Operating Systems

Lecture 2a

Dr Ronald Grau

School of Engineering and Informatics

Spring term 2018

Previous Lecture

Operating system architectures

- Abstractions
  - Processes
  - Virtual memory
  - Files
  - System call interface
- User mode vs. Kernel mode

- Basic design principles
  - Separation of interface and implementation
  - Separation of policy and mechanism
  - Portability
  - Coherence
- Basic architectures
  - Layers & modules
  - Monolithic kernels
  - Microkernels

Today

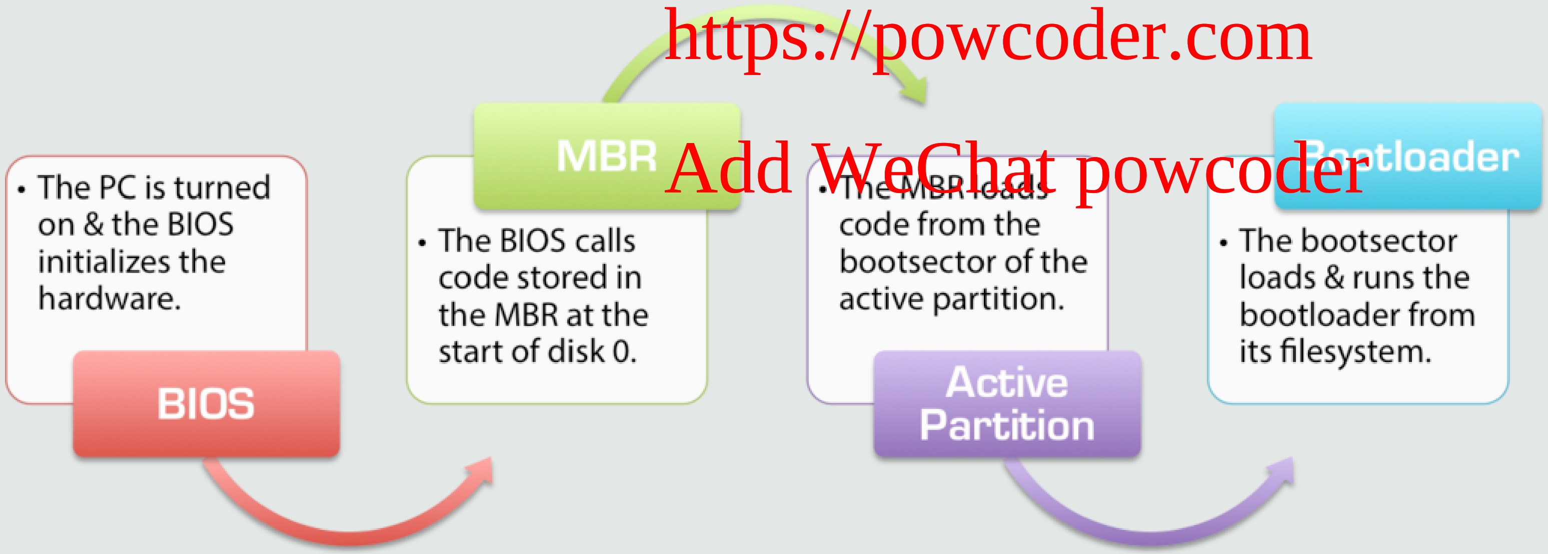
Processes

- Bootstrapping
- Processes
  - Creation
  - Management
  - Execution
  - Termination

Bootstrapping

How to start up an operating system?

- There is usually a program (called bootstrap) in Read-Only Memory (ROM)
- It loads the operating system files from persistent storage into main memory



- The PC is turned on & the BIOS initializes the hardware.
- The BIOS calls code stored in the MBR at the start of disk 0.
- The MBR loads code from the bootsector of the active partition.
- The bootsector loads & runs the bootloader from its filesystem.

Example: A local operating system on a PC

BIOS:
Basic Input/Output System

MBR:
Master Boot Record

Bootsector

Bootloader

BIOS now largely replaced by UEFI (Unified Extensible Firmware Interface)

Recap: The process abstraction

A process is a program in execution

- Operating system duties:
  - Process creation:
    Load the executable file from secondary storage into primary memory (RAM)
  - Handle concurrency
  - Scheduling
  - Coordination of resource requests
  - Synchronisation
  - Signalling of events (e.g. I/O)
  - Inter-process communication (IPC)

Recap: The process abstraction

Why do we need the process abstraction? Can’t we just run programs?

- On modern operating systems, we often want programs to run at the same time.
- We need mechanisms to coordinate their concurrent execution. This means these programs will have to share available resources during execution.
- We also want to provide the „illusion“ that each program can have exclusive access to the CPU, memory, and other resources – without burdening the developers of these programs with the finer details about how all that will work during runtime.
- The process abstraction provides the operating system with the means to effectively manage different aspects of program execution.

Processes

A few fun facts about processes

- Processes can be created and killed.
- Processes can have children.
- A process is more than just a program.

Process creation

When is a process created?

- Whenever some kind of program is started, e.g.:
  - Operating system startup
  - OS starting a service (daemon)
  - User login
  - User opening an application
  - Process creating another process (child)

