Lecture 20

COP3402 FALL 2015 – DR. MATTHEW GERBER – 11/30/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

Multiprogramming and Processes

Concept of Multiprogramming

Originally, multiprogramming was intended to improve the efficiency of large computers at completing tasks.

- When programs initiated I/O, the CPU remained idle until the I/O operation was completed.
- That time could be used to run other people's programs!

As computers have become cheaper the reasons for multiprogramming are more varied:

- Answering E-mail while running a compiler in the background
- Keeping E-mail available at all times rather than having to reload a program to check it
- Playing music in the background
- Running a voice communication program while playing a video game

And multiprogramming is now expected at every level of computing…

- …all the way down to cell phones!

Definitions and Notes

Multiprogramming is a mode of operation that provides for the interleaved execution of two or more programs by a single processor.

- In the modern era we need to consider the fact that a single processor is really multiple execution cores – this complicates synchronization
- However, that's another discussion for another course

Multiprogramming requires all programs in execution to reside in main memory at the same time, so that when one program is not using the CPU, another program can be assigned to it.

- The iconic case of "not using the CPU" is waiting for input or output.

Multiprogramming was the primary motivation for the concept of the process.

Processes (Review)

A *process* is *a program in execution.*

- Conceptually, it's an asynchronous activity that can run independent of the OS and other processes
  - In reality it's *heavily* dependent on the OS and other processes
  - But *basic execution* still proceeds independently

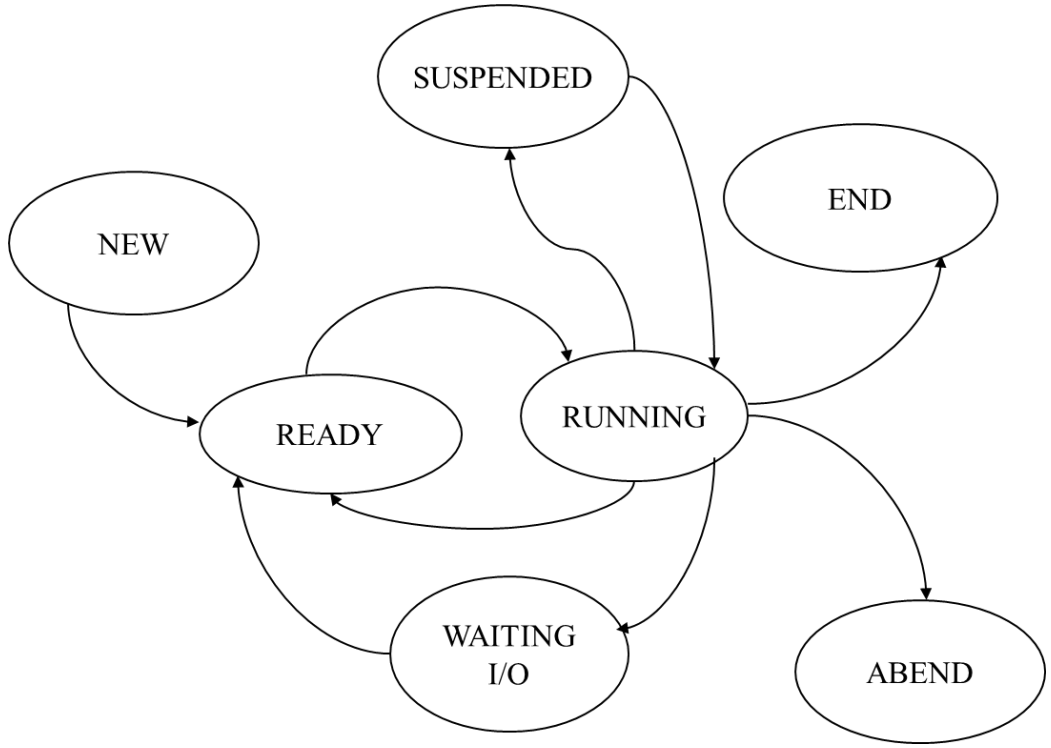
To system software, a process can be viewed as the *locus of control* of a program in execution

Process States

A process can be in one of many different **states** – characterizations of its current disposition with relation to the CPU.

New	Newly created, not yet ready to run.
Ready	Ready to use the CPU, when available.
Running	Currently running on the CPU.
Waiting	Waiting for an event, typically I/O.
Abend	Halted due to an error.
End	Finished executing normally.
Suspend	Stopped temporarily while the OS uses the CPU.

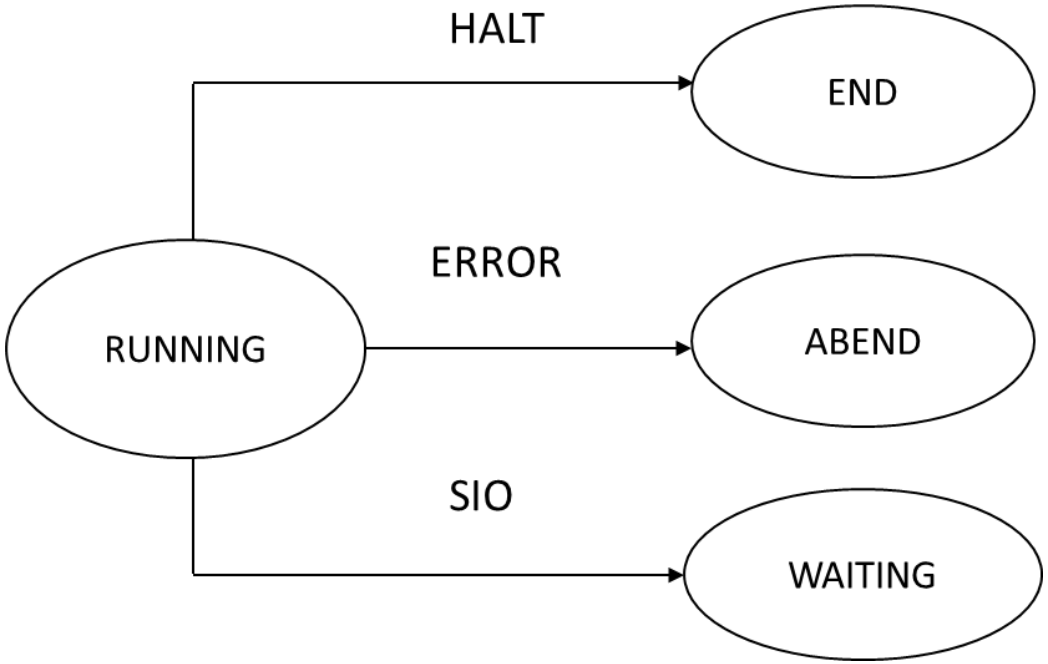
On the right we see the possible transitions between states. Note that only certain transitions make sense!



State Changes

Processes change states both due to normal execution and as the result of interrupts. Here are some possible state changes:

Current	New	Reason
Running	End	HALT (normal end)
Running	ABEND	Error (invalid operation)
Running	Waiting	System Call (SIO)



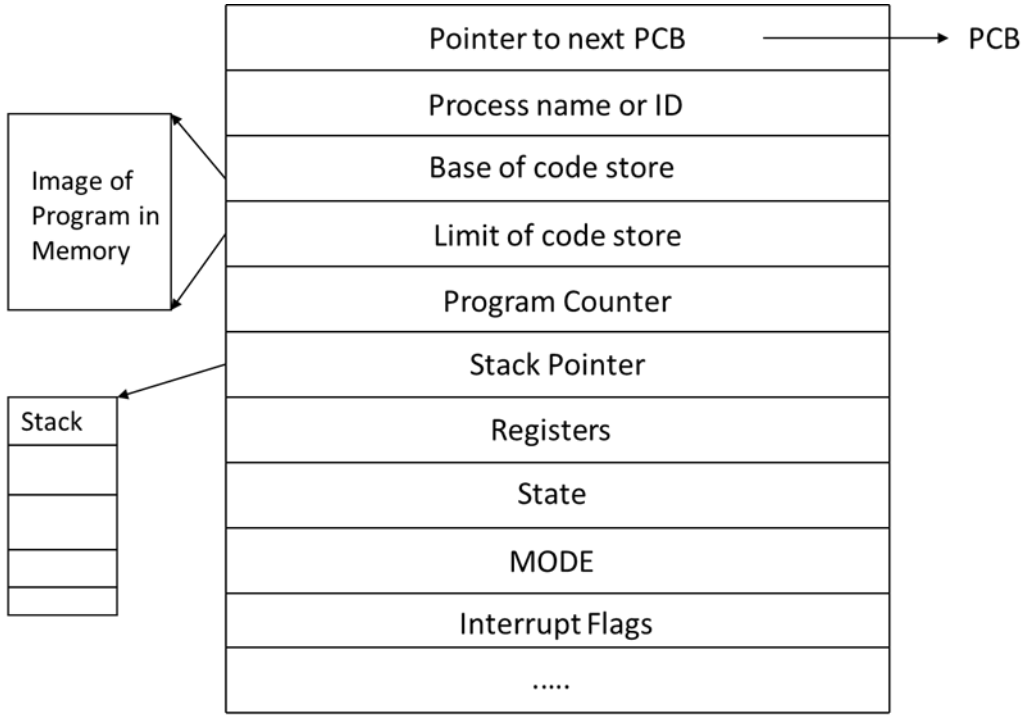
The PCB

An operating system will invariably maintain some sort of *process control block* (PCB) for every running process.

A process may be roughly thought of as the couple of:

- The PCB
- The image of the program in memory

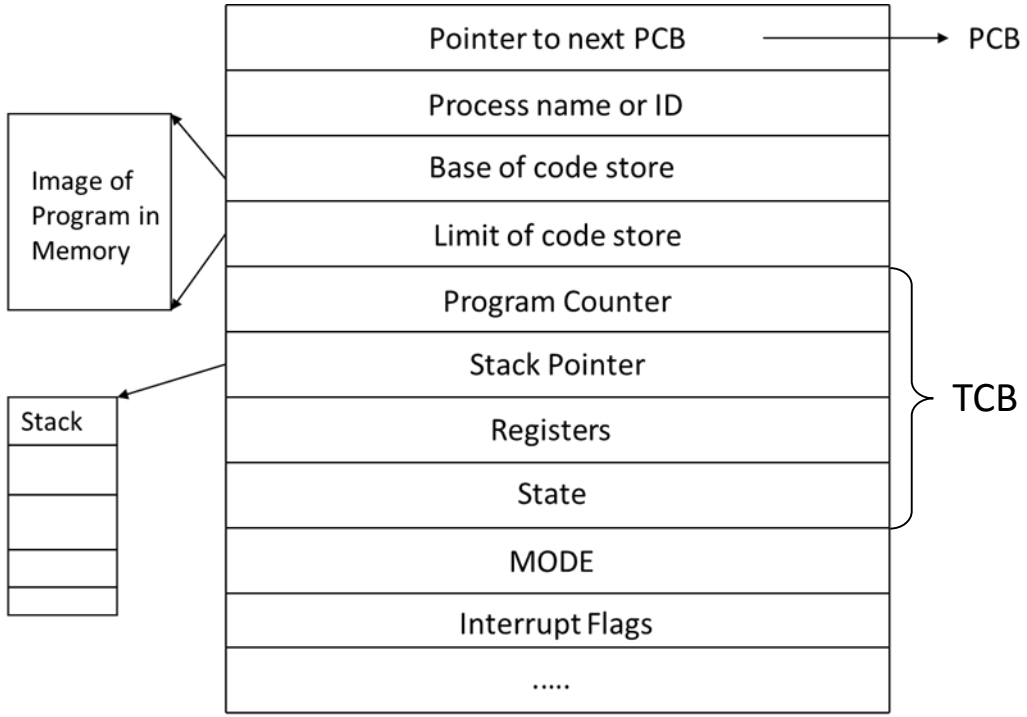
Structure of the PCB



The PCB contains information about processes. For example:

- A unique process identifier
- The current state of the process
- The process's priority
- The contents of some CPU registers
- The program counter
- Base and limit registers
- Time limit information
- I/O status information

Structure of the PCB

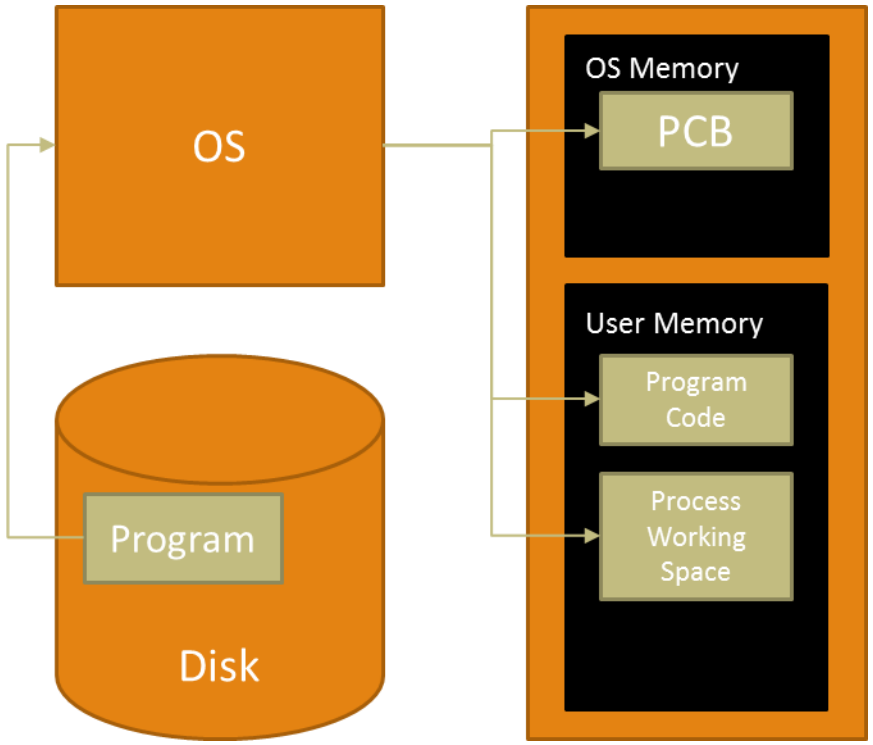


In systems that support multiple threads per process, a thread is a unit of execution of a process.

Each thread therefore needs its own **thread control block**. The TCB contains at least:

- The program counter
- Stack Pointer
- The contents of some CPU registers
- The current state of the thread

Process Creation



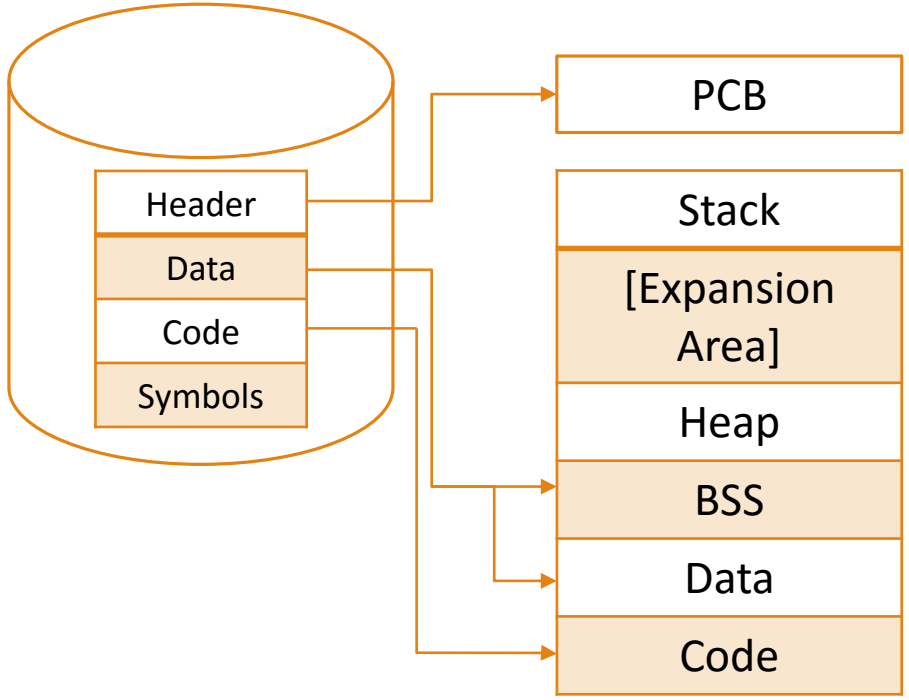
Let's look at process creation a bit more carefully.

The OS:

- Reads the program in from disk
- Copies the image of the program's code into memory, so that it can execute
- Creates a working space for the process – we'll get to that next slide
- Creates a PCB for the process

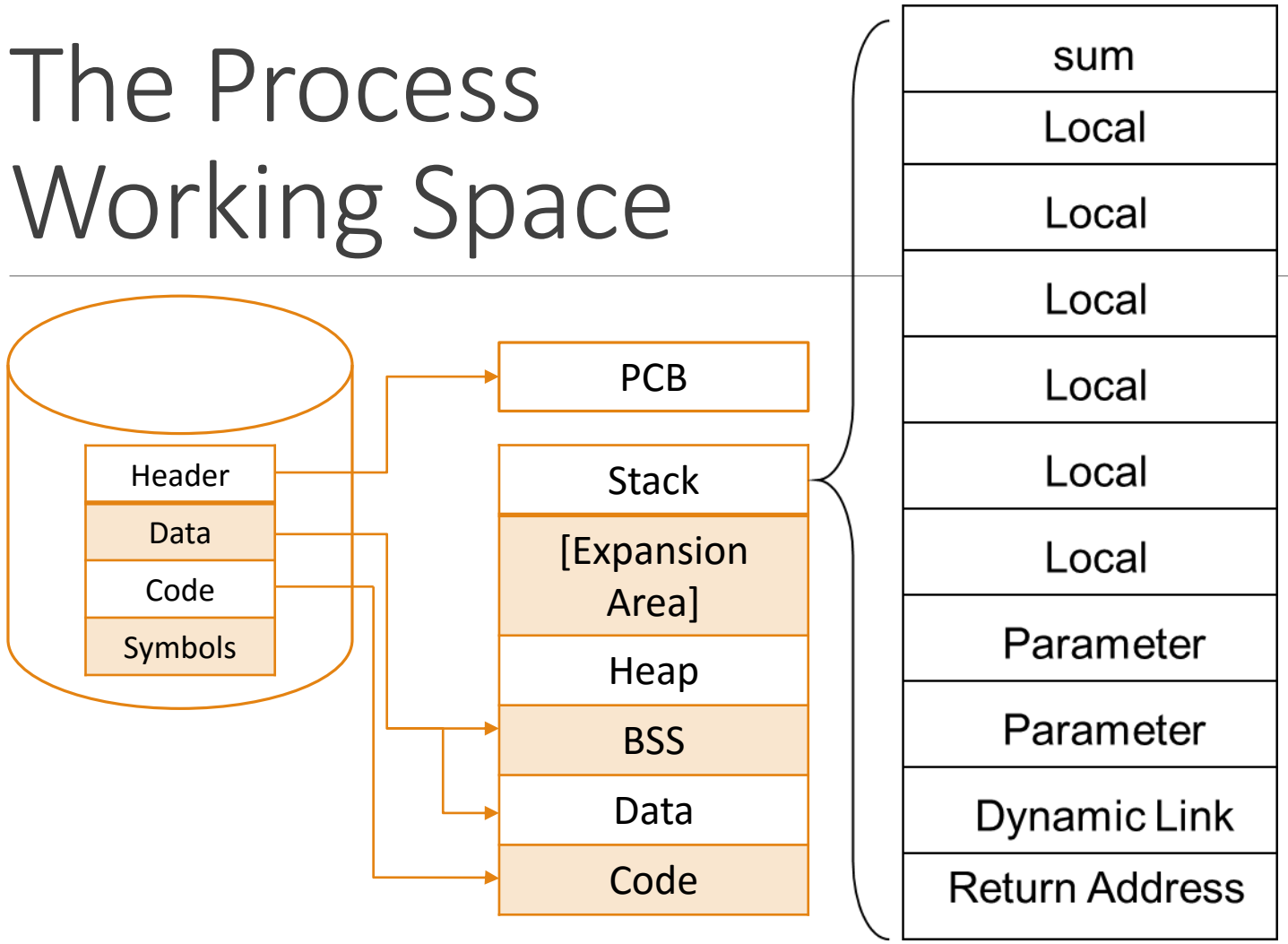
If we're on a threaded operating system, the OS will also need to create a TCB so that there's a unit of execution.

The Process Working Space



- The **heap** is where dynamically allocated memory lives
  - When you malloc() or construct something, it goes here
- The **BSS** and **data** areas are similar for these purposes
- The **code** area is where the process executes its code from
  - Only code execution should happen in this area
  - If anything writes to the code area mid-process, something is *very wrong!*

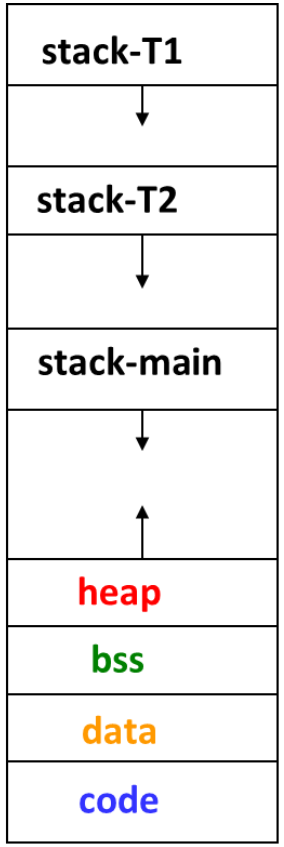
The Process Working Space



You're already *quite* familiar with the stack.

- Generally, the stack expands downward from the top of the address space; the heap expands upward from the bottom of it
- Procedure-level local variables still live here
- This gets much more complicated in the case of multithreaded processes

Stacks in Multithreaded Processes



Multithreaded processes need a stack per thread.

- (Threads have their own execution context)
- (That's the whole point of a thread)

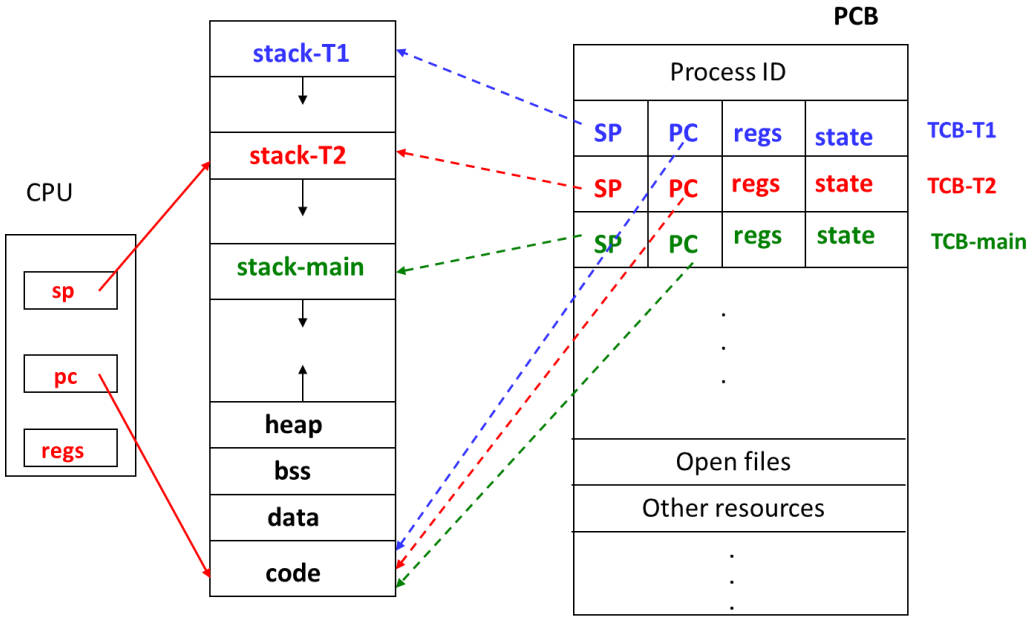
…but they all have to share the same address space!

- Each thread has its own TCB, and therefore, its own stack pointer

You're probably wondering how serious a problem stack size is

- It would be an unworkable mess without virtual memory
- …which you'll learn about in another course

Threads



We've mentioned that threads have their own execution context – let's expand on that a bit.

- A thread **does** have an execution context – a call stack, a program counter, a set of registers, and a state
- A thread does **not** have its own resources – all threads of the same process share the same heap, open files, and other resources
- How time is allocated between threads depends on the OS

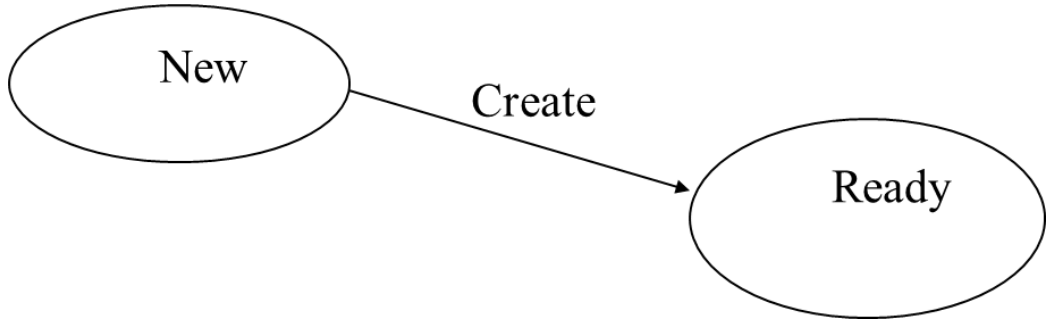
Among many other implications, this means that confusing stack-allocated variables with heap-allocated variables in a multithreaded context is a **very serious mistake.**

Sub-process Creation and the Ready State

During the course of execution, a user process can request the creation of new processes and threads.

- The creating process is called a **parent**.
- The created process is called a **child**.

One of the last steps in creating a process is changing its state to indicate that it is now **ready** to run. It begins at the back of the **ready queue** and makes its way toward execution via the **scheduling algorithm** (well beyond the scope of this course).



Next Time: Transitions and Wrapping Up Processes