IN1011

Operating Systems

Lecture 02: Processes

Today’s Questions

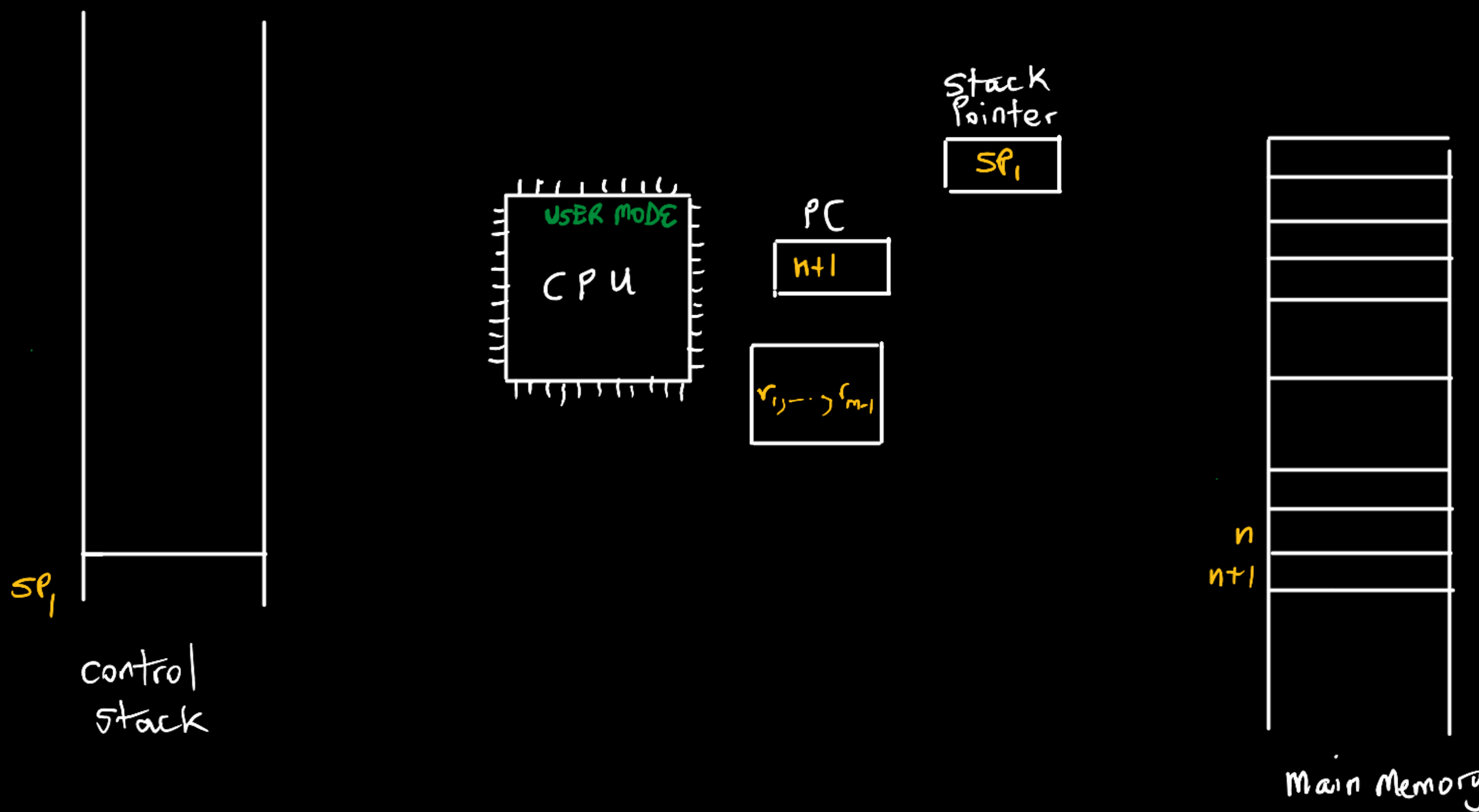
- What happens when a program’s execution is interrupted?
- How does the OS interleave the execution of multiple programs?
- How does the OS ensure new processes are not waiting while the CPU is idle (faster response times)?
- How can limited RAM be shared among interleaved processes?
- What does a process look like to an OS?
- What does an OS do when creating a process?
- Is the OS a process?

Questions

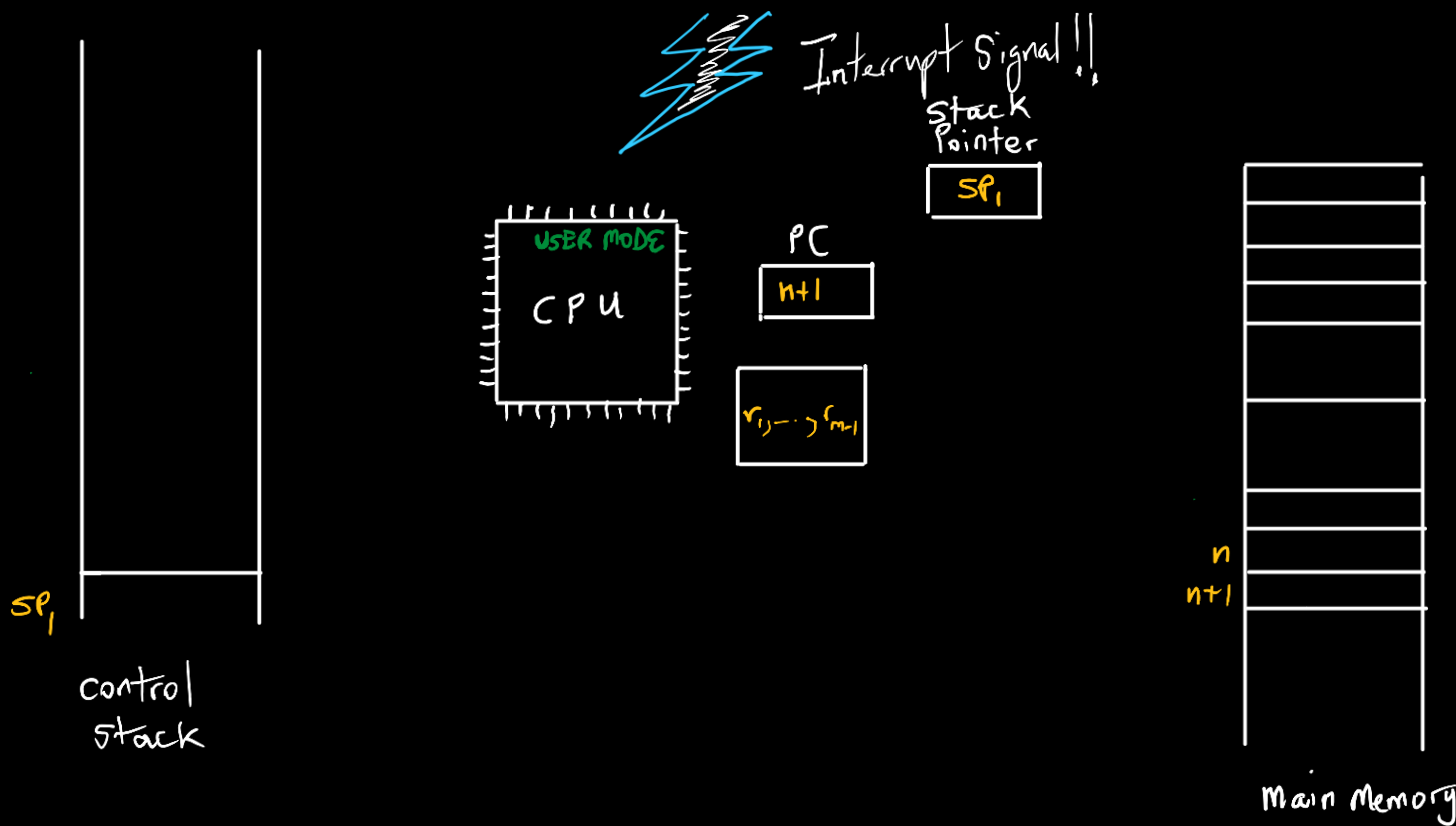
- What happens when a program's execution is interrupted ?

“Interrupting” Processes

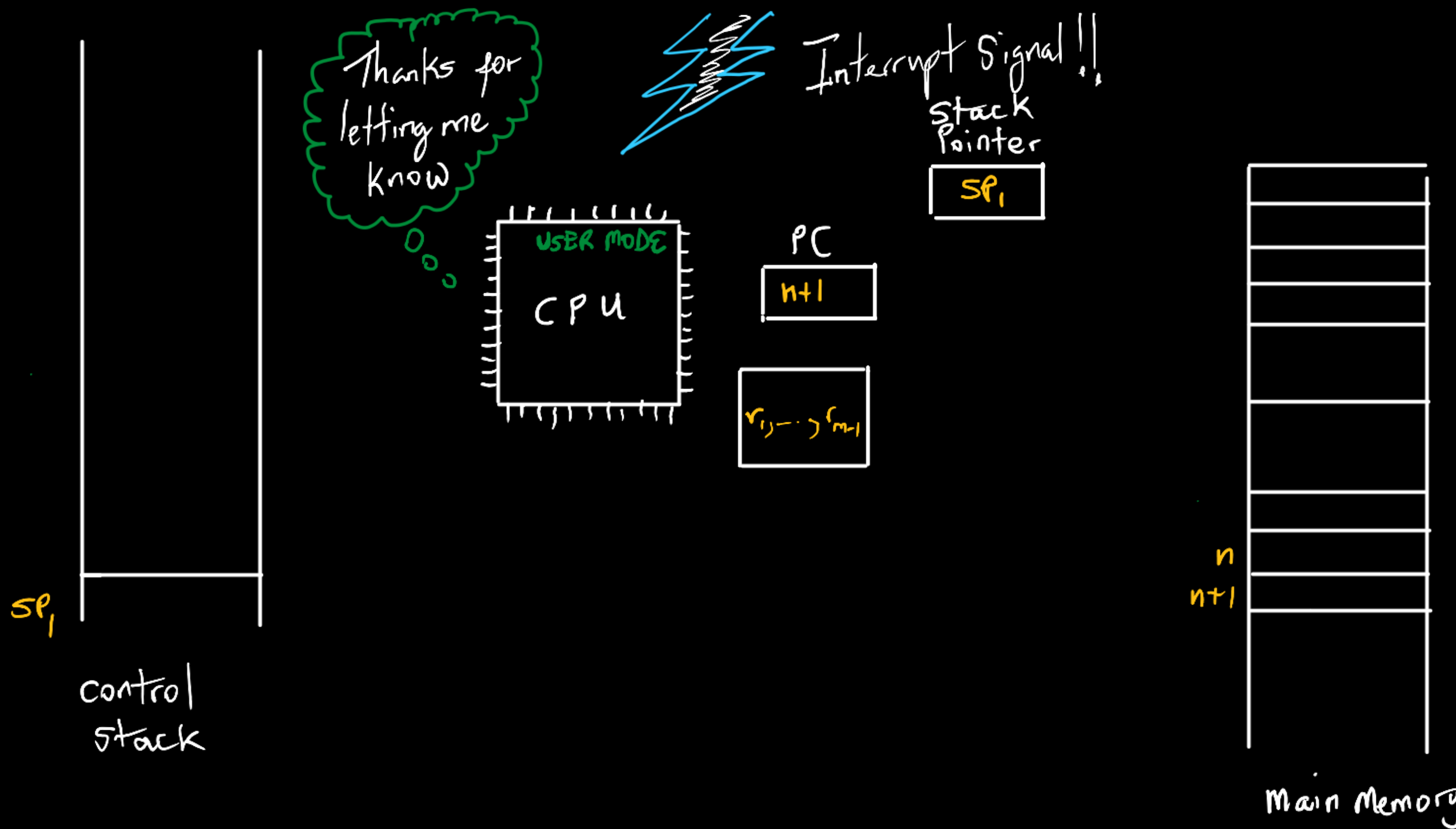
Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function



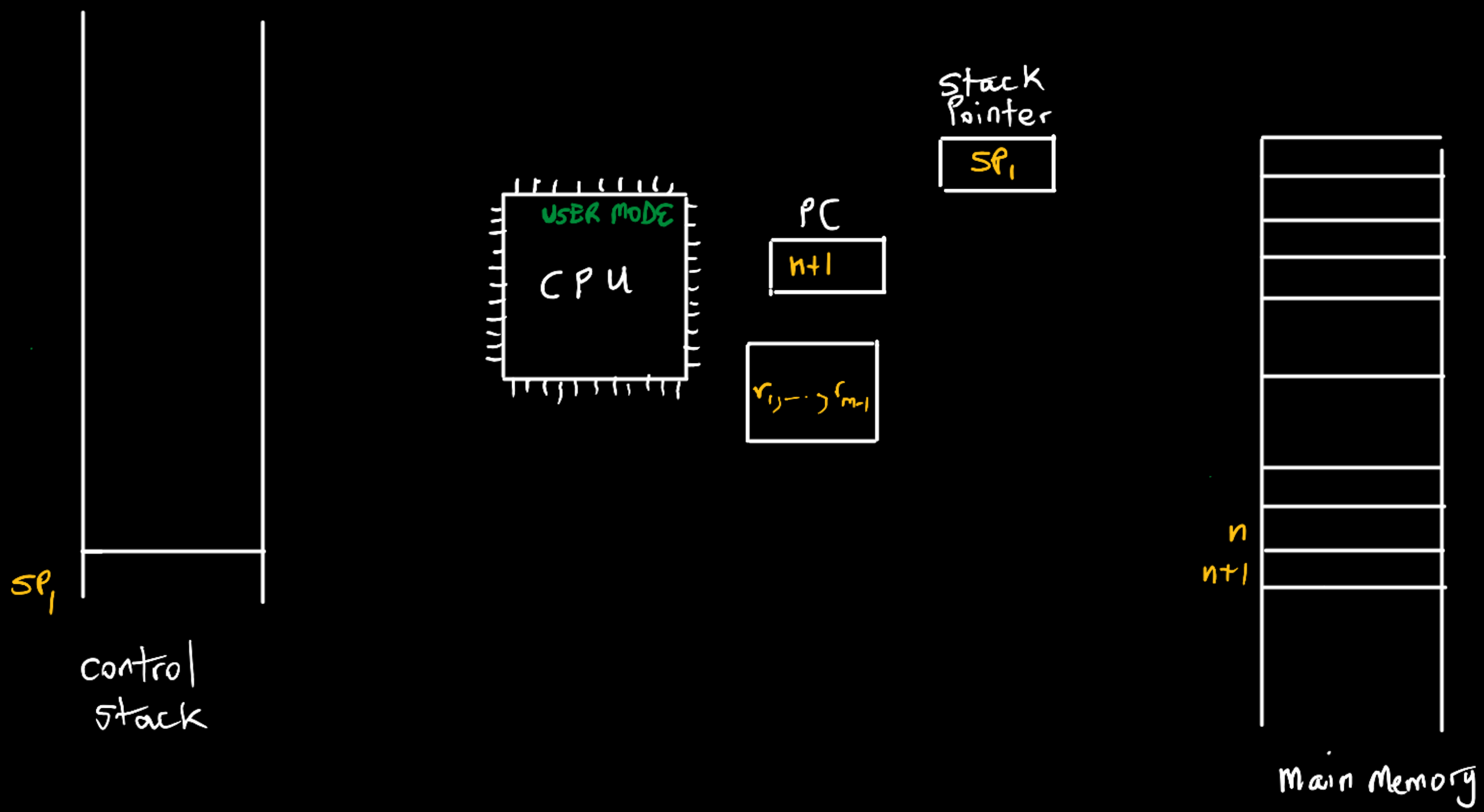
Stack Pointer
SP1
USER MODE
CPU
PC
n+1
r1, ..., rm-1
SP1
control stack
n
n+1
Main Memory

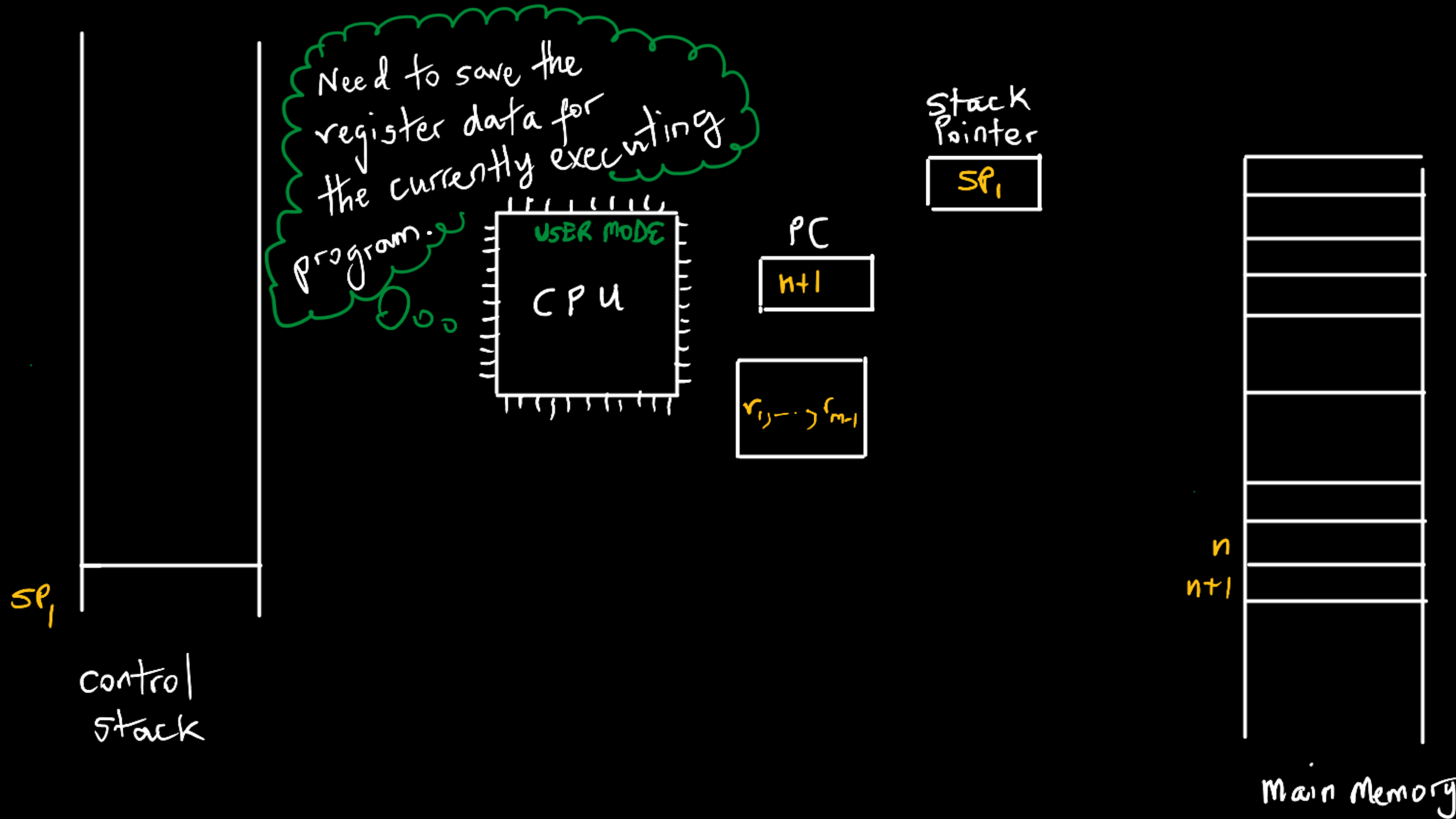


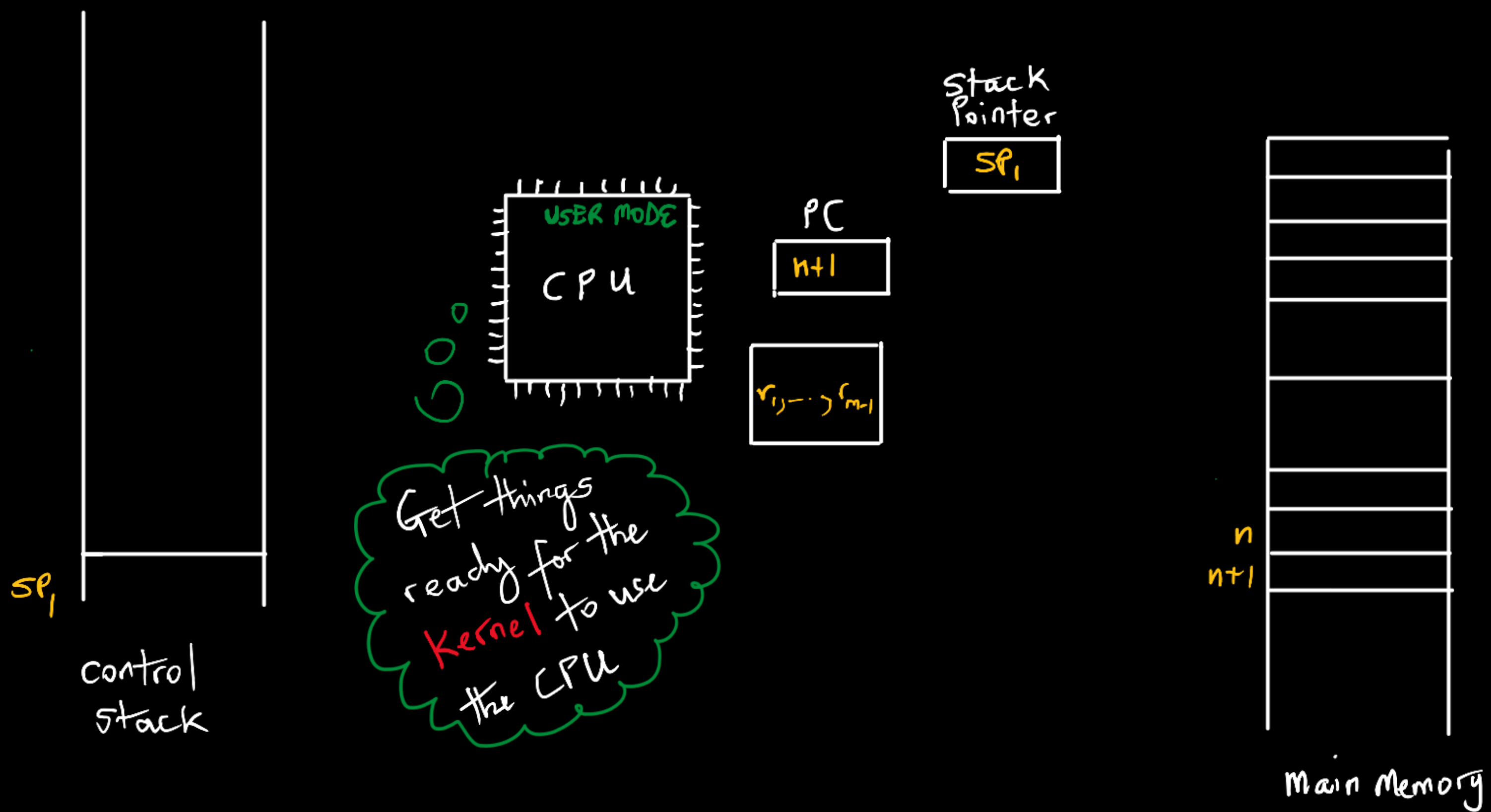
Interrupt Signal !!
Stack Pointer
sp_1
USER MODE
CPU
PC
n+1
$r_1, \ldots, r_{m-1}$
sp_1
control stack
n
n+1
Main Memory

