CPSC 457

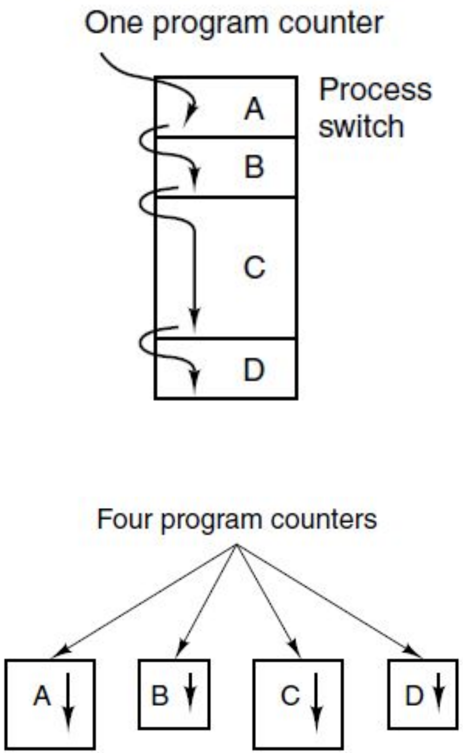
Processes - part 2

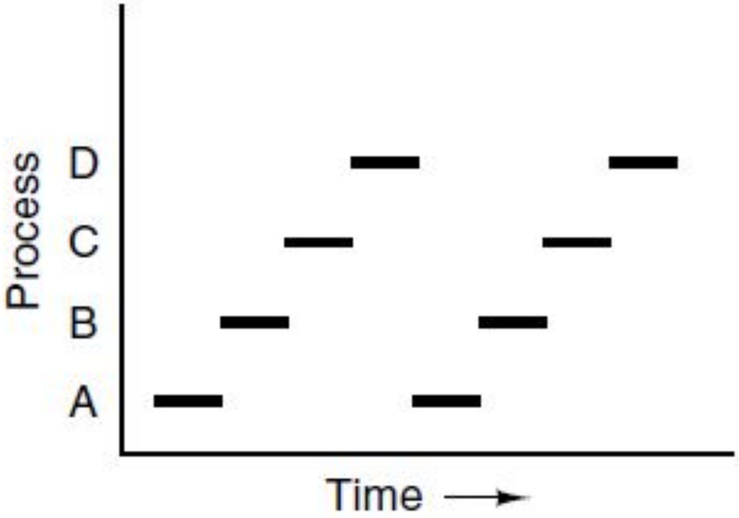
Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Outline

- CPU utilization
- processes creation, termination
- process scheduling
- process states
- context switching

Multiprogramming on a single CPU





CPU utilization

- example:
  - OS is running 4 processes, P1, P2, P3 and P4
    - P1 spends 40% of the time waiting on I/O
    - P2 spends 20% of the time waiting on I/O
    - P3 spends 50% of the time waiting on I/O
    - P4 spends 90% of the time waiting on I/O
  - if there is only one CPU, what will be the CPU's utilization?

    i.e. what percentage of the time is the CPU going to be running 'something'?
- Answer:
  - CPU utilization = probability that at least one of the processes is not waiting on I/O

    = ? ? ?

CPU utilization

- example:
  - OS is running 4 processes, P1, P2, P3 and P4
    - P1 spends **40%** of the time waiting on I/O
    - P2 spends **20%** of the time waiting on I/O
    - P3 spends **50%** of the time waiting on I/O
    - P4 spends **90%** of the time waiting on I/O
  - if there is only one CPU, what will be the CPU's utilization?

    i.e. what percentage of the time is the CPU going to be running 'something'?
- Answer:
  - CPU utilization = probability that at least one of the processes is not waiting on I/O

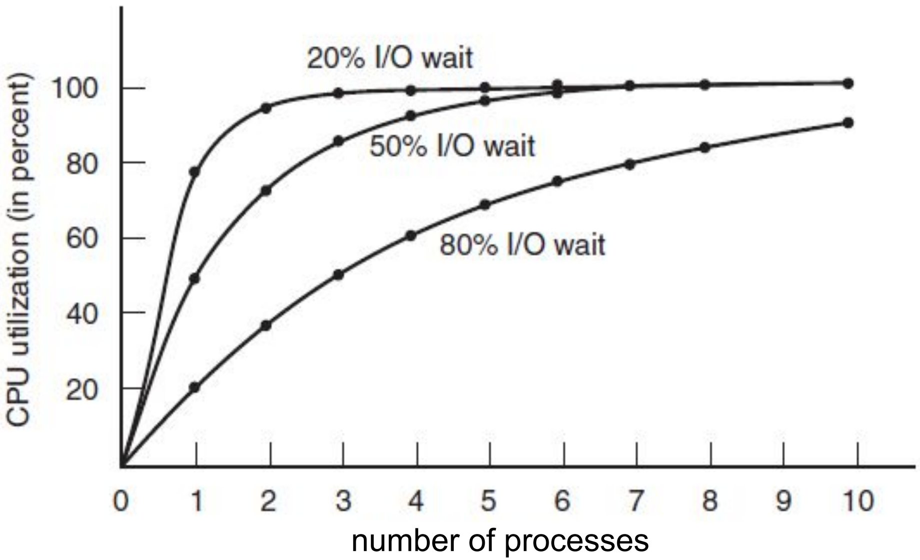
    = 1 - (probability that all processes are waiting on I/O)

    = 1 - (0.4 * 0.2 * 0.5 * 0.9) = 0.964 = 96.4%

CPU utilization - under simplistic multiprogramming model

- assume N similar processes
- each process spends the same fraction P of its time waiting on I/O
- then:

CPU utilization = $1 - P^N$



CPU utilization as a function of the number of processes in memory.

Tanenbaum & Bo,Modern Operating Systems:4th ed.

CPU utilization example

- example:
  - computer has 8GB of RAM
  - let's say 2GB are taken up by OS, leaving 6GB available to user programs
  - user wants to run multiple copies of a program that needs 2GB RAM, with average 80% I/O
  - with 6GB remaining, user could run 3 copies of the program
  - CPU utilization would be = $1 - 0.8^3$ ~= 49%
- is it a good idea to buy 8GB more of RAM?

CPU utilization example

- example:
  - computer has 8GB of RAM
  - let's say 2GB are taken up by OS, leaving 6GB available to user programs
  - user wants to run multiple copies of a program that needs 2GB RAM, with average 80% I/O
  - with 6GB remaining, user could run 3 copies of the program
  - CPU utilization would be = $1 - 0.8^3$ ~= 49%
- is it a good idea to buy 8GB more of RAM?
  - with 14GB remaining, user could run 7 copies of the program
  - CPU utilization would be = $1 - 0.8^7$ ~= 79%
  - throughput increased by 79% - 49% = **30%**
- is it a good idea to buy 8GB more?
  - CPU utilization = $1 - 0.8^{11}$ ~= 91%
  - throughput increased only by 91% - 79% = **12%** (diminishing returns)

Process creation

- in UNIX
  - `init` process is created at boot time by kernel (special case)
  - afterwards, only an existing process can create a new processes, via **`fork()`**
  - therefore all processes are descendants of **`init`**
  - in many modern Linux distributions **`init`** is replaced by **`systemd`**
  - **`fork()`** often followed by **`exec*()`** to spawn a different program,

    or you could use a the **`system()`** convenience function

- in Windows: **`CreateProcess()`** is used to create processes,

  but the behavior is quite different from **`fork()`**

Process creation

- a process can create new processes for a variety of reasons
  - system initialization (boot)
    - background processes – daemons, services
  - application decides to spawn additional processes
    - eg. to execute external programs or to do parallel work
  - a user requests to create a new process
    - eg. launching an application from desktop
  - starting a batch job
    - mainframes

Address space

- each process has its own address space, created during process creation

Parent
Stack
Heap
Data
Text Code: `fork()`
PCB

`fork()` →

- duplicate address space (almost identical)
- next instruction is the same
- but *code flow* may differ

Parent	Child
Stack	Stack
Heap	Heap
Data	Data
Text Code: `wait()`	Text Code: `execlp(...)`
PCB	PCB

Resource allocation

- several options for allocating resources for a new process, for example:
  - child obtains resources directly from the OS
    - most common, easiest to implement
  - child obtains subset of parent's resources
    - eg. parent decides to give up some if its resources so that a child can use them
  - parent shares some/all resources with the child
  - hybrids
- cons/pros...
  - should a process be allowed to exhaust resources of the entire OS?

