# Multiprogramming

# Early computer systems

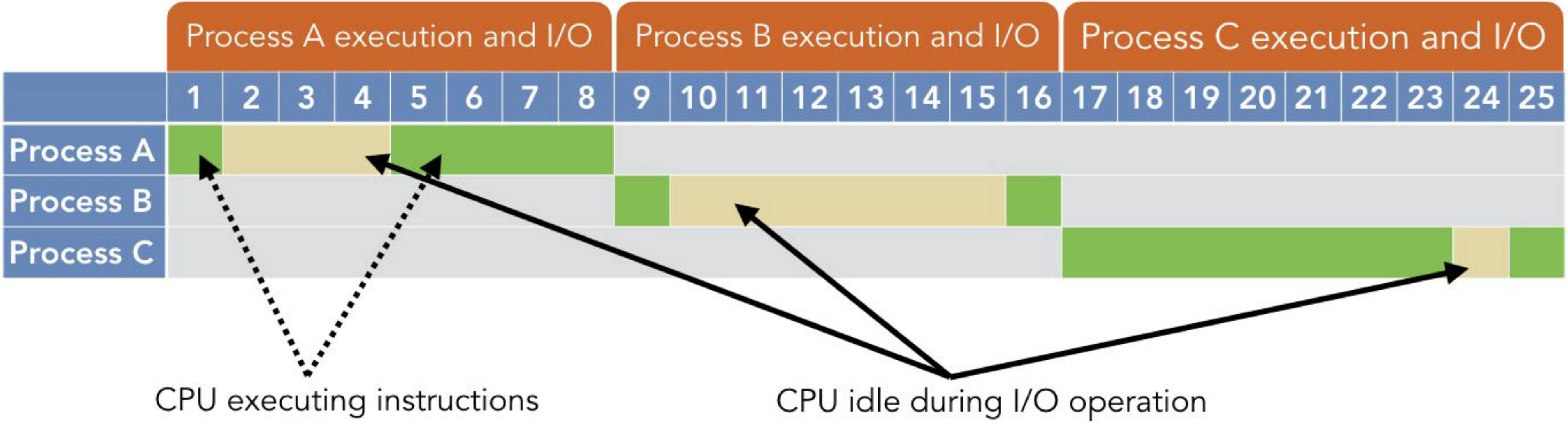
- Run a single program at a time
- Not efficient
- Waste a lot of time to wait

# Batch

- Adding and removing the monitor code as needed
- New job could be started immediately after the previous job finished
- This strategy is called uniprogramming

# Uniprogramming

- One program was started and run in full to completion
- The next job would start immediately after the first one finished
- Programs: CPU instructions and I/O operations
- CPU instructions were very fast, as they consisted of electrical signals
- I/O operations were very slow


Process A execution and I/O
Process B execution and I/O
Process C execution and I/O
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
Process A
Process B
Process C
CPU executing instructions
CPU idle during I/O operation

| Process | CPU time | I/O time |
| --- | --- | --- |
| A | 5 | 3 |
| B | 2 | 6 |
| C | 8 | 1 |
| **Total** | **15** | **10** |

$$\text{CPU utilization} = \frac{5+2+8}{5+2+8+3+6+1} = \frac{15}{25} = 60\%$$

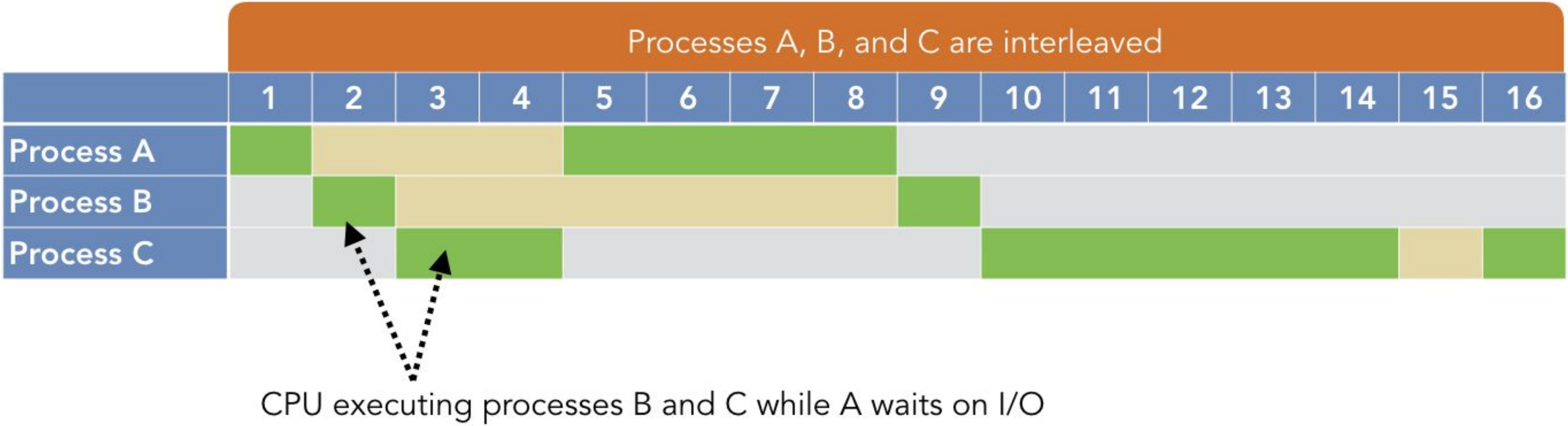
# Uniprogramming

- A significant amount of system resources can be wasted by waiting on I/O operations to complete
- This problem still exists in modern systems
- I/O bottlenecks tend to be among the most significant barriers to high-speed performance