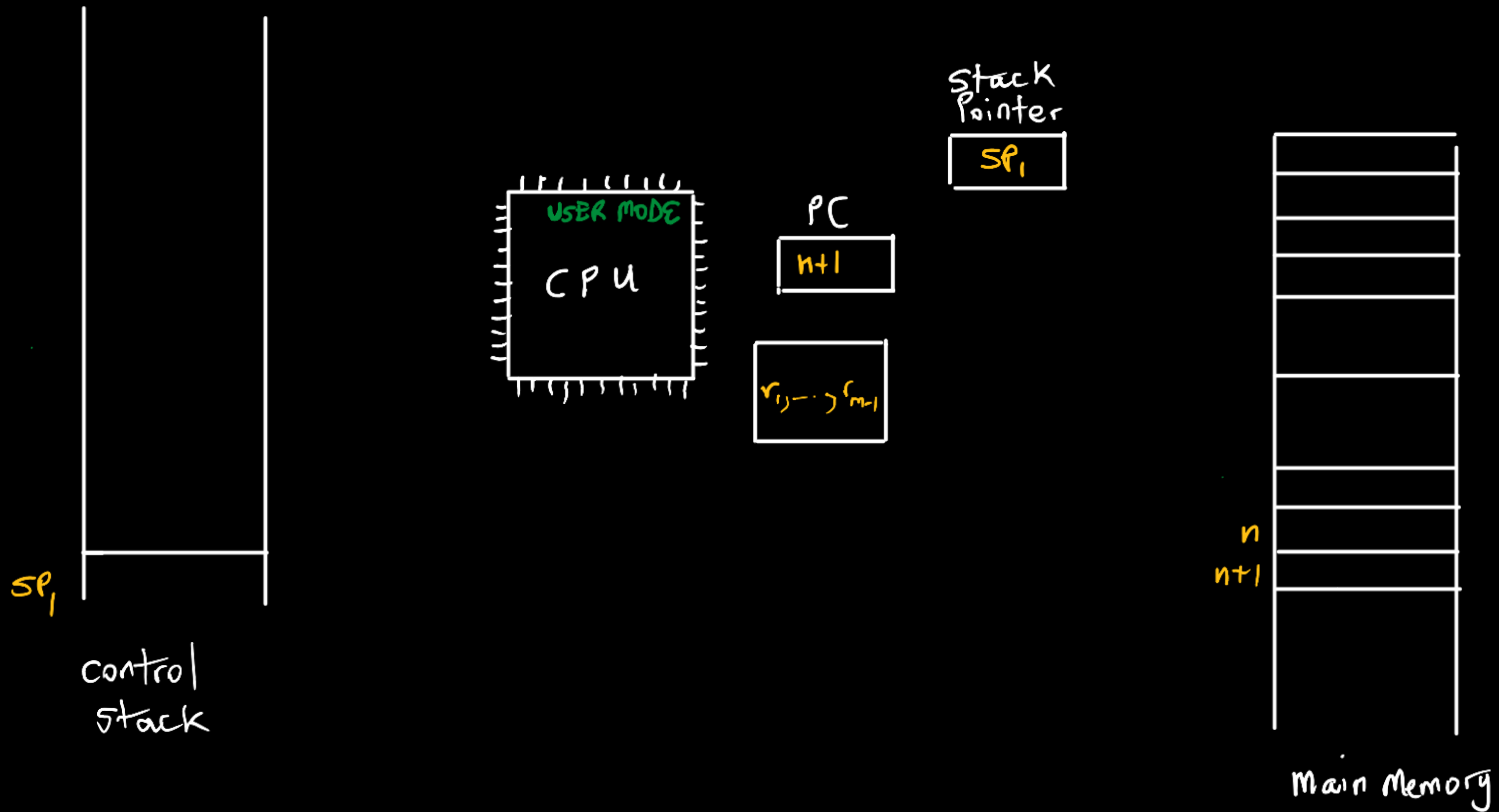


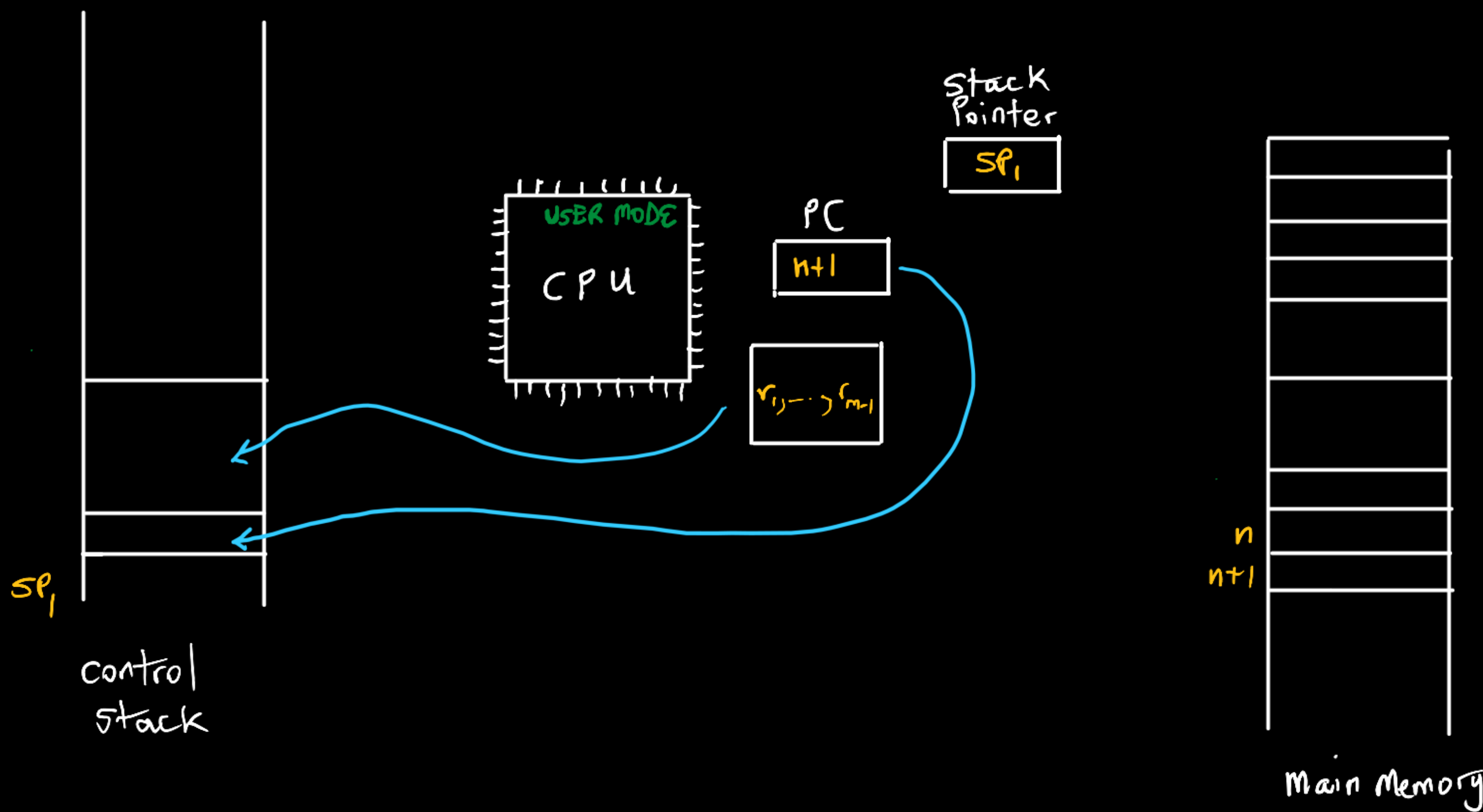
Thanks for letting me know
Interrupt Signal!!
Stack Pointer
SP1
USER MODE
CPU
PC
n+1
r1, ..., rm-1
SP1
Control Stack
n
n+1
Main Memory

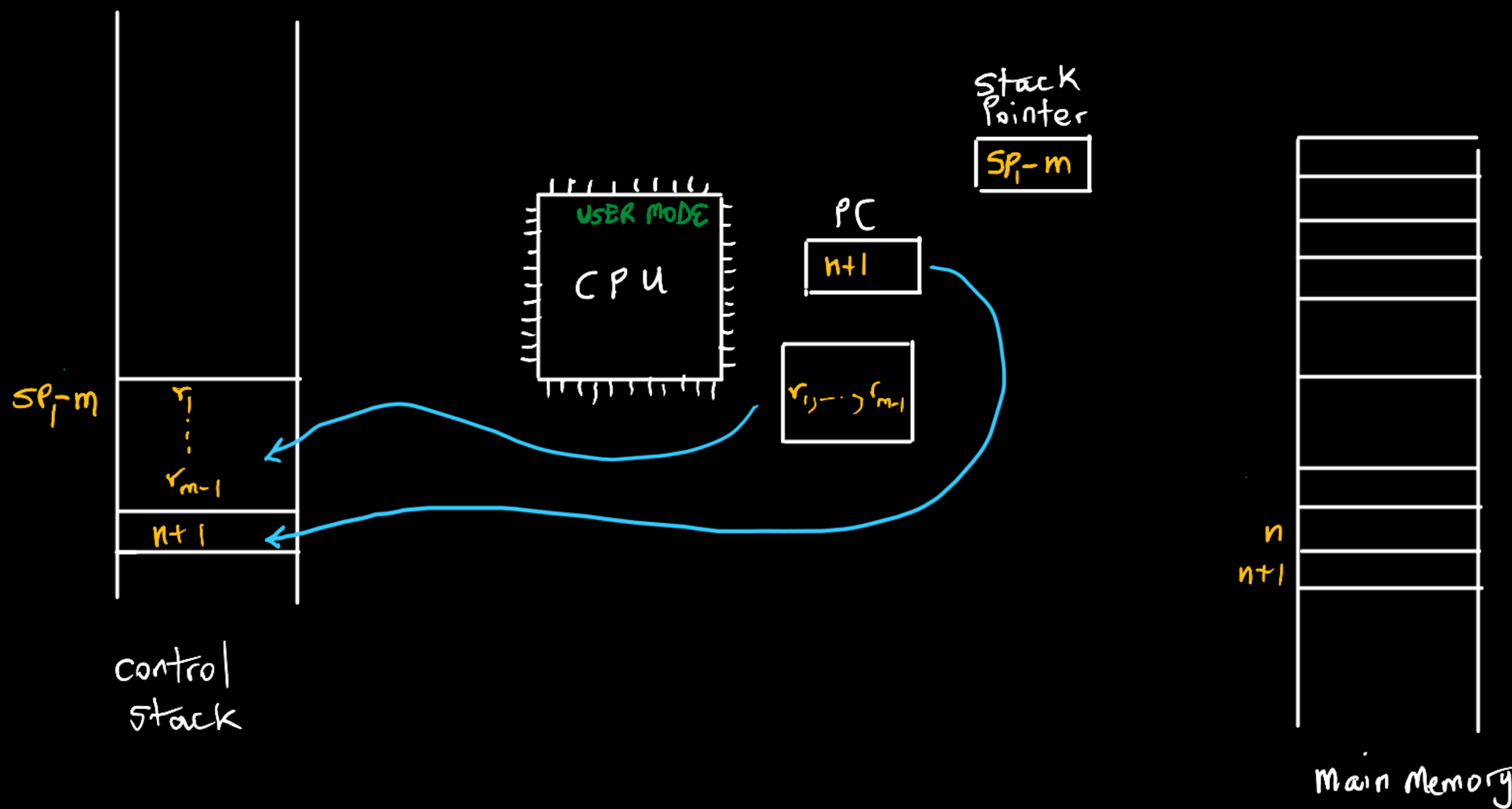

SP1
control stack
USER MODE
CPU
PC
n+1
r1, ..., rm-1
Stack Pointer
SP1
n
n+1
Main Memory


Need to save the register data for the currently executing program.
USER MODE
CPU
PC
n+1
Stack Pointer
SP1
r1, ..., rm-1
SP1
control stack
n
n+1
Main Memory

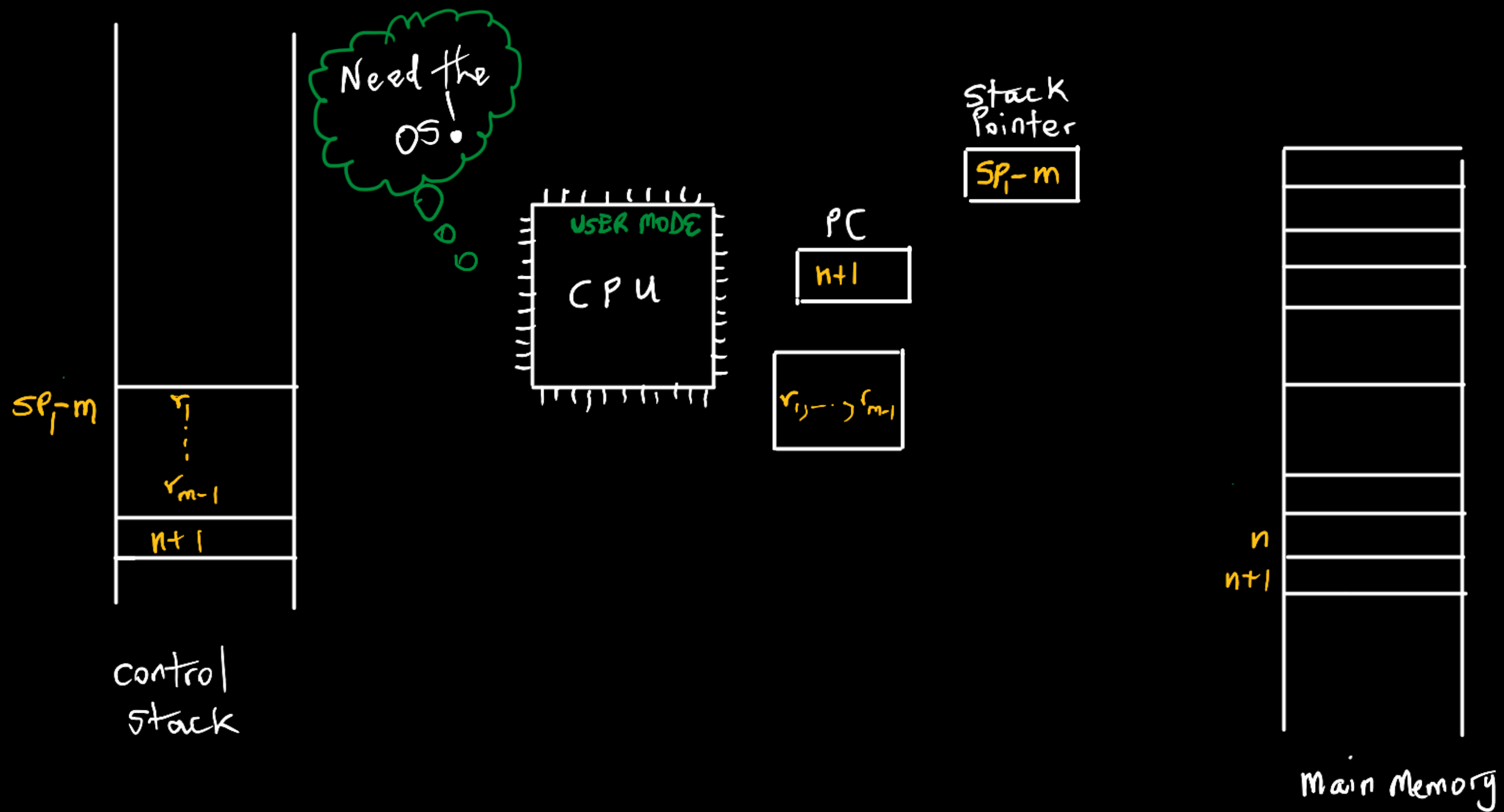

Stack Pointer
SP_1
USER MODE
CPU
PC
n+1
$r_1, \ldots, r_{m-1}$
Get things ready for the Kernel to use the CPU
SP_1
control stack
n
n+1
Main Memory


Stack Pointer
sp1
PC
n+1
USER MODE
CPU
r1, ..., rm-1
sp1
control stack
n
n+1
Main Memory

