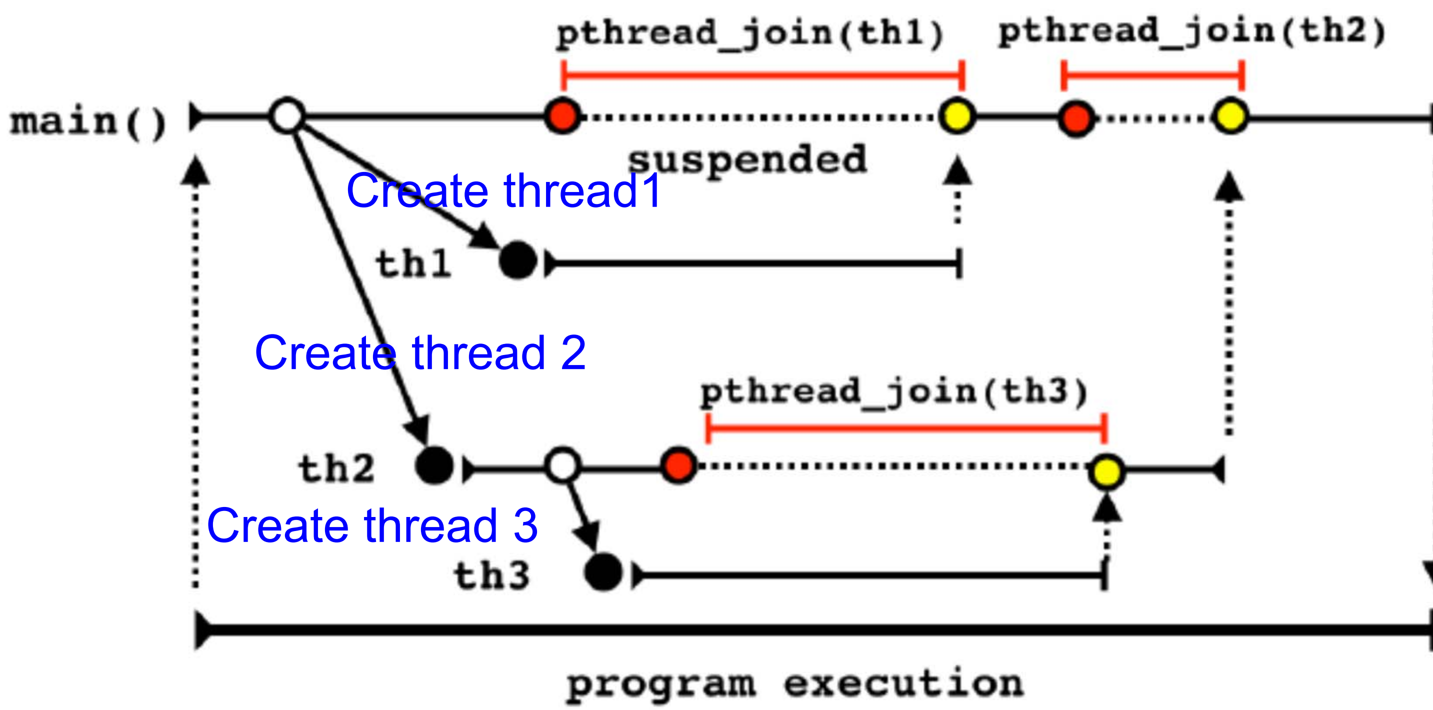
- The pthread_join() function waits for the thread specified by ***thread*** to terminate. If that thread has already terminated, then pthread_join() returns immediately.



pthread_join() is a blocking call, which is similar to **wait()** and **waitpid()** in fork()

pthread_join()
Multithreaded program lifecycle

- New threads are dynamically created using pthread_create()
- **pthread_join() suspends the calling thread until the thread it waits for terminates.**



Pthread Joining Example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );
main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;
   /* Create independent threads each of which will execute function */
    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

This example demonstrates how to "wait" for thread completions by using the **pthread_join().**

exit() will terminate the entire process including any threads it created.

Results:
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0

"print_message_function" is used in each thread, but the arguments are different.

exploring the living universe...

POS

COMP 3230

Zombie Threads

- A thread is said to be a ***zombie thread*** if it has terminated and no other thread has collected its return value with **pthread_join()** yet.
- **Zombie threads** pose very similar problems to **zombie processes**—waste of system resources and, eventually, thread ID (PID in Linux) exhaustion (making impossible to create new threads).
- **To avoid thread zombies:**
  - **use pthread_join() properly**, or
  - make the thread ***non-joinable*** using **pthread_detach**();

pthread_detach()

int pthread_detach(pthread_t *thread)

- **Joinable threads** must be reaped or killed by other threads using pthread_join()
  - **By default all threads are joinable,** so to make a thread detached we need to call **pthread_detach() explicitly** with thread id.
- **Non-joinable threads** are automatically reaped at termination without the need for another thread to join with the terminated thread
  - pthread_attr_setdetachstate((&attr, PTHREAD_CREATE_DETACHED)

```
void* func(void* arg)
{
    // detach the current thread
    // from the calling thread
    pthread_detach(pthread_self());

    printf("Inside the thread\n");

    // exit the current thread
    pthread_exit(NULL);
}
```

pthread_detach()

int pthread_detach(pthread_t *thread)

- **Note:** The pthread_detach() function just "**marks**" the thread identified by ***thread*** as detached.
  - Normally you call pthread_detach from either the **new thread itself** or the **creating thread** (e.g., right after pthread_create).
  - **Note: If the thread *thread* has not terminated, pthread_detach() does not cause the thread to terminate.**
- When a detached thread terminates (pthread_exit() or complete the function call), its allocated resources (e.g., stack-allocated data) are automatically released back to the system.

Please free up my resource when I exit

Q: Will printf() be executed?

```c
/* function to be executed by the new thread */
void* PrintHello(void* data)
{
    int my_data = (int)data;            /* data received by thread */

    pthread_detach(pthread_self());
    printf("Hello from new thread - got %d\n", my_data);
    pthread_exit(NULL);                 /* terminate the thread */
}

/* like any C program, program's execution begins in main */
int main(int argc, char* argv[])
{
    int         rc;             /* return value                      */
    pthread_t   thread_id;      /* thread's ID (just an integer)     */
    int         t         = 11; /* data passed to the new thread     */

    /* create a new thread that will execute 'PrintHello' */
    rc = pthread_create(&thread_id, NULL, PrintHello, (void*)t);
    if(rc)                      /* could not create thread */
    {
        printf("\n ERROR: return code from pthread_create is %d \n", rc);
        exit(1);
    }
    printf("\n Created new thread (%u) ... \n", thread_id);

    pthread_exit(NULL);         /* terminate the thread */
}
```