Common parent-child execution scenarios

- when child process is created, parent process usually does one of three things:
  1. the parent waits until the child process is finished
     - often used when child executes another program, eg. **`fork/exec()`**, or **`system()`**
  2. the parent continues to execute concurrently and independently of the child process
     - eg. autosave feature
  3. the parent continues to execute concurrently, but synchronizes with the child
     - can be quite complicated to synchronize

```
pid = fork()
if pid > 0 :
    wait()
```

```
pid = fork()
if pid > 0 :
    do_whatever()
    exit()
```

```
pid = fork()
if pid > 0 :
    do_something_1()
    synchronize()
    do_something_2()
    synchronize()
    ...
```

Process termination

- typical conditions which terminate a process:
  - voluntary:
    - **normal exit** - eg. application decides it's done, or user instructs an app to 'close'
      - application calls **`exit()`** or **`ExitProcess()`**
    - **error exit** - application detects an error, optionally notifies user
      - application calls **`exit()`** or **`ExitProcess()`**
  - involuntary:
    - **fatal error**
      - usually due to a bug in the program, detected by OS
      - eg. accessing invalid memory, division by zero
    - **involuntary** - killed by another process
      - parent, or another process calls **`kill()`** or **`TerminateProcess()`**
      - eg. during shutdown, pressing <ctrl-c> in terminal, closing GUI window

Process termination

- parent may terminate its children for different reasons, for example:
  - the child has exceeded its usage of some of the resources
  - the task assigned to the child is no longer required
  - the parent needs/wants to exit and wants to clean up first
- in Unix, when a parent process is terminated:
  - the child processes may be terminated, or assigned to the grandparent process, or to the `init` process
  - process hierarchy is always maintained
- default behavior on Linux is to **reparent** the child process to the `init` process
  - this can be changed (eg. to kill children, reparent to some other process)
  - see `prctl()` for more details

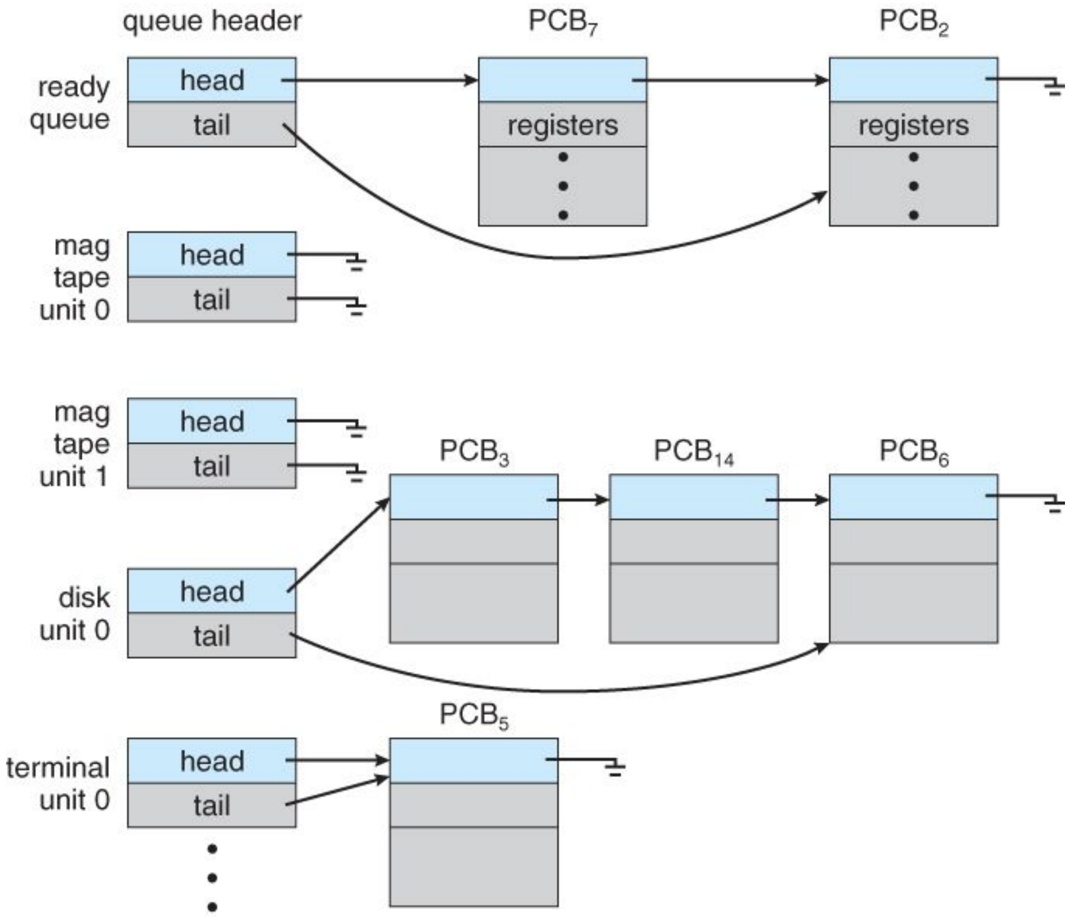
Process termination

- when terminating a process, what happens to resources owned by the process?
  - OS must free all related resources, eg.
  - free memory used by the process
  - delete PCB
  - delete process from process table
  - kill children or assign them a new parent
  - close files
  - close network connections
  - …

