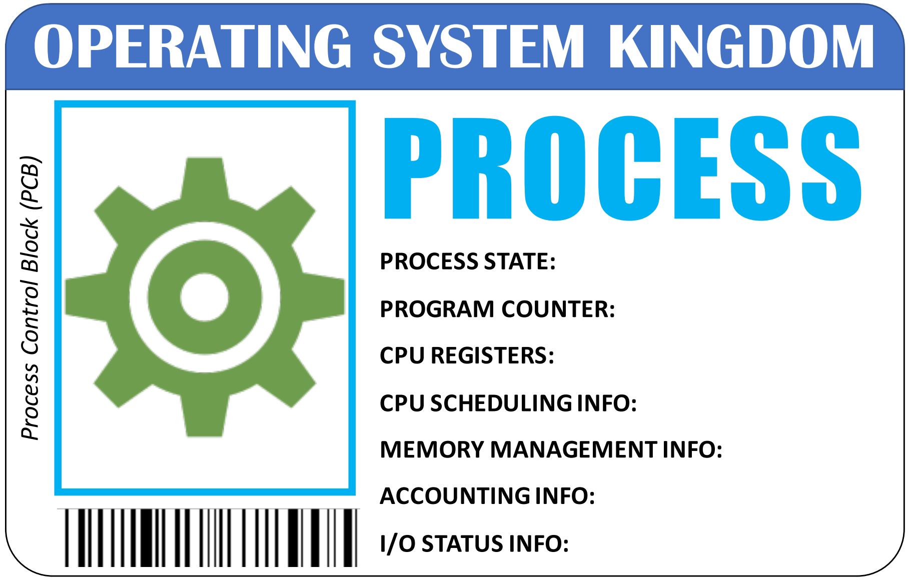
CPE 460 Operating System Design

Chapter 6: A Thread Story

Ahmed Tamrawi

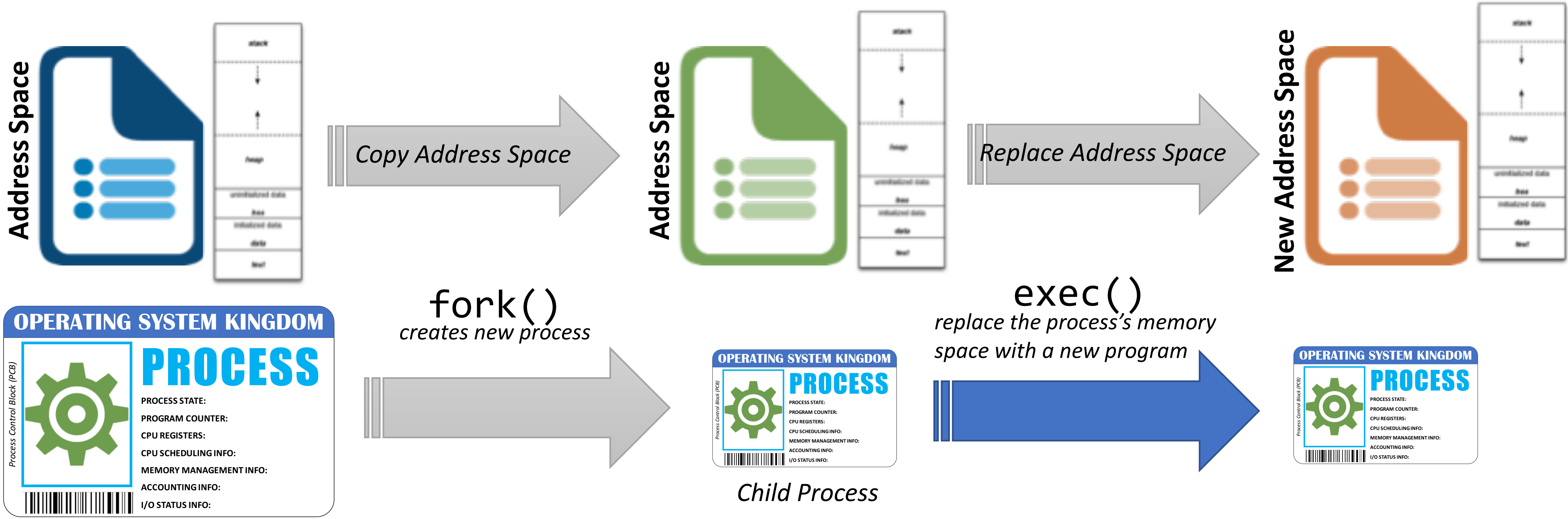
Any program to run **must** be loaded in memory


جمهورية مصر العربية
بطاقة تحقيق الشخصية
كرستيانو
عبد الستار محمود على رونالدو
قرية الوسطانيه
كفر الدوار - البحيره
٢ ٦١ ٠٦ ١٦ ٢١ ٠٠٨٧٥
AM3784462


OPERATING SYSTEM KINGDOM
Process Control Block (PCB)
PROCESS
PROCESS STATE:
PROGRAM COUNTER:
CPU REGISTERS:
CPU SCHEDULING INFO:
MEMORY MANAGEMENT INFO:
ACCOUNTING INFO:
I/O STATUS INFO:

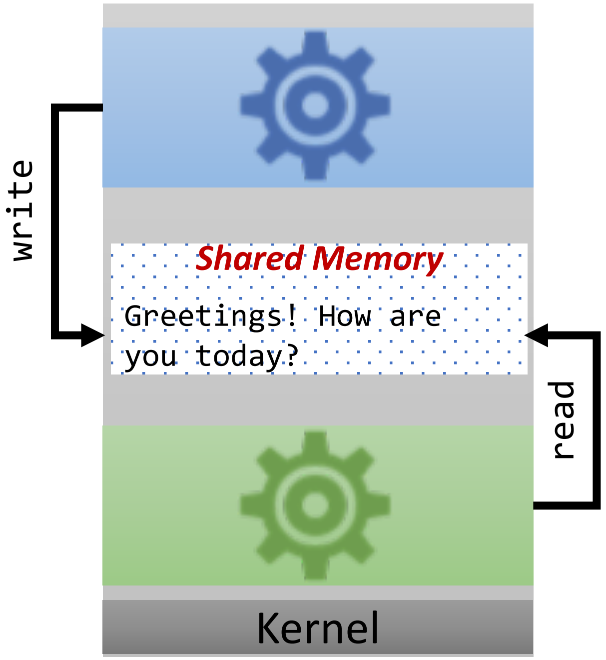
Process Creation

Each process has its own copy of address space



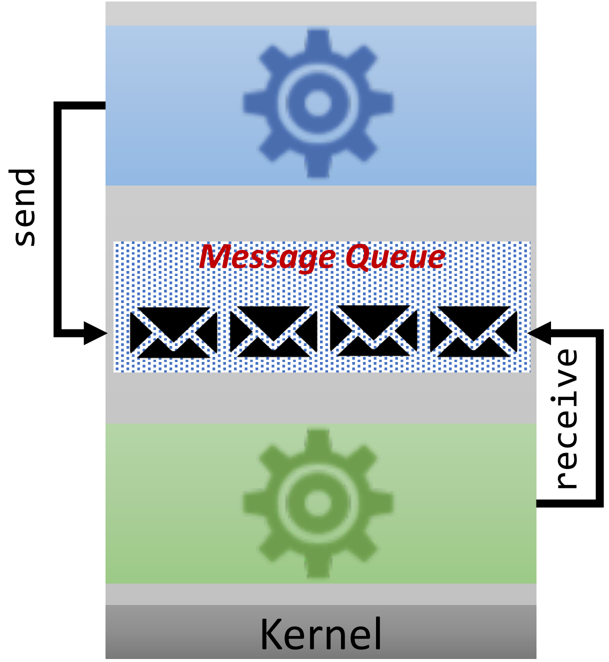
Cooperating processes need interprocess communication (IPC)

The operating system provides multiple mechanisms that allow processes to exchange data and information



Speed

Many Implementations

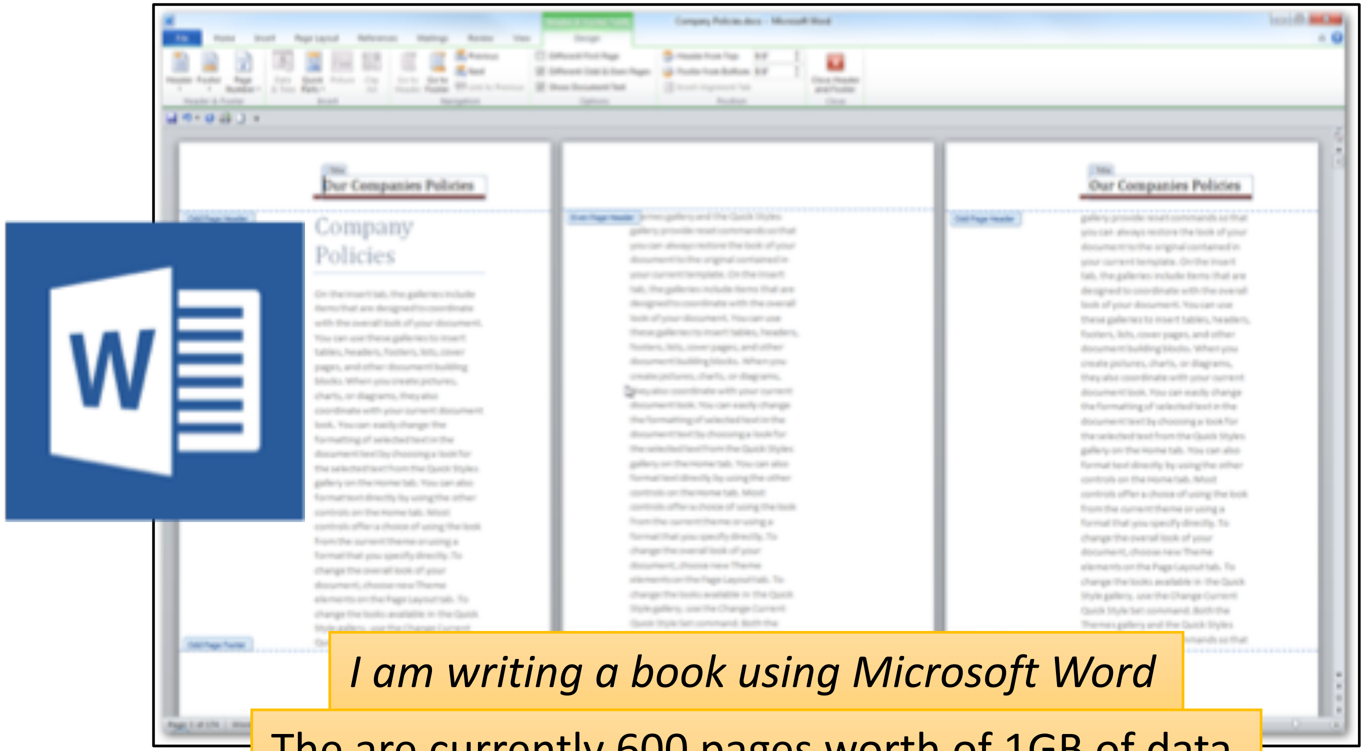


Shared Memory

A region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region

Message Passing

Communication takes place by means of messages exchanged between the cooperating processes