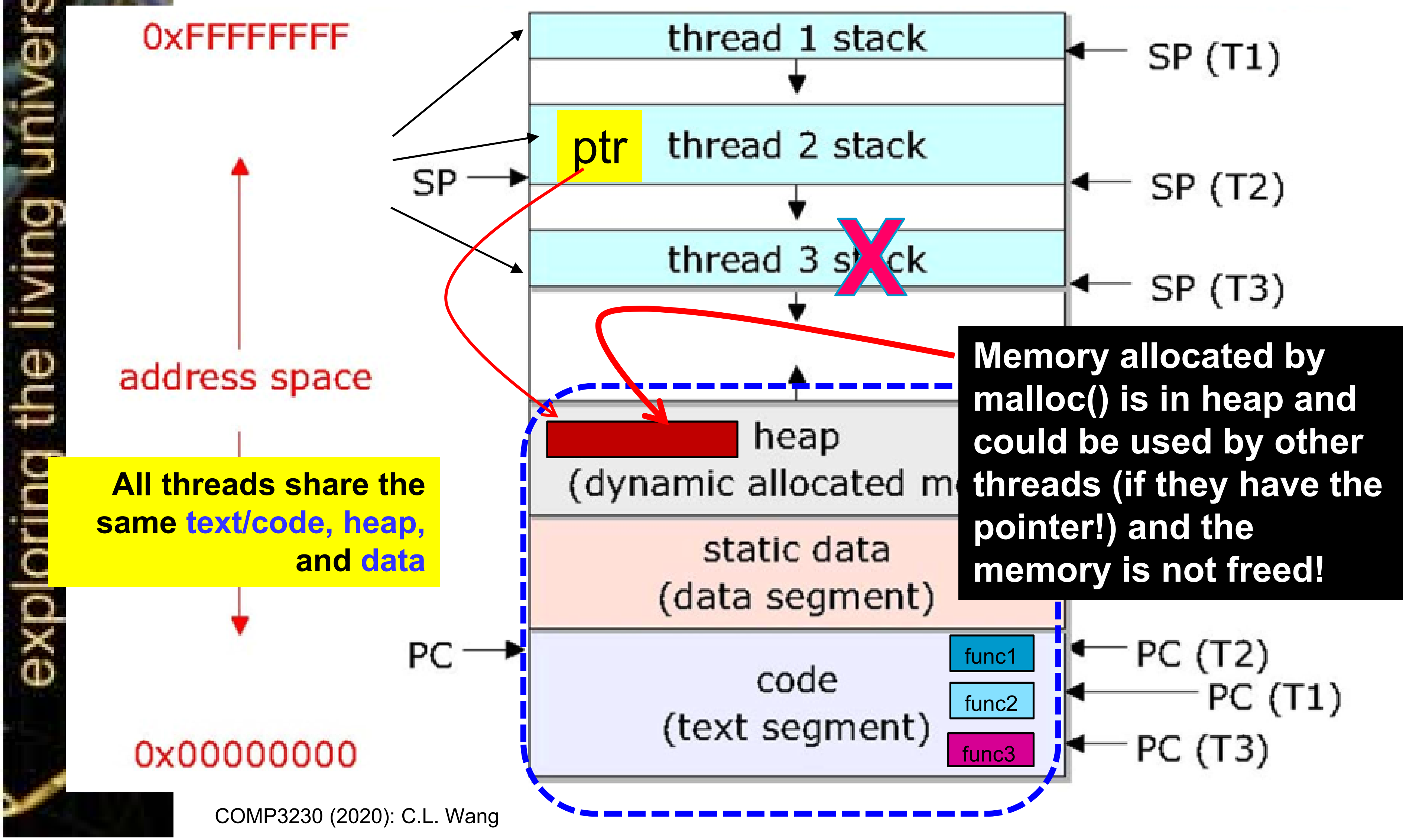
No pthread_join() needed!

```
void *test_thread(void * arg)
{
pthread_detach(pthread_self());
 int *c = malloc(2048);
  pthread_exit(NULL);
}
```

Self-test Question

- **Q: Will a thread's malloc()'ed memory (in heap) be automatically released by pthread_detach() call?**
- **Answer: No.**
  - **Any memory allocated via malloc() still needs to be free()'ed manually**. Either the thread needs **free up the memory before it exits**, or a pointer to the malloc'ed memory needs to be **made available to another thread** to clean up.
  - **Note:** the only thing that **pthread_detach** does for you is that you don't need to call **pthread_join** to clean up the internal thread data structures.

Recall Address Space of a Multi-threaded Program



Differences between pthread_exit, pthread_join, pthread_detach

- **pthread_exit** is called from the ***thread itself*** to terminate its execution
- **pthread_join** is called from ***another thread*** (usually the thread that created it) to wait for a thread to terminate and **obtain its return value**.
- **pthread_detach** can be called from either the ***thread itself*** or ***another thread***, and indicates that you don't want the thread's **return value** nor need to wait for it to finish.

exploring the living universe...

COMP 3230

Thread-safeness

(Example: Race condition)

- Considering the following: data=0 initially

THREAD 1

```
a = data;
a++;
data = a;
```

THREAD 2

```
b = data;
b--;
data = b;
```

If execute concurrently

	THREAD 1	THREAD 2	
0	`a = data;`		0
		`b = data;`	
1	`a++;`		
		`b--;`	-1
1	`data = a;`		
		`data = b;`	-1

If execute sequentially (thread 1 → thread 2): value of *data=0* (doesn't change).

Race condition: one possible result can be data = -1

A ***race condition*** is a condition of a program where its behavior depends on relative timing or interleaving of multiple threads or processes → **Unpredictable results.**

Thread safe
mutex (MUTual EXclusion)

- **Thread safe**:
  - Implementation is guaranteed to be **free of race conditions** when accessed by multiple threads simultaneously
  - All threads behave properly and fulfill their design specifications without unintended interaction.
- **Mutual exclusion**: Access to shared data is serialized
  - Provides **locking/unlocking** critical code sections where ***shared data (e.g., "data" in slide 76)*** is modified.
  - A ***mutex*** can be owned, i.e. "locked", **by at most one thread at any given time.**
- ***Mutex object*** is one of the primary means of implementing thread synchronization and for protecting shared data when **multiple writes occur**.

Race Condition: Example

- An example of a race condition involving a bank transaction

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

Balance=balance+200

Balance=balance+200

A **mutex** should be used to lock the **"Balance"** while a thread is using this shared data resource.

Thread Synchronization using Mutex

- **Basic Mutex Functions:**
  - int **pthread_mutex_init**(pthread_mutex_t ***mutex**,
  - const pthread_mutexattr_t *mutexattr);
  - int **pthread_mutex_lock**(pthread_mutex_t ***mutex**);
  - int **pthread_mutex_unlock**(pthread_mutex_t ***mutex**);
  - int **pthread_mutex_destroy**(pthread_mutex_t ***mutex**);

- Data type named **pthread_mutex_t** is designated for **mutexes**
- The attribute of a ***mutex*** can be controlled by using the **pthread_mutex_init()** function

pthread_mutex_lock/unlock()

- **int pthread_mutex_lock (pthread_mutex_t *mutex);**
  - The mutex object referenced by ***mutex*** shall be locked by calling pthread_mutex_lock().
  - If the ***mutex*** is already locked, the calling thread shall block until the mutex becomes available.
- **int pthread_mutex_unlock (pthread_mutex_t *mutex);**
  - The pthread_mutex_unlock() function shall release the mutex object referenced by mutex.
- **The code in-between the lock and unlock calls is called a *critical section*.**

Thread Synchronization: Mutex

- A typical sequence in the use of a mutex is as follows:
  - Create and initialize a **mutex** variable
  - Several threads attempt to lock the **mutex**
  - **Only one succeeds and that thread owns the mutex (the losers block at that call)**
  - The owner thread performs some set of actions
  - **The owner unlocks the mutex**
  - Another thread acquires the mutex and repeats the process **(but which one? Starvation?)**
  - Finally the mutex is **destroyed**

pthread_mutex_lock/unlock

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

#define NUM_THREADS 5

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

/* shared data between threads */
double shared_x;
pthread_mutex_t lock_x;
```

```c
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);
  /* get mutex before modifying and printing shared_x */
  pthread_mutex_lock(&lock_x);
    shared_x += data->stuff;
    printf("x = %f\n", shared_x);
  pthread_mutex_unlock(&lock_x);

  pthread_exit(NULL);
}
```

critical section

```c
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* initialize shared data */
  shared_x = 0;

  /* initialize pthread mutex protecting "shared_x" */
  pthread_mutex_init(&lock_x, NULL);

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    thr_data[i].stuff = (i + 1) * NUM_THREADS;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

Updated by threads

*Within the **thr_func()** we call **pthread_mutex_lock()** before reading or modifying the shared data.*

This program creates NUM_THREADS threads

pthread_join waits for the thread specifies in the first argument to exit

Thread Synchronization: Programmer's Responsibility

- When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so.
- For example, if 3 threads are updating the same data, but only two use a mutex, the data can still be corrupted.
- **Other Problems: deadlocks, livelocks, and starvation (Lecture 6).**