Process creation

About parents and children

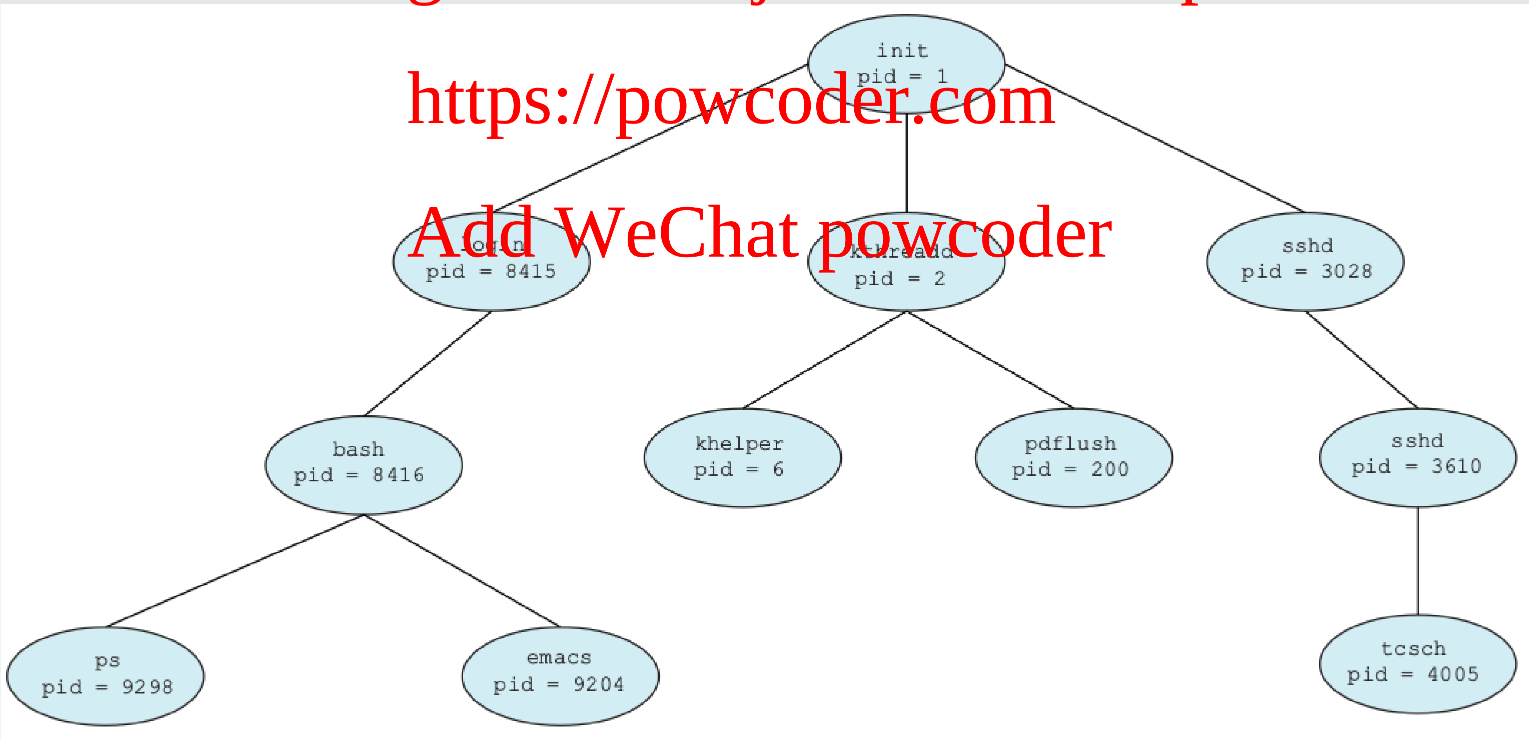
- A process becomes a parent when it creates other processes (its children).
- Child processes „receive“ resources from their parent or independently from the OS.
- Flexible execution: Parent and child can either run concurrently, or the former waits for the latter to finish.
- Flexible use of address space: The child can be directly derived from the parent (a copy of the same program & data) or run an entirely different program.

- Related system call (UNIX): **fork**

Process creation

Process hierarchy

- Operating systems like LINUX have a root process (init) that spawns all others



Example of a process hierarchy (LINUX)

Process management

Some book-keeping the operating system does

- Many things to watch: processes, memory, files, devices, etc.
- The operating system maintains a process table to keep track of active processes.
- For each process, a process image is stored in virtual memory, which contains all relevant information about its state.

Process management

Process image contents:

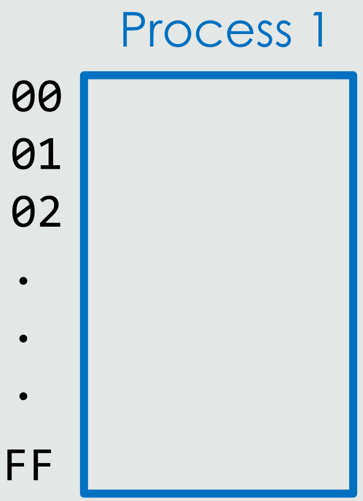
- Process Control Block (PCB)
  - Process ID, parent process ID, user ID
  - Process state (e.g., current activity, register contents, program counter)
  - CPU scheduling information (e.g., priority)
  - Memory management information, process privileges
  - Accounting information (e.g., resource use statistics, time limits)
  - I/O status information (e.g., devices & files allocated to the process)
- User program
- User data
- Stack
- Heap

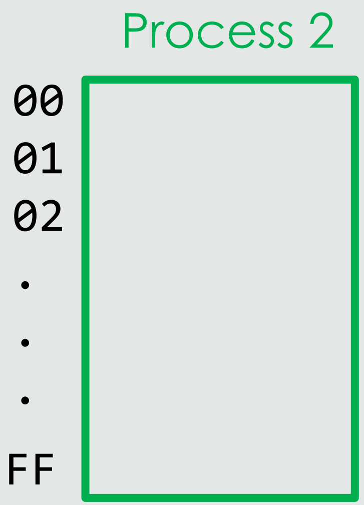
All of this must be stored in virtual memory while the process is active

Process management

Virtual address space

- Processes use a private virtual address space that “pretends” they have exclusive access to main memory