Process scheduling

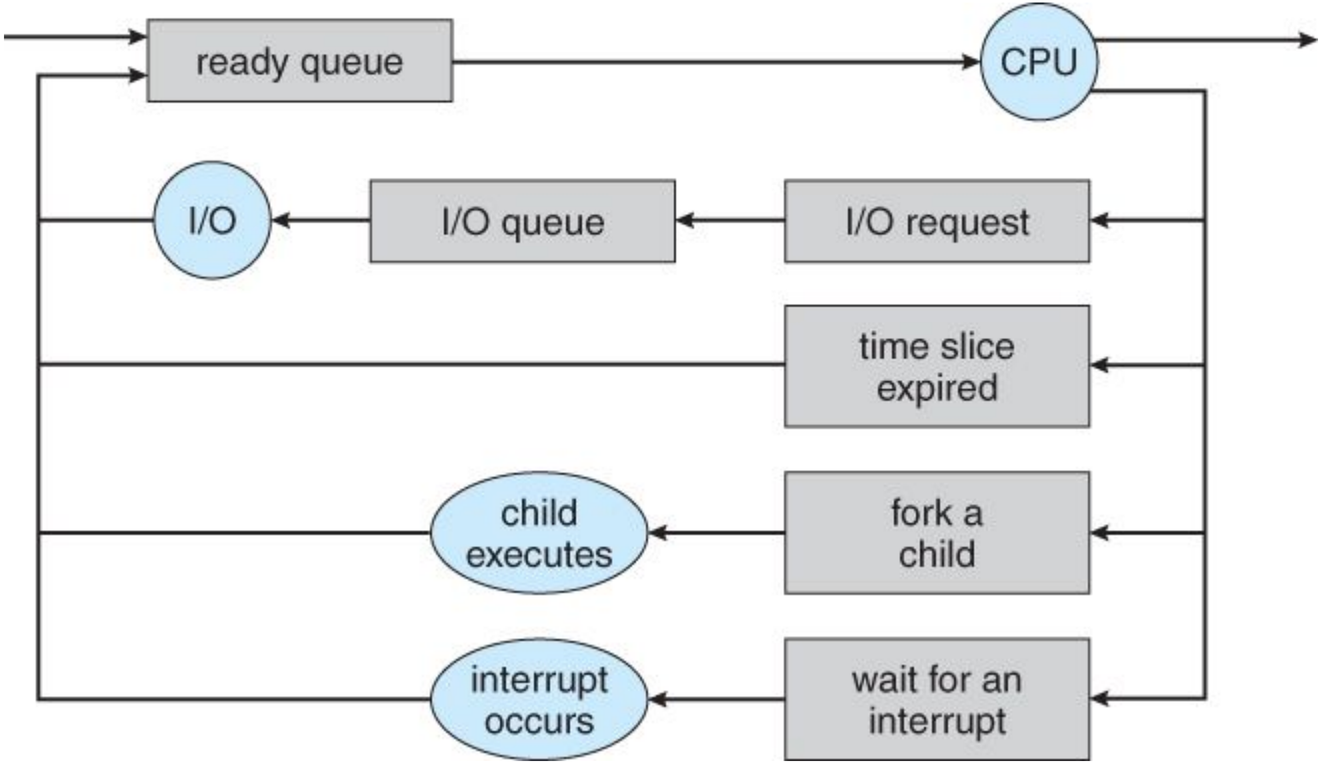
- part of multitasking is deciding which process gets the CPU next
- typical objective is to maximize CPU utilization
- **process scheduler**:
  - kernel routine/algorithm that selects an available process to execute on the CPU
  - selected from processes in a ready queue
- OS maintains many different scheduling queues:
  - job queue: all programs waiting to run, usually found in batch systems
    - eg. priority queue
  - ready queue: all processes that are ready to execute their next instruction
    - eg. linked list, implemented via a pointers in PCBs
  - device queues: processes waiting for a particular device
    - each device has its own queue

Queues



https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html

Process scheduling diagram



https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html

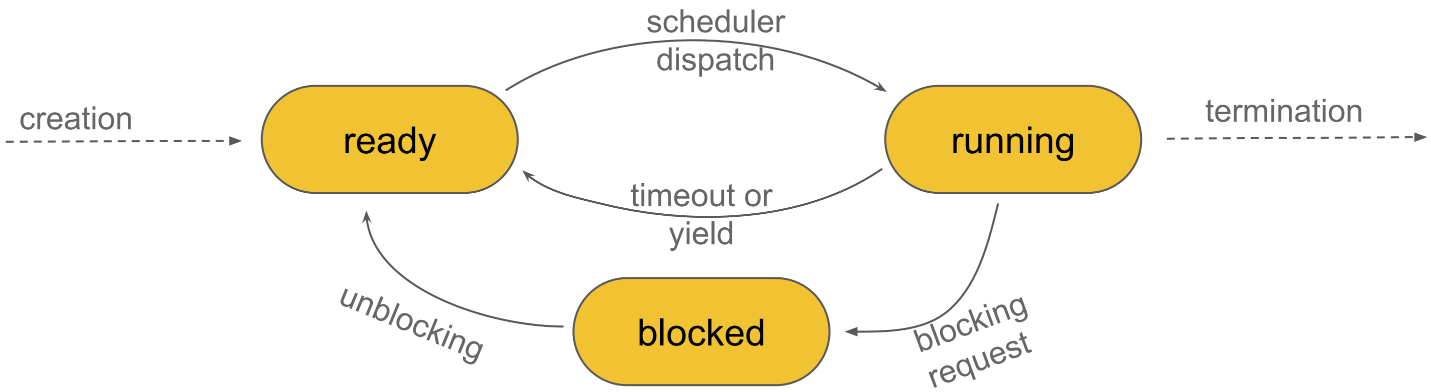
Process states

3 process states:

- **running** — actually running on the CPU
- **blocked** — waiting for some event to occur, eg. I/O
- **ready** — the process is ready to execute on CPU

only 4 transitions are possible:

- ready → running
- running → ready
- running → blocked
- blocked → ready

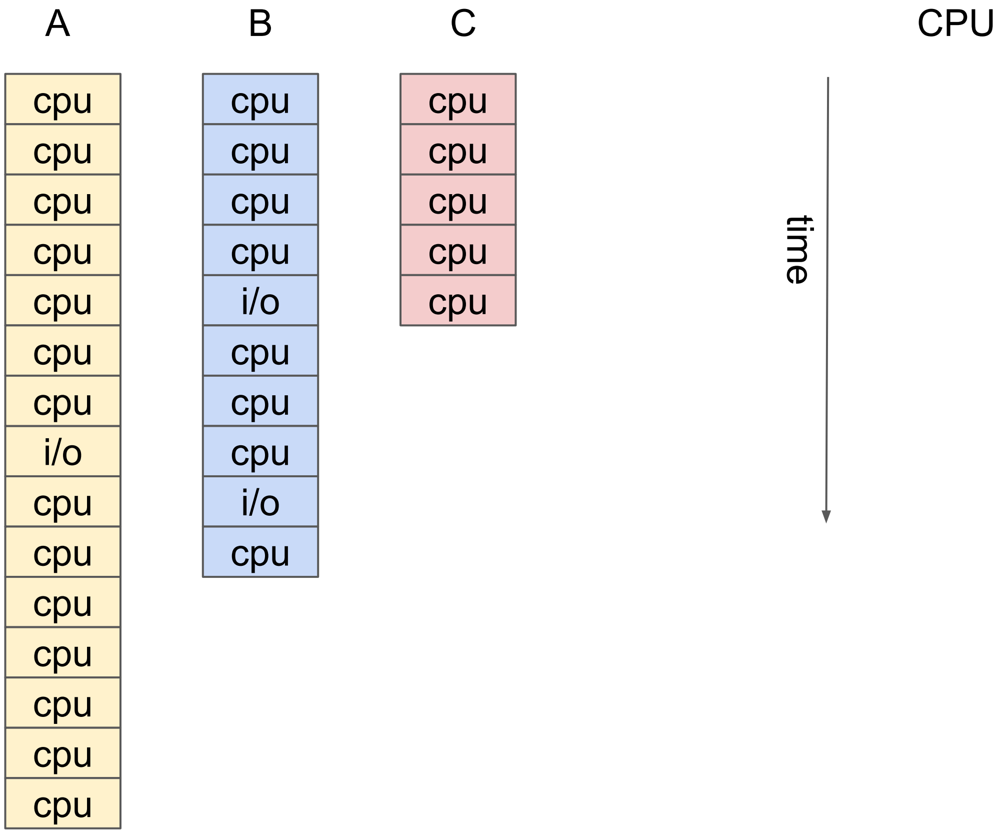


Exercise – simulating round-robin scheduling

- simulate 3 processes A, B, C
  - A: cpu x 7, i/o, cpu x 7
  - B: cpu x 4, i/o, cpu x 3, i/o, cpu x 1
  - C: cpu x 5
- time slice = 3 cycles
  - give the CPU to a process for at most 3 cycles
  - after 3 cycles switch to the next process
  - if process is doing I/O switch to the next process
- assume I/O is very short, only 1 CPU cycle

A	B	C
cpu	cpu	cpu
cpu	cpu	cpu
cpu	cpu	cpu
cpu	cpu	cpu
cpu	i/o	cpu
cpu	cpu	
cpu	cpu	
i/o	cpu	
cpu	i/o	
cpu	cpu	
cpu		
cpu		
cpu		
cpu		
cpu		

