Operating System Concepts

Lecture 4b: Process Abstraction

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

- Process Abstraction
  - How does the OS create this abstraction?
  - Why is it useful?
  - What happens during context switching?

Process abstraction

- Definition: a process is a program during execution
  it is an execution environment with restricted rights

Process abstraction

- Definition: a process is a program during execution
  it is an execution environment with restricted rights
  - has its own resources (CPU registers, memory to contain program **code** and **data**, file descriptors, etc.)

Process abstraction

- Definition: a process is a program during execution
  it is an execution environment with restricted rights
  - has its own resources (CPU registers, memory to contain program **code** and **data**, file descriptors, etc.)
  - encapsulates one or more **threads** sharing process resources

Process abstraction

- Definition: a process is a program during execution
  it is an execution environment with restricted rights
  - has its own resources (CPU registers, memory to contain program **code** and **data**, file descriptors, etc.)
  - encapsulates one or more **threads** sharing process resources
    - we assume one thread per process today

Process abstraction

- Definition: a process is a program during execution
  it is an execution environment with restricted rights
  - has its own resources (CPU registers, memory to contain program **code** and **data**, file descriptors, etc.)
  - encapsulates one or more **threads** sharing process resources
    - we assume one thread per process today
  - is characterized by a unique identifier (PID)

Process abstraction

- Definition: a process is a program during execution
  it is an execution environment with restricted rights
  - has its own resources (CPU registers, memory to contain program **code** and **data**, file descriptors, etc.)
  - encapsulates one or more **threads** sharing process resources
    - we assume one thread per process today
  - is characterized by a unique identifier (PID)
- different processes may run different instances of the same program (process $\neq$ program)

Process abstraction

- Definition: a process is a program during execution
  it is an execution environment with restricted rights
  - has its own resources (CPU registers, memory to contain program **code** and **data**, file descriptors, etc.)
  - encapsulates one or more **threads** sharing process resources
    - we assume one thread per process today
  - is characterized by a unique identifier (PID)
- different processes may run different instances of the same program (process ≠ program)
  - e.g., you can run several instances of a web browser

Process abstraction

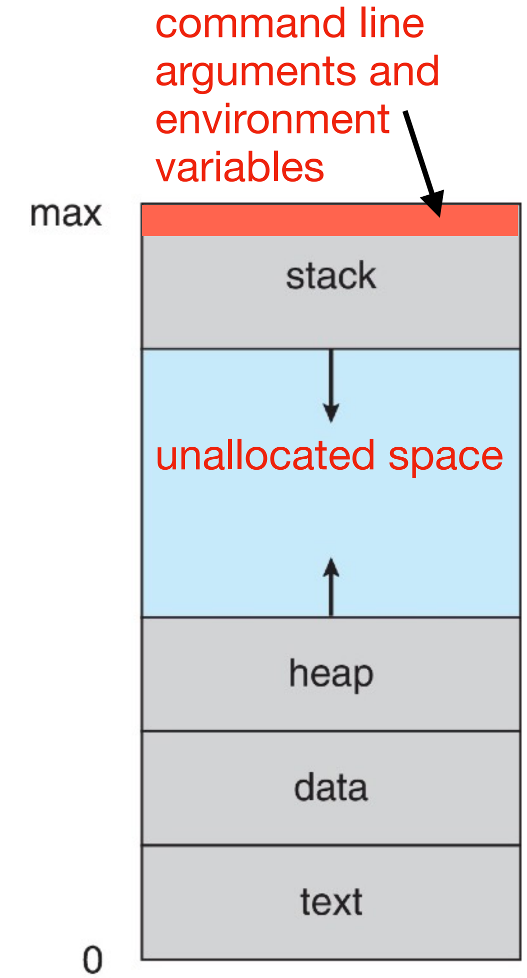
- Definition: a process is a program during execution
  it is an execution environment with restricted rights
  - has its own resources (CPU registers, memory to contain program **code** and **data**, file descriptors, etc.)
  - encapsulates one or more **threads** sharing process resources
    - we assume one thread per process today
  - is characterized by a unique identifier (PID)
- different processes may run different instances of the same program (process $\neq$ program)
  - e.g., you can run several instances of a web browser
- why do we need the process abstraction?

Process abstraction

- Definition: a process is a program during execution
  it is an execution environment with restricted rights
  - has its own resources (CPU registers, memory to contain program **code** and **data**, file descriptors, etc.)
  - encapsulates one or more **threads** sharing process resources
    - we assume one thread per process today
  - is characterized by a unique identifier (PID)
- different processes may run different instances of the same program (process $\neq$ program)
  - e.g., you can run several instances of a web browser
- why do we need the process abstraction?
  - necessary for concurrent execution and protection

Memory layout for process

- each process has multiple parts
  - text section containing the program code
  - data section containing global variables (initialized and uninitialized)
  - stack containing temporary data, function parameters, return addresses, and local variables
  - heap containing memory dynamically allocated during run time using malloc( ) from glibc or the **`sbrk( )` system call**



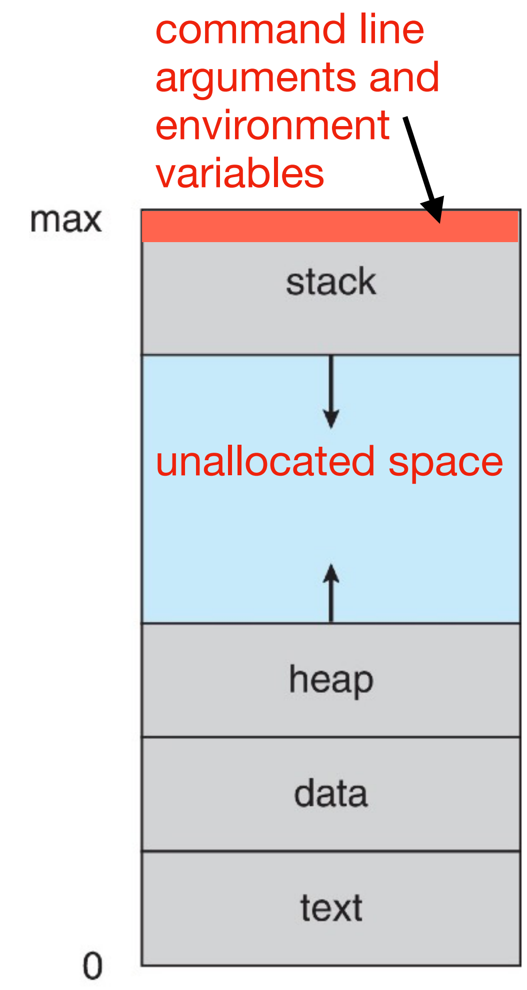
Programmer's view of memory: single space containing only this one program

Memory layout for process

- each process has multiple parts
  - text section containing the program code
  - data section containing global variables (initialized and uninitialized)
  - stack containing temporary data, function parameters, return addresses, and local variables
  - heap containing memory dynamically allocated during run time using malloc( ) from glibc or the **`sbrk( )` system call**

Which variables are in the stack?

