Reading Reference: Textbook 1 Chapter 2

INTRODUCTION TO UNIX PROCESS

Tanzir Ahmed

CSCE 313 Spring 2020

Process Definition

- Process is an instance of a running programming. This happens through its **Abstraction**
  - *Also defined as "A Program in Action"*
  - *Provided by the OS and used by the program*
- To the OS, Process is a **data structure (and more)** representing a running program
  - *Used to save a program's state*
  - *A program can be kicked out and restored using it*
- To the program, **Process** is a view of the entire system – memory, CPU, disk and all I/O devices
  - *Whatever the program needs, Process has it*
  - *Example: File handles, network socket etc. are kept inside the process*
  - *This view of the system is also called **machine state***

Process Data Structures

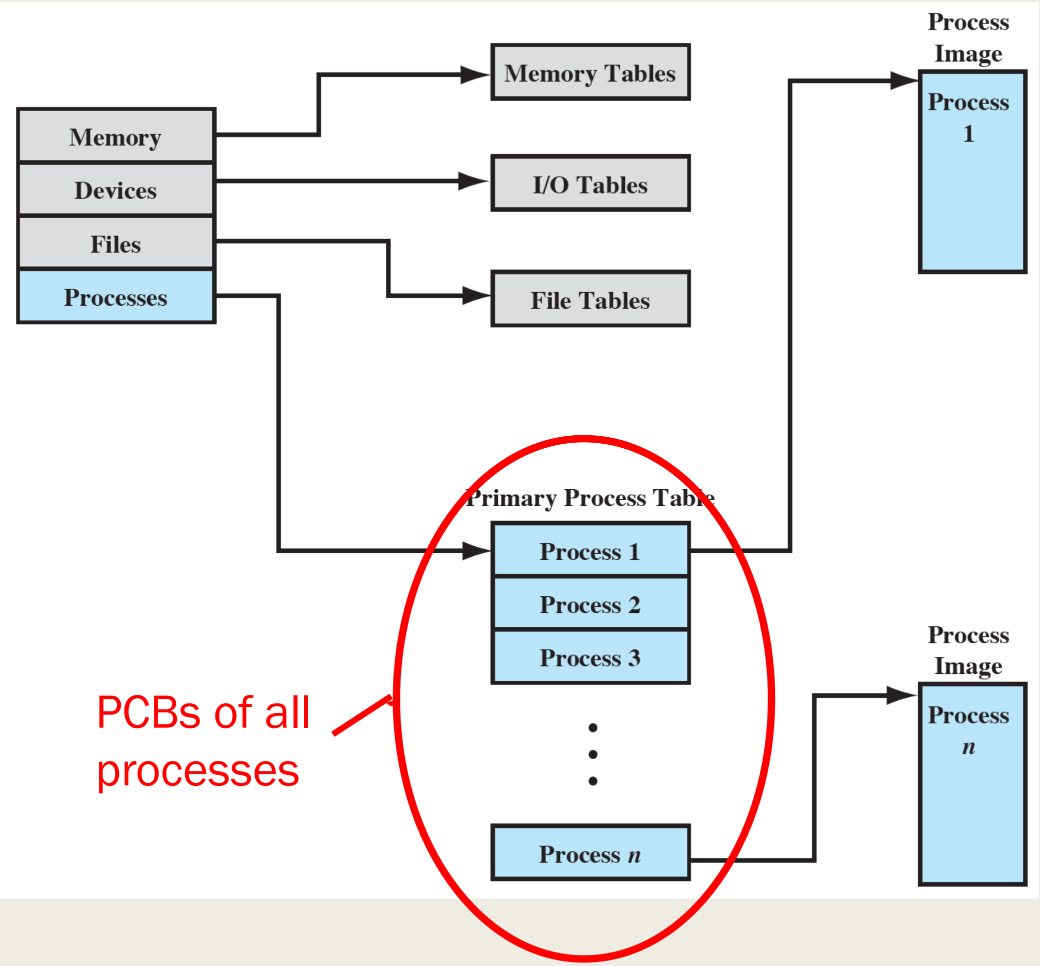
- How does the OS represent a process in the kernel?
  - *Using a data structure called Process Control Block (PCB)*
  - *The PCB is where the OS keeps of a process' hardware execution state (PC, SP, regs, open file handles etc.) when the process is not running*
    - This state is everything that is needed to restore the "*kicked out*" process in the same hardware configuration – as if nothing has happened
    - Required for transparent "kickout" and "bringing back in"
- The OS maintains a list of all Processes that are currently running (or some other relevant state)
  - *Sometimes the OS needs to enumerate through processes*
- Note that program data is not same as PCB
  - *PCB is more like a process's metadata*
  - *Program's data (variables, allocated memory) and code are kept elsewhere (i.e., address space)*

Process Control Block (PCB)

process identification *(use: to **locate** a processes)*	• process id • parent process id • user id
processor state information *(use: to **restore** a processes as it was)*	• register set • All general regs • Specials (e.g., PC, SP, EFLAGS) • condition codes • Overflow, jump to take or not • processor status
process control information *(use: to **treat/run** a processes appropriately)*	• process state • scheduling information • event (wait-for) • memory-mgmt information • owned resources (e.g., list of opened files)

OS’s Internal Tables

- An OS keeps a lot of information in the main memory
- In summary, much of this info is about:
  - *Resources (i.e., devices)*
  - *Running programs/processes*