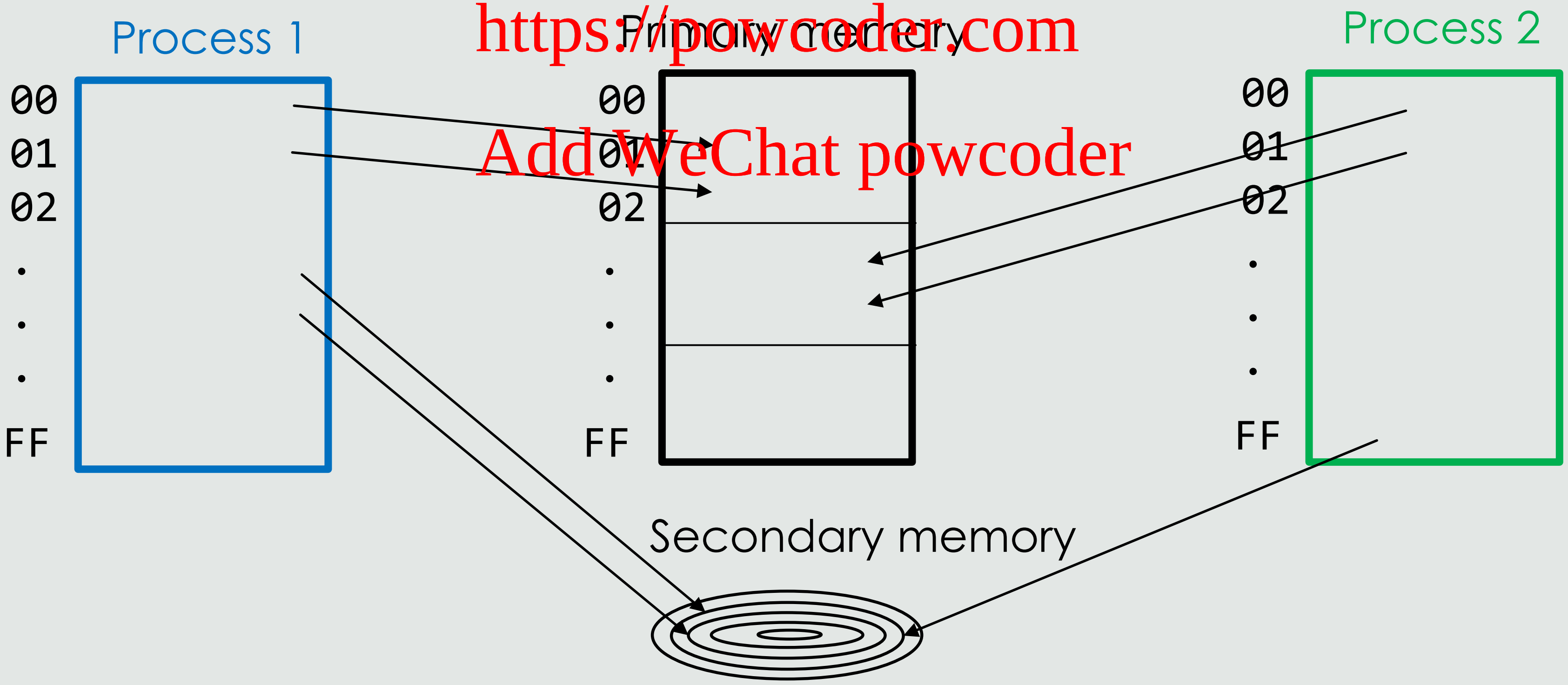




Process management

Virtual address space

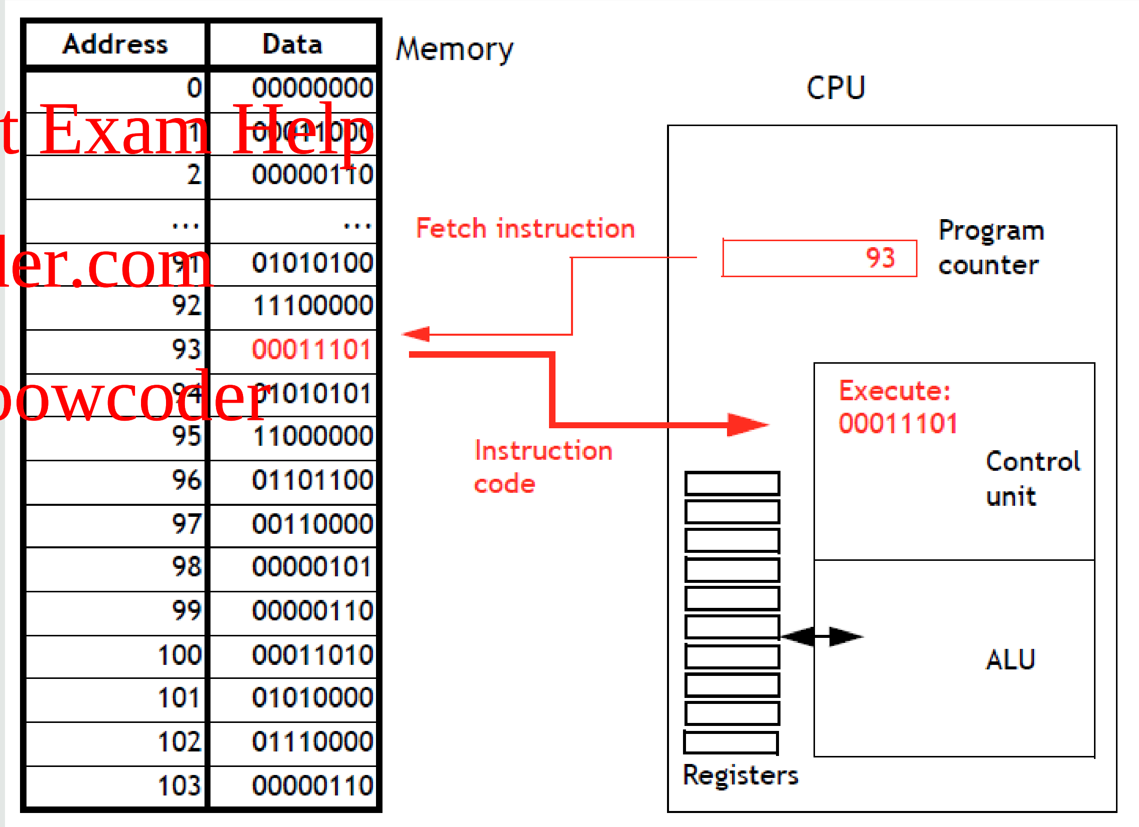
- In reality, processes will occupy different addresses of the same physical memory
- Some of the memory pages may be swapped to secondary memory



Process execution

Control flow #1

- In principle, a central processing unit (CPU) will execute individual instructions in sequence.
- Instructions are retrieved from main memory (or a cache) to be executed by the CPU's control unit.
- The flow of control is represented by the sequence of values stored in the program counter as the system operates.



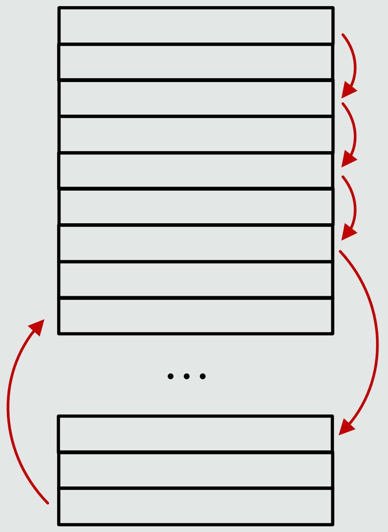
Address	Data
0	00000000
1	00011000
2	00000110
…	…
91	01010100
92	11100000
93	00011101
94	01010101
95	11000000
96	01101100
97	00110000
98	00000101
99	00000110
100	00011010
101	01010000
102	01110000
103	00000110

Basic operation cycle of a computing system

Process execution

Control flow #2

- A simple flow of control might be a sequence of instructions that are stored in adjacent memory locations.
- However, this simple flow of control can change when the sequence is interrupted by use of control structures at the program level (e.g., conditional and unconditional jumps), but also if there are lower-level events that the CPU needs to attend to.