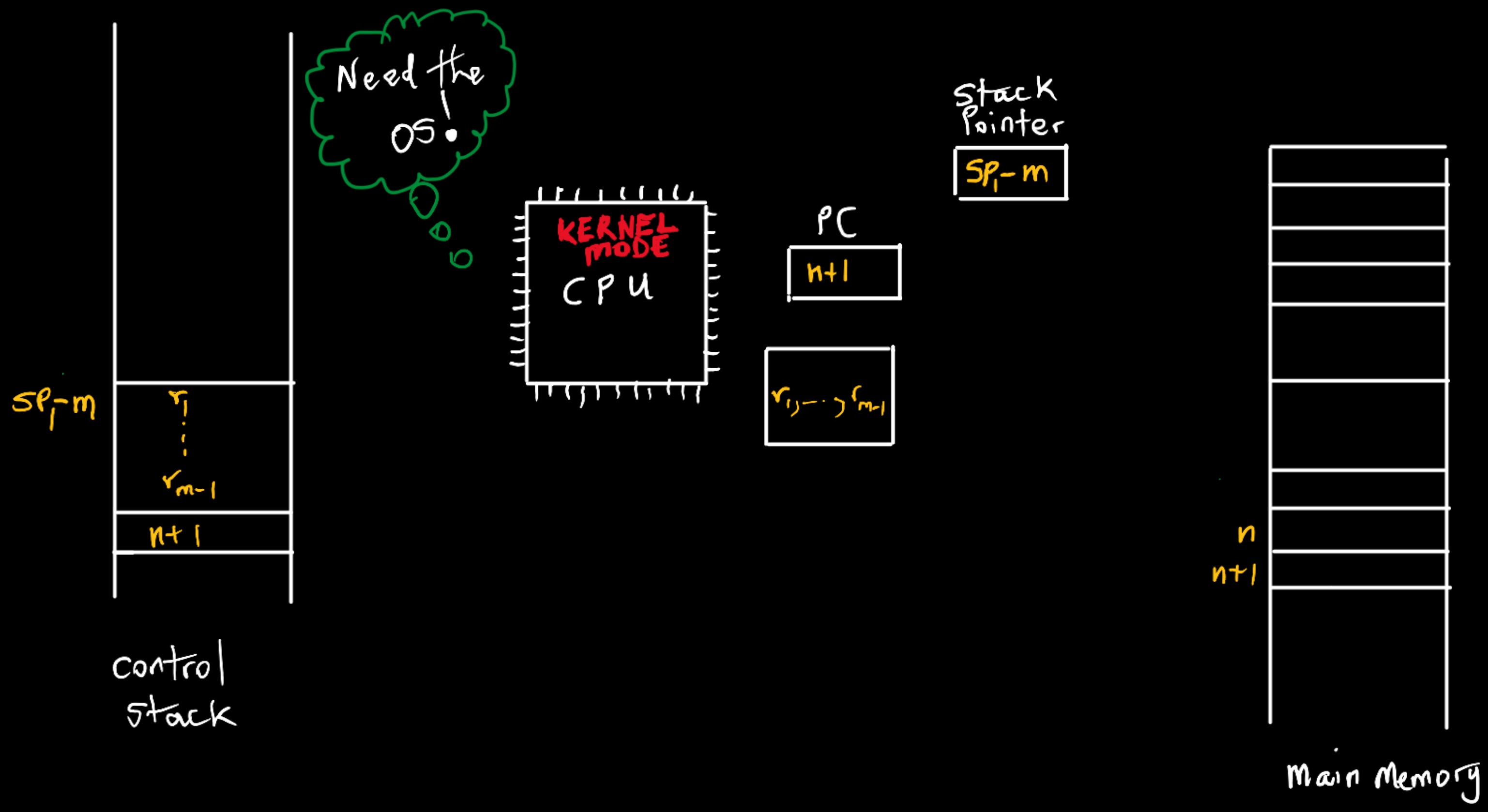

Stack Pointer
SP_1
USER MODE
CPU
PC
n+1
$r_1, \ldots, r_{m-1}$
SP_1
control stack
n
n+1
Main Memory



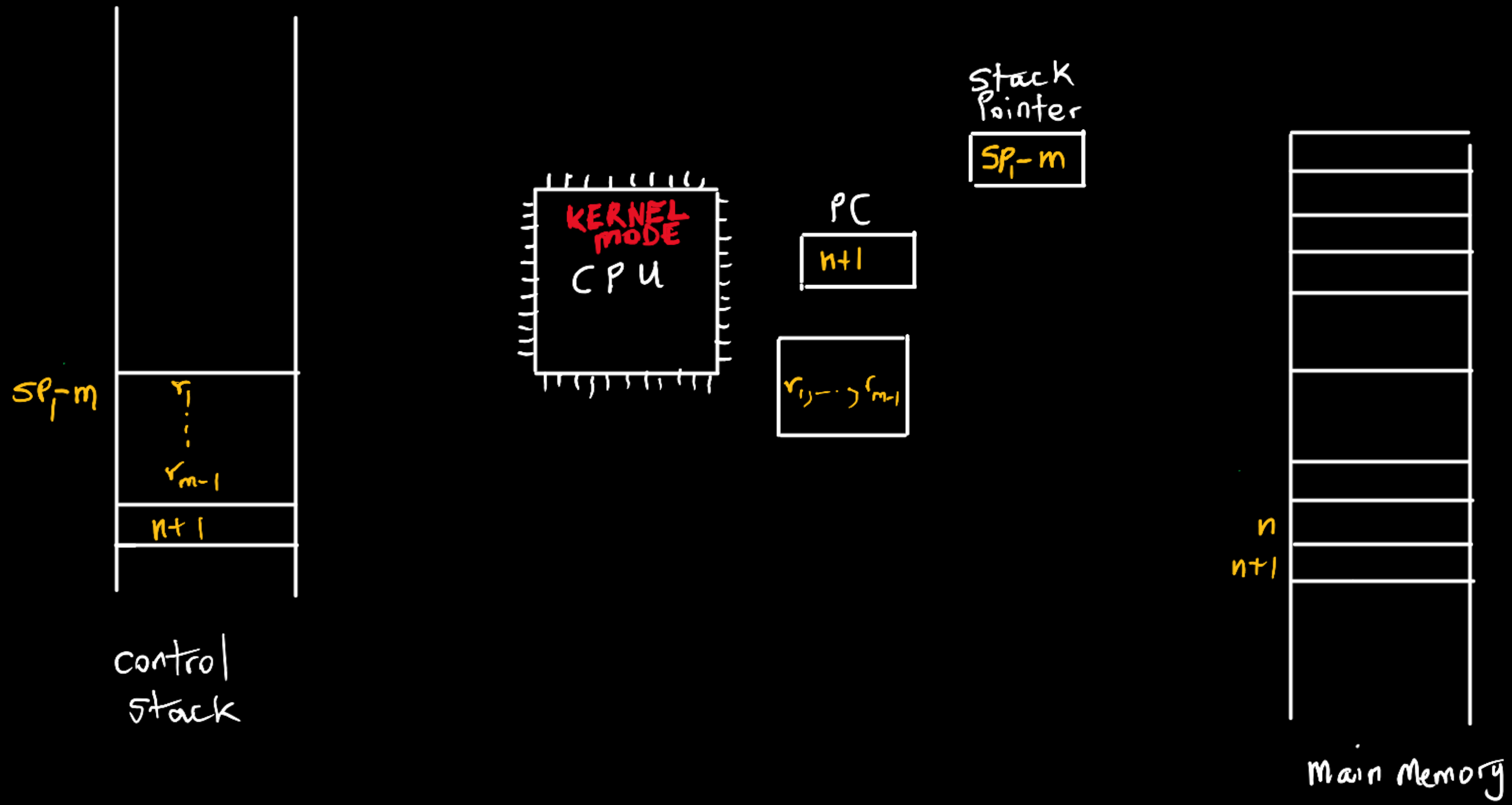
USER MODE
CPU
Stack Pointer
$SP_1 - m$
PC
$n+1$
$r_1, \ldots, r_{m-1}$
$SP_1 - m$
r_1
r_{m-1}
$n+1$
control stack
n
$n+1$
Main Memory

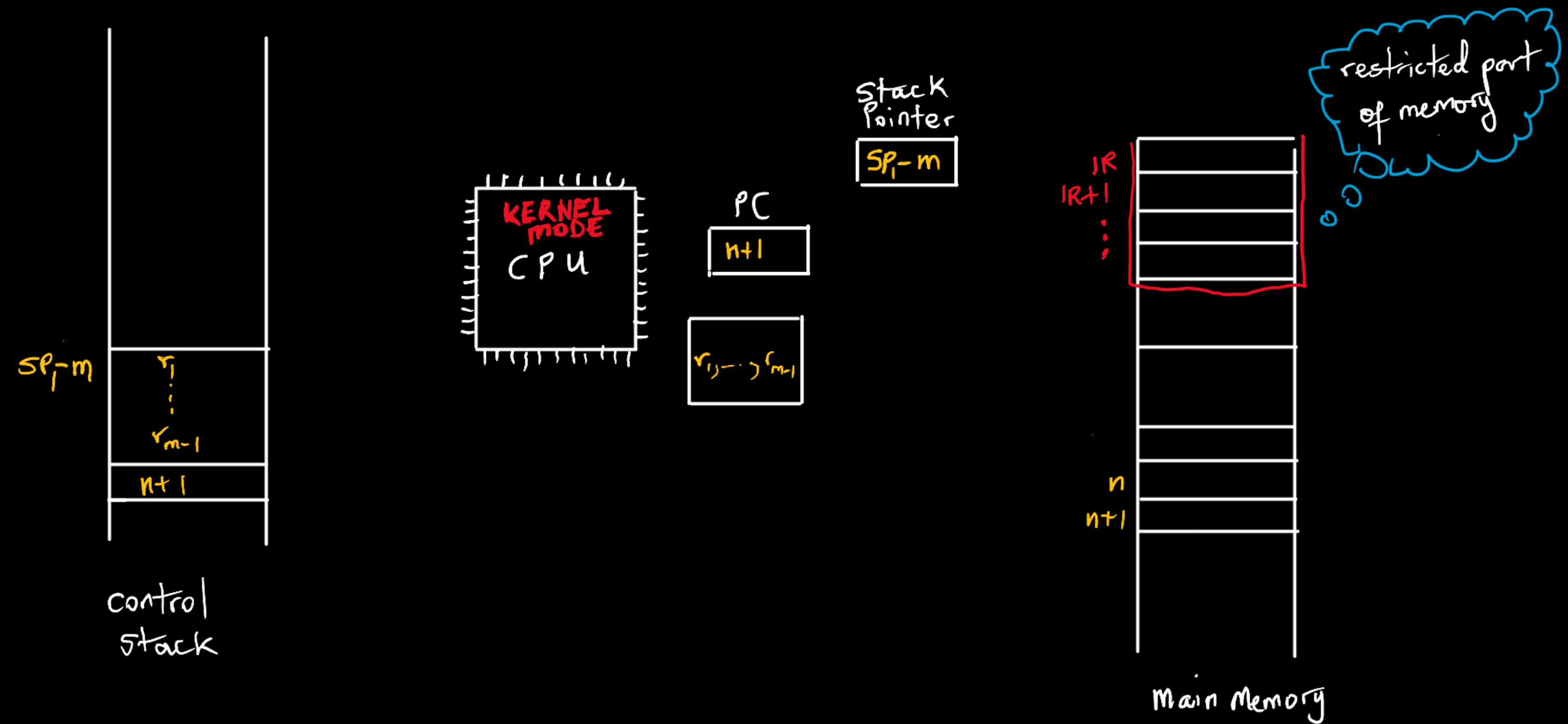


Need the OS!
USER MODE
CPU
PC
n+1
Stack Pointer
SP1 - m
r1, ..., rm-1
SP1 - m
r1
rm-1
n+1
Control stack
n
n+1
Main Memory

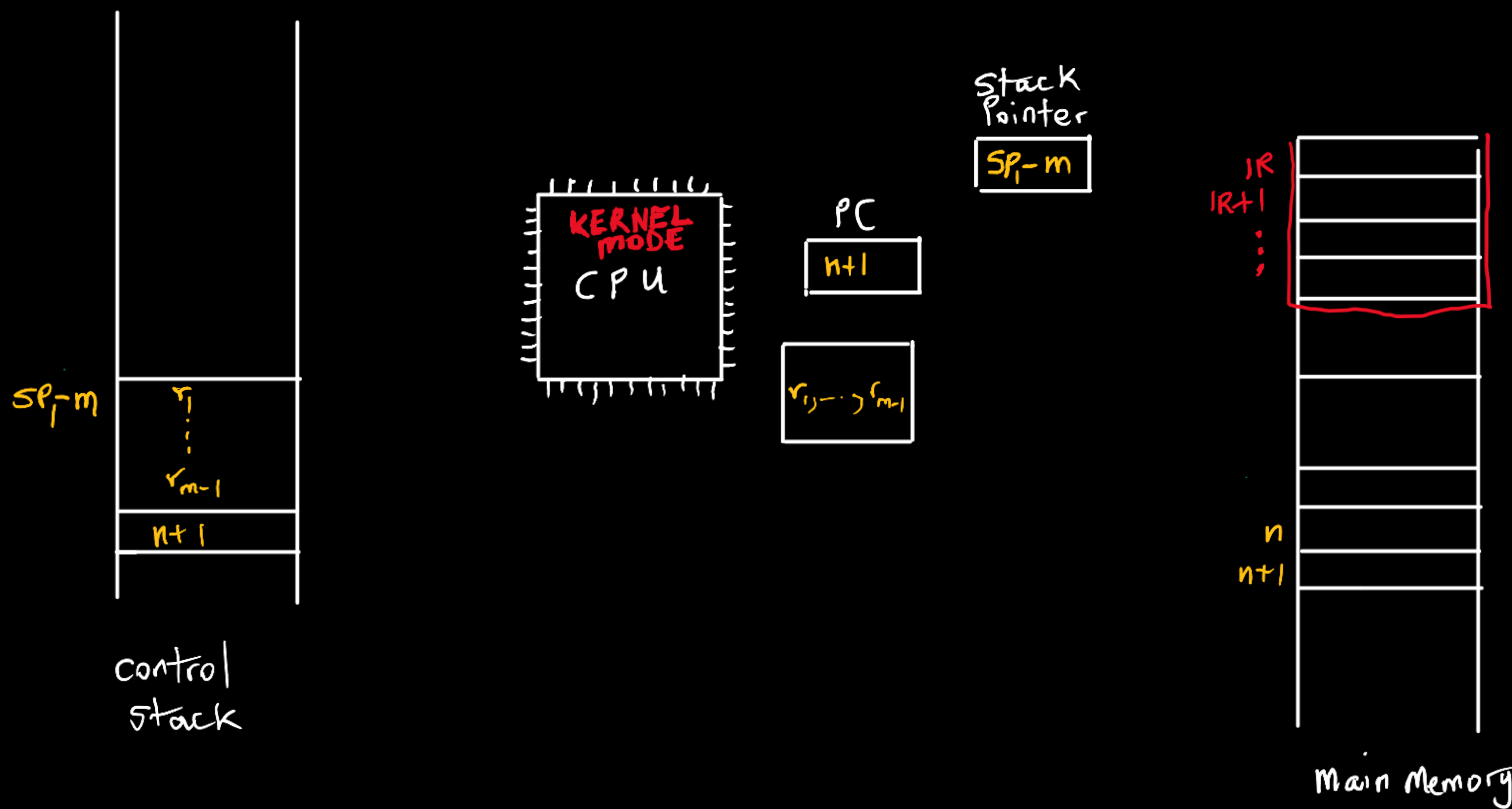


Need the OS!
KERNEL MODE
CPU
PC
n+1
Stack Pointer
$sp_1 - m$
$r_1, \dots, r_{m-1}$
$sp_1 - m$
r_1
r_{m-1}
n+1
control stack
n
n+1
Main Memory

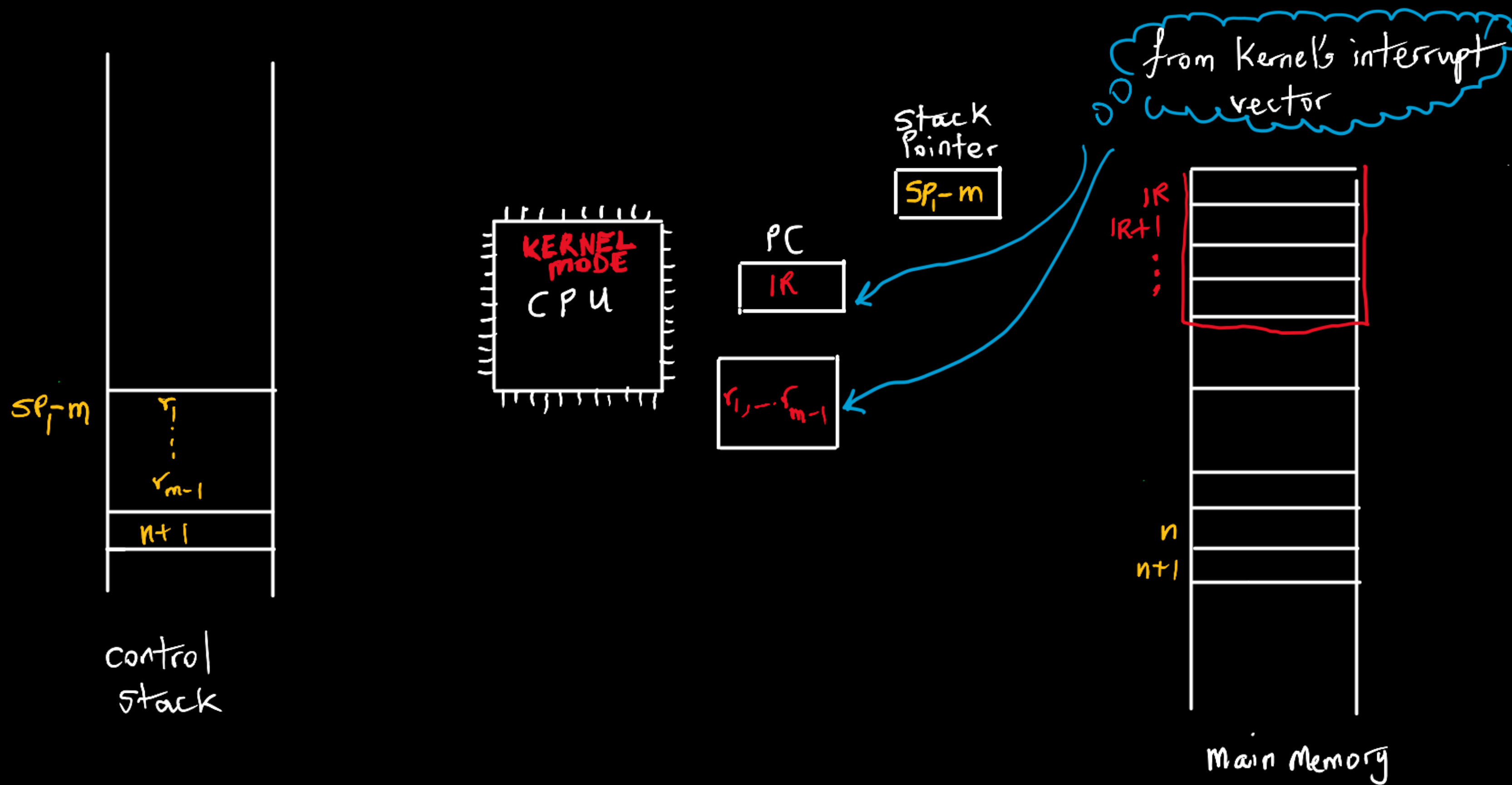

Stack Pointer
sp1 - m
KERNEL MODE
CPU
PC
n+1
r1, ..., rm-1
sp1-m
r1
rm-1
n+1
Control Stack
n
n+1
Main Memory

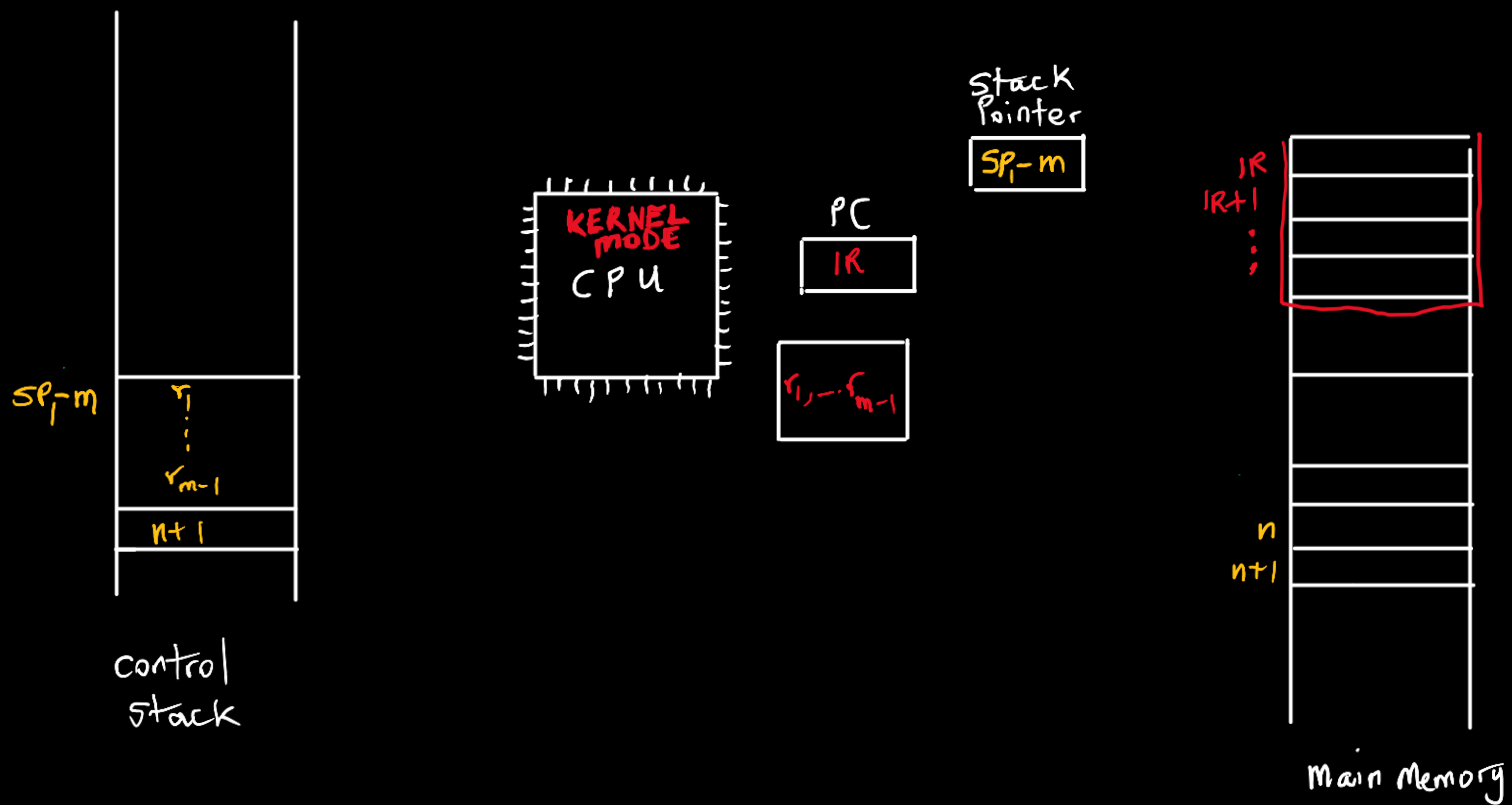


$sp_1 - m$
r_1
r_{m-1}
n+1
control stack
KERNEL MODE
CPU
PC
n+1
$r_1, \ldots, r_{m-1}$
Stack Pointer
$sp_1 - m$
IR
IR+1
n
n+1
Main Memory
restricted part of memory