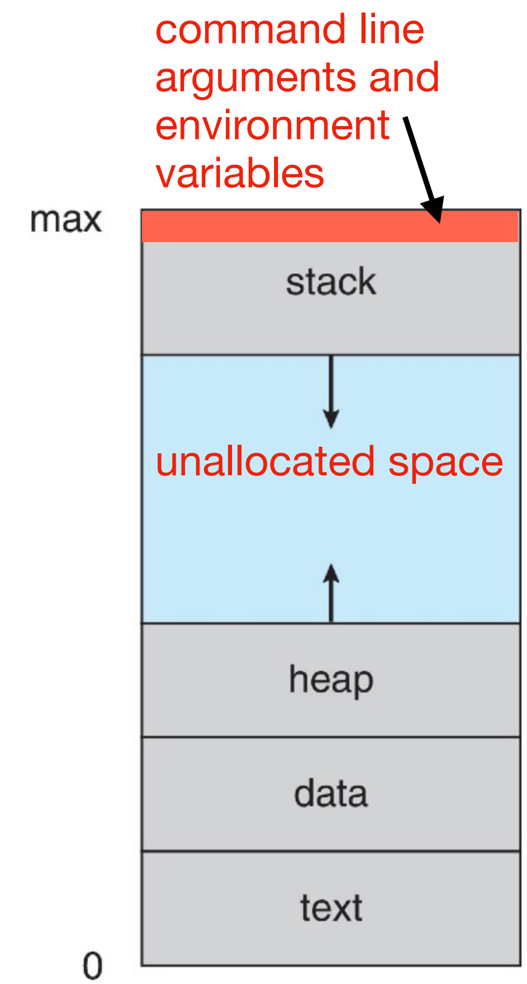
```
#include <stdio.h>
void foo (int n) {
   int i, a[5], *b;
   if (n == 0) return;
   b = new int[n];
   printf ("foo(%d): %p,%p,%p,%p \n", n, &i, a, &b, b);
   foo(n-1);
}
main ( ) { foo(10); }
```



Programmer's view of memory: single space containing only this one program

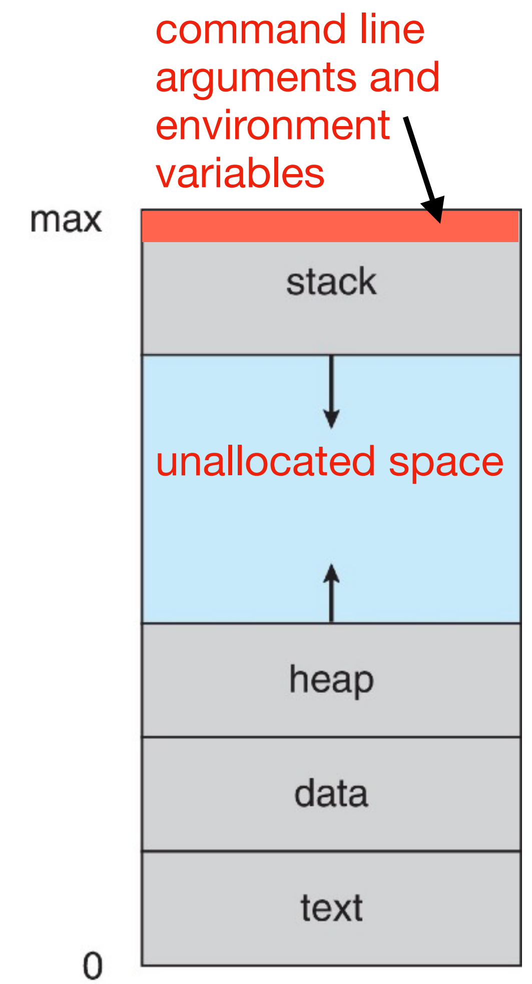
Memory layout for process

- each process has multiple parts
  - text section containing the program code
  - data section containing global variables (initialized and uninitialized)
  - stack containing temporary data, function parameters, return addresses, and local variables
  - heap containing memory dynamically allocated during run time using malloc( ) from glibc or the **`sbrk( )` system call**
- each process has a distinct and isolated **address space** (i.e., addresses that can be accessed by its code)
  - addresses in the executable file are as if it is loaded at memory address 00000000
  - these addresses need to be adjusted when the program is **relocated** to somewhere else
  - no process can read or write memory of another process



Memory layout for process

- each process has multiple parts
  - text section containing the program code
  - data section containing global variables (initialized and uninitialized)
  - stack containing temporary data, function parameters, return addresses, and local variables
  - heap containing memory dynamically allocated during run time using malloc( ) from glibc or the **`sbrk( )` system call**
- each process has a distinct and isolated **address space** (i.e., addresses that can be accessed by its code)
  - addresses in the executable file are as if it is loaded at memory address 00000000
  - these addresses need to be adjusted when the program is **relocated** to somewhere else
  - no process can read or write memory of another process
- hardware translates from virtual to physical addresses