# Multiprogramming

- Multiprogramming
- Several processes are all loaded into memory and available to run
- Whenever a process initiates an I/O operation, the kernel selects a different process to run on the CPU
- Allows the kernel to keep the CPU active and performing work as much as possible

# Multiprogramming



# Multiprogramming

$$\text{CPU utilization} = \frac{5+2+8}{5+2+8+1} = \frac{15}{16} = 93.75\%$$

# Multiprogramming

- There are two forms of multiprogramming that have been implemented
- The most common technique is preemptive multitasking
- The other one is  cooperative multitasking

# Multiprogramming - preemptive multitasking

- Processes are given a maximum amount of time to run
- This amount of time is called a quantum, typically measured in milliseconds
- If a process issues an I/O request before its quantum has expired, the kernel will simply switch to another process early
- If the quantum has expired (i.e., the time limit has been reached), the kernel will preempt the current process and switch to another

# Multiprogramming - cooperative multitasking

- A process can run for as long as it wants until it voluntarily relinquishes control of the CPU or initiates an I/O request
- Cooperative multitasking has a number of advantages, including its simplicity of design and implementation
- If all processes are very interactive (meaning they perform many I/O operations), it can have low overhead costs
- Vulnerable to rogue processes that dominate the CPU time
- EX: if a process goes into an infinite loop that does not perform any I/O operation, it will never surrender control of the CPU and all other processes are blocked from running. As a result, modern systems favor preemptive multitasking.

# Multiprogramming - concurrency

- Multiprogramming creates the foundation for concurrent execution of software
- Concurrency: The ability for multiple entities to make progress toward a goal within a single period of time; creates the appearance of parallel execution (which may be real or illusory)
- EX: Concurrency through multiprogramming is what makes it possible to use a web browser while listening to music with an MP3 player at the same time

# Multiprogramming - concurrency

- A simple way to illustrate multiprogramming in modern software is with the sleep() function
- This function's only argument is the number of seconds to pause the current process
- During this time, the system will switch to other processes that need to run
- Cooperative multitasking.

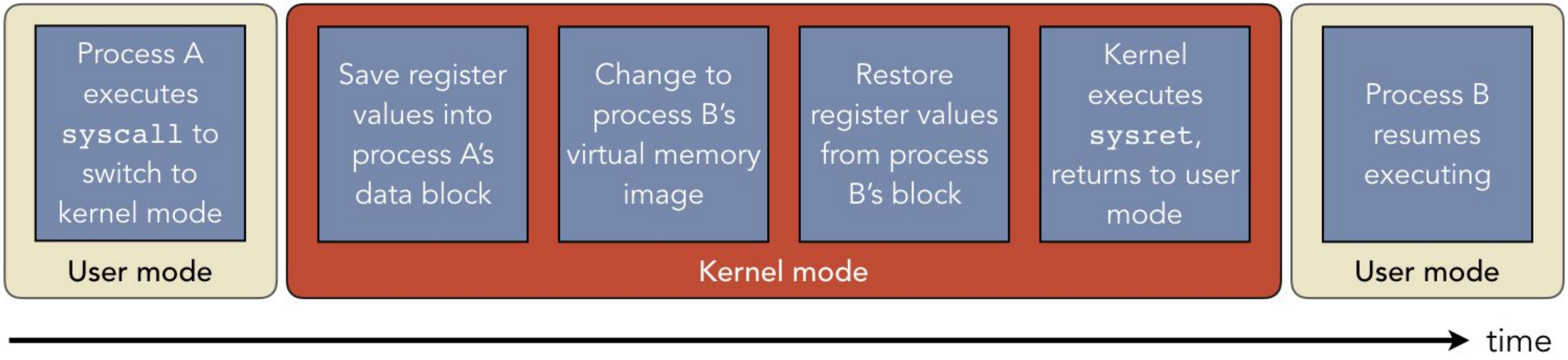
# Multiprogramming - concurrency

```
for (int i = 0; i < 10; i++)
 {
        printf ("Wait after this line!\n");
        sleep (1);
 }
```

# Context Switches

- Context switches form the basis of multiprogramming
- A context switch is the change from one process's virtual memory image to another
- The kernel portion of virtual memory does not change. However, the user-mode code, data, heap, and stack segments that the CPU uses are changed
- Both the old and new processes still reside in physical memory

# Context Switches



# Context Switches

- Although context switches are critical for multiprogramming, they also introduce complexity and overhead costs in terms of wasted time

# Context Switches - Problems

- When the kernel determines that it needs to perform a context switch, it must decide which process it will now make active; this choice is performed by a scheduling routine that takes time to run
- Scheduling: The multiprogramming kernel responsibility to select which process to execute in user mode whenever an interrupt or exception occurs

# Context Switches - Problems

- Context switches introduce delays related to the memory hierarchy. In early cache systems, a context switch required all cache data to be invalidated. As such, all cache data for the old process had to be flushed and replaced with the new process's data
- Newer systems allow different parts of the cache to be associated with different processes; however, this reduces the total amount of data that the cache can store, increasing the number of cache misses

# Context Switches

- How to solve these problems ?
- Increase the quantum given to each process, thereby letting it run longer before forcing a context switch = cooperative multitasking
- The risk is that increasing the time quantum too much allows some processes to dominate the CPU time, effectively monopolizing the system resources
- This trade-off has led to a common practice of 4 - 8 ms per quantum in modern systems

# Some other concepts

- Multiprocessing – A computer using more than one CPU at a time.
- Multitasking – Tasks sharing a common resource (like 1 CPU).
- Multithreading is an extension of multitasking.

# That’s all!