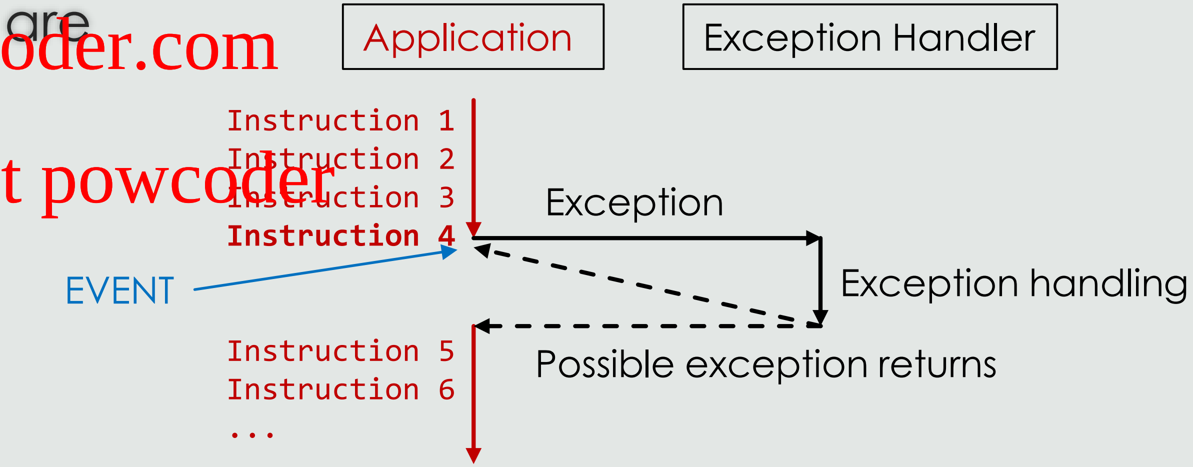


Memory locations

Process execution

Exceptional control flow (ECF)

- Exceptions do not only exist in user applications but also on the operating system level.
- Typical exception classes at those levels are Interrupt, Trap, Fault, and Abort.
- Whenever an exception occurs, control is transferred to an appropriate exception handler.
- ECF is the underlying mechanism for how operating systems control the execution of processes.



Application

Exception Handler

```
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
...
```

EVENT

Exception

Exception handling

Possible exception returns

Process execution

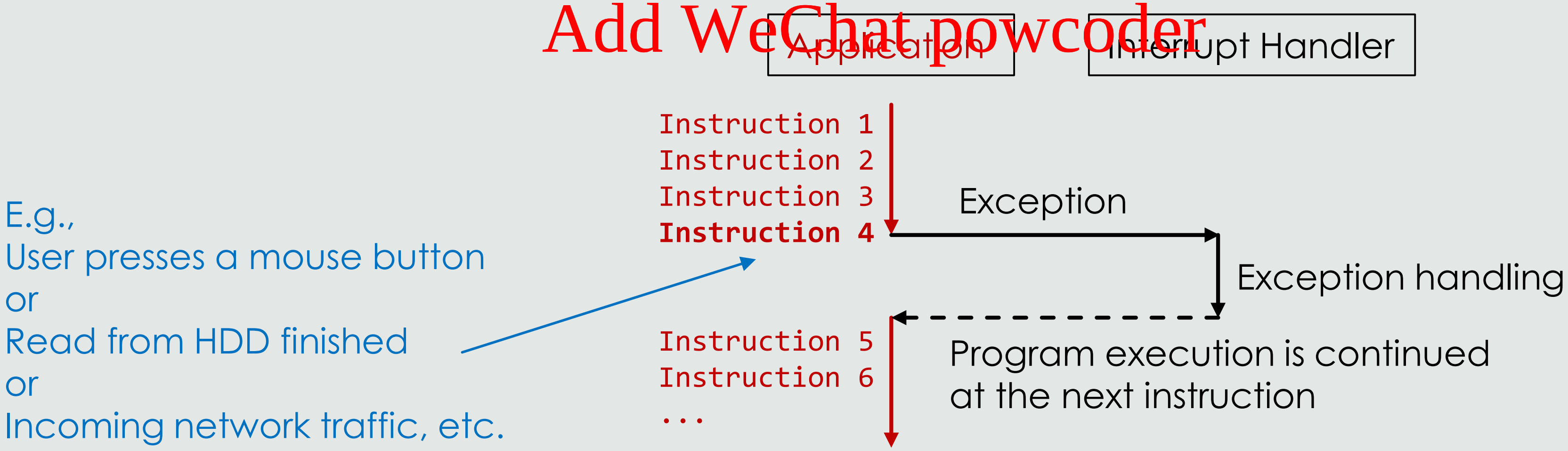
Exception handling

- An exception triggers a sudden transfer of control from the user program to the OS
- Each exception has a unique number
- The OS looks up the correct exception handler from an exception table
- The exception handler runs in kernel mode

Process execution

Interrupt Exception

- Interrupts occur asynchronously as a result of signals from I/O devices that are external to the processor.
- They are not “caused” by program execution as such.

