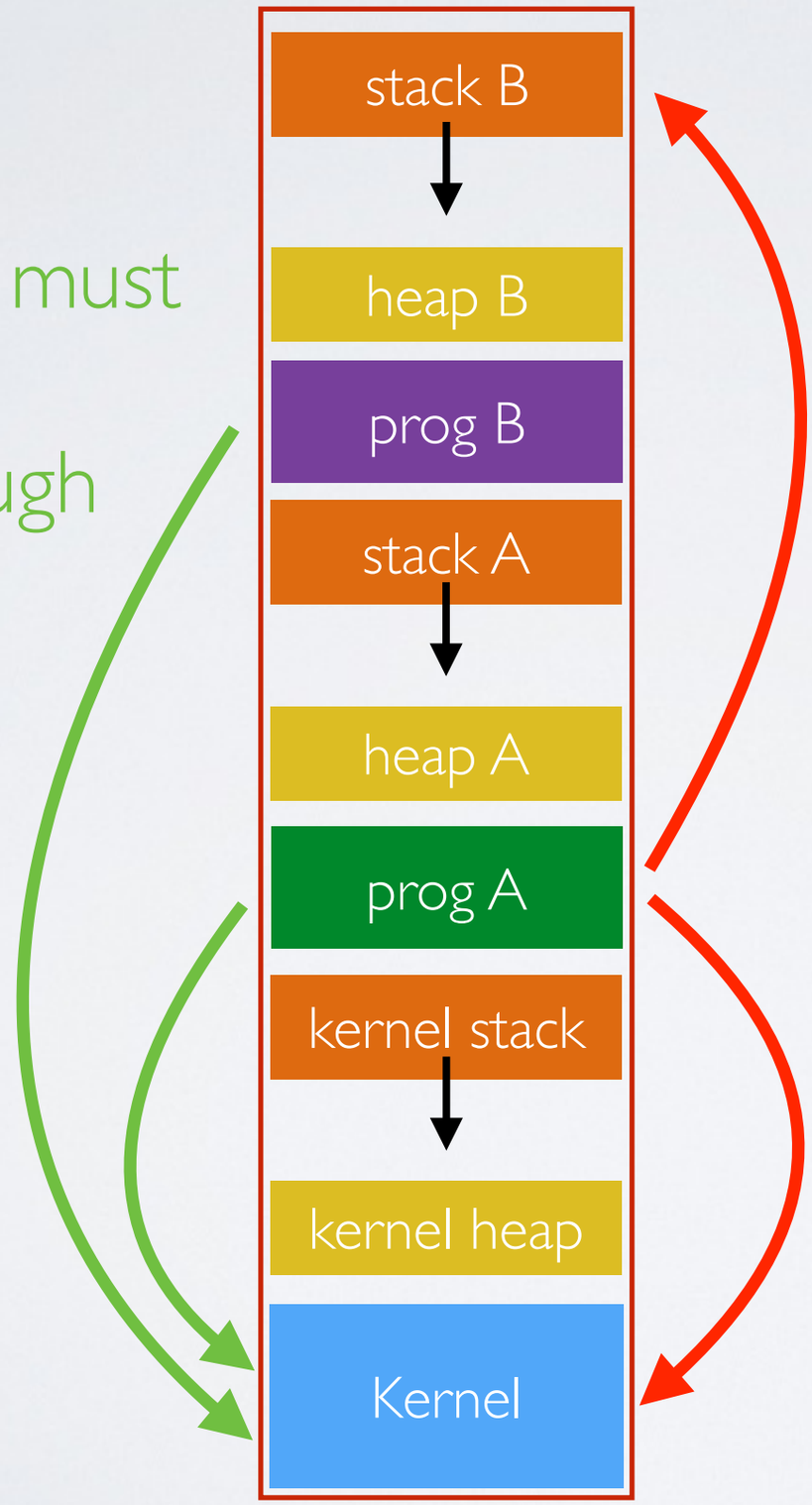
User Programs

Thierry Sans

The need for protection

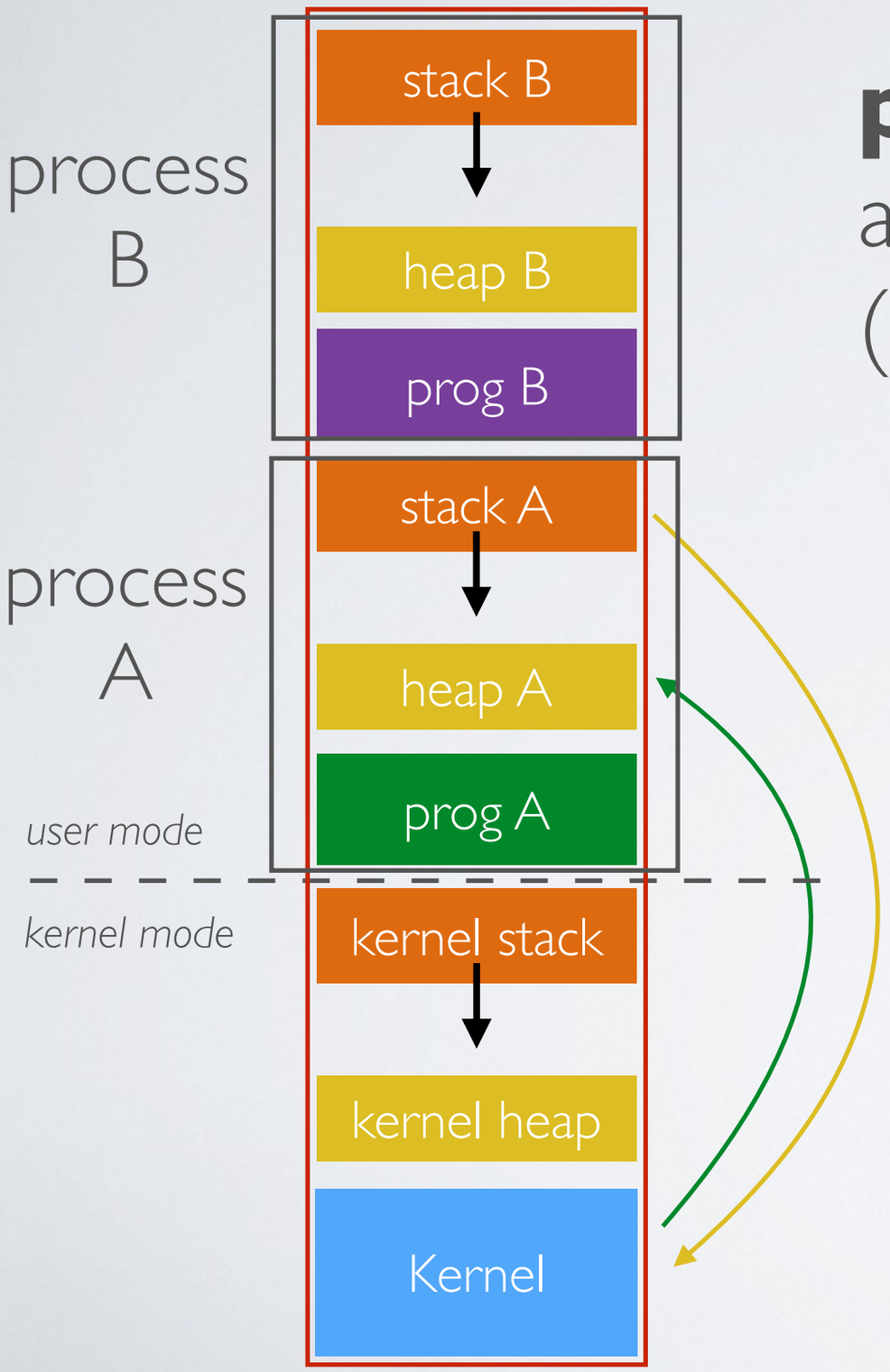
User programs must access shared resources through the kernel



User programs might interfere with each other if they share the same memory

User program can interfere with the kernel

Definition of the process and system calls



principle 1: user programs are run as processes isolated from each other (user mode)

principle 2: the kernel has privileged access to the entire memory (kernel mode)

principle 3: process can access resources through kernel system-calls

How can we isolate processes and kernel memory spaces?

➡ The Virtual Memory (coming soon)

In a nutshell

- User programs do not directly access the memory but the virtual memory (that is somehow mapped onto the real memory)
- The kernel manages the virtual memory for all processes

System Calls

The need for abstraction for accessing resources

How to write a user program like the *Bash* shell that reads keyboard inputs from the user?

➡ Read input data from the I/O device directly? But which one?

- The one connected to the PS2 port?
- The one connected to the USB?
- The one connected to the bluetooth?
- The remote one connected to the network?

How do you synchronize access with other programs using the keyboard as well?