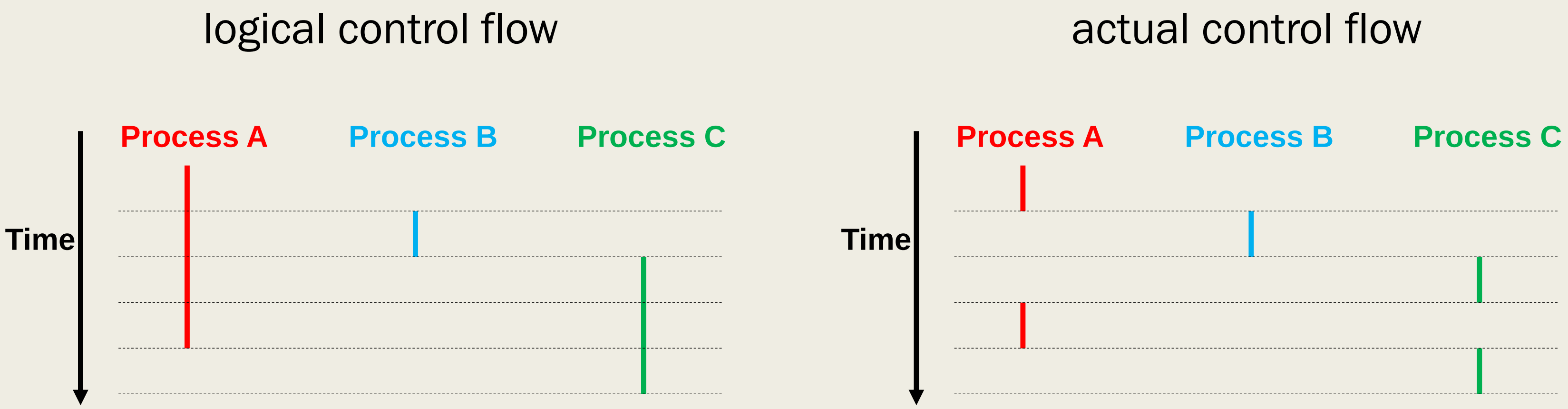
Source: OS – Internals and Design Principles

Abstraction Mechanism

- Process provides each program with two key abstractions:
  - ***Logical control flow*** *for* ***CPU virtualization***
    - Each program seems to have exclusive use of the CPU
  - ***Private address space*** *for* ***Memory virtualization***
    - Each program seems to have exclusive use of main memory
- How are these illusions maintained?
  - *Process executions* ***interleaved*** *(multitasking)*
  - *Private address spaces managed by virtual memory system (will describe that in a bit)*

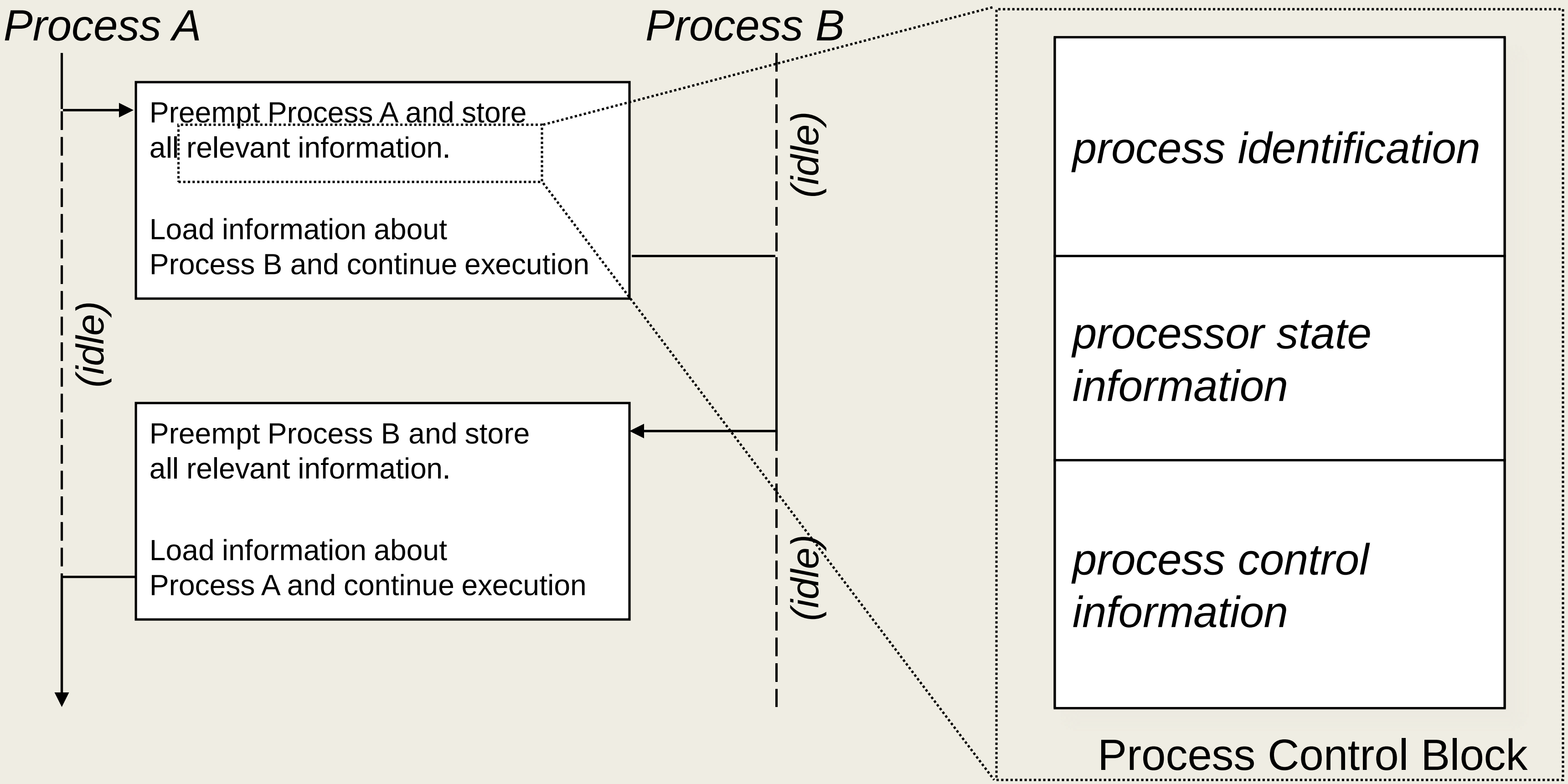
Logical Control Flows

- Each process has its own logical control flow, which can be far from reality



- Control flows for concurrent processes are physically disjoint in time (except on multi-core machines)
- However, we can think of concurrent processes as running in ‘parallel’ with each other