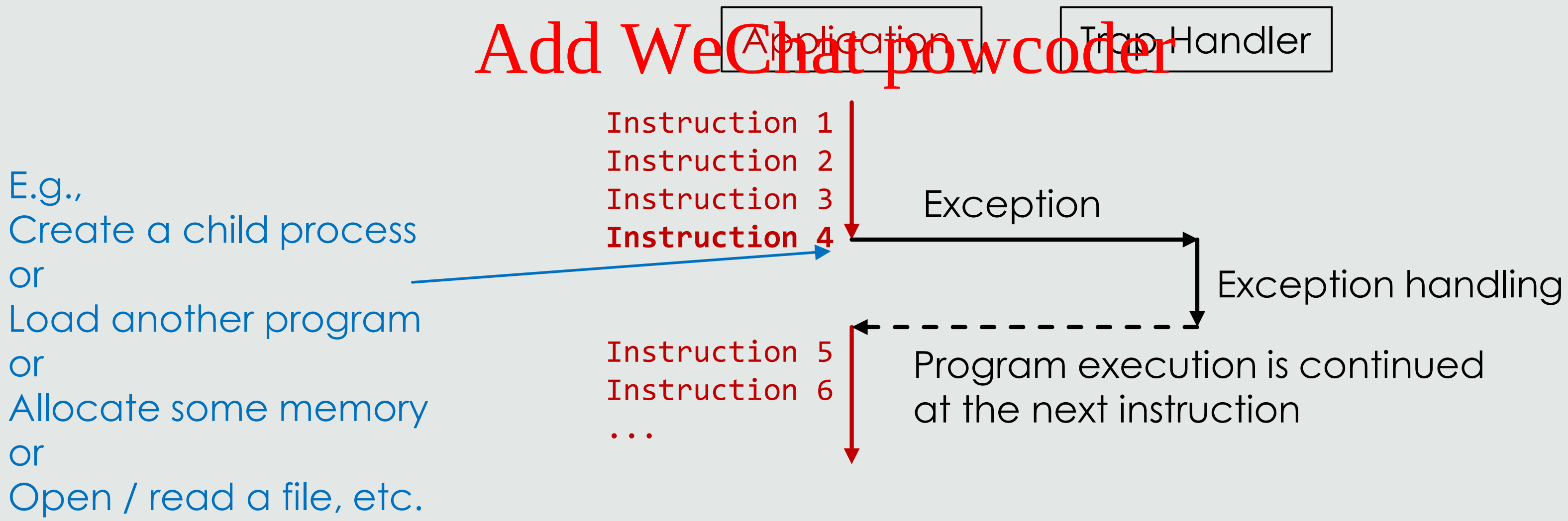


E.g.,
User presses a mouse button
or
Read from HDD finished
or
Incoming network traffic, etc.

Process execution

Trap Exception

- Traps are intentional exceptions that occur as a result of executing an instruction that triggers control transfer to the operating system, usually a system call.



Application

Trap Handler

```
Instruction 1
Instruction 2
Instruction 3
Instruction 4
```

Exception

Exception handling

```
Instruction 5
Instruction 6
...
```

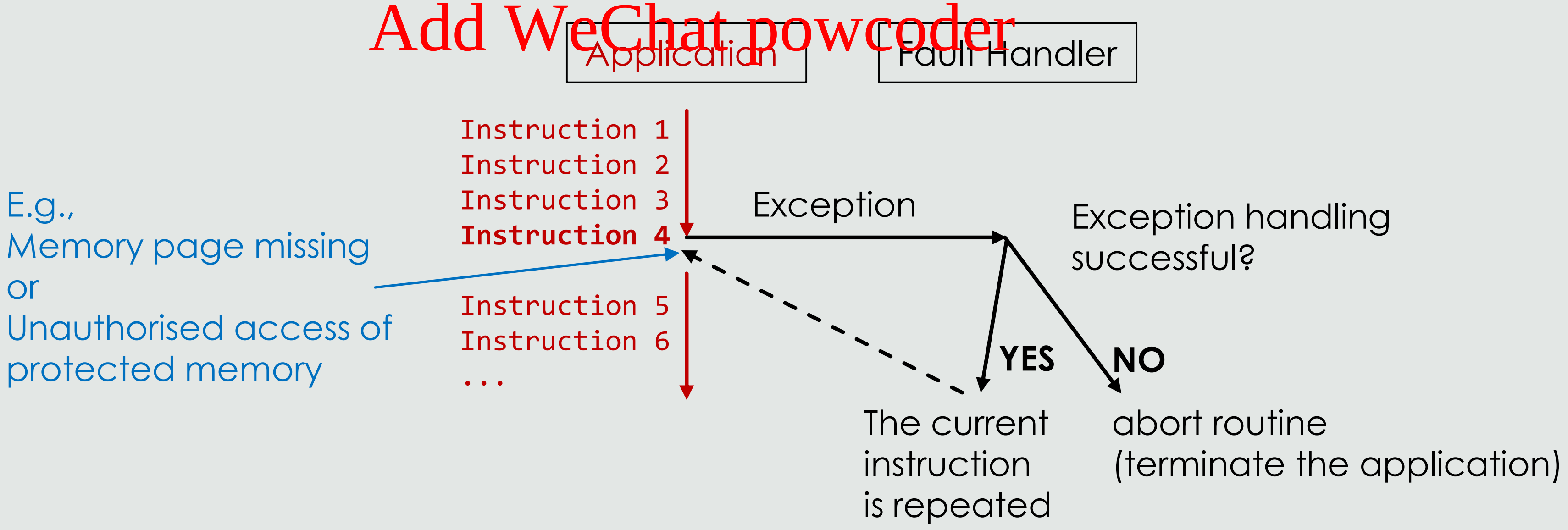
Program execution is continued at the next instruction

E.g.,
Create a child process
or
Load another program
or
Allocate some memory
or
Open / read a file, etc.

Process execution

Fault Exception

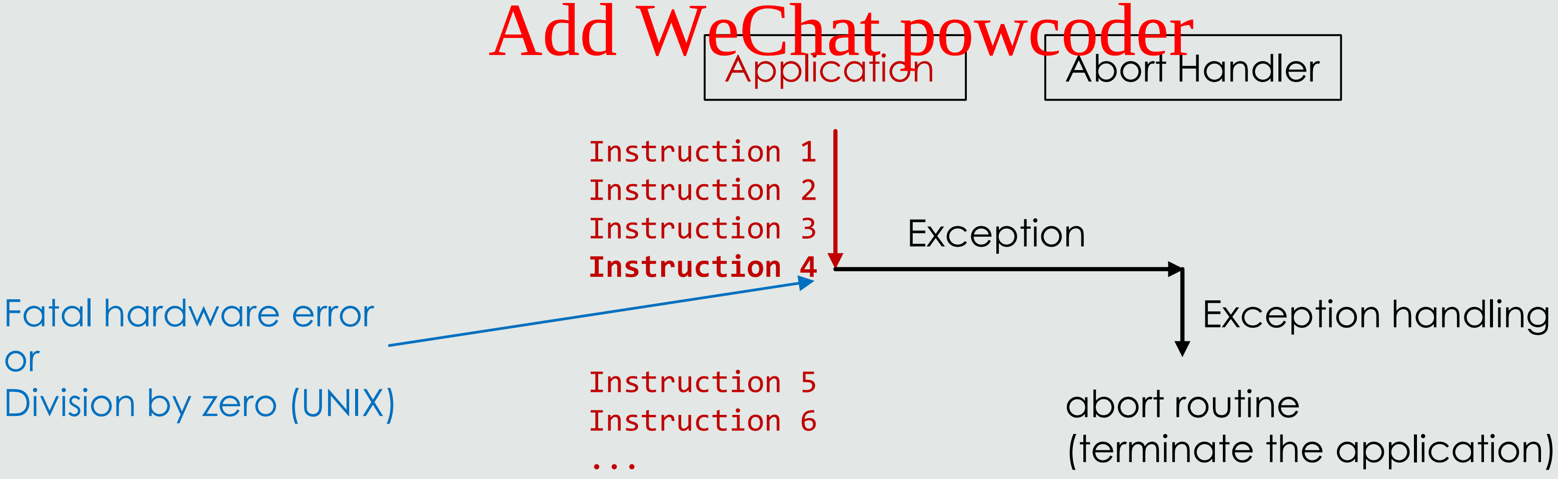
- Faults result from error conditions that a handler might be able to fix.
- If the handler is able to correct the error condition, it returns control to the current instruction for re-execution. Else, the handler runs a routine that terminates the process.