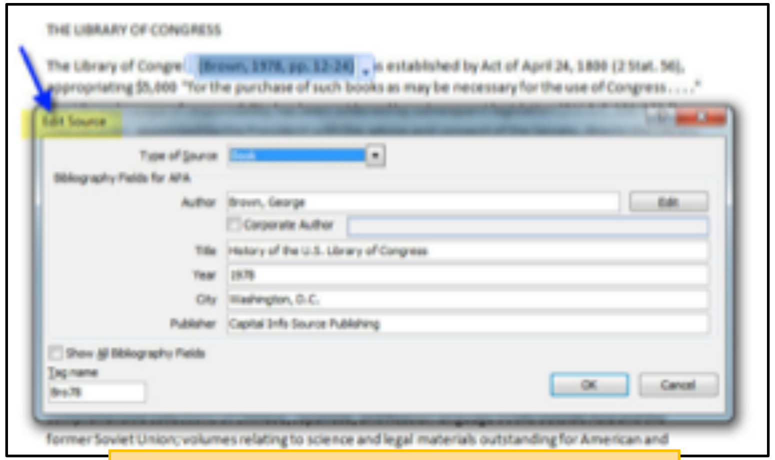


I am writing a book using Microsoft Word

The are currently 600 pages worth of 1GB of data

Spell and Grammar Check the 600 pages

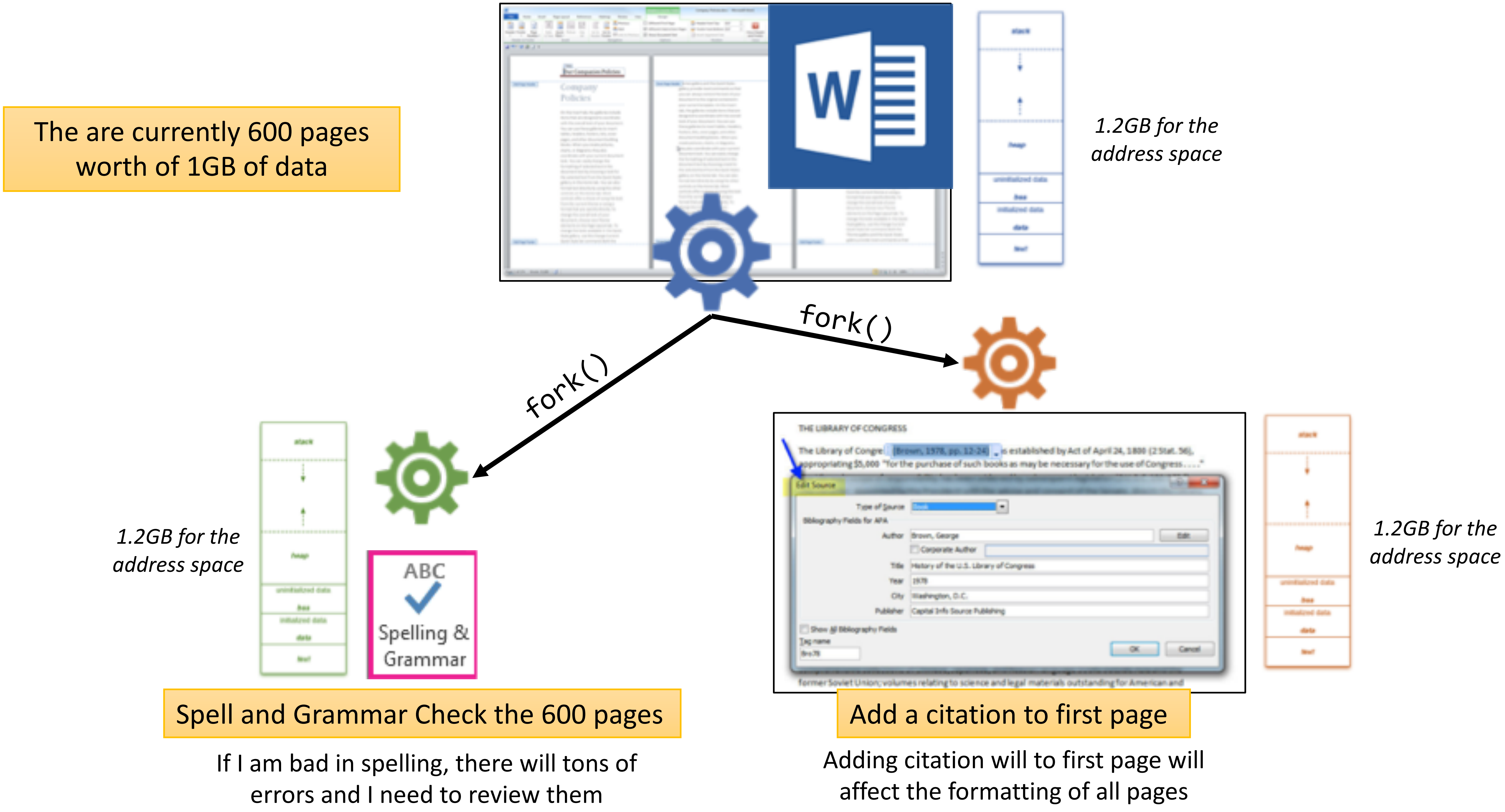
If I am bad in spelling, there will be tons of errors and I need to review them