Exercise

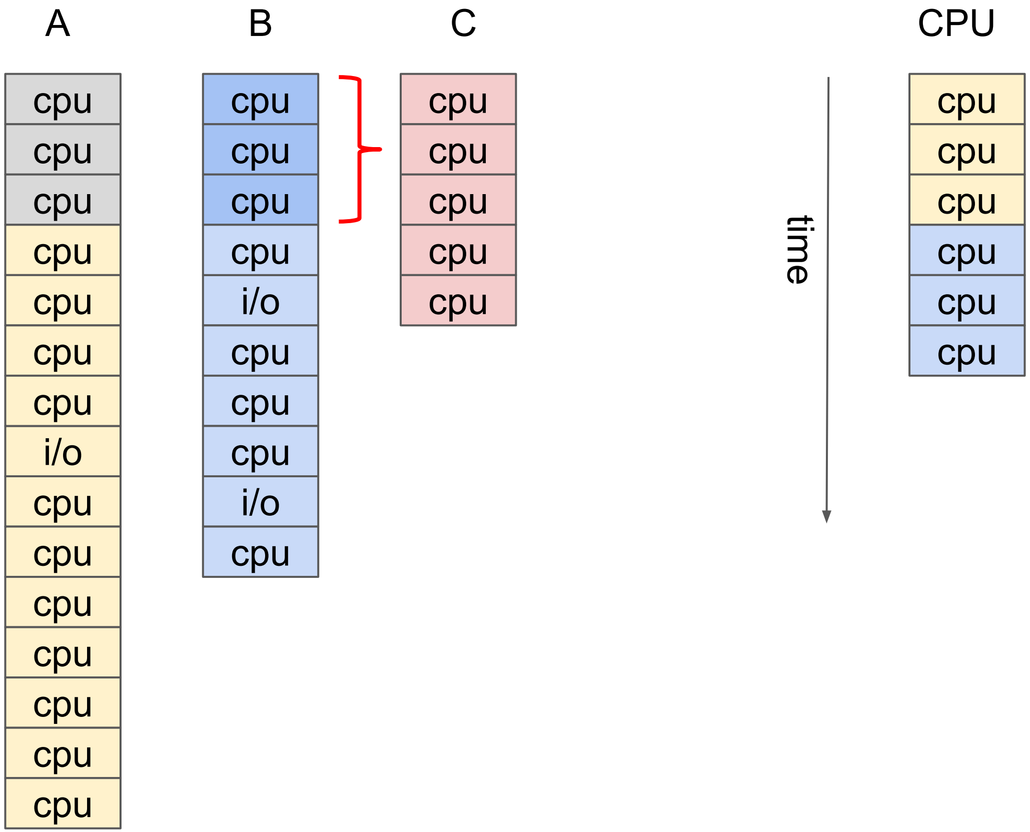

A
B
C
CPU
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
time

Exercise

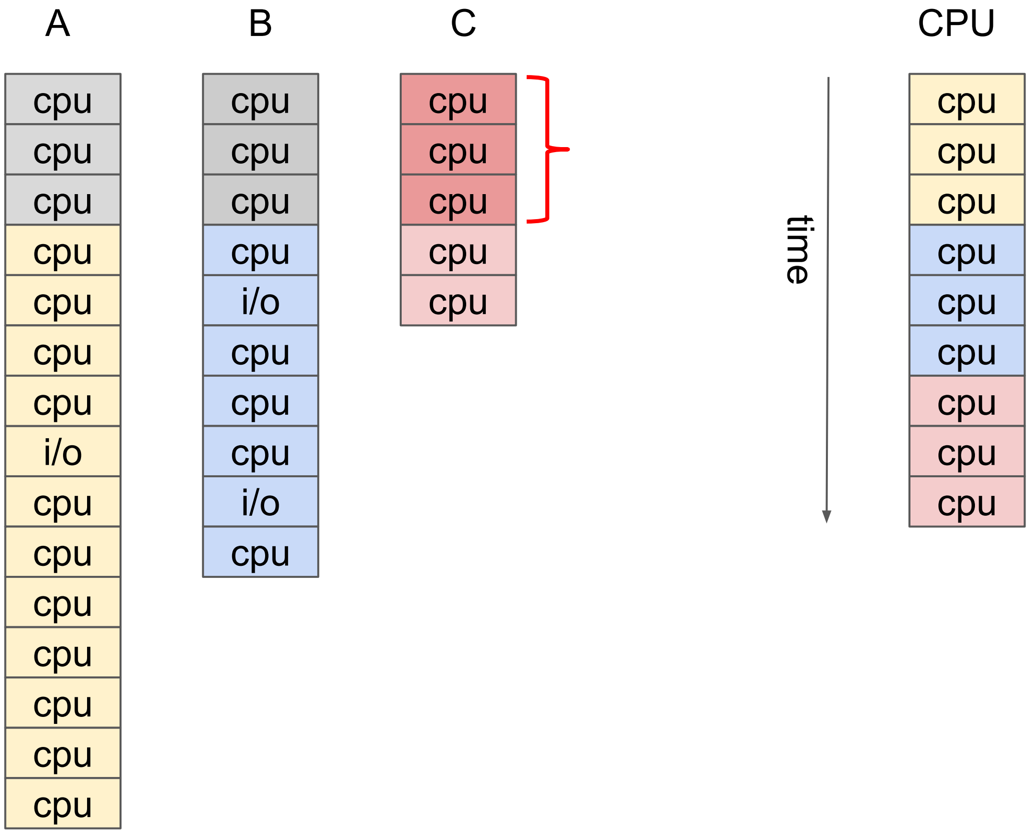
A	B	C	CPU
cpu	cpu	cpu	cpu
cpu	cpu	cpu	cpu
cpu	cpu	cpu	cpu
cpu	cpu	cpu	
cpu	i/o	cpu	
cpu	cpu		
cpu	cpu		
i/o	cpu		
cpu	i/o		
cpu	cpu		
cpu			
cpu			
cpu			
cpu			
cpu			