Thread 1	Thread 2	Thread 3
Lock	Lock	
A = 2	A = A+1	A = A*B
Unlock	Unlock	

What is the "correct" result you expected → Programmer's Responsibility

CPU Affinity

- int **pthread_setaffinity_np**(*pthread_t thread*, *size_t cpusetsize*, *const cpu_set_t *cpuset*);
  - Set CPU affinity of a thread
  - "_np" in the name stands for "non-portable".
- int **pthread_getaffinity_np**(*pthread_t thread*, *size_t cpusetsize*, *const cpu_set_t *cpuset*);
  - Get CPU affinity of a thread
- Arguments:
  - 1. *pthread_t thread*: thread to be bind to CPU cores specified in cpuset (3rd argument)
  - 2. *size_t cpusetsize*: the length (in bytes) of the buffer pointed to by cpuset (3rd argument). Typically, this argument would be specified as *sizeof(cpu_set_t)*
  - 3. *const cpu_set_t *cpuset*: data structure represents a set of CPUs implemented as a bit mask

(Discuss more in Tutorial 3)

CPU Affinity: CPU Set

- All manipulation of CPU sets should be done via the macros

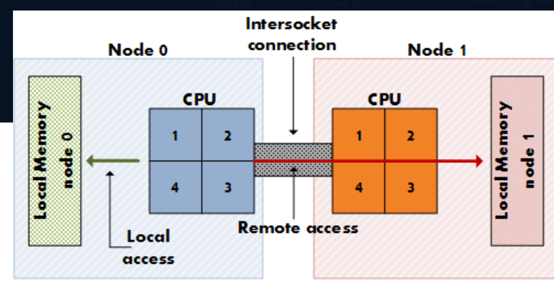
MACRO	Description
void **CPU_ZERO**(cpu_set_t *set);	Clears set, so that it contains no CPUs. Initialize set
void **CPU_SET**(int cpu, cpu_set_t *set);	Add CPU cpu to set
void **CPU_CLR**(int cpu, cpu_set_t *set);	Remove CPU cpu from set
int **CPU_ISSET**(int cpu, cpu_set_t *set);	Test to see if CPU cpu is a member of set
int **CPU_COUNT**(cpu_set_t *set);	Return the number of CPUs in set

(Discuss more in Tutorial 3)

Check NUMA nodes on workbench
(Slide from Tutorial 3)

- lscpu

```
yczhong@workbench:~$ lscpu | egrep -i 'core.*:|socket'
Thread(s) per core:    2
Core(s) per socket:    22
Socket(s):             2
```



- numactl

```
yczhong@workbench:~$ numactl -aH
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 4
6 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86
node 0 size: 192123 MB
node 0 free: 135411 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 4
7 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87
node 1 size: 193526 MB
node 1 free: 101019 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

RAM size: Node 1: 192GB
RAM size: Node 2: 193GB

Example (vec_sum.c)
(Slide from Tutorial 3)

- Setting CPU affinity makes multithreading program faster on a NUMA machine

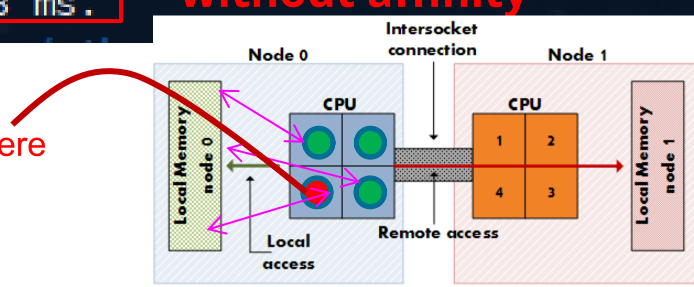
```
yczhong@workbench:~/repos/pthread-tutorial$ ./vec_sum 40960000 12
vector len=40960000. thread num=12
The elapsed time is 7.69 ms.
Main thread runs on CPU 61
Set affinity mask to include CPUs (1, 3, 5, ...  2n+1)
The elapsed time is 6.41 ms.
yczhong@workbench:~/repos/pthread-tutorial$ ./vec_sum 40960000 12
vector len=40960000. thread num=12
The elapsed time is 8.12 ms.
Main thread runs on CPU 10
Set affinity mask to include CPUs (0, 2, 4, ...  2n)
The elapsed time is 6.48 ms.
```

without affinity

with affinity

without affinity

Main thread running here



exploring the living universe...

POS

Reference

- Threads: Basic Theory and Libraries
  - http://www.cs.cf.ac.uk/Dave/C/node29.html
- The Linux Process Model (2000)
  - http://www.linuxjournal.com/article.php?sid=3814
- Green Threads (user-space threads)
  - http://en.wikipedia.org/wiki/Green_threads
- Using the Clone() System Call
  - https://www.linuxjournal.com/article/5211
  - https://eli.thegreenplace.net/2018/launching-linux-threads-and-processes-with-clone/
- Linux Tutorial: POSIX Threads
  - http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

Self-Test Question

- Why is <u>switching threads</u> less costly than <u>switching processes</u>?
  - Less state needs to be saved and restored (threads share the same address space).
  - Switching between threads benefits from caching (also TLB can be reused); whereas, switching between processes has to flush the L1/L2 CPU cache and TLB. (Lecture Note **4**)
  - Note: flushing the cache is expensive: if the cache is dirty, it has to be flushed back to memory (memory accesses!)

Sample Exam Question (1)

- Which of the following is shared between threads of the same process?
  1. Program counter.
  2. Heap memory.
  3. Stack memory.
  4. Global variables.
  5. Open files.

Threads share:
- Address space
- Heap
- Static data
- Code segments
- File descriptors
- Global variables
- Child processes
- Pending alarms
- Signals and signal handlers
- Accounting information

Threads have their own:
- Program counter
- Registers
- Stack (stack pointer)
- State

Sample Exam Question (2)

- Discuss two main differences between **user-level** threads (e.g., Pthreads' PTHREAD_SCOPE_PROCESS) and **kernel-level** threads (PTHREAD_SCOPE_SYSTEM).
- **Answer:**
  - User-level threads are unknown by the kernel, whereas the kernel is aware of kernel threads.
  - User-level threads are scheduled by the user-space **thread library** (allow each process to have its own customized scheduling algorithm), while kernel-level threads are scheduled by the kernel scheduler.

User-level Threads

- **Many-to-one**: **OS maps all threads created by a program to the same process.**
- **Examples:**
  - **Solaris: "Green Threads"**
  - **GNU Portable Threads**
- **Thread scheduling/switching:**
  - Managed entirely by the run-time system (user-level library). thread_create, thread_exit, thread_wait, and thread_yield are all executed in user mode.
  - Each process needs its own private thread table in user space (analogous to the kernel's process table) to keep track of the threads in that process.
- **Thread switching does not require kernel mode privileges (all in user mode, no mode switch)**

Thread context switch in user-level threads

- **Very simple for user-level threads belonged to the same process:**
  - Save context of currently running thread
    - **Push CPU state onto thread stack**
  - Restore context of the next thread
    - **Pop CPU state from next thread's stack**
  - Return as the new thread
    - **Execution resumes at PC of next thread**
  - Note: no changes to memory mapping required!
- **This is all done by assembly language → very fast (also no mode switching: user → kernel)**

Sample Exam Question (3)

- **(2010) In a many-to-one thread model (user-level thread), why does the operating system block the entire program when a single thread blocks ?**
  - The OS has no notion of threads. Process as whole gets one time slice (e.g., 200ms) irrespective of whether process has 1 thread or 100 threads within it (all threads share the given time slice).
  - Since all threads are mapped to a single process, when a user-level thread is blocked on an I/O event (e.g., read()/write() system calls), the whole process is blocked until the disk I/O is complete.

Sample Exam Question (5)

- True/**False**: Linux kernel does not distinguish between a process and a thread. Therefore, pthread_create() usually calls a fork() system call to create a new thread. (X, clone)
- **True**/False: The pthreads can be implemented as pure user-space threads, kernel-supported threads, or the combination of the two. (**2017)**
- **True**/False: In Linux, all the threads created by the same process share the same PID, which is internally the **thread group ID (TGID).** (**2017**)
- **True**/False: Each thread in Linux kernel is represented with the same data structure "task_struct" like the process. (**2017**)

Sample Exam Question (6)

- **(2014)** Which of the following statements about "threads" (within the same process) is **INCORRECT**? **(4)**
  1) A thread is part of a process; a process may contain several different threads.
  2) Two threads of the same process have different values of the program counter; different stacks (local variables); and different registers (e.g., PC, SP).
  3) Two threads share the same code, data, and heap memory as they shared the same process table.
  4) Java and C# are programming languages that support threads. They can only be executed in an OS that supports kernel-level threads. (X, depends on implementation, e.g., Green Threads in early JVMs are "user-level threads". Green thread memory is **allocated from the heap** rather than having a stack created for it by the OS)

exploring the living universe...

POS

COMP 3230

Sample Exam Question (7)

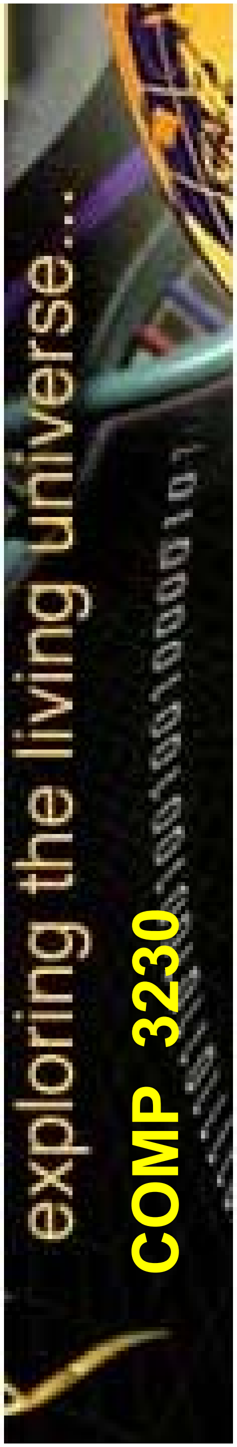
- **Q: Do threads have their own page table?**
- **Answer: No, all threads of a process share the same page table.**
  - The point of having threads is to be able to have multiple tasks operating on the same memory. Threads are supposed to be fast, so there would be a lot of overhead if the entire page table had to be copied whenever a thread was created. **(Read extra slide to see the performance comparison)**
  - We also want context switching to be fast, so we want to avoid having to switch between page tables when a different thread starts to execute. If they share the same page table, some virtual-to-physical address mappings can be stored in TLB for reuse → faster address translation. **(Lecture 3)**

Sample Exam Question (8)