◉ User programs do not operate I/O devices directly

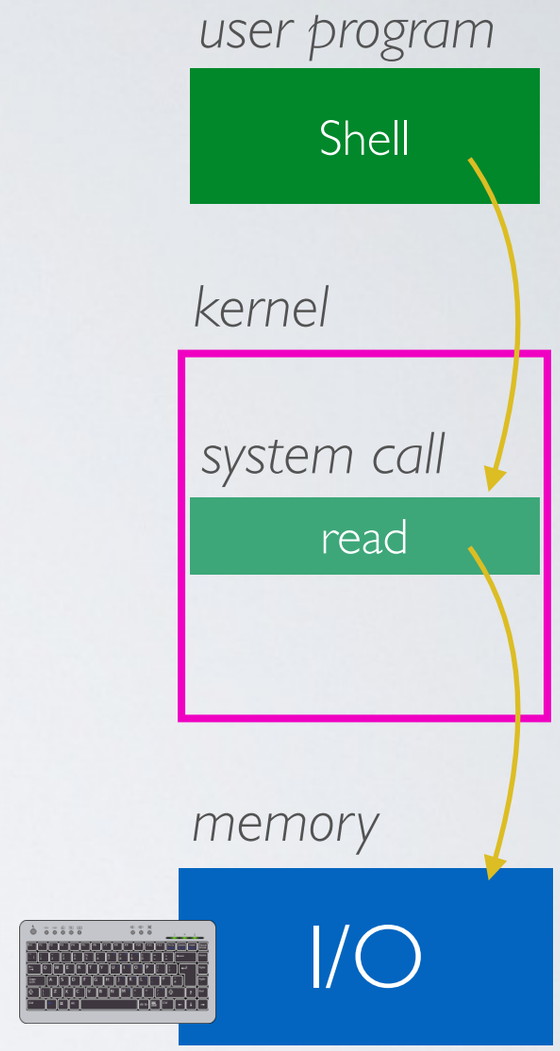
✓ The OS abstracts those functionalities and manage access through **system calls**

System Calls

➡ Provide user programs with an API to use the services of the operating system

There are 5 categories of system calls

- Process control
- File management
- Device management
- Information/maintenance (system configuration)
- Communication (IPC)
- Protection

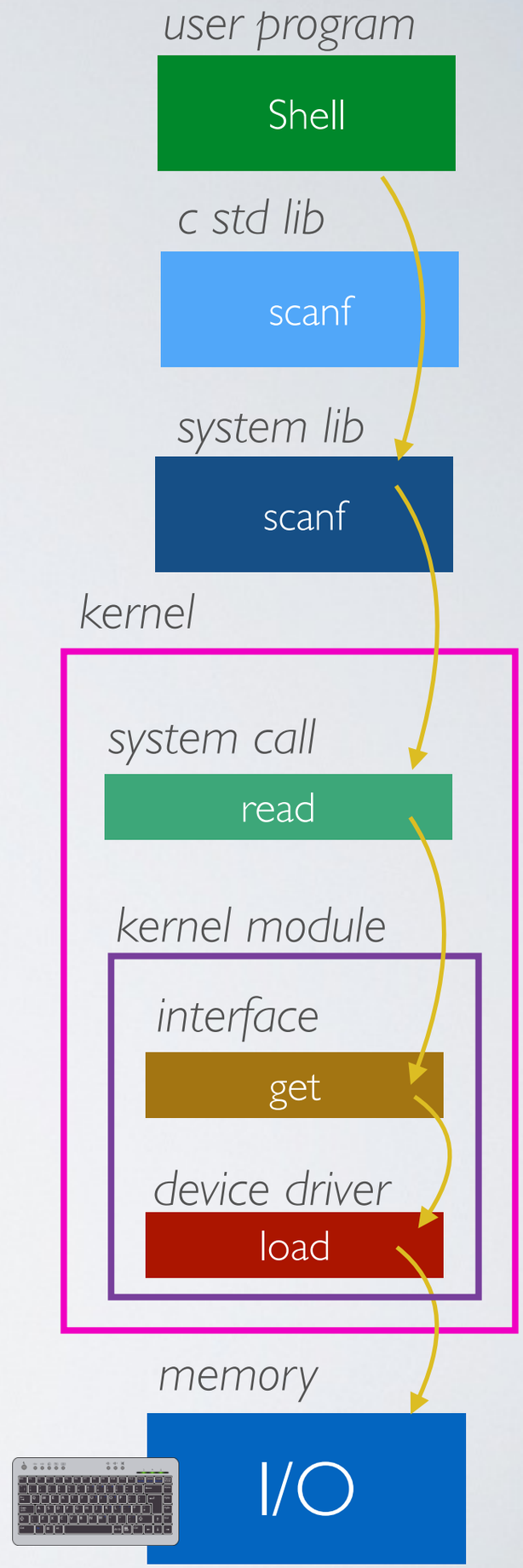


✓ There are 393 system calls on Linux 3.7

http://www.cheat-sheets.org/saved-copy/Linux_Syscall_quickref.pdf

In reality, many (many) level of abstraction and modularity

- This is what makes developing OS very challenging (CSCB07)