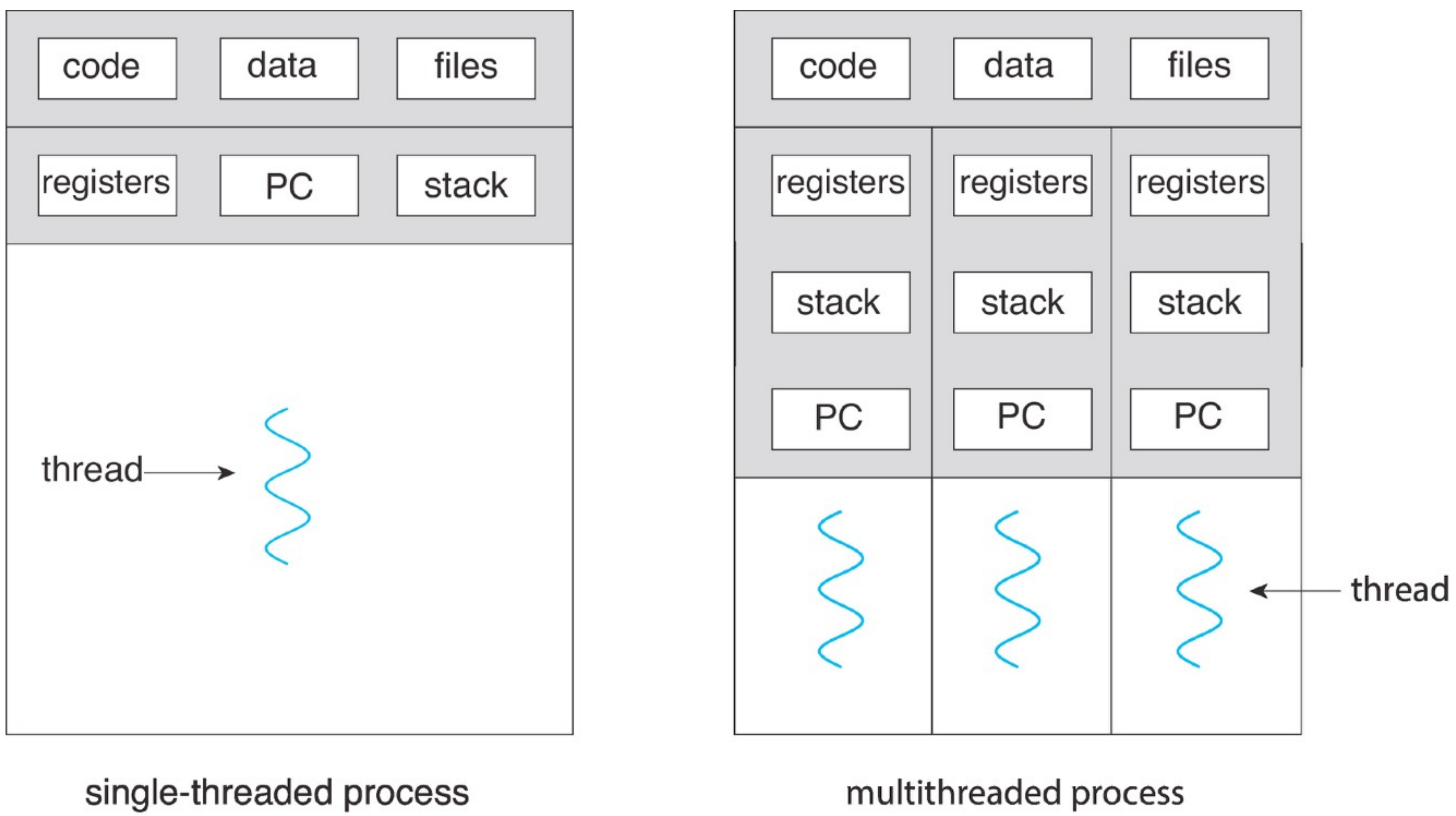


Single vs. multi-threaded process

- <u>Definition</u>: a thread is a sequential execution stream of instructions

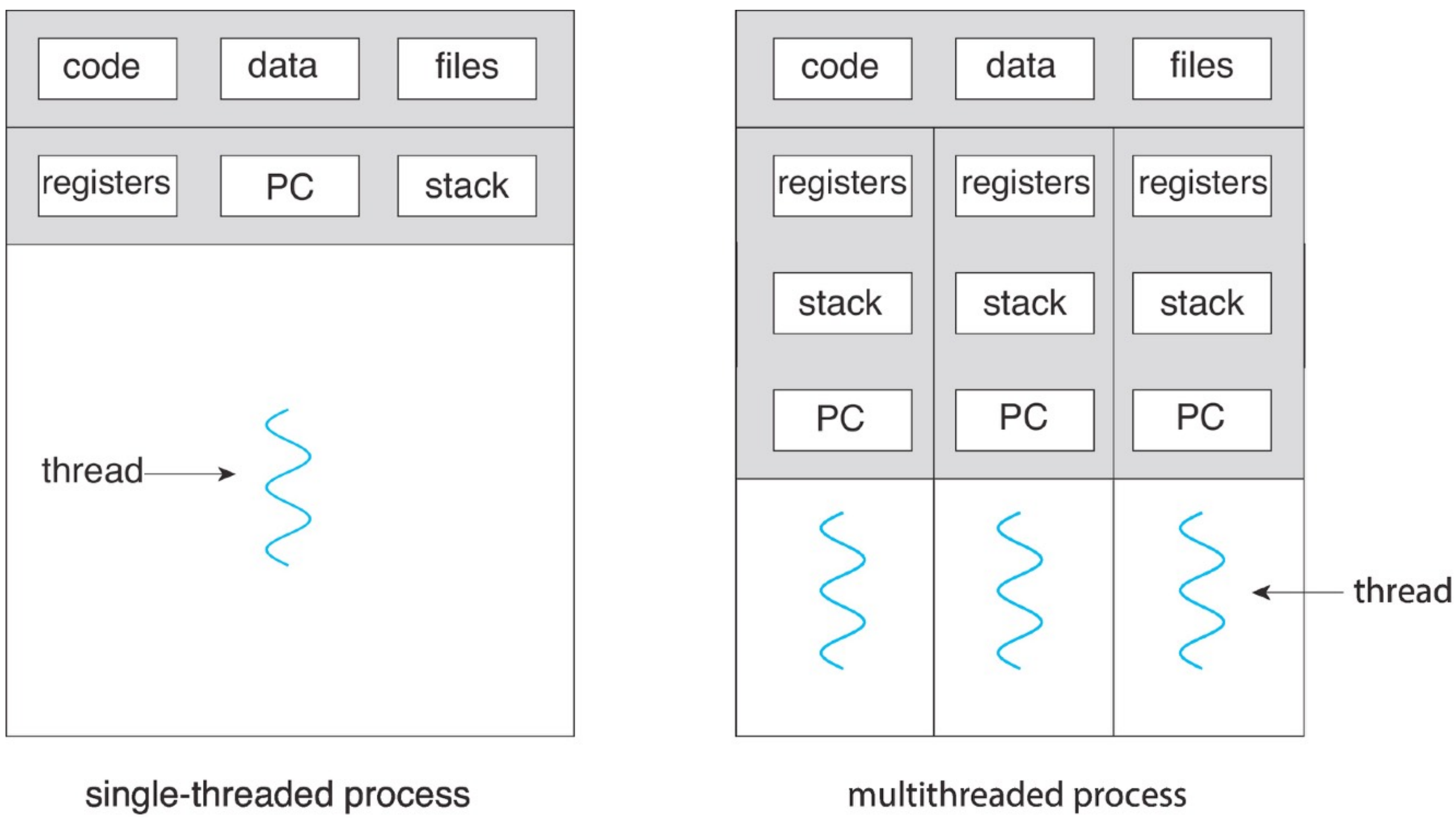
Single vs. multi-threaded process

- Definition: a thread is a sequential execution stream of instructions
- the address space of the process is shared among its threads
  - they share heap, text, static data sections in addition to file descriptors



Single vs. multi-threaded process

- <u>Definition</u>: a thread is a sequential execution stream of instructions
- the address space of the process is shared among its threads
  - they share heap, text, static data sections in addition to file descriptors
- threads can execute simultaneously on different cores of a multicore system
  - in a word processor you can simultaneously type a character and run the spell checker!



Process control block

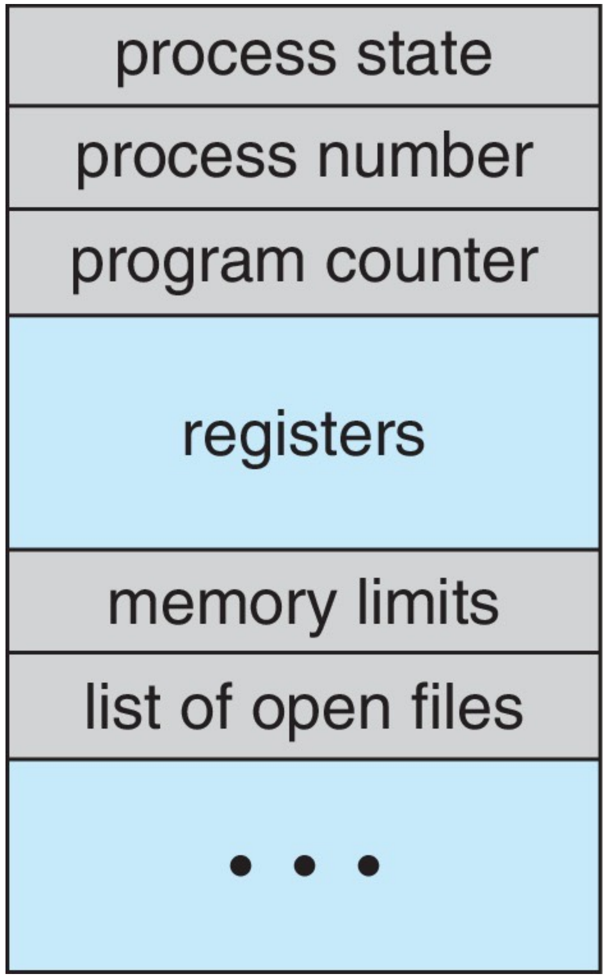
- for every process, OS keeps the state of process execution (metadata) in a **process control block** (PCB)

Process control block

- for every process, OS keeps the state of process execution (metadata) in a **process control block** (PCB)
- PCB is a kernel data structure in memory; it represents run-time information about the process, defining its **context**
  - Process status (running, ready, blocked/waiting)
  - Process ID (PID) and its children's PIDs
  - CPU registers' states, including program counter (PC), stack pointer (SP), heap pointer (HP), base/relocation and limit registers, page-table base register (PTBR), and general-purpose registers