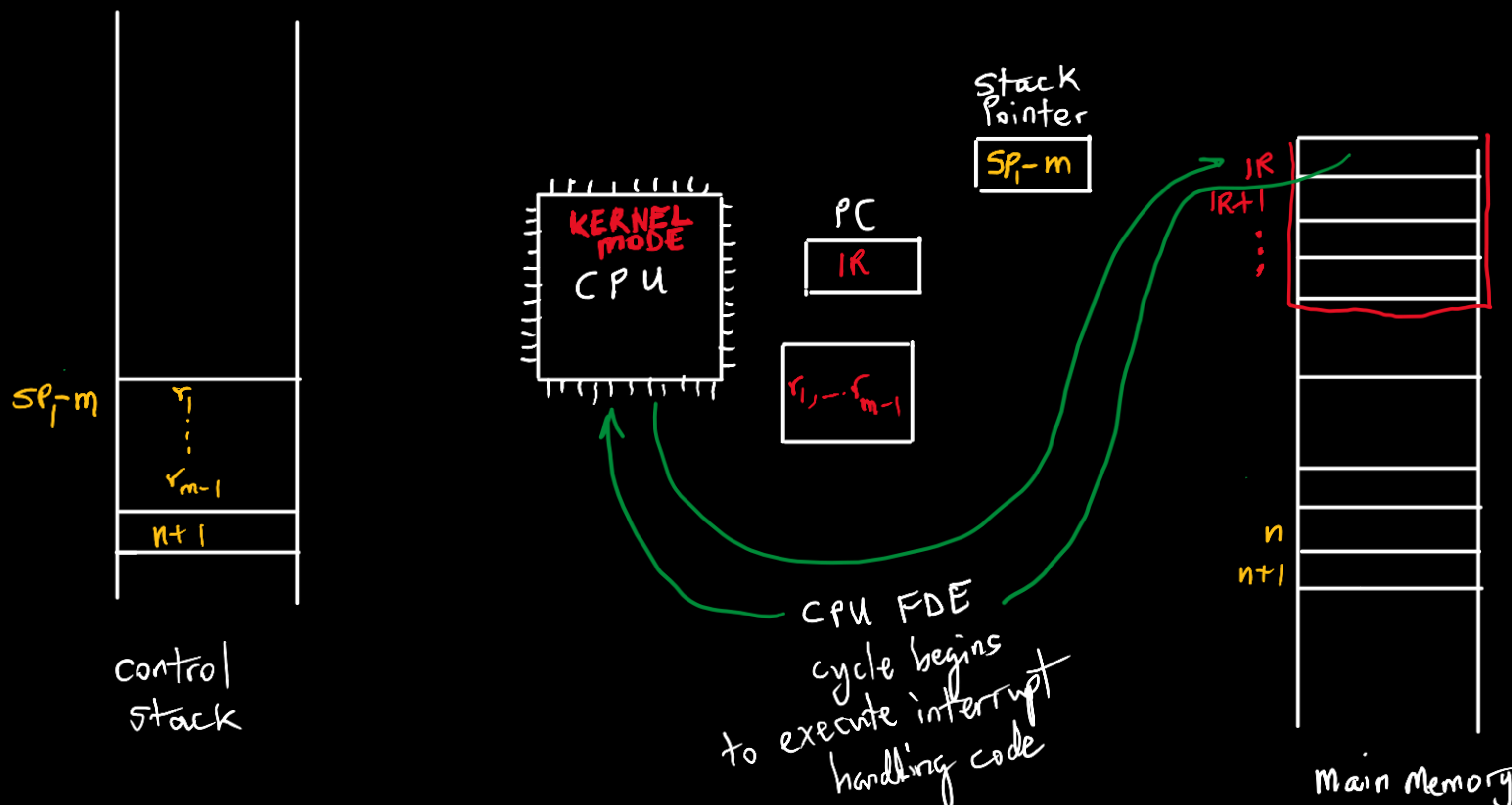


$SP_1 - m$
r_1
r_{m-1}
$n+1$
Control stack
KERNEL MODE
CPU
PC
$n+1$
$r_1, \dots, r_{m-1}$
Stack Pointer
$SP_1 - m$
IR
IR+1
n
n+1
Main Memory


from Kernel's interrupt vector
Stack Pointer
$sp_1 - m$
KERNEL MODE
CPU
PC
IR
$r_1, \ldots, r_{m-1}$
IR
IR+1
n
n+1
Main Memory
$sp_1 - m$
r_1
r_{m-1}
n+1
control stack

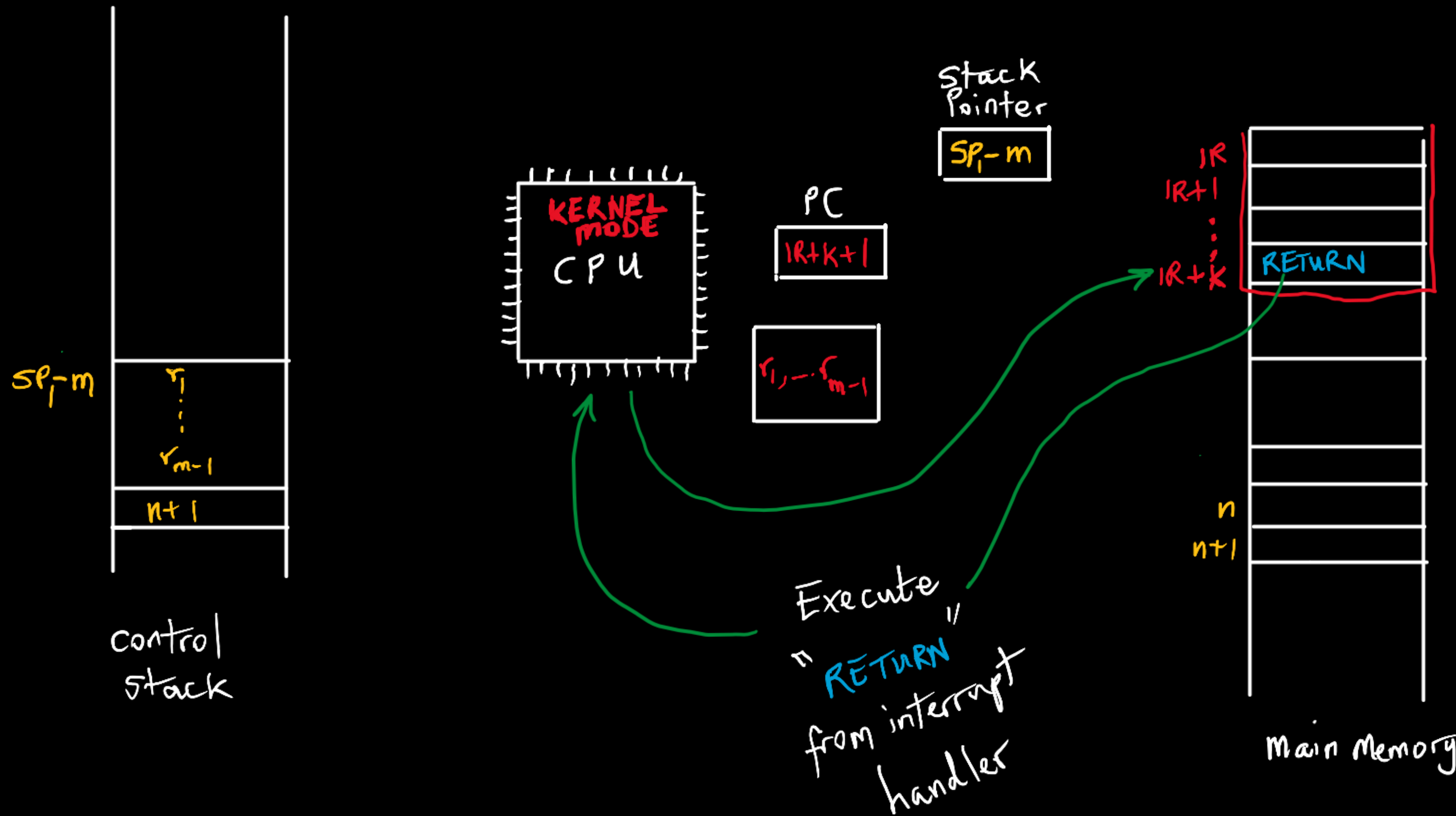


KERNEL mode
CPU
PC
IR
$r_1, \ldots, r_{m-1}$
Stack Pointer
$sp_1 - m$
$sp_1 - m$
r_1
r_{m-1}
n+1
Control stack
IR
IR+1
n
n+1
Main Memory

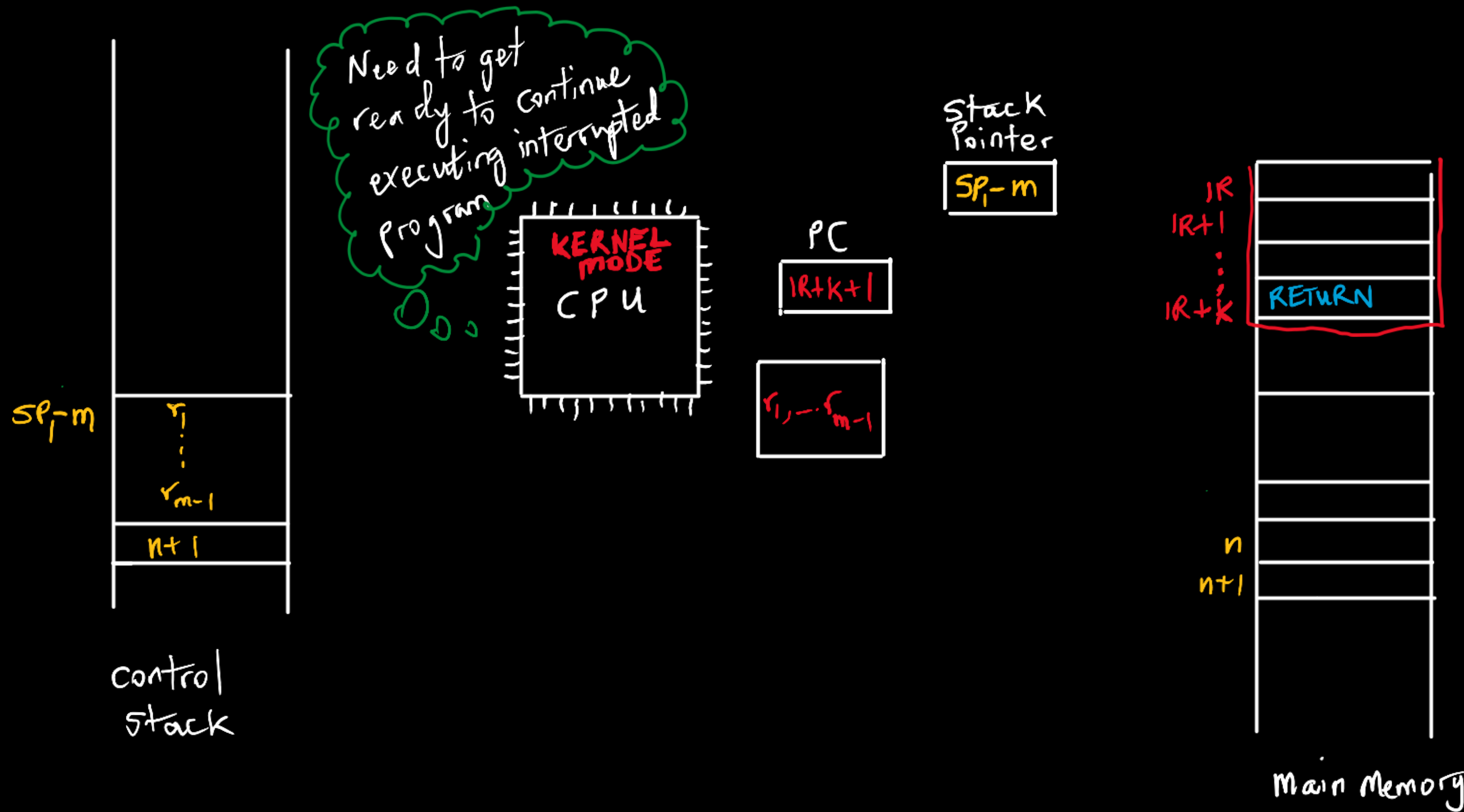


$sp_1 - m$
r_1
r_{m-1}
$n+1$
control stack
KERNEL MODE
CPU
PC
IR
$r_1, \ldots, r_{m-1}$
Stack Pointer
$sp_1 - m$
IR
IR+1
n
n+1
Main Memory
CPU FDE cycle begins to execute interrupt handling code

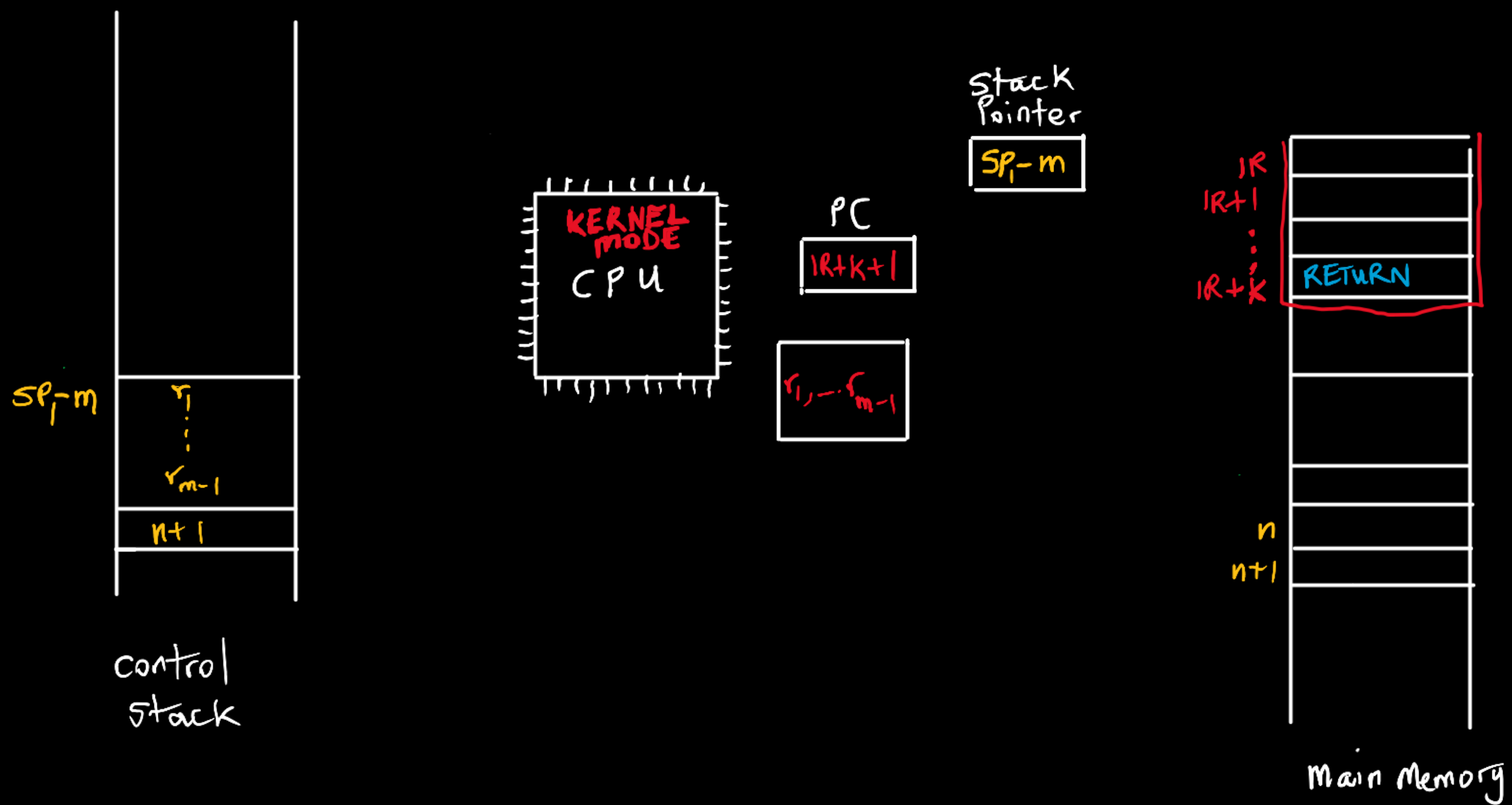
. . . Sometime Later . . .