time

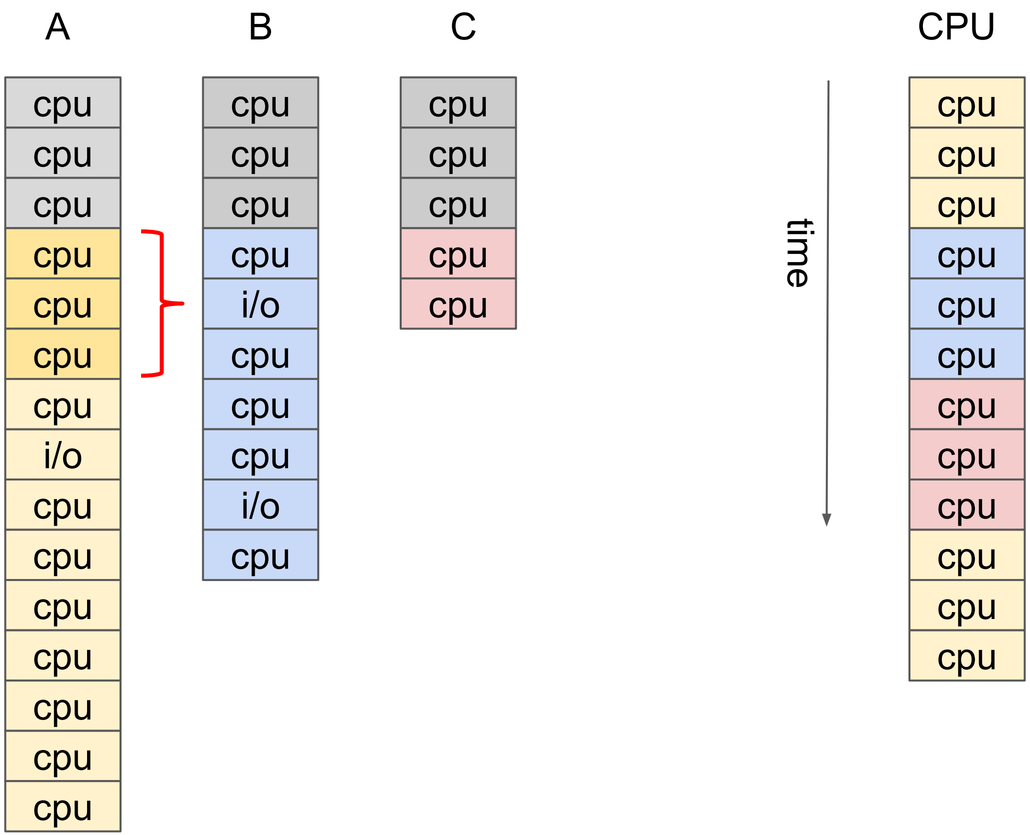
Exercise



Exercise

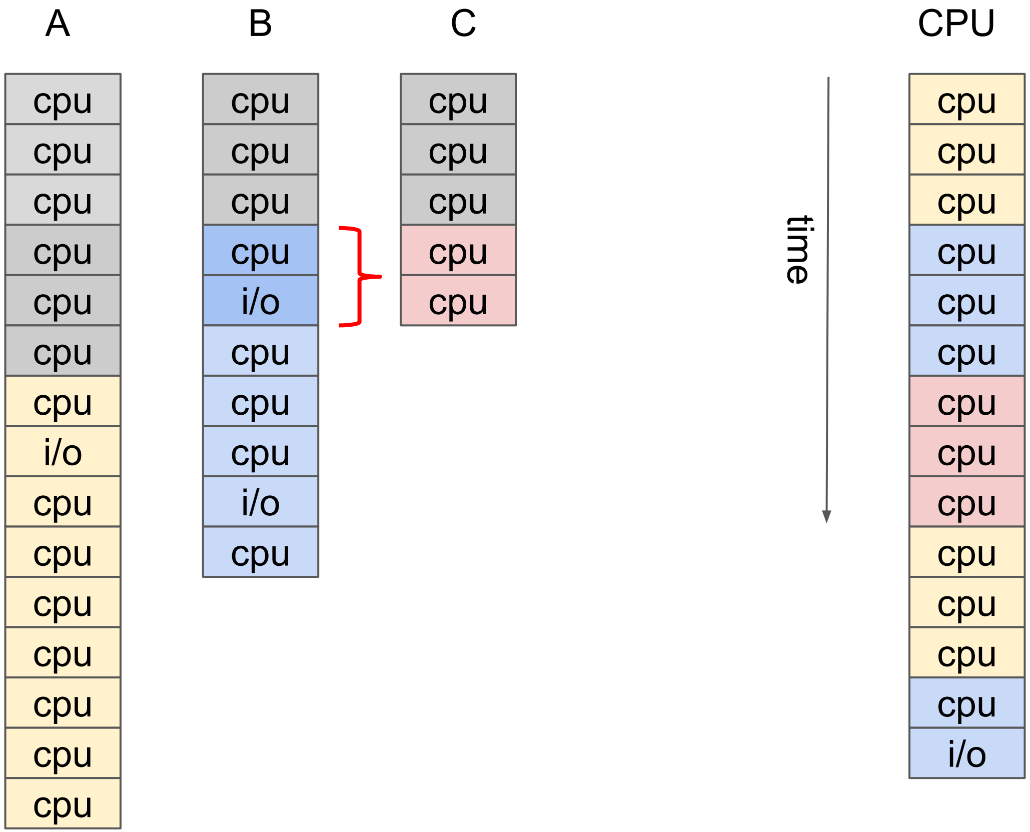

A
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
B
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
C
cpu
cpu
cpu
cpu
cpu
time
CPU
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu

Exercise



A
B
C
CPU
time
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu

Exercise

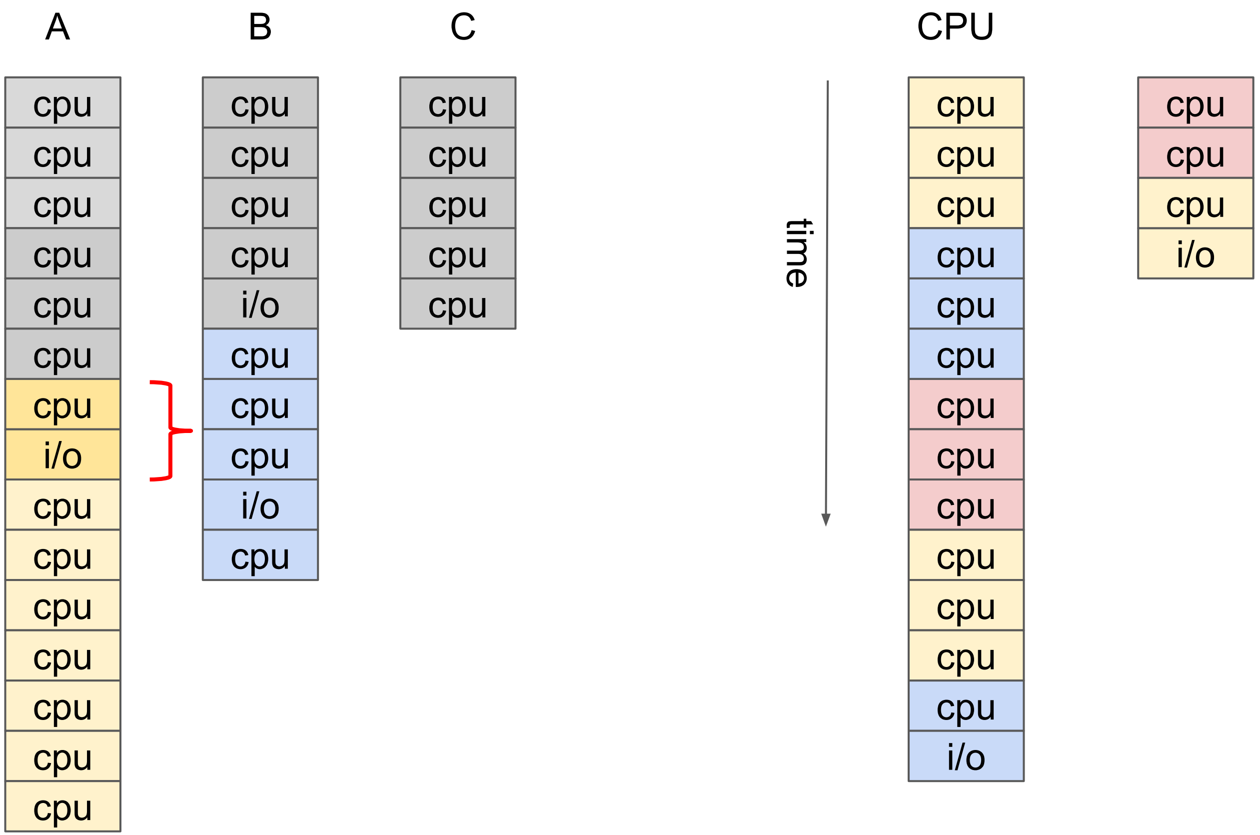

A
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
B
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
C
cpu
cpu
cpu
cpu
cpu
time
CPU
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o