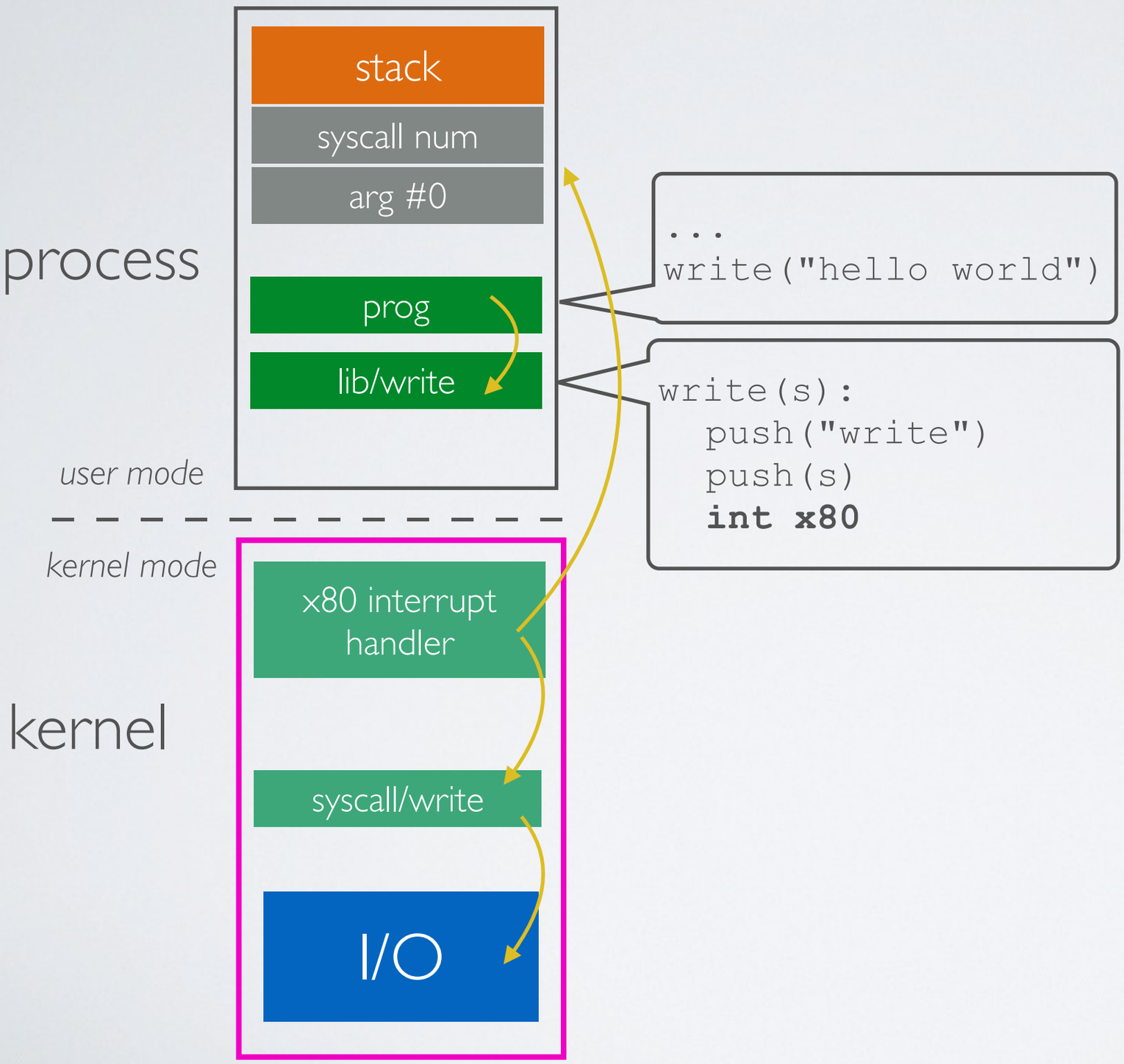
How to invoke system calls

The system calls look like some sort of "kernel API"

➡ Yes but how to invoke a system call like an API call if the process cannot access the kernel memory?