  - Thread control block(s)
  - Address space
  - Accounting information (e.g., execution time, time elapsed since start)
  - Scheduling information (e.g., priorities, queue pointers for state queues)
  - Set of OS resources in use (e.g., list of open files, I/O devices allocated to the process)
  - Username of owner
  - ...

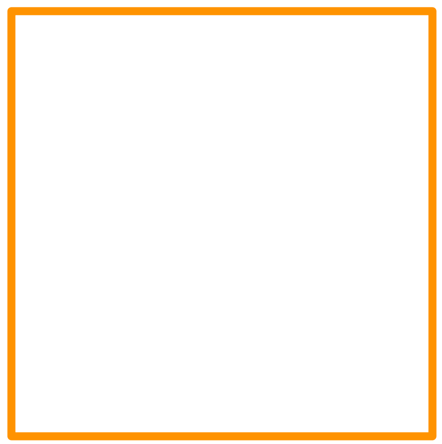
Process control block

- for every process, OS keeps the state of process execution (metadata) in a **process control block** (PCB)
- PCB is a kernel data structure in memory; it represents run-time information about the process, defining its **context**
  - Process status (running, ready, blocked/waiting)
  - Process ID (PID) and its children's PIDs
  - CPU registers' states, including program counter (PC), stack pointer (SP), heap pointer (HP), base/relocation and limit registers, page-table base register (PTBR), and general-purpose registers
  - Thread control block(s)
  - Address space
  - Accounting information (e.g., execution time, time elapsed since start)
  - Scheduling information (e.g., priorities, queue pointers for state queues)
  - Set of OS resources in use (e.g., list of open files, I/O devices allocated to the process)
  - Username of owner
  - ...
- PCB in Linux is represented by the C structure called task_struct which is defined in `<linux/sched.h>`
  - task_struct contains mm_struct which represents the address space of a given process