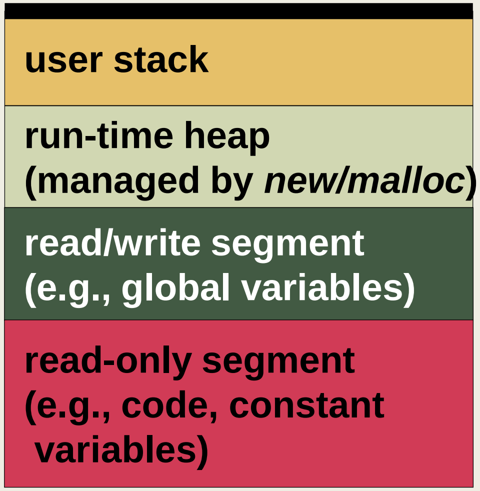
Logical Control Flow using Context Switch



Private Address Space

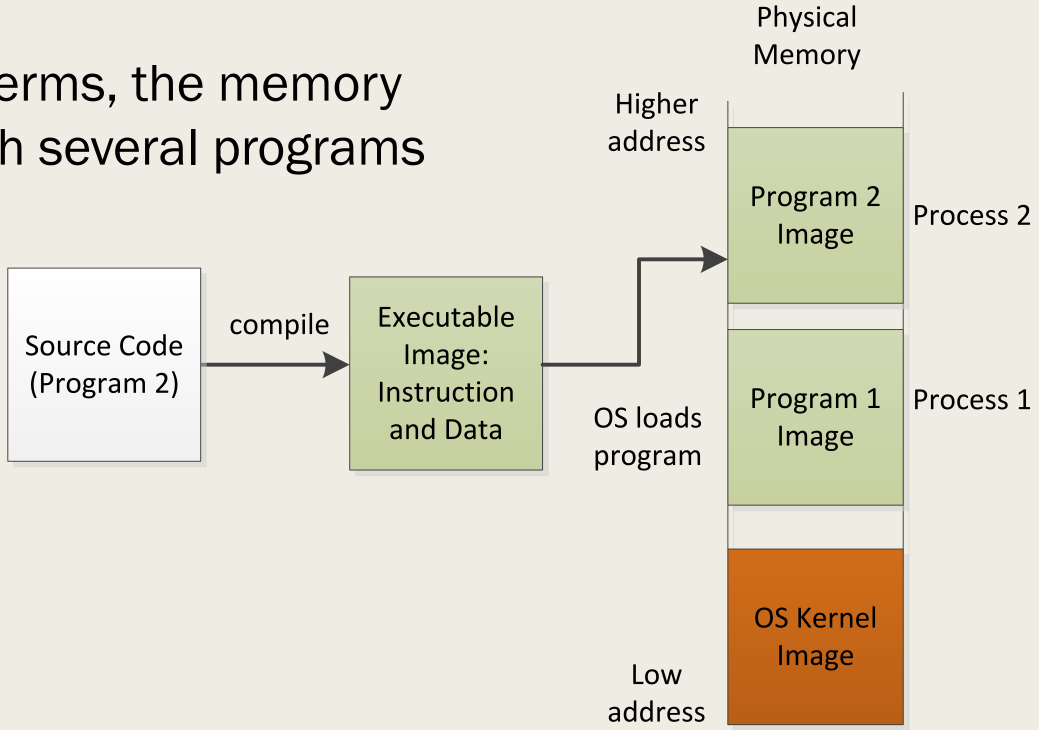
- First, every program needs to store and use some data
  - *Declaring variables, allocating memory etc.*
  - *A program's instructions also need to store*
- These data are kept in the disk, but need to be loaded to main memory before running the program
  - *Now, how and where do we load these data?*
  - *Answer: the "Address Space" of the corresponding Process*
  - *The address space is also "private" (i.e., isolated from other processes)*
- Let us see how it works

Address Space



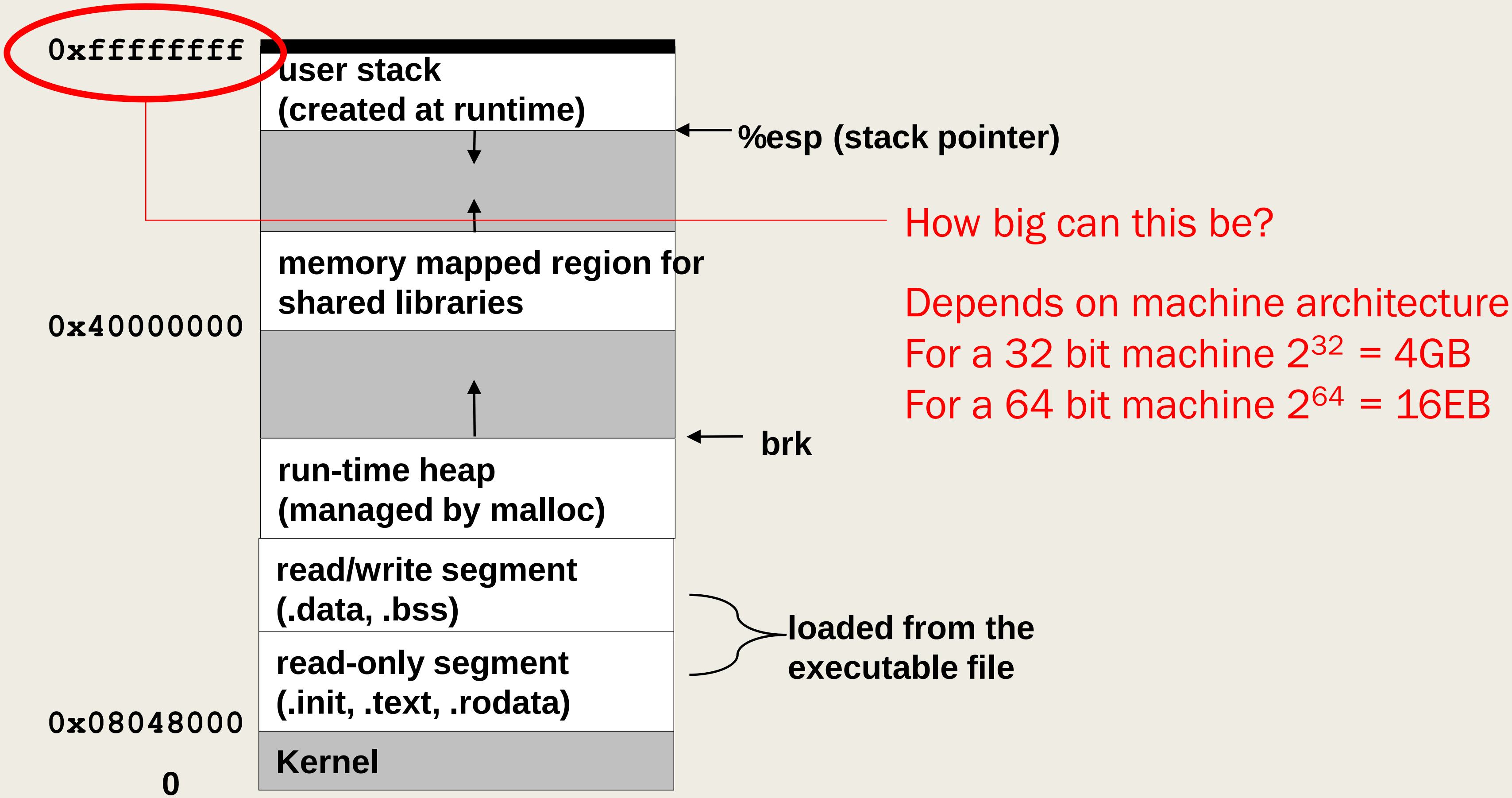
- A program’s data can be divided into different parts; therefore they are loaded as groups into memory

- Therefore, in very simple terms, the memory looks like the following with several programs loaded:



Private Address Space Illusion

- But each process is made to believe that the memory looks like the following – thanks to Virtual Memory:

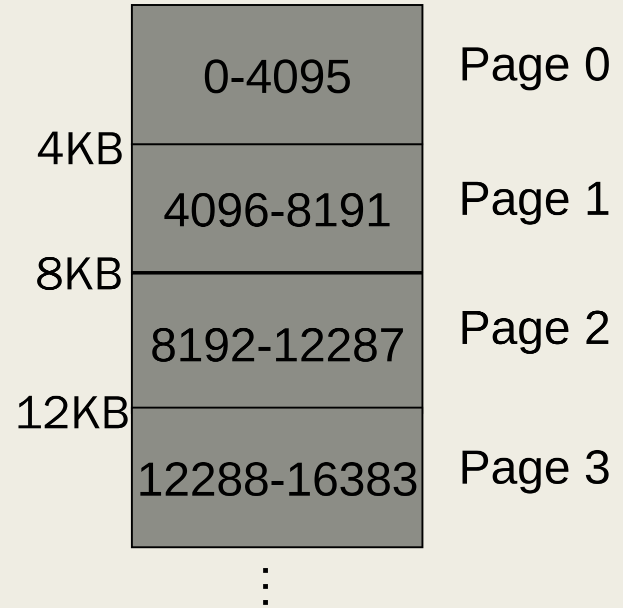


Question

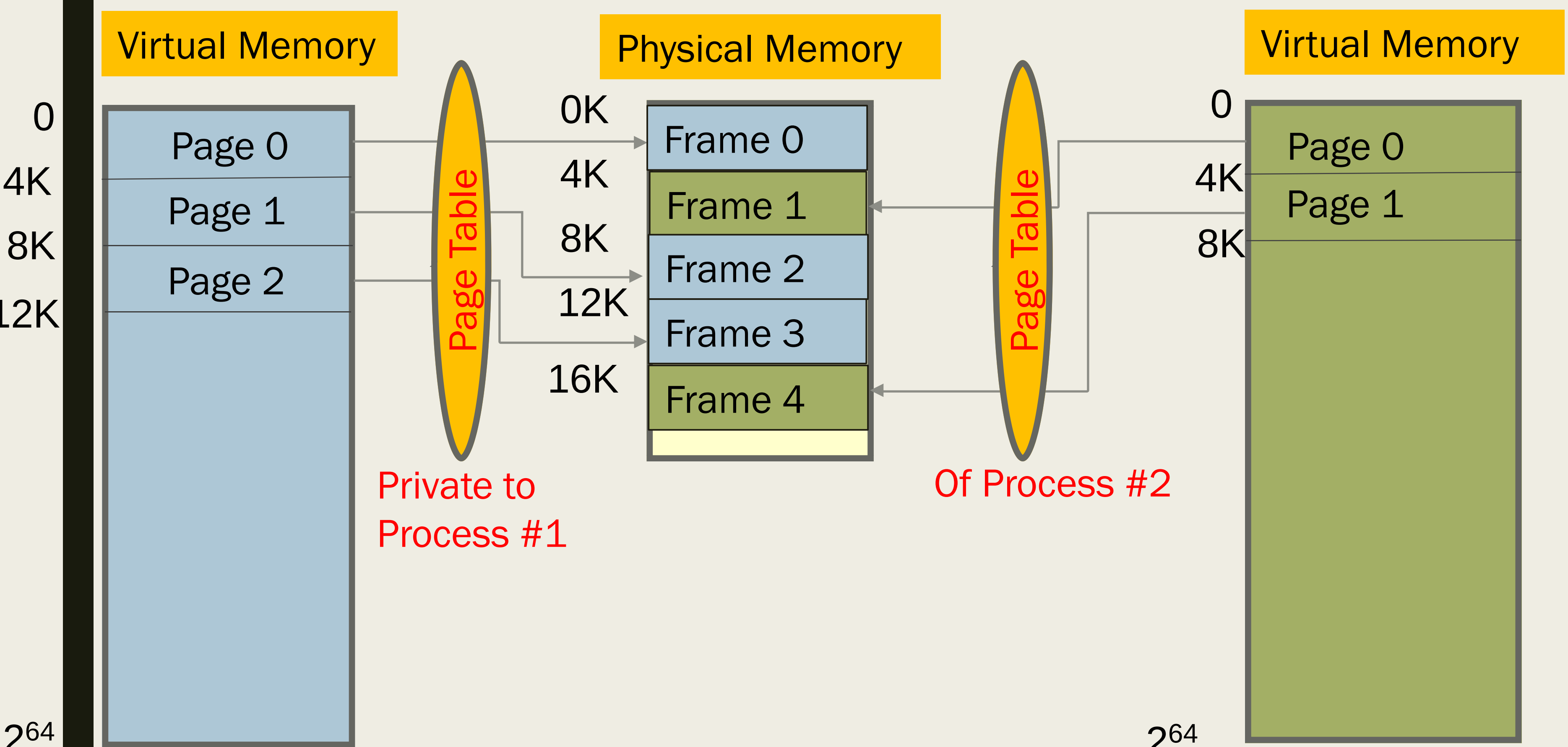
- Is a process image really 2^{32} = 4GB long?
  - *It would be nice for us, programmers*
  - *But typically, physical memory is quite limited*
  - *Virtual memory in action*
- What is then Virtual Memory?
  - *For that, we now need to a little understanding on how memory is organized in a system*

Memory Organization

- Memory is logically divided into *pages*, which are fixed in size and aligned regions of memory
  - *Typical size: 4KB/page*
- But pages are associated to a process only when necessary (i.e., read or written)
  - *When not necessary, they are evicted to the disk*
  - *Thus a running process can be stored back to disk again!!!!*
- The mechanism of such **lazy-allocation** is through **Page Fault**
  - *If a non-existent page is accessed (R/W), a page fault happens and fault-handler allocates the required page in physical memory*



Mapping from Virtual to Physical Memory



Summarizing Virtual Memory

- The private address space of a process is made up of pages
  - *These pages can be spread according to the wish of the kernel*
  - *Contiguous memory in processes' view are not necessarily contiguous in physical memory – they can be physically scattered, but **stitched** together by Virtual Memory system*
- Because memory is scarce, the whole private address space does not need to be allocated all the time:
  - *Actual pages can be allocated/mapped only when needed (i.e., read or written) through **page faults***
  - *Even allocated but inactive pages can be swapped back to disk to make room for more active pages*
- Memory accesses can be slowed down by Page Table accesses
  - *Each address must be translated to physical address by looking up in the process's page table, which is also in memory*
  - *Thus each memory access is actually **2 memory accesses**: 1 for page table, another for the actual memory access*
  - *There is a cache called Translation Lookaside Buffer (TLB) which stores popular translations – thus hiding the double latency*

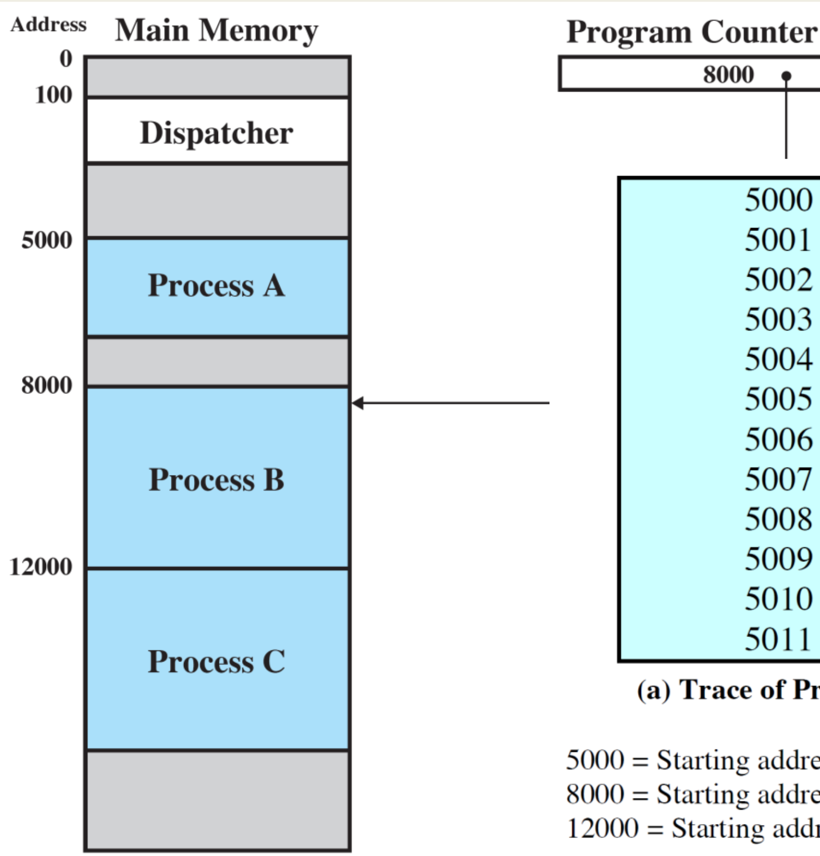
Virtual Memory – An Interesting Video

- Please watch this video, which is very informative, yet small:

https://www.youtube.com/watch?v=qlH4-oHnBb8

Process States - An Example

- Assume the following processes A, B, C are loaded in memory
  - *Assume we are not using Virtual Memory in this case*
  - *The processes are completely loaded in memory*



(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A (b) Trace of Process B (c) Trace of Process C

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Process Trace

- **Trace:** The sequence of instructions that execute for a process

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
--------------------Timeout	
7	100
8	101
9	102
10	103
11	104
12	105
13	8000
14	8001
15	8002
16	8003
-------------- I/O Request	
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003
27	12004
28	12005
--------------------Timeout	
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
--------------------Timeout	
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
--------------------Timeout	

Process Trace Discussion

- Dispatcher is show in blue shaded box
  - *The same dispatcher code runs between any 2 processes*
- The figure shows several (52) instruction cycles from the 3 processes
- The trace starts with process A by overwriting PC with 5000, which is the first instruction of A
- The OS allows exactly 6 instruction before the timer fires
  - *i.e., kicks out the currently running process*
  - *This prevents programs from monopolizing the CPU*
- CPU goes from A to B after A is kicked out for the timer
- B gets kicked out because of requesting I/O
- Then the CPU alternates between A and C because B is still waiting for the I/O

Process States

- Let us start with elementary 2-state model
- This queue is some sort of a priority queue,
  - *priority is based on some scheduling metric*
- Typical content of the Queue:
  - *Pointer to PCB*
- Limitation:
  - *Does not handle I/O operation well*
  - *Can bring a process that is still waiting on I/O*
  - *The process will stay idle*

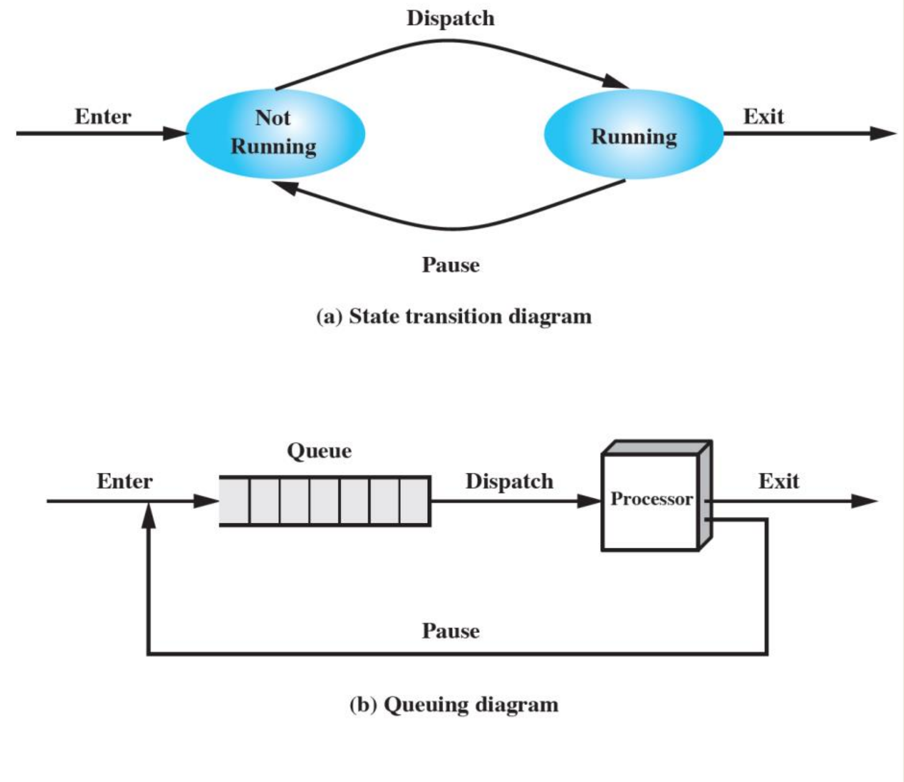
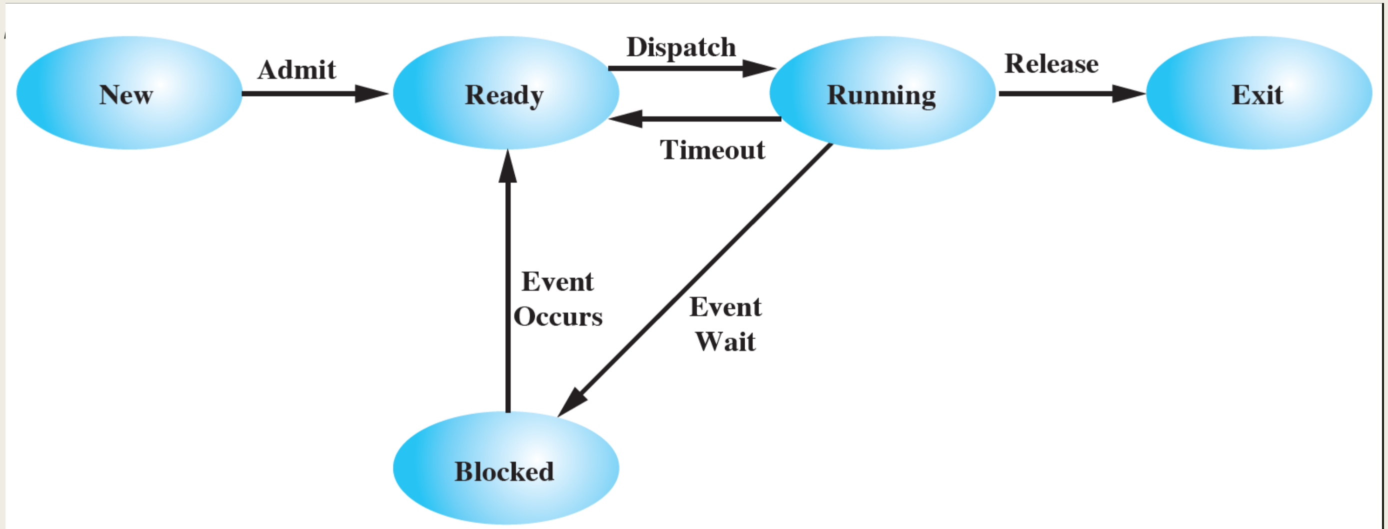


Figure 3.5 Two-State Process Model

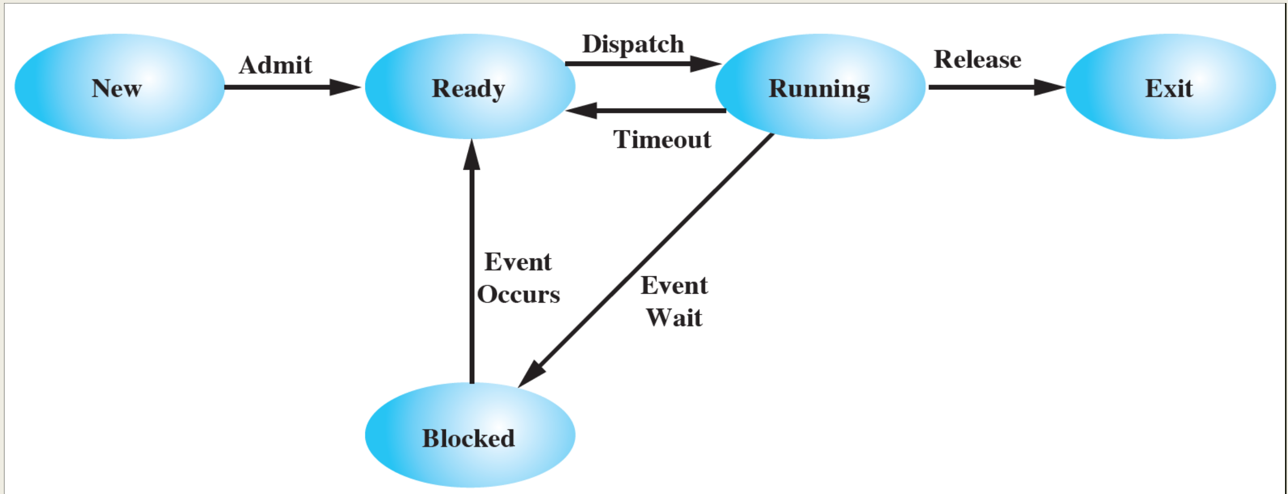
Refined Process State Model

- It has 5 states
  - *Newly added states: “New”, “Blocked”, “Exit”*
- Addition of “Blocked” state is obvious
  - *Avoids bringing the process back to ready queue before I/O operation or the event actually finished*
- “New” state is when the PCB is created, but the process is not loaded in memory yet
  - *Either because it is the usual delay*
  - *Or*

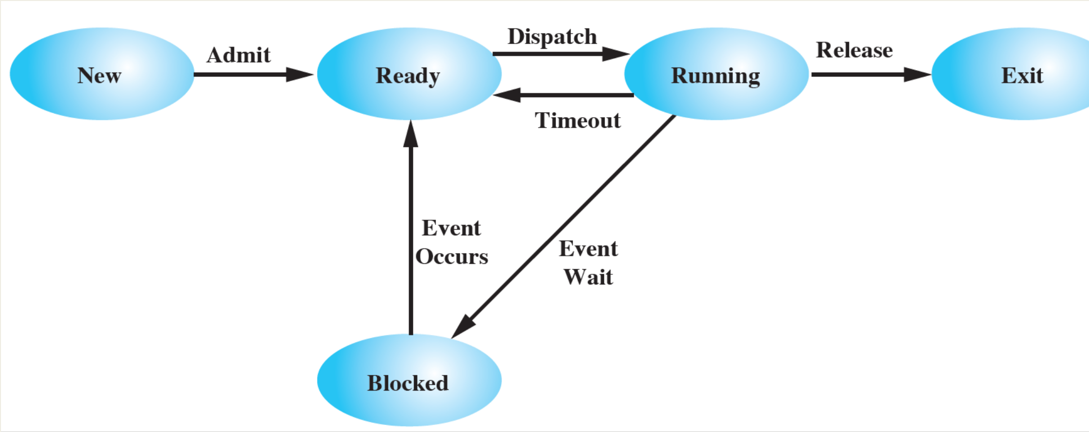


Refined Process State Model (2)

- Exit State: The process is removed from the list of executable processes, but it is held by the OS for collecting some information about it
  - *E.g., an accounting program collecting information about all processes*
- New State: Is used for memory management
  - *There might a limit on how many processes can be there in the main memory*
  - *The process's PCB is in memory, but not its executable image*



Now, the Transitions



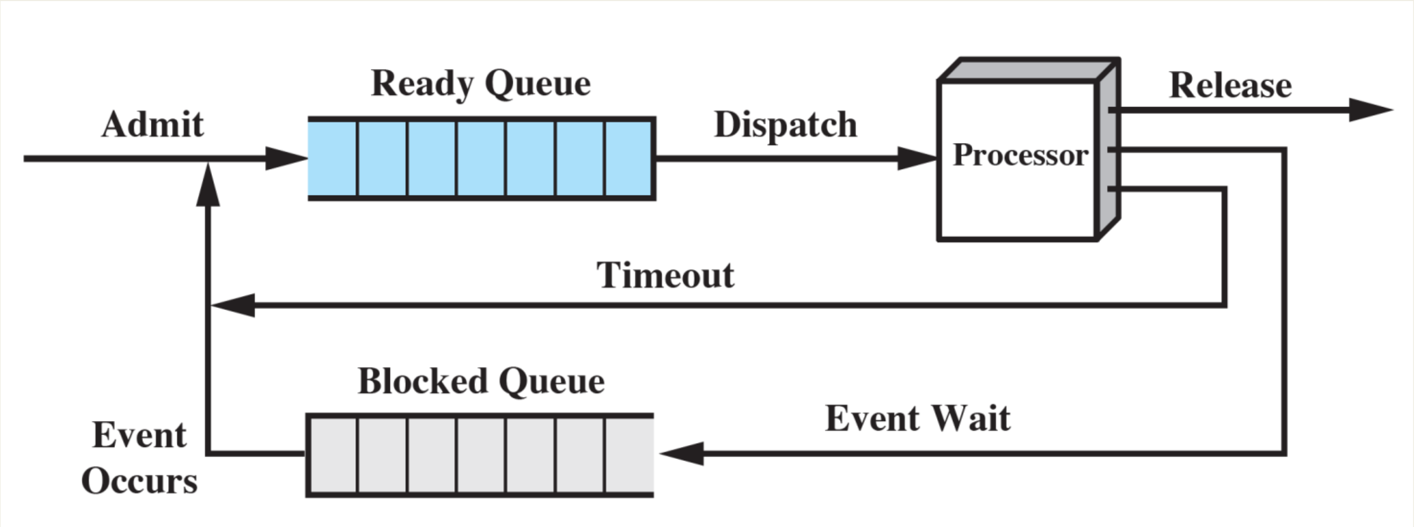
- New->Ready:
  - *When there is room for a new process in memory*
- Ready->Running:
  - *When the scheduler picks this process*
- Running->Exit:
  - *Normal or due to some unavoidable/unrecoverable error (e.g., segmentation fault, divide by 0)*
- Running->Ready:
  - *Timer fired*
  - *A low-priority process is running in CPU and a higher priority process got unblocked for I/O finish*

Transitions

- Running->Blocked
  - *I/O or other event request that are not ready, or would take time*
- Blocked->Ready
  - *Event on which the process is waiting has finished*
- Ready->Exit
  - *Parent process terminates child before it could even run*
- Blocked->Exit
  - *Parent process killed the child while it was waiting*

Queueing Model for Proc. States

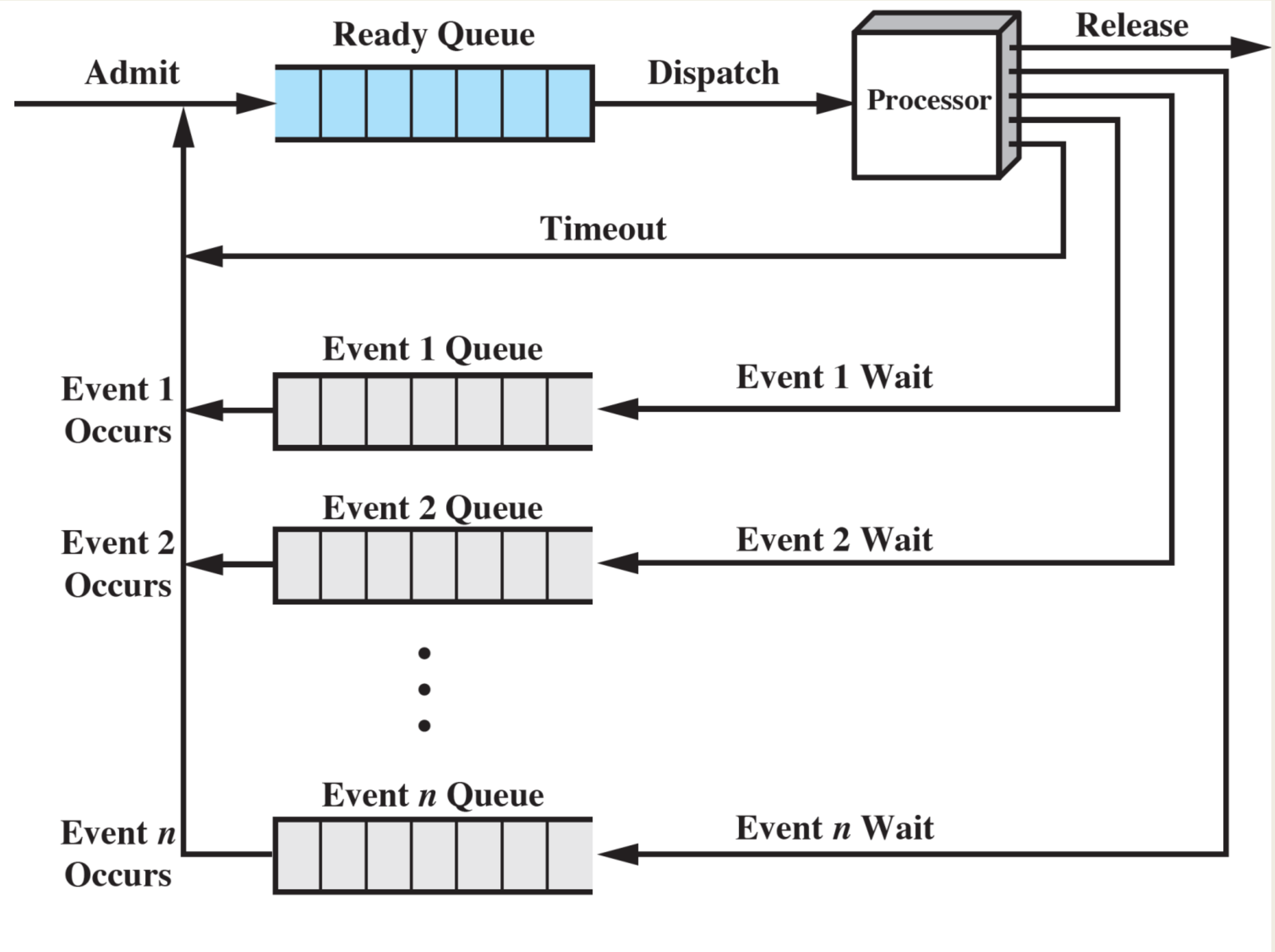
- To implement the 5-state model, we will now need 2 queues



(a) Single blocked queue

- However, 2-queue model is also not enough
  - *Does not make sense to make a file request from disk and URL request behind the Network card wait in the same queue*
- In reality, each I/O device, each lock, and thus each event needs a separate wait queue

Queueing Model (2)



(b) Multiple blocked queues