✓ Using software interrupts (a.k.a syscall trap)

Invoking a System Call



1) the program calls a library function

2) the library function pushes the syscall number and its arguments onto the stack and triggers a software interrupt

3) the interrupt handler reads the stack of the interrupted program to extract the system call number and the arguments

4) the interrupt handler calls the corresponding kernel system call function

5) the system call function executes and (possibly returns value by pushing them onto the stack of the interrupted program

Process

From the programmer's perspective

- Create and terminate
- Communicate
- Get information
- Control process (stop and resume)

Process Control Block

PCB (Process Control Block) - data structure to record process information

- Pid (process id) and ppid (parent process)
- (optional) User
- Address space (forthcoming lecture on memory management)
- Open files (coming next with filesystem)
- Others

Create a process

- A process is created by another process
  (concept of parent process and child process)

- The kernel creates the root parent as part of the booting
  e.g shell program for a simple OS
  e.g Window Manager for a GUI OS

Process creation on Unix using `fork`

```
int fork()
```

1. Creates and initializes a new PCB
2. Creates a new address space
3. Initializes the address space with a copy of the entire contents of the address space of the parent (with one exception)
4. Initializes the kernel resources to point to the resources used by parent (e.g., open files)
5. Create a kernel thread associated with this process and place that thread onto the ready queue

Why `fork` and `exec`?

`fork` is very useful when the child…

- is cooperating with the parent
- relies upon the parent's data to accomplish its task

➡ Simple interface

Example : a web server

```
while (1) {
   int sock = accept();
   if ((child_pid = fork()) == 0) {
      // Handle client request
   } else {
      // Close socket
   }
}
```

Process creation on Unix using `exec`

```
int exec(char *prog, char *argv[])
```

1. Stops the current process
2. Loads the program "prog" into the process' address space
3. Initializes hardware context and args for the new program
4. Places the PCB onto the ready queue

➡ **Actually, `exec` does not create a new process**

Spawning

- ✓ Most calls to `fork` are followed by `exec` (a.k.a `spawn`)
  - `minish.sh`
  - `redirsh.c`
  - `pipesh.c`

Argument against `fork`

"A fork() in the road"

Andrew Baumann (Microsoft Research), Jonathan Appavoo, Orran Krieger (Boston University), Timothy Roscoe (ETH Zurich) - *In Proceedings of HotOS 2019*

https://www.microsoft.com/en-us/research/uploads/prod/2019/04/fork-hotos19.pdf

➡ The main argument is security

Process creation on Windows

```
CreateProcess: BOOL CreateProcess(char *prog, char *args)
```

1. Creates and initializes a new PCB
2. Creates and initializes a new address space
3. Loads the program specified by "prog" into the address space
4. Copies "args" into memory allocated in address space
5. Initializes the saved hardware context to start execution at main (or wherever specified in the file)
6. Places the PCB on the ready queue

Wait for a process

Unix:`wait(int *wstatus)`

Windows:`WaitForSingleObject`

Terminate a process

Unix: `exit(int status)`

Windows: `ExitProcess(int status)`

➡ The OS will cleanup after the process:

- Terminates all threads (coming next)
- Closes open files, network connections
- Frees allocated memory (and VM pages out on disk)
- Removes PCB from kernel data structures

User thread

The cost of multi-processing

```
while (1) {
  int sock = accept();
  if ((child_pid = fork()) == 0) {
    // Handle client request
  } else {
    // Close socket
  }
}
```