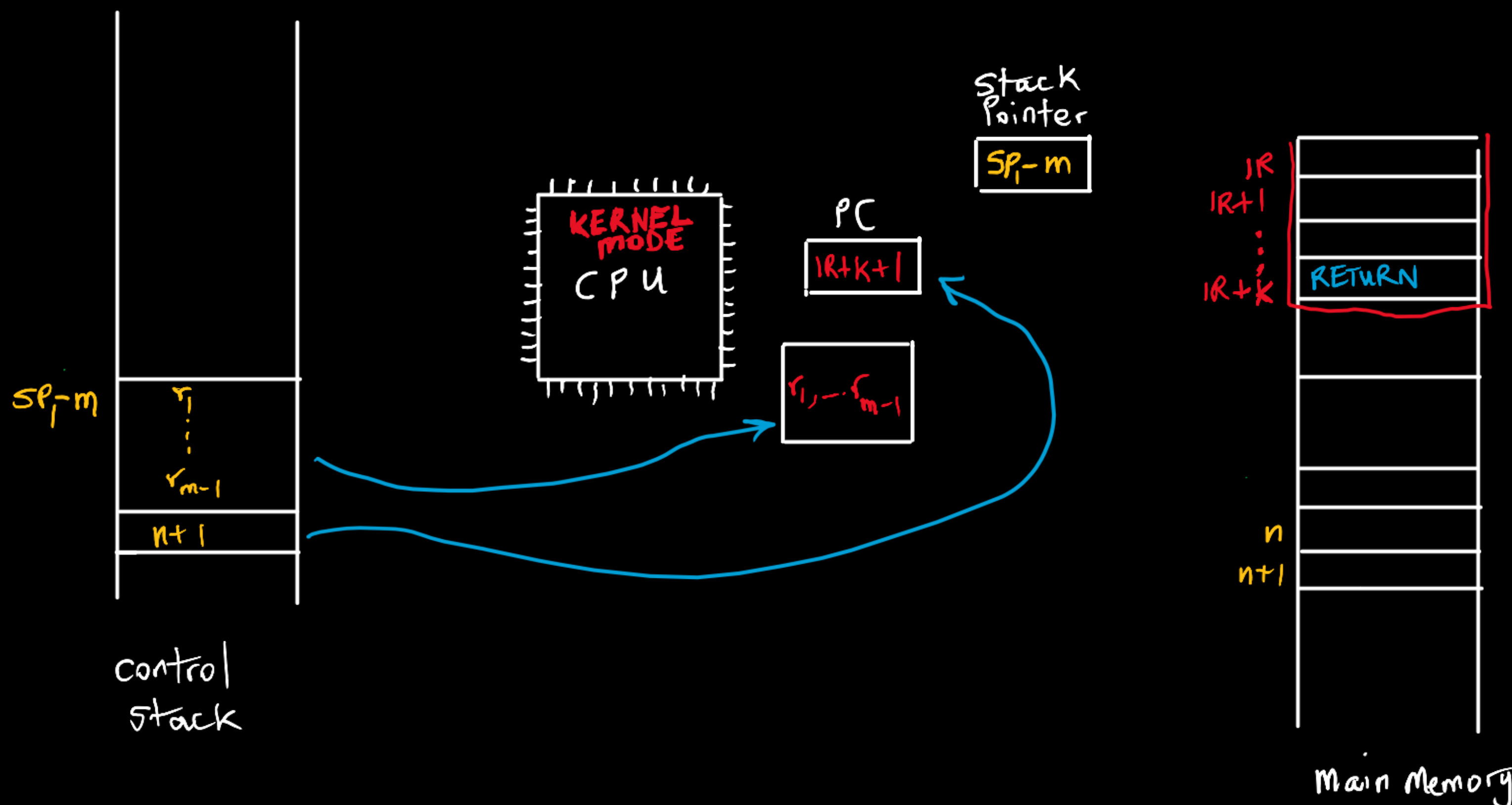


$sp_1 - m$
r_1
r_{m-1}
$n+1$
control stack
KERNEL MODE
CPU
PC
$IR+K+1$
Stack Pointer
$sp_1 - m$
$r_1, \ldots, r_{m-1}$
IR
IR+1
IR+K
RETURN
n
n+1
Main Memory
Execute "RETURN" from interrupt handler

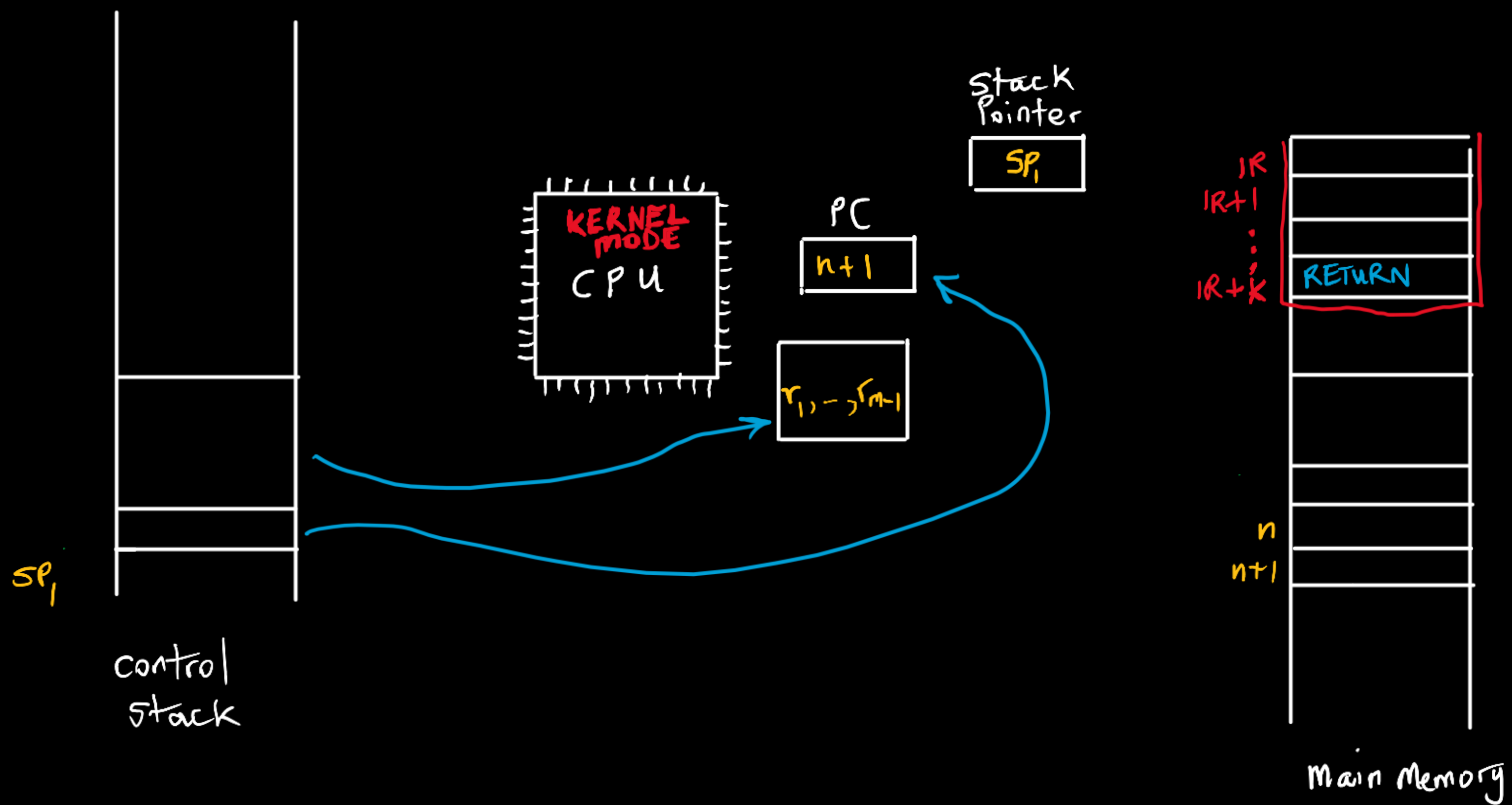


Need to get ready to continue executing interrupted program
KERNEL MODE
CPU
PC
IR+K+1
$r_1, ..., r_{m-1}$
Stack Pointer
$SP_1 - m$
$SP_1 - m$
r_1
r_{m-1}
n+1
Control Stack
IR
IR+1
IR+K
RETURN
n
n+1
Main Memory

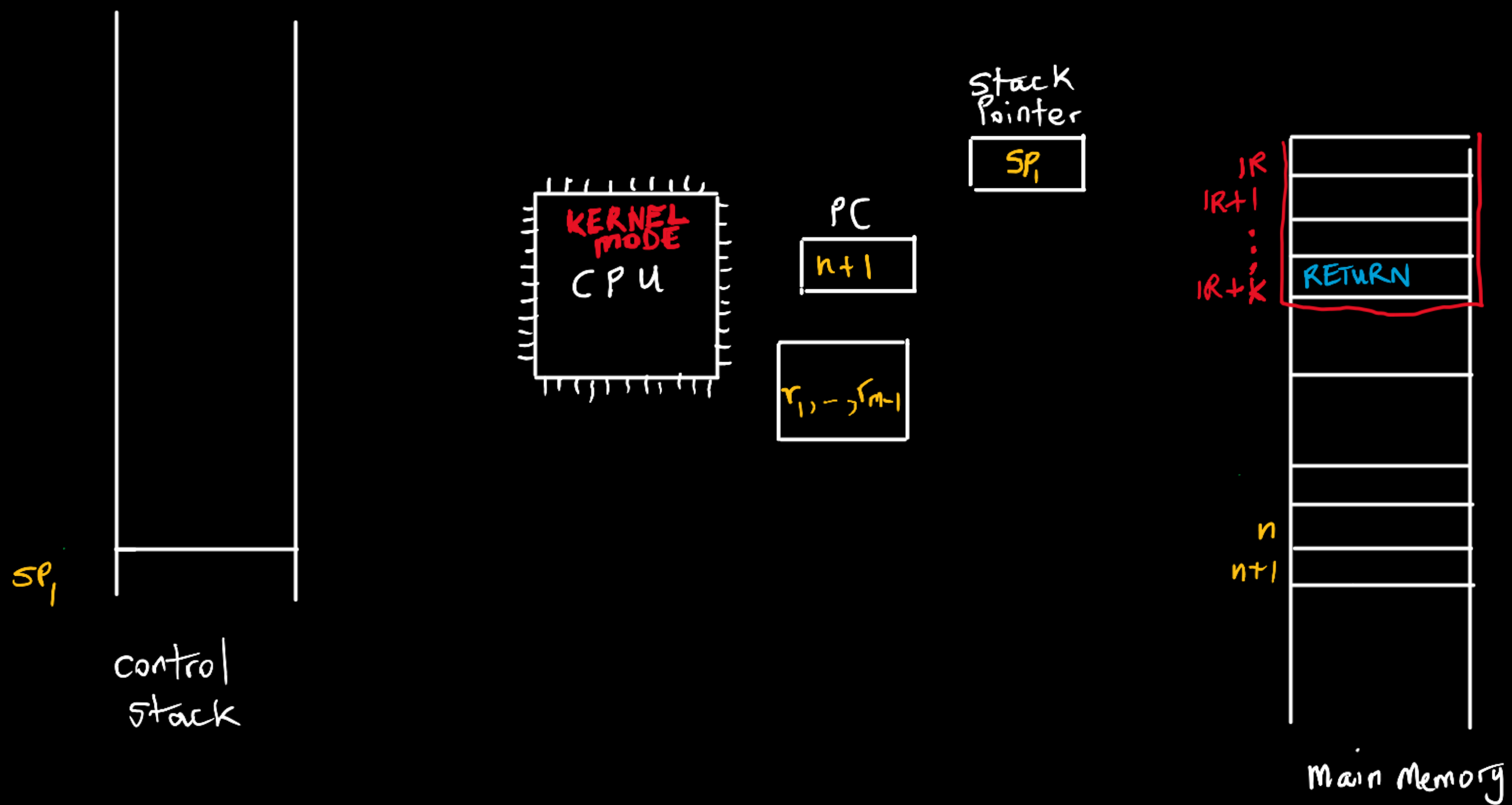

Stack Pointer
$sp_1 - m$
KERNEL MODE
CPU
PC
IR+K+1
$r_1, \ldots, r_{m-1}$
$sp_1 - m$
r_1
r_{m-1}
n+1
Control stack
IR
IR+1
IR+K
RETURN
n
n+1
Main Memory



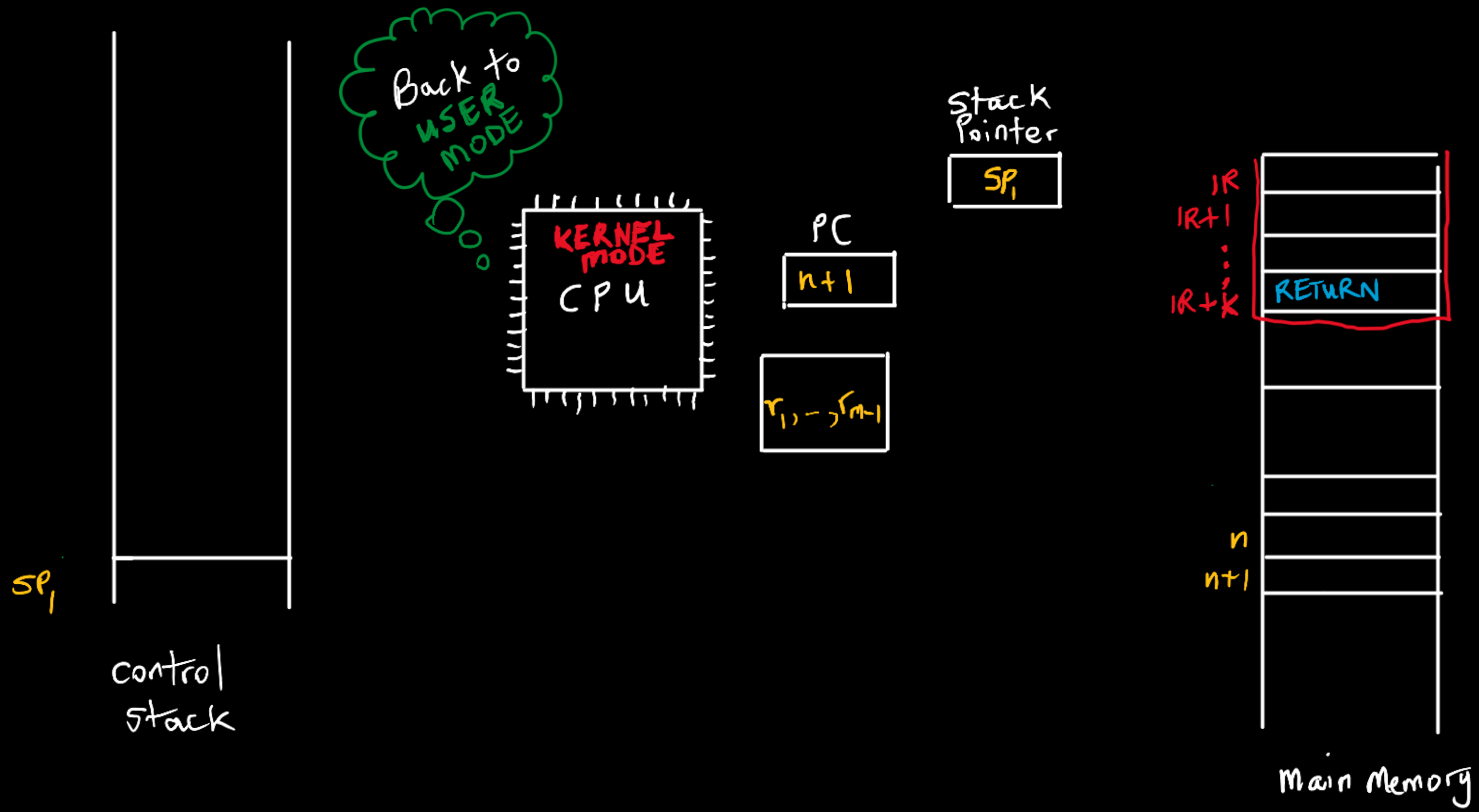
Stack Pointer
SP1 - m
KERNEL MODE
CPU
PC
IR+K+1
r1,...rm-1
IR
IR+1
IR+K
RETURN
SP1-m
r1
rm-1
n+1
n
n+1
control stack
Main Memory

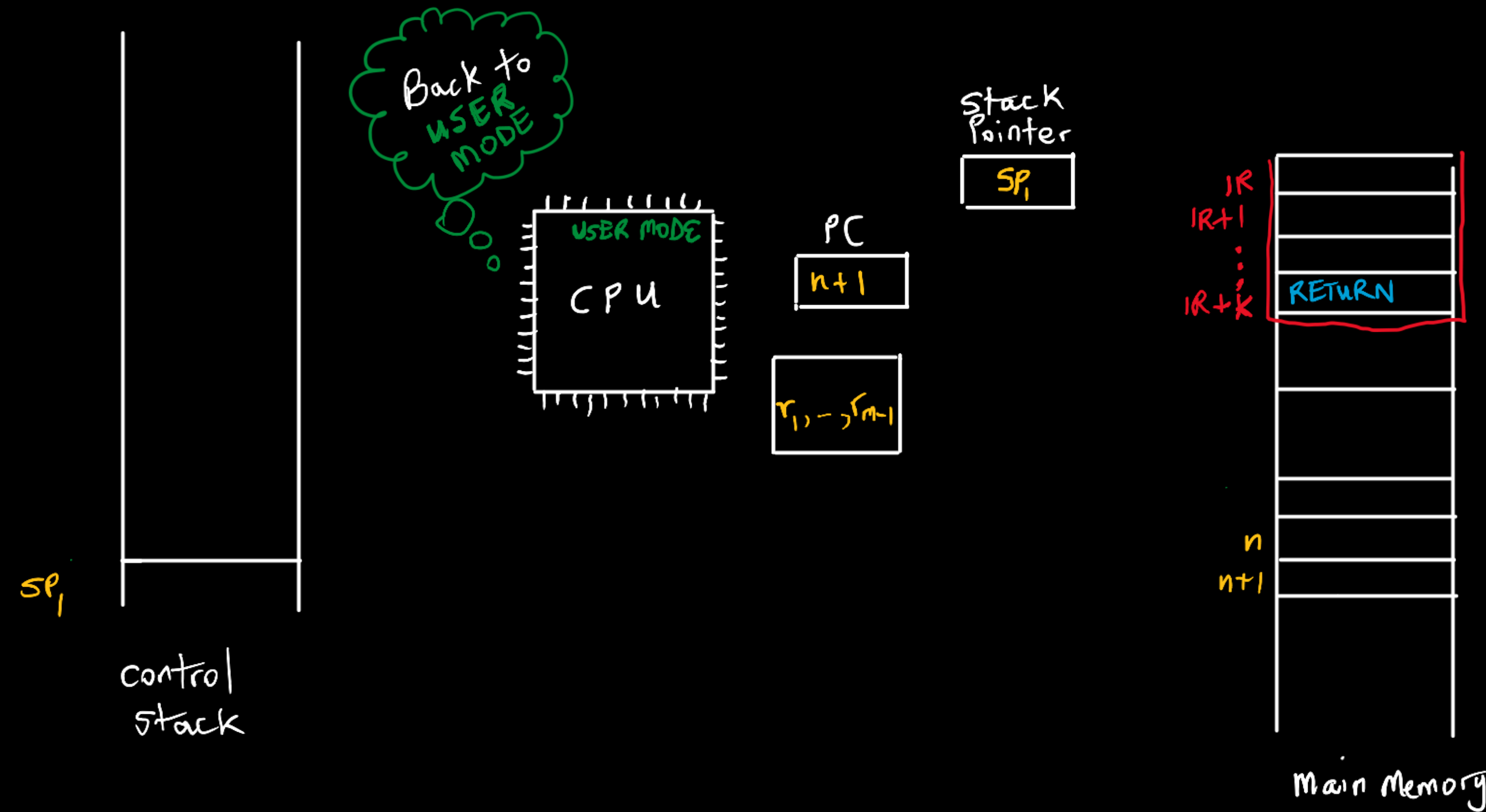


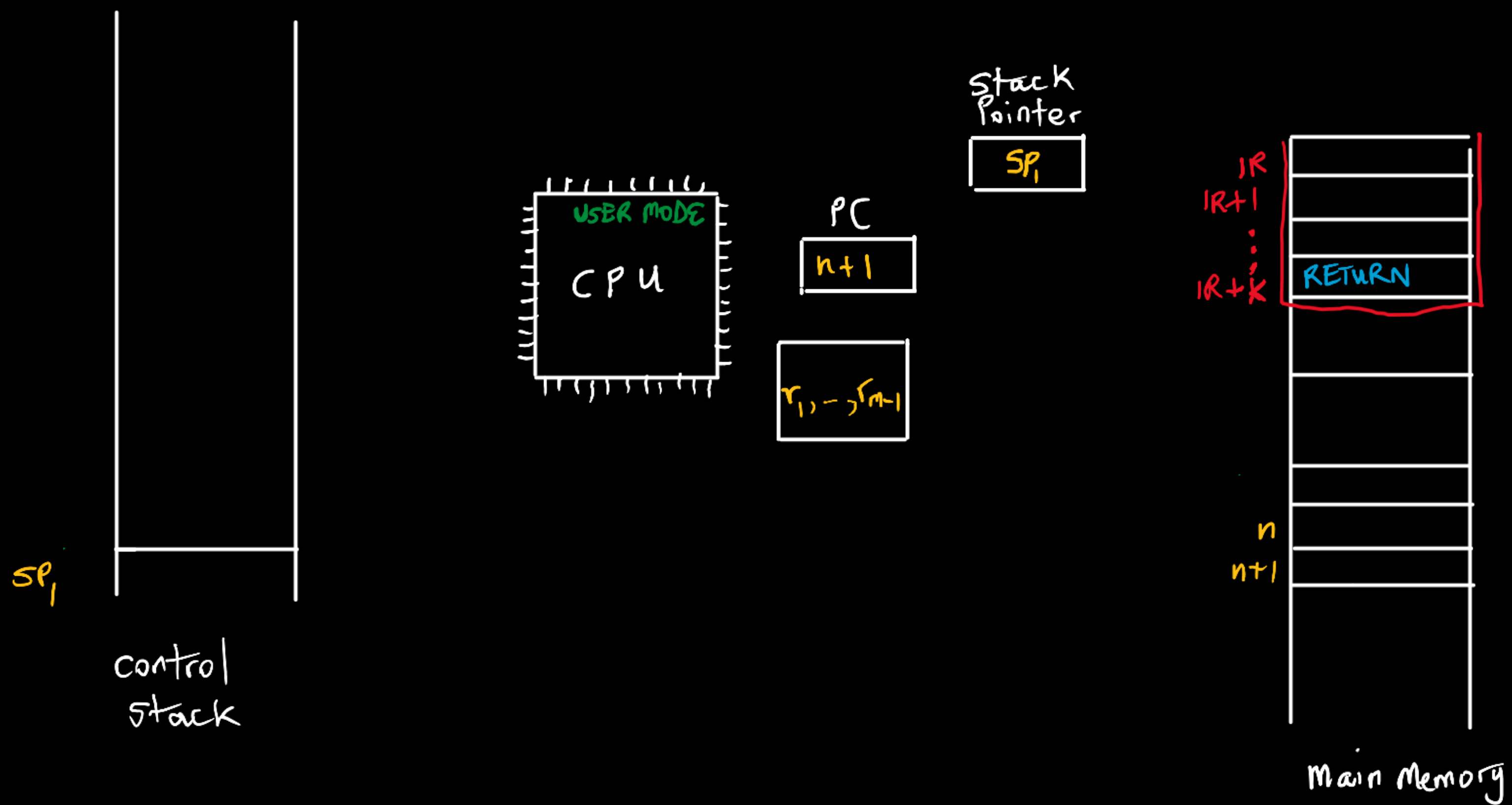
KERNEL mode
CPU
Stack Pointer
SP_1
PC
n+1
$r_1, \ldots, r_{m-1}$
IR
IR+1
IR+K
RETURN
n
n+1
SP_1
control stack
Main Memory