Process execution

Abort Exception

- Aborts result from unrecoverable fatal errors, typically hardware errors such as parity errors that occur when memory bits are corrupted.
- Abort handlers do not return control back to the application program.



Application

Abort Handler

```
Instruction 1
Instruction 2
Instruction 3
Instruction 4

Instruction 5
Instruction 6
...
```

Exception

Exception handling

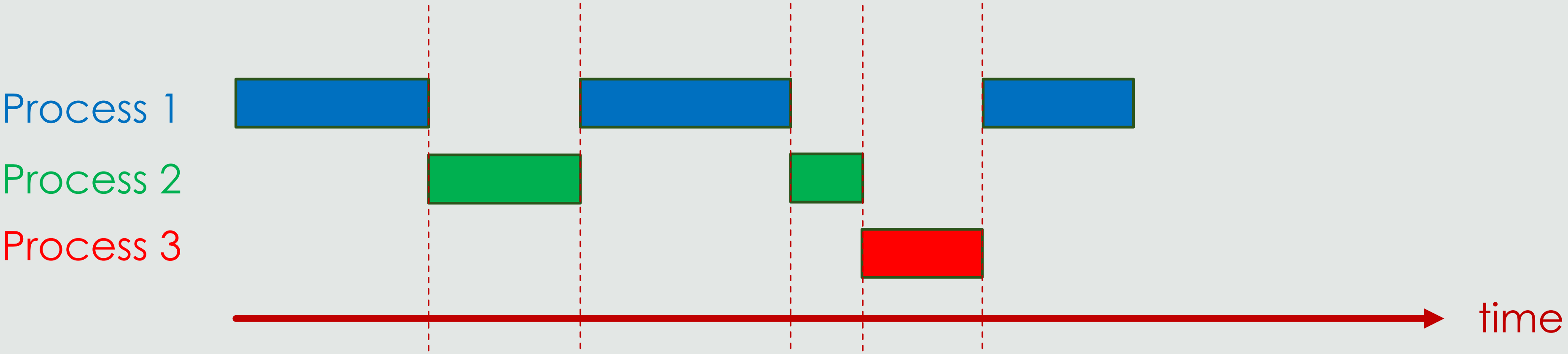
Fatal hardware error
or
Division by zero (UNIX)

abort routine
(terminate the application)

Process execution

Flow of control for concurrent processes

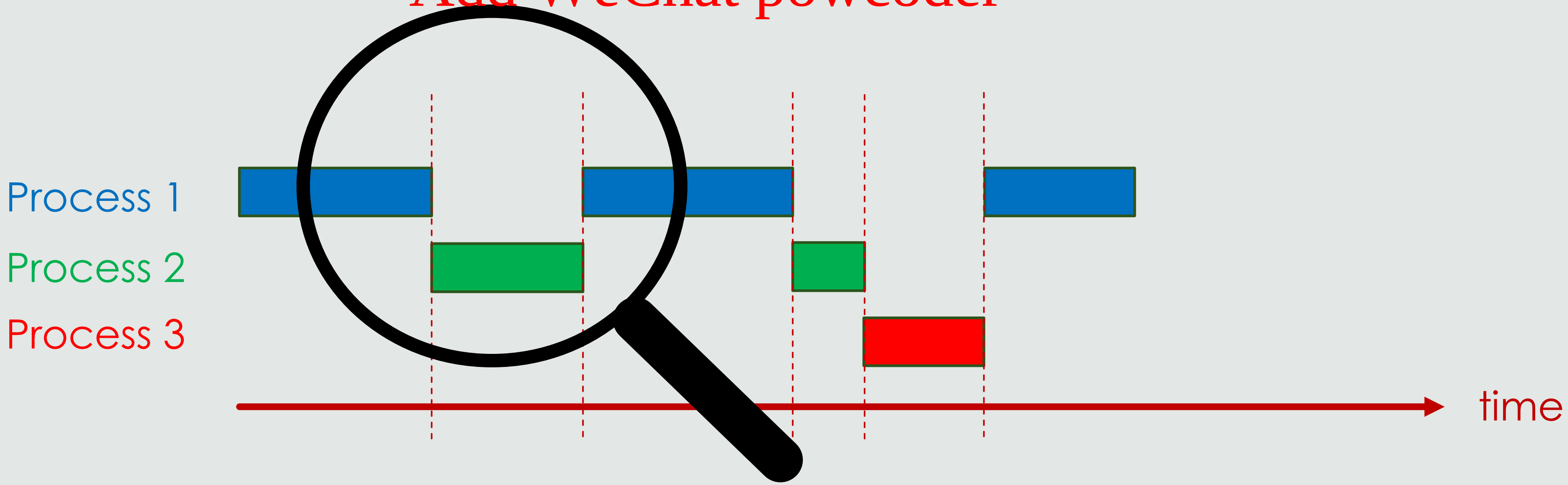
- Processes running concurrently do not have exclusive access to the CPU (Although multi-core CPUs can support parallel execution).
- Each process executes a portion of its normal flow and may then be pre-empted (temporarily suspended) while the other processes take their turns.
- To a program running in the context of one of these processes, appearances are that it has exclusive use of the CPU.