```c
#include <pthread.h>

#define PAGE_SIZE 4096
#define STK_SIZE (10 * PAGE_SIZE)

void *stack;
pthread_t thread;
pthread_attr_t attr;

void *dowork(void *arg)
{
 int data[2*STK_SIZE];
 int i = 0;
 for(i = 0; i < 2*STK_SIZE; i++) {
   data[i] = i;
 }
}
```

```c
int main(int argc, char **argv)
{
  //pthread_attr_t *attr_ptr = &attr;
  posix_memalign(&stack,PAGE_SIZE,STK_SIZE);
  pthread_attr_init(&attr);
  pthread_attr_setstack(&attr,&stack,STK_SIZE);
  pthread_create(&thread,&attr,dowork,NULL);
  pthread_exit(0);
}
```

- If you run this program what will happen?



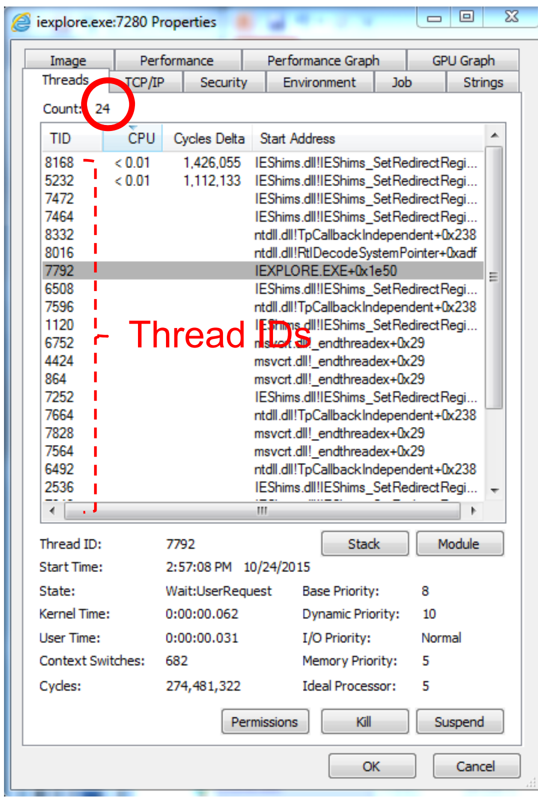
Extra Slides

Show Threads in IE

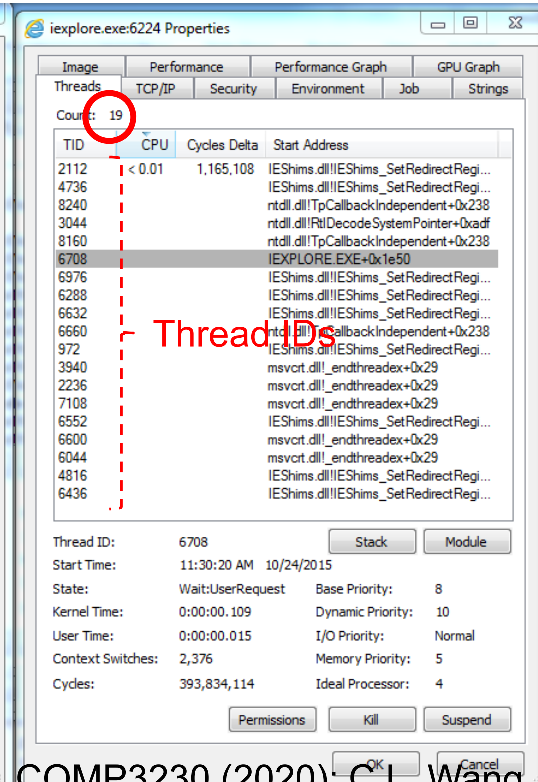
(After open 3 IE Explorers)

				PID		
iexplore.exe	0.01	31,712 K	52,332 K	6464	Internet Explorer	Microsoft Corporation
iexplore.exe	< 0.01	49,620 K	55,992 K	6224	Internet Explorer	Microsoft Corporation
iexplore.exe	4.64	308,228 K	276,848 K	7940	Internet Explorer	Microsoft Corporation
iexplore.exe	0.04	119,156 K	142,584 K	7280	Internet Explorer	Microsoft Corporation

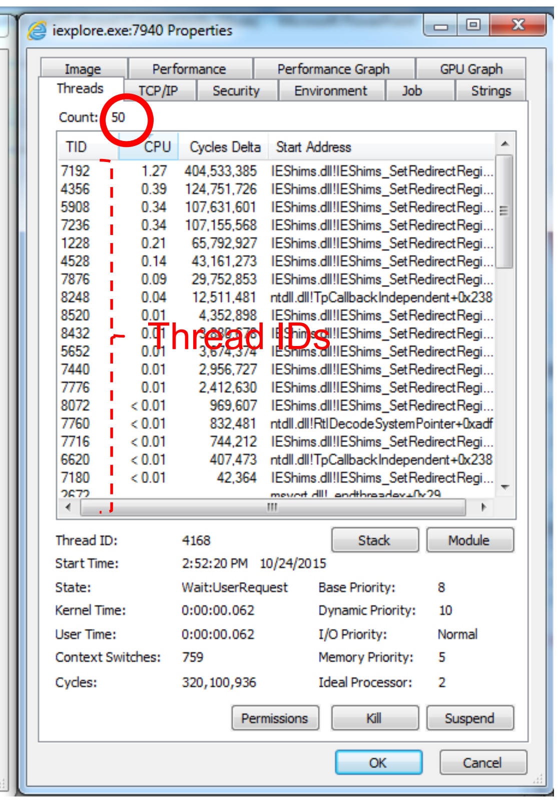
24 threads created (**PID=7280**)



19 threads created (**PID=6224**)



50 threads created (**PID=7940**)



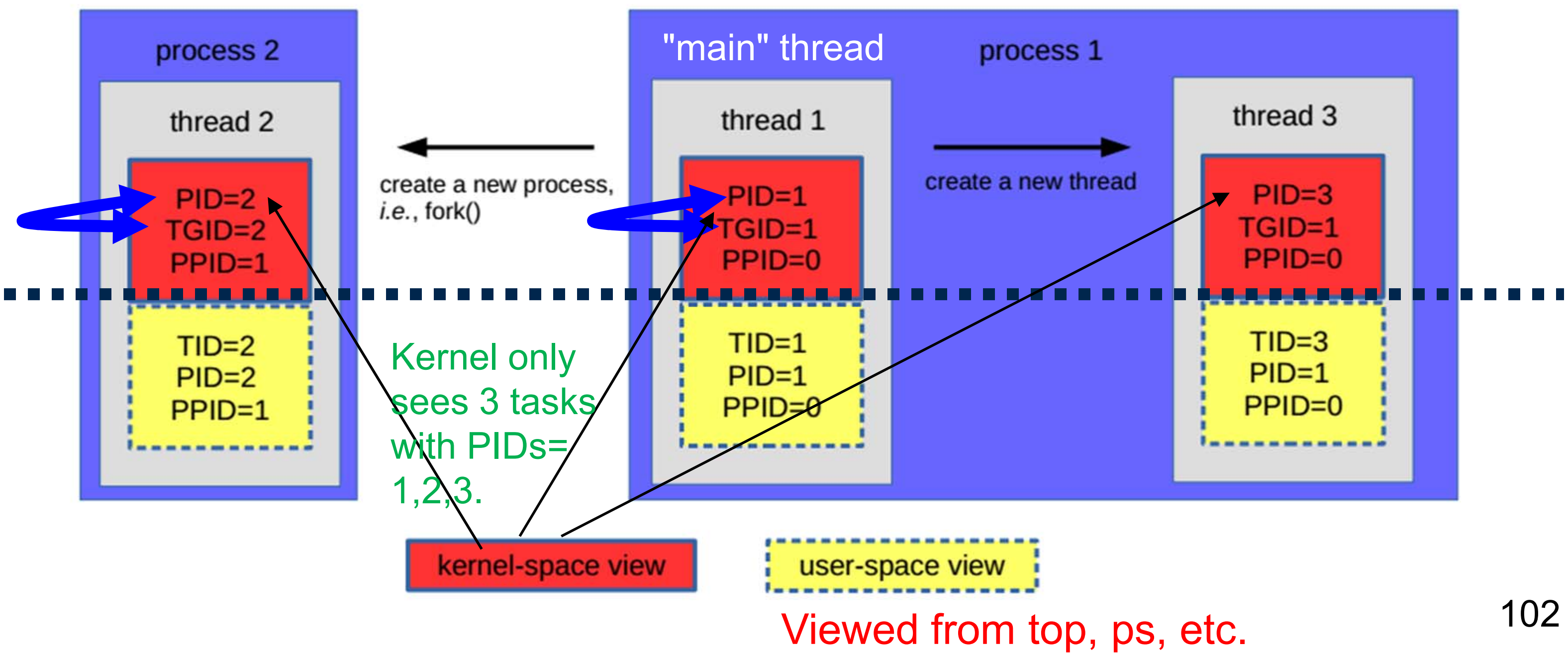
PID, TID, LWP, PPID, PGID

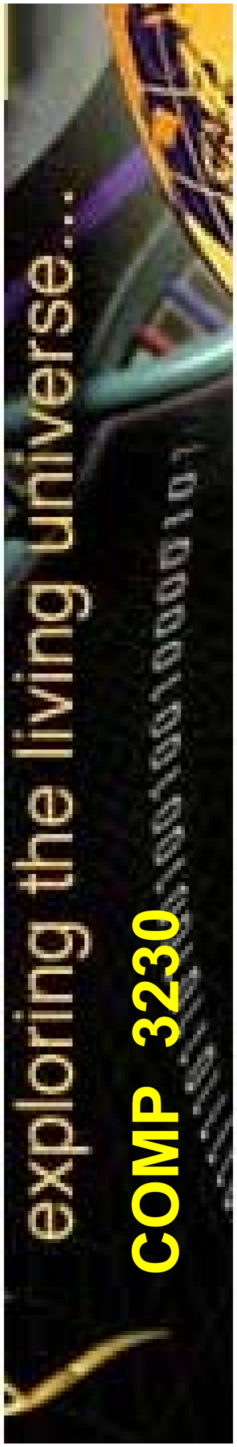
```
clwang@workbench:~$ ps -o pid,tid,lwp,ppid,pgid,cmd
  PID   TID   LWP  PPID  PGID CMD
73970 73970 73970 73969 73970 -bash
79881 79881 79881 73970 79881 ps -o pid,tid,lwp,ppid,pgid,cmd
clwang@workbench:~$ _
```

- **LWP = TID (Thread ID)**
  - TID is retrieved by sys_gettid(), syscall(__NR_gettid)
- **TGID: thread group ID**
  - all threads of a multithreaded program share the same **TGID** (= PID of the main thread)
- **PGID: process group ID**