Recall our Web Server example
we need to fork a child process for each request

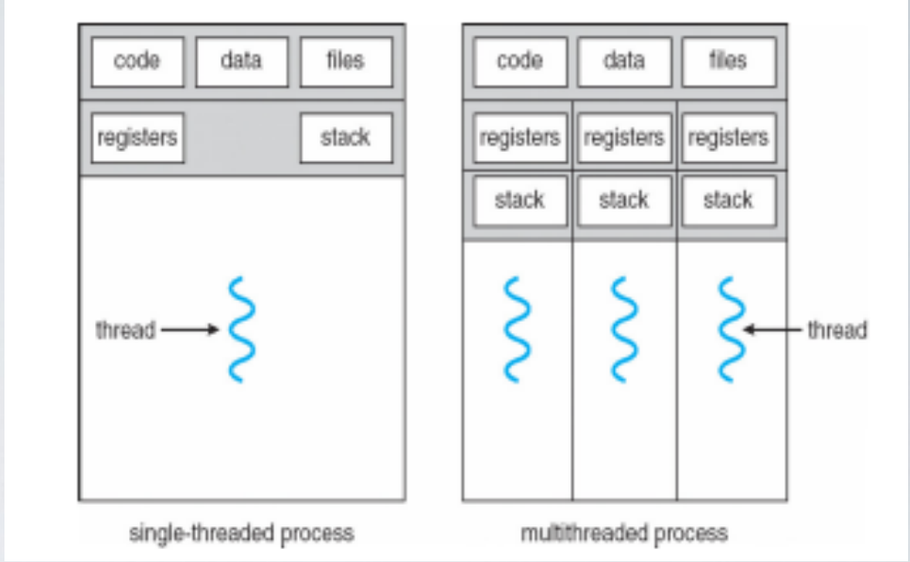
- Create a new PCB
- Copy the address space and the resources
- Have the OS execute this child process
  (switching process involves remapping the virtual memory)
- Use signals and pipes if the child wants to send information back to the parent process

A good but costly abstraction

✓ Good to avoid processes interfering with each other but ...

- Creating a process is costly (space and time)
- Context switching is costly (time)
- Inter-process communication is costly (time)