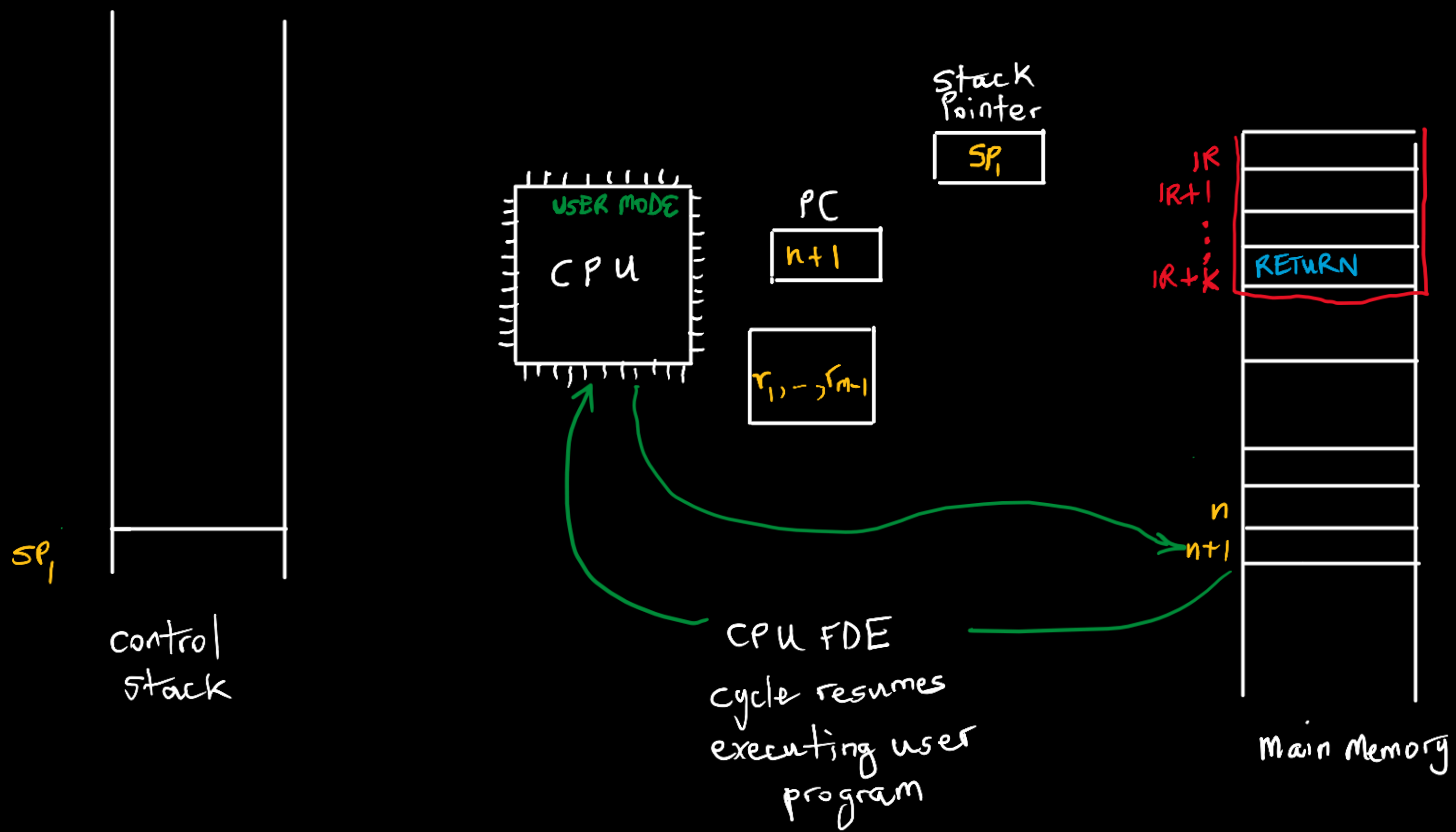
SP_1
Control stack
KERNEL MODE
CPU
PC
n+1
$r_1, \ldots, r_{m-1}$
Stack Pointer
SP_1
IR
IR+1
IR+K
RETURN
n
n+1
Main Memory


Back to
USER
MODE
KERNEL
MODE
CPU
PC
n+1
r1, ..., rm-1
Stack
Pointer
SP1
control
stack
IR
IR+1
IR+k
RETURN
n
n+1
Main Memory


Back to USER MODE
USER MODE
CPU
PC
n+1
$r_1, \ldots, r_{m-1}$
Stack Pointer
SP_1
SP_1
Control Stack
IR
IR+1
RETURN
IR+K
n
n+1
Main Memory



SP_1
control stack
USER MODE
CPU
PC
n+1
$r_1, -, r_{m-1}$
Stack Pointer
SP_1
IR
IR+1
IR+K
RETURN
n
n+1
Main Memory



Stack Pointer
SP_1
IR
IR+1
IR+K
RETURN
USER MODE
CPU
PC
n+1
$r_1, \ldots, r_{m-1}$
n
n+1
SP_1
control stack
CPU FDE cycle resumes executing user program
Main Memory

Questions

- How does the OS manage the executions of programs?
  - partial ans: OS manages processes
- But,…what is a process?

Definition of a Process

- A process is:
  - a program in execution
  - an instance of a program running on a computer
  - the entity that can be assigned to, and executed on, a processor
  - An executing set of related machine instructions
  - A unit of activity characterised by a single sequential thread of execution, a current state, and an associated set of system resources
- Can be useful to think of programs as being *passive*, while processes are *active*

“Passive” vs “Active”

- A musical score (i.e. sheet music) is “passive”, while the music produced by an orchestra playing the score is “active”
- A cooking recipe is “passive”, while cooking with the recipe is “active”
- A film script is “passive”, while performers acting based on the script is “active”
- In essence, the work done by following “passive” instructions is regarded as “active”

Management of Program Execution: Processes

- Processes were first used by designers of the Multics OS in the 1960s
- They are the <u>most important abstraction</u> provided by all modern OSes
- The OS uses processes to ensure:
  - Resources are made available to multiple applications
  - The CPU is switched among multiple applications, i.e. each application will appear to progress
  - The CPU and I/O devices can be used efficiently, securely and reliably

Process Uses Program Elements

Two essential program elements used by a process are:

Program code

- which may be shared with other processes that are executing the same program

A set of data associated with that code

Questions

- How does the OS interleave the execution of multiple programs?
- More specifically, how does the OS re-assign hardware resources among multiple programs?

Interleaving Processes

Trace

- The behavior of an individual process can be viewed as a sequence of instructions that execute for that process
- The behavior of the processor (CPU) can, itself, be viewed as interleaved traces of the various processes

Dispatcher

- A small program that switches the processor (CPU) from one process to another

Interleaving Processes

More about the **dispatcher** when we cover the topic of *scheduling*

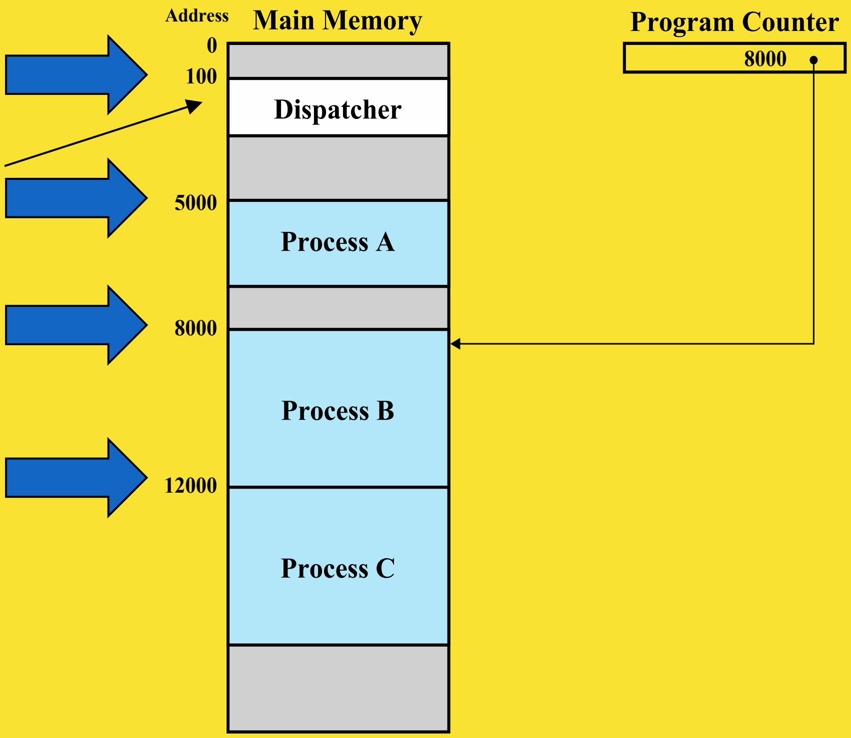


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

Interleaving Processes: Each Program's Viewpoint

(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

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

Figure 3.3 Traces of Processes of Figure 3.2

Interleaving Processes: CPU's Viewpoint

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	

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

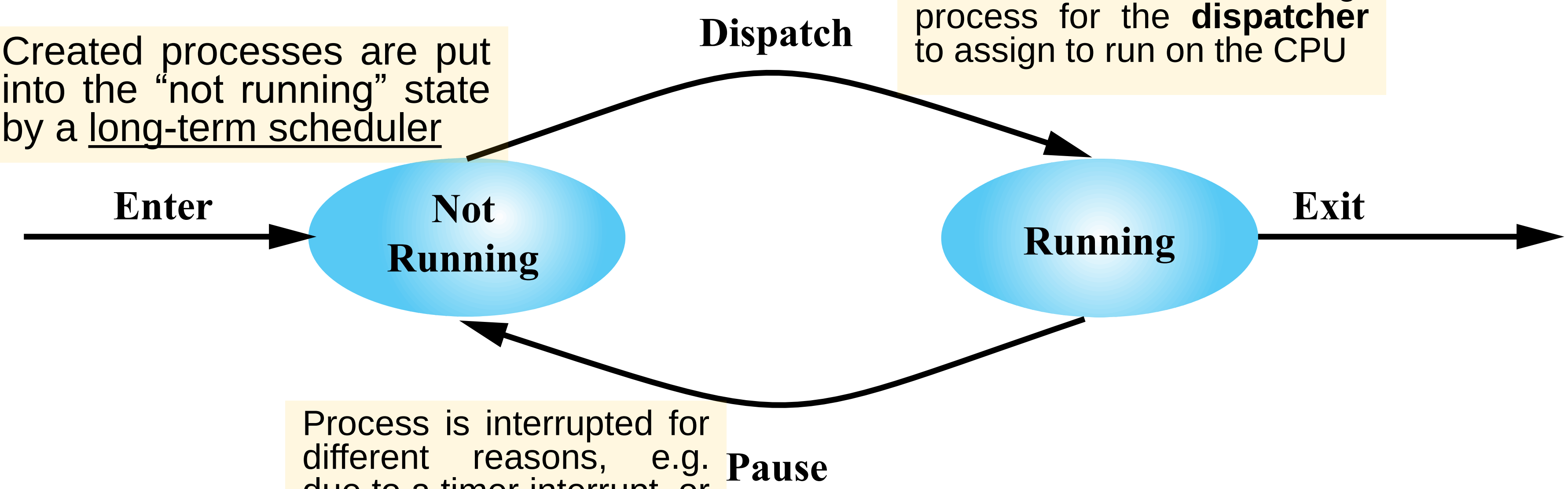
Figure 3.4 Combined Trace of Processes of Figure 3.2

Two-State Process Model

- Simplest model:
  - Process is created
  - Process is always either running or not running
  - Process terminates

A *short-term scheduler* chooses a “not running” process for the **dispatcher** to assign to run on the CPU

Created processes are put into the “not running” state by a long-term scheduler



Process is interrupted for different reasons, e.g. due to a timer interrupt, or an I/O interrupt

(a) State transition diagram

Reasons for Process Creation

- New batch job
- User logon
- OS services
- Spawned by existing process

Reasons for Process Termination

- Normal completion
- Time limit
- Insufficient computer resource available (e.g. RAM)
- Significant errors
  - I/O
  - division by zero
  - Arithmetic
- Parent process request or termination
- User termination
- Privileged instruction

Question

- How can an OS interleave processes so that new processes are not waiting while the CPU is idle?

Response Time

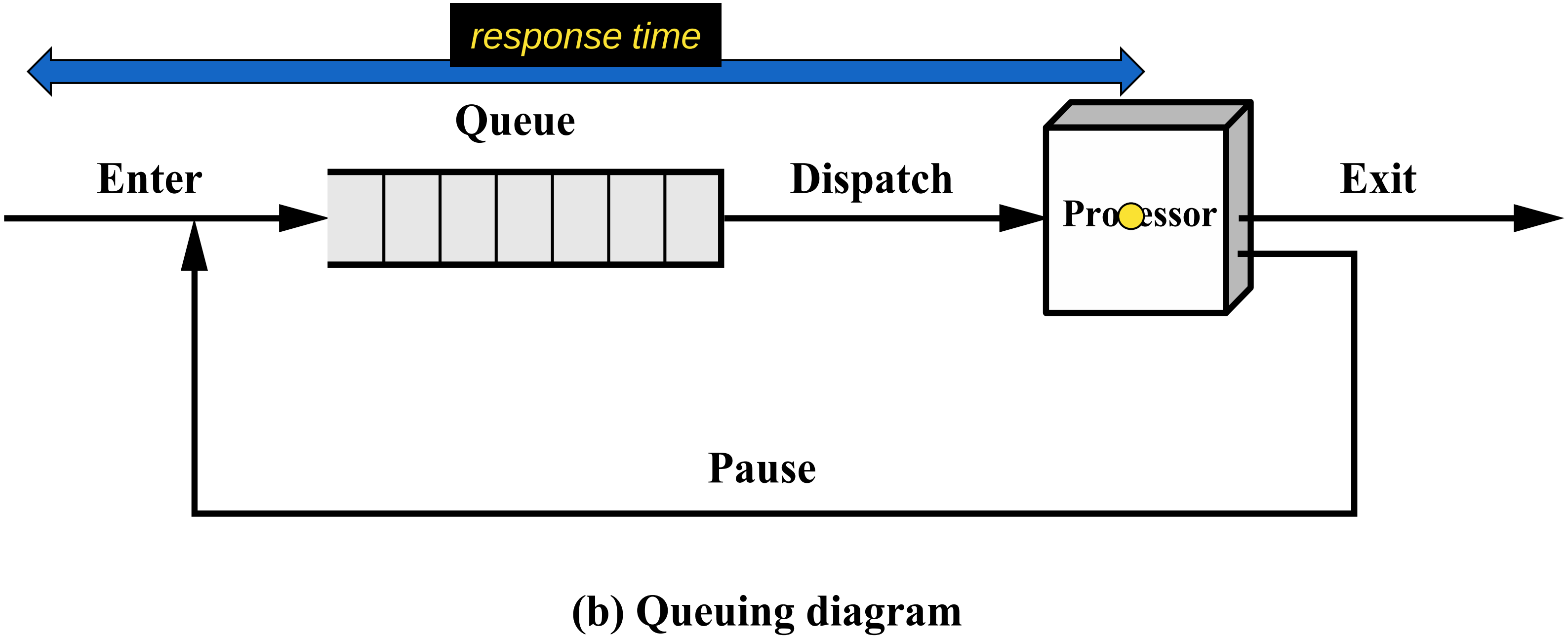
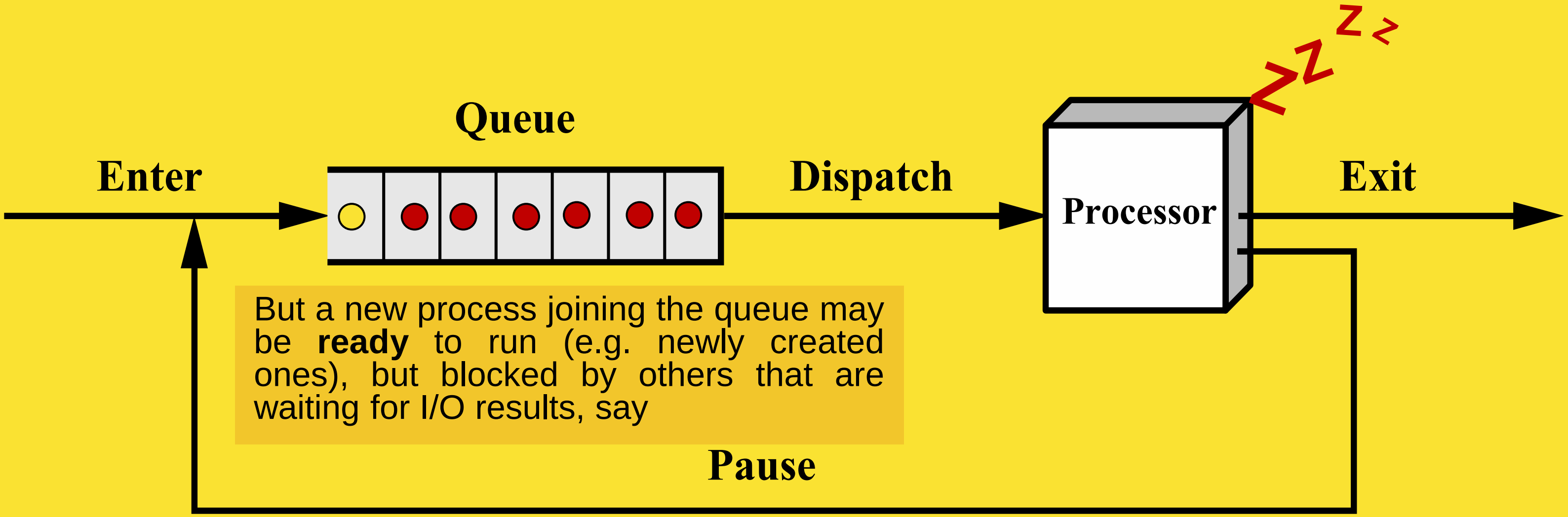


Figure 3.5 Two-State Process Model

But …



But a new process joining the queue may be **ready** to run (e.g. newly created ones), but blocked by others that are waiting for I/O results, say