  - PID of the process group leader
  - When a child process is forked, it inherits the PGID from its parent
  - PGID can be retrieved getpgrp(), and set by setpgid()

Relationship among PID, TID, PPID, and TGID

- When a new process starts by invoking **fork(),** it is assigned a new **TGID**. This newly forked process is created with a single thread, whose **PID** is the same as the **TGID**.
- User-space "PIDs", like those shown in ps/top or in /proc, **are actually "TGIDs" in the kernel.**





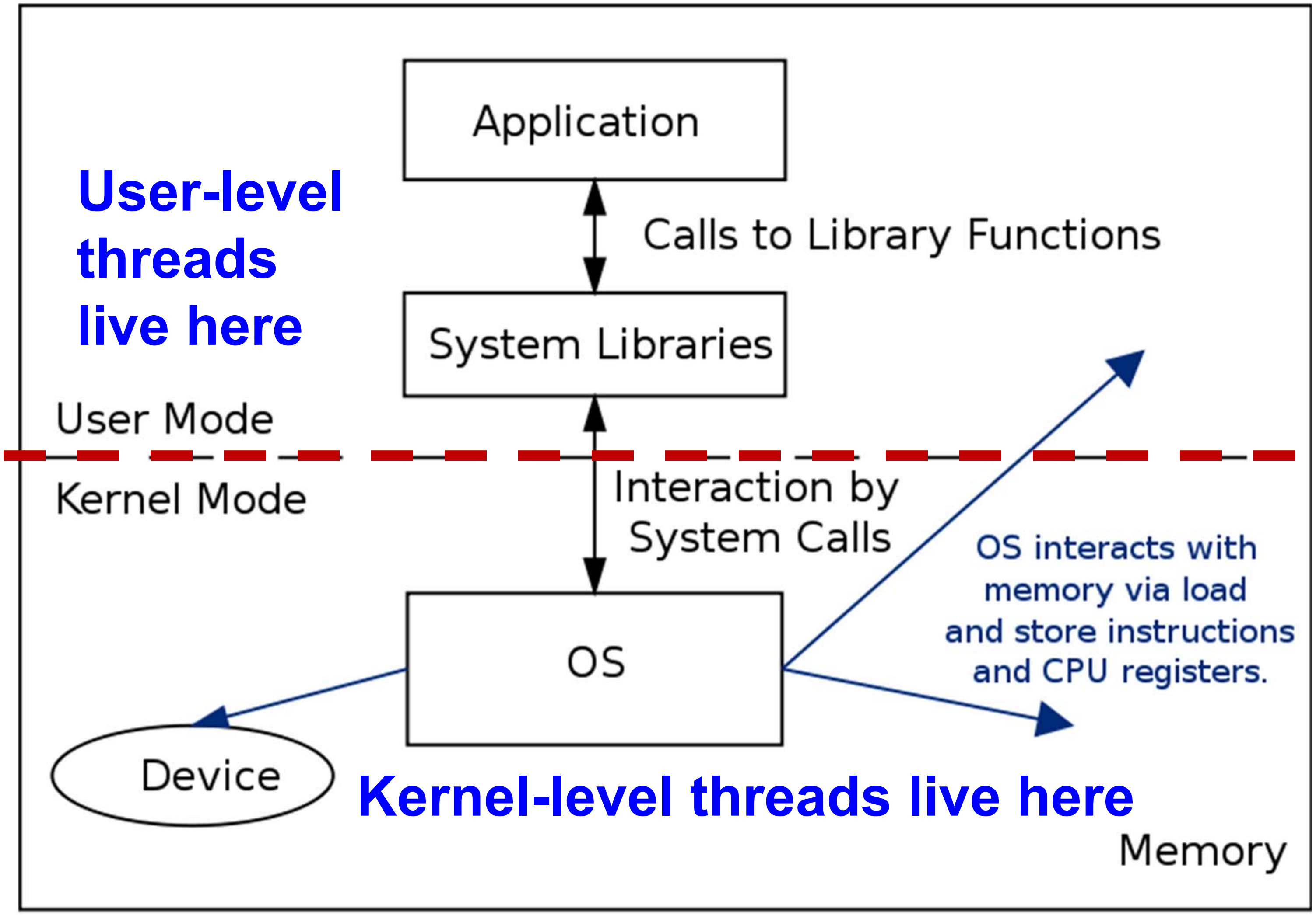
Implementation Models

<u>Three models:</u>

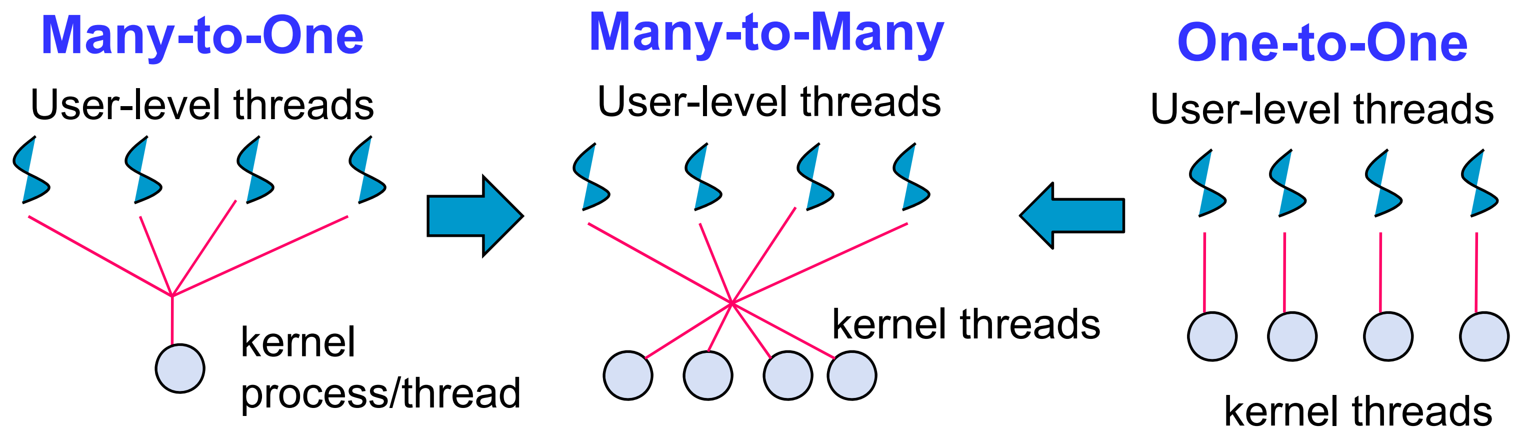
1. **User-level threads (Many-to-One)**
2. **Kernel-level threads (One-to-One)**
3. **Hybrid: Combination of user- and kernel-level threads (Many-to-Many)**

"Implementation-specific": *The Pthreads standard can be implemented as pure user-space threads, kernel-supported threads (most common), or a hybrid approach.*

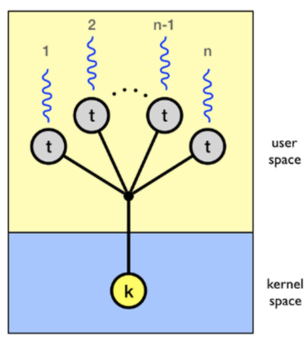
Recall User Mode vs Kernel Mode



Thread Models: Overview

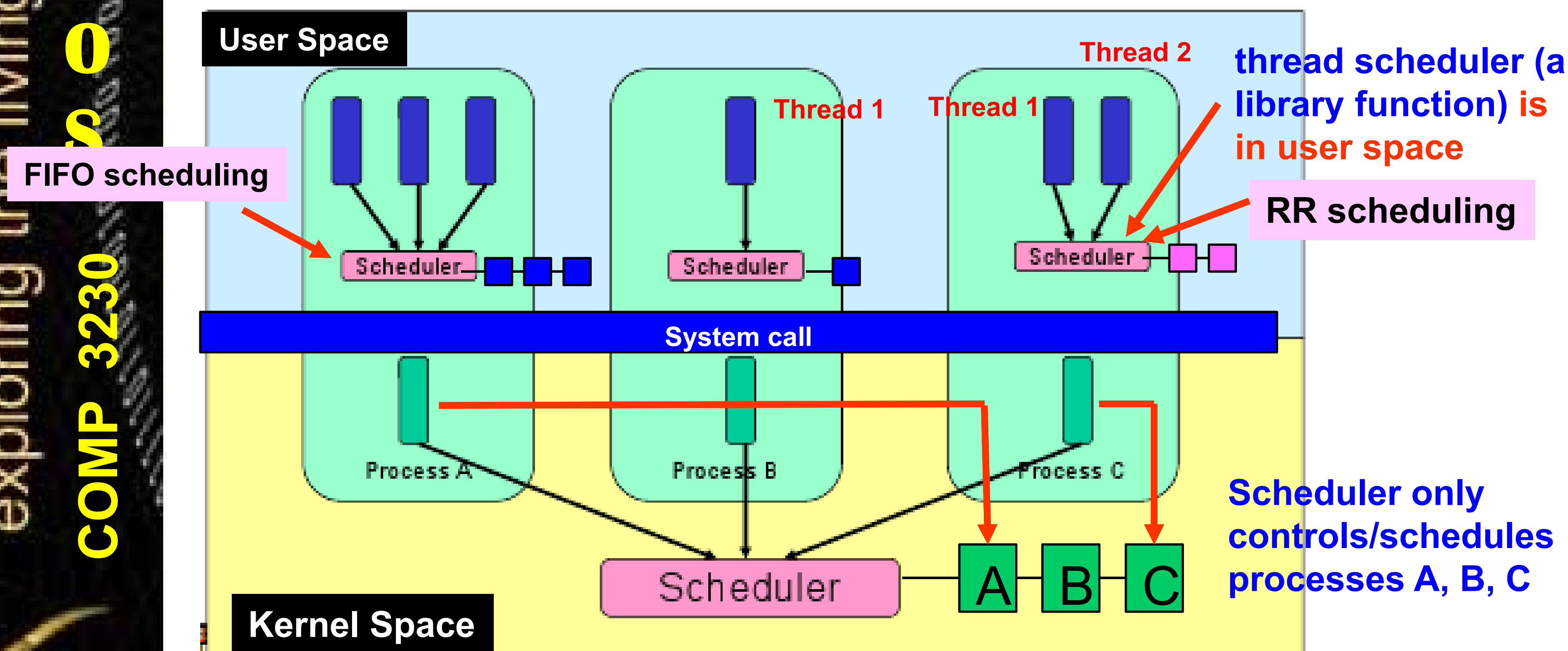


- Many-to-one (M:1)
  - All threads created in your program are mapped to one single process → OS only sees a single process.
- One-to-one (1:1)
  - Map one user thread to one kernel thread (real thread)
- Many-to-many (M:N)
  - Combined user- and kernel-level threads
  - M user-level threads are mapped onto N kernel-level threads, and M>N



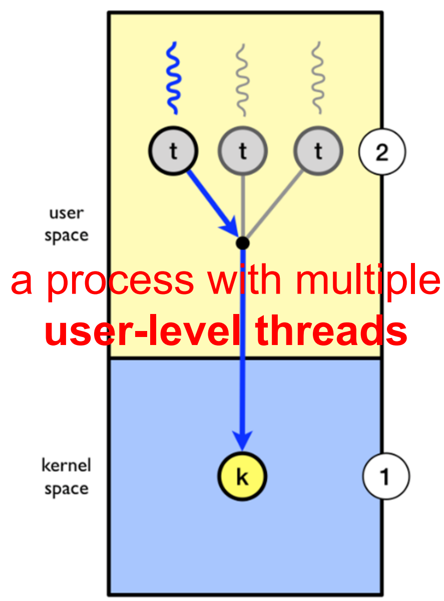
(1) User-level Threads (2)

- The kernel is **not aware** of the existence of threads. It sees only "process" (i.e., process **A, B, C** in this example).
- All thread management and **scheduling** are done using the **thread library at user space (thread scheduler is in user space).**