Exercise

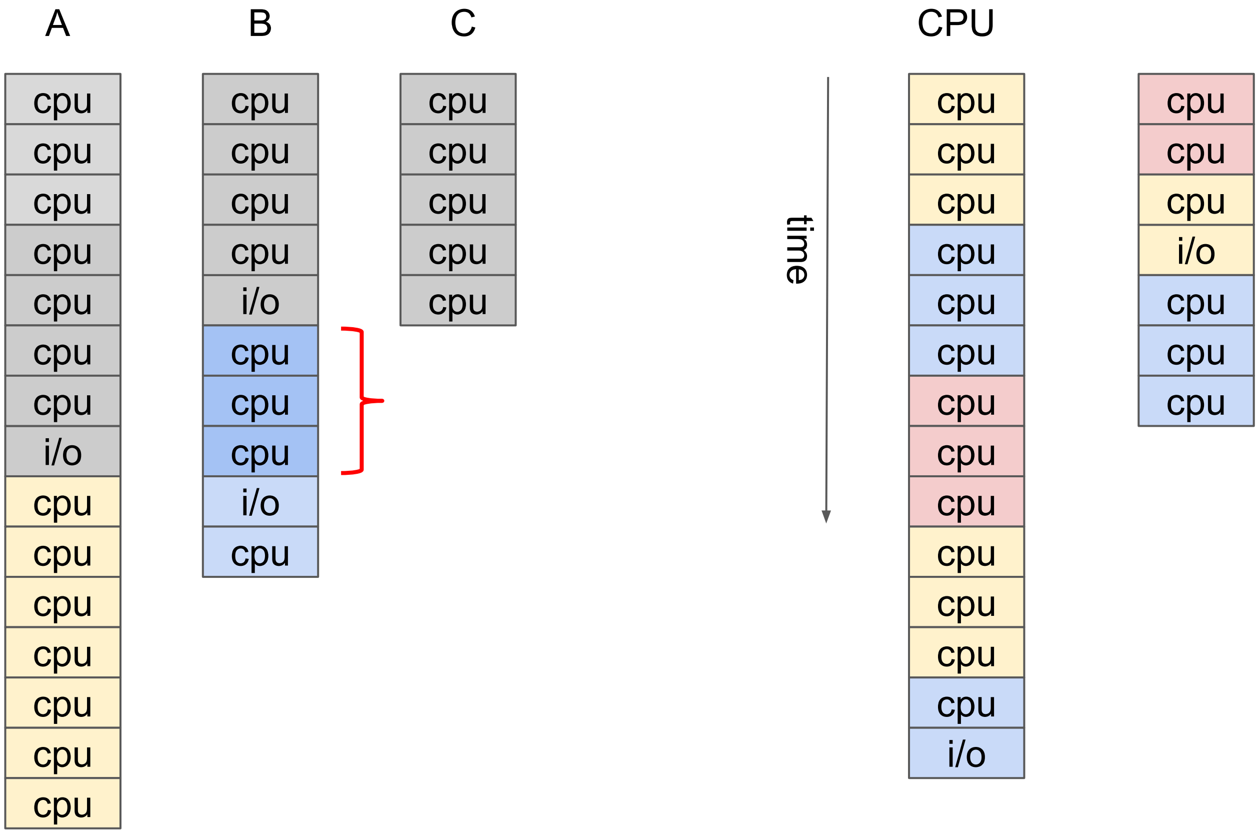
A	B	C	CPU	
cpu	cpu	cpu	cpu	cpu
cpu	cpu	cpu	cpu	cpu
cpu	cpu	cpu	cpu	
cpu	cpu	cpu	cpu	
cpu	i/o	cpu	cpu	
cpu	cpu		cpu	
cpu	cpu		cpu	
i/o	cpu		cpu	
cpu	i/o		cpu	
cpu	cpu		cpu	
cpu			cpu	
cpu			cpu	
cpu			cpu	
cpu			i/o	
cpu				

time

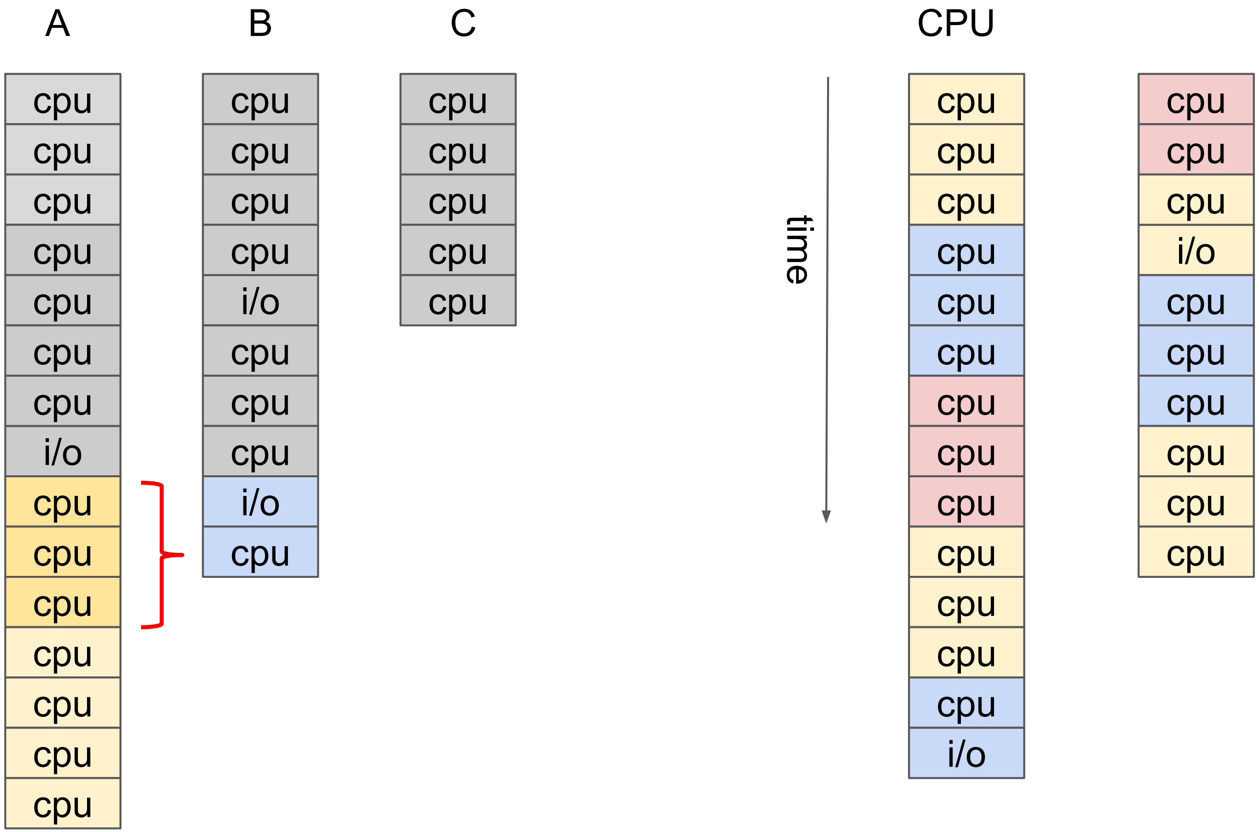
Exercise


A
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
B
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
C
cpu
cpu
cpu
cpu
cpu
time
CPU
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o