User Threads

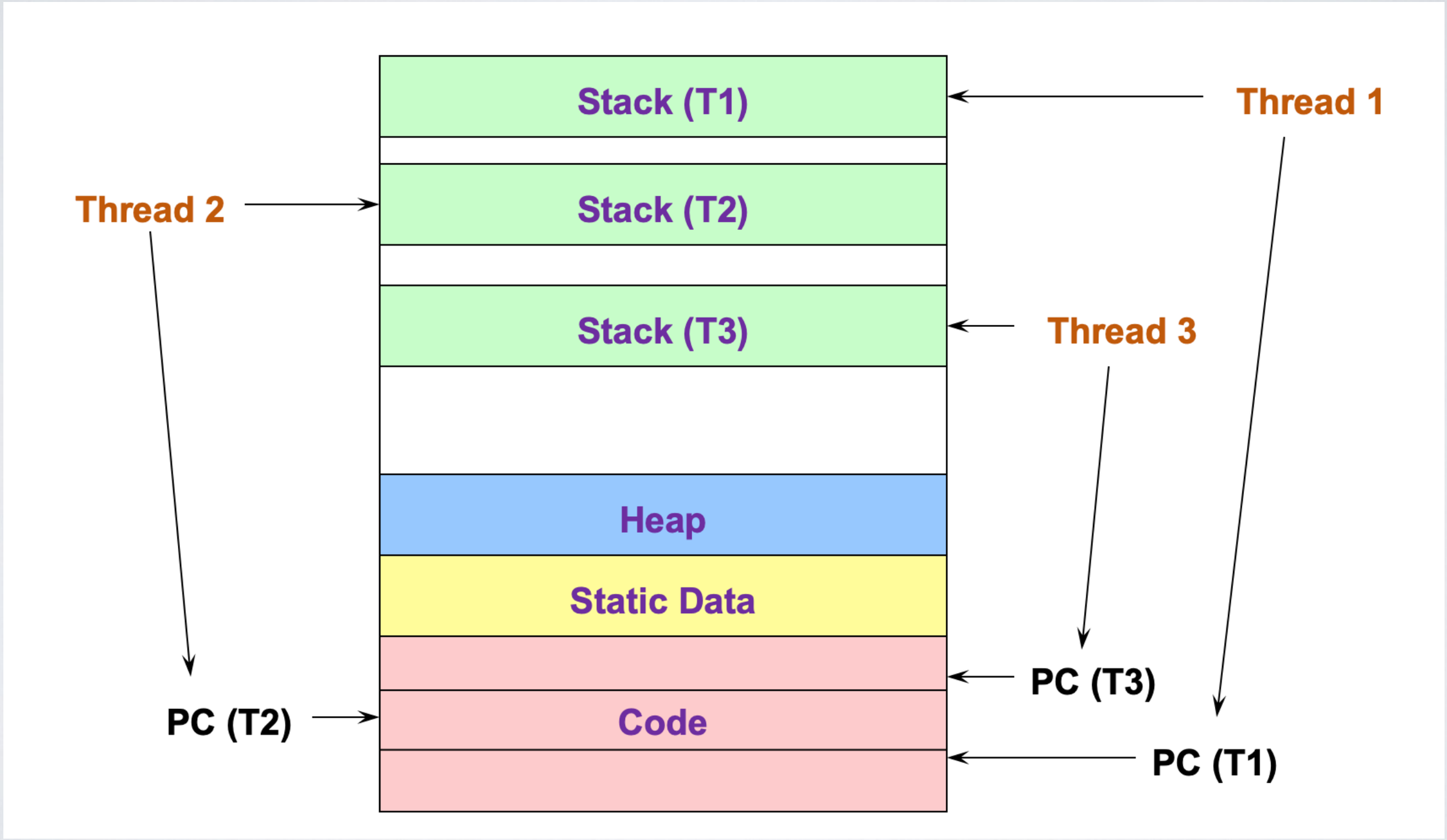


Modern OSes separate the concepts of processes and threads

- The thread defines a sequential execution stream within a process (PC, SP, registers)
- The process defines the address space and general process attributes (everything but threads of execution)

✓ A thread is bound to a single process but a process can have multiple threads

Threads within a process



Our web server becomes

```
web_server() {
    while (1) {
        int sock = accept();
        thread_fork(handle_request, sock);
    }
}

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

Benefits

- **Responsiveness**
  an application can continue running while it waits for some events in the background

- **Resource sharing**
  threads can collaborate by reading and writing the same data in memory (instead of asking the OS to pass data around)

- **Economy of time and space**
  no need to create a new PCB and switch the entire context (only the registers and the stack)

- **Scalability in multi-processor architecture**
  the same application can run on multiple cores

Multithreading Model

Disambiguation

- **Process**
  running instance of a program

User space

- **User thread**
  user-defined concurrency within a process

- -

- **Kernel thread**
  the unit of scheduling

Kernel space

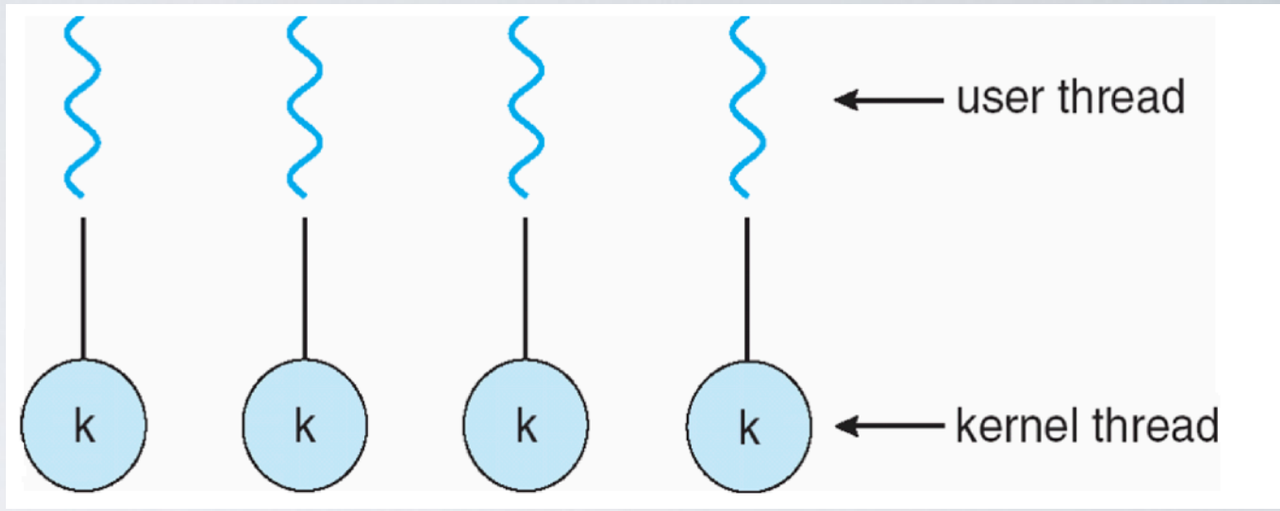
Multithreading models

- One-to-one model
  Kernel-level threads (a.k.a native threads)

- Many-to-one model
  User-level threads (a.k.a green threads)

- Many-to-many model
  Hybrid threads (a.k.a n:m threading)

One-to-one model
Kernel-level threads
(a.k.a native threads)



The kernel manage and schedule threads

- e.g Windows threads
- e.g POSIX pthreads `PTHREAD_SCOPE_SYSTEM`
- e.g (new) Solaris lightweight processes (LWP)

➡ All thread operations are managed by the kernel

✓ good for scheduling

✓ bad for speed

POSIX Thread API

- Create a new thread, run fn with arg