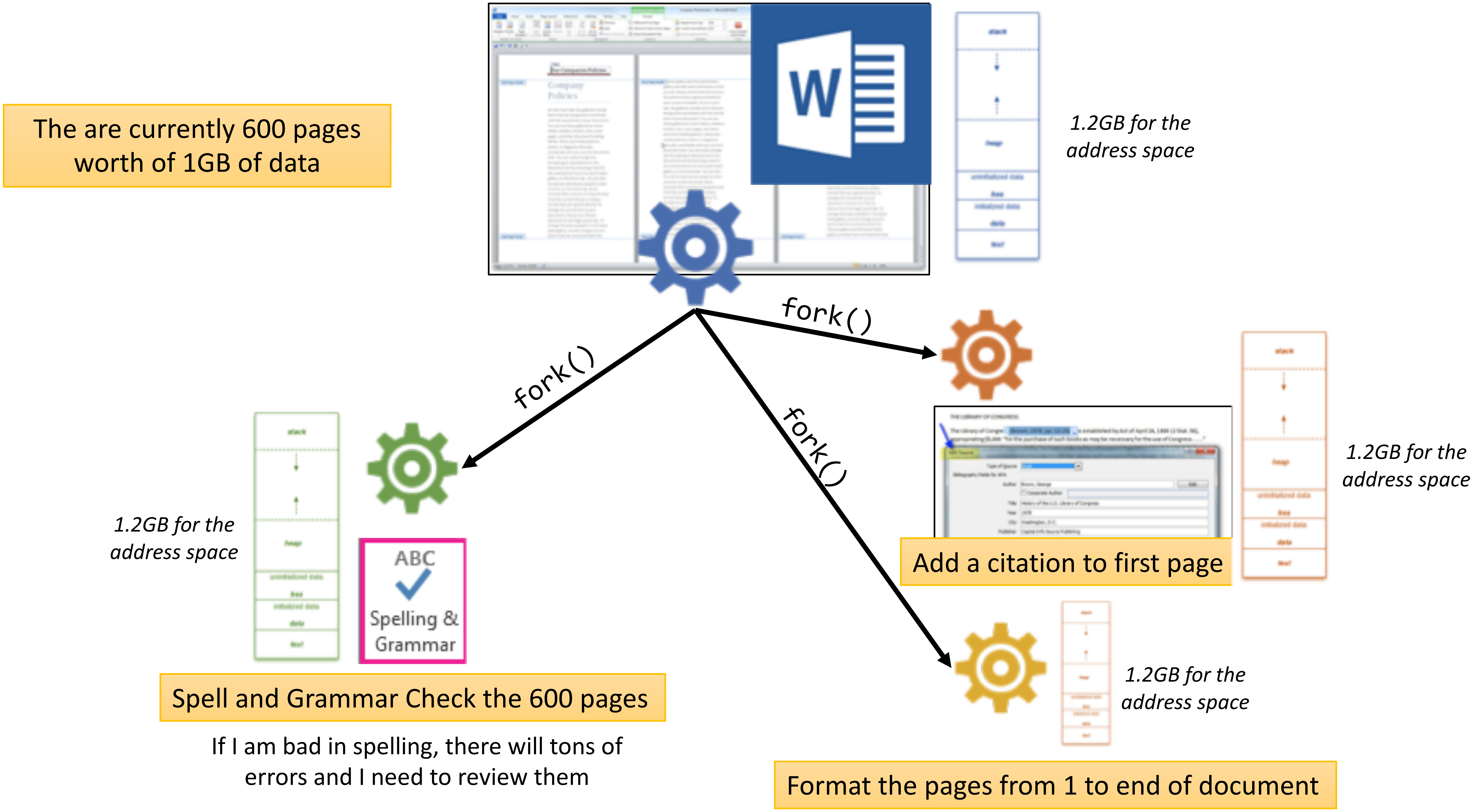


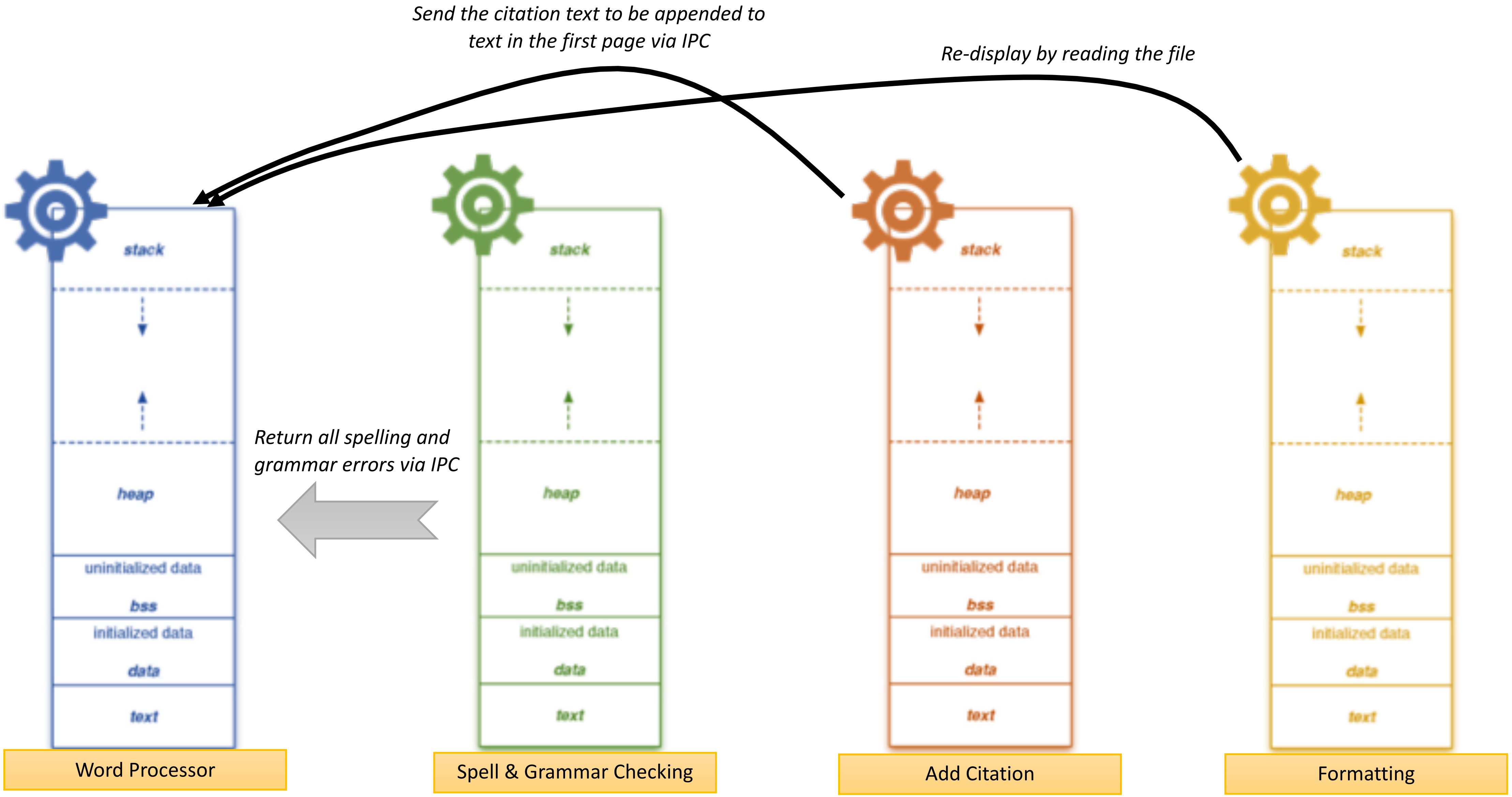
Add a citation to first page

Adding citation will to first page will affect the formatting of all pages


شو الحل يا خال


The are currently 600 pages worth of 1GB of data
1.2GB for the address space
fork()
fork()
1.2GB for the address space
ABC
Spelling & Grammar
1.2GB for the address space
Spell and Grammar Check the 600 pages
If I am bad in spelling, there will tons of errors and I need to review them
Add a citation to first page
Adding citation will to first page will affect the formatting of all pages

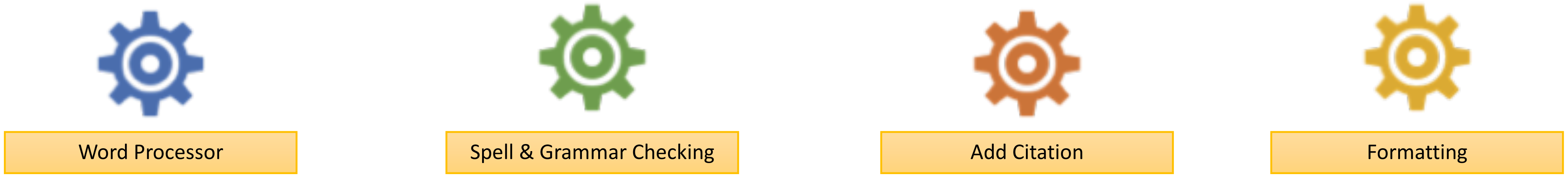

The are currently 600 pages worth of 1GB of data
1.2GB for the address space
fork()
fork()
fork()
1.2GB for the address space
Add a citation to first page
1.2GB for the address space
ABC
Spelling & Grammar
Spell and Grammar Check the 600 pages
If I am bad in spelling, there will tons of errors and I need to review them
1.2GB for the address space
Format the pages from 1 to end of document



Send the citation text to be appended to text in the first page via IPC
Re-display by reading the file
Return all spelling and grammar errors via IPC
stack
heap
uninitialized data
bss
initialized data
data
text
Word Processor
Spell & Grammar Checking
Add Citation
Formatting

This is not a feasible solution

It leads to substantial delays leading to unhappy user



Recall that there is a limit on the amount of data that can be communicated via IPC

IPC is, in general, expensive due to the need for system calls

Context switching among large processes is costly and time consuming

Forking and copying address spaces is time and space consuming

We need a way to share the same address space between all cooperating processes so we can work on parallel

We need a way to share the same address space between all cooperating processes so we can work on parallel

Word Processor

Spell & Grammar Checking

Add Citation

Formatting

What is similar in these cooperating processes?

Share the same privileges

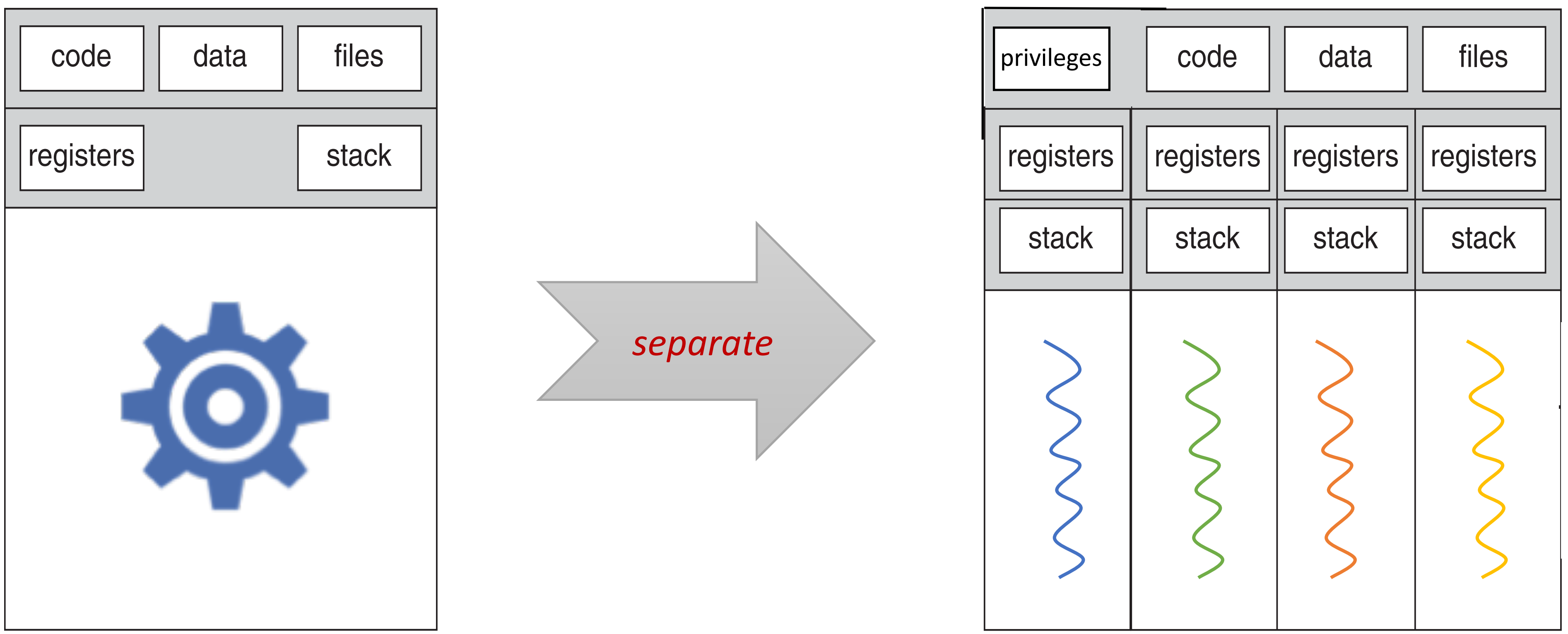
Share the same code and data (address space)

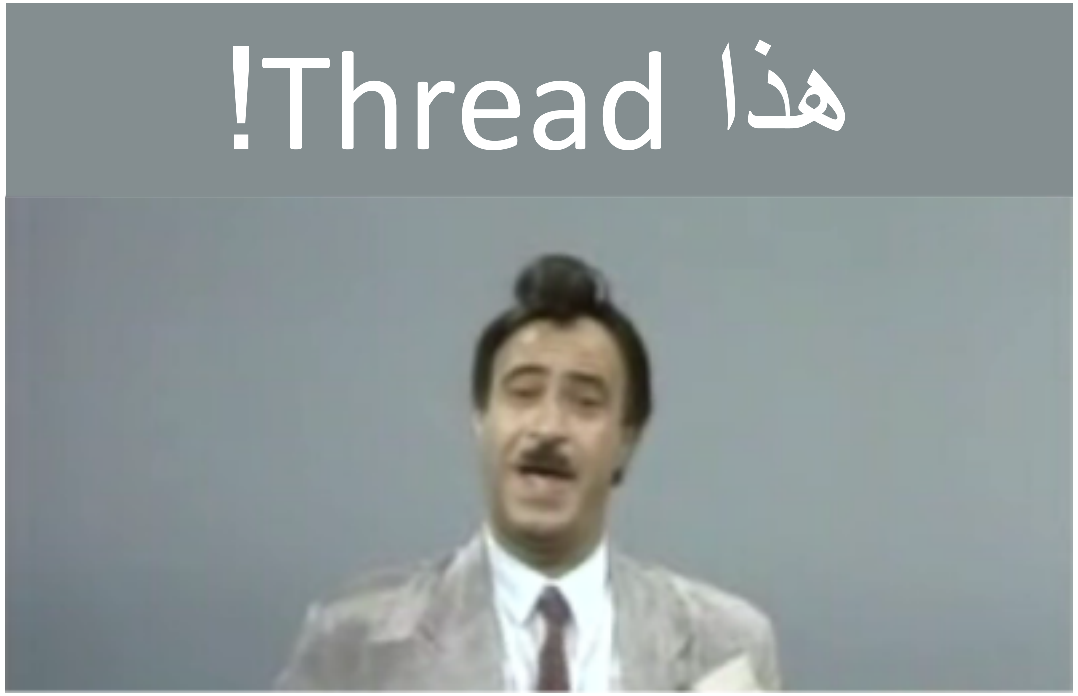
Share the same resources

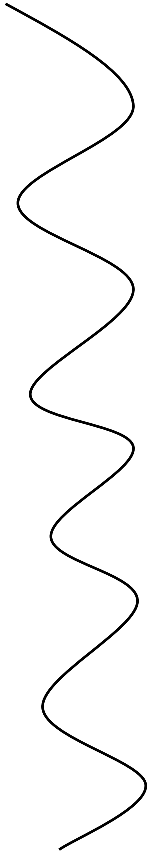
What is not shared among them?

Each has its own execution state: PC, SP, and registers

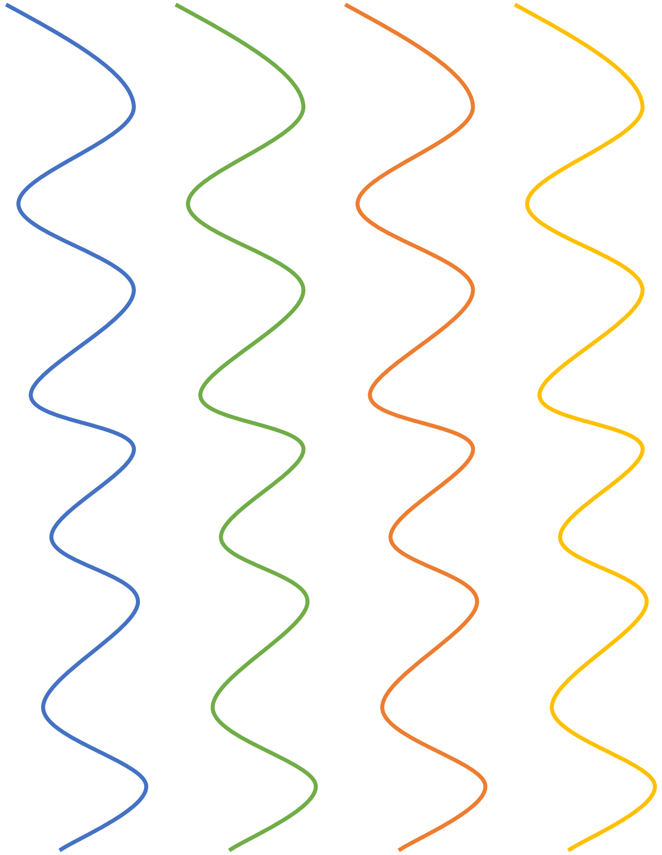
Why don't we *separate* the concept of a **process** from its **execution state**?

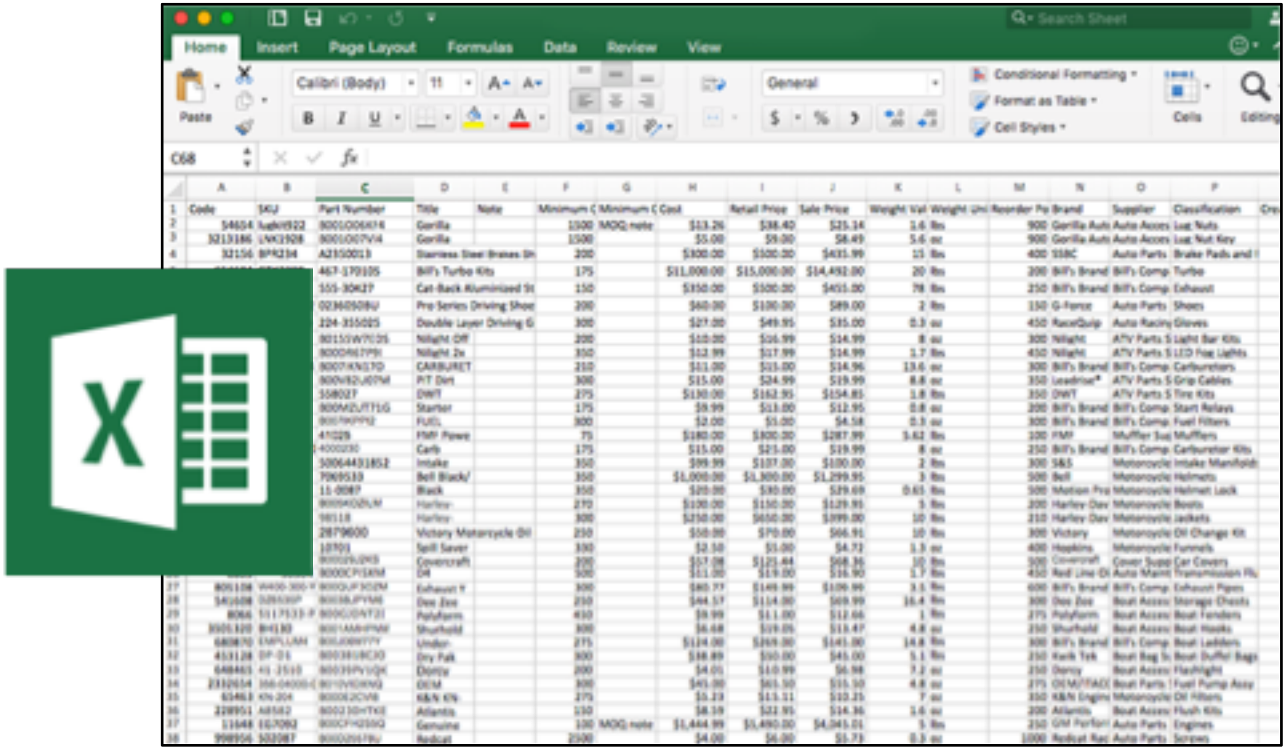


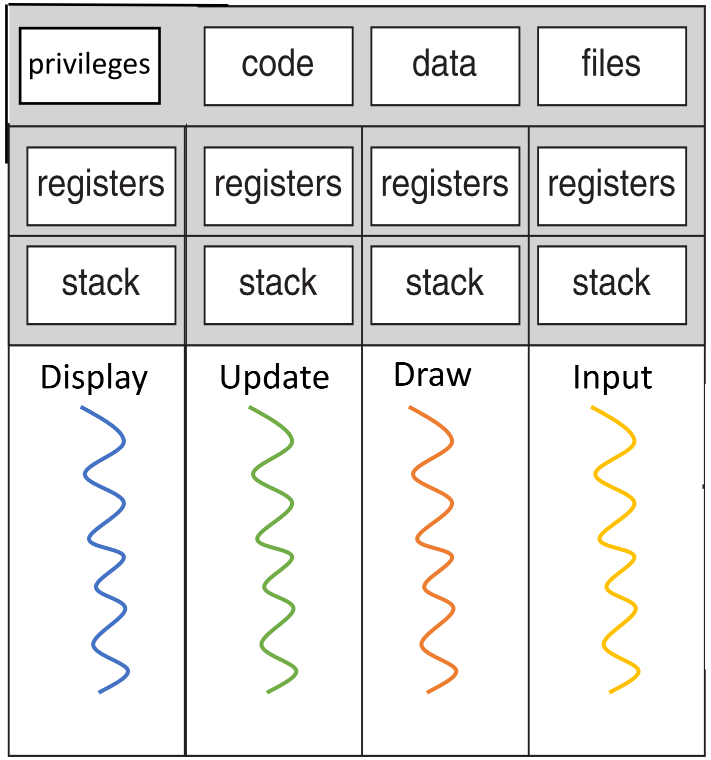

هذا Thread!



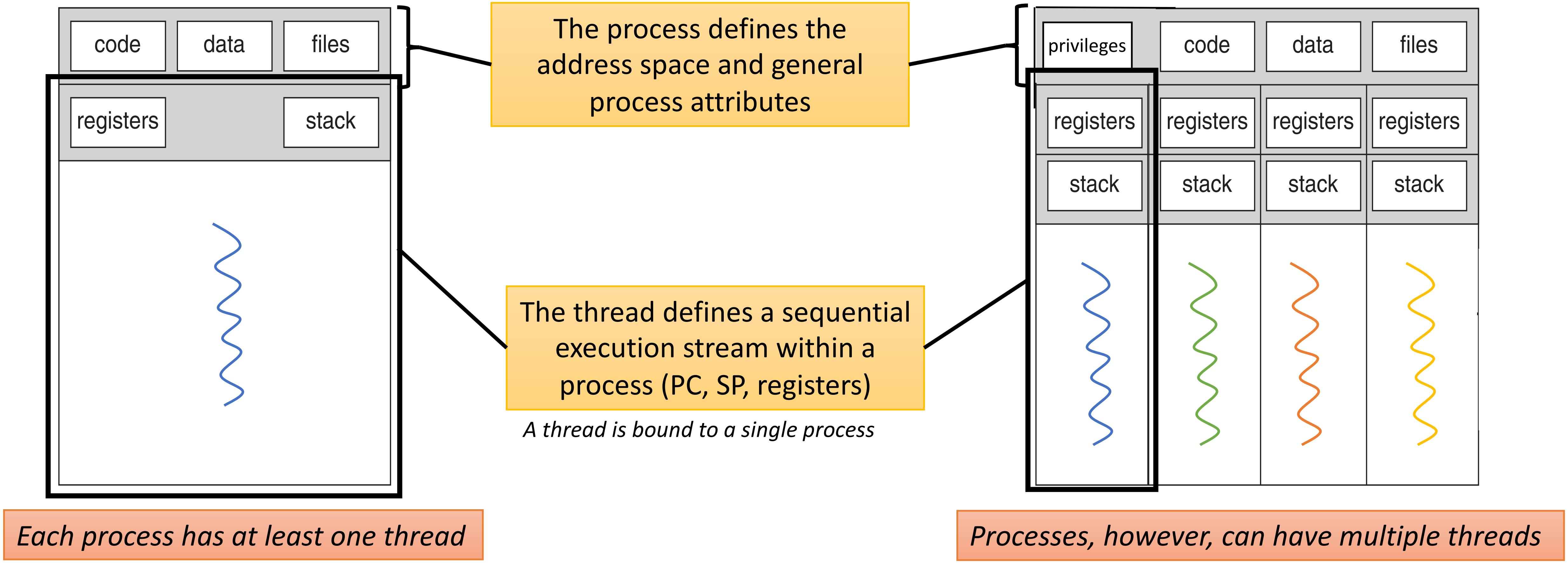

هذه Threads!



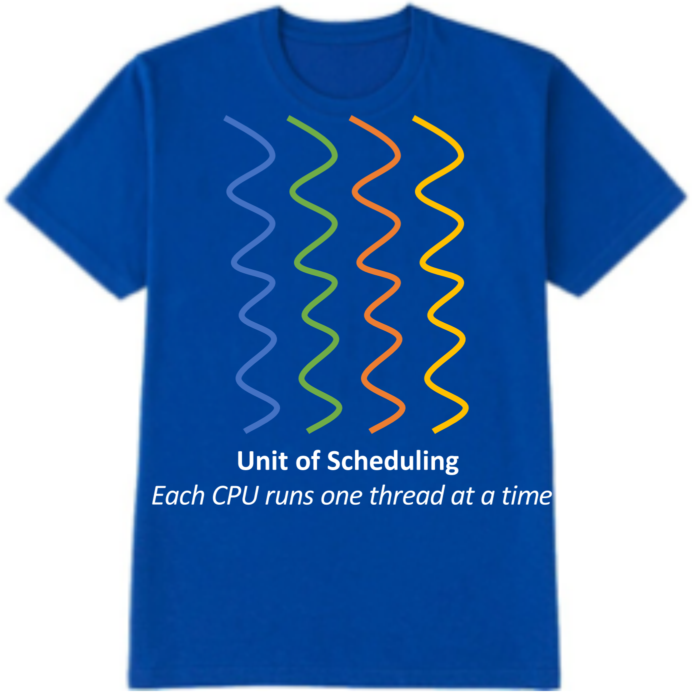

Home
Insert
Page Layout
Formulas
Data
Review
View


privileges
code
data
files
registers
registers
registers
registers
stack
stack
stack
stack
Display
Update
Draw
Input

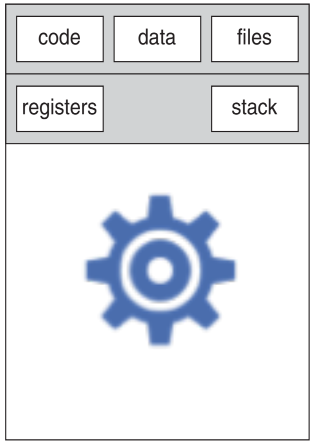
Modern OSes separate the concepts of **processes** and **threads**



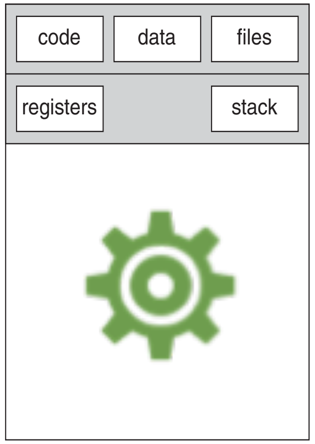

فلـــة شـمعـة منـــــــورة


Unit of Scheduling
Each CPU runs one thread at a time

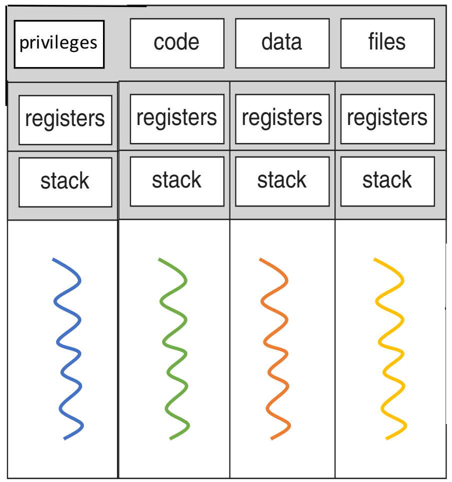

Processes are just fat, We are lightweight!



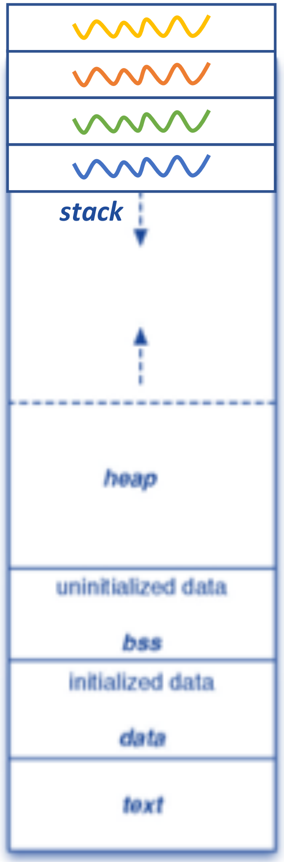
Process



Child Process



Multithreaded Process



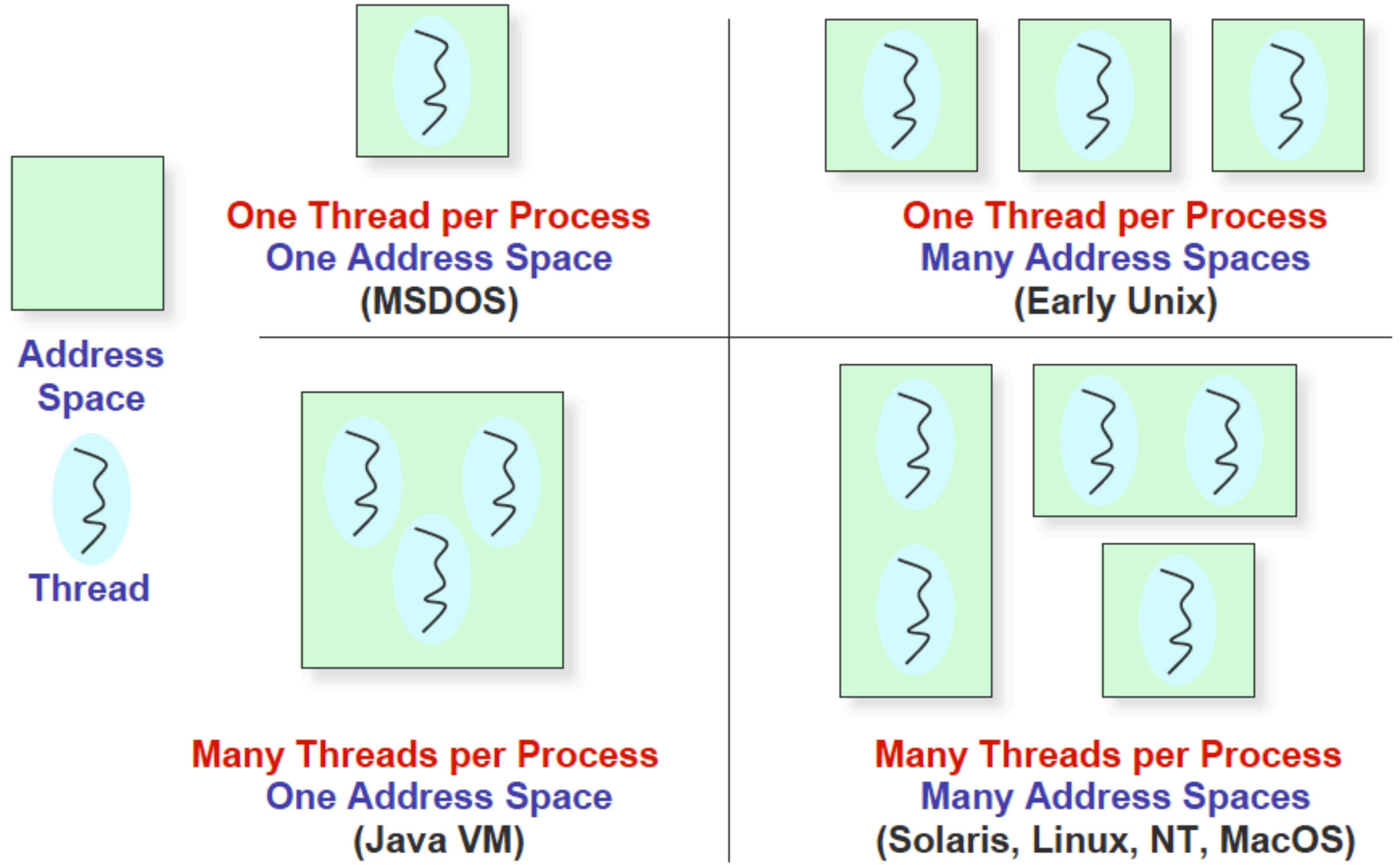
Processes

VS

Threads

Processes	Threads
Process is called *heavyweight* process	Threads are called *lightweight* processes
Process switching needs interface with the OS	Threads switching is handled by the **threading library** and does **not** need to notify the OS or cause any interrupts
Multiple processes can execute the same code but each one has its own **address space**	All threads **share the same address space** and see the same code
If the process is blocked, nothing is executed until the process is unblocked	While one thread within a process is blocked and waiting, other threads in the same task can run
Multiple redundant processes uses more resources than multiple threads	Multiple threaded processes uses fewer resources than multiple redundant process
In multiple process, each process separates independently of the others	One thread can read, write or even completely wipe out another threads stack

Thread Design Space



Why use thread?

Threads are economical

cheaper than process creation, thread switching lower overhead than context switching

Per process items
Address space
Global variables
Open files
Child processes
Pending alarms
Signals and signal handlers
Accounting information

Per thread items
Program counter
Registers
Stack
State

Threads enable Scalability

Process can take advantage of multiprocessor architectures

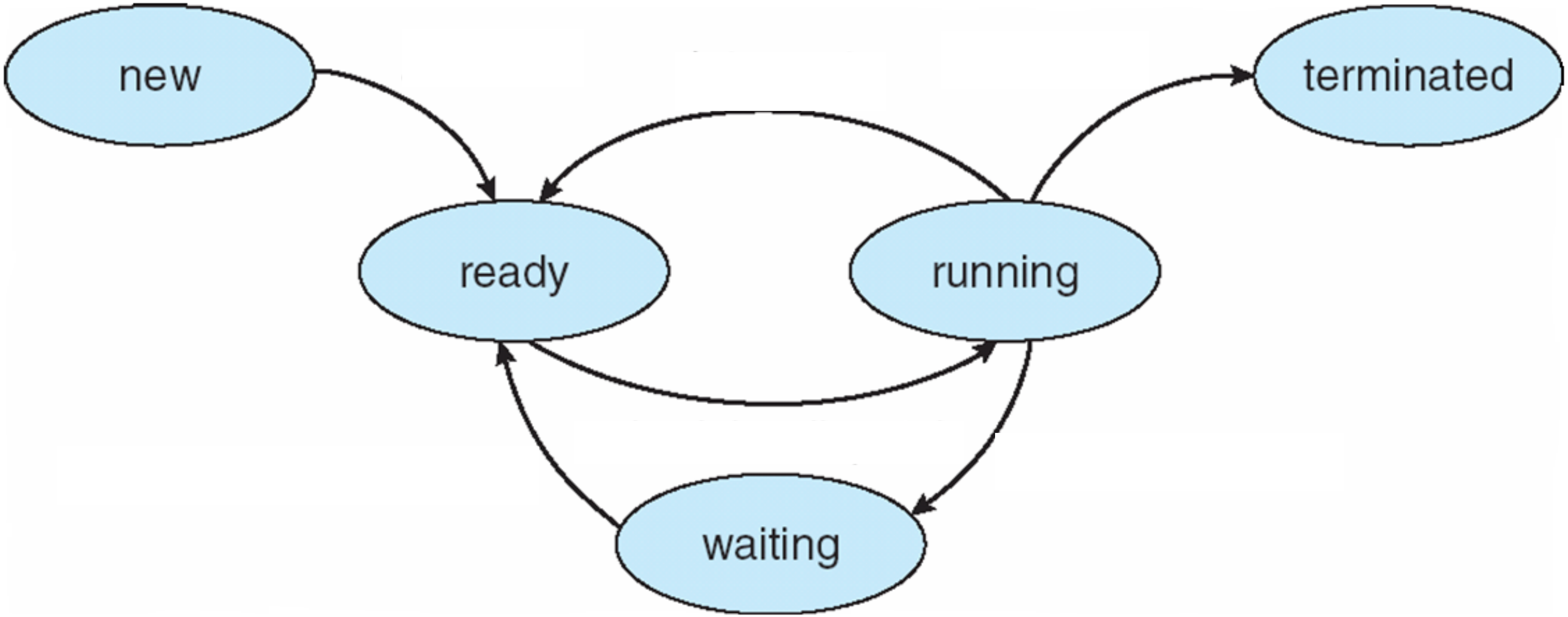
Threads share Resources

Threads share resources of process, easier than shared memory or message passing

Threads enables Responsiveness

Threads may allow continued execution if part of process is blocked, especially important for user interfaces

Threads have the same as Process states



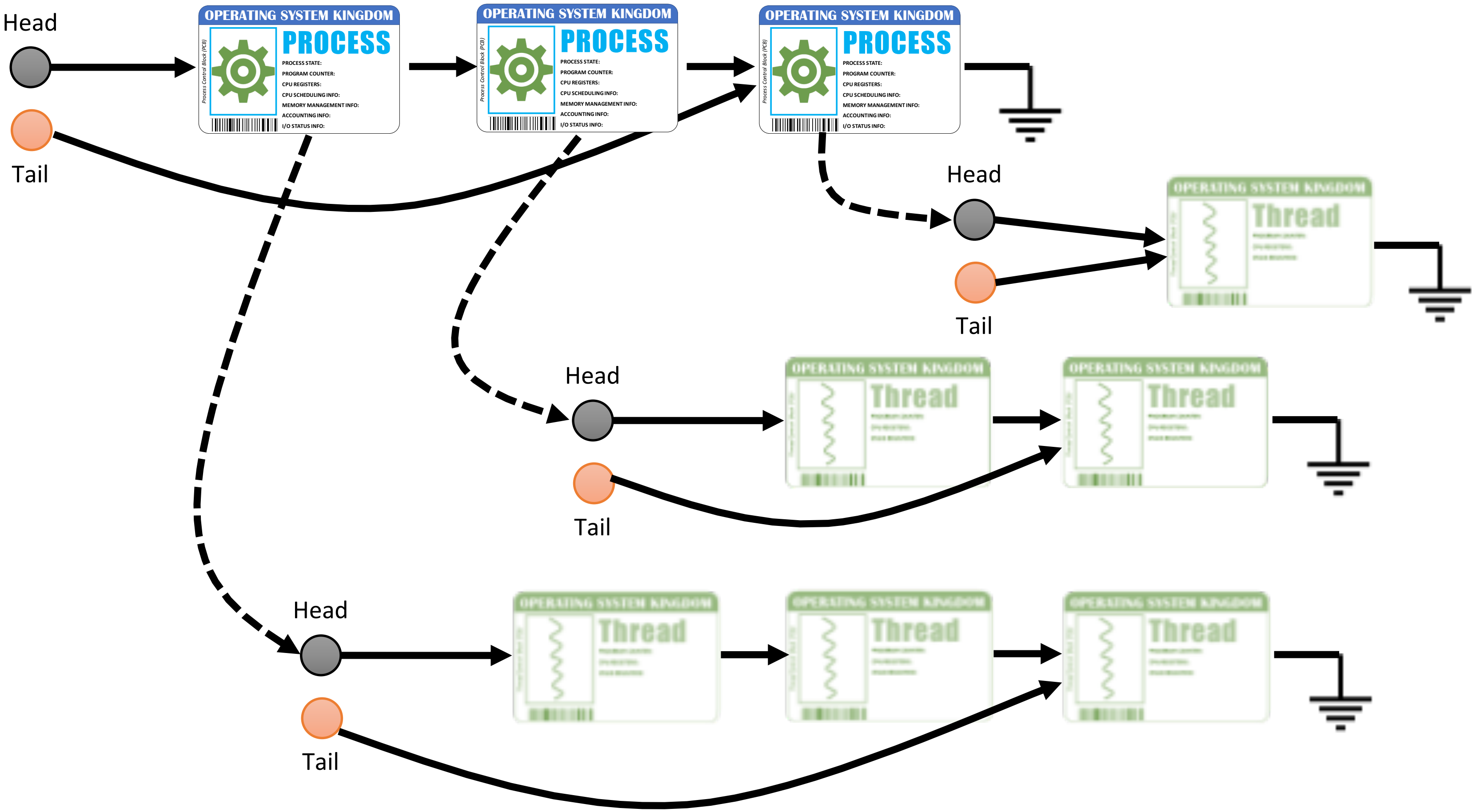
Java Threads: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html

How the operating system manages threads?

Each PCB will point to a list of Thread Control Blocks

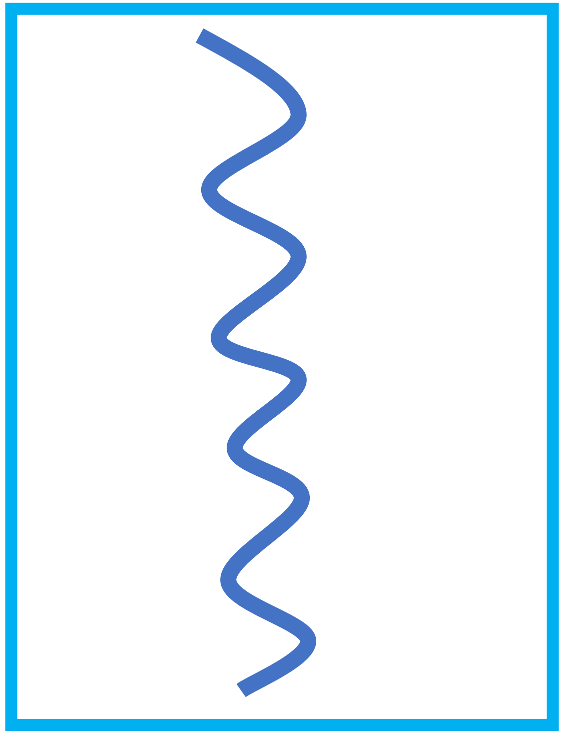
Process Table List

Set of all processes in the system



OPERATING SYSTEM KINGDOM

Thread Control Block (TCB)



Thread

PROGRAM COUNTER:

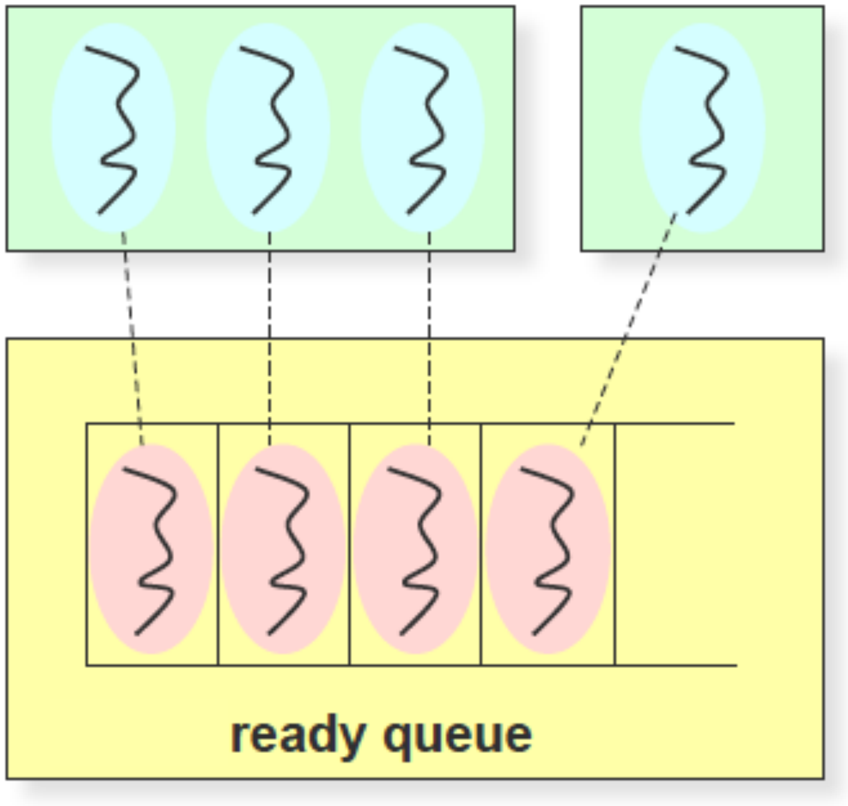
CPU REGISTERS:

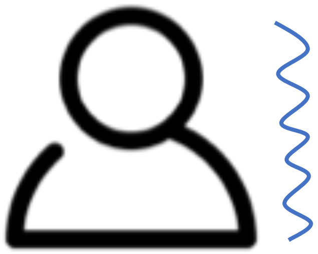
STACK REGISTERS:

THREAD STATE:

Kernel-Level Threads

Managed directly by the operating system.



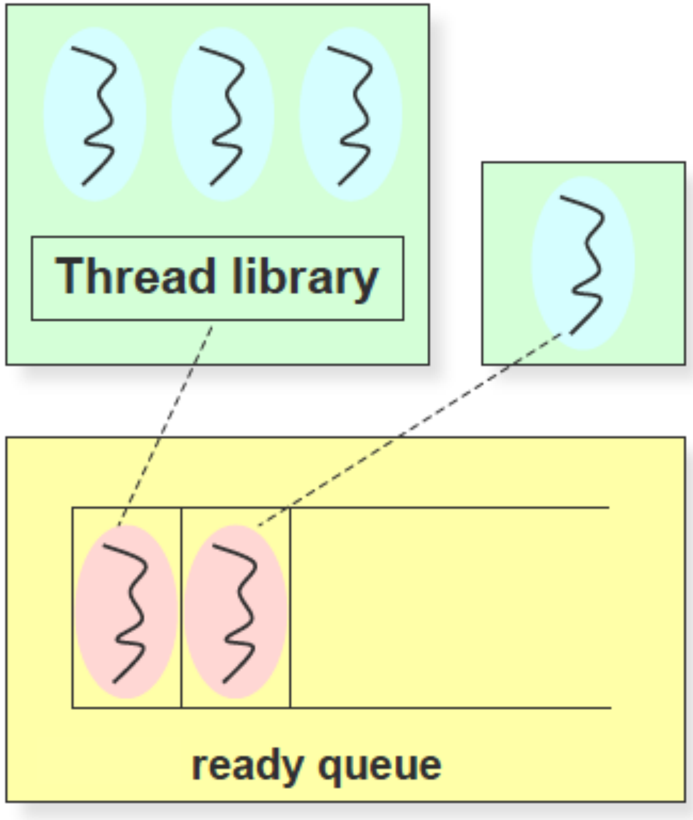


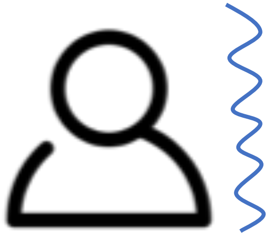
User-Level Threads

Management done by user-level threads library

Thread Libraries

(POSIX Pthreads,
Windows threads,
Java threads)

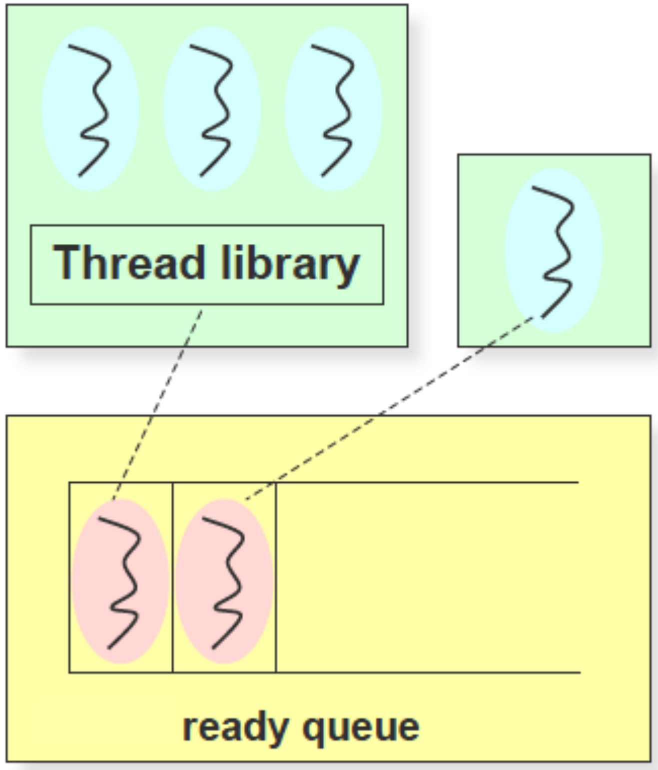




Invisible to Kernel

User-Level Threads

Management done by user-level threads library



All **thread management is done by the thread library** and the **kernel is not aware of the existence** of the user-level threads

A thread represented inside process by a PC, registers, stack, and small thread control block (TCB)

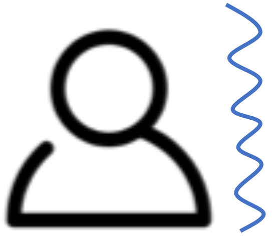
The thread library contains code for creating and destroying, for passing messages and data between threads.

The thread library contains code for scheduling execution and for saving and restoring thread contexts "***Context Switching***"

The process begins with one threads and begins running on that threads. Then, it starts spawning new threads as needed

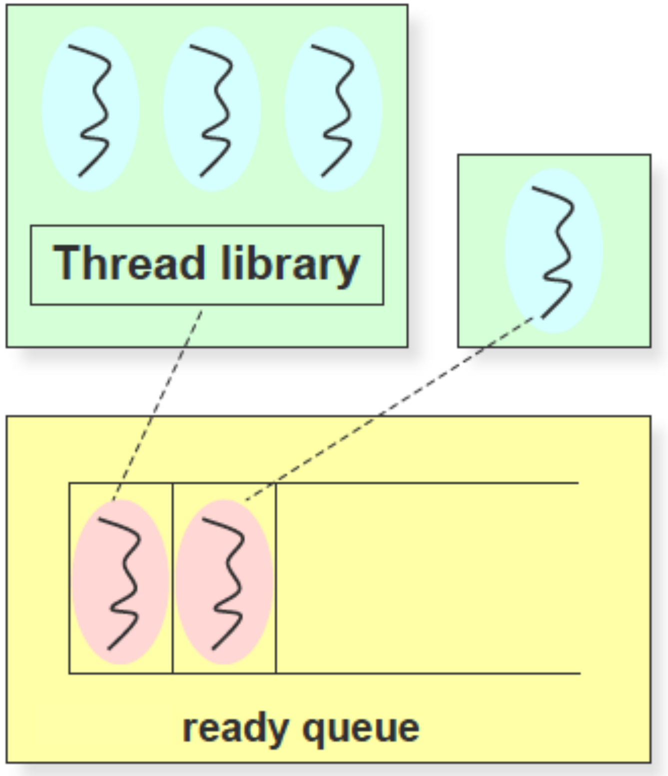
User-level threads are fast to create and manage as there is no kernel involvement

Further Reading: https://books.google.com/books?id=4nxq-Ab6J-EC&pg=SA2-PA23

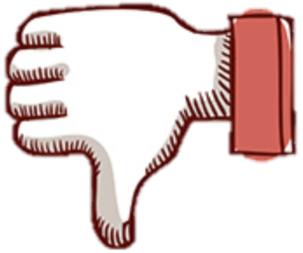


User-Level Threads

Management done by user-level threads library



- Thread switching does not require kernel mode privileges
- User-level thread can run on any operating system
- Scheduling can be application specific
- User-level threads are fast to create and manage
- Creating a new thread, switching, and synchronizing threads are done via **user-level procedure call**

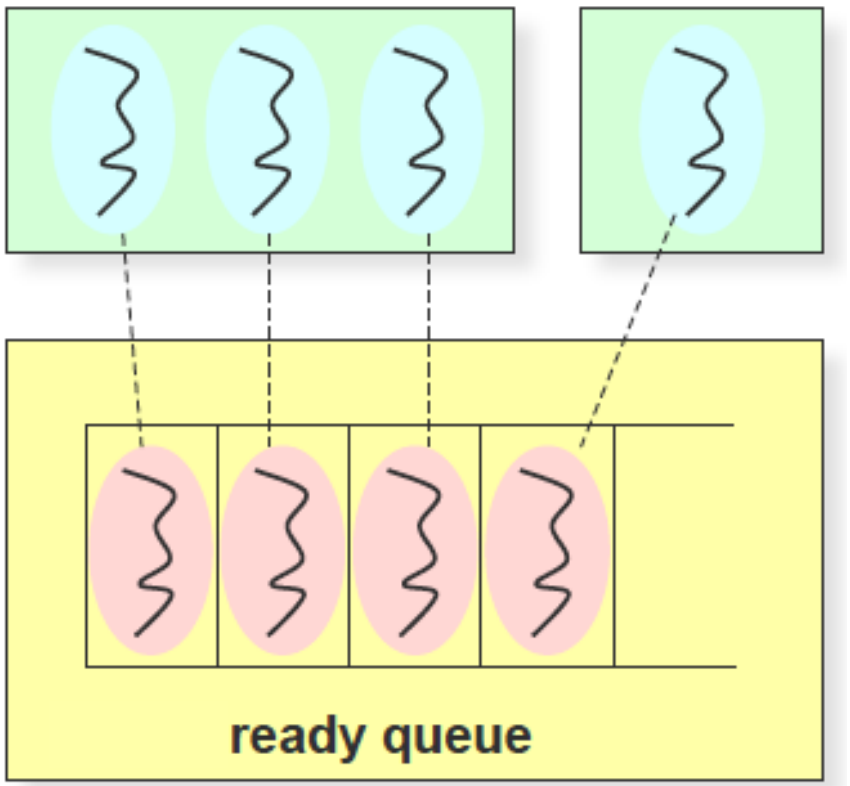


- Most system calls are blocking
- Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
- Multithreaded application cannot take advantage of multiprocessing as these threads are invisible to kernel

There should be communication between the kernel and the user-level thread manager

Kernel-Level Threads

Managed directly by the operating system.



OS-managed threads are called kernel-level threads or lightweight processes (LWP)

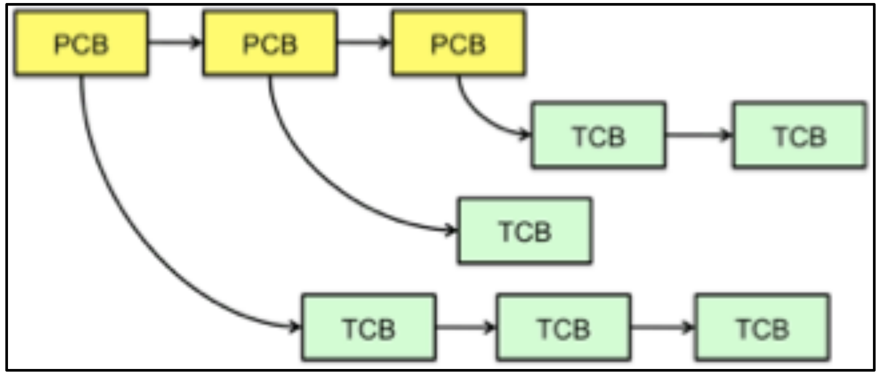
Thread management is done by the kernel and are supported directly by the operating system

The kernel maintains context information for the process as a whole and for the individual threads within the process

No longer scheduling processes, but scheduling in kernel is now on thread basis. (**Scheduler deals in threads**)

The kernel performs thread creation, scheduling, and management in kernel space

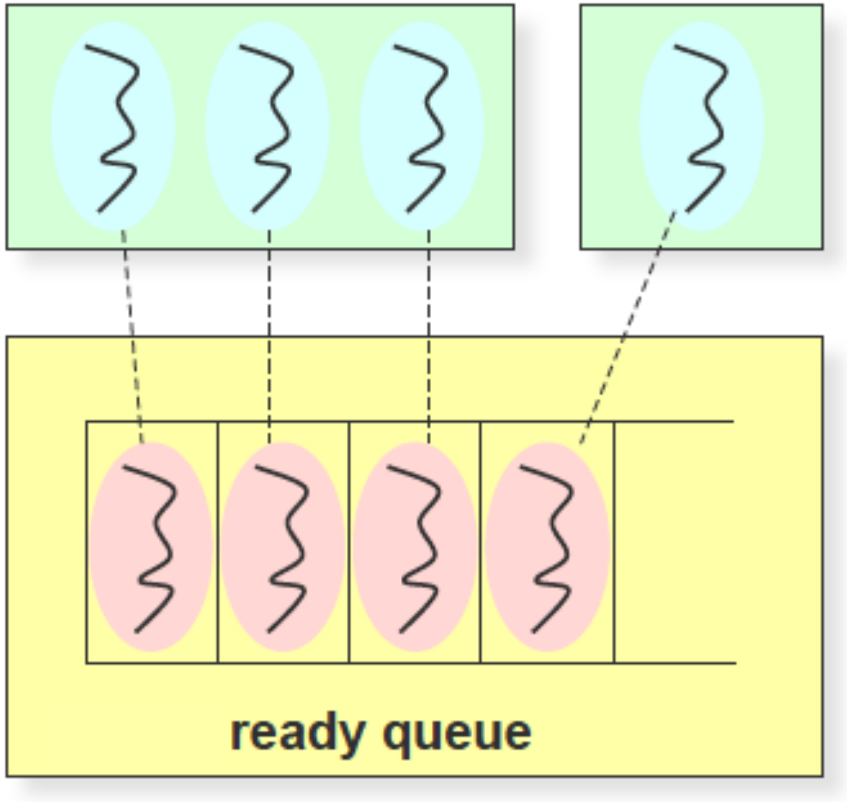
Kernel-level threads are generally slower to create and manage than the user-level threads; **Full Context Switching**



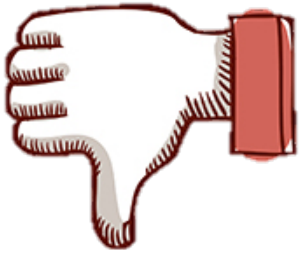
Further Reading: https://www.cs.rutgers.edu/~pxk/416/notes/05-threads.html

Kernel-Level Threads

Managed directly by the operating system.



- Kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread is blocked, the Kernel can schedule another thread in the same process



- Kernel threads are generally slower to create and manage than user-level threads
- Transfer of control from one thread to another thread within the same process requires a mode switch to the kernel

Further Reading: https://www.cs.rutgers.edu/~pxk/416/notes/05-threads.html

Kernel-Level Threads

Managed directly by the operating system.

- Slower to create and manage
- Directly supported by operating system
- Specific to the operating system
- Kernel routines can be multithreaded

User-Level Threads

Management done by user-level threads library

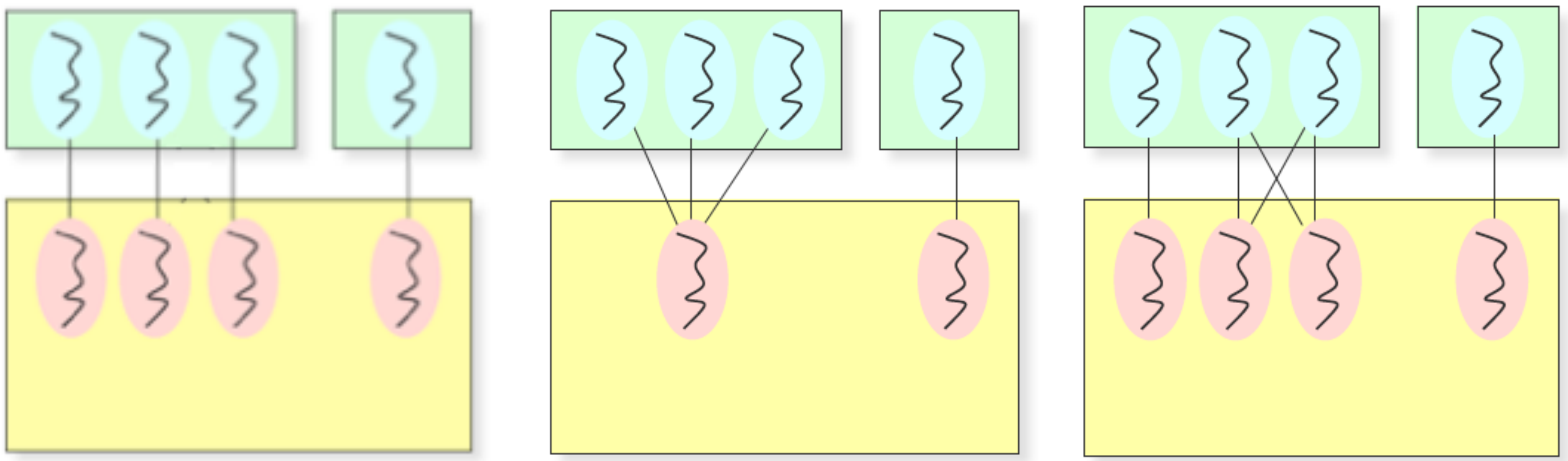
- **100x Faster** to create and manage
- Implemented by a thread library at the user level
- Can run on any operating system
- Multithreaded applications cannot take advantage of multiprocessing

Combining user and kernel-level threads

it's possible to have a program use both user-level and kernel-level threads

An example of why this might be desirable is to have the thread library create several kernel threads to ensure that the operating system can take advantage of **hyperthreading** or **multiprocessing** while using more efficient user-level threads when a very large number of threads is needed.

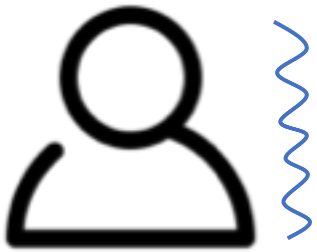
Several user level threads can be run over a single kernel-level thread



One-to-One **Many-to-One** **Many-to-Many**
"Hybrid Threading"

Threads Libraries

provides programmer with API for creating and managing threads



User-Level Thread Library

All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

Kernel-Level Thread Library

All code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

POSIX Pthreads

*Pthreads are IEEE Unix standard library calls "**Specification not Implementation**"*

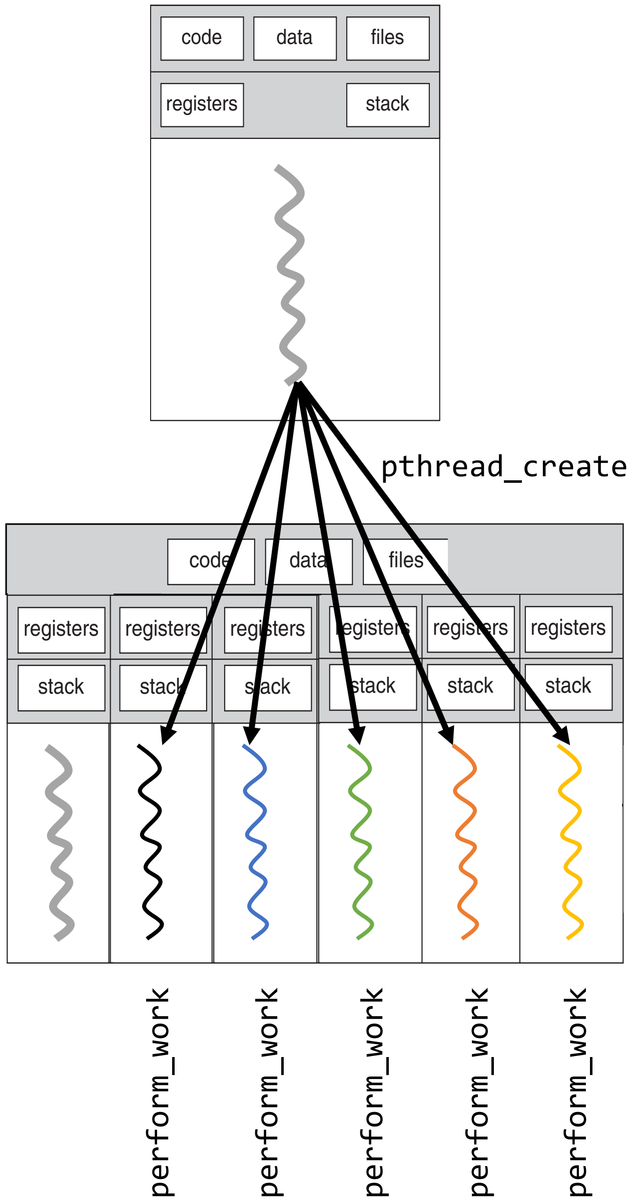
Linux Implementation: https://www.gnu.org/software/hurd/libpthread.html
Windows Implementation: https://sourceforge.net/projects/pthreads4w/

```
$ locate libpthread.so
```

There are around 100 Pthreads procedures, all prefixed `pthread_`

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread’s attribute structure
Pthread_attr_destroy	Remove a thread’s attribute structure

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

#define NUM_THREADS     5

void* perform_work(void *argument){
  int passed_in_value;
  passed_in_value = *( ( int* )argument );
  printf( "Hello World! It's me, thread with argument %d!\n", passed_in_value );
  return NULL;
}

int main( int argc, char** argv ){
  pthread_t threads[ NUM_THREADS ];
  int thread_args[ NUM_THREADS ];
  int result_code;
  unsigned index;

  // create all threads one by one
  for( index = 0; index < NUM_THREADS; ++index ){
    thread_args[ index ] = index;
    printf("In main: creating thread %d\n", index);
    result_code = pthread_create( threads + index, NULL, perform_work, thread_args + index );
    assert( !result_code );
  }

  // wait for each thread to complete
  for( index = 0; index < NUM_THREADS; ++index ){
    // block until thread 'index' completes
    result_code = pthread_join( threads[ index ], NULL );
    assert( !result_code );
    printf( "In main: thread %d has completed\n", index );
  }

  printf( "In main: All threads completed successfully\n" );
  exit( EXIT_SUCCESS );
}
```



Synchronous Threading

https://en.wikipedia.org/wiki/POSIX_Threads#Example

Linux Kernel Threads

https://github.com/torvalds/linux/blob/master/include/linux/sched.h

https://github.com/torvalds/linux/blob/master/kernel/kthread.c

Further Reading: http://www.crashcourse.ca/wiki/index.php/Kernel_threads

つづく