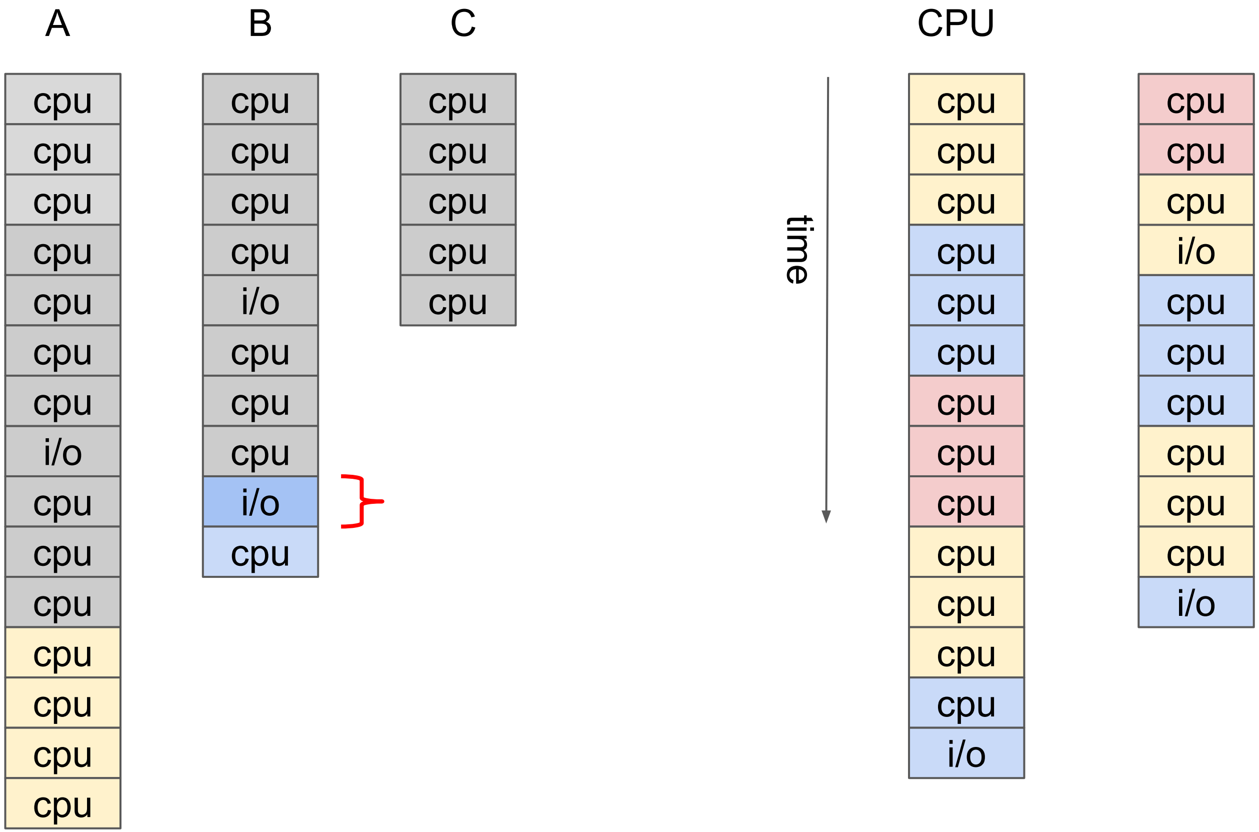
Exercise


A
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
B
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
C
cpu
cpu
cpu
cpu
cpu
time
CPU
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
cpu
cpu

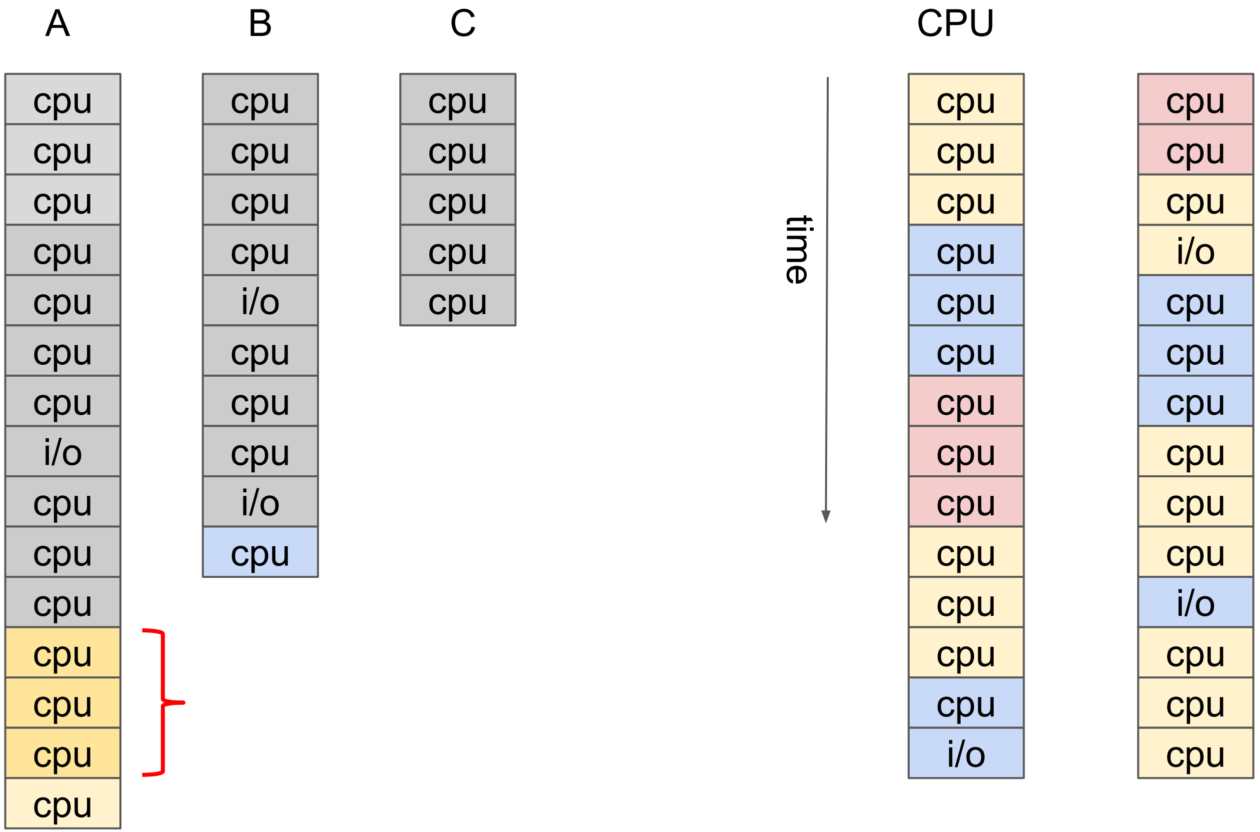
Exercise


A
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
B
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
C
cpu
cpu
cpu
cpu
cpu
CPU
time
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu

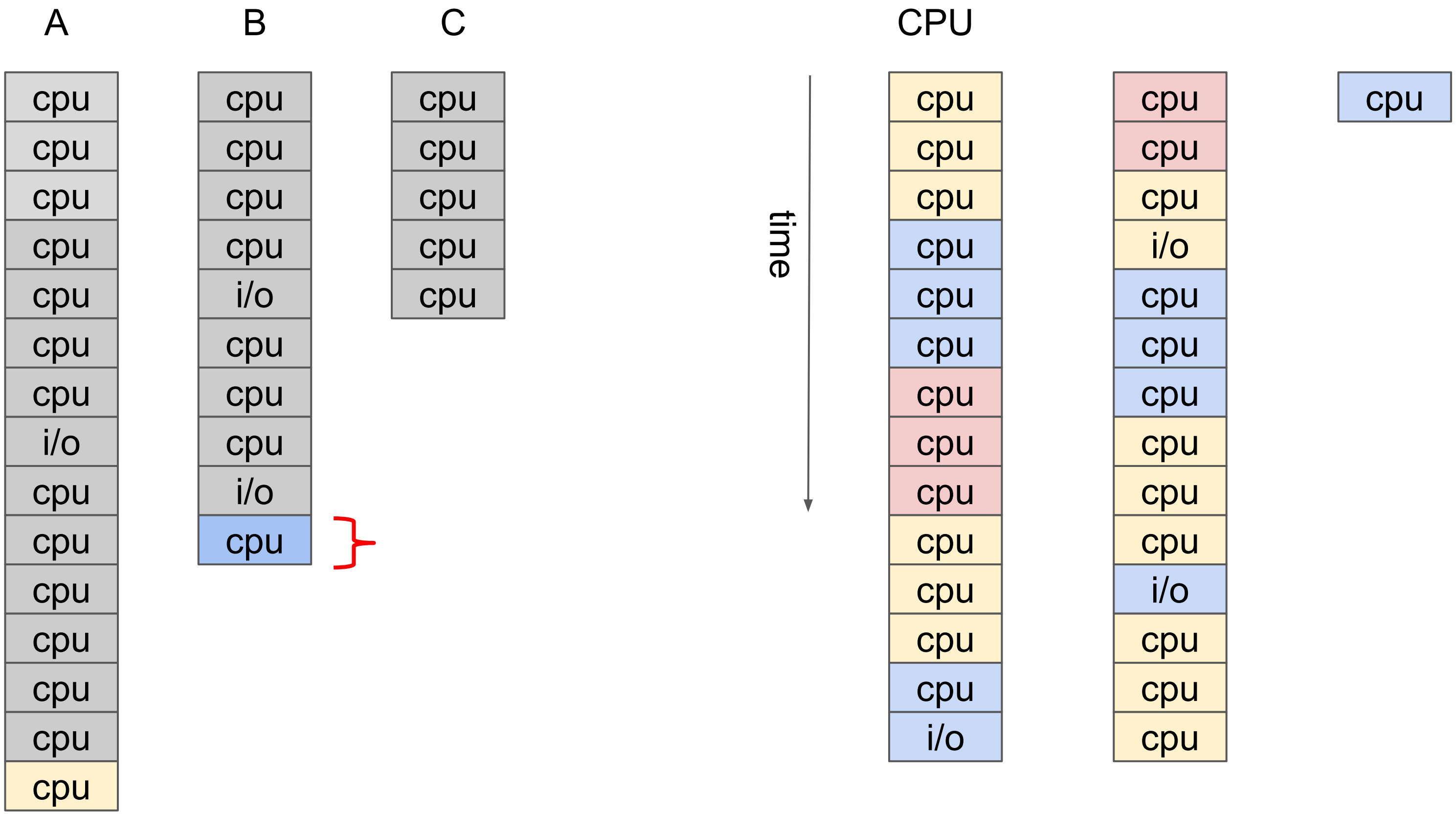
Exercise


A
B
C
CPU
time
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
i/o

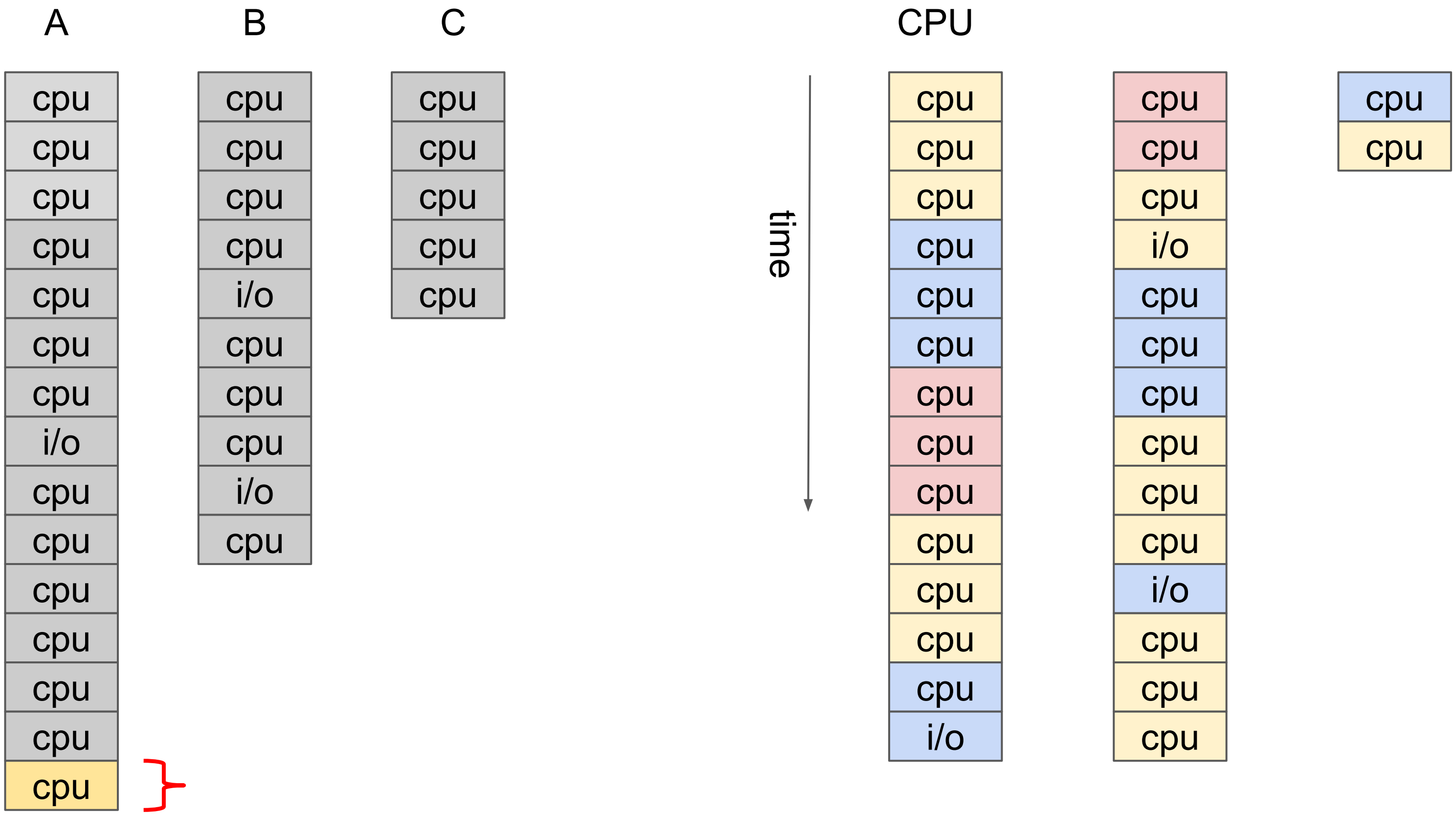
Exercise


A
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
cpu
B
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
C
cpu
cpu
cpu
cpu
cpu
time
CPU
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu
i/o
cpu
cpu
cpu
cpu
cpu
cpu
i/o
cpu
cpu
cpu

Exercise


A
B
C
CPU
time
cpu cpu cpu cpu cpu cpu cpu i/o cpu cpu cpu cpu cpu cpu cpu
cpu cpu cpu cpu i/o cpu cpu cpu i/o cpu
cpu cpu cpu cpu cpu
cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu i/o
cpu cpu cpu i/o cpu cpu cpu cpu cpu cpu i/o cpu cpu cpu
cpu

Exercise


A
cpu cpu cpu cpu cpu cpu cpu i/o cpu cpu cpu cpu cpu cpu cpu
B
cpu cpu cpu cpu i/o cpu cpu cpu i/o cpu
C
cpu cpu cpu cpu cpu
CPU
time
cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu cpu i/o
cpu cpu cpu i/o cpu cpu cpu cpu cpu cpu i/o cpu cpu cpu
cpu cpu

Context switching

- essential feature of any multitasking OS
- allows the illusion of sharing of a single CPU (or limited number of CPUs) among many processes
- we need context switching to implement multitasking when # CPUs < # processes

- OS maintains a context (state) for each process
  - usually part of PCB
- when OS switches between processes A and B:
  - OS first saves A's state in A's PCB
    - eg. save current CPU registers into PCB_A
  - OS then restores B's state from B's PCB
    - eg. load CPU registers from PCB_B

Context switching

- context switch occurs in kernel mode:
  - for example when process exceeds its **time slice**
    - enforcing time slice policy usually implemented via timer interrupt
  - or when current process voluntarily relinquishes (**yields**) CPU, eg. by sleeping
  - or when current process requests a blocking I/O operation, or any blocking system call
  - or due to other events, such as keyboard, mouse, network interrupts
- context switch introduces time overhead
  - CPU spends cycles on no "useful" work, eg. saving/restoring CPU registers
  - context switch routine is one of the most optimized parts of kernels
- context switch performance could be improved by hardware support:
  - eg. CPU supporting saving/restoring registers in a single instruction
  - or CPU could support multiple sets of registers
  - software based context switch is slower, but more customizable, and often more efficient

Questions?