```
tid thread_create (void (*fn) (void *), void *);
```

  - Allocate Thread Control Block (TCB)
  - Allocate stack
  - Put func, args on stack
  - Put thread on ready list

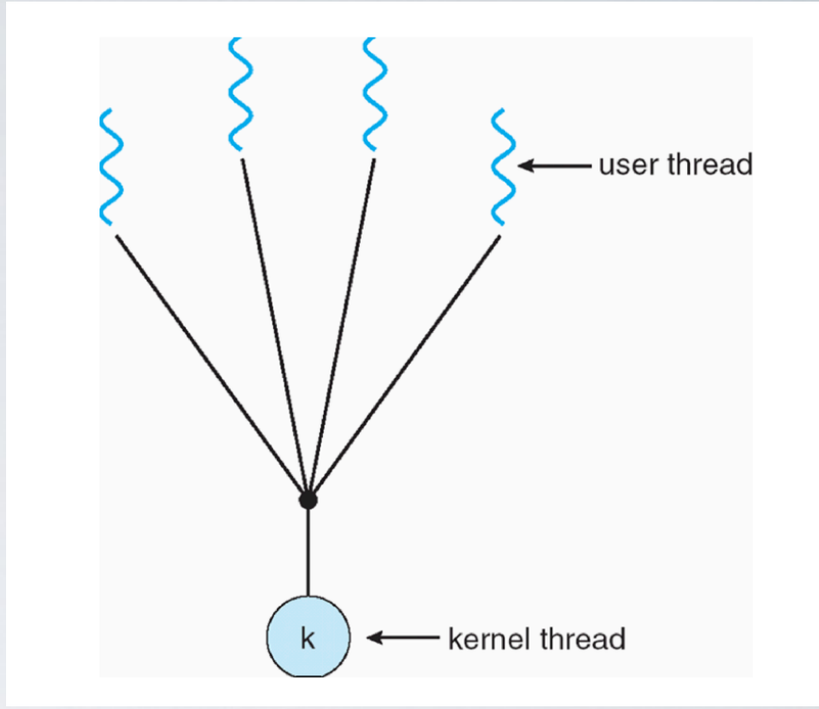
- Destroy current thread

```
void thread_exit ();
```

- Wait for thread to exit

```
void thread_join (tid thread);
```

Many-to-one model
User-level threads
(a.k.a green threads)



One kernel thread per process
thread management and scheduling is delegated to a library

- e.g pthreads `PTHREAD_SCOPE_PROCESS`
- e.g Java threads

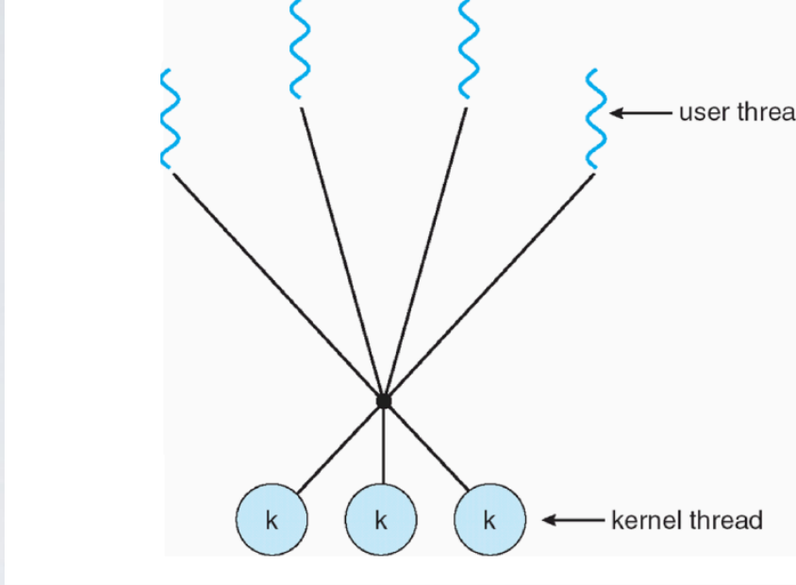
➡ The kernel is not involved

✓ Very lightweight and fast

✓ All threads can be blocked if one of them is waiting or an event

✓ Cannot be scheduled on multiple cores

Many-to-many model
Hybrid threads
(a.k.a n:m threading)



User threads implemented on kernel threads

- e.g (old) Solaris

➡ Multiple kernel-level threads per process