(b) Queuing diagram

Figure 3.5 Two-State Process Model

Five-State Process Model

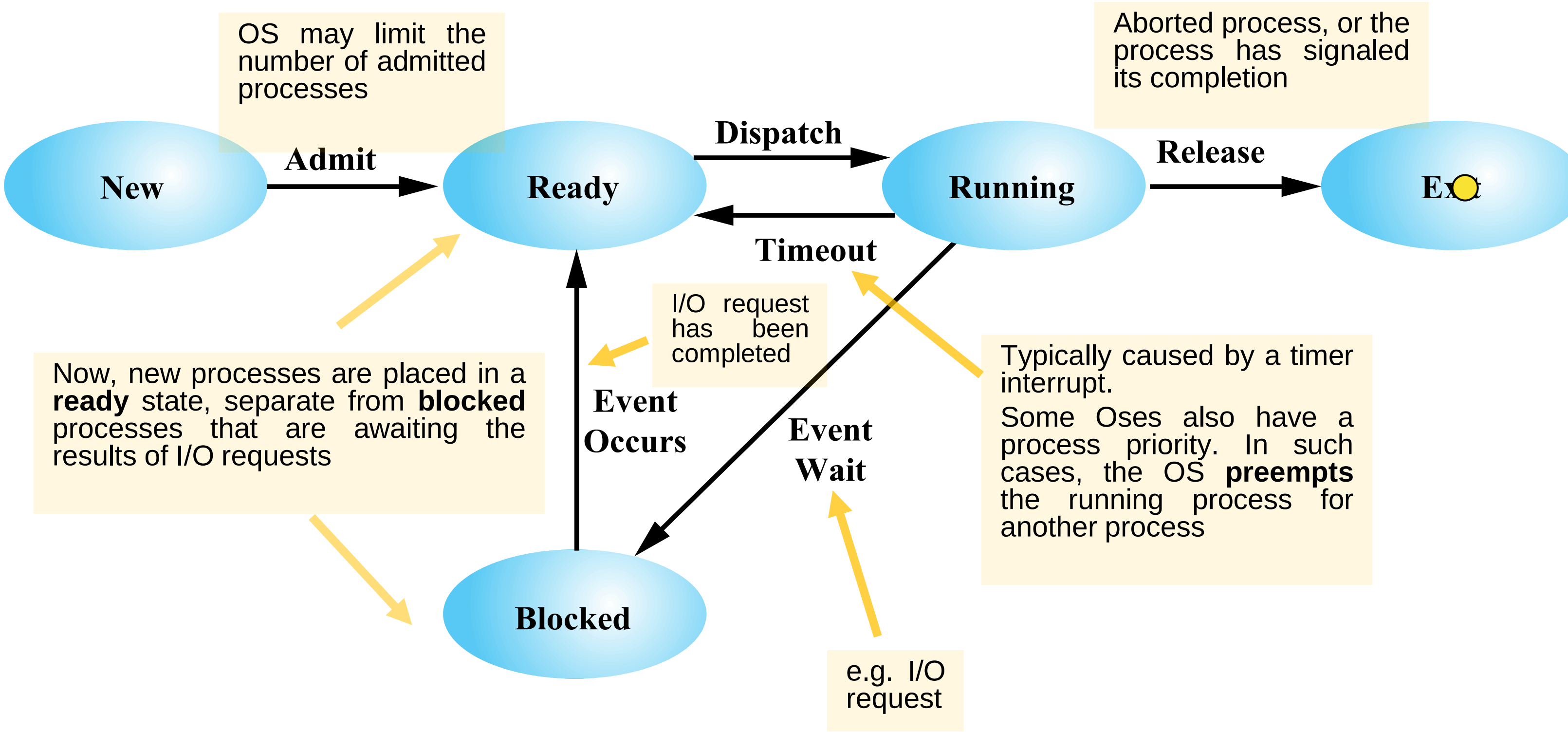
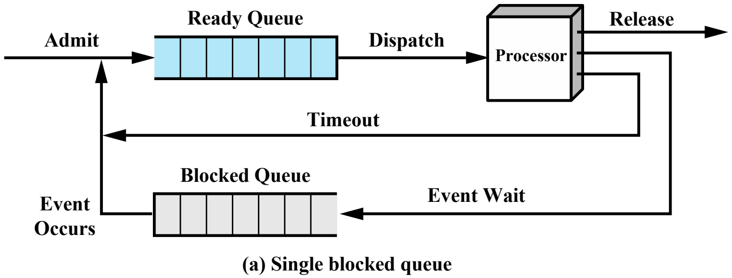
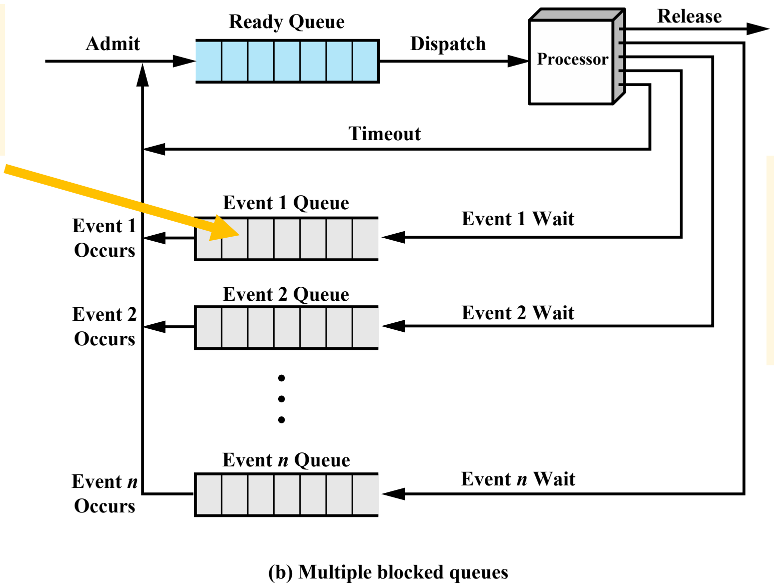


Figure 3.6 Five-State Process Model



Within each queue, different processes may be awaiting results from requests made to the same I/O device, say



Processes may have made I/O requests of different priorities (see last lecture, where interrupts can have different priorities) – each queue is for a given priority level

Figure 3.8 Queuing Model for Figure 3.6

Recall Interleaving Processes A, B and C

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	

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

Five-State Process Model: Interleaving Processes

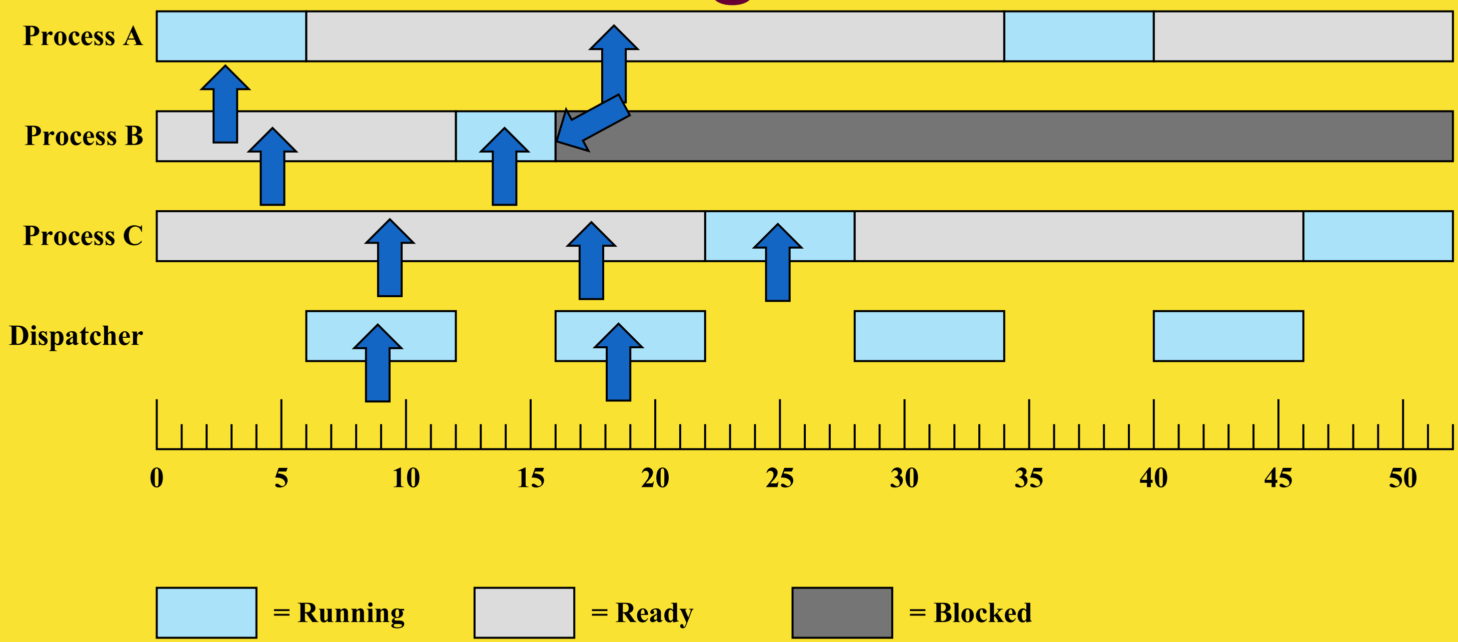


Figure 3.7 Process States for Trace of Figure 3.4

Question

- How is the limited amount of main memory shared among many interleaved processes?

But …

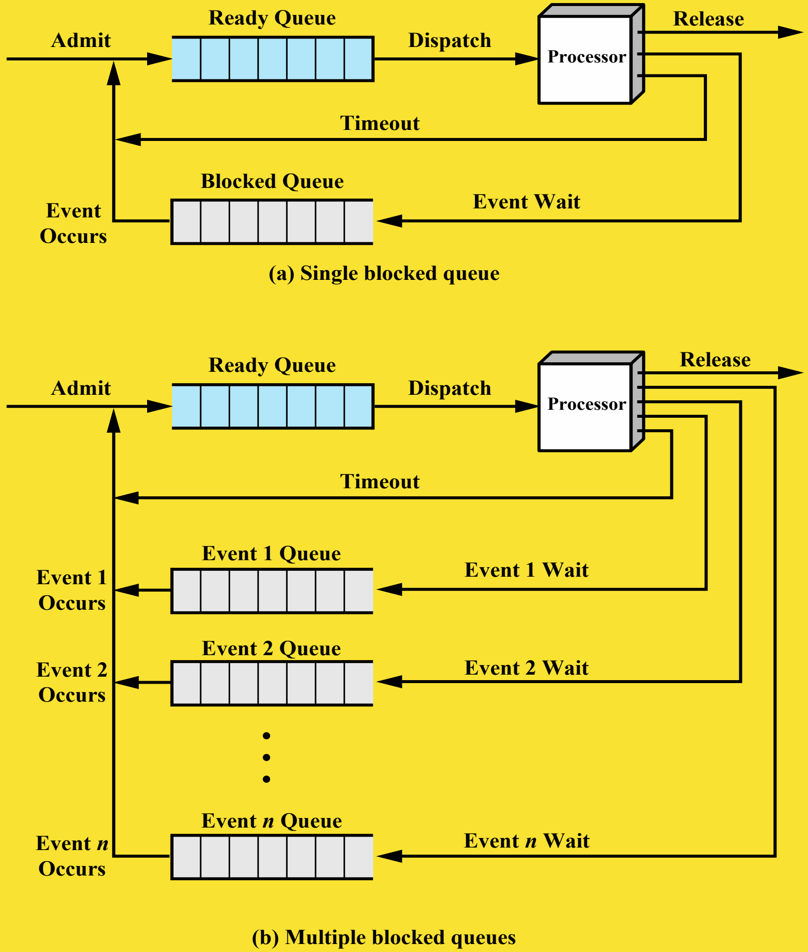


Figure 3.8 Queuing Model for Figure 3.6

CPUs are very fast, so these queues could grow very quickly! With limited *main memory*, this could be a challenge.

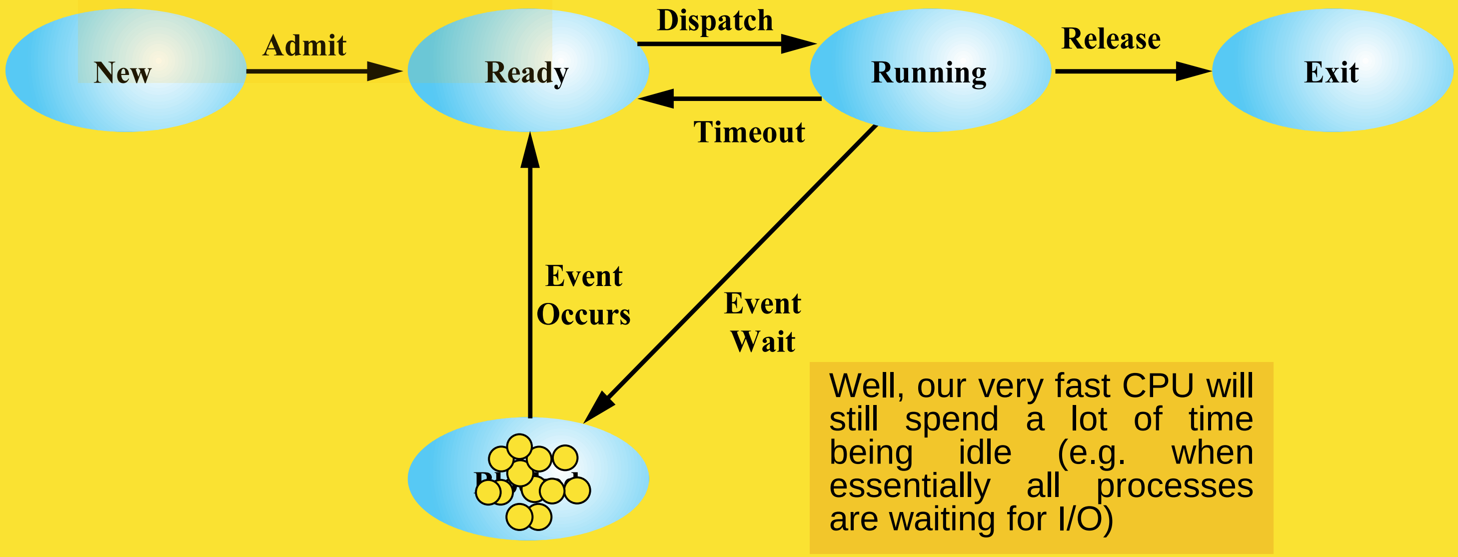
Why isn't having more main memory a good solution for this?

Ans:

- extra main memory comes with extra cost making the system less affordable
- In any case, new programs tend to require more and more main memory, so eventually the same problem of full queues will arise

And, But …

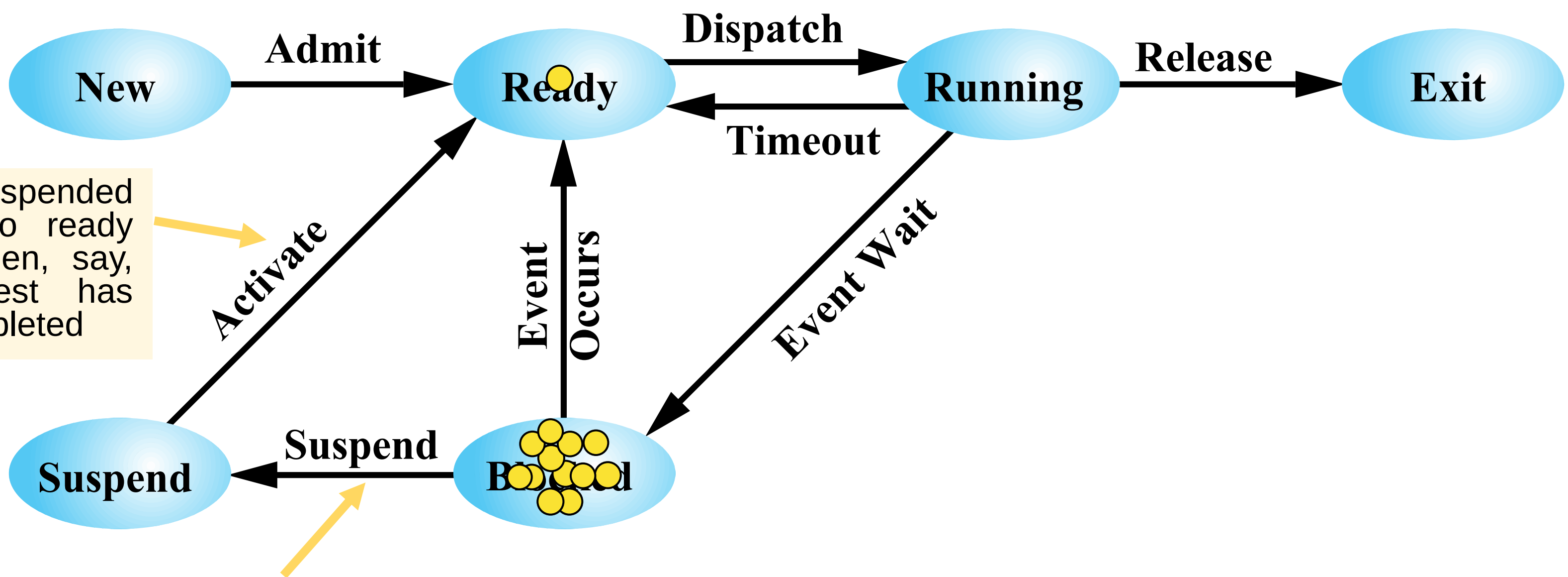
OS may limit the number of admitted processes … that means smaller queues, no?



Well, our very fast CPU will still spend a lot of time being idle (e.g. when essentially all processes are waiting for I/O)

Figure 3.6 Five-State Process Model

Six-State Process Model



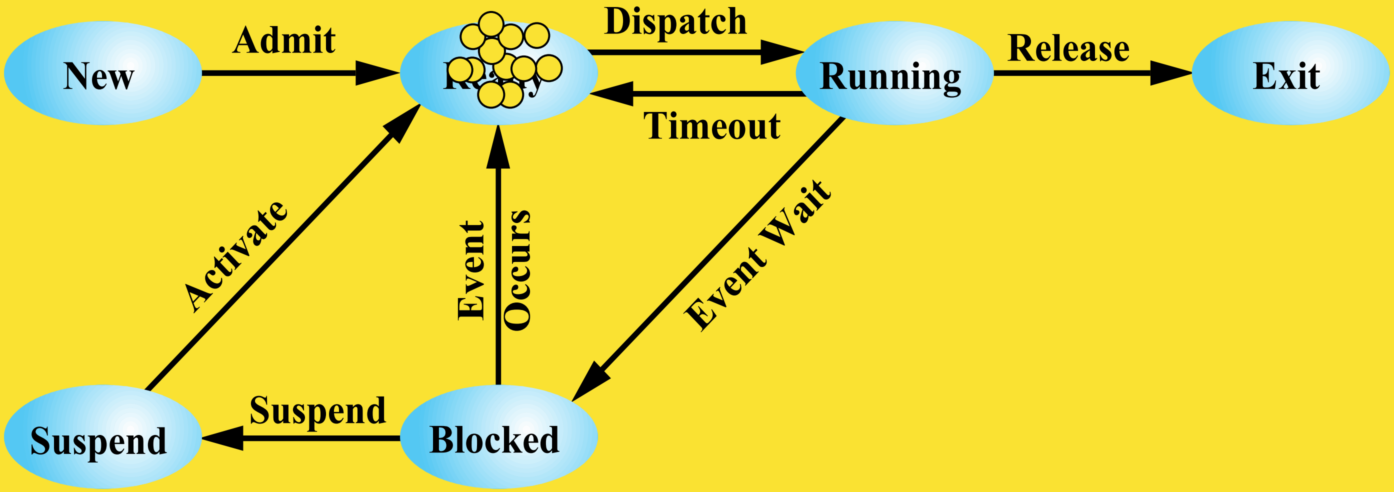
Move suspended process to ready queue when, say, I/O request has been completed

Move part or all of some blocked processes to the hard disk

(a) With One Suspend State

But …

What happens if the ready queue becomes full (e.g. if, by servicing many new high priority users all at once reaches, the OS reaches its limits on the number of ready processes)?



Could there be other reasons for a process to be taken out of main memory temporarily? E.g. directly from the running state?