Process execution

Process switching

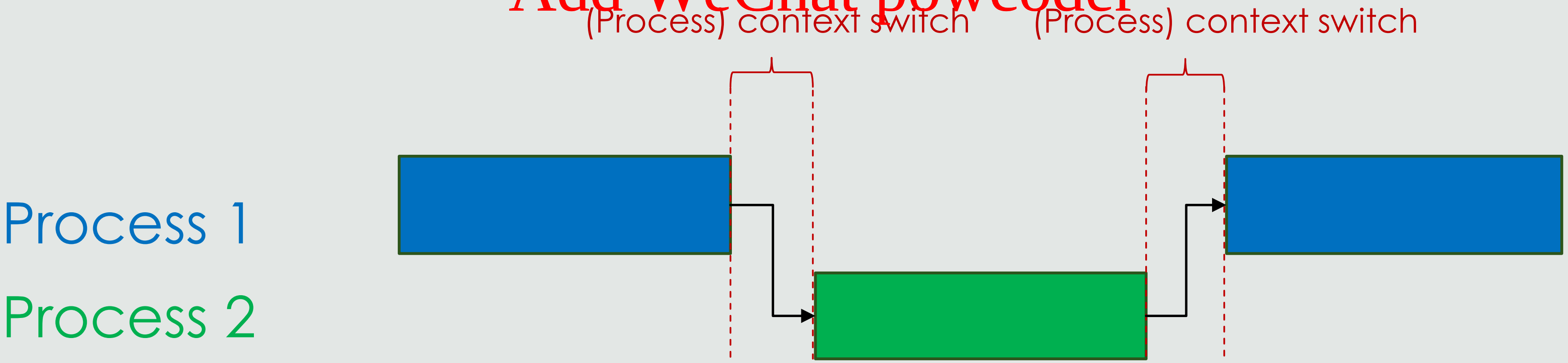
- A process switch is an exception-like activity that the operating system kernel uses to assign a particular process to the CPU to work on.
- The Kernel maintains the context of every process (remember the process image?) and can use this information to pre-empt and resume different concurrent processes.



Process execution

Process switching

- During a process switch, the kernel (1) saves the context of the current process (remember the process image?), then (2) pre-empts that process, and (3) restores the context of another process before (4) passing control to it.

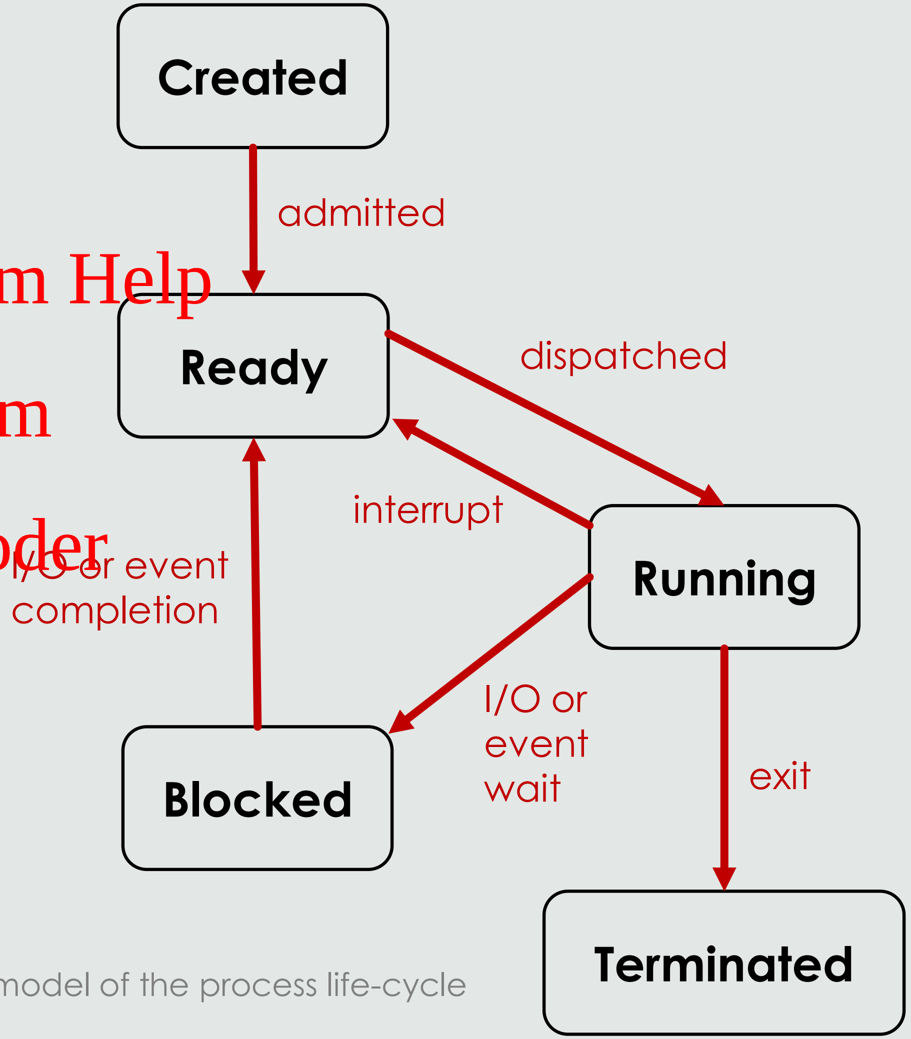


Process execution

Basic process model

- As a process executes, it changes state:
  - Created: Waiting to be admitted for scheduling
  - Ready: Waiting to be assigned to CPU
  - Running: Instructions are being executed
  - Blocked: Waiting for something to happen
  - Terminated: Stopped/finished execution, deletion imminent

How does this work with swapping?