- Library calls help to switch CPU among different threads;

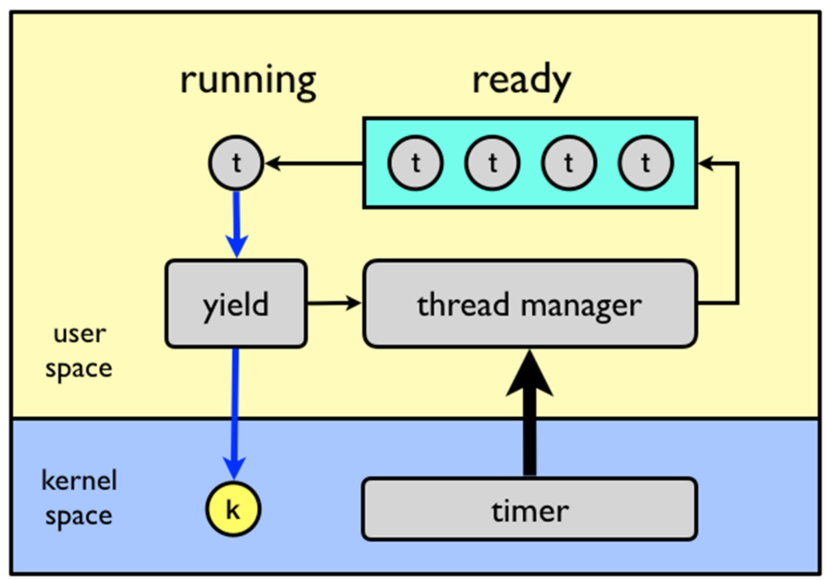


Many-to-one: Implementation

- Threads may **yield** to each other voluntarily, by calling a **yield()** function provided by the user-level thread library; or
- A **timer** is used to cause execution flow to **jump to a central thread manager** (@user space), which chooses the next thread to run.



"scheduling of threads in user-space"



User-level Threads

- **Advantages**
  - ***Flexible scheduling***: User-space scheduler can schedule its threads to optimize performance (easy to change the policy)
  - ***Cheap synchronization:*** Synchronization performed outside kernel, **avoids context switches** → **cheaper and faster.**
  - ***Fast thread creation:*** share context with existing threads, just need to create/allocate **thread stack**, **PC**, and **registers**.
  - ***Resource efficiency:*** Kernel memory isn't wasted for each user thread as all threads share one PCB (**task_struct**)
  - ***More portable:*** implemented entirely with user-space standard library calls
- **Disadvantage**
  - All threads are blocked if a thread issues I/O
  - **No speedup in multicore:** Different threads cannot be scheduled on multiple cores at the same time.

Thread Creation Time Comparison

Testing platform: 700MHz Pentium running Linux 2.2.16:

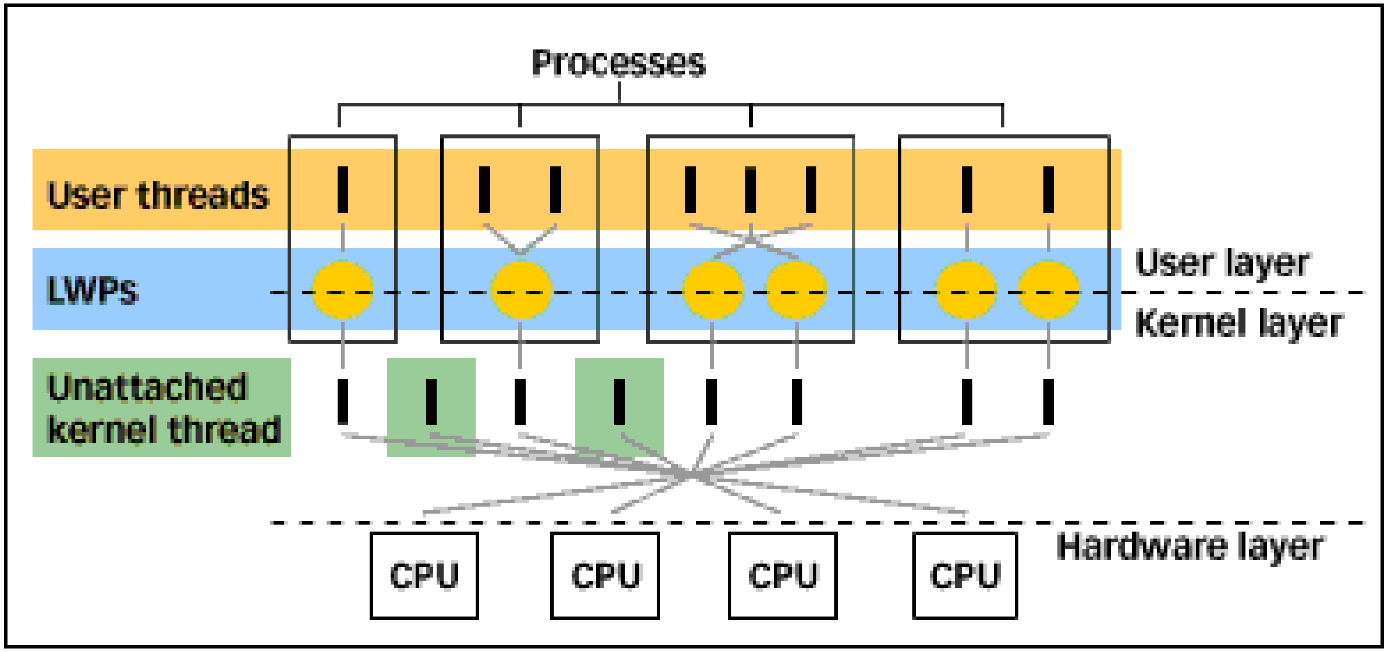
- Processes:
  - fork/exit: **251 ms**
- Kernel threads
  - pthread_create()/pthread_join(): **94 ms** (2.5x faster – ~150ms faster)
- User-level threads
  - pthread_create()/pthread_join: **4.5 ms** (another 20x faster - ~100ms faster)

COMP 3230 POS exploring the living universe...

(3) Many-to-Many

- **Support both <u>user-level threads</u> and <u>kernel threads</u> to the programmer.**