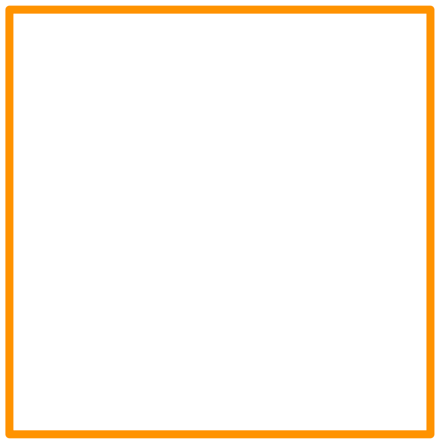
Loading — revisited

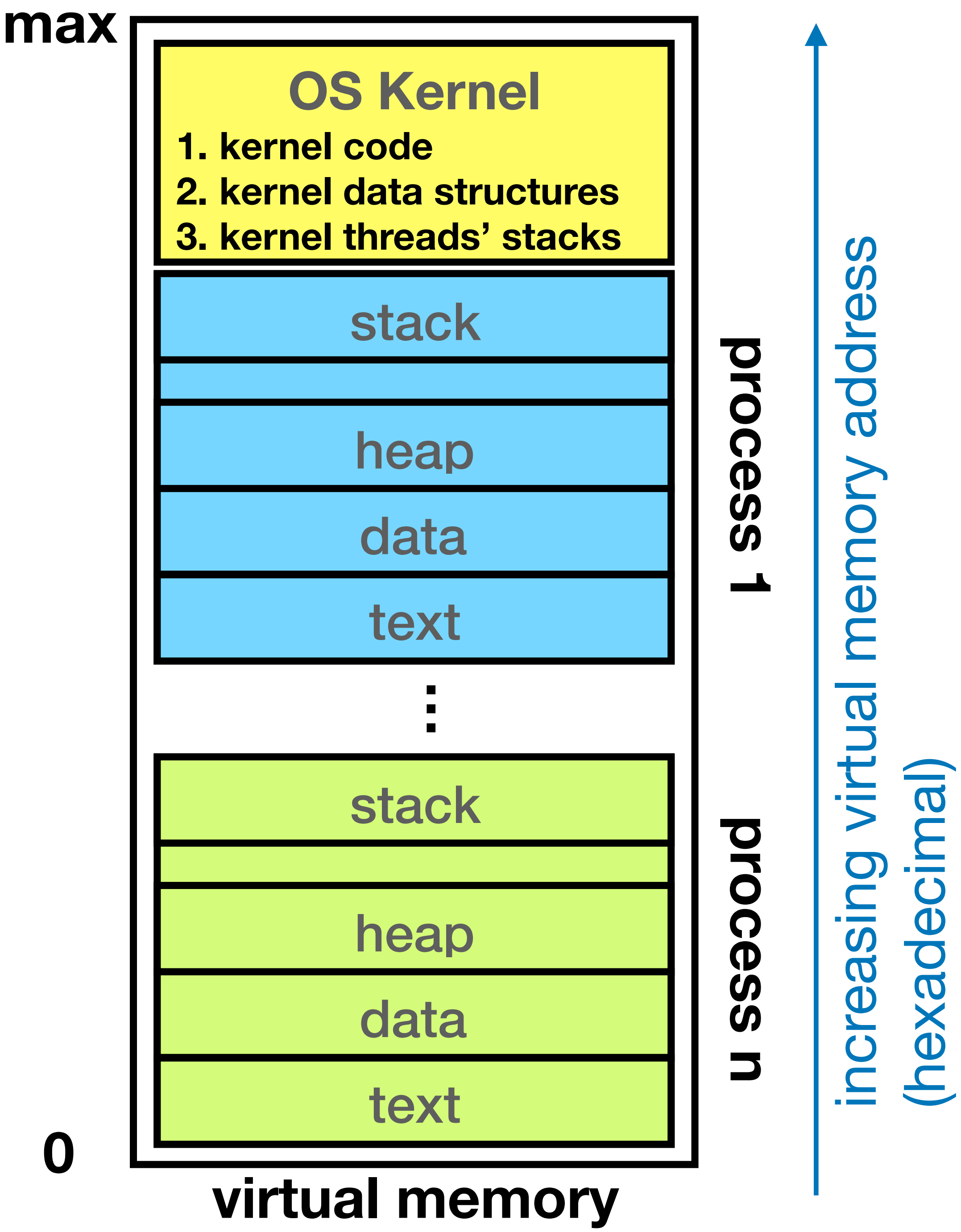
a.out



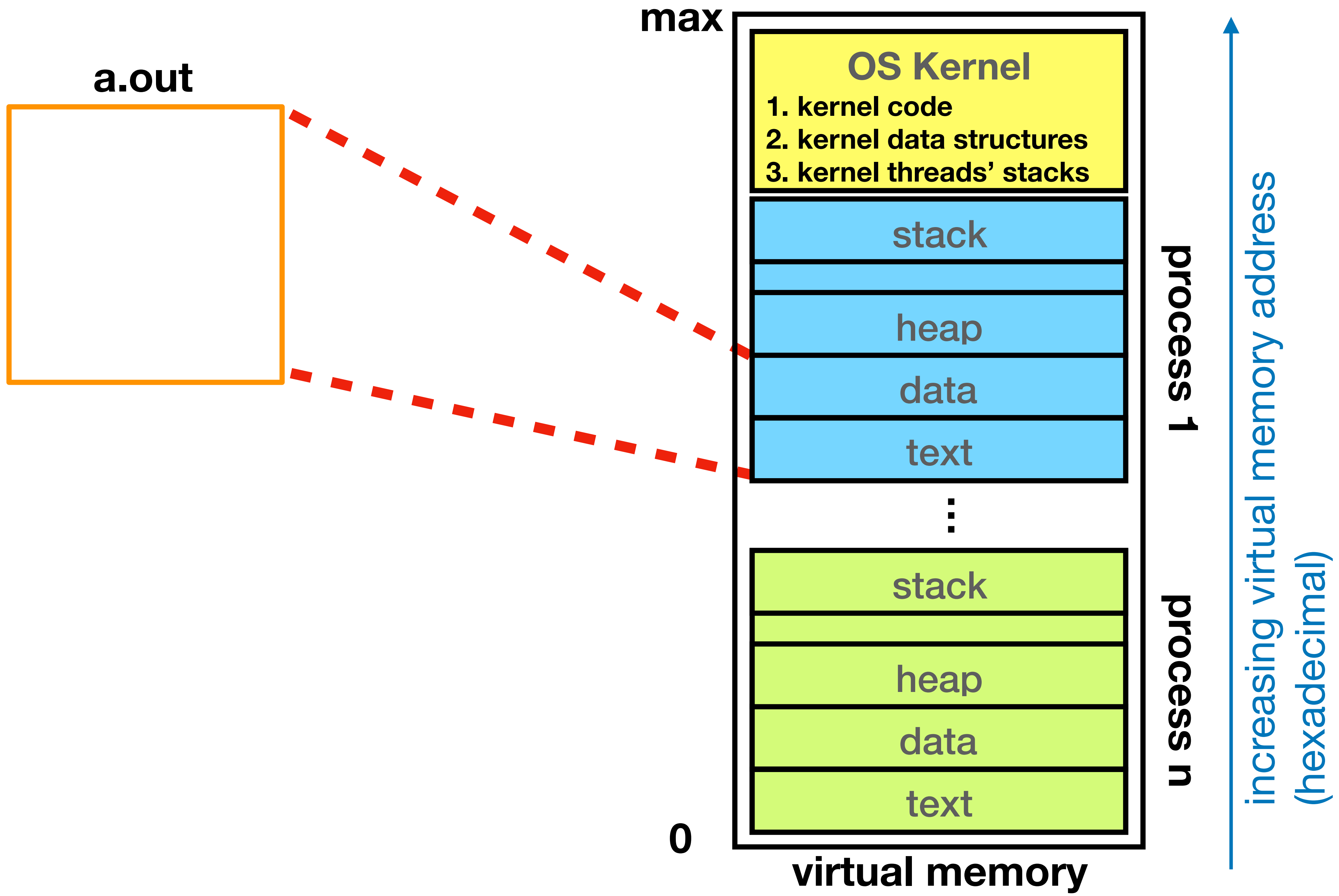
Loading — revisited

a.out

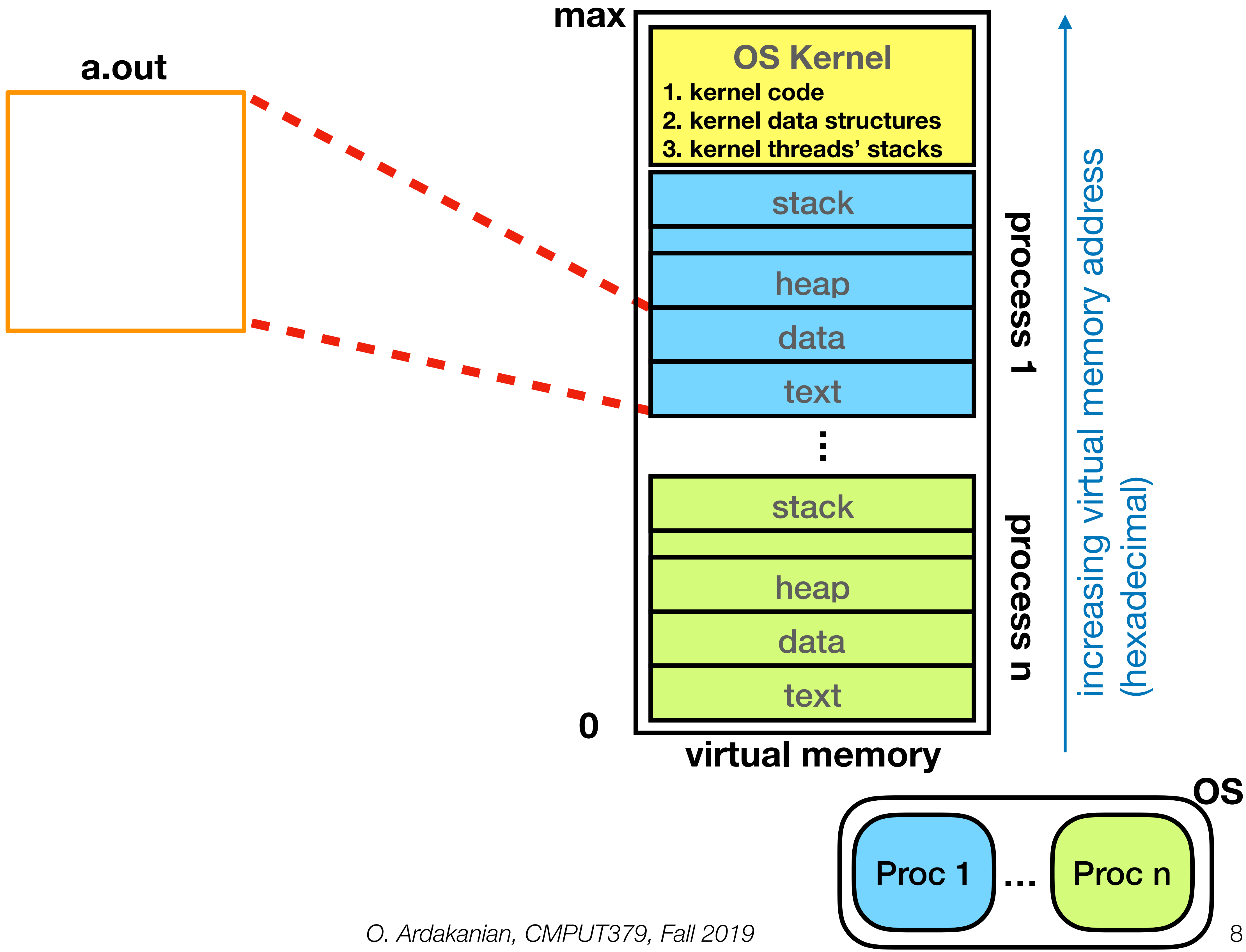




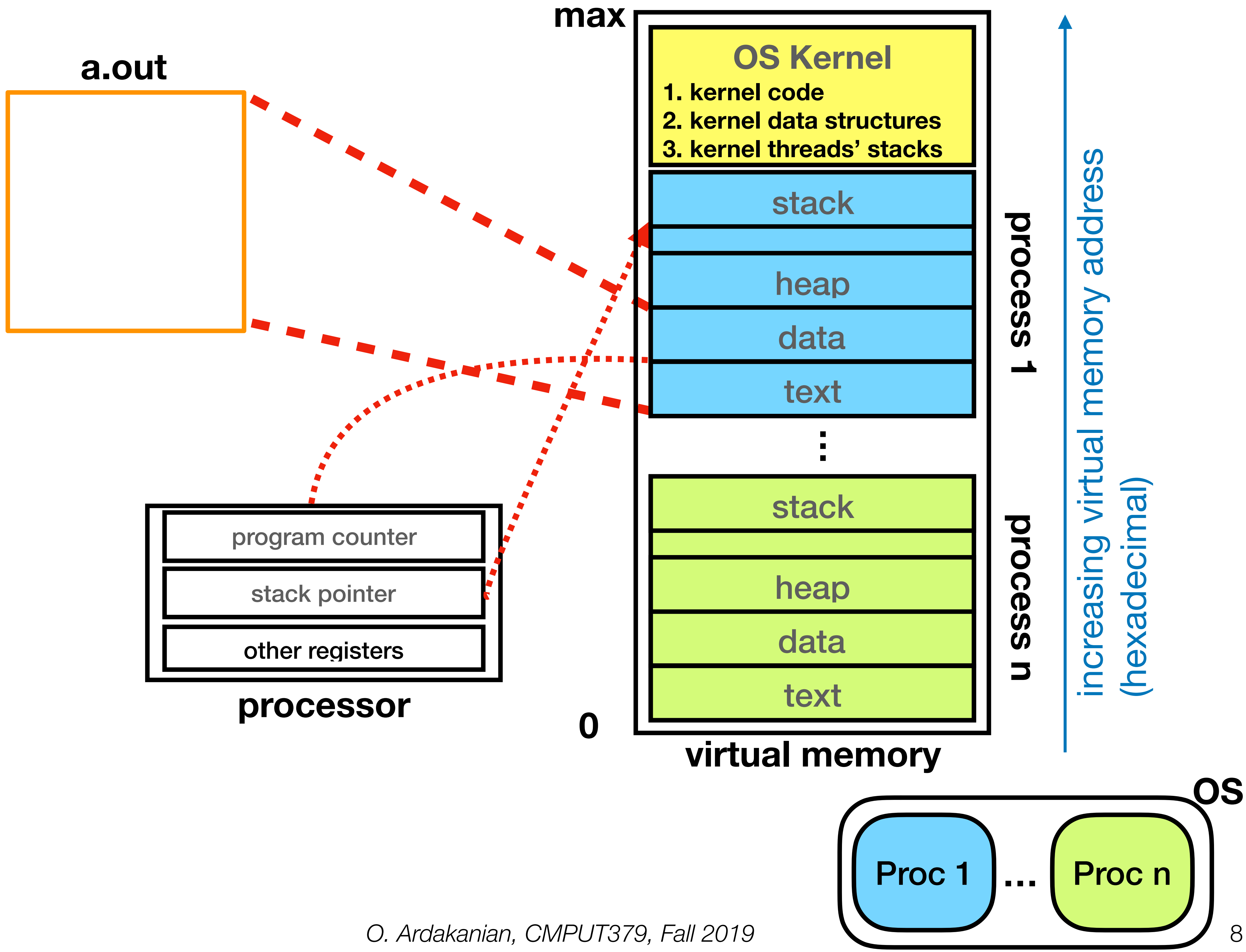
Loading — revisited



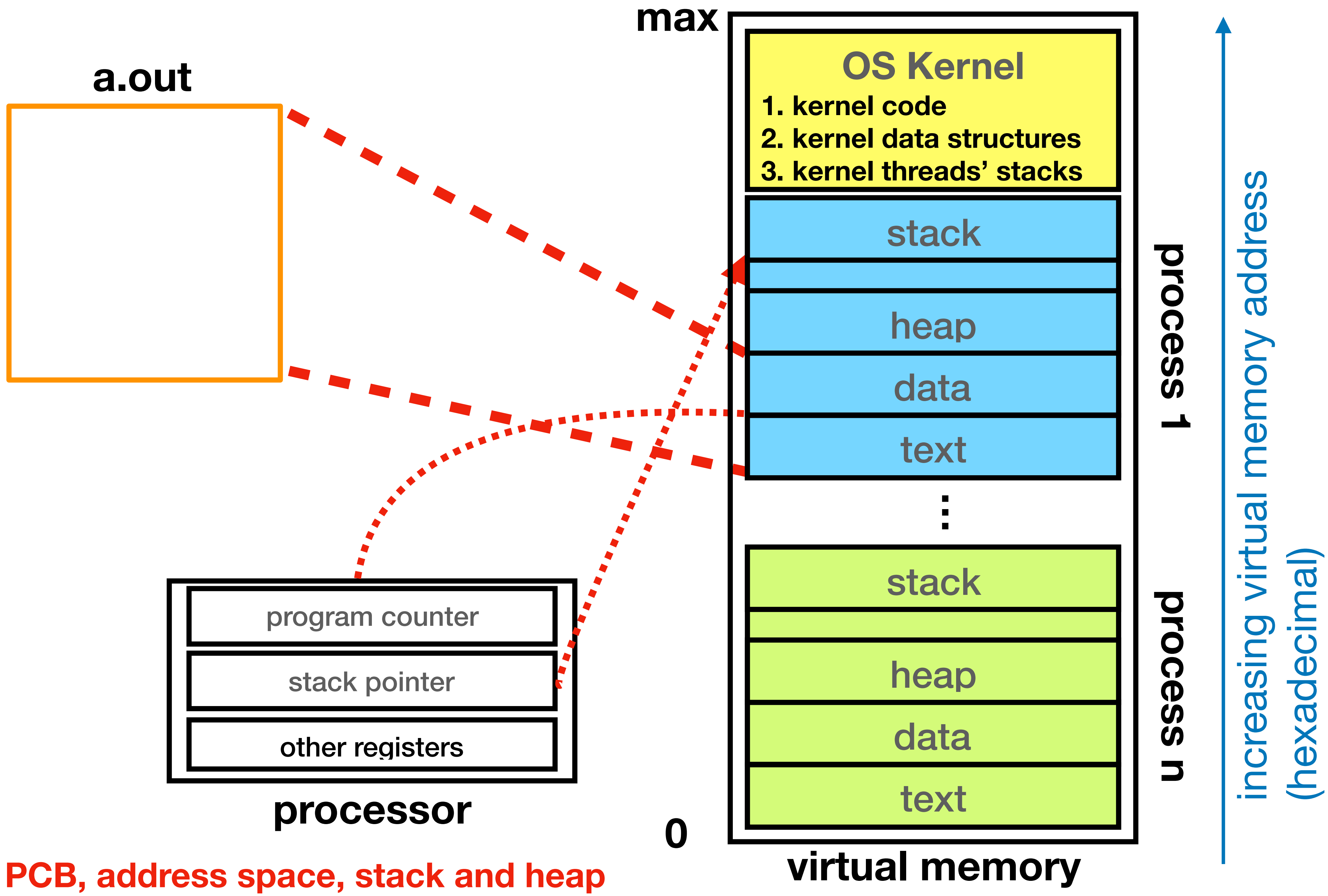
Loading — revisited