(a) With One Suspend State

Figure 3.9 Process State Transition Diagram with Suspend States

Reasons for Entering Suspend State

Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Qualities of a Process in the Suspend State

- A suspended process
  - is not immediately available for execution
  - is placed in a suspended state by a suspending agent
    - Itself (e.g. debugging), a parent process (e.g. for synch), or the OS
  - may not be waiting for an event (e.g. an I/O completion)
  - May not be removed from this state except by the suspending agent

Seven-State process model

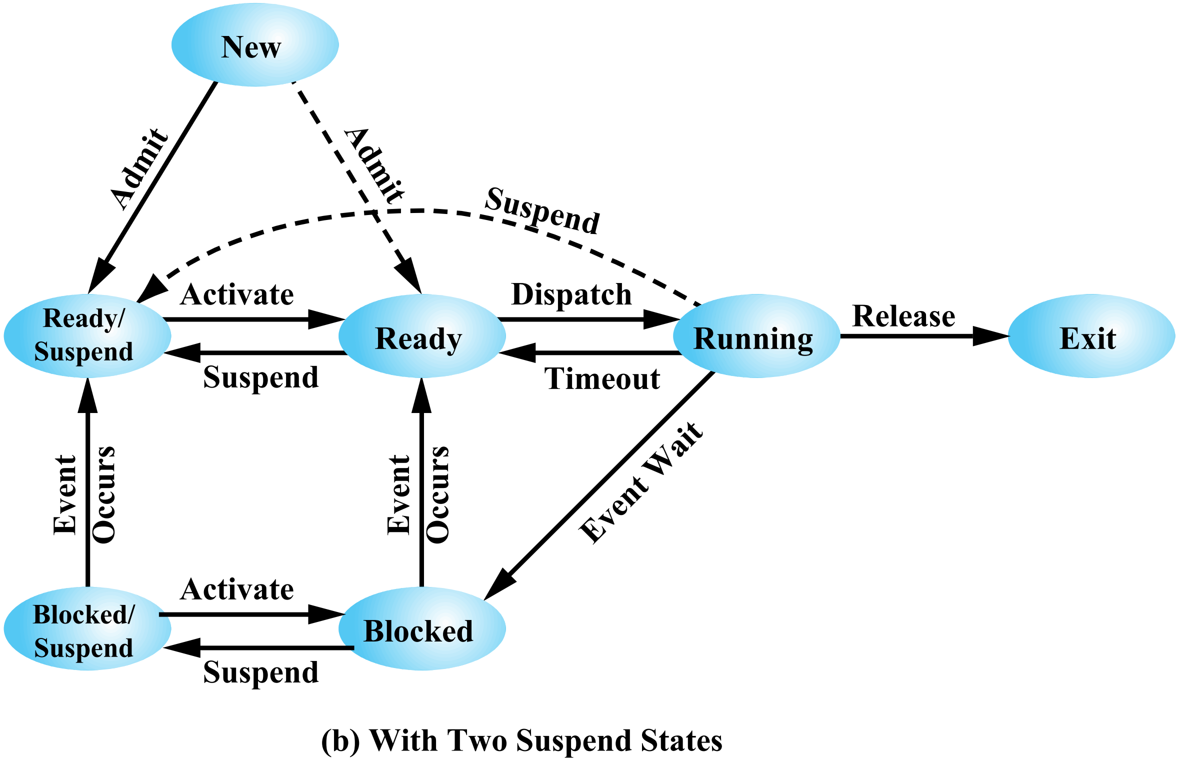


Figure 3.9 Process State Transition Diagram with Suspend States

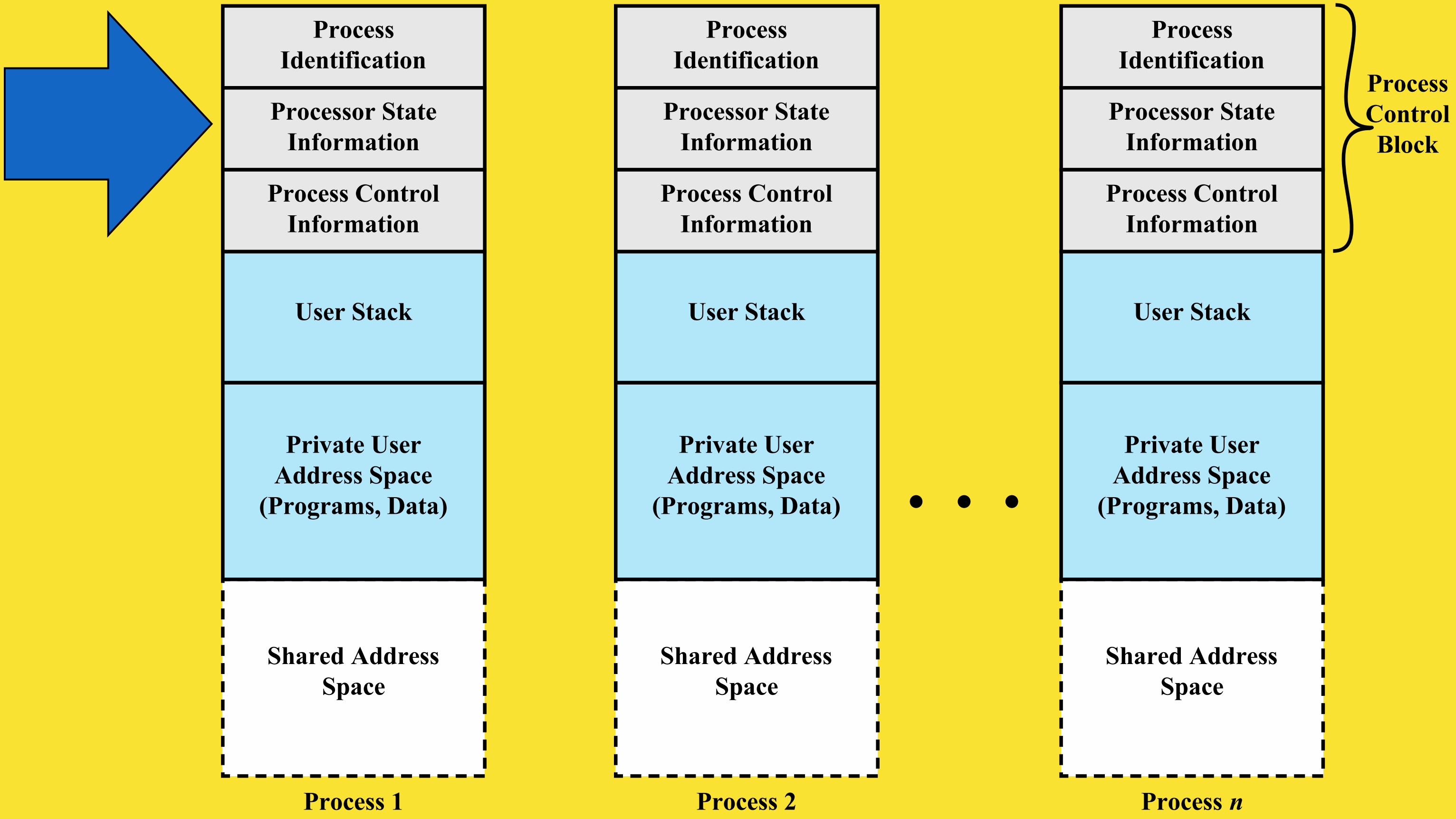
Questions

- What does a process look like to the OS?
- How does an OS keep track of a process – the process's state, attributes, and the memory locations assigned to the process?
  - ans : special data structures in RAM and on the hard disk are used by the OS to do this
- What are these data structures?

What does a process look like to the OS?

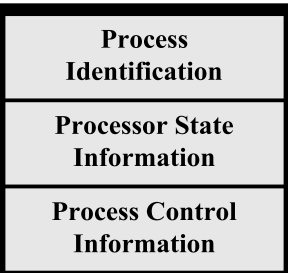
- A process is allocated a portion of main memory called the **process image**. This includes:
  - sufficient memory to hold the code and program data for the process
  - a **call stack**, used to keep track of a process's procedure calls, and parameter passing between procedures
  - memory for a special data structure called the **process control block** (PCB), that contains the process's attributes. These attributes are used by the OS for process control
- the precise location of the process image (in main memory) will depend on the memory management scheme used by the OS

Process Images in Memory



The Role of the Process Control Block (PCB)

- The <u>most important</u> data structure in an OS
  - Contains all of a process's information needed by the OS
  - PCBs are read and modified by <u>virtually every</u> module in the OS
  - The collection of all PCBs define the OS state
- Access to PCBs is easy: use process ID
- Protecting PCBs from being damaged is more challenging
  - A bug in a single OS routine could damage a PCB
  - A design change in the structure or semantics of the PCB could affect a number of modules within the OS



Process Control Block (PCB)

- The OS maintains information about a process's attributes in PCB
- It resides in protected parts of memory
- Consists of 3 sections:
  - Process identification
  - Processor state information
  - Process control information

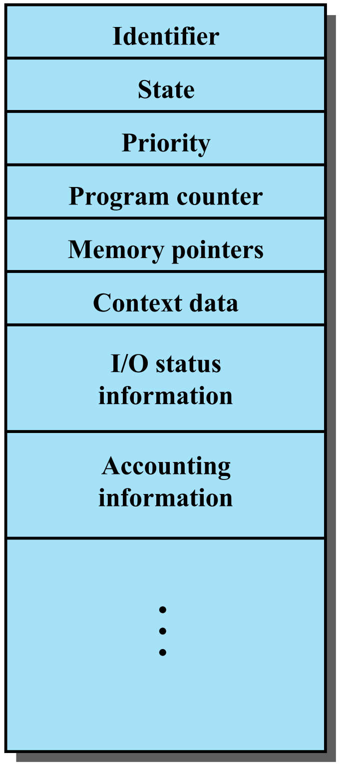
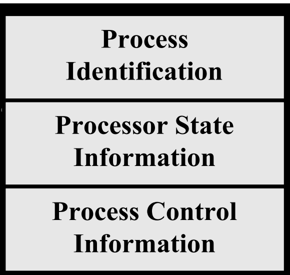
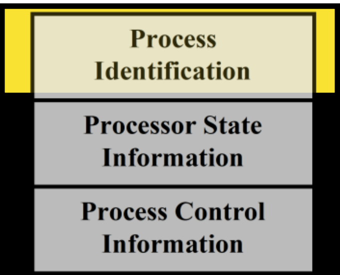


Figure 3.1 Simplified Process Control Block



PCB Section : Process Identification

- The **process ID** is a unique number identifying the process
  - an index into the **primary process table**
  - part of a mapping the OS uses to locate process information in control tables
    - **Memory tables**
    - **I/O and file tables**
- When processes communicate, the process ID informs the OS of the destination of a particular communication
- When a process creates another process, process IDs distinguish the <u>parent process</u> from its <u>child processes</u>



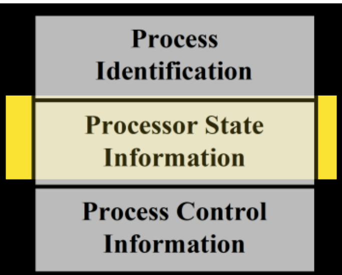
PCB Section : Processor State Information

The contents of processor registers

- User-visible registers
- Control and status registers
- Stack pointers

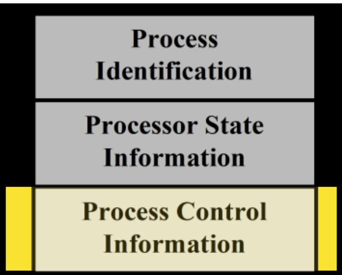
Program status word (PSW)

- Contains condition codes plus other status information
- EFLAGS register is an example of a PSW used by any OS running on an x86 processor

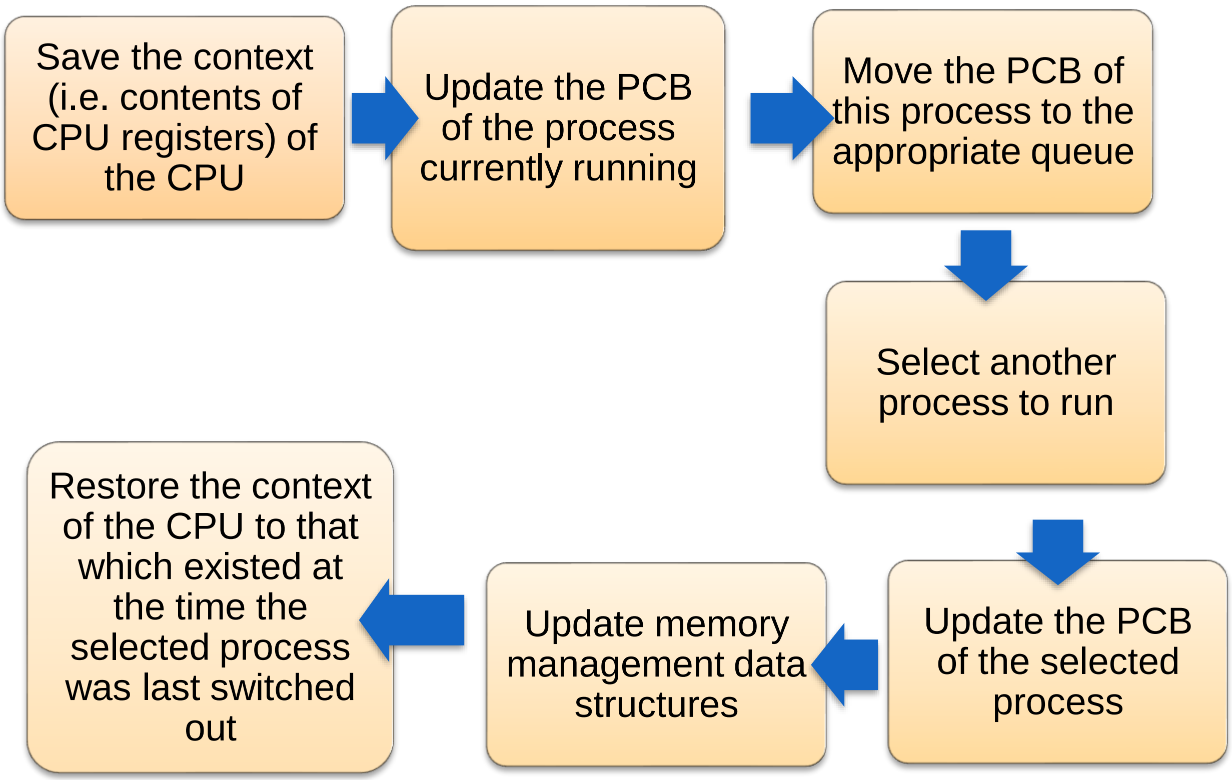


PCB Section : Process Control Information

- Used by the OS to assign the CPU to processes
  - Process state
  - Priority
  - Scheduling-related information
  - Identity of any event the process may be awaiting
- Links/pointers to other processes
- Interprocess communication
- Process privileges
- Memory management
- Resources assigned to the process



OS re-assigning the CPU : Context Switch using PCBs



What happens when an OS re-assigns the CPU to a different process?

OS Control Structures

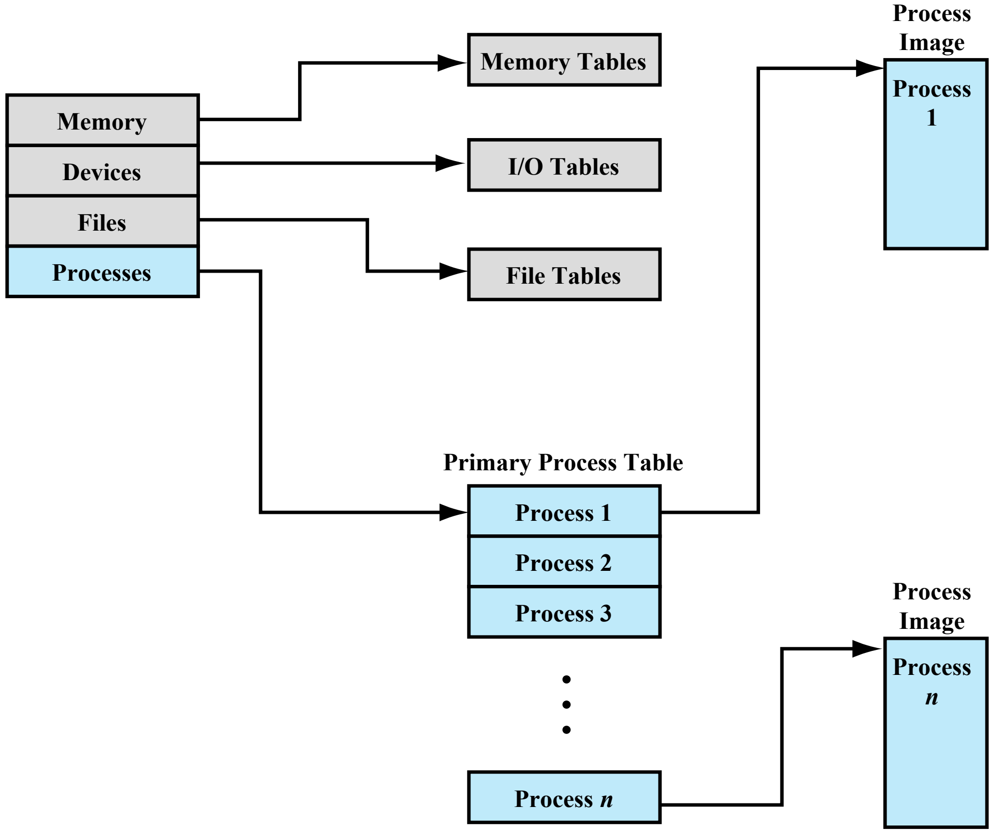


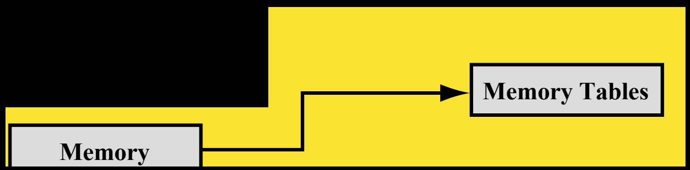
Figure 3.11 General Structure of Operating System Control Tables

Memory Tables

- Used to keep track of both main and secondary memory
- Processes are maintained on secondary memory (i.e. hard-disk) using **virtual memory** or some swapping mechanism

Must include information on:

- Allocation of main memory to processes
- Allocation of secondary memory to processes
- Protection attributes of blocks of main or virtual memory
- Information needed to manage virtual memory



I/O Tables

- Used to manage I/O devices and channels
- At any given time, an I/O device may be either available or assigned to a particular process

If an I/O operation is in progress, the OS needs to know:

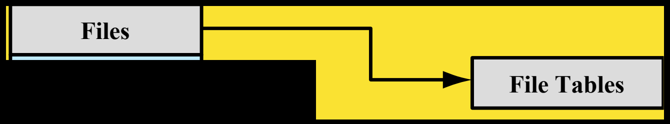
- The status of the operation
- The main memory location being used as the source or destination of the I/O transfer

Devices → I/O Tables

File Tables

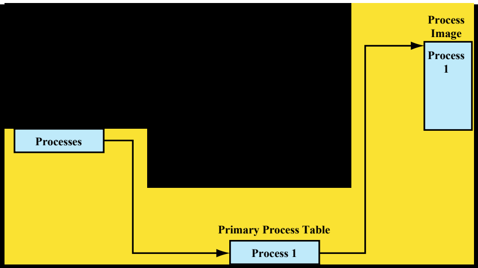
These tables provide information about:

- Existence of files
- Location on secondary memory
- Current status
- Other attributes



Process Tables

- Maintained by OS to manage processes
- Must include references to memory, I/O, and file tables, either directly or indirectly
- These tables must be accessible by the OS and, therefore, are subject to memory management



Questions

- What does an OS do when it creates a process?
- Is the OS a process?

Creating Processes

- Assigns a unique process identifier to the new process
- Allocates memory for the process image
- Initializes the PCB
- Sets the appropriate linkages
- Creates or expands other data structures

What does an OS do when it creates a process?

Is the OS a Process?

No, it is completely **separate** from all processes

No, OS routines run **within** the address space of each running process

No, it is **several** processes, running concurrently and performing different tasks

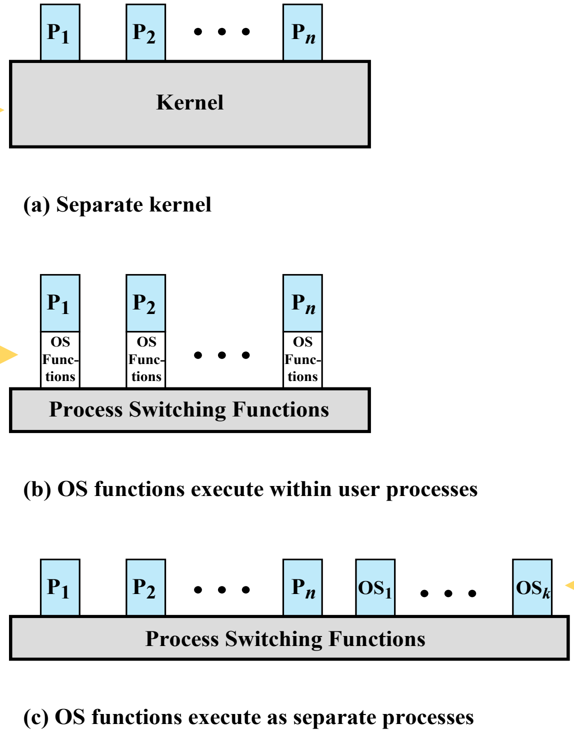


Figure 3.15 Relationship Between Operating System and User Processes