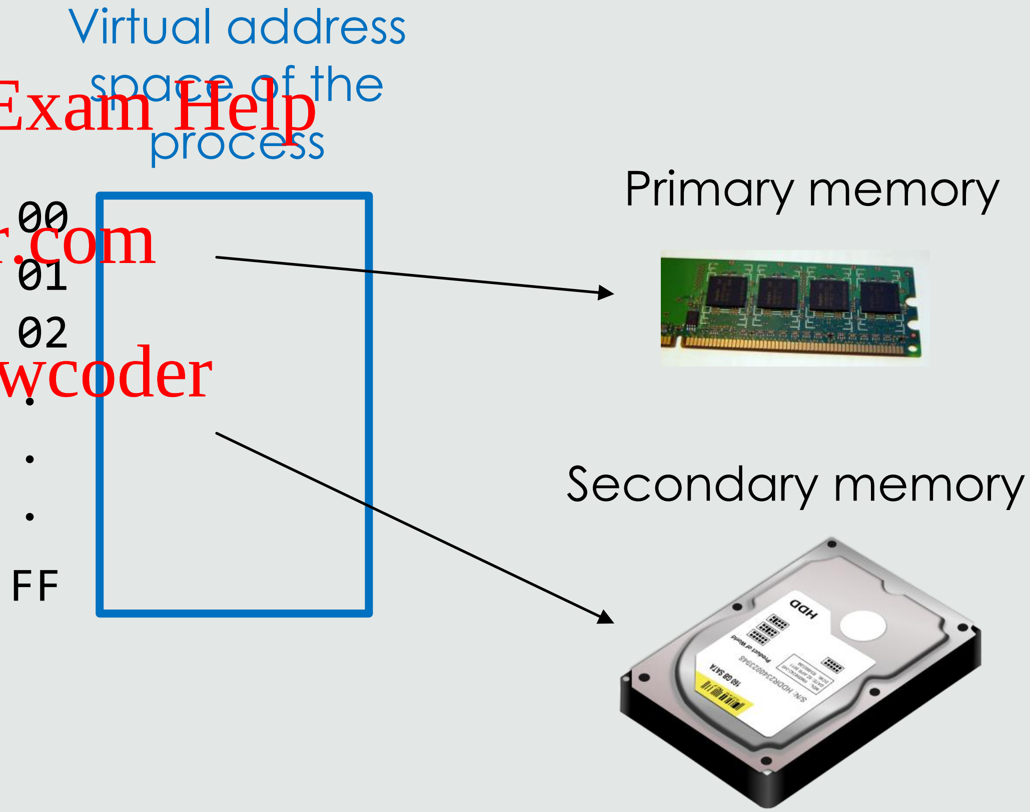


Basic model of the process life-cycle

Process memory management

Swapping

- Virtual memory can make use of secondary storage in order to temporarily suspend certain processes
- Possible reasons:
  - Insufficient memory
  - Process suspected of causing a problem
  - User request, e.g. debugging
  - Timing, e.g. periodic process
  - Parent process request



Virtual address space of the process

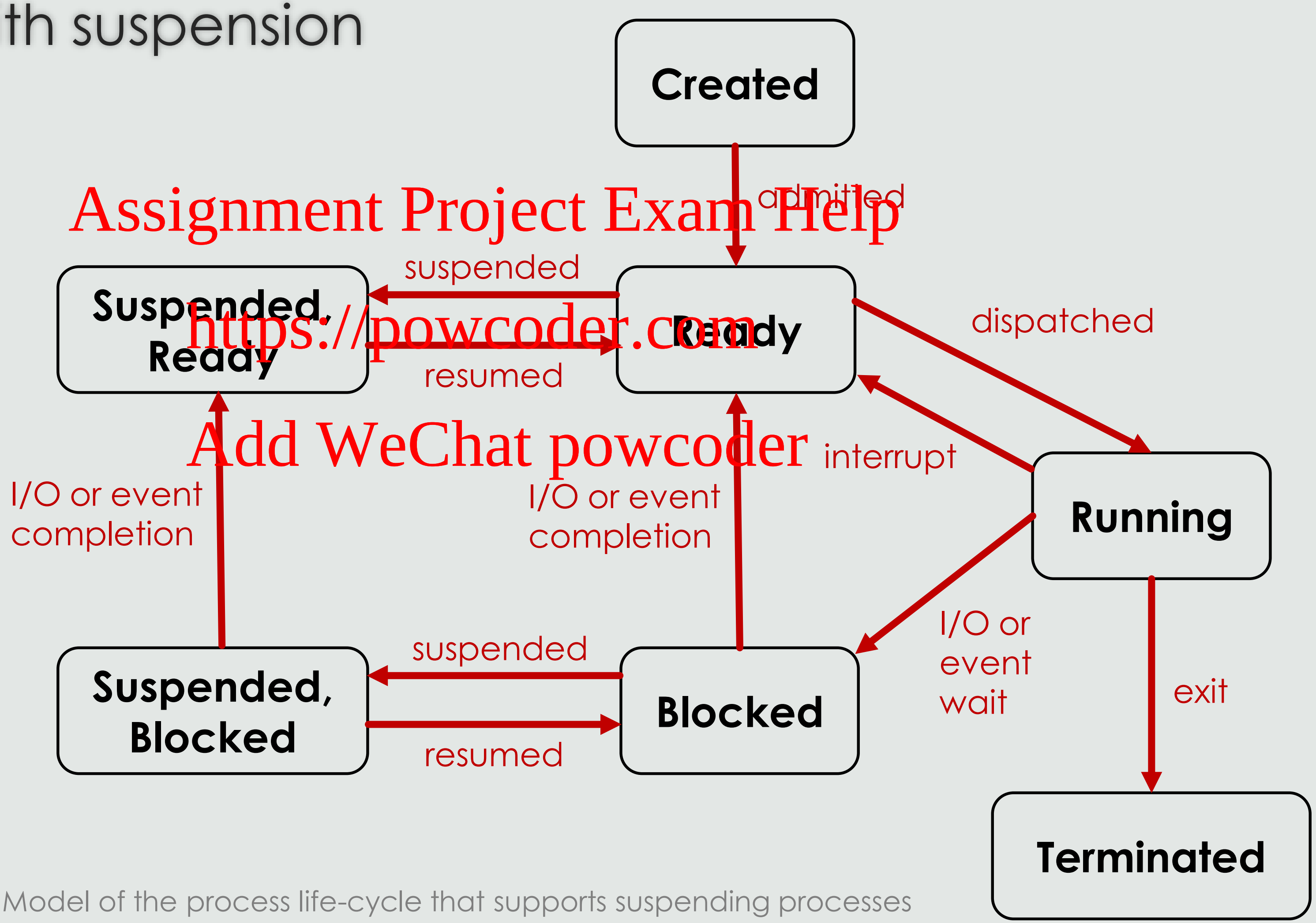
00
01
02
.
.
.
FF

Primary memory

Secondary memory

Process execution

Process model with suspension



Model of the process life-cycle that supports suspending processes

Process termination

Execution stops indefinitely

- Program finished - job done.
- Programmatic error routine - e.g., exception handling
- An unexpected fatal error - e.g., division by 0, illegal memory reference, etc.
- Killed by another process – requires authorisation, e.g., a superior process getting rid of its helper processes.

- Related system calls: **`exit`** **`,`** **`kill`**

Summary

Processes

- Bootstrapping
- Processes
  - Creation (`fork` system call, parents, children)
  - Management (process table, private virtual memory address space, PCB, swapping)
  - Execution (exceptional control flow, scheduling, process switch, process models)
  - Termination (`kill` and `exit` system calls)

Read

- Tanenbaum & Bos., Modern Operating Systems
  - Chapter 2
- Silberschatz et al., Operating System Concepts
  - Chapter 3

Next Lecture

- Introduction
- Operating System Architectures
- Processes
- **Threads**
- Process Scheduling
- Process Synchronisation
- Deadlocks
- Memory Management
- File Systems
- Input / Output
- Security and Virtualisation