  - **<u>User-level threads</u> are *multiplexed* on top of <u>kernel-level threads</u>, which in turn are scheduled on top of processors/cores.**
  - **The kernel knows only about the kernel-level threads.**
- **A hybrid approach, aims to combine the advantages of both the many-to-one model and the one-to-one model, while minimizing these models' disadvantages.**
- **Examples: Solaris 8 and earlier, IRIX, HP-UX, Tru64 UNIX, Windows NT/2000 with the ThreadFiber package.**

Many-to-Many (M:N) : Solaris 2



- **User Threads**: created explicitly by the programmer. Invisible to the OS. Every user thread shares a process address space (Many-to-one).
- **Kernel threads**: schedulable entities in OS.
- **Lightweight processes (LWP)** supports one or **more user-level threads** and maps to exactly one kernel-level thread (one-to-one)

fork() vs. pthreads_create()

Timings reflect **50,000** process/thread creations, were performed with the **time** utility, and units are in seconds, no optimization flags. (Source)

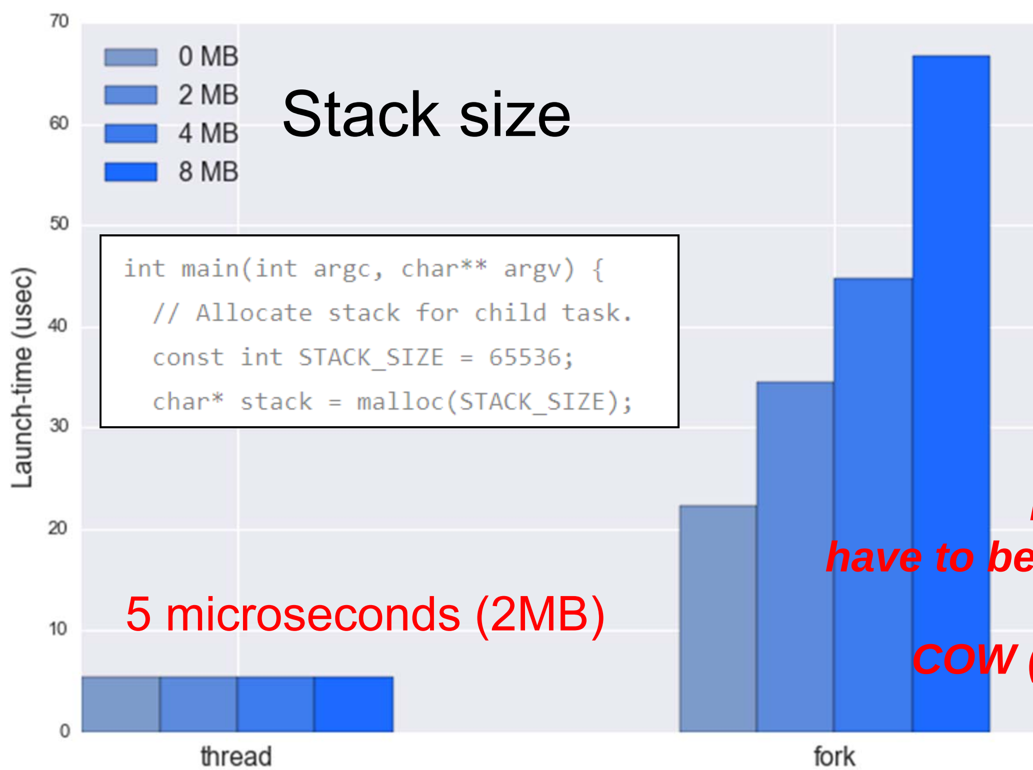
Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	**8.1**	0.1	2.9	**0.9**	**0.2**	**0.3**
Intel 2.8 GHz Xeon 5660 (12 cores/node)	**4.4**	0.4	4.3	**0.7**	**0.2**	**0.5**
AMD 2.3 GHz Opteron (16 cores/node)	**12.5**	1.0	12.5	**1.2**	**0.2**	**1.3**
AMD 2.4 GHz Opteron (8 cores/node)	**17.6**	2.2	15.7	**1.4**	**0.3**	**1.3**
INTEL 2.4 GHz Xeon (2 cpus/node)	**54.9**	1.5	**20.8**	**1.6**	**0.7**	**0.9**
INTEL 1.4 GHz Itanium2 (4 cpus/node)	**54.5**	1.1	**22.2**	**2.0**	**1.2**	**0.6**

Real: wall clock time, time from start to finish of the call.
User: CPU time spent in **user-mode code** (outside the kernel);
Sys: CPU time spent in the **kernel** (time spent in system calls)

Creating a thread using clone() is much faster!

```
char buf[100];
strcpy(buf, "hello from parent");
if (clone(child_func, stack + STACK_SIZE, flags | SIGCHLD, buf) == -1) {
  perror("clone");
  exit(1);
```

flags |= CLONE_VM




```
int main(int argc, char** argv) {
  // Allocate stack for child task.
  const int STACK_SIZE = 65536;
  char* stack = malloc(STACK_SIZE);
```

5 microseconds (2MB)

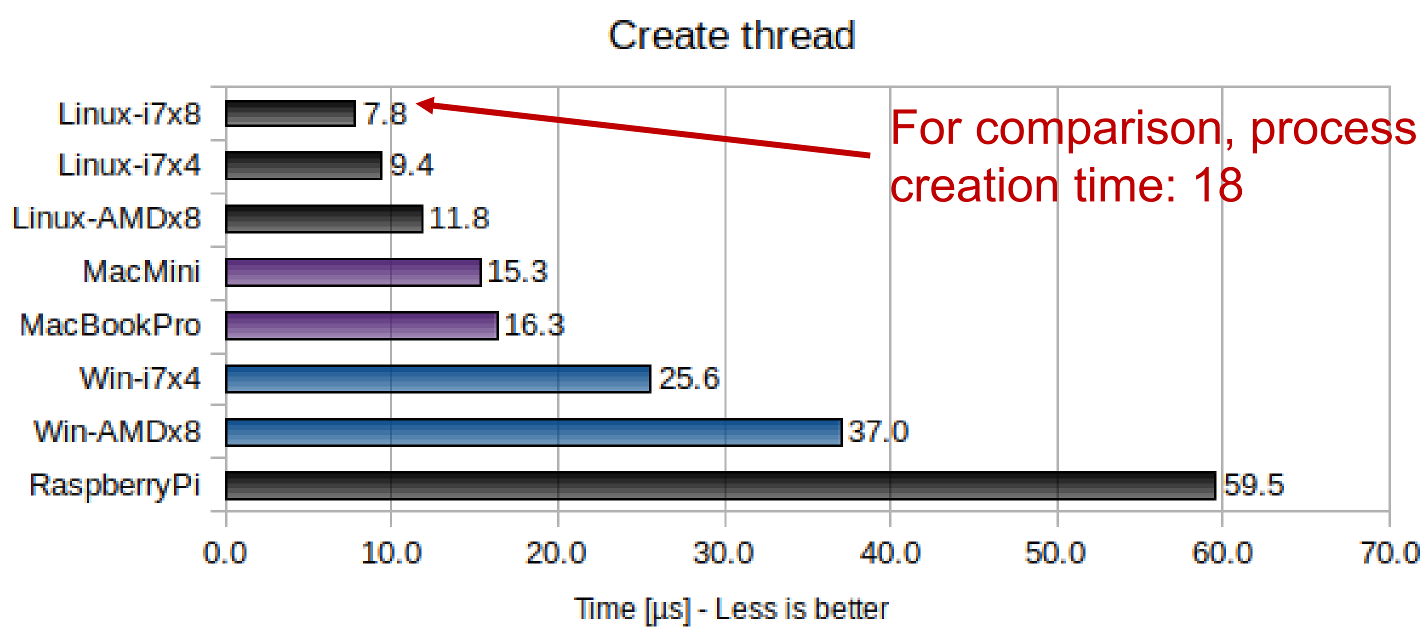
35 microseconds (2MB)

Page tables still have to be copied (tens of kilobytes) + COW (Copy On Write)

https://golangnews.org/2018/08/launching-linux-threads-and-processes-with-clone/

Thread Creation Time

- 100 threads are created. Each thread terminates immediately without doing any work, and the main thread waits for all child threads to terminate
  - **POSIX: pthread_create(), pthread_join()**
  - **WIN32: _beginthreadex(), WaitForSingleObject(), CloseHandle()**



https://www.bitsnbites.eu/benchmarking-os-primitives/

Process Creation Time

- This benchmark is almost identical to the previous benchmark. However, here 100 child processes are created and terminated (using **fork()** and **waitpid()**).