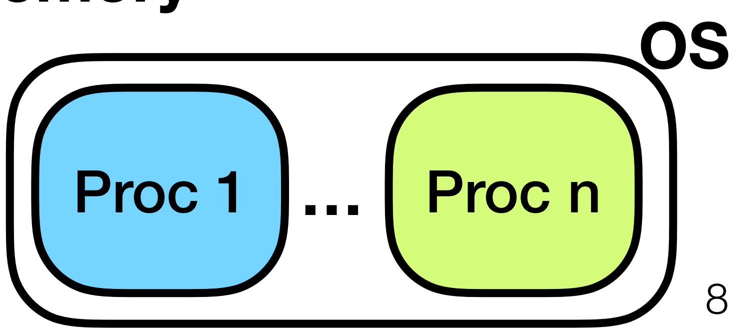
Loading — revisited



Loading — revisited



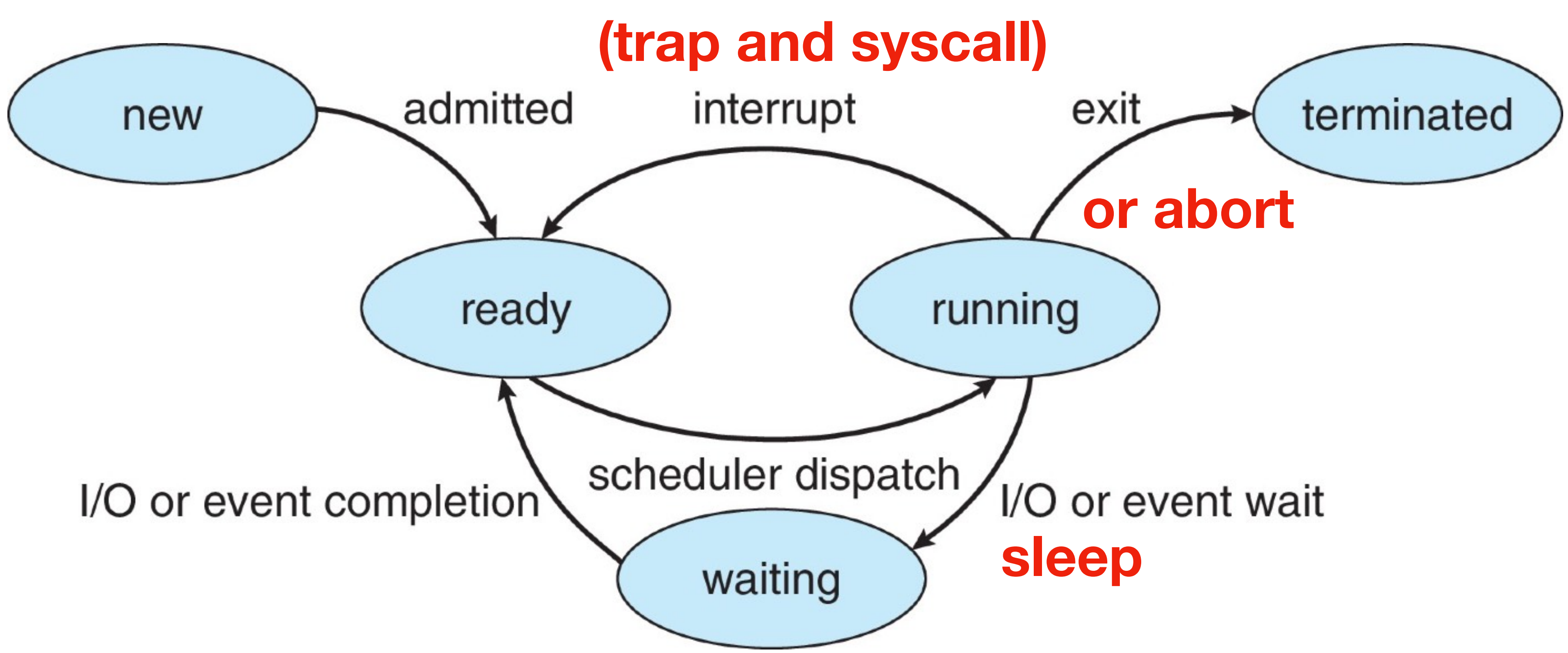
1. create PCB, address space, stack and heap
2. initialize registers and program counter
3. load instruction and data segments of executable file into memory



Keeping track of processes

- OS juggles many processes at a time
  - only one process can be running per core at a time (the kernel maintains a pointer to this process)
  - but many processes can be in ready and waiting states
- OS puts PCBs of the active processes in appropriate queues
  - ready queue (organized by the process-scheduling priority, the arrival time, etc.)
  - wait queue for each device
  - **zombie** queue (child processes terminated with no waiting parent)
- state change happens as a result of the process actions (e.g., termination or invoking system calls), OS actions (scheduling), and external actions (hardware interrupts)

Process lifecycle



Process lifecycle

new process created using `fork( )` and `exec( )`

`init` or `systemd